Technische Universität Wien

Vienna University of Technology

Dissertation

SSA-Based Code Generation Techniques for Embedded Architectures

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

a.o.Univ.-Prof. Dipl.-Ing. Dr. Andreas Krall
E185/1
Institut für Computersprachen

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing. Dietmar Ebner
Matr.Nr.: 9926321
Schlagergasse 6/15
A-1090 Wien

Wien, Juni 2009

Kurzfassung

Eingebettete Systeme sind in zahlreichen Gebieten wie Unterhaltungselektronik, mobile Kommunikation, Bildverarbeitung oder Fahrzeugbau längst allgegenwärtig. Entwickler sind heute mit Rechenanforderungen konfrontiert, die vor nicht allzu langer Zeit noch Stand der Technik im Supercomputing-Bereich waren. Immer kürzere Entwicklungszyklen bei gleichzeitig steigender Komplexität erfordern optimierende Übersetzer, die Hochsprachen möglichst optimal auf die jeweilige Zielarchitektur übersetztzen.

Nahezu in jedem modernen Übersetzer wird die Quellsprache zumindest an ausgewählten Punkten in eine Zwischendarstellung basierend auf sogenannter SSA-Form (static single assignment) transformiert. Dabei wird das Quellprogramm in eine Form gebracht, in der jede skalare Variable genau einmal im gesamten Programm definiert wird. Dadurch wird die Analyse und Verwaltung von erreichbaren Definitionen überflüssig und zahlreiche Analysen und Optimierungen werden erheblich vereinfacht. Bisher wurde SSA-Form hauptsächlich für maschinenunabhängige Optimierungen verwendet. Eine Reihe interessanter Eigenschaften macht sie jedoch auch zu einer vorteilhaften Basis für maschinenabhängige Codegenerierungstechniken. In dieser Arbeit werden im Speziellen zwei Teilprobleme betrachtet, die erheblichen Einfluss auf die Qualität optimierender Übersetzer haben: Instruktionsauswahl und Registerzuteilung. In beiden Fällen ziehen heute vorherrschende Techniken noch keinen Nutzen aus den Vorteilen von SSA-Form.

Die Aufgabe der Instruktionsauswahl ist es, die Zwischendarstellung des Übersetzters auf Instruktionen der Zielarchitektur in effektiver Art und Weise zu übersetzen. Eine weit verbreitete Technik dafür ist so genanntes Pattern Matching. Dabei wird der Befehlssatz der Zielarchitektur mit Hilfe mehrdeutiger Grammatiken modelliert. Diese werden verwendet, um eine kostenminimale Überdeckung des Eingabeprogramms in Form von Datenflussbäumen zu berechnen. Eine solche Überdeckung entspricht einer konkreten Auswahl von Instruktionen der Zielarchitektur. Traditionelle Techniken sind auf Datenflussbäume beschränkt. In den letzten Jahren wurden jedoch Ansätze vorgeschlagen, die es erlauben, die Techniken für ganze Funktionen mit im Allgemeinen zyklischem Kontrollfluss zu erweitern. Die verwendeten Muster müssen jedoch in einer Baumgrammatik vorliegen, in der jede Produktion eine einfache baumartike Struktur hat. Zahlreiche Instruktionen verbreiteter Architekturen können nicht mit derartigen Grammatiken modelliert werden. Diese Arbeit stellt eine Generalisierung bestehender Techniken vor, die es erlaubt, allgemeinere Graphgrammatiken zu verarbeiten um damit den Befehlssatz verbreiteter Architekturen vollständig zu beschreiben.

Das zweite in dieser Arbeit behandelte Optimierungsproblem ist Spilling – ein Teilproblem der Registerzuteilung. Ziel is es, die beliebig große Menge an temporären Variablen, die während der Instruktionsauswahl generiert wurden, einer endlichen Anzahl von Maschinenregistern zuzuordnen. Im Allgemeinen muss eine Teilmenge dieser Variablen in den Hauptspeicher ausgelagert werden. Dieser Vorgang ist in der Literatur unter dem Begriff Spilling bekannt und hat wesentlichen Einfluss auf das Laufzeitverhalten des erzeugten Maschinencodes aufgrund der hohen Speicherlatenzzeiten. Üblicherweise wird das Registerzuteilungsproblem erst nach Elimination der SSA-Form behandelt. Entwicklungen der letzten Jahre auf dem Gebiet der SSA-basierten Registerzuteilung ermöglichen jedoch eine getrennte Betrachtung der einzelnen Teilprobleme, insbesondere Spilling. Diese Arbeit stellt einen neuen flexiblen Ansatz für Spilling vor. Die neue Technik bietet wesentliche Vorteile, insbesondere für Architekturen mit sehr wenigen Registern.

Anstatt problemspezifischer Algorithmen werden beide betrachteten Teilprobleme mit Hilfe von allgemeinen kombinatorischen Optimierungsproblemen modelliert und gelöst. Im Fall der Instruktionsauswahl verwenden wir Partitioned Binary Quadratic Programming (*PBQP*) – ein generalisiertes quadratisches Zuordnungsproblem. Für das Spilling-Problem entwickeln wir ein Schnittproblem unter Nebenbedingungen, das so genannte Constrained Min-Cut (CMC) Problem. Beide betrachteten Probleme sind *NP*-vollständig. Nichtsdestotrotz zeigen Experimente mit umfangreichen Testprogrammen, dass beweisbar optimale Lösungen unter akzeptablen Zeitvorgaben berechnet werden können.

Abstract

Embedded systems have become prevalent in various areas such as mobile communication, consumer electronics, image processing, and automotive. Often, the complexity of these systems has reached a point that would have been state-of-the-art in the supercomputing domain not long ago. With decreasing time-to-market cycles and increased software complexity, optimizing compilers that effectively translate applications written in high-level languages to a particular target platform have already become indispensable tools.

Almost all modern compiler infrastructures translate high-level programming languages at some point to an internal intermediate representation based on static single assignment (SSA) form. The main idea is to transform a program such that each scalar variable has exactly one definition in the program. Explicitly maintaining so-called use-def chains becomes dispensable and several traditional optimization and analysis passes simplify to a great extent. While SSA form has been traditionally used for high-level optimizations, it has some interesting properties that make it also a valuable tool for backend code generation techniques. In this thesis, we consider two subproblems that are vital for high-quality code generators: instruction selection and register allocation. In both cases, the prevalent techniques are based on traditional program representations that do not take advantage of SSA form.

Instruction selection aims to translate a compiler's intermediate code representation to a target-dependent form in a way that is efficient for a particular micro-architecture. A popular algorithmic approach is pattern matching: the target instruction set is modeled using ambiguous cost-annotated graph grammars such that a cover of the program represented in the form of data flow trees can be used to obtain a favorable and semantically equivalent machine representation. Previous work extends the scope of these techniques to the computational flow of a whole function using so-called SSA graphs. However, these techniques are confined to simple tree grammars where each production has tree structure. In this thesis, we show that numerous architectural features of popular embedded architectures cannot be modeled using tree grammars and present a generalization of previous work that is able to deal with generalized graph grammars.

The second subproblem considered in this thesis is spilling for programs in SSA form. Spilling is a subtask of register allocation, which aims to map an unlimited set of temporaries produced by the instruction selector to a finite set of machine registers. In general, a subset of these variables has to be deferred to main memory. This process is known as spilling in register allocation literature and has profound effects on the code quality of compilers for embedded systems due to large memory access latencies. Traditionally, compilers eliminate SSA form prior to register allocation. However, recent work shows that programs in SSA form have several interesting properties. In particular, these properties allow for a decoupled approach of spilling that is independent from the remaining register allocator. We propose a new approach to the spilling problem in the most-flexible spill-everywhere model and demonstrate its advantage compared to traditional heuristic techniques, especially for machines with few registers.

In both cases, the proposed techniques are based on a modeling of the problem in the form of generic combinatorial optimization problems. In the case of instruction selection, the underlying problem is partitioned binary quadratic programming (*PBQP*), which is a generalized quadratic assignment problem. Our approach to spilling is based on constrained min-cut (CMC) problems. In both cases, we proof that the underlying decision problem is *NP* complete. However, we present experimental evidence using major benchmark suites showing that algorithms delivering optimal or near-optimal results are feasible, even for very large programs.

Acknowledgments

First of all, I want to thank my advisor Andreas Krall for his outstanding support during the last years. He motivated me to pursue a Ph.D. in this field and I owe him most of my knowledge about compilers and embedded systems.

Special thanks also belong to Bernhard Scholz from the University of Sydney. Almost all of the research work in this thesis has been done together with him and his excellent ideas and his enthusiasm often helped me through the hard times.

Thanks also to Christoph Keßler from Linköping University for reviewing this thesis. My thanks also belong to Jens Knoop, the head of the computer languages group, who has always been supportive and encouraging and who contributed invaluable contacts to the international scientific community.

Most of the work in this thesis was partially funded by the Christian Doppler Foundation, which is an excellent source of funding for application-oriented research. Thanks also to OnDemand Microelectronics for their input, their funding, and their contributions during our three-year research project.

Many people contributed to the instruction selector that is part of this thesis. Thanks to Peter Wiedermann, who was an exceptional masters student, for the port to LLVM. Thanks also to Albrecht Kadlec and Florian Brandner for helping out with the ARM grammar.

Last but not least, I want to thank Anna for the love she has given me, all my friends for the time we spent together, and my parents for their support during all those years.

Contents

List of Figures

1 Introduction

Embedded systems have become a prevalent part of our everyday life and it is very unlikely that this trend is going to decline anytime soon. In fact, general-purpose computing accounts for less than 1% of the microprocessors sold every year [Tur99]. The vast majority of microprocessors are employed in embedded applications accounting for about half of the annual microprocessor revenue. Application areas for those systems range from consumer electronics to the communications and automotive market, frequently imposing real-time requirements and heavy computational workloads.

As a consequence, demands imposed on embedded systems are fundamentally different from the general-purpose computing domain. Depending on the particular application area, energy requirements, production costs, or physical dimensions are often the determining aspects. On the other hand, common requirements from the general-purpose computing domain such as binary backward compatibility and standard compliance are much less of an issue. Embedded systems are often designed to run a single application for their entire live range. These applications are tailored to the underlying hardware. Instead of binary compatibility, embedded system designers are thus more concerned with source code and tool compatibility. Thus, the compiler becomes a critical tool gluing more and more complex applications written in high-level languages to specialized target architectures.

At the same time, applications such as media processing impose computational workloads that stretch system capabilities to their limits. Traditional superscalar techniques require for a 2 – 3x speedup in performance very roughly an increase of about 80x in area and, maybe even more important, about 12x in power consumption [KTJR05]. For numerous mobile applications with critical energy and cost requirements, this is a cost too high to bear. Embedded system designers thus frequently employ application specific accelerators and simple explicit parallel RISC architectures. The latter architectural paradigm – $\underline{\text{V}}$ery $\underline{\text{L}}$arge $\underline{\text{I}}$nstruction $\underline{\text{W}}$ord (*VLIW*) – originated in the supercomputing domain and gradually found its way into today's embedded system architectures, e.g., ST2xx, TI C6xxx, NXP TriMedia. For those systems, the burden to effectively discover and exploit $\underline{\text{i}}$nstruction-$\underline{\text{l}}$evel $\underline{\text{p}}$arallelism (*ILP*) is solely left to the compiler, thus emphasizing the vital role of optimizing compilers for embedded systems.

While both the superscalar and *VLIW* paradigm have their advocates, it is often overlooked that the complex problems in analyzing and exploiting the available parallelism is common to both architectural styles. Thus, compiler complexity depends rather on the amount of *ILP* that has to be obtained than on the particular architectural style [FFY05]. The main difference is that *VLIW* architectures usually make more fine grained parallelism practical.

Today, compilers mainly rely on highly tuned and computationally efficient heuris-

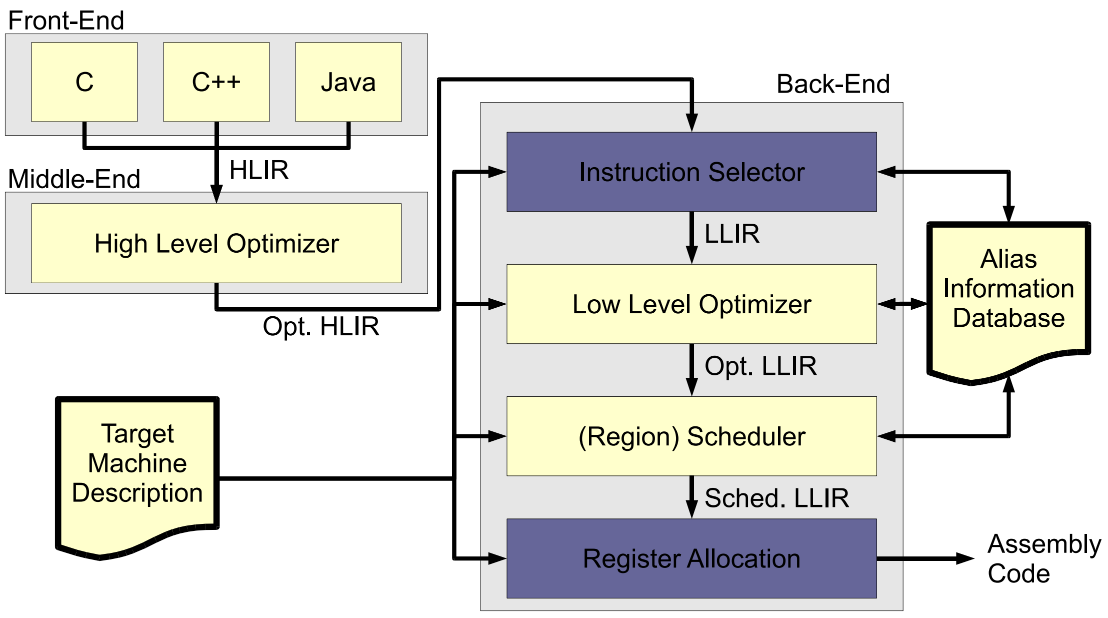


Figure 1.1: Phase layout of optimizing compilers.

tics that produce reasonably good code quality in a short amount of time. However, in the context of embedded systems where application-specific extensions and architectural variants are employed to tune those systems for a single or a small number of applications, compilers have to fulfill additional requirements that are hard to meet using heuristic approaches. First, hardware architectures and application-specific extensions usually evolve much faster than the corresponding compilers. Therefore, easily retargetable backends that are capable to cope with architectural peculiarities and asymmetries are required. Second, heuristic techniques often fail to effectively exploit architectural features requiring significant amount of engineering to tune applications to a particular hardware platform.

Optimizing compilers are huge and complex software projects, e.g., recent versions of *gcc* contain about 1.2 million lines of C-code, disregarding the large *C++* and Java runtime libraries. Modern compilers are composed of several components, each of them operating on different intermediate representations of the original source code; see Figure 1.1. The *front-end* is responsible to transform the input program into a high-level intermediate representation that is largely independent of the particular source language. So-called *high-level optimizers* perform the traditional set of scalar and loop optimizations, do function inlining and cache re-organizations, and remove dead code and redundant computations. Some compilers go through various forms of high-level intermediate representations with decreasing abstraction level. Finally, the *back-end* is responsible to convert the source and target independent intermediate representation into machine-dependent instructions, perform target-dependent optimizations and code re-organizations, and to do static instruction scheduling and register allocation.

As depicted in Figure 1.1, most compiler backends are structured into several phases that concentrate on a particular subproblem, e.g., instruction selection, register allocation, or scheduling. This approach, which we follow in this thesis, is beneficial from a software engineering point of view as it allows for a separation of concern. Also, with most of these subproblems being *NP* complete in general, performance requirements are an even stronger argument for this separation. However, it is important to note that these advantages come for significant costs. Interdependencies among the particular phases are usually ignored, leading to suboptimal code. This dilemma – known as phase ordering problem – is difficult to resolve since several phases have conflicting goals. One prominent example are interdependencies among scheduling and register allocation. Allocating machine registers to program variables introduces additional "false" dependencies that limit the freedom for the instruction scheduler. On the other hand, scheduling may overlap otherwise independent live ranges and thus increase the register pressure, which may lead to additional spill code. Likewise, there are interdependencies among instruction selection and scheduling as the costs of a particular instruction in general depend on interference and resource constraints imposed by a particular schedule.

One approach proposed in literature is to account for phase ordering issues heuristically, e.g., Goodman and Hsu [GH88] propose different scheduling techniques that favor pipeline delays and register demand respectively and that are applied dynamically according to the estimated register pressure. Similar but more sophisticated techniques have been proposed more recently by Norris et al. [NP98]. Another approach is to combine several phases to a single optimization problem. Bednarski and Keßler [BK04, KB06] propose dynamic programming for combined instruction selection, scheduling, and register allocation. Likewise, both Wilson et al. [WGB94] and Chang et al. [CCK97] propose techniques that simultaneously perform these subproblems using integer linear programming. However, for large programs, the complexity of the formulations results in prohibitively high computation times. Despite significant amount of research, phase ordering issues are thus still considered to be an open research question.

Most modern compiler frameworks transform intermediate code, at least at some stage in the compilation process, into so called static single assignment (SSA) form [CFR$^+$91a]. The basic idea which was proposed by researchers from IBM in the late eighties is to transform a program such that each scalar variable has exactly one point of definition in the program. Thus, explicitly maintaining use-def chains is dispensable and several traditional optimization and analysis passes simplify to a great extent. While SSA form has been traditionally used for high-level optimizations, it has some interesting properties that make it also a valuable tool for backend code generation techniques.

In this thesis, two subproblems of code generators are considered: instruction selection and spilling. The corresponding stages in the phase diagram are highlighted in purple in Figure 1.1. While those are very different problems in general, there are two important characteristics in the approach presented in this thesis that are common to both:

1. Both techniques are based on properties for programs in SSA form and can be

used to maintain SSA form until late in the compilation process.

2. While traditional techniques are based on problem specific heuristics, we use in both cases a reduction to general combinatorial optimization problems. This allows us to exploit well-known results from graph theory and to apply generic solver libraries for different subproblems.

Contributions

The contributions of this work are two-fold. First, we propose new SSA-based algorithms for two important subproblems of optimizing compilers: instruction selection and spilling. For the instruction selection problem, we propose a generalization of previous work that allows to use general graph grammars instead of tree grammars for the modeling of the target architecture. For the spilling problem, we introduce a new mathematical formulation for the most-general spill-everywhere model and develop effective algorithms that deliver optimal or near-optimal solutions. For both techniques, we present experimental results that show significant improvements compared to previous techniques. Experiments with large benchmark sets show that our techniques scale even to very large programs. We consider two example architectures representing different architectural paradigms from the embedded systems domain, i.e., an embedded VLIW architecture designed for video and audio decoding and a traditional ARM core as representative for simple RISC architectures.

The second major category of contributions in this work are theoretical results for the underlying combinatorial optimizations problem. We implement and extend previously proposed algorithms for the so-called partitioned binary quadratic programming problem. The work for spilling is based on a new variant of the well-known min-cut problem with additional side constraints. We present an exact *ILP* formulation and a Lagrange relaxation that is used to guide simple greedy heuristics towards the optimal solution. Both problems are *NP* complete. Nevertheless, optimal solutions can be found within reasonable time limits, even for very large programs.

Guide to this Thesis

Chapter 1 gives a short introduction to embedded systems and compilers, in particular *LLVM* [LA04] which is used throughout this thesis for the evaluation of the proposed techniques. Preliminaries for the remaining chapters and summaries of results that are important for the rest of the thesis are given in Section 1.4. Chapter 2 introduces partitioned binary quadratic programming (*PBQP*) – a generalized combinatorial optimization problem that is used later on as an algorithmic vehicle for the instruction selection problem. Several algorithms are presented and evaluated for problem instances from various applications such as address mode selection, register allocation, and instruction selection. We present a new approach to generalized instruction selection that is able to cope with complex machine instructions in Chapter 3. The problem is solved by reducing the problem of matching SSA graphs with generalized graph grammars to

PBQP. We hereby achieve both improved code quality and improved retargetability to new architectures or variants. The latter is achieved by supporting more expressive graph grammars that are flexible enough to describe popular embedded instruction set architectures to their full extend. Chapter 4 summarizes recent advances in SSA based register allocation and presents a new formulation of the spilling problems to the so-called constrained min-cut problem. The latter is a special variant of the well-known min-cut problem with additional knapsack constraints. We implemented and evaluated several algorithms and present results for two examples of real-world architectures using major benchmark suites. Conclusions and an outlook to future work is given in Chapter 5.

1.1 The Embedded Computing Landscape

Embedded Computing is a very wide field stretching from tiny 8-bit micro-controllers to full-featured parallel RISC architectures for mobile multimedia applications. Lifetimes span from less than a year for average mobile phones to often more than thirty years for infrastructure such as telephone switches. Consequently, there is no sharp definition that distinguishes embedded systems from general-purpose computing. As a result, most definitions found in literature characterize embedded systems by what they are *not* rather than what they are.

One of these features that distinguish embedded computing is the lack of versatility. Usually, embedded devices serve few or even only a single purpose. Software is often closely tied to the hardware and pre-installed by the manufacturer with no or only limited ways of changing it, e.g., firmware upgrades. However, this distinction becomes more and more fuzzy with PDAs and mobile phones capable to execute third-party applications and Java bytecode.

Another characteristic property that is typical for embedded systems is the lack of binary backward compatibility. In the general purpose computing, strict binary compatibility has been the only viable path to commercial success. This is hardly an issue in the embedded domain. Developers are willing to adopt and recompile the application for each new product generation. Many systems are built using very simple operating systems that are closely tied to the underlying hardware or even no operating system at all. However, with rising complexity of the applications used within embedded systems, rewriting the whole software stack for each hardware generation becomes quickly infeasible. Thus, compilers and tools form the compatibility layer that matters most for embedded system development.

A very informal but probably the most concise definition for embedded systems is in terms of their intended *use*. Usually, embedded devices are commodities that serve a particular purpose that is very different from “computing”. This is also the main reason for the very different requirements imposed on embedded systems. While higher performance is almost always an appealing goal in the general-purpose computing domain, this is not necessarily the case for embedded applications. The main goal is to be as fast as necessary using the minimum amount of resources possible. Apart from production costs, mobile applications are also often constrained by two more aspects that are hardly an issue for general purpose computing, i.e., energy demand and physical dimensions. For several applications such as mobile communications or mobile multimedia processing, these are usually the determining "performance" criteria rather than raw computational power.

1.1.1 Types of Embedded Processors

Processing cores used in embedded systems are usually divided into four main categories that range from very small serial controllers to full-blown micro-processors. In increasing complexity, these categories can be summarized as follows:

1. **Microcontrollers** These cores are commodity components that can be found in almost all electronic devices. Usually, they include small memories, I/O buses, and peripherals and are based on very simple 8- or 16-bit microprocessors.

2. **Digital Signal processors (DSPs)** These are processing units built for very special compute-intensive loop-oriented kernels. They are designed with a small family of signal processing algorithms in mind and are optimized for very efficient arithmetic operations. Usually, full floating arithmetic is replaced with simple fixed-point representation. Special instructions, e.g., multiply-accumulate, are very common to allow for efficient implementation of common vector and matrix operations such as dot products.

3. **Embedded General-Purpose Microprocessors** These are often general purpose processor designs or scaled-down versions of former architectures from the general purpose domain such as ARM, MIPS, PowerPC, or Motorola 68K. Characteristics are 32-bit datapaths, complete and standardized instruction set architectures, and relatively large external memories. Most of these processors are sold in the form of *soft-cores*, i.e., licensed building blocks that are combined by the application designer to larger systems-on-a-chip (SoCs).

4. **Computational Microprocessors** These are devices that can be found in high-end portable devices and are usually close descendants of full-featured workstation designs such as PowerPC, SPARC, or Alpha. With companies such as VIA and Intel entering the embedded market with specialized micro-architectures for x86, their adapted designs also fall into this category, e.g., VIA Nano, Intel Atom. They adopt full 32 or 64-bit datapaths and are usually combined with peripherals and memory management units in order to support full-featured operating systems. Application areas are commodity devices such as PDAs or netbooks.

Traditionally, DSPs have been the fastest growing sector in the embedded semiconductor market. Modern designs found in applications such as mobile communication, filtering, audio compression, or signal synthesis, combine one or more traditional DSPs with general-purpose microprocessors. DSPs hereby often execute a single program on a stream of data. The irregular datapaths and designs make it hard to compile high-level programs efficiently. Thus, even today, it is not uncommon to develop code in assembly language, which is a very cost and labor intensive endeavor. Thus, several analysts predict that classic DSPs will migrate into designs that are much closer to general purpose microprocessors and *VLIW* designs.

The techniques proposed in this thesis mainly apply to the last two groups of processing cores: computational microprocessors and embedded general-purpose microprocessors. However, the borders to classical DSPs become more and more blurred and some techniques might also be relevant for this field. This is true especially for the part on instruction selection where the additional flexibility compared to traditional techniques might help to cope with the irregularities found in typical DSP designs.

1.1.2 Application Areas

In this section, we briefly summarize a rough taxonomical characterization for 16-bit and wider architectures based on their application area as given by Fisher et al. [FFY05]. The authors identify three major embedded computing markets with distinct requirements:

1. **Image Processing and Consumer Electronics** This market is among the fastest growing segments of embedded computing and includes consumer electronics such as printers, digital cameras, and (mobile) video and audio devices. These devices operate on digital media, which is today comprised of huge amounts of data that has to be transferred and processed within embedded systems. Most systems thus employ embedded general purpose microprocessors with a large amount of RAM and fast memory connectors. The high computational requirements of modern digital video codecs such as H264 or VC-1 requires significant computational power. Several companies such as ST Microelectronics or Texas Instruments thus offer specialized designs based on *VLIW* principles with special accelerator units and large memory bandwidth. Consumer electronics are high-volume commodities that have to cope with huge cost pressure and often sell for less than the cheapest computational microprocessor available. This fact as well as size and power constraints are important factors for the choice of microprocessors in this field.

2. **Communications** This is a very diverse market ranging from wired and cellular phone network infrastructure to mobile cellular phone handsets and networking infrastructures such as routers. All this application have very different requirements and characteristics.

   Wired and mobile phone infrastructure has to support a huge number of relatively low bandwidth clients, summing up to vast data rates. Additionally, wired networks are based on analog signal transmission, requiring the analog-to-digital and digital-to-analog conversion in the data center. On the other hand, cellular networks are based on various digital encoding and protocol techniques such as GSM or UMTS.

Even larger data rates can be found in typical network backbones. A typical router has about 32 interfaces with a bandwidth of about 10 Gbit/s each, translating to several hundred million routing decisions per interface on smaller package sizes. Routing decisions are based on so-called routing tables with about 50.000 to 100.000 entries. Almost all components exceed the capabilities of general purpose microprocessors. Thus, systems are usually based on customized processors, so-called application specific integrated circuits (ASICs), or field-programmable gate arrays (FPGAs).

A very different set of characteristics can be found in cellular phone handsets. These systems are usually based on a combination of general purpose microprocessors and DSPs, e.g., Texas Instruments OMAP series. Processors traditionally run on relatively low clock rates of a few hundred MHz and include a relatively small number of RAM. This trend has changed with recent developments in the smartphone market where more powerful systems are adopted that can be used to drive built-in digital cameras and multimedia decoders. A limiting factor in this market is energy consumption. Typical standby power is a few milliwatts while transmission power rises drastically to several hundred milliwatts.

3. **Automotive** The use of embedded microprocessors in the automotive market has grown dramatically in the last decade. Today, it is not uncommon to find between 50 to 100 processors in a single vehicle. These chips are used for a wide range of applications such as controlling, safety features, instrumentation, and entertainment. Consequently, different architectural styles are used. Fault tolerant low level real time systems are often based on DSPs or simple RISC microprocessors. High-end microprocessors are usually based on traditional RISC architectures such as PowerPC and are designed for relatively low frequencies.

   A distinguishing characteristic of the automotive market is the high degree of integration among the different subsystems, e.g., self-diagnostic and reporting, sensor networks, or controlling. Individual components, so-called “smart-nodes”, hook up to standardized communication systems such as CAN or FlexRay.

1.2 Example Architectures

The techniques presented in this thesis have been evaluated using two example architectures for embedded general purpose microprocessors. One of which is ARM, a simple 32-bit RISC core that is by far the most widely used architecture in the mobile electronics market. The second architecture, OnDemand CHILI, is a representative for explicitly parallel VLIW architectures that are frequently found within numerically intensive applications such as video en-/decoding.

Embedded processors are hardly sold on their own. Instead, multiple processing cores along with external memories and peripherals are combined to a single system-on-a-chip (SoC). A reference design for such a system for mobile multimedia applications is shown

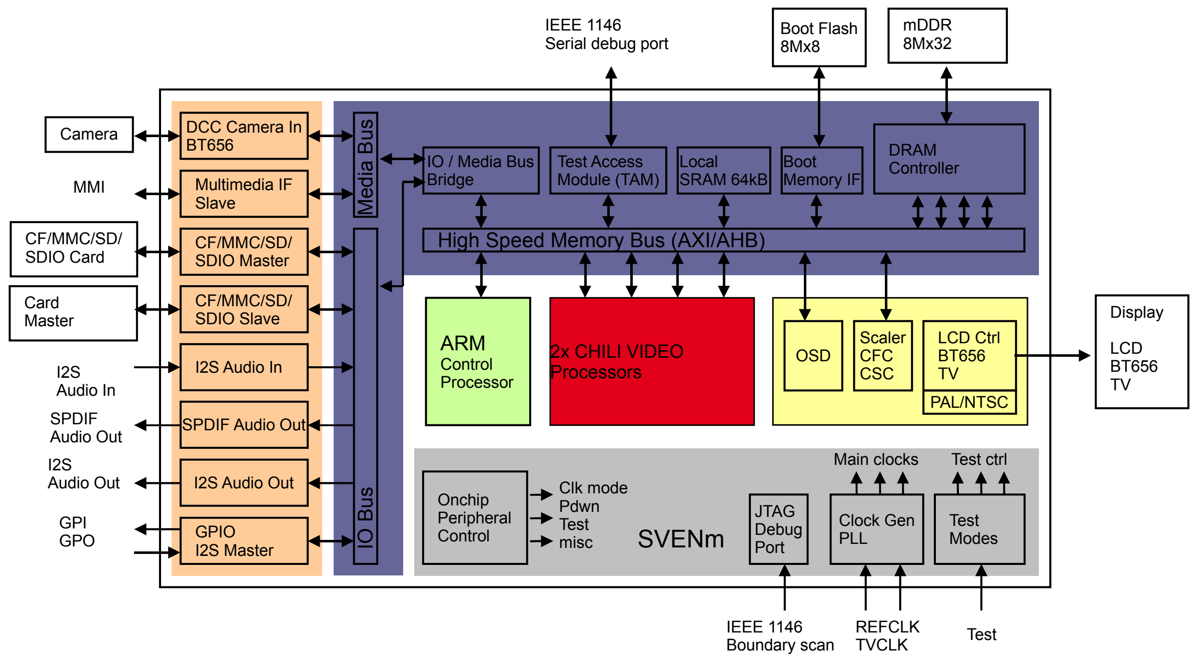


Figure 1.2: Schematic layout of a reference design for mobile multimedia decoding (SVENM).

in Figure 1.2. The system combines an ARM control processor capable to drive full-featured operating systems such as Linux with two CHILI *VLIW* cores that are used to run computationally intensive video decoders such as H264. There is a small amount of on-chip core memory on each of the VLIWs and all the processing cores are connected to a global mobile DRAM controller. Additionally, there is a large number of peripheral controllers and interfaces such as audio and video connectors, flash memory controller, and debug interfaces. A picture of the reference design shown in Figure 1.2 in real silicon is given in Figure 1.3[1].

1.2.1 ARM

ARM processors today account for approximately 90% of all 32-bit microprocessors found in embedded systems. The architecture has been developed by ARM Ltd. – a spin-off of Acorn Computers founded in the late nineties. It is important to note that ARM Ltd. does not fabricate real silicon itself but licenses ARM cores to semiconductor partners that include them into customized SoCs.

Basic Architecture ARM is a simple 32-bit Von Neumann architecture featuring 16 general purpose registers (two of which are reserved for the stack pointer and link register and `r15` denotes the program counter). A so-called program status register (`CPSR`)

[1]Image included with permission of OnDemand Microelectronics

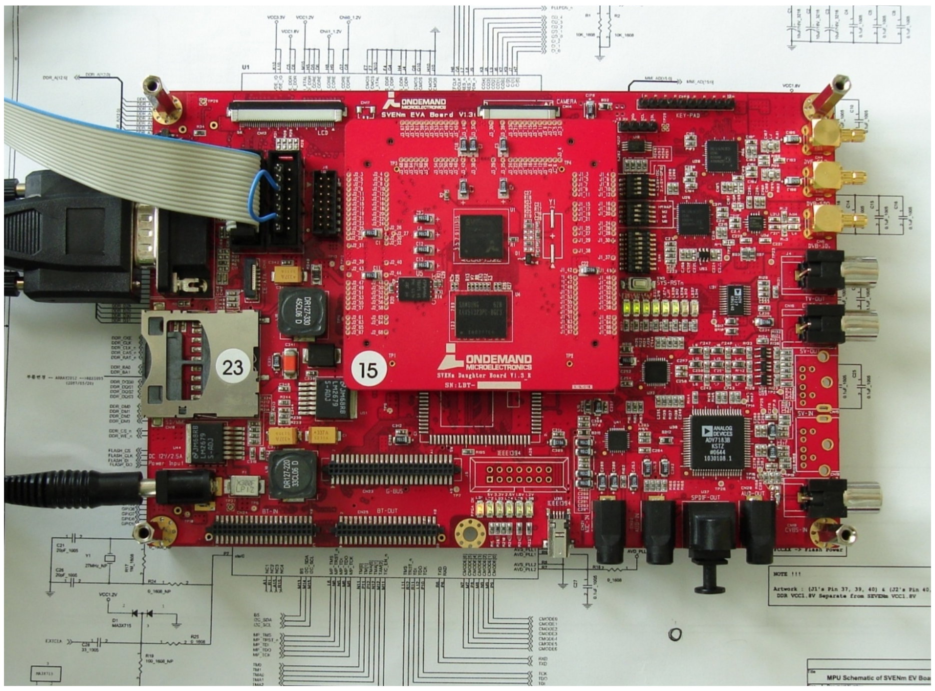


Figure 1.3: Developer board with multiple VLIW cores, peripherals, off-chip memory, and a general purpose control processor (ARM).

reflects the current processor state. There are four condition code flags (negative/zero result, carry, overflow) that allow for conditional execution of instructions. This can be used to improve code density and to reduce the number of branch instructions using if-conversion, i.e., the conversion of control into data dependencies by the compiler. Dedicated flags are used to enable/disable interrupts and switch among two operations modes, i.e., ARM mode and Thumb mode.

ARM Mode In ARM mode, all instructions are 32-bit wide and have to be word aligned. All instructions have full access to the available general purpose registers. Instructions can be executed conditionally by post-fixing them with the corresponding condition code field. A special field (signify bit) in the instruction word defines if data processing instructions affect the processor status word.

ARM Thumb Mode Thumb mode has been introduced to improve code density for compiler generated code (marketing data claims code size reductions of about 65%). ARM Thumb is a 16-bit instruction set implementing only a subset of the functionality available in ARM mode. A special branch instruction (`bx`) can be used to switch among the two operation modes. For most instructions, one source and the destination register have to be identical. With few exceptions, instructions are only allowed to address the lower half of the register file.

pre-increment	LDR|STR {B} <Rd>,	[<Rn>, #± <imm12>]! [<Rn>, ± <Rm>]! [<Rn>, ± <Rm> <shift> #<imm>]!
	LDR|STR {H|SH|SB} <Rd>,	[<Rn>, #± <imm8>]! [<Rn>, ± <Rm>]!
post-increment	LDR|STR {B} <Rd>,	[<Rn>], #± <imm12> [<Rn>], ± <Rm> [<Rn>], ± <Rm> <shift> #<imm>
	LDR|STR {H|SH|SB} <Rd>,	[<Rn>], #± <imm8> [<Rn>], ± <Rm>

Table 1.1: Available ARM pre-/postincrement addressing modes with implicit address calculation.

A major adjustment has been proposed in 2003 with the introduction of Thumb-2. The instruction set extends the Thumb mode with some additional 32-bit instruction and can be seen as a compromise between ARM and ARM Thumb mode with the goal to achieve both high code density and decent performance.

The Barrel Shifter The 32-bit barrel shifter is only available in ARM mode and can be used without performance penalty in combination with most arithmetic instructions and address calculations. The shift amount can be specified either using a 5-bit immediate value or using the least significant byte of an additional source register. The operation can be a logic or arithmetic shift operation or a rotate right operation (with and without carry bit).

Addressing Modes Powerful indexed addressing modes are available for both load and store operations. There are variants operating on signed/unsigned bytes, half-words, or words. Addresses are specified using a base register plus an optional offset, which can either be added to or subtracted from the base registers. The offset can be a register optionally shifted by an immediate value or an unsigned 12-bit immediate. For half-word and signed half-word/byte operations, the offset is restricted to 8-bit immediates or an unshifted register. Additionally, ARM supports a large range of pre- and post-increment addressing modes with implicit address calculation; see Table 1.1.

Core Extensions The ARM architecture has gradually evolved from the early ARM1 series to ARM11 and, most recently, the ARM Cortex family. Discussing the differences among all the available variants is beyond the scope of this work, but there is a number of widely-used core extensions that are important in practice.

Many embedded applications do not require floating point at all and are based on (saturated) integer arithmetic. For those that do, several instruction set extensions are available that provide full single and double precision IEEE 754 floating point arithmetic.

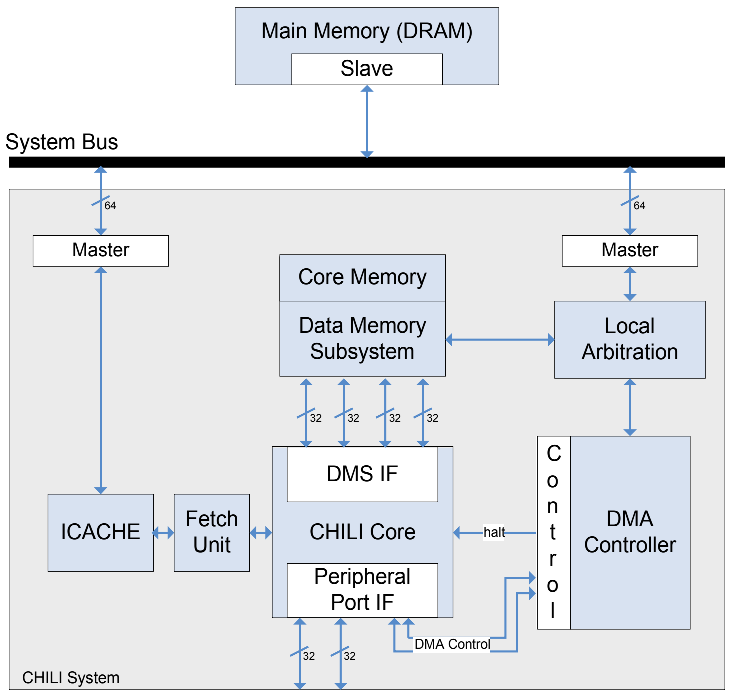


Figure 1.4: Overview of the CHILI architecture.

The most popular of which is *VFP*, which even supports short vector instructions.

Another core extensions is an advanced 128-bit SIMD (single instruction multiple data) unit marketed under the term *NEON*. It features a separated register file and additional functional units that support integer and single-precision floating point operations from 8 to 64-bits, processing up to 16 operations in the same cycle. The primary market for NEON are mobile media and signal processing applications.

Further extensions such as *TrustZone* are found in a few variants only and target niche markets with security-sensitive applications.

1.2.2 CHILI

The CHILI processor is a good example for wide-issue *VLIW* cores that are popular for media processing applications with high computational requirements. The architecture has been developed by OnDemand Microelectronics, an Austrian start-up company, and features 4 parallel execution units. Each slot has access to an unclustered general purpose register file offering 64 32-bit registers. The instruction set is a general load/store RISC architecture extended with a small number of general purpose instructions for multimedia processing and limited support for SIMD processing.

A block diagram for the CHILI architecture is given in Figure 1.4. As common for *VLIW* architectures, encoding and instruction memory bandwidth is a critical issue. Thus, each processing core is connected to a relatively large instruction cache that is managed by a dedicated fetch unit, which pre-fetches and aligns instructions from main memory. Each core has a small on-chip memory (SRAM) and is connected to a DMA controller using a peripheral port interface.

Memory Subsystem The CHILI core is split into the processing core and the data memory subsystem (DMS). The latter is responsible for DMA transfers among core memory and external DRAM and provides non-blocking memory accesses for the processing core, i.e., an issued load instruction does not stall the processing core unless the value is actually required to proceed with the calculations. This allows to pre-fetch required values from slow external memories while the core is busy with other operations and helps to hide the long access latencies. Each load and store operation supports direct addressing modes using a register or a fully qualified 32-bit immediate value as value as register indirect addressing modes. In the latter case, the offset can be specified using another source register or a 32-bit immediate value. The DMS allows transparent access to off-chip external memories. However, in general it is more efficient to transfer larger blocks to the core memory using the DMA engine, which is controlled using a peripheral port interface.

Conditional Execution Most existing general purpose architectures have either only partial support for predicated execution (mostly restricted to conditional moves, e.g., DEC Alpha, Sun Sparc v9) or nullify the result based on the value of an additional boolean source predicate (Itanium, ARM), which has to be evaluated beforehand. The CHILI differs from these architectures in that conditions are evaluated alongside to the instruction to be predicated within the same bundle. Therefore, the full range of binary comparisons is provided in addition to special test instructions that evaluate to true if a particular bit is set or unset respectively. However, these computations require an additional slot in the instruction word. In particular, even slots can be used to evaluate the predicate for the instruction in the directly succeeding slot. The only exception are currently load and store instructions, which cannot be executed conditionally. If multiple instructions are defining a register within the same bundle, the value produced in the slot with the highest index is kept.

SIMD Extensions CHILI's SIMD extensions allow to perform parallel 8-bit and 16-bit arithmetic on each of the processor's 32-bit wide execution units. This allows to execute up to 16 8-bit operations and up to 8 16-bit operations in a single cycle. Special permute instructions can be used to efficiently pack and unpack values into registers. Permutations can either be specified using immediate values or can be programmed using special permute registers that can be loaded using dedicated instructions. These extensions are especially valuable for video and audio decoding algorithms that often operate on vectors of 8-bit values.

Instruction Set Extensions Apart from the SIMD extensions, there are some additional instruction set extensions that are uncommon for RISC machines and which are mainly heritages from the DSP world. These instruction include multiply-accumulate

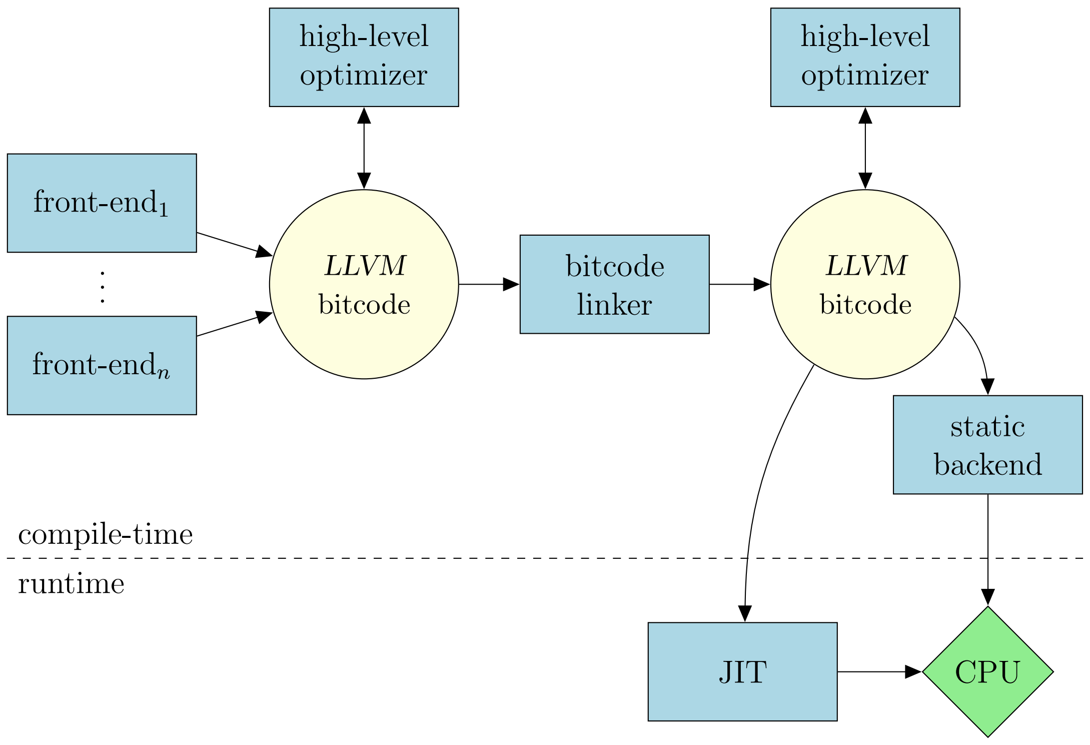


Figure 1.5: Overview of the *LLVM* compiler infrastructure.

operations, clip instructions that can be used to enforce upper and lower bounds, or sum of absolute differences, which is very common within signal processing algorithms. Furthermore, there are specialized instructions for alignment and rounding and for counting of leading zeros/ones. All of which have applications in the architecture's target market.

1.3 The LLVM Compiler Infrastructure

This section gives a short introduction to the LLVM compiler infrastructure – a modern, open source compiler framework that originated at the University of Illinois and that served as the framework for the experimental evaluation of the techniques proposed in this thesis.

LLVM is an acronym for "low-level virtual machine" and is both a well-defined virtual instruction set as well as a set of components that can be used to build various tools such as static code generators, just-in-time translators (JITs), static code analyzers, or dynamic simulators.

1.3.1 Overview

LLVM is based on a modular design of loosely coupled libraries that operate on a well-defined intermediate representation. The main purpose of LLVM is the development of static and dynamic compilers. Figure 1.5 show the basic structure of code generators based on LLVM. Various front-ends translate high-level languages such as C, C++, Objective C, or Fortran into so-called LLVM bitcode. In addition, there are active projects implementing LLVM front-ends for virtual machine languages such as Java Bytecode or Microsoft's CLI. At the time of writing, the main front-end for LLVM is based on the *gcc* compiler infrastructure and translates *gcc*'s intermediate representation (GIMPLE) into LLVM bitcode. This is a relatively simple task and abstracts away much of the complexity of dealing with the various source languages. Most features of modern high-level programming languages are hereby lowered to very simple constructs.

LLVM bitcode serves as both input and output format for various readily available analysis and optimization passes. This includes most of the traditional scalar optimizations, loop transformations, code motion and redundancy elimination, and sophisticated analysis passes such as alias analysis or loop dependence tests. Theses optimizations can be applied in the traditional way separated to each translation unit. Alternatively, the design of LLVM allows for transparent link-time optimizations by linking LLVM bitcode modules together before optimizations are applied.

The backend infrastructure permits code generation ahead-of-time as well as during runtime. The latter allows to ship LLVM bitcode files instead of machine dependent binaries. These can be translated for a particular machine on-the-fly (JIT) or on the first execution (install-time optimization). Furthermore, there is limited support for profile-guided optimizations, i.e., training data is used to gather additional information for a particular application that can be used to improve the code quality of a subsequent translation pass.

1.3.2 LLVM Virtual Instruction Set

The intermediate representation used by LLVM modules is a fully-typed RISC-like virtual instruction set. In contrast to many other compiler infrastructures, there is no hierarchy of intermediate representations with varying abstraction level. Instead, all analyzers and optimization passes operate on the same representation, which is lightweight and low-level while being rich enough for most interesting tasks. The language is universal in that it is independent of the particular source language and comes with its own basic type system.

It is important to note that LLVM bitcode is fundamentally different from virtual machines such as Java bytecode or CLI. First, LLVM bitcode is *not* target machine independent. Peculiarities defined by the particular application binary interface (ABI) or by the target machine such as alignment, mapping of high-level types, or calling conventions, are explicitly represented by the front-end. Thus, object code represented by LLVM bitcode is not portable from one architecture to another in general. Another characteristic that distinguishes LLVM bitcode is the lack of features such as garbage collection, that one would expect from virtual machines. However, it is possible to implement those features on-top of LLVM.

LLVM bitcode can be represented in one of different forms, all of which are semantically equivalent. First, there is an in-memory compiler intermediate representation

```
float
dot_product(float *a,
            float *b,
            unsigned n) {
  float sum = .0f;
  unsigned i;
  for(i=0; i < n; ++i)
    sum += a[i] * b[i];
  return sum;
}
```

```
define float @dot_product(float* %a,
                          float* %b,
                          i32 %n) nounwind {
entry:
        br label %bb1

bb:             ; preds = %bb1
        %0 = getelementptr float* %a, i32 %i.0
        %1 = load float* %0, align 4
        %2 = getelementptr float* %b, i32 %i.0
        %3 = load float* %2, align 4
        %4 = mul float %1, %3
        %5 = add float %4, %sum.0
        %6 = add i32 %i.0, 1
        br label %bb1

bb1:            ; preds = %bb, %entry
        %i.0 = phi i32 [ 0, %entry ], [ %6, %bb ]
        %sum.0 = phi float [ 0.0, %entry ], [ %5, %bb ]
        %7 = icmp ult i32 %i.0, %n
        br i1 %7, label %bb, label %bb2

bb2:            ; preds = %bb1
        br label %return

return:         ; preds = %bb2
        ret float %sum.0
}
```

Figure 1.6: Example LLVM bitcode corresponding to a simple c-function computing a vector dot product.

that is used by LLVM analyzers and optimization passes. It is represented in a C++ class hierarchy and there are various methods for in-memory construction and debugging. Next, there is an efficient on-disk representation that is suitable for fast loading for just-in-time compilers. The third form is a human readable text representation that is very helpful for debugging and can even be used to write LLVM code by hand using an assembly-like language. There are tools to convert among all these forms without loss of information.

A verification pass asserts that LLVM bitcode is well-formed, i.e., it adheres LLVM syntax and its type system and each scalar value is in strict SSA form. There are no restrictions concerning reads from and writes to memory locations. However, a dedicated pass (`mem2reg`) identifies memory locations whose address is never taken and promotes them to virtual registers, inserting phi functions where appropriate. This effectively removes burden of SSA construction from the front-end by mapping local variables to stack references that are processed by `mem2reg` later on.

A simple example showing LLVM bitcode for a vector dot product is shown in Figure 1.6. Note, that all variables are fully typed. A special function (`getelementptr`) is used to obtain the address of an aggregate data structure in a type-safe way. Memory access instructions are represented explicitly by loads and stores. The example is in strict SSA form, i.e., each variable is defined exactly once and each use is dominated by its definition. Phi-functions are used to disambiguate different values at join points, e.g., at the beginning of basic block `bb1`. Note that the actual semantics of each operation depend on the type of its arguments, e.g., floating-point versus integer `add`.

1.3.3 Backend Infrastructure

A "backend" translates LLVM bitcode into a usually equivalent program representation. In most cases, this is the assembly language for a particular hardware platform. However, there are also backends that translate LLVM bitcode to C or virtual machines. The C-backend is interesting as it allows to use the LLVM infrastructure on platforms that are not explicitly supported by applying a generic vendor compiler to LLVM's output.

At the time of writing, LLVM supports about a dozen different target architectures and variants such as X86(_64), PowerPC (32/64), Alpha, Sparc, Itanium, ARM, or Cell. A *VLIW* backend and the necessary additions for the OnDemand CHILI core has been developed by our group. All backends are based on a target independent code generator infrastructure, which consists of five main components:

- **Abstract Target Machine Descriptions** Target machine characteristics that are important for code generation such as descriptions of the particular instruction set architecture, functional units, or register files, are defined using a dedicated description language called `TableGen`. Descriptions are independent from a particular component and can be effectively organized in a class hierarchy to obtain concise descriptions by factoring out common information, e.g., most arithmetic operations such as `add` or `sub` differ only in their opcode while the set of operands and register constraints remains the same. `TableGen` descriptions are independent of a particular algorithm. However, there are different "backends" that are used to generate target specific data structures and algorithms at compiler compile time.

- **Machine Code Infrastructure** LLVM provides a set of classes to represent target specific machine instructions in a target independent way. Some components such as the set of valid opcodes or registers are generated from backend descriptions at compiler compile time. Code is organized in the form of classical control graphs while straight line segments without control flow are, as usual, represented by basic blocks. Apart from few exceptions, the semantics of a particular operation are not known any more. It is the job of the instruction selector to convert LLVM bitcode into a semantically equivalent machine representation for a particular target.

- **Target-Independent Components** Almost all major backend components are implemented in the form of target-independent algorithms that make use of the backend descriptions during runtime or that are partially generated for a particular target at compiler compile time. This includes instruction selection, register allocation, and scheduling. For several tasks, multiple algorithms are implemented that can be selected using command line options.

- **Abstract Interfaces for Target-Specific Hooks** The generic LLVM backend infrastructure provides a small number of abstract target interfaces that are implemented for each of the available target architectures. These interfaces provide functionality that cannot be derived from the abstract machine descriptions, e.g., construction of frequently used machine instructions, folding of memory operands, or abstract interpretation of branch instructions. In addition, several backends implement target-specific low-level optimizations and transformations that are important for effective code generation.

- **JIT Infrastructure** The LLVM JIT is largely target-independent and shares most of the backend infrastructure that is also required for the static code generator. The main additional requirement is a description of the encoding of machine instructions in the corresponding `TableGen` description.

1.4 Preliminaries

This section gives a brief introduction into related areas that are heavily used in the subsequent chapters. For a more detailed description, the reader is referred to the provided citations.

1.4.1 Static Single Assignment Form

Static single assignment (SSA) form is a program representation in which each variable has a single static assignment in the source code [CFR$^+$91a]. This property does in general not hold for intermediate representations derived from generic imperative programming languages. Thus, multiple definitions of the same variable are usually disambiguated by sub-scripting them with unique version numbers. The main advantage of SSA form is that so-called use-def chains, i.e., a datastructure that maps a use of a variable to the set of potential reaching definitions, are explicit in SSA form. This simplifies several analyses and optimizations, e.g., constant propagation, dead code elimination, partial redundancy elimination, strength reduction, or value numbering. In practice, SSA form is usually linear in the size of the original program while use-def chains in general require a quadratic amount of space[2] .

The conversion of generic programs into SSA form is well-understood and, in general, requires the insertion of so-called Φ-nodes to disambiguate definitions of the same variable at program join points. These Φ-nodes always have as many arguments as a particular basic block has predecessors and its semantics are such that the result is the $\imath$-th argument if and only if control flow entered the block via the $\imath$-th predecessor.

To illustrate these concepts with an example, consider the code fragment in Figure 1.7, which shows the result of SSA conversion of the example to the left. The input program has two assignments for variable i. Therefore, it is not in SSA form. The code is

[2]There are worst-case examples for which SSA form is also quadratic in the size of the original program.

```
x = foo();                 x0 = foo();
if(x > 0)                  if(x0 > 0)
  i = 1;                     i1 = 1;
else                       else
  i = 2;                     i2 = 2;
                           i3 = Φ(i1,i2);
print(x, i);               print(x0, i3);
```

Figure 1.7: SSA form (right) for the program fragment to the left. So-called Φ-functions are used to disambiguate multiple definitions at join points in the control flow graph.

transformed into SSA form by splitting variable i into variable i_1 and i_2. A Φ-function merges the values of program variable i_1 and i_2 and assigns the result to variable i_3.

It is important to note that Φ-functions have special semantics that have to be carefully considered.

- Event though Φ-functions are usually denoted as a sequence of individual statements at the beginning of a linear code sequence, they have parallel semantics, i.e., *all* the arguments of Φ-functions in a particular block are read *before* they are written.
- Liveness is usually defined by the existence of a path from a given program point to a use of a particular variable that does not containing a definition. This does not hold for Φ-functions as a particular argument is only used when control flow enters via the corresponding predecessor. However, we can maintain the usual definition of liveness by thinking of Φ-functions as parallel copies of the corresponding right hand side to the variables at the left hand side at the very end of the particular predecessor block.

A simple SSA construction algorithm might attempt to insert Φ-functions at *every* join point. This is clearly unfeasible in practice as there will be a large number of unnecessary Φ-functions. Thus, most compilers generate so-called *pruned* SSA form, which contains Φ-functions only for variables that are actually live at a potential insertion point.

Several analyses and transformations require to insert code along *edges* in the control flow graph. This can be accomplished easily if either the source has only a single successor block or the target has only a single predecessor block by inserting the code in one of these blocks. If both conditions are violated, an edge is called *critical.* Critical edges can easily be removed by inserting an empty block along the edge. Programs in SSA form where all critical edges have been removed are in so-called *edge-split* SSA form.

Notation In the following, we consider control flow graphs $CFG(\mathcal{N}, \mathcal{E})$ in SSA form. The nodes $\mathbf{s}$ and $\mathbf{e}$ are distinguished nodes denoting the (artificial) start and end vertex respectively. Edges $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ denote the transfer of control in the program. We denote with $\mathcal{V}$ the set of program variables in *CFG*. Each node in *CFG* corresponds to a labeled single instruction ℓ of the form

$$\ell : (d_1, \ldots, d_m) \leftarrow \mathrm{op}(u_1, \ldots, u_n).$$

The sets $\mathcal{D}_\ell \subseteq \mathcal{V}$ and $\mathcal{U}_\ell \subseteq \mathcal{V}$ denote the subset of variables used and defined at ℓ respectively. For each $v \in \mathcal{V}$, $\mathrm{def}(v)$ denotes the unique label defining v. Each label ℓ has an ordered set of predecessors $\mathrm{preds}(\ell)$ which are denoted by $(\mathrm{pred}_\ell^1, \ldots, \mathrm{pred}_\ell^k)$. As usual, confluence points of multiple reaching definitions are disambiguated using Φ functions. A label ℓ *dominates* a label ℓ' (denoted by $\ell \succeq \ell'$) if all paths from $\mathbf{s}$ to ℓ' contain ℓ. Label ℓ *strictly* dominates ℓ', if $\ell \succeq \ell'$ and $\ell \neq \ell'$.

We say a variable v is live at label ℓ, if there is a path from ℓ to a usage of v that does not contain a definition of v. This definition is not precise in the context of Φ-functions as their result depends on the particular predecessor by which a label has been reached. Therefore, we treat Φ-arguments like virtual usages right after at the corresponding predecessor. The set of variables live at label ℓ is denoted by $\mathcal{L}_\ell$. Two variables $v, v' \in \mathcal{V}$ are said to *interfere* iff there exists a label in the program where both are live.

A program in SSA form is called *strict*, if each usage of a variable v is dominated by its definition $\mathrm{def}(v)$. The usage of strict SSA form is no restriction as we can easily add artificial definitions for undefined values without altering the program semantics. For some languages such as Java, strictness is even a requirement.

SSA Construction SSA form can be computed efficiently. The most widely used algorithm which is also used in this thesis is based on work by Cytron et al. [CFR$^+$91a]. The algorithm can be divided into three basic steps:

(1) Compute the so-called *dominance frontier* from the control flow graph

(2) For each variable, determine the location for Φ-functions

(3) Rename the original variables by sub-scripting them with unique version numbers

The *dominance frontier* for a label ℓ is the set of all nodes ℓ' such that ℓ dominates a predecessor of ℓ', but does *not* strictly dominate ℓ'. Loosely speaking, the dominance frontier describes the "border" between dominated and undominated labels. The dominance frontier can be computed efficiently in time proportional to the size of the original control flow graph plus the size of the dominance frontiers. Aside from pathological worst-case examples with very large dominance frontiers, the execution time of the algorithm is usually dominated by the size of the graph and thus runs in "practically" linear time. The required dominator tree can be easily computed using a classic bitvector data-flow algorithm. However, this might be very slow in the worst-case. A more efficient algorithm, which has been implemented for the experimental evaluation in this thesis, is the so-called Lengauer-Tarjan algorithm [LT79], which has time complexity $O((|\mathcal{N}| + |\mathcal{E}|)\log(\mathcal{N}| + |\mathcal{E}|))$.

The SSA construction algorithm outlined above is based on the observation that whenever a label ℓ contains a definition of a variable a, then any node ℓ' needs a Φ-function for a (*(iterated) dominance-frontier criterion*). Thus, the insertion of Φ-functions outlined in step (2) can be accomplished by a simple worklist algorithm. In the worst-case, the number of inserted Φ-functions can be quadratic in the size of the graph. Again, there is strong experimental evidence that, in practice, the number of inserted Φ-functions grows only linearly.

After Φ-functions have been inserted, the algorithm walks the dominator tree in step (3), renaming different definitions with a fresh sub-scripted version. Uses of a variable are renamed to reflect the closest definition of the variable that is above in the dominator tree. The algorithm has time complexity proportional to the size of the control flow graph after insertion of Φ-functions.

SSA Destruction At some point in the code generation process, Φ-functions have to be eliminated as they are not useful for execution on a real machine. The traditional approach is to translate a program in SSA form to an equivalent general program without Φ-functions. Although it is tempting to simply remove the Φ-functions and to drop the indexes, thereby assigning each occurrence of a variable its original name, this is an incorrect transformation in general. The reason is that program transformations on SSA form can lead to interfering live ranges of individual versions derived from the same variable.

The simplest algorithm for SSA elimination replaces each Φ-function of the form $v = \Phi(v_0, \ldots, v_n)$ with n move instructions of the form $v = v_i$ at the very end of the i-th predecessor. There are more elaborate strategies that minimize the number of additional move instructions [BCH$^+$02].

Recent work suggests that is beneficial to *maintain* SSA form until register allocation. This is the approach that is also taken in this thesis; see Chapter 4 for a detailed discussion.

Properties of Programs in SSA Form Programs in SSA form have a number of interesting properties. First, their interference graphs are chordal [HGG06, PP05, BDMS05]. A graph is called chordal if each cycle of length four or more has a *chord*, which is an edge joining two non-adjacent nodes in the cycle. As chordal graphs are a subset of perfect graphs, they inherit their properties. Most importantly, the chromatic number of perfect graphs equals the size of the largest clique. This property even holds for each induced subgraph. There is a so-called *perfect elimination order* that allows to color perfect graphs optimally in polynomial time. Furthermore, for strict SSA form, the following properties hold [BCH$^+$02]:

- If variables $v, w \in \mathcal{V}$ interfere, then either $\mathrm{def}(v) \succeq \mathrm{def}(w)$ or $\mathrm{def}(w) \succeq \mathrm{def}(v)$.
- If variables $v, w \in \mathcal{V}$ interfere and $\mathrm{def}(v) \succeq \mathrm{def}(w)$, then v is live at $\mathrm{def}(w)$.
- Each label ℓ at which a value v is live, is dominated by $\mathrm{def}(v)$ [HG06].

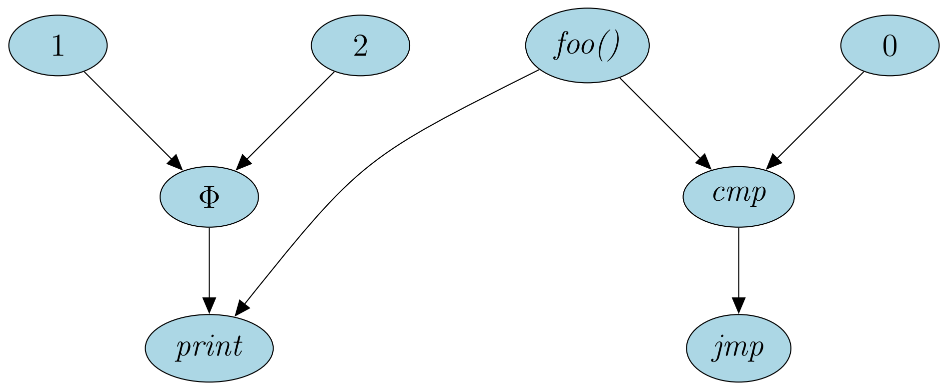


Figure 1.8: Example SSA graph for the control flow graph fragment shown Figure 1.7.

One of the most important properties for SSA-based register allocation is that for each clique C in the interference graph, there exists a label where all variables from C are live. As interference graphs are perfect, the size of the largest clique corresponds precisely to the number of registers required. This can easily be determined by computing the maximum number of variables live at some label.

SSA Graphs Large parts of this thesis make use of so-called *SSA graphs* [GSW95]. SSA graphs are an abstract representation of procedures in SSA form and accurately describe the flow of computation for the whole function. Nodes in the SSA graph represent a single operation while edges describe a flow of data that is produced at the source node to the target node. Note that incoming edges have an order which reflects the argument order of the particular operation. The edges of the SSA graph accurately describe the data dependencies among the operations. Note that there are no explicit nodes for the definition or use of a variable. Consequently, move-operations are omitted entirely from the graph. In contrast to classical tree or DAG representations, SSA graphs allow for *cycles* that describe the flow of computation across loop boundaries. To illustrate the concept of SSA graphs, Figure 1.8 shows the SSA graph for the code fragment introduced in Figure 1.7.

Formally, a SSA graph is denoted as a quadruple $G = (V, E, \mathrm{op}, \mathrm{opnum})$ with a set of nodes V, a set of edges $E \subseteq V \times V$, a function $\mathrm{op} : V \to \Sigma$, and a function $\mathrm{opnum} : E \to \mathbb{N}$. The set Σ is a ranked alphabet of operand symbols. Each node in V has an associated arity $\tau_V : V \to \mathbb{N}$. For an edge $e = (u, v)$, $1 \leq \mathrm{opnum}(e) \leq \tau_V(v)$ denotes the order of arguments for the operation $\mathrm{op}(v)$. For any node u, $|\mathit{preds}(u)| = \tau_V(u)$ and for any two incoming edges (v, u), (w, u) $v, w \in \mathit{preds}(u), v \neq w$ we require that $\mathrm{opnum}((v, u)) \neq \mathrm{opnum}((w, u))$. For all operations except Φ-nodes, the arity $\tau_V(u)$ of a node $u \in V$ and the arity of its operation $\tau_\Sigma(\mathrm{op}(u))$ are equal and can be used interchangeably. A (data) path π is a sequence of nodes $v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for all $1 \leq i < k$. A path is cyclic if there are several occurrences of a node in the path. The length of a path π is given by $|\pi|$.

1.4.2 Integer Linear Programming

Parts of this work make use of (integer) linear programming and duality theory. A linear program (LP) is an optimization problem consisting of an objective function and several constraints. George B. Dantzig proposed the following standard model:

$$\begin{gathered}\text{maximize } c^T x \\ \text{subject to } Ax \leq b \\ x \geq 0\end{gathered}$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$. The objective is to find a vector $\hat{x} \in \mathbb{R}^n$ such that $c^T \hat{x} = \max\{c^T x \mid Ax \leq b\}$.

Linear Programs can be solved in polynomial time. The most widespread algorithms are the simplex [Chv83], the ellipsoid and the interior point method [RT97]. Unfortunately, adding so-called *integrality constraints* renders the problem *NP*-complete in the general case [GJ79]. A special but very common case occurs when all variables have to be zero or one. This variant is called 0/1 integer linear program (ILP).

For every linear program

$$P = \max\{c^T x \mid Ax \leq b\}$$

(primal problem) the corresponding *dual problem* D is defined as

$$D = \min\{y^T b \mid A^T y = c^T, y \geq 0\}.$$

Primal and their dual problems are closely connected. If D is the dual problem of a primal problem P, then the dual of D is equal to P. Duality can be used to prove the optimality of linear programs in an elegant way:

Theorem 1.4.1 (Weak duality)**.** *If $\hat{x}$ is a feasible solution of P and $\hat{y}$ is a feasible solution of D, then $c^T \hat{x} \leq \hat{y}^T b$.*

Theorem 1.4.2 (Strong duality)**.** *If P has an optimum solution $\hat{x}$, then D has an optimum solution $\hat{y}$ and $c^T \hat{x} = \hat{y}^T b$.*

In other words, every feasible solution of D gives us a bound on the optimum value of D and we can use a solution to the dual problem as a certificate for the optimality of the primal problem.

1.4.3 Network Flow Theory

Given a digraph $G(V, E)$, a capacity function $c(u, v)$ for each edge $(u, v) \in E$, and two distinguished nodes s (source) and t (sink), the *maximum flow problem* is to maximize the net outward flow from s under the constraint that the flow into any vertex other than s and t equals the flow out of it and the flow along each edge (u, v) is non-negative and does not exceed its capacity $c(u,v)$ [AMO93]. More formally, the maximum flow problem can be stated as a linear combinatorial optimization problem as follows:

$$\begin{array}{ll} \max \sum_{(s,j)\in E} x_{s,j} & \\ \text{subject to} & \\ \sum_{(i,j)\in E} x_{i,j} - \sum_{(j,k)\in E} x_{j,k} \leq 0 & \forall j \in V \setminus \{s,t\} \\ & \textit{flow conservation constraints} \\ 0 \leq x_{i,j} \leq c_{i,j} & (i,j) \in E \\ & \textit{capacity constraints} \end{array} \tag{1.1}$$

The flow conservation constraints state that for each node, the total flow into this node is at most the total flow out of it. It is easy to show that if these inequalities hold for all nodes, the they must be in fact satisfied with equality as a deficit at one of the nodes implies a surplus at some other nodes.

A *minimum cut* is a separation of V into disjoint subsets S and T such that $s \in S$, $t \in T$, and $\sum_{(u,v)\in S\times T} c(u,v)$ is minimal. Edges $e \in S \times \bar{S}$ are called cut-edges.

A very famous statement in graph theory known as *max-flow min-cut theorem* [FD62] states that the maximum flow through a given network is exactly equal to the weight of a minimum cut. In fact, this is a special case of the more general strong duality theorem; see Theorem 1.4.2. Introducing new variables y_i for the right hand side of the flow conservation constraints in Equation 1.1 and variables $z_{i,j}$ for capacity constraints, we can derive the dual problem as follows:

$$\begin{array}{ll} \min \sum_{(i,j)\in E} c_{i,j} z_{i,j} & \\ \text{subject to} & \\ y_j - y_i + z_{i,j} \geq 0 & \forall (i,j) \in E \\ y_t = 1 & \\ y_s = 0 & \\ z_e \geq 0 & \\ y_i \geq 0 & \end{array} \tag{1.2}$$

Equation 1.2 is an exact characterization of the min-cut problem. Variable $z_{i,j}$ is one if (i,j) is a cut-edge and zero otherwise. The constraint matrix is *totally unimodular* [AMO93], i.e., the solution of the relaxed problem gives the optimal integral solution. This implies that a polynomial time solver for linear programming may be employed to solve the problem (e.g. interior-point). Better performing *s-t* min-cut algorithms exist [SW97] or max-flow algorithms can determine the solution of an instance of the *s-t* min-cut problem. For the maximum-flow problem, a variety of algorithms exist. Their complexity is usually in the order of $O(V^3)$, with some sophisticated algorithms slightly below.

2 Partitioned Boolean Quadratic Programming

Introduction Partitioned Boolean Quadratic Programming (*PBQP*) is a generalized quadratic assignment problem that has proven to be effective for a wide range of applications in embedded code generation, e.g., instruction selection, register assignment, address mode selection [Eck03], or bank selection for architectures with partitioned memory [SBX08]. Instead of problem-specific algorithms, these problems can be modeled in terms of generic *PBQP*s that are solved using a common solver library. *PBQP* is flexible enough to model irregularities of embedded architectures that are hard to cope with using traditional heuristic approaches.

In this chapter, we give a formal definition of *PBQP* and present two algorithmic approaches that have proven to be effective tools in practice, i.e., a polynomial time heuristic solver that is optimal for a subclass of *PBQP* and an exponential branch & bound algorithm. We give a simple proof that shows the *NP*-completeness of *PBQP* and give an intuitive graphical interpretation of the problem. In Section 2.3 we shortly describe the application of *PBQP* for various sub-problems of embedded code generators. Experimental evidence for the effectivity of the proposed algorithms is given in Section 2.4.

2.1 Problem Definition

A *PBQP* is a generalized quadratic assignment problem proposed by Scholz and Eckstein for various sub-problems of embedded code generation [SE02a, Eck03]. Consider a set of discrete variables $X = \{x_1, \dots, x_n\}$ and their finite domains $\{\mathbb{D}_1, \dots, \mathbb{D}_n\}$. A solution of *PBQP* is a simple function $h : X \to D$ where D is $\mathbb{D}_1 \cup \dots \cup \mathbb{D}_n$; for each variable x_i we choose an element d_i in $\mathbb{D}_i$. The quality of a solution is based on the contribution of two sets of terms:

1. for assigning variable x_i to the element d_i in $\mathbb{D}_i$. The quality of the assignment is measured by a *local cost function* $c(x_i, d_i)$.

2. for assigning two related variables x_i and x_j to the elements $d_i \in \mathbb{D}_i$ and $d_j \in \mathbb{D}_j$. We measure the quality of the assignment with a *related cost function* $C(x_i, x_j, d_i, d_j)$.

Thus, the total cost of a solution h is given as

$$f = \sum_{1 \le i \le n} c(x_i, h(x_i)) + \sum_{1 \le i < j \le n} C\left(x_i, x_j, h(x_i), h(x_j)\right). \tag{2.1}$$

PBQP asks for an assignment with minimum total costs.

We can alternatively formulate *PBQP* using matrix notation: a discrete variable x_i is represented as a boolean vector $\overrightarrow{x_i}$ whose elements are zeros and ones and whose length is determined by the number of elements in its domain $|\mathbb{D}_i|$. Each 0-1 element of $\overrightarrow{x_i}$ corresponds to an element of $\mathbb{D}_i$. An assignment of x_i to d_i is represented as a unit vector whose element for d_i is set to one. Hence, a valid assignment for a variable x_i is modeled by the constraint $\overrightarrow{x}_i^T\overrightarrow{1} = 1$ that restricts vectors $\overrightarrow{x_i}$ such that exactly one vector element is assigned one; all other elements are set to zero.

The related cost function $C(x_i, x_j, d_i, d_j)$ is decomposed for each pair (x_i, x_j). The costs for the pair are represented as matrix $\mathcal{C}_{ij}$. A matrix element corresponds to an assignment (d_i, d_j). Similarly, the local cost function $c(x_i, d_i)$ is mapped to cost vectors $\overrightarrow{c_i}$. Quadratic forms and scalar products are employed to formulate *PBQP* as a mathematical program:

Definition 2.1.1. *Let $X = \langle \overrightarrow{x}_1, \ldots, \overrightarrow{x}_n \rangle$ be a n-tuple of boolean decision vectors $\overrightarrow{x}_i \in \{0,1\}^{|\mathbb{D}_i|}$ and let $c_i \in \mathbb{R}^{|\mathbb{D}_i|}$ and $C_{i,j} \in \mathbb{R}^{|\mathbb{D}_i| \times |\mathbb{D}_j|}$ denote local and related cost vectors and matrices respectively. The partitioned boolean quadratic problem (PBQP) is defined as the following minimization problem:*

$$\min f(X) = \sum_{1 \le i \le n} \overrightarrow{x}_i^T \overrightarrow{c}_i + \sum_{1 \le i < j \le n} \overrightarrow{x}_i^T \mathcal{C}_{ij} \overrightarrow{x}_j. \tag{2.2}$$

$$s.t.\ \forall 1 \le i \le n : \overrightarrow{x}_i \in \{0,1\}^{|\mathbb{D}_i|} \tag{2.3}$$

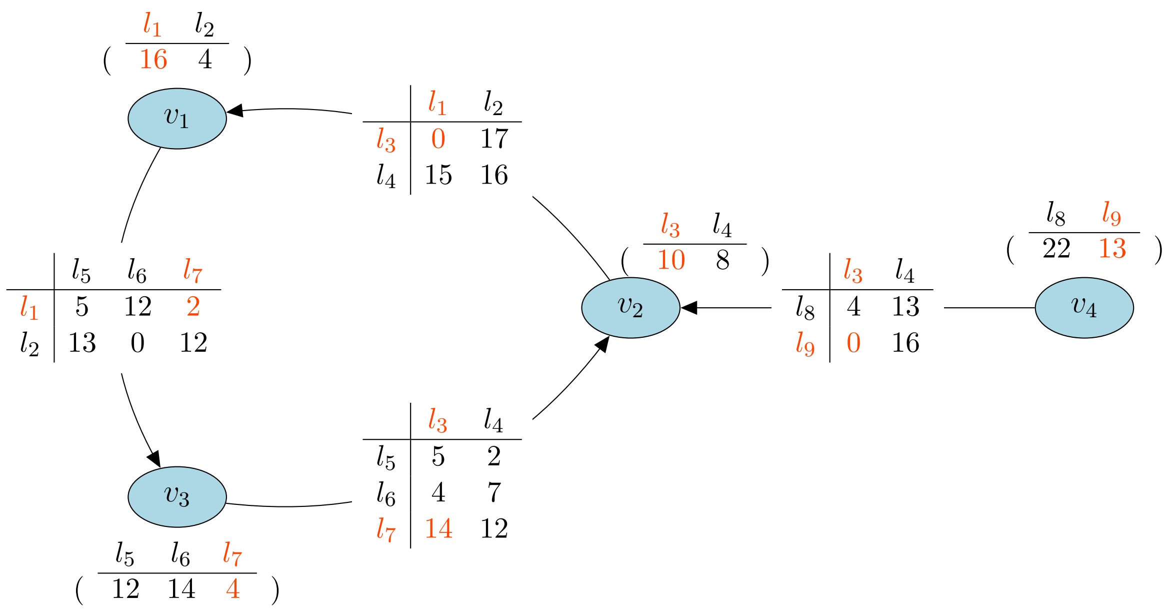
$$\forall 1 \le i \le n : \overrightarrow{x}_i^T \overrightarrow{1} = 1 \tag{2.4}$$

A solution satisfying Constraints 2.3 and 2.4 maps each boolean decision vector $\overrightarrow{x}_i$ to a binary vector that contains a single one element. This defines a one-to-one mapping among decision vectors $\overrightarrow{x}_i$ and elements in their domain $\mathbb{D}_i$. Thus, the domain for the objective function is the cross product of the domains for the individual variables $\mathbb{D}_1 \times \cdots \times \mathbb{D}_n$. The solution to a *PBQP* is not necessarily unique, i.e., there are in general multiple solution vectors $\bar{X}$ with the same minimal objective value.

PBQP-Graphs It is more instructive to denote a *PBQP* in a graphical representation. Such an interpretation can be obtained by an directed graph $G = (V, E, C, c)$. For each decision vector $\overrightarrow{x}_i$, we construct a node $v_i \in V$. Likewise, we introduce edges $e = (v_i, v_j)$ for each pair of decision variables whose cost matrix $\mathcal{C}_{i,j}$ is different from the zero matrix. The cost functions c and C map nodes and edges to the original cost vectors and matrices respectively.

Note, that due to the properties of quadratic forms, there is an implicit reverse edge $e' = (v_j, v_i)$ for each edge $e = (v_i, v_n)$ and $C(e) = C(e')^T$. Note also that a *PBQP*-graph is free of self- and multi-edges.

Example To illustrate the concepts introduced so far, image a multinational company that tries to optimize the location of its production sites. For each manufacturing base, there is a number of choices. Each of these choices has associated costs that represent



l_1 l_2 (16 4)
v_1
l_1 l_2 / l_3: 0 17 / l_4: 15 16
l_5 l_6 l_7 / l_1: 5 12 2 / l_2: 13 0 12
l_3 l_4 (10 8)
v_2
l_3 l_4 / l_8: 4 13 / l_9: 0 16
l_8 l_9 (22 13)
v_4
l_3 l_4 / l_5: 5 2 / l_6: 4 7 / l_7: 14 12
v_3
l_5 l_6 l_7 (12 14 4)

Figure 2.1: Example *PBQP* represented as a *PBQP*-graph. The optimal solution is highlighted and has objective value 59.

location-dependent expenses such as rent, taxes, and costs of labor. At the same time, goods have to be shipped among the different production sites. These costs depend on the particular choice for the location of the production sites involved. Some of the premises do not interact at all. Thus, their interaction costs are zero no matter where they are located. Denoting with x_i the decision vector for production site v_i, the example can easily be stated as a *PBQP* as follows:

$$\begin{aligned}\min f(X) = \\ \vec{x}_1^T \begin{pmatrix}16 & 4\end{pmatrix} + \vec{x}_2^T \begin{pmatrix}10 & 8\end{pmatrix} + \vec{x}_3^T \begin{pmatrix}12 & 14 & 4\end{pmatrix} + \vec{x}_4^T \begin{pmatrix}22 & 13\end{pmatrix} + \\ \vec{x}_1 \begin{pmatrix}5 & 12 & 2\\ 13 & 0 & 12\end{pmatrix} \vec{x}_3^T + \vec{x}_3 \begin{pmatrix}5 & 2\\ 4 & 7\\ 14 & 12\end{pmatrix} \vec{x}_2^T + \\ \vec{x}_2 \begin{pmatrix}0 & 17\\ 15 & 16\end{pmatrix} \vec{x}_1^T + \vec{x}_4 \begin{pmatrix}4 & 13\\ 0 & 16\end{pmatrix} \vec{x}_2^T\end{aligned}$$

The same example represented as a *PBQP* graph is shown in Figure 2.1. There are 4 production sites v_1 to v_4, three of which interact heavily. Node v_4 represents a site that exchanges goods only with site v_2. Edges are labeled with a cost matrix that represents the interaction costs among the adjacent nodes. A solution to the *PBQP* selects one of the possible locations for each production site such that the overall costs are minimized. For the given example, a valid solution vector would be $\bar{X} = \langle(1,0),(1,0),(0,0,1),(0,1)\rangle$ corresponding to the assignment highlighted in Figure 2.1. The optimal objective value for this example is 59.

In general, finding a solution to this minimization problem is *NP* hard. However, for many practical cases, the *PBQP* instances are sparse, i.e., many of the cost matrices $\mathcal{C}_{i,j}$ are zero matrices and do not contribute to the overall solution. Thus, optimal or near-optimal solutions can often be found within reasonable time limits.

2.1.1 Related Problems

It is sometimes tempting to reduce an optimization problem to a related problem for which efficient algorithmic approaches are available. In this section, we consider two choices that suggest them-self: linear programming and the quadratic assignment problem.

Integer Linear Programming We have already introduced (integer) linear programming in Section 1.4.2. A compact *linearization* of *PBQP* can be obtained as follows. First, the *PBQP* decision vectors $\overrightarrow{x}_i$ are mapped to 0-1 variables y_{ij} where j is in the range between 1 and $|\mathbb{D}_i|$. A constraint is added to the integer program that restricts the solution of y_{ij} such that exactly one of the variables is set to one, i.e., only one element of the domain is assigned to *PBQP* variable x_i. This gives us the following formulation:

$$\begin{aligned} \min f = \sum_{1\le i\le n}\sum_{1\le j\le|\mathbb{D}_i|} c(x_i,d_j)y_{ij} + \sum_{1\le i\le n}\sum_{1\le j\le|\mathbb{D}_i|}\sum_{1\le k\le n}\sum_{1\le l\le|\mathbb{D}_k|} C(x_i,x_k,d_j,d_l)y_{ij}y_{kl} \\ s.t.\, \forall 1\le i\le n : \forall 1\le j\le |\mathbb{D}_i| : y_{ij}\in\{0,1\} \\ \forall 1\le i\le n : \sum_{1\le j\le|\mathbb{D}_i|} y_{ij}E = 1 \end{aligned}$$

The remaining quadratic term $C(x_i,x_k,d_j,d_l)\ y_{ij}y_{kl}$ in the objective function for the matrix elements can be linearized using a standard technique [AF05] resulting in a quadratic number of 0-1 variables in the linear integer program.

Quadratic Assignment Problem The quadratic assignment Problem (QAP) [CEL98, KB57, BÇPP98] is a well-known *NP*-complete optimization problem that can be defined by the following problem statement: consider the set $\{1,2,\dots,n\}$ and two $n\times n$ matrices $A,B\in\mathbb{R}^{n\times n}$. Denoting with S_n the set of all permutations of $\{1,2,\dots,n\}$, the quadratic assignment problems is to find a permutation $\pi\in S_n$ that minimizes the following term:

$$\min_{\pi\in S_n}\sum_{i=1}^{n}\sum_{j=1}^{n} A_{\pi(i),\pi(j)}B_{i,j}. \tag{2.5}$$

A generalized variant of QAP (*generalized Koopmans-Beckmann QAP*) includes an additional matrix $F\in\mathbb{R}^{n\times n}$ that represents a linear term in the objective function:

$$\min_{\pi\in S_n}\sum_{i=1}^{n}\sum_{j=1}^{n} A_{\pi(i),\pi(j)}B_{i,j} + \sum_{i=1}^{n} F_{\pi(i),i}. \tag{2.6}$$

A frequently used motivation for QAP is the *facility location problem*, in which n facilities are to be assigned to n locations. In this case, A represents a flow matrix that denotes the amount of material being shipped among two facilities and B is a distance matrix that represents the distance among two locations. QAP has several further applications in various fields such as scheduling, statistical data analysis, or wiring problems in electronics.

A permutation $\pi \in S_n$ can be seen as a *bijection* from the set $1, 2, \ldots, n$ to itself, i.e., both the domain and the codomain are the same set and we are looking for an one-to-one mapping among them. Note that this is different from *PBQP* where each decision variable has its own domain.

Several variants of QAP have been considered in literature. The one with probably the most similarities to *PBQP* is the so-called *Quadratic Semi-Assignment Problem* (QSAP), which has been investigated by Malucelli and Petrolani [MP94, MP95]. The main difference to QAP is that n facilities have to be assigned to m locations while each of the locations can have 0, one, or many facilities assigned. Similar to the approach presented in Section 2.2.1, Malucelli and Petrolani identify a subclass of QSAP that can be solved in polynomial time.

2.1.2 Complexity

PBQP is well-known to be *NP*-complete. Eckstein presents a proof [Eck03] by reduction from MAXCUT, which is one of Karp's original *NP*-complete problems. An alternative proof has been proposed by Jackschitsch [Jak04] using a reduction from SAT. The same result can easily be obtained by means of the quadratic assignment problem which has been introduced above.

Theorem 2.1.2. *PBQP is NP-complete.*

Proof. We reduce an instance of the generalized Koopmans-Beckmann QAP with coefficient matrices $A, B, F \in \mathbb{R}^{n \times n}$ to *PBQP*. The *PBQP* graph is given by K_n — the complete graph with n nodes and $\frac{n(n-1)}{2}$ edges. Nodes are denoted by $v_1, \ldots, v_n$ and have domain $\{1, 2, \ldots, n\}$. The cost vector for node v_i corresponds to the i-th row of coefficient matrix F. Among any two nodes v_i and v_j, the cost matrix $\mathcal{C}_{i,j} = (c_{k,l})$ is defined as follows:

$$\mathcal{C}_{i,j} = (c_{k,l}) = \begin{cases} \infty & k = l \\ A_{k,l} B_{i,j} & \text{otherwise} \end{cases}$$

The ∞ coefficients in the diagonals of the constraint matrices make sure that no two facilities are assigned the same location. The objective value of the QAP corresponds exactly to a solution of the constructed *PBQP*. □

2.2 Algorithms for PBQP

In this section, we introduce two algorithmic approaches for *PBQP* that have been proven to be efficient in practice for sparse instances, i.e., a polynomial-time heuristic algorithm and a branch-&-bound based algorithm with exponential worst case complexity.

2.2.1 Heuristic Algorithm

In this section, we introduce an algorithm for *PBQP* proposed by Scholz and Eckstein [SE02a, Eck03]. For a certain subclass of *PBQP*, their algorithm produces provably optimal solutions in time $\mathcal{O}(nm^3)$, where n is the number of discrete variables and m is the maximal number of elements in their domains, i.e., $m = \max(|\mathbb{D}_1|, \ldots, |\mathbb{D}_n|)$. For general *PBQP*s, however, the solution may be sub-optimal.

Algorithm 1 *PBQP* Heuristic

```
{Phase I: reduction}
while $\exists v \in V : \deg v > 0$ do
  choose vertex $v \in V : 0 < \deg(v) \leq \deg(v') \; \forall v' \in V : \deg(v') > 0$
  if $\deg(v) == 1$ then
    RI(v)
  else if $\deg(v) == 2$ then
    RII(v)
  else
    RN(v) {solution may be sub-optimal if RN is applied}
  remove $v$ from the graph

{Phase II: trivial solution}
for all $v_i \in V$ do
  determine solution for $v_i$ by finding the minimum element in $c_i$

{Phase III: back-propagation}
re-insert the remaining nodes in reversed elimination order until a solution for the
original graph is obtained
```

The algorithm works in three phases; see Algorithm 1. First, the *PBQP* graph is reduced until only nodes of degree 0 are left. For these nodes, a solution can easily be found using a local minimum computation. In a last step, the eliminated nodes are re-inserted in reversed order thereby computing a solution for the original problem instance.

Phase I: Reduction

Eliminating nodes from the *PBQP* graph is equivalent to the elimination of decision vectors from the original problem. The algorithm removes nodes from the graph until only nodes of degree 0 remain. Nodes to be removed are chosen according to their degree in increasing order. In practice, this can be accomplished efficiently by putting nodes into buckets according to their degree. The algorithm selects a node from the bucket with the smallest index that is non-empty.

Nodes with degree one or two can be eliminated without loss of optimality using reductions RI and RII respectively. If the algorithm reaches a point where only nodes with degree three or more are left, the problem becomes irreducible and a heuristic is applied, which is called RN. The heuristic chooses a beneficial discrete variable and a good assignment for it by searching for local minima. The obtained solution is guaranteed to be optimal if the reduction RN is not used [Eck03].

RI Reduction Let v_i be a node with degree one and let v_j denote its only adjacent vertex in the *PBQP* graph, we can eliminate v_i and the incident edge (v_i, v_j) simply by removing them from the *PBQP* graph. The cost vector associated with v_j is increment by the minimum costs over all possible choices of v_i. More formally, the updated cost vector $\overrightarrow{c'}_j$ is given by

$$\overrightarrow{c}'_j(a) = \overrightarrow{c}_j(a) + \min(\mathcal{C}_{j,i}(a,:) + \overrightarrow{c}_i).$$

As for reduction RII, it is easy to show that the optimal objective value for the modified graph corresponds to the optimal value of the original problem. A formal proof is given by Eckstein [Eck03].

RII Reduction Reduction RII follows the same idea as reduction RI. The operation is applied to nodes v_i with degree two. The two adjacent nodes are denoted by v_j and v_k respectively. Vertex v_i is removed and the minimal costs for v_i depending on choices for v_j and v_k are added to the cost matrix $\mathcal{C}_{j,k}$. The new cost matrix $\mathcal{C}'_{j,k}$ is defined as follows:

$$\mathcal{C}'_{j,k}(a,b) = \mathcal{C}_{j,k}(a,b) + \min(\mathcal{C}_{j,i}(a,:) + \mathcal{C}_{k,i}(b,:) + \overrightarrow{c}_i).$$

RN Reduction *PBQP* graphs that cannot be reduced using reductions RI and RII are called *irreducible*. The smallest example for an irreducible graph is the complete graph with four nodes K_4. One possibility to deal with irreducible graphs is recursive enumeration. However, recursive enumeration has exponential worst-case complexity which makes it an infeasible approach for most applications. Instead, a heuristic called RN is applied that reduces a node using a local minimum computation. The basic idea is to make a decision as if the node and the adjacent vertices are disconnected from the rest of the *PBQP* graph. While reductions RI and RII can be shown to maintain the optimality of the obtained solution, RN does not and leads to sub-optimal solutions in general. Let v_i denote a vertex of degree three or more, we heuristically select the index with minimal costs from the cost vector $\overrightarrow{c}'$ defined as follows:

$$\overrightarrow{c}'(a) = \overrightarrow{c}_i(a) + \sum_{(v_i,v_j)\in E} \min(\mathcal{C}_{i,j}(a,:) + \overrightarrow{c}_j).$$

Let s_i denote such an index with minimal costs in $\overrightarrow{c}'$, it remains to update the cost vectors for adjacent nodes accordingly. For each adjacent node v_j, the new modified cost vector $\overrightarrow{c}'_j$ evaluates to $\overrightarrow{c}'_j = \overrightarrow{c}_j + \mathcal{C}_{i,j}(s_i,:)$.

Phase II: Trivial Solution

Upon completion of the reduction phase, there are only nodes with degree 0 left in the *PBQP* graph. This corresponds to a *PBQP* in matrix notation where all cost matrices are the zero matrix and the cost function is reduced to

$$\min f(X) = \sum_{1 \le i \le n} \overrightarrow{x}_i^T \overrightarrow{c}_i.$$

Since there are no dependencies among the decision variables, the minimization problem is equivalent to the evaluation of the term

$$\sum_{1 \le i \le n} \min_{x_i} \overrightarrow{x}_i^T \overrightarrow{c}_i.$$

Thus, we can determine the solution for each decision variable x_i by finding the smallest element in its associated cost vector c_i.

Phase III: Back-Propagation

Algorithm 2 Back-Propagation

```
for all $v_i \in V$ do
  $s_{v_i} = k : c_i(k) = \min(c_i)$
while $S$ not empty do
  pop node $v_i$ from S
  $\overrightarrow{c} = \overrightarrow{c}_i$
  for all $(v_j, v_i) \in E$ do
    $\overrightarrow{c} += \mathcal{C}_{j,i}(s_{v_j}, :)$
  $s_{v_i} = k : c(k) = \min(c)$
```

In the back-propagation phase, nodes are re-inserted in reversed elimination order. At each step, the solution for the newly inserted node can be easily determined as the decision vectors for adjacent nodes are already known; see [Eck03] for a formal proof. The algorithm for a *PBQP* graph $G = (V, E, C, c)$ is shown in pseudo-code in Algorithm 2. We denote with S the stack of eliminated nodes from Phase I.

2.2.2 Branch & Bound

Branch & Bound (*B&B*) is a very generic and efficient enumeration scheme for combinatorial optimization problems that has been successfully applied to a large set of NP hard optimization problems, e.g., integer programming, knapsack problem, traveling salesman problem, maximum satisfiability problem, or QAP, which has been introduced in Section 2.1.1.

The main idea is to arrange the search space such that large parts can be explored only implicitly. At any point in the optimization process, we maintain a pool of yet

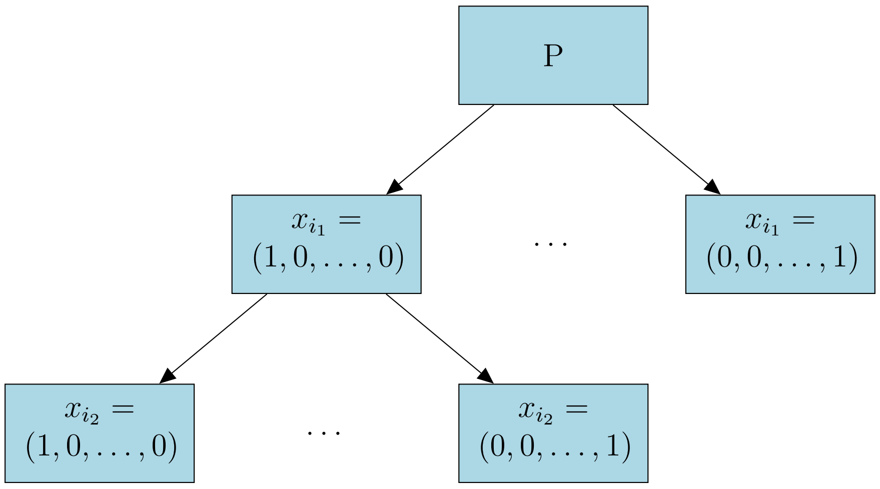

P
$x_{i_1} =$ $(1, 0, \ldots, 0)$
$\ldots$
$x_{i_1} =$ $(0, 0, \ldots, 1)$
$x_{i_2} =$ $(1, 0, \ldots, 0)$
$\ldots$
$x_{i_2} =$ $(0, 0, \ldots, 1)$

Figure 2.2: A fragment of the search tree for a *B&B* based *PBQP* algorithm.

unexplored subspaces together with the best solution found so far. Initially, there is only one subset representing the whole solution space and the best solution is set to ∞[1]. Subspaces are created dynamically and arranged in a so-called search tree. The key-idea is that large portions of these subspaces can be eliminated by comparing their lower bounds with the best solution found so far.

Any *B&B* algorithm consists of the following key ingredients:

1. *Bounding procedure:* for a given subspace of the solution space, compute a *tight* lower bound on the best solution value that is obtainable within this subspace.

2. *Selection strategy:* select the next live subproblem to be investigated in the search procedure. Popular techniques are breath first search (BFS), depth first search (DFS), and best first search. The latter always selects the subproblem with the lowest bound among the set of live subproblems. An experimental evaluation of the various strategies for QAP can be found in [CP99].

3. *Branching rule:* divide the considered subspace into two or more subspaces to be considered in subsequent iterations of the algorithm.

An advantage of *B&B* based algorithms that becomes more and more important with the rise of multi- and many-core systems is their ability to scale naturally with the number of cores.

PBQP allows for a natural mapping to the *B&B* scheme outline so far [HS06]. As in Section 2.2.1, reductions RI and RII can be applied until a irreducible graph remains. A subspace of the solution space is represented by a node in a search tree as shown in Figure 2.2. A subspace is divided into smaller subspaces by selecting a yet unconsidered decision vector x_i from one of the leaf nodes and creating $|\mathbb{D}_i|$ child nodes, each of which

[1]Without loss of generality, we assume minimization problems within this chapter.

representing one of the possible assignments for x_i. The whole set of partial assignments for a subspace of the solution space can be obtained by walking the search tree back to the root node.

A simple lower bound f_l on the objective function can be obtained using the following term:

$$f_l = \sum_{1 \le i \le n} \min c_i \sum_{1 \le i < j \le n} \min \mathcal{C}_{i,j}.$$

For inner nodes on the search tree, some of the variables x_i are already included in the partial assignment and evaluate to constants that replace the minimum computation and lead to tighter bounds in general. Note that we can derive the new lower bound after a branching operation in an incremental way using the lower bound of the parent node.

Leaf nodes that are not yet solved are called *live*. We maintain the set of live nodes in a priority queue according to their lower bound. The selection strategy is to expand the node with smallest lower bound first. This corresponds to the *best first search* strategy in the taxonomy introduced above.

A whole subspace can be pruned from the search if its lower bound exceeds a global upper bound on the value of the optimal solution. Initially, the global upper bound can be set to ∞. However, faster convergence can be achieved in general by applying a heuristic algorithm such as the method proposed in Section 2.2.1 in a pre-processing step. This will allow the *B&B* algorithm to prune large subspaces early in the optimization process.

2.3 Applications

PBQP has its origins in optimizing code generators for embedded systems. In the following, we give a brief description of three applications from this field and how they map to *PBQP*, i.e., instruction selection, register allocation, and addressing mode selection.

Instruction Selection We discuss instruction selection in detail in Chapter 3. The goal is to translate a compiler's intermediate code representation into a low-level intermediate representation or to machine code. A popular algorithmic approach for the instruction selection problem is pattern matching. The basic idea is to map instruction selection to a graph covering problem. The target instruction set is described using an *ambiguous* cost-annotated graph grammar. The instruction selector seeks for a cost-minimal cover. Each of the selected rules has an associated semantic action that is used to emit the corresponding machine instruction. Each rule consists of so-called terminal symbols and nonterminals. Terminal symbols match the corresponding labels of the dataflow trees. Nonterminals are used to chain individual rules together. Rules that translate from one nonterminal to another are called chain rules.

PBQP nicely maps to the pattern matching problem [EKS03, EBS$^+$08] for whole *SSA* graphs; c.f. Section 1.4.1. The input grammars are normalized such that they are either simple base rules that cover a single node in the SSA graph or chain rules that translate among nonterminals. The *PBQP* graph is structural equivalent to the SSA graph. The domain for each node is defined by the set of applicable base rules. Cost matrices along the edges of the SSA graph reflect the costs for the translation among different nonterminals. In practice, the *PBQP* instances are relatively sparse. Thus, the heuristic proposed in Section 2.2.1 is highly effective, even for very large graphs.

Register Allocation Register allocation aims to map an unlimited set of temporary variables produced by the instruction selector to a finite set of machine registers such that variables with interfering live ranges are assigned different registers. Variables that cannot be assigned during this process are mapped to locations in main memory, which is costly in almost every respect: code size is increased due to additional instructions transferring data to and from their assigned locations (spill code), performance is decreased due to large memory access latencies, and overall energy dissipation suffers from the additional workload.

Scholz and Eckstein were the first to propose *PBQP* for the register allocation problem [SE02b, Eck03, HKS03]. More recently, Hames et al. published experimental results for large benchmark sets using various *PBQP* algorithms [HS06]. The register allocation problem can be mapped to *PBQP* as follows: there is a node in the *PBQP* graph for each symbolic register in the considered code fragment. The domain for each node is basically defined by the set of machine registers that can be assigned to a particular live range. Additionally, each node can be assigned to a distinguished state *sp* that represents the option of spilling the corresponding variable to main memory. Local cost vectors reflect the costs for a particular assignment. For the *sp*-state, these are the costs for spilling the corresponding variables to main memory. The remaining elements correspond to the costs for the assignment of the associated machine register. Preferred registers can easily be modeled by adding a penalty to the undesired options.

Cost matrices along the edges of the *PBQP* graph are used to model the constraints of the register allocation problem. Most importantly, variables that are live at the same time in the program cannot be assigned to the same machine register. In this case, so-called *interference matrices* are introduced whose costs are ∞ if the particular choices for the incident nodes alias. Note that the distinguished spill option *sp* aliases with nothing, not even with itself.

Copy instructions from one variable to another can be eliminated if both the source and the target variable are assigned to the same machine register. This optimization is known as *coalescing* in register allocation literature and can easily be modeled by introducing so-called *affinity edges* with negative costs if the register for source and target nodes coincide. Furthermore, the *PBQP* based approach allows to model irregular architectures that are hard to cope with using traditional techniques.

Experimental results [HS06] show that the basic heuristic proposed in Section 2.2.1 performs poorly for large register allocation problems. However, the authors propose effective modifications to the general scheme that change the order in which irreducible nodes are processed. Their technique is similar to existing graph-coloring heuristics. The modified algorithm produces solutions whose costs are within 3% of the optimal solution.

Addressing Mode Selection Addressing mode selection (AMS) [ES03, Eck03] is an optimization for embedded architectures, e.g., DSP processors, with explicit address generation units. Even for simple architectures, AMS is a *NP* complete optimization problem [ORA$^+$01]. The goal is to select among different addressing modes supported by the target architecture, e.g., indirect addressing (with/without offset), pre-/post increment/decrement addressing, or modulo addressing. These modes often differ in their pipeline characteristics or in the size of their encoding.

The algorithm proposed by Eckstein et al. [ES03] performs address mode selection for each address register separately. The input for their algorithm is a control flow graph (CFG) of the program. For each node in the CFG, the AMS algorithm decides among the various addressing modes based on a cost model. The *offset value* denotes the difference among the value of the address register and the access value of a particular addressing mode. For an addressing mode, the offset values before and after execution of an instruction are called entry and exit offsets and are denoted by the pair $\langle e_i, x_i \rangle$. For some addressing modes there is only one such pair, e.g., indirect addressing is specified using the set $\{\langle 0, 0 \rangle\}$, post increment addressing can be described by $\{\langle 0, 1 \rangle\}$, and so on. However, for several addressing modes, these sets can become large, e.g., indirect addressing with offset is characterized by the set $\bigcup_{c \in C} \{\langle -c, -c \rangle\}$.

The main idea of addressing mode selection is to shift the value of the address register in consideration between consecutive instructions, i.e., each instruction i is replaced by the sequence $ar = ar - e_i; i; ar = ar + x_i$. To maintain the program semantics, exit and entry values of consecutive instructions must match. A peephole optimization is used to eliminate sequences of add instructions and to rewrite the addressing modes.

The mapping of AMS to *PBQP* includes decision vectors whose domain is defined by the set of offset values. A node in the *PBQP* graph represents a set of edges from the CFG that share a common source or destination node. On the other hand, edges in the *PBQP* graph correspond to nodes in the CFG.

There are no local costs for nodes in the *PBQP* graph, i.e., all cost vectors c_i are zero vectors. Cost matrices among nodes in the *PBQP* graph represent the minimum costs among all applicable addressing modes given the offset assignments at the source and destination node. A configurable cost function can be used to optimize for performance, code size, or a trade-off among both.

Experimental results show that *PBQP* graphs generated in this way are hardly irreducible. This means that the heuristic introduced in Section 2.2.1 delivers optimal results for most cases in practice. The reason is that these control flow graphs are derived from structural high-level languages like `C`. However, in general these languages include features such as `goto` that can lead to irreducible graphs.

data-set	#problems	reducible		optimal		*B&B* solved	
BANKSEL	14	8	57.2%	14	100.0%	14	100.0%
ALLOC	3.020	715	23.7%	819	27.1%	2837	93.9%
ISEL	181.202	178.727	98.6%	180.886	99.8%	181.201	100.0%

Table 2.1: Solver statistics for three different data-sets derived from real-world applications.

2.4 Experimental Evaluation

We implemented and evaluated both the heuristic and the *B&B* based algorithm described in Section 2.2. First, we consider three different *PBQP* data-sets obtained from real-world applications:

- BANKSEL: these are instances for a bank selection problem for architectures with partitioned memory [SBX08]. The data-set represents smaller embedded benchmarks.
- ALLOC: these are instances from a *PBQP* based register allocator implemented for LLVM. The data files have been obtained using the SPEC2000 suite, which represents large real-world applications. The target architecture is X86.
- ISEL: instances in this data-set have been obtained from an ARMv5 instruction selector based on *PBQP*. Again, SPEC2000 has been used. While the register allocator processes a function at a time, the instances in this data-set represent pattern matching problems for individual blocks.

Solver statistics for these data-sets are given in Table 2.1. All benchmarks have been executed on an Intel Xeon based workstation running at 3GHz. The column "reducible" shows the number of graphs that can be fully reduced using reductions RI and RII. Thus, the heuristic proposed in Section 2.2.1 is guaranteed to deliver an optimal solution. For the data-set ISEL, this is the case in about 98.6% of the cases, i.e., the *PBQP* graphs are very sparse. Much harder instances are contained in the data-set ALLOC, where more than 76% of all instances are irreducible.

The *B&B* based algorithm could solve all instances from data-sets BANKSEL and ISEL within a time limit of 300 CPU seconds. This picture changes considering data-set ALLOC. More than 180 problems could not be solved within the time limit. Increasing the time limit does not change this picture significantly. The optimal solutions obtained from the *B&B* algorithm confirm that the heuristic solutions for irreducible graphs are optimal in almost all cases for data-sets BANKSEL and ISEL. For the register allocator, the majority of all solutions is suboptimal and requires problem specific heuristics [HS06].

To show that the heuristic algorithm scales nicely, we generate a set of random *PBQP* instances with varying number of nodes. Among any pair of nodes, we create an edge with probability p which varies from 10% to 60%. In each setting, we generate 100 randomly generated problems. Note, that the number of edges grows quadratically in

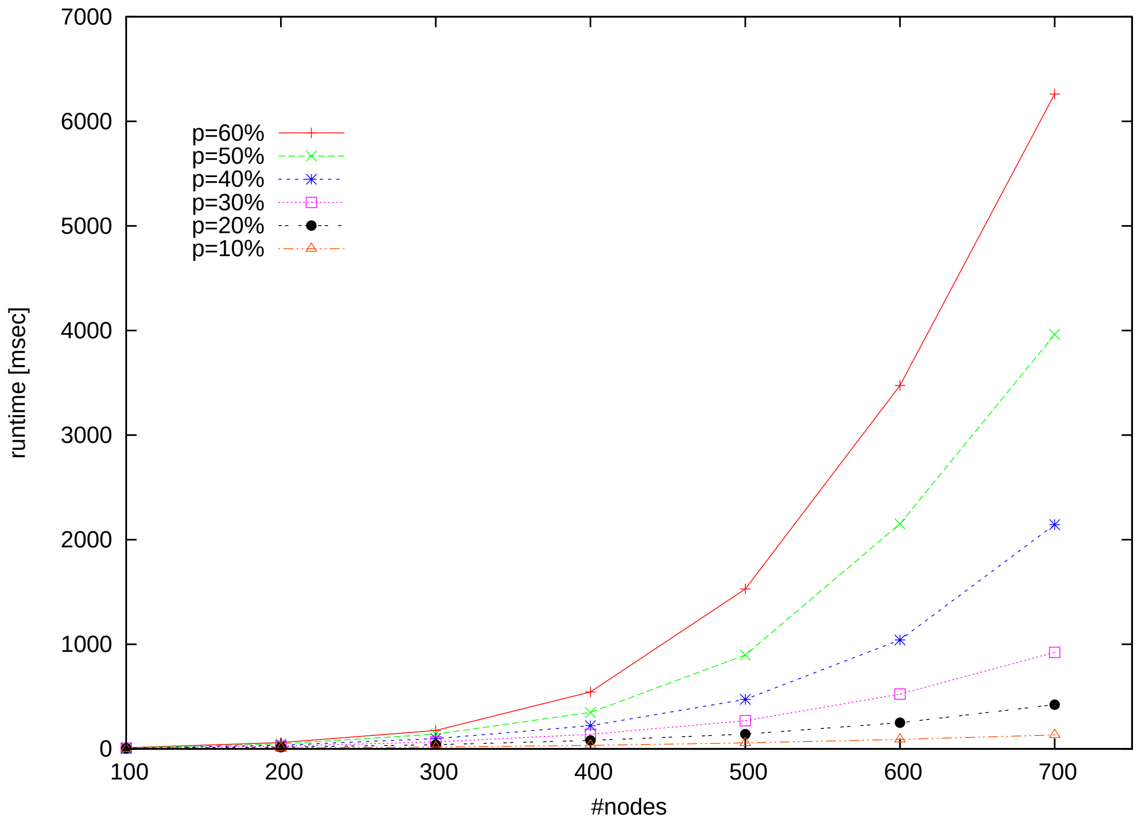


Figure 2.3: Average runtime for the heuristic algorithm on a large set of randomly generated *PBQP* problems for various densities.

	p=10%	p=20%	p=30%	p=40%	p=50%	p=60%
100	44.52	63.55	73.24	79.16	83.21	86.24
200	125.57	155.13	168.38	175.85	181.05	184.76
300	214.52	250.79	265.90	274.40	279.79	283.90
400	307.62	347.74	364.44	373.32	379.12	383.42
500	402.30	445.94	463.18	472.66	478.73	483.00
600	498.26	544.24	562.06	572.23	578.44	582.69
700	595.14	643.03	661.59	671.45	678.03	682.48

Table 2.2: Average number of necessary RN reductions for the randomly generated *PBQP* graphs.

the number of nodes, e.g., graphs with 700 nodes and density $p = 70\%$ contain on average 147.000 non-zero cost matrices. Figure 2.3 shows the runtime of the heuristic algorithm for these graphs with various edge probabilities. Note that the runtime is not strictly linear in the number in the nodes. The reason is that we select the RN node to be reduced heuristically. The strategy selects nodes with maximum degree and minimal local minimum. Using this strategy, the algorithm becomes quadratic in the number of nodes for bounded domains in the worst-case.

All the graphs in the random data set contain a relatively large number of edges compared to the sparse graphs obtained from real-world data sets. There are no reducible graphs. Thus, the computed solutions are not optimal in general. The average number of necessary RN reductions for the various settings is shown in Table 2.2. The table clearly shows that most nodes require RN reductions (87% over the whole benchmark set). This is very different from our real-world data-sets. For the benchmark set ISEL, the maximum percentage of RN reductions is 23%, with only 0.1% on average. Instances from data-set BANKSEL show similar characteristics (0.8% max, 0.2% on average). The instances obtained from the register allocation problem are significantly harder with a maximum of 70% and an average of 13.6% of RN reductions.

Solving *PBQP* problems with a large number of irreducible nodes to optimality with the *B&B* solver is hopeless. While the algorithm performs very well for sparse real-world data-sets, almost none of the problems from the randomly generated instances can be solved within a time limit of 300 CPU seconds. This is consistent with closely related problems [CEL98] for which experimental results suggest that optimal techniques are feasible up to about 30 nodes.

To show to which extend the *B&B* based algorithm is applicable, we consider a set of random sparse graphs that resembles the instances derived from practical applications more closely. Instead of varying edge probabilities, we select p such that the number of edges is proportional to the number of nodes by a factor $\rho \approx \frac{|E|}{|V|}$. Again, we consider graphs with varying number of nodes, generating 100 random samples for each configuration.

The experimental results for both the heuristic algorithm and the *B&B* algorithm are given in Table 2.3. The top left element of each cell shows the number of graphs out of 100 samples for which the heuristic could compute a provably optimal solution, i.e., they where reducible. The top right element denotes the number of graphs for which the *B&B* algorithm could compute an optimal solution within a time limit of 300 CPU seconds. Note that both the number of reducible graphs and the number of instances solved to optimality is strictly decreasing with increasing ρ and with an increasing number of nodes. The bottom left number denotes the number of graphs for which an optimal solution is known and for which the heuristic produced a suboptimal result. Note that in the vast majority of cases, the heuristic produced actually an optimal solution even though RN reductions had to be applied. Even if the result is suboptimal, the result is very close to the optimal solution. The bottom right number shows the average ratio in percent of the heuristically computed objective value compared to the optimum solution from the *B&B* algorithm. We only consider problems that are known to be suboptimal. Even in these cases, the average ratio never falls below 99%, i.e., the produced solutions

	ρ=0.7		ρ=0.8		ρ=0.9		ρ=1.0		ρ=1.1		ρ=1.2	
100	93	100	65	100	34	100	13	100	1	100	0	93
	3	99.55	10	99.45	20	99.52	29	99.47	50	99.42	60	99.23
200	89	100	48	100	21	100	2	90	0	55	0	17
	1	99.72	10	99.65	24	99.68	47	99.57	43	99.61	13	99.45
300	86	100	37	99	7	89	0	59	0	9	0	0
	2	99.82	17	99.83	36	99.77	41	99.66	6	99.77	-	-
400	86	100	37	99	3	78	0	23	0	0	0	0
	0	-	14	99.87	29	99.84	13	99.84	-	-	-	-
500	83	100	30	100	3	62	0	9	0	0	0	0
	2	99.88	19	99.91	34	99.87	7	99.85	-	-	-	-

Table 2.3: Solver statistics for a random set of sparse graphs. Each cell containts the number of reducible graphs for which the heuristic delivers optimal solutions (top left), the number of graphs solved to optimality by the *B&B* solver within 300 CPU seconds (top right), the number of graphs for which an optimal solution is known and the heuristic produced an actual suboptimal result (bottom left), and the average quality among these problems compared to the optimal solution (bottom right).

are very close to the optimum.

The runtime of the *B&B* algorithm is largely determined by the number of RN nodes in practice. Figure 2.4 shows the solver time for the *B&B* algorithm on the sparse graphs used before compared to the number of RN nodes. Note that the y-axis is drawn with logarithmic scale. The algorithm is clearly exponential in the number of RN nodes for the considered instances.

Short Summary Quadratic assignment problems are among the hardest optimizations problems considered in literature. *PBQP* is a generalized variant that has proven to be useful for various subtasks of embedded code generators. For most applications in practice, the arising problems are relatively sparse and can be solved to optimality even for large problems. Our results show that a previously proposed heuristic performs well in these cases and produces provably optimal solutions or solutions that are very close to the optimum. We compare this technique with a new *B&B* based algorithm that is exponential in general. In practice, its runtime depends largely on the number of irreducible nodes. For our real-wrold instances, most problems could be solved within a time limit of 300 CPU seconds.

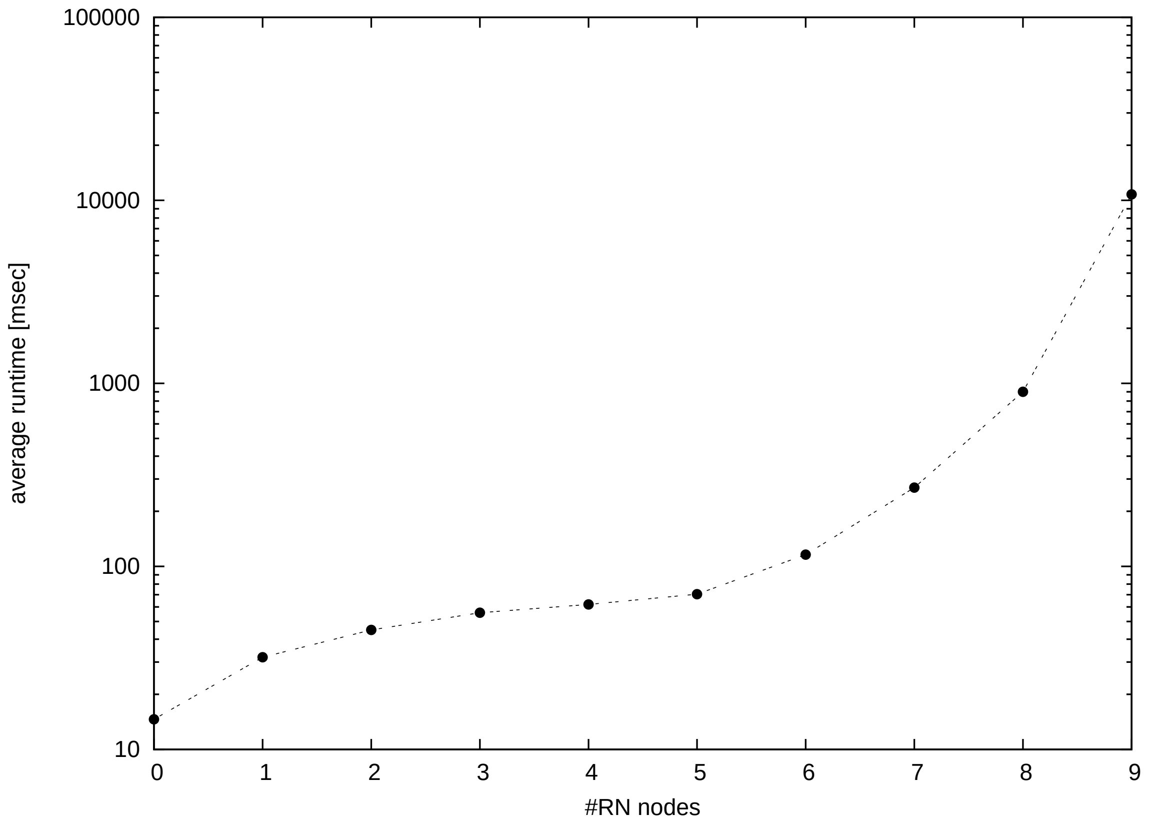


Figure 2.4: Average runtime for the *B&B* algorithm compared to the number of irreducible nodes. Note the logarithmic scale on the y-axis.

3 Code Generation for SSA Graphs

3.1 Introduction

Instruction selection is a transformation step in a compiler which translates the intermediate code representation into a low-level intermediate representation or to machine code; c.f. Figure 3.1. Usually, a machine description is used to adapt a generic algorithm for a particular instruction set. The various techniques found in literature differ in when this specialization happens, i.e., during compile time or even before during compiler compile time by generating a specialized version of the algorithm for each target machine.

Most Standard techniques confine their scope to statements or basic blocks achieving only locally optimal code. In this chapter, we describe and evaluate a new technique that extends previous work by Eckstein et al. [EKS03, Jak04] and that is able to perform instruction selection for whole functions in SSA form.

Our approach uses a discrete optimization problem for selecting instructions. Similar to classic tree pattern matching [Ert99, FHP92a], this approach maps the instruction selection problem to a graph grammar parsing problem where production rules have associated costs. The grammar parser seeks for a cost minimal syntax derivation for a given input graph. The parsed graph is the SSA graph [GSW95]; c.f. Section 1.4.1.

Previous approaches restrict patterns to trees such that complex patterns with multiple inputs and multiple results cannot be matched. For example, the `DIVU` instruction in the Motorola 68K architecture performs the division and the modulo operation for the same pair of inputs. The approach in [EKS03] cannot take advantage of coalescing both operations into a single `DIVU`. Other examples of instructions are the `RMW` (read-modify-write) instructions on the IA32/AMD64 architecture, autoincrement- and decrement addressing modes of several embedded systems architectures, the `IRC` instruction of the HPPA architecture, and `fsincos` instructions of various math libraries. These complex patterns are usually handled in tree-based approaches using a local peephole optimizer in a post-processing step for code strengthening or are exposed to the programmer in the form of compiler known functions (*intrinsics*) requiring significant efforts. To overcome these deficiencies, we introduce in this chapter an algorithm that is able to handle general graph patterns with arbitrary cost functions while accounting for potential memory dependencies [EBS$^+$08].

This chapter is organized as follows: In Section 3.2, we give an introduction to instruction selection and survey related work. Our generalized approach is motivated in Section 3.3, and in Section 3.4 we outline the algorithm for instruction selection. Implementation details are given in Section 3.5 and in Section 3.6 we discuss experimental

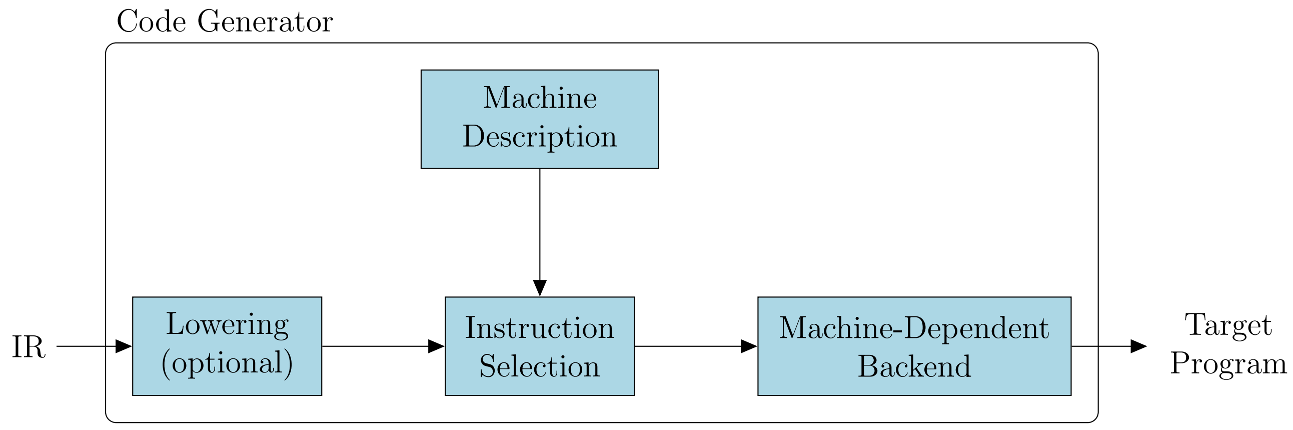


Figure 3.1: An instruction selector translates a compiler's IR to a low-level machine-dependent representation.

results.

3.2 Related Work

Due to its significant contribution to the overall code quality of a compiler, instruction selection received a lot of attention in the recent past [ECG06, GLM$^+$06, DR06, EKS03, LB00, Ert99, NK97]. In this section, we give a brief overview of the most important techniques found in literature ordered by increasing scope. This includes efficient local techniques such as RTL-based instruction selection, (tree) pattern matching, and whole-function instruction selection based on *PBQP*. The latter is the basis for the generalized pattern matching algorithm proposed in this chapter and is thus described in more detail.

With few exceptions, most of these approaches are limited to tree-shaped patterns. A notable exception are some integrated techniques that combine instruction selection with additional subproblems such as register allocation and scheduling. Keßler and Bednarski propose an enumeration scheme for such an integrated approach [BK04, KB06] based on branch & bound techniques. The authors do not implement pattern matching as discussed in Section 3.2.2. However, instructions in their machine description are allowed to be composed of several tree-shaped subgraphs (forest patterns) or general DAGs. An enumeration algorithm inductively searches the solution space, which is organized in so-called enumeration trees. As their algorithm solves an integrated code generation problem, it is only suitable for relatively small regions known to be important for overall performance. The authors report that problems up to 20 to 40 nodes can be solved to optimality in practice.

Another approach that is able to deal with a special class of DAG patterns, i.e., SIMD (single instruction multiple data) instructions, has been presented by Leupers and Bashford [LB00]. The authors propose to decompose data flow graphs into simple trees that can be processed using standard tree pattern matching techniques. Instead of computing only the cheapest tree cover, all possible alternatives are stored during the labeling phase. Later, an integer linear programming based algorithm is applied that selects SIMD instructions by selecting among the possible alternatives. Constraints in this model ensure consistency and a valid packing of individual nodes to SIMD instructions. The objective function is to maximize the use of SIMD instructions across the data flow graph.

A similar heuristic technique with support for so-called multi-output instruction has recently been proposed by Scharwächter et al. [SYL+07]. The authors consider instruction set extensions that execute frequently used tuples of instructions concurrently. These patterns are also a special case of generalized DAG patterns considered in this work. Again, so-called candidate sets are precomputed that denote the set of applicable tree rules. In general, a node can be a candidate for several multi-output instructions. The problem of finding a covering for these shared nodes is a maximum weighted independent set problem that is solved heuristically.

3.2.1 RTL-Based Instruction Selection

Register Transfer Lists (RTL) are a popular machine-independent intermediate representation used in several compiler infrastructure such as Zehpyr/VPO [RD98] or GCC [GCC]. A RTL is a simultaneous composition of list of effects. An effect computes the value of an expression and stores it in a location, which is either a single mutable storage cell or an aggregate of mutable cells. The value of an expression can either be constant, fetched from a storage location, or the result of the application of an operator to a list of argument expressions. Even though an RTL is machine-independent, its semantics *do* depend on the particular target architecture.

Algorithm 3 RTL-based code generation scheme.

```
1: Forward substitution
2: Combination of independent effects
3: Removal of useless effects
4: Validation based on a machine-description
```

Code generators in RTL-based compilers are based on the four-step scheme depicted in Algorithm 3. Forward substitution forms more complex expressions by substituting sub-expressions with their defining RTL. Likewise, step (2) combines individual RTLs to a single expression. After application of these rules, the results are verified against the machine descriptions. Step (3) simplifies redundant effects that might result from the application of the previous steps. RTL expressions that do not represent a native instruction are discarded in step (4).

The scope of this approach is limited to simple patterns and constitutes a form of "poor-man's instruction selection" achieving only locally optimal code. Thus, the code quality of RTL-based compilers such as GCC strongly depends on post-pass RTL-based optimizations such as common subexpression elimination (CSE) that make up for missed opportunities.

```
imm <- IMM                : 0
reg <- REG                : 0
reg <- imm                : 1
reg <- SHL(reg, reg)      : 1
reg <- SHL(reg, imm)      : 1
reg <- ADD(reg, reg)      : 1
reg <- LDW(reg)           : 1
reg <- LDW(ADD(reg, reg)) : 1
reg <- LDW(ADD(reg, SHL(reg, imm))) : 1
```

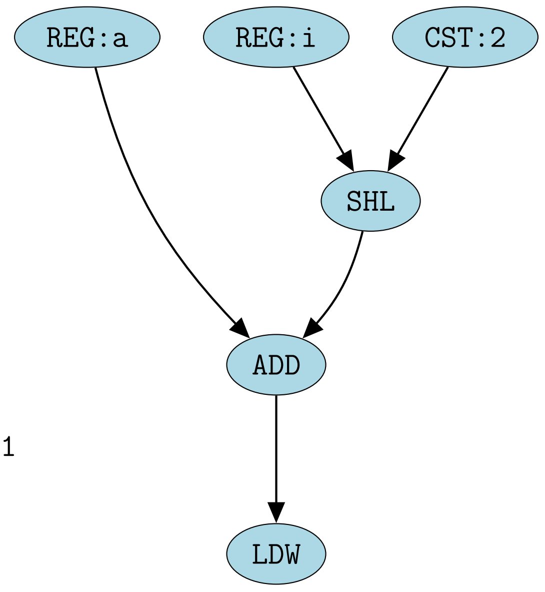


Figure 3.2: Example of a data flow tree and a rule fragment with associated costs.

3.2.2 Tree Pattern Matching

Tree pattern matching is a well known and widely used technique for instruction selection introduced by Aho and Johnson [AJ76]. The unit of translation is a single statement represented in the form of a data flow tree(DFT). The basic idea is to describe the target instruction set using an *ambiguous* cost-annotated graph grammar. The instruction selector seeks for a cost-minimal *cover* of the DFT. Each of the selected rules has an associated semantic action that is used to emit the corresponding machine instruction, either by constructing a new intermediate representation or by rewriting the DFT bottom-up.

An example DFT along with a set of rules representing valid ARM instructions is shown in Figure 3.2. Each rule consists of terminal symbols (denoted in upper-case) and nonterminals (lower-case). Terminal symbols match the corresponding labels of the dataflow trees. Nonterminals are used to chain individual rules together. Rules that translate from one nonterminal to another are called chain rules. Note that there are multiple possibilities to obtain a cover of the data-flow tree using the rule fragment to the left. Each rule has associated costs. The cost of a tree cover is the sum of the costs of the selected rules.

Aho and Johnson [AJ76] were the first to propose a dynamic programming algorithm in order to obtain cost-minimal covers. Balachandra et al. [BDB90] present an important extension that reduces the algorithm to linear time by precomputing *itemsets*, i.e., static lookup tables, at compiler compile time. Their approach is based on the following two-pass scheme:

(1) *labeling:* use dynamic programming in order to determine a minimum-cost cover of the given DFT bottom-up.

(2) *reduce*: traverse the DFT in postfix order and execute the semantic actions associated with the chosen rules to obtain a semantically equivalent target program.

The same technique was applied by Fraser et al. [FHP92a] in order to develop `burg` — a tool that converts a specification in the form of a tree grammar into an optimized tree pattern matcher written in `C`. While `burg` computes costs at generator generation time and thus requires constant costs, `iburg` [FHP92b] can handle dynamic costs by shifting the dynamic programming algorithm to instruction selection time. This allows the use of dynamic properties for cost computations, e.g., concrete values of immediates. The additional flexibility is traded for a small penalty in execution time. Ertl et al. [ECG06] save the computed states for tree nodes in a lookup table. This approach retains the flexibility of dynamic cost computations at nearly the speed of precomputed tree parsing automata.

In general, these approaches are limited in scope to simple statements. *DAG* matching techniques are an approach to overcome these limitations. However, *DAG* matching is an *NP*-complete problem [Pro98].

Ertl [Ert99] presents a generalization of tree pattern matching for *DAG*s. The algorithm makes its choice of rules as if it was parsing a tree, irrespective of sharing of subgraphs. This approach does not lead to optimal solutions in general. However, the author shows that it does for a certain class of grammars. A checker (DBurg) can determine if a given grammar is within this class or not. The author also shows that several grammars derived from practice indeed are DAG optimal or have only few rules that may produce suboptimal results.

Liao et al. [LDKT95]present a *DAG* matcher based on a mapping to the binate covering problem, which is known to be *NP*-complete. The authors consider accumulator-based architectures that require implicit partial scheduling of the nodes in the DAG. Both branch & bound and heuristic algorithms are used to solve these problems. The authors do consider data transfer costs and memory spill costs but do not use general tree or graph grammars as discussed before. Experimental results show that optimal solutions can only be found for small to moderate sized basic blocks. In many cases, these optimal solutions are better than heuristically produced results. Large basic blocks can either be broken into simpler basic blocks, which can be covered using their exact algorithm, or heuristics can be used that restrict the number of matches.

3.2.3 PBQP-Based Instruction Selection

PBQP-based instruction selection extends the scope of traditional *DAG*-based approaches to the computational flow of a whole function, which leads to cyclic data flow trees in general. Clearly, the *NP*-completeness of *DAG* matching extends to SSA graphs as well. As shown by Eckstein et al. [EKS03] the instruction selection problem is modeled as *PBQP* in a straightforward fashion. A solution for the *PBQP* instance induces a complete cost minimal cover of the SSA graph. The *PBQP* formulation overcomes many of the deficiencies of traditional techniques [FHP92a, FHP92b, AJ76], which often fail to fully exploit irregular instruction sets of modern architectures and need to employ ad-hoc techniques for irregular features (e.g., peep-hole optimizations, etc.). Instead of acyclic data flow trees, the authors propose so-called SSA-graphs; c.f. Section 1.4.1.

In the *PBQP* based approach [EKS03] an ambiguous graph grammar consisting of tree patterns with associated costs and semantic actions is used to find a cost-minimal cover of the SSA-graph. The input grammar is normalized, i.e., each rule is either a *base rule* or a *chain rule*. A base rule is a production p of the form $\texttt{nt}_0 \leftarrow op(\texttt{nt}_1, \dots, \texttt{nt}_{k_p})$ where $\texttt{nt}_i$ (for all i, $0 \leq i \leq k_p$) are non-terminals and op is a terminal symbol (i.e. an operation that is represented as a node in the SSA graph). A chain-rule is a production of the form $\texttt{nt}_0 \leftarrow \texttt{nt}_1$, where $\texttt{nt}_0$ and $\texttt{nt}_1$ are non-terminals. A production rule $\texttt{nt} \leftarrow op_1(\alpha, op_2(\beta), \gamma))$ can be normalized by rewriting the rule into two production rules $\texttt{nt} \leftarrow op_1(\alpha, \texttt{nt}', \gamma)$ and $\texttt{nt}' \leftarrow op_2(\beta)$ where $\texttt{nt}'$ is a new non-terminal symbol and α, β and γ denote sequences of operands of arbitrary length. This transformation can be iteratively applied until all production rules are either chain rules or base rules.

The instruction selection problem for SSA graphs is modeled in *PBQP* as follows. For each node u in the SSA graph, a *PBQP* variable x_u is introduced. The domain of the variable x_u is the subset of base rules $R_u = \{r_1, \dots, r_{k_u}\}$ whose operations op match the operation of the SSA node u. The cost vector $\overrightarrow{c}_u = w_u \cdot \langle \text{cost}(r_1), \dots, \text{cost}(r_{k_u}) \rangle$ of variable x_u encodes the costs of selecting a base rule r_i where $\text{cost}(r_i)$ denotes the associated cost of base rule r_i. Weight w_u is used as a parameter to optimize for various objectives including speed (e.g. w_u is the expected execution frequency of the operation in node u) and space (e.g. the w_u is set to one).

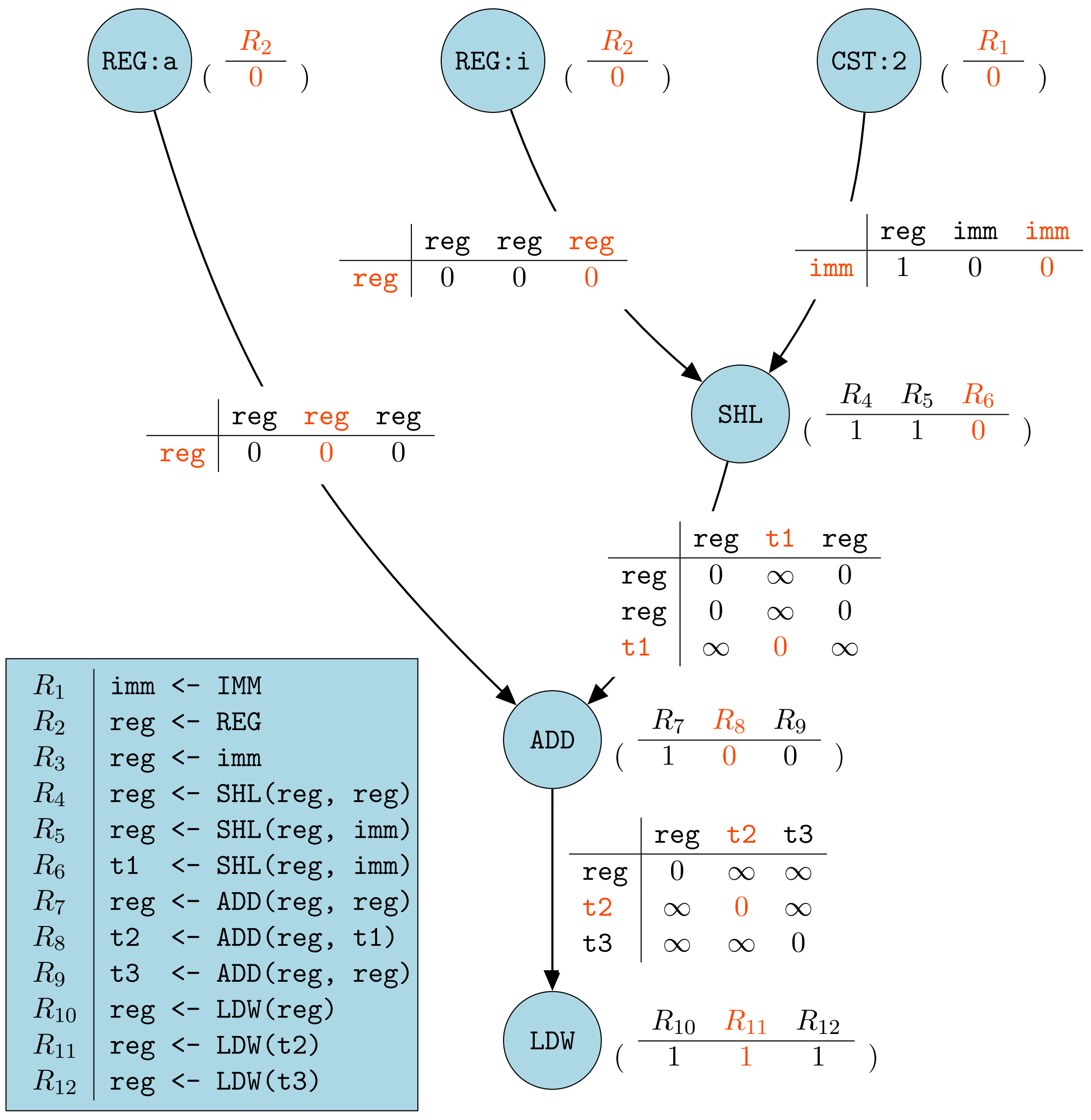
An edge in the SSA graph represents data transfer between the result of an operation u, which is the source of the edge, and the operand v which is the tail of the edge. To ensure consistency among base rules and to account for the costs of chain rules, we impose costs dependent on the selection of variable x_u and variable x_v in the form of a cost matrix $\mathcal{C}_{uv}$. An element in the matrix corresponds to the costs of selecting a specific base rule $r_u \in R_u$ of the result and a specific base rule $r_v \in R_v$ of the operand node. Assume that r_u is $\texttt{nt} \leftarrow op(\dots)$ and r_v is $\dots \leftarrow op(\alpha, \texttt{nt}', \beta)$ where `nt'` is the non-terminal of operand v whose value is obtained from the result of node u. There are three possible cases:

1. If the nonterminal `nt` and `nt`' are identical, the corresponding element in matrix $\mathcal{C}_{uv}$ is zero, since the result of u is compatible with the operand of node v.

2. If the nonterminals `nt` and $\texttt{nt}'$ differ and there exists a rule $r : \texttt{nt}' \leftarrow \texttt{nt}$ in the transitive closure of all chain-rules, the corresponding element in $\mathcal{C}_{uv}$ has the costs of the chain rule, i.e. $w_v \cdot \text{cost}(r)$.

3. Otherwise, the corresponding element in $\mathcal{C}_{uv}$ has infinite costs prohibiting the selection of incompatible base rules for the result u and operand v.

To illustrate this transformation, consider the example shown in Figure 3.3. The figure shows the cost vectors and matrices for the original example introduced in Figure 3.2. The grammar has been normalized as described above by introducing artificial nonterminals `t1`, `t2`, and `t3`. The domain for each node is the set of applicable base rules, e.g.,



R_1	`imm <- IMM`
R_2	`reg <- REG`
R_3	`reg <- imm`
R_4	`reg <- SHL(reg, reg)`
R_5	`reg <- SHL(reg, imm)`
R_6	`t1  <- SHL(reg, imm)`
R_7	`reg <- ADD(reg, reg)`
R_8	`t2  <- ADD(reg, t1)`
R_9	`t3  <- ADD(reg, reg)`
R_{10}	`reg <- LDW(reg)`
R_{11}	`reg <- LDW(t2)`
R_{12}	`reg <- LDW(t3)`

Figure 3.3: *PBQP*-instance derived from the example shown in Figure 3.2. The grammar has been normalized by introducing additional nonterminals.

```
void convert(char *txt,char *ds,int b,int x)
{
  int d;
  char *p=txt;
  do {
    d = x % b;
    x = x / b;
    *p++ = ds[d];
  } while(x > 0);
  *p=0;
  reverse(txt); // reverse string
}

char buf[N],digits[]="0123456789ABCDEF";
convert(buf,digits,10,4711);
```

Figure 3.4: Motivating example showing a number conversion route.

the `SHL` has alternatives `R4`, `R5`, and `R6` while the last rule is a result of normalizing the grammar. The highlighted elements depict a cost-minimal solution of the *PBQP* with costs one. A solution of *PBQP* determines which base rules and chain rules are to be selected. A traversal over the basic blocks using the SSA graph is sufficient to execute the associated semantic rules in order to emit the code.

Chain Rule Placement Chain rules can either be emitted at the basic block corresponding to the source node or right before each use, i.e., at the destination node. In [SS07], a more sophisticated technique is introduced that allows a more efficient placement of chain rules across basic block boundaries. This technique is orthogonal to the generalization to complex patterns presented in this chapter. An optimal placement is computed by the construction of a min-cut problem for the given control flow graph. A solution for this problem can be found in polynomial time. There is a trade-off among performance and code size that is captured accurately in the proposed network flow model.

3.3 Motivation

Even though *PBQP*-based instruction selection is a suitable technique for whole-function instruction selection, the technique as proposed in [EKS03] is limited to tree patterns that restrict the modeling of advanced features found in common embedded architectures. In particular, there is not support for machine instructions with multiple results.

To motivate our generalization, consider the `C` fragment given in Figure 3.4 that shows a number conversion routine. On an architecture, which supports a `divmod` in-

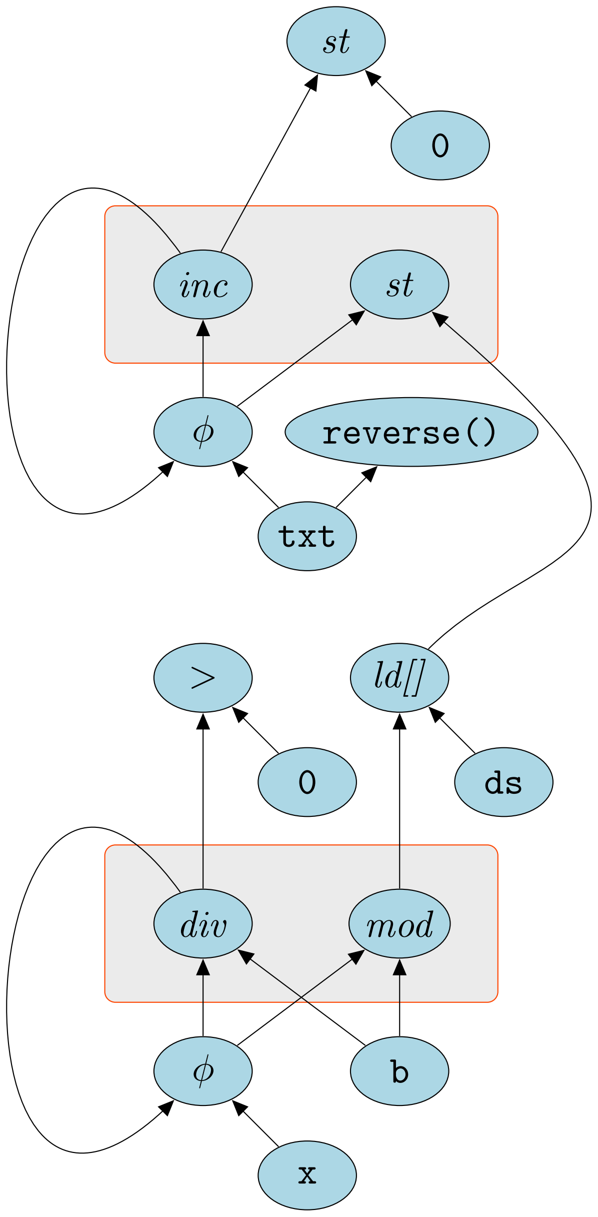


Figure 3.5: SSA-graph of the motivating example introduced in in Figure 3.4.

P1) $\langle \texttt{lo} \leftarrow \mathit{div}(x:\texttt{reg}_1, y:\texttt{reg}_2), \texttt{hi} \leftarrow \mathit{mod}(x,y)\rangle$
    $[2]\,\{\mathit{emit}$ `divmod` $r(\texttt{reg}_1)$`,`$r(\texttt{reg}_2)\}$
P2) $\langle \leftarrow \mathit{st*}(x:\texttt{reg}_1, \texttt{reg}_2), \texttt{reg} \leftarrow \mathit{inc}(x)\rangle$
    $[3]\,\{\mathit{emit}$ `addi tmpreg,`$r(\texttt{reg}_1)$`,0;`
        `movsw` $r(\texttt{reg}_2)$`,(tmpreg)++)`$\}$
P3) $\leftarrow \mathit{st*}(\texttt{reg}_1, \texttt{reg}_2)$
    $[2]\,\{\mathit{emit}$ `sw` $r(\texttt{reg}_2)$`,(`$r(\texttt{reg}_1)$`))`$\}$
P4) $\texttt{reg} \leftarrow \mathit{inc}(\texttt{reg})$
    $[2]\,\{\mathit{emit}$ `addi` $r(\texttt{tmpreg})$`,`$r(\texttt{reg})$`,4`$\}$
C1) $\texttt{reg} \leftarrow \texttt{lo}$
    $[2]\,\{\mathit{emit}$ `mflo` $r(\texttt{reg})\}$
C2) $\texttt{reg} \leftarrow \texttt{hi}$
    $[2]\,\{\mathit{emit}$ `mfhi` $r(\texttt{reg})\}$

Figure 3.6: Fragment of rules with complex patterns for div-mod and postincrement store instructions.

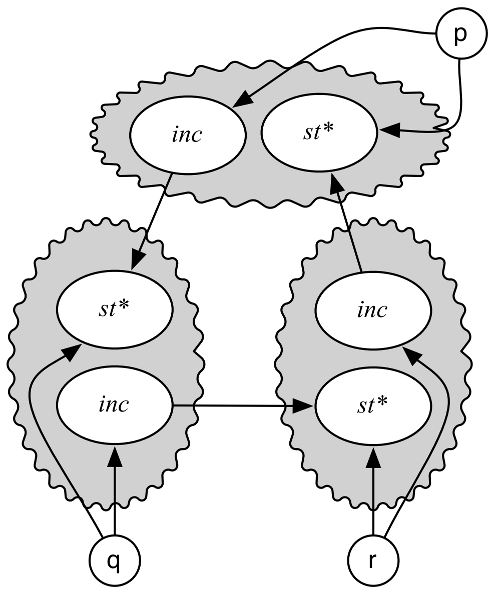
struction and post-increment addressing modes, the instruction selector could exploit these features for reducing code size and improving the execution speed of the program. However, neither the pattern for `divmod` nor the pattern for the post-increment store can be expressed in terms of tree shaped productions as depicted in the SSA graph in Figure 3.5. Both patterns have multiple in-coming and out-going edges and cover multiple non-adjacent nodes in the SSA graph at the same time.

In this work we introduce a new approach that is able to cope with complex patterns as shown in our motivating example [EBS+08]. An excerpt of a cost augmented graph grammar describing the `divmod` instruction and the post-increment addressing mode is listed in Figure 3.6. In the graph grammar, each pattern is a *tuple* of productions constituting a *DAG* shaped pattern, costs, and the semantic actions. For example the `divmod` pattern `P1` shown in Figure 3.6 can only be applied if the arguments for the *div* and the *mod* node are identical. This is expressed by naming the arguments of the `div` node with x and y. These labels are re-used in the rule for `mod` expressing that the same arguments have to match. The associated cost function for a pattern is shown in brackets. The underlying architecture of the example assumes a *MIPS R2000* like division instruction, i.e., both the quotient and the remainder are stored in dedicated registers. The rules `C1` and `C2` emit the move instructions (`mflo` and `mfhi` respectively) to retrieve the values of the `divmod` instruction.

Tree patterns do not destroy the topological order for emitting the code, however, complex patterns can: a cyclic data dependency occurs if a set of operations in the SSA graph is matched by a pattern for which there exists a path in the SSA graph that exits and re-enters the complex pattern within the basic block. This cycle would imply that operations are executed on the target hardware before the values of the operands are available. Hence, the matcher must prohibit those cycles in the minimum cost cover by finding a topological order among the patterns. The example in Figure 3.7 illustrates the

```
*p:=r+4;
*q:=p+4;
*r:=q+4;
```

(a) Input Block



(b) SSA Graph

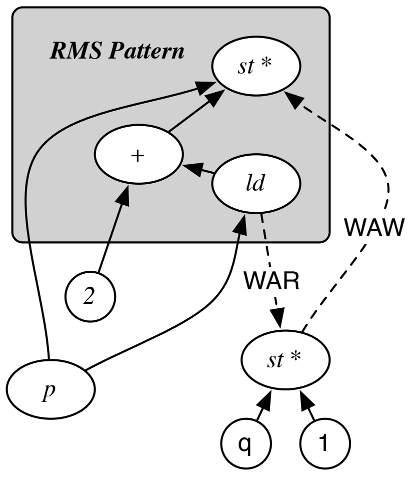
Figure 3.7: Example: topology constraints.

problem of finding a cover that does not cause any cyclic data dependencies. The code fragment contains three feasible instances of a post-increment store pattern (cf. P2, P3, P4 in Figure 3.6). Assuming that we know that p, q, and r point to mutually distinct memory locations, there are no further dependencies apart from the edges shown in the SSA graph. The example obviously gives rise to a topological order of the semantic rules as long as we do not select all three instances of the post-increment store pattern concurrently.

Modeling memory accesses in the instruction selection of a compiler is a challenging problem. SSA graphs do not reflect memory dependencies. However, they do have memory operations that impose data dependencies among memory operations including loads and stores. For example consider the example shown in Figure 3.8 that depicts a typical read-modify-write (*RMW*) pattern such as "`add r/m32, imm32`" in the IA32/AMD64 architecture. A corresponding production rule might be formulated as `stmt` $\leftarrow$ $st*(x : \texttt{reg}_1, +(ld(x), \texttt{imm}))$. If we have to assume that `p` and `q` might address the same memory location, we have to account for the antidependency among statements `(1)` and `(2)` and the output dependency among statements `(2)` and `(4)`; depicted in Figure 3.8*(b)* with dotted lines. There is obviously no topological order among the highlighted part forming the *RMW* pattern and the store corresponding to instruction `(2)`, i.e., we cannot apply the pattern even if it is the cheapest graph cover. To ensure the existence of a topological order among the chosen productions, the SSA graph is augmented with additional edges representing potential data dependencies.

```
(1) x₁:=*p;
(2) *q:=1;
(3) x₂:=x₁+2;
(4) *p:=x₂;
```

(a) Input Block



(b) SSA Graph

Figure 3.8: Example: memory dependencies.

3.4 Instruction Selection using Complex Patterns

The extension of the instruction selector [EKS03] is mainly concerned about prohibiting cycles in the selection of patterns and considering memory dependencies for the instruction selection. We can restrict the algorithm to *normalized* grammars that consist of the following types of productions: (1) *chain rules* of the form $\mathtt{nt}_0 \leftarrow \mathtt{nt}_1$, and (2) tuples of *base rules* of the form $\mathtt{nt}_0 \leftarrow op(\mathtt{nt}_1, \ldots, \mathtt{nt}_{k_p})$

Algorithm 4 Generalized *PBQP* instruction selection

```
1: identify instances of complex patterns within basic blocks
2: transform the problem to an instance of PBQP
3: obtain a solution for the PBQP instance using a generic solver
4: for all basic blocks $b$ do
5:     compute a topological order for the subgraph $S_b \subseteq B$ that is induced by basic
       block $b$
6:     apply the semantic rules associated with the chosen productions in the order
       computed in step (5).
```

The main scheme of our algorithm for matching complex *DAG* patterns is shown in Algorithm 4. Steps (1), (2), and (5) differ from the approach described in [EKS03]. First, we identify concrete tuples of nodes in the SSA graph that can be used to form patterns specified in the input grammar. Next, we transform the problem to an instance of *PBQP* that is processed using a generic solver library.

The problem formulation ensures the existence of a topological order among the chosen productions and allows for a straight-forward back-transformation that maps a solution vector of *PBQP* to a complete graph cover. The partial order among the particular nodes is defined by the edges in the SSA graph and additional data dependencies among load and store instructions. We can thus use a reversed post-order traversal to apply the semantic actions associated with the chosen productions in a proper order on the subgraphs induced by individual basic blocks. This process rewrites those subgraphs in a bottom-up fashion into target specific *DAG*s that are directly passed to a prepass list scheduler.

3.4.1 Identifying Patterns in SSA Graphs

As described in Section 3.3, generalized productions cover a tuple instead of individual nodes in the SSA graph. The matcher has to choose among them based on associated cost functions. Therefore, we enumerate *instances* of complex patterns in step `(1)` of Algorithm 4, i.e., concrete tuples of nodes that match the terminal symbols specified in a particular production. More formally, an instance of a complex production p is a $|p|$-tuple

$$(v_1, \ldots, v_{|p|}) \in V^{|p|} \qquad v_i \neq v_j \quad \forall\ 1 \leq i < j \leq |p|$$

of nodes in the SSA graph such that $o_i = \mathrm{op}(v_i)\ \forall\ 1 \leq i \leq |p|$, i.e., each node matches the terminal symbol of the corresponding base rule. An instance l is called *viable* if $costs_p(l) < \infty$. The set of all viable instances for a production p and an SSA graph G is denoted by $I_p(G)$.

A dependency between two instances of complex patterns p and q within a basic block b is denoted by $p \prec_b q$. Note that this relation might have cycles as shown in examples in Section 3.3. The relation defines the partial order in which the semantic actions have to be applied and can be naturally derived from the edges in the SSA graph augmented with potential memory dependencies.

3.4.2 Problem Transformation

This section describes the transformation of the generalized instruction selection problem to an instance of *PBQP*. We define the set of decision variables $X = \{x_1, \ldots, x_n\}$ along with their finite domains $\{\mathbb{D}_1, \ldots, \mathbb{D}_n\}$. A local cost vector $c_i = (c_1, \ldots, c_{|\mathbb{D}_i|})$ specifies the costs of assigning variable x_i to a particular element in its domain. For related variables x_i and x_j, we establish matrix costs C_{ij} that valuate a particular assignment of x_i and x_j.

Decision Variables Decision variables are created both for nodes in the SSA graph and for each of the enumerated instances of complex patterns. The whole set of variables $X = X_1 \uplus X_2$ is defined as follows.

For each SSA node $u \in V$, we introduce a variable $x_u \in X_1$. The domain of x_u is defined by the set of applicable base rules arising from two different sources:

1. Simple productions consisting of a single base rule; these are handled just like in previous approaches

2. Base rules arising from complex productions. These rules are treated as a set of simple base rules, e.g., the production

$$\langle \texttt{stmt} \leftarrow \mathit{st*}(x : \texttt{reg}_1, \texttt{reg}_2), \texttt{reg} \leftarrow \mathit{inc}(x)\rangle$$

is decomposed into $\texttt{stmt}\langle \leftarrow \mathit{st*}(x : \texttt{reg}_1, \texttt{reg}_2)\rangle$ and $\langle \texttt{reg} \leftarrow \mathit{inc}(x)\rangle$. All base rules with the same signature obtained from the decomposition of complex productions contribute only to a single element to the domain for x_u. Base rules derived from productions p for which u does not appear in any of the instances in $I_p(G)$ can be safely omitted.

While the former group represents the set of patterns that can be used to obtain a cover for node u, the second class of base rules can be seen as a proxy for the whole set of instances of (possibly different) complex productions in which u arises. The costs for elements in x_u are 0 for the proxy states corresponding to the selection of a complex instance, otherwise they reflect the real costs of the corresponding simple rule.

For each instance $l \in I_p(G)$ of a complex production p, we create a distinct decision variable $x_l \in X_2$ that encodes whether the particular instance is chosen or not, i.e., the domain consists of the elements *on* and *off*. As we will describe later, it is sometimes necessary to further refine the state *on* in order to guarantee the existence of a topological order among the chosen nodes. The local costs for x_l are set to be 0 if x_l is *off* and $\mathit{costs}_p(l)$ otherwise.

Constraints Constraints can be formulated in *PBQP* in terms of quadratic cost functions represented by cost matrices that "glue" the particular variables together. Among the two sets of variables X_1 and X_2 we create three different types of related costs, i.e., $X_1 \to X_1$, $X_1 \to X_2$, and $X_2 \to X_2$.

The first type of cost matrices is established among adjacent variables $u, v \in X_1$. Therefore, we add matrix costs C_{uv} as outlined in Section 3.3 that enforce compatibility between two rules and account for the cost of chain rules. If no derivation exists, the costs are set to ∞ with the effect that the transition is prohibited. Among identical non-terminals, costs are 0. More formally, let $e = (u, v)$ be an edge in the SSA graph and let $\texttt{nt}_0^u \leftarrow o_u(\texttt{nt}_1^u, \ldots, \texttt{nt}_n^u)$ and $\texttt{nt}_0^v \leftarrow o_v(\texttt{nt}_1^v, \ldots, \texttt{nt}_m^v)$ denote the base rules corresponding to variables u and v. We define

$$C_{uv}^{X_1 \to X_1} = w_e \operatorname{mincosts}(\texttt{nt}_0^u, \texttt{nt}^v_{\mathrm{opnum}(e)})$$

while $\operatorname{mincosts}(\texttt{nt}_i, \texttt{nt}_j)$ denotes the minimal costs for all chain rule derivations from $\texttt{nt}_i$ to $\texttt{nt}_j$. The function mincosts can be easily derived by computing the transitive closure for all chain rules in the grammar, e.g., using the Floyd-Warshall algorithm [Flo62].

For each variable $x_l \in X_2$ corresponding to an instance l, we need to create constraints ensuring that the corresponding proxy state is selected on all variables $x_u \in X_1$ that represent the SSA nodes u forming l. Therefore, we create matrix costs $C_{ul}^{X_1 \to X_2}$ such that the costs are zero if x_l is set to *off* or x_u is set to a base rule that is not associated to the instance l. Otherwise, costs are set to ∞. Thus, when one of the instances correlated to a particular node u in the SSA graph is selected, the only remaining element in the domain of u with costs less than ∞ is the associated proxy state corresponding to the particular base rule fragment.

So far, the formulation allows the trivial solution where all of the related variables encoding the selection of a complex pattern are set to *off* (accounting for 0 costs) even though the artificial proxy state for x_u has been selected. We overcome this problem by adding a large integer value M to the costs for all proxy states. In exchange, the costs $c(v)$ for variables $x_v \in X_2$ are set to $(c(v) - |l|M)$ while $|l|$ denotes the number of nodes for instance l. Thus, the penalties for the proxy states are effectively eliminated unless an invalid solution is selected.

The last type of matrix costs is established among variables $x_u \in X_2$ and $x_v \in X_2$ where $x_u \neq x_v$. These matrices ensure that

- two instances l_u and l_v covering the same nodes in the SSA graph cannot be selected at the same time, i.e. assigned to the state *on*
- the set of selected instances does not induce cyclic data dependencies

The basic idea is to reduce the problem to the task of finding an induced acyclic subgraph within the dependence graph $D_b(G)$ that can be defined as follows.

- there is a node $w \in D_b(G)$ for every instance $l_w \in I_p(G)$ consisting of SSA nodes in block b
- there is a directed arc $(w_1, w_2) \in D_b(G)$ iff $l_{w_1} \prec_b l_{w_2}$

Any subset of instances that is selected at the same time induces a subgraph $G' \subseteq D_b(G)$ that has to be acyclic to allow for a valid emit order. We exploit the property that every acyclic directed subgraph of $D_b(G)$ gives rise to a not necessarily unique topological order. Note that it is sufficient to reduce the problem to the strongly connected components of $D_b(G)$. We can integrate this idea into the problem formulation obtained so far as follows:

1. for every strongly connected component S_i of $D_b(G)$, we compute an upper bound $\max(S_i)$ on the number of instances represented by nodes in S_i that can possibly be selected at the same time without multi-coverage of SSA nodes. In general, this subtask can be reduced to the maximum independent set problem which is known to be *NP* complete. However, it is sufficient to solve the problem heuristically since the bounds are only used to decrease the problem size of the *PBQP* instance.

2. for all decision variables representing complex instances within a non-trivial strongly connected component S_i, i.e., its cardinality is greater than one, we replace the state *on* in their domain with the elements $1, \ldots, |\max(S_i)|$ representing their index in a topological order. The costs of those elements corresponds to the costs of the former *on* state.

3. we establish matrix costs C_{uv} among variables $x_u, x_v \in X_2$ for instances u and v respectively as follows

$$C_{uv}^{X_2 \to X_2} = \begin{cases} \infty, & \text{if } x_u \neq \mathit{off} \ \wedge \ x_v \neq \mathit{off} \wedge \\ & \quad (x_u = x_v \ \vee \ u \cap v \neq \emptyset \vee \\ & \quad ((u,v) \in S_i \subseteq D_b \wedge x_u > x_v)), \\ & \\ 0, & \text{otherwise.} \end{cases}$$

   If one or both instances are set to *off*, the element of $C_{uv}^{X_2 \to X_2}$ is zero. Otherwise, if both u and v are within the same strongly connected component in $D_b(G)$ and $u \prec_b v$, we want to make sure that the index assigned to u is less than the index assigned to v. Similarly, costs are set to ∞ if $x_u = x_v$ or $u \cap v \neq \emptyset$ in order to ensure that no two instances can be assigned to the same index and instances covering a common node cannot be selected at the same time. These cost matrices constrain the solution space such that no cyclic data dependencies can be constructed in any valid solution.

The decision variables and matrices described above constitute a complete *PBQP* formulation for t'he generalized instruction selection problem.

Example One way to think of an instance of *PBQP* is as a directed labeled graph. Nodes represent decision variables that are annotated with the local cost vectors and edges among nodes represent non-zero cost matrices. For each node, the solver selects a unique element from its domain such that the corresponding overall costs are minimized.

Using this notation, we illustrate the *PBQP* formulation presented above in Figure 3.9 using the example SSA graph shown in Figure 3.7 and the rule fragments given in Figure 3.6. Base rules and cost matrices for the address variables p, q, and r are omitted for simplicity. Decision variables X_1 for SSA nodes are denoted in circles while those for complex instances are represented by rounded squares. We use k as a placeholder for the term $3 - 2M$[1] representing the costs for production `P3` minus the penalty that has been added on adjacent variables in X_1. The example shows all three types of matrix costs that can arise in the problem transformation. Note, that the corresponding nodes for all three instances $(2,1)$, $(3,5)$, and $(6,4)$ of production `P3` are within one and the same strongly connected component in the dependence graph $D_b(G)$.

[1] M denotes a sufficiently large integer constant.

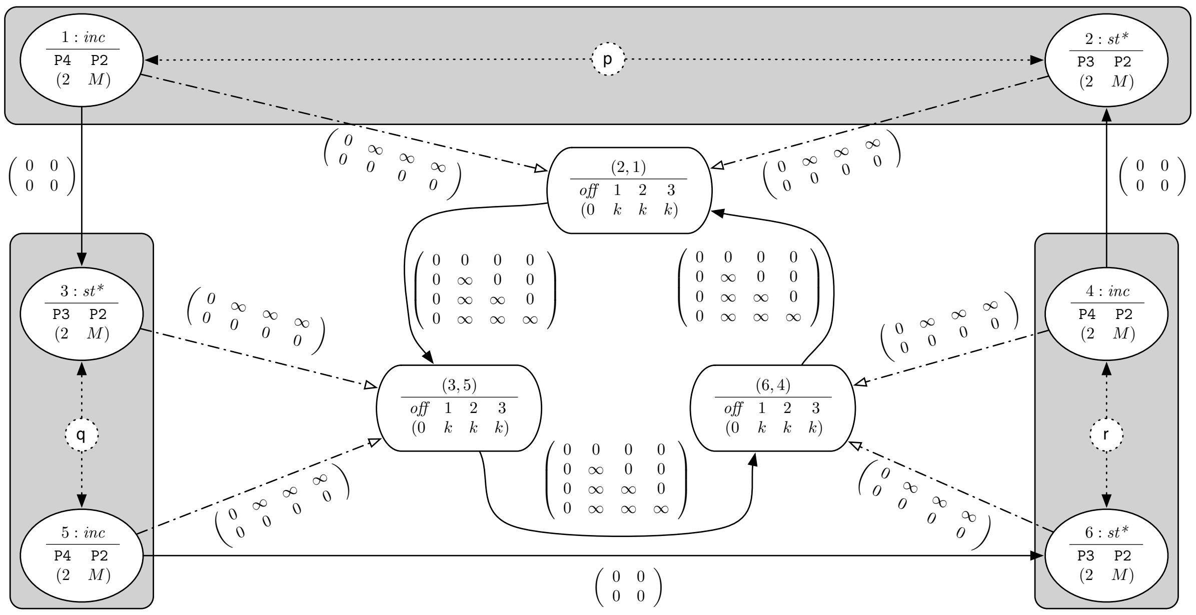


Figure 3.9: *PBQP* graph for the example shown in Figure 3.7. We use k as a shorthand for the term $3 - 2M$.

Correctness In order to show the correctness of our formulation, we have to show that (a) the solution of the *PBQP* corresponds to a complete pattern matching of the SSA graph and (b) there is a topological order in which the semantic actions associated with the selected patterns can be executed.

Assuming M is a value larger than the costs of any feasible matching, it is easy to show that there is an isomorphism among a solution of the *PBQP* with objective value less than M and a complete cover of the graphs with rules from the input grammar. In particular, an optimal solution to the *PBQP* corresponds to a min-cost matching for the input graph. The selected rules are determined by the assignments of the decision vectors in the *PBQP* and vice versa.

Tree patterns cannot introduce circular dependencies except for indirect memory dependencies among inner nodes such as demonstrated in the example in Figure 3.8. These cases can be eliminated beforehand and cannot induce cyclic data dependencies among chosen pattern fragments. However, as shown before, *DAG* shaped patterns can. The proposed modeling effectively enforces the existence of a concrete topological order among the selected patterns. No cycles are possible in any valid solution unless at least one ∞ term is chosen.

For general grammars and applied to irreducible graphs, the heuristic algorithm proposed in Section 2.2.1 cannot be guaranteed to deliver a solution that satisfies all constraints modeled in terms of ∞ costs. This would be a *NP*-complete problem. One way to work around this limitation is to include a small set of rules that cover each node individually and that can be used as a fallback rule in situations where no feasible solution has been obtained. This corresponds to macro substitution techniques and ensures

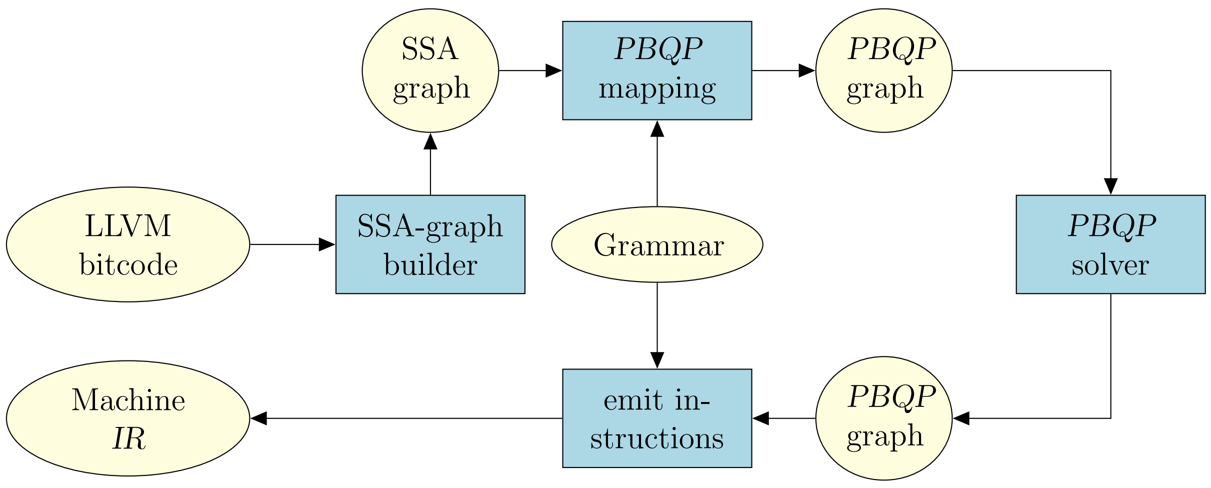


Figure 3.10: Overview of the *PBQP*-based instruction selector.

a correct but possibly suboptimal matching. In practice, this is no severe limitation as grammars are usually written by defining a simple but complete set of rules covering each node individually and adding more complex rules later on. These limitations do not apply to exact techniques such as the *B&B* algorithm introduced in Section 2.2.2. It is also straight-forward to extend the heuristic algorithm with a backtracking scheme on RN reductions, which would clearly also be exponential in the worst case.

3.5 Implementation Details

We have implemented the global instruction selector described in Section 3.4 within *LLVM*; c.f. Section 1.3. Benchmarks are converted using the *gcc* based frontend (`llvm-gcc`) into *LLVM* intermediate code that is further processed using the standard set of machine-independent optimizations and fed into the code generation backend.

A rough overview of the implementation of the improved instruction selector is given in Figure 3.10. We create SSA graphs from the LLVM intermediate representation. For each backend, a rule grammar with normalized rules and precomputed chain closures is generated at compiler compile time using a dedicated `tablegen` backend; see Section 1.3. The SSA graph and the grammar are used to map the problem to a generic *PBQP* instance as shown in Section 3.4.2. A generic solver library is used to solve these instances. The selected graph cover determines a unique baserule for each node in the *SSA* graph that is used to execute the associated semantic actions. These actions rewrite the intermediate representation bottom-up such that, upon completion, each node corresponds to a native machine instruction.

Both the existing *LLVM* instruction selector and our *PBQP* instruction selector are implemented as graph transformations that rewrite a selection graph representing LLVM intermediate code into target dependent machine instructions. Prior to code generation, a legalize phase that is common to both instruction selectors lowers certain *DAG* nodes to target dependent constructs, e.g., floating point instructions are converted into library calls and 64bit operations are lowered into 32bit arithmetic.

A subsequent prepass scheduler converts the result graphs into a sequence of machine instructions while accounting for resource constraints of the target processor. This approach is superior to the workflow of most existing compilers that usually have to rebuild a data dependence graph from a fixed topological order during scheduling, since the same data structure along with precious annotations from alias analysis can be passed from one phase to another without loss of information.

The existing *LLVM* instruction selector implements a bottom up pattern matching approach on the scope of basic blocks. Most architecture dependent parts are generated from a target description at compiler compile time. While the algorithm efficiently handles simple patterns, custom *C++* code has to be used in order to match instructions that cannot be expressed using the existing infrastructure. While this approach makes it difficult to retarget the code generator and to implement application specific instruction set extensions, it is very efficient in terms of compile time and is applicable in the realm of just in time compilers.

We consider the existing ARMv5 backend of LLVM 2.1 and implement a corresponding grammar for our new instruction selector. Most of the complex addressing modes available on ARM cannot be handled by the bottom up approach implemented in LLVM. Therefore, a preprocessing algorithm tries to identify pre- and post-increment memory accesses and rewrites them into target dependent *DAG* nodes. Additionally, the instruction selector is bypassed for certain nodes such as `cmov` instructions, multiplies, or the complex addressing modes available both for arithmetic/logic and memory access instructions. Those cases are handled by handwritten, target dependent *C++* procedures aside from the generic algorithm.

In contrast to the existing LLVM instruction selector, our algorithm can be fully retargeted using a grammar with the extensions presented in Section 3.4 and does not necessitate the ad-hoc techniques implemented for LLVM. The grammar consists of a total number of 555 normalized rules; 46 rules are complex rules consisting of multiple base rules that could not be handled with previous approaches. A base set of 80 rules has been automatically derived from the existing machine description. About 40 rules are used for the various ARM addressing modes. Dedicated nonterminals are used to efficiently describe repeating pattern fragments such as the arithmetic operations with flexible addressing mode 1 that implicitly shift/rotate one of the source registers by another register or immediate value.

Composite rules are necessary for the available pre- and post-increment addressing modes on ARMv5 which cannot be expressed as simple tree patterns (see Table 3.1). An example of a post-increment store pattern has already been shown in Figure 3.6. In our prototype implementation, the cost functions account for move instructions that inevitably have to be inserted by the register allocator if the base register is used (maybe indirectly) by another SSA node that has to be scheduled *after* the load/store instruction that is part of the pattern. In those cases, the old value has to be saved into a temporary register, which effectively increases the costs of our patterns. We compute those cost

pre-increment	LDR|STR {B} <Rd>,	[<Rn>, #± <imm12>]! [<Rn>, ± <Rm>]! [<Rn>, ± <Rm> <shift> #<imm>]!
	LDR|STR {H|SH|SB} <Rd>,	[<Rn>, #± <imm8>]! [<Rn>, ± <Rm>]!
post-increment	LDR|STR {B} <Rd>,	[<Rn>], #± <imm12> [<Rn>], ± <Rm> [<Rn>], ± <Rm> <shift> #<imm>
	LDR|STR {H|SH|SB} <Rd>,	[<Rn>], #± <imm8> [<Rn>], ± <Rm>

Table 3.1: ARMv5 pre-/postincrement addressing modes.

functions efficiently using precomputed successor sets.

Since SSA form is maintained in *LLVM* until register allocation, machine instructions cannot both read and define the same operand. Therefore, all instructions with autoincrement addressing have an additional (virtual) destination operand $<R_t>$ along with a constraint for the register allocator of the form $<R_t> = <Rn>$. While our approach would be capable to capture some complex ARM instructions such as `LDRD|STRD` (load/-store double) and `LDM|STM` (load/store multiple), those pattern require constraints of the form $<R_i> = <R_j>$+1, which currently cannot be handled by the register allocator. Those modifications are beyond the scope of this work.

In addition to pre- and post-increment loads and stores, we implement complex patterns for swap instructions (`swp`, `swpb`), and the signify versions of various instructions such as `adds` and `movs` that implicitly set the `Z` flag in the processor status register (`CPSR`). Those instructions can be effectively used to replace an explicit `cmp` instruction in counting loops. However, since the induction variable in most counting loops is increased, we use a simple prepare-pass that checks for loop carried dependencies and reverts them, thereby frequently allowing for the application of typical `subs` patterns.

Even though there is neither a hardware `div` nor a `mod` instruction on ARMv5, we can fold the necessary calls into the runtime library (`libgcc`) into a combined function that delivers both the quotient and the remainder at the same time (`__aeabi_[u]idivmod`).

3.6 Experimental Results

We apply our prototype implementation to three different suites of benchmarks, i.e., typical DSP kernels mostly taken from the fixed point branch of the DSPstone suite [uVSM94], medium sized applications from the MiBench suite [lm], and general purpose programs represented by the SPECINT 2000 benchmark suite [SPC].

All programs have been cross compiled using one core of a Xeon DP 5160 3GHz with 24GB of main memory. The DSP kernels and the MiBench suite are executed with the

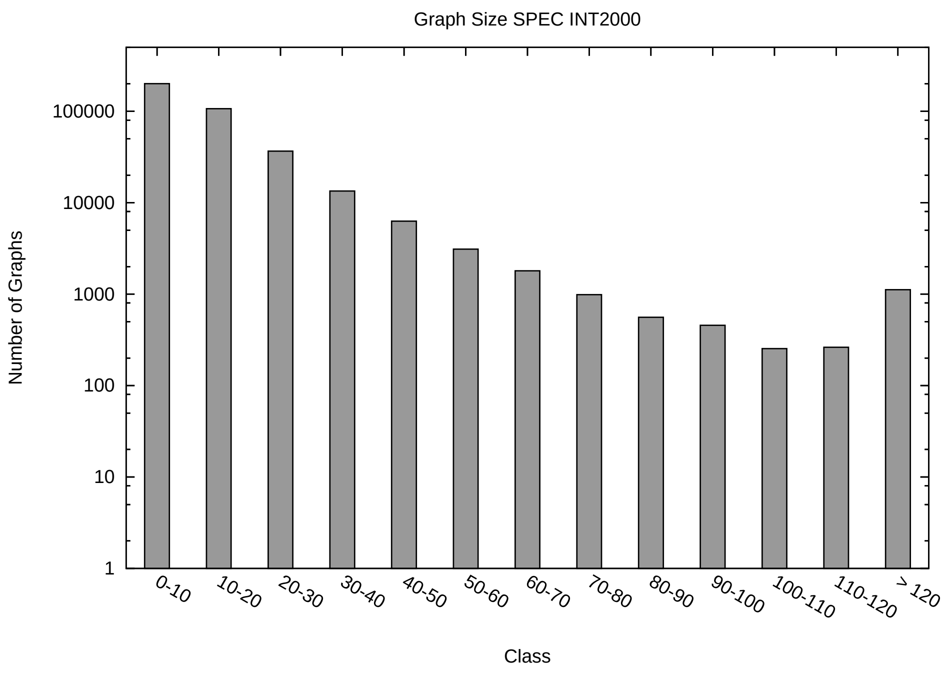


Figure 3.11: Number of instances per graph size.

free, cycle accurate instruction set simulator included in the `gdb`[2] project. This approach is not feasible for large benchmarks such as those from the SPEC suite. Therefore, we execute them on real hardware. The target board running a Linux 2.4.22 kernel is equipped with an Intel XScale IOP80321 (600MHz) and 512MB of memory. For floating point operations, we use the IEEE754 implementation that ships with *gcc* since there is no hardware floating point unit available on our target. Both the original backend and our *PBQP* based implementation have been verified against a *gcc* 4.0.2 cross compiler which has also been used to build `binutils` and `glibc`. Execution times have been gathered using the unix `time` utility considering the best out of 10 runs on the unloaded machine.

Benchmarks compiled for the instruction set simulator have been linked with `newlib` – a `C` library implementation for embedded systems. We omit those benchmarks from the MiBench suite that use operating system features such as sockets and pipes that are not implemented in `newlib`. Likewise, we do not provide results for the most simple benchmarks in the DSPstone suite such as `complex_update` or `startup` since all considered compilers produce the same few instructions.

For the DSP kernels, we extend the simulator with a simple stopwatch facility that is triggered by dedicated reserved opcodes and allows us to obtain cycle accurate measures for inner loops without startup and I/O overhead.

The most difficult *PBQP* instances are generated for the SPEC suite. We present results for all benchmarks except `252.eon` which is written in *C++* and therefore cannot

[2]`http://sourceware.org/gdb/`

benchmark	*gcc*	*LLVM*	*PBQP*	$\frac{LLVM}{PBQP}$
dsp-fft	768393	868252	741807	1.17
dsp-fir	333	167	150	1.11
dsp-fir2dim	2430	1149	1149	1.00
dsp-lms	812	598	553	1.08
dsp-matrix	16127	16191	13893	1.17
misc-cmac	1691443	1608654	1565287	1.03
misc-convert	2117	1924	1228	1.57
misc-dct8x8	196377	116682	113594	1.03
misc-qsort	22187557	24541181	21219621	1.16
misc-serpent	3463333	2062079	2067729	1.00
misc-vdot	20707	20717	18716	1.11

Table 3.2: Execution time [cycles] for inner loops of various DSP benchmarks (mostly taken from the DSPstone suite).

be compiled with our prototype implementation. Figure 3.12 shows the number of SSA graphs over the whole benchmark set compared to the number of nodes (partitioned into classes of size ten). Note the logarithmic scale of the y-axis. The vast majority of graphs (99.5%) has less than 100 nodes. The largest graph over the whole benchmark set can be found in 176.gcc and consists of 1613 nodes and 1026 edges.

In order to solve the *PBQP* instances, we compare the heuristic approach described in [SE02a, Eck03] with an optimal algorithm based on branch & bound [HS06]. Furthermore, the solver time for the *PBQP* instances is compared to a linearization of the problems that are solved with ILOG CPLEX 10. The *PBQP* is translated to a linear program with 0-1 variables.

Computational Results Cycle accurate results for the DSP kernels and the MiBench suite are shown in Table 3.2 and 3.3 respectively. We compare the results obtained with *gcc*, the original *LLVM* 2.1 backend, and our new instruction selector based on *PBQP*. Speedups for the *DSP* kernels are up to 57% (`misc-convert`, see Figure 3.4) with an average of 13%. The largest gains for the MiBench suite could be achieved for `automotive-susan` with a speedup of 10%. Only a single benchmark (`consumer-lame`) shows a slowdown by 5% that is caused by spill code due to an inferior register allocation. All results have been obtained with the heuristic *PBQP* solver.

Next, we consider the benchmarks from the SPECINT 2000 suite. Detailed results are shown in tables 3.4 and 3.5 respectively. All of the benchmarks could be compiled with the heuristic *PBQP* solver within half a minute, most of them took only a couple of seconds. The compile time slowdown compared to *LLVM* is about a factor of 2 and is mainly caused by the overhead for building the SSA graphs on top of the standard selection graph data structures and the immature prototype implementation of our matcher. Column *mem.* denotes the maximum amount of memory required to represent

benchmark	*gcc*	*LLVM*	*PBQP*	$\frac{LLVM}{PBQP}$
basicmath	6980.14	6992.17	6989.30	1.00
bitcount	93.06	109.35	106.67	1.03
susan	679.63	763.69	696.55	1.10
jpeg	15.53	16.36	15.18	1.08
lame	2447.82	2470.31	2592.58	0.95
dijkstra	482.79	323.46	323.43	1.00
stringsearch	9.96	10.28	9.94	1.03
blowfish	1.42	1.41	1.41	1.00
rijndael	897.08	540.06	535.59	1.01
sha	19.63	19.82	18.73	1.06
crc32	833.12	753.29	753.29	1.00
fft	1552.64	1558.51	1558.22	1.00
adpcm	656.61	854.71	801.46	1.07
gsm	3054.84	3103.48	3077.01	1.01

Table 3.3: Execution time [megacycles] for the MiBench suite.

the *PBQP* instances (max. 642 KB). None of the benchmarks compiled with the *PBQP* based instruction selector is slower than the *LLVM* compiled version while speedups are up to 10%. Over the whole benchmark suite, the average speedup is about 5%.

For the simple approach where each rule is either a base rule or a chain rule, the size of the *PBQP* problem for a particular grammar is at most linear in the size of the graph. This is no longer the case for our generalization since we enumerate *combinations* of nodes. In general, the number of instances for a k-ary pattern in a SSA graph with n nodes is bound by $O(\binom{n}{k})$ which is in $O(n^k)$. Thus, for worst case examples, the exhaustive enumeration for composite patterns quickly renders the problem intractable.

However, as our experiments show, this does not appear to be a burden in practice since there is usually only a reasonably small number of viable alternatives for complex patterns within a basic block. Figure 3.12 shows the average problem size in bytes per graph size that is necessary to represent the *PBQP* problem. The graph shows an almost linear behavior in the size of the input graphs.

The number of decision variables for *PBQP* is determined by the size of the input graph and the number of instances that could be identified. Over the whole benchmark set, only 1.1% of all variables were used to select among compound rule alternatives. Likewise, about 94.9% of nonzero matrices were established among nodes representing simple operations, 2.8% had to be used to enforce consistency among regular nodes and pattern variables, and about 2.2% were required to ensure the existence of a topological order among them. Over the whole benchmark set, about 18.618 opportunities for pre- and post-increment instructions could be identified; a maximum of 92 within a single graph.

If there are no RN nodes in the reduction phase of the heuristic solver, the solution is

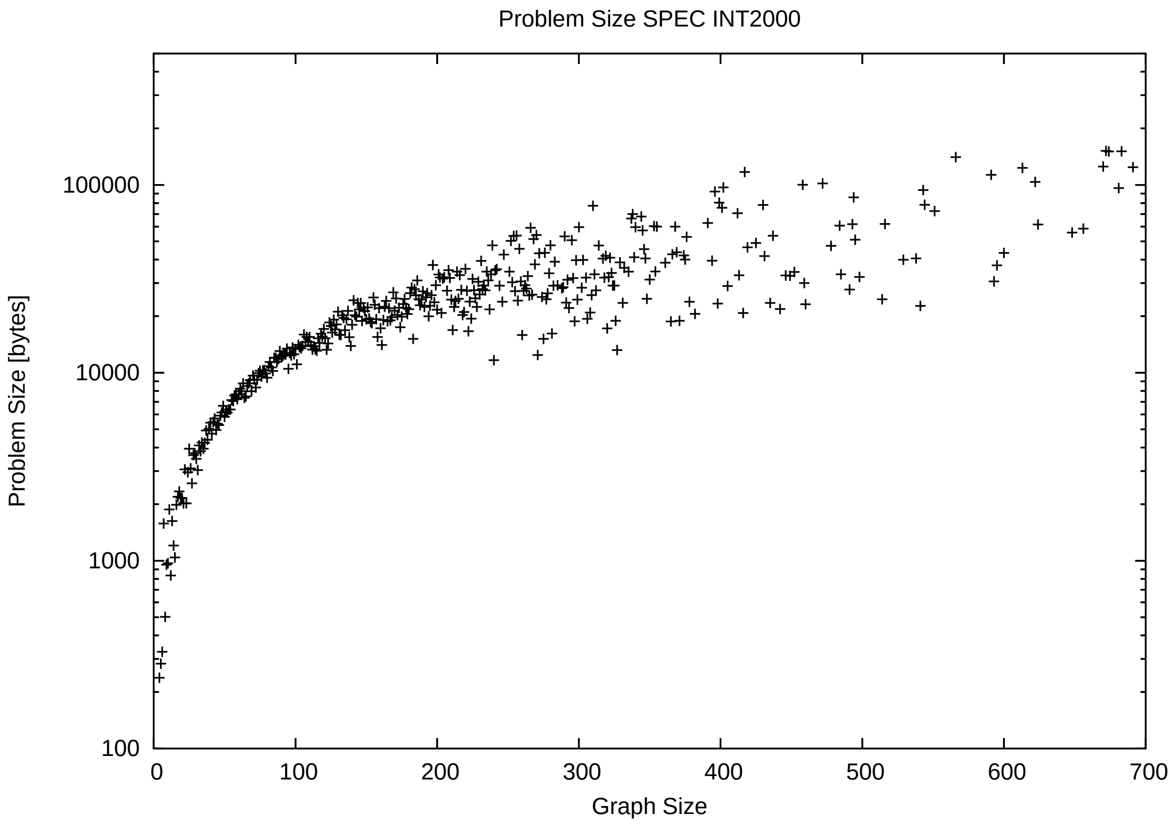


Figure 3.12: *PBQP* problem size.

benchmark	num. graphs	execution time [sec]			
		gcc	*LLVM*	*PBQP*	$\frac{LLVM}{PBQP}$
164.gzip	1204	385.95	427.34	392.47	1.09
175.vpr	5630	219.34	262.39	242.49	1.08
176.gcc	75500	40.60	42.18	41.65	1.01
181.mcf	416	328.21	326.20	324.79	1.00
186.crafty	7341	389.95	402.81	376.72	1.07
197.parser	5997	74.46	77.12	76.54	1.01
253.perlbmk	32748	92.03	130.43	120.90	1.08
254.gap	28886	62.43	54.31	49.22	1.10
255.vortex	18270	174.74	140.03	133.65	1.05
256.bzip2	1005	314.41	288.98	288.08	1.00
300.twolf	9104	171.76	179.78	173.74	1.03

Table 3.4: Execution time for the SPECINT 2000 suite for various instruction selection algorithms.

optimal. If RN nodes occur in the reduction phase, we are interested in the quality of the obtained solution. Note that almost all of the input graphs (177.870) could be solved without RN reductions and, hence, are optimally solved by the heuristic solver. For the remaining graphs (7968), we compare the solution with an optimal solution obtained by the branch & bound solver.

Results are given in the column *"solver statistics"* in Table 3.5. The first column (opt_1) contains the number of instances that could be solved directly to provable optimality by the heuristic solver. The remaining cases have been verified by the B&B solver. Most of them could not be improved further (opt_2) while only a small number (shown in column *sub.*) was suboptimal. This shows that in practice the solution of the heuristic *PBQP* solver coincides with the optimal solution or is very close to the optimal solution.

To show the effectiveness of the *PBQP* approach for instruction selection, we compare the branch & bound solver with a state of the art integer linear programming ILOG(tm) CPLEX 10 solver. Therefore, We obtain a linear program for *PBQP* by applying standard techniques to linearize the *PBQP* objective function; see Chapter 2. For the SPEC benchmark the total solver time for all *PBQP* instances for instruction selection was 196 seconds whereas the ILP solver required more than 163 hours. The *PBQP* branch & bound solver solved all instances optimally whereas CPLEX could not find an optimal solution for 15 instances within a 10 hours time cut-off. Note the use of the branch & bound solvers increases the compile time by 50% on average. However, the compile time slowdown to the heuristic solver can be substantial (e.g. 186.crafty benchmark) reaching factors up to 30. Detailed compile time statistics for both our heuristic *PBQP* solver and the optimal *B&B* algorithm are given in Table 3.5.

	mem	compile time [sec]					solver statistics		
benchmark	[kb]	*LLVM*	HEU	*B&B*	$\frac{\text{HEU}}{\text{LLVM}}$	$\frac{B\&B}{\text{LLVM}}$	opt_1	opt_2	sub.
gzip	49	0.16	0.34	0.49	2.13	3.06	1080	118	6
vpr	148	1.07	2.09	3.07	1.95	2.87	5176	403	51
gcc	537	10.24	21.78	33.05	2.13	3.23	72751	2640	109
mcf	99	0.06	0.13	0.18	2.17	3.00	381	35	0
crafty	220	1.42	3.38	102.84	2.38	72.42	6527	765	49
parser	55	0.68	1.44	2.05	2.12	3.01	5729	245	23
perlbmk	642	4.20	9.4	12.92	2.24	3.08	31526	1176	46
gap	384	3.37	7.69	11.24	2.28	3.34	27292	1587	7
vortex	200	2.34	5.18	7.15	2.21	3.06	18107	161	2
bzip2	69	0.13	0.27	0.41	2.08	3.15	879	124	2
twolf	101	1.64	3.25	4.56	1.98	2.78	8422	668	14

Table 3.5: Compile time and solver statistics for the SPECINT 2000 suite.

Short Summary Significant improvements of up to 57% for typical DSP code and up to 10% for MiBench and SPECINT 2000 benchmarks (5% on average) prove that there is significant potential in comparison with standard instruction selection techniques. Using a heuristic *PBQP* solver, all benchmarks could be compiled within less than half a minute, with about 99.83% of all problem instances solved to optimality. The comparison of the *PBQP* instruction selector with a linearization to integer linear programming confirms the efficiency and effectiveness of instruction selection based on *PBQP* solvers.

4 Spilling in the Context of SSA-Based Register Allocation

4.1 Introduction

Register allocation is a crucial task for optimizing compilers. The objective is to map an unlimited set of temporaries produced by the instruction selector to a finite set of machine registers such that temporaries with interfering live ranges are assigned different registers. Those temporaries that cannot be assigned during this process are mapped to locations in main memory. These memory accesses are very costly in almost every respect: code size is increased due to additional instructions transferring data to and from their assigned locations (spill code), performance is decreased as there is a widening gap among processor and memory speed, and – last but not least – overall energy dissipation significantly suffers from the additional workload. Thus, whether for general purpose computing or embedded systems, register allocation is a critical subtask.

In this work, we assume that register allocation is performed with respect to a fixed order of the instructions computed by a so-called pre-pass scheduler. This approach is by no means optimal as there are well known interdependencies among register allocation and scheduling. The order in which nodes in the dependence graph are evaluated has an important impact on the required number of registers. On the other hand, register allocation introduces additional false dependencies that constrain subsequent schedulers. While there are linear-time algorithms to compute an evaluation order requiring as few registers as possible for trees [SU70], the problems becomes *NP* complete for unrestricted DAGs [Set73]. Several authors propose scheduling heuristics that dynamically minimize register pressure if necessary [GH88, NP98]. Furthermore, Keßler and Rauber propose an optimal algorithm based on an efficient enumeration scheme that is able to compute optimal contiguous evaluations, even for relatively large DAGs [KR95]. Approaches that integrate several dependent code generation phases such as instruction selection, scheduling, and register allocation within a combined optimization model are briefly discussed in Chapter 1.

Register allocation has been among the first compiler-related problems that attracted significant interest in the research community. Already in the early eighties, Chaitin [Cha82] made one of the most influential contributions by establishing the close connection of register allocation with general graph coloring – a well known NP-complete problem. The graphs to be colored (so-called interference graphs) represent the unlimited set of virtual registers as nodes that are adjacent if they are concurrently live at any point in the program. In this setting, register allocation can be solved by finding a vertex coloring with at most K colors, where K denotes the number of machine registers. There are recent extensions for architectures with irregular register files [RN03, SRH04] and numerous follow-up papers have been published on two particular subproblems: *spilling* and *coalescing* [CH90, BCT94, GA96, PM98]. The objective of spilling is to defer a subset of the variables to memory so to allow for a valid allocation of the remaining temporaries while coalescing aims to minimize the overhead of register-to-register moves by assigning them to the same machine register. What is common to graph coloring based approaches and most techniques proposed around the turn of the century [GW96a, PS99, SE02b] is that both spilling and coalescing are considered to be inherent subproblems that are solved concurrently.

Appel and George [AG01] were among the first who proposed a *two-phase* approach where the spilling problem and the actual register assignment problem are decomposed and solved separately. More importantly, they showed empirically that the decomposition does not significantly degrade the overall allocation quality. Their original motivation was to decouple the tasks to allow for more efficient algorithms. However, in general it is not guaranteed that there is a valid k coloring even if there are no more than k variables simultaneously live at any given point in the program. To work around this, the authors introduced parallel copies at every program point that assign each variable to a freshly named temporary in the hope that subsequent coalescing will remove most of them.

The necessity for this last step disappears for programs in SSA form due to interesting properties of the corresponding interference graphs that have been discovered independently by several research groups [HG06, BDMS05]. The authors show that interference graphs of programs in SSA form are *chordal*. This special class of graphs allows to solve several problems, known to be NP complete for general graphs, in polynomial time, e.g., maximum clique, maximum independent set, minimum clique covering, and minimum coloring. In particular, one can compute an optimal coloring of a graph $G(V, E)$ in $\mathcal{O}(|V| + |E|)$ time. The chromatic number corresponds exactly to the maximum number of registers that are simultaneously live. Furthermore, Φ-operations can be eliminated in a way that no additional registers are necessary, though for the cost of additional swap operations. Thus, spilling can be performed as a separate step before coloring, coalescing, and SSA destruction while avoiding iterations among the particular steps. The fact that most modern compiler infrastructures maintain SSA form throughout the compilation process anyway is another strong incentive for recent SSA based approaches [HG06, PP08]. The large number of additional copy-related variables caused by Φ-nodes complicates coalescing. However, recent work [GH07] suggests that the problem can be solved efficiently in practice, and near-optimal polynomial time algorithms are likely to be found in the near future.

This chapter presents a new approach to spilling by modelling it as a discrete combinatorial optimization problem. Most previous work on this topic is based on the *spill-everywhere* model, i.e., a variable is removed entirely from the interference graph requiring a corresponding store instruction after each definition and a re-load instruction right before each use. This model is largely motivated by graph coloring register allocation where nodes that cannot be colored are simply removed from the graph, effectively spilling the entire live-range. In this chapter, we consider a more flexible model also known as *load-store optimization*, where live-ranges can be split arbitrarily. We assume a *RISC* like instruction set with explicit load and store instructions to transfer values from and to dedicated memory locations. Programs are converted to *strict edge-split* SSA form i.e., there is a single static definition for each value, each use of a variable is dominated by its definition, and there is no control flow edge that leads from a node with multiple successors to a node with multiple predecessors. The main motivation for our choice is that spilling is most useful within such a setting and some descriptions are more comprehensible. However, our concepts easily translate to general programs with multiple definitions per variable.

This chapter is organized as follows: The next section gives an overview of SSA-based register allocation. Section 4.7 covers related work for the spilling problem. A formal problem description along with an 0/1 linear programming formulation is presented in Section 4.3 and Section 4.4 respectively. We show how this formulation can be used to derive a polynomial time algorithm using Lagrangian relaxation in Section 4.5. The technique is evaluated both for a traditional embedded ARM micro-architecture and CHILI – a 4-way VLIW processor. Experiments with SPECINT 2000 benchmarks and embedded media applications (Section 4.6) show that spilling is a major performance contributor and can be solved to optimality even for large benchmarks in reasonable time.

4.2 SSA-Based Register Allocation

Our approach to spilling is based on programs in SSA forms due to the following reasons

- The chromatic number of interference graphs for programs in SSA form equals the maximum number of registers that are simultaneously live. Thus, there is no necessity to insert additional spill code in the register allocator after the pre-spilling phase.
- Programs in strict SSA form have the useful property that each variable is defined exactly once and each use is dominated by a definition. Thus, the defining label constitutes a natural choice for the insertion of spill code that saves a value to memory.

It is important to note that the proposed algorithm can also be combined with traditional register allocators that are applied after SSA elimination. In this case it cannot be guaranteed that there is an allocation without further spills and the register allocator has to bring its own spilling heuristics. However, the amount of additional spill code is usually much smaller after a pre-spilling phase, decreasing the effect of simplistic heuristics on the final code quality.

Traditional graph-coloring register allocators invoke a spilling heuristic if they reach a point where no more physical registers are available; see Figure 4.1 (a). This can be an expensive process and requires to tightly integrate spilling and coalescing heuristics

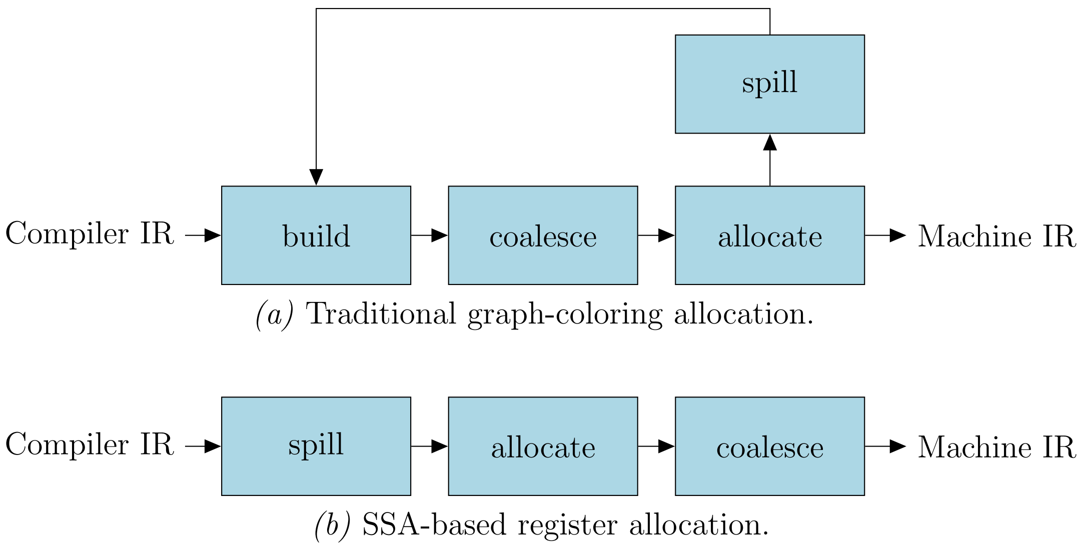


Figure 4.1: Properties of programs in SSA form allow to execute the spilling phase only once (b) instead of the typical iteration scheme found in graph-coloring allocators (a).

with the register allocator. In contrast, SSA-based register allocators allow for a phase decoupling as depicted in Figure 4.1 (b). Each phase is executed only once. SSA destruction is usually handled in the coalescing phase.

Assignment Phase A very tempting technique for the assignment phase is graph coloring for the reasons already pointed out in the introduction, i.e., interference graphs for programs in SSA are chordal and allow thus for polynomial time algorithms for graph coloring. This can be accomplished by finding a so-called *perfect elimination order*, which can be obtained by successively removing *simplicial* nodes until all vertices have been processed. A vertex u is called simplicial, if u together with its adjacent nodes induces a clique in the interference graph. There are always at least two simplicial vertices in a chordal graph [Dir61]. Furthermore, removing a node from a chordal graph leaves it chordal. Thus, the procedure always succeeds. The chromatic number K equals exactly the size of the largest clique. Once a perfect elimination order has been obtained, finding an optimal coloring can simply be done by re-inserting the nodes in reverse order. After each insertion, the node is colored with a color that is not yet used by its neighbors. The vertex is simplicial, thus its neighbors form a clique of at most $K - 1$ nodes and there is at least one color left.

A technique that is able to cope with irregular register files and aliasing is proposed by Pereira et al. [PP08]. Their algorithm views the allocation problem as the problem of solving a set of puzzles. The register file corresponds to the puzzle board and program variables represent the puzzle pieces. The complexity of the puzzles depends on the structure of the puzzle board. Spilling is invoked if there is no solution to a given puzzle.

However, the algorithm can also be combined with a pre-spilling phase as proposed in this work such that no further spilling is necessary.

Coalescing Phase Coalescing aims to remove register-to-register copies in a program by assigning the same register to both the source and the destination variable. If such an assignment succeeds, the original copy instruction can be removed, which benefits both performance and code size. However, there is also a trade-off: coalescing live ranges has a negative impact on the interference graph that might lead to additional spill code. Several approaches account for these effects and restrict coalescing or avoid harmful transformations by speculative coalescing that can be undone if necessary [BCT94, GA96].

For SSA-based register allocators, coalescing plays a special role. Implementing Φ nodes in a naïve way by replacing them with simple copy instructions may raise the register demand and destroy the chordality of interference graphs. In order to account for the parallel semantics of Φ nodes, they are interpreted as *permutations* the registers on incoming control flow edges [Hac07]. These permutations can be implemented by swap instructions in practice.[1] Thus, coalescing has to maximize the number of *fixed points* of these permutations.

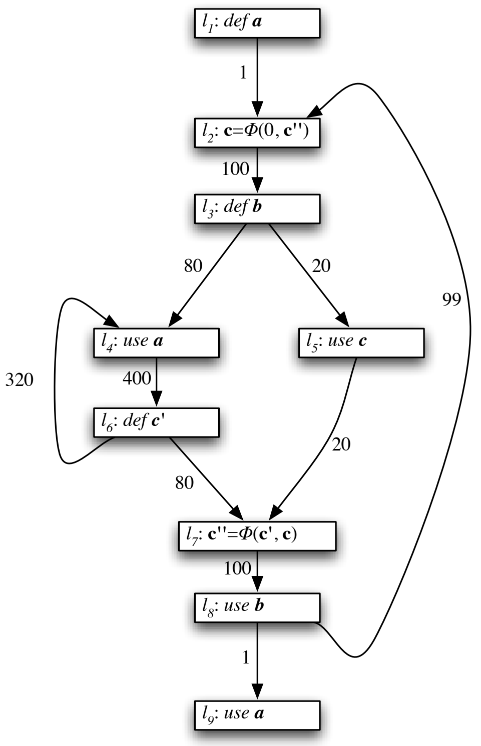
The starting point for coalescing is an interference graph $G = (V, E, A)$ where V and E denote the set of vertices and edges of the interference graphs as usual and A is an additional set of edges representing the *affinity* among the adjacent nodes. These affinity edges have positive weight that represents a penalty whenever the adjacent edges are assigned *different* colors. Affinity edges are inserted for each explicit copy instruction. Additionally, there are affinity edges among operands of a Φ function and the corresponding variable at the left-hand-side. The objective is to find a valid coloring such that the penalties incurred by affinity edges is minimized.

Coalescing for programs in SSA form is known to be *NP* complete [Hac07]. However, solving these problems to optimality seems to be feasible in practice even for large sized instances. Experimental evidence therefore is given by Grund et al. [GH07] who propose an algorithm based on integer linear programming.

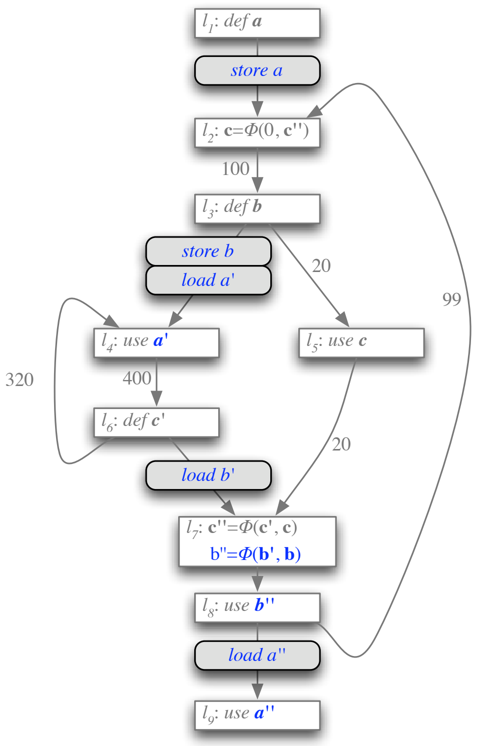
4.3 Motivation and Modeling

Our approach to the spilling problem is based on a reduction to a well-defined combinatorial optimization problem. The graph in Figure 4.2(a) shows a node-labeled control flow graph (CFG) in SSA form with program variables $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$. For the latter variable, Φ-functions have been inserted that disambiguate multiple reaching definitions. Edge frequencies are denoted along the arcs in the CFG. Liveness for strict variables is defined in the usual way: a variable v is *live* at label ℓ, if there is a (possibly empty) path from ℓ to a label ℓ' such that $v \in \mathcal{U}_{\ell'}$ and the path does not include $\text{def}(v)$. In our example, variable $\boldsymbol{a}$ is live at labels l_2 to l_9 while the live range of variable $\boldsymbol{b}$ spans across labels l_4 to l_8. Note the special meaning of ϕ-functions in this respect: their arguments are only

[1]On architectures without a native swap instructions, a sequence of simple xor instructions can be used instead.



(a) Input program in SSA form



(b) Spilling transformation for two registers

Figure 4.2: Motivating example.

used if control flow enters a label along the corresponding in-edge, e.g., variable $\boldsymbol{c}$ is live at label l_5 but not at l_6. For the consideration of liveness, we can treat phi functions as if their arguments are used at the end of the particular predecessor block. We are assuming a RISC architecture where all arguments to an instruction have to reside in a register. Likewise, results are always written into one or more destination registers. Transfers from memory into registers and vice versa can be accomplished using explicit load and store instructions. Consequently, the number of registers live at a particular label ℓ cannot exceed $k - |\mathcal{D}_\ell|$ in order to allow for a coloring with k registers.

The objective of spilling is to insert load- and store-instructions along the edges of the CFG in order to split live ranges such that that the overall costs are minimized. Costs may be constant to minimize for code size, proportional to the edge frequencies to optimize for execution time, or any combination of the two. Furthermore, we can easily add support for re-materialization by justifying the cost-function accordingly. This is most useful for constants or constant expressions such as frame-pointer indirect addressing.

Assuming a total number of two registers, we show a cost-minimal transformation with respect to the given edge-weights for our example in Figure 4.2(b). Live ranges for variables $\boldsymbol{a}$ and $\boldsymbol{b}$ have been split by storing them to a dedicated location in memory at the point of definition and re-loading them before they are used. In order to maintain SSA form, we have to insert additional Φ-nodes and rename references to reflect those changes accordingly. A very efficient standard algorithm therefore is given by Cytron et al. [CFR+91b].

Many algorithms insert re-load instructions for spilled variables right before they are used. However, this can be arbitrarily bad in general, e.g., inserting a re-load for variable $\boldsymbol{a}$ in Figure 4.2(b) within the inner loop right before label l_4 increases the costs of the transformation significantly.

Considering a *single* variable v, we can compute the overall costs for reloading its value from memory using a simple min-cut computation. Therefore, we transform the control flow graph into a weighted network N_v as shown in Figure 4.3. For each node in the CFG at which v is live, we also generate a node in our min-cut network. Likewise, edges are introduced that reflect the cost of inserting a re-load instruction. Furthermore, we introduce an artificial source s and sink t respectively. For all successors of $\mathrm{def}(v)$, we add an additional edge from s to the corresponding node in N_v with weight zero. This reflects the fact that a value is always available right after its definition. For each label at which v is used, an arc with cost ∞ is inserted. The intention of this transformation is as follows: any s-t cut with weight less than ∞ corresponds to a valid segmentation of the original live range such that re-load instructions are placed on every cut-edge passing from a node in S to a node in partition T. The weight of the cut reflects precisely the costs of the transformation in the chosen cost model. The cut that has been chosen for the example in Figure 4.2 is depicted by the bold dotted line. It consists of the edges (l_3, l_4) and (l_8, l_9) which are exactly the places where we inserted the re-load instructions before.

Solving the min-cut problems isolated for each variable does not lead to a meaningful solution as the model always allows for the trivial solution where the S-partition is

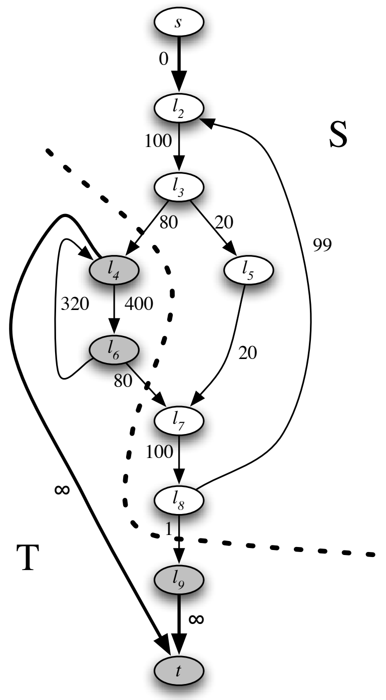


Figure 4.3: Modeling for a single variable.

constituted by nothing but the s-node. This is the equivalent of assigning a machine register to each variable for the entire live range. In order to account for register constraints, we identify source and sink nodes for each of the generated networks (one per variable). Thus, we obtain a combined network flow problem for the whole function. Nodes in the combined graph are assigned to disjoint partitions. For each label ℓ in the CFG, we define a partition Q_ℓ that includes all the nodes from networks N_v that correspond to label ℓ. Thus, each partition Q_ℓ includes exactly one node per variable live at ℓ. Furthermore, a partition Q_ℓ has *capacity* $k - |\mathcal{D}_\ell|$. Intuitively, the capacity of a partition denotes the number of live ranges that may pass through a particular label without exceeding register constraints.

We can thus reduce the problem where to insert re-loads to the problem of finding a minimum cut in the combined network subject to the conditions $|T \cap Q_\ell| \leq k - |\mathcal{D}_\ell|$ for each partition Q_ℓ. This model allows us to formulate spilling as a well-defined combinatorial optimization problem and its similarity with generic network flow problems allows us to derive some interesting properties from graph theory. The model also allows for the design of simple greedy heuristics as it is straight-forward to find feasible solutions, e.g., by heuristically assigning $|Q_\ell| - k + |\mathcal{D}_\ell|$ nodes to the S partition.

The proposed model assumes that the value of each definition is always available in memory. This is easy to achieve in strict SSA form as all uses are dominated by their definition. It is sufficient but not necessarily optimal to store a value right after its point

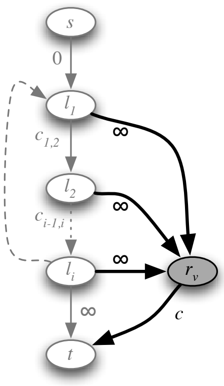


Figure 4.4: Accounting for store costs.

of definition. As these spill instructions usually do not come for free, it might be desirable to account for their costs within the optimization model. One way to incorporate them to the constrained min-cut problem presented so far is shown in Figure 4.4. For each variable v, an additional node r_v is inserted. We introduce an additional arc (r_v, t) whose weight corresponds to the costs of storing the particular variable. We have to account for those costs only if there is at least one re-load operation. In other words, there is a node other than s that is assigned to the S partition. We can model this constraint by adding additional arcs with weight ∞ from each label other than s and t to the newly introduced node r_v. Those edges imply that $r_v \in S$ if any of the adjacent nodes other than t is in S. For values that allow for re-materialization such as initializations with constants or constant expressions, store costs are usually 0 and the additional node can be safely removed. Otherwise, the same considerations discussed for re-loads apply: the cost function can be used to optimize for any combination of code size and performance.

4.4 Constrained Min-Cut

We are now going to formalize the proposed constrained min-cut problem (CMC) in more detail.

Definition 4.4.1. *Let $G(V, E)$ be a digraph with edge weights $w(u, v) \in \mathbb{Z}^+$, for each edge $(u, v) \in E$. Further, let $s, t \in V$ denote two distinguished vertices and let $P = \{\{s\}, \{t\}, Q_3, \ldots, Q_r\}$ denote a disjoint partitioning of the nodes in V. Each disjoint set Q_i, for all i, $(1 \leq i \leq r)$, has associated a capacity $c_i \in \mathbb{N}$.*

Find a separation of V into two disjoint sets S and T such that $s \in S$, $t \in T$, for all i, $(1 \leq i \leq r)$, $|T \cap Q_i| \leq c_i$, and $\sum_{(u,v)\in(S\times T)\cap E} w(u, v)$ is minimal.

We may formulate CMC as a quadratic integer program as follows.

$$\begin{aligned} \min \quad & \sum_{(u,v)\in E} w(u,v)(1-x_u)x_v & \\ s.t \quad & x_t - x_s \geq 1 & \\ & \textstyle\sum_{u\in Q_i} x_u \leq c_i & \text{for all } 1 \leq i \leq r \\ & x_u \in \{0,1\} & \text{for all } u \in V \end{aligned} \tag{4.1}$$

Decision variables x_u are 0-1 integer variables with the interpretation that x_u is zero if vertex u is in S and one if it is in T. Linearizing the given model leads to the following ILP.

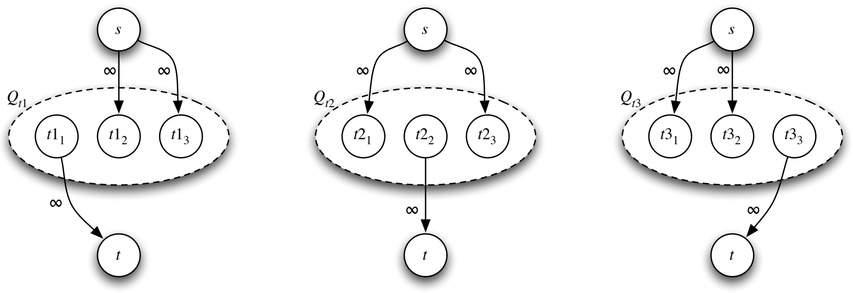
$$\begin{aligned} \min \quad & \sum_{(u,v)\in E} w(u,v)y_{uv} & \\ \text{s.t.} \quad & x_t - x_s \geq 1 & \\ & x_u - x_v + y_{uv} \geq 0 & \text{for all } (u,v) \in E \\ & \sum_{u\in Q_i} x_u \leq c_i & \text{for all } 1 \leq i \leq r \\ & x_u \in \{0,1\} & \text{for all } u \in V \\ & y_{uv} \in \{0,1\} & \text{for all } (u,v) \in E \end{aligned} \tag{4.2}$$

Disregarding the capacity constraints in line four of Equation 4.2, the model corresponds exactly to the well-known min-cut problem, which has some interesting properties [AMO93, Vaz04]. First, its constraint matrix is *totally unimodular*. Thus, even if we relax the integrality constraints, each extreme point solution can be shown to be integral, with each coordinate being 0 or 1. This clearly implies that the problem can be solved in polynomial time. In fact, there are max-flow algorithms that can be used to compute a minimum cut in time $\mathcal{O}(|V||E|\log|V|)$. Unfortunately, adding the capacity constraints renders the problem NP complete.

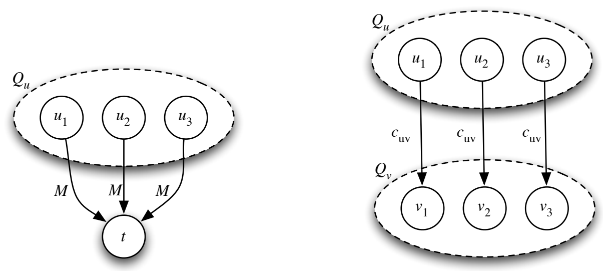
Theorem 4.4.2. *The decision problem of CMC is NP complete.*

Proof. The problem is clearly in NP because a node u is either in set S or in set T. This decision can be non-deterministically taken per node and it can be verified in polynomial time whether a solution is feasible, i.e., the weight of the cut is below a given threshold. We show the NP completeness by reducing the multi-way cut problem to the CMC problem.

The *multi-way cut problem* is defined as follows: Given a digraph $G(V,E)$, edge weights $w(u,v) \in \mathbb{Z}^+$ for all edges, and terminal vertices $F \subseteq V$, a multiway cut is a set of edges whose removal disconnects the terminals from each other. The multiway cut problem asks for the minimum weight of such a cut set. The corresponding decision problem is



(a) Terminal t_1 (b) Terminal t_2 (c) Terminal t_3



(d) Node $u \in V \setminus \{t_1, t_2, t_3\}$ (e) Edge $(u, v) \in E$

Figure 4.5: Reduction of multi-way cut to CMC.

NP complete [DJP+94] for $|F| \geq 3$. The multi-way cut problem can also be seen as a placement problem, i.e., a node is assigned to a disjoint set of a terminal vertex in F. Cut edges are edges whose source is assigned to a different disjoint set than its tail.

We reduce all instances of the 3-multi-way cut problem to CMC as outlined in Figure 4.5. For each node u in the multi-way cut we construct a disjoint set Q_u with capacity c_u equal to one in the CMC problem. Every disjoint set Q_u contains three nodes $\{u_1, u_2, u_3\}$ and, thus, at most one of the nodes can be placed in set T. There are four possible solutions in the CMC problem for the three nodes in Q_u: (S, S, S), (T, S, S), (S, T, S), and (S, S, T) where S and T refer to the placement in which disjoint set CMC places the three nodes of Q_u. To reduce the number of possible solutions for disjoint set Q_u to three possible solutions, we add the edges $\{(u_1, t), (u_2, t), (u_3, t)\}$ with edge weight M to the digraph as shown in Figure 4.5(d). With these three additional edges possible solution (S, S, S) attracts $3M$ additional costs per node. If M is a sufficiently large integer number, (S, S, S) is excluded as a possible solution because solutions (T, S, S), (S, T, S), and (S, S, T) have costs of $2M$.

The interpretation of a possible solution is the placement of node u in either disjoint set of terminal vertex t_1, t_2, or t_3 of the multi-way cut problem. For terminal vertices t_1, t_2, and t_3 we make sure that there exists only a single possible solution by adding edges with infinite costs as shown in Figure 4.5(a)-(c) forcing CMC to assign node t_1 the solution (T, S, S), t_2 (S, T, S) and t_3 (S, S, T). Modeling the costs of an edge in the multi-way cut problem is given in Figure 4.5(e). Every edge (u, v) in the 3-multi-way cut problem is mapped to three edges in the CMC problem as shown in Figure 4.5(e). An edge (u, v) in the multi-way cut problem connects two partitions Q_u and Q_v in the 3-multi-way cut problem. There are three possible solutions for Q_u and Q_v and the costs for each pair of solutions is given below in the table:

(Q_u, Q_v)	(T, S, S)	(S, T, S)	(S, S, T)
(T, S, S)	0	w_{uv}	w_{uv}
(S, T, S)	w_{uv}	0	w_{uv}
(S, S, T)	w_{uv}	w_{uv}	0

If solutions of Q_u and Q_v are the same, the edges in CMC are no cut edges; otherwise there exists exactly one cut edge between the two disjoint sets Q_u and Q_v since one edge is from S to S, one edge is from T to S and is no cut edge, and one edge is from S to T which is a cut edge and attracts of costs of w_{uv}. □

The ILP formulation given in Equation 4.2 provides already a feasible algorithmic approach to the CMC problem. As our experiments show, mature solver technology can be used to solve instances, even from very large functions, within reasonable time limits. However, the special structure of the polytope permits a *decomposition* of the problem by relaxing a subset of the constraints. The remaining problem can be solved using generic algorithms for network flow problems allowing us to draw upon standard algorithms from graph theory that are widely available.

4.5 Lagrangian Relaxation

Lagrangian relaxation [AMO93] is a general solution approach for mathematical programs that allows for the decomposition of problems in order to exploit their special properties. In particular, this approach is perfectly tailored for problems with an embedded network structure. We are now going to apply this technique to the CMC problem in order to derive a polynomial-time near-optimal algorithmic solution approach that does not rely on integer linear programming.

Therefore, consider the CMC formulation presented in Equation 4.2 and let X denote the set of solutions that satisfy the constraints of the basic min-cut problem without the additional capacity constraints. We can reformulate the constrained min-cut problem as follows:

$$z^* = \min \{cx : x \in X, Ax \leq d\}.$$

The side constraints $Ax \leq d$ denote the capacity constraints for partitions Q_i with capacity vector d. Relaxing those side constraints, we obtain the following Lagrangian relaxation:

$$L(\mu) = \min \{cx + \mu(Ax - d) : x \in X\}.$$

We hereby effectively remove a subset of the constraints bringing them into the objective function with associated *Lagrangian multipliers* μ. Thus, a solution to the relaxed problem is not necessarily feasible for the original problem. However, LP theory states that for any value for the Lagrangian multipliers μ, the value $L(\mu)$ is a *lower bound* on the optimal objective value of the original problem. The sharpest possible bound is given by a solution to the so-called *Lagrangian multiplier problem* $L^* = \max_{\mu \geq 0} L(\mu)$. We have the following well-known relations among those optimization problems

$$L(\mu) \leq L^* \leq z^*.$$

This is useful as it provides us with an optimality test: for any vector of Lagrangian multipliers μ, if x is a feasible solution to the original optimization problem and $L(\mu) = cx$, then $L(\mu)$ is an optimal solution of the Lagrangian multiplier problem and x is an optimal solution for the original CMC problem. Furthermore, if for some choice of μ, the solution x^* of the Lagrangian relaxation is feasible for the original optimization problem and additionally x^* satisfies the *complementary slackness condition* $\mu(Ax^* - b) = 0$, then x^* is also an optimal solution to the original CMC problem.

Lagrangian relaxation is mainly useful if we have a way to solve the Lagrangian relaxation efficiently. Considering the situation for the CMC problem, we can state $L(\mu)$ as follows:

$$\begin{array}{lll}
\min & \sum\limits_{(u,v)\in E} w(u,v)y_{uv} + \sum_{i=1}^{r} \mu_i \sum_{u \in Q_i} (x_u - c_i) & \\
\text{s.t.} & x_t - x_s \geq 1 & \\
& x_u - x_v + y_{uv} \geq 0 & \text{for all } (u,v) \in E \\
& x_u \in \{0,1\} & \text{for all } u \in V \\
& y_{uv} \in \{0,1\} & \text{for all } (u,v) \in E
\end{array} \tag{4.3}$$

The polyhedron defined by the set of inequalities already corresponds to a standard min-cut problem. The main difference is in the objective function: instead of minimizing solely the weight of the cut, there is an additional Lagrangian term $\sum_{i=1}^{r} \mu_i \sum_{u \in Q_i} (x_u - c_i)$. In other words, for each partition Q_i, there is an associated Lagrangian multiplier μ_i that gets added to the objective function for each node that ends up in the T-partition. Note that, for a constant μ_i, in the Lagrange term $\mu_i c_i$ evaluates to a constant and we can rewrite the term to $\sum_{i=1}^{r} \mu_i \sum_{u \in Q_i} x_u$ and consequently to $\sum_{u \in V} \mu_{\chi(u)} x_u$ where $\chi(u)$ is index i such that $u \in Q_i$. Note that index i is unique because the node set V is disjointly partitioned. We are now going to present how we can transform the problem to a standard s-t min-cut instance that can easily be solved using an appropriate max-flow algorithm.

In fact, the problem at hand is a special case of *Stone's problem* [Sto77]. The original motivation was a process allocation problem: a set of tasks $T = \{t_1, \ldots, t_r\}$ is to be mapped to two processors α and β. Each task may have a different execution time on each of the processors and there are communication costs if two tasks are mapped to distinct processors. More formally, let $w_\alpha(t)$ and $w_\beta(t)$ denote the executions costs for mapping task t to processors α or β and let $w_{\alpha\beta}(t)$ denote the communication costs, the

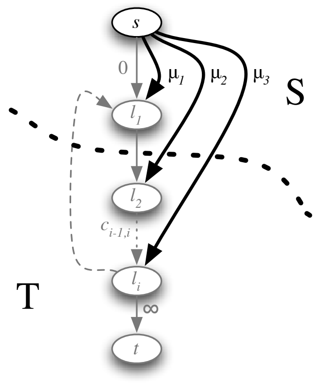


Figure 4.6: Reduction of the relaxed problem $L(\mu)$ to a standard s-t min-cut instance.

objective is to minimize the following objective function:

$$\min \sum_{t\in\alpha} w_\alpha(t) + \sum_{t\in\beta} w_\beta(t) + \sum_{t_1\in\alpha, t_2\in\beta} w_{\alpha\beta}(t_1, t_2). \tag{4.4}$$

Stone solves the problem algorithmically by employing a s-t min-cut reduction that is very similar to our approach. Tasks are represented by nodes in the network. Edges among a distinguished source s and each node correspond to the costs of executing a particular task at processor α. Likewise, there are edges among nodes and an artificial sink t that correspond to the execution costs at processor β. Communication costs are represented using edges among the particular nodes. It is easy to see that any s-t min-cut directly corresponds to an optimal solution for the minimization problem given in Equation 4.4.

We use basically the same construction in order to account for the additional costs of nodes mapped to the T partition for the proposed relaxation $L(\mu)$. The original network is augmented with edges of the form (s, v); see Figure 4.6. As there is at most one partition Q_i that includes v, the weight of those edges reflects exactly the value of the corresponding Lagrangian multiplier μ_i. Each edge contributes to the weight of a s-t min-cut in the augmented network if and only if the corresponding node v is in the T partition. Thus, we can use a generic max-flow algorithm to solve $L(\mu)$ quickly. In practice, we use an implementation of the push-relabel algorithm with time complexity $\mathcal{O}(|V|^3)$. However, there are algorithms with a slightly better time characteristic of $\mathcal{O}(|V||E|\log|V|)$.

While we now have an efficient vehicle in order to solve the relaxed problem for a fixed μ, it remains to show how to solve the Lagrangian multiplier problem $L^* = \max_{\mu\geq 0} L(\mu)$. We therefore use a variant of *subgradient optimization* that is an adaption of Newton's method for solving systems of non-linear equations. The basic principle is to gradually adapt the vector of Lagrangian multipliers for an initial choice μ^0 as follows:

$$\mu^{k+1} = [\mu^k + \theta_k(Ax^k - d)]^+.$$

The scalar θ_k denotes the *step length* in the k-th iteration and x^k is a solution vector of the subproblem $L(\mu^k)$. As we are relaxing inequalities, we never consider negative elements in our vector μ. Thus, μ^k is set to zero if the update strategy would cause it to become negative (denoted by $[\ ]^+$). The choice of the step length is important in practice for convergence of the process. We use a popular standard heuristic that adapts the step length after every iteration as follows: $\theta_k = \frac{\lambda_k(U-L(\mu^k))}{||Ax^k-d||^2}$. In this expression, U denotes the best upper bound to the problem found so far, λ_k is a scalar that is gradually decreased and $||Ax^k - d||^2$ denotes the Euclidean norm of the inner term.

The process allows for an intuitive representation: if for any partition Q_i, the term $(Ax^k - d)_i$ is zero or negative, the capacity constraints imposed on Q_i are satisfied. However, if the term is greater than zero, μ_i serves as a *penalty* that directs the min-cut algorithm to put some of the nodes from the T to the S-partition. The step length directs the algorithm to move quickly towards the optimum at the beginning of the approximation, while we proceed with more care once we are close to the optimal value L^*.

A theoretical discussion of convergence criteria and the rationale for choosing the step length is beyond the scope of this work. As it is a standard technique in combinatorial optimization, the interested reader is referred to relevant literature, e.g., Ahuja et al. [AMO93].

In general, a solution to the Lagrangian multiplier problem is not necessarily feasible for the original optimization problem. A popular approach is to use the values obtained from the relaxation in order to solve the remaining problem using enumeration techniques such as branch-&-bound. The efficiency of those techniques largely depends on the size of the *duality gap*. An interesting result in combinatorial optimization states that the bound obtained from Lagrangian relaxation is always as sharp as the bound obtained from an LP relaxation. An even stronger statement for problems satisfying the *integrality property* (such as s-t min-cut) guarantees that both bounds are exactly equal. Thus, solving the Lagrangian multiplier problem is equivalent to solving the LP relaxation but does not rely on linear programming and can often be solved more efficiently.

An alternative technique that can be used to obtain near-optimal solutions is the use of so-called *Lagrange heuristics*. We have implemented several greedy strategies and present their results in the following section. The main idea is that solutions to the Lagrangian multiplier problem are usually very close to the solution of the original problem while only a small number of relaxed constraints remain violated. A Lagrange heuristic is an algorithm that resolves violated constraints in a greedy manner, often achieving excellent results. An additional advantage is that the computed bound provides us with an *performance guarantee* in respect to the optimal solution.

4.6 Experimental Evaluation

We have implemented and evaluated the proposed techniques using LLVM; see Section 1.3.. All programs have been cross compiled using one core of a Xeon DP 5160 3GHz. The ILP formulation for CMC is solved using ILOG CPLEX(tm) 10. Standard graph algorithms such as an implementation of the push-relabel max-flow algorithm are taken from the boost graph library.

First, we are interested in the potential for exact spilling compared to heuristics. As a reference, we use the default allocator of LLVM 2.4 – an improved implementation of linear scan register allocation with backtracking in the case of spilling. We compare results for a varying number of available registers with a modified backend where we perform spilling right before the standard register allocator. Register allocation in LLVM is traditionally performed after SSA elimination. Thus, even though we spill a sufficiently large number of registers, the register allocator might insert additional spill code that is not strictly necessary. We present benchmarks for two representative embedded architectures: a 4-way VLIW core for audio-/video-decoding and an ARM processor as an example for an embedded RISC architecture.

The first experiment is based on an OnDemand(TM) CHILI core; see Section 1.2. Execution times have been gathered using a cycle-accurate simulator. We use typical benchmarks reflecting the characteristics of embedded media applications provided by OnDemand(TM), most of them are taken from the freely available MiBench suite. Both the simulator and the compiler have been modified to support a varying number of registers, allowing us to make experiments for several different settings.

We present data for various configurations with 8, 12, 16, and 32 registers in Fig. 4.7. For small register files, the achieved speedup on top of LLVM is substantial for all benchmarks, ranging from about 8% to almost 30%. Not surprisingly, the potential for improved spilling techniques decreases with increasing number of machine registers. On average, there is an improvement of about 15.5% for the scarcest setting, gradually decreasing to 6.5%, 3%, and 0.9% for 12, 16, and 32 registers respectively. None of the benchmarks showed additional speedup for 64 registers.

Solver times for those benchmarks are definitely within practical limits. Our ILP-based algorithm finished within a few seconds for most of the benchmarks, the most difficult being automotive-susan and video-h263 with 24 and 23 seconds respectively. This includes the time spent on the CMC reduction as well as SSA reconstruction after insertion of spill code.

In order to test our approach on larger benchmarks, we consider the widely-used SPECINT 2000 suite. Running those benchmarks on a bare-metal VLIW is infeasible as they require an underlying operating system and a complete `libc` implementation. Therefore, we use an ARMv7 board (OMAP3 EVM) at 500 MHz with 128MB of main memory running a Linux 2.6.22 kernel. Profiles are obtained using the input set “train” while the reported execution times were gathered using the “test” inputs. Floating point operations are emulated using a IEEE754 softfloat library. We measured execution times using the unix `time` utility considering the best out of 10 runs on an unloaded machine for each configuration.

ARM processors support two different instruction sets: ARM and ARM Thumb. While in ARM mode, the processor fetches 32-bit instructions and has access to the full register file with 16 registers. Thumb mode is a more compressed 16-bit instruction set architecture that allows for smaller code size at a significant performance penalty. In Thumb mode, most instructions can only access the lower half of the register file

			ARM			ARM Thumb		
Benchmark	Source [LOC]	CPLEX [sec]	BLY [sec]	LLVM [%]	CMC [%]	BLY [sec]	LLVM [%]	CMC [%]
164.gzip	5615	12.72	12.11	13.07	15.44	14.00	12.00	17.25
175.vpr	11301	103.44	12.85	6.64	6.73	16.50	1.54	0.61
176.gcc	132922	483.39	7.89	8.83	13.20	n/a	n/a	n/a
181.mcf	1494	1.16	1.30	1.56	1.56	1.41	1.44	4.44
186.crafty	12939	75.37	30.31	13.86	12.18	40.30	16.40	12.32
197.parser	7763	19.88	12.18	0.91	3.92	14.14	-5.80	2.54
253.perlbmk	72206	285.81	1.70	0.00	8.97	1.78	1.00	4.71
254.gap	35759	47.15	3.67	-0.81	3.67	4.21	-1.64	34.94
255.vortex	49232	1026.44	33.76	-0.79	-1.86	42.39	6.99	10.30
256.bzip2	3236	18.32	27.83	0.69	6.02	40.35	9.98	10.55
300.twolf	17822	124.74	1.34	11.67	17.54	n/a	n/a	n/a
Mean			13.18	**4.91**	**7.79**	19.45	**4.33**	**10.44**

Table 4.1: Experimental results using the SPECINT 2000 benchmark suite showing the execution times for both ARM and ARM Thumb mode. We compare three different algorithms: a generalized furthest first heuristic (BLY), the linear scan register allocator from LLVM 2.4 (LLVM), and our improved spill code placement algorithm (CMC). All results are relative to BLY. We use the geometric mean for speedups; for the absolute figures we use the arithmetic mean.

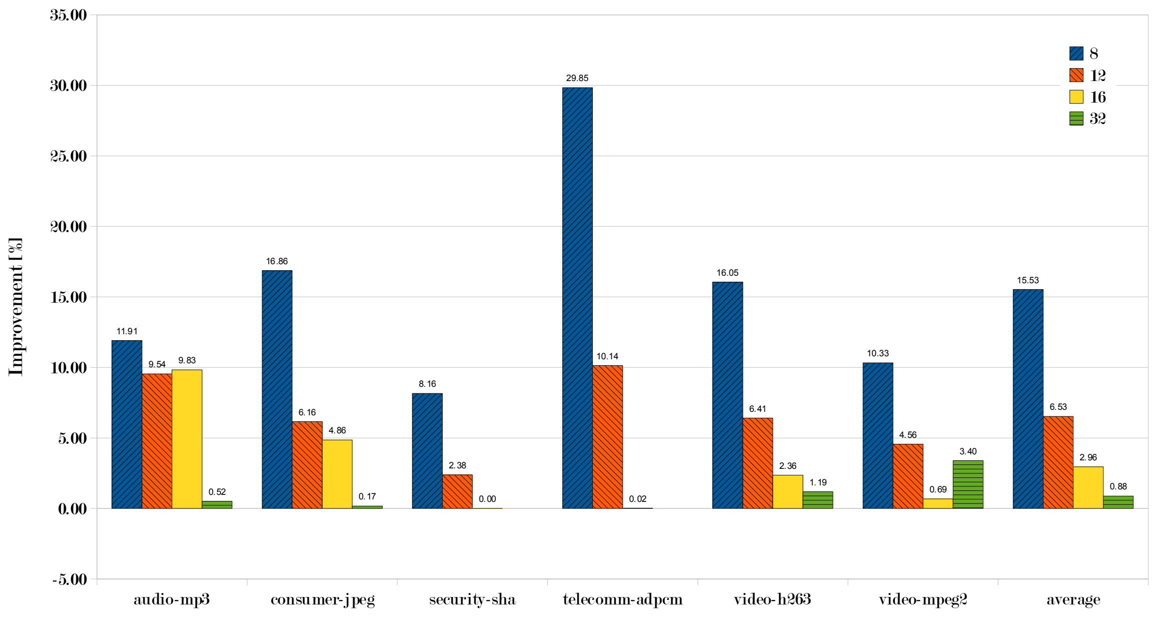


Figure 4.7: Improvement for a varying number of registers for spill code placement compared to the linear scan allocator of LLVM. Execution times have been gathered using a cycle-accurate simulator for a 4-way VLIW processor.

and only a subset of the addressing modes is available. Hence, the register pressure is much higher than in ARM mode. Note, that some of the architectural registers are special-purpose and cannot be used for generic program variables, e.g., program counter or stack pointer.

Table 4.1 shows the results for both ARM and ARM Thumb execution. We use (1) a generalization of Belady's algorithm as proposed in [HGG06] (BLY) as a baseline, (2) LLVM 2.4 standard spilling heuristic, and (3) our spill code placement based on CMC using CPLEX to compute an optimal solution. Note that in ARM Thumb mode two benchmarks fail due to bugs in the LLVM backend that we could not yet solve (i.e., 176.gcc and 300.twolf). On average, LLVM is about on par with a gcc cross-compiler at its highest optimization level. The spilling heuristic of LLVM shows an improvement of 4.91 and 4.33 percent for ARM and ARM Thumb mode respectively compared to the generalized Belady heuristic. The algorithm has been implemented as described in [HGG06], i.e., we compute partial solutions for basic blocks that are heuristically combined to obtain a solution for the whole function. The main reason for the performance regressions compared to the other algorithms is that there is no explicit considerations of loop structures and block weight, which often leads to avoidable spill code within inner loops. The speedup obtained with our CMC reduction compared to BLY is about 10.44% on average for ARM Thumb mode and 7.79% for the generic ARM instruction set.

Not surprisingly, most of the solver time is spent in a few large functions. Fig. 4.8

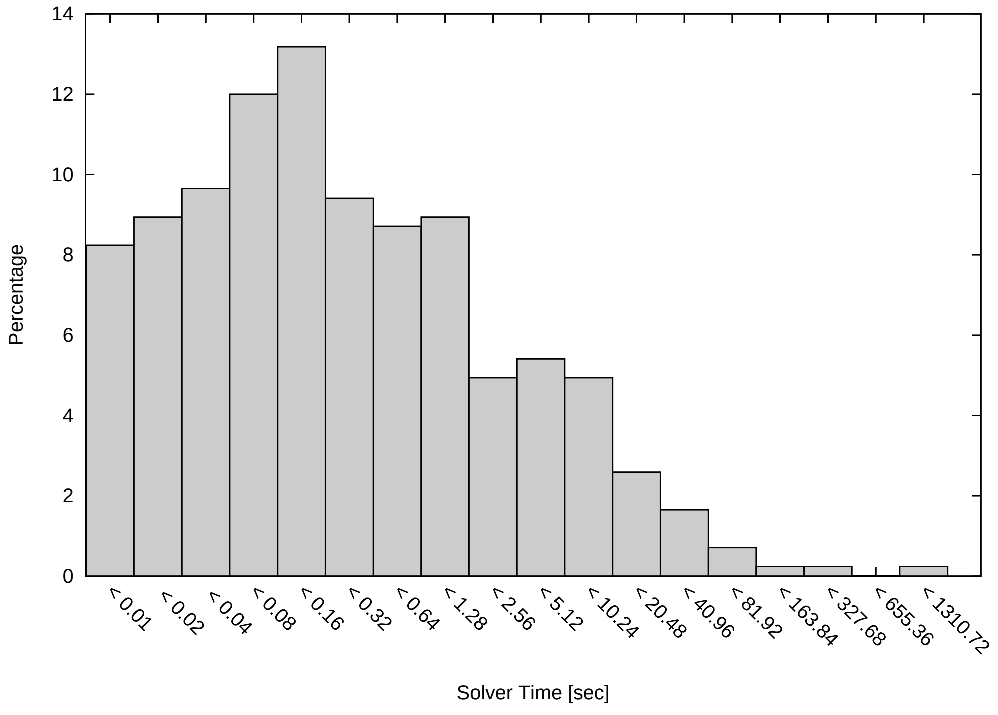


Figure 4.8: Distribution of ILP solver times for SPECINT 2000 with 14 general purpose registers (ARM) using ILOG CPLEX 10.

shows the distribution of solver times for CPLEX over the whole benchmark set. Note the logarithmic scale on the x-axis. Almost all of the benchmarks can be solved within 60 seconds. There are only three functions exceeding this limit, the largest of which is `BMT_Test` from 255.vortex consisting of about 4.500 basic blocks. The CMC reduction leads to a network with about 375.000 nodes and 418.000 edges and can be solved to optimality in slightly more than 13 minutes.

Lagrange Relaxation To overcome the limitations of ILP solvers, we have shown in Section 4.5 how the relaxation of capacity constraints leads to a problem that can be solved using efficient generic network flow algorithms. We evaluate the general solution approach for three different greedy Lagrange heuristics with increasing algorithmic complexity. For any partial solution, we visit each partition Q_i in order, while evicting $[|T \cap Q_i| - c_i]^+$ many nodes from the T partition according to one of the following strategies:

- *SIMPLE* Nodes are simply ordered according to their effect on the objective function, preferring those causing a low penalty.

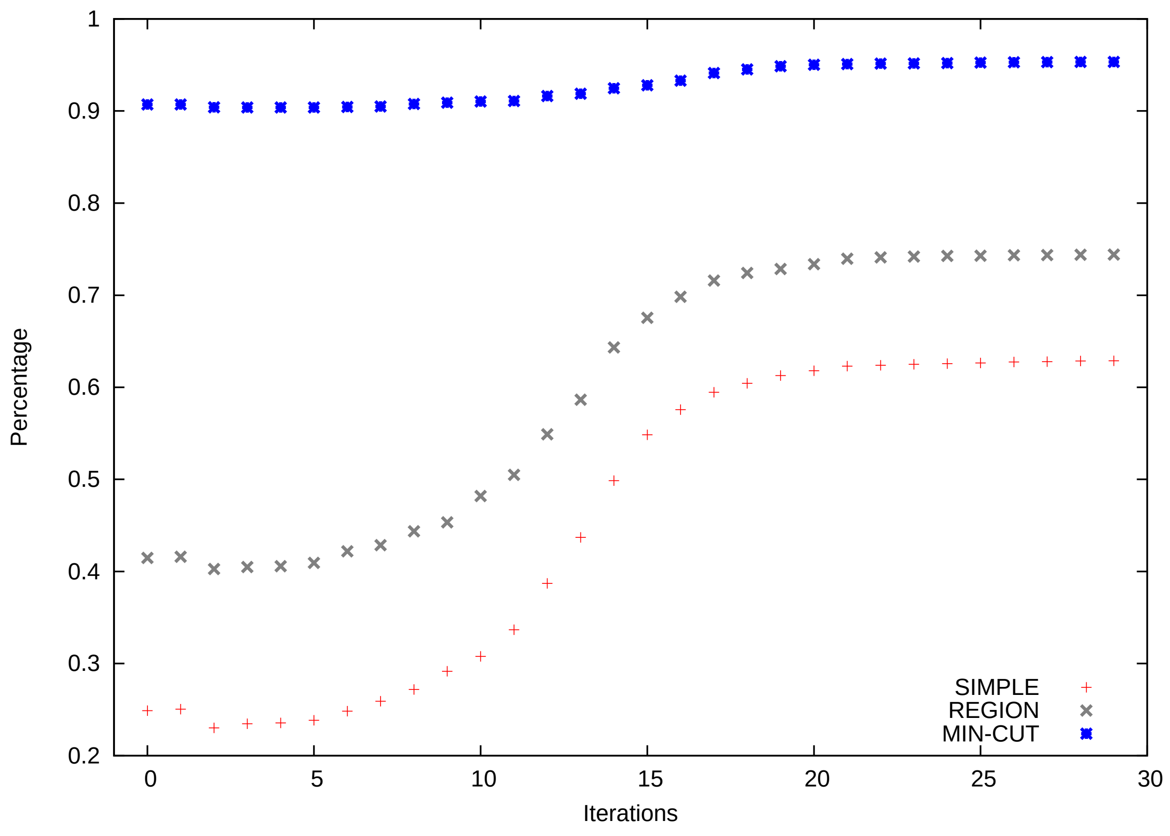


Figure 4.9: Average quality of three different Lagrange heuristics compared to the precomputed optimal solution over the whole SPECINT 2000 benchmark set. The x-axis denotes the number of iterations for the subgradient optimization algorithm.

- *REGION* This is basically the same strategy as before with the addition that we also remove nodes in the neighborhood as long as this does not *increase* the overall penalty. As we might well decrease the overall effect on the objective function, this approach can be seen as a simple hill-climbing heuristic.
- *MIN-CUT* This algorithms computes for each node within a partition the optimal set of nodes to be moved from the T to the S partition such that the overall weight of the cut is minimized.

Fig. 4.9 shows the average ratio of the optimal solution and the solution obtained from each of the greedy heuristics. The metric corresponds to the weighted spill costs as modelled in the CMC formulation. The x-axis denotes the number of iterations for the sub-gradient approximation algorithm. The average quality for the pure heuristic without the Lagrangian relaxation corresponds to $x = 0$. The two simple strategies perform initially very poor with an average of only 25% and 41% respectively. The graph clearly shows how approximations to the Lagrangian multiplier problem effectively guide the heuristics towards the optimum. After thirty iterations, the average quality for the simple strategies is improved to 63% and 75%. The MIN-CUT heuristic shows initially an average quality of almost 91% and climbs up to more than 95%.

Detailed performance results for the various algorithms are given in Figure 4.10. Each figure shows the runtime for all non-trivial benchmarks from the SPECINT 2000 suite. The y-axis shows the runtime of the algorithm in seconds. Each of the plots (a) to (e) features a polynomial asymptotic function that has been computed using a least squares approximation. Note that y-axis are drawn with logarithmic scale and that (a) and (b) show a smaller value range.

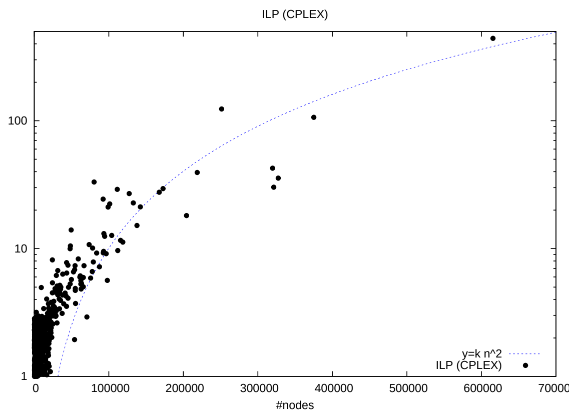
The performance of integer linear programming based algorithms strongly depends on the particular solver. It is very interesting that most of the problems are integral, even if integrality constraints are dropped. Among the whole benchmark suite, only 18 problems have a non-integral solution and require branch & bound. Thus, in practice almost all the time is spent in the simplex algorithm solving the LP relaxation. We compare two different ILP solvers: ILOG CPLEX and the open source GNU linear programming kit (glpk)[2]. Performance results for both solvers are shown in Figure 4.10 (a) and (e) respectively. While both algorithms indicate an asymptotic quadratic runtime in practice, the constant differs by more than an order of magnitude. In fact, there were 15 instances that could not be solved by glpk within a time limit of half an hour while all benchmarks where solved by CPLEX.

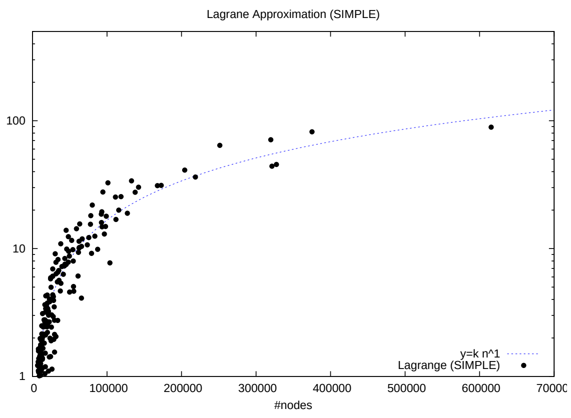
The performance of the algorithms based on Lagrangian relaxation largely depends on the particular Lagrangian heuristic. At most 20 iterations have been used for the Newton approximation. The strategy SIMPLE is very fast and shows an asymptotic linear runtime in practice ($|V|^{1.02}$). Note that the worst-case performance is determined by the max-flow algorithm, which is in the order of $\mathcal{O}(|V||E|\log|V|)$. The more sophisticated Lagrangian heuristics REGION and MINCUT show quadratic behavior, but with a much larger constant.

A comparison among all algorithms is given in Figure 4.10 (f). Note that the CPLEX based algorithm is among the fastest techniques while the GLPK solver performs worst compared to the rest of the field. The MINCUT heuristic delivers solutions that are very close to the optimum, but for significant computational costs.

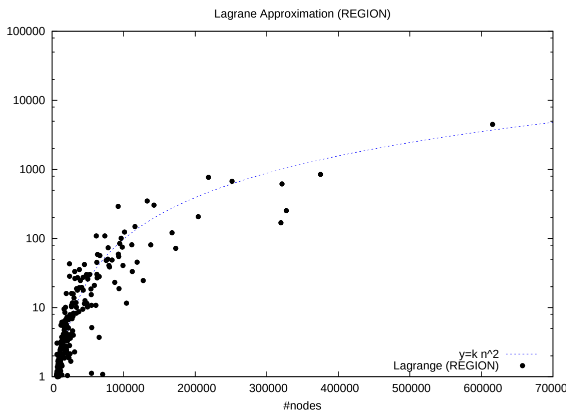
Apart from Lagrangian heuristics, the proposed relaxation is very tempting for two more reasons. First, it provides us with bounds that can be used to give a provable certificate on the quality of solutions. Second, those bounds are valuable for enumeration schemes such as branch-&-bound in order to prune the search space more effectively. One last advantage we want to point out is that Lagrangian heuristics lead to *progressive* algorithms that deliver quickly feasible solutions which are gradually improved as the algorithm proceeds. Thus, we can effectively trade compile time for code quality, which is a very appealing property for optimization algorithms.

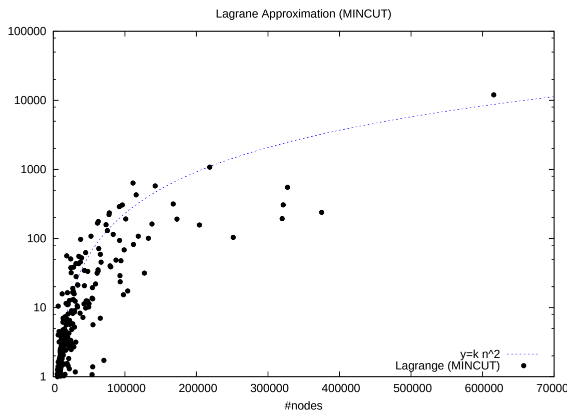
[2]`http://www.gnu.org/software/glpk`



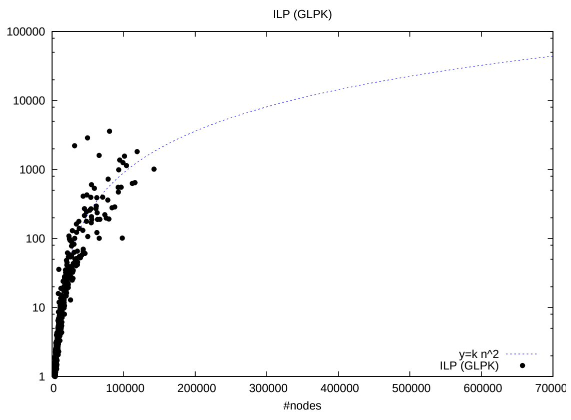


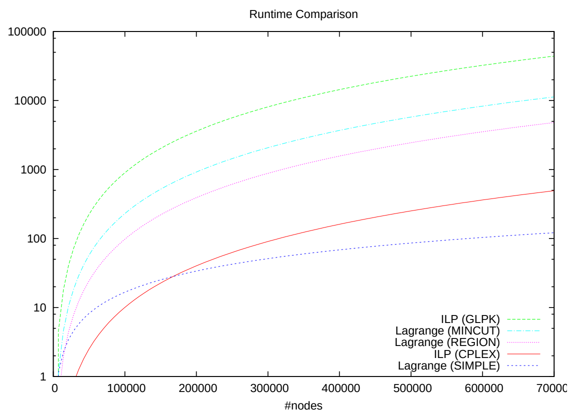
(a) (b)





(c) (d)





(e) (f)

Figure 4.10: Runtime comparison for the various CMC algorithms.

4.7 Related Work

Spilling has been mainly considered in the context of a particular register allocation scheme. Thus, the proposed strategies and metrics are often based on the particular program representation that is used by the allocation algorithm, e.g., live intervals in the case of linear scan allocators or interference graphs for graph coloring based techniques. Usually, spilling is invoked only if the allocation algorithm fails, often leading to backtracking or iteration of the whole process.

Apart from iterative models, comprehensive optimal formulations have been proposed that consider spilling as an inherent subproblem [GW96b, FW02]. In these approaches, all aspects of register allocation such as spilling, coalescing, rematerialization, live range splitting, or architectural irregularities are modeled within a unified framework, e.g., ORA [GW96b] uses an ILP solver to obtain optimal or near optimal solutions. However, these approaches do not consider spilling as a separate optimization problem.

One of the first heuristic algorithms directly solving a simple variant of the spilling problem has been proposed by Belady [Bel66] in a slightly different context: paging of virtual memory with write back. The problem basically corresponds to unweighted spilling for basic blocks. Belady's algorithm is a furthest-first heuristic, i.e., the variable that is used furthest in the future is evicted. Recently, Guo et al. [GGP04] empirically showed that this simple heuristic can be both efficient and effective, especially for large basic blocks.

A generalization of Belady's algorithm to the scope of a whole function is presented by Hack [Hac07]. The heuristic is applied individually to each basic block. The notion of "furthest first" is extended to the global scope by recursively computing the minimum over all successor blocks. In a separate step, the partial solutions computed by applying the heuristic locally are combined to obtain a solution for the whole function. Let $N(\ell, v)$ denote the applied distance metric for variable v and label ℓ, the authors propose the following generalization:

$$N'(\ell, v) = \begin{cases} \infty, & \text{if } v \text{ is not live at } \ell \\ 1 + min_{\ell' \in succs(\ell)} N(\ell', v), & \text{otherwise.} \end{cases}$$

$$N(\ell, v) = \begin{cases} 0, & \text{if } v \text{ is used at } \ell \\ N'(\ell, v), & \text{otherwise.} \end{cases}$$

This approach can be implemented efficiently for SSA form. We include an implementation of their method in our experiments in Section 4.6.

Another approach for the local spilling problem is given by Hsu et al. [HFG89]. The problem is mapped to the task of finding a shortest path within a weighted DAG that contains nodes for each instruction and each configuration that may occur at this node. Edges correspond to transition costs. As the graph grows exponentially with the number of variables and registers, the authors propose pruning rules that improve the performance of their algorithm, allowing them to solve blocks with about up to 100 nodes.

Important theoretic insight into *local* register allocation has been contributed by Farach [FL98, FCL00]. Similar to our approach, Farach considers both a simplified version where stores are disregarded (weighted caching) and a more complex variant where both spills and re-loads are minimized. For the first problem, Farach presents an *ILP* with the consecutive ones property. Thus, there is an equivalent minimum cost network flow problem that can be solved in polynomial time. Variables are spilled at the very beginning and reloaded at the very end of so-called *value ranges*, which are maximal sequences of program points where a live variable can be evicted. A unit of flow in the network corresponds to a surplus of live variables on a node. There are backward arcs among nodes on which the flow is unrestricted. Forward arcs for each value range account for spilling costs. For the more complex variant where store costs are included in the cost model, Farach proves NP completeness using a reduction from the set cover problem. There is a multi-commodity flow formulation for the harder model. Farach also presents an efficient 2-approximation using a variant of the furthest-first heuristic.

Work presented so far did only consider the spilling problem for straight-line code segments. The first relevant complexity result for global unweighted spilling in the spill-everywhere model is given by Yannakakis et al. [YG87]. The authors show that determining the maximum k-colorable subgraph of a chordal graph is NP complete. While there is a polynomial time algorithm for fixed k, its time complexity is exponential in k, which is prohibitive for most practical applications.

Pereira et al. propose an efficient linear-time heuristic based on enumeration of maximal cliques in interference graphs. [PP05]. While this is a hard problem in general, it can be solved efficiently for chordal graphs. Their heuristic iteratively removes nodes that appear in most of the cliques until the graph is k-colorable. Complexity results and heuristics for various variants of the spilling problem in the spill-everywhere model are also given by Bouchez et al. [BDR07].

Few papers are about the more-general load-store optimization problem that is considered in this work. Probably the most important piece of related work is given by Appel and George [AG01]. They present an AMPL[3] model to describe and generate an ILP formulation that solves the global weighted spilling problem. Additionally, as they focus on X86 architectures, they also include address mode selection for CISC instruction that optionally obtain their operand(s) directly from a specified memory location. For each variable live at a particular program point, four binary decision variables are introduced that encode the location (register or main memory) of the variable before and after the label. Constraints ensure the consistency among those variables and proper resource allocation. The most important result of their work is the insight that the spilling problem can be solved efficiently in practice and that the decomposition from register allocation does not significantly degrade the overall allocation quality.

One last work we want to point out even though it addresses a much wider problem is presented by Koes and Goldstein [KG06]. The authors propose a reduction of the register allocation problem including spilling, assignment, rematerialization, and limited support for instruction selection to multicommodity flows. As in this work, the authors propose Lagrangian relaxation combined with greedy heuristics to obtain a progressive algorithm. Global register allocation cannot be expressed with multicommodity flows

[3]`http://www.ampl.com`

and is modeled using so-called split- and merge-nodes at block boundaries with special semantics. The authors hereby solve a significantly harder problem: instead of deciding if a variable should be held in memory or register at a particular program point, they explicitly model every possible assignment to a particular register along with all the possible storage class transitions.

Short Summary The results show that traditional heuristics perform sufficiently well when the number of machine registers is large, but leave significant potential for improvement on architectures with few registers. The separation of spilling from allocation and coalescing is favorable in several respects. First, it allows a separation of concerns, thereby simplifying the design and implementation of allocation/spilling frameworks for compilers. Second, it allows us to take advantage of efficient algorithms for allocation and coalescing that benefit from the chordality of interference graphs for programs in SSA form. Empirical results show that optimal spill code placement lead to performance improvements of more than 15% on average for machines with few registers.

5 Conclusions

Embedded systems usually operate in very restricted environments and have to meet strict performance and energy requirements. Thus, compilers have to generate high-quality code that effectively utilizes the resources of the target architecture.

In this thesis, we have considered two important subproblems for embedded code generators: instruction selection and spilling. For both techniques, SSA form proved to be a valuable tool. This encourages a compiler design where SSA form is maintained until register allocation. Also, we advocate the use of general combinatorial optimization problems instead of problem-specific algorithms. This allows to draw upon existing algorithms and theoretical insights. At the same time, these algorithms can be shared for various applications.

Instruction Selection The first considered code generation problem, instruction selection for irregular architectures, still imposes considerable challenges in spite of the remarkable amount of attention it has received in the past. First, the limited scope of most standard approaches is leading to suboptimal code not accounting for the computational flow of a whole function. Second, many architectural features commonly found in the area of embedded systems cannot be expressed using well-known techniques such as tree pattern or *DAG* matching.

We present a generalization to *PBQP* based instruction selection that can cope with complex *DAG* patterns with multiple results. The approach has been implemented in *LLVM* for an embedded ARMv5 architecture. Extensive experiments show improvements of up to 57% for typical DSP code and up to 10% for MiBench and SPECINT 2000 benchmarks (5% on average). Using a heuristic *PBQP* solver, all benchmarks could be compiled within less than half a minute, with about 99.83% of all problem instances solved to optimality. The comparison of the *PBQP* instruction selector with a linearization to integer linear programming confirms the efficiency and effectiveness of instruction selection based on *PBQP* solvers.

Spilling SSA form allows us to consider spilling as an independent optimization problem. Our experiments show that traditional heuristics perform sufficiently well when the number of machine registers is large, but leave significant potential for improvement on architectures with few registers. The separation of spilling from allocation and coalescing is favorable in several respects. First, it allows a separation of concerns, thereby simplifying the problem at hand. Second, it allows us to take advantage of efficient algorithms for allocation and coalescing that benefit from the chordality of interference graphs for programs in SSA form.

We present a reduction of the spilling problem in the load-store optimization model to a well-defined combinatorial framework: constrained min-cut problems. The proposed model is interesting as it is based on a generic network flow substructure. We present an ILP formulation and a Lagrange relaxation that takes advantage of these properties and thus allows us to solve the problem using generic max-flow algorithms. Empirical results show that optimal solutions are feasible, even for large functions, and lead to performance improvements of more than 15% on average for machines with few registers.

Bibliography

[AF05] Warren P. Adams and Richard J. Forrester. A simple recipe for concise mixed 0-1 linearizations. *Oper. Res. Lett.*, 33(1):55–61, 2005.

[AG01] Andrew W Appel and Lal George. Optimal spilling for cisc machines with few registers. In *International Conference on Programming Languages Design and Implementation*, pages 243 – 253. ACM Press, 2001.

[AJ76] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *J. ACM*, 23(3):488–501, 1976.

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[BCH$^+$02] Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. In *PLDI*, pages 25–32, 2002.

[BÇPP98] R. E. Burkard, E. Çela, P. M. Pardalos, and L. S. Pitsoulis. The quadratic assignment problem. Technical Report 126, Graz University of Technology, 1998.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428 – 455, 1994.

[BDB90] A. Balachandran, D. M. Dhamdhere, and S. Biswas. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages*, 15(3):127–140, 1990.

[BDMS05] Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial-time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*. ACM Press, 2005.

[BDR07] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of spill everywhere under SSA form. In Santosh Pande and Zhiyuan Li, editors, *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07), San Diego, California, USA, June 13-15, 2007*, pages 103–112. ACM, 2007.

[Bel66] Laszlo A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Syst. J.*, 5:78–101, 1966.

[BK04] Andrzej Bednarski and Christoph W. Keßler. Exploiting symmetries for optimal integrated code generation. In Hamid R. Arabnia, Minyi Guo, and Laurence Tianruo Yang, editors, *Proceedings of the International Conference on Embedded Systems and Applications, ESA '04 & Proceedings of the International Conference on VLSI, VLSI '04, June 21-24, 2004, Las Vegas, Nevada, USA*, pages 83–92. CSREA Press, 2004.

[CCK97] C-M Chang, C-M Chen, and C-T King. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors, July 18 1997.

[CEL98] F. CELA. *The Quadratic Assignment Problem: Theory and Algorithms.* Kluwer, Massachessets, USA, 1998.

[CFR$^+$91a] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CFR$^+$91b] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, October 1991.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, 1990.

[Cha82] G J Chaitin. Register allocation and spilling via graph coloring. *Symposium on Compiler Construction*, 17(6):98 – 105, 1982.

[Chv83] V. Chvátal. *Linear Programming.* W. H. Freeman and Company, New York, 1983.

[CP99] J. Clausen and M. Perregaard. On the best search strategy in parallel branch-and-bound - best-first-search vs. lazy depth-first-search. *Annals of OR*, (90):1–17, 1999.

[Dir61] G A Dirac. On rigid circuit graphs. In *Abhandlungen aus dem Mathematischen Seminar der Universiat Hamburg*, volume 25, pages 71 – 75. University of Hamburg, 1961.

[DJP$^+$94] Elias Dahlhaus, David S. Johnson, Christos H. Papadimitriou, P. D. Seymour, and Mihalis Yannakakis. The complexity of multiterminal cuts. *SIAM J. Comput.*, 23(4):864–894, 1994.

[DR06] João Dias and Norman Ramsey. Converting intermediate code to assembly code using declarative machine descriptions. In Alan Mycroft and Andreas Zeller, editors, *CC*, volume 3923 of *Lecture Notes in Computer Science*, pages 217–231. Springer, 2006.

[EBS$^+$08] Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. Generalized instruction selection using SSA -graphs. In Krisztián Flautner and John Regehr, editors, *Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08), Tucson, AZ, USA, June 12-13, 2008*, pages 31–40. ACM, 2008.

[ECG06] M. Anton Ertl, Kevin Casey, and David Gregg. Fast and flexible instruction selection with on-demand tree-parsing automata. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 52–60, New York, NY, USA, 2006. ACM Press.

[Eck03] Erik Eckstein. Code Optimization for Digital Signal Processors. *PhD Thesis. TU Wien*, November 2003.

[EKS03] Erik Eckstein, Oliver König, and Bernhard Scholz. Code Instruction Selection Based on SSA-Graphs. In Andreas Krall, editor, *SCOPES*, volume 2826 of *Lecture Notes in Computer Science*, pages 49–65. Springer, 2003.

[Ert99] M. Anton Ertl. Optimal Code Selection in DAGs. In *Principles of Programming Languages (POPL '99)*, 1999.

[ES03] Erik Eckstein and Bernhard Scholz. Addressing mode selection. In *CGO*, pages 337–346. IEEE Computer Society, 2003.

[FCL00] Farach-Colton and Liberatore. On local register allocation. *ALGORITHMS: Journal of Algorithms*, 37, 2000.

[FD62] Jr. L.R. Ford and D.R.Fulkerson. *Flows in Networks.* Princeton University Press, Princeton, N.J., 1962.

[FFY05] Joseph A. Fisher, Paolo Faraboschi, and Clifford Young. *Embedded computing: a VLIW approach to architecture, compilers and tools.* Morgan Kaufmann Publishers, 2005.

[FHP92a] C. Fraser, R. Henry, and T. Proebsting. BURG – Fast Optimal Instruction Selection and Tree Parsing. *ACM SIGPLAN Notices*, 27(4):68–76, April 1992.

[FHP92b] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[FL98] Martin Farach and Vincenzo Liberatore. On local register allocation. In *SODA*, pages 564–573, 1998.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[FW02] Changqing Fu and Kent D. Wilken. A faster optimal register allocator. In *MICRO*, pages 245–256. ACM/IEEE, 2002.

[GA96] Lal George and Andrew W Appel. Iterated register coalescing. *Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):300 – 324, 1996.

[GCC] GCC Website. `http://gcc.gnu.org`.

[GGP04] Jia Guo, Maria Jesus Garzaran, and David A. Padua. The power of belady's algorithm in register allocation for long basic blocks. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing, (16th LCPC'03)*, volume 2958 of *Lecture Notes in Computer Science (LNCS)*, pages 374–390. Springer-Verlag (New York), College Station, Texas, USA, October 2003, Revised Papers 2004.

[GH88] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. In *1988 International Conference on Supercomputing (2nd ICS'88)*, pages 442–452, St. Malo, France, July 1988. ACM Press.

[GH07] Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. In Shriram Krishnamurthi and Martin Odersky, editors, *Compiler Construction, 16th International Conference, CC 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 26-30, 2007, Proceedings*, volume 4420 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2007.

[GJ79] M. Garey and D. Johnson. Computers and interactability: A guide to the theory of NP-completeness, 1979.

[GLM$^+$06] J. Guo, T. Limberg, E. Matus, B. Mennenga, R. Klemm, and G. Fettweis. Code generation for STA architecture. In *Proc. of the 12th European Conference on Parallel Computing (Euro-Par'06)*. Springer LNCS, 2006.

[GSW95] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, 1995.

[GW96a] Goodwin and Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *SOFTPREX: Software–Practice and Experience*, 26, 1996.

[GW96b] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software Practice and Experience*, 26(8):929–965, August 1996.

[Hac07] Sebastian Hack. *Register Allocation for Programs in SSA Form*. PhD thesis, Universität Karlsruhe, October 2007.

[HFG89] W. Hsu, C. Fischer, and J. Goodman. On the minimization of loads/stores in local register allocation. *IEEE Trans. on Softw. Eng.*, 15(10):1252, October 1989.

[HG06] Sebastian Hack and Gerhard Goos. Optimal register allocation for ssa-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, May 2006.

[HGG06] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register Allocation for Programs in SSA-Form. In Andreas Zeller and Alan Mycroft, editors, *Compiler Construction 2006*, volume 3923, pages 247–262. Springer, March 2006.

[HKS03] Ulrich Hirnschrott, Andreas Krall, and Bernhard Scholz. Graph coloring vs. optimal register allocation for optimizing compilers. In László Böszörményi and Peter Schojer, editors, *Modular Programming Languages, Joint Modular Languages Conference, JMLC 2003, Klagenfurt, Austria, August 25-27, 2003, Proceedings*, volume 2789 of *Lecture Notes in Computer Science*, pages 202–213. Springer, 2003.

[HS06] Lang Hames and Bernhard Scholz. Nearly optimal register allocation with PBQP. In David E. Lightfoot and Clemens A. Szyperski, editors, *JMLC*, volume 4228 of *Lecture Notes in Computer Science*, pages 346–361. Springer, 2006.

[Jak04] Hannes Jakschitsch. "Befehlsauswahl auf SSA-Graphen". Master's thesis, Fakultät für Informatik, Universität Karlsruhe (TH),Germany, 2004.

[KB57] T. C. Koopmans and M. J. Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 25:53–76, 1957.

[KB06] Christoph W. Keßler and Andrzej Bednarski. Optimal integrated code generation for VLIW architectures. *Concurrency and Computation: Practice and Experience*, 18(11):1353–1390, 2006.

[KG06] David Ryan Koes and Seth Copen Goldstein. A global progressive register allocator. *ACM SIGPLAN Notices*, 41(6):204–215, June 2006.

[KR95] Christoph W. Keßler and Thomas Rauber. Generating optimal contiguous evaluations for expression DAGs. *Comput. Lang*, 21(2):113–127, 1995.

[KTJR05] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *IEEE Computer*, 38(11):32–38, 2005.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, CGO 2004*, pages 75–88, Palo Alto, CA, March 2004. IEEE Computer Society.

[LB00] Rainer Leupers and Steven Bashford. Graph-based code selection techniques for embedded processors. *ACM Transactions on Design Automation of Electronic Systems.*, 5(4):794–814, 2000.

[LDKT95] Stan Liao, Srinivas Devadas, Kurt Keutzer, and Steve Tjiang. Instruction selection using binate covering for code size optimization. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 393–399, 1995.

[lm] MiBench Website. `http://www.eecs.umich.edu/mibench/`.

[LT79] Thomas Lengauer and Robert E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[MP94] F. Malucelli and D. Pretolani. Quadratic semi-assignment problems on structured graphs. *Ricerca Operative*, 69:57–78, 1994.

[MP95] F. Malucelli and D. Pretolani. Lower bounds for the quadratic semi-assignment problem. *European Journal of Operational Research*, 83:365–375, 1995.

[NK97] Albert Nymeyer and Joost-Pieter Katoen. Code generation based on formal BURS therory and heuristic search. *Acta Inf.*, 34(8):597–635, 1997.

[NP98] Cindy Norris and Lori L. Pollock. Experiences with cooperating register allocation and instruction scheduling. *International Journal of Parallel Programming*, 26(2):241–283, 1998.

[ORA$^+$01] Guilherme Ottoni, Sandro Rigo, Guido Araujo, Subramanian Rajagopalan, and Sharad Malik. Optimal live range merge for address register allocation in embedded programs. *Lecture Notes in Computer Science*, 2027, 2001.

[PM98] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (7th PACT'98)*, pages 196–204, Paris, France, October 1998. IEEE Computer Society.

[PP05] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In Kwangkeun Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2005.

[PP08] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation by puzzle solving. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 216–226. ACM, 2008.

[Pro98] Todd A. Proebsting. Least-Cost Instruction Selection in DAGs is NP-Complete. *http://research.microsoft.com/˜ toddpro/papers/proof.htm*, 1998.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895 – 913, 1999.

[RD98] Norman Ramsey and Jack W. Davidson. Machine descriptions to build tools for embedded systems. *Lecture Notes in Computer Science*, 1474:176ff, 1998.

[RN03] Johan Runeson and Sven-Olof Nyström. Retargetable graph-coloring register allocation for irregular architectures. In Andreas Krall, editor, *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems, Vienna, Austria, April 2003*, volume 2826 of *ACM International Conference Proceeding Series*, pages 240–254, 2003.

[RT97] C. Roos and T. Terlaky. Advances in linear optimization, 1997.

[SBX08] Bernhard Scholz, Bernd Burgstaller, and Jingling Xue. Minimal placement of bank selection instructions for partitioned memory architectures. *ACM Trans. Embedded Comput. Syst*, 7(2), 2008.

[SE02a] Bernhard Scholz and Erik Eckstein. Register Allocation for Irregular Architectures. In *LCTES-SCOPES '02: Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*, pages 139–148, 2002.

[SE02b] Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. *ACM SIGPLAN Notices*, 37(7):139–148, July 2002.

[Set73] Ravi Sethi. Complete register allocation problems. In *5th annual ACM symposium on Theory of computing*, pages 182 – 195. ACM Press, 1973.

[SPC] SPEC2000 Website. `http://www.spec.org`.

[SRH04] Michael D. Smith, Norman Ramsey, and Glenn H. Holloway. A generalized algorithm for graph-coloring register allocation. In William Pugh and Craig Chambers, editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 277–288. ACM, 2004.

[SS07] Stefan Schäfer and Bernhard Scholz. Optimal chain rule placement for instruction selection based on SSA graphs. In *SCOPES '07: Proceedingsof the 10th international workshop on Software & compilers for embedded systems*, pages 91–100, Nice, France, 2007. ACM.

[Sto77] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, January 1977.

[SU70] Sethi and Ullman. The generation of optimal code for arithmetic expressions. *JACM: Journal of the ACM*, 17, 1970.

[SW97] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, 1997.

[SYL$^+$07] Hanno Scharwächter, Jonghee M. Yoon, Rainer Leupers, Yunheung Paek, Gerd Ascheid, and Heinrich Meyr. A code-generator generator for multi-output instructions. In Soonhoi Ha, Kiyoung Choi, Nikil D. Dutt, and Jürgen Teich, editors, *Proceedings of the 5th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2007, Salzburg, Austria, September 30 - October 3, 2007*, pages 131–136. ACM, 2007.

[Tur99] Jim Turley. Embedded processors by the numbers. *Embedded Systems Programming*, 1999.

[uVSM94] Vojin živojnović, Juan M. Velarde, Christian Schläger, and Heinrich Meyr. DSPSTONE: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing and Technology (ICSPAT'94)*, 1994.

[Vaz04] V. V. Vazirani. *Approximation Algorithms.* Springer, 2004.

[WGB94] Thomas Charles Wilson, Gary William Grewal, and Dilip K. Banerji. An ILP solution for simultaneous scheduling, allocation, and binding in multiple block synthesis. In *ICCD*, pages 581–586. IEEE Computer Society, 1994.

[YG87] Mihalis Yannakakis and Fanica Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, 24(2):133 – 137, 1987.