Die approbierte Originalversion dieser Dissertation ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

DISSERTATION

Compilation Techniques for Reducing Energy Consumption of Embedded 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.techn. Andreas Krall
Institut für Computersprachen

Arbeitsbereich für Programmiersprachen und Übersetzerbau

eingereicht an der

Technischen Universität Wien

Fakultät für Informatik

von

Dipl.-Ing. Ulrich Hirnschrott
9240416
Halbgasse 26/11
1070 Wien

Wien, im Mai 2005

Kurzfassung

Die Bedeutung von *System-on-Chip* und *System-in-Package* Lösungen im Bereich *eingebetteter Systeme* hat während der letzten Jahren stetig zugenommen. Aufgrund der steigenden Komplexität eingebetteter Anwendungen und der irregulären Prozessorarchitekturen werden hoch optimierende Übersetzer benötigt, um die hohen Anforderungen an Chipgröße und Leistungsaufnahme zu erfüllen.

Der Energieverbrauch aktueller Prozessoren ist hauptsächlich durch die dynamische Leistungsaufnahme bestimmt. Diese kann durch Minimierung von Speicherzugriffen, Minimierung der Ausführungszyklen und Minimierung der Schaltvorgänge an Bussen reduziert werden. In dieser Dissertation werden Erweiterungen der Registerzuweisung für irreguläre Architekturen vorgestellt, welche die Auslagerungsbefehle reduzieren. Weiters wird eine Programmoptimierung präsentiert, welche die dynamischen Schaltvorgänge am Programmspeicherbus minimiert. Die vorgestellten Techniken wurden für die Architektur des digitalen Signalprozessors *xDSPcore* implementiert.

Registerzuweisung bildet die Variablen eines Programms auf die Register eines Prozessors ab oder lagert diese in den Speicher aus. Die gebräuchlichste Methode ist das Färben eines *Konfliktgraphens*, welcher durch Analyse der statischen Lebensdauer von Variablen konstruiert werden kann. Diese Analyse muss die Besonderheiten einer Architektur berücksichtigen. Für die xDSPcore Architektur sind das ein in Bänke geteilter Registersatz, teilbare Register und bedingte Ausführung von Befehlen. Die Beiträge dieser Arbeit sind eine Lebensdaueranalyse unter Berücksichtung bedingter Ausführung und geteilter Register, sowie ein erweitertes Modell des Konfliktgraphens. Beide Techniken sind nötig, um einerseits *teilweise Überschneidungen* von Variablen, welche durch nicht orthogonale Befehle entstehen können, zu erfassen und um andererseits präzise Lebensdauerinformation für bedingte Befehle zu erhalten. Zusätzlich wurde eine auf *partitionierter Boolscher quadratischen Programmierung (PBQP)* beruhende Registerzuweisung implementiert. Dieser Ansatz modelliert Einschränkungen auf Programmvariablen und Auslagerungsentscheidungen durch Kostenfunktionen und Matrizen. Ein PBQP Problemlöser liefert optimale Registerzuweisungen und Auslagerungsentscheidungen. Diese Dissertation trägt die nötigen Kostenfunktionen und Matrizen für xDSPcore bei, welche wiederum geteilte Register und teilweise Überschneidungen modellieren. Die beiden Lösungsansätze werden empirisch verglichen.

Zugriffe auf den Programmspeicher tragen wesentlich zum Energieverbrauch eines Prozessors bei. Dieser kann durch Minimierung der Schaltvorgänge zwischen aufeinander folgenden Speicherzugriffen reduziert werden. Die Schaltvorgänge können durch die Hamming Distanz der binär kodierten Befehlsworte modelliert werden. Der Beitrag dieser Dissertation ist eine Programmoptimierung, welche die Befehlsworte von Funktionen so anordnet, dass eine globale Hamming Distanz der Befehlsworte minimal ist. Dies wird durch Permutation der Befehle innerhalb einzelner Ausführungseinheiten und durch Vertauschen der Operanden bei kommutativen Operationen erreicht. Ein präzises Optimierungsmodell für xDSPcore und die damit verbundenen Algorithmen werden vorgestellt. Im Unterschied zu bekannten Methoden behandeln diese Algorithmen Kanten im Kontollflussgraphen ohne Heuristiken und können auch mit nicht ausgerichteten Ausführungseinheiten umgehen.

If-Umwandlung ist eine bekannte Technik, um Sprunganweisungen aus dem Programm zu eliminieren. Eine Analyse typischer Programme zeigt die Notwendigkeit und Wichtigkeit dieser Transformation. In dieser Dissertation wird eine Implementierung vorgestellt, welche auf eine kleine Teilmenge von Kontrollflussmustern limitiert ist. Weiters werden Abschätzungsfunktionen für maximale Ausführungszeit und Programmgröße vorgestellt, welche während der Transformation verwendet werden, um potenzielle negative Auswirkungen zu vermeiden.

Befehlsanordnung ist eine Technik, um Parallelität auf Befehlsebene auszunützen. Gebräuchliche vorwärts planende Algorithmen leiden an der oftmals zu frühen Einordnung von Befehlen und erzeugen dadurch künstlich Registerdruck. Diese Arbeit analysiert das der Anordnung zugrunde liegende Modell des Datenabhängigkeitsgraphens. Dadurch zeigt sich, dass die Algorithmen adaptiv gestaltet werden müssen, um mit den verschiedenartigen Strukturen der Graphen besser umgehen zu können. Detailverbesserungen an den Anordnungsentscheidungen für Auslagerungsbefehle werden präsentiert.

Ein wichtiger Aspekt auf Ebene des Systementwurfs ist das Untersuchen des architekturellen Gestaltungsraumes. Dadurch wird versucht, den Unterschied in der Effizienz von fest verdrahteten Hardware-Lösungen und programmierbaren Komponenten auszugleichen. In dieser Dissertation wird ein auf Hochsprachen basierender Ansatz namens *DSPxPlore* präsentiert. Dieser erlaubt es die Mikroarchitektur in manchen Parametern so zu skalieren, dass sie den Anforderungen der eingebetteten Anwendung exakt entspricht. Zentraler Punkt von DSPxPlore ist eine Konfigurationsdatei, welche die gesamte Beschreibung der Architektur enthält.

Abstract

The importance of *System–on–Chip* and *System–in–Package* solutions in the domain of *embedded systems* was steadily increasing during the last years. Due to the rising complexity of embedded applications and irregular processor architectures, highly optimizing compilers are needed to meet the stringent chip area and power dissipation requirements of such platforms.

The energy consumption of current processors is dominated by the dynamic power dissipation, which can be reduced largely by minimizing the number of memory accesses, minimizing execution cycles, and minimizing switching activities on buses. This thesis contributes improvements on register allocation for an irregular architecture which reduce memory accesses and execution cycles, and a post–pass code optimization for minimizing the dynamic switching on the instruction memory bus. The techniques presented in this thesis are implemented in the context of the *xDSPcore* architecture which will be introduced shortly.

Register allocation maps the program variables to the registers of a processor or *spills* them to a memory location. The most common technique is coloring an *interference graph* which is constructed through *liveness analysis.* Liveness analysis on code for the xDSPcore has to deal with an irregular and banked register file, and with predicated execution. The contributions of this thesis are a *predicated liveness analysis* on *shared registers* and an *augmented interference graph.* Both methods are necessary for modeling *partial interferences* which arise from non–orthogonal instructions, and for precise liveness information of predicated code. Additionally, a register allocation approach based on *partitioned boolean quadratic programming (PBQP)* is implemented. This approach models architectural constraints and the problem of spilling decisions by cost functions and matrices. Register assignments and spilling decisions are calculated by a solver that delivers optimal results. This thesis contributes the necessary cost functions for xDSPcore that model shared registers and partial interferences. An empirical comparison of the graph–coloring and the PBQP–based approach is given.

Code memory accesses during *instruction fetch* of the processor make a substantial contribution to power dissipation which can be minimized by reducing the switching activities between successive fetches. Switching activities are modeled by the Hamming distance of the binary encoded instruction words. This thesis contributes a post–pass optimization that finds the code arrangement for a function which yields a globally minimized Hamming distance of all instruction words. This is done through *permuting* instructions of the *execution bundles* and through *swapping* the operands of commutative operations. A precise optimization model for xDSPcore and the associated optimization algorithms are introduced. In contrast to existing work, the algorithms consider control flow edges without using heuristics and can operate on non–aligned execution bundles which can even cross fetch word boundaries.

If-conversion is a known technique for reducing the amount of branching instructions in assembly code. The necessity and importance of this transformation is pointed out by analysis of typical algorithms. This thesis presents an implementation which is limited to a small subset of control flow graph patterns and contributes estimation functions for *worst-case execution time* and *code size.* These functions are applied for guiding the transformation in order to avoid negative impacts and interferences with other dependent optimizations.

Instruction scheduling is a technique for exploiting *instruction level parallelism* in code sequences. Top-down list scheduling algorithms suffer from over-scheduling instructions and thus create artificial register pressure. In this thesis, an extensive analysis of the underlying problem model, the *data dependence graph*, is presented. This analysis shows that scheduling heuristics have to be adapted to the particular graph-structures. Refinements on root node selection and on handling spilling instructions are presented.

Design space exploration based on a high-level language is an approach of embedded systems design that attempts to close the efficiency gap between dedicated hardware circuits and software programmable components. This thesis contributes a methodology named *DSPxPlore* which allows to scale important micro-architectural features of xDSPcore in order to adapt the processor to the requirements of the embedded application. A unique configuration file contains the entire architectural description and constitutes the central part of this methodology. It is used for setting up the *toolchain* at run time in order to generate and simulate machine code for the currently specified architecture.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Embedded Digital Signal Processors

The continuous growth of the embedded systems market in recent years and the increasing complexity of embedded software are a big challenge: Heterogeneous hardware and software platforms have to be handled by the system designers, emerging topics like *Ubiquitous Computing* demand for new solutions. In parallel, the energy budget for mobile devices is not growing, whereas upcoming audio, imaging, and video techniques require highest computing power. In contrast to this evolution, the product development cycles are getting shorter and shorter.

Decreasing feature size and rising cost pressure lead to highly integrated solutions where a complete system is realized on one die (System–on–Chip, SoC) or in one package (System–in–Package, SiP). Computing intensive algorithms are partly mapped to hardware accelerators to meet the power dissipation and chip area requirements. *Embedded Digital Signal Processors (DSP)* are used for parts of the application with lower data rates which have to be kept flexible due to run–time adaption requirements and changing standards, whereas micro–controllers and protocol processors are used for control and protocol parts of the application.

The digital signal processors of the application are facing specific requirements and can therefore be considered as *application specific processors* with the following distinctive properties:

- fixed–point arithmetic
- accumulator registers supporting higher precision for internal calculation
- multiply–accumulate instructions for filter structures
- zero–overhead hardware loop instructions
- dual (multi) banked data memory

- complex addressing modes like auto-increment and –decrement, modulo–addressing, bit–reversed addressing, etc.
- explicit instruction level parallelism (VLIW)
- single instruction, multiple data parallelism (SIMD)
- predicated execution

Due to these architectural features, DSPs are considered as compiler-unfriendly, and therefore existing compilers for DSPs do not attain the necessary code quality in terms of performance and especially code size. Therefore most of today's application code for DSPs is still written in assembly code in order to meet the stringent chip area and performance requirements.

With increasing system complexity and the upcoming of more powerful and complex DSP core architectures, writing assembly code manually is not longer feasible. Additionally, assembly code suffers from its lack in portability to other architectures. Therefore the necessity for having highly optimizing compilers is given. Further on, the compilation, linking, and simulation tools (the *toolchain*), have to integrate smoothly into existing system design flows.

1.2 Compilers

Compilers belong to the fundamentals of computing science. They translate high–level language programs into executable machine code. Usually a compiler is divided into two parts: The *frontend* is responsible for reading a program, transforming it to an intermediate representation, and performing hardware independent optimizations. The *backend* transforms the intermediate representation to machine code, doing machine specific optimizations. Figure 1.1 shows a general structure of a compiler for different input languages and different target machines.

Traditional optimizations done in the frontend of the compiler include dead code elimination, constant propagation, function inlining, common sub–expression elimination, strength reduction, induction variable elimination, etc. The major optimizations in the backend part are instruction selection, instruction scheduling and register allocation. This thesis discusses design and implementation of backend optimizations for a digital signal processor architecture named *xDSPcore*. These optimizations are built upon the frontend of the *Open Compiler Environment (OCE)*. The OCE was developed by Atair Software GmbH, Vienna, Austria.

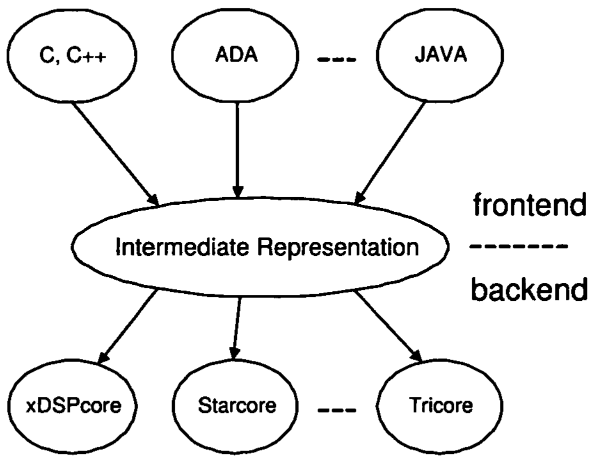


Figure 1.1: General structure of a compiler

1.3 Power & Energy

The traditional focus of compiler optimizations was improving the execution time of programs. For todays' embedded processors, *code size* is of significant importance as well, because it highly impacts silicon area and therefore production costs. Prior research has found many approaches and solutions so that these criteria can be solved to a satisfying extent.

In recent years, power dissipation raised as another dimension in the optimization space. The increased power dissipation causes heavy thermal problems. Special packaging and cooling techniques had to be developed in order to overcome those problems and to keep systems reliable. Current desktop processors seem to have reached the critical point in thermal density. Without cooling measures, they would not even be operable any more. Only very careful system design, mainly seen at laptops and notebooks, allows un–cooled operation. Figure 1.2, originally published by Intel around 2000, shows the trend of power dissipation in recent years and further emphasizes the related problems.

This picture illustrates that new ideas are required to overcome the increased power dissipation of new generation processors. Applications based on embedded systems do not draw such a drastic picture like the standalone processors of the desktop world. Nevertheless, it is reasonable to look at such a picture, as it gives a good indication for the problem.

Organization of this thesis

Chapter 2 introduces the reader to the field of energy consumption of CMOS technology driven computing, and figures out the optimization opportunities. Chapter 3 provides an overview of related work. Chapter 4 describes the target

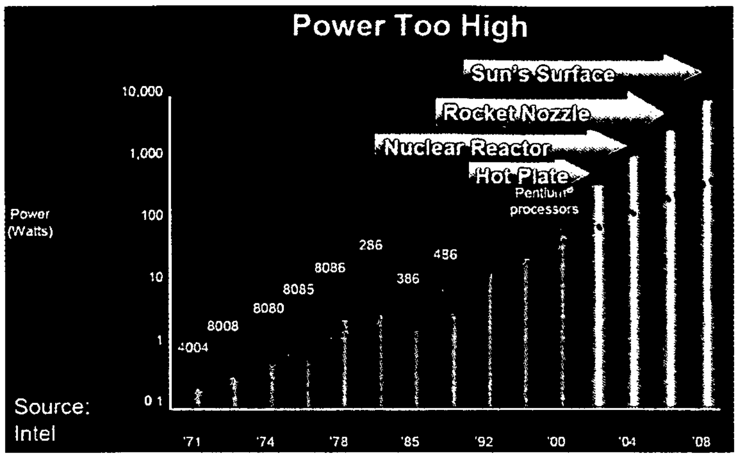


Figure 1.2: Trend of power dissipation

architecture which has been considered within this thesis. The main body of this thesis is covered in Chapter 5, which is divided into two parts: The first part presents extensions to register allocation for irregular architectures and an optimization to reduce switching activities on the instruction memory bus. The second part describes some minor improvements of if conversion and instruction scheduling, and also introduces a design space exploration methodology. Chapter 6 concludes this thesis and gives an outlook on further work to be conducted in the future.

Chapter 2

Towards energy reduction

The rising computational requirements of current embedded software and signal processing algorithms result in raised energy demands. Factors like packaging costs, chip area, voltage and clock supply are also tightly connected to the terms *power dissipation* and *energy consumption.* This asks for new power and energy optimizations to be investigated.

Before developing such optimizations, it is necessary to identify the main contributing factors and to clarify the technical terms. What else has to be stated here is, that only the contribution of the digital signal processor and its attached memories is considered within this thesis.

2.1 Definitions

The terms *power dissipation* and *energy consumption* have already been stated several times. The following paragraphs give definitions for both of them, mostly taken from [7]. However, hardware experts may find some slight inaccuracies, but the definitions are sufficiently accurate for software optimization models.

Equation (2.1) shows the main three sources of power dissipation in current CMOS technology.

$$P_{total} = P_{leakage} + P_{short} + P_{dynamic} \tag{2.1}$$

$P_{leakage}$ is the power which is dissipated through leakage current that arises from substrate injection and sub–threshold effects. It is primarily determined by fabrication technology considerations. P_{short} comes from the direct–path short circuit current, which arises when both NMOS and PMOS transistors are simultaneously active. $P_{dynamic}$ comes from charging and discharging of capacitive loads due to logical changes. From these facts and as stated more precisely in [7], it becomes apparent that leakage and short circuit current can be handled only during hardware design and implementation, whereas dynamic switching power widely depends on the software running on the system. Therefore, the dynamic power has to be considered in more depth.

$$P_{dynamic}(t) = \frac{1}{2}\, CV_{dd}^2\, f_{cl}D(t) \tag{2.2}$$

Equation (2.2) shows the factors which contribute to the dynamic power dissipation. C is the switched capacitance, V_{dd} the supply voltage, f_{cl} the clock frequency and $D(t)$ the density of bit transitions (i.e. switching activity in the current hardware state).

$$E_{dynamic} = \int P_{dynamic}(t)\; dt \tag{2.3}$$

In general, energy is the product of power and time ($E = Pt$) as can be read in physics textbooks. As indicated in Equation (2.2), power dissipation is a function in time. Therefore, energy consumption has to be calculated through integration as stated in Equation (2.3).

At first sight, the optimization objectives can be easily deduced from these equations. Any reduction of either of the factors results in a reduction of power dissipation and thus reduces the amount of consumed energy. However, the factors are not completely independent and only careful analysis and integral optimization techniques will lead to an energy consumption optimized embedded system.

2.2 Optimization targets

The following sub–sections will discuss how each of the contributing factors can be tackled and how those are particularly related to each other.

2.2.1 Physical Capacitance

The switched capacitance is mostly determined through the low–level circuit design and therefore seems to be out of scope for software optimization. Nevertheless, the capacitances of the different hardware building blocks range through a wide spectrum. As an example, external buses connecting memories and ALU generally have considerably higher capacitances than ALU internal blocks. This makes memory accesses costly in terms of energy.

From the statements above, it becomes apparent that reducing memory accesses also leads to a reduction of energy consumption. Traditional compiler optimizations already focus on optimizing *data* memory accesses, but mainly motivated from the perspective of locality in memory hierarchies. Apart from that, *program* memory accesses occur more often than data memory accesses and therefore have to be optimized as well.

Additionally, memory accesses can even be optimized on application level through carefully choosing suitable algorithms (and implementations) amongst the available alternatives. System engineers are expected to tend towards low memory throughput and footprint algorithms, which maybe have higher computational costs, but are "cheaper" in terms of energy overall. Only careful system analysis and profiling can deliver the right answers in those matters.

2.2.2 Supply voltage and clock frequency

The quadratic contribution of supply voltage promises large energy savings by voltage reduction. However, voltage and clock frequency cannot be considered isolated from each other, because reducing one induces a reduction of the other.

On the hardware level, each of the required voltage levels and clock frequencies have to be implemented and supported. This can eventually lead to some overhead which has to be payed. If a multi–voltage supply is already implemented on the embedded system, then running the core with different voltage levels is possible with only little hardware overhead. Otherwise, additional hardware blocks like *on–chip DC–to–DC converters* or *Low Dropout Regulators (LDO)* which deliver the required voltage levels have to be provided. These additional blocks itself cause additional power dissipation and especially on–chip LDOs may lead to thermal problems as well. The energy savings on the core through reducing its supply voltage may thus be nullified. Therefore this efficient method can only be applied successfully under careful application and system analysis.

Another problem is the transition between different voltage and frequency settings. This takes a considerable amount of time, during which the core does not contribute to the system performance. Additionally, voltage and frequency can not be reduced arbitrarily. Hard real–time requirements demand task completion within a strict deadline. Overly aggressive voltage and frequency reduction can lead to a potential violation of such a deadline.

In spite of those drawbacks, voltage reduction/scaling is an important optimization opportunity for low–power systems. It is most promising, when implemented in a real–time operating system on task level, but can also be taken to a finer granularity.

2.2.3 Switching activity

Switching activity is caused by charge and discharge of capacitors during changes between the logical states *1* and *0*. In contrast to technology related power saving modes, switching activity is an issue which can be directly influenced by software development.

Switching activity is caused by charging and discharging of capacitors due to changes in logical state ($0 \rightarrow 1$, $1 \rightarrow 0$). Therefore it is directly influenced by the executed code and thus is a major optimization target. Optimizations have to

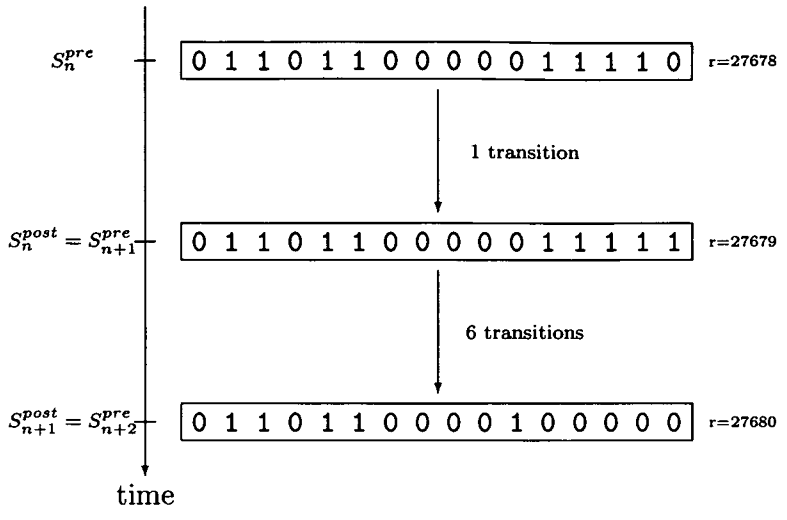


Figure 2.1: Architectural switching on a 16 bit register

focus on architectural features which are visible to the software layer, for example registers and memory buses.

Modeling the architectural switching activities is straight–forward. Each execution cycle n is divided into two states S_n^{pre} and S_n^{post}. For each of the architectural features, internal values in S_n^{pre} and S_n^{post} are represented through bit vectors $\vec{B_{pre}}$ and $\vec{B_{post}}$ ($\vec{B} = b_0 b_1 \ldots\ b_n | b_i \in \{0,1\}$). Calculating the Hamming distance $D(\vec{a}, \vec{b}) = \#\{i | a_i \neq b_i\}$ (that means the number of differing bits in $\vec{a}$ and $\vec{b}$) of B_n^{pre} and B_n^{post} gives the number of bit transitions within this cycle. S_{n+1}^{pre} of the next cycle is then assigned to S_n^{post} of the current one. Figure 2.1 depicts a 16 bit register to illustrate architectural switching activity.

The Hamming distance is no exact metric for the *absolute* amount of energy that is consumed in an execution cycle. But it delivers an adequate and consistent model for creating objective functions of energy optimizations. Whenever the Hamming distance of two consecutive states is reduced, less energy will be consumed.

2.2.4 Time

Reducing *task completion time* leads to a reduction of energy consumption. The task completion time depends mostly on the number of execution cycles (computations) a task needs for completion and on the clock frequency of the system. Thus, a smaller cycle count and a higher clock frequency leads to a shorter task completion time. Admittedly, this is a rather simple but indeed practical model.

As can be seen from Equation (2.2) and Section 2.2.2, increasing clock frequency is counterproductive in terms of power and therefore compensates the savings from reduced task completion time. Nevertheless, in certain circumstances, where a processor has several predefined power states, it may be beneficial to run a task at maximum performance and then switch the processor to a low–power profile (or turn it off completely). Such techniques are similar to those mentioned in Section 2.2.2 and will be discussed in Section 3.2.2.

As a rule of thumb, it can be claimed that a run time optimized program is also energy optimized (with few exceptions [8]). Minimizing the run time of a task is a traditional domain of compiler optimizations. Lots of source code analysis and target code synthesis techniques have been explored and are still under investigation. Most of these techniques focus on minimizing the average case execution time (ACET). While ACET of DSP algorithms is still an important quality criterion on code optimizations, worst–case execution time (WCET) and its analysis becomes an additional and crucial factor in hard real time systems. Therefore compiler optimizations in the real time domain have to obey additional objectives:

- WCET minimization
- code predictability

The first one directly affects real–time task scheduling. A reduced WCET results in larger schedule slacks and thus gives more freedom in voltage scaling. The second objective mainly targets at the quality of WCET analysis results. If the execution behavior of machine code is predictable, then the estimates of WCET have closer confidence margins. Section 5.2.1 and [9] cover these aspects in more detail.

The previous pages have given a short survey on the different topics related to energy consumption. The most important optimization targets have been identified and shortly classified regarding at which level the optimizations have to take place: *Physical capacitance* is mostly determined through hardware design, but can be regarded from the software layer in certain aspects. Building blocks with high capacitances (buses, memory) have a larger potential for energy reduction than those with lower capacitance (datapath). *Supply voltage* and *clock frequency* are overall system related topics and can be optimized on operating system level through *voltage scaling*. *Switching activity* depends mostly on the instructions and instruction sequences that are executed on the processor. It can be minimized by reordering of instructions and a careful choice of registers and addresses at compile–time. *Execution time* depends on clock frequency and the number of execution cycles and thus has to be considered from both real–time and code optimization aspects.

This diversity of topics gives the opportunity to deal with power and energy aspects on different software layers. Depending on system domain and context, appropriate models for optimization guidance have to be deduced. As the title of this thesis indicates, the main techniques which have been investigated in this thesis are in the context of an embedded DSP core, the architecture of which will be briefly summarized in Chapter 4. Section 5.1 focuses on compilation techniques that reduce execution time, data memory accesses, and switching activities on the instruction memory bus. A considerable amount of work on compiler supported application analysis and design space exploration has been carried out and is described in Section 5.2.3.

Due to this narrow domain, optimization models can be kept relatively simple. Missing details from system and implementation level make it unnecessary to include detailed power and energy estimates into the compilation toolchain. Key figures like *code size*, *execution cycles*, *bit transitions on memory buses*, number of *memory accesses* (both data and code) are sufficient and serve as main optimization objectives. Nevertheless, this does not relieve the tools from 'knowing' how each of these factors impact energy consumption.

Chapter 3

Related Work

This chapter is divided into two parts. The first part covers traditional compiler optimizations which served as starting points for this thesis' contributions. The second part discusses work which has been carried out in the closer context of power and energy topics.

3.1 Traditional compiler optimizations

Compiler research focused largely on techniques for optimizing the execution time of programs. The next sub–sections will introduce some existing techniques which are relevant for this thesis.

3.1.1 Register allocation

Register allocation is of vital importance in optimizing compilers. In general, compilers use an internal program representation which assumes an unlimited number of available registers. Register allocation maps these so–called *symbolic registers* (or live–ranges) to physically available hardware–registers of the target architecture. Depending on its scope, a classification into local, global, and inter–procedural register allocation is done. This work focuses on global register allocation. Details on local and inter–procedural register allocation can be found in [10] and [11, 12, 13], respectively.

The most common technique for solving register allocation is based on *graph–coloring* and was first introduced by Chaitin [14]. Based on global liveness analysis, an *interference graph* is built. For each symbolic register, a node is inserted in the interference graph. Whenever two symbolic registers are live at the same time, an edge between their representing nodes is inserted. If this graph can be colored with k colors (k is the number of architectural registers), this coloring model is a valid register assignment for the symbolic registers. If no such coloring can be found, some of the symbolic registers have to be spilled to data memory.

Spilling means, that after each definition point of the symbolic register, its value has to be stored to memory, and before every use of the symbolic register, the value has to be re–loaded from the spill location again.

Coloring the graph can be done through graph simplification. Nodes with a degree less than k (these are trivial to color) are removed from the graph and put onto a stack repeatedly. When the empty graph remains, the nodes are reinserted into the graph in stack order and assigned an available color. When the simplification leaves a non–empty graph, spilling candidates have to be chosen heuristically and the corresponding spill code has to be inserted. Then the whole process starts again with building the interference graph, which now is less constrained and therefore more likely to be colorable.

Briggs [15] improves this coloring scheme by delaying the spill decision to the reinsert phase. This benefits from the fact, that nodes of significant degree can be colored when two adjacent nodes have been assigned the same color. This technique is called *optimistic coloring.* Further refinements of Briggs are presented in [16, 17]. A technique called *rematerialization* is introduced. Briggs points out that for certain symbolic registers, the value often can be easily re–computed. Therefore it is beneficial not to spill those values, but rather insert appropriate instructions for re–computation if necessary. Other improvements introduced consider closeness of use–points when inserting spill code. Whenever two use–points of a live–range are close, this live–range is kept in register instead of being spilled and re–loaded.

A sub–problem of register allocation is *coalescing* of copy–related symbolic registers (source and destination operands of copy instructions). Whenever these registers are assigned the same color, the copy instruction is redundant and can be eliminated. Chaitin's allocator performs *aggressive coalescing*, where every non–interfering pair of copy–related symbolic registers is coalesced to a single live–range. This may impair colorability. Therefore Briggs suggested *conservative coalescing*, where two registers only are coalesced, if it is guaranteed that the new live–range is not spilled. George [18] suggests a method called *iterated coalescing*, where coalescing and graph simplification is done iteratively. A superior approach called *optimistic coalescing* is presented by Park in [19]. Positive and negative impact of coalescing are elaborated in detail by showing examples for improvement or degradation of colorability. It is shown that the conservative approaches of Briggs and George do not benefit from the positive impact. The optimistic approach therefore performs aggressive coalescing on an optimistic register allocator. When a live range xy becomes an actual spill, it is split into two live ranges x and y again, recovering the original edges of the interference graph. It might now be possible to color x or y or both, because the degree of the nodes in the graph is likely to be reduced.

A considerably different approach to register allocation is presented by Chow [20] and is *called priority–based coloring.* Different to a Chaitin–style allocator, this approach assumes live–ranges to reside in memory at first. The register allocator calculates which live–ranges are most profitable when kept in a register. The order in which live–ranges are assigned registers is determined by a priority function that is based on the control flow graph and a cost model for memory loads and stores. This approach does not need an iterative step and therefore is faster for heavily constrained interference graphs. Several internal assumption of this approach (like reserving registers for temporal values introduced by the code generator) make this approach problematic for embedded processors, where register resources are scarce. The remainder of this work will focus on graph–coloring techniques, which are superior in this sense.

The graph–coloring approach is best–fitted for RISC machines with a regular general–purpose register file. For irregular architectures as often seen for embedded processors, some modification are necessary for making graph–coloring work. Briggs [21] and Smith [22] introduce similar techniques for coloring register pairs. Register pairs allow to combine two single precision registers two one of double precision. These two registers must be adjacent and aligned. Briggs solves this by introducing multi–edges in the interference graph. Smith puts weights on the nodes. Both approaches model the additional constraints through the colorability property of the nodes in the graph. Smith recently published some more extensions to his work in [23].

Other work which focuses on irregular architectures is done by Runeson [24]. A more complex colorability test, called $< p, q >$–test is introduced. This approach is superior to those of Briggs and Smith as it models a larger range of irregularities.

All these approaches for handling irregularities suffer from the fact that it is necessary to change the fundamental model of the interference graph and its colorability test.

An approach for *optimal* register allocation is proposed by Goodwin in [25] and is based on 0–1 integer programming. An analysis module analyzes the control flow graph of a function, where register allocation decisions have to be made. These points are assigned a *register allocation action* and decision variable. Actions can be such like *define register assignment, keep register assignment, load from memory, store to memory.* A '1' in the decision variable indicates that this action is taken, '0' means not taken. The decision variables and associated spill–code overhead for each of them are used to formulate a 0–1 integer program, which is then solved using a standard solver. Kong [26] presents an extension to this approach which focuses on irregular architectures. Several classes of irregularities are identified and proper cost models for handling them are introduced.

3.1.2 If–Conversion & Predicated Execution

Predicated execution is a technique that helps on transforming control dependencies into data dependencies through *if conversion.* In predicated execution, the execution of instructions depends on a logical expression called *guard.* The instruction is only executed if the run time value of the guard is *true*. The original motivation for this transformation is handling (conditional) branches in loop vectorization. Branches in general make the execution of instructions d at the branch target dependent on the execution of the branch b itself. Therefore, instruction d are called *control dependent* on b. These control dependencies impose limitations on loop vectorization and scheduling. For each of those control dependencies, a logical expression (from branch conditions) is derived and assigned as a guard to the dependent instruction. Therefore, the control dependence is transformed to a data dependence and can be handled more easily during loop vectorization and instruction scheduling.

The fundamentals of predicated code and if–conversion were introduced by Allen [27]. Park [28] presented a refined transformation with optimal predicate calculation and assignment. Mahlke [29] points out several short–comings of unconstrained if–conversion on unbalanced control flow regions and with speculative execution. A structure called *hyperblock*, which consists of a set of predicated blocks and has a single entry but multiple exits, is introduced to overcome these problems. August [30] points out further inherent problems of if–conversion and presents a refined version of hyperblock selection. It is based on *partial reverse if–conversion* which is used to keep the balance between control–flow and predication.

Work presented in [31, 32, 33] focus on the impact of predicated code onto other code analysis techniques like register allocation. Several modifications and enhancements of existing algorithms are shown.

3.1.3 Instruction scheduling

Scheduling is a technique that reorders instructions so that maximum instruction level parallelism is exploited and thus execution time is minimized. It is a crucial optimization for pipelined architectures, where data dependencies or resource constraints between instructions can cause hazards which would lead to pipeline stalls and performance degradation. For super–scalar architectures, an order of instructions has to be determined which reduces such stalls. Parallelization of the instructions will be done implicitly by the hardware. Scheduling for VLIW architectures needs a more complex approach because it has to be specified explicitly which instructions are executed in parallel. Nevertheless, both strategies have to model architectural resources which may cause hardware contention. This is mostly done by *reservation tables* or *resource vectors.*

Scheduling can be done on local scope (basic blocks) or global scope (whole function). Global techniques offer a better exploitation of instruction level parallelism but often suffer from the drawback of enlarged code size due to code duplication or compensation code. This is barely acceptable for embedded systems, where code size of the program significantly influences chip area. The remainder of this work will focus on local scheduling, more details on global scheduling can be found in [34, 35, 36, 37, 38].

In *list scheduling* as presented in [39], a directed acyclic graph for the instructions of a basic block is constructed. This graph expresses the scheduling constraints between the instructions. A topological sort of this graph then yields the schedule. This sort is determined through several heuristics, for example root nodes with the longest path to the leaves have to be scheduled first. Warren [40] presents refinements on selecting the next root node to be issued and on minimizing liveness.

A considerably different approach is *software pipelining.* In software pipelined loops, the execution of the next loop iteration is started before the prior one has completed. This yields a better utilization of functional units during execution of loops and thus improves performance. Lam [41] showed a basic approach how to implement software pipelining in compiler optimization by making use of an iterative approach for scheduling cyclic graphs. A technique called *hierarchical reduction* is introduced, which allows software pipelining in presence of conditional statements.

Rau [42] presented another algorithm for software pipelining which is called *iterative modulo scheduling (IMS).* This algorithm searches for a schedule where a new loop iteration is initiated after a pre–defined number of cycles, called initiation interval. IMS begins with a *minimum initiation interval*, which is determined by resource– or recurrence–constraints and tries to build a schedule. If no solution can be found, the initiation interval is increased and the scheduling process is started again.

Bala [43] presents an efficient method to perform complex resource contention checks during scheduling, based on finite state automatons. It is more time– and space efficient than prior techniques which are based on reservation tables.

Liu [44] presents a near–optimal approach to instruction scheduling. The method is based on totally enumerating possible schedules with pruning the search space if it can be proved, that no feasible schedule can be found in the current enumeration candidate.

3.2 Power– and Energy–Related Aspects

In recent years, more and more scientific work has been carried out in the field of low–power and power–aware software techniques. While there have been some interesting and fundamental insights, the field is still rather young and sometimes controversial. A profound knowledge of the underlying hardware technology and a precise context of the presented techniques is necessary to fully understand and evaluate the consequences of the presented work. This section only presents a selection of some of the relevant topics in order to show the diversity and complexity of matters. It does not cover the particular work in whole depth.

3.2.1 Compiler techniques

This sub–section gives an overview of work which is conducted in the closer context of compiler techniques.

General

Lorenz [45] presents a phase–coupled code generation technique for DSP processors. The inherent interdependencies of code selection, instruction scheduling and register allocation are pointed out and a genetic algorithm for solving these problems concurrently is presented. The fitness function of the genetic algorithm includes both execution time and energy estimations from a sufficiently accurate model. Evaluation justifies the claim of reducing energy consumption. In [46], Lorenz refines his work by including loop vectorization and exploiting zero–overhead hardware loops. A more extensive experimental evaluation is given.

Marwedel [47] provides a general overview and motivation on embedded systems compilation techniques. Some low–power optimizations are pointed out and verified through measurements on an ATMEL evaluation board. Some compilation techniques for exploiting DSP characteristics are presented and key metrics for code quality identified.

Wehmeyer [48] presents an energy–aware compilation framework for an ARM processor which uses power–models gained from empirical measurements on an evaluation board. The size of the register file is then considered as a parameter for compilation and the effects of spilling and register usage are evaluated regarding performance and energy consumption. The work shows that a good trade–off between register file size and spilling has to be taken early at design time.

Register allocation

Chang [49] describes a technique for calculating the switching activity of set of registers shared by different data values. Given a probability density function of the input values, register assignment is done by minimum cost clique covering of an compatibility graph. The evaluation proved this algorithm to effectively reduce switching activity on the register file.

Gebotys [50] focuses on simultaneous memory and register allocation. The objective is minimization of energy dissipation in onchip and offchip storage components. Modeling as network flow and solving the minimum cost problem, an (energy–)optimal assignment of data variables to registers or memory per basic block is found.

Work from Zhang [51, 52] targets on the same problem as [49, 50], but extends it to global scope. The basic blocks of the control flow graph of a function are topologically ordered with a special handling of branch, merge and loop structures within the network flow problem. The experimental evaluation, based on static and activity–based energy models, showed that energy consumption can be reduced compared to a traditional style graph–coloring register allocator.

Scheduling

In [53], Lee exploits special features of a Fujitsu DSP processor in order to reduce energy consumption. A minimal VLIW scheduling called instruction packing and operand swapping of a Booth multiplier are implemented and evaluated by physical measurements. Introducing these optimizations, an energy reduction of up to 56% was achieved (dependent on the chosen multiplier structure and the input data).

Tiwari [54] provides an extensive overview on instruction level power modeling and points out possible software energy optimization techniques. An experimental evaluation based on physical measurement for three different processors is given.

Toburen [55] presents a modified list–scheduling algorithm which uses a machine description model for power estimation. The list scheduler fills cycles with instructions up to a certain energy threshold. When the threshold is reached, scheduling proceeds with the next cycle. This technique reduces current spikes during program execution and virtually has no overhead due to schedule slacks in the code.

Lee [56] investigates techniques to reduce the power consumption on the instruction bus of VLIW architectures. A horizontal scheduling pass reorders the instructions within one VLIW. A vertical pass moves instructions throughout different cycles. The overall goal is minimization of the bit transitions on the instruction bus. Experiments with 4–way and 8–way VLIW architectures showed reduction of switching activity by 13.3% and 20.15% on average for horizontal scheduling, and an extra improvement by 7–10% when also vertical scheduling is applied.

In [57], Parikh presents several list scheduling algorithms with different node selection strategies. While the classic approach is purely performance oriented, an energy oriented top–down, an energy oriented bottom–up, and an energy oriented with look–ahead strategies are introduced for reducing energy consumption. Further several unified strategies are investigated. The experimental evaluation only uses a very simple architecture and a modest energy–cost model, applied on randomly generated DAGs.

Shin [58, 59] presents an optimization for reducing power consumption in VLIW instruction fetch. The problem is divided into a local (per basic block) and global (per function) step. Both are modeled as shortest–path problem in a graph. For solving the global step, some transformations and heuristics are necessary in order to use the same algorithm. Experiments with a TMS320C6201 from Texas Instruments show a reduction of the switching activity during instruction fetch by 34% through local optimization only. Global optimization gave additional

2.9% reduction.

Choi [60] introduces an enhanced scheduling algorithm which is based on a modified control flow and data dependence graph. Energy–costs from inter–instruction effects are modeled by additionally inserting edges between independent nodes. This yields the so–called weighted strongly connected graph, for which a precedence constrained Hamiltonian path has to be found. Minimum spanning tree and simulated annealing techniques are applied for solving this problem. Evaluation is done using the SimpleScalar toolset. Power savings between 3 and 30% have been observed.

Yun [61] shows a modified iterative modulo scheduling approach, which both optimizes step–power and peak–power of VLIW architectures. This is reached by generating a balanced parallel schedule without sacrificing the performance. Experiments on a SPARC–based VLIW testbed showed, that the balancing reduces the number of cycles which dissipate more than 70% of the maximum power of an 8–way machine from 58% to 8%. The standard deviation of power distribution is reduced from 0.31 to 0.08.

3.2.2 Techniques on Operating System and System Design Level

The following paragraphs emphasize the importance of system considerations in the low–power domain and present topics on various distinct levels.

In [62], Benini provides a brief overview on system techniques to be applied for power optimization. Each technique is accompanied by a concise explanation and an extensive list of pointers.

In [63, 64], the authors present a technique to optimize switching activity on a multiplexed DRAM address bus, called Pyramid code. An extensive overview on several bus coding techniques and on DRAM technology is given. The switching problem is formulated as finding a Eulerian path in a so called RC–graph. The implementation shows a reduction of the switching activity by 50% in case of sequential accesses.

Peymandoust [65] presents a low–power technique that works on specification level by using algebraic transformations. Algorithmic parts with high computational costs, for example calculating a cosine, are replaced by simpler and sufficiently accurate calculations. The presented tool *SymSoft* was used to optimize an MP3 decoder. Besides an increased performance it was possible to decrease energy consumption.

As already stated in Chapter 2, the quadratic contribution of supply voltage to power dissipation offers a great potential for reducing energy consumption by *voltage scaling*. This term subsumes optimization techniques which focus on assigning different supply voltage levels to tasks and sub–tasks in order to minimize energy consumption. Voltage assignments have to be carried out in such a way, that real–time constraints of the tasks are still fulfilled when running at reduced voltage levels (and thus at a lower clock frequency).

Ishihara [66] and Okuma [67] present static voltage scaling techniques for dynamically variable voltage processors. Given a set of real–time parameters of a task and the available clock and voltage specifications, an *integer linear programming (ILP)* formulation of the scheduling problem gives ideal static voltage assignments in order to minimize energy consumption.

Hsu [68] presents a compiler–directed voltage scaling technique which takes into account the speed gap between processor and memory. Regions with potential energy savings are identified and the appropriate slow–down instructions inserted. Experiments with SimpleScalar and the SPECfp95 showed energy savings of 4–23% at a performance penalty of 1–2%.

Shin [69] points put some weaknesses in inter–task voltage scaling techniques and present an intra–task based solution. Based on worst–case execution time analysis, the variations in task execution time due to different control paths are gathered. Each basic block of the task is then assigned a proper speed so that timing constraints are fulfilled and energy consumption is minimized. Remaining task slack time is also exploited and used for further voltage reduction.

Saputra [70] shows how to apply compiler loop optimizations to static voltage scaling. Whenever the number of cycles needed for execution of a program is reduced, the required frequency can be reduced and therefore also the supply voltage (with quadratic relation to power dissipation). The absolute execution time keeps the same. Additionally, an approach of dynamic voltage scaling within the execution of one task is presented. It makes use of integer linear programming and assigns voltage levels to the loop nests encountered in the code.

3.2.3 Profiling and Simulation

Low–power design also needs substantial support on simulation level. Different approaches have already been presented, where the range of accuracy spans from architectural level to finer granularity.

Brooks [71] presents a tool for power analysis on the architectural level. It allows sufficiently fast power estimation early in the design process with an accuracy nearly similar to lower–level tools.

Ye [72] introduces an add–on to the SimpleScalar framework for doing cycle–accurate energy estimation and bases on transitive–sensitive energy models. Energy estimates for the processor datapath, memory, and on–chip buses are delivered.

Some other work in [73, 74, 75] also focus on energy estimation and profiling but will not be further discussed.

Chapter 4

Target architecture xDSPcore

This thesis has been carried out in the domain of programmable embedded digital signal processors. Generally, the presented techniques can be applied to virtually any of current DSP architectures. In order to approach the topics on a conceptual level, a prototype being currently developed at Infineon Technologies Austria has been chosen as target architecture. This chapter will briefly introduce the xDSPcore architecture, a more elaborate presentation can be found in [76].

4.1 Design objectives and philosophy

Designing an architecture from scratch is of course a longsome, complicated but indeed interesting matter. It was mainly motivated by the deficiencies of currently available solutions and often driven by the enthusiasm and curiosity of the team members only.

One of the key ideas in xDSPcore is *scalability*. The demands on an embedded digital signal processor come from different directions. As already pointed out, there are stringent requirements from the perspective of hardware, which demand a great flexibility in implementation. From the perspective of programmers, the processor has to be programmable in high–level languages, which means that its architecture should be compiler–friendly. And last but not least, algorithms may demand special functionality to be implemented efficiently. All these factors have to be considered already during the design phase of an architecture. This philosophy is briefly depicted by Figure 4.1, a broader discussion can again be found in [76]. xDSPcore is therefore not *one* architecture. It is a *family* of architectures with a similar top–level design, but different in certain scalability factors. These factors will be discussed later in Section 5.2.3. The next section introduces the top–level design of xDSPcore based on one architectural instance which served as a reference during carrying out this thesis.

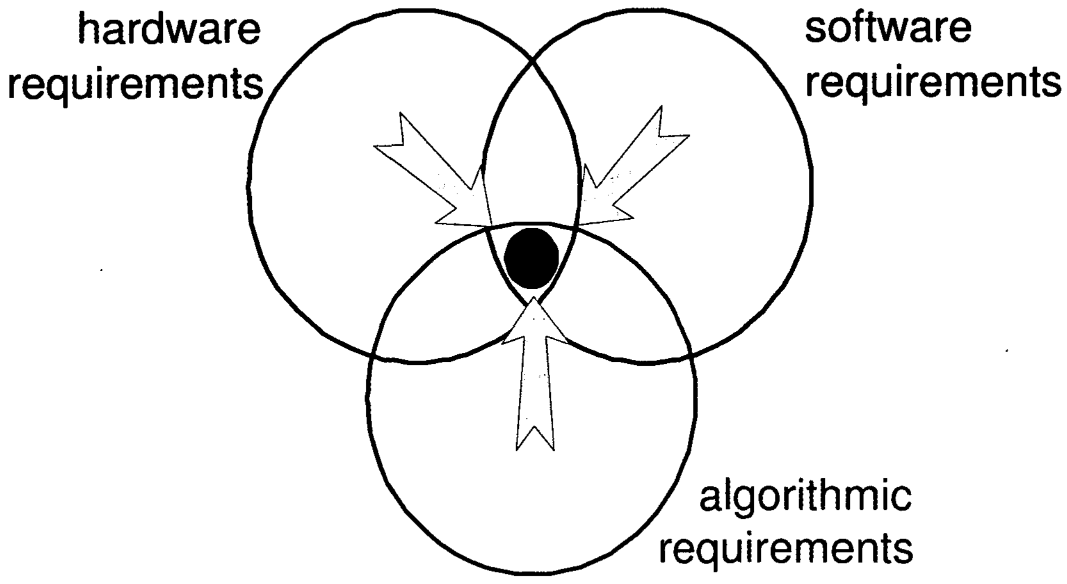


Figure 4.1: Design philosophy of xDSPcore

4.2 Example architecture

The xDSPcore is based on a modified, dual Harvard load-store architecture [77, 78]. Figure 4.2 shows a top–level overview of the architecture.

xDSPcore uses a variable–length VLIW programming model. In general, VLIW suffers from poor code density. This problem is solved by introducing a code compression technique called xLIW [79]. xLIW is based on a *variable length execution set (VLES)*, which enables a decoupling of fetch and execution bundles. Compared to VLES, xLIW permits a reduction of the size of the program memory port (and therefore reduces the wiring effort) without limiting the architecture's peak performance. The program memory port size is 4 instruction words whereas an xLIW instruction can use up to 10 instruction words. An instruction buffer overcomes the possible bandwidth mismatch resulting from the reduced size of the program memory port [80]. The align unit, closely related to the instruction buffer, is responsible for re–building the execution sets from the fetch bundles.

Two independent data buses connect the xDSPcore data path to the data memory. The data memory is split into two banks, x– and y–memory. Two concurrent memory accesses in one execution cycle can be performed only if these do not refer to the same bank. Otherwise, a memory stalling occurs and the accesses are serialized by the hardware. This happens transparently to the software.

For load-store architectures, the register file is a central part of the data path.

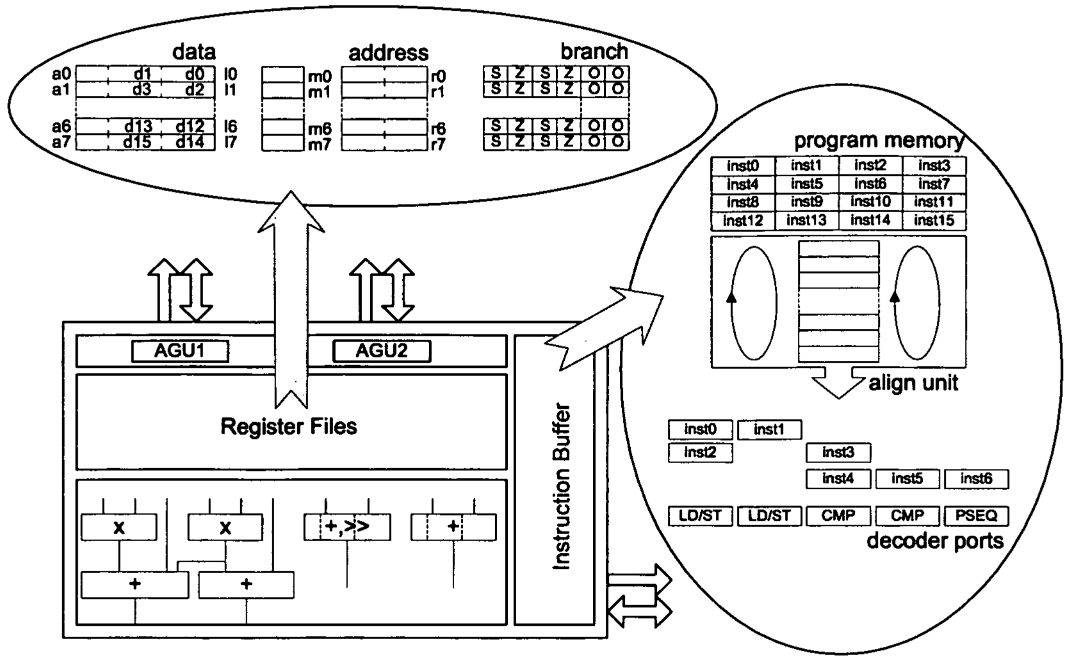


Figure 4.2: Architecture Overview

Separate instructions are used for moving data between a register file and data memory (MOV instruction class); all arithmetic and control–flow instructions (CMP and BR instruction classes respectively) only use register operands from the register file.

The register file of xDSPcore is split into three parts: data register file, address register file including modifier registers and a separate branch file (which is not fully visible to the instruction set). The structure of the register file is orthogonal. There are no restrictions on the usage of registers for special instructions.

The data register file supports three types of register sizes. 40 bit accumulator registers (Ax) constitute the base registers. The 32 least significant bit of an accumulator can be accessed as a long register (Lx). A long register can be split and accessed as two 16 bit short registers (Dx).

Address registers (Rx) are 20 bit wide. Each one has a 12 bit modifier register (Mx) attached, which is used for modulo– and bit-reversed addressing modes.

The branch file contains status information of the core architecture. It holds static information about the register content (e.g. sign and zero flag for each register of the data register file) and dynamic information updated by the data flow (e.g. overflow flags or flags indicating loop status). The separate branch file is implemented in order to relax the number of read/write ports associated with the register files, which are already stressed by the orthogonality requirements [81].

xDSPcore offers a fully predicated instruction set [82]. Predication is register based (not on special flags) and offers much flexibility in building complex conditional expressions. ('true', 'false', and 'dont care'; conjunction, disjunction, loop status, etc ...). More details on predicates will be given in Section 5.2.1.

The RISC–like pipeline of xDSPcore is split into three phases: *fetch*, *decode* and *execute*. Each of the phases can be split into several clock cycles, which results in higher clock frequencies. The reference architecture has a five stage pipeline:

1. IF ... instruction fetch
2. IA ... instruction align
3. ID ... instruction decode
4. EX1 ... execute stage 1 (for simple ALU instructions)
5. EX2 ... execute stage 2 (for memory accesses and multiplications instructions)

However, splitting the instruction fetch into several clock cycles increases the number of branch delays. Spending several clock cycles on the execution phase increases load–use and define–use dependencies. Compensation methods for the arising drawbacks (bypasses, branch prediction) are available [83, 84, 85, 86, 87] but increase core complexity and silicon area.

Accesses to the register file occur in the pipeline stages where values are actually used or computed. In this way, register pressure is reduced and software pipelining can be implemented without the need for rotating register files and modulo variable renaming. Figure 4.3 shows examples for the timings of register file accesses of *MOV* and *CMP* instructions. *read rx* denotes the pipeline edge where *MOV* instruction read the memory address from the address register file. *update rx* denotes when the updated address (in case of auto–modifying *MOV* instructions) is written back to the address register. *read dx* shows the pipeline edge, where the data to be stored by a *MOV* instruction is read from the register file, whereas *write dx* shows the edge where a loaded value is written to the register file. Finally, *write mem* is the edge where a stored value is written into the memory. The upper *read op* of *CMP* instruction denotes the pipeline edge where simple ALU operations and multiplications read the source operands. Its corresponding *write dest* is the edge where the result of simple ALU operations is written to the register file. The lower *read op* denotes the edge where a *multiply–accumulate* instruction reads the accumulator. Finally, the lower *write dest* is the edge where all multiplication instructions write the result to the register file.

The size of the native instruction word of the reference implementation is 20 bit. An instruction word contains three bits used to indicate operation class and containing alignment information. The remaining bits are used for instruction

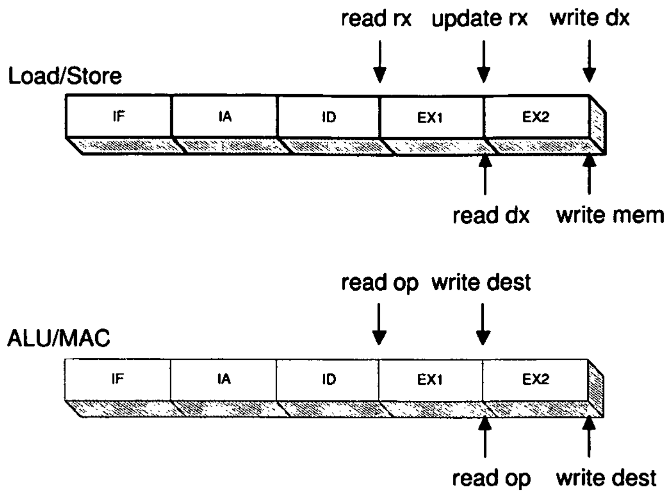


Figure 4.3: Examples for register file accesses

coding. An additional parallel word is available for coding long immediate values or offsets, thus resulting in a 40 bit instruction word (in short this coding is called *20/40–bit coding*). Other codings might be *16/32*, or *12/24/36*.

Chapter 5

Investigated techniques

This chapter presents the contributions of this thesis. The presented techniques have been implemented in an optimizing compiler backend for the xDSPcore architecture. Figure 5.1 gives a brief overview of the backend tasks and the chosen order of optimization. The individual passes and the motivation for this particular phase–ordering are explained in further detail in the rest of this chapter.

Assembly code notation

The assembly code examples in the next sections will follow the rules given subsequently:

An instruction is built from a *mnemonic* and a list of operands. Source operands appear first, the destination operand is at the last position. For example `add D0,D1,D2` adds the registers D0 and D1 and writes the result to register D2.

Each instruction can be assigned a predicate, which is placed before the mnemonic and surrounded by parentheses. The instruction is then executed conditionally, depending on the run time value of the predicate (for example `(D0>0) add D1,D2,D3` executes an addition of D1 and D2 to D3 only if the value of D0 is greater than 0).

An *execution bundle* is made up from multiple instructions written on one line and separated by a double–pipe (||) symbol. All these instructions are executed in the same execution cycle. The bundle

```
add D0,D1,D2 || sub D0,D1,D3
```

performs an addition and a subtraction of D0 and D1. The results are written to D2 and D3 respectively.

Bundles which contain predicated instructions have to be flagged by a corresponding *pe**[1] instruction, depending on the set of predicates. The bundle

```
(D0>0) add D1,D2,D3 || (D0<=0) sub D1,D2,D3 || pe
```

is an example for a simple predicated bundle, where either an addition or a subtraction of D1 and D2 is performed (depending on the value of D0).

Branch instructions in the examples will have two delay slots. These delay slots have to be made explicit by either inserting instructions which can be executed before the branch takes place or by appending no-ops.

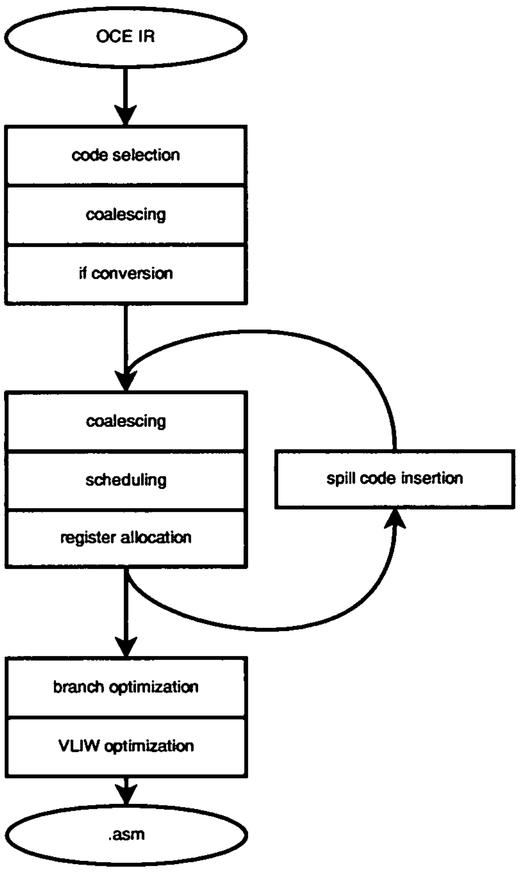


Figure 5.1: Backend tasks and ordering

[1]pe,pel,peal,pexl: these offer different possibilities for predication, including conjunction and disjunction of up to three simple predicates

5.1 Major contributions

This section presents two optimizations: (1) register allocation techniques for irregular architectures, which are necessary to reduce the data memory accesses due to spilling, and (2) a post–pass code optimization method which reduces the switching activities on the instruction memory bus.

5.1.1 Register allocation

Section 3.1.1 already gave a short introduction to the field of global register allocation, covering the graph–coloring approach and its descendants. In this section, the topics relevant to xDSPcore will be presented in more detail. This includes basics like liveness analysis for predicated code or modeling architectural constraints like shared registers. Additionally, an optimal register allocator based on a PBQP approach is implemented. Figure 5.2 gives a detailed overview of the sub–tasks for a graph–coloring register allocator. The PBQP–based register allocator looks quite similar and is depicted in Figure 5.3.

Peculiarities

A register allocator for the xDSPcore architecture has to consider some architectural peculiarities which are only partly covered in related work. Disregarding these peculiarities leads to an overly constrained interference graph. This yields conservative colorings and thus results in inefficient spilling code. Both an overhead in code size through the additional spilling instructions (static overhead) and an increase in data memory accesses (dynamic overhead) will occur.

Predicated liveness and interference The most fundamental difference to traditional register allocation techniques arises from predicated code. Liveness analysis as commonly formulated does not deliver correct results, and construction of the interference graph may lead to overly constrained nodes. An extended formulation of these algorithms is needed in order to achieve a proper modeling of the constraints on symbolic registers.

Liveness analysis is a backward data–flow problem. It is solved by iteratively applying data flow equations (5.1) for all instructions i until reaching a fixed–point in the liveness sets (see [88]).

$$\begin{aligned} IN_i &= USE_i \cup (OUT_i \setminus DEF_i) \\ OUT_i &= \bigcup_{s \in Succ_i} IN_i \end{aligned} \tag{5.1}$$

USE_i is the set of variables used by instruction i, DEF_i the set of variables defined by i, IN_i the set of live–in variables at instruction i, and finally OUT_i the live–out set at i. $Succ_i$ is the set of all direct successors of instruction i in the

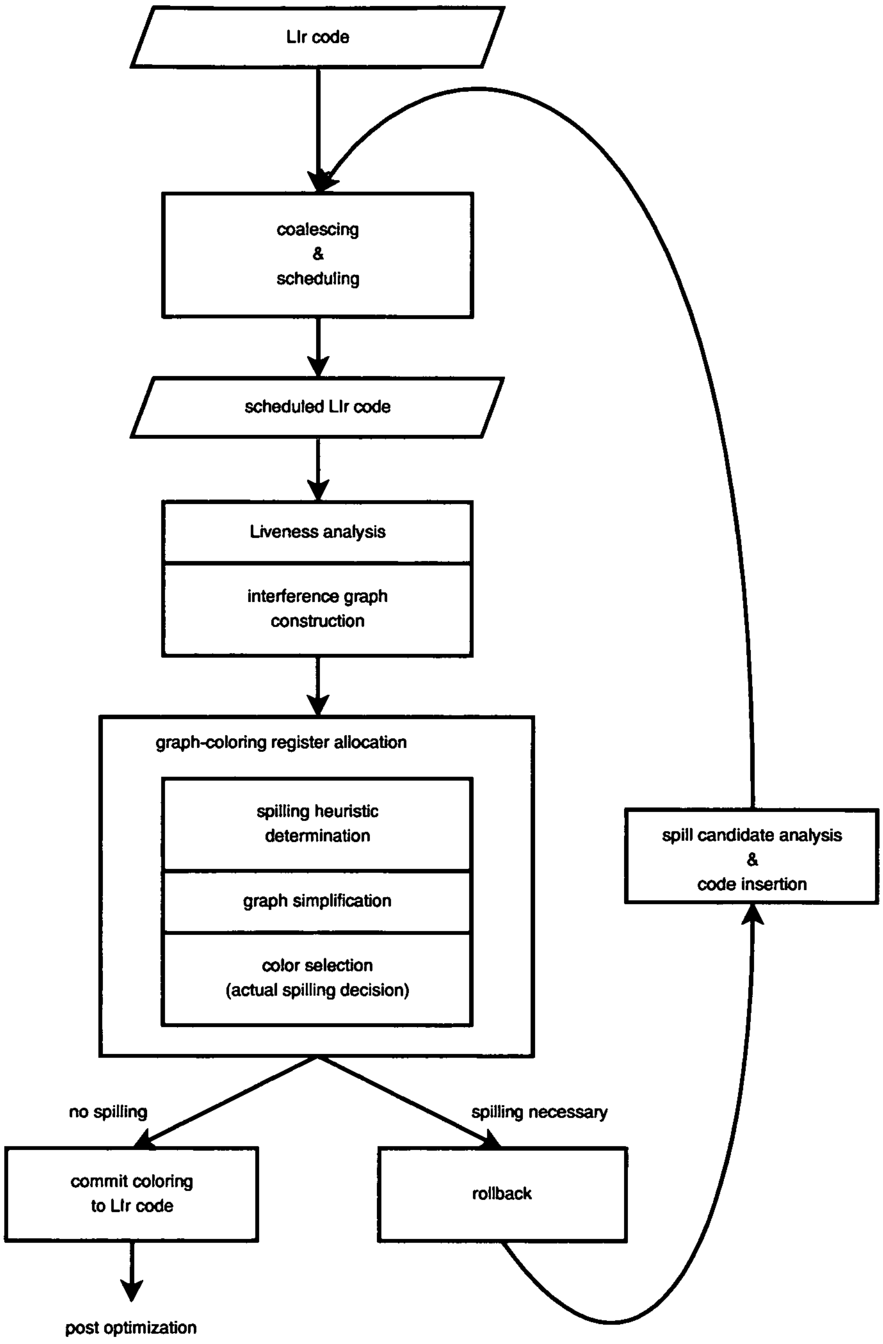
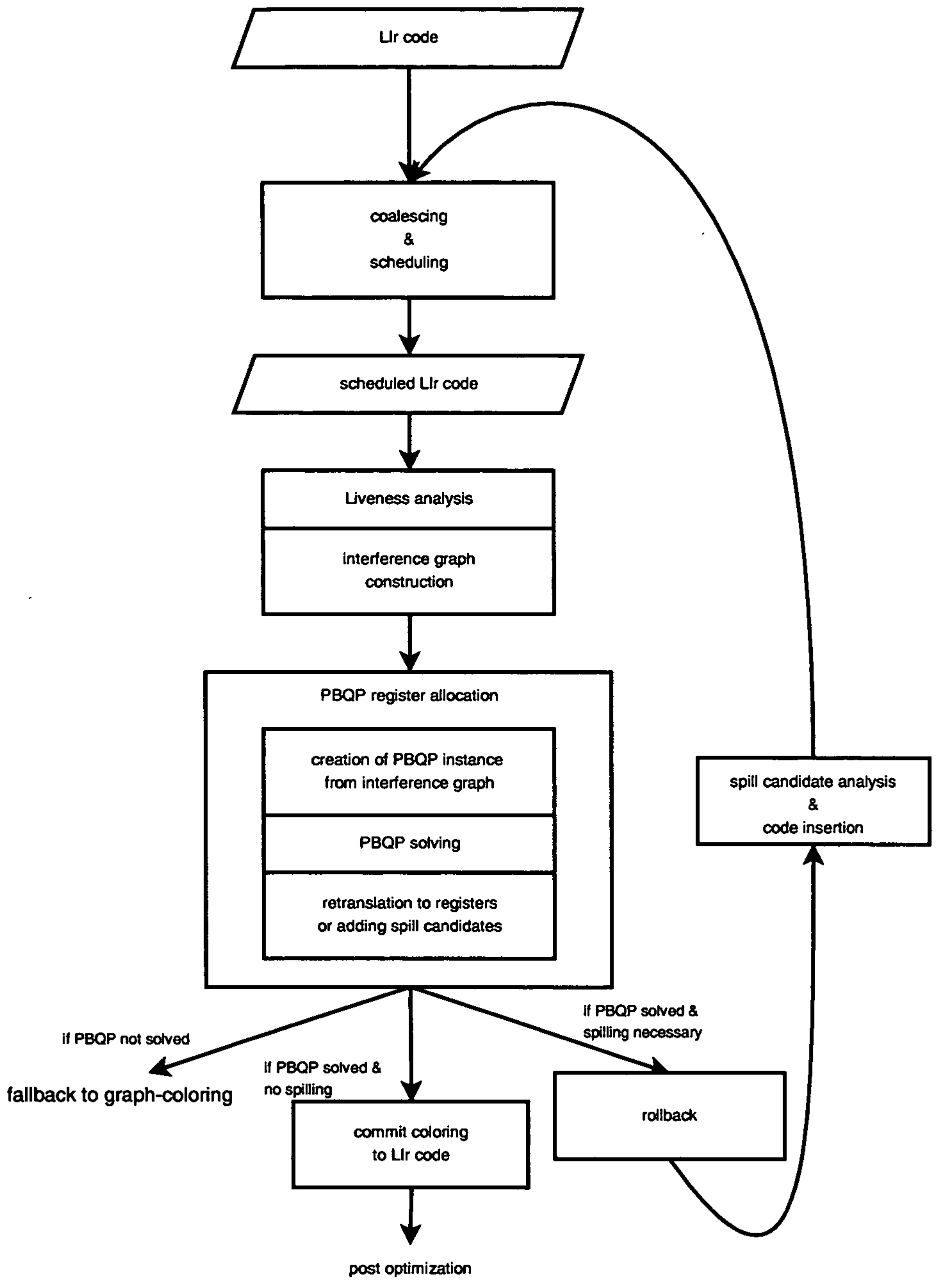


Figure 5.2: Sub–tasks of graph–coloring



Llr code
coalescing & scheduling
scheduled Llr code
Liveness analysis
interference graph construction
PBQP register allocation
creation of PBQP instance from interference graph
PBQP solving
retranslation to registers or adding spill candidates
spill candidate analysis & code insertion
if PBQP not solved
fallback to graph-coloring
if PBQP solved & no spilling
if PBQP solved & spilling necessary
commit coloring to Llr code
rollback
post optimization

Figure 5.3: Sub–tasks of PBQP register allocation

control flow graph of the function. The IN_i and OUT_i liveness sets are initialized to empty sets before the analysis starts.

While the general idea is still applicable to predicated code, the set representation and operators have to be adapted in order to include predication of the instructions. Set members are a 2–tuple of a *predicate id* and a *symbolic register*. The set operators $\setminus$ (difference) and $\cup$ (union) are defined as drafted by the examples in Equation (5.2).

$$\begin{aligned} \{(id1, reg1)\} \setminus \{(id2, reg1)\} &= \{(id1 \wedge \neg id2, reg1)\} \\ \{(id1, reg1)\} \cup \{(id2, reg1)\} &= \{(id1 \vee id2, reg1)\} \end{aligned} \tag{5.2}$$

A *predicate registry* and the special predicates $\top$ (always true) and $\bot$ (always false) are introduced in order to fully implement a predicated liveness analysis. The predicate registry is a database that keeps track of the points of predicate definitions. Those predicate definitions are realized as dummy instructions with only the predicate identifier in their DEF–sets. These dummy instructions are created during if–conversion (described in Section 5.2.1) and hold information about the predicate–defining conditional expression (from the eliminated conditional branch). The predicate registry allows to do simple "reasoning" about logical relations between different predicate identifiers. Due to the simple model of if–conversion which only introduces mutually exclusive simple predicates, the predicate registry only needs a simple implementation of boolean algebra. Predicate minimization by the *Quine–McCluskey* algorithm helps at reducing the overhead from introducing complex predicates from Equation (5.2). The predicate registry is used during construction of the interference graph. In general, edges between all members of the $DEF[n]$ and $OUT[n]$ sets are inserted in the graph. In presence of predicated code, this only has to be done if both predicates are possibly true at the same time. After liveness analysis is completed, the interference graph can be constructed according to the algorithm given in Figure 5.4.

Weighted interference graph with partial interferences Chapter 4 presented the layout of the register file for the xDSPcore architecture. The register file is banked (data and address registers), and the data registers are split into individually addressable sub–parts (shared registers).

Bank restrictions are handled completely at instruction selection. Each symbolic register is annotated with a type tag that encodes the bank to which it has to be assigned later. The instructions themselves can take operands of the correct bank only. No interference edges between live–ranges of different banks have to be added, and the colorability test for a node in the interference graph is performed for each bank individually. The interference graph itself is therefore split into two disjoint sub–graphs, one for the address and one for the data register bank. During the select phase of the register allocator, the bank tag of the live–range is used to restrict color selection to the particular subset. In other

```
procedure buildInterferenceGraph(function F)
begin
  interference_graph ifg = new interference_graph();
  forall instructions i ∈ F do
    forall d ∈ DEF(i) do
      forall o ∈ OUT(i) do
        if ( (d≠o) ∧ ¬ (i is copy o→d) ∧
          F.registry.maybe_both_true(d.predicate,o.predicate) )
        then
          ifg.addEdge(d.liverange,o.liverange)
        endif
      endfor
    endfor
  endfor
  F.setInterferenceGraph(ifg);
end
```

Figure 5.4: Interference graph construction

words, live–ranges from one bank interfere implicitly with the registers of the other bank.

The shared data registers need a different modeling in the interference graph. Generally, the graph encodes restrictions on symbolic registers that arise from control and data flow in the program (liveness and interference). This only works well for regular general purpose register files, where no *architectural* restrictions on registers exist. In irregular architectures, those architectural restrictions have to be incorporated as well. The approach in the xDSPcore backend is similar to the proposal presented by Smith in [22, 23], a *weighted interference graph.*

In such a graph, the edges encode the restrictions from data flow. Additionally, architectural restrictions are represented by node weights. For example an A–register has double the weight (2) of a D-register (1). One crucial factor is the colorability–test of a node. The original approach of Chaitin uses the degree of a node to test if this node is colorable or not. In the weighted interference graph this is more complicated, because the weights of the adjacent nodes have to be considered as well. Equation (5.3) shows how the weighted degree of a node in the *WIG* is determined.

$$d_n = \sum_{j \,\in\, adj(n)} \lceil \frac{w_j}{w_n} \rceil * w_n, \tag{5.3}$$

where $adj(n)$ is the adjacency list of the node n, w_n the weight of n. If d_n is less than the number of architectural registers (of corresponding weight) multiplied by their weight, the node is of insignificant degree and can be colored.

```
ldao (global_A),d72        || ldao (global_B),d74   ; DEF={d72,d74}, OUT={d72,d74}
nop                                                 ; DEF={}, OUT={d72,d74}
mul d72,d74,a75                                     ; DEF={a75}, OUT={d72,d74,a75}
add d72,d74,d76                                     ; DEF={d76}, OUT={d74,d76,a75,d72}
stao A75_{15,0},(global_C) || stao d72,(global_R)   ; DEF={}, OUT={d74,d76}
stao d74,(global_S)        || stao d76,(global_T)   ; DEF={}, OUT={}
```

Figure 5.5: Example assembly code

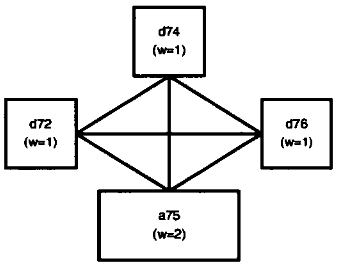


Figure 5.6: Resulting WIG

Figure 5.5 gives a short sequence of assembly code, annotated with the results from liveness analysis. The notation $An_{u,l}$ in the example from Figure 5.5 means, that only the bits u to l of register An are used as operand in the corresponding instruction.

The graph yielded from Example 5.5 is shown in Figure 5.6. When inserting nodes for shared registers (like for example $A75_{15,0}$), it is necessary to use the corresponding base register ($a75$). The base register imposes the strongest constraint on colorability, and the simplification phase of the graph–coloring register allocator then has to judge colorability of these base registers. When assigning a color to the base register during re–construction of the interference graph, the shared registers are implicitly assigned a color as well. Applying Equation (5.3) to the nodes of the graph in Figure 5.6 gives results as shown in Table 5.1.

From the resulting values, it can be argued that this code needs at least three

node	d_n
d72	4
d74	4
a75	6
d76	4

Table 5.1: Colorability test for example graph

node	register
d72	D4
d74	D3
a75	A0
d76	D2

Table 5.2: Register assignment

accumulators to be colored without spilling. Simplification removes the nodes in the order $d72$, $d74$, $d76$, and $a75$. Then the nodes are re–inserted and assigned an available color. $a75$ is assigned the first available accumulator $A0$, $d76$ conflicts with $a75$ and therefore has to be assigned to a shared register of another accumulator ($D2$). $d74$ conflicts with both $a75$ and $d76$ and is assigned register $D3$. Finally, $d72$ conflicts with all other registers and has to be assigned a shared register from an additional accumulator ($D4$). The register assignment is again shown in Table 5.2. Although this is better than without any modeling of shared registers (potentially needing four accumulators), this is still not optimal. The interference graph does not yet model *partial interferences* like those arising at the `add` instruction of the example. In fact, `d76` interferes with $A75_{15,0}$, which is only the lower part of the accumulator $a75$. Therefore, the edges of the interference graph are augmented with information, which *parts* of the adjacent nodes interfere.

In order to get this information, liveness analysis has to be done based on *register leaves*. These leaves correspond to the smallest indivisible and separately addressable parts of a base register. For example, the register $A0$ can exist "standalone" or can also act as the two leaves $D0$ and $D1$. This depends on which instructions access this register or parts of it. In the example, referencing $A75_{15,0}$ in the `stao` instruction causes the register $a75$ to be handled as an accumulator consisting of two register leaves which are handled separately in the liveness sets.

With this improvement, liveness analysis of the example yields the refined results as depicted in Figure 5.7. Interference graph construction now still adds edges $(a75, d72)$ and $(a75, d74)$ which arise at the `mul` instruction. When handling the `add`, the edge $(d76, a75)$ is inserted but augmented with the partial interference information. The augmented graph is depicted in Figure 5.8. Coloring this graph works different than before. The nodes are removed in order $d76$, $d72$, $d74$, and $a75$. $a75$ is assigned to $A0$, $d74$ and $d72$ both conflict with $a75$ and are assigned registers $D2$ and $D3$. $d76$ conflicts with $d72$, $d74$, and the lower part of $a75$. Therefore it is possible to assign it to register $D1$. This coloring is again shown in Table 5.3. Only two accumulators are needed now.

```
ldao (global_A),d72        || ldao (global_B),d74   ; DEF={d72,d74}, OUT={d72,d74}
nop                                                  ; DEF={}, OUT={d72,d74}
mul d72,d74,a75                                      ; DEF={a75}, OUT={d72,d74,$A75_{15,0}$}
add d72,d74,d76                                      ; DEF={d76}, OUT={d74,d76,$A75_{15,0}$,d72}
stao $A75_{15,0}$,(global_C) || stao d72,(global_R)    ; DEF={}, OUT={d74,d76}
stao d74,(global_S)        || stao d76,(global_T)   ; DEF={}, OUT={}
```

Figure 5.7: Example assembly code with refined liveness sets

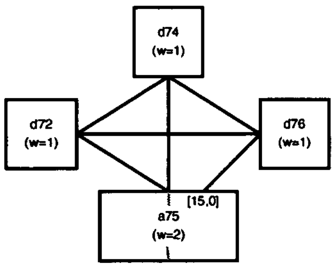


Figure 5.8: Augmented WIG

node	register
d72	D2
d74	D3
a75	A0
d76	**D1**

Table 5.3: Register assignment for augmented WIG

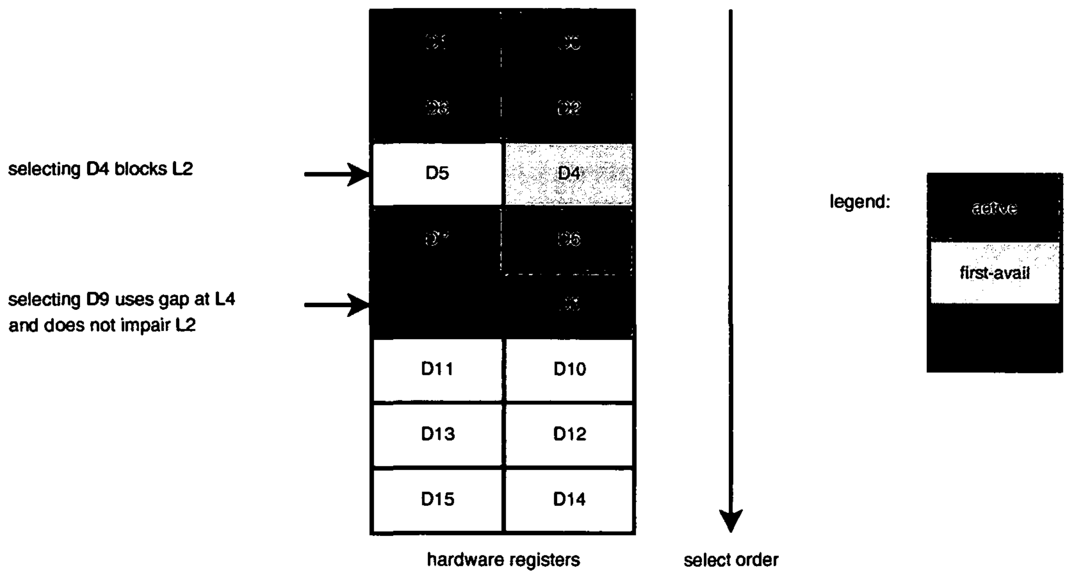


Figure 5.9: Selecting colors from gaps

Selecting colors

Selecting colors from the available set is done during the last phase of the graph–coloring register allocation. Every time a node n is re–inserted in the graph, the set of *active registers* A is determined (that are those registers that are assigned to adjacent nodes of n). With H the set of all hardware registers, an unused register from the set $C = H \setminus A$ is assigned to the re–inserted node n. If C is the empty set, n becomes an *actual spill* and is put onto a list of spilling candidates for later treatment.

The optimistic coloring approach benefits from the fact that some of the adjacent nodes probably are assigned the same color. Therefore, also significant nodes can be colored in many cases. For regular register files, the order of color selection from the set C is not overly important. For register files with shared registers, color selection has to be done more carefully.

During the select phase, it can occur that the set A has some gaps between active registers. When selecting colors for a D-register with a 'first–available' strategy, these gaps are not necessarily used, but instead a new base register gets occupied. That may impose stronger constraints on any node that is colored later. When first looking for gaps in the active register set, these are re–used and filled, and therefore a larger set of available base registers remains. Figure 5.9 briefly depicts an example.

Spilling heuristics

Generally, k–coloring a graph is done by graph simplification through repeatedly removing nodes with insignificant degree. In situations where only nodes with significant degree are left in the graph, one node has to be chosen for removal. Afterwards, simplification recovers again.

Choosing the node is done by a heuristic function that considers several metrics of the node. In general, it is most likely to remove a node with high degree because this brings most benefit for colorability. On the other hand, spilling costs have to be kept low and therefore a node with few definition and use points has to be favored. These metrics are conflicting and there is no best combined metric function.

As already suggested by Briggs in [17], a best–of–N coloring scheme should be performed in order to reduce the so–called *heuristic noise.* A lot of other works focused on finding better and more appropriate spilling heuristics. In the xDSPcore compiler, a total of 12 heuristic functions is implemented. Equations (5.4) show the implemented heuristics, where p is the calculated spill priority for node n, c the spilling costs for the node, d the degree of n as defined in Equation (5.3), and a the area of the live range of n. Spilling costs are determined through a weighted sum of all use and definition points of the live range. The area of n is the sum of the numbers of live–variables at all instructions where n is live–out.

Best–of–N coloring does graph–simplification and color selection for each of the N heuristic functions and calculates the respective spill costs from the actual spills. Any time a heuristic decision has to be taken, the node with smallest value p from the current heuristic function is selected. The heuristic function which causes least overall spill costs is chosen, and the actual spills from this particular select phase are passed over to the spill code inserter. If any heuristic succeeds at finding an allocation without spilling, the whole process is done and the register assignment of the interference graph is committed to the low–level intermediate representation of the code.

Early experiments have shown, that 4 of those spilling heuristics are predominant and significantly used more often than the others. These are heuristics from Equations (5.4a), (5.4d), (5.4f), and (5.4l), which have been made 'default' for compilation.

Optimal register allocation

The algorithm used for optimal register allocation is based on a *Partitioned Boolean Quadratic Problem (PBQP).* The PBQP approach is a unified approach for register allocation that can model a wide range of peculiarities. In addition coalescing is an integral part of the register assignment which is necessary for achieving good allocations.

The PBQP approach uses cost functions for register assignment decisions.

$$p = c \tag{5.4a}$$
$$p = \frac{1}{d} \tag{5.4b}$$
$$p = \frac{1}{a} \tag{5.4c}$$
$$p = \frac{c}{d} \tag{5.4d}$$
$$p = \frac{c}{a} \tag{5.4e}$$
$$p = c * d \tag{5.4f}$$
$$p = c * a \tag{5.4g}$$
$$p = \frac{c}{d * a} \tag{5.4h}$$
$$p = \frac{c}{d^2} \tag{5.4i}$$
$$p = \frac{c}{a^2} \tag{5.4j}$$
$$p = \frac{c^2}{d * a} \tag{5.4k}$$
$$p = \frac{c^3}{d * a} \tag{5.4l}$$

Figure 5.10: Spilling heuristics in the xDSPcore compiler

The cost functions have to fulfill two tasks: (1) cost functions that express mathematically the model of spilling costs of the architecture, and (2) cost functions that describe interference constraints, coalescing benefits, and constraints which stem from the CPU architecture. Basically, there are two classes of cost functions. One class of cost functions model the costs and constraints involved for one symbolic register, and the second class of cost functions model the costs and constraints of two dependent symbolic registers.

Table 5.4 lists the cost functions for the xDSPcore architecture. Spilling is modeled by a cost function for one symbolic register. The parameter c gives the costs for spilling and parameter a determines the allocation of symbolic register $\mathbf{s}$. A symbolic register is either spilled (that means a is equal to sp) or a register is assigned to it (that means $a \in \{R_0, R_1, \dots\}$). Depending on this decision different costs are involved. The architecture features four register classes (A,L,D,R) which can be modeled by $c_{\mathbf{s}}(a)$, where $class(\mathbf{s})$ is the set of registers of corresponding type which can be assigned to $\mathbf{s}$. Registers which are disabled for a symbolic register have ∞ costs and therefore are excluded for register assignments. An interference of two symbolic registers $\mathbf{s}_1$ and $\mathbf{s}_2$ is given by $i_{\mathbf{s}_1\mathbf{s}_2}(a_1, a_2)$. Either both allocations have different register assignments ($a_1 \neq a_2$) or one of the registers is spilled ($a_1 = sp$). Again, ∞ costs will be raised if for both symbolic register the same CPU register is allocated. The shared register interference constraint is given in equation for $d_{\mathbf{s}_1\mathbf{s}_2}(a_1, a_2)$. This constraint is necessary since two short registers share the same memory of one long register in the architecture. Coalescing costs of two symbolic registers are given by $p^{(b)}_{\mathbf{s}_1\mathbf{s}_2}(a_1, a_2)$. If both register assignments of $\mathbf{s}_1$ and $\mathbf{s}_2$ are identical, a coalescing benefit is obtained expressed as a negative number $-b$. Table 5.5 shows cost functions for the example of Figure 5.7 (for a small register file consisting of two accumulators only).

The cost functions are used for constructing cost matrices and cost vectors for the NP-hard PBQP problem (see Figure 5.11). The problem instance is solved by a dynamic programming approach. After problem solving, the solution has to be re–translated. This is done either by committing of the register assignments to the low–level intermediate representation (when all are assigned), or by putting the spill candidates on the list for later treatment.

The PBQP problem can be solved by dynamic programming as proposed in [89]. In each step of the algorithm a vector $\vec{x}_i$ is eliminated until the objective function f becomes trivial, i.e. the first part of the sum $\sum_{1\leq i<j\leq n} \vec{x}_i \cdot C_{ij} \cdot \vec{x}_j^T$ vanishes. Then, the solution of remaining vectors in the objective function is determined. Reduced vectors can be computed by reconstructing the original objective function. Unfortunately, not all reductions can be applied in polynomial time. Therefore, a recursive enumeration is necessary for obtaining the optimal solution. Basically, there are three reductions: reduction RI for nodes of only one cost matrix, reduction RII for nodes of two cost matrices, and reduction RN for nodes with arbitrary number of cost matrices. Reductions RI and RII can be solved in polynomial time — reduction RN needs exponential time. Figure 5.13

Spilling:

$$s^{(c)}_{\mathbf{s}}(a) = \begin{cases} c, & \text{if } a = sp \\ 0, & \text{otherwise} \end{cases}$$

Class Constraint:

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

Interference:

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

Shared Register(Interference)

$$d_{\mathbf{s}_1\mathbf{s}_2}(a_1, a_2) = \begin{cases} \infty, & \text{if } a_1 \in \mathit{shared}(a_2) \\ 0, & \text{otherwise} \end{cases}$$

Coalescing:

$$p^{(b)}_{\mathbf{s}_1\mathbf{s}_2}(a_1, a_2) = \begin{cases} -b, & \text{if } a_1 = a_2 \wedge a_1 \neq sp \\ 0, & \text{otherwise} \end{cases}$$

Table 5.4: Cost functions for the xDSPcore

$$\min f = \left[\sum_{1 \leq i < j \leq n} \vec{x}_i \cdot C_{ij} \cdot \vec{x}_j^T \right] + \left[\sum_{1 \leq i \leq n} \vec{c}_i \cdot \vec{x}_i^T \right]$$

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

$$\forall a_k, a_l \in A : C_{ij}(k, l) = \sum_{f_{\mathbf{s}_i\mathbf{s}_j} \in F_{\mathbf{s}_i\mathbf{s}_j}} f_{\mathbf{s}_i\mathbf{s}_j}(a_k, a_l)$$

$$\forall a_k \in A : \vec{c}_i(k) = \sum_{f_{\mathbf{s}_i} \in F_{\mathbf{s}_i}} f_{\mathbf{s}_i}(a_k)$$

Figure 5.11: The PBQP problem formulation

	$s_{\mathbf{S}}^{(c)}(a)$				$c_{\mathbf{S}}(a)$			
	d72	d74	a75	d76	d72	d74	a75	d76
A0	0	0	0	0	∞	∞	0	∞
A1	0	0	0	0	∞	∞	0	∞
L0	0	0	0	0	∞	∞	∞	∞
L1	0	0	0	0	∞	∞	∞	∞
D0	0	0	0	0	0	0	∞	0
D1	0	0	0	0	0	0	∞	0
D2	0	0	0	0	0	0	∞	0
D3	0	0	0	0	0	0	∞	0
sp	4	4	2	2	0	0	0	0

$i_{\mathbf{S}_{d72}\mathbf{S}_{d74}}$	D0	D1	D2	D3	*sp*
D0	∞	0	0	0	0
D1	0	∞	0	0	0
D2	0	0	∞	0	0
D3	0	0	0	∞	0
sp	0	0	0	0	0

$i_{\mathbf{S}_{d72}\mathbf{S}_{a75}}$	A0	A1	*sp*
D0	∞	0	0
D1	∞	0	0
D2	0	∞	0
D3	0	∞	0
sp	0	0	0

$i_{\mathbf{S}_{d72}\mathbf{S}_{d76}}$	D0	D1	D2	D3	*sp*
D0	∞	0	0	0	0
D1	0	∞	0	0	0
D2	0	0	∞	0	0
D3	0	0	0	∞	0
sp	0	0	0	0	0

$i_{\mathbf{S}_{d74}\mathbf{S}_{a75}}$	A0	A1	*sp*
D0	∞	0	0
D1	∞	0	0
D2	0	∞	0
D3	0	∞	0
sp	0	0	0

$i_{\mathbf{S}_{d74}\mathbf{S}_{d76}}$	D0	D1	D2	D3	*sp*
D0	∞	0	0	0	0
D1	0	∞	0	0	0
D2	0	0	∞	0	0
D3	0	0	0	∞	0
sp	0	0	0	0	0

$i_{\mathbf{S}_{d76}\mathbf{S}_{a75}}$	A0	A1	*sp*
D0	∞	0	0
D1	0	0	0
D2	0	∞	0
D3	0	0	0
sp	0	0	0

Table 5.5: Cost functions for example 5.7

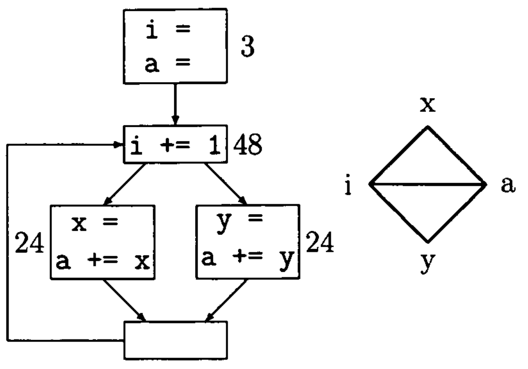


Figure 5.12: Control flow and interference graph

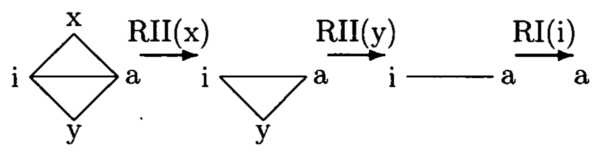


Figure 5.13: Example of Reduction

depicts the reduction steps for the interference graph in Figure 5.12. An edge between two nodes represents a cost function and a node a vector $\vec{x}$. The example does not impose RN reductions and can be solved in polynomial time.

The RN reduction for the optimal solution is given in Figure 5.14. The first loop enumerates all possible solutions of vector x. For a given solution the costs are determined. If it is smaller than the current minimum the solutions of the remaining vectors are saved. The reduction of the vector can split the PBQP graph in several independent sub-graphs (scc). The performance of the algorithm can be substantially improved by solving the independent sub-graphs on their own. For reducing the number of RN nodes it is a good heuristic to select the vector with the highest number of cost matrices.

Coalescing

High-level optimization based on *static single assignment (SSA)* form produce a considerable amount of copy instructions in the low-level code (arising from ϕ-terms). Low-level code transformations and techniques as shown in the next sections introduce additional copy instructions as well. Coalescing eliminates such copy instructions by merging the related operands into one single operand. The operands can be merged if they do not interfere. As shown in Section 3.1.1, there are different strategies for merging the copy-related nodes (*aggressive, conservative, optimistic*).

```
procedure ReduceN($x$)
begin
   $min := \infty$;
   for i:=1 to $|\vec{c}_x|$ do
      $h := \vec{c}_x(i)$;
      for all $y \in adj(x)$ do
         $\vec{c}_y := \vec{c}_y + C_x y(i,:)$;
      end
      remove $x$;
      for all $scc \in G$ do
         solve scc;
         $h := h$+cost(scc);
      endfor
      if $h < min$ then
         save solutions
      endif
      reconstruct node $x$
   endfor
   restore min. solution
end
```

Figure 5.14: RN Reduction

In the traditional approach of register allocation, coalescing is implemented as an intrinsic sub–task. For the xDSPcore backend, where register allocation is done after instruction scheduling, this would be too late. The presence of the (unnecessary and therefore volatile) copy instruction at scheduling–time has strong impacts on the resulting schedule. The physical resources for the copy instructions had to be allocated, potentially creating conflicts with other non–volatile instructions. These may be artificially delayed and thus result in extended schedules. Furthermore, pruning the copy instructions leaves back holes in the schedules, resulting in bad resource utilization.

For these reasons, coalescing is implemented as a stand–alone optimization pass for sequential code. This also allows to perform coalescing at any place prior to scheduling. Later sections on if–conversion (5.2.1) and instruction scheduling (5.2.2) will introduce estimation functions for the schedule length. For the appropriateness of these estimations, it is necessary to apply coalescing even before if–conversion.

In order not to impair colorability through over–coalescing of symbolic registers, only conservative strategies are applied in these early optimization phases. Both George's and Briggs' strategies are implemented and can be user–selected. George's strategy coalesces two registers a and b to a register ab if for every neighbor t of a either t already interferes with b or the degree of t is $< K$ (where K is the number of available registers). Briggs' strategy coalesces two registers a and b to a register ab if ab has fewer than K neighbors with a degree $\geq K$.

The actual coalescing in the register allocation loop is performed depending on the current register allocation approach. Figure 5.15 gives the details on coalescing for both register allocation strategies.

For graph–coloring, both aggressive and conservative strategies can be applied, depending on current compiler setup. An interference graph for the sequential code has to be built for doing the actual coalescing decisions.

Coalescing in case of PBQP register allocation works differently and needs an invocation of register allocation for the sequential code. The register assignments and spilling decisions made in this pass are discarded and only the coalescing decisions are considered for performing the actual coalescing. The second PBQP invocation (now on scheduled code) then only has to consider register assignments or spilling decisions. No further coalescing decisions have to be taken now.

doConservativeBriggs, doConservativeGeorge: boolean flags which specify the coalescing strategy (user–specified)

```
procedure coalesceGraphColoring(function F)
begin
  interference_graph ifg=F.getInterferenceGraph();
  forall ( copy $\in$ F.getCopyInstructions() ) do
    sym_reg source=copy.getSourceReg();
    sym_reg destination=copy.getDestReg();
    if ( $\neg$ ifg.doInterfere(source,destination) $\wedge$
      ( doConservativeBriggs $\wedge$ ifg.isSaveByBriggs(source,destination)
      $\vee$ doConservativeGeorge $\wedge$ ifg.isSaveByGeorge(source,destination)
      $\vee$ ($\neg$ doConservativeBriggs $\wedge$ $\neg$ doConservativeGeorge) )
    ) then
      F.coalesce(source,destination);
      F.remove(copy);
    endif
  endfor
end

procedure coalescePBQP(function F)
begin
  interference_graph ifg=F.getInterferenceGraph();
  doPBQPRegisterAllocation(ifg);
  forall ( copy $\in$ F.getCopyInstructions() ) do
    sym_reg source=copy.getSourceReg();
    sym_reg destination=copy.getDestReg();
    color source_color=ifg.getAssignedColor(source);
    color dest_color=ifg.getAssignedColor(destination);
    if ( source_color = dest_color ) then
      F.coalesce(source,destination);
      F.remove(copy);
    endif
  endfor
  ifg.revokeColoring();
end
```

Figure 5.15: Details of coalescing

n–operand	(n-1)–operand	assembly example
$d := unary(s)$	$d := s$	`mov s,d`
	$d := unary(d)$	`abs d`
$d := binary(s1, s2)$	$d := s1$	`mov s1,d`
	$d := binary(d, s2)$	`lslr s2,d`
$d := ternary(s1, s2, s3)$	$d := s1$	`mov s1,d`
	$d := ternary(d, s2, s3)$	`mac s2,s3,d`

Table 5.6: Code transformation for (n-1)–operand instructions

Handling of special instructions

The xDSPcore instruction set includes instructions that have to be treated in a special way. Those instructions can be classified in:

- one–operand unary instructions (e.g. `abs Dx` ... absolute value, $d = |s|$)
- two–operand binary instructions (e.g. `lslr Dx,Dx` ... logical shift left by register, $d = s1 << s2$)
- three–operand ternary instructions (e.g. `mac Dx,Dx,Ax` ... multiply–accumulate, $d = s1 + s2 * s3$)

These instructions have an implicit source operand (which corresponds to the destination) and thus a slight code transformation for proper handling of those instructions is needed. Table 5.6 shows the necessary code transformations, with d meaning 'destination operand' and s, $s1$,$s2$,$s3$ meaning 'source operand(s)'.

As the examples in the table show, the problem of (n-1)–operand instructions can be handled by inserting additional copy instructions. Those additional copy instructions place an additional burden onto register coalescing.

Handling of calling conventions and pre–colored nodes

In general, calling conventions are a set of rules that clarify the responsibilities of preserving values over sub–routine boundaries. There are two basic categories of registers: (1) caller–saved registers, and (2) callee–saved registers. Caller–saved registers have to be saved by the calling routine (the *caller*) if they contain values that are still live after the sub–routine call. Callee–saved registers have to be saved by the called function (the *callee*) if it uses any of those registers. Therefore, the caller can rely on those registers having the same values *after* the callee has returned than they had before. Additionally, calling conventions specify which registers are used for parameter and result passing, and which register is used as the stack (or frame) pointer. Unused argument registers are used as additional caller–saved registers, the stack pointer is callee–saved. Calling conventions can be properly implemented by introducing special pre–colored nodes.

Argument registers Argument registers are involved twice: (1) for the incoming parameters of the current function, and (2) for passing outgoing arguments to a called sub–routine.

The parameters of the current function are realized as local variables and are represented by a symbolic register. In order to receive the correct value at function entry, special copy instructions from the parameters' entry locations (the argument registers) to the parameters' function locations (the symbolic registers) are inserted in the LIr code. The source operands of those copy instructions are pre–colored to the particular argument registers.

At function calls, the arguments are evaluated into symbolic registers, and again copy operations from the arguments' locations to the corresponding registers for parameter passing are inserted. That means, that the destination of the copy instruction is pre–colored to the particular register.

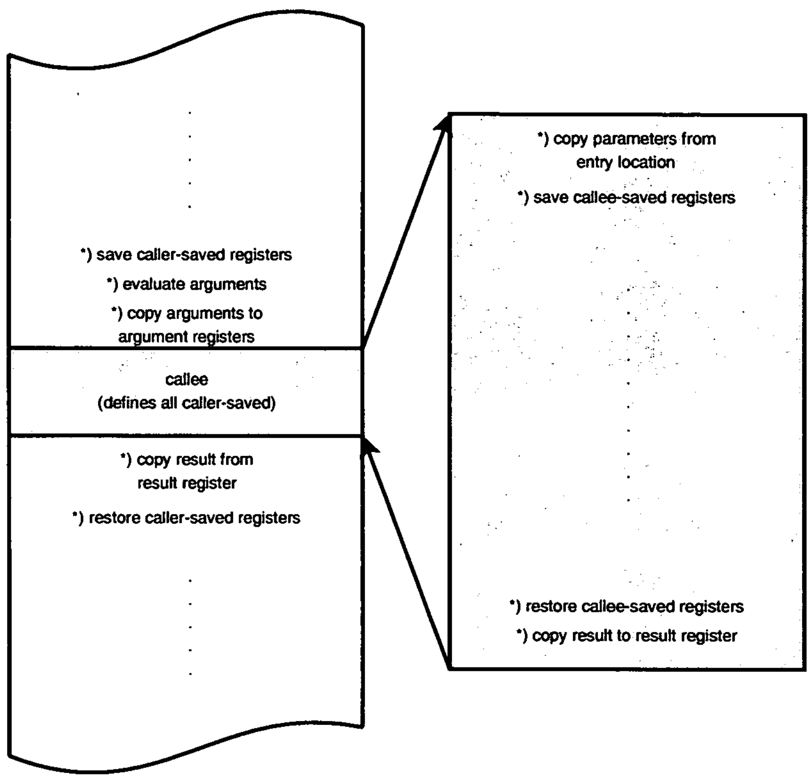
The result of the function call (if any) is handled the same way as parameters. It is copied from its return location (the pre–colored return register) to the function location (a symbolic register).

Caller–saved registers Saving and restoring caller–saved registers is done by using the regular spilling mechanism. Each sub–routine call is annotated with a *define*–set of all caller–saved registers. Thus, if any symbolic register is live across a call, it interferes with all caller–saved registers and therefore either has to be assigned a callee–saved register or spilled to memory. The spilling heuristics will decide, which of the symbolic registers that are live across the call will be assigned a register or spilled.

Callee–saved registers Saving and restoring callee–saved registers is handled after register allocation has finished. Whenever a function uses a callee–saved register for any of its symbolic registers, appropriate store and load instructions are inserted at function entry and exit.

Figure 5.16 depicts the responsibilities and actions for a proper implementation of calling conventions. Additional copy instructions are introduced and have to be treated during register coalescing.

Coalescing a symbolic register s and a pre–colored register p can only be done if s does not interfere with any other register that is equally pre–colored as p. This situation can occur at symbolic registers that are live across a sub–routine call. Coalescing such registers with a pre–colored one (and thus implicitly pre–coloring it) violates the interference constraint at the call and thus results in an improper implementation of calling conventions. Without coalescing, the symbolic register can be either assigned a callee–saved register or spilled at the sub–routine call.



*) save caller-saved registers
*) evaluate arguments
*) copy arguments to argument registers
callee
(defines all caller-saved)
*) copy result from result register
*) restore caller-saved registers
*) copy parameters from entry location
*) save callee-saved registers
*) restore callee-saved registers
*) copy result to result register

Figure 5.16: Illustration of calling conventions

Termination

The register allocation loop has no clear termination condition [2] and in early implementations, cascading spills due to misplaced spill code have led to a virtual endless loop.

Nevertheless, termination can be proved (rather informally) by the following considerations: It is necessary (1) to identify situations with maximum register pressure, (2) to check if those situations are locally colorable, and (3) to find a spill code placement for such situations to make them also globally colorable.

Maximum register pressure can be constructed through a schedule where the maximum of register reads or writes is performed at one pipeline edge, assuming that all the operands are pairwise different. In general, the number of source operands is greater than the number of destination operands, therefore it is sufficient to focus on register reads. Due to the banked register file and the pipelined execution of instructions, it is necessary to consider read matrices $R = \left(\begin{smallmatrix} r_0 & r_1 & \cdots & r_n \\ d_0 & d_1 & \cdots & d_n \end{smallmatrix}\right)$ with the components r_n for reads of address registers in stage n and d_n for reads of data registers in stage n. Each instruction of the instruction set is assigned such a read matrix. Using instructions with component wise maximums, a code sequence with maximum register reads can be constructed. For the current instruction set of xDSPcore and the current setup of functional units (2xMOV,2xCMP,1xBR), this yields the following sequence for the data registers:

```
st R1,(M)       || st R2,(M)
mac R3,R4,R5    || mac R6,R7,R8  || jmpr R9
```

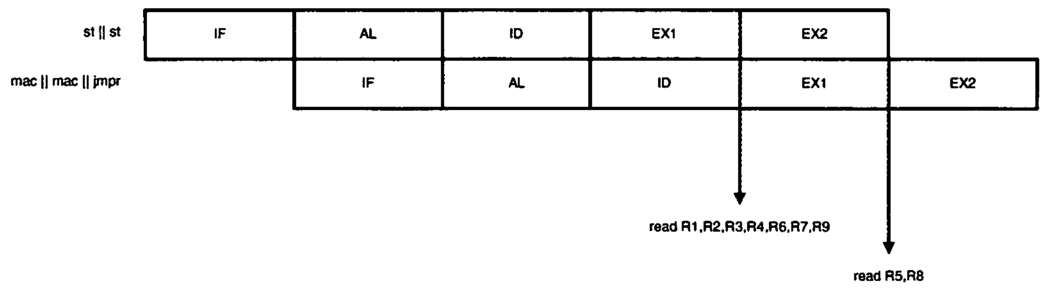


Figure 5.17: Maximum reads on data register file

The maximum of register reads at any pipeline edge is therefore less than or equal to 7, so this sequence is guaranteed to be locally colorable with the given register file (8 accumulators in the data register file). The same can be shown for the address registers.

[2]Termination relies on the heuristic behavior, that iteratively inserting spill code makes the code colorable at some point. In general, this is not guaranteed.

Making such a sequence globally colorable depends on liveness of other variables which are not directly involved and the liveness of the input values. Variables which are *live-trough* (that means, they are both live–in and live–out but neither in one of the def or *use* sets of the code sequence) can be spilled. This makes them disappear from the local register sets, reducing register pressure.

The destination operands then have to be assigned to registers which are not already used for the input operands. Register of input operands which are not live–out can be used as well. If this set of registers cannot hold the destination operands, some of the input operands have to be spilled as well. This makes them disappear from the live–out set and thus the register can be re–used for the destination.

From these considerations, it becomes apparent that the register allocation loop terminates if the spilling instructions are placed appropriately (and thus successfully reduce register pressure). Section 5.2.2 and Figure 5.41 will give some explanations on scheduling spilling instructions.

Complexity

Register allocation in general is known to be an NP–complete problem. Nevertheless, there are some heuristic approaches which can be applied and often deliver an optimal solution.

For the graph–coloring approach, optimality is reached when no spilling decisions have to be drawn. If spilling is necessary, the heuristic functions which select the live range to be spilled have to fulfill two conflicting goals (improve colorability by minimal spilling costs) and thus optimality cannot be guaranteed any more. Only a good approximation can be achieved but the run time behavior is nearly linear (in the size of symbolic registers and number of edges in the interference graph).

On the other hand, the PBQP approach finds a spill cost optimal solution in each iteration of the register allocation loop. Optimality can be guaranteed by applying special reductions to the PBQP problem. If only type RI and RII are applied, the optimal solution is found polynomial time. Reductions of type RN introduce exponential run time behavior and thus may not be accomplished in reasonable time (but often do). This makes the actual complexity of the algorithm sensitive to the *structure* of the interference graph.

Both strategies use the *interference graph* as a vehicle for the complete register allocation process. Building the interference graph itself is already a costly process, as it involves (an iterative) liveness analysis of the code, and has a quadratic component (in the def and out sets of each instruction). Especially for graph–coloring, building the interference graph dominates the run time of the actual coloring.

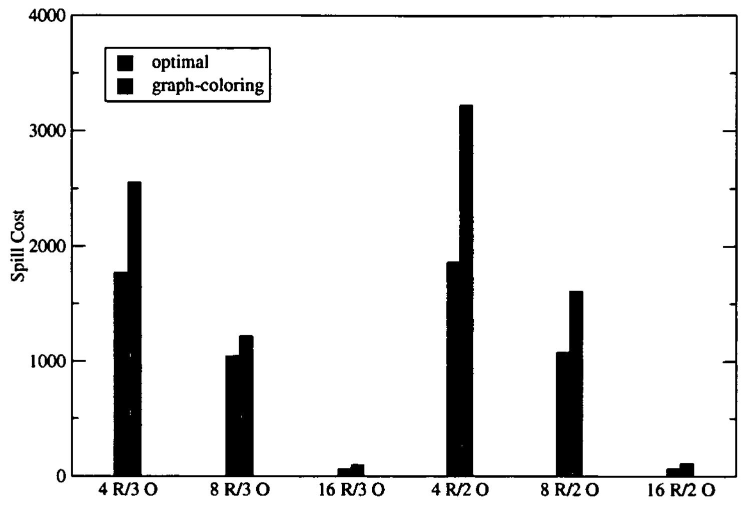


Figure 5.18: Spill cost comparison

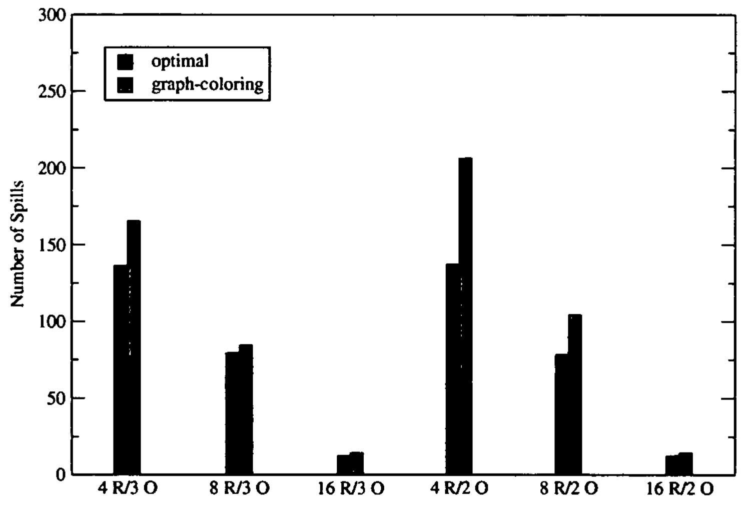


Figure 5.19: Number of spills comparison

Evaluation

Both register allocation approaches have been compared regarding static spilling costs. Several architectural models have been implemented. The number of base registers was set to 4,8, and 16 respectively. Additionally, code for three–operand mode and code for only two–operand mode, needing additional copy instructions and therefore coalescing decisions, was generated. The spilling costs and the number of spilled live–ranges of the particular register allocator for all benchmarks have been accumulated. Figure 5.18 shows the comparison of spilling costs in numbers of executed spilling instructions. Figure 5.19 shows the number of spilled live–ranges.

From these figures, it can be seen that the PBQP–based register allocator outperforms the extended graph–coloring approach. Spilling instructions are further reduced. This is essential for reducing energy consumption of the executed code.

5.1.2 VLIW optimization

Motivation

As shown in Section 2, communication on core buses is a main contributor to the total energy consumption of a processor. Since instruction fetching is inherent in today's processor architectures and instruction buses of VLIW architectures can become wide, there is a great potential in reducing the energy needs of the processor by reducing energy consumption during instruction fetch. The approach in xDSPcore is two–fold:

1. during execution of hardware loops, instruction memory accesses are reduced by stalling instruction fetch (after all instructions of the loop body have been fetched to the instruction buffer),

2. switching activities on the instruction memory bus are reduced by software optimization.

The optimization presented here refines VLIW operations in such a way, that fetching a function's instruction words from code memory causes as little dynamic switching as possible. This goal can be reached by minimizing the Hamming distance of consecutively fetched words.

Basics

Figure 5.20 shows an example of how an instruction sequence (upper part) is mapped into code memory (lower part). Each clock cycle, one complete line of code memory (i.e. a *fetch word*) is loaded into the instruction buffer. Due to the xLIW code compression and the instruction buffer, an *execution bundle* (depicted by the dashed lines) may start and end at any offset within the fetch word, even crossing fetch word boundaries. The order of instructions within one bundle is arbitrary, dispatching of instructions to the correct functional unit is solely done by the aligner unit. Optimization of the instruction–to–code–memory mapping can be done through the following refinement possibilities:

- permuting the operations of one execution bundle,

- operand swapping of commutative operations.

It has to be noted, that this optimization works as a post–pass optimization that does not impact performance nor code size of the application. For exploiting the full power of this technique, an implementation within the linker would be best. Only at this point in the compilation workflow, the complete binary coding information is available. However, implementing the optimization in the compiler is possible when applying a heuristic. For any two instruction words where several coding bits are not yet known, for example offset bits of branch instructions ('o'),

$i_{11} \parallel i_{13}$
$i_{21} \parallel i_{22} \parallel i_{24}$
$i_{31} \parallel i_{32} \parallel i_{33} \parallel i_{34}$
$i_{41} \parallel i_{42} \parallel i_{43}$
...
(i_{xy} means instruction i is executed in cycle x at functional unit y)

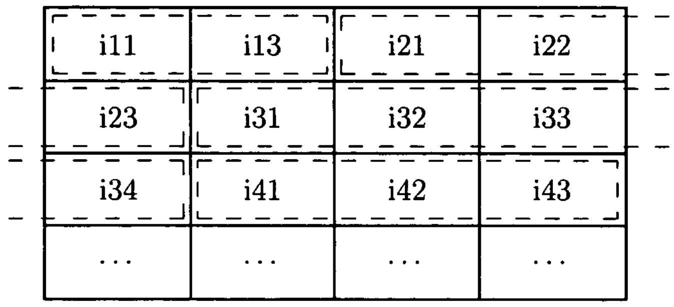


Figure 5.20: Instruction sequence and possible xLIW mapping to code memory

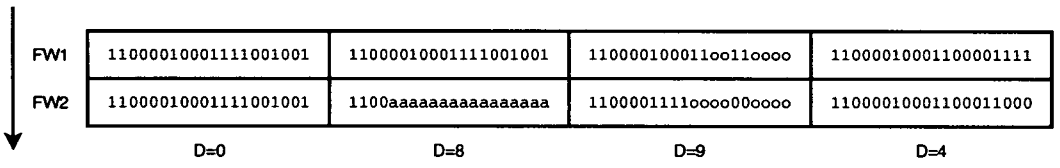


Figure 5.21: Illustration of switching heuristic during compilation

or absolute addresses of loads ('a'), it is assumed that half of the affected bits cause a bit transition. The example in Figure 5.21 depicts this heuristic. The first instruction pair of fetch words $FW1$ and $FW2$ contains identical instructions and has Hamming distance 0. The second instruction word pair has one fixed instruction word. The second instruction word has 16 'a' bits, the remaining bits are identical to the corresponding bits of the first word. This yields the heuristic Hamming distance 8. The third instruction word pair has 'o' bits in each of the words, affecting 8 bit positions. The remaining 12 bit positions are different at 5 positions, yielding a heuristic Hamming distance of 9. Finally, the last pair contains two fixed instruction words with a Hamming distance of 4. The overall Hamming distance for $FW1$ and $FW2$ used in later optimization results in 21.

Problem model

Calculating the Hamming distances of successive fetch words is straight–forward for linear program flow but is limited to the scope of basic blocks. A global approach for a whole function has to take inter–basicblock effects into account. Equation (5.5) shows the objective function for modeling the so–called 'global' Hamming distance for instruction fetch of a whole function F.

$$Dist_{glob}^{F} = \sum_{b \in B} f_b * \left(Dist_b^{int} + \sum_{s \in S_b} p_{b \to s} * Dist_{b \to s}^{ext} \right) \tag{5.5}$$

B is the set of F's basic blocks, f_b the execution frequency of block b, $Dist_b^{int}$ the internal Hamming distance of block b (see below), S_b the successor blocks of b, $p_{b\to s}$ the probability of branching from b to s, and $Dist_{b\to s}^{ext}$ the so–called external Hamming distance of block b to block s. Execution frequencies and branch probabilities are estimated by the following heuristic functions:

$$f_b = 10^l \tag{5.6}$$

with l the loop nesting level of b, and

$$p_{b \to s} = \begin{cases} \frac{1}{|S_b|} & \text{for non–loop branches,} \\ 0.9 & \text{for the back–edge of a conventional loop,} \\ 0.1 & \text{for the end–loop edge of a conventional loop,} \\ 0 & \text{for the back–edge of a hardware loop,} \\ \frac{1}{f_b} & \text{for the end–loop edge of a hardware loop.} \end{cases} \tag{5.7}$$

The internal Hamming distance of a block b is calculated as follows:

$$Dist_b^{int} = \sum_{i=1}^{N_{FWord}^b - 1} Dist\left(FWord_i^b, FWord_{i+1}^b\right) \tag{5.8}$$

where N_{FWord}^b is the number of fetch words in block b, and $Dist\left(FWord_i^b, FWord_{i+1}^b\right)$ the Hamming distance of the ith and $(i+1)$th fetch words of block b.

Calculating the external Hamming distance of a block to its successors works mostly analogous. There is only a difference if the first or last execution bundles of a basic block are not aligned. This has to be taken into account correctly. Figure 5.22 depicts an example. For edge $I \to T$, the Hamming distance of fetch words 3 and 6 has to be calculated, For $I \to E$ 3 and 4, for $T \to J$ 9 and 10, and for $E \to J$ 6 and 9.

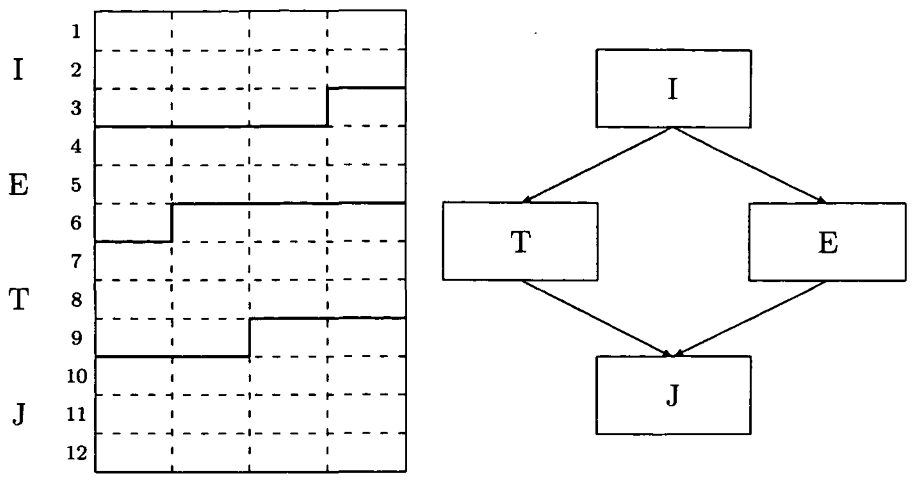


Figure 5.22: Illustration of block alignment

Algorithm

Due to the structure of the objective function, the optimization is done in two steps. The first step computes a **set** of locally optimal operation arrangements for each basic block. The second step then selects those local solutions that yield the minimal global Hamming distance under consideration of inter–basic block effects (as defined in Equation (5.5)).

Local Optimization With the xLIW programming model, the instructions of one execution bundle are not bound to one fetch word. In other words, when choosing one particular permutation, an instruction might be placed into a different fetch word than it would have been placed with another permutation. Solutions from previous work [56, 59] are restricted to aligned bundles that do not cross fetch word boundaries. Therefore those algorithms (bi–partite graph matching, shortest path) do not work for the xDSPcore architecture. Instead of graph methods, an enumerative approach with dynamic programming was implemented.

In principle, all possible arrangements (permutations and operand swapping) of each of the execution bundles in the basic block have to be enumerated to find the arrangement which results in the minimal internal Hamming distance. This enumeration yields an exponential run time complexity, but this can be reduced by a dynamic programming approach. The dynamic program makes use of the circumstance, that only the first and the last fetch words of the blocks are needed for the global expansion step. So only such combinations have to be considered, that differ in their resulting first and last fetch words. All other combinations can be eliminated in each step of the dynamic program. This reduces run time complexity to a virtually linear behavior [3], so that the algorithm is reasonably fast for typical code. It also still yields the optimal result, because **all** of the locally optimal solutions [4] are generated and later on used in the global expansion. The pseudo code in Figure 5.23 gives the details of the dynamic program.

Global Expansion Global expansion picks one particular local optimum for each block in order to minimize the *global* Hamming distance as defined in Equation (5.5). Two different strategies were implemented. The first one is a total recursive enumeration of all possible combinations of local optima, whereas the second strategy implements a genetic evolution for the global expansion step. Calculating the global Hamming distance as defined in Equation (5.5) is performed as shown in the pseudo–code of Figure 5.24.

Total Recursive Enumeration This strategy originally served as a reference implementation that always finds the optimal solution of the global expansion. It loops through all the basic blocks and their corresponding local optima recursively. Every time the last block of the function is reached, the global Hamming distance is evaluated according to Equation (5.5). If it is smaller than the currently known best solution, then the solution is memorized, otherwise it is discarded. This is continued until all combinations are checked. The pseudo code in Figure 5.25 gives the details. It is apparent that this exponential algorithm soon reaches its limits. Nevertheless, the limited problem structure [5] allows to enumerate almost all of the benchmark programs in reasonable time.

Genetic Evolution The search space for the global expansion is non–linear because of the presence of loops in the control flow graph. Therefore it is not straight–forward to find and implement a 'cheap' but complete algorithm. In recent years of compiler research, evolutionary approaches get more accepted. Therefore a genetic evolution for the global expansion set was modeled.

An individual of the population represents one possible combination of particular local optima. For each basic block the local index of the chosen optimum is stored in a string of integers (indexed by basic block indices). Two–point crossover between two individuals is done by interchanging parts of two strings and is performed during reproduction of a generation. The points for this operation are chosen randomly. Figure 5.26 depicts one crossover operation. Mutation

[3]linear in the number of bundles, but with some high constant factor which arises from the number of arrangements in first and last fetch word

[4]all combinations of different first and last fetch words ("borders")

[5]There are only a few different local optima because the parallelism of the basic block tails is often very limited. Additionally, the number of basic blocks in a function is also within modest bounds.

t_bundle: a set of instructions

t_dp_state: the current state of the dynamic program, i.e. the particular arrangements of the currently contained bundles.

t_container: a container for *t_dp_states*.

GenArrangementsOf(t_bundle): returns the set of all possible instruction arrangements of one bundle (considering permutations and operand swapping)

concat(t_dp_state first,t_dp_state second): concatenates two states of the dynamic program and returns the resulting state, i.e. the bundle(s) in 'second' are appended to the bundle(s) in 'first'

t_container.findEqualStateTo(t_dp_state other): tries to find a state in the container, that equals 'other'. Two states are considered equal, if both contain identical instruction arrangements in their first and last fetch words.

HammingDist(t_dp_state): evaluates the internal block Hamming distance of the currently contained bundles according to Equation (5.8).

```
procedure generateLocalSolutions(t_block basic_block)
begin
  t_bundle first_bundle=basic_block.getBundle(1);
  t_container initial_states = GenArrangementsOf(first_bundle)
  for (index=2 to basic_block.getNumberOfBundles() ) do
    t_container next_states = new t_container();
    t_bundle next_bundle=basic_block.getBundle(index);
    t_container P = GenArrangementsOf(next_bundle);
    forall s ∈ initial_states do
      forall p ∈ P do
        t_dp_state newstate=concat(s,p);
        t_dp_state existingstate=next_states.findEqualStateTo(newstate);
        if (¬∃ existingstate) then
          next_states.add(newstate);
        elseif (HammingDist(newstate)<HammingDist(existingstate)) then
          next_states.replaceStateBy(existingstate,newstate);
        else
          delete newstate;
        endif
      endfor
    endfor
    delete initial_states;
    initial_states=next_states;
  endfor
  basic_block.setLocalSolutions(initial_states);
end
```

Figure 5.23: DP algorithm for local VLIW optimization

getExtHD(t_function f, int solution[], t_block p, t_block s):
calculates the external Hamming distance of blocks 'p' and 's' under the specified operation arrangements from 'solution'.

```
function getGlobalHammingDist(t_function f, int solution[]):int
begin
 t_control_flow_graph cfg = f.getControlFlowGraph();
 int num_blocks = f.getNumberOfBlocks();
 int result = 0;
 for ( index = 1 to num_blocks) do
   t_block block = f.getBlock(index);
   t_container local_optima=f.getLocalSolutions(block);
   int block_solution = solution[index];
   result += local_optima.getInternalHammingDistFor(block_solution);
   for ( t_block succ ∈ cfg.getSuccessorsOf(block) ) do
     int succ_solution = solution[f.getIndexOf(succ)];
     result += getExtHD(f,solution,block,succ);
   endfor
 endfor
 return result;
end
```

Figure 5.24: Calculation of the global Hamming distance

```
procedure
 TRE(t_function f, int blockindex, int current[], int best[], int bestval)
begin
 if (blockindex > f.getNumberOfBlocks() ) then
  int hd=getGlobalHammingDist(f,current);
  if (hd<bestval) then
   copyArray(best,current);
   bestval=hd;
  endif
 else
  t_block block = function.getBlock(blockindex);
  t_container local_optima=function.getLocalSolutions(block);
  for int index=1 to local_optima.getNumberOfSolutions() do
   current[blockindex]=index;
   TRE(f,blockindex+1,current,best,bestval);
  endfor
 endif
end

procedure GlobalExpansionTRE(t_function f)
begin
 int num_blocks = f.getNumberOfBlocks();
 int current[] = new int[num_blocks];
 int best[] = new int[num_blocks];
 int bestval = MAX_INT;
 TRE(f,1,current,best,bestval);
 f.setGlobalSolution(best);
end
```

Figure 5.25: Total recursive enumeration

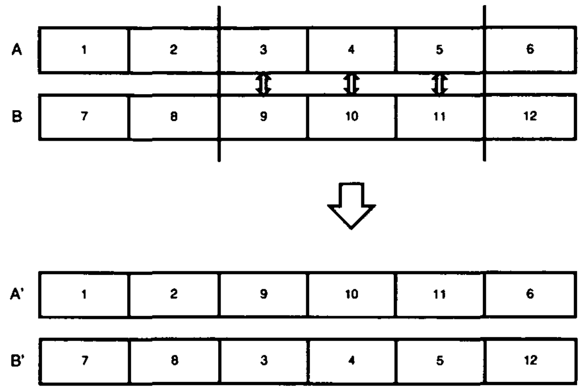


Figure 5.26: Illustration of two–point crossover

is done by changing one of the indices to another valid index and is done for a fixed ratio of randomly chosen genes. The global Hamming distance as defined in Equation (5.5) serves as fitness function. In order to achieve minimization, individuals with smaller fitness values are chosen for reproduction. The size of the population depends linearly on the number of possible incarnations, but is bound to a maximum of 1000 (in case of more than 1.000.000 possibilities). The first population is generated randomly, but includes an individual made of all the *absolute* local optima. Parents for the new generation are selected randomly within the fittest 50% of the population. The individual with best fitness survives unchanged (cloned reproduction) in order to guarantee a continuous evolution. If evolution stagnates [6] for a pre–defined number of generations, the current solution is considered as global optimum and evolution is stopped. The pseudo code in Figure 5.27 shows details of the genetic evolution.

Empirical Evaluation

For static evaluation of the improvements achieved by this refinement strategy, five different architectures are modeled. The maximum size for an execution bundle varies from 3 to 6 operations respectively. Table 5.7 shows these five architectural models, using the notation of instruction classes as introduced in Chapter 4.

Static evaluation covered a total of 239 functions and was done for all different architecture models. An *enhanced full rate* coder application contributes 95 functions, 16 functions are taken from the DSP kernels of the DSPstone benchmark suite, the rest of the benchmarks are various vector operations, digital filters, algorithms like Quicksort, Bubblesort, MD5, compress, etc.

The baseline for optimization was obtained through an evaluation of each

[6]fitness does not increase

t_individual: an array of integers, the elements of the array encode the index of the local optima for a basic block
t_population: a set of *t_individuals*, sorted by their global Hamming distances.
generateRandomPopulationFor(t_function f): generates a randomized set of individuals, evaluates and stores the global Hamming distance for each of them and inserts the individuals into the resulting *t_population*.
t_population.getFittestIndividual(): get the individual with the smallest global Hamming distance.
t_individual.getGlobHD(): returns the previously evaluated global Hamming distance of this individual.
t_population.selectParents(): select a random set of parents from the first half of the population.
t_population.reproduce(): perform a sexual reproduction of the parents with a two–point crossover until the original population size is reached.
t_population.doMutation(): mutates 0.1% of the genes.
t_population.extinct(): deletes the population and its individuals.

```
procedure GlobalExpansionGen(t_function f, int threshold)
 t_population G = generateRandomPopulationFor(f);
 int currentHD = G.getFittestIndividual().getGlobHD();
 int stagnation_count = 0;
 forever
  t_population P = G.selectParents();
  t_population C = P.reproduce();
  if (C.getFittestIndividual().getGlobHD() < currentHD) then
   currentHD=C.getFittestIndividual().getGlobHD();
   stagnation_count=0;
  else
   stagnation_count+=1;
   if (stagnation_count > threshold) then
    break;
   endif
  endif
  C.doMutation();
  G.extinct();
  G = C;
 endfor
 f.setGlobalSolution(G.getFittestIndividual());
end
```

Figure 5.27: Genetic Evolution

	MOV	CMP	BR
$VLIW_3$	1	1	1
$VLIW_{4,c}$	1	2	1
$VLIW_{4,m}$	2	1	1
$VLIW_5$	2	2	1
$VLIW_6$	2	3	1

Table 5.7: Number of parallel execution units

Model	LOC	GEN	TRE
$VLIW_3$	6.1 %	7.7 %	7.8 %
$VLIW_{4,c}$	6.1 %	7.8 %	7.9 %
$VLIW_{4,m}$	7.3 %	10.0 %	10.3 %
$VLIW_5$	7.6 %	10.4 %	10.8 %
$VLIW_6$	8.3 %	11.1 %	11.5 %

Table 5.8: Averages of Hamming distance reduction

function's unoptimized global Hamming distance. In the next step, the local optimization algorithm was applied. For each basic block, the overall minimum out of the set of optima was selected. The global Hamming distance for those blocks has been evaluated and yields a only locally optimized Hamming distance. Based on local optimization, the genetic evolution was applied for global optimization. Finally, a total recursive enumeration for solving the global problem was performed for all functions with less than a million different combinations of local optima.

Table 5.8 presents the geometric means of reduction of the global Hamming distance for all models. The improvements in are all relative to the unoptimized global Hamming distance. Table 5.9 shows a comparison of non–optimal and optimal solutions reached by genetic evolution.

	no. of solutions	
Model	sub–optimal	optimal
VLIW3	63	160
VLIW4c	68	144
VLIW4m	66	126
VLIW5	64	117
VLIW6	52	125

Table 5.9: Optimality of genetic evolution

5.2 Minor contributions

This section describes minor improvements on if conversion and instruction scheduling, and also presents a design space exploration methodology for embedded processors.

If conversion and instruction scheduling as described here do not primarily focus on optimizing energy consumption. Nevertheless, improvements in code efficiency in general reduce energy consumption and are therefore necessary to bring down the *energy consumption baseline* from which dedicated optimizations can start from.

Design space exploration is a method for embedded systems design and thus operates at one level above compiler optimization. The intention here is tailoring an embedded processor exactly to the needs of an application. Again, this is a method for improving efficiency of software and hardware and thus helps at reducing the energy consumption of the system.

5.2.1 If conversion

Motivation

Today's algorithms in signal processing (e.g. video codecs) contain a significant amount of not linear code (control code) in inner loops. These control code sections yield complex control flow graphs with many branch instructions. In VLIW architectures, branch instructions suffer from branch delay slots which impair performance when not filled with useful instructions. Further, complex control flow graphs complicate WCET analysis resulting in imprecise and over–estimated upper bounds for the worst–case execution time. With the *single–path programming paradigm* [9, 90], both drawbacks can be tackled at once. This paradigm can be supported substantially by the compiler through *if–conversion.* The single–path programming paradigm builds upon a code transformation that removes data–dependent branch instructions from the code. This yields a program which is fully temporally predictable, because it always executes on one single execution path.

Details

If conversion is a compiler optimization that translates control dependencies into data dependencies by using predicated execution (Section 3.1.2). While the original approach transforms the complete body of an innermost natural loop to one block, the implementation within the xDSPcore compiler backend is restricted to smaller control flow graph patterns. These patterns result from *if statements* or *conditional operators* in the source program.

Figure 5.28 shows the set of patterns which are included in the transformation, and how each of these patterns is transformed. This approach is similar

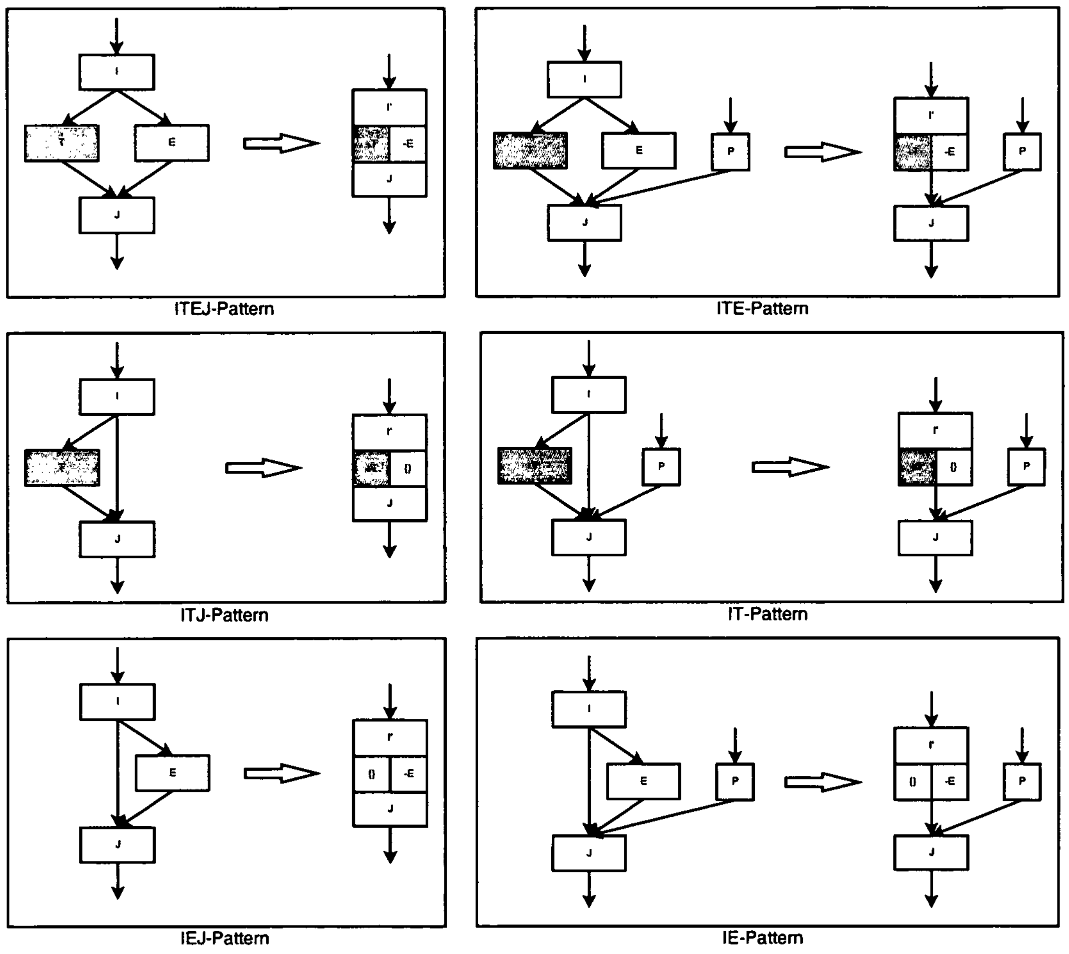


Figure 5.28: CFG patterns and transformed hyperblocks

```
   ...               # ...
   eval C            # eval C
   branchif C, true  # (C) t1 || (-C) f1 || pe
   nop               # (C) t2 || (-C) t2 || pe
   nop               # j1
false:               # j2
   f1                # ...
   f2                #
   branch join       #
   nop               #
   nop               #
true:                #
   t1                #
   t2                #
join:                #
   j1                #
   j2                #
   ...               #
```

Figure 5.29: An ITEJ pattern using branches or predicated execution

to the concept of the *hyperblock*([29]), except for still having a single exit. The notation *+T* in the hyperblocks stands for the set of instructions of block *T* which are now assigned a predicate, built from the conditional expression of the trailing branch in *I*. In contrast, *-E* denotes the instructions from block *E* which are assigned the negated conditional expression of the branch. Through these predicates, the instructions of *+T* and *-E* are no longer control dependent and thus can be moved into the block *I'*, which consists of the instructions of *I* without the trailing conditional branch. In general, the *ITJ (IT)* and the *IEJ (IE)* patterns are isomorphic. Nevertheless, different predicates have to be assigned to the conditional block, therefore these patterns are kept separated for an easier implementation of the transformation.

Example The example in Figure 5.29 shows assembly code for an *it–then–else–join*–pattern. The left side implements the if–pattern by using branches, the right side with predicated instructions.

A 'visual' evaluation already shows the effectiveness of this transformation. Detailed evaluation as shown in Table 5.10 has to consider code size, cycles when taken, and cycles when not taken. The results from this simple example are promising, but an un–constrained transformation may sometimes be too aggressive. Out–of–balance patterns or excessively large 'T' and 'E' blocks may impair

ITEJ	by branch	by predicate
code size	13	9
cycles taken	8	5
cycles not taken	11	5

Table 5.10: ITEJ Evaluation

the performance of the final assembly code: (1) Additional register pressure arises from the predicate expression, which has to be available at all instructions of 'T' and 'E'. (2) The additional predicates cause code size overhead. (3) Worst–case execution time may be increased due to resource contention from large 'T' and 'E' blocks. Therefore, the transformation has to be guided by proper estimation functions which will be introduced subsequently.

Algorithm The implementation of if–conversion is split into two phases. The first phase traverses the control flow graph and collects the candidate patterns for later transformation. Candidates where the 'T' or the 'E' block already contain an instruction of the BR class are discarded because the predicate instructions to be added are executed on the branch unit of the xDSPcore datapath. The second phase then transforms the remaining patterns. For each of the patterns, cycle count and code size estimation functions can be applied which judge if the transformation is performed or revoked. After each transformation, the control flow graph has to be updated. Figure 5.30 shows pseudo–code for the whole transformation.

While collecting the patterns and doing the actual transformation is straight forward, estimating the impact of the transformation in an early compilation phase (before instruction scheduling) is more complex. The next paragraphs will give explanations on the related topics.

Cycles The number of execution cycles for a code sequence I is determined either through the *critical path* of the *data dependence graph (DDG)* (see Section 5.2.2) or through the accumulated resource utilization of these instructions divided by the available resources per cycle. The maximum of both numbers gives a lower bound for the execution cycles. This estimation function is called e_{cyc} further on. Equation (5.9) shows how e_{cyc} is calculated.

```
procedure doIfConversion(t_function f, boolean do_constrained)
  t_cfg cfg=f.getControlFlowGraph();
  t_patterns patterns=collectIfPatterns(cfg);
  forall ( p ∈ patterns ) do
    int cycles_before=estimateCycles(p);
    t_pattern p'=doTransformation(p);
    int cycles_after=estimateCycles(p');
    if ( do_constrained ∧ cycles_after > cycles_before ) then
      f.revokeTransform(p);
    else
      f.commitTransform(p');
    endif
    cfg.update();
  endfor
end
```

Figure 5.30: If conversion algorithm

$$\begin{aligned} e_{cyc}(I) &= max(e_{ddg}(I), e_{res}(I)) \\ e_{ddg}(I) &= \ldots critical\ path\ of\ DDG\ for\ instructions\ I \\ e_{res}(I) &= max(\bigcup_{c \in C} e_{res}(I,c)) \\ e_{res}(I,c) &= \frac{\#\{i | i \in I \wedge i \in c\}}{c\ per\ cycle} \end{aligned} \tag{5.9}$$

C is the set of instruction classes $\{MOV, CMP, BR\}$, the sharp symbol means 'number of elements', and 'c per cycle' is the number of instructions of the particular class that can be executed in one execution cycle.

With this estimation function, the cycle count *before* transformation is approximated as shown in Equation (5.10) and memorized. In order to get the worst–case estimation, the maximum of either 'T' or 'E' is taken into account. Utilizing the same estimation function, the cycle count is again approximated for the transformed code sequence as shown in Equation (5.11). Depending on the current pattern and the resulting hyperblock, the corresponding right hand side of Equation (5.11) is used. These equations also reflect the fact, that the code is fully predictable now and that execution time analysis will give precise results. If the transformed code sequence yields an increased cycle estimation, the transformation can be revoked and the original code containing branches is restored.

$$cycles_{untransformed} = e_{cyc}(I) + \max(e_{cyc}(T), e_{cyc}(E)) + e_{cyc}(J) \tag{5.10}$$

$$cycles_{transformed} = \begin{cases} e_{cyc}(ITEJ) & \text{for the ITEJ pattern} \\ e_{cyc}(ITJ) & \text{for the ITJ pattern} \\ e_{cyc}(IEJ) & \text{for the IEJ pattern} \\ e_{cyc}(ITE) + e_{cyc}(J) & \text{for the ITE pattern} \\ e_{cyc}(IT) + e_{cyc}(J) & \text{for the IT pattern} \\ e_{cyc}(IE) + e_{cyc}(J) & \text{for the IE pattern} \end{cases} \tag{5.11}$$

Code size The overall code size of the pattern depends on several factors. A decrease comes from removing the branches and their possibly empty branch delay slots. The exact number depends on basic block order, type of pattern and number of architectural branch delay slots. In the example above, two branch instructions and four 'no-ops' in branch delay slots can be removed. An increase of code size occurs through the additionally needed predicate instructions in the execution bundles of the hyperblock. In the example above, two predicate instructions are necessary. This gives the overall reduction of code size by four instructions. From this it becomes apparent, that with larger patterns a considerable increase in code size may occur.

Nevertheless, sometimes it makes sense to permit transformation even if one of the constraints is violated. If code predictability and precise execution time analysis is required, then transforming patterns which potentially increase the number of execution cycles may be allowed. For small loop kernels where code size is negligible, it may be better to do the transformation and to accept the slight increase in code size. In the following, an extensive empirical evaluation of some applications is given.

Evaluation Evaluation has been done in two parts. The first part covers a qualitative analysis of typical applications in order to show the need for such an optimization. The second part covers quantitative data which show the impact of if–conversion on code size and execution cycles.

Qualitative evaluation is done by counting the occurrence of the particular patterns of Figure 5.28 within the benchmarks applications (see Figure 5.31). It turns out that two of the six patterns are dominating, namely IEJ and ITEJ. The remaining patterns occurred as well, but almost one magnitude less. From this, some quantitative impact from doing the transformation is expected.

$$B = \frac{\#(T)}{\#(T) + \#(E)} \tag{5.12}$$

Another qualitative figure which gives more insight into the structure of the patterns is the *pattern balance* (see Equation (5.12), where the sharp (#) symbol

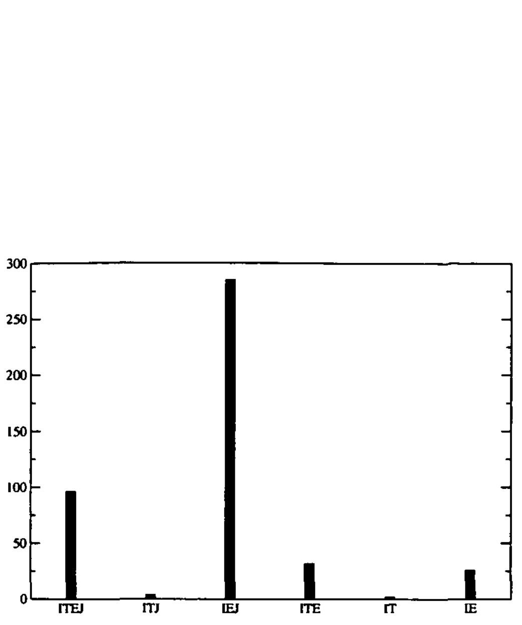


Figure 5.31: Occurrence of patterns

means 'number of instructions'). A balance of 0 indicates an empty T block, a balance of 1 indicates an empty E block. If the balance is 0.5 both T and E blocks have same size. A distribution of these balances is shown in Figure 5.32.

As expected, the dominating balance is 0. The balances of the ITEJ and ITE patterns are almost evenly distributed in a range between $[0.1, 0.8]$, with two little peeks at 0.5 and 0.72.

Quantitative evaluations are made by compiling (and simulating) the benchmarks several times with different if–conversion settings. The baseline is given when no transformation at all is applied. The next setup is an unconstrained transformation, where all patterns are transformed. The last setup is a transformation where cycle estimations are applied to the patterns. Patterns which potentially cause larger cycle counts are not transformed. Table 5.11 summarizes the results. The first row of each application shows the code size (in instruction words), the second row shows the execution cycles.

In general, a decrease of code size and execution cycles can be achieved. Only the DCT (discrete cosine transform) kernels show a slight increase of code size. What else can be seen, is that the cycle constrained transformation delivers almost identical results as the unconstrained one. Only g721 and Serpent show another behavior. In both cases code size is slightly bigger than in the unconstrained version, but in both cases the number of execution cycles has been decreased. On the one hand, this indicates that the estimation function works properly, otherwise different behavior would be seen. On the other hand, the transformation increases instruction level parallelism. This results from removing branches (and branch delay slots) from the code which yields a larger scope (the *hyperblock*) for instruction scheduling. Data independent instructions from all included blocks can be scheduled and executed concurrently and thus resource utilization is increased. Additionally, removing branch instructions makes the code more predictable through better approaching the *single path paradigm*. This yields tighter bounds of WCET analysis and thus may influence task scheduling or system design decisions.

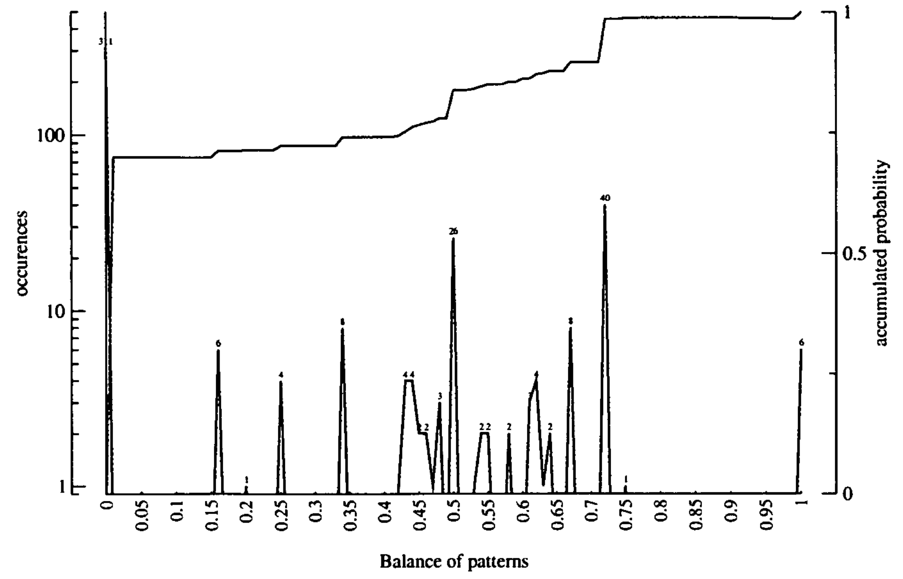


Figure 5.32: Balance of patterns

benchmark	no if–conversion	unconstrained	cycle constrained
ADPCM (codesize)	419	379 (-9.6%)	379 (-9.6%)
ADPCM (cycles)	1246494	915599 (-26.5%)	915599 (-26.5%)
Blowfish (codesize)	392	389 (-0.8%)	389 (-0.8%)
Blowfish (cycles)	1451252	1314593 (-9.5%)	1314593 (-9.5%)
CMAC (codesize)	3663	3622 (-1.2%)	3622 (-1.2%)
CMAC (cycles)	1570582	1515424 (-3.6%)	1515424 (-3.6%)
g721 (codesize)	2042	1980 (-3.1%)	1998 (-2.2%)
g721 (cycles)	19092960	18567846 (-2.8%)	18373751 (-3.8%)
ghs (codesize)	2699	2620 (-3%)	2620 (-3%)
ghs (cycles)	8836	7281 (-17.6%)	7281 (-17.6%)
DCT 32 (codesize)	683	698 (+2%)	698 (+2%)
DCT 32 (cycles)	4349	3875 (-10.9%)	3875 (-10.9%)
DCT 8x8 (codesize)	474	496 (+4.6%)	496 (+4.6%)
DCT 8x8 (cycles)	114851	111049 (-3.4%)	111049 (-3.4%)
Rijndael (codesize)	4218	4101 (-2.8%)	4101 (-2.8%)
Rijndael (cycles)	244354	241349 (-1.3%)	241349 (-1.3%)
Serpent (codesize)	5963	5735 (-3.9%)	5741 (-3.8%)
Serpent (cycles)	2316423	1788365 (-22.8%)	1749994 (-24.5%)

Table 5.11: Quantitative results of if–conversion

5.2.2 Instruction scheduling

Section 3.1.3 briefly surveyed instruction scheduling and the fundamental techniques. The concept of *list scheduling* based on a *data dependence graph* was shortly explained. Some differences between scheduling for super–scalar and VLIW architectures have been pointed out. This section will present the implementation of a local instruction scheduler for the xDSPcore architecture, based on the scheduler presented in [91].

Original implementation

The original scheduler was written at a time when register allocation was not completed and no evaluation environment (benchmark applications and simulator) has been available. Main focus was achieving minimal schedule length, which was accomplished by a list scheduler with limited backtracking. In the following, a short description of this algorithm is given.

The first step is building a data dependence graph from the list of instructions in the basic block. True–, anti–, output–, and "negative" control dependencies (for pushing instructions in the branch delay slots) are analyzed and encoded in the graph. A true dependency occurs between an instruction that defines an operand and the instruction which uses this operand (RAW, read after write). An anti–dependency is the opposite of a true dependency and occurs between an instruction that reads an operand and a later instruction that (re)defines the same operand (WAR, write after read). An output dependence occurs between two instructions that write to the same destination (WAW, write after write). Finally, control dependencies occur the execution of an instruction depends on the execution of another prior instruction. Control dependencies are inserted for all instruction of the BR class. Instructions that are causally before the BR instruction have an control dependency edge to this one with a negative delay ($-num\_branch\_delays$). This allows exploitation of branch delay slots by moving the branch before data independent instructions. Successor instructions of the branch instructions have a control dependency edge with a delay of $num\_branch\_delays + 1$. The maximum path length of an instruction to the sink node of the graph is used as a *scheduling priority* for this instruction. Edges in the graph are annotated with a value which encodes the *minimal* distance of an instruction to the succeeding one. Each instruction holds a *resource vector* which encodes hardware constraints. In conjunction with an *available resources* vector (for one bundle), hardware contention checks can be performed as shown in Equation (5.13).

$$\sum_{i \in B} \vec{r_i} \leq \vec{A} \tag{5.13}$$

B is the bundle which is checked, $\vec{r_i}$ the resource vector of instruction i, and

$$\vec{A} = \begin{pmatrix} 2 \\ 2 \\ 2 \\ 2 \\ 4 \\ 1 \end{pmatrix}$$

$$\vec{r}_{ldao} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \vec{r}_{(C)\,add} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \vec{r}_{asr} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \vec{r}_{dmac} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 2 \\ 0 \\ 0 \end{pmatrix}, \vec{r}_{brc} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 4 \\ 1 \end{pmatrix}, \ldots$$

ldao ...load absolute offset, (*C*) *add* ...predicated addition, *asr* ...arithmetic shift right, *dmac* ...dual multiply–accumulate, *brc* ...branch conditional

Figure 5.33: Available resources and sample resource vectors

$\vec{A}$ the available resources per bundle. Resource vectors consist of six components:

$$\vec{r} = \begin{pmatrix} MOV \\ CMP \\ SHIFT \\ MULT \\ PRED \\ BR \end{pmatrix}$$

MOV, CMP, and BR correspond to the basic parallel functional units. $SHIFT$ and $MULT$ are used to model restrictions on shift and multiplication instructions, and $PRED$ is used for proper resource contention checking of the branch unit in case of predicated instructions. The available resources $\vec{A}$ for the reference architecture and some resource vectors for different instructions are depicted in Figure 5.33.

The execution bundles are then built step–by–step. In each step, a new empty bundle is created and appended to the current schedule. Then, a root node of the data dependence graph is selected and put in the schedule. This is done by recursively moving the instruction from the current bundle towards its *earliest cycle* as far as possible. If none of the bundles in this *scheduling window* can take the instruction due to resource contention, this candidate is discarded and left in the graph. If a slot is found, the instruction is removed from the graph. Selecting nodes and placing them into the schedule is repeated until no more nodes can be removed in this scheduling step. By applying back–tracking, those nodes from the graph which yield the highest accumulated scheduling priority are finally selected and put into the schedule (selections which combine equivalent predicates are favored in order to reduce code size overhead from predicated instructions). Then the algorithm advances to the next step and repeats this process until all nodes of the graph are removed. Figures 5.34 and 5.35 show pseudo code for this algorithm.

The upward code–motion in this strategy exploits *instruction level parallelism (ILP)* in the code sequence to its maximum. Potential resource contention conflicts of later instructions are avoided, and also branch delay slots are filled with instructions. Nevertheless, this strategy has some drawbacks which will be outlined in the following.

Drawbacks The major drawback of this scheduling strategy is artificial register pressure. Due to the upward code–motion scheme, it often happens that low-priority root nodes are placed very early in the schedule. The corresponding consuming instruction is placed late because of data dependencies, and therefore a long live–range which potentially conflicts with many others is created. Placing the root node in a later cycle of the scheduling window does not extend the schedule, but reduces register pressure.

This situation is already adverse for the regular instruction sequence, but it gets almost pathological when spilling instructions are involved. The intention of spilling is reducing register pressure. Unfortunately, the spill loads are root nodes in the data dependence graph and thus suffer from the situation explained above. Register pressure is not reduced at all and then leads to *cascading spills.*

Another problem of this strategy is the creation of unbalanced schedules. Regions with maximum instruction level parallelism (ILP) alternate with regions of weak ILP. As elaborated by Yun in [61], this leads to increased step– and peak–power dissipation. A more even distribution of the instructions over the schedule (without elongation) overcomes this problem. Additionally, design space parameters like the required parallel functional units are also mainly influenced by the quality of the scheduling algorithm.

Problem analysis

The drawbacks outlined above appeared un-predictable. Therefore an extensive analysis of the fundamental problem was done. The data dependence graphs of the benchmark applications were investigated for quantitative and structural information. This should help at drawing conclusions how to modify the existing algorithm or design a new one that does not show such behavior.

The first and obviously most important figure (besides number of instructions) is the critical path length of the data dependence graph. It gives a lower bound for the minimal schedule length. The ratio $\overline{p} = \frac{n}{c}$ (n is the number of instructions, c the critical path length) is a metric for the *average* instruction level parallelism.

```
procedure scheduleBlock(t_block basic_block)
begin
   t_ddg ddg=createDDG(basic_block.getInstructionList());
   t_schedule schedule=new t_schedule();
   integer cycle_count=1;
   while ( $\neg$ ddg.isEmpty() ) do
      t_bundle bundle=new t_bundle(cycle_count++);
      schedule.append(bundle);
      t_nodes current=new t_nodes();
      t_nodes best=new t_nodes();
      ddg.resetHandledFlags();
      findBestScheduleSet(ddg,bundle,current,best);
      forall (t_node n $\in$ best) do
         putInSchedule(bundle,n,ddg);
      endfor
   endwhile
   block.setSchedule(schedule);
end
procedure findBestScheduleSet(
   t_ddg ddg, t_bundle bundle, t_nodes current, t_nodes best)
begin
   forall (t_node root $\in$ ddg.getRootNodes() ) do
      if ( $\neg$ root.isHandled() $\wedge$ canTakeInstruction(bundle,root) ) then
         putInSchedule(bundle,root,ddg);
         current.add(root);
         if (getPriority(current) > getPriority(best) ) then
            copy(best,current);
         endif
         findBestScheduleSet(ddg,bundle,current,best);
         current.remove(root);
         removeFromSchedule(bundle,root,ddg);
         root.markAsHandled();
      endif
   endfor
end
function getPriority(t_nodes nodes):integer
begin
   integer result=0;
   forall ( t_node n $\in$ nodes) do result += n.getPriority(); endfor
   return result;
end
```

Figure 5.34: Original scheduling algorithm

```
function canTakeInstruction(t_bundle bundle, t_node node):boolean
begin
  if (bundle.getCycleNumber() $\geq$ node.getEarliestCycle() ) then
    if (canTakeInstruction(bundle.getPreviousBundle(),node)) then
      return true;
    else
      if (bundle.hasFreeSlotFor(node)) then
        return true;
      endif
    endif
  endif
  return false;
end
function putInSchedule(t_bundle bundle, t_node node, t_ddg ddg):boolean
begin
  if (bundle.getCycleNumber() $\geq$ node.getEarliestCycle() ) then
    if (putInSchedule(bundle.getPreviousBundle(),node,ddg)) then
      return true;
    else
      if (bundle.hasFreeSlotFor(node)) then
        bundle.add(node);
        ddg.remove(node);
        ddg.updateEarliestCycle(node,bundle.getCycleNumber());
        return true;
      endif
    endif
  endif
  return false;
end
procedure removeFromSchedule(t_bundle bundle, t_node node, t_ddg ddg)
begin
  if (node $\in$ bundle) then
    bundle.remove(node);
    ddg.reinsert(node);
  else
    removeFromSchedule(bundle.getPreviousBundle(),node,ddg);
  endif
end
```

Figure 5.35: Original scheduling algorithm

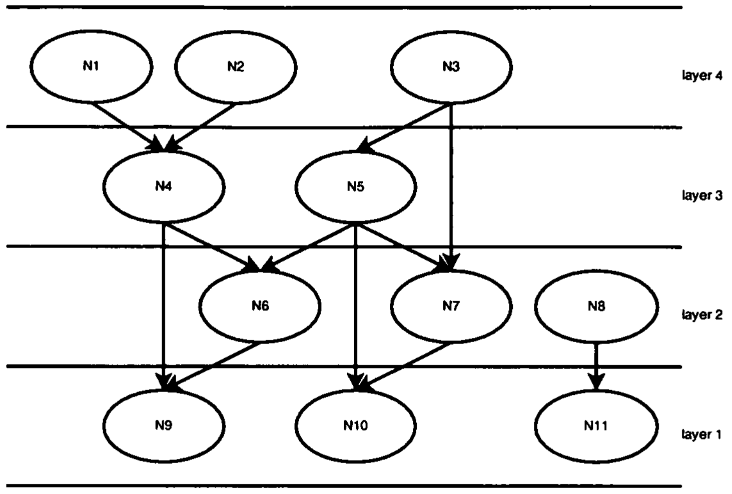


Figure 5.36: Layering of a data dependence graph

	min	max	$\overline{x}$	$\overline{s}$	σ
n	1	147	6.51	4.49	8.54
c	1	78	5.21	2.90	5.28
p	0.33	6	1.33	0.61	0.77
m	1	11	2.12	1.14	1.44
a	1	268	12.88	11.37	21.15
u	0.18	1	0.63	0.18	0.22

Table 5.12: Basic data dependence graph statistics

Dividing the graph in layers of equal path lengths gives feedback on the *maximum* instruction level parallelism m (defined by the layer with most instructions). Figure 5.36 shows an example of how layering a data dependence graph is done. The example assumes all edges having latency 1.

The individual components of the accumulated resource vectors show the demand for either memory (MOV), computational (CMP), or branch (BR) resources in the code sequences. Finally, the ratio between n and the *spanning area* $a = m * c$ indicate average resource utilization $u = \frac{n}{a}$ of the layered graph.

The benchmark applications deliver a large set of over 1600 graphs, covering a wide variety. Only condensed data which give most insight are presented. Table 5.12 gives minimum, maximum, averages, statistical spread and standard deviation for the key figures explained above. A large dynamic range for n,c and a can be observed. Individual graphs have up to 147 instructions and critical paths of up to 78 cycles. The average ILP of the graphs ranges up to 6 instructions per cycle, and the maximum even reaches 11. The maximum square area of

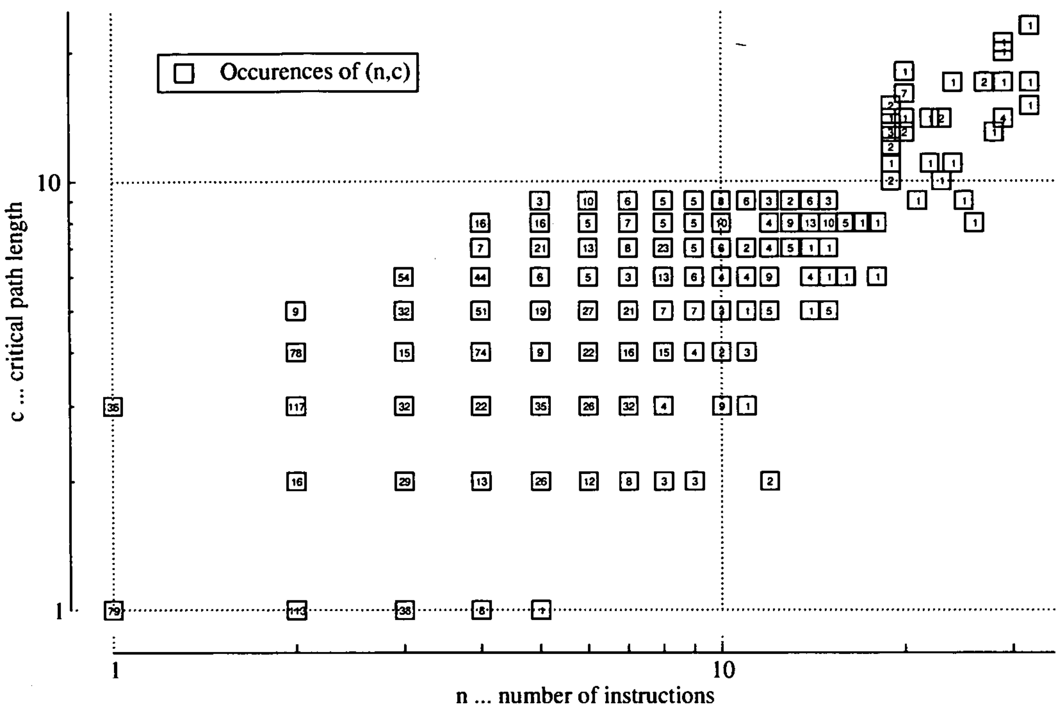


Figure 5.37: Histogram of (n,c) pairs

an individual graph is 268 *instructions* × *cycles*. In contrast, the minimum of resource utilization is at only 18%.

Figure 5.37 depicts a histogram of (n, c)–occurrences for the range $(1, 1)$ to $(35, 25)$. From this, a relatively strong correlation between n and c can be deduced, but also a considerable spread of critical paths for fixed length sequences is observed. Figure 5.38 gives a similar histogram of (m, c)–occurrences for the range $(1, 1)$ to $(11, 30)$. From this it can be seen that the structures of the graphs are spread over a large spectrum. On the one hand, there are wide graphs with short critical paths, on the other hand also narrow graphs with long critical paths are present. This impacts the choice of the scheduling strategy.

	MOV	CMP	BR
Σ	2067 (21%)	6567 (66%)	1369 (14%)
max	73	73	10
$\overline{x}$	1.26	4	0.83
$\overline{x_p}$	15 %	62 %	23 %

Table 5.13: Statistics for MOV, CMP, BR

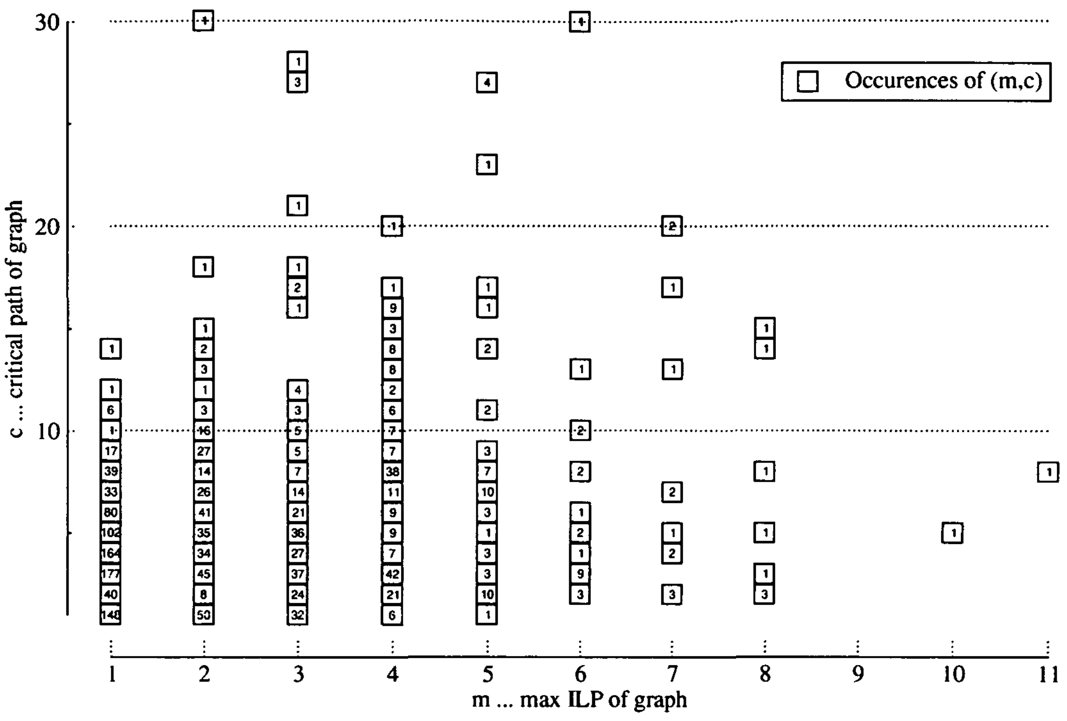


Figure 5.38: Histogram of (m,c) pairs

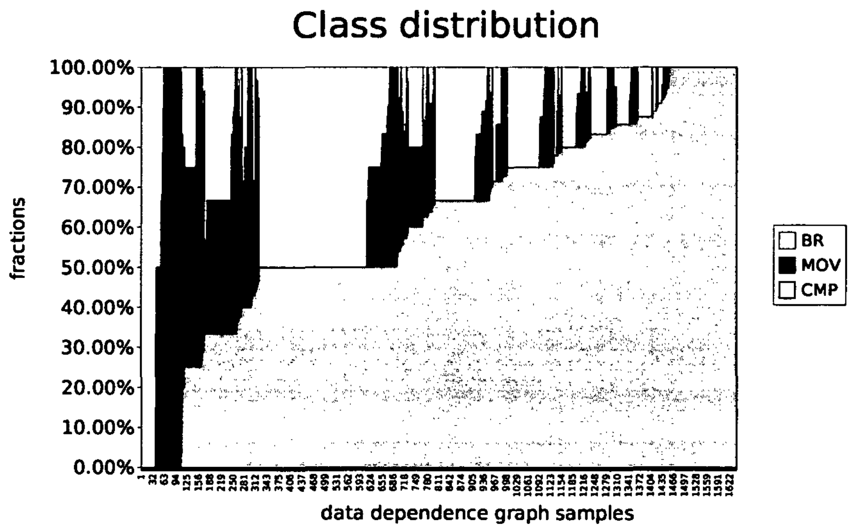


Figure 5.39: Distribution of class fractions

	regr.slope	correlation
c	0.55	0.86
CMP	0.58	0.89
MOV	0.4	0.79
BR	0.02	0.22

Table 5.14: Regression analysis for n

Figure 5.39 shows the fractions of different instruction classes on an individual graph basis. It can be observed that CMP instructions are clearly predominant. An identical picture can be seen in Table 5.13. Here, the total number, maximum, averages, and average of percentages for MOV, CMP, and BR instructions are compared.

Finally, Table 5.14 shows the linear regression and the correlation between n and MOV, CMP, BR, c, respectively. A strong correlation between n and c, n and CMP, but also n and MOV can be observed. The number of branch instructions does not correlate much with the length of the instruction sequence.

Refinements

Analyzing the data dependence graphs and the internal decision process of the scheduling algorithm showed, that in most of the problematic cases a relatively wide graph was to be scheduled. These are not constrained by critical path, but rather through resource contention. Therefore, using the path length and the backtracking mechanism for guiding scheduling decisions leads to the drawbacks mentioned earlier. Instead of this breadth–first–like strategy, a rather depth–first exploitation of the dependence graph would be better. Nevertheless, for narrow graphs the breadth–first strategy is better, so a mixed strategy which does self–adaptation has to be found.

Primary focus of the refinements was reducing register pressure and improving the placement of spilling instructions (especially the loads). The basic scheme of the algorithm is the same as before, but more attention is paid on which nodes are selected in each scheduling step.

Root nodes are classified into such nodes with common direct successors, and such nodes without. Figure 5.40 shows an example for common successor root nodes, with $R1$, $R2$, and $R3$ having common successors, and $R4$ and $R5$ without common successors. In a first intermediate step, only nodes from the common successor sub–set are considered and selected for scheduling. If such nodes can be moved upwards, then it is likely that the common successor can also be scheduled in the same step. This keeps the distance between producer and consumer instruction smaller (and thus reduces register pressure). In the example above, scheduling and moving up $R1$, $R2$ and $R3$ allows immediate scheduling

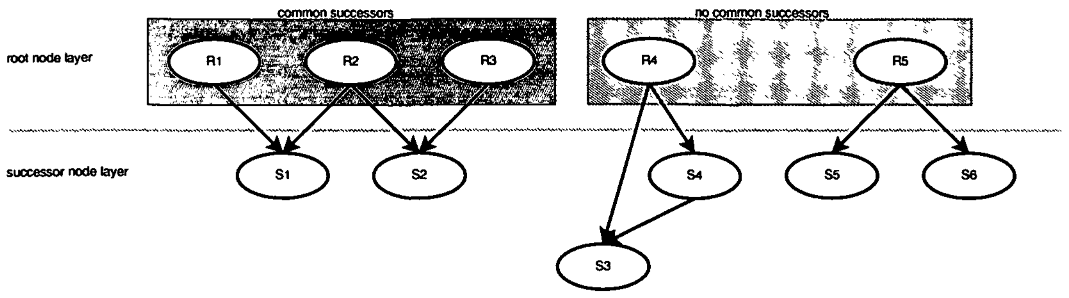


Figure 5.40: Common successor root nodes

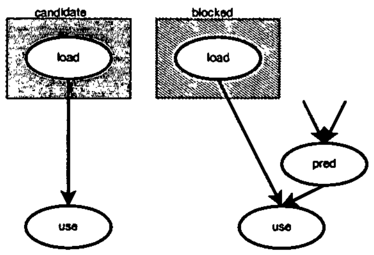


Figure 5.41: Spilling loads selection

of $S1$ and $S2$, thus consuming all values produced by the selected root nodes. In contrast, selecting $R4$ and $R5$, produces values which have to be consumed by four successor instructions. Those are less likely to be scheduled, which thus extends the live ranges of the values and increases register pressure.

For the second intermediate step, a resource vector based cycle estimation of the remaining instructions in the data dependence graph is done. If the maximum scheduling priority of the remaining root nodes is greater than half of this estimation, a rather narrow graph is indicated. This graph is mostly constrained from its critical path, therefore instructions from the current candidate set are further selected. Otherwise, the graph is constrained from resource contention and scheduling proceeds to the next bundle immediately. By doing this, instructions that are not necessarily on the critical path, but already in the root node set are held back until they get on the critical path.

Load instructions of a spilled live–range are scheduled by doing a one–step look–ahead. If the successor instruction (the use of the spilled live–range) has no further predecessors, the load is considered as a candidate for scheduling. If the load finally gets scheduled, the scheduling priority of the successor is increased in order to assure that it is selected in the next scheduling step. Figure 5.41 depicts examples for blocked and non–blocked spill–loads.

The refinements on the algorithm have been made ad–hoc and are still far from optimal. The intention was to get a better idea how a mixed strategy has to look like and what kind of "decision guides" can be applied to obtain optimal schedules. Optimal schedules should have the following properties:

- minimal length
- minimal local register pressure
- balanced ILP
- minimal overhead from predicated instructions
- exploiting branch delay slots

Through the refinements explained above, the original scheduling algorithm was brought closer to these rationales, but there is still some lack in algorithm design. Balancing has not yet been tackled enough and for some cases, a look–ahead is applied in order to draw better decisions. Nevertheless, some improvements could be achieved through only slight modifications. This fact indicates the sensitivity of the overall code quality to the particular scheduling strategy. Table 5.15 shows a comparison of the original and the refined scheduling strategy. Some benchmarks give the same results in both strategies, some are improved, but unfortunately some were also worsened. A slight overall cycle improvement of less than 1% was achieved.

benchmark	original	refined
ADPCM (codesize)	379	377 (-0.6%)
ADPCM (cycles)	915599	889954 (-2.9%)
Blowfish (codesize)	389	389 (±0%)
Blowfish (cycles)	1314593	1314593 (±0%)
CMAC (codesize)	3621	3624 (+0.1%)
CMAC (cycles)	1512614	1499817 (-0.9%)
g721 (codesize)	1980	1982 (+0.1%)
g721 (cycles)	18567846	18548244 (-0.2%)
ghs (codesize)	2619	2620 (+0%)
ghs (cycles)	7271	7271 (±0%)
DCT 32 (codesize)	706	707 (+0.1%)
DCT 32 (cycles)	3894	3868 (-0.7%)
DCT 8x8 (codesize)	496	494 (-0.5%)
DCT 8x8 (cycles)	111049	109754 (-1.2%)
Rijndael (codesize)	4101	4101 (±0%)
Rijndael (cycles)	241349	241369 (±0%)
Serpent (codesize)	5733	5743 (+0.1%)
Serpent (cycles)	1749997	1746191 (-0.3%)
Total (codesize)	20024	20037 (+0%)
Total (cycles)	24424212	24361061 (-0.3%)

Table 5.15: Evaluation of scheduling modification

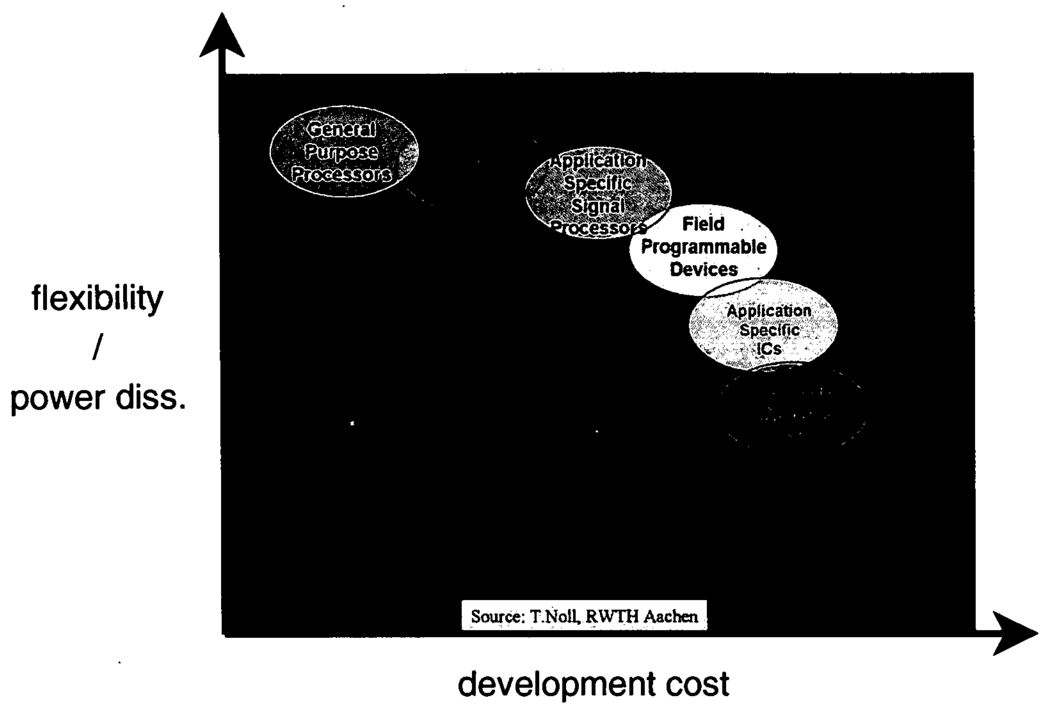

General Purpose Processors
Application Specific Signal Processors
Field Programmable Devices
Application Specific ICs
Source: T.Noll, RWTH Aachen
flexibility / power diss.
development cost

Figure 5.42: HW/SW tradeoff

5.2.3 Design Space Exploration

One of the major decisions during design of an embedded system is a balanced hardware/software partitioning. Implementation in dedicated hardware gives great efficiency in terms of chip area, execution time, and power dissipation. On the other hand these solutions are lacking in flexibility. This causes long turn–around time for variants of the same product or for adaptations due to changes in standards. Software solutions based on high level languages increase the flexibility and simplify customer specific adaption of an application. On the contrary, these solutions often lack in efficiency in terms of chip area and power dissipation. Therefore an efficient solution is based somewhere in the middle, having those parts flexible, which can help to modify the product to changing requirements and implement the remaining parts in dedicated logic. Figure 5.42 shows a qualitative picture of this tradeoff.

Design space exploration is an attempt to close the gap between hardware and software solutions. The idea is "tailoring" a hardware platform that meets the requirements of the application. This is done by analyzing the application on high–level language and specification level and quantifying its demands. The analysis data then are used for guiding system design decisions, whether parts can be implemented efficiently in software (and how the processor architecture has to look like), or which parts have to be implemented directly in hardware.

The remaining part of this section focuses on the design space exploration for sub–tasks of applications which are implemented on a software programmable embedded digital signal processor. Scalability factors will be identified and discussed regarding their implications on application software and on hardware design.

Existing concepts like LISA [92] or ArchC [93] are based on automatic generation of an application specific architecture from a behavioral description. The inherent problem of these approaches is that automatic generation of *optimizing* compilers for such architectures is not yet solved to a satisfying extent. A broader discussion of these approaches and their drawbacks is found in [5, 76].

The approach of the xDSPcore attempts to solve the problem of design space exploration by considering compiler requirements already in architectural and micro–architectural design (Figure 4.1). A RISC–like architecture makes the implementation of an optimizing compiler feasible. Instead of exploring the design space on a low behavioral level, several important scalability factors on a higher level have been elaborated and investigated for suitability and feasibility. This high–level approach limits the design space to a manageable dimension, but still allows efficient description of hardware components and optimal code generation for each of the architectural instances.

The design space exploration methodology for the xDSPcore architecture is called *DSPxPlore* and was introduced in [3]. Central point of this methodology is an *architecture configuration file.* This file contains a complete description of the *instruction set architecture*, a high–level description of architectural features, documentation aspects, and also software and firmware aspects like calling conventions, startup code, and interrupt handler information. The toolchain (compiler, linker, simulator) is designed in such a way that it does automatic adaption to the changes in this configuration file. Application analysis is then done by compiling the source code and simulating the program in a cycle–true simulator. Static and dynamic information of the program is provided and has to be analyzed by the system designer. Decisions on architectural modifications have to be drawn and upstreamed to the configuration file. Then a new analysis cycle can start again. When an optimal configuration is found, the exploration loop can stop and hardware description and documentation can be generated according to the current configuration). Figure 5.43 briefly depicts the DSPxPlore work flow. A detailed explanation of the configuration file is presented in [94].

Design space

The xDSPcore architecture was designed to be scalable in several parameters. While the direct effects of modifying a parameter can be captured very easily, there are often indirect effects caused by inter–dependencies with other parameters. These interdependencies and side–effects make the n–dimensional design

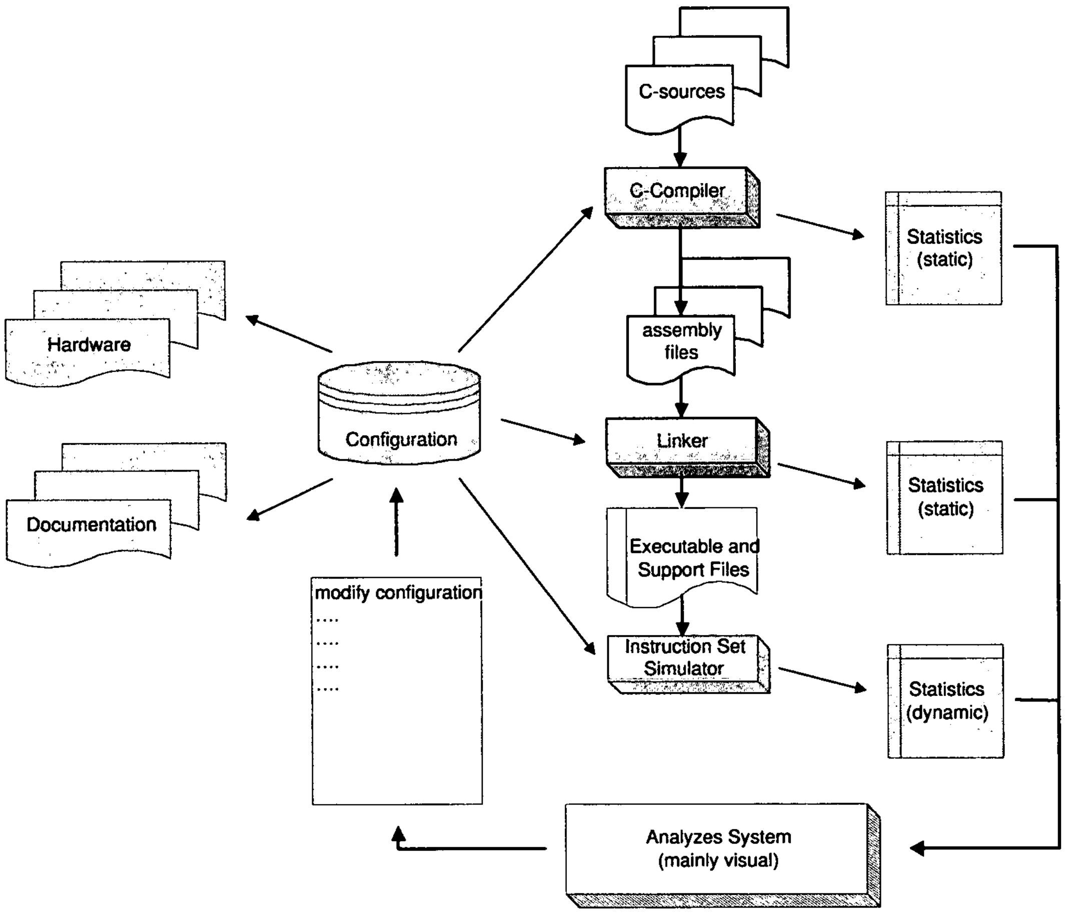


Figure 5.43: Overview of DSPxPlore

space non–uniform and non–linear, which itself makes design space exploration complicated. An approach for guiding design space exploration using *Pareto* curves was presented by Agosta in [95].

In DSPxPlore, architectural refinements are made semi–automatic under guidance of the system engineer who is assisted by the analysis data. The following parameters of xDSPcore can be varied within some reasonable limits:

- register file size
- number of functional units
- pipeline depth
- instruction buffer size
- depth of hardware loop stack
- binary instruction encoding

The following paragraphs will explain each of these parameters and their implications in detail.

Register file size In RISC architectures, the register file has a central role. It has to provide space for program variables needed during a computation (local variables, temporaries). If it is too small for a particular task, these variables have to be spilled to memory. Therefore a large register file is preferable from the view of the application program. On the other hand, enlarging the register file results in increasing chip area, increasing power dissipation within the register file, and limiting the reachable clock frequency. Further, the binary encoding of instructions needs more bit for addressing a larger number of registers. These factors push for smaller register files. Therefore a reasonable tradeoff between providing many registers and dealing with spilling overhead has to be found.

Number of functional units The number of parallel functional units defines the peak performance of the architecture. Depending on average and maximum instruction level parallelism of the application and on the characteristics of the code (memory–centric or computation–centric), a suitable number of functional units has to be found. For the application code it would be best to have as many functional units as the maximum ILP demands. On the other hand, numerous functional units increase hardware complexity and chip area. Due to the gap between maximum and average ILP, usage of these resources is weak and a reasonable tradeoff has to be made.

Pipeline depth Increasing the pipeline depth is a method for increasing the maximum clock frequency of a processor, but strongly impacts application software. One problem arises from the delayed branches in VLIW architectures. An increased number of pipeline stages before the branch stage increases the number of branch delay slots as well. Additional stages in the execution phase of the pipeline relax timing constraints on the memory and register file ports, but lengthen the latencies of data dependent instructions. Depending on application code characteristics, the performance increase through a higher clock frequency may then be nullified or even lead to performance degradation.

Instruction buffer size The instruction buffer has a central role during execution of loops. The instructions of a loop body are loaded into the buffer during the first loop iteration. For the remaining iterations, fetching is stalled and no further accesses to the instruction memory occur. Obviously, this can be done only when the entire loop body fits into the instruction buffer. Otherwise fetching is done as in regular sequential execution. A program benefits most, if the buffer is big enough for all loop bodies, but overly large buffers increase chip area and circuit complexity. This increases power dissipation of the core components which might nullify the savings from reducing the instruction memory accesses.

Hardware loop stack Status information of zero–overhead hardware loops is stored in a special hardware loop stack. Providing an arbitrary level of nested hardware loops is preferable from the perspective of the application program, but each additional level causes overhead. Chip area is increased due to the additional storage requirements. Stack handling has to be implemented in hardware and demands additional circuitry for each level. This results in additional chip area and increased power dissipation during hardware loop handling. Further, the entire loop stack status has to be saved to memory during interrupt handling. This causes additional cycles in the interrupt service routines and extends task switching time of real–time operating systems.

Binary instruction encoding The binary encoding of the instruction set has great impact on code size. It depends on various parameters like number of different instructions, number and types of registers to be addressed, occurrence of different *immediate* operands, and number of functional units. Besides those factors, some requirements from the low–level hardware implementation of the instruction decoder and from the xLIW programming model have to be met.

In general, small native instruction words have to be favored, but reducing the word width forces a larger sub–set of the instructions to be encoded via long words. If an application requires many instructions out of this sub–set, the code size reduction from the smaller native word is nullified through the increased usage of long words.

5.3 Summary & Additional optimizations

The previous sections have covered the main building blocks of the xDSPcore compiler backend (*if–conversion, instruction scheduling, register allocation*). A post–pass optimization method for optimizing switching on the instruction bus was presented as well.

One interesting point is splitting register allocation and coalescing. The main motivation for this is the interdependence with instruction scheduling. Any instruction from the code sequence before register allocation has to be put into a valid slot of the schedule. When some of the instructions are removed after register allocation, the schedule then might be sub–optimal. Remaining instructions which are scheduled later due to resource contention or data dependencies with the removed ones possibly can be moved upwards, yielding a better overall schedule.

Additionally, the importance of the different optimizations is somewhat different in super–scalar or VLIW architectures. For super–scalar architectures, most effort has been spent on register allocation, while instruction scheduling was considered as minor important (but indeed as important). In VLIW architectures, it is necessary to focus on scheduling first, but then also apply good register allocation. When the schedule is bad, register allocation will yield bad results as well.

Besides those major optimizations, some other peep–hole optimizations have been implemented. The first is a post–instruction selection optimization which eliminates superfluous branches. These arise at basic blocks where control–flow falls through to the linear successor in the block list. Due to the control–flow graph modification during if–conversion, this optimization had to be delayed to a post–if–conversion phase of the backend.

Another important optimization is the generation of auto–modifying memory accesses (increment or decrement of address before or after the access). This optimization searches for pairs of instructions which operate on the same address register within a basic block. If such pairs of *(access,modification)* are found they are combined into one single instruction.

The limited scope of if–conversion cannot handle all (conditional) branches in a function. Some branches still remain, and maybe have empty branch delay slots. A post–pass optimization which moves data–independent instructions of the branch target into the empty delay slots has been implemented. Finally, the remaining branches with empty delay slots can make use of so–called *non–delayed branches*. The delay slots for those branches do not have to be encoded explicitly, but are hidden by the hardware. Thus, there is no gain in performance, but the code size is reduced a little further.

Chapter 6

Conclusions

Power dissipation and energy consumption are of vital interest in embedded systems. Low power design techniques are necessary on all levels of the design process. Due to the rising amount of software programmable components in System–on–Chip and System–in–Package solutions, the demand on low power software techniques even gets more important. On compiler level, optimizations focus on reducing execution time, memory accesses, and switching activities on memory buses. Especially for embedded digital signal processors, design space exploration is needed for closing the efficiency gap to dedicated hardwired circuits, while still providing flexibility in application.

This thesis contributes extensions of register allocation for irregular architectures and a code optimization for reducing switching activities on the instruction memory bus. Minor contributions are made on if conversion, instruction scheduling, and on design space exploration.

Global register allocation is performed in order to find a good mapping of the program variables to the processor registers. This mapping has to be carried out in such a way, that register resource utilization is saturated and that memory accesses due to spilling of variables are minimized. Any overhead in spilling causes additional and thus avoidable energy consumption due to the costly memory accesses. The common approach of graph coloring is well suited for regular architectures with a large general purpose register file. Applying this method on irregular architectures needs several extensions. For the xDSPcore architecture, register allocation has to deal with a banked register file (address and data register bank) and shared registers. Un–orthogonal instructions cause special interference constraints on program variables, and liveness analysis has to consider the guarded execution model.
This thesis presents solutions for these irregularities by contributing a *predicate enhanced liveness analysis*, a *split interference graph* for the banked register file, and *partial interference information* on symbolic registers. Predicated liveness analysis for *register leaves* is based on modified set operators of conventional liveness analysis and on a *predicate registry* which keeps track of predicate se-

mantics and does predicate simplification by the Quine–McCluskey algorithm. Bank restrictions are handled during instruction selection. Symbolic registers are annotated with a bank information tag, which is used during register allocation to select registers of the correct bank. Shared registers are modeled by a *weighted interference graph*, where a node weight encodes the required register file resources of a symbolic register. Additionally, the edges in this interference graph are augmented with a partial interference information field. This allows modeling of precise interference constraints for un–orthogonal instructions.

A considerably different approach is a *partitioned boolean quadratic problem* based optimal register allocator. Cost functions and matrices are used to model interference and architectural constraints as well as spilling costs. A PBQP problem solver then delivers spilling decisions and register assignments. The thesis contributes cost functions and matrices for the xDSPcore architecture, and a solution to the phase ordering and phase coupling problem of instruction scheduling with register allocation and coalescing. A comparison of the graph–coloring and the PBQP–based approach show the superiority of the PBQP–based approach in the achieved code quality.

Instruction fetch in VLIW architectures is a major factor of power dissipation. This thesis contributes an optimization to reduce the switching activities on the instruction memory bus. Measures for optimization are (1) permutation of the instructions within single execution bundles, and (2) swapping the source operands of commutative operations. To take full advantage of the xLIW programming model, the solution has to be able to cope with *unaligned execution bundles* which even may cross *fetch word boundaries*. In contrast to existing graph based optimization approaches, the optimization algorithms contributed in this thesis can fulfill those demands.

The optimization is based upon an objective function that models a *global Hamming distance* of all fetch words of a function. A hybrid algorithm of dynamic programming and genetic evolution is applied. The dynamic programming part solves a local sub–problem at scopes of basic blocks, while the genetic part selects those local solutions which yield the overall minimum of the global Hamming distance. Local solutions include all optimal solutions which have different *fetch word borders* and thus do not restrict the solution space.

If conversion is an important transformation for control code dominated programs. On VLIW architectures, these programs suffer from the inherent problem of unused branch delay slots. Through if conversion, branch instructions and therefore the delay slots can be removed. This yields larger scheduling scopes and effectively reduces execution time of the programs. This thesis contributes estimation functions for *execution time* and for *code size.* Applying these estimations avoids the potential negative impact of if conversion on code quality.

Exploiting instruction level parallelism is a crucial task in VLIW architectures. Instruction scheduling techniques based on a data dependence graph and the list scheduling algorithm are known to be good methods for exploiting the full range of parallelism. Nevertheless, the used heuristics cause some problems like excessive register pressure or unbalanced resource utilization. This thesis presents an extensive problem analysis which serves as a basis for some refinements on scheduling decisions. These refinements result in reducing register pressure while still achieving the same amount of parallelism.

A high–level language based design space exploration methodology called *DSPxPlore* has been presented in this thesis. It is built upon a processor architecture with several scalable micro–architectural features. A unique configuration file contains a complete description of an architectural instance and is crucial for having a consistent and inter–operable toolchain. After design space exploration is completed, hardware description, documentation, and files containing firmware and operating system stubs are generated.

Bibliography

[1] Ulrich Hirnschrott, Andreas Krall, and Bernhard Scholz. Graph Coloring vs. Optimal Register Allocation for Optimizing Compilers. In *JMLC*, pages 202–213, 2003.

[2] Ulrich Hirnschrott and Andreas Krall. VLIW Operation Refinement for Reducing Energy Consumption. In *Proceedings of the International Symposium on System-On-Chip (SOC) 2003*, November 2003.

[3] Christian Panis, Ulrich Hirnschrott, Gunther Laure, Wolfgang Lazian, and Jari Nurmi. DSPxPlore: Design Space Exploration Methodology for an Embedded DSP Core. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 876–883. ACM Press, 2004.

[4] Christian Panis, Ulrich Hirnschrott, Andreas Krall, Gunther Laure, Wolfgang Lazian, and Jari Nurmi. FSEL - Selective Predicated Execution for a Configurable DSP Core. In *ISVLSI*, pages 317–320, 2004.

[5] Andreas Krall, Ulrich Hirnschrott, Christian Panis, and Ivan Pryanishnikov. xDSPcore: A Compiler-Based Configurable Digital Signal Processor. *IEEE Micro*, 24(4):67–78, July/August 2004.

[6] Christian Panis, Ulrich Hirnschrott, Andreas Krall, Stefan Farfeleder, Gunther Laure, Wolfgang Lazian, and Jari Nurmi. A Scalable DSP Core for SoC Applications. In *Proceedings of the International Symposium on System-On-Chip (SOC) 2004*, November 2004.

[7] A. Chandrakasan, S. Sheng, and R. Brodersen. Low–Power CMOS Digital Design, 1992.

[8] Stefan Steinke, Rüdiger Schwarz, Lars Wehmeyer, and Peter Marwedel. Low Power Code Generation for a RISC Processor by Register Pipelining. Technical Report 754, University of Dortmund, Dortmund, March 2001.

[9] Raimund Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis.* PhD thesis, Technische Universität Wien, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, May 2003.

[10] Martin Farach and Vincenzo 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.

[11] David W. Wall. Global Register Allocation at Link Time. In *Proceedings of the 1986 SIGPLAN symposium on Compiler contruction*, pages 264–275. ACM Press, 1986.

[12] Fred C. Chow. Minimizing Register Usage Penalty at Procedure Calls. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94, Atlanta, Georgia, USA, June 1988.

[13] Steven M. Kurlander and Charles N. Fischer. Minimum Cost Interprocedural Register Allocation. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 230–241, New York, NY, USA, 1996. ACM Press.

[14] G. J. Chaitin. Register Allocation and Spilling via Graph Coloring. In *SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, 1982.

[15] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring Heuristics for Register Allocation. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 275–284, 1989.

[16] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 311–321, 1992.

[17] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[18] Lal George and Andrew W. Appel. Iterated Register Coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[19] Jinpyo Park and Soo-Mook Moon. Optimistic Register Coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, 2004.

[20] Frederick Chow and John Hennessy. Register Allocation by Priority-based Coloring. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 222–232. ACM, 1984.

[21] Preston Briggs, Keith D. Cooper, and Linda Torczon. Coloring Register Pairs. *ACM Letters on Programming Languages and Systems*, 1(1):3–13, March 1992.

[22] Michael D. Smith and Glenn Holloway. Graph-Coloring Register Allocation for Irregular Architectures. Technical report, Harvard University, 2000.

[23] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A Generalized Algorithm for Graph–Coloring Register Allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288. ACM Press, 2004.

[24] Johan Runeson and Sven-Olof Nyström. Retargetable graph-coloring register allocation for irregular architectures. In *SCOPES*, pages 240–254, 2003.

[25] David W. Goodwin and Kent D. Wilken. Optimal and Near–Optimal Global Register Allocations using 0/1 Integer Programming. *Softw. Pract. Exper.*, 26(8):929–965, 1996.

[26] Timothy Kong and Kent D. Wilken. Precise register allocation for irregular architectures. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 297–307. IEEE Computer Society Press, 1998.

[27] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 177–189. ACM, January 1983.

[28] Joseph C. H. Park and Michael S. Schlansker. On Predicated Execution. Technical Report HPL-91-58, Hewlett Packard Laboratories, May 1991.

[29] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 45–54, 1992.

[30] David I. August, Wen mei W. Hwu, and Scott A. Mahlke. A Framework for Balancing Control Flow and Predication. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 92–103. IEEE Computer Society, 1997.

[31] Alexandre E. Eichenberger and Edward S. Davidson. Register Allocation for Predicated Code. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 180–191, Ann Arbor, Michigan, November 29–December 1 1995. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[32] David M. Gillies, Dz-ching Roy Ju, Richard Johnson, and Michael Schlansker. Global Predicate Analysis and its Application to Register Allocation. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 114–125, Paris, December 2–4 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[33] Richard Johnson and Michael Schlansker. Analysis Techniques for Predicated Code. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 100–113, Paris, December 2–4 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[34] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Computers*, 30(7):478–490, 1981.

[35] David Bernstein and Michael Rodeh. Global Instruction Scheduling for Superscalar Machines. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 241–255. ACM Press, 1991.

[36] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Warter, and Wen mei W. Hwu. IMPACT: An Architectural Framework for Multiple–Instruction–Issue Processors. In *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*, pages 266–275. ACM Press, 1991.

[37] Scott A. Mahlke, William Y. Chen, Wen mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. Sentinel Scheduling for VLIW and Superscalar Processors. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 238–247. ACM Press, 1992.

[38] David Gregg. *Compilation Techniques for Instruction Level Parallelism in the Presence of Loops and Branches.* PhD thesis, Technische Universität Wien, 2001.

[39] Philip B. Gibbons and Steven S. Muchnick. Efficient Instruction Scheduling for a Pipelined Architecture. In *Proceedings of the 1986 SIGPLAN symposium on Compiler contruction*, pages 11–16. ACM Press, 1986.

[40] H. S. Warren, Jr. Instruction Scheduling for the IBM RISC System/6000 Processor. *IBM J. Res. Dev.*, 34(1):85–92, 1990.

[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] B. Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74. ACM Press, 1994.

[43] Vasanth Bala and Norman Rubin. Efficient Instruction Scheduling Using Finite State Automata. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 46–56. IEEE, November 1995.

[44] Jack Liu and Fred Chow. A Near–Optimal Instruction Scheduler for a Tightly Constrained, Variable Instruction Set Embedded Processor. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, pages 9–18. ACM Press, 2002.

[45] Markus Lorenz, Rainer Leupers, Peter Marwedel, Thorsten Dräger, and Gerhard P. Fettweis. Low–Energy DSP Code Generation Using a Genetic Algorithm. In *ICCD*, Austin, September 2001.

[46] Markus Lorenz, Lars Wehmeyer, and Thorsten Dräger. Energy Aware Compilation for DSPs with SIMD Instructions. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 94–101. ACM Press, 2002.

[47] Peter Marwedel, Stefan Steinke, and Lars Wehmeyer. Compilation Techniques for Energy-, Code–size-, and Run–time–efficient Embedded Software. Technical report, University of Dortmund, 2001.

[48] L. Wehmeyer, M.K. Jain, S. Steinke, P. Marwedel, and M. Balakrishnan. Analysis of the Influence of Register File Size on Energy Consumption, Code Size and Execution Time. In *IEEE TCAD*, volume 20 of *11*, November 2001.

[49] Jui-Ming Chang and Massoud Pedram. Register Allocation and Binding for Low Power. In *Proceedings of the 32nd ACM/IEEE conference on Design automation conference*, pages 29–35. ACM Press, June 12–16 1995.

[50] Catherine H. Gebotys. Low Energy Memory and Register Allocation using Network Flow. In *Proceedings of the 34th annual conference on Design automation conference*, pages 435–440. ACM Press, 1997.

[51] Yumin Zhang, Xiaobo (Sharon) Hu, and Danny Z. Chen. Global Register Allocation for Minimizing Energy Consumption. In *Proceedings 1999 international symposium on Low power electronics and design*, pages 100–102. ACM Press, 1999.

[52] Yumin Zhang, Xiaobo (Sharon) Hu, and Danny Z. Chen. Efficient Global Register Allocation for Minimizing Energy Consumption. *ACM SIGPLAN Notices*, 37(4):42–53, 2002.

[53] Mike Tien-Chien Lee, Vivek Tiwari, Sharad Malik, and Masahiro Fujita. Power Analysis and Low–power Scheduling Techniques for Embedded DSP Software. In *Proceedings of the eighth international symposium on System synthesis*, pages 110–115. ACM Press, 1995.

[54] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. Instruction Level Power Analysis and Optimization of Software. *Journal of VLSI Signal Processing*, pages 1–18, 1996.

[55] Mark C. Toburen, Thomas M. Conte, and Matt Reilly. Instruction Scheduling for Low Power Dissipation in High Performance Processors. In *Proceedings of the Power Driven Micro-architecture Workshop at ISCA'98*. ACM, June 1998.

[56] Chingren Lee, Jenq Kuen Lee, and TingTing Hwang. Compiler Optimization on Instruction Scheduling for Low Power. In *Proceedings of the 13th conference on International Symposium on System Synthesis*, pages 55–60. ACM Press, 2000.

[57] A. Parikh, Mahmut T. Kandemir, N. Vijaykrishnan, and Mary Jane Irwin. Instruction Scheduling Based on Energy and Performance Constraints. In *Annual Workshop on VLSI (WVLSI'00)*. IEEE, 2000.

[58] Dongkun Shin and Jihong Kim. An Operation Rearrangement Technique for Low-Power VLIW Instruction Fetch. In *Proceedings of Workshop on Complexity-Effective Design*, June 2000.

[59] Dongkun Shin, Jihong Kim, and Naehyuck Chang. An Operation Rearrangement Technique for Power Optimization in VLIW Instruction Fetch. In *Proceedings of Design, Automation and Test in Europe, Date'01*, pages 809–817. ACM, March 2001.

[60] Kyu won Choi and Abhijit Chatterjee. Efficient Instruction–level Optimization Methodology for Low-power Embedded Systems. In *Proceedings of the international symposium on Systems synthesis*, pages 147–152. ACM Press, 2001.

[61] Han-Saem Yun and Jihong Kim. Power–aware Modulo Scheduling for High–performance VLIW Processors. In *Proceedings of the 2001 international symposium on Low power electronics and design*, pages 40–45. ACM Press, 2001.

[62] Luca Benini and Giovanni De Micheli. System–level Power Optimization: Techniques and Tools. In *Proceedings 1999 international symposium on Low power electronics and design*, pages 288–293. ACM Press, 1999.

[63] Wei-Chung Cheng and Massoud Pedram. Power–optimal Encoding for DRAM Address Bus (poster session). In *Proceedings of the 2000 international symposium on Low power electronics and design*, pages 250–252. ACM Press, 2000.

[64] Wei-Chung Cheng and Massoud Pedram. Low Power Techniques for Address Encoding and Memory Allocation. In *Proceedings of the conference on Asia South Pacific Design Automation Conference*, pages 245–250. ACM Press, 2001.

[65] Armita Peymandoust, Tajana Simunic, and Giovanni De Micheli. Low Power Embedded Software Optimization using Symbolic Algebra. In *Proceedings of the Design Automation and Test in Europe*, pages 1052–1058, 2002.

[66] Tohru Ishihara and Hiroto Yasuura. Voltage Scheduling Problem for Dynamically Variable Voltage Processors. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED-98)*, pages 197–202, New York, August 10–12 1998. ACM Press.

[67] Takanori Okuma, Tohru Ishihara, and Hiroto Yasuura. Software Energy Reduction Techniques for Variable Voltage Processors. *Design and Test of Computers*, 18(2):31–41, March-April 2001.

[68] Chung-Hsing Hsu, Ulrich Kremer, and Michael Hsiao. Compiler–Directed Dynamic Voltage/Frequency Scheduling for Energy Reduction in Microprocessors. In *International Symposium on Low Power Electronics and Design (ISPLED'01)*, August 2001.

[69] Dongkun Shin, Jihong Kim, and Seongsoo Lee. Intra–Task Voltage Scheduling for Low–Energy, Hard Real–Time Applications. *Design and Test of Computers*, 18(2):20–30, March-April 2001.

[70] H. Saputra, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and J.S. Hu. Energy–Conscious Compilation Based On Voltage Scaling. In *LCTES SCOPES 2002*, 2002.

[71] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A Framework for Architectural–level Power Analysis and Optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94. ACM Press, 2000.

[72] W. Ye., N. Vijaykrishnan, Mahmut T. Kandemir, and Mary Jane Irwin. The Design and Use of SimplePower: A Cycle–Accurate Energy Estimation Tool. In *Proceedings of the 37th Conference on Design Automation (DAC-00)*, pages 340–345, NY, 2000. ACM/IEEE.

[73] H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. A Framework for Energy Estimation of VLIW Architectures. In *IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD '01)*, pages 40–45, Washington - Brussels - Tokyo, September 2001. IEEE.

[74] Sheayun Lee, Andreas Ermedahl, Sang Lyul Min, and Naehyuck Chang. An Accurate Instruction–Level Energy Consumption Model for Embedded RISC Processors. In Seongsoo Hong and Santosh Pande, editors, *LCTES'01 Workshop on Languages, Compilers and Tools for Embedded Systems*, Snowbird, June 2001. ACM.

[75] Amit Sinha and Anantha P. Chandrakasan. JouleTrack - A Web Based Tool for Software Energy Profiling. In *Proceedings of the 38th Conference on Design Automation (DAC-01)*, pages 220–225, NY, June 18–22 2001. ACM/IEEE.

[76] Christian Panis. *Scalable DSP Core Architecture Addressing Compiler Requirements.* PhD thesis, Tampere Univerity of Technology, 2004.

[77] J.L.Hennessy and D.A. Patterson. *Computer Architecture. A Quantitative Approach.* Morgan Kaufmann Publishers, 1996.

[78] A. Shoham P. Lapsley J. Bier and E.A. Lee. DSP Processor Fundamentals, Architectures and Features. *IEEE Press*, 1997.

[79] Christian Panis, Raimund Leitner, and Herbert Grünbacher. xLIW – A Scalable Long Instruction Word. In *ISCAS 2003*, Bangkok, Thailand, 2003.

[80] Christian Panis, Michael Bramberger, Herbert Grünbacher, and Jari Nurmi. A Scalable Instruction Buffer for a Configurable DSP Core. In *ESSCIRC*, Estoril, Portugal, 2003.

[81] Christian Panis, Gunther Laure, Wolfgang Lazian, Herbert Grünbacher, and Jari Nurmi. A Branch File for a Configurable DSP core. In *VLSI-03*, Las Vegas, Nevada, USA, 2003.

[82] Scott A. Mahlke, Richard E. Hank, James E. McCormack, David I. August, and Wen mei W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In *Proc. of the 22nd Annual International Symposium on Computer Architecture (22nd ISCA'95) ACM SIGARCH Computer Architecture News*, pages 138–149, Santa Margherita, Italy, June 1995.

[83] D. Sima, T. Fountain, and P. Kacsuk. *Advanced Computer Architectures: A Design Space Approach.* Addison Wesley Publishing Company, 1997.

[84] J.E. Smith. A Study of Branch Prediction Strategies. In *Proceedings 8th ISCA*, 1981.

[85] J.K.F Lee and A.J. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *Computer*, 17(1):6–22, 1984.

[86] T.-Y. Yeh and Y.N Patt. Alternative Implementations of Two–Level adaptive Branch Predictions. In *Proceedings 19th ISCA*, 1992.

[87] D.N. Pnevmatikos and G.S. Soshi. Guarded Execution and Branch Prediction in Dynamic ILP Processors. In *Proceedings 21st ISCA*, 1994.

[88] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, Cambridge, UK, 1998.

[89] Bernhard Scholz and Erik Eckstein. Register Allocation for Irregular Architectures. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 139–148. ACM Press, 2002.

[90] Christian Trödhandl and Peter Puschner. Support of the Single–Path Paradigm and Elimination of Side–Effects in Single–Path Code. Technical report, Vienna University of Technology, Real–Time Systems Group, December 2003.

[91] Karl Vögler. A DSP C–Compiler. Master's thesis, Vienna University of Technology, Vienna, January 2002.

[92] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. In *Proceedings of the 36th Design Automation Conference (DAC' 99)*, pages 933–938, New York, June 1999. Association for Computing Machinery.

[93] *www.archc.org.*

[94] Gunther Laure and Wolfgang Lazian. A Configurable Component Based Framework for Simulating Digital Signal Processors. Master's thesis, TU Graz, 2005.

[95] Giovanni Agosta, Gianluca Palermo, and Cristina Silvano. Multi–objective Co–exploration of Source Code Transformations and Design Space Architectures for Low-power Embedded Systems. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 891–896, New York, NY, USA, 2004. ACM Press.