Program Analyses and Transformations Exploiting Parallelism in DSPs

Björn Franke

University of Edinburgh
Department of Computer Science
September 14, 2000
Contents

1 Introduction ........................................... 8
   1.1 Overview ........................................... 10

2 Related Work ......................................... 11
   2.1 DSP Architectures ................................... 11
       2.1.1 Single Processor DSP .......................... 12
       2.1.2 Integrated Multiprocessor DSP ............... 18
   2.2 DSP Applications .................................... 22
       2.2.1 Theoretical Background ......................... 22
       2.2.2 Real Applications .............................. 28
       2.2.3 Benchmarks ..................................... 31
   2.3 DSP Compilers ....................................... 34
       2.3.1 Code Generation ................................. 35
       2.3.2 Address Code Generation ....................... 36
       2.3.3 C specific problems ......................... 36
       2.3.4 Scalar Data Flow Analysis ..................... 37
       2.3.5 Array Data Dependence Test .................. 41

3 Infrastructure ........................................ 47
   3.1 Philips TriMedia TM-1 ............................ 47
   3.2 Octave ............................................ 49
       3.2.1 Graph Template Library ....................... 50
### 3.3 TriMedia Software Development Environment  50
### 3.4 DSPstone  52

### 4 Pointer Clean-up Conversion  54
  4.1 Motivation  54
  4.2 Related Work  57
  4.3 Basic Idea  58
  4.4 Assumptions and restrictions  58
  4.5 Algorithm  61
  4.6 Extensions  67
    4.6.1 Arrays passed as parameters of functions  67
    4.6.2 Perfect Loop Nests  68
    4.6.3 General Loop Nests  69
    4.6.4 Conditional branches  71
    4.6.5 Multi-dimensional arrays  72

### 5 Software-Pipelining  74
  5.1 Motivation  74
  5.2 Basics  77
  5.3 Minimum Initiation Interval  78
    5.3.1 Resource-constrained Minimum Initiation Interval  78
    5.3.2 Recurrence-constrained Minimum Initiation Interval  80
  5.4 Scheduling a New Loop Kernel  80
  5.5 Extensions  81
    5.5.1 Software-Pipelining of Loops with Conditional Branches  83
  5.6 Implemented Software-Pipelining Algorithm  84
6 Empirical Results

6.1 Tests and Goals .................................................. 88
6.2 Observations and Results ...................................... 89
  6.2.1 Software-Pipelining .......................................... 89
  6.2.2 Pointer Clean-up Conversion .......................... 93
  6.2.3 Other Tests .................................................... 93
6.3 Interpretation ..................................................... 95

7 Conclusions and Outlook .......................................... 97

Bibliography .......................................................... 99
List of Figures

2.1 Multimedia SIMD instructions [49] ......................... 14
2.2 Address Generation Unit (AGU) with Address, Modify and Length
    Register Files ............................................. 17
2.3 TI TMS320C80 DSP Architecture [62] ......................... 19
2.4 Architecture of an individual DSP of the TI TMS320C80 [62]  21
2.5 Sampling of a sine wave ..................................... 23
2.6 Sampling as a multiplication .................................. 23
2.7 Spectrum of a signal before and after sampling ............... 24
2.8 Structure of an FIR filter ................................... 27
2.9 Serial implementation of an FIR filter ......................... 27
2.10 MPEG video stream data structure [23] ....................... 29
2.11 Block diagram of an MPEG-2 encoder [23] ................... 30
2.12 Block diagram of an MPEG-2 decoder [23] ................... 30
2.13 Expression \((a \times b) + (c \times d)\) and instruction patterns [40] 35
2.14 Array Data Dependences in a Perfect Loop Nest .............. 43
3.1 TriMedia TM-1100 Architecture [49] ......................... 48
3.2 TM-1100 Issue Slots, Functional Units and Latencies [49] .... 49
3.3 Convolution program of the DSPstone benchmark suite ....... 53
4.1 Original pointer-based loop ................................... 55
4.2 Loop after conversion of pointer-based accesses into explicit array
    accesses ..................................................... 56
4.3 Simple, structured loop ............................................ 59
4.4 Legal pointer initialisations ................................. 60
4.5 Function calls changing pointer arguments .......... 60
4.6 First pass of clean-up conversion and values of the map data structure ............................................. 65
4.7 Second pass of clean-up conversion, values of the offsetMap data structure and affine index functions ........ 66
4.8 Example of pointer clean-up conversion of arrays passed as function arguments ................................ 67
4.9 Pointer clean-up conversion of perfect loop nests .... 68
4.10 General structure of perfect loop nest with pointer-based array access ..................................................... 68
4.11 Perfect loop nest after pointer clean-up conversion .. 69
4.12 General loop nest .................................................. 70
4.13 Example of pointer clean-up conversion of general loop nests ................................................................. 71
4.14 Example of pointer clean-up conversion of conditional branches . 72
4.15 Example of pointer clean-up conversion of a 2-dimensional array 73

5.1 Sequential Loop and Software-Pipelined Loop ........ 75
5.2 Software-Pipelining of a simple loop ..................... 76
5.3 Data Dependence Graph and schedule with resource-constraints 79
5.4 Example of Modulo Scheduling .............................. 82
5.5 Example of if-conversion ......................................... 83

6.1 Results for biquad_nb_sections, n_real_updates, convolution and fir ......................................................... 90
6.2 Results for different loop ranges of n_complex_updates .... 92
6.3 Results of pointer clean-up conversion for matrix1, matrix2, lms and matrix3 ............................................. 94
Chapter 1

Introduction

In 1995 the New York Times has estimated that the average American comes into contact with about 60 microprocessors every day, and some of today’s top-level cars include at least 100 microprocessors [40]. Most of these microprocessors are part of Embedded Systems rather than of familiar Desktop PCs. Although ubiquitous, frequently these Embedded Systems are not even realised as present. The typical application domains of Embedded Systems are Automatic Control and Digital Signal Processing (DSP) which is the area this report focuses on.

Digital Signal Processing is the main task of many video, audio, graphics and communication systems. Although on the first sight these fields may appear as very different in their nature, they have a lot in common. They are all dominated by high performance requirements and rapid changes of specifications. Not at least the quickly growing number of Multimedia applications with their excessive resource consumptions and frequently changing standards underpin this statement. The required flexibility in the DSP domain can often only be fulfilled by programmable processors rather than application-specific integrated circuits (ASICs). Programmable processors with specialised features for Digital Signal Processing are called Digital Signal Processors (DSPs). Often real-time constraints of the applications and the short time-to-market make software development for DSPs a very hard task. Other obstructions to program development for DSPs are the strict limits of code and data size imposed by fixed-size on-chip memories, instruction sets with specialised instructions, complex addressing modes and the need for low power dissipation in battery-driven devices.

In order to meet the performance constraints of the application, many DSP programs are implemented in assembly languages. Although the required performance can be achieved with this approach, it contradicts the goals of flexibility, portability, low cost and short development times. In general-purpose computing the well-known solution to meet these goals is the use of High-Level Languages (HLL) and Compilers. Unfortunately, the quality of code generated
by many available compilers for DSPs is very poor [31]. The gap between the commonly used programming language C and the underlying DSP architecture is quite big and cannot be bridged sufficiently by ordinary compilers. The importance and the growth of the DSP market on the one hand side, and the immature state of software development tools, in particular compilers, on the other side, motivate the research into new and more powerful optimisations for DSP compilers.

One way of optimising DSP applications is to exploit their parallelism. Different DSPs show very different levels of available hardware parallelism. Fine-grain parallelism is available on instruction level in many DSPs, whereas Multiprocessor DSPs provide additional Coarse-grain parallelism. The highly challenging task for the compiler is to identify the parallelism in the application and to map it onto the architecture such that performance is maximised and the other constraints are not violated.

DSP applications have been found to be numerically intensive, spending most of their their processing time in loop code with very high degrees of Instruction Level Parallelism (ILP) [59]. A transformation technique that is suitable for exploiting ILP in loops is Software-Pipelining. Software-Pipelining seeks to overlap consecutive loop iterations in such a way that instructions from different loop iterations are in state of execution simultaneously. Usually, software-pipelining is a technique that is applied in the backend of a compiler at a very machine-specific level of intermediate representation. Because by far not all commercially available compilers perform software-pipelining, it is desirable to integrate it. Since the source codes of commercial DSP compilers are usually not available, software-pipelining cannot easily integrated into such compilers. But there are possibilities to perform this loop transformation as a Source-to-Source transformation before the actual compiler is used. Although not no specific machine knowledge is available at source level, software-pipelining at that level can make the instruction scheduling easier and more efficient for the DSP manufacturers’ compilers. During this project a software-pipelining approach on source level has been implemented and evaluated. Although empirical results showed that for simple loops there is little or no improvement, for some more complex loops performance gains of up to 50% have been observed.

Because early DSP compilers were not able to generate efficient addressing code, programmers tend to use pointers for array traversals. The presence of pointers prevents many program analyses to be applied, which in turn prevents many optimisations of more recent DSP compilers. During this project an algorithm for the conversion of a certain class of pointer-based array accesses into explicit array accesses has been developed and implemented. This Pointer Clean-up Conversion is applied in combination with the software-pipelining approach and alone. Since many of the built-in optimisations of the comparatively sophisticated TriMedia compiler that is used for generating object code become applicable after pointer clean-up conversion, performance improvements of up to 68% can be achieved. On average, these figures are lower but still significant.
1.1 Overview

Chapter 2 gives an overview of DSP Architectures, Applications and Compilers and introduces fundamental concepts. In chapter 4 a Pointer Clean-up Conversion is described which supports the Software-Pipelining algorithm, but additionally it can applied successfully in combination with other optimizing transformations and their analyses. Software-Pipelining and its implementation as a part of the Octave compiler is explained in chapter 5. In chapter 6 empirical results achieved with Software-Pipelining and Pointer Clean-up Conversion are documented. Finally, the conclusions of this work are summarised in chapter 7.
Chapter 2

Related Work

Digital Signal Processing is a wide field of research. Unlike some other Computer Science related areas, it cannot be studied as separate subjects corresponding to applications, hardware and compiler technology. Because of the strong interactions of the requirements of the applications, idiosyncratic features of the hardware and the resulting hard task of the compilers to map the applications onto the hardware efficiently, it is obvious to think of DSP systems rather than of single components.

The following sections present fundamentals of DSP Architectures, Applications and Compilers characterising problems and possible attempts to solutions. Because of the huge quantity of available publications on the topic, a selection of subjects with special consideration of the main contribution of this thesis has been made rather than giving a comprehensive survey.

2.1 DSP Architectures

Since the first specialised DSP architectures became commercially available in the early 1980s, a large number of manufacturers has developed an even larger variety of DSPs. Nowadays, DSPs for very different fields of application and widely varying requirements are offered. The only common requirements are the demand for high performance at typical DSP applications (see section 2.2) and low power consumption at the same time [55].

Due to the diversified nature of the applications, several different approaches of DSP architectures have been constructed in order to fulfil the specific requirements. Although the trend to ever-increasing clock rates as it is known from the domain of general purpose CPUs can be observed in the world of DSPs, too, this way of achieving the required performance has its distinctive drawbacks. The power consumption of CMOS-based switching circuits is directly and linearly dependent on the frequency of the clock. Therefore, clock rates must be kept low in order fulfil the design goal of low power consumption which is in
some applications at least as important as high performance. Another strategy
to meet the performance criteria is to execute as many tasks as possible at the
same time – without increasing the clock rate. Parallelisation is the key to
trade off performance and power consumption.

In the following sections two highly parallel concepts of DSP are introduced
that can tackle high-performance – often real-time – constraints imposed by
many multimedia applications. The first approach (TriMedia TM-1) is the use
of a single DSP with a very high level of available Instruction Level Parallelism
(ILP). Rau [52] gives following statement to ILP processors:

An instruction-level parallel (ILP) processor is a parallel pro-
cessor in which the unit of computation, for which decisions such
as scheduling and synchronization are made, is the individual op-
eration, e.g., an add, multiply, load or store. It is not necessary
that any or all of these decisions be made during the execution of
the program; they could be made at compile time. For instance,
with VLIW processors (…), all these decisions are made during
compilation.

The second approach (TMS320C80) shows the integration of several indepen-
dent DSPs together with a sophisticated interconnection-network into a single
chip.

Low-power low-performance DSPs are not the subject of this work and are
left out from this survey just as Multiprocessor DSPs consisting of network-
memory-coupled “ordinary” DSPs. For more information on these subjects the
reader is referred to Lapsley [26] and Koch [25], respectively.

2.1.1 Single Processor DSP

This section introduces single processor DSP (SP-DSP) architectures and their
characteristic features. Register Sets, Arithmetic Logic Units, Instruction Level
Parallelism and its support by the Instruction Set, as well as Address Generation
Units and Memory Organisation are covered in this survey. A specific example
of a SP-DSP – which was used for running simulations during this project –
will be explained in more detail in chapter 3.1.

Register Sets

Among available SP-DSP two different organisations of register sets are popular.
On the one hand heterogeneous registers sets can be found, i.e. registers are
distributed and are integral parts of the data paths. On the other hand, SP-
DSP with homogeneous register sets become more and more dominant and
these register sets are organised as registers files in a similar fashion as known
from RISC architectures. Homogeneous register sets are usually bigger than their heterogeneous counterparts and registers are freely available rather than having specific assignments to functional units like adders and multipliers etc.

**Heterogeneous Register Sets** are tightly coupled to functional units of the DSP and its data path. This kind of structural organisation of register sets support common and frequently used operations of DSP applications. Many common used computations can be done efficiently. In a variety of DSP applications, e.g. digital filters (see 2.2.1), more or less the same operations are repeated again and again. These instruction sequences do not benefit from a general register set architecture with random access opportunities. Hence, adaption of the register set to the application characteristics not only serve the purpose of increasing the efficiency, but also of decreasing complexity, power consumption and costs. An example of a DSP with a heterogeneous register is the Analog Devices ADSP-21xx Family [5].

**Homogeneous Register Sets** support the requirements of more and more complex applications that cause problems with very application specific heterogeneous register sets. The application domain of DSP has become very broad and requirements as well as typical computations can vary widely. Thus, a domain-specific design of an DSP architecture can be a disadvantage, because of its inflexibility to adapt to new domains. On the other hand, heterogeneous register sets challenge assembly programmers as well as compilers. Compiler generated code for machines with heterogeneous register sets usually does not perform as good as manually tuned code. Generally available registers, i.e. registers without fixed assignment to functional units, make code optimisation much easier for existing register allocators and code generators. Therefore, virtually all modern high-performance DSPs have homogeneous register sets. Examples of DSPs with homogeneous registers sets are the Philips TriMedia TM-1 and the Texas Instruments TMS320C80 (see 3.1 and 2.1.2, respectively).

**Arithmetic Logic Units**

*Arithmetic Logic Units (ALU)* of DSPs are tailored to the needs of Multimedia and Digital Signal Processing applications. Because arithmetic computations are the dominant part of such applications, their efficient execution is a major design goal. In order to achieve a high efficiency, many operations are implemented in hardware (short latencies) and *Data Paths* are duplicated (high throughput).

Among the most “popular” operations implemented in hardware in DSP data paths is the *Multiply-Accumulate (MAC)* operation. A Multiply-Accumulate is a 3-address-operation for accumulating a sum of products. It efficiently supports the computation of sums of the form $\sum_{i=0}^{N} a_i \times b_i$. Together with *Zero-Overhead*
Loops initialised to the loop range \( N \), the computation of finite product-sums becomes very efficient. The importance of such sums becomes more apparent after some introduction in Digital Signal Processing Theory (see 2.2.1). A MAC operation \( \text{MAC} \ acc, \ op1, \ op2 \) uses three registers as operands. One of the registers is used as an accumulator, the others are operands corresponding to the terms to be multiplied. The product is added to the value stored in the accumulator register. Often this operation is implemented in such a way that it only takes a single cycle to finish.

Encoding and decoding of video/audio streams take up a considerable amount of CPU time [23]. Because these operations are key Multimedia operations, special hardware solutions can often be found in DSPs. Hardware support ranges from special bit-field instructions implemented with barrel-rotators and masking registers to dedicated functional units like Variable Length Decoders that are relief the actual CPU from all bit-stream manipulations.

A further idiosyncrasy of DSPs (and Multimedia Extensions of General-Purpose Processors) are Single Instruction Multiple Data (SIMD) operations. The idea is to split up wide registers into smaller segments capable of holding smaller data types like bytes or half-words and to apply arithmetical operations on all segments in parallel. Figure 2.1 shows an example of the \( \text{UME8UU} \) operation of the Philips Trimedia TM-1 processor. Two 32-bit registers serve as source registers. Both registers are divided into four parts each containing a byte value. The operation subtracts corresponding operands before the product of the intermediate differences is computed. The final result is then stored in the destination register. SIMD operations are a way to exploit Data Level Parallelism and are a simple form of vector operations that can usually only can be found in usually considerably larger vector computers.

**UME8UU:** Sum of absolute values of unsigned 8-bit differences

![Diagram of UME8UU operation](image)

The \( \text{ume8ee} \) operation, commonly used for motion estimation in video compression, implements 11 simple operations in one Trimedia special op.

Figure 2.1: Multimedia SIMD instructions [49]
2.1. DSP Architectures

Architecture Model

Two different approaches of exploiting Instruction Level Parallelism are common among modern processors. Whereas many general-purpose processors use Super-scalar Architectures and Dynamic Instruction Scheduling, DSPs rely heavily on Very Large Instruction Word (VLIW) Architectures and Static Instruction Scheduling, i.e. scheduling is done entirely by the compiler.

VLIW architectures have wide instruction words that contain several fields for controlling different functional units at a time. The difference to typical superscalar architectures is the total control of utilisation of the functional units by the program that is generated by the compilers. Rather than having a control unit that co-ordinates the different functional units depending on the stream of instruction to execute, VLIW architectures rely on a compiler to co-ordinate their functional units already at compile time. No dynamic scheduling, i.e. the re-ordering of the execution order of issued instructions at run-time, is performed. The lack of hardware support makes the task of code generation for the compiler considerably harder. Not only the compiler has to do a further task, it has also less information at its disposal. Some information about the program behaviour is not earlier available than at program execution. Although it may appear that the VLIW approach of DSPs has some distinctive drawbacks, this design decision is backed by some good reasons that can only be understood in context of whole DSP systems. An additional unit for Dynamic Scheduling and several queues for instructions waiting for different functional units increase the hardware complexity substantially. But more complex hardware increases the power consumption and the chip size. An increased chip size on the other hand, results in increased prices. For embedded systems an increased price and a higher power consumption are often not tolerable. Additionally, dynamic scheduling techniques introduce some hardly predictable timing behaviour. In time critical applications, predictability and worst-case guarantees are essential constraints. Therefore, static scheduling and VLIW execution models are the preferred architecture models for DSPs.

Although hardware complexity, power consumption and costs are important issues, one of the main design goals still remains processor performance. In order to meet the high performance constraints, many DSPs use pipelining. The execution of operations in their functional units is divided into stages and consecutive operations can overlap partially. For every single instruction the actual execution time stays the same, but for a sequence of instructions the total time can be reduced drastically. This technique and its effects are well-known from general-purpose processors.

The number of operations that can be issued as a single instruction at a time – the Issue Width – lies in a range from 2 to \(\approx 8\). Earlier DSPs only allowed for issuing an integer- and a floating-point operation simultaneously which often resulted in unbalanced workloads between the two units. Modern DSPs are much more flexible and offer in addition more functional units of the same type. A further increase of the issue width is not necessarily recommendable, because in
the interest of a high utilisation of available resources an corresponding number
of operations that can be executed in parallel must be found by the compiler.
Not all applications offer enough fine-grained parallelism, i.e. data independent
operations, in order to utilise a high number of functional units. Because the
characteristics of different applications vary widely, approaches of scalable DSP
architectures [56, 14] have been developed and become more popular.

**Address Generation Units**

Typical DSP programs (see section 2.2) contain many array data structures with
regular access patterns. In order to relief the arithmetic logic units of the load of
address computations, modern DSP utilise dedicated *Address Generation Units
(AGU)*. An AGU can perform address computations independently and at the
time as the data paths do their tasks.

AGUs support post-/pre-increment/decrement addressing modes and additionally
circular addressing. Therefore, a memory access and the determination of
the address of the next access according to some offset can be done simulaneous.
Efficient addressing of array elements [7, 28] becomes feasible because of
overlapping address and user data arithmetic. Circular addressing modes support
the implementation of ring-buffers that are often used in DSP applications
(compare 2.2.1).

Figure 2.2 (based on [28]) shows the structure of an AGU with Address, Modify
and Length Register Files. With help of the block diagram a post-increment
operation, for example, looks like this. The Address Register Pointer selects a
single register from the Address Register Set. Its value reaches the Address Bus
as effective address which is used for accessing the memory. Additionally, the
value serves as input of an adder/subtractor whose result is put into the modulus
logic. For non-circular addressing modes the modulus logic is not effective and
passes the unchanged input value on. Finally, the value is is written back into
the register file via its input multiplexer. The other operand that is added or
subtracted comes either from the Modify Register File or is an immediate value
or the constant 1. The Modify Register File contains a set of possible modifiers;
a single one can be selected by the Modify Register Pointer. The Modulus Logic
becomes active for circular addressing modes and is supplied with an additional
modulo operator that comes from the Length Register File. Then, the Modulus
Logic passes on the remainder of the division of its input value by the supplied
length value.

For the traversal of an *short int* array at memory address 100 an address
register is loaded with the value 100 and a modify register is loaded with 2.
This value 2 corresponds to the distance expressed in bytes of two consecutive
16-bit values in memory. After the first post-increment operation the value in
the address register will be replaced by 102. Succeeding element addresses are
104, 106, .... If there is another array with *long int* elements from memory
address 0 on, another address register must be initialised to 0 and a modify
2.1. DSP Architectures

![Diagram of Address Generation Unit (AGU) with Address, Modify and Length Register Files]

Figure 2.2: Address Generation Unit (AGU) with Address, Modify and Length Register Files

register to 4. The addresses in ascending order are 0, 4, 8, ... If the long int array is a circular buffer with 10 elements, a length register contains the value $10 \times 4 = 40$. After the 9th element has been addressed at address 36, the address written back to the register file is 0 rather than 40.

Memory Organisation

Often DSPs are organised as Harvard Architectures with separate data and instruction buses. Both buses can be used simultaneously and therefore they can transfer data and instructions at the same time. This results in higher utilisation of DSP internal resources and increases efficiency. Sometimes these buses are multiplexed at their interface to external components.

Several DSPs contain internal memory (On-Chip-RAM). Additionally, they can access external memory via a bus system. On-Chip-RAM is mostly quite small (2...64kB) compared to the overall address space (up to 4 GB), but it is fast because it can be accessed with only small latencies. Effectively, single cycle accesses are possible in pipelined memory systems. The external memory can have considerable size (several MB) and performance is dependent on the memory technology (SRAM, DRAM, ...) and the bus system. Internal and external memory often share the same address space, i.e. there is no need for special instructions for accessing the one or the other memory and neither there are segments registers etc. that have to loaded.
Since the internal memory is often too small to keep all the data, only the most frequently accessed objects should be stored in it in order to make utilise it best. Otherwise, the high memory bandwidth is not used efficiently. Methods for partitioning data between On-Chip and external RAM can be found in [46, 57, 54].

Dual Memory Execution is the ability of some DSPs to access two operands at a time. Two memory accesses can be executed in parallel in such a way that e.g. two operands for a operation can be transferred from memory to registers simultaneously.

2.1.2 Integrated Multiprocessor DSP

Integrated Multiprocessor DSP contain multiple DSPs, micro-controllers and other devices together with some interconnection-network on the same chip. The number of integrated Processing Elements (PE) varies widely (2 \ldots \approx 1500) and also their characteristics. Because of the limited space available on a chip, the complexity of a single PE can be higher when their total number is small. This work focuses on Integrated Multiprocessor DSP with individual processing elements being full and independent DSPs, rather than comparatively simple elements like data-paths under central control (e.g. FUZION 150 or HiPAR-DSP, see [19] and [65], respectively, for more information).

DSPs integrated on a single chip can be differentiated between homogeneous and heterogeneous Multiprocessor DSP (MP-DSP). In a homogeneous MP-DSP all DSPs are of the same type and cannot be distinguished by their architectural properties, whereas in heterogeneous MP-DSPs structurally completely different elements can be found. An example of a homogeneous MP-DSP is the Analog Devices Quad-SHARC DSP Multiprocessor Family (see [4]), and typical examples of heterogeneous MP-DSP are the Texas Instruments TMS320C8x [61] and the Philips R.E.A.L. [47].

The usual application domains of MP-DSP are either high-performance Digital Signal Processing (Quad-SHARC, TMS320C8x) or highly specialised areas which the MP-DSP is adapted to (R.E.A.L.). The more general perspective of the former domain is much more interesting from a compiler developer’s point of view and, therefore, prioritised in this thesis.

MP-DSP aim for exploiting Task-Level Parallelism, i.e. coarse grain parallelism that can be found at the level of tasks or execution threads in a program. Different tasks can be e.g. the stages of a pipelined algorithm or a problem instance of a bigger problem in a task-farming approach. Different tasks of the application are distributed to the DSPs which co-operate in order to speed up the overall computation. The high bandwidth and low latencies of the internal interconnection-network of MP-DSP can keep the communication costs that arise in addition to computation costs low.
MP-DSP can provide enough performance for executing algorithms under real-
time constraints, where no single DSP is capable enough, and additionally, clock
rates – and total power consumption – can be comparatively low.

**Texas Instruments TMS320C80**

The 5-processor Integrated MP-DSP TMS320C80 was presented by Texas In-
struments in 1995. Although a little bit aged by now, it still serves as a good
example of an architecture combining several DSP on a single chip.

Basically, the TMS320C80 contains three different types of components. These
are

- processors and controllers,
- memory,
- and an interconnection network.

All the processors, controllers and memory banks are connected to a crossbar.
Each DSP has its own local memory bank, but the address space is shared, i.e. every processor can access every memory bank. For an overview of the
processor architecture see figure 2.3.

---

**Figure 2.3:** T1 TMS320C80 DSP Architecture [62]
The TMS320C80 is a heterogeneous MP-DSP in the way that it contains structurally different processors. These are a video controller, a transfer controller, a master processor and four DSPs. The video controller has the task of controlling video input and output simultaneously and therefore has two ports to the environment. The transfer controller enables the TMS320C80 to access external memory and to interact with its environment. The actual processing elements are a RISC-like master processor and the four DSPs.

The master processor is tailored for common tasks like packing and unpacking of data streams, but it can also be used for floating-point calculations. However, its main purpose is the coordination tasks among the all the components of the TMS320C80. It is connected to the crossbar interconnection network via two ports, i.e. a 32-bit wide instruction port and a 64-bit wide port to a data cache.

Each of the four DSP of the TMS320C80 has three ports connected to the crossbar. With their 32-bit wide global data port they can access every memory bank in the system, whereas the other 32-bit wide local port only allows for accesses to their own local memory bank. It is possible to access data in two different memory segments over the local and the global port at the same time. Each DSP reads its instructions over a 64-bit wide instruction port. These wide instruction words have several fields that independently control different functional units within each DSP. Here some VLIW principles are used in order to exploit ILP. Additionally, these DSP support certain SIMD instructions, i.e. a single operation can be applied to several different pairs of operands simultaneously (compare 2.1.1).

Each of the memory banks (10K Bytes each) is split up into smaller segments, because every segment can only handle a single access at a time. With more segments the probability of two or more processors accessing the same segment is lower than with just one big memory bank. In addition to the data RAM, there is an instruction cache for each DSP. This is another means of reducing network access conflicts and speeding up the execution of small loop constructs.

The crossbar itself is more than just a switched network, it contains some additional logic to resolve and serialise access conflicts automatically. The crossbar operates in pipelined fashion. Each pipeline stage takes one machine cycle to complete. In the first stage, a processor sends a request containing the most significant part of the address of the next memory access to the crossbar. In case of a conflict, i.e. more than one processor requesting access to a single memory segment, the crossbar logic determines which units is granted access. In the second stage, the processor that was granted access send the LSBs of the address over the crossbar to the RAM in order to perform the actual memory access. The second stage finishes when the memory access has been finished.

Figure 2.4 shows a block diagram of a single DSP of the TMS320C80 with its components. Basically, these components are the Register Set, the Data Unit, the Address Units and the Program Flow Control Unit.

The Register Set contains 44 registers. All of them are visible to the user and
TMS320C8x Parallel Processing Advanced Digital Signal Processors (PP) Block Diagram

Figure 2.4: Architecture of an individual DSP of the TI TMS320C80 [62]

can serve as sources and destinations of ALU and memory operations. The register set is homogeneous, although it is divided into files. By dividing the register set into files the number of register ports and multiplexers can be kept low. This reduces hardware complexity and therefore decreases costs and power consumption and increases the maximal possible clock rate. However, registers in the data unit support over eight accesses in a single cycle whereas most other registers still allow for more than one access in a cycle. Register Files can be found in the Data Unit, the Address Unit and the Program Flow Control Unit.

The *Data Unit* contains Data Registers, Status Registers, a Multiplier Data Path and a ALU Data Path. The ALU and Multiplier Data Path can be used simultaneously in order to exploit ILP. Special operations that are frequently used in DSP like Multiply-Accumulate, Barrel Shifting, Type Expansion and Bit-Field Masking are supported by special hardware of the ALU Data Path. Most of the operations do not take more than a single cycle to complete.

Each DSP contains two *Address Units*. Although nearly identical, the first unit is dedicated to local memory accesses whereas the second one addresses global memory. Both units can perform independent and simultaneous memory operations. Not only the Address Units are independent of each other, they are also independent of the Data Unit. Possible address generation operations are similar to those explained in Section 2.1.1. An immediate index or a register index can be added to an address register in order to generate the address of a memory load or store. Memory arrays can be traversed by using the result of an address computation to modify the content address register. Whenever an
Address Unit is not used for address computations, it can be used for general arithmetic computations on register data. At such occasions, it behaves like a component of the data path.

A DSP support pipelined instruction execution under control of the Program Flow Control Unit. Its tasks are instruction fetch and decode, co-ordination of handshaking with the Transfer Controller, handling of interrupt requests and their prioritisation. More important from a compiler writer’s point of view are the instruction controller, the program counter registers, the cache controller and three hardware loop controllers. The latter support Zero-Overhead Loops by hardware – initiated by dedicated instructions of the instruction set.

### 2.2 DSP Applications

This section is dedicated to DSP applications and their specific properties. DSP applications differ substantially from ordinary kernel code or scientific computations. With the intention to provide an overview of the subject, first a short survey of the theoretical background of Digital Signal Processing is given. The real applications on the example of the MPEG-2 algorithm are discussed. Finally, existing benchmarks suites (DSPstone and MediaBench) are introduced as a feasible way for obtaining comparable and reproducible performance figures.

#### 2.2.1 Theoretical Background

In order to understand the specific requirements to DSP systems, some background knowledge of the theoretical foundations of Digital Signal Processing is – at least – useful. It only supplies a brief and by far not comprehensive overview. For more details the reader might find [63, 8] – on which this section is based on – and [45, 17, 11] an interesting further reading.

The process of generating a sequence of numbers from an analog, or continuous, waveform is known as sampling. The amplitude of an input is measured periodically at a regular interval. The resulting values are represented as numbers over the points in time of taking the measure. Figure 2.5 shows the input waveform – a sine wave –, the periodic sampling pulses with frequency \( f = \frac{1}{T} \), and the resulting discrete-time sampled signal.

Another way to look at sampling is a multiplication process of two signals, i.e. the input signal and the sample pulses. The sample pulses are actually Dirac pulses \( \delta \) with infinite amplitude and infinitesimal pulse width, but with well-defined pulse area which corresponds to strength of the pulse. Sampling as a multiplication is shown in figure 2.6.
2.2. DSP Applications

Figure 2.5: Sampling of a sine wave

Figure 2.6: Sampling as a multiplication
More formally, the resulting equation of multiplying the input signal $U_i(t)$ by a series of Dirac impulses $\delta$ at sampling times $t_\mu$ and sampling interval $T_s$ is

$$U_i(t) = \sum_{\mu=0}^{\infty} U_i(t_\mu) T_s \delta(t - t_\mu).$$  \hfill (2.1)

Figure 2.7 shows a spectrum of an input signal before and after sampling. In order to distinguish the different “replications” of the original signal after sampling, the maximum frequency of the input signal must not be higher than half the sampling frequency. This is known as Nyquist theorem or Sampling theorem

$$f_s \geq 2f_{\text{max}}.$$ \hfill (2.2)

If the input signal contains frequencies $> \frac{1}{2} f_s$, the replicated hulls of the original spectrum overlap in the spectrum of the sampled signal. Therefore, frequency components cannot be identified unambiguously anymore. This effect is known as Aliasing and is highly undesirable, because signals become distorted. The original signal can easily obtained by low-pass filtering the sampled signal if the sampling frequency is high enough. The remaining signal, i.e. the signal passing the filter, corresponds to the original signal.

![Spectrum of a signal before and after sampling](image)

**Figure 2.7: Spectrum of a signal before and after sampling**

Transformation of a signal from the time domain into the frequency domain is done by Fourier Transformation. Using the frequency domain is very helpful for operations concerning the spectrum of a signal, e.g. filtering, modulation and demodulation. Application of the (Discrete) Fourier Transformation on equation 2.1 results in

$$\tilde{X}(f) = T_s \sum_{\mu=0}^{\infty} U_i(\mu T_s) e^{-2\pi i \mu f_s}.$$ \hfill (2.3)
By using Euler’s rule, equation 2.3 can be transformed into

\[ \hat{X}(jf) = T_s \sum_{\mu=0}^{\infty} U_i(\mu T_s)[\cos(2\pi \mu \frac{f}{f_s}) - j\sin(2\pi \mu \frac{f}{f_s})] \]  

(2.4)

Any particular value of \( \hat{X}(jf) \) is a complex number representing the amplitude and the phase of signal at some given frequency \( f \). Each component can be obtained according to following rule

\[ \hat{X}(jf) = a + jb \]

\[ |\hat{X}(jf)| = \sqrt{a^2 + b^2} \]

\[ \theta(\hat{X}(jf)) = \tan^{-1} \frac{b}{a} \]  

(2.5)

Usually, after performing the desired operation on the signal in the frequency domain, the modified signal is converted back to the time domain. This corresponds to retaining the waveform from the spectrum. The process of reversing the Fourier Transformation is called Inverse (Discrete) Fourier Transformation (IDFT). Mathematically, the IDFT can be expressed as

\[ U_o(n) = \frac{1}{N} \sum_{k=0}^{N-1} \hat{X}(k)e^{-2\pi jnk/N} \]  

(2.6)

or

\[ U_o(n) = \frac{1}{N} \sum_{k=0}^{N-1} \hat{X}(k)[\cos(2\pi nk/N) + jsin(2\pi nk/N)] \]  

(2.7)

for a finite number \( N \) of spectral lines.

A component of a Digital Signal Processing system can be treated as a black box that is supplied with a signal and responds with a characteristic output signal. The characteristic function describing its behaviour is the Transfer Function \( H(z) \). If \( x(n) \) is the input signal and \( y(n) \) is the output signal of a DSP component, then \( X(z) \) and \( Y(z) \), respectively, are the corresponding \( z \)-transforms. The transfer function is then defined as

\[ H(z) = \frac{Y(z)}{X(z)}. \]  

(2.8)

In \( z \)-transforms one-sample delays are written as \( z^{-1} \), or in general, \( n \)-sample delays as \( z^{-n} \).

Another important concept in digital signal processing is the notion of Impulse Response. A unit-impulse sequence \( \delta(n) \) is put into a component as input signal,
whereas its response is denoted $h(n)$. Responses to can be either finite or infinite, resulting in Finite Impulse Response (FIR) or Infinite Impulse Response (IIR) systems. Infinite impulse responses come from feedback path from the output back to the input of a system. If no feedback exists in the system, the impulse response will be finite.

Transfer functions and impulse responses are closely related. It is possible to determine the response of a linear system to an input signal exactly, if the impulse response is known. A particular input sample will cause an output that is equal to the input signal times the impulse response. If the input sample is equal to the unit pulse, the output will be the impulse response. Otherwise, if the input sample is scaled by some factor, the output will also be scaled by the same factor. When now the entire input sequence is modelled as a superposition of scaled unit pulses, the output is the linear superposition of scaled impulse responses. More formally, this can be written as

$$y(n) = \sum_{k=-\infty}^{\infty} x(n-k)h(k)$$  \hspace{1cm} (2.9)

or

$$y(n) = \sum_{k=-\infty}^{\infty} x(k)h(n-k).$$  \hspace{1cm} (2.10)

Equation 2.10 is known as Convolution Sum and the process of taking this sum is Convolution.

**Digital Filters**

Important and ubiquitous applications of Digital Signal Processing Theory are Digital Filters. Based on the introductory theory, this section attempts to give a short summary of Digital Filters with special emphasis on the consequences for DSP hardware and software.

Figure 2.8 shows the structure of an FIR filter. Because there exists no feedback path in the filter structure, it has a finite impulse response. The input is scaled by filter coefficients $a_k$ before passing a delay element, or tap. After a single-cycle delay, the value is added to the result of the next stage. Finally, the accumulated sum leaves the system. The difference equation and the transfer function of the shown filter are

$$y_N = a_0x_N + a_1x_{N-1} + \cdots + a_N x_0, \quad y_N = \sum_{k=0}^{N} a_k x_{N-k}$$  \hspace{1cm} (2.11)

and

$$H(z) = \frac{Y(z)}{X(z)} = \sum_{k=0}^{N} a_k z^{-k},$$  \hspace{1cm} (2.12)
2.2. DSP Applications

More suitable for implementation in a serial IC or as a piece of software is the filter shown in figure 2.9. This implementation is based on two ring-buffers that are used cyclically. Instead of a series of delays, adders and multipliers, two ring-buffers, one adder and one multiplier are sufficient. Its function in time domain is

\[ y(t_N) = \sum_{k=0}^{N} \alpha_k x(t_{N-k}) \quad (2.13) \]

FIR filters directly implement convolution, as can be seen by comparing equations 2.13 and 2.10.

After all this theory, it becomes clear how characteristic features of DSP introduced in section 2.1.1 related to the requirements of the application domain. Many sums are accumulations of products. Therefore, a hardware implementation of a Multiply-Accumulate operation can speed up the computation drastically. The unusual addressing modes supported by the AGU, in particular
circular modulo addressing, support software implementation of ring-buffers without further software overhead.

2.2.2 Real Applications

As the most widely used multimedia application the MPEG-2 audio and video encoding and decoding algorithm is selected as an example for the discussion of characteristics of multimedia applications in general. A summary of the properties is given at the end of the section.

MPEG, which stands for Moving Picture Expert Group, is the name of a family of international standards for coding audio-visual information in a digital compressed format. Members of the MPEG family of standards are MPEG-1, MPEG-2 and MPEG-4. MPEG-1 is suited for coding audio-visual signals at bit rates of \( \approx 1.5 \text{ Mb/s} \) and is suitable for storing of video signals with a quality comparable to VHS. MPEG-2 extends MPEG-1 to support higher bit-rates and therefore higher quality. It aims to applications like digital TV. MPEG-4 is a protocol for transmission and storing of video signals at low bit-rates and lower quality than MPEG-1.

MPEG-1/2 are lossy video compression standards using the Discrete Cosine Transformation (DCT), run-length coding and motion compensation. MPEG-2 supports different image sizes and coding schemes which are classified by profiles and levels. Main Profile at Main Level (MP@ML) is the most common profile and level, and can be used for many different application domains like Broadcast Satellite Service or Digital Video Disk (DVD). Motion compensation allows for higher compression rates than ordinary compression because temporal redundancy in video streams resulting from minor changes in subsequent frames is efficiently exploited.

An MPEG stream has a hierarchical data structure for storing its information content. Figure 2.10 shows the levels \textit{video sequence}, \textit{group of pictures (GOP)}, \textit{picture (frame)}, \textit{slice}, \textit{macroblock} and \textit{block}. The MPEG standard differentiates between three different types of pictures. The first picture type of pictures are \textit{Intra (I)} pictures. All macroblocks of intra pictures are stored without motion compensation. Intra pictures are independent in such a way that no further pictures are needed for decoding and therefore they can serve as starting points in the MPEG stream for decoding. In \textit{Predicted (P)} and \textit{Bidirectionally predicted (B)} pictures some macroblocks are encoded using motion compensation based on previous I- or P-pictures (P), or either on previous or succeeding I- or P-pictures (B), respectively. However, some macroblocks might be still encoded without motion compensation.

In 2.11 a block diagram of an MPEG-2 encoder is shown. This encoder consists of several stages which are \textit{Motion Estimation (ME)}, \textit{Forward Discrete Cosine Transformation (FDCT)}, \textit{Quantisation (Q)}, \textit{Variable Length Coding (VLC)}, \textit{Inverse Quantisation (IQ)}, \textit{Inverse Discrete Cosine Transformation (IDCT)},...
Figure 2.10: MPEG video stream data structure [23]
Motion Compensation (MC) and Rate Control and Mode Decision. The encoder which can be implemented either in hardware or in software takes video data as its input and the output is a serial bitstream.

The MPEG-2 decoder in figure 2.12 performs the inverse operation to the decoder. It takes a serial bitstream as an input and outputs video data. Its stages are Variable Length Decoding (VLD), Inverse Quantisation (IQ), Inverse Discrete Cosine Transformation (IDCT), Motion Compensation and a stage for co-ordinating the Decode Control.

The MPEG-2 algorithms contain high levels of coarse-grain and fine-grain parallelism. Coarse-grain parallelism stems from its structure as composition of individual building blocks. Each of the blocks can operate in parallel and e.g. in a pipelined approach of parallelisation each block can correspond to a stage. Fine-grain parallelism can be exploited within each of the blocks. The preferred technique for exploiting this parallelism is the use of ILP processors. Iwata [23] presents some highly interesting results of parallelising the MPEG-2 algorithm for single-chip multiprocessors.

General characteristics of multimedia applications are their need for high computational capacities to accommodate the computational intensity and high data rates. Whereas many applications show processing regularity that can be efficiently utilised, some more modern applications (MPEG-4) are less regular. Generally, multimedia and DSP algorithms are often complex and contain extensive computational demanding parts. On the other hand, there is often a
2.2. DSP Applications

A high degree of parallelism on all levels to be found. Liao [36] gives theoretical upper bounds of parallelism of some multimedia applications (some of them are part of the MediaBench benchmark suite, see 2.2.3) and showed possible speed-up from exploiting ILP (using an ambitious machine model) in a range of 32.8 to 1000. Many multimedia applications have multiple computational phases and deeply nested multi-level loops. Both are suitable targets for exploiting coarse-grain parallelism. A further property of multimedia applications worthwhile mentioning is their high speed of change and new developments. Because multimedia is an active area of research and development, new applications and protocols with higher performance requirements emerge constantly.

2.2.3 Benchmarks

Real applications are often too large and too complex to serve as tools for benchmarking systems. Additionally, often input data can change the results substantially. Therefore, agreed benchmark suites are more preferable for compiler development and performance comparison.

Most ILP compilers have been tested and developed in the context of general-purpose applications. Rather than taking embedded applications, other more general benchmarks have been used for the development of optimisation passes and their performance testing. As seen earlier, DSP and multimedia applications differ enough from other applications to justify an own set of benchmark programs.

Two major benchmark suites have been developed for benchmarking DSP and Multimedia applications. The DSPstone suite [66] contains several smaller programs that contain the actual kernel parts and loops of DSP applications. The MediaBench benchmark [27, 43] takes a more “whole application” point of view and contains a set of programs like JPEG, MPEG, GSM, G.721 encoding/decoding, PGP encryption/decryption, Ghostscript PostScript interpretation, Mesa OpenGL graphic rendering and some more.

**DSPstone** is a benchmark suite consisting of small programs written in C. The programs represent small kernel loops and are written in a C style that can often be found in “real” DSP applications. A description of the programs of the DSPstone benchmark suite is shown in table 2.1.

**MediaBench** benchmark applications aim to be representative workloads of multimedia and communications systems. MediaBench contains 19 full applications from different domains like image and video processing, audio and speech processing, encryption and computer graphics. Table 2.2 gives some brief descriptions of the individual applications of the MediaBench suite.

---

1. This style tends to be completely incomprehensible and it seems reasonable to suspect that the programmers actually were electronic engineers rather than computer scientists.
<table>
<thead>
<tr>
<th><strong>ADPCM</strong></th>
<th>Adaptive Differential Pulse Code Modulation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>complex_multiply</strong></td>
<td>Multiplication of two complex numbers</td>
</tr>
<tr>
<td><strong>complex_update</strong></td>
<td>Update of a complex number, similar to Multiply Accumulate with complex numbers, but with arbitrary destination</td>
</tr>
<tr>
<td><strong>convolution</strong></td>
<td>Computation of a convolution sum (see 2.2.1)</td>
</tr>
<tr>
<td><strong>dot_product</strong></td>
<td>Dot-product of a (1, 2) and a (2, 1) vector</td>
</tr>
<tr>
<td><strong>fir</strong></td>
<td>FIR filter with parameterisable number of taps (see 2.2.1)</td>
</tr>
<tr>
<td><strong>fir2dim</strong></td>
<td>FIR filter for image filtering, i.e. the filter is applied to matrix data rather than a sequence of values</td>
</tr>
<tr>
<td><strong>biquad_N_sections</strong></td>
<td>IIR biquad filter with parameterisable number of sections</td>
</tr>
<tr>
<td><strong>biquad_one_section</strong></td>
<td>Computations for one section of an IIR biquad filter</td>
</tr>
<tr>
<td><strong>lms</strong></td>
<td>Implementation of an adaptive DLMS filter</td>
</tr>
<tr>
<td><strong>matrix</strong></td>
<td>Two programs (matrix1, matrix2) for the multiplication of two matrices of arbitrary dimensions</td>
</tr>
<tr>
<td><strong>matrix1x3</strong></td>
<td>Multiplication of a (3, 3) matrix by a (3, 1) vector</td>
</tr>
<tr>
<td><strong>n_complex_updates</strong></td>
<td>Updates of n complex numbers in a way similar to Multiply-Accumulate</td>
</tr>
<tr>
<td><strong>n_real_updates</strong></td>
<td>Updates of n complex numbers with values coming from three different arrays and multiplication and addition as operators</td>
</tr>
<tr>
<td><strong>real_update</strong></td>
<td>A single real update</td>
</tr>
</tbody>
</table>

Table 2.1: Description of the DSPstone benchmark suite

According to Lee [27] the initial goals of MediaBench are to:

1. **Accurately represent the workload of emerging multimedia and communications systems.**

2. **Focus on portable applications written in high-level languages, as processor architectures and software developers are moving in this direction.**

3. **Precisely establish the benefits of MediaBench compared to existing alternatives, e.g. integer SPEC.**

4. **Develop a tool that is effective for system evaluation as well as system synthesis.**

The applications in table 2.2 labelled with an asterisk (*) are not actual part of the MediaBench 1.0 benchmark suite. Nonetheless, often these applications are added to test series in order incorporate some more computational intensive applications that represent emerging multimedia applications.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADPCM</td>
<td>A simple adaptive differential pulse code modulation coder ((\text{rawaudio})) and decoder ((\text{rawaudio}))</td>
</tr>
<tr>
<td>EPIC</td>
<td>An image compression coder ((\text{epic})) and decoder ((\text{unepic})) based on wavelets and including run-length/Huffman entropy coding</td>
</tr>
<tr>
<td>G.721</td>
<td>Voice compression coder ((\text{encode})) and decoder ((\text{decode})) based on the G.711, G.721 and G.723 standards</td>
</tr>
<tr>
<td>Ghostscript</td>
<td>An interpreter ((\text{gs})) for the PostScript language; performs file I/O but no graphical display</td>
</tr>
<tr>
<td>GSM</td>
<td>Full-rate speech transcoding coder ((\text{gsmencode})) and decoder ((\text{gsmdecode})) based on the European GSM 06.10 provisional standard</td>
</tr>
<tr>
<td>H-263 (*)</td>
<td>A very low bitrate video coder ((\text{h263enc})) and decoder ((\text{h263dec})) based on the H.263 standard; provided by Telenor R&amp;D</td>
</tr>
<tr>
<td>JPEG</td>
<td>A lossy image compression coder ((\text{cjpeg})) and decoder ((\text{djpeg})) for colour and grayscale images, based on the JPEG standard; performs file I/O but no graphical display</td>
</tr>
<tr>
<td>Mesa</td>
<td>A 3-D graphics library clone of OpenGL; includes three demo programs ((\text{mipmap, osdema, texgen})); performs file I/O but no graphical display</td>
</tr>
<tr>
<td>MPEG-2</td>
<td>A motion video compression coder ((\text{mpeg2enc})) and decoder ((\text{mpeg2dec})) for high-quality video transmission, based on the MPEG-2 standard; perform file I/O but no graphical display</td>
</tr>
<tr>
<td>MPEG-4 (*)</td>
<td>A motion video compression coder ((\text{mpeg4enc})) and decoder ((\text{mpeg4dec})) for coding video using the video object model; based on the MPEG-4 standard; perform file I/O but no graphical display; provided by the European ACTS project MuMoSys</td>
</tr>
<tr>
<td>PEGWIT</td>
<td>A public key encryption and authentication coder ((\text{pegwitenc})) and decoder ((\text{pegwitdec}))</td>
</tr>
<tr>
<td>PGP</td>
<td>A public key encryption coder ((\text{pgpenc})) and decoder ((\text{pgpdec})) including support for signatures</td>
</tr>
<tr>
<td>RASTA</td>
<td>A speech recognition application ((\text{rasta})) that supports the PLP, RASTA and Jäh-RASTA feature extraction techniques</td>
</tr>
</tbody>
</table>

Table 2.2: Description of the MediaBench benchmark suite [18]


2.3 DSP Compilers

For a long time, compilers for embedded systems, in particular DSPs, have been neglected whereas compiler research for general-purpose CPUs and parallel computers was flourishing. Lately, researchers have realised specific requirements of DSP compilers and focused their work on embedded systems as targets of compilation and optimisation [37, 39, 34].

Because there are many more limitations during code generation for DSP processors than for general purpose processors, code generation for DSPs is much harder. Not only there are more constraints to observe in DSP code generation, there are also other priorities in the goals to achieve. Whereas for general-purpose processors power consumption is no issue at all – at least from the compiler developer’s point of view – one of the most important goals for DSP system developers is to minimise power consumption. At the same time, programs must meet tight timing constraints and must fit into small on-chip ROMs. Gebotys [20] characterises one of the challenging problems of DSP code generation as follows:

Assume we are given a sequence of operations, represented by an ordered list of operations. The objective is to select instructions, compact code, allocate registers, and perform memory addressing for the target processor. The set of operations is mapped into a set of instructions (assembly code). The resultant code must meet code size, and energy dissipation constraints.

The following sections give brief introductions to some specific problems of compilation for DSPs. Further, some introduction to data flow analysis is presented because data flow information is needed at various compilation stages. For a more comprehensive overview of the highly interesting topic of DSP compilers the reader is referred to [40, 9, 30, 31, 28, 32].

Compilers for general-purpose processors are expected to be fast. Generally, algorithms with bigger than linear computational complexity are considered to be unsuitable for compilers. The situation is different for DSP compilers. Often, DSP software is developed manually on assembly level. Therefore, every support by a compiler that can ensure roughly the same code quality (performance, code size) is considered as a valuable contribution to decrease the time-to-market. In this context, it is permissible for DSP compilers to take up large CPU resources. Software prototyping can be done without sophisticated optimisations, just to get the program running. The final version can then be optimised in a last step that might take longer. Compilation times for that last step of up to several hours are acceptable [30].

A few experimental compilation systems (LANCE [12], SUIF [2]) specifically developed for DSPs are available and form a basis for further research and development [51].
2.3.1 Code Generation

Code generation algorithms for DSP processors can be based on Data Flow Trees [40]. For a data flow tree, e.g. of an expression, the goal is to cover the whole tree with tree patterns that correspond to available instructions of the processor. Each pattern is assigned a cost equivalent to the cost of the represented instruction. The optimum tree covering is the one with the lowest total cost. Starting from a data flow tree and a set of instruction patterns with their associated costs (i.e. the instruction set of the target machine), the covering can be done by a Dynamic Programming algorithm. For machines with complex instructions like the MAC-instruction of DSPs, it is often a hard task to find the optimum covering. Sometimes the number of possible coverings is large, and also there might be several different optimal coverings. Figure 2.13 shows an example of an expression and some available instruction patterns. The nodes of the trees represent operations and the edges data flows. Each dotted circle or oval represents an instruction. As it can be seen in the figure, there are different possibilities to cover the add and mul operations. Implementing all operations of the expression is equivalent to finding a cover of that expression by instructions. It quickly becomes clear that the task of determining the cover for which the cost is minimal is not trivial. For more information on code generation see [41, 10, 33].

<table>
<thead>
<tr>
<th>Instr</th>
<th>Cost</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>1</td>
<td>t1 := Mem</td>
</tr>
<tr>
<td>mov2</td>
<td>1</td>
<td>r2 := r1</td>
</tr>
<tr>
<td>mov3</td>
<td>1</td>
<td>r3 := r1</td>
</tr>
<tr>
<td>add</td>
<td>2</td>
<td>r1 := r1 + r2</td>
</tr>
<tr>
<td>mul</td>
<td>2</td>
<td>r1 := r1 * r2</td>
</tr>
<tr>
<td>mac</td>
<td>3</td>
<td>r1 := (r1 * r2) + r3</td>
</tr>
</tbody>
</table>

Figure 2.13: Expression \((a \times b) + (c \times d)\) and instruction patterns [40]

The instructions sets of modern DSPs support Multimedia SIMD instructions (see 2.1.1 and 3.1). Whereas a machine word is 32-bit wide, many Multimedia applications require only 8- or 16-bit data types. The idea is to split up a 32-bit data path in to 2 or 4 segments of each 16 or 8 bit and to perform identical operations on these parts simultaneously. For a compiler the generation of SIMD operations is quite difficult. As described above, usual code generation techniques use tree pattern matching to cover expression trees with small fragments corresponding to available instructions. A SIMD operation is not a single fragment of a tree anymore, but it consists of several tree patterns corresponding to each partial operation the entire instructions performs. Additionally, alignment constraints of the individual operations must be considered. Because of these
difficulties, many DSP compilers do not support automatic SIMD instruction generation. The user has the choice of either using library functions that were manually coded using SIMD instructions or inlining its own assembly routines. Both ways are not very flexible and not portable at all. A novel approach has been developed by Leupers [30, 35]. It uses an extended graph covering algorithm in addition to Integer Linear Programming methods to generate code utilising SIMD instructions. Alignment constraints are respected automatically.

2.3.2 Address Code Generation

Typically, DSPs only have few addressing modes. Hence, there are only few influences on the code quality by the choice of the addressing mode. However, opportunities for code optimisations arise from the availability of pre/post-increment/decrement modes (see chapter 2.1.1). For an efficient use of the AGU the way data variables are arranged in memory and how the address registers correspond to memory reference is essential.

The problem of Address Code Generation can be divided into sub-problems. The first problem is to arrange the data layout in memory, and the second problem is concerned with the optimal use of available address and index registers of the AGU. Based on graph algorithms the address code generation problem can be solved for scalar variables as well as for subscripted variables. The available algorithms [7, 31] can find optimal or near optimal solutions, respectively, and are efficient. By using these algorithms that are not yet included in all commercially available DSP compilers the use of pointer-based array accesses for array traversal becomes superfluous.

Another problem related to address code generation is the data partitioning between different memory banks. Since some DSPs support Dual Memory Execution (see chapter 2.1.1), two memory accesses to different banks can be issued simultaneously. In order to achieve the best performance, variables that need to be accessed at the same time should be placed in different memory banks. For this problem some approaches to solutions can be found in [46, 57].

2.3.3 C specific problems

Problems so far dealt with specific properties at machine level. But at the other end, the processing of high-level programming languages, some problems typical for DSP applications can be found.

Since the only high-level programming language used for DSP programming is C, the programmers do not have any language constructs specifically designed for their application domain and are forced to use C elements that are actually adapted to different kind of processor architectures. This gap between the programming language and the underlying architecture makes it often difficult for a compiler to generate optimal code from its input. Sometimes programmers
2.3. DSP Compilers

attempt to gives hints to a specific compiler they are familiar with by using certain constructs. If this code is compiled by a different compiler, this coding style may prevent successful optimisation.

An example of the abundant use of pointers is given in chapter 3.4. Usually, DSP compilers do not perform any pointer or alias analysis. Therefore, pointer use can be a major obstacle on performing compiler-directed program optimisations. Interesting starting points for a further reading on pointer analysis of C code are [38, 64, 21].

Many DSPs support fix-point arithmetic. Unfortunately, there are no corresponding data types in C. Originally, programmers used the available integer data types as objects for storing and manipulating fix-point values, but the use of constants was inconvenient. In order to express a constant, the corresponding bit patterns must be used. In the meantime, an extended DSP-C has been standardised. It contains a \texttt{frac} data type for fix-point values and corresponding operators.

2.3.4 Scalar Data Flow Analysis

Data Flow Analyses for scalars are a well researched subject and methods found their way into common Compiler Textbooks [1, 6, 44]. Nonetheless, a brief overview is given in order to present the basics of their application and possible extensions.

Data Flow Analysis aims to determine properties, e.g. concerning data dependences, for a given program. These properties are represented by values of a domain $D$. During and after the analysis, all nodes of the Control Flow Graph corresponding to the program to be examined, are labelled with values from $D$. The values at some node $n$ represent the properties before and after execution of the instruction represented by $n$. The effects of that instruction on the properties are modelled by a Transfer Function $f : D \rightarrow D$. Whenever control flow passes the node $n$, its transfer function is applied to the data valid at its entry. The output data is then passed on to its control flow successors. In this context, the transfer functions of a control flow graph form a special equation system, the Data Flow Equation System. With formal methods this equation system can be solved, the resulting values can be used to extract information about the properties of interest.

The domain $D$ of properties the analysis aims to determine is usually realised by some Lattice $L$. A lattice offers next to a set of values — corresponding to the properties — some operators working on these values. These allow for combining of different properties coming from different control flow predecessors of a node.

**Definition 2.3.1** A lattice $L$ consists of a set of elements, the Carrier Set, and two operators $\cap$ (meet) and $\sqcup$ (join) such that:
1. **Closedness:**
\[ \forall x, y \in L : \exists z, w \in L : x \sqcap y = z \land x \sqcup y = w \]

2. **Commutativity:**
\[ \forall x, y \in L : x \sqcap y = y \sqcap x \land x \sqcup y = y \sqcup x \]

3. **Associativity:**
\[ \forall x, y \in L : (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z) \land (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z) \]

4. **Supremum, Infimum**

- \[ \exists \bot \in L : \forall x \in L : x \sqcap \bot = \bot \]
- \[ \exists \top \in L : \forall x \in L : x \sqcup \top = \top \]

The elements \( \bot \) and \( \top \) are called bottom and top, respectively.

**Definition 2.3.2** A lattice \( L \) is a Distributive Lattice, if in addition
\[ \forall x, y, z \in L : (x \sqcap y) \sqcup z = (x \sqcup z) \sqcap (y \sqcup z) \land (x \sqcup y) \sqcap z = (x \sqcap z) \sqcup (y \sqcap z) \]

hold.

**Definition 2.3.3** The partial order \( (L, \sqsubseteq) \) induced by some lattice is defined by \( \forall x, y, z \in L : x \sqsubseteq y \iff x \sqcap y = x \). An equivalent definition is possible based on the \( \sqcup \)-operator, \( \sqcap, \sqsubseteq \) and \( \sqsupseteq \) are defined analogously.

**Definition 2.3.4** The height of a lattice \( L \) is defined as
\[ \text{height}(L) = \max\{n | \exists x_1, \ldots, x_n : \bot = x_1 \sqsubseteq x_2 \sqsubseteq \ldots \sqsubseteq x_n = \top\} \].

Properties modelled by data flow lattices are subject to change on instruction execution. Hence, a formalism that can model this change is required. Transfer function as described above are a suitable formalism for this purpose. Formally, transfer functions are function \( f : L \to L \) that are assigned to each node of a Control Flow Graph \( CFG = (N, E, s, e) \), with set of nodes \( N \), set of edges \( E \), start node \( s \) and end node \( e \). The mapping of nodes to transfer functions has the form \( tf : N \to (L \to L) \). The specific form of a transfer functions depends on the instruction to be modelled, its operands and the data flow property to be modelled. Most of the times, the specific operation of an instruction is less important than the way its operands are accessed. Reading the value of a variable is a Use, whereas writing a value into it is a Definition. Uses and Definitions are commonly termed as References. Depending upon whether a property initially holds or does not hold anymore after some reference, this reference is called generating or killing reference. The determination of generating and killing references is only possible in context of a specific data flow problem.
An important properties of a data flow framework are the monotony of its transfer functions and the effective height of its data flow lattices. Both properties have strong influence on the computational complexity and even on termination itself of data flow analyses.

**Definition 2.3.5** A function \( f : L \to L \) is monotone, iff
\[
\forall x, y \in L : x \subseteq y \Rightarrow f(x) \subseteq f(y)
\]
holds.

**Definition 2.3.6** The Effective Height of a lattice \( L \) with respect to some function \( f : L \to L \) is
\[
\text{height}^f_{\epsilon f}(L) = \max \{ n | \exists x_1, x_2 = f(x_1), x_3 = f(x_2), \ldots, x_n = f(x_{n-1}) \in L : x_1 \sqsubseteq x_2 \sqsubseteq x_3 \sqsubseteq \ldots \sqsubseteq x_n \sqsubseteq T \}.
\]
The effective height denotes the longest, strictly monotone ascending chain of iterated applications of the function \( f \) on elements of the lattice \( L \).

In general, a node \( n \) is reachable not only on a single path, but there a several path from the start node \( s \) to the node \( n \). Hence, a valid solution has to include all these alternative paths. Depending on the following use of the analysis, the user might be interested in properties that may hold or that must hold over all paths. The difference is that a may-solution includes values that may hold on a single path, whereas the must-solution includes on values that are valid on all paths. Information coming from different preceding nodes into some node of meeting control flow edges, are combined with the meet-operator of the lattice. According to “may” or “must” there are two different representations of properties holding at some node \( n \).

- Let \( \pi(n) \) the set of all paths \( s \leadsto n \) for a node \( n \in N \). Then, the Merge-over-all-path (MOP)-solution for the May-problem is
\[
MOP_{May}(n) = \bigcup_{p \in \pi(n)} f_p(\iota) \tag{2.14}
\]
for some initial value \( \iota \in L \).

- Let \( \pi(n) \) the set of all paths \( s \leadsto n \) for a node \( n \in N \). Then, the Merge-over-all-path (MOP)-solution for the Must-problem is
\[
MOP_{Must}(n) = \bigcap_{p \in \pi(n)} f_p(\iota) \tag{2.15}
\]
for some initial value \( \iota \in L \).
The MOP-solutions are exact solutions to data flow problems. If the transfer functions are not monotone, the number of different control paths in the program is (potentially) infinite or the effective height of the data flow lattice is unbounded the MOP solutions cannot be determined. In such a case, an approximation solution can be computed. The precision of the approximation is the quality of the approximation solution, i.e. the difference to the actual solution. Though, more important is the kind of permissible error. To ensure correctness of optimisation using the data flow approximation solution, in case of must-problems must underestimate the actual solution whereas in case of may-problem an overestimation is safe.

Safe approximations can be computed for monotone transfer functions and bounded data flow lattices. Starting from some initial value the transfer functions can be applied iteratively until after some iterations the data flow values do not change anymore. In this case a Fixpoint has been reached.

**Definition 2.3.7** A Fixpoint of a function \( f : L \to L \) is an element \( x \in L \) such that \( f(x) = x \).

Every fixpoint is a valid solution to the data flow equation system. Further applications of transfer functions do not change any values anymore. This means that all dependences have been propagated, even cyclically. The entire equation system is in state of equilibrium. The data flow values at the nodes are ready for use.

A fixpoint can be determined in an iterative process starting with some initial values. The following equation formalise the notions of *Minimal* and *Maximal Fixpoints* for may- and must-problems, respectively.

- A **Minimal Fixpoint** (MFP) for a May-problem is
  \[
  MFP_{May}(n) = \begin{cases} 
  \bigcup_{m \in \text{pred}(n)} f_n(n)(MFP(m)) & \text{if } n = s \\
  f_n(n)(MFP(m)) & \text{otherwise}
  \end{cases} \quad (2.16)
  \]
  for some initial value \( v \in L \).

- A **Maximal Fixpoint** (MFP) for a Must-problem is
  \[
  MFP_{Must}(n) = \begin{cases} 
  \bigcap_{m \in \text{pred}(n)} f_n(n)(MFP(m)) & \text{if } n = s \\
  f_n(n)(MFP(m)) & \text{otherwise}
  \end{cases} \quad (2.17)
  \]
  for some initial value \( v \in L \).
In general, i.e. without further assumptions, it is \( \forall n \in N : \text{MOP}_{\text{Mast}}(n) \supseteq \text{MFP}_{\text{Mast}}(n) \) and \( \forall n \in N : \text{MOP}_{\text{Mas}(n)} \supseteq \text{MFP}_{\text{Mas}(n)} \), respectively. However, if all transfer functions are distributive, the MOP in this special case can be computed and \( \forall n \in N : \text{MFP}(n) = \text{MOP}(n) \) holds.

The most widely used algorithm for computing data flow fixpoints is the Iterative Data Flow Analysis. This algorithm directly implements the concept of fixpoints for a given Control Flow Graph. Each node \( n \) of the \( \text{CFG} = (N, E; s, e) \) is labelled with two values \( \text{IN}[n], \text{OUT}[n] \in L \). These values denote data flow reaching and leaving the node, respectively. The must-data flow equations for a node \( n \) look like this:

\[
\text{IN}[n] = \begin{cases} 
\ell & \text{if } n = s \\
\bigcap_{m \in \text{pre}(n)} \text{OUT}[m] & \text{otherwise}
\end{cases} 
\]

(2.18)

\[
\text{OUT}[n] = f_n(\text{IN}[n])
\]

(2.19)

\( \text{IN}[n] \) is the greatest lower bound of the data flow values coming from preceeding nodes. Another representation of \( \text{OUT}[n] \) which is based on sets of generating and killing references \( G \) and \( K \) is possible and has following form:

\[
\text{OUT}[n] = G[n] \cup (\text{IN}[n] - K[n])
\]

(2.20)

A Worklist Algorithm traverses the \( \text{CFG} \) in Reverse Postorder and applies the transfer function each time a new node is encountered. The node is removed from the worklist and its \( \text{OUT} \) values are computed and propagated to its succeeding nodes. If a nodes’ \( \text{IN} \) value is changed in course of the value propagation, it is added to the worklist. The algorithm terminates when there are no more elements in its worklist.

For some data flow problems, so called Reverse Data Flow Problems, the role of the \( \text{IN} \) and \( \text{OUT} \) sets must be interchanged. The data flow moves from \( \text{OUT} \) to \( \text{IN} \).

A framework for scalar data flow analysis has been developed during this project and integrated into the Octave experimental compiler.

### 2.3.5 Array Data Dependence Test

Data dependences not only occur among scalar variables, but also between subscripted variables, i.e. array elements. The problem of determining whether or not there is a data dependence becomes substantially harder for array elements, because a single array reference in the source code is mapped to a set of memory locations dependent on the iteration space of enclosing loops and index functions of the array access. In the following, a short introduction to the problem
is given before two different approaches (Memory Disambiguation and Array Data Flow Dependence) are discussed based on two specific algorithms.

Data Dependences in loop can be distinguished by the number of iterations involved. A loop-independent dependence exists regardless of the loop structure. Whereas loop parallelisation is not inhibited, the statement order in the loop body is affected. On the other hand, there are loop carried dependences which are induces by the iterations of a loop. Loop carried dependences restrict or prevent safe loop parallelisation. Usually, when a statement \( S_2 \) is dependent on another statement \( S_1 \), they are in a dependence relation \( \delta \) to each other, or symbolically \( S_1 \delta S_2 \).

Depending on the combinations of the types of memory accesses (use/definition), dependences can be further differentiated. A True/Flow Dependence occurs when \( S_1 \) writes a memory location that \( S_2 \) later reads, denoted as \( S_1 \delta_{\text{flow}} S_2 \). An Anti Dependence \( S_1 \delta_{\text{anti}} S_2 \) occurs when \( S_1 \) reads a memory location that is later written by \( S_2 \). Two statements \( S_1 \) and \( S_2 \) are involved in an Output Dependence, if \( S_1 \) writes to a memory location that \( S_2 \) writes later, too. Finally, an Input Dependence exists, when \( S_1 \) reads a memory location that later \( S_2 \) reads. True, Anti and Output Dependences restrict the execution order of the dependent statements, i.e. dependent statements cannot be interchanged in their execution order without violating semantic correctness. On the contrary, Input Dependences do not restrict execution order. True Dependences are the only dependences imposed by some algorithm, the others are just artifacts of the actual programming or code generation. By using the Static Single Assignment (SSA) Form for representing programs, Anti, Output and Input Dependences can be removed.

Assume a loop nest as shown in figure 2.14. If there is a dependence between statement \( S_1 \) and \( S_2 \), then the Source \( S_1 \) must be executed before the Sink \( S_2 \). It is not important when exactly \( S_1 \) or \( S_2 \) is executed, as long as it ensured that the value used by \( S_2 \) is defined beforehand by \( S_1 \).

The problem of Array Data Dependence Analysis is to determine whether or not there is a dependence between two statements \( S_1 \) and \( S_2 \) containing array references. Mathematically, this is the question if for vectors of integers \( \alpha = (i_1, \ldots, i_n) \) and \( \beta = (i'_1, \ldots, i'_n) \) with \( L_k \leq i_k, i'_k \leq U_k \) following equation holds

\[
\exists \alpha \leq \beta, \text{ such that } f_k(\alpha) = g_k(\beta) \quad \forall k, 1 \leq k \leq m. \tag{2.21}
\]

Most of the times, the index functions \( f_1, \ldots, f_m \) and \( g_1, \ldots, g_m \) are not formed arbitrary, but they are of a simple structure. Usually, the index functions are linear functions in their induction variable and the form \( f(i) = a \times i + b \) for constants \( a \) and \( b \). Such functions are called Affine Functions. If there are no restrictions imposed to the index functions, the general Array Data Dependence Problem is intractable.

In order to solve the Array Data Dependence Problem for affine index functions, an integer solution to a set of linear equations has to be found. This problem
for($i_1 = L_1; \ i_1 < U_1; \ i_1++)
{  
for($i_2 = L_2; \ i_2 < U_2; \ i_2++)
{  
  :  
  for($i_n = L_n; \ i_n < U_n; \ i_n++)
  {  
    a[f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)] = \ldots; \quad S_1  
    \ldots = a[g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n)]; \quad S_2  
  }  
}  
}

Figure 2.14: Array Data Dependences in a Perfect Loop Nest

is known to be NP-complete. Because of practical considerations an exponential run-time algorithm cannot be used in compilers. Even for medium-sized problems the performance is too poor. Therefore, different approximations, i.e. inexact solution methods, have been developed that need considerably less resources. A class of algorithms that only take memory accesses into account regardless of the actual control and data flow, is summarised under the name Memory Disambiguation. Two statements are considered data dependent, when they potentially access the same memory location. A simple Memory Disambiguation algorithm is the Banerjee Test which is discussed in the next section. More precise Array Data Flow Analyses that determine actual use-definition pairs of statements are subject of the over-next section.

Banerjee Test

The Banerjee Test for data dependence testing is an inexact test. The basic idea is to test for a real solution to the integer equations. Real solutions can be obtained much more easily than integer solutions, i.e. the problem is not NP-complete anymore and the algorithm has therefore lower computational complexity.

For two points $I = (i_1, \ldots, i_n)$ and $I' = (i'_1, \ldots, i'_n)$ in the iteration space let

$$ h(I, I') = \alpha I - \beta I' $$

and

$$ h^+_k(I_k, I'_k) = \max R_k h(I_k, I'_k) \quad (2.23) $$

$$ h^-_k(I_k, I'_k) = \min R_k h(I_k, I'_k). \quad (2.24) $$
\( I_k D I_k' \) is the relation imposed by the direction vector element (either “\(<\)”, “\(>\)” or “\(=\)”).

Then, the Banerjee inequality states, that for a given direction vector \( D \)

\[
\exists \text{ real solution to } \alpha I - \beta I' \iff \sum_{i=1}^{H} H_i - D_i \leq \beta_0 - \alpha_0 \leq \sum_{i=0}^{H} H_i + D_i.
\]

(2.25)

The Banerjee test has been implemented for data dependence testing of array elements during this project. It has been integrated into the software-pipelining framework as part of the data dependence analysis.

**Array Data Flow Analysis**

More powerful, i.e. more precise, than Memory Disambiguation are Array Data Flow Analysis algorithms. Array Data Flow Analysis is the extension of Scalar Data Flow Analysis to arrays. Whereas in the case of scalars, a program variable directly corresponds to a single memory location, an array reference in a program might correspond to several memory locations. In the following one Array Data Flow Analysis algorithm that extends the Scalar Data Flow Analysis naturally is presented. The description is based on [13, 16].

The algorithm works on *Loop Control Flow Graphs* (*LCFGs*). An LCFG is a control flow graph of a loop with an additional *Exit Node* at the end of the loop body. A further extension is the property that an inner loops can be contracted to a single summary node after it has been processed. The output of the algorithm are the *Iteration Distance Vectors* of the data flow dependent array access. Based on the *Reaching Definitions* problem (see [1, 44]) an *Iteration Dependence* can be defined as follows:

**Definition 2.3.8** A definition \( d \) reaches a point \( p \) with Iteration Distance \( \delta \), if all of the last \( \delta \) instances of \( d \) reach \( p \). An iteration distance is maximal, if \( \delta \) is the biggest value of the iteration distance for which the definition \( d \) reaches the point \( p \).

The property to be modelled by a data flow lattice is the iteration distance \( \delta \). Therefore, the lattice must have at least as many elements as the loop has iterations and an additional element to represent independence. A suitable carrier set of the array data flow lattice is \( \{ \bot, 0, 1, \ldots, \top = UB - 1 \} \) with the upper loop bound \( UB \). The lattice is a chain lattice, i.e. it forms a strictly monotone ascending chain of elements \( \bot \leq 0 \leq 1 \leq \ldots \leq \top \) with the usual comparison operator \( \leq \). The operators *meet* and *join* are chosen as maximum (max) and minimum (min). After all, the data flow lattice \( L \) looks like this:

\[
L = ( \{ \bot, 0, \ldots, \top = UB - 1 \}, \bot, \top, \leq, \wedge, \lor )
\]

(2.26)
such that

\[
\forall x_i \in [\bot, 0, \ldots, T = UB - 1]: x_i < x_{i+1}
\]  \hspace{1cm} (2.27)

and

\[
\land(x, y) = \begin{cases} 
\bot & \text{, if } x = \bot \text{ or } y = \bot \\
x & \text{, if } y = \top \\
y & \text{, if } x = \top \\
\min(x, y) & \text{, otherwise}
\end{cases}
\]

\[
\lor(x, y) = \begin{cases} 
\top & \text{, if } x = \top \text{ or } y = \top \\
x & \text{, if } y = \bot \\
y & \text{, if } x = \bot \\
\max(x, y) & \text{, otherwise}
\end{cases}
\]  \hspace{1cm} (2.29)

Additionally, a further operation is needed for modelling the transition between iterations. This operation is a modified increment operator that is defined for all elements of \( L \):

\[
x++ = \begin{cases} 
\top & \text{, if } x = \top \\
\bot & \text{, if } x = \bot \\
x + 1 & \text{, otherwise}
\end{cases}
\]  \hspace{1cm} (2.30)

In a similar fashion as with scalar data flow analysis, array data flow analysis needs transfer functions to model the effects of execution an instruction. Corresponding to generating and killing array references \( G[n] \) and \( K[n] \), there are Generate and Preserve Functions. Additionally, a new class of Exit Functions is introduced. Exit functions represent the step from one iteration into the next one. For detailed information on the construction of the Generate, Preserve and Exit Function have at the original publication [13].

Each node \( n \) of the LCFG is labelled with two vectors of \( m \) lattice elements, if \( m \) is the number of array references in the loop to be analysed. These vectors are \( \mathit{IN}[n] = (x_1, \ldots, x_m) \) and \( \mathit{OUT}[n] = (y_1, \ldots, y_m) \) and have the same function as their scalar counterparts. After the data flow analysis is finished, the data flow solution can be read from \( \mathit{IN} \) and \( \mathit{OUT} \). The transfer functions together with the \( \mathit{IN} \) and \( \mathit{OUT} \) vectors are the data flow equation system which must be solved. The equation system can be solved with iterative worklist algorithm that is already known from the previous section. For initialisation the nodes of the LCFG are traversed in Reverse Postorder. For every node \( n \) and \( \forall d \in [1, \ldots, m] \) following assignment is performed:
\[ IN[n, d]^0 = \begin{cases} \bot & \text{if } n = 1 \text{ Loop Entry} \\ \bigwedge_{m \in \text{pred}(n)} OUT[m, d]^0 & \text{otherwise} \end{cases} \quad (2.31) \]

\[ OUT[n, d]^0 = \begin{cases} \top & \text{if } d \in G[n] \\ \bigwedge_{m \in \text{pred}(n)} IN[n, d]^0 & \text{otherwise} \end{cases} \quad (2.32) \]

The succeeding iteration steps that are also done in Reverse Postorder apply following equations\(^2\):

\[ IN[n, d]^{i+1} = \begin{cases} \bigwedge_{m \in \text{pred}(n)} OUT[m, n]^i & \text{if } n = 1 \\ \bigwedge_{m \in \text{pred}(n)} OUT[m, d]^{i+1} & \text{otherwise} \end{cases} \quad (2.33) \]

\[ OUT[n, d]^i = f^d(IN[n, d]^i) \quad (2.34) \]

Because all analysed loops are structured loops which are Single-entry/Single-exit loops, equation 2.33 can be simplified:

\[ IN[n, d]^{i+1} = \begin{cases} OUT[\text{Exit}, n]^i & \text{if } n = 1 \\ \bigwedge_{m \in \text{pred}(n)} OUT[m, d]^{i+1} & \text{else} \end{cases} \quad (2.35) \]

The shown framework takes no more than three passes over the loop body and is therefore a linear-time algorithm. There are possibilities for parameterising the analysis in order to solve very different array data flow problems. Additionally, perfect loop nests and multi-dimensional arrays can be handled.

A prototype implementation of this algorithm has been used during this project. Although not integrated into Octave, analysis results of the Banerjee Test and the Array Data Flow Analysis were compared and evaluated. In the final version, the Array Data Flow Analysis is not used anymore, because many of the quite simple test programs showed no or only little gain from this analysis. Though, for more complex programs this analysis has its justification.

---

\(^2\)Because the original publications contains some errors in these equation, their correct form is given here. For more extensions and improvements of the framework see [16].
Chapter 3

Infrastructure

This chapter aims to show the hardware and software infrastructure used during this project. First, an overview of the Philips TriMedia TM-1x00 which was used as hardware platform is given. Then, a short description of the experimental Octave compilation system follows. Finally, some benchmark programs from the DSPstone suited are explained in more detail than in the previous overview chapter.

3.1 Philips TriMedia TM-1

The TriMedia TM-1 is a single processor DSP tailored for Multimedia applications. Figure 3.1 shows a block diagram of the TriMedia Architecture. Functional units for Video-In, Video-Out and Audio-In and Audio-Out reflect its suitability for the multimedia domain. All components are connected to an internal bus that allows for high-bandwidth transfers. For the communication with peripheral devices there are interfaces such as an I²C interface, a synchronous serial interface and a PCI interface. The main components for performing computations are the actual VLIW CPU, the Image Coprocessor and a Variable-Length Decoder (VLD) that can decode Huffman-coded bit-streams autonomously. These bit-streams occur e.g. in the MPEG-2 video/audio format that was explained in chapter 2.2.2. The CPU itself is speeded up by an instruction and a data cache. Additionally, the caches can help to reduce the bus traffic. Between the internal bus and the external SDRAM a main memory interface takes over control.

The VLIW CPU has 32-bit registers and data paths. It contains 27 functional units that work in pipelined mode. Effectively, most operations take therefore only a single cycle to finish. Some more complex operations may take longer. The register set is homogeneous and contains 128 general-purpose registers. There are no register files and every register is freely available to all functional units. The instruction execution is based on a VLIW model. So there is no
need for any scheduling hardware, because scheduling is done statically by the compiler. In each cycle up to 5 instructions can be issued, and these instructions may access up to 5 different functional units simultaneously. Figure 3.2 shows the available issue slot, their functional units and latencies. Certain restriction exists in the choice of what operations can be packed into an instructions. For example, no more than two memory operations can be started simultaneously, because their are only two available memory units. The CPU supports SIMD Multimedia operations as explained earlier in chapter 2.1.1.

The TriMedia TM-1 supports Conditional Execution. This means, that all operations can be “guarded” by some condition, and only if the condition is fulfilled the operation is executed. If the condition is not fulfilled, the operation is skipped and remains without effect. The guard expressions are actually registers which hold results of previous comparison, ... operations. Depending on their outcome, the guarded operation is executed or not. The purpose of Conditional Execution is to reduce penalties resulting from ordinary branch instructions, because guarded operations do not cause pipeline stalls. For more information on exploiting conditional instructions in code generation see Leupers [29].
3.2. Octave

Octave is an experimental compilation system developed at the University of Edinburgh. Rather than being a full compiler, it is a front-end supporting different input languages. At the moment of writing, there is support available for Fortran, C, C++ and Java. Octave itself is written in C++ and exploits the features of the object-oriented programming style. After parsing an input file, Octave converts it into an Abstract Syntax Tree (AST) and offers methods for traversing and manipulating its elements.

For this project, only parts of Octave relevant to the C input language were used. Other programming languages are uncommon for DSP programming, although Java is an alternative for low-performance embedded systems such as microcontrollers.

Mainly, methods for manipulating statements and expressions were used in this project. These are encapsulated in Statement and Expression classes and unify data structures and methods. Additional code that realises analyses and optimisations is based on the classes provided by Octave and therefore easily access the entire AST.

Octave can be extended by other libraries available for C++. Nonetheless, the current version of Octave sometimes clashes with libraries that use the
same class names. Hopefully, in later version this problem will be solved. One library that has been used to extend Octave is the Graph Template Library that is described in the next section.

3.2.1 Graph Template Library

The Graph Template Library (GTL) [15] is a collection of data structures and algorithms which are frequently used in graph algorithms. In its architecture and application program interface (API) the GTL is very similar to the Standard Template Library (STL). Because the STL does not support graphs, the GTL attempts to fill this niche.

The most important data structure offered by the GTL is the Graph. A graph can be either directed or undirected. Graphs support creation and deletion of nodes and edges, but also hiding and restoring of nodes and edges, their traversal and labeling is actively supported. Nodes and edges are modeled by separate classes. Further data structures are PQ-Trees and some elementary container classes.

The GTL contains some basic graph algorithms like Depth-First-Search (DFS), Breadth-First-Search (BFS), Biconnectivity-Test and Topological Sorting. Additionally, more complex algorithms such as graph Bi-Partitioning (Fiduccia/ Mattheyses and Wei/Cheng), ST-Numbering, Maximum-Flow algorithms (Edmonds/Karp and Malhotra/Kumar/Maheshwari) and Planarity-Test are offered.

In this project the GTL has been used to implement Data Dependence Graphs and Control Flow Graphs and algorithms operating on them.

3.3 TriMedia Software Development Environment

The TriMedia Software Development Environment (SDE) contains an optimising C/C++ compiler, an assembler, a linker, a debugger, a profiler and a simulator. A VLIW scheduler is a separate program that supports the compiler as well as the assembler.

The compilation system was used to create object files from C source files for the TriMedia TM-1. The simulator that is able to provide cycle-accurate timings was used to obtain the performance figures presented in later chapters of this thesis.

The compiler offers the choice between four different optimisation levels. At the lowest level (O0) no optimisation at all is performed. Level 1 (O1) enables decision-tree local optimisations, in level 2 (O2) an additional inter-decision-tree register allocation is enabled. Level 3 (O3) adds some further global optimisations.
3.3. TriMedia Software Development Environment

Local optimisations supported by the TriMedia compiler are

- Local Copy Propagation,
- Local Common Subexpression Elimination,
- Local Constant Folding, and
- Local Load Eliminations.

Global optimisation work within a single decision tree as well as between decision trees. The global optimisations are

- Global Copy Propagation,
- Global Common Subexpression Elimination,
- Global Constant Folding,
- Global Load Elimination,
- Determination of Induction Expression,
- Expression Tree Height Reduction,
- Back Arc Common Subexpression Elimination,
- Back Arc Load Elimination,
- Loop Invariant Code Hoisting,
- Loop Redundant Stores Down Hoisting,
- Code Placement,
- Unreachable Code Elimination,
- Dead Code Elimination, and
- Load Pipelining.

Additionally, some alias analysis is performed. For more information on ordinary optimisations see [44, 1, 6], and on specific optimisation of the TriMedia SDE see [50].
3.4 DSPstone

The DSPstone benchmark suite supplied the set of test programs that were used for performance evaluation during this project. This suite was used because its programs are typical for DSP applications in their computational characteristics and their style of coding. On the other hand, these programs are sufficiently small to be run on a (software-based) simulator in acceptable times. Furthermore, correctness of analyses and optimisations can still be checked manually. The small size of the programs is unfortunately also their biggest disadvantage. Modern multimedia applications are complex programs and contain much more code than a few lines of kernel loops. In this way, performance data can only be obtained for small fractions of “real” code and small sets of data rather than whole applications.

Figure 3.3 shows the source code of the *Convolution* program of the DSPstone benchmark suite. It is a typical representative for the suite, although there are several more complex programs as well. The program offers the possibility of parameterising the type of its data, its storage class and the length. The length actually influences the total amount of used memory and the loop ranges of its loops. The data type has a more indirect influence on the execution speed. By reducing the data size to values smaller than a machine word, SIMD instructions might become feasible to perform several operations simultaneously. The storage class can influence the work of the register allocator. If no storage class is given, it is the task of the compiler to decide which values to keep in registers and which to store at memory. The excessive use of pointers for traversing arrays becomes apparent. Although pointer assignments and pointer arithmetic are simple, it is interesting to see how a compiler can handle such constructs. For profiling purposes, additional instructions can be inserted just before and after the loop in the main program.
#define STORAGE_CLASS register
#define TYPE int
#define LENGTH 16

void pin_down(TYPE * px, TYPE * ph)
{
    STORAGE_CLASS TYPE i;

    for (i = 0; i < LENGTH; ++i)
    {
        *px++ = 1;
        *ph++ = 1;
    }
}

TYPE main()
{
    static TYPE x[LENGTH];
    static TYPE h[LENGTH];

    STORAGE_CLASS TYPE y;
    STORAGE_CLASS TYPE i;

    STORAGE_CLASS TYPE *px = x;
    STORAGE_CLASS TYPE *ph = &h[LENGTH - 1];

    pin_down(&x[0], &h[0]);

    y = 0;

    for (i = 0; i < LENGTH; ++i)
    {
        y += *px++ * *ph--;
    }

    return ((TYPE) y);
}

Figure 3.3: Convolution program of the DSPstone benchmark suite
Chapter 4

Pointer Clean-up Conversion

This chapter presents a method for conversion of a restricted class of pointer-based memory accesses into array accesses with explicit index functions. C programs with pointer accesses to array elements, data independent pointer arithmetic and structured loops can be converted into semantically equivalent representations with explicit array accesses.

Using some motivating examples, we present the application of the pointer conversion as well as the potential benefits for further compiler analysis and optimisation phases. In the next section we give an overview of the basic idea of the method, before assumptions and restrictions are given. Finally, the algorithm is presented and discussed.

4.1 Motivation

Pointer accesses to array data are quite common in typical DSP programs. Since early compilers were not able to generate code that uses available AGUs efficiently from sources containing explicit array references, programmers started to code their programs using pointer-based accesses and pointer arithmetic in order to give “hints” to the compiler on how and when to use post-/pre-increment/decrement addressing modes. A typical example is the kernel loop of the *DSPstone* benchmark **biquadN_sections** in figure 4.1. Each ordinary pointer access is translated into a simple memory access. However, from pointer accesses with immediately following pointer increments corresponding post-increment accesses are generated.

If there are no more optimisation passes to be applied by the compiler, in particular memory access optimisations, use of pointer-based array accesses and their direct translation into assembly/machine code utilising available AGUs is easy and efficient. On the other hand, the necessary analyses supporting the optimisations with the required information become much more difficult if there are further optimisations that need information on access patterns or on the
int w, f;

int *ptr_coeff, *ptr_wi1, *ptr_wi2;

int wi[2 * NumberOfSections];
int coefficients[5 * NumberOfSections];
int x, y;

ptr_coeff = &coefficients[0];
ptr_wi1 = &wi[0];
ptr_wi2 = &wi[1];

x = pin_down(x, coefficients, wi);

y = x;

for (f = 0; f < NumberOfSections; f++)
{
    w = y - *ptr_coeff++ * *ptr_wi1;
    w -= *ptr_coeff++ * *ptr_wi2;

    y = *ptr_coeff++ * w;
    y += *ptr_coeff++ * *ptr_wi1;
    y += *ptr_coeff++ * *ptr_wi2;

    *ptr_wi2++ = *ptr_wi1;
    *ptr_wi1++ = w;

    ptr_wi2++;
    ptr_wi1++;
}

Figure 4.1: Original pointer-based loop
availability of values at certain nodes in the program. Techniques for analysing array accesses have a long tradition in compiler construction and are well developed and therefore often incorporated in commercially available compilers for general purpose computers and, more recently, for DSP. These techniques rely on explicit array index representations and cannot cope with pointer references. In order to maintain semantic correctness compilers use conservative strategies, i.e. many possible array access optimisations are not applied in the presence of pointers. Obviously, this limits the maximal performance of the produced code.

It is highly desirable to overcome this drawback, but without losing the benefit gained from using the AGUs efficiently which was the initial reason for the introduction of pointer-based accesses. The basic idea is to collect information from pointer-based code to regenerate the original accesses with explicit indexes that are suitable for further analyses. The critical performance can be retained by using methods for generating optimal AGU code even for code with the latter kind of array accesses. Such techniques have been developed [7, 28] since early compilers tempted programmers to “abuse” pointers.

```c
 int w, f;
 int wi[2 * NumberofSections];
 int coefficients[5 * NumberofSections];
 int x, y;

 x = pin_down(x, coefficients, wi);
 y = x;

 for (f = 0; f < NumberofSections; f++)
 {
     w = y - coefficients[(5 * f + 0)] * wi[(2 * f + 0)];
     w -= coefficients[(5 * f + 1)] * wi[(2 * f + 1)];

     y = coefficients[(5 * f + 2)] * w;
     y += coefficients[(5 * f + 3)] * wi[(2 * f + 0)];
     y += coefficients[(5 * f + 4)] * wi[(2 * f + 1)];

     wi[(2 * f + 1)] = wi[(2 * f + 0)];
     wi[(2 * f + 0)] = w;
 }
```

Figure 4.2: Loop after conversion of pointer-based accesses into explicit array accesses

Figure 4.2 shows a loop with explicit array indexes that is semantically equiv-
4.2. Related Work

Alent to the previous loop in figure 4.1. Not only it is easier to read and understand for a human reader, but also easier to analyse by ordinary compilers. It is now easy to detect that array accesses to \( w_i \) are involved in several loop independent anti- and input-dependences and that there are no loop carried dependences with respect to \( w_i \) and coefficients. This information can be used for eliminating redundant memory accesses as well as for software-pipelining or scheduling and many other optimisations.

After pointing out all the benefits of array access analysis as opposed to pointer analysis, it may appear that array data dependence analysis is in general “easier” or “more powerful” than pointer analysis. This is not the case! Both problems are without further restrictions intractable [42]. It is just easier to find suitable restrictions of array data dependence problem while keeping the resulting algorithm applicable to real-world programs, i.e. without restricting it to an overly small subset of cases on the one hand and without consuming enormous amounts of resources (time, memory) on the other hand. Hence, array analysis are well researched and developed and have found their way into commercially available compilers whereas pointer analysis techniques are still uncommon, although existent, and are mainly found in experimental prototype compilers.

4.2. Related Work

Allan and Johnson [3] use their vectorisation and parallelisation framework based on C as an intermediate language for induction variable substitution. Pointer-based array accesses together with pointer arithmetic in loops are regarded as induction variables that can be converted into expressions directly dependent on the loop induction variable. This approach does not regenerate the index expressions, but it supplies equivalent pointer expressions. Their main objective is to produce loop representations that are suitable for vectorisation. Therefore, their method only treats loops individually rather than in a context of a whole function. The approach is based on a heuristic that mostly works efficiently, although backtracking is possible. Hence, in the worst case this solution is inefficient. No information is supplied about treatment of loop nests and multi-dimensional arrays.

In his case study of the Intel Reference Compilers for the i386 Architecture Family Muchnick [44] mentions briefly some technique for regenerating array indexes from pointer-based array traversal. Unfortunately, no more details including assumptions, restrictions or capabilities are given.

The complementary conversion, i.e. from explicit array accesses to pointer-based accesses with simultaneous generation of optimal AGU code, has been studied by Leupers [28] and Araujo [7].
4.3 Basic Idea

Pointer clean-up conversion uses two stages during processing. In the first stage information on arrays and pointer initialisation, pointer increments and decrements as well as loop properties is collected. The second step then uses this information in order to replace pointer uses by corresponding array accesses and to remove pointer arithmetic completely.

During data acquisition in the first stage the algorithm traverses the control flow graph of a function and collects information when a pointer is given its initial reference to an array element and keeps track of all subsequent changes. Note that only changes of the pointer itself and not of the value of the object pointed to are traced. When loops are encountered, simple pointer changes within the loop body have multiplied effects caused by repeated execution of the loop body. Therefore, information on loop bounds and induction variables is compulsory for the following reconstruction of array index functions. The program analysis step has similarities to abstract program interpretation and creates summary information of pointer-to-array-mappings at each node of the traversed CFG.

The main objective of the second phase is to replace pointer accesses to array by explicit array accesses with affine index functions. Array accesses not only differ from pointer accesses by using the array variable instead of the original pointer variable, but also they have an additional index function. The mapping between pointers and arrays can be extracted from information gathered from pointer initialisation in the first phase. Array index functions outside loops are constant, whereas inside loops they are dependent on the loop induction variables of the corresponding loops. In order to determine the coefficients of the index functions, information on pointer changes based on pointer arithmetic collected during the first stage is used. Finally pointer-based array references are replaced by semantically equivalent explicit accesses, whereas expressions only serving the purpose of modifying pointers are deleted.

The pointer conversion algorithm can be applied on whole functions and can handle one- and multi-dimensional arrays, general loop nests of structured loops and several consecutive loops with code in between. It is therefore not restricted to handling single loops. Loop bodies can contain conditional control flow.

4.4 Assumptions and restrictions

As mentioned above the general problems of array dependence analysis and pointer analysis are intractable. After simplifying the problem by introducing certain restrictions, analysis might not only be possible but also efficient. In order to facilitate pointer clean-up conversion conversion and to guarantee its correctness following assumptions must hold:
4.4. Assumptions and restrictions

1. structured loops
2. no pointer assignments apart from – maybe repeated – initialisation to some array start element
3. no data dependent pointer arithmetic
4. no function calls that might change pointers itself
5. equal number of pointer increments in all branches of conditional statements

Structured loops (see figure 4.3 are loops with a normalised iteration range going from the lower bound 0 to some constant upper bound N. The step is normalised to 1. It is essential for a structured loop that there are no statements within the loop body that change the value of the induction variable i, or leave the loop prematurely, i.e. a goto to some jump target outside the loop body as well as break are not permissible whereas continue is allowed. Structured loops are common as do-loops in Fortran, but because C is less strict and does not prevent loops with side effects, some extra caution is necessary when processing C loops.

```
for(i = 0; i < N; i++)
{
   <Basic Block>
}
```

Figure 4.3: Simple, structured loop

Pointer assignments apart from initialisations to some start element of the array to be traversed are not permitted. In particular, dynamic memory allocation and deallocation cannot be handled because of the potentially unbounded complexity of dynamic data structures. On the contrary, initialisations of pointers to array elements may be done repeatedly and even in dependence on some induction variable, i.e. within a loop construct. Figure 4.4 shows a program fragment with initialisations of the pointers `ptr1` and `ptr2`. Whereas `ptr1` is statically initialised, `ptr2` is initialised repeatedly within a loop construct and with dependence on the outer iteration variable i.

Data dependent pointer arithmetic is the change of a pointer itself (i.e. not the value pointed to) in a way that is dependent on the data processed and which might change from one program execution to the other. Because it is not known in advance which data will be processed by future program runs, the compiler cannot know at compile time which final value the pointer will eventually have. Although there are some powerful methods available e.g. [64] for this problem of pointer or alias analysis, these kind of program constructs are not considered by
int array1[100], array2[100];
int *ptr1 = &array1[5];
int *ptr2;

for(i = 0; i < N; i++)
{
    ptr2 = &array2[i];
    for(j = 0; j < M; j++)
    {
        ...
    }
}

Figure 4.4: Legal pointer initialisations

the pointer clean-up conversion algorithm and are therefore not permitted. In
general, all pointer expressions that can be evaluated statically can be handled.

In a similar way as data dependent pointer arithmetic, function calls might
change pointers involved in the conversion (see figure 4.5). If there are functions
that take pointers to pointer as arguments, the actual pointers passed to the
function itself and not only their content can be changed. Hence, it must be
ensured that no function calls of this type occur within the program fragment
to be converted.

ptr = &array[0];

function1(ptr);  /* OK */

function2(&ptr);  /* not OK */

Figure 4.5: Function calls changing pointer arguments

The pointer conversion can only be applied if the resulting index functions of
all array accesses are affine functions. These functions must not be dependent
on any other variables apart from induction variables of some enclosing loops.
If all pointer increments/decrements are constant, this can be ensured easily.

Finally, the number of increments of a pointer must be equal in all branches
of conditional statements. The compiler cannot determine during compile time
which branch will actually be taken during run-time, so no information on the
total number of pointer increments after leaving the condition statement is
available. Situations with unequal number of pointer increments are extremely
rare and typically not found in DSP code.
4.5. Algorithm

After giving this long list of assumptions, it must be mentioned that overlapping arrays and access to single arrays via several different pointers is explicitly allowed. Because pointer conversion does not change the semantic meaning of a program, but only changes its representation these kind of constructs that often prevent standard program analysis do not interfere with the conversion algorithm.

Not all of the previously mentioned conditions must be necessarily as strict as given here. Some constraints can be relaxed as described below, but for the purpose of clarity and simplicity of the algorithm to be presented the stricter set of assumptions will be used in the outline of the algorithm.

So far, do-while and while-do loops have not been considered. The main reason for doing so is the fact that termination criteria of these kind of loops often tends to be dependent on the content of some variable different from a basic induction variable. In such a case the upper loop bound can be data dependent and might not be evaluated during compile time. If the lower and upper loop bounds can be determined and are known after preceding analysis at compile time, no further obstacles prevent treatment of these loops. Even in case that the upper bound of the iteration space is not known, the loop can be converted. But conversion of the pointer accesses cannot be applied to any code following that loop because the final point in iteration space, which is the index of the corresponding array at further replacements, is not known. If there is no code following the loop or pointer clean-up conversion is only of interest for the code so far, the algorithm can be extended easily to cover the given cases.

4.5 Algorithm

After the coarse outline on how the algorithm works given above a more precise and formal presentation is given in this section.

The presented algorithm is suitable for handling functions with simple loops, one-dimensional arrays and conditional branches. In the interest of simplicity of presentation the algorithm does not cover enhanced features like loops nests. These features will be introduced in the next section.

The algorithm keeps a list of nodes to visit. This list of nodes is obtained by traversing the control flow graph which is supplied as a parameter to the algorithm in preorder. As long as there are nodes in this list, the next one will be taken and processed. Depending on the type of statement, different actions will be started. If the statement is a pointer declaration and initialisation or a pointer assignment, i.e. it has either the form TYPE *ptr = &array[index] or ptr = &array[index], an entry (ptr, array, index, 0) is added to a data structure map containing the actual mappings of pointers to arrays. In this entry ptr is the name or id of a pointer, array the corresponding array, index the index of the element of the array the pointer initially points to, and the
Algorithm 1 Pointer clean-up conversion for CFG $G$

map $\leftarrow \emptyset$
L $\leftarrow$ preorderList(G);
while L not empty do
    stmt $\leftarrow$ head(L);
    removeHead(L);
    if stmt is pointer declaration and initialisation then
        map $\leftarrow$ map $\cup$ (pointer, array, offset, 0)
    else if stmt is for loop then
        processLoop(stmt, map)
    else if stmt is pointer assignment statement then
        if (pointer, array, *, *) $\in$ map then
            map $\leftarrow$ map $-$ (pointer, array, *, *)
        end if
        map $\leftarrow$ map $\cup$ (pointer, array, offset, 0)
    else if stmt contains pointer reference then
        Look up (pointer, array, offset, *) $\in$ map
        if stmt contains pointer-based array access then
            replace pointer-based access by $\text{array}[\text{initial index} + \text{offset}]
        else if stmt contains pointer arithmetic then
            map $\leftarrow$ map $-$ (pointer, array, offset, *)
            calculate new offset
            map $\leftarrow$ map $\cup$ (pointer, array, new offset, 0)
        end if
    end if
end if
end while
last element of the quadruple is the number of increments of the pointer in a loop body. This number corresponds to the step between two consecutive iterations and is initialised to 0. In case of some statement changing a pointer, i.e. a statement containing pointer arithmetic, the expression is evaluated and the entry in the map is updated. If a pointer is dereferenced, the name of the corresponding array and its current offset are looked up in map, and with this information it is easy to replace the pointer-based access by an explicit array access. Loop statements are handled in a separate procedure that receives the loop statement (together with its loop body) and the map as parameters.

The procedure for handling loops is given as algorithm 2. Similar to the basic algorithm the loop handling procedure proceeds a preorder traversal of the nodes of the loop body. Two passes over all nodes are made: The first pass counts the total offset within one loop iteration of pointers traversing arrays, the second pass then is the actual replacement phase. The total offset of a pointer is the absolute sum of all increments and decrements from the beginning to the end of the loop body. Whenever the loop finishes an iteration, the pointers are moved by the number of elements given by this offset. The second phase then takes this information in order to replace the pointer-accesses and all expressions containing pointer arithmetic and to construct affine index functions for the array accesses. The offset of the first pass is the multiplicative coefficient to the induction variable, i.e. the step which is used from one iteration to the other. Single pointer increments or decrements within a loop body affect the additive component of affine index functions. The procedure keeps track of the pointer increments/decrements it encounters and uses the current distance from the value at the entry of the loop body as the local offset when performing the replacement.

The values for the introductory example 4.1 as computed by the algorithm and stored in map are shown in figures 4.6 and 4.7. The final array index functions \( f_j(i) \) are computed as

\[
f_j(i) = I \times i + (\Delta + \delta)
\]

with the total increment \( I \), induction variable \( i \), initial offset \( \Delta \) and local offset \( \delta \). Each of resulting index functions \( f_j(i) \) for all array accesses \( j \) are affine functions.

The algorithm passes every simple node once, and every node enclosed in a loop construct twice. Hence, the algorithm uses time \( O(n) \) with \( n \) being the number of nodes of the CFG. The exact time depends on the number and the size of loops. Improvements in handling loops are possible, e.g. by storing the nodes with pointer accesses for further replacement rather than spending a second pass. However, the algorithm will still run in linear time. Space complexity is linearly dependent on the number of different pointers used for accessing array elements, because for every pointer there is a separate entry in the map data structure.
Algorithm 2 Procedure processLoop(statement stmt, mapping map)

/* count all pointer increments in loop body and update map accordingly */
L = preorderList(loopBody)
while L not empty do
  stmt ← head(L);
  removeHead(L);
  if stmt contains array arithmetic then
    Look up (pointer,array,offset,increment) ∈ map
    map ← map - (pointer,array,offset,increment)
    calculate new increment
    map ← map ∪ (pointer,array,offset,new increment)
  end if
end while
/* replace all pointer increments in loop body according to map */
L = preorderList(loopBody)
while L not empty do
  if stmt contains pointer reference then
    Look up (pointer,array,offset,increment) ∈ map
    Look up (pointer,local offset) ∈ offsetMap
    if (pointer,local offset) ∉ offsetMap then
      offsetMap ← offsetMap ∪ (pointer,0)
    end if
  end if
  if stmt contains pointer-based array access then
    index function ← increment × ind.var. + offset + local offset
    replace pointer-based access by array[index function]
  else if stmt contains pointer arithmetic then
    offsetMap ← offsetMap - (pointer,local offset)
    calculate new local offset
    offsetMap ← offsetMap ∪ (pointer,new local offset)
  end if
end if
end while
for all (pointer,*,*) ∈ map do
  map ← map - (pointer,array,offset,increment)
  new offset ← increment × upper loop bound + offset + local offset
  map ← map ∪ (pointer,array,new offset,0)
end for
int w, f;
int *ptr_coeff, *ptr_wi1, *ptr_wi2;
int wi[2 * NumberOfSections];
int coefficients[5*NumberOfSections];
int x, y;

ptr_coeff = &coefficients[0];
ptr_wi1 = &wi[0];
ptr_wi2 = &wi[1];
x = pin_down(x, coefficients, wi);
y = x;

for (f=0; f<NumberOfSections; f++)
{
   w = y - *ptr_coeff ++ * ptr_wi1;
   w -= *ptr_coeff ++ * ptr_wi2;
   y = *ptr_coeff ++ * w;
   y += *ptr_coeff ++ * ptr_wi1;
   y += *ptr_coeff ++ * ptr_wi2;
   *ptr_wi2++ = *ptr_wi1;
   *ptr_wi1++ = w;
   ptr_wi2++;
   ptr_wi1++;
}

Contents of data structure map:
{(ptr, array, offset, increment),...}

(ptr_coeff, coefficients, 0, 0)
(ptr_coeff, coefficients, 0, 0), (ptr_wi1, wi, 0, 0)
(ptr_coeff, coefficients, 0, 0), (ptr_wi1, wi, 0, 0), (ptr_wi2, wi, 1, 0)
(ptr_coeff, coefficients, 0, 1), (ptr_wi1, wi, 0, 0), (ptr_wi2, wi, 1, 0)
(ptr_coeff, coefficients, 0, 2), (ptr_wi1, wi, 0, 0), (ptr_wi2, wi, 1, 0)
(ptr_coeff, coefficients, 0, 3), (ptr_wi1, wi, 0, 0), (ptr_wi2, wi, 1, 0)
(ptr_coeff, coefficients, 0, 4), (ptr_wi1, wi, 0, 0), (ptr_wi2, wi, 1, 0)
(ptr_coeff, coefficients, 0, 5), (ptr_wi1, wi, 0, 0), (ptr_wi2, wi, 1, 0)
(ptr_coeff, coefficients, 0, 5), (ptr_wi1, wi, 0, 0), (ptr_wi2, wi, 1, 1)
(ptr_coeff, coefficients, 0, 5), (ptr_wi1, wi, 0, 1), (ptr_wi2, wi, 1, 1)
(ptr_coeff, coefficients, 0, 5), (ptr_wi1, wi, 0, 1), (ptr_wi2, wi, 1, 2)
(ptr_coeff, coefficients, 0, 5), (ptr_wi1, wi, 0, 2), (ptr_wi2, wi, 1, 2)
int w, f;
int *ptr_coeff, *ptr_wi1, *ptr_wi2;
int wi[2 * NumberOfSections];
int coefficients[5 * NumberOfSections];
int x, y;

ptr_coeff = &coefficients[0];
ptr_wi1 = &wi[0];
ptr_wi2 = &wi[1];

x = pin_down(x, coefficients, wi);
y = x;

for (f=0; f<NumberOfSections; f++)
{
    w = y - *ptr_coeff++ * *ptr_wi1;
    w -= *ptr_coeff++ * *ptr_wi2;
    y = *ptr_coeff++ * w;
    y += *ptr_coeff++ * *ptr_wi1;
    y += *ptr_coeff++ * *ptr_wi2;
    *ptr_wi2++ = *ptr_wi1;
    *ptr_wi1++ = w;
    ptr_wi2++;
    ptr_wi1++;
}

Contents of data structure offsetMap:
{(ptr, local offset),...}

Array index functions

<table>
<thead>
<tr>
<th>coeff.</th>
<th>wi (1)</th>
<th>wi (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr_coeff,0</td>
<td>5x + 0</td>
<td>2x + 0</td>
</tr>
<tr>
<td>ptr_coeff,1</td>
<td>5x + 1</td>
<td>2x + 1</td>
</tr>
<tr>
<td>ptr_coeff,2</td>
<td>5x + 2</td>
<td>2x + 2</td>
</tr>
<tr>
<td>ptr_coeff,3</td>
<td>5x + 3</td>
<td>2x + 0</td>
</tr>
<tr>
<td>ptr_coeff,4</td>
<td>5x + 4</td>
<td>2x + 1</td>
</tr>
<tr>
<td>ptr_coeff,5</td>
<td>2x + 0</td>
<td>2x + 0</td>
</tr>
</tbody>
</table>

⇒ (ptr_coeff,5), (ptr_wi1,2), (ptr_wi2,2)
4.6 Extensions

So far only simple loops with “plain” loop bodies have been considered. Extensions to the algorithm are shown in this section which among others things enable treatment of general loops nest – and therefore also perfect loop nests –, conditional branches and multi-dimensional arrays.

Not all of the presented extensions have been implemented during this project. So far, only “array parameters” are fully functional and “perfect loop nests” are prototyped, respectively. The other extensions have been developed and tested manually.

4.6.1 Arrays passed as parameters of functions

If no legal pointer assignment can be found, but the pointer used to traverse an array is a formal parameter of the function to be processed, it can be used as the name of the array. Kernighan and Ritchie [24] give following definition:

A postfix expression followed by an expression in square brackets is a postfix expression denoting a subscripted array reference. One of the two expressions must have type “pointer to T”, where T is some type, and the other must have integral type; the type of the subscript expression is T. The expression $E_1[E_2]$ is identical (by definition) to $*(*(E_1)+(E_2))$.

Figure 4.8 shows a function with parameters $\star px$ and $\star ph$. Both pointers can be legally interpreted as pointing to some position inside arrays as done in the original program on the left side. After converting the pointer accesses the program on the right side results which is semantically identical.

```c
void pin_down(int *px, int *ph) {
    int i;

    for (i = 0; i < LENGTH; ++i) {
        *px++ = 1;
        *ph++ = 1;
    }
}
```

Figure 4.8: Example of pointer clean-up conversion of arrays passed as function arguments
4.6.2 Perfect Loop Nests

Perfect loop nests of structured loops are different from simple loops in the way that the effect of a pointer increment/decrement in the loop body not only multiplies by the iteration range of its immediately enclosing loop construct, but by the ranges of all outer loops. Therefore, all outer loops have to be considered when converting pointers of a perfect loop nest.

```
int image[X*Y];
int *pimage = &image[0];

for(i = 0; i < X; i++)
{
    for(j = 0; j < Y; f++)
    {
        *pimage++ = 1;
    }
}
```

```
int image[X*Y];

for(i = 0; i < X; i++)
{
    for(j = 0; j < Y; f++)
    {
        image[(Y*i) + (1*f) + 0] = 1;
    }
}
```

Figure 4.9: Pointer clean-up conversion of perfect loop nests

Figure 4.9 shows on the left side a perfect loop nest with a simple loop body. The pointer `pimage` traverses the array `image` and initialises all elements to 1. When analysing the outer loop, i.e. the loop with induction variable `i`, its loop body is another loop that moves the pointer by `Y` steps. Consequently, each iteration step of the outer loop is multiplied by `Y`. The inner loop contributes to the resulting index function as an additive component dependent on `j` as known from simple loops. Thus, the converted loop has the form as shown on the right side of figure 4.9.

```
for(i_1 = 0; i_1 < N_1; i_1++)
{
    for(i_2 = 0; i_2 < N_2; i_2++)
    {
        ...
        for(i_n = 0; i_n < N_n; i_n++)
        {
            ... *ptr ...
        }
    }
}
```

Figure 4.10: General structure of perfect loop nest with pointer-based array access

Resulting index functions are still affine functions, but they are now functions
for(i₁ = 0; i₁ < N₁; i₁ + +)
{
    for(i₂ = 0; i₂ < N₂; i₂ + +)
    {
        ...
        array[\sum_{j=1}^{n-1}(i_j \times N_{j+1}) + i_n + \Delta + \delta]] ...
    }
}

Figure 4.11: Perfect loop nest after pointer clean-up conversion

in all enclosing induction variables. It is important to note that all loops of the perfect loop nest are structured loops and no iteration range of an inner loop is dependent on any of the enclosing loops. Figures 4.10 and 4.11 show general forms of perfect loop nests and their form after pointer clean-up conversion, respectively.

Handling perfect loop nests does not require extra passes over the loop. It is sufficient to detect perfect loop nest and descent to the inner loop body while keeping track of the outer loops. Once the actual loop body is reached conversion can be performed as usual, but with incorporating the additional outer loop variables and loop ranges as part of the index functions. Hence, asymptotical run-time complexity of the algorithm is not affected.

4.6.3 General Loop Nests

General loop nests differ from perfect loop nests in the possibility for more statements than a single loop statement on each level of the outer loops. General loops nests do not have any restrictions on the number and type of statements their loop bodies might contain. Hence, it is possible that a loop contains several consecutive inner loops among among other non-loop statements. Ordinary statement can occur before, between or after inner loops. Additionally, there are no restrictions that prevent these ordinary statements from being statements containing pointer arithmetic or from re-initialising pointers, i.e. pointer assignments. Figure 4.12 shows the structure of a general loop nest.

In context of general loop nest several problems must be solved:

1. pointer assignments in outer loops,
2. pointer arithmetic in outer loops,
\begin{figure}
\begin{verbatim}
for(i_1 = 0; i_1 < N_1; i_1++)
{
    S_{11}
    for(i_{21} = 0; i_{21} < N_{21}; i_{21}++)
    {
        ...
    }
    S_{12}
    for(i_{22} = 0; i_{22} < N_{22}; i_{22}++)
    {
        ...
    }
    S_{13}
    ...
}
\end{verbatim}
\caption{General loop nest}
\end{figure}

3. consecutive loops with references to same arrays,

4. mixed ordinary and loop statements.

Although this list of problems gives the impression of a quickly rising complexity of a possible algorithm, none of the listed problems is really “hard”. Eventually, even the linear time complexity remains unchanged. Problems 1) and 2) can be solved with generalisations of already used techniques. The bookkeeping of loop information must be extended in order to store information on all enclosing loops. Additionally, it is necessary to keep track of the loops a pointer movement is dependent on. Pointers are not automatically dependent on all outer loops, e.g. when a pointer re-initialisation takes place all outer loops from this point on do not affect the pointer movement. Problems 3) and 4) are actually no new problems, they already occur in functions with simple loops. But this time, loops and statements are not at the top level, i.e. the level of a function body, but on loop level. Therefore, the same distinction between cases as seen in the base algorithm has to be introduced in the loop handling procedure. After finished one loop, the pointer-to-array-mapping must be updated according to the pointer movement in the loop body and the iteration ranges of enclosing loops. With this knowledge ordinary non-loop statements before, between or after inner loops are natural extensions to the framework.

An example is shown in figure 4.13. Since the pointer \texttt{p.x} is re-initialised within the \texttt{i}-loop but outside the \texttt{f}-loop, the access via \texttt{p.x} inside the inner loop body is only dependent on \texttt{f} and not on \texttt{i}. The opposite is true with the pointer access \texttt{p.y}. It is not changed in the \texttt{f}-loop, but in the \texttt{i}-loop. The access to the
4.6. Extensions

```c
int *p_x = &x[0] ;
int *p_h = &h[0] ;
int *p_y = &y[0] ;

for (i = 0 ; i < 3; i++)
{
    p_x = &x[0] ;
    for (f = 0 ; f < 3; f++)
    {
        *p_y += *p_h++ * *p_x++ ;
        y[i] += h[3*i+f] * x[f];
    }
    p_y++ ;
}
```

Figure 4.13: Example of pointer clean-up conversion of general loop nests

array h is dependent on i as well as on f.

4.6.4 Conditional branches

Virtually all programs, not only DSP code, contains not only sequential sequences of instructions, but also conditional branches. Because different branches of such constructs are executed mutually exclusively, effects of all branches on pointer conversion must be considered independently from each other.

An important assumption is that pointer increments must be equal in all branches. This assumption seems to be strict, but it is fulfilled already in nearly all programs. The way arrays are traversed is often static and not dependent on any decision taken at run-time. Thus, there are only a very few programs that changes array traversal dependent on the outcome of some expression evaluation at run-time.

Figure 4.14 shows a loop with a conditional branch before and after pointer clean-up conversion. Each branch contains two references to ptr. The local offset is calculated independently in each branch starting with the value valid at the node of control flow branching, i.e. the if-statement.

The extended algorithm just has to check if the condition of matched pointer increments if fulfilled, and if so counting and replacement can be performed in all branches with the same initial mapping of pointers to arrays. Because all pointer changes must match in all branches, any one of the possible mappings can be chosen as the resulting mapping after leaving the conditional branch construct.


```c
int *ptr = &array[0];

for (i = 0; i < N; i++)
{
    if ( ... )
    {
        ... ptr++ ...
        ... ptr++ ...
    }
    else
    {
        ... ptr++ ...
        ... ptr++ ...
    }
}
```

Figure 4.14: Example of pointer clean-up conversion of conditional branches

Switch-statements are generalisations of conditional branches. Basically, switch-statements are equivalent to cascaded if-statements. For this reason they can be handled in the same way as if-statements, but now more than two branches must be considered.

### 4.6.5 Multi-dimensional arrays

Multi-dimensional arrays in C are in no way different from one-dimensional arrays. They are just nested one-dimensional arrays, i.e. the base type of an n-dimensional array is an (n-1)-dimensional array etc. Nonetheless, differences show up in pointer-based traversal of multi-dimensional arrays. It is not possible anymore to traverse a multi-dimensional array with a single pointer, because movement in any dimension requires pointers of the corresponding type of that dimension. Because of the inconvenience of having pointer of different types, i.e. pointers to actual elements and pointers to “sub-arrays”, multi-dimensional arrays are rarely object of pointer traversal. However, when pointer type information is used, the dimension of the pointer increment can be reconstructed. Figure 4.15 shows an example of the application of pointer clean-up conversion of a 2-dimensional array.

More often, multi-dimensional arrays can be found already linearised in order to enable traversal by a single pointer. Linearisation of multi-dimensional arrays is the mapping of elements of higher dimensions onto some sequential sequence of elements. For example, a 2-dimensional array can be stored as a linear array with elements stored row by row.

Pointer accesses to linearised multi-dimensional arrays cannot be distinguished from “ordinary” accesses to 1-dimensional arrays. Hence, the technique for
4.6. Extensions

int a[10][20];
int (*ptr_y)[20] = &a[0];
int *ptr_x;

for(i = 0; i < 10; i++)
{
    ptr_x = *ptr_y++;
    for(j = 0; j < 20; j++)
    {
        *ptr_x++ = 5;
    }
}

Figure 4.15: Example of pointer clean-up conversion of a 2-dimensional array

pointer clean-up conversion is applicable without any changes. After converting the array accesses a further step that re-transforms the arrays to their original multi-dimensional form is not only possible, but also useful for further analysis and optimisations, e.g. data dependence analysis and automatic partitioning of data and computation.
Chapter 5

Software-Pipelining

Software-Pipelining is a loop optimisation technique that seeks to exploit Instruction Level Parallelism by overlapping consecutive loop iterations in such a way that statements from different iterations are in state of execution simultaneously. Since the performance of VLIW architectures like DSPs is strongly dependent on the compilers' ability to fill most of the wide instruction words with independent operations, this technique plays a key role in DSP code generation.

5.1 Motivation

Software-Pipelining attempts to generate an instruction schedule for a loop in such a way that the total execution time of the loop is minimised. The idea is to take independent operations from subsequent iterations and to execute them in parallel. This of course is only possible if the underlying architecture supports Instruction Level Parallelism.

Figure 5.1 shows a simple loop in its original, i.e. sequential, form and additionally the same loop after applying software-pipelining. On the first view it is not obvious why the software-pipelined loop is superior to the original one. A look at figure 5.2 can make this fact more clear. The loop body of the original loop body is shown in a). If no further optimisations are performed, sequential code is executed. One instruction follows the other, all iterations are executed sequentially, i.e. one iteration is finished before the next one is started. Each iteration takes 6 steps to finish. The total execution time \(^1\) of the sequential loop is therefore \(N \times 6\). b) shows the loop body after performing Local Compaction. Local compaction schedules the instruction of a single loop body in such a way that data independent operations can be executed simultaneously.

\(^1\) All example cost calculations neglect potential branch and jump penalties. The purpose is to demonstrate the possible performance gain rather than giving a cycle accurate loop timing.
for (f = 0; f < N; f++)
{
  t_0 = Z;
  t_2 = A[0];
  t_3 = B[0];
  t_1 = t_2 * t_3;
  t_0 += t_1;
  Z = t_0;
}
t_0 = Z;
t_2 = A[0];
t_3 = B[0];
t_1 = t_2 * t_3;
for (f = 0; f < N-1; f++)
{
  Z = t_0;
  t_0 = Z;
  t_2 = A[1 * f+1];
  t_3 = B[1 * f+1];
  t_1 = t_2 * t_3;
  t_0 += t_1;
}
Z = t_0;

A data dependent instruction must be scheduled after the instruction it is dependent on. If no further assumptions are made on the number and type of instructions that can be executed in parallel, the schedule in b) results. The number of steps required for one iteration has been decreased to 4. Hence, the total number of steps for the entire loop is \( N \times 4 \). Part c) of the figure shows the software-pipelined loop based on the locally compacted loop. Two consecutive iterations can be overlapped as long as data dependences are respected. Although it may appear as if the simultaneous execution of \( Z = t_0 \) and \( t_0 = Z \) might violate data dependence constraints, this schedule is legal under the assumption that all read accesses are performed before any write access. The resulting New Loop Kernel which is framed by a blue box contains more parallel instructions and takes only 3 steps. However, it becomes apparent that the new loop kernel represents some steady state of the loop. Instructions before and after it perform the necessary initialisation and proper end. These code segments are called Prologue and Epilogue, respectively. The new loop kernel takes 3 steps, and the prologue and epilogue together take 4 steps. The loop range is decreased by 1 in order to take of respect of the prologue and epilogue, therefore \( N - 1 \) iterations remain. The total number of steps is \( (N - 1) \times 3 + 4 \). When the number of steps for the locally compacted and the software-pipelined loops are compared, the software-pipelined version is superior for any reasonable number \( N \) of iterations:

\[
\begin{align*}
Cost_{ic} & \geq Cost_{sp} \\
4N & \geq 3(N - 1) + 4 \\
N & \geq 1
\end{align*}
\]  

(5.1)

Because instruction scheduling is a task of a compiler that is done on a very low level, i.e. the intermediate representation used is very machine-specific
a) Sequential Loop

\[
\begin{align*}
    t_0 &= Z; \\
    t_2 &= A[1 \times f]; \\
    t_3 &= B[1 \times f]; \\
    t_1 &= t_2 \times t_3; \\
    t_0 &= t_0 + t_1; \\
    Z &= t_0;
\end{align*}
\]

Iteration 1

b) Locally Compacted Loop

\[
\begin{align*}
    t_0 &= Z; \\
    t_2 &= A[1 \times f]; \\
    t_3 &= B[1 \times f]; \\
    t_1 &= t_2 \times t_3; \\
    t_0 &= t_0 + t_1; \\
    Z &= t_0;
\end{align*}
\]

Iteration 2

c) Software-Pipelined Loop

\[
\begin{align*}
    t_0 &= Z; \\
    t_2 &= A[1 \times f]; \\
    t_3 &= B[1 \times f]; \\
    t_1 &= t_2 \times t_3; \\
    t_0 &= t_0 + t_1; \\
    Z &= t_0;
\end{align*}
\]

Figure 5.2: Software-Pipelining of a simple loop
and detailed machine information is available, software-pipelining is usually performed in some compiler back-end or code generator. The compiler needs to know which machine instructions are available, how many instructions can be issued simultaneously and which latencies these instructions have. The exact and optimal solution to this scheduling is a NP-complete problem and therefore many compilers do not perform software-pipelining at all or generate only sub-optimal schedules.

The approach of software-pipelining taken in this project is different. It assumes a commercial DSP compiler that does not support software-pipelining and whose source code is not available for an integration. In order to support the existing compiler to generate better loop schedules software-pipelining is performed as a Source-to-Source Transformation during a preprocessing stage. The software-pipelining program takes plain C code as input and outputs again C code. This modified code is that used as an input to the actual compiler that generates the object code.

5.2 Basics

Since optimal scheduling a loop is a computationally hard problem, most software-pipelining techniques aim to achieve a reasonably good rather than optimal schedule. Giving up the goal of optimality of the solution in favour of efficient compilation algorithms is just one of the many trade-offs compiler developers have to find. One of the many different efficient software-pipelining methods is Modulo Scheduling [53]. It is characterised by its properties to take the same schedule for each iteration of the loop and to initiate consecutive iterations at a fixed distance to each other. The constant distance between the iteration initiations is the Initiation Interval (II). To achieve the highest benefit from Modulo Scheduling the Initiation Interval should be as small as possible, i.e. the number of overlapped steps of consecutive iterations as high as possible. The smallest possible and still valid initiation interval with respect to data dependences and resource constraints is the Minimum Initiation Interval (MII). After this has been determined, the actual process of scheduling a new loop kernel can be started. Since is a theoretical lower bound, it cannot always be reached in practice. An iteration algorithm starts with the MII and increases it until a valid loop kernel can be generated. Finally, code for the new loop kernel as well as for the prologue and epilogue is created and output.

The algorithm only works for simple (i.e. inner) loops without conditional branches. Conditional branches introduce several possible execution paths through the loop body which all must be considered during software-pipelining. Alternative algorithms with extended capabilities to handle even more complex loops are described in a later section. First, the following sections give an overview of the determination of the MII and construction of a new loop kernel.
5.3 Minimum Initiation Interval

The Minimum Initiation Interval is a theoretical lower bound for the distance of initiations of consecutive iterations and is based on estimations. The MII is influenced by resource constraints such as the number of available functional units of a certain type and the different latencies of the performed operations. Another influence on the MII stems from data dependences between operations in different iterations. An operations must not be initiated before an operation it is dependent on in some earlier iteration is finished. The first component of the MII which is based on resource-constraints is denoted as $ResMII$ and the other one is the recurrence-constraint MII $RecMII$. The final MII is determined as the maximum of $ResMII$ and $RecMII$:

$$MII = \max(ResMII, RecMII)$$  \hspace{1cm} (5.2)

Figure 5.3 shows the Data Dependence Graph (DDG) and a Schedule for the loop of the introductory example. In the DDG the nodes represent instructions of the loop body and the edges dependences between instructions. In this example only true dependences are displayed. However, output and anti dependences must be considered too in the actual computation. The dependence edges are labelled with latencies of the operations. The backwards edge between $Z = t_0$ and $t_0 = Z$ is a loop-carried dependence, the others are loop independent dependences. The resulting schedule for the loop body is given below. If the hypothetical machine has 4 identical functional units, the loop can be scheduled into 5 cycles while latencies (of the multiplication) and data dependences are respected.

5.3.1 Resource-constrained Minimum Initiation Interval

Resource availability imposes constraints on the Minimum Initiation Interval. The Resource-constrained Minimum Initiation Interval ($ResMII$) is determined by summing up all resource requirements of one iteration on the one hand side, and all resources available on the other. Obviously, the available resources must meet the consumption of all operation in one loop iteration. The $ResMII$ is estimated as the maximal consumption of a specific resource type divided by the number of those resources, or

$$ResMII = \max_{i \in FU \text{ types}} \left( \frac{\# \text{ FU type } i \text{ required}}{\# \text{ FU type } i \text{ in hardware}} \right)$$  \hspace{1cm} (5.3)

For the example in figure 5.3 this means to take the number of instructions and divide it by the number of functional units as all functional units are equivalent. Therefore, $ResMII = \frac{6}{4} = \frac{3}{2}$. 
5.3. Minimum Initiation Interval

Figure 5.3: Data Dependence Graph and schedule with resource-constraints
5.3.2 Recurrence-constrained Minimum Initiation Interval

The second kind of lower limit of the MII results from data dependences. Whereas data dependences of a single iteration form a spanning tree of the DDG, loop-carried dependences are backward edges in this graph. Loop independent and loop-carried dependences can form cycles in the DDG. The difference time an instruction is issued and the instruction it is dependent on must be bigger than the sum of latencies on the path between them. Let \( c \) be a circuit in the DDG, \( \text{Delay}(c) \) the total sum of latencies along that circuit and \( \text{Distance}(c) \) the number of different iterations spanned by the dependence circuit. Because all scalar definitions form output dependences with itself, i.e. a value written by some instance of a definition in some iteration is overwritten by the another instance of that definition at some later iteration, the same operation must not be issued earlier than \( \text{Delay}(c) \) later. Or written another way

\[
\text{II} \times \text{Distance}(c) \geq \text{Delay}(c) \tag{5.4}
\]

Because there might be several dependence cycles in a DDG, the RecMII must be a conservative estimation and is therefore defined as the worst-case constraint:

\[
\text{RecMII} = \max_{c \in \text{dependence circuits}} \left( \frac{\text{Delay}(c)}{\text{Distance}(c)} \right). \tag{5.5}
\]

The example in figure 5.3 contains only a single dependence cycle\(^2\). The sum of latencies \( \text{Delay}(c) \) on this cycle is 3, the number of different iterations \( \text{Distance}(c) \) involved in it is 2. Hence, \( \text{RecMII} = \frac{3}{2} \) holds. With RecMII and RecMII together, the MII for the example is calculated as

\[
\text{MII} = \lceil \max(\text{RecMII}, \text{RecMII}) \rceil = \lceil \max\left( \frac{3}{2}, \frac{3}{2} \right) \rceil = 2 \tag{5.6}
\]

With Modulo Scheduling this lower limit cannot be reached. However, this MII gives a good initialisation value for the iterative search for an II that can be practically realised.

5.4 Scheduling a New Loop Kernel

After the MII has been determined, a new loop kernel can be scheduled. Starting with the MII, the algorithm tries to find a schedule that overlaps consecutive loops, respects data dependences and uses no more than the available resources. Not always such a schedule can be found; in such a case the algorithm increases the II by one and starts again. The algorithm terminates when

---

\(^2\)Although there are actually more dependence cycles these are not shown in the figure in order to keep the example simple.
a valid schedule is found or the II has reached a certain threshold. In the latter case, software-pipelining is considered as not profitable and some other loop scheduling technique should be applied.

Figure 5.4 shows the example loop from above and some overlapping of loop iterations. Because Modulo Scheduling constructs a single loop schedule for all iterations and a single II, no II corresponding to the MII of 2 can be found. II must be increased to 4 until a valid schedule is found. The instructions of different iterations are put together into a single scheduling block that corresponds to the generated instruction stream. From this scheduling table, the Prologue, the Epilogue and the new loop kernel can be extracted and output.

For the construction of the new loop kernel a Modulo Resource Reservation Table (MRT) is used. This table stores information on the utilisation of resources as a result of instruction scheduling. The MRT stores for every operation which functional unit is used and for how long it is occupied. Since Modulo Scheduling produces repetitive schedules, the MRT has exactly II rows. Each column represents a functional unit. An entry in some field of the table indicates that the corresponding resource is busy at that time; on the contrary, an empty field stands for some available resource.

The main task of the modulo scheduler is to place the instructions of the new loop kernel into the MRT in an optimal way. Different scheduling algorithms with varying complexity and quality have been proposed. However, most of the times some variant of List-Scheduling is used. List-Scheduling is a one-pass scheduling that schedules operations according operations in a basic block via a topological traversal of its DDG. Independent operations are scheduled as early as possible, whereas data dependent operations are delayed until the operations they are dependent on are finished. Often this algorithm constructs that are near-optimal schedule. Another advantage is the low computational complexity of this algorithm itself, which allows for application even on bigger loops.

The final step is the code generation. The operations in the Epilogue, Prologue and new loop kernel must be compacted and assembled into VLIW instructions which are then output.

### 5.5 Extensions

Since Software-Pipelining is a potentially very powerful loop optimisation, many extensions to the basic algorithm have been developed. Extensions range from handling of loops with conditional branches and software-pipelining of loop nests to more complex techniques for generating schedules with multiple initiation intervals and techniques for generating better local schedules of the new loop kernel. One of the many techniques for handling loops with conditional branches [22] is discussed briefly in next section.
### Figure 5.4: Example of Modulo Scheduling

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Iteration 1</th>
<th>Iteration 2</th>
<th>Iteration 3</th>
<th>Iteration 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.5. Extensions

5.5.1 Software-Pipelining of Loops with Conditional Branches

Software-Pipelining Loops with conditional branches is substantially harder because there are different paths through the loop body which have to be considered. Different combinations of control flow paths can be taken in consecutive iterations, therefore all possible effects of overlapping operations from these paths must be covered by the new loop kernel. Because each path might have its own set of iterations, this could lead to multiple IIIs and different schedules for each branch combination, respectively. An overview of many different approaches to this problem can be found in [22, 58]. At this place, only Predicated Modulo Scheduling that relies on Conditional Execution (see chapter 3.1) is presented.

The two key elements of Predicated Modulo Scheduling are if-conversion and Conditional Execution. In a first step, all conditional branches are converted into guarded operations. This corresponds to a conversion of control dependences into data dependences. The resulting code does not contain any more conditional branches, but guarded operations. The second step is then Modulo Scheduling of the loop with a single basic block as its loop body.

Figure 5.5 shows an example of the if-conversion. On the left side the original program with an if-statement is shown. The right side shows the equivalent program fragment after if-conversion. The expression that is taken to decide which branch to take, is evaluated and its results is stored in P. The following instructions are guarded with either P or its complementary value \(!P\) depending on the branch the instructions originate from. The resulting piece of code is a single basic block, i.e. it contains no branches anymore.

\[
\begin{align*}
\text{if (exp)} & \quad \text{P = exp; } \\
\{ & \\
\text{Statement 1;} & \quad [P] \text{ Statement 1;} \\
\text{...} & \quad [P] \text{ ...} \\
\text{else} & \quad [!P] \text{ Statement 2;} \\
\{ & \\
\text{Statement 2;} & \quad [!P] \text{ ...} \\
\text{...} &
\end{align*}
\]

Figure 5.5: Example of if-conversion

Because many DSPs such as the Philips TriMedia (see chapter 3.1) support Conditional Execution, Predicated Modulo Scheduling can be easily adapted to such an architecture. Benefits clearly arise from the fact that loops with conditional branches can be software-pipelined and therefore speeded up by a comparatively simple extension to the basic algorithm. However, Predicated Modulo Scheduling has some disadvantages, too. Because there are instructions with complementary predicates, only one group is executed at a time.
The other group is completely ignored. Although these instructions are not actually executed, they are assigned to some functional unit that is not available anymore to other instructions at that time. The overall utilisation of the processor resources decreases as a result. An additional effect is the increase of the MII to a larger value than that of any of the individual paths alone. This is caused by the fact that the if-converted loop is a single basic block with as many instructions as the sum of the instructions of both branches before the conversion. The resource consumption of this larger basic block must be fulfilled by available functional units, which results in a larger ResMII. This larger ResMII seems not to be a major drawback as Hu [22] refers to results that indicate that the with Predicated Modulo Scheduling achievable II is often close to the MII.

For this project, Predicated Modulo Scheduling was not considered because of its dependence on the ability to generate Predicated Operations. The actual Software-Pipelining approach in this project is on Source Level, i.e. C programming language, and there are no equivalent constructs available. Although, the conditional expression (exp1 ? exp2 : exp3) can be compiled into Predicated Operations, there is no guarantee that this will really be done by the compiler.

5.6 Implemented Software-Pipelining Algorithm

During this project a Modulo Scheduling Software-Pipeliner on Source-to-Source level has been implemented and integrated into Octave. Due to the different level, i.e. source level as opposed to machine-specific level, some adaptations of the presented basic method were necessary. These concern mainly the reduction of complexity of C expressions and the modelling of a machine instruction set in C. The main ideas are based on previous works by Su et al. [60] and were extended and changed to study the effects in a different environment.

In the following an overview of the different stages of the software-pipeliner are given. The individual stages are explained briefly.

1. Pointer Clean-up Conversion
   The Pointer Clean-Up Conversion is the transformation described in chapter 4 and serves the purpose to make data dependence analysis easier. After Pointer Clean-up Conversion Array Data Dependence Analyses can be applied.

2. Decomposition into Pseudo-3-Address-Form
   This steps breaks down C expressions into much simpler expressions and statements that resemble to 3-address machine instructions. Since there is a big gap between potentially arbitrary expressions in C and simple 3-address instructions of the DSP, this step attempts to reduce this gap. The
5.6. Implemented Software-Pipelining Algorithm

expected benefits from the expression decomposition are increased flexibility at the scheduling stage and more precise instruction cost modelling. The scheduler cannot freely move expressions if these are very complex and show many data dependences. By decomposing them into simpler expressions, these intermediate expressions are much easier to move. On the other hand, cost modelling for complex expressions is complicated. It becomes much easier, if the statements and expressions correspond closely to "real" machine instructions. For different instruction types, specific costs in order to model latencies can be introduced\(^3\)

3. Data Dependence Analysis

During Data Dependence Analysis a Data Dependence Graph (DDG) is constructed. Different algorithms are used to take account of scalar and subscripted variables.

- **Scalar Dependences**
  
  For scalar variable a full data flow analysis is used. It determines True, Anti and Output Dependences. Potentially the data flow analysis can be used for programs with arbitrary control flow. But the Software-Pipelining step is restricted to simple loops the full power of this analysis is hardly used. Its flexible approach of implementation allows for future re-use. Although an automatic *Program Analyzer Generator (PAG)* \(^4\) was available, the scalar data flow analysis was implemented from scratch in C++ because the complexity of the available tool is quite high and so the expected training period.

- **Array Dependences**
  
  Array Dependence Analysis seeks to determine data dependence relations between array accesses. As already outlined in chapter 2.3 Memory Disambiguation can be distinguished from Array Data Flow Analysis. During the project an Array Data Flow Analysis has initially been compared to a Memory Disambiguation Method. Because for the very most cases both methods could prove independence of some array accesses between iterations with the same precision\(^4\) of the solution, the Banerjee Test – a Memory Disambiguation method – was finally integrated into Octave.

4. Software-Pipelining The implemented software-pipelining method is a Modulo Scheduling procedure. It considers each C statement as an instruction that can be scheduled individually. Because in a previous the complexity of C expressions and statements has been reduced to Pseudo-3-Address-Form, the pipeliner has much flexibility even for smaller loops

\(^3\)The original plan of utilising a cost model based on the Pseudo-3-Address-Form was abandoned during the project because it would have made the software-pipelining worse. Nonetheless, it is just a simple step to integrate it for other applications based on this program representation.

\(^4\)Array Data Flow Analysis and Memory Disambiguation usually achieve results at very different precision. Due to the relative simplicity of the used benchmark programs that showed very little loop-carried array dependences the results were similar. For more complex programs the Array Data Flow Analysis is more capable of achieving better results.
with only a few original statements. The Abstract Syntax Tree is traversed until a for-loop construct is found. while- and do-while-loops are ignored, because they have no structured loops. If the for-loop construct is a simple structured loop, this loop can be processed directly. Otherwise, if the loop is a loop nest, its inner loop is determined. Only this innermost loop will be pipelined. The first step of the software-pipelining process is to generate a locally compacted loop representation. Each statement is scheduled as early as possible. Statements that have intra-loop dependences on other statements are delayed until all statements it is dependent on are scheduled. The next step is to determine the II. The initial II is incremented until some overlapping of consecutive loop iterations – two or more, depending on the amount of overlapping – is found that respects all inter-loop dependences. When an II can be determined that is smaller than the length of the compacted loop body, the new loop kernel is constructed. If no such II exists, the software-pipelining process is abandoned. The construction of the new loop kernel is basically the same as shown in figure 5.4. The new loop kernel is constructed from all overlapped iterations. If operations to be scheduled contain array accesses the iteration the operation is taken from must be considered. The number of the iteration in the overlapped block is added to the constant part of the affine index function. In this way e.g., an array access \( a[i] \) might become \( a[i+2] \), if the instance of that access is located in the third iteration (i.e. iteration number 2). Finally, the software-pipelined loop is output. The new loop with adjusted loop bounds are placed in-between the Epilogue and Prologue. Because the output language C does not support constructs for generating parallel instruction blocks, the new loop kernel is output as a sequential list of instructions. Beginning with the statements of the earliest cycle, all statements of that cycle are written in order of ascending iteration numbers. This guarantees that the implicit assumption that every read access is performed before any write access in each cycle is obeyed. When all loops of the program are processed, the entire program is written back into a C output file. This file can be used as an input file for any C compiler. The loop transformation might make it easier to create better schedules although no direct transfer of information on the independent operations that can be scheduled in parallel can be transferred to that compiler.

The original idea to model machine characteristics such as latencies of different operations and the issue width at the very high source level was abandoned during the project. Because performance figures showed that additional constraints imposed to the scheduling process would not generate better solutions, no further time was spent on integrating this part. The free schedule, i.e. the schedule only respecting data dependences and assuming unbounded issue width and only single-cycle latencies, is superior to more restricted schedules and therefore it is the only one generated by the software-pipeliner.
All parts of the program are implemented in C++, because the Octave frontend is written in C++. Extensive use of the object-oriented programming style during the project allowed for modularity and simplified design, implementation, test and integration of classes. Additionally, changes of the implementation at later stages of the programming process did not influence other classes because the class interfaces remained unchanged. The Pointer Clean-up Conversion, the Expression Decomposition and the Software-Pipelining are separate programs. This separation keeps the individual programs simple and easier to maintain than a monolithic program. Furthermore, all transformations can be applied individually and in any order.
Chapter 6

Empirical Results

During this project a Software-Pipelining algorithm on Source-to-Source Level has been implemented. Because it is hard to reason about the behaviour of a complex system such as a compiler in theory alone, a series of experiments has been conducted in order to quantify the effects of the software-pipelining loop transformation.

6.1 Tests and Goals

The environment for all tests consisted of the programs developed during this project (Pointer Clean-up Conversion, Statement and Expression Decomposition, Software-Pipelining), the Philips TriMedia Compiler and Simulator, and the DSPstone benchmark suite. First, some transformations on source level were applied, before the transformed program was used as input to the TriMedia compiler. This compiler generates object code that can be evaluated with help of the TriMedia simulator. The simulator generates profiling output that was used for run-time comparisons. Because the TriMedia compiler does not generate any additional assembler files, the actual machine code was not analysed with respect to the instruction scheduling and final code size. Only the number of clock periods for the individual loops were measured.

The aim of the test series was to get empirical results on how the different transformations, in particular software-pipelining, influence the performance of a DSP system, i.e. DSP applications compiled with a C compiler for a DSP processor. Because DSP applications differ significantly from other applications, it is important to observe the effects for this type of applications rather than for programs of different domains or artificial program constructs that can be easily chosen to produce virtually every desired result. In order to get meaningful results, all transformed programs were compared to the original unmodified program. The transformations were applied individually as well as in combinations to learn more about their individual behaviour and their
interaction with other optimisations that are part of the TriMedia compiler. The individual behaviour of a transformation can only be observed if no other optimisations blur the results. On the other hand, some transformations rely on other optimisations or enable or disable them. Therefore, all tests were repeated for all available optimisation levels of the TriMedia compiler. Because software-pipelining creates some static overhead in form of a loop prologue and epilogue, for some programs a performance gain can only be expected after a certain number of iterations. To analyse the dependence of the program performance on the loop range, some tests were repeated with different numbers of iterations. Another often used source level optimisation is loop unrolling. Because loop unrolling serves similar purposes than software-pipelining, in some tests these two techniques were compared.

6.2 Observations and Results

In this section the results obtained during the empirical evaluation are presented. Because the Pointer Clean-up Conversion is a transformation that not only supports the data dependence analysis of the software-pipeliner, but also has its justification in context of other optimisation, it is presented separately.

6.2.1 Software-Pipelining

Figure 6.1 shows the performance figures for the programs biquad_N_sections, n_real_updates, convolution and fir of the DSPstone benchmark suite. For all programs the run-time of the unmodified program and after clean-up and decomposition alone, after clean-up and composition, after clean-up, decomposition and software-pipelining and after decomposition and software-pipelining are shown. Additionally, each diagram shows the run-times for each of these tests for the optimisation levels 00, 01, 02 and 03 (compare chapter 3.3).

For the lower optimisation levels the programs after clean-up conversion usually take less cycles than the unmodified programs. An exception is convolution; although without any further optimisation the cleaned-up program is superior, the original programs performs better at optimisation level 01. At higher levels the original programs differ not so much from the programs after clean-up conversion. Performances can be equal (n_real_updates), better (fir) or worse (biquad_N_sections). If one version outperforms the other at a certain level, it does not mean that this is true for the next higher level. For these four programs the pointer clean-up conversion is able to increase performance significantly at lower optimisation levels, and at higher levels performance is similar to that of the pointer-based code.

The introduction of the decomposition step usually decreases the performance at the lower optimisation levels. For the higher levels this penalty does not exist any more. Whereas the decomposition by itself has no justification since
Figure 6.1: Results for biquad_N_sections, n_real_updates, convolution and fir
6.2. Observations and Results

it is only a supporting technique for the software-pipelining, it is important to know that it does not deteriorate performance of the latter at higher optimisation levels. The main reason for the decreased performance at level 00 is the introduction of many temporary variables and the resulting high number of register to register copy operations. After the TriMedia compiler applies Copy-Propagation these copy operations do not exist any more.

Because the software-pipelined programs are first transformed by the decomposition transformation, they all perform badly without any further optimisations. At level 00 there is still the above described penalty. For higher optimisation levels the results become more realistic. In general, software-pipelined programs do not perform better than the original program. The run-times of the original programs and the software-pipelined programs are equal or close to each other. Surprisingly, no real benefit is achieved from software-pipelining.

Different version of the n_complex_updates program are shown in figure 6.2. In addition to the comparison of the different transformations and optimisation levels, the number of iterations is varied. Furthermore, 4-way loop unrolling was applied to the version with 1024 iterations. The three version of n_complex_updates with different numbers of iterations (16, 64, 1024) do not show significant changes over the increase of the loop range. Of course, absolute times increase, but the relations between the original and transformed programs remain unchanged. For this program there is no influence of the number of iterations on the performance. The overhead of the loop prologue and epilogue is either not important at all or only significant for very small loop ranges. For n_complex_updates software-pipelining shows some positive effect.

At higher optimisation levels the software-pipelined program in pointer-based representation performs better than the unmodified program. The number of cycles can be reduced by up to 50% with decomposing and software-pipelining this program. Because this program contains many pointer-based memory references that are hard to analyse, the original program suffers from pipeline stalls due to memory access latencies. After software-pipelining, some of these latencies can be hidden by other operations. When software-pipelining is compared to 4-way loop unrolling, software-pipelining is better at higher optimisation levels whereas unrolling improves the program without optimisations. At the higher levels - when software-pipelining is not influenced by the decomposition any more - software-pipelining achieves run-times of about 50% of the unrolled program. Again, the pointer representation prevents the optimiser to take advantage from a usually very powerful program transformation. Interestingly, the performance of the cleaned up code is without exception much better than the original code. In this program pointer clean-up conversion alone is as powerful as software-pipelining at higher optimisation levels. Additionally, pointer clean-up conversion already improves the code at lower levels very much.
Figure 6.2: Results for different loop ranges of n_complex_updates
6.2.2 Pointer Clean-up Conversion

Because Pointer Clean-up Conversion is not only useful for software-pipelining, but also for other fine-grain and coarse-grain parallelisation techniques, its influences on program performance are evaluated individually. In addition to the programs shown in the previous section, more tests with programs from the DSPstone benchmark suite were carried out.

As already seen in figure 6.2, the pointer clean-up conversion has a big potential for improving the performance of DSP programs. In that example an improvement of up to 68% was achieved (64 iterations, 01, from 54 to 17 cycles). In this case the pointer references prevent the TriMedia compiler from successfully performing its dependence analysis. This analysis is crucial not only for software-pipelining, but also for VLIW code generation. After the clean-up conversion, already relative simple array dependence analyses can be used to determine the dependences. The instruction scheduler that uses this information can produce more efficient code.

Figure 6.3 shows the run-times of the matrix1, matrix2, lms and matrix3 programs. The original version is compared to the cleaned version for all available optimisation levels. For the levels 00 and 01 all cleaned programs take less cycles than the corresponding original programs. The difference can be up to 50%. For higher optimisation levels the differences become smaller. At level 02, in particular, the original programs show either better or the same performance. On the contrary, at level 03 the cleaned version again perform better. In general, with pointer clean-up conversion some substantial benefit can be achieved without the need for further optimisations. Because all four programs are not too complex, the original programs can perform as good as the transformed programs at higher levels. But as seen in figure 6.2, for more complex programs the conversion provides some substantial advantage.

6.2.3 Other Tests

Further tests have been conducted to compare Memory Disambiguation and Array Data Flow Analysis as well as to examine the behaviour of the software-pipelining approach in combination with some other target architecture and compiler. Because of restrictions in the available time for this project and machine resources, these tests had a more experimental rather than systematic nature.

In order to compare the Banerjee test as a Memory Disambiguation method and a full Array Data Flow Analysis (see chapter 2.3) implementations of both methods were tested against each other. The Banerjee test was integrated in Octave, whereas the Array Data Flow Analysis was a prototyped stand-alone program. Dependences determined by both approaches were compared to each other. After some tests with programs from the DSPstone benchmark suite,
Figure 6.3: Results of pointer clean-up conversion for `matrix1`, `matrix2`, `lms` and `matrix3`
it became clear that both approaches provide equal results most of the times. Because the benchmark programs do not contain conditional branches in their loop body their data flow relations are quite simple. Additionally, many of the DSPstone benchmark programs do not show loop-carried dependences. In such cases, it is not a big challenge to simple Memory Disambiguation tests to prove independence of different iterations. A more sophisticated Array Data Flow Analysis cannot provide more accurate results for such programs. Other Multimedia applications with more complex loop bodies and data dependence relations could benefit more from better analyses, but no further tests were carried out.

To learn more about the effects of software-pipelining on source level, a completely different target architecture was chosen. The target was the Intel Celeron, a super-scalar general-purpose processor. The code for this architecture was generated by the GNU-C compiler. Partly, the results varied from the TriMedia performance figures. In particular, cases could be identified in that the transformed programs performed better or worse than the original programs although the contrary was true for the other architecture/compiler. Because of the complexity of the architecture/compiler system it cannot be clearly determined what the actual cause for this behaviour is. The interaction between compiler phases and architectural features make it hard or impossible to find a single reason for some specific behaviour. It is more the complex interaction that often prevents accurate predictions of program performance in a changed environment.

6.3 Interpretation

Software-Pipelining on Source-to-Source level has the potential to increase the performance of a certain class of DSP programs. When loop bodies contain many pointer-based array references that cannot be analysed successfully by a commercial compiler, software-pipelining can support hiding memory latencies and instruction scheduling. On the other hand, if the loop bodies are simple and contain only a few array accesses, these can be analysed and optimised by techniques integrated in manufacturers’ DSP compilers more successfully than on the very high source level. During the last years DSP manufacturers constantly improved their compilers and integrated many more sophisticated analyses and optimisation techniques. Whereas source level transformations were a promising approach to increasing code quality of DSP compilers a few years ago [60], the situation has changed in the meantime. For example, the TriMedia compiler can perform simple pointer alias analysis and load-pipelining. The first technique can provide dependence information on simple and regular pointer accesses whereas the latter is a simplified form of software-pipelining. Both techniques compete with source level transformations; alias analysis can prove independence of statements that can be scheduled with load-pipelining at a level with very machine-specific information. For many programs, in particular
programs with simple loop structures, these built-in techniques are superior to the external source transformations. For more complex situations a high level approach to software-pipelining is still advantageous, because the higher level of abstraction can help to extract the desired dependence information more easily. Not only compiler technology has strong influences, but also the machine architecture. The TriMedia CPU has an instruction and data cache. The latter is uncommon for a DSP. The effects of some program transformations are very hard to predict in presence of a data cache. Therefore, often the only way to reason about the effect of a transformation is actually to apply the transformation and examine its effects by program execution. In general, the effects of source level transformations are subject to complex interactions between other optimisations and the architecture.

The Pointer Clean-up Conversion has been proofed to be a powerful technique supporting other optimisations and existing compilers as well. Pointer Clean-up Conversion transforms pointer-based programs that often achieve only suboptimal performance with modern architectures and compilers into programs that can be analysed successfully with existing Array Dependence tests and therefore can be better optimised. Apart from a preprocessing step for the implemented software-pipelining approach of this project, the Clean-up Conversion has more justifications. Applied by itself, the TriMedia compiler is enabled to produce better code than it can do with the original programs. Not only code generation for ILP processors benefits from this transformation, but also coarse-grain parallelisation techniques such as the automatic partitioning of data and computation are supported. Most of these techniques only work with explicit array representation and are not applicable in case of pointer presence. After the conversion originally pointer-based DSP programs are open for these techniques.
Chapter 7

Conclusions and Outlook

During this project a software-pipelining approach at source-to-source level has been implemented and evaluated. A series of tests based on the DSPstone benchmark suite showed only little influence on simple loops, but significant performance gains of up to 50\% for more complex loops with many pointer-based array references. Simpler loops can be analysed and optimised by the TriMedia compiler, whereas this compiler is not able to process more complex loops. In these cases the source-level software-pipelining transformation can help to rearrange the loop body in such a way that better instruction schedules can be generated. Although the number of complex loops in the DSPstone benchmark suite is small, more recent and future multimedia codes will show more often complex constructs due to increasing overall complexity of these applications. However, software-pipelining on source-to-source level is highly dependent on the available compiler technology. If the compiler used to generate the object code contains sophisticated analyses and powerful optimisations, the source level transformation can have either no effect at all or even detrimental effect. On the other hand, conventional compilers that perform only standard optimisations can be effectively supported by the software-pipelining transformation.

A contribution to compiler technology is the Pointer Clean-up Conversion that has been developed and implemented during this project. Apart from its original task of supporting the software-pipeliner by providing explicit array accesses that are easier to analyse than pointer-based accesses, this conversion is very useful in other contexts, too. The resulting explicit array accesses can be handled better by the TriMedia compiler, so that the performance after pointer conversion alone can be increased by up to 68\%. Additionally, automatic coarse-grain parallelisation techniques benefit from the explicit array representation. It is promising to further extend the capabilities of the pointer conversion and to study more real applications, other compilers and other optimisations benefitting from it.

The experiments conducted during the project have shown that the behaviour
of complex system such as a compiler and a DSP architecture are hard to predict. Already minor changes in the source program can have unexpected consequences when the generated object code is executed. Further research on application, compiler and architecture interactions are necessary. In particular, the influence of different optimisations on each other and the ordering of optimisation passes in order to achieve optimal code for a given architecture are interesting fields of research.

Whereas parallelisation for single processors DSPs, more precisely the exploitation of instruction level parallelism and data level parallelism, is an area in that some progress has been made, little effort has been put into parallelisation techniques for Multiprocessor DSPs. Multiprocessor DSPs combine features that can be found in other architectures only individually, but not in this unique combination. High-speed on-chip interconnection networks, processors with SIMD instructions and dual memory accesses to several small high-speed memory banks justify research on these architectures. Since current compilers for these architectures are rudimentary at best, more work on creating a cost model for arbitrary computations and developing methods for minimising the cost of given DSP and Multimedia programs are required for taking full advantage of the availability of single-chip multiprocessor DSPs.
Bibliography

    Compiler: Principles, Techniques, and Tools
    Addison-Wesley, Reading, MA. 1986.

[2] Aigner, G., David Heine, Constantine Sapuntzakis, Amer Diwan, Monica
    S. Lam
    The SUIF2 Compiler Infrastructure
    Computer Systems Laboratory, Stanford University, 1999.

    Compiling C for Vectorization, Parallelization, and Inline Expansion
    Proceedings of the SIGPLAN ’88 Conference of Programming Languages
    Design and Implementation, pp. 241-249, Atlanta, Georgia, June 22-24,

    Quad-SHARC DSP Multiprocessor Family AD14160/AD14160L

    ADSP-2183 Datasheet

    Modern Compiler Implementation in C

[7] de Araujo, Guido C.S.
    Code Generation Algorithms for Digital Signal Processors
    Dissertation, Princeton University, Department of Electrical Engineering,
    June 1997.

    The ARRL Handbook for Radio Amateurs
    Newington, CT, USA, 1999.

    Software Synthesis and Code Generation for Signal Processing Systems

Code Generation for a DSP Processor

Digital Signal Processing: A System Design Approach

LANCE - LS12 ANSI C Compilation Environment, User’s Guide

A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations

Breaking new grounds over 3000 MOPS: A broadband mobile multimedia modem DSP

[15] Forster, M., Andreas Pick and Marcus Rainer
The Graph Template Library (GTL) Manual

[16] Franke, Björn
Analysen und Methoden optimierender Compiler zur Steigerung der Effizienz von Speicherzugriffen in eingebetteten Systemen
Diplomarbeit, Fachbereich Informatik, Lehrstuhl XII, Universität Dortmund, August 1999.

[17] Frerking, A.E.
Digital Signal Processing in Communication Systems

[18] Fritts, J., Wayne Wolf and Bede Liu
Understanding multimedia application characteristics for designing programmable media processors
SPIE Photonics West, Media Processors '99, San Jose, CA, USA, 1999.

[19] PixelFusion Ltd.
FUZION 150 Product Overview
Complexities in DSP Software Compilation: Performance, Code Size, Power, Retargetability

[21] Horwitz, S., P. Pfeiffer and T. Reps
Dependence Analysis for Pointer Variables

[22] Hu, Ping
The Techniques for Software Pipelining Loops with Conditional Constructs

[23] Iwata, E. and Kunle Olukotun
Exploiting Coarse-Grain Parallelism in the MPEG-2 Algorithm

[24] Kernighan, Brian W. and Dennis M. Ritchie


[26] Lapsley, Phil
DSP processor fundamentals : architectures and features

[27] Lee, C., Miodrag Potkonjak and William H. Mangione-Smith
MediaBench: A Tool for Evaluating and Synthesizing Multimedia Communications Systems

[28] Leupers, Rainer
Novel Code Optimization Techniques for DSPs

[29] Leupers, Rainer
Exploiting Conditional Instructions in Code Generation for Embedded VLIW Processors
DATE ’99, München, Germany, 1999.
[30] Leupers, Rainer
Schneller Code statt schnelle Compiler
Elektronik, No. 22, WEKA Verlag, München, Germany, 1999.

[31] Leupers, Rainer
Optimierende Compiler für DSPs: Was ist verfügbar?
DSP Deutschland '97, München, 1997.

[32] Leupers, R. and Peter Marwedel
Flexible Compiler-Techniken für anwendungsspezifische DSPs
DSP Deutschland '96, München, 1996.

[33] Instruction Selection for Embedded DSPs with Complex Instruction
European Design Automation Conference (EURO-DAC), Geneva,
Switzerland, 1996.

[34] Leupers, Rainer
Compiler Optimizations for Media Processors

[35] Leupers, Rainer
Code Selection for Media Processors with SIMD Instructions
DATE '2000.

[36] Liao, H. and Andrew Wolfe
Available Parallelism in Video Applications
IEEE MICRO-30, 30th Annual Internation Symposium on Microarchitecture,

[37] Liao, S., S. Devedas, K. Keutzer, S. Tjiang and A. Wang
Code Optimization Techniques for Embedded DSP Microprocessors

[38] Lu, J.
Interprocedural Pointer Analysis for C
Ph.D. thesis, Department of Computer Science, Rice University, Houston,

Code Generation for Embedded Processors

[40] Marwedel, Peter
Compilers for Embedded Systems
http://ls12-www.informatik.uni-dortmund.de, Invited Embedded

[41] Marwedel, Peter
Code generation for core processors
34th Conference on Design Automation (DAC), Anaheim, CA, USA,
1997.
Effectiveness of Data Dependence Analysis  


[44] Muchnick, Steven S.  
Advanced Compiler Design and Implementation  

Digital Signal Processing  

[46] Panda, P.R., N.D. Dutt and A. Nicolau  
Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications  

[47] Philips Semiconductors  
R.E.A.L. DSP Core  

[48] Philips Semiconductors  
TM-1300 Media Processor Data Book  

[49] TM-1300 Media Processor  

[50] Philips Semiconductors  
Philips TriMedia SDE Reference I, Part 2: Program Development Tools  

[51] Rajan, S.P., Ashok Sudarsanam and Sharad Malik  
Development of an Optimizing Compiler for a Fujitsu Fixed-Point Digital Signal Processor  

[52] Rau, Bob Ramakrishna.  
Data Flow and Dependence Analysis for Instruction Level Parallelism  

[53] Rau, Bob Ramakrishna  
Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops  
ACM MICRO, San Jose, CA, California, USA, 1994.
[54] Saghir, M.A.R., Paul Chow, and Corinna G. Lee
   Exploiting Dual Data-Memory Banks in Digital Signal Processors
   Proceedings of the 7th International Conference on Architectural Support
   for Programming Languages and Operating Systems, pp. 234-243, October
   1996.

[55] Saghir, M.A.R., Paul Chow and Corinna G. Lee
   A Comparison of Traditional and VLIW Architectures for Compiled DSP
   Applications
   International Workshop on Compiler and Architecture Support for Em-

[56] Siroyan
   Product strategy

[57] Sjödin, J., B. Fröderberg and T. Lindgren
   Allocation of Global Data Objects in On-Chip RAM
   Whole Program Optimization for Embedded Systems Project, University

[58] Stoodley, Mark G. and Corinna G. Lee
   Software Pipelining Loops with Conditional Branches
   Proceedings of the 29th annual IEEE/ACM International Symposium on

[59] Stotzer, Eric
   Modulo Scheduling for the TMS320C6x VLIW DSP Architecture

[60] Su, B., Andrew Esguerra and Jian Wang
   A Study of Source Level Loop Optimization for DSP Code Generation
   Technical Report No. 118, College of Science and Health, William Paters-
   on University, 1998.

[61] Texas Instruments, Inc.
   TMS320C8x System-Level Synopsys

   TMS320C80 Multimedia Video Processor Data Sheet

[63] Tietze, Ulrich, and Christoph Schenk
   Halbleiter-Schaltungstechnik

[64] Wilson, R.P.
   Efficient Context-Sensitive Pointer Analysis for C Programs
   Ph.D. thesis, Stanford University, Computer Systems Laboratory, Decem-
   ber 1997.
[65] Wittenburg, Jens P., Willm Hinrichs, Johannes Kneip, Martin Ohnmacht, Mladen Bereković, Hanno Lieske, Helge Kloos and Peter Pirsch
Realization of a Programmable Parallel DSP for High Performance Image Processing Applications

DSPstone: A DSP-Oriented Benchmarking Methodology