DISSERTATION

Code Optimizations for Digital Signal Processors

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

ao.Univ.Prof. Dipl.-Ing. Dr. Andreas Krall

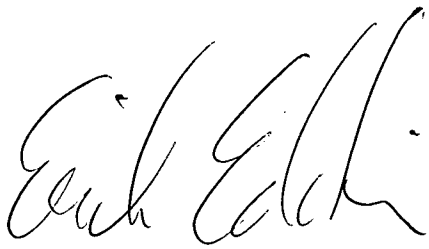
E185
Institut für Computersprachen

eingereicht an der Technischen Universität Wien

Fakultät für Technische Naturwissenschaften und Informatik

von

Dipl.-Ing. Erik Eckstein
9125255
Vöslauerstrasse 26/2/3
2500 Baden



Wien, am
5. November 2003

Kurzfassung

Digitale Signal Prozessoren (DSPs) spielen im Marktsegment der *Eingebetteten Computer Systeme* eine wichtige Rolle. Die Einsatzgebiete reichen von Mobiltelefonie, digitaler Bildverarbeitung bis hin zur Motorsteuerung. Die Anforderungen an die Signalprozessoren sind dabei meist hohe Rechenleistung bei niedrigem Stromverbrauch. Bis vor wenigen Jahren wurden DSPs in Assemblercode programmiert, da sonst die Anforderungen nicht zu erfüllen gewesen wären. Mittlerweile haben sowohl die Applikationen wie auch die Prozessorarchitekturen einen derart hohen Komplexitätsgrad erreicht, dass der Einsatz automatischer Übersetzer notwendig ist. Ein Übersetzer erzeugt aus einem Hochsprachenprogramm ein Maschinenprogramm, das auf dem DSP ausgeführt wird. Durch die Verwendung einer Hochsprache können die Software-Entwicklungskosten erheblich verringert werden. Entscheidend ist jedoch, dass der Übersetzer eine Code-Qualität erzeugt, die mit dem handoptimierten Assemblercode vergleichbar ist.

Die Übersetzer-Technologie wurde hauptsächlich für reguläre Prozessorarchitekturen entwickelt. Da DSP Architekturen viele Irregularitäten und applikationsspezifische Funktionalitäten aufweisen, können konventionelle Optimierungsmethoden des Übersetzerbaues nicht ohne weiteres für DSP-Übersetzer eingesetzt werden.

In dieser Dissertation werden Algorithmen vorgestellt, die drei wesentliche Problemstellungen im Übersetzerbau abdecken: Code-Generierung, Auswahl von Adressierungsmodi und Register-Allokation. Alle drei Algorithmen werden mit Hilfe von *Partitionierten Boolschen Quadratischen Problemen* (PBQP) beschrieben. Ein PBQP ist eine Art von quadratischem Zuordnungsproblem, das im Allgemeinen NP-vollständig ist und daher sehr schwierig zu lösen ist. Die Darstellung eines PBQP erfolgt als Kostenfunktion oder als PBQP-Graph, der mit Kostenvektoren und Kostenmatrizen annotiert ist. Es wird ein Lösungsverfahren präsentiert, das eine optimale Lösung berechnen kann, sofern der PBQP-Graph mit Hilfe von definierten Reduktionsregeln reduzierbar ist. Falls der Graph nicht reduzierbar ist, muss auf eine Heuristik zurückgegriffen werden. Da der Berechnungsaufwand nur linear mit der Anzahl der Knoten steigt, liefert der Lösungsalgorithmus meist sehr schnell ein Ergebnis.

In der Code-Generierung eines Übersetzers werden aus der Zwischendarstellung des zu übersetzenden Programms Instruktionen für die Zielarchitektur generiert. Traditionelle Algorithmen gehen dabei von einer baumartig strukturierten Zwischendarstellung aus. Durch einen gerichteten zyklischen Graphen kann der Datenfluss einer Funktion genauer beschrieben werden als durch einen Baum. Der vorgestellte Algorithmus bildet das Code-Generierungsproblem auf ein PBQP ab. Da die meisten PBQP-Graphen reduzierbar sind, findet der Lösungsalgorithmus in vielen Fällen die optimale Instruktionsreihenfolge.

Viele DSP-Architekturen besitzen spezielle Funktionseinheiten zur Adressgenerierung. Die Aufgabe des Übersetzers ist es, die Adressierungsmodi so auszuwählen, dass die Codegröße oder die Ausführungszeit des Programms minimiert wird. Die Beschreibung als PBQP erlaubt es verschiedenartige und irreguläre Adressierungsmodi zu modellieren. Aufgrund der Beschaffenheit der PBQP-Graphen kann für das Adressierungsmodi Problem in fast allen Fällen eine optimale Lösung berechnet werden.

Die Register-Allokation bildet eine meist große Anzahl von Programmvariablen auf eine beschränkte Anzahl von Prozessor-Registern ab. Die meisten Register-Allokations-Algorithmen basieren auf der Methode des Färbens von Interferenz-Graphen. Obwohl diese Methode für reguläre Architekturen sehr gute Ergebnisse liefert, hat sie den Nachteil, dass andere Optimierungsparameter, außer Register-Interferenzen, nur bedingt einfließen können. Mithilfe eines PBQPs ist es möglich irreguläre Optimierungsparameter genau zu beschreiben. Da jedoch die resultierenden PBQP-Graphen meist nicht reduzierbar sind, baut die Heuristik im PBQP-Lösungsalgorithmus auf einen Graph-Färbungsalgorithmus auf um die Vorteile beider Methoden zu vereinen.

Alle vorgestellten Algorithmen wurden in einen Übersetzer für einen kommerziellen DSP integriert. Anhand von typischen DSP Applikationen wurden sie mit existierenden Methoden verglichen. Dabei zeigte sich, dass durch die Verwendung des PBQP-Lösungsansatzes erhebliche Verbesserungen in allen drei Optimierungsbereichen erzielt wurden.

Abstract

Digital Signal Processors (DSPs) play an important role in the embedded systems market. The application areas range from mobile telecommunication, digital image processing to engine control. The requirements for signal processors are often high performance at low power consumption. In the past years DSPs were merely programmed in assembly language to meet the hard requirements. In recent times the complexity of both the DSP applications and the DSP architectures have increased dramatically. Therefore the use of compilers became necessary. A compiler translates a high level language into machine code, which can be executed on a DSP. The use of a high level language reduces the software development times significantly. The compiler has to generate a code quality which is comparable to hand-optimized assembly code.

Compiler technology was mainly developed for regular processor architectures. As DSP architectures contain many irregularities and application specific functions, conventional optimization methods can not be directly applied to DSP compilers.

In this work, algorithms are presented which cover three important compilation problems: code generation, addressing mode selection and register allocation. All three algorithms are described with the help of *partitioned boolean quadratic problems* (PBQPs). A PBQP is a kind of quadratic assignment problem, which is NP-complete in general and therefore it is hard to solve. A PBQP can be formulated as a cost function or as a PBQP-graph, which is annotated with cost vectors and cost matrices. The presented solver is able to calculate a optimal solution if the PBQP-graph is reducible with defined reduction rules. If the graph is not reducible, heuristics are used to obtain a solution. The computational complexity is linear with the number of nodes in the PBQP-graph. Therefore the solver yields a solution in short time.

The code generation phase in a compiler translates the intermediate representation of a program to instructions for the target architecture. Traditional methods assume a tree-like intermediate representation. But cyclic directed graphs can describe the data flow of a function more precisely. The presented algorithm maps the code generation problem to a PBQP. As most of the PBQP-graphs are reducible, the solver can find an optimal instruction selection in many cases.

Many DSP architectures provide functional units for generating addresses. The task of the compiler is to select addressing modes so that the execution time or code size of the program is minimized. The formulation as a PBQP allows modeling of various irregular addressing modes. The PBQP-graphs of the addressing mode selection problem are reducible in almost all cases, enabling the calculation of optimal results.

Register allocation maps a large number of program variables to a limited number of processor registers. Most register allocation algorithms are based on the method of coloring an interference graph. This method yields good results for regular ar-

chitectures, but it is difficult to model register constraints other than interferences. The PBQP enables the exact formulation of all kinds of irregular register constraints. As the resulting PBQP-graph are not reducible in many cases, the graph coloring heuristics are integrated into the PBQP solver. This combines the advantages of both methods.

All presented algorithms were integrated into a compiler for a commercial DSP. Typical DSP applications were used to compare the algorithms with existing methods. The results have shown that the PBQP method achieved significant improvements for all three optimization problems.

Acknowledgments

Many people have contributed to this work. Without the help of the people from university and the company, my family and my friends I couldn't have completed or even started writing this thesis.

First of all, I would like to thank Professor Andreas Krall, my thesis adviser. Andreas Krall motivated me for doing this work at all. He helped and inspired me since I started doing compiler research. Most of my compiler knowledge I owe to him. Many thanks go to Eduard Mehofer, my secondary adviser, for reviewing the thesis.

I'd like to thank all the people which I am glad to work with. Especially I want to thank Bernhard Scholz. We spent a lot of time together and many ideas were born in countless fruitful discussions I had with him. Almost all of the research work contained in this thesis was done together with him. Bernhard is not only a great colleague but also a good friend who encouraged me in good and in bad times. Finally I thank him for reviewing this thesis.

Many thanks go to my colleagues at Atair Software, especially to Oliver König, Christian Pirker, Albrecht Kadlec, Alexander Wolf, Paul Wögerer and Sylvain Lelait. Being able to work together with this great team was the indispensable base for doing compiler research – and it is a lot of fun. Special thanks to Oliver for implementing the SSA-graph matcher. I like to thank Anton Ertl from the University of Technology in Vienna for his comments on pattern matching. I'd also like to thank R. Nigel Horspool from the University of Victoria for reviewing the thesis and for giving valuable comments.

Finally I give my wholehearted thanks to my family. I'd like to thank my beloved parents, Elsa Eckstein and Kurt Eckstein for supporting me throughout my life and enabling me to go to the university.

Especially I want to thank Heidi Rotteneder for being the sunshine in my life. She encouraged and supported me all over the time. Thanks to my two little children, Rahel and Esther, for bringing so much joy into my life.

Contents

1. Introduction

1.1. Overview

In recent years embedded systems have become very popular in the electronics industry. An embedded system is a computer system, consisting of hardware and software, which is part of a larger device.

The requirements of an embedded system are different from a general-purpose computer system, like a workstation or server. Many embedded systems operate inside a mobile device. Therefore low power consumption is an important aspect. On the other hand embedded systems often have to meet real-time constraints. Often the design of embedded systems is driven by the goal of reducing manufacturing costs, because they are produced in large quantities.

As a result, embedded systems usually have smaller memory sizes and run at a lower clock rate than general-purpose computers.

Digital signal processors (DSPs) play an important role in the embedded system market. DSPs evolved from custom digital hardware solutions which replaced analog signal processing circuits. The custom hardware was too error-prone and too inflexible. At this point DSPs provided a programmable alternative to expensive custom hardware. The application area of DSPs is computationally intensive signal processing algorithms. Since DSPs are often used in mobile devices they provide high computational performance with very low power consumption.

DSPs are used in many embedded devices like cellular phones, digital cameras, video game consoles, engine controls, medical equipments, network devices and many more. Two of the most prominent signal processing applications are GSM speech coding and decoding, and MP3 decoding.

1.2. DSP Architectures

Nearly all architectural features in a DSP arise from the needs of signal processing algorithms. In this sense DSPs are application specific processors. As in the early days of digital signal processing DSPs were exclusively used for numerical signal processing, today's architectures are more general. The borders between micro-controllers and signal processors are diminishing. New DSP architectures include micro-controller functions and vice versa.

Although there are many different DSPs on the market today, they have some properties and features in common:

- *fixed point arithmetic:* Most DSPs do not have floating point units. Values are represented as fixed point values in the range from -1 to 1. Fixed point arithmetic is very similar to integer arithmetic. Minor differences are encountered with multiplications and divisions.

- *40 bit accumulator registers:* With the help of 40 bit registers the fixed point value range of -1 to 1 is extended by 8 bits to -256 to 256. It is used to hold the overflow when performing accumulation loops.
- *explicit instruction level parallelism (VLIW):* To increase computational performance without having to push up the clock frequency, DSP architectures allow the execution of multiple instructions in parallel.
- *single instruction, multiple data support (SIMD):* To overcome the problem of large codes sizes in VLIW architectures, instructions are provided which can manipulate multiple data with a single instruction.
- *multiply-accumulate support:* The vector dot product is a very common operation in DSP applications. Therefore all DSP architectures incorporate a multiply-accumulate function, which enables the execution of a multiplication and addition in minimal time.
- *high memory bandwidth:* To cope with the high computational ability of a DSP it is necessary to fetch many operands from memory in short time.
- *memory spaces:* Many DSPs solve the memory-bandwidth-problem by implementing multiple memory spaces, which can be accessed in parallel.
- *addressing modes:* Some addressing modes allow the address registers to be updated and modified without any overhead. Therefore no explicit address arithmetic is necessary when accessing memory in a loop.
- *modulo addressing:* This kind of addressing mode allows implementation of circular buffers without any overhead.
- *bit reverse addressing:* For efficient implementations of fast Fourier transformations (FFT), which are very important in signal processing algorithms, DSPs provide an addressing mode for accessing buffers in a butterfly pattern. This is achieved by reversing the bit order of an incrementing address pointer.
- *zero overhead loops:* Most DSPs provide facilities to implement loops without any branching overhead. In most architectures special loop instructions perform this task.
- *application specific functions:* Dedicated instructions and registers are used to implement often used signal processing operations. An example is a decision back-trace register to implement Viterbi decoders.
- *user responsible restrictions:* Usually DSP hardware does not provide checks on the validity of instructions and operands. All hardware restrictions must be ensured by the user, this means by the assembler programmer or the compiler.

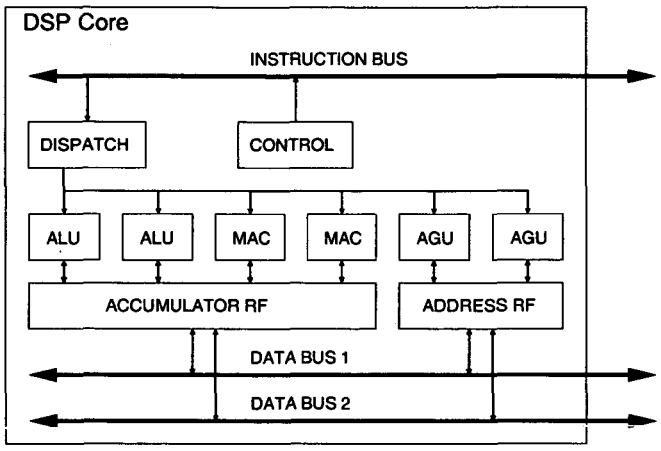


Figure 1.1.: Functional units of a DSP core

Figure 1.1 shows a block diagram of the functional units of a typical DSP. The instructions are fetched from the instruction bus and dispatched to the functional units. In this example the core contains two arithmetic units (ALU), two multiply-accumulate units (MAC) and two address generation units (AGU). The instruction stream contains one instruction for each unit (VLIW-architecture) so the units can operate in parallel. The units have access to the register files. The accumulator register file contains accumulators (e.g. 40 bits wide) to hold the calculation results. The address register file contains registers for generating addresses on the data busses. There are multiple data busses, e.g. one bus for each memory space, to increase the memory bandwidth of the system.

A DSP core is packed together with memory components, input-output (IO) components and other peripheral components on an integrated circuit. Large "system on a chip" designs may include multiple DSP cores together with other cores, like micro-controllers.

1.3. Code Generation

Code generation for DSPs started by programming applications in assembler. Early DSPs had very limited resources and good compilers were not available. Even today the heavily executed loop kernels are implemented in assembly language to achieve the required performance.

Nowadays most of signal processing software development is done with a high level language, in particular the C-language. The change from assembly language to C happened because of the following reasons: (1) DSP architectures got too complex for

hand coding, (2) the signal processing applications got very large, (3) time to market is reduced by using a high level language, (4) migration to a different architecture is eased with a high level language.

Compiler technology was mainly developed for RISC systems, which are very orthogonal. However, compilers for signal processors have to cope with irregularities:

- Fixed point arithmetic is not naturally expressible in C. Therefore C programs emulate fixed point calculations with integer arithmetic. The compiler must match the integer operations to the 40 bit fixed point arithmetic of the DSP.
- The compiler must handle all hardware restrictions of the DSP. The restrictions must be mapped to the optimization models in the compiler.
- Irregular instruction sets and register sets require flexible and configurable cost models in the compiler.
- DSP specific functions are not expressible in C, for example complex addressing modes, explicit memory spaces or application specific functions.

In addition to this requirements, the compiler must produce a code quality which is near to hand coded assembly language. Otherwise the code will not meet the constraints imposed by limited hardware resources.

Another aspect where code generation for embedded systems differs from RISC code generation is that code size is an important factor. The code size directly relates to the size of the silicon and to the power consumption, which is important for mobile devices. For this reason it must be possible to configure the compiler optimizations to either optimize for minimal execution time or to optimize for minimal code size. In signal processing applications the heavily executed loop kernels must be optimized for minimal execution time, the remaining code must be optimized for minimal code size.

1.4. Contribution of this Work

This work presents an optimization framework to overcome the difficulties in code generation for DSP architectures. It allows us to generate exact models for irregular optimization problems. Although most of the problems are NP-complete we present an algorithm, which yields near optimal results. It is based on *partitioned boolean quadratic problems* (PBQPs).

The PBQP is an optimization problem which is similar to quadratic assignment problems used in operations research. We present a solver for the PBQP which runs in near linear time for a certain subclass of problems. For general general PBQPs a solver based on heuristics is used.

In this work we introduce three optimization problems where we employ the PBQP solver: code selection, addressing mode selection and register allocation. We show the mapping of the optimization problems to the PBQP and demonstrate how to model architecture specific constraints. The optimizers based on PBQP were implemented into an DSP production compiler which was used to obtain experimental results. The experiments show that the optimizers based on PBQP yield better results than traditional approaches.

In Chapter 2 we present the problems of code selection, addressing mode selection and register allocation. A survey of existing work on these optimization problems and quadratic assignment problems is given.

In Chapter 3 we introduce the PBQP and show that it is NP-complete. The PBQP can either be formulated as a quadratic equation or as a graph – the *PBQP-graph*. We present an optimal solver, which can derive a solution for the subclass of reducible PBQP-graphs. For non-reducible PBQP-graphs a general solver is used, which implements a heuristic.

The *code selection problem* is addressed in Chapter 4. Code selection is performed by matching a graph, which represents the statements of the input program. The PBQP approach allows matching of a SSA-graph (which can be directly used as PBQP-graph), which includes all statements of a function. Even cyclic dependencies in the SSA-graph can be handled by the PBQP matcher. It is shown how to generate the PBQP formulation out of the grammar definition used by traditional tree pattern matchers.

In Chapter 5 the problem of *addressing mode selection* (AMS) is presented. The goal of AMS is to select addressing modes in the code instructions. Examples of widely used addressing modes are listed and it is demonstrated how to model them in the PBQP. With AMS, the PBQP-graph is derived from the control flow graph (CFG) of the input program.

The third optimization based on PBQP is introduced in Chapter 6. Various kinds of constraints are introduced, which are imposed by the register allocation problem in general and by irregular architecture features. The constraints are formulated as cost functions which are used to define the PBQP. The PBQP-graph is derived from the interference graph by adding edges which describe additional constraints.

Finally Chapter 7 concludes this work by comparing the three PBQP based optimization algorithms.

2. Related Work

2.1. Overview

This work presents three applications of optimizations based on PBQP: SSA-graph matching, addressing mode selection and register allocation. They were first presented in [16, 18, 60]. All three applications can be assigned to the class of *back-end* optimizations.

In a compiler, the back-end optimizations are performed after high level optimizations to generate machine code from the high level intermediate representation (IR) of the program. High level optimizations [3, 51] are important for improving the result of the generated code. They include dead code elimination, constant propagation, function inlining, common subexpression elimination, arithmetic simplifications and strength reduction. An important class of high level optimizations is loop transformations [4], which are often used in DSP compilers to optimize numerical code. Loop transformations can rearrange the statements in a loop nest to improve the parallelism of the code.

Traditionally there are three major building blocks in a compiler back-end: (1) the code generator, (2) the scheduler and (3) the register allocator. As this work is focused on back-end optimizations for embedded systems and especially for DSPs, we add a fourth important building part to the compiler back-end – the addressing code optimization (see Figure 2.1).

The execution order of the back-end optimizations may vary. Only the code generator is always performed first, because it represents the interface from the high level intermediate language to the low level intermediate language. The order of scheduling, addressing code optimization and register allocation imposes a phase ordering problem. In highly optimizing compilers these phases are performed multiple times.

In this work no attention is given to the scheduling part of the back-end optimizations. We used existing scheduling algorithms [28, 41, 58] and it turned out that they work well for our DSP compiler back-end. Therefore we did not investigate using a PBQP method for the scheduling optimizations in the compiler.

In the following sections we describe existing approaches to code generation, ad-

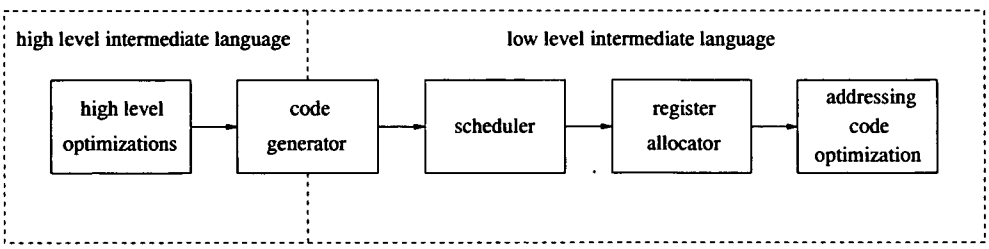


Figure 2.1.: Overview of a compiler

dressing code optimization and register allocation, and the relation to our work.

2.2. Code Generation

The code generator is the interface between the high level IR and the low level IR. It maps the high level IR statements to low level IR statements, which are strongly related to the machine instructions of the target architecture. Therefore code generation is also referred to as *instruction selection*. There are several approaches for code generation. Most approaches rely on a tree-like high level IR. The tree may be explicitly available as a graph structure or linearized in a string of symbols. Usually the unit of translation is a statement which represents a single data flow tree (DFT). The program to be compiled consists of many statements, so it is a collection of multiple DFTs. The nodes of a DFT are called *operators* and represent the expressions of the program.

Modern code generators are built automatically from a machine description. The path from the machine description to the generated code involves two abstraction layers, which are shown in Figure 2.2. The code generator-generator builds the code generator from the machine description. This happens at *compile-compile time*, i.e. when the compiler is built. The code generator itself reads the input program and generates the code. This process runs at *compile time*.

Graham and Glanville [30] first identified the needs of a systematic method for building code generators. The method should have the same properties as methods for table-driven syntax analysis: modular, provable correct and easy to use. Graham and Glanville introduced a method which is very similar to a $LR(1)$ parser. The differences are that the grammar for code generation is ambiguous and that a parse error would indicate a compiler bug. The format of the IR is a parenthesis-free prefix notation, which expresses the data flow trees.

A shift-reduce algorithm is used to parse the IR. The parser consists of a state S, a stack q and two tables. The NEXT-table is the state transition table, which selects

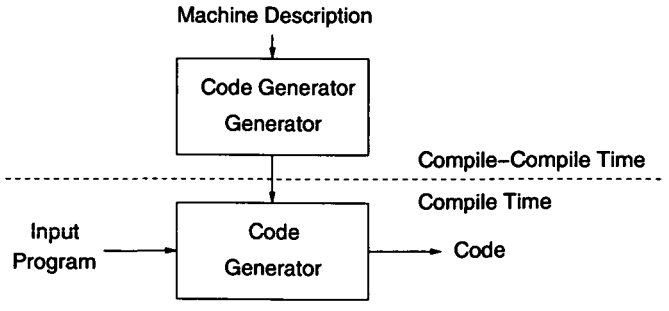


Figure 2.2.: Abstraction layers of code generation

a parser state depending on the current state and the next input symbol from the IR. The ACTION-table selects the action *shift*, *reduce*, *accept* or *error* depending on the state and the next input symbol. The *shift*-action pushes the next input symbol and the state onto the stack, whereas the *reduce*-action removes a whole right side of a grammar rule from the stack. At the *reduce*-actions, the code generation template of the reduced rule is emitted. The *accept*- and *error*-actions are the final states of the parser.

As the grammar is ambiguous conflicts may occur, which are solved by using a simple heuristic. For shift-reduce conflicts the parser performs the shift. This prefers "powerful" instructions, which map to larger parts of the IR. Reduce-reduce conflicts are either resolved by additional semantic constraints or by sorting the instructions according to a certain cost criteria.

A Graham-Glanville style code generator already has significant advantages compared to hand-crafted code generators: it produces good code, it is fast and it is automatically generated from a machine description – the grammar. Unfortunately the results are not optimal, because heuristics are used to resolve the ambiguities of the grammar. In addition the parsing is done in a left-to-right fashion. Therefore the selection for a left operand of an operator is always independent of the selection for the right operand.

Tree pattern matchers, based on the BURS (bottom-up rewrite systems) theory overcome the limitations of the Graham-Glanville style code generators. They were first introduced by Pelegri-Llopart and Graham [54].

A tree pattern matcher is defined by an ambiguous tree grammar. A reduction rule in the grammar describes a tree which is matched with a subtree in the DFT and replaced with the resulting non-terminal. Such rules are called *pattern rules* and are of the form `nt` $\rightarrow$ ***`pattern`***, where the pattern contains a tree with non-terminals and terminals. The terminals are operators in the DFT. In addition to pattern rules, a grammar may contain *chain rules* (of the form `nt1` $\rightarrow$ `nt2`) which allow transitions between non-terminals. Both types of rules have an associated cost value and a code generation action. For each reduction the code generation action inserts low level code in the output IR. The cost value represents the execution time or code size of the inserted code. As the grammar is ambiguous, there may be many possible matches for a tree. The task of the matcher is to select an *optimal match*, which is the cover of the IR tree so that the sum of all applied rule costs is a minimum.

A BURS-based code generator contains a state automaton and works in two phases: labeling and reducing. In the labeling phase the DFT is traversed bottom-up and the operators are labeled with the states. In the second phase the code is emitted in a top-down walk by using the state information. The BURS automaton is built by constructing all possible states from the grammar using dynamic programming at compile-compile time. A state corresponds to a set of items, containing three pieces of information: the derived non-terminal, the relative costs and the rule, which

generated the non-terminal. In an implementation an *itemset* is an array of {*rule*, *cost*} pairs, indexed by the non-terminal. The relative costs are the costs normalized to the minimum cost value in the itemset. Therefore each itemset contains at least one cost value of zero. In the state construction algorithm all state transitions for each state and operator combination are generated. This process is repeated until no more new states are derived.

The advantage of a BURS-based code generator is that it is very fast, because the hard work of building the state transition table is done at compile-compile time and not at compile time. But there are two problems which must be addressed. First the state transition table can get very large. If the number of states is n, a direct encoding of the state transition table of a binary operator would contain n^2 entries. As the number of states can be large (e.g. up to 1000 on a CISC machine [56]), this leads to large table sizes. Proebsting proposed several methods to reduce the table sizes [56, 57].

The second problem is that the cost values in the grammar must be available at compile-compile time. Therefore it is not possible to compute them dynamically in the code generator. For example a cost value could be depend on a constant value in an operator. In some cases this problem can be avoided by splitting rules into separate rules for each distinct cost value.

A popular example for a BURS-based matcher generator is BURG, which was presented by Fraser et al. [24]. It generates a code generator from machine specification, which is similar to the rule syntax of YACC [34]. Fraser et al. also proposed a modification of BURG, called IBURG, which performs the dynamic programming at compile time (in contrast to compile-compile time). This eliminates the two drawbacks of the BURS code generators at the expense of a sightly higher runtime.

In the labeling phase of the IBURG code generator the minimum costs are calculated for each node and each non-terminal combination and stored in cost vectors in the nodes. This is done by traversing the DFT bottom-up. A cost vector element represents the reduction with minimal cost for this node and non-terminal. For each node first all applicable base rules are checked. Then all chain rules are applied for a node until the cost vector does not change any more. In the second phase, the reduction phase, the DFT is traversed top-down, starting at the root node while selecting the non-terminal with minimal cost. The rule, which led to the selected non-terminal is identified at each node and its code generation action is applied.

The matcher generators IBURG [23] and BEG [19] use this technique. The time consumed for each node of the DFT is n^2 where n is the number of non-terminals in the grammar. Obviously this is slower than the constant time needed for a state transition in a BURS-based code generator. But the advantage is that no large state transition table is needed and the rule costs can be selected at compile time.

The drawback of all techniques outlined above is that they can only be performed on trees. This is a considerable limitation, because even if IRs are tree-like they contain

DAG structures in some ways. DAGs are either introduced by common subexpressions or by multiple def-use relations. If the computational flow inside the IR is taken into account, the input graphs for code generation are even directed graphs, which may contain cycles.

Unfortunately pattern matching on DAGs or directed graphs is NP-complete [55]. Several approaches have been proposed to overcome this problem. Many code generators simply split the DAG into trees and perform the pattern matching on the trees [2, 22, 19]. In our experiments we compare our SSA-graph matcher with this approach. It is shown that splitting the DAG into trees yields suboptimal results in many cases. The performance difference is up to 83%.

In the work of Anton Ertl [20] an approach is presented, which modifies the tree pattern matcher algorithm so that it can be used on DAGs. The tree traversal algorithm is extended by a visited flag so that it can be used on DAGs. Apart from that change, the labeling and reduction phase work like in the tree pattern matcher. The question arises if this straight forward extension also yields optimal results for DAGs. Ertl shows that this depends on the grammar. A checker, called *DBURG*, analyzes the grammar and reports if the grammar, applied on DAGs, does not yield optimal results. The checker constructs an inductive proof over all DAGs. For all ways of sharing a subgraph it is checked if all derivations are optimal. This approach differs from our approach in some points: First, for a shared node, code may be duplicated, because for each share a different non-terminal may be selected and code is generated for all selected non-terminals. Second, it is not possible to perform the algorithm on a graph containing cycles, because it still relies on the bottom-up and top-down phases of the tree pattern matcher. Cycles occur in the SSA-graph for example if the input program contains loops. As loops are important to optimize, the code generator should be able to handle cycles.

Liao et al. [46] formulate DAG pattern matching as *binate covering problem*. The algorithm works in three steps. First all matches of rules in the subject DAG are identified. A boolean variable represents the successful match of a rule for a node in the DAG. In the second step a covering matrix is built. The matrix expresses the conditions for a legal cover of the DAG. For each match, i.e. boolean variable, there is a column in the matrix. The rows represent disjunctive clauses where each variable appears with its true and complement form (therefore the problem is called binate covering problem). There are rows for each node which describe the possible matches for the nodes. In addition there are rows which are needed to formulate the dependencies — that means the connections — between the matches.

In the third step a cover with minimum cost is generated, where the cost of a match (column) is the cost of the corresponding pattern. A set of columns is selected with minimum total cost by not violating the clauses represented by the rows. A branch and bound algorithm is used to obtain the cover.

In addition to the DAG pattern matching problem, this approach also allows the

inclusion of other code generation tasks in the problem formulation. Liao et al. formulate DAG pattern matching, scheduling and spill code generation for a single accumulator architecture. Unfortunately exact solutions can only be found for small to medium sized basic blocks because of the exponential complexity of the binate covering problem. In addition this approach still does not consider the computational flow of functions.

Leupers introduced code generation to utilize SIMD instructions, based on *integer linear programming* (ILP) [42]. The difficulty of SIMD instruction selection is that a single SIMD instruction represents operations in different data flow trees. Leupers uses an algorithm which first performs tree pattern matching for each data flow tree separately. But the matcher is modified so that it gives the set of *all* optimal matches instead of one optimal match. The second part of the algorithm selects between the optimal matches to maximize SIMD instructions. This selection is formulated as an ILP problem.

2.3. Addressing Code Optimization

Addressing code optimization is a rather new topic in the field of back-end optimizations. An overview of current research work can be found in [1]. The goal of addressing code optimization is to utilize the address generation units (AGUs) of the target CPU. DSPs feature AGUs to efficiently generate memory addresses in numerical algorithms. But AGUs can also be found in micro controllers and CISC architectures (like the Motorola 68K) AGUs can be found. The main purpose of AGUs is to perform address calculations in parallel with other units of the processor. The most prominent operation of an AGU is to automatically increment or decrement an address register after accessing the memory.

Addressing code optimization is a collection of different optimization techniques. It mainly consists of three separate optimization domains:

- Offset assignment: allocating memory locations (offsets) for local variables to utilize the auto-increment addressing modes.
- Address register assignment: assigning address registers to access data for which the memory layout has been already defined.
- Addressing Mode Selection (AMS): selecting the best addressing modes for the instructions. AMS is used in offset assignment and address register assignment to generate the resulting code.

Much work has been done in the field of offset assignment and address register assignment. On the other hand there is little work concerning addressing mode selection. Although AMS is used in offset assignment and address register assignment

algorithms, the AMS problem was first formulated as a separate problem in [18]. The AMS problem is independent from offset assignment and address register assignment and this work focuses only on addressing mode selection. Nevertheless offset assignment and address register assignment can be used to improve the result of AMS.

Offset assignment is the problem of assigning a frame-relative offset to each of the local variables of a function to minimize the number of address-arithmetic instructions required to execute a basic block. Bartlay [8] was the first to address the offset assignment problem and presented an approach based on finding a Hamiltonian path of maximum weight on the graph.

Liao [47, 48] et al. formulated the *simple offset assignment problem* (SOA) which is an offset assignment problem with a single address register. They modeled the problem as a graph theoretic optimization problem similar to Bartlay and showed that the SOA problem is equivalent to the *maximum weighted path covering* (MWPC) problem and proved that it is NP-complete. In addition they extended the SOA to the *general offset assignment* (GOA) problem, which can handle multiple address registers. They proposed a heuristic algorithm to solve both the SOA and GOA problems. The work of Liao et al. built the base for a set of extensions. Leupers and Marwedel [44] proposed a tie-breaking heuristic and a variable partitioning strategy to improve the SOA and GOA result. They also used modify registers to reduce the solution costs. Sudarsanam et al. [62] extended the SOA and GOA problems by allowing an auto-increment/decrement addressing mode within a range from $-l$ to $+l$.

In [48] Liao also deals with the generation of auto-increment values for a given access sequence, which in fact is a basic form of addressing mode selection. The calculation of auto-increment values is trivial within a linear sequence of accesses, i.e. a basic block: the increment value is the difference between the offset of two consecutive accesses. The calculation of auto-increment values for a whole control flow graph (CFG) is more complicated. Liao proposes an algorithm which first identifies equivalence classes of edges. Two edges are in the same class if they have the same predecessor or successor node. An edge class represents an zig-zag pattern in the CFG. This method of identifying edge classes is also used in our approach for building the PBQP-graph for AMS. Liao argues that the number of edges in a class is limited to a small number. The problem is to find the places and values for update instructions for an edge class so that the execution time penalty is minimal. Due to the small number of possibilities, Liao's algorithm enumerates all variants and selects the one with smallest costs. The limitation of this approach is that every basic block must contain at least one access and only post modification addressing modes can be handled. Only these assumptions allow the use of the local solution within the adjacent basic blocks of a single edge class. As soon as a basic block contains no accesses or – for example – indirect-with-offset addressing modes are available, information can flow across basic block boundaries and the local solution is not guaranteed to be optimal.

In [17] we already described the problem of selecting post modifications, with the

possibility of basic blocks containing no memory accesses. But the proposed algorithm uses a heuristic and therefore it can not yield the optimal solution in many cases.

Address register assignment is used to assign address registers to variable accesses for which the offsets have been already determined. The goal is to reduce update instructions by using a minimal set of address registers to access arrays or variables inside a loop. The problem of assigning address registers to array references inside a single basic block was first described by Araujo [6] and Leupers [43]. They introduced the *local reference allocation* (LRA) problem which is solved by formulating a path covering problem on an *indexing graph* (IG). The nodes of the IG represent array references in the basic block. For each possibility for an auto-increment or auto-decrement addressing mode between two accesses, the IG contains an edge. So a path in the IG represents a sequence of accesses which can be addressed by a single address register with exclusively using auto-increment or auto-decrement.

Cintra et al. [14] and Ottoni et al. [52] extended the LRA problem to the *global reference allocation* (GRA) which can handle multiple basic blocks rather than a single basic block. In the GRA algorithm, live range merging tries to utilize all available address registers to minimize update instructions in a loop. The used technique is called *live range growth* (LRG) which repeatedly merges pairs of address register live ranges. The algorithm starts by assigning a range to each reference in the loop. Then ranges are merged pairwise until the number of live ranges is no greater than the number of available address registers. The key point of the algorithm is a merge operator which determines the costs of merging. It yields the number of explicit update instruction which is imposed by merging two live ranges.

For evaluating the merge operator Ottoni et al. construct a ϕ-dependence graph (DG_ϕ), which is derived from a static single assignment form (SSA) of the program. The DG_ϕ represents an equation system where the unknowns are *virtual references*. Each ϕ term forms an equation which imposes a decision of update values between the virtual references of the ϕ term arguments and the virtual reference of the ϕ term result. The problem of solving the equation system is NP-complete. A ϕ-Solution graph (SG_ϕ) is derived from the DG_ϕ which represents all possible solutions. If the DG_ϕ is a tree an optimal solution can be found using an algorithm based on dynamic programming which operates on the SG_ϕ. Otherwise heuristics are used to calculate a solution.

The evaluation of the merge operator is very similar to the definition of addressing mode selection, because the merge operator minimizes the number of update instructions needed for a live range merge. Compared to our approach for addressing mode selection, the merge operator algorithm has some significant limitations:

- Only auto-increment, auto-decrement and update instructions are considered. It is not possible to define other addressing modes with a flexible cost model.
- The algorithm works for a single loop. It is not obvious how to handle the control

flow graph of a whole procedure.

- In many cases the DG_ϕ is not a tree and heuristics are used to obtain a solution. In contrast our solver for the AMS problem yields an optimal solution in almost all cases, even if the DG_ϕ is not a tree.

2.4. Register Allocation

Register allocation is an essential part in a compiler back-end. It maps live ranges to physical registers. A live range is the collection of program points where a data value (e.g. a local variable or temporary) is live. Before register allocation the input program may contain a large number of live ranges which have to be mapped to a – for most embedded architectures – small number of CPU registers. If no mapping can be found for a live range, it is spilled to memory and additional load and store instructions are inserted.
Register allocation algorithms can be classified by the scope they operate on.

- *Local register allocation* is performed on a linear sequence of instructions, i.e. a basic block. It is often used as a pre-pass to global register allocation.
- *Global register allocation* allocates registers for a procedure. All basic blocks of the control flow graph are taken into account. The main focus in research has been put on global register allocation and many commercial compilers implement only this kind of register allocation. If not explicitly stated otherwise, the term "register allocation" refers to global register allocation.
- *Inter-procedural register allocation* selects which registers are available for each procedure in the program. The register allocator decides which registers must be saved during a function call.

In the following we concentrate on global register allocation and use the term "register allocation" for this allocation method. For details on local and inter-procedural register allocation the reader may refer to [21] and [40], respectively.

The commonly used technique for register allocation is graph coloring. It was first introduced by Chaitin et al. [12]. In the graph coloring approach an *interference graph* is first built. The nodes of this graph represent live ranges and the edges represent interferences, i.e. there is an edge between two live ranges if they are both live at some point in the program. Coloring the interference graph with k colors models the allocation problem if k CPU registers are available.

Figure 2.3 shows the phases of Chaitin's algorithm. The renumber phase identifies the live ranges in the program and in the build phase the interference graph is built. Coalescing tries to allocate two live ranges, which are in a copy relation, to the same

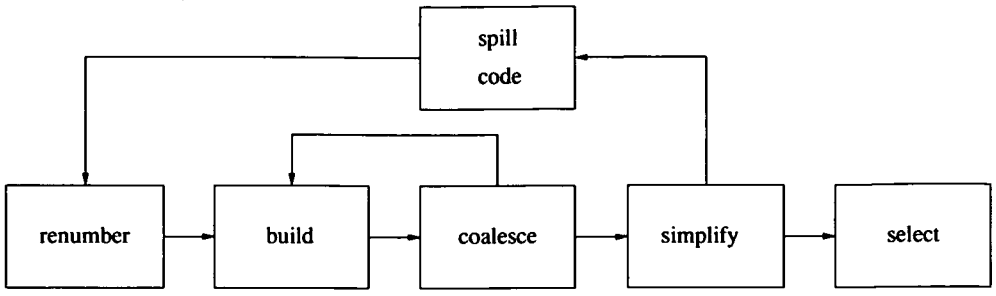


Figure 2.3.: The phases of Chaitin's register allocator

CPU register to minimize copy instructions. The simplify and select phases actually perform the graph coloring.

A node of degree less than k is trivial to color. Therefore the simplify phase eliminates nodes of degree less than k (low-degree node) from the graph and pushes them onto a stack. Eventually the graph is empty or all remaining nodes have a degree of at least k (significant-degree nodes). In the latter case a significant-degree node is marked as spilled, removed from the graph and simplification continues. Finally the select phase assigns colors, i.e. CPU registers, to the live ranges by popping nodes from the stack. For each popped node a color is selected which is distinct from the adjacent node colors. The simplify-select heuristics for solving graph coloring proved to be very efficient.

Chow and Hennessy proposed the *priority-based coloring* algorithm [13]. It has several significant differences to Chaitin's algorithm. First the register allocation runs *before* code generation on the high level IR. This means that temporary registers introduced by the code generator are not handled by register allocation. Instead of this, a fixed number of CPU registers are reserved for being used as temporaries (Chow and Hennessy propose to reserve four registers). This strategy might be a problem on embedded architectures, which typically have a small number of CPU registers. The priority-based allocator does not need to handle coalescing constraints, because it leaves this task to high level optimizations, which run earlier.

The second important difference to Chaitin's allocator is the coloring algorithm. In Chaitin's algorithm the starting point is that all live ranges are assumed to be in registers. During the algorithm some live ranges may be allocated in memory, i.e. spilled. In the priority-based algorithm at the starting point all live ranges are assumed to be in memory. Live ranges are assigned to registers in an order determined by a priority function. The algorithm stops if no more registers are available. Therefore there is no need to iterate the register allocator, like in Chaitin's model.

Another difference between the register allocation algorithms is the unit of allo-

cation. Chaitin's allocator takes an instruction as smallest unit whereas the priority-based allocator works on whole basic blocks. This makes the register allocator faster at the expense of a coarser granularity of allocation.

Chow and Hennessy also proposed a technique, called *live range splitting* to improve the coloring. If no color can be found for a live range, the live range is split so that at least a part of the original live range can be allocated to a register. The splitting algorithm starts with a basic block, preferably a definition at the entry point of the live range. Then all successor blocks are added where a CPU register is available for the live range. This is repeated in a breath-first traversal of the CFG. The so created new live range can be colored. If the remaining live range can also be colored the splitting algorithm terminates. Otherwise it is applied on the remaining live range again. The second possibility for termination is that the remaining live range can not be split anymore. In this case it is allocated to memory.

Almost all work on register allocation is based on Chaitin's or on Chow and Hennessy's approaches. Briggs et al. [10] improved Chaitin's coloring algorithm by a method called *optimistic coloring*. If only significant-degree nodes are left, such a node is pushed onto the stack instead of spilling it (it is called *potential spill*). In the select phase it is popped from the stack and tried to color. If still no color is available, it is spilled (*actual spill*). By delaying the spill decision to the select phase there is a chance that some significant-degree nodes can still be colored.

In their work, Briggs et al. also proposed an extension called *rematerialization*. For values, which are never killed, it is possible to reconstruct the value where it is needed instead of spilling and reloading it. Such values include all kinds of constant values and addresses in the static data area and in the stack frame. The algorithm for rematerialization works on the SSA-form of the procedure which in fact is a method for live range splitting. The values are propagated through the SSA-graph by using a algorithm similar to Wegman and Zadeck's constant propagation algorithm [64]. Code generation for ϕ-nodes is performed by inserting copy instructions. The *aggressive coalescing* used in Chaitin's register allocator coalesces all copy-related nodes. Therefore it would also eliminate all these copy instructions and destroy the benefit of split live ranges. To overcome this problem Briggs et al. proposed *conservative coalescing* which coalesces two nodes only if it is guaranteed that the resulting node is not spilled.

George and Appel [26] experienced that in their compiler aggressive coalescing produced too many spills, but conservative coalescing left too many copy instructions. Therefore they proposed *iterated coalescing*. They integrated conservative coalescing tightly into a loop together with the simplify step. This increases the chances of coalescing to identify that the resulting node after coalescing will not be spilled.

But even that missed some optimization opportunities. Park and Moon [53] concentrated on the positive aspects of aggressive coalescing. They transfered the idea of optimistic coloring to *optimistic coalescing*. In their register allocator first aggressive coalescing removes all copy-related live ranges. Later if a coalesced live range becomes

an actual spill, then the live range is split by undoing the coalescing.

Vegdahl addresses another potential for improvements in register allocation. In [63] he proposes an algorithm to improve the graph coloring heuristics used inside the Chaitin style register allocators. Nodes of the interference graphs are merged which results in a better coloring.

Ambrosch et al. address another problem of interference graph based register allocation in [5]. Usually the interference graph is built from the instruction lists in the basic blocks of the control flow graph. But as this order is dependent on previous passes in the compiler, the number of edges in the interference graph may vary. Ambrosch et al. introduce a *minimal interference graph*, which is built from the data dependence graph. It contains only those interferences which are required to maintain the data dependencies. They also propose a new coloring method, called *dependence-conscious register selection*. It tries to minimize anti-dependencies to improve the scheduling of the resulting allocation.

The drawback of all the interference graph based coloring methods is that they can not be extended to irregular architectures in a straight forward way. It is not possible to model register constraints other than interferences. Even the coalescing constraint, which is also very important on regular architectures, is done in a separate phase.

Beside the traditional graph coloring approaches, which are a success story for RISC-like architectures, there is some work on allocating registers for irregular architectures. Briggs [9] and Smith et al. [61] address the problem of allocating register pairs. Register pairs can be found on many CPU architectures. For example pairs of single-precision registers are used to hold double-precision floating point values. Usually two constraints are imposed by register pairs: First, the registers of the pair must be adjacent (e.g. register R3, R4) and second, the pair must be aligned on an even register number (e.g. R2, R3). Both Briggs and Smith et al. concentrate on computing the colorability of the interference graph nodes, which is used by the simplify phase to distinct between low- and significant-degree nodes. Briggs modifies the interference graph by adding edges to nodes which represent paired registers. The intention is that the degree of a node always reflects its colorability, regardless of whether the node represents a single register or a register pair.

Smith et al. do not add edges to the interference graph but define weights for all nodes, which represent the pairing constraints. They call the extended graph a *weighted interference graph*. The weights of a node and its adjacent nodes are then used in the simplify phase to compute the colorability of the node.

A similar approach is presented by Runeson and Nyström in [59]. They formulate a $\langle p, q \rangle$-test for computing the colorability of a node. In contrast to Smith and Holloway's approach, the $\langle p, q \rangle$-test covers a wider range of irregularities and it is shown how to generate the test automatically from formal architecture descriptions.

The drawback of these approaches is that the computed colorability is a worst case value. If any of the adjacent and aligned constraints are imposed, the worst case

value does not reflect the real colorability and potential spills are generated in the simplify-phase. Moreover these techniques can handle only a small subset of possible register constraints.

Koseki et al. [38] introduced *preference directed graph coloring* for handling irregularities. The idea of this method is that register preferences can be satisfied by choosing the right order of registers in the select phase. Koseki et al. build a *register preference graph* and a *color preference graph* which are used to determine the coloring order of nodes in the interference graph. The register preference graph describes register preferences among live ranges. In their paper they list four types of such preferences:

- Dedicated register usage: a register is dedicated for a special purpose, like parameter or return value passing.
- Limited register usage: some instructions can only operate on a limited set of registers.
- Preferred register usage: registers are preferred for a live range, e.g. non-volatile registers are preferred for live ranges over function calls.
- Dependent register usage: the register selection for a live range depends on another live range. Examples are coalescing and paired registers.

In the selection algorithm the color for a live range is chosen by honoring the preferences to already allocated live ranges. Therefore the result of the allocation highly depends on the order of node selection. The color preference graph represents all possible orders which do not destroy the colorability. So the register selector tries to find the ordering of selection which allows it to honor as many preferences as possible. No separate coalescing phase is employed in the algorithm. Koseki et al. show in their experiments that the preference directed algorithm has the same coalescing capabilities as the optimistic coalescing approach [53]. If the register model has irregularities, which are not handled by the optimistic coalescing register allocator, the preference directed graph coloring yields better results in terms of execution time.

The preference directed register allocator – and all other approaches based on Chaitin's graph coloring algorithm – perform the register selection for a single live range at one time. Therefore the solution for a single live range is only a locally optimal solution, but not a global optimal solution for the whole problem. This seems to be sufficient for regular architectures, but leads to suboptimal results if many irregular constraints are involved.

Hirnschrott, Krall and Scholz compare two register allocation methods in [32]. The first is based on Briggs' allocator with aggressive coalescing. In addition Smith and Holloway's extension for handling irregularities [61] is included. The second method is based on our first PBQP approach [60], but enabling exhaustive recursive enumeration

for obtaining optimal results. The comparison shows that the Chaitin-based algorithm causes significantly more spill costs than the optimal algorithm. The difference is even larger if only a few registers are available and spills occur often. Because of the exponential complexity, the optimal algorithm can only find a solution for small problem sizes in acceptable time.

Beside the graph coloring approaches, a register allocation algorithm based on integer linear programming (ILP) was introduced by Goodwin and Wilken [31] and improved by Fu and Wilken [25]. The approach maps the register allocation problem to an integer linear program which is solved by an NP-complete ILP-solver. Each allocation decision is mapped to a binary decision variable. It has the value 1 (the allocation action is performed) or 0 (the allocation action is not performed). Allocation actions include decisions, whether a live range should be mapped to a CPU register, whether the allocation should continue, or whether a spill or restore should be executed. The decisions have to be made at specific points in the input program. A *0-1 integer program* is constructed and solved by a commercial integer program solver.

The work of Goodwin and Wilken was extended by Kong and Wilken [36] for irregular architectures. As an example, they chose the IA-32 architecture and added additional features such as address mode selection. The approach can handle irregularities very nicely. However, the algorithms for solving the integer linear programs have exponential running time and are therefore not used in commercial compilers.

2.5. Partitioned Boolean Quadratic Problems

Our methods for code generation, addressing mode selection and register allocation are based on *partitioned boolean quadratic problems* (PBQPs). The PBQP is a kind of *quadratic assignment problem* (QAP) which can be found in the field of operational research. It was first formulated by Koopmans and Beckmans in [37] for describing the problem of assigning plants to locations. Burkard et al. give a comprehensive overview of the QAP and related problems in [11]. They state that the QAP is one of the hardest optimization problems.

The QAP can be described as the problem of assigning facilities to locations. The goal of the optimization is to minimize cost. The distance and flow between the facilities and the cost of a facility assigned to a location contribute to the total cost. The original formulation of the QAP is as follows:

$$\min_{\phi \in S_n} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{\phi(i)\phi(j)} + \sum_{i=1}^{n} e_{i\phi(i)} \tag{2.1}$$

where n is the number of facilities and locations. The set $N = 1, \ldots, n$ and S_n is the set of all assignments $\phi : N \to N$. The values f_{ij} describe the flow between facility i

and j and d_{kl} is the distance between the location k and l. The value e_{ik} is the cost of placing facility i at location k.

The QAP can also be formulated as a quadratic integer program as shown in Equation 2.2. In this form the boolean variables x_{ij} select the assignment.

$$\min \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} f_{ij}d_{kl}x_{ik}x_{jl} + \sum_{i,j=1}^{n} e_{ij}x_{ij} \tag{2.2}$$

$$\textit{subject to} \quad \sum_{i=1}^{n} x_{ij} = 1,\ j = 1,2,\ldots,n \tag{2.3}$$

$$\sum_{j=1}^{n} x_{ij} = 1,\ i = 1,2,\ldots,n \tag{2.4}$$

$$x_{ij} \in \{0,1\},\ i,j = 1,2,\ldots,n \tag{2.5}$$

Burkard et al. list several algorithms for exact and heuristic solutions. In addition they describe a number of similar problems to the QAP. One related problem is the *Quadratic Semi-Assignment problem* (QSAP), which can be compared to our definition of the PBQP. The QSAP was investigated by Malucelli and Pretolani in [49, 50].

The difference between the QSAP and the QAP is that in the QAP there must be the same number of facilities and locations and in the QSAP there may be n facilities and m locations. In the QAP the assignment is bijective, i.e. each location gets a single facility. This is enforced by equations 2.3 and 2.4. In the QSAP each location can have zero, one or many facilities assigned. The quadratic integer programming formulation of the QSAP is equivalent to the QAP formulation, except that the constraint 2.3 is not included.

An example for the QSAP is the assignment of n processes to m processors [49]. Process i exchanges f_{ij} units of information with process j whereas the flow of one unit from processor k to processor l takes d_{kl} time. Process i needs e_{ik} time to run on processor k.

The QSAP formulation can be easily transformed to a PBQP formulation (Definition 1 in Section 3.3) by setting $C_{ij}(k,l) = f_{ij} \cdot d_{kl}$ and $\vec{c}_i(j) = e_{ij}$. The boolean vectors in the PBQP correspond to the boolean variables in the QSAP: $\vec{x}_i(j) = x_{ij}$. In contrast to a QSAP, the decision vectors in the PBQP can have different sizes.

Malucelli and Pretolani identify a class of QSAPs which can be solved in polynomial time. These are problems whose associated communication graph are reducible by application of three reduction rules: (1) *tail reduction*, which is applied on nodes with degree one, (2) *series reduction*, which is applied to nodes with degree two and (3) *parallel reduction*, which is applied to parallel edges between two nodes. The rules are applied until a single node remains or all nodes have a degree more than two. In

the first case the result is exact, in the latter case the QSAP can not be solved in polynomial time.

This reduction method is also used by our optimal PBQP solver. The PBQP solver uses reduction rule RI which corresponds to the tail reduction. Rule RII is a combination of the series and parallel reduction.

Malucelli and Pretolani also propose methods to solve general QSAPs. They use a branch and bound algorithm and try to obtain a sharp lower bound for the QSAP. This differs to our approach, because the general PBQP solver uses heuristics if the point of non-reducibility is reached. This is fast and it is sufficient for our PBQP applications. In addition to the QSAP algorithm we employ simplifications on the PBQP-graph to improve the reducibility.

Beside of the QSAP there are other related problems to the QAP, like the *bottleneck quadratic assignment problem* (BQAP) and the *bi-quadratic assignment problem.* However these QAP variants have fewer similarities to the PBQP than the QSAP and the QAP and are therefore not in the focus of our work.

3. Partitioned Boolean Quadratic Problems

3.1. Overview

This chapter introduces the *partitioned boolean quadratic problem* (PBQP). It is used to formulate the optimization problems presented in this work – SSA-graph matching, addressing mode selection and register allocation.

The PBQP formulation has two main advantages over conventional approaches. First, the PBQP provides a unified interface to describe the problems with the help of cost vectors and cost matrices. So it is easy to formulate irregularities imposed by the target architecture. Second, there exists a solver which can yield optimal or near-optimal solutions for our optimization problems.

A PBQP can be described as a cost function or a graph problem. The cost function is the formal definition, whereas the graph representation is more descriptive. The graph representation is used by the solver.

As the complexity of a PBQP is NP it can not be solved optimally in general. But for a certain subclass of PBQPs an optimal solution can be computed in polynomial time. We present an optimal solver which computes the optimal solution if it is possible. If the optimal solver fails, a general solver can be used which employs heuristics for solving general PBQPs.

3.2. Background

Vectors and Matrices A *matrix* A is any rectangular array of elements. If a matrix A has m *rows* and n *columns*, we call A a $m \times n$ matrix. Let $A(i,j)$ be the ij^{th} element in the i^{th} row and j^{th} column. The i^{th} row is denoted by $A(i,:)$ and the j^{th} column by $A(:,j)$. A matrix in which all elements are zero is called a *zero matrix*.

A *vector* $\vec{v}$ is a $1 \times n$ matrix with n elements. Let $\vec{v}(i)$ be the i^{th} element of the vector and let $|\vec{v}| = n$ be the length of a vector. Vector $\vec{1}$ in which all elements are one is called a *one vector*.

Given any $m \times n$ matrix A, the *transpose* for A (written A^T) is the $n \times m$ matrix whose ij^{th} element $A^T(i,j) = A(j,i)$. Let A and B be two $m \times n$ matrices. The matrix $C = A + B$ is defined to be the $m \times n$ matrix whose ij^{th} element is $C(i,j) = A(i,j) + B(i,j)$. The *matrix product* $C = A \cdot B$ of A and B is the $m \times n$ matrix whose ij^{th} element is determined by $C(i,j) = \sum_{k=1}^{l} A(i,k)B(k,j)$ where l is the number of columns in matrix A and the number of rows in matrix B. The product of two vectors $\vec{u}$ and $\vec{v}^T$, both of length n, is called the *dot product*. It yields a 1×1 matrix – a scalar.

A *quadratic form* is defined to be a vector-matrix-vector product $\vec{x}A\vec{y}^T$, which has the following properties:

$$\vec{x}A\vec{y}^T = \vec{y}A^T\vec{x}^T \tag{3.1}$$

$$\vec{x}A\vec{y}^T + \vec{x}B\vec{y}^T = \vec{x}(A+B)\vec{y}^T \tag{3.2}$$

In this work vector and matrix elements represent cost values. They are defined over the domain of real numbers including infinity (∞). We define the arithmetic operations for ∞ as follows:

$$\begin{aligned} \infty \cdot 0 = 0 \cdot \infty &= 0 \\ \infty \cdot x = x \cdot \infty &= \infty \quad \forall x \neq 0 \\ \infty + x = x + \infty &= \infty \quad \forall x \\ x &\leq \infty \quad \forall x \end{aligned}$$

Minimum Operation The minimum over parameter x of function $f(x)$ is defined as follows,[1]

$$\min f(x) = f(\bar{x}) \quad | \quad \exists \bar{x} \in D, \forall x \in D : f(\bar{x}) \leq f(x) \tag{3.3}$$

where D is the domain of x and $\bar{x}$ is a solution of the minimum operation.[2] The *partial minimum* over parameter x of function $f(x, y)$ is defined as follows,

$$\min_x f(x, y) = f(\bar{x}(y), y) \quad | \quad \forall y \in D, \exists \bar{x}(y) \in D, \forall x \in D : f(\bar{x}(y), y) \leq f(x, y) \tag{3.4}$$

We use the bar-notation (e.g. $\bar{x}$) for solutions of minimum problems throughout this work. Note that the minimum function yields a value whereas the partial minimum function yields a function. One exception is if the partial minimum function defines the minimum over all parameters. In this case it is equivalent to the minimum function.

$$\min_{\langle x,y \rangle} f(x, y) = \min f(x, y) \tag{3.5}$$

The partial minimum operator has following properties. Let $\langle \bar{x}, \bar{y} \rangle$ be the solution of $\min f(x, y)$.

$$\min_{\langle x,y \rangle} f(x, y) = \min_x \min_y f(x, y) \tag{3.6}$$

$$\min_{\langle x,y \rangle} f(x, y) = \min_y f(\bar{x}, y) = \min_x f(x, \bar{y}) \tag{3.7}$$

The minimum of vector $\vec{v}$ is defined as

[1] Note that the parameters of the minimum functions may also be n-tuples.

[2] In this work the domains of the minimum parameters are obvious and well defined. Therefore they are not explicitly quoted at each min operator.

$$\min(\vec{v}) = \min_i \vec{v}(i) \tag{3.8}$$

where $1 \leq i \leq |\vec{v}|$, is the smallest element c of vector $\vec{v}$ such that $\forall 1 \leq i \leq |\vec{v}| : c \leq \vec{v}(i)$. The *minimum index* $i_{min}(\vec{v})$ of a vector is defined to be the smallest index of minima and the following must hold: $\vec{v}(i_{min}(\vec{v}))) = \min(\vec{v})$. If there are more elements where the equation holds, the element with the smallest index is taken.

Graphs Let $G\langle V, E\rangle$ be a *directed graph*, where V is a set and E a relation on V. The elements of V are called nodes and the ordered pairs in E are called edges. A graph is *undirected* if E is symmetric, i.e. $(u, v) \in E \Leftrightarrow (v, u) \in E$. Removing edge (u, v) from an undirected graph is denoted by $E - (u, v)$, which is equivalent to $E - \{(u, v), (v, u)\}$. Adding an edge is handled similarly: $E + (u, v) = E \cup \{(u, v), (v, u)\}$. Let $adj(u) = \{v | (u, v) \in E\}$ be a set of *adjacent nodes* of u and $deg(u) = |\, adj(u)\,|$ the *degree* of node u. A node u is said to be *disconnected* if $deg(u) = 0$. To traverse a directed graph we define the following four functions. Let $u, v \in V$ be a node and $e = (u, v)$ be an edge.

$$succ(u) = \bigcup_{(u,v)\in E} \{v\} \qquad source(e) = u$$

$$pred(v) = \bigcup_{(u,v)\in E} \{u\} \qquad target(e) = v$$

3.3. PBQP Definition

The PBQP is defined as a cost function over a set of boolean decision vectors $\vec{x}_i$. Each element in a decision vector is either zero or one. In addition the dot product is exactly one for each vector $\vec{x}_i$, in other words: exactly one element of a vector is one, all other elements are zero. This explains the term "decision vector". The goal of the PBQP solver is to decide which elements should be selected, i.e. should be one, to minimize the cost function f. The cost function is the sum of all vector-matrix-vector dot products between the decision vectors and vector-vector dot products. The matrices C_{ij} specify the costs between decision vectors and the cost vectors $\vec{c}_i$ specify the costs for each decision vector separately.

Definition 1. *A partitioned boolean quadratic problem (PBQP) is defined over a n-*

tuple of boolean decision vectors $X = \langle \vec{x}_1, ..., \vec{x}_n \rangle$ *as follows:*

$$\min f(X) = \sum_{1 \le i < j \le n} \vec{x}_i \cdot C_{ij} \cdot \vec{x}_j^T + \sum_{1 \le i \le n} \vec{c}_i \cdot \vec{x}_i^T \tag{3.9}$$

$$\textit{subject to: } \forall i \in 1 \ldots n : \vec{1}^T \cdot \vec{x}_i = 1 \tag{3.10}$$

where n *is the number of decision vectors.*

The domain D_i of a single decision vector $\vec{x}_i$, satisfying constraint 3.10, is the set of all vectors which have a single one-element: $D_i = \{\vec{x} \mid \vec{x} \cdot \vec{1}^T = 1\}$. The domain of the parameter X of the objective function $f(X)$ is the cross product of the decision vector domains: $D_X = D_1 \times ... \times D_n$. Note that the decision vectors may have different lengths. The sizes of vectors $\vec{c}_i$ and matrices C_{ij} must match with the decision vector lengths so that the products in Equation 3.9 are defined.

Due to the symmetric properties of quadratic forms, the cost function is a triangular sum. A solution of the PBQP is formulated as the decision vectors where the objective function is a minimum. It is denoted as $\bar{X} = \langle \bar{\vec{x}}_1, ..., \bar{\vec{x}}_1 \rangle$.

$$\min f(X) = f(\bar{X}) \tag{3.11}$$

As the PBQP can have more than one solution, $\bar{X}$ is just one representative of the solution space. As the decision vectors must have a single one-element, the solution $\bar{X}$ can also be specified by the index of the one-element in the decision vectors: $S = \langle s_1, ..., s_n \rangle$ where s_i is the index of the one-element in decision vector $\bar{\vec{x}}_i$. Each solution element s_i is in the range $1 \le s_i \le |\bar{\vec{x}}_i|$.

Let us take a closer look at the matrices C_{ij} in the cost function. The term $\vec{x}_i \cdot C_{ij} \cdot \vec{x}_j^T$ selects one element of the matrix, because exactly one element of $\vec{x}_i$ and one element of $\vec{x}_j$ is one. Let s_i be the index of the one-element in $\vec{x}_i$ and s_j the one-element in $\vec{x}_j$, then the term $\vec{x}_i \cdot C_{ij} \cdot \vec{x}_j^T$ yields $C_{ij}(s_i, s_j)$. In other words the matrix element $C_{ij}(s_i, s_j)$ contributes to the objective function if element s_i is selected in $\vec{x}_i$ and element s_j is selected in $\vec{x}_j$. So matrix C_{ij} specifies the cost values for each combination of selected elements in $\vec{x}_i$ and $\vec{x}_j$. Figure 3.1 visualizes the cost matrix in an alternative way: each matrix element is represented by a line connecting two adjacent decision vector elements. Therefore a cost matrix element can also be viewed as the transition costs from one decision vector element to the adjacent decision vector element.

Similar to the cost matrices, cost vectors $\vec{c}_i$ contribute to the cost functions, but are selected by a single decision vector instead of two decision vectors. The term $\vec{c}_i \cdot \vec{x}_i^T$ selects one element of the cost vector $\vec{c}_i$ because exactly one element of $\vec{x}_i$ is one. Let s_i be the index of the one-element in $\vec{x}_i$ then the term $\vec{x}_i \cdot \vec{c}_i$ yields $\vec{c}_i(s_i)$. Again, element $\vec{c}_i(s_i)$ contributes to the objective function if element s_i is selected in $\vec{x}_i$.

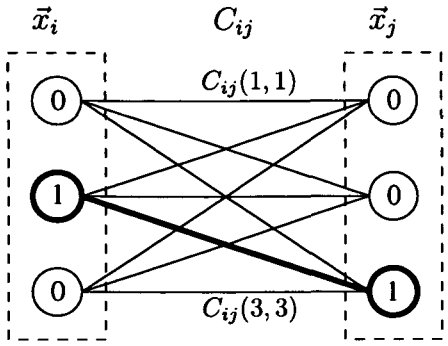


Figure 3.1.: The cost matrix shown as transition costs. Each matrix element is represented by a line connecting two adjacent decision vector elements. In this example the matrix element $C_{ij}(2,3)$ is selected by the decision vectors $\vec{x}_i = (0\ 1\ 0)$ and $\vec{x}_j = (0\ 0\ 1)$.

$$
\begin{aligned}
minf = \vec{x}_1 \cdot C_{12} \cdot \vec{x}_2^T + \vec{x}_1 \cdot C_{13} \cdot \vec{x}_3^T + \vec{x}_1 \cdot C_{14} \cdot \vec{x}_4^T + \\
\vec{x}_2 \cdot C_{23} \cdot \vec{x}_3^T + \vec{x}_2 \cdot C_{24} \cdot \vec{x}_4^T + \\
\vec{x}_3 \cdot C_{34} \cdot \vec{x}_4^T + \\
\vec{c}_1 \cdot \vec{x}_1^T + \vec{c}_2 \cdot \vec{x}_2^T + \vec{c}_3 \cdot \vec{x}_3^T + \vec{c}_4 \cdot \vec{x}_4^T =
\end{aligned}
$$

$$
\begin{aligned}
= \vec{x}_1 \cdot \begin{pmatrix} 2 & 7 \\ 0 & 9 \end{pmatrix} \cdot \vec{x}_2^T + \vec{x}_1 \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot \vec{x}_3^T + \vec{x}_1 \cdot \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \cdot \vec{x}_4^T + \\
\vec{x}_2 \cdot \begin{pmatrix} 3 & 6 & 0 \\ 6 & 2 & 6 \end{pmatrix} \cdot \vec{x}_3^T + \vec{x}_2 \cdot \begin{pmatrix} 1 & 8 \\ 7 & 9 \end{pmatrix} \cdot \vec{x}_4^T + \\
\vec{x}_3 \cdot \begin{pmatrix} 2 & 0 \\ 2 & 3 \\ 7 & 5 \end{pmatrix} \cdot \vec{x}_4^T + \\
\begin{pmatrix} 6 & 7 \end{pmatrix} \cdot \vec{x}_1^T + \begin{pmatrix} 5 & 3 \end{pmatrix} \cdot \vec{x}_2^T + \begin{pmatrix} 5 & 6 & 2 \end{pmatrix} \cdot \vec{x}_3^T + \begin{pmatrix} 9 & 1 \end{pmatrix} \cdot \vec{x}_4^T
\end{aligned}
$$

Figure 3.2.: Example PBQP

Figure 3.2 shows an example PBQP. It contains four decision vectors: $\vec{x}_1$, $\vec{x}_2$ and $\vec{x}_4$ have a length of 2 and $\vec{x}_3$ has a length of 3.

The difficulty of finding a solution to this minimization problem is that the contributing products can not be treated locally (which would be trivial). The decision vectors make it a global problem which is NP-hard to solve. But if the PBQP is sparse, this means it contains many zero cost matrices C_{ij}, a solver can yield an optimal or near optimal solution. In the example matrices, C_{13} and C_{14} are zero matrices and therefore the terms $\vec{x}_1 \cdot C_{13} \cdot \vec{x}_3^T$ and $\vec{x}_1 \cdot C_{14} \cdot \vec{x}_4^T$ do not contribute to the objective function.

3.3.1. The PBQP-Graph

For the solver algorithm we construct an undirected *PBQP-graph* $G\langle V, E, w\rangle$, which is an equivalent representation of the PBQP. In the PBQP-graph decision vector $\vec{x}_i$ is represented by a node $v_i \in V$ $(1 \leq i \leq n)$. Nodes $v_i \in V$ and $v_j \in V$ $(1 \leq i < j \leq n)$ are connected by an edge $(v_i, v_j) \in E$ if cost matrix C_{ij} is not zero. The cost function w maps nodes $v_i \in V$ to cost vectors $\vec{c}_i$ and edges $(v_i, v_j) \in E$ to cost matrices C_{ij}.

As each undirected edge between nodes v_i and v_j consists of two directed edges (v_i, v_j) and (v_j, v_i), the cost function w maps each directed edge to cost matrices C_{ij} and C_{ji}, respectively. Due to the properties of quadratic forms (see Section 3.2), matrix C_{ij} is the transposed matrix of C_{ji}, i.e. ${C_{ij}}^T = C_{ji}$. The row index of C_{ij} relates to the decision elements of the predecessor node v_i and the column index relates to the decision element of successor node v_j of the directed edge (v_i, v_j).

In the following we denote C_{uv} as cost matrix of edge (u, v) which is equivalent to $w(u, v)$. Due to the structure of PBQPs, G has no reflexive edge $(v_i, v_i) \notin E$, since we have no cost matrices C_{ii} in objective function f and there is at most one edge between two nodes.

Figure 3.3 shows the PBQP-graph of the example PBQP. The graph contains edges between two nodes if the corresponding matrix is not a zero matrix. In the example the matrices C_{13} and C_{14} are zero matrices, therefore the graph contains no edges between nodes $v_1 - v_3$ and between nodes $v_1 - v_4$.

For convenience we define some graph theoretical terms for the PBQP formulation as cost function, too. The degree of a decision vector $\vec{x}$ is defined as the degree of the corresponding node v_i in the PBQP-graph. This means the degree of a decision vector $\vec{x}$ is the number of non-zero matrices which are multiplied with $\vec{x}$ in the objective function f.

Definition 2. $deg(\vec{x}_i) = deg(v_i)$

Similarly the set of adjacent decision vectors of a decision vector $\vec{x}$ is defined as the set of decision vectors which correspond to the set of adjacent nodes of the corresponding node v_i in the PBQP-graph. This means the set of adjacent decision

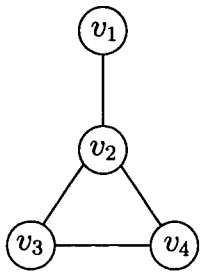


Figure 3.3.: The PBQP-graph of the example PBQP

vectors of $\vec{x}$ is the set of decision vectors which are in the same multiplication term as $\vec{x}$ and a non-zero cost matrix in the objective function f.

Definition 3. $adj(\vec{x_i}) = \{x_j | v_j \in adj(v_i)\}$

3.3.2. PBQP Complexity

The PBQP is NP-complete.[3] To show this we use the PBQP to describe the MAX-CUT problem. The MAXCUT problem is one of the Karp's original NP-complete problems [35]. Given a graph $G = \langle V, E \rangle$, the cut associated with the node set S is the set of edges that have one endpoint in S and the other endpoint not in S. For each edge $e_i \in E$ the function $m(e_i) = m_i$ defines the weight for the edge (the weighting function m of the MAXCUT graph should not be confused with the cost function w of the PBQP). The weight of a cut is the sum of the weights of the cut edges. The MAXCUT problem is to find a node set S that maximizes the weight of the cut. Determining a maximum cut in an arbitrary graph is a NP-complete problem.

We construct PBQP-graph $G'\langle V', E', w \rangle$ with a cost function, equivalent to the dual problem of the MAXCUT problem: the PBQP defines the problem of minimizing the edges, which are *not* in the cut.

We define $V' = V$, so there is a decision vector for each node $v \in V$ of the MAXCUT problem. The length of all decision vectors is two. A decision vector of a node indicates if the node is included in S or not.

Definition 4. $v_i \in S$ *iff* $\vec{x}_i = (\ 1\ \ 0\ )$, $\quad v_i \notin S$ *iff* $\vec{x}_i = (\ 0\ \ 1\ )$

As for the nodes we define $E' = E$, so the PBQP-graph is equivalent to the MAXCUT graph. The cost function w defines a cost matrix C for each edge and a zero cost vector for each node.

[3] This can be derived from the fact that it is used to model the three optimization problems, presented in this work (SSA-graph matching, addressing mode selection and register allocation), which are themselves NP-complete. Nevertheless we provide a proof for NP-completeness of the PBQP in this section.

Definition 5. $w(v_i) = (\ 0 \quad 0\ )\ \forall v_i \in V \qquad w(e_i) = \begin{pmatrix} m_i & 0 \\ 0 & m_i \end{pmatrix}\ \forall e_i \in E$

Each edge $e_i \in E$ contributes to the objective function by the term

$$\vec{x}_j \cdot \begin{pmatrix} m_i & 0 \\ 0 & m_i \end{pmatrix} \cdot \vec{x}_k^T$$

where $\vec{x}_j$ and $\vec{x}_k$ are the decision vectors of the adjacent nodes of the edge e_i. There are four possibilities in the decisions of the adjacent nodes.

$$(\ 1 \quad 0\ ) \cdot \begin{pmatrix} m_i & 0 \\ 0 & m_i \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \quad m_i$$

$$(\ 1 \quad 0\ ) \cdot \begin{pmatrix} m_i & 0 \\ 0 & m_i \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \quad 0$$

$$(\ 0 \quad 1\ ) \cdot \begin{pmatrix} m_i & 0 \\ 0 & m_i \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \quad 0$$

$$(\ 0 \quad 1\ ) \cdot \begin{pmatrix} m_i & 0 \\ 0 & m_i \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \quad m_i$$

In cases where both adjacent nodes are included in S or both adjacent nodes are not included in S, the term for the edge contributes by a value of m_i, otherwise by a value of 0. So the objective function f yields the sum of the weights of edges which are not in the cut.

$$MC = \sum_{e_i \in E} m(e_i) - f \tag{3.12}$$

Equation 3.12 yields the sum of the weights of edges which are included in the cut. As the objective function f is minimized, MC is maximized. Therefore the solution of the PBQP is the maximum cut.

3.4. Optimal Solver

In this section we present a solver which can yield optimal results for a certain class of PBQPs. For general PBQPs a general solver, which is presented in the next section, can be used.

The optimal solver is well suited to be applied on a certain subclass of PBQP problems: PBQPs which are sparse, i.e. have a sparse PBQP-graph. The solver works in three phases. In the first phase the PBQP is reduced with defined reduction rules until the objective function becomes trivial. In the second phase we determine the solution for the trivial objective function. In the third phase the solution of the reduced decision vectors is computed. The third phase is called *back propagation* and works exactly in the reverse order as the first phase.

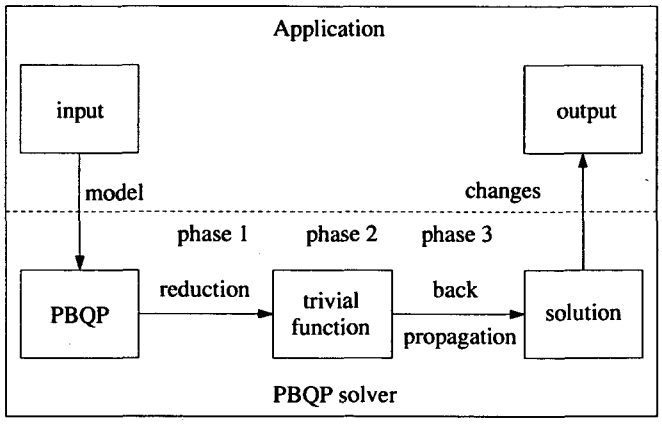


Figure 3.4.: Overview of the PBQP solver

Figure 3.4 gives an overview of the PBQP solver, which is used in an application. From the input data a PBQP is modeled. Then the solver runs in the three phases. Finally the output is generated from the solution.

The limitation of the optimal solver is that it can not yield the result for arbitrary PBQPs. An optimal solution can only be generated if the PBQP-graph can be reduced with a certain set of reduction rules. The challenge is to find reduction rules, which can be performed in polynomial time such that the optimality of PBQP is not destroyed. In general this is not possible since the underlying problem is NP-complete. In the following, decision vectors, which can be reduced by the optimal solver, are called *reducible*. On the other hand, decision vectors which can only be reduced by the general solver are called *irreducible*.

In the following we describe the solver from two different views: first we describe the solver by looking at the original definition of the PBQP – the objective function f. In addition we provide the definitions also for the PBQP-graph. This is the more intuitive way to look at the solver and it is also used by the algorithm implementation in Section 3.6.

3.4.1. Phase 1: Reduction

In the reduction phase we eliminate nodes from the PBQP-graph until only nodes with degree zero remain. As each node represents a decision vector, eliminating a node from the PBQP-graph means to eliminate a decision vector from the objective function f. We denote the original objective function as f^0 and the consecutive k^{th} reduced functions as f^k. Similarly we denote the original graph as Graph G^0. Each application of a reduction rule transforms some reduced PBQP-graph G^k to G^{k+1} until

all nodes of the resulting graph are disconnected, i.e. have degree zero.

For eliminating the k^{th} reduced decision vector from the n-tuple X^k we introduce the notation of $X^{k\backslash x}$.

Definition 6. *Let $\vec{x}_i$ be the k^{th} reduced decision vector and*

$$X^k = \langle \vec{x}_a, \ldots, \vec{x}_b, \vec{x}_i, \vec{x}_c, \ldots, \vec{x}_d \rangle$$

Then $X^{k\backslash x}$ is defined as follows.

$$X^{k\backslash x} = \langle \vec{x}_a, \ldots, \vec{x}_b, \vec{x}_c, \ldots, \vec{x}_d \rangle$$

We denote the original parameter n-tuple as X^0 and the parameter of the consecutive k^{th} reduced functions as X^k. Obviously, $X^{k\backslash x}$ is an equivalent notation to X^{k+1}. In addition we denote the solution of the original objective function as $\bar{X}^0$ and $\bar{X}^k$ denotes the solution of the k^{th} reduced objective function.[4]

The optimality of the reduction algorithm of the solver is expressed in Equation 3.13. It says that the solution of the reduced function is equivalent to the solution of the original function, excluding the reduced decision vector.

$$\bar{X}^{k\backslash x} = \bar{X}^{k+1} \tag{3.13}$$

Equation 3.13 is used later to prove the optimality of the solver. In addition we observe a property of the reductions of the optimal solver, which is expressed in Equation 3.14. It states that the resulting minimum of the reduced objective function is equal to the minimum of the original function. This property is is not needed for the optimality of the solver but it is useful to obtain the total minimum costs of a PBQP.

$$\min f^k(X^k) = \min f^{k+1}(X^{k+1}) \tag{3.14}$$

In the following we introduce the reduction rules which meet the conditions stated in Equations 3.13 and 3.14.

The solver employs two reduction rules, which are RI and RII. Rule RI is applied on nodes with degree one, rule RII is applied on nodes with degree two. If during the reduction process there are no nodes of degree one or two left, the optimal solver can not calculate a result.

[4] Note that from the definition of $\bar{X}^k$ it does not follow that $\bar{X}^{k\backslash x}$ is equivalent to $\bar{X}^{k+1}$. Instead we *state* that both terms must be equal to satisfy the optimality of the solver and we have to prove it.

Rule RI In the following, definitions are given for the reduction rule RI which is applied to a node x of degree one.

Definition 7 is given for applying rule RI on the objective function f. Note that because there is only one adjacent vector $\vec{y} \in adj(\vec{x})$, the equation in the following definition lists all terms which contain $\vec{x}$ and contribute to the objective function. For the sake of simplicity we assume that the index of $\vec{x}$ in X is lower than the index of $\vec{y}$ in X: index($\vec{x}$) < index($\vec{y}$). As the order of decision vectors in X is arbitrary, this assumption is permissible.[5]

Definition 7. *Let $f^k(X^k)$ be the k^{th} reduced objective function. Let $\vec{x}$ be a decision vector of degree one and $\vec{y}$ the adjacent decision vector of $\vec{x}$. Then*

$$RI(f^k(X^k), \vec{x}) \Rightarrow f^{k+1}(X^{k+1}) = f^k(X^k) - \vec{x} \cdot C_{xy} \cdot \vec{y}^T - \vec{c_x} * \vec{x}^T + \vec{\delta} \cdot \vec{y}^T$$

where $\vec{\delta}$ is defined as follows:

$$\vec{\delta}(i) = min(C_{yx}(i,:) + \vec{c_x}) \tag{3.15}$$

Vector $\vec{\delta}$ considers the minimal costs of node $\vec{x}$ dependent on the decision of $\vec{y}$. It replaces the costs of the reduced node and its adjacent edge. Therefore no cost terms involving $\vec{x}$ are present in the reduced function f^{k+1}. The new costs of δ are accumulated to the costs of the adjacent decision vector $\vec{y}$. Following definition is given for reduction rule RI applied on the PBQP-graph:

Definition 8. *Let $G^k\langle V, E, w\rangle$ be the k^{th} reduced PBQP-graph. Let x be a node of degree one and y the adjacent node of x. Then*

$$RI(G^k\langle V, E, w^k\rangle, x) \Rightarrow G^{k+1}\langle V - \{x\}, E - (x,y), w^{k+1}\rangle$$

The cost function w^{k+1} is identical to cost function w^k except for node y. The cost vector $\vec{c}_y$ is incremented by $\vec{\delta}$. The reduction rule RI can also be illustrated graphically as shown in Figure 3.5. Each element of vector $\vec{c}_y$ is incremented by the minimum cost for all possibilities of vector $\vec{x}$.

Lemma 2 provides the optimality criteria for reduction rule RI. The proofs for the following two lemmas are shown in Appendix A.

Lemma 1. *Let $f^{k+1} = RI(f^k, \vec{x})$. Then* $\min f^k(X^k) = \min f^{k+1}(X^{k+1})$ *(Equation 3.14).*

Lemma 2. *Let $f^{k+1} = RI(f^k, \vec{x})$. Then $\bar{X}^{k\backslash x} = \bar{X}^{k+1}$ (Equation 3.13).*

[5]The order ensures that function f contains the term $\vec{x} \cdot C_{xy} \cdot \vec{y}^T$ and not $\vec{y} \cdot C_{yx} \cdot \vec{x}^T$.

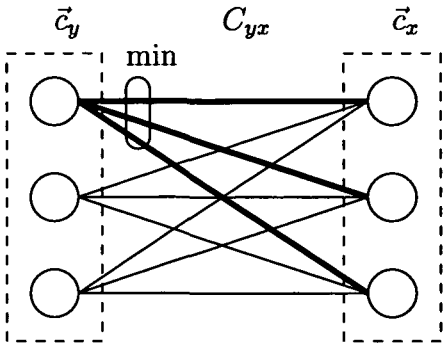


Figure 3.5.: Calculation of $\vec{\delta}(1)$: It is the minimum of all possible paths to the reduced node x (indicated by bold lines).

Rule RII In the following we describe reduction rule RII which follows the same schema as rule RI. Rule RII is applied to a node x of degree two.

Definition 9 is given for applying rule RII on the objective function f. Note again that because there are only two adjacent vectors $\vec{y}, \vec{z} \in adj(\vec{x})$, the equation in the following definition lists all terms which contain $\vec{x}$ and contribute to the objective function. As for rule RI we assume an order of the vectors in X: index$(\vec{x})$ < index$(\vec{y})$ < index$(\vec{z})$.

Definition 9. *Let $f^k(X^k)$ be the k^{th} reduced objective function. Let $\vec{x}$ be a decision vector of degree two, $\vec{y}$ and $\vec{z}$ the adjacent decision vectors of $\vec{x}$. Then*

$$RII(f^k(X^k), \vec{x}) \Rightarrow f^{k+1}(X^{k+1}) = f^k(X^k) - \vec{x} \cdot C_{xy} \cdot \vec{y}^T - \vec{x} * C_{xz} \cdot \vec{z} - \vec{c_x} \cdot \vec{x}^T + \vec{y} \cdot \Delta \cdot \vec{z}^T$$

where Δ is defined as follows:

$$\vec{\Delta}(i,j) = min(C_{yx}(i,:) + C_{zx}(j,:) + \vec{c_x}) \tag{3.16}$$

Matrix Δ considers the minimal costs of node $\vec{x}$ dependent on the decision of $\vec{y}$ and $\vec{z}$. It replaces the costs of the reduced node and its adjacent edges. This removes the cost terms involving $\vec{x}$ in the reduced function f^{k+1}. The new cost matrix Δ is added to the cost matrix C_{yz} between $\vec{y}$ and $\vec{z}$. Now we give the definitions of rule RII for the PBQP-graph:

Definition 10. *Let $G^k\langle V, E, w\rangle$ be the k^{th} reduced PBQP-graph. Let x be a node of degree two and y and z adjacent nodes of x. Then*

$$RII(G^k\langle V, E, w^k\rangle, x) \Rightarrow G^{k+1}\langle V - \{x\}, E + (y,z) - (x,y) - (x,z)\}, w^{k+1}\rangle$$

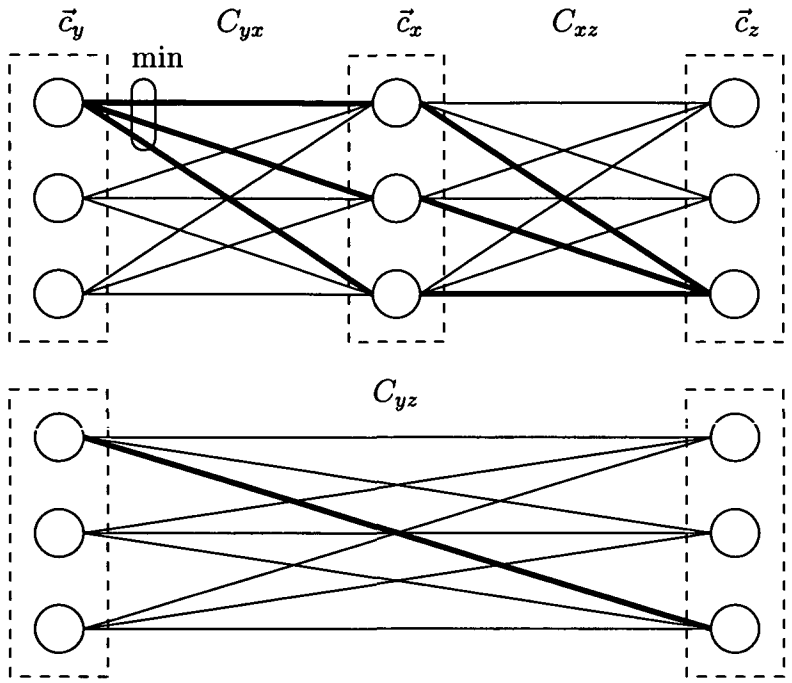


Figure 3.6.: Calculation of $\Delta(1,3)$: it is the minimum of all possible paths to the reduced node x (indicated by bold lines).

The cost function of the edge (y,z) is determined by the Δ matrix. If edge (y,z) already exists in the graph, we add Δ to cost matrix C_{yz}. Otherwise, a new edge (y,z) is inserted and the cost function of edge (y,z) yields the cost matrix Δ. As with rule RI, the reduction rule RII can also be illustrated graphically which is shown in Figure 3.6. Each element of cost matrix C_{yz} is incremented by the minimum cost for all possibilities of vector $\vec{x}$.

For reduction rule RII the optimality criteria is provided by Lemma 4. The proofs for the following two lemmas are shown in Appendix A.

Lemma 3. *Let* $f^{k+1} = RII(f^k, \vec{x})$. *Then* $\min f^k(X^k) = \min f^{k+1}(X^{k+1})$ *(Equation 3.14).*

Lemma 4. *Let* $f^{k+1} = RII(f^k, \vec{x})$. *Then* $\bar{X}^{k\backslash x} = \bar{X}^{k+1}$ *(Equation 3.13).*

3.4.2. Phase 2: Trivial Solution

Reduction rules RI and RII are applied until only nodes with degree zero remain. This means that all cost matrices in the objective function are zero matrices and the

objective function f is reduced to

$$\min f = \sum_{1 \leq i \leq n} \vec{c}_i \cdot \vec{x}_i^T \tag{3.17}$$

Because each summand in the objective function depends on exactly one decision vector, the minimum operator can be specified for each summand separately as shown in Equation 3.18.

$$\min \left[\sum_{1 \leq i \leq n} \vec{c}_i \cdot \vec{x}_i^T \right] = \sum_{1 \leq i \leq n} \min_{\vec{x}_i} \left[\vec{c}_i \cdot \vec{x}_i^T \right] \tag{3.18}$$

The solutions of the decision vectors $\vec{x}_i$ can be determined by finding the smallest element of $\vec{c}_i$ since there is no dependence between decision vectors. Therefore, solution of $\vec{x}_i$ is $s_i = i_{min}(\vec{c}_i)$ where s_i is the index of the element in $\vec{x}_i$ whose value is set to one. So the minimum value of the trivial function f can expressed as follows:

$$\min f = \sum_{1 \leq i \leq n} \min(\vec{c}_i) \tag{3.19}$$

3.4.3. Phase 3: Back Propagation

In the third phase we compute solutions for decision vectors which were eliminated in the first phase by propagating the solution through the eliminated decision vectors. The back propagation phase re-inserts decision vectors in the exact reverse order as they were eliminated in the reduction phase. The solution of each re-inserted decision vector can be calculated because all other decision vectors of the solution are already known.

Definition 11. *Let $f^k(X^k)$ the k^{th} reduced objective function. Let $\vec{x}$ be the decision vector which is eliminated by reduction from f^k to f^{k+1}. Further let $\bar{X}^k = \min f^k(X^k)$ the solution of the k^{th} reduced objective function. Then*
$BP(\bar{X}^{k+1}, f^k, \vec{x}) = \bar{X}^k$

So back propagation yields the solution of the k^{th} objective function provided with the solution of the next reduced objective function. The proof for Lemma 5 is given in Appendix A.

Lemma 5. *Let $f^k(X^k)$ the k^{th} reduced objective function. Let $BP(\bar{X}^{k+1}, f^k, x) = \bar{X}^k$. Given the solution $\bar{X}^{k+1}$ the solution vector $\bar{\vec{x}}$ is obtained by following equation*

$$s = i_{min} \left[\vec{c}_x + \sum_{\vec{y} \in adj(\vec{x})} \bar{\vec{y}} \cdot {C_{xy}}^T \right] \tag{3.20}$$

where s is the index of the one-element in $\vec{x}$.

With back propagation we can put all together and show the optimality of the solver.

Theorem 1. *The solution $\bar{X}$ of a PBQP* $\min f(X)$ *can be obtained by first applying reduction rules RI and RII until the objective function becomes trivial and then applying back propagation.*

Proof. Theorem 1 is proved by complete induction on the number of decision vectors n in f. The induction start is n so that $deg(\vec{x}) = 0 \ \forall \vec{x}$ in f. For such an objective function a solution can be calculated from Equation 3.19. Induction step: We show that the solution $\bar{X}^k$ for function f^k can be obtained from the solution $\bar{X}^{k+1}$ for f^{k+1}, if the function f^{k+1} is derived from f^k by reduction rules RI or RII.

According to Lemma 2 and Lemma 4, the solution $\bar{X}^{k+1}$ is equivalent to $\bar{X}^{k\backslash x}$, which is the solution of f^k without the reduced decision vector. The solution for the reduced decision vector $\bar{\vec{x}}$ can be derived from $\bar{X}^{k+1}$ by applying Equation 3.20. The complete solution $\bar{X}^k$ is composed by adding $\bar{\vec{x}}$ to $\bar{X}^{k\backslash x}$.

□

3.4.4. Simplifications

There are two simplification steps which are performed: (1) elimination of vectors which have length one and (2) elimination of independent edges. Both steps reduce the degree of decision vectors and thus improve the reducibility of the PBQP.

The first simplification step can be performed prior to the reduction phase. It removes decision vectors which have only one element. Since there is only one possibility for the decision for such a vector, the vector can be removed from the PBQP. This process is equivalent to splitting a vector into separate vectors for each adjacent edge, which are then reduced by RI reductions.

The second simplification step is performed whenever a cost matrix changes. This is initially done for all cost matrices prior to the reduction phase and after each RII reduction. The second simplification eliminates edges with independent transition costs. Independent transition costs are costs which do not result in a decision dependence between the two adjacent vectors, i.e. the decision of one adjacent vector does not depend on the decision of the other adjacent vector. A simple example for independent transition costs is a zero matrix. In general all matrices which can be reduced to a zero matrix by subtracting a column vector and a row vector are independent.

Definition 12. *Let C be a matrix and $\vec{u}$ and $\vec{v}$ be vectors with n and m elements respectively. The $n \times m$ matrix C is independent iff $C(i,j) = \vec{u}(i) + \vec{v}(j) \ \forall \ i \in 1...n, j \in 1...m$*

The following theorem allows elimination of an independent matrix from the PBQP by adding the vectors $\vec{u}$ and $\vec{v}$ to the adjacent cost vectors of the matrix. Vector $\vec{u}$ is added to the predecessor cost vector and vector $\vec{v}$ is added to the successor cost vector. A proof is given in Appendix A.

Theorem 2. *Let C be an $n \times m$ independent matrix and $\vec{x}$ and $\vec{y}$ boolean decision vectors. Then $\vec{x} \cdot C \cdot \vec{y}^T = \vec{u} \cdot \vec{x}^T + \vec{v} * \vec{y}^T$*

3.5. General Solver

The optimal solver can only be used for PBQPs which are reducible with rules RI and RII. For general PBQPs we must use a general solver which implements heuristics for vectors which are irreducible. In this section we describe two strategies to eliminate irreducible vectors.

3.5.1. Local Minimum

The first method for irreducible vectors is to use an additional rule RN to reduce such vectors. In contrast to rule RI and RII, a decision is made during reduction for the eliminated vector. This decision only depends on the adjacent cost vectors and matrices. Therefore applying rule RN destroys the optimality of the solution. Rule RN is defined as follows for the objective function:

Definition 13. *Let $f^k(X^k)$ be the k^{th} reduced objective function. Let $\vec{x}$ be a decision vector of degree greater than two. Further let $\bar{\vec{x}}$ be the local decision for vector $\vec{x}$. Then*

$$RN(f^k(X^k), \vec{x}) \Rightarrow f^{k+1}(X^{k+1}) = f^k(X^k) \mid \vec{x} = \bar{\vec{x}}$$

Let

$$\vec{c}(i) = \sum_{\vec{y} \in adj(\vec{x})} min(C_{xy}(i,:) + \vec{c}_y)$$

Then the local solution s_x, which is the index of the one-element in $\bar{\vec{x}}$ is defined as $s_x = i_{min}(\vec{c})$.

Setting the reduced vector $\vec{x}$ to the local decision $\bar{\vec{x}}$, the $\bar{\vec{x}} \cdot C_{xy} \cdot \vec{y}^T$ terms in the objective function evaluate to cost vector terms for the adjacent vectors $\vec{\delta}_y \cdot \vec{y}^T$. The resulting cost vectors $\vec{\delta}_y$ are added to the existing cost vectors $\vec{c}_y$ for all adjacent vectors $\vec{y} \in adj(\vec{x})$.

This means that the decision for vector $\vec{x}$ is made as if the adjacent vectors would be disconnected from the remaining PBQP-graph. This wrong assumption leads to a suboptimal solution.

Finally we give the definition of rule RN for the PBQP-graph:

Definition 14. *Let $G^k\langle V, E, w\rangle$ be the k^{th} reduced PBQP graph. Let x be a node. Then*

$$RN(G^k\langle V, E, w\rangle, x) \Rightarrow G^{k+1}\langle V - \{x\}, E - \cup_{y \in \text{adj}(x)}(x, y), w'\rangle$$

The cost function w^{k+1} is identical to cost function w^k except for the adjacent nodes of x.

3.5.2. Recursive Enumeration

To improve the solution of a PBQP containing irreducible vectors, recursive enumeration can be applied. Recursive enumeration has exponential complexity. Therefore it can not be applied to an arbitrary PBQP. But it is possible to limit the number of permutations. After the number of permutations exceeds the limit, reduction rule RN can be used. With this method, a trade off between complexity and exactness can be achieved. In this work we use recursive enumeration for comparing with the optimal result in the SSA-graph matching application. In the register allocator it is used to improve the solution of the general solver.

Enumerating a irreducible vector $\vec{x}$ means that the vector is eliminated and the remaining PBQP is solved with all possible values of $\vec{x}$. So if the solver encounters n irreducible vectors, recursive enumeration has a complexity of $O(m^n)$ where m is the length of the irreducible vectors.

An improvement to recursive enumeration is to detect connected components in the PBQP-graph and solve each connected component separately. Eliminating an irreducible vector may disconnect the PBQP-graph into separate connected components. Therefore the graph separation must be done after each elimination. An example is shown in Figure 3.7. The irreducible node x connects three separate components which also contain one irreducible node each. After eliminating x, recursive enumeration can be continued for each sub-graph separately. Therefore the complexity reduces from $O(m^4)$ to $O(m \cdot (m + m + m)) = O(m^2)$.

3.6. Implementation

This section shows a pseudo code for the implementation of the general solver.

Figure 3.8 lists the procedure *ReduceGraph* that is responsible for the first phase of the solver. The parameter *general* selects whether the solver works as a general or an optimal solver. The procedure returns true if a solution can be calculated and false otherwise (this can only be returned by the optimal solver).

To select reducible nodes in constant time, *buckets* are used. For each node-degree there exists a bucket and a node belongs to bucket d if it has a degree of d. Lines 3

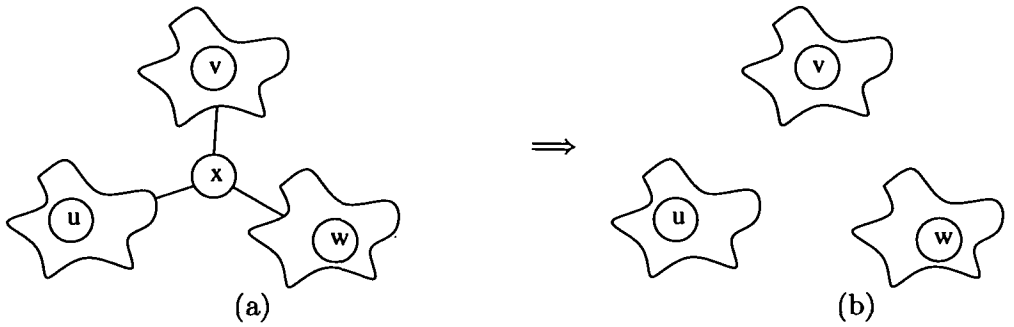


Figure 3.7.: Eliminating irreducible node x splits the graph into three connected components.

to 5 of procedure *ReduceGraph* build up the buckets. After constructing the buckets, nodes are reduced until all nodes are in bucket 0 and the objective function is trivial.

The procedure *ReduceGraph* calls the reduction procedures *RI*, *RII* and *RN*, which are shown in Figure 3.10. Procedure PushVertex(x) pushes node x onto a stack, and removes it from the PBQP-graph. Then, it reorders adjacent nodes in the buckets according to their degrees.

After reducing all nodes, the objective function is trivial and we can determine a solution for nodes which are held in bucket 0. Then, we propagate the solution through the eliminated nodes and reconstruct the graph. The second and the third phase of the dynamic program are given in Procedure *PropagateSolution* of Figure 3.9.

In the following we demonstrate the algorithm on the example shown in Figure 3.2. Figure 3.11 shows the reduction sequence. First node v_1 is eliminated by a RI rule. The additive vector $\vec{\delta_1} = \begin{pmatrix} 7 & 13 \end{pmatrix}$ is added to $\vec{c}_2$ which results in a new $\vec{c}_2^{\,2} = \begin{pmatrix} 12 & 16 \end{pmatrix}$. Then v_3 is reduced by a RII rule (the solver could also select v_2 or v_4 for a RII reduction). The resulting matrix

$$\Delta_3 = \begin{pmatrix} 9 & 7 \\ 10 & 11 \end{pmatrix}$$

is added to C_{23} which results in a new matrix

$$C_{23}^3 = \begin{pmatrix} 10 & 15 \\ 17 & 20 \end{pmatrix}$$

Finally v_2 is eliminated, again by a RI rule. This adds $\vec{\delta_2} = \begin{pmatrix} 22 & 27 \end{pmatrix}$ to $\vec{c}_4$. The resulting cost vector is $\vec{c}_4^{\,5} = \begin{pmatrix} 31 & 28 \end{pmatrix}$. Now the PBQP is trivial – the single node v_4 with degree zero remains. The minimum element $min(\vec{c}_4^{\,5}) = 28$ can be selected to $s_4 = 1$. In the back-propagation phase first the node v_2 is re-inserted into the graph and the minimum element can be selected to $s_2 = 0$. Then back-propagation re-inserts

```
1:  procedure ReduceGraph(general)
2:  begin
3:    forall nodes x ∈ V do
4:      insert x into bucket deg(x)
5:    endfor
6:    while nodes left (buckets ≥ 1) do
7:      if a nodes x exists in bucket 1 then
8:        RI(x);
9:      elsif a nodes x exists in bucket 2 then
10:       RII(x);
11:     elsif general
12:       select node x
13:       RN(x);
14:     else
15:       return false;
16:     endif
17:   endwhile
18:   return true;
19: end
```

Figure 3.8.: Reduction

```
1:  procedure PropagateSolution
2:  begin
3:    forall nodes x in bucket 0 do
4:      s_x := i_min(c_x)
5:    endfor
6:    while reducible stack not empty do
7       pop node x from reducible stack
8:      c := c_y;
9:      forall nodes y ∈ adj(x) do
10:       c := c + C_yx(s_y, :);
11:     endfor
12:     s_x = i_min(c);
13:   endwhile
14: end
```

Figure 3.9.: Back propagation

```
1:  procedure RI(x)
2:  begin
3:    {y} := adj(x);
4:    for i:=1 to |c_y| do
5:      δ(i) := min(C_yx(i,:) + c_x);
6:    endfor
7:    c_y := c_y + δ;
8:    PushVertex(x)
9:  end
10: procedure RII(x)
11: begin
12:   {y, z} := adj(x);
13:   for i := 1 to |c_y| do
14:     for j := 1 to |c_z| do
15:       Δ(i, j) := min(C_yx(i,:) + C_zx(j,:) + c_x)
16:     endfor
17:   endfor
18:   if (y, z) ∈ E then
19:     C_yz := C_yz + Δ
20:   else
21:     add edge (y, z)
22:     C_yz := Δ;
23:   endif
24:   PushVertex(x)
25: end
26: procedure RN(x)
27: begin
28:   for i := 1 to |c_x| do
29:     c(i) := 0;
30:     forall nodes y ∈ adj(x) do
31:       c(i) := c(i)+
32:         min(C_xy(i,:) + c_y)
33:     endfor
34:   endfor
35:   s_x = i_min(c);
36:   forall nodes y ∈ adj(x) do
37:     c_y := c_y + C_xy(s_x, :)
38:   endfor
39:   PushVertex(x)
40: end
```

Figure 3.10.: Reduction procedures

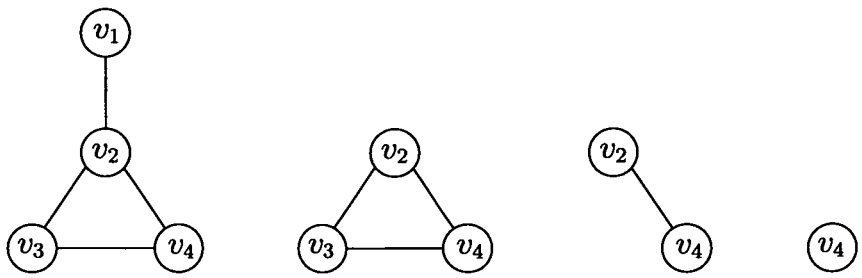


Figure 3.11.: Reduction sequence of the example

v_3 and selects the minimum element to $s_3 = 2$. Finally node v_1 is re-inserted and the element $s_1 = 1$ is selected. The solution of the example PBQP is $S = \langle 1, 2, 0, 1 \rangle$.

3.6.1. Complexity of the Solver

Our algorithm partitions nodes in four subsets of V: (1) a partition V_0 of nodes that are solved trivially in the objective function, (2) a partition V_1 of nodes reduced by rule RI, (3) a partition V_2 of nodes reduced by rule RII, and (4) a partition V_3 of nodes reduced by rule RN. All four partitions exhibit a different time complexity. We use n_{V_i} to denote the number of elements in the i^{th} partition, where their sum is equal to n, i.e. the number of nodes. Then, the time complexity is given by

$$O(n_{V_0} \cdot m + n_{V_1} \cdot m^2 + n_{V_2} \cdot m^3 + n_{V_3} \cdot m^3) \tag{3.21}$$

where m is the length of the decision vectors. The length can vary for each decision vector. This depends on the PBQP application. For example in register allocation the length is the number of CPU registers plus one. This time complexity equation is given for the basic PBQP solver, which - in this form - is used by the SSA-graph matching and the register allocation optimizations. Addressing mode selection implements a sparse matrix representation where the complexity is computed differently. But for all variants of the PBQP solver the factor m (for addressing mode selection this is not the length) is small compared to the number of nodes n and bound to an upper limit. Therefore for large n the complexity is linear with n.

4. Code Generation

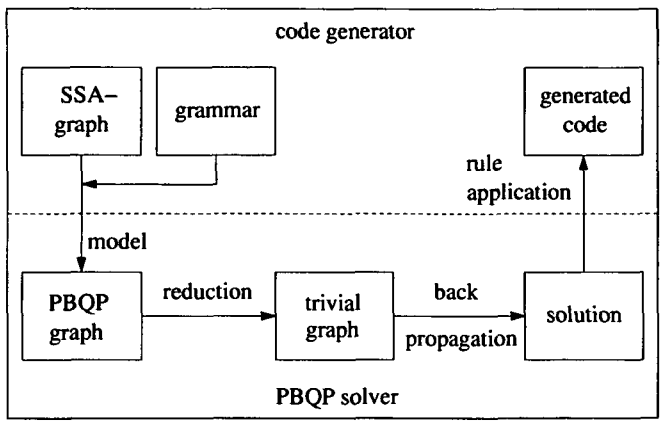


Figure 4.1.: SSA-graph matching overview

4.1. Overview

Code generation translates the high level IR to a low level IR. Usually a matching algorithm is employed on the high level IR to find an optimal match (see Section 2.2). Tree pattern matching is a widely used technique [2, 7, 22, 19], because it is fast and yields optimal results. But most high level IRs have DAG structures or cyclic structures. The general matching problem on DAGs or directed graphs is NP-complete [55].

We introduce a method for mapping the problem of code generation to a PBQP. With the PBQP we can model pattern matching for DAGs and even for cyclic graphs. It enables us to take the computational flow of a whole function into account. We call the matcher a *SSA-matcher*, because the matching algorithm is performed on the SSA-graph of a function. For representing the computational flow, the SSA-graph is used, which combines data flow trees (DFT) and def-use relations of a function.

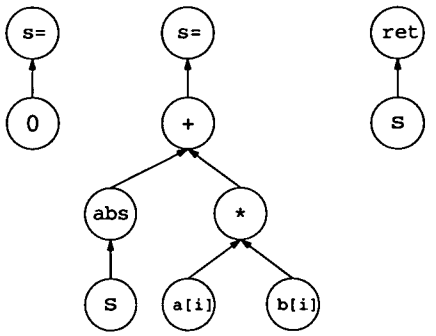
The SSA-matcher is defined by an ambiguous grammar, just like a tree pattern matcher. The grammar definition consists of production rules and a set of non-terminals. A rule is of the form $\mathtt{nt} \rightarrow \mathit{pattern}$, where `nt` is a non-terminal and *pattern* describes a tree of terminals and non-terminals. A node in the pattern tree is either a non-terminal or a terminal $P[s_1, \ldots, s_n]$, where s_i are the child nodes in the pattern tree. A terminal may also have no child nodes, i.e. a leaf in the DFT.

In addition, production rules have cost terms and code templates. Cost terms are used to find the derivation with minimal overall costs.

The basic concept of our SSA-graph matching algorithm is shown in Figure 4.1. First, the SSA-graph with its ambiguous grammar is mapped to PBQP. Second, the PBQP solver computes the grammar derivation with minimal costs. Third, based on the grammar derivation, code is produced.

```
    int f(short *a, short *b)
    {
(1)   int s = 0;
      loop(i) {
(2)     s = abs(s) + a[i] * b[i];
      }
(3)   return s;
    }
```

(a)



(b)

Figure 4.2.: Example source code (a) and data flow trees (b)

Note that the PBQP solver consists of two phases: In the first phase the graph is reduced until a trivial solution remains. In the second phase the solution is back-propagated. The two phases of the PBQP solver are very similar to the two phases of the dynamic programming algorithm of tree pattern matchers. In fact, if the PBQP-graph is a tree, the two algorithms are almost identical. The significant difference is that a tree pattern matcher decides between non-terminals whereas the PBQP solver decides between rules.

The quality of the solution, obtained from the PBQP solver, depends on the reducibility of the input graph. We have implemented our approach in a production DSP compiler. Our experiments show that the SSA-graph is reducible in most cases, which makes the PBQP method very suitable for the matching problem. Only for a negligible number of SSA-graph nodes (see Section 4.6), can no optimal solution be computed and heuristics must be applied. Consequently, the PBQP solution is nearly optimal.

Compared to the result of a tree pattern matcher, the SSA-matcher produces significantly faster code. This shows the importance of matching the whole SSA-graph of a function rather than the separate data flow trees.

4.2. Motivation

The example in Figure 4.2(a) shows a typical DSP code. The elements of two vectors `a` and `b` are multiplied and the absolute value of the last iteration is added. The example focuses on the accumulator variable `s` which occurs in the three statements (1), (2) and (3). The loop control code is only shown in pseudo code because it is not relevant for this example.

Let us assume that the computations for variable `s` are performed in fixed point

arithmetic on a DSP processor. In contrast to standard processors, DSP processors have multiplication units that perform a multiplication by shifting the result by one bit to the left. However, for compilers it is difficult to exploit this shift. Without knowing the context of the computation an additional shift operation is needed to re-adjust the multiplication result.

For obtaining faster code, computations inside the loop should be performed with a shifted result by one bit to the left. Otherwise an additional shift-operation would be introduced inside the loop and would worsen the runtime. Since the return statement requires an un-shifted value, a shift operation has to be inserted prior to the return statement outside of the loop.

The non-terminals in the grammar describe sub-graphs and the productions describe how non-terminals are derived and at which costs. The non-terminals are used to express architectural computation properties. A non-terminal defines how a value is stored and how it is interpreted. As the grammar is ambiguous, there are multiple ways to derive a sub-graph to different non-terminals. So there are different ways to store a value resulting from a sub-graph. The following list contains some examples of non-terminal interpretations.

- The non-terminal specifies the register class. For example there are different non-terminals for accumulator registers and address registers.
- The non-terminal specifies the interpretation of the unused high significant bits in a register. This is used for values which are smaller than the register size. There may be non-terminals for sign-extended, zero-extended or garbage-extended values.
- The non-terminal specifies the location of the value inside the register, if the value is smaller than the register size. This is used in our example. There may be non-terminals for values aligned at the least significant bit or values shifted by a specific number of bits.
- The non-terminal specifies the arithmetic interpretation of the value in a register. There may be non-terminals for negated values or inverted values.

For our running example we define a grammar which is shown in Figure 4.3. A production has a left-hand side and a right-hand side, i.e. `nt` $\rightarrow$ *`pattern,cost,code`*.

On the left-hand side a non-terminal specifies the result of the computation. On the right-hand side there is a *pattern* that consists of terminals and non-terminals. In addition the matching *cost* and the *code* template are given (separated by commas). Note that the code templates are only shown for better understanding of the rules, but they do not influence the matching algorithm.

In our grammar the shift property of the multiplication is represented by two non-terminals: `reg` and `sreg`. Nonterminal `reg` represents an un-shifted value whereas

(1) `reg → const(0)[],1,r=0`
(2) `sreg → const(0)[],1,r=0`
(3) `reg → +[reg,reg],3,r=r+r`
(4) `sreg → +[sreg,sreg],3,r=r+r`
(5) `reg → abs[reg],2,r=abs(r)`
(6) `sreg → abs[sreg],2,r=abs(r)`
(7) `sreg → *[reg,reg],4,r=r*r`
(8) `reg → load[ptr],5,r=*ptr`
(9) `top → ret[reg],1,ret`
(10) `reg → sreg,1,r=r>>1`
(11) `sreg → reg,1,r=r<<1`
(12) `reg → s[],0`
(13) `top → s=[reg],0`

Figure 4.3.: Production rules

`sreg` represents a value which is shifted left by one bit. For example the multiplication rule requires two un-shifted input values and produces a shifted value (Rule 7). Plus operations and absolute value operations can be performed with un-shifted values (Rules 3 and 5) or shifted values (Rules 4 and 6). The constant 0 can either be loaded as shifted or un-shifted value (Rules 1 and 2). The memory load is represented by Rule 8 and can only produce an un-shifted value. Return statements require un-shifted values to preserve program semantics (Rule 9). Rules 10 and 11 are chain-rules that convert a shifted value to an un-shifted value and vice versa.

In the example three statements contain the accumulator variable `s`. Figure 4.2(b) shows the DFTs of the three statements which are processed by a typical tree pattern matcher. Two additional rules are required to match the DFTs: a rule to match variable uses (Rule 12) and a rule to match variable definitions (Rule 13). But these rules can only exist for a single non-terminal (either `reg` or `sreg`). Otherwise occurrences of a variable in various places would be interpreted differently. This means that with a tree pattern matcher the non-terminals for variables must be selected before matching.

4.3. The SSA-Graph

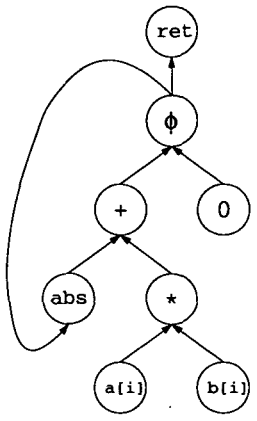
To overcome the limitations of a tree pattern matcher we extend the scope of the matcher to SSA-graphs [27]. The base for SSA-graphs is the *static single assignment* form [15]. The essential idea behind SSA is that each use has only a single definition. If there are multiple definitions for a use in the non-SSA form, in the SSA form a ϕ-term is inserted. Figure 4.4(a) shows the SSA form of our example program. It contains a ϕ-term for `s` at the loop head where the definition of the initialization and the definition of the computation of the last iteration are merged.

A SSA-graph describes the flow of computation for a whole function. Basically, the data structure combines the data flow trees (DFT) with def-use relations. For our running example the SSA-graph is shown in Figure 4.4(b).

The SSA-Graph $S\langle V, E\rangle$ is a graph where set of nodes V contains the operators in

```
     int f(short *a, short *b)
     {
(1)    int s1 = 0;
       loop(i) {
         s2 = φ(s1, s3)
(2)      s3 = abs(s2) + a[i] * b[i];
       }
(3)    return s2;
     }
```

(a)



(b)

Figure 4.4.: SSA-form (a) and SSA-graph (b) of the running example

the DFTs, including the ϕ-terms of the SSA form. The edges in the set E represent the flow of data between the operators. There is an edge between two operators if the result of an operator is an input operand of another operator. So the outgoing edges from an operator indicate data dependencies to other nodes which use the result.

Note that SSA-graphs do not contain explicit nodes for variable uses (`s`) and variable definitions (`s=`). In contrast to classical approaches which use DAG and tree representations of the computations, cycles are possible in the SSA-graphs.

For our running example we have several node types in the SSA-graph. E.g., plus operations(`+`), absolute value operations(`abs`), multiplications(`*`), element access(`a[i]`), ϕ-nodes, constant nodes, and a return node(`ret`) for the return statement. The incoming edges specify the inputs of the computation. For example, the multiplication node has two incoming edges. One edge is from the operand `a[i]` the other edge is from the operand `b[i]`. In the SSA-graph all nodes, except the return node, pass their result on to other nodes.

The grammars used for matching SSA-graphs are similar to grammars used by tree pattern matchers. As the SSA-graph does not contain explicit nodes for variable uses and variable definitions, no rules are required to match such nodes. Instead a grammar for SSA-graph matching must contain rules for matching ϕ-terms. In the example grammar the Rules (12) and (13) are no longer needed. Instead we need following rules to match the ϕ-term nodes.

(14) `reg` $\rightarrow$ ϕ`[reg,...,reg],0`
(15) `sreg` $\rightarrow$ ϕ`[sreg,...,sreg],0`

In contrast to matching rules of other nodes, ϕ-nodes do not emit any code. They

are only needed to match non-terminals of the same type. Rules 14 and 15 handle shifted and un-shifted values respectively for ϕ-nodes. As these rules do not generate any code, they do not have a code template.

4.4. Mapping to a PBQP

In this section we describe the mapping of the SSA-graph matching problem to a PBQP. The mapping from the SSA-graph matching problem is done in three steps: (1) construct the PBQP-graph based on the SSA-graph, (2) determine cost vectors of nodes, and (3) determine cost matrices of edges.

4.4.1. Normal Form

For performing the mapping, the grammar has to be transformed to normal form [7]. A grammar is in normal form if there are only production rules which are either base or chain rules. A base rule has the form $nt_0 \rightarrow P[nt_1, \ldots, nt_n]$ where nt_i are non-terminals and P is a terminal symbol. A chain rule is given by $nt_1 \rightarrow nt_2$ where on the left-hand side and on the right-hand side of the production are non-terminals. Production rules, which are neither chain rules nor base rules, can be decomposed into base and chain rules by introducing a new non-terminal.

For each pattern sub-tree (i.e. not the root of a pattern tree) of the form P or $P[S]$, where P is a terminal and S are the child nodes, following transformation is done: A new non-terminal nt_P is introduced and a new production rule $nt_P \rightarrow P[S]$ is added. The original occurrence of the sub-tree P is replaced by the non-terminal nt_P. The cost term of the new generated rules are set to zero, because the costs are already counted in the original rule. This process is repeated until all rules in the grammar are base or chain rules.

For example rule `reg` $\rightarrow$ `+[reg, *[reg, reg]],2` is neither a base nor a chain rule. By introducing a new non-terminal nt we can decompose the rule in `reg` $\rightarrow$ `+[reg, nt],2` and `nt` $\rightarrow$ `*[reg, reg],0`.

4.4.2. The PBQP-Graph

The main idea is that the PBQP-graph is equivalent to the SSA-graph (step 1). Nodes in the SSA-graph are nodes in the PBQP-graph and vice versa. Similarly, edges of the PBQP-graph are edges in the SSA-graph.

$$G(V, E, w) = S(V, E)$$

where $S(V, E)$ is the SSA-Graph and $G(V, E, w)$ is the PBQP-graph. The weighting function w describes the cost vectors and matrices, which are determined in step 2 and 3.

The SSA-matcher can be configured to different optimization goals, e.g. optimize for minimal execution time or optimize for minimal code size. The parameterization is done with an execution weighting function *ew*, which is defined for all nodes of the PBQP-Graph.

When optimizing for minimal code size the function should yield 1 for all nodes, because a code size penalty is independent of the point of code insertion. When optimizing for minimal execution time, the execution weighting function yields the dynamic execution count, which can be obtained by profiling. In this case the weight function for nodes describes how often the operation of the node is executed.

Definition 15. *Let $v \in V$ be a node in the PBQP-Graph. Then $ew(v)$ is is the dynamic execution count of the basic block in which the operation of v is executed or ∞ if v is a ϕ-term.*

For ϕ-terms it yields ∞, which is necessary for constructing the cost matrices in Section 4.4.4. With the help of the execution weighting function, the SSA-matcher prefers to generate code in rarely executed parts instead of heavily executed parts.

For our example we assume that the loop is executed 10 times. This yields an execution weight value of 10 for all nodes inside the loop, i.e. all nodes except `0` and `ret`.

4.4.3. Defining Cost Vectors

For each node in the SSA-graph there are several base rule options. The number of these alternatives determines the size of the boolean vector for this node. So each vector element corresponds to a base rule. The cost vector of this node is derived from the base rule costs of the node.

For each node we enumerate all applicable base rules. The cost vector for the node is the vector of rule costs scaled by the execution weight function.

Let $R_i = \{r_1^i, .., r_k^i\}$ be the set of matching rules for node v_i and let $cost(r)$ be the cost of rule r. Then we can construct the cost vector $\vec{c}_i$ according Equation 4.1.

$$\vec{c}_i(j) = cost(r_j^i) \cdot ew(i) \qquad \forall r_j^i \in R_i \tag{4.1}$$

Note that we define $0 \cdot \infty = 0$. So for ϕ-terms, where the rule costs are 0 and the execution weighting function is ∞, the cost vector element is 0.

For our example all matching rules for its nodes are listed in Figure 4.5. As we can see that for some nodes we have only one alternative which maps to a boolean decision vector with only one element. For others we have two alternatives. Therefore, the size for their boolean decision vectors is two. The cost vectors of the matching rules are given in Figure 4.6(a). For nodes inside the loop the cost elements are multiplied by 10 since we assume that the loop is executed 10 times. For nodes outside the loop the execution weight function yields one.

$R_{\texttt{ret}} = \{$ `top` $\rightarrow$ `ret[reg]` $\}$
$R_{0} = \{$ `reg` $\rightarrow$ `const(0)[]`, `sreg` $\rightarrow$ `const(0)[]` $\}$
$R_{+} = \{$ `reg` $\rightarrow$ `+[reg,reg]`, `sreg` $\rightarrow$ `+[sreg,sreg]` $\}$
$R_{\texttt{abs}} = \{$ `reg` $\rightarrow$ `abs[reg]`, `sreg` $\rightarrow$ `abs[sreg]` $\}$
$R_{*} = \{$ `sreg` $\rightarrow$ `*[reg,reg]` $\}$
$R_{\texttt{a[i]}} = R_{\texttt{b[i]}} = \{$ `reg` $\rightarrow$ `load[ptr]` $\}$
$R_{\phi} = \{$ `reg` $\rightarrow \phi$`[reg, reg]`, `sreg` $\rightarrow \phi$`[sreg, sreg]` $\}$

Figure 4.5.: Matching rule sets of the running example

4.4.4. Defining Cost Matrices

The last step in the PBQP definition is to determine the transition cost matrices for all edges in the graph, which express the chain rule costs between two operators.

For convenience we define a function *chaincost*, which is used to get chain costs between two rules rather than between two non-terminals.

Definition 16. *Let* $r = nt_0^r \rightarrow P[nt_1^r, .., nt_k^r]$ *and* $s = nt_0^s \rightarrow Q[nt_1^s, .., nt_l^s]$ *be base rules. Then,* $\text{chaincost}(r, s, i) = c$*, where* c *are minimal costs of all chain rule derivations from* nt_0^r *to* nt_i^s*. If there is no chain rule derivation from* nt_0^r *to* nt_i^s*, then* $c = \infty$*.*

Function *chaincost*(r, s, i) yields the chain costs between the result non-terminal of rule r and the i^{th} source non-terminal of rule s. For chain costs between two identical non-terminals we have zero costs. The cost between two different non-terminals depends on whether a derivation with chain rules exists. If there exists at least one derivation, the chaining costs are determined by the derivation with minimal total cost. If no derivation exists, the transition is prohibited and the chaining costs are ∞. The minimal cost derivation is computed with the same algorithm as used in tree pattern matchers [56].

The nodes in the SSA-Graph represent operations which have input operands and a result. The input operands are represented by the incoming edges of a node. However, graphs do not define an order for incoming edges which is required for constructing the cost matrices. To overcome this problem, we define a mapping function *opnum*$(\langle p, s\rangle)$ that determines the index of the operand of the edge $\langle p, s\rangle$ in the expression tree.

Definition 17. *Let* $s \in V$ *be a node in the PBQP-Graph and* $\langle p, s\rangle \in E$ *be a predecessor edge of* s*. Then* $\text{opnum}(\langle p, s\rangle) = i$ *where* i *is the index of the operand represented by* $\langle p, s\rangle$*, of the operator represented by* s*.*

A matrix of an edge contains the costs of a transition between the non-terminals of two adjacent rules. The matrix C_{ij} defines the costs of applying chain rules from the result non-terminal of the predecessor rule r_i to the source non-terminal of the

successor rule r_j. The selection of the source non-terminal in the successor rule pattern is determined by the *opnum* function for the edge.

Based on functions *chaincost* and *opnum*, the cost matrices of edges in the PBQP-graph are computed. The elements of a cost matrix are given according to Equation 4.2.

$$C_{\langle p,s\rangle}(i,j) = chaincost(r_i^p, r_j^s, opnum(\langle p,s\rangle)) \cdot ew(s) \qquad (4.2)$$
$$\forall r_i^p \in R_p, r_j^s \in R_s$$

where $\langle p,s\rangle$ is the edge between nodes p and s. The matching rules of nodes p and s are denoted by r_i^p and r_j^s, respectively.

The chain rule costs are scaled by the execution weight successor node, because chain rule code is generated at the input operands of operators. Sometimes it is advantageous to generate chain rule code for the result of an operator, e.g. if the result has multiple uses or if the use is in a basic block which has a higher execution frequency than the basic block of the operator. This can simply be achieved by inserting a unary dummy node after such operators. Then the chain rule code can be generated at the input operand of the dummy node.

Note that for a ϕ-term node, the function *ew* yields ∞. This disables the generation of chain rule code before ϕ-term statements.

For our example the cost matrices are given in Figure 4.6(b). Matrix $C_{\langle \texttt{abs},+\rangle}$ contains a zero diagonal, the remaining elements are 10. Both the `abs` and `+` nodes have two rules, where the first rules only contain `reg` non-terminals and the second rules only contain `sreg` non-terminals. The transition costs between the first rule of `abs` and first rule of `+` are the chain rule costs of deriving `reg` from `reg`. Obviously this is zero. The same holds for the transition costs between the second rules. All other transitions need a chain rule from `reg` to `sreg` or vice versa. The rule costs for these chain rules are one, which is weighted by 10 (the execution count of the loop).

4.5. Solving the PBQP

In our example all nodes, which have only one matching rule, can be eliminated by simplification. These nodes are `ret`, `*`, `a[i]` and `b[i]`. With the first simplification step the cost vectors of ϕ-nodes and `+` change to the following values:

$$\vec{c}_+ = (\ 30\ \ 30\ ) + (\ 10\ \ 0\ ) = (\ 40\ \ 30\ )$$
$$\vec{c}_\phi = (\ 0\ \ 0\ ) + (\ 0\ \ 1\ ) = (\ 0\ \ 1\ )$$

$$\vec{c}_{\texttt{ret}} = (\ 1\ )$$
$$\vec{c}_{0} = (\ 1\ \ 1\ )$$
$$\vec{c}_{+} = (\ 30\ \ 30\ )$$
$$\vec{c}_{\texttt{abs}} = (\ 20\ \ 20\ )$$
$$\vec{c}_{*} = (\ 40\ )$$
$$\vec{c}_{\texttt{a[i]}} = \vec{c}_{\texttt{b[i]}} = (\ 50\ )$$
$$\vec{c}_{\phi} = (\ 0\ \ 0\ )$$

(a)

$$C_{\langle \texttt{a[i]},* \rangle} = C_{\langle \texttt{b[i]},* \rangle} = (\ 0\ )$$
$$C_{\langle *,+ \rangle} = (\ 10\ \ 0\ )$$
$$C_{\langle 0,\phi \rangle} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$
$$C_{\langle \texttt{abs},+ \rangle} = C_{\langle +,\phi \rangle} = C_{\langle \phi,\texttt{abs} \rangle} = \begin{pmatrix} 0 & 10 \\ 10 & 0 \end{pmatrix}$$
$$C_{\langle \phi,\texttt{ret} \rangle} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

(b)

Figure 4.6.: Cost vectors (a) and transition costs (b) of the example

Figure 4.7 shows the reduction sequence of the example graph. The `*`, `a[i]`, `b[i]` and `ret` nodes are already eliminated by simplification, because only a single rule can be matched on these nodes. The remaining graph contains one node with degree one, i.e. node `0`. In the first step it is eliminated by the RI reduction. This increments the cost vector of the ϕ-node to (1 2). Three nodes with degree 2 remain (ϕ, `+` and `abs`). One of them - in this example the `abs` node - is eliminated by applying the RII reduction. The resulting edge of the reduction has a cost matrix of

$$\Delta = \begin{pmatrix} 20 & 30 \\ 30 & 20 \end{pmatrix}$$

It is combined with the existing edge between ϕ and `+`, which results in a new cost matrix

$$C_{\langle \phi,+ \rangle} = \begin{pmatrix} 20 & 40 \\ 40 & 20 \end{pmatrix}$$

In the last step the ϕ-node can be eliminated with the RI reduction which results in a cost vector of (61 52) for the remaining node `+`. It has degree zero and the second rule (`sreg` → `+[sreg,sreg]`) can be selected, because the second vector element (which is 52) is the element with minimal costs. Because no RN reduction had to be applied for the example graph, the solution of this PBQP is optimal.

After reduction, only nodes with degree zero remain and the rules can be selected by finding the index of the minimum vector element. The rules of all other nodes can be selected by reconstructing the PBQP-graph in the reverse order of reduction. In each reconstruction step one node is re-inserted into the graph and the rule of this node is selected. Selecting the rule is done by choosing the rule with minimal costs for the node. This can be done, because the rules of all adjacent nodes are already known.

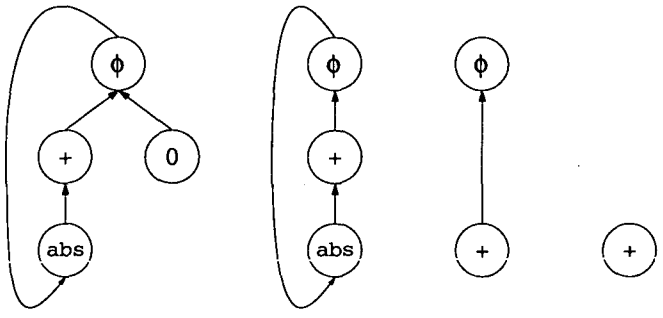


Figure 4.7.: Reduction sequence of the running example

```
(1)   _f:   r0 = 0
(2)         loop {
(3)                 r1 = *ptr1
(4)                 r2 = *ptr2
(5)                 r3 = r1 * r2
(6)                 r0 = abs(r0)
(7)                 r0 = r0 + r3
(8)         }
(9)         r0 = r0 >> 1
(10)        ret
```

Figure 4.8.: The resulting code

The back-propagation process for our example graph reconstructs the ϕ-node. The second rule is selected for this node (`sreg` $\rightarrow$ ϕ`[sreg, sreg]`). Then the `abs` and `0` nodes are re-inserted, with a rule selection of `sreg` $\rightarrow$ `abs[sreg]` and `sreg` $\rightarrow$ `const(0)[]`, respectively. The nodes `ret`, `*`, `a[i]` and `b[i]` need not be reconstructed, because the first (and only) rule has already been selected in the simplification phase for these nodes.

The solution of the PBQP yields the rule selections for the SSA-graph nodes. The code is generated by applying the code generation actions of the selected rules. This works in the same way as the code generation in tree pattern matchers. As the SSA-graph does not contain any control flow information, the places where the code is generated must be derived from the input program. So the code for a specific node is generated in the basic block which contains the operation of the node. Chain rule code is inserted at the input operands of the successor operator. The order of code generation within a basic block is also defined by the statement order and operator order in the input program. As with tree pattern matchers, the order can be directly derived by traversing the statement DFTs.

Figure 4.8 shows the resulting code after register allocation (for sake of clarity the loop control code and addressing code are not shown in this figure). As we can see in the generated code, inside the loop the addition operation and the `abs` function is performed with a shifted value. Prior to the return statement the value of variable `s` is converted to an un-shifted value.

4.6. Experimental Results

We have integrated the SSA-matcher into the Atair C-Compiler for the NEC μPD77050 DSP family. The μPD77050 is a low-power DSP for mobile multimedia applications that has VLIW features [39, 33]. Seven functional units (two MAC, two ALU, two load/store, one system unit) can execute up to four instructions in parallel. The register set consists of eight 40 bit general purpose registers and eight 32 bit pointer registers.

The grammar contains 724 rules and 23 non-terminals. The non-terminals select between address registers or general purpose registers. For the general purpose registers, there are separate non-terminals for sign-extended values and non-sign-extended values and there are various non-terminals which place a smaller value at different locations inside a 40 bit register.

We have conducted experiments with a number of DSP benchmarks. The first group of benchmarks contains three complete DSP applications: AAC (advanced audio coder), MPEG, and GSM (gsm half rate). All three benchmarks are real-world applications that contain some large PBQP-graphs. The second group of benchmarks are DSP-related algorithms of small size. These kind of benchmarks allow the detailed

analysis of the algorithm for typical loop kernels of DSP applications. All benchmarks are compiled "out-of-the-box", i.e. the benchmark source codes are not rewritten and tuned for the CC77050 compiler.

In Table 4.1 the number of the graphs "Graphs num." and the sizes of the graphs are given. In the "num." columns the accumulated values over the whole benchmark are shown and in the "max." columns the maximum value over all graphs is given. The total number of cost vector elements in the graph and the maximum number of cost vector elements for each node is shown in the last two columns. The number of cost vector elements is the number of matching rules of a node. These numbers depend on the used grammar. With our test grammar a maximum of 62 rules per node occurs in the graphs.

An important question when using a PBQP solver arises regarding the quality of the solution. It highly depends on the density of the PBQP-graphs. If a graph can be reduced with RI and RII rules, the solution is optimal. Figure 4.9 shows the distribution of reductions. 31% of nodes can be eliminated by simplification, because they are trivial, i.e. only a single rule can match these nodes. An important observation is that only a small fraction (less than 1%) of all nodes are RN nodes. Therefore the solutions obtained from the PBQP solver are near optimal. The distribution of nodes in Figure 4.9 also shows the structure of the PBQP-graph: The fraction of degree zero nodes "R0" indicates the number of independent sub graphs in the SSA-graphs, i.e. a third of the nodes form own sub-graphs. RI nodes are nodes which are part of a tree, whereas RII and RN nodes are part of a more complex subgraph. Simplification can eliminate 37% of all edges, because of independent transition costs.

An effective way to improve the solution is to recursively enumerate the first RN nodes in a graph. In many graphs only a few RN nodes exist and by moderate enumeration an optimal solution can be achieved. We have performed our benchmarks in three different configurations: (1) reducing all RN nodes with heuristics "H", (2) enumerate the first 100 permutations before applying heuristics "E 100" and (3) enumerate the first two million permutations "E 2M" before applying heuristics. The third configuration can yield the optimal solution in almost all cases. It is used to compare the other configurations against the optimum. Table 4.2 shows the percentages of optimally solved graphs and optimally reduced nodes in each configuration. The left columns "gropt" show the percentage of optimally solved graphs in each benchmark, the right columns "rnopt" show the percentage of RN nodes, which are reduced by enumeration and do not destroy the optimality of the solution. A value of 100% is also given if there are no RN nodes in a benchmark. In the first configuration "H", no enumeration was applied, therefore all RN nodes are reduced with the heuristics (0% in the "H/rnopt" column or 100% if there are no RN nodes in a benchmark). Even without enumeration most of the graphs "H/gropt" can be solved optimally. The results of the second configuration "E 100" show that with a small number of permutations almost all graphs "E 100/gropt" and a majority of RN nodes "E 100/rnopt"

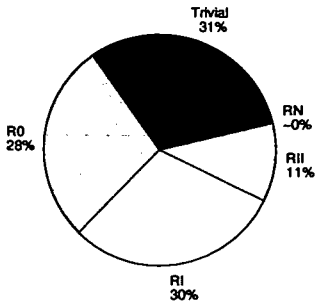


Figure 4.9.: Reduction statistics

can be solved optimally.

For the performance evaluation, we compare the SSA-graph matcher with a conventional tree pattern matcher, using the same grammar. For the tree-pattern matcher we had to make a pre-assignment of non-terminals to local variable definitions and uses. We assigned the most reasonable non-terminals to local variables, e.g. a pointer non-terminal to pointer variables, a register low-part non-terminal to 16 bit integer variables, etc. This is how a typical tree pattern matcher would generate code. It is equivalent to the approach of splitting the graph into trees. The performance improvements for all three configurations is shown in Figure 4.10. The configuration which enumerates 100 permutations gives a (marginal) improvement in just one benchmark (AAC). And the near optimal configuration does not improve the result anymore. This indicates that the heuristic for reducing RN nodes is sufficient for this problem. The performance improvement for the small benchmarks is higher than for the large applications, because the applications contain much control code beside the numerical loop kernels.

The compile time overhead for the three DSP applications is shown in Table 4.3 (the compile time overhead for the small DSP algorithms is negligible and therefore not shown). The table compares the total compile time of two compilers, the first with SSA-graph matching, the second with tree pattern matching. The table compares the compile time overhead of the SSA-graph matching compiler to the tree matching compiler in percent for all three configurations. The overhead of the first two configurations ("H" and "E 100") is equivalent. This means that it is feasible to allow a small number of permutations for RN nodes.

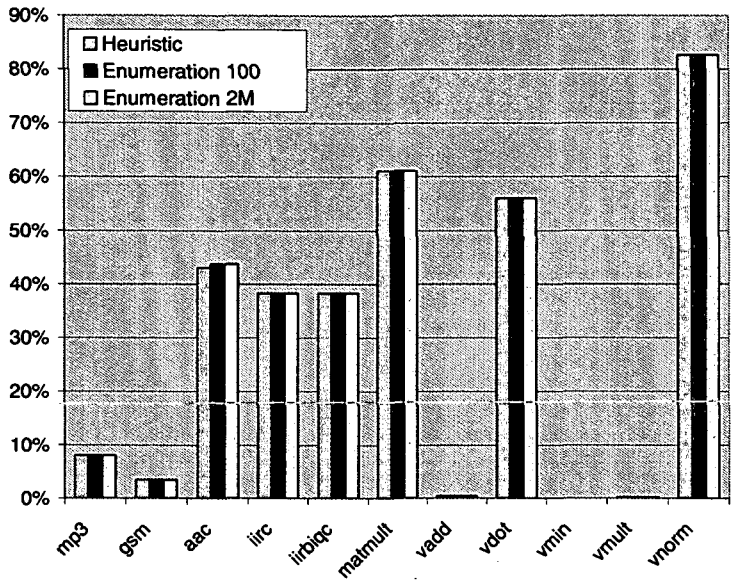


Figure 4.10.: Performance improvement

Benchmark	Graphs num.	Nodes num.	Nodes max.	Edges num.	Edges max.	vec. elements num.	vec. elements max.
mp3	60	37197	8491	40321	8854	556819	62
gsm	129	71376	24175	76884	26154	1138903	62
aac	71	25875	13093	26886	13523	405220	62
iirc	1	263	263	271	271	4877	62
iirbiqc	4	986	493	1002	501	17760	62
matmult	2	640	320	656	328	12182	62
vadd	2	244	122	242	121	4390	33
vdot	2	268	134	268	134	4812	62
vmin	2	306	153	304	152	5652	33
vmult	2	276	138	274	137	4976	62
vnorm	2	252	126	252	126	4590	62
sum/max	277	137683	24175	147360	26154	2160181	62

Table 4.1.: Problem sizes

Benchmark	H		E 100		E 2M	
	gropt	rnopt	gropt	rnopt	gropt	rnopt
mp3	83.33	0.00	98.33	54.76	98.33	73.81
gsm	93.02	0.00	99.22	82.35	100.00	100.00
aac	91.55	0.00	98.59	75.00	100.00	100.00
iirc	0.00	0.00	100.00	100.00	100.00	100.00
iirbiqc	50.00	0.00	100.00	100.00	100.00	100.00
matmult	100.00	100.00	100.00	100.00	100.00	100.00
vadd	100.00	100.00	100.00	100.00	100.00	100.00
vdot	100.00	100.00	100.00	100.00	100.00	100.00
vmin	100.00	100.00	100.00	100.00	100.00	100.00
vmult	100.00	100.00	100.00	100.00	100.00	100.00
vnorm	100.00	100.00	100.00	100.00	100.00	100.00

Table 4.2.: Optimal graph and node reductions in percent

Benchmark	H	E 100	E 2M
mp3	14	14	4252
gsm	6	6	7
aac	3	3	349

Table 4.3.: Compile time overhead in percent

5. Addressing Mode Selection

5.1. Overview

Embedded CPU architectures, like DSP processors, have addressing generation units facilitating a variety of addressing modes. In this chapter we introduce a compiler optimization called *addressing mode selection* (AMS) that selects the optimal addressing modes for address registers in an input program. AMS is a complementary optimization to *offset assignment* and *address register assignment*. The AMS is performed after the offsets and address registers have already been assigned.

The AMS is a representative problem for all kinds of mode optimization problems. The AMS problem formulation can be easily adapted for other mode optimization problems, e.g. the problem of optimally setting CPU mode registers. Whereas most mode optimization problems have to decide between a small number of modes (e.g. mode is on and mode is off), the additional difficulty of the AMS is that it has to decide between a large number of values. Therefore the PBQP solver for the AMS uses a sparse matrix representation, which is not necessary for other mode optimization problems.

The AMS turns out to be a hard problem mainly for two reasons: First, even if we assume a very simple architecture, the AMS problem is NP-complete [52]. Second, in practice the addressing modes of embedded system processors are very diverse and non-homogeneous. The PBQP allows us to construct a flexible cost model in which all kinds of addressing modes can be handled. The most important aspect is that the PBQP solver can yield an optimal result for nearly all input programs.

5.2. Motivation

Consider the pseudo code of our running example in Figure 5.1(a). The goal is to optimize the addressing modes of register `ar0`. The underlying target architecture supports the indirect addressing mode `*(ar)`, the post modification addressing mode `*(ar0++)`, `*(ar0--)`, `*(ar0+=c)`,[1] and the indirect-with-offset addressing mode `*(ar+c)`.

For sake of simplicity we assume that post modification can be executed without additional overhead. The indirect-with-offset addressing mode (`ar + c`) has worse pipeline characteristics than post modification and a longer instruction encoding as well. An explicit add instruction for address register `ar0` needs considerably more time than employing addressing modes and should be avoided in general.

In the example of Figure 5.1(a), the loop is executed 10 times and the condition `c` is true in 7 iterations and false in 3 iterations. The optimal output program for minimal execution time is shown in Figure 5.1(b). The add instruction can be moved out of the loop and post modification addressing modes can be used instead of indirect-with-

[1] In contrast to many programming languages, the `+=` operator has a post modification semantic in the context of addressing modes.

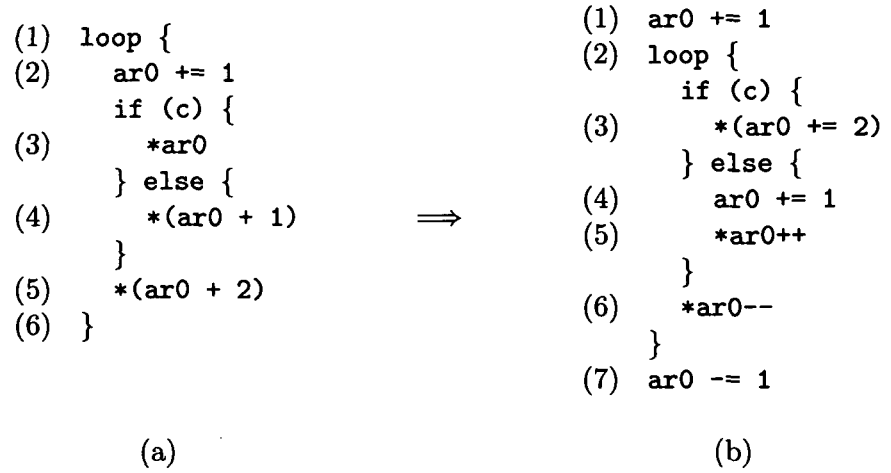


Figure 5.1.: Running example: the original code (a) and the optimized code (b)

offset addressing modes in line (5) and (6). An explicit add instruction for the address register must be inserted prior to the loop and in line (4), but this is less expensive (in terms of execution time) than the original program.

In our experience, which are presented in Section 5.7, we have seen that the AMS is a very important problem for embedded system processors. We have integrated AMS into the Atair C-Compiler for the NEC uPD77050 DSP family and used typical digital signal processing applications as benchmarks. The experiments show that code size reductions up to 50% and speedups of more than 60% are achievable.

5.3. Modeling of the AMS Problem

The AMS problem is formulated for a single address register. As a CPU architecture typically provides a set of address registers, the AMS algorithm is performed for each address register separately. In the following we refer to the currently optimized address register as the *address register*. The input for the AMS algorithm is the control flow graph (CFG) of a program. Each CFG node represents a single instruction. This means that we decompose a basic block, which is a linear sequence of instructions, into a linear list-type subgraph in the CFG containing a node for each instruction. For each node in the CFG, the AMS algorithm decides which addressing mode is the best, based on a cost model. This decision cannot be done locally as demonstrated in our running example of Figure 5.1.

For a better understanding we give some examples of addressing modes which can be found on various architectures, especially DSP architectures. An addressing mode

is the method of generating an address in the address generation unit of the target architecture. The addressing mode defines how the address is calculated from the address register value and how the address register is modified when generating the address.[2]

5.3.1. Addressing Modes

We distinguish between real addressing modes and pseudo addressing modes. Real addressing modes generate an address and access the memory with the generated address. This address is called the *access value*. In general, the access value is not known at compile time and may even change for different executions of an instruction, e.g. a load instruction in a loop.

In the following we list examples of real addressing modes.

- The basic addressing mode is indirect addressing. The memory is accessed at the address value of the address register and the address register is not modified. So the access value is the value of the address register. An example is a load from memory into general purpose register: `r1 = *ar0`.

- Post increment and post decrement addressing modes are basically the same as the indirect addressing mode, except that the address register is incremented or decremented after the memory access, respectively. Usually the increment or decrement value is equal to the access size in memory. These addressing modes are useful for sweeping over an array in a loop. Examples are `r1 = *ar0++` or `r1 = *ar0--`. The access value is the value of the address register before modification.

- A more general form is the post modification addressing mode. In contrast to the post increment/decrement addressing modes, the modification value can be specified explicitly within a given range, e.g. `r1 = *(ar0+=2)`. The available range for the modification value depends on the architecture. Usually it is only a subset of the whole address space. The drawback of this addressing mode is that it needs more coding space compared to the post increment/decrement modes. The access value is the value of the address register before modification.

- Some architectures provide an indirect-with-offset addressing mode. The access value is obtained by adding a constant offset to the address register value, but the value of the address register is not changed, e.g. `r1 = *(ar0+2)`. The indirect-with-offset addressing mode implies an addition before accessing the memory that may result in pipeline hazards.

[2]We do not consider addressing modes which do not involve address registers, like direct addressing.

- In many architectures both, the post modification and indirect-with-offset addressing modes, are also available with an index register instead of an immediate value, as shown in the following example: `r1 = *(ar0+=ir0)` and `r1 = *(ar0+r0)`. The index register variants need some special treatment in the cost model by defining pseudo values for the index registers in the value domain.

- Usually DSPs provide modulo and bit-reverse addressing modes for accessing circular buffers and implementing FFT algorithms. These addressing modes must be explicitly specified by the programmer, because the source language (which is C) provides no means of describing bit-reversed and modulo addressing. Therefore the modulo and bit-reversed modes are not automatically generated by the AMS optimization.

In contrast to real addressing modes, the pseudo addressing modes do not access the memory. These are all instructions which use the address register, but not for memory access. For convenience we define an access value for most of the pseudo addressing modes, too.

- Instructions which initialize the address register are considered as address register definitions. An example is a move instruction from a general purpose register (`ar0 = r1`) which initializes the address register before a loop. The access value is defined as the value to which the address register is set. Definitions are different from all other addressing modes, because the value of the address register is arbitrary before the definition.

- The counterpart of the the address register definition is the address register read. Such an instruction reads the address register value — the access value — but does not access the memory, for example a move to a general purpose register (`r1 = ar0`). For the problem model, this pseudo addressing mode is equivalent to the indirect addressing mode.

- The definition of an index register is the only addressing mode in which the address register is not involved directly. But it must be considered by the AMS because it influences the real addressing modes which use the index register.

- For the AMS problem, explicit address register add instructions have a great potential for optimizing the code. First, add instructions might be eliminated by using addressing modes. For example, the code sequence `r0=*ar;ar+=1` can be replaced by `r0=*ar++`, which saves one instruction. Second, an add instruction might be inserted by the AMS algorithm to optimize the program at another place that is executed more frequently. E.g., an add instruction is inserted at node (4) in Figure 5.1 for obtaining a better code inside the more frequently

executed then-branch of the if-statement. For add instructions no access value is defined.

- Finally, a node in the CFG, which does not contain address registers, can be treated as an add instruction with zero increment. If the AMS algorithm selects a constant different from zero, an explicit add instruction must be inserted.

Now we describe a formal method to express addressing modes. We define an *offset value*, which is the offset of the address register value compared to the access value of an addressing mode. The offset value is either an integer constant or a symbolic value. The symbolic value ρ is used for addressing modes which involve an index register. In this case ρ denotes the index register value, which is not known at compile time.

For an addressing mode we have two offset values: the *entry offset* is the offset before the instruction with the addressing mode is executed, the *exit offset* is the offset after the instruction is executed.

Definition 18. *Addressing mode am is defined as set of entry- and exit-value pairs.*

$$am = \bigcup_{1 \leq i \leq n} \{\langle e_i, x_i \rangle\} \tag{5.1}$$

where n is the number of offset pairs, e_i are the entry offsets and x_i are the exit offsets of addressing mode am.

For the most common addressing modes, like the indirect addressing `*(ar)`, the set *am* contains a single value pair, i.e. $\{\langle 0, 0 \rangle\}$. Addressing modes which can encode a whole set of constant values, like the general post modification `*(ar+=c)`, contain a value pair for each possible constant value.

For addressing modes, which do not define an access value, like add instructions, the entry- and exit values are not relative to the access value. The difference between entry- and exit values define the value which is added to the address register.

The following table lists the sets for the previously introduced addressing modes. The set D is the domain of offset values. The set C denotes the available constants in an addressing mode. Usually C is a range of integer values $[-x, x-1]$ where x is a power of two. This number range is normally smaller than the domain D in order to keep the instruction word small.

Addressing Mode	Syntax	*am*
indirect addressing	`*ar`	$\{\langle 0,0\rangle\}$
post increment	`*ar++`	$\{\langle 0,1\rangle\}$
post decrement	`*ar--`	$\{\langle 0,-1\rangle\}$
post modification	`*(ar+=c)`	$\bigcup_{c\in C}\{\langle 0,c\rangle\}$
indirect-with-offset	`*(ar+c)`	$\bigcup_{c\in C}\{\langle -c,-c\rangle\}$
post register modification	`*(ar+=ir)`	$\{\langle 0,\rho\rangle\}$
register indexing	`*(ar+ir)`	$\{\langle -\rho,-\rho\rangle\}$
register definition	`ar=x`	$\bigcup_{i\in D}\{\langle i,0\rangle\}$
register read	`x=ar`	$\{\langle 0,0\rangle\}$
index register definition	`ir=x`	$\bigcup_{i\in D\setminus\{\rho\}}\{\langle i,i\rangle\}$
add	`ar+=c`	$\bigcup_{i\in D\setminus\{\rho\},c\in C}\{\langle i,i+c\rangle\}$
register add	`ar+=ir`	$\{\langle 0,\rho\rangle\}$
empty	–	$\bigcup_{i\in D}\{\langle i,i\rangle\}$

5.3.2. Basic Idea

The goal of AMS is to replace addressing modes with other addressing modes, which are cheaper according to a defined cost model. The principal idea is that we shift the value of the address register `ar` between two consecutive instructions. For this purpose we insert an add instruction before and after each instruction. Each instruction `i` is replaced by `ar=ar-`e_i`; i; ar=ar+`x_i, where e_i is the entry value and x_i is the exit value of instruction i. To maintain correct program semantics, the exit and entry values of two consecutive instructions i and j must match, i.e. $x_i = e_j$. Finally a peephole optimization can eliminate sequences of add instructions and addressing modes to a cheaper addressing mode.

In Figure 5.2 the shift of `ar` is illustrated. Figure 5.2(a) shows the original instructions. The offset value is zero per definition. Figure 5.2(b) shows the program after inserting the add instructions. On the right side the offset value is shown.

Memory accesses and address modifications of the input program can be rewritten by several addressing modes with different costs. Basically, we are interested in choos-

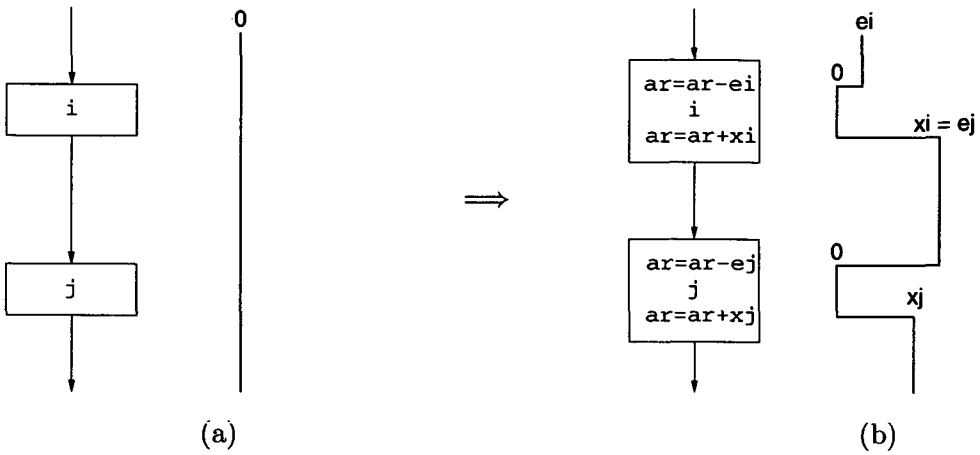


Figure 5.2.: Shift of address register

ing addressing modes for the input programs such that the overall costs are minimal. Note that the selection of an addressing mode is not a local decision because it induces offset constraints for the entry and exit offsets. Preceding and succeeding instructions must have matching offset values – otherwise program semantics is destroyed.

5.4. Mapping to a PBQP

In this section we describe the mapping of the AMS problem to a PBQP. Two steps are involved: (1) construct the PBQP-graph based on the CFG and (2) determine cost matrices of edges. The third step of building the cost vectors of the nodes is not required, because for the AMS problem all cost vectors are zero vectors.

The main idea of the mapping is that the decision vectors in the PBQP correspond to the set of offset values. Each element corresponds to a specific offset value. As the domain of offset values D is very large, the implementation of the cost model needs to be represented in a compact form (see Section 5.6).

5.4.1. The PBQP-Graph

As already shown in Figure 5.2, the exit and entry offset values of two subsequent instructions must match. For a CFG node, which may have several successors, we can generalize the previous observation.

$$x_n = e_m \quad \forall m \in SUCC(n) \tag{5.2}$$

Offset values are propagated along CFG edges and enforce a consistent offset value between two subsequent instructions. This constraint imposes a partitioning of edges. In a partition, all the entry- and exit-offsets of their associated edges must have the same offset value. An edge class is given by following definition.

Definition 19. *The set* $L = \{l_1, \ldots, l_n\}$ *is the set of CFG edge classes. Two edges* $e_i, e_j \in E$ *are in the same edge class* l_k *iff* target(e_i) = target(e_j) *or* source(e_i) = source(e_j).

The condition above can only be relaxed if a use of an address register can not be reached on any path starting from a join point in the CFG. Then a consistent offset value is not needed at the join point. An easy way to handle this exception is to perform a liveness analysis prior to the AMS algorithm and exclude all CFG nodes and edges, where the address register is not alive.

For solving the AMS problem we map the original CFG to a new graph, the PBQP-graph. A node in the PBQP-graph combines all CFG edges which belong to one edge class and represents this set of edges. An edge in the PBQP-graph represents a CFG node. For the start and end node in the CFG node we would not have source and target nodes in the PBQP-graph. Therefore we introduce the artificial nodes $\bot$ and $\top$.

An edge class is the transitive closure of all edges which have common source or target nodes (an edge class is a zig-zag pattern in the CFG). The PBQP-graph is constructed from the CFG by exchanging the meaning of *edges* and *nodes*. With Definition 19 the PBQP-graph construction algorithm can be formulated as follows:

1. Group all edges in the CFG into edge classes.
2. Generate a PBQP edge for each CFG node n from PBQP node l_i to l_j, where $\forall p \in PRED(n) : p \in l_i$ and $\forall s \in SUCC(n) : s \in l_j$.
3. Add entry the node $\top$ and the exit node $\bot$.
4. Generate a PBQP edge for the CFG entry node e from PBQP node $\top$ to l_u, where $\forall s \in SUCC(e) : s \in l_u$.
5. Generate a PBQP edge for the CFG exit node x from PBQP node l_v to $\bot$, where $\forall p \in PRED(x) : p \in l_v$.

Figure 5.3 shows the CFG and the related PBQP-graph of our example. It consists of three edge classes (a, b, c) and the entry and exit classes ($\top$, $\bot$). As there are no register definitions in the program, all CFG nodes and edges are included in the PBQP-graph construction process.

The crucial point is now that for the AMS problem almost all graphs can be reduced without applying the RN reduction. So the solution is optimal for almost

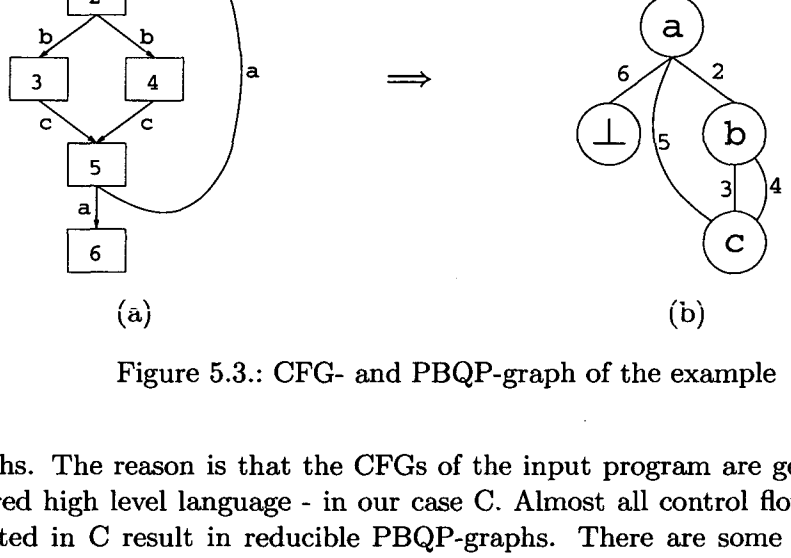


Figure 5.3.: CFG- and PBQP-graph of the example

all graphs. The reason is that the CFGs of the input program are generated from a structured high level language - in our case C. Almost all control flow which can be formulated in C result in reducible PBQP-graphs. There are some exceptions, like the goto statement and program transformations, which may produce non reducible CFGs. But even if the CFG is not reducible, the PBQP is reducible in many cases.

5.4.2. Defining the Costs

An edge in the PBQP-graph represents a node in the CFG, which contains a single instruction for which an addressing mode has to be selected. We can model the costs of addressing modes by defining the cost matrices for the edges in the PBQP-graph.

As stated above, no cost vectors have to be defined, because all vectors are zero-vectors. There is one exception if the address register is forced to an offset value of 0 at the function entry and exit. In this case the cost vector of the $\top$ and $\bot$ nodes have a zero element at offset zero and all other elements are infinite.

The AMS model can be configured by a cost function $ac(am, i)$ which yields a cost value for a specific addressing mode am when used in instruction i. These are the costs of replacing the sequence `ar=ar-`e_i`;` i `;` `ar=ar+`x_i with the instruction i', where i' uses the selected addressing mode am. If no addressing mode is available for the given entry- and exit-values e_i and x_i, the adjacent add instructions must remain and contribute to the cost.

Different optimization goals can be selected with the cost function ac, e.g. optimize for minimal execution time or optimize for minimal code size. When optimizing for minimal execution time, the cost function yields the number of cycles for executing the addressing mode, weighted by the dynamic execution weight ew of the instruction,

which can be obtained by profiling. When optimizing for minimal code size, the function yields the number of coding bits for the addressing mode. In this case the function ac does not depend on i, because a code size penalty is independent of the point of code insertion.

For each edge $e_i = \langle p, s \rangle$, which relates to instruction i, the cost matrix C_{ps} has to be defined. First we have to analyze the instruction i and find out the offset pair $\langle u, v \rangle$ of the instruction's original addressing mode. For example if the original instruction contains an address register modification, like `*(ar+=3)`, the offset pair $\langle u, v \rangle$ is $\langle 0, 3 \rangle$. If the original instruction is an add instruction `ar=ar+c`, then the offset pair $\langle u, v \rangle$ is $\langle 0, c \rangle$. The cost matrix C_{ps} can be calculated according to Equation 5.3.

$$C_{ps}(e, x) = \min_{\langle e-u, x-v \rangle \in am} ac(am, i) \tag{5.3}$$

An offset pair is associated with an element of a cost matrix, i.e. element $C(e, x)$ of matrix C gives the costs for the cheapest addressing mode for the offset pair $\langle e, x \rangle$. Note that the values of e and x might also be negative. To get positive row and column indices a mapping function is required, i.e. the minimum negative value in D is subtracted.

We can now formulate the cost matrices for our example in Section 5.2. Figure 5.3(a) shows the CFG of the input program. We assign costs of 0 for the post modification mode and costs of 0.2 for the indirect-with-offset mode, because it has worse characteristics than post modification (e.g. larger coding). Inserting an add instruction contributes a cost of one. From the assumed loop iteration count of 10 and the condition evaluation of 7 times true, we get dynamic execution weights for the CFG nodes of $ew(1) = 1$, $ew(2) = 10$, $ew(3) = 7$, $ew(4) = 3$, $ew(5) = 10$, and $ew(6) = 1$. Because we want to optimize for minimal execution time, we multiply the addressing mode costs by the dynamic execution counts of the nodes. For this example we limit the domain of offset values to the set of $\{0, 1, 2\}$ to keep the matrices small. Of course the real implementation of the algorithm has to take the whole domain of available values into account. Therefore sparse matrix representations are required.

Let us construct the cost matrix of the empty node (1). Since the node does not contain an instruction, which accesses memory or modifies the address register, we treat it is an add instruction with zero increment. For each offset pair in the the domain we find the cheapest addressing mode. The cost matrix and the corresponding addressing modes for node (1) are listed below.

$$C_1 = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

am	0	1	2
0	`ar+=0`	`ar+=1`	`ar+=2`
1	`ar-=1`	`ar+=0`	`ar+=1`
2	`ar-=2`	`ar-=1`	`ar+=0`

The matrix C_1 represents the costs for all offset pairs. The table on the right-hand side of the matrix gives the associated addressing modes. Note that the rows relate to the entry offset values and the columns to the exit offset values. Any transition from an entry offset value to an exit different offset value needs an add instruction to be inserted with a cost of 1. If the entry and exit offset values are identical the add instruction is not necessary since it has a zero increment.

The matrix for node (2), which represents the add instruction `ar = ar + 1`, and the associated addressing modes are given as follows:

$$C_2 = 10 \cdot \begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix}$$

am	0	1	2
0	`ar+=1`	`ar+=2`	`ar+=3`
1	`ar+=0`	`ar+=1`	`ar+=2`
2	`ar-=1`	`ar+=0`	`ar+=1`

Node (2) is executed 10 times and therefore the cost matrix is multiplied by a factor of 10. Moreover, the cost matrix contains two elements whose values are -1. The associated addressing modes of those elements eliminate the add instruction. For all other offset pairs the add instruction remains in the program.

The instruction `*(ar)` of node (3) imposes a more complex cost matrix and addressing mode table. For one offset pair there can be more than one choice. For such a case we have to take the addressing mode with the cheapest costs. In addition some offset pairs require an additional add instruction to update the value of the address register. The cost matrix of node (3) is given by

$$C_3 = 7 \cdot \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0.2 & 1 \\ 1 & 1 & 0.2 \end{pmatrix}$$

where the addressing mode table is listed below

am	0	1	2
0	`*(ar)`	`*(ar++)`	`*(ar+=2)`
1	`ar-=1;*(ar)`	`*(ar-1)`	`ar-=1;*(ar+=2)`
2	`ar-=2;*(ar)`	`ar-=2;*(ar++)`	`*(ar-2)`

The zeros in the first row result from the post modification addressing mode and the elements in the remaining diagonal whose values are 0.2, result from the indirect-with-offset addressing mode. For all other offset pairs an add instruction must be inserted.

The cost matrices and addressing mode tables of nodes (4) and (5) are constructed akin to the previous instructions. The difference to matrix C_3 is that the element

values are shifted by -0.2, because the original nodes (4) and (5) already contain an indirect-with-offset addressing mode.

$$C_4 = 3 \cdot \begin{pmatrix} 0 & 0.8 & 0.8 \\ -0.2 & -0.2 & -0.2 \\ 0.8 & 0.8 & 0 \end{pmatrix}, \quad C_5 = 10 \cdot \begin{pmatrix} 0 & 0.8 & 0.8 \\ 0.8 & 0 & 0.8 \\ -0.2 & -0.2 & -0.2 \end{pmatrix}$$

where the addressing mode of node (4) `*(ar0 + 1)` is

am	0	1	2
0	`*(ar+1)`	`ar+=1;*(ar);`	`ar+=1;*(ar++)`
1	`*(ar--)`	`*(ar)`	`*(ar++)`
2	`ar-=1;*(ar--);`	`ar-=1;*(ar)`	`*(ar-1)`

and of node (5) `*(ar0 + 2)` it is

am	0	1	2
0	`*(ar+2)`	`ar+=2;*(ar--);`	`ar+=2;*(ar)`
1	`ar-=1;*(ar-=2)`	`*(ar+1)`	`ar-=1;*(ar)`
2	`*(ar-=2);`	`*(ar--)`	`*(ar)`

For node (6) we obtain the same cost matrix as already presented for node (1), i.e. $C_1 = C_6$.

5.5. Solving the PBQP

After generating a PBQP from the AMS problem, the PBQP must be solved. In the sequel we show the reduction and back-propagation phase of the solver for our example. The reduction steps are depicted in Figure 5.4. First, we have to combine edges 3 and 4, because only a single edge is allowed between two nodes. The resulting matrix is the sum of matrix C_3 and C_4 which yields

$$C_{34} = \begin{pmatrix} 0 & 2.4 & 2.4 \\ 6.4 & 0.8 & 6.4 \\ 9.4 & 9.4 & 1.4 \end{pmatrix}$$

The first step is the reduction of the degree-one nodes $\top$ and $\bot$ which adds two vectors (0 1 1) to the node vector a. Note that the cost vectors of $\top$ and $\bot$ are infinite, except the first element, because at function entry and exit the address register must not be changed.

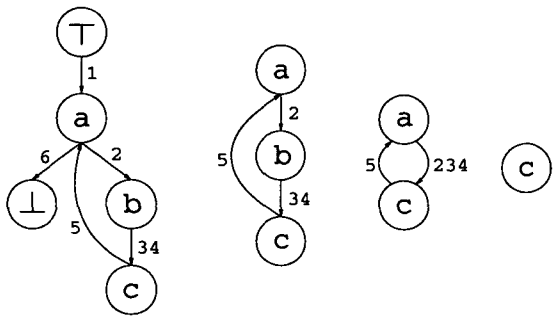


Figure 5.4.: Reduction sequence

The resulting node vector is $v_a = (\ 0 \quad 2 \quad 2\ )$. The remaining cycle of three nodes is reduced by reducing (any) node with the RII reduction. In the example we select node b. The new edge gets a matrix of

$$C_{234} = \begin{pmatrix} 0 & 2.4 & 2.4 \\ 6.4 & 0.8 & 6.4 \\ 9.4 & 9.4 & 1.4 \end{pmatrix}$$

In the next step we have to combine edges 234 and 5 by adding C_{234} and C_5^T.

$$C_{2345} = \begin{pmatrix} 0 & 8.8 & -0.6 \\ -2 & -7.6 & -9.6 \\ 4.4 & -1.2 & -5.6 \end{pmatrix}$$

The last reduction step is a RI of node a. The only remaining node is c with vector $(\ 0 \quad -5.6 \quad -7.6\ )$ and the offset value can be selected by taking the index of the minimal element -7.6, i.e. $s_c = 2$. The minimal element -7.6 represents the total cost of the optimization. It should be negative, because the optimization should bring a benefit, rather than extra costs. Now the reduction process is reversed and offset values are selected for all nodes in the order $s_c = 2$, $s_a = 1$, $s_b = 0$, $s_\perp = 0$, $s_\top = 0$.

The solution of the PBQP problem yields an offset value in each node of the PBQP-graph. The offset values are then transferred to the CFG. The entry value e_i of instruction i is the PBQP solution of the predecessor edge class of i, the exit value x_i is the PBQP solution of the successor edge class of i. The selection of the addressing mode for an instruction is done by selecting the am with the minimum cost which contains the pair $\langle i, j \rangle$. Add instructions with zero additive constants can be deleted.

In our example, the entry and exit values can be obtained from the predecessor and successor edge values respectively: $e_1 = 0$, $x_1 = 1$, $e_2 = 1$, $x_2 = 0$, $e_3 = 0$, $x_3 = 2$, $e_4 = 0$, $x_4 = 2$, $e_5 = 2$, $x_5 = 1$, $e_6 = 1$, $x_6 = 0$. From these entry and exit values, the output program, which is already shown in Section 5.2, can be generated.

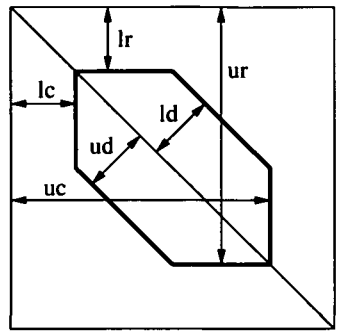


Figure 5.5.: Definition of a matrix region

5.6. Sparse Matrix Implementation

The size of the vectors and matrices used in the PBQP algorithm is the number of available values which an address register may contain, i.e. $|D|$. In practice this number is very large (e.g. 2^{16} or 2^{32}). Although the number is constant it is not possible to implement such large vectors and matrices as arrays of values. To get a handle on the problem, a sparse vector and matrix representation is required.

Usually a sparse matrix implementation stores single elements, which are not zero [29]. In our implementation we store *regions* of equal elements, which are not *infinite.*

5.6.1. Sparse Matrix Representation

A matrix is expressed by a set of cost regions. Each region defines a six-sided area in the matrix with a specific cost value.

Definition 20. *Let* $r = \langle lr, ur, lc, uc, ld, ud, rc \rangle$ *be a cost region. Then*

$$c(r,i,j) = \begin{cases} rc, & \text{if } lr \leq i \leq ur \ \wedge \ lc \leq j \leq uc \ \wedge \ ld \leq i-j \leq ud \\ \infty, & \text{otherwise} \end{cases}$$

The values lr and ur specify the row-interval. It delimits the region area in the row-dimension. The values lc and uc specify the column-interval which delimits the region in the column-dimension. This rectangle shaped area is further delimited by a diagonal-interval, specified by ld and ud. The cost matrices for many addressing modes, like add instructions, contain such diagonal oriented cost areas. Therefore it is very useful to define this kind of six-sided region instead of rectangular regions. It is also possible that a region is not six-sided, but five-sided or rectangular.This is the case if $ld \leq lc - ur$ or $ud \geq lr - uc$. A graphical representation of a region is shown in Figure 5.5

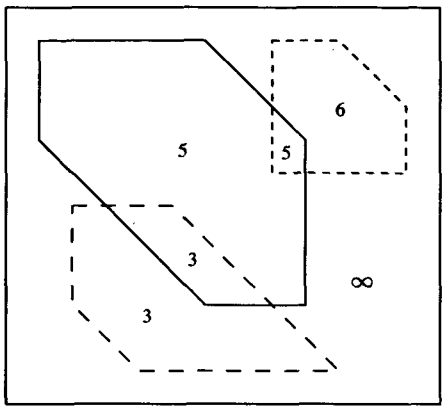


Figure 5.6.: The sparse representation of a matrix by three regions. In the region intersections the regions with the minimum cost values define the resulting matrix element values.

The function c yields the region value for matrix element i, j. Inside the region area, the function c yields the cost value rc. Outside the area the value of c is infinite. Note that it is not required that the lower bound of an interval is less than the upper bound. If this is not the case for any of the intervals (row-, column-, or diagonal-interval), — according to the definition — function c is infinite for all values of i and j.

With the help of cost regions we can now specify the sparse representation of a matrix.

Definition 21. *Let $R_M = \{r_1, \ldots, r_n\}$ be a set of regions. Then the matrix M is defined by*

$$M(i,j) = \min_{r \in R_M} c(r,i,j)$$

The definition imposes no constraints on the region's bounds, which allows region areas to overlap each other. The resulting value of an overlapped area is the minimum of the region costs. The possibility for overlapped regions minimizes the number of regions in cases where a small region with less cost is embedded in a larger region with higher cost. Figure 5.6 shows an example of a matrix consisting of three regions.

5.6.2. Sparse Vector Representation

The sparse vector representation is equivalent to a $1 \times n$ matrix representation. But for a vector, only the column-interval is required to define a one-dimensional region.

Therefore we introduce a simplified region definition for the sparse vector representation.

Definition 22. *Let $r = \langle l, u, rc \rangle$ a cost region. Then*

$$c(r,i) = \begin{cases} rc, & \text{if } l \leq i \leq u \\ \infty, & \text{otherwise} \end{cases}$$

The region defines only a single interval with the values l and u. Again, the cost function c yields the region value for vector element i. The sparse vector representation is defined similar to the sparse matrix representation.

Definition 23. *Let $R_v = \{r_1, \ldots, r_n\}$ be a set of regions. Then the vector $\vec{v}$ is defined by*

$$\vec{v}(i) = \min_{r \in R_v} c(r,i)$$

5.6.3. Operations

We have to describe all operations, which are used by the PBQP solver, for the sparse matrix representation. The operations include vector and matrix addition, the calculation of the $\vec{\delta}$ vector and the Δ matrix in the reduction rules, matrix transpose, and the $i_{min}(\vec{v})$ operator on a vector.

Matrix and Vector Addition The matrix and vector additions are equivalent, because a vector has the same representation as a $1 \times N$ matrix. Let $C = A + B$ be matrices of size $N \times M$. Matrix C is defined by

$$C(i,j) = A(i,j) + B(i,j) \tag{5.4}$$

Inserting the sparse matrix representation we get

$$C(i,j) = \min_{r_A \in R_A} c(r_A,i,j) + \min_{r_B \in R_B} c(r_B,i,j) \tag{5.5}$$

Combining the minimum operations yields

$$C(i,j) = \min_{\langle r_A, r_B \rangle \in R_A \times R_B} [c(r_A,i,j) + c(r_B,i,j)] = \min_{r_C} c(r_C,i,j) \tag{5.6}$$

where $\langle r_A, r_B \rangle$ is the set of all possible combinations of regions in R_A and R_B. For constructing the region set R_C for matrix C we build a region r_C for each combination $\langle r_A, r_B \rangle$. For region r_C the following equation must hold

$$c(r_C,i,j) = c(r_A,i,j) + c(r_B,i,j) \quad \forall i,j : 1 \leq i \leq N, 1 \leq j \leq M \tag{5.7}$$

The cost function $c(r_C, i, j)$ evaluates to ∞ if one of the operand cost functions yields ∞. Otherwise it evaluates to $rc_A + rc_B$. We can now state the conditions for $c(r_C, i, j)$ not yielding ∞.

$$lr_A \leq i \leq ur_A \tag{5.8}$$
$$lr_B \leq i \leq ur_B \tag{5.9}$$
$$lc_A \leq j \leq uc_A \tag{5.10}$$
$$lc_B \leq j \leq uc_B \tag{5.11}$$
$$ld_A \leq i - j \leq ud_A \tag{5.12}$$
$$ld_B \leq i - j \leq ud_B \tag{5.13}$$

From these conditions the region r_C can be derived. Inequalities 5.8 and 5.9 describe the new row-interval. Similarly 5.10 and 5.11 describe the new column-interval and 5.12 and 5.13 describe the new diagonal-interval. The area of region r_C is the intersection of the region areas of r_A and r_B.

$$\begin{aligned} r_C = \langle & max(lr_A, lr_B), min(ur_A, ur_B), \\ & max(lc_A, lc_B), min(uc_A, uc_B), \\ & max(ld_A, ld_B), min(ud_A, ud_B), \\ & rc_A + rc_B \rangle \end{aligned}$$

The worst-case complexity of the addition is $O(n \cdot m)$, where n and m are the number of regions in matrices A and B, respectively. Before the addition operation is performed, the regions are sorted in row order. In practice, many row-intervals are small and the regions are distributed over the row-dimension. Therefore only a fraction of all possible region combinations have to be calculated and the actual complexity is less than quadratic.

Calculation of δ The $\vec{\delta}$ vector is used in the RI reduction rule. It is defined as

$$\vec{\delta_x}(i) = \min_k [A(i,k) + \vec{c}(k)]$$

where A and $\vec{c}$ are the matrix and the vector involved in the RI reduction. Inserting the sparse matrix representation we get

$$\delta(\vec{i}) = \min_k \left[\min_{r_A \in R_A} c(r_A, i, k) + \min_{r_c \in R_c} c(r_c, k) \right] \tag{5.14}$$

Combining the minimum operations yields

$$\vec{\delta(i)} = \min_{k} \min_{\langle r_A, r_c \rangle \in R_A \times R_c} [c(r_A, i, k) + c(r_c, k)] \tag{5.15}$$

As for the matrix addition we build a region r_δ for each region combination of $\langle r_A, r_c \rangle$. The following equation must hold

$$c(r_\delta, i) = \min_{k} [c(r_A, i, k) + c(r_c, k)] \quad \forall i : 1 \leq i \leq N \tag{5.16}$$

Again, the cost function $c(r_\delta, i)$ can yield only two values: $rc_A + rc_c$ and ∞. The conditions for $c(r_\delta, i)$ for not yielding ∞ are as follows.

$$lr_A \leq i \leq ur_A \tag{5.17}$$
$$lc_A \leq k \leq uc_A \tag{5.18}$$
$$ld_A \leq i - k \leq ud_A \tag{5.19}$$
$$l_c \leq k \leq u_c \tag{5.20}$$

For constructing the $\vec{\delta}$-regions we have to eliminate k. We derive a new inequality by adding 5.19 and 5.20, which yields 5.21.

$$ld_A + l_c \leq i \leq ud_A + u_c \tag{5.21}$$

From these conditions the region r_δ can be constructed.

$$r_\delta = \langle max(lr_A, ld_A + l_c), min(ur_A, ud_A + u_c), rc_A + rc_c \rangle$$

Inequalities 5.18 and 5.20 indicate that the resulting r_δ region is only generated if the column- and the vector-intervals overlap.

$$max(lc_A, l_c) \leq min(uc_A, u_c)$$

The worst case complexity of the δ-computation is $O(n \cdot m)$ where n is the number of regions in matrix A and m is the number of regions in the vector $\vec{c}$. As for the matrix addition, the actual complexity is less than the worst case complexity.

Calculation of Δ The calculation of the Δ matrix in the RII reduction is defined as

$$\Delta(i, j) = \min_{k} [A(i, k) + B(j, k) + \vec{c}(k)]$$

where A, B and $\vec{c}$ are the matrices and the vector involved in the RII reduction. Inserting the sparse matrix representation we get

$$\Delta(i,j) = \min_k \left[\min_{r_A \in R_A} c(r_A,i,k) + \min_{r_B \in R_B} c(r_B,j,k) + \min_{r_c \in R_c} c(r_c,k) \right] \tag{5.22}$$

Combining the minimum operations yields

$$\Delta(i,j) = \min_k \min_{\langle r_A,r_B,r_c\rangle \in R_A \times R_B \times R_c} [c(r_A,i,k) + c(r_B,j,k) + c(r_c,k)] \tag{5.23}$$

As for the matrix addition we build a region r_Δ for each region combination of $\langle r_A, r_B, r_c \rangle$. The following equation must hold

$$c(r_\Delta,i,j) = \min_k [c(r_A,i,k) + c(r_B,j,k) + c(r_c,k)] \quad \forall i,j : 1 \le i \le N, 1 \le j \le M \tag{5.24}$$

Again, the cost function $c(r_\Delta,i,j)$ can yield only two values: $rc_A + rc_B + rc_c$ and ∞. The conditions for $c(r_\Delta,i,j)$ for not yielding ∞ are as follows.

$$lr_A \le i \le ur_A \tag{5.25}$$
$$lc_A \le k \le uc_A \tag{5.26}$$
$$ld_A \le i - k \le ud_A \tag{5.27}$$
$$lr_B \le j \le ur_B \tag{5.28}$$
$$lc_B \le k \le uc_B \tag{5.29}$$
$$ld_B \le j - k \le ud_B \tag{5.30}$$
$$l_c \le k \le u_c \tag{5.31}$$

For constructing the Δ-regions we have to eliminate k. First we invert Inequality 5.30 which results in Equation 5.32

$$-ud_B \le k - j \le -ld_B \tag{5.32}$$

Now we derive five new inequalities. Adding 5.27 and 5.32 yields 5.33, adding 5.27 and 5.29 yields 5.34, adding 5.27 and 5.31 yields 5.35, adding 5.30 and 5.26 yields 5.36 and adding 5.30 and 5.31 yields 5.37.

$$ld_A - ud_B \le i - j \le ud_A - ld_B \tag{5.33}$$
$$ld_A + lc_B \le i \le ud_A + uc_B \tag{5.34}$$
$$ld_A + l_c \le i \le ud_A + u_c \tag{5.35}$$
$$ld_B + lc_A \le j \le ud_B + uc_A \tag{5.36}$$
$$ld_B + l_c \le j \le ud_B + u_c \tag{5.37}$$

From these conditions the region r_Δ can be constructed.

$$\begin{aligned} r_\Delta = \langle & max(lr_A, ld_A + lc_B, ld_A + l_c), min(ur_A, ud_A + uc_B, ud_A + u_c), \\ & max(lr_B, ld_B + lc_A, ld_B + l_c), min(ur_B, ud_B + uc_A, ud_B + u_c), \\ & ld_A - ud_B, ud_A - ld_B,\ rc_A + rc_B + rc_c \rangle \end{aligned}$$

Inequalities 5.26, 5.29 and 5.31 indicate that the resulting r_Δ region is only generated if the column-intervals and the vector-interval overlap.

$$max(lc_A, lc_B, l_c) \leq min(uc_A, uc_B, u_c)$$

The worst case complexity of the Δ-computation is $O(n \cdot m \cdot o)$ where n, m and o are the number of regions in matrices A, B and vector $\vec{c}$, respectively. Again, in practice many regions have small intervals and the regions are distributed. As the regions are sorted, only those combinations are calculated, where the row-intervals of the two matrix regions and the interval of the vector region overlap. This is only a small fraction of all possible combinations and therefore the actual complexity is far less than the worst case complexity.

Matrix Transpose Obviously the regions of a transposed matrix C^T can be obtained by swapping column and row parameters of all regions of matrix C. The following equation describes how a region is transposed.

$$\langle lr, ur, lc, uc, ld, ud, rc \rangle^T = \langle lc, uc, lr, ur, -ud, -ld, rc \rangle$$

The complexity of transposing a matrix is $O(n)$.

Minimum Index The index of the smallest vector element $i_{min}(\vec{v})$ is calculated by finding the region with minimum cost. The lower bound of this region yields the minimum index. The complexity of this operation is $O(n)$, where n is the number of regions in the vector.

5.6.4. Complexity

The complexity of the PBQP solver, using the sparse matrix representation, depends on the number of regions in the matrices and vectors. The critical operations are matrix addition and the calculation of $\vec{\delta}$ and Δ. These operations may produce more regions in the result than in the operands, which result in a high computational effort.

To overcome the problem of the high computational effort, simplification is performed after each operation. For each region in a matrix or a vector, the following simplification steps are performed

- The region is removed if any of the row-, column- or diagonal-intervals is invalid, i.e. the lower bound is greater than the upper bound.
- The region is removed if it is completely contained in another region with equal or less costs.
- The region is shrunken if it is partly covered by another region with equal or less costs.
- The interval bounds are set to the minimal possible interval without changing the region area.

Our experiments show that with simplification the overall complexity is acceptable. It is almost linear with the number of decision vectors in the PBQP.

5.7. Experimental Results

For our experiments we have used the Atair C-Compiler for the NEC uPD77050 DSP family, which was introduced in Section 4.6. The load/store units of the uPD77050 facilitate various addressing modes for 8 address registers. Most of the addressing modes of the uPD77050 are discussed in Section 5.3, e.g. indirect addressing, post increment/decrement, indirect-with-offset, post modification with index register. In addition, post modification can wrap around a modulo value to implement circular buffers. Furthermore, a bit reverse addressing mode can be selected for efficiently accessing FFT buffers. All of these addressing modes can be modeled by the AMS algorithm. The bit reverse addressing mode and the modulo addressing modes require the use of functions, known by the compiler, and are not generated by the AMS optimization automatically.

Addressing mode related optimizations are performed between register allocation and scheduling on a low-level intermediate representation that is related to the uPD77050 assembly language. The addressing mode selection is performed after assigning the offsets for the function stack frames [45]. Because of the enormous complexity, it is not possible to combine all these phases into one overall optimization problem. Therefore register allocation, offset assignment, AMS and scheduling are performed in separate steps.

The AMS algorithm runs for each address register separately. Two of the address registers are used as "floating-frame-pointers". They are used to access two stack frames (one address register per stack frame). As a result of the AMS optimization, the frame pointers do not point to the beginning of the stack frames, but may point to any location within the execution of a function.

Benchmark	# of graphs	max nodes	avg nodes	compile overhead
mp3	419	2009	134.66	8.90
gsm	900	1930	143.07	4.77
aac	487	1509	85.84	7.57
trcbk	6	20	15.33	4.65
cfirc	16	61	37.62	4.24
firc	44	58	31.32	4.90
iirc	6	25	16.33	3.69
iirbiqc	13	65	35.00	6.59
lmsc	15	70	34.73	5.14
matmult	8	26	18.62	5.69
vadd	6	15	9.67	6.09
vdot	4	8	6.50	5.76
vmin	4	13	8.25	6.54
vmult	6	15	9.67	6.92
vnorm	3	8	6.00	5.80

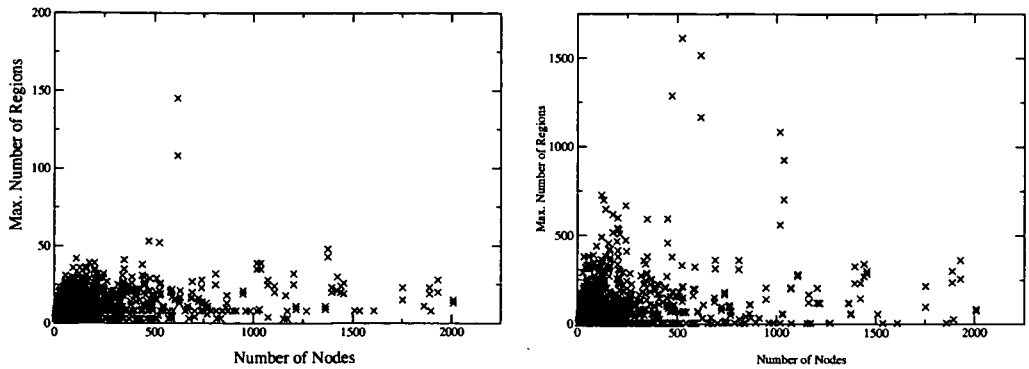
Table 5.1.: Problem size and compile time overhead of the benchmarks

For the AMS experiments we have used the same DSP benchmark suite as for the SSA-matcher experiments (see Section 4.6). The benchmark programs consists of three DSP applications and a number of small DSP specific algorithm kernels.

The benchmarks and the problem sizes of the benchmarks are listed in Table 5.1. The first column "# of graphs" shows the number of PBQP-graphs that are solved for the optimizations. The number of graphs is determined by the number of functions in a program and the number of used address registers (at most 8 for the uPD77050 architecture). The computational complexity of AMS mainly depends on the number of nodes in a PBQP-graph. The second column "max nodes" of Table 5.1 gives the number of nodes for the largest graph in the benchmark suite. In the last column "avg nodes" shows the average number of nodes for a benchmark.

Table 5.1 also shows the compile-time overhead of AMS compared to the overall compile time in percent. It ranges from 4% to 9%. This is within acceptable bounds for a production DSP compiler, taking the high quality code improvements into account. The table also shows that the overhead for the large applications is not significantly higher than for the small benchmarks. This indicates that the AMS scales almost linear with the problem size.

In the following we give some performance details of the PBQP solver. Almost all PBQP-graphs can be solved optimally. The most frequent reductions are RI (60.9%), followed by RII (25.4%), and 13.6% of all nodes have degree zero. Only 8 reductions



(b) Matrix Regions (a) Vector Regions

Figure 5.7.: Maximum number of regions

out of 230481 reductions are RN reductions which are solved by heuristics. One of the most important observation of our experiments is that the result of the address mode selection is optimal in almost all benchmarks.

Our PBQP solver employs sparse representation techniques of large vectors and matrices. Figure 5.7 illustrates the relation between number of nodes in a PBQP-graph and the maximum number of regions occurring in a vector and matrix respectively. As in the graph depicted, the maximum number of regions does not correlate with the number of nodes. In practice, the maximum number of regions is bounded and does not grow exponentially.

In the sequel we show the performance results of the AMS optimization. The baseline for the comparison is obtained by a disabling the AMS optimization in the compiler. So the baseline code contains address computations which are originally inserted by the code generator after performing strength reduction. The address register is set up only before the first access, but access and update instructions are not combined. All compiler optimizations are performed in the baseline, except AMS. We evaluated the achieved code size reduction and runtime improvements for different parameterizations of the compiler.

In the best case, our AMS algorithm achieves code size reductions up to 50% and speedups of more than 60%. Since we have a VLIW architecture, where add instructions can be scheduled without the penalty of additional execution cycles, we measured the effect of AMS by emitting VLIW code and by linear code. Moreover, we conducted experiments with two different cost models. The first cost model minimizes execution time and the second cost model minimizes code size. In Figures 5.8 and 5.9 the code reductions of the benchmark programs with linear and VLIW code are given whereas Figures 5.10 and 5.11 show the runtime improvement achieved by AMS.

We used different models for execution time and code size optimizations. The execution time model reflects the execution cycles and delay cycles of the target instructions. The costs are weighted with estimated execution counts of the basic blocks. The execution count estimation is based on the loop structure of the function. It turned out that the accuracy of the estimation is sufficient for our purpose. The cost model for code size optimization is derived from the instruction code length of the target hardware. The costs of addressing modes directly correspond to the code size which is required by addressing modes. In the code size model the costs are not weighted by the execution counts of the basic blocks.

The execution time improvements are significantly larger for small benchmarks than for bigger applications. Nevertheless, there are impressive code size reductions for bigger applications, e.g. GSM. The reason is that small benchmarks mainly contain kernels of typical DSP algorithms. The execution time improvements, which are achieved in the kernel loops, directly affect the overall improvement. For larger application more "control code" (e.g. function calls) is executed, which gives less opportunity for runtime improvements. However, as shown in Figures 5.8 and 5.9, the code size of larger applications can be significantly reduced. For some small benchmarks, e.g. trcbk, there is no improvement at all since there is no potential to optimize the addressing mode selection in this cases.

For our target architecture the compiler is able to schedule add instructions without the penalty of additional execution cycles. Even if the AMS optimization is disabled, the scheduler might find a free VLIW slot for placing an address register add instruction. In order to simulate a architecture without VLIW capabilities we conducted performance experiments on linear code (no VLIW code is generated). The code size improvements are roughly the same as with VLIW code generation but the execution time improvements are considerable larger.

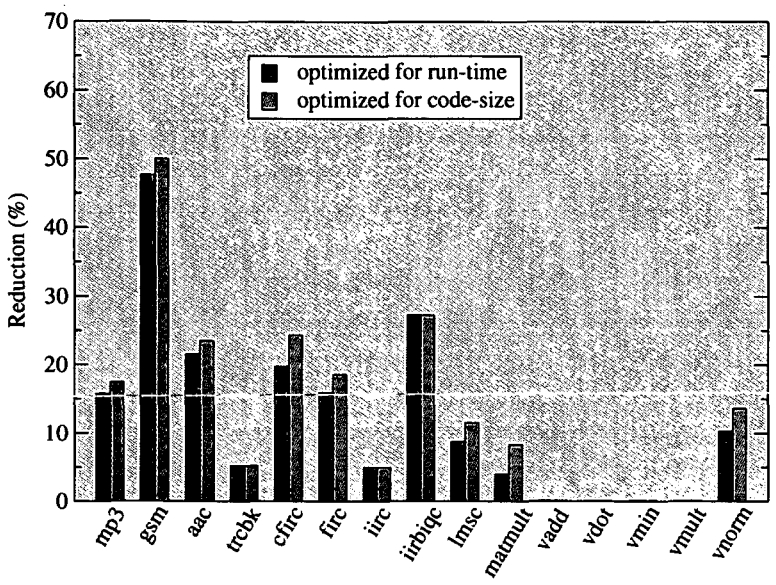


Figure 5.8.: Code size reduction with linear code

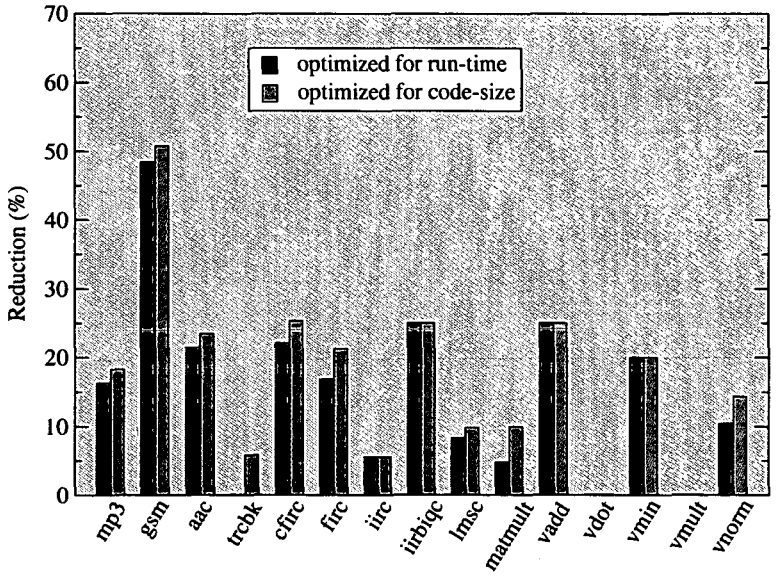


Figure 5.9.: Code size reduction with VLIW code

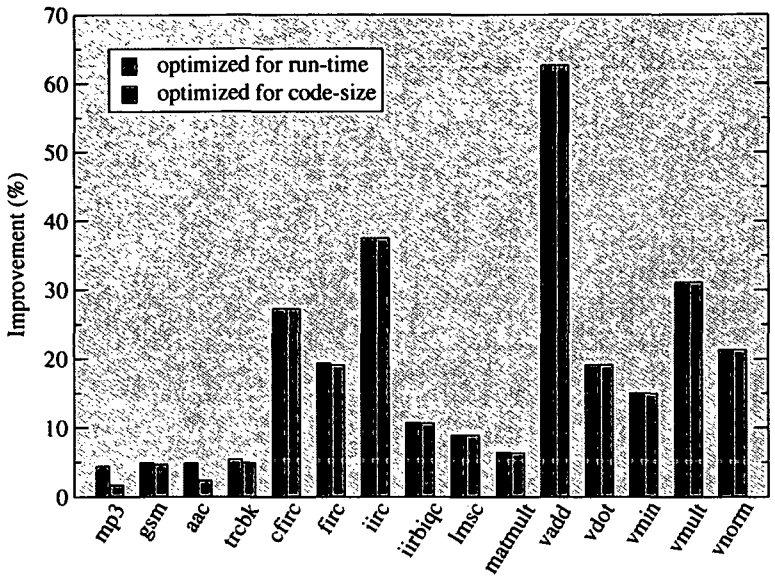


Figure 5.10.: Runtime improvement with linear code

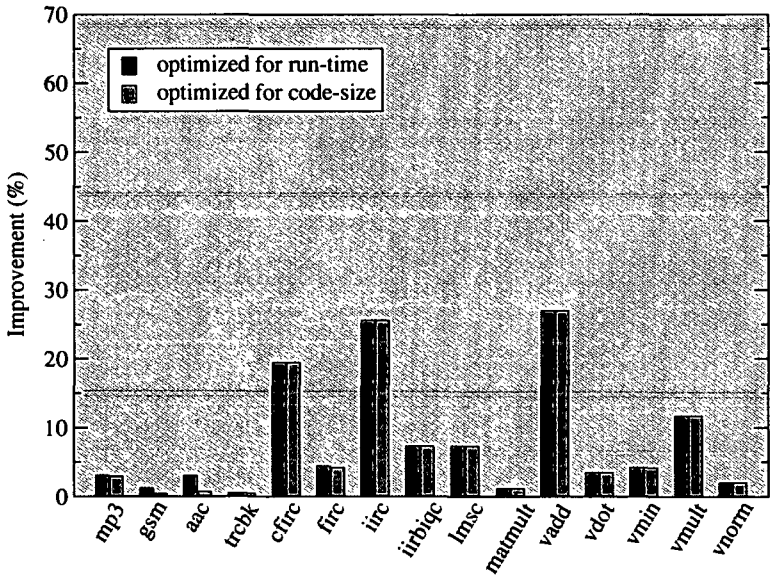


Figure 5.11.: Runtime improvement with VLIW code

6. Register Allocation

6.1. Overview

Global register allocation is of vital importance for optimizing compilers. Especially for embedded CPU architectures, register allocation has to cope with irregular register sets and register constraints.

The task of register allocation is to assign CPU registers to live ranges and to spill live ranges to memory if not enough CPU registers are available. The allocator has to pursue two goals: (1) the number of expensive spills should be as small as possible and (2) it should perform copy propagation (coalescing) which is achieved by giving source and target of a copy instruction the same register whereby the copy instruction can be removed afterwards [53].

Traditionally, register allocation abstracts the problem of assigning CPU registers to live ranges into the problem of coloring nodes in an *interference graph* [12, 10]. Unfortunately graph coloring cannot be simply adopted for irregular architectures.

In this chapter we present a new register allocation approach for irregular architectures using a mapping to a PBQP. The PBQP-graph is an extension to the interference graph. Beside the interferences, the PBQP-graph includes other types of constraints, like coalescing relations and constraints imposed by irregularities. The solution of the PBQP represents the final mapping of CPU registers to live ranges.

The main idea of the PBQP register allocator is that it tries to find a global optimal solution. In contrast, traditional approaches make a register selection based on a local decision for each live range. Unfortunately the PBQP-graphs for register allocation are very dense in general. Therefore the PBQP solver has to use the RN reduction heuristics in many cases. In our first implementation [60] we used the standard PBQP solver (with a slightly modified heuristics for the RN rule) and integrated the register allocator into the C-Compiler for the Carmel DSP, which has a highly irregular register model. The problem was that for a considerable number of test cases the graph coloring allocator based on the work of Smith et al. [61] yielded better results than our PBQP allocator.

To overcome this problem we make a simple modification to the PBQP solver: the order for selecting RN nodes during the reduction phase is determined by the selection order of a traditional graph coloring approach. With this improvement the benefits of traditional graph coloring and the PBQP method are combined. Our experiments show that the PBQP register allocator yields at least the same code quality as the traditional approach or exceeds the result.

In Section 2.4 other approaches are listed, which try to solve the register allocation problem for irregular architectures. They are still based on the interference graph or use ILP. Our method goes beyond previous work in various aspects. First it provides a unified approach for all types of register constraints which allows precise and easy modeling of irregularities by means of cost functions. This implies that coalescing is an integral part of the algorithm, because it is treated like any other constraint. Because

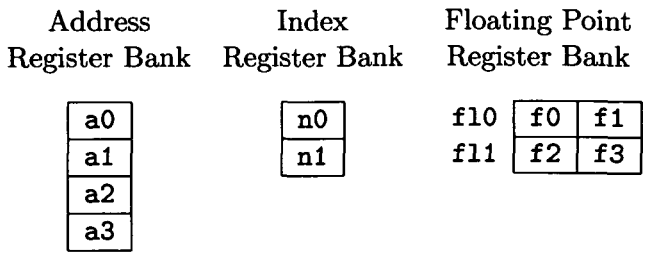


Figure 6.1.: Hypothetical CPU architecture

our algorithm combines the benefit of Chaitin-style graph coloring algorithms and the PBQP method, it yields significantly better results compared to graph coloring approaches. In contrast to ILP approaches, our solver yields a result in almost linear time.

6.2. Motivation

In this section we motivate our approach by presenting a hypothetical irregular architecture and a running example that computes an inner dot-product of two vectors. The irregular architecture has similarities with an embedded CPU architecture, consisting of various register classes as shown in Figure 6.1. Our hypothetical CPU architecture has four register classes where some of the registers are paired. Namely, it has four address registers in the address register bank, two index registers in the index register bank, and two 64-bit floating point registers. Moreover, each 64-bit floating point register can either be accessed as one 64-bit register or as two 32-bit floating point registers.

The addressing unit of our hypothetical CPU architecture supports a register indexed addressing mode that computes the memory address by adding an address register and an index register. However, index register `n0` can only be paired with address registers `a0` and `a1` – index register `n1` is similar, which can only be paired with address registers `a2` and `a3`. The following scheme shows the allowed pairs of address and index registers:

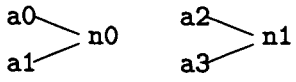


Figure 6.2(a) lists our running example. In the example we have live ranges for each register class, i.e. live ranges for address registers are denoted as `sa`i, index register live ranges as `sn`i, floating point registers as `sf`i, and long floating point registers as `sfl`i. Our example consists of a prologue (line 1–5), a loop (line 6–13), and an epilogue (line 14–15). As shown in line 1–3 we have a mixed argument passing for address

```
1:  sa0 = stack0
2:  sa1 = stack1
3:  sa2 = a2
4:  sfl0 = 0
5:  sn0 = 0
6:  loop {
7:    sn1 = sn0
8:    sf1 = *(sa0 + sn1)
9:    sn2 = sn0
10:   sf2 = *(sa1 + sn2)
11:   sfl0 += sf1 * sf2
12:   sn0 += 1
13: }
14: sn3 = sn0
15: *(sa2 + sn3) = sfl0
```

(a)

$\Longrightarrow$

```
1:  a0 = stack0
2:  a1 = stack1
3:  fl1 = 0
4:  n0 = 0
5:  loop {
6:    f0 = *(a0 + n0)
7:    f1 = *(a1 + n0)
8:    fl1 += f0 * f1
9:    n0 += 1
10: }
11: n1 = n0
12: *(a2 + n1) = fl1
```

(b)

Figure 6.2.: Running example

register live ranges sa0, sa1, and sa2. Live range sa2 is initialized by an argument passed in the CPU register a2, the others are initialized from the calling stack. In the loop, two floating point values are read from memory locations (line 8 and line 10). For the memory accesses an indexed addressing mode *(sa*i* + sn*j*) is used. Line 11 contains the accumulate instruction of the dot product and line 12 increments the index register for memory locations. In the epilogue the result is stored in memory by using the indexed addressing mode.

The move instructions in line 7, 9 and 14 are inserted by the code generator before allocating registers. This is done because without the move instructions no register allocation can be found without spilling. Therefore the code generator should always insert move instruction at each place where a coupled register is used.

The register allocator should assign CPU registers of our example architecture to the live ranges so that the spills are minimized. In addition the register allocator should try to allocate the same CPU register for the sources and targets of the move instructions. If this is possible, move instructions can be eliminated. While optimizing for these goals, the register allocator must maintain the constraints imposed by the architecture. For our example we will see that no spill is necessary but not all move instructions can be eliminated. Eliminating moves in line 7, 9, and 14 would imply a pairing of a single index register to three different address registers (indexed addressing modes in line 8, 10 and 15). In our architecture we can only pair two address register

to one index register. Therefore at least one move instruction cannot be eliminated — preferable the move instruction that is outside of the loop (line 14). Figure 6.2(b) shows the result of the register allocation.

Our hypothetical architecture exhibits another constraint for register allocation, called *register pairing*. Two short floats are paired and can also be used as one long float. These registers are no longer independent of each other and the register allocator has to take care of it. E.g., our running example has three floating point register live ranges (one long and two short floats) and the allocator should pack registers `sf1` and `sf2` into one long floating point register — otherwise float register live ranges must be spilled to memory.

6.3. Register Constraints

For register allocation, it is of paramount importance to have an accurate cost model to obtain a good decision for which live ranges are to be spilled and which live ranges are stored in CPU registers. In contrast to RISC-like architectures, irregular architectures exhibit a non-uniform cost model.

In this section we describe various register constraints, including constraints which are imposed by all architectures and constraints which can only be found on irregular architectures.

Cost functions are used to formalize the constraints. They are an exact model and are used to map the register allocation problem to a PBQP. The constraints can be distinguished by their importance.

- *Hard constraints* must be satisfied to obtain correct code, for example register interferences. The cost function yields infinity for an unsatisfied constraint.
- *Soft constraints* specify optimization opportunities. The register allocator can decide which soft constraints to satisfy. An example is the coalescing constraint. The cost function just yields a non-infinite penalty for the unsatisfied constraint.

The constraints can also be classified on the number of the dependent registers.

- Constraints on one live range. The decision to satisfy the constraint does not depend on another live range. The cost function has a single parameter.
- Constraints between two live ranges. The decision to satisfy the constraint depends on two live ranges, e.g. the interference between two live ranges. Therefore the cost function has two parameters.

For real world architectures it is sufficient to describe constraints between two live ranges at most. Even if there is a dependence between more than two live ranges, it can

be exactly modeled by multiple constraints between two live ranges. An example is the coupling of address-, index- and modulo-registers on a DSP architecture when using the modulo addressing mode. This constellation can be described by two constraints: one between address- and index-registers and the other between address- and modulo-registers.

6.3.1. Constraints on one Live Range

Constraints on one live range can be described by cost functions which take a single location $r \in A$ as parameter. The set A is an assignment set and consists of all allocatable CPU registers and a location for spilling the live range (sp).

Definition 24. *Let* $\mathbf{s}$ *be a live range and* $A = \{sp, R_1, R_2, \ldots, R_m\}$ *be an assignment set. Then* $f_{\mathbf{s}}(a)$, $a \in A$ *is the cost function describing a constraint on live range* $\mathbf{s}$.

The cost function $f_{\mathbf{s}}$ expresses the costs either for assigning live range $\mathbf{s}$ one of the CPU registers $R_1, \ldots, R_m$ or for spilling live range $\mathbf{s}$ (denoted by $sp \in A$). If assignment $a \in A$ is not valid for $\mathbf{s}$, cost function $f_{\mathbf{s}}$ maps assignment a to ∞. Usually there is more than one cost function for a live range $\mathbf{s}$ and we aggregate them in a set of cost functions denoted by $F_{\mathbf{s}}$.

In the following we want to consider some typical kinds of cost functions by looking at our running example. In our architecture we have 12 CPU registers which constitute the assignment set, together with the spill assignment sp.

$$A = \{sp, \texttt{a0}, \ldots, \texttt{a3}, \texttt{n0}, \texttt{n1}, \texttt{f0}, \ldots, \texttt{f3}, \texttt{f10}, \texttt{f11}\}$$

Spilling For modeling spilling costs we introduce function $s_{\mathbf{s}}(a)$:

$$s_{\mathbf{s}}(a) = \begin{cases} spillcost(\mathbf{s}), & \text{if } a = sp \\ 0, & \text{otherwise} \end{cases} \tag{6.1}$$

If $\mathbf{s}$ is spilled($a = sp$), function $s_{\mathbf{s}}(a)$ maps assignment a to the costs of spilling the live range. The value $spillcost(\mathbf{s})$ denotes the costs of spilling live range $\mathbf{s}$. If a register is assigned to $\mathbf{s}$ ($a \in \{R_1, \ldots, R_m\}$), function $s_{\mathbf{s}}(a)$ has zero costs.

We assume that the loop in our example is performed 10 times and sa0 is accessed 11 times. Therefore, we insert $s_{\texttt{sa0}}$ into the function set $F_{\texttt{sa0}}$. The spill-costs $spillcost(\texttt{sa0})$ evaluate to 110 by assuming that a read accesses costs 10 units if sa0 is spilled.

Register Classes The register allocator must take care which CPU register is a valid assignment for a live range. More formally, a class $class(\mathbf{s})$ is a sub-set of valid CPU registers for live range $\mathbf{s}$ and the cost function is given as follows:

$$cl_{\mathbf{s}}(a) = \begin{cases} 0, & \text{if } a \in class(\mathbf{s}) \cup \{sp\}, \\ \infty, & \text{otherwise} \end{cases} \tag{6.2}$$

The live ranges of our example also have dedicated register classes. For sake of simplicity there is a one to one relationship between live range classes and CPU register classes. E.g., live range sa0 can only be assigned one of the registers a0...a3. The class constraint of sa0 is expressed as $cl_{\mathtt{sa0}}$ with $class(\mathtt{sa0}) = \{a0, a1, a2, a3\}$. The function $cl_{\mathtt{sa0}}$ is added to $F_{\mathtt{sa0}}$.

Preferred Register Preferred registers occur when there are move relations between live ranges and CPU registers. An example is parameter passing when parameters are passed via registers. In this case one operand of the copy instruction is a CPU register rather than a live range. The copy instruction can be eliminated if the live range is assigned to the same CPU register. The following cost function models the benefit if the copy instruction can be eliminated:

$$pr_{\mathbf{s}}(a) = \begin{cases} -movecost(\mathbf{s}), & \text{if } a = pref(\mathbf{s}), \\ 0, & \text{otherwise} \end{cases} \tag{6.3}$$

where $-movecost(\mathbf{s})$ is the benefit of eliminating the copy instruction between s and the preferred CPU register $pref(\mathbf{s})$. In line 3 of our example live range sa2 is assigned to a parameter which is passed in a2. If live range sa2 is stored in CPU register a2, the copy instruction can be eliminated. Let us assume that we need 5 cost units for a copy instruction. Then, the cost function for sa2 is given by $pr_{\mathtt{sa2}}$ with $movecost(\mathtt{sa2}) = 5$ and $pref(\mathtt{sa2}) = \mathtt{a2}$. It is added to $F_{\mathtt{sa2}}$.

6.3.2. Constraints on two Live Ranges

Constraints on two live ranges are formulated by cost functions which take two locations $r_1, r_2 \in A$ as parameters. Again, the set A is an assignment set and consists of all allocatable CPU registers and a location for spilling the live range (sp).

Definition 25. *Let* $\mathbf{s}_i$ *and* $\mathbf{s}_j$ *be live ranges and* $A = \{sp, R_1, R_2, \ldots, R_m\}$ *be an assignment set. Then* $f_{\mathbf{s}_i\mathbf{s}_j}(a1, a2)$, $a1, a2 \in A$ *is the cost function describing a constraint between live ranges* $\mathbf{s}_i$ *and* $\mathbf{s}_j$.

Cost function $f_{\mathbf{s}_i\mathbf{s}_j}$ expresses the costs for two dependent live ranges. If a combination of two register assignments $a_1, a_2 \in A$ for $\mathbf{s}_i$ and $\mathbf{s}_j$ is not valid, the cost function maps a_1 and a_2 to ∞. For every pair of live ranges we have a set of cost functions denoted by $F_{\mathbf{s}_i\mathbf{s}_j}$.

In the following we list typical cost functions for two live ranges and formulate them for our running example.

Interference The most prominent constraint in register allocation is the interference constraint. The interference constraint is represented by a cost function for two live ranges as follows:

$$i_{\mathbf{s}_1\mathbf{s}_2}(a_1, a_2) = \begin{cases} 0, & \text{if } a_1 \neq a_2 \vee a_1 = sp \\ \infty, & \text{otherwise} \end{cases} \tag{6.4}$$

The cost function implies that live ranges $\mathbf{s}_1$ and $\mathbf{s}_2$ must be assigned to different CPU registers. In our example all address register live ranges interfere with each other. Therefore we have cost functions $i_{\mathtt{sa0\,sa1}}$, $i_{\mathtt{sa0\,sa2}}$, and $i_{\mathtt{sa1\,sa2}}$ which are inserted to the corresponding function sets $F_{\mathtt{sa0\,sa1}}$, $F_{\mathtt{sa0\,sa2}}$, and $F_{\mathtt{sa1\,sa2}}$.

Coalescing Coalescing is achieved by eliminating a copy instruction if source and target are assigned to the same CPU register. The costs for eliminating a copy instruction are expressed as a function of the source and destination registers,

$$c_{\mathbf{s}_1\,\mathbf{s}_2}(a_1, a_2) = \begin{cases} -\mathit{movecost}(\mathbf{s}_1, \mathbf{s}_2), & \text{if } a_1 = a_2 \wedge a_1 \neq sp \\ 0, & \text{otherwise} \end{cases} \tag{6.5}$$

where $-\mathit{movecost}$ is the benefit of eliminating the copy instruction. In our example lines 7, 9 and 14 contain copy instructions. We assume that we save 5 cost units by eliminating a copy instruction, and that the loop is executed 10 times. Then, for coalescing, the following cost functions are imposed: $c_{\mathtt{sn1\,sn0}}$, $c_{\mathtt{sn2\,sn0}}$, and $c_{\mathtt{sn3\,sn0}}$. For the first two copy instructions we have 50 cost units since the loop is executed 10 times. Therefore $\mathit{movecost}(\mathtt{sn1}, \mathtt{sn0}) = \mathit{movecost}(\mathtt{sn2}, \mathtt{sn0}) = 50$. For eliminating the copy instruction in the epilogue 5 cost units are saved, yielding $\mathit{movecost}(\mathtt{sn3}, \mathtt{sn0}) = 5$. Again, the cost functions need to be added to the corresponding function sets $F_{\mathtt{sn1\,sn0}}$, $F_{\mathtt{sn2\,sn0}}$, and $F_{\mathtt{sn3\,sn0}}$.

Paired Registers In many architectures paired registers[1] are used to implement long registers by combining two or more short registers. Paired short registers can not be used at the same time as the corresponding long register. Therefore registers impose an interference constraint between live ranges, although they belong to different register classes. For every CPU register $a \in \{R_1, \ldots, R_m\}$ we have a set of shared registers $\mathit{shared}(a)$. So the paired short registers are in the *shared* set of the long register and vice versa. The cost function is given as follows:

$$pa_{\mathbf{s}_1\,\mathbf{s}_2}(a_1, a_2) = \begin{cases} \infty, & \text{if } a_1 \in \mathit{shared}(a_2) \wedge a_1 \neq sp \wedge a_2 \neq sp \\ 0, & \text{otherwise} \end{cases} \tag{6.6}$$

[1] In [36] the term *overlapped registers* is used for paired registers.

The cost function pa yields ∞ if there is an interference between shared live ranges and none of the live ranges is spilled.

In our architecture, long floating point registers (`fl0`,`fl1`) are shared with paired short floating point registers (`f0`...`f3`). Therefore, we have an interference between `sfl0` and `sf1`, and an interference between `sfl0` and `sf2`, which are denoted by $pa_{\texttt{sfl0 sf1}}$ and $pa_{\texttt{sfl0 sf2}}$. Both are added to $F_{\texttt{sfl0 sf1}}$ and $F_{\texttt{sfl0 sf2}}$ respectively.

Coupled Registers Coupled registers impose a 1-to-1 or n-to-1 relationship between registers of different register classes. If an instruction needs two coupled registers, only one register has to be encoded in the instruction word. The register of the other class can be derived from the first register. For example, in many DSP architectures, index- or modulo-registers are coupled with address registers. We model generic dependencies of two live range operands by the following cost function

$$cp_{\mathtt{s}_1\mathtt{s}_2}(a_1, a_2) = \begin{cases} 0, & \text{if } a_1 \in \mathit{couple}(a_2) \vee a_1 = sp \vee a_2 = sp \\ \infty, & \text{otherwise} \end{cases} \tag{6.7}$$

where $\mathit{couple}(a_2)$ is a set of allowed register combinations for assignment a_1. Note that if a live range is spilled g yields zero because in this case the coupling constraint is not violated.

The indexed addressing accesses of our running example in line 8, 10 and 15 imply such constraints. In our architecture we can only couple `a0` and `a1` with `n0`, and `a2` and `a3` with `n1`. The resulting sets are $\mathit{couple}(\texttt{n0}) = \{\texttt{a0}, \texttt{a1}\}$ and $\mathit{couple}(\texttt{n1}) = \{\texttt{a2}, \texttt{a3}\}$. We have the following cost functions: $cp_{\texttt{sa0 sn1}}$, $cp_{\texttt{sa1 sn2}}$, and $cp_{\texttt{sa3 sn3}}$, which are added to $F_{\texttt{sa0 sn1}}$, $F_{\texttt{sa1 sn2}}$ and $F_{\texttt{sa3 sn3}}$ respectively.

6.4. Mapping to a PBQP

With the help of the cost functions we can now generate a PBQP model of the register allocation problem. This is done in three steps: (1) build the PBQP-graph, (2) construct the cost vectors from the functions for one live range and (3) construct the cost matrices from the functions for two live ranges.

For a live range, the allocator has to decide whether it stores the live range in one of the CPU registers $R_1, \ldots, R_m$ or it spills the live range. The elements of the decision vectors in the PBQP represent these individual assignments, which are elements of the assignment set A. So the PBQP describes the problem of deciding between CPU registers or spilling for each live range.

6.4.1. The PBQP-Graph

Each node of the PBQP-graph represents a live range. An edge is inserted between two live ranges $\mathbf{s}_i$ and $\mathbf{s}_j$, if there is at least one constraint between the two live ranges, i.e. $F_{\mathbf{s}_i\,\mathbf{s}_j} \neq \emptyset$.

The PBQP-graph is an extension of the interference graph. Both graphs have the same sets of nodes, but different set of edges. The interference graph only contains edges for the interference constraints whereas the PBQP-graph contains edges for all kind of constraints between two live ranges.

Figure 6.3 shows the PBQP-graph for our example. In the graph, interference constraints are drawn by straight lines, coalescing constraints by dashed lines, and coupling constraints by dotted lines. Moreover, the graph shows the interference constraints between all floating point register live ranges, although the type of interference differs: There is an interference of shared registers between short floats `sf1`, `sf2` and the long float `sfl0`, and a classical interference between `sf1` and `sf2`. The interferences of address register live ranges due to the indexed addressing modes are depicted as well. Coupling constraints are drawn between address register live ranges (`sa0`...`sa2`) and index register live ranges (`sn1`...`sn3`). Coalescing constraints are drawn between index register live range `sn0` and register live ranges `sn1`...`sn3`.

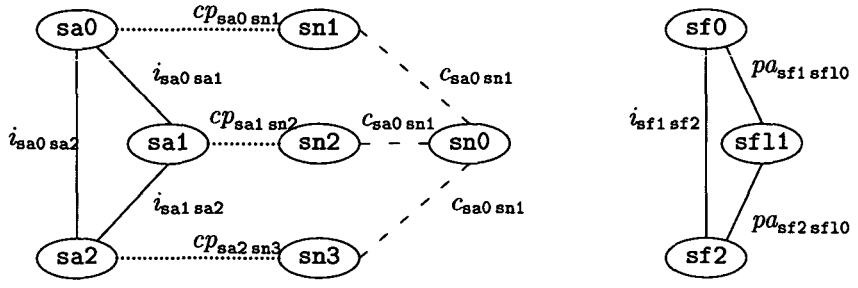


Figure 6.3.: The PBQP-graph of the running example

6.4.2. Defining Cost Vectors

The cost vectors describe the constraints for one live range. Each element corresponds to an assignment of A and contains accumulated costs of all constraints for this assignment. To construct a cost vector for live range $\mathbf{s}$, the cost functions in $F_{\mathbf{s}}$ are added and constitute cost vector $\vec{c}_{\mathbf{s}}$

$$\forall a \in A : \vec{c}_{\mathbf{s}}(\phi_a) = \sum_{f_{\mathbf{s}} \in F_{\mathbf{s}}} f_{\mathbf{s}}(a) \tag{6.8}$$

where ϕ_a is the index of assignment a.

For sake of simplicity, we have an order of live ranges and we refer to live ranges with their unique ordinal number. E.g., sa0 is the first live ranges and the cost vector of $\vec{c}_{\mathtt{sa0}}$ is equivalent to $\vec{x}_1$. We do the same for cost matrices to simplify the notation.

For example the cost vector of live range sa2 is composed by its cost functions:

$$\vec{c}_{\mathtt{sa2}}(\phi_a) = \sum_{f_{\mathtt{sa2}} \in F_{\mathtt{sa2}}} f_{\mathtt{sa2}}(a) = s_{\mathtt{sa2}}(a) + cl_{\mathtt{sa2}}(a) + pr_{\mathtt{sa2}}(a)$$

The parameters of the cost functions are: *spillcost*(sa2) = 110, *movecost*(sa2) = 5, *class*(sa2) = $\{a0, a1, a2, a3\}$ and *pref*(sa2) = a2. Let us assume that assignment a is mapped to indices ϕ_a by the following order: *sp*, a0, ..., a3, n0, ..., n3, f0, ..., f3, f10, f11. Then we obtain the following cost vector for live range sa2:

$$\vec{c}_{\mathtt{sa2}} = (110, 0, 0, -5, 0, \infty, \ldots, \infty)$$

The first element of vector $\vec{c}_{\mathtt{sa2}}$ represents the spill costs from function $s_{\mathtt{sa2}}(a)$. The second through the the 5^{th} elements indicate an allowed assignment whereas the remaining elements are disabled by class constraint $cl_{\mathtt{sa2}}(a)$ since address register live range sa2 can only be stored in CPU address registers a0 to a3. The forth element of the cost vector is the cheapest assignment due to cost function $pr_{\mathtt{sa2}}(a)$ for eliminating copy instruction in line 3 of our example.

6.4.3. Defining Cost Matrices

The cost matrices describe the constraints between two live ranges. The cost matrix C_{ij} relates to the constraints between live ranges $\mathtt{s}_i$ and $\mathtt{s}_j$. Each element $C_{ij}(k, l)$ corresponds to the relation of the assignment k of live range i to the assignment l of live range j. Each element contains the accumulated costs for all constraints of the assignments. The cost functions in $F_{\mathtt{s}_i\mathtt{s}_j}$ are added to calculate the accumulated costs. The cost matrix is computed by function sets as follows:

$$\forall a_1, a_2 \in A : C_{\mathtt{s}_1\,\mathtt{s}_2}(\phi_{a_1}, \phi_{a_2}) = \sum_{f_{\mathtt{s}_1\,\mathtt{s}_2} \in F_{\mathtt{s}_1\,\mathtt{s}_2}} f_{\mathtt{s}_1\,\mathtt{s}_2}(a_1, a_2) \tag{6.9}$$

where ϕ_{a_1} and ϕ_{a_2} are the indices of assignments a_1 and a_2, respectively.

For example, an interference constraint of sa0 and sa1 imposes the following cost matrix:

$$C = \begin{pmatrix} 0 & 0 & \ldots & 0 \\ 0 & \infty & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & \infty \end{pmatrix}.$$

The cost matrix weights pairs of decision vector elements, which belong to the same register, with infinite costs. All other pairs of decision vector elements, i.e. for different registers or spilling, have zero costs.

6.5. The PBQP Solver for Register Allocation

The PBQP solver is used to get the results from the register allocation PBQP. In contrast to the SSA-matching and AMS problems, the PBQP-graphs for register allocation are dense. Therefore the general PBQP solver must be used and the RN reductions may not be neglected.

The quality of the results from the general solver depends highly on the order for selecting nodes for the RN reduction. Determining this order can be compared to the simplify-phase of graph coloring algorithms. The RN reduction selects a register for a live range and it can be compared to the select-phase in graph coloring algorithms.

To get a good solution for the PBQP, we first perform a graph coloring algorithm to obtain a good ordering for RN reductions. In our implementation we perform the *preference directed* coloring algorithm [38] prior to the PBQP solver.[2] This pre-pass does not color the live ranges but it just yields a selection order. In the general PBQP solver this order is used for selecting the RN nodes if no degree-one and degree-two nodes are available for reduction. In addition we employ the register selection strategy of the RN reduction (Section 3.5.1) also for selecting a register in the graph coloring algorithm, because it is based on the exact model for all constraints on a live range.

If the RI and RII reductions are disabled in the PBQP solver, i.e. the RN reduction is also applied to degree-one and degree-two nodes, the result of the PBQP solver is always equivalent to the result of the graph coloring algorithm. The result of the PBQP solver (including RI and RII reductions) is equivalent or better than the result of graph coloring in most cases, because all nodes, which are reduced by RI and RII reductions, contribute to the global solution. The question is how big is the benefit we get from the RI and RII reductions. Our experiments have shown that only for about 10% of all graphs the PBQP solver is better than graph coloring. Therefore we allow recursive enumeration for one RN node. The overhead for the recursive enumeration is marginal (the execution time for the solver is far below one second in most cases) but the percentage of better results is about 70% of all graphs. So we can state that the PBQP solver is able to yield better results than graph coloring because (1) recursive enumeration can be applied and (2) the reduction rules RI and RII improve the global solution instead of selecting a local solution.

Figure 6.4 shows the reduction sequence of the example PBQP-graph. In the first step (b) nodes with a degree of two are reduced. In the next step (c) `sf2` can be

[2] Our method does not depend on the preference directed graph coloring algorithm. Any other graph coloring approach can be used.

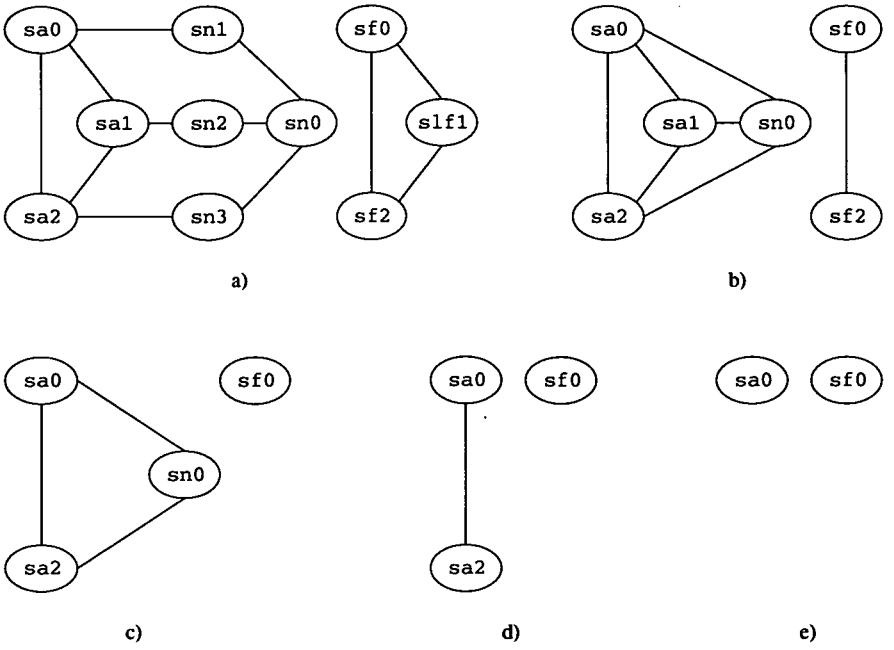


Figure 6.4.: Reduction sequence of the example

eliminated with the RI rule, but for the address register subgraph no nodes are left with degree one or two. Therefore rule RN is applied to **sa1**. Then **sn0** can be eliminated (d). By removing **sa2** the trivial graph remains (e).

The solution $S = \langle s_1, ..., s_n \rangle$, obtained from the PBQP solver, yields the register allocation. Live range $\mathbf{s}_i$ is mapped to the the i^{th} assignment in A. This can be either a CPU register $R_1, \ldots, R_m$ or *sp*. If the PBQP solver selects *sp* for some nodes, the live ranges have to be spilled. Spill code must be inserted and the register allocator restarts.

The final code with allocated registers for the example is shown in Figure 6.2(b).

6.6. Experimental Results

We have integrated the PBQP register allocator into the Atair C-Compiler for the NEC uPD77050 DSP family (see Section 4.6). We used the compiler only for generating the problem graphs but we didn't use the register model of the uPD77050 architecture. Instead we defined a synthetic register model which employs all kinds of register constraints, presented in this chapter. The register model contains small registers, shared with large registers. Adjacent small registers are paired and can be used as large registers, similar to the floating point register in our example. On the other hand, this 2-to-1 relationship between small and large registers also imposes coupled constraints,

	min	max	avg
nodes	4	1230	81.42
interferences	3	4739	289.60
coalescing	0	122	9.59
coupling	0	131	10.82
pairing	3	4782	285.55

Table 6.1.: Problem sizes

like the address-index register coupling in the example. Register class constraints are introduced by interferences with CPU registers, e.g. if a live range overlaps a function call. Register preferences are introduced by copy instructions between live ranges and CPU registers.

We compare the PBQP register allocation with the preference-directed graph coloring approach which represents one of the latest improvements to Chaitin based register allocators so far. We also integrated Smith and Holloway's approach for handling pairing constraints into the simplification-phase of the preference-directed register allocator. In our PBQP register allocator we used exactly the same preference-directed algorithm for determining the selection order for RN-reductions. In addition the color selection strategy of the reference algorithm is equivalent to the local minimum selection of the RN reduction in the PBQP algorithm.

For evaluation we use the graphs, produced by compiling the three DSP applications which are also used for the code selection and AMS evaluation (see Section 4.6). There is a total number of 273 graphs in the benchmarks.

In Table 6.1 we show the problem sizes of the graphs. For each quantity the minimum, maximum and average value is given. The table contains the number of live ranges ("nodes"), the number of interference constraints ("interferences"), the number of copy instructions ("coalescing"), the number of instructions resulting in coupling constraints ("coupling") and the number of pairing constraints ("pairing").

We performed the evaluation in different configurations by changing following parameters:

- We used two cost models. The first model is used to optimize for minimal execution time ("dynamic"). The costs of spill and reload instructions are three and the costs for copy instructions are one. The instruction costs are multiplied by an estimated execution weight of the containing basic block. The second cost model is used to optimize for minimal code size ("static"). The costs of all instructions (spill, reload and copy) are one and the costs are not multiplied by the execution weights.

- Two register set sizes are used. The first set contains 16 small registers or alternatively 8 large register ("16/8"). The second set contains 32 small register

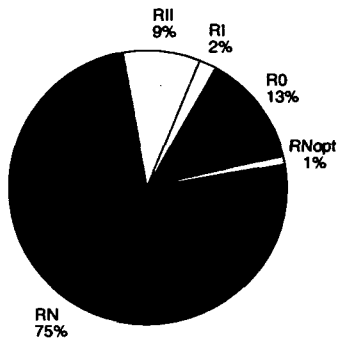


Figure 6.5.: Reduction statistics

or 16 large registers ("32/16").

- The PBQP solver was used in one configuration which allows the recursive enumeration of one RN node ("enum1") and a second configuration which allows the enumeration of two RN nodes ("enum2").

First we give the statistics about the distribution of reductions in Figure 6.5. The distribution is nearly equivalent in all configurations, except that in "enum2" the number of RN nodes, eliminated by recursive enumeration ("RNopt"), is doubled. In contrast to the statistics in SSA-graph matching and addressing mode selection, most of the reductions are RN reductions. This implies that there is no guarantee of optimality in most cases.

In the following we present statistics in comparison with the reference graph coloring approach. Table 6.2 give an overview of the evaluation results. Each row shows the result of a different configuration. The column "avg impr" gives the average improvement. It is the average ratio of the costs obtained from the preference-directed allocator and the costs obtained by the PBQP allocator. The next two columns list the average and maximum solve time of the PBQP solver in seconds. The last three columns list the number of graphs, where the PBQP result is worse ("–"), equal ("=") and better ("+") than the result of the reference algorithm.

For all configurations the average improvement is significant (from a factor of 1.53 up to 2.18). As expected, the "enum2" configuration gives better average results than the "enum1" (about 4% to 8%). But interestingly with "enum2", more results are worse than with "enum1". This is because recursive enumeration is done at the beginning in the reduction sequence, and the color selection in both algorithms can go in different directions – with bad luck it goes in the wrong direction. On the other hand this difference causes the better results of the PBQP allocator. Note that if there is no recursive enumeration and no RI and RII rules, both algorithms yield the same results.

enum nodes	num regs	cost model	avg impr	solvetime avg	solvetime max	graph −	graph =	graph +
enum1	16/8	dynamic	1.91	0.14	8.72	2	85	186
		static	1.29	0.16	9.73	3	79	189
	32/16	dynamic	2.18	0.42	11.96	2	78	193
		static	1.53	0.16	9.73	4	79	190
enum2	16/8	dynamic	2.07	0.76	21.42	16	71	186
		static	1.34	0.16	9.73	5	78	190
	32/16	dynamic	2.31	5.44	212.83	6	68	199
		static	1.63	4.55	195.39	3	68	202

Table 6.2.: Evaluation summary

The comparison with different numbers of registers indicates that the PBQP algorithm performs better with a larger number of registers. The reference algorithm is dominated by the goal of eliminating spills. As with a larger number of registers, the spilling problem has less impact and satisfying the other constraints get more important. The PBQP algorithm handles all constraints in the same manner and therefore it can perform better with a larger number of registers.

The dynamic cost model has larger performance improvements than the static cost model, because in the presence of loops the dynamic costs can get large. The static model reflects the number of eliminated spill, restore and copy instructions. The PBQP allocator produces 23% to 39% less spill, restore and copy instructions than the reference algorithm.

The runtime of the PBQP solver is very small in most cases because the complexity is linear with the number of live ranges. As recursive enumeration adds an exponential complexity, the the solve times for the "enum2" configuration are considerable larger than for "enum1" in some cases. For a larger number of registers the solve times increase significantly, because the complexity for RII reductions is $O(n^3)$ where n is the number of CPU registers.

Four configurations are shown in more detail in Figures 6.6, 6.7, 6.8 and 6.9. Each unit on the X-axis represents a single graph. The graphs are sorted from worst to best result so that the functions are monotonic increasing. The Y-axis is the ratio of the costs from the reference algorithm and the costs of the PBQP algorithm. The green lines indicate the average improvement. The blue vertical lines indicate the borders between worse, equal and better results.

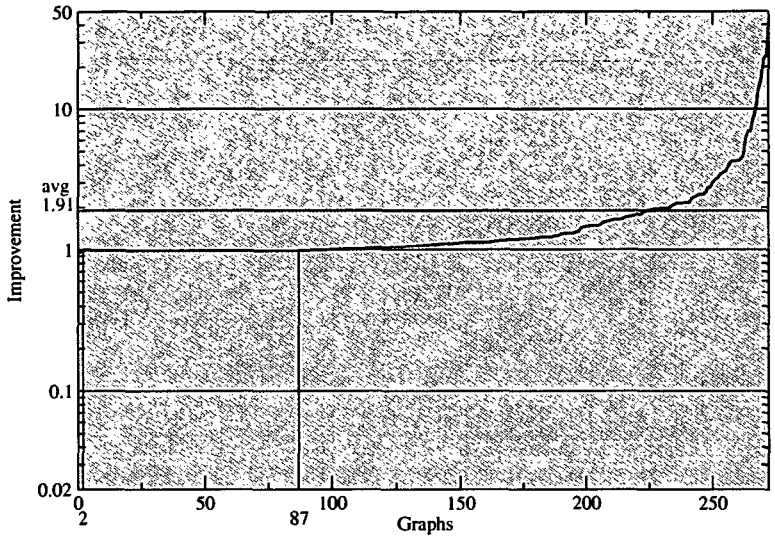


Figure 6.6.: Comparison using the configuration "enum1, 16/8, dynamic"

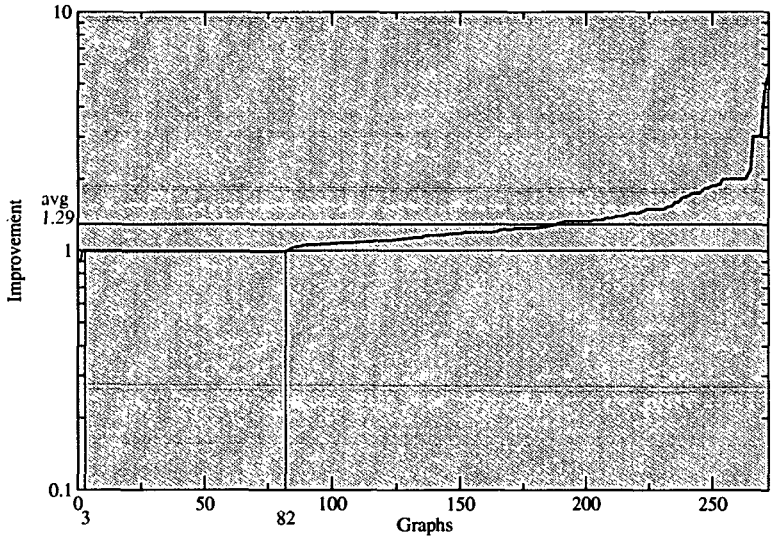


Figure 6.7.: Comparison using the configuration "enum1, 16/8, static"

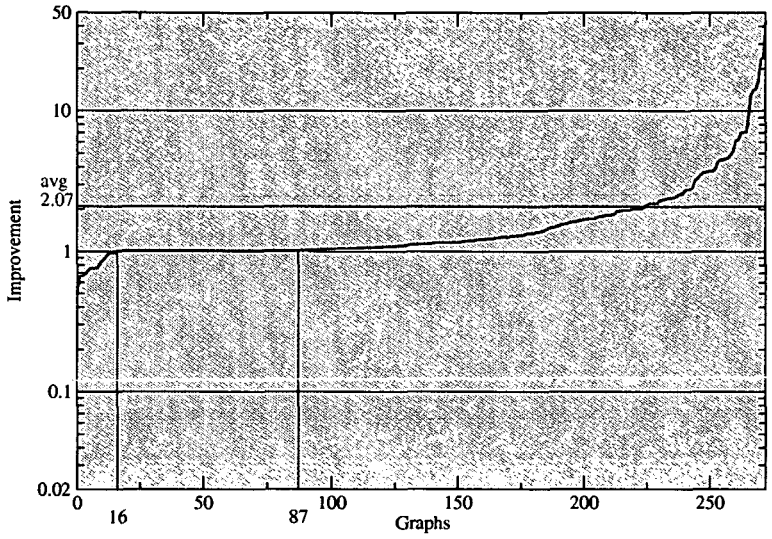


Figure 6.8.: Comparison using the configuration "enum2, 16/8, dynamic"

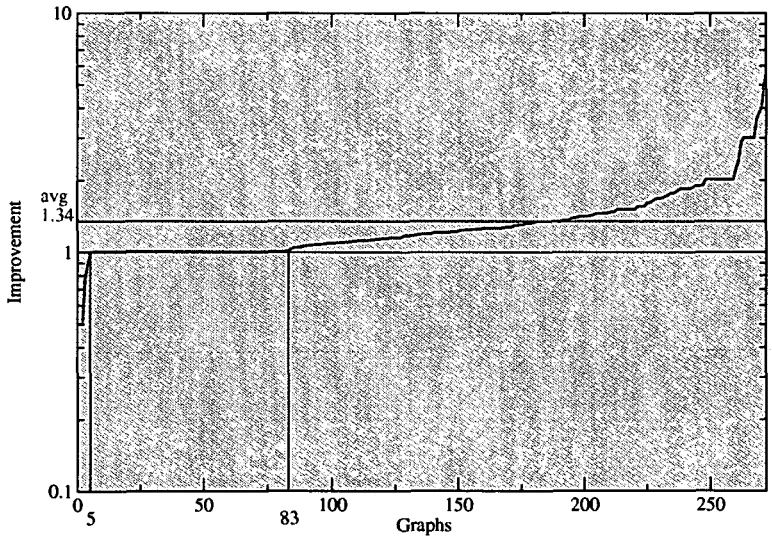


Figure 6.9.: Comparison using the configuration "enum2, 16/8, static"

7. Conclusion

Compilers for embedded systems have to meet different requirements than compilers for conventional computer systems. First, embedded systems impose irregular architectural features, which must be handled by the compiler. Second, code size optimization is as important as performance optimization. The optimizations in the compiler must be configurable in respect to the optimization goal. Third, a compiler for an embedded system is expected to generate code quality which is comparable to hand optimized assembly code. In most cases the resources of an embedded system are very restricted, so that the overhead caused by automatic code generation is not acceptable.

Compiler optimization techniques have emerged from regular RISC architectures. Unfortunately these techniques can not be efficiently reused for irregular embedded architectures. This work concentrates on code generation and optimization for digital signal processors (DSPs).

A novel approach, which is used for a whole set of compiler optimization problems, has been presented. A framework based on *partitioned boolean quadratic problems* (PBQP) is used to handle the problems of *code selection*, *addressing mode selection* and *register allocation*. Although all three problems are NP-complete, the PBQP solver is able to yield an optimal or near optimal solution in almost linear time. The quality of the solution depends on the PBQP-graph, which is used to formulate an optimization problem. If the graph is reducible, using reduction rules of the solver, the obtained solution is optimal. Otherwise heuristics are used to approximate the optimal solution.

Code selection For irregular architectures such as DSPs, the code selection phase in the compiler contributes significantly to the performance of the compiler. With traditional tree pattern matchers only separate data flow trees of a function can be matched, which has a negative impact on the quality of the code. Only if the whole computational flow of a function is taken into account, the matcher is able to generate optimal code.

We used the PBQP solver to match the whole SSA-graph of a function. In the PBQP-model the SSA-graph is equivalent to the PBQP-graph. It turned out that the PBQP-graphs of real-world DSP benchmark programs are reducible in many cases and only for a few nodes must the heuristics be applied. So the obtained solution is near optimal.

Our experiments have shown that the performance gain of a SSA-graph matcher compared to a tree pattern matcher is significant (up to 82%) in comparison to classical tree matching methods. These results were obtained without modifying the grammar. Though the overhead of the PBQP solver is higher than tree matching methods, the compile time overhead is within acceptable bounds.

Addressing Mode Selection The addressing mode selection (AMS) optimization tries to select the best addressing modes for the instructions of the input program. With the help of the PBQP formulation, we are able to describe a wide set of possible addressing modes which can be found on DSP architectures.

In our mapping to the PBQP, we derive the PBQP-graph from the control flow graph (CFG) of a function. Because CFGs are defined by the control structures of the high level input language, the resulting PBQP-graphs are reducible in almost all cases. The number of non-reducible nodes is negligible and therefore the resulting solution is optimal for nearly all input programs.

As the runtime improvements are up to 60% and code size reductions up to 50%, we can state that the optimization is of vital importance for architectures which provide complex addressing mode features.

Register Allocation Irregular architectures impose constraints on the register allocator, which can not be described by traditional graph coloring approaches. We have formulated the global register allocation problem for irregular architectures as PBQP. We are able to formulate any constraints on one or between two live ranges as cost functions.

The PBQP-graph in this model is an extension to the interference graph. It contains edges between live ranges which are related by a constraint. Because there may be many constraints between live ranges the nodes of the graph have high degrees. Therefore many nodes are not reducible and the reduction rule with heuristics must be applied in many cases. The reduction sequence is selected by applying the simplification heuristics used in traditional graph coloring algorithms. Therefore the PBQP approach can also benefit from good coloring characteristics of graph coloring algorithms.

In our experiments we have shown that for a DSP with a non orthogonal register set the generated code is better in many cases than using the graph coloring register allocator. This is because with the graph coloring approach it is not possible to describe the register constraints of the architecture.

The main observation from our PBQP optimization implementations is that the reducibility of the PBQP-graphs is the key to good optimization result. We can state following order of reducibility:

1. Addressing mode selection and all mode selection problems in general: the PBQP-graph is derived from the CFG. It is reducible in almost all cases. The number of non reducible nodes can be neglected. Therefore the PBQP approach is ideal for this kind of NP-complete problems.

2. Code selection: There is a small fraction of non-reducible nodes in the SSA-graphs, but the heuristic used on irreducible nodes still yields near optimal results. So the PBQP approach is a very good method to solve the code selection problem.

3. Register allocation: The graphs resulting from various constraints on live ranges are very dense in general. The heuristics for non-reducible nodes must be applied often. Therefore the reduction sequence for non-reducible nodes is adopted from graph coloring algorithms to get a good solution. Although the results are better than with graph coloring, an optimal solution can not be guaranteed in most cases.

The possibility of getting optimal or near optimal results for NP-complete optimization problems is the main advantage of the PBQP approach. Beside of this there are other benefits which contribute to the success of the PBQP optimizers.

As the PBQP models work with cost models, it is easy to configure the optimizers. First it is easy to model various architectural irregularities, i.e. it is not necessary to implement new algorithms to handle irregularities. Second, the optimization goal – performance or code size – can be selected simply by adapting the cost model.

As another benefit it turned out that the implementation effort is reduced dramatically, because the PBQP solver can be reused for all PBQP applications in the compiler. Therefore the implementation task is restricted to defining the model of the optimization problems. Re-targeting the optimization just needs adapting the cost model to a new architecture.

A. Proofs

Lemma 1 in Section 3.4.1

Proof. Let $\vec{x}$ be a decision vector of degree one which is eliminated by rule RI: $RI(f^k(X^k), \vec{x})) = f^{k+1}(X^{k+1})$. Vector $\vec{y} \in adj(\vec{x})$ is the adjacent vector of $\vec{x}$.

First we introduce the reduced decision vector into Equation 3.15. Because the multiplications with $\vec{x}$ select the i^{th} element of the cost vectors, where i is the index of the one-element in $\vec{x}$, the vector minimum can be replaced by a partial minimum. We also replace the dot product $C_{yx}(i,:) \cdot \vec{x}^T$ by the equivalent dot product $\vec{x} \cdot C_{xy}(:,i)$. According to Equation 3.8 we get

$$\vec{\delta}(i) = \min_{\vec{x}} \left[\vec{x} \cdot C_{xy}(:,i) + \vec{c_x} \cdot \vec{x}^T\right]$$

As $\vec{\delta} \cdot \vec{y}^T = \vec{\delta}(i)$ and $C_{xy}(:,i) = C_{xy} \cdot \vec{y}^T$ if i is the one-element in $\vec{y}$, we obtain Equation A.1.

$$\vec{\delta} \cdot \vec{y}^T = \min_{\vec{x}} \left[\vec{x} \cdot C_{xy} \cdot \vec{y}^T + \vec{c_x} \cdot \vec{x}^T\right] \; \forall \, \vec{y} \in D^y \tag{A.1}$$

We define a helper function *h1* as follows:

$$h1(X^{k\backslash x}) = f(X^k) - \vec{x} \cdot C_{xy} \cdot \vec{y}^T - \vec{c}_x \cdot \vec{x}^T \tag{A.2}$$

Function *h1* eliminates all contributions of vector $\vec{x}$ from the objective function f. Therefore the parameter of *h1* is $X^{k\backslash x}$ instead of X^k. From the left side of Equation 3.14 we split the minimum operator by applying Equations 3.5 and 3.6.

$$\min f^k(X^k) = \min_{X^{k\backslash x}} \min_{\vec{x}} f^k(X^k)$$

Next we substitute f^k with the helper function *h1*.

$$\min f^k(X^k) = \min_{X^{k\backslash x}} \min_{\vec{x}} \left[h1(X^{k\backslash x}) + \vec{x} \cdot C_{xy} \cdot \vec{y}^T + \vec{c}_x \cdot \vec{x}^T\right]$$

As $h1(X^{k\backslash x})$ does not depend on $\vec{x}$ we obtain

$$\min f^k(X^k) = \min_{X^{k\backslash x}} \left[h1(X^{k\backslash x}) + \min_{\vec{x}}(\vec{x} \cdot C_{xy} \cdot \vec{y}^T + \vec{c}_x \cdot \vec{x}^T)\right]$$

According to Equation A.1 we get

$$\min f^k(X^k) = \min_{X^{k\backslash x}} \left[h1(X^{k\backslash x}) + \vec{\delta} \cdot \vec{y}^T\right]$$

After back substitution of *h1* we get

$$\min f^k(X^k) = \min_{X^{k\backslash x}} \left[f(X^k) - \vec{x} \cdot C_{xy} \cdot \vec{y}^T - \vec{c}_x \cdot \vec{x}^T + \vec{\delta} \cdot \vec{y}^T\right]$$

According to Definition 7 we obtain

$$\min f^k(X^k) = \min_{X^{k\setminus x}} f^{k+1}(X^{k\setminus x})$$

which is equivalent to

$$\min f^k(X^k) = \min f^{k+1}(X^{k+1})$$

□

Lemma 2 in Section 3.4.1

Proof. Let $\vec{x}$ be a decision vector of degree one which is eliminated by rule RI: $RI(f^k(X^k), \vec{x})) = f^{k+1}(X^{k+1})$. Vector $\vec{y} \in adj(\vec{x})$ is the adjacent vector of $\vec{x}$. Applying Equation 3.11 to the k^{th} reduced objective function yields

$$\min f^k(X^k) = f^k(\bar{X}^k)$$

We substitute f^k with the helper function *h1* as defined in Equation A.2.

$$\min f^k(X^k) = h1(\bar{X}^{k\setminus x}) + \bar{\vec{x}} \cdot C_{xy} \cdot \bar{\vec{y}}^T + \vec{c}_x \cdot \bar{\vec{x}}^T$$

Next we introduce a minimum operator for the reduced vector $\vec{x}$ according to Equation 3.7.

$$\min f^k(X^k) = \min_x \left[h1(\bar{X}^{k\setminus x}) + \vec{x} \cdot C_{xy} \cdot \bar{\vec{y}}^T + \vec{c}_x \cdot \vec{x}^T \right]$$

As *h1* does not depend on $\vec{x}$, we get

$$\min f^k(X^k) = h1(\bar{X}^{k\setminus x}) + \min_x \left[\vec{x} \cdot C_{xy} \cdot \bar{\vec{y}}^T + \vec{c}_x \cdot \vec{x}^T \right]$$

According to Equation A.1 we get

$$\min f^k(X^k) = h1(\bar{X}^{k\setminus x}) + \vec{\delta} \cdot \bar{\vec{y}}^T$$

After back substitution of *h1* we obtain

$$\min f^k(X^k) = f(\bar{X}^k) - \bar{\vec{x}} \cdot C_{xy} \cdot \bar{\vec{y}}^T - \vec{c}_x \cdot \bar{\vec{x}}^T + \vec{\delta} \cdot \bar{\vec{y}}^T$$

According to Definition 7 this is equivalent to

$$\min f^k(X^k) = f^{k+1}(\bar{X}^{k\setminus x})$$

The minima of the k^{th} and $k+1^{th}$ reduced functions are equal according to Lemma 1 and we can replace the left side of the equation.

$$\min f^{k+1}(X^{k+1}) = f^{k+1}(\bar{X}^{k\backslash x})$$

Therefore the solution of the k^{th} reduced function – excluding the reduced vector – is also a solution to the $k+1^{th}$ reduced function and we can write

$$\bar{X}^{k\backslash x} = \bar{X}^{k+1}$$

□

Lemma 3 in Section 3.4.1

Proof. Let $\vec{x}$ be a decision vector of degree two which is eliminated by rule RII: $RII(f^k(X^k), \vec{x})) = f^{k+1}(X^{k+1})$. Vectors $\vec{y}, \vec{z} \in adj(\vec{x})$ are the adjacent vectors of $\vec{x}$.

First we introduce the reduced decision vector into Equation 3.16. Because the multiplications with $\vec{x}$ select the i^{th} element of the cost vectors, where i is the index of the one-element in $\vec{x}$, the vector minimum can be replaced by a partial minimum. We also replace the dot products $C_{yx}(i,:) \cdot \vec{x}^T$ and $C_{zx}(j,:) \cdot \vec{x}^T$ by the equivalent dot products $\vec{x} \cdot C_{xy}(:,i)$ and $\vec{x} \cdot C_{xz}(:,j)$, respectively. According to Equation 3.8 we get

$$\Delta(i,j) = \min_{\vec{x}} \left[\vec{x} \cdot C_{xy}(:,i) + \vec{x} \cdot C_{xz}(:,j) + \vec{c_x} \cdot \vec{x}^T\right]$$

As $\vec{y} \cdot \Delta \cdot \vec{z}^T = \Delta(i,j)$, $C_{xy}(:,i) = C_{xy} \cdot \vec{y}^T$ and $C_{xz}(:,j) = C_{xz} \cdot \vec{z}^T$ (i and j are the indices of the one-elements in $\vec{y}$ and $\vec{z}$, respectively), we obtain Equation A.3.

$$\vec{y} \cdot \Delta \cdot \vec{z}^T = \min_{\vec{x}} \left[\vec{x} \cdot C_{xy} * \vec{y}^T + \vec{x} \cdot C_{xz} \cdot \vec{z}^T + \vec{c_x} \cdot \vec{x}^T\right] \; \forall \, \vec{y} \in D^y, \vec{z} \in D^z \qquad \text{(A.3)}$$

We define a helper function $h2$ as follows:

$$h2(X^{k\backslash x}) = f(X^k) - \vec{x} \cdot C_{xy} \cdot \vec{y}^T - \vec{x} \cdot C_{xz} \cdot \vec{z}^T - \vec{c}_x \cdot \vec{x}^T \qquad \text{(A.4)}$$

Function $h2$ eliminates all contributions of vector $\vec{x}$ from the objective function f. Therefore the parameter of $h2$ is $X^{k\backslash x}$ instead of X^k. From the left side of Equation 3.14 we split the minimum operator by applying Equations 3.5 and 3.6.

$$\min f^k(X^k) = \min_{X^{k\backslash x}} \min_{\vec{x}} f^k(X^k)$$

Next we substitute f^k with the helper function $h2$.

$$\min f^k(X^k) = \min_{X^{k\backslash x}} \min_{\vec{x}} \left[h2(X^{k\backslash x}) + \vec{x} \cdot C_{xy} \cdot \vec{y}^T + \vec{x} \cdot C_{xz} \cdot \vec{z}^T + \vec{c}_x \cdot \vec{x}^T\right]$$

As $h2(X^{k\backslash x})$ does not depend on $\vec{x}$ we obtain

$$\min f^k(X^k) = \min_{X^{k\backslash x}} \left[h2(X^{k\backslash x}) + \min_{\vec{x}} (\vec{x} \cdot C_{xy} \cdot \vec{y}^T + \vec{x} \cdot C_{xz} \cdot \vec{z}^T + \vec{c}_x \cdot \vec{x}^T) \right]$$

According to Equation A.3 we get

$$\min f^k(X^k) = \min_{X^{k\backslash x}} \left[h2(X^{k\backslash x}) + \vec{y} \cdot \Delta \cdot \vec{z}^T \right]$$

After back substitution of $h2$ we get

$$\min f^k(X^k) = \min_{X^{k\backslash x}} \left[f(X^k) - \vec{x} \cdot C_{xy} \cdot \vec{y}^T - \vec{x} \cdot C_{xz} \cdot \vec{z}^T - \vec{c}_x \cdot \vec{x}^T + \vec{y} \cdot \Delta \cdot \vec{z}^T \right]$$

According to Definition 9 we obtain

$$\min f^k(X^k) = \min_{X^{k\backslash x}} f^{k+1}(X^{k\backslash x})$$

which is equivalent to

$$\min f^k(X^k) = \min f^{k+1}(X^{k+1})$$

□

Lemma 4 in Section 3.4.1

Proof. Let $\vec{x}$ be a decision vector of degree two which is eliminated by rule RII: $RII(f^k(X^k), \vec{x})) = f^{k+1}(X^{k+1})$. Vectors $\vec{y}, \vec{z} \in adj(\vec{x})$ are the adjacent vectors of $\vec{x}$. Applying Equation 3.11 to the k^{th} reduced objective function yields

$$\min f^k(X^k) = f^k(\bar{X}^k)$$

We substitute f^k with the helper function $h2$ as defined in Equation A.4.

$$\min f^k(X^k) = h2(\bar{X}^{k\backslash x}) + \bar{\vec{x}} \cdot C_{xy} \cdot \bar{\vec{y}}^T + \bar{\vec{x}} \cdot C_{xz} \cdot \bar{\vec{z}}^T + \vec{c}_x \cdot \bar{\vec{x}}^T$$

Next we introduce a minimum operator for the reduced vector $\vec{x}$ according to Equation 3.7.

$$\min f^k(X^k) = \min_{x} \left[h2(\bar{X}^{k\backslash x}) + \vec{x} \cdot C_{xy} \cdot \bar{\vec{y}}^T + \vec{x} \cdot C_{xz} \cdot \bar{\vec{z}}^T + \vec{c}_x \cdot \vec{x}^T \right]$$

As $h2$ does not depend on $\vec{x}$, we get

$$\min f^k(X^k) = h2(\bar{X}^{k\backslash x}) + \min_{x} \left[\vec{x} \cdot C_{xy} \cdot \bar{\vec{y}}^T + \vec{x} \cdot C_{xz} \cdot \bar{\vec{z}}^T + \vec{c}_x \cdot \vec{x}^T \right]$$

According to equation A.3 we get

$$\min f^k(X^k) = h2(\bar{X}^{k\setminus x}) + \bar{\vec{y}} \cdot \Delta \cdot \bar{\vec{z}}^T$$

After back substitution of *h2* we obtain

$$\min f^k(X^k) = f(\bar{X}^k) - \bar{\vec{x}} \cdot C_{xy} \cdot \bar{\vec{y}}^T - \bar{\vec{x}} \cdot C_{xz} \cdot \bar{\vec{z}}^T - \vec{c}_x \cdot \bar{\vec{x}}^T + \bar{\vec{y}} \cdot \Delta \cdot \bar{\vec{z}}^T$$

According to Definition 7 this is equivalent to

$$\min f^k(X^k) = f^{k+1}(\bar{X}^{k\setminus x})$$

The minima of the k^{th} and $k+1^{th}$ reduced functions are equal according to Lemma 3 and we can replace the left side of the equation.

$$\min f^{k+1}(X^{k+1}) = f^{k+1}(\bar{X}^{k\setminus x})$$

Therefore the solution of the k^{th} reduced function – excluding the reduced vector – is also a solution to the $k+1^{th}$ reduced function and we can write

$$\bar{X}^{k\setminus x} = \bar{X}^{k+1}$$

□

Lemma 5 in Section 3.4.3

Proof. We define a helper function *hb* as follows:

$$hb(X^{k\setminus x}) = f(X) - \vec{c}_x \cdot \vec{x}^T - \sum_{\vec{y} \in adj(\vec{x})} \vec{x} \cdot C_{xy} \cdot \vec{y}^T \tag{A.5}$$

The function *hb* contains all terms of $f(X)$ expect the summands of the reduced decision vector $\vec{x}$. In the first step we substitute f with *hb*.

$$f^k(\bar{X}^k) = hb(\bar{X}^{k\setminus x}) + \vec{c}_x \cdot \bar{\vec{x}}^T + \sum_{\vec{y} \in adj(\vec{x})} \bar{\vec{x}} \cdot C_{xy} \cdot \bar{\vec{y}}^T$$

By introducing a minimum operator for the reduced vector $\vec{x}$ according to Equation 3.7 we obtain

$$f^k(\bar{X}^k) = \min_{\vec{x}} \left[hb(\bar{X}^{k\setminus x}) + \vec{c}_x \cdot \bar{\vec{x}}^T + \sum_{\vec{y} \in adj(\vec{x})} \vec{x} \cdot C_{xy} \cdot \bar{\vec{y}}^T \right]$$

As *hb* does not depend on $\vec{x}$ we get

$$f^k(\bar{X}^k) = hb(\bar{X}^{k\setminus x}) + \min_{\vec{x}} \left[\vec{c}_x \cdot \vec{x}^T + \sum_{\vec{y} \in adj(\vec{x})} \vec{x} \cdot C_{xy} \cdot \vec{\bar{y}}^T \right]$$

According to the properties of quadratic forms (Equation 3.1 and Equation 3.2) we can apply the distributive law and get

$$f^k(\bar{X}^k) = hb(\bar{X}^{k\setminus x}) + \min_{\vec{x}} \left[(\vec{c}_x + \sum_{\vec{y} \in adj(\vec{x})} \bar{\vec{y}} \cdot {C_{xy}}^T) \cdot \vec{x}^T \right]$$

The multiplication with $\vec{x}^T$ selects the s^{th} element where s is the index of the one-element in $\vec{x}$. Therefore we can express the solution $\bar{\vec{x}}$ with the minimum index of the minimum argument vector in the previous equation.

$$s = i_{min} \left[\vec{c}_x + \sum_{\vec{y} \in adj(\vec{x})} \bar{\vec{y}} \cdot {C_{xy}}^T \right]$$

□

Theorem 2 in Section 3.4.4

Proof. Let $C = A + B$ where $A(i,j) = \vec{u}(i)\ \forall\ j \in 1...m$ and $B(i,j) = \vec{v}(j)\ \forall\ i \in 1...n$. Then $C(i,j) = A(i,j) + B(i,j) = \vec{u}(i) + \vec{v}(j)$. We can write

$$\vec{x} \cdot C \cdot \vec{y}^T = \vec{x} \cdot (A + B) \cdot \vec{y}^T$$

Applying the distributive law (Equation 3.2) we get

$$\vec{x} \cdot C \cdot \vec{y}^T = \vec{x} \cdot A \cdot \vec{y}^T + \vec{x} \cdot B \cdot \vec{y}^T$$

The term $A \cdot \vec{y}^T$ can be expressed as $A(:,j)^T$ where j is the index of the one-element in $\vec{y}$. Matrix A has identical column vectors $A(:,j) = \vec{u}\ \forall\ j \in 1...m$. Similar $\vec{x} \cdot B$ can be expressed as $B(i,:)$ where i is the index of the one-element in $\vec{x}$. Matrix B has identical row vectors $B(i,:) = \vec{v}\ \forall\ i \in 1...n$. Therefore we can write

$$\vec{x} \cdot C \cdot \vec{y}^T = \vec{x} \cdot \vec{u}^T + \vec{v} \cdot \vec{y}^T$$

According to Equation 3.1 this is equivalent to

$$\vec{x} \cdot C \cdot \vec{y}^T = \vec{u} \cdot \vec{x}^T + \vec{v} \cdot \vec{y}^T$$

□

Bibliography

[1] Web portal for research in address code optimization. *http://www.address-code-optimization.org.*

[2] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. In *Journal of the ACM*, volume 23, pages 488–501. 1976.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architecture.* Morgan Kaufmann Publishers, 2002.

[5] W. Ambrosch, M. A. Ertl, F. Beer, and A. Krall. Dependence-conscious global register allocation. In Jürg Gutknecht, editor, *Programming Languages and System Architectures*, volume 782 of *Lecture Notes in Computer Science*, pages 125–136. Springer Verlag, March 1994.

[6] G. Araujo, A. Sudarsanam, and S. Malik. Instruction set design and optimizations for address computation in DSP architectures. In *ISSS*, pages 105–107, 1996.

[7] 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.

[8] D. H. Bartlay. Optimizing stack frame accesses for processors with restricted addressing modes. *Software – Practice and Experience*, 22(2):101–110, February 1992.

[9] P. Briggs. Register allocation via graph coloring. Technical Report TR92-183, 24, 1998.

[10] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, May 1994.

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

[12] G. J. Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices*, 17(6):98–105, June 1982.

[13] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[14] M. Cintra and G. Araujo. Array reference allocation using SSA-form and live range growth. *Lecture Notes in Computer Science*, 1985:48+, 2001.

[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM Press, 1989.

[16] E. Eckstein, O. König, and B. Scholz. Code instruction selection based on ssa-graphs. In *Proceedings of the SCOPES*. IEEE/ACM, September 2003.

[17] E. Eckstein and A. Krall. Minimizing cost of local variables access for DSP-processors. In Y. Annie Liu and Reinhard Wilhelm, editors, *LCTES'99 Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 34(7), pages 20–27, Atlanta, 1999.

[18] E. Eckstein and B. Scholz. Address mode selection. In *Proceedings of the International Symposium of Code Generation and Optimization (CGO'03)*, San Francisco, March 2003. IEEE/ACM.

[19] H. Emmelmann, F.-W. Schröer, and R. Landwehr. BEG – a generator for efficient back ends. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 227–237. ACM Press, 1989.

[20] M. A. Ertl. Optimal code selection in DAGs. In *Principles of Programming Languages (POPL '99)*, 1999.

[21] M. Farach and V. Liberatore. On local register allocation. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 564–573. Society for Industrial and Applied Mathematics, 1998.

[22] C. W. Fraser and D. R. Hanson. A code generation interface for ANSI C. *Software - Practice and Experience*, 21(9):963–988, 1991.

[23] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(3):213–226, 1992.

[24] C. W. Fraser, R. R. Henry, and T. A. Proebsting. Burg: fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, 1992.

[25] C. Fu and K. Wilken. A faster optimal register allocator. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 245–256. IEEE Computer Society Press, 2002.

[26] L. George and A. W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[27] M. P. Gerlek, E. Stoltz, and M. 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, January 1995.

[28] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *ACM SIGPLAN 86 Symposium on Compiler Construction*, pages 11–16. June 1986.

[29] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in Matlab: design and implementation. Technical report, 1981.

[30] R. S. Glanville and S. L. Graham. A new method for compiler code generation. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 231–254. ACM Press, 1978.

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

[32] U. Hirnschrott, A. Krall, and B. Scholz. Graph coloring vs. optimal register allocation for optimizing compilers. In *Proceedings of the Joint Modular Languages Conference*, 2003.

[33] M. Ikekawa. The upd77050: A new low-power dsp for mobile multimedia applications. *Embedded Processor Forum 2002*, 2002.

[34] S. C. Johnson. Yacc: Yet another compiler-compiler. 1978.

[35] R. M. Karp. *Complexity of Computer Computations.* Plenum Press, New York, NY, 1972.

[36] T. Kong and K. D. Wilken. Precise register allocation for irregular register architectures. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 297–307, Los Alamitos, November 30–December 2 1998. IEEE Computer Society.

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

[38] A. Koseki, H. Komatsu, and T. Nakatani. Preference-directed graph coloring. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 33–44. ACM Press, 2002.

[39] T. Kumura, D. Ishii, M. Ikekawa, I. Kuroda, and M. Yoshida. A low-power programmable dsp core architecture for 3g mobile terminals. *ICASSP 2001 (International Conference on Acoustics, Speech and Signal Processing)*, 2001.

[40] S. M. Kurlander and C. N. Fischer. Minimum cost interprocedural register allocation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 230–241. ACM Press, 1996.

[41] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 318–328. ACM Press, 1988.

[42] R. Leupers. Code generation for embedded processors. In *Proceedings of the 13th International Symposium on System Synthesis (ISSS'00)*, page 173. IEEE Computer Society, 2000.

[43] R. Leupers, A. Basu, and P. Marwedel. Optimized array index computation in DSP programs. In *Asia and South Pacific Design Automation Conference*, pages 87–92, 1998.

[44] R. Leupers and P. Marwedel. Algorithms for address assignment in dsp code generation. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 109–112. IEEE Computer Society, 1996.

[45] S. Liao. *Code generation and optimization for embedded digital signal processors.* PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1996.

[46] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction selection using binate covering for code size optimization. In *International Conference on Computer Aided Design*, pages 393–401, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press.

[47] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *SIGPLAN Notices*, 30(6):186–195, June 1995.

[48] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18(3):235–253, May 1996.

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

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

[51] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, San Francisco, 2000.

[52] G. Ottoni, S. Rigo, G. Araujo, S. Rajagopalan, and S. Malik. Optimal live range merge for address register allocation in embedded programs. In *Compiler Construction : 10th International Conference, CC 2001*, volume 2027, page 274. Springer Verlag, 2001.

[53] J. Park and S.-M. Moon. Optimistic register coalescing. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 196–204, Paris, France, October 12–18, 1998. IEEE Computer Society Press.

[54] E. Pelegri-Llopart and S. L. Graham. Optimal code generation for expression trees: an application of BURS theory. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 294–308. ACM Press, 1988.

[55] T. A. Proebsting. Least-cost instruction selection in dags is np-complete. *http://research.microsoft.com/~ toddpro/papers/proof.htm.*

[56] T. A. Proebsting. Simple and efficient burs table generation. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 331–340. ACM Press, 1992.

[57] T. A. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3):461–486, 1995.

[58] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining. In *International Symposium on Microarchitecture (MICRO-27)*, pages 63–74, 1994.

[59] J. Runeson and S.-O. Nyström. Retargetable graph-coloring register allocation for irregular architectures. In Andreas Krall, editor, *Software and Compilers for Embedded Systems*, volume 2826 of *Lecture Notes in Computer Science*, pages 240–254. Springer Verlag, September 2003.

[60] B. Scholz and E. Eckstein. Register allocation for irregular architecture. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02)*, Berlin, June 2002. ACM.

[61] M. D. Smith and G. Holloway. Graph-coloring register allocation for irregular architectures. Technical report, Harvard University, 2000.

[62] A. Sudarsanam, S. Liao, and S. Devadas. Analysis and evaluation of address arithmetic capabilities in custom dsp. *Design and Automation of Embedded Systems*, 4(1):5–22, Jan 1999.

[63] S. R. Vegdahl. Using node merging to enhance graph coloring. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 150–154, Atlanta, Georgia, May 1–4, 1999.

[64] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

LEBENSLAUF

ERIK ECKSTEIN

PERSÖNLICHE DATEN

Geburtsdatum: 18. März 1972
Geburtsort: Lilienfeld
Eltern: Elsa und Kurt Eckstein
Wohnort: Baden bei Wien

AUSBILDUNG

1978 bis 1982	Besuch der Volksschule in Rohrbach an der Gölsen
1982 bis 1986	Besuch des Bundesgymnasiums in Lilienfeld
1986 bis 1991	Besuch der Höheren Technischen Bundeslehr- und Versuchsanstalt St. Pölten
1991 bis 1996	Studium der Technischen Informatik an der Technischen Universität Wien
2000 bis 2003	Doktoratsstudium am Institut für Computersprachen der Technischen Universität Wien

BERUF

1987 und 1988	Ferialpraktika bei der Firma Erich Schmid GmbH.
1989	Ferialpraktikum bei der Firma Schrack AG
1992	Freiberuflicher Mitarbeiter der Firma BETA Datenverarbeitungsges.m.b.H.
1996 bis 1997	Zivildienst beim Roten Kreuz in St. Pölten
1997	Wissenschaftlicher Mitarbeiter am Institut für Computergraphik der Technischen Universität Wien
seit 1997	Beschäftigt bei der Firma Atair Software GmbH. als Projektmanager im Bereich Compiler Entwicklung

PUBLIKATIONEN

E. Eckstein und A. Krall. Minimizing cost of local variables access for DSP-processors. In Y. Annie Liu and Reinhard Wilhelm, Herausgeber, *LCTES'99 Workshop on Languages, Compilers and Tools for Embedded Systems*, Band 34(7), Seiten 20–27, Atlanta, 1999.

B. Scholz und E. Eckstein. Register allocation for irregular architectures. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02)*, Berlin, Juni 2002. ACM.

E. Eckstein und B. Scholz. Address mode selection. In *Proceedings of the International Symposium of Code Generation and Optimization (CGO'03)*, San Francisco, März 2003. IEEE/ACM.

E. Eckstein, O. König, und B. Scholz. Code instruction selection based on SSA-graphs. In *Proceedings of the SCOPES*. IEEE/ACM, September 2003.