Universal Instruction Selection

GABRIEL HJORT BLINDELL

Doctoral Thesis in Information and Communication Technology
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden 2018

School of Electrical Engineering
and Computer Science
KTH Royal Institute of Technology
SE-164 40 Kista
SWEDEN

TRITA-EECS-AVL-2018:11
ISBN: 978-91-7729-683-6

Akademisk avhandling som med tillstånd av Kungliga Tekniska Högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen på fredagen den 6 april 2018 kl. 13:15 i Sal B, Electrum, Kungliga Tekniska Högskolan, Kistagången 16, Kista.

Printed by Universitetsservice US-AB

I may not have achieved all that I wanted to achieve
but I am proud of what I did achieve

Abstract

In code generation, instruction selection chooses instructions to implement a given program under compilation, global code motion moves computations from one part of the program to another, and block ordering places program blocks in a consecutive sequence. Local instruction selection chooses instructions one program block at a time while global instruction selection does so for the entire function. This dissertation introduces a new approach called *universal instruction selection* that integrates global instruction selection with global code motion and block ordering. By doing so, it addresses limitations of existing instruction selection techniques that fail to exploit many of the instructions provided by modern processors.

To handle the combinatorial nature of these problems, the approach is based on constraint programming, a combinatorial optimization method. It relies on a novel model that is simpler and more flexible compared to the techniques used in modern compilers and that captures crucial features ignored by other combinatorial approaches. The dissertation also proposes extensions to the model for integrating instruction scheduling and register allocation, two other important problems of code generation.

The model is enabled by a novel, graph-based representation that unifies data and control flow for entire functions. The representation is crucial for integrating instruction selection with global code motion and for modeling sophisticated instructions, whose behavior contains both data and control flow, as graphs.

Through experimental evaluation, universal instruction selection is demonstrated to handle architectures with a rich instruction set and scales up to functions with hundreds of operations. For these functions, it generates code of equal or better quality compared to the state of the art. The dissertation also demonstrates that there is sufficient data parallelism to be exploited through selection of SIMD instructions and that this exploitation benefits from global code motion. With these results, it is argued that constraint programming is a flexible, practical, competitive, and extensible approach for combining global instruction selection, global code motion, and block ordering.

Sammanfattning

Inom kodgenerering väljer instruktionsselektering (eng. *instruction selection*) instruktioner för att implementera ett givet program under kompilering, global kodförflyttning (eng. *global code motion*) flyttar beräkningar från en del av programmet till en annan, och blockläggning (eng. *block ordering*) placerar programblock i en sekventiell följd. Lokal instruktionsselektering väljer instruktioner ett programblock i taget medan global instruktionsselektering gör så för hela funktionen. Denna avhandling introducerar en ny metod, kallad *universell instruktionsselektering*, som integrerar global instruktionsselektering med global kodförflyttning och blockläggning. På så vis åtgärdar den begränsningar hos befintliga instruktionsselekteringsmetoder som misslyckas med att utnyttja många av instruktionerna som erbjuds av moderna processorer.

För att hantera den kombinatoriska naturen av dessa problem tillämpas villkorsprogrammering (eng. *constraint programming*), en teknik för kombinatorisk optimering. Metoden använder en innovativ model som är enklare och mer flexibel jämfört med metoderna som används i moderna kompilatorer och som fångar viktiga särdrag som ignoreras av andra kombinatoriska metoder. Avhandlingen föreslår också utökningar av modellen för att integrera instruktionsschemaläggning (eng. *instruction scheduling*) och registerallokering (eng. *register allocation*), två andra viktiga kodgenereringsproblem.

Modellen möjliggörs av en innovativ, grafbaserad representation som förenar data- och kontrollflöde för hela funktioner. Representationen är avgörande för att integrera instruktionsselektering med global kodförflyttning och för att modellera sofistikerade instruktioner, vars beteende omfattar både data- och kontrollflöde, som grafer.

Genom experimentell utvärdering visas att universell instruktionsselektering kan hantera arkitekturer med ett rikt instruktionsset och skalar upp till funktioner med hundratals beräkningar. För dessa funktioner genererar den kod av motsvarande eller bättre kvalitet än den senaste tekniken. Avhandlingen visar också att det finns tillräckligt med dataparallellism att utnyttja genom selektering av SIMD-instruktioner och att denna exploatering gynnas av global kodförflyttning. Med dessa resultat argumenteras för att villkorsprogrammering är en flexibel, praktisk, konkurrenskraftig, och utökningsbar metod för att kombinera global instruktionsselektering, global kodförflyttning, och blockläggning.

Acknowledgements

First and foremost, I want to thank my main supervisor Christian Schulte for accepting me as his student and overseeing my studies. Your scholarship and wisdom has been a respectable source of knowledge and an inspiration, which has helped me grow both intellectually and personally. In particular, you have motivated me to persevere. For example, one morning I decided – after having endured months of great distress – that I would abandon my studies. I went to your office to inform my decision but, after listening to your enlightening words, I amended my decision, thinking that I would give it one more go. Had I had any other mentor at that pivotal moment, this dissertation would undoubtedly not have seen the light of day.

I want to thank my co-supervisor Mats Carlsson for scrutinizing and improving my work. Your expertise has been invaluable for putting my ideas into practice, especially in devising many of the solving techniques. Without your help, my research would not have matured to the level it is today.

I want to thank my co-supervisor Ingo Sander for giving me a different perspective on things when I needed a second opinion.

I want to thank Roberto Castañeda Lozano for being my closest colleague during my studies. It has been a privilege working next to you all these years, and I could not have asked for a better co-worker to share ideas, problems, nerdy comments, and laughter with.

I want to thank Prof. Peter van Beek for acting as opponent and Prof. Christoph Kessler, Prof. Krzysztof Kuchcinski, and Prof. Christine Solnon for acting as PhD committee on my doctoral defense. I also want to thank Prof. Christoph Kessler for contributing to my PhD proposal, and I want to thank Prof. Elena Dubrova for examining my dissertation for internal review and attending my PhD proposal.

I want to thank Frej Drejhammar, Mattias Jansson, and Karl Johansson for setting up the framework to capture target machines as a high-level, machine-readable description. I also want to thank Muhammad Waseem Arshad for helping me using this framework for modeling Hexagon.

I want to thank the Swedish Research Council (VR grant 621-2011-6229) for funding my research, and RISE SICS for letting me use its facilities. I also want to thank my colleagues at KTH and RISE SICS for creating a friendly working environment.

I want to thank everyone at TEX StackExchange (`tex.stackexchange.com`) for their help with all LATEX problems I encountered during my studies. Without it, writing this dissertation would have been a much more arduous task.

I want to thank my parents for their loving support and encouragement.

Last but not least, I want to thank Linda Åkerlund for being my companion in so many aspects of life: eating buckets of popcorn while watching TV, folding laundry together, setting up excel sheets for everything, laughing until our cheeks hurt, crying in each other's arms, traveling to faraway places, trusting one another through thick and thin, loving the night away, and dancing for hours on end.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ACK	Amsterdam Compiler Kit
AMOP	abstract machine operation
ASIP	application-specific instruction set processor
ASP	answer set programming
AST	abstract syntax tree
AVX	advanced vector extensions
BEG	Back End Generator
BURS	bottom-up rewriting system
CBC	Common Bus Compiler
CGG	Code-Generator Generator
CGL	Code Generator Language
CGPL	Code Generator Preprocessor Language
CI	confidence interval
CISC	complex instruction-set computer
CNF	conjunctive normal form
CO	connection operation
CP	constraint programming
CV	coefficient of variation
DAG	directed acyclic graph
DP	dynamic programming
DSP	digital signal processor
DTB	dual-target branch
ERI	extended resource information
FBB	functional building block
FHC	Fortran H Compiler
FRT	factorized register transfer
GA	genetic algorithm
GCC	GNU Compiler Collection
GMI	geometric mean improvement
GRiP	global resource-constrained percolation
ICL	Interpretive Coding Language
ILP	integer linear programming
SIMD	single-instruction, multiple-data
IP	integer programming
IR	intermediate representation
ISE	instruction set extension
ISFG	internal signal-flow graph
JHSC	Java Hotspot Server Compiler
JIT	just-in-time

LCC Little C Compiler
LCG lazy clause generation
LR local rewrite
MIML Machine-Independent Macro Language
MIMOLA Machine Independent Microprogramming Language
MIS maximal independent set
ML Metalanguage
MWIS maximal/minimal weighted independent set
OMML Object Machine Macro Language
OVA optimal value array
PAS preferred attribute set
PBQP partitioned Boolean quadratic problem
PCC Portable C Compiler
PO Peephole Optimizer
PQCC Production Quality Compiler-Compiler
QAP quadratic assignment problem
RISC reduced instruction-set computer
RT register transfer
RTL register transfer list
SAT Boolean satisfiability
SLM source language machine
SSA static single assignment
SSE streaming SIMD extensions
TEL Template Language
UF universal function
UI uniquely invertable
UP universal pattern
VLIW very long instruction word
VLSI very large scale integration
XL Extensible Language
YC Y Compiler

CHAPTER

1

Introduction

Processors are built to execute a vast range of programs, from tiny *Hello, world!* samples to large-scale Earth simulations. Most importantly, the processors are built to minimize the execution time for these programs. To this end, CPU manufacturers continuously extend their processors with new, sophisticated instructions that allow complex computations to be executed using fewer instructions. Such instructions are especially common in *digital signal processors (DSPs)* that appear in most contemporary mobile phones. But while the technology behind modern processors continues to advance, the techniques for *instruction selection* – the task of choosing the instructions for a given program – have not. In fact, the state-of-the-art compilers, which are tools for translating programs into assembly code, essentially apply the same instruction selection techniques as were used in the 1980s.

Problem Statement Due to underlying assumptions, many of the instructions currently available in modern processors cannot be handled by these techniques. In particular, they rely on representations that are too limited for modeling these instructions. Instead, compiler developers are forced to implement hand-written routines for checking whether a specific instruction is applicable and, if so, greedily selecting it. With over 100 million microprocessors being shipped every quarter [199], through release cycles that become shorter and shorter, there is a growing need for new and improved instruction selection techniques.

Furthermore, the set of selectable instructions is highly dependent on other compiler tasks. One such task is *global code motion*, which involves moving computations from one part of the program to another. Integrating global code motion with instruction selection enables a larger set of combinations of computations, some of which may be implemented using sophisticated instructions. Another task is *block ordering*, which involves placing the program blocks in a consecutive sequence. Depending on the processor, one set of selected instructions may impact the possible block sequence and vice versa. Consequently, in order to generate high-quality code, these tasks must be performed in unison.

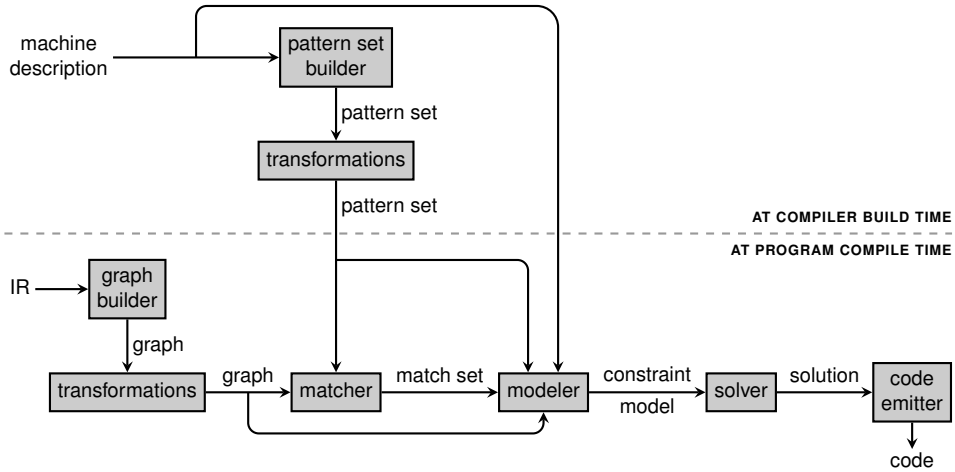


Figure 1.1: Overview of universal instruction selection.

Universal Instruction Selection This dissertation introduces *universal instruction selection* – a new approach that addresses the problems recently described.[1] Outlined in Fig. 1.1, the approach is the first to combine instruction selection with global code motion and block ordering. In doing so, the approach alleviates selection of sophisticated instructions that would otherwise not have been selectable.

To handle the combinatorial nature of these problems, the approach is based on a combinatorial optimization method called *constraint programming*. It relies on a novel combinatorial model that is simpler and more flexible compared to the techniques currently used by modern compilers. In addition, it captures crucial features that are ignored by other, existing combinatorial approaches. The dissertation also proposes extensions for integrating instruction scheduling and register allocation, which are two other code generation tasks known to impact instruction selection.

The model is enabled by a novel, graph-based representation that unifies data flow and control flow for entire functions. Not only is this representation crucial for combining instruction selection with global code motion, it also enables instructions whose behavior contains both data and control flow to be modeled as graphs. Hence there is no longer any need for hand-written routines to handle instructions that violate underlying assumptions about the instruction set.

1.1 Thesis Statement

Constraint programming is a flexible, practical, competitive, and extensible approach for combining global instruction selection, global code motion, and block ordering.

[1] The source code is freely available on `github.com/unison-code/uni-instr-sel`.

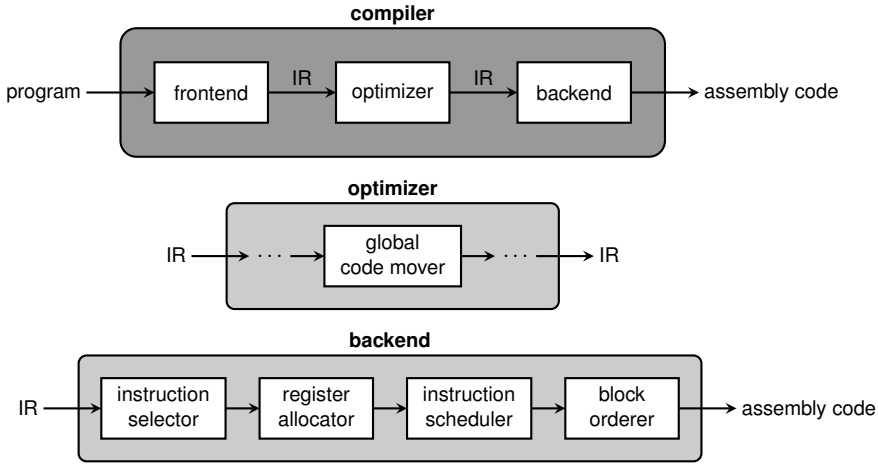


Figure 1.2: Overview of a typical compiler.

By *flexible*, it means that the approach can handle hardware architectures with a rich instruction set. By *practical*, it means that the approach can select instructions for programs of sufficient complexity and scales to medium-sized functions (measured in hundreds of operations). By *competitive*, it means that the approach generates code of equal or better quality compared to the state of the art. By *extensible*, it means that the approach can be extended to integrate other code generation tasks.

1.2 Motivation

A *compiler* is a tool that takes a program, written in some programming language, as input and produces equivalent assembly code for a specific processor, called the *target machine*, as output. As shown in Fig. 1.2, a compiler typically consists of three parts: a frontend, an optimizer, and a backend.

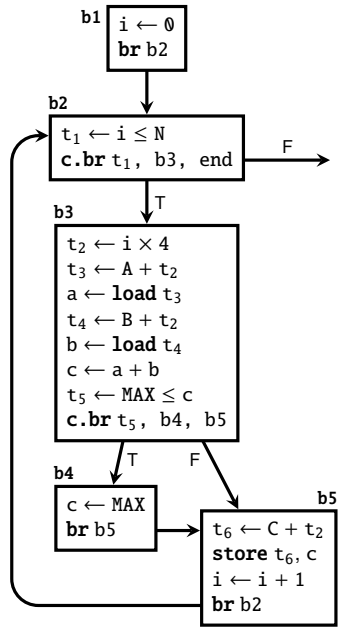
The *frontend* performs syntactic and semantic analysis on the program under compilation, making sure that the program is syntactically and semantically valid. After passing all checks, it then transforms the program into an *intermediate representation (IR)* and passes the code to the optimizer.

The *optimizer* (sometimes called *middle-end*) consists of many target-independent program optimizations, such as constant folding, dead code elimination, and loop unrolling. Consequently, it is often the largest component of most compilers. Global code motion is typically also included in this component, where operations are moved across block boundaries in order to move expensive operations into blocks with lower execution frequency. Once optimized, the IR code is then passed to the backend.

The *backend* performs code generation, which also consists of many tasks but

```
int i = 0;
while (i < N) {
  int a = A[i];
  int b = B[i];
  int c = a + b;
  if (MAX < c) c = MAX;
  C[i] = c;
  i++;
}
```

(a) C code.



(b) Corresponding IR and control-flow graph.

Figure 1.3: Example to illustrate the need for new techniques and the interaction between instruction selection and global code motion. The program computes the saturated sums of two arrays `A` and `B` as a new array `C`, all of which are assumed to be of equal length and stored in memory. The variables `N` and `MAX` are constants representing the array length and the upper limit, respectively. An integer is assumed to be four bytes.

of which three tasks are most prominent: instruction selection, which we are already familiar with; *register allocation*, where temporaries are assigned to registers; and *instruction scheduling*, where instructions are reordered to increase instruction-level parallelism and avoid stalls. Among other tasks, the backend then performs block ordering in order to minimize the number of jump instructions. After these steps the program has been fully transformed into assembly code, which can then be translated into machine code to be executed by the target machine.

1.2.1 The Need for New Techniques and Representations

Figure 1.3 shows a program that computes the saturated sums of two integer arrays. In *saturation arithmetic*, the result of an arithmetic operation will always stay within a range fixed by a minimum and maximum value. If the operation would produce a value outside of this range, then the value is set ("clamped") to the closest limit,

thus becoming "saturated".

Assume that the target machine has an instruction capable of implementing the saturated-add operation used in the program shown in Fig. 1.3. Hence the instruction would implement the following five operations: the `a + b` addition, the `MAX` $\leq$ `c` comparison, the conditional jump to either of blocks `b4` and `b5`, the `c` $\leftarrow$ `MAX` assignment, and the unconditional jump to `b5`. Selecting this instruction can have tremendous impact on performance. Assume, for example, that each operation can be implemented using an instruction that takes one cycle to execute. Hence executing one iteration of the loop takes 16 cycles, and selecting the saturated-add instruction would reduce the execution time by 25 %.

Existing instruction selection techniques and representations, however, do not support selection of such instructions. Since the operations above reside in separate blocks (`b3` and `b4`), making use of the saturated-add instruction requires an instruction selector that is capable of processing multiple basic blocks simultaneously. In comparison, traditional instruction selection techniques only consider one basic block at a time. Moreover, most approaches represent the instructions as graphs. As the saturated-add instruction contains operations for both data and control flow, modeling it as a graph requires a representation that captures both data and control flow. In comparison, existing representations only capture data flow.

1.2.2 For Combining Instruction Selection and Global Code Motion

Assume that the target machine also has a SIMD instruction for addition.[2] Revisiting the example shown in Fig. 1.3, there are four additions in the program (`A` + t_2, `B` + t_2, `C` + t_2, and `i + 1`) which are independent from one another and can therefore be executed in parallel. Assuming again that all instructions take one cycle to execute, implementing these four additions using a single SIMD instruction would reduce execution time by almost 19 %. This requires, however, that the two additions in block `b5` be moved to block `b4`, which is the task of global code motion. Since global code motion is commonly considered to be a target-independent optimization, this task is often done before code generation. Consequently, the global code mover may take decisions which prevent selection of such instructions.

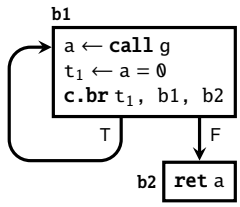
1.2.3 For Taking Cost of Data Copying Into Account

Although selecting SIMD instructions may significantly improve code quality – like in the previous example – doing so carelessly may also have the opposite effect. Assume, for example, that the SIMD instruction uses a limited set of registers. If the other selected instructions cannot directly write to and read from these registers, then additional instructions must be emitted to copy the values between the general registers and the SIMD registers. In the case of the program shown in Fig. 1.3, eight

[2] A *single-instruction, multiple-data (SIMD) instruction* is an instruction that executes the same operation over multiple sets of input data.

```
int f() {
  int a;
  do {
    a = g();
  } while (a == 0);
  return a;
}
```

(a) C code.



(b) Corresponding IR and control-flow graph.

```
b1: call a ← g()
    cmp  t1 ← a = 0
    jmp  t1, b1
b2: ret  a
```

(c) Selecting basic jump instruction, after block ordering. Cycle count: 6.

```
b1: call a ← g()
    jmp  a ≠ 0, b2

b2: ret  a
```

(d) Selecting complex jump instruction, before block ordering. Cycle count: 5.

```
b1: call a ← g()
    jmp  a ≠ 0, b2
    jmp  b1
b2: ret  a
```

(e) Selecting complex jump instruction, after block ordering. Cycle count: 8.

Figure 1.4: Example to illustrate the interaction between instruction selection and block ordering. The function `f` calls another function `g` until it returns a non-zero value, and then returns that value.

such instructions would be needed, which instead increases execution time with 31 %. The task of inserting these copy instructions is known as *data copying*, and the instruction selector must be aware of the cost of data copying in order to make effective use of SIMD instructions.

1.2.4 For Combining Instruction Selection and Block Ordering

Figure 1.4 shows a function that keeps calling another function (with side effects) until it returns a non-zero value. Assume that the target machine has three instructions for handling control flow: a `jmp` p, b instruction, which branches to block b if the value in register p corresponds the Boolean value *true*; a `jmp` $r \neq 0$, b instruction, which branches to block b if the condition $r \neq 0$ holds, where r is a register; and a `jmp` b instruction, which unconditionally branches to block b. Assume also that these branch instructions take three cycles compared to the other instructions in the target machine, which take one cycle.

At first glance it appears that only the first jump instruction is selectable for implementing the conditional branch (see Fig. 1.4c). Selecting this instruction leads to a total of six cycles for the entire function. But by flipping the condition and swapping block labels (conditionally jumping to `b2` instead of `b1`), the more complex jump instruction becomes selectable (see Fig. 1.4d). Selecting this instruction brings the cycle count to five cycles, thus reducing the execution time by almost 17 %. However, although this decision may appear better at the point of instruction

selection, it requires an additional jump instruction when ordering the blocks (because block `b1` cannot fall-through to the top of itself; see Fig. 1.4e). This code takes eight cycles to execute, thus increasing execution time with 33 %. The instruction selector must therefore be aware of additional jump instructions that may be required when making such decisions.

1.3 Contributions

The dissertation makes six contributions to the areas of code generation and constraint programming:

C1 It presents a comprehensive and systematic survey that
  a. examines over four decades of research on instruction selection, covering a significantly wider scope and time span compared to existing surveys [51, 68, 153, 242, 260] which are either too old or incomplete.

  The survey identifies
  b. four fundamental instruction selection principles – macro expansion, tree covering, DAG covering, and graph covering –

  and
  c. five instruction characteristics – single-output, multi-output, disjoint-output, inter-block, and interdependent –

  and systematically classifies the techniques along these two dimensions. In addition, the survey
  d. identifies connections between instruction selection and other code generation problems that have yet to be investigated.

C2 It introduces a novel program and instruction representation that
  a. captures both data and control flow for entire functions and instructions,

  which enables
  b. an unprecedented range of instruction behaviors to be captured and modeled as graphs.

  In addition, the representation is crucial for
  c. combining instruction selection and global code motion and solving these two problems in unison.

C3 It introduces a constraint model and related transformations for universal instruction selection which, for the first time, enables
  a. uniform selection of data and control instructions,

  and integration of
  b. global instruction selection with
  c. global code motion.

  In addition, the constraint model integrates
  d. data copying,
  e. value reuse, and
  f. block ordering.

chapter	C1	C2	C3	C4	C5	C6
2	✓	·	·	·	·	·
4	·	✓	·	·	·	·
5	·	·	✓	·	·	·
6	·	·	·	✓	·	·
7	·	·	·	·	✓	·
8	·	·	·	·	·	✓
A	✓	·	·	·	·	·
B	✓	·	·	·	·	·
C	✓	·	·	·	·	·
D	✓	·	·	·	·	·

Table 1.1: Contributions per chapter.

C4 It introduces techniques to improve solving of the constraint model, which are essential for scalability and making the approach work in practice.

C5 It presents thorough experiments demonstrating that the approach scales to medium-sized programs and yields equal or better code than the state of the art.

C6 It describes how the constraint model can be extended to integrate other code generation tasks, such as instruction scheduling and register allocation.

Tab. 1.1 shows in which part of the dissertation each contribution is manifested.

The material presented in Chap. 6 is based on ideas conceived in collaboration with Mats Carlsson.

1.4 Publications

This dissertation is based on material presented in the following publications:

Books

P1 G. Hjort Blindell. *Instruction Selection: Principles, Methods, and Applications*. Springer, 2016.

Conference Papers

P2 G. Hjort Blindell, R. Castañeda Lozano, M. Carlsson, and C. Schulte. "Modeling Universal Instruction Selection". In: *Proceedings of CP'15*. Springer, 2015, pp. 609–626.

Contribution The author of this dissertation designed and implemented the work presented in the paper, oversaw the writing of the paper, wrote the

publication	C1	C2	C3						C4	C5	C6
			a	b	c	d	e	f			
P1	✓	·	·	·	·	·	·	·	·	·	·
P2	·	✓	✓	✓	✓	✓	·	✓	·	✓	·
P3	·	✓	·	·	·	·	✓	·	✓	✓	·

Table 1.2: Contributions per publication.

majority of the text, designed the figures, and assisted in experiment data gathering and analysis.

Articles

P3 G. Hjort Blindell, M. Carlsson, R. Castañeda Lozano, and C. Schulte. "Complete and Practical Univeral Instruction Selection". In: *ACM Transactions on Embedded Computing Systems* 16.5s (2017), 119:1–119:18.

Contribution The author designed and implemented the work presented in the paper, gathered and analyzed the experiment data, oversaw the writing of the paper, wrote the majority of the text, and designed the figures.

Tab. 1.2 shows the relation between the contributions and the publications above. Note that contribution C6 only appears in this dissertation and in none of the publications.

The author also participated in the following publications not included in the dissertation:

Book Chapters, Conference Papers, and Workshop Papers

P4 G. Hjort Blindell. *Survey on Instruction Selection: An Extensive and Modern Literature Study*. Tech. rep. Stockholm, Sweden: KTH Royal Institute of Technology, 2013.

P5 R. Castañeda Lozano, G. Hjort Blindell, M. Carlsson, F. Drejhammar, and C. Schulte. "Constraint-based Code Generation". In: *Proceedings of M-SCOPES'13*. Springer, 2013, pp. 93–95.

Contribution The author assisted in writing the paper.

P6 R. Castañeda Lozano, M. Carlsson, G. Hjort Blindell, and C. Schulte. "Combinatorial Spill Code Optimization and Ultimate Coalescing". In: *Proceedings of LCTES'14*. ACM, 2014, pp. 23–32.

Contribution The author assisted in experiment data gathering and analysis, and in writing the paper.

P7 R. Castañeda Lozano, M. Carlsson, G. Hjort Blindell, and C. Schulte. "Register Allocation and Instruction Scheduling in Unison". In: *Proceedings of CC'16*. ACM, 2016, pp. 263–264.

Contribution The author assisted in writing the paper.

P8 G. Hjort Blindell, C. Menne, and I. Sander. "Synthesizing Code for GPGPUs from Abstract Formal Models". In: *Languages, Design Methods, and Tools for Electronic System Design*. Vol. 361. Lecture Notes in Electrical Engineering. Springer, 2016, pp. 115–134.

Contribution The author designed and implemented the work presented in the paper, gathered and analyzed the experiment data, oversaw the writing of the paper, wrote the majority of the text, and designed the figures.

P4 is excluded as it is subsumed and extended by P1. P5–P7 are excluded as they are only partially related to the dissertation (they apply constraint programming to solve register allocation and instruction scheduling without considering instruction selection). P8 is excluded as it belongs to a different topic entirely (high-level code generation for graphics processors).

1.5 Research Methodology

We begin with a comprehensive and systematic literature survey to identify the strengths and limitations of existing instruction selection techniques and common denominators among them. As part of this survey, four fundamental instruction selection principles and five instruction characteristics are identified, and the techniques are systematically classified accordingly. This classification enables us to recognize that certain classes of instructions are poorly supported by existing instruction selection techniques. This is in particular due to lack of appropriate program and instruction representations.

Having established the need for new representations, we identify a set of requirements that such a representation must fulfill. We then build a new representation by unifying two existing, well-established representations – one for capturing data flow and another for capturing control flow – and then augment the representation as needed until all requirements are met. As is common, we then apply a traditional subgraph isomorphism algorithm for performing pattern matching on the new representation.

With the new representation at hand, we proceed with building the constraint model. For each task to be integrated, we first identify what constitutes a solution to this task and then add the necessary variables and constraints to enforce such solutions. If more than one design choice exists for integrating the task, then we implement both as separate models and evaluate which is better before proceeding. This is because the tasks are orthogonal from one another and can therefore be evaluated in isolation. The evaluation methodology is described in Sect. 1.6.

Once all tasks are integrated, we design a range of solving techniques in order to increase scalability and robustness. Because these techniques influence one another, we first evaluate the usefulness of each solving technique individually in order to form groups of solving techniques and then evaluate each group as a whole. This is to avoid unreasonably long experiment runtimes.

Using the constraint model with the best design choices and solving techniques, we then evaluate the significance of universal instruction selection by comparing the code it generates with that generated by the state of the art.

1.6 Evaluation Methodology

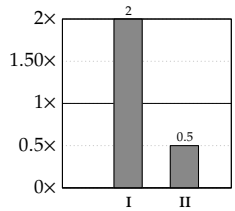
1.6.1 Experiment Setup

To evaluate a constraint model, we implement it using *MiniZinc* 2.1.6 [280], which is a high-level constraint modeling language. The algorithms for transforming a given function into a graph, performing pattern matching, and producing the data to instantiate the constraint model are implemented in Haskell. The presolving techniques are implemented in Python. The model is solved using *Chuffed* [76] – a lazy clause generation-based solver included in MiniZinc – on a Linux machine with an Intel Xeon W3530 at 2.80 GHz and 16 GB main memory. We invoke Chuffed with flags `-f -rnd-seed 3218642`.

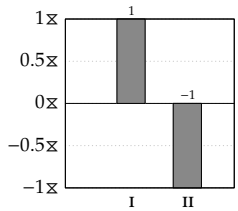
To mitigate deviations in the measurements, we run each experiment ten times and then take the arithmetic average together with the *coefficient of variation (CV)*. When summarizing ratios, we instead take the geometric mean since this is more appropriate in such cases [134].

As problem instances, we use the functions provided by *MediaBench* [241] – a benchmark suite for embedded systems – and the instructions in Hexagon V5 – a DSP with a rich instruction set. The MediaBench suite consists of 6313 functions, which are compiled into LLVM IR code – the intermediate format used by LLVM – using LLVM 3.8 with optimization flag `-O3`. Due to insufficient support in the current tool chain, we remove all functions that operate on non-integer data types (such as floating point and vector data types). This leaves 3094 functions, on which further filtering will be performed as needed for the given experiment.

For all experiments in this dissertation, the remaining pool of functions is too large for all to be included in the experiment as that would lead to unreasonably long experiment runtimes. We therefore draw a limited number of samples from this pool. To attain a diverse set of samples, we apply k-means clustering [302] to divide the functions into twenty clusters based on three features. The first feature is the application from which the function is derived since, intuitively, functions from different applications should exhibit different characteristics. The second feature is the size in number of LLVM IR instructions. This is to evaluate how the constraint model behaves as the functions grow larger. The third feature is the number of memory instructions. This is because, due to its many addressing modes, the memory instructions constitute a large part of the more sophisticated



(a) Values normalized using speedup method (Eq. 1.1).



(b) Values normalized using zero-centered normalization (Eq. 1.2).

Figure 1.5: Comparison between the two methods for normalizing measurements. The comparison is done for two problem instances: one where the subject is twice as fast as the baseline (I), and one where the baseline is twice as fast as the subject (II).

instructions in the instruction set. Based on these features, from each cluster we then randomly select one function, giving a total of twenty functions.

1.6.2 Normalizing Measurements

When we are interested in the relative difference between the measurements, we normalize all values to those obtained from one model which has been chosen as the *baseline*. In this context, the other models are called the *subjects*.

For time measurements, the most typical method for normalization is to calculate the *speedup*, which is computed as

$$\frac{t_b}{t_a} \tag{1.1}$$

where t_a and t_b denote time measurements obtained for a given problem instance from subject model a respectively baseline model b. A value greater than 1 means a is faster than b, a value less than 1 means b is faster than a, and a value of exactly 1 means a and b are equally fast.

This method, however, creates problems when plotting the normalized values as bar charts. Since the normalized values are centered around 1, when a is twice as fast as b we intuitively expect the normalized value to be equally far away from 1 as when b is twice as fast as a. With Eq. 1.1, this is not the case. See for example Fig. 1.5a, illustrating two cases: I, where a is twice as fast as b; and II, where b is twice as fast as a. When plotting such values, we therefore normalize the values using a different method, called *zero-centered normalization*, which is computed as

$$\begin{cases} \frac{t_b - t_a}{t_a} & \text{if } t_b \geq t_a, \\ \frac{t_b - t_a}{t_b} & \text{otherwise.} \end{cases} \tag{1.2}$$

As seen in Fig. 1.5b, this method results in normalized values that are centered around 0 (hence its name) and, unlike the speedup method, are equally far apart

in the case of I and II. A normalized value v using zero-centered normalization corresponds a speedup of $v + 1$ if $v \geq 0$, otherwise $\frac{1}{1-v}$. To distinguish between the two, we suffix speedup and zero-centered values with a × and ⧖, respectively (e.g. 3.50× and 2.50⧖).

Zero-centered normalization is also applied when comparing solution costs as they represent an estimate of the time it would take to execute the code yielded by the solutions (the current implementation is not yet able to hook back into the compiler in order to generate executable code).

When summarizing normalized values, however, we do not use zero-centered normalization as the geometric mean cannot be computed for such values. Instead we compute the geometric mean for the values normalized using the speedup method, which we call the *geometric mean improvement (GMI)*. A result is considered positive or negative if the GMI is greater than respectively less than 1. In terms of solving time, for example, a subject model a leads to an overall reduction over another baseline model b if the GMI is greater than 1.

1.6.3 Attaining Statistically Significant Results

Because we can only run experiments on a small set of functions sampled from much larger pool, the GMI we attain from the sampled set may not be representative for the pool as a whole. In other words, the result may not be statistically significant. Therefore, for the GMI value we also compute the *confidence interval (CI)* [283], which is a method of estimating the uncertainty of a value computed from a set of samples. Every CI is computed with a predefined degree of confidence, typically written as the X^{th} CI. This means that if the X^{th} CI of the GMI value is greater than or less than 1, then we can statistically conclude with X % confidence that the result is positive respectively negative. If the CI contains 1, then the result is inconclusive.

For the experiments described in this dissertation, we compute the 95^{th} CI as is common for most statistical experiments. Since we know nothing about the underlying distributions for our observations, we compute the CI using percentile bootstrapping with 100 000 iterations [111].

1.7 Outline

As shown in Fig. 1.6, the dissertation is structured into four parts:

Background Provides necessary background material: Chap. 3 describes constraint programming; and Ap. F provides exact definitions of graphs and related terms used throughout the dissertation.

Literature survey Discusses existing instruction selection techniques: Chap. 2 covers the techniques most relevant for universal instruction selection; Aps. A–D cover the remaining techniques; and Ap. E lists a summary of all discussed techniques.

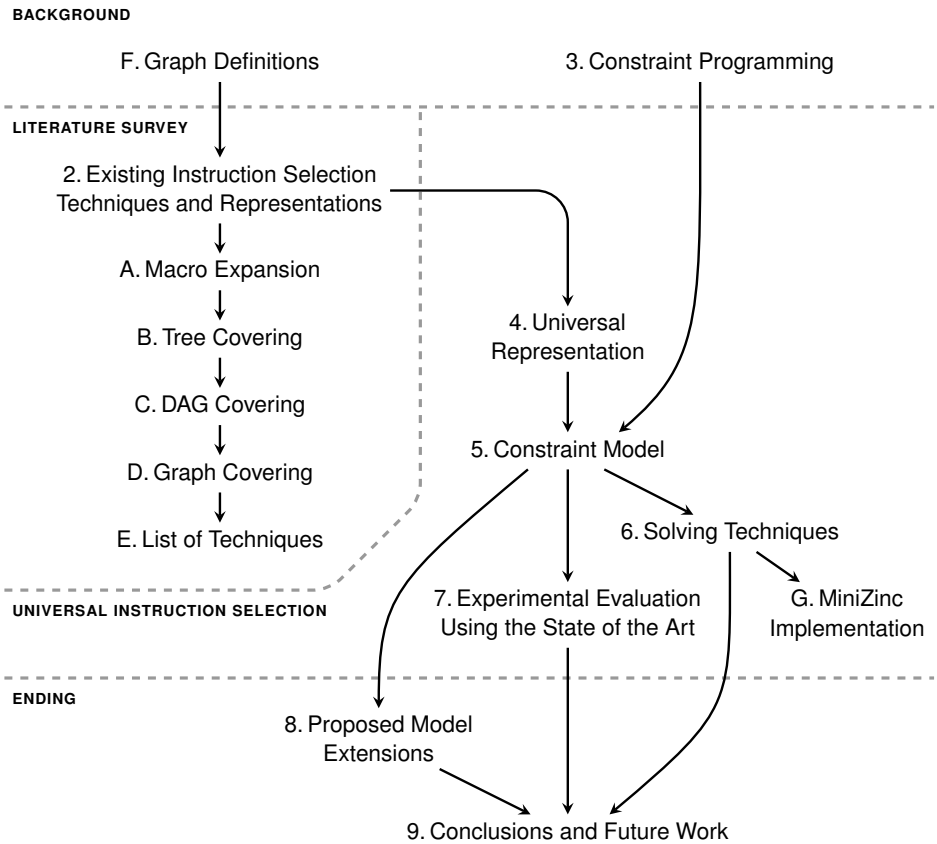


Figure 1.6: Structure of the dissertation.

Universal instruction selection Presents the approach: Chap. 4 introduces the universal representation; Chap. 5 introduces the constraint model; Chap. 6 introduces the solving techniques; Chap. 7 evaluates the approach using the state of the art; and Ap. G provides an implementation of the constraint model, written in MiniZinc.

Ending Closes the dissertation: Chap. 8 proposes model extensions; and Chap. 9 presents conclusions and future work.

CHAPTER

2

Existing Instruction Selection Techniques and Representations

With the first publications beginning to appear at the end of the 1960s, instruction selection has been actively researched for over four decades. When surveying these techniques, it was discovered that essentially all apply one of four fundamental *principles* of instruction selection: macro expansion, tree covering, DAG covering,[1] and graph covering. The trend of applying these principles over time is shown in Fig. 2.1. It was also discovered that the capabilities of these techniques can be compared in terms of handling instructions with five *characteristics*. The approaches can thus be systematically classified according to their principle and supported instruction characteristics.

Since a full survey is not needed in order to understand universal instruction selection – which apply graph covering – we examine in this chapter only the techniques most relevant to universal instruction selection. Section 2.1 introduces the instruction characteristics, and Sects. 2.2, 2.3, 2.4, and 2.5 discusses macro expansion, tree covering, DAG covering, and graph covering, respectively. The remaining techniques are discussed in Aps. A–D, and the full survey is also available in [186]. Lastly, Sect. 2.6 discusses the limitations of these techniques.

Without loss of generality, we henceforth assume that the input to the instruction selector consists of a single function, which in turn consists of many basic blocks (henceforth referred to as simply *blocks*). Of these blocks exactly one represents the function's point of entry, called the *entry block*. Instruction selection can then be reduced into two subproblems:

1. Finding all instances of instructions that can implement one or more operations in the function. This problem is called the *matching problem*.
2. Selecting a subset of these instances such that all operations are implemented. This problem is called the *selection problem*.

[1] DAG stands for *directed acyclic graph*.

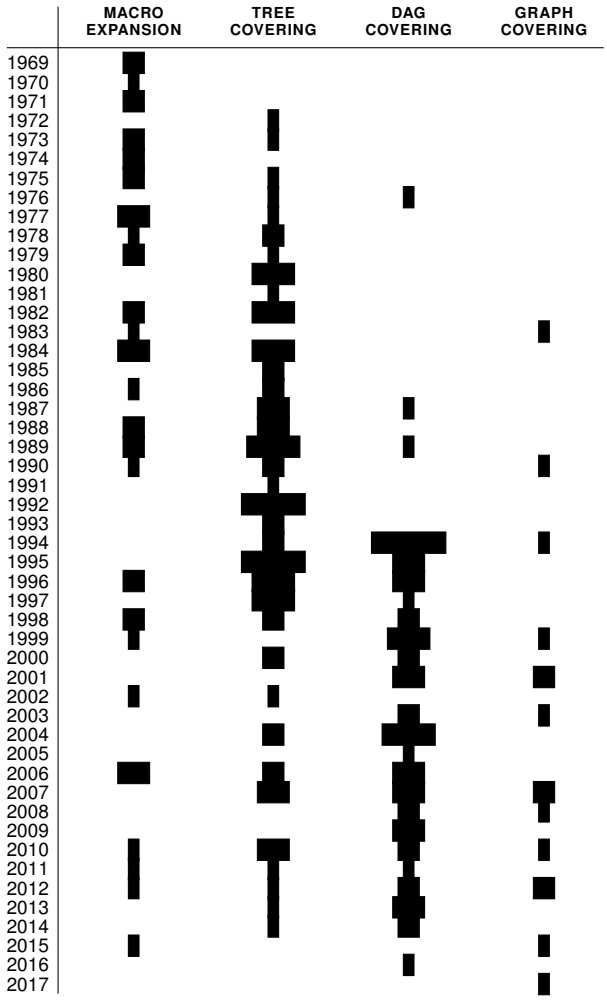


Figure 2.1: Diagram showing how research on instruction selection with respect to the fundamental principles has progressed over time. With 197 publications in total, the width of each bar indicates the number of relative publications for a given year (▮ represents one publication).

Unless the second subproblem is constructed such that it impacts the first – and the author has yet to come across a situation where this is required – both subproblems can be solved in isolation without compromising code quality.

2.1 Instruction Characteristics

An instruction can be said to exhibit five characteristics: single-output, multi-output, disjoint-output, inter-block, and interdependent. The first three characteristics form sets of instructions that are disjoint from one another, whereas the last two characteristics can be combined as appropriate with each other and with any of the other characteristics.

Single-Output Instructions The simplest kind of instruction forms the set of *single-output instructions*. These produce only a single observable output, in the sense that "observable" means a value that can be accessed through the assembly code. This includes all instructions that implement a single operation (such as addition, multiplication, and bit operations), but it also includes more complicated instructions that implement several operations (such as memory operations with complicated addressing modes). As long as the observable output constitutes a single value, a single-output instruction can be arbitrarily complex.

This class comprises the majority of instructions in most instruction sets, and in simple *reduced instruction-set computers (RISCs)*, such as MIPS architectures, nearly all instructions are single-output instructions. Naturally, all instruction selectors are expected to support this kind of instruction.

Multi-Output Instructions As expected from their name, *multi-output instructions* produce more than one observable output from the same input. Examples include `divmod` instructions, which compute both the quotient and the remainder of two input values, as well as arithmetic instructions that, in addition to computing the result, also set one or more status flags.[2] For this reason such instructions are often said to have side effects, but in reality these bits are nothing else but additional output values produced by the instruction, and will thus be referred to as multi-output instructions. Memory load and store instructions, which access a memory value and then increment the address pointer, are also considered multi-output instructions.

Many architectures such as X86, ARM, and Hexagon provide instructions of this class, although they are typically not as common as single-output instructions.

Disjoint-Output Instructions Instructions producing many observable output values from many different input values are called *disjoint-output instructions*. These

[2] A *status flag* (sometimes also known as a *condition flag* or a *condition code*) is a single bit that signifies additional information about the result of a computation, for example if there was a carry overflow or the result was equal to 0.

are similar to multi-output instructions with the exception that all output values in the latter originate from the same input values. Another way to put it is that if one formed the patterns that correspond to each output – this will be explained in Sect. 2.3 – then all these patterns would be disjoint from one another. This typically includes SIMD instructions, which execute the same operations simultaneously on many distinct input values.

Disjoint-output instructions are common in high-throughput graphics architectures and DSPs, but also appear in X86 as extensions under names like *streaming SIMD extensions (SSE)* and *advanced vector extensions (AVX)* [200]. Recently, certain ARM processors are also equipped with such extensions [21].

Inter-Block Instructions Instructions whose behavior essentially spreads across multiple blocks are called *inter-block instructions*. Examples of such instructions are those implementing saturation arithmetic[3] and hardware loop instructions, which repeat a fixed sequence of operations a certain number of times.

Instructions with this characteristic typically appear in customized architectures and DSPs such as ARM's Cortex-M7 [20] and TI's TMS320C55x [353]. But because of their complexity, capturing the behavior of these instructions require sophisticated techniques that are currently not available for most compilers. Instead, individual instructions are supported either via customized program optimization routines or through compiler intrinsics. If no such routine or compiler intrinsic is available, making use of these instructions requires the program to be written directly in assembly code.

Interdependent Instructions The last class is the set of *interdependent instructions*. This includes instructions exhibiting additional constraints that appear when they are combined with other instructions in certain ways. An example includes an `add` instruction, again from the TMS320C55x instruction set, which cannot be combined with an `rpt` instruction if a particular addressing mode is used for the `add` instruction.

Interdependent instructions are rare and can typically be found in complex, heterogeneous architectures such as DSPs. This is another class of instructions that most instruction selectors struggle with, mainly because these instructions typically violate the assumptions made by the underlying instruction selection techniques.

2.2 Macro Expansion

The first principle to emerge was *macro expansion*, with applications starting to appear in the 1960s. In macro expansion, the instructions are expressed as *macros* which consist of two parts: a *template* to be matched over the function under

[3] Recently, a request was made to extend the LLVM compiler with *compiler intrinsics* – a kind of special IR operations – to facilitate selection of such instructions [49].

```
expand($3 ← $1 + $2) {
  r1 = getRegOf($1);
  r2 = getRegOf($2);
  r3 = mkNewReg($3);
  print "add " + r3 + ", " + r1 + ", " + r2;
}
```

Figure 2.2: Example of a macro expanding an IR addition into assembly code. The template to match is given as argument to `expand`, and the procedure to run upon expansion is given as `expand`'s body.

compilation, and an *expand procedure* to be executed upon the part of the function that was matched. An example of such a procedure is given in Fig. 2.2. A *macro expander* traverses the function and tries to match the templates of the macros, one after another. Upon a match it executes the corresponding expand procedure and then resumes the traversal with the next, unmatched part until the entire function has been expanded. Consequently, matching and selection is combined into a single task as the first macro matched is also the selected macro.

The main benefit of macro expansion is that it is intuitive and straightforward to apply. Because the macro expander is implemented separately from the macros, the former can be kept generic and simple while the latter can be customized as needed for the target machine. This also allows the macro expander to be void of any target-specific details, thus requiring only the macros to be rewritten when retargeting the compiler to another target. To this end, the macros are typically written in some dedicated language to simplify the retargeting task.

But with its simplicity comes two shortcomings. Depending on the complexity of the macros, macro expansion could in principle support all kinds of instructions. In practice, however, macro-expanding instruction selectors are typically limited to single-output instructions. In addition, they often only expand one IR operation at at time, resulting in poor code quality. Another disadvantage is that because of idiosyncrasies of the dedicated language, the macros are often hard to read and understand, making them difficult to extend and maintain. We call this variant of the principle *naive macro expansion*.

To mitigate these problems, macro expansion can be combined with peephole optimization.[4] First, a naive macro expander expands the function under compilation one IR operation at at time. Once fully expanded, a peephole optimizer runs over the result and replaces inefficient sequences of instructions with more competent equivalences. Consequently, the macros can be divided up into those required for correctness – that is, the simple, single-operation macros, ensuring that code can always be produced – and those used for efficiency. Consequently, the retargeting

[4] A *peephole optimizer* is a program that combines two or more adjacent instructions into a single instruction. The term *peephole* come from the narrow window of operation, as a peephole optimizer only considers a small number of instructions at a time.

task becomes simpler and more incremental. Due to the people who pioneered the idea, this scheme is known as the *Davidson-Fraser approach* [93].

Because this principle has little relevance for universal instruction selection, we will not examine applications of naive macro expansion and the Davidson-Fraser approach in this chapter.

2.3 Tree Covering

Beginning of 1970s, a principle called *tree covering* began to emerge. Unlike macro expansion, tree covering approaches instruction selection as a graph problem and, in doing so, separates the selection problem from the matching problem. This gives several advantages over macro expansion. First, capturing the behavior of instructions becomes simpler. Second, trade-offs in selecting certain combinations of matches can be considered, which improves code quality. Third, due to machine grammars (to be described shortly), tree covering can be based on a formal foundation that enables proof of completeness.

2.3.1 Instruction Selection as a Graph Problem

First, the IR code is transformed into a *data-flow graph*, where nodes represent operations and edges represent data dependencies between the operations. Data-flow graphs built from the function under compilation are called *expression trees* if they are limited to single, tree-shaped expressions, *block DAGs* if they capture many expressions in a block as a DAG, and *function graphs* if they capture the data flow of entire functions. An instruction selector is *local* if it selects instructions for expression trees or block DAGs, and *global* if it does so for function graphs.

Corresponding data-flow graphs are also built to represent the instructions provided by the target machine. Such data-flow graphs are called either *pattern trees*, *pattern DAGs*, or *pattern graphs*, depending on whether they are shaped as trees, DAGs, or graphs, respectively. When the shape is clear from the context, they are simply called *patterns*. The set of patterns for a particular target machine constitute a *pattern set*.

The matching problem can be reduced to finding all instances where a pattern from the pattern set is subgraph isomorphic to G, where G denotes a data-flow graph derived from a function. Each such instance is called a *match*, and the set of all matches constitute a *match set*, denoted by M. Hence, in this context the matching problem is referred to as *pattern matching*. Pattern matching can be done in linear time if both G and all patterns are tree-shaped, otherwise it is an NP-complete problem [154, 193].

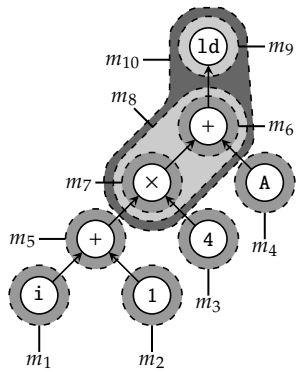
Having found M, the selection problem – which in this context is referred to as *pattern selection* – can be reduced to selecting a set of matches that covers G. A subset $M' \subseteq M$ *covers* G if every node in G appears in at least one match in M'. Such a subset is called a *cover*. A cover is an *exact cover* if every node in G appears in

```
x = A[i + 1];
```

(a) C code.

`mv`	$r \leftarrow var$
`add`	$r \leftarrow s + t$
`mul`	$r \leftarrow s \times t$
`muladd`	$r \leftarrow s \times t + u$
`load`	$r \leftarrow *s$
`maload`	$r \leftarrow *(s \times t + u)$

(b) Instructions. The $*s$ notation means "get value at address s in memory".



(c) Expression tree and its matches.

Figure 2.3: Example demonstrating the pattern matching and selection problem for a program that loads a value from integer array `A` at offset `i + 1`. It is assumed that `i` is stored in register, that `A` is stored in memory, and that an integer is four bytes. Exact covers are $\{m_1, \ldots, m_7, m_9\}$, $\{m_1, \ldots, m_5, m_8, m_9\}$, $\{m_1, \ldots, m_5, m_{10}\}$, and $\{m_1, \ldots, m_5, m_8, m_9\}$ (non-exact covers are ignored for brevity). Variable assignments need not be explicitly represented as nodes since this information can be propagated from the root node after having found a cover.

exactly one match in the cover. Most instruction selection approaches assume exact coverage. Examples of covers are shown in Fig. 2.3.

For a given function and target machine, there often exists many valid combinations of instructions. In terms of G and M this means there exist many covers of G, which each may result in code where quality differs significantly. In certain cases, for example, the performance of two sets of selected instructions may differ by as much as two orders of magnitude [385]. Consequently, the pattern selection problem – originally defined to accept any valid cover – is augmented into an optimization problem called *optimal pattern selection*, where only covers with least cover are accepted. The cost of a cover M' is the sum of the costs for the matches appearing in M', where the cost of a match is set to reflect a desired characteristic in the generated code.

For example, assume that the `mv`, `add`, `mul`, and `muladd` instructions in Fig. 2.3 all take one cycle to execute whereas the `load` and `amload` instructions take five cycles to execute. Assume further that the compiler should maximize performance. The corresponding matches $m_1, m_2, \ldots, m_8, m_9, m_{10}$ are therefore assigned costs $0, 1, \ldots, 1, 5, 5$, respectively (m_1 has zero cost since variable `i` is already in a register). Then, of the exact covers $\{m_1, \ldots, m_7, m_9\}$, $\{m_1, \ldots, m_5, m_8, m_9\}$, and $\{m_1, \ldots, m_5, m_{10}\}$, only the last cover is considered optimal as it has a total cost

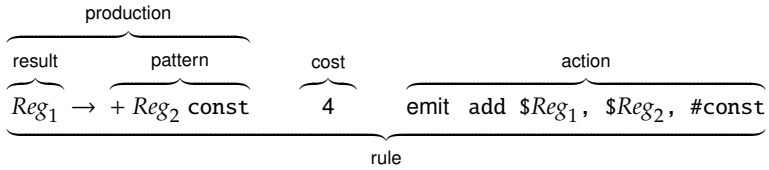


Figure 2.4: Anatomy of a rule in a machine grammar.

of 9 whereas the other two covers have costs 11 and 10, respectively. There is typically a strong correlation between the size of a cover and its cost – smaller covers typically lead to less cost and ultimately better code – but this depends heavily on the properties of the target machine.

For practical reasons, the instruction set of a target machine must be described in a machine-readable format, which we call *machine description*. A common method is to model the instructions as a machine grammar, which we will now describe.

2.3.2 Machine Grammars

Machine grammars (or simply called *grammar*) are based on context-free grammars [9], which are typically used for describing language syntax. A grammar consists of terminals, nonterminals, and rules. In this context, a *terminal* is a symbol representing an operation (e.g. +, <, `load`), and a *nonterminal* is a symbol representing an abstract result (e.g. *Reg*) produced by the instruction. To distinguish between the two, terminals are written entirely in lower case whereas nonterminals start with a capital letter and are set in italics.

A *rule* describes the behavior of an instruction and consists of a production, a non-negative cost, and an action. *Productions* describe how to derive nonterminals, and are written as

$$\alpha \rightarrow \beta\gamma \ldots$$

where the left-hand side is a single nonterminal and the right-hand side is a sequence of terminals and nonterminals. Each instruction gives rise to one or more productions, where the right-hand side of a production captures a pattern of the instruction and the left-hand side denotes the result produced by the instruction. Hence the left-hand and right-hand sides of a rule are referred to as the rule's *result* and *pattern*, respectively. The *action* is what to perform when a rule is selected. Typically this is to emit the corresponding assembly code, but it could also include other tasks such as bookkeeping. The rule structure is illustrated in Fig. 2.4, and examples of rules are given in Tab. 2.1.

To avoid the need for parentheses, the productions are typically written in *Polish notation* (for example, 1 + (2 + 3) is written as + 1 + 2 3). Similarly, the expression tree shown in Fig. 2.3c can be expressed as `load + × + i 1 4 A`.

#	production	cost	action
1	$Reg_1 \rightarrow$ `load` + Reg_2 `const`	1	emit `load $`Reg_1`, const(`$`$Reg_2$`)`
2	$Reg_1 \rightarrow$ `load` + `const` Reg_2	1	emit `load $`Reg_1`, const(`$`$Reg_2$`)`
3	$Reg_1 \rightarrow$ `load` Reg_2	1	emit `load $`Reg_1`, 0(`$`$Reg_2$`)`

Table 2.1: Example of grammar rules corresponding to a `load $t, o($s)` instruction that loads a value from memory at the address given in register `s`, offset by an immediate value `o`, and stores the loaded value in register `t`, in one cycle. The subscripts are only needed for referencing the right nonterminal in the action.

#	production	cost	action
1	$Reg \rightarrow$ `load` A	1	emit `load $`Reg`,` $A.C$`.const(`$`$A.Reg$`)`
4	$A \rightarrow +\ Reg\ C$	0	
2	$Reg \rightarrow$ `load` B	1	emit `load $`Reg`,` $B.C$`.const(`$`$B.Reg$`)`
5	$B \rightarrow +\ C\ Reg$	0	
6	$C \rightarrow$ `const`	0	
3	$Reg_1 \rightarrow$ `load` Reg_2	1	emit `load $`Reg_1`, 0(`$`$Reg_2$`)`

Table 2.2: The grammar from Tab. 2.1 in normal form. Nonterminals A, B and C and rules 4–6 are introduced in order to transform rules 1 and 2 into base rules.

With a grammar at hand, the pattern selection problem becomes equivalent to finding a sequence of rule applications, called *rule reductions*, that reduces the expression tree to a given nonterminal. A method for finding the sequence with least cost is described shortly.

Normal Form To simplify pattern matching and pattern selection, a grammar can be rewritten into normal form [32]. A grammar is in *normal form* if every rule in the grammar has a production in one of the following forms:

1. $N \rightarrow$ `op` $A_1\ A_2 \ldots A_n$, where `op` is a terminal, representing an operation that takes n arguments, and all A_i are nonterminals. Such rules are called *base rules*.
2. $N \rightarrow$ `t`, where `t` is a terminal. Such rules are also called base rules.
3. $N \rightarrow A$, where A is a nonterminal. Such rules are called *chain rules*.

A grammar can be mechanically rewritten into normal form by introducing new nonterminals and breaking down illegal rules into multiple, smaller rules until the grammar is in normal form. For example, rewriting the grammar shown in Tab. 2.1 into normal form results in the grammar shown in Tab. 2.2. Note that the new rules have zero cost and no action as these are only intermediary steps towards enabling reduction of the original rule.

Since all productions in a grammar have at most one terminal, the pattern matching problem becomes trivial (simply match the node type against the terminal in all base rules). Otherwise another bottom-up traversal of the expression tree would have to be made in order to find all matches, which can be done in linear time for most reasonable grammars [193]. As we will see, this also simplifies pattern selection as the patterns on the right-hand side in all productions have uniform height.

2.3.3 Optimal Pattern Selection on Expression Trees

Aho et al. [8] introduced a method for finding the optimal cover for any given expression tree in linear time, which is also the most common and well known technique based on tree covering.

The technique is centered around the following assumption. Given a node n in an expression tree and a rule r, the cost of applying r on n is the cost of r plus the costs of reducing all children of n to the appropriate nonterminals appearing on the right-hand side of r. If r is a chain rule then the cost is computed as the cost of r plus the cost of reducing n to the nonterminal appearing in the pattern of r. The recursive nature of these costs can be exploited using dynamic programming, resulting in the algorithm shown in Alg. 2.1 which computes the least cost of reducing a given expression tree to a particular nonterminal.

The algorithm works as follows. It first constructs a cost matrix C, where rows represent nodes in the expression tree and columns represent nonterminals in the grammar, assumed to be in normal form.[5] The cost in each element $C[i][j]$ is initialized to infinity, indicating that there exists no sequence of rule reductions that reduces node i to nonterminal j. It then computes the costs by traversing the expression tree bottom up. At each node n and for each matching base rule r, with nonterminal s as result, it computes the cost c of applying r at n to produce s according to the scheme stated above. If c is less than the currently recorded cost for reducing n to s, then the cost and rule information for n is updated accordingly. The same is then done for all chain rules until it reaches a fixpoint (which must eventually be reached as all rule costs are non-negative and an update only occurs when the cost is strictly less). Since every node is also only processed once, the algorithm runs in linear time with respect to the size of the expression tree. Having computed the costs, the optimal order of rule reductions – which is equivalent to the least-cost cover – can be found using the algorithm shown in Alg. 2.2.

In many cases this technique produces code of sufficient quality. In fact, for architectures with simple instruction sets, where the rule patterns can naturally be modeled as trees (such as single-output instructions), it is often optimal or near optimal. The MIPS instruction set [347], for example, is one such architecture.

[5] The algorithm can be adapted to accept any grammar by expanding the FindMatchingRules and ComputeReductionCost functions to handle rules of arbitrary form.

```
function ComputeCosts(expression tree T, normal-form grammar G):
    S ← {s | s is a nonterminal in G}
    C ← matrix of size |T| × |S|, costs initialized to ∞
    ComputeCostsRec(root node of T)
    return C

function ComputeCostsRec(node n):
    foreach child m of n do
        ComputeCostsRec(m)
    foreach base rule r ∈ FindMatchingRules(n) do
        c ← ComputeReductionCost(n, r)
        l ← result of r
        if c < C[n][l].cost then
            C[n][l].cost ← c
            C[n][l].rule ← r
    repeat
        foreach chain rule r ∈ G do
            c ← ComputeReductionCost(n, r)
            l ← result of r
            if c < C[n][l].cost then
                C[n][l].cost ← c
                C[n][l].rule ← r
    until no change to C

function FindMatchingRules(node n):
    M ← ∅
    foreach base rule r ∈ G do
        if terminal in pattern of r = node type of n then
            M ← M ∪ {r}
    return M

function ComputeReductionCost(node n, rule r):
    c ← cost of r
    if r is a chain rule then
        s ← nonterminal in pattern of r
        c ← c + C[n][s].cost      // here cost of node itself is taken instead of its children
    else
        for i ← 1 to number of children for n do
            m ← ith child of n
            s ← ith nonterminal in pattern of r
            c ← c + C[m][s].cost
    return c
```

Algorithm 2.1: Computes the optimal sequence of rules that reduces the given expression tree to a particular nonterminal.

```
   function SelectRules (expression tree T, goal nonterminal g, cost matrix C):
1  |  n ← root node of T
2  |  r ← C[n][g].rule
3  |  if r is a chain rule then
4  |  |  s ← result of r
5  |  |  SelectRules (T, s, C)
6  |  else
7  |  |  for i ← 1 to number of children for n do
8  |  |  |  m ← ith child of n
9  |  |  |  s ← ith nonterminal in pattern of r
10 |  |  |  SelectRules (expression tree rooted at m, s, C)
11 |  |  execute actions associated with r
```

Algorithm 2.2: Selects optimal sequence of rules that reduces a given expression tree to a given nonterminal, based on costs computed by Alg. 2.1.

2.3.4 Precomputing Costs and Rule Decisions

Shortly after Aho et al. published their approach, it was recognized that the costs computed by Alg. 2.1 could be precomputed for any expression trees. Hence the ComputeReductionCost call in Alg. 2.2 can be replaced by a constant-time table lookup. Although this improvement does not affect the asymptotic time complexity of the algorithm, it still significantly reduce the actual runtime. Although pioneered by Hatcher and Christopher [179], the idea was first successfully applied by Pelegrı-Llopart and Graham [298], which was later improved and simplified by Balachandran et al. [32] and Proebsting [306].

The idea is as follows. The expression tree is traversed bottom up and labeled with a *state*. For a given labeled node n, the state essentially holds enough information to optimally reduce the expression tree rooted at n to any nonterminal. At first glance it would seem that this requires an infinite number of states as expression trees can be of arbitrary size, but this only applies if the *full* cost of the entire expression tree is taken into account. When considering the *relative* cost differences between rules for any given node in the expression tree, it is sufficient with a finite number of states. The algorithm for labeling an expression tree is shown in Alg. 2.3. To derive the rule selection algorithm, one only needs to adapt line 2 in Alg. 2.2 to perform the appropriate table lookups.

The idea for computing the states – which will only be described briefly – works as follows. For each terminal representing a k-argument operation, an k-dimensional matrix is maintained. This is called the terminal's *state table*, which indicates the state to assign such nodes given the labels of its children. First the states for all leaf nodes are built, considering only base rules with a single terminals on the right-hand side in the production. The costs and rule decisions are computed

```
   function LabelTree (expression tree T, list L of state tables):
1  |  n ← root node of T
2  |  k ← number of children for n
3  |  for i ← 1 to k do
4  |  |  m_i ← ith child of n
5  |  |  LabelTree (expression tree rooted at m_i, L)
6  |  S ← L[terminal corresponding to n]
7  |  n.label ← S[m_1.label, ..., m_k.label]
```

Algorithm 2.3: Labels an expression tree for optimal pattern selection.

using the same logic as in Alg. 2.1, lines 8–21. The leaf state are then pushed onto a queue. Each popped state is used as the ith child to all base rules with a non-leaf terminals in combination with all other existing states. If any combination gives rise to a new set of costs and rule decisions, then a new state is created and pushed onto the queue after having updated the state tables. This process continues until the queue is empty, whereupon all necessary states have been built.

2.3.5 Limitations of Tree Covering

The main disadvantage of operating on expression trees is that common subexpressions have to be either split along the edges or duplicated when building the IR. These transformations are referred to as *edge splitting* and *node duplication*, respectively. Depending on the instruction set, these decisions can prevent selection of instructions that would lead to better code quality.

An example illustrating this effect is shown in Fig. 2.5. If the IR is represented as trees, where the common subexpression for computing `t` has its own expression tree (Fig. 2.5c), then matches $m_1, \ldots, m_7$, and m_9 must be selected, which results in a total cost of $0 + \cdots + 0 + 2 + 3 + 5 = 10$. If represented as a block DAG (Fig. 2.5d), then it becomes possible of selecting matches m_8 and m_{10}, resulting in a total cost of $0 + \cdots + 0 + 4 + 5 = 9$. Duplicating the nodes of the common subexpression would, in this case, yield the same covers, but would have resulted in suboptimal code in cases where the `addmul` and `addload` instructions are not available.

2.4 DAG Covering

By replacing the expression trees used in tree covering with block DAGs, and allowing instructions to be modeled either as pattern trees or pattern DAGs, we attain the more general principle called *DAG covering*.

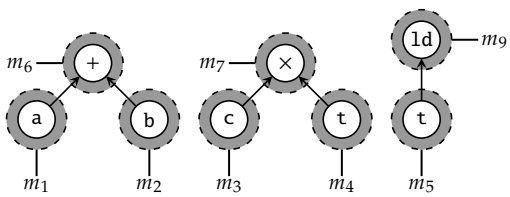
DAG covering has several advantages of tree covering. First, the block DAG does not need to be decomposed into expression trees, which compromises code

```
t = a + b;
x = c * t;
y = *((int*) t);
```

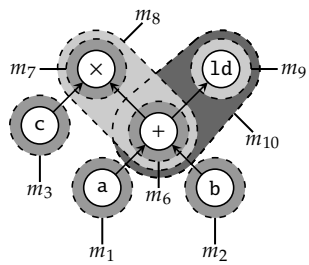
(a) C code.

instruction	cost
`add` $r \leftarrow s + t$	2
`mul` $r \leftarrow s \times t$	3
`addmul` $r \leftarrow (s + t) \times u$	4
`load` $r \leftarrow *s$	5
`addload` $r \leftarrow *(s + t)$	5

(b) Instructions. The $*s$ notation means "get value at address s in memory".



(c) Expression trees after edge splitting.



(d) Block DAG.

Figure 2.5: Example illustrating that using block DAGs results in better code compared to using expression trees. It is assumed variables `a`, `b`, `c`, and `t` are stored in registers.

quality. Second, it supports use of pattern DAGs, which are needed for modeling multi-output instructions.

But unlike tree covering, which can be solved optimally in linear time, finding the least-cost cover of a block DAG is NP-complete [224, 305]. The proof is also available in Ap. C on p. 206. Consequently, DAG covering started to gain traction in the beginning of the 1990s after exponential increase in computing power and significant progress made in the field of combinatorial optimization enabled such methods to become practical for instruction selection. Most combinatorial approaches support DAG-shaped patterns, but finding matches for such patterns can no longer be done in linear time. Such approaches therefore typically apply generic subgraph-isomorphism algorithms when pattern matching.

2.4.1 Pattern Matching as a Subgraph Isomorphism Problem

The *subgraph isomorphism problem* is to find instances where a graph $G_P = (N_P, E_P)$ is isomorphic to a subgraph in another graph $G_F = (N_F, E_F)$. G_P is *isomorphic* to G_F if and only if there exists a mapping $m : N_P \rightarrow N_F$ such that $(n, o) \in E_P$ implies $(f(n), f(o)) \in E_F$. In the context of instruction selection, G_P denotes a pattern, G_F

denotes a graph derived from a function, and m denotes a match.

The subgraph isomorphism problem, which is known to be NP-complete [85], appears in many other fields. Consequently, much research has been devoted to this problem (see for example [86, 123, 124, 146, 172, 185, 228, 272, 342, 354]). Due to its simplicity, however, most combinatorial approaches apply the VF2 algorithm, which is described in Sect. 2.5.4.

2.4.2 Maximum Munch

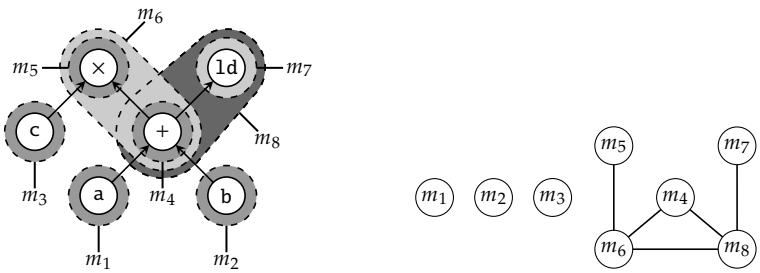
The most common approach for greedy pattern selection on block DAGs is called *maximum munch* (coined by Cattell [69]). The idea is to traverse the block DAG top down, select the largest pattern that matches the current node, and repeat the process for remaining, uncovered parts of the block DAG. The approach – which is used in for example LLVM [235] – works well for architectures with a regular instruction set and where there is a strong correlation between the effectiveness of the instruction and the size of its pattern. In addition to being non-optimal, however, it also suffers from the same drawback as expression trees regarding whether to select covers that effectively split or duplicate the common subexpressions.

2.4.3 Approaches for Balancing Splitting and Duplication

Several approaches have been made in attempting to balance edge splitting and node duplication. Fauth et al. [129] designed an heuristic algorithm that rewrites the block DAG into expression trees before instruction selection. Using a rough estimate of the cover cost, the algorithm first favors node duplication and resorts to edge splitting when the former becomes too costly. Once rewritten, the expression trees are covered using an improved variant of Alg. 2.1.

Ertl [121] showed that, for certain grammars, Alg. 2.1 can be adapted to produce optimal covers for block DAGs. The idea is to first compute the costs for each node as if the block DAG had been rewritten into a expression tree using node duplication. Then, if several rules reduce the same node to the same nonterminal N, then N can be shared between the rules whose patterns all contain N. Ertl also introduced an algorithm for checking whether optimal pattern selection is guaranteed for a given grammar.

Koes and Goldstein [224] combined the ideas by Fauth et al. and Ertl by introducing a design that first uses Alg. 2.1 to compute the costs for a duplicated expression tree. Then, at each node n where several patterns in the optimal cover overlap in the block DAG, two costs are estimated: the cost incurred by allowing overlap, and the cost incurred by splitting the edges. If the latter is cheaper then n is marked as *fixed*, meaning it can only be covered by patterns where n is their root node. Once all such nodes have been processed, another pass is performed to recompute the costs, this time forbidding overlap at fixed nodes.



(a) Block DAG with matches. (b) Interference graph.

Figure 2.6: Example of modeling pattern selection as a MIS problem. Maximal independent sets of the interference graph are $\{m_1, \ldots, m_5, m_7\}$, $\{m_1, m_2, m_3, m_5, m_8\}$, and $\{m_1, m_2, m_3, m_6, m_7\}$, which correspond to exact covers of the block DAG.

2.4.4 MIS- and MWIS-based Approaches

Another approach to modeling instruction selection is to model it as a problem of finding independent sets. Given a graph $G = (N, E)$, a set $S \subseteq N$ is an *independent set* if no pairs of nodes $m, n \in S$ are adjacent in G. An independent set is called a *maximal independent set (MIS)* if no more nodes can be added and still be an independent set. If each node in the graph has a weight, then a *maximal/minimal weighted independent set (MWIS)* is a MIS that maximizes/minimizes $\sum_n weight(n)$. In general, finding a MIS or MWIS is NP-complete [154].

Modeling instruction selection as either a MIS or MWIS problem is done as follows. After pattern matching, an *interference graph* is constructed where a node represents a match and an edge represents overlapping between two matches. If all operations in the block DAG or function graph can be covered by at least one match, then a MIS of the interference graph corresponds to a cover. Likewise, a MWIS corresponds to a least-cost cover. An example is shown in Fig. 2.6.

Applications Scharwaechter et al. [329] appears to have pioneered the modeling of instruction selection as a MWIS problem, although the main contribution of their paper is the extension of machine grammars to handle multi-output instructions. The idea is to model such instructions using *complex rules*, which each consists of multiple productions – one for each result. In this dissertation, such productions and their patterns are called *proxy rules*[6] and *proxy patterns*, respectively, whereas rules with a single production and their patterns are called *simple rules* and *simple patterns*, respectively. The rule structure is also illustrated in Fig. 2.7.

The proxy patterns are matched individually together with the simple patterns. After pattern matching, matches derived from proxy patterns are then either combined – indicating use of the complex rule – or kept – indicating use of rules

[6] In the original paper, they are called *split rules*.

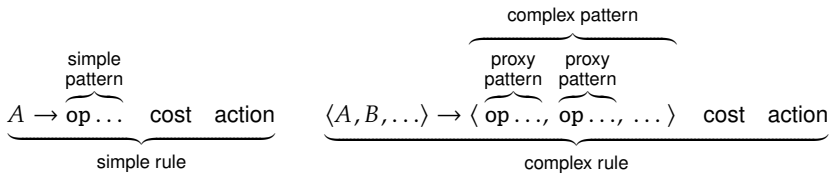


Figure 2.7: Anatomy of simple and complex rules in an extended machine grammar.

that only produce a single result. This is done according to a heuristic that estimates the cost saved by using the complex rule versus the cost incurred by having to duplicate nodes in common subexpressions. Once these decisions have been taken, the interference graph is built and the MWIS found using a greedy heuristic [325].

The approach was later extended by Ahn et al. [3] to include scheduling dependency conflicts between complex patterns in order to facilitate register allocation. In both designs, however, the complex rules can only consist of disconnected patterns, hence forbidding sharing of values across the proxy patterns. This shortcoming was addressed by Youn et al. [377] by introducing the use of index subscripts for nodes representing the input arguments.

2.4.5 IP-based Approaches

Several approaches model instruction selection using *integer programming (IP)* – often also referred to as *integer linear programming (ILP)* – which is a method for solving combinatorial optimization problems (see [370] for an overview). Formally, an IP problem is defined as follows.

Definition 2.1 – IP Let $\vec{c}$ and $\vec{b}$ be integer vectors, A be an integer matrix, and $\vec{\mathbf{x}}$ be a vector of integer decision variables. Then

$$\begin{aligned} \text{maximize or minimize} \quad & \vec{c}^{\mathsf{T}}\vec{\mathbf{x}} \\ \text{subject to} \quad & A\vec{\mathbf{x}} \leq \vec{b}, \\ & A\vec{\mathbf{x}} \in \mathbb{Z}^{n\times n}, \\ \text{and} \quad & \vec{\mathbf{x}} \in \mathbb{N}^n. \end{aligned}$$

Such problems are NP-complete in general, but extensive research in the field has made IP a practical tool for solving problems containing tens of thousands of variables.

In most IP-based approaches, pattern selection is modeled as

$$\forall n \in N : \sum_{\substack{m \in M \text{ s.t.} \\ n \in \mathit{covers}(m)}} \vec{\mathbf{x}}[m] \geq 1, \tag{2.1}$$

where N denotes the set of nodes in a block DAG, M denotes the match set, $covers(m)$ denotes the set of nodes covered by match m, and $\vec{\mathbf{x}}[m]$ is a Boolean ($\{0, 1\}$) *decision variable* indicating whether match m is selected. For exact coverage, the inequality is replaced with equality. When there is no risk of confusion, we refer to decision variables simply as *variables*.

Applications Although mostly known for their work in *integrated code generation* – meaning instruction selection, instruction scheduling and register allocation is solved in unison – Wilson et al. [368] also pioneered the use of IP for modeling instruction selection. Their design performs global instruction selection on a function graph that has been augmented with additional copy operations to represent potential register spills.[7] In most cases, not all copies are needed and therefore not all operations must be covered. Consequently, Wilson et al. model pattern selection as $\sum_m \vec{\mathbf{x}}[m] \leq 1$.

Another approach for integrated code generation was made by Bednarski and Kessler [38], whose design was later reused by Eriksson et al. [119, 120]. Unlike all other instruction selection approaches, Bednarski and Kessler combine pattern matching and pattern selection. Consequently, in addition to the variables that decide which matches are selected, their IP model also has variables that, for each match, maps nodes and edges in the block DAG to nodes and edges in a pattern. An upper bound on the number of needed matches is computed beforehand using a heuristic.

An IP-based approach for selecting multi-output instruction was introduced by Leupers and Marwedel [245, 249]. Each DAG-shaped pattern of a multi-output instruction is first decomposed into trees, which are used for covering an expression tree. After having found a least-cost cover using Alg. 2.3, the expression tree is collapsed into a tree of *super nodes*, where each super node represents a set of nodes in the expression tree covered by the same pattern. The problem is then to try to merge super nodes such that the combination can be implemented using a multi-output instruction. This is often called *instruction compaction*. As there is an abundance of overlap between such combinations, Leupers and Marwedel solve this problem using IP.

Leupers [243] later introduced another IP-based approach for selecting disjoint-output instructions like SIMD instructions. Like before, each DAG-shaped pattern of a SIMD instruction is first decomposed into trees, but now the trees are used for covering a block DAG that has been transformed into expression trees through edge splitting. The potential of using disjoint-output instructions can be increased by allowing them to cover nodes from multiple expression trees, but covering each tree individually often leads to suboptimal code. To address this problem, Leupers first extended a machine grammar with additional nonterminals to indicate whether a SIMD instruction is used in covering a particular node and then modified Alg. 2.3 to compute all optimal covers instead returning a single solution. Once all least-cost

[7] *Spilling* is the act of temporarily storing a register value to memory in order to free up the register.

covers have been found for all expression trees in a block, an IP model is built to decide how to make the best use of SIMD instructions for this block. Leupers's model was later extended by Tanaka et al. [348] to take data copying into account by extending the block DAG with additional copy nodes, which is needed for architectures with irregular instruction sets.

The last IP-based approach to be discussed is that of Gebotys [155], who applied to the theory of Horn clauses to code generation. A *Horn clause* is a disjunctive Boolean formula that contains at most one positive (non-negated) literal. IP models built using Horn clauses can be solved in linear time [194], and Gebotys exploited this fact in developing an IP model where Horn clauses are applied to model register allocation instruction compaction. Pattern selection, however, is still modeled as in Eq. 2.1.

2.4.6 CP-based Approaches

Constraint programming (CP) is another method for solving combinatorial optimization problems, which is discussed in detail in Chap. 3. Therefore, only a brief introduction will be given here.

Like IP, a model in CP consists of a set of variables, a set of constraints over the variables, and typically also an objective function to be either minimized or maximized. A crucial difference, however, is that constraints are not limited to linear equations. Instead, relations among multiple variables are modeled using global constraints, which simplifies modeling and improves solving.

Modeling Pattern Selection Using Global Constraints Pattern selection can be modeled using a global constraint called the *global cardinality constraint*. The constraint, referred to as GCC, constrains the number of variables assigned a particular value (which may also be a variable). Given a set $v_1, \ldots, v_k$ of values and two sets $\mathbf{x}_1, \ldots, \mathbf{x}_n$ and $\mathbf{c}_1, \ldots, \mathbf{c}_k$ of variables, the constraint holds if, for each $i = 1, \ldots, k$, exactly $\mathbf{c}_i$ variables in the set $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are assigned value v_i (see also Def. 3.2 on p. 47). For example, $\text{GCC}(\langle 5, \mathbf{c}_1 = 0\rangle, \langle 3, \mathbf{c}_2 = 1\rangle, \mathbf{x}_1 = 2, \mathbf{x}_2 = 3)$ holds because no $\mathbf{x}$ variable is assigned value 5 and exactly one $\mathbf{x}$ variable is assigned value 3. Similarly, $\text{GCC}(\langle 3, \mathbf{c}_1 \in \{0, 2\}\rangle, \mathbf{x}_1 = 2, \mathbf{x}_2 = 3)$ does not holds because either none or both $\mathbf{x}$ variables must be assigned value 3.

To model exact pattern selection using GCC, two new sets of variables are needed. Assume that N denotes the set of nodes to be covered, M denotes the match set, and $\mathit{covers}(m)$ denotes the set of nodes covered by match m. Then, variable $\mathbf{match}_n \in \{m \mid m \in M, n \in \mathit{covers}(m)\}$ decides which match covers node n, and variable $\mathbf{count}_m \in \{0, |\mathit{covers}(m)|\}$ decides how many nodes are covered by match m. Hence each match covers either no nodes or all nodes in its pattern. With these variables, pattern selection can be modeled as

$$\text{GCC}(\cup_{m \in M} \langle m, \mathbf{count}_m\rangle, \cup_{n \in N} \mathbf{match}_n), \tag{2.2}$$

which, according to Floch et al. [135], offers stronger propagation than Eq. 2.1 and thus reduces solving time.

Applications The use of CP-based instruction selection appears to have been pioneered by Bashford and Leupers [36]. To generate code for highly irregular DSPs, Bashford and Leupers used CP to model the interactions between instruction selection and the use of processor resources, such as functional units and registers. Consequently, the approach essentially integrates instruction selection with a form of register allocation. Glossing over the details, the approach works as follows. For each operation, a so-called *factorized register transfer (FRT)* is built which encodes the resource requirements for the operands and the result as well as the cost of every instruction that may be used to implement such operations. Taking a block DAG as input, the problem is to cover all nodes using FRTs such that all resource requirements are fulfilled. Special resources are available for instructions that cover multiple nodes, allowing adjacent nodes to be covered by the same instruction. Bashford and Leupers used CP to solve this problem.

Martin et al. [266, 267] developed a model that integrates instruction selection and instruction scheduling. Because they target *application-specific instruction set processors (ASIPs)*, the patterns are not predefined but must be found prior to instruction selection. This is known as the *instruction set extension (ISE) problem*, which has been widely researched (see for example [11, 17, 28, 29, 37, 44, 50, 56, 80, 198, 210, 278, 279, 285, 381], and see [147] for a survey). For this task Martin et al. applied a CP-based pattern-matching algorithm, described in [369]. Pattern selection is modeled as Eq. 2.1 in another model, which was later improved by Floch et al. [135] who replaced the pattern selection constraints with Eq. 2.2. The approach was also extended by Arslan and Kuchcinski [26] for targeting VLIW processors[8] with SIMD instructions. Selection of such instructions is done by first splitting the patterns into multiple tree-shaped patterns, which are matched individually, and then enforcing that two selected matches belonging to the same instruction must be scheduled in the same cycle.

Beg [40] developed a constraint model that, unlike the approaches above, only concerns instruction selection. Both pattern matching and pattern selection are integrated into the same constraint model, and Beg also applied Alg. 2.1 in computing an upper bound on the cost in order to improve solving.

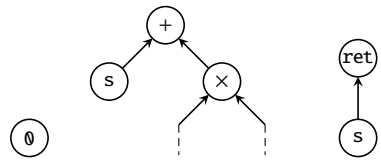
2.4.7 Limitations of DAG Covering

Although DAG coverings addresses the issue of whether to split or duplicate common subexpressions within a block, the problem still remains for expressions that are spread across multiple blocks. To fully address this problem, one must resort to graph covering.

[8] A *very long instruction word (VLIW) processor* is a processor that executes multiple instructions in parallel, where the schedule has been computed by the compiler and is part of the assembly code.

```
int f(int* A, int* B, int N) {
  int s = 0;
  for (int i = 0; i < N; i++) {
    s = s + A[i] * B[i];
  }
  return s;
}
```

(a) C code.



(b) Block graphs involving variable `s`. For brevity, the subtrees concerning `A[i]` and `B[i]` are not included.

rules	
$Reg \rightarrow$ `const`	$SReg \rightarrow \times\ Reg\ Reg$
$SReg \rightarrow$ `const`	$Null \rightarrow$ `ret` Reg
$Reg \rightarrow +\ Reg\ Reg$	$Reg \rightarrow SReg \quad (r \ll 1)$
$SReg \rightarrow +\ SReg\ SReg$	$SReg \rightarrow Reg \quad (r \gg 1)$

(c) Rules. *Null* is a dummy nonterminal since `ret` does not return anything, yet all productions must have a result. All rules are assumed to have equal cost. For brevity, the actions are not included.

Figure 2.8: Example illustrating the limitation of block DAGs.

This also applies to other situations where decisions made for one block can inhibit subsequent decisions for other blocks, such as enforcing specific storage locations or value modes. For example, Fig. 2.8 shows a function that multiplies the elements of two arrays and adds the results. Assume that the arrays consist of fixed-point values. For efficiency, a common idiosyncrasy in many DSPs is that multiplication of two fixed-point values return a value that is shifted one bit to the left. For such target machines, both the value `0` and the accumulator variable `s` should be in shifted mode throughout the entire function, and only restored into normal mode before returning from the function. Otherwise the accumulated value would be needlessly shifted back and forth within the loop. Achieving this, however, is difficult when limited to covering only a single block DAG at a time. Assume for example that the function had no multiplication. In that case, deciding to load value `0` in shifted mode would instead lower code quality as the value would needlessly have to be shifted back before returning, which takes an extra instruction.

Lastly, most of these approaches are restricted to tree-shaped patterns, meaning they only support single-output instructions. Many instruction sets, however, contain multi-output instructions which must be modeled as pattern DAGs.

2.5 Graph Covering

The most general principle is called *graph covering*, where entire functions are modeled as function graphs and instructions are allowed to be modeled using any shape of patterns. Unfortunately, compared to the other principles there exist relatively few applications of graph covering, the most well known appearing in the 2000s.

The main advantage is that graph covering support selection of inter-block instructions, whose behavior entail both control and data flow and must therefore be captured as pattern graphs. The representations typically used, however, only model data flow, thus stressing the need for new representations in order to handle the instructions of modern and forthcoming processors which are growing increasingly complex.

First we will look at a few representations for capturing entire functions as graphs. Because they result in relatively high nodes counts, such representations are colloquially referred to as *sea-of-nodes IRs*.

2.5.1 Sea-of-Nodes IRs

In the context of instruction selection, there are two sea-of-nodes IRs that are of interest. The first captures the data flow for entire functions, and the second is an extension of the first in order to also capture control flow.

Capturing Data Flow of Entire Functions In order to simply many compiler tasks, Cytron et al. [91] introduced a function representation called *static single assignment (SSA) form*.

A function is said to be in SSA form if every variable is defined exactly once. For example, the function shown in Fig. 2.9a is not in SSA form as variables `f` and `n` are redefined within the loop. By introducing new variables and connecting these using *φ-functions* where the value depends on control flow, the function can be rewritten into SSA form, as shown in Fig. 2.9b.

From an SSA-based function, we can construct a data-flow graph called the *SSA graph* [159]. Like in data-flow graphs, each operation in the function (including the φ-functions) is represented as a node. These nodes are connected using data-flow edges, ignoring the fact that the operations may belong to different blocks. For the example above, this results in the SSA graph shown in Fig. 2.9c. Since operations are not pre-assigned to specific blocks, the same IR can be used for performing global code motion.

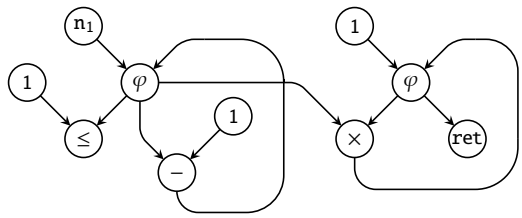
Capturing Both Data And Control Flow Click and Paleczny [81] introduced a sea-of-nodes IR that captures both data and control flow. The data flow is modeled exactly as in the SSA graph, and the control flow is captured using nodes to represent the blocks in the function and edges to represent jumps between blocks. To capture dependencies between the data and control flow – for example, when the target of

```
int factorial(int n) {
  entry:
    int f = 1;
  head:
    if (n <= 1) goto end;
  body:
    f = f * n;
    n = n - 1;
    goto head;
  end:
    return f;
}
```

(a) C implementation of factorial.

```
int factorial(int n1) {
  entry:
    int f1 = 1;
  head:
    int f2 = φ(f1:entry, f3:body);
    int n2 = φ(n1:entry, n3:body);
    if (n2 <= 1) goto end;
  body:
    int f3 = f2 * n2;
    int n3 = n2 - 1;
    goto head;
  end:
    return f2;
}
```

(b) Code in SSA form.



(c) SSA graph.

Figure 2.9: Example of an SSA graph.

a jump depends on a Boolean value – such jumps flow through special `if` nodes. For lack of a better name we will call this the *Click-Paleczny graph*, and an example is shown in Fig. 2.10.

2.5.2 Pattern Selection on the Click-Paleczny Graph

Paleczny et al. [295] introduced an approach for performing instruction selection based on the Click-Paleczny graph.

The approach first divides the function graph into a set of possibly overlapping expression trees. This is done by labeling certain nodes in the function graph as tree roots. Root candidates are nodes representing operations whose result are shared or operations with side effects and may therefore not be duplicated. The selection of roots is geared towards duplicating address computations and other expressions that can be subsumed into a single instruction. Once labeled, each expression tree is covered using a variant of Alg. 2.3. The instructions are then emitted and placed in blocks using a method described in [82].

Although the function is represented as a function graph, the instructions must still be modeled as pattern trees. Consequently, only single-output instructions can be selected using this approach.

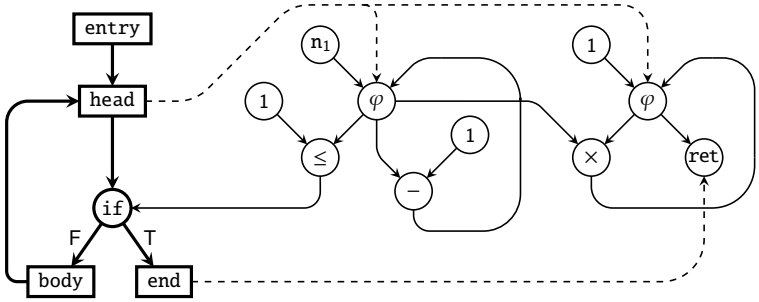


Figure 2.10: Example of a Click-Paleczny graph, corresponding to the function shown in Fig. 2.9. Thin-lined nodes and edges denote data operations and data flow. Thick-lined nodes and edges denote control operations and control flow. Dashed edges indicate to which block an operation belongs.

2.5.3 PBQP-based Approaches

Similar to IP-based approaches, another method of modeling instruction selection is to model it as a *partitioned Boolean quadratic problem (PBQP)*. First introduced by Scholz and Eckstein [330] to perform register allocation, PBQP is a variant of the *quadratic assignment problem (QAP)*, which is a fundamental combinatorial optimization problem in the field of operations research (see [254] for a survey). Although both problems are NP-complete in general, a subclass of PBQP can be solved in linear time which inspired Scholz and Eckstein in developing a greedy, linear-time solver.

Formally, a PBQP is defined as follows.

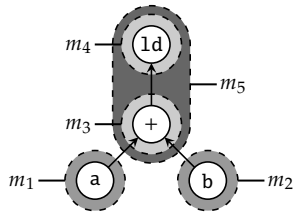
Definition 2.2 – PBQP Let $\vec{c}_1, \ldots, \vec{c}_n$ be integer vectors, and let $C_{1,1}, C_{1,2}, \ldots, C_{1,n}$, $C_{2,2}, \ldots, C_{n,n}$ be integer matrices of size $|c_i| \times |c_j|$ for $i = 1, \ldots, n$ and $j = i, \ldots, n$. Let also $\vec{\mathbf{x}}_1, \ldots, \vec{\mathbf{x}}_n$ be vectors of integer variables, where $\mathbf{x}_i \in \{0,1\}^{|c_i|}$ for $i = 1, \ldots, n$. Then

$$\begin{aligned} &\text{minimize} \sum_{1 \le i < j \le n} \vec{\mathbf{x}}_i^{\mathsf{T}} C_{ij} \vec{\mathbf{x}}_j + \sum_{1 \le i \le n} \vec{c}_i^{\mathsf{T}} \vec{\mathbf{x}}_i, \\ &\text{subject to} \quad \forall 1 \le i \le n : \vec{1}^{\mathsf{T}} \vec{\mathbf{x}}_i = 1. \end{aligned}$$

Intuitively, one can interpret this definition as follows. Assume that a problem consists of n decisions, each with k choices. Then $\vec{\mathbf{x}}_i$ is a decision variable with k elements, where $\vec{\mathbf{x}}_i[j] = 1$ means that choice j has been selected for decision i. For each variable, the condition $\vec{1}^{\mathsf{T}} \vec{\mathbf{x}}_i = 1$ ensures that exactly one choice is selected. The cost of selecting a particular choice for decision i is represented through a cost vector $\vec{c}_i$, and the cost of combining two decisions i and j are represented through a cost matrix C_{ij}.

rules	cost
$Reg \rightarrow \texttt{var}$	0
$Reg \rightarrow + \; Reg \; Reg$	1
$Reg \rightarrow \texttt{load} \; Addr$	3
$Reg \rightarrow \texttt{load} + \; Reg \; Reg$	5
$Addr \rightarrow Reg$	2

(a) Rules. For brevity, the actions are not included.



(b) SSA graph.

$$\vec{\mathbf{x}}_{\texttt{a}} \in \{0,1\} \qquad \vec{c}_{\texttt{a}} = \begin{bmatrix} 0 \end{bmatrix} \begin{matrix} m_1 \end{matrix} \qquad C_{\texttt{a}+} = \begin{matrix} m_3 & m_5 \end{matrix} \begin{bmatrix} 0 & 0 \end{bmatrix} \begin{matrix} m_1 \end{matrix}$$

$$\vec{\mathbf{x}}_{\texttt{b}} \in \{0,1\} \qquad \vec{c}_{\texttt{b}} = \begin{bmatrix} 0 \end{bmatrix} \begin{matrix} m_2 \end{matrix} \qquad C_{\texttt{b}+} = \begin{matrix} m_3 & m_5 \end{matrix} \begin{bmatrix} 0 & 0 \end{bmatrix} \begin{matrix} m_2 \end{matrix}$$

$$\vec{\mathbf{x}}_{+} \in \{0,1\}^2 \qquad \vec{c}_{+} = \begin{bmatrix} 1 \\ 5 \end{bmatrix} \begin{matrix} m_3 \\ m_5 \end{matrix}$$

$$\vec{\mathbf{x}}_{\texttt{load}} \in \{0,1\}^2 \qquad \vec{c}_{\texttt{load}} = \begin{bmatrix} 3 \\ 5 \end{bmatrix} \begin{matrix} m_4 \\ m_5 \end{matrix} \qquad C_{+\texttt{load}} = \begin{matrix} m_4 & m_5 \end{matrix} \begin{bmatrix} 2 & \infty \\ \infty & 0 \end{bmatrix} \begin{matrix} m_3 \\ m_5 \end{matrix}$$

(c) PBQP instance. The rows and columns in the cost vectors and matrices are labeled with the matches they represent. Cost matrices for uninteresting combinations are assumed to consist of 0s.

Figure 2.11: Example of modeling instruction selection as a PBQP.

In this context, $\vec{\mathbf{x}}_i$ decides whether to select a particular match to cover node i, $\vec{c}_i$ contains the cost for each such match, and C_{ij} contains the cost of additional instructions that may need to be selected due to certain combinations of matches. For example, assume two nodes i and j where j depends on i. Assume further that the instructions are represented as a normal-form machine grammar, and that i and j can be covered using two rules r_i and r_j, with productions $A \rightarrow \texttt{op}_i \; A \; A$ and $B \rightarrow \texttt{op}_j \; B \; B$, respectively. Since the result of r_i does not match the operands of r_j, this rule combination requires a chain rule – or a combination of these, if necessary – that derives B from A. Illegal combinations are prevented by assigning infinite cost. An example of a PBQP instance is shown in Fig. 2.11.

Handling Patterns DAG The PBQP model above assumes that all patterns are shaped as trees. To handle pattern DAGs, the model must be extended. First assume an extended grammar where multi-output instructions are described using complex rules (described on p. 30, see also Fig. 2.7). For each combination of matches derived from proxy rules that can be combined into an instance of a complex rule, a *complex match* is created. Each complex match i in turn introduces a variable $\vec{\mathbf{x}}_i \in \{0,1\}^2$ to decide whether i is selected. Because of the $\vec{1}^{\mathsf{T}}\vec{\mathbf{x}}_i = 1$ condition, every such variable has exactly two elements (one representing *on* and the other *off*). Like with the

simple rules, the costs of selecting a complex rule and interactions between these – for example, two complex matches are not allowed to overlap or cause cyclic data dependencies – are represented through the cost vectors and matrices.

In order to select a complex rule, all of its proxy rules must also be selected. This is achieved by first extending, for each node i, the domain of its variable $\vec{\mathbf{x}}_i$ with matches derived from proxy rules. Then a new set of cost matrices D_{ij} is created such that, for a node i and complex match j, the costs are 0 if $\vec{\mathbf{x}}_j = \mathit{off}$ or $\vec{\mathbf{x}}_i$ is set to a proxy rule associated with j. Otherwise the costs are ∞. Consequently, if a complex match covering some node n is selected, then the only choice for $\vec{\mathbf{x}}_n$ with non-infinite cost is an associated proxy rule. The PBQP model is thus augmented with another sum

$$\sum_{i \in N,\, j \in M_{\mathrm{C}}} \vec{\mathbf{x}}_i^{\mathsf{T}} D_{ij} \vec{\mathbf{x}}_j \tag{2.3}$$

where N denotes the set of nodes in the SSA graph and M_{C} denotes the set of complex matches.

This alone, however, allows solutions where all proxy rules but none of the complex rules are selected. This is resolved by assigning an artificially large cost K to the selection of proxy rules, which is offset when selecting the corresponding complex rule. For example, if a complex rule r with cost 2 consists of three proxy rules, then the new cost of selecting r is $2 - 3K$.

Applications Eckstein et al. [109] were first with modeling instruction selection as a PBQP, and Ebner et al. [108] extended their approach to support DAG-shaped patterns. Buchwald and Zwinkau [60] reused the PBQP model but replaced the use of machine grammars with rewrite rules based on algebraic graph transformations [258].

2.5.4 The VF2 Algorithm

For pattern matching, most combinatorial approaches apply the *VF2 algorithm* [86]. The algorithm, given in Alg. 2.4, recursively checks every node-mapping candidate and applies a set of rules for checking whether the current mapping yields a match. The rules are categorized into syntactic and semantic rules. The *syntactic rules* check that the graph structure is preserved, and the *semantic rules* check that node and edge attributes are compatible. In the worst case, $O(N!N)$ mappings need to be checked.

Computing the Mapping Candidate Set The set of mapping candidates under consideration to be added to s is called the *mapping candidate set*, denoted $P(s)$. The set is computed as follows.

Definition 2.3 – Mapping Candidate Set Let $G_{\mathrm{F}} = (N_{\mathrm{F}}, E_{\mathrm{F}})$ and $G_{\mathrm{P}} = (N_{\mathrm{P}}, E_{\mathrm{P}})$ be a function graph respectively a pattern graph, and s be a set of mappings from nodes

```
function FindMatches(function graph G_F = (N_F, E_F), pattern graph G_P = (N_P, E_P)):
1   M ← ∅
2   FindMatchRec(∅)
3   return M
4   function FindMatchRec(set s of mappings):
5       if |s| = |N_P| then                                   // if match found
6           M ← M ∪ {s}
7       else
8           compute mapping candidate set P(s)                // see Def. 2.3
9           foreach (n, m) ∈ P(s) do
10              if R_SYN(s, n, m) ∧ R_SEM(s, n, m) then       // see Defs. 2.4–2.5
11                  FindMatchRec(s ∪ {(n, m)})
```

Algorithm 2.4: VF2 algorithm.

in G_F to nodes in G_P. Let $N_F(s)$ and $N_P(s)$ denote the sets of nodes in G_F and G_P, respectively, that appear in s. Also let T_F^{OUT} and T_P^{OUT} denote the set of nodes in G_F and G_P, respectively, that are targets of edges from nodes appearing in s. Likewise, let T_F^{IN} and T_P^{IN} denote the set of nodes in G_F and G_P, respectively, that are sources of edges to nodes appearing in s. Then

$$P(s) \equiv \begin{cases} \{(n,m) \mid n \in T_F^{OUT}, m \in T_P^{OUT}\} & \text{if } T_F^{OUT} \neq \emptyset \wedge T_P^{OUT} \neq \emptyset, \\ \{(n,m) \mid n \in T_F^{IN}, m \in T_P^{IN}\} & \text{if } T_F^{IN} \neq \emptyset \wedge T_P^{IN} \neq \emptyset, \\ \{(n,m) \mid n \in N_F \setminus N_F(s), m \in N_P \setminus N_P(s)\} & \text{otherwise.} \end{cases}$$

The last clause is needed when G_F or G_P consists of disconnected subgraphs.

Syntactic Rules The syntactic rules check five properties:

$$R_{SYN}(s,n,m) \equiv R_{PRED}(\ldots) \wedge R_{SUCC}(\ldots) \wedge R_{IN}(\ldots) \wedge R_{OUT}(\ldots) \wedge R_{NEW}(\ldots). \quad (2.4)$$

The first two rules, R_{PRED} and R_{SUCC}, check the consistency of the partial match obtained when the candidate (n, m) is added to s. Intuitively, if there exists an edge between two mapped nodes in the pattern graph, then a corresponding edge must also exist in the function graph.[9] The next two rules, R_{IN} and R_{OUT}, are 1-look-ahead rules that check whether there exists a sufficient number of unmapped nodes adjacent to n in the function graph for mapping the remaining nodes adjacent to m

[9] In the paper [86], R_{PRED} and R_{SUCC} also check the inverse – that is, an edge between two mapped nodes in the function graph must have a corresponding edge in the pattern graph – thus requiring that G_P is an induced subgraph of G_F. In the context of instruction selection, however, it is sufficient to only maintain the structure of G_P. Hence the condition above has been removed from Def. 2.4.

in the pattern graph. The last rule, R_{NEW}, is similar to R_{IN} and R_{OUT} but perform a 2-look-ahead check. Formally, the rules are defined as follows.

Definition 2.4 – Syntactic Rules Let $pred(G, n)$ and $succ(G, n)$ denote the sets of predecessor and successor nodes, respectively, to node n in graph G. Also let $T_{\text{F}} = T_{\text{F}}^{\text{IN}} \cup T_{\text{F}}^{\text{OUT}}$ and $\overline{N}_{\text{F}} = N_{\text{F}} \setminus N_{\text{F}}(s) \setminus T_{\text{F}}$, with similar definitions for T_{P} and $\overline{N}_{\text{P}}$. Then

$$
\begin{aligned}
R_{\text{PRED}}(s, n, m) &\equiv \forall m' \in N_{\text{P}}(s) \cap pred(G_{\text{P}}, m), \exists n' \in N_{\text{F}}(s) : (n', m') \in s, \\
R_{\text{SUCC}}(s, n, m) &\equiv \forall m' \in N_{\text{P}}(s) \cap succ(G_{\text{P}}, m), \exists n' \in N_{\text{F}}(s) : (n', m') \in s, \\
R_{\text{IN}}(s, n, m) &\equiv |succ(N_{\text{F}}, n) \cap T_{\text{F}}^{\text{IN}}| \geq |succ(N_{\text{P}}, n) \cap T_{\text{P}}^{\text{IN}}| \wedge \\
&\quad\ |pred(N_{\text{F}}, n) \cap T_{\text{F}}^{\text{IN}}| \geq |pred(N_{\text{P}}, n) \cap T_{\text{P}}^{\text{IN}}|, \\
R_{\text{OUT}}(s, n, m) &\equiv |succ(N_{\text{F}}, n) \cap T_{\text{F}}^{\text{OUT}}| \geq |succ(N_{\text{P}}, n) \cap T_{\text{P}}^{\text{OUT}}| \wedge \\
&\quad\ |pred(N_{\text{F}}, n) \cap T_{\text{F}}^{\text{OUT}}| \geq |pred(N_{\text{P}}, n) \cap T_{\text{P}}^{\text{OUT}}|, \\
R_{\text{NEW}}(s, n, m) &\equiv |\overline{N}_{\text{F}} \cap pred(G_{\text{F}}, n)| \geq |\overline{N}_{\text{P}} \cap pred(G_{\text{P}}, m)| \wedge \\
&\quad\ |\overline{N}_{\text{F}} \cap succ(G_{\text{F}}, n)| \geq |\overline{N}_{\text{P}} \cap succ(G_{\text{P}}, m)|.
\end{aligned}
$$

Semantic Rules The semantic rules check two properties:

$$R_{\text{SEM}}(s, n, m) \equiv R_{\text{NODE}}(\ldots) \wedge R_{\text{EDGE}}(\ldots). \tag{2.5}$$

The first rule checks that the node types are compatible, while the second rule checks compatibility between edges. Formally, the rules are defined as follows.

Definition 2.5 – Semantic Rules Let $\preceq$ represent a binary relation for comparing the compatibility between nodes and edges. Also let $E_{\text{F}}(s)$ and $E_{\text{P}}(s)$ denote the sets of edges in G_{F} and G_{P}, respectively, for all pairs of nodes appearing in s. Then

$$
\begin{aligned}
R_{\text{NODE}}(s, n, m) &\equiv n \preceq m, \\
R_{\text{EDGE}}(s, n, m) &\equiv \big(\forall (n', m') \in E_{\text{F}}(s) : (n, n') \in E_{\text{F}} \Rightarrow (n, n') \preceq (m, m')\big) \wedge \\
&\quad\ \big(\forall (n', m') \in E_{\text{F}}(s) : (n', n) \in E_{\text{F}} \Rightarrow (n', n) \preceq (m', m)\big).
\end{aligned}
$$

2.6 Limitations of Existing Approaches

To solve the problems described in Chap. 1 on p. 3, none of the approaches discussed in this chapter can be applied directly. The greedy approaches are only concerned with pattern selection, making it unclear how to extend these to integrate other code generation tasks. The combinatorial approaches show more promise in that regards as they apply generic solving techniques, but instead the models of these approaches are too limited.

First, while many combinatorial approaches combine instruction selection with instruction scheduling, none combines instruction selection with global code motion. For most of these, it is not clear how to extend the model to integrate this task.

Second, all combinatorial approaches only handle tree- and DAG-shaped patterns. This excludes support for inter-block instructions which extend over multiple blocks and must therefore be modeled as pattern graphs (one such example is given in Chap. 1 on p. 4).

Third, with the exception of Tanaka et al. [348], no combinatorial approach takes the cost of data copying into account. Failing to consider this cost could lead to greedy use of SIMD instructions, which in turn degrades code quality.

Fourth and last, all combinatorial approaches only deal with data flow. Problems concerning control flow, such as selection of branch instructions and block ordering, must be handled separately, which could potentially result in suboptimal code.

To summarize, we need a more general combinatorial model that integrates the problems described in Chap. 1. To that end, we also need a more powerful means of representing functions and instructions.

CHAPTER

3

Constraint Programming

This chapter describes constraint programming (CP), which is a method for solving combinatorial problems. Like similar methods such as *integer programming (IP)* and *Boolean satisfiability (SAT)*, in CP we first *model* the problem and then we *solve* the model. The modeling and solving aspects are described in Sects. 3.1 and 3.2, respectively. Comprehensive overviews of CP, IP, and SAT are given in [321], [370], and [46], respectively. In Sect. 3.3 we briefly describe lazy clause generation, which is a solving technique that CHUFFED – the constraint solver used in the experiments – is based.

In terms of modeling, CP offers a higher level of abstraction compared to other methods. For example, CP provides dedicated constraints for capturing many recurring problem structures that must be decomposed and reformulated in IP or SAT. This makes CP particularly suited for modeling the problems introduced in Chap. 1.

3.1 Modeling

To solve a problem using CP, it must first be formulated as a constraint model. Modeling strategies are discussed in detail by Smith [339].

A *constraint model* (or just *model*) consists of two components: a set of variables, and a set of constraints. *Variables* represent problem decisions and take their values from a finite domain. The *domain* of a variable $\mathbf{x}$, denoted $D(\mathbf{x})$, is typically is a set of integers, but it can also consist of real numbers and complex structures such as string, sets, and graphs [160]. A variable $\mathbf{x}$ is *assigned* if $|D(\mathbf{x})| = 1$, and we abbreviate a variable assignment $\mathbf{x} \in \{v\}$ to $\mathbf{x} = v$.

Constraints express relations between variables and forbid assignments that are illegal in the problem. Given a set of variables $\mathbf{x}_1, \ldots, \mathbf{x}_k$ and a constraint C, an assignment to $\mathbf{x}_1, \ldots, \mathbf{x}_k$ is a *solution to* C if $C(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ holds. An assignment to

variables	constraints
$\mathbf{x} \in \{1, 2\}$	$\mathbf{x} \neq \mathbf{y}$
$\mathbf{y} \in \{1, 2\}$	$\mathbf{x} \neq \mathbf{z}$
$\mathbf{z} \in \{1, 2, 3, 4\}$	$\mathbf{y} \neq \mathbf{z}$

(a) Constraint model.

x	y	z
1	2	3
2	1	3
1	2	4
2	1	4

(b) Solutions.

Table 3.1: Example of a constraint model, corresponding to a problem where three variables must be assigned values which are different from one another.

all variables fulfilling all constraints in a model M is a *solution to* M. An example of a constraint model and its solutions is shown in Tab. 3.1.

The use of variables and constraints results in constraint models that are *compositional*, meaning they can easily be extended to capture additional problems to be solved in unison.

3.1.1 Global Constraints

If a *binary constraint* is a constraint involving two variables, then a *global constraint* is a constraint over an arbitrary number of variables [192]. Global constraints capture recurring problem structures and improve solving compared to relations modeled using multiple binary constraints.

The All-Different Constraint Arguably, the most well-known global constraint is the *all-different constraint* [236] (see [191] for a survey), which enforces all variables in a given set to take distinct values. We refer to this constraint as ALLDIFFERENT, which is defined as follows.

Definition 3.1 – ALLDIFFERENT Let $\mathbf{x}_1, \ldots, \mathbf{x}_k$ be a list of variables. Then

$$\text{ALLDIFFERENT}(\mathbf{x}_1, \ldots, \mathbf{x}_k) \equiv \bigwedge_{1 \leq i < j \leq k} \mathbf{x}_i \neq \mathbf{x}_j.$$

Hence the constraints in Tab. 3.1 can be replaced by ALLDIFFERENT$(\mathbf{x}, \mathbf{y}, \mathbf{z})$.

The Global Cardinality Constraint In Chap. 2 we saw another global constraint – the *global cardinality constraint* [293] – and how it can be used to model the pattern selection problem (see Eq. 2.2 on p. 33). This constraint is a generalization of the all-different constraint, and for completeness we give here the its formal definition.

Definition 3.2 – GCC Let $v_1, \ldots, v_k$ be a list of values, and let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ and $\mathbf{c}_1, \ldots, \mathbf{c}_k$ be lists of variables. Then

$$\text{GCC}(\langle v_1, \mathbf{c}_1 \rangle, \ldots, \langle v_k, \mathbf{c}_k \rangle, \mathbf{x}_1, \ldots, \mathbf{x}_n) \equiv \bigwedge_{1 \leq i \leq k} \left| \{ \mathbf{x}_j \mid \forall 1 \leq j \leq n : \mathbf{x}_j = v_i \} \right| = \mathbf{c}_i.$$

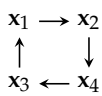
For example, GCC($\langle 5, \mathbf{c}_1 = 0\rangle, \langle 3, \mathbf{c}_2 = 1\rangle, \mathbf{x}_1 = 2, \mathbf{x}_2 = 3$) holds because no $\mathbf{x}$ variable is assigned value 5 and exactly one $\mathbf{x}$ variable is assigned value 3. Similarly, GCC($\langle 3, \mathbf{c}_1 \in \{0, 2\}\rangle, \mathbf{x}_1 = 2, \mathbf{x}_2 = 3$) does not holds because either none or both $\mathbf{x}$ variables must be assigned value 3.

The Circuit Constraint Another relevant example is the *circuit constraint* [236], which enforces that the variables representing adjacency forms a Hamiltonian cycle. We refer to this constraint as CIRCUIT, which is defined as follows.

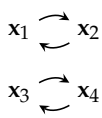
Definition 3.3 – CIRCUIT Let $\mathbf{x}_1, \ldots, \mathbf{x}_k$ be a list of variables, and let $P = d_1, \ldots, d_k$ be a permutation of domain values such that $d_i \in D(\mathbf{x}_i)$ for all $i = 1, \ldots, k$. Given P, form a graph $G = (N, E)$ such that there is exactly one node $n_i \in N$ and exactly one edge $n_i \rightarrow n_{d_i}$ for all $i = 1, \ldots, k$. P is considered *cyclic* if and only if G contains a Hamiltonian cycle. Then

$$\text{CIRCUIT}(\mathbf{x}_1, \ldots, \mathbf{x}_k) \equiv \mathbf{x}_1, \ldots, \mathbf{x}_k \text{ is cyclic.}$$

For example, CIRCUIT($\mathbf{x}_1 \in \{2\}, \mathbf{x}_2 \in \{4\}, \mathbf{x}_3 \in \{1\}, \mathbf{x}_4 \in \{3\}$) holds because the assignment forms the following cycle:



However, CIRCUIT($\mathbf{x}_1 \in \{2\}, \mathbf{x}_2 \in \{1\}, \mathbf{x}_3 \in \{4\}, \mathbf{x}_4 \in \{3\}$) does not hold because the assignment forms two non-Hamiltonian cycles:



CIRCUIT is used in Chap. 5 to model block ordering.

The Table Constraint The *table constraint* constrains a list of variables such that the values appear as a row in a given matrix. By encoding legal variable assignments into the matrix, any relation can be expressed using a table constraint, thus belonging to a group of so-called *extensional constraints* [339, Sect. 11.5.4]. We refer to this constraint as TABLE, which is defined as follows.

Definition 3.4 – TABLE Let $\mathbf{x}_1, \ldots, \mathbf{x}_k$ be a list of variables, and let T be an $m \times k$ matrix, where $m \in \mathbb{N}$. Then

$$\text{TABLE}(\langle \mathbf{x}_1, \ldots, \mathbf{x}_k \rangle, T) \equiv \langle \mathbf{x}_1, \ldots, \mathbf{x}_k \rangle \in T.$$

For example, assume we are given a matrix

$$A = \begin{bmatrix} 1 & 1 \\ 2 & 4 \end{bmatrix}.$$

Then $\text{TABLE}(\mathbf{x}_1 = 2, \mathbf{x}_2 = 4, A)$ holds because the tuple $\langle 2, 4 \rangle$ appears as a row in A. Similarly, $\text{TABLE}(\mathbf{x}_1 = 3, \mathbf{x}_2 = 3, A)$ does not hold because the tuple $\langle 3, 3 \rangle$ appears as a row in A.

TABLE is used in Chap. 6 to refine the model and to model several of the solving techniques.

The Value-Precede-Chain Constraint The *value-precede-chain constraint* [237] requires a list of variables to be sorted according to a given chain of values. We refer to this constraint as VPC, which is defined as follows.

Definition 3.5 – VPC Let $c_1, \ldots, c_n$ be a list of values and $\mathbf{x}_1, \ldots, \mathbf{x}_k$ be a list of variables. Then

$$\text{VPC}(c_1, \ldots, c_n, \mathbf{x}_1, \ldots, \mathbf{x}_k) \equiv \bigwedge_{\substack{1 \leq i \leq k, \\ 1 \leq j < n}} \mathbf{x}_i = c_{j+1} \Rightarrow (\exists l < j : \mathbf{x}_l = c_j).$$

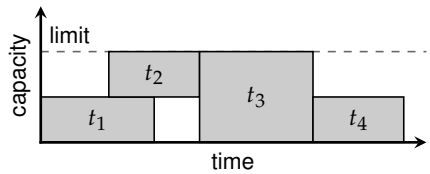
For example, $\text{VPC}(6, 5, 4, \mathbf{x}_1 = 6, \mathbf{x}_2 = 1, \mathbf{x}_3 = 5, \mathbf{x}_4 = 4)$ holds because the 4 is preceded by a 5, which in turn is preceded by a 6, in the list of $\mathbf{x}$ variables. Likewise, $\text{VPC}(5, 4, \mathbf{x}_1 = 5, \mathbf{x}_2 = 1)$ also holds because 4 does not appear among the $\mathbf{x}$ variables (the fact that 5 appears in the list does not matter). However, $\text{VPC}(5, 4, \mathbf{x}_1 = 1, \mathbf{x}_2 = 4)$ does not hold because the 4 is not preceded by a 5.

VPC is used in Chap. 6 to implement a dominance breaking constraint.

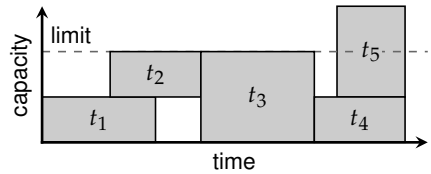
The Cumulative Constraint The *cumulative constraint* [2] is used in scheduling to constrains the scheduling times for a given set of tasks such that the capacity of a given resource is not exceeded. We refer to this constraint as CUMULATIVE, which is defined as follows.

Definition 3.6 – CUMULATIVE Let $c \in \mathbb{N}$ represent the capacity of a resource to be used by k optional tasks. For each task $i = 1, \ldots, k$, let $\mathbf{s}_i \in \mathbb{N}$ be a variable representing the time at which i is scheduled to start, and $\mathbf{b}_i \in \{0, 1\}$ be a variable representing whether i is scheduled. Let also $l_i \in \mathbb{N}$ represent task i's duration, and $u_i \in \mathbb{N}$ represent the amount of resource required by i, Lastly, let $t_{\text{MAX}} = max(\cup_{1 \leq j \leq k} D(\mathbf{s}_j))$. Then

$$\text{CUMULATIVE}(c, \langle \mathbf{s}_1, l_1, u_1, \mathbf{b}_1 \rangle, \ldots, \langle \mathbf{s}_k, l_k, u_k, \mathbf{b}_k \rangle) \equiv \bigwedge_{0 \leq t < t_{\text{MAX}}} \sum_{\substack{1 \leq i \leq k \text{ s.t.} \\ \mathbf{s}_i \leq t < \mathbf{s}_i + l_i}} \mathbf{b}_i \times u_i \leq c.$$

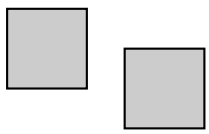


(a) A solution.

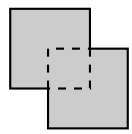


(b) Not a solution; the capacity is exceeded.

Figure 3.1: Examples illustrating the cumulative constraint. Each box represents a task.



(a) A solution.



(b) Not a solution.

Figure 3.2: Examples illustrating the no-overlap constraint.

For example, the schedule shown in Fig. 3.1a is a solution to this constraint because at no time is the capacity exceeded. Likewise, the schedule shown in Fig. 3.1b is not a solution because tasks t_4 and t_5 exceed the capacity when scheduled in parallel.

CUMULATIVE is used in Chap. 8 to model instruction scheduling.

The No-Overlap Constraint The last global constraint we will look at is the *no-overlap constraint* (often also called the *diffn constraint* [42]), which is used in rectangle packing problems to enforce that no two rectangles may overlap. We refer to this constraint as NOOVERLAP, which is defined as follows.

Definition 3.7 – NOOVERLAP For each rectangle i, let $\mathbf{xl}_i, \mathbf{xr}_i, \mathbf{yl}_i, \mathbf{yu}_i \in \mathbb{N}$ be variables representing the rectangle's left, right, lower, respectively upper boundary. Then

$$\text{NOOVERLAP}(\langle \mathbf{xl}_1, \mathbf{xr}_1, \mathbf{yl}_1, \mathbf{yu}_1 \rangle, \ldots, \langle \mathbf{xl}_k, \mathbf{xr}_k, \mathbf{yl}_k, \mathbf{yu}_k \rangle) \equiv \bigwedge_{1 \le i < j \le k} \mathbf{xr}_i \le \mathbf{xl}_j \vee \mathbf{xr}_j \le \mathbf{xl}_i \vee \mathbf{yu}_i \le \mathbf{yl}_j \vee \mathbf{yu}_j \le \mathbf{yl}_i.$$

For example, Fig. 3.2a is a solution to this constraint because the two squares do not overlap. Likewise, Fig. 3.2b is a not solution because the squares do overlap.

NOOVERLAP is used in Chap. 8 to model register allocation.

3.1.2 Optimization

An optimization problem is modeled by maximizing or minimizing a variable $\mathbf{c}$ whose value is constrained according to an objective function. For example, if $\mathbf{x}_m \in \{0, 1\}$ is variable representing whether a match m is selected and c_m denotes the cost of selecting m, then a CP idiom for modeling optimal pattern selection is

$$\begin{aligned} &\text{minimize} && \mathbf{c} \\ &\text{subject to} && \mathbf{c} = \sum_m c_m \mathbf{x}_m \end{aligned} \tag{3.1}$$

In this context, $\mathbf{c}$ is called a *cost variable*. Note that the objective function is orthogonal to the rest of the model, thus allowing it to be easily customized to fit the desired optimization criterion.

3.2 Solving

A *constraint solver* (or just *solver*) finds solutions to a constraint model by interleaving propagation and search. *Propagation* removes domain values that are known to be in conflict with a constraint, and *search* attempts several alternatives when propagation alone is insufficient for finding a solution.

In practice, however, this alone is often not enough for many problem instances as the search space is simply too large. In such cases, the search space can be further reduced by extending the constraint model with additional constraints to strengthen propagation and remove uninteresting solutions and by performing presolving.

3.2.1 Propagation

Performing propagation requires an array of domain-pruning algorithms and a system that allows these algorithms to interact. Propagation theory is discussed in detail by Bessiere [45], and constraint programming systems are thoroughly discussed by Schulte and Carlsson [331].

Constraint solver typically keep track of variables and their domains using constraint stores. A *constraint store* (or just *store*) is a data structure that maps a set of variables to sets of domains. A store S_1 is *stronger* than another store S_2, denoted $S_1 \leq S_2$, if $D_1(\mathbf{x}) \subseteq D_2(\mathbf{x})$ for all variables $\mathbf{x}$, where $D_i(\mathbf{x})$ denotes the domain of variable $\mathbf{x}$ in store S_i.

A function that takes a constraint store as input and produces another store is called a *propagator* (or *filtering algorithm*). A propagator implements a constraint if it does not remove any solutions to the constraint and only keeps variable assignments that are part of a solution. For solving to be well-behaved, propagators are also expected to be *decreasing* – it does not add any values, hence $p(S) \leq S$ – and *monotonic* – if $S_1 \leq S_2$, then $p(S_1) \leq p(S_2)$. A propagator for which $p(S) = S$ holds is said to be at *fixpoint*, and a store is at fixpoint if all propagators are at fixpoint for that store. A

event	store		
	x	**y**	**z**
Initial store	{1, 2}	{1, 2}	{1, 2, 3, 4}
Propagate until fixpoint	{1, 2}	{1, 2}	{1, 2, 3, 4}
Search by attempting $\mathbf{z} = 1$	{1, 2}	{1, 2}	{1 }
Propagate $\mathbf{y} \neq \mathbf{z}$	{1, 2}	{ 2}	{1 }
Propagate $\mathbf{x} \neq \mathbf{z}$	{ 2}	{ 2}	{1 }
Propagate $\mathbf{x} \neq \mathbf{y}$	{ }	{ 2}	{1 }
Failure reached; backtrack	{1, 2}	{1, 2}	{ 2, 3, 4}
⋮			

(a) Solving with value-consistent inequality constraints.

event	store		
	x	**y**	**z**
Initial store	{1, 2}	{1, 2}	{1, 2, 3, 4}
Propagate ALLDIFFERENT($\mathbf{x}, \mathbf{y}, \mathbf{z}$)	{1, 2}	{1, 2}	{ 3, 4}
⋮			

(b) Solving with domain-consistent all-different constraint.

Table 3.2: Example illustrating propagation for two versions of the model given in Tab. 3.1, one using the all-different constraint and the other using a binary decomposition.

propagator that returns a store with at least one empty domain has *failed*, meaning there are no solutions in that part of the search space.

Propagators implementing the same constraint can differ in the amount of propagation they perform. A propagator is *value-consistent* if it only propagates when one of its variables becomes assigned, *bounds-consistent* if it only reduces the bounds of a domain, and *domain-consistent* if it removes all values that do not appear in any solution to the constraint. For example, Tab. 3.2 shows solving of two versions of the model given in Tab. 3.1, one using ALLDIFFERENT and another using inequality constraints. Because only value consistency can be achieved for inequality constraints, they cannot propagate anything until at least one variable becomes assigned (Tab. 3.2a). As all propagators are already at fixpoint, the solver must resort to search. In this case, the solver makes a wrong guess and is forced to backtrack. In comparison, a domain-consistent propagator for the all-different constraint can remove values 1 and 2 from the domain of variable $\mathbf{z}$ as these values do not appear in any solutions (Tab. 3.2b). As the search space grows exponentially with the number of variables and size of the domains, maximizing propagation is key in making solving tractable.

As to be expected, stronger propagation comes at a price of greater complex-

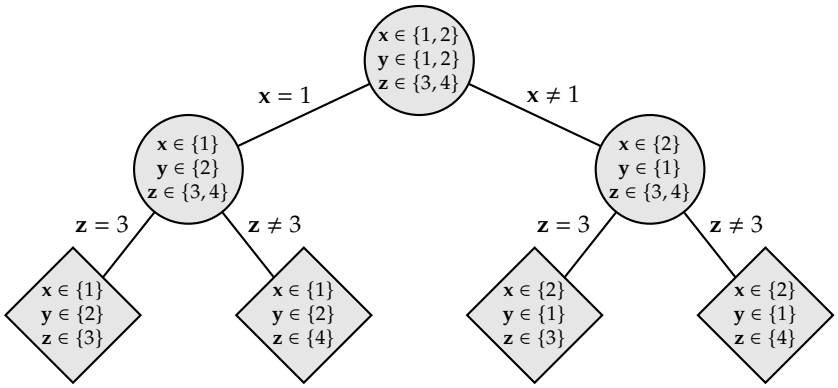


Figure 3.3: Example of a search tree for the model given in Tab. 3.1. Diamond-shaped nodes represent solutions.

ity. For the all-different constraint, there exist bounds and domain-consistent propagators with worst-case time complexities $O(n \log n)$ [255] and $O(n^{2.5})$ [317], respectively, where n denotes the number of variables. The same can be achieved for the global cardinality constraint at similar cost [309, 318].

Domain consistency for circuit constraint cannot be achieved in polynomial time as it involves finding Hamiltonian cycles, which is NP-complete [154]. An incomplete, polynomial-time filtering algorithm is given in [211].

Several domain-consistent propagator exist for the table [96, 238, 239, 240, 263, 300], cumulative [35], and no-overlap constraint [41, 184], but these exhibit exponential worst-case time complexity. For the value-precede-chain constraint, there exist domain-consistent propagators with linear time complexity [237].

3.2.2 Search

When no more propagation can be performed – that is, when all propagators are at fixpoint – the solver resorts to search. This is discussed in detail by Van Beek [39].

In exploring the search space, two decisions need to be made repeatedly: (i) select a variable on which to branch, and (ii) select one or more values (but not all) from its domain. These decisions constitute a *branching strategy*. The branching strategy arranges the search space into a *search tree*, where each node represents a store at fixpoint (see Fig. 3.3 for an example). Since the solutions (and failures) appear at the leaf nodes, the search tree is typically explored depth-first.

For search to be well-behaved, a branching strategy must preserve all solutions in the search space and must not duplicate any solution. A common strategy in choosing a variable, called the *first-fail principle*, is to select the variable most likely to cause a failure [176]. Other strategies involve selecting the variable with the smallest or largest value in its domain, or selecting a random variable. Similar strategies

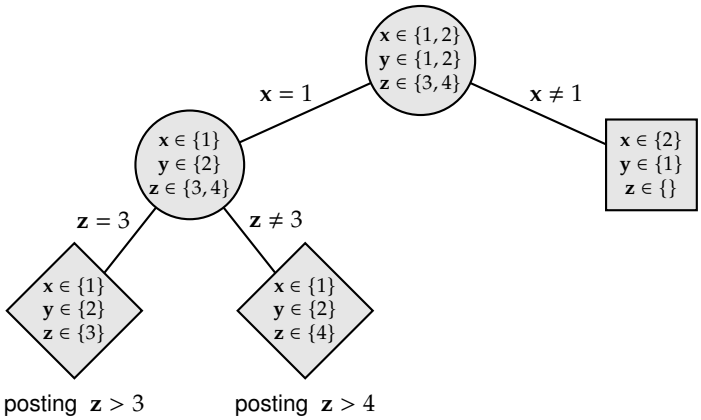


Figure 3.4: Example of a search tree for the model given in Tab. 3.1 with the additional requirement that the value of **z** should be maximized. The search tree is explored depth first, left to right. Diamond-shaped nodes represent solutions and square-shaped nodes represent failures.

are applied in value selection, which is typically done by posting constraint when branching. Most common is to post unary constraint that divides a domain into an assigned part an a non-assigned part. For example, at the root node in Fig. 3.3, search branches on variable **x** by posting $\mathbf{x} = 1$ in one branch and $\mathbf{x} \neq 1$ in the other. Another strategy is to split the domain by posting inequality constraints (for example, $\mathbf{x} \leq 3$ in one branch and $\mathbf{x} > 3$ in the other), which is useful for solving models with arithmetic constraints. More than one branching strategy can be used for the same model and, if needed, they can be customized by the user, making it a key strength of CP.

Branch and Bound Solutions to optimization problems are found using a method called *branch and bound* [231]. During search, the best solution found so far is kept and a constraint is added to enforce all subsequent solutions to have strictly less (or greater) cost. Hence time can be traded for quality on a continuous time scale. When the entire search space has been explored, the last found solution is guaranteed to be optimal. An example of shown in Fig. 3.4. Assuming the search tree is explored depth first, left to right, the first solution to be found is $S_1 = \langle \mathbf{x} = 1, \mathbf{y} = 2, \mathbf{z} = 3 \rangle$. Since the value of **z** is to be maximized, this causes the constraint $\mathbf{z} > 3$ to be posted. The next solution to be found is $S_2 = \langle \mathbf{x} = 1, \mathbf{y} = 2, \mathbf{z} = 4 \rangle$, which is clearly better than S_1, causing the constraint $\mathbf{z} > 4$ to be posted. Since the domain of **z** has no value greater than 4, the constraint $\mathbf{z} > 4$ causes a failure when the exploring the other branch at the root. At this point the entire search space has been explored, making S_2 the optimal solution.

3.2.3 Solving Techniques

Solving can be improved by applying various solving techniques, which can be divided into two categories. The first category involves additional constraints that are added to the model in order to increase propagation and also reduce the search space. These constraints can be divided into three categories – implied, symmetry breaking, and dominance breaking – which are discussed in detail by Smith [339], Gent et al. [158], and Chu and Stuckey [77], respectively.

The second category – presolving – involves applying methods that reduce the number of variables or shrink the variable domains, thereby reducing the search space.

Implied Constraints An *implied constraint* is a constraint that strengthens propagation without removing any solutions. For example, assume a naive model for solving the *magic sequence problem*, which is defined as finding a sequence $x_0, \ldots, x_{n-1}$ of integers such that for all $0 \leq i < n$, the number i appears exactly x_i times in the sequence. Using the global cardinality constraint and n variables, this can be modeled as

$$\text{GCC}(\cup_{0 \leq i < n} \langle i, \mathbf{x}_i \rangle, \mathbf{x}_0, \ldots, \mathbf{x}_{n-1}). \tag{3.2}$$

While this constraint is sufficient in capturing the problem, propagation can be increased by adding the following implied constraint:

$$\sum_{0 \leq i < n} \mathbf{x}_i = n. \tag{3.3}$$

This always holds because the sum of all occurrences – that is, the number of items in the sequence – must always be equal to the length of the sequence.

Symmetry Breaking Constraints A *symmetry breaking constraint* is a constraint that removes solutions considered to be symmetric to one another. For example, assume a model for solving a problem of packing n squares of sizes $1, \ldots, n$ inside another, larger square. Given a solution to this problem, more solutions can found by rotating, flipping, and mirroring the initial solution. As these solutions are essentially the same, only one of them should be kept in the search space. A simple method of removing most (but not all) symmetric solutions is to force one of the squares to be packed into one of the quadrants of the enclosing square.

Dominance Breaking Constraints A *dominance breaking constraint* is a constraint that removes solutions known to be dominated by another solution. Dominance breaking is therefore a generalization of symmetry breaking. For an example, let us revisit the model capturing the square packing problem and assume the two partial solutions shown in Fig. 3.5. In the left-most solution, the positioning of the two squares form an empty 2×3 rectangle in the corner. If this is extended to a complete solution, another solution can be found by sliding the 3×3 square all the

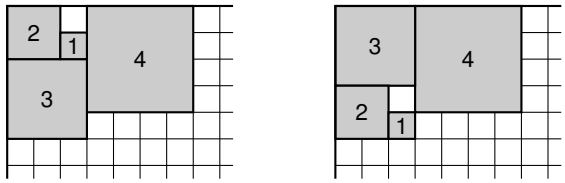


Figure 3.5: Example of two partial solutions to the square packing problem, where the left-most solution is dominated by the right-most solution [225].

way up and moving any squares packed above into the space created below. Hence the left-most solution is dominated by the right-most solution. We can remove such dominated solutions from the search space by forbidding each $k \times k$ square from being placed a certain distance away from the edge of the enclosing square such that the $k \times k$ square and the edge forms a rectangle large enough to pack all smaller squares inside it.

The benefit of a given implied, symmetry breaking, or dominance breaking constraint depends on the amount of search space it prunes and the cost of propagating the constraint. For example, if a constraint is expensive to run and only has marginal effect on the variable domain, then adding it to a model will *increase* solving time instead of decreasing it. In addition, it is well known that such constraints often have synergy effects among each other, meaning a constraint may not be useful on its own but may have a positive effect when combined with another constraint. Consequently, the decision of whether to add a implied, symmetry breaking, or dominance breaking constraint to a model must be based on careful and thorough experimental evaluation.

Presolving *Presolving* is the process of applying problem-specific algorithms to reduce the number of variables or to shrink the variable domains before solving. Fewer variable and smaller domains means smaller constraint models, which means shorter solving times.

When dealing with optimization problems, a common presolving technique is to precompute lower and upper bounds on the cost variable. A lower bound can be found by solving a relaxed, and hence simpler, version of the constraint model, which enables pruning of parts in the search space that contain no solutions. An upper bound can be found by solving the problem using a greedy but fast heuristic, which enables pruning of parts in the search space that only contain inferior solutions. If applying the upper bound yields a search space with no solutions, then we know that the heuristic has already found the optimal solution.

In the context of instruction selection, another presolving technique is to remove matches that we know cannot participate in any solution. Several such techniques are introduced in Chap. 6.

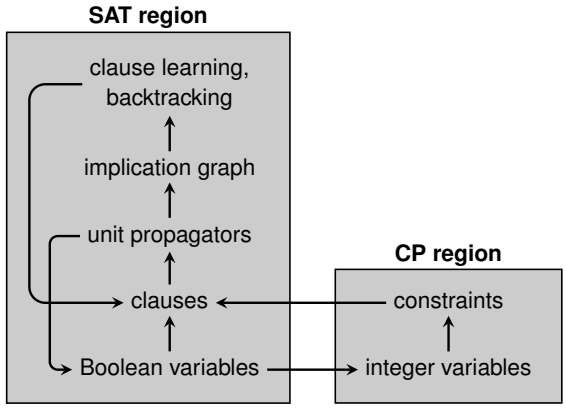


Figure 3.6: Overview of a typical LCG-based constraint solver.

3.3 Lazy Clause Generation

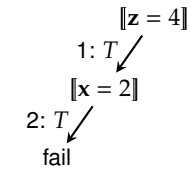
Lazy clause generation (LCG) is a solving technique where the constraint solver, when reaching a failure during search, learns new constraints in order to avoid repeating the same mistakes. The technique originates from SAT [265] but has been adapted to constraint programming by adding CP techniques on top of SAT solving [292]. An overview of a typical LCG-based constraint solver is shown in Fig. 3.6.[1]

To begin with, every variable has two dual representations: one based on Boolean values; and one based on integers, which is the domain typically used in CP. Every integer variable $\mathbf{x} \in [l, \ldots, u]$ results in two sets $[\![\mathbf{x} = l]\!], \ldots, [\![\mathbf{x} = u]\!]$ and $[\![\mathbf{x} \leq l]\!], \ldots, [\![\mathbf{x} \leq u - 1]\!]$ of Boolean variables, where $[\![e]\!]$ denotes a Boolean variable representing whether the expression e holds. For example, $\mathbf{x} \in [1, 2, 3]$ results in $[\![\mathbf{x} = 1]\!]$, $[\![\mathbf{x} = 2]\!]$, $[\![\mathbf{x} = 3]\!]$, $[\![\mathbf{x} \leq 1]\!]$, and $[\![\mathbf{x} \leq 2]\!]$. These Boolean variables are sufficient for capturing any constraint since $\mathbf{x} < k$, $\mathbf{x} > k$, and $\mathbf{x} \geq k$ are equivalent to $[\![\mathbf{x} \leq k - 1]\!]$, $\neg[\![\mathbf{x} \leq k - 1]\!]$, and $\neg[\![\mathbf{x} \leq k - 1]\!]$.

Constraints are applied over the integer variables as before, but during propagation the constraints do not directly shrink the domain of the integer variables. Instead they generate clauses consisting of disjunctions of literals, where a *literal* is a Boolean variables that may optionally be negated. For example, if ALLDIFFERENT propagates that two variables $\mathbf{x}, \mathbf{y} \in \{1, 2, 3\}$ cannot be assigned value 2, then it will produce two clauses $\neg[\![x = 2]\!] \vee [\![y = 1]\!] \vee [\![y = 3]\!]$ and $\neg[\![y = 2]\!] \vee [\![x = 1]\!] \vee [\![x = 3]\!]$ (these are equivalent to $\mathbf{x} = 2 \Rightarrow \mathbf{y} \in \{1, 3\}$ and $\mathbf{y} = 2 \Rightarrow \mathbf{x} \in \{1, 3\}$, respectively). Hence the Boolean dual model is built lazily as solving proceeds.

If a clause consist of one unassigned Boolean variable v and all other literals resolve to false, then in order for the clause to hold v must be assigned such that

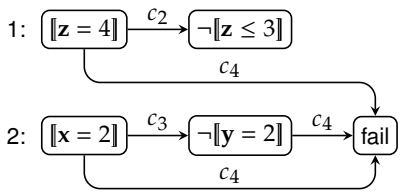
[1] Feydy and Stuckey [133] describe how to re-engineer the original implementation by embedding a SAT solver inside a constraint solver instead of vice versa.



(a) Search tree.

#	constraint
c_1	$[\![\mathbf{z} \leq 3]\!] \vee [\![\mathbf{x} = 2]\!] \vee [\![\mathbf{y} = 2]\!]$
c_2	$\neg[\![\mathbf{z} = 4]\!] \vee \neg[\![\mathbf{z} \leq 3]\!]$
c_3	$\neg[\![\mathbf{x} = 2]\!] \vee \neg[\![\mathbf{y} = 2]\!]$
c_4	$\neg[\![\mathbf{z} = 4]\!] \vee \neg[\![\mathbf{x} = 2]\!] \vee [\![\mathbf{y} = 2]\!]$

(b) Constraints.



(c) Implication graph.

$\neg[\![\mathbf{z} = 4]\!] \vee \neg[\![\mathbf{x} = 2]\!]$

(d) No-good.

Figure 3.7: Example of no-good learning. First $[\![\mathbf{z} = 4]\!]$ is assigned to true (T) through search. This triggers unit propagation of constraint c_2, which assigns $\neg[\![\mathbf{z} \leq 3]\!] = T$. Since no further propagation can be done, search is resumed by assigning $[\![\mathbf{z} = 4]\!] = T$. This triggers unit propagation of constraint c_3, which assigns $\neg[\![\mathbf{y} = 2]\!] = T$. This in turn causes constraint c_4 to fail. By choosing the literals closest to the failure node such that all decisions flow through those literals, we learn the no-good $\neg[\![\mathbf{z} = 4]\!] \vee \neg[\![\mathbf{x} = 2]\!]$.

the literal becomes true. This is called *unit propagation* [92]. For example, if we are given a clause $[\![\mathbf{x} = 1]\!] \vee \neg[\![\mathbf{y} = 4]\!]$ and $\mathbf{x} = 2$, then the Boolean variable $[\![\mathbf{y} = 4]\!]$ must be assigned false, meaning $\mathbf{y} \neq 4$.

When these unit propagations are caused by assignments that were made during search, we can use this information to build an *implication graph*. In this directed graph, each node represents a literal that resolves to true and each edge between two literals l_1 and l_2 represents the fact that l_1 causes l_2 to become true. When failure is reached, we can then use the implication graph to derive an explanation for why a constraint failed. This explanation can be captured as a clause, called *no-good*, to be added to the existing body of clauses. The no-good prevents the solver from making the same mistake again, thus effectively cutting away those parts of the search space. An example is given in Fig. 3.7. For more details, see [265].

It is well known that the effectiveness of LCG is heavily impacted by how the problem is modeled as that affects which no-goods can be learned [78, 332, 333]. For the same reason, LCG is also influenced by the implied, symmetry breaking, and dominance breaking constraints that have been added to the model. Consequently, due to LCG there may arise strong synergy effects between such constraints that would not have appeared when using a non-LCG constraint solver.

CHAPTER

4

Universal Representation

This chapter introduces a new representation used for modeling functions and instructions. Based on the conclusions drawn in Chap. 2, we begin in Sect. 4.1 with listing the design requirements that must be fulfilled by a graph-based representation. From these requirements, in Sect. 4.2 we design such a representation, called *universal representation*, and describe how it is used for modeling functions. We then do the same for instructions in Sect. 4.3. In Sect. 4.4, we describe how to perform pattern matching using the universal representation. In Sect. 4.5, we compare universal representation with other, existing sea-of-nodes IRs. Lastly, a summary is given in Sect. 4.6.

4.1 Design Requirements

As discussed in Chap. 2, in order to address the limitations of existing approaches we need a constraint model capable of capturing the problems described in Chap. 1. To this end, we need a graph-based representation that fulfills the following requirements:

R1 *It must capture the data and control flow of an entire function.* This is needed for modeling global instruction selection, which demands access to the entire function under compilation. This is also needed for uniform selection of data and control instructions, which requires that both data and control flow be captured in a single graph.

R2 *Blocks must be explicitly represented as nodes.* This is needed for pattern matching, where we must not be allowed to match patterns whose control flow is inconsistent with the function graph. For example, assume that the pattern is derived from a saturated addition instruction. Such a pattern will consist of three blocks b_1, b_2, and b_3, where b_1 represents the instruction's point of entry, b_2 represents the part where clamping is performed, and b_3 represents the

instruction's point of exit. The control flow in this pattern will be such that there are two conditional jumps from b_1 to either b_2 or b_3, and an unconditional jump from b_2 to b_3. To be matched, part of the function graph must exhibit the same control flow structure. This is also useful for modeling global code motion as each such node will correspond to a variable in the constraint model.

R3 *Data and control operations must be explicitly represented as nodes.* This is to retain the notion of coverage and to treat instructions uniformly regardless of whether they operate on data or control flow.

R4 *Values produced and used by the operations must be explicitly represented as nodes.* This is useful for modeling global code motion and data copying as every such node will introduce a variable in the constraint model.

R5 *In a function graph, every node representing a value must have exactly one inbound data-flow edge.* This ensures that every value has exactly one match defining that value, which is useful when modeling global code motion.

R6 *The block in which a particular operation in the function is to be performed must not be fixed.* Without this, global code motion is not possible.

R7 *Data operations must not be placed in blocks that will break program semantics.* This is needed to ensure correctness when performing global code motion.

R8 *The representation must be based on SSA.* This is needed in modeling global code motion as it explicitly states which values must be defined in which blocks in order to preserve program semantics. This is also useful for practical reasons as most IRs used in modern compilers are already based on SSA.

While there exist many graph-based representations (see [345] for a survey), most fulfill only some of the requirements but not all. Consequently, a new representation has to be designed.

4.2 Program Representation

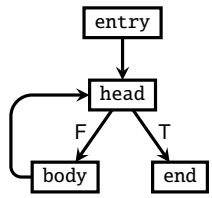
The universal representation is essentially a combination of two existing representations – the SSA graph and the control-flow graph[1] – which are extended to fulfill the missing requirements and then merged into a single graph. This makes for a simple construction process as the control-flow and SSA graphs are already used inside principally all modern compilers.

Capturing Control Flow We start with the control-flow graph. As it captures the control flow for an entire function, R1 is already partially fulfilled. R2 is also

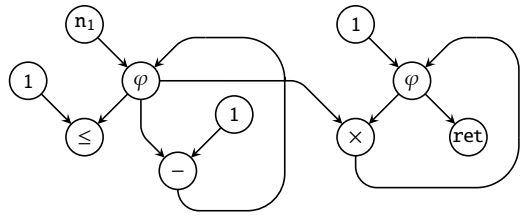
[1] A *control-flow graph* is a graph where each node represents a block in the function and each edge represents a jump from one block to another. Edges representing conditional jumps are labeled with a Boolean value indicating under which conditions the jump is taken. An example is given in Fig. 4.1b.

```
int factorial(int n1) {
  entry:
    int f1 = 1;
  head:
    int f2 = φ(f1:entry, f3:body);
    int n2 = φ(n1:entry, n3:body);
    bool b = n2 <= 1;
    if b goto end;
  body:
    int f3 = f2 * n2;
    int n3 = n2 - 1;
    goto head;
  end:
    return f2;
}
```

(a) Function in SSA form.



(b) Control-flow graph.



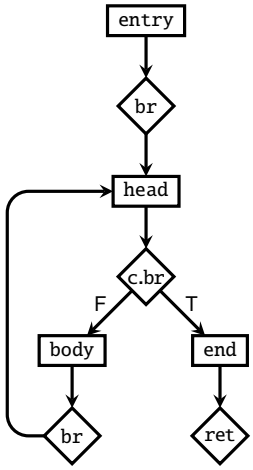
(c) SSA graph.

Figure 4.1: Running example of a function and its corresponding control-flow and SSA graph, which will be used in describing the program representation.

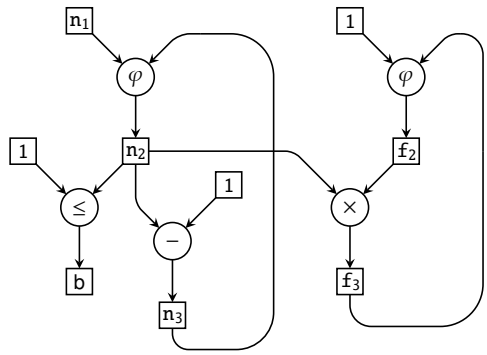
fulfilled since blocks in the control-flow graph are already represented as nodes, which we call *block nodes*. In contexts where is no risk of confusion, the terms *blocks* and *block nodes* can be used interchangeably.

To partially achieve R3, we insert *control nodes* to represent operations that change the control flow from on block to another. We also redirect the edges such that control flows through these nodes. For example, in the control-flow graph shown in Fig. 4.1b, control nodes representing unconditional branches are inserted along the edges between the `entry` and `head` nodes and between the `body` and `head` nodes. For the conditional control flow originating from the `head` block, a control node representing a conditional branch is inserted and connected to the `head` node, and the labeled edges are redirected to the new node. Lastly, a control node representing a function return is inserted and connected to the `end` node. This results in the graph shown in Fig. 4.2a.

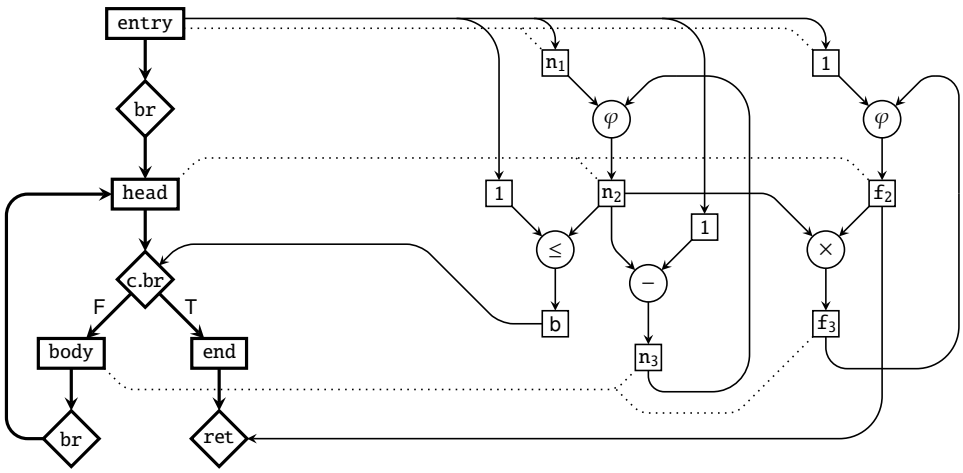
An invariant here is that each control node has exactly one edge flowing *from* a block node, and each block node has exactly one edge flowing *to* a control node. In other words, every control operation belongs to exactly one block, and every



(a) Extended control-flow graph.



(b) Extended SSA graph.



(c) Universal function graph.

Figure 4.2: Example of a universal function graph, built from the function shown in Fig. 4.1. Thick-lined diamonds, boxes, and arrows represent control nodes, block nodes, and control-flow edges, respectively. Thin-lined circles, boxes, and arrows represent computation nodes, value nodes, and data-flow edges, respectively. Dotted lines represent definition edges.

block has exactly one point where changes in control occur. This also means that the extended control-flow graph forms a bipartite graph, with block nodes on one end and control nodes on the other.

Capturing Data Flow We continue with the SSA graph. As it captures the data flow for an entire function and represents data operations as nodes – we call these *computation nodes* – the remaining parts of R1 and R3 are fulfilled. R8 is inherently fulfilled as the SSA graph requires the function to be in SSA form.

To achieve R4, we insert *value nodes* to represent the entities produced and used by the data operations. We also redirect the edges in same fashion as when extending the control-flow graph. Nodes representing function returns are removed as these are already represented in the extended control-flow graph. Using the SSA graph shown in Fig. 4.1c as example, this results in the graph shown in Fig. 4.2b.

Note that at this point the invariant specified in R5 that every value node has exactly one inbound data-flow edge is broken, but this will be addressed shortly.

Combining The Graphs We now connect the two extended graphs together. First, data-flow edges are inserted to connect control operations with the values used by these operations. In the case of our running example, such edges are added from values `b` and $\texttt{f}_2$ to the `c.br` and `ret` operations, respectively.

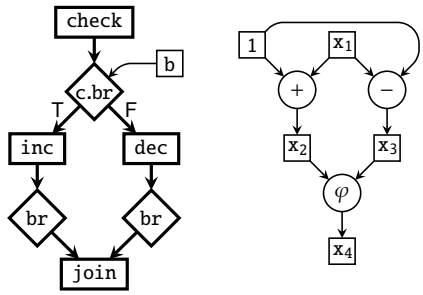
To achieve the invariant specified in R5, data-flow edges are also inserted from the entry block node to each value node representing constants and function arguments. Intuitively, this means that such values are produced at the point of entry to the function. Like with the extended control-flow graph, the extended SSA graph also forms a bipartite graph, with value nodes on one end and computation nodes on the other.

Since there are no edges connecting computation nodes with block nodes, the assignment of data operations to blocks is free, thus fulfilling R6. This alone, however, permits operations to be moved to blocks that will break program semantics. For example, assume the code snippet and corresponding, extended control-flow and SSA graphs shown in Fig. 4.3. In the original code, the addition should be performed in the `inc` block while the subtraction should be performed in the `dec` block. But according to the graph, swapping the placement of these operations would be considered a valid move, which clearly results in a different program. We recognize that such problems occur exactly in situations where a value is expected to be produced in a particular block, which are precisely the constraints captured by the φ-functions. Consequently, for each value-block pair (v, b) appearing as argument to a φ-function, we add a *definition edge* between the corresponding value and block node. This forces v to be produced in b, which in turn prevents the operation producing v from being moved out of b. Hence R7 is achieved.

Lastly, for convenience we prevent φ-functions from being moved by inserting a definition edge between the value produced by the φ-function and the block

```
  ...
  int x1 = ...;
check:
  bool b = ...;
  if b goto dec;
inc:
  int x2 = x1 + 1;
  goto merge;
dec:
  int x3 = x1 - 1;
join:
  int x4 = φ(x2:inc, x3:dec);
```

(a) Code snippet.



(b) UF subgraph.

Figure 4.3: Example illustrating the need for definition edges to prevent certain operations from being moved into blocks that will break program semantics.

wherein the φ-function originally resides in the function. This results in the graph shown in Fig. 4.2c,[2] which is called *universal function (UF) graph*.

Refining the Notion of Coverage Since new nodes have been introduced, we must refine the definition of coverage to apply for UF graphs. If an *operation* denotes either a computation or control node in a UF graph G, then a subset $M' \subseteq M$, where M is a match set, *covers* G if every operation in G appears in at least one match in M'. Similarly, exact coverage is also redefined as above.

4.2.1 Representing Constants As Single Or Multiple Nodes

Duplicated constants may either be represented using individual value nodes (as in Fig. 4.2c) or through a single value node (as in Fig. 4.3b). The former is simpler from a code generation perspective, but may result in redundant instructions where the same constant is needlessly reloaded. This can be avoided by using the latter together with a technique to be described in Sect. 5.4 in the next chapter, but this also requires the UF graph to be transformed and extended with additional operations in order to guarantee correctness. For example, assume a function containing two φ-functions φ(..., 1:a, ...) and φ(..., 1:b, ...). If the constant 1 is represented using a single value node, then the value node will have two definition edges, one from block a and another from block b. Since a value cannot be defined in two block simultaneously, there exist no solution for this UF graph. This problem is fixed by applying copy extension, which will be described in Chap. 5.

[2] In this case, the definition edges from the entry node to the n_1 and 1 nodes are redundant since the data-flow edges are sufficient to force these values to be produced in the entry block.

4.2.2 Data Types of Values

Both programs and instructions expect their argument values to be represented in a specific format. For example, in integer arithmetic the values are typically represented using signed or unsigned two's complement values of a certain width. It is also common that the result is given using the same format as the argument values. As this could lead to overflow, mechanisms are often in place to detect such occurrences. However, if the selected instruction produces results of wider bit width than specified in the function graph, the expected overflow may not occur. Consequently, for both function and pattern graphs the data type of a value is also specified in the corresponding value node. For integers this entails the value's bit width, and two values are compatible if they have the same bit width. Value nodes representing constants also have a value range, which is a singleton for constants appearing in the UF graph. A constant c is compatible with another constant d if the value range of d is a subset of the value range of c. Note that this relation is not necessarily commutative.

4.2.3 Common Subexpression Elimination

Common subexpressions may appear as part of legalizing the function before passing it to the instruction selector. For example, in LLVM there is an operation called `getelementptr` which takes care of computing the address when accessing an array element or object field. To construct the corresponding UF graph, these operations first need to be lowered into a series of additions and multiplications. However, if two or more `getelementptr`s compute the same address then these expressions will be duplicated, thus resulting in redundant instructions. Consequently, such common subexpressions should be eliminated after having performed legalization but before constructing the UF graph.

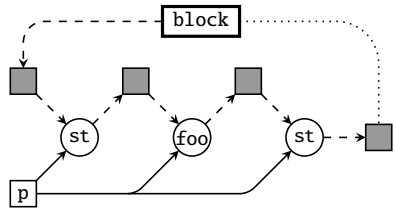
4.2.4 Handling Implicit Dependencies

The representation described thus far is sufficient for handling simple operations, such as arithmetic computations. Certain Operations, such as memory operations and function calls, require additional graph structures as they may implicitly depend on one another. Consequently, if these dependencies are not taken into account for global code motion, then such operations can be moved in a way that breaks program semantics.

We capture implicit dependencies using *state nodes*. In addition to its normal value arguments, an operation that may implicitly depend on another operation takes exactly one state node as input and produces another state node. Hence, if an operation o_1 implicitly depends on another operation o_2, then o_1 takes as input the state node produced by o_2. For each block in the function, a new state node is created and remembered as the last state node. When a computation node with potential for implicit dependencies is created, a state-flow edge is also inserted

```
block:
  ...
  store p, ...
  call foo, p
  store p, ...
```

(a) Code snippet.



(b) UF subgraph. Gray boxes represent state nodes, and dashed lines represent state-flow edges.

Figure 4.4: Example illustrating how to handle implicit dependencies in UF graphs.

from the last state node to the currently last state node. Afterwards a new state node is created and set as last, and another state-flow edge is inserted from the computation node to the just-created state node. Once all operations in the block has been processed, the first and last state nodes are connected to the block node through a state-flow edge and a definition edge, respectively. Because the first and the last state node must both be defined within the block, this effectively forbids operations with implicit dependencies from being moved out of the block, which could break program semantics. An example is shown in Fig. 4.4.

Note that, unlike with value nodes, the state-flow edge must not be drawn from the entry block node. Hence the same invariant for value nodes also apply for state nodes.

4.2.5 Edge Numbers

During graph construction, every edge is given two *edge numbers* which allows the edges to be ordered w.r.t. to a given node. This is needed for pattern matching, which will be described in Sect. 4.4. For a given edge $e = n_1 \rightarrow n_2$ of type t, where t represents either control flow, data flow, or state flow, the *inbound edge number*, denoted $in(e) = i$, indicates that e is the ith ingoing edge of type t connected to n_2. Similarly, the *outbound edge number*, denoted $out(e) = i$, indicates that e is the ith outgoing edge of type t connected to n_1. Consequently, all edges which are of the same type and have the same source node must be given distinct outbound edge numbers such that they form a contiguous sequence. Similarly, the same also applies for the inbound edge numbers of all edges with the same type and target node.

An example is given in Fig. 4.5. Note that the definition edges in the example are oriented and labeled with the same edge numbers as the data-flow edges connected to the same value node. This is to be able to identify which definition edge belongs to which data-flow edge when there exist multiple such edges, which is needed when performing copy extension (to be described in Chap. 5). In terms of restricting the definition placements, however, the orientation of the definition edges does not matter and is thus skipped in cases where this information is not needed.

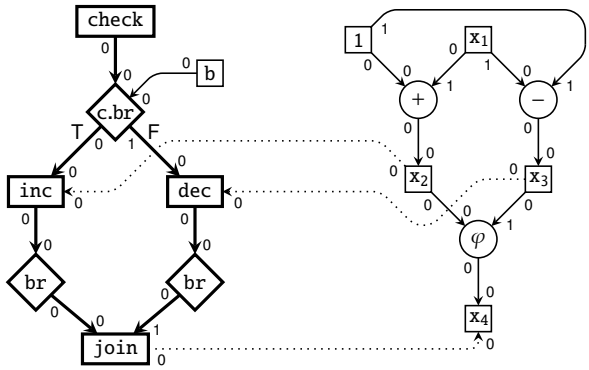


Figure 4.5: Example of edge numbers. The inbound and outbound edge numbers are attached to the edges' heads and tails, respectively.

4.3 Instruction Representation

Modeling instructions as patterns is identical to modeling functions with two exceptions. First, the control-flow graph becomes empty if the output is not dependent on control flow. Hence a pattern has either none or exactly one entry block. Second, no additional data-flow edges are added for value nodes representing constants and instruction input. In other words, the invariant specified in R5 that every value node has exactly one inbound data-flow edge does not need to (and should not) hold for such values. This results in a graph called *universal pattern (UP) graph*, for which two examples are given in Fig. 4.6.

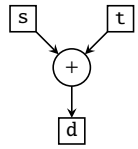
For the Haskell prototype used in the experiments, the UP graphs are generated from a proprietary machine description format where the behavior of each instruction is captured as LLVM IR code.

4.3.1 Covering φ-Nodes

Since covering of a UF graph also includes the nodes representing φ-functions – we call these *φ-nodes*, which typically do not correspond to any instruction on the target machine – we need a special pattern for covering such nodes. Consequently, we assume that the pattern set always includes a *φ-pattern*, which is illustrated in Fig. 4.7. The φ-pattern has a variable number of input values since a φ-function may take an arbitrary number of arguments. In our experiments, it proved sufficient to duplicate the φ-pattern for $k = 2, \ldots, 15$. A match derived from a φ-pattern is called a *φ-match*, which has zero cost and emits nothing if selected.

```
int add(int s, int t) {
  entry:
    int d = s + t;
    return d;
}
```

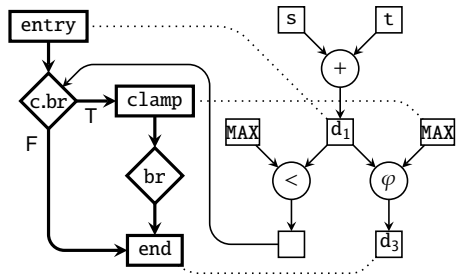
(a) Semantic behavior of an `add $d, $s, $t` instruction.



(b) UP graph of `add`.

```
int satadd(int s, int t) {
  entry:
    int d1 = s + t;
    if (d1 > MAX) goto clamp;
  clamp:
    int d2 = MAX;
  end:
    int d3 = φ(d3:entry, d2:clamp);
    return d3;
}
```

(c) Semantic behavior of a `satadd $d, $s, $t` instruction.



(d) UP graph of `satadd`.

Figure 4.6: Example of universal pattern graphs.

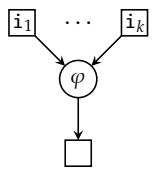


Figure 4.7: The φ-pattern.

4.4 Pattern Matching

Because neither UF nor UP graphs are necessarily tree-shaped, a subgraph isomorphism algorithm is needed for performing pattern matching. The prototype applies the VF2 algorithm [86] due to its simplicity and ease of implementation.[3] The algorithm is described in detail in Chap. 2 on p. 40.

To adapt it for the universal representation, we need to customize the semantic rules (see Def. 2.5 on p. 42). For two nodes n and m, $n \simeq m$ holds if n and m are of the same type. That is, the nodes must either be two block nodes, two

[3] In his dissertation, McCreesh [272] cautions against using VF2 after showing that, for a large body of problem instances, it exhibits considerably worse performance compared to two other subgraph isomorphism algorithms (*LAD* [341] and *Glasgow* [273]). In case of instruction selection, however, the problem instances are typically small enough that the time to find all matches is negligible compared to the time to solve the constraint model.

computation nodes, two value nodes, etc. For value nodes, their data types must also be compatible (as described in Sect. 4.2.2).

The same applies for two edges e and d, with the additional condition that, for certain nodes, the order of compatible edges must also match. Consequently, in a match where e and d are connected to nodes representing non-commutative operations – such as subtraction and division, but also memory stores and function calls – the inbound edge numbers of e and d must be identical. This prevents the arguments of such instructions from being swapped, which would obviously break program semantics. Similarly, in a match where e and d are connected to nodes with multiple outgoing edge of the same type – such as conditional jump operations – the outbound edge numbers of e and d must be identical, thus preventing swapping of target labels.

4.4.1 Matching SIMD Instructions Efficiently

Although the VF2 algorithm supports matching of patterns derived from disjoint-output instructions, doing so directly will result in many redundant matches. For example, if a SIMD pattern consists of k identical operations and the function contains n such operations, then this pattern alone will result in $n!/(n-k)!$ matches. However, since all SIMD instructions consist of disjoint patterns that are symmetric to one another, the order in which the pattern nodes are mapped to function nodes does not matter. Hence at most $\binom{n}{k}$ matches should be produced.

To this end, instead of matching the full SIMD pattern over the UF graph, we do so only for one of the disjoint subgraphs in the pattern. After having found all matches for the subgraph, we then compute all combinations of these matches and construct for each combination a match of the full SIMD pattern. We also ignore combinations that will lead to cyclic data dependencies by applying a variant of the method to be described in Sect. 5.1.2 in the next chapter.

To further reduce the number of matches, we remove all SIMD matches where at least one of its arguments is a constant value. This is because such values will typically always require the SIMD instruction to be preceded by a copy instruction, which would typically nullify the benefit of the SIMD instruction.

4.5 Comparison with Other Sea-of-Nodes IRs

The UF graph is similar to the Click-Paleczny graph (compare for example Fig. 4.2c on p. 62 with Fig. 2.10 on p. 38). Both are combinations of control-flow and SSA graphs, they represent (some) of the control operations as nodes, and they can restrict the placement of operations to blocks through auxiliary edges. However, the Click-Paleczny graph does not completely fulfill R3 as unconditional branches are not represented as nodes, nor does it fulfill R4 or R5.

Braun et al. [54] introduced another graph-based IR, called *Firm*, that is based on the Click-Paleczny graph and fulfills R3. However, it does not fulfill R6 as every operation is pre-assigned to a specific block, and it also does not fulfill R4 nor R5.

4.6 Summary

In this chapter, we have introduced a novel graph-based representation, called universal representation, that models both data and control flow on a global scope. This means that entire function can be captured as a single graph and that complex instructions, with or without control flow, can be modeled as patterns. The universal representation also provides enough freedom to move computations from one block to another without breaking program semantics. These features jointly enable the problems of global instruction selection, global code motion, and data copying to be modeled as a constraint model, which is introduced in the next chapter.

CHAPTER

5

Constraint Model

This chapter introduces the constraint model for universal instruction selection. We build the model by integrating one task a time. To this end, Sects. 5.1–5.5 describe the variables and constraints for modeling global instruction selection, global code motion, data copying, value reuse, and block ordering, respectively. We then add the objective function, which is described in Sect. 5.6. With all crucial components in place, we discuss the limitations of the model in Sect. 5.7. Lastly, a summary is given in Sect. 5.8.

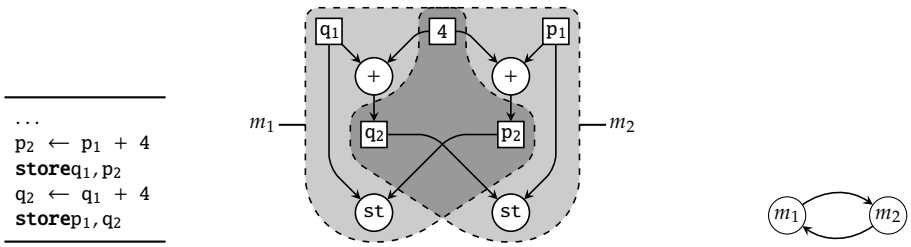
5.1 Modeling Global Instruction Selection

Modeling global instruction selection entails that all operations are covered and all data are defined. This could, however, lead to situations resulting in cyclic data dependencies, which must be prevented.

5.1.1 Covering Operations and Defining Data

In global instruction selection, a set of matches must be selected such that every operation in a given UF graph is covered. We model exact coverage since it enables use of many solving techniques that are essential for curbing solving time and increasing scalability. Similarly, we model that every value and state must be produced by exactly one selected match. If a *datum* d denotes either a state or value node in the UF graph, then we say that a match m *defines* d if there exists an inbound state-flow or data-flow edge to d in the UP graph from which m is derived. Likewise, m *uses* d if there exists an outbound state-flow or data-flow edge to d in the UP graph.

Variables Given a UF graph G and a set M of matches, the set of variables $\mathbf{sel}[m] \in \{0, 1\}$ models whether match $m \in M$ is selected. Hence m is selected if



(a) IR.

(b) UF graph, covered by two matches derived from an auto-increment store instruction. For brevity, the state nodes are not included.

(c) Dependency graph.

Figure 5.1: Example of cyclic data dependencies.

sel$[m] = 1$, abbreviated **sel**$[m]$, and not selected if **sel**$[m] = 0$, abbreviated $\neg$**sel**$[m]$.

The set of variables **omatch**$[o] \in M_o$ models which selected match covers operation $o \in O$, where O denotes the set of operations in G and $M_o \subseteq M$ denotes the set of matches covering o. Similarly, the set of variables **dmatch**$[d] \in M_d$ models which selected match defines datum $d \in D$, where D denotes the set of data in G and $M_d \subseteq M$ denotes the set of matches defining d.

Constraints The constraint that every operation must be covered is modeled as

$$\forall o \in O, \forall m \in M_o : \textbf{omatch}[o] = m \Leftrightarrow \textbf{sel}[m]. \tag{5.1}$$

This constraint gives equally strong propagation as Eq. 2.2, making it redundant to add the latter as an implied constraint to the model.

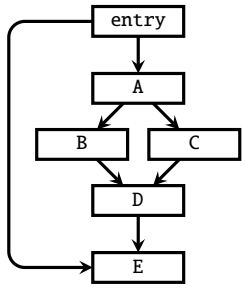
Likewise, the constraint that every datum must be defined is modeled as

$$\forall d \in D, \forall m \in M_d : \textbf{dmatch}[d] = m \Leftrightarrow \textbf{sel}[m]. \tag{5.2}$$

We assume that the pattern set has been extended with a special *null-def pattern*, with graph structure $b \rightarrow d$ where b is a entry block and d is a datum, that defines d at zero cost. This pattern is needed for defining data representing function arguments and constants since these are not produced by any operation.

5.1.2 Preventing Cyclic Data Dependencies

In certain cases, selecting matches of instructions that produce multiple results could lead to cyclic data dependencies [108]. For example, many modern processors provide memory instructions that load or store a value while also incrementing or decrementing the address. An example of such a situation is given in Fig. 5.1. If both matches are selected, then either value $\mathtt{p}_2$ or value $\mathtt{q}_2$ will be used before it is available (depending on the instruction order), thus resulting in incorrect code.



(a) Control-flow graph.

block	dominates
entry	{entry, A, B, C, D, E}
A	{A, B, C, D}
B	{B}
C	{C}
D	{D}
E	{E}

(b) Dominance.

Figure 5.2: Example of block dominance.

Such combinations, which could involve more than two matches, must therefore be identified and prevented.

We detect such combinations by first constructing a *dependency graph*, where each node represents a match and each edge $n \to m$ indicates that match m uses data produced by match n. φ-matches are not taken into consideration as they do not incur true cyclic data dependencies. A cycle in the dependency graph corresponds a combination of matches which will lead to a cyclic data dependency if all matches are selected. For each cycle in the dependency graph, we add a constraint to the constraint model that forbids selection of all matches appearing in the cycle. The cycles can be found using any cycle-finding algorithm (see for example [202]).

Constraints Given a set $F \subseteq 2^M$ of cycles found for the dependency graph built from a UF graph and match set M, the constraint forbidding cyclic data dependencies is modeled as

$$\forall f \in F : \sum_{m \in f} \mathbf{sel}[m] < |f|. \tag{5.3}$$

5.2 Modeling Global Code Motion

The global code motion problem entails that data are placed in blocks such that each definition of a datum d precedes all uses of d. This condition can be expressed in terms of block dominance. Given a function f, a block b in f *dominates* another block c in f if b appears on every control-flow path from f's entry block to c. By definition, a block always dominates itself (see Fig. 5.2 for an example).

Hence a placement of matches into blocks is a solution to the global code motion problem if each datum d is defined by some selected match placed in a block b, and every non-φ-match using d is placed in a block dominated by b. The φ-matches must be excluded since, due to the definition edges, at least one datum used by such matches must be defined in a block that does not dominate the block wherein the φ-match must be placed.

Variables The set of variables $\mathbf{oplace}[o] \in B$ models in which block operation o is placed, where B denotes the set of blocks in G. Likewise, the set of variables $\mathbf{dplace}[d] \in B$ models in which block the definition of datum d is placed.

Constraints Intuitively, all operations covered by a match m must be placed in the same block wherein m itself is placed. Hence, if $\mathit{covers}(m) \subseteq O$ denotes the set of operations covered by match m, then this constraint is modeled as

$$\forall m \in M, \forall o_1, o_2 \in \mathit{covers}(m) : \mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o_1] = \mathbf{oplace}[o_2]. \tag{5.4}$$

This also enables the placement of m to be deduced from any of the corresponding **oplace** variables.

We prevent control-flow operations from being moved to another block – which in all likelihood would break program semantics – by forcing selected matches with control flow to be placed in the block matched by the UP graph's entry block. Hence, if $\mathit{entry}(m) \subseteq B$ returns either the empty set or a set containing only the entry block of match m (when the UP graph has such a node), then this constraint is modeled as

$$\forall m \in M, \forall o \in \mathit{covers}(m), \forall b \in \mathit{entry}(m) : \mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o] = b. \tag{5.5}$$

As stated previously, each datum d must be defined in some block $b \in B$ such that b dominates every block wherein d is used, excluding uses made by the φ-matches. To this end, let $\mathit{defines}(m) \subseteq D$ and $\mathit{uses}(m) \subseteq D$ denote the set of data defined respectively used by match m. Let also $\mathit{dom}(b) \subseteq B$ denote the set of blocks dominated by block b. With these definitions together with the fact that the placement of a match is deduced from its **oplace** variables, the constraint can naively be modeled as

$$\begin{gathered}\forall m \in M_{\overline{\varphi}}, \forall d \in \mathit{uses}(m), \forall o \in \mathit{covers}(m) : \\ \mathbf{sel}[m] \Rightarrow \mathbf{dplace}[d] \in \mathit{dom}(\mathbf{oplace}[o]),\end{gathered} \tag{5.6}$$

where $M_{\overline{\varphi}} \subseteq M$ denotes the match set without the φ-matches. This implementation has a number of flaws that will be explained and addressed in Chap. 6. We therefore only use Eq. 5.6 for sake of describing the model, keeping in mind that a refined version is applied in practice.

Next, we need to constrain the **dplace** variables to depend on where a selected match is placed. Intuitively, every datum defined by a match m should be placed in the same block as m together with all operations covered by m. This alone, however, could result in an over-constrained model that prevents selection of certain matches. For example, assume a match m of the UP graph shown in Fig. 4.6b on p. 68, where the block nodes `entry`, `clamp`, and `end` are matched to blocks in G labeled `A`, `B`, and `C`, respectively. Because of Eq. 5.5, m must be placed in the `A` block. But because of Eq. 5.9, one of its value nodes must be placed in the `B` block.

Consequently, to allow such situations we relax the constraint as follows. First, we say that a match *spans* the blocks matched by the UP graph's block nodes (hence

m spans blocks A, B, and C). We also say that a match *consumes* any matched blocks where the corresponding block node has both inbound and outbound control-flow edges in the UP graph (hence m consumes block B). Using these definitions, we now enforce that every datum d defined by a match m must be placed in the same block as m only if m spans no blocks. Otherwise d may be defined in any of the blocks spanned by m (one of which is equal to $\mathbf{oplace}[m]$). If $spans(m) \subseteq B$ denotes the set of blocks spanned by match m, then this constraint is modeled as

$$\begin{gathered}\forall m \in M, \forall d \in defines(m), \forall o \in covers(m) : \\ \mathbf{sel}[m] \Rightarrow \mathbf{dplace}[d] \in \{\mathbf{oplace}[o]\} \cup spans(m).\end{gathered} \tag{5.7}$$

If a match consumes some block, then it means that the corresponding instruction assumes full control of the control flow to and from that block. Consequently, no operations covered by other matches can be placed in that block. Hence, if $consumes(m) \subseteq B$ denotes the set of blocks consumed by match m, then this constraint is modeled as

$$\begin{gathered}\forall m \in M, \forall o \in O \setminus covers(m), \forall b \in consumes(m) : \\ \mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o] \neq b.\end{gathered} \tag{5.8}$$

Lastly, the restrictions imposed by the definition edges are modeled as

$$\forall d \rightarrow b \in E : \mathbf{dplace}[d] = b, \tag{5.9}$$

where E denotes the set of definition edges in G. It is assumed that the edges in E have been reoriented such that all sources are either state or value nodes and all targets are block nodes.

5.3 Modeling Data Copying

The cost of data copying is taken into account by keeping track of the storage requirements for the data used and produced by the selected matches. The idea is as follows. For each value v in the UF graph, let a variable $\mathbf{x}$ decide in which location on the target machine v is stored. In this context a *location* is an abstract representation, typically representing a register but it could also indicate that the value is for example stored in memory. A match m that either uses or defines v and requires v to be in one of a set L of locations can then enforce that $\mathbf{x} \in L$.

Variables The set of variables $\mathbf{loc}[d] \in L \cup \{l_{\text{INT}}\}$ models in which location datum d is available, where L denotes the *location set* provided by the target machine and l_{INT} denotes a special location for values that cannot be reused across instructions. The special location is used for *intermediate values*, which are values produced within an instruction and can only be accessed by this very instruction. For example, the address computed by a memory load instruction with a sophisticated addressing mode is produced within the pipeline and therefore cannot be reused by other instructions. A value which is not an intermediate value is called an *exterior value*, meaning it can be accessed by other instructions.

Constraints Every datum must be made available in a location that is compatible for all selected matches. Let $stores(m, d) \subseteq L \cup \{l_{\text{INT}}\}$ denote the set of compatible locations (including the special location for intermediate values) for a datum d defined or used by a match m. With this definition, the constraint is modeled as

$$\forall m \in M, \forall d \in defines(m) \cup uses(m) : \mathbf{sel}[m] \Rightarrow \mathbf{loc}[d] \in stores(m, d). \tag{5.10}$$

As expected, φ-matches require all of its data to be stored in the same location. This is modeled as

$$\forall m \in M_{\varphi}, \forall d_1, d_2 \in defines(m) \cup uses(m) : \mathbf{sel}[m] \Rightarrow \mathbf{loc}[d_1] = \mathbf{loc}[d_2]. \tag{5.11}$$

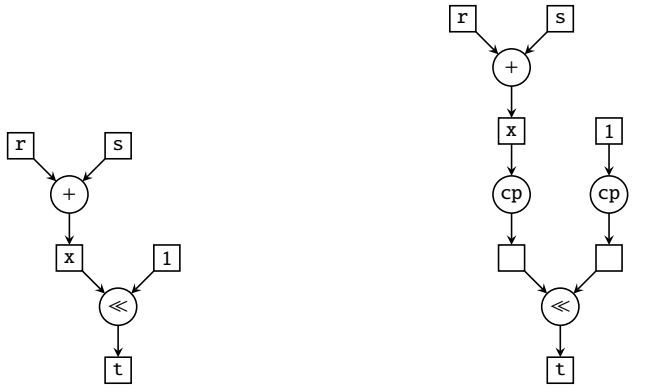
5.3.1 Copy Extension

Depending on the target machine, Eq. 5.10 can result in an over-constrained model. For example, in many target machines the SIMD instructions use a different set of registers than the other, general instructions. In such situations, the matches derived from the SIMD instructions and the matches derived from the general instructions will have non-overlapping locations on the same data (that is, $stores(m_1, d) \cap stores(m_2, d) = \varnothing$). Hence selection of such matches is prevented.

Since non-overlapping locations entail the need for copy instructions, we extend the UF graph with *copy nodes* through a process called *copy extension*. For each data-flow edge $v \rightarrow o$, where v is a value node and o is an operation, we remove this edge and insert a new copy node c, a new value node v', and new data-flow edges such that $v \rightarrow c \rightarrow v' \rightarrow o$. If o is a φ-node then the corresponding definition edge connected to v – this is the edge with the same outbound edge number as the data-flow edge – is also moved to v'. This is to ensure that the placement restrictions are applied only on the data actually used by the φ-functions (that is, the copied value and not the original value).

We also extend the pattern set with a special *null-copy pattern*, with graph structure $v \rightarrow c \rightarrow v'$, that covers c at zero cost provided that $\mathbf{loc}[v] = \mathbf{loc}[v']$. Matches covering exactly one copy node are called *copy matches*, and matches derived from the null-copy pattern are also called *null-copy matches*. Obviously, null-copy patterns emit nothing if selected. If the null-copy match cannot be selected for covering a particular copy node, then this means an appropriate copy instruction must be emitted whose cost will be accounted for.

In order to retain pattern matching, we also need to perform copy extension on every UP graph in the pattern set. See for example Fig. 5.3. The UP graph captures the behavior of an instruction that adds two values `r` and `s` and then shifts the result by one bit to the right (Fig. 5.3a). Since we want to preserve the ability of selecting copy instructions for moving data between instructions, we only copy-extend the values in a UP graph which are both defined and used by the pattern. We also copy-extend any constant values since these obviously do not require a separate copy instruction to be used by the pattern. The resulting UP graph will now yield the same matches as before copy extension (Fig. 5.3b).



(a) Original UP graph. (b) After copy extension.

Figure 5.3: Example of copy-extending a pattern.

5.3.2 Handling Calling Conventions

The data copying method can also be used for handling calling conventions of the specific target machine.[1] Constraints that callee arguments must reside in a specific location are modeled as

$$\forall d \in A : \mathbf{loc}[d] \in argLoc(d). \tag{5.12}$$

where $A \subseteq D$ denotes the set of function arguments in G and $argLoc(d) \subseteq L$ denotes the set of locations in which argument d resides. Constraints that certain arguments must reside on the stack can be captured by introducing another special location representing memory. Hence, if an instruction requires the argument to reside in a register, then the copy node must be covered by a copy match that emits a memory load instruction.

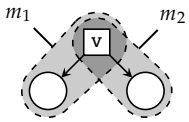
Locations for caller arguments can be enforced either through Eq. 5.12 or through Eq. 5.10. If the calling convention depends on the instruction selected – which is an unlikely scenario – then the former is needed. Otherwise the latter is more suitable as the restrictions can be enforced before a match is selected.[2]

5.4 Modeling Value Reuse

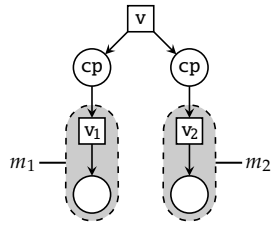
Code quality can be increased if instructions are allowed to reuse copies of values, which is a crucial feature to be expected in the code generated by any modern

[1] A *calling convention* is an implementation scheme that defines how arguments and return values are passed and received when making a function call. The function from which the call is made is typically called the *caller*, and the function to which the call is made is called the *callee*. A calling convention is specific for the target machine and may therefore differ from one target to another.

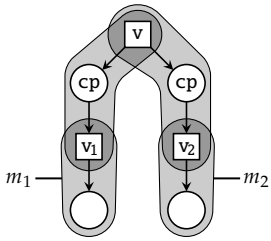
[2] If exactly one match m can cover a given function call node, then both constraints provide the same amount of propagation as $\mathbf{sel}[m] = 1$ will always hold for the implication in Eq. 5.10.



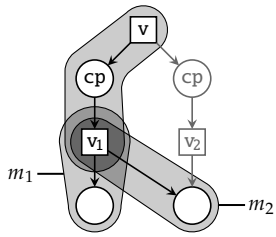
(a) A UF graph with two matches.



(b) UF graph after copy extension.



(c) Redundant copying of values.



(d) Reuse of copied value.

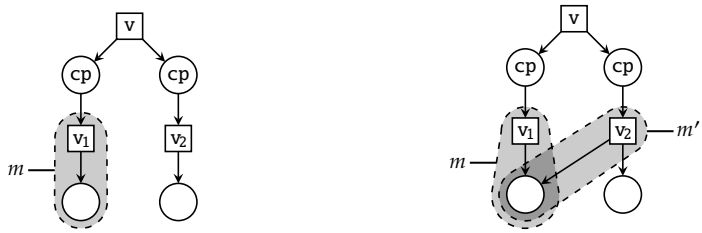
Figure 5.4: Example of value reuse.

instruction selector. See for example Fig. 5.4. Originally, the UF graph has a value v which is used by two operations coverable by matches m_1 and m_2 (Fig. 5.4a). After copy extension, m_1 and m_2 use their own private copy of $\mathtt{v} - \mathtt{v}_1$ and $\mathtt{v}_2$, respectively (Fig. 5.4b). Assume that both m_1 and m_2 require its copy of v to reside in a location different from v – for example, v may reside on the stack – which means that selection of m_1 or m_2 also entails emission of a copy instruction. With the model introduced thus far, two copy instructions will be emitted if both m_1 and m_2 are selected (Fig. 5.4c). However, if $\mathtt{v}_1$ and $\mathtt{v}_2$ could reside in the same location then either of the values could be reused by either match, thus requiring only one copy instruction (Fig. 5.4d). We call this notion *value reuse*.

In this dissertation we consider two methods for reusing values: match duplication and alternative values. We first introduce each in turn and then present experiments showing that one is superior to the other.

5.4.1 Match Duplication

The idea behind *match duplication* is to duplicate appropriate matches in the match set where value reuse is possible. We first say that two values v_1 and v_2 are *copy-related* if and only if they are copies of the same value and v_1 and v_2 have the same data type. The second clause is necessary as copies of constants may be of different data types and are therefore not interchangeable. Then, for each match m we create a new match for each permutation of data that are copy-related with the



(a) Original match set. (b) After match duplication.

Figure 5.5: Example of match duplication.

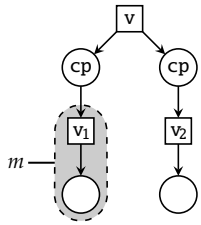
input data in m, where an *input datum* is a datum used but not defined by m. See for example Fig. 5.5. The match set contains a match m that uses value v_1, which is copy-related with value v_2 (Fig. 5.5a). Because v_1 is an input datum in m, we duplicate m to instead use v_2, resulting in match m' (Fig. 5.5b).

The main advantage of match duplication is that no changes need to be done for the constraint model; the decision of which value to use (and reuse) depends entirely on the selection of matches. However, this comes at a cost of inflating the match set, which in turn inflates the search space. If a match has k input data, each with n copy-related values, then $O(n^k)$ new matches will be created. Although the decision of value reuse is intuitively orthogonal to the decisions of selecting a match and placing it into a block, these decisions must be remade for each new match. Consequently, the search space is enlarged with many symmetric solutions.

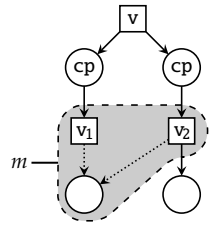
5.4.2 Alternative Values

Instead of expanding the match set (like in match duplication), we can postpone the decision of which input datum to use for a particular match and integrate it as part of the constraint model. The idea is as follows. For each match m, let every input data d in m be mapped to any datum that is copy-related to d. In other words, unlike before when a match was 1-to-1 mapping between nodes in the UP graph and nodes in the UF graph, we now allow certain mappings to be a 1-to-n mapping. See for example Fig. 5.6. Again, the match set contains a match m that uses value v_1, which is copy-related with value v_2 (Fig. 5.6a). Because v_1 is an input datum in m, we extend the mapping from n to v_1 to include v_2, where n is the corresponding value node in the UP graph of m (Fig. 5.6b). In this context, v_1 and v_2 are said to be *alternative values* to m. For convenience, we assume that the set of alternative values for each non-input datum d in a match contains only the state or value node to which d is already mapped.

Special care must be taken to matches derived from the φ-pattern. Assume, for example, that the match m using value v_1 in Fig. 5.6a is a φ-match. This means that v_1 will participate in a definition edge, forcing v_1 to be defined in some block. If m is extended as in Fig. 5.6b, then m is allowed to make use of value v_2, which does not



(a) Original match set.



(b) With alternative values.

Figure 5.6: Example of alternative values.

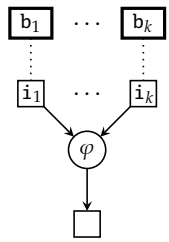


Figure 5.7: Extended φ-pattern.

participate in the definition edge and hence could break program semantics. There are two approaches to fixing this problem: (i) either all φ-matches are excluded from being extended with alternative values; or (ii) the φ-pattern is extended with block nodes and definition edges to allow the value placement restrictions to be captured as part of the match. The former is simpler but interferes with an implied constraint to be introduced in Chap. 6 (Eq. 6.33), which may remove potentially optimal solutions. Consequently we apply the latter, as shown in Fig. 5.7. The extended φ-pattern assumes that no value is used more than once by the same φ-node, and that no pair of values used by the same φ-node have definition edges to the same block. These invariants can be achieved by transforming the function as needed before pattern matching.

After having extended the match set with alternative values, the next step is to extend the constraint model with an another level of indirection wherever a constraint refers to a datum.

Variables Assume first that each definition or usage of data within each match introduces a unique *operand*. Instead of defining and using data, we now assume that all matches define and use operands. The set of variables $\mathbf{alt}[p] \in D_p$ models to which datum operand $p \in P$ is mapped, where $D_p \subseteq D$ denotes the set of alternative values for p and P denotes the set of operands introduced by M.

Constraints As stated previously, the aim is to add another level of indirection whenever a constraint refers to a datum. To this end, let $\mathit{defines}(m) \subseteq P$ and $\mathit{uses}(m) \subseteq P$ now denote the set of operands (instead of data) defined respectively used by match m. With these definitions, Eqs. 5.6, 5.7, 5.10, and 5.11 are adjusted accordingly (the changes are highlighted in gray):

$$\begin{gathered}\forall m \in M_{\overline{\varphi}}, \forall p \in \mathit{uses}(m), \forall o \in \mathit{covers}(m) : \\ \mathbf{sel}[m] \Rightarrow \mathbf{dplace}[\,\mathbf{alt}[p]\,] \in \mathit{dom}(\mathbf{oplace}[o]),\end{gathered} \tag{5.13}$$

$$\begin{gathered}\forall m \in M, \forall p \in \mathit{defines}(m), \forall o \in \mathit{covers}(m) : \\ \mathbf{sel}[m] \Rightarrow \mathbf{dplace}[\,\mathbf{alt}[p]\,] \in \{\mathbf{oplace}[o]\} \cup \mathit{spans}(m),\end{gathered} \tag{5.14}$$

$$\forall m \in M, \forall p \in \mathit{defines}(m) \cup \mathit{uses}(m) : \mathbf{sel}[m] \Rightarrow \mathbf{loc}[\,\mathbf{alt}[p]\,] \in \mathit{stores}(m, p), \tag{5.15}$$

$$\begin{gathered}\forall m \in M_{\varphi}, \forall p_1, p_2 \in \mathit{defines}(m) \cup \mathit{uses}(m) : \\ \mathbf{sel}[m] \Rightarrow \mathbf{loc}[\mathbf{alt}[p_1]] = \mathbf{loc}[\mathbf{alt}[p_2]].\end{gathered} \tag{5.16}$$

Due to these changes, there may be data that is not used by any match. However, Eq. 5.2 still requires that every datum must be defined by some match. We address this by extending the pattern set with a *kill pattern* with graph structure $v \to c \to v'$, where v and v' are value nodes and c is a copy node. Matches derived from this pattern are called *kill matches*, which have zero cost and emit nothing if selected. A datum is said to be *killed* if and only if it is defined by a kill match, and non-kill matches are only allowed to make use of non-killed data.

To model whether a datum is killed, we extend the location set with a special location l_{KILLED} and require that a kill match m is selected if and only if the location of the datum defined by m is l_{KILLED}. This is modeled as

$$\forall m \in M_{\times}, \forall p \in \mathit{defines}(m) : \mathbf{sel}[m] \Leftrightarrow \mathbf{loc}[\mathbf{alt}[p]] = l_{\text{KILLED}}, \tag{5.17}$$

where $M_{\times} \subseteq M$ denotes the set of kill matches.

Lastly, we need to enforce the value placements appearing in the φ-matches. Let E_M denote this set of value placements, encoded as a tuple $\langle m, b, p \rangle$ for each definition edge between a block b and an operand p appearing in match m. These constraints are then modeled as

$$\forall \langle m, b, p \rangle \in E_M : \mathbf{sel}[m] \Rightarrow \mathbf{dplace}[\mathbf{alt}[p]] = b. \tag{5.18}$$

5.4.3 Experimental Evaluation

We first evaluate the two different methods for value reuse. Based on the results, we then evaluate the impact of value reuse using the superior method.

When filtering, we remove all functions that have fewer than ten LLVM IR instructions and more than 50 instructions. Anything smaller will most likely not be benefited by value reuse, and anything larger will lead to unreasonably long solving times. This leaves a pool of 453 functions, from which we then draw 20 samples.

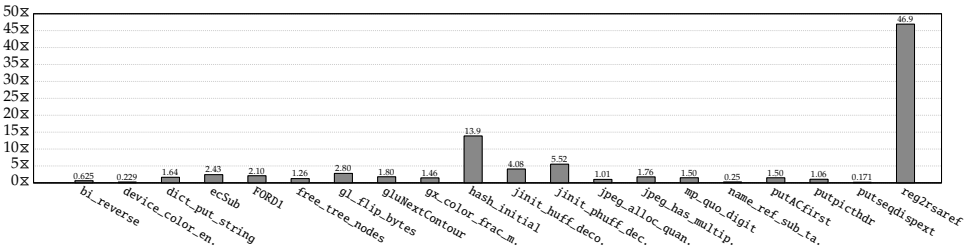


Figure 5.8: Normalized solving times (incl. presolving time) for two constraint models supporting value reuse: one based on match duplication (baseline), and one based on alternative values (subject). GMI: 3.35×, CI [2.24, 4.70]×.

Match Duplication vs. Alternative Values We evaluate the different methods of value reuse by comparing the solving time exhibited by two models: (I) one based on match duplication; and (II) one based on alternative values. Since match duplication yields an exponential increase in number of matches compared to alternative values, we expect model II to perform better than model I.

Figure 5.8 shows the normalized solving times (including presolving time) for the two constraint models described above, with model I as baseline and model II as subject. All functions are solved to optimality. The solving times range from 0.010 s to 1.95 s with a CV of 0.11. The GMI is 3.35× with CI [2.24, 4.70]×.

We see clearly that model II results in considerably shorter solving times than model I. For one function (`reg2rsaref`), the solving time is improved by 46.9⧖. This is due to a high rate of copy-related values for which alternative values results in only 94 matches whereas match duplication results in 537 matches. Hence alternative values is a better design choice over match duplication when implementing value reuse.

Impact of Value Reuse We evaluate the impact of value reuse by comparing the cost (that is, the total number of cycles as described in Sect. 5.6) of the optimal solutions produced by two models: (I) one without value reuse support; and (II) one with this support (based on alternative values). Since value reuse reduces the number of selected copy instructions, we expect model II to produce solutions with less cost compared with model I.

Figure 5.9 shows the normalized solution costs for the two constraint models describe above, with model I as baseline and model II as subject. All functions are solved to optimality. The costs range from 56 cycles to 4476 cycles. The GMI is 1.04× with CI [1.019, 1.07]×.

We see clearly that model II produces solutions with significantly less cost than those produced by model I. For one function (`putseqdispext`), the cycle count is reduced from 256 cycles to 208 cycles (an improvement of 0.231⧖). This is because two constants are frequently used as arguments to function call instructions and

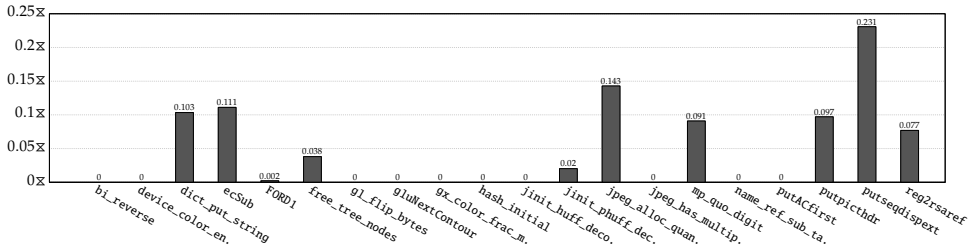


Figure 5.9: Normalized optimal solution costs for two constraint models: one without value reuse support (baseline), and one with such support (subject). GMI: 1.04×, CI: [1.019, 1.07]×.

thus cannot be loaded as immediates. Hence value reuse is essential for reducing the number of copy instructions and, subsequently, lowering cost.

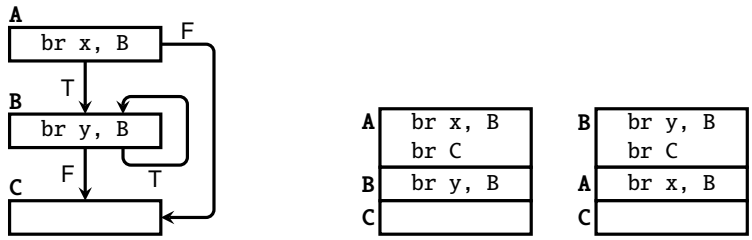
Conclusions From the results for these experiments, we conclude: (i) that alternative values are superior to match duplication; and (ii) that value reuse significantly improves code quality.

5.5 Modeling Block Ordering

Ordering the blocks in a function entails finding a sequence s such that each block appears exactly once in s. Depending on the control-flow instructions selected, some blocks may need to be adjacent. For example, assume a block b that branches to either of two blocks c and d depending on whether a condition holds. Assume also that the conditional branching in b is implemented using an instruction that branches to c if the condition holds, otherwise it continues the execution with the next instruction in the assembly code. This method of branching is called *fall-through*, and due to this d must be placed immediately after b in s.

For some combinations of functions and target machines with fall-through instructions, there exists no valid block sequence without inserting one or more additional jump instructions. See for example Fig. 5.10. Blocks `A` and `B` both contain control-flow instructions that branch to the beginning of `B` if the condition holds, otherwise they branch to block `C` (Fig. 5.10a). Because of the fall-through constraint, `A` and `B` cannot both have `C` as its successor block. Hence an additional jump instruction that directly branches to `C` must be inserted after the control-flow instruction in either `A` or `B` (Fig. 5.10b).

In this dissertation we consider two methods for inserting jump instructions when required: branch extension and dual-target branch patterns. We first introduce the variables and constraints for modeling block ordering before introducing each method in turn, and then present experiments showing that one is superior to the other.



(a) Control-flow graph. (b) Valid block sequences, after jump insertion.

Figure 5.10: Example that requires additional jump instructions. It is assumed that the conditional `br` instruction falls through to the next instruction if the condition is false.

Variables The set of variables $\mathbf{succ}[b] \in B$ models the successor of block b. For example, if $\mathbf{succ}[b] = b'$, then block b' appears immediately after block b in the block ordering sequence.

Constraints A solution to the block ordering problem is a sequence of block successors such that they form a Hamiltonian cycle. Using the circuit constraint defined in Chap. 3 on p. 47, this constraint is modeled as

$$\textsc{circuit}(\mathbf{succ}[b_1], \ldots, \mathbf{succ}[b_n]), \tag{5.19}$$

where $b_1, \ldots, b_n = B$.

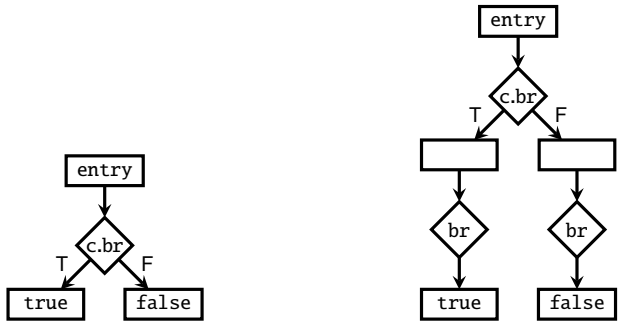
If a match m with an entry block is derived from an instruction that performs a fall-through to block b, then the constraint can naively be modeled as $\mathbf{sel}[m] \Rightarrow \mathbf{succ}[entry(m)] = b$. However, this constraint is too limiting as it disallows empty blocks to be placed between $entry(m)$ and b, thus forcing redundant jump instructions to be emitted. A block b is considered *empty* if either no matches are placed in b or every match in b is a *null match*, which is a match that emits nothing if selected. As empty blocks are not uncommon to appear in the function under compilation – especially when having performed global code motion – this may have a significant impact on code quality.

Hence we extend the naive implementation above into a disjunction, where the second clause models fall-throughs via single empty blocks. Let J denote a set of pairs (m, b), where m is a match and b is a block through which m will fall if selected, and let $M_\perp$ denote the set of null matches. With these definitions, the fall-through constraint is modeled as

$$\begin{gathered}\forall (m, b) \in J : \mathbf{sel}[m] \Rightarrow \mathbf{succ}[entry(m)] = b \vee \\ \left(\mathbf{succ}[\mathbf{succ}[entry(m)]] = b \wedge isEmpty(\mathbf{succ}[entry(m)])\right),\end{gathered} \tag{5.20}$$

where

$$isEmpty(b) \equiv \bigwedge_{o \in O} (\mathbf{oplace}[o] \neq b \vee \mathbf{omatch}[o] \in M_\perp). \tag{5.21}$$



(a) Original UF subgraph. (b) After branch extension.

Figure 5.11: Example of branch extension.

If a block b unconditionally branches to another block b' and b' appears immediately after b in the block sequence, then a jump instruction is redundant. To prevent emission of such jump instructions, we extend the pattern set with a special *null-jump pattern*, with graph structure $b \to c \to b'$, that covers a control node c at zero cost provided that $\mathbf{succ}[b] = b'$.

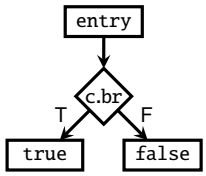
Fall-throughs to or via the function's entry block is never allowed since that block must always be placed first in the block sequence. If b_{F} denotes the function's entry block, then this constraint is modeled as

$$\forall (m, \cdot) \in J : \mathbf{sel}[m] \Rightarrow \mathbf{succ}[entry(m)] \neq b_{\mathrm{F}}. \tag{5.22}$$

5.5.1 Branch Extension

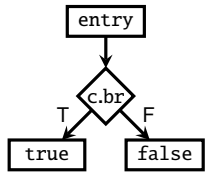
One method of inserting jump instructions is to extend the UF graph with additional block and control nodes. The idea, called *branch extension*, is as follows. For each control-flow edge (c, b), where c is a control node representing a conditional branch and b is a block node, we remove this edge and insert a new block node b', a newcontrol node c', and new control-flow edges such that $c \to b' \to c' \to b$. An example is shown in Fig. 5.11. If the new control node is indeed redundant, then it can be covered by a match derived from the null-jump pattern. This in turn causes the new blocks to become empty and appear immediately before the target block. Like with copy extension, to retain pattern matching we also perform branch extension on each UF graph in the pattern set.

The disadvantage of branch extension is that it inflates the search space. The number of block and control nodes both increase by $O(nk)$, where n is the number of blocks before branch extension and k is the highest number of outbound control-flow edges from a control node in the UF graph. This leads to more operations to be covered and more blocks wherein an operation may be placed. In addition, as the majority of the new blocks will be empty, situations often arise where a control-flow



emit	cost
`br b, true`	2

(a) A pattern that falls through to the `false` block.



emit	cost
`br b, true` *emits*(p_{JMP})	$2 + cost(p_{\text{JMP}})$

(b) New pattern, without fall-through.

Figure 5.12: Example of creating a DTB pattern.

instruction could successfully fall through more than one block. Because of Eq. 5.20, however, it can only fall through at most one empty block, causing emission of redundant jump instructions that would not have been emitted had branch extension not been performed.

5.5.2 Dual-target Branch Patterns

Another method is to extend the pattern set with so-called *dual-target branch (DTB) patterns*. Given a pattern set S, first find a pattern $p_{\text{JMP}} \in S$ corresponding to an unconditional jump instruction that directly branches to a given label (it is reasonable to assume such a pattern always exist for any given target machine). Let $cost(p)$ and $emits(p)$ denote the cost of pattern p respectively the sequence of instructions emitted by p if selected. Then, for each pattern $p \in S$ corresponding to a conditional jump instruction that falls through to a given block b, we add a new pattern to S. This new pattern is a copy of p but has no fall-through constraint, it emits $emits(p)$ followed by $emits(p_{\text{JMP}})$, and it has cost $cost(p) + cost(p_{\text{JMP}})$. An example is shown in Fig. 5.12. Because a DTB pattern has no fall-through constraint, it essentially models a conditional jump instruction capable of directly branching to two blocks (hence the name).

Consequently, if a pattern set contains k patterns with fall-through constraints and a UF graph contains n control nodes representing conditional jumps, then using DTB patterns will enlarge the match set by $O(nk)$ matches. Unlike branch extension, however, the UF graph does not need to be extended with additional blocks wherein operations may be placed, which results in a significantly smaller search space.

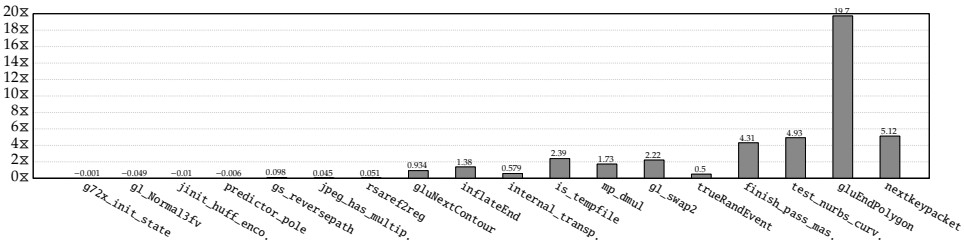


Figure 5.13: Normalized solving times (incl. presolving time) for two constraint models supporting jump insertion: one based on branch extension (baseline), and one based on DTB patterns (subject). GMI: 2.43×, CI [1.57, 3.23]×.

5.5.3 Experimental Evaluation

We evaluate the different methods for inserting jump instructions by comparing the solving times exhibited and the cost of the optimal solutions produced by two versions of the constraint model: (I) one based on branch extension; and (II) one based on DTB patterns. Since branch extension yields a larger number of blocks compared to DTB patterns, we expect model II to outperform model I. Moreover, many of the blocks introduced by branch extension will most likely be empty and thereby cause emission of redundant jump instructions. Consequently, we also expect model II to yield better code quality than model I.

When filtering, we remove all functions that have fewer than 20 LLVM IR instructions and more than 100 instructions. Anything smaller will most likely not require any additional jump instructions, and anything larger will lead to unreasonably long experiment runtimes. This leaves a pool of 413 functions, from which we then draw 20 samples.

When clustering, we replace the number of memory instructions as feature with the number of blocks. This is to evaluate how the methods behave as the number of blocks grow larger.

Impact on Solving Time Figure 5.13 shows the normalized solving times (including presolving time) for the two constraint models described above, with model I as baseline and model II as subject. All functions are solved to optimality and arranged in increasing order of number of conditional jump instructions. The solving times range from 0.020 s to 50.9 s with a CV of 0.14. The GMI is 2.43× with CI [1.57, 3.23]×.

We see clearly that model II results in considerably shorter solving times than model I. For one function (`gluEndPolygon`), the solving time is improved by 19.7⧖. We also observe that, as expected, when the number of conditional jump instructions increases – up to 13, in the case of `nextkeypacket` – the search space for model I grows faster than for model II. Hence, in terms of solving time DTB patterns is a

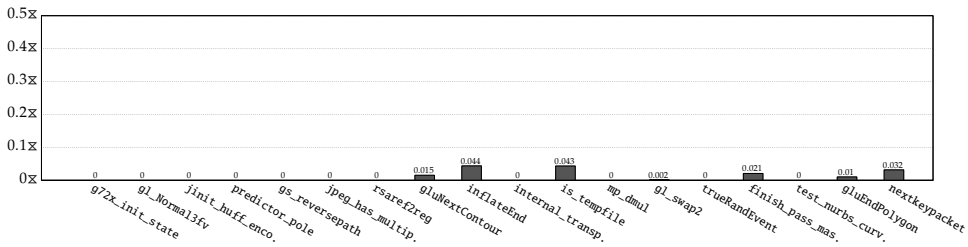


Figure 5.14: Normalized optimal solution costs for two constraint models supporting jump insertion: one based on branch extension (baseline), and one based on DTB patterns (subject). GMI: 1.01×, CI [1.003, 1.02]×.

better design choice over branch extension when implementing insertion of jump instructions.

Impact on Code Quality Figure 5.14 shows the normalized solution costs for the two constraint models described above, with model I as baseline and model II as subject. All functions are solved to optimality and arranged in increasing order of number of conditional jump instructions. The costs range from 56 cycles to 17 991 cycles. The GMI is 1.01× with CI [1.003, 1.02]×.

We see clearly that model II produces optimal solutions with slightly lower cost compared those produced by model I (up to 0.0437⧖ improvement). Hence, in terms of code quality DTB patterns is a better design choice over branch extension when implementing insertion of jump instructions.

Conclusions From the results for these experiments, we conclude that DTB patterns are superior to branch extension, both in terms of solving time and code quality.

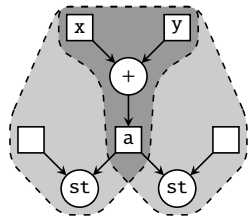
5.6 Objective Function

During instruction selection, most compilers optimize for performance by using execution latencies of the instructions as match costs. In addition, they factor in the execution frequency of the block in which the selected matches are placed. The intuition is that the instructions selected for "hot" blocks – for example, those belonging to a loop with many iterations – will have greater impact than those selected for scarcely executed blocks. In this chapter we introduce a straightforward but naive implementation of the objective function, which is refined in the next chapter.

Variables The cost variable **cost** $\in \mathbb{N}$ models the total cost of the selected matches. It is assumed the total cost can never be negative.

```
a ← x + y
store a, ...
...
store a, ...
```

(a) Code snippet.



(b) UF subgraph, covered by two matches derived from a store instruction with base-plus-index addressing mode. For brevity, the state nodes are not included.

Figure 5.15: Example illustrating when recomputation is preferred over value reuse.

Constraints A straightforward implementation of the objective function described above can be modeled as

$$\mathbf{cost} = \sum_{m \in M} \mathbf{sel}[m] \times cost(m) \times freq(blockOf(m)), \tag{5.23}$$

where $cost(m) \in \mathbb{N}$ denotes the cost of match m, $freq(b) \in \mathbb{N}$ denotes the execution frequency of block b,[3] and

$$blockOf(m) \equiv \begin{cases} \mathbf{oplace}[min(covers(m))] & \text{if } covers(m) \neq \emptyset, \\ \mathbf{dplace}[\mathbf{alt}[min(defines(m))]] & \text{otherwise.} \end{cases} \tag{5.24}$$

5.7 Limitations

The constraint model described in this chapter has several limitations that affect code quality. The first limitation concerns recomputation of values, the second concerns if-conversion, and the third concerns implicit sign and zero extensions and truncations. Consequently, a solution that is considered optimal with respect to this model may still be inferior to the code produced by an instruction selector without these limitations.

Recomputation For some combinations of functions and target machines, it may be beneficial to *recompute* values appearing in common subexpressions instead of reusing them. See for example Fig. 5.15. The function performs two memory stores using the same address value `a` (Fig. 5.15a). If the target machine provides a memory instruction with base-plus-index addressing mode (that is, the address to be used is the sum of the values appearing in a base and an index register), then no add instruction is needed for computing `a`. In the context of instruction

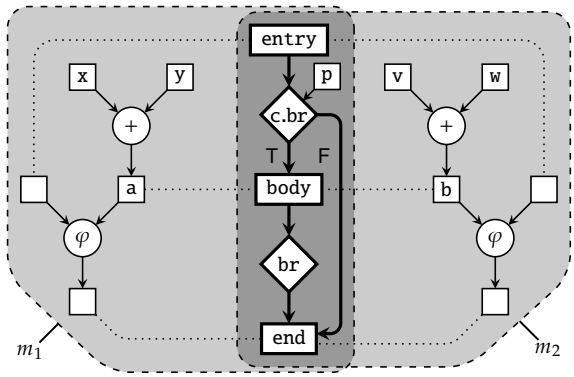
[3] In order to curb the size of the domain of the cost variable, the execution frequencies must often be scaled down. In our experiments, limiting the execution frequencies to 1000 proved sufficient.

```
entry:
  ...
  if p goto body
  else goto end;
body:
  a ← x + y
  b ← v + w
end:
  ...
```

(a) Code snippet.

```
entry:
  ...
  if (p) add a, x, y
  if (p) add b, v, w
  ...
```

(b) Assembly code with predicated instructions.



(c) UF subgraph, covered by two matches derived predicated add instructions.

Figure 5.16: Example of if-conversions.

selection, this means letting the operation representing the addition to be covered by more than one match (Fig. 5.15b). However, Eqs. 5.1 and 5.2 require that every operation and datum is covered respectively defined by exactly one selected match, thus preventing such solutions. Although these constraints can be relaxed to allow operations and data to be covered respectively defined by at least one selected match, many of the solving techniques introduced in this dissertation rely on exact coverage.

If-Conversions In most target machines, performing a branch incurs a performance penalty. Some architectures therefore allow the instructions to be predicated with a Boolean flag for optional execution, which allows functions with if-then-else structures to be transformed into linear code. This process is called *if-conversion*.

Although the universal representation enables predicated versions of the instructions to be captured as patterns, selection of such patterns is typically prevented by the constraint model. See for example Fig. 5.16. Assume that a function contains two sums, `a` and `b`, which are conditionally computed given a certain predicate `p` (Fig. 5.16a). Because this constitutes an if-then-else structure, this code snippet is eligible for if-conversion (Fig. 5.16b). Representing the predicated versions of add instructions as patterns gives rise to two matches, m_1 and m_2, which can collectively cover the computation and control nodes in the corresponding UF graph (Fig. 5.16c). However, since m_1 and m_2 both cover the same control nodes, only one of the matches can be selected (due to Eqs. 5.1 and 5.2). But because both matches consume the `body` block, no other operations may be placed in `body` if either is selected (due to Eq. 5.8). This means that either both or none of the matches must

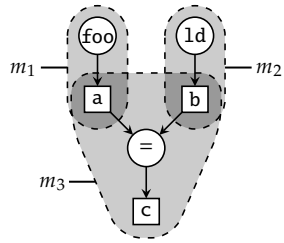


Figure 5.17: Example of implicit sign or zero extensions.

be selected. Together with the constraint of exact coverage, in all solutions neither of m_1 or m_2 is selected. This problem can be fixed by relaxing the constraint of exact coverage, but – as in the case of recomputation – this inhibits many of the solving techniques introduced in this dissertation.

Implicit Sign or Zero Extensions Depending on the hardware, the constraint model may produce code with redundant instructions in cases where the function contains sign or zero extensions. See for example Fig. 5.17, which depicts a UF subgraph coverable by matches m_1, m_2, and m_3. Assume that `foo` represents a function call and that `a`, `b`, and `c` represent 8-bit values stored in 32-bit registers. As is common, m_3 is derived from an instruction that checks whether the full contents of two registers are equal. Consequently, as a precaution the upper bytes of the registers need to be zero-extended (that is, those bits are all set to 0) before doing the comparison. Since nothing can be assumed about the value returned by `foo`, this is certainly necessary for the register of `a`. However, it may be redundant for the register of `b`. For example, m_2 may be derived from a single-byte load instruction that clears the entire register before loading the value. But since this information is lost in the constraint model, m_3 must assume that both registers need to be zero-extended.

One solution to this problem is to extend the pattern set with additional patterns that capture these situations. For example, merging the patterns of m_2 and m_3 results in a match which, if selected, emits the instruction of m_2 followed by the instructions of m_3 without the redundant zero extension of `b`. Depending on the instruction set, however, this may result in an exponential number of patterns. If the instruction set contains n instructions with implicit extensions and m instructions that each takes k values which must first be sign- or zero-extended, then this will result in $O(n^k m)$ additional patterns.

Another solution is to apply the same mechanism used in copy extension. In the same manner as with copy nodes, the UF graph is first extended with *extension nodes*. Hence, for each data-flow edge $v \to o$, where v is a value node and o is an operation, we remove this edge and insert a new extension node e, a newvalue node v', and new data-flow edges such that $v \to e \to v' \to o$. Then the constraint

model is extended with two sets of variables, $\mathbf{sext}[d] \in \{0,1\}$ and $\mathbf{zext}[d] \in \{0,1\}$, denoting whether a value has been sign- respectively zero-extended. We also extend the pattern set with a special *null-extend pattern*, with graph structure $v \rightarrow e \rightarrow v'$, that covers e at zero cost provided that $(\mathbf{sext}[v] \vee \neg\mathbf{sext}[v']) \wedge (\mathbf{zext}[v] \vee \neg\mathbf{zext}[v'])$ holds. Obviously, a match derived from the null-extend pattern emits nothing if selected. If the null-extend pattern cannot be selected for covering a particular extension node, then this means an appropriate extension instruction must be emitted.

5.8 Summary

In this chapter, we have introduced a constraint model that integrates the problems of global instruction selection, global code motion, data copying, value reuse, and block ordering. When multiple design choice exist for a given task, we have performed a thorough evaluation to decide which design choice is better. We have also discussed the limitations of this model and how these affect the assembly code that can be produced. A full implementation of the model, written in MiniZinc, is available in Ap. G.

CHAPTER

6

Solving Techniques

This chapter introduces the techniques applied for improving solving of the constraint model introduced in the previous chapter. We begin in Sects. 6.1 and 6.2 with refining the constraint model to strengthen propagation. We continue to augment the model in Sects. 6.3 and 6.4 by adding implied, symmetry breaking, and dominance breaking constraints. We then describe techniques for tightening the cost bounds in Sect. 6.5 and describe branching strategies in Sect. 6.6. In Sect. 6.7, we describe presolving techniques for removing matches that, for one reason or another, can be removed before instantiating the model. With all solving techniques in place, we then evaluate the impact of these techniques, both individually and as groups, in Sect. 6.8. Lastly, a summary is given in Sect. 6.9.

6.1 Refining the Define-Before-Use Constraint

In Chap. 5 on p. 81, a simple but naive implementation is used for implementing the constraint that all data must be defined before used (Eq. 5.13). To begin with, it requires the use of set variables,[1] which are expensive to use and not necessarily supported by all constraint solvers. This is for example the case of the constraint solver used in the experiments, thus preventing us from evaluating the impact of the naive implementation. In addition, if we know in which blocks a datum is used, then many implied constraints can be applied to strengthen propagation.

We first eliminate the set variables by capturing the information in the *dom* function as a dominance relation matrix

$$\left[\langle b_1, b_2 \rangle \mid b_1, b_2 \in B, b_1 \in dom(b_2) \right] \tag{6.1}$$

where each row denotes the fact that a block b_1 is dominated by another block b_2. Next, we introduce the variables needed for capturing uses of data.

[1] This is because the **dplace** variables need to be members of the *dom* function, which is implemented as an array into which the **oplace** variables are used as indices.

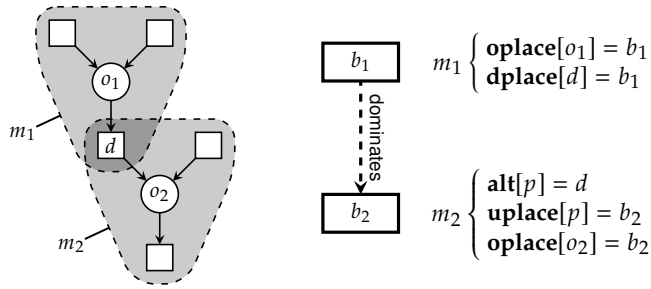


Figure 6.1: Example illustrating the refined define-before-use constraint. The assignments on the right-hand side show how these variables should be set if the two matches m_1 and m_2 are placed in blocks b_1 respectively b_2, where b_1 is assumed to dominate b_2.

Variables The set of variables $\mathbf{uplace}[p] \in B$ models in which block the datum connected to operand p is used. Note that unlike the **dplace** variables, which are indexed using a datum, the **uplace** variables are indexed using an operand.

Constraints Obviously, every use of data must be dominated by its definition. This constraint is modeled as

$$\forall p \in P_{\overline{\varphi}} : \text{TABLE}(\langle \mathbf{uplace}[p], \mathbf{dplace}[\mathbf{alt}[p]] \rangle, R), \tag{6.2}$$

where $P_{\overline{\varphi}} \subseteq P$ denotes the set of operands not appearing in any φ-match and R is the dominance relation matrix computed using Eq. 6.1. We exclude such operands for the same reasons φ-matches are excluded in Eq. 5.6.

Next, if a match m is selected and placed in block b, then all uses of data made by m must also occur in b. This is modeled as

$$\forall m \in M_{\overline{\varphi}}, \forall o \in \mathit{covers}(m), \forall p \in \mathit{uses}(m) : \mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o] = \mathbf{uplace}[p]. \tag{6.3}$$

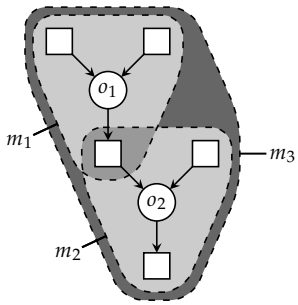
An example illustrating the interaction between Eq. 6.2 and Eq. 6.3 is shown in Fig. 6.1.

Due to Eq. 6.3, the assignment to the **uplace** variables for non-selected matches does not matter as long as Eq. 6.2 is satisfied. This gives rise to symmetric solutions. To break these symmetries, we fix the assignments in such cases to the block wherein the datum is defined as a block always dominates itself. This is modeled as

$$\forall m \in M_{\overline{\varphi}}, \forall p \in \mathit{uses}(m) : \neg\mathbf{sel}[m] \Rightarrow \mathbf{uplace}[p] = \mathbf{dplace}[\mathbf{alt}[p]], \tag{6.4}$$

where $M_{\overline{\varphi}} \subseteq M$ denotes the set of matches without the φ-matches.

Since neither of the above constraints apply to operands belonging to φ-matches, the assignment to their **uplace** variables does not matter, again giving rise to



(a) UF graph.

op	match	block	opcost
o_1	m_1	b_1	$4 \times 10 = 40$
o_1	m_1	b_2	$4 \times 1 = 4$
o_1	m_3	b_1	$3 \times 10 = 30$
o_1	m_3	b_2	$3 \times 1 = 3$
o_2	m_2	b_1	$1 \times 10 = 10$
o_2	m_2	b_2	$1 \times 1 = 1$
o_2	m_3	b_1	$2 \times 10 = 20$
o_2	m_3	b_2	$2 \times 1 = 2$

(b) Cost matrix.

Figure 6.2: Example of match costs distributed over operations. It is assumed that matches m_1, m_2, and m_3 have costs 4, 1, and 5, respectively, and that they can be placed in one of two blocks, b_1 and b_2, with execution frequencies 10 and 1, respectively. The cost of m_3 distributed over o_1 and o_2 is 3 and 2, respectively.

symmetric solutions. We therefore fix the assignment in such cases, which is modeled as

$$\forall p \in P_\varphi : \mathbf{uplace}[p] = min(B). \tag{6.5}$$

6.2 Refining the Objective Function

The straightforward implementation of the objective function (Eq. 5.23) is naive because it fails to reason on how cost is distributed across the operations that need to be covered. This in turn results in poor propagation. See for example Fig. 6.2. Assume that a UF graph can be covered by three matches m_1, m_2, and m_3 with costs 4, 1, and 5, respectively (Fig. 6.2a). Assume further that these matches can be placed in one of two blocks, b_1 and b_2, with execution frequencies 10 and 1, respectively. Because Eq. 5.23 is modeled as a summation, it can only propagate the bounds of the cost variable. Consequently, any match m for which $\mathbf{sel}[m]$ is still undecided incurs a cost between zero (if not selected) and $cost(m) \times max(\cup_{b \in B} freq(b))$ (if selected and placed in the block with highest execution frequency). In the example above, this means the cost variable is initially bounded as $0 \leq \mathbf{cost} \leq 100$. These are very weak bounds as we know that either both m_1 and m_2 are selected and placed in b_2, or m_3 is selected and placed in b_2, resulting in a cost of at least 5. Also, in the worst case all selected matches are placed in b_1, resulting in a cost of at most 50.

Instead of reasoning about the cost incurred by the matches – which may or may not be selected – a better approach is to deduce the cost incurred on the operations – which must always be covered. The idea is as follows. First, for each match m, evenly divide the cost of m over each operation o covered by m. If a strict partial order $<$ exists over the set of operations, and $covers(m)$ returns an ordered list that

can be indexed starting from 1, then the cost can be computed as

$$cost(m,o) = \begin{cases} q+1 & \text{if } o < covers(m)[r+1], \\ q & \text{otherwise,} \end{cases} \tag{6.6}$$

where $q = \lfloor cost(m)/|covers(m)| \rfloor$ and $r = cost(m) \mod |covers(m)|$. Consequently, for any match m, $cost(m) = \sum_{o \in covers(m)} cost(m,o)$. Then, for each block b we multiply $cost(m,o)$ with the execution frequency of b. This information can be represented as a cost matrix

$$\left[\ o, m, b, \big(cost(m,o) \times freq(b)\big) \ \middle|\ m \in M, o \in covers(m), b \in B\ \right] \tag{6.7}$$

where each row denotes the cost of an operation o if covered by a match m and placed in a block b. From the cost matrix given in Fig. 6.2b, we can deduce that the cost of covering operations o_1 and o_2 is between 3 and 40 and between 1 and 20, respectively. Hence the total cost can be bounded as $4 \leq \mathbf{cost} \leq 60$, which is a much tighter bound compared to that achieved using the naive objective function.

An alternative method for computing the cost is to *first* multiply the match cost with the execution frequency and *then* evenly divide the product over the covered operations. We call the first method the *divide-then-multiply method*, and the second method the *multiply-then-divide method*. For the latter, the cost matrix is computed as

$$\left[\ \langle o, m, b, cost(m,o,b)\rangle \ \middle|\ m \in M, o \in covers(m), b \in B\ \right] \tag{6.8}$$

where

$$cost(m,o,b) = \begin{cases} q+1 & \text{if } o < covers(m)[r+1], \\ q & \text{otherwise,} \end{cases} \tag{6.9}$$

where $q = \lfloor d/|covers(m)| \rfloor$, $r = d \mod |covers(m)|$, and $d = cost(m) \times freq(b)$. Consequently, for any match m and block b, $cost(m,o) \times freq(b) = \sum_{o \in covers(m)} cost(m,o,b)$.

At first glace this design decision would appear to make no difference, but they unexpectedly exhibit significantly different solving time characteristics, as will be seen later in the experimental evaluation.

Variables The set of variables $\mathbf{ocost}[o] \in \mathbb{N}$ models the cost incurred by covering operation o. It is assumed the domain is the same as for the cost variable.

Constraints For each operation o, the combination o, $\mathbf{omatch}[o]$, $\mathbf{oplace}[o]$, $\mathbf{ocost}[o]$ must appear as a row in the cost matrix. Given a cost matrix C computed using either Eq. 6.7 or 6.8, this constraint is modeled as

$$\forall o \in O : \textsc{table}(\langle o, \mathbf{omatch}[o], \mathbf{oplace}[o], \mathbf{ocost}[o]\rangle, C). \tag{6.10}$$

The total cost is then modeled as

$$\mathbf{cost} = \sum_{o \in O} \mathbf{ocost}[o]. \tag{6.11}$$

Note that the cost computed by Eq. 6.11 is exactly the same as that computed by Eq. 5.23. The proof is as follows. Because every operation must be covered by exactly one selected match, there is a one-to-one correspondence between a selected match m and the set Q of operations covered by m. Consequently, the total cost incurred by m – which is $cost(m) \times freq(b)$, where b is the block wherein m is placed – should be exactly the same as that incurred by the operations in Q. Due to Eq. 6.11, the total cost incurred by the operations in Q is $\sum_{o \in Q} \mathbf{ocost}[o]$. Due to Eqs. 6.7 and 6.10, we know that for each $o \in Q$, $\mathbf{ocost}[o] = cost(m, o) \times freq\, b$. Since $cost(m) = \sum_{o \in covers(m)} cost(m, o)$, the total cost incurred by operations in Q is $cost(m) \times freq(b)$. □

6.3 Implied Constraints

As explained in Chap. 3, implied constraints are constraints that strengthen the propagation while preserving all solutions. Stronger propagation leads to less search, which in turn leads to shorter solving times. In this section, we introduce such constraints that are relevant for the model.

6.3.1 Implied Operation and Data Placements

Due to Eqs. 5.6 and 5.7, if a selected match defines some datum d_1 and uses some other datum d_2, then the block wherein d_2 is defined must dominate the block wherein d_1 is defined. From this observation, we can infer that if all matches covering a non-φ-node operation o do not span any blocks, define some datum d_1, and use some datum d_2, then the block wherein d_2 is defined must dominate the block wherein d_1 is defined. In addition, o must be placed in the same block wherein d_1 is defined. This is modeled as

$$\begin{array}{c} \forall o \in \{o' \mid o' \in O_{\overline{\varphi}}, m \in M_{o'} \text{ s.t. } consumes(m) = \varnothing\}, \\ \forall d_1 \in \{d \mid d \in dataOf(o, defines), m \in M_o, \exists p \in defines(m) : D_p = \{d\}\}, \\ \forall d_2 \in \{d \mid d \in dataOf(o, uses), m \in M_o, \exists p \in uses(m) : D_p = \{d\}\} : \\ \textsc{table}(\langle \mathbf{dplace}[d_1], \mathbf{dplace}[d_2] \rangle, R) \wedge \mathbf{oplace}[o] = \mathbf{dplace}[d_1]. \end{array} \tag{6.12}$$

where

$$dataOf(o, f) \equiv \bigcup_{\substack{m \in M_o,\, p \in f(m) \text{ s.t.} \\ covers(m) = \{o\}}} D_p \tag{6.13}$$

From the same observation, we can also infer that if all matches covering the same non-φ-node operation span a set S of blocks and define some datum d, then d must be defined in one of the blocks in S. This is modeled as

$$\forall S \in 2^B, \forall d \in D, \forall o \in \left\{ o' \;\middle|\; \begin{array}{l} o' \in O_{\overline{\varphi}}, m \in M_{o'}, \exists p \in defines(m) : \\ spans(m) = S \wedge D_p = \{d\} \end{array} \right\} : \mathbf{dplace}[d] \in S. \tag{6.14}$$

From Eq. 5.5, we can infer that if all non-φ-matches covering operation o have entry block b, then o must for sure be placed in b. This is modeled as

$$\begin{gathered}\forall b \in B, \forall o \in \{o' \mid o' \in O, m \in M_{o'} \setminus M_{\varphi} \text{ s.t. } \mathit{entry}(m) = \{b\}\} : \\ \mathbf{oplace}[o] = b.\end{gathered} \tag{6.15}$$

From the same constraint, we can also infer that if the matches covering the same non-φ-node operation all have identical entry blocks, say b, and make use of some datum d, then the block wherein d is defined must dominate b. This is modeled as

$$\begin{gathered}\forall b \in B, \forall d \in \left\{ d' \;\middle|\; \begin{array}{l} o' \in O_{\overline{\varphi}}, m \in M_{d'}, \exists p \in \mathit{uses}(m) : \\ \mathit{entry}(m) = \{b\} \wedge D_p = \{d\} \end{array} \right\} : \\ \textsc{table}(\langle b, \mathbf{dplace}[d] \rangle, R).\end{gathered} \tag{6.16}$$

From Eq. 5.9, we can infer that if a datum d appears in a definition edge $d \longrightarrow b$ and is defined by φ-matches only, then the operation covered by these matches must be placed b. This is modeled as

$$\forall d \longrightarrow b \in E, \forall o \in \{o' \mid m \in M_d \cap M_{\varphi}, o' \in \mathit{covers}(m)\} : \mathbf{oplace}[o] = b. \tag{6.17}$$

It is assumed that the edges in E have been reoriented such that all sources are either state or value nodes and all targets are block nodes.

6.3.2 Implied Constraints Due to the Define-Before-Use Refinement

From Eqs. 6.2–6.5, we can infer the following implied constraints.

If a non-φ-match m spanning no blocks is selected, then all data used and defined by m must take place in the same block. This is modeled as

$$\begin{gathered}\forall m \in \{m' \mid m \in M_{\overline{\varphi}}, \mathit{spans}(m) = \varnothing\}, \forall p_1, p_2 \in \mathit{uses}(m) \text{ s.t. } p_1 < p_2 : \\ \mathbf{sel}[m] \Rightarrow \mathbf{uplace}[p_1] = \mathbf{uplace}[p_2],\end{gathered} \tag{6.18}$$

$$\begin{gathered}\forall m \in \{m' \mid m \in M_{\overline{\varphi}}, \mathit{spans}(m) = \varnothing\}, \forall p_1, p_2 \in \mathit{defines}(m) \text{ s.t. } p_1 < p_2 : \\ \mathbf{sel}[m] \Rightarrow \mathbf{dplace}[\mathbf{alt}[p_1]] = \mathbf{dplace}[\mathbf{alt}[p_2]],\end{gathered} \tag{6.19}$$

$$\begin{gathered}\forall m \in \{m' \mid m \in M_{\overline{\varphi}}, \mathit{spans}(m) = \varnothing\}, \forall p_1 \in \mathit{uses}(m) \setminus \mathit{defines}(m), \\ \forall p_2 \in \mathit{defines}(m) : \mathbf{sel}[m] \Rightarrow \mathbf{uplace}[p_1] = \mathbf{dplace}[\mathbf{alt}[p_2]].\end{gathered} \tag{6.20}$$

In addition, if a non-φ-match spanning some blocks is selected, then all uses of its input data must occur in the same block. This is modeled as

$$\begin{gathered}\forall m \in \{m' \mid m \in M_{\overline{\varphi}}, \mathit{spans}(m) \neq \varnothing\}, \\ \forall p_1, p_2 \in \mathit{uses}(m) \setminus \mathit{defines}(m) \text{ s.t. } p_1 < p_2 : \\ \mathbf{sel}[m] \Rightarrow \mathbf{uplace}[p_1] = \mathbf{uplace}[p_2].\end{gathered} \tag{6.21}$$

6.3.3 Implied Data Locations

From Eq. 5.10, we can infer several implied constraints.

If all non-kill matches covering some operation require some non-state datum d as input, then d cannot be an intermediate value nor be killed. Such data is said to be *available*, meaning they cannot be located in either l_{INT} or l_{KILLED}. If the input can be one of several values (due to alternative values), then at least one of those values must be made available. This is modeled as

$$\begin{gathered}\forall S \in 2^{D_{\overline{\square}}}, \forall o \in \{o' \mid o' \in O, m \in M_{o'}, \exists p \in uses(m) \setminus defines(m) : D_p = S\}, \\ \exists d \in S : \mathbf{loc}[d] \notin \{l_{\text{INT}}, l_{\text{KILLED}}\},\end{gathered} \tag{6.22}$$

where $D_{\overline{\square}} \subseteq D$ denotes the set of data without the state nodes.

If all non-kill matches defining a non-state datum d have d as an exterior value, then d must be made available. This is modeled as

$$\begin{gathered}\forall d \in \left\{ d' \;\middle|\; \begin{array}{l} d' \in D_{\overline{\square}}, m \in M_{d'} \setminus M_{\times}, \exists p \in defines(m) : \\ D_p = \{d'\} \wedge isExt(m, p) \end{array} \right\}, \\ \mathbf{loc}[d] \notin \{l_{\text{INT}}, l_{\text{KILLED}}\},\end{gathered} \tag{6.23}$$

where $isExt(m, p)$ denotes whether an operand p in a match m represents an exterior value.

We can always constrain the locations of a non-state datum d to those locations where the definers can put d. The intuition here is to take the union of all those locations, which is modeled as

$$\begin{gathered}\forall d \in D_{\overline{\square}}, \forall S \in 2^{L \cup \{l_{\text{INT}}, l_{\text{KILLED}}\}} \text{ s.t.} \\ S = \{l \mid m \in D_d \setminus M_{\times}, p \in defines(m), l \in stores(m, p) \text{ s.t. } d \in D_p\} : \\ \mathbf{loc}[d] \in S.\end{gathered} \tag{6.24}$$

Likewise, we can always constrain the locations of a non-state datum d to those locations where the users can access d. Assuming there is always at least one match making use of d), this is similarly modeled as

$$\begin{gathered}\forall d \in D_{\overline{\square}}, \forall S \in 2^{L \cup \{l_{\text{INT}}, l_{\text{KILLED}}\}} \text{ s.t.} \\ S = \{l \mid m \in M_{\overline{\times}}, p \in uses(m), l \in stores(m, p) \text{ s.t. } d \in D_p\} \wedge S \neq \varnothing : \\ \mathbf{loc}[d] \in S.\end{gathered} \tag{6.25}$$

6.3.4 Implied Fall-Throughs

Due to Eq. 5.20, if for any two blocks b_1 and b_2 there exists a match requiring b_2 to follow b_1 but there are no matches requiring any other block to follow b_1 nor requiring b_2 to follow any other block, then it is always safe to force b_2 to follow b_1. This is modeled as

$$\begin{gathered}\forall b_1, b_2 \in B \text{ s.t. } \{entry(m) \mid (m, b_2) \in J\} = \{b_1\} \wedge \\ \{b \mid (m, b) \in J \text{ s.t. } entry(m) = \{b_1\}\} = \{b_2\} : \mathbf{succ}[b_1] = b_2.\end{gathered} \tag{6.26}$$

6.4 Symmetry and Dominance Breaking Constraints

As explained in Chap. 3, symmetry and dominance breaking constraints are constraints that remove solutions from the search space that are either symmetric to one another or dominated by some other solution. Since this leads to a smaller search space, the solving time is reduced. In this section, we introduce such constraints that are relevant for the model.

6.4.1 Location of State Nodes

Since data also includes the state nodes, a **loc** variable will be introduced for every state node. However, since state nodes are abstract entities used only to capture implicit dependencies between certain operation, the assignment to these variables has no impact on code quality. Hence many symmetric solutions arise. We remove these symmetries by always fixing the location for each state node, which is modeled as

$$\forall d \in D_\square : \mathbf{loc}[d] = l_{\text{INT}}, \tag{6.27}$$

where $D_\square \subseteq D$ denotes the set of state nodes.

6.4.2 Operands of Non-Selected Matches

The **alt** variables of matches that are not selected still need to be assigned a value. Since this assignment has no impact on code quality, it gives rise to many symmetric solutions. We therefore fix the **alt** assignments in such cases, which is modeled as

$$\forall m \in M, \forall p \in \mathit{defines}(m) \cup \mathit{uses}(m) : \neg\mathbf{sel}[m] \Rightarrow \mathbf{alt}[p] = \mathit{min}(D_p). \tag{6.28}$$

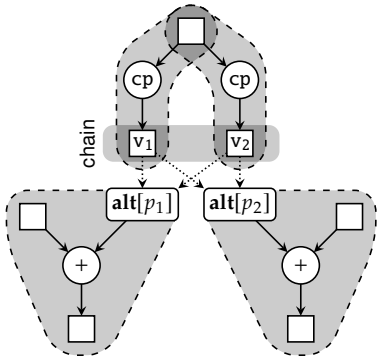
The symmetry breaking constraint above also implies that if an operand representing input with multiple data does not take its minimum value, then the corresponding match must be selected. In addition, the corresponding datum must be made available. This is modeled as

$$\begin{gathered}\forall m \in M, \forall p \in \mathit{uses}(m) \setminus \mathit{defines}(m) \text{ s.t. } |D_p| > 1 : \\ \mathbf{alt}[p] \neq \mathit{min}(D_p) \Rightarrow \mathbf{alt}[p] \notin \{l_{\text{INT}}, l_{\text{KILLED}}\}.\end{gathered} \tag{6.29}$$

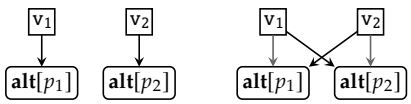
6.4.3 Interchangeable Data

As described in Chap. 5 on p. 77, data in the UF graph that are copies of the same value are copy-related and therefore interchangeable. This is another source for symmetric solutions, which is illustrated in Fig. 6.3.

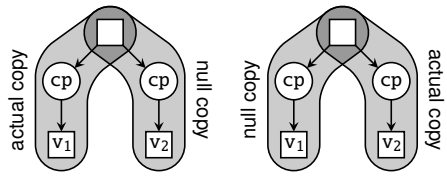
Assume that a UF graph contains two copy-related values, $\mathtt{v}_1$ and $\mathtt{v}_2$, which may both be connected to two operands p_1 and p_2 (Fig. 6.3a). We say that a set of values constitute a chain of *interchangeable data* if they can be swapped in a solution without affecting the program semantics. This is the case if the values are all copy-related and none are both defined and used by some match. In the above example, $\mathtt{v}_1$



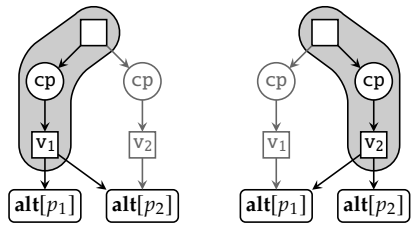
(a) UF graph, where the values v_1 and v_2 constitute a chain of interchangeable data.



(b) Symmetries due to how data can be connected to operands.



(c) Symmetries due to how null-copy matches can be selected.



(d) Symmetries due to how kill matches can be selected.

Figure 6.3: Example of interchangeable data and how these give rise to symmetries.

and v_2 constitute such a chain and can therefore be swapped for p_1 and p_2, giving rise to unwanted symmetric solutions (Fig. 6.3b). The intuition here is to prevent solutions containing "cross-over" connections between the values in a chain and the **alt** variables. As a precaution, however, we exclude operands used by φ-matches which may require such cross-over connections due to the definition edges.

If we assume that there exists a partial order $\leq$ for D, then we can remove these symmetries by enforcing an order of the values assigned to the **alt** variables. To this end, we use the value-precede-chain constraint introduced in Chap. 3 on p. 48. Let I denote the set of chains of interchangeable data and $P_{\lceil}\varphi] \subseteq P$ denote the set of operands not used by φ-matches. With these definitions, this constraint is modeled as

$$\forall c \in I, \forall p_1, \ldots, p_k \in P_{\lceil}\varphi] \text{ s.t. } p_1 \neq \cdots \neq p_k \wedge (\forall 1 \leq i \leq k : D_{p_i} = c) : \\ \textsc{vpc}(c, \mathbf{alt}[p_1], \ldots, \mathbf{alt}[p_k]). \tag{6.30}$$

Additional symmetries may appear due to null-copy matches. Returning to the example above, if one of the two copy nodes needs to be covered using a copy match derived from actual copy instruction, then we are free to decide which. Intuitively, we want to prevent solutions where selected null-copy matches "appear to the left" of a non-null-copy match. Let $I_\circ$ denote the set of chains of data that can only be defined by copy matches, $M_\Phi \subseteq M$ denote the set of null-copy matches, and

$M_d \subseteq M$ denote the set of matches that can define a datum d. Assuming there exists exactly one null-copy match to cover each copy node, this is modeled as

$$\begin{gathered}\forall c \in I_\circ, \forall 1 \leq i < k, \exists m_i \in M_{c[i]} \cap M_\Phi : \\ \textsc{increasing}(\mathbf{sel}[m_1], \ldots, \mathbf{sel}[m_k]),\end{gathered} \tag{6.31}$$

where

$$\textsc{increasing}(\mathbf{x}_1, \ldots, \mathbf{x}_k) \equiv \bigwedge_{1 \leq i < k} \mathbf{x}_i \leq \mathbf{x}_{i+1}. \tag{6.32}$$

Similarly to null-copy matches, symmetries can also arise due to kill matches. Intuitively, we want to prevent solutions where killed data "appear to the right" of non-killed data. Assuming there exists exactly one kill match to cover each copy node, this is modeled as

$$\begin{gathered}\forall c \in I_\circ, \forall 1 \leq i < k, \exists m_i \in M_{c[i]} \cap M_\times : \\ \textsc{increasing}(\mathbf{sel}[m_1], \ldots, \mathbf{sel}[m_k]),\end{gathered} \tag{6.33}$$

where $M_\times$ denotes the set of kill matches.

6.5 Tightening the Cost Bounds

As explained in Chap. 3, in CP optimization problems are solved using branch and bound. In other words, when a solution is found a constraint is added to the model, forcing any subsequently found solutions to be strictly better. For this to be effective, however, the cost bounds must be tight. A tight upper bound enables the constraint solver to prune away parts of the search space that only contains inferior solutions. This is most useful for proving whether a particular solution is optimal. Likewise, a tight lower bound enables the constraint solver to prune away parts of the search space that contains no solutions. This is partially achieved by the refined objective function introduced in Sect. 6.2, but the bounds can be further tightened using complementary mechanisms.

The upper bound can be further tightened by solving the same problem using a greedy but fast heuristic. To this end, any modern compiler can be used. The lower bound can be further tightened by solving a relaxed – and thereby simpler – version of the constraint model. In this context, a relaxed version corresponds to a model that only integrates global instruction selection and block ordering. Hence the relaxed model consists of only the **omatch**, **opcosts**, **sel**, **succ**, and **cost** variables, Eq. 5.1, relaxed versions of Eqs. 5.19 and 5.20 that allow fall-throughs via non-empty blocks, and modified versions of Eqs. 6.6, 6.7, and 6.11 that are optimistic about the execution frequencies.

If C_{RLX} and C_{HEUR} denote the cost computed from the relaxation and by the heuristic, respectively, then the cost variable is bounded as

$$C_{\text{RLX}} \leq \mathbf{cost} < C_{\text{HEUR}}. \tag{6.34}$$

6.6 Branching Strategies

As explained in Chap. 3, the branching strategy decides how to explore the search space. Following the first-fail principle (see Sect. 3.2.2 on p. 52), we first branch on the **ocost** variables. When branching on these variables, we select the variable v with largest difference between the two smallest values in its domain – this is typically called the maximum *regret* [334] – and the smallest value in the domain of v. The intuition here is that, because the objective function strives to minimize the total cost, we wish to minimize the cost incurred per operation. Hence we try to cover the operations for which the cost of bad decisions is largest, and we try to do so at least cost.[2] Note that this branching strategy is only possible due to the refined objective function introduced in Sect. 6.2. Remaining decisions are left to the constraint solver's discretion; in the experiments, we use *free search* which alternates between user-specified and activity-based search when search is restarted (set to 100).

To improve solving, we make sure to arrange the matches in order of increasing cost. If there is a tie between two matches, and either of them is a kill match, then the kill match comes first. Otherwise, the match covering more operations comes first (hence mimicking the scheme of maximum munch). This is because the constraint solver will most likely attempt to select matches in the order given to the model. In such a setting, it is generally a good approach to first try a kill match – which incurs no cost and encourages value reuse – and then try the match incurring the least cost.

6.7 Presolving

As explained in Chap. 3, presolving is the process of applying problem-specific algorithms to reduce the number of variables or to shrink the variable domains before solving.[3] In this section, presolving is used to remove matches which can be safely removed without compromising code quality. This directly translates to fewer **alt**, **sel**, and **uplace** variables, smaller **dplace** and **oplace** domains, as well as fewer constraints that need to be managed by the constraint solver.

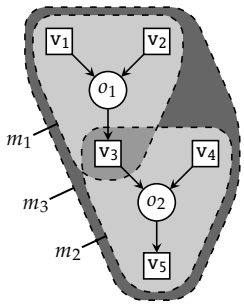
6.7.1 Dominated Matches

If two matches are equal in all respects except cost, then the match with greater cost is dominated and can thus safely be removed from the match set. A match m_1 is *dominated* if there exists another match m_2 such that

- m_1 has greater than or equal cost to m_2,
- both cover the same operations,
- both have the same entry blocks (if any),

[2] In the field of decision theory, this is known as the *minimax approach* [327].

[3] In this sense, the bound tightening technique described in Sect. 6.5 is a form of presolving.



match	v_1	v_2	v_3	v_4	v_5
m_1	$l_1 \dots l_4$	$l_1 \dots l_4$	$l_1 \dots l_4$		
m_2			$l_3 \dots l_4$	$l_3 \dots l_4$	$l_3 \dots l_6$
m_3	$l_1 \dots l_6$	$l_1 \dots l_6$	l_{INT}	$l_1 \dots l_6$	$l_1 \dots l_6$

Figure 6.4: Example where matches m_1 and m_2 are jointly dominated by match m_3 and can therefore safely be removed (provided that no other match uses value v_3). The table contains the location restrictions enforced by each match.

- both span the same blocks (if any),
- both have the same definition edges (if any),
- m_1 has at least as strong location requirements on its data as m_2 – that is, $\forall p_1 \in \mathit{uses}(m_1) \cup \mathit{defines}(m_1) : \exists p_2 \in \mathit{uses}(m_2) \cup \mathit{defines}(m_2) : D_{p_1} \subseteq D_{p_2} \wedge \mathit{stores}(m_1, p_1) \subseteq \mathit{stores}(m_2, p_2)$ – and
- both apply the same additional constraints (if any) when selected.

As a precaution, we assume that null matches, φ-matches, and matches with fall-through conditions can never be dominated.

The method above can be generalized to letting combinations of matches to be jointly dominated by another match. Intuitively, if the combination of matches can be selected, then the solution can always be improved by replacing them with the single match. The idea is to combine the matches into a single match m, and then check whether the above conditions for dominance apply with the additional check that none of intermediate values of m are used by other matches. An example is shown in Fig. 6.4.

6.7.2 Illegal Matches

Depending on the instruction set, the match set may contain matches that will – for one reason or another – never participate in any solution. Such matches are said to be *illegal*.

One set of illegal matches are those which would leave some operation uncoverable if selected. In Fig. 6.5, for example, selecting match m_1 would leave operation o_2 uncovered since it can only be covered by match m_2. However, selection of m_2 is inhibited if m_1 is selected. Hence this set of illegal matches is computed as

$$\{m \mid m \in M, o_1, o_2 \in O \text{ s.t. } M_{o_1} \subset M_{o_2} \wedge m \in M_{o_2}\}. \tag{6.35}$$

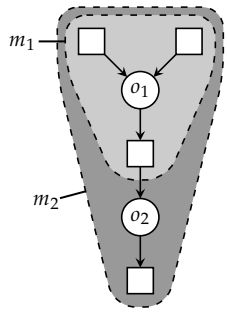


Figure 6.5: Example of an illegal match, where selecting match m_1 would leave operation o_2 uncovered.

Likewise, a match is illegal if selecting it would leave some datum undefined. With similar reasoning, this set of illegal matches is computed as

$$\{m \mid m \in M, d_1, d_2 \in D \text{ s.t. } M_{d_1} \subset M_{d_2} \wedge m \in M_{d_2}\}. \tag{6.36}$$

If a kill match m defines a datum d and every match using d has no alternatives but d, then m is illegal as d must be defined by a non-kill match. This set of illegal matches is computed as

$$\left\{ m_1 \;\middle|\; \begin{array}{l} m_1 \in M_\times, p_1 \in \mathit{defines}(m_1), d \in D_{p_1}, \\ m_2 \in M_{\bar{\times}}, p_2 \in \mathit{uses}(m_2) \text{ s.t. } d \in D_{p_2} \Rightarrow D_{p_2} = \{d\} \end{array} \right\}. \tag{6.37}$$

If a match m is not a kill match and defines a datum d in a location that cannot be accessed by any of the matches using of d, then m is illegal. This set of illegal matches is computed as

$$\left\{ m \;\middle|\; \begin{array}{l} m \in M_{\bar{\times}}, p \in \mathit{defines}(m), d \in D_p \text{ s.t.} \\ \mathit{isExt}(m, p) \wedge \mathit{cupUseLocsOf}(d) \neq \varnothing \wedge \\ \mathit{stores}(m, p) \cap \mathit{cupUseLocsOf}(d) = \varnothing \end{array} \right\}, \tag{6.38}$$

where

$$\mathit{cupUseLocsOf}(d) \equiv \bigcup_{\substack{m \in M_d \setminus M_\times, \\ p \in \mathit{uses}(m) \text{ s.t. } d \in D_p}} \mathit{stores}(m, p). \tag{6.39}$$

Note that if $\mathit{cupUseLocsOf}(d) = \varnothing$ holds, then the matches using datum d have themselves conflicting location requirements. In such cases, we cannot infer whether a match defining d is illegal.

Similarly, if a match m is not a kill match and uses a datum d from a location that cannot be written to by any of the matches defining d, then m can never be

selected and is thus illegal. This set of illegal matches is computed as

$$\left\{ m \;\middle|\; \begin{array}{l} m \in M_{\overline{\times}}, p \in uses(m) \setminus defines(m), d \in D_p \text{ s.t.} \\ cupDefLocsOf(d) \neq \varnothing \wedge stores(m,p) \cap cupDefLocsOf(d) = \varnothing \end{array} \right\}, \tag{6.40}$$

where

$$cupDefLocsOf(d) \equiv \bigcup_{\substack{m \in M_d \setminus M_\times, \\ p \in defines(m) \text{ s.t. } d \in D_p}} stores(m,p). \tag{6.41}$$

6.7.3 Redundant Matches

In certain circumstances a match can safely be removed without compromising code quality. Such matches are said to be *redundant*.

For example, if there exists a null-copy match to cover a copy node c, then the kill match covering c is redundant since it is always safe to select the null-copy match over the kill match. Consequently, all kill matches covering copy nodes that take a non-constant value as input – which can never be covered using a null-copy match – are redundant. This makes sense as the kill matches are added to the match set as a consequence of alternative values, which are used to handle cases where loaded constants could be reused among matches. Hence this set of redundant matches is computed as

$$\{m \mid m \in M_\times, o \in covers(m) \text{ s.t. } M_o \cap M_{\text{Φ}} \neq \varnothing\}. \tag{6.42}$$

Another case concerns non-null matches that cover a copy node. Assume a copy chain $v_1 \to c \to v_2$, where c is a copy node and v_1 and v_2 are value nodes. If every match defining v_1 writes the value to a location that can be used by all matches using v_2, then all non-null matches covering c are redundant since a null-copy match can always be selected to cover c. We exclude, however, copy nodes that take a constant value as input since such nodes can never be covered by a null-copy match. We also exclude copy nodes whose defined datum is used by some φ-match since an actual copy may be needed to satisfy Eq. 5.11. Let $M_\circ \subseteq M$ denote the set of copy matches and $D_\blacksquare \subseteq D$ denote the set of data representing constant values. Let also $capUseLocsOf(d) \subseteq L$ and $capDefLocsOf(d) \subseteq L$ denote the intersection of all locations for all match where a datum d is used respectively defined. Using these definitions, this set of redundant matches is computed as

$$\left\{ m \;\middle|\; \begin{array}{l} m \in M_\circ \setminus M_\perp, d_1 \in uses(m), d_2 \in defines(m) \\ \text{s.t. } D_{d_1} \cap M_\varphi = \varnothing \wedge D_{d_2} \cap M_\varphi = \varnothing \wedge d_1 \notin D_\blacksquare \\ \wedge\ capUseLocsOf(d_1) \cap capDefLocsOf(d_2) \neq \varnothing \end{array} \right\}. \tag{6.43}$$

As can be deduced from their name, *capUseLocsOf* and *capDefLocsOf* are defined similarly to *cupUseLocsOf* and *cupDefLocsOf* (see Eqs. 6.39 and 6.41), with the exception that locations known to violate Eq. 5.11 are removed from these sets.

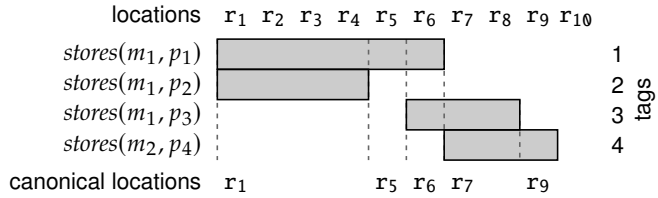


Figure 6.6: Example of canonical locations for a location set with ten registers.

6.7.4 Canonical Locations

For most architectures, its instructions read from and write to the same set of registers. This gives rise to many symmetric solutions as the exact location assigned to a value often does not matter. We can remove these symmetries by removing locations which are considered symmetric to one another.

The idea is to select a location as a representative for each distinct intersection made by the storage requirements. See for example Fig. 6.6. For sake of discussion, each storage requirement has been labeled with a *tag*. Given the location set and storage requirements shown in the figure, they give rise to five intersections with respect to the locations: $\{\mathtt{r}_1, \ldots, \mathtt{r}_4\}$ due to tags 1 and 2, $\{\mathtt{r}_5\}$ due to tag 1, $\{\mathtt{r}_6\}$ due to tag 1 and 3, $\{\mathtt{r}_7, \mathtt{r}_8\}$ due to tags 3 and 4, and $\{\mathtt{r}_9\}$ due to tag 4. From each of these intersections we select a representative, and the union of these representative locations constitute the set of *canonical locations*. An algorithm for computing this set based on the intuition above is shown in Alg. 6.1.

Once computed, we substitute all locations appearing in the storage requirements with their canonical representative and then replace the original location set with the canonical set, thus shrinking the domains of the **loc** variables.

6.8 Experimental Evaluation

Excluding the refinement described in Sect. 6.1, whose naive equivalence cannot be implemented on our experimental setup and therefore not be evaluated, we now evaluate the impact of the solving techniques introduced in this chapter.

When filtering, we remove all functions that have fewer than 50 LLVM IR instructions and more than 150 instructions. Anything smaller will most likely not show the impact of the given solving technique, and anything larger will lead to unreasonably long experiment runtimes. This leaves a pool of 284 functions, from which we then draw 20 samples.

To curb experiment runtimes, we apply a time limit of 600 s to the constraint solver. For any given function, the last solution found is considered optimal if and only if the solver has finished its execution within the time limit. When using an upper cost bound, we take the cost for the solution computed by LLVM 3.8 for the given function.

```
function CanonicalizeLocs (location set L, match set M):
    T ← vector with |L| elements initialized to ∅
    t ← 1
    for m ∈ M do                                              // assign tags
        for p ∈ uses(m) ∪ defines(m) do
            for l ∈ stores(m, p) do
                T[l] ← T[l] ∪ {t}
            t ← t + 1
    G ← {T[l] | l ∈ L}                                        // find all groups of tags
    L_C ← ∅
    for g ∈ G do
        l_C ← min({l | l ∈ L s.t. T[l] = g})                  // find representative for this group of tags
        L_C ← L_C ∪ {l_C}
    return L_C
```

Algorithm 6.1: Computes the canonical locations from a given location set. If location restrictions for some data are already enforced by the function, then these are also tagged and processed accordingly.

6.8.1 Objective Function Refinements and Cost Bounding

We first evaluate the different methods for computing the cost matrix. Based on the results, we then use the superior method to evaluate the impact that different implementations of the objective function, coupled with cost bounding, have on solving time and code quality.

Multiply-then-Divide vs. Divide-then-Multiply We evaluate the different methods for computing the cost matrix by comparing the solving times exhibited by two versions of the constraint model: (I) one based on the multiply-then-divide method (Eqs. 6.6 and 6.7); and (II) one based on divide-then-multiply methods (Eqs. 6.8 and 6.9). No hypothesis is attempted on which model is better.

Figure 6.7 shows the normalized solving times (including presolving time) for the two constraint models described above in the first experiment, with model I as baseline and model II as subject. The solving times range from 0.633 s to 636 s[4] with a CV of 0.01. The GMI is 3.34× with CI [1.82, 5.12]×.

We see clearly that model II results in considerably shorter solving times than model I, thus underscoring the fact that seemingly trivial changes to a constraint model may have considerable impact on solving time. For one function (`build_ycc_rgb_t`) the solving time is improved by 84.2⧖, and for no function is solving time degraded. Hence the divide-then-multiply method is a better design

[4]The solving time given here greater than the time limit because it also includes the presolving time while the time limit is only applied on the constraint solver.

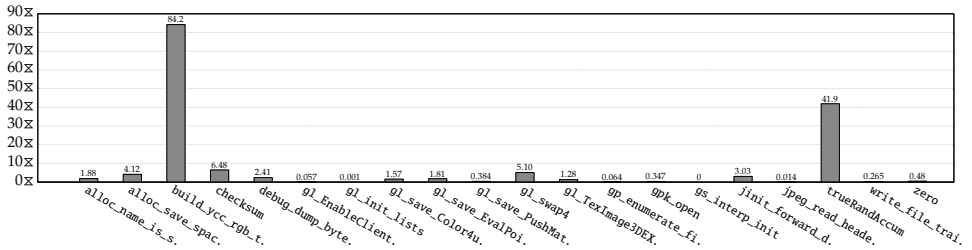


Figure 6.7: Normalized solving times (incl. presolving time) for two constraint models using different operation cost functions: one based on the multiply-then-divide method (baseline), and one based on the divide-then-multiply method (subject). GMI: 3.34×, CI: [1.82, 5.12]×.

choice over multiply-then-divide method when implementing the refined objective function.

One possible explanation is that they yield different bounds. For the example given in Fig. 6.2 on p. 95, the multiply-then-divide method bounds the total cost as $4 \leq \mathbf{cost} \leq 65$, whereas the divide-then-multiply method bounds it to $4 \leq \mathbf{cost} \leq 60$. However, this does not always hold as the divide-then-multiply method may yield a tighter bound than the multiply-then-divide method for other problem instances. Another possible explanation is that the divide-then-multiply method results in operation costs that are even multiples of the execution frequencies. In comparison, the multiply-then-divide method could potentially result in arbitrary cost values that in turn would lead to larger domains for the cost variable.

Impact on Solving Time We evaluate the impact that the refined objective function together with cost bounding have on solving by comparing the number of functions that can be solved optimally by six versions of the constraint model: (I) one based on the naive implementation of the objective function (Eq. 5.23), without cost bounds; (II) another naive model but with lower and upper bound; (III) one based on the refined implementation (Eqs. 6.10 and 6.11) using the divide-then-multiply method, without bounds; (IV) another refined model but with lower bound; (V) another refined model but with upper bound; and (VI) another refined model but with both bounds.

Since the refined objective function enables tighter bounds to be derived for the cost variable, we expect models III, IV, V, and VI to find a greater number of optimal solutions compared with models I and II. Due to further tightening of the cost bounds, we expect model II to outperform model I, models IV, V, and VI to outperform model III, and model VI to outperform models IV and V.

Figure 6.8 shows the percentage of optimal solutions found over time for the five constraint models described above in the second experiment. The solving times range from 0.319 s to 608 s with a CV of 0.04.

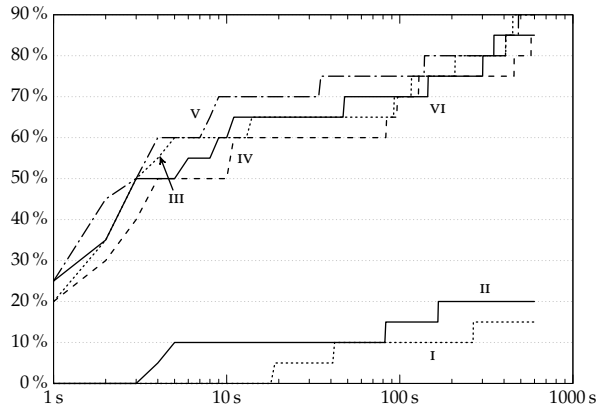


Figure 6.8: Percentage of optimal solutions found over time for six constraint models: (I) one based on the naive implementation of the objective function without cost bounds; (II) another naive model but with lower and upper bound; (III) one based on the refined implementation without bounds; (IV) another refined model but with lower bound; (V) another refined model but with upper bound; and (VI) another refined model but with both bounds.

We see that models III, IV, V, and VI clearly outperform models I and II. We see also that applying an upper cost bound has a positive effect for both objective functions, although the benefit is greater for the refined objective function. This gain is due to the fact that, for some functions, the solution computed by LLVM is already optimal with respect to the model. Consequently, the solver need only prove that there exists no better solution instead of exploring the entire search space. Hence, in terms of solving time the refined objective function coupled with an upper cost bound is a better design choice over naive objective function. Moreover, this decision is crucial for scalability.

Applying a lower cost bound, however, does not appear to be equally beneficial. In fact, using a lower bound causes models IV and VI to fail to find the optimal solution for two functions whereas models III and V manages to find the optimal solution for all functions but one. A possible explanation is that the lower bound computed by the relaxed constraint model is too weak to be lead to any propagation and instead only interferes with CHUFFED's LCG engine (see Chap. 3 on p. 56 for a description of LCG).

Impact on Code Quality We evaluate the impact on code quality by comparing the cost of the best solution found within the time limit by models II (naive model with bounds) and VI (refined model with bounds). For the same reason as in the impact-on-solving-time experiment, we expect the solutions produced by model VI to be of significantly better quality compared with those produced by model II.

Figure 6.9 shows the normalized solution costs for the two constraint models

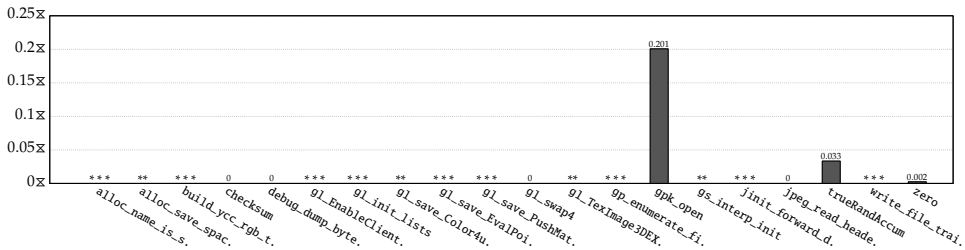


Figure 6.9: Normalized best solution costs for two constraint models: one implementing the naive objective function (baseline), and one implementing the refined objective function (subject). GMI: 1.03×, CI: [1.0003, 1.09]×. Both models use lower and upper cost bounds. Functions marked with ** are those for which the naive objective function fails to produce any solution, and functions marked with *** are those where the solution produced by LLVM is already optimal w.r.t. the model.

described above in the third experiment, with model II as baseline and model V as subject. The costs range from 1162 cycles to 28 668 cycles with a CV of 0.00. The GMI is 1.03× with CI [1.0003, 1.09]×.

We see clearly that the models based on the refined objective function yield better code quality than those based on the naive objective function. For one function (`gpk_open`), the code quality is improved by 0.201⧖, and for no function is code quality degraded. Hence, in terms of code quality the refined objective function coupled with an upper cost bound is a better design choice over naive objective function.

Conclusions From the results for these experiments, we conclude: (i) that the divide-then-multiply method is superior to the multiply-then-divide method; (ii) that the refined implementation of the objective function is superior to the naive implementation; and (iii) that using the refined objective function together with an upper cost bound is crucial for scalability.

6.8.2 Implied Constraints

We first evaluate the implied constraints individually. Based on the results, we then arrange them into groups to find the best combination of implied constraints. To curb experiment runtimes, we include all other solving techniques in the models because, without them, a very long time limit would have to be applied to discover the impact on solving time.

Impact of a Single Constraint We evaluate the impact of each implied constraint (Eqs. 6.12–6.26) by comparing the solving times exhibited by two versions of the constraint model: (I) one without a particular implied constraint; and (II) one with all constraints.

Because the pattern set used in these experiments contains only patterns that do not span any blocks, we expect Eqs. 6.14 and 6.21 to have no impact on solving time. For the rest, we expect some constraints to lead to an overall solving time improvement while others may degrade solving time. As described in Chap. 3, this is because some constraints may be too expensive to execute compared with the amount of propagation they provide.

Figure 6.10 shows the normalized solving times (including presolving time) for the two constraint models described above in the first experiment, with model I as baseline and model II as subject. The solving times range from 0.625 s to 636 s with a CV of 0.06. The GMIs and CIs are given in Fig. 6.10.

We see that Eqs. 6.12 and 6.16 lead to an overall solving time improvement, and that Eqs. 6.14, 6.17, and 6.21 have no impact on solving time. For Eqs. 6.15, 6.18, 6.19, 6.20, 6.22, 6.23, 6.24, 6.25, and 6.26, the results are inconclusive.

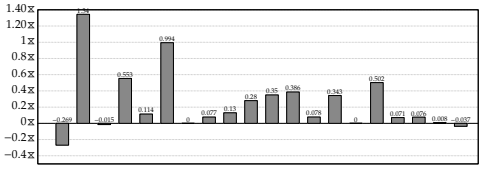
Impact of Groups of Constraints We find the best combination of implied constraints by arranging them into groups and evaluating the impact of each such group. Because a full evaluation would require us to test every combination of constraints – which would result in an unreasonably large number of experiments – we limit ourselves to only comparing the solving times exhibited by three versions of the model: (I) one with no implied constraints; (II) one with only those who individually lead to an overall solving time improvement; and (III) one with all constraints. We expect models II and III to all perform better than model I. No hypothesis is attempted regarding the relative performance between models II and III.

Figure 6.11 shows the normalized solving times (including presolving time) for three of the constraint models described above, with model I as baseline and models II and III as subjects. The solving times range from 0.725 s to 635 s with a CV of 0.02.

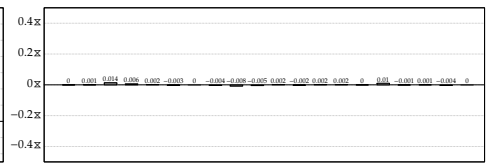
To begin with, we observe that the GMI for model II over model I is 1.43× with CI [1.06, 1.72]× (Fig. 6.11a). Hence combining Eqs. 6.12 and 6.16 yields a greater overall solving time improvement than when picked individually (up to 5.37⧖). However, we note that this combination yields a non-negligible solving time degradation for one function (down to −2.43⧖).

In comparison, we observe that the GMI for model III over model I is 1.53× with CI [1.23, 1.84]× (Fig. 6.11b), which is a greater improvement compared to model II. Although the maximum improvement is less than for model II (up to 4.82⧖), this combination yields no considerable degradation for any function (down to at most −0.111⧖). Hence it is most beneficial to include all implied constraints in the model.

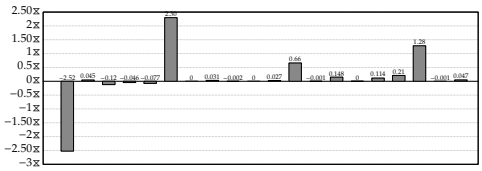
Conclusions From the results for these experiments, we conclude: (i) that some of the implied constraints have individually a positive impact on solving time; but (ii) it is most beneficial to include all implied constraints in the model.



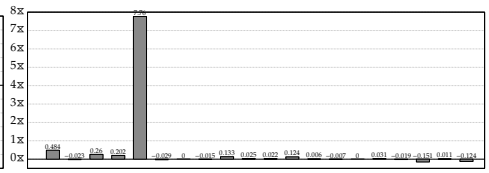
(a) Eq. 6.12. GMI: 1.23×, CI: [1.09, 1.36]×.



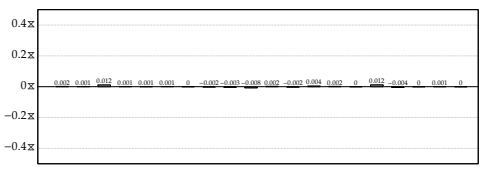
(b) Eq. 6.14. GMI: 1.00×, CI: [1.00, 1.00]×.



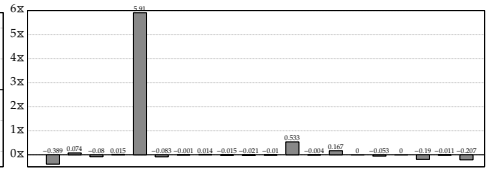
(c) Eq. 6.15. GMI: 1.13×, CI: [0.89, 1.31]×.



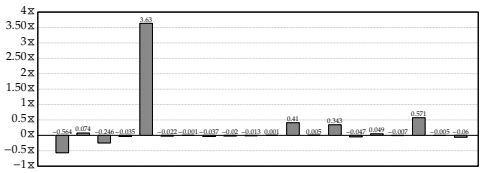
(d) Eq. 6.16. GMI: 1.21×, CI: [1.004, 1.47]×.



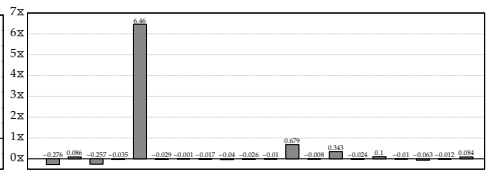
(e) Eq. 6.17. GMI: 1.00×, CI: [1.00, 1.00]×.



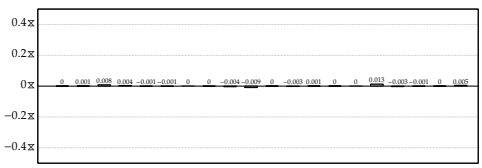
(f) Eq. 6.18. GMI: 1.13×, CI: [0.94, 1.36]×.



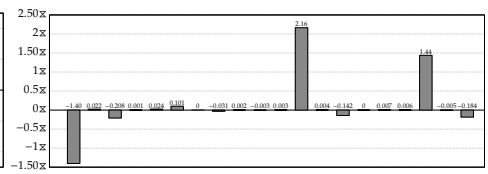
(g) Eq. 6.19. GMI: 1.13×, CI: [0.96, 1.32]×.



(h) Eq. 6.20. GMI: 1.17×, CI: [0.97, 1.42]×.

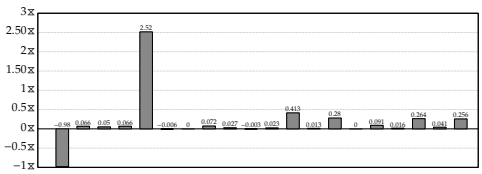


(i) Eq. 6.21. GMI: 1.00×, CI: [1.00, 1.00]×.

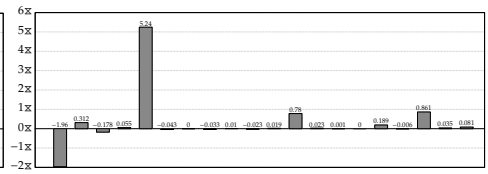


(j) Eq. 6.22. GMI: 1.08×, CI: [0.89, 1.24]×.

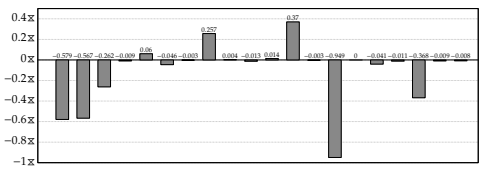
Figure 6.10: Normalized solving times (incl. presolving time) for two constraint models: one without a particular implied constraint (baseline), and one with all constraints (subject).



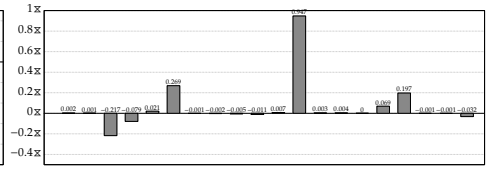
(k) Eq. 6.23. GMI: 1.14×, CI: [0.97, 1.30]×.



(l) Eq. 6.24. GMI: 1.19×, CI: [0.91, 1.43]×.

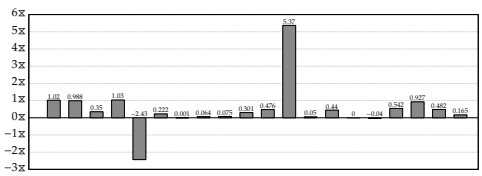


(m) Eq. 6.25. GMI: 0.93×, CI: [0.83, 1.01]×.

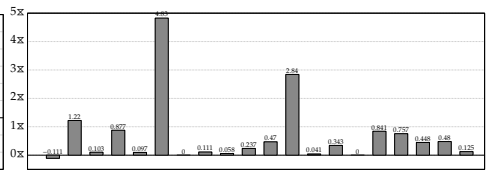


(n) Eq. 6.26. GMI: 1.05×, CI: [0.98, 1.13]×.

Normalized solving times (incl. presolving time) for two constraint models: one without a particular implied constraint (baseline), and one with all constraints (subject).



(a) Model I vs. II. GMI: 1.43×, CI: [1.06, 1.72]×.



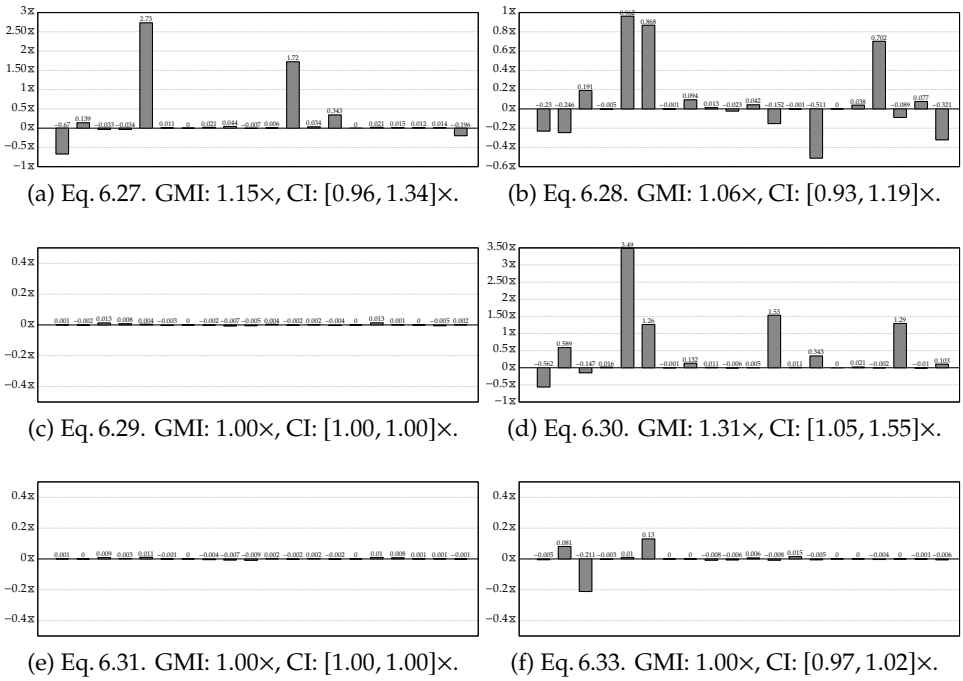
(b) Model I vs. III. GMI: 1.53×, CI: [1.23, 1.84]×.

Figure 6.11: Normalized solving times (incl. presolving time) for three constraint models: (I) one with no implied constraints (baseline); (II) one with only those who individually lead to an overall solving time improvement (Eqs. 6.12 and 6.16; subject); and (III) one with all constraints (subject).

6.8.3 Symmetry and Dominance Breaking Constraints

Like in Sect. 6.8.2, we first evaluate the symmetry and dominance breaking constraints individually. Based on the results, we then arrange them into groups to find the best combination of symmetry and dominance breaking constraints. To curb experiment runtimes, we include all other solving techniques in the models because, without them, a very long time limit would have to be applied to discover the impact on solving time.

Impact of a Single Constraint We evaluate the impact of each symmetry and dominance breaking constraint (Eqs. 6.27–6.33) by comparing the solving times exhibited by two versions of the constraint model: (I) one without a particular



(a) Eq. 6.27. GMI: 1.15×, CI: [0.96, 1.34]×.

(b) Eq. 6.28. GMI: 1.06×, CI: [0.93, 1.19]×.

(c) Eq. 6.29. GMI: 1.00×, CI: [1.00, 1.00]×.

(d) Eq. 6.30. GMI: 1.31×, CI: [1.05, 1.55]×.

(e) Eq. 6.31. GMI: 1.00×, CI: [1.00, 1.00]×.

(f) Eq. 6.33. GMI: 1.00×, CI: [0.97, 1.02]×.

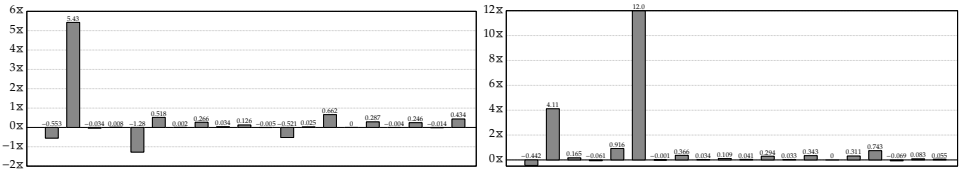
Figure 6.12: Normalized solving times (incl. presolving time) for two constraint models: one without a particular symmetry or dominance breaking constraint (baseline), and one with all constraints (subject).

symmetry or dominance breaking constraint; and (II) one with all constraints. For the same reason as with the implied constraints, we expect some constraints to lead to an overall solving time improvement while others may degrade solving time.

Figure 6.12 shows the normalized solving times (including presolving time) for two constraint models described above in the first experiment, with model I as baseline and model II as subject. The solving times range from 0.634 s to 635 s with a CV of 0.03. The GMIs and CIs are given in Fig. 6.12.

We observe that Eq. 6.30 leads to an overall solving time improvement and that Eq. 6.29 has no impact on solving time. For Eqs. 6.27, 6.28, 6.31, and 6.33, the results are inconclusive.

Impact of Groups of Constraints We find the best combination of symmetry and dominance breaking constraints by arranging them into groups and evaluating the impact of each such group. Again, for practicality we limit ourselves to only comparing the solving times exhibited by three versions of the model: (I) one with no symmetry or dominance breaking constraints; (II) one with only those who individually lead to an overall solving time improvement; and (III) one with all



(a) Model I vs. II. GMI: 1.18×, CI: [0.92, 1.42]×. (b) Model I vs. III. GMI: 1.49×, CI: [1.09, 1.90]×.

Figure 6.13: Normalized solving times (incl. presolving time) for three constraint models: (I) one with no symmetry or dominance breaking constraints (baseline); (II) one with only those who individually lead to an overall solving time improvement (Eq. 6.30; subject); and (III) one with all constraints (subject).

constraints. We expect models II and III to all perform better than model I. No hypothesis is attempted regarding the relative performance between models II and III.

Figure 6.13 shows the normalized solving times (including presolving time) for the three constraint models described, with model I as baseline and models II and III as subjects. The solving times range from 1.02 s to 635 s with a CV of 0.04.

To begin with, we observe that the GMI for model II over model I is 1.18× with CI [0.92, 1.42]× (Fig. 6.13a). This means that combining only the symmetry or dominance breaking constraints that individually have a positive impact on solving time is not sufficient. Although solving time is improved considerably for one function (up to 5.43⧖), this combination also degrades solving time for several functions (down to −1.28⧖).

In comparison, we observe that the GMI for model III over model I is 1.49× with CI [1.09, 1.90]× (Fig. 6.13b), which is a statistically significant, positive impact. In addition, the maximum improvement is greater than for model II (up to 12.0⧖), and this combination yields no considerable degradation for any function (down to at most −0.442⧖). Hence it is most beneficial to include all such constraints in the model.

Conclusions From the results for these experiments, we conclude: (i) that some of the symmetry and dominance breaking constraints have individually a positive impact on solving time; but (ii) it is most beneficial to include all symmetry and dominance breaking constraints in the model.

6.8.4 Presolving

Like in Sects. 6.8.2 and 6.8.3, we first evaluate the presolving techniques individually. Based on the results, we then arrange them into groups to find the best combination of presolving techniques. To curb experiment runtimes, we include all other solving techniques in the models because, without them, a very long time limit would have to be applied to discover the impact on solving time.

Impact of a Single Technique We evaluate the impact of each presolving technique (Sect. 6.7.1, Eqs. 6.35–6.43, and Sect. 6.7.4) by comparing the solving times exhibited by two versions of the constraint model: (I) one without a particular presolving technique; and (II) one with all techniques. For the same reason as with the implied, symmetry breaking, dominance breaking constraints, we expect some techniques to lead to an overall solving time improvement while others may degrade solving time.

Figure 6.14 shows the normalized solving times (including presolving time) for the two constraint models described above in the first experiment, with model I as baseline and model II as subject. The solving times range from 0.599 s to 635 s with a CV of 0.07. The GMIs and CIs are given in Fig. 6.14.

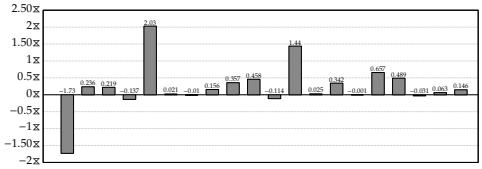
We observe that Eqs. 6.42 and 6.43, and canonical locations lead to an overall solving time improvement, that Eqs. 6.38 and 6.40 lead to an overall solving time degradation, and that Eq. 6.35 has no impact on solving time. For dominated matches and Eqs. 6.36 and 6.37, the results are inconclusive.

Equations 6.38 and 6.40 degrade solving time in this experiment because they are expensive to compute and only identify redundant matches that appear for target machines with irregular instruction sets (meaning the instructions access different sets of registers). Since Hexagon has a regular instruction set, these presolving techniques identify too few redundant matches to offset the cost in finding them.

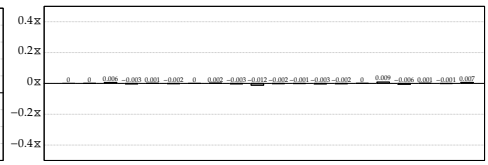
Impact of Groups of Techniques We find the best combination of presolving techniques by arranging them into groups and evaluating the impact of each such group. Again, for practicality we limit ourselves to only comparing the solving times exhibited by four versions of the model: (I) one with no presolving techniques implied constraints; (II) one with only those who individually lead to an overall solving time improvement; (III) one without those who individually lead to an overall solving time degradation; and (IV) one with all techniques. We expect models II, III, and IV to all perform better than model I. This is because many of these presolving techniques require computations that are expensive to execute but whose result can be shared among the techniques, allowing this cost to be amortized when the techniques are executed in unison. No hypothesis is attempted regarding the relative performance between models II, III, and IV.

Figure 6.15 shows the normalized solving times (including presolving time) for the four constraint models described above in the second experiment, with model I as baseline and models II, III, and IV as subjects. The solving times range from 0.581 s to 628 s with a CV of 0.67.

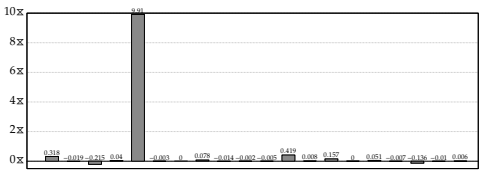
To begin with, we observe that the GMI for model II over model I is 1.77× with CI $[1.22, 2.38]\times$ (Fig. 6.15a), which is a statistically significant, positive impact. But although the improvement is considerable for some functions (up to 28.2⧖), we note that this combination yields a considerable degradation for others (down to -0.647⧖).



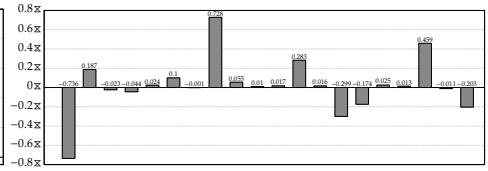
(a) Dominated matches. GMI: 1.23×, CI: [0.99, 1.41]×.



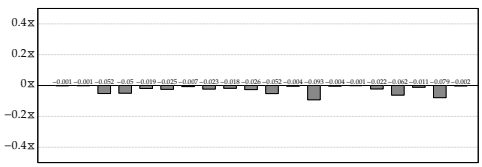
(b) Eq. 6.35. GMI: 1.00×, CI: [1.00, 1.00]×.



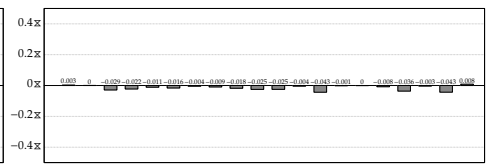
(c) Eq. 6.36. GMI: 1.22×, CI: [0.99, 1.51]×.



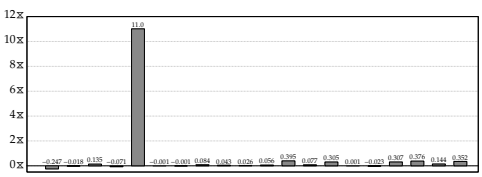
(d) Eq. 6.37. GMI: 1.03×, CI: [0.92, 1.12]×.



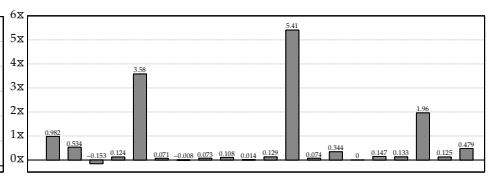
(e) Eq. 6.38. GMI: 0.97×, CI: [0.96, 0.98]×.



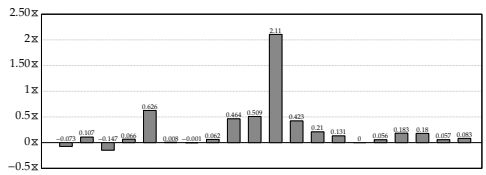
(f) Eq. 6.40. GMI: 0.99×, CI: [0.98, 0.99]×.



(g) Eq. 6.42. GMI: 1.30×, CI: [1.04, 1.61]×.

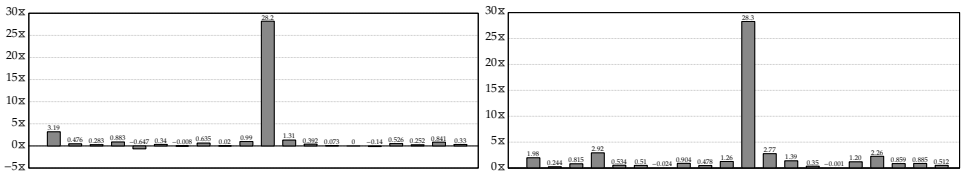


(h) Eq. 6.43. GMI: 1.50×, CI: [1.16, 1.83]×.

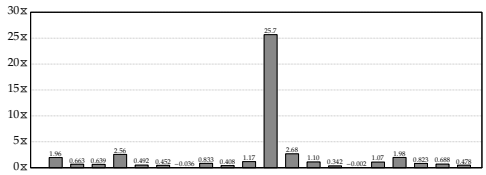


(i) Canonical locations. GMI: 1.22×, CI: [1.08, 1.37]×.

Figure 6.14: Normalized solving times (incl. presolving time) for two constraint models: one without a particular presolving technique (baseline), and one with all techniques (subject).



(a) Model I vs. II. GMI: 1.77×, CI: [1.22, 2.38]×. (b) Model I vs. III. GMI: 2.30×, CI: [1.66, 3.07]×.



(c) Model I vs. IV. GMI: 2.22×, CI: [1.62, 2.93]×.

Figure 6.15: Normalized solving times (incl. presolving time) for four constraint models: (I) one with no presolving techniques (baseline); (II) one with only those who individually lead to an overall solving time improvement (Eqs. 6.42 and 6.43, and canonical locations; subject); (III) one without those who individually lead to an overall solving time degradation (Eqs. 6.38 and 6.40; subject); and (IV) one with all techniques (subject).

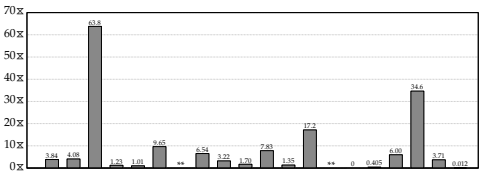
In comparison, we observe that the GMI for model III over model I is 2.30× with CI [1.66, 3.07]× (Fig. 6.15b), which is a greater solving time improvement compared to model II. In addition, the maximum improvement is greater than for model II (up to 28.3⧖), and this combination yields no considerable degradation for any function (down to at most -0.0239⧖).

Lastly, we observe that the GMI for model IV over model I is 2.22× with CI [1.62, 2.93]× (Fig. 6.15c), which is still better than model II but worse than model III. In addition, the maximum improvement is less than for model III (up to at most 25.7⧖). Hence, in this experiment it is most beneficial to exclude Eqs. 6.38 and 6.40 from the model. However, for target machines with irregular instruction sets it may be beneficial to keep them in the model.

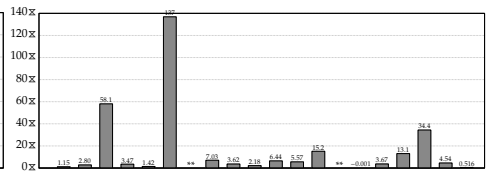
Conclusions From the results for these experiments, we conclude: (i) that some of the presolving techniques have a positive impact on solving time while others have a negative impact; and (ii) that it is most beneficial to exclude the techniques that have a negative impact from the model.

6.8.5 Impact of All Solving Techniques

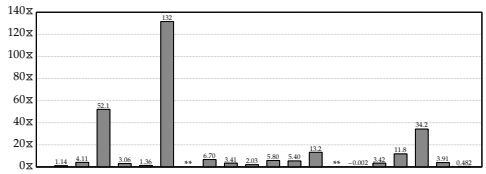
We now evaluate the solving-time impact made by different collections of solving techniques by comparing the solving times exhibited by four versions of the



(a) Model I vs. II. GMI: 4.79×, CI: [2.61, 7.24]×.



(b) Model I vs. III. GMI: 6.43×, CI: [3.36, 10.4]×.



(c) Model I vs. IV. GMI: 6.21×, CI: [3.29, 10.0]×.

Figure 6.16: Normalized solving times (incl. presolving time) for four constraint models: (I) one with no solving techniques (baseline); (II) one with only those who individually lead to an overall solving time improvement (subject) (Eqs. 6.12, 6.16, 6.30, 6.42, and 6.43, and canonical locations; subject); (III) one without those who individually lead to an overall solving time degradation (Eqs. 6.38 and 6.40; subject); and (IV) one with all techniques (subject). Functions marked with ** are those for which model I fails to produce any solution.

constraint model: (I) one with no solving techniques; (II) one with only those who individually lead to an overall solving time improvement; (III) one with only those who individually lead to an overall solving time degradation; and (IV) one with all techniques. We expect models II, III, and IV to all perform better than model I. No hypothesis is attempted regarding the relative performance between models II, III, and IV.

Figure 6.16 shows the normalized solving times (including presolving time) for the four constraint models described above, with model I as baseline and models II, III, and IV as subjects. The solving times range from 0.578 s to 628 s with a CV of 0.05. The GMIs and CIs are given in Fig. 6.16.

To begin with, we see clearly that all models significantly improve solving time over model I. For several functions, the improvement is considerable (up to 137⧖), and no combination yields considerable degradation for any function (down to at most 1×). In fact, for two functions model I is not even able to produce a solution within the time limit. Hence the solving techniques are crucial for scalability.

Next, we observe that model III yields the largest GMI over model I (6.43× with CI [3.36, 10.4]×), closely followed by model IV (6.21× with CI [3.29, 10.0]×), which in turn considerably outperforms model II (4.79× with CI [2.61, 7.24]×). We also note that the maximum improvement for model III is greater than for model IV (up to 137⧖ vs. up to 132⧖). Hence, only picking solving techniques

that have a statistically significant, positive effect on solving time when evaluated individually is too conservative. Also, rejecting the presolving techniques that have an overall negative effect on solving time when evaluated individually yields, in this experiment, the constraint model with best performance. Keep in mind, however, that for target machines with irregular instruction sets it may be beneficial to keep them in the model.

Conclusions From the results for these experiments, we conclude: (i) that the solving techniques introduced in this chapter are crucial for scalability; (ii) that picking only solving techniques that have a statistically significant, positive effect on solving time when evaluated individually is too conservative; and (iii) that rejecting Eqs. 6.38 and 6.40 gives the constraint model with best performance for target machines with regular instruction sets.

6.9 Summary

In this chapter, we have introduced a wide range of techniques for improving solving of the constraint model introduced in the previous chapter. Through experimental evaluation, the techniques were demonstrated to be crucial for scalability; for one function, the solving time was improved by 132⧖. Rejecting presolving techniques Eqs. 6.38 and 6.40 increased the improvement further to 137⧖ since the target machine is fairly regular (meaning its instructions can access the same set of registers). If the target machine has an irregular instruction set, however, then it may be worthwhile to keep these presolving techniques.

CHAPTER

7

Experimental Evaluation Using the State of the Art

This chapter evaluates how universal instruction selection compares against a state-of-the-art compiler. This is done in Sect. 7.1. Since the approach is able to leverage selection of SIMD instructions, we also evaluate this impact in Sect. 7.2.

7.1 Unison vs. LLVM

We evaluate the impact of the approach by comparing the cost (that is, the total number of cycles, as described in Chap. 5 on p. 88) of solutions produced by the approach with the solutions produced by LLVM 3.8 – a state-of-the-art compiler.

Setup When filtering, we remove all functions that have fewer than 50 LLVM IR instructions and more than 200 instructions. Anything smaller will most likely not show any gain using the approach, and anything larger will lead to unreasonably long experiment runtimes. This leaves a pool of 284 functions up to medium size, from which we then draw 20 samples.

To curb experiment runtimes, we apply a time limit of 600 s to the constraint solver. For any given function, the last solution found is considered optimal if and only if the solver has finished its execution within the time limit. When using an upper cost bound, we take the cost for the solution computed by LLVM for the given function.

Results Figure 7.1 shows the normalized solution costs, with LLVM as baseline and universal instruction selection as subject. The size of the UF graphs range from 189 to 1524 nodes. The costs range from 1129 cycles to 26 670 cycles, with a maximum coefficient of variation of 0.00. The solving times range from 0.590 s to 636 s with a CV of 0.01. The GMI is 1.03× with CI $[1.01, 1.06]\times$.

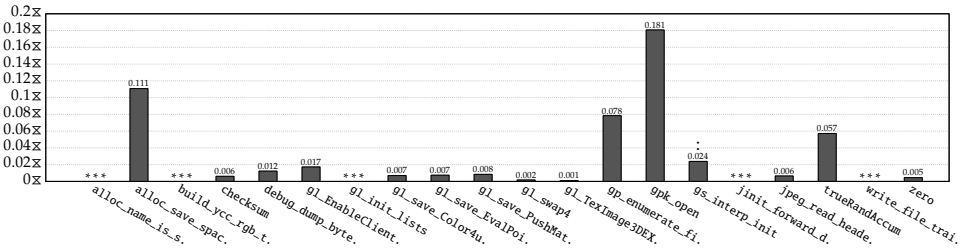


Figure 7.1: Solution costs produced by universal instruction selection, normalized to those produced by LLVM. GMI: 1.03×, CI: [1.01, 1.06]×. Functions whose bars are marked with two dots are those for which the subject does not find the optimal solution, and functions marked with * * * are those where the solution produced by LLVM is already optimal w.r.t. the model.

We see that universal instruction selection produces solutions with significantly less cost than those produced by LLVM (up to 0.181⧖ improvement). This is predominantly due to the combination of global instruction selection, global code motion, and block ordering. In three cases (`alloc_save_spac`, `gp_enumerate_fi`, and `gpk_open`), for example, the approach is able to reduce cost by lifting computations, in particular constant loads, out of blocks with high execution frequency into blocks with lower frequency. In another case (`checksum`), the approach is able to move an addition and memory operation into the same block and implement both using a single auto-increment memory instruction, whereas LLVM must implement these computations using two instructions. Such improvements are only possible when integrating global instruction selection with global code motion.

In three other cases (`jpeg_read_header`, `gl_TexImage3DEX`, and `gl_EnableClient`), the approach is able to reorder the blocks to remove one to two jump instructions. Such improvements are only possible when integrating global instruction selection with block ordering.

Conclusions From the results for these experiments, we conclude that universal instruction selection generates code of equal or better quality compared to the state of the art for up to medium-sized functions.

7.2 Impact of SIMD instructions

We evaluate the impact of selecting SIMD instructions by comparing the cost of solutions produced from two pattern sets derived from Hexagon: (I) one with no SIMD instructions; and (II) one with 2- and 4-way `add`, `sub`, `and`, and `or` instructions.

Setup When filtering, we again all functions that have fewer than 50 LLVM IR instructions and more than 150 instructions. Anything smaller will most likely

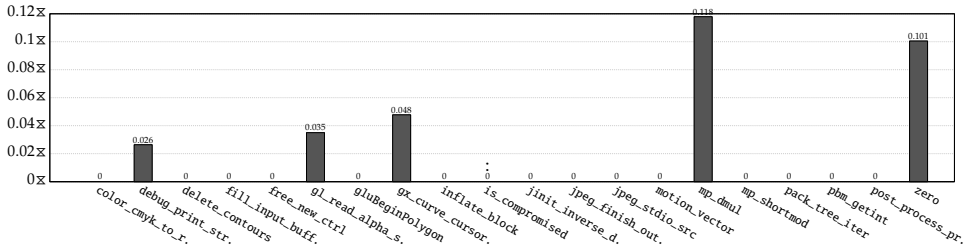


Figure 7.2: Normalized solution costs for two pattern sets: one without SIMD instructions (baseline), and one with such instruction (subject). GMI: 1.02×, CI: [1.003, 1.03]×. Functions whose bars are marked with two dots are those for which the subject does not find the optimal solution.

not have enough data parallelism for selection of SIMD instructions, and anything larger will lead to unreasonably long experiment runtimes. To increase the chance of data parallelism, we also remove all functions not containing at least two addition, subtraction, logical-and, or logical-or instructions. This leaves a pool of 221 functions, from which we then draw 20 samples.

In this experiment we do not apply an upper bound in this case as that may prevent interesting solution that make use of SIMD instruction. Note that no loop unrolling[1] is performed on any of the functions prior to instruction selection.

Results Figure 7.2 shows the normalized solution costs for the two pattern sets describe above, with pattern set I as baseline and pattern set II as subject. The costs range from 390 cycles to 18 963 cycles, with a CV of 0.00. The solving times from 0.642 s to 609 s, with a CV of 0.02. The GMI is 1.02× with CI [1.003, 1.03]×.

We see that the pattern set II yields solutions with significantly less cost than those yielded by pattern set I (up to 0.118⧖ improvement). The five cases with less cost (`debug_print_str`, `gl_read_alpha_s`, `gx_curve_cursor`, `mp_dmul`, and `zero`), universal instruction selection is able to combine pairs of additions or subtractions into 2-way SIMD instructions. In addition, in one of these cases (`gl_read_alpha_s`) the additions originally reside in different blocks, but due to global code motion the approach is able to move the computations to the same block and implement these using a single instruction.

Conclusions From the results for these experiments, we conclude that there is sufficient data parallelism to be exploited through selection of SIMD instructions without having to resort to loop unrolling. In addition, this exploitation benefits from global code motion as that allows computations to be gathered from different blocks and implemented using a single SIMD instruction.

[1] *Loop unrolling* is the task of duplicating the body of a loop in order to increase data parallelism at the cost of increasing code size.

CHAPTER

8

Proposed Model Extensions

In this chapter, we discuss how the constraint model can be extended to integrate other problems related to code generation. In Sect. 8.1 we propose an extension for integrating instruction scheduling. Based on this idea, we propose in Sect. 8.2 an extension for integrating register allocation.

8.1 Integrating Instruction Scheduling

As described in Chap. 1, apart from instruction selection the other two main problems of code generation are instruction scheduling and register allocation. More importantly, it is well known that these three problems interact with one another. For example, when discussing the experiments in Chap. 6 we observed that the constraint model pushes loading of constants into blocks with low execution frequencies to decrease cost. Such moves, however, may result in code where the definition and uses of a value are spread far apart, resulting in high register pressure. This in turn could force the register allocator to perform lots of spilling, which is potentially much more expensive than the cost saved by moving the loads. To take this cost into account, the constraint model introduced in Chap. 5 must be extended to integrate these problems.

In this context, the instruction scheduling problem can be defined as follows. Assume that we are given a set M' of selected matches, all placed in a block b. Let $lat(m)$ denote the instruction latency of a match $m \in M'$. Then instruction scheduling problem is to assign to each match $m \in M'$ an issue cycle c_m such that $c_m + lat(m) \leq c_{m'}$ holds for every match $m' \in M'$, $m \neq m'$, that uses data defined by m. To model this problem, we introduce two new variables.

Variables The set of variables $\mathbf{cycle}[m] \in \mathbb{N}$ models at which cycle a match m is scheduled, and the set of variables $\mathbf{sched}[m, b] \in \{0, 1\}$ models whether m is scheduled in block b.

Constraints To begin with, a match m must be scheduled in block b if and only if m is selected and placed in b. This can be modeled as

$$\forall b \in B, \forall m \in M : \mathbf{sched}[m, b] \Leftrightarrow \mathbf{sel}[m] \wedge \mathit{blockOf}(m) = b. \tag{8.1}$$

Next, if a selected match m_1 defines a datum d which is used by another selected match m_2, and m_1 and m_2 are both placed in the same block, then m_1 must be scheduled before m_2. This can be modeled as

$$\begin{aligned} \forall m_1, m_2 \in M \text{ s.t. } m_1 \neq m_2, \forall p_1 \in \mathit{defines}(m_1), \forall p_2 \in \mathit{uses}(m_2) \setminus \mathit{defines}(m_2) : \\ \big(\mathbf{sched}[m_1, b] \wedge \mathbf{sched}[m_2, b] \wedge \mathbf{alt}[p_1] = \mathbf{alt}[p_2]\big) \\ \Rightarrow \mathbf{cycle}[m_1] + \mathit{lat}(m_1) \leq \mathbf{cycle}[m_2]. \end{aligned} \tag{8.2}$$

For instructions with latency greater than 1, we must make sure that the result is not used before it is produced. To model these restrictions, we will apply the same approach as in [64] by using the cumulative constraint introduced in Chap. 3 on p. 48. Let $\mathit{cap}(r) \in \mathbb{N}$ denote the capacity of a resource $r \in R$, where R denotes the set of resources in the target machine. In this case, we assume there is a resource r used by all instructions for which $\mathit{cap}(r) = 1$, thus preventing the durations of the instructions from overlapping. Let also $\mathit{lat}(m) \in \mathbb{N}$ denote the instruction latency of a match m, and $\mathit{req}(m, r) \in \mathbb{N}$ denote the amount of resource r used by match m. With these definitions, this constraint can be modeled as

$$\begin{aligned} \forall r \in R, \forall b \in B : \\ \textsc{cumulative}\big(\mathit{cap}(r), \cup_{m \in M} \langle \mathbf{cycle}[m], \mathit{lat}(m), \mathit{req}(m, r), \mathbf{sched}[m, b] \rangle\big). \end{aligned} \tag{8.3}$$

Note that Eq. 8.3 supports modeling of VLIW architecture,[1] as the number of instructions that can be run in parallel, and the functional units used by the instructions, can be modeled as additional resources.

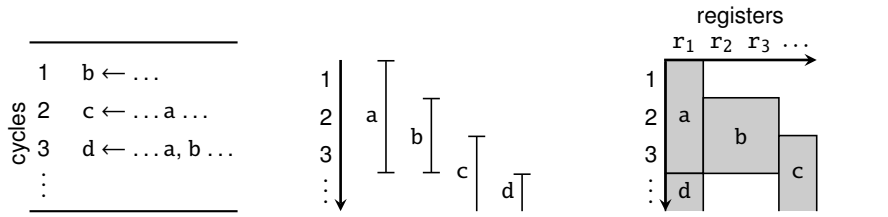
8.2 Integrating Register Allocation

In this context, register allocation can be described as the problem of assigning a location to each datum d such that the value in the location is preserved until the last use of d. Like with instruction selection, this problem can be considered either at local or global scope. In *local register allocation*, registers are allocated to the variables in the function one block at a time, whereas *global register allocation* does so for the entire function. We begin by modeling the local problem and then extend it into the global problem.

8.2.1 Local Register Allocation

We will describe the problem using the example is shown in Fig. 8.1. Assume

[1] A *very long instruction word (VLIW)* architecture is a processor where multiple instructions can be executed in parallel. The schedule is static and thus computed by the compiler.



(a) Scheduled instructions. (b) Live ranges. (c) Rectangle packing problem.

Figure 8.1: Example of local register allocation. It is assumed that variables a, c, and d each fit inside a single register whereas variable b requires two adjacent registers.

that a set of instructions in a given block have already been scheduled (Fig. 8.1a). From the schedule a *live range* is computed for each variable, which is the point in time where the variable is defined until the point where it is last used (Fig. 8.1b). If two variables have overlapping live ranges, then they cannot be assigned the same register as one of the values would be overwritten before its last use. By representing each variable by a rectangle, the problem of assigning registers can be modeled as a rectangle packing problem [310] (Fig. 8.1c). The height of each rectangle corresponds to variable's live range, and the width corresponds to the size of the value (that is, the number of registers it requires).

For simplicity, let us assume for now that we only consider functions containing a single block as this removes the need of having to deal with live ranges spanning multiple blocks. Since the live ranges depend on instruction scheduling, we build upon the constraint model proposed in Sect. 8.1.

Variables The two sets of variables $\mathbf{start}[d] \in \mathbb{N}$ and $\mathbf{end}[d] \in \mathbb{N}$ model the start respectively end of the live range for a datum d. In addition, the set of variables $\mathbf{alive}[d] \in \{0, 1\}$ models whether d is alive (in other words, d is not killed), and the set of variables $\mathbf{using}[m, d] \in \{0, 1\}$ models whether match m uses datum d. The register to which d is assigned is already modeled by the **loc** variables.

Constraints First, we constrain the **alive** variables according to the definition of what it means for data to be killed. This can be modeled as

$$\forall d \in D : \mathbf{alive}[d] \Leftrightarrow \mathbf{loc}[d] = l_{\text{KILLED}}. \tag{8.4}$$

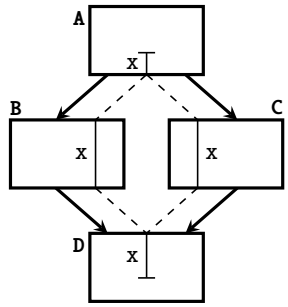
Now, if a datum d is not killed, then the start of a live range for a datum d is determined by the match defining d. Otherwise, the start is set to zero. This can be modeled as

$$\forall d \in D, \forall m \in M_d : \mathbf{alive}[d] \Rightarrow \mathbf{start}[d] = \mathbf{cycle}[\mathbf{dmatch}[m]], \tag{8.5}$$

$$\forall d \in D : \neg\mathbf{alive}[d] \Rightarrow \mathbf{start}[d] = 0. \tag{8.6}$$

```
A: x ← ...
   c.br ..., B, C
B: ...
   br D
C: ...
   br D
D: ...
   y ← ...x...
```

(a) Scheduled instructions.



(b) Live ranges of x.

Figure 8.2: Example of global register allocation.

Similarly, the end of a live range for a live datum d is determined by the match making the last use of d. If there is no use of d, then $\mathbf{end}[d]$ is set to $\mathbf{start}[d]$ to not affect register allocation and instruction scheduling for other data. This can jointly be modeled as

$$\forall d \in D : \mathbf{end}[d] = max\big(\cup_{m \in usersOf(d)} \mathbf{cycle}[m] \times \mathbf{using}[m, d] \cup \{\mathbf{start}[d]\}\big), \quad (8.7)$$

where

$$usersOf(d) \equiv \{m \mid m \in M, \exists p \in uses(m) : d \in D_p\}. \quad (8.8)$$

A match m uses a datum d if and only if m is selected and one of its operands is connected to d. This can be modeled as

$$\forall d \in D, \forall m \in M, \forall p \in uses(m) : \mathbf{using}[m, d] \Leftrightarrow \mathbf{sel}[m] \wedge \mathbf{alt}[p] = d. \quad (8.9)$$

To model the rectangle packing problem, we use the no-overlap constraint introduced in Chap. 3 on p. 49. Thus the rectangle packing problem can be modeled as

$$\textsc{nooverlap}\big(\cup_{d \in D} \langle \mathbf{start}[d], \mathbf{end}[d], \mathbf{loc}[d], \mathbf{loc}[d] + widthOf(d)\rangle\big), \quad (8.10)$$

where $widthOf(d) \in \mathbb{N}$ denotes the number of registers required for datum d.

8.2.2 Global Register Allocation

The main problem of extending local register allocation to global scope is that placements of matches to blocks need to be taken into consideration. Furthermore, the live ranges are no longer necessarily limited within a single block but may span multiple blocks. An example is shown in Fig. 8.2. Assume a function where a variable x is defined in one block A and used in another block D (Fig. 8.2a). Consequently, x must be live to the end of A, through all blocks between A and D (that is, blocks B and C), and until the last use in D (Fig. 8.2b).

We extend the constraint model proposed for local register allocation by first introducing **start** and **end** variables for each block in the function, together with an additional set of variables, and then extending the constraints to handle multiple blocks.

Variables The two sets of variables $\mathbf{start}[d, b] \in \mathbb{N}$ and $\mathbf{end}[d, b] \in \mathbb{N}$ model the start respectively end of the live range for a datum d in block b. In addition, given a datum d and a block b the set of variables $\mathbf{useafter}[d, b] \in \{0, 1\}$ models whether there exists some selected match using d in some blocks that can be reached from b through one or more jumps.

Constraints Even with multiple blocks, the start of a live range is still determined by the match defining the datum. Moreover, the constraint that no two rectangles belonging to the same block may overlap still applies. Hence Eqs. 8.5, 8.6, and 8.10 are adjusted accordingly (the changes are highlighted in gray):

$$\begin{gathered}\forall d \in D, \forall b \in B, \forall m \in M_d : \\ (\mathbf{alive}[d] \wedge \mathbf{dplace}[d] = b) \Rightarrow \mathbf{start}[d, b] = \mathbf{cycle}[\mathbf{dmatch}[m]],\end{gathered} \tag{8.11}$$

$$\forall d \in D, \forall b \in B : (\neg\mathbf{alive}[d] \vee \mathbf{dplace}[d] \neq b) \Rightarrow \mathbf{start}[d, b] = 0, \tag{8.12}$$

$$\begin{gathered}\forall b \in B : \\ \textsc{nooverlap}\big(\cup_{d \in D} \langle \mathbf{start}[d, b], \mathbf{end}[d, b], \mathbf{loc}[d], \mathbf{loc}[d] + \mathit{widthOf}(d)\rangle\big).\end{gathered} \tag{8.13}$$

We now constrain the **useafter** variables according to the definition above. This can be modeled as

$$\begin{gathered}\forall d \in D, \forall b_1, b_2 \in B \text{ s.t. } b_2 \in \mathit{branches}(b_1) : \\ \mathbf{useafter}[d, b_1] \Leftrightarrow \big(_g\exists m \in M : \mathbf{using}[m, d] \wedge \mathbf{sched}[m, b_2]\big) \vee \mathbf{useafter}[d, b_2],\end{gathered} \tag{8.14}$$

where $\mathit{branches}(b) \subseteq B$ gives the set of blocks that can be reached from block b through a single branch.

Intuitively, given a datum d and a block b, if d is used in some block that can be reached from b, then the live range of d must extend until the end of the schedule for b. This can be modeled as

$$\begin{gathered}\forall d \in D, \forall b \in B : \\ \mathbf{useafter}[d, b] \Rightarrow \mathbf{end}[d, b] = \mathit{max}(\cup_{d' \in D \text{ s.t. } d' \neq d} \mathbf{end}[d', b]) \cup \{0\}.\end{gathered} \tag{8.15}$$

Moreover, for every block b, either d is not used in b or the last use of d occurs in b. This can jointly be modeled as

$$\begin{gathered}\forall d \in D, \forall b \in B : \neg\mathbf{useafter}[d, b] \Rightarrow \\ \mathbf{end}[d, b] = \mathit{max}\left(\begin{array}{c}\{\mathbf{start}[d, b]\} \cup \\ \cup_{m \in \mathit{usersOf}(d)} \mathbf{cycle}[m] \times \mathbf{sched}[m, b] \times \mathbf{using}[m, d]\end{array}\right)\end{gathered} \tag{8.16}$$

Lastly, Eqs. 8.15 and 8.16 implicitly assume that φ-matches are always scheduled first.[2] This can be ensured by

$$\forall m \in M_\varphi : \mathbf{cycle}[m] = 0. \tag{8.17}$$

[2]This is because data defined and used within loops must be live across the entire loop, meaning the live ranges must start at the beginning of the block. Since the φ-matches define the data to be used within the loop, these must appear first in the schedule.

CHAPTER

9

Conclusions and Future Work

This chapter closes the dissertation by presenting conclusions in Sect. 9.1 and future work in Sect. 9.2.

9.1 Conclusions

This dissertation has introduced *universal instruction selection* – a new approach that, for the first time, integrates instruction selection, global code motion, and block ordering.[1]

By doing so, it addresses several limitations of existing instruction selection techniques that have been identified through a comprehensive and systematic literature survey. First, none of these can readily be extended for integrating global code motion or block ordering. Second, all existing combinatorial approaches are restricted to tree- and DAG-shaped patterns. Third, with the exception of Tanaka et al. [348], no combinatorial approach takes the cost of data copying into account. Fourth, all combinatorial approaches only deal with data flow. Consequently, the existing approaches fail to exploit many of the instructions provided by modern processors, thereby sacrificing code quality.

To handle the combinatorial nature of these problems, the approach is based on constraint programming. It relies on a novel combinatorial model that is simpler and more flexible compared to the techniques currently used in modern compilers. In addition to integrating instruction selection, global code motion, and block ordering, the model also integrates data copying and value reuse. Data copying takes the cost of moving data into account, which is crucial for avoiding greedy use of SIMD instructions. This feature is currently ignored by nearly all existing combinatorial approaches. Value reuse enables copies of values to be shared among instructions, which is crucial for code quality. The dissertation has also proposed extensions to

[1] The source code is freely available on `github.com/unison-code/uni-instr-sel`.

the model for integrating instruction scheduling and register allocation, which are two other important tasks of code generation.

The model is enabled by the *universal representation* – a novel, graph-based representation that unifies data flow and control flow for entire functions. Not only is the universal representation crucial for combining instruction selection with global code motion, it also enables instructions whose behavior contains both data and control flow to be modeled as graphs. Hence there is no longer need for hand-written routines to handle instructions that violate underlying assumptions about the instruction set.

To make the approach work in practice, numerous solving techniques have been introduced. Through experimental evaluation, it has been shown that these techniques collectively improve solving time up to 132⧖, with a GMI of 6.21×. Hence these solving techniques are crucial for scalability and enable the approach to scale up to medium-sized functions.

The approach has been experimentally evaluated in comparing the code quality it produces for Hexagon – a DSP with a rich instruction set – with that produced by LLVM – a existing, state-of-the-art compiler. In these experiments, the approach improves the estimated execution time of functions by up to 0.181⧖, with a GMI of 1.03×. Hence the approach can handle hardware architectures with rich instruction sets and generates code of equal or better quality compared to the state of the art.

Experiments have also been performed to evaluate the impact of SIMD instruction selection. The results show that the approach can improve code quality when such instructions are available. In one case the computations originally reside in different blocks, but due to global code motion the approach is able to move the computations to the same block and implement these using a single instruction. Hence there is sufficient data parallelism to be exploited through selection of SIMD instructions without having to resort to loop unrolling Moreover, this exploitation benefits from global code motion.

With these results, the dissertation has shown that constraint programming is a flexible, practical, competitive, and extensible approach for combining global instruction selection, global code motion, and block ordering.

9.2 Future Work

Generating Executable Code So far the quality of the code generated using universal instruction selection has only been statically estimated. For a more accurate evaluation – and for putting the approach into practical use – the generated code must be hooked into the post-instruction selection step of an existing compiler. While this is primarily an engineering task, it is also a method for evaluating its applicability.

Selecting Instructions for X86 Since X86 is extremely common among today's processors, it is of great interest to evaluate universal instruction selection for

this architecture. Like Hexagon, X86 has a rich instruction set that is especially geared towards SIMD instructions, with AVX-512 as its latest extension [200]. The instruction set also has many instructions, such as hardware loops, whose behavior contains control flow and can therefore not be modeled using standard representations. In addition, most modern compilers are highly optimized for generating code for this architecture, which means that even a minor gain in code quality is considered a significant improvement.

Integrating Recomputation As discussed in Chap. 5, the constraint model does not support recomputation (the task of recomputing values appearing in common subexpressions instead of reusing them), which can have negative impact on code quality. Moreover, recomputation is essential for supporting if-conversion (the task of converting if-then-else statements into straight-line assembly code), which can improve code quality.

Adapting the constraint model to integrate recomputation is a complex undertaking as many of the solving techniques rely on the fact that every operation and datum is covered respectively defined exactly once. Hence further research is needed for either reformulating the solving techniques or redesigning the model.

Integrating Instruction Scheduling and Register Allocation It is well known that the three main problems of code generation – instruction selection, instruction scheduling, and register allocation – are interconnected with one another. Therefore, in order to generate truly optimal code one must solve all these problems in unison.

Such extensions of the constraint model have already been proposed, but they lack the necessary model refinements and solving techniques that are crucial for making the model scale beyond anything larger but the smallest of functions. Hence further research is needed for making such an approach work in practice.

APPENDIX

A

Macro Expansion

This appendix considers techniques based on macro expansion. First, we introduce the principle in Sect. A.1. We then describe early applications in Sect. A.2, moving on to more sophisticated techniques in Sect. A.3. We discuss limitations of this principle in Sect. A.4 and then summarize in Sect. A.5.

The appendix is based on material presented in [186, Chap. 2] that has been adapted for this dissertation. To not disturb the flow of reading, material already presented in Chap. 2 is duplicated in this appendix. The techniques described here includes all those covered in earlier surveys by Cattell [68] and Ganapathi et al. [153], several of which are also discussed in depth by Lunell [260]. In [153] this principle is called *interpretative code generation*.

A.1 The Principle

The first principle to emerge was macro expansion, with applications starting to appear in the 1960s. In macro expansion, the instructions are expressed as *macros* which consist of two parts: a *template* to be matched over the function under compilation, and an *expand procedure* to be executed upon the part of the function that was matched. An example of such a procedure is given in Fig. A.1. A *macro expander* traverses the function and tries to match the templates of the macros, typically in the order they are declared in the machine description. Upon a match it executes the corresponding expand procedure and then resumes the traversal with the next, unmatched part until the entire function has been expanded. Consequently, the matching and selection problems are combined into a single task as the first macro matched is also the selected macro.

The main benefit of macro expansion is that it is intuitive and straightforward to apply. Because the macro expander is implemented separately from the macros, the former can be kept generic and simple while the latter can be made as customized as needed for the target machine. This also allows the macro expander to be void

```
expand($3 ← $1 + $2) {
  r1 = getRegOf($1);
  r2 = getRegOf($2);
  r3 = mkNewReg($3);
  print "add " + r3 + ", " + r1 + ", " + r2;
}
```

Figure A.1: Example of a macro expanding an IR addition into assembly code. The template to match is given as argument to `expand`, and the procedure to run upon expansion is given as `expand`'s body.

```
* = CAR.*.
    I = CDR('21)
    CDR('11) = CAR(I).
.X
```

(a) A macro definition.

```
A = CAR B.
```

(b) String that matches the template.

```
I = CDR(38)
CDR(36) = CAR(I)
```

(c) After macro expansion.

Figure A.2: Example of macro expansion using SIMCMP [294].

of any target-specific details, thus requiring only the macros to be rewritten when retargeting the compiler to another machine. To this end, the macros are typically written in some dedicated language in order to simplify this task by raising the level of abstraction.

A.2 Naive Macro Expansion

A.2.1 Early Applications

We will refer to instruction selectors that directly apply the principle just described as *naive macro expanders*, for reasons that will soon become apparent. In the first such implementations, the macros were either written by hand – like in the Pascal compiler developed by Ammann et al. [12, 13] – or generated automatically from a machine description, typically written in some dedicated language. Consequently, many such languages and related tools have appeared – and then disappeared – over the years (see for example [58] for an early survey).

One such example is *SIMCMP*, a macro expander developed in 1969 by Orgass and Waite [294]. Designed to facilitate bootstrapping,[1] SIMCMP read its input line by line, compared the line against the templates of the available macros (see Fig. A.2 for an example), and then executed the first macro that matched.

Another example is the GCL, developed by Elson and Rake [112], which was used in a PL/1 compiler for generating assembly code from *abstract syntax trees (ASTs)*,

[1] *Bootstrapping* is the process of writing a compiler in the programming language it is intended to compile.

(a) IR code. (b) Expression tree.

Figure A.3: Example of an expression tree.

which are graph-based representations of the source code that are always shaped like trees. The most important feature of these trees is that only a syntactically valid function can be transformed into an AST, which simplifies the task of the instruction selector. However, the basic principle of macro expansion remains the same.

A.2.2 Using IR Instead of ASTs

Performing instruction selection directly on the source code, either in its textual form or on the AST, carries the disadvantage of tightly coupling the backend to a particular programming language. Most compiler infrastructures therefore rely on some lower-level, machine-independent *intermediate representation (IR)* which isolates the subsequent target-independent optimizations and the backend from the details of the programming language. The IR code is often represented as an *expression tree*, which is a tree-shaped data-flow graph (see Fig. A.3). It is common to omit any intermediate variables from the expression tree and only keep those signifying the input and output values of the expression, as shown in the example. This also means that an expression tree can only represent a set of computations performed within the same block, which thus may contain more than one expression tree. Since these representations only capture data flow, the function's control flow is represented separately as a control-flow graph.

One of the first IR-based schemes was developed by Wilcox [367]. Implemented in a PL/C compiler, the AST is first transformed into machine-independent code consisting of *source language machine (SLM) instructions*. The instruction selector then maps each SLM instruction into one or more target-specific instructions using macros defined in a language called *Interpretive Coding Language (ICL)* (see Fig. A.4 for an example). In practice, these macros turned out to be tedious and difficult to write. Many details, such as addressing modes and data locations, had to be dealt with manually from within the macros. In the case of ICL, the macro writer also had to keep track of which variables were part of the final assembly code, and which variables were auxiliary and only used to aid the code generation process. In an attempt to simplify this task, Young [378] proposed (but never implemented) a higher-level language called *Template Language (TEL)* that would abstract away some of the implementation-oriented details. The idea was to first express the macros as TEL code and then to automatically generate the lower-level ICL macros from the

```
ADDB   BR    A,ADDB1        If A is in a register, jump to ADDB1
       BR    B,ADDB2        If B is in a register, jump to ADDB2
       LGPR  A              Generate code to load A into register

ADDB1  BR   B,ADDB3         If B is in a register, jump to ADDB3
       GRX  A,A,B           Generate A+B
       B    ADDB4           Merge

ADDB3  GRR   AR,A,B         Generate A+B
ADDB4  FREE  B              Release resources assigned to B
ADDB5  POP   1              Remove B descriptor from stack
       EXIT

ADDB2  GRI   A,B,A          Generate A+B
       FREE  A              Release resources assigned to A
       SET   A,B            A now designates result location
       B     ADDB5          Merge
```

Figure A.4: A binary addition macro in ICL [367].

machine description.

A.2.3 Generating the Macros from a Machine Description

As with Wilcox's design, many of the early macro-expanding instruction selectors depended on macros that were intricate and difficult to write. In addition, many compiler developers often incorporated register allocation into these macros, which further exacerbated the problem. For example, if the target machine exhibits multiple sets of registers, called *register classes*, and has special instructions to move data from one class to another, a record must be kept of which value reside in which register. Then, depending on the register assignment, the instruction selector needs to emit the appropriate data-transfer instructions in addition to the rest of the assembly code. Due to the exponential number of possible situations, the complexity that the macro designer has to manage can be immense.

Automatically Inferring Necessary Data Transfers The first attempt to address this problem was made by Miller [275]. In his master's thesis from 1971, Miller introduces a code generation system called *Dmacs* that automatically infers the necessary data transfers between memory and different register classes. By encapsulating this information in a separate machine description, Dmacs was also the first system to allow the details of the target machine to be declared separately instead of being implicitly embedded into the macros.

Dmacs relies on two proprietary languages. The first language, *Machine-Independent Macro Language (MIML)*, declares a set of procedural two-argument commands that serves as the IR format (see Fig. A.5 for an example). The second language, *Object Machine Macro Language (OMML)*, is a declarative language used for

```
1:  SS    C,J
2:  IMUL  1,D
3:  IADD  2,B
4:  SS    A,I
5:  ASSG  4,3
```

Figure A.5: An example on how an arithmetic expression `A[I] = B + C[J] * D` is represented using MIML commands [275]. The `SS` command is used for data referencing and the `ASSG` command assigns a value to a variable. The arguments to the MIML commands are referred to either by a variable symbol or by line number.

```
rclass REG:r2,r3,r4,r5,r6
rclass FREG:fr0,fr2,fr4,fr6
...
rpath WORD->REG: L REG,WORD
rpath REG->WORD: ST REG,WORD
rpath FREG-WORD: LE FREG,WORD
rpath WORD->FREG: STE FREG,WORD
...
ISUB s1,s2
from REG(s1),REG(s2) emit SR s1,s2  result REG(s1)
from REG(s1),WORD(s2) emit S s1,s2  result REG(s2)

FMUL m1,m2 (commutative)
from FREG(m1),FREG(m2) emit MER m1,m2  result FREG(m1)
from FREG(m1),WORD(m2) emit ME m1,m2   result FREG(m1)
```

Figure A.6: Partial machine description for IBM-360 in OMML [275]. The `rclass` command declares a register class, and the `rpath` command declares a permissible transfer between a register class and memory (or vice versa) along with the instruction that implements the transfer.

implementing the macros that will transform each MIML command into assembly code. So far this scheme is similar to the one applied by Wilcox.

When adding support for a new target machine, a macro designer first specifies the set of available register classes (including memory) as well as the permissible transfer paths between these classes. The macro designer then defines the OMML macros by providing, for each macro, a list of instructions that implements the corresponding MIML command on the target machine. If necessary, a sequence of instructions can be given to emulate the effect of a single MIML command. Lastly, constraints are added that force the input and output data to reside in the locations expected of the instruction. Figure A.6 shows excerpts of an OMML specification for an IBM machine.

Dmacs uses this information to generate a collection of finite state automata (or *state machines*, as they are also called) to determine how a given set of input values can be transferred into locations that are permissible for a given OMML macro. Each state machine consists of a directed graph where a node represents a

specific configuration of register classes and memory, some of which are marked as permissible. The edges indicate how to transition from one state to another, and are labeled with the machine instruction that will enable the transition when executed on a particular input value. During compilation the instruction selector consults the appropriate state machine as it traverses from one MIML command to the next, using the input values of the former to initialize the state machine. As the state machine transitions from one state to another, the machine instructions appearing on the edges are emitted until the state machine reaches a permissible state.

The work by Miller was pioneering but limited: DMACS only handled arithmetic expressions consisting of integer and floating-point values, its addressing mode support was limited, and it could not model other target machine classes such as stack-based architectures. In his 1973 doctoral dissertation, Donegan [104] extended Miller's ideas by proposing a new language called *Code Generator Preprocessor Language (CGPL)*. Donegan's proposal was put to the test in the 1978 master's thesis by Maltz [264], and was later extended by Donegan et al. [103]. Similar techniques have also been developed by Tirrell [350] and Simoneaux [338]. Ganapathi et al. [153] also describe in their survey another state machine-based compiler called UGEN, which was derived from a virtual machine called *U-CODE* [301].

Further Improvements In 1975, Snyder [340] implemented one of the first fully operational and portable C compilers, where the target machine-dependent parts could be automatically generated from a machine description. The design is similar to Miller's in that the frontend first transforms the function into an equivalent representation for an abstract machine. In Snyder's design this representation consists of *abstract machine operations (AMOPs)*, which are then expanded into target-specific instructions via macros. The abstract machine and macros are specified in a machine description language which is also similar to Miller's, but handles more complex data types, addressing modes, alignment, as well as branching and function calls. If needed, more complicated macros can be defined as customized C functions. We mention Snyder's work primarily because it was later adapted by Johnson [203] in his implementation of PCC, which we will discuss in Ap. B.

Fraser [143, 144] also recognized the need for human knowledge to guide the code generation process, and implemented a system with the aim of facilitating the addition of handwritten rules when these are required. First the function is transformed into a representation based on a programming language called *Extensible Language (XL)*, which is akin to high-level assembly code. For example, XL provides primitives for array accesses and for loops. As in the cases of Miller and Snyder, the instructions are provided via a separate description that maps directly to a distinct XL primitive. If some portion of the function cannot be implemented by any of the available instructions, the instruction selector will invoke a set of rules to rewrite the XL code until a solution is found. For example, array accesses are broken down into simpler primitives, and the same rule base can also be used to improve the code quality of the generated assembly code. Since these rules are provided as

instruction	bit string
L B2, D(0, BD)	XXXXXXXX00000000
LH B2, D(0, B2)	0000111100000000
LR R1, R2	0000110100001101

Table A.1: Example of instruction bit strings [259]. An X means that it will always match any bit in a given bit string.

a separate machine description, they can be customized and augmented as needed to fit a particular target machine.

As we will see, this idea of "massaging" the function until a solution can be found has been applied, in one form or another, by many instruction selectors that both predate and succeed Fraser's design. Although they represent a popular approach, a significant drawback of such schemes is that the instruction selector may get stuck in an infinite loop if the set of rules is incomplete for a particular target machine, and determining if this is the case is often far from trivial. Moreover, such rules tend to be hard to reuse for other target machines.

A.2.4 Reducing Compilation Time with Tables

Despite their already simplistic nature, macro-expanding instruction selectors can be made even more so by representing the 1-to-1 or 1-to-n mappings as sets of tables. This further emphasizes the separation between the machine-independent core of the instruction selector from the machine-dependent mappings, as well as allows for denser implementations that require less memory and potentially reduce the compilation time.

Instruction Selection Using Bit Strings In 1969 Lowry and Medlock [259] introduced one of the first table-driven methods for code generation.

In their implementation of the *Fortran H Compiler (FHC)*, Lowry and Medlock essentially perform instruction selection after register allocation depending on the status of the operands to a computation. These statuses are represented as bit strings, which are then matched against the corresponding bit strings for the instructions (see Tab. A.1 for examples). The bits represent restrictions applied by the instructions, for example that the first operand must be fetched from memory, the second operand must reside in a register whose content will be erased because the result will be placed in the same register.

The main disadvantage of Lowry and Medlock's design was that the tables could only be used for the most basic of instructions, and had to be written by hand in the case of FHC. More extensive designs were later developed by Tirrell [350] and Donegan [104], but these also suffered from similar disadvantages of making too many assumptions about the target machine, thus hindering compiler retargetability.

Expanding Macros Top-Down Later Krumme and Ackley [229] introduced a table-driven design which, unlike the earlier techniques, exhaustively enumerates all valid combinations of selectable instructions, schedules, and register allocations for a given expression tree. Implemented in a C compiler targeting DEC-10 machines, the technique also allows code size to be factored in as an optimization goal, which was an uncommon feature at the time. Krumme and Ackley's backend applies a recursive algorithm that begins by selecting instructions for the root in the expression tree, and then working its way down. In comparison, the bottom-up techniques we have examined so far all start at the leaves and then traverse upwards. We settle with this distinction for now as we will resume and deepen the discussion of bottom-up vs. top-down instruction selection in Ap. B.

Enumerating all valid combinations in code generation leads to a combinatorial explosion, thus making it impossible to actually produce and check each and every one of them. To curb this immense complexity, Krumme and Ackley applied a variant of branch and bound as search strategy (see Chap. 3 on p. 53).[2] The problem is how to prove that a given branch in the search space will definitely lead to solutions that are worse than what we already have (and can thus be skipped). Krumme and Ackley only partially tackled this problem by pruning away branches that for sure will eventually lead to failure and thus yield no solution whatsoever. Without going into too much detail, this is done by using not just a single instruction table but several – one for each so-called *mode* – which are constructed in a hierarchical manner. In this context, a mode is oriented around the result of an expression, for example whether it is to be stored in a register or in memory. Using these tables, the instruction selector can look ahead and detect whether the current set of already-selected instructions will lead to a dead end. With this as the only method of branch pruning, however, the instruction selector will make many needless revisits in the search space, and consequently does not scale to larger expression trees.

A.2.5 Falling Out of Fashion

Despite the improvements we have just discussed, they still do not resolve the main disadvantage of macro-expanding instruction selectors – namely, that they can only handle macros that expand a single AST or IR node at a time. The limitation can be somewhat circumvented by allowing information about the visited nodes to be forwarded from one macro to the next, thereby postponing assembly code emission in the hopes that more efficient instructions can be used. However, if done manually – which was often the case – this quickly becomes an unmanageable task for the macro writer, in particular if backtracking becomes necessary due to faulty decisions made in prior macro invocations.

[2] In their paper, Krumme and Ackley actually call this *α-β pruning*, which is an entirely different search strategy, but their description of it fits more the branch and bound approach. Both are well explained in [323].

Thus naive macro expanders are effectively limited to supporting only single-output instructions.[3] This has a detrimental effect on code quality for target machines exhibiting more complicated features, such as multi-output instructions Consequently, instruction selectors based solely on naive macro expansion were quickly replaced by newer, more powerful techniques that started to appear in the late 1970s. One of these we will discuss later in this appendix.

Rekindled Application in the First Dynamic Code Generation Systems Having fallen out of fashion, naively macro-expanding instruction selectors later made a brief reappearance in the first dynamic code generation systems that were developed in the 1980s and 1990s. In such systems the function is first compiled into *byte code*, which is a kind of target-independent machine code that can be interpreted by an underlying runtime environment. By providing an identical environment on every target machine, the same byte code can be executed on multiple systems without having to be recompiled.

The cost of this portability is that running a function in interpretive mode is typically much slower than executing native machine code. This performance loss can be mitigated by incorporating a compiler into the runtime environment. First, the byte code is profiled as it is executed. Frequently executed segments, such as inner loops, are then compiled into native machine code. Since the code segments are compiled at runtime, this scheme is called *just-in-time (JIT) compilation*, which allows performance to be increased while retaining the benefits of the byte code. If the performance gap between running byte code instead of native machine code is large, then the compiler can afford to produce assembly code of low quality in order to decrease the overhead in the runtime environment. This was of great importance for the earliest dynamic runtime systems where hardware resources were typically scarce, which made macro-expanding instruction selection a reasonable option. A few examples include interpreters for Smalltalk-80 [99] and Omniware [1] (a predecessor to Java). More examples include code generation systems such as *VCODE* [117], *GBURG* [142] (used inside a small virtual machine), and *GNU LIGHTNING* [15] (a code generation library inspired by VCODE).

As technology progressed, however, dynamic code generation systems also began to transition to more powerful techniques for instruction selection such as tree covering, which will be described in Ap. B.

A.3 Improving Code Quality with Peephole Optimization

An early but still applied method of improving the quality of generated assembly code is to perform a subsequent program optimization step that attempts to combine and replace several instructions with shorter, more efficient alternatives. These

[3] This is a truth with modification. A macro expander can emit multi-output instructions, but only one of its output values will be retained in the assembly code.

routines are known as *peephole optimizers* for reasons which will soon become apparent.

A.3.1 What Is Peephole Optimization?

In 1965, McKeeman [274] advocated the use of a simple but often neglected program optimization procedure known as *peephole optimization*. As a post-step to code generation, peephole optimization inspects a small sequence of instructions in the assembly code and attempts to combine two or more adjacent instructions with a single instruction. The name is thus derived from its narrow window of observation, and similar ideas were also suggested by Lowry and Medlock [259] around the same time. Doing this reduces code size and improves performance as using complex instructions is often more efficient than using several simpler instructions to implement the same functionality.[4]

Modeling Instructions with Register Transfer Lists Since this kind of optimization is tailored for a particular target machine, the earliest implementations were (and still often are) done ad hoc and by hand. For example, in 2002, Krishnaswamy and Gupta [227] wrote a peephole optimizer by hand which reduces code size by replacing known patterns of ARM code with smaller equivalents. Recognizing the need for automation, Fraser [139] introduced in 1979 the first technique that allowed peephole optimizers to be generated from a formal description. The technique is also described in a longer article by Davidson and Fraser [95].

Like Miller, Fraser described the semantics of the instructions separately in a symbolic machine description. The machine description describes the observable effects that each instruction has on the target machine's registers. Fraser called these effects *register transfers (RTs)*, and each instruction thus has a corresponding *register transfer list (RTL)*. For example, assume that we have a three-address `add` instruction which adds an immediate value `imm` to the value in register $\mathtt{r_s}$, stores the result in register $\mathtt{r_d}$, and sets a zero flag `Z`. For this instruction, the corresponding RTL would be expressed as

$$RTL(\mathtt{add}) = \left\{ \begin{array}{rcl} \mathtt{r_d} & \leftarrow & \mathtt{r_s} + \mathtt{imm} \\ \mathtt{Z} & \leftarrow & \mathtt{r_s} + \mathtt{imm} = 0 \end{array} \right\}.$$

The RTLs are then fed to a program called *Peephole Optimizer (PO)*, which produces a program optimization routine that makes two passes over the generated assembly code. The first pass runs backwards across the assembly code to determine the observable effects (that is, the RTL) of each instruction in the assembly code. This allows effects that have no impact on the function's observable behavior to be removed. For example, if the value of a status flag is not read by any subsequent instruction, it is considered to be *unobservable* and can thus be ignored. The second pass then checks whether the combined RTLs of two adjacent instructions are equal

[4] This idea was applied by Cho et al. [74] for reselecting instructions in order to improve iterative modulo schedules for DSPs.

to that of some other instruction (in PO this check is done via a series of string comparisons). If such an instruction is found, the pair is replaced and the routine backs up one instruction in order to check the combination of the new instruction with the following instruction in the assembly code. This way replacements can be cascaded and many instructions reduced into a single equivalent, provided there exists an appropriate instruction for each intermediate step.

Pioneering as it was, PO also had several limitations. The main drawbacks were that it only supported combinations of two instructions at a time, and that these had to be lexicographically adjacent in the assembly code. The instructions were also not allowed to cross block boundaries, meaning that they had to belong to the same block. Davidson and Fraser [93] later removed the limitation of lexicographical adjacency by making use of data-flow graphs instead of operating directly on the assembly code. In addition, they extended the size of the instruction window from pairs to triples.

Further Developments Much research has been dedicated to improving automated approaches to peephole optimization. In 1983, Giegerich [161] proposed a formal design that eliminates the need for a fixed-size instruction window. Shortly after, Kessler [215] introduced a method where RTL combinations and comparisons can be precomputed as the compiler is built, thus decreasing compilation time. Kessler [214] later expanded his work to incorporate an n-size instruction window, similar to that of Giegerich, although at an exponential cost.

Another scheme was developed by Massalin [271] who implemented a system called the *Superoptimizer*, and similar systems have subsequently been referred to as *superoptimizers*. The Superoptimizer accepts small functions written in assembly code, and then exhaustively combines sequences of instructions to find shorter implementations that exhibit the same behavior as the original function.[5] Granlund and Kenner [170] later adapted Massalin's ideas into a method that minimizes the number of branches. Both implementations, however, were implemented by hand and customized for a particular target machine. Moreover, neither makes any guarantees on correctness. A technique for automatically generating peephole optimization-based superoptimizers was developed by Bansal and Aiken [34], where the superoptimizer learns to optimize short sequences of instructions from a set of training functions. A couple of designs that guarantee correctness have been developed by Joshi et al. [205, 206] and Crick et al. [90], who applied automatic theorem proving and *answer set programming (ASP)* (see [171] for an overview of ASP), respectively. Recently, a similar technique based on quantifier-free bit-vector logic formulas was introduced by Srinivasan and Reps [343].

[5] The same idea has also been applied by El-Khalil and Keromytis [217] and Anckaert et al. [14], where the assembly code of compiled functions is modified in order to support *steganography* (the covert insertion of secret messages). For example, Anckaert et al. used this technique on nine functions from the *SPECint 2000* benchmark suite in order to embed and extract William Shakespeare's play *King Lear*.

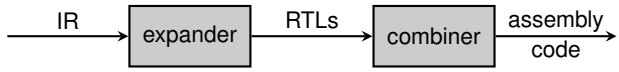


Figure A.7: Overview of the Davidson-Fraser approach.

A.3.2 Combining Naive Macro Expansion with Peephole Optimization

Up to this point peephole optimizers had mainly been used to improve already-generated assembly code – in other words, *after* instruction selection had been performed. In 1984, however, Davidson and Fraser [93] developed an instruction selection technique that incorporates the power of peephole optimization with the simplicity of macro expansion. Similar yet unsuccessful strategies had already been proposed earlier by Auslander and Hopkins [30] and Harrison [177], but Davidson and Fraser struck the right balance between compiler retargetability and code quality which made it a viable option for production-quality compilers. This scheme has hence become known as the *Davidson-Fraser approach*, and variants of it have been used in several compilers, such as the *Y Compiler (YC)* [94], the ZEPHYRVPO system [16], the *Amsterdam Compiler Kit (ACK)* [349], and – most famously – the *GNU Compiler Collection (GCC)* [218, 344].

The Davidson-Fraser Approach In the Davidson-Fraser approach the instruction selector consists of two parts: an *expander* and a *combiner* (see Fig. A.7). The task of the expander is to transform the function into a series of RTLs. The transformation is done by executing simple macros that expand every node in the expression tree (assuming the function is represented as such) into a corresponding RTL that describes the effects of that node. Unlike the previous macro expanders we have discussed, these macros do not incorporate register allocation. Instead the expander assigns each result to a virtual storage location called a *temporary*, of which it is assumed there exists an infinite amount. A subsequent register allocator then assigns each temporary to a register, potentially inserting additional code that saves some values to memory for later retrieval when the number of available registers is not enough (this is called *spilling*). After expansion, but before register allocation, the combiner is run. Using the same technique as that behind PO, the combiner tries to improve code quality by combining several RTLs in the function into a single, larger RTL that corresponds to some instruction on the target machine. For this to work, both the expander and the combiner must at every step adhere to a rule, called the *machine invariant*, which dictates that every RTL in the function must be implementable by a single instruction.

By using a subsequent peephole optimizer to combine the effects of multiple RTLs, the instruction selector can effectively extend over multiple nodes in the AST or expression tree, potentially across block boundaries. The instruction support in Davidson and Fraser's design is therefore in theory only restricted by the number of instructions that the peephole optimizer can compare at a time. For example,

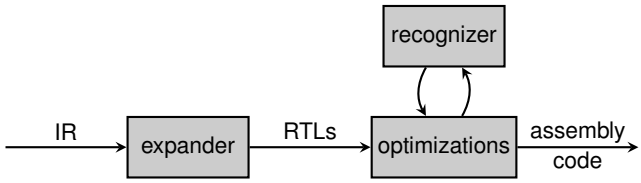


Figure A.8: Overview of Dias and Ramsey's design.

opportunities to replace three instructions by a single instruction will be missed if the peephole optimizer only checks pair combinations. But increasing the window size typically incurs an exponential cost in terms of added complexity, thus making it difficult to handle complicated instructions that require large instruction windows.

Further Improvements Fraser and Wendt [138] later expanded the work by Davidson and Fraser. In a paper from 1988, Fraser and Wendt describe a method where the expander and combiner are effectively fused together into a single component. The idea is to generate the instruction selector in two steps. The first step produces a naive macro expander that is capable of expanding a single IR node at a time. Unlike Davidson and Fraser, who implemented the expander by hand, Fraser and Wendt devised a design that can be automatically generated from a machine description. The design relies on an elaborate scheme consisting of a series of switch and goto statements, which effectively implement a state machine. Once produced, the macro expander is executed on a carefully designed training set. Using function calls embedded into the instruction selector, a retargetable peephole optimizer is executed in tandem which discovers and gathers statistics on target-specific optimizations that can be done on the generated assembly code. Based on these results, the beneficial optimization decisions are then selected and incorporated directly into the macro expander. This effectively enables the macro expander to expand multiple IR nodes at a time, thus removing the need for a separate peephole optimizer in the final compiler. Fraser and Wendt argued that as the instruction selector only implements the optimization decisions that are deemed to be "useful," the code quality is improved with minimal overhead. Wendt [364] later improved the technique by providing a more powerful machine description format, also based on RTLs, which subsequently evolved into a compact standalone language used for implementing code generators (see Fraser [137]).

Enforcing the Machine Invariant with a Recognizer The Davidson-Fraser approach was also extended by Dias and Ramsey [101]. Instead of requiring each separate RTL-oriented optimization routine to abide by the machine invariant, Dias and Ramsey's design employs a *recognizer* to determine whether an optimization decision violates the aforementioned restriction (see Fig. A.8). The idea is that, by

```
default attribute
  add(rd, rs1, rs2) is $r[rd] := $rs[rs1] + $r[rs2]
```

Figure A.9: An `add` instruction from the PowerPC instruction set, specified using λ-RTL [100].

doing so, the optimization routines can be simplified and generated automatically as they no longer need to internalize the machine invariant.

In a paper from 2006, Dias and Ramsey demonstrate how the recognizer can be produced from a declarative machine description written in λ-RTL. Originally developed by Ramsey and Davidson [313], *λ-RTL* is a high-level functional language based on *Metalanguage (ML)* and raises the level of abstraction for writing RTLs (see Fig. A.9 for an example). In their paper, Dias and Ramsey claim that λ-RTL-based machine descriptions are more concise and simpler to write compared to those of many other designs, including GCC. In particular, λ-RTL is precise and unambiguous, which makes it suitable for automated tool generation and verification. The latter has been explored by Fernández and Ramsey [132] and Bailey and Davidson [31].

The recognizer checks whether an RTL in the function fulfills the machine invariant by performing a syntactic comparison between that RTL and the RTLs of the instructions. However, if a given RTL in the function has n operations, and a given λ-RTL description contains m instructions whose RTL contains l operations, then a naive implementation would take $O(nml)$ time to check a single RTL. Instead, using techniques to be discussed in Ap. B, Dias and Ramsey automatically generate the recognizer as a finite state automaton that can compare a given RTL against all RTLs in the λ-RTL description with a single check.

"One Program to Expand Them All" In 2010, Dias and Ramsey [100, 314] introduced a scheme where the macro expander only needs to be implemented once per every distinct *architecture family* instead of once per every distinct *instruction set*. For example, register-based and stack-based machines are two separate architecture families, whereas X86, PowerPC, and Sparc are three different instruction sets. In other words, if two target machines belong to the same architecture family, then the same expander can be used despite the differing details in their instruction sets. This is useful because the correctness of the expander only needs to be proven once, which is a difficult and time-consuming process if it is written by hand.

The idea is to have a predefined set of tiles that are specific for a particular architecture family. A *tile* represents a simple operation which is required for any target machine belonging to that architecture family. For example, stack-based machines require tiles for push and pop operations, which are not necessary on register-based machines. Then, instead of expanding each IR node in the function into a sequence of RTLs, the expander expands it into a sequence of tiles. Since the set of tiles is identical for all target machines within the same architecture family,

the expander only needs to be implemented once. After macro expansion the tiles are replaced by the instructions used to implement each tile, and the resulting assembly code can then be improved by the combiner.

A remaining problem is how to find instructions to implement a given tile for a particular target machine. In the same papers, Dias and Ramsey describe a scheme for doing this automatically. By expressing both the tiles and the instructions as λ-RTL, Dias and Ramsey developed a technique where the RTLs of the instructions are combined such that the effects equal that of a tile. In broad outline, the algorithm maintains a pool of RTLs which initially contains those of the instructions found in the machine description. Using algebraic laws and combining existing RTLs to produce new RTLs, the pool is grown iteratively until either all tiles have been implemented, or some termination criterion is reached. The latter is necessary, as Dias and Ramsey proved that the general problem of finding implementations for arbitrary tiles is undecidable.

Although the primary aim of Dias and Ramsey's design is to facilitate compiler retargetability, some experiments suggest that it potentially also yields better code quality than the original Davidson-Fraser approach. When a prototype was compared against the default instruction selector in GCC, the results favored the former. However, this was seen only when all target-independent optimizations were disabled; when they were reactivated, GCC still produced better results.

A.3.3 Running Peephole Optimization Before Instruction Selection

In the techniques just discussed, the peephole optimizer runs after code generation. But in a scheme developed in 1989 by Genin et al. [157], a similar routine is executed *before* code generation. Targeting DSPs, their compiler first transforms the function into an *internal signal-flow graph (ISFG)*, and then executes a routine – Genin et al. called it a *pattern matcher* – which attempts to find several low-level operations in the ISFG that can be merged into single nodes.[6] Code generation is then done following the conventional macro expansion approach. For each node the instruction selector invokes a rule along with the information about the current context. The invoked rule produces the assembly code appropriate for the given context, and can also insert new nodes to offload decisions that are deemed better handled by the rules corresponding to the inserted nodes.

According to Genin et al., experiments show that their compiler generated assembly code that was five to 50 times faster than that produced by other, contemporary DSP compilers, and comparable with manually optimized assembly code. A disadvantage of this design is that it is limited to functions where prior knowledge about the application area – in this case digital signal processing – can be encoded into specific optimization routines, which most likely has to be done manually.

[6] The paper is not clear on how this is done exactly, but presumably Genin et al. implemented the routine as a handwritten peephole optimizer since the intermediate format is fixed and does not change from one target machine to another.

A.3.4 Interactive Code Generation

The aforementioned techniques yield peephole optimizers which are static once they have been generated, meaning they will only recognize and optimize assembly code for a fixed set of patterns. A method to overcome this issue has been designed by Kulkarni et al. [230], which is also the first and only one to the author's knowledge.

In a paper from 2006, Kulkarni et al. describe a compiler system called *VISTA*, which is an interactive compilation environment where the user is given greater control over the compiler. Among other things, the user can alter RTLs derived from the function's source code and add new customized peephole optimization patterns. Hence optimization privileges which normally are limited to low-level assembly code programmers are also granted to higher-level programming language users. In addition, Kulkarni et al. employed genetic algorithms – these will be explained in Ap. B – in an attempt to automatically derive a combination of user-provided optimization guidelines to improve the code quality of a particular function. Experiments show that this scheme reduced code size on average by 4 % and up to 12 % for a selected set of functions.

A.4 Limitations of Macro Expansion

Because macro-expanding instruction selectors only visit and execute macros one IR node at a time, they require a 1-to-1 or 1-to-n mapping between the IR nodes and the instructions in order to generate efficient assembly code. The limitation can be mitigated by incorporating additional logic and bookkeeping into the macros, but this quickly becomes an unmanageable task for the macro writer if done manually. Consequently, the code quality yielded by such techniques will typically be low. Moreover, as instructions are often emitted one at a time, it also becomes difficult to make use of instructions that can have unintended effects on other instructions.

The Davidson-Fraser approach is therefore a more robust approach, where the instruction selector is augmented with a peephole optimizer. Peephole optimization is inherently only limited by the size of its window of observation and can in theory support instructions of arbitrary complexity. Because of this versatility, the Davidson-Fraser approach remains one of the most powerful instruction selection techniques to date. For example, a variant is still applied in GCC as of version 4.8.2.

A.5 Summary

In this appendix we have discussed techniques and designs based on a principle known as macro expansion, which was the first approach to perform instruction selection. The idea behind the principle is to expand the nodes in the AST or IR code into one or more target-specific instructions. The expansion is done via template matching and macro invocation, which yields instruction selectors that are resource-effective and straightforward to implement. Early techniques, however,

applied this principle naively, yielding poor code quality. Later designs, based on the Davidson-Fraser approach, mitigate this problem by combining instruction selection with peephole optimization, and such techniques are still popular today.

In Ap. B we will explore another principle of instruction selection, which solves the problem of implementing several AST or IR nodes using a single instruction in a more direct fashion.

APPENDIX

B

Tree Covering

This appendix considers techniques based on tree covering, which is the most common principle of techniques found in the current literature. First, we introduce the principle in Sect. B.1. We then describe the first tree-based approaches in Sect. B.2. In Sect. B.3 we describe parser-based approaches, where methods typically used for syntactic analysis is reinstrumented for instruction selection. The techniques thus far are all bottom-up-oriented, and in Sect. B.4 we describe the first top-down-oriented approaches. In Sect. B.5 we describe the first techniques that separate the matching and selection problems, allowing the latter to be solved optimally. In Sect. B.6 we describe other tree-based approaches that do not fit into any of the sections above. Limitations of this principle are discussed in Sect. B.7, and we summarize in Sect. B.8.

The appendix is based on material presented in [186, Chap. 3] that has been adapted for this dissertation. To not disturb the flow of reading, material already presented in Chap. 2 is duplicated in this appendix.

B.1 The Principle

As we saw in Ap. A on p. 152, the main limitation of most instruction selectors based on macro expansion is that the scope of expansion is restricted to a single AST or IR node. Hence exploitation of many instructions is excluded, resulting in low code quality. Another problem is that macro-expanding instruction selectors typically combine matching and selection into a single step, thus making it very difficult to consider combinations of instructions and then pick the one that yields the best assembly code. These problems can be addressed by employing tree covering.

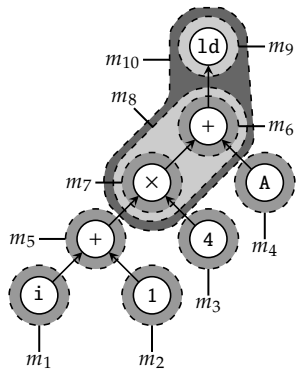
First, the IR code is transformed into an expression tree, as we saw in Ap. A (see Sect. A.2.2 on p. 139). Corresponding data-flow graphs, called *pattern trees*, are also built to represent the instruction provided by the target machine. When the shape

```
x = A[i + 1];
```

(a) C code.

`mv`	$r \leftarrow var$
`add`	$r \leftarrow s + t$
`mul`	$r \leftarrow s \times t$
`muladd`	$r \leftarrow s \times t + u$
`load`	$r \leftarrow *s$
`maload`	$r \leftarrow *(s \times t + u)$

(b) Instructions. The $*s$ notation means "get value at address s in memory".



(c) Expression tree and its matches.

Figure B.1: Example demonstrating the pattern matching and selection problem for a function that loads a value from integer array `A` at offset `i + 1`. It is assumed that `i` is stored in register, that `A` is stored in memory, and that an integer is four bytes. Exact covers are $\{m_1, \ldots, m_7, m_9\}$, $\{m_1, \ldots, m_5, m_8, m_9\}$, $\{m_1, \ldots, m_5, m_{10}\}$, $\{m_1, \ldots, m_5, m_8, m_9\}$ (for brevity, non-exact covers are ignored). Variable assignments need not be explicitly represented as nodes since this information can be propagated from the root node after having found a cover.

is clear from the context, they are simply called *patterns*. The set of patterns for a particular target machine constitute a *pattern set*.

The matching problem can be reduced to finding all instances where a pattern from the pattern set is subgraph isomorphic to the expression tree. Each such instance is called a *match*, and the set of all matches constitute a *match set*. Hence, in this context the matching problem is referred to as *pattern matching*. In many contexts, matches and patterns can be used interchangeably.

Having found the match set, the selection problem – which in this context is referred to as *pattern selection* – can be reduced to selecting a set of matches that covers the expression tree. A subset $M' \subseteq M$, where M is a match set, *covers* a data-flow graph G, derived from a function, if every node in G appears in at least one match in M'. Such a subset is called a *cover*. A cover is an *exact cover* if every node in G appears in exactly one match in the cover. Most instruction selection approaches assume exact coverage. Examples of covers are shown in Fig. B.1.

For most target machines there will be a tremendous amount of overlap among the patterns, meaning that one pattern may match (either partially or fully) the nodes matched by another pattern in the expression tree. Typically we want to use as few patterns as possible to cover the expression tree. This is for two reasons:

- Striving for the smallest number of patterns means favoring larger patterns

over smaller ones. This in turn leads to the use of more complex instructions which typically yield higher code quality.
- The amount of overlap between the selected patterns is limited, which means that the same values will be computed multiple times only when necessary. Keeping redundant work to a minimum is another crucial factor for performance as well as for reducing code size.

In general, an *optimal* solution to the pattern selection problem is not defined as the one that minimizes the *number* of selected patterns, but as the one that minimizes the *total cost* of the selected patterns. This allows the pattern costs to be chosen such that they fit the desired optimization criteria, although there is usually a strong correlation between the number of patterns and the total cost. Note, however, that an optimal solution to the pattern selection problem need not necessarily be an optimal solution for the final assembly code.

Finding the optimal solution to a pattern selection problem is not a trivial task, and it becomes even less so if only certain combinations of patterns are allowed. To be sure, most would be hard-pressed just to come up with an efficient method that finds all valid matches of the entire pattern set. We therefore begin by exploring the first methods that address the pattern matching problem, but do not necessarily address the pattern selection problem, and then gradually transition to those that do.

B.2 First Techniques to Use Tree-Based Pattern Matching

In 1972 and 1973, the first code generation techniques known to use tree-based pattern matching were introduced by Wasilew [361] and Weingart [362], respectively. Unfortunately only Weingart's work appears to be recognized by other literature, even though Wasilew's ideas have more in common with later tree-based instruction selection techniques. We will briefly cover both in this dissertation, as described by Lunell [260], who gives a more detailed account in his doctoral dissertation.

Wasilew's Design To begin with, Wasilew devised an intermediate representation where the functions are represented using *postfix notation* (or *reverse Polish notation* as this is also called; we will discuss more on Polish notation in Sect. B.3.2). An example is shown in Fig. B.2. Wasilew also developed his own programming language, which is transformed into IR code as part of compilation. The instructions of the target machine are described in a table, where each instruction comprises execution time and code size information, a string constituting the assembly code, and the pattern to be matched against the function. For each line in the function, pattern matching is done starting at a leaf in the tree corresponding to the current line. For this subtree, all matches are found by comparing it against all patterns in the pattern set. The subtree is then grown to include its parent, and the new subtree is again compared against the patterns. This continues until no new matches are found. Once the largest match has been found, the subtree is replaced with the

```
AWAY m YHPASS assign
K AMA m PMFI 7 - assign
Z K AMA m ANS assign assign
X 8 + m HEAD X 6 + m I1 + m X 6 + m I2 + m assign
X Y FR AA transfer assign
X INC if-AZ BB transfer
OR m MAJ 4FCOID 4FCOIN if-equal2 1 J + transfer
```

Figure B.2: Example of a function expressed using Wasilew's IR [260].

result of the pattern, and the process is repeated for the remaining parts in the tree. If multiple largest matches are found for any subtree, the process is repeated for each such match. This results in an exhaustive search that finds all combinations of patterns for a given tree. Once all combinations have been found, the cheapest combination – whose cost is based on the instructions' execution time and code size – is selected.

Compared to the early macro-expanding instruction selectors (at least those prior to Davidson-Fraser), Wasilew's design had a more extensive instruction support as it could include patterns that extend over multipleIR nodes. However, its exhaustive nature makes it considerably more expensive in terms of compilation time. In addition, the notations used by Wasilew are difficult to read and write.

Weingart's Design In comparison to Wasilew, Weingart's design is centered around a single tree of patterns – Weingart called this a *discrimination net* – which is automatically derived from a declarative machine description. Using a single tree of patterns, Weingart argued, allows for a compact and efficient means of representing the pattern set. The process of building the AST is then extended to simultaneously push each new AST node onto a stack. In tandem, the discrimination net is progressively traversed by comparing the nodes on the stack against the children of the current node in the net. A match is found when the process reaches a leaf in the discrimination net, whereupon the instruction associated with the match is emitted.

Like Wasilew's design, Weingart's had a more extensive instruction support compared to the contemporary techniques as it could include patterns extending over multiple AST nodes. However, when applied in practice, the design suffered from several problems. First, structuring the discrimination net to support efficient pattern matching proved difficult for certain target machines; it is known that Weingart struggled in particular with the PDP-11. Second, the design assumes that there exists at least one instruction on the target machine that corresponds to a particular node type of the AST, which turned out to not always be the case. Weingart partly addressed this problem by introducing *conversion patterns*, which could transform mismatched parts of the AST into another form that hopefully would be matched by some pattern at a later stage. However, these had to be added manually and could potentially cause the compiler to get stuck in an infinite loop.

```
ASG PLUS,   INAREG,
            SAREG,   TINT,
            SNAME,   TINT,
                     0,       RLEFT,
                     "        add       AL,AR\n",
...
ASG OPSIM,   INAREG|FORCC,
             SAREG,       TINT|TUNSIGNED|TPOINT,
             SAREG|SNAME|SOREG|SCON,   TINT|TUNSIGNED|TPOINT,
                          0,           RLEFT|RESCC
                          "            OI            AL,AR\n"
```

Figure B.3: A machine description sample for PCC, consisting of two patterns [204]. The first line specifies the node type of the root (+=, for the first pattern) together with its cookie ("result must appear in an A-type register"). The second and third lines specify the left and right descendants, respectively, of the root. The left subtree of the first pattern must be an `int` allocated in an A-type register, and the right subtree must be a `NAME` node, also of type `int`. The fourth line indicates that no registers or temporaries are required and that the matched part in the expression tree is to be replaced by the left descendant of the pattern's root. The fifth and last line declares the assembly string, where lowercase letters are output verbatim and uppercase words indicate a macro invocation – `AL` stands for "Address form of Left operand", and likewise for `AR` – whose result is then put into the assembly string. In the second pattern we see that multiple restrictions can be or'ed together, thus allowing multiple patterns to be expressed in a more concise manner.

Third, like its macro-expanding predecessors, the process immediately selects a pattern as soon as a match is found.

PCC Another early pattern matching technique was developed by Johnson [203], which was implemented in the *Portable C Compiler (PCC)* – a renowned system that was the first standard C compiler to be shipped with UNIX. Johnson based his design on the earlier work by Snyder [340] (which we discussed in Sect. A.2.3), but replaced the use of macro expansion with a method that performs tree rewriting. For each instruction, a expression tree is formed together with a rewrite rule, subgoals, resource requirements, and an assembly string which is emitted verbatim. This information is given in a machine description format that allows multiple, similar patterns to be condensed into a single declaration. An example is shown in Fig. B.3.

The pattern matching process is then relatively straightforward. For a given node in the expression tree, the node is compared against the root of each pattern. If these match, a similar check is done for each corresponding subtree in the pattern. Once all leaves in the pattern are reached, a match has been found. As this algorithm – whose pseudo-code is given in Alg. B.1 – exhibits quadratic time complexity, it is desirable to minimize the number of redundant checks. This is done by maintaining a set of code generation goals which are encoded into the instruction selector as an

```
    function FindMatchSet (expression tree T, pattern set P):
1       M ← array of size |T|, initialized to ∅
2       foreach node n_T ∈ T do
3           foreach pattern p ∈ P do
4               n_P ← root node of P
5               if Matches (n, n_P) then
6                   M[n_T] ← M[n_T] ∪ {p}
7       return M
8       function IsMatch (expression tree rooted at node n_T, pattern tree rooted at node n_P):
9           |n_T| ← number of children for n_T
10          |n_P| ← number of children for n_P
11          if n_T ≃ n_P and |n_T| = |n_P| then
12              foreach child n'_T, n'_P of n_T, n_P do
13                  if not IsMatch (n'_T, n'_P) then
14                      return false
15              return true
16          else
17              return false
```

Algorithm B.1: A straightforward algorithm for pattern-matching trees. The algorithm has $O(n^2p)$ time complexity, where n is the number of nodes in the expression tree and p is the number patterns in the pattern set. The relation $n_1 \simeq n_2$ holds for two nodes n_1 and n_2 if both are of the same type.

integer. For historical reasons this integer is called a *cookie*, and each pattern has a corresponding cookie indicating the situations in which the pattern may be useful. If both the cookies and the pattern match, an attempt is made to allocate whatever resources are demanded by the pattern (for example, a pattern may require a certain number of registers). If successful, the corresponding assembly string is emitted, and the matched subtree in the expression tree is replaced by a single node as specified by the rewrite rule. This process of matching and rewriting repeats until the expression tree consists of only a single node, meaning that the entire expression tree has been successfully converted into assembly code. If no pattern matches, the instruction selector enters a heuristic mode where the expression tree is partially rewritten until a match is found. For example, to match an `a = reg + b` pattern, an `a += b` expression could first be rewritten into `a = a + b` and then another rule could try to force operand `a` into a register.

Although successful for its time, PCC had several disadvantages. Like Weingart, Johnson used heuristic rewrite rules to handle mismatching situations. Without formal methods of verification there was always the risk that the current set of rules

would be inadequate and potentially cause the compiler to never terminate for certain functions. Reiser [319] also noted that the investigated version of PCC only supported unary and binary patterns with a maximum height of 1, thus excluding many instructions, such as those with complex addressing modes. Lastly, PCC – and all other techniques discussed so far – still adhered to the *first-matched-first-served* approach when selecting patterns.

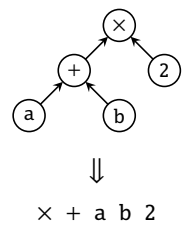
B.3 Using LR Parsing to Cover Trees Bottom-Up

As already noted, a common flaw among the first designs is that they apply the greediest form of pattern selection, and typically lack a formal methodology. In contrast, *syntactic analysis* – which is the task of parsing the source code – is arguably the best understood area of compilation, and its methods also produce completely table-driven parsers that are very fast and resource-efficient.

In 1978, Glanville and Graham [163] presented a seminal paper that describes how techniques of syntactic analysis can be adapted to instruction selection.[1] Due to its pioneers, we refer to this as the *Glanville-Graham approach*. We first describe grammars, which is the representation used by Glanville and Graham for modeling the instructions.

B.3.1 Modeling Instructions as Machine Grammars

To begin with, a well-known method of removing the need for parentheses in arithmetic expressions without making them ambiguous is to use *Polish notation*. For example, 1 + (2 + 3) can be written as + 1 + 2 3 and still represent the same expression. Glanville and Graham recognized that by using this form the instructions can be expressed as a context-free grammar. This concept is already well described in most compiler textbooks (see for example [9]), so we will proceed with only a brief introduction.



A *context-free grammar* (or simply *grammar*) consists of terminals, nonterminals, and rules. In this context, a *terminal* is a symbol representing an operation (e.g. +, <, load), and a *nonterminal* is a symbol representing an abstract result (e.g. *Reg*) produced by the instruction. To distinguish between the two, terminals are written entirely in lower case whereas nonterminals start with a capital letter and are set in italics. A *rule* describes the behavior of an instruction and consists of a production, a non-negative cost, and an action. *Productions* describe how to derive nonterminals, and are written as

$$\alpha \rightarrow \beta\gamma \ldots$$

where the left-hand side is a single nonterminal and the right-hand side is a sequence of terminals and nonterminals. Each instruction therefore gives rise to one or more

[1] This had also been vaguely hinted at ten years earlier in an article by Feldman and Gries [130].

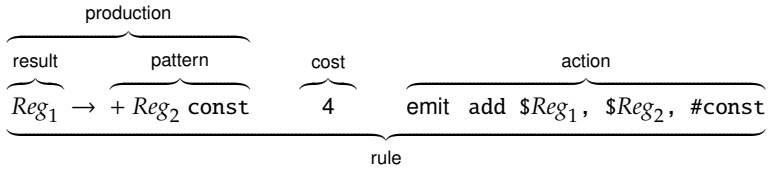


Figure B.4: Anatomy of a rule in a machine grammar.

#	production	cost	action
1	Reg_1 → load + Reg_2 const	1	emit load $$Reg_1$, const($$Reg_2$)
2	Reg_1 → load + const Reg_2	1	emit load $$Reg_1$, const($$Reg_2$)
3	Reg_1 → load Reg_2	1	emit load $$Reg_1$, 0($$Reg_2$)

Table B.1: Example of grammar rules corresponding to a `load $t, o($s)` instruction that loads a value from memory at the address given in register `s`, offset by an immediate value `o`, and stores the loaded value in register `t`, in one cycle. The subscripts are only needed for referencing the right nonterminal in the action.

productions, where the right-hand side of a production captures a pattern of the instruction and the left-hand side denotes the result produced by the instruction. Hence the left-hand and right-hand sides of a rule are referred to as the rule's *result* and *pattern*, respectively. The cost should be self-explanatory at this point, and the *action* is the activity to perform when the rule is selected (typically, this is to emit a string of assembly code). The rule structure is also illustrated in Fig. B.4, and an example of a few rules is given in Tab. B.1. The collection of rules for a particular target machine is called the *machine grammar* of that machine.

In most literature, rules and patterns usually have the same connotations. In this dissertation, however, in the context of grammars a rule refers to a tuple of production, cost, and action, and a pattern refers to the right-hand side of the production appearing in a rule.

Not shown in the example, a machine grammar also consists of a *goal symbol*, whose purpose will be explained shortly.

Normal Form To simplify pattern matching and pattern selection, a grammar can be rewritten into *normal form* [32]. A grammar is in normal form if every rule in the grammar has a production in one of the following forms:

1. $N \rightarrow$ `op` $A_1\ A_2 \ldots A_n$, where `op` is a terminal, representing an operation that takes n arguments, and all A_i are nonterminals. Such rules are called *base rules*.
2. $N \rightarrow$ `t`, where `t` is a terminal. Such rules are also called base rules.
3. $N \rightarrow A$, where A is a nonterminal. Such rules are called *chain rules*.

#	production	cost	action
1	$Reg \rightarrow$ `load` A	1	emit `load` $Reg, A.$`const(`$A.Reg$`)`
4	$A \rightarrow$ + Reg C	0	
2	$Reg \rightarrow$ `load` B	1	emit `load` $Reg, B.$`const(`$B.Reg$`)`
5	$B \rightarrow$ + C Reg	0	
6	$C \rightarrow$ `const`	0	
3	$Reg_1 \rightarrow$ `load` Reg_2	1	emit `load` Reg_1, `0(`Reg_2`)`

Table B.2: The grammar from Tab. 2.1 in normal form. Nonterminals A, B and C and rules 4–6 are introduced in order to transform rules 1 and 2 into base rules.

A grammar can be mechanically rewritten into normal form by introducing new nonterminals and breaking down illegal rules into multiple, smaller rules until the grammar is in normal form. For example, rewriting the grammar shown in Tab. 2.1 into normal form results in the grammar shown in Tab. B.2. Note that the new rules have zero cost and no action as these are only intermediary steps towards enabling reduction of the original rule.

Since all productions in a grammar have at most one terminal, the pattern matching problem becomes trivial (simply match the node type against the terminal in all base rules). Otherwise another bottom-up traversal of the expression tree would have to be made in order to find all matches, which can be done in linear time for most reasonable grammars [193]. As we will see, this also simplifies pattern selection as the patterns on the right-hand side in all productions have uniform height.

B.3.2 The Glanville-Graham Approach

The machine grammar provides us with a formal methodology for modeling instructions, but it does not address the problems of pattern matching and pattern selection. For that, Glanville and Graham applied an already-known technique called *LR parsing* [220]. Because this technique is mostly associated with syntactic analysis, the same application on trees is commonly referred to as *tree parsing*. An extremely deep and thorough account of the theory and practice of this approach is given by Henry [183].

Tree Parsing As an example, let us use the machine grammar given in Fig. B.5a to generate assembly code for the expression tree given in Fig. B.5b such that the result ends up in a register. We assume that `a` and `b` are variables already stored in registers and that `2` is an integer constant. Hence each node in the expression tree is either of type +, ×, `reg`, or `const`. These will be our terminals. Because the result should be end up in a register, we say that *Reg* is our goal symbol.

After transforming the expression tree into a sequence of terminals (as in Fig. B.5c), we traverse the sequence from left to right. When doing so, we either

#	production	cost	action
1	$Reg_1 \rightarrow + \; Reg_2 \; Reg_3$	1	emit `add` $\$Reg_1$`,` $\$Reg_2$`,` $\$Reg_3$
2	$Reg_1 \rightarrow \times \; Reg_2 \; Reg_3$	1	emit `mul` $\$Reg_1$`,` $\$Reg_2$`,` $\$Reg_3$
3	$Reg \rightarrow$ `const`	1	emit `mv` $\$Reg$`, #const`
4	$Reg \rightarrow$ `reg`	1	

(a) Machine grammar.

`+ a × b 2`

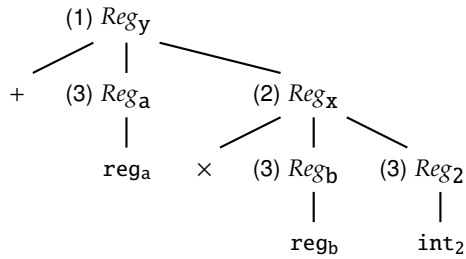
(b) Expression tree.

$+ \; reg_a \; \times \; reg_b \; const_2$

(c) Sequence of terminals.

s s r_4 s s r_4 s r_3 r_2 r_1

(d) Sequence of shifts and rule reductions.



(e) Parse tree. The rule numbers are shown in parentheses. The subscripts denote book keeping data.

```
mv  $Reg2, #2
mul $Regx, $Regb, $Reg2
add $Regy, $Rega, $Regx
```

(f) Assembly code.

Figure B.5: Example of tree parsing.

shift the just-traversed symbol onto a stack, or replace symbols currently on the stack via a *rule reduction*. We denote a shift by s and a rule reduction by r_x, where x is the number of the reduced rule. A rule reduction consists of two steps. First, the symbols are popped according to those that appear on the pattern of the rule. The number and order of symbols popped must match exactly for a valid rule reduction. Once popped, the nonterminal appearing on the left-hand side is pushed onto the stack. When the last rule reduction has been performed, the stack must contain only the goal symbol. For our example, a valid sequence of shifts and rule reductions is given in Fig. B.5d. The performed rule reductions can also be represented as a *parse tree*, illustrating the terminals and nonterminals which were used to parse the sequence of terminals. Figure B.5e shows the corresponding parse tree for the sequence shown in Fig. B.5d. Lastly, when performing a rule reduction we also execute the action associated with the rule. For our example, the resulting assembly code is shown in Fig. B.5f.

The problem that remains is how to know when to shift and when to reduce. This can be addressed by consulting a state table which has been generated for a specific grammar. How this table is produced is out of scope for this dissertation, but an example generated from the machine grammar shown in Fig. B.6 is given in Fig. B.7. A walk-through of executing an instruction selector with this state table using an LR parser is provided in Fig. B.8.

The subscripts that appear in some of the productions in Fig. B.6 are semantic qualifiers, which are used to express restrictions that may appear for some of the instructions. For example, all two-address arithmetic instructions store the result in one of the registers provided as input. Using semantic quantifiers, this could be expressed as $R_1 \rightarrow + R_1 R_2$, indicating that the destination register must be the same as that of the first operand. To make this information available during parsing, the parser pushes it onto the stacking along with its corresponding terminal or nonterminal symbol. Glanville and Graham also incorporated a register allocator into their parser, thus constituting an entire code generator.

Resolving Conflicts and Avoiding Blocking As instruction sets are rarely orthogonal, most machine grammars are *ambiguous*, meaning multiple valid parse trees may exist for the same expression tree. This causes the instruction selector to have the option of performing either a shift or a rule reduction, which is known as *shift-reduce conflict*. To solve this kind of conflict, Glanville and Graham's state table generator always decides to shift. The intuition is that this will favor larger patterns over smaller ones as a shift postpones a decision to pattern select while allowing more information about the expression tree to accumulate on the stack.[2]

Unfortunately, this scheme can cause the instruction selector to fail even though a valid parse tree exists. This is called *syntactic blocking*. To avoid such situations, the grammar designer must augment the machine grammar with auxiliary rules that patch the top of the stack when necessary. This allows the parser to recover from situations when it greedily decides to shift instead of applying a necessary rule reduction.

Likewise, there is also the possibility of *reduce-reduce conflicts*, where the parser has the option of choosing between two or more rules in a . Glanville and Graham resolved these by selecting the rule with the longest pattern. If the grammar contains rules that differ only in their semantic quantifiers, then there may still exist more than one rule to reduce (in Fig. B.6, rules 5 and 6 are two such rules). These are resolved at parse time by checking the semantic restrictions in the order in which they appear in the grammar (see for example state 20 in Fig. B.7).

If all rules in this set are semantically constrained, then situations can arise where the parser is unable to apply any rule due to semantic mismatch. This is called *semantic blocking* and can be resolved by always providing a default rule that can be invoked when all other semantically constrained rules fail. This

[2]The approach of always selecting the largest possible pattern is a scheme commonly known as *maximum munch*, which was coined by Cattell in his doctoral dissertation [69].

#	production	action
1	$R_2 \rightarrow$ + ld + c R_1 R_2	add R_2,c,R_1
2	$R_1 \rightarrow$ + R_1 ld + c R_2	add R_1,c,R_2
3	$R \rightarrow$ + ld c R	add R,c
4	$R \rightarrow$ + R ld c	add R,c
5	$R_1 \rightarrow$ + R_1 R_2	add R_1,R_2
6	$R_2 \rightarrow$ + R_1 R_2	add R_2,R_1
7	$\rightarrow$ = ld + c R_1 R_2	store R_2,*c,R_1
8	$\rightarrow$ = + c R_1 R_2	store R_2,c,R_1
9	$\rightarrow$ = ld c R	store R,*c
10	$\rightarrow$ = c R	store R,c
11	$\rightarrow$ = R_1 R_2	store R_2,R_1
12	$R_2 \rightarrow$ ld + c R_1	load R_2,c,R_1
13	$R_2 \rightarrow$ + c R_1	load R_2,=c,R_1
14	$R_2 \rightarrow$ + R_1 c	load R_2,=c,R_1
15	$R_2 \rightarrow$ ld R_1	load R_2,*R_1
16	$R \rightarrow$ ld c	load R,=c
17	$R \rightarrow$ c	mv R,c

Figure B.6: Example of a machine grammar [163]. c ("const"), ld ("load"), +, and = are all terminals, R is a nonterminal indicating that the result will be stored in a register, and subscripts denote the semantic qualifiers. All rules have the same unit cost. Rules 7–11 do not have a nonterminal on the left-hand side as memory store instructions do not produce anything. A dummy nonterminal can also be used if needed.

#	$	R	c	+	ld	=
0	accept					s1
1		s2	s3	s4	s5	
2		s6	s7	s8	s9	
3		s10	s7	s8	s9	
4		s11	s12	s8	s13	
5		s14	s15	s16	s9	
6	r11	r11	r11	r11	r11	r11
7	r17	r17	r17	r17	r17	r17
8		s11	s17	s8	s13	
9		s14	s18	s19	s9	
10	r10	r10	r10	r10	r10	r10
11		s20	s21	s8	s22	
12		s23	s7	s8	s9	
13		s14	s24	s25	s9	
14	r15	r15	r15	r15	r15	r15
15		s26	s7	s8	s9	
16		s11	s27	s8	s13	
17		s28	s7	s8	s9	
18	r16	r16	r16	r16	r16	r16
19		s11	s29	s8	s13	
20	r5/6	r5/6	r5/6	r5/6	r5/6	r5/6
21	r14	r14	r14	r14	r14	r14
22		s14	s30	s31	s9	
23		s32	s7	s8	s9	
24		s33	s7	s8	s9	
25		s11	s34	s8	s13	
26	r9	r9	r9	r9	r9	r9
27		s35	s7	s8	s9	
28	r13	r13	r13	r13	r13	r13
29		s36	s7	s8	s9	
30	r4	r4	r4	r4	r4	r4
31		s11	s37	s8	s13	
32	r8	r8	r8	r8	r8	r8
33	r3	r3	r3	r3	r3	r3
34		s38	s7	s8	s9	
35		s39	s7	s8	s9	
36	r12	r12	r12	r12	r12	r12
37		s40	s7	s8	s9	
38		s41	s7	s8	s9	
39	r7	r7	r7	r7	r7	r7
40	r2	r2	r2	r2	r2	r2
41	r1	r1	r1	r1	r1	r1

Figure B.7: State table generated from the machine grammar given in Fig. B.6 [163]. si indicates a shift to the next state i, rj indicates the reduction of rule j, and a blank entry indicates an error.

step	state stack	symbol	input	action
1	0		= + c_a R_7 + ld + c_b ld R_7 ld c_c $	shift to 1
2	0 1	=	+ c_a R_7 + ld + c_b ld R_7 ld c_c $	shift to 4
3	0 1 4	= +	c_a R_7 + ld + c_b ld R_7 ld c_c $	shift to 12
4	0 1 4 12	= + c_a	R_7 + ld + c_b ld R_7 ld c_c $	shift to 23
5	0 1 4 12 23	= + c_a R_7	+ ld + c_b ld R_7 ld c_c $	shift to 8
6	0 1 4 12 23 8	= + c_a R_7 +	ld + c_b ld R_7 ld c_c $	shift to 13
7	0 1 4 12 23 8 13	= + c_a R_7 + ld	+ c_b ld R_7 ld c_c $	shift to 25
8	0 1 4 12 23 8 13 25	= + c_a R_7 + ld +	c_b ld R_7 ld c_c $	shift to 34
9	0 1 4 12 23 8 13 25 34	= + c_a R_7 + ld + c_b	ld R_7 ld c_c $	shift to 9
10	0 1 4 12 23 8 13 25 34 9	= + c_a R_7 + ld + c_b ld	R_7 ld c_c $	shift to 14
11	0 1 4 12 23 8 13 25 34 9 14	= + c_a R_7 + ld + c_b ld R_7	ld c_c $	reduce rule 15 ($R_2 \rightarrow$ ld R_1) assign result to register 8 emit load R_8,*R_7 shift to 38
12	0 1 4 12 23 8 13 25 34 38	= + c_a R_7 + ld + c_b R_8	ld c_c $	shift to 9
13	0 1 4 12 23 8 13 25 34 38 9	= + c_a R_7 + ld + c_b R_8 ld	c_c $	shift to 18
14	0 1 4 12 23 8 13 25 34 38 9 18	= + c_a R_7 + ld + c_b R_8 ld c_c	$	reduce rule 16 ($R \rightarrow$ ld c) assign result to register 9 emit load r9,c shift to 41
15	0 1 4 12 23 8 13 25 34 38 41	= + c_a R_7 + ld + c_b R_8 R_9	$	reduce rule 1 ($R_2 \rightarrow$ + ld + c R_1 R_2) emit add R_9,b,R_8 shift to 32
16	0 1 4 12 23 32	= + c_a R_7 R_2	$	reduce rule 8 ($\rightarrow$ = + c R_1 R_2) emit store R_9,a,R_7
17	0		$	accept

Figure B.8: A walk-through of executing the table from Fig. B.7 on a sequence = + c_a R_7 + ld + c_b ld R_7 ld c_c (for simplicity, registers in the sequence are represented directly as nonterminals). Note that a rule reduction may involve two operations: the mandatory reduce operation, followed by an optional operation which may be a shift or another rule reduction. For example, let us examine step 11. First, a reduce is executed using rule 15, which pops ld R_7 from the symbol stack. This is followed by pushing the result of the rule, R_8, on top. At the same time, states 9 and 14 are popped from the stack, which leaves state 34 on top. The top elements of both stacks are now used to consult the state table for inferring the next, additional action (if any). In this case, input symbol R_8 at state 34 leads to a shift to state 38.

fallback rule typically uses multiple, shorter instructions to simulate the effect of the more complex rule, and Glanville and Graham devised a clever trick to infer them automatically. For every semantically constrained rule *r*, tree parsing is then performed over the tree representing the pattern of *r*. The instructions selected to implement this tree thus constitute the implementation of the fallback rule for *r*.

Advantages Subsequent experiments and evaluations showed that this design proved simpler and more general than contemporary designs [10, 153, 168, 169, 232]. Due to this, the Glanville-Graham approach has been acknowledged as one of the most significant breakthroughs in this field which has influenced many later techniques in one way or another. In addition, by relying on a state table a Glanville-Graham-style instruction selector is completely table-driven since it is implemented by a core that basically consists of a series of table lookups.[3] Hence the time it takes for the instruction selector to generate the assembly code is linearly proportional to the size of the expression tree. Although the idea of table-driven code generation was not novel in itself – we have seen several examples of it in Ap. A – earlier attempts had all failed to provide an automated procedure for producing the tables. In addition, many decisions regarding pattern selection are precomputed by resolving and reduce-reduce conflicts at the time that the state table is generated, thus reducing compilation time.

Another advantage of the Glanville-Graham approach is its formal foundation, which enables means of automatic verification. For instance, Emmelmann [115] presented one of the first methods of proving the completeness of an machine grammar.[4] The intuition behind Emmelmann's automatic prover is to find all expression trees that can appear in the function but cannot be handled by the instruction selector. Let us denote an machine grammar by G and a grammar describing the expression trees by T. If we further use $L(X)$ to represent the set of all trees accepted by a grammar X, we can then determine whether the machine grammar is incomplete by checking if $L(T) \setminus L(G)$ yields a nonempty set. Emmelmann recognized that this intersection can be computed by creating a product automaton which essentially implements the language that accepts only the trees in this set of counterexamples. From this automaton it is also possible to derive the rules that are missing from the machine grammar. Brandner [52] recently extended this method to handle productions that contain predicates – we will discuss these shortly when exploring attribute grammars – by splitting terminals to expose these otherwise-hidden characteristics.

[3] Pennello [299] developed a technique to express the state table directly as assembly code, thus eliminating even the need to perform table lookups. This was reported to improve the efficiency of LR parsing by six to ten times.

[4] Note, however, that even though an machine grammar has been proven to be complete, a greedy instruction selector may still fail to use a necessary rule. Consequently, Emmelmann's checker assumes that an optimal instruction selector will be used for the proven machine grammar.

Disadvantages Although it addressed several of the problems with contemporary instruction selection techniques, the Glanville-Graham approach also had disadvantages of its own. First, since an LR parser can only reason on syntax, any restrictions regarding specific values or ranges must be captured by its own nonterminal. In conjunction with the limitation that each production can match only a single pattern, this typically meant that rules for versatile instructions with several or operand modes had to be duplicated for each such mode. For most target machines this turned out to be impracticable. For example, in the case of the VAX machine – a *complex instruction-set computer (CISC)* architecture from the 1980s, where each instruction accepted a multitude of operand modes [71] – the machine grammar would contain over eight million rules [169]. By introducing auxiliary nonterminals to combine features shared among the rules – a task called *refactoring* – the number was brought down to about a thousand rules, but this had to be done carefully to not have a negative impact on code quality. Second, since the parser traverses from left to right without backtracking, assembly code regarding one operand has to be emitted before any other operand can be observed. This can potentially lead to poor decisions which later have to be undone by emitting additional code, as in the case of recovering from syntactic blocking. Hence, to design an machine grammar that was both compact and yielded good code quality, the developer had to possess extensive knowledge about the implementation of the instruction selector.

B.3.3 Extending Grammars with Semantic Handling

In purely context-free grammars there is just no way to handle semantic information. For example, the exact register represented by a *reg* nonterminal is not available. Glanville and Graham worked around this limitation by pushing the information onto the stack, but even then their modified LR parser could reason upon it using only simple equality comparisons. Ganapathi and Fischer [149, 150, 151, 152] addressed this problem by replacing the use of traditional, context-free grammars with the use of a more powerful set of grammars known as *attribute grammars*. There are also *affix grammars*, which can be thought of as a subset of attribute grammars. In this dissertation, however, we will only consider attribute grammars, and, as with the Glanville-Graham approach, we will discuss how they work only at a high level.

Attribute Grammars Attribute grammars were introduced in 1968 by Knuth [221], who extended context-free grammars with attributes. *Attributes* are used to store, manipulate, and propagate additional information about individual terminals and nonterminals during parsing, and an attribute is either *synthesized* or *inherited*. Using parse trees as the point of reference, a node with a synthesized attribute forms its value from the attributes of its children, and a node with an inherited attribute copies the value from the parent. Consequently, information derived from synthesized attributes flows *upwards* along the tree while information derived from inherited attributes flows *downwards*. We therefore distinguish between synthesized

#	production	predicates	actions
1	$Byte \uparrow r \rightarrow + \; Byte \uparrow a \; Byte \uparrow r$	$IsOne(\downarrow a)$, $NotBusy(\downarrow r)$	emit `incb` $\downarrow r$
2	$Byte \uparrow r \rightarrow + \; Byte \uparrow r \; Byte \uparrow a$	$IsOne(\downarrow a)$, $NotBusy(\downarrow r)$	emit `incb` $\downarrow r$
3	$Byte \uparrow r \rightarrow + \; Byte \uparrow a \; Byte \uparrow r$	$TwoOp(\downarrow a, \downarrow r)$	emit `addb2` $\downarrow a$, $\downarrow r$
4	$Byte \uparrow r \rightarrow + \; Byte \uparrow r \; Byte \uparrow a$	$TwoOp(\downarrow a, \downarrow r)$	emit `addb2` $\downarrow a$, $\downarrow r$
5	$Byte \uparrow r \rightarrow + \; Byte \uparrow a \; Byte \uparrow b$		get register $\uparrow r$ emit `addb3` $\downarrow r$, $\downarrow a$, $\downarrow b$

Table B.3: Example of an instruction set expressed as attribute grammar [151].

and inherited attributes by a $\uparrow$ or $\downarrow$, respectively, which will be prefixed to the attribute of the concerned symbol. For example, the synthesized attribute x of a *Reg* nonterminal is written as $Reg \uparrow x$.

The attributes are then used within predicates and actions. *Predicates* are used for checking the applicability of a rule, and, in addition to emitting assembly code, actions are used to produce new synthesized attributes. Hence, when modeling instructions we can use predicates to express the constraints, and actions to indicate effects, such as code emission, and which register the result will be stored in. Let us look at an example.

In Tab. B.3 we see a set of rules for modeling three byte-adding instructions: (i) an increment version `incb` (increments a register by 1, modeled by rules 1 and 2); (ii) a two-address version `add2b` (adds two registers and stores the result in one of the operands, modeled by rules 3 and 4); and (iii) a three-address version `add3b` (the result can be stored elsewhere, modeled by rule 5). Naturally, the `incb` instruction can only be used when one of the operands is a constant of value 1, which is checked by the *IsOne* predicate. In addition, since this instruction destroys the previous value of the register, it can only be used when no subsequent operation uses the old value (meaning the register is not "busy"), which is checked by the *NotBusy* predicate. The emit action then emits the corresponding assembly code. Since addition is commutative, we require two rules to make the instruction applicable in both cases. Similarly, we have two rules for the `add2b` instruction, but the predicates have been replaced by a *TwoOp*, which checks if one of the operands is the target of assignment or if the value is not needed afterwards. Since the last rule does not have any predicates, it also acts as the default rule, thus preventing situations of semantic blocking which we discussed when covering the Glanville-Graham approach.

Advantages The use of predicates removes the need of introducing new nonterminals for expressing specific values and ranges, resulting in a more concise machine grammar compared to a context-free grammar. For example, for the VAX machine, the use of attributes leads to a grammar half the size (around 600 rules) compared to that required for the Glanville-Graham approach, even without applying any extensive refactoring [150]. Attribute grammars also facilitate incremental development of the machine descriptions. One can start by implementing the most general rules

to achieve an machine grammar that produces correct but inefficient code. Rules for handling more complex instructions can then can be added incrementally, making it possible to balance implementation effort against code quality. Another useful feature is that other program optimization routines, such as constant folding, can be expressed as part of the grammar instead of as a separate component. Farrow [126] even made an attempt at deriving an entire Pascal compiler from an attribute grammar.

Disadvantages To permit attributes to be used together with LR parsing, the properties of the machine grammar must be restricted. First, only synthesized attributes may appear in nonterminals. This is because an LR parser constructs the parse tree bottom-up and left-to-right, starting from the leaves and working its way up towards the root. Hence an inherited value only becomes available after the subtree of its nonterminal has been constructed. Second, since predicates may render a rule as semantically invalid for rule reduction, all actions must appear last in the rules. Otherwise they may cause effects that must be undone after a predicate fails its check. Third, as with the Glanville-Graham approach, the parser has to take decisions regarding one subtree without any consideration of sibling subtrees that may appear to the right. This can result in assembly code that could have been improved if all subtrees had been available beforehand, and this is again a limitation due to the use of LR parsing. Ganapathi [148] later made an attempt to resolve this problem by implementing an instruction selector in Prolog – a logic-based programming language – but this incurred exponential worst-case time complexity of the instruction selector.

B.3.4 Maintaining Multiple Parse Trees for Better Code Quality

Since LR parsers make a single pass over the expression trees – and thus only produce one out of many possible parse trees – the quality of the produced assembly code is heavily dependent on the machine grammar to guide the parser in finding a "good" parse tree.

Christopher et al. [75] attempted to address this concern by using the concepts of the Glanville-Graham approach but extending the parser to produce *all* parse trees, and then select the one that yields the best assembly code. This was achieved by replacing the original LR parser with an implementation of Earley's algorithm [107], and although this scheme certainly improves code quality – at least in theory – it does so at the cost of enumerating all parse trees, which is often too expensive in practice.

In 2000, Madhavan et al. [261] extended the Glanville-Graham approach to achieve optimal selection of patterns while allegedly retaining the linear time complexity of LR parsing. By incorporating a new version of LR parsing [336], s that were previously executed directly as matching rules were found are now allowed to be postponed by an arbitrary number of steps. Hence the instruction selector essentially keeps track of multiple parse trees, allowing it to gather enough

information about the function before committing to a decision that could turn out to be suboptimal. In other words, as in the case of Christopher et al. the design by Madhavan et al. also covers all parse trees but it immediately discards those determined to result in less efficient assembly code Hence the scheme resembles the branch and bound search strategy (see Chap. 3 on p. 53).

To do this efficiently, the design also incorporates offline cost analysis, which we will explore later in Sect. B.5.3. More recently, Yang [375] proposed a similar technique involving the use of *parser cactuses*, where deviating parse trees are branched off a common trunk to reduce space requirements. In both designs, however, the underlying principle still prohibits the modeling of many typical target machine features such as multi-output instructions since their grammars only allow rules that produce a single result.

B.4 Using Recursion to Cover Trees Top-Down

The tree covering techniques we have examined so far – in particular those based on LR parsing – all operate *bottom-up*. The instruction selector begins to cover the leaves in the expression tree. Based on the decisions taken for the subtrees, it then progressively works upwards along the tree until it reaches the root, continually matching and selecting applicable patterns along the way. This is by no means the only method of covering, as it can also be done *top-down*. In such designs, the instruction selector covers the expression tree starting from the root, and then recursively works its way downwards. Consequently, the flow of semantic information, such as the particular register in which a result will be stored, is also different. A bottom-up instruction selector lets this information trickle upwards along the expression tree – either via auxiliary data structures or through tree rewriting – whereas a top-down implementation decides upon this beforehand and pushes this information downwards. The latter is therefore said to be *goal-driven*, as pattern selection is guided by a set of additional requirements which must be fulfilled by the selected pattern. Since this in turn will incur new requirements for the subtrees, most top-down techniques are implemented recursively. This also enables backtracking, which is a necessary feature, as selection of certain patterns can cause the lower parts of the expression tree to become uncoverable.

B.4.1 First Applications

Using Means-End Analysis to Guide Instruction Selection To the best of the author's knowledge, Newcomer [281] was the first to develop a scheme that uses top-down tree covering to address instruction selection. In his 1975 doctoral dissertation, Newcomer proposes a design that exhaustively finds all combinations of patterns that cover a given expression tree, and then selects the one with lowest cost. Cattell [68] also describes this in his survey paper, which is the main source for the discussion of Newcomer's design.

The instructions are modeled as *T-operators*, which are basically pattern trees with costs and attributes attached. The attributes describe various restrictions, such as which registers can be used for the operands. There is also a set of T-operators that the instruction selector uses to perform necessary transformations of the function – its need will become clear as the discussion continues. The scheme takes an AST as expected input and then covers it following the aforementioned top-down approach. The instruction selector first attempts to find all matching patterns for the root of the AST, and then proceeds to recursively cover the remaining subtrees for each match. Pattern matching is done using a straightforward technique that we know from before (see Alg. B.1 on Alg. B.1). For efficiency, all patterns are indexed according to the type of their root. The result of this procedure is thus a set of pattern sequences each of which covers the entire AST. Afterwards, each sequence is checked for whether the attributes of its patterns are equal to those of a *preferred attribute set (PAS)*, which corresponds to a goal. If not, the instruction selector will attempt to rewrite the subtree using the transformation T-operators until the attributes match. To guide this process, Newcomer applied a heuristic search strategy known as *means-end analysis*, which was introduced by Newell and Simon [282] in 1959. The intuition behind means-end analysis is to recursively minimize the quantitative difference between the current state (that is, what the subtree looks like *now*) and a goal state (what it *should* look like). How to calculate this quantitative difference, however, is not mentioned in [68]. To avoid infinite looping, the transformation process stops once it reaches a certain depth in the search space. If successful, the applied transformations are inserted into the pattern sequence; if not, the sequence is dropped. From the found pattern sequences the one with the lowest total cost is selected, followed by assembly code emission.

Newcomer's design was pioneering as its application of means-end analysis made it possible to guide the process of modifying the function, without having to resort to target-specific mechanisms, until it could be implemented on the target machine. But the design also had several significant flaws. First, it had little practical application, as Newcomer's implementation only handled arithmetic expressions. Second, the T-operators used for modeling the instructions as well as transformations had to be constructed by hand – a task that was far from trivial – which hindered compiler retargetability. Third, the process of transforming the function could end prematurely due to the search space cut-off, causing the instruction selector to fail to generate any assembly code whatsoever. Lastly, the search strategy proved much too expensive to be usable in practice except for very small expression trees.

Making Means-End Analysis Work in Practice Cattell et al. [67, 70, 250] later improved and extended Newcomer's work into a more practical framework which was implemented in the *Production Quality Compiler-Compiler (PQCC)*, a derivation of the *Bliss-11* compiler originally written by Wulf et al. [372]. Instead of performing the means-end analysis as the function is compiled, their design does it as a

preprocessing step when generating the compiler itself – much as with the Glanville-Graham approach.

The patterns are expressed as a set of templates which are formed using recursive composition, and are thus similar to the productions found in machine grammars. But unlike Glanville and Graham's and Ganapathi and Fischer's designs – where the grammars were written by hand – the templates in PQCC are derived automatically from a machine description. Each instruction is modeled as a set of *machine operations* that describe the effects of the instruction. The machine operations are thus akin to the RTLs introduced by Fraser [139] in Ap. A on p. 146. These effects are then used by a separate tool, called the *Code-Generator Generator (CGG)*, to create the templates which will be used by the instruction selector.

In addition to producing the trivial templates corresponding directly to an instruction, CGG also produces a set of single-node patterns as well as a set of larger patterns that combine several instructions. The former ensures that the instruction selector is capable of generating assembly code for all functions (since any expression tree can thereby be trivially covered), while the latter reduces compilation time as it is quicker to match a large pattern than many smaller ones. To do this, CGG uses a combination of means-end analysis and heuristic rules which apply a set of axioms (such as $\neg\neg E \Leftrightarrow E$, $E + 0 \Leftrightarrow E$, and $\neg(E_1 \geq E_2) \Leftrightarrow E_1 < E_2$) to manipulate and combine existing patterns into new ones. However, there are no guarantees that these "interesting" patterns will ever be applicable in practice. Once generated, instruction selection is performed in a greedy, top-down fashion that always selects the lowest-cost template matching the current node in the expression tree. Pattern matching is done using a scheme identical to that of Newcomer. If there is a tie, the instruction selector picks the template with the least number of memory loads and stores.

Compared to the LR parsing-based methods discussed previously, the design by Cattell et al. has both advantages and disadvantages. The main advantage is that the instruction selectors is less at risk of failing to generate assembly code for some function. There is the possibility that the set of predefined templates is insufficient to produce all necessary single-node patterns. In such cases, however, CGG can at least issue a warning (in Ganapathi and Fischer's design this correctness has to be ensured by the grammar designer). The disadvantage is that it is relatively slow. Whereas the tree parsing-based instruction selectors exhibit linear time complexity – both for pattern matching and selection – the instruction selector by Cattell et al. has to match each template individually, which could take quadratic time in the worst case.

Recent Designs To the best of the author's knowledge, the only recent technique (less than 20 years old) to use this kind of recursive top-down methodology for tree covering is that of Nymeyer et al. [289, 290]. In two papers from 1996 and 1997, Nymeyer et al. introduce a method where A* search – another strategy for exploring the search space (see [323]) – is combined with BURS theory. We will discuss BURS

theory in more detail later in this appendix (the eager reader can skip directly to Sect. B.5.3). Until then, let it for now be sufficient to say that grammars based on BURS allow transformation rules, such as rewriting $X + Y$ into $Y + X$, to be included as part of the machine grammar. This potentially simplifies and reduces the number of rules required for expressing the instructions, but unfortunately the authors did not publish any experimental results. Hence it is difficult to judge whether the A*-BURS theory combination would be an applicable technique in practice.

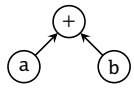
B.4.2 A Note on Tree Rewriting vs. Tree Covering

At this point some readers may feel that tree rewriting – where patterns are iteratively selected for rewriting the expression tree until it consists of a single node of some goal type – is something entirely different compared to tree covering – where compatible patterns are selected for covering all nodes in the expression tree. The same argument applies to and graph covering, although rewriting-based techniques are less common for those principles. Indeed, there appears to be a subtle difference, but a valid solution to a problem expressed using tree rewriting is also a valid solution to the equivalent problem expressed using tree covering, and vice versa. It could therefore be argued that the two are interchangeable, but we regard tree rewriting as a *means* to solving the tree covering problem, which we regard as the fundamental *principle*.

Handling Chain Rules in Purely Coverage-Driven Designs Another objection that may arise is how tree covering, as a principle, can support chain rules. A chain rule is a rule whose pattern consists of a single nonterminal, and the name comes from the fact that s using these rules can be chained together one after another. Consequently, chain rules are often used to represent data transfers and other value-preserving transformations (an example of this is given in Ap. D).

Let us first assume that we have as input the expression tree shown in Fig. B.9a, which will be covered using the machine grammar shown in Fig. B.9b. Let us further assume that we have an instruction selector where pattern matching is performed strictly through node comparison. This instruction selector is clearly based on tree covering, but it will fail to find a valid cover for the aforementioned expression tree as it will not be able to match and select the necessary chain rules (see Fig. B.9c).

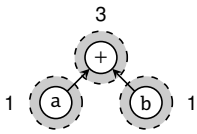
There are three ways of solving this problem. The simplest method is to simply ignore the incompatibilities during pattern selection, and then inject the assembly code for the necessary chain rules afterwards. But this obviously compromises code quality as the cost of the chain rules is not taken into account. A better approach is to consider all chain rule applications during pattern matching, thus essentially combining regular patterns with chain rules to yield new patterns (see Fig. B.9d). Finding all such combinations is known as computing the *transitive closure*. The third and last approach is to augment the expression tree by inserting auxiliary nodes, each of which each represents the application of a chain rule (see Fig. B.9e).



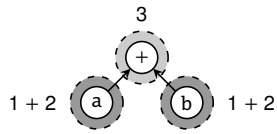
(a) Expression tree to cover. Variables a and b are assumed to be stored in A-type registers.

#	production
1	*RegA* → `reg`
2	*RegB* → *RegA*
3	*RegB* → + *RegB RegB*

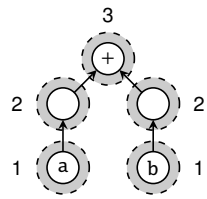
(b) Machine grammar.



(c) Invalid cover due to incompatibilities between the selected patterns.



(d) Covering using the transitive closure method.



(e) Covering using the augmentation method.

Figure B.9: Examples illustrating how chain rules can be supported by tree covering-based techniques. The numbers represent rule numbers.

The transitive closure and augmentation methods both come with certain benefits and drawbacks. The former method allows chain rules to be applied in any combination and of any length, but it complicates the tasks of pattern matching and pattern selection. The latter method requires no change in the pattern matcher and pattern selector. However, it enlarges the expression tree and requires an additional dummy rule to indicate that no chain rule is applied. If more than one chain rule needs to be applied, then several auxiliary nodes must be inserted one after another. As we have seen, several designs ignore this problem by assuming a homogeneous target architecture, and a few techniques apply the inefficient idea of code injection. The transitive closure approach is typically limited to tree covering-based methods, while the augmentation method is mostly applied when covering more general forms such as directed acyclic graphs and graphs (which we will discuss in the coming appendices).

B.5 Separating Pattern Matching from Pattern Selection

In the previously discussed techniques based on tree covering, the tasks of pattern matching and pattern selection are unified into a single step. Although this enables single-pass code generation, it typically also prevents the instruction selector from considering the impact of certain combinations of patterns. By separating these two concerns and allowing the instruction selector to make multiple passes over

	0	1	2	3	4	5	6	7	8
input string	a	b	c	a	b	c	a	b	d
pattern string	a	b	c	a	b	d			
						↑			
				a	b	c	a	b	d
							↑		

Table B.4: Example of string matching without full backtracking.

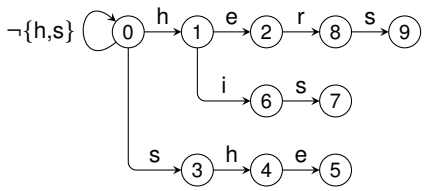
the expression tree, it can gather enough information about all applicable patterns before having to commit to premature decisions.

But the pattern matchers we have seen so far – excluding those based on LR parsing – have all been implementations of algorithms with quadratic time complexity. Fortunately, we can do better.

B.5.1 Algorithms for Linear-Time, Tree-Based Pattern Matching

Over the years many algorithms have been discovered for finding all matches given a subject tree and a set of pattern trees (see for example [73, 83, 106, 193, 209, 308, 311, 335, 363, 373]). For tree covering, most pattern matching algorithms have been derived from methods of string-based pattern matching. This was first discovered by Karp et al. [209] in 1972, and their ideas were later extended by Hoffmann and O'Donnell [193] to form the algorithms most applied by tree-based instruction selection techniques. Hence, in order to understand pattern matching with trees, let us first explore how this is done with strings.

Matching Trees Is Equivalent to Matching Strings The algorithms most commonly used for string matching were introduced by Aho and Corasick [6] and Knuth et al. [222] (also known as the *Knuth-Morris-Pratt algorithm*) in 1975 and 1977, respectively. Independently discovered from one another, both algorithms operate in the same fashion and are thus nearly identical in their approach. The intuition is that when a partial match of a pattern with a repetitive substring fails, the pattern matcher does not need to return all the way to the input character where the matching initially started. This is illustrated in Tab. B.4, where the pattern string `abcabd` is matched against the input string `abcabcabd`. The arrow indicates the current character under consideration. At first, the pattern matches the beginning of the input string up until the last character (position 5). When this fails, instead of returning to position 1 and restarting the matching from scratch, the matcher remembers that the first three characters of the pattern (`abc`) have already been matched at this point. Therefore, it continues to position 6 and attempts to match the fourth character in the pattern. Hence all occurrences of the pattern can be found in linear time. We continue our discussion with Aho and Corasick's design as it is capable of matching multiple patterns whereas the algorithm of Knuth



(a) Goto function, represented as a state machine.

i	$output(i)$
2	{he}
5	{she, he}
7	{his}
9	{hers}

(b) Output function.

i	0	1	2	3	4	5	6	7	8	9
$failure(i)$	0	0	0	0	1	2	0	3	0	3

(c) Failure function.

Figure B.10: Example of a string-matching machine [6].

et al. only considers a single pattern (although it can easily be extended to handle multiple patterns as well).

Aho and Corasick's algorithm relies on three functions – *goto*, *failure*, and *output* – where the first function is implemented as a state machine and the two latter ones are implemented as simple table lookups. How these are constructed is out of scope for our purpose – the interested reader can consult the referenced paper – and we will instead illustrate how the algorithm works on an example. In Fig. B.10 we see the corresponding functions for matching the strings `he`, `she`, `his`, and `hers`. As a character is read from an input string, say `shis`, it is first given as argument to the *goto* function. Having initialized to state machine to state 0, *goto*(`s`) first causes a transition to state 3, and *goto*(`h`) causes a subsequent transition to state 4. For each successful transition to some state i we invoke $output(i)$ to check whether some pattern string has been matched, but so far no match has been found. For the next input character `i`, however, there exists no corresponding edge from the current state (that is, *goto*(`i`) causes a failure). At this point *failure*(4) is invoked, which dictates that the state machine should fall back to state 1. We then retry *goto*(`i`), which takes us to state 6. With the last input character, *goto*(`s`) causes a transition to state 7, where *output*(7) indicates a match with the pattern string `his`.

The Hoffmann-O'Donnell Algorithm Hoffmann and O'Donnell [193] developed two algorithms incorporating the ideas of Aho and Corasick and Knuth et al. In a paper from 1982, Hoffmann and O'Donnell first present an $O(np)$ algorithm that matches pattern trees in a top-down fashion. In the same paper, Hoffmann and O'Donnell then present an $O(n + m)$ bottom-up algorithm that trades linear-time pattern matching for longer preprocessing times (n is the size of the expression tree, p is the number of patterns, and m is the number of matches found).

The bottom-up algorithm is outlined in Alg. B.2. Starting at the leaves, each

```
function LabelTree (expression tree E, set T of lookup tables):
    n_E ← root node of E
    LabelNode (n_E)
    function LabelNode (expression tree rooted at n):
        foreach child m_i of n do
            LabelNode (m_i)
        t ← node type of n
        label n with T_t[labels of m_1, ..., m_k]
```

Algorithm B.2: Hoffmann-O'Donnell algorithm for labeling expression trees [193].

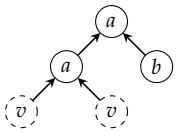
node is labeled with an identifier denoting the set of patterns that match the subtree rooted at that node. We call this set the *match set*. The label to assign a particular node is retrieved by using the labels of the children as indices in a table that is specific to the type of the current node. For example, label lookups for nodes representing addition are done using one table, while lookups for nodes representing subtraction are done using another table. The dimension of the table is equal to the number of children that the node may have. For example, binary operation nodes have two-dimensional tables while nodes representing constant values have 0-dimensional tables, which simply consist of a single value. A fully labeled example is shown in Fig. B.11g, and the match sets are then retrieved via a subsequent top-down traversal of the labeled tree.

Since the bottom-up algorithm introduced by Hoffmann and O'Donnell has had a historical impact on instruction selection, we will spend some time discussing the details of how the lookup tables are produced.

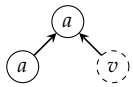
Definitions We begin by introducing a few definitions. To our aid, we will use two pattern trees I and II, shown in Figs. B.11a and B.11b, respectively. The patterns in our pattern set thus consist of nodes with symbols a, b, c, or v, where an a-node always has exactly two children, and b, c, and v-nodes always have no children. The v-symbol is a special *nullary symbol*, as such nodes represent placeholders that can match any subtree. We say that these symbols collectively constitute the *alphabet* Σ of our pattern set. The alphabet needs to be finite and *ranked*, meaning that each symbol in Σ has a ranking function that gives the number of children for a given symbol. Hence, in our case $rank(a) = 2$ and $rank(b) = rank(c) = rank(v) = 0$.

Following the terminology used in Hoffmann and O'Donnell's paper, we also introduce the notion of a Σ-*term* and define it as follows:

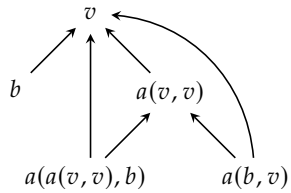
1. Each $i \in \Sigma$ with $rank(i) = 0$ is a Σ-term.
2. If $i \in \Sigma$ and $rank(i) > 0$, then $i(t_1, \ldots, t_{rank(i)})$ is a Σ-term provided every t_i is a Σ-term.
3. Nothing else is a Σ-term.



(a) Pattern I.



(b) Pattern II.



(c) Subsumption graph, excluding loop edges.

T_a	0	1	2	3	4
0	2	2	2	2	2
1	3	3	3	3	3
2	2	4	2	2	2
3	2	4	2	2	2
4	2	4	2	2	2

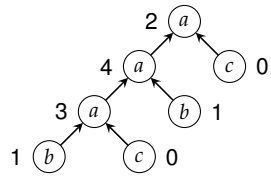
(d) Lookup table for symbol a.

T_a	1

(e) Lookup table for symbol b.

T_v	0

(f) Lookup table for symbol v.



(g) Labeled expression tree.

Figure B.11: Example of tree pattern matching using Hoffmann-O'Donnell [193]. Nullary nodes v are indicated with a dashed border. The subpatterns v, b, $a(v, v)$, $a(b, v)$, and $a(a(v, v), b)$ are labeled 0, 1, 2, 3, and 4, respectively.

A pattern tree is therefore a Σ-term, allowing us to write patterns I and II as $a(a(v, v), b)$ and $a(b, v)$, respectively. Σ-terms are also *ordered*, meaning $a(b, v)$ for example is different from $a(v, b)$. Consequently, commutative operations, such as addition, must be handled through pattern duplication (as in the Glanville-Graham approach).

We continue with some definitions concerning patterns. First, let us denote by $mtrees(p)$ the set of trees that can be matched by the pattern p at the root of any valid tree.[5] Depending on the alphabet, this set could be infinite. Then, a pattern p is said to *subsume* another pattern q (written $p \geq q$) if and only if any match set including p always also includes q (hence $mtrees(q) \subseteq mtrees(p)$). For example, given two patterns $a(b, b)$ and $a(v, v)$, we have that $a(b, b) \geq a(v, v)$, since the v-nodes must obviously also match whenever the b-nodes match. By this definition every pattern also subsumes itself. Furthermore, p *strictly subsumes* q (written $p > q$) if and only if $p \geq q$ and $p \neq q$, and p *immediately subsumes* q (written $p >_i q$) iff $p > q$ and there exists no other pattern r such that $p > r$ and $r > q$.

We also say that two patterns p and q are *inconsistent* if and only if both patterns never appear in the same match set (hence $mtrees(q) \cap mtrees(p) = \emptyset$). Lastly, p and

[5] This definition is not used by Hoffmann and O'Donnell in their paper, but having it will simplify the discussion to come.

q are *independent* iff there exist three distinct trees t, t', and t'' (that is, $t \neq t' \neq t''$), such that t is matched by p but not q (hence $mtrees(p) \nsubseteq mtrees(q)$), t' is matched by q but not p (hence $mtrees(q) \nsubseteq mtrees(p)$), and t'' is matched by both p and q (hence $mtrees(q) \cap mtrees(p) \neq \emptyset$).

Pattern sets that contain no independent patterns are known as *simple pattern sets*.[6] For example, the pattern set consisting of patterns I and II is simple as there exists no tree for which both match. As we will see, simple pattern sets have two important properties that we will use for generating the lookup tables.

Generating Lookup Tables for Simple Pattern Sets In general, the size of each lookup table is exponential to the size of the pattern set, as is the time to generate these tables. But Hoffmann and O'Donnell recognized that, for simple pattern sets, the number of possible match sets is equal to the number of patterns, making it tractable to generate the tables for such sets.

Furthermore, Hoffmann and O'Donnell found that each possible match set for a simple pattern set can be represented using a single pattern tree. The intuition is as follows. If a pattern p strictly subsumes another pattern q, then by definition it means that q will appear in every match set where p appears. Consequently, q does not need to be explicitly encoded into the match set since it can be inferred from the presence of p. Therefore, for every match set M we can select a subset of patterns in M to encode the entire match set. Let us call this subset the *base* of M, which we will denote by M_0. It can be proven that different match sets must have different bases, and that all patterns in M_0 must be pair-wise independent. However, in simple pattern sets we have no such patterns, and therefore the base of every match set must consist of a single pattern. We will call this pattern the *base pattern* of a match set, and it is the labels of the base patterns that will appear as entries in the lookup tables.

The key insight behind labeling is that in order to find the match set for some expression tree $T = a(T_1, T_2)$, it is sufficient to only consider the match sets for T_1 and T_2 in the context of a instead of T in its entirety. If the pattern set is simple, then we know that every match set has a base pattern. Let p_1 and p_2 denote the base patterns of the match sets of T_1 and T_2, respectively. With these we can transform T into $T' = a(p_1, p_2)$, and finding the match set for T' will then be equivalent to finding the match set for T. Since every entry in a lookup table refers to a match set (which is represented by its base pattern), and each symbol in Σ has its own table, we can produce the tables simply by finding the base patterns of the match set for the tree represented by every table entry. For example, if labels 1 and 2 respectively refer to the patterns b and $a(v, v)$, then the table entry $T_c[2, 1]$ will denote the tree $c(a(v, v), b)$, and we are then interested in finding the match set for that tree.

The next problem is thus to find the base pattern of a given match set. For simple pattern sets it can be proven that if we have three distinct patterns p, p', and p'', and p subsumes both p' and p'', then it must hold that either $p' > p''$ or $p'' > p'$.

[6] In Hoffmann and O'Donnell's paper these are called *simple pattern forests*.

```
function BuildSubsumptionGraph (set S of subpatterns):
1   G_S ← empty graph
2   foreach subpattern s ∈ S do
3       add node s to G_S
4       add edge s → s to G_S
5   foreach subpattern s ← a(s_1, ..., s_m) ∈ S in increasing height order do
6       foreach subpattern s' ∈ S s.t. height of s' ≤ height of s do
7           if s' = v or s' = a(s'_1, ..., s'_m) s.t. ∀1 ≤ i ≤ m : edge s_i → s'_i ∈ G_S then
8               add edge s → s' to G_S
```

Algorithm B.3: Algorithm for building the subsumption graph [193].

Consequently, for every match set M we can form a subsumption order among the patterns appearing in M. In other words, if a match set M contains m patterns, then we can arrange these patterns such that $p_1 > p_2 > \ldots > p_m$. The pattern appearing first in this order (in this case, p_1) is the base pattern of M as it strictly subsumes all other patterns in M. Hence, if we know the subsumption order, then we can easily find the base pattern.

For this purpose we first enumerate all unique subtrees, called the *subpatterns*, that appear in the pattern set. In the case of patterns I and II, this includes v, b, $a(v, v)$, $a(b, v)$, and $a(a(v, v), b)$, and we denote the set of all subpatterns as S. We then assign each subpattern in S a sequential number, starting from 0, which will represent the labels (the order in which these are assigned is not important).

Building the Subsumption Graph Next we form the *subsumption graph* for S, denoted by $\overline{G}_S$, where each node n_i represents a subpattern $s_i \in S$ and each edge $n_i \to n_j$ indicates that s_i subsumes s_j. For our pattern set, we get the subsumption graph illustrated in Fig. B.11c,[7] which we produce using the algorithm given in Alg. B.3. The algorithm basically works as follows. First, we add a node for every subpattern in S, together with a loop edge, as every pattern always subsumes itself. Next we iterate over all pair-wise combinations of subpatterns and check whether one subsumes the other, and add a corresponding edge to $\overline{G}_S$ if this is the case. To test whether a subpattern q subsumes another pattern q, we check whether the roots of p and q are of the same symbol and whether every subtree of p subsumes the corresponding subtree of q. This last check can be done by checking whether a corresponding edge exists in $\overline{G}_S$ for each combination of subtrees. Hence we should iterate this process until $\overline{G}_S$ reaches a fixpoint, but we can minimize the

[7] For every subsumption graph $\overline{G}_S$, there is also a corresponding *immediate subsumption graph* G_S. In general, G_S is shaped like a directed acyclic graph, but for simple pattern sets it is always a tree.

```
function GenerateTable (set S of subpatterns, subsumption graph G_S symbol a ∈ Σ):
1   T_a ← matrix of size |Σ| × |Σ|, initialized to v ∈ S
2   foreach subpattern s ← a(s_1, ..., s_m) ∈ S in increasing subsumption order do
3       foreach m-tuple ⟨s'_1, ..., s'_m⟩ s.t. ∀1 ≤ i ≤ m : s'_i ≥ s_i do
4           T_a[s'_1, ..., s'_m] ← s
```

Algorithm B.4: Algorithm for generating the lookup tables [193].

number of checks by first ordering the subpatterns in S by increasing height order and then comparing the subpatterns in that order.

Building the Lookup Tables Once we have $\overline{G}_S$, we can generate the lookup tables following the algorithm given in Alg. B.4. First, we find the subsumption order for all patterns by doing a topological sort of the nodes in $\overline{G}_S$ (see Ap. F for a definition of topological sort). Next, we initialize each entry in the table with the label of the subpattern consisting of a single nullary symbol, and then incrementally update an entry with the label of the next, larger pattern that matches the tree corresponding to that entry. By iterating over the patterns in increasing subsumption order, the last assignment to each entry will be that of the largest matching pattern in the pattern set. For our example, this results in the tables shown in Figs. B.11d, B.11e, and B.11f.

As already stated, since patterns are required to be ordered, we need to duplicate patterns containing commutative operations by swapping the subtrees of the operands. But doing this yields patterns that are pair-wise independent, destroying the property of the pattern set being simple. In such cases, the algorithm is still able to produce usable lookup tables, but the resulting match sets will include only one of the commutative patterns and not the other (which one depends on the subpattern last used during table generation). Consequently, not all matches will be found during pattern matching, which may in turn prevent optimal pattern selection.

Compressing the Lookup Tables Chase [72] further advanced Hoffmann and O'Donnell's table generation technique by developing an algorithm that compresses the final lookup tables. The key insight is that the lookup tables often contain redundant information as many rows and columns are duplicates. For example, this can be seen clearly in T_a from our previous example, which is also shown in Tab. B.5a. By introducing a set of *index maps*, the duplicates can be removed by mapping identical columns or rows in the index map to the same row or column in the lookup table. The lookup table can then be reduced to contain only the minimal amount of information, as seen in Tab. B.5b. By denoting the compressed version of T_a by τ_a, and the corresponding index maps by $\mu_{a,0}$ and $\mu_{a,1}$, we replace a previous lookup $T_i[l_0, \ldots, l_m]$ for symbol i with $\tau_i[\mu_{i,0}[l_0], \ldots, \mu_{i,m}[l_m]]$.

T_a	0	1	2	3	4
0	2	2	2	2	2
1	3	3	3	3	3
2	2	4	2	2	2
3	2	4	2	2	2
4	2	4	2	2	2

(a) Uncompressed lookup table.

$\mu_{a,1}$	0	1	2	3	4
	0	1	0	0	0

$\mu_{a,0}$	
0	0
1	1
2	2
3	2
4	2

τ_a	0	1
0	2	2
1	3	3
2	2	4

(b) Compressed lookup table.

Table B.5: Example of compressing the lookup table T_a [72].

Table compression also provides another benefit in that, for some pattern sets, the lookup tables can be so large that they cannot even be constructed in the first place. But Chase discovered that the tables can be compressed as they are generated, thus pushing the limit on how large lookup tables can be produced. Cai et al. [61] later improved the asymptotic bounds of Chase's algorithm.

B.5.2 Optimal Pattern Selection with Dynamic Programming

Once it became possible to find all match sets for the entire expression tree in linear time, techniques started to appear that also tackled the problem of optimal pattern selection in linear time. According to the literature, Ripken [320] was the first to propose a viable method for optimal linear-time instruction selection, which is described in a 1977 technical report. Ripken based his method on the dynamic programming algorithm by Aho and Johnson and later extended it to handle more realistic instruction sets with multiple register classes and addressing modes. For brevity, we will henceforth abbreviate *dynamic programming* as DP.

Although Ripken appears to have been the first to propose a design of an optimal DP-based instruction selector, it only remained that – a proposal. The first *practical* attempt was instead made in 1986 by Aho et al. [7, 8, 352] with the introduction of a compiler generator called *Twig*.

Twig As in Ripken's design, Twig uses a version of Aho and Johnson's DP algorithm for selecting the optimal set of pattern trees to cover a given expression tree. The machine description is expressed as a machine grammar (see Sect. B.3.2) using a language called *Code Generator Language (CGL)*, which as introduced by Aho and Ganapathi [7] in 1985. An excerpt of such a machine description is shown in Fig. B.12. Twig takes this machine description and generates an instruction selector that makes three passes over the expression tree. The first pass is a top-down

```
node const mem assign plus ind;
label reg no_value;
reg:const                                          /* Rule 1 */
  {  cost = 2; }
  ={ NODEPTR regnode = getreg( );
     emit(''MOV'', $1$, regnode, 0);
     return(regnode);
  };
no_value: assign(mem, reg)                         /* Rule 3 */
  {  cost = 2+$%1$->cost; }
  ={ emit(''MOV'', $2$, $1$, 0);
     return(NULL);
  };
reg: plus(reg, ind(plus(const, reg)))              /* Rule 6 */
  {  cost = 2+$%1$->cost+$%2$->cost; }
  ={ emit(''ADD'', $2$, $1$, 0);
     return($1$);
  };
```

Figure B.12: Examples of grammar rules for TWIG, written in CGL [8].

labeling pass that finds all match sets for every node in the expression tree[8] using an implementation of the Aho-Corasick string matching algorithm [6]. The second pass is a bottom-up cost computation pass that gives the cost of selecting a particular pattern for a given node. As we will see, the costs are computed using DP and hence the computation constitutes the core of this design. The last pass is a recursive top-down pass that finds the least-cost cover of the expression tree. This pass also executes the actions associated with the selected patterns, which in turn emits the corresponding assembly code.

The design is centered around the following assumption. Given a node n in an expression tree and a rule r, the cost of applying r on n is the cost of r plus the costs of reducing all children of n to the appropriate nonterminals appearing on the right-hand side of r. If r is a chain rule then the cost is computed as the cost of r plus the cost of reducing n to the result of r. The recursive nature of these costs can be exploited using dynamic programming, resulting in the algorithm shown in Alg. B.5 which computes the least cost of reducing a given expression tree to a particular nonterminal.

The algorithm works as follows. It first constructs a cost matrix C, where rows represent nodes in the expression tree and columns represent nonterminals in the grammar, which is assumed to be in normal form.[9] The cost in each element $C[i][j]$ is initialized to infinity, indicating that there exists no sequence of rule reductions which reduces node i to nonterminal j. It then computes the costs by traversing the expression tree bottom up. At each node n and for each matching base rule r,

[8] Remember that, when using machine grammars, a pattern found in the match set during pattern matching corresponds to the right-hand side of a production.

[9] The algorithm can be adapted to accept any grammar by expanding the FindMatchingRules and ComputeReductionCost functions to handle rules of arbitrary form.

```
function ComputeCosts(expression tree T, normal-form grammar G):
    S ← {s | s is a nonterminal in G}
    C ← matrix of size |T| × |S|, costs initialized to ∞
    ComputeCostsRec(root node of T)
    return C

function ComputeCostsRec(node n):
    foreach child m of n do
        ComputeCostsRec(m)
    foreach base rule r ∈ FindMatchingRules(n) do
        c ← ComputeReductionCost(n, r)
        l ← result of r
        if c < C[n][l].cost then
            C[n][l].cost ← c
            C[n][l].rule ← r
    repeat
        foreach chain rule r ∈ G do
            c ← ComputeReductionCost(n, r)
            l ← result of r
            if c < C[n][l].cost then
                C[n][l].cost ← c
                C[n][l].rule ← r
    until no change to C

function FindMatchingRules(node n):
    M ← ∅
    foreach base rule r ∈ G do
        if terminal in pattern of r = node type of n then
            M ← M ∪ {r}
    return M

function ComputeReductionCost(node n, rule r):
    c ← cost of r
    if r is a chain rule then
        s ← nonterminal in pattern of r
        c ← c + C[n][s].cost      ▷ here cost of node itself is taken instead of its children
    else
        for i ← 1 to number of children for n do
            m ← ith child of n
            s ← ith nonterminal in pattern of r
            c ← c + C[m][s].cost
    return c
```

Algorithm B.5: Computes the optimal sequence of rules that reduces the given expression tree to a particular nonterminal.

```
function SelectRules (expression tree T, goal nonterminal g, cost matrix C):
1   n ← root node of T
2   r ← C[n][g].rule
3   if r is a chain rule then
4       s ← result of r
5       SelectRules (T, s, C)
6   else
7       for i ← 1 to number of children for n do
8           m ← ith child of n
9           s ← ith nonterminal in pattern of r
10          SelectRules (expression tree rooted at m, s, C)
11  execute actions associated with r
```

Algorithm B.6: Selects optimal sequence of rules that reduces a given expression tree to a given nonterminal, based on costs computed by Alg. B.5.

with nonterminal s as result, it computes the cost c of applying r at n to produce s according to the scheme stated above. If c is less than the currently recorded cost for reducing n to s, then the cost and rule information for n is updated accordingly. The same is then done for all chain rules until it reaches a fixpoint (which must eventually be reached as all rule costs are non-negative and an update only occurs when the cost is strictly less). Since every node is also only processed once, the algorithm runs in linear time with respect to the size of the expression tree.

Having computed the costs, the optimal order of rule reductions – which is equivalent to the least-cost cover – can be found using the algorithm shown in Alg. 2.2. Starting from the root, we select the rule that reduces this node of the expression tree to a particular nonterminal. The same is then done recursively for each nonterminal that appears on the pattern in the selected rule, acting as the goal for the corresponding subtree. Since it is assumed that the machine grammar is in normal form, every pattern is exactly one node which makes it trivial to find the next subtree in the expression tree. The algorithm also correctly applies the necessary chain rules, as the use of such a rule causes the routine to be reinvoked on the same node but with a different goal nonterminal.

DP vs. LR Parsing The DP scheme has several advantages over those based on LR parsing. First, conflicts are automatically handled by the cost-computing algorithm, removing the need of ordering the rules which could affect the code quality yielded by LR parsers. Second, rule cycles that cause LR parsers to get stuck in an infinite loop no longer need to be explicitly broken. Third, machine descriptions can be made more concise as rules differing only in cost can be combined into a single rule. Again taking the VAX machine as an example, Aho et al. reported that the entire Twig specification could be implemented using only 115 rules, which is about half

the size of Ganapathi and Fischer's attribute-based machine grammar for the same target machine.

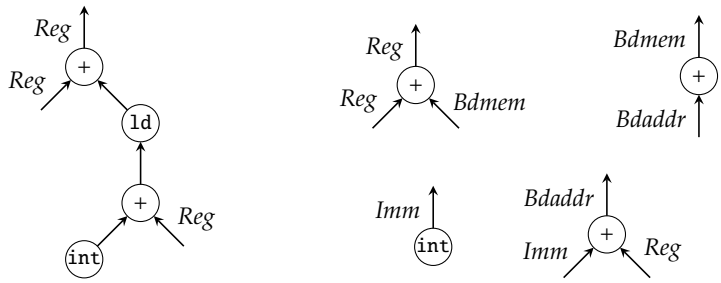
However, the DP approach requires that the code generation problem exhibit properties of optimal substructure, meaning that it is possible to generate optimal assembly code by solving each of its subproblems to optimality. However, this is not always the case. Some solutions, whose total sum is greater compared to another set of selected patterns, can actually lead to better assembly code in the end.

Further Improvements Several improvements of TWIG were later made by Yates and Schwartz [376] and Emmelmann et al. [113]. Yates and Schwartz improved the rate of pattern matching by replacing TWIG's top-down approach with the faster bottom-up algorithm proposed by Hoffmann and O'Donnell, and also extended the attribute support for more powerful predicates. Emmelmann et al. modified the DP algorithm to be run as the expression trees are built by the frontend, which also inlines the code of auxiliary functions directly into the DP algorithm to reduce the overhead. Emmelmann et al. implemented their improvements in a system called the *Back End Generator (BEG)*, and a modified version of this is currently used in the *CoSy* compiler [89].

Fraser et al. [140] made similar improvements in a system called *IBURG* that is both simpler and faster than TWIG; IBURG requires only 950 lines of code compared to TWIG's 3,000 lines of C code, and generates assembly code of comparable quality at a rate that is 25 times faster. IBURG has also been used in several compilers, such as *RECORD* [248, 268] and *REDACO* [226]. Gough and Ledermann [165, 166] later extended the predicate support of IBURG in an implementation called *MBURG*. Both IBURG and MBURG have later been reimplemented in various programming languages, such as the Java-based *JBURG* [178], *OCAMLBURG* [312], which is written in C--, and *GPBURG* [167], which is written in C#.

According to Leupers and Marwedel [247] and Cao et al. [63], Tjiang [351] later merged the ideas of TWIG and IBURG into a new implementation called *OLIVE* (the name is a spin-off of TWIG). Tjiang also made several additional improvements such as supporting rules to use arbitrary cost functions instead of fixed, numeric values. This supports more versatile instruction selection, as rules can be dynamically deactivated by setting infinite costs, which can be controlled from the current context. OLIVE is also used in *SPAM* [346] – a fixed-point DSP compiler – and Araujo and Malik [18] employed it in an attempt to integrate instruction selection with scheduling and register allocation.

Code Size-Reducing Instruction Selection In 2010, Edler von Koch et al. [110] modified the backend in CoSy to perform code generation in two stages in order to reduce code size for architectures with mixed 16-bit and 32-bit instructions, where the former is smaller but can only access a reduced set of registers. In the first stage, instruction selection is performed by aggressively selecting 16-bit instructions. Then, during register allocation, whenever a memory spill is required due to the



(a) Original pattern. (b) Components.

Figure B.13: Example of breaking down a pattern into single-node components in order for it to be supported by a macro-expanding instruction selector [197].

use of a 16-bit instruction, the node "causing" this spill is annotated with a special flag. Once register allocation is finished, another round of instruction selection is performed but this time no nodes which have been annotated are allowed to be covered by patterns originating from 16-bit instructions. Experiments showed that this scheme reduced code size by about 17 % on average compared to CoSy for the selected target architecture and benchmark suite.

Combining DP with Macro Expansion After arguing that Glanville and Graham's method attacked the instruction selection problem from the wrong direction – that is, by defining the instructions in terms of IR operations – Horspool [197] developed in 1987 a technique that essentially is an enhanced form of macro expansion. Because macro-expanding instruction selectors only visit and execute macros one IR node at a time (see Ap. A on p. 152), they do not inherently support instructions where there is an n-to-1 mapping between theIR nodes and the instructions. This limitation can be worked around by incorporating additional logic and bookkeeping into the macro definitions, but doing so by hand often proves to be infeasible. By including an edge labeling step prior to macro expansion, Horspool found a way of supporting such instructions while at the same time simplifying the macro definitions.

The idea is to first break down every pattern into single-node components (see Fig. B.13). As part of the breakdown process the intermediate edges are labeled with *storage classes* which serve as a form of glue between the components, allowing them to be reconnected during macro expansion. The same storage classes can be used across multiple patterns if this is deemed appropriate, which is akin to refactoring an machine grammar in order to reduce the number of rules.

The goal is then to label the edges of the expression tree with storage classes such that they correspond to a least-cost cover of the , which can be done using dynamic programming (but the paper does not go into detail about how the component costs should be assigned). Once the expression tree has been labeled, the assembly code can be emitted using a straightforward macro expander that uses the current

node's type and the storage classes of its edges as indices to a macro table. Since the bookkeeping is essentially lifted into the storage classes, the macro definitions become much simpler compared to those of traditional macro-expanding techniques. Moreover, there is no need to handle backtracking, as such a combination of edge labels would imply an illegal cover of the expression tree.

In principle, Horspool's design is comparable to that of Aho et al., and should yield similar code quality. However, Horspool appears to have had to implement his instruction selection tables by hand whereas Aho et al. built a tool to do it for them.

B.5.3 Faster Pattern Selection with Offline Cost Analysis

In the DP approach just discussed, the rule costs needed for selecting the patterns are dynamically computed while the pattern matcher is completely table-driven. It was later discovered that these calculations can also be done beforehand and represented as tables, improving the speed of the pattern selector as it did for pattern matching. We will refer to this aspect as *offline cost analysis*, which means that the costs of covering any given expression tree are precomputed as part of generating the compiler instead at compile time.

Extending Match Set Labels with Costs To make use of offline cost analysis, we need to extend the labels to not only represent match sets, but also incorporate the information about which pattern will lead to the lowest covering cost given a specific goal. To distinguish between the two, we refer to this extended form of label as a state. A state is essentially a representation of a specific combination of goals, patterns, and costs, where each possible goal g is associated with a pattern p and a relative cost c. A goal in this context typically dictates where the result of an expression must appear, like a particular register class or memory, and in grammar terms this means that each nonterminal is associated with a rule and a cost. This combination is such that

1. for any expression tree whose root has been labeled with a particular state,
2. if the goal of the root must be g,
3. then the entire expression tree can be covered with minimal cost by selecting pattern p at the root. The relative cost of this covering, compared to the scenario in which the goal is something else, is equal to c.

A key point to understand here is that a state does not necessarily need to carry information about how to optimally cover the *entire* expression tree. Indeed, such an attempt would require an infinite number of states. Instead, the states only convey enough information about how to cover the distinct key shapes that can appear in any expression tree. To explain this further, let us observe how most target machines typically operate. Between the execution of two instructions, the data is synchronized by storing it in registers or in memory. The manner in which

```
function LabelTree (expression tree T, list L of state tables):
1   n ← root node of T
2   k ← number of children for n
3   for i ← 1 to k do
4       m_i ← ith child of n
5       LabelTree (expression tree rooted at m_i, L)
6   S ← L[terminal corresponding to n]
7   n.label ← S[m_1.label, ..., m_k.label]
```

Algorithm B.7: Labels an expression tree using states.

some data came to appear in a particular location has in general no impact on the execution of the subsequent instructions. Consequently, depending on the available instructions, one can often break a expression tree at certain key places without compromising code quality. This yields a set of many, smaller expression trees, each with a specific goal at the root, which then can be optimally covered in isolation. In other words, the set of states only needs to collectively represent enough information to communicate where these cuts can be made for all possible expression trees. This does not mean that the expression tree is *actually* partitioned into smaller pieces before pattern selection, but thinking about it in this way helps us understand why we can restrict ourselves to a finite number of states and still get optimal pattern selection.

The algorithm for labeling an expression tree using states is given in Alg. B.7. Since a state is simply an extended form of a label, this algorithm is very similar to the one used in Hoffmann-O'Donnell (compare with Alg. B.2 on p. 179). Pattern selection and assembly code emission is then done as described in Alg. B.8. This is more or less identically to the selection algorithm when computing the costs directly (compare with Alg. 2.2 on p. 26). However, we have yet to describe how to compute the states.

First Technique to Apply Offline Cost Analysis Due to a 1986 paper, Hatcher and Christopher [179] appear to have been pioneers in applying offline cost analysis to pattern selection. Hatcher and Christopher's design, which is an extension of the work by Hoffmann and O'Donnell, can intuitively be described as follows. Given a expression tree whose root has been assigned a label l, find the rule to apply such that the entire tree can be reduced to a given nonterminal at lowest cost. Hatcher and Christopher argued that for optimal pattern selection we can consider each pair of a label l and nonterminal N, and then always apply the rule that will reduce the largest expression tree T_l, which is representative of l, to N at the lowest cost. In Hoffmann and O'Donnell's design, where there is only one nullary symbol that may match any subtree, T_l is equal to the largest pattern appearing in the match set. However, to accommodate machine grammars Hatcher and Christopher's version

```
function Select (labeled expression tree T, goal nonterminal g, list L of rule lookup tables):
1   n ← root node of T
2   R ← L[n.label]
3   r ← R[g]
4   if r is a chain rule then
5       s ← result of r
6       Select (T, s, L)
7   else
8       for i ← 1 to number of children for n do
9           m ← ith child of n
10          s ← ith nonterminal in pattern of r
11          Select (expression tree rooted at m, s, L)
12  execute actions associated with r
```

Algorithm B.8: Selects optimal sequence of rules that reduces a given labeled expression tree to a given nonterminal.

includes one nullary symbol per nonterminal. This means that T_l has to be found by overlapping all patterns appearing in the match set. We then calculate the cost of transforming a larger pattern p into a subsuming, smaller pattern q (hence $p > q$) for every pair of patterns. This cost, which is later annotated to the subsumption graph, is calculated by recursively rewriting p using other patterns until it is equal to q. Hence the cost of this transformation is equal to the sum of all applied patterns. We represent this cost with a function $\mathit{reducecost}(p \xrightarrow{*} q)$. With this information, we retrieve the rule that leads to the lowest-cost of T_l to a goal g by finding the rule r for which

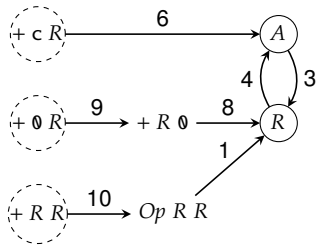
$$\mathit{reducecost}(T_l \xrightarrow{*} g) = \mathit{reducecost}(T_l \xrightarrow{*} \text{pattern tree of } r) + \text{cost of } r.$$

This will select either the largest pattern appearing in the match set of l, or, if one exists, a smaller pattern that in combination with others has a lower cost. We have of course glossed over many details, but this covers the main idea of Hatcher and Christopher's design.

By encoding the selected rules into an additional table to be used during pattern matching, we achieve a completely table-driven instruction selector which also performs optimal pattern selection. Hatcher and Christopher also augmented the original algorithm so that the returned match sets contain all patterns that were duplicated due commutative operations. However, if the pattern set contains patterns which are truly independent, then Hatcher and Christopher's design does not always guarantee that the expression trees can be optimally covered. It is also not clear whether optimal pattern selection for the largest expression trees representative of the labels is an accurate approximation for optimal pattern selection for all possible expression trees.

#	pattern
1	$R \rightarrow Op\ A\ A$
2	$R \rightarrow \texttt{r}$
3	$R \rightarrow A$
4	$A \rightarrow R$
5	$A \rightarrow \texttt{c}$
6	$A \rightarrow +\ \texttt{c}\ R$
7	$\texttt{c} \rightarrow \texttt{0}$
8	$X \rightarrow +\ X\ \texttt{0}$
9	$+\ Y\ X \rightarrow +\ X\ Y$
10	$Op\ X\ Y \rightarrow +\ X\ Y$

(a) BURS grammar.



(b) Example of an LR graph based on the expression tree + 0 + c c and the grammar on the left-hand side. Dashed nodes represent subtrees of the expression tree and fully drawn nodes represent goals. Edges indicate rule applications, with the number of the applied rule appearing next to the edge.

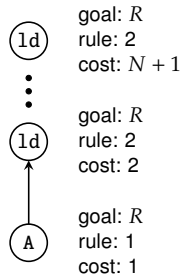
Figure B.14: Example of a BURS grammar and an LR graph [298].

Generating the States Using BURS Theory A different and more well-known method for generating the states was developed by Pelegrı-Llopart and Graham [298]. In a seminal paper from :1988, Pelegrı-Llopart and Graham prove that the methods of tree rewriting can always arranged such that all rewrites occur at the leaves of the tree, resulting in a *bottom-up rewriting system (BURS)*. We say that a collection of such rules constitute a *BURS grammar*, which is similar to the grammars already seen, with the exception that BURS grammars allow multiple symbols – including terminals – to appear on the left-hand side of a production. An example of such a grammar is given in Fig. B.14a. As an extension to the work of Zimmermann and Gaul [383], Dold et al. [102] later developed a method for proving the correctness of BURS grammars using abstract state machines.

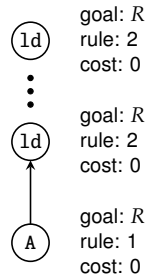
Using BURS theory Pelegrı-Llopart and Graham developed an algorithm that computes the tables needed for optimal pattern selection based on a given BURS grammar. The idea is as follows. For a given expression tree T, a *local rewrite (LR) graph* is formed where each node represents a specific subtree appearing in T and each edge indicates the application of a particular rewrite rule on that subtree (an example is shown in Fig. B.14b). Setting some nodes as goals (that is, the desired results of tree rewriting), a subgraph called the *uniquely invertable (UI) LR graph* is then selected from the LR graph such that the number of rewrite possibilities is minimized. Each UI LR graph then corresponds to a state, and by generating all LR graphs for all possible expression trees that can be given as input, we can find all the necessary states. Since finding a UI LR graph is an NP-complete problem, Pelegrı-Llopart and Graham applied a heuristic that iteratively removes nodes

#	pattern	cost
1	$R \rightarrow$ `var`	1
2	$R \rightarrow$ `load` R	1

(a) A BURS grammar that may lead to an unbounded number of states.



(b) Expression tree labeled with states that incorporate the full cost (requires $N + 1$ states).



(c) Same tree but labeled with states that incorporate the delta costs (requires only two states).

Figure B.15: Example illustrating how incorporating costs into states can result in an infinite number of states [298].

which are deemed "useless" until a UI LR graph is achieved.

Achieving a Bounded Number of States To achieve optimal pattern selection, the LR graphs are augmented such that each node no longer represents a pattern tree but a (p, c) pair, where c denotes the minimal cost of covering the corresponding subtree with pattern p. This is the information embodied by the states as discussed earlier. A naive approach would be to include the *full* cost of reaching a particular pattern into the state, but depending on the rewrite system this may require an infinite number of states. An example where this occurs is given in Fig. B.15b.

A better method is to instead account for the *relative* cost of a selected pattern. This is achieved by computing c as the difference between the cost of p and the smallest cost associated with any other pattern appearing in the LR graph. This yields the same optimal pattern selection but the number of needed states is bounded, as seen in Fig. B.15c. This cost is called the *delta cost* and the augmented LR graph is thus known as a *δ-LR graph*. To limit the memory footprint when generating the δ-LR graphs, Pelegrı-Llopart and Graham used an extension of Chase's table compression algorithm [72] (which we discussed in Sect. 4).

During testing, Pelegrı-Llopart and Graham reported that their implementation yielded state tables only slightly larger than those produced by LR parsing. They also reported that it generated assembly code of quality comparable to Twig's but at a rate that was about five times faster.

BURS ⇏ Offline Cost Analysis Since Pelegrı-Llopart and Graham's 1988 paper, many later publications mistakenly associate to the idea of offline cost analysis with BURS theory, typically using terms like *BURS states*, when these two aspects are in fact orthogonal to each other. Although the work by Pelegrı-Llopart and

Graham undoubtedly led to making offline cost analysis an established aspect of modern instruction selection, the application of BURS theory is only *one* means to achieving optimal pattern selection using tables.

For example, in 1990 Balachandran et al. [32] introduced an alternative method for generating the states that is both simpler and more efficient than that of Pelegrı-Llopart and Graham. At its heart their algorithm iteratively creates new states using those already committed to appear in the state tables. Remember that each state represents a combination of nonterminals, rules, and costs, where the costs have been normalized such that the lowest cost of any rule appearing in that state is 0. Hence two states are identical if the rules selected for all nonterminals and costs are the same. Before a new state is created it is first checked whether it has already been seen – if not, then it is added to the set of committed states – and the process repeats until no new states can be created. We will go into more detail shortly.

Compared to Pelegrı-Llopart and Graham, this algorithm is less complicated and also faster as it directly generates a smaller set of states instead of first enumerating all possible states and then reducing them. In addition, Balachandran et al. expressed the instructions as a more traditional machine grammar – like those used in the Glanville-Graham approach – instead of as a BURS grammar.

A Work Queue Approach for State Table Generation Another state-generating algorithm similar to the one by Balachandran et al. was proposed by Proebsting [303, 306]. This algorithm was also implemented by Fraser et al. [141] in a renowned code generation system called *Burg*.[10] Since its publication in 1992, the paper has sparked a naming convention within the compiler community which we call the *BURGer phenomenon*.[11] Although Balachandran et al. were first, we will continue with studying Proebsting's algorithm as it is better documented. More details are also available in Proebsting's doctoral dissertation [304].

The idea for computing the states – which will only be described briefly – works as follows. For each terminal representing a k-argument operation, an k-dimensional matrix is maintained. This is called the terminal's *state table*, which indicates the state to assign such nodes given the labels of its children. First the states for all leaf nodes are built, considering only base rules with a single terminals on the right-hand side in the production. The costs and rule decisions are computed using the same logic as in Alg. B.5, lines 8–21. The leaf state are then pushed

[10] The keen reader will notice that Fraser et al. also implemented the DP-based system IBURG which was introduced in Sect. 11. The connection between the two is that IBURG began as a testbench for the grammar specification to be used as input to BURG. Fraser et al. later recognized that some of the ideas for the testbench showed some merit in themselves, and therefore improved and extended them into a stand-alone generator. Unfortunately the authors neglected to say in their papers what these acronyms stand for. The author's tentative guess is that BURG was derived from the BURS acronym and stands for *Bottom-Up Rewrite Generator*.

[11] During the research for this dissertation, the author came across the following systems, all with equally creative naming schemes: BURG [141], CBURG [329], DBURG [121], GBURG [142], GPBURG [167], HBURG [47], IBURG [140], JBURG [178], LBURG [175], MBURG [165, 166], OCAMLBURG [312], and WBURG [307].

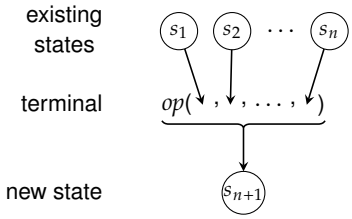


Figure B.16: Creation of a new state.

onto a queue. Each popped state is used as the ith child to all base rules with a non-leaf terminals in combination with all other existing states (see Fig. B.16). If any combination gives rise to a new set of costs and rule decisions, then a new state is created and pushed onto the queue after having updated the state tables. This process continues until the queue is empty, whereupon all necessary states have been built.

Further Improvements The time required to generate the state tables can be decreased if the number of committed states can be minimized. According to Proebsting [306], the first attempts to do this were made by Henry [182], whose methods were later improved and generalized by Proebsting [303, 306]. Proebsting developed two methods for reducing the number of generated states: *state trimming*, which extends and generalizes Henry's ideas; and a new technique called *chain rule trimming*. Without going into details, state trimming increases the likelihood that two created states will be identical by removing the information about nonterminals that can be proven to never take part in a least-cost covering. Chain rule trimming then further minimizes the number of states by attempting to use the same rules whenever possible. This technique was later improved by Kang and Choe [207, 208], who exploited properties of common machine descriptions to decrease the amount of redundant state testing.

More Applications The approach of extending pattern selection with offline cost analysis has been applied in numerous compiler-related systems. Some notable applications that we have not already mentioned include *UNH-Codegen* [181], *DCG* [118], *LBurg* [175], and *WBurg* [307]. Burg is also available as a Haskell clone called *HBurg* [47], and has been adapted by Boulytchev [50] to assist instruction set selection. LBurg was developed to be used in the *Little C Compiler (LCC)* [175], and was adopted by Brandner et al. [53] in designing an architecture description language from which the instructions can automatically be inferred. LBurg was also extended by Farfeleder et al. [125] to support certain multi-output instructions by adding an additional, handwritten pass in the pattern matcher.

B.5.4 Generating States Lazily

The two main approaches for achieving optimal pattern selection – those that dynamically compute the costs as the function is compiled, and those that rely on statically computed costs via state tables – both have their respective advantages and drawbacks. The former have the advantage of being able to support dynamic costs (meaning the pattern cost is not fixed but depends on the context), but they are also considerably slower than their purely table-driven counterparts. The latter yield faster but larger instruction selectors due to the use of state tables, which are also very time-consuming to generate – for pathological grammars this may even be infeasible – and they only support grammar rules with fixed costs.

Combining the Best of State Tables and DP In 2006, Ertl et al. [122] introduced a method that allows the state tables to be generated lazily and on demand. The intuition is that instead of generating the states for *all* possible expression trees in advance, one can get away with only generating the states needed for the expression trees that actually appear in the function.

The scheme can be outlined as follows. As the instruction selector traverses a expression tree, the states required for covering its subtrees are created using dynamic programming. Once the states have been generated, the subtree is labeled and patterns are selected using the familiar table-driven techniques. Then, if an identical subtree is encountered elsewhere – either in the same expression tree or in another tree of the function – the same states can be reused. This allows the cost of state generation to be amortized as the subtree can now be optimally covered faster than if it had been processed using a purely DP-based pattern selector. Ertl et al. reported the overhead of state reuse was minimal compared to purely table-driven implementations. They also reported that the time required to first compute the states and then label the expression trees was on par with selecting patterns using ordinary DP-based techniques. Moreover, by generating the states lazily it is possible to handle larger and more complex machine grammars which otherwise would require an intractable number of states.

Ertl et al. also extended this design to support dynamic costs by recomputing and storing the states in hash tables whenever the costs at the expression tree roots differ. The authors noted that while this incurs an additional overhead, their instruction selector was still faster than a purely DP-based instruction selector.

B.6 Other Tree-Based Approaches

So far we have discussed the conventional methods of covering trees: LR parsing, top-down recursion, dynamic programming, and the use of state tables. In this section we will look at other designs which also rely on trees, but solve the instruction selection problem using alternative methods.

B.6.1 Techniques Based on Formal Frameworks

Homomorphisms and Inversion of Derivors In order to simplify the machine descriptions and enable formal verification, Giegerich and Schmal [162] proposed in 1988 an algebraic framework intended to support all aspects of code generation, including instruction scheduling and register allocation. In brief terms Giegerich and Schmal reformulated the instruction selection problem into a "problem of a hierarchic derivor," which essentially entails the specification and implementation of a mechanism

$$\gamma : T(Q) \to T(Z),$$

where $T(Q)$ and $T(Z)$ denote the term algebras for expressing functions in an intermediate language and target machine language, respectively. Hence γ can be viewed as the resulting instruction selector. Most machine descriptions, however, are typically expressed in terms of Z rather than Q. We therefore view the machine specification as a homomorphism

$$\delta : T(Z) \to T(Q),$$

and the task of an instruction selection-generator is thus to derive γ by inverting δ. Usually this is achieved by resorting to pattern matching, but for optimal instruction selection the generator must also interleave the construction of the inverse δ^{-1} with a *choice function* ξ whenever some $q \in T(Q)$ has several $z \in T(Z)$ such that $\delta(q) = z$. Conceptually this gives us the following functionality:

$$T(Q) \xrightarrow{\delta^{-1}} 2^{T(Z)} \xrightarrow{\xi} T(Z).$$

In the same paper, Giegerich and Schmal also demonstrate how some other methods, such as tree parsing, can be expressed using this framework. A similar scheme based on rewriting techniques was later proposed by Despland et al. [97, 98] in an implementation called PAGODE [62].

Equational Logic Shortly after Giegerich and Schmal, Hatcher [180] developed a design similar to that of Pelegrı-Llopart and Graham that relies on *equational logic* [291] instead of BURS theory. The two are closely related in that both apply a set of predefined rules to rewrite the expression tree into a single goal term. However, an equational specification has the advantage that all such rules – which are derived from the instructions and axiomatic transformations – are based on a set of so-called *built-in operations*. Each built-in operation has a cost and implicit semantics, expressed as assembly code emission. The cost of a rule is then equal to the sum of all built-in operations it applies, removing the need to set the rule costs manually. In addition, no built-in operations are predefined, but are instead given as part of the equational specification, providing a very general mechanism for describing target machines. Experimental results with an implementation called

UCG[12] show that it could, for a selected set of problems, generate assembly code of comparable quality to that of contemporary techniques but in less time.

B.6.2 More Tree Rewriting-Based Methods

We have already discussed numerous techniques which perform instruction selection by rewriting the expression tree such that it finally reaches a particular goal. For completeness we will in this section examine the remaining such designs, but without going into much detail.

Using Finite Tree Automata, Series Transducers, and Pushdown Automata Emmelmann [114] introduced in 1992 a technique that relies on the theories of finite tree automata (see for example [156] for an overview), which was later extended by Ferdinand et al. [131]. In their 1994 paper, Ferdinand et al. demonstrate how finite tree automata can be used to solve both pattern matching and pattern selection – greedily as well as optimally – and also present algorithms for how to produce these automata. An experimental implementation demonstrated the feasibility of this technique, but the results were not compared to those of other techniques. Similar designs were later proposed by Borchardt [48] and Janoušek and Málek [201], who made use of *tree series transducers* (see for example [116] for an overview) and pushdown automata, respectively.

Rewriting Strategies In 2002, Bravenboer and Visser presented a design where rule-based function transformation systems [359] are adapted to instruction selection. Through a system called *STRATEGO* [360], a machine description can be augmented by pattern selection strategies, allowing the pattern selector to be tailored to that particular target machine. Bravenboer and Visser refer to this as providing a rewriting strategy, and their system supports modeling of several strategies such as exhaustive search, maximum munch, and dynamic programming. Purely table-driven techniques, however, do not seem to be supported at the time of writing, which excludes the application of offline cost analysis. In their paper, Bravenboer and Visser argue that this setup allows several pattern selection techniques to be combined, but they do not provide an example of where this would be beneficial.

B.6.3 Techniques Based on Genetic Algorithms

To solve the pattern selection problem, Shu et al. [337] employed the theories of *genetic algorithms (GA)*, which mimic the process of natural selection (see for example [316] for an overview).[13] The idea is to formulate a solution as a string, called a

[12] The paper does not say what this acronym stands for.

[13] On a related note, Wu and Li [371] applied *ant colony optimization* – a meta-heuristic inspired by the shortest-path searching behavior of various ant species [105] – to improve overall code size by alternating between instruction sets on a per-function basis.

chromosome (or *gene*), and then mutate it in order to hopefully end up with a better solution. For a given expression tree whose match sets have been found using an $O(nm)$ pattern matcher, Shu et al. formulated each chromosome as a binary bit string where a 1 indicates the selection of a particular pattern. Likewise, a 0 indicates that the pattern is not used in the tree covering. The length of a chromosome is therefore equal to the sum of the number of patterns appearing in all match sets. The objective is then to find the chromosome which maximizes a *fitness function* f, which Shu et al. defined as

$$f(c) = \frac{1}{k * p_c + n_c},$$

where k is a tweakable constant greater than 1, p_c is the number of selected patterns in the chromosome c, and n_c is the number of nodes in c which are covered by more than one pattern. Hence patterns are allowed to overlap in covering the expression tree. First, a fixed number of chromosomes is randomly generated and evaluated. The best ones are kept and subjected to standard GA operations – such as fitness-proportionate reproduction, single-point crossover, and one-bit mutations – in order to produce new chromosomes, and the process repeats until a termination criterion is reached. The authors claim to be able to find optimal tree covers in "reasonable" time for medium-sized expression trees, but these include at most 50 nodes. Moreover, due to the nature of GAs, optimality cannot be guaranteed for all expression trees. A similar technique was devised by Eriksson et al. [119], which also incorporates instruction scheduling, for generating assembly code for clustered VLIW architectures.

B.6.4 Techniques Based on Trellis Diagrams

The last instruction selection technique that we will examine in this appendix is a rather unusual design by Wess [365, 366]. Specifically targeting digital signal processors, Wess's design integrates instruction selection with register allocation through the use of trellis diagrams.

A *trellis diagram* is a graph where each node consists of an *optimal value array (OVA)*. An element in an OVA represents that the data is stored either in memory (`m`) or in a particular register ($\mathtt{r_x}$), and its value indicates the lowest accumulated cost from the leaves to the node. An example is shown in Tab. B.6, where *TL* denotes the target location of the data produced at a given node, and *RA* denotes the set of registers that may be used when producing the data. The cost is computed similarly as in the DP-based techniques. To facilitate the following discussion, let us denote by $TL(i, n)$ and $RA(i, n)$ the target location and set of available registers, respectively, for the ith element in the OVA of node n.

We create the trellis diagram using the following scheme. For each node in the expression tree, a new node representing an OVA is added to the trellis diagram. For the leaves an additional node is added in order to handle situations where the values first need to be transferred to another location before being used (this is needed for example if the value resides in memory). Next we add the edges. Let

i	0	1	2	3	4
OVA					
TL	m	r_1	r_1	r_2	r_2
RA	$\{r_1, r_2\}$	$\{r_1\}$	$\{r_1, r_2\}$	$\{r_2\}$	$\{r_1, r_2\}$

Table B.6: Example of an OVA for a target machine with two registers r_1 and r_2 [365].

us denote by $e(i, n)$ the ith element in the OVA of a node n. For a unary operation node n with a child m, we add an edge between $e(i, n)$ and $e(j, m)$ if there exists a sequence of instructions that implements the operation of n, stores the result in $TL(i, n)$, takes as input the value stored in $TL(j, m)$, and exclusively uses the registers in $RA(i, n)$. Similarly, for a binary operation node o with two children n and m, we add an edge pair from $e(i, n)$ and $e(j, m)$ to $e(k, o)$ if there exists a sequence of instructions that implements the operation of o, stores the result in $TL(k, o)$, takes as input the two values stored in $TL(i, n)$ and $TL(j, m)$, and exclusively uses the registers in $RA(k, o)$. This can be generalized to n-ary operations. An example is given in Fig. B.17.

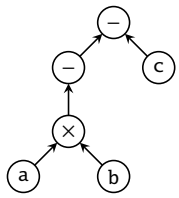
The edges in the trellis diagram thus correspond to the possible combinations of instructions and registers that implement a particular operation in the expression tree. A path from every leaf in the trellis diagram to its root thus represents a selection of such combinations. By keeping track of the costs, we can get the optimal instruction sequence by selecting the path which ends at the OVA element with the lowest cost in the root of the trellis diagram.

The strength of Wess's design is that target machines with asymmetric register classes – where different instructions are needed for accessing different registers – are easily handled as instruction selection and register allocation is done simultaneously. The drawback is that the number of nodes in the trellis diagram is exponential in the number of registers. This problem was mitigated by Fröhlich et al. [145], who augmented the algorithm to build the trellis diagram in a lazy fashion. However, both schemes nonetheless require a 1-to-1 mapping between the nodes in a trellis diagram and the instructions in order to be effective.

This, in combination of how instructions are selected, makes one wonder whether these techniques actually conform to the principles of and DAG covering. The author certainly struggled with deciding how to categorize them, and finally opted against creating a separate principle, as that would indeed be a very short appendix.

B.7 Limitations of Tree Covering

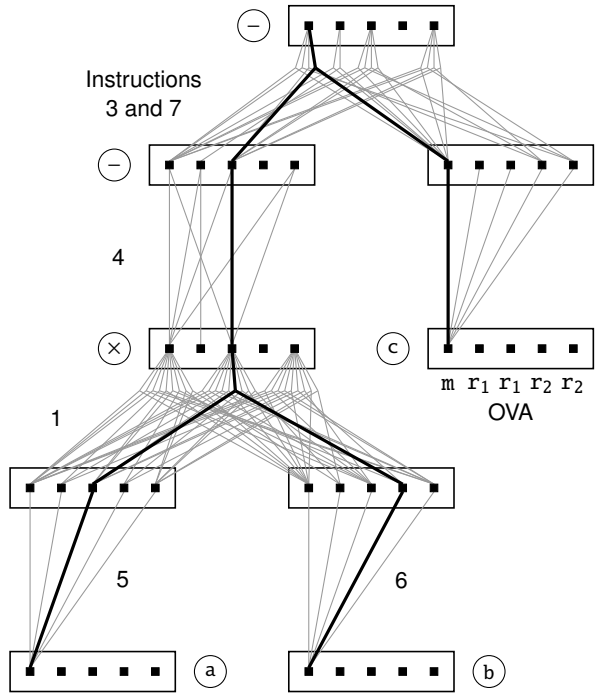
While tree covering enables use of more complex patterns compared to macro expansion, tree covering has several disadvantages of its own.



(a) Expression tree.

#	production
1	$r_1 \leftarrow r_1 \times r_2$
2	$r_1 \leftarrow r_1 \times m$
3	$r_1 \leftarrow r_1 - m$
4	$r_1 \leftarrow - r_1$
5	$r_1 \leftarrow m$
6	$r_2 \leftarrow m$
7	$m \leftarrow r_1$
8	$m \leftarrow r_2$
9	$r_2 \leftarrow r_1$
10	$r_1 \leftarrow r_2$

(b) Instruction set. All instructions are assumed to have equal cost. Note that this is not a machine grammar.



(c) Trellis diagram. Gray edges represent available paths, and black edges indicate the (selected) optimal path.

Figure B.17: A trellis diagram corresponding to an expression $-(a \times b) - c$ for a two-register target machine. The variables a, b, and c are assumed to initially be stored in memory. Note that two instructions are selected for the root as the result is also required to be stored in memory [365].

The first disadvantage of trees has to do with expression modeling. Due to the definitions of trees, common subexpressions cannot be properly modeled in a expression tree. For example, the inlined code cannot be modeled directly without applying one of the following workarounds:

```
x = a + b
y = x + x
```

1. Repeating the shared operations, which in Polish notation results in

$$= y + + a b + a b.$$

2. Splitting the expression, which results in

$$= x + a b$$
$$= y + x x.$$

The first approach leads to additional instructions in the assembly code, while the second hinders the use of more complex instructions. Hence code quality is compromised in both cases.

The second disadvantage is limited instruction set support. For example, since trees only allow a single root, multi-output instructions cannot be represented as pattern trees as such instructions would require multiple roots. Even disjoint-output instructions, where each individual operation can be modeled as trees, cannot be selected because tree covering-based instruction selectors can only consider a single pattern tree at a time.

The third disadvantage is that expression trees typically cannot model control flow. For example, a for loop statement requires a cyclic edge between blocks, which violates the definition of trees. For this reason, tree-based instruction selectors are limited to selecting instructions for a single expression tree at a time, which is known as *local instruction selection*. Moreover, handling of control flow must be done separately, which excludes matching and selection of inter-block instructions, whose behavior incorporates control flow.

To summarize, although the principle of tree covering greatly improves code quality over the principle of pure macro expansion (ignoring peephole optimization, that is), the inherent restrictions of trees prevent exploitation of many instructions provided by modern target machines.

B.8 Summary

In this appendix, we have looked at numerous techniques that are based on the principle of tree covering. In contrast to macro expansion, tree covering enables use of more complex patterns, allowing a wider range of instructions to be selected. By applying dynamic programming, optimal covers can be found in linear time, thereby further improving the quality of the generated assembly code. Several techniques also incorporate offline cost analysis into the instruction selector generator to reduce compilation time. In other words, this kind of implementation is very fast and efficient while also supporting a wide array of target machines. Consequently, tree covering has become the most known – although perhaps no longer the most applied – principle of instruction selection.

Restricting oneself to trees, however, has several inherent disadvantages, and in the next appendix we will look at a more general principle that addresses some of these issues.

APPENDIX

C

DAG Covering

This appendix considers techniques based on DAG covering. First, we introduce the principle in Sect. C.1. We then prove in Sect. C.2 that optimal pattern selection using DAGs is NP-complete. We describe straightforward, greedy approaches in Sect. C.3, moving on to exhaustive techniques in Sect. C.4. In Sect. C.5 we describe techniques that extend methods from tree covering to DAGs. In Sect. C.6 we describe techniques that model instruction selection as a MIS or MWIS problem. In Sect. C.7 we describe techniques that model instruction selection as a unate or binate covering problem. In Sects. C.8 and C.9, we describe techniques based on methods from combinatorial optimization. Other DAG-based approaches that do not fit into any of the sections above are discussed in Sect. C.10. Lastly, we discuss limitations of this principle in Sect. C.11 and summarize in Sect. C.12.

The appendix is based on material presented in [186, Chap. 4] that has been adapted for this dissertation. To not disturb the flow of reading, material already presented in Chap. 2 is duplicated in this appendix.

C.1 The Principle

As we saw in Ap. B, the principle of tree covering has two significant disadvantages. The first is that common subexpressions cannot be properly expressed in expression trees, and the second is that many instruction characteristics – such as multi-output instructions – cannot be modeled as pattern trees. As these shortcomings are primarily due to the restricted use of trees, we can achieve a more powerful approach to instruction selection by operating DAGs, thereby extending tree covering to DAG covering.

By lifting the restriction that every node in the expression tree have exactly one parent, we attain a *block DAG*. Because DAGs permit nodes to have multiple parents, the intermediate values in an expression can be shared and reused within the same block DAG. This also enables *pattern DAGs* that contain multiple root nodes, which

signify the production of multiple output values. Hence the instruction set support is extended to include multi-output instructions.

Since DAGs are less restrictive compared to trees, transitioning from tree covering to DAG covering requires new methods for solving the problems of pattern matching and pattern selection. Pattern matching is typically addressed using one of the following methods:

- First split the pattern DAGs into trees, then match these individually, and then recombine the matched pattern trees into their original DAG form. In general, matching trees on DAGs is NP-complete [154], but designs applying this technique typically sacrifice completeness to retain linear time complexity.
- Match the pattern DAGs directly using a generic subgraph isomorphism algorithm. Although such algorithms exhibit exponential worst-case time complexity, in the average case they often finish in polynomial time and are therefore used by several DAG covering-based designs discussed in this appendix.

Optimal pattern selection on block DAGs, however, does not offer the same range of choices in terms of complexity.

C.2 Optimal Pattern Selection on DAGs Is NP-Complete

The cost of the gain in generality and modeling capabilities that DAGs give us is a substantial increase in complexity. As we saw in Ap. B, selecting an optimal set of patterns to cover a expression tree can be done in linear time, but doing the same for block DAGs is an NP-complete problem. Proofs were given in 1976 by Bruno and Sethi [59] and Aho et al. [5], but these were most concerned with the optimality of instruction scheduling and register allocation. In 1995, Proebsting [305] gave a very concise proof for optimal pattern selection, and a longer, more detailed proof was given by Koes and Goldstein [224] in 2008. In this dissertation, we will paraphrase the longer proof.

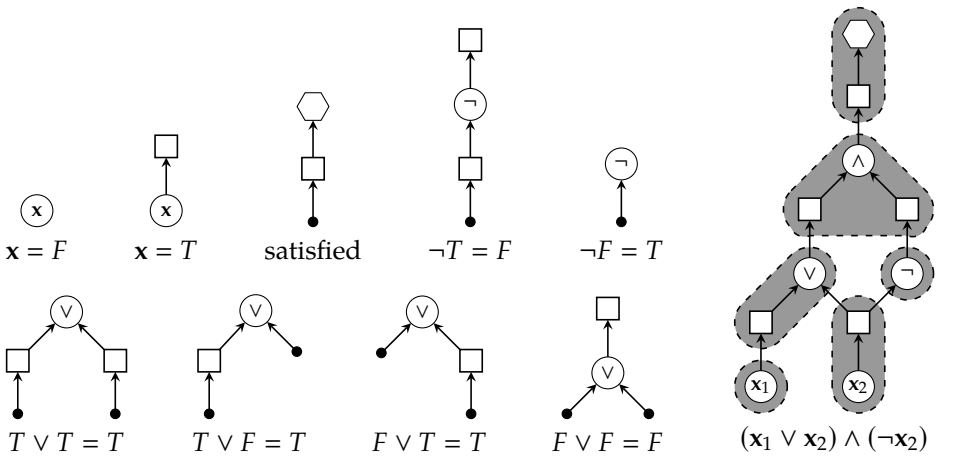
C.2.1 The Proof

The idea behind the proof is to transform the SAT problem into an optimal – that is, least-cost – DAG covering problem. The SAT problem is the task of deciding whether a Boolean formula, written in *conjunctive normal form (CNF)*, can be satisfied. A CNF formula is an expression consisting of conjunctions of disjunctions of Boolean variables In other words, a formula is in CNF if it has the following structure:

$$(\mathbf{x}_1 \vee \mathbf{x}_2 \vee \ldots) \wedge (\mathbf{x}_{n+1} \vee \mathbf{x}_{n+2} \vee \ldots) \wedge \ldots$$

where $\mathbf{x}_i \in \{\text{true}, \text{false}\}$ and $\vee$ and $\wedge$ denotes logical-or and logical-and, respectively. A variable $\mathbf{x}$ can also be negated, written as $\neg\mathbf{x}$.

Since the SAT problem is NP-complete, all polynomial-time transformations from SAT to any other problem P must also render P NP-complete.



(a) The SAT patterns. For brevity, the patterns for the ∧ operation are omitted as these can be easily inferred from the ∨ patterns. All patterns are assumed to have the same unit cost.

(b) Example of a SAT problem represented as a DAG covering problem.

Figure C.1: Transforming SAT to DAG covering [224].

Modeling SAT as a Covering Problem First, we transform an instance S of the SAT problem into a block DAG. The goal is then to find an exact cover for the DAG in order to deduce the truth assignment for the Boolean variables from the set of selected patterns. For this purpose we will use $\vee$, $\wedge$, $\neg$, v, $\square$, and $\hexagon$ as node types, and define $type(n)$ as the type of a node n. Nodes of type $\square$ and $\hexagon$ will be referred to as *box nodes* and *stop nodes*, respectively. Now, for each Boolean variable $x \in S$ we create two nodes n_1 and n_2 such that $type(n_1) = v$ and $type(n_2) = \square$, and add these to the block DAG. At the same time we also add an edge $n_1 \rightarrow n_2$. The same is done for each binary Boolean operator $op \in S$ by creating two nodes n'_1 and n'_2 such that $type(n'_1) = op$ and $type(n'_2) = \square$, along with an edge $n'_1 \rightarrow n'_2$.

To model the connection between the op operation and its two input operands x and y, we add two edges $n_x \rightarrow n'_1$ and $n_y \rightarrow n'_1$ such that $type(n_x) = type(n_y) = \square$. For the unary operation $\neg$ we obviously only need one such edge, and since $\vee$ and $\wedge$ are commutative it does not matter in what order the edges are arranged with respect to the operator node. Hence, in the resulting block DAG, only box nodes will have more than one outgoing edge. An example of such a DAG is shown in Fig. C.1, which can be constructed in linear time simply by traversing the Boolean formula.

Boolean Operations as Patterns To cover the block DAG, we will use the pattern trees given in Fig. C.1a, and we will refer to this pattern set as P_{SAT}. Every pattern in P_{SAT} adheres to the following invariant:

1. If a variable **x** is set to true (T), then the selected pattern covering the **x** node will also cover the corresponding box node of **x**.
2. If the result of an operation *op* evaluates to false (F), then that pattern will not cover the corresponding box node of *op*.

Another way of looking at it is that an operator in a pattern *consumes* a box node if its corresponding value must be set to T, and *produces* a box node if the result must evaluate to F. Using this scheme, we can easily deduce the truth assignments to the variables by inspecting whether the patterns selected to cover the DAG consume the box nodes of the variables. Since the only pattern to contain a stop node also consumes a box node, the entire expression will be forced to evaluate to T.

In addition to the node types that can appear in the block DAG, the patterns can also contain nodes of an additional type, $\bullet$, which we will refer to as *anchor nodes*. Let $numch(n)$ denote the number of children of n, and $child(i, n)$ the ith child of n. We now say that a pattern p, with root node p_r, *matches* the part of a block DAG $\langle N, E\rangle$ which is rooted at a node $n \in N$ if and only if:

1. $type(n) = type(p_r)$,
2. $numch(n) = numch(p_r)$, and
3. $\forall 1 \leq i \leq numch(n) : type(child(i, n)) = \bullet \vee child(i, n)$ matches $child(i, p_r)$.

In other words, the structure of the pattern tree – which includes the node types and edges – must correspond to the structure of the matched subgraph (excluding anchor nodes, which can match any node in the block DAG).

We introduce several new definitions. Given a block DAG $G = \langle N, E\rangle$, let M_n be the set of patterns in P_{SAT} that match at node $n \in N$. Also, given a pattern $\langle N_p, E_p\rangle$, let $matched(p, n_p)$ be the set of nodes in N that are matched by a node $n_p \in N_p$. Lastly, we say that G is *covered* by a function $f : N \to 2^{P_{\text{SAT}}}$, which maps nodes in the block DAG to a set of patterns, if and only if, for each $n \in N$,

1. $\forall p \in f(n) : p$ matches n,
2. $type(n) = \circ \Rightarrow f(n) \neq \emptyset$, and
3. $\forall p = \langle N_p, E_p\rangle \in f(v), n_p \in N_p : type(n_p) = \bullet \Rightarrow f(matched(n, n_p)) \neq \emptyset$.

The first constraint enforces that only valid matches are selected. The second constraint enforces that some match has been selected to cover the stop node. The third constraint enforces that matches have been selected to cover the rest of the DAG. An optimal cover is thus a mapping f which covers the block DAG $\langle N, E\rangle$ and also minimize

$$\sum_{n \in N} \sum_{p \in f(n)} cost(p),$$

where $cost(p)$ is the cost of pattern p.

Optimal Solution to DAG Covering ⇒ Solution to SAT We now postulate that if the optimal cover has a total cost equal to the number of non-box nodes in the block DAG, then the corresponding SAT problem is satisfiable. Since all patterns in P_{SAT} cover exactly one non-box node and have equal unit cost, the condition above means that every node in the DAG is covered by exactly one pattern. This in turn means that exactly one value will be assumed for every Boolean variable and operator result, which is easy to deduce through inspection of the selected matches.

We have thereby shown that an instance of the SAT problem can be solved by transforming it, in polynomial time, to an instance of the optimal DAG covering problem. Hence optimal DAG covering – and therefore also optimal instruction selection based on DAG covering – is NP-complete. □

C.3 Straightforward, Greedy Techniques

Since instruction selection on DAGs with optimal pattern selection is computationally difficult, most instruction selectors based on this principle are suboptimal. One of the first code generators to operate on DAGs was developed by Aho et al. [5]. In their 1976 paper, Aho et al. introduce some simple greedy heuristics for producing assembly code for a commutative one-register target machine. However, these methods assume a 1-to-1 mapping between the nodes in a block DAG and the instructions and thus effectively ignore the instruction selection problem.

C.3.1 LLVM

A more flexible, but still greedy, heuristic is applied in the well-known *LLVM* compiler infrastructure [235]. According to a blog entry by Bendersky [43] – which at the time of writing provides the only documentation, except for the source code itself – the instruction selector is basically a greedy DAG-to-DAG rewriter.[1]

The patterns – which are limited to trees – are expressed in a machine description that allows common features to be factored out into abstract instructions. A tool called *TableGen* expands the abstract instructions into pattern trees, which are then processed by a matcher generator. To ensure a partial order among all patterns, the matcher generator first performs a lexicographical sort on the pattern set, in the following order: (i) by decreasing complexity, which is the sum of the pattern's size and a constant that can be tweaked to give higher priority for particular instructions; (ii) by increasing cost; and (iii) by increasing size of the subgraph that replaces the covered part in the block DAG (if the corresponding pattern is selected). Once sorted, the patterns are converted into small recursive programs which essentially check whether the corresponding pattern matches at a given node in the block DAG. These programs are then compiled into a form of byte code and assembled into a matcher table, arranged such that the lexicographical sort is preserved. The

[1] LLVM is also equipped with a "fast" instruction selector, but it is implemented as a typical macro expander and is only intended to be used when compiling without extensive program optimization.

instruction selector applies this table by simply executing the byte code, starting with the first element. When a match is found, the pattern is greedily selected and the matched subgraph is replaced with the output (usually a single node) of the selected pattern. This process repeats until there are no nodes remaining in the original block DAG.

Although in extensive use (as of version 3.4), LLVM's instruction selector has several drawbacks. The main disadvantage is that any pattern that is not supported by TableGen has to be handled manually through custom C functions. Unlike GCC – which applies macro expansion combined with peephole optimization (see Sect. A.3.2) – this includes all multi-output instructions, since LLVM is restricted to pattern trees only. In addition, the greedy scheme compromises code quality.

C.4 Techniques Based on Exhaustive Search

Although optimal pattern selection can be achieved through exhaustive search, in practice this is typically infeasible due to the exponential number of possible combinations. Nonetheless, there do exist a few techniques that do exactly this, but they apply various techniques to prune the search space.

C.4.1 Extending Means-End Analysis to DAGs

Twenty years after Newcomer and Cattell et al. (see Sect. B.4.1), Yu and Hu [379, 380] rediscovered means-end analysis as a method for instruction selection. They also made two major improvements. First, Yu and Hu's design supports block and pattern DAGs whereas those by Newcomer and Cattell et al. are both limited to trees. Second, it combines means-end analysis with *hierarchical planning* [324], which is a search strategy that relies on the fact that many problems can be arranged in a hierarchical manner for handling larger and more complex problem instances. Using hierarchical planning enables exhaustive exploration of the search space while at the same time avoiding the situations of dead ends and infinite looping that may occur in straightforward implementations of means-end analysis (Newcomer and Cattell et al. both circumvented this problem by enforcing a cut-off when a certain depth in the search space had been reached).

Although this technique exhibits a worst time execution that is exponential in the search depth, Yu and Hu assert that a depth of 3 is sufficient to yield results of equal quality to that of handwritten assembly code. This claim notwithstanding, it is unclear whether it can be extended to support complex instructions such as inter-block and interdependent instructions.

C.4.2 Relying on Semantic-Preserving Transformations

In 1996, Hoover and Zadeck [195] developed a system called *Toast* with the goal of automating the generation of entire compiler frameworks – including instruction scheduling and register allocation – from a declarative machine description. In Toast

the instruction selection is done by applying semantic-preserving transformations during pattern selection to make better use of the instruction set. For example, although $x * 2$ is semantically equivalent to $x \ll 1$, where x is arithmetically shifted one bit to the right which is a faster computation than multiplication. Most instruction selectors, however, will fail to select instructions implementing the latter when the former appears in the block DAG as the patterns are syntactically different from one another.

Their design works as follows. First, the frontend emits block DAGs consisting of semantic primitives, a kind of IR code also used for describing the instructions. The block DAG is then semantically matched using single-output patterns derived from the instructions. Semantic matches – which Hoover and Zadeck call *toe prints* – and are found by a semantic comparator. The *semantic comparator* first performs syntactic matching – that is, checking that the nodes are of the same type, which is done using a straightforward $O(nm)$ algorithm – and then resorts to semantic-preserving transformations for when syntactic matching fails. To bound the exhaustive search for all possible toe prints, a transformation is only applied if it will lead to a syntactic match later on. Once all toe prints have been found, they are combined into *foot prints*, which correspond to the full effects of an instruction. A foot print can consist of just a single toe print (as with single-output instructions) or several (as with multi-output instructions), but the paper lacks details on how this is done exactly. Lastly, all combinations of foot prints are considered in pursuit of the one leading to the most effective implementation of the block DAG. To further prune the search space, this process only considers combinations where each selected foot print syntactically matches at least one semantic primitive in the block DAG. In addition, only "trivial amounts" of the block DAG (such as nodes representing constants) may be included in more than one foot print.

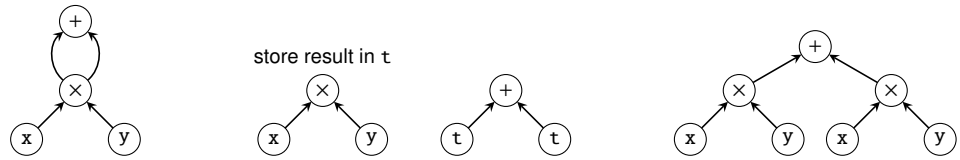
Using a prototype implementation, Hoover and Zadeck reported that almost 10^{70} "implied instruction matches" were found for one of the test cases, but it is unclear how many of them were actually useful. Moreover, in its current form the design appears to be unpractical for generating assembly code for all but very small functions.

C.5 Extending Tree Covering Techniques to DAGs

Another common approach to DAG covering is to reuse already-known, linear-time methods from tree covering. This can be achieved either by transforming the block DAGs into trees, or by generalizing the tree-based algorithms for pattern matching and pattern selection. We begin by discussing designs that apply the first technique.

C.5.1 Undagging Block DAGs

The simplest approach for reusing tree covering techniques is to transform the block DAG into several expression trees. We will refer to this idea as *undagging*.



(a) Block DAG with matches. (b) After edge splitting. (c) After node duplication.

Figure C.2: Example of undagging a block DAG.

As illustrated in Fig. C.2, a block DAG can be undagged into expression trees in two ways. The first approach is to split the edges involving shared nodes – these are nodes where reuse occurs due to the presence of common subexpressions – which results in a set of disconnected expression trees that can then be covered individually. Not surprisingly, this approach is called *edge splitting*. An implicit connection between the expression trees is maintained by forcing the values computed at the shared nodes to be stored and read from a specific location, typically in memory. An example of such an implementation is DAGON, a technology binder developed by Keutzer [216], which maps technology-independent descriptions onto circuits. The second approach is to duplicate the nodes involved in computing the shared value, which is known as *node duplication*. This results in a single but larger expression tree compared to those produced with edge splitting.

Common for both schemes is that they compromise code quality: too aggressive edge splitting produces many small trees that cannot be covered using larger patterns, inhibiting use of more efficient instructions; and too aggressive node duplication incurs a larger computational workload where many operations are needlessly re-executed in the final assembly code. Moreover, the intermediate results of an edge-split block DAG must be forcibly stored in specific locations, which can be troublesome for heterogeneous memory-register architectures (this particular problem was studied by Araujo et al. [19]).

Balancing Splitting and Duplication In 1994, Fauth et al. [129, 277] developed a technique that tries to mitigate the deficiencies of undagging by balancing the use of node duplication and edge splitting. Implemented in the *Common Bus Compiler (CBC)*, the instruction selector applies a heuristic algorithm that first favors node duplication, and resorts to edge splitting when the former is deemed too costly. The decision about whether to duplicate or to split is taken by comparing the cost of the two solutions and selecting the cheapest one. The cost is calculated as a weighted sum $w_1 n_{\text{dup}} + w_2 n_{\text{split}}$, where n_{dup} is the number of nodes in the block DAG (a rough estimate of code size), and n_{split} is the expected number of nodes executed along each execution path (a rough estimate of execution time). Once this is done, each resulting expression tree is covered by an improved version of IBURG (see Ap. B on p. 188) with extended match condition support. However,

the experimental data is too limited for us to judge how efficient this technique is compared to a design where the block DAGs have been transformed into expression trees using just one method.

Register-Sensitive Instruction Selection In 2001, Sarkar et al. [326] developed an instruction selection technique that attempts to reduce the *register pressure* – that is, the number of registers needed by the function – in order to facilitate register allocation.[2]

The design works as follows. The block DAG is first augmented with additional edges to signify scheduling dependencies between memory operations, and then it is split into a several expression trees using a heuristic to decide which edges to break. The expression trees are then covered individually using conventional methods based on tree covering. However, instead of being the usual number of execution cycles, the cost of each instruction is set so as to reflect the amount of register pressure incurred by that instruction (unfortunately, the paper lacks details on how these costs are computed exactly). Once patterns have been selected, the nodes which are covered by the same pattern are merged into super nodes. The resulting graph is then checked for whether it contains any cycles, which may appear due to the extra data dependencies that were added at the earlier stage. If it does, it means that there exist cyclic scheduling dependencies between two or more memory operations, making it an illegal cover. The splits are then reverted and the process repeats until a legal cover is found.

Sarkar et al. implemented their register-sensitive design in *Jalapeño*, a register-based Java virtual machine developed by IBM. For a small set of problems the performance increased by about 10 %, which Sarkar et al. claim to be due to fewer instructions needed for register spilling compared to the default instruction selector. Although innovative, it is doubtful that the technique can be extended much further.

C.5.2 Extending the Dynamic Programming Approach to DAGs

To avoid the application of ad hoc heuristics, several DAG-based instruction selectors perform pattern selection by applying an extension of the tree-based DP algorithm originally developed by Aho and Johnson [4]. According to the literature, Liem et al. [253, 296, 297] appear to have been the first to have done so.

In a seminal paper from 1994, Liem et al. introduce a design which is part of *CodeSyn*, a well-known code synthesis system, which in turn is part of a development environment for embedded systems called *FlexWare*. For pattern matching, Liem et al. applied the same technique as Weingart [362] (see Sect. B.2) by combining all available pattern trees into a single tree of patterns. This pattern tree is traversed in tandem with the block DAG, and for each node an $O(nm)$ pattern

[2] Another register-aware instruction selection technique was developed in 2014 by Xie et al. [374], with the aim of reducing the number of writes to a nonvolatile register file. However, the instructions are selected using a proprietary and greedy heuristic hat does not warrant in-depth discussion.

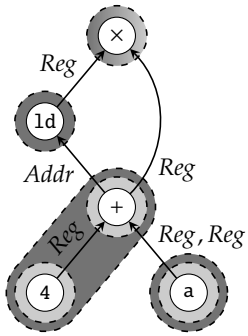


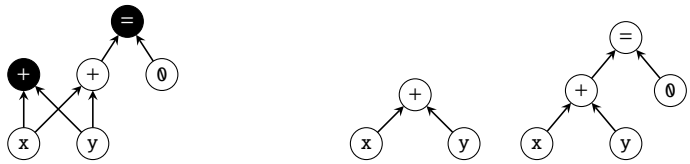
Figure C.3: A block DAG to be covered using tree patterns [121]. The nonterminal produced by a given match is given along the edge where the result is used. Note that the a node can be covered by two matches, both of which reduce the node to the same nonterminal. Hence only one match is needed as the result can be shared by the two matches making use of that nonterminal.

matcher is used to find all match sets. Pattern selection is then performed using an extended version of the DP algorithm, but the paper does not explain how this is done exactly. Moreover, the algorithm is only applied on the data flow of the block DAG – control flow is covered separately using a simple heuristic – and no guarantees are made that the pattern selection is optimal, as that is an NP-complete problem.

Potentially Optimal Pattern Selection In a paper from 1999, Ertl [121] introduces a design which guarantees optimal pattern selection on block DAGs for certain machine grammars. The idea is to first make a bottom-up pass over the block DAG and compute the costs using the conventional DP algorithm as discussed in Ap. B. Each node is thus labeled with the same costs, as if the block DAG had first been transformed into a tree through node duplication. But Ertl recognized that if several patterns reduce the same node to the same nonterminal, then the reduction to that nonterminal can be shared between several rules whose patterns contain the nonterminal. Hence the instructions for implementing shared nonterminals only need to be emitted once, decreasing code size and also improving performance, since the amount of redundant computation is reduced. With appropriate data structures, a linear-time implementation can be achieved.

An example illustrating such a situation is given in Fig. C.3, where we see an addition that will have to be implemented twice, as its node is covered by two separate patterns each of which reduces the subtree to a different nonterminal. The `reg` node, on the other hand, is reduced twice to the same nonterminal (*Reg*), and can thus be shared between the rules that use this nonterminal in the patterns.

As said earlier, however, this technique yields optimal pattern selection only for certain machine grammars. Ertl therefore devised a checker, called *DBurg*, that detects when the grammar does not belong into this category and thus cannot guarantee optimality. The basic idea is to check whether every locally optimal decision is also globally optimal by performing inductive proofs over the set of all possible block DAGs. To do this efficiently, Ertl implemented DBurg using the ideas behind Burg (hence the name).



(a) Original pattern DAG. (b) Partial pattern trees.

Figure C.4: Example of converting a pattern DAG into partial pattern trees. The pattern DAG represents an `add` instruction that also sets a status flag if the result is equal to 0. The black nodes indicate the output nodes.

Combining DP and Edge Splitting Koes and Goldstein [224] extended Ertl's ideas by providing a heuristic that splits the block DAG at points where node duplication is estimated to have a detrimental effect on code quality. Like Ertl's algorithm, Koes and Goldstein's first selects patterns optimally by performing a tree-like, bottom-up DP pass which ignores the fact that the input is a DAG. Then, at points where multiple patterns overlap, two costs are calculated: an *overlap-cost* and a *cse-cost*. The overlap-cost is an estimate of the cost of letting the patterns overlap and thus incur duplication of operations in the final assembly code. The cse-cost is an estimate of the cost of splitting the edges at such points. If cse-cost is lower than overlap-cost, then the node where overlapping occurs is marked as *fixed*. Once all such nodes have been processed, a second bottom-up DP pass is performed on the block DAG, but this time no patterns are allowed to span across fixed nodes, which can only be matched at the root of a pattern. Lastly, a top-down pass emits the assembly code.

For evaluation purposes Koes and Goldstein compared their own implementation, called *Noltis*, against an implementation based on integer programming – we will discuss such techniques later in this appendix – and found that Noltis achieved optimal pattern selection in 99.70 % of the test cases. More details are given in Koes's doctoral dissertation [223]. But like Ertl's design, Koes and Goldstein's is limited to pattern trees and thus cannot support more complex instructions such as multi-output instructions.

Supporting Multi-output Instructions In most instruction selection techniques based on DAG covering, it is assumed that the outputs of a pattern DAG always occur at the root nodes. But in a design by Arnold and Corporaal [23, 24] (originally introduced in a technical report by Arnold [22]), the nodes representing output can be marked explicitly. The advantage of this is that it allows the pattern DAGs to be fully decomposed into trees such that each output value receives its own pattern tree, which Arnold and Corporaal call *partial patterns*. An example is given in Fig. C.4.

The partial patterns are then matched over the block DAG using an $O(nm)$ algorithm. After matching, another algorithm attempts to merge appropriate combinations of partial matches into matches of the original pattern DAG. This is

done in a straightforward manner by maintaining, for each match, an array that maps the nodes in the pattern DAG to the covered nodes in the block DAG. Then, a check is made on whether two partial patterns belong to the same original pattern DAG and have compatible mappings. If a pair of pattern nodes belong to different partial patterns but correspond to the same node in the original pattern DAG, then both pattern nodes must cover the same node in the block DAG. For pattern selection, Arnold and Corporaal applied a variant of the DP scheme but combined it with a greedy heuristic in order to enforce that each node is covered exactly once. Hence code quality is compromised.

C.6 Modeling Instruction Selection as an M(W)IS Problem

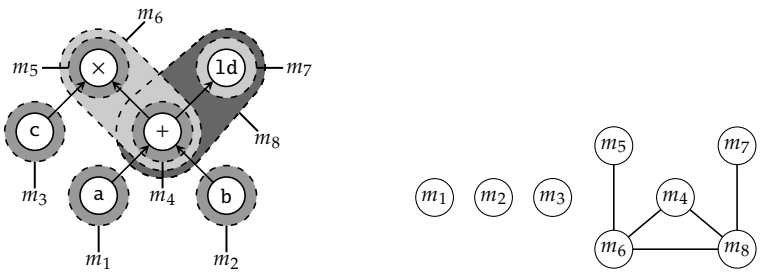
In the techniques discussed so far, the instruction selector operates directly on the block DAG when performing pattern selection. The same applies for most designs based on tree covering. But another approach is to indirectly solve the pattern selection problem by first transforming it into an instance of some other problem for which there already exist efficient solving methods. When that problem has been solved, the answer can be translated back into a solution for the original pattern selection problem.

One such problem is the *maximal independent set (MIS) problem*, where the task is to select the largest set of nodes from a graph such that no pairs of selected nodes have an edge between them. In the general case, finding such a solution is NP-complete [154], and the pattern selection problem is transformed into an MIS problem as follows. From the match sets found by pattern matching, a corresponding *conflict graph* – or *interference graph*, as it is sometimes called – is formed. Each node in the conflict graph represents a match, and there exists an edge between two nodes if and only if the corresponding matches overlap. An example of this is given in Fig. C.5. By solving the MIS problem for the conflict graph, we obtain a selection of matches such that every node in the block DAG is covered by exactly one match.

But a solution to the MIS problem does not necessarily yield an optimal solution to the pattern selection problem, as the former does not incorporate costs. We address this limitation by transforming the MIS problem into a *maximal/minimal weighted independent set (MWIS) problem*, where the task is to find a solution to the MIS problem that maximizes (or minimizes) $\sum_p \mathit{weight}\, p$, and assign as weights the costs of the patterns. We can get the solution with minimal total cost simply by negating the weights. Note that although the MWIS-based techniques discussed in this dissertation have all been limited to block DAGs, the approach can just as well be applied in graph covering, which will be introduced in Ap. D.

C.6.1 Applications

In 2007, Scharwaechter et al. [329] introduced what appears to be the first instruction selection technique to use the MWIS approach for selecting patterns. But despite



(a) Block DAG with matches. (b) Interference graph.

Figure C.5: Example of modeling instruction selection as a MIS problem. Valid maximal independent sets of the interference graph are $\{m_1, \ldots, m_5, m_7\}$, $\{m_1, m_2, m_3, m_5, m_8\}$, and $\{m_1, m_2, m_3, m_6, m_7\}$, which correspond to valid covers of the block DAG.

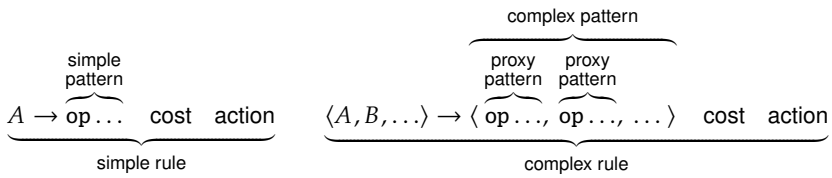


Figure C.6: Anatomy of simple and complex rules in an extended machine grammar.

this novelty, the most cited contribution of their design is its extensions to machine grammars to support multi-output instructions.

Machine Grammars with Multiple Left-Hand Side Nonterminals Scharwaechter et al. [329] appears to have pioneered the modeling of instruction selection as a MWIS problem, although the main contribution of their paper is the extension of machine grammars to handle multi-output instructions. The idea is to model such instructions using *complex rules*, which each consists of multiple productions – one for every result. In this dissertation, such productions and their patterns are called *proxy rules*[3] and *proxy patterns*, respectively, whereas rules with a single production and their patterns are called *simple rules* and *simple patterns*, respectively. The rule structure is also illustrated in Fig. C.6.

Pattern matching is a two-step process. First, the match sets are found for the simple and proxy patterns, using conventional tree-based pattern matching techniques. Second, the match sets for the complex patterns are found by combining the matches of proxy patterns into matches of complex patterns where appropriate. The pattern selector then checks whether it is worth applying a complex pattern for

[3] In the original paper, they are called *split rules*.

covering a certain set of nodes, or if they should be covered using the simple patterns instead. Since the intermediate results of nodes within complex patterns cannot be reused for other patterns, selecting a complex pattern can incur an additional overhead cost as nodes in the block DAG may need to be covered using multiple patterns. Consequently, a complex pattern will only be selected if the cost reduced by replacing a set of simple patterns with this pattern is greater than the cost incurred by code duplication.

After these decisions have been taken, the next step is to perform pattern selection. For this, Scharwaechter et al. solve the corresponding MWIS problem in order to limit solutions to those of exact covering only. The weights are calculated as the negated sum of the proxy pattern costs, but the paper is ambiguous on how these costs are calculated. Since the MWIS problem is known to be NP-complete, Scharwaechter et al. employed a greedy heuristic called *GWMIN2* by Sakai et al. [325]. Lastly, proxy patterns which have not been merged into complex patterns are replaced by corresponding simple patterns before assembly code emission.

Scharwaechter et al. implemented a prototype called *CBURG* as an extension of OLIVE (see Ap. B on p. 188), and then ran some experiments by targeting a MIPS-like architecture. In these experiments CBURG generated assembly code which improved performance by almost 25 %, and reduced code size by nearly 22 %, compared to assembly code which was only allowed to make use of single-output instructions. Measurements of CBURG also indicate that this technique exhibits near-linear time complexity. Ahn et al. [3] later broadened this work by including scheduling dependency conflicts between complex patterns, and incorporating a feedback loop with the register allocator to facilitate register allocation.

A shortcoming of both designs by Scharwaechter et al. and Ahn et al. is that complex rules can only consist of disconnected pattern trees (hence there can be no sharing of nodes between the proxy patterns). Youn et al. [377] address this problem in a 2011 paper – which is a revised and extended version of the original paper by Scharwaechter et al. – by introducing index subscripts for the operand specification of the complex rules. However, the subscripts are restricted to the input nodes of the pattern, still hindering support for completely arbitrary pattern DAGs.

Targeting Machines with Echo Instructions In 2004, Brisk et al. [57] introduced a technique to perform instruction selection for target machines with special *echo instructions*, which are small markers that refer back to an earlier portion in the assembly code for re-execution. This allows the assembly code to be compressed by basically using the same idea that is applied in the LZ77 algorithm [384].[4] Since echo instructions do not incur a branch or a procedure call, the assembly code can be reduced in size without sacrificing performance. Consequently, unlike for traditional target machines, the pattern set is not fixed in this case but must be

[4] The algorithm performs string compression by replacing recurring substrings that appear earlier in the string with pointers, allowing the original string to be reconstructed by "copy-pasting."

determined as a precursor to pattern matching. This is known as the *ISE problem*, which appears when generating code for ASIPs where the processor can be partially customized for executing a particular program.

The intuition behind this design is to use echo instructions where code duplication is most prominent. To find these cases in a given function, Brisk et al. first enumerate all subgraphs from the block DAGs, and then match each subgraph over the block DAGs. Pattern matching is done using VF2, which is a generic subgraph isomorphism algorithm (see Chap. 2 on p. 40). Summing the sizes of the resulting match sets gives a measure of code duplication for each subgraph, but this value will be an overestimation as the match sets may contain overlapping matches. Brisk et al. addressed this by first solving the MIS problem on the conflict graph for each match set, and then adding up the sizes of *these* sets. After selecting the most beneficial subgraph, the covered nodes in the block DAGs are collapsed into single nodes to reflect the use of echo instructions. This process of matching and collapsing is then repeated until no new subgraph better than some user-defined value criterion can be found. Brisk et al. performed experiments on a prototype using a selected set of benchmark applications, which showed code size reductions of 25 % to 36 % on average.

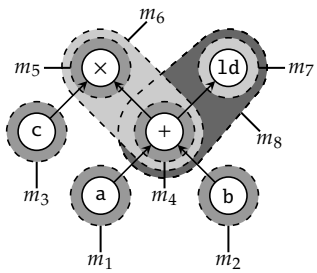
C.7 Modeling Instruction Selection as a Unate/Binate Covering Problem

Another approach to solving pattern selection is to translate it to a corresponding *unate* or *binate covering problem*. The concepts behind the two are identical with the exception of one detail, and both unate and binate covering can be used directly for covering graphs even though the designs discussed in this dissertation have only been applied on block DAGs.

Although binate covering-based techniques actually appeared first, we will begin with explaining unate covering, as binate covering is an extension of unate covering.

Unate Covering The idea of unate covering is to create a Boolean matrix M, where each row represents a node in the block DAG and each column represents a match covering one or more nodes in the block DAG. If we denote m_{ij} as row i and column j in M, then $m_{ij} = 1$ indicates that node i is covered by pattern j. Hence the pattern selection problem is equivalent to finding a set of columns in M such that the sum for every row is exactly 1. An example is given in Fig. C.7. Unate covering is an NP-complete problem, but as with the MIS and MWIS problems there exist several efficient techniques for solving it heuristically (see [87, 164] for an overview).

Unate covering alone, however, does not incorporate all necessary constraints of pattern selection when some patterns require – and prevent – the selection of other patterns in order to yield correct assembly code. Using machine grammars this can be enforced with the appropriate use of nonterminals, but for unate covering we



(a) Block DAG with matches.

node	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8
a	1	0	0	0	0	0	0	0
b	0	1	0	0	0	0	0	0
c	0	0	1	0	0	0	0	0
+	0	0	0	1	0	1	0	1
×	0	0	0	0	1	1	0	0
load	0	0	0	0	0	0	1	1

(b) Boolean matrix.

Figure C.7: Example of unate covering.

have no means of expressing this constraint. We therefore turn to binate covering, where this is possible.

Binate Covering We first rewrite the Boolean matrix from the unate covering problem into a Boolean CNF formula. If $\mathbf{x}_i \in \{0, 1\}$ represents a variable deciding whether match m_i is selected, then the Boolean matrix in Fig. C.7b can be rewritten as

$$\mathbf{x}_1 \wedge \mathbf{x}_2 \wedge \mathbf{x}_3 \wedge (\mathbf{x}_4 \vee \mathbf{x}_6 \vee \mathbf{x}_8) \wedge (\mathbf{x}_5 \vee \mathbf{x}_6) \wedge (\mathbf{x}_7 \vee \mathbf{x}_8).$$

Now, the difference between unate covering and binate covering is that all variables must be non-negated in the former, but may be negated in the latter. Hence binate covering is equivalent to SAT.

C.7.1 Applications

According to Liao et al. [251, 252] and Cong et al. [84], the pioneering use of binate covering to solve DAG covering was done by Rudell [322] in 1989 as a part of a *very large scale integration (VLSI)* synthesis design. Liao et al. [251, 252] later adapted it to instruction selection in a method that optimizes code size for one-register target machines. To prune the search space, Liao et al. perform pattern selection in two iterations. In the first iteration, patterns are selected such that the block DAG is covered but the costs of necessary data transfers are ignored. After this step the nodes covered by the same pattern are collapsed into single nodes, and a second binate covering problem is constructed to minimize the costs of data transfers. Although these two problems can be solved simultaneously, Liao et al. chose not to do so as the number of necessary implication clauses would become very large. Recently, Cong et al. [84] also applied binate covering as part of generating application-specific instructions for configurable processor architectures.

Unate covering was applied by Clark et al. [79] in generating assembly code for acyclic computation accelerators, which can be partially customized in order to increase performance of the currently executed function. Described in a paper

from 2006, the target machines are presumably homogeneous enough that implication clauses are unnecessary. The work by Clark et al. was later expanded by Hormati et al. [196] to reduce the number of interconnects as well as data-centered latencies in accelerator designs.

Martin et al. [266, 267] also applied unate covering to solve a similar problem concerning reconfigurable processor extensions. However, they combined instruction selection with instruction scheduling and solved both in tandem using constraint programming – we will discuss this approach later in this appendix – which they also applied for solving the pattern matching problem. Unlike in the cases of Clark et al. and Hormati et al., who solved their unate covering problems using heuristics, the assembly code generated by Martin et al. is potentially optimal.

C.8 Modeling Instruction Selection Using IP

It is well known that performing instruction selection, instruction scheduling, or register allocation in isolation will typically always yield suboptimal assembly code. But since each subproblem is already NP-complete on its own, attaining *integrated code generation* – where all these problems are solved simultaneously – is an even more difficult problem.

These challenges notwithstanding, Wilson et al. [368] introduced in 1994 what appears to be the first design that could be said to yield truly optimal assembly code. Wilson et al. accomplished this by using *IP*, which is a method for solving combinatorial optimization problems (sometimes IP is also called *integer linear programming*). In IP, a problem is expressed using sets of integer variables and linear equations, and a solution to an IP model is an assignment to all variables such that all equations are fulfilled (see Def. 2.1 on p. 31 for a formal definition). In general, solving an IP model is NP-complete, but extensive research in this field has made many problem instances tractable. For a comprehensive overview of IP, see in [370].

Modeling Pattern Selection Using Linear Inequality In their seminal paper, Wilson et al. describe that the pattern selection problem can be expressed as the following linear inequality:

$$\forall n \in N : \sum_{m \in M_n} \mathbf{x}_m \leq 1.$$

This reads: for every node n in the block DAG $\langle N, E \rangle$, at most one match m from the match set involving n (represented by M_n) may be selected.[5] The decision is represented by a $\mathbf{x}_m \in \{0, 1\}$ variable.

Similar linear equations can be formulated for modeling instruction scheduling and register allocation – which Wilson et al. also included in their model – but these

[5] The more common constraint is that *exactly one* match must be selected, but in the design by Wilson et al. nodes are allowed to become inactive and thus need not be covered.

are out of scope for this dissertation. In fact, any constraint that can be formulated in this way can be added to an existing IP model, making this approach a suitable code generation method for targeting irregular architectures. Furthermore, this is the first design we have seen that could potentially support interdependent instructions (although this was not the main focus of Wilson et al.).

Solving this monolithic IP model, however, typically requires considerably more time compared to the previously discussed techniques of instruction selection. But the trade-off for longer compilation time is better code quality; Wilson et al. reported that experiments showed that the generated assembly code was of comparable code quality to that of manually optimized assembly code. In theory, optimal assembly code can be generated, although this is in practice only feasible for small enough functions. Another much-valued feature is the ability to extend the model with additional constraints in order to support complicated target machines, which cannot be properly handled by the conventional designs as that typically violates assumptions made by the underlying heuristics.

C.8.1 Approaching Linear Solving Time with Horn Clauses

Although IP models are NP-complete to solve in general, it was discovered that for a certain class of problem instances – namely those based on Horn clauses – an optimal solution can be found in linear time [194]. A *Horn clause* is a disjunctive Boolean formula which contains at most one non-negated term. This can also be phrased as a logical statement that has at most one conclusion. For example, the statement

$$\text{if } \mathbf{x}_1 \text{ and } \mathbf{x}_2 \text{ then } \mathbf{x}_3$$

can be expressed as $\neg\mathbf{x}_1 \vee \neg\mathbf{x}_2 \vee \mathbf{x}_3$, which is a Horn clause, as only $\mathbf{x}_3$ is not negated. This can then easily be rewritten into the linear inequality

$$(1 - \mathbf{x}_1) + (1 - \mathbf{x}_2) + \mathbf{x}_3 \geq 1.$$

Moreover, statements that do not yield Horn clauses in their current form can often be rewritten so that they do. For example,

$$\text{if } \mathbf{x}_1 \text{ then } \mathbf{x}_2 \text{ and } \mathbf{x}_3$$

can be expressed as $\neg\mathbf{a} \vee \mathbf{b} \vee \mathbf{c}$ and is thus not a Horn clause because it has more than one non-negated term. But by rewriting it into

$$\begin{gathered}\text{if } \mathbf{x}_1 \text{ then } \mathbf{x}_2 \\ \text{if } \mathbf{x}_1 \text{ then } \mathbf{x}_3\end{gathered}$$

the statement can now be expressed as $\neg\mathbf{x}_1 \vee \mathbf{x}_2$ and $\neg\mathbf{x}_1 \vee \mathbf{x}_3$, which are two valid Horn clauses.

Gebotys [155] exploited this property in 1997 by developing an IP model for TMS320C2x – a common DSP at the time – where many of the constraints imposed

by the target architecture, instruction selection, and register allocation, and a part of the instruction scheduling problem, are expressed as Horn clauses. Using only Horn clauses may require a larger number of constraints than are otherwise needed, but Gebotys claims that the number is still manageable. When compared against a then-contemporary industrial DSP compiler, Gebotys demonstrated that an implementation based on IP yielded a performance improvement mean of 44 % for a select set of functions, while attaining reasonable compilation times. However, the solving time increased by orders of magnitude when Gebotys augmented the IP model with the complete set of constraints for instruction scheduling, which cannot be expressed entirely as Horn clauses.

C.8.2 IP-Based Designs with Multi-output Instruction Support

Leupers and Marwedel [245, 249] expanded the work of Wilson et al. – whose design is restricted to pattern trees – by developing an IP-based instruction selector which also supports multi-output instructions. In a paper from :1996, Leupers and Marwedel describe a scheme where the pattern DAGs of multi-output instructions – Leupers and Marwedel refer to these as complex patterns – are first decomposed into multiple pattern trees according to their RTs. RTs are akin to Fraser's RTLs [139] (see Ap. A on p. 146), and essentially mean that each observable effect gets its own pattern tree. Each individual RT may in turn correspond to one or more instructions, but unlike in Fraser's design this is not strictly required.

Assuming the block DAG has already been undagged, each expression tree is first optimally covered using IBURG. The RTs are expressed as rules in an machine grammar that has been automatically generated from a machine description written in MIMOLA (we will come back to this in Sect. C.10.3). Once RTs have been selected, the expression tree is reduced to a tree of super nodes, where each super node represents a set of nodes covered by some RT that have been collapsed into a single node. Since multi-output and disjoint-output instructions implement more than one RT, the goal is now to cover the super node graph using the patterns which are formed when the instructions are modeled as RTs. Leupers and Marwedel addressed this problem by applying a modified version of the IP model by Wilson et al.

But because the step of selecting RTs to cover the expression tree is separate from the step which implements them with instructions, the generated assembly code is not necessarily optimal for the whole expression tree. To achieve this property, the covering of RTs and selection of instructions must be done in tandem.

C.8.3 IP-Based Designs with Disjoint-output Instruction Support

Leupers [243] later made a more direct extension of the IP model by Wilson et al. in order to support SIMD instructions, which belong to the class of disjoint-output instructions. Described in a paper from 2000, Leupers's design assumes every SIMD instruction performs two operations, each of which takes a disjoint set of input

operands. This is collectively called a *SIMD pair*, and Leupers then extended the IP model with linear equations for combining SIMD pairs into SIMD instructions and defined the objective function so as to maximize the use of SIMD instructions.

In the paper, Leupers reports experiments where the use of SIMD instructions reduced code size by up to 75 % for the selected test cases and target machines. But since this technique assumes that each individual operation of the SIMD instructions is expressed as a single node in the block DAG, it is unclear whether the method can be extended to more complex SIMD instructions and whether it scales to larger functions. Tanaka et al. [348] later expanded Leupers's work for selecting SIMD instructions while also taking data copying into account by introducing auxiliary transfer nodes and transfer patterns into the block DAG.

C.8.4 Modeling the Pattern Matching Problem with IP

In 2006, Bednarski and Kessler [38] developed an integrated code generation design where both pattern matching and pattern selection are solved using integer programming. The scheme – which later was applied by Eriksson et al. [119], and is also described in an article by Eriksson and Kessler [120] – is an extension of their earlier work where instruction selection had previously more or less been ignored (see [212, 213]).

In broad outline, the IP model assumes that a sufficient number of matches has been generated for a given block DAG G. This is done using a pattern matching heuristic that computes an upper bound. For each match m, the IP model contains integer variables that:

- map a pattern node in m to a node in G;
- map a pattern edge in m to an edge in G; and
- decide whether m is used in the solution. Remember that we may have an excess of matches, so they cannot all be selected.

Hence, in addition to the typical linear equations we have seen previously for enforcing coverage, this IP model also includes equations to ensure that the selected matches are valid matches.

Implemented in a framework called *Optimist*, Bednarski and Kessler compared their IP model against another integrated code generation design based on dynamic programming. This DP algorithm, however, which was developed by the same authors (see [212]), has nothing to do with the conventional DP algorithm by Aho et al. [8]). Bednarski and Kessler found that Optimist substantially reduced compilation time while retaining code quality. However, for several test cases – the largest block DAG containing only 33 nodes – Optimist failed to generate any assembly code whatsoever within the set time limit. One reasonable cause could be that the IP model also attempts to solve pattern matching – a problem which we have seen can be solved externally – and thus further exacerbates an already computationally difficult problem.

C.9 Modeling Instruction Selection Using CP

Although integer programming allows auxiliary constraints to be included into the IP model, they may be cumbersome to express as linear equations. This issue can be alleviated by using *CP*, which is another method for solving combinatorial optimization problems but has more flexible modeling capabilities compared to IP. For a brief introduction to CP, see Chap. 3.

First Application In 1990, Bashford and Leupers [36] pioneered the use of CP in code generation by developing a constraint model for integrated code generation that targets DSPs with highly irregular architectures (the work is also discussed in [243, 244]). Like Leupers and Marwedel's IP-based design, Bashford and Leupers's first breaks down the instruction set of the target machine into a set of *RTs* which are used to cover individual nodes in the block DAG. As each RT concerns specific registers on the target machine, the covering problem essentially also incorporates register allocation. The goal is then to minimize the cost of covering by combining multiple RTs that can be executed in parallel as part of some instruction.

For each node in the block DAG a FRT is introduced, which basically embodies all RTs that match a particular node and is formally defined as the following tuple:

$$\left\langle op, \mathbf{d}, [\mathbf{u}_1, \ldots, \mathbf{u}_n], \mathbf{f}, \mathbf{c}, \mathbf{t}, CS \right\rangle .$$

op is the operation of the node. $\mathbf{d}$ and $\mathbf{u}_1, \ldots, \mathbf{u}_n$ are variables representing the storage locations of the result and the respective inputs to the operation. These are typically the registers that can be used for the operation, but also include *virtual storage locations* which convey that the value is produced as an intermediate result in a chain of operations (for example, the multiplication term in a multiply-accumulate instruction is such a result). Then, for every pair of operations that are adjacent in the block DAG, a set of constraints is added to ensure that there exists a valid data transfer between the storage locations of $\mathbf{d}$ and $\mathbf{u}_i$ if these are assigned to different registers. In addition, if either of $\mathbf{d}$ and $\mathbf{u}_i$ resides in a virtual storage location, then both must be identical. $\mathbf{f}$, $\mathbf{c}$, and $\mathbf{t}$ are all variables which collectively represent the *extended resource information (ERI)*. The ERI specifies at which functional unit the operation will be executed, at what cost in terms of number of execution cycles, and by which instruction type. A combination of a functional unit and an instruction type is later mapped to a particular instruction. Multiple RTs can be combined into the same instruction when the destination of the result is a virtual storage location by setting $\mathbf{c} = 0$ and letting the last node in the operation chain account for the required number of execution cycles. The last entity, *CS*, is the set of constraints for defining the range of values for the variables and the dependencies between $\mathbf{d}$ and $\mathbf{u}_i$, as well as other auxiliary constraints that may be required for the target machine. For example, if the set of RTs matching a node consists of $\{\mathtt{r_c} = \mathtt{r_a} + \mathtt{r_b}, \mathtt{r_a} = \mathtt{r_c} + \mathtt{r_b}\}$, then the corresponding *FRT* becomes

$$\langle +, \mathbf{d}, [\mathbf{u}_1, \mathbf{u}_2], \mathbf{f}, \mathbf{c}, \mathbf{t}, \{\mathbf{d} \in \{\mathtt{r_c}, \mathtt{r_a}\}, \mathbf{u}_1 \in \{\mathtt{r_a}, \mathtt{r_c}\}, \mathbf{u}_2 = \mathtt{r_b}, \mathbf{d} = \mathtt{r_c} \Rightarrow \mathbf{u}_1 = \mathtt{r_a}\}\rangle.$$

For brevity, we omit several details such as the constraints concerning the ERI.

This constraint model is then solved to optimality using a constraint solver. But since optimal covering using FRTs is NP-complete, Bashford and Leupers applied heuristics to curb the complexity by splitting the block DAG into smaller pieces along edges where intermediate results are shared. Once split, instruction selection is then performed on each expression tree in isolation.

C.9.1 Taking Advantage of Global Constraints

So far we have discussed several techniques that apply constraint programming for solving the problems of pattern matching and pattern selection – namely those by Bashford and Leupers and Martin et al. Recently, Beg [40] introduced another constraint model for instruction selection as well as new methods for improving solving. For example, in order to reduce the search space, Beg applied conventional DP-based techniques to compute an upper bound on the cost. However, the constraint model mainly deals with the problem of pattern matching rather than pattern selection. Moreover, Beg noticed only a negligible improvement (less than 1 %) in code quality compared to LLVM, mainly because the target machines (MIPS and ARM) were simple enough that greedy heuristics generate near-optimal assembly code. In addition, the block DAGs of the benchmark functions were fairly tree-shaped [355], for which optimal code can be generated in linear time. In any case, none of these designs take advantage of a key feature of constraint programming, which is the use of global constraints. A *global constraint* captures relations among multiple variables and results in more search space pruning than if it had been expressed using a decomposition of constraints.

Hence, when Floch et al. [135] in 2010 adapted the constraint model by Martin et al. to support processors with reconfigurable cell fabric, they replaced the method of pattern selection with constraints that are radically different from those incurred by unate covering. In addition, unlike in the case of Bashford and Leupers, the design by Floch et al. applies the more direct form of pattern matching instead of first breaking down the patterns into RTs and then selecting instructions that combine as many RTs as possible.

Modeling Pattern Selection Using Global Cardinality Constraint As described in their 2010 paper, Floch et al. use the global cardinality constraint to enforce the requirement that every node in the block DAG must be covered by exactly one match.[6] The constraint, referred to as GCC, constrains the number of variables assigned a particular value (which may also be a variable). Given a set $v_1, \ldots, v_k$ of values and two sets $\mathbf{x}_1, \ldots, \mathbf{x}_n$ and $\mathbf{c}_1, \ldots, \mathbf{c}_k$ of variables, the constraint holds if, for each $i = 1, \ldots, k$, exactly $\mathbf{c}_i$ variables in the set $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are assigned value v_i (see also Def. 3.2 on p. 47). For example, $\text{GCC}(\langle 5, \mathbf{c}_1 = 0\rangle, \langle 3, \mathbf{c}_2 = 1\rangle, \mathbf{x}_1 = 2, \mathbf{x}_2 = 3)$ holds

[6] It is also possible to enforce pattern selection through a global set covering constraint developed by Mouthuy et al. [276], but no implementation is known to do so.

because no **x** variable is assigned value 5 and exactly one **x** variable is assigned value 3. Similarly, GCC($\langle 3, \mathbf{c}_1 \in \{0, 2\}\rangle, \mathbf{x}_1 = 2, \mathbf{x}_2 = 3$) does not holds because either none or both **x** variables must be assigned value 3.

To model pattern selection using GCC, two new sets of variables are needed. Assume that N denotes the set of nodes to be covered, M denotes the match set, and $covers(m)$ denotes the set of nodes covered by match m. Then, variable $\mathbf{match}_n \in \{m \mid m \in M, n \in covers(m)\}$ decides which match covers node n, and variable $\mathbf{count}_m \in \{0, |covers(m)|\}$ decides how many nodes are covered by match m. Hence each match covers either no nodes or all nodes in its pattern. With these variables, pattern selection can be modeled as

$$\text{GCC}(\cup_{m \in M} \langle m, \mathbf{count}_m \rangle, \cup_{n \in N} \mathbf{match}_n),$$

which offers stronger propagation than the corresponding linear inequality constraint and thus reduces solving time [135].

Accommodating VLIW Architectures The constraint model by Floch et al. was also further extended by Arslan and Kuchcinski [25, 26, 27] to accommodate VLIW architectures and disjoint-output instructions. First, every disjoint-output instructions is split into multiple *subinstructions*, each modeled by a disjoint pattern which is mapped onto the block DAG using a generic subgraph isomorphism algorithm. Pattern selection is then modeled as an instance of the constraint model with the additional constraints to schedule the subinstructions such that they can be replaced by the original disjoint-output instruction. Consequently, unlike previous techniques that recombine partial matches into complex matches prior to pattern selection (see for example Scharwaechter et al. [329], Ahn et al. [3], Arnold and Corporaal [22, 23, 24]), Arslan and Kuchcinski instead solve these two problems in tandem. Their design is also capable of accepting multiple, disconnected block DAGs as a single input.

Limitations An inherent limitation to the constraint models applied by Martin et al., Floch et al., and Arslan and Kuchcinski is that they do not model the necessary data transfers between different register classes. This in turn means that the cost model is only accurate for target machines equipped with a homogeneous register architecture, which could compromise code quality for more complicated target machines.

C.10 Other DAG-Based Approaches

C.10.1 More Genetic Algorithms

Seemingly independently from the earlier work by Shu et al. [337] (discussed in Ap. B on p. 199), Lorenz et al. [256, 257] introduced in 2001 another technique where genetic algorithms are applied to code generation. But unlike the design by Shu

et al., the one by Lorenz et al. takes block DAGs instead of trees as input and also incorporates instruction scheduling and register allocation. Lorenz et al. recognized that contemporary compilers struggled with generating efficient assembly code for DSPs equipped with very few registers and typically always spill the results of common subexpressions to memory and reload them when needed. Compared to optimal assembly code, this may incur more memory accesses than needed.

The design by Lorenz et al. is basically an iterative process. First, the operations within a block are scheduled using *list scheduling*, which is a traditional method of scheduling (see for example [315]). For every scheduled operation, a gene is formulated to encode all the possible decisions to take in order to solve the problems of instruction selection and register allocation. These decisions are then taken over multiple steps using standard GA operations, where the values are selected probabilistically. In each step the gene is mutated and crossed over in order to produce new, hopefully better genes, and a fitness function is applied to evaluate each gene in terms of expected execution time. After a certain number of generations, the process stops and the best gene is selected. Certain steps are also followed by a routine based on constraint programming that prunes the search space for the subsequent decisions by removing values which will never appear in any valid gene. Although every gene represents a single node in the block DAG, complex patterns can still be supported through an additional variable for selecting the instruction type for the node. If nodes with the same instruction type have been scheduled to be executed on the same cycle, then they can be implemented using the same instruction during assembly code emission.

Lorenz et al. originally developed this technique in order to reduce power usage of assembly code generated for constrained DSPs, and later extended the design to also incorporate instruction compaction and address generation. Experiments indicate that the technique for a selected set of test cases resulted in energy savings of 18 % to 36 % compared to a traditional tree covering-based compiler, and reduced execution time by up to 51 %. According to Lorenz et al., the major contribution to this reduction is due to improved usage of registers for common subexpression values, which in turn leads to less use of power-hungry and long-executing memory operations. But due to the probabilistic nature of GA, optimality cannot be guaranteed, making it unclear how this technique would fare against other DAG covering-based designs which allow a more exhaustive exploration of the search space.

C.10.2 Extending Trellis Diagrams to DAGs

In 1998, Hanono and Devadas [173, 174] proposed a technique that is similar to Wess's use of trellis diagrams, which we discussed in Ap. B on p. 200. Implemented in a system called *Aviv*, Hanono and Devadas's instruction selector takes a block DAG as input and duplicates each operation node according to the number of functional units in the target machine on which that operation can run. Special *split* and *transfer nodes* are inserted before and after each duplicated operation node

to allows the data flow to diverge and then reconverge before passing to the next operation node in the block DAG. The use of transfer nodes also allow the cost of transferring data from one functional unit to another to be taken into account. Similarly to the trellis diagram, instruction selection is thus transformed to finding a path from the leaf nodes in the block DAG to its root node. But differently from the optimal, DP-oriented design of Wess, Hanono and Devadas applied a greedy heuristic that starts from the root node and makes it way towards the leaves.

Unfortunately, as in Wess's design, this technique assumes a 1-to-1 mapping between the nodes in the block DAG and the instructions in order to generate efficient assembly code. In fact, the main purpose behind Aviv was to generate efficient assembly code for VLIW architectures, where the focus is on executing as many instructions as possible in parallel.

C.10.3 Hardware Modeling Techniques

In 1984, Marwedel [269] developed a retargetable system called *MSS* for microcode generation,[7] where a machine description written in *Machine Independent Microprogramming Language (MIMOLA)* [382] is used for modeling the entire data path of the processor, instead of just the instruction set as we have commonly seen. This is commonly used for DSPs where the processor is small but highly irregular. Although MSS consists of several tools, we will concentrate on the compiler – *MSSQ* – as its purpose is most aligned with instruction selection. MSSQ was developed by Leupers and Marwedel [246] as a faster version of *MSSC* [288], which in turn is an extension of the tree-based *MSSV* [270].

The MIMOLA specification contains the processor registers as well as all the operations that can be performed on these registers within a single cycle. From this specification, a hardware DAG called the *connection-operation (CO) graph* is automatically derived. An example is given in Fig. C.8. A pattern matcher then attempts to find subgraphs within the CO graph to cover the expression trees. Because the CO graph contains explicit nodes for every register, a match found on this graph – called a *version* – is also an assignment of function variables (and temporaries) to registers. If a match cannot be found (due to a lack of registers), the expression tree will be rewritten by splitting assignments and inserting additional temporaries. The process then backtracks and repeats in a recursive fashion until the entire expression tree is covered. A subsequent process then selects a specific version from each match set and tries to schedule them so that they can be combined into bundles for parallel execution.

Although microcode generation is at a lower hardware level than assembly code generation – which is usually what we refer to with instruction selection – we see several similarities between the problems that must be solved in each. For this reason it is included in this survey, and further examples include [33, 233, 262]. In

[7] *Microcode* is essentially the hardware language that processors use internally for executing instructions. For example, microcode controls how the registers and program counter should be updated for a given instruction.

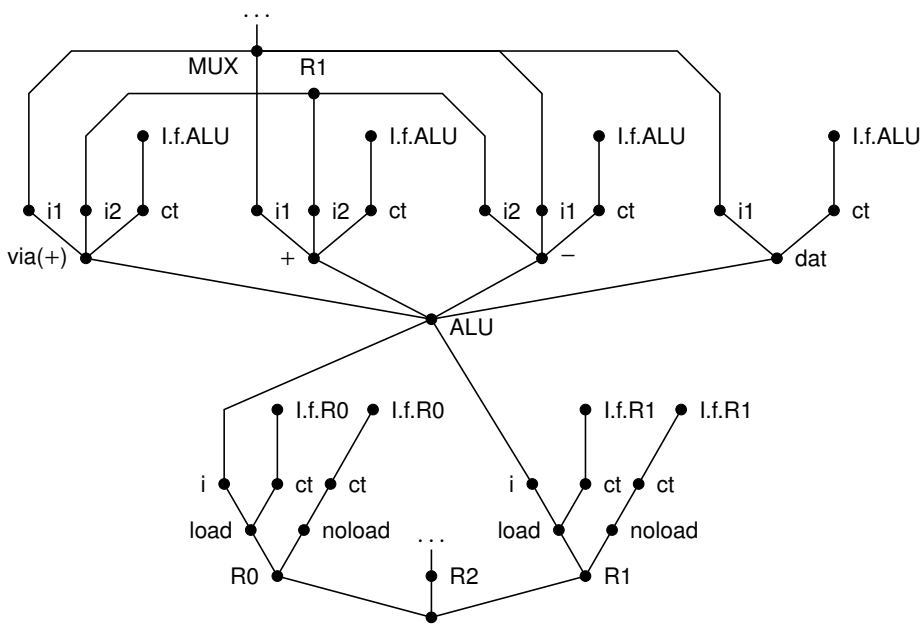


Figure C.8: Example of a CO graph for a simple processor [288], containing an arithmetic logic unit, two data registers, a program counter, and a control store.

Ap. D, we will see another design that also models the entire processor but applies a more powerful technique.

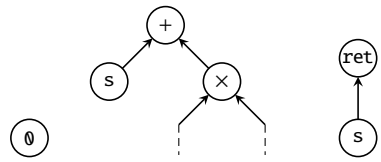
C.11 Limitations of DAG Covering

Although DAG coverings addresses the issue of whether to split or duplicate common subexpressions within a block, the problem still remains for expressions that are spread across multiple blocks. To fully address this problem, one must resort to graph covering.

This also applies to other situations where decisions made for one block can inhibit subsequent decisions for other blocks, such as enforcing specific storage locations or value modes. For example, Fig. C.9 shows a function that multiplies the elements of two arrays and sums the results. Assume that the arrays consist of fixed-point values. For efficiency, a common idiosyncrasy in many DSPs is that multiplication of two fixed-point values return a value that is shifted one bit to the left. For such target machines, both the value `0` and the accumulator variable `s` should be in shifted mode throughout the entire function, and only restored into normal mode upon return. Otherwise the accumulated value would be needlessly shifted back and forth within the loop. Achieving this, however, is difficult when limited to covering only a single block DAG at a time. Assume for example that

```
int f(int* A, int* B, int N) {
  int s = 0;
  for (int i = 0; i < N; i++) {
    s = s + A[i] * B[i];
  }
  return s;
}
```

(a) C code.



(b) Block graphs involving variable `s`. For brevity, the subtrees concerning `A[i]` and `B[i]` are not included.

rules	
$Reg \rightarrow$ `const`	$SReg \rightarrow \times\ Reg\ Reg$
$SReg \rightarrow$ `const`	$Null \rightarrow$ `ret` Reg
$Reg \rightarrow +\ Reg\ Reg$	$Reg \rightarrow SReg \quad (r \ll 1)$
$SReg \rightarrow +\ SReg\ SReg$	$SReg \rightarrow Reg \quad (r \gg 1)$

(c) Rules. For brevity, the actions are not included. *Null* is a dummy nonterminal since `ret` does not return anything, yet all productions must have a result. All rules are assumed to have equal cost.

Figure C.9: Example illustrating the limitation of block DAGs.

the function had no multiplication. In that case, deciding to load value `0` in shifted mode would instead lower code quality as the value would needlessly have to be shifted back before returning, which takes an extra instruction.

Lastly, most of these approaches are restricted to tree-shaped patterns, meaning they only support single-output instructions. Many instruction sets, however, contain multi-output instructions and require DAG-shaped patterns, which violate underlying assumptions made by many of the aforementioned approaches.

C.12 Summary

In this appendix, we have investigated several methods that rely on the principle of DAG covering, which is a more general form of tree covering. Operating on DAGs instead of trees has several advantages. Most importantly, common subexpressions can be directly modeled, and a larger set of instructions – including multi-output and disjoint-output instructions – can be supported and exploited during instruction selection. This in turn leads to improved performance and reduced code size. Consequently, techniques based on DAG covering are today one of the most widely applied methods for instruction selection in modern compilers.

The ultimate cost of transitioning from trees to DAGs, however, is that optimal pattern selection can no longer be achieved in linear time as it is NP-complete. At the same time, DAGs are not expressive enough to allow the proper modeling of

all aspects featured in the functions and instructions. For example, statements such as for loops incur loops in the graph representing the function, restricting DAG covering to the scope of blocks and excluding the modeling of inter-block instructions. Another disadvantage is that optimization opportunities for storing function variables and temporaries in different forms and at different locations across the function are forfeited.

In the next appendix, we will discuss the last and most general principle of instruction selection, which addresses some of the aforementioned deficiencies of DAG covering.

APPENDIX

D

Graph Covering

This appendix considers techniques based on graph covering. First, we introduce the principle in Sect. D.1 and describe representations that enable this principle in Sect. D.2. In Sect. D.3 we describe techniques that extend methods from tree covering to graphs. In Sect. D.4 we describe techniques that model instruction selection as a PBQP. Other graph-based approaches that do not fit into any of the sections above are discussed in Sect. D.5. Lastly, we summarize in Sect. D.6.

The appendix is based on material presented in [186, Chap. 5] that has been adapted for this dissertation. To not disturb the flow of reading, material already presented in Chap. 2 is duplicated in this appendix.

D.1 The Principle

In DAG covering-based instruction selection, functions can only be modeled one block at a time as cycles are forbidden to appear in the block DAGs. Lifting this restriction results in a *function graphs*, thus capturing the data flow for an entire function as a single graph. Depending on the representation, some function graphs also capture the control flow for the function.

Selecting instructions for such graphs is known as *global instruction selection* and has several advantages over local instruction selector. First, with an entire function as input, a global instruction selector can account for the effects of local pattern selection across the block boundaries and is thereby better informed when making its decisions. In addition, it enables global code motion. Second, to support inter-block instructions – which require modeling of both data and control-flow information – it is imperative that the patterns be expressible using graphs that may contain cycles. This makes graph covering one of the key approaches for making use of fewer but more efficient instructions, which is becoming more and more crucial for modern target machines – especially embedded systems – where both power consumption and heat emission are becoming increasingly important factors.

representation	pattern matching	optimal pattern selection
trees	linear	linear
DAGs	NP-complete	NP-complete
graphs	NP-complete	NP-complete

Table D.1: Time complexities for solving the pattern matching and optimal pattern selection problems using various representations.

However, when transitioning from pattern DAGs to pattern graphs, we can no longer apply pattern matching techniques designed for trees and DAGs. Instead we must resort to methods from the field of subgraph isomorphism in solving this problem (see Tab. D.1 for time complexity comparison). The pattern selection problem, on the other hand, can still be solved using many of the techniques which were discussed in Ap. C. Therefore, in this appendix we will only examine techniques that were originally designated for graph covering.

D.2 Sea-of-Nodes IRs

With and DAG covering, it is sufficient to represent the function on block level. Consequently, functions are typically modeled as a forest of expression trees or a set of block DAGs. But, as previously stated, this becomes an impediment when applied in graph covering-based techniques, forcing us to instead use a graph-based intermediate representation. We will therefore continue by looking briefly at how functions can be expressed using such representations, which are colloquially referred to as *sea-of-nodes IRs*.

In the context of instruction selection, there are two sea-of-nodes IRs that are of interest. The first captures the data flow for entire functions, and the second is an extension of the first in order to also capture control flow. For a comprehensive survey on function representations, see [345].

Capturing Data Flow of Entire Functions In order to simply many compiler tasks, Cytron et al. [91] introduced a function representation called *SSA form*.

A program is said to be in SSA form if every variable is defined exactly once. One of the main benefits of this is that the live range of each variable is contiguous. The live range of a variable can be loosely described as the length within the program where the value of that variable must not be destroyed. This in turn means that each variable corresponds to a single value, which simplifies many program optimization routines.

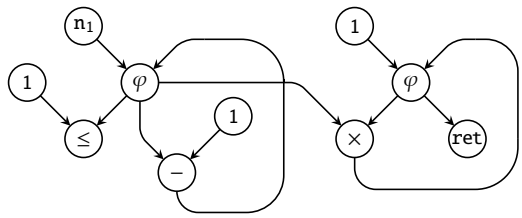
For example, the function shown in Fig. D.1a is not in SSA form as variables `f` and `n` are redefined within the loop. By introducing new variables and connecting these using *φ-functions* where the value depends on control flow, the function can be rewritten into SSA form, as shown in Fig. D.1b.

```
int factorial(int n) {
  entry:
    int f = 1;
  head:
    if (n <= 1) goto end;
  body:
    f = f * n;
    n = n - 1;
    goto head;
  end:
    return f;
}
```

(a) C implementation of factorial.

```
int factorial(int n1) {
  entry:
    int f1 = 1;
  head:
    int f2 = φ(f1:entry, f3:body);
    int n2 = φ(n1:entry, n3:body);
    if (n2 <= 1) goto end;
  body:
    int f3 = f2 * n2;
    int n3 = n2 - 1;
    goto head;
  end:
    return f2;
}
```

(b) Code in SSA form.



(c) SSA graph.

Figure D.1: Example of an SSA graph.

From an SSA-based function, we can construct a data-flow graph called the *SSA graph* [159]. Like in data-flow graphs, each operation in the function (including the φ-functions) is represented as a node. These nodes are connected using data-flow edges, ignoring the fact that the operations may belong to different blocks. For the example above, this results in the SSA graph shown in Fig. D.1c, thus giving a complete view of the data flow in the function.

But since the SSA graph is devoid of any control-flow information, it is often used as a supplement alongside one or more other IRs. Obviously this also prevents selection of instructions for implementing branches and procedure calls.

Capturing Both Data And Control Flow Click and Paleczny [81] introduced a sea-of-nodes IR that captures both data and control flow. The data flow is modeled exactly as in the SSA graph, and the control flow is captured using nodes to represent the blocks in the function and edges to represent jumps between blocks. To capture dependencies between the data and control flow – for example, when the target of a jump depends on a Boolean value – such jumps flow through special *if* nodes. For lack of a better name we will call this the *Click-Paleczny graph*, and an example is shown in Fig. D.2.

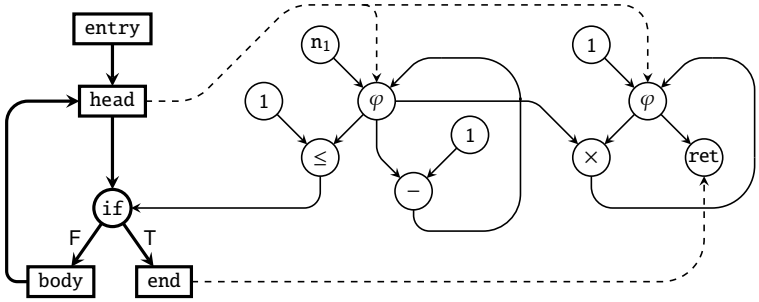


Figure D.2: Example of a Click-Paleczny graph, corresponding to the program shown in Fig. D.1. Thin-lined nodes and edges denote data operations and data flow. Thick-lined nodes and edges denote control operations and control flow. Dashed edges indicate to which block an operation belongs.

D.3 Extending Tree Covering Techniques to Graphs

Paleczny et al. [295] introduced an approach for performing instruction selection based on the Click-Paleczny graph.

Implemented in the *Java Hotspot Server Compiler (JHSC)*, the approach first divides the function graph into a set of possibly overlapping expression trees. This is done by labeling certain nodes in the function graph as tree roots. Root candidates are nodes representing operations whose result are shared or operations with side effects and may therefore not be duplicated. The selection of roots is geared towards duplicating address computations and other expressions that can be subsumed into a single instruction. Once labeled, each expression tree is covered using a variant of Alg. B.7 (see Ap. B on p. 191). The instructions are then emitted and placed in blocks using a global code motion method described in [82].

Although the function is represented as a function graph, the instructions must still be modeled as pattern trees. Consequently, only single-output instructions can be selected using this approach.

D.4 Modeling Instruction Selection as a PBQP

In 2003, Eckstein et al. [109] recognized that limiting instruction selection to local scope can decrease code quality of assembly code generated for fixed-point arithmetic DSPs. For more details, see Ap. C on p. 230.

PBQP To overcome this problem, Eckstein et al. developed a design that takes SSA graphs as input – making this technique the first to do so – and transforms the pattern selection problem into a *partitioned Boolean quadratic problem (PBQP)*. First introduced by Scholz and Eckstein [330] to model and solve register allocation,

PBQP is a variant of the *QAP*, which is a fundamental combinatorial optimization problem in the field of operations research (see [254] for a survey). Although both problems are NP-complete in general, a subclass of PBQP can be solved in linear time which inspired Scholz and Eckstein in developing a greedy, linear-time solver.

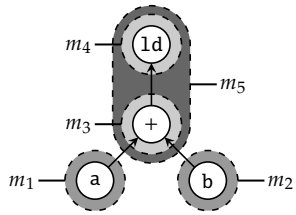
A formal definition of PBQP is given in Chap. 2 on p. 38, which can intuitively be explained as follows. Assume a problem consists of n decisions, each with k choices. Then $\vec{\mathbf{x}}_i$ is a decision variable with k elements, where $\vec{\mathbf{x}}_i[j] = 1$ means that choice j has been selected for decision i. For each variable, the condition $\vec{1}^\intercal \vec{\mathbf{x}}_i = 1$ ensures that exactly one choice is selected. The cost of selecting a particular choice for decision i is represented through a cost vector $\vec{c}_i$, and the cost of combining two decisions i and j are represented through a cost matrix C_{ij}.

In Context of Instruction Selection In this context, $\vec{\mathbf{x}}_i$ decides whether to select a particular match to cover node i, $\vec{c}_i$ contains the cost for each such match, and C_{ij} contains the cost of additional instructions that may need to be selected due to certain combinations of matches. For example, assume two nodes i and j where j depends on i. Assume further that the instructions are represented as a normal-form machine grammar, and that i and j can be covered using two rules, r_i and r_j, with productions $A \longrightarrow \mathtt{op}_i\ A\ A$ and $B \longrightarrow \mathtt{op}_j\ B\ B$, respectively. Since the result of r_i does not match the operands of r_j, this rule combination requires a chain rule – or a chain of these, if necessary – that derives B from A. The C_{ij} costs are calculated by computing the transitive closure for all chain rules. For this Eckstein et al. seem to have used the Floyd-Warshall algorithm [136], and Schäfer and Scholz [328] later discovered a method that finds the optimal sequence of chain rules. Illegal combinations are prevented by assigning infinite cost. An example of a PBQP instance is shown in Fig. D.3.

Using a prototype implementation, Eckstein et al. ran experiments on a selected set of fixed-point programs exhibiting the behavior discussed earlier. The results indicate that their scheme improved performance by 40–60 % on average – and at most 82 % for one function – compared to traditional tree covering-based instruction selectors. According to Eckstein et al., this considerable gain in performance comes from more efficient use of value modes to which tree covering-based techniques must make premature assignments, and thus could have a detrimental effect on code quality. For example, if chosen poorly, the instruction selector may need to emit additional instructions in order to undo decisions regarding value modes, which obviously reduces performance and needlessly increases the code size. Although the technique by Eckstein et al. clearly mitigates these concerns, their design also has limitations of its own. Most importantly, their PBQP model can only support pattern trees and consequently hinders exploitation of many common target machine features, such as multi-output instructions.

rules	cost
$Reg \rightarrow$ `var`	0
$Reg \rightarrow$ `+` $Reg\ Reg$	1
$Reg \rightarrow$ `load` $Addr$	3
$Reg \rightarrow$ `load` `+` $Reg\ Reg$	5
$Addr \rightarrow Reg$	2

(a) Rules. For brevity, the actions are not included.



(b) SSA graph.

$$\vec{\mathbf{x}}_{\mathtt{a}} \in \{0,1\} \qquad \vec{c}_{\mathtt{a}} = \begin{bmatrix} 0 \end{bmatrix} \begin{matrix} m_1 \end{matrix}$$

$$\vec{\mathbf{x}}_{\mathtt{b}} \in \{0,1\} \qquad \vec{c}_{\mathtt{b}} = \begin{bmatrix} 0 \end{bmatrix} \begin{matrix} m_2 \end{matrix}$$

$$\vec{\mathbf{x}}_{+} \in \{0,1\}^2 \qquad \vec{c}_{+} = \begin{bmatrix} 1 \\ 5 \end{bmatrix} \begin{matrix} m_3 \\ m_5 \end{matrix}$$

$$\vec{\mathbf{x}}_{\mathtt{load}} \in \{0,1\}^2 \qquad \vec{c}_{\mathtt{load}} = \begin{bmatrix} 3 \\ 5 \end{bmatrix} \begin{matrix} m_4 \\ m_5 \end{matrix}$$

$$C_{\mathtt{a}+} = \begin{matrix} m_3 & m_5 \end{matrix} \begin{bmatrix} 0 & 0 \end{bmatrix} \begin{matrix} m_1 \end{matrix}$$

$$C_{\mathtt{b}+} = \begin{matrix} m_3 & m_5 \end{matrix} \begin{bmatrix} 0 & 0 \end{bmatrix} \begin{matrix} m_2 \end{matrix}$$

$$C_{+\mathtt{load}} = \begin{matrix} m_4 & m_5 \end{matrix} \begin{bmatrix} 2 & \infty \\ \infty & 0 \end{bmatrix} \begin{matrix} m_3 \\ m_5 \end{matrix}$$

(c) PBQP instance. The rows and columns in the cost vectors and matrices are labeled with the matches they represent. Cost matrices for uninteresting combinations are assumed to consist of 0s.

Figure D.3: Example of modeling instruction selection as a PBQP.

D.4.1 Handling DAG-shaped Patterns

To handle DAG-shaped patterns, the PBQP model must be extended. First assume an extended grammar where multi-output instructions are described using complex rules (described in Ap. C on p. 217, see also Fig. 2.7). For each combination of matches derived from proxy rules that can be combined into an instance of a complex rule, a *complex match* is created. Each complex match i in turn introduces a variable $\vec{\mathbf{x}}_i \in \{0,1\}^2$ to decide whether i is selected. Because of the $\vec{1}^{\mathsf{T}}\vec{\mathbf{x}}_i = 1$ condition, every such variable has exactly two elements (one representing *on* and the other *off*). Like with the simple rules, the costs of selecting a complex rule and interactions between these – for example, two complex matches are not allowed to overlap or cause cyclic data dependencies – are represented through the cost vectors and matrices.

In order to select a complex rule, all of its proxy rules must also be selected. This is achieved by first extending, for each node i, the domain of its variable $\vec{\mathbf{x}}_i$ with matches derived from proxy rules. Then a new set of cost matrices D_{ij} is created such that, for a node i and complex match j, the costs are 0 if $\vec{\mathbf{x}}_j = \mathit{off}$ or $\vec{\mathbf{x}}_i$ is set to a proxy rule associated with j. Otherwise the costs are ∞. Consequently, if a complex match covering some node n is selected, then the only choice for $\vec{\mathbf{x}}_n$ with non-infinite cost is an associated proxy rule. The PBQP model is thus augmented

with another sum

$$\sum_{i \in N,\, j \in M} \vec{\mathbf{x}}_i^{\mathsf{T}} D_{ij} \vec{\mathbf{x}}_j \tag{D.1}$$

where N denotes the set of nodes in the SSA graph and M denotes the set of complex matches.

This alone, however, allows solutions where all proxy rules but none of the complex rules are selected. This is resolved by assigning an artificially large cost K to the selection of proxy rules, which is offset when selecting the corresponding complex rule. For example, if a complex rule r with cost 2 consists of three proxy rules, then the new cost of selecting r is $2 - 3K$.

D.4.2 Using Rewrite Rules Instead of Grammar Rules

In 2010, Buchwald and Zwinkau [60] introduced another technique based on PBQPs. But unlike Eckstein et al. and Ebner et al., Buchwald and Zwinkau approached the task of instruction selection as a formal graph transformation problem, for which much previous work already exist. Hence, in Buchwald and Zwinkau's design the instructions are expressed as *rewrite rules* instead of grammar rules. As these rewrite rules are based on a formal foundation, the resulting instruction selector can be automatically verified to handle all possible programs. If this check fails, the verification tool can also provide the necessary rewrite rules that are currently missing from the instruction set.

The technique works as follows. First the SSA graph is converted into a DAG-like form by duplicating each ϕ-node into two nodes, which effectively breaks any cycles appearing in the SSA graph. After finding all applicable rewrite rules for this DAG (this is done using traditional pattern matching), a corresponding instance of the PBQP is formulated and solved as before.

Buchwald and Zwinkau also discovered and addressed flaws in the PBQP solver by Eckstein et al., which may fail to find a solution in certain situations due to inadequate propagation of information. However, Buchwald and Zwinkau also cautioned that their own implementation does not scale well when the number of overlapping patterns grows. In addition, since the SSA graph is devoid of control-flow information, none of the PBQP-based techniques can support inter-block instructions.

D.5 Other Graph-Based Approaches

D.5.1 More Hardware Modeling Techniques

In Ap. C we saw a technique for performing microcode generation where the entire processor of the target machine is modeled as a graph instead of by just deriving the patterns for the available instructions. Here we will look at a few techniques that rely on the same modeling scheme, but address the more traditional problem of instruction selection.

CHESS Lanneer et al. [234] developed in 1990 a design that was later adopted by Van Praet et al. [356, 357] in their implementation of *CHESS*, a well-known compiler targeting DSPs and ASIPs.

Comparing CHESS to MSSQ (see Ap. C on p. 229), we find two striking differences. First, in MSSQ the data paths of the processor are given by a manually written machine description, whereas CHESS derives these automatically from a specification written in nML [127, 128].

Second, the method of bundling – which is the task of scheduling operations for parallel execution – is different. The instruction selector in MSSQ uses techniques from DAG covering to find patterns in the hardware graph, which can subsequently be used to cover the expression trees. After pattern selection, another routine attempts to schedule the selected instructions for parallel execution. In contrast, CHESS takes a more incremental approach. From the program CHESS first constructs a *chaining graph*, where each node represents an operation in the program that has been annotated with a set of functional units capable of executing that operation. Since the functional units on a DSP are commonly grouped into *functional building blocks (FBBs)*, the chaining graph also contains an edge between every pair of nodes that could potentially be executed on the same FBB. A heuristic algorithm then attempts to collapse the nodes in the chaining graph by selecting an edge and replacing the two nodes with a new node. Once replaced, the algorithm tries to remove the edges between nodes of operations that can no longer be executed on the same FBB as the operation of the new node. This process iterates until no more nodes can be collapsed, and every remaining node in the chaining graph thus constitutes a *bundle*. The same authors later extended this design in [356] to consider selection between *all* possible bundles using branch-and-bound search. The new design also allows some measure of code duplication by allowing the same operations in the function graph to appear in multiple bundles.

Using this incremental scheme to form the bundles, the design by Van Praet et al. is capable of bundling operations that potentially reside in different blocks. Their somewhat-integrated code generation approach also allows efficient assembly code to be generated for complex architectures, making it suitable for DSPs and ASIPs where the data paths are very irregular. It may be possible to also extend the technique to support inter-block instructions as well, but interdependent instructions are most likely out of reach due to its heuristic nature.

Generating Assembly Code Using Simulated Annealing Another, albeit unusual, code generation technique was proposed by Visser [358] in 1999. Like MSSQ and CHESS, Visser's approach is an integrated code generation design but solves the problem using simulated annealing.[1] In brief terms, an initial solution is found by randomly mapping each node in the function graph to a node in the hardware graph – which models the entire processor – and then a schedule is found using traditional

[1] *Simulated annealing* is a meta-heuristic that avoids getting stuck in a local maximum when searching for optimal solutions. For an overview, see for example [219].

list scheduling. A fitness function is then applied to judge the effectiveness of the solution, but the exact details are omitted from the paper. A proof-of-concept prototype was developed and tested on a simple program, but it appears no further research has been conducted on this idea.

D.5.2 Improving Code Quality with Mutation Scheduling

The last item we will discuss is a technique called *mutation scheduling*[2] which was introduced in 1994 by Novack et al. [286, 287]. Mutation scheduling is technically a form of instruction scheduling that primarily targets VLIW architectures, but it also integrates a sufficient amount of instruction selection to warrant being included in this dissertation. On the other hand, the amount of instruction selection that *is* incorporated is in turn not really based on graph covering. But as with trellis diagrams (see Aps. B and C on pp. 200 and 228, respectively), the author decided against discussing it in its own appendix.

Broadly speaking, mutation scheduling essentially tries to reduce the make span of programs for which assembly code have already been generated (hence instruction selection, instruction scheduling, and register allocation has already been performed).[3] This is done by progressively moving the computations, one at a time, such that they can be executed in parallel with other instructions and thus finish sooner. If such a move cannot be made, for example due to violation of some resource constraint or data dependency, then mutation scheduling tries to alter the value. This is called *value mutation*, which means that the current operation is replaced by other, equivalent operations that conform to the restrictions. These operations are selected from a *mutation set*, which is conceptually a recursive data structure, as an expression in the mutation set may use intermediate values that in turn necessitate mutation sets of their own. Novack et al. compute these mutation sets by first taking the original operation and then applying a series of semantic-preserving functions that have been derived from various logical axioms, algebraic theorems, and the characteristics of the target machine. For example, if the value X is computed as $Y + 5$, then Y can later be obtained by computing $X - 5$. Another example is multiplication by powers of 2, which can be replaced with `shift` instructions, provided such instructions are available. If this is beneficial, a value can also be recomputed instead of copied from its current location. This idea is known as *recomputation* (or *rematerialization*), which is a method for reducing register pressure, as it allows registers to be released at an earlier point in the assembly code.

[2] Despite its name, the idea of mutation scheduling is completely orthogonal to the theory of genetic algorithms.

[3] Although it is depicted here primarily as a post-step to code generation, one could just as well design a Davidson-Fraser-style compiler (see Ap. A on p. 148). In such a design, simple methods are applied to generate correct but naive assembly code, and then mutation scheduling is used to improve the code quality.

In mutation scheduling, "shorter" mutations are preferred over longer ones. This is because a value mutation of v can lead to a cascade of new computations, which all will need to be scheduled before v can be is computed. Note that these computations can be scheduled such that they appear in blocks preceding the block in which the computations of v appear. Hence the length of a mutation is loosely defined as the number of instruction bundles that may need to be modified in order to realize the mutation. Moreover, since the new computations of a successful mutation consume resources and incur dependencies of their own, the existing candidates appearing in mutation sets may need to be removed or modified. The "best" combination of mutations is then decided heuristically, but the paper is vague on how this is done exactly.

Novack et al. implemented a prototype by extending an existing scheduler based on *global resource-constrained percolation (GRiP)*, which is another global instruction scheduling technique developed by the same authors (see [284]). Subsequent experiments using a selected set of benchmark programs demonstrated that the mutation scheduling-based design yielded a two- to threefold performance improvement over the GRiP-only-based counterpart, partly due to its ability to apply rematerialization in regions where register pressure is high. Unfortunately the authors neglected to say anything about the time complexity of mutation scheduling, and whether it scales to larger programs.

D.6 Summary

In this appendix we have considered a number of techniques that are founded, in one form or another, on the principle of graph covering. Such techniques are among the most powerful methods of instruction selection since they perform global instruction selection as well as have more extensive instruction support compared to most and DAG covering-based designs.

Unfortunately this has not been fully exploited in existing techniques, partly due to limitations in the program representations or to restrictions enforced by the underlying solving techniques. Moreover, performing global instruction selection is computationally much harder compared to local instruction selection. Therefore, we will most likely only see these techniques applied in compilers whose users can afford very long compilation times (for example when targeting embedded systems with extremely high demands on performance, code size, power consumption, or a combination thereof).

APPENDIX

E

List of Techniques

The list starts on the next page, with its legend appearing at the end of the list. The techniques are ordered chronologically.

Note that the capabilities of all techniques have been set to reflect those exhibited by current implementation prototypes and known applications, not the capabilities that could potentially be achieved through extensions of the technique.

REFERENCES	PR	SC	OP	MO	DO	IB	IN	KNOWN APPLICATIONS
Lowry and Medlock [259]	ME	L	○	○	○	○	○	FHC
Orgass and Waite [294]	ME	L	○	○	○	○	○	SIMCMP
Elson and Rake [112]	ME	L	○	○	○	○	○	
Miller [275]	ME	L	○	○	○	○	○	DMACS
Wilcox [367]	ME	L	○	○	○	○	○	
Wasilew [361]	TC	L	○	○	○	○	○	
Donegan [104]	ME	L	○	○	○	○	○	
Tirrell [350]	ME	L	○	○	○	○	○	
Weingart [362]	TC	L	○	○	○	○	○	
Ammann et al. [12, 13]	ME	L	○	○	○	○	○	
Young [378]	ME	L	○	○	○	○	○	
Newcomer [281]	TC	L	○	○	○	○	○	
Simoneaux [338]	ME	L	○	○	○	○	○	
Snyder [340]	ME	L	○	○	○	○	○	
Fraser [143, 144]	ME	L	○	○	○	○	○	
Ripken [320]	TC	L	●	○	○	○	○	
Glanville and Graham [163]	TC	L	○	○	○	○	○	
Johnson [203, 204]	TC	L	○	○	○	○	○	PCC
Harrison [177]	ME$^+$	L	○	○	○	○	○	
Cattell et al. [67, 70, 250]	TC	L	○	○	○	○	○	PQCC
Auslander and Hopkins [30]	ME$^+$	L	○	○	○	○	○	
Ganapathi and Fischer [149, 150, 151, 152]	TC	L	○	○	○	○	○	
Krumme and Ackley [229]	ME	L	○	○	○	○	○	
Deutsch and Schiffman [99]	ME	L	○	○	○	○	○	
Christopher et al. [75]	TC	L	●	○	○	○	○	
Davidson and Fraser [93]	ME$^+$	L	○	●	●	○	○	ACK, GCC, ZEPHYRVPO
Henry [183]	TC	L	●	○	○	○	○	
Aho et al. [7, 8, 352]	TC	L	●	○	○	○	○	TWIG
Hatcher and Christopher [179]	TC	L	●	○	○	○	○	
Horspool [197]	TC	L	●	○	○	○	○	

REFERENCES	PR	SC	OP	MO	DO	IB	IN	KNOWN APPLICATIONS
Fraser and Wendt [138]	ME$^+$	L	○	○	○	○	○	
Giegerich and Schmal [162]	TC	L	○	○	○	○	○	
Hatcher and Tuller [181]	TC	L	●	○	○	○	○	UNH-Codegen
Pelegrı-Llopart and Graham [298]	TC	L	●	○	○	○	○	
Yates and Schwartz [376]	TC	L	●	○	○	○	○	
Emmelmann et al. [113]	TC	L	●	○	○	○	○	BEG, CoSy
Ganapathi [148]	TC	L	○	●	○	○	○	
Genin et al. [157]	ME$^+$	L	○	●	○	○	○	
Nowak and Marwedel [288]	DC	L	○	●	●	○	○	MSSC
Balachandran et al. [32]	TC	L	●	○	○	○	○	
Despland et al. [62, 97, 98]	TC	L	○	○	○	○	○	Pagode
Wendt [364]	ME$^+$	L	○	●	○	○	○	
Hatcher [180]	TC	L	●	○	○	○	○	UCG
Fraser et al. [140]	TC	L	●	○	○	○	○	IBurg, Record, Redaco
Fraser et al. [141, 303, 304, 306, 307]	TC	L	●	○	○	○	○	Burg, HBurg, JBurg, WBurg, OCamlBurg
Emmelmann [114]	TC	L	○	○	○	○	○	
Wess [365, 366]	TD	L	●	○	○	○	○	
Marwedel [270]	TC	L	○	●	●	○	○	MSSV
Tjiang [351]	TC	L	●	○	○	○	○	Olive, Spam
Engler and Proebsting [118]	TC	L	●	○	○	○	○	DCG
Fauth et al. [129, 277]	DC	L	○	●	○	○	○	CBC
Ferdinand et al. [131]	TC	L	●	○	○	○	○	
Liem et al. [253, 296, 297]	DC	L	○	○	○	○	○	CodeSyn
Lanneer et al. [234, 356, 357]	GC	G	●	●	●	○	○	Chess
Wilson et al. [368]	DC	L	●	●	○	○	○	
Yu and Hu [379, 380]	DC	L	●	●	○	○	○	
Novack et al. [286, 287]	MS	G	●	●	○	○	○	
Hanson and Fraser [175]	TC	L	●	○	○	○	○	LBurg, LCC
Liao et al. [251, 252]	DC	L	●	●	○	○	○	
Adl-Tabatabai et al. [1]	ME	L	○	○	○	○	○	Omniware
Engler [117]	ME	L	○	○	○	○	○	VCode

REFERENCES	PR	SC	OP	MO	DO	IB	IN	KNOWN APPLICATIONS
Hoover and Zadeck [195]	DC	L	○	●	●	○	○	
Leupers and Marwedel [245, 249]	DC	L	○	●	●	○	○	
Nymeyer et al. [289, 290]	TC	L	○	○	○	○	○	
Shu et al. [337]	TC	L	○	○	○	○	○	
Gough [165, 166, 167]	TC	L	●	○	○	○	○	MBurg, GPBurg
Gebotys [155]	DC	L	●	●	○	○	○	
Hanono and Devadas [173, 174]	TD	L	○	○	○	○	○	Aviv
Leupers and Marwedel [246]	DC	L	○	●	●	○	○	MSSQ
Bashford and Leupers [36]	DC	L	●	●	●	○	○	
Ertl [121]	DC	L	●	●	○	○	○	DBurg
Fraser and Proebsting [142]	ME	L	○	○	○	○	○	GBurg
Fröhlich et al. [145]	TD	L	●	○	○	○	○	
Visser [358]	GC	G	○	●	●	○	○	
Leupers [243]	DC	L	○	●	●	○	○	
Madhavan et al. [261]	TC	L	●	○	○	○	○	
Arnold and Corporaal [22, 23, 24]	DC	L	○	●	○	○	○	
Sarkar et al. [326]	DC	L	○	○	○	○	○	Jalapeño
Paleczny et al. [295]	GC	G	●	○	○	○	○	JHSC
Lorenz et al. [256, 257]	DC	L	○	●	○	○	○	
Bravenboer and Visser [55]	TC	L	●	○	○	○	○	
Krishnaswamy and Gupta [227]	ME$^+$	L	○	●	●	○	○	
Eckstein et al. [109]	GC	G	○	○	○	○	○	
Tanaka et al. [348]	DC	L	○	●	●	○	○	
Borchardt [48]	TC	L	●	○	○	○	○	
Brisk et al. [57]	DC	L	○	●	○	○	○	
Cong et al. [84]	DC	L	○	●	●	○	○	
Lattner and Adve [235]	DC	L	○	○	○	○	○	LLVM
Kessler et al. [38, 119, 120]	DC	L	●	●	○	○	○	Optimist
Clark et al. [79]	DC	L	○	●	●	○	○	
Dias and Ramsey [101]	ME$^+$	L	○	●	○	○	○	
Ertl et al. [122]	DC	L	●	○	○	○	○	

REFERENCES	PR	SC	OP	MO	DO	IB	IN	KNOWN APPLICATIONS
Farfeleder et al. [125]	DC	L	○	◐	○	○	○	
Kulkarni et al. [230]	ME$^+$	L	○	●	●	○	○	VISTA
Hormati et al. [196]	DC	L	○	●	●	○	○	
Scharwaechter et al. [329]	DC	L	○	○	●	○	○	CBURG
Ebner et al. [108]	GC	G	○	●	●	○	○	
Koes and Goldstein [224]	DC	L	○	●	○	○	○	NOLTIS
Ahn et al. [3]	DC	L	○	●	●	○	○	
Martin et al. [266, 267]	DC	L	●	●	●	○	○	
Buchwald and Zwinkau [60]	GC	G	●	○	○	○	○	
Dias and Ramsey [100, 314]	ME$^+$	L	○	●	○	○	○	
Edler von Koch et al. [110]	TC	L	○	○	○	○	○	
Floch et al. [135]	DC	L	●	●	●	○	○	
Yang [375]	TC	L	●	○	○	○	○	
Youn et al. [377]	DC	L	○	◐	●	○	○	
Arslan and Kuchcinski [25, 26, 27]	DC	L	●	●	●	○	○	
Janoušek and Málek [201]	TC	L	○	○	○	○	○	
Andrade [15]	ME	L	○	○	○	○	○	GNU LIGHTNING
Hjort Blindell et al. [188, 189]	GC	G	●	●	●	●	○	

Fundamental principle (PR) on which each technique is based: macro expansion (ME), macro expansion with peephole optimization (ME$^+$), tree covering (TC), trellis diagrams (TD), DAG covering (DC), graph covering (GC), and mutation scheduling (MS). Scope of instruction selection (SC): local (L, isolated to a single block), and global (G, considers entire functions). Whether the technique is claimed to be optimal (OP). Supported instruction characteristics: single-output (supported by all techniques), multi-output (MO), disjoint-output (DO), inter-block (IB), and interdependent (IN) instructions. The symbols ○, ◐, and ● indicate no, partial, and full support, respectively.

APPENDIX

F

Graph Definitions

Nodes, Edges, and Graphs A *graph* is defined as a tuple (N, E) where N is a set of *nodes* (also known as *vertices*) and E is a set of *edges*, each consisting of a pair of nodes $n, m \in N$. A graph $G = (N, E)$ is a *subgraph* of another graph $G' = (N', E')$, written $G \subseteq G'$, if and only $N \subseteq N'$ and $E \subseteq E'$. A graph is *undirected* if its edges have no direction, and *directed* if they do. Edges with a direction are written either (n, m) or $n \to m$, whichever is most convenient, and edges without are written $\{n, m\}$. In a directed graph, we say that an edge $e = (n, m)$ is *outgoing* (or *outbound*) with respect to n, and *ingoing* (or *inbound*) with respect to m. We also call n and m the *source* and *target*, respectively, of e, and introduce two functions, $source\colon E \to N$ and $target\colon E \to N$, that respectively returns an edge's source and target. Edges for which $source(e) = target(e)$ are known as *loop edges* (or simply *loops*).

If there exists more than one edge between the same pair of nodes then the graph is a *multigraph*, otherwise it is a *simple* graph. A *bipartite graph* (or *bigraph*) is a graph whose nodes can be divided into two disjoint sets N and M such that every edge connects a node in N with a node in M. Hence there are no edges between pairs drawn exclusively from N or M.

Paths and Connectivity A sequence of edges that describe how to get from one node to another is called a *path*. More formally, we define a path between two nodes n and m in a directed graph (N, E) as an ordered sequence $p = \langle e_1, \ldots, e_l \rangle$ for which $\forall e_i \in p : e_i \in E$, and $\forall 1 \leq i < l - 1 : target(e_i) = source(e_{i+1})$. Paths for undirected graphs are similarly defined and will thus be skipped. A path $\langle e_1, \ldots, e_l \rangle$ for which $source(e_1) = target(e_l)$ is called a *cycle*, and a cycle that visits all nodes in the graph exactly once is called a *Hamiltonian cycle*.

Two nodes n and m, where $n \neq m$, are said to be *connected* if there exists a path from n to m, and if the path is of length 1 then n and m are also *adjacent*. A directed graph containing no cycles is known as a *directed acyclic graph (DAG)*. An undirected graph is *connected* if and only if there exists a path for every distinct pair of nodes.

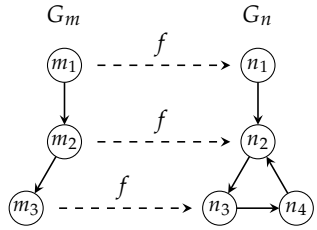


Figure F.1: An example of two simple, directed graphs G_m and G_n. Through the graph homomorphism f we see that G_m is an isomorphic subgraph of G_n. Both graphs are weakly connected, and G_n also has a strongly connected subgraph, consisting of n_2, n_3, and n_4 which form a cycle.

For completeness, a directed graph is *strongly connected* if and only if, for every pair of distinct nodes n and m, there exists a path from n to m and a path from m to n. Also, a directed graph is *weakly connected* if replacing all edges with undirected edges yields a connected undirected graph. An example is shown in Fig. F.1.

Trees A simple, undirected graph that is connected, contains no cycles, and has exactly one path between any two nodes, is called a *tree*. A tree T is a *subtree* of another tree T' if and only if $T \subseteq T'$. A set of trees constitutes a *forest*. Nodes in a tree with exactly one neighbor are known as *leaves*. A *directed tree* is a directed graph that would become a tree when ignoring the direction of its edges. A *rooted directed tree* is a directed tree where one node has been assigned as *root* and all edges either point away or towards the root. In a rooted directed tree, a *parent* of a node n is the node adjacent to n that is closest to the root. Likewise, if a node n is the parent of another node m, then m is a *child* of n. In this dissertation, we assume all trees to be rooted directed trees, and a tree will always be drawn with its root appearing at the top.

Isomorphisms A *graph homomorphism* is a mapping between two graphs such that their structure is preserved. More formally, a graph homomorphism f from a graph $G = (N, E)$ to another graph $G' = (N', E')$ is a mapping $f : N \to N'$ such that $\{u, v\} \in E$ implies $\{f(u), f(v)\} \in E'$. If the graph homomorphism f is an injective function, then f is also called a *subgraph isomorphism*. If there exists such a mapping then we say that G is an *isomorphic subgraph* of G', and an example of this is given in Fig. F.1. If f is a bijection, whose inverse function is also a graph homomorphism, then f is called a *graph isomorphism*.

Topological Sorts Lastly we introduce the notion of *topological sort*, where the nodes of a graph (N, E) are arranged in an ordered sequence $\langle n_1, \ldots, n_n \rangle$ such that $\forall 1 \leq i \leq n : n_i \in N$, and for no pair of nodes n_i and n_j, where $i < j$, does there exist an edge $(n_j, n_i) \in E$. In other words, if the edges are added to the list then all edges

will go point forward from left to right (hence topological sorts are only defined for DAGs). Several methods exists for achieving a topological sort, see for example in Cormen et al. [88, Sect. 22.4].

APPENDIX

G

MiniZinc Implementation

```
%
%  Main authors:
%    Gabriel Hjort Blindell <ghb@kth.se>
%    Mats Carlsson <matsc@sics.se>
%
%  Contributing authors:
%    Roberto Castaneda Lozano <rcas@sics.se>
%
%  Copyright (c) 2012-2018, Gabriel Hjort Blindell <ghb@kth.se>
%  All rights reserved.
%
%  Redistribution and use in source and binary forms, with or without
%  modification, are permitted provided that the following conditions are
%  met:
%  1. Redistributions of source code must retain the above copyright notice,
%     this list of conditions and the following disclaimer.
%  2. Redistributions in binary form must reproduce the above copyright
%     notice, this list of conditions and the following disclaimer in the
%     documentation and/or other materials provided with the distribution.
%  3. Neither the name of the copyright holder nor the names of its
%     contributors may be used to endorse or promote products derived from
%     this software without specific prior written permission.
%
%  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
%  IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
%  THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
%  PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE
%  FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
%  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
%  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
%  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
%  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
%  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
%  THE POSSIBILITY OF SUCH DAMAGE.

```

```
37
38
39 %=====================
40 % EXTERNAL PARAMETERS
41 %=====================
42
43 % Function data.
44 int: numOperationsInFunction;
45 int: numDataInFunction;
46 int: numBlocksInFunction;
47 int: entryBlockOfFunction;
48 int: execFrequencyGCD;
49 array[allBlocksInFunction] of set of int: domSetsOfBlocksInFunction;
50 array[allOperationsInFunction] of set of int: depsOfOpsInFunction; % NOT IN
51                                                                    % USE
52 array[allDataInFunction] of set of int: depsOfDataInFunction; % NOT IN USE
53 set of allOperationsInFunction: copiesInFunction;
54 set of allOperationsInFunction: controlOpsInFunction;
55 set of allDataInFunction: statesInFunction;
56 array[int, int] of int: defEdgesInFunction;
57 array[allBlocksInFunction] of int: execFrequencyOfBlockInFunction;
58 array[int] of set of int: interchangeableDataInFunction;
59 set of allDataInFunction: dataInFunctionUsedAtLeastOnce; % NOT IN USE
60
61 % Target machine data.
62 int: numLocations;
63 set of int: canonicalDataLocs;
64
65 % Match data.
66 int: numMatches;
67 int: numOperands;
68 array[allOperands] of set of int: operandAlternatives;
69 array[allMatches] of set of int: operationsCoveredByMatch;
70 array[allMatches] of set of int: operandsDefinedByMatch;
71 array[allMatches] of set of int: operandsUsedByMatch;
72 array[allMatches] of set of int: operandsExteriorToMatch;
73 array[allMatches] of set of int: operandsIntermediateToMatch;
74 array[allMatches] of set of int: entryBlockOfMatch;
75 array[allMatches] of set of int: spannedBlocksInMatch;
76 array[allMatches] of set of int: consumedBlocksInMatch;
77 array[int, int] of int: validDataLocsInMatch;
78
79 array[int, int] of int: inputDefEdgesInMatch;
80 array[int, int] of int: outputDefEdgesInMatch;
81 array[allMatches] of int: codeSizeOfMatch;
82 array[allMatches] of int: latencyOfMatch;
83 set of allMatches: copyInstrMatches;
84 set of allMatches: killInstrMatches;
85 set of allMatches: nullInstrMatches;
86 set of allMatches: phiInstrMatches;
87 array[allMatches] of allMatches: allMatchesBySize;
88
89 int: costLowerBound;
90 int: costUpperBound;
91
```

```
% Arrays that encode constraints
array[int, int] of int: validDataLocsInFunction;
array[int, int] of int: validDataLocRangesInFunction;
array[int, int] of int: sameDataLocsInFunction;
array[int, int] of int: validDataLocRangesInMatch;
array[int, int] of int: sameDataLocsInMatch;
array[int, int] of int: fallThroughBlockOfMatch;
array[int] of set of int: illegalMatchCombinations;

% These variables will be set by concatenating the definitions to the end of
% this file.


%=====================
% INTERNAL PARAMETERS
%=====================

% Total number of location values. Two additional values will be needed for
% representing the intermediate value location (for when the datum cannot be
% reused by other matches) and the killed location (for when the datum is
% defined by a kill match).
int: numLocValues = numLocations + 2;

% Reference to to the intermediate value location.
int: locValueForInt = numLocValues - 1;

% Reference to to the killed location.
int: locValueForKilled = numLocValues;

% Total number of data values. An additional value will be needed for
% representing the null datum (for operands used in non-selected matches).
int: numDataValues = numDataInFunction + 1;

% Reference to to the null datum.
int: datumValueForNull = numDataValues;

% Total number of blocks values. An additional value will be needed for
% representing the null block (to which non-selected matches will be
% placed).
int: numBlockValues = numBlocksInFunction + 1;

% Reference to the null block.
int: blockValueForNull = numBlockValues;

% Sets to be used as array ranges.
set of int: allOperationsInFunction = 1..numOperationsInFunction;
set of int: allDataInFunction = 1..numDataInFunction;
set of int: allBlocksInFunction = 1..numBlocksInFunction;
set of int: allMatches = 1..numMatches;
set of int: allOperands = 1..numOperands;
set of int: allLocValues = { x | x in canonicalDataLocs
                                    ++ [ locValueForInt
                                       , locValueForKilled
                                       ]
                           };
```

```
147
148 % The 'domRelMatrix' matrix is a 2D matrix with 2 columns:
149 %    col 1: a block i
150 %        2: a block j that dominates i
151 % In other words, domRelMatrix has a row [i, j] if and only if j belongs to
152 % domSetsOfBlocksInFunction[i].
153 %
154 % Eq. 6.1 in dissertation
155 int: numDomMatrixRows =
156   sum (b in allBlocksInFunction)
157   ( card(domSetsOfBlocksInFunction[b]) );
158 array[1..numDomMatrixRows, 1..2] of allBlocksInFunction: domRelMatrix =
159   array2d( 1..numDomMatrixRows
160          , 1..2
161          , [ if k = 1 then i else j endif
162            | i in allBlocksInFunction
163            , j in domSetsOfBlocksInFunction[i]
164            , k in 1..2
165            ]
166          );
167
168 % The set of matches that can cover a particular operation.
169 array[allOperationsInFunction] of set of allMatches: matchsetOfOp =
170   [ { m | m in allMatches where op in operationsCoveredByMatch[m] }
171   | op in allOperationsInFunction
172   ];
173
174 % The set of matches that can define a particular datum.
175 array[allDataInFunction] of set of allMatches: defsetOfDatum =
176   [ { m | m in allMatches
177         , p in operandsDefinedByMatch[m]
178           where d in operandAlternatives[p]
179     }
180   | d in allDataInFunction
181   ];
182
183 % The set of matches that can define a particular datum and does not cover
184 % any operations.
185 array[int] of set of allMatches: defsetOfDatumOnly =
186   [ M | M in defsetOfDatum
187         where forall (i in index_set(matchsetOfOp))
188               ( matchsetOfOp[i] != M )
189   ];
190
191 % Maps a match to an operation. This is only needed for figuring out in
192 % which block a selected match is to be placed.
193 array[allMatches] of int: opOfM =
194   [ min({ o | o in allOperationsInFunction where m in matchsetOfOp[o] })
195   | m in allMatches
196   ];
197
198 % Maps an operand to a match. This is only needed for figuring out which
199 % operands have not been assigned a datum.
200 array[allOperands] of int: matchOfP =
201   [ m | p in allOperands
```

```
      , m in allMatches
        where p in operandsDefinedByMatch[m] union operandsUsedByMatch[m]
  ];

% The total cost is computed as:
%
%   sum (m in allMatches)
%   ( latencyOfMatch[m] * execFrequencyOfBlocks[place[m]] )
%
% but this implementation yields poor propagation. To improve propagation,
% we use the cost per operation instead of cost per selected match.
%
% First, we split the cost incurred by selecting a given match over the
% operations that it covers. As the latency of a match times the exequtions
% frequency of a given block may not be evenly divisible by the number of
% operations covered, some operations may have a greater cost than
% others. Then, we multiply each operation cost with the execution
% frequency of the block wherein the operation may be placed.
%
% This information is put in a matrix called 'costPerOpMatrix'.

% The 'costPerOpMatrix' is a 2D matrix with 4 columns:
%    col 1: an operation o
%        2: a match m that covers o
%        3: a block b in which m can be placed
%        4: the cost incurred by o if m is selected and placed in b
% For simplicity, we first create a list and then the matrix using the list.
%
% Eqs. 6.6 and 6.7 in dissertation
array[int] of int: costPerOpList =
  [ if      k = 1 then o
    else if k = 2 then m
    else if k = 3 then b
    else let
         { int: d = card(operationsCoveredByMatch[m])
         , int: q = latencyOfMatch[m] div d
         , int: r = latencyOfMatch[m] mod d
         } in if operationsCoveredByMatch[m][r+1] > o
              then (q+1) * execFrequencyOfBlockInFunction[b]
              else q * execFrequencyOfBlockInFunction[b]
              endif
    endif
    endif
    endif
  | o in allOperationsInFunction
  , m in allMatches
  , b in if card(entryBlockOfMatch[m]) > 0 then entryBlockOfMatch[m]
         else allBlocksInFunction
         endif
  , k in 1..4
    where o in operationsCoveredByMatch[m]
  ];
int: numCostPerOpMatrixRows = card(index_set(costPerOpList)) div 4;
array[1..numCostPerOpMatrixRows, 1..4] of int: costPerOpMatrix =
  array2d(1..numCostPerOpMatrixRows, 1..4, costPerOpList);
```

```
257
258 % A set with all the possible costs that can be incurred by any operation.
259 set of int: allOpCosts = { costPerOpMatrix[i, 4]
260                           | i in 1..numCostPerOpMatrixRows
261                           };
262
263 % The set of all non-phi instruction use operands.
264 set of int: nonPhiUseOperands =
265   { p
266   | m in allMatches diff phiInstrMatches
267   , p in operandsUsedByMatch[m]
268   };
269
270 % The set of operands which are either used or defined by kill instructions.
271 set of int: killOperands =
272   array_union( [ operandsDefinedByMatch[m] | m in killInstrMatches ]
273                ++
274                [ operandsUsedByMatch[m] | m in killInstrMatches ]
275              );
276
277
278 %===========
279 % VARIABLES
280 %===========
281
282 % Match selection.
283 array[allMatches] of var bool: sel;
284
285 % Blocks wherein the data are placed.
286 array[allDataInFunction] of var allBlocksInFunction: dplace;
287
288 % The block in which a particular operation is placed.
289 array[allOperationsInFunction] of var allBlocksInFunction: oplace;
290
291 % Data locations.
292 array[allDataInFunction] of var allLocValues: loc;
293
294 % Data selected for the operands.
295 array[allOperands] of var allDataInFunction: alt;
296
297 % Block ordering (succ[b] is the block appearing immeditely after block b
298 % in the generated code).
299 array[allBlocksInFunction] of var allBlocksInFunction: succ;
300
301 % Cost.
302 var int: totalcost;
303 array[allOperationsInFunction] of var allOpCosts: opcosts;
304
305
306 %================
307 % DUAL VARIABLES
308 %================
309
310 % The match that covers a particular operation.
311 array[allOperationsInFunction] of var allMatches: omatch;
```

```
312
313 % The match that defines a particular datum.
314 array[allDataInFunction] of var allMatches: dmatch;
315
316 % For selected, non-null, non-phi instruction use operands:
317 %   block where the datum of a given operand is used.
318 % For non-selected, non-null, non-phi instruction use operands:
319 %   block where the datum of a given operand is defined.
320 % For other operands:
321 %   1.
322 array[allOperands] of var allBlocksInFunction: uplace;
323
324
325 %====================
326 % GLOBAL CONSTRAINTS
327 %====================
328
329 include "globals.mzn";
330
331
332 %===========
333 % FUNCTIONS
334 %===========
335
336 % Bypasses the alt[.] variable if the given operand only has one
337 % alternative. Valid under the assumption that its match was selected.
338 var allDataInFunction: Alt(allOperands: p) =
339   let { allDataInFunction: d = min(operandAlternatives[p]) }
340   in if card(operandAlternatives[p]) = 1 then d else alt[p] endif;
341
342 % True if the value of a given operand for sure must not be in the
343 % intermediate value nor in the killed location.
344 test valueOfOpMustBeAvailable(allOperands: p) =
345   exists (i in index_set_1of2(validDataLocRangesInMatch))
346   ( validDataLocRangesInMatch[i, 2] = p /\
347     validDataLocRangesInMatch[i, 4] < locValueForInt
348     % Since locValueForInt < locValueForKilled, this also entails that the
349     % range does not include locValueForKilled
350   );
351
352 % True if a datum for sure must not be in the intermediate value nor in the
353 % killed location.
354 test valueMustBeAvailable(allDataInFunction: d) =
355   forall ( m in defsetOfDatum[d] )
356   ( not (m in killInstrMatches) /\
357     exists ( use2 in operandsDefinedByMatch[m] intersect
358                      operandsExteriorToMatch[m]
359            )
360     ( operandAlternatives[use2] = {d} )
361   );
362
363
364 %==================
365 % BASE CONSTRAINTS
366 %==================
```

```
367
368 % Constrain locations of data representing values.
369 constraint
370   forall (d in allDataInFunction)
371   ( let { set of int: locs = { validDataLocsInFunction[i, 2]
372                              | i in index_set_1of2(validDataLocsInFunction)
373                                where validDataLocsInFunction[i, 1] = d
374                              }
375         }
376     in if card(locs) > 0 then loc[d] in locs else true endif
377   );
378
379 % Constrain locations of data that must be within a specific range.
380 constraint
381   forall (i in index_set_1of2(validDataLocRangesInFunction))
382   ( let { int: d = validDataLocRangesInFunction[i, 1]
383         , int: l = validDataLocRangesInFunction[i, 2]
384         , int: u = validDataLocRangesInFunction[i, 3]
385         }
386     in loc[d] in l..u
387   );
388
389 % Constrain locations of data which must be assigned the same location.
390 constraint
391   forall (i in index_set_1of2(sameDataLocsInFunction)) (
392     let { int: p1 = sameDataLocsInFunction[i, 1]
393         , int: p2 = sameDataLocsInFunction[i, 2]
394         }
395     in loc[Alt(p1)] = loc[Alt(p2)]
396   );
397
398 % Constrain alternatives of operands.
399 constraint
400   forall (p in allOperands)
401   ( alt[p] in operandAlternatives[p] );
402
403 % All operands exterior to a match (excluding kill matches), and which are
404 % not states, must not be located in the intermediate value location nor
405 % the killed location.
406 constraint
407   forall ( m in allMatches diff killInstrMatches
408          , p in operandsExteriorToMatch[m]
409            where not (operandAlternatives[p] subset statesInFunction)
410               /\ not valueOfOpMustBeAvailable(p)
411          )
412   ( if forall (d in operandAlternatives[p])
413        ( valueMustBeAvailable(d) )
414     then true
415     else sel[m] -> ( loc[Alt(p)] != locValueForInt /\
416                      loc[Alt(p)] != locValueForKilled
417                    )
418     endif
419   );
420
421 % All operands intermediate to a match must be located in the intermediate
```

```
% value location.
constraint
  forall ( m in allMatches
         , p in operandsIntermediateToMatch[m]
           where not (operandAlternatives[p] subset statesInFunction)
         )
  ( sel[m] -> loc[Alt(p)] = locValueForInt );

% If a match representing a phi instruction is selected, then its operands
% must be placed in the same location.
%
% Eq. 5.16 in dissertation
constraint
  forall ( m in phiInstrMatches
         , p1 in operandsExteriorToMatch[m]
         )
  ( let { int: p2 = min(operandsExteriorToMatch[m]) }
    in if p1 != p2 then sel[m] -> loc[Alt(p1)] = loc[Alt(p2)]
       else true
       endif
  );

% Constrain locations of operands representing values for selected matches.
%
% Eq. 5.15 in dissertation
constraint
  forall ( m in allMatches
         , p in operandsDefinedByMatch[m] union operandsUsedByMatch[m]
         )
  ( let { set of int: locs = { validDataLocsInMatch[i, 3]
                             | i in index_set_1of2(validDataLocsInMatch)
                               where validDataLocsInMatch[i, 1] = m
                                     /\
                                     validDataLocsInMatch[i, 2] = p
                             }
        }
    in if card(locs) > 0
       then sel[m] -> loc[Alt(p)] in locs
       else true
       endif
  );

% Constrain locations of operands that must be within a specific range.
%
% If no entry appears in validDataLocRangesInMatch for a given match and
% operand, then it means no location restrictions are applied.
constraint
  forall (i in index_set_1of2(validDataLocRangesInMatch)) (
    let { int: m = validDataLocRangesInMatch[i, 1]
        , int: p = validDataLocRangesInMatch[i, 2]
        , int: l = validDataLocRangesInMatch[i, 3]
        , int: u = validDataLocRangesInMatch[i, 4]
        }
    in sel[m] -> loc[Alt(p)] in l..u
  );
```

```
477
478 % For selected matches that require two or more of its operands to have the
479 % same location, enforce them to be the same.
480 %
481 % Similar to Eq. 5.16 in dissertation
482 constraint
483   forall (i in index_set_1of2(sameDataLocsInMatch)) (
484     let { int: m = sameDataLocsInMatch[i, 1]
485         , int: p1 = sameDataLocsInMatch[i, 2]
486         , int: p2 = sameDataLocsInMatch[i, 3]
487         }
488     in sel[m] -> loc[Alt(p1)] = loc[Alt(p2)]
489   );
490
491 % If a match representing a phi instruction is selected, then its data must
492 % be defined in the blocks indicated by the definition edges.
493 %
494 % Eq. 5.18 in dissertation
495 constraint
496   forall (i in index_set_1of2(inputDefEdgesInMatch)) (
497     let { int: m = inputDefEdgesInMatch[i, 1]
498         , int: b = inputDefEdgesInMatch[i, 2]
499         , int: p = inputDefEdgesInMatch[i, 3]
500         }
501     in sel[m] -> dplace[Alt(p)] = b
502   );
503 constraint
504   forall (i in index_set_1of2(outputDefEdgesInMatch)) (
505     let { int: m = outputDefEdgesInMatch[i, 1]
506         , int: b = outputDefEdgesInMatch[i, 2]
507         , int: p = outputDefEdgesInMatch[i, 3]
508         }
509     in sel[m] -> dplace[Alt(p)] = b
510   );
511
512 % A datum with a definition edge must be defined in the block of that edge.
513 %
514 % Eq. 5.9 in dissertation
515 constraint
516   forall (i in index_set_1of2(defEdgesInFunction)) (
517     let { int: b = defEdgesInFunction[i, 1]
518         , int: d = defEdgesInFunction[i, 2]
519         }
520     in dplace[d] = b
521   );
522
523 % If a match is selected, then all operations covered by that match must be
524 % placed in the same block.
525 %
526 % Eq. 5.4 in dissertation
527 constraint
528   forall ( m in allMatches
529          , o in operationsCoveredByMatch[m]
530            where o != opOfM[m]
531          )
```

```
532   ( sel[m] -> oplace[opOfM[m]] = oplace[o] );
533
534 % If a selected match m has an entry block b, then all operations covered
535 % by m must be placed in b.
536 %
537 % If a match has no entry block, then this set will be empty and hence there
538 % will be no such constraint. It is assumed that there will be at most one
539 % entry.
540 %
541 % Eq. 5.5 in dissertation
542 constraint
543   forall ( m in allMatches
544          , b in entryBlockOfMatch[m]
545          , o in operationsCoveredByMatch[m]
546          )
547   ( sel[m] -> oplace[o] = b );
548
549 % Data defined by a selected match m must be defined either in one of the
550 % blocks spanned by m, or the block wherein the operations covered by m are
551 % placed.
552 %
553 % Eq. 5.14 in dissertation
554 constraint
555   forall ( m in allMatches
556          , p in operandsDefinedByMatch[m]
557          )
558   ( sel[m] -> if card(spannedBlocksInMatch[m]) > 0
559               then dplace[Alt(p)] in spannedBlocksInMatch[m]
560               else forall (o in operationsCoveredByMatch[m])
561                    ( dplace[Alt(p)] = oplace[o] )
562               endif
563   );
564
565 % No operations may be placed in a block which is consumed by some selected
566 % match.
567 %
568 % Eq. 5.8 in dissertation
569 constraint
570   forall ( m in allMatches
571          , o in allOperationsInFunction diff operationsCoveredByMatch[m]
572          , b in consumedBlocksInMatch[m]
573          )
574   ( sel[m] -> oplace[o] != b );
575
576 % For each selected match m that apply fall-through, enforce either:
577 %    - that the fall-through block of m is the immediate successor of the
578 %      entry block of m, or
579 %    - that the fall-through block of m is one block away from the entry
580 %      block of m and the block in between contains no
581 %      non-null-instructions.
582 %
583 % Eqs. 5.19-5.22 in dissertation
584 constraint
585   forall (i in index_set_1of2(fallThroughBlockOfMatch))
586   ( let { int: m = fallThroughBlockOfMatch[i, 1]
```

```
587         , int: fall_b = fallThroughBlockOfMatch[i, 2]
588         }
589     in sel[m] -> falls_through(m, fall_b)
590   );
591
592 predicate falls_through(allMatches: m, allBlocksInFunction: fall_b) =
593   let { int: entry_b = min(entryBlockOfMatch[m])
594       , var int: succ_b = succ[entry_b]
595       }
596   in succ_b != entryBlockOfFunction /\
597      ( succ_b = fall_b \/
598        ( succ[succ_b] = fall_b /\
599          forall (o in allOperationsInFunction)
600          ( oplace[o] != succ_b \/
601            omatch[o] in nullInstrMatches
602          )
603        )
604      );
605
606 % Enforce that, for each operation o, exactly one match must be selected
607 % such that o is covered.
608 %
609 % THIS HAS BEEN REPLACED WITH DUAL VARIABLE CONSTRAINT.
610
611 % Enforce that every datum is defined in a block such that the block
612 % dominates all blocks wherein the datum is used. This constraint shall not
613 % be applied to the generic phi patterns, nor to null instructions.
614 %
615 % This used to be enforced by the following constraint:
616 %
617 %   constraint
618 %     forall ( m in allMatches
619 %            , p in operandsUsedByMatch[m]
620 %              where not (m in phiInstrMatches)
621 %            )
622 %     ( dplace[alt[p]] in domSetsOfBlocksInFunction[place[m]] );
623 %
624 % but it has been reformulated using table constraints to avoid the use of
625 % set variables.
626 %
627 % This assumes that matches which use some data cover at least one
628 % operation, which should always hold.
629 %
630 % Eq.6.2 in dissertation
631 constraint
632   forall (p in nonPhiUseOperands)
633   ( table([uplace[p], dplace[Alt(p)]], domRelMatrix) );
634
635 % Eq.6.3 in dissertation
636 constraint
637   forall ( m in allMatches diff phiInstrMatches
638          , p in operandsUsedByMatch[m]
639          , o in operationsCoveredByMatch[m]
640          )
641   ( sel[m] -> oplace[o] = uplace[p] );
```

```
642
643 % Eq. 6.4 in dissertation
644 constraint
645   forall ( m in allMatches diff phiInstrMatches
646          , p in operandsUsedByMatch[m]
647          )
648   ( not sel[m] -> uplace[p] = dplace[Alt(p)] );
649
650 % Eq. 6.5 in dissertation
651 constraint
652   forall (p in allOperands diff nonPhiUseOperands)
653   ( uplace[p] = 1 );
654
655 % A kill match is selected if and only if the location of the defined datum
656 % is in the killed location.
657 %
658 % Eq. 5.17 in dissertation
659 constraint
660   forall ( m in killInstrMatches
661          , p in operandsDefinedByMatch[m]
662          )
663   ( sel[m] <-> loc[Alt(p)] = locValueForKilled );
664
665 % Ensure that succ forms a circuit (thus resulting in an ordering of
666 % blocks).
667 %
668 % Eq. 5.19 in dissertation
669 constraint
670   if card(allBlocksInFunction) > 1
671   then circuit(succ) :: domain
672   else true
673   endif;
674
675 % Forbid matches in an illegal combination from all being selected.
676 %
677 % Eq. 5.3 in dissertation
678 constraint
679   forall (c in illegalMatchCombinations)
680   ( sum (m in c) (bool2int(sel[m])) < card(c) );
681
682 % Constrain the cost that can be incurred by each operation.
683 %
684 % Eq. 6.10 in dissertation
685 constraint
686   forall (o in allOperationsInFunction)
687   ( table([o, omatch[o], oplace[o], opcosts[o]], costPerOpMatrix) );
688
689 % The total cost is the sum of the costs incurred by all operations.
690 %
691 % Eq. 6.11 in dissertation
692 constraint
693   totalcost = sum(opcosts);
694
695 % Constraint the lower bound of the cost.
696 %
```

```
% Eq. 6.34 in dissertation
constraint
  if costLowerBound > 0
  then totalcost >= costLowerBound
  else true
  endif;

% Constraint the upper bound of the cost (retrieved from LLVM).
%
% Eq. 6.34 in dissertation
constraint
  if costUpperBound > 0
  then totalcost < costUpperBound
  else true
  endif;


%============================
% DUAL VARIABLE CONSTRAINTS
%============================

% For each operation o, exactly one match must be selected such that o is
% covered.
%
% This replaces the constraint that, for each operation o, exactly one match
% must be selected such that o is covered:
%
%   constraint
%     forall (o in allOperationsInFunction)
%     ( let { set of int: mset = { m
%                                | m in allMatches
%                                  where o in operationsCoveredByMatch[m]
%                                }
%           }
%       in sum (m in mset) (bool2int(sel[m])) = 1
%     );
%
% Eq. 5.1 in dissertation
constraint
  forall (o in allOperationsInFunction)
  ( omatch[o] in matchsetOfOp[o]
    /\
    forall (m in matchsetOfOp[o])
    ( omatch[o] = m <-> sel[m] )
  );

% For each datum d, exactly one match must be selected such that d is
% defined.
%
% This replaces the constraint that that, for each datum d, exactly one
% match must be selected such that d is defined:
%
%   constraint
%     forall (d in allDataInFunction)
%     ( let { set of int: mset = { m | m in allMatches
```

```
%                                       , p in operandsDefinedByMatch[m]
%                                         where d in operandAlternatives[p]
%                                   }
%           }
%       in sum (m in mset) (bool2int(sel[m])) = 1
%     );
%
% This is an implied constraint, but it also enforces that the patterns for
% defining the function input and constants are selected. Such patterns do
% not cover any operations, they are not entailed in the above constraint
% for exactly covering each operation.
%
% Eq. 5.2 in dissertation
constraint
  forall (d in allDataInFunction)
  ( dmatch[d] in defsetOfDatum[d]
    /\
    forall (m in defsetOfDatum[d])
    ( dmatch[d] = m <-> sel[m] )
  );


%=======================
% DOMINANCE CONSTRAINTS
%=======================

% Constrain the loc value for all data that are states.
%
% Eq. 6.27 in dissertation
constraint
  forall (d in statesInFunction)
  ( loc[d] = locValueForInt );

% Fix operand value if match was not selected.
%
% Eq. 6.28 in dissertation
constraint
  forall ( m in allMatches
         , p in operandsDefinedByMatch[m] union operandsUsedByMatch[m]
           where card(operandAlternatives[p]) > 1
         )
  ( not sel[m] -> alt[p] = min(operandAlternatives[p]) );

% Break symmetries introduced by interchangeable data.
%
% This could remove potential optimal solutions in situations where data is
% interchangeable WITHIN a group of matches, but not BETWEEN groups of
% matches. Have yet to see such an occurrance, though.
%
% Eq. 6.30 in dissertation
constraint
  forall ( chain in interchangeableDataInFunction )
  ( let { set of int: xset = { p | p in nonPhiUseOperands
                                   where operandAlternatives[p] = chain
                              }
```

```
807         }
808      in if card(xset) > 1
809         then value_precede_chain(chain, [alt[p] | p in xset])
810         else true
811         endif
812    );
813
814  % A consequence of symmetry breaking (concerns selection of null copies).
815  %
816  % Eqs. 6.31 and 6.32 in dissertation
817  constraint
818    forall ( chain in interchangeableDataInFunction
819             where forall (d in chain)
820                   ( defsetOfDatum[d] subset copyInstrMatches )
821           )
822    ( let { array[int] of int: copies =
823              [ min( defsetOfDatum[d] intersect
824                     nullInstrMatches diff
825                     killInstrMatches
826                   )
827              | d in chain
828                where card( defsetOfDatum[d] intersect
829                            nullInstrMatches diff
830                            killInstrMatches
831                          ) > 0
832              ]
833          }
834      in increasing (m in copies)
835         ( sel[m] )
836    );
837
838  % A consequence of symmetry breaking (concerns selection of kill matches).
839  %
840  % Eq. 6.33 in dissertation
841  constraint
842    forall ( chain in interchangeableDataInFunction
843             where forall (d in chain)
844                   ( defsetOfDatum[d] subset copyInstrMatches )
845           )
846    ( let { array[int] of int: kills =
847              [ min(defsetOfDatum[d] intersect killInstrMatches)
848              | d in chain
849                where card(defsetOfDatum[d] intersect killInstrMatches) > 0
850              ]
851          }
852    in if length(kills) > 0
853       then increasing (m in kills)
854            ( sel[m] )
855       else true
856       endif
857    );
858
859
860  %=====================
861  % IMPLIED CONSTRAINTS
```

```
%====================

% If all matches covering an operation o are not phi instruction, do not
% span any blocks, use datum du, and define datum dd, then the block
% defining du must dominate the block defining dd, and o must be placed in
% the block defining dd.
%
% Eqs. 6.12 and 6.13 in dissertation
constraint
  forall ( o in allOperationsInFunction
           where forall (m in matchsetOfOp[o])
                 ( not (m in phiInstrMatches) /\
                   card(spannedBlocksInMatch[m]) = 0
                 )
         )
  ( let { set of allMatches: M = matchsetOfOp[o]
        , int: mo = min({ m1 | m1 in M
                               where forall (m2 in M)
                                     ( card(operationsCoveredByMatch[m1]) <=
                                       card(operationsCoveredByMatch[m2])
                                     )
                         })
        , set of allDataInFunction: DD =
            { d | p in operandsDefinedByMatch[mo]
                , d in operandAlternatives[p]
            }
        , set of allDataInFunction: DU = { d | p in operandsUsedByMatch[mo]
                                             , d in operandAlternatives[p]
                                         }
        }
    in forall (dd in DD, du in DU)
       ( if forall (m in M)
            ( exists (p in operandsDefinedByMatch[m])
              ( operandAlternatives[p] = {dd} )
              /\
              exists (p in operandsUsedByMatch[m])
              ( operandAlternatives[p] = {du} )
            )
         then table([dplace[dd], dplace[du]], domRelMatrix) /\
              oplace[o] = dplace[dd]
         else true
         endif
       )
  );

% If all matches in the matchset covering a particular operation have
% identical entry blocks, and use datum d, then the block defining d must
% dominate the entry block.
%
% Eq. 6.16 in dissertation
constraint
  forall ( M in matchsetOfOp
         , d in allDataInFunction
           where forall (m in M)
                 ( not (m in phiInstrMatches)
```

```
                    /\
                    card(spannedBlocksInMatch[m]) > 0
                  )
          )
   ( let { int: Entry = min(entryBlockOfMatch[min(M)]) }
     in if forall (m in M)
           ( min(entryBlockOfMatch[m]) = Entry /\
             exists (p in operandsUsedByMatch[m])
             ( operandAlternatives[p] = {d} )
           )
        then table([Entry, dplace[d]], domRelMatrix)
        else true
        endif
   );

% If all matches in the matchset that cover a particular operation have
% identical spanned blocks, and define datum d, then d must be placed in a
% spanned block.
%
% Eq. 6.14 in dissertation
constraint
  forall ( M in matchsetOfOp
         , d in allDataInFunction,
           where forall (m in M)
                 ( card(spannedBlocksInMatch[m]) > 0 )
         )
  ( let { set of int: Spanned = spannedBlocksInMatch[min(M)] }
    in if forall (m in M)
          ( spannedBlocksInMatch[m] = Spanned /\
            exists (p in operandsDefinedByMatch[m])
            ( operandAlternatives[p] = {d} )
          )
       then dplace[d] in Spanned
       else true
       endif
  );

% If all matches for an operation o have identical entry blocks, then o
% must be placed in the entry block.
%
% Eq. 6.15 in dissertation
constraint
  forall ( o in allOperationsInFunction
           where forall (m in matchsetOfOp[o])
                 ( card(spannedBlocksInMatch[m]) > 0 )
         )
  ( let { int: Entry = min(entryBlockOfMatch[min(matchsetOfOp[o])]) }
    in if forall (m in matchsetOfOp[o])
          ( min(entryBlockOfMatch[m]) = Entry )
       then oplace[o] = Entry
       else true
       endif
  );

% If all matches covering an operation o are all phi instructions and define
```

```
% the same datum, then o must be placed in the same block as the datum.
%
% Eq. 6.17 in dissertation
constraint
  forall (i in index_set_1of2(defEdgesInFunction)) (
    let { int: b = defEdgesInFunction[i, 1]
        , int: d = defEdgesInFunction[i, 2]
        }
    in if defsetOfDatum[d] subset phiInstrMatches
       then forall (o in operationsCoveredByMatch[min(defsetOfDatum[d])])
            ( oplace[o] = b )
       else true
       endif
  );

% If for any two given blocks p and q, and fallThroughBlockOfMatch contains
% [_, p, q] but does not contain [_, p, q'] or [_, p', q], then succ[p] = q
% can only help, never hurt.
%
% Eq. 6.26 in dissertation
constraint
  let { array[allBlocksInFunction] of set of allBlocksInFunction: fwd =
          array1d( allBlocksInFunction
                 , [ { s | i in index_set_1of2(fallThroughBlockOfMatch)
                         , m in {fallThroughBlockOfMatch[i, 1]}
                         , s in {fallThroughBlockOfMatch[i, 2]}
                           where entryBlockOfMatch[m] = {b}
                     }
                   | b in allBlocksInFunction
                   ]
                 )
      , array[allBlocksInFunction] of set of allBlocksInFunction: bwd =
          array1d( allBlocksInFunction
                 , [ { s | i in index_set_1of2(fallThroughBlockOfMatch)
                         , m in {fallThroughBlockOfMatch[i, 1]}
                         , s in entryBlockOfMatch[m]
                           where fallThroughBlockOfMatch[i, 2] = b
                     }
                   | b in allBlocksInFunction
                   ]
                 )
      }
  in forall ( p in allBlocksInFunction
            , q in allBlocksInFunction
            )
     ( if fwd[p] = {q} /\ bwd[q] = {p}
       then succ[p] = q
       else true
       endif
     );

% For non-phi matches, no spanned blocks:
% if selected, blocks of used and defined data must be equal.
%
% Eqs. 6.18-6.20 in dissertation
```

```
constraint
  forall ( m in allMatches diff phiInstrMatches
           where card(spannedBlocksInMatch[m]) = 0
         )
  ( let { array[int] of int: uses = [p | p in operandsUsedByMatch[m]]
        , array[int] of int: defs = [p | p in operandsDefinedByMatch[m]]
        }
    in forall ( i in index_set(uses)
              , j in index_set(uses)
                where i < j
              )
       ( sel[m] -> uplace[uses[i]] = uplace[uses[j]] )
       /\
       forall ( i in index_set(defs)
              , j in index_set(defs)
                where i < j
              )
       ( sel[m] -> dplace[Alt(defs[i])] = dplace[Alt(defs[j])] )
       /\
       forall ( i in index_set(uses)
              , j in index_set(defs)
              )
       ( sel[m] -> uplace[uses[i]] = dplace[Alt(defs[j])] )
  );

% For non-phi matches, spanned blocks:
% if selected, blocks of use of inputs must be all equal.
%
% Eq.6.21 in dissertation
constraint
  forall ( m in allMatches diff phiInstrMatches
           where card(spannedBlocksInMatch[m]) > 0
         )
  ( let { array[int] of int: use = [ p | p in operandsUsedByMatch[m]
                                         diff
                                         operandsDefinedByMatch[m]
                                   ]
        }
    in forall ( i in index_set(use)
              , j in index_set(use)
                where i < j
              )
       ( sel[m] -> uplace[use[i]] = uplace[use[j]] )
  );

% [MC 2]
%
% If all matches in the matchset covering a particular operation uses some
% datum d as input, then d cannot be placed in the intermediate value nor
% killed location.
%
% Eq.6.22 in dissertation
constraint
  forall ( M in matchsetOfOp )
  ( let { int: mo = min(M)
```

```
        , set of int: uses1 = operandsUsedByMatch[mo] diff
                              operandsDefinedByMatch[mo]
        }
    in forall (use1 in uses1)
       ( let { set of int: data1 = operandAlternatives[use1] }
         in if forall (d in data1)
               ( not (d in statesInFunction) )
               /\
               forall ( m in M where m != mo )
               ( exists ( use2 in operandsUsedByMatch[m] diff
                                  operandsDefinedByMatch[m]
                        )
                 ( operandAlternatives[use2] = data1 )
               )
               /\
               forall (d in data1)
               ( not valueMustBeAvailable(d) )
            then exists (d in data1)
                 ( loc[d] != locValueForInt /\ loc[d] != locValueForKilled )
            else true
            endif
       )
  );

% If all matches in the matchset that defines a non-state datum d, are
% active, and define it in an exterior operand, then d must not be placed
% in the intermediate value nor killed location.
%
% This constraint does not dominate [MC 2], nor vice versa.
%
% Eq.6.23 in dissertation
constraint
  forall (d in allDataInFunction diff statesInFunction )
  ( if valueMustBeAvailable(d)
    then loc[d] != locValueForInt /\ loc[d] != locValueForKilled
    else true
    endif
  );

% If an exterior operand does not take its min value, then its match must
% be selected and hence the datum cannot be in the intermediate value nor
% killed location.
%
% Eq.6.29 in dissertation
constraint
  forall ( m in allMatches
         , p in operandsUsedByMatch[m] intersect operandsExteriorToMatch[m]
                where card(operandAlternatives[p]) > 1
                   /\ not valueOfOpMustBeAvailable(p)
         )
  ( alt[p] != min(operandAlternatives[p])
    ->
    (loc[Alt(p)] != locValueForInt /\ loc[Alt(p)] != locValueForKilled)
  );
```

```
% Constrain the location of d to be where its definers can put it.
%
% Eq.6.25 in dissertation
constraint
  forall (d in allDataInFunction diff statesInFunction)
  ( let { array[int] of int: P =
            [p | m in defsetOfDatum[d] diff killInstrMatches
               , p in operandsDefinedByMatch[m]
                 where d in operandAlternatives[p]
            ]
        , array[int] of int: I =
            [ i | i in index_set_1of2(validDataLocRangesInMatch)
                  where validDataLocRangesInMatch[i, 2] in {p | p in P}
            ]
        }
    in if length(P) = length(I)
       then let { set of int: L =
                    { l | l in canonicalDataLocs
                        , i in I
                          where l in validDataLocRangesInMatch[i, 3]
                                     ..
                                     validDataLocRangesInMatch[i, 4]
                    }
                }
            in if card(L) < card(canonicalDataLocs)
               then loc[d] in L union {locValueForInt, locValueForKilled}
               else true
               endif
       else true
       endif
  );

% Constrain the location of d to be where its users can access it. Valid for
% such d that are used at least once.
%
% Eq.6.24 in dissertation
constraint
  forall (d in allDataInFunction)
  ( let { array[int] of int: P =
            [ p | p in array_union(operandsUsedByMatch) diff killOperands
                  where d in operandAlternatives[p]
            ]
        , array[int] of int: I =
            [ i | i in index_set_1of2(validDataLocRangesInMatch)
                  where validDataLocRangesInMatch[i, 2] in {p | p in P}
            ]
        }
    in if length(P) > 0 /\ length(P) = length(I)
       then let { set of int: L =
                    { l | l in canonicalDataLocs
                        , i in I
                          where l in validDataLocRangesInMatch[i, 3]
                                     ..
                                     validDataLocRangesInMatch[i, 4]
                    }
```

```
                }
            in if card(L) < card(canonicalDataLocs)
               then loc[d] in L union {locValueForInt, locValueForKilled}
               else true
               endif
       else true
       endif
  );


%==================
% SOLVE AND OUTPUT
%==================

solve
  :: seq_search(
       [ % Try the smallest cost for the operation with the largest
         % difference between its two smallest values.
         int_search(opcosts, max_regret, indomain_min, complete)
         % Remaining decisions are left to the solver.
       ])
  minimize totalcost;

% oplace, omatch, dmatch are handy for debugging
output [ "sel=", show(sel), "\n"
       , "alt=", show([ if not fix(sel[matchOfP[p]])
                        then datumValueForNull
                        else alt[p] endif
                      | p in allOperands
                      ]), "\n"
       , "dplace=", show(dplace), "\n"
       , "loc=", show(loc), "\n"
       , "place=", show([ if not fix(sel[m])
                          then blockValueForNull
                          else if card(operationsCoveredByMatch[m]) = 0
                               then min(entryBlockOfMatch[m])
                               else oplace[opOfM[m]]
                               endif
                          endif
                        | m in allMatches
                        ]), "\n"
       , "oplace=", show(oplace), "\n"
       , "omatch=", show(omatch), "\n"
       , "dmatch=", show(dmatch), "\n"
       , "succ=", show(succ), "\n"
       , "entry=", show(entryBlockOfFunction), "\n"
       , "block_value_for_null=", show(blockValueForNull), "\n"
       , "loc_value_for_int=", show(locValueForInt), "\n"
       , "loc_value_for_killed=", show(locValueForKilled), "\n"
       , "alt_value_for_null=", show(datumValueForNull), "\n"
       , "cost=", show(execFrequencyGCD * totalcost), "\n"
       ];
```

References

[1] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. “Efficient and Language-Independent Mobile Programs”. In: *Proceedings of PLDI’96*. ACM, 1996, pp. 127–136.

[2] A. Aggoun and N. Beldiceanu. “Extending CHIP in Order to Solve Complex Scheduling and Placement Problems”. In: *Mathematical and Computer Modelling* 17.7 (1993), pp. 57–73.

[3] M. Ahn, J. M. Youn, Y. Choi, D. Cho, and Y. Paek. “Iterative Algorithm for Compound Instruction Selection with Register Coalescing”. In: *Proceedings of DSD’09*. IEEE Computer Society, 2009, pp. 513–520.

[4] A. V. Aho and S. C. Johnson. “Optimal Code Generation for Expression Trees”. In: *Journal of the ACM* 23.3 (1976), pp. 488–501.

[5] A. V. Aho, S. C. Johnson, and J. D. Ullman. “Code Generation for Expressions with Common Subexpressions”. In: *Proceedings of POPL’76*. ACM, 1976, pp. 19–31.

[6] A. V. Aho and M. J. Corasick. “Efficient String Matching: An Aid to Bibliographic Search”. In: *Communications of the ACM* 18.6 (1975), pp. 333–340.

[7] A. V. Aho and M. Ganapathi. “Efficient Tree Pattern Matching: An Aid to Code Generation”. In: *Proceedings of POPL’85*. ACM, 1985, pp. 334–340.

[8] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. “Code Generation Using Tree Matching and Dynamic Programming”. In: *ACM Transactions on Programming Languages and Systems* 11.4 (1989), pp. 491–516.

[9] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools*. 2nd ed. Pearson, 2006. Chap. 4, pp. 197–209.

[10] P. Aigrain, S. L. Graham, R. R. Henry, M. K. McKusick, and E. Pelegrı-Llopart. “Experience with a Graham-Glanville Style Code Generator”. In: *Proceedings of CC’84*. ACM, 1984, pp. 13–24.

[11] O. Almer, R. Bennett, I. Böhm, A. Murray, X. Qu, M. Zuluaga, B. Franke, and N. Topham. “An End-to-End Design Flow for Automated Instruction Set Extension and Complex Instruction Selection based on GCC”. In: *Proceedings of GROW’09*. 2009, pp. 49–60.

[12] U. Ammann, K. V. Nori, K. Jensen, and H. Nägeli. *The PASCAL (P) Compiler Implementation Notes*. Tech. rep. Eidgenössische Technishe Hochschule, Zürich, Switzerland: Instituts für Informatik, 1974.

[13] U. Ammann. *On Code Generation in a PASCAL Compiler*. Tech. rep. Eidgenössische Technishe Hochschule, Zürich, Switzerland: Instituts für Informatik, 1977.

[14] B. Anckaert, B. Sutter, D. Chanet, and K. Bosschere. "Steganography for Executables and Code Transformation Signatures". In: *Proceedings of ICICS'05*. Springer, 2005, pp. 425–439.

[15] P. Andrade. *GNU lightning*. 2015. URL: `www.gnu.org/software/lightning` (accessed 2015-06-03).

[16] A. Appel, J. Davidson, and N. Ramsey. *The Zephyr Compiler Infrastructure*. Tech. rep. Charlottesville, Virginia, USA: University of Virginia, 1998.

[17] P. Arató, S. Juhász, Z. A. Mann, A. Orbán, and D. Papp. "Hardware-Software Partitioning in Embedded System Design". In: *Proceedings of ISP'03*. IEEE Computer Society, 2003, pp. 197–202.

[18] G. Araujo and S. Malik. "Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures". In: *Proceedings of ISSS'95*. ACM, 1995, pp. 36–41.

[19] G. Araujo, S. Malik, and M. T.-C. Lee. "Using Register-Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures". In: *Proceedings of DAC'96*. ACM, 1996, pp. 591–596.

[20] *ARM Cortex-M7 Devices: Generic User Guide*. ARM DUI 0646A. ARM. Mar. 19, 2015.

[21] *ARM11 MPCore Processor*. ARM DDI 0360F. Version r2p0. ARM. Oct. 15, 2018.

[22] M. Arnold. *Matching and Covering with Multiple-Output Patterns*. Tech. rep. Delft, The Netherlands: Delft University of Technology, 1999.

[23] M. Arnold and H. Corporaal. "Automatic Detection of Recurring Operation Patterns". In: *Proceedings of CODES'99*. ACM, 1999, pp. 22–26.

[24] M. Arnold and H. Corporaal. "Designing Domain-Specific Processors". In: *Proceedings of CODES'01*. ACM, 2001, pp. 61–66.

[25] M. A. Arslan. "Code Generation for Custom Architectures using Constraint Programming". Doctoral thesis. Lund, Sweden: Lund University, 2016.

[26] M. A. Arslan and K. Kuchcinski. "Instruction Selection and Scheduling for DSP Kernels". In: *Microprocessors and Microsystems* 38.8, Part A (2014), pp. 803–813.

[27] M. A. Arslan and K. Kuchcinski. "Instruction Selection and Scheduling for DSP Kernels on Custom Architectures". In: *Proceedings of DSD'13*. IEEE Computer Society, 2013.

[28] K. Atasu, G. Dündar, and C. özturan. "An Integer Linear Programming Approach for Identifying Instruction-Set Extensions". In: *Proceedings of CODES+ISSS'05*. ACM, 2005, pp. 172–177.

[29] K. Atasu, L. Pozzi, and P. Ienne. "Automatic Application-Specific Instruction-Set Extensions Under Microarchitectural Constraints". In: *Proceedings of DAC'03*. ACM, 2003, pp. 256–261.

[30] M. Auslander and M. Hopkins. "An Overview of the PL.8 Compiler". In: *Proceedings of CC'82*. ACM, 1982, pp. 22–31.

[31] M. W. Bailey and J. W. Davidson. "Automatic Detection and Diagnosis of Faults in Generated Code for Procedure Calls". In: *Transactions on Software Engineering* 29.11 (2003), pp. 1031–1042.

[32] A. Balachandran, D. M. Dhamdhere, and S. Biswas. "Efficient Retargetable Code Generation Using Bottom-Up Tree Pattern Matching". In: *Computer Languages* 15.3 (1990), pp. 127–140.

[33] M. Balakrishnan, P. C. P. Bhatt, and B. B. Madan. "An Efficient Retargetable Microcode Generator". In: *Proceedings of MICRO'86*. ACM, 1986, pp. 44–53.

[34] S. Bansal and A. Aiken. "Automatic Generation of Peephole Superoptimizers". In: *Proceedings of ASPLOS'06*. ACM, 2006, pp. 394–403.

[35] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling*. Kluwer Academic Publishers, 2001.

[36] S. Bashford and R. Leupers. "Constraint Driven Code Selection for Fixed-Point DSPs". In: *Proceedings of DAC'99*. ACM, 1999, pp. 817–822.

[37] L. Bauer, M. Shafique, and J. Henkel. "Run-Time Instruction Set Selection in a Transmutable Embedded Processor". In: *Proceedings of DAC'08*. IEEE Computer Society, 2008, pp. 56–61.

[38] A. Bednarski and C. W. Kessler. "Optimal Integrated VLIW Code Generation with Integer Linear Programming". In: *Proceedings of Euro-Par'06*. Springer, 2006, pp. 461–472.

[39] P. van Beek. "Backtrack Search Algorithms". In: *Handbook of Constraint Programming*. Elsevier, 2006. Chap. 4, pp. 85–134.

[40] M. O. Beg. "Combinatorial Problems in Compiler Optimization". Doctoral thesis. Ontario, Canada: University of Waterloo, 2013.

[41] N. Beldiceanu and M. Carlsson. "Sweep as a Generic Pruning Technique Applied to the Non-Overlapping Rectangles Constraint". In: *Proceedings of CP'01*. Springer, 2001, pp. 377–391.

[42] N. Beldiceanu and E. Contejean. "Introducing Global Constraints in CHIP". In: *Mathematical and Computer Modelling* 20.12 (1994), pp. 97–123.

[43] E. Bendersky. *A Deeper Look into the LLVM Code Generator: Part 1*. Feb. 25, 2013. URL: `eli.thegreenplace.net/2013/02/25/a-deeper-look-into-the-llvm-code-generator-part-1` (accessed 2013-05-10).

[44] R. V. Bennett, A. C. Murray, B. Franke, and N. Topham. "Combining Source-to-Source Transformations and Processor Instruction Set Extensions for the Automated Design-Space Exploration of Embedded Systems". In: *Proceedings of LCTES'07*. ACM, 2007, pp. 83–92.

[45] C. Bessiere. "Constraint Propagation". In: *Handbook of Constraint Programming*. Elsevier, 2006. Chap. 3, pp. 29–83.

[46] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009.

[47] I. Boehm. *HBURG*. 2007. URL: `www.bytelabs.org/hburg.html` (accessed 2014-02-11).

[48] B. Borchardt. "Code Selection by Tree Series Transducers". In: *Proceedings of CIAA'04*. Springer, 2004, pp. 57–67.

[49] A. Bougacha. *[LLVMdev] [RFC] Integer Saturation Intrinsics*. 2015-01-14. URL: `groups.google.com/forum/#!topic/llvm-dev/fHThmnh8zkI` (accessed 2015-06-09).

[50] D. Boulytchev. "BURS-Based Instruction Set Selection". In: *Proceedings of PSI'06*. Springer, 2007, pp. 431–437.

[51] D. Boulytchev and D. Lomov. "An Empirical Study of Retargetable Compilers". In: *Proceedings of PSI'01*. Springer, 2001, pp. 328–335.

[52] F. Brandner. "Completeness of Automatically Generated Instruction Selectors". In: *Proceedings of ASAP'10*. IEEE Computer Society, 2010, pp. 175–182.

[53] F. Brandner, D. Ebner, and A. Krall. "Compiler Generation from Structural Architecture Descriptions". In: *Proceedings of CASES'07*. ACM, 2007, pp. 13–22.

[54] M. Braun, S. Buchwald, and A. Zwinkau. "Firm—A Graph-Based Intermediate Representation". In: *Proceedings of WIR'11*. 2011, pp. 61–68.

[55] M. Bravenboer and E. Visser. "Rewriting Strategies for Instruction Selection". In: *Proceedings of RTA'02*. Springer, 2002, pp. 237–251.

[56] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. "Instruction Generation and Regularity Extraction for Reconfigurable Processors". In: *Proceedings of CASES'02*. ACM, 2002, pp. 262–269.

[57] P. Brisk, A. Nahapetian, and M. Sarrafzadeh. "Instruction Selection for Compilers That Target Architectures with Echo Instructions". In: *Proceedings of M-SCOPES'04*. Springer, 2004, pp. 229–243.

[58] P. Brown. "A Survey of Macro Processors". In: *Annual Review in Automatic Programming* 6.2 (1969), pp. 37–88.

[59] J. Bruno and R. Sethi. "Code Generation for a One-Register Machine". In: *Journal of the ACM* 23.3 (1976), pp. 502–510.

[60] S. Buchwald and A. Zwinkau. "Instruction Selection by Graph Transformation". In: *Proceedings of CASES'10*. ACM, 2010, pp. 31–40.

[61] J. Cai, R. Paige, and R. Tarjan. "More Efficient Bottom-Up Multi-pattern Matching in Trees". In: *Theoretical Computer Science* 106.1 (1992), pp. 21–60.

[62] P. Canalda, L. Cognard, A. Despland, M. Jourdan, M. Mazaud, D. Parigot, F. Thomasset, and D. de Voluceau. *PAGODE: A Realistic Back-End Generator*. Tech. rep. Rocquencourt, France: INRIA, 1995.

[63] Z. Cao, Y. Dong, and S. Wang. "Compiler Backend Generation for Application Specific Instruction Set Processors". In: *Proceedings of APLAS'11*. Springer, 2011, pp. 121–136.

[64] R. Castañeda Lozano, M. Carlsson, G. Hjort Blindell, and C. Schulte. "Combinatorial Spill Code Optimization and Ultimate Coalescing". In: *Proceedings of LCTES'14*. ACM, 2014, pp. 23–32.

[65] R. Castañeda Lozano, M. Carlsson, G. Hjort Blindell, and C. Schulte. "Register Allocation and Instruction Scheduling in Unison". In: *Proceedings of CC'16*. ACM, 2016, pp. 263–264.

[66] R. Castañeda Lozano, G. Hjort Blindell, M. Carlsson, F. Drejhammar, and C. Schulte. "Constraint-based Code Generation". In: *Proceedings of M-SCOPES'13*. Springer, 2013, pp. 93–95.

[67] R. G. Cattell. "Automatic Derivation of Code Generators from Machine Descriptions". In: *Transactions on Programming Languages and Systems* 2.2 (1980), pp. 173–190.

[68] R. G. G. Cattell. *A Survey and Critique of Some Models of Code Generation*. Tech. rep. Pittsburgh, Pennsylvania, USA: School of Computer Science, Carnegie Mellon University, 1979.

[69] R. G. G. Cattell. "Formalization and Automatic Derivation of Code Generators". Doctoral thesis. Pittsburgh, Pennsylvania, USA: Carnegie Mellon University, 1978.

[70] R. G. Cattell, J. M. Newcomer, and B. W. Leverett. "Code Generation in a Machine-Independent Compiler". In: *Proceedings of CC'79*. ACM, 1979, pp. 65–75.

[71] P. E. Ceruzzi. *A History of Modern Computing*. 2nd ed. MIT Press, 2003.

[72] D. R. Chase. "An Improvement to Bottom-Up Tree Pattern Matching". In: *Proceedings of POPL'87*. ACM, 1987, pp. 168–177.

[73] T. Chen, F. Lai, and R. Shang. "A Simple Tree Pattern Matching Algorithm for Code Generator". In: *Proceedings of COMPSAC'95*. IEEE Computer Society, 1995, pp. 162–167.

[74] D. Cho, A. Ravi, G.-R. Uh, and Y. Paek. "Instruction Re-selection for Iterative Modulo Scheduling on High Performance Multi-issue DSPs". In: *Emerging Directions in Embedded and Ubiquitous Computing*. Springer, 2006, pp. 741–754.

[75] T. W. Christopher, P. J. Hatcher, and R. C. Kukuk. "Using Dynamic Programming to Generate Optimized Code in a Graham-Glanville Style Code Generator". In: *Proceedings of CC'84*. ACM, 1984, pp. 25–36.

[76] G. G. Chu. "Improving Combinatorial Optimization". Doctoral thesis. The University of Melbourne, Australia, 2011.

[77] G. Chu and P. J. Stuckey. "Dominance breaking constraints". In: *Constraints* 20.2 (2015), pp. 155–182.

[78] G. Chu and P. J. Stuckey. *Structure Based Extended Resolution for Constraint Programming*. June 19, 2013. URL: arxiv.org/abs/1306.4418.

[79] N. Clark, A. Hormati, S. Mahlke, and S. Yehia. "Scalable Subgraph Mapping for Acyclic Computation Accelerators". In: *Proceedings of CASES'06*. ACM, 2006, pp. 147–157.

[80] N. Clark, H. Zhong, and S. Mahlke. "Processor Acceleration Through Automated Instruction Set Customization". In: *Proceedings of MICRO'03*. IEEE Computer Society, 2003, pp. 129–140.

[81] C. Click and M. Paleczny. "A Simple Graph-based Intermediate Representation". In: *Proceedings of IR'95*. ACM, 1995, pp. 35–49.

[82] C. Click. "Combining Analyses, Combining Optimizations". Doctoral thesis. Houston, Texas, USA: Rice University, 1995. Chap. 7.

[83] R. Cole and R. Hariharan. "Tree Pattern Matching and Subset Matching in Randomized $O(n \log^3 m)$ Time". In: *Proceedings of STOC'97*. ACM, 1997, pp. 66–75.

[84] J. Cong, Y. Fan, G. Han, and Z. Zhang. "Application-Specific Instruction Generation for Configurable Processor Architectures". In: *Proceedings of FPGA'04*. ACM, 2004, pp. 183–189.

[85] S. A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of STOC'71*. ACM, 1971, pp. 151–158.

[86] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. "An Improved Algorithm for Matching Large Graphs". In: *Proceedings of GbRPR'01*. Springer, 2001, pp. 149–159.

[87] R. Cordone, F. Ferrandi, D. Sciuto, and R. Wolfler Calvo. "An Efficient Heuristic Approach to Solve the Unate Covering Problem". In: *Proceedings of DATE'00*. IEEE Computer Society, 2000, pp. 364–371.

[88] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 3rd ed. MIT Press, 2009.

[89] *CoSy Compilers: Overview of Construction and Operation*. ACE Associated Compiler Experts. 2003.

[90] T. Crick, M. Brain, M. Vos, and J. Fitch. "Generating Optimal Code Using Answer Set Programming". In: *Proceedings of LPNMR'09*. Springer, 2009, pp. 554–559.

[91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Transactions on Programming Languages and Systems* 13.4 (1991), pp. 451–490.

[92] A. Darwiche and K. Pipatsrisawat. "Complete Algorithms". In: *Handbook of Satisfiability*. IOS Press, 2009. Chap. 3, pp. 99–130.

[93] J. W. Davidson and C. W. Fraser. "Code Selection Through Object Code Optimization". In: *ACM Transactions on Programming Languages and Systems* 6.4 (1984), pp. 505–526.

[94] J. W. Davidson and C. W. Fraser. "Eliminating Redundant Object Code". In: *Proceedings of POPL'82*. ACM, 1982, pp. 128–132.

[95] J. W. Davidson and C. W. Fraser. "The Design and Application of a Retargetable Peephole Optimizer". In: *Transactions on Programming Languages and Systems* 2.2 (1980), pp. 191–202.

[96] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J.-C. Régin, and P. Schaus. "Compact-Table: Efficiently Filtering Table Constraints with Reversible Sparse Bit-Sets". In: *Proceedings of CP'16*. Springer, 2016, pp. 207–223.

[97] A. Despland, M. Mazaud, and R. Rakotozafy. "Code Generator Generation Based on Template-Driven Target Term Rewriting". In: *Proceedings of RTA'87*. Springer, 1987, pp. 105–120.

[98] A. Despland, M. Mazaud, and R. Rakotozafy. "Using Rewriting Techniques to Produce Code Generators and Proving Them Correct". In: *Science of Computer Programming* 15.1 (1990), pp. 15–54.

[99] L. P. Deutsch and A. M. Schiffman. "Efficient Implementation of the Smalltalk-80 System". In: *Proceedings of POPL'84*. ACM, 1984, pp. 297–302.

[100] J. Dias and N. Ramsey. "Automatically Generating Instruction Selectors Using Declarative Machine Descriptions". In: *Proceedings of POPL'10*. ACM, 2010, pp. 403–416.

[101] J. Dias and N. Ramsey. "Converting Intermediate Code to Assembly Code Using Declarative Machine Descriptions". In: *Proceedings of CC'06*. Springer, 2006, pp. 217–231.

[102] A. Dold, T. Gaul, V. Vialard, and W. Zimmermann. "ASM-Based Mechanized Verification of Compiler Back-Ends". In: *Proceedings of ASM'98*. Springer, 1998, pp. 50–67.

[103] M. K. Donegan, R. E. Noonan, and S. Feyock. "A Code Generator Generator Language". In: *Proceedings of CC'79*. ACM, 1979, pp. 58–64.

[104] M. K. Donegan. "An Approach to the Automatic Generation of Code Generators". Doctoral thesis. Houston, Texas, USA: Rice University, 1973.

[105] M. Dorigo and T. Stützle. “Ant Colony Optimization: Overview and Recent Advances”. In: *Handbook of Metaheuristics*. 2nd ed. Vol. 146. Springer, 2010. Chap. 8, pp. 227–263.

[106] M. Dubiner, Z. Galil, and E. Magen. “Faster Tree Pattern Matching”. In: *Journal of the ACM* 41.2 (1994), pp. 205–213.

[107] J. Earley. “An Efficient Context-Free Parsing Algorithm”. In: *Communications of the ACM* 13.2 (1970), pp. 94–102.

[108] D. Ebner, F. Brandner, B. Scholz, A. Krall, P. Wiedermann, and A. Kadlec. “Generalized Instruction Selection Using SSA-Graphs”. In: *Proceedings of LCTES’08*. ACM, 2008, pp. 31–40.

[109] E. Eckstein, O. König, and B. Scholz. “Code Instruction Selection Based on SSA-Graphs”. In: *Proceedings of M-SCOPES’03*. Springer, 2003, pp. 49–65.

[110] T. J. Edler von Koch, I. Böhm, and B. Franke. “Integrated Instruction Selection and Register Allocation for Compact Code Generation Exploiting Freeform Mixing of 16- and 32-bit Instructions”. In: *Proceedings of CGO’10*. ACM, 2010, pp. 180–189.

[111] B. Efron and R. Tibshirani. *An Introduction to the Boostrap*. Chapman and Hall, 1994.

[112] M. Elson and S. T. Rake. “Code-Generation Technique for Large-Language Compilers”. In: *IBM Systems Journal* 9.3 (1970), pp. 166–188.

[113] H. Emmelmann, F.-W. Schröer, and R. Landwehr. “BEG: A Generator for Efficient Back Ends”. In: *Proceedings of PLDI’89*. ACM, 1989, pp. 227–237.

[114] H. Emmelmann. “Code Selection by Regularly Controlled Term Rewriting”. In: *Code Generation—Concepts, Tools, Techniques*. Springer, 1992, pp. 3–29.

[115] H. Emmelmann. “Testing Completeness of Code Selector Specifications”. In: *Proceedings of CC’92*. Springer, 1992, pp. 163–175.

[116] J. Engelfriet, Z. Fülöp, and H. Vogler. “Bottom-Up and Top-Down Tree Series Transformations”. In: *Journal of Automata, Languages and Combinatorics* 7.1 (July 2001), pp. 11–70.

[117] D. R. Engler. “VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System”. In: *Proceedings of PLDI’96*. ACM, 1996, pp. 160–170.

[118] D. R. Engler and T. A. Proebsting. “DCG: An Efficient, Retargetable Dynamic Code Generation System”. In: *Proceedings of ASPLOS’94*. ACM, 1994, pp. 263–272.

[119] M. V. Eriksson, O. Skoog, and C. W. Kessler. “Optimal vs. Heuristic Integrated Code Generation for Clustered VLIW Architectures”. In: *Proceedings of M-SCOPES’08*. ACM, 2008, pp. 11–20.

[120] M. Eriksson and C. Kessler. “Integrated Code Generation for Loops”. In: *ACM Transactions on Embedded Computing Systems* 11S.1 (June 2012), 19:1–19:24.

[121] M. A. Ertl. "Optimal Code Selection in DAGs". In: *Proceedings of POPL'99*. ACM, 1999, pp. 242–249.

[122] M. A. Ertl, K. Casey, and D. Gregg. "Fast and Flexible Instruction Selection with On-Demand Tree-Parsing Automata". In: *Proceedings of PLDI'06*. ACM, 2006, pp. 52–60.

[123] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. "Incremental Graph Pattern Matching". In: *Proceedings of SIGMOD'11*. ACM, 2011, pp. 925–936.

[124] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. "Graph Pattern Matching: From Intractable to Polynomial Time". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 264–275.

[125] S. Farfeleder, A. Krall, E. Steiner, and F. Brandner. "Effective Compiler Generation by Architecture Description". In: *Proceedings of LCTES'06*. ACM, 2006, pp. 145–152.

[126] R. Farrow. "Experience with an Attribute Grammar-based Compiler". In: *Proceedings of POPL'82*. ACM, 1982, pp. 95–107.

[127] A. Fauth, M. Freericks, and A. Knoll. "Generation of Hardware Machine Models from Instruction Set Descriptions". In: *Proceedings of the Workshop on VLSI Signal Processing*. IEEE Computer Society, 1993, pp. 242–250.

[128] A. Fauth, J. Van Praet, and M. Freericks. "Describing Instruction Set Processors Using nML". In: *Proceedings of EDTC'95*. IEEE Computer Society, 1995, pp. 503–507.

[129] A. Fauth, G. Hommel, A. Knoll, and C. Müller. "Global Code Selection of Directed Acyclic Graphs". In: *Proceedings of CC'94*. Springer, 1994, pp. 128–142.

[130] J. Feldman and D. Gries. "Translator Writing Systems". In: *Communications of the ACM* 11.2 (1968), pp. 77–113.

[131] C. Ferdinand, H. Seidl, and R. Wilhelm. "Tree Automata for Code Selection". In: *Acta Informatica* 31.9 (1994), pp. 741–760.

[132] M. Fernández and N. Ramsey. "Automatic Checking of Instruction Specifications". In: *Proceedings of ICSE'97*. ACM, 1997, pp. 326–336.

[133] T. Feydy and P. J. Stuckey. "Lazy Clause Generation Reengineered". In: *Proceedings of CP'09*. Springer, 2009, pp. 352–366.

[134] P. J. Fleming and J. J. Wallace. "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results". In: *Communications of the ACM* 29.3 (1986), pp. 218–221.

[135] A. Floch, C. Wolinski, and K. Kuchcinski. "Combined Scheduling and Instruction Selection for Processors with Reconfigurable Cell Fabric". In: *Proceedings of ASAP'10*. IEEE Computer Society, 2010, pp. 167–174.

[136] R. W. Floyd. “Algorithm 97: Shortest Path”. In: *Communications of the ACM* 5.6 (1962), p. 345.

[137] C. W. Fraser. “A Language for Writing Code Generators”. In: *Proceedings of PLDI’89*. ACM, 1989, pp. 238–245.

[138] C. W. Fraser and A. L. Wendt. “Automatic Generation of Fast Optimizing Code Generators”. In: *Proceedings of PLDI’88*. ACM, 1988, pp. 79–84.

[139] C. W. Fraser. “A Compact, Machine-Independent Peephole Optimizer”. In: *Proceedings of POPL’79*. ACM, 1979, pp. 1–6.

[140] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. “Engineering a Simple, Efficient Code-Generator Generator”. In: *Letters on Programming Languages and Systems* 1.3 (1992), pp. 213–226.

[141] C. W. Fraser, R. R. Henry, and T. A. Proebsting. “BURG—Fast Optimal Instruction Selection and Tree Parsing”. In: *SIGPLAN Notices* 27.4 (1992), pp. 68–76.

[142] C. W. Fraser and T. A. Proebsting. “Finite-State Code Generation”. In: *Proceedings of PLDI’99*. ACM, 1999, pp. 270–280.

[143] C. W. Fraser. “A Knowledge-Based Code Generator Generator”. In: *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*. ACM, 1977, pp. 126–129.

[144] C. W. Fraser. “Automatic Generation of Code Generators”. Doctoral thesis. New Haven, Connecticut, USA: Yale University, 1977.

[145] S. Fröhlich, M. Gotschlich, U. Krebelder, and B. Wess. “Dynamic Trellis Diagrams for Optimized DSP Code Generation”. In: *Proceedings of ISCAS’99*. IEEE Computer Society, 1999, pp. 492–495.

[146] B. Gallagher. *The State of the Art in Graph-Based Pattern Matching*. Tech. rep. Livermore, California, USA: Lawrence Livermore National Laboratory, 2006.

[147] C. Galuzzi and K. Bertels. “The Instruction-Set Extension Problem: A Survey”. In: *Transactions on Reconfigurable Technology and Systems* 4.2 (May 2011), 18:1–18:28.

[148] M. Ganapathi. “Prolog Based Retargetable Code Generation”. In: *Computer Languages* 14.3 (1989), pp. 193–204.

[149] M. Ganapathi. “Retargetable Code Generation and Optimization Using Attribute Grammars”. Doctoral thesis. Madison, Wisconsin, USA: The University of Wisconsin–Madison, 1980.

[150] M. Ganapathi and C. N. Fischer. “Affix Grammar Driven Code Generation”. In: *Transactions on Programming Languages and Systems* 7.4 (1985), pp. 560–599.

[151] M. Ganapathi and C. N. Fischer. “Description-Driven Code Generation Using Attribute Grammars”. In: *Proceedings of POPL’82*. ACM, 1982, pp. 108–119.

[152] M. Ganapathi and C. N. Fischer. *Instruction Selection by Attributed Parsing*. Tech. rep. Stanford, California, USA: Stanford University, 1984.

[153] M. Ganapathi, C. N. Fischer, and J. L. Hennessy. "Retargetable Compiler Code Generation". In: *ACM Computing Surveys* 14.4 (1982), pp. 573–592.

[154] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[155] C. H. Gebotys. "An Efficient Model for DSP Code Generation: Performance, Code Size, Estimated Energy". In: *Proceedings of ISSS'97*. IEEE Computer Society, 1997, pp. 41–47.

[156] F. Gecseg and M. Steinby. *Tree Automata*. Akadémiai Kiadó, 1984.

[157] D. Genin, J. De Moortel, D. Desmet, and E. Van de Velde. "System Design, Optimization and Intelligent Code Generation for Standard Digital Signal Processors". In: *Proceedings of ISCAS'90*. IEEE Computer Society, 1989, pp. 565–569.

[158] I. P. Gent, K. E. Petrie, and J.-F. Puget. "Symmetry in Constraint Programming". In: *Handbook of Constraint Programming*. Elsevier, 2006. Chap. 10, pp. 329–376.

[159] M. P. Gerlek, E. Stoltz, and M. Wolfe. "Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-driven SSA Form". In: *ACM Transactions on Programming Languages and Systems* 17.1 (1995), pp. 85–122.

[160] C. Gervet. "Constraints over Structured Domains". In: *Handbook of Constraint Programming*. Elsevier, 2006. Chap. 17, pp. 605–638.

[161] R. Giegerich. "A Formal Framework for the Derivation of Machine-Specific Optimizers". In: *Transactions on Programming Languages and Systems* 5.3 (1983), pp. 478–498.

[162] R. Giegerich and K. Schmal. "Code Selection Techniques: Pattern Matching, Tree Parsing, and Inversion of Derivors". In: *Proceedings of ESOP'88*. Springer, 1998, pp. 247–268.

[163] R. S. Glanville and S. L. Graham. "A New Method for Compiler Code Generation". In: *Proceedings of POPL'78*. Springer, 1978, pp. 231–254.

[164] E. I. Goldberg, L. P. Carloni, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. "Negative Thinking in Branch-and-Bound: The Case of Unate Covering". In: *Transactions of Computer-Aided Design of Integrated Ciruits and Systems* 19.3 (2006), pp. 281–294.

[165] K. J. Gough. *Bottom-Up Tree Rewriting Tool MBURG*. Tech. rep. Brisbane, Australia: Faculty of Information Technology, Queensland University of Technology, July 18, 1995.

[166] K. J. Gough and J. Ledermann. "Optimal Code-Selection using MBURG". In: *Proceedings of ACSC'97*. Sydney, Australia, 1997.

[167] K. Gough. "Reconceptualizing Bottom-Up Tree Rewriting". In: *Patterns, Programming and Everything*. Springer, 2012, pp. 31–44.

[168] S. L. Graham. "Table-Driven Code Generation". In: *Computer* 13.8 (1980), pp. 25–34.

[169] S. L. Graham, R. R. Henry, and R. A. Schulman. "An Experiment in Table Driven Code Generation". In: *Proceedings of CC'82*. ACM, 1982, pp. 32–43.

[170] T. Granlund and R. Kenner. "Eliminating Branches Using a Superoptimizer and the GNU C Compiler". In: *Proceedings of PLDI'92*. ACM, 1992, pp. 341–352.

[171] G. Grasso, N. Leone, and F. Ricca. "Answer Set Programming: Language, Applications and Development Tools". In: *Web Reasoning and Rule Systems*. Springer, 2013, pp. 19–34.

[172] Y. Guo, G. J. Smit, H. Broersma, and P. M. Heysters. "A Graph Covering Algorithm for a Coarse Grain Reconfigurable System". In: *Proceedings of LCTES'03*. ACM, 2003, pp. 199–208.

[173] S. Z. Hanono. "AVIV: A Retargetable Code Generator for Embedded Processors". Doctoral thesis. Cambridge, Massachusetts, USA: Massachusetts Institute of Technology, 1999.

[174] S. Hanono and S. Devadas. "Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator". In: *Proceedings of DAC'98*. ACM, 1998, pp. 510–515.

[175] D. R. Hanson and C. W. Fraser. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[176] R. M. Haralick and G. L. Elliott. "Increasing Tree Search Efficiency for Constraint Satisfaction Problems". In: *Artificial Intelligence* 14.3 (1980), pp. 263–313.

[177] W. H. Harrison. "A New Strategy for Code Generation the General-Purpose Optimizing Compiler". In: *Transactions Software Engineering* 5.4 (1979), pp. 367–373.

[178] T. Harwood, K. Kumar, and N. Bereton. *JBURG*. 2013. URL: `jburg.sourceforge.net` (accessed 2014-02-11).

[179] P. J. Hatcher and T. W. Christopher. "High-Quality Code Generation via Bottom-Up Tree Pattern Matching". In: *Proceedings of POPL'86*. ACM, 1986, pp. 119–130.

[180] P. Hatcher. "The Equational Specification of Efficient Compiler Code Generation". In: *Computer Languages* 16.1 (1991), pp. 81–95.

[181] P. Hatcher and J. W. Tuller. "Efficient Retargetable Compiler Code Generation". In: *Proceedings of ICCL'88*. IEEE Computer Society, 1988, pp. 25–30.

[182] R. R. Henry. *Encoding Optimal Pattern Selection in Table-Driven Bottom-Up Tree-Pattern Matcher*. Tech. rep. Seattle, Washington, USA: University of Washington, 1989.

[183] R. R. Henry. "Graham-Glanville Code Generators". Doctoral thesis. Berkeley, California, USA: EECS Department, University of California, May 1984.

[184] P. van Hentenryck, V. Saraswat, and Y. Deville. "Design, Implementation, and Evaluation of the Constraint Language `cc(FD)`". In: *The Journal of Logic Programming* 37.1 (1998), pp. 139–164.

[185] T. Hino, Y. Suzuki, T. Uchida, and Y. Itokawa. "Polynomial Time Pattern Matching Algorithm for Ordered Graph Patterns". In: *Proceedings of ILP'12*. Springer, 2012, pp. 86–101.

[186] G. Hjort Blindell. *Instruction Selection: Principles, Methods, and Applications*. Springer, 2016.

[187] G. Hjort Blindell. *Survey on Instruction Selection: An Extensive and Modern Literature Study*. Tech. rep. Stockholm, Sweden: KTH Royal Institute of Technology, 2013.

[188] G. Hjort Blindell, M. Carlsson, R. Castañeda Lozano, and C. Schulte. "Complete and Practical Univeral Instruction Selection". In: *ACM Transactions on Embedded Computing Systems* 16.5s (2017), 119:1–119:18.

[189] G. Hjort Blindell, R. Castañeda Lozano, M. Carlsson, and C. Schulte. "Modeling Universal Instruction Selection". In: *Proceedings of CP'15*. Springer, 2015, pp. 609–626.

[190] G. Hjort Blindell, C. Menne, and I. Sander. "Synthesizing Code for GPGPUs from Abstract Formal Models". In: *Languages, Design Methods, and Tools for Electronic System Design*. Vol. 361. Lecture Notes in Electrical Engineering. Springer, 2016, pp. 115–134.

[191] W.-J. van Hoeve. "The Alldifferent Constraint: A Survey". In: *Proceedings of the Annual Workshop of the ERCIM Working Group on Constraints*. 2001.

[192] W.-J. van Hoeve and I. Katriel. "Global Constraints". In: *Handbook of Constraint Programming*. Elsevier, 2006. Chap. 6, pp. 169–208.

[193] C. M. Hoffmann and M. J. O'Donnell. "Pattern Matching in Trees". In: *Journal of the ACM* 29.1 (1982), pp. 68–95.

[194] J. N. Hooker. "Resolution vs. Cutting Plane Solution of Inference Problems: Some Computational Experience". In: *Operations Research Letters* 7.1 (Feb. 1988), pp. 1–7.

[195] R. Hoover and K. Zadeck. "Generating Machine Specific Optimizing Compilers". In: *Proceedings of POPL'96*. ACM, 1996, pp. 219–229.

[196] A. Hormati, N. Clark, and S. Mahlke. "Exploiting Narrow Accelerators with Data-Centric Subgraph Mapping". In: *Proceedings of CGI'07*. IEEE Computer Society, 2007, pp. 341–353.

[197] R. N. Horspool. “An Alternative to the Graham-Glanville Code-Generation Method”. In: *Software* 4.3 (May 1987), pp. 33–39.

[198] I. Huang and A. M. Despain. “Synthesis of Application Specific Instruction Sets”. In: *Transactions on Computer Aided Design of Integrated Circuits and Systems* 14.6 (June 1995), pp. 663–675.

[199] *Intel Reports Record Third-Quarter Revenue of $14.6 Billion*. Intel. Oct. 14, 2014. URL: `newsroom.intel.com/news-releases/intel-reports-record-third-quarter-revenue-of-14-6-billion/` (accessed 2017-09-23).

[200] *Intel 64 and IA-32 Architectures: Software Developer's Manual*. Intel. Apr. 2015.

[201] J. Janoušek and J. Málek. “Target Code Selection by Tilling AST with the Use of Tree Pattern Pushdown Automaton”. In: *Proceedings of SLATE'14*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 159–165.

[202] D. B. Johnson. “Finding All the Elementary Circuits of a Directed Graph”. In: *SIAM Journal on Computing* 4.1 (1975), pp. 77–84.

[203] S. C. Johnson. “A Portable Compiler: Theory and Practice”. In: *Proceedings of POPL'78*. ACM, 1978, pp. 97–104.

[204] S. C. Johnson. “A Tour Through the Portable C Compiler”. In: *Unix Programmer's Manual*. 7th ed. Vol. 2B. AT&T Bell Laboratories, 1981. Chap. 33.

[205] R. Joshi, G. Nelson, and K. Randall. “Denali: A Goal-Directed Superoptimizer”. In: *Proceedings of PLDI'02*. ACM, 2002, pp. 304–314.

[206] R. Joshi, G. Nelson, and Y. Zhou. “Denali: A Practical Algorithm for Generating Optimal Code”. In: *Transactions on Programming Languages and Systems* 28.6 (2006), pp. 967–989.

[207] K. Kang. “A Study on Generating an Efficient Bottom-Up Tree Rewrite Machine for JBURG”. In: *Proceedings of ICCSA'04*. Springer, 2004, pp. 65–72.

[208] K. Kang and K. Choe. *On the Automatic Generation of Instruction Selector Using Bottom-Up Tree Pattern Matching*. Tech. rep. Daejeon, South Korea: Korea Advanced Institute of Science and Technology, 1995.

[209] R. M. Karp, R. E. Miller, and A. L. Rosenberg. “Rapid Identification of Repeated Patterns in Strings, Trees and Arrays”. In: *Proceedings of STOC'72*. ACM, 1972, pp. 125–136.

[210] R. Kastner, A. Kaplan, S. O. Memik, and E. Bozorgzadeh. “Instruction Generation for Hybrid Reconfigurable Systems”. In: *Transactions on Design Automation of Electronic Systems* 7.4 (2002), pp. 605–627.

[211] L. G. Kaya and J. N. Hooker. “A Filter for the Circuit Constraint”. In: *Proceedings of CP'06*. Springer, 2006, pp. 706–710.

[212] C. W. Kessler and A. Bednarski. “A Dynamic Programming Approach to Optimal Integrated Code Generation”. In: *Proceedings of LCTES'01*. ACM, 2001, pp. 165–174.

[213] C. W. Kessler and A. Bednarski. “Optimal Integrated Code Generation for Clustered VLIW Architectures”. In: *Proceedings of LCTES/M-SCOPES'02*. ACM, 2002, pp. 102–111.

[214] P. B. Kessler. “Discovering Machine-Specific Code Improvements”. In: *Proceedings of CC'86*. ACM, 1986, pp. 249–254.

[215] R. R. Kessler. “PEEP: An Architectural Description Driven Peephole Optimizer”. In: *Proceedings of CC'84*. ACM, 1984, pp. 106–110.

[216] K. Keutzer. “DAGON: Technology Binding and Local Optimization by DAG Matching”. In: *Proceedings of DAC'87*. ACM, 1987, pp. 341–347.

[217] R. El-Khalil and A. D. Keromytis. “Hydan: Hiding Information in Program Binaries”. In: *Proceedings of ICICS'04*. Springer, 2004, pp. 187–199.

[218] U. Khedker. “Workshop on Essential Abstractions in GCC”. Lecture. GCC Resource Center, Department of Computer Science and Engineering, IIT Bombay. Bombay, India, June 30–July 3, 2012.

[219] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. “Optimization by Simulated Annealing”. In: *Science* 220.4598 (1983), pp. 671–680.

[220] D. E. Knuth. “On the Translation of Languages From Left to Right”. In: *Information and Control* 8 (6 Dec. 1965), pp. 607–639.

[221] D. E. Knuth. “Semantics of Context-Free Languages”. In: *Mathematical Systems Theory* 2.2 (1968), pp. 127–145.

[222] D. E. Knuth, J. H. J. Morris, and V. R. Pratt. “Fast Pattern Matching in Strings”. In: *SIAM Journal of Computing* 6.2 (1977), pp. 323–350.

[223] D. Koes. “Towards a More Principled Compiler: Register Allocation and Instruction Selection Revisited”. Doctoral thesis. Pittsburgh, Pennsylvania, USA: Carnegie Mellon University, 2009.

[224] D. R. Koes and S. C. Goldstein. “Near-Optimal Instruction Selection on DAGs”. In: *Proceedings of CGO'08*. ACM, 2008, pp. 45–54.

[225] R. E. Korf. “Optimal Rectangle Packing: New Results”. In: *Proceedings of ICAPS'04*. AAAI Press, 2004, pp. 142–149.

[226] W. Kreuzer, W. Gotschlich, and B. Wess. “REDACO: A Retargetable Data Flow Graph Compiler for Digital Signal Processors”. In: *Proceedings of ICSPAT'96*. Miller Freeman, 1996, pp. 742–746.

[227] A. Krishnaswamy and R. Gupta. “Profile Guided Selection of ARM and Thumb Instructions”. In: *Proceedings of LCTES/M-SCOPES'02*. ACM, 2002, pp. 56–64.

[228] E. B. Krissinel and K. Henrick. “Common Subgraph Isomorphism Detection by Backtracking Search”. In: *Software–Practice & Experience* 34.6 (2004), pp. 591–607.

[229] D. W. Krumme and D. H. Ackley. “A Practical Method for Code Generation Based on Exhaustive Search”. In: *Proceedings of CC’82*. ACM, 1982, pp. 185–196.

[230] P. Kulkarni, W. Zhao, S. Hines, D. Whalley, X. Yuan, R. v. Engelen, K. Gallivan, J. Hiser, J. Davidson, B. Cai, M. Bailey, H. Moon, K. Cho, and Y. Paek. “VISTA: VPO Interactive System for Tuning Applications”. In: *Transactions on Embedded Computer Systems* 5.4 (Nov. 2006), pp. 819–863.

[231] A. H. Land and A. G. Doig. “An Automatic Method of Solving Discrete Programming Problems”. In: *Econometrica* 28.3 (1960), pp. 497–520.

[232] R. Landwehr, H.-S. Jansohn, and G. Goos. “Experience with an Automatic Code Generator Generator”. In: *Proceedings of CC’82*. ACM, 1982, pp. 56–66.

[233] M. Langevin and E. Cerny. “An Automata-Theoretic Approach to Local Microcode Generation”. In: *Proceedings of EDAC’93*. IEEE Computer Society, 1993, pp. 94–98.

[234] D. Lanneer, F. Catthoor, G. Goossens, M. Pauwels, J. Van Meerbergen, and H. De Man. “Open-Ended System for High-Level Synthesis of Flexible Signal Processors”. In: *Proceedings of EURO-DAC’90*. IEEE Computer Society, 1990, pp. 272–276.

[235] C. Lattner and V. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of CGO’04*. IEEE Computer Society, 2004, pp. 75–86.

[236] J.-L. Laurière. “A Language and a Program for Stating and Solving Combinatorial Problems”. In: *Artificial Intelligence* 10.1 (1978), pp. 29–127.

[237] Y. C. Law and J. H. M. Lee. “Global Constraints for Integer and Set Value Precedence”. In: *Proceedings of CP’04*. Springer, 2004, pp. 362–376.

[238] C. Lecoutre. “STR2: Optimized Simple Tabular Reduction for Table Constraints”. In: *Constraints* 16.4 (2011), pp. 341–371.

[239] C. Lecoutre, C. Likitvivatanavong, and R. Yap. “STR3: A Path-optimal Filtering Algorithm for Table Constraints”. In: *Artifial Intelligence* 220 (2015), pp. 1–27.

[240] C. Lecoutre and R. Szymanek. “Generalized Arc Consistency for Positive Table Constraints”. In: *Proceedings of CP’06*. Springer, 2006, pp. 284–298.

[241] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems”. In: *MICRO’97*. IEEE, 1997, pp. 330–335.

[242] R. Leupers. “Code Generation for Embedded Processors”. In: *Proceedings of ISSS’00*. IEEE Computer Society, 2000, pp. 173–178.

[243] R. Leupers. “Code Selection for Media Processors with SIMD Instructions”. In: *Proceedings of DATE’00*. ACM, 2000, pp. 4–8.

[244] R. Leupers and S. Bashford. "Graph-Based Code Selection Techniques for Embedded Processors". In: *Transactions on Design Automation of Electronic Systems* 5 (4 2000), pp. 794–814. ISSN: 1084-4309.

[245] R. Leupers and P. Marwedel. "Instruction Selection for Embedded DSPs with Complex Instructions". In: *Proceedings of EURO-DAC/EURO-VHDL'96*. IEEE Computer Society, 1996, pp. 200–205.

[246] R. Leupers and P. Marwedel. "Retargetable Code Generation Based on Structural Processor Description". In: *Design Automation for Embedded Systems* 3.1 (1998), pp. 75–108.

[247] R. Leupers and P. Marwedel. *Retargetable Compiler Technology for Embedded Systems*. Kluwer Academic Publishers, 2001.

[248] R. Leupers and P. Marwedel. "Retargetable Generation of Code Selectors from HDL Processor Models". In: *Proceedings of EDTC'97*. IEEE Computer Society, 1997, pp. 140–144.

[249] R. Leupers and P. Marwedel. "Time-Constrained Code Compaction for DSPs". In: *Proceedings of ISSS'95*. ACM, 1995, pp. 54–59.

[250] B. W. Leverett, R. G. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz, and W. A. Wulf. "An Overview of the Production-Quality Compiler-Compiler Project". In: *Computer* 13.8 (1980), pp. 38–49.

[251] S. Liao, K. Keutzer, S. Tjiang, and S. Devadas. "A New Viewpoint on Code Generation for Directed Acyclic Graphs". In: *Transactions on Design Automation of Electronic Systems* 3.1 (1998), pp. 51–75.

[252] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. "Instruction Selection Using Binate Covering for Code Size Optimization". In: *Proceedings of ICCAD'95*. IEEE Computer Society, 1995, pp. 393–399.

[253] C. Liem, T. May, and P. Paulin. "Instruction-Set Matching and Selection for DSP and ASIP Code Generation". In: *Proceedings of EDAC/ETC/EUROASIC'94*. IEEE Computer Society, 1994, pp. 31–37.

[254] E. M. Loiola, N. M. Maia de Abreu, P. O. Boaventura-Netto, P. Hahn, and T. Querido. "A Survey for the Quadratic Assignment Problem". In: *European Journal of Operational Research* 176.2 (2007), pp. 657–690.

[255] A. Lopez-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. "A Fast and Simple Algorithm for Bounds Consistency of the All Different Constraint". In: *Proceedings of IJCAI'03*. Morgan Kaufmann Publishers Inc., 2003, pp. 245–250.

[256] M. Lorenz, R. Leupers, P. Marwedel, T. Drager, and G. Fettweis. "Low-Energy DSP Code Generation Using a Genetic Algorithm". In: *Proceedings of ICCD'01*. IEEE Computer Society, 2001, pp. 431–437.

[257] M. Lorenz and P. Marwedel. "Phase Coupled Code Generation for DSPs Using a Genetic Algorithm". In: *Proceedings of DATE'04*. Vol. 2. IEEE Computer Society, Feb. 2004, pp. 1270–1275.

[258] M. Löwe and H. Ehrig. "Algebraic Approach to Graph Transformation Based on Single Pushout Derivations". In: *Proceedings of WG'90*. Springer, 1991, pp. 338–353.

[259] E. S. Lowry and C. W. Medlock. "Object Code Optimization". In: *Communications of the ACM* 12.1 (1969), pp. 13–22.

[260] H. Lunell. "Code Generator Writing Systems". Doctoral thesis. Linköping, Sweden: Linköping University, 1983.

[261] M. Madhavan, P. Shankar, S. Rai, and U. Ramakrishna. "Extending Graham-Glanville Techniques for Optimal Code Generation". In: *Transactions on Programming Languages and Systems* 22.6 (2000), pp. 973–1001.

[262] M. Mahmood, F. Mavaddat, and M. Elmastry. "Experiments with an Efficient Heuristic Algorithm for Local Microcode Generation". In: *Proceedings of ICCD'90*. IEEE Computer Society, 1990, pp. 319–323.

[263] J.-B. Mairy, P. van Hentenryck, and Y. Deville. "Optimal and Efficient Filtering Algorithms for Table Constraints". In: *Constraints* 19.1 (2014), pp. 77–120.

[264] I. J. Maltz. "Implementation of a Code Generator Preprocessor". MA thesis. Berkeley, California, USA: University of California, 1978.

[265] J. Marques-Silva, I. Lynce, and S. Malik. "Conflict-Driven Clause Learning SAT Solvers". In: *Handbook of Satisfiability*. IOS Press, 2009. Chap. 4, pp. 131–153.

[266] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, and F. Charot. "Constraint Programming Approach to Reconfigurable Processor Extension Generation and Application Compilation". In: *ACM Transactions on Reconfigurable Technology and Systems* 5.2 (2012), 10:1–10:38.

[267] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, and F. Charot. "Constraint-Driven Instructions Selection and Application Scheduling in the DURASE System". In: *Proceedings of ASAP'09*. IEEE Computer Society, 2009, pp. 145–152.

[268] P. Marwedel. "Code Generation for Core Processors". In: *Proceedings of DAC'97*. IEEE Computer Society, 1997, pp. 232–237.

[269] P. Marwedel. "The MIMOLA Design System: Tools for the Design of Digital Processors". In: *Proceedings of DAC'84*. IEEE Computer Society, 1984, pp. 587–593.

[270] P. Marwedel. "Tree-Based Mapping of Algorithms to Predefined Structures". In: *Proceedings of ICCAD'93*. IEEE Computer Society, 1993, pp. 586–593.

[271] H. Massalin. "Superoptimizer: A Look at the Smallest Program". In: *Proceedings of ASPLOS'87*. IEEE Computer Society, 1987, pp. 122–126.

[272] C. McCreesh. "Solving hard Subgraph Problems in Parallel". Doctoral thesis. Glasgow, United Kingdom: University of Glasgow, 2017.

[273] C. McCreesh and P. Prosser. "A Parallel, Backjumping Subgraph Isomorphism Algorithm Using Supplemental Graphs". In: *Proceedings of CP'15*. Springer, 2015, pp. 295–312.

[274] W. M. McKeeman. "Peephole Optimization". In: *Communications of the ACM* 8.7 (July 1965), pp. 443–444.

[275] P. L. Miller. "Automatic Creation of a Code Generator from a Machine Description". MA thesis. Cambridge, Massachusetts, USA: Massachusetts Institute of Technology, 1971.

[276] S. Mouthuy, Y. Deville, and G. Dooms. "Global Constraint for the Set Covering Problem". In: *Proceedings of JFPC'07*. INRIA, Domaine de Voluceau, Rocquencourt, Yvelines, France, June 2007, pp. 183–192.

[277] C. Müller. *Code Selection from Directed Acyclic Graphs in the Context of Domain Specific Digital Signal Processors*. Tech. rep. Berlin, Germany: Humboldt-Universität, 1994.

[278] A. C. Murray. "Customising Compilers for Customisable Processors". Doctoral thesis. Edinburgh, Scotland: University of Edinburgh, 2012.

[279] A. Murray and B. Franke. "Compiling for Automatically Generated Instruction Set Extensions". In: *Proceedings of CGO'12*. ACM, 2012, pp. 13–22.

[280] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. "MiniZinc: Towards a Standard CP Modelling Language". In: *Proceedings of CP'07*. Springer, 2007, pp. 529–543.

[281] J. M. Newcomer. "Machine-Independent Generation of Optimal Local Code". Doctoral thesis. Pittsburgh, Pennsylvania, USA: Carnegie Mellon University, 1975.

[282] A. Newell and H. A. Simon. *The Simulation of Human Thought*. Tech. rep. Santa Monica, California, USA: Mathematics Devision, RAND Corporation, June 1959.

[283] J. Neyman. "Outline of a Theory of Statistical Estimation Based on the Classical Theory of Probability". In: *Philosophical Transactions of the Royal Society of London* 236.767 (1937), pp. 333–380.

[284] A. Nicolau and S. Novack. "An Efficient Global Resource Constrained Technique for Exploiting Instruction Level Parallelism". In: *Proceedings of ICPP'92*. 1992, pp. 297–301.

[285] R. Niemann and P. Marwedel. "An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming". In: *Design Automation for Embedded Systems* 2.2 (1997), pp. 165–193. ISSN: 0929-5585.

[286] S. Novack, A. Nicolau, and N. Dutt. "A Unified Code Generation Approach Using Mutation Scheduling". In: *Code Generation for Embedded Processors*. Vol. 317. Springer, 2002. Chap. 12, pp. 203–218.

[287] S. Novack and A. Nicolau. "Mutation Scheduling: A Unified Approach to Compiling for Fine-Grain Parallelism". In: *Proceedings of LCPC'94*. Springer, 1995, pp. 16–30.

[288] L. Nowak and P. Marwedel. "Verification of Hardware Descriptions by Retargetable Code Generation". In: *Proceedings of DAC'89*. ACM, 1989, pp. 441–447.

[289] A. Nymeyer and J.-P. Katoen. "Code Generation Based on Formal Bottom-Up Rewrite Systems Theory and Heuristic Search". In: *Acta Informatica* 34.4 (1997), pp. 597–635.

[290] A. Nymeyer, J.-P. Katoen, Y. Westra, and H. Alblas. "Code Generation = A* + BURS". In: *Proceedings of CC'06*. Springer, 1996, pp. 160–176.

[291] M. J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.

[292] O. Ohrimenko, P. J. Stuckey, and M. Codish. "Propagation = Lazy Clause Generation". In: *Proceedings of Principles and Practice of Constraint Programming*. Springer, 2007, pp. 544–558.

[293] A. Oplobedu, J. Marcovitch, and Y. Tourbier. "CHARME: Un Langage Industriel de Programmation par Contraintes, Illustrè par Un Application chez Renault". In: *Proceedings of the 9th International Workshop on Expert Systems and their Applications*. Vol. 1. 1989, pp. 55–70.

[294] R. J. Orgass and W. M. Waite. "A Base for a Mobile Programming System". In: *Communications of the ACM* 12.9 (1969), pp. 507–510.

[295] M. Paleczny, C. Vick, and C. Click. "The Java Hotspot™ Server Compiler". In: *Proceedings of JVM'01*. USENIX Association, 2001.

[296] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala. "DSP Design Tool Requirements for Embedded Systems: A Telecommunications Industrial Perspective". In: *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology* 9.1–2 (1995), pp. 23–47.

[297] P. P. Paulin, C. Liem, T. May, and S. Sutarwala. "CodeSyn: A Retargetable Code Synthesis System". In: *Proceedings of HLSS'94*. IEEE Computer Society, 1994, pp. 94–95.

[298] E. Pelegrı-Llopart and S. L. Graham. "Optimal Code Generation for Expression Trees: An Application of BURS Theory". In: *Proceedings of POPL'88*. ACM, 1988, pp. 294–308.

[299] T. J. Pennello. "Very Fast LR Parsing". In: *Proceedings of CC'86*. ACM, 1986, pp. 145–151.

[300] G. Perez and J.-C. Régin. "Improving GAC-4 for Table And MDD Constraints". In: *Proceedings of CP'14*. Springer, 2014, pp. 606–621.

[301] D. R. Perkins and R. L. Sites. "Machine-Independent PASCAL Code Optimization". In: *Proceedings CC'79*. ACM, 1979, pp. 201–207.

[302] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. "Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites". In: *Proceedings of ISPASS'05*. IEEE, 2005, pp. 10–20.

[303] T. A. Proebsting. "BURS Automata Generation". In: *Transactions on Programming Language Systems* 17.3 (1995), pp. 461–486.

[304] T. A. Proebsting. "Code Generation Techniques". Doctoral thesis. Madison, Wisconsin, USA: The University of Wisconsin–Madison, Nov. 1992.

[305] T. A. Proebsting. *Least-Cost Instruction Selection in DAGs is NP-Complete*. 1995. URL: web.archive.org/web/20081012050644/http://research.microsoft.com/~toddpro/papers/proof.htm (accessed 2013-04-23).

[306] T. A. Proebsting. "Simple and Efficient BURS Table Generation". In: *Proceedings of PLDI'92*. ACM, 1992, pp. 331–340.

[307] T. A. Proebsting and B. R. Whaley. "One-Pass, Optimal Tree Parsing—With or Without Trees". In: *Proceedings of CC'06*. Springer, 1996, pp. 294–306.

[308] P. W. Purdom Jr. and C. A. Brown. "Fast Many-to-One Matching Algorithms". In: *Proceedings of RTA'85*. Springer, 1985, pp. 407–416.

[309] C.-G. Quimper, A. Golynski, A. López-Ortiz, and P. van Beek. "An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint". In: *Constraints* 10.2 (2005), pp. 115–135.

[310] F. M. Quintão Pereira and J. Palsberg. "Register Allocation by Puzzle Solving". In: *Proceedings of PDLI'08*. ACM, 2008, pp. 216–226.

[311] R. Ramesh and I. V. Ramakrishnan. "Nonlinear Pattern Matching in Trees". In: *Journal of the ACM* 39.2 (1992), pp. 295–316.

[312] N. Ramsey. *C– Downloads*. 2006. URL: www.cs.tufts.edu/~nr/c-/code.html (accessed 2018-01-31).

[313] N. Ramsey and J. W. Davidson. "Machine Descriptions to Build Tools for Embedded Systems". In: *Proceedings of LCTES'98*. Springer, 1998, pp. 176–192.

[314] N. Ramsey and J. Dias. "Resourceable, Retargetable, Modular Instruction Selection Using a Machine-Independent, Type-Based Tiling of Low-Level Intermediate Code". In: *Proceedings of POPL'11*. ACM, 2011, pp. 575–586.

[315] B. R. Rau and J. A. Fisher. "Instruction-Level Parallel Processing: History, Overview, and Perspective". In: *Journal of Supercomputing* 7.1–2 (May 1993), pp. 9–50.

[316] C. R. Reeves. "Genetic Algorithms". In: *Handbook of Metaheuristics*. 2nd ed. Vol. 146. Springer, 2010. Chap. 5, pp. 109–139.

[317] J.-C. Régin. "A Filtering Algorithm for Constraints of Difference in CSPs". In: *Proceedings of AAAI'94*. AAAI Press, 1994, pp. 362–367.

[318] J.-C. Régin. “Generalized Arc Consistency for Global Cardinality Constraint”. In: *Proceedings of AAAI'96*. AAAI Press, 1996, pp. 209–215.

[319] J. F. Reiser. “Compiling Three-Address Code for C Programs”. In: *The Bell System Technical Journal* 60.2 (1981), pp. 159–166.

[320] K. Ripken. *Formale Beschreibung von Maschinen, Implementierungen und Optimierender Maschinencodeerzeugung aus Attributierten Programmgraphen*. Tech. rep. Munich, Germany: Institut für Informatik, Technical University of Munich, July 1977.

[321] F. Rossi, P. van Beek, and T. Walsh, eds. *Handbook of Constraint Programming*. Elsevier, 2006.

[322] R. L. Rudell. “Logic Synthesis for VLSI Design”. Doctoral thesis. Berkeley, California, USA: University of California, 1989.

[323] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Pearson Education, 2010.

[324] E. D. Sacerdoti. “Planning in a Hierarchy of Abstraction Spaces”. In: *Proceedings of IJCAI'73*. Morgan Kaufmann, 1973, pp. 412–422.

[325] S. Sakai, M. Togasaki, and K. Yamazaki. “A Note on Greedy Algorithms for the Maximum Weghted Independent Set Problem”. In: *Discrete Applied Mathematics* 126.2-3 (2003), pp. 313–322.

[326] V. Sarkar, M. J. Serrano, and B. B. Simons. “Register-Sensitive Selection, Duplication, and Sequencing of Instructions”. In: *Proceedings of ICS'01*. ACM, 2001, pp. 277–288.

[327] L. J. Savage. “The Theory of Statistical Decision”. In: *Journal of the American Statistical Association* 46.253 (1951), pp. 55–67.

[328] S. Schäfer and B. Scholz. “Optimal Chain Rule Placement for Instruction Selection Based on SSA Graphs”. In: *Proceedings of M-SCOPES'07*. ACM, 2007, pp. 91–100.

[329] H. Scharwaechter, J. M. Youn, R. Leupers, Y. Paek, G. Ascheid, and H. Meyr. “A Code-Generator Generator for Multi-Output Instructions”. In: *Proceedings of CODES+ISSS'07*. ACM, 2007, pp. 131–136.

[330] B. Scholz and E. Eckstein. “Register Allocation for Irregular Architectures”. In: *Proceedings of LCTES/M-SCOPES'02*. ACM, 2002, pp. 139–148.

[331] C. Schulte and M. Carlsson. “Finite Domain Constraint Programming Systems”. In: *Handbook of Constraint Programming*. Elsevier, 2006. Chap. 14, pp. 495–526.

[332] A. Schutt, T. Feydy, P. J. Stuckey, and M. G. Wallace. “Explaining the Cumulative Propagator”. In: *Constraints* 16.3 (2011), pp. 250–282.

[333] A. Schutt and P. J. Stuckey. “Explaining Producer/Consumer Constraints”. In: *Proceedings of CP'2016*. 2016, pp. 438–454.

[334] R. P. Sen. *Operations Research: Algorithms and Applications*. PHI Learning, 2010.

[335] R. Shamir and D. Tsur. "Faster Subtree Isomorphism". In: *Journal of Algorithms* 33.2 (1999), pp. 267–280.

[336] P. Shankar, A. Gantait, A. R. Yuvaraj, and M. Madhavan. "A New Algorithm for Linear Regular Tree Pattern Matching". In: *Theoretical Computer Science* 242.1-2 (2000), pp. 125–142.

[337] J. Shu, T. C. Wilson, and D. K. Banerji. "Instruction-Set Matching and GA-based Selection for Embedded-Processor Code Generation". In: *Proceedings of VLSID'96*. IEEE Computer Society, 1996, pp. 73–76.

[338] D. C. Simoneaux. "High-Level Language Compiling for User-Defineable Architectures". Doctoral thesis. Monterey, California, USA: Naval Postgraduate School, 1975.

[339] B. M. Smith. "Modelling". In: *Handbook of Constraint Programming*. Elsevier, 2006. Chap. 11, pp. 377–406.

[340] A. Snyder. "A Portable Compiler for the Language C". MA thesis. Cambridge, Massachusetts, USA, 1975.

[341] C. Solnon. "AllDifferent-based Filtering for Subgraph Isomorphism". In: *Artificial Intelligence* (2010), pp. 850–864.

[342] S. Sorlin and C. Solnon. "A Global Constraint for Graph Isomorphism Problems". In: *Proceedings of CPAIOR'04*. Springer, 2004, pp. 287–301.

[343] V. Srinivasan and T. Reps. "Synthesis of Machine Code from Semantics". In: *Proceedings of PLDI'15*. ACM, 2015, pp. 596–607.

[344] R. Stallman. *Internals of GNU CC*. Version 1.21. Free Software Foundation, Inc. Apr. 24, 1988. URL: `trinity.engr.uconn.edu/~vamsik/internals.pdf` (accessed 2013-05-29).

[345] J. Stanier and D. Watson. "Intermediate Representations in Imperative Compilers: A Survey". In: *ACM Computing Surveys* 45.3 (July 2013), 26:1–26:27.

[346] A. Sudarsanam, S. Malik, and M. Fujita. "A Retargetable Compilation Methodology for Embedded Digital Signal Processors Using a Machine-Dependent Code Optimization Library". In: *Design Automation for Embedded Systems* 4.2–3 (1999), pp. 187–206.

[347] D. Sweetman. *See MIPS Run*. 2nd ed. Morgan Kaufmann, 2006.

[348] H. Tanaka, S. Kobayashi, Y. Takeuchi, K. Sakanushi, and M. Imai. "A Code Selection Method for SIMD Processors with PACK Instructions". In: *Proceedings of M-SCOPES'03*. Springer, 2003, pp. 66–80.

[349] A. S. Tanenbaum, H. van Staveren, E. G. Keizer, and J. W. Stevenson. "A Practical Tool Kit for Making Portable Compilers". In: *Communications of the ACM* 26.9 (1983), pp. 654–660.

[350] A. K. Tirrell. “A Study of the Application of Compiler Techniques to the Generation of Micro-Code”. In: *Proceedings of SIGPLAN/SIGMICRO Interface Meeting*. ACM, 1973, pp. 67–85.

[351] S. W. K. Tjiang. *An Olive Twig*. Tech. rep. Synopsys Inc., 1993.

[352] S. W. K. Tjiang. *Twig Reference Manual*. Tech. rep. Murray Hill, New Jersey, USA: AT&T Bell Laboratories, 1986.

[353] *TMS320C55x DSP Mnemonic Instruction Set Reference Guide*. SPRU374G. Texas Instruments. Oct. 2002.

[354] J. R. Ullmann. “An Algorithm for Subgraph Isomorphism”. In: *Journal of the ACM* 23.1 (1976), pp. 31–42.

[355] P. Van Beek. Private correspondence. Nov. 2014.

[356] J. Van Praet, D. Lanneer, W. Geurts, and G. Goossens. “Processor Modeling and Code Selection for Retargetable Compilation”. In: *Transactions on Design Automation of Electronic Systems* 6.3 (2001), pp. 277–307.

[357] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man. “Instruction Set Definition and Instruction Selection for ASIPs”. In: *Proceedings of ISSS’94*. IEEE Computer Society, 1994, pp. 11–16.

[358] B.-S. Visser. “A Framework for Retargetable Code Generation Using Simulated Annealing”. In: *Proceedings of EUROMICRO’99*. IEEE Computer Society, 1999, pp. 1458–1462.

[359] E. Visser. “A Survey of Strategies in Rule-Based Program Transformation Systems”. In: *Journal of Symbolic Computation* 40.1 (2005), pp. 831–873.

[360] E. Visser. “Stratego: A Language for Program Transformation Based on Rewriting Strategies - System Description of Stratego 0.5”. In: *Proceedings of RTA’01*. Springer, 2001, pp. 357–361.

[361] S. G. Wasilew. “A Compiler Writing System with Optimization Capabilities for Complex Object Order Structures”. Doctoral thesis. Evanston, Illinois, USA: Northwestern University, 1972.

[362] S. W. Weingart. “An Efficient and Systematic Method of Compiler Code Generation”. Doctoral thesis. New Haven, Connecticut, USA: Yale University, 1973.

[363] B. Weisgerber and R. Wilhelm. “Two Tree Pattern Matchers for Code Selection”. In: *Proceedings of CCHSC’89*. Springer, 1989, pp. 215–229.

[364] A. L. Wendt. “Fast Code Generation Using Automatically-Generated Decision Trees”. In: *Proceedings of PLDI’90*. ACM, 1990, pp. 9–15.

[365] B. Wess. “Automatic Instruction Code Generation Based on Trellis Diagrams”. In: *Proceedings of ISCAS’92*. IEEE Computer Society, 1992, pp. 645–648.

[366] B. Wess. "Code Generation Based on Trellis Diagrams". In: *Code Generation for Embedded Processors*. Ed. by P. Marwedel and G. Goossens. Springer, 1995. Chap. 11, pp. 188–202.

[367] T. R. Wilcox. "Generating Machine Code for High-Level Programming Languages". Doctoral thesis. Ithaca, New York, USA: Cornell University, 1971.

[368] T. Wilson, G. Grewal, B. Halley, and D. Banerji. "An Integrated Approach to Retargetable Code Generation". In: *Proceedings of ISSS'94*. IEEE Computer Society, 1994, pp. 70–75.

[369] C. Wolinski and K. Kuchcinski. "Computation Patterns Identification for Instruction Set Extensions Implemented as Reconfigurable Hardware". In: *Proceedings of ERSA'07*. CSREA Press, 2007, pp. 175–181.

[370] L. A. Wolsey. *Integer Programming*. Wiley, 1998.

[371] S. Wu and S. Li. "Instruction Selection for ARM/Thumb Processors Based on a Multi-objective Ant Algorithm". In: *Computer Science – Theory and Applications*. Springer, 2006, pp. 641–651.

[372] W. A. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke. *The Design of an Optimizing Compiler*. Elsevier, 1975.

[373] H.-T. L. Wuu and W. Yang. "A Simple Tree Pattern-Matching Algorithm". In: *Proceedings of ATC'00*. Chiyayi, Taiwan, 2000, pp. 1–8.

[374] M. Xie, C. Pan, J. Hu, C. Xue, and Q. Zhuge. "Non-Volatile Registers Aware Instruction Selection for Embedded Systems". In: *Proceedings of RTCSA'14*. IEEE Computer Society, Aug. 2014, pp. 1–9.

[375] W. Yang. "A Fast General Parser for Automatic Code Generation". In: *Proceedings of MTPP'10*. Springer, 2010, pp. 30–39.

[376] J. S. Yates and R. A. Schwartz. "Dynamic Programming and Industrial-Strength Instruction Selection: Code Generation by Tiring, but not Exhaustive, Search". In: *SIGPLAN Notices* 23.10 (1988), pp. 131–140.

[377] J. M. Youn, J. Lee, Y. Paek, J. Lee, H. Scharwaechter, and R. Leupers. "Fast Graph-Based Instruction Selection for Multi-Output Instructions". In: *Software—Practice & Experience* 41.6 (2011), pp. 717–736.

[378] R. Young. "The Coder: A Program Module for Code Generation in High Level Language Compilers". MA thesis. Urbana-Champaign, Illinois, USA: Computer Science Department, University of Illinois, 1974.

[379] K. H. Yu and Y. H. Hu. "Artificial Intelligence in Scheduling and Instruction Selection for Digital Signal Processors". In: *Applied Artificial Intelligence* 8.3 (1994), pp. 377–392.

[380] K. H. Yu and Y. H. Hu. "Efficient Scheduling and Instruction Selection for Programmable Digital Signal Processors". In: *Transactions on Signal Processing* 42.12 (1994), pp. 3549–3552.

[381] P. Yu and T. Mitra. "Scalable Custom Instructions Identification for Instruction-Set Extensible Processors". In: *Proceedings of CASES'04*. ACM, 2004, pp. 69–78.

[382] G. Zimmermann. "The MIMOLA Design System: A Computer Aided Digital Processor Design Method". In: *Proceedings of DAC'79*. IEEE Computer Society, 1979, pp. 53–58.

[383] W. Zimmermann and T. Gaul. "On the Construction of Correct Compiler Back-Ends: An ASM-Approach". In: *Journal of Universal Computer Science* 3.5 (1997), pp. 504–567.

[384] J. Ziv and A. Lempel. "A Universal Algorithm for Sequential Data Compression". In: *Transactions on Information Theory* 23.3 (1977), pp. 337–343.

[385] V. Živojnović, J. Martınez Velarde, C. Schläger, and H. Meyr. "DSPstone: A DSP-Oriented Benchmarking Methodology". In: *Proceedings of ICSPAT'94*. DSP Assoc., 1994, pp. 715–720.

Index