Optimized and formally-verified compilation for a VLIW processor

Cyril Six

To cite this version:

Cyril Six. Optimized and formally-verified compilation for a VLIW processor. Performance [cs.PF]. Université Grenoble Alpes [2020-..], 2021. English. NNT : 2021GRALM025 . tel-03326923v2

HAL Id: tel-03326923
https://hal.science/tel-03326923v2

Submitted on 10 Dec 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Cyril SIX

Thèse dirigée par **David MONNIAUX**, Université Grenoble Alpes
et codirigée par **Sylvain BOULME**, Grenoble INP

préparée au sein du **Laboratoire VERIMAG**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Compilation optimisante et formellement prouvée pour un processeur VLIW

Optimized and formally verified compilation for a VLIW processor

Thèse soutenue publiquement le **13 juillet 2021**,
devant le jury composé de :

Monsieur DAVID MONNIAUX
DIRECTEUR DE RECHERCHE, CNRS DELEGATION ALPES, Directeur de thèse
Monsieur ADAM CHLIPALA
PROFESSEUR ASSOCIE, Massachusetts Institute of Technology, Rapporteur
Monsieur XAVIER LEROY
PROFESSEUR, COLLEGE DE FRANCE, Rapporteur
Madame LAURE GONNORD
MAITRE DE CONFERENCE HDR, UNIVERSITE LYON 1 - CLAUDE BERNARD, Examinatrice
Madame DELPHINE DEMANGE
MAITRE DE CONFERENCE, UNIVERSITE RENNES 1, Examinatrice
Monsieur ALAIN GIRAULT
DIRECTEUR DE RECHERCHE, INRIA CENTRE GRENOBLE-RHONE-ALPES, Président

Abstract

Software programs are used for many critical roles. A bug in those can have a devastating cost, possibly leading to the loss of human lives. Such bugs are usually found at the source level (which can be ruled out with source-level verification methods), but they can also be inserted by the compiler unknowingly. CompCert is the first commercially available optimizing compiler with a formal proof of correctness: compiled programs are proven to behave the same as their source programs. However, because of the challenges involved in proving compiler optimizations, CompCert only has a limited number of them. As such, CompCert usually generates low-performance code compared to classical compilers such as GCC. While this may not significantly impact out-of-order architectures such as x86, on in-order architectures, particularly on VLIW processors, the slowness is significant (code running half as fast as GCC -O2). On VLIW processors, the intra-level parallelism is explicit and specified in the assembly code through "bundles" of instructions: the compiler must bundlize instructions to achieve good performance.

In this thesis, we identify, investigate, implement and formally verify several classical optimizations missing in CompCert. We start by introducing a formal model for VLIW bundles executing on an interlocked core and generate those bundles through a postpass (after register allocation) scheduling. Then, we introduce a prepass (before register allocation) superblock scheduling, implementing static branch prediction and tail-duplication along the way. Finally, we further increase the performance of our generated code by implementing loop unrolling, loop rotation and loop peeling – the latter being used for Loop-Invariant Code Motion. These transformations are verified by translation validation, some of them with hash-consing to achieve reasonable compilation time.

We evaluate each introduced optimization on benchmarks, including Polybench and TACleBench, on the KV3 VLIW core, ARM Cortex A53, and RISC-V "Rocket" core. Thanks to this work, our version of CompCert is now only 16% slower (respectively 12% slower and 30% slower) than GCC -O2 on the KV3 (respectively ARM and RISC-V), instead of 50% (respectively 38% and 45%).

Keywords: Embedded, Optimization, Formal Verification, VLIW, Compilation

Résumé

On utilise des logiciels pour différents rôles, parfois critiques. La présence d'un bogue dans un logiciel peut avoir un coût dévastateur, pouvant aller jusqu'à la perte de vies humaines dans les cas les plus extrêmes. Les bogues viennent en général du code source (ceux-là peuvent être détectés par des méthodes de vérification formelle sur le langage source), mais ils peuvent être également générés par le compilateur. CompCert est le premier compilateur optimisant commercialisé comportant une preuve formelle de correction : il est formellement prouvé que les programmes compilés avec cet outil ont le même comportement que leurs programmes sources originels. Cependant, comme prouver les optimisations d'un compilateur relève d'un défi considérable, CompCert n'en a qu'un nombre limité. Ainsi, en général CompCert génère du code moins performant comparé à d'autres compilateurs classiques tels que GCC. Bien que cela n'impacte peu les architectures à exécution dans le désordre telles que x86, sur des architectures à exécution dans l'ordre, et en particulier sur des processeurs VLIW, le ralentissement est important (le code résultant est deux fois plus lent que GCC -O2). Sur un processeur VLIW, le parallélisme d'instructions est explicite et spécifié dans le code assembleur par des *bundles* (paquets) d'instructions : le compilateur doit paquétiser les instructions pour obtenir une bonne performance.

Le but de cette thèse est d'identifier, d'enquêter, d'implémenter et de vérifier formellement plusieurs optimisations classiques manquantes dans CompCert. Tout d'abord, j'introduis un model formel d'exécution de *bundles* VLIW sur processeur à pipeline imbriqué. Nous générons ces *bundles* à partir d'un réordonnancement en *postpass* (après allocation de registres). Ensuite, j'introduis un réordonnancement de superblocks en *prepass* (avant allocation de registres). Ce travail implique aussi l'introduction d'une prédiction statique de branchements, et une duplication de queue. Enfin, pour continuer d'augmenter la performance du code généré, j'introduis du déroulage, rotation, et pelage de boucles – ce dernier étant utilisé dans du déplacement d'invariants de boucles. Ces transformations sont vérifiées par validation de translation, et certaines utilisent du partage maximal afin d'obtenir des temps de compilation raisonnables.

Nous avons évalué chacune de ces optimisations sur des tests de performance, dont Polybench et TACLeBench, sur le coeur VLIW KV3, l'ARM Cortex A53, et le coeur RISC-V "Rocket". Grâce à nos travaux, notre version de CompCert est maintenant seulement 16% plus lente (respectivement 12% et 30% plus lente) que GCC -O2 sur KV3 (respectivement ARM et RISC-V), au lieu de 50% (respectivement 38% et 45%).

Mots-clefs : Embarqué, Optimisations, Vérification Formelle, VLIW, Compilation

Acknowledgements

I deeply thank Adam Chlipala and Xavier Leroy for their detailed and insightful reviews. I also thank Laure Gonnord, Delphine Demange and Alain Girault for taking part in this thesis jury. It is an honor for me.

I started this thesis without any prior notion of formal proofs or functional programming and barely any practical experience in compilers. This adventure was quite interesting (but very rewarding), to say the least. I was glad to have Sylvain Boulmé by my side for helping me navigate through the jungle of formal proof engineering in Coq and formal concepts in general. I have learned a lot from him and his teachings; his deep knowledge of Coq and proof methods has been invaluable. Complementary to Sylvain, David Monniaux was also very helpful both for general thesis direction and for having excellent ideas that, more often than not, led to promising results. Last but not least, I want to thank Benoît Dupont de Dinechin for ensuring I have an excellent environment at Kalray for working on my thesis and for his helpful insights on optimizing compilers.

I want to thank the persons of the VERIMAG laboratory for providing me with a great work environment. I also want to thank the doctoral school EDMSTII for having been very understanding and amenable throughout the thesis. In particular, they participated in the opportunity for me to attend the DeepSpec Summer School of 2018 (DSSS'18) at the Princeton University. Thanks to their most excellent program and teachers, my knowledge level of Coq and formal proofs got boosted through the roof.

I want to thank two additional persons: Florent-Bouchez Tichadou and Fabrice Rastello. I believe that most of life is the result of lucky opportunities; these two persons introduced me to the CORSE research team and gave me the will to pave my own way into optimizing compilers studies. I sincerely thank both of them. I also want to thank my wonderful teacher Christophe Brouillard, who showed me the wonderful world of mathematics and gave me the first glimpse of what research looks like. Finally, I thank Duco Van Amstel for having such a pleasant and memorable discussion, after which I was sure to head towards doctoral studies!

I thank my lab and company student mates Amaury, Arthur, Bai, Clément, Maëva, Matheus, Rémy, Valentin, Vincent, Yanis. Many meals, funny discussions, and horoscopes were shared. I also thank my former lab (CORSE) student mates for all the interesting discussions about compilation, performance, or any other subject. Among them, I'm thinking of Antoine, Diogo, Emmanuelle, Fabian, Imma, Luis, Nicolas, Thomas; but there are others!

Three years (and a half) is a lot of time, during which I met many people who inspired me on a personal level and have encouraged me throughout this adventure. I want to thank the Rainbow Swingers chorus, particularly Andrea, Antoine (x2), Benjamin, Cécile, Charlotte,

Chloé, Clémentine, Cyrielle, Geoffroy, Juliette, Kevin, Marion, Maxime, Maximin, Raphaël, Solène, Sophia, Véronique (and many others!) with who I had such a wonderful time. Lastly, I want to thank my other wonderful friends who have inspired me on many occasions: Alex, Aurélien, Bastien, Clémentine, Corentin, Freddy, Guillaume, Hannah, Inès, Jean, Joseph, Julien, Laurie, Lucie, Marine, Martin, Maxime, Nicolas, Olga, Pierre, Virginie.

Lastly, I want to thank the three persons that are the most important to me.
My grandmother has always adored me and has been ever-loving since I was born.
My father taught me to always keep working for my goals and never give up.
My mother encouraged me to follow my dreams, whatever happens.
These persons have always been there to support me throughout my whole life.
Thank you immensely for being there and believing in me!

Remerciements

Je remercie profondément Adam Chlipala et Xavier Leroy qui ont pris le temps de lire cette thèse en tant que rapporteurs : leurs remarques détaillées ont été extrêmement utiles. Je remercie également Laure Gonnord, Delphine Demange et Alain Girault pour faire parti de ce jury. C'est un immense honneur pour moi.

Le démarrage de cette thèse fut une épopée. Je n'avais pratiquement aucune connaissance en matière de preuve formelle, programmation fonctionnelle, ou même de l'expérience pratique en compilation. Cette thèse a été une aventure fructueuse et formatrice, notamment grâce à mon encadrant Sylvain Boulmé. Sylvain était mon guide dans la jungle des preuves formelles en Coq et des formalismes en général. J'ai énormément appris de lui, autant de par sa personne que de par ses enseignements; sa connaissance de Coq et des méthodes formelles a été un atout indispensable pour ma thèse. Mon directeur de thèse, David Monniaux, a également été formidable, autant pour la direction de la thèse que pour ses idées qui ont quasiment toujours mené à des résultats prometteurs. Enfin, je remercie Benoît Dupont de Dinechin pour m'avoir instauré un cadre de travail idyllique à Kalray, en plus d'avoir été une référence en matière de compilation optimisante.

J'ai passé trois super années de thèse au sein de VERIMAG, et je remercie toutes les personnes de ce laboratoire pour cela. Je remercie également l'école doctorale EDMSTII : ça a toujours été un plaisir d'avoir affaire à eux sur le côté administratif, et ils m'ont donné l'opportunité de participer à des écoles d'été, en particulier l'école d'été DeepSpec Summer School 2018 (DSSS'18) à l'université de Princeton. Les cours exemplaires de DSSS'18 m'ont permis de relever de façon conséquente mon niveau de connaissances en preuves formelles, ce qui s'est révélé être un atout considérable pour ma thèse.

Je tiens également à remercier Florent-Bouchez Tichadou et Fabrice Rastello. Si je devais résumer ma vie, je dirais que c'est tout d'abord des rencontres — ces deux personnes en particulier m'ont introduit au sujet de la recherche en compilation, et m'ont donné l'envie de poursuivre dans cette voie ! Je tiens également à remercier mon professeur de prépa Christophe Brouillard; il m'a montré la beauté des mathématiques, et m'a permis d'avoir un premier aperçu de la recherche. Enfin, je remercie Duco Van Amstel pour une discussion autour d'un verre qui a scellé mon envie de poursuivre en doctorat.

Je remercie mes camarades du laboratoire ou de Kalray : Amaury, Arthur, Bai, Clément, Maëva, Matheus, Rémy, Valentin, Vincent, Yanis. Avec eux j'ai passé beaucoup de repas partagés, de discussions intéressantes, et même d'horoscopes amusants. Je remercie aussi les collègues de mon ancienne équipe (CORSE) pour les discussions orientés performance, compilation, ou juste

tout autre sujet. Je pense à Antoine, Diogo, Emmanuelle, Fabian, Imma, Luis, Nicolas, Thomas; mais il y en a d'autres !

Trois ans (et un petit demi) c'est une sacré durée. J'ai eu l'occasion de rencontrer beaucoup d'autres personnes qui m'ont encouragé tout au long de ma thèse, et qui ont été source d'inspiration sur un plan personnel. Je remercie les personnes de la chorale des Rainbow Swingers, en particulier Andrea, Antoine (x2), Benjamin, Cécile, Charlotte, Chloé, Clémentine, Cyrielle, Geoffroy, Juliette, Kevin, Marion, Maxime, Maximin, Raphaël, Solène, Sophia, Véronique (et beaucoup d'autres !) avec qui j'ai passé des moments inoubliables. Je remercie également tous mes autres amis proches et adorables : Alex, Aurélien, Bastien, Clémentine, Corentin, Freddy, Guillaume, Hannah, Inès, Jean, Joseph, Julien, Laurie, Lucie, Marine, Martin, Maxime, Nicolas, Olga, Pierre, Virginie. Cela a été un réel plaisir de partager du temps avec vous tout au long de ma thèse !

Enfin, je veux remercier trois personnes qui ont été là tout au long de ma vie, et ont toujours cru en moi.
Mamie, tu as toujours été très aimante et une grande confidente depuis que je suis né.
Papa, tu m'as appris à ne jamais abandonner et à toujours viser plus haut.
Maman, tu as toujours été là pour m'encourager dans ma vie peu importe les situations.
Vous avez toujours été là pour me soutenir dans ma vie, et je suis fier de vous avoir comme parents et grand-parent.

Contents

Contents ix

I Background 1

1 Introduction 3
1.1 General Motivation . . . 3
1.1.1 Software Verification . . . 3
1.1.2 Compilation . . . 4
1.1.3 The Need for Formally Verified Compilation . . . 5
1.1.4 Verifying Software with Coq . . . 6
1.1.5 VLIW Processors in the Context of Critical Applications . . . 7
1.2 The Kalray KV3 Architecture . . . 9
1.3 The CompCert Formally Verified Compiler as of Today . . . 11
1.3.1 Overview of CompCert . . . 11
1.3.2 CompCert's Framework for Proving Transformation Passes Correct . . . 13
1.4 Motivations of this Thesis and Contributions . . . 15

2 Related Work 19
2.1 Compiler Optimizations for VLIW Processors . . . 19
2.1.1 Instruction Scheduling . . . 19
2.1.2 Loop Transformations . . . 20
2.1.3 Other Optimizations . . . 21
2.2 Formally Proven Compiler Transformations . . . 22
2.2.1 Advanced Compiler Optimizations in CompCert . . . 22
2.2.2 Other Related Work . . . 24

II Formally Verified Optimizations 25

3 Formal Verification of Basic-Block Scheduling at the Asm Level 27
3.1 AsmVLIW: Semantics for a VLIW Assembly Language . . . 30

3.1.1 Overview of CompCert's Asm IR 30
3.1.2 Syntax of Bundles & Basic Blocks 31
3.1.3 Parallel Semantics of AsmVLIW 32
3.1.4 Sequential Semantics in Asmblock 34
3.2 Retrieving the Basic-Block Structure with Machblock 35
3.2.1 The Mach IR of CompCert 35
3.2.2 Necessity of Constructing Basic Blocks at the Mach Level 37
3.2.3 Machblock: Generating Basic Blocks from Mach 37
3.3 Asmblock: Adapting the Mach-to-Asm Translation for Basic Blocks 39
3.3.1 Translating and Proving the Translation of Mach Code to Asm 39
3.3.2 Translating Machblock to Asmblock 41
3.3.3 Proof of Forward Simulation 47
3.4 Formal Verification of the Basic-Block Postpass Scheduler 52
3.4.1 AbstractBasicBlock IR 53
3.4.2 Parallelizability Checker 54
3.4.3 Verifying Intrablock Reordering 55
3.4.4 Generic and Verified Hash-Consing 57
3.4.5 Peephole Optimization 58
3.4.6 Atomic Sequences of Assignments in AbstractBasicBlock 59

4 General Concepts of Superblock Scheduling **61**
4.1 Guiding Trace Selection with Branch Prediction 62
4.1.1 What we Expect out of Static Branch Prediction 63
4.1.2 Storing the Prediction 65
4.1.3 Acquiring Prediction Information 66
4.1.4 Driving the Selection Based on Predictions 68
4.2 Superblock Transformations before Scheduling 70
4.2.1 Tail Duplication 73
4.2.2 Loop Unrolling 73
4.3 Superblock Scheduling 75
4.3.1 Elements of Superblock Scheduling Correctness 77
4.3.2 Choosing an Appropriate Scheduling Oracle 79
4.3.3 Application to our Example 80
4.4 Superblock Transformations and RTL 81
4.4.1 Existing RTL Transformations in regard to Superblock Transformations 82
4.4.2 Order of Transformations 84

5 Introducing Superblock Execution Semantics with RTLpath **85**
5.1 Modifying CompCert's RTL Representation 85

5.1.1 Overview of RTL . . . 86
5.1.2 Modeling Speculative Loads in RTL . . . 89
5.1.3 Modeling Branch Prediction . . . 91
5.2 Representing Superblocks as Execution Paths in RTLpath . . . 92
5.2.1 RTLpath Syntax and Informal Semantics . . . 93
5.2.2 RTLpath Formal Semantics . . . 96
5.2.3 Bisimulation between RTLpath and RTL Semantics . . . 98
5.3 Generating RTLpath Paths from RTL . . . 101
5.3.1 RTLpath Generation Oracle . . . 103
5.3.2 Pathmap Verifiers . . . 107
5.3.3 RTLpath State Equality Modulo Liveness . . . 109

6 Formal Verification of Code-Duplication Transformations 111
6.1 Proof of Correction through a Forward Simulation . . . 112
6.2 Verifier Implementation . . . 114
6.3 Module System . . . 114
6.4 Modifications to Support Branch Swapping . . . 115

7 Formal Verification of Superblock Scheduling 117
7.1 Abstract Symbolic Execution for Superblocks . . . 119
7.1.1 Abstract Symbolic States and Execution Semantics . . . 121
7.1.2 Superblock Symbolic Execution . . . 129
7.1.3 Bisimulation between RTLpath and Symbolic-Execution Semantics . . . 130
7.2 Proving Lockstep Simulation of Scheduling from Sufficient Conditions on Abstract Symbolic Executions . . . 134
7.2.1 Simulation of Superblocks . . . 134
7.2.2 Simulation Test Specification and Superblock Scheduling Lockstep Simulation . . . 140
7.3 Refined Simulation Test with Hash-Consing . . . 143
7.3.1 Refined Symbolic States and Refinement Relations . . . 144
7.3.2 Simulation Test Specifications and Implementation . . . 150
7.3.3 Canonization of Refined Symbolic Values . . . 155

III Experiments 159

8 Performance Tuning of Superblock Formation, Selection and Linearization 161
8.1 Trace Selection and Tail-Duplication . . . 161
8.1.1 Trace Partitioning . . . 163
8.1.2 Tail-Duplication . . . 165

8.2 Loop Unrolling, Peeling and Rotation . . . 168
8.2.1 Innermost-Loop Detection . . . 170
8.2.2 Loop Unrolling . . . 171
8.2.3 Loop Rotation . . . 172
8.3 Adapting the `Linearize` Oracle for Superblocks . . . 173
8.3.1 Original `Linearize` Heuristics from CompCert . . . 175
8.3.2 Modifications to the Heuristics for Superblocks . . . 178
8.4 Static Branch Prediction . . . 179
8.4.1 Implementing Branch Prediction for Free . . . 179
8.4.2 A new Heuristic for Predicting Branches Inside Loops . . . 180

9 Implementing the Scheduling Oracles **183**
9.1 Postpass Scheduling Oracle . . . 184
9.1.1 Postpass Frontend . . . 185
9.1.2 Postpass Backend . . . 188
9.2 Prepass Scheduling Oracle . . . 189
9.2.1 Additional Dependencies for Conditional Branches . . . 190
9.2.2 Prepass Scheduling Algorithms . . . 191

10 Experiments **195**
10.1 Experimental Setup . . . 195
10.1.1 Measured Processors . . . 196
10.1.2 Measurement Methods . . . 197
10.1.3 Benchmarks Used . . . 197
10.2 Impact of Postpass Scheduling . . . 199
10.2.1 Performance . . . 199
10.2.2 Compilation Times of Postpass Scheduling on KV3 . . . 201
10.3 Impact of Superblock-Scheduling-Related Optimizations . . . 202
10.3.1 Evaluation of our Static Branch Prediction . . . 202
10.3.2 Evaluation of our Superblock Linearizer . . . 204
10.3.3 Evaluation of Loop Unrolling, Loop Rotation and Tail-Duplication Thresholds . . . 205
10.3.4 Performance Impact of each Added Optimization . . . 208
10.3.5 Evaluation of the Different Prepass-Scheduling Strategies . . . 210
10.3.6 Gains of Loop Unrolling without Prepass Scheduling . . . 210
10.4 CompCert vs GCC in our Current Iteration . . . 212
10.4.1 CompCert vs GCC on the KV3 Architecture . . . 213
10.4.2 CompCert vs GCC on the AArch64 and RISC-V Architectures . . . 217

11 Conclusion **219**

11.1 Summary 219
11.2 Reflexions and Future Work 220

A Non-exhaustive Instruction Listing of the KV3 Core **A1**

Bibliography

Part I

Background

1 Introduction

1.1 General Motivation

1.1.1 Software Verification

In the past decades, our reliance on software to conduct repetitive and sensitive tasks has grown tremendously. Whether it be to fly a space rocket, manage bank deposits, or just ordering your next meal, software programs are now almost everywhere.

Due to their human manufacturing, software programs are prone to errors. A wrong design, a miscommunication within the development team, or just a faulty line inserted on a Friday afternoon: it suffices to introduce a tiny error, a tiny misconception, to make a software program crash – or worse, behave erratically. These errors (called "bugs"[1]) may have varying effects based on the nature of the application.

If there is a bug on your meal ordering application, you could probably have the wrong meal ordered or have to order it a second time because the application crashed. A bug in your message delivery service and your message might not get sent at all. These represent classes of applications that are non-critical: a bug in these might make people lose time, or in general, have minor inconveniences.

Some other applications can have more nasty repercussions if there is a bug in them. In 2013, software defects were found in the United Kingdom's Post Office computer system: people were wrongly accused of stealing money, some of them even going to jail as a result, even though the actual cause was false accounting within the software [68].

Software bugs in aerospace applications can have dire consequences. In 1996, the Ariane 5 space rocket physically exploded on its maiden flight because of miscasting a 64-bit floating-point variable into a 16-bit integer, causing the propulsion system to flip the rocket [59]. In 2009, four unfortunate passengers of an ES350 Toyota car died after their car's accelerator pedal got stuck and the brakes did not apply enough pressure to stop the car. Several investigations were led in the early 2010s to investigate this incident among others; software bugs were found to be the

[1] The term “bug” became popular after an unfortunate moth caused a hardware error on the Mark II computer, though it has been part of the engineering jargon since the 1870s [60].

likely cause of this incident [45]. In 2021, a software bug from Orange made all urgency calls unreachable for several hours in France, resulting in at least five deaths [35].

The prevention and detection of software bugs have been an ongoing area of research for the past decades. Ensuring that critical software is bug-free (or at least free of any bug that could cause harm) may involve various techniques. Some involve executing the program to look for faulty behaviors, either through extensive testing (Parnas et al. [70]; Parnas et al. [69]) or runtime verification (Leucker and Schallhart [53]; Bartocci et al. [7]). Others rely on deriving properties based on the source code itself: this can be done through static analysis (Blanchet et al. [12]) or formal verification (Yoo et al. [101]).

1.1.2 Compilation

The execution of software programs on processors requires them to be encoded in binary machine code. Traditionally, this machine code is obtained by translating a textual representation called *assembly language*. Each separate code module, written in assembly, is translated into *object machine code* through an *assembler*. Then, a *linker* is used for gathering all objects code into one binary machine code, ready to be executed on a given machine.

One caveat of assembly language is that it is too close to the machine: it does not provide enough abstraction for developing software. As such, producing software directly in assembly language is inefficient and error-prone.

To leverage this issue, high-level languages were invented to ease software development. Among them, the C language became one of the most popular for embedded systems primarily because of its portability (the same C code can be compiled for many different processor architectures) and its closeness to actual machine code, making it possible to fine-tune C code to get similar performance as with concrete machine code.

The process of translating C code to machine code is called *compilation* and is done through a *compiler*. The compilation process is usually split into several passes: a *frontend* that transforms C code into a given *intermediate representation (IR)* of the compiler, and finally a *backend* that gradually transforms this IR into an assembly code suitable for the target machine. Once this is done, the *assembler* and *linker* can then generate the final machine code.

During the compilation process, the compiler can optimize the code: performing transformations that aim to produce faster machine code, smaller machine code, or sometimes both.

1.1.3 The Need for Formally Verified Compilation

The compiler, assembler and linker are all pieces of software: there may be bugs in them. In particular, compilers are considered one of the most complex pieces of software; they usually involve intricate transformation passes, especially when optimizing the code.

As such, properties that were proved at the source level might not be valid anymore if the compiler introduced a bug. Compilers are usually thoroughly tested to trim out any major bug. However, it is not physically possible to test all the possible cases because there are infinitely many of them: even with thorough testing, there might still be bugs hidden in corners.

While this is of little to no concern for most applications, it is concerning for critical applications. The probability of having a major accident happen one day because of a compiler bug is not null.

To tackle this potential issue, some classes of applications require traceability between the object code and the source code (for example, the DO-178B requirements for avionics [42]). Such traceability is generally challenging to get from compilers without relying on manual annotations [79]. It is also compromised by many compiler optimizations, though some recent research attempts to tackle this issue (Li et al. [54]). In some critical systems, traceability is ensured by turning off all optimizations of compilers [8].

Turning off all optimizations comes with a downside: the resulting code is usually bigger and takes more time to execute. This, in turn, increases the worst-case execution time (WCET) and the energy consumption (more cycles are spent for performing the same task).

One way to tackle these issues is to use a formally verified compiler.

The CompCert certified compiler (Leroy [50, 51]) is the first optimizing C compiler with a formal proof of correctness that is used in industry [8; 43]. In particular, it does not have the middle-end bugs usually found in compilers [100], thus making it a significant success story of software verification.

CompCert features several middle-end optimizations (constant propagation, inlining, common subexpression elimination, etc.) as well as some backend optimizations (register allocation using live ranges, clever instruction selection on some platforms).

Using CompCert allows to have both compiler optimizations and formal guarantees: it proves a property of semantic preservation between the source code and the generated assembly code. In other words, the proofs done on the C source code remain valid on the assembly code generated by CompCert.[2]

[2] Small note: CompCert's proof of semantic preservation only holds for executions free of undefined behaviours.

CompCert is used in several projects, among which:

- Vélus [18; 17], a formally verified compiler for the Lustre dataflow language [39]. It consists of a Lustre frontend branched into the Clight IR of CompCert. Our version of CompCert is compatible with a modified version of Vélus [80].
- CertiKOS: an architecture for building formally verified concurrent OS kernels [38]. They use CompCertX, a thread-safe version of CompCert [37].
- The Verified Software Toolchain [2], comprising tools to prove properties about the program's behavior at source level using a dialect of C (Verifiable C [3]). The program is then plugged into CompCert via the Clight intermediate representation.

1.1.4 Verifying Software with Coq

Using Coq is one possible technique to verify software [10]. Coq is both a (functional) programming language and a proof description language. The typical workflow in Coq is to write code in Gallina (Coq's programming language) and then specify theorems about that code. Coq's code is then extracted to other languages, typically OCaml or Haskell.

Given a list sorting algorithm `sort_list` written in Gallina, an example of a theorem could be "For each possible input list `l`, `(sort_list l)` is sorted".

To prove such theorems, one writes a list of tactics to apply, such as decomposing the theorem into different cases, introducing variables from `forall` quantifiers, or using the result of another (previously proved) theorem.

The trusted computing base (TCB) of Coq is intentionally small: most of its datatypes are provided in the form of formally proven libraries instead of built-in types. For example, Coq does not natively support integers or even booleans: instead, several integer libraries exist, each supplying its definition of what integers (or booleans) are, the possible operations on them and their associated lemmas (such as the associativity of addition).

Coq proofs give a high level of assurance; if there is a bug in code verified in Coq, it may come from either:

- The Coq extractor (the part of Coq that converts a Gallina program into OCaml (or Haskell), which can then be compiled); though recent work has been done towards a fully verified Coq extractor, such as Savary Bélanger et al. [84] or Forster and Kunze [34].
- The compiler of the language used after extraction (e.g. OCaml's compiler).
- A problem in the specification written in Coq: under-specification, wrong modeling, or axiomatizing wrong hypotheses.

Most of the time, bugs of programs written in Coq come from the latter.

A disadvantage of Coq is that some proofs can be very tedious to perform (it usually takes much more time to prove code than write it). Proving code in Coq usually involves cutting each piece of code finely into smaller fragments, each of which can then be proven more easily, and then patching all these fragments together into one bigger theorem. When the code can hardly be cut in such fragments, one must rely on finding proof invariants or other techniques. In particular, most of the time, the Coq proof must be changed when the Coq code is modified; though some automation tactics exist to remedy that, an overly automated proof can be very tricky to untangle when it becomes invalid because of a change.

One way to avoid this caveat for some challenging problems, as has initially been explored by Tristan and Leroy [95], is to apply *a posteriori* verification: an untrusted oracle (e.g. written in OCaml) returns its result to a formally proven verifier written in Coq. The verifier then checks the oracle's result for correctness: if the verdict is positive, we have the formal proof that the result is correct.

With this approach, it is also much easier to modify the program transformation (performed by the oracle). Indeed, if it stays within a certain class of program transformations, and if the verifier is complete in regards to that class, then modifying the oracle does not require to modify the verifier or its proof. This is a major advantage, especially for complex optimizations that might require slight changes before reaching their full potential.

The other advantage is that it opens up the range of possible proofs: some problems that would be very hard to prove structurally may become much easier to prove through an a posteriori verifier.

In this thesis, we used a posteriori verification profusely: this allowed us to fine-tune our compiler optimizations without much need for proof modification.

1.1.5 VLIW Processors in the Context of Critical Applications

In most architectures, processors are synchronous entities that execute instructions to the beat of a clock. Executing an instruction involves a several-step process: fetching the instruction from memory, decoding the instruction, reading the registers indicated by the instruction, performing the instruction's operation, and finally writing back the result to the destination registers. As such, each instruction typically take several cycles to be completed.

Modern processors are organized so that several instructions are evaluated in parallel. Each step is assigned to a processor unit: while a given instruction is decoded, another instruction is getting fetched, and yet another instruction is being evaluated. This process is called *instruction*

pipelining, and the hardware responsible for that is the *processor pipeline*. Most processors have *interlocked* pipelines: if an operand of an instruction to be executed is not yet ready, the processor inserts *stalls* in the pipeline, which halts the execution of that instruction until its data is ready.

Thanks to this method, as long as there is no starvation in the pipeline and no stall is inserted, processors can virtually execute one instruction per cycle. The process of feeding an instruction through the pipeline is called *issuing* an instruction.

In addition to instruction pipelining and the ever-going quest to get more performance, various techniques have emerged to have processors execute code faster. A first technique has historically been to increase the processors' frequencies; in just decades, the processor frequencies' order of magnitude increased from the MHz all the way to the GHz. However, this technological progress soon hit a wall, as increasing frequency further started to pose challenging issues in terms of miniaturization and temperature increase within the chips.

Another technique has been to issue several instructions per cycle instead of just one, either by allowing the pipeline to process two or more instructions per step or having different distinct pipelines.

In the most extreme case, some processors, including all current x86 architecture processors, can execute instructions ahead of time to fill idle units: these are categorized as *out-of-order* processors since the concrete order of execution might not be the same as the original order of the assembly code. Having such processors allows to have good performance without having to change the assembly code: they minimize the number of cases where instructions have to wait on each other in the pipeline.

However, this comes with a cost: their design is more complex (their hardware circuit requires more space), they are more susceptible to side-channel information leaks (especially when speculative execution is used to decrease the cost of control branches), and they tend to be less energy-efficient.

In contrast, *in-order* processors do not try to reorder instructions. Their design is then simpler, more energy-efficient, and they are also more predictable, which makes them a good fit for embedded systems. Indeed, reasonably tight bounds on worst-case execution time (in most safety-critical applications, the WCET must be estimated by a sound analysis [9]) are more easily obtained on in-order processors than on out-of-order processors. In addition, a simpler design may be more reliable.[3]

In-order superscalar processors have hardware logic to detect code dependency at runtime and issue multiple independent instructions in the same cycle. VLIW (Very Large Instruction Word)

[3]For instance, Intel's Skylake processor had a bug that crashed programs under complex conditions [52].

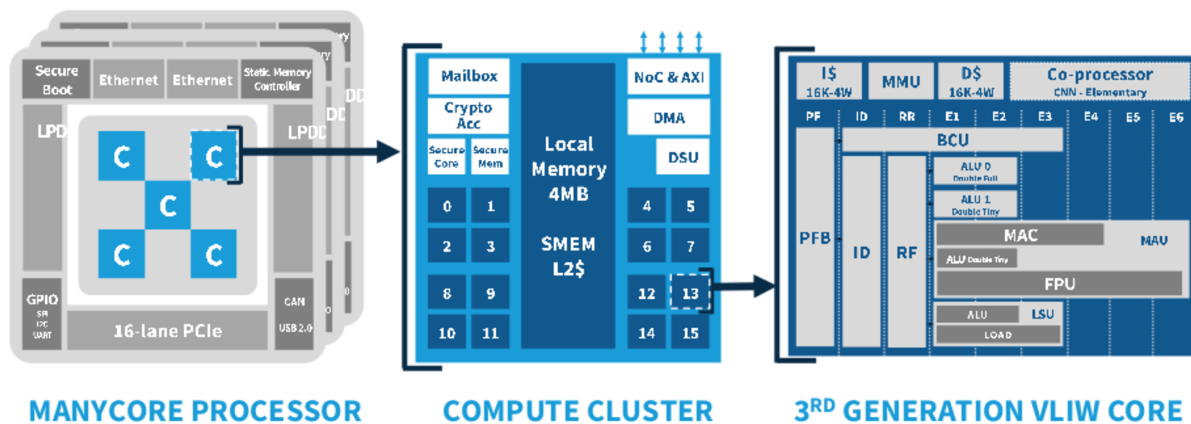


Figure 1.1.: The Coolidge architecture

processors can also issue several instructions in parallel [32], but unlike superscalar processors, the groups of instructions to be issued together (called *bundles*) are expressed directly in the assembly code: each bundle is syntactically separated by a `;;` token.

The caveat of these processors is that since they do not have any reordering at runtime, they rely on compilers' optimizations to produce adequate bundles and, more generally, to ensure good performance. In this thesis, we will mostly tackle two levels of performance issues for VLIW processors:

- The instruction ordering: a bad ordering could lead to either the processor spending most of its cycles stalling or having certain units be idle while they could be executing an instruction. Modern out-of-order architectures do not suffer from that, thanks to their runtime reordering.
- The cost of branching: on such processors, taking a branch induces a penalty because the pipeline has to be refilled. This penalty is bigger for a conditional branch than for an unconditional branch. Again, modern out-of-order architectures suffer less from this, thanks to their speculative branch execution.

1.2 The Kalray KV3 Architecture

The Kalray Coolidge is a manycore processor, as displayed in figure 1.1. It features 5 compute clusters interconnected by a Network-on-Chip. Each compute cluster runs its own code: to deploy an application to the Coolidge architecture, one uses the Kalray compilation tools to create a multi-binary file, which can then be deployed to the processor. External libraries handle the matters of spawning and communication between the clusters; the compiler does not have to worry about it: as such, in this thesis, we focus on programs running on a single cluster.

Each compute cluster contains 16 KV3 VLIW cores, sharing a local memory of 4 MB. Support for parallelism is provided either through libraries made available in a C API or through higher-level languages such as OpenCL combined with specific compiler support [90]. In our case, CompCert has no support for anything related to multi-threading, so we focus here on the execution on one single core (of a single compute cluster).

The Kalray KV3 core implements a 6-issue Fisher-style VLIW architecture [33] (partial predication, dismissible loads, no rotating registers), with an instruction pipeline of 8 stages. It has 64 general-purpose registers of 64 bits used for both integer and floating-point data (a given register can store an integer or a float). It sequentially executes blocks of instructions called *bundles*, with parallel execution within them.

Bundles A *bundle* is a block of instructions that are to be issued into the pipeline at the same cycle. They execute in parallel with the following semantics. If an instruction writes into a register that is read by another instruction of the same bundle, then the value that is read is the value of the register before executing the bundle. If two instructions of the same bundle write to the same register, then the behavior at runtime is non-deterministic. For example, the bundle written in pseudo-code "$R_1 := 1; R_1 := 2$" assigns R_1 non-deterministically. On the contrary, "$R_1 := R_2; R_2 := R_1$" is deterministic and swaps the contents of the R_1 and R_2 registers in one atomic execution step. In assembly code, bundles are delimited by `;;` (Fig. 1.2). Compilers must ensure that each bundle does not require more resources than available—e.g., the KV3 has only one load/store unit, thus a bundle should contain at most one load/store instruction. The assembler refuses ill-formed bundles.

Execution Pipeline In the case of the KVX, bundles are executed through an 8-stage interlocked pipeline: the first stage prefetches the next bundle (PF stage), the second decodes it (ID stage), the third reads the registers (RR stage), then the last five stages (E1 through E5) perform the actual computation and write to the destination registers; depending on the instructions, the writes occur sooner or later (e.g., an addition takes fewer stages than a multiplication). If, during the RR stage[4], one of the read registers of an instruction in the bundle is not available, the pipeline *stalls*: the bundle stops advancing through the pipeline until the register gets its result (Fig. 1.2).[5]

[4] Or the ID stage, for some instructions such as conditional branching.
[5] When a register is read before some prior instruction has written to it, *non-interlocked* VLIW processors use the old value. The compiler must then take instruction latencies and pipeline details into account to generate correct code, including across basic blocks. This is not the case for the KVX, where these aspects are just matters of code efficiency, not correctness.

Cycle	ID	RR	E1	E2	E3	E4+E5
1	B1					
2	B2	B1				
3	B3	B2	B1			
4	B4	B3	B2	B1		
5	B4	B3	STALL	B2	B1	
6	B4	B3	STALL	STALL	B2	
7	B5	B4	B3	STALL	STALL	

$B1 : R_1 := load(R_0 + 0);;$
$B2 : R_2 := load(R_0 + 4);;$
$B3 : R_3 := R_1 + R_2;\ R_4 := R_1 * R_2;;$
$B4 : R_3 := R_3 + R_4;;$
$B5 : store(R_0, R_3);;$
$B6 : R_6 := R_7 + R_8;;$

Figure 1.2.: The pipeline stalls at cycles 5 and 6 because B3 is waiting for the results of R_1 and R_2 from bundles B1 and B2, which are completed at stage E3. Stage PF (not shown here) happens just before the ID stage.

1.3 The CompCert Formally Verified Compiler as of Today

1.3.1 Overview of CompCert

Usual compilers (GCC, Clang/LLVM, ICC) split the compilation process into several components. In the case of CompCert, a *frontend* first parses the source code into an *intermediate representation* (IR)—called Cminor—that is independent of the target machine [13]. Then, a *backend* transforms the Cminor program into an assembly program for the target machine [51]. Each of these components introduces several IRs, which are linked by *compilation passes*. A compilation pass can either transform a program from an IR to another (transformation pass) or optimize within an IR (optimization pass). As illustrated in Fig. 1.3, CompCert introduces more IRs than usual compilers. This makes its whole proof more modular and manageable because each compilation pass comes with its own proof of *semantic preservation*.

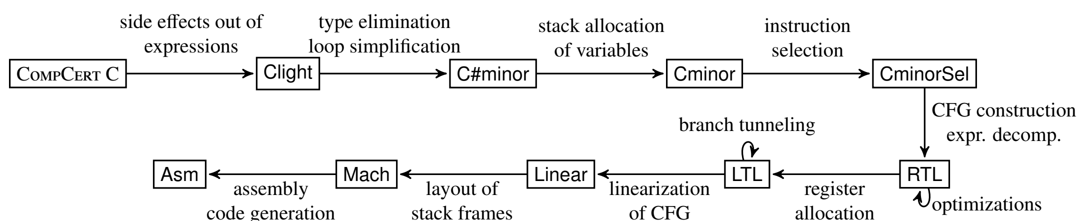


Figure 1.3.: The Intermediate Languages of CompCert

The CompCert compiler [41] executes an external C preprocessor then parses the result. It accepts a large subset of the C99 language [1], [6] as well as some C11 extensions.

[6]The main C99 features absent from CompCert C are variable-length arrays and certain forms of unstructured “switch” statements. On most platforms, the `setjmp()/longjmp()` system is not supported either.

In particular:

- The *backend* starts at the CminorSel level: we have at this level a precise abstract syntax tree of the C program.
- From CminorSel to RTL is the construction of the Control-Flow Graph, using an infinite supply of *pseudo-registers*. Most of the optimizations are done on the RTL level, such as constant propagation, common subexpression elimination, tail call elimination. Compared to more traditional compilers whose control-flow graph is organized in basic blocks, in RTL, each node of the control-flow graph is a single 3-address instruction. RTL will be detailed in chapter 5.
- The *register allocation* pass transforms an RTL program (infinite supply of registers) into an LTL program (finite supply of registers). Another difference between LTL and RTL: each node is a basic block instead of a single instruction.
- The Linear language is a linearized (laid out) version of the control flow. A certain positioning is chosen for each basic block, then control-flow instructions (conditional or unconditional jumps, labels) are inserted to replicate the control-flow graph. The original CompCert heuristics used by this pass are detailed in chapter 8.
- The Mach language lays out the activation records. Mach is a language close to assembly that abstracts from the ABI (Application Binary Interface) and the ISA (Instruction Set Architecture). It is further detailed in chapter 3.
- The final language Asm models the concrete assembly's semantics, with all the necessary instructions, functionally modelled.

As detailed in Section 1.3.2, the semantics of each IR are specified using a Labeled Transition System (LTS)—a set of transitions between states, with a (possibly empty) execution trace assigned to each transition. While the intermediate languages of the frontend have a very rich state, with various environment variables, the intermediate languages of the backend have increasingly simple state, with the final language Asm featuring a very barebones state mainly containing the state of the registers (including Program Counter (PC) and Return Address (RA)) and the state of the memory.

These semantics (from Clight to the target assembly) are designed as deterministic.[7] Indeed, this design choice simplifies the correctness proof of the compiler. It reduces to proving the following: if the source code execution is valid (does not end in an invalid state) and produces an execution trace, then the execution of the compiled assembly code will also be valid and will produce the same execution trace. For instance, for a program containing just arithmetic computations and `printf()` calls, the execution of the assembly code should produce the same sequence of `printf()` calls as dictated by the semantics of the source program.

[7] Undefined behaviours [1, §3.4.3] are either defined to lead to an invalid execution or are assigned precise semantics. Unspecified behaviours [1, §3.4.4] are also given precise semantics.

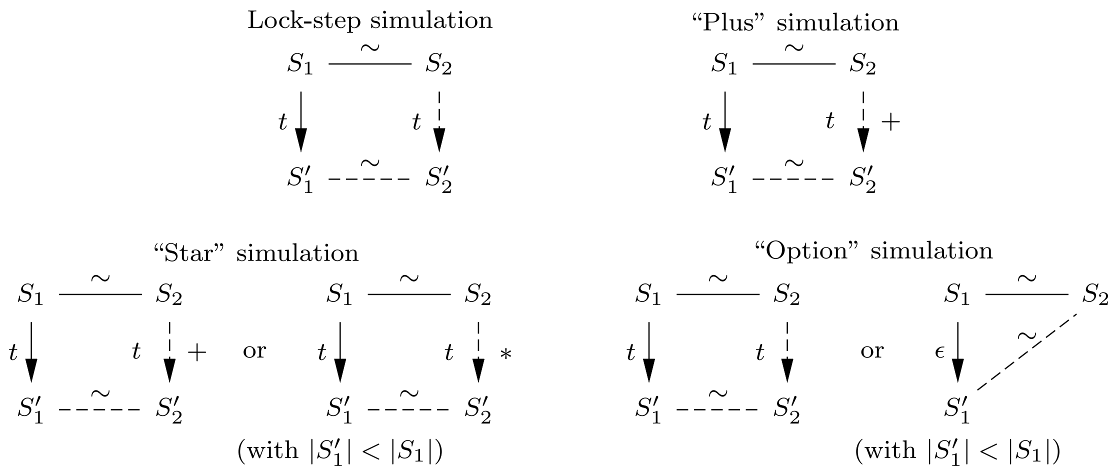


Figure 1.4.: Examples of Simulation Diagrams in CompCert, from Leroy [51]

Within the backend, compilers usually first introduce an unbounded number of *pseudo-registers*, which are then mapped to actual machine registers, with possible *spills* (saving on the stack, then reloading) when needed. This mapping is performed by the *register allocation* pass. Compiler backend passes can then be divided into two groups: those happening before register allocation (prepass) and those happening after (postpass).

As of today, the mainstream version of CompCert [26] does not have any reordering optimization, nor does it have any loop optimization such as loop unrolling, loop rotation or loop-invariant code motion. Because of that, CompCert's performance on in-order processors is much lower than GCC, despite the already present optimizations.

1.3.2 CompCert's Framework for Proving Transformation Passes Correct

In CompCert [51], the semantics of a program P consists of predicates for describing initial and final states, as well as a predicate $S \xrightarrow{t} S'$ (usually named `step`) indicating if *one* execution step *can* run from state S to state S' by generating a *trace* t—where t is either a single *observable event* (*e.g.* an external call or an access to a volatile variable) or ϵ (absence of observable event). The formal correctness property of CompCert expresses that, given a source program P_1 *without undefined behavior* (i.e. that can always run `step` from a non-final state), if the compilation of P_1 produces some assembly program P_2, then the "*observational behaviors*" of P_2 are included in those of P_1, as formalized by Leroy [50, 51]. In order to simplify correctness proofs of its successive passes (Fig. 1.3), CompCert uses an alternative definition for correctness. One of them is *forward simulation*, applicable to passes between *deterministic* languages.

In its simplest form (*lockstep simulation*), given a relation ($\sim$) matching states between P_1 and P_2, a forward simulation involves proving that:

1. the initial (resp. final) states of P_1 match those of P_2;
2. given two matching states, if P_1 steps to a state S'_1, then P_2 steps to a state S'_2 matching S'_1.

In the semantic model of CompCert, each external call, access to volatile variable and builtin will produce an *event*. An event is composed of a list of input values (e.g. the arguments to a function call) and a return value if any (e.g. the return value of the function call). A *trace* is then defined as a sequence of events. CompCert classifies program *behaviors* in three categories:

- `converges(r, t)`: the program eventually terminates, with a return code r and a finite trace t.
- `diverges(t)`: the program never terminates, with a possibly infinite trace t.
- `goeswrong(t)`: the program fails (e.g. division by zero or out-of-bound access), with a finite trace t.

Let S and C be respectively the source and compiled programs. The notation $S \Downarrow B$ denotes that B is an observed behavior of the source program S. Let *Spec* be a set of behaviors that do not go wrong (i.e. without undefined behaviors like invalid memory accesses). The notation $S \models Spec$ means that all the observed behaviors of S are in *Spec*:

$$S \models Spec \text{ iff } \forall B,\ S \Downarrow B \implies B \in Spec \tag{1.1}$$

CompCert proves a property of semantic preservation: if S verifies a specification *Spec*, then so does C.

$$S \models Spec \implies C \models Spec \tag{1.2}$$

The modular design of CompCert splits the compilation into different Intermediate Representations (IR), and proves semantics preservation between each of these IR (Fig. 1.3). Actually, when languages at both sides of the pass are deterministic, CompCert proves the semantic preservation from an easier property known as forward simulation: it states that if S has an observable behavior B that does not go wrong, then C will observe the same behavior: [50; 51]

$$\forall B \notin Wrong,\ S \Downarrow B \implies C \Downarrow B \tag{1.3}$$

Given semantics of the source and target languages (that is, a mapping from a program to its observable behaviors), the property can be proven using structural induction on the predicate

$S \Downarrow B$. This is still not practical to prove directly, so CompCert uses specific schemes (e.g., lock-step simulation) that each imply forward simulation.

A semantics of a language L in CompCert consists of:

- An inductive type `State` describing the states of execution of a program in a language L
- A predicate ($\texttt{step}\ L_1\ S_1\ t\ S'_1$) indicating if we can transition from state S_1 to state S'_1 by generating a trace t in the language L_1
- A predicate ($\texttt{initial_state}\ L\ prog\ S$) indicating if state S is an initial state of the program $prog$, in the language L
- A predicate ($\texttt{final_state}\ L\ S\ ret$) indicating if state S returning with value ret is a final state, in the language L

Let us assume we want to write a compilation pass from language L_1 to L_2 and that we have a ($\texttt{match_state}\ S_1\ S_2$) predicate indicating if the states S_1 from L_1 and S_2 from L_2 match. Let us also assume the following hypotheses:

- The public symbols are preserved across the transformation (not formalized here)
- The initial states match: $\forall S_1,\ \texttt{initial_state}\ L_1\ S_1 \implies \exists S_2,\ \texttt{initial_state}\ L_2\ S_2 \land \texttt{match_state}\ S_1\ S_2$
- The final states match: $\forall S_1\ S_2\ r,\ \texttt{match_state}\ S_1\ S_2 \land \texttt{final_state}\ L_1\ S_1\ r \implies \texttt{final_state}\ L_2\ S_2\ r$

Lock-step simulation consists of: "if the source program can take a step, then the transformed program can take a matching step". It is expressed as:

$$\begin{aligned} &\forall S_1\ t\ S'_1\ S_2,\ \texttt{step}\ L_1\ S_1\ t\ S'_1 \land \texttt{match_states}\ S_1\ S_2 \\ &\implies \exists S'_2,\ \texttt{step}\ L_2\ S_2\ t\ S'_2 \land \texttt{match_states}\ S'_1\ S'_2 \end{aligned}$$

Lock-step simulation can be visually represented as a diagram, along with other kinds of simulation that also imply a forward simulation (cf figure 1.4).

The "plus" simulation is like the lock-step one, but one may execute several instructions of the target program for one in the source program, instead of just one.

1.4 Motivations of this Thesis and Contributions

The motivations of this thesis have then been the following:

- To identify optimizations that would increase performance on in-order processors, and in particular on VLIW processors.
- Implement those optimizations and measure whether they were successful in improving the performance.
- Find efficient ways to formally verify these optimizations.

In this thesis, I focused on the KV3 interlocked VLIW processor from Kalray, though most of its results also extend to other in-order processors including non-VLIW processors such as ARM Cortex A53 and RISC-V Rocket (for which our optimizations have been easily ported, with the help of Léo Gourdin).

With the help of David Monniaux and Sylvain Boulmé, I contributed the following:

- A port of CompCert to the KV3 architecture, inspired by the existing RISC-V port and the partial port of Barany [5] on the prior KV2 architecture.[8]
- A modelling of instruction bundles for the KV3 (chapter 3). This work could be adapted for any other interlocked VLIW processor.
- A postpass scheduling optimization: an instruction reordering pass happening after register allocation (see chapter 3). In our case, it happens directly at the assembly IR of CompCert and transforms each basic block into a list of bundles that are then emitted by the assembly printer. The verification is done using a similar symbolic execution technique to that of Tristan and Leroy [95], but in Asm instead of Mach, and with hash-consing. To formally verify this optimization, we also had to define semantics for basic-block execution, reconstructing them at the Mach level and re-adapting the Mach to Asm translation accordingly (chapter 3). When designing this optimization, we kept in mind ease of adaptation to other architectures; Léo Gourdin, Sylvain Boulmé and David Monniaux later ported it to AArch64 with relative ease (a couple of person-months) [89].
- A superblock scheduling optimization: an instruction reordering pass before register allocation, on the scope of superblocks (a kind of extended basic-blocks). This work includes the implementation of static branch prediction (chapter 8), the definition of new superblock execution semantics in RTL (chapter 5), a minor change of the `Linearize` oracle for laying out superblocks contiguously in memory (chapter 8), and the verified superblock scheduling transformation itself (chapter 7) with hash-consing.
- A series of more minor optimizations consisting of duplicating instructions to increase optimization opportunities (chapter 8): loop unrolling, tail-duplication, loop peeling and loop rotation. These transformations are all verified a posteriori within a lightweight

[8] Barany's backend generates only one instruction per bundle. He also faced the challenge of representing pairs of 32-bit registers in CompCert for handling 64-bit floating-point values on the KV2. The KV3 natively has 64-bit registers, so that has not been an issue for us.

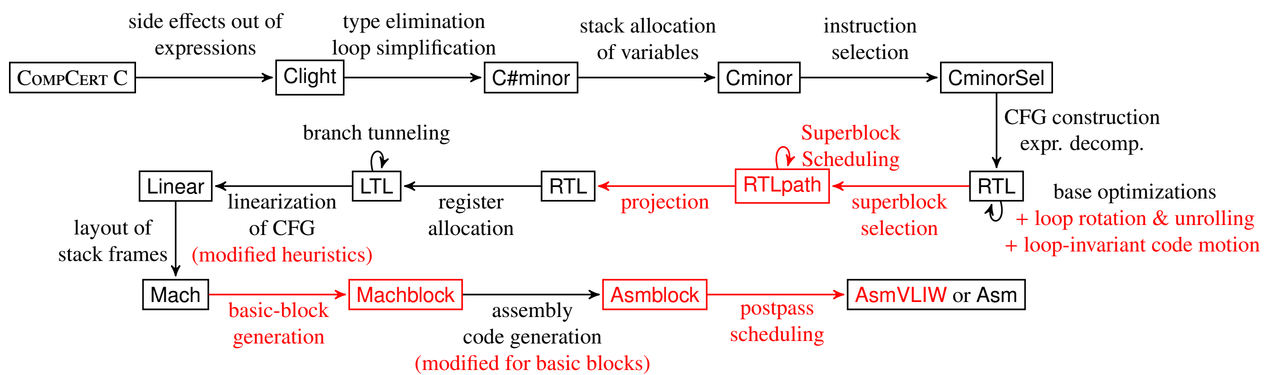


Figure 1.5.: CompCert compilation flow with our added optimizations in red

module described in chapter 6. In particular, when loop peeling is combined with the new CSE3 pass of David Monniaux, we achieve Loop Invariant Code Motion [63].

- An evaluation of CompCert on the Polybench and TACleBench benchmarks, with an evaluation of the performance gain induced by each introduced optimization, and some results obtained for other in-order processors (chapter 10). In addition to the Kalray KV3 core, with the further help of Léo Gourdin, we evaluated CompCert with our optimizations on an ARM Cortex A53 (on a Raspberry Pi 3) and a "Rocket" RISC-V core (on FPGA).

I also participated in the following publications:

- Six et al. [88] describes the formally verified postpass basic-block scheduling with bundles semantics, published in OOPSLA'20.
- Monniaux and Six [63] describes a formally verified Loop Invariant Code Motion, using code duplication to hoist instructions outside of the loop (in particular, the loop-invariant instructions are hoisted), then CSE3 to eliminate loop-invariant computations within the loop. This was published in LCTES'21.
- Six et al. [89] describes the formally verified prepass superblock scheduling, submitted for publication.

Figure 1.5 shows the new compilation flow of CompCert with our new optimizations.

Thanks to this work, compared to GCC -O2, and for most of our benchmarks, code compiled with CompCert is now, on average, only 16% slower (instead of 50% slower) on KV3, 12% slower (instead of 38%) for the ARM Cortex A53, and 30% slower (instead of 45%) for the "Rocket" RISC-V core.

The full source code of this work is available on `https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-kvx`. The URL points to the most recently updated version of our forked CompCert; the version corresponding to the time of my thesis manuscript submission can be found under the tag `csix-PhD`.

During my thesis, I was an employee of Kalray under CIFRE (Convention Industrielle de Formation par la REcherche) agreement number 2018/0441.

2 Related Work

2.1 Compiler Optimizations for VLIW Processors

2.1.1 Instruction Scheduling

Scheduling methods (for the *compaction* problem: how to parallelize a sequential micro-code) have been thoroughly studied by Fisher [30], Rau et al. [78], Chang and Hwu [20]. Some of these methods were implemented in the MULTIFLOW compiler [55] for a VLIW architecture. Dupont de Dinechin [25] did a thorough study of adapting the machine scheduling algorithms from the litterature to VLIW processors.

Instruction scheduling for micro-code is a particular instance of scheduling with timing and resource constraints, which is a classical problem [62, §5.4].

In its simplest form, instruction scheduling is done in basic blocks; however, scheduling on super blocks (sequences of basic blocks where each basic block has a single predecessor, except the first) yields better results due to the bigger scheduling scope. The scheduling can be acyclic or cyclic if the scope is a whole inner loop. Lastly, it may run before (prepass scheduling) or after (postpass scheduling) register allocation.

If the scheduling happens before register allocation, then extra care must be taken to not increase register pressure too much — that is, the register allocation should not generate more register spills as a result of the prior scheduling. On the other hand, if the scheduling happens after register allocation, the reuse of registers introduced by the register allocation may hamper the scheduling (*false dependencies*).

Ideally, having the register allocation and instruction scheduling in a single pass would solve this issue; however doing so is challenging, though there is existing work like Motwani et al. [65], or more recently Lozano et al. [57]. Instead, the approach usually taken by compilers is to have two separate scheduling passes: one before register allocation, and another after register allocation.

Pouzet [77] considered the scheduling of functional programs on VLIW architectures, including forms of software pipelining (for tail-recursive programs), with equivalence defined as term

equivalence, possibly up to rewriting. The work is similar to our scheduling transformations, but Pouzet did not investigate formal proofs of the transformation.

Schulte et al. apply constraint programming to instruction selection, register allocation, code motion and other optimizations [14; 58]. Their process can even be optimal (w.r.t. their cost model) on medium-sized functions. They consider a wider class of optimizations than we do, but they do not provide any machine-verified proof of correctness. The primary goal of their approach is to identify weak points in production compilers. The machine code programs generated by their experimental compiler based on constraint solving and the production compiler are compared; if the production compiler is clearly sub-optimal, compiler designers investigate and devise an optimization pattern that may eventually be integrated into the production compiler.

Though ample work exists on scheduling optimizations, none of what was mentioned above are used in the context of a formally verified compiler such as CompCert. Some of these transformations, although promising, might require significant work to prove. In CompCert, our choice of scheduling algorithm was then partly driven by proof constraints: we wanted to implement scheduling techniques that both were likely to give good results but also were affordable to verify given the time we had. We settled for a basic block scheduler in postpass and a superblock scheduler in prepass.

2.1.2 Loop Transformations

Most programs spend most of their time on a small part of the code; that code is usually made of loops. In particular, innermost loops can be decisive in terms of performance: small improvements of even one cycle can, in some cases, result in an unexpected overall performance improvement [24]. The first loop transformations were investigated early on with the introduction of dependence graphs [46]. Today, many classical loop transformations exist to improve program performance [74, §2.4.1]. More recently, the Polyhedral model was introduced to formalize the problem of finding the right sequence of loop transformations to apply to get optimal performance [99].

In this thesis, we experimented with simple classical loop transformations: loop unrolling, loop rotation and loop-invariant code motion.

Loop Unrolling

Davidson and Jinturkar [24] did a thorough study of loop unrolling, including a classification of loops worthy to unroll, and how to determine how many times a given loop should be unrolled. As in our approach, loop unrolling is done at the backend level to not be reliant on loop counting

information. Our version is more minimalistic: we unroll one time any innermost loop whose instruction count is below a given threshold.

Loop Rotation

We implemented in this thesis loop rotation, a method to push the conditional branch at the end of the loop instead of the start to induce better postpass scheduling. This transformation is mentioned in Platzer [74, §2.5.3], though in this thesis we perform this transformation independently of loop-invariant code motion, and it is not required for loop-invariant code motion. Also, another difference is that our loop rotation is performed in RTL, an intermediate representation close to machine code. In particular, information such as loop counters (or the type of loop) is lost at that level.

Such work is probably subsumed by cyclic scheduling transformations such as rotation scheduling [21] or software pipelining [47]; but none of these methods are formally verified. Even though our approach is more limited, it has the advantage of requiring only minimal proof.

Loop-Invariant Code Motion

Traditionally, Loop-Invariant Code Motion is performed by identifying expressions whose values do not vary across successive iterations, then hoisting them before the identified loop [74].

Here, we followed a more modular approach to ease the proofs: first, we hoist an entire iteration of the loop (loop peeling), then the new common subexpression elimination pass of D. Monniaux [63] naturally removes in the loop body what was initially computed before.

2.1.3 Other Optimizations

Static Branch Prediction

For some transformations such as superblock scheduling (or, more generally, trace scheduling), the compiler must have some knowledge of which part of the code will be executed the most [20]. Inserting profiling information and running the code on a "training" dataset (a dataset whose execution is representative of the dataset to run) is one way to get such information.

However, training datasets are not always available – also, their execution might not cover 100% of the code, leaving no choice to the compiler but to make guesses for the rest.

Ball and Larus [4] studied static branch prediction algorithms to have the compiler make fair guesses under the absence of such information. We drew from this work to provide CompCert with static branch prediction.

Code Positioning

Code positioning consists in laying out the code in memory in such a way that improves performance, typically by minimizing the penalty of branching. When having access to profile information, this problem was tackled by Pettis and Hansen [72]. One advantage here compared to scheduling transformations is that CompCert already has a verifier for code positioning; the heuristics giving the positioning can then be modified without changing the proof. However, their work requires to have exact branch profiling information.

We do not always have access to exact branch profiling information, so we did not use this work for positioning our superblocks; instead, we adapted the existing CompCert heuristics with a small modification.

2.2 Formally Proven Compiler Transformations

2.2.1 Advanced Compiler Optimizations in CompCert

Translation validation, first introduced by Pnueli et al. [75], consists in feeding both the translated code and the original code through a verifier, which will check for semantic discrepancies between the two versions.

This verifier may use various techniques: from static analyses (e.g. [81; 92]) to generation of verification conditions in SMT solvers (e.g. [104; 28]). While SMT solving has been successful to formally prove the correctness of the compilation of the seL4 operating system [86; 85], this technique seems too compute-intensive to include in a general-purpose compiler. Moreover, [67] and [94] established that *symbolic execution* combined with *rewriting* is effective enough in *validating* the code produced by state-of-the-art compilers applying various optimizations.

In particular, using translation validation to certify instruction scheduling was previously explored by Tristan and Leroy [96]. They extended CompCert with a certified postpass list-scheduler, split into

1. an untrusted oracle written in OCaml that computes a schedule for each *basic block*[1] in order to minimize pipeline stalls
2. a checker—certified in Coq—that verifies the oracle results.

Compared to our postpass scheduler, their scheduling operates at the Mach level, simpler than Asm. Since some aspects (stack and control flow handling) are only detailed in the Mach to Asm pass, a model of latencies and pipeline use at the Mach level can only be less accurate. Furthermore, our postpass scheduling needed access to the concrete Asm instructions to construct well-formed bundles.

Also, their checker has exponential complexity w.r.t. the size of basic blocks, making it slow or even impractical as the number of instructions within a basic block grows [95, §7][93, §6.7.1]. This is problematic since certain programs, such as unrolled computational loops or some forms of cryptography, may have up to a thousand instructions within a block. We thus needed to devise new algorithms that scale much better but that can still be proved correct in Coq.

Compared to our superblock scheduling, trace scheduling (as introduced by Fisher [31]) is more general since code may be moved below or above side entrances on a given trace. Moving code below a side exit requires duplicating instructions (also called *bookkeeping*); this is very similar to the tail-duplication used in superblock scheduling. Moving code above a side entrance, however, requires more complex bookkeeping. The trace scheduling of Tristan and Leroy [96] was implemented at the Mach level, as an extension to their prior list scheduling. Aside from the exponential blow-up, there are two further differences between our approach and theirs:

- Our approach is more modular. We dissociate tail-duplication (bookkeeping) from superblock scheduling. We can then apply optimizations such as common subexpression elimination or constant propagation between the bookkeeping and the scheduling.
- In the trace scheduling of Tristan and Leroy [96], the instructions are systematically duplicated when moving above a side exit. On our end, however, we are able to move instructions above side exits without duplication under sufficient conditions expressed through a liveness analysis.

Tristan and Leroy [97] use a mapping from RTL original nodes to transformed nodes to prove Lazy Code Motion; we use a similar concept for the `Duplicate` verifier in chapter 6 and for the superblock scheduling verifier of chapter 7.

Lazy Code Motion subsumes Loop-Invariant Code Motion; the software pipelining of Tristan and Leroy [98] probably subsumes loop rotation as well. One should note however that none of the above-cited work was integrated into CompCert due to excessive compilation times. Also, their work was implemented on a CompCert from 10 years ago: it is hard to say the effort that

[1] A *basic block* is defined as a sequence of instructions with a single entry point (possibly named by a label in front of the sequence) and a single exit point (e.g. a control-flow instruction at the end of the sequence).

would involve a re-adaptation of their work today. Finally, their lack of evaluation of generated code efficiency or description of the underlying oracles makes it hard to compare their work to ours in terms of optimization gain.

In comparison, our approach has been more pragmatic: the goal of this thesis was to improve performance as much as possible, with minimal proof effort.

Inspired by their work, Barthe et al. [6] did the ambitious task of integrating a formally verified SSA-based middle-end within CompCert, using translation validation. This entails giving precise semantics to SSA, transforming from RTL to SSA, implementing Global Value Numbering (GVN) and sparse conditional constant propagation (SCCP) within SSA, then transforming SSA back into RTL. The proof effort is significant: approximately 15000 lines of Coq code. The CSE3 optimization of Monniaux and Six [63] achieves a similar gain in the generated code efficiency compared to SSA-GVN, but in a much lighter approach.

2.2.2 Other Related Work

Besides CompCert, there are other projects that generate low-level code in a formally verified fashion.

The CakeML system is very similar to CompCert, except that its source language is functional (a variant of ML) instead of imperative [91]. It features the same ideas as CompCert: splitting the translation into many smaller transformations, with 12 intermediate languages and some optimizations. Their formally verified backend can generate ARM, x86, or RISC-V code. To the best of our knowledge, it does not feature any scheduling optimization.

Pit-Claudel et al. [73] recently introduced a novel approach to rely on using the Coq extractor to generate assembly code instead of a compiler that is proven once and for all. High-level specifications are written in Gallina, then refined into non-deterministic functional programs in a Gallina DSL. These programs are then soundly extracted to an imperative intermediate language (Facade) with a new proof-generating extraction. The generated programs are then translated to Bedrock assembly [22] by a more traditional verified compiler approach. This comes with the drawback of having longer compilation times. However, compared to certified compilers, their approach is lighter, more easily extensible, and easier to prove.

Some other work on verifying compiler transformations includes Vellvm, an ambitious project to formally verify the transformations done in the LLVM compiler [103; 102]. To the best of our knowledge, they did not formalize yet any of the optimizations introduced in this thesis.

Part II

Formally Verified Optimizations

3

Formal Verification of Basic-Block Scheduling at the Asm Level

Reordering instructions (which can include bundling in the case of VLIW architectures) was our first step in our quest to improve performance for the in-order VLIW KV3 processor. Indeed, CompCert does not attempt to reorder operations, which are issued in almost the same order as written in the source code. This may not be so important on processors with out-of-order or speculative execution (e.g., x86) since such hardware may dynamically find an efficient ordering on its own. Yet, it hinders performance on in-order processors, especially superscalar ones (multiple execution units can execute several instructions at once, in parallel).

The task of reordering instructions to minimize latencies is traditionally done by two different scheduling passes, one before register allocation (prepass) and one after (postpass). For VLIW architectures, traditionally, the postpass scheduling is also in charge of generating the bundles of instructions: two instructions are in the same bundle if they are scheduled in the same time slot.

Previous work by Tristan and Leroy [95] studied *a posteriori* verification of basic-block and trace scheduling at the Mach level. However, the Mach level is not a good fit for us: modeling bundles at the Mach level would have been challenging and imprecise. Indeed, the Asm generation pass can expand a Mach instruction in several Asm instructions. Instead, expressing our postpass scheduling at the Asm level allows us to have more precise latency and resource constraints information. In terms of scope, we chose to implement basic-block scheduling: it is much simpler than trace scheduling and is enough to get a substantial performance increase.

Implementing a formally verified basic-block scheduler in CompCert presents us with several challenges:

- *How to represent bundles in* CompCert*?* The existing Asm languages are all sequential. The key challenge for modeling bundle execution lies in defining semantics: on the concrete processor, each instruction within a bundle is executed in parallel! More precisely, the register reads are performed first, and then the register writes are done. Thus, it is

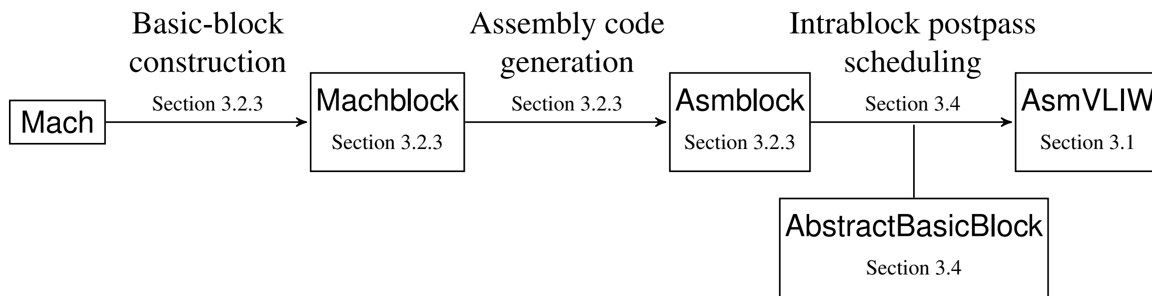


Figure 3.1.: Architecture of our postpass scheduling

possible to write and read from the same register in the same bundle. However, writing twice in the same register is not deterministic and should be avoided.

- *How to formally verify a postpass scheduling in* CompCert*?* For a VLIW architecture, the postpass scheduling performs two tasks: reordering the instructions according to a given schedule and packing the instructions of the same timeslot into the same bundle. The compiler must ensure that the transformations of the initial sequential code into a given sequence of bundles are correct and that each bundle does not exceed the available resources on the target processor. In CompCert, we do not need to verify formally that the bundles obey the resource constraints limits: if we emit bundles impossible to encode, the assembler will return an error. However, we still need to ensure formally that the generated sequence of bundles simulates the initial sequential basic block.
- *How to recover the basic-block structure in* Asm*?* We want our optimization to transform each basic block into a sequence of bundles: this raises the issue of modeling basic-block execution semantics (instead of instruction-per-instruction execution semantics).

To tackle these challenges, we adopted the code architecture in figure 3.1. First, we created a new IR called Machblock, whose sole purpose is to recover the basic-block structure. Indeed, we found it easier to do that task at the Mach level, which features semantics easier to work with than Asm. The IR, as well as the transformation pass to recover this basic-block structure, is described in section 3.2.3.

We then introduce Asmblock: an Asm IR with basic-block execution semantics. Once the basic blocks are recovered in Machblock, each instruction inside is translated to an Asm instruction. The Machblock to Asmblock transformation is inspired by the existing Mach to Asm transformation, but adapted to basic-block execution semantics — this is described in section 3.3.

Finally, our postpass scheduler (described in section 3.4) takes each basic block from Asmblock and transforms them into sequences of bundles from the IR AsmVLIW, which I describe in section 3.1.

The AsmVLIW language is our final representation. It formalizes the assembly semantics of our VLIW target: the bundles are defined as basic blocks with parallel execution inside.

Our postpass scheduler is formalized as a transformation on basic blocks. It takes as input our Asmblock IR, which shares its syntax with AsmVLIW, but with sequential execution inside basic blocks instead of parallel execution.

Our postpass scheduler from Asmblock to AsmVLIW takes each block from Asmblock, performs intra-block scheduling via an external untrusted oracle, and uses a certified, that is, formally verified, checker to verify the generated AsmVLIW bundles.

The core of our scheduling checker—involving symbolic evaluation of basic blocks with hash-consing—operates on a new auxiliary IR, called AbstractBasicBlock.

Our certified scheduler is made of:

- An oracle, written in OCaml, producing a sequence of bundles for each basic block. We implemented a greedy list scheduler with a priority heuristic based on latencies. More details on its implementation can be found in chapter 9.
- A generic certified scheduling checker, written in Coq, with a proof of semantic preservation, implementing two independent checks:
    1. Verifying that, assuming sequential execution within each bundle, the reordered basic block preserves the sequential semantics of the original one. This is achieved by comparing the symbolic execution of two basic blocks, as did Tristan and Leroy [96]. The exponential complexity of their approach is avoided by introducing (verified) hash-consing.
    2. Verifying that, for each bundle, the sequential and parallel executions have the same semantics. This reduces to checking that each bundle never uses a register after writing to it.

Both the oracle and checker are *generic*; we instantiated them with the instruction set and (micro-)architecture of the Kalray KVX core.

These checks are performed on a new IR, called AbstractBasicBlock, which makes them easier to implement and prove, and which is moreover generic w.r.t the instruction set. The core of our certified scheduler is independent of the instruction set: it has been reused for the AArch64 architecture [89], though this is not detailed here.

3.1 AsmVLIW: Semantics for a VLIW Assembly Language

The Asm language of our target processor introduces syntax and semantics for *bundles* of instructions. Bundles can be seen as a special case of *basic blocks*: zero or more labels giving (equivalent) names to the *entry-point* of the block; followed by zero or more *basic instructions* – i.e. instructions that do not branch, such as arithmetic instructions or load/store; and ended with at most one *control-flow instruction*, such as conditional branching to a label.

Semantically, a basic block has a single entry-point and a single exit-point: branching from/to the middle of a basic block is impossible. Thus, it is possible to define semantics that step through each block atomically, sequentially executing the program block by block. We call such semantics *blockstep semantics*. The notion of basic blocks is interesting for scheduling optimizations: in the context of a single-threaded sequential execution, reordering the sequence of *basic instructions* in a basic block without changing its (local) blockstep does not change the (global) semantics of the surrounding program.

We provide two such blockstep semantics, which only differ on how they combine instructions within basic blocks: an IR with sequential semantics called Asmblock, and one with parallel semantics called AsmVLIW. Our backend first builds an Asmblock program from a Mach program by detecting syntactically its basic block structure (Section 3.2). Then, each basic block is split into bundles of the AsmVLIW IR (Section 3.4). Below, Section 3.1.1 gives an overview of the Asm IR of CompCert. Then Section 3.1.2 defines the syntax shared between AsmVLIW and Asmblock. Finally, Section 3.1.3 defines AsmVLIW, and Section 3.1.4 defines Asmblock.

3.1.1 Overview of CompCert's Asm IR

CompCert defines one Asm language per target processor. An Asm program consists of functions, each with a function body. Asm states are of a single kind "`State(rs, m)`" where `rs` is the register state (a mapping from register names to values), and `m` is the memory state (a mapping from addresses to values). An initial state is one where the PC register points to the first instruction of the `main` function, and the memory is initialized with the program code. The instructions of Asm are those of the target processor, each with its associated semantics that specifies how the instruction modifies the registers and the memory.

In each CompCert backend the instruction semantics are modeled by an `exec_instr` function which takes as argument an instruction i, a register state rs and a memory state m, and returns

the next state (rs', m') given by the execution of i; or the special state `Stuck` if the execution failed.

Here are some instructions that can be modeled in Asm (taking the KV3 architecture as example):

- `Pcall(`s`)`: calls the function from symbol s (saves PC into RA and then sets PC to the address of s);
- `Paddw(`rd`, `$r1$`, `$r2$`)`: writes in rd the result of the addition of the lower 32-bits of $r1$ and $r2$;
- `Pcb(`bt`, `r`, `l`)`: evaluates the test bt on register r—if it evaluates to true, PC jumps to the label l;
- `Pigoto(`r`)`: PC jumps to the value of the register r.

Most of the instructions modelled in Asm directly correspond to the actual instructions of the processor. However, there can also be a few pseudo-instructions like `Pallocframe` or builtins, which are specified in Asm, then replaced by a sequence of instructions in a non-certified part of CompCert. Due to distinctions in immediate vs. register operands, word vs. doubleword operand size etc., there are as many as 193 instructions to be modelled in the KVX processor.

3.1.2 Syntax of Bundles & Basic Blocks

We first split the instructions into two main inductive types: `basic` for basic instructions and `control` for control flow ones. Then, a basic block (or a bundle) is syntactically defined as a record of type `bblock` with three fields: a list of labels, a list of basic instructions, and an optional control flow one.

```
1  Record bblock := { header: list label; body: list basic; exit: option control;
2                     correct: wf_bblock body exit }
```

In our AsmVLIW and Asmblock semantics, on a `None` exit, the PC is incremented by the number of instructions in the block. This convention makes reasoning easier when splitting a basic block into a sequence of smaller ones. To avoid infinite stuttering potentially caused by PC incrementations by 0 (in the case of empty blocks), we further require that a block should contain at least one instruction.

Sections 3.1.3 and 3.1.4 define, respectively, the parallel and the sequential blockstep semantics of this syntax. A state in AsmVLIW and Asmblock is expressed the same way as in Asm: it is either a pair (rs, m) where rs (register state) maps registers to values and m (memory) maps addresses to values, or a `Stuck` in case of failure (e.g. division by zero). Hence, executing a

single instruction in our semantics gives an `outcome` defined as either a (`Next rs m`) state or a `Stuck` execution. Then, each blockstep takes as input an initial state (rs, m), fetches the block pointed by rs[PC] (the value of PC in rs), and executes the content of that block. Finally, that blockstep either returns the next state or propagates any encountered failure.

3.1.3 Parallel Semantics of AsmVLIW

A bundle is a group of instructions that are to be *issued* in the same cycle through the pipeline. The pipeline stages of our interlocked VLIW processor can be abstracted into:

- **Reading Stage**: the contents of the registers are fetched.
- **Computing Stages**: the output values are computed, which can take several cycles. Once an output is computed, it is *available* to other bundles waiting at the reading stage.
- **Writing Stage**: the results are written to the registers.[1]

Our processor stalls at a reading stage whenever the result is not yet available. The exact number of cycles required to compute a value thus only impacts the performance: our formal semantics abstracts away the computing stages and only considers the reading and writing stages.

Reads are always deterministic: they happen at the start of the execution of our bundle. However, the order of writes is not necessarily deterministic, *e.g.* if the same register is written twice within the same bundle. We first introduce deterministic semantics where the writes are performed in the order in which they appear in the bundle. For instance, the bundle "$R_0 := 1; R_0 := 2$" assigns 2 to R_0, in our in-order semantics. The actual non-deterministic semantics is then defined by allowing the execution to apply an arbitrary permutation on the bundle before applying the in-order semantics.

In-Order Parallel Semantics

We model the reading stage by introducing an internal state containing a copy of the initial state (prior to executing the bundle). Such an internal state is thus of the form (rsr, rsw, mr, mw) where (rsr, mr) is the copy of the initial state, and (rsw, mw) is the running state where the values are written. Fig. 3.2 schematizes the semantics.

- The function (`bstep` b rsr rsw mr mw) executes the basic instruction b, fetching the values from rsr and mr, and performing the writes on rsw and mw to give an `outcome`.

[1] This includes the Program Counter register, which is updated at the end of each bundle execution.

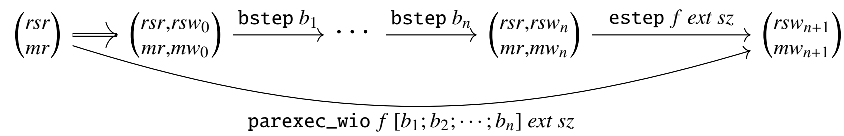


Figure 3.2.: Parallel In-Order Blockstep

- The function (`estep` f ext sz rsr rsw mw) does the same with the optional control flow instruction ext: if there is no instruction, then it just increments PC by sz, the size of the block; here, f is the current function—in which branching instructions look for labels, like in other Asm semantics.
- The function (`parexec_wio` ... rsr mr) is the composition of the basic and control steps.

For example, the bundle "$R_0 := R_1;\ R_1 := R_0;$ `jump@toto`" runs over an initial register state $rsw_0 = rsr$ in 3 steps:

1. "$R_0 := R_1$" leads to $rsw_1 = rsw_0[R_0 \leftarrow rsr[R_1]]$
2. "$R_1 := R_0$" leads to $rsw_2 = rsw_1[R_1 \leftarrow rsr[R_0]]$
3. "`jump @toto`" leads to $rsw_3 = rsw_2[\mathrm{PC} \leftarrow \texttt{@toto}]$

The final register state of the parallel in-order blockstep is

$$rsw_3 = rsr[R_0 \leftarrow rsr[R_1];\ R_1 \leftarrow rsr[R_0];\ \mathrm{PC} \leftarrow \texttt{@toto}]$$

As expected, this bundle swaps the contents of R_0 and R_1.

The in-order parallel execution of a list of basic instructions is formally defined in Coq by the following function, where "`NEXT` $rs, m \leftarrow e_1$ `IN` e_2" is a notation for:

"`match` e_1 `with Next` rs $m \Rightarrow e_2$ `| _ ⇒ Stuck end`"

```
1  Fixpoint parexec_wio_body bdy rsr rsw mr mw : outcome :=
2  match bdy with nil ⇒ Next rsw mw
3  | bi::bdy' ⇒ NEXT rsw', mw' ← bstep bi rsr rsw mr mw IN parexec_wio_body bdy' rsr rsw' mr mw'
4  end
```

The in-order parallel execution of a block (defined below) first performs a parallel in-order execution on the body (the list of basic instructions) and then performs a parallel execution with the optional control-flow instruction. Here, `f` is the current function, and `sz` is the offset by which PC is incremented in the absence of a control-flow instruction.

```
1  Definition parexec_wio f bdy ext sz rs m :=
2  NEXT rsw', mw' ← parexec_wio_body bdy rs rs m m IN estep f ext sz rs rsw' mw'
```

Deterministic Out-of-Order Parallel Semantics

The in-order parallel semantics defined above is not very representative of how a VLIW processor works, since concurrent writes may happen in any order. This issue is solved by relation (parexec_bblock f b rs m o), which holds if there exists a permutation of instructions such that the in-order parallel execution of block b with initial state (rs, m) gives the outcome o.

```
1 Definition parexec_bblock f b rs m o: Prop :=
2 ∃ bdy1 bdy2, Sorting.Permutation (bdy1++bdy2) b.(body)
3              ∧ o=(NEXT rsw', mw' ← parexec_wio f bdy1 b.(exit) (Ptrofs.repr (size b)) rs m
4                  IN parexec_wio_body bdy2 rs rsw' m mw')
```

Formally, the execution takes any permutation of the body and splits this permutation into two parts bdy1 and bdy2. It first executes bdy1, then the control flow instruction, then bdy2. While PC is possibly written before the end of the execution of the bundle, the effect on the control flow takes place when the next bundle is fetched, reflecting the behaviour detailed in Footnote 1.

This semantics gives a fair abstraction of the actual VLIW processor. However, the proof of semantic preservation of CompCert requires the target language to be deterministic. Consequently, we force our backend to emit bundles that have the same semantics irrespectively of the order of writes. This is formalized by the relation below:

```
1 Definition det_parexec f b rs m rs' m': Prop :=
2   ∀ o, parexec_bblock f b rs m o → o = Next rs' m'
```

Given (rs', m'), the above holds only if all possible outcomes o satisfying (parexec_bblock f b rs m o) are exactly (Next rs' m'); that is, it only holds if (rs', m') is the only possible outcome. We then use the det_parexec relation to express the step of our AsmVLIW semantics: if det_parexec does not hold then it is not possible to construct a step.

3.1.4 Sequential Semantics in Asmblock

Asmblock is the IR just before AsmVLIW: instructions are grouped in basic blocks. These are not reordered and split into bundles yet; execution within a block is sequential.

The given sequential semantics of a basic block, called exec_bblock below, is similar to the semantics of a single instruction in other Asm representations of CompCert. Just like AsmVLIW, its execution first runs the body and then runs the control-flow instruction. Moreover, our sequential semantics of single instructions reuses bstep and estep by using the same state for reads and writes. Our semantics of single instructions is thus shared between the sequential Asmblock and the parallel AsmVLIW.

```
Fixpoint exec_body bdy rs m: outcome :=
  match body with nil ⇒ Next rs m
  | bi::bdy' ⇒ NEXT rs', m' ← bstep bi rs rs m m IN exec_body bdy' rs' m'
  end
Definition exec_bblock f b rs m: outcome :=
  NEXT rs', m' ← exec_body b.(body) rs m IN estep f b.(exit) (Ptrofs.repr (size b)) rs' rs' m'
```

3.2 Retrieving the Basic-Block Structure with Machblock

The preliminary stage of our backend constructs the Asmblock program from the Mach program. The basic block structure cannot be recovered from the usual Asm languages of CompCert. Thus, we recover it from Mach, through a new IR—called Machblock—whose syntax reflects the basic block structure of Mach programs, and then we translate each basic block from Machblock to obtain an Asmblock program.

In this section, I motivate and present our solution to reconstruct basic blocks in CompCert, necessary for the later scheduling pass.

3.2.1 The Mach IR of CompCert

A program written in the Mach IR consists of a set of functions, each with a function body. Roughly speaking, Mach is a simplified assembly language (with 3-address code instructions handling the actual registers of the target processor, except for some special registers like the program counter PC) where:

- The ABI (Application Binary Interface) is abstracted into Mach instructions allowing access to the stack and function parameters `Mgetstack`, `Msetstack` and `Mgetparam`.
- Loads and stores stay generic and are not yet expanded into the "real" load/store instructions. The same applies to branching instructions.
- Calling instructions `Mcall` and `Mtailcall` can only branch either on a function symbol or a register on an address that must be the first address of a function.
- Branching instructions such as `Mgoto` branch to labels in the current function (like in LLVM).
- There is neither a PC (Program Counter) nor a RA (Return Address) register. The remaining code to be executed is an explicit part of the current state.

Mach states describe the register state `rs`, the global memory state `m`, and the stack state `st`. They are of three kinds, with the following meanings:

- `(State st f c rs m)`: the first instruction of code `c` is about to be run in current function `f`;
- `(Callstate st f rs m)`: function `f` is about to be run, and the caller context has just been pushed on stack `st`;
- `(Returnstate st rs m)`: a caller context is about to be restored from stack `st`.

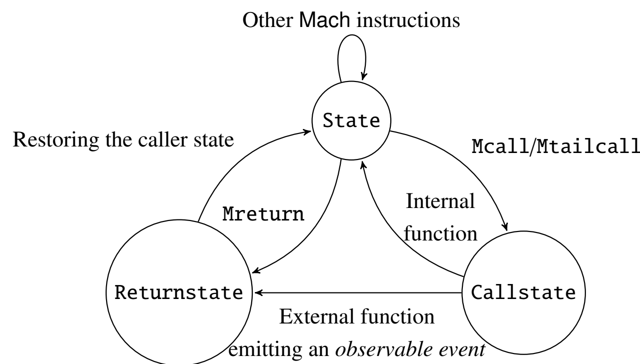


Figure 3.3.: Execution Steps between Mach States

The states of a Mach program are either of:

- `State(stack, f, sp, c, rs, m)`: we are in the current function `f`, with a call stack `stack`, a stack pointer `sp`, a Mach register [2] state `rs`, and a memory state `m`. The remaining code of `f` to execute is `c`.
- `Callstate(stack, f, rs, m)`: like `State`, but this time we are about to call `f`. A transition between `State` and `Callstate` is typically done by a `Mcall` or `Mtailcall` instruction. From there, if it's an internal function, we go back to a `State` inside the designed function - if it's an external function, we go directly to `Returnstate` and generate a trace (external event).
- `Returnstate(stack, rs, m)`: the execution of the function is over, we have now to return to the latest function in the callstack `stack`. A transition from a `State` to a `Returnstate` is typically done by a `Mreturn`.

The instructions of the Mach IR are 3-address code, providing basic interactions between the register state, memory, the function arguments, and the actual stack where temporary variables can be stored.

- `Mgetstack`/`Msetstack`/`Mgetparam`: moves between stack/function parameters and registers

[2] a Mach register basically corresponds to a general purpose register of the target assembly

- `Mop`: arithmetic operations
- `Mload`/`Mstore`: moves between memory and registers
- `Mlabel`: pseudo instruction signaling a label, used by `Mcond` and `Mgoto`
- `Mgoto`/`Mcond`: respectively unconditional and conditional branching
- `Mcall`/`Mtailcall`/`Mreturn`: calling and returning. Tailcalls are distinguished from normal calls as a small optimization.
- `Mbuiltin`: builtin instruction, whose execution generates an external event

3.2.2 Necessity of Constructing Basic Blocks at the Mach Level

One major difference between Mach and Asm lies in their semantics: the "next" Mach instructions to be executed are directly in a Mach state (as a list of instructions), whereas the Asm instructions are stored in memory, accessed by the PC register. In such Asm semantics, the PC register can jump anywhere within the function body, not necessarily to a label.[3] On the contrary, the Mach semantics ensures that jumps can only branch to particular points (labels, function entry points and return addresses). This property is not carried within the Asm IR. Unfortunately, this makes reconstructing basic blocks from Asm (in a hypothetical Asm to Asmblock pass) impossible: our AsmVLIW semantics requires that PC never fall inside a basic block.

Moreover, proving that a block-step semantics simulates an instruction-step semantics is not trivial. Hence, it seems interesting to catalyze this effort into a generic component w.r.t the processor.

Our solution is to construct the basic blocks earlier, at the Mach level, by introducing a new Machblock IR and an architecture-independent Mach to Machblock translation, then adapting the former Mach to Asm pass to a new Machblock to Asmblock pass. Not only does introducing Machblock allows separating the proof of the basic block reconstruction from the proof of the Mach to Asm translation, but it also makes part of the process reusable for other backends.

3.2.3 Machblock: Generating Basic Blocks from Mach

The basic-block syntax in Machblock is similar to that of Asmblock, except that instructions are Mach instructions and that empty basic blocks are allowed (but not generated by our translation from Mach). We have only defined sequential semantics for Machblock, which matches the Mach semantics, except that a whole basic block is run in one step. This block-step is internally made up of several computation steps, one by basic or control-flow instruction.

[3] See the `igoto` instruction in Section 3.1.1.

```
1 Record bblock := { header: list label; body: list basic_inst; exit: option control_flow_inst }
```

The code of our translation from Mach to Machblock is straightforward: it groups successive Mach instructions into a sequence of "as-big-as-possible" basic blocks while preserving the initial order of instructions. Indeed, each Mach instruction corresponds syntactically to either a label, a basic instruction, or a control-flow instruction of Machblock.

Proving that this straightforward translation is a forward simulation is much less simple than naively expected. Our proof is based on a special case of "Option" simulation (Fig. 3.5). Intuitively, the Machblock execution stutters until the Mach execution reaches the last execution of the current block, then while the Mach execution runs the last step of the current block, the Machblock execution runs the whole block in one step. Hence, the measure over Mach states—that indicates the number of Machblock successive stuttering steps—is simply the size of the block (including the number of labels) minus 1. Formally, we have introduced a dedicated simulation scheme, called "Block" simulation, in order to simplify this simulation proof. This specialized scheme avoids defining the simulation relation—written "∼" in Fig. 3.5—relating Mach and Machblock states in the stuttering case. In other words, our scheme only requires relating of Mach and Machblock states at the beginning of a block. Indeed, the "Block" simulation scheme of a Mach program P_1 by a Machblock program P_2 is defined by the two conditions below (where S_1 and S'_1 are states of P_1; S_2 and S'_2 are states of P_2; and t is a trace):

1. stuttering case of P_2 for one step of P_1 (this condition is only used within the body of a basic block: there is thus no emitted trace):

$$|S_1| > 0 \land S_1 \xrightarrow{t} S'_1 \implies t = \epsilon \land |S_1| = |S'_1| + 1$$

2. one step of P_2 for $|S_1|+1$ steps of P_1:

$$S_1 \sim S_2 \land S_1 \xrightarrow{t}{}^{|S_1|+1} S'_1 \implies \exists S'_2, S_2 \xrightarrow{t} S'_2 \land S'_1 \sim S'_2$$

Naively, relation $S_1 \sim S_2$ would be defined as "S_2 = `trans_state`(S_1)" where `trans_state` translates the Mach state in S_1 into a Machblock state, only by translating the Mach codes of S_1 into Machblock codes. However, this simple relation is not preserved by `goto` instructions on labels. Indeed, in Mach semantics, a `goto` branches to the instruction *following* the label. On the contrary, in Machblock semantics, a `goto` branches to the block *containing* the label. Hence, we define $S_1 \sim S_2$ as the following relation: "*either* S_2 = `trans_state`(S_1), *or the next Machblock step from* S_2 *reaches the same Machblock state as the next step from* `trans_state`(S_1)". The condition (2) of the "Block" simulation is then proved according to the decomposition of Figure 3.4.

On the right-hand side, c and b::bl are, respectively, the initial Mach and Machblock codes where b is the basic block at the head of the Machblock code. Relation match_state is the Coq name of the simulation relation (also noted "∼" in the paper). The Machblock step from b::bl simulates the following |b| Mach steps from c. First, skip all labels: this leads to Mach code c0. Second, run all basic instructions: this leads to Mach code c1. Finally, run the optional control-flow: this leads to code c2. Each of these three subdiagrams is an independent lemma of our Coq proof.

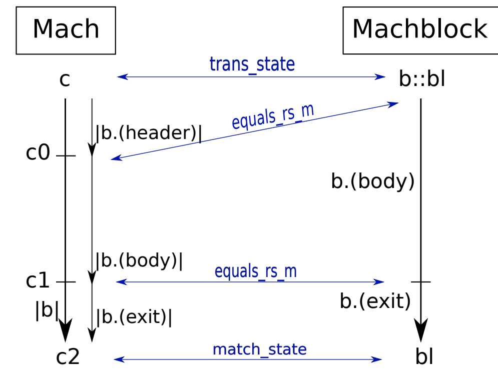


Figure 3.4.: Overview of our Proof for Condition (2) of the "Block" Simulation of Mach by Machblock

3.3 Asmblock: Adapting the Mach-to-Asm Translation for Basic Blocks

The Machblock-to-Asmblock pass is adapted from the Mach-to-Asm pass of other backends (see Section 3.3.1), but with a consequential change: instead of manipulating instructions (and reasoning on the execution of single instructions), we are now manipulating basic blocks of instructions and must reason on the execution of an entire basic block (in one single step). I give here details to orient the reader on the differences between the two approaches, particularly for the star simulation proof.

3.3.1 Translating and Proving the Translation of Mach Code to Asm

The major difference between Mach and Asm lies in the execution semantics. In Mach, the remaining code to be executed is directly in the state, and Mach semantics provide a clean notion of *internal function call*: execution can only enter into an internal function by its syntactic entry-point. In Asm, the code to be executed resides in memory and is pointed to by the PC register. Through jumps into registers and a bit of pointer arithmetic, an Asm program can jump into the middle of a function, like in Return-Oriented-Programming (ROP) [19]. Thus, proving the code-generation pass from Mach to Asm implies ensuring that the generated code does not have such a behaviour (assuming that the Mach program does not have any undefined behaviour): it simulates Mach execution where such behaviour does not exist.[4]

[4]Thus, ROP attacks on code generated by CompCert are only possible from undefined behaviours of the source code.

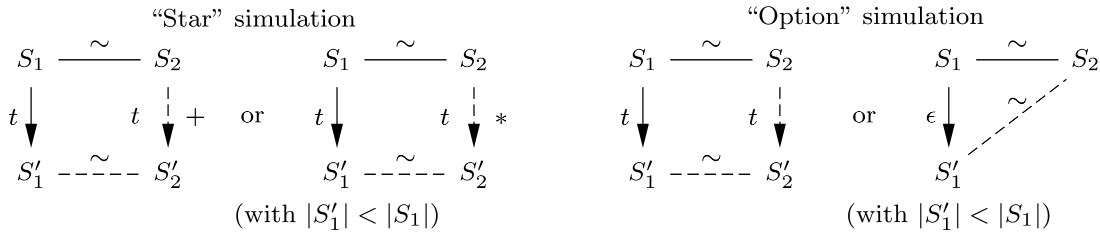


Figure 3.5.: Simulation Diagrams with Stuttering in CompCert

Formally, it involves introducing a suitable "∼" relation matching Mach states with Asm states. The gist of it consists of expressing a correspondence between the register states as well as the memory, in addition to the following properties depending on the Mach state: if it is a `State`, then the PC register points to the Asm code generated from Mach; if it is a `Callstate`, then the PC should point to the callee function, and the RA register to the return address (i.e. the address following the call instruction in the caller); otherwise, it is a `Returnstate`, and the PC should point to the return address.

Then, the proof involves a "Star" simulation. Such a simulation of a program P_1 by a program P_2 is defined — for a relation $S_1 \sim S_2$ matching states S_1 from P_1 with states S_2 from P_2 — by the following conditions:

- The initial states match

$$\forall S_1,\ P_1.\texttt{istate}\ S_1 \implies \exists S_2,\ P_2.\texttt{istate}\ S_2 \wedge S_1 \sim S_2$$

- The final states match

$$\forall S_1\ S_2\ r,\ S_1 \sim S_2 \wedge P_1.\texttt{fstate}\ S_1\ r \implies P_2.\texttt{fstate}\ S_2\ r$$

- The execution steps match through the "Star" simulation diagram (also depicted in Figure 1.4)

$$\forall S_1\ S_2\ S'_1\ \texttt{t},\ S_1 \sim S_2 \wedge S_1 \xrightarrow{\texttt{t}} S'_1 \implies \exists S'_2,\ S'_1 \sim S'_2 \wedge (S_2 \xrightarrow{\texttt{t}}{}^{+} S'_2 \vee (S_2 \xrightarrow{\texttt{t}}{}^{*} S'_2 \wedge |S'_1| < |S_1|))$$

The "Star" simulation diagram expresses that each single step of P_1 producing a trace t can be simulated by several steps of P_2 producing the same trace t. In particular, when P_1 performs an internal step (where $t = \epsilon$), P_2 can *stutter*, i.e. perform no computation step. But, if P_1 loops forever without producing any observable event, then P_2 cannot stutter infinitely.[5] Indeed, stuttering is only allowed if the step of P_1 makes the state decrease for a well-founded order (hence, sequences of *successive* stutterings cannot be infinite).

[5]Otherwise an infinite silent loop P_1 could be compiled into a program P_2 returning in one step, and this would be incorrect.

The "Star" simulation from Mach to Asm thus corresponds to proving that one Mach step (i.e. one transition of Fig. 3.3) gives the same result as several Asm instructions. For instance, the Mach step from `Callstate` into `State` is simulated by the steps of the Asm function prologue that allocate the stack frame and save it into registers `FP` (Frame Pointer) and `RA`. The Mach conditional-branching step is simulated by the Asm steps that compute the result of the condition and then branch accordingly. Actually, the only stuttering step of Asm w.r.t Mach corresponds to the *Restoring* step from `Returnstate`.

3.3.2 Translating Machblock to Asmblock

Overview

Given a basic block from Machblock, the translation consists of emitting an Asmblock basic block that performs the same operation.

The Mach to Asm pass of other backends translates code function by function, then within each function, instruction by instruction.

Our translation goes through the Machblock code function by function, then basic block by basic block. Within each basic block, the labels, basic instructions, then the control-flow instruction are translated consecutively.

Label translation is straightforward. A label in CompCert is a positive identification number, so we translate a label from Machblock to Asmblock by applying the identity function.

The translation of Machblock basic instructions gives Asmblock basic instructions. Indeed, if a Machblock instruction does not modify the control, then there is no reason for the corresponding Asmblock instructions to do so. By "not modifying the control" we mean here: not taking any branch and going to the "next" instruction in line instead. This is done by the `transl_basic_code` and `transl_instr_basic` with the following signatures (the Boolean is there for an optimization explained in the next section):

```
1 transl_basic_code:
2     Machblock.function -> list Machblock.basic_inst -> bool -> list
        Asmblock.basic_inst
3 transl_instr_basic: Machblock.function -> Machblock.basic_inst -> bool ->
    list Asmblock.basic_inst
```

The translation of Machblock control-flow instructions is less trivial. It gives a list of Asmblock basic instructions (possibly empty), followed by a single Asmblock control-flow instruction.

For example, the `(MBcond cond args lbl)` instruction which evaluates the arguments `args` with the condition `cond` and jumps to `lbl` if this evaluation is `true`. Evaluating a comparison of a register to 0 can be done in one instruction in the Kalray assembly. However, comparing two registers must be done by at least two instructions: one `(compd rd rs1 rs2)` instruction comparing the two registers `rs1` and `rs2`, writing the Boolean result to `rd`, then a `cb` (Conditional Branch) instruction using the value of `rd` to decide of the branch.

The translation of control-flow instructions is done by a `transl_instr_control` function of signature:

```
transl_instr_control:
    Machblock.function -> option Machblock.control_flow_inst -> list
       Asmblock.instruction
```

If we write $(L;\ lb;\ c)$ for Machblock basic block, the translation then consists in generating the Asmblock basic block $(L; t_b(lb) \mathbin{++} l';\ c')$ where t_b is `transl_basic_code`, and l', c' are defined such that $t_c(c) = l' \mathbin{++} (c' :: \texttt{nil})$, with t_c being `transl_instr_control`.

The Argument-Pointer Code Optimization in CompCert

For handling the loading of function arguments into registers, CompCert's backends perform an optimization that I present here.

There are many possible ways to pass the arguments to a function in assembly code. To ensure compatibility between the different compilers and hand-written assembly code, each architecture defines an ABI (Application Binary Interface), a specification of how arguments should be handled.

The arguments are usually first passed into registers and then onto the stack if there isn't enough room in the registers. In the case of the Kalray Coolidge ABI (but it is also the case for some other ABIs like the RISC-V), when there are too many arguments to be contained by the registers, the caller pushes the extra arguments on the stack. Since each stack-frame is adjacent, the callee can then access the arguments directly via its `SP` (Stack Pointer) register. For instance, if the stack size of the callee is 24 bytes, and the stack is in decreasing order (parents are in higher addresses), then the first extra argument will be at address `(SP+32)`.

This poses an issue within the memory model of CompCert. A CompCert address is a pair (b, off), where b is an integer representing the memory block, and off is the offset within that memory block. In this model, memory blocks are not contiguous. Thus, if we take the above example, `(SP+32)` would be pointing to invalid memory. Consequently, in CompCert, function arguments cannot be accessed with the `SP` register directly.

The solution adopted by CompCert backends is to keep the value of the "old SP " somewhere on the stack. It is then part of the specifications that the function prologue should save SP before modifying SP. This is done by the Pallocframe pseudo-instruction, for which we give below the specification:

```
| Pallocframe sz pos ⇒
    let (mw, stk) := Mem.alloc mr 0 sz in
    let sp := (Vptr stk Ptrofs.zero) in
    match Mem.storev Mptr mw (Val.offset_ptr sp pos) rsr#SP with
    | None ⇒ Stuck
    | Some mw ⇒ Next (rsw #FP ← (rsr SP) #SP ← sp #RTMP ← Vundef) mw
    end
```

This pseudo-instruction is performing two actions in regards to the old value of SP:

- Storing it at (Val.offset_ptr sp pos)
- Copying it to the register FP, a register arbitrarily chosen among the general-purpose registers that can be used for any operation.

This pseudo-instruction does not correspond to any particular assembly instruction: it is expanded into a sequence of assembly instructions by a not-formally-verified (but trusted) part of CompCert[6].

The copy into FP is part of a small optimization done during code generation. Loading a value from memory is expensive in most architectures, and having to load this "old SP" for each access of a parameter would result in particularly inefficient code. To alleviate this, CompCert backends remember during the code generation whether they previously loaded the old SP or not. Then, when translating a MBgetparam, additional code is inserted if the old SP isn't already loaded in FP. For instance, translating two consecutive MBgetparam writing a parameter in a different register than FP will result in at most one FP reload operation.

In order to remember whether FP is loaded, a Boolean value parameter ep is added to each translation function. The translation starts with ep=true (the prologue initially loads the old SP into FP) , then throughout the translation sets ep to true if it has been reloaded or to false if it might have been destroyed. For instance, ep is set to false every time a label is encountered, since it is not possible to know where control originated from.

More precisely, the next value of ep (outside of labels, handled separately) is given by this function (in the case of the Kalray backend, but other backends also have similar functions):

[6]It can be a source of bugs to perform such a critical operation in the uncertified part of CompCert, especially when taking into account the complex support for variadic arguments. A solution is to perform careful testing to rule them out.

```
Definition fp_is_parent (before: bool) (i: Machblock.basic_inst) : bool :=
  match i with
  | MBgetstack ofs ty dst => before && negb (mreg_eq dst MFP)
  | MBsetstack src ofs ty => before
  | MBgetparam ofs ty dst => negb (mreg_eq dst MFP)
  | MBop op args res => before && negb (mreg_eq res MFP)
  | MBload chunk addr args dst => before && negb (mreg_eq dst MFP)
  | MBstore chunk addr args res => before
  end
```

MFP stands for the Mach register corresponding to FP. In the case of an operation whose result is to be stored in FP, we must invalidate FP by setting ep to false, hence the (negb (mreg_eq dst MFP)) used in most cases of the match.

Constraints of an Asmblock Basic Block

There are two main constraints to verify when forming an Asmblock basic block. These are enforced by a wf_bblock predicate inside a bblock record.

The first constraint is that the bblock should never be empty. Indeed, by our semantics, PC is incremented by the size of the block at the end of its execution. If there are three instructions in the bblock, then, at the end of the bblock step, PC is incremented by three. If the bblock were allowed to be empty, then PC would not get modified; we would then execute the same bblock, which would cause infinite stuttering.

The second constraint comes from the Pbuiltin special instructions. A Pbuiltin instruction represents a *builtin* instruction, that is, an instruction that is inserted directly from the C code to the assembly code, without any translation from the compiler.

Here is an example of a C program containing such a builtin:

```
/* Invalidating data cache */
__builtin_k1_dinval();

int v = tab[40];
printf("%d\n", v+3);
```

The __builtin_k1_dinval builtin produces directly the instruction dinval, which is used to invalidate the data cache before accessing a value on the heap. Such instructions are specially handled by compilers.

In terms of CompCert assembly semantics, they are treated as external function calls (generating a trace). However, they pose the problem that, in assembly, we do not know exactly what they are made of. Indeed, their expansion lies in the uncertified part of CompCert, with the

only restriction that a builtin cannot create new functions or sections. As a result, each builtin instruction could generate an entire block of code, with its own labels and more than one control-flow instruction.

We could treat them the same way we treat function-call instructions, but then the uncertified scheduler could return us a schedule where the builtin is to be bundled with another basic instruction, which could lead to absurd code if the builtin is eventually expanded to several instructions and labels.

To prevent the above case, we chose to isolate each builtin with one desired property: if an Asmblock basic block has a builtin instruction, then that instruction must be the only instruction of the basic block.

This ensures that the uncertified expansion should not affect the rest of the generated code - though this is not proven formally by lack of precise builtin semantics. Like the function prologue and epilogue, test benchmarks remedy this lack of formal certification.

Let us also recall a third constraint, which is not specific to a `bblock` but rather a direct definition of our Asmblock semantics: we cannot branch in the middle of a `bblock`, *i.e.* a `Stuck` state is induced when the PC register does not point to the beginning of a `bblock`. This third constraint is necessary to define a blockstep semantics.

Function Prototypes for Implementation

I describe here the actual functions used for implementing the translation.

For each function, linearly, each basic block is translated via a `transl_blocks` function, starting with `ep=true`:

```
1 Definition transl_function (f: Machblock.function) :=
2   do lb ← transl_blocks f f.(Machblock.fn_code) true;
3   OK (mkfunction f.(Machblock.fn_sig) (make_prologue f lb))
```

The `make_prologue` consists in inserting the `Pallocframe` pseudo-instruction, as well as instructions making sure the return address is stored on the stack, to have correct linking when executing the `ret` instruction of the function epilogue.

The `transl_blocks` function goes through each basic block of the list, translates it, then propagates `ep` if the block does not have any label.

```
Fixpoint transl_blocks (f: Machblock.function) (lmb: list Machblock.bblock) (ep: bool) :=
  match lmb with
  | nil ⇒ OK nil
  | mb :: lmb ⇒
      do lb ← transl_block f mb (if Machblock.header mb then ep else false);
      do lb' ← transl_blocks f lmb false;
      OK (lb @@ lb')
  end
```

transl_block is split into three functions: transl_basic_code which translates linearly each basic instruction of the Machblock basic block, then transl_exit_code which translates the optional control-flow instruction, and finally, gen_bblocks which makes one or several basic blocks based on the results of the two last functions.

```
Definition transl_block f fb ep : res (list bblock) :=
  do c ← transl_basic_code f fb.(Machblock.body) ep;
  do ctl ← transl_instr_control f fb.(Machblock.exit);
  OK (gen_bblocks fb.(Machblock.header) c ctl)
```

The gen_bblocks function ensures that we generate bblocks that satisfy the two earlier described constraints:

- a bblock cannot be empty: while we believe this should never happen, we must nevertheless tackle what happens in the case that it does to ensure a forward simulation. We have chosen to generate a bblock with a single nop instruction if we ever come across an empty Machblock basic block.
- a builtin instruction must be alone in its bblock. When we encounter one, we split the basic block into two bblocks.

```
Program Definition gen_bblocks (hd: list label) (c: list basic) (ctl: list instruction) :=
match (extract_ctl ctl) with
| None ⇒ match c with
  | nil ⇒ {| header := hd; body := Pnop::nil; exit := None |} :: nil
  | i::c ⇒ {| header := hd; body := ((i::c)++extract_basic ctl); exit := None |} :: nil
  end
| Some (PExpand (Pbuiltin ef args res)) ⇒ match c with
  | nil ⇒ {| header := hd; body := nil; exit := Some (PExpand (Pbuiltin ef args res)) |} :: nil
  | _ ⇒ {| header := hd; body := c; exit := None |} ::
        {| header := nil; body := nil; exit := Some (PExpand (Pbuiltin ef args res)) |} :: nil
  end
| Some ex ⇒ {| header := hd; body := (c++extract_basic ctl); exit := Some ex |} :: nil
end
```

Finally, the extract_basic and extract_ctl functions (not detailed here) extract respectively the basic and the control-flow instructions of a list of instructions.

3.3.3 Proof of Forward Simulation

Issues of Using the Usual Mach-to-Asm Diagram

The usual diagram consists of starting from two states s_1 and s_2 for which a certain `match_states` relation holds, executing one Mach instruction, then executing the translated Asm instruction and proving that the two obtained states s_1' and s_2' also verify the `match_states` relation that we give below for our backend:[7]

```
Inductive match_states: Machblock.state → Asmvliw.state → Prop :=
  | match_states_intro:
      ∀ s fb sp c ep ms m m' rs f tf tc
        (STACKS: match_stack ge s)
        (FIND: Genv.find_funct_ptr ge fb = Some (Internal f))
        (MEXT: Mem.extends m m')
        (AT: transl_code_at_pc ge (rs PC) fb f c ep tf tc)
        (AG: agree ms sp rs)
        (DXP: ep = true → rs#FP = parent_sp s),
      match_states (Machblock.State s fb sp c ms m)
                   (Asmvliw.State rs m')
```

This relation ensures several things, among which:

- `AG` and `MEXT` ensure the values of registers and memory match those of the Machblock state.
- `AT` ensures that, in the memory pointed to the PC register, the Machblock code `c` is translated into the Asmblock code `tc` with the Boolean parameter `ep`.
- `DXP` ensures that if `ep` is `true`, then FP must contain the value of the old SP.

In the other CompCert backends, the `match_states` relation holds between each execution of a Mach instruction. For each possible Mach instruction, lemmas are then proved to ensure that executing, on the one hand, the Mach instruction, and on the other hand, the translated Asm instructions, lead to the same result.

In our case, using the above `match_states` directly is tricky: instead of executing one Mach instruction and reasoning on its translation, we execute an entire basic block of Machblock instructions and must reason on the translation of the whole block.

A possibility for us could be to define a finer-grain `step` relation, which would execute a single instruction of a basic block on the Machblock part. In Machblock, executing one instruction could be as simple as executing its effects, then removing it from the current basic block. On the Asmblock side, one could suggest incrementing PC instruction-by-instruction instead of doing it

[7] I am here only describing the particular case of executing code within an internal function.

all at the end of the `bblock`; however that would not be compatible with the constraint that we must not branch in the middle of a `bblock`.

A New Diagram to Prove Machblock to Asmblock

In light of this difficulty, we chose to introduce a new state definition used for reasoning at a finer grain. We call it a Codestate and define it as follows:

```
1  Record codestate :=
2    Codestate { pstate: state;          (* projection to Asmblock.state *)
3                pheader: list label;    (* list of labels *)
4                pbody1: list basic;     (* list of basics coming from the Machblock body *)
5                pbody2: list basic;     (* list of basics coming from the Machblock exit *)
6                pctl: option control;   (* exit instruction, coming from the Machblock exit *)
7                ep: bool;               (* reflects the [ep] variable used in the translation *)
8                rem: list AB.bblock;    (* remaining bblocks to execute *)
9                cur: bblock             (* current bblock to execute - useful to increment PC *)
10             }
```

A Codestate augments a state of Asmblock by including the instructions to execute, much like Machblock. It also includes the value of ep used when translating the first instruction of Codestate.

With this new Codestate, we can decompose the `match_states` relation into two relations.

The first of these two relations is `match_codestate`, which ensures an agreement between a Machblock state and a Codestate, namely: the code residing in the Codestate must have been a result of a translation of a Machblock code, the memory and register states should correspond, and also the ep value of the Codestate should match with the one used in the translation.

```
1  Inductive match_codestate fb: Machblock.state → codestate → Prop :=
2    | match_codestate_intro:
3        ∀ s sp ms m rs0 m0 f tc ep c bb tbb tbc tbi
4          (STACKS: match_stack ge s)
5          (FIND: Genv.find_funct_ptr ge fb = Some (Internal f))
6          (MEXT: Mem.extends m m0)
7          (TBC: transl_basic_code f (MB.body bb) (if MB.header bb then ep else false) = OK tbc)
8          (TIC: transl_instr_control f (MB.exit bb) = OK tbi)
9          (TBLS: transl_blocks f c false = OK tc)
10         (AG: agree ms sp rs0)
11         (DXP: (if MB.header bb then ep else false) = true → rs0#FP = parent_sp s)
12         ,
13       match_codestate fb (Machblock.State s fb sp (bb::c) ms m)
14         {| pstate := (Asmvliw.State rs0 m0);
15            pheader := (MB.header bb);
16            pbody1 := tbc;
17            pbody2 := extract_basic tbi;
18            pctl := extract_ctl tbi;
19            ep := ep;
```

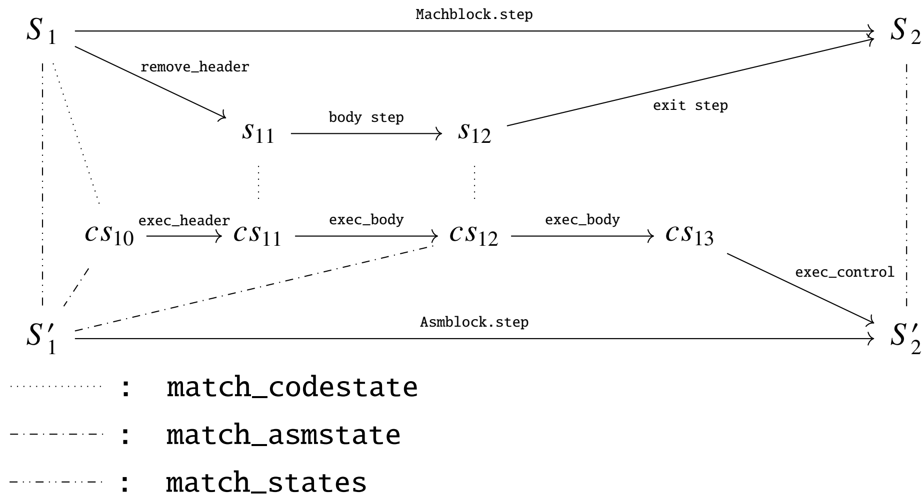


Figure 3.6.: Diagram of the Simulation Proof

```
20          rem := tc;
21          cur := tbb
22      |}
```

The second relation is `match_asmstate` between a Codestate and an Asmblock state, ensuring that the code present in a Codestate resides in memory of the Asmblock state, at the address pointed by the PC register.

```
1  Inductive match_asmstate fb: codestate → Asmvliw.state → Prop :=
2    | match_asmstate_some:
3        ∀ rs f tf tc m tbb ofs ep tbdy tex lhd
4          (FIND: Genv.find_funct_ptr ge fb = Some (Internal f))
5          (TRANSF: transf_function f = OK tf)
6          (PCeq: rs PC = Vptr fb ofs)
7          (TAIL: code_tail (Ptrofs.unsigned ofs) (fn_blocks tf) (tbb::tc))
8          ,
9        match_asmstate fb
10         {|  pstate := (Asmvliw.State rs m);
11             pheader := lhd;
12             pbody1 := tbdy;
13             pbody2 := extract_basic tex;
14             pctl := extract_ctl tex;
15             ep := ep;
16             rem := tc;
17             cur := tbb |}
18         (Asmvliw.State rs m)
```

Both relations take an extra `fb` parameter, which is the function `block`, an integer value identifying the current function.

The details of proving the Machblock to Asmblock pass with these two new `match` relations are very cumbersome (it is, in general, the case for all of the Mach to Asm proofs of the various

backends - there are a lot of details and corner cases to consider). In particular, we are not covering here the case of a builtin instruction. However, we are giving below the general idea of simulating a `bblock` without builtin, assuming we already have an existing Mach to Asm proof to base ourselves on.

Figure 3.6 depicts the diagram we went for. It can be decomposed in three theorems for simulating respectively the Machblock header, body, and exit. `exec_header` is a predicate which removes the header from the `bblock` and sets `ep` to false if there was a header:

```
1  Inductive exec_header: codestate → codestate → Prop :=
2    | exec_header_cons: ∀ cs1,
3        exec_header cs1 {| pstate := pstate cs1; pheader := nil; pbody1 := pbody1 cs1;
4                           pbody2 := pbody2 cs1; pctl := pctl cs1;
5                           ep := (if pheader cs1 then ep cs1 else false); rem := rem cs1;
6                           cur := cur cs1 |}
```

We start by the theorem `match_state_codestate`, allowing us to decompose (`match_states` S_1 S'_1) into a (`match_codestate` S_1 cs_{10}) and a (`match_asmstate` cs_{10} S'_1) with a fitting cs_{10}:

```
1  Theorem match_state_codestate:
2    ∀ mbs abs s fb sp bb c ms m,
3    (∀ ef args res, MB.exit bb <> Some (MBbuiltin ef args res)) →
4    (MB.body bb <> nil ∨ MB.exit bb <> None) →
5    mbs = (Machblock.State s fb sp (bb::c) ms m) →
6    match_states mbs abs →
7    ∃ cs fb f tbb tc ep,
8      match_codestate fb mbs cs ∧ match_asmstate fb cs abs
9      ∧ Genv.find_funct_ptr ge fb = Some (Internal f)
10     ∧ transl_blocks f (bb::c) ep = OK (tbb::tc)
11     ∧ body tbb = pbody1 cs ++ pbody2 cs
12     ∧ exit tbb = pctl cs
13     ∧ cur cs = tbb ∧ rem cs = tc
14     ∧ pstate cs = abs
```

The Machblock header simulation is then straightforward and is proven by the `step_simu_header` theorem below. After this theorem, the state s_{11} is free of any header.

```
1  Theorem step_simu_header:
2    ∀ bb s fb sp c ms m rs1 m1 cs1,
3    pstate cs1 = (State rs1 m1) →
4    match_codestate fb (MB.State s fb sp (bb::c) ms m) cs1 →
5    (∃ cs1',
6         exec_header cs1 cs1'
7      ∧ match_codestate fb (MB.State s fb sp (mb_remove_header bb::c) ms m) cs1')
```

The body simulation is then proven by induction on the list of basic instructions of s_{11}, and each individual case is covered by adapting the old proofs of Mach to Asm for the basic instructions.

```
1  Theorem step_simu_body:
2    ∀ bb s fb sp c ms m rs1 m1 ms' cs1 m',
3    MB.header bb = nil →
4    (∀ ef args res, MB.exit bb <> Some (MBbuiltin ef args res)) →
5    body_step ge s fb sp (MB.body bb) ms m ms' m' →
6    pstate cs1 = (State rs1 m1) →
7    match_codestate fb (MB.State s fb sp (bb::c) ms m) cs1 →
8    (∃ rs2 m2 cs2 ep,
9         cs2 = {| pstate := (State rs2 m2); pheader := nil; pbody1 := nil; pbody2 := pbody2 cs1;
10                 pctl := pctl cs1; ep := ep; rem := rem cs1; cur := cur cs1 |}
11      ∧ exec_body tge (pbody1 cs1) rs1 m1 = Next rs2 m2
12      ∧ match_codestate fb (MB.State s fb sp
13          ({| MB.header := nil; MB.body := nil; MB.exit := MB.exit bb |}::c) ms' m') cs2)
```

This theorem gives a state s_{12} without any body anymore, just the exit instruction. We can then use the last theorem, `step_simu_control`:

```
1  Theorem step_simu_control:
2    ∀ bb' fb fn s sp c ms' m' rs2 m2 t S'' rs1 m1 tbb tbdy2 tex cs2,
3    MB.body bb' = nil →
4    (∀ ef args res, MB.exit bb' <> Some (MBbuiltin ef args res)) →
5    Genv.find_funct_ptr tge fb = Some (Internal fn) →
6    pstate cs2 = (Asmvliw.State rs2 m2) →
7    pbody1 cs2 = nil → pbody2 cs2 = tbdy2 → pctl cs2 = tex →
8    cur cs2 = tbb →
9    match_codestate fb (MB.State s fb sp (bb'::c) ms' m') cs2 →
10   match_asmstate fb cs2 (Asmvliw.State rs1 m1) →
11   exit_step return_address_offset ge (MB.exit bb') (MB.State s fb sp (bb'::c) ms' m') t S'' →
12   (∃ rs3 m3 rs4 m4,
13       exec_body tge tbdy2 rs2 m2 = Next rs3 m3
14   ∧  exec_control_rel tge fn tex tbb rs3 m3 rs4 m4
15   ∧  match_states S'' (State rs4 m4))
```

That last theorem gives us a `match_states` between a S_2' and S_2, which is what we are looking for. The final theorem `step_simulation_bblock` then uses the four theorems to prove the *plus simulation*.

```
1  Theorem step_simulation_bblock:
2    ∀ sf f sp bb ms m ms' m' S2 c,
3    body_step ge sf f sp (Machblock.body bb) ms m ms' m' →
4    (∀ ef args res, MB.exit bb <> Some (MBbuiltin ef args res)) →
5    exit_step return_address_offset ge (Machblock.exit bb)
6          (Machblock.State sf f sp (bb :: c) ms' m') E0 S2 →
7    ∀ S1', match_states (Machblock.State sf f sp (bb :: c) ms m) S1' →
8    ∃ S2' : state, plus step tge S1' E0 S2' ∧ match_states S2 S2'
```

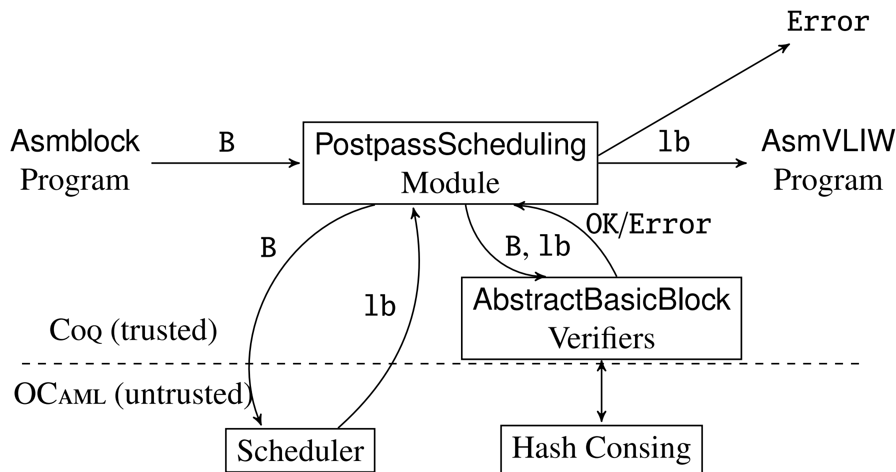


Figure 3.7.: Certified Scheduling from Untrusted Oracles

3.4 Formal Verification of the Basic-Block Postpass Scheduler

Our postpass scheduling takes place during the pass from Asmblock to AsmVLIW (Fig. 3.1). This pass has two goals:

1. reordering the instructions in each basic block to minimize stalls;
2. grouping into bundles the instructions that can be executed in the same cycle.

Similarly to Tristan and Leroy [96], our scheduling is computed by an untrusted oracle that produces a result which is checked by Coq-proved verifiers. A significant benefit of this design is the ability to change the untrusted oracle without modifying our Coq proofs.

The verifiers check semantic correctness only. If some generated bundle exceeds resource constraints, it will be rejected by the assembler.

Scheduling is performed block-by-block from the Asmblock program. As depicted in Fig. 3.7, it generates a list `lb` of AsmVLIW bundles from each basic block `B`. More precisely, a basic block `B` from Asmblock enters the PostpassScheduling module. This module sends `B` to an external untrusted scheduler, which returns a list of bundles `lb`, candidates to be added to the AsmVLIW program (scheduling is detailed in chapter 9). The PostpassScheduling module then checks that `B` and `lb` are indeed semantically equivalent through dedicated verifiers. Then, PostpassScheduling either adds `lb` to the AsmVLIW program or stops the compilation if the verifier returned an error.

In Coq, the scheduler is declared as a function[8] splitting a basic block "`B:bblock`" into a value that is then transformed into a sequence of bundles "`lb:list bblock`".[9]

```
1 Axiom schedule: bblock → (list (list basic))*(option control)
```

The proof of the pass uses a *"Plus" simulation* (Fig. 1.4): one step of the initial basic block `B` in the sequential Asmblock semantics is simulated by stepping sequentially all bundles of `lb` for the parallel AsmVLIW semantics. This forward simulation results from the composition of two others:

1. A *plus simulation* ensuring that executing `B` is the same as executing `lb` in the sequential Asmblock semantics, which proves the reordering part of the postpass scheduling.
2. A *lockstep simulation* ensuring that executing each bundle of `lb` with the Asmblock semantics gives the same result as executing this bundle with the parallel AsmVLIW semantics.

Each of these two forward simulations is derived from the correctness property of a dedicated verifier. In other words, we prove that if each of these two verifiers returns "`OK`", then the corresponding forward simulation holds. The following sections describe these two verifiers and their correctness proof. We first introduce AbstractBasicBlock, a helper IR that we use for both simulations. Then we describe the "parallelizability checker" ensuring a lockstep simulation (2). Finally, we describe the "simulation checker" ensuring a plus simulation (1).

3.4.1 AbstractBasicBlock IR

The core of our verifiers lies in the AbstractBasicBlock representation, designed by Sylvain Boulmé. AbstractBasicBlock provides a simplified syntax, in which the registers that are read or assigned by each instruction appear syntactically. The memory is encoded by a pseudo-register denoted by m.[10] We illustrate in Example 3.4.1 how we have translated some instructions into AbstractBasicBlock assignments.

Example 3.4.1 (Syntax of AbstractBasicBlock)**.** *Examples of translations into AbstractBasicBlock:*

1. *the addition of two registers r_2 and r_3 into r_1 is written "$r_1 := \mathsf{add}[r_2, r_3]$";*

[8] The scheduler is declared as a pure function like other CompCert oracles.

[9] It would be unsound to declare `schedule` returning directly a value of type "`list bblock`", since the "`correct`" proof field of `bblock` does not exist for the OCaml oracle.

[10] This encoding should be refined in the future to introduce alias analysis

2. *the load into register r_1 of memory address "$ofs[r_2]$" (representing $ofs + r_2$ where ofs is an integer constant) is written "$r_1 := (\texttt{load}\ ofs)[m, r_2]$";*
3. *the store of register r_1 into memory address "$ofs[r_2]$" is written "$m := (\texttt{store}\ ofs)[m, r_1, r_2]$".*

AbstractBasicBlock is dedicated to intra-block analyses. In AbstractBasicBlock, a block is encoded as a list of assignments, meant to be executed either sequentially or in parallel depending on the semantics. Each assignment involves an expression composed of operations and registers.

The syntax and semantics of AbstractBasicBlock are generic and have to be instantiated with the right parameters for each backend. I did that task for the KV3 backend, though this is easily extendable to other backends such as AArch64 [89].

AbstractBasicBlock provides a convenient abstraction over assembly instructions like most IRs of CompCert except Asm: it leverages the hundreds of AsmVLIW instructions into a single unified representation. Compared to the Mach IR—used in the initial approach of Tristan and Leroy [96]— AbstractBasicBlock is more low-level: it allows to represent instructions that are not present at the Mach level, like those presented in Section 3.4.6. It is also more abstract: there is no particular distinction between basic and control-flow instructions, the latter being represented as an assignment of a regular pseudo-register PC. The simplicity of its formal semantics is probably the key design point that allows us to program and prove the verifiers efficiently.

The following sections summarize how we used AbstractBasicBlock to certify the scheduling of Asmblock/AsmVLIW. A more detailed presentation of AbstractBasicBlock is available in Six et al. [88, Section 5 & Appendix C].

3.4.2 Parallelizability Checker

To check whether a certain bundle's sequential and parallel semantics give the same result, we translate the bundle into AbstractBasicBlock with a `trans_block` function, then we use the `is_parallelizable` function from AbstractBasicBlock. This checker is used right after checking that the transformed (scheduled) basic block simulates the original basic block, on each bundle emitted by the untrusted scheduler oracle.

Function `is_parallelizable` analyzes the sequence of AbstractBasicBlock assignments and checks that no pseudo-register is read or rewritten after being written once. For example, blocks "$r_1 := r_2;\ r_3 := r_2$" and "$r_1 := r_2;\ r_2 := r_3$" are accepted as parallelizable. However, "$r_1 := r_2;\ r_2 := r_1$" and "$r_1 := r_2;\ r_1 := r_3$" are rejected, because r_1 is used after being written.

When is_parallelizable returns true, the list of assignments has the same behaviour in both the sequential and the parallel semantics. This property at the AbstractBasicBlock level can be lifted back to the AsmVLIW/Asmblock level because the list of assignments returned by trans_block is proven to *bisimulate* the input block—both for the sequential and the parallel semantics. This bisimulation is also useful for the simulation checker described next section.

Proving the previously mentioned forward simulation (2) then relies on proving the following lemma bblock_para_check_correct which is proven by using the above bisimulation property.

```
1  Definition bblock_para_check bundle: bool := is_parallelizable (trans_block bundle)
2  Lemma bblock_para_check_correct ge f bundle rs m rs' m': bblock_para_check bundle = true →
3    exec_bblock ge f bundle rs m = Next rs' m' → det_parexec ge f bundle rs m rs' m'
```

The syntax of AbstractBasicBlock and its semantics are generic, in the sense that the IR is parametrized by the names of pseudo-registers and the syntax and semantics of operators. Thus, as claimed in the introduction, it can be easily reused for other processors or other IRs of CompCert.

3.4.3 Verifying Intrablock Reordering

To reason on reordering, we define a concatenation predicate: "is_concat tb lb" means that the bblock tb is the *concatenation* of the list of bundles lb. Formally, lb must be non-empty, only its head may have a non-empty header, only its tail may have a control-flow instruction, tb.(body) must be the concatenation of all the bodies of lb, and the header (resp. exit) of the head (resp. tail) of lb must correspond to the header (resp. exit) of tb.

We also define a *block simulation*: a block b is *simulated* by a block b′ *if and only if* when the execution of b is not Stuck, executing b and b′ from the same initial state gives the same result (using the sequential semantics). That is, b′ preserves any non-Stuck outcome of b.

```
1  Definition bblock_simu ge f b b' := ∀ rs m,
2    exec_bblock ge f b rs m <> Stuck → exec_bblock ge f b rs m = exec_bblock ge f b' rs m
```

The forward simulation (1) reduces to proving the correctness of our verified_schedule function on each basic block, as formalized by this property:

```
1  Theorem verified_schedule_correct: ∀ ge f B lb,
2    (verified_schedule B) = (OK lb) → ∃ tb, is_concat tb lb ∧ bblock_simu ge f B tb
```

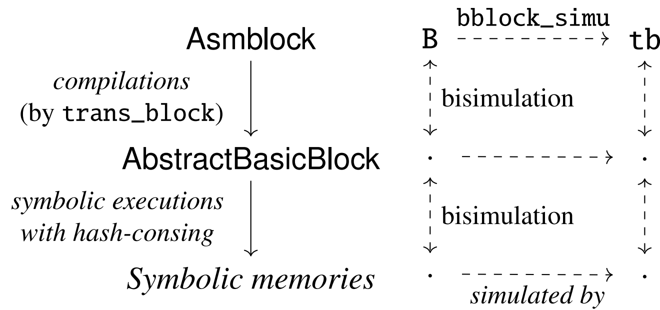


Figure 3.8.: Diagram of `bblock_simub` Correctness

In detail, `(verified_schedule B)` calls our untrusted scheduler `(schedule B)` and then builds the sequence of bundles `lb` as well as their concatenation `tb`. Then, it checks each bundle for parallelizability with the `bblock_para_check` function described in the previous section. Finally, it calls a function `bblock_simub : bblock → bblock → bool` checking whether `tb` simulates `B`.[11]

Function `bblock_simub` is composed of two steps. First, each basic block is compiled (through the `trans_block` function mentioned in Section 3.4.2) into a sequence of AbstractBasicBlock assignments. Second, like in [96], the simulation test symbolically executes each AbstractBasicBlock code and compares the resulting *symbolic memories* (Fig. 3.8). A symbolic memory roughly corresponds to a parallel assignment equivalent to the input block. More precisely, this symbolic execution computes a term for each pseudo-register assigned by the block: this term represents the final value of the pseudo-register as a function of its initial value.

Example 3.4.2 (Equivalence of symbolic memories)**.** *Let us consider the two blocks B_1 and B_2 below:*

(B_1) $r_1 := r_1 + r_2;\ r_3 := \texttt{load}[m, r_2];\ r_1 := r_1 + r_3$

(B_2) $r_3 := \texttt{load}[m, r_2];\ r_1 := r_1 + r_2;\ r_1 := r_1 + r_3$

They are both equivalent to this parallel assignment:

$$r_1 := (r_1 + r_2) + \texttt{load}[m, r_2] \parallel r_3 := \texttt{load}[m, r_2]$$

Indeed, B_1 and B_2 bisimulate (they simulate each other).

Collecting only the final term associated with each pseudo-register is actually incorrect: an incorrect scheduling oracle could insert additional failures. Thus, the symbolic memory must also collect a list of all intermediate terms on which the sequential execution may fail and that have disappeared from the final parallel assignment. See Example 3.4.3 below. Formally, the symbolic memory and the input block must be bisimulable as pictured in Fig 3.8.

Example 3.4.3 (Simulation on symbolic memories)**.** *Consider:*

(B_1) $r_1 := r_1 + r_2;\ r_3 := \texttt{load}[m, r_2];\ r_3 := r_1;\ r_1 := r_1 + r_3$

[11] More precisely, if the result of "`bblock_simub B tb`" is `true`, then "`bblock_simu ge f B tb`" holds.

(B_2) $r_3 := r_1 + r_2;\ r_1 := r_3 + r_3$
Both B_1 and B_2 lead to the same parallel assignment:

$$r_1 := (r_1 + r_2) + (r_1 + r_2) \parallel r_3 := r_1 + r_2$$

However, B_1 is simulated by B_2, whereas the converse is not true. This is because the memory access in B_1 may cause its execution to fail, whereas this failure cannot occur in B_2. Thus, the symbolic memory of B_1 should contain the term "`load`*$[m, r_2]$" as a potential failure. We say that* a symbolic memory d_1 is simulated by a symbolic memory d_2 *if and only if their parallel assignments are equivalent, and the list of potential failures of d_2 is included in the list of potential failures of d_1.*

As illustrated in Examples 3.4.2 and 3.4.3, the computation of symbolic memories involves many duplications of terms. Thus, comparing symbolic memories with structural equalities of terms, as performed in [96], is exponential time in the worst case. To solve this issue, we have developed a generic verified hash-consing factory for Coq. Hash-consing consists of memoizing the constructors of some inductive data-type to ensure that two structurally equal terms are allocated to the same object in memory. This enables us to replace (expensive) structural equalities with (constant-time) pointer equalities.

3.4.4 Generic and Verified Hash-Consing

Below is given a brief overview of the hash-consing mechanism.

Hash-consing a data-type simply consists of replacing the usual constructors of this data-type by *smart constructors* that perform the memoization of each constructor. This memoization is usually delegated to a dedicated function that can, in turn, be generated from a generic factory [29]. Our hash-consing technique follows this principle. However, whereas the memoization factory of Filliâtre and Conchon [29] (in OCaml) has no formal guarantee, ours satisfies a simple correctness property that is formally verified in Coq: each memoizing function *observationally behaves* like an identity.

Our Coq memoization function on terms invokes an external untrusted OCaml oracle that takes as input a given term and returns a memoized term (possibly memoizing the input term in the process). Then, our Coq memoization function dynamically checks that the memoized term and the input term are isomorphic or aborts the computation if it cannot ensure they are. This check is kept constant-time by using OCaml *pointer equality* to compare *already memoized* subterms: $f(t_1, \ldots, t_m)$ and $g(u_1, \ldots, u_n)$ are tested for isomorphism by checking that the head symbols f and g are identical, the numbers of arguments m and n are the same, and for all i, t_i and u_i are the same pointer. We have thus imported OCaml pointer equality into Coq.

```
1  Axiom phys_eq: ∀ {A}, A → A → ??bool
2  Extract Constant phys_eq ⇒ "(==)"
3  Axiom phys_eq_true: ∀ A (x y:A), phys_eq x y ⤳ true → x=y
```

Importing an OCAML function into COQ is carried out by declaring the type of this OCAML function through an axiom: the COQ axiom is replaced by the actual OCAML function at extraction. Using a pure function type in this COQ axiom implicitly assumes that the OCAML function is *logically deterministic* (like any COQ function): calling the function twice on *equal* inputs should give *equal* outputs—where *equality* is COQ equality: structural equality. In contrast, the OCAML *pointer equality* does not satisfy this property: two *structurally equal* values do not necessarily have the same pointer.[12]

We solve this issue by using the *pointer equality* from the IMPURE library of Boulmé [15], which embeds OCAML functions in COQ through a *non-deterministic monad*. In particular, it represents OCAML *pointer equality* as a non-deterministic function.[13] We then use the axiom from IMPURE stating that, if pointer equality returns `true`, then the two values are (structurally) equal.

Here, "`??bool`" is logically interpreted as the type of all "subsets" of Booleans; `phys_eq` is the physical equality, later extracted as the OCAML `(==)`; and "⤳" is the may-return relation of the IMPURE library: "`phys_eq x y ⤳ true`" means that "`true`" is a *possible* result of "`phys_eq x y`". In other words, even if "`phys_eq x y ⤳ true`" and "`phys_eq x y ⤳ b`", then we cannot conclude that "`b=true`" (we could also have "`b=false`"). The IMPURE library does not even assume that "`∀ x, phys_eq x x ⤳ true`".

These axioms are proved to be non-contradictory w.r.t. the COQ logic. They express a correctness property of OCAML physical equality, from which we derive efficient and formally verified hash-consing.

For more detailed information about this technique, I refer the reader to Sylvain Boulmé's HdR ([15], section 3.3.2).

3.4.5 Peephole Optimization

We have expressed the semantics of assembly instructions by decomposing them into atomic operations, which we used to define the symbolic execution. This means that distinct groups

[12] Hence, if we model pointer equality (OCAML's ==) as an infix function "`phys_eq:∀ {A}, A → A → bool`", then we are able to prove this wrong proposition (when `x` and `y` are structural equals but are distinct pointers):

```
y=x -> (phys_eq x y)=false -> (phys_eq x x)=true -> false=true
```

[13] In the COQ logic, two occurrences of variable "`x`" may correspond to two distinct objects in memory (e.g. after substituting `y=x` in "`P x y`"). This is why `phys_eq` must appear as "non-deterministic" to COQ's eyes.

of instructions that decompose into the same atomic operations are considered equivalent. We exploited this to implement *peephole optimization*: local replacement of groups of instructions by faster ones prior to scheduling. In Fig. 3.7, this untrusted optimization is performed by a preliminary pass of the "Scheduler" oracle and thus dynamically checked by our `bblock_simub` trusted verifier.

Currently our only peephole optimizations are the replacement of two (respectively, four) store instructions from an aligned group of double-word (64-bit) registers (e.g. `$r18`, `$r19`) to a succession of offsets from the same base register (e.g. `8[$r12]`, `16[$r12]`) by a single quadruple-word (respectively, octuple-word) store instruction; and the same with loads. In practice, this quickens register spilling and restoring sequences, such as those around function calls: CompCert spills and restores registers in ascending order, whereas in general, it would only be sheer chance that register allocation placed data that is consecutive in memory into an aligned group of consecutive registers. Léo Gourdin implemented a similar but slightly more complex optimization for the AArch64 architecture: loads and stores to consecutive memory locations, but not necessarily consecutive register indices, are searched for in the assembly code and grouped together.

3.4.6 Atomic Sequences of Assignments in AbstractBasicBlock

In the previous examples, each AsmVLIW instruction is translated to a single assignment of AbstractBasicBlock. However, in many cases (extended-word accesses of Sect. 3.4.5, control-flow instructions, frame-handling pseudo-instructions, etc), AsmVLIW instructions cannot correspond to a single assignment: they are rather translated to AbstractBasicBlock as an *atomic sequence of assignments* (ASA). A form of parallelism may occur in such a sequence through the special operator $\texttt{Old}(e)$ where e is an expression, meaning that the evaluation of e occurs in the initial state of the ASA. And an AsmVLIW bundle (resp. an Asmblock basic-block) corresponds to the parallel (resp. sequential) composition of a list of such ASAs.

For example, the parallel load from a 128-bit memory word involves two contiguous (and adequately aligned) destination registers d_0 and d_1 that are distinct from each other by construction—but not necessarily from the base address register a. This parallel load is thus translated into the following ASA, which emulates a parallel assignment of d_0 and d_1, even if $a = d_0$:

$$d_0 := (\texttt{load}\ i)[m, a]\ ;\quad d_1 := (\texttt{load}\ (i+8))[m, (\texttt{Old}\ a)]$$

4 General Concepts of Superblock Scheduling

We manipulated basic blocks within the postpass scheduling optimisation–sequences of instructions within a function, each with only one entry point and one exit point. Throughout the remaining chapters, we will study prepass scheduling, in RTL, at the level of superblocks: a generalisation of basic blocks where additional exit points (called *side exits*) are allowed. Compared to our previous work, superblocks are much like how we defined basicblocks (several *basic* instructions, followed by an optional *control-flow* instruction) except that this time, conditional branches are also allowed among the "basic" instructions. This chapter introduces the basic notions of superblocks: it informally describes the transformations we perform on superblocks and briefly discusses their correctness.

Superblock optimisation techniques originate from the compiler literature — in particular, the papers Hwu et al. [40] and Lee et al. [49] describe the implementation of superblock scheduling in particular compilers, along with other superblock-related optimisations. Much like trace scheduling (introduced by Fisher [31]), superblock scheduling optimises a given path of the program by moving instructions before conditional branches[1]. If the optimised path is frequently taken, the function execution will be faster. However, this is to the detriment of other paths, whose execution will be slower. Hence, while basic-block scheduling is an optimisation in the scope of a basic block, superblock scheduling has the more general scope of a trace (later turned into a superblock) within a function.

The first step to form superblocks is to identify an execution trace likely to be executed (section 4.1). That trace selection, driven mainly by *branch prediction*, outlines traces that have the potential to become superblocks of interest. To ensure faster execution (rather than a slower), it is of great interest to have sufficiently accurate branch prediction. Then, transformations are done on each interesting trace to superblockify them and increase the optimisation potential of certain superblocks (section 4.2). In particular, we perform *tail duplication* and *loop unrolling*. Finally, we schedule the resulting superblock (section 4.3): reordering the instructions, potentially moving them upwards across branches, aiming to get faster resulting machine code. We depict in section 4.4 how our optimisations are inserted among the already existing RTL optimisations.

[1] The transformation also allows to move them below conditional branches, at the discretion of the scheduler.

Similar work was done by Tristan and Leroy [95] to implement *trace scheduling* at the Mach level, in an older version of CompCert. *Trace scheduling* is more general than *superblock scheduling*: it also allows to move code below or above side entrances. Moving code below *side exits* requires duplicating instructions (this is called *bookkeeping*), much like *tail duplication* for superblocks. However, moving code above or below *side entrances* requires more complex bookkeeping. For our prepass, we chose *superblock scheduling* instead of the *trace scheduling* technique from Tristan and Leroy [95] for the following reasons:

- In trace scheduling, bookkeeping and scheduling are done in the same pass. Superblock scheduling, on the other hand, allows us to dissociate tail duplication from the actual scheduling. In particular, we can apply optimisations such as CSE3[2], constant propagation or dead code elimination, between tail-duplication (which might introduce redundant computations) and the actual scheduling, which might improve the scheduling opportunities.
- In contrast to the original trace scheduling from Fisher [31], the *trace scheduling* of Tristan and Leroy [95] systematically duplicates instructions when they are moved above a side exit; our superblock scheduling, on the other end, can move instructions above side exits without any duplication (under certain conditions expressed through a liveness analysis).

In this chapter, I will not detail the verifiers or their proofs of correctness, and I will not lay out the actual algorithms used either — this will be tackled in the later chapters. This chapter is, instead, a general-picture overview of superblock scheduling and the new passes introduced.

In practice, the transformations detailed here happen at the RTL level of CompCert. However, to properly analyse the performance, we have to consider the other passes between RTL and the final assembly code. Thus, most of the examples shown in this chapter will be shown in assembly. In particular, I will briefly discuss how the transformations presented here relate to the further passes, particularly Linearize, Postpass Scheduling and CSE3.

4.1 Guiding Trace Selection with Branch Prediction

Predicting which branch is likely to be taken is necessary to ensure that our optimisations result in a performance gain rather than a loss. Indeed, the superblock scheduling optimisation is only beneficial if the superblock is executed frequently in its entirety.

Branch prediction affects:

- Trace selection: only the traces likely to be executed are selected.
- Tail duplication, as a direct consequence of trace selection being affected.

[2]CSE3 is a common subexpression elimination that analyses across branches.

- Loop unrolling: we only unroll loops that cover a superblock, but also determining the direction of the loop (which condition leads to the loop body) is reliant on branch prediction.
- Superblock scheduling: the superblocks to be scheduled depend on branch prediction and the former passes.

If profiling information is available, we can use it to drive the above passes. However, it is still possible to "guess" which way the program is more likely to go if there is none. Our static branch prediction is based on Ball and Larus [4], with some additions and fine-tuning to their techniques. We then use predictions to select traces, using a technique inspired by Chang and Hwu [20].

4.1.1 What we Expect out of Static Branch Prediction

Let us consider a program consisting of two nested loops, like the kind shown in figure 4.1. The innermost loop is the one that interests us: this is usually where most of the computation time lies. Our goal is to apply loop transformations on that loop (such as loop unrolling) and then perform superblock scheduling. However, that loop has one other branch inside.

Ideally, we would like our static branch prediction to perform well in two aspects:

- The branch prediction should be able to predict the conditional branch correctly from the innermost loops. It should privilege the trace going within the loop instead of the one going out. Also, if there is a branch leading to exiting the loop, then that branch should be predicted to stay within the loop. This ensures that our later trace selection (followed by tail duplication) will indeed form a superblock encompassing the loop.
- In the given example (figure 4.1), the branch "in the middle" is a "may-exit" branch. However, there are other examples with branches for which both ends remain within the loop. This would happen if the body of the `if` does not have any `return` or `break`. In such a case, we should not try to wild-guess it. Indeed, without profiling information, we might be lucky and guess right — just like we might be wrong and end up decreasing performance, sometimes by a large margin. We have found it best to leave such examples untouched: the trace (superblock) would then stop at the branch instead of going through it (and consequently, we do not unroll such loops).

Generally speaking, a false negative (an unpredicted branch that could have been predicted) will just result in a lack of optimisation: it will not decrease the performance compared to the original code. However, a false positive (a predicted branch with a wrong prediction) can be harmful to performance.

```c
#include <stdbool.h>

int int_arr_eq(int **t, int **u, int nx
    , int ny){
  for (int i = 0 ; i < nx ; i++){
    for (int j = 0 ; j < ny ; j++){
      if (t[i][j] != u[i][j])
        return false;
    }
  }
  return true;
}
```

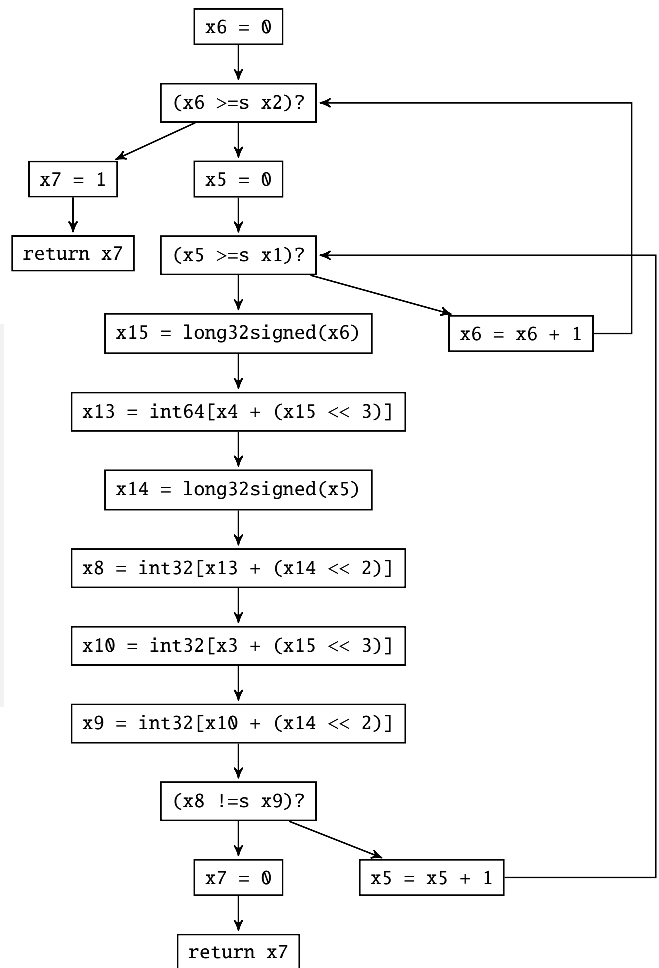


Figure 4.1.: Two nested loops

As a result, we tried to tune the static branch prediction in such a way as to get a low number of false positives but still be able to identify most of the superblockification opportunities out of innermost loops. This work has been mostly empirical — involving a lot of trying out algorithms, witnessing the impact on performance, then analysing individual examples where we saw a decrease in performance and iterating on the compiler code until we got to a satisfying state.

4.1.2 Storing the Prediction

The compiler literature (such as Lee et al. [49]) usually advises storing the profiling information under one float number per branch edge, which indicates the probability to take that edge when branching. Then, for trace selection, a path is dug following edges that have more than 50% probability.

In our case, our superblock transformations are scattered across several elementary passes. There is a need for each of these passes to get consistent information for knowing which branch is privileged. In particular, if we were annotating each conditional branch with such a probability number, then we would need each of our passes to interpret such a number consistently, to prevent cases where say a tail-duplication treated a particular set of instructions as a superblock, but the superblock scheduling treated another, different set of instructions for scheduling.

Given our compiler architecture, we also think it is better not only to store which branch should be privileged but also to point out whether there are branches for which we are *unsure* of the privileged direction. To illustrate that, let us examine the program shown in figure 4.2. This program is fictional — each letter represents one instruction, with A and D being conditional branches. If each branch has say a 51% probability to take the privileged branch, and we move instruction E_1 to the top of instruction A (because the scheduler ruled that this would give a faster execution *under the assumption that we follow the trace* $A - B - D - E_1 - E_2 - H$), then we would only have a 25% chance to get faster execution. Indeed, on that particular example, three different paths are possible: $A - B - D - E_1 - E_2 - H$ (26%), $A - B - D - F - H$ (25%) or $A - C - G - H$ (49%). That move would be only beneficial for the first path and detrimental for the other two because of the waste of time executing E_1 (wrongly) speculatively. For that particular case, we would have labelled each branch to be of *unsure* direction and stop the trace, rather than identifying a *too big* trace whose scheduling results in worse performance.

The two above points led us to adopt one `Option bool` prediction p_c per conditional branch c, with the following informal semantics:

- p_c = `None`: no prediction attached for c. Either we are unsure of which way the branch goes, or the static/profiling prediction has not been run yet.

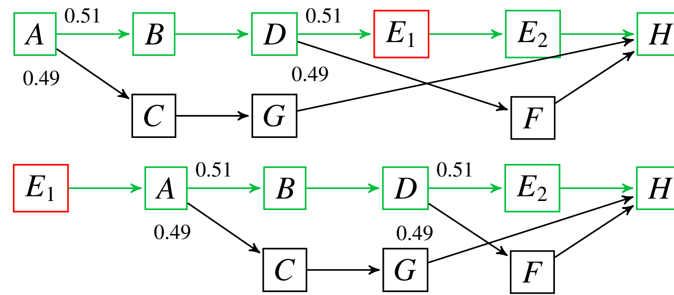


Figure 4.2.: Program with two branches. Top figure: original program. Bottom figure: the instruction E_1 is moved before A (under the non-trivial assumption that it is correct to do so). An identified trace (that could be turned into a superblock) is shown in green. Each conditional branch edge is annotated with its probability.

- p_c = `Some true`: the branch instruction c is predicted to be very likely to follow the `ifso` branch (condition evaluates to `true`).
- p_c = `Some false`: the branch instruction c is predicted to be very likely to follow the `ifnot` branch (condition evaluates to `false`).

4.1.3 Acquiring Prediction Information

Prediction information can be acquired through two means, the first of which is to use profiling. The code is instrumented to record profile information in the form of a file. Then, that file is used on the subsequent compilations to guide the prediction. In the absence of profiling information, static heuristics can be used. By exploiting specific patterns in the program, it is possible to make a fair, educated guess on the direction of most branches. In particular, innermost loops and their exit branches can be detected by static prediction: this paves the way for the later superblock optimisations.

With Profiling

We use the profiling from our version of CompCert [16] (see modules `Profiling` and `ProfilingExploit`). This profiling system was developed by D. Monniaux. The profiling instrumentation is done through the addition of additional code on top of the existing CompCert code:

- Counters are inserted into each object file as local symbols.
- Right after each branch, a special CompCert builtin `EF_profiling` is inserted, which increments the appropriate counter.
- When the `main` function exits, all the counters are written to a file. This is done through additional linker code added to each compiled object file.

Once the profiling is recorded by executing the program one time, it is then possible to use that information to add prediction information to branches. For now, the profiler-based heuristic consists of simply assigning the prediction to `None` when the two branch counts are equal, or `Some b` if one branch is executed more than another (with b chosen accordingly). In the future, this could be refined to be less aggressive (say, by choosing `None` when the relative difference between the two branch counts is too small).

D. Monniaux also implemented support for `__builtin_expect`: a compiler built-in to be placed in branches to indicate the most likely branch direction.

With Heuristics

If no profiling is available, we have certain heuristics to make an educated guess at the privileged direction. Most of these are heavily inspired from Ball and Larus [4]. The following heuristics are run in sequence until one of them decides a prediction. If none of them could decide a prediction, then the prediction information will remain `None`.

- We look at the comparison used in the conditional branch: a comparison such as $(x < 0)$? is seen as something likely to be an error-code check, so we predict that the check succeeds (that is, the condition is not taken). In the same spirit, float equality checks are predicted to be `false` most of the time.
- If a given branch leads to a `return`, then that branch is unlikely to be taken.
- If a branch leads to a backedge (a looping edge), and if the other branch does not also lead to a backedge[3] , then this is very likely to be a conditional branch from an innermost loop with one branch diving inside the loop and the other branch exiting the loop. We then predict that the looping branch should be privileged. This heuristic is critical for later identifying the superblock following that innermost loop.
- Finally, if a given branch leads to a `call` instruction and the other does not, then the branch without `call` is privileged.

Experimentally, these heuristics seem to be enough to detect most branches of interest. However, we still ran into one or two minor edge cases where the prediction failed (particularly for the first heuristic on conditions).

[3]In regards to CompCert's RTL representation and how it generates loops, two nested loops usually lead to such a case where the two branches from the conditional branch lead to a backedge. This will be explained in more detail in chapter 8.

4.1.4 Driving the Selection Based on Predictions

Once branch prediction has been done, several phases of selection are done:

- For tail-duplication, a trace selection for electing the traces that will be transformed into superblocks.
- For the loop-unrolling optimisation, a superblock selection to elect the superblocks that encompass innermost loops.
- For the superblock scheduling, another superblock selection, which aims to partition the entire code into superblocks.
- Finally, for the `Linearize` phase (linearizing the code from `LTL` to `Mach`), yet another superblock selection, with the same goal as above: partitioning the code into superblocks.

Trace Selection for Tail-Duplication

This phase is the most complex of the four mentioned above and ends up defining which parts of the code will be optimised. We mainly use the algorithm from Chang and Hwu [20], which consists of selecting a node that has the largest execution count, then growing a trace forward (by going through the "best successors" in regards to execution count), and then growing it backward. Then, the trace stops, and the next unvisited node is selected to grow a new trace.

A few key differences that our implementation has in regards to their algorithm:

- It is unclear whether their algorithm allows for traces crossing each other (the presence of nodes that would belong to strictly more than one trace). In any case, since we intend for our traces to become superblocks, we chose to prohibit the traces from crossing each other.
- They have access to precise execution counts for each node. However, in the most general case (when we have no profiling information), we would be unable to get any fair approximation of such execution counts. In the case of the presence of profiling information, it could be possible to get them. A first method would be to compute an approximation of the execution count based on our profiler's edge information. Another method would be to add profiling counters to each RTL basic block[4] and then use them to propagate the execution count on RTL nodes. We found that the effort required was non-negligible, for an unclear gain, so we focused on other aspects of the optimisations instead.

Our algorithm has then been adapted (compared to theirs) to take into account these two differences. We stop the trace if we encounter a node already visited to prevent traces from crossing each other. And then, instead of reasoning by execution count to decide the "best

[4] These would have to be manually reconstructed since each RTL node consists of a single instruction.

successor" or "best predecessor", we decide it based on the privileged nodes, as indicated by the branch-prediction information, stopping the trace when that information is `None`.

Superblock Selection for the Other Passes

The other phases are more straightforward since their goal is to identify superblocks, not traces. One issue here is the replication of the selection algorithms: we detect superblocks three times over each function (even if for loop unrolling, we only select superblocks that encompass innermost loops). Ideally, one would like to have some way to store the superblock information somewhere once and for all, and then the selection would just be about looking up that current information. In terms of data structure, this information could be a list of RTL nodes used for each superblock.

However, introducing and maintaining such information would require two things:

1. To be able to pass that information along with the successive passes. We would either need to use some global pointer in an OCaml module, used by our oracles — but that would break the Coq assumption that OCaml functions are pure. To overcome this, we could use the `Impure` library again, but that might add significant proof effort.[5] Alternatively, we could add other pieces of information to the existing RTL language, just like we did for storing branch information. However, we felt like it would be cumbersome to maintain for the existing passes, in particular, because of the second point below.
2. Updating such superblock information whenever any change is done to the code. Indeed, RTL transformations can potentially add new instructions, remove some others, or just change the whole structure of the program. Updating the branch-prediction information is easy enough, and we do that already.[6] However, updating a list of instructions used by each superblock would be another story: it could potentially add significant effort to each RTL transformation pass, regardless of any proof, as well as adding maintenance burden.

This led us to recompute the superblock selection once per superblock optimisation pass. On the one hand, this can potentially lead to inconsistencies: one pass optimising a superblock a certain way, while another pass optimises it in a completely different way because the superblocks are not the same. On the other hand, our selection algorithms all rely on making superblocks as big as possible (following the preferred direction every time, if there is one).[7] So my personal belief

[5] Though it might be trivial to do so, we never know until we do try it. To quote the famous idiom: "The devil is in the details".

[6] Say if a branch is inverted: just negate the boolean stored in the prediction. For more complex transformations, one could always set it to `None` for particularly nasty cases.

[7] See algorithm 2 for an example.

```
1  // returns (t[0] * t[1]) + (t[2] * t[3]) + ...
2  int sum_product (int *t, int n) {
3    int S = 0;
4    for (int i = 0; i < n; i += 2){
5      if (i == n-1)
6        S += t[i]; // only remaining element
7      else
8        S += t[i] * t[i+1];
9    }
10   return S;
11 }
```

Figure 4.3.

is that this has no impact on the overall quality of the produced code — and, when examining the code structure on our benchmarks, we did not witness such oddity.[8]

Consequently, each pass implements a variation of the following informal algorithm:

- We compute *join points*: those are nodes that are reached by more than one predecessor.
- We form superblocks starting from the unique *entrypoint*:
  - If the node has a *preferred successor*, and it is not a *join point*, we add it to the superblock and continue the formation through that successor.
  - Else, we stop the current superblock and start creating new superblocks from the exit nodes that were recorded while forming the former superblock by order of encounter.

This allows us to have (as far as we have observed) consistency between each pass regarding superblock selection.

4.2 Superblock Transformations before Scheduling

To illustrate the superblock transformations, let us analyse the C sample given by figure 4.3. In this piece of code, we perform the multiplication of each pair of consecutive elements, which we then sum together. If the array has an odd number of elements, the last element will be summed to the rest. One way to handle that is to get an `if..else` construct within the `for` loop, to take into account the edge case of the last iteration.

That example (disregarding any bundle) is translated into the following simplified assembly code in figure 4.4, after basic-block scheduling.[9] It is annotated with timing information.

[8]Except, of course, while debugging our introduced algorithms. Like any uncertified code, our oracles may be subject to implementation bugs.

[9]Note: the `Linearize` oracle of CompCert here chose to lay out the `ELSE` branch before the `IFSO`. It uses heuristics such as the code length to decide. More details can be found in section 8.3.

Indeed, on the Kalray architecture, assuming the instruction and data caches never miss, it is possible to compute by hand the number of cycles that a piece of code takes: each bundle is issued in one cycle, to which penalties may have to be added — either data-dependency penalties (e.g. using a register which has not been computed yet), or branch penalties (unconditional jumps give 1 cycle penalty, conditional branches give 2 cycles when the branch is taken). We annotated the assembly code with this timing information (assuming the `ELSE` branch is taken). An example of such penalties is on Line 22: the `maddw` instruction uses `$r5`, which needs the `lws.xs` to complete before being used; the `goto` induces another cycle penalty on top of that.

This code is inefficient in two ways.

First, because of the presence of the `ELSE` basic block, the code has to execute two unconditional jumps (`goto` instructions) to loop: at the end of the `ELSE` block and then at the end of the `LOOPINCR` block. This adds 1 cycle per loop iteration. On tiny loops such as this one, this represents 7% of the 14 cycles required to execute one iteration, which is not negligible.

Second, if we were able to move instructions atop conditional branches, then the `sxwd` and `addw` instructions from the `ELSE` basic block could be executed in the same bundle as `compw.eq` from the `SWITCH` basicblock. This would make us gain another cycle.

The first issue could be tackled by transforming the code so that the basic blocks (`LOOPCOND`, `SWITCH`, `ELSE` and `LOOPINCR`) end up as one monolithic (without any branch taken, except for going out) trace. This could be done by moving the `IFSO` basicblock right after the `LOOPINCR` basic block in the linearised code. The `IFSO` basic block would then need an additional `goto` instruction; however, this would allow us to remove the `goto` from the `ELSE` basic block.

Our trace would then be executable with just one `goto` branch taken. However, we would still be unable to use superblock scheduling to tackle the second issue: since `IFSO` has `LOOPINCR` as a successor, our trace has two different entrypoints.

To solve these issues, we use the following techniques:

- First, we use *tail duplication*, detailed in section 4.2.1, to transform our identified trace into a superblock.
- Second, if our superblock is an innermost loop, we apply *loop unrolling* to make it more fruitful for scheduling. This is detailed in section 4.2.2.
- Then, we perform *superblock scheduling* to re-order instructions across the given superblock, detailed in section 4.3.
- Finally, we augmented the Linearize heuristic used by CompCert, to ensure that each superblock is laid out contiguously in memory. This part is detailed in section 8.3.

Our tail-duplication and loop-unrolling techniques are inspired by what is described in Lee et al. [49].

```
.L100: /* [ LOOPCOND ] */
  compw.ge  $r32 = $r6, $r1
;; // T=1
  cb.wnez $r32? .L101
;; // T=2 /* [ SWITCH ] */
  addw  $r8 = $r1, -1
;; // T=3
  compw.eq  $r32 = $r6, $r8
;; // T=4
  cb.wnez $r32? .L102
;; // T=5 /* [ ELSE ] */
  sxwd  $r3 = $r6
  addw  $r10 = $r6, 1
;; // T=6
  lws.xs  $r3 = $r3[$r0]
  sxwd  $r7 = $r10
;; // T=7
  lws.xs  $r5 = $r7[$r0]
;; // T=8
  maddw $r2 = $r3, $r5
  goto  .L103
;; // T=9 + 2 (load) + 1 (goto)
.L102: /* [ IFSO ] */
  sxwd  $r4 = $r6
;;
  lws.xs  $r11 = $r4[$r0]
;;
  addw  $r2 = $r2, $r11
;;
.L103: // T=12 /* [ LOOPINCR ] */
  addw  $r6 = $r6, 2
  goto  .L100
;; // T = 13 + 1 (goto) = 14
.L101: /* [ EXIT ] */
```

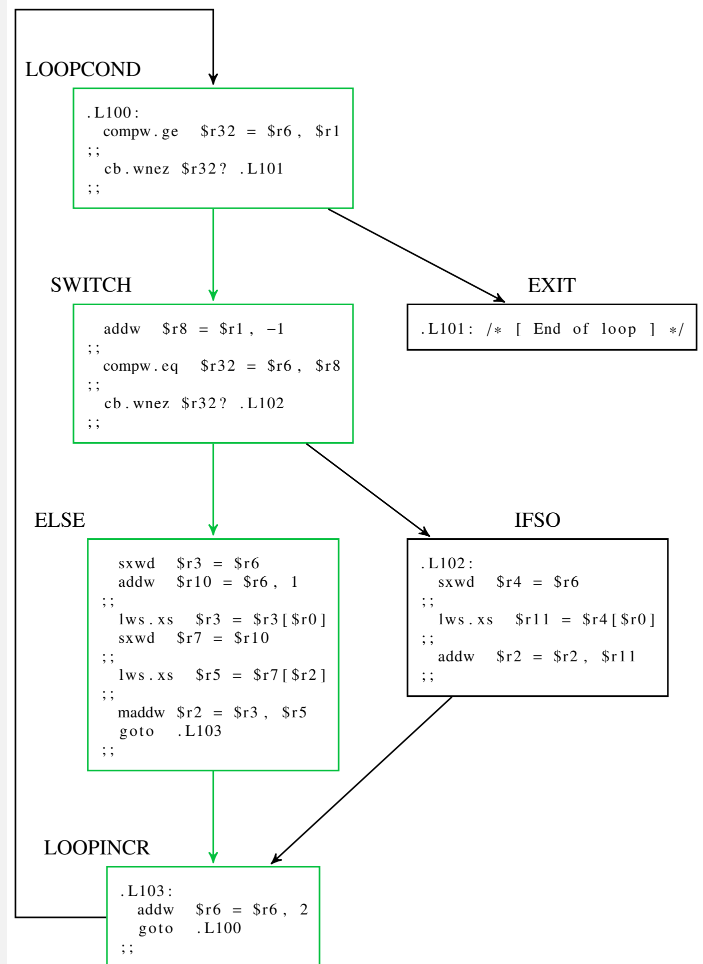


Figure 4.4.: On the left, the assembly generated code, annotated with timing information. On the right, its basic-block representation, with a trace of interest shown in green. In the assembly code, the registers targeted by load instructions along the trace are written in red. See appendix A for an instruction reference.

4.2.1 Tail Duplication

As it stands, the trace (`LOOPCOND`, `SWITCH`, `ELSE` and `LOOPINCR`) is not a superblock yet: indeed, `IFSO` points within the trace, so it has a second entrypoint. Given a selected trace, tail-duplication consists of identifying the first additional entrypoint and then duplicating the tail of the trace starting from the identified entrypoint. Anything that would point to the tail is then modified to point to the duplicated tail instead. This thus transforms the original trace into a superblock.

Figure 4.5 shows the result of performing tail-duplication on our selected trace. We show both the RTL representation (in which we perform the translation) and the resulting assembly code, annotated with timing information just like the previous.

We duplicate the `LOOPINCR` block of code and change `IFSO` to point to the new `LOOPINCR'` instead. Our trace is now a superblock: we will be able to perform superblock scheduling on it later on. We linearised the code in such a way that the resulting code of `ELSE` does not contain any `goto` instruction: our superblock is a contiguous block of code, which makes us win one cycle (we lose the latency from the `goto` that was present in the previous code). Furthermore, as a side effect of doing so, the `ELSE` and `LOOPINCR` blocks can now be fused into one basic block: compared to our previous code, our postpass scheduling gains one additional cycle by mixing their instructions together.

In this particular example, the result of performing tail-duplication (and smartly linearising the code) is then to gain 2 cycles out of the initial 14 required to compute one iteration.

4.2.2 Loop Unrolling

Loop unrolling consists of duplicating the body of a loop so that several iterations are run before looping back. We perform a specific kind of loop unrolling, which duplicates the conditional branch for going out of the loop. The result of such a transformation can be seen in figure 4.6.

This transformation is interesting because it preserves the superblock structure: the trace (`LOOPCOND`, `SWITCH`, `ELSE`, `LOOPINCR`, `LOOPCOND2`, `SWITCH2`, `ELSE2`, `LOOPINCR2`) is now a two-times-bigger superblock than the original. Without any prepass scheduling, this has two effects on the resulting code, in terms of performance:

- When executing two iterations, this removes one `goto`: we gain one cycle
- Laid out like this, the `CSE` optimization pass was able to determine that the `x13 = x1 + -1` instruction from `SWITCH2` is redundant: indeed, `x1` did not change. That instruction was then removed, which made us win another cycle.

```
.L100: /* [ LOOPCOND ] */
  compw.ge  $r32 = $r4, $r1
;; // T=1
  cb.wnez $r32? .L101
;; // T=2 /* [ SWITCH ] */
  addw  $r8 = $r1, -1
;; // T=3
  compw.eq  $r32 = $r4, $r8
;; // T=4
  cb.wnez $r32? .L102
;; // T=5 /* [ ELSE+LOOPINCR ] */
  sxwd  $r6 = $r4
  addw  $r7 = $r4, 1
  addw  $r4 = $r4, 2
;; // T=6
  lws.xs  $r10 = $r6[$r0]
  sxwd  $r5 = $r7
;; // T=7
  lws.xs  $r3 = $r5[$r0]
;; // T=8
  maddw $r2 = $r10, $r3
  goto  .L100
;; // T=9 + 2 (ld) + 1 (goto) = 12
.L102: /* [ IFSO+LOOPINCR' ] */
  sxwd  $r3 = $r4
  addw  $r4 = $r4, 2
;;
  lws.xs  $r11 = $r3[$r0]
;;
  addw  $r2 = $r2, $r11
  goto  .L100
;;
.L101: /* [ EXIT ] */
```

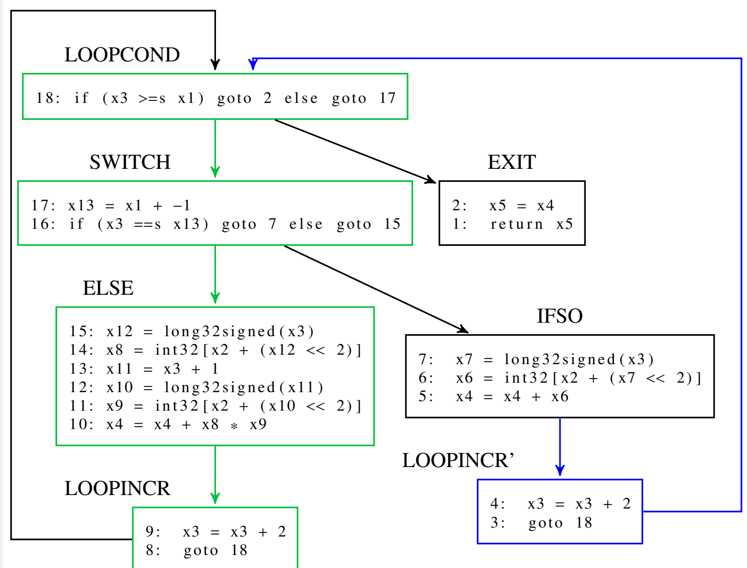


Figure 4.5.: Tail-duplicating the identified trace.

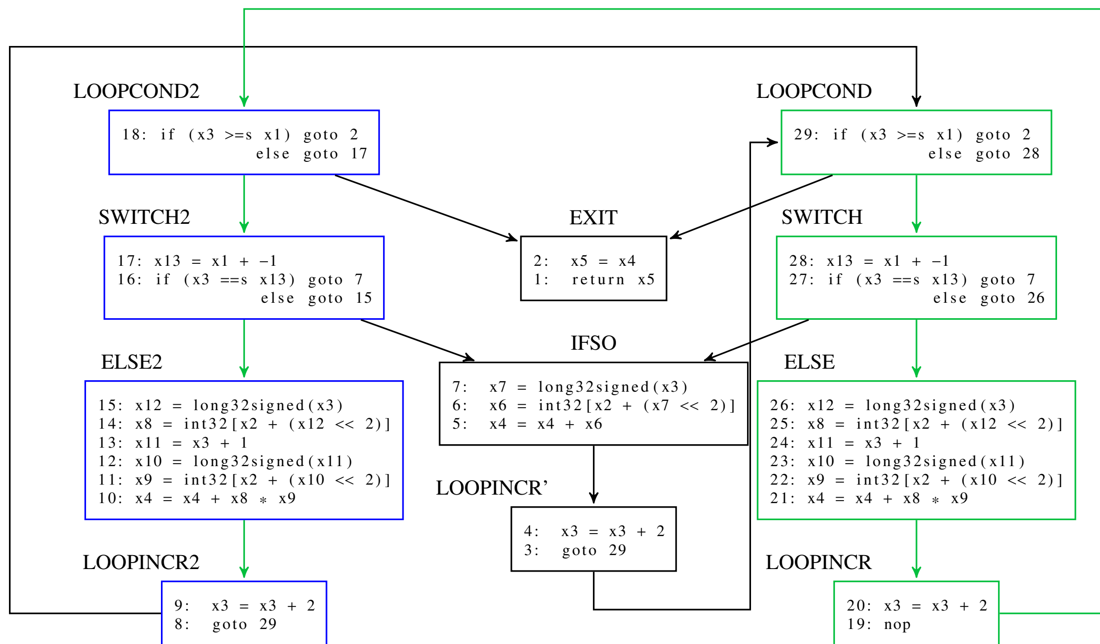


Figure 4.6.: Performing loop unrolling one time.

In total, after loop unrolling, executing two iterations is now 20 cycles. On average, an iteration is then 10 cycles (instead of 12 before loop unrolling).

4.3 Superblock Scheduling

Once a superblock has gone through tail-duplication and potentially unrolling, it is ready for being scheduled. Much like basic-block scheduling, the goal of superblock scheduling is to minimise execution time: here, minimising those of the paths that go through the entire superblock, *i.e.* taking no branch (all conditional branches from that superblock must result in their *fallthrough* cases).

Under such an assumption, it is possible to approximate the time it takes to execute a given superblock. Indeed, the only control-flow instructions allowed inside a superblock are conditional branches. They each supposedly go through their *fallthrough* cases, which induces no cycle penalty on the Kalray architecture.

In terms of scheduling oracle, there are two major differences compared to the postpass scheduling oracle:

- We manipulate RTL instructions, which are not exactly assembly instructions (unlike postpass scheduling). However, we can use the translation functions from the later passes (in particular, the translation from Mach to Asm) and combine it with the timing functions

```
 1  .L100: /* [ LOOPCOND ] */
 2    compw.ge  $r32 = $r7, $r1
 3  ;; // T=1
 4    cb.wnez $r32? .L101
 5  ;; // T=2 /* [ SWITCH ] */
 6    addw  $r8 = $r1, -1
 7  ;; // T=3
 8    compw.eq  $r32 = $r7, $r8
 9  ;; // T=4
10    cb.wnez $r32? .L102
11  ;; // T=5
12  /* [ELSE+LOOPINCR+LOOPCOND2] */
13    sxwd  $r33 = $r7
14    addw  $r4 = $r7, 1
15    addw  $r7 = $r7, 2
16  ;; // T=6
17    lws.xs  $r15 = $r33[$r0]
18    sxwd  $r5 = $r4
19    compw.ge  $r32 = $r7, $r1
20  ;; // T=7
21    lws.xs  $r17 = $r5[$r0]
22  ;; // T=8
23    maddw $r2 = $r15, $r17
24    cb.wnez $r32? .L101
25  ;; // T=9 + 2 (load)
26  /* [ SWITCH2 ] */
27    compw.eq  $r32 = $r7, $r8
28  ;; // T=12
29    cb.wnez $r32? .L102
30  ;; // T=13
31  /* [ ELSE2 + LOOPINCR2 ] */
32    sxwd  $r34 = $r7
33    addw  $r9 = $r7, 1
34    addw  $r7 = $r7, 2
35  ;; // T=14
36    lws.xs  $r11 = $r34[$r0]
37    sxwd  $r6 = $r9
38  ;; // T=15
39    lws.xs  $r3 = $r6[$r0]
40  ;; // T=16
41    maddw $r2 = $r11, $r3
42    goto  .L100
43  ;; // T=17 + 2 (ld) + 1 (goto) = 20
44  /* [ IFSO ] */
45  .L102:
46    sxwd  $r10 = $r7
47    addw  $r7 = $r7, 2
48  ;;
49    lws.xs  $r3 = $r10[$r0]
50  ;;
51    addw  $r2 = $r2, $r3
52    goto  .L100
53  ;;
54  .L101: /* [ EXIT ] */
```

Figure 4.7.: Final code after tail duplication and loop unrolling.

from the postpass scheduler oracle to make an educated guess at the RTL instruction latencies as well as their resource usage. This guess might be inexact: in particular, if the register allocator (RTL to LTL translation) introduces spills, then our scheduled time will be off. Further optimisations between LTL and Asm might also reduce the actual number of cycles for a given superblock, compared to our guess.

- Like the postpass scheduling, data dependencies between instructions have to be considered. In addition to that, some additional dependencies have to be inserted involving conditional branches: even though our oracle assumes (for the sake of computing timings) that each conditional branch is not taken, there is a possibility for a conditional branch to be taken, in which case the program should remain correct.

Once the dependencies and resource constraints are computed (the additional dependencies are described in 4.3.1), we can choose any scheduling oracle. Subsection 4.3.2 discusses informally two variants of list scheduling in regards to scheduling opportunities and register allocation. Then, I show the result of applying superblock scheduling to the example from section 4.2, in subsection 4.3.3.

4.3.1 Elements of Superblock Scheduling Correctness

Compared to basicblock scheduling, several new elements have to be checked to ensure correctness before deciding to execute an instruction speculatively. To illustrate that, figure 4.8 shows two examples of scheduling our tail-duplicated (but not unrolled, for simplicity) program from earlier.

The first schedule attempts to move all instructions from `ELSE` (conserving their own partial order) before the conditional branch `(x3 ==s x13)?`. We mark them in red. In terms of data dependencies among the superblock, this is a correct move: the conditional-branch instruction does not write to any register, so neither Write-After-Read nor Read-After-Write are possible. However, there is one issue: `x4` is written to in the `IFSO` branch! By moving the instruction `x4 = x4 + x8 * x9` before the conditional branch, we are actually introducing a new *Write-after-Write* dependency on the (`SWITCH, IFSO`) path that wasn't there before. This move is then incorrect.

The second schedule does the same, but only for instructions that do not introduce any additional *Read-after-Write* or *Write-after-Write* dependencies. In terms of data writes, the program is correct. There is still one additional issue: the RTL `int32[...]` instructions represent load instructions, and these may trap if their addresses are incorrect. In a normal execution (one without speculation), this might happen if the program has undefined behaviours — but then it would be a problem lying in the application source code rather than a compiler bug. However, in

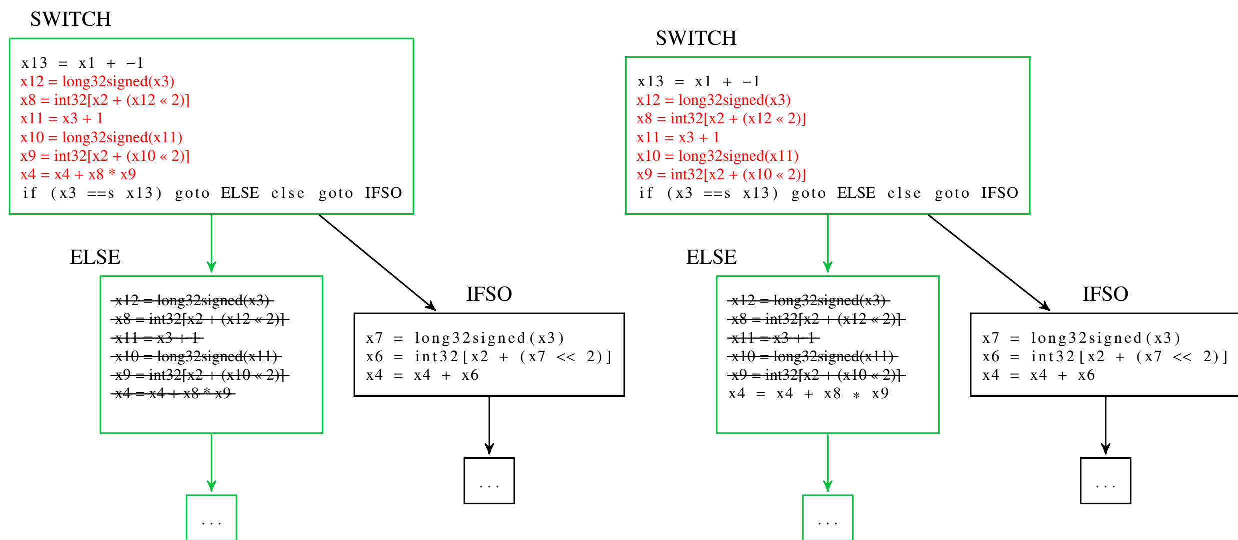


Figure 4.8.: Two scheduling examples — the moved instructions are in red. The left schedule is incorrect: an instruction assigning `x4` was moved before a branch which later uses `x4`. The right schedule is correct.

the case of speculative execution (moving an instruction before a branch), one might execute an instruction before a branch ensured that its operands would be valid. Such an example could be a C program containing "`if (ptr != NULL) x = ptr->foo;`": evaluating `ptr->foo` before the branch could trap if `ptr == NULL`, whereas the normal execution might not have trapped at all.

At the end of the day, in addition to the dependency constraints we saw for basic-block scheduling, the following has to be checked to ensure that a given superblock scheduling is correct:

- The moved instructions (either above or below a side exit) should not change the result of any instruction in the "exit" part of the conditional branch. This can be checked with a liveness analysis: the live registers at the time of branching should not be modified by the reordering.
- No trapping instruction should be moved before a conditional branch.[10] To remedy that, certain architectures like the KVX have support for *speculative* instructions which do not trap when their operands are invalid. Since the KVX supports *speculative* loads, we turn loads into speculative loads when they move before a conditional branch.

Certifying the superblock scheduling turned out to be challenging in terms of proof effort (contrary to tail-duplication and loop unrolling, which were easy to prove). The later chapters will describe the proofs involved, from designing an `IR` to execute superblocks to certifying the actual scheduling.

[10]More generally, moving an instruction before a side exit should not add any additional undefined behaviour.

```
x5 = int32[x1 + 0]
x6 = int32[x1 + 4]
x3 = x5 + x6
x4 = int32[x1 + 8]
x2 = x3 + x4
```

```
lws   $r1 = 0[$r0]
lws   $r2 = 4[$r0]
addw  $r1 = $r1, $r2
lws   $r2 = 8[$r0]
addw  $r1 = $r1, $r2
```

Figure 4.9.: Example illustrating the impact of register allocation on scheduling. Left: RTL code example; it is possible to move instruction 4 before instruction 3 to reduce stalls. Right: Assembly (before scheduling) example; that move is not possible because of a false dependency with `$r2`.

4.3.2 Choosing an Appropriate Scheduling Oracle

Doing superblock scheduling before register allocation has two advantages. First, as mentioned earlier, we profit from the more extensive scheduling scope (superblock instead of basic block). However, since the registers are not allocated yet, there are no false dependencies in the code, unlike postpass scheduling. Figure 4.9 shows a simple RTL code computing `t[0] + t[1] + t[2]`, before register allocation. One possible outcome of assembly generation (after register allocation and before postpass scheduling) is shown on the right. To prevent a stall in the `x2 = x3 + x4` assignment, a valid scheduling could be to move the load assigning `x4` before `x3 = x5 + x6`. However, such scheduling is not possible after register allocation because the register allocator was pressured to reuse the `$r2` register instead of introducing a new register `$r3`.[11] Prepass scheduling does not have this issue.

The downside of scheduling before register allocation is that it may increase *register pressure* — that is, the number of registers that are *live* at a given time. Going back to the example in figure 4.9: in the unscheduled version, only `x5` and `x6` have to be live during the first addition. Moving the load assigning `x4` before the addition would add `x4` to the list of live registers, thus increasing the register pressure. When there is too much register pressure, the register allocator has no choice but to insert *spills*: saving a register to the stack, then reloading it later. In which case, prepass scheduling can worsen performance instead of improving speed.

The general problem of easing the interactions between the scheduler and register allocation is a tough one. There has been work to merge both optimisations into one, like Motwani et al. [65] and more recently Lozano et al. [57]. This is far beyond our scope: implementing such solutions would require rewriting the register allocator of CompCert and its proof and merging scheduling within: this would be a titanic task.

[11] On this particular example, the CompCert register allocator would have chosen to allocate a new register. However, in large functions, it is common for the register allocator to reuse registers. Whether it reuses the "right" register (one that would not introduce false dependency) or not is then a matter of luck.

One possible solution for not having too much pressure on the register allocation is to do *backward list scheduling*, as described in Cooper et al. [23]. The core idea is to reverse all edges from the dependency graph and schedule the *finish times* instead of the *starting times*. Whereas forward list scheduling tends to cluster operations to the beginning of the scheduled block, backward scheduling tends to cluster operations to the bottom of it, which should ease the register pressure.[12]

The code presented in this chapter uses this variant. Though we have found in our benchmarks that applying regular forward scheduling gives better performance on average, which can be partly explained by the KVX architecture having many available registers, we rarely suffer the effects of increased register pressure. An alternative would be to make the scheduler sensitive to register pressure [36; 87]; this is left to future work.

4.3.3 Application to our Example

The optimisations that we presented before scheduling were able to shave a few cycles off our example. However, that was just a side effect — their real goal was to increase the superblock's scheduling potential. At last, let us apply superblock scheduling to our example. Figure 4.10 shows the new processor code obtained after activation of the prepass scheduler. We can notice significant progress compared to the code that was presented in figure 4.7.

- Many instructions were able to be moved before conditional branches, which allowed to "fill the gaps" in the bundles. If we take the first line from the code after unrolling, there was a single instruction `compw.ge` in the bundle. In the code with superblock scheduling, that instruction is executed alongside `sxwd` and `addw`. Having such a denser packing contributes to having fewer bundles executed: for the identified trace, we went from 15 bundles (unrolled version) down to 13. *A priori*, this can make us save 2 cycles.
- The most significant result is the lack of any stall, thanks to the use of speculative loads. All the load instructions were able to be scheduled in such a way as to not induce any stall. This makes us save 4 cycles compared to the unrolled version.

In the end, it takes 14 cycles to execute our new code, so 7 cycles per iteration. Compared to the unrolled version of 10 cycles per iteration, that is 30% fewer cycles. Compared to the original version (without any of the optimisations presented in this chapter: no tail-duplication, no loop unrolling, no superblock scheduling), which was at 14 cycles, that is 50% fewer cycles. This results in twice faster execution, on the privileged path, for this particular example.

[12] A variant of this technique is used in Sarkar et al. [83] to decrease register pressure.

```
1  .L100: /* [ SUPERBLOCK ] */
2    sxwd  $r9 = $r34
3    addw  $r6 = $r34, 1
4    compw.ge  $r32 = $r34, $r1
5  ;; // T=1
6    lws.s.xs  $r7 = $r9[$r0]
7    sxwd  $r8 = $r6
8  ;; // T=2
9    cb.wnez $r32? .L101
10 ;; // T=3
11   addw  $r4 = $r1, -1
12   lws.s.xs  $r11 = $r8[$r0]
13 ;; // T=4
14   compw.eq  $r32 = $r34, $r4
15 ;; // T=5
16   cb.wnez $r32? .L102
17 ;; // T=6
18   addw  $r34 = $r34, 2
19   maddw $r2 = $r7, $r11
20 ;; // T=7
21   sxwd  $r5 = $r34
22   addw  $r33 = $r34, 1
23   compw.ge  $r32 = $r34, $r1
24 ;; // T=8
25   lws.s.xs  $r3 = $r5[$r0]
26   sxwd  $r5 = $r33
27 ;; // T=9
28   lws.s.xs  $r15 = $r5[$r0]
29   cb.wnez $r32? .L101
30 ;; // T=10
31   compw.eq  $r32 = $r34, $r4
32 ;; // T=11
33   cb.wnez $r32? .L102
34 ;; // T=12
35   maddw $r2 = $r3, $r15
36   addw  $r34 = $r34, 2
37   goto  .L100
38 ;; // T=13 + 1 (goto)
39 /* [ IFSO ] */
40 .L102:
41   sxwd  $r17 = $r34
42   addw  $r34 = $r34, 2
43 ;;
44   lws.xs  $r10 = $r17[$r0]
45 ;;
46   addw  $r2 = $r2, $r10
47   goto  .L100
48 ;;
49 .L101: /* [ EXIT ] */
```

Figure 4.10.: Our example, with prepass scheduling activated. It is annotated with timing information.

4.4 Superblock Transformations and RTL

Each code optimisation is supposed to, on its own, induce an improvement in the resulting program. Still, a specific order has to be chosen for each code optimisation to be run sequentially.

In the case of superblock transformations, we already have a straightforward partial order. First, branch prediction (either through profiling or heuristics) should annotate the conditional branches since all the other superblock passes rely on that. Tail-duplication allows turning the body of specific innermost loops into superblocks, making it logical to do it after branch prediction and before the other superblock transformations. The same goes for loop unrolling, except that by applying tail-duplication before, we have more chance to find innermost loops eligible for unrolling (they must follow a superblock structure for the unrolling to do any benefit). Finally, superblock scheduling should be the last of all superblock transformations since the former transformations aim at increasing the scheduling potential.

However, it is a bit more unclear how these passes should fit into existing (and future) RTL transformations of CompCert. The superblock transformations might result in better optimisation opportunities for later passes — on the other hand, optimisations might result in better trace selection or tail-duplication. In subsection 4.4.1, we give an overview of each RTL optimisation

and discuss its impact on superblock transformation passes. Then, in subsection 4.4.2, we lay out the actual used order.

4.4.1 Existing RTL Transformations in regard to Superblock Transformations

Let us analyse each RTL transformation and discuss how it affects our superblocks passes — or perhaps, how our superblock passes may affect it.

Tailcall Generation

This optimisation detects functions that end with calling another function. It allows for the called function to reuse the stack frame of the (tail) caller instead of creating its own new stackframe, thus reducing stack memory consumption. In practice, this optimization just transforms certain `Icall` instructions into `Itailcall`.

This has no impact on any of our superblock transformations. Furthermore, our superblock transformations would not have any positive impact on this transformation. So it is safer for this transformation to be placed before our superblock optimisations.

Function Inlining

Function inlining consists of identifying small functions and replacing some calls to such functions by their entire code.

Our superblock transformations can harm this transformation since we might make functions bigger by tail-duplication or loop unrolling, making them uneligible for inlining. Also, this transformation can help a great deal with our transformations by replacing calls, thus extending the number of instructions to schedule. A natural choice is then to perform this optimisation before ours.

Common Subexpression Elimination

This optimisation aims to detect expressions that are computed more than once: the additional computations are then replaced by the appropriate register containing the result of the computation. Informally, code such as `x0 = x3 + x4; x1 = x2 * (x3 + x4)` would become

`x0 = x3 + x4; x1 = x2 * x0`. In our version of CompCert, we have two variants of this optimization:

- The original `CSE` optimization, from mainstream CompCert. That optimisation performs the above optimisation on *extended basic blocks*: groups of basic blocks with only one predecessor each (except the first). In practice, this optimisation stops at *join points*. In particular, it does not traverse loops (since a loop header has two predecessors: the node dominating the loop and the last node of the loop).
- A new `CSE3` optimization from D. Monniaux [63], which propagates the analysis across *join points*.

In our compilation flow we perform two common subexpression eliminations: one before static branch prediction, since it might have an influence in the prediction heuristics, and both the CSE and CSE3 versions after the instruction-duplication passes.

CSE combined with loop unrolling can increase performance from naturally replacing loop-invariant expressions by their assigned registers (see the example of section 4.2.2). As an extension, CSE3 combined with loop peeling[13] achieves Loop-Invariant Code Motion [63].

Constant Propagation

Constant propagation consists of pre-computing the operations whose results can be known at compile time. For example, `x0 = 42; x1 = x0 + 3` could be turned into `x0 = 42; x1 = 45`. It might probably help static branch prediction if constant propagation is done before. Like CSE, a good fit would probably be to do it before and after the instruction duplication passes. In our experiments, we do it only once, after instruction duplication.

Deadcode Elimination

Dead-code elimination consists of eliminating the computations that are, in fact, never used. So, in a program such as `x0 = x4 * x5; return x2`, the `x0 = x4 * x5` could be removed, since that result is discarded when returning `x2`. It should not matter whether we perform this optimisation before or after ours — we do it among the last optimizations, just before superblock scheduling.

[13] Loop peeling copies one iteration out of the loop body, see section 8.2.

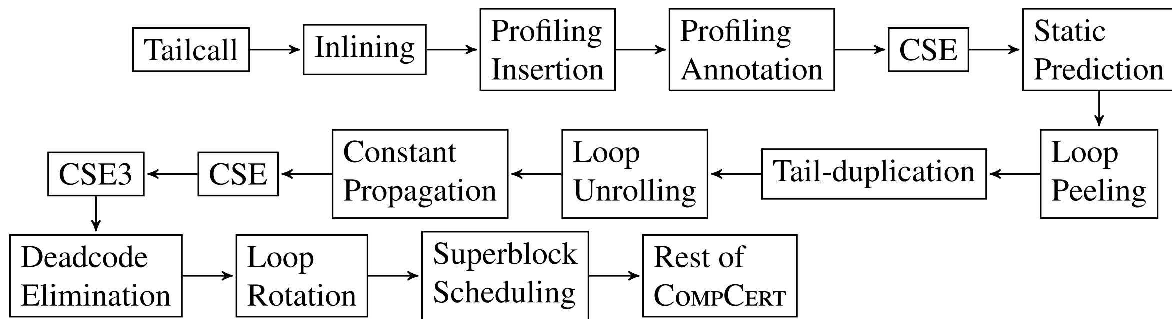


Figure 4.11.: Our order of RTL optimizations

4.4.2 Order of Transformations

Figure 4.11 shows the order of each RTL transformation, including those that we introduced:

1. Tailcall optimization
2. Inlining, before our superblock optimizations. Indeed, our instruction-duplication oracles make the decision of duplicating based on the number of instructions that would be duplicated. Inlining functions after that decision is made could invalidate it, because of the increase in instruction count.
3. Profiling insertion, then, annotating branches based on profiling information if they exist.
4. The CSE pass just before static prediction, to possibly improve the branch-prediction heuristics.
5. The static branch-prediction pass, which annotates the branches that were not annotated from profiling.
6. Loop peeling: duplicating one iteration of loops, with the prospect that the loop-invariant expressions will be later eliminated with CSE3.
7. Tail-duplication, to transform relevant traces into superblocks.
8. Loop Unrolling, to increase scheduling opportunities and also reduce branching penalties due to looping (see section 10.3.6 for more details on the impact of reducing branch penalties).
9. Constant Propagation and CSE for eliminating potential redundancies introduced by instruction duplication.
10. CSE3, for simplifying the loop-invariant expressions within the loop body, which were copied out of the loop via loop peeling. This achieves loop-invariant code motion [63].
11. Redundancy elimination
12. Loop rotation, to improve the later postpass scheduling.

5 Introducing Superblock Execution Semantics with RTLpath

Chapter 4 introduced what superblock scheduling is, how we generate and modify superblocks, and the impact of these transformations on performance. We touched briefly on some elements of correctness for superblock scheduling, though we did not dive into any formalism. In particular, we saw that load instructions must be turned into *speculative loads* when they are scheduled ahead of a side-exit.

In this chapter, we will study the first step in certifying superblock scheduling: introducing a semantics for executing a program with steps covering whole superblocks. I first introduce the RTL representation in more detail, along with our modifications to model *speculative loads* in CompCert, as well as keeping track of branch prediction throughout RTL (section 5.1). Then we define a new IR based on RTL, named RTLpath, which acts as an anchor point to perform the prepass scheduling (from RTLpath to RTLpath) discussed in the later chapter 7. After describing its formal semantics, I detail a bisimulation between RTL and RTLpath, which allows us to go back and forth between RTL and RTLpath (section 5.2). Finally, I detail its code generation, which involves selecting the superblocks and computing their liveness information by an oracle, then verifying the oracle result for correctness (section 5.3).

The work of designing RTLpath has been mostly done by S. Boulmé, with D. Monniaux implementing support for *speculative loads* in RTL, and me filling the gaps: finishing the RTL to RTLpath proof stub, supporting *speculative loads* in RTLpath, and then implementing the superblock selection (the `RTLpathLivegen` oracle, described later).

5.1 Modifying CompCert's RTL Representation

RTL (Register Transfer Language) is the IR right after CminorSel, featuring 3-address code. Its control-flow graph structure, combined with its unbounded number of registers, makes it a

```
1  int fold_add(int *t, int length){
2    int S = 0;
3    for (int i = 0; i < length; i++)
4      S += t[i];
5    return S;
6  }
```

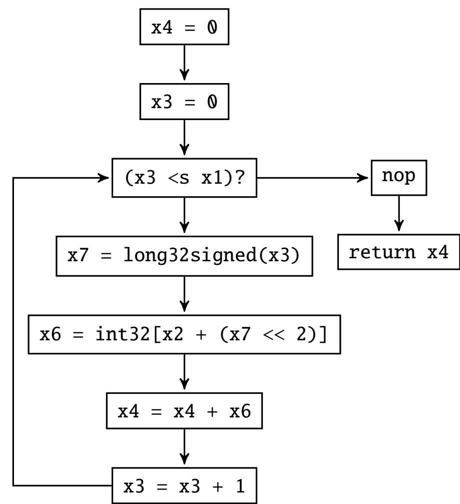


Figure 5.1.: C code, and RTL representation of `fold_add` as a control-flow graph

natural choice to implement optimizations such as constant propagation, dead-code elimination, and common subexpression elimination.

5.1.1 Overview of RTL

Much like the other IRs of CompCert, a RTL program is composed of several functions. In RTL, each instruction is identified by a positive integer, called a *node*. Each RTL instruction has its successor nodes encoded within the syntax. In order to retrieve the instruction referenced by a node, there is a mapping from the nodes to the (syntactic) instructions. This mapping is called an RTL *code*. In addition to that, the *entrypoint* is the first node of the function to be executed.

A RTL function is then made of: a signature, a list of parameters (input registers), a stack size, some code, and an entrypoint.

Control-Flow Graph Structure

RTL implements a control-flow graph structure, where each node is made of one single instruction, and there is an edge from i_1 to i_2 if the positive integer identifying i_2 is among the successors of i_1.

Figure 5.1 shows an RTL function code with a single loop, represented as such a graph.

```
9 ↦ Iop      (Ointconst 0) [] x4 8
8 ↦ Iop      (Ointconst 0) [] x3 7
7 ↦ Icond    (Ccomp Clt [x3; x1]) 6 4
6 ↦ Iop      Ocast32signed [x3] x7 5
5 ↦ Iload    Mint32 (Aindexed2XS 2) [x2; x7] x6 4
4 ↦ Iop      Oadd [x4; x6] x4 3
3 ↦ Iop      (Oaddimm 1) [x3] x3 7
2 ↦ Inop     1
1 ↦ Ireturn x4
```

Figure 5.2.: The RTL code of `fold_add`, represented syntactically. The instruction nodes are represented in red. The entrypoint of that code is 9. For clarity, I represent registers as "x" followed by their numbers – though in the actual syntax, registers are just positive integers.

Syntax

The RTL representation is composed of 10 instructions:

- `Inop pc'`: this is a no-operation instruction. It can be generated by some optimizations like common subexpression elimination.[1] Branches to the instruction indicated by `pc'`.
- `Iop op lr r pc'`: performs the operation `op` on the list of source registers `lr`, writes the result in the destination register `r`, branches to `pc'`.
- `Iload chunk addr lr r pc'`: loads a `chunk` quantity, from the address computed by the addressing mode `addr` and a list of registers `lr`, writes to the destination register `r`, branches to `pc'`.
- `Istore chunk addr lr r pc'`: same as `Iload`, but stores the register `r` to memory instead of loading it.
- `Icall sig ros lr r pc'`: calls a function of signature `sig`, pointed to `ros` which represents either a register or a symbol. The list of arguments is `lr`, the destination register for the returned value is `r`. Finally, after the call, branches to `pc'`.
- `Itailcall sig ros lr`: same as `Icall`, but for a *tailcall*.
- `Ibuiltin ef args dest pc'`: calls the external function `ef`, with arguments `args`. The return value is stored in `dest`, then it branches to `pc'`.
- `Icond cond lr ifso ifnot`: evaluates the condition `cond` on the values of `lr`. If the condition evaluation is true, branches to `ifso`. Otherwise, it branches to `ifnot`.
- `Ijumptable r lpc`: selects a value of the list `lpc` based on the register value `r`, then branches to this selected value.
- `Ireturn or`: ends the execution of the function, returning the value given by the optional register `or`. If `or` is `None`, then it returns an undefined value `Vundef`.[2]

[1] In terms of formal correctness, it is much easier to replace a useless expression by an `Inop` rather than removing an entire node from the control-flow graph. The `Inop` can then be removed by a dedicated pass.
[2] This is the case for a `void` function.

Figure 5.2 shows the same example as 5.1, but not pretty-printed (the actual syntax of each instruction is shown).

Compared to an Asm language, RTL has much fewer instructions. Much like Mach, it abstracts most of the architecture-specifics parts, which allows for simpler proofs (fewer cases to handle).

Semantics

The semantics of RTL are a lot like Mach semantics, but for two major differences. First, RTL fetches the next instruction differently than Mach: the PC value (equal to the next instruction node) is stored within a `State`, and remains internal to a function execution. Second, there are no variables stored on the stack.[3] Each variable lies in a register. Since there can be infinitely many registers, it is not possible to run out of registers: RTL optimizations are free to introduce any amount of fresh registers.

A RTL *state* is either:

- `State stack f sp pc rs m`: we are inside the function `f`, the call stack is `stack`, the stack pointer is `sp`,[4] and we are about to execute the instruction whose node is `pc` with register state `rs` and memory state `m`.
- `Callstate stack f args m`: we are about to call `f` with arguments `args`, the call stack is `stack`, and the memory state `m`.
- `Returnstate stack v m`: the call stack is `stack`, the memory state is `m`, and we are returning the value `v` from a function.

Figure 5.3 gives an overview of the *small-step* relation defining the execution of a given RTL instruction.

Like Mach, the failure of an instruction execution (such as a division by zero) is encoded by an absence of transition in the *small-step* relation. Listing 5.1 gives the Coq code defining the execution of `Iop` and `Iload`. For `Iop`, if the operation evaluations fails, then `eval_operation` will return `None`, thus the predicate required by the constructor `exec_Iop` will not be fullfilled. As for `Iload`, this instruction can fail in two distinct parts: the computing of the address might fail (if `eval_addressing` returns `None`), or the computed address might lie in an invalid memory region (if `Mem.loadv` returns `None`).

[3] The stack size, which is part of a RTL function, is only there to be used by the later passes.
[4] Here, this is not the same stack pointer as in Asm. This is just the identifier of the memory block (the stack) allocated for the function.

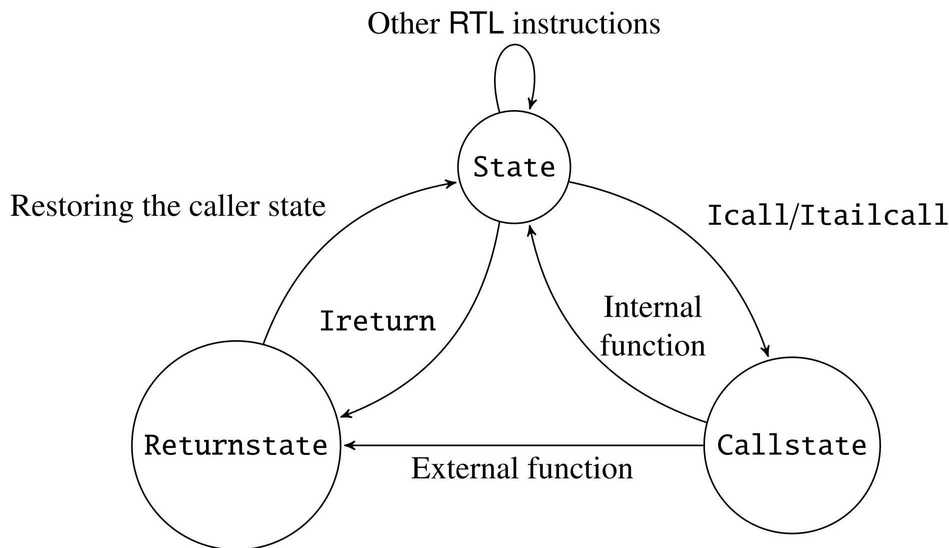


Figure 5.3.: Execution steps between RTL states

```
1  Inductive step: state → trace → state → Prop :=
2    (* ... *)
3    | exec_Iload:
4        ∀ s f sp pc rs m chunk addr args dst pc' a v,
5        (fn_code f)!pc = Some(Iload chunk addr args dst pc') →
6        eval_addressing ge sp addr rs##args = Some a →
7        Mem.loadv chunk m a = Some v →
8        step (State s f sp pc rs m)
9          E0 (State s f sp pc' (rs#dst ← v) m)
10   | exec_Iop:
11       ∀ s f sp pc rs m op args res pc' v,
12       (fn_code f)!pc = Some(Iop op args res pc') →
13       eval_operation ge sp op rs##args m = Some v →
14       step (State s f sp pc rs m)
15         E0 (State s f sp pc' (rs#res ← v) m)
```

Listing 5.1.: Execution of `Iop` and `Iload` in RTL

5.1.2 Modeling Speculative Loads in RTL

As was mentioned in chapter 4, some architectures like the Kalray core support *speculative loads* — that is, load instructions that do not fail in the case of invalid addressing.[5]

[5] There exist older architectures with more general *speculative execution* for most of the integer operations, such as in Lee et al. [49]. Speculative execution as described in that paper is only relevant for in-order architectures supporting those, since modern out-of-order architectures support speculative memory accesses right away, without the need to change the instruction into a speculative variant.

Modeling such instructions require us to add a new field to the syntax of `Iload`, to be able to distinguish between the speculative and non-speculative variants of the instruction. We thus included a `trap` field in the `Iload` syntax (listing 5.2).

```
Inductive trapping_mode : Type := TRAP | NOTRAP
Inductive instruction: Type :=
  (* ... *)
  | Iload: trapping_mode → memory_chunk → addressing
      → list reg → reg → node → instruction
```

Listing 5.2.: Introducing trapping modes in `Iload`

Then, we added semantics to the execution of a non-trapping load with two new relations: `exec_Iload_notrap1` and `exec_Iload_notrap2` (listing 5.3).

```
Inductive step: state → trace → state → Prop :=
  (* ... *)
  | exec_Iload_notrap1:
      ∀ s f sp pc rs m chunk addr args dst pc',
      (fn_code f)!pc = Some(Iload NOTRAP chunk addr args dst pc') →
      eval_addressing ge sp addr rs##args = None →
      step (State s f sp pc rs m)
        E0 (State s f sp pc' (rs#dst ← Vundef) m)
  | exec_Iload_notrap2:
      ∀ s f sp pc rs m chunk addr args dst pc' a,
      (fn_code f)!pc = Some(Iload NOTRAP chunk addr args dst pc') →
      eval_addressing ge sp addr rs##args = Some a →
      Mem.loadv chunk m a = None→
      step (State s f sp pc rs m)
        E0 (State s f sp pc' (rs#dst ← Vundef) m)
```

Listing 5.3.: Trapping load semantics in RTL

Informally, if `eval_addressing` evaluates to `None`, then we write a `Vundef` value in the destination register. This is different from the concrete processor execution, which returns 0 when a speculative load reaches an invalid address. However, replacing this `Vundef` value by 0 would be an incorrect specification since a load instruction in RTL might fail more than its concrete counterpart on the processor. A typical example would be an out-of-bounds access on the stack. Since the function stacks are not contiguous in the memory model of CompCert, such a load instruction would fail in RTL (the load would return 0), whereas on the concrete processor, it would succeed and return the value in the given address. Using `Vundef` instead of 0 ensures that the result will never be read in the remaining of the execution (in CompCert semantics, an instruction reading `Vundef` automatically fails).

We do the same if `eval_addressing` returned a value, but `Mem.loadv` returned `None`. In practice, though, only the latter case is possible, because it is guaranteed within the Mach to

Asm code generation that `eval_addressing` never fails (but we must still take it into account in our semantics).

Correctness of Making an `Iload` Speculative

Compared to a non-speculative load, the only change in the semantics is in the case of failure: a speculative load does not fail, while a regular load does. Since CompCert proof of correctness only holds for programs that do not fail, we can deduce that transforming a non-speculative `Iload` into a speculative `Iload` is correct: nothing changes in terms of semantics (if the `step` relation from the code before transformation was constructed with an `exec_Iload`, then we can apply this same `exec_Iload` on the speculative version of the `Iload`).

It is thus possible to implement a dedicated pass like listing 5.4, which would make each `Iload` speculative, and prove it correct by a lockstep simulation.

```
1 Definition transf_instr (pc: node) (instr: instruction) :=
2   match instr with
3   | Iload trap chunk addr args dst s ⇒ Iload NOTRAP chunk addr args dst s
4   | _ ⇒ instr
5   end
```

Listing 5.4.: Turning each `Iload` into a non-trapping version

However, in terms of application development, although correct, this might not be desirable. Application-level bugs would indeed be much harder to find, since any load could potentially fail silently, producing wrong values instead of signaling a trap to the underlying operating system.

In practice, in our superblock scheduling (described in chapter 7) we only make our loads speculative when necessary — that is, when they move ahead of a side-exit, as described informally in section 4.3.1.

5.1.3 Modeling Branch Prediction

Branch-prediction information is used by the passes that we introduced in chapter 4 and by our modified version of `Linearize`, which uses this information to retrieve the superblock structure and emit a `Linear` code in which each identified superblock is contiguous in memory.

This branch prediction is stored through an additional blank field in each `Icond` instruction (listing 5.5).

```
Inductive instruction: Type :=
  (* ... *)
  | Icond: condition → list reg → node → node
      → option bool → instruction
```

Listing 5.5.: Definition of `Icond` in our version of RTL

This new field is ignored by all the certified passes. It is only used by the external uncertified oracles to guide their heuristics. For these oracles, this `option bool` field has the following meaning based on its value:

- `None` indicates a lack of prediction for the given `Icond`. Either the prediction pass has not yet been performed, or the heuristics were unable to devise a preferred branch.
- `Some true` indicates that the *ifso* branch is privileged. According to the heuristics, the program should spend more time in *ifso* than in *ifnot*.
- Vice-versa, `Some false` indicates that the *ifnot* branch is privileged.

Since we need to preserve this information until `Linear`, we did the same change for LTL.

This choice is motivated by the previous section 4.1.2 — there are other possible alternatives. In the future, changing that data format would only imply little effort on modifying the underlying proofs. Most of the effort would be on adapting the heuristics to cope with the change.

5.2 Representing Superblocks as Execution Paths in RTLpath

To reason about superblock scheduling, we first need to provide semantics for executing superblocks. This is, in a way, similar to the work we did in chapter 3 for the Machblock representation: we would like an RTL-like IR within which we can execute whole superblocks. However, compared to Mach, we have several differences to cope with:

- Mach code is linear: executing Mach code is simply executing a list of instructions and possibly re-loading the instruction list by jumping to `Mlabel` instructions. RTL code, on the other hand, follows a control-flow graph structure in which the next instruction to execute must be computed from the current instruction execution.
- In Machblock we were very interested in keeping that basic-block structure, to translate it to Asmblock (assembly code generation), and then to AsmVLIW (scheduling + formation of bundles). For our prepass optimization, we need to output a RTL program (empty of any superblock structure) for the later register-allocation pass to happen. This implies that we would rather make the transition back to RTL as painless as possible.

- We intend to give semantics for a structure more complex than a basic block. In particular, for the later certification of superblock scheduling, we require to access liveness information for each side exit of a superblock.

We present in this section RTLpath, an IR dedicated to outlining the superblock structure from the input RTL program. This IR acts as a supplementary semantics augmenting the RTL program — translating a RTL program to RTLpath does not involve any change within the RTL syntax, so the RTL program (and semantics) can be retrieved very easily from RTLpath.

5.2.1 RTLpath Syntax and Informal Semantics

Schedulable and Non-schedulable Instructions

In Machblock, we split the Mach instructions into two groups: those that alter the control flow and those that do not. For RTLpath, we also split the RTL instructions in two groups:

- Those that have preferred successors, and which we would be able to reorder. `Inop`, `Iop`, `Iload`, `Istore` belong naturally to that group: each has only one possible successor and does not generate any trace.
- The rest, who will always be sitting at the ends of superblocks: `Icall`, `Itailcall`, `Ibuiltin`, `Ireturn`, `Ijumptable`.

The case of `Icond` requires a bit of discussion. We are able to devise a preferred successor on `Icond` in most cases, but not always. If we lack branch prediction, or if the branch prediction fails to deliver a verdict, then there is no way to tell which `Icond` branch to prefer. We could be looking at the `option bool` field of `Icond` and decide based on the value of that field whether a given `Icond` belongs inside a superblock (is schedulable) or should remain at the end. However, that would make the semantics and proofs of RTLpath harder. So we chose that `Icond` always has one privileged successor, the fallthrough case (`ifnot` field).[6] To ensure that this is actually the case, we do two things:

- Before RTLpath, during the *branch-prediction* RTL pass, we invert the conditions of the `Icond`s that would have their `ifso` branches as privileged. This makes them privilege `ifnot` instead.
- In order to respect the `None` prediction (no branch is preferred), we will rely on the RTLpath and scheduler oracles to ensure these `Icond`s are placed at the ends of superblocks, and remain that way.

[6]This choice is completely arbitrary. One could argue that choosing `ifnot` instead of `ifso` results in better performance, since executing the `fallthrough` case of a conditional branch induces 0 penalty (compared to 2 cycles otherwise) — but such issue would be corrected by the later `Linearize` pass anyway.

As for those that we excluded: since `Itailcall` and `Ireturn` finish the execution of a function, it would break the semantic to reorder them. `Icall` and `Ibuiltin` only have one successor each, but since their semantics involve going out of the current function, it would be complex to include it in our reordering, for unclear benefits. Finally, `Ijumptable` is more complex than `Icond` to handle, since there are more than one possible side-exit. Also, we found its presence to be very rare in our benchmarks, the associated tail-duplication could involve duplicating much more code than if it was an `Icond`, and `Ijumptables` are always translated into unconditional jumps (1 cycle penalty). The benefit of including it in our scheduling is unclear and would require potentially more cumbersome proofs, so we decided not to include it.

Paths and Pathmaps

The notion of *path* that we will present here is very close to the notion of *trace* in the compiler litterature; but since CompCert already has a notion of trace completely unrelated[7] , to prevent confusion, we name them *paths*. An *execution path* is a group of instructions with the following semantics:

- Each path has an entry node, from which the path execution starts.
- During the path execution, it is possible to step out of the path if an *early exit* instruction is met. Here, the only early exit instructions are `Icond`s.
- The number of instructions to execute for a path is bounded and annotated on the entry node. If no early exit is triggered, the path execution stops as soon as this number is reached.

The information of the maximum number of instructions to execute, along with liveness information, is stored in a `path_info` data structure.[8] We define a *pathmap* as a partial mapping from RTL nodes to such data structure: path entries are those for which the mapping has a `path_info`. Listing 5.6 shows our code listing defining these notions. One tiny detail should be noted: we took the convention to keep track of the "number of instructions to execute before the last instruction", so a value `psize=0` indicates there is only one instruction to execute in the path.

```
1  Record path_info := {
2      psize: nat; (* number minus 1 of instructions in the path *)
3      input_regs: Regset.t;
4      output_regs: Regset.t; (* not yet used by proofs *)
5  }
```

[7] In CompCert, a trace is a (possibly infinite) sequence of observational events (*e.g.* external function calls) emitted by the execution of a given piece of code.

[8] In contrast to `psize`, the liveness information has no impact on the actual RTLpath semantics. It is, however, verified for correctness (but not for completeness).

```
6
7 Definition path_map: Type := PTree.t path_info
8
9 Definition path_entry (pm: path_map) (n: node): Prop := pm!n <> None
```

Listing 5.6.: Definitions of `path_info`, `path_map` and `path_entry`. `psize` details the number of instructions to execute in the path. `input_regs` are the live registers at the start of the path. `output_regs` is a field used by the scheduler oracle, without any impact on the proofs.

A `RTLpath` function is a `RTL` function, augmented with a pathmap. To be able to execute a whole function with such a pathmap, we need to have certain guarantees:

- The entrypoint of the function must be a path entry.
- Each successor of a given path must be a path entry.

If these two requirements are not met, the semantics will not be able to derive an execution (the execution will be invalid). These requirements are defined as predicates of well-formedness in listing 5.7: a path is *well-formed* if each of its successors is a path entry; a pathmap is *well-formed* if each path is *well-formed*; a `RTLpath` function is *well-formed* if its entrypoint is a path entry, and its pathmap is *well-formed*.

```
1  Inductive wellformed_path (c:code) (pm: path_map): nat → node → Prop :=
2    | wf_last_node i pc:
3        c!pc = Some i →
4        (∀ n, List.In n (successors_instr i) → path_entry pm n) →
5        wellformed_path c pm 0 pc
6    | wf_internal_node path i pc pc':
7        c!pc = Some i →
8        default_succ i = Some pc' →
9        (∀ n, early_exit i = Some n → path_entry pm n) →
10       wellformed_path c pm path pc' →
11       wellformed_path c pm (S path) pc
12
13 (* all paths defined from the path_map are wellformed *)
14 Definition wellformed_path_map (c:code) (pm: path_map): Prop :=
15   ∀ n path, pm!n = Some path → wellformed_path c pm path.(psize) n
16
17 Record function : Type :=
18  { fn_RTL:> RTL.function;
19    fn_path: path_map;
20    (* condition 1 below: the entrypoint of the code is an entrypoint of a path *)
21    fn_entry_point_wf: path_entry fn_path fn_RTL.(fn_entrypoint);
22    (* condition 2 below: the path_map is well-formed *)
23    fn_path_wf: wellformed_path_map fn_RTL.(fn_code) fn_path
24  }
```

Listing 5.7.: Definitions of well-formedness for paths, pathmaps and functions.

Figure 5.4 shows three different examples of pathmaps for the example that was introduced in figure 5.1. The pathmap A is not well-formed: 7 is not a path entry, so the unconditional jump from 3 to 7 would result in an invalid execution. The pathmap B is well-formed, but the path

from 9 to 3 does not constitute a superblock, since there is a side-entrance at instruction 7. The pathmap C is well-formed, and each path forms a superblock: this is the kind of pathmap that we desire.[9]

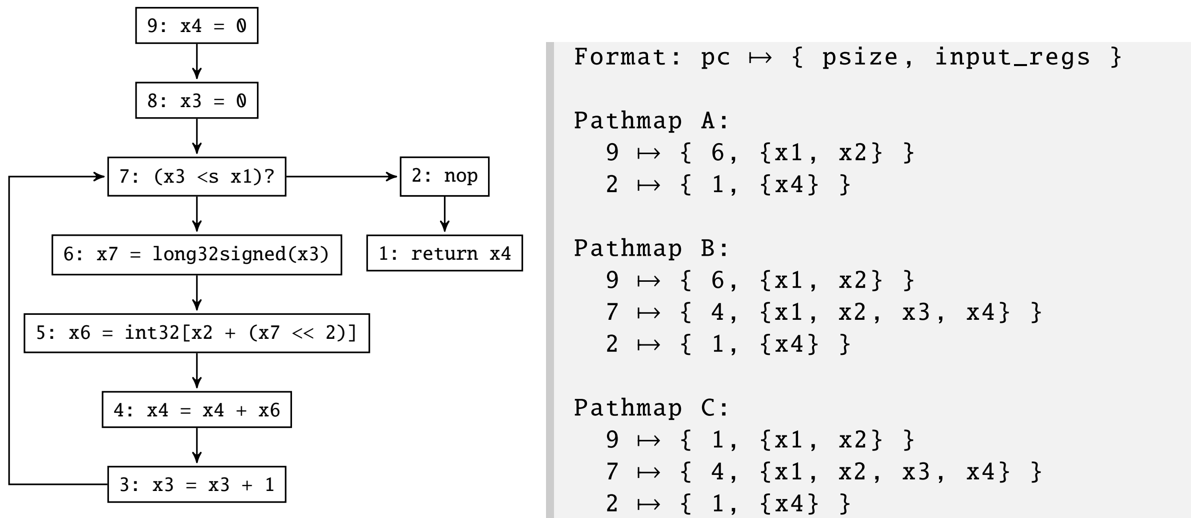


Figure 5.4.: Three examples of pathmaps for the RTL code on the left. We omit the `output_regs` field. Pathmap A would lead to an incorrect execution. Pathmap B is correct but undesirable because the path starting at 9 is not a superblock. Pathmap C is correct and desirable.

5.2.2 RTLpath Formal Semantics

A RTLpath state is the same as a RTL state, with one exception: instead of having RTL functions in the stack frames and other fields, we have RTLpath functions. As such, an RTLpath state is either a `State` (an execution inside the current function), a `Returnstate` (returning from a function), or a `Callstate` (about to call a function).

The only significant change is in the step semantics. Instead of executing a single RTL instruction, we execute an entire path (listing 5.8): the `pc` value of an RTLpath state has to point to a path entry, from which the path information `path` is retrieved.

```
1  Inductive step ge pge: state → trace → state → Prop :=
2    | exec_path path stack f sp rs m pc t s:
3        (fn_path f)!pc = Some path →
4        path_step ge pge path.(psize) stack f sp rs m pc t s →
5        step ge pge (State stack f sp pc rs m) t s
```

Listing 5.8.: Executing an entire path in one step

[9]In terms of formal correctness, only the A case will be ruled out by our verifier. The B case is a valid choice in terms of RTLpath semantics. However, the B case would later fail during the superblock scheduling verification. More details in chapter 7.

To model the path execution, we introduce a notion of internal state (listing 5.9). This state has four components: `icontinue` dictates whether a side-exit is taken (if a conditional branch got triggered), `ipc` points to the next instruction to execute within the path, and `irs` and `imem` are the internal states of the registers and memory.

```
1  Record istate := mk_istate { icontinue: bool; ipc: node; irs: regset; imem: mem }
```

Listing 5.9.: RTLpath internal state

This internal state is transformed by the `istep` function, which evaluates the new `irs`, `imem` and `ipc`, as well as updating `icontinue`. I detail two examples in listing 5.10. For most of the schedulable instructions (like `Iop`), `istep` is mostly just a computational variant of the RTL (single-) step semantics. For `Icond`, it requires in addition to set `icontinue` according to the condition evaluation.

```
1   Definition istep (ge: RTL.genv) (i: instruction) (sp: val) (rs: regset) (m: mem):
2       option istate :=
3     match i with
4     | Iop op args res pc' ⇒
5         SOME v ← eval_operation ge sp op rs##args m IN
6         Some (mk_istate true pc' (rs#res ← v) m)
7     | Icond cond args ifso ifnot _ ⇒
8         SOME b ← eval_condition cond rs##args m IN
9         Some (mk_istate (negb b) (if b then ifso else ifnot) rs m)
10    | (* ... *)
11    | _ ⇒ None
12    end
```

Listing 5.10.: Internal execution of `Iop` and `Iload`

With `istep` defined, we can then code the `isteps` function which executes successively each instruction until there is no more instruction to execute but one (we reached the bounded end of the path) or until `icontinue` equals `false`. This `isteps` is guided by `path`, a natural number defining how many instructions to execute at maximum within the path. If `path` is 0, nothing is executed and `isteps` returns the current state.

```
1   Fixpoint isteps ge (path:nat) (f: function) sp rs m pc: option istate :=
2     match path with
3     | O ⇒ Some (mk_istate true pc rs m)
4     | S p ⇒
5       SOME i ← (fn_code f)!pc IN
6       SOME st ← istep ge i sp rs m IN
7       if (icontinue st) then
8         isteps ge p f sp (irs st) (imem st) (ipc st)
9       else
10        Some st
11    end
```

Listing 5.11.: Executing several instructions along a path with `isteps`

The last instruction is executed by a `path_last_step` relation, which mimics the `step` relation from RTL. The `path_step` relation (listing 5.12) bridges it all together: either `icontinue` is false, in which case there is no more instruction to execute; or `icontinue` is true, implying that we execute the final instruction with `path_last_step`.

```
Inductive path_step ge pge (path:nat) stack f sp rs m pc: trace → state → Prop :=
  | exec_early_exit st:
     isteps ge path f sp rs m pc = Some st →
     (icontinue st) = false →
     path_step ge pge path stack f sp rs m pc E0
       (State stack f sp (ipc st) (irs st) (imem st))
  | exec_normal_exit st t s:
     isteps ge path f sp rs m pc = Some st →
     (icontinue st) = true →
     path_last_step ge pge stack f sp (ipc st) (irs st) (imem st) t s →
     path_step ge pge path stack f sp rs m pc t s
```

Listing 5.12.: Executing an entire path

Figure 5.5 gives a summary of these semantics. The starting point is an RTLpath `State`, from which we capture rs, m and pc into an internal state. We then distinguish two cases: either we execute the whole path, in which case we get an internal state $(ipc_1, irs_1, im_1, true)$ and we perform one final step with `path_last_step` to get the state s'_1 (which can be a `Callstate`, a `Returnstate`, or just a `State`). Or, after n instructions, the execution reaches a conditional branch whose condition evaluates to true, in which case `icontinue` is set to $false$, and the final state is constructed with ipc_2, irs_2 and im_2.

5.2.3 Bisimulation between RTLpath and RTL Semantics

To plug RTLpath in the compilation flow of CompCert, we have to prove a bisimulation between RTL and RTLpath semantics: this will make it possible and certified to use RTLpath instead of RTL, and vice-versa. This comes in two steps: proving a simulation from RTLpath to RTL, then doing the same for RTL to RTLpath.

Simulation from RTLpath to RTL

Proving that we can go back to RTL semantics from RTLpath is the easiest part of the bisimulation. The gist of the proof lies in exhibiting that all internal transitions from RTLpath can be seen as RTL transitions.

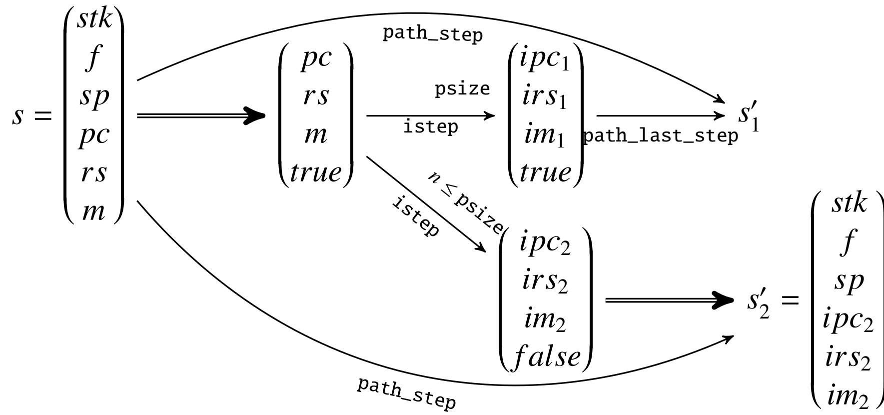


Figure 5.5.: Executing a path from s (an `RTLpath.State`). Two possible executions are shown. One executes an entire path by computing `istep psize` times, finishing by a `path_last_step` which gives the final state s'_1. The other computes `istep` only n times, then reaches a point where `icontinue = false`: it exits the path and returns a `State` s'_2. The entire execution is a `path_step`.

To begin, we prove by case analysis that the `istep` function simulates the `RTL.step` relation.

```
Lemma istep_correct ge i stack (f:function) sp rs m st :
  istep ge i sp rs m = Some st →
  ∀ pc, (fn_code f)!pc = Some i →
  RTL.step ge (State stack f sp pc rs m) E0 (State stack f sp st.(ipc) st.(irs) st.(imem))
```

Listing 5.13.: Simulation from `istep` to `RTL.step`

Then, by induction, we can prove a simulation between `isteps` and `RTL.step`. In the most general case, this simulation is of the *star* type, since `isteps` can be run with a `path=0` argument. However, if an early exit is taken (if `icontinue` of the final internal state is `false`), then we can prove that this is a *plus* (instead of a *star*) simulation, since we would have executed at least one conditional branch instruction. Similarly, we can prove a step simulation between `path_last_step` and `RTL.step`.

Using the above, we can prove a plus simulation from RTLpath to RTL by a case analysis on `path_step`: either the `path_step` was constructed with the `exec_early_exit` constructor, and thus we have directly a plus simulation — or it was constructed with the `exec_normal_exit` constructor, and we thus have a star simulation for `isteps` combined with a step simulation for `path_last_step`, so, *in conclusion*, a plus simulation.

Simulation from RTL to RTLpath

This part is more complex. On the one hand, we have an instruction-per-instruction execution semantics. On the other hand, we execute a whole path at each step. Proving the simulation requires then to introduce a notion of *stuttering*, much like the `Mach` to `Machblock` proof: we must have a `match_states` relation which allows for the RTL program to advance one step, while the RTLpath stutters (but with progression in the internal state).

We call this new relation `match_inst_states_goal`, with the following definition:

```
Inductive match_inst_states_goal (idx: nat) (s1:RTL.state): RTLpath.state → Prop :=
  | State_match path stack f sp pc rs m is2:
    (fn_path f)!pc = Some path →
    (idx ≤ path.(psize))%nat →
      isteps ge (path.(psize)-idx) f sp rs m pc = Some is2 →
      s1 = State stack f sp is2.(ipc) is2.(irs) is2.(imem) →
      match_inst_states_goal idx s1 (State stack f sp pc rs m)
```

Listing 5.14.: `match_inst_states_goal definition`

Informally, `match_inst_states_goal idx s1 s2` holds if a certain number of instructions have already been executed in the path indicated by the `pc` from `s2`, there remain `idx` instructions to execute before the syntactic end of the path, and the current internal execution state `is2` matches structurally with the RTL state `s1`. I give a more graphical representation of this relation in figure 5.6: several states from RTL match to the same state from RTLpath, but with a different `idx` value. Once this value reaches 0, or when an early exit is taken, the RTLpath state changes with a `path_step`.

The actual `match_step` relation combines `match_inst_states_goal` with a more classical structural equality for `Callstates` and `Returnstates`. `wf_stackframe` is a necessary predicate to ensure that `pc` points to a valid path after exiting another.

```
Definition is_inst (s: RTL.state): bool :=
  match s with RTL.State stack f sp pc rs m ⇒ true | _ ⇒ false end

Definition match_inst_states (idx: nat) (s1:RTL.state) (s2:state): Prop :=
  if is_inst s1 then match_inst_states_goal idx s1 s2 else s1 = state_RTL s2

Definition match_states (idx: nat) (s1:RTL.state) (s2:state): Prop :=
    match_inst_states idx s1 s2 ∧ wf_stackframe (stack_of s2)
```

Listing 5.15.: `match_states` definition for RTL to RTLpath simulation

Proving the forward simulation from RTL to RTLpath relies then on a case analysis based on the initial RTL transition. The possible cases are the following (referring to figure 5.6 for illustration):

1. The RTL step corresponds to a RTLpath stuttering within a path. This is the case of the olive transition from the left subfigure, from S_{11} to S_{12}. We have then a *local stuttering* with `idx` as decreasing measure.
2. Or, the RTL step corresponds to the last instruction of a given path. This is the case of the red transition from the left subfigure, from S_{1n} to S'_1. We have then a *local lockstep*, with the transition from S_2 to S'_2.
3. Finally, the RTL step could be right after an early exit was taken. We have a `match_inst_states_goal (n-i)` between S_{1n} and S_2, and we also know that `icontinue` from s_{1i} is `false` by case analysis (if it was `true`, we would be in either case 1 or case 2). We can then construct a `path_step` from S_2 to S'_2. To analyze what happens next, we perform a case analysis on the length of the new path, n':
   - Either $n' = 0$ (not shown in the figure) in which case we get one additional `path_step` (the RTL step actually corresponds to the one and only instruction from the new path). In total, we have then two RTLpath transitions.
   - Or $n' > 0$ (case shown in the figure), in which case we have a stuttering after the RTLpath transition.

   In both cases, we have a *local plus simulation*.

We were able to construct in Coq a *forward simulation* with the combination of these three possible cases, which certifies the use of RTLpath semantics instead of RTL.

5.3 Generating RTLpath Paths from RTL

Now that we have a semantics for executing paths and two forward simulations between RTL and RTLpath, the only missing piece for a complete RTL-to-RTLpath pass is the generation of the *pathmap*.

This pathmap is generated by an oracle, described in subsection 5.3.1.

Then, its result is verified for two purposes:

1. The pathmap has to satisfy the well-foundedness predicate. In other words, the pathmap must have a valid entry for the RTL entrypoint, and each successor of each path must be a valid entry. This is necessary to prove the RTL-to-RTLpath forward simulation from the previous section. This verification is described in subsection 5.3.2.
2. Even though our forward simulation does not involve any property about the correctness of the liveness information provided in our `path_info` structure, it is required later on for the superblock scheduling correctness. So we include within the pass a liveness verificator,

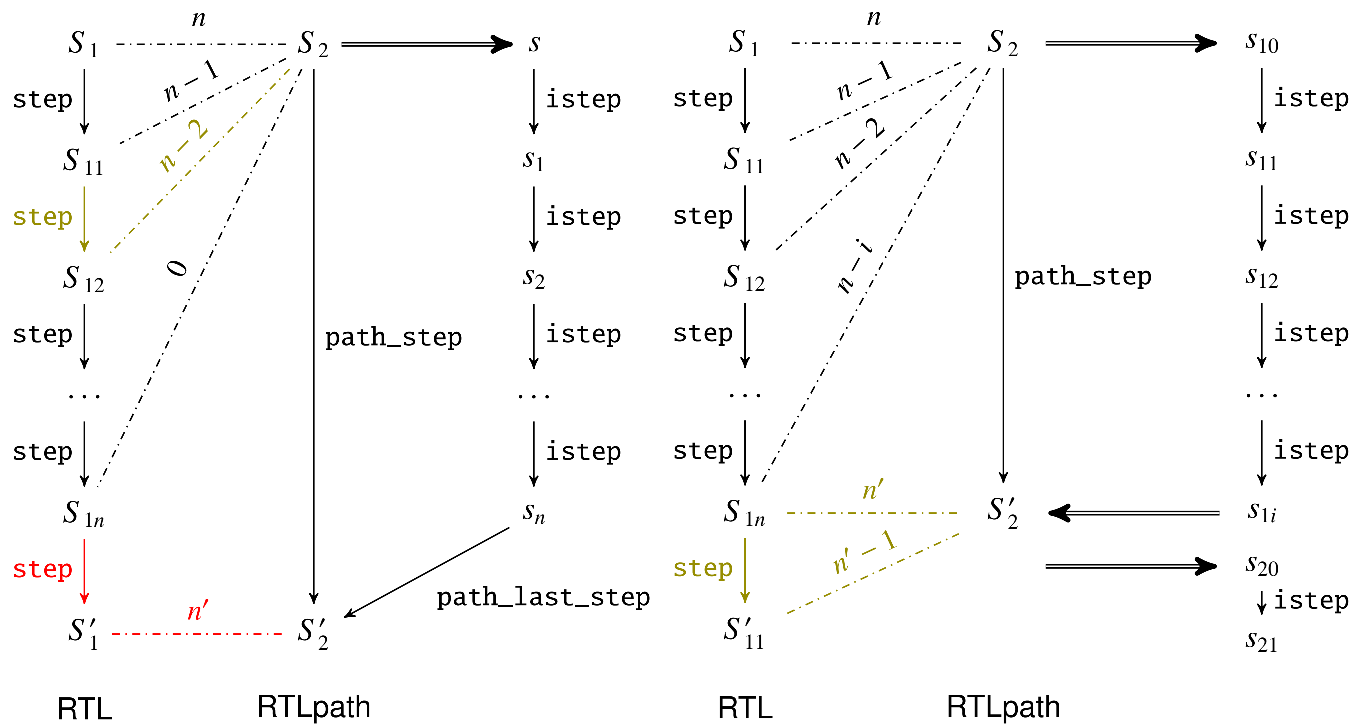


Figure 5.6.: Bridging RTL and RTLpath execution. The left figure shows an execution through a whole path: the right figure shows an execution with the path ending before the end because of an early exit. The red transition outlines the RTL execution of the last instruction from the path; the olive transitions outline RTL executions corresponding to RTLpath stutterings. Dotted edges correspond to `match_inst_states_goal idx` with each `idx` value displayed on top of an edge.

which checks that the liveness information returned by the oracle is correct. This is described in subsection 5.3.2.

5.3.1 RTLpath Generation Oracle

Our oracle performs two main tasks: identifying the superblocks to schedule and generating liveness information for each superblock.

Superblock Identification

As was described in chapter 4, superblocks are contiguous units of code with single entrances. In the RTL to RTLpath pass, we suppose that all the pre-scheduling superblock-related passes (tail-duplication, loop unrolling) have already been done. We then just have to identify the superblocks to schedule: given an RTL control-flow graph (CFG) and its entrypoint, we want to partition that graph into paths that are superblocks.

We want those superblocks to be non-overlapping. Indeed, if we were to create a pathmap containing overlapping superblocks, the scheduling pass could fail. Let us imagine a superblock containing the instructions `[i1; i2]` and the other containing `[i1; i2; i3]`. One possible transformation would be to schedule the first into `[i1; i2]` and the second into `[i3; i1; i2]`. The latter superblock scheduling would be correct; however the former would not be correct, since a new instruction would have been added to the superblock.

To identify the superblocks, we can start by identifying the *join points* among RTL nodes. The *join points* are the nodes in the CFG that have multiple predecessors. These cannot be included in a superblock (unless they are the first instruction of the superblock) because that would create side entrances.

We compute the join points by a graph DFS (Depth-First Search) traversal starting from the entrypoint. The core idea of this traversal is to memorize whether a node was visited or not. If, during a branch traversal, we come accross a node that was already visited, then that node is a *join point*, and we stop the search for that branch. Algorithm 1 shows pseudo-code for computing them. It should be noted that this algorithm is the same as the one used in mainstream CompCert for the `Linearize` pass.[10]

Once the join points are identified, we start from the entrypoint and grow a superblock from there, with the following rules:

[10] Computing join points is of interest for figuring out a good linearized layout, that is, one that minimizes the impact of jump penalties. More details can be found in section 8.3.

Algorithm 1. Identifying join points in RTL code

```
1:  function GET_JOIN_POINTS(code, entry)
2:      visited ← { all ↦ false }
3:      join_points ← { all ↦ false }
4:      function TRAVERSE(pc)
5:          if visited[pc] then
6:              join_points[pc] := true
7:          else
8:              visited[pc] := true
9:              traverse_all(successors(code[pc]))
10:     function TRAVERSE_ALL(l)
11:         match l with
12:             case []
13:                 pass
14:             case pc :: l
15:                 traverse(pc)
16:                 traverse_all(l)
17:     traverse(entry)
18:     return join_points
```

- If we encounter an `Icond` with a privileged direction, we continue growing in that direction and keep track of the early exit. Encountering an unpredicted `Icond` results in stopping the superblock growth.
- If the next instruction is a join point, we stop the growth.
- If we encounter an instruction that does not have exactly one successor and which is not an `Icond`, we include that instruction in the superblock and stop the growth.

Once we are done growing a superblock, we grow another one from one of its successor nodes (either an early exit or a successor from the last instruction). We also keep track of which node was visited in order to prevent infinite loops from CFG cycles.

I write below our algorithm for identifying superblocks. This algorithm does not update the liveness information — this is done later.

Computing Liveness Information

Given a set of instructions to execute, it is possible to compute the set of registers whose initial values may alter the execution of an RTL code. For example, in the code below:

```
1 1: x0 = x1 + x2;
2 2: x3 = x1 * x0;
3 3: ret x3
```

Algorithm 2. Identifying superblocks in RTL code

```
 1: function GET_PATH_MAP(code, entry, join_points)
 2:   visited ← { all ↦ false }
 3:   pathmap ← { }
 4:   to_visit ← { entry }
 5:   while to_visit ≠ ∅ do
 6:     e ← to_visit.pop()
 7:     if visited[e] then
 8:       continue
 9:     visited[e] := true
10:     psize ← -1
11:     pc ← e
12:     loop
13:       inst ← code[pc]
14:       ps ← predicted_successor(inst)
15:       to_visit += non_predicted_successors(inst)
16:       psize += 1
17:       match ps with
18:         case None
19:           break
20:         case Some s
21:           if join_points[s] then
22:             break
23:           pc := s                                    ▷ The path is finished
24:     pathmap[e] := { psize = psize; input_regs = ∅; output_regs = ∅ }
25:   return pathmap
```

This set would be $\{x1, x2\}$: changing the initial value of $x1$ or $x2$ changes the value returned by the function. In the compiler litterature, such an analysis is called *liveness analysis*.

This analysis is already done by the `Liveness` pass of CompCert but in a slightly different fashion. The `Liveness` module of CompCert assumes that the instructions that compute a useless result (a result that isn't live) will later be removed, so it ignores these instructions. In our case, we want a complete analysis, since we will not be removing any instructions.

Since the `Liveness` module of CompCert was not a perfect fit for us, and implementing a verifier was not that costly to do in terms of implementation and proof effort, we implemented our own, re-using most of the definitions of the `Liveness` module in our oracle.

For a branchless code, a liveness analysis can be done with a backward approach: given a set of live registers after a given instruction, it is possible to compute the set of live registers before that instruction. In the above example, the set of live registers would be: before `3`, $\{x3\}$; before `2`, $\{x1, x0\}$ ($x3$ is removed because it was assigned); before `1`, $\{x1, x2\}$ ($x0$ is removed because it was computed — $x2$ is added since it's in the input operands of `1`). The function which performs this operation of adding/removing registers is called the *transfer function*. We give below an example of a transfer function for three instructions.

Algorithm 3. Transfer function for liveness analysis

```
1: function TRANSFER(code, pc, live_after)
2:     match code[pc] with
3:         case Inop _
4:             return live_after
5:         case Iop _ args res _
6:             return live_after - { res } + args
7:         case Istore _ _ args src _
8:             return live_after + { res } + args
9:         ...
```

For code containing branches but without loops, it is possible to perform this same backward analysis, except that when a *split* is encountered (a *join point* under the perspective of a backward analysis), the set of live registers before the split is $S_1 \cup S_2$ with S_1 and S_2 the respective sets of live registers before each branch of the split.

For code containing loops, we need to use a more refined algorithm such as Kildall's (Kildall [44]). This algorithm allows, given a transfer function and a semi-lattice (a partially ordered set equipped with a least-upper-bound function), to propagate the analysis to graphs containing cycles. In our case, we manipulate sets of integers, they are partially ordered by the inclusion operator, and the least-upper-bound function is the union operator.

We run this analysis using CompCert's `Kildall` module on the RTL code with our own `transfer` function — then, extracting the `input_regs` field of a given path is as simple as fetching the set of live registers before the entry of the path.

The `output_regs` field, used by the scheduling oracle, represents the registers that are live after the last instruction of the path. It is computed as the union of all the `input_regs` from the successors of the last instruction of the path. This field is used only by the scheduling oracle and has no impact on correctness.

5.3.2 Pathmap Verifiers

Since the pathmaps are generated by external oracles, their information needs to be verified.

We have two distinct cases of pathmap generation: right before superblock scheduling (during the RTL-to-RTLpath pass) and within the superblock scheduling pass. In both cases, we need to verify the well-foundedness of the pathmap. The liveness information, however, only needs to be verified for the former: the superblock scheduling will rely on that information, so it must be correct; but this information is of no use once the scheduling is actually done.

The next subsections will detail how we verify well-foundedness and liveness for the RTL-to-RTLpath pass.

Well-Foundedness Verification

The correctness of the RTL-to-RTLpath transformation partly resides in the guarantee that all paths are well-founded; that is, each successor of a path must be a valid path entry.

First of all, to verify that a given path is well-founded, we go through each instruction of a path: if this instruction is an `Icond`, then we check that the exit branch is a path entry. Finally, we do the same for the successors of the final instruction of the path.

Once we have this verification in place, verifying the well-foundedness of our pathmap is done by checking that the RTL entrypoint is a valid path entry, and that each path from the pathmap is well-founded (listing 5.16). We have then formally proven the two theorems in listing 5.17, which ensure that our verifier is correct.

```
1  Definition function_checker (f: RTL.function) pm: bool :=
2    pm!(f.(fn_entrypoint)) &&& list_path_checker f pm (PTree.elements pm)
```

Listing 5.16.: Well-foundedness verifier

```
1 Lemma function_checker_wellformed_path_map f pm:
2   function_checker f pm = true → wellformed_path_map f.(fn_code) pm
3
4 Lemma function_checker_path_entry f pm:
5   function_checker f pm = true → path_entry pm (f.(fn_entrypoint))
```

Listing 5.17.: Correctness theorems for the well-foundedness verifier

Liveness Analysis Verification

Since the liveness information is computed by an oracle, we also need to verify its correctness. This is done by taking the `input_regs` field of each path and performing a simple forward analysis: go through each instruction of the path in order, check that each read register is in the set of currently live registers, and add the written register to the set of live registers. For each encountered side exit, we also verify that the set of live registers matches the liveness information of the side-exit path. This is summarized by the below algorithm.[11]

Algorithm 4. Verifying liveness information

```
 1: function PATH_CHECKER(path, entry, code)
 2:     live ← path.input_regs
 3:     psize ← path.psize
 4:     pc ← entry
 5:     loop
 6:         inst ← code[pc]
 7:         read ← read_regs(inst)
 8:         written ← written_reg(inst)
 9:         assert(read ⊆ live)
10:         live := live + written
11:         psize := psize - 1
12:         if psize == 0 then
13:             break
14:         succ ← successors(inst)
15:         pc := preferred_successor(inst)
16:         foreach side_exit ∈ succ - {pc}
17:             assert(f.fn_path[side_exit].input_regs ⊆ live)
18:     return OK
```

This check is done during the RTL-to-RTLpath generation. To remember that it succeeded for the later scheduling pass, we define the following predicate:

```
1 Definition liveness_ok_function (f: function): Prop :=
```

[11] The actual Coq implementation is a bit more complex since it is done through recursion and with several separate individual functions, but the gist of it is the same.

```
2   ∀ pc path, f.(fn_path)!pc = Some path → path_checker f f.(fn_path) pc path = Some tt
```

Finally, instead of just generating a RTLpath function, we generate a habitant from the set of RTLpath functions fulfilling the predicate liveness_ok_function $f' \wedge f'$.(fn_RTL) $= f$ where f' is the new RTLpath function and f is the original RTL function.

```
1 Program Definition transf_function (f: RTL.function):
2   { r: res function | ∀ f', r = OK f' → liveness_ok_function f' ∧ f'.(fn_RTL) = f } :=
3   let pm := build_path_map f in
4   match function_checker f pm with
5   | true ⇒ OK {| fn_RTL := f; fn_path := pm |}
6   | false ⇒ Error(msg "RTLpathGen: function_checker failed")
7   end
```

5.3.3 RTLpath State Equality Modulo Liveness

As was described briefly in section 4.3.1, scheduling an instruction above an early-exit requires to check whether the instruction writes in a register that is live in the other branch. In order to reason on that, we introduce a notion of *state equality modulo liveness*.

Given live predicate stating whether a given register is live, eqlive_reg live rs1 rs2 holds if rs1 and rs2 have the same values for each live register.

```
1 Definition eqlive_reg (live: Regset.elt → Prop) (rs1 rs2: regset): Prop :=
2   ∀ r, (live r) → rs1#r = rs2#r
```

Listing 5.18.: Definition of regset equality modulo liveness

We can then build a notion of state equality eqlive_states using this definition (I show only the definition for an RTLpath State). ext live is a shorthand for writing $r \mapsto r \in$ live.

```
1 Inductive eqlive_states: state → state → Prop :=
2   | eqlive_states_intro
3       path st1 st2 f sp pc rs1 rs2 m
4       (STACKS: list_∀2 eqlive_stackframes st1 st2)
5       (LIVE: liveness_ok_function f)
6       (PATH: f.(fn_path)!pc = Some path)
7       (EQUIV: eqlive_reg (ext path.(input_regs)) rs1 rs2):
8       eqlive_states (State st1 f sp pc rs1 m) (State st2 f sp pc rs2 m)
9   (* ... *)
```

Listing 5.19.: State equality modulo liveness

This definition of regset equality is weaker than the complete regset equality we used until now. In particular, this definition alone may be too weak to be able to simulate a RTLpath

program. Indeed, if we were to prove a step simulation from a given code to itself (a "self simulation"), and if for one particular path we had an incorrect `input_regs` field (say, one where `input_regs` = ∅), then the `eqlive_reg` predicate would not be giving any meaningful information (since no register is in ∅, this predicate automatically holds). For instance, let us consider the code `if (x1) then f() else g()`, with `f` and `g` being two external functions. Without any information about what the value of `x1` could possibly be, there is no way to know which of `f` or `g` will end up being executed: we might get a different trace, which prevents any forward simulation. In fact, if the liveness oracle had returned a correct result, then we would have had $x1 \in$ `live`, which would have implied an equality $rs1[x1] = rs2[x1]$ and thus the same trace.

So we need to combine this definition with the liveness verifier of subsection 5.3.2 to prove such a self simulation. I write below the self simulation that we prove:

```
1 Lemma path_step_eqlive path stk1 f sp rs1 m pc t s1 stk2 rs2:
2   path_step ge pge (psize path) stk1 f sp rs1 m pc t s1 →
3   list_∀2 eqlive_stackframes stk1 stk2 →
4   eqlive_reg (ext (input_regs path)) rs1 rs2 →
5   liveness_ok_function f →
6   (fn_path f) ! pc = Some path →
7    ∃ s2, path_step ge pge (psize path) stk2 f sp rs2 m pc t s2 ∧ eqlive_states s1 s2
```

Listing 5.20.: Self simulation of RTLpath with state equality modulo liveness

The above lemma is a step simulation within the same function `f`, using `eqlive_states`. It states that if we execute a path with a given `input_regs` field (by a RTLpath step), and we have an initial state equality modulo `input_regs`, and that field was indeed verified by our liveness verifier, then we can execute that path and the next path will start with another `eqlive_states` binding the registers from its `input_regs`. Proving such a lemma involves a case analysis on which RTLpath step is taken (an early exit or going through the entire path), as well as breaking it into smaller lemmas guaranteeing the same kind of property, but applied to each internal step function or relation (such as `istep`) instead.

The `path_step_eqlive` lemma will be very useful for proving our superblock scheduling in chapter 7.

6 Formal Verification of Code-Duplication Transformations

Chapter 5 explained how the superblocks are selected, the superblock execution semantics, as well as an introduction to the RTL language. Chapter 4 introduced certain transformations done before superblock scheduling: tail-duplication and loop unrolling, but without giving much detail about those. These transformations feature a common point: they do not change the instructions; they just duplicate them and update their node pointers.

In this chapter, we introduce `Duplicate`, a module used to verify any transformation that duplicates instructions while preserving the control-flow structure within RTL. The module itself is very light: the `coqwc` tool indicates 350 lines of specifications and 300 lines of proof.

This module can be instantiated for potentially any number of fitting optimizations, which allows to easily add or remove any such optimization with barely any modification of the underlying Coq proofs. In our case, we use it for tail-duplication, loop unrolling and loop rotation; with a slight modification, we also support branch swapping.[1]

A `Duplicate` module instance is made of an untrusted OCaml oracle and a common verifier. We typically create one instance per different OCaml oracle optimization.[2]

The untrusted oracles have the following type in Coq:

```
1  Parameter duplicate_aux: function → code * node * (PTree.t node)
```

Given a function to perform a transformation (defined by the oracle) on, it returns:

- The new code on which the transformation was applied.
- The new entrypoint of the function (on the new code).
- A mapping from nodes of the new code to the nodes of the original code. We call this mapping a *reverse map*: its information will guide the verifier.

[1] This is a necessary transformation for RTLpath, where the predicted branch of `Icond` instructions is supposed to be in the `ifnot` field. More details in section 5.2.1.

[2] The actual oracle implementations are detailed in chapter 8.

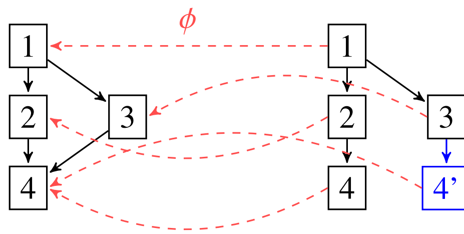


Figure 6.1.: Example of code duplication, with its reverse mapping. To the left, the original code. To the right, the transformed code. The reverse mapping is shown in red.

I give in figure 6.1 an example of code duplication. The nodes 1, 2, 3 and 4 are unchanged. However, a new node 4, named $4'$ in the figure, is introduced: it is a duplication from node 4. The node 3 is then modified so that its successor becomes $4'$. The reverse mapping ϕ indicates which node originated from where. Thanks to ϕ, we will be able to ensure formally that the transformation is correct.

In section 6.1 I detail the forward simulation used for the formal proof of the underlying pass. Next, in section 6.2, I give some details of the verifier implementation. Then, in section 6.3, I detail a bit of the Coq engineering that makes it possible to share efficiently (in terms of proof effort) a common verifier with many different OCaml external oracles. Finally, I give in section 6.4 the modifications made to support branch swapping.

Details about each actual transformation oracle will be given in the remaining sections.

6.1 Proof of Correction through a Forward Simulation

Let us note an original function f and a transformed function f' acquired through replacing the code and entrypoint with those given by the oracle, and a reverse mapping ϕ given by the oracle. The gist of the forward simulation lies in exhibiting that the execution of a node n from f always corresponds to an execution of a node n' from f', with n' verifying $\phi(n') = n$. Each element of the mapping ϕ will be verified to ensure the following property: $\phi(n') = n \implies n \equiv n' \cap \phi(succ(n')) = succ(n)$, where $succ(n)$ denotes the successors of n, and $\equiv$ is the `match_inst` relation defined below. The mapping ϕ is actually named `dm` or `dupmap` in the code (for "duplication map"). The `dm ! n' = Some n` notation stands for: "There is a mapping in dm associated to n', and it points to n.".

```
1  Inductive match_inst (dupmap: PTree.t node): instruction → instruction → Prop :=
2    | match_inst_op: ∀ n n' op lr r,
3        dupmap!n' = (Some n) → match_inst dupmap (Iop op lr r n) (Iop op lr r n')
4    | match_inst_cond: ∀ ifso ifso' ifnot ifnot' c lr i i',
5        dupmap!ifso' = (Some ifso) → dupmap!ifnot' = (Some ifnot) →
```

```
6        match_inst dupmap (Icond c lr ifso ifnot i) (Icond c lr ifso' ifnot' i')
7     | (* ... *)
```

I give two examples for this relation: `Iop` and `Icond`. There can be a `match_inst` between `(Iop op lr r n)` and `(Iop op' lr' r' n')` only if `op = op'`, `lr = lr'`, `r = r'` and $\phi(n') = n$. The `Icond` instructions must follow a similar property, with the exception that there are two nodes to check for the reverse mapping (`ifso` and `ifnot`), and the branch-prediction field is unchecked (since it does not have formal semantics). In a way, this relation is a structural equality, but modulo the pointers of successors, which must verify the mapping ϕ instead.

Given two functions (original and transformed) f and f', we then define a `match_function` relation:

```
1  Record match_function dupmap f f': Prop := {
2    dupmap_correct: ∀ n n', dupmap!n' = Some n →
3      (∀ i, (fn_code f)!n = Some i → ∃ i', (fn_code f')!n' = Some i' ∧ match_inst dupmap i i');
4    dupmap_entrypoint: dupmap!(fn_entrypoint f') = Some (fn_entrypoint f);
5    preserv_fnsig: fn_sig f = fn_sig f';
6    preserv_fnparams: fn_params f = fn_params f';
7    preserv_fnstacksize: fn_stacksize f = fn_stacksize f'
8  }
```

This relation ensures the following:

- The entrypoints obey the reverse mapping: this ensures that executing a function starts in the same point for both the original and the transformed code (modulo duplication).
- The `dupmap_correct` relation ensures that the associated instructions of each pair (n', n) from the reverse mapping follow a `match_inst` relation.
- The rest of the premises ensure that only the code and entrypoint differ, and the remainder of the data defining a RTL function are the same.

We then use a lockstep simulation to prove the transformation pass, with the following `match_states` relation (shown only for the case of a `State`; the other cases for `Returnstate` and `Callstate` are straightforward).

```
1  Inductive match_states: state → state → Prop :=
2    | match_states_intro
3        dupmap st f sp pc rs m st' f' pc'
4        (STACKS: list_∀2 match_stackframes st st')
5        (TRANSF: match_function dupmap f f')
6        (DUPLIC: dupmap!pc' = Some pc):
7        match_states (State st f sp pc rs m) (State st' f' sp pc' rs m)
8    | (* ... *)
```

This denotes a simulation where the register and memory states are structurally the same, but only the program counters pc and pc' change: they must verify $\phi(pc') = pc$.

6.2 Verifier Implementation

The core of the verifier is implemented through a simple function comparing two instructions, returning OK if each field is structurally the same, except for the successor pointers (and the branch prediction field of Icond instructions). Here is the implementation for Iop:

```
Definition verify_match_inst dupmap inst tinst :=
  | Iop op lr r n ⇒ match tinst with
      Iop op' lr' r' n' ⇒
          do u ← verify_is_copy dupmap n n';
          if (eq_operation op op') then
            if (list_eq_dec Pos.eq_dec lr lr') then
              if (Pos.eq_dec r r') then
                OK tt
              else Error (msg "Different r in Iop")
            else Error (msg "Different lr in Iop")
          else Error(msg "Different operations in Iop")
        | _ ⇒ Error(msg "verify_match_inst Inop") end
```

We use decidable equality functions to compute the verdict. These tests should run fast. We have not performed detailed measurements, but we have not noticed any odd timings when using these new duplication passes. The function verify_is_copy verifies that the pair (n', n) is indeed a binding of the reverse mapping dupmap.

We prove the correctness of the above function by the theorem verify_match_inst_correct:

```
Lemma verify_match_inst_correct:
  ∀ dupmap i i',
  verify_match_inst dupmap i i' = OK tt →
  match_inst dupmap i i'
```

The verify_match_inst function is run for each binding found in the reverse mapping. Along with an additional reverse-map check on the entrypoints, we can prove that verifying both functions in such a way ensures a match_function relation, from which the forward simulation is then proven.

6.3 Module System

The Duplicate verifier is embedded into the Coq module system, parametrized by the untrusted OCaml oracle. The duplicate_aux function from earlier is actually embedded in a DuplicateOracle module type.

```
Module Type DuplicateOracle
  Parameter duplicate_aux: function → code * node * (PTree.t node)
End DuplicateOracle
```

We then use two modules that are parametrized by a `DuplicateOracle`, following the CompCert way of structuring passes. The `Duplicate` module contains the implementation of the actual pass (the Coq code that calls the OCaml oracle and checks its result), and the `Duplicateproof` module contains the proofs of the former.

```
Module Duplicate (D: DuplicateOracle)
  Export D
  (* ... *)
End Duplicate

Module Duplicateproof (D: DuplicateOracle)
  Include Duplicate D
  (* ... *)
End Duplicateproof
```

Once these two modules are implemented following the content of the previous subsections, we just have to instantiate them with particular OCaml code to define formally verified transformation passes. We give below the instantiation of the tail-duplicate transformation as an example (the other transformations described in this chapter are instantiated the same way).

```
Module TailDuplicateOracle <: DuplicateOracle
 Axiom duplicate_aux : function → code * node * (PTree.t node)
 (* Duplicateaux.tail_duplicate is the actual name of the OCaml function *)
 Extract Constant duplicate_aux ⇒  "Duplicateaux.tail_duplicate"
End TailDuplicateOracle

Module Tailduplicateproof := DuplicateProof TailDuplicateOracle
Module Tailduplicate := Tailduplicateproof
```

The resulting modules can then be integrated into the standard CompCert RTL compilation flow.

6.4 Modifications to Support Branch Swapping

Branch swapping is the action of negating a condition and inverting the two branches from a conditional branch instruction. In essence, it turns a `(Icond c lr ifso ifnot p)` into a `(Icond (negate c) lr ifnot ifso p)`, where `negate` is a function negating a condition (already present in CompCert). The `Linearize` pass of CompCert already performs this kind of

transformation to minimize branch penalties in the generated program[3], but here we need it at the RTL level for RTLpath.

In our case, branch swapping is done by the static prediction oracle: it inserts prediction information in each Icond, and it also swaps branches accordingly. Verifying code duplication is then not enough to tackle this: we need to add support to the existing Duplicate module. Thankfully, though, this is relatively easy and straightforward.

To start, we add a new inductive rule for the match_inst relation:

```
Inductive match_inst (dupmap: PTree.t node): instruction → instruction → Prop :=
  (* ... *)
  | match_inst_revcond: ∀ ifso ifso' ifnot ifnot' c lr i i',
      dupmap!ifso' = (Some ifso) → dupmap!ifnot' = (Some ifnot) →
      match_inst dupmap (Icond c lr ifso ifnot i)
        (Icond (negate_condition c) lr ifnot' ifso' i')
```

We then modify the proof of the forward simulation to account for this new rule. The gist of the forward simulation remains unchanged: we have a simulation modulo the program counters. However, given identical register values, the execution of the original Icond and the modified Icond must lead to the same branch. This is ensured through using the eval_negate_condition lemma from upstream CompCert (given below), and exploiting the RTL semantics of Icond instructions.

```
Lemma eval_negate_condition:
  ∀ cond vl m,
  eval_condition (negate_condition cond) vl m = option_map negb (eval_condition cond vl m)
```

Finally, we modify our verifier by checking two possible cases on conditions c and c' respectively from the original and transformed Icond:

- Either $c = c'$, in which case we check that $\phi(\texttt{ifso}') = \texttt{ifso}$ and $\phi(\texttt{ifnot}') = \texttt{ifnot}$.
- Or $c = negate(c')$: we then check that $\phi(\texttt{ifso}') = \texttt{ifnot}$ and $\phi(\texttt{ifnot}') = \texttt{ifso}$.
- Or neither of the two cases: we reject the program.

[3]More details in section 8.3.

7 Formal Verification of Superblock Scheduling

In chapter 4 we saw the basic principles of superblock scheduling. In chapter 5 we defined the RTLpath semantics, based on RTL, for executing whole superblocks.

In this chapter, we use the RTLpath semantics to write a formally proven superblock scheduling verifier. Much like chapter 3, the pass consists of an external untrusted oracle returning a modified superblock and a verifier comparing the modified superblock with the original through symbolic execution.

In the case of superblock scheduling, the scheduling oracle is defined as such:

```
1  Axiom untrusted_scheduler: RTLpath.function → code * node * path_map * (PTree.t node)
```

Given an original function from RTLpath, the oracle devises a schedule for each superblock (each path from the function). It returns a tuple (tc, te, tpm, dm) where:

- tc is the modified code obtained by devising and applying a schedule on each superblock.
- te is the new node representing the entrypoint. Indeed, the untrusted scheduler can move the instruction pointed by the entrypoint further down, so the entrypoint might differ on the scheduled function.[1]
- tpm is the new pathmap: for the same reason as above, each path might have a different entry node.
- Finally, dm is the reverse mapping: much like chapter 8, we use this reverse mapping to ensure the successors of each superblock still point to the right paths and also to guide the verifier.

Each pair (sb', sb) of superblocks from the reverse mapping dm will be examined and compared through symbolic execution, explained below.

The goal of symbolic execution is to be able to derive a piece of data (a symbolic state) that can be used in conjunction with any initial state (before execution) to deduce its final state

[1] In practice, for debugging purposes, we chose that our untrusted scheduler should not modify the existing node values: we change the node content without changing the nodes themselves.

(after execution). We name the rules used to deduce the final state from the initial state (given a symbolic state) *symbolic execution semantics*. These have to be in bisimulation with the RTLpath semantics to ensure correctness. The big picture is then to compute the symbolic state of the original unscheduled superblock, do the same for the scheduled superblock, and compare them. If the scheduled block simulates (according to some simulation relation to be defined later) the input block, then, since their semantics are in bisimulation with the RTLpath semantics, the simulation should also hold in the RTLpath semantics.

Compared to our previous work in chapter 3 for basic-block scheduling, there are several additional difficulties:

- For basic-block symbolic execution, the control flow has only one possible direction: through the basic block. For superblock execution, if we have n conditional branches within the superblock, then the control flow has n different outcomes. This will need to be reflected within the definition of the symbolic state.
- The notion of RTL state is more complex than that of Asm. In particular, in Asm, we only needed to take into account the registers and memory. In RTL, we also have to take into account whether the final state should be a `State`, a `Returnstate`, or a `Callstate`; along with additional information such as the RTL program counter and the stack.
- Hoisting instructions above conditional branches requires to check that they do not write a register which is live in the other branch. This requires reasoning about liveness.

In our proof architecture, symbolic execution is split into two versions: an abstract symbolic execution, used to facilitate the bisimulation between this symbolic execution and RTLpath semantics; and a concrete symbolic execution, using refined symbolic states (with hash-consing) that can be compared with decidable functions. The link between abstract and concrete (refined) symbolic states is expressed through refinement relations.

First, I define in 7.1 the abstract symbolic execution for superblocks, along with its semantics and the necessary bisimulation required for correctness. Then, I describe in 7.2 the notion of symbolic simulation we use for comparing superblocks and outline a property of the simulation test sufficient to prove a forward simulation on the scheduling pass. Finally, I show the hash-consed implementation of the simulation test, along with the proof that this implementation meets the above property, in section 7.3. This implementation works on refined symbolic states.

The implementation of the scheduling oracle will be described in chapter 8; this chapter only contains the elements of correctness of our formally proven verifier. The work presented in this chapter was done in collaboration with Sylvain Boulmé for the design and some parts of the implementation.

7.1 Abstract Symbolic Execution for Superblocks

I introduce in this section abstract symbolic execution. I avoid repeating the word "abstract": in the text that will follow, "symbolic value" means "abstract symbolic value", and likewise for the other related terms. Concrete (refined) symbolic execution will be introduced in a later section.

The notions of symbolic value and symbolic memory form the building blocks of symbolic execution. A symbolic value is an expression, possibly including input registers or symbolic memory. These are defined below with mutual induction:

```
1  Inductive sval :=
2    | Sinput (r: reg)
3    | Sop (op:operation) (lsv: list_sval)  (sm: smem)
4    | Sload (sm: smem) (trap: trapping_mode) (chunk:memory_chunk) (addr:addressing)
5        (lsv:list_sval)
6  with list_sval :=
7    | Snil
8    | Scons (sv: sval) (lsv: list_sval)
9  with smem :=
10   | Sinit
11   | Sstore (sm: smem) (chunk:memory_chunk) (addr:addressing) (lsv:list_sval) (srce: sval)
```

Listing 7.1.: Symbolic value definition

The core idea of symbolic execution is to produce a symbolic state which maps registers and memory to symbolic values. The final values can then be deduced out of symbolic values and initial values by the evaluation functions `seval_sval` and `seval_smem`, whose types are given below.

```
1  seval_sval : RTL.genv → val → sval → regset → mem → option val
2  seval_smem : RTL.genv → val → smem → regset → mem → option mem
```

Figure 7.1 shows a simplified example on a small branchless code. The code contains one addition and one store. The symbolic execution gives a mapping associating the registers and memory to their symbolic values. Here, only the mappings of r_0, r_1, r_2 and m are represented [2]. r_0 and r_1 are not changed, so their symbolic values are simply `(Sinput` r_i`)`. r_2 and m are changed: r_2 contains the symbolic value constructed by a `Sop`, and the symbolic value of m is constructed with a `Sstore`, represented here by a "`mem[address ↦ value]`" notation, where `address` and `value` are symbolic values and `mem` is a symbolic value of memory. We can note that r_2 does not appear anywhere within the symbolic value of memory: the final memory is a

[2] Since an RTL state contains infinitely many of registers, the mapping also has to account for infinitely many registers. In our case, this is handled by having a default mapping: by convention, any unmentioned register r has the associated symbolic value `(Sinput r)`.

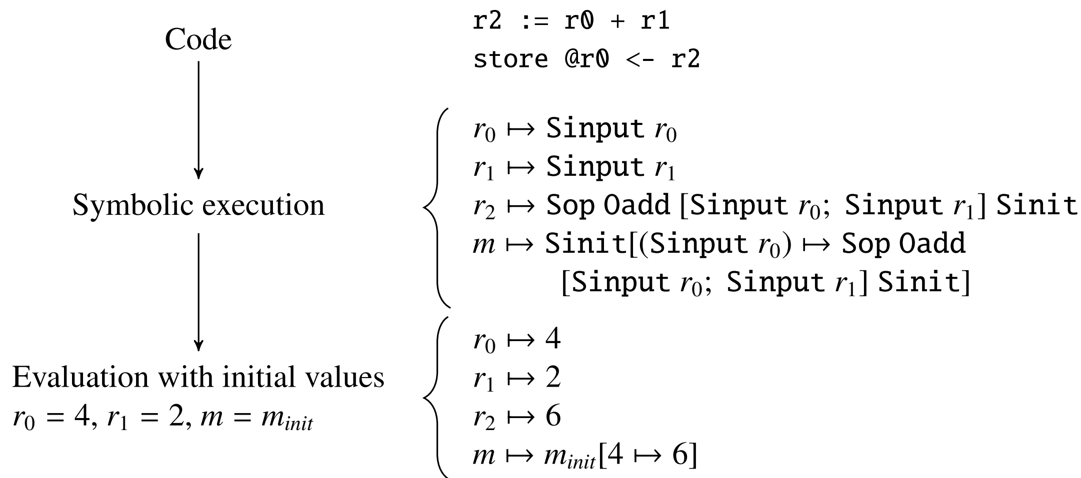


Figure 7.1.: Example of symbolic execution, then evaluation of the symbolic states

function of just the initial values of r_0, r_1 and m. Symbolic execution computes a kind of parallel assignment – called a symbolic state – that is equivalent to the sequential code. The semantics of this symbolic state evaluates it with the given initial values to deduce the final values of the registers and memory.

Another vital piece of information to store, aside from the final values, is whether there could be a failure (such as a division by zero or an invalid load) during the execution of the block of code. The main theorem of semantic preservation of CompCert assumes behaviours that do not fail. This property is then propagated through the forward simulations, so we have the guarantee that the original code will never fail. However, we do have to guarantee that the transformed code will not fail either. One way to handle this is to remember all the computed symbolic values (including the intermediate ones) and to test whether the scheduled block does not introduce any additional symbolic value that was not present in the original block. If that is the case, we have the guarantee that if the original block does not fail, then the scheduled block should not fail either.

Along with storing the symbolic values of registers and memory, this would be enough to reason about the execution of a branchless basic block. However, for superblocks, we need to store more information. In subsection 7.1.1, I describe a model of symbolic states for a superblock and its associated semantics. Then, I exhibit in 7.1.2 the process of performing the symbolic execution. Finally, I describe in 7.1.3 the main theorems ensuring a bisimulation between RTLpath and the presented symbolic-execution semantics.

7.1.1 Abstract Symbolic States and Execution Semantics

To execute a superblock symbolically, we need to have information on both the case where the whole superblock is executed (all the conditions from the conditional branches evaluated to `false`) and any case where the superblock exits at a given conditional branch (whose condition evaluation was `true`). Since we do not know at compile time which it will be, we need a symbolic data structure storing all the possibilities.

I present this data structure by following a bottom-up presentation:

- First, I present `sistate_local`, which is a symbolic internal state staying local to the superblock. This structure does not encode anything regarding exiting the superblock—it just stores the symbolic values of registers and memory.
- Then, I present `sistate_exit`: this structure holds both the symbolic state achieved when exiting the superblock, as well as the symbolic value of the condition, to later evaluate whether the condition is taken.
- The next step is to define a `sistate`, using the two above definitions: a `sistate_local` to keep track of the current register/memory symbolic values, and a list of `sistate_exit` to account for all the possible early exits of the superblock.
- Finally, we define a `sstate`, the actual symbolic execution structure. It combines a `sistate` with a `sfval`, a symbolic final value of the superblock. This `sfval` allows us to store symbolic information for the final instruction of the superblock, which could be any of `Icall`, `Ireturn`, `Ibuiltin`, `Itailcall` or `Ijumptable`.

Figure 7.2 shows a tree representation of our data structure. The following subsections will explain the technical details behind these choices.

Model of Local Internal Symbolic States

A `sistate_local` is defined like this:

```
1  Record sistate_local := { si_pre: RTL.genv → val → regset → mem → Prop;
2    si_sreg: reg → sval; si_smem: smem }
```

`si_sreg` and `si_smem` hold the symbolic values of respectively the registers and the memory. `si_pre` is a precondition on the initial values that must hold for the superblock execution to succeed.

Semantically, `(ssem_local ge sp st rs0 m0 rs m)` holds if, given initial registers/memory values rs_0 and m_0, evaluating the local internal symbolic state st would give the new register and memory states rs and m, and the `si_pre` precondition is met.

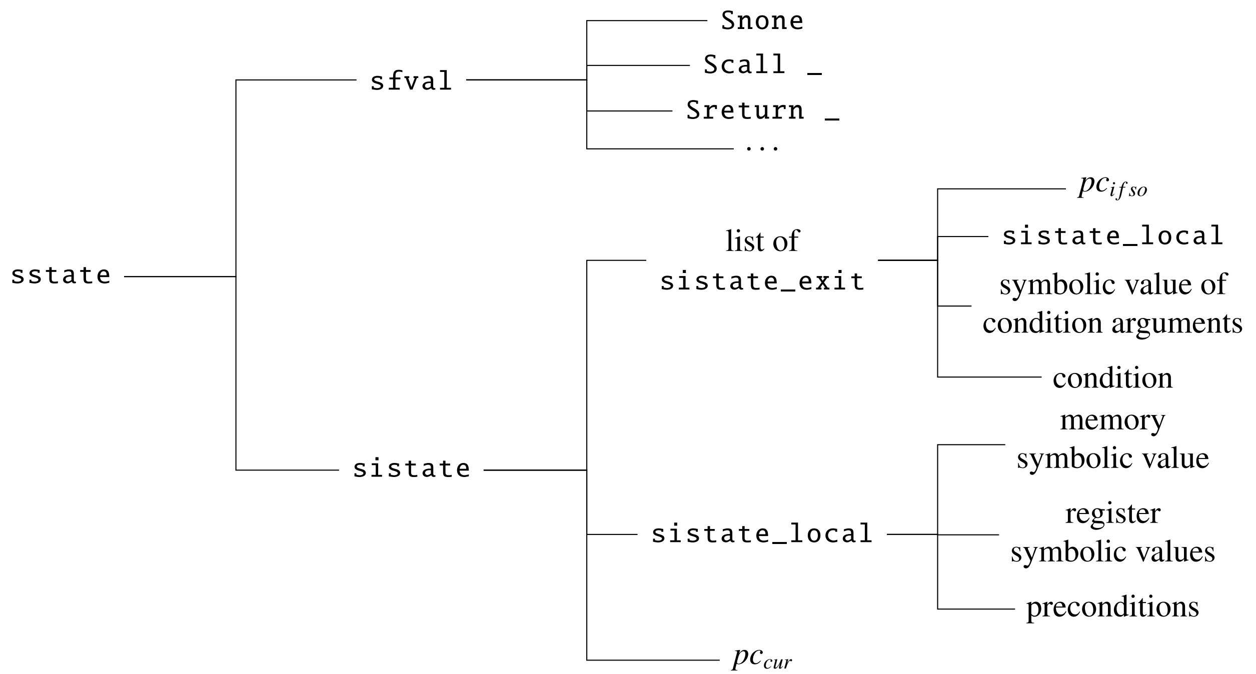


Figure 7.2.: Tree representation of a `sstate`

```
1 Definition ssem_local (ge: RTL.genv) (sp:val) (st: sistate_local) (rs0: regset) (m0: mem)
2   (rs: regset) (m: mem): Prop :=
3   st.(si_pre) ge sp rs0 m0
4   ∧ seval_smem ge sp st.(si_smem) rs0 m0 = Some m
5   ∧ ∀ (r:reg), seval_sval ge sp (st.(si_sreg) r) rs0 m0 = Some (rs#r)
```

To later reason on whether the superblock execution succeeded or not, we also need to define a relation defining whether the local symbolic state's evaluation is a success. Below is a predicate that holds if one of the evaluations of `ssem_local` failed: either the precondition was not met, or the memory evaluation failed, or one of the register evaluations failed.

```
1 Definition sabort_local (ge: RTL.genv) (sp:val) (st: sistate_local) (rs0: regset)
2   (m0: mem): Prop :=
3   ~(st.(si_pre) ge sp rs0 m0)
4   ∨ seval_smem ge sp st.(si_smem) rs0 m0 = None
5   ∨ ∃ (r: reg), seval_sval ge sp (st.(si_sreg) r) rs0 m0 = None
```

Throughout the rest of this chapter, I will refer to such relations as *abort semantics*.

Exit Internal Symbolic State

A `sistate_exit` encodes what happens if the condition of the early exit evaluates to `true`, as well as the symbolic value of the condition:

```
Record sistate_exit := mk_sistate_exit
  { si_cond: condition; si_scondargs: list_sval; si_elocal: sistate_local;
    si_ifso: node }
```

`si_cond` is the condition to evaluate (could be "greater than" or "less than" for instance), `si_scondargs` are the arguments of that condition stored as symbolic values, `si_elocal` is the local symbolic state (of registers and memory) when exiting the superblock, and `si_ifso` is a pointer to the next superblock to execute when the condition evaluates to `true`.

It is essential to store the local symbolic state of the early exit and to store under which terms the early exit is taken: for the scheduled superblock to simulate the original, given the same initial values, their execution paths must go through the same early exit. Storing (and then comparing) the symbolic values of the condition arguments is one way to ensure this.

Semantically speaking, `(ssem_exit ge sp ext rs m rs' m' pc')` holds if, given initial registers/memory values (rs, m), an exit symbolic state *ext*, and given that the condition of *ext* evaluates to `true`[3] , then evaluating the local symbolic state of *ext* would give the register and memory states (rs, m), and the early exit *ext* points to pc' (to be later interpreted as the entry of the next superblock to execute).

```
Definition ssem_exit (ge: RTL.genv) (sp: val) (ext: sistate_exit) (rs: regset) (m: mem)
  rs' m' (pc': node) : Prop :=
    seval_condition ge sp (si_cond ext) (si_scondargs ext) ext.(si_elocal).(si_smem) rs m
      = Some true
 ∧ ssem_local ge sp (si_elocal ext) rs m rs' m'
 ∧ (si_ifso ext) = pc'
```

Regarding the abort semantics of the early exit, the first step to consider is the evaluation of the condition arguments. If the condition evaluation fails, then the early-exit evaluation fails. If the condition is evaluated to `true`, then the early-exit evaluation only fails if the evaluation of its associated local symbolic state fails. Otherwise, if the condition is evaluated to `false`, then the associated local symbolic state is discarded: there is thus no failure on the early-exit evaluation. This results in the definition below:

```
Definition sabort_exit (ge: RTL.genv) (sp: val) (ext: sistate_exit) (rs: regset) (m: mem)
  : Prop :=
  let sev_cond := seval_condition ge sp (si_cond ext) (si_scondargs ext)
    ext.(si_elocal).(si_smem) rs m in
  sev_cond = None
  ∨ (sev_cond = Some true ∧ sabort_local ge sp ext.(si_elocal) rs m)
```

[3]One technical remark here is how `seval_condition` requires the memory for evaluation. In CompCert, the memory data structure is required to evaluate a pointer comparison: they must point to valid memory blocks. This information is only available within the memory data structure.

Internal Symbolic State

We can now define the actual internal symbolic state. This is our running state when executing the superblock symbolically: it holds information about a partial execution of the superblock up to a certain point (named `si_pc` below).

```
Record sistate := { si_pc: node; si_exits: list sistate_exit; si_local: sistate_local }
```

`si_pc` is a pointer to the next node to execute in the superblock. `si_local` is the local symbolic state, holding the symbolic values of registers and memory. Finally, `si_exits` is the list of all symbolic exit states encountered so far. For technical reasons, this list is stored in reverse order: the first element of the list is the last exit state of the superblock.

When executing a superblock, it should be noted that at most one exit state condition of `si_exits` will evaluate to `true`. Indeed, as soon as such a condition evaluates to `true`, the current superblock execution stops to then start the execution of another superblock. Also, another noteworthy point is that if a given exit state condition evaluates to `true`, then we have the guarantee that all the exit conditions before it evaluated to `false`. To express these properties, we introduce the following definitions `all_fallthrough` and `all_fallthrough_upto_exit`:

```
Definition all_fallthrough ge sp (lx: list sistate_exit) rs0 m0: Prop :=
  ∀ ext, List.In ext lx →
  seval_condition ge sp ext.(si_cond) ext.(si_scondargs)
    ext.(si_elocal).(si_smem) rs0 m0 = Some false

Definition all_fallthrough_upto_exit ge sp ext lx' lx rs m : Prop :=
  is_tail (ext::lx') lx ∧ all_fallthrough ge sp lx' rs m
```

`(all_fallthrough ge sp lx rs0 m0)` holds if, given initial `(rs0, m0)`, each exit condition of `lx` evaluates to `false`. `(all_fallthrough_upto_exit ge sp ext lx' lx rs m)` holds if, given initial `(rs, m)`, all the exit conditions of `lx'` evaluated to `false`, and `lx'` represents all the exit conditions up to `ext` within `lx`. This last part is ensured through a `is_tail` relation.

Figure 7.3 depicts a graphical representation of the `all_fallthrough_upto_exit` relation: from right to left (the last element of `si_exits` is the first exit state of the superblock), all the conditions up to `ext` evaluated to `false`, and the relation gives no information about the other conditions.

Now that we have these two definitions, we define the semantics of an internal symbolic state. `(ssem_internal ge sp st rs m is)` holds if, given initial `(rs, m)`, evaluating the internal symbolic state `st` would lead to the RTLpath `istate is`. This reflects a partial

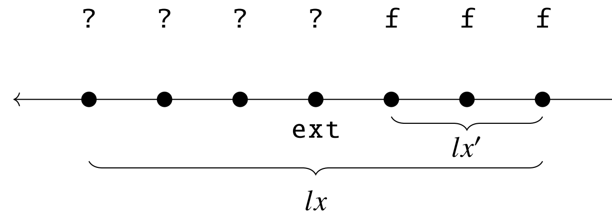


Figure 7.3.: Graphical representation of `all_fallthrough_upto_exit` relation. The superblock is executed from right to left; all conditions up to `ext` have been evaluated to `false` (`f` in the drawing). The other conditions remain to be evaluated.

evaluation of the superblock: we are evaluating all the past instructions since the superblock entry, all at once.

```
Definition ssem_internal (ge: RTL.genv) (sp:val) (st: sistate) (rs: regset) (m: mem)
  (is: istate): Prop :=
  if (is.(icontinue)) then
    ssem_local ge sp st.(si_local) rs m is.(irs) is.(imem)
    ∧ st.(si_pc) = is.(ipc)
    ∧ all_fallthrough ge sp st.(si_exits) rs m
  else ∃ ext lx,
    ssem_exit ge sp ext rs m is.(irs) is.(imem) is.(ipc)
    ∧ all_fallthrough_upto_exit ge sp ext lx st.(si_exits) rs m
```

The relation is defined based on the value of `(icontinue is)` which represents (according to RTLpath semantics) whether an early branch was taken or not.

- Either no early branch was taken (`icontinue` is true): the relation holds only if the local symbolic state evaluates to the registers and memory values from is, the identifier of the next instruction to execute is equal to the `pc` of is, and all the conditions of the early exits evaluated to `false`.
- Or an early branch was taken. The relation then only holds if there exists an exit symbolic state such that:
  - All the branches prior to it evaluated to false.
  - The registers, memory and PC of the `istate` obey the `ssem_exit` relation with that exit symbolic state.

I schematize in figure 7.4 the `ssem_internal` relation. Both left and right parts feature the same superblock. On the left, none of the symbolic exit state conditions was evaluated to `true`, and the local state evaluates to $(irs, imem)$: there is then a `ssem_internal` relation to the internal state $(irs, imem, pc, true)$. On the right, the two first symbolic exit state conditions were evaluated to false, but not the third, which presented a `ssem_exit` relation to $(irs, imem, pc)$. There is then a `ssem_internal` relation to $(irs, imem, pc, false)$: according to RTLpath semantics, the superblock execution is about to stop (to move on to another superblock).

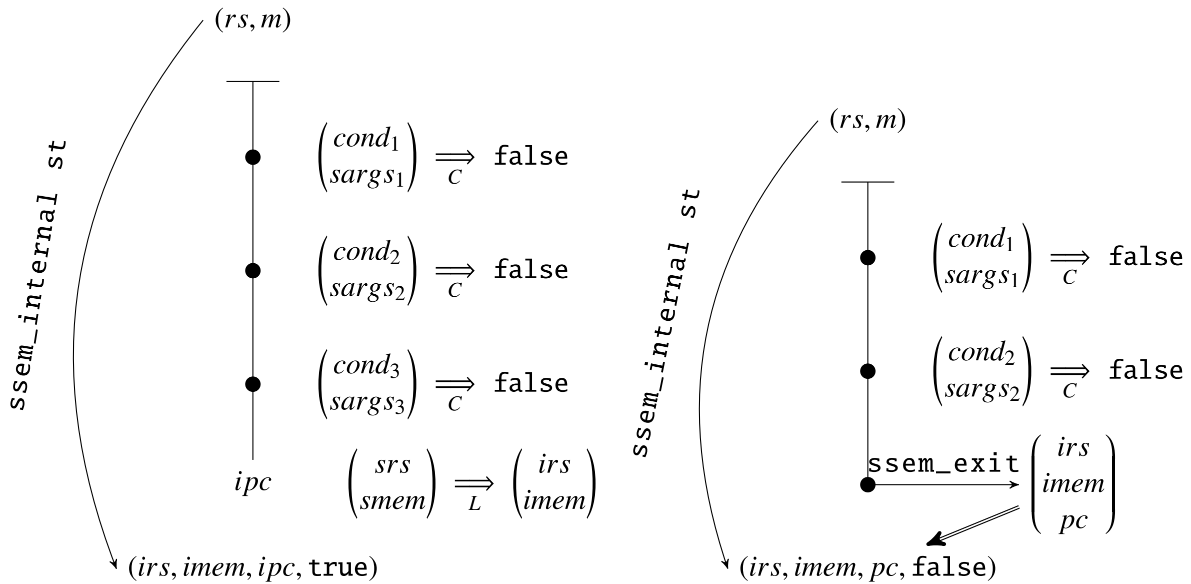


Figure 7.4.: Summary of `ssem_internal` semantics. The same superblock is presented in two scenarios: on the left, no early exit was triggered so far, so we continue to execute the superblock. On the right, one early exit was encountered: the next internal state will then have `icontinue` as false. The symbols $\underset{C}{\Longrightarrow}$ and $\underset{L}{\Longrightarrow}$ respectively denotate the condition and local (registers and memory) evaluation functions (over symbolic values).

The abort semantics for the internal symbolic state follows the same kind of reasoning as for the exit state: either all conditions evaluated to `false`, but the evaluation of the local symbolic state aborted, or only a certain number of exit state conditions evaluated to `false`, but there was an abort during the evaluation of the following exit state.

```
1  Definition sabort (ge: RTL.genv) (sp: val) (st: sistate) (rs: regset) (m: mem): Prop :=
2    (* No early exit was met but we aborted on the si_local *)
3    (all_fallthrough ge sp st.(si_exits) rs m ∧ sabort_local ge sp st.(si_local) rs m)
4    (* OR we aborted on an evaluation of one of the early exits *)
5    ∨ (∃ ext lx, all_fallthrough_upto_exit ge sp ext lx st.(si_exits) rs m
6        ∧ sabort_exit ge sp ext rs m)
```

Abstract Symbolic Final State

As was described in section 5.2.1, in RTLpath, we partition the instructions between those that are schedulable and those that are not—in the context of scheduling, we call the latter *final instructions*. These final instructions can be either `Icall`, `Itailcall`, `Ibuiltin`, `Ijumptable` or `Ireturn`.

Since they have arguments whose values might depend on the execution, we have to consider them in our symbolic execution. This is handled by the `sfval` definition below:

```
Inductive sfval :=
  | Snone
  | Scall (sig:signature) (svos: sval + ident) (lsv:list_sval) (res:reg) (pc:node)
  | Stailcall: signature → sval + ident → list_sval → sfval
  | Sbuiltin (ef:external_function) (sargs: list (builtin_arg sval))
      (res: builtin_res reg) (pc:node)
  | Sjumptable (sv: sval) (tbl: list node)
  | Sreturn: option sval → sfval
```

A symbolic final value is either `Snone` (which indicates that no such final instruction was found), or it represents the final instruction of the superblock directly. The arguments are encoded as variants of the type of symbolic values `sval`, mostly following the definitions of their RTL variants.

The evaluation semantics of symbolic final values are a bit special. I will start by illustrating the example of the `Scall` evaluation.

```
  | exec_Scall rs m sig svos lsv args res pc fd:
      sfind_function pge ge sp svos rs0 m0 = Some fd →
      funsig fd = sig →
      seval_list_sval ge sp lsv rs0 m0 = Some args →
      ssem_final pge ge sp npc stack f rs0 m0 (Scall sig svos lsv res pc) rs m
        E0 (Callstate (Stackframe res f sp pc rs :: stack) fd args m)
```

The first part of `Scall` evaluation involves finding the function to be called: this is done through the `svos` argument, which is of the union type `sval + ident`. Either the function symbol is directly encoded as an identifier, or the function's address resides in a symbolic value (this would be the case when generating a program with function pointers). If this is the latter, then the symbolic value must be evaluated with the initial $(rs0, m0)$ values, which is what `sfind_function` does.

In RTL, the semantics of making a call involves creating a new stack frame containing, in particular, the state of registers when the call was made.[4] We must then have access to the register state right before the call was made: here, this is represented by the rs parameter.

In general, most symbolic final values require having access to both the initial register and memory states to evaluate the symbolic values of their arguments and the register and memory states from the superblock execution up to the final instruction. Their semantics is then represented by the relation `(ssem_final pge ge sp npc stk f rs0 m0 sfv rs m t s)` which holds if, given npc the PC of the next superblock (only matters if the symbolic final value is `Snone`), $(rs0, m0)$ the initial register and memory states, (rs, m) the register and memory states reached

[4]In a sense, this is RTL's way of ensuring that all registers are saved when performing a call.

right before the final instruction of the superblock, evaluating sfv would lead to the RTLpath state s, and would emit the trace t.[5]

We illustrated this relation for `Scall`, but the other symbolic final values go through a similar treatment.

It should be noted that since these instructions are not scheduled, and since the register/memory states before their execution will be evaluated (see the following subsection), there is no need to exhibit abort semantics.

Abstract Symbolic State

Now that we have all the building blocks, a symbolic state `sstate` is simply a symbolic final value combined with the symbolic internal state from right before the final instruction.

```
1  Record sstate := { internal:> sistate; final: sfval }
```

In terms of semantics, the relation `(ssem pge ge sp st stk f rs0 m0 t s)` holds if, given initial memory and register states $(rs0, m0)$, evaluating the symbolic state st leads to the RTLpath state s and emits a trace t.

```
1  Inductive ssem pge (ge: RTL.genv) (sp:val) (st: sstate) stack f
2    (rs0: regset) (m0: mem): trace → state → Prop :=
3    | ssem_early is:
4       is.(icontinue) = false →
5       ssem_internal ge sp st rs0 m0 is →
6       ssem pge ge sp st stack f rs0 m0 E0 (State stack f sp is.(ipc)
7        is.(irs) is.(imem))
8    | ssem_normal is t s:
9       is.(icontinue) = true →
10      ssem_internal ge sp st rs0 m0 is →
11      ssem_final pge ge sp st.(si_pc) stack f rs0 m0 st.(final)
12       is.(irs) is.(imem) t s →
13      ssem pge ge sp st stack f rs0 m0 t s
```

I schematize this relation in figure 7.5. The `ssem` relation is a lot like the `path_step` relation: either an early exit was encountered, or we went through the whole superblock, in which case we only have to execute the final instruction.

Since the generalization from internal symbolic state to symbolic state does not introduce additional abort semantics, we do not introduce any abort semantics for a symbolic state: the later proofs only reason on whether an internal symbolic state evaluation aborted or not.

[5]The other parameters of the relation, such as `pge` or `stk` are there to forward information about the execution environment. These mainly reflect RTL semantics.

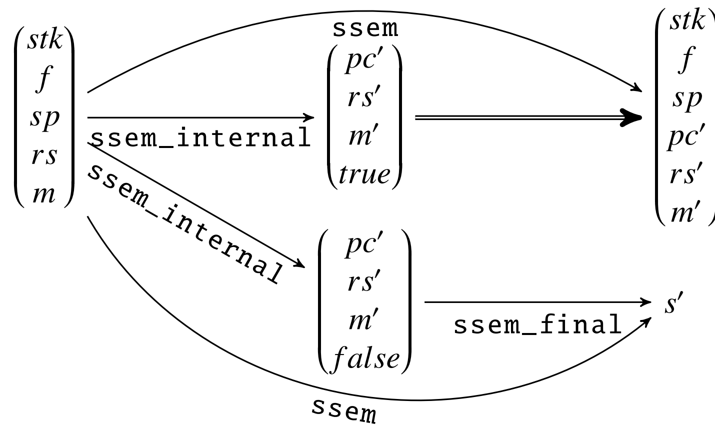


Figure 7.5.: Evaluating a symbolic state with the `ssem` relation.

7.1.2 Superblock Symbolic Execution

Now that we have a data structure to hold symbolic information about superblocks, with associated evaluation semantics, we need to devise a way to generate such data. This is done by going through each instruction from the superblock, one by one, accumulating and updating the information of a symbolic internal state as we go.

Internal Symbolic Execution

Our internal symbolic execution function, `siexec_inst`, has the type `instruction -> sistate -> option sistate`: given an instruction and a current internal symbolic state, it returns either the internal symbolic state after execution or `None` in the case that the instruction to be executed is actually a final instruction.

The new internal symbolic state is constructed as such:

- If the instruction sets a register, update the register state by its new symbolic value sv, and add the precondition (in the local symbolic state) that its evaluation (given an initial (rs, m)) should not fail.
- Do the same if an instruction changes the memory: the evaluation of the new memory should not fail.
- If a conditional branch is encountered, create a new symbolic exit state by recording the symbolic value of the condition arguments[6] and the local symbolic state.

[6]The condition arguments of a `Icond` are registers, so this just entails recording their symbolic values.

Abstract Symbolic Execution of the Final Instruction

This is done through the function `sexec_final`, of type `instruction -> sistate_local -> sfval`. It requires the current local internal symbolic state to capture the symbolic value of the potential arguments of the final instruction. Its implementation is mechanical; it follows directly the `sfval` definition.

Abstract Symbolic Execution of a Superblock

Executing a superblock symbolically simply amounts to executing all of its instructions but the final, and then executing the final instruction. Since the final instruction might be schedulable, the `None` return value of `siexec_inst` indicates that `sexec_final` should be used instead.

This is done by the function `sexec` below. (`siexec_path size`) is a function that executes symbolically a number *size* of instructions with `siexec_inst`.

```
1 Definition sexec (f: function) (pc:node): option sstate :=
2   SOME path ← (fn_path f)!pc IN
3   SOME st ← siexec_path path.(psize) f (init_sistate pc) IN
4   SOME i ← (fn_code f)!(st.(si_pc)) IN
5   Some (match siexec_inst i st with
6        | Some st' ⇒ {| internal := st'; final := Snone |}
7        | None ⇒ {| internal := st; final := sexec_final i st.(si_local) |}
8        end)
```

First, the superblock information is retrieved from the pathmap. Then, we execute symbolically a number `path.(psize)` of instructions: this gives an internal symbolic state `st` which contains, in particular, the PC of the final instruction. That final instruction is then executed: first, we try to execute it as if it was schedulable with `siexec_inst`. If it succeeds, we update the symbolic internal state accordingly and set the final symbolic value to `Snone`. If it fails, we keep the old symbolic internal state, and we use `sexec_final` to compute the symbolic final value.

This gives us a superblock symbolic state, representing the execution of a whole superblock.

7.1.3 Bisimulation between RTLpath and Symbolic-Execution Semantics

An important intermediate theorem of our formalization states that the RTLpath semantics (`path_step`) are bisimulated by the semantics of symbolic states (`ssem`), the latter being computed by symbolic execution (`sexec`).

We formulate the bisimulation as the observational equivalence between RTLpath semantics and symbolic execution semantics and prove it in two parts. First, we prove that, if there is a `path_step`, then we can execute the superblock symbolically and evaluate the result with `ssem` to get the same final state. This proves that the symbolic execution semantics are faithful to the RTLpath semantics. Then, we prove the reverse: if we executed a superblock symbolically, and evaluated it to a certain final state, then we should be able to construct a `path_step` leading to the same state, thus proving that the symbolic execution semantics is complete in regards to RTLpath semantics.

In both directions of the simulation, we have two subcases: the case where the superblock execution (or evaluation) succeeds and the case where it fails (aborts execution). Not only do we have to prove that a successful execution in one part (RTLpath or symbolic) leads to the same successful execution in the other part, but we also have to prove that failures are adequately reflected.

We will start by describing some key lemmas used for ensuring a kind of bisimulation at the internal level of a superblock. Then we will describe the two theorems that, together, ensure a bisimulation between RTLpath and symbolic execution.

Bisimulation at the Internal Level

In order to express lemmas reasoning on both the success and abort cases, we introduce two helper definitions: `ssem_internal_opt2` and `ssem_internal_opt`.

```
1  Definition ssem_internal_opt ge sp (st: sistate) rs0 m0
2    (ois: option istate): Prop :=
3    match ois with
4    | Some is ⇒ ssem_internal ge sp st rs0 m0 is
5    | None ⇒ sabort ge sp st rs0 m0
6    end
7
8  Definition ssem_internal_opt2 ge sp (ost: option sistate) rs0 m0
9    (ois: option istate) : Prop :=
10   match ost with
11   | Some st ⇒ ssem_internal_opt ge sp st rs0 m0 ois
12   | None ⇒ ois=None
13   end
```

I recall that `istep` is a function which returns an `option istate`: it may return `None` either if the execution of the instruction failed (an error was encountered) or if we tried to execute a final instruction. *ois* is here to be interpreted as the result of an `istep`. Similarly, `siexec_inst` returns `None` if it attempted to execute symbolically a final instruction; *ost* is to be interpreted as the result of a `siexec_inst`.

We then express the following lemma, which gives a kind of bisimulation at the level of internal instructions:

```
Lemma siexec_inst_correct ge sp i st rs0 m0 rs m:
  ssem_local ge sp st.(si_local) rs0 m0 rs m →
  all_fallthrough ge sp st.(si_exits) rs0 m0 →
  ssem_internal_opt2 ge sp (siexec_inst i st) rs0 m0 (istep ge i sp rs m)
```

In the context of this lemma, we are executing symbolically a superblock instruction-per-instruction. To an existing internal symbolic state *st*, we are about to add the result of the symbolic execution of instruction *i*. We have two premises, ensuring that evaluating the internal symbolic state *st* indeed would bring us to the point of executing *i*:

- Evaluating the local symbolic state must succeed and lead to a certain register/memory state (rs, m).
- All the symbolic exit state conditions encountered so far have to evaluate to `false`.

Under these two conditions, `ssem_internal_opt2` bridges `siexec_inst` with `istep` in the following way:

- If `siexec_inst` returned `None` (because it attempted to execute a final instruction), then `istep` must also have returned `None`.
- Or else, if `istep` returned `None` (because of an execution failure), then the symbolic state must satisfy `sabort` (the superblock evaluation must fail as well).
- Or else, then the symbolic state must satisfy a `ssem_internal` relation: both the `istep` execution and the evaluation of the symbolic state augmented by `siexec_inst` must lead to the same internal state.

This lemma is proved by case analysis on the instruction `i` to be executed symbolically. The proof is mechanical for most instructions. In the case of a conditional branch, if the `istep` is successful, we have to prove that the symbolic condition evaluation will reflect the execution flow of `istep` on the register/memory state (rs, m). But since we have the first premise in the lemma, we know that evaluating the superblock up to instruction `i` with initial states $(rs0, m0)$ gives the same register/memory state (rs, m). So we can deduce that the symbolic values of the condition arguments will evaluate to the same result as the concrete (in RTLpath) condition evaluation, thus guaranteeing that the execution flow will be the same between the RTLpath execution and the superblock evaluation.

This lemma is then used for proving the more general lemma below, reasoning on the execution of all instructions but the last one:

```
Lemma siexec_path_correct_true ge sp n (f:function) rs0 m0: ∀ st is,
  is.(icontinue)=true →
```

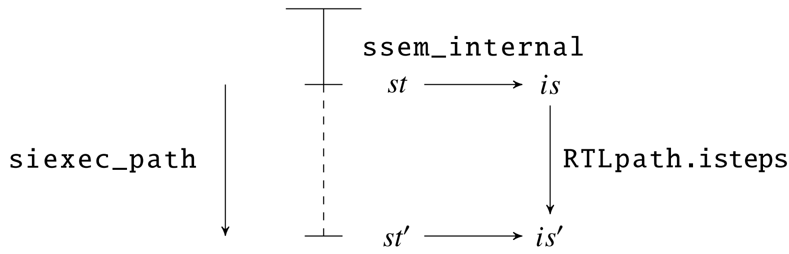


Figure 7.6.: Graphical representation of `siexec_path_correct` lemma. The superblock is symbolically executed up to a certain point, giving an internal symbolic state *st*. The evaluation of this internal symbolic state succeeds and gives an internal state *is*. The lemma proves then a correspondence between the remaining symbolic execution via `siexec_path`, and `isteps`.

```
3    ssem_internal ge sp st rs0 m0 is →
4    nth_default_succ (fn_code f) n st.(si_pc) <> None →
5    ssem_internal_opt2 ge sp (siexec_path n f st) rs0 m0
6                           (isteps ge n f sp is.(irs) is.(imem) is.(ipc))
```

This lemma states that, given an initial symbolic internal state *st*, and its successful evaluation *is* (assuming the `icontinue` field is true: all the conditions evaluated to `false`), and if the *n*-th default successor of *st* exists (this can be deduced from the property that each path from RTLpath is well-founded), then evaluating the *n* next instructions (in addition to the existing internal symbolic state *st*) should lead to the same result as the actual RTLpath execution. This is summarized in figure 7.6 for a more graphical representation of the lemma.

Bisimulation at the Superblock Level

With the above lemmas, we can deduce the `sexec_correct` and `sexec_exact` theorems below:

```
1  Theorem sexec_correct f pc pge ge sp path stack rs m t s:
2    (fn_path f)!pc = Some path →
3    path_step ge pge path.(psize) stack f sp rs m pc t s →
4    ∃ st, sexec f pc = Some st ∧ ssem pge ge sp st stack f rs m t s
5
6  Theorem sexec_exact f pc pge ge sp path stack st rs m t s1:
7    (fn_path f)!pc = Some path →
8    sexec f pc = Some st →
9    ssem pge ge sp st stack f rs m t s1 →
10   ∃ s2, path_step ge pge path.(psize) stack f sp rs m pc t s2 ∧
11             equiv_state s1 s2
```

These theorems, together, express a bisimulation between RTLpath and symbolic execution. The `siexec_path_correct` lemma gives a similar property for the execution of the superblock up to the last instruction, so the remaining part to prove is what happens for the execution of the last

instruction. This is proven by case analysis—some parts of the proof are non-trivial and a bit tedious due to some corner cases, so I will not detail them here. I refer the reader to the available source code for more information.

One thing to note is that, for the `sexec_exact` theorem, the states $s2$ (result of the actual execution) and $s1$ (result of the superblock evaluation) are not structurally the same: indeed, if we take an original code `r0 := 1; r0 := 2`, the value of the regset after RTLpath execution would look like $rs_{init}[0 \mapsto 1][0 \mapsto 2]$, while its symbolic execution semantics would rather be just $rs_{init}[0 \mapsto 2]$. We then defined a relation `equiv_state` that is a structural equality, except for the `regset` where only functional equality is required to satisfy the relation.

7.2 Proving Lockstep Simulation of Scheduling from Sufficient Conditions on Abstract Symbolic Executions

Now that we have a symbolic execution for superblocks and bisimulation properties with RTLpath, the only missing pieces are how to compare two symbolic states and which properties need to be verified to prove a lockstep simulation for the scheduling.

In this section, I present the latter: this will give a simulation property that will have to be fulfilled by the simulation test; the actual hash-consed simulation test is described in the next section. First, I detail in 7.2.1 the definitions of symbolic simulation that we use to reason semantically on whether one superblock simulates the other. Then, I use these definitions in 7.2.2 to outline properties that the verifier must fulfill to ensure a forward simulation from RTLpath to itself, to formally prove the correction of superblock scheduling.

7.2.1 Simulation of Superblocks

To prove a lockstep simulation between the original code and the scheduled code, we would like to split the work in two: first, we want to find some properties such that, when fulfilled, a lockstep simulation can be deduced; second, we want to implement a verifier such that its success would imply the verification of the simulation properties. These properties need to be complete and verifiable.

First, we will introduce a notion of simulation at the RTLpath `istate` level: under which conditions can we say that two `istates` are equivalent, for the sake of verifying superblock

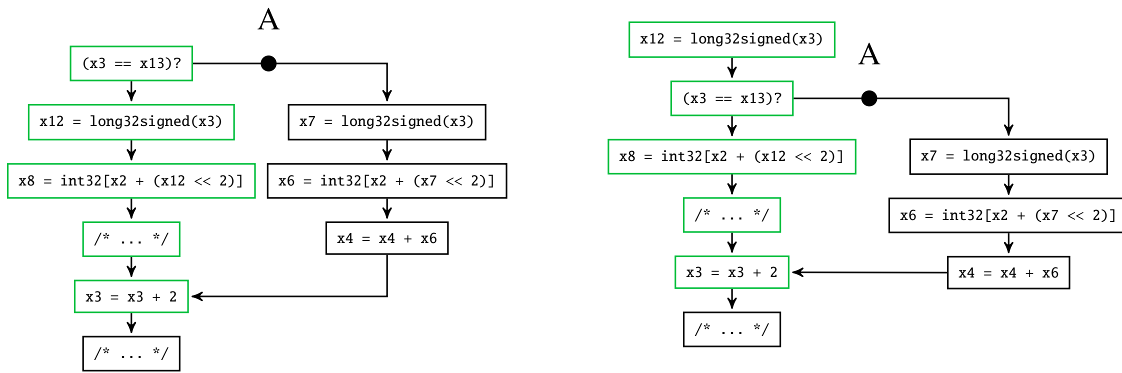


Figure 7.7.: Two semantically equivalent superblocks (in green) but who reach different states when taking the branch A.

scheduling. Then, we will use this notion to choose a symbolic simulation definition for two symbolic states (the original and the scheduled code, executed symbolically).

Simulation at the RTLpath Level

A first issue to consider when designing a simulation property at the RTLpath level is that the entry node value (the index from which we get the RTLpath superblock) might change between the original and the scheduled code. Indeed, if a superblock comprised of instructions `[1; 2; 3]` is rescheduled into `[2; 1; 3]`, then the path entry of the scheduled superblock will be 2 instead of 1. So we need a way to keep track of which is the original entry of the scheduled superblock. Much like for verifying code duplication (see chapter 8), we use a reverse mapping that is returned by the scheduling oracle. This reverse mapping also guides the verification effort and ensures that each superblock from the pathmap is verified.

Another difficulty is that scheduling an instruction before a conditional branch does not give the same `istate` when executing the superblock if the branch is taken. For instance, the two superblocks shown in figure 7.7 are semantically equivalent: no matter which path is taken, the result after execution will be the same. However, the states achieved when taking branch A will not be the same: the right superblock will have an additional assignment for `x12`.

We already have introduced a notion of state simulation at the RTLpath level that allows for certain registers to have different values between the original and the scheduled code: this is the `eqlive_states` definition from section 5.3.3. We have proven the `path_step_eqlive` lemma (from that section), which states that, supposing a given early exit is executed, then only the live registers from the exit branch need to have the same value to ensure the same execution.

First, we define `istate_simulive` as a structural equality modulo liveness between two internal states:

```
Definition istate_simulive alive (is1 is2: istate): Prop :=
     is1.(icontinue) = is2.(icontinue)
     ∧ eqlive_reg alive is1.(irs) is2.(irs)
     ∧ is1.(imem) = is2.(imem)
```

Then, we extend this definition to include: the liveness information to use when taking an early exit; and also the required property that the exit branches from the scheduled and the original code must obey the reverse mapping (named `dm` in the definition below).

```
Definition istate_simu f (dm: PTree.t node) is1 is2: Prop :=
  if is1.(icontinue) then
     istate_simulive (fun _ ⇒ True) is1 is2
  else
     ∃ path, f.(fn_path)!(is1.(ipc)) = Some path
     ∧ istate_simulive (fun r ⇒ Regset.In r path.(input_regs)) is1 is2
     ∧ dm!(is2.(ipc)) = Some is1.(ipc)
```

Two cases are distinguished, based on the value of `icontinue` from $is1$:

- Either the superblock continues the execution: in that case, we must have a strict structural equality between $is1$ and $is2$.[7]
- Or we are taking (right now) an early exit: in that case, there must be a valid next superblock to execute chosen by $is1.ipc$, both $is2.ipc$ and $is1.ipc$ must obey the reverse map, and $is1$ and $is2$ must be equal modulo the liveness of the superblock to be executed in the original code.

This brings a definition of semantic simulation for partial execution of a superblock for concrete RTLpath executions.

Simulation for Symbolic States

I will follow a bottom-up presentation to describe our simulation definition for symbolic states, starting from the actual symbolic superblock simulation and then crawling upwards to the underlying definitions.

[7] Actually, it does not have to be strict: verifying it modulo liveness of the current superblock would be enough. For superblock scheduling, making it strict made the proof easier. Recently, Léo Gourdin and Sylvain Boulmé modified it to be verified modulo liveness, which enabled them to implement rewriting rules within superblock scheduling [89].

Given a reverse mapping *dm*, two RTLpath functions $f1$ and $f2$ (the original and the scheduled function), and two entry nodes $pc1$ and $pc2$ from which we extract the superblocks to compare, we define a simulation as such:

```
Definition sexec_simu dm (f1 f2: RTLpath.function) pc1 pc2: Prop :=
    ∀ st1, sexec f1 pc1 = Some st1 →
    ∃ st2, sexec f2 pc2 = Some st2
      ∧ ∀ ctx, sstate_simu dm f1 st1 st2 ctx
```

The superblock starting at node $pc2$ simulates the one starting at $pc1$ if, given a successful symbolic execution of the first, there exists a successful symbolic execution of the second, such that for any environment context (detailed later), the second symbolic state simulates the first one. In a way, this defines a property of lockstep simulation: the intent is that, when combined with the bisimulation property of RTLpath with regards to symbolic execution, this should imply the forward simulation for the whole transformation.

The environment context contains all the elements that are necessary to evaluate a superblock symbolic state, ensure that the global environments (used to deduce a function out of a symbol) of the scheduled and the original function match, and record that the liveness information of the original pathmap is correct:

```
Record simu_proof_context {f1: RTLpath.function} := {
   liveness_hyps: liveness_ok_function f1;
   the_ge1: RTL.genv;
   the_ge2: RTL.genv;
   genv_match: ∀ s, Genv.find_symbol the_ge1 s = Genv.find_symbol the_ge2 s;
   the_sp: val;
   the_rs0: regset;
   the_m0: mem
}
```

Now, I recall that a symbolic state contains a symbolic final state (which only matters if no early exit is taken during the concrete execution) and an internal symbolic state which is used to rule out whether we exit early (and in which state) or we continue through the final instruction. So we take both into account for the definition of a symbolic state simulation, `sstate_simu`:

```
Definition sstate_simu dm f (s1 s2: sstate) (ctx: simu_proof_context f): Prop :=
       sistate_simu dm f s1.(internal) s2.(internal) ctx
    ∧ ∀ is1,
           ssem_internal (the_ge1 ctx) (the_sp ctx) s1 (the_rs0 ctx) (the_m0 ctx) is1 →
           is1.(icontinue) = true →
           sfval_simu dm f s1.(si_pc) s2.(si_pc) ctx s1.(final) s2.(final)
```

This can be read as: given a reverse mapping *dm*, an original function f, and an execution context *ctx*, a symbolic state $s2$ simulates $s1$ if the simulation holds on their internal symbolic states, and,

if no early exit was taken (ensured through lines 4 and 5 of the definition), then the simulation also holds on their final symbolic values.

Line 4 is quite lengthy because it involves calling the `the_ge1` projection (and others) on `ctx`—since many of the further definitions will share this syntax cumbersomeness, I will write `ctx.ge1` instead of `(the_ge1 ctx)` and so on.

For the final symbolic-state simulation, since the final instructions (call, return, etc. . .) are not scheduled, we just have to check that they are of the same nature and that their argument evaluations are the same. Here is an example for `Sreturn` and `Snone`—the other final instructions are expressed similarly.

```
1  Inductive sfval_simu (dm: PTree.t node) (f: RTLpath.function) (opc1 opc2: node)
2    (ctx: simu_proof_context f): sfval → sfval → Prop :=
3    | Snone_simu:
4        dm!opc2 = Some opc1 →
5        sfval_simu dm f opc1 opc2 ctx Snone Snone
6    | Sreturn_simu_none: sfval_simu dm f opc1 opc2 ctx (Sreturn None) (Sreturn None)
7    | Sreturn_simu_some sv1 sv2:
8        (seval_sval ctx.ge1 ctx.sp sv1 ctx.rs0 ctx.m0)
9        = (seval_sval ctx.ge2 ctx.sp sv2 ctx.rs0 ctx.m0) →
10       sfval_simu dm f opc1 opc2 ctx (Sreturn (Some sv1)) (Sreturn (Some sv2))
```

It should be noted that, for `Snone`, we also check that the nodes indicating the next superblock to execute obey the reverse map, much like the code verifier for code duplication transformations from chapter 8.

Finally, the simulation for internal symbolic states is defined like this:

```
1  Definition sistate_simu (dm: PTree.t node) (f: RTLpath.function) (st1 st2: sistate)
2    (ctx: simu_proof_context f): Prop :=
3    ∀ is1, ssem_internal ctx.ge1 ctx.sp st1 ctx.rs0 ctx.m0 is1 →
4    ∃ is2, ssem_internal ctx.ge2 ctx.sp st2 ctx.rs0 ctx.m0 is2
5              ∧ istate_simu f dm is1 is2
```

*st*2 simulates *st*1 if, given a successful evaluation of *st*1 into the internal state *is*1, then the evaluation of *st*2 is also successful into an internal state *is*2, and *is*2 simulates *is*1 under the `istate_simu` relation (equality modulo liveness or just structural equality).

Lockstep Simulation on Symbolic States

We have defined simulation properties on symbolic states—but we still must prove they are sufficient to ensure a lockstep simulation within the symbolic-execution semantics.

First of all, we define a notion of `match_function`: an original function `f1` and a scheduled function `f2` match if the following holds:

```
Record match_function (dm: PTree.t node) (f1 f2: RTLpath.function): Prop := {
  preserv_fnsig: fn_sig f1 = fn_sig f2;
  preserv_fnparams: fn_params f1 = fn_params f2;
  preserv_fnstacksize: fn_stacksize f1 = fn_stacksize f2;
  preserv_entrypoint: dm!(f2.(fn_entrypoint)) = Some f1.(fn_entrypoint);
  dupmap_path_entry1: ∀ pc1 pc2, dm!pc2 = Some pc1 → path_entry (fn_path f1) pc1;
  dupmap_path_entry2: ∀ pc1 pc2, dm!pc2 = Some pc1 → path_entry (fn_path f2) pc2;
  dupmap_correct: ∀ pc1 pc2, dm!pc2 = Some pc1 → sexec_simu dm f1 f2 pc1 pc2;
}
```

This `match_function` ensures the following properties binding `f1` and `f2`:

- The signature, parameters, and stack size must be the same.
- Their respective entrypoints should obey the reverse map.
- Each pair `(pc2, pc1)` of the reverse map should correspond to two path entries: one in the original function, one in the scheduled function.
- Each pair `(pc2, pc1)` must follow a `sexec_simu` simulation—this is a way to say that each such pair must have gone through the verifier.

Another necessary definition that we need before expressing our simulation theorem is a notion of `match_states`:

```
Inductive match_states: state → state → Prop :=
  | match_states_intro dupmap st f sp pc rs1 rs2 m st' f' pc' path
      (STACKS: list_∀2 match_stackframes st st')
      (TRANSF: match_function dupmap f f')
      (DUPLIC: dupmap!pc' = Some pc)
      (LIVE: liveness_ok_function f)
      (PATH: f.(fn_path)!pc = Some path)
      (EQUIV: eqlive_reg (ext path.(input_regs)) rs1 rs2):
      match_states (State st f sp pc rs1 m) (State st' f' sp pc' rs2 m)
  | match_states_call st st' f f' args m
      (STACKS: list_∀2 match_stackframes st st')
      (TRANSF: match_fundef f f')
      (LIVE: liveness_ok_fundef f):
      match_states (Callstate st f args m) (Callstate st' f' args m)
  | match_states_return st st' v m
      (STACKS: list_∀2 match_stackframes st st'):
      match_states (Returnstate st v m) (Returnstate st' v m)
```

This definition follows a typical `match_states` you would find for RTL, except for the following points:

- The `match_function` defined above must hold between the original and translated functions.

- Each time we execute a path from a `State` of RTLpath, the PC of the scheduled version must obey the reverse map with the PC of the original version.
- The original function must have had its liveness information verified for all of its paths.
- The stack pointer and memory must be structurally equal; however, the registers must only coincide modulo the liveness of the path about to be executed.

We have also defined a `match_stackframes` with similar conditions.

We can then express below the main theorem of symbolic state simulation, `ssem_state_simu`:

```
Theorem ssem_sstate_simu dm f f' stk stk' sp st st' rs m t s:
  match_function dm f f' →
  liveness_ok_function f →
  list_∀2 match_stackframes stk stk' →
  ssem pge ge sp st stk f rs m t s →
  (∀ ctx: simu_proof_context f, sstate_simu dm f st st' ctx) →
  ∃ s', ssem tpge tge sp st' stk' f' rs m t s' ∧ match_states s s'
```

This theorem states that, given some premises about the context, if we evaluate successfully a symbolic state *st* into a state *s*, and that the symbolic states *st* and *st'* are in simulation (according to `sstate_simu` and independently of the execution context), then evaluating the symbolic state *st'* will be successful and lead to a state *s'* that will match the original state *s*. This ensures a lockstep simulation within the symbolic-execution semantics.

This theorem is proven directly by using the simulation properties, with several undetailed helper lemmas.

7.2.2 Simulation Test Specification and Superblock Scheduling Lockstep Simulation

Given two superblocks to compare, we defined in the last subsection a simulation property `sexec_simu`, as well as a theorem `ssem_state_simu` outlining a forward simulation within symbolic execution.

The `sexec_simu` property is actually acquired with the following lemma, `simu_check_correct`:

```
Lemma simu_check_correct dm f tf:
  simu_check dm f tf = OK tt →
  ∀ pc1 pc2, dm ! pc2 = Some pc1 →
  sexec_simu dm f tf pc1 pc2
```

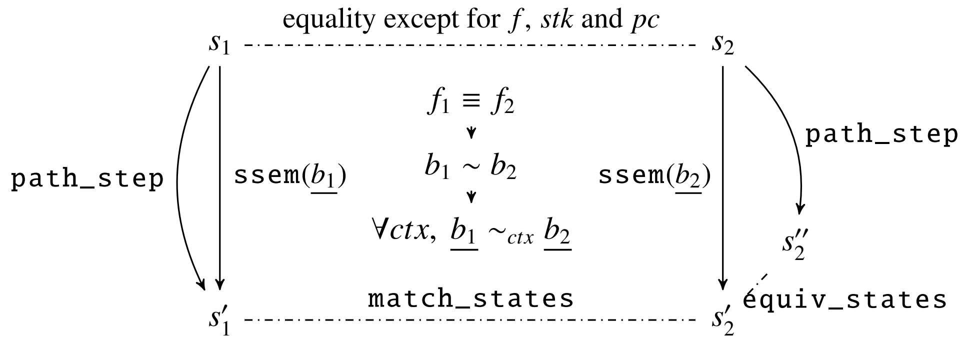


Figure 7.8.: Diagram used for proving the simulation in superblock scheduling. s_1 is actually the state made of $(f, sp, stk_1, rs, m, pc_1)$, and s_2 is made of $(f', sp, stk_2, rs, m, pc_2)$. $\underline{b_i}$ is the symbolic execution of the superblock from pc_i.

`simu_check` is the hash-consed simulation test (described section 7.3). The lemma states that if the original function f was compared (superblock-per-superblock) with the scheduled function tf, then, for each pair $(pc2, pc1)$ of the reverse mapping, we have the `sexec_simu` simulation. This lemma represents the required property from the simulation test to prove the forward simulation.

The forward simulation is itself mostly expressed by the following theorem.

```
1  Lemma exec_path_simulation dupmap path stk1 stk2 f1 f2 sp rs m pc1 pc2 t s1':
2    (fn_path f1)!pc1 = Some path →
3    path_step ge pge path.(psize) stk1 f1 sp rs m pc1 t s1' →
4    list_∀2 match_stackframes stk1 stk2 →
5    dupmap ! pc2 = Some pc1 →
6    match_function dupmap f1 f2 →
7    liveness_ok_function f1 →
8    ∃ path' s2', (fn_path f2)!pc2 = Some path'
9      ∧ path_step tge tpge path'.(psize) stk2 f2 sp rs m pc2 t s2'
10     ∧ match_states s1' s2'
```

To prove this theorem, we use the past theorems `sexec_correct` (simulation of RTLpath by symbolic execution), `sexec_exact` (simulation of symbolic execution by RTLpath), `simu_check_correct` (correctness of the verifier) and `ssem_state_simu` (forward simulation within symbolic execution). I expose in figure 7.8 the simulation diagram involved.

An attentive eye will notice that we consider the regsets from s_1 and s_2 to be structurally equal. In practice, they are only equal modulo liveness: we will see right after the theorem explanation that it does not matter for the proof.

Since there is a `path_step` from s_1 to s_1', we can apply the theorem `sexec_correct` and deduce the following: the symbolic execution of the superblock pointed by pc is successful and gives a symbolic state $\underline{b_1}$ whose evaluation results in the same state s_1'.

Using the properties from the `match_function`, we can deduce that pc_2 is a valid path entry of f_2, and we have a simulation `sexec_simu` between pc_1 and pc_2: this implies that the symbolic execution of the superblock pointed by pc_2 is successful and gives a symbolic state $\underline{b_2}$. We also get the property of a `sstate_simu` simulation between $\underline{b_1}$ and $\underline{b_2}$.

We have then all the premises required to use the theorem `ssem_state_simu` and deduce that the evaluation of $\underline{b_2}$ will be successful and result in a state s_2' which is in a `match_states` relation with s_1' (modulo liveness).

The final part is then to apply the theorem `sexec_exact`: since pc_2 was successfully symbolically executed to $\underline{b_2}$, and that $\underline{b_2}$ was itself successfully evaluated to s_2', then there is a state s_2'' in `equiv_state` relation with s_2'.

The actual theorem is then deduced from the use of one last small lemma, `match_states_equiv`, that states a transitivity between `match_states` and `equiv_states`:

```
1  Lemma match_states_equiv s1 s2 s3:
2    match_states s1 s2 → equiv_state s2 s3 →
3    match_states s1 s3
```

This proves the theorem `exec_path_simulation`. However, this is not exactly a forward simulation yet: we would like to reason on any initial s_1 and s_2 that are in `match_states`, not just those that are structurally equal.

This is where the lemma `path_step_eqlive` from chapter 5 comes in handy.

```
1  Lemma path_step_eqlive path stk1 f sp rs1 m pc t s1 stk2 rs2:
2    path_step ge pge (psize path) stk1 f sp rs1 m pc t s1 →
3    list_∀2 eqlive_stackframes stk1 stk2 →
4    eqlive_reg (ext (input_regs path)) rs1 rs2 →
5    liveness_ok_function f →
6    (fn_path f) ! pc = Some path →
7     ∃ s2, path_step ge pge (psize path) stk2 f sp rs2 m pc t s2 ∧ eqlive_states s1 s2
```

This lemma allows to transform a `path_step` with initial regset $rs1$ into a `path_step` with initial regset $rs2$ such that $rs2$ and $rs1$ are in structural equality modulo liveness. This can be used to generalize the lemma above for the more general case where the two states s_1 and s_2 are equivalent modulo liveness (following a `match_states` instead of structural equality).

The generalization is then used to prove a forward simulation for superblock scheduling, thus ensuring its correctness.

7.3 Refined Simulation Test with Hash-Consing

The previous section outlined a simulation property, `sexec_simu`, used for proving the forward simulation. We would like now to implement a test such that its success would imply the simulation. This test should take two symbolic states and deliver a boolean value: either `true` or `false` based on whether the two symbolic states are equivalent.

Using the abstraction level provided by our model of symbolic states allowed us to prove the bisimulation more easily; however, now we need to refine them to give an efficient implementation of their comparison. Indeed, a `sstate` contains terms whose values cannot be compared with any terminating executable function. For example, the `si_pre` component of a `sistate_local` cannot be compared with another `si_pre`: it requires to have the execution context, that we do not have at compile-time. `si_sreg` is also another field we cannot compare: we cannot check in finite time an infinite number of register values.

Some fields, however, like `si_smem`, could be compared. But we have an additional issue, first discovered by Tristan and Leroy [95]: in the worst case, we have an exponential blow-up due to the tree structure of the memory representation (we could end up comparing identical subtrees several times in different places of the tree). Also, our symbolic states do not allow trapping loads to be turned into non-trapping loads: they will not be structurally the same. Finally, in CompCert, some operations like pointer comparison depend on memory: they need to know which memory blocks were allocated. We remove this dependency in our refined values by exploiting the property that the instructions inside superblocks never modify the memory allocation.

The refined version should have comparable data structures. Still, it should also include hash-consing to prevent the exponential blow-up and also provide a way to canonize values (such as trapping loads) to accept certain transformations on the level of symbolic values. We do not want to define whole new semantics for these new types, so we will define projections and refinement relations that will link a refined symbolic state *hst* and its semantically equivalent symbolic state *st*.

The hash-consing method used is the same as the one from chapter 3, so we will not describe it again here. However, we will detail the refined symbolic state definitions, explain how we compare those, and then outline lemmas showing that the refined states we introduced are equivalent to the symbolic states from earlier.

7.3.1 Refined Symbolic States and Refinement Relations

The concrete (refined) symbolic states mimic the symbolic states in terms of organization: a refined `sistate_local` will be called `hsistate_local`, a refined `sstate` will be called `hsstate`, and so on.

For each of them, we will present their definition and their refinement relation, that is, how do they relate to their non-refined variant in terms of semantics. Then we will describe the main theorem ensuring that the refined symbolic execution is correct regarding symbolic-execution semantics.

Refined Symbolic Values and Final Symbolic Values

Their definition is very close to the definition of symbolic values:

```
Inductive hsval :=
  | HSinput (r: reg) (hid: hashcode)
  | HSop (op: operation) (lhsv: list_hsval) (hid: hashcode)
  | HSload (hsm: hsmem) (trap: trapping_mode) (chunk: memory_chunk) (addr: addressing)
    (lhsv: list_hsval) (hid: hashcode)
with list_hsval :=
  | HSnil (hid: hashcode)
  | HScons (hsv: hsval) (lhsv: list_hsval) (hid: hashcode)
with hsmem :=
  | HSinit (hid: hashcode)
  | HSstore (hsm: hsmem) (chunk: memory_chunk) (addr: addressing) (lhsv: list_hsval)
      (srce: hsval) (hid:hashcode)
```

Each constructor is exactly like a symbolic value, except that it also has a `hashcode`, used for hash-consing. However one key difference is for `HSop`: this one does not depend on (symbolic) memory, while `Sop` does. This is a simplification to allow for operational instructions to be intermingled with memory stores. Then, when translating a refined symbolic value into a symbolic value (thus giving it a semantics), we reintroduce the symbolic memory as `Sinit` (which evaluates to the initial memory right before executing the superblock). This simplification is correct because the few operations requiring the memory (mostly pointer-comparison operations) do not depend on the values stored in memory: they rely on other fields (that do not change through a superblock) such as memory-block allocation. This only adds a few lines of proof, while tackling this issue directly in the abstract symbolic-execution model would have been more challenging.

The translation from refined symbolic values to symbolic values (that we call projection) is defined below. Aside from the `HSop` case, the rest is quite straightforward.

```
Fixpoint hsval_proj hsv :=
  match hsv with
  | HSinput r _ ⇒ Sinput r
  | HSop op hl _ ⇒ Sop op (hsval_list_proj hl) Sinit
  | HSload hm t chk addr hl _ ⇒ Sload (hsmem_proj hm) t chk addr (hsval_list_proj hl)
  end
with hsval_list_proj hl :=
  match hl with
  | HSnil _ ⇒ Snil
  | HScons hv hl _ ⇒ Scons (hsval_proj hv) (hsval_list_proj hl)
  end
with hsmem_proj hm :=
  match hm with
  | HSinit _ ⇒ Sinit
  | HSstore hm chk addr hl hv _ ⇒ Sstore (hsmem_proj hm) chk addr (hsval_list_proj hl)
    (hsval_proj hv)
  end
```

The refined symbolic final values, of type `hsfval`, are constructed in much the same way as a `sfval`, except that they use refined symbolic values instead of symbolic values.

```
Inductive hsfval :=
  | HSnone
  | HScall (sig: signature) (svos: hsval + ident) (lsv: list_hsval) (res: reg)
    (pc: node)
  | HStailcall (sig: signature) (svos: hsval + ident) (lsv: list_hsval)
  | HSbuiltin (ef: external_function) (sargs: list (builtin_arg hsval))
    (res: builtin_res reg) (pc: node)
  | HSjumptable (sv: hsval) (tbl: list node)
  | HSreturn (res: option hsval)
```

Their projection is implemented in much the same way. It should be noted that we do not hash-cons them. Since they are only found at the end of a superblock, and there is only one of them per superblock, hash-consing would not reduce the number of equality tests performed to check the simulation of final symbolic values.

Refined Local Symbolic State

We have modelled local symbolic state with registers and memory symbolic values (stored as functions acting as mappings, like `reg → sval`) and a function on register and memory values to express a required precondition to evaluate the symbolic state. These types were great to reason semantically—but now we need to define types that can be compared. This is done through `hsistate_local`:

```
Record hsistate_local :=
  {
    hsi_smem:> hsmem;
    hsi_ok_lsval: list hsval;
```

```
    hsi_sreg:> PTree.t hsval
  }
```

First of all, to represent register values, we use a `PTree.t` positive map, a type introduced by CompCert. Internally, this positive map is a tree encoded so that each leaf from the tree is assigned a positive index.

This positive map will store all the refined symbolic values of the registers that are not identity: so there is only a finite number to check, those that are explicitly stored in `hsi_sreg`. The semantics of the positive map is actually implemented in the `hsi_sreg_proj` function below:

```
Definition hsi_sreg_proj (hst: PTree.t hsval) r: sval :=
   match PTree.get r hst with
   | None ⇒ Sinput r
   | Some hsv ⇒ hsval_proj hsv
   end
```

If the mapping does not have a corresponding register entry, it is assumed that the register was not touched (its symbolic value is identity).

To remember which values were computed (necessary to ensure that scheduling does not introduce additional failures), we store them in the list `hsi_ok_lsval`. We only store them if they might incur failure.

Finally, storing the memory is just done through a simple `hsmem` type. Since the refined memory has the history of all the memory assignments (we do not perform any rewriting within the refined symbolic memory), there is no need to store them in a separate list.

We define a property `hsok_local` ensuring that, given a refined local symbolic state, its evaluation will be successful. We also define a `sok_local` property to better reason whether a non-refined local symbolic state would fail.

```
Definition sok_local (ge: RTL.genv) (sp:val) (rs0: regset) (m0: mem) (st: sistate_local):
  Prop :=
  (st.(si_pre) ge sp rs0 m0)
  ∧ seval_smem ge sp st.(si_smem) rs0 m0 <> None
  ∧ ∀ (r: reg), seval_sval ge sp (si_sreg st r) rs0 m0 <> None

Definition hsok_local ge sp rs0 m0 (hst: hsistate_local) : Prop :=
     (∀ hsv, List.In hsv (hsi_ok_lsval hst) → seval_hsval ge sp hsv rs0 m0 <> None)
  ∧ (seval_hsmem ge sp (hst.(hsi_smem)) rs0 m0 <> None)
```

The refinement relation between local refined symbolic state *hst* and non-refined symbolic state *st* is then defined as the conjunction of four properties:

```
Definition hsilocal_refines ge sp rs0 m0 (hst: hsistate_local) (st: sistate_local) :=
      (sok_local ge sp rs0 m0 st ↔ hsok_local ge sp rs0 m0 hst)
  ∧  (hsok_local ge sp rs0 m0 hst →
    smem_refines ge sp rs0 m0 (hsi_smem hst) (st.(si_smem)))
  ∧  (hsok_local ge sp rs0 m0 hst → ∀ r, hsi_sreg_eval ge sp hst r rs0 m0
    = seval_sval ge sp (si_sreg st r) rs0 m0)
  ∧  (∀ m b ofs, seval_smem ge sp st.(si_smem) rs0 m0 = Some m →
    Mem.valid_pointer m b ofs = Mem.valid_pointer m0 b ofs)
```

The first property ensures a simulation, in terms of success, between the refined and non-refined variants. If the evaluation of one fails, then the evaluation of the other must also fail. This simulation is necessary: we need to forward the information that the evaluation succeeded, from a symbolic state to its refined state, then from the refined state to the scheduled refined state, and finally from the scheduled refined state back to the scheduled symbolic state.

The second and third properties ensure that the evaluation of the refined memory/registers states (through projection) coincide with the non-refined version, but only if the evaluation succeeds. If the evaluation of the refined state does not succeed, the property does not give any information; however, we do not need to have this kind of information in such a case. Lastly, the fourth property is there to act as an invariant necessary to have the right to remove the memory dependency on `HSop`. These ensure a bisimulation between refined and abstract symbolic states.

We can note that the execution context (the initial values of registers and memory) must rule out whether there is such a refinement or not because the refinement is expressed in terms of symbolic execution semantics.

Refined Symbolic Exit State

The refined symbolic exit state is like its non-refined counterpart, except that it uses the refined version of the local symbolic state and the symbolic condition values.

```
Record hsistate_exit := mk_hsistate_exit
  { hsi_cond: condition; hsi_scondargs: list_hsval; hsi_elocal: hsistate_local;
    hsi_ifso: node }
```

Because of a technical issue I will point out in the next subsection, we have to split the refinement of exit states into two parts: a static part, independent of the execution context, and a dynamic part:

```
Definition hsiexit_refines_stat (hext: hsistate_exit) (ext: sistate_exit): Prop :=
  hsi_ifso hext = si_ifso ext

```

```
Definition hsiexit_refines_dyn ge sp rs0 m0 (hext: hsistate_exit) (ext: sistate_exit):
  Prop :=
   hsilocal_refines ge sp rs0 m0 (hsi_elocal hext) (si_elocal ext)
   ∧ (hsok_local ge sp rs0 m0 (hsi_elocal hext) →
        hseval_condition ge sp (hsi_cond hext) (hsi_scondargs hext)
          (hsi_smem (hsi_elocal hext)) rs0 m0
         = seval_condition ge sp (si_cond ext) (si_scondargs ext)
          (si_smem (si_elocal ext)) rs0 m0)
```

The static part is straightforward: it just checks that the node values for `ifso` are the same. The dynamic part is a bit more intricate. First of all, we must have a refinement within the local state of the early exit. I recall that this local state is a snapshot of the state we had right before executing the conditional branch.

We must also refine the condition arguments, but this is only required if the local state succeeded. It is a valid weakening of the premise: the condition-evaluation refinement is only necessary if we did not fail earlier in the superblock.

These two definitions are generalized to a list of refined symbolic exit states by the definitions below:

```
Definition hsiexits_refines_stat lhse lse :=
  list_∀2 hsiexit_refines_stat lhse lse

Definition hsiexits_refines_dyn ge sp rs0 m0 lhse se :=
  list_∀2 (hsiexit_refines_dyn ge sp rs0 m0) lhse se
```

Each pair of exit states from both lists must verify the above predicate for the generalized predicate to hold.

Refined Internal Symbolic State

A refined internal symbolic state is like an internal symbolic state but with refined types.

```
Record hsistate := { hsi_pc: node; hsi_exits: list hsistate_exit;
  hsi_local: hsistate_local }
```

The refinement definitions are as such:

```
Definition hsistate_refines_stat (hst: hsistate) (st:sistate): Prop :=
  hsi_pc hst = si_pc st
  ∧ hsiexits_refines_stat (hsi_exits hst) (si_exits st)

Definition hsistate_refines_dyn ge sp rs0 m0 (hst: hsistate) (st:sistate): Prop :=
     hsiexits_refines_dyn ge sp rs0 m0 (hsi_exits hst) (si_exits st)
  ∧ hsilocal_refines ge sp rs0 m0 (hsi_local hst) (si_local st)
```

```
8    ∧ nested_sok ge sp rs0 m0 (si_local st) (si_exits st)
```

They are mostly straightforward, except for the third premise of the dynamic refinement: this one is an invariant ensuring that if a local state somewhere in the superblock (either at the end or an exit state) evaluates successfully, then each of the upwards local states also evaluates successfully. This is a necessary property to have to later reason on the condition evaluation of each exit state since they each require a `hsok_local` property.

More precisely, here is the definition of the predicate `nested_sok`:

```
1  Inductive nested_sok ge sp rs0 m0: sistate_local → list sistate_exit → Prop :=
2      nsok_nil st: nested_sok ge sp rs0 m0 st nil
3    | nsok_cons st se lse:
4       (sok_local ge sp rs0 m0 st → sok_local ge sp rs0 m0 (si_elocal se)) →
5       nested_sok ge sp rs0 m0 (si_elocal se) lse →
6       nested_sok ge sp rs0 m0 st (se::lse)
```

Refined Symbolic State

At last, a refined symbolic state is defined below. It is comprised of a refined symbolic internal state and a refined symbolic final value.

```
1  Record hsstate := { hinternal:> hsistate; hfinal: hsfval }
```

Here is its refinement property:

```
1  Definition hsstate_refines (hst: hsstate) (st:sstate): Prop :=
2     hsistate_refines_stat (hinternal hst) (internal st)
3    ∧ (∀ ge sp rs0 m0, hsistate_refines_dyn ge sp rs0 m0 (hinternal hst) (internal st))
4    ∧ (∀ ge sp rs0 m0, hsok_local ge sp rs0 m0 (hsi_local (hinternal hst)) →
5      hfinal_refines ge sp rs0 m0 (hfinal hst) (final st))
```

We have the static, context-independent part, which involves checking out the static parts from the internal state refinement. And then we have the dynamic part, but this time, valid for any value of the original context: refinement over the internal state and then refinement over the final values, but only if the local-state (from the internal state) evaluation is successful.

Refined Symbolic Execution and Correctness Property

The refined symbolic-execution function `hsexec` is coded in much the same way as `sexec`, except that it uses and updates refined types. We will not detail it, except for one different

aspect that we will describe in subsection 7.3.3: `hsexec` also performs a kind of canonization of hash-consed terms, *e.g.* to turn trapping loads into non-trapping loads.

Its correctness theorem, `hsexec_correct`, is a variation of the following (the actual theorem includes hash-consing specifics that makes it harder to read):

```
1  Lemma hsexec_correct_aux f pc:
2    WHEN hsexec f pc ⇝ hst THEN
3    ∃ st, sexec f pc = Some st ∧ hsstate_refines hst st
```

Given a successful refined symbolic execution of a path starting at *pc* giving *hst*, it ensures that the non-refined symbolic execution would also be successful and return *st*, and that *hst* is a refinement of *st* based on the definitions above.

We will see in the following subsection how we use `hsexec_correct`, the refinement definitions, and some simulation test specifications that we will define to prove the `sexec_simu` relation required for the whole forward simulation.

7.3.2 Simulation Test Specifications and Implementation

In the last subsection, we saw refined symbolic states and projection functions for refined symbolic values.

Prior to that, we saw that in order to achieve a forward simulation for the superblock pass, we need to prove that if the simulation test succeeds, then we must have a `siexec_simu` relation, which I recall below:

```
1  Definition sexec_simu dm (f1 f2: RTLpath.function) pc1 pc2: Prop :=
2      ∀ st1, sexec f1 pc1 = Some st1 →
3      ∃ st2, sexec f2 pc2 = Some st2 ∧ ∀ ctx, sstate_simu dm f1 st1 st2 ctx
```

To construct this relation, we first exhibit specifications that the simulation test[8]; must follow; then, we outline a theorem ensuring that these specifications indeed allow us to deduce the `sexec_simu` simulation. As we have done before, we will follow the structure of refined symbolic states to guide our description.

[8]By simulation test, I mean here the executable part of the verifier which tests that the superblocks simulate each others

Simulation Test of Refined Local Symbolic States

To begin with, the specification for the simulation test on local symbolic states is given as follows:

```
Definition hsilocal_simu_spec (oalive: option Regset.t) (hst1 hst2: hsistate_local) :=
     List.incl (hsi_ok_lsval hst2) (hsi_ok_lsval hst1)
  ∧ (∀ r, (match oalive with Some alive ⇒ Regset.In r alive | _ ⇒ True end)
      → PTree.get r hst2 = PTree.get r hst1)
  ∧ hsi_smem hst1 = hsi_smem hst2
```

Given an original refined local symbolic state *hst*1 and a scheduled one *hst*2, the simulation test should make sure that:

- *hst*2 does not introduce additional trapping computations compared to *hst*1.
- They have the same (structurally) refined symbolic values for the registers in the given set of registers (this set is to be interpreted as the set of live registers).
- They have the same refined symbolic memory.

The actual test is done through two functions based on whether we would like to perform the test modulo liveness or not:

```
Definition hsilocal_simu_check hst1 hst2 : ?? unit :=
  phys_check (hsi_smem hst2) (hsi_smem hst1);;
  PTree_eq_check (hsi_sreg hst1) (hsi_sreg hst2);;
  Sets.assert_list_incl mk_hash_params (hsi_ok_lsval hst2) (hsi_ok_lsval hst1)

Definition hsilocal_frame_simu_check frame hst1 hst2 : ?? unit :=
  phys_check (hsi_smem hst2) (hsi_smem hst1);;
  PTree_frame_eq_check frame (hsi_sreg hst1) (hsi_sreg hst2);;
  Sets.assert_list_incl mk_hash_params (hsi_ok_lsval hst2) (hsi_ok_lsval hst1)
```

In the actual code, we also have debugging information that can be toggled[9] ; for clarity, we omit it here.

We use the `Impure` library from Boulmé [15], upon which hash-consing was added (see appendix C.4.3 of Six et al. [88]). Since we use hash-consing, checking the structural equality of two hash-consed terms amounts to checking physical (pointer) equality. More details can be found in chapter 3: we apply the same mechanisms here. In addition to that, `Impure` also implements an efficient function for testing the inclusion of two lists of hash-consed terms, `Sets.assert_list_incl`.

[9] Debugging such a verifier can be tricky: without debugging information, we know that it fails, but we do not know where. In our development, we used a mix of debugging information, the `ocamldebug` tool, and sometimes even manually instrumenting the generated OCaml code.

Finally, `PTree_eq_check` is a function that checks the structural equality of each tree, the leaves being checked through `phys_check`. `PTree_eq_check_frame` does the same check but only for the concerned registers, by iterating through the `frame` list.

The correctness of these functions is then proven through lemmas such as this one:

```
Lemma hsilocal_frame_simu_check_correct hst1 hst2 alive:
  WHEN hsilocal_frame_simu_check (Regset.elements alive) hst1 hst2 ⤳ _ THEN
  hsilocal_simu_spec (Some alive) hst1 hst2
```

Since `hsilocal_frame_simu_check` involves calling OCaml oracles through the `Impure` framework, termination is not guaranteed—moreover, the `Impure` library does not guarantee that two executions of the same function would give the same result. The `WHEN term ⤳ x THEN P` syntax from `Impure` means: if the computation of `term` finishes and gives a value `x`, then the property `P` holds.

This lemma certifies that the actual simulation test obeys its specification.

Simulation Test of Refined Exit Symbolic States

We define the specification of simulation for exit states:

```
Definition hsiexit_simu_spec dm f (hse1 hse2: hsistate_exit) :=
  (∃ path, (fn_path f) ! (hsi_ifso hse1) = Some path
    ∧ hsilocal_simu_spec (Some path.(input_regs)) (hsi_elocal hse1) (hsi_elocal hse2))
  ∧ dm ! (hsi_ifso hse2) = Some (hsi_ifso hse1)
  ∧ hsi_cond hse1 = hsi_cond hse2
  ∧ hsi_scondargs hse1 = hsi_scondargs hse2
```

Two refined symbolic exit states are semantically equivalent if:

- Their conditions and condition refined symbolic arguments are structurally equal.
- Their respective `ifso` branches obey the reverse mapping
- The `ifso` path from the first exit state exists, and their local states are equivalent modulo liveness of the path.

The simulation test follows closely this specification:

```
Definition hsiexit_simu_check (dm: PTree.t node) (f: RTLpath.function)
  (hse1 hse2: hsistate_exit): ?? unit :=
  struct_check (hsi_cond hse1) (hsi_cond hse2);;
  phys_check (hsi_scondargs hse1) (hsi_scondargs hse2);;
  revmap_check_single dm (hsi_ifso hse1) (hsi_ifso hse2);;
  DO path ⇜ some_or_fail ((fn_path f) ! (hsi_ifso hse1));;
  hsilocal_frame_simu_check (Regset.elements path.(input_regs))
    (hsi_elocal hse1) (hsi_elocal hse2)
```

To ensure a check modulo liveness, the `input_regs` field is taken from the `ifso` path.

We also define and prove a correctness theorem in much the same way we did for local states.

Simulation Test of Refined Symbolic Internal States

Almost last, we define the simulation of two refined symbolic internal states:

```
Definition hsistate_simu_spec dm f (hse1 hse2: hsistate) :=
     list_∀2 (hsiexit_simu_spec dm f) (hsi_exits hse1) (hsi_exits hse2)
  ∧ hsilocal_simu_spec None (hsi_local hse1) (hsi_local hse2)
```

This amounts to checking the simulation of each exit state two-by-two and then checking the simulation of the local state for all registers.

The simulation test implementation is straightforward.

Simulation Test of Refined Symbolic Final Values

The simulation of final symbolic values is defined as structural equality on the symbolic final projections (`final_simu_spec`, not shown here, is more or less structural equality but with some checks on the reverse mapping):

```
Definition hfinal_simu_spec (dm: PTree.t node) (f: RTLpath.function) (pc1 pc2: node)
  (hf1 hf2: hsfval): Prop :=
  final_simu_spec dm f pc1 pc2 (hfinal_proj hf1) (hfinal_proj hf2)
```

The implementation of the simulation test is a bit more lengthy (we have to take special care for builtins that have their own definitions) but still relatively straightforward.

Simulation Test of Refined Symbolic State

Finally, the simulation of two refined symbolic states is expressed by combining the above definitions:

```
Definition hsstate_simu_spec (dm: PTree.t node) (f: RTLpath.function) (hst1 hst2: hsstate) :=
     hsistate_simu_spec dm f (hinternal hst1) (hinternal hst2)
  ∧ hfinal_simu_spec dm f (hsi_pc (hinternal hst1)) (hsi_pc (hinternal hst2))
      (hfinal hst1) (hfinal hst2)
```

Its implementation `hsstate_simu_check` is also straightforward, using the implementation of the underlying smaller structures.

Refinement Simulation from Simulation Test

We define a notion of simulation between two refined symbolic states:

```
1 Definition hsstate_simu dm f (hst1 hst2: hsstate) ctx: Prop :=
2   ∀ st1 st2,
3   hsstate_refines hst1 st1 →
4   hsstate_refines hst2 st2 → sstate_simu dm f st1 st2 ctx
```

Two refined states *hst*1 and *hst*2 simulate each other if their non-refined counterparts simulate each other.

We can prove that the specification given above indeed imply this notion of simulation (though we will not detail the proof here):

```
1 Theorem hsstate_simu_spec_correct dm f ctx hst1 hst2:
2   hsstate_simu_spec dm f hst1 hst2 →
3   hsstate_simu dm f hst1 hst2 ctx
```

Let us now combine what we have so far to prove the `simu_check_correct` theorem, which proves the necessary property for the forward simulation of the scheduling pass. I write the theorem below, with `sexec_simu` unfolded to get a clearer view.

```
1 Lemma simu_check_correct dm f tf:
2   simu_check dm f tf = OK tt →
3   ∀ pc1 pc2, dm ! pc2 = Some pc1 →
4   ∀ st1, sexec f1 pc1 = Some st1 →
5   ∃ st2, sexec f2 pc2 = Some st2
6     ∧ ∀ ctx, sstate_simu dm f1 st1 st2 ctx
```

To help in the description of the proof of this theorem, I drew the diagram in figure 7.9. The terms that I will write below are present in the diagram.

From the premises, we know that `simu_check` succeeded, so each pair $(pc2, pc1)$ of the reverse mapping was checked through refined symbolic execution (which all succeeded) and simulation test of each pair of refined symbolic states. Thus, for the particular $pc1$ and $pc2$ from the theorem, we know that their refined symbolic execution succeeded and gave the refined symbolic states hst_1 and hst_2.

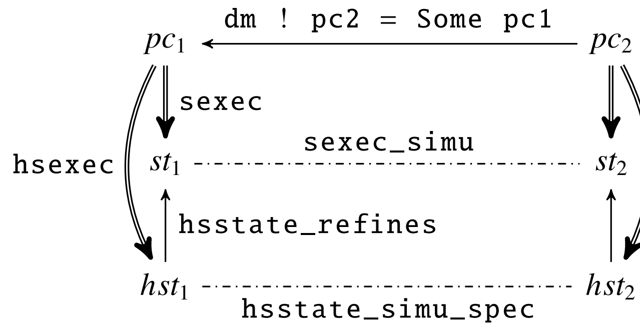


Figure 7.9.: Diagram used for proving the refinement simulation.

By using `hsexec_correct` on these refined symbolic executions, we deduce that the non-refined symbolic executions would also have been successful[10] , and they would have given respectively the non-refined symbolic states st_1[11] and st_2, from which hst_1 and hst_2 are refinements.

Since hst_1 and hst_2 were checked by the verifier, we know they meet the `hsstate_simu_spec` relation; since this relation is correct (by the lemma `hsstate_simu_spec_correct`), we deduce that they follow the `hsstate_simu` simulation.

We have the two refinements: between hst_1 and st_1 and between hst_2 and st_2. So we have a `sstate_simu` relation between st_1 and st_2, which proves the theorem.

7.3.3 Canonization of Refined Symbolic Values

The previous subsections showed how we defined a refined symbolic execution and implemented a simulation test from which we deduce a forward simulation. However, we have not discussed how we can perform small transformations within superblock scheduling, such as transforming trapping loads into non-trapping loads. Indeed, without any change, a trapping load refined symbolic value and a non-trapping load value are structurally different.

We solve this problem by introducing a mechanism of canonization. By simplifying certain symbolic terms into semantically equivalent terms (in the case of loads, we simplify all loads into non-trapping loads), our modified refined symbolic values become comparable again.

[10] I employ the conditional tense here because during the compilation, we never actually execute the non-refined versions! They are only here to reason semantically.

[11] Formally, the st_1 described above (let us name it st'_1) and st_1 from the theorem are two different states. However, since they are both equal to `sexec f1 pc1`, they are actually the same by transitivity of structural equality: $st'_1 = st_1$.

Generalized Operations and Refinement Preservation

To reason about simplifications, we introduce the type `root_sval`:

```
Inductive root_sval: Type :=
| Rop (op: operation)
| Rload (trap: trapping_mode) (chunk: memory_chunk) (addr: addressing)
```

In a way, this represents the "root" of a symbolic value: it abstracts the operations to perform on the symbolic arguments of `Sop` and `Sload`. Here, we will call it a *generalized operation*: it is like an `operation`, except that memory loads are also taken in this abstraction.

The semantics of generalized operators is given by the `root_apply` function below.

```
Definition root_apply (rsv: root_sval) (lr: list reg) (st: sistate_local): sval :=
  let lsv := list_sval_inj (List.map (si_sreg st) lr) in
  let sm := si_smem st in
  match rsv with
  | Rop op ⇒ Sop op lsv sm
  | Rload trap chunk addr ⇒ Sload sm trap chunk addr lsv
  end
```

Given a generalized operator, a list of registers, and a local symbolic state, we fetch the symbolic value of the arguments; then, we insert these in the generalized operator to give a symbolic value.

We also define the same function but for refined states:

```
Definition root_happly (rsv: root_sval) (lr: list reg) (hst: hsistate_local) : ?? hsval :=
  DO lhsv ⇜ hlist_args hst lr;;
  match rsv with
  | Rop op ⇒ hSop op lhsv
  | Rload trap chunk addr ⇒ hSload hst trap chunk addr lhsv
  end
```

The correctness of `root_happly` is ensured through a lemma of refinement preservation (given below): if two local symbolic states refine each other, then applying `root_apply` on one and `root_happly` on the other preserves the refinement.

```
Lemma root_happly_correct (rsv: root_sval) lr hst:
  WHEN root_happly rsv lr hst ↝ hv' THEN ∀ ge sp rs0 m0 st
    (REF:hsilocal_refines ge sp rs0 m0 hst st)
    (OK:hsok_local ge sp rs0 m0 hst),
    sval_refines ge sp rs0 m0 hv' (rsv lr st)
```

Generalized Operators Simplification

The actual canonization is done through the function `simplify` below:

```
Definition simplify (rsv: root_sval) (lr: list reg) (hst: hsistate_local): ?? hsval :=
  match rsv with
  | Rop op ⇒
     match is_move_operation op lr with
     | Some arg ⇒ hsi_sreg_get hst arg
     | None ⇒
       DO lhsv ⇜ hlist_args hst lr;;
       hSop op lhsv
     end
  | Rload _ chunk addr ⇒
       DO lhsv ⇜ hlist_args hst lr;;
       hSload hst NOTRAP chunk addr lhsv
  end
```

Right now, this function has two purposes:

- It simplifies any memory load into a non-trapping memory load, which is correct if there is no failure (see section 5.1.2).
- It simplifies any move operation (operations that copy the content of a register to another register) into copying the symbolic value of the register directly. For instance, `(Sop Omove [sv] Sinit)` will be simplified to just `sv`. This is also a correct simplification.

Correctness is proven through a lemma of refinement preservation, much like `root_happly_correct`, that we prove by case analysis:

```
Lemma simplify_correct rsv lr hst:
  WHEN simplify rsv lr hst ⤳ hv THEN ∀ ge sp rs0 m0 st
    (REF: hsilocal_refines ge sp rs0 m0 hst st)
    (OK0: hsok_local ge sp rs0 m0 hst)
    (OK1: seval_sval ge sp (rsv lr st) rs0 m0 <> None),
    sval_refines ge sp rs0 m0 hv (rsv lr st)
```

This mechanism thus allows us to replace memory loads with their speculative versions and perform some other small optimizations on the size of the symbolic state without breaking the model on which we prove the superblock scheduling.

Optimizing the Size of the Refined Symbolic States

I will present now two optimizations that decrease the simulation test execution time by further reducing the size of the refined symbolic state under certain specific conditions.

The first optimization consists of removing from the `PTree` of certain registers: bindings of the form $r \mapsto$ `(HSinput r _)`. Indeed, in the absence of a binding for a given register of a refined local symbolic state, the refined evaluation semantics consider the register's symbolic value to be an identity. The optimization is done through using the function `red_PTree_set` below whenever we would like to set such a register value[12] :

```
1  Definition red_PTree_set (r: reg) (hsv: hsval) (hst: PTree.t hsval): PTree.t hsval :=
2    match hsv with
3    | HSinput r' _ ⇒
4       if Pos.eq_dec r r'
5       then PTree.remove r' hst
6       else PTree.set r hsv hst
7    | _ ⇒ PTree.set r hsv hst
8    end
```

In terms of RTL semantics, this optimization amounts to replacing $rs[r \mapsto rs[r]]$ by rs, where rs is a regset.

Another optimization consists of limiting the number of symbolic values present in the `hsi_ok_lsval` field of a refined symbolic local state. Indeed, out of all the computed symbolic values, we only need to store those that might incur failures. If we can prove that a given refined symbolic value would never have any failure, then we can safely remove it from the list of computed symbolic values. For instance, speculative loads never fail, so there would be no need to include them.

This is done during the refined symbolic execution: before changing the refined local state, the generalized operator is checked by the function `may_trap` below (`|||` is here a lazy boolean OR operator):

```
1  Definition may_trap (rsv: root_sval) (lr: list reg): bool :=
2    match rsv with
3    | Rop op ⇒ is_trapping_op op ||| negb (Nat.eqb (length lr) (args_of_operation op))
4    | Rload TRAP _ _   ⇒ true
5    | Rload NOTRAP _ _  ⇒ false
6    end
```

If `may_trap` returns `false`, then we know that it is impossible for any evaluation of the refined symbolic value to fail, so we do not add the value to the list `hs_ok_lsval`. This allows the simulation test to work on smaller sets when performing the inclusion test.

These optimizations should improve the execution time of the simulation test.

[12]This function accepts rewriting rules. For instance, `r1 := r2; r2 := r1` is simulated by `r1 := r2`. We do not use this feature in superblock scheduling.

Part III

Experiments

8 Performance Tuning of Superblock Formation, Selection and Linearization

In the previous chapters I outlined a number of formally proven verifiers related to superblock scheduling: the RTL-to-RTLpath pathmap and liveness verifiers from chapter 5, the generic `Duplicate` verifier from chapter 6 and finally the superblock scheduling verifier from chapter 7. In the two following chapters, I will detail the implementation of the underlying oracles (except the RTLpath liveness oracle): even though their implementation does not affect the correctness of the passes thanks to the verifiers, they are in control of the performance aspects of each pass. A bug in the implementation of any of those will result in either a compilation failure (the oracle returns an incorrect result: the verifier rejects the compilation) or bad performance (the oracle did detrimental transformations).

This chapter is dedicated to the oracles of tail-duplication (section 8.1), loop unrolling and rotation (section 8.2), branch prediction (section 8.4) and linearization (section 8.3). If you have not already, before reading this chapter, I encourage the reader to read chapter 4 which introduces the general concepts of superblocks: it gives a quick and graphical overview of the various transformations described here.

8.1 Trace Selection and Tail-Duplication

Once branch prediction is performed (either statically via section 8.4, through `__builtin_expect` user annotations, or through profiling), the next step is to form up execution traces that follow the likely paths of execution and then possibly transform the traces that are not superblocks into superblocks.

I do this combined work in two distinct steps:

- I partition the RTL CFG in a set of traces (subsection 8.1.1)
- Each trace is tail-duplicated as long as the amount of duplicated nodes does not go over a certain threshold to prevent duplicating too much (subsection 8.1.2).

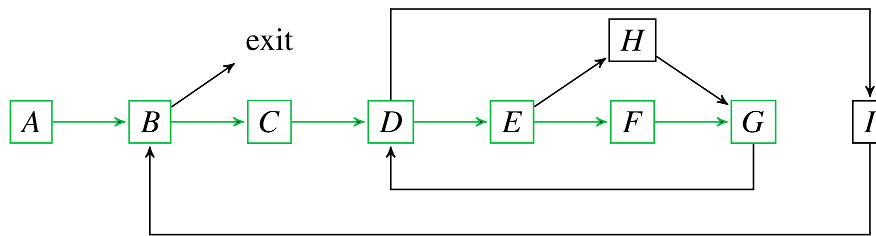


Figure 8.1.: RTL code for two nested loops. The identified trace is shown in green.

These two steps are done in a single pass, called `TailDuplicate`. That pass is an instance of the `Duplicate` module from chapter 6: in particular, we must be careful in the OCaml oracle to exhibit a reverse mapping to help the correctness verifier.

The pass does not perform the actual superblock selection: this is done later, through the RTL to RTLpath oracle. The emphasis here is to transform the CFG so that the later actual superblock selection will be as fruitful as possible in terms of performance gain; however, there is no exact correspondence between a selected trace and a future superblock.

To give a practical example, figure 8.1 shows the typical example of two nested loops, as generated by CompCert. Each conditional branch from the loops is assumed to be predicted correctly (the static prediction ruled that the execution will remain within the loop most of the time), and let us assume there is a conditional branch within the innermost loop that was predicted.

An obvious trace for that code would be $A - B - C - D - E - F - G$: it surely regroups the most executed nodes from that code. That trace is not a superblock because there are side entrances: the two loop edges $I \rightarrow B$ and $G \rightarrow D$, plus the edge $H \rightarrow G$, most likely coming from an if-then-else construct.

Applying tail-duplication naively would involve duplicating instructions starting from the first side entrance, so starting at node B. Figure 8.2 shows the result of applying such a tail-duplication. This would create a superblock following the first execution of the outermost and innermost loops, but as soon as the second iteration of the innermost loop would be reached, the execution would go back to the duplicated (non-superblock) code.

On the other hand, tail-duplicating only from node G (shown figure 8.3) would provide just what we want: a superblock encompassing the innermost loop.

There are two ways to tackle this issue. Either we cut the trace so that only the innermost loop from the trace is selected for tail-duplication, or we skip the loop headers when going through the trace. I found the latter to be more practical to implement, but the former could also have been a possibility and would have required a similar amount of effort.

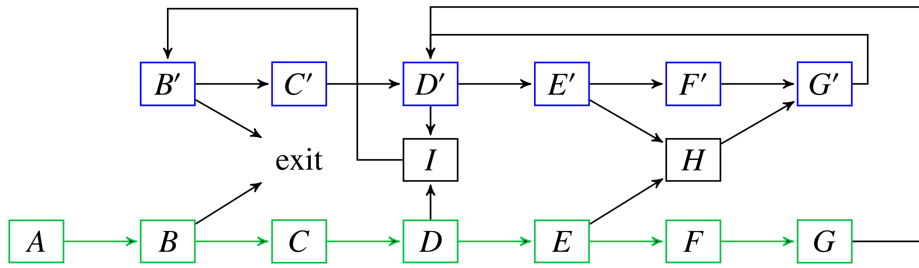


Figure 8.2.: RTL code for two nested loops, with a tail-duplication starting at B. The identified trace shown in green is now a superblock, although unwanted. The duplicated nodes as a result of tail-duplication are shown in blue.

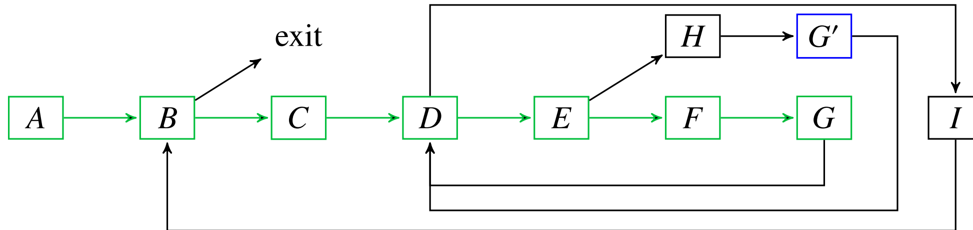


Figure 8.3.: RTL code for two nested loops, with an interesting tail-duplication done on the green trace. The duplicated node is in blue.

The remaining subsections explain the exact algorithms performing trace partitioning and tail-duplication.

8.1.1 Trace Partitioning

Given an RTL control-flow graph annotated with prediction information, we would like to partition it so that each node will belong to only one trace: we would not want to apply tail-duplication to a set of nodes that already went through tail-duplication. We are particularly interested in traces that follow the most used execution flow, potentially entering loops (our tail-duplication will not duplicate the whole loop) and, whenever predicted, following the predicted branch of a conditional branch. Needless to say, fair branch prediction is necessary to prevent this pass from selecting "bad traces".

As was summarized in section 4.1.4, I largely base my algorithm on the one from Chang and Hwu [20], except that I forbid traces from crossing each other, and I rely only on the prediction information from conditional branch instructions. I recall that these are of ternary nature (either `None`, `Some true` or `Some false`).[1]

[1] See section 5.1.3 for more details on prediction information.

First of all, I define in algorithms 5 and 6 the functions `best_successor_of` and `best_predecessor_of` that return respectively a node that is picked to grow the trace either forward or backward.

Our representation of branch prediction makes it straightforward to select the `best_successor`: we just have to follow the predicted branch in the case of a predicted conditional branch or follow the only possible branch for a node that only has one successor. For the other cases, `None` is returned: this will then be interpreted as stopping to grow the trace. It also returns `None` when the predicted successor is already visited to prevent creating overlapping traces.

The implementation of `best_predecessor_of`, which selects a predecessor to grow the trace backwards, is a bit less strict. We only have the notion of "predicted successor" in our representation; we do not have any notion of "predicted predecessor". Furthermore, since we store branch prediction in a ternary way, we do not have enough information to determine which predecessor is more likely to have been crossed by the execution. The function `(best_predecessor_of n)` then simply returns a node n' for which the predicted successor of n' is n, and n' is not visited. If there are several of them, the algorithm chooses based on the DFS order of the graph.

Algorithm 5. Choosing privileged successor for trace selection.

```
1:  function BEST_SUCCESSOR_OF(node, code, visited)
2:      function SELECT(node)
3:          match code[node] with
4:              case Icond _ _ ifso ifnot pred
5:                  match pred with
6:                      case None
7:                          return None
8:                      case Some true
9:                          return Some ifso
10:                     case Some false
11:                         return Some ifnot
12:             case Ijumptable _
13:                 return None
14:             case i
15:                 return either None or (Some s) based on the presence of a successor
16:     chosen ← select(node)
17:     if visited[chosen] then
18:         return None
19:     else
20:         return Some chosen
```

The set of successors of a given node is straightforward to obtain (just extract them from the RTL instruction syntax). The set of predecessors is trickier and requires to go through the whole graph. I abstracted this detail in the algorithm, but in practice, I pre-compute the set of predecessors for

Algorithm 6. Choosing privileged predecessor for trace selection.

```
1:  function BEST_PREDECESSOR_OF(node, code, visited)
2:      foreach pred ∈ predecessors[node] in DFS order
3:          if visited[pred] then
4:              continue
5:          match code[pred] with
6:              case Icond _ _ ifso ifnot pred
7:                  if (pred = Some true && node = ifso) || (pred = Some false && node = ifnot) then
8:                      return Some pred
9:                  else
10:                     continue
11:             case Ijumptable _
12:                 continue
13:             case _
14:                 return Some pred        ▹ Other instructions have either 1 or 0 successor.
15:     return None
```

each node by a simple DFS graph traversal before running the actual trace-selection algorithm. Another detail I abstracted is that computing a DFS graph traversal order requires knowing which node is the entrypoint. In the implementation, the DFS order is pre-computed, then given as a parameter to the functions.

Once we have these two functions to select a successor/predecessor to grow a trace, we can perform the actual trace selection (algorithm 7). The trace selection algorithm starts from an unvisited node (selected by DFS order). It tries to grow the trace forward through the successors; then, it tries to expand it backwards through the predecessors when it cannot grow further in the forward direction. The algorithm marks nodes as visited as they are added to a trace. Once the trace is finished (impossible to grow the trace in both forward and backwards directions), the trace is stored, and we start a new one from an unvisited node, by DFS order.

8.1.2 Tail-Duplication

Once the traces are selected, tail-duplication is applied to them, but in such a way that whole loops will not be tail-duplicated. The tail-duplication is written in algorithm 8. It takes as input the code to modify, the current reverse mapping (since each trace is tail-duplicated one-by-one, the current reverse mapping needs to be updated accordingly), and the trace to perform tail-duplication on.

The gist of the algorithm is to go through each node, starting from the first node of the trace, then analyze if that node has any other predecessor than the previous node of the trace: ignoring

Algorithm 7. Trace-selection algorithm based on Chang and Hwu [20]

```
1:  function SELECT_TRACES(code)
2:      visited ← { all ↦ false }
3:      traces ← []
4:      while ∃n ∈ code s.t. visited[n] = false do
5:          seed ← first node n of DFS order s.t. visited[n] = false
6:          current ← seed
7:          add_to_trace(trace, current)
8:          while true do
9:              match best_successor_of(current, visited) with
10:                 case None
11:                     break
12:                 case Some s
13:                     trace := trace + node
14:                     visited[node] := true
15:                     current ← s
16:         current ← seed                    ▷ Resetting to grow the trace backwards
17:         while true do
18:             match best_predecessor_of(current, visited) with
19:                 case None
20:                     break
21:                 case Some p
22:                     trace := node + trace
23:                     visited[node] := true
24:                     current ← p
25:         traces += trace                   ▷ The trace is finished growing
26:     return traces
```

Algorithm 8. Tail-duplication algorithm

```
 1: function TAIL_DUPLICATE(code, revmap, trace)
 2:     counter ← 0
 3:     code ← copy(code)
 4:     npc ← next_free_pc(code)
 5:     is_first ← true
 6:     foreach n ∈ trace
 7:         if is_first then          ▷ Special case for the trace entry: it is never duplicated.
 8:             is_first ← false
 9:             prev ← n
10:             continue
11:         lp ← predecessors(n) - prev - backedges(n)
12:         if lp ≠ ∅ then
13:             code[npc] := copy(code[n])
14:             reroute_nodes(code, lp, n ↦ npc)
15:             revmap := revmap[npc ↦ n]
16:             if code[n] is not a Inop then
17:                 counter += 1
18:         prev ← n
19:     return (code, revmap, counter)
```

loop edges (whole loops must not be tail-duplicated), if there is any predecessor, then it duplicates the node and points the predecessors into that duplicated node instead. In the algorithm, `backedges(n)` return the nodes having backedges to `n`, `reroute_nodes` is a function that changes the pointers of each node from a list to make them point to another node instead, and `copy` is an abstract function symbolizing a deep copy.

The algorithm returns a triple of: the modified code, the reverse mapping (useful for the verifier), and also a counter keeping track of the number of duplicated nodes (not counting `Inop` instructions: these are eliminated later on in CompCert anyway, so they are not counted). If the counter reaches a certain threshold, the returned code is ignored: the transformation is an identity (no tail-duplication is performed).

This is to prevent cases where the code-size penalty due to tail-duplication would be too big compared to the performance gain. Right now, this is a very simple heuristic. Probably some smarter heuristics could be implemented in the future.

At the end of the algorithm, we should have a trace where any side entrance (except loop backedges) is redirected to duplicated nodes, which should, in the case of a trace going inside an innermost loop, transform the innermost loop body into a superblock. I iterate the algorithm for each trace found in the RTL function by the `select_traces` function from earlier.

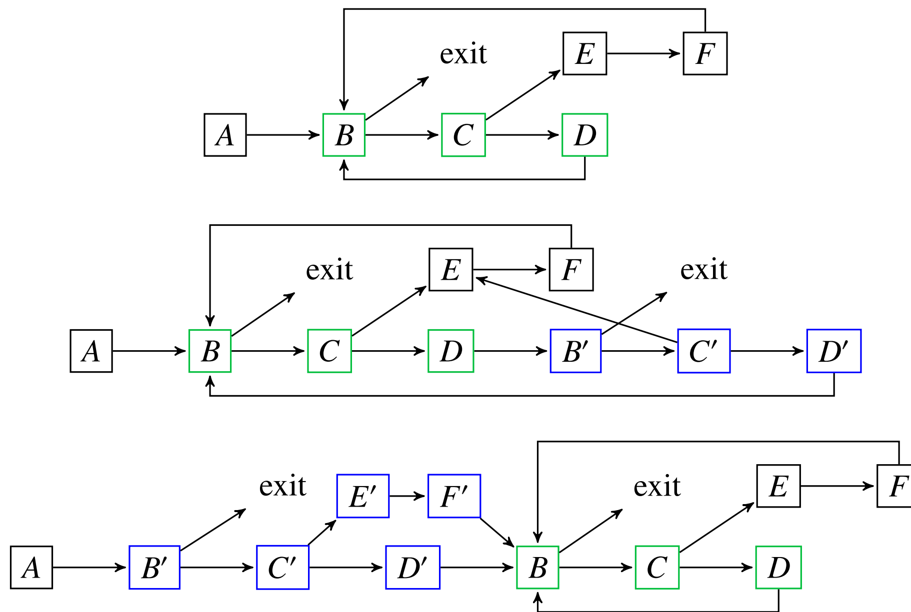


Figure 8.4.: Example of loop unrolling and loop peeling: above is the original code, in the middle is the unrolled code, at the bottom is the code with loop peeling. The nodes that are part of the original superblock are shown in green. The duplicated nodes are shown in blue.

We have hand-tested our implementation on many examples to ensure that it behaves correctly, though a bug in there would not compromise correctness, thanks to our `Duplicate` verifier.

8.2 Loop Unrolling, Peeling and Rotation

Loop unrolling and rotation are two transformations that do not do much on their own but, when combined with prepass and postpass scheduling, can have some performance impact. Much like tail-duplication, these two transformation passes are instances of the generic `Duplicate` module.

Figure 8.4 shows an example of loop unrolling on a simple innermost loop, which went through a tail-duplication (it's a similar code to figure 8.3 but simplified). Interestingly, this unrolling can be done just by duplicating nodes and changing node pointers: this makes it an excellent fit for our `Duplicate` verifier. Also, such an unrolling preserves the superblock structure: this then increases the scheduling potential of a superblock.

We also perform loop peeling: we duplicate the loop body once, outside of the loop (see figure 8.4). Loop peeling allows to copy in particular loop-invariant expressions outside of the loop, in order for CSE3 to remove these expressions from the main loop body [63]. Loop peeling is very similar to loop unrolling; it relies on duplicationg the relevant nodes, then adjusting the successor pointers. I do not detail its algorithm here.

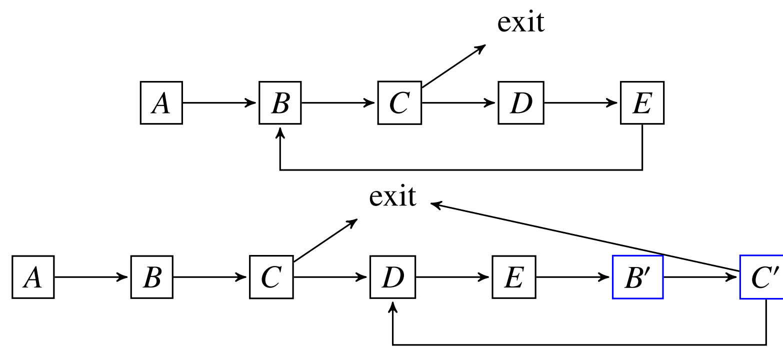


Figure 8.5.: Example of loop rotation: above is the original code, below is the rotated code, with the duplicated nodes shown in blue.

Finally, loop rotation is another optimization of interest, illustrated by figure 8.5. This optimization does not have much impact on the prepass scheduling; however, it may impact the postpass scheduling for a technical reason inherent to the architecture-specific CompCert backends.

In RTL, a conditional-branch instruction is a single instruction that performs both the comparison computation and the actual branching. In most ISAs, however, this is done in two different instructions: one that performs the comparison, setting a specific register to the result of that comparison; then another that uses that result to branch accordingly.[2]

As a result, for an innermost loop whose condition would require a separate instruction, the generated code from Mach to Asm consists of the code performing the comparison, then the conditional branch, and finally the actual loop body. The later basic-block scheduling will then be unable to schedule the comparison code with the rest of the loop body since there is a conditional branch between. The prepass scheduling also will not be able to do much with the comparison code: it cannot move the comparison code below the conditional branch (it would conflict with the register used for comparison), and in some cases, it cannot move instructions above the branch either (if these would trap).

A solution consists of rotating the loop in the way of figure 8.5: enough instructions are duplicated for the first iteration to be correct; then, the loop is shifted in such a way that the conditional-branch instruction is last in the loop body. This allows both basic block and superblock schedulings to potentially schedule the comparison instructions with the rest of the loop body.

Both algorithms rely on a prior innermost-loop detection, which outlines innermost loops and valuable information for the later passes. I first describe our innermost-loop detection in subsection 8.2.1, then I describe the two algorithms: loop unrolling in subsection 8.2.2, loop rotation in subsection 8.2.3.

[2]This is the case for the KVX ISA, but also more mainstream ones like the Risc-V instruction set.

8.2.1 Innermost-Loop Detection

Much like tail-duplication required to have traces to perform the optimization, our loop unrolling and rotation transformations need us to know beforehand where the innermost loops are.

For that, I use the innermost-loop detection implemented by David Monniaux for the Loop Invariant Code Motion optimization [63]: given a set of loop headers to investigate (these are computed with a graph traversal), it returns a mapping where to each loop header is associated the list of nodes composing the innermost loop (if it is indeed an innermost loop). This is done by using a dominator dataflow analysis.

For each innermost loop identified, the following data structure is then generated by using simple graph-traversal analysis (not detailed here).

```
type innerLoop = {
  preds: P.t list;
  body: P.t list;
  head: P.t;
  finals: P.t list;
  sb_body: P.t list option;
}
```

- `preds` contains the predecessors of the loop, that is, which nodes have pointers to the loop entry.
- `head` is the first instruction of the loop (the entry).
- `body` is the list of instructions (including the head) of the loop.
- `finals` is the list of the instructions that loop back to the head. There can be more than one such instruction: for example, a tail-duplicated loop will have at least two final instructions.
- If the innermost loop wraps a whole superblock, then `sb_body` are its instructions. Acquiring `sb_body` involves starting from the head of the loop, then following the predicted path until it either reaches back to the head or stops because of a `None` prediction.

For example, if we examine the original code from figure 8.4: `preds` is `[A]`, `head` is `B`, `body` is the whole loop, so `[B, C, D, E, F]`, and `finals` is `[E, F]`. However, we are not interested in unrolling the whole loop since `E` and `F` are not part of any superblock; so, in addition to this information, we also have `sb_body` = `[B, C, D]` that contains the actual superblock we are interested in.

Once this structure is acquired, the loop-unrolling and loop-rotation transformations below are executed.

8.2.2 Loop Unrolling

To unroll loops, I first define a function `clone(code, revmap, ln)` whose goal is to duplicate the instructions from the list `ln`, on the original code `code`; and with the original reverse mapping `revmap` as parameter to update it. The function is defined in algorithm 9. `bindings(ln, ln')` is an abstract function returning a mapping that binds two-by-two each element of `ln` with the corresponding element of `ln'`.

Algorithm 9. Algorithm to clone a list of instructions

```
 1: function CLONE(code, revmap, ln)
 2:     head' ← next_free_pc(code)
 3:     ln' ← map(ln, n ↦ n + head')
 4:     fwmap ← bindings(ln, ln')
 5:     revmap' ← revmap[bindings(ln', ln)]
 6:     code' ← code
 7:     foreach n ∈ ln
 8:         n' ← fwmap[n]
 9:         code'[n'] := copy(code[n])
10:         reroute_nodes(code', [n'], fwmap)
11:     return (code', revmap', ln', fwmap)
```

In a way, when `ln` is a contiguous set of nodes from the RTL graph, the algorithm duplicates both the nodes and the edges within that set. The nodes are first copied structurally (including their pointers), then the pointers are changed through applying a forward mapping `fwmap`: the pointers that initially pointed to nodes from the set now point to the duplicated nodes instead.

To ensure there is no collision (the new node integers used for the duplicated instructions do not collide with existing nodes), I use a trick: the next available integer is computed (the next free PC) and then added to the node from the instruction to duplicate. The result of that addition is the new node. Doing it this way also ensures a certain consistency between the original code structure (in terms of used integers) and the modified code, which makes the transformation pass slightly easier to debug or analyze.

The loop unrolling, described in algorithm 10, then consists of the following steps:

1. The superblock body `sb_body` is cloned into new instructions `sb_body'`.
2. The last instruction of `sb_body` is modified to point to the head of `sb_body'` (instead of looping back).
3. The last instruction of `sb_body'` is modified to loop back to the head.

Like tail-duplication, if too many instructions to duplicate are counted (except `Inop` instructions), the loop is left without change.

Algorithm 10. Algorithm to unroll the body of a loop

```
 1: function UNROLL_BODY(code, revmap, iloop)
 2:     match iloop.sb_body with
 3:         case None
 4:             return (code, revmap)
 5:         case Some sb_body
 6:             if too_many_instructions(sb_body) then
 7:                 return (code, revmap)
 8:             sb_final ← last(sb_body)
 9:             (code', revmap', sb_body', fwmap) ← clone(code, revmap, sb_body)
10:             head' ← fwmap[iloop.head]
11:             sb_final' ← fwmap[sb_final]
12:             reroute_nodes(code', [sb_final], iloop.head ↦ head')
13:             reroute_nodes(code', [sb_final'], head' ↦ iloop.head)
14:             return (code', revmap')
```

8.2.3 Loop Rotation

The loop rotation is done similarly: first, a group of nodes to duplicate is identified - then, they are duplicated, and the pointers are changed to make a loop rotation. In the case of loop rotation, the group of nodes to duplicate is all the nodes from the head to the conditional branch.

I define a function `extract_upto_icond`, not detailed here. Given a head, this function attempts to follow the execution path until it reaches a conditional branch. If the function finds an instruction that does not have exactly one successor, it returns `None`. Similarly, if the function reaches back the head, it also returns `None` (this might happen in the case of an infinite loop in the C source program: a loop without any conditional branch).

I write in algorithm 11 the loop-rotation algorithm.

Algorithm 11. Algorithm to rotate a loop

```
 1: function ROTATE_LOOP(code, revmap, iloop)
 2:     match extract_upto_icond(code, iloop.head) with
 3:         case None
 4:             return (code, revmap)
 5:         case Some header
 6:             if too_many_instructions(header) then
 7:                 return (code, revmap)
 8:             (code', revmap', header', fwmap) ← clone(code, revmap, header)
 9:             head' ← fwmap[iloop.head]
10:             reroute_nodes(code', iloop.preds, iloop.head ↦ head')
11:             return (code', revmap')
```

Exactly like tail-duplication and loop unrolling, no modification is performed if too many nodes would be duplicated.

8.3 Adapting the `Linearize` Oracle for Superblocks

Chapter 1 gave an overview of each transformation pass and intermediate representation. In particular, the `Linearize` transformation pass generates a `Linear` code out of an `LTL` code. The `LTL` representation is like `RTL`, except for three major differences:

- Whereas an `RTL` code is a graph of instructions, an `LTL` code is a graph of basic blocks, each basic block being represented as a list of instructions: the instructions within the basic block do not have successor pointers, except for the final instruction. Unconditional jumps are represented by a `Lbranch` instruction.
- The registers are allocated, so the registers appearing in `LTL` directly correspond to machine registers (there is only a finite number of them).
- To handle spilling registers into the stack, two new instructions `Lgetstack` and `Lsetstack` are introduced, which respectively get a value from the stack and set a value on the stack, at a given position.

The `Linear` representation is like `LTL`, except that this time, the control-flow graph has been flattened. A `Linear` code is a list of instructions, with the same instructions as `LTL`, except for two new instructions that are added to keep the control flow intact: `Llabel`, defining a label (an additional entrypoint), and `Lgoto`, defining an unconditional jump to one of the defined labels. The semantics of `Linear` is then much like `Mach`: the list of instructions composing the function is executed, instruction-by-instruction, until an `Lgoto` is encountered — the list of instructions is then refreshed to contain the list of instructions starting from the corresponding `Llabel` instruction. If an `Lcond` is encountered, and the condition evaluates to `false`, the semantics of the `Lcond` is to simply ignore the instruction and execute the rest of the list.

One way to translate an `LTL` code into `Linear` is to concatenate the list of instructions of each basic block together (along a certain order), and then insert `Lcond`, `Lgoto` and `Llabel` instructions accordingly to preserve the control-flow structure of the original graph.

However, one must be careful to give a "good order": being careless could lead to bad performance. Indeed, in most architectures, unconditional jumps are not free to execute: not only do they incur a cycle penalty for some architectures (due to introducing a gap in the processor pipeline), they also might invalidate parts of the instruction cache if the jump bridges two memory addresses too far apart. In the case of the KVX, an unconditional jump induces a penalty of 1 cycle in the best-case scenario.

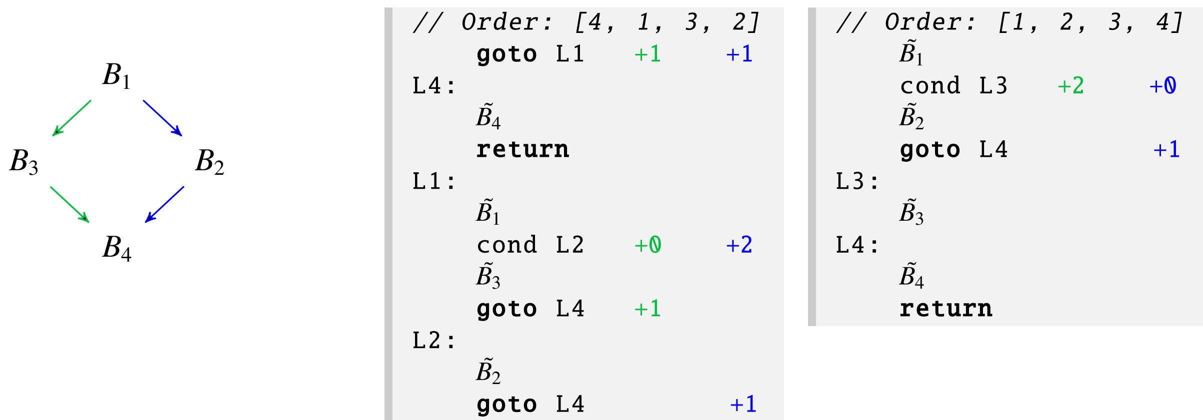


Figure 8.6.: Two possible code layouts for the LTL code shown on the left. Each code layout is annotated with branching penalties from two possible executions: either through B_3 (green) or through B_2 (blue). On average, the layout on the right has lower branching penalties than the layout on the left.

Another thing to consider is that in most architectures, conditional branches work the following way: either the condition is evaluated to `true`, in which case a jump is performed to a particular label; or the condition is evaluated to `false`, and then the next instruction is executed (as if the conditional-branch instruction was a `nop` instruction). For the KVX, if the conditional jump is taken, a 2-cycle penalty is suffered in the best-case scenario. No penalty is induced if the conditional jump is not taken. The positioning of code can then significantly impact the performance speedup, especially when considering hot loops (loops in which the program spends a significant amount of time) that have only a few cycles of execution time.

I give in figure 8.6 an example of possible orders for basic blocks, on a simple code composed of four basic blocks B_1, B_2, B_3 and B_4. I denote $\tilde{B}_i$ as B_i without its final instruction ($\tilde{B}_i$ can be empty). The presented code could typically be the result of an if/then/else construct. The execution flow goes either through B_3 or B_2, but there is no way to know which, at compile time. I annotated the `Linear` code with the branch penalties, distinguishing between the two execution cases: the first column indicates the penalty induced by the path through B_3, the second indicates the penalty caused by the path through B_2. The `Linear` code from the left would lead to either 2 cycles of penalty when going through B_3 or 4 cycles of penalty when going through B_2. On the other hand, the code on the right would lead to either 2 cycles of penalty when going through B_3 or 1 cycle when going through B_2: on average, many fewer cycles would be spent jumping!

In CompCert, the `Linearize` pass is separated into two parts: an untrusted oracle in OCaml that returns the order in which the basic blocks should be laid out and the actual transformation that takes the order from the oracle and inserts `Lgoto`, `Llabel` and `Lcond` accordingly to preserve the LTL control flow.

The untrusted oracle has the following type:

```
1 Parameter enumerate_aux: LTL.function → PMap.t bool → list node
```

Given a function from LTL (which contains, in particular, the LTL code and entrypoint), and also given a node-to-bool mapping indicating whether a given node is reachable[3] , the oracle should return an order encoded by a list of nodes.

The only constraint of the order returned by the oracle (verified *a posteriori*) is that each reachable node from LTL must appear once in the order.

In subsection 8.3.1 I describe the original heuristics used by CompCert; then in subsection 8.3.2 I detail why the original heuristics might not be a good fit for superblocks, explaining my modifications to make them better for superblocks.

8.3.1 Original `Linearize` Heuristics from CompCert

The original `Linearize` heuristic used by CompCert builds up chains of basic blocks that each have only one predecessor and one successor (except for the first basic block of the chain). Each basic block of the chain is then laid out contiguously in the returned order. When a join point (a basic block with more than one predecessor) is met, the chain is broken, and a new chain starts. When a basicblock with more than one successor (such as a conditional branch) is met, the chain ends; new chains start from each of the successors.

Much like RTL, in an LTL code, each basic block has an associated integer (the node): the LTL code is then a mapping from integers (nodes) to their content (basic blocks). When constructing a chain, the lowest node (by integer order) of its basicblocks is recorded. The final order is then computed by laying out each chain in decreasing order of their respective lowest nodes.

I represent the original heuristics of CompCert by the algorithm 12 below. The actual implementation is different, as it is written in a recursive way instead of iterative, but the core idea remains the same.

The function `build_chain` bridges together as many basic blocks as it can, stopping before a join point is reached or when a basic block with more than one successor is met. Then, the successors of the last basic block are added to the nodes to visit, and the building of another chain starts. While building a chain, the node's minimum value is stored: the chain is then annotated

[3]The `Linearize` oracle from CompCert does not actually use this parameter: unreachable nodes are naturally left out by the underlying algorithm from the oracle.

Algorithm 12. Original `Linearize` oracle ("`enumerate_aux`" in the Coq source)

```
1: function ENUMERATE(f)
2:     visited ← ∅
3:     join_points ← compute_join_points(f)
4:     function GET_NEXT_PC(pc)
5:         match last_instruction_of(pc) with
6:             case Lbranch s
7:                 return (Some s, [])
8:             case i
9:                 return (None, successors(i))
10:    function BUILD_CHAIN(pc)
11:        chain ← []
12:        minpc ← pc
13:        while true do
14:            chain += pc
15:            visited ← visited ∪ pc
16:            if minpc > pc then
17:                minpc ← pc
18:            (npc, nexts) ← get_next_pc(pc)
19:            match npc with
20:                case None
21:                    break
22:                case Some npc
23:                    if n ∈ join_points then
24:                        break
25:                    pc ← npc
26:        return (chain, minpc, nexts)
27:    to_visit ← [ entrypoint ]
28:    while to_visit ≠ ∅ do
29:        pc ← to_visit.pop()
30:        if pc ∈ visited then
31:            continue
32:        (chain, minpc, nexts) ← build_chain(pc)
33:        to_visit += nexts
34:        chains += (minpc, chain)
35:    chains ← chains sorted by decreasing order of minpc
36:    return flatten(map(snd, chains))          ▹ Concatenating each chain together
```

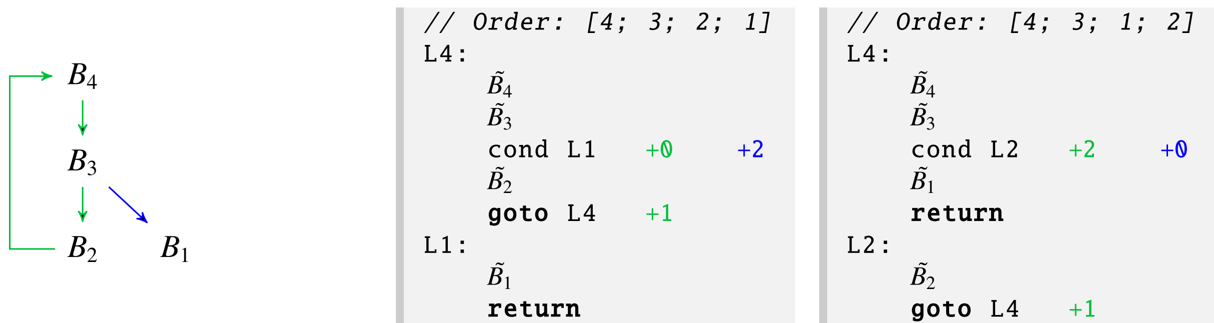


Figure 8.7.: Two possible code layouts for the LTL looping code shown on the left. The left layout is from the original CompCert heuristics, while the right layout is from a hypothetical heuristics where the chains are laid out in increasing order.

with this value. The chains are then ordered by this `minpc` value (by decreasing value), and then, they are flattened into one list of nodes.

The following is my best guess as to why this algorithm works well (and is a good fit, in general, for CompCert).

I believe this algorithm works reasonably well for CompCert, mainly because the nodes are generated so that the body of a loop will always have node identifiers greater than those of the exit branch of a loop. More formally, I think these heuristics work for orderings such that each pair (A, B) of strongly connected components verify the property: "if a successor of A is in B, then each node of A has an identifier greater than the nodes of B".

The `RTLgen` pass generates such a well-structured code by initiating code generation for loop exits before handling the loop body. The `Renumber` pass is then careful to preserve this structure.

To illustrate these heuristics, I give a simple loop in figure 8.7, where I note B_i the basic block of node i. I give two possible code layouts: on the left, the one given by the original CompCert heuristics; on the right, what would have happened if the heuristics had selected an increasing order instead of decreasing (in both cases, the basic blocks B_4 and B_3 are in the same chain, so they are laid out contiguously).

Without any branch-prediction information, CompCert heuristics chose to privilege the basic blocks within the loop on their own: this was the right choice since it only incurs 1 cycle of penalty when looping (assuming the code does take the loop most of the time). The other solution, consisting of laying out the exit branch first, would incur 3 cycles of penalty when looping, which would be very detrimental.

8.3.2 Modifications to the Heuristics for Superblocks

CompCert already naturally makes the right call for laying out code from loops. However, there are two issues when taking into consideration superblocks:

- Even though we are lucky to have a fair `Linearize` for loops, this is only the case if the code indeed follows a numbering where the loop-body nodes have a greater value than the nodes of the exit branch. This property could be potentially undermined by future optimizations.
- The issue remains when considering non-looping branches that were predicted: the original `Linearize` heuristics, clueless of such branches, may guess wrong and lay out the wrong branch first.

Some work exists to solve the general problem of code positioning, such as Pettis and Hansen [72], that relies on branching-edge frequencies to devise a good positioning. Since we do not have exact frequencies in our representation, I settled for a much simpler solution that would extend the existing heuristics for superblocks without incurring any performance downgrade compared to the original heuristics when no branch prediction is provided.

The solution consists in modifying what I named `get_next_pc` in algorithm 12. The new function is in algorithm 13.

Algorithm 13. Modified `get_next_pc` for superblocks

```
 1: function GET_NEXT_PC(pc)
 2:     match last_instruction_of(pc) with
 3:         case Lbranch s
 4:             return (Some s, [])
 5:         case Lcond c _ ifso ifnot prediction
 6:             match prediction with
 7:                 case None
 8:                     return (None, [ifso, ifnot])
 9:                 case Some true
10:                     return (Some ifso, [ifnot])
11:                 case Some false
12:                     return (Some ifnot, [ifso])
13:         case i
14:             return (None, successors(i))
```

Another modification: instead of recording the minimum PC of the chain, the PC of the first block of the chain is recorded. I have measured slightly better performance in some instances, though it is mostly negligible for most of the benchmarks.

Through experiments (see chapter 10), I ensured that this new version indeed provides only slightly worse performance in the worst case.[4]

8.4 Static Branch Prediction

Static branch prediction (in the absence of profiling information) is the most critical heuristic of those presented in this chapter: an inaccuracy in branch prediction could result in a significant loss of performance due to the above heuristics taking wrong decisions.

In our case, we want branch prediction that is both reasonably accurate (as few false positives as possible) and yet permissive enough to allow for certain straightforward formations (such as loops) to be detected correctly. In particular, detecting the conditional branches of innermost loops is of great importance to us: this is where our optimizations will have a more significant impact. A bad prediction on an innermost-loop conditional branch would imply an impossibility to unroll it, or rotate it, or schedule it properly (since the header would then be on a different superblock than the body).

I based my work on Ball and Larus [4], in which several heuristics are given to infer the preferred direction of a given branch. I first briefly describe in 8.4.1 my implementation of the heuristics presented in Ball and Larus [4], then I introduce a new heuristic that I use for conditional branches within loops in 8.4.2.

8.4.1 Implementing Branch Prediction for Free

The heuristics of Ball and Larus [4] are described assuming an intermediate representation consisting of a control-flow graph of basic blocks. Then, a lot of their heuristics involve checking the instructions of the basic block for specific properties. In our case, this would be more directly suited to the `LTL` representation. Still, we can adapt it to `RTL` easily. Instead of iterating through the instructions of a basic block to check a property, we can traverse the control-flow graph node-by-node until we encounter an instruction with multiple successors.

For each conditional-branch instruction encountered, we can then check each of its edges to look for the properties used by the original paper. The first (and perhaps most important) property is for loop conditional branches: if a conditional branch has a backedge and a forward edge, they predict the backedge. Indeed, the backedge will be the edge used for looping; also, in general, and for a given loop, we loop more often than we exit the loop.

[4] A funny note: it is through debugging the new `Linearize` oracle performance that I was able to rule out many bugs related to faulty branch prediction. Performance bugs are rarely where you would expect them to be.

Some others of these properties involve checking the branch for the presence of certain classes of instructions like a return, a call, or a store. There is also a heuristic based on the comparison of the conditional branch: e.g. if it is a "less than 0" comparison, one might assume it is an error-checking branch and that the condition will not be taken most of the time.

Each heuristic is applied following a specific order - if no heuristic is successful, the branch is randomly predicted.

I have a certain number of differences compared to the original paper:

- I do not perform any random prediction when no heuristic applied: trying to over-predict proved experimentally detrimental to the performance, compared to just leaving the branch as unpredicted.
- Similarly, I found that the store heuristic was giving too many false positives: the average performance across our benchmarks was better with the store heuristic deactivated.
- I modified the recommended order of heuristics: doing the heuristics in the order [`opcode`, `return`, `loop`, `call`] proved to be more performant.
- I did not implement the guard and pointer-comparison heuristics for time reasons (they are more complex to implement). However, implementing them in the future could improve the performance a bit.

Another difference is that I modified the loop heuristic (checking whether the branch is a backedge) because it was unfit for CompCert. Indeed, in CompCert, the conditional branches are placed at the beginning of the loop, not at the end: both of their branches are then forward edges (not backedges). The actual backedge happens on the last instruction of the loop. So the loop heuristic, as described in the original paper, would have no impact on our case.

In the following subsection, I describe my modification for loop-branches prediction to account for this issue.

8.4.2 A new Heuristic for Predicting Branches Inside Loops

In CompCert, a loop consists of the loop header, then the conditional branch (possibly exiting the loop), then the actual loop body, which then jumps back to the loop header. In our case, the loop heuristic of Ball and Larus [4] would only work for simple innermost loops containing single basic blocks. As soon as there would be branches within the loop (such as an `if (cond) break;` construct, or nested loops), both the loop conditional branch and the other branches would not be predicted.

To tackle this issue, my new heuristics rely on computing the body of a given loop (computing the instructions that are part of the loop). This is done by first computing the backedges, then

applying an algorithm to deduce the natural loop spawned by the backedge (I use the `Nat_Loop` algorithm from figure 7.21, page 192, of Muchnick et al. [66]). The list of all loop bodies is computed by the `get_loop_bodies` function, not detailed here.

The new heuristic is then implemented through the `get_loop_info` function (algorithm 14): it outputs a `loop_info` mapping from nodes to predictions, which is then directly consulted by my new `loop` heuristic.

Algorithm 14. Predicting conditional branches inside loops

```
 1: function GET_LOOP_INFO(code, entrypoint)
 2:     loop_bodies ← get_loop_bodies(code, entrypoint)
 3:     loop_info ← { all ↦ None }
 4:     foreach body ∈ loop_bodies
 5:         foreach n ∈ body
 6:             match code[n] with
 7:                 case Icond _ _ ifso ifnot _
 8:                     if loop_info[n] ≠ None then
 9:                         continue
10:                     b1 ← (ifso ∈ body)
11:                     b2 ← (ifnot ∈ body)
12:                     if b1 && b2 then
13:                         continue
14:                     else if b1 then
15:                         loop_info[n] := Some true
16:                     else if b2 then
17:                         loop_info[n] := Some false
18:                 case _
19:                     continue
20:     return loop_info
```

I verified on most of our benchmarks that the loops were indeed correctly predicted and that `if (..) break` constructs did not break the prediction.

9

Implementing the Scheduling Oracles

Chapters 3 and 7 indicated how we formally verify scheduling respectively for basic blocks (at the Asm level) and superblocks (at the RTLpath level). In this chapter, I briefly describe the inner workings of our scheduling oracles.

For this thesis, we are interested in scheduling an in-order VLIW pipelined processor (though the scheduler can also be applied to other pipelined in-order processors, not necessarily VLIW). I refer the reader to chapter 1 for a description of the KVX core pipeline and the definition of VLIW and in-order processors.

Given a set of n instructions to be executed sequentially, scheduling is the problem of finding an execution order which minimizes the execution time without making the code incorrect. Formally, we want to compute a function $t : 0 \ldots n-1 \rightarrow \mathbb{N}$ assigning a time slot to each instruction. These time slots are used to reorder instructions accordingly; or, in the case of basic-block scheduling, produce bundles: the first bundle has all instructions j such that $t(j) = 0$, the next bundle has all those such that $t(j) = 1$, etc.

This schedule must satisfy several classes of constraints, given below.

- **Semantic and Latency Dependencies**
  Read and write dependencies are examined for each processor register, as well as memory. In both schedulers, we consider that a store always invalidates the whole memory (not just the address of the store) since our verifiers do not have any memory alias analysis yet. For the sake of expressing the dependencies below, the memory is treated like a register.
  - *Read after write*: If instruction j writes to register r and this is the last write to r before an instruction j' reading from r, then the schedule should respect $t(j') - t(j) \geq c(j, j')$, where $c(j, j')$ is the latency (in number of cycles) it takes for j' to be able to read the content of register r. Since certain instructions read their registers at different stages of the pipeline, and since some instructions take longer to compute their results than others, this latency depends on both j and j'. We computed those latencies by reading the KVX microarchitecture documentation. Since the KVX processor is interlocked, a wrong latency value will only incur worse performance; it will not

affect the architectural semantics (the processor will stall if the operand is not ready to be read yet).

  - *Write after write*: If instruction j writes to register r and this is the last write to r before an instruction j' writing to r, then the schedule should respect $t(j') - t(j) \geq c(j, j')$, with $c(j, j')$ described above.
  - *Write after read*: If instruction j reads from r, the next write to r is instruction j', then $t(j') - t(j) \geq 0$. Consequently, in the case of the KVX core, it is possible to have both a register read from r and a register write to r in the same bundle.

- **Resource-Usage Constraints**

  The processor has a limited number of processing units. Therefore, a bundle of instructions must not request more processing units of a given kind than available. Also, there is a limit on the number of instruction words ("syllables") inside a bundle. The assembler rejects bundles that do not abide by these rules.

  The architecture documentation describes these limitations as a constant vector of available resources $\mathbf{r} \in \mathbb{N}^m$ and, for each instruction kind k, a vector $\mathbf{u}(k) \in \mathbb{N}^m$. The constraint is that the sum of all $\mathbf{u}(K(j))$ for all instructions j scheduled within the same bundle i should be coordinate-wise less than or equal to $\mathbf{r}$.

These constraints are computed by a *scheduler frontend*, dependent on the IR level used for scheduling (namely, `RTLpath` or `Asmblock`), and given to a *scheduler backend*, independent of the IR used. This allows reusing the scheduling algorithms between the postpass and prepass. Also, most of the prepass scheduler frontend is independent of the target architecture to be reused for others, such as ARM or RISC-V.

First of all, I describe in section 9.1 the postpass scheduler: the frontend, then which backend algorithm we use. Then, I describe in section 9.2 the prepass scheduler: the frontend and which backend algorithms are available.

A significant portion of the work presented in this chapter was done by David Monniaux: the scheduling backend used by both schedulers and part of the frontend for superblock scheduling. I developed the frontend for basic-block scheduling, part of the frontend for superblock scheduling. Also, I conducted experiments to analyze the performance of the scheduled code (in chapter 10).

9.1 Postpass Scheduling Oracle

For basic-block scheduling (whose formal verification is shown in chapter 3), the scheduling oracle has the following Coq type:

```
1 Axiom schedule: bblock → (list (list basic)) * option control
```

Given a basic block, the scheduler should return a list of bundles (a bundle is a list of instructions), and optionally a control-flow instruction that is to be attached to the last bundle (the last bundle can be empty to emulate a control-flow instruction being alone in its bundle if that is the wish of the scheduler).

To get this result, first, the `bblock` is analyzed to extract the instruction dependencies and the resource vectors of each instruction, as well as possibly separating builtins from their basic blocks (subsection 9.1.1). Then, the backend operates through an algorithm that can be changed through the compiler options (subsection 9.1.2). Finally, the frontend interprets the result of the backend to pack it into the type required by the Coq module `PostpassScheduling` (not detailed here).

This work was initially done only for the KVX architecture, but it is possible to adapt the postpass scheduling verifiers and oracles to other architectures. Recently, this work has been ported to AArch64 by Léo Gourdin, Sylvain Boulmé and David Monniaux (see section 2.2 of Six et al. [89]), which provided non-negligible performance improvements (performance gain of about 10%).

9.1.1 Postpass Frontend

The frontend is in charge of giving correct information to the scheduler about the following:

- The resource vectors. These should not be under-specified, and it is better if they are exact. Suppose the resource vectors are under-specified (too few resources are allocated to a given instruction, whereas in reality, the instruction consumes more). In that case, it might incur a compilation failure from the assembler because of a bundle using too many resources. If, on the contrary, the resource vectors are over-specified, this would result in an under-optimized code because of missed scheduling opportunities.
- The latency dependencies. These should be enough to output a correct code, and it is better if they are exact. If too few latency dependencies are given, the output code might have a different execution result: our scheduling verifier would return an error. If, on the contrary, too many latency dependencies are given, the output code might be underperforming because some instructions would not be packed into bundles as they should.

I show in figure 9.1 an example of a basic block, along with the resource vectors (on a simplified model of the KVX), and the dependency graph:

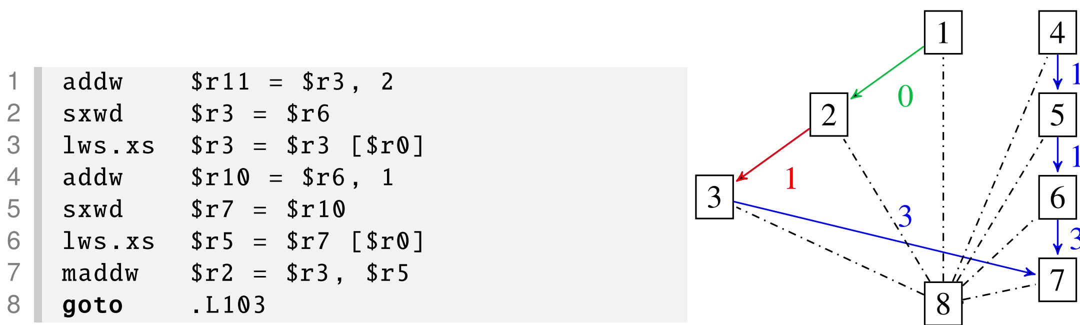


Figure 9.1.: Example of dependency graph. The Read-After-Write dependencies are in black; the Write-After-Read are in green; the Write-After-Write are in red. The dashed edges represent dependencies of latency 0.

- The instruction 2 has a write in `$r3` while the instruction 1 makes a read from `$r3`: this incurs a Write-After-Read dependency, of latency 0. As a result, instructions 1 and 2 might be in the same bundle, but instruction 2 should never be scheduled before instruction 1.
- The instruction 3 has both a Write-After-Write and a Read-After-Write with instruction 2, regarding register `$r3`. In that case, instruction 3 has to be scheduled at least one cycle after instruction 2: the duration of one cycle corresponds here to the time required for `$r3` to be available for the `lws.xs` instruction.
- The memory-load instructions (3 and 6) each incur a Read-After-Write dependency of 3 cycles: this is the duration required for a value to be fetched from memory, assuming that value is in the cache.
- The instruction 8 is a particular case: since it changes the control flow, it may not be scheduled before any other instruction. But it is still possible to schedule it simultaneously as a given other "basic" instruction. We model it by a dependency edge of latency 0.

The frontend computes these dependencies using a simple dependency analysis described in algorithm 15. Each instruction is examined in turn: their read and written registers are checked for RAW/WAW/WAR dependencies in regards to the previous accesses of the registers in read or write. These previous accesses are stored in the variables `written` and `read`, which are updated after each instruction is done analyzing. During this update, a write invalidates all the reads of a given register (if instruction 1 reads a given register, and instruction 2 writes to that same register, there is no need to keep track of the read of instruction 1). Finally, pseudo-WAR latencies are added from any basic instruction to the control-flow instruction to ensure the control-flow instruction remains at the end.

The frontend also computes the resources required by each instruction: for the KVX, the tables are available in its internal documentation.

Algorithm 15. Computing dependencies within the basic block

```
 1: function GET_ACCESSES(source_of, locations)
 2:     accesses ← []
 3:     foreach loc ∈ locations
 4:         accesses += source_of[loc]
 5:     return accesses
 6: function GET_DEPENDENCIES(instructions)
 7:     edges ← []
 8:     written ← { }
 9:     read ← { }
10:     foreach i ∈ instructions
11:         raw ← get_accesses(written, i.read_locs)
12:         waw ← get_accesses(written, i.write_locs)
13:         war ← get_accesses(read, i.write_locs)
14:         foreach j ∈ raw + waw
15:             edges += (j, i, latency(j, i))
16:         foreach j ∈ war
17:             edges += (j, i, 0)
18:         if is_control(i) then
19:             foreach j ∈ instructions - [i] edges += (j, i, 0)
20:         foreach loc ∈ i.write_locs
21:             written[loc] := [i]
22:             read[loc] := []
23:         foreach loc ∈ i.read_locs
24:             read[loc] += [i]
25:     return edges
```

```
type latency_constraint = {
  instr_from : int;
  instr_to : int;
  latency : int;
  }

type problem = {
    max_latency : int;
    resource_bounds : int array;
    instruction_usages: int array array;
    latency_constraints : latency_constraint list
  };;
```

Listing 9.1.: Definition of a scheduling problem

Finally, once the dependencies and resources are computed, it calls the backend, which returns a schedule and translates that schedule into a list of bundles (two instructions scheduled at the same cycle are bundled together), which is returned back to the Coq verifier.

9.1.2 Postpass Backend

The scheduler backend operates on a type abstracting only the necessary information, named `problem` (listing 9.1).

A scheduling problem is encoded with:

- A vector of available resources of the processor. If no such information is available, it is possible to give an empty array so that the scheduler will disregard resource constraints.
- An array of resource vectors. The array represents the list of instructions to schedule: to each instruction corresponds the resource vector required by the instruction.
- The latency constraints, encoded as a list of (`from`, `to`, `lat`) tuples where an instruction `to` must wait at least `lat` cycles after `from` in the schedule. This encodes a graph of dependencies where the nodes are the instructions to be scheduled (the indices of `instruction_usages`), and the edges are the latency dependencies. If there is a constraint of latency 0 between f and t, this means that the instruction t cannot be scheduled before instruction f (but they may be scheduled at the same time, to be in the same bundle).

Our default solver is based on a variant of Coffman-Graham list scheduling (see Dupont de Dinechin [25] and §5.4 of Micheli [62]) with one heuristic: instructions with longer latency paths to the exit get priority. This is fast (quasi-linear time) and computes an optimal schedule in almost all practical cases.

To analyze the impact of list scheduling, we conducted the following experiments:

- David Monniaux implemented the reduction of the scheduling problem to Integer Linear Programming, or pseudo-Boolean linear optimization, solved by an external tool (e.g. Gurobi). This showed that linear scheduling indeed returns an optimal schedule across almost all of our benchmarks for the scheduling model that we use. More details can be found in Six et al. [88], appendix D.
- I compared the performance results with a greedy scheduler that packs instructions into bundles without reordering them and with a trivial scheduler returning one instruction per bundle, without any reordering. The results of this comparison are in chapter 10.

9.2 Prepass Scheduling Oracle

The prepass oracle has the following Coq type:

```
1 Axiom untrusted_scheduler: RTLpath.function → code * node * path_map * (PTree.t node)
```

There are several differences compared to the postpass oracle type:

- The oracle performs scheduling of a whole RTLpath function, instead of just one block. However, the oracle does not have to recompute the blocks to schedule: these are taken directly from the pathmap of the RTLpath function.
- The scheduling is done at the granularity of a superblock, instead of a basic block: there will be additional latencies compared to a basic block.
- The list of instructions to schedule has to be computed from the RTL control-flow graph.
- Unlike Asm, we do not know the exact resources and latencies of each RTL instruction. We have to make educated guesses.

To tackle these differences but still reuse the backend of postpass scheduling, we used the code architecture given figure 9.2.

- `InstructionScheduler` is the postpass backend that also serves as the backend for the prepass. In addition to list scheduling, David Monniaux added reverse list scheduling and zigzag scheduling, which I informally describe in subsection 9.2.2.
- `PostpassSchedulingOracle` is the postpass frontend, which was described in section 9.1.2.
- `Opweights` is a new module for prepass written by David Monniaux that gives an educated guess of the resources and latency of each operation, condition, load and store. It uses some functions of the `Asmblockgen` module (part of the assembly backend for KVX) for getting an approximate translation of the concerned RTL instructions. Then, part of the

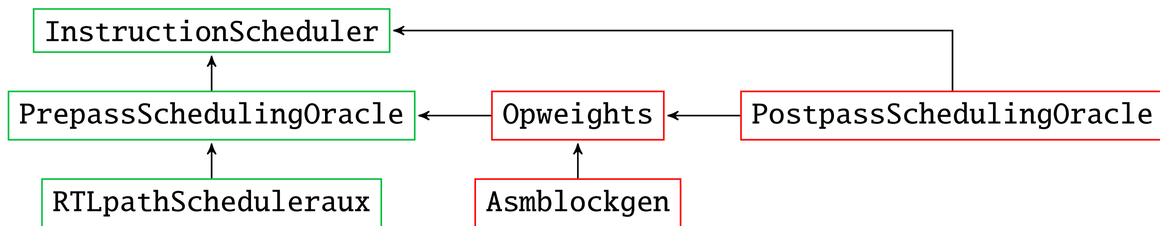


Figure 9.2.: Our code architecture for implementing the schedulers. The architecture-dependent parts are in red, the architecture-independent parts are in green.

`PostpassSchedulingOracle` module is used for getting the used resources and underlying latency. When a given operation would be translated into several assembly instructions, only the first instruction is taken into account. This gives an under-approximation of the actual latency and resource cost. In the future, this could probably be more finely tuned to give more accurate approximations.

- Since we work on whole CFGs instead of instruction lists, the `RTLpathScheduleraux` module does the adaptation of translating CFGs into lists of instructions to schedule and then modifying the CFG based on the schedule returned by the scheduler. It also turns trapping loads into non-trapping loads as necessary, based on whether they were scheduled before a given conditional branch.
- `PrepassSchedulingOracle`, implemented by David Monniaux, computes the dependencies and latencies in much the same way as our postpass frontend, except that it uses the approximated resources and latencies from `Opweights` and adds additional dependencies for the conditional branches (see subsection 9.2.1). It also introduces zigzag scheduling as the combination of a forward and a backward list scheduler (see subsection 9.2.2).

Out of these newly introduced modules for prepass, only `Opweights` is target-dependent and needs to be readapted for each architecture. Compared to postpass, this allows having a prepass scheduler almost "for free" for any architecture supported by CompCert, as long as the approximated latencies/resources are entered in an `Opweights` module. David Monniaux, in particular, implemented an `Opweights` for RISC-V and Aarch64, so our prepass superblock optimization can also be used on these architectures.

9.2.1 Additional Dependencies for Conditional Branches

The scheduler should be careful not to schedule instructions that might trap on top of conditional branches. If the instruction may trap (this is the case for non-speculative memory loads, memory stores, and certain operations such as integer division or modulo), then an additional dependency of latency 1 is added between the instruction and the last conditional branch above the instruction.

```
1   if (x13 < x14) goto OUT
2   x13 = x1 + -1
3   if (x3 ==s x13) goto ELSE
4   x12 = long32signed(x3)
5   x8 = int32[x2 + (x12 << 2)]
6   x11 = x3 + 1
7   x10 = long32signed(x11)
8   x9 = int32[x2 + (x10 << 2)]
9   x15 = x4 + x8 * x9
10  x16 = x3 + 2
11  x17 = long32signed(x16)
12  x18 = int32[x2 + (x17 << 2)]
13  x19 = x15 + x18
```

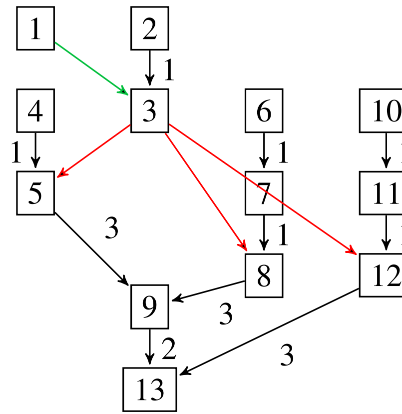


Figure 9.3.: Example of dependency graph of a superblock, for architectures that do not have speculative loads. The usual RAW dependencies are in black, the extra conditional-branch dependencies coming from trapping instructions are in red, the additional dependencies between the conditional branches are in green.

Also, since our superblock verifier compares exit states two-by-two, the scheduler must keep the conditional branches in the same order: a conditional branch might not be scheduled on top of another. This is handled by adding an additional dependency of latency 1 for each successive conditional branch.

I give in figure 9.3 an example of linearized RTL code with two conditional branches. Compared to a typical code from Asm, there are barely any Write-After-Read or Write-After-Write dependencies, thanks to the infinite amount of registers available. This creates fewer constraints for scheduling. To account for the conditional branches, two constraints are added: an edge of latency 1 between instructions 1 and 3, to conserve the order of conditional branches; and edges of latency 1 between 3 and the load instructions, to ensure these are not scheduled before (though, in the case of the KVX, these edges are not added: instead, we make the memory loads speculative).

9.2.2 Prepass Scheduling Algorithms

We have three possible prepass scheduling algorithms available:

- Forward list scheduling: the same algorithm from the postpass scheduling, filling the time slots greedily, with the heuristic of privileging the longest execution path when given the choice.
- Backward list scheduling: forward list scheduling, but applied on a *reverse* problem where the edges from the dependency graph have been inverted, and the timeslots are allocated

backwards. As was briefly presented in 4.3.2, the advantage of backward list scheduling compared to forward scheduling is to decrease register pressure.

- Zigzag scheduling: it uses forward scheduling to place the conditional branches, then locks them on particular timeslots by using special dependencies and runs backward scheduling. This was an attempt at implementing a mix between forward scheduling, which tends to deliver better schedules when register allocation is not taken into account, and backward scheduling, which balances it out.

Compared to postpass scheduling, there is no bundling here: and yet, an order has to be chosen for instructions scheduled at the same time slot. We arbitrarily keep the instructions within the same time slot in the same order as they originally appeared in the superblock.[1]

To illustrate these different schedules, I give in figure 9.4 the result of the three schedules on the code given earlier in figure 9.3, for the KVX architecture (supporting speculative loads: so the extra conditional branch dependencies are ignored).

Since we only have few dependencies, the list scheduler can schedule many instructions from the first timeslot (instructions 1, 4, 6 and 10). The stalls from the load instructions 5 and 8 are partly mitigated (no stall for 5, one stall for 8). In this example, and according to our approximated latencies, list scheduling would result in the last instruction being scheduled at timeslot 7. This is a good time. However, a disadvantage of list scheduling can be seen: many registers are live at each given time. For instance, the register x12, x11, x16 and x13 are both live at timeslot 0.

Reverse list scheduling gives a schedule with a similar final timeslot (the last instruction is also scheduled at timeslot 7), but fewer registers are live at a given time: for timeslot 0, only the registers x12 and x11 are live. However, since the schedule was done in reverse, the conditional branches that are usually at the beginning of the block find themselves at the end, so if our predictions end up incorrect, we will get a significant performance penalty (many instructions executed that end up being discarded).

Zigzag scheduling fixed the conditional branches to the timeslots that were predicted in forward scheduling and then tried backward scheduling with these additional constraints. With zigzag scheduling, we have fewer instructions scheduled before conditional branches. Still, we pay the tradeoff with higher latency: the final instruction is scheduled at timeslot 8 (instead of timeslot 7 for the other two schedules).

We measured the impact of each of these three schedules on the KVX and other architectures in chapter 10: for the KVX architecture, list scheduling gives slightly better results than reverse

[1]This could be slightly improved by ordering the basic instructions before a conditional branch (as long as it respects the WAR dependencies) when they all are affected to the same timeslot: the postpass scheduling would then be able to bundle the instructions with the conditional branch.

```
// List: [0 ↦ {1, 4, 6, 10};
//    1 ↦ {2, 5, 7, 11}; 2 ↦ {3, 8}
//    3 ↦ {12}; 5 ↦ {9}; 7 ↦ {5}]
1:  if (x13 < x14) goto OUT
4:   x12 = long32signed(x3)
6:   x11 = x3 + 1
10: x16 = x3 + 2
2:   x13 = x1 + -1
5:  x8 = int32.s[x2 + (x12 « 2)]
7:   x10 = long32signed(x11)
11: x17 = long32signed(x16)
3:  if (x3 ==s x13) goto ELSE
8:  x9 = int32[x2 + (x10 « 2)]
12:  x18 = int32[x2 + (x17 « 2)]
9:   x15 = x4 + x8 * x9
13: x19 = x15 + x18
```

```
// Rev: [0 ↦ {4, 6}; 1 ↦ {5, 7};
//    2 ↦ {8, 10}; 3 ↦ {11};
//    4 ↦ {12}; 5 ↦ {9};
//    6 ↦ {1, 2}; 7 ↦ {3, 13}]
4:   x12 = long32signed(x3)
6:   x11 = x3 + 1
5:  x8 = int32.s[x2 + (x12 « 2)]
7:   x10 = long32signed(x11)
8:  x9 = int32.s[x2 + (x10 « 2)]
10: x16 = x3 + 2
11: x17 = long32signed(x16)
12:  x18 = int32.s[x2 + (x17 « 2)]
9:   x15 = x4 + x8 * x9
1:  if (x13 < x14) goto OUT
2:   x13 = x1 + -1
3:  if (x3 ==s x13) goto ELSE
13: x19 = x15 + x18
```

```
// Zigzag: [0 ↦ {1}; 1 ↦ {2, 4, 6}; 2 ↦ {3, 5, 7}; 3 ↦ {8, 10};
//    4 ↦ {11}; 5 ↦ {12}]; 6 ↦ {9}; 8 ↦ {13}]
1:  if (x13 < x14) goto OUT
2:   x13 = x1 + -1
4:   x12 = long32signed(x3)
6:   x11 = x3 + 1
3:  if (x3 ==s x13) goto ELSE
5:  x8 = int32[x2 + (x12 « 2)]
7:   x10 = long32signed(x11)
8:  x9 = int32[x2 + (x10 « 2)]
10: x16 = x3 + 2
11: x17 = long32signed(x16)
12:  x18 = int32[x2 + (x17 « 2)]
9:   x15 = x4 + x8 * x9
13: x19 = x15 + x18
```

Figure 9.4.: Our three possible prepass schedules on the previous example. The code is annotated with the initial position of each line of code for clarity. I also show the "time ↦ instructions" mapping of each schedule. The conditional branches are outlined in red. The load instructions are marked in green, and the conditional branches in red, for easier recognition.

list scheduling; for the other architectures, reverse list scheduling tends to provide better results, although it is very dependent on the benchmarks.

10 Experiments

I introduced in the previous chapters formally verified techniques to optimize the generated code from CompCert: the basic-block scheduling from chapters 3 and 9, the loop unrolling and loop rotation from chapters 6 and 8, the superblock linearizer from chapter 8, and finally, the superblock scheduling from chapters 7 and 9.

In this chapter, I evaluate the impact of each of these optimizations, mainly on the KV3 architecture and other architectures such as AArch64 and RISC-V, thanks to the porting efforts of Léo Gourdin, Sylvain Boulmé, and David Monniaux (see Six et al. [89]). Throughout my thesis, I focused only on the KV3 architecture.

I first present the benchmarks used and the means of measuring performance in section 10.1. Secondly, I study the impact of the postpass scheduling on KV3 and its experimental time complexity in section 10.2. Then, I look in section 10.3 at the impact of all optimizations related to superblock scheduling: superblock linearization, the effect of loop unrolling, rotation; I also study the impact of Loop Invariant Code Motion, introduced by D. Monniaux [63]. Finally, I do a performance breakdown of our best version of CompCert compared to GCC in section 10.4.

10.1 Experimental Setup

Evaluating the performance of a given optimization has a certain number of challenges.

A given optimization might affect the performance impact of another optimization. A typical example: when developing the first iteration of superblock scheduling, I found that the performance was overall not as good as I would have imagined it. In some cases, it was even detrimental. My first thought was that probably there were bugs in the scheduler or some wrong latencies. But the actual reason was just that there had been a bug that had gone unnoticed in the static branch prediction. As a result, some loops were mispredicted, resulting in a superblock crossing the loop instead of staying within. As a result, instructions that did not belong to the loop found themselves inside it: we had achieved reverse loop-invariant code motion, in a way. Because of this issue, it is essential to have a way to easily examine the assembly generated by any

benchmark so that the assembly code might be easily looked up on any benchmark producing abnormal performance.

For specific optimizations, it is not clear which "way" is better. For instance, list scheduling, reverse list scheduling and zigzag scheduling have their strengths and weaknesses, but it is impossible to know before experimenting. This experimentation has to be done on a significant number of benchmarks because a given scheduling method might be biased for a specific source-code pattern. Also, there must be a way to easily compare different sets of flags.

Finally, throughout my thesis, we asked ourselves the following question: "which optimizations can we implement to bridge the gap as best as we can, within the time and resources that we have". In some cases, looking at the code generated by the other compilers (GCC and LLVM) has been a way to try to infer what was missing from our version of CompCert and then try to close the gap a little bit further. For this reason, it was essential to have our benchmarks also compiled with GCC and LLVM.

David Monniaux and I implemented a benchmark harness (not detailed here) working with "flattened" benchmarks (transformed in such a way that they can be compiled with simple rules). We implemented a standardized way of compiling, running and measuring all benchmarks, keeping track of all intermediate files (assembly, binary, standard output log files) from different compilers and different sets of flags. This facilitated our production of results, as well as our subsequent analysis.

I detail in this section the settings of our measurements: subsection 10.1.1 detail the processors used, subsection 10.1.2 gives our method of measuring performance, and subsection 10.1.3 describes the benchmarks that we used.

10.1.1 Measured Processors

Within the duration of this thesis, I measured the benchmarks on the KV3 processor which was presented in chapter 1. I recall the main characteristics here: it features a 6-issue VLIW architecture with an instruction pipeline of 8 stages. It has 64 general-purpose registers of 64-bits used for both integer and floating-point data (a given register can store an integer or a float). It also features support for scalar vectors and SIMD instructions. Still, we do not take advantage of that, except for some memory loads that we group into octuple and quadruple loads when appropriate, thanks to a peephole optimization coded by David Monniaux verified at the same time as basic block scheduling. The instruction-cache capacity is 16 kB, and the data cache

also has 16 kB. When performing a memory load, if the value is in the cache, only 3 cycles are required.[1]

David Monniaux and Léo Gourdin also experimented on an ARM Cortex A53 (AArch64) inside a Raspberry Pi 3 and a Rocket RISC-V core [56] running on an FPGA. Both run under a Linux operating system which produces some noise on the performance measures.

10.1.2 Measurement Methods

For the benchmarks executed on the KV3 processor, we use the hardware performance counter of the number of cycles. Similar counters are used for AArch64 and RISC-V. To prevent timing IO system calls (such as sending a message to the standard output), which can be very costly on the KV3, we isolate the computing part and measure only that part.

The KV3 is a predictable architecture as long as the execution remains within a compute cluster: in our case, executing several times one of our benchmarks gives the same result in the number of cycles. There is then no need to have multiple executions. However, for RISC-V and AArch64, the benchmarks were executed 20 times each and averaged to reduce the measuring noise.

In some figures, I show the compilation time: these are either obtained through time counters added by instrumenting the generated OCaml code or using the already present time counters of CompCert.

I also provide some metrics to measure the validity of our static branch prediction: this was obtained by instrumenting the code to force a static prediction pass even when profiling was present and increment relevant counters based on the results from static branch prediction.

Finally, to study the impact of instruction-cache misses for certain optimizations, I measured the number of cycles spent waiting for the instruction cache to deliver data to the prefetch buffer. This is done through a specific counter of the KV3 named “Prefetch Buffer Starvation Cycle”, whose description in the manual is “+1 at every cycle during which a PFB (Prefetch Buffer) fetch request to the ICache is not granted by the latter”.

10.1.3 Benchmarks Used

We evaluated CompCert on approximately 90 different benchmarks. On AArch64 and RISC-V, about a fourth of them were removed because their execution time was too low; they were

[1] If there is a cache miss, the data cache needs to perform a line refill: a refill stops any request to/from the data cache for approximately 20 cycles in most cases.

too sensitive to the external environment (operating system). In particular, on RISC-V, the experimental noise is significant. On KV3, we do not have this issue since there is no variation in the number of cycles across several executions, so they are all evaluated.

Polybench

The Polybench benchmarks include computational kernels, originally meant for benchmarking polyhedral optimizations [76]. I chose to port this benchmark to our harness because they are easy to benchmark and analyze. Thanks to their minimalistic structure, the computational part is clearly identified in all benchmarks. Any performance issue is then relatively easy to track on the C source code to trace back the cause to our optimizations.

Polybench code typically includes one or several nested loops, with little to no branching within them.

TACLeBench

TACLeBench is a collection of benchmarks used for worst-case-execution-time research[27]. Compared to Polybench, they have a wider variety of code patterns. In particular, most of them feature branches within their hot computational part. Like Polybench, they feature several extracted kernels that are easy to analyze. Léo Gourdin recently ported this benchmark to our harness.

Handpicked Benchmarks

David Monniaux originally started by gathering particular benchmarks of interest relevant to the Kalray KV3 core: critical embedded systems and computational benchmarks.

Among them, we have:

- *radiotrans*, *convertible* and *heater-control*: these are benchmarks compiled from synchronous dataflow programming languages (respectively Heptagon, Lustre v6 and Lustre v4). For instance, such languages are used to specify and implement fly-by-wire aircraft controls [9].
- *lift*: directly taken from TACLeBench
- *bitsliced-aes* and *sha-256*: cryptography primitives taken from Mosnier [64] and Patrick [71].
- *glpk* runs GLPK (GNU Linear Programming Kit [61]) on an example.
- *picosat* is an optimized SAT solver [11] run over a Sudoku example.

- *float-mat* is a textbook implementation of floating-point matrix multiplication.
- *float-mat-v2* is a version with loop unrolling done at the source level. Comparing *float-mat* and *float-mat-v2* typically gave us a first clue to the performance we could gain on a similar code, before implementing the optimizations.
- *jpeg-6b* is from the `libjpeg` [48].

I denote these benchmarks as the **handpicked** benchmark.

10.2 Impact of Postpass Scheduling

In this section, we study the impact of our postpass-scheduling optimization from chapter 3, both in performance and compilation times.

10.2.1 Performance

On KV3 Core

Instruction Level Parallelism (ILP) is achieved through two means:

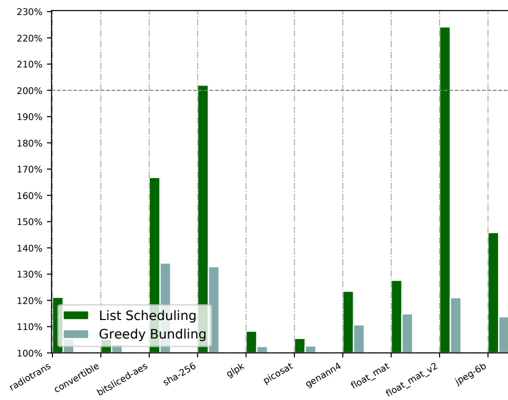
- Instruction pipelining: the KV3 has a pipeline of 8 stages (though in practice, only the first 4-6 stages are used most of the time). This many instructions can be processed simultaneously, given there is enough to feed the pipeline, and no stall (due to an interlock) is introduced.
- Issuing multiple instructions simultaneously: this is done through the VLIW design of the architecture. Up to 6 instructions can be issued together.

Unscheduled code, with only one instruction per bundle, can be expected to run quite poorly: not only is there only one instruction issued at a given time, but also, the code is likely to run into stalls, typically because of an instruction using the result of a memory load right away.

Postpass scheduling is the first step in fixing this performance issue by reordering code to minimize stalls and bundling together instructions.

In figure 10.1, I measure the impact of bundling alone (without any reordering: just packing together instructions into bundles, greedily) and of our actual postpass scheduling (which performs both reordering and bundling).

A first result: for most benchmarks, on the KV3, postpass scheduling gives a performance increase of at least 30%. Some benchmarks such as *convertible*, however, are not affected much:



Setup	KV3		
	Q1	Med	Q3
Greedy Bundling	+6.28%	+13.03%	+18.57%
List Scheduling	+13.96%	+31.25%	+46.62%

Figure 10.1.: Performance gain of postpass scheduling and “greedy bundling” on KV3. The left figure shows each of our handpicked benchmarks. The right figure shows the results across all the benchmarks.

the main loop features around 800 different variables in the same scope, which do not fit into the 64 registers of the KV3. Register spills are thus generated, which prevents scheduling since we consider memory to be invalidated entirely by a store instruction. We would require an alias analysis to solve this issue: then, certain memory accesses could be reordered.

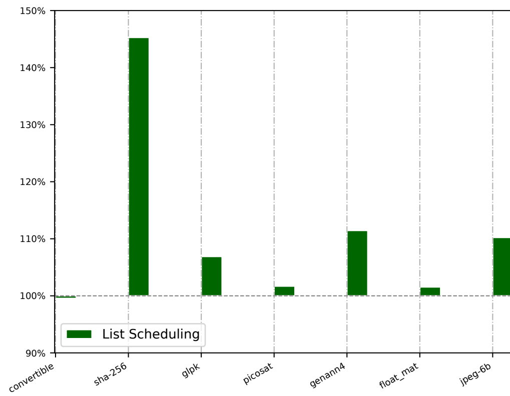
A second result is that reordering has a much more significant impact than just bundling instructions together. On the KV3 architecture, stalls have a substantial effect on performance.

Another interesting result: though greedy bundling did not affect much of our `float_mat_v2` benchmark, list scheduling dramatically improved its performance. This gave us a glimpse that loop unrolling can have a significant impact on performance.

On AArch64 Core

Léo Gourdin and David Monniaux ported the KV3 postpass scheduling module to AArch64 by using latency information from LLVM. I display the results in figure 10.2.

Certain benchmarks can profit significantly from the postpass scheduling on AArch64. However, it is less potent than on the KV3 architecture and less consistent: a bit more than a quarter of the benchmarks are barely affected (+5% gain on the first quantile). However, a quarter of the benchmarks have a performance gain of at least 20%, thanks to postpass scheduling.

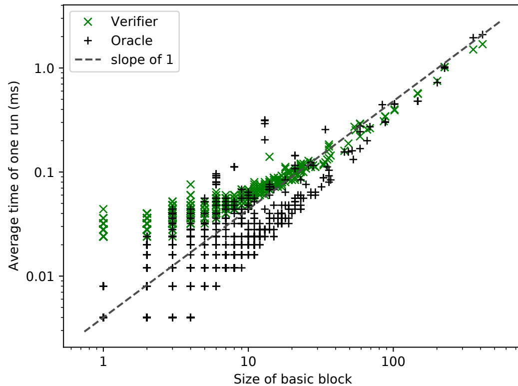


Setup	AArch64		
	Q1	Med	Q3
Postpass Scheduling	+8.42%	+17.08%	+23.71%

Figure 10.2.: Performance gain of postpass scheduling on AArch64. The left figure shows each of our handpicked benchmarks. The right figure shows the results across all the benchmarks.

10.2.2 Compilation Times of Postpass Scheduling on KV3

To ensure that hash-consing works as intended for reducing the execution cost of the optimization, I instrumented the OCaml generated code to include timers and basic-block-size counters. I also measured the execution timings on the particular benchmark *bitsliced-aes*, which contains basic blocks with more than 100 instructions. The results are displayed in figure 10.3.



Compiling *bitsliced-aes*	
Time	Transformation pass
0.02s	Parsing
0.01s	Elaboration
0.04s	Common Subexpression Elimination
0.02s	Constant Propagation
0.02s	Redundancy Elimination
0.08s	Register Allocation
0.01s	Postpass Scheduling

Figure 10.3.: On the left, compilation times based on the basic-block size for all our benchmarks (except TACLeBench, which was added recently) in logarithmic scales. On the right, the timings of compiling *bitsliced-aes* (showing only the passes with non-negligible computational time).

Each point in the left figure corresponds to an actual basic block from our benchmarks, verified and scheduled (for the list scheduler) 1000 times. The verifier is generally a little slower than the oracle, but both are experimentally of linear complexity. The biggest basic block we came across, of around 500 instructions, was scheduled and verified in approximately 4 ms, which

is the same time required for other CompCert optimizations such as constant propagation or common subexpression elimination. The timings of the *bitsliced-aes* benchmark show that the compile times are in line with other optimization passes of CompCert.

10.3 Impact of Superblock-Scheduling-Related Optimizations

In this section, we study the impact of most of the optimizations introduced in chapters 7, 8 and 9:

- Ensuring that our static branch prediction is not detrimental (because of wrong predictions).
- Testing the performance of our modified linearizer compared to the original one.
- Evaluating different threshold values for our loop-unrolling and loop-rotation optimizations.
- Evaluating the different prepass strategies: list scheduling, reverse list scheduling, zigzag scheduling.
- Evaluating the performance gain of each optimization individually.

10.3.1 Evaluation of our Static Branch Prediction

To evaluate our static branch prediction, we conducted several experiments.

First of all, we measured the number of wrong predictions experimentally by cross-referencing the guesses with those of the profiling. Our static branch prediction returns, for each conditional branch, either `None` if no prediction could be made or `Some true` (resp. `Some false`) if the condition was predicted to be evaluated to `true` (resp. `Some false`). Our profiler outputs the same kind of information, based on the execution count of each branch: `None` if both branches were executed an equal amount of times each, or `Some b` with `b` a boolean indicating which branch got executed more. I then distinguish four different cases to evaluate our static branch prediction:

- Correct predictions: the static prediction guessed a branch that was indeed taken more often in the concrete execution.
- Mispredictions: the static prediction guessed the branch that was taken the least often (it should have either guessed the other one instead or not guessed anything).
- Missed opportunities: the static prediction did not guess anything, and yet there was a privileged branch in the execution.

- No verdict: the profiler did not give any verdict. This could happen either because both branches were executed equally or because the code was not reached during profiling. In these cases, we can not evaluate the static prediction.

In addition, we also counted the number of right and wrong predictions for each used heuristic: opcode (automatic prediction for certain classes of comparisons), return (predicting the branch that does not lead to a return instruction), loop (testing if the branch remains within the loop) and call (predicting the branch that does not have a call instruction) heuristics.

Correct Predictions	3551
Wrong Predictions	926
Missed Opportunities	2605
No verdict	27234

Heuristics	Right Predictions	Wrong Predictions
Opcode	319	108
Return	882	223
Loop	1284	125
Call	1066	470

Figure 10.4.: On the left, statistics of the static branch prediction, evaluated in regards to profiling. On the right, details for each heuristic.

We show the results in figure 10.4. Ideally, we would like as few wrong predictions as possible since these may compromise our scheduling optimization by scheduling the "wrong" superblocks. In our benchmarks, among the conditional branches for which profiling could give a verdict, about 13% of them were mispredicted, and roughly 50% were predicted correctly. Missed predictions are numerous (36%) but not harmful: they will just result in a lack of optimization.

A significant number of conditional branches could not be evaluated by the profiler: we believe this is because we have a low code coverage on our benchmarks for the datasets we use.

Another way to evaluate our static branch prediction is to replace it by profiling and look at the performance difference. The setup is the following: we run our best version of CompCert (with all the optimizations presented in this thesis activated and on their best versions) with static prediction, then we run our best version of CompCert but with profiling information instead of static branch prediction, then we compare them. The results are given in figure 10.5.

All of the Polybench benchmarks except one are untouched by using profiling: this is because their code is simple enough that the implemented heuristics were able to guess correctly. One benchmark suffers a 4% penalty, probably because our profiling got too sure of itself, inferring the same behaviour as was described in section 4.1.2: right now, our profiling gives a verdict if $n_{ifso} > n_{ifnot}$ with n_{ifso} and n_{ifnot} being the execution counts of each branch. This can lead to a performance penalty if an instruction is scheduled ahead of two consecutive branches for which the execution counts are very close.

The gain of profiling is much more prominent on TACLeBench, because of the code being overall more complex. In the general case, the gain is negligible, except for a fifth of the benchmarks

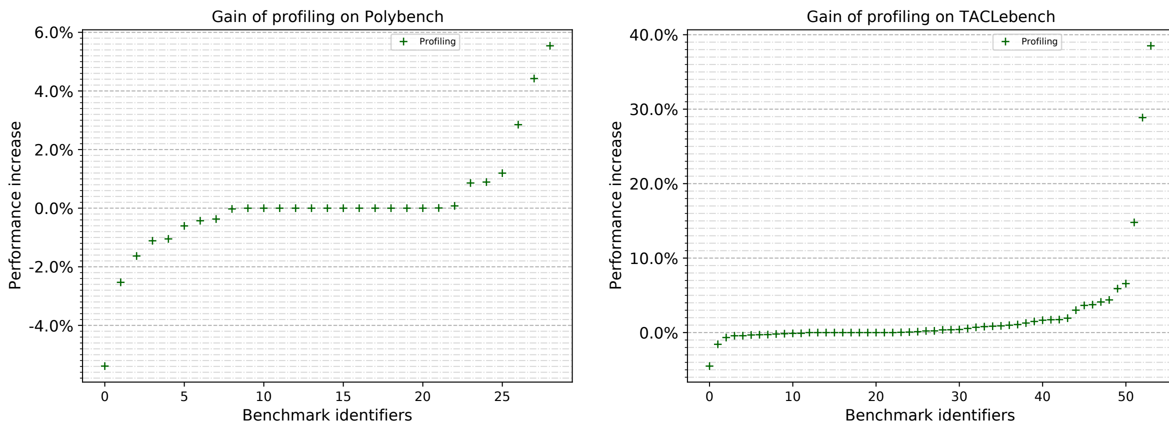


Figure 10.5.: Gain of using profiling instead of branch prediction. On the left, we evaluate this gain on the Polybench benchmarks. On the right, we evaluate it on the Taclebench benchmarks.

where the gain is between 2% and 10%. One benchmark, in particular, has a very high gain when using profiling (a benchmark implementing Djikstra's algorithm): 35% performance is gained, most probably because of a static-branch-prediction heuristic failing in a hot part of the code.

These performance results combined with our measures of predictions give us an informal guarantee that our static branch prediction performs well enough for most of the encountered cases.

10.3.2 Evaluation of our Superblock Linearizer

I evaluate here the superblock linearizer's performance in regards to the base linearizer from CompCert, on two setups.

The first setup is the baseline CompCert with just postpass scheduling activated and the other is the same setup but with branch prediction and superblock linearization activated. Evaluating this without prepass is a way to isolate the effect of linearization away from prepass scheduling to assert whether this linearizer version is (in the absence of prepass) better or worse than the base version.

The second setup is like the first, except that we include the prepass and instruction-duplication optimizations (loop rotation, loop unrolling, tail-duplication). This setup evaluates the real impact of linearization when combined with all the passes introduced in this thesis.

The graphs from figure 10.6 show the impact of adding superblock linearization for these two scenarios.

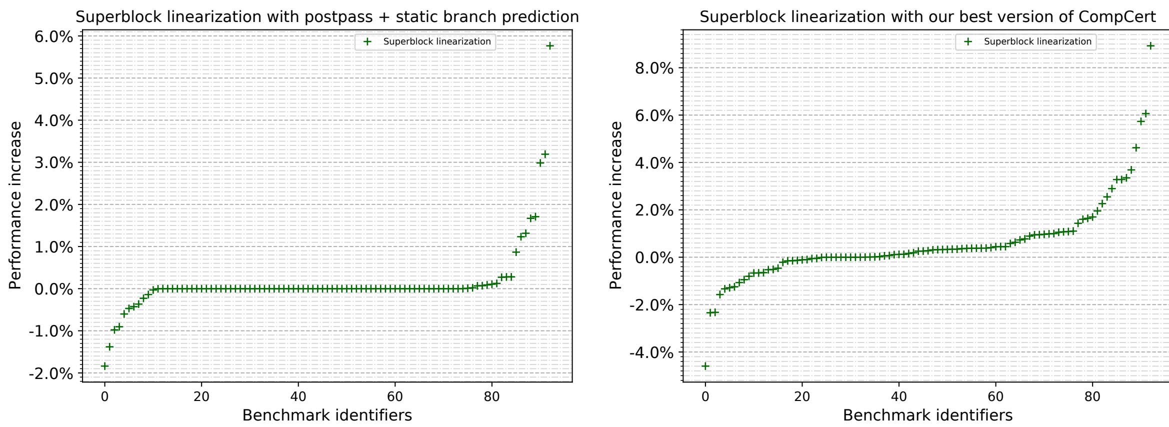


Figure 10.6.: Gain of using superblock linearization instead of base linearization. On the left, the setup includes just postpass scheduling and static branch prediction. On the right, it includes the best version of our optimizations.

In the case where we only run postpass scheduling and static branch prediction: for a few benchmarks, our linearization performs slightly worse, though it is mostly negligible: only 2% in the worst case. This can be explained by the presence of certain wrong predictions. A couple of benchmarks see an improvement from using our modified linearizer, though the gain is small.

For the case where all optimizations are run, our linearization has a slightly more significant impact, though it barely ever goes beyond 4% performance improvement. On the other hand, it also incurs slightly worse performance on a couple of benchmarks, probably because the various instruction-duplication passes aggravated the cases where superblocks were not representative enough of the actual execution flow.

10.3.3 Evaluation of Loop Unrolling, Loop Rotation and Tail-Duplication Thresholds

Our heuristic for deciding whether to loop-unroll, loop-rotate or tail-duplicate right now is very barebones: we count the number of instructions that would be duplicated (except *nop* instructions), and if it is below a certain threshold, we unroll, rotate or tail-duplicate accordingly.

More is not necessarily better in this case: a too-big threshold would imply too many duplicated nodes. This could then impact performance if the instruction-cache is refilled too often due to the increased code size.

I measured the impact of different levels of threshold on instruction cache misses and performance as a whole for the KV3 architecture. I measure in figure 10.7 four different optimizations: loop

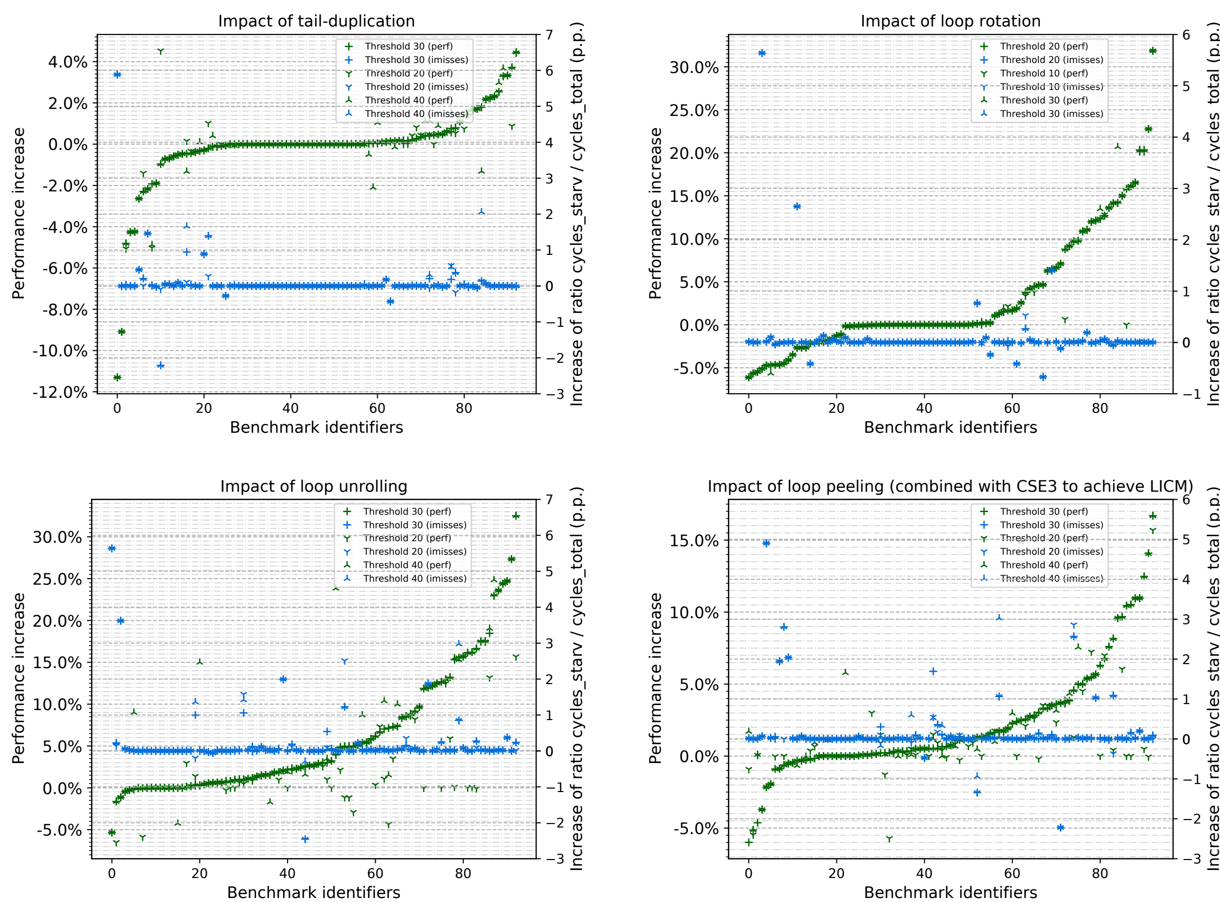


Figure 10.7.: Impact of the four instruction-duplication optimizations on performance and pipeline stalls from instruction-cache misses.

rotation, loop unrolling, tail-duplication, but also a form of loop peeling that we use for loop-invariant code motion. For each optimization, the other 3 are deactivated. So, for instance, to measure the effect of loop rotation, I deactivate loop unrolling, tail-duplication and loop peeling.

The figure is produced in the following way:

- For each benchmark and each threshold level (three of them), the relative performance increase is computed compared to the reference (the setup where the optimization is absent). To ease reading, the middle threshold level is represented by a cross, the upper threshold is a symbol pointing upwards, and the lower threshold is a symbol pointing downwards.
- We then order the benchmarks in increasing order of this relative performance boost, according to the *topmost threshold* in the legend. This topmost threshold is the middle threshold. This allows having a graph easier to read and forms a baseline to evaluate the lower and higher thresholds. For instance, for loop rotation, we order the benchmarks

based on the performance of loop rotation with a threshold of 20: this allows then to compare the thresholds of 10 (lower) and 30 (higher) relatively to this threshold of 20.

- In parallel, from the measure of the number of cycles spent waiting for the instruction cache, I compute the percentage of time spent waiting (because of an instruction-cache miss), i.e. the ratio $n_{PFB_starvation_cycles}/n_{total_cycles}$. I display the increase of this ratio compared to the reference, expressed in percentage points (p.p.).

Tail-Duplication

Tail-duplication is the least impactful of the optimizations, both in terms of performance and cache-miss penalties, across most benchmarks.

The benchmarks getting a performance increase (or decrease, in some cases) are those having one or more branches in a hot loop that were able to be decided with static prediction.

The reason for bad performance on tail-duplication does not seem to come from instruction-cache misses since their impact is minimal. Instead, I believe it comes from wrong predictions: the prepass scheduling is activated for these measures, so a bigger superblock that got the wrong prediction can be harmful to performance. For the best benchmark on tail-duplication, the benefit of tail-duplication seems to outweigh the drawback from having more instruction-cache misses.

In some rare cases, tail-duplication induces fewer instruction-cache misses. This could be a luck effect: adding instructions changes the memory alignment of the other instructions.

At a glance, for tail-duplication, increasing the threshold looks to be beneficial, except for a couple of benchmarks.

Loop Rotation

Loop rotation has a much more significant impact. A third of the benchmarks have their performance increased by more than 5%, and some of them are in the range of +10% to +20%. However, some other benchmarks have slightly decreased performance (-5%), though they are very few. This performance decrease does not seem to be correlated with the number of instruction-cache misses: this makes sense since the duplication caused by loop rotation does not add any instruction to the loop core, only outside the loop.

Much like tail-duplication, we also have a couple of benchmarks for which loop rotation induces fewer instruction-cache misses.

Increasing the threshold does not seem to have much impact on instruction-cache misses. It should be noted that it is rare for an innermost loop to have more than 10 RTL instructions before the conditional branch, which is why increasing the threshold only improves the performance of two benchmarks.

An unfortunate drawback of loop rotation is that since the conditional branch is placed at the end of the loop, one must be taking the conditional branch (and not the fallthrough case) to loop. This incurs a penalty of 1 more cycle (compared to a jump). Thankfully, the benefits of loop rotation (improving postpass scheduling) outweigh the drawbacks for most benchmarks.

Loop Unrolling

Loop-unrolling has a significant impact on performance for most of the benchmarks (between +5% and +20%). It sometimes increases cache misses, though only rarely.

Increasing the threshold seems to be overall more beneficial, except for one benchmark.

I will detail why loop unrolling produces such an impact in subsection 10.3.6.

Loop Peeling

Loop peeling allows the CSE3 pass to perform Loop Invariant Code Motion.

Increasing the threshold seems to be more beneficial since it allows for more loops to be considered, except for a couple of cases.

This optimization seems to have the most significant impact on the instruction cache, with some benchmarks spending between 2 and 5 more p.p. on pipeline starvation. This is generally correlated with worse performance.

10.3.4 Performance Impact of each Added Optimization

In this subsection, I evaluate the performance of each optimization gradually introduced compared to mainstream CompCert 3.8, including the results from the measurements performed by Léo Gourdin and David Monniaux on the AArch64 and RiscV cores. The measures are displayed in table 10.1. Tail-duplication (with threshold 40) is included in the prepass line. For Loop Invariant Code Motion, a loop peeling of threshold 30 was used. The thresholds of loop unrolling and loop rotation were set to respectively 30 and 20. When unavailable, the measures are displayed as a "-". Each setup is evaluated with regards to the mainstream CompCert.

Setup	AArch64			KV3			RiscV		
	Q1	Med	Q3	Q1	Med	Q3	Q1	Med	Q3
+Postpass	+5.03%	+13.93%	+21.98%	+13.96%	+31.25%	+46.62%	-	-	-
+Inlining	-	-	-	+21.67%	+44.03%	+58.28%	-	-	-
+LICM	+11.49%	+29.22%	+45.73%	+27.37%	+51.47%	+74.32%	-0.92%	+1.26%	+5.83%
+Prepass	+14.03%	+32.06%	+58.01%	+28.15%	+51.53%	+75.23%	+0.57%	+9.02%	+15.72%
+Loop unroll.	+18.47%	+40.82%	+75.45%	+28.82%	+58.76%	+86.75%	+2.46%	+12.76%	+16.92%
+Loop rotate	+20.73%	+40.85%	+76.34%	+31.01%	+62.37%	+96.87%	+3.50%	+14.63%	+20.88%

Table 10.1.: Improvement of Cumulative Optimizations w.r.t. mainstream CompCert

- For both the AArch64 and KV3 cores, postpass scheduling has a significant impact. For the KV3, this impact is more prominent, as expected on a VLIW architecture.
- CompCert can inline functions; however, it is very timid at doing so, unless the function is explicitly marked as "inline". David Monniaux implemented a more aggressive inlining heuristic relying on function code size (in the number of RTL instructions) to choose to inline or not; the +Inlining I show here is with this added heuristic, at threshold 50. This was only tested on KV3. The performance boost is significant on a non-negligible number of benchmarks, from +8% up to +17%.
- LICM is another meaningful optimization, producing a gain of 15% on some benchmarks.
- Prepass scheduling[2] (without any loop unrolling) also helps, increasing performance by 5-10% for the AArch64 and RISC-V cores. This is mainly due to removing the false dependencies (compared to the postpass scheduling). The KV3 core, on the other hand, is not affected much since it features 64 user registers.
- Loop unrolling increases performance by another 5-10%.
- Loop rotation used alone has a small impact on AArch64 and RISC-V (about +3% on the latter). Still, the postpass (on AArch64) benefits from it as the rotation may provide more scheduling opportunities. It shines mostly for the KV3 architecture since it results in a more efficient bundling of the loop header in postpass.

In total, the gain of using our optimizations compared to the original CompCert is significant. On KV3, more than half of the benchmarks gain a performance boost of 60%, and a fourth of the benchmarks have a performance increase of at least a factor of 2.

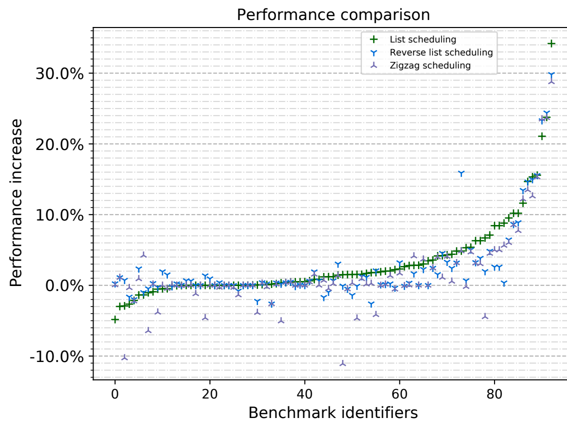
This number is a bit less significant on AArch64, though still substantial: 40% more performance for half of the benchmarks, +75% for a quarter of the benchmarks.

The results on RISC-V are less significant: +15% for half of the benchmarks, +20% for a quarter of the benchmarks. Less tuning work was done on RISC-V compared to KV3 and AArch64. The performance boost is, however, non-negligible.

[2]On the KV3, due to a temporary technical issue not related to CompCert, we disabled the support for non-trapping loads.

10.3.5 Evaluation of the Different Prepass-Scheduling Strategies

For prepass scheduling, we have three different strategies: forward list scheduling, reverse list scheduling, and zigzag scheduling. I first evaluate these 3 different strategies in figure 10.8. For this experiment, all optimizations are activated.



Setup	KV3 Q1	KV3 Med	KV3 Q3
List Scheduling	+0.00%	+1.22%	+4.18%
Reverse List Scheduling	+0.00%	+0.39%	+2.38%
Zigzag Scheduling	-0.18%	+0.08%	+2.74%

Figure 10.8.: Performance gain of prepass scheduling on KV3. The left figure shows each benchmark individually. The right figure shows statistics across all the benchmarks.

From the results, no scheduling is absolutely superior: some benchmarks prefer zigzag scheduling, while some others prefer reverse list scheduling, and yet other benchmarks prefer list scheduling. However, for most of the benchmarks we have tested, (forward) list scheduling seems to be the best choice, probably because the KV3 has 64 user registers available, so the chance of suffering from register spilling during register allocation is slim.

In general, the gain of using prepass scheduling on the KV3 is relatively low for 2/3 of the benchmarks, though it can be exceptional for some benchmarks. Indeed, there are occasions where the postpass scheduler cannot find a good schedule because of register dependencies introduced by the register allocator — prepass scheduling helps a lot in those cases.

The performance of zigzag scheduling is non-negligibly detrimental for approximately 10 benchmarks out of 90.

10.3.6 Gains of Loop Unrolling without Prepass Scheduling

One oddity that struck me compared to what I was initially expecting is that loop unrolling seems to gain a lot of performance across many benchmarks on the KV3, regardless of the presence of prepass scheduling. To illustrate that, let us deactivate prepass scheduling and focus only on the

absence or not of loop unrolling. All the other optimizations are activated. The result is shown figure 10.9.

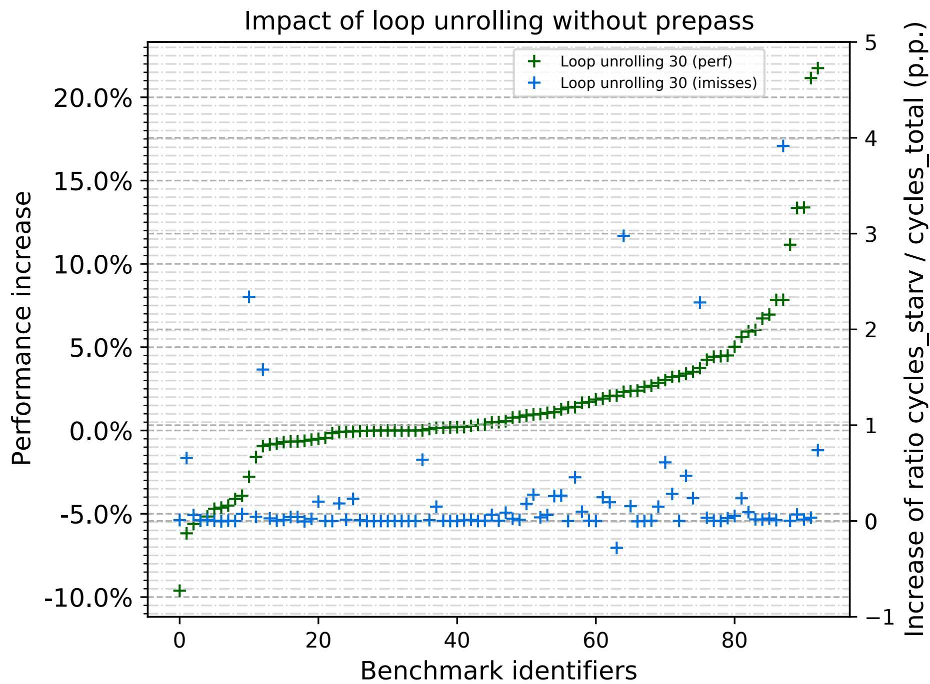


Figure 10.9.: Impact of loop unrolling, without prepass, with a threshold of 30.

The performance gain of 20% was particularly impressive and unexpected. Let us dive down the rabbit hole to figure out what happened. This gain corresponds to the benchmark *matrix1* from TACLeBench, whose code is shown below.

```
1  void matrix1_main( void ) {
2    register int *p_a = &matrix1_A[ 0 ];
3    register int *p_b = &matrix1_B[ 0 ];
4    register int *p_c = &matrix1_C[ 0 ];
5
6    register int f, i, k;
7
8    for ( k = 0; k < Z; k++ ) {
9      p_a = &matrix1_A[ 0 ]; /* point to the beginning of array A */
10
11     for ( i = 0; i < X; i++ ) {
12       p_b = &matrix1_B[ k * Y ]; /* take next column */
13
14       *p_c = 0;
15       for ( f = 0; f < Y; f++ ) /* do multiply */
16         *p_c += *p_a++ * *p_b++;
```

```
17
18          p_c++; } } }
```

On that code, most of the execution time is spent in the innermost loop. It is a very short innermost loop, so its code is relatively easy to analyze. I give in figure 10.10 two versions of the assembly code implementing this innermost loop: one without loop unrolling and one with loop unrolling.

The first version spends a total of 10 cycles per iteration: 5 cycles for issuing the bundles (5 bundles), 2 for the branch penalty (the conditional branch at the end), 2 for the stall on `$r39`, and 1 cycle for the stall on `$r1` (2 cycles are required for the `maddw` instruction).

The second version spends a total of 17 cycles for two iterations: 9 cycles for the bundles (9 bundles), 2 for the branch penalty (last conditional branch), and 6 cycles for stalls (the same stalls as above, but duplicated). On average, the second version spends 8.5 cycles per iteration, hence a 15% performance boost.

So, where does this performance boost come from, and how did loop unrolling affect it? First of all, on the second version, we have one fewer instruction: the `lws $r45` that was line 11 does not appear anywhere in the second iteration. This was taken off by the base CSE optimization from CompCert,[3] thus saving us half a cycle per iteration. Finally, the biggest gain comes from the absorption of the conditional-branch penalty. There is no branch penalty in one iteration out of two because the next iteration is laid out right next to the conditional branch (see line 16). This makes us gain two cycles per couple of iterations, so one cycle per iteration on average.

This case is particularly fruitful because the loop was rotated, so the branch penalty was 2 cycles. However, even if the loop were not rotated, there would still be a penalty of 1 cycle (penalty of unconditional jump): loop unrolling would have saved half a cycle per iteration (instead of 1).

In conclusion, loop unrolling alone can have a significant impact on small loops, regardless of any other optimization. CSE and prepass scheduling can help further on top of it.

10.4 CompCert vs GCC in our Current Iteration

In this section, I evaluate our version of CompCert with all enabled optimizations versus GCC, for KV3 (subsection 10.4.1), and for AArch64 and RISC-V (subsection 10.4.2).

[3] CSE3 is also able to do this substitution, but CSE is executed before CSE3.

```
1  .L106:
2    addd  $r3 = $r2, 0
3    addd  $r8 = $r7, 0
4    lws $r1 = 0[$r35]
5    addw  $r10 = $r10, 1
6  ;;
7    addd  $r2 = $r3, 4
8    addd  $r7 = $r8, 4
9    lws $r41 = 0[$r3]
10   compw.lt  $r32 = $r10, 10
11 ;;
12   lws $r39 = 0[$r8]
13 ;;
14   maddw $r1 = $r41, $r39
15 ;;
16   sw  0[$r35] = $r1
17   cb.wnez $r32? .L106
18 ;;
```

```
1  .L110:
2    addd  $r3 = $r2, 0
3    addd  $r9 = $r6, 4
4    lws $r5 = 0[$r43]
5    addw  $r37 = $r42, 1
6  ;;
7    addd  $r2 = $r3, 4
8    lws $r10 = 0[$r3]
9    compw.ge  $r32 = $r37, 10
10 ;;
11   lws $r45 = 0[$r6]
12 ;;
13   maddw $r5 = $r10, $r45
14 ;;
15   sw  0[$r43] = $r5
16   cb.wnez $r32? .L111
17 ;;
18   addd  $r8 = $r2, 0
19   addd  $r6 = $r9, 4
20   lws $r38 = 0[$r9]
21   addw  $r42 = $r37, 1
22 ;;
23   addd  $r2 = $r8, 4
24   lws $r15 = 0[$r8]
25   compw.lt  $r32 = $r42, 10
26 ;;
27   maddw $r5 = $r15, $r38
28 ;;
29   sw  0[$r43] = $r5
30   cb.wnez $r32? .L110
31 ;;
```

Figure 10.10.: Two versions of the innermost loop from *matrix1*: non-unrolled on the left, unrolled on the right. See appendix A for an instruction reference.

10.4.1 CompCert vs GCC on the KV3 Architecture

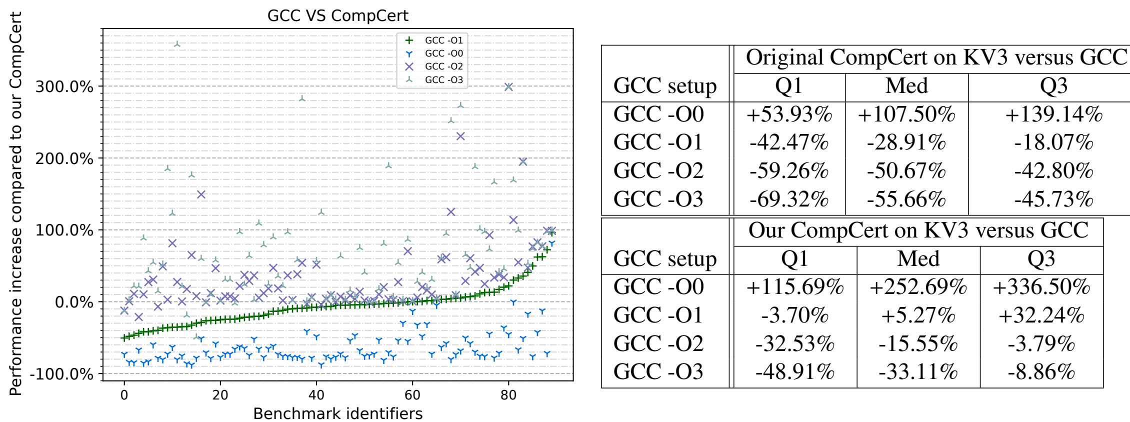
Figure 10.11 shows GCC with four different optimization levels, compared to CompCert.

First of all, for all the benchmarks we have run, CompCert is always better than GCC -O0: for safety-critical applications that require traceability (e.g. DO178 level-A avionics), CompCert is a better choice in terms of performance (and the preservation proof of CompCert makes it also possible to have traceability, see Bedin França et al. [8]). On the KV3, GCC -O0 delivers particularly inefficient code because it does not use the 64 available registers (retrieving stack values through load/store rather than assigning registers to them), and postpass scheduling is deactivated: bundles contain only single instructions. As a result, using CompCert instead of GCC -O0 results in 250% more performance for half of the benchmarks.

CompCert is also slightly better than GCC -O1 for the majority of the benchmarks, thanks to our additional introduced optimizations. Also, GCC -O1 does not perform any bundling nor any reordering. However, in some cases, we still have poorer performance than GCC, even in -O1.

Compared to GCC -O2, on the KV3 architecture, we still have room for improvement, although there are some benchmarks where we are very close: in a fourth of the benchmarks, we are only 4% worse than GCC -O2. Also, GCC -O3 beats us by a large margin.

I will explain further below some elements that are missing either in CompCert or, more generally, in our KV3 port.



GCC setup	Original CompCert on KV3 versus GCC		
	Q1	Med	Q3
GCC -O0	+53.93%	+107.50%	+139.14%
GCC -O1	-42.47%	-28.91%	-18.07%
GCC -O2	-59.26%	-50.67%	-42.80%
GCC -O3	-69.32%	-55.66%	-45.73%

GCC setup	Our CompCert on KV3 versus GCC		
	Q1	Med	Q3
GCC -O0	+115.69%	+252.69%	+336.50%
GCC -O1	-3.70%	+5.27%	+32.24%
GCC -O2	-32.53%	-15.55%	-3.79%
GCC -O3	-48.91%	-33.11%	-8.86%

Figure 10.11.: Comparing the performance of CompCert and GCC on KV3. The left figure shows GCC compared relatively to CompCert across all the benchmarks. The right figure shows the reverse: the gain of using CompCert instead of GCC, for the original version of CompCert and the version with our included optimizations.

Elements Missing in CompCert Compared to GCC -O1

To get a sense of the missing elements, I picked the worst benchmark for CompCert vs GCC -O1: this is *rad2deg* from TACLeBench. The core of the source code is straightforward:

```
1  #define rad2deg(r) ((r)*180/PI)
2  float rad2deg_X, rad2deg_Y;
3  void rad2deg_main( void )
4  {
5    for ( rad2deg_X = 0.0f; rad2deg_X <= ( 2 * PI + 1e-6f ); rad2deg_X += (
         PI / 180 ) )
6      rad2deg_Y += rad2deg( rad2deg_X );
7  }
```

It features floating-point multiplications and divisions primarily. The KV3 does not have any floating-point division, so both GCC and CompCert have to rely on an external function provided

as a library for performing the floating-point division. I give in figure 10.12 the code generated for GCC and CompCert. I turned off loop unrolling to make it easier to analyze – it decreases a bit the performance, but this is negligible compared to the issues I will outline. `__divsf3` is the library function providing floating-point division for 32-bits floats.

First of all, one may notice that in the CompCert code, two calls to `__divsf3` are done, compared to just one for GCC. This is because we have no constant propagation for division with floats in our port, so CompCert has to recompute the constant `(PI/180)` every time.[4] Since this is done through a call to an external function, this is not caught by the Loop Invariant Code Motion of D. Monniaux [63].

Another point: the handling of global variables. GCC was smart enough to figure that during the loop execution, the global variable `rad2deg_Y` would only be accessed by the loop; all functions called in that particular loop will not access or modify `rad2deg_Y`. As a result, the global variable is loaded from the heap before the loop, then stored after the loop. In CompCert, the global variable value is stored or reloaded after each access, resulting in extra load and store instructions. Although the CSE pass is able to eliminate redundant load and stores, it does not traverse loops, and also the presence of a builtin can compromise the optimization (since builtins might modify the program memory). This, in turn, introduces extra stalls that were not present in the original code: in total, 6 additional cycles of stalls are introduced (from the two `faddw` instructions that each require 4 cycles to complete).

Finally, on some particular benchmarks, we noticed that GCC can expand an aggregate (such as an array or a structure) into independent scalar variables, which can then be allocated to registers. In other words, the values are first loaded from the aggregate to the registers; then, the computation is done with the allocated registers; finally, the values are stored back to the aggregate. On the affected benchmarks, the performance difference can be significant since doing so removes store and load instructions from the main loop.

There are some subtler differences, such as GCC having simpler prologues and epilogues for leaf functions that use no stack variables or GCC having better instruction selection in some instances.

Additional Elements of GCC -O2 Compared to us

There are several things that GCC -O2 does more or better than us, in addition to the relevant points from GCC -O1:

[4] We do not have any constant propagation for float division because the division is inserted as a builtin before constant propagation occurs. It might be possible to solve this technical issue in the future with the improved builtin support of CompCert 3.6.

```
 1    make $r0 = rad2deg_Y
 2    ;;
 3    lwz $r19 = 0[$r0]
 4    ;;
 5    make $r18 = 360
 6    ;;
 7    /* ... */
 8  .L4:
 9    copyw $r1 = $r21
10    ;;
11    fmulw $r0 = $r14, $r22
12    ;;
13    call __divsf3
14    ;;
15    faddw $r19 = $r19, $r0
16    ;;
17    faddw $r14 = $r14, $r20
18    ;;
19    addw $r18 = $r18, -1
20    ;;
21    cb.dnez $r18? .L4
22    /* ... */
23    ;;
24    make $r0 = rad2deg_Y
25    ;;
26    sw 0[$r0] = $r19
```

```
 1    make  $r20 = rad2deg_Y
 2    make  $r21 = rad2deg_X
 3    /* ... */
 4  .L102:
 5    lws $r23 = 0[$r20]
 6    fmulw $r0 = $r0, $r18
 7    addd  $r1 = $r19, 0
 8    call  __divsf3
 9  ;;
10    faddw $r3 = $r23, $r0
11    addd  $r1 = $r18, 0
12    addd  $r0 = $r19, 0
13  ;;
14    sw  0[$r20] = $r3
15  ;;
16    lws $r23 = 0[$r21]
17    call  __divsf3
18  ;;
19    faddw $r0 = $r23, $r0
20  ;;
21    sw  0[$r21] = $r0
22  ;;
23    fcompw.oge  $r32 = $r22, $r0
24  ;;
25    cb.wnez $r32? .L102
26  ;;
```

Figure 10.12.: Compiled codes of `rad2deg_main`: on the left, the code from GCC -O1; on the right, the code from CompCert without loop unrolling. See appendix A for an instruction reference.

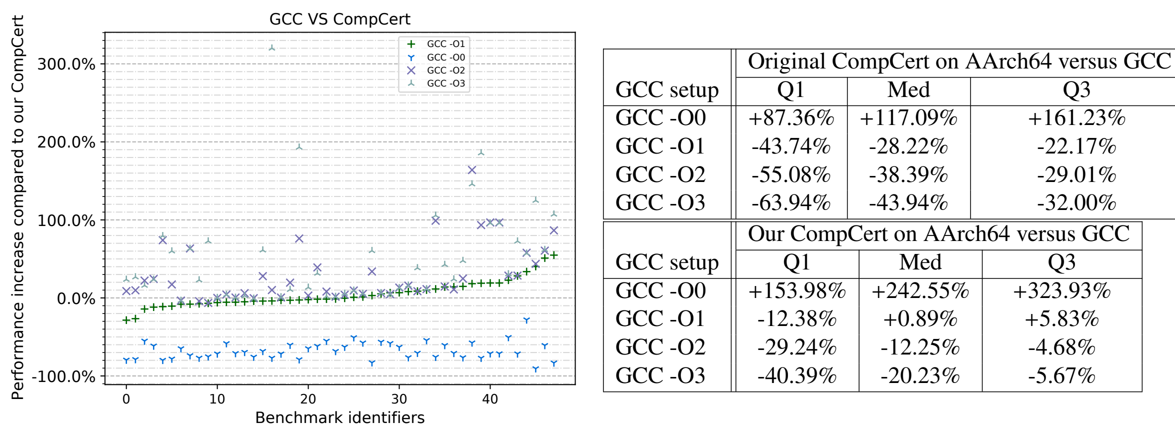
- GCC -O2 is able when the number of iterations is known at compile time to unroll completely a given loop (if the number of iterations is small), which removes any conditional branching and allows to reorder the whole unrolled loop body. In certain cases, the performance gained is quite significant: at least one cycle is gained from removing the loop branch, and a couple more cycles per iteration can be gained with the finer scheduling that follows.
- GCC has memory alias analysis: it can reorder stores and loads when it can rule out that they will never cross each other (when their accesses never overlap). On the other hand, we do not use any alias analysis yet; memory is seen as a single resource, invalidated by any store. An alias analysis already exist in RTL, though we have not looked yet into integarting it in our prepass scheduler. Propagating this alias analysis to Asm could prove challenging. Alternatively, a more limited form could be added in the postpass scheduler to reorder memory stores from the same base register but with different offsets; however, the verifier would need to be enhanced for this.

- The KV3 architecture features hardware loops: it is possible to set up special registers pointing to the beginning and end of a loop and a counter register; then the given section is repeated as many times as specified by the register counter.
- GCC is generally more aggressive than us when it comes to unrolling loops: so far, we only unroll loops once, but we could probably try to unroll them several times. In general, there is room for improvement in our current heuristics.
- GCC can perform loop strength reduction: instead of incrementing the loop index by 1, then multiplying by a loop-invariant constant c, it increments the loop index by c directly. This saves one dependency and a multiplication, usually resulting in at least one gained cycle.

Bringing CompCert up to speed with GCC -O2 would require implementing the above optimizations.

10.4.2 CompCert vs GCC on the AArch64 and RISC-V Architectures

We also evaluated CompCert with regards to AArch64 and RISC-V with our new optimizations. The results can be found in figures 10.13 and 10.14.

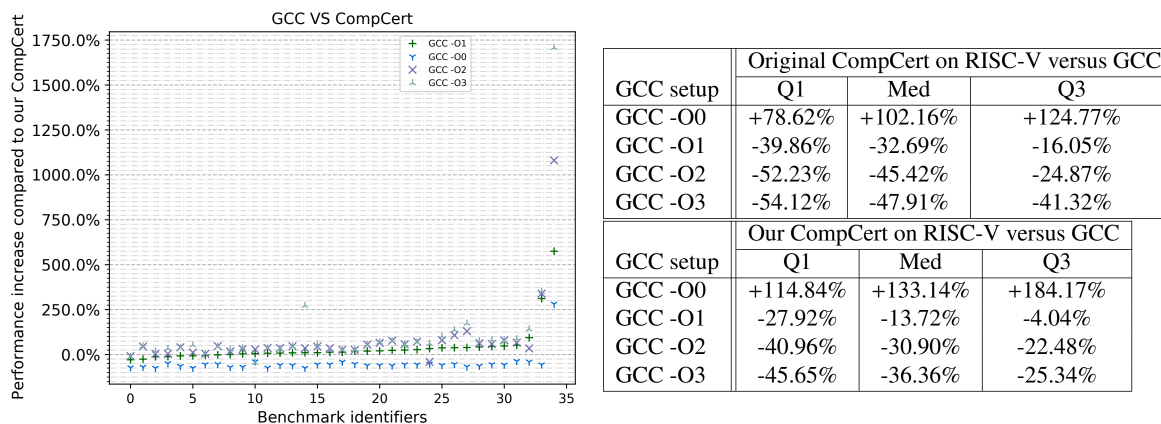


GCC setup	Original CompCert on AArch64 versus GCC		
	Q1	Med	Q3
GCC -O0	+87.36%	+117.09%	+161.23%
GCC -O1	-43.74%	-28.22%	-22.17%
GCC -O2	-55.08%	-38.39%	-29.01%
GCC -O3	-63.94%	-43.94%	-32.00%

GCC setup	Our CompCert on AArch64 versus GCC		
	Q1	Med	Q3
GCC -O0	+153.98%	+242.55%	+323.93%
GCC -O1	-12.38%	+0.89%	+5.83%
GCC -O2	-29.24%	-12.25%	-4.68%
GCC -O3	-40.39%	-20.23%	-5.67%

Figure 10.13.: Comparing the performance of CompCert and GCC on AArch64. The left figure shows GCC compared relatively to CompCert across all the benchmarks. The right figure shows the reverse: the gain of using CompCert instead of GCC, for the original version of CompCert and the version with our included optimizations.

For AArch64, the results are less pronounced than for the KV3; however, the same trends as for KV3 can be observed. While GCC -O1 is for most benchmarks faster than our version of CompCert, our optimizations bring CompCert closer to GCC -O1.



GCC setup	Original CompCert on RISC-V versus GCC		
	Q1	Med	Q3
GCC -O0	+78.62%	+102.16%	+124.77%
GCC -O1	-39.86%	-32.69%	-16.05%
GCC -O2	-52.23%	-45.42%	-24.87%
GCC -O3	-54.12%	-47.91%	-41.32%

GCC setup	Our CompCert on RISC-V versus GCC		
	Q1	Med	Q3
GCC -O0	+114.84%	+133.14%	+184.17%
GCC -O1	-27.92%	-13.72%	-4.04%
GCC -O2	-40.96%	-30.90%	-22.48%
GCC -O3	-45.65%	-36.36%	-25.34%

Figure 10.14.: Comparing the performance of CompCert and GCC on RISC-V. The left figure shows GCC compared relatively to CompCert across all the benchmarks. The right figure shows the reverse: the gain of using CompCert instead of GCC, for the original version of CompCert and the version with our included optimizations.

Results for RISC-V are not as good as on AArch64, though still noteworthy compared to the original CompCert. For one benchmark in particular (*audiobeam* from TACLeBench), CompCert produces particularly inefficient code: CompCert is almost 3 times slower than GCC -O0 for that benchmark. Excluding that particular benchmark, the lack of postpass scheduling is likely to be a reason for the worse performance. Also, less work went into the RISC-V prepass backend compared to AArch64 and KV3; this remains a work in progress.

11 Conclusion

11.1 Summary

We introduced a new Asm language, AsmVLIW, modelling the execution of bundles in CompCert. We also recovered the basic-block structure, introducing two new IRs: Machblock and Asmblock, which are identical to Mach and Asm, except that the code is organized in basic blocks, run in one semantic step, which we call block-step. The difference between Asmblock and AsmVLIW lies only in the execution semantics: basic blocks have sequential execution semantics, while AsmVLIW bundles have parallel execution semantics.

We introduced a postpass scheduler on basic blocks to generate bundles, formally verified by translation validation with hash-consing. The scheduler and verifier are independent in our approach: the scheduler is implemented through an untrusted oracle (in OCaml), while the verifier is implemented in Coq and is formally proved. To tackle the considerable number of instructions in the KV3 backend, we used a new IR, AbstractBasicBlock. It abstracts the details of each instruction to focus on which resources are being read or written and which operations are performed. The AbstractBasicBlock IR is equipped with two semantics (sequential and parallel) and symbolic evaluation. By writing a translation between Asmblock and AbstractBasicBlock, and by proving a bisimulation between the two IRs, we were able to prove the postpass scheduler verifier in two steps: first, checking that the concatenation of the bundles (returned by the scheduler oracle) is equivalent to the original basic block; then checking that each bundle has the same sequential and parallel semantics.

We then explored a prepass scheduler, at the RTL level, on superblocks. Forming superblocks requires having knowledge of the execution-flow tendencies. We implemented static branch prediction with profiling and user-annotation support (via the builtin `__builtin_expect`) and modified RTL to include such information. Before superblock scheduling, we implemented tail-duplication to transform relevant traces into superblocks. Tail-duplication is formally verified through a generic and lightweight checker (`Duplicate`) that specializes in transformations that duplicate instructions without changing the semantics.

We introduced a new IR, RTLpath, with block-step semantics executing whole superblocks. This IR is just RTL augmented with block-step semantics and with a “pathmap” giving the partitioning of the control-flow graph into superblocks. In particular, RTL and RTLpath bisimulate each

other. Within this IR, we integrated a liveness checker and a new simulation modulo liveness. An external oracle creates the superblocks (and their associated liveness information); the RTLpath checker then checks the liveness information. We adapted the linearizing heuristics of CompCert to ensure our superblocks are laid out contiguously in memory.

We used this RTLpath to construct a superblock scheduling checker. For that, we introduced an extension of symbolic execution for superblocks and a refined symbolic execution with hash-consed terms. Through adequate properties of bisimulation between RTLpath and superblock symbolic execution, and the simulation modulo liveness of RTLpath, we were able to formally verify the checker.

We also used the `Duplicate` verifier for transformations that further increases performance: loop unrolling, loop rotation, loop peeling. In particular, loop peeling was used in conjunction with a new common-subexpression pass of D. Monniaux to achieve loop-invariant code motion; this was published in Monniaux and Six [63].

Finally, we evaluated each of our optimizations on a set of benchmarks, including Polybench and TACLeBench. Our optimizations give a significant performance increase to CompCert: on the KV3 VLIW core, our optimizations increase the performance of most benchmarks by 62%.

Although our version of CompCert is beating GCC -O1 for most benchmarks, we are still 12% worse than GCC -O2. We made a detailed analysis in section 10.4 and gave starting points for future work.

11.2 Reflexions and Future Work

The goal of this thesis was to bring CompCert closer to modern compilers such as GCC in terms of the performance of the generated code. This thesis was focused on the KV3 processor, a VLIW architecture with an interlocked pipeline. However, most of our optimizations can be used for other targets as well:

- Loop unrolling, loop rotation and loop-invariant code motion can be directly plugged in the other backends. Prepass superblock scheduling can also be plugged, though some backend-specific code needs to be written for encoding the resources and latencies of each Asm instruction.
- Our postpass scheduling can be adapted to other architectures: it was recently ported to AArch64 after three person-months of development [89].

In particular, we chose an approach where most optimizations are performed in smaller and easier to prove passes:

- Instead of proving loop-invariant code motion directly, we split it into two passes: a common-subexpression pass that goes through control-flow merges and a code duplication pass that replicates an entire loop body.
- Tail-duplication, superblock selection and superblock scheduling are three independent passes; compared to the heavier trace scheduling, this allowed us to ease the proof effort.
- Postpass scheduling is split into the generation of basic blocks at Mach level, translation of basic blocks from Machblock to Asmblock, and finally AsmVLIW bundles generation from Asmblock.

This approach was fruitful in our development and allowed us to tackle optimizations that would have probably been tougher to prove otherwise.

We also showed in this thesis that the concept of *a posteriori verification*, initially studied within CompCert by Tristan and Leroy [95], can be coupled with hash-consing to provide reasonable compilation times and can be used for a wide range of optimizations providing a significant increase in performance. On the KV3 processor, these optimizations bring CompCert close to GCC -O2. There is still progress to be done; we gave in section 10.4 a non-exhaustive list of the reasons why CompCert remains behind GCC.

Among the possible directions for progression, the following points could be studied:

- We unroll each loop only once; it could be interesting to try unrolling a given loop several times. Ideally, the code size of a loop (in bytes) should never be larger than the instruction cache. Davidson and Jinturkar [24] points that the loop unrolling pass should be positioned after traditional optimizations that modify the code size to have a fair approximation of the size of the unrolled loop.
- We do not perform any register renaming when unrolling the loop: this leads to false dependencies in scheduling and could be one reason why our prepass scheduling does not gain much from unrolling. A register renaming transformation (using fresh registers) could be introduced by an oracle and verified *a posteriori* by symbolic superblock execution (with a verification modulo liveness).
- It could be possible to implement induction variable expansion, as described in Hwu et al. [40]. In terms of verification, this would require adding more rewriting rules (e.g. for the addition operation) in superblock symbolic evaluation semantics. Loop strength reduction could probably be supported as well.
- None of our schedulers support alias analysis. At the RTL level, we could probably reuse the results of the alias analysis pass from Robert and Leroy [82]. It would probably be more challenging to forward these results to the Asm IR for the postpass scheduler.
- Our prepass scheduling does not pay much attention to the register pressure: while backward list scheduling tends to ease the register pressure compared to forward list scheduling, there is no guarantee that the pressure never exceeds a given threshold.

- It could be interesting to support hardware loops in CompCert, though it would be challenging: the exact number of iterations would need to be inferred down the intermediate representations. Also, one would need to prove some invariant that the special loop registers will never be modified outside of the given loop.

The above lays the ground for potential future work to bring CompCert even closer to production compilers in terms of the performance of the generated code.

A

Non-exhaustive Instruction Listing of the KV3 Core

The KV3 processor features 64-bit registers that can be used for storing either floating-point or integer values.

I write below the instructions that were mentioned in the manuscript, with their assembly semantics. I use registers for most operands, however it is possible most of the time to use immediate values instead. This usually results in a bigger instruction depending on the size of the immediate: it will take more slots in the VLIW bundle.

- "`addw $r0 = $r1, $r2`" performs `$r1 + $r2`, then clears the 32 upper bits and stores the result in `$r0`.
- "`addd $r0 = $r1, $r2`" performs `$r1 + $r2`, and stores the result in `$r0`.
- "`call symbol`" performs a call to `symbol`. In practice, this is done by copying the current value of PC (Program Counter) into RA (Return Address), then setting PC to the address of `symbol`.
- "`cb.dnez $r0? .L1`" evaluates a unary condition on `$r0`, based on the modifier. If the comparison succeeds, PC jumps to the `.L1` label. In this example, the modifier is `.dnez` which stands for "Double word Not Equal to Zero": it triggers the branch if `$r0` is not equal to 0. On the KV3, the branch penalty is 2 cycles if the branch is taken, 0 cycle if the branch is not taken.
- "`compw.ge $r0 = $r1, $r2`" evaluates a binary condition on the 32 lower bits of registers `$r1` and `$r2`, then stores the boolean result in `$r0`. The modifier expresses the condition: here, `.ge` stands for "Greater Than or Equal".
- "`copyw $r0 = $r1`" copies the 32 lower bits of `$r1` into `$r0`. The 32 higher bits of `$r0` are cleared.
- "`faddw $r0 = $r1, $r2`" stores in `$r0` the addition of `$r1` and `$r2`, interpreted as 32-bits floating-point values. The result is rounded to a 32-bits floating-point value according to a rounding mode set in the `CS` register. It is also possible to override the rounding mode via a modifier.
- "`fcompw.oge $r0 = $r1, $r2`" is the same as `compw`, but for floating-point values. It also features different modifiers since comparing two floating-point values is more

complex than comparing two integers values. In this example, `.oge` stands for "Ordered and Greater Than or Equal".

- "`fmulw $r0 = $r1, $r2`" is the same as `faddw`, but for multiplication.
- "`goto .L1`" makes PC jump to the `.L1` label.
- "`lws $r0 = $r1[$r2]`" loads the 32-bits memory value at address `$r1 + $r2` into `$r0`, and performs sign extension to the result. There are also variants for loading 64-bits, or even 128-bits and 256-bits. There are modifiers to add scaling to the offset: for instance, `lws.xs` loads the value of `4 * $r1 + $r2`. This is extremely helpful for array accesses. The optional `.s` modifier turns the instruction into a speculative load, i.e., one that does not trap if an incorrect address is given.
- "`maddw $r0 = $r1, $r2`" performs a multiply-add on the least significant words: `$r0 += $r1 * $r2`. The resulting word is zero-extended.
- "`make $r0 = imm`" sets `$r0` to the immediate value `imm` (up to 64 bits).
- "`sxwd $r0 = $r1`" performs sign-extension on `$r1` (from 32-bits to 64-bits) and stores the result in `$r0`.
- "`sw $r0[$r1] = $r2`" stores the 32 lower bits of `$r2` into the memory address `$r0 + $r1`. It also has the `.xs` modifier. However, it does not have the `.s`: memory stores are never speculative on the KV3 architecture.

List of Figures

1.1 The Coolidge architecture . . . 9

1.2 The pipeline stalls at cycles 5 and 6 because B3 is waiting for the results of R_1 and R_2 from bundles B1 and B2, which are completed at stage E3. Stage PF (not shown here) happens just before the ID stage. . . . 11

1.3 The Intermediate Languages of CompCert . . . 11

1.4 Examples of Simulation Diagrams in CompCert, from Leroy [51] . . . 13

1.5 CompCert compilation flow with our added optimizations in red . . . 17

3.1 Architecture of our postpass scheduling . . . 28

3.2 Parallel In-Order Blockstep . . . 33

3.3 Execution Steps between Mach States . . . 36

3.4 Overview of our Proof for Condition (2) of the “Block” Simulation of Mach by Machblock . . . 39

3.5 Simulation Diagrams with Stuttering in CompCert . . . 40

3.6 Diagram of the Simulation Proof . . . 49

3.7 Certified Scheduling from Untrusted Oracles . . . 52

3.8 Diagram of `bblock_simub` Correctness . . . 56

4.1 Two nested loops . . . 64

4.2 Program with two branches. Top figure: original program. Bottom figure: the instruction E_1 is moved before A (under the non-trivial assumption that it is correct to do so). An identified trace (that could be turned into a superblock) is shown in green. Each conditional branch edge is annotated with its probability. . . . 66

4.3 . . . 70

4.4 On the left, the assembly generated code, annotated with timing information. On the right, its basic-block representation, with a trace of interest shown in green. In the assembly code, the registers targeted by load instructions along the trace are written in red. See appendix A for an instruction reference. . . . 72

4.5 Tail-duplicating the identified trace. . . . 74

4.6 Performing loop unrolling one time. . . . 75

4.7 Final code after tail duplication and loop unrolling. . . . 76

4.8 Two scheduling examples — the moved instructions are in red. The left schedule is incorrect: an instruction assigning `x4` was moved before a branch which later uses `x4`. The right schedule is correct. 78

4.9 Example illustrating the impact of register allocation on scheduling. Left: RTL code example; it is possible to move instruction 4 before instruction 3 to reduce stalls. Right: Assembly (before scheduling) example; that move is not possible because of a false dependency with `$r2`. 79

4.10 Our example, with prepass scheduling activated. It is annotated with timing information. 81

4.11 Our order of RTL optimizations 84

5.1 C code, and RTL representation of `fold_add` as a control-flow graph 86

5.2 The RTL code of `fold_add`, represented syntactically. The instruction nodes are represented in red. The entrypoint of that code is 9. For clarity, I represent registers as "x" followed by their numbers – though in the actual syntax, registers are just positive integers. 87

5.3 Execution steps between RTL states 89

5.4 Three examples of pathmaps for the RTL code on the left. We omit the `output_regs` field. Pathmap A would lead to an incorrect execution. Pathmap B is correct but undesirable because the path starting at 9 is not a superblock. Pathmap C is correct and desirable. 96

5.5 Executing a path from s (an `RTLpath.State`). Two possible executions are shown. One executes an entire path by computing `istep psize` times, finishing by a `path_last_step` which gives the final state s'_1. The other computes `istep` only n times, then reaches a point where `icontinue = false`: it exits the path and returns a `State` s'_2. The entire execution is a `path_step`. 99

5.6 Bridging RTL and RTLpath execution. The left figure shows an execution through a whole path: the right figure shows an execution with the path ending before the end because of an early exit. The red transition outlines the RTL execution of the last instruction from the path; the olive transitions outline RTL executions corresponding to RTLpath stutterings. Dotted edges correspond to `match_inst_states_goal idx` with each `idx` value displayed on top of an edge. 102

6.1 Example of code duplication, with its reverse mapping. To the left, the original code. To the right, the transformed code. The reverse mapping is shown in red. . . 112

7.1 Example of symbolic execution, then evaluation of the symbolic states 120

7.2 Tree representation of a `sstate` 122

7.3 Graphical representation of `all_fallthrough_upto_exit` relation. The superblock is executed from right to left; all conditions up to `ext` have been evaluated to `false` (`f` in the drawing). The other conditions remain to be evaluated. 125

7.4 Summary of `ssem_internal` semantics. The same superblock is presented in two scenarios: on the left, no early exit was triggered so far, so we continue to execute the superblock. On the right, one early exit was encountered: the next internal state will then have `icontinue` as false. The symbols $\underset{C}{\Longrightarrow}$ and $\underset{L}{\Longrightarrow}$ respectively denotate the condition and local (registers and memory) evaluation functions (over symbolic values). 126

7.5 Evaluating a symbolic state with the `ssem` relation. 129

7.6 Graphical representation of `siexec_path_correct` lemma. The superblock is symbolically executed up to a certain point, giving an internal symbolic state st. The evaluation of this internal symbolic state succeeds and gives an internal state is. The lemma proves then a correspondence between the remaining symbolic execution via `siexec_path`, and `isteps`. 133

7.7 Two semantically equivalent superblocks (in green) but who reach different states when taking the branch A. 135

7.8 Diagram used for proving the simulation in superblock scheduling. s_1 is actually the state made of $(f, sp, stk_1, rs, m, pc_1)$, and s_2 is made of $(f', sp, stk_2, rs, m, pc_2)$. $\underline{b_i}$ is the symbolic execution of the superblock from pc_i. 141

7.9 Diagram used for proving the refinement simulation. 155

8.1 RTL code for two nested loops. The identified trace is shown in green. 162

8.2 RTL code for two nested loops, with a tail-duplication starting at B. The identified trace shown in green is now a superblock, although unwanted. The duplicated nodes as a result of tail-duplication are shown in blue. 163

8.3 RTL code for two nested loops, with an interesting tail-duplication done on the green trace. The duplicated node is in blue. 163

8.4 Example of loop unrolling and loop peeling: above is the original code, in the middle is the unrolled code, at the bottom is the code with loop peeling. The nodes that are part of the original superblock are shown in green. The duplicated nodes are shown in blue. 168

8.5 Example of loop rotation: above is the original code, below is the rotated code, with the duplicated nodes shown in blue. 169

8.6 Two possible code layouts for the LTL code shown on the left. Each code layout is annotated with branching penalties from two possible executions: either through B_3 (green) or through B_2 (blue). On average, the layout on the right has lower branching penalties than the layout on the left. 174

8.7 Two possible code layouts for the LTL looping code shown on the left. The left layout is from the original CompCert heuristics, while the right layout is from a hypothetical heuristics where the chains are laid out in increasing order. 177

9.1 Example of dependency graph. The Read-After-Write dependencies are in black; the Write-After-Read are in green; the Write-After-Write are in red. The dashed edges represent dependencies of latency 0. 186
9.2 Our code architecture for implementing the schedulers. The architecture-dependent parts are in red, the architecture-independent parts are in green. 190
9.3 Example of dependency graph of a superblock, for architectures that do not have speculative loads. The usual RAW dependencies are in black, the extra conditional-branch dependencies coming from trapping instructions are in red, the additional dependencies between the conditional branches are in green. 191
9.4 Our three possible prepass schedules on the previous example. The code is annotated with the initial position of each line of code for clarity. I also show the "time $\mapsto$ instructions" mapping of each schedule. The conditional branches are outlined in red. The load instructions are marked in green, and the conditional branches in red, for easier recognition. 193

10.1 Performance gain of postpass scheduling and “greedy bundling” on KV3. The left figure shows each of our handpicked benchmarks. The right figure shows the results across all the benchmarks. 200
10.2 Performance gain of postpass scheduling on AArch64. The left figure shows each of our handpicked benchmarks. The right figure shows the results across all the benchmarks. 201
10.3 On the left, compilation times based on the basic-block size for all our benchmarks (except TACLeBench, which was added recently) in logarithmic scales. On the right, the timings of compiling *bitsliced-aes* (showing only the passes with non-negligible computational time). 201
10.4 On the left, statistics of the static branch prediction, evaluated in regards to profiling. On the right, details for each heuristic. 203
10.5 Gain of using profiling instead of branch prediction. On the left, we evaluate this gain on the Polybench benchmarks. On the right, we evaluate it on the Taclebench benchmarks. 204
10.6 Gain of using superblock linearization instead of base linearization. On the left, the setup includes just postpass scheduling and static branch prediction. On the right, it includes the best version of our optimizations. 205
10.7 Impact of the four instruction-duplication optimizations on performance and pipeline stalls from instruction-cache misses. 206

10.8 Performance gain of prepass scheduling on KV3. The left figure shows each benchmark individually. The right figure shows statistics across all the benchmarks. 210
10.9 Impact of loop unrolling, without prepass, with a threshold of 30. 211
10.10 Two versions of the innermost loop from *matrix1*: non-unrolled on the left, unrolled on the right. See appendix A for an instruction reference. 213
10.11 Comparing the performance of CompCert and GCC on KV3. The left figure shows GCC compared relatively to CompCert across all the benchmarks. The right figure shows the reverse: the gain of using CompCert instead of GCC, for the original version of CompCert and the version with our included optimizations. 214
10.12 Compiled codes of `rad2deg_main`: on the left, the code from GCC -O1; on the right, the code from CompCert without loop unrolling. See appendix A for an instruction reference. 216
10.13 Comparing the performance of CompCert and GCC on AArch64. The left figure shows GCC compared relatively to CompCert across all the benchmarks. The right figure shows the reverse: the gain of using CompCert instead of GCC, for the original version of CompCert and the version with our included optimizations. . . 217
10.14 Comparing the performance of CompCert and GCC on RISC-V. The left figure shows GCC compared relatively to CompCert across all the benchmarks. The right figure shows the reverse: the gain of using CompCert instead of GCC, for the original version of CompCert and the version with our included optimizations. . . 218

List of Tables

10.1 Improvement of Cumulative Optimizations w.r.t. mainstream CompCert 209

Bibliography

[1] 9899:1999. *International standard—Programming languages—C*. Technical Report. ISO/IEC.

[2] Andrew W. Appel. 2011. Verified Software Toolchain. In *Programming Languages and Systems*, Gilles Barthe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17.

[3] Andrew W Appel. 2016. Verifiable C.

[4] Thomas Ball and James R. Larus. 1993. Branch Prediction for Free. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 300–313. https://doi.org/10.1145/155090.155119

[5] Gergö Barany. 2018. A more precise, more correct stack and register model for CompCert. In *LOLA 2018 - Syntax and Semantics of Low-Level Languages 2018*. Oxford, United Kingdom. https://hal.inria.fr/hal-01799629

[6] Gilles Barthe, Delphine Demange, and David Pichardie. 2014. Formal Verification of an SSA-Based Middle-End for CompCert. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 4 (March 2014), 35 pages. https://doi.org/10.1145/2579080

[7] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. *Introduction to Runtime Verification*. Springer International Publishing, Cham, 1–33. https://doi.org/10.1007/978-3-319-75632-5_1

[8] Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. 2012. Formally verified optimizing compilation in ACG-based flight control software. In *ERTS2 2012: Embedded Real Time Software and Systems*. AAAF, SEE, Toulouse, France. https://hal.inria.fr/hal-00653367

[9] Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. 2011. Towards Formally Verified Optimizing Compilation in Flight Control Software. In *PPES 2011: Predictability and Performance in Embedded Systems (OpenAccess Series in Informatics)*, Vol. 18. Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik, Grenoble, France, 59–68. https://doi.org/10.4230/OASIcs.PPES.2011.59

[10] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

[11] Armin Biere. 2008. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation* 4, 2-4 (2008), 75–97.

[12] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2002. *Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software*. Springer Berlin Heidelberg, Berlin, Heidelberg, 85–108. https://doi.org/10.1007/3-540-36377-7_5

[13] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. 2006. Formal Verification of a C Compiler Front-End. In *FM 2006: Formal Methods*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 460–475.

[14] Gabriel Hjort Blindell, Mats Carlsson, Roberto Castañeda Lozano, and Christian Schulte. 2017. Complete and Practical Universal Instruction Selection. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 119 (Sept. 2017), 18 pages. https://doi.org/10.1145/3126528

[15] Sylvain Boulmé. 2021. *Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles)*. Habilitation à diriger des recherches. Université Grenoble-Alpes. https://hal.archives-ouvertes.fr/tel-03356701 See also http://www-verimag.imag.fr/ boulme/hdr.html.

[16] Sylvain Boulmé, David Monniaux, and Cyril Six. 2021. Our version of CompCert for KVX architecture. https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-kvx

[17] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A Formally Verified Compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 586–601. https://doi.org/10.1145/3062341.3062358

[18] Timothy Bourke, Lélio Brun, and Marc Pouzet. 2018. Towards a Verified Lustre Compiler with Modular Reset. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES '18)*. Association for Computing Machinery, New York, NY, USA, 14–17. https://doi.org/10.1145/3207719.3207732

[19] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/1455770.1455776

[20] P. P. Chang and W. W. Hwu. 1988. Trace Selection for Compiling Large C Application Programs to Microcode. In *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture (MICRO 21)*. IEEE Computer Society Press, Washington, DC, USA, 21–29.

[21] Liang-Fang Chao, A.S. LaPaugh, and E.H.-M. Sha. 1997. Rotation scheduling: a loop pipelining algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16, 3 (1997), 229–239. https://doi.org/10.1109/43.594829

[22] Adam Chlipala. 2013. The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 391–402. https://doi.org/10.1145/2500365.2500592

[23] Keith D Cooper, Philip J Schielke, and Devika Subramanian. 1998. An experimental evaluation of list scheduling. *TR98* 326 (1998).

[24] Jack W Davidson and Sanjay Jinturkar. 1995. *An aggressive approach to loop unrolling*. Technical Report. Citeseer.

[25] Benoît Dupont de Dinechin. 2004. From Machine Scheduling to VLIW Instrution Scheduling. *ST Journal of Research* 1, 2 (Sept. 2004), 1–35. https://www.cri.ensmp.fr/classement/doc/A-352.ps Also as Mines ParisTech research article A/352/CRI.

[26] Xavier Leroy et al. 2021. The CompCert verified C compiler. https://github.com/AbsInt/CompCert

[27] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis*. Toulouse, France. https://doi.org/10.4230/OASIcs.WCET.2016.2

[28] Yi Fang and Lenore D. Zuck. 2007. Improved Invariant Generation for Tvoc. *Electronic Notes in Theoretical Computer Science* 176, 3 (2007), 21–35. https://doi.org/

10.1016/j.entcs.2006.06.016 Proceedings of the 5th International Workshop on Compiler Optimization meets Compiler Verification (COCV 2006).

[29] Jean-Christophe Filliâtre and Sylvain Conchon. 2006. Type-Safe Modular Hash-Consing. In *Proceedings of the 2006 Workshop on ML (ML '06)*. Association for Computing Machinery, New York, NY, USA, 12–19. https://doi.org/10.1145/1159876.1159880

[30] Fisher. 1981. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Comput.* C-30, 7 (1981), 478–490. https://doi.org/10.1109/TC.1981.1675827

[31] Fisher. 1981. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Comput.* C-30, 7 (1981), 478–490. https://doi.org/10.1109/TC.1981.1675827

[32] Joseph A. Fisher. 1983. Very Long Instruction Word Architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture (ISCA '83)*. Association for Computing Machinery, New York, NY, USA, 140–150. https://doi.org/10.1145/800046.801649

[33] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. 2005. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[34] Yannick Forster and Fabian Kunze. 2016. Verified extraction from Coq to a lambda-calculus. In *Coq Workshop*, Vol. 2016.

[35] France24. 2021. La panne des numéros d'urgence due à un "bug", selon l'enquête d'Orange. https://www.france24.com/fr/france/20210611-la-panne-des-num%C3%A9ros-d-urgence-due-%C3%A0-un-bug-selon-l-enqu%C3%AAte-d-orange

[36] J. R. Goodman and W.-C. Hsu. 1988. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proceedings of the 2nd International Conference on Supercomputing (ICS '88)*. Association for Computing Machinery, New York, NY, USA, 442–452. https://doi.org/10.1145/55364.55407

[37] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 595–608. https://doi.org/10.1145/2676726.2676975

[38] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[39] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320. https://doi.org/10.1109/5.97300

[40] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. 1993. *The Superblock: An Effective Technique for VLIW and Superscalar Compilation*. Springer US, Boston, MA, 229–248. https://doi.org/10.1007/978-1-4615-3200-2_7

[41] INRIA 2019. *The CompCert C verified compiler—Documentation and user's manual* (version 3.5 ed.). INRIA.

[42] Leslie A Johnson et al. 1998. DO-178B: Software considerations in airborne systems and equipment certification. *Crosstalk, October* 199 (1998), 11–20.

[43] Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. 2018. CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In *ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems*. 3AF, SEE, SIE, Toulouse, France, 1–9. https://hal.inria.fr/hal-01643290

[44] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945

[45] Phil Koopman. 2014. A Case Study of Toyota Unintended Acceleration and Software Safety. https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf

[46] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. 1981. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '81)*. Association for Computing

Machinery, New York, NY, USA, 207–218. https://doi.org/10.1145/567532.567555

[47] Monica S Lam. 1989. Software pipelining. In *A Systolic Array Optimizing Compiler*. Springer, 83–124.

[48] Tom Lane and the Independent JPEG Group (IJG). 1998. Libjpeg. http://libjpeg.sourceforge.net/

[49] M. Lee, P. Tirumalai, and T.-F. Ngai. 1993. Software pipelining and superblock scheduling: compilation techniques for VLIW machines. In *[1993] Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences*, Vol. i. 202–213 vol.1. https://doi.org/10.1109/HICSS.1993.270744

[50] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[51] Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. http://xavierleroy.org/publi/compcert-backend.pdf

[52] Xavier Leroy. 2017. How I found a crash bug with hyperthreading in Intel's Skylake processors. https://thenextweb.com/contributors/2017/07/05/found-crash-bug-hyperthreading-intels-skylake-processors/

[53] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303. https://doi.org/10.1016/j.jlap.2008.08.004 The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[54] Hanbing Li, Isabelle Puaut, and Erven Rohou. 2014. Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems (RTNS '14)*. Association for Computing Machinery, New York, NY, USA, 97–106. https://doi.org/10.1145/2659787.2659805

[55] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. 1993. *The Multiflow Trace Scheduling Compiler*. Springer US, Boston, MA, 51–142. https://doi.org/10.1007/978-1-4615-3200-2_4

[56] lowRISC. 2021. Rocket core overview. https://www.cl.cam.ac.uk/~jrrk2/docs/tagged-memory-v0.1/rocket-core/

[57] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2019. Combinatorial Register Allocation and Instruction Scheduling. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 17 (July 2019), 53 pages. https://doi.org/10.1145/3332373

[58] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2019. Combinatorial Register Allocation and Instruction Scheduling. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 17 (July 2019), 53 pages. https://doi.org/10.1145/3332373

[59] Jamie Lynch. 2017. The Worst Computer Bugs in History: The Ariane 5 Disaster. https://www.bugsnag.com/blog/bug-day-ariane-5-disaster

[60] Alexander B. Magoun and Paul Israel. 2013. Did you Know? Edison Coined the Term "Bug". https://spectrum.ieee.org/the-institute/ieee-history/did-you-know-edison-coined-the-term-bug

[61] Andrey Makhorin. 2012. GNU Linear Programming Kit. https://www.gnu.org/software/glpk/

[62] Giovanni De Micheli. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill.

[63] David Monniaux and Cyril Six. 2021. Simple, Light, Yet Formally Verified, Global Common Subexpression Elimination and Loop-Invariant Code Motion. (2021).

[64] Alain Mosnier. 2019. SHA-256 implementation. https://github.com/amosnier/sha-2

[65] Rajeev Motwani, Krishna V Palem, Vivek Sarkar, and Salem Reyen. 1995. Combining register allocation and instruction scheduling. *Courant Institute, New York University* (1995).

[66] Steven Muchnick et al. 1997. *Advanced compiler design implementation*. Morgan kaufmann.

[67] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 83–94. https://doi.org/10.1145/349299.349314

[68] BBC News. 2013. Bug found in Post Office row computer system. https://www.bbc.com/news/uk-23233573

[69] David Lorge Parnas, GJK Asmis, and Jan Madey. 1991. Assessment of safety-critical software in nuclear power plants. *Nuclear safety* 32, 2 (1991), 189–198.

[70] David L. Parnas, A. John van Schouwen, and Shu Po Kwan. 1990. Evaluation of Safety-Critical Software. *Commun. ACM* 33, 6 (June 1990), 636–648. `https://doi.org/10.1145/78973.78974`

[71] Conor Patrick. 2015. Bitsliced AES implementation. `https://github.com/conorpp/bitsliced-aes`

[72] Karl Pettis and Robert C. Hansen. 1990. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. Association for Computing Machinery, New York, NY, USA, 16–27. `https://doi.org/10.1145/93542.93550`

[73] Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. 2020. Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs. In *Automated Reasoning*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer International Publishing, Cham, 119–137.

[74] André Platzer. 2010. Lecture Notes on Loop-Invariant Code Motion. (2010).

[75] A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernhard Steffen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–166.

[76] Louis-Noël Pouchet. 2012. The Polyhedral Benchmark suite. `http://web.cs.ucla.edu/~pouchet/software/polybench/`

[77] Marc Pouzet. 1994. *Compaction des langages fonctionnels*. Ph.D. Dissertation. Université Paris VII.

[78] B. Ramakrishna Rau, Christopher D. Glaeser, and Raymond L. Picard. 1982. Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support. In *Proceedings of the 9th Annual Symposium on Computer Architecture (ISCA '82)*. IEEE Computer Society Press, Washington, DC, USA, 131–139.

[79] J. Richardson and J. Green. 2004. Automating traceability for generated software artifacts. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.* 24–33. `https://doi.org/10.1109/ASE.2004.1342721`

[80] Lionel Rieg and David Monniaux. 2021. Modified version of Vélus for compatibility. `https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/velus`

[81] Xavier Rival. 2004. Symbolic Transfer Function-Based Approaches to Certified Compilation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/964001.964002

[82] Valentin Robert and Xavier Leroy. 2012. A Formally-Verified Alias Analysis. In *Certified Programs and Proofs*, Chris Hawblitzel and Dale Miller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 11–26.

[83] Vivek Sarkar, Mauricio J. Serrano, and Barbara B. Simons. 2001. Register-Sensitive Selection, Duplication, and Sequencing of Instructions. In *Proceedings of the 15th International Conference on Supercomputing (ICS '01)*. Association for Computing Machinery, New York, NY, USA, 277–288. https://doi.org/10.1145/377792.377849

[84] Olivier Savary Bélanger et al. 2019. Verified Extraction for Coq. (2019).

[85] Thomas Sewell. 2017. *Translation validation for verified, efficient and timely operating systems*. Ph.D. Dissertation. University of New South Wales, Sydney, Australia. http://handle.unsw.edu.au/1959.4/58861

[86] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 471–482. https://doi.org/10.1145/2491956.2462183

[87] Ghassan Shobaki, Maxim Shawabkeh, and Najm Eldeen Abu Rmaileh. 2013. Preallocation Instruction Scheduling with Register Pressure Minimization Using a Combinatorial Optimization Approach. *ACM Trans. Archit. Code Optim.* 10, 3, Article 14 (Sept. 2013), 31 pages. https://doi.org/10.1145/2512432

[88] Cyril Six, Sylvain Boulmé, and David Monniaux. 2020. Certified and Efficient Instruction Scheduling: Application to Interlocked VLIW Processors. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 129 (Nov. 2020), 29 pages. https://doi.org/10.1145/3428197

[89] Cyril Six, Léo Gourdin, Sylvain Boulmé, and David Monniaux. 2021. Verified Superblock Scheduling with Related Optimizations. (April 2021). https://hal.archives-ouvertes.fr/hal-03200774 preprint.

[90] Stéphane Strahm. 2021. High Performance Programming: Offloading on Manycore Architecture.

[91] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The Verified CakeML Compiler Backend. *Journal of Functional Programming* 29 (February 2019). `https://kar.kent.ac.uk/71304/`

[92] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. Association for Computing Machinery, New York, NY, USA, 264–276. `https://doi.org/10.1145/1480881.1480915`

[93] Jean-Baptise Tristan. 2009. *Formal verification of translation validators*. Ph.D. Dissertation. Université Paris 7 Diderot.

[94] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-Graph Translation Validation for LLVM. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 295–305. `https://doi.org/10.1145/1993498.1993533`

[95] Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 17–27. `https://doi.org/10.1145/1328438.1328444`

[96] Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 17–27. `https://doi.org/10.1145/1328438.1328444`

[97] Jean-Baptiste Tristan and Xavier Leroy. 2009. Verified Validation of Lazy Code Motion. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 316–326. `https://doi.org/10.1145/1542476.1542512`

[98] Jean-Baptiste Tristan and Xavier Leroy. 2010. A Simple, Verified Validator for Software Pipelining. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. Association for Computing Machinery, New York, NY, USA, 83–92. `https://doi.org/10.1145/1706299.1706311`

[99] Sven Verdoolaege. 2016. Presburger formulas and polyhedral compilation. (2016).

[100] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[101] Junbeom Yoo, Eunkyoung Jee, and Sungdeok Cha. 2009. Formal Modeling and Verification of Safety-Critical Software. *IEEE Software* 26, 3 (2009), 42–49. https://doi.org/10.1109/MS.2009.67

[102] Yiji Zhang and Lenore D. Zuck. 2018. Formal Verification of Optimizing Compilers. In *Distributed Computing and Internet Technology*, Atul Negi, Raj Bhatnagar, and Laxmi Parida (Eds.). Springer International Publishing, Cham, 50–65.

[103] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. Association for Computing Machinery, New York, NY, USA, 427–440. https://doi.org/10.1145/2103656.2103709

[104] Lenore D. Zuck, Amir Pnueli, and Benjamin Goldberg. 2003. VOC: A Methodology for the Translation Validation of Optimizing Compilers. *J. UCS* 9, 3 (2003), 223–247. https://doi.org/10.3217/jucs-009-03-0223