MHAOTEU: Memory Hierarchy Analysis and Optimization Tools for The End-User

REPORT M3.D2

Guided Transformations

Jaume Abella, Cédric Bastoul, Jean-Luc Béchennec, Nathalie Drach, Christine Eisenbeis, Paul Feautrier, Bjoern Franke, Grigori Fursin, Antonio Gonzalez, Toru Kisku, Peter Knijnenburg, Josep Llosa, Michael O'Boyle, Julien Sébot, Xavier Vera

February 6, 2001

Contents

Abstract

This deliverable consists consists of 5 parts. The first two parts are concerned with optimising non-scientific or numerical programs. Part 1 examines DSP based applications and develops a mechanism to convert pointer based accesses in C into array references. Part 2 examines the particular characteristics of multi-media programs and examines how they may be optimised for the SIMD instructions now found in many general purpose processors.

Parts 3,4 and 5 examine advanced techniques for optimising applications. Part 3 examines a genetic algorithm to determine padding and tiling factors using the cache miss equations as an evaluation criteria. Part 4 extends this theme by using a variety of algorithms to search through a transformation space using actal execution time as the selection criteria. Finally Part 5 examines reodering methods as a means of improving data locality.

Part I
Program Analyses and Transformations Exploiting Parallelism in DSPs

1 Introduction

In 1995 the New York Times has estimated that the avarage 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 ubiquitious, 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-pipelinig 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.

2 Overview

Section 3 gives an overview of DSP Architectures, Applications and Compilers and introduces fundamental concepts. In section 12 a Pointer Clean-up Conversion is described which supports the Software-Pipelining algorithm, but additionally it can applied successfully in combination with other optimising transformations and their analyses. Software-Pipelining and its implementation as a part of the Octave compiler is explained in section 19. In section 46 empirical results achieved with Software-Pipelining and Pointer Clean-up Conversion are documented. Finally, the conclusions of this work are summarised in section 30.

3 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.

4 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 5) 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 [70] gives following statement to ILP processors:

> An **instruction-level parallel (ILP)** processor is a parallel processor in which the unit of computation, for which decisions such as scheduling and synchronization are made, is the individual operation, 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 independent 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- or memory-coupled “ordinary” DSPs. For more information on these subjects the reader is referred to Lapsley [26] and Koch [25], respectively.

4.1 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 section 8.

4.1.1 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 5.1), 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 8 and 4.2.1, respectively).

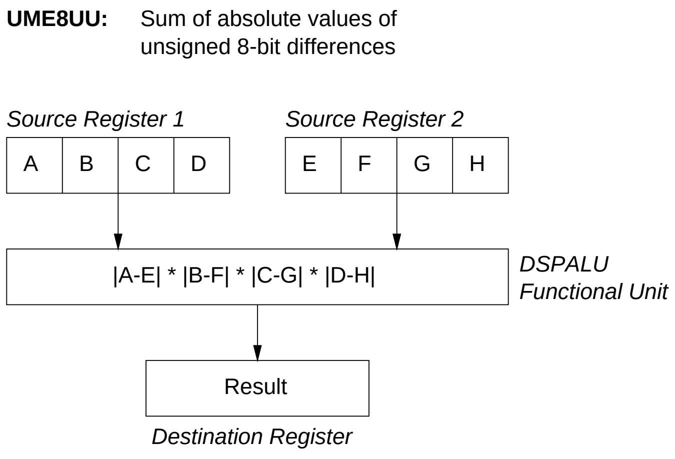
4.1.2 Arithmetic Logic Units

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 5.1). A MAC operation `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 1 shows an example of the `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.



The ume8ee operation, commonly used for motion estimation in video compression, implements 11 simple operations in one Trimedia special op.

Figure 1: Multimedia SIMD instructions [49]

4.1.3 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 super-scalar 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 ≈ 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.

4.1.4 Address Generation Units

Typical DSP programs (see section 5) 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 simultaneously. 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 5.1).

Figure 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/subtracter 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.

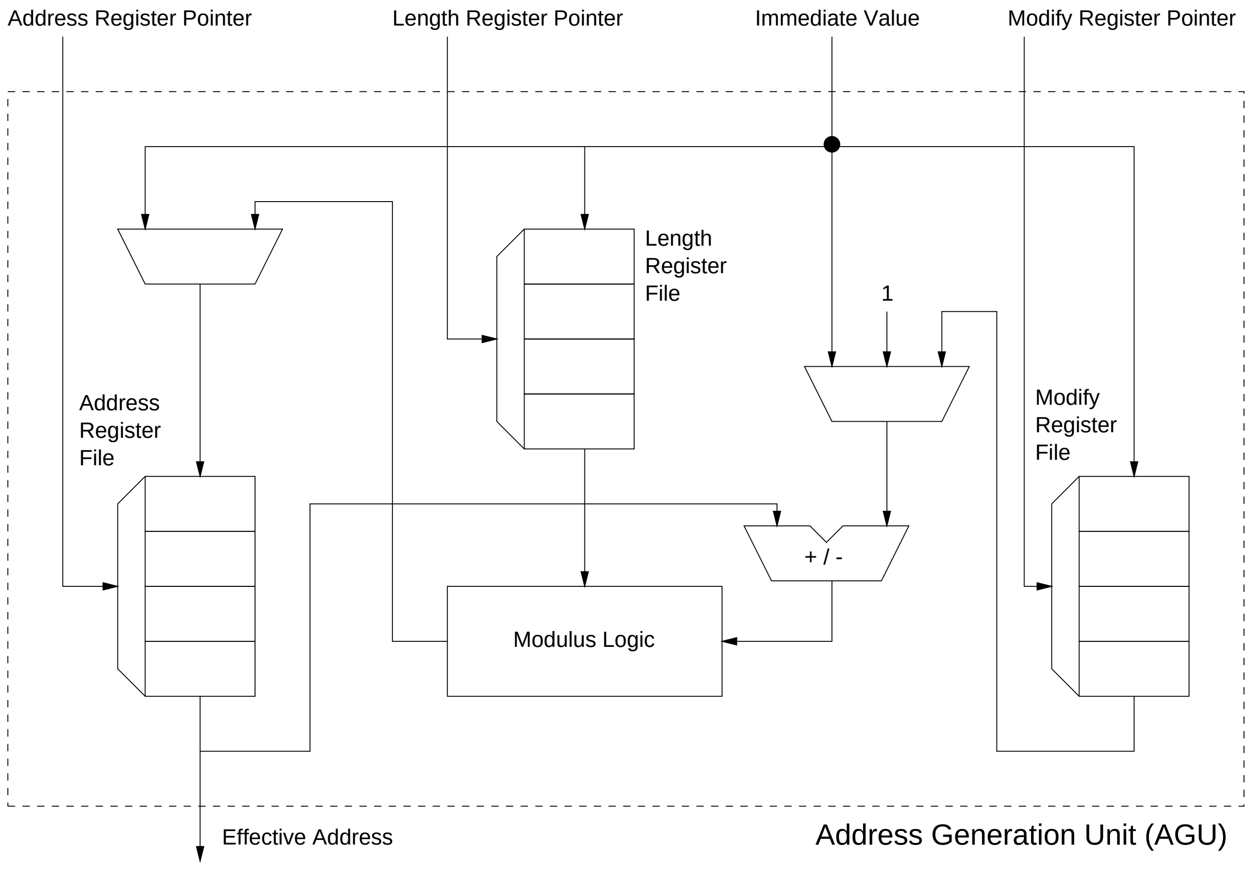


Figure 2: Address Generation Unit (AGU) with Address, Modify and Length Register Files

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 $104, 106, \ldots$. 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 register to 4. The addresses in ascending order are $0, 4, 8, \ldots$. 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.

4.1.5 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 ($2 \ldots 64kB$) 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 transfered from memory to registers simultaneously.

4.2 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.

4.2.1 Texas Instruments TMS320C80

The 5-processor Integrated MP-DSP TMS320C80 was presented by Texas Instruments 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 3.

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

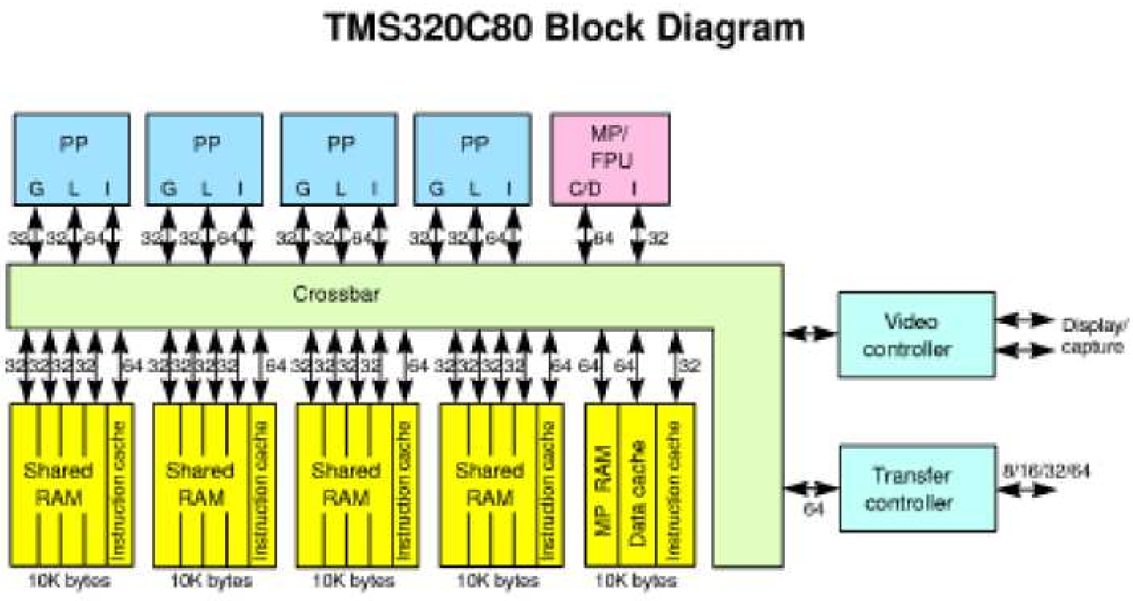


Figure 3: TI TMS320C80 DSP Architecture [62]

support certain SIMD instructions, i.e. a single operation can be applied to several different pairs of operands simultaneously (compare 4.1.2).

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.

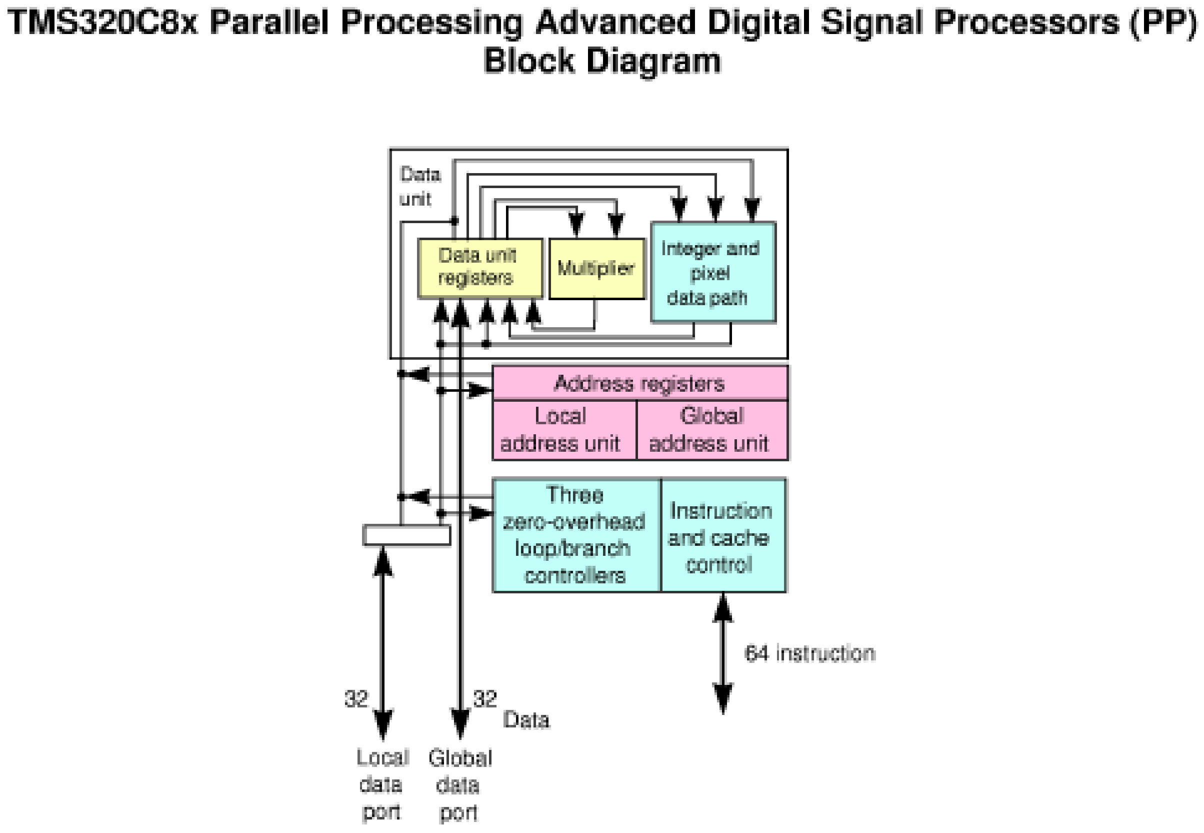

TMS320C8x Parallel Processing Advanced Digital Signal Processors (PP)
Block Diagram
Data unit
Data unit registers
Multiplier
Integer and pixel data path
Address registers
Local address unit
Global address unit
Three zero-overhead loop/branch controllers
Instruction and cache control
64 instruction
32
32
Data
Local data port
Global data port

Figure 4: Architecture of an individual DSP of the TI TMS320C80 [62]

Figure 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 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 4.1.4. 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.

5 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.

5.1 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 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.

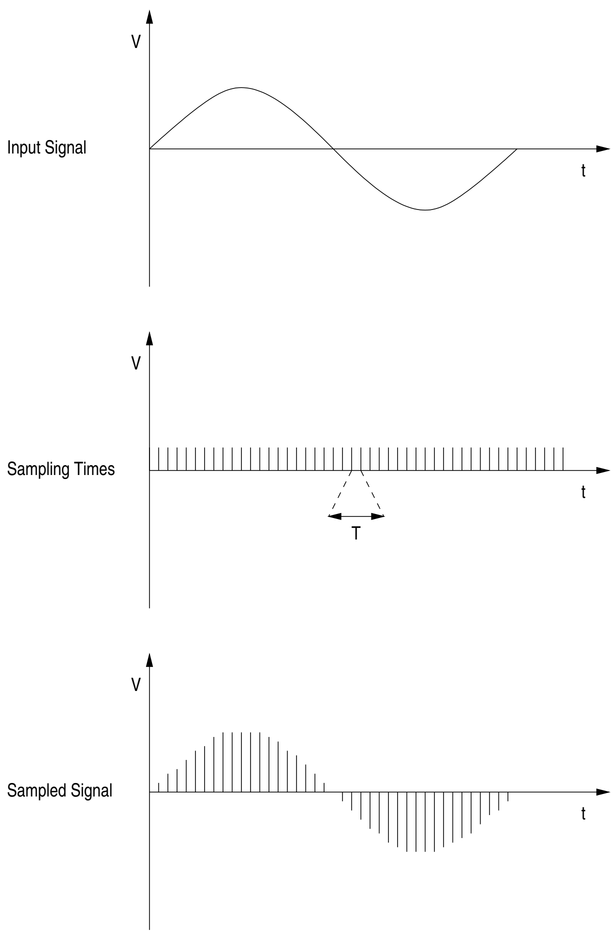


Figure 5: Sampling of a sine wave

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* δ 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 6.

More formally, the resulting equation of multiplying the input signal $U_i(t)$ by a series of Dirac impulses δ at sampling times t_μ and sampling interval T_s is

$$\tilde{U}_i(t) = \sum_{\mu=0}^{\infty} U_i(t_\mu) T_s \delta(t - t_\mu). \tag{1}$$

Figure 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*

$$f_s \geq 2 f_{max}. \tag{2}$$

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

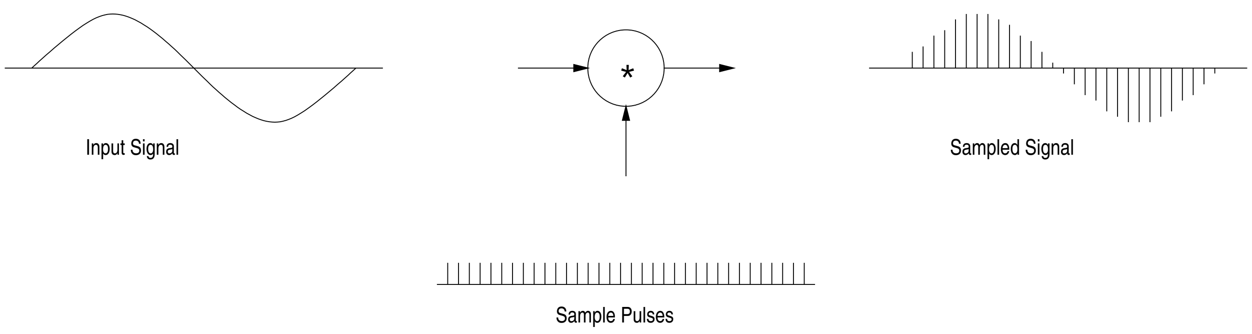


Figure 6: Sampling as a multiplication

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.

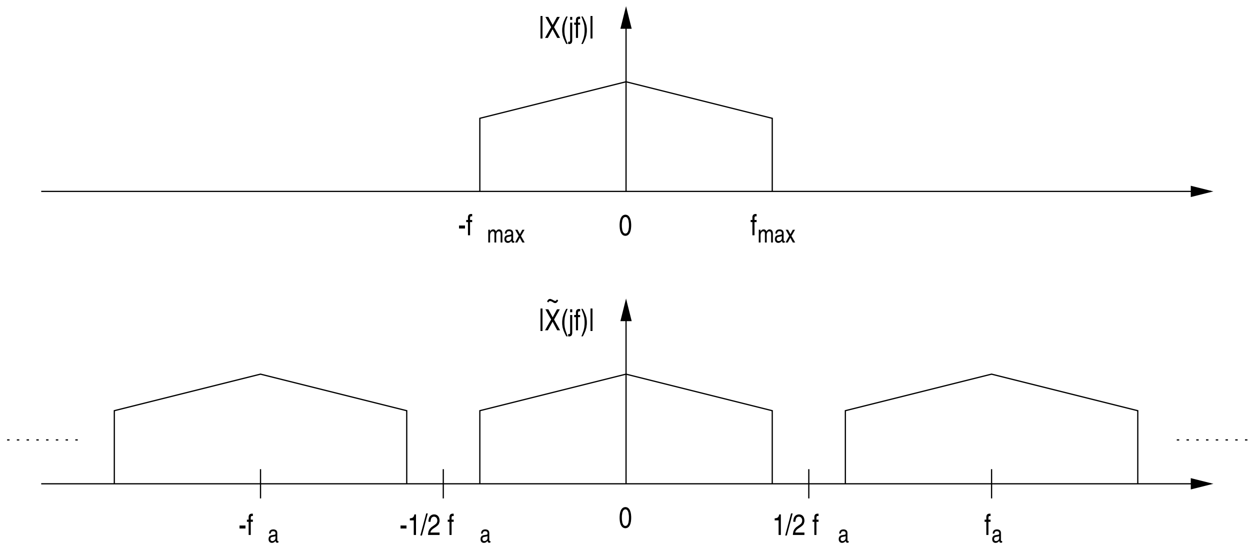


Figure 7: Spectrum of a signal before and after sampling

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 1 results in

$$\tilde{X}(\mathrm{j}f) = T_s \sum_{\mu=0}^{\infty} U_i(\mu T_s) e^{-2\pi \mathrm{j} \mu f / f_s}. \tag{3}$$

By using Euler's rule, equation 3 can be transformed into

$$\tilde{X}(\mathrm{j}f) = T_s \sum_{\mu=0}^{\infty} U_i(\mu T_s)[cos(2\pi\mu \frac{f}{f_s}) - \mathrm{j}sin(2\pi\mu \frac{f}{f_s})] \tag{4}$$

Any particular value of $\tilde{X}(\mathrm{j}f)$ 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

$$\tilde{X}(\mathrm{j}f) = a + \mathrm{j}b$$

$$|\tilde{X}(\mathrm{j}f)| = \sqrt{a^2 + b^2}$$

$$\theta(\tilde{X}(\mathrm{j}f)) = tan^{-1}\frac{b}{a} \tag{5}$$

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

$$U_o(n) = \frac{1}{N}\sum_{k=0}^{N-1} \tilde{X}(k)e^{-2\pi \mathrm{j} nk/N} \tag{6}$$

or

$$U_o(n) = \frac{1}{N}\sum_{k=0}^{N-1} \tilde{X}(k)[cos(2\pi nk/N) + \mathrm{j}sin(2\pi nk/N)] \tag{7}$$

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

$$H(z) = \frac{Y(z)}{X(z)}. \tag{8}$$

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

$$y(n) = \sum_{k=-\infty}^{\infty} x(n-k)h(k) \tag{9}$$

or

$$y(n) = \sum_{k=-\infty}^{\infty} x(k)h(n-k). \tag{10}$$

Equation 10 is known as *Convolution Sum* and the process of taking this sum is *Convolution*.

5.1.1 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 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 α_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

$$y_N = \alpha_0 x_N + \alpha_1 x_{N-1} + \cdots + \alpha_{N-1} x_0, \quad y_N = \sum_{k=0}^{N} \alpha_k x_{N-k} \tag{11}$$

and

$$\tilde{H}(z) = \frac{Y(z)}{X(z)} = \sum_{k=0}^{N} \alpha_k z^{-k} \text{ , respectively.} \tag{12}$$

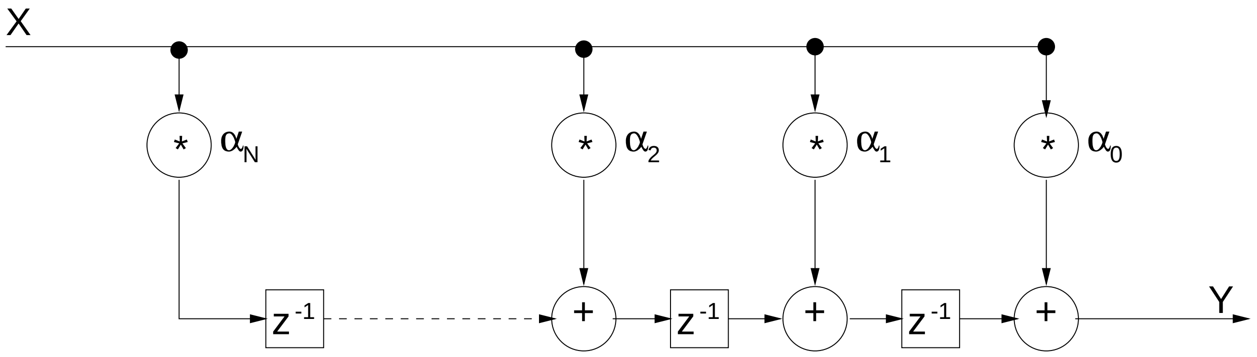


Figure 8: Structure of an FIR filter

More suitable for implementation in a serial IC or as a piece of software is the filter shown in figure 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

$$y(t_N) = \sum_{k=0}^{N} \alpha_k x(t_{N-k}) \tag{13}$$

FIR filters directly implement convolution, as can be seen by comparing equations 13 and 10.

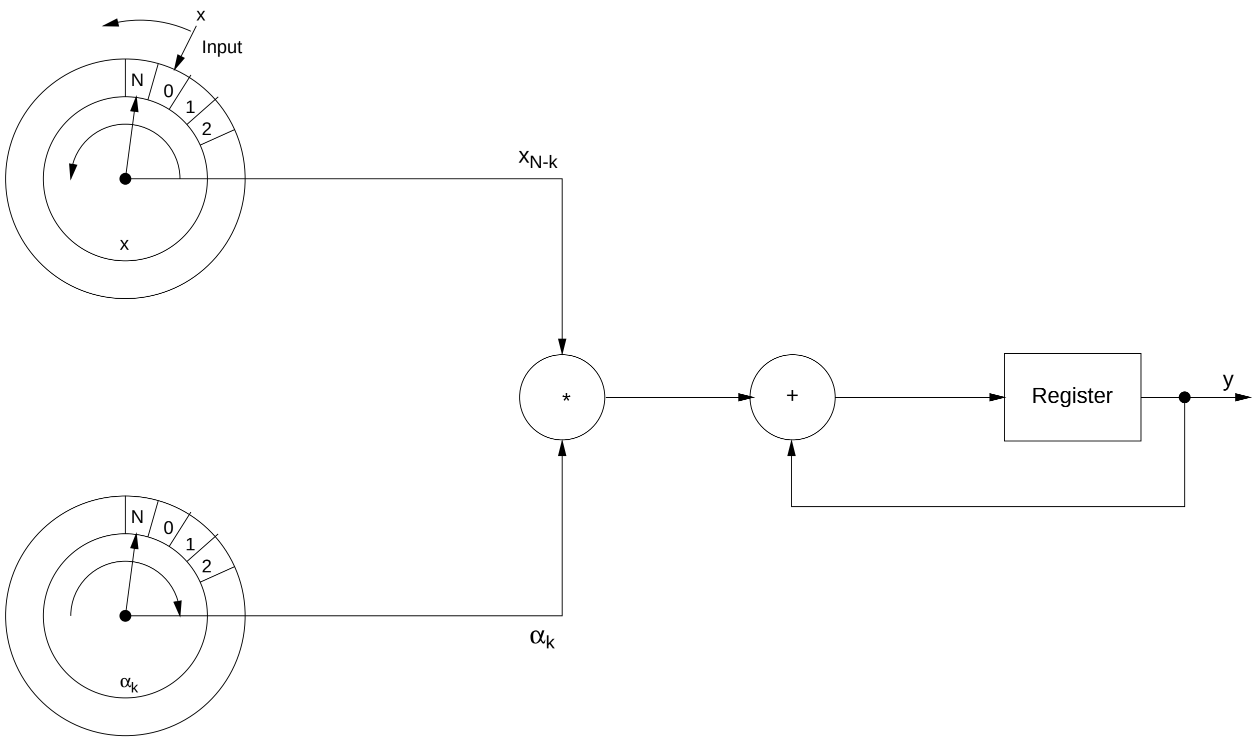


Figure 9: Serial implementation of an FIR filter

After all this theory, it becomes clear how characteristic features of DSP introduced in section 4.1 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.

5.2 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 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 10 shows the levels *video sequence*, *group of pictures (GOP)*, *picture (frame)*, *slice*, *macroblock* and *block*. The MPEG standard differentiates between three different types of pictures. The first picture type of pictures are *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 *Predicted (P)* and *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.

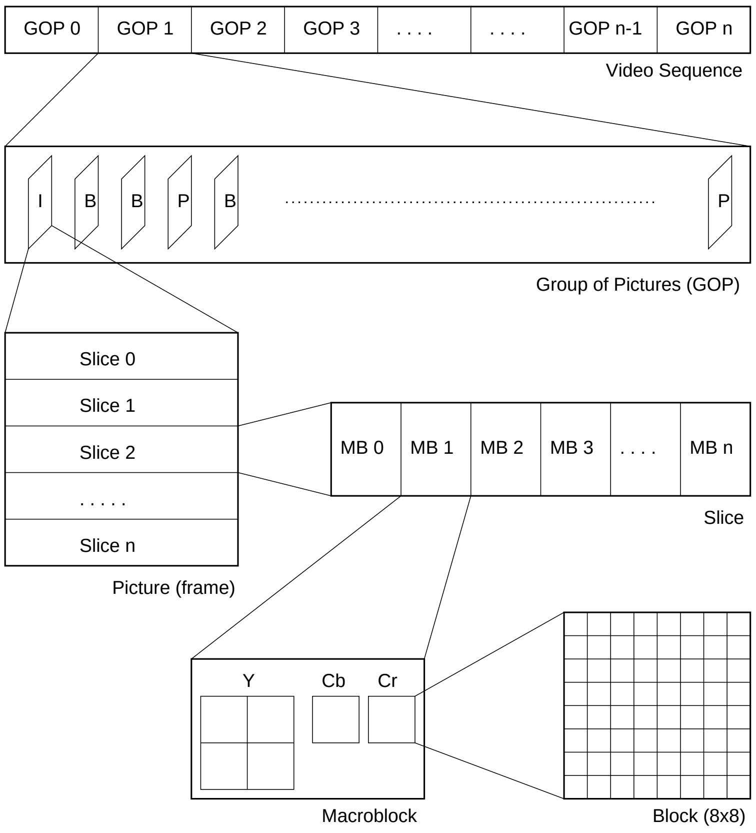


Figure 10: MPEG video stream data structure [23]

In 11 a block diagram of an MPEG-2 encoder is shown. This encoder consists of several stages which are *Motion Estimation (ME)*, *Forward Discrete Cosine Transformation (FDCT)*, *Quantisation (Q)*, *Variable Length Coding (VLC)*, *Inverse Quantisation (IQ)*, *Inverse Discrete Cosine Transformation (IDCT)*, *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 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*

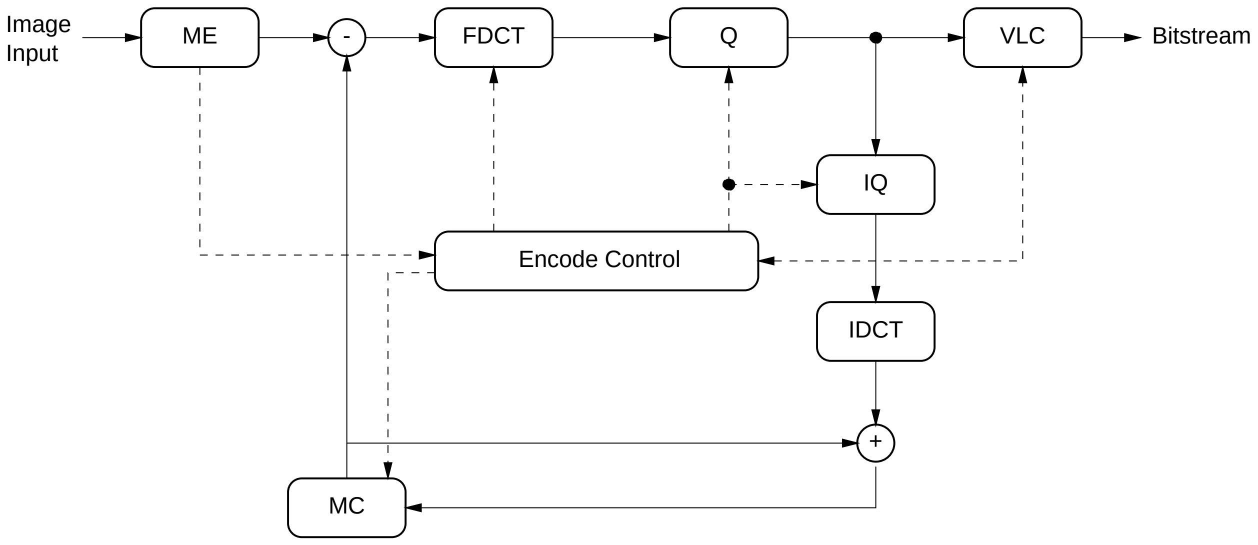


Figure 11: Block diagram of an MPEG-2 encoder [23]

and a stage for co-ordinating the *Decode Control*.

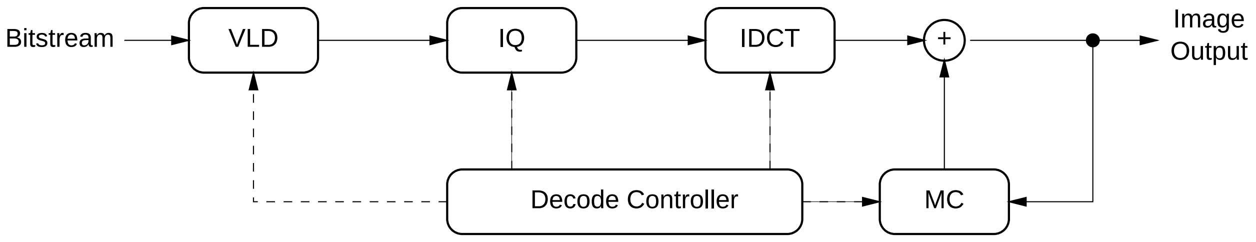


Figure 12: Block diagram of an MPEG-2 decoder [23]

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 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 5.3) 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.

5.3 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 [50, 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 [1] that can often be found in "real" DSP applications. A description of the programs of the DSPstone benchmark suite is shown in table 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 gives some brief descriptions of the individual applications of the MediaBench suite.

According to Lee [50] 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 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.

[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.

ADPCM	Adaptive Differential Pulse Code Modulation
complex_multiply	Multiplication of two complex numbers
complex_update	Update of a complex number; similar to Multiply Accumulate with complex numbers, but with arbitrary destination
convolution	Computation of a convolution sum (see 5.1)
dot_product	Dot-product of a $(1, 2)$ and a $(2, 1)$ vector
fir	FIR filter with parameterisable number of taps (see 5.1)
fir2dim	FIR filter for image filtering, i.e. the filter is applied to matrix data rather than a sequence of values
biquad_N_sections	IIR biquad filter with parameterisable number of sections
biquad_one_section	Computations for one section of an IIR biquad filter
lms	Implementation of an adaptive DLMS filter
matrix	Two programs (*matrix1, matrix2*) for the multiplication of two matrices of arbitrary dimensions
matrix1x3	Multiplication of a $(3, 3)$ matric by a $(3, 1)$ vector
n_complex_updates	Updates of n complex numbers in a way similar to Multiply-Accumulate
n_real_updates	Updates of n complex numbers with values coming from three different arrays and multiplication and addition as operators
real_update	A single real update

Table 1: Description of the DSPstone benchmark suite

ADPCM	A simple adaptive differential pulse code modulation coder (*rawcaudio*) and decoder (*rawdaudio*)
EPIC	An image compression coder (*epic*) and decoder *unepic*) based on wavelets and including run-length/Huffman entropy coding
G.721	Voice compression coder (*encode*) and decoder (*decode*) based on the G.711, G.721 and G.723 standards
Ghostscript	An interpreter (*gs*) for the PostScript language; performs file I/O but no graphical display
GSM	Full-rate speech transcoding coder (*gsmencode*) and decoder (*gsmdecode*) based on the European GSM 06.10 provisional standard
H-263 (*)	A very low bitrate video coder (*h263enc*) and decoder (*h263dec*) based on the H.263 standard; provided by Telenor R&D
JPEG	A lossy image compression coder (*cjpeg*) and decoder (*djpeg*) for colour and grayscale images, based on the JPEG standard; performs file I/O but no graphical display
Mesa	A 3-D graphics library clone of OpenGL; includes three demo programs (*mipmap, osdemo, texgen*); performs file I/O but no graphical display
MPEG-2	A motion video compression coder (*mpeg2enc*) and decoder (*mpeg2dec*) for high-quality video transmission, based on the MPEG-2 standard; perform file I/O but no graphical display
MPEG-4 (*)	A motion video compression coder (*mpeg4enc*) and decoder (*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
PEGWIT	A public key encryption and authentication coder (*pegwitenc*) and decoder (*pegwitdec*)
PGP	A public key encryption coder (*pgpenc*) and decoder (*pgpdec*) including support for signatures
RASTA	A speech recognition application (*rasta*) that supports the PLP, RASTA and Jah-RASTA feature extraction techniques

Table 2: Description of the MediaBench benchmark suite [18]

6 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 consumptions 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.

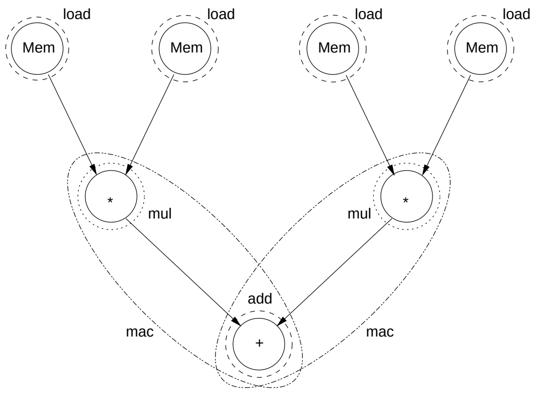
The following sections give brief introductions to some specific problems of compilation for DSPs. Further, some introduction to data flow analysis is presented because data flow information is needed at various compilation stages. For a more comprehensive overview of the highly interesting topic of 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].

6.1 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 is 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 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 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, 17, 33].



Instr.	Cost	Pattern
load	1	t1 := Mem
mov2	1	r2 := r1
mov3	1	r3 := r1
add	2	r1 := r1 + r2
mul	2	r1 := r1 * r2
mac	3	r1 := (r1 * r2) + r3

Figure 13: Expression $(a \times b) + (c \times d)$ and instruction patterns [40]

The instructions sets of modern DSPs support Multimedia SIMD instructions (see 4.1 and 8). 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 simulaneously. 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.

6.2 Address Code Generation

Typically, DSPs only have few addressing modes. Hence, there a 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 section 4.1.4). 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 section 4.1), 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].

6.3 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 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 section 11. 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 unconvenient. 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 `frac` data type for fix-point values and corresponding operators.

6.4 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, 59]. 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* $f : D \to D$. 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 6.1 *A lattice L consists of a set of elements, the* Carrier Set*, and two operators* $\sqcap$ (meet) *and* $\sqcup$ (join) *such that:*

1. *Closedness:*
   $\forall x, y \in L : \exists z, w \in L : \quad x \sqcap y = z \quad \wedge \quad x \sqcup y = w$
2. *Commutativity:*
   $\forall x, y \in L : \quad x \sqcap y = y \sqcap x \quad \wedge \quad x \sqcup y = y \sqcup x$
3. *Associativity:*
   $\forall x, y \in L : \quad (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z) \quad \wedge \quad (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
4. *Supremum, Infimum*

   - $\exists \bot \in L : \forall x \in L : \quad x \sqcap \bot = \bot$
   - $\exists \top \in L : \forall x \in L : \quad x \sqcup \top = \top$

   The elements $\bot$ and $\top$ are called bottom *and* top*, respectively.*

Definition 6.2 *A lattice L is a* Distributive Lattice*, if in addition*

$$\forall x, y, z \in L : (x \sqcap y) \sqcup z = (x \sqcup z) \sqcap (y \sqcup z) \quad \wedge \quad (x \sqcup y) \sqcap z = (x \sqcap z) \sqcup (y \sqcap z)$$

hold.

Definition 6.3 *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. $\sqsubset$, $\sqsupset$ and $\sqsupseteq$ are defined analogously.*

Definition 6.4 *The* height *of a lattice L is defined as*

$$height(L) = max\{n | \exists x_1, \ldots, x_n : \bot = x_1 \sqsubset x_2 \sqsubset \ldots \sqsubset x_n = \top\}.$$

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 $f : L \rightarrow L$ 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 $tf : N \rightarrow (L \rightarrow L)$. 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 6.5 *A function $f : L \to L$ is* monotone*, iff*

$$\forall x, y \in L : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

holds.

Definition 6.6 *The* Effective Height *of a lattice L with respect to some function $f : L \to L$ is*

$$height^f_{eff}(L) = max\{n | \exists x_1, x_2 = f(x_1), x_3 = f(x_2), \ldots , x_n = f(x_{n-1}) \in L :$$

$$x_1 \sqsubset x_2 \sqsubset x_3 \sqsubset \ldots \sqsubset x_n \sqsubseteq \top\}.$$

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 preceeding 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 $s \rightsquigarrow n$ for a node $n \in N$. Then, the *Merge-over-all-path (MOP)*-solution for the *May*-problem is

  $$MOP_{May}(n) = \bigsqcup_{p \in \pi(n)} f_p(\iota) \qquad (14)$$

  for some initial value $\iota \in L$.

- Let $\pi(n)$ the set of all paths $s \rightsquigarrow n$ for a node $n \in N$. Then, the *Merge-over-all-path (MOP)*-solution for the *Must*-problem is

  $$MOP_{Must}(n) = 180 \bigsqcup_{p \in \pi(n)} f_p(\iota) \qquad (15)$$

  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 6.7 *A* Fixpoint *of a function $f : L \rightarrow L$ 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

$$MFP_{May}(n) = \begin{cases} \iota & \text{, if } n = s \\ \bigsqcup\limits_{m \in pred(n)} f_n(n)(MFP(m)) & \text{, otherwise} \end{cases} \tag{16}$$

  for some initial value $\iota \in L$.

- A *Maximal Fixpoint (MFP)* for a *Must*-problem is

$$MFP_{Must}(n) = \begin{cases} \iota & \text{, if } n = s \\ 180 \bigsqcup\limits_{m \in pred(n)} f_n(n)(MFP(m)) & \text{, otherwise} \end{cases} \tag{17}$$

  for some initial value $\iota \in L$.

In general, i.e. without further assumptions, it is $\forall n \in N : MOP_{Must}(n) \supseteq MFP_{Must}(n)$ and $\forall n \in N : MOP_{May}(n) \subseteq MFP_{May}(n)$, respectively. However, if all transfer functions are distributive, the MOP in this special case can be computed and $\forall n \in N : MFP(n) = MOP(n)$ 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 $CFG = (N, E; s, e)$ is labelled with to values $IN[n], OUT[n] \in L$. These values denote data flow reaching and leaving the node, respectively. The must-data flow equations for a node n look like this:

$$IN[n] = \begin{cases} \iota & \text{, if } n = s \\ 180 \bigsqcup\limits_{m \in pred(n)} OUT[m] & \text{, otherwise} \end{cases} \tag{18}$$

$$OUT[n] = f_n(IN[n]) \tag{19}$$

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

$$OUT[n] = G[n] \cup (IN[n] - K[n]) \tag{20}$$

A *Worklist Algorithm* traverses the 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 OUT values are computed and propagated to its succeeding nodes. If a nodes' 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 IN and OUT sets must be interchanged. The data flow moves from OUT to IN.

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

6.5 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 δ 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 $S_1\,\delta_{flow}\,S_2$. An *Anti Dependence* $S_1\,\delta_{anti}\,S_2$ 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 sematic 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 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

$$\exists\, \alpha \leq \beta \text{ , such that } f_k(\alpha) = g_k(\beta) \quad \forall k, 1 \leq k \leq m. \tag{21}$$

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.

```
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[$f_1(i_1, \dots, i_n), \dots, f_m(i_1, \dots, i_n)$] = ... ;      $S_1$
             ... = a[$g_1(i_1, \dots, i_n), \dots, g_m(i_1, \dots, i_n)$];       $S_2$
         }
   }
}
```

Figure 14: Array Data Dependences in a Perfect Loop Nest

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 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.

6.5.1 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 $I = (i_1, \dots, i_n)$ and $I' = (i'_1, \dots, i'_n)$ in the iteration space let

$$h(I, I') = \alpha I - \beta I' \tag{22}$$

and

$$h_i^+(I_k, I'_k) = max_{R_k} h(I_k, I'_k) \tag{23}$$

$$h_i^-(I_k, I'_k) = min_{R_k} h(I_k, I'_k). \tag{24}$$

$I_k \ D \ I'_k$ is the relation imposed by the direction vector element (either "<", ">" or "=").

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

$$\exists \text{ real solution to } \alpha I - \beta I' \quad \Leftrightarrow \quad \sum_{i=1}^{H} H_i - D_i \leq \beta_0 - \alpha_0 \leq \sum_{i=0}^{H} H_i + D_i. \tag{25}$$

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.

6.5.2 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 accesses. Based on the *Reaching Definitions* problem (see [1, 59]) an *Iteration Dependence* can be defined as follows:

Definition 6.8 *A definition d reaches a point p with* Iteration Distance δ*, if all of the last δ instances of d reach p. An iteration distance is* maximal*, if δ 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 δ. 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 $\{\perp, 0, 1, \ldots, \top = UB - 1\}$ with the upper loop bound UB. The lattice is a chain lattice, i.e. it forms a strictly monotone ascending chain of elements $\perp \leq 0 \leq 1 \leq \ldots\ldots \top$ 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:

$$L = ([\perp, 0, \ldots, \top = UB - 1], \perp, \top, \sqsubseteq, \wedge, \vee) \tag{26}$$

such that

$$\sqsubseteq : \forall x_i \in [\perp, 0, \ldots, \top = UB - 1] : x_i < x_{i+1} \tag{27}$$

and

$$\wedge(x,y) = \begin{cases} \perp & \text{, if } x = \perp \text{ or } y = \perp \\ x & \text{, if } y = \top \\ y & \text{, if } x = \top \\ min(x,y) & \text{, otherwise} \end{cases} \tag{28}$$

$$\vee(x,y) = \begin{cases} \top & \text{, if } x = \top \text{ or } y = \top \\ x & \text{, if } y = \perp \\ y & \text{, if } x = \perp \\ max(x,y) & \text{, otherwise} \end{cases} \tag{29}$$

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:

$$x++ = \begin{cases} \top & \text{, if } x = \top \\ \bot & \text{, if } x = \bot \\ x+1 & \text{, otherwise} \end{cases} \tag{30}$$

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 $IN[n] = (x_1, \ldots, x_m)$ and $OUT[n] = (y_1, \ldots, y_m)$ and have the same function as their scalar counterparts. After the data flow analysis is finished, the data flow solution can be read from IN and OUT. The transfer functions together with the IN and 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:

$$IN[n,d]^0 = \begin{cases} \bot & \text{, if } n = 1 \text{ Loop Entry} \\ \bigwedge\limits_{m \in pred(n)} OUT[m,d]^0 & \text{, otherwise} \end{cases} \tag{31}$$

$$OUT[n,d]^0 = \begin{cases} \top & \text{, if } d \in G[n] \\ \bigwedge\limits_{m \in pred(n)} IN[n,d]^0 & \text{, otherwise} \end{cases} \tag{32}$$

The succeeding iteration steps that are also done in Reverse Postorder apply following equations[2]:

$$IN[n,d]^{i+1} = \begin{cases} \bigwedge\limits_{m \in pred(n)} OUT[m,n]^i & \text{, if } n = 1 \\ \bigwedge\limits_{m \in pred(n)} OUT[m,d]^{i+1} & \text{, otherwise} \end{cases} \tag{33}$$

$$OUT[n,d]^i = f_n^d(IN[n,d]^i) \tag{34}$$

Because all analysed loops are structured loops which are *Single-entry/Single-exit* loops, equation 33 can be simplified:

$$IN[n,d]^{i+1} = \begin{cases} OUT[Exit,n]^i & \text{, if } n = 1 \\ \bigwedge\limits_{m \in pred(n)} OUT[m,d]^{i+1} & \text{, else} \end{cases} \tag{35}$$

[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].

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.

7 Infrastructure

This section 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 section.

8 Philips TriMedia TM-1

The TriMedia TM-1 is a single processor DSP tailored for Multimedia applications. Figure 15 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 I^2C 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 section 5.2. The CPU itself is speeded up by an instruction and a data cache. Additionally, the caches can help to reduce the bus traffic. Betweeen the internal bus and the external SDRAM a main memory interface takes over control.

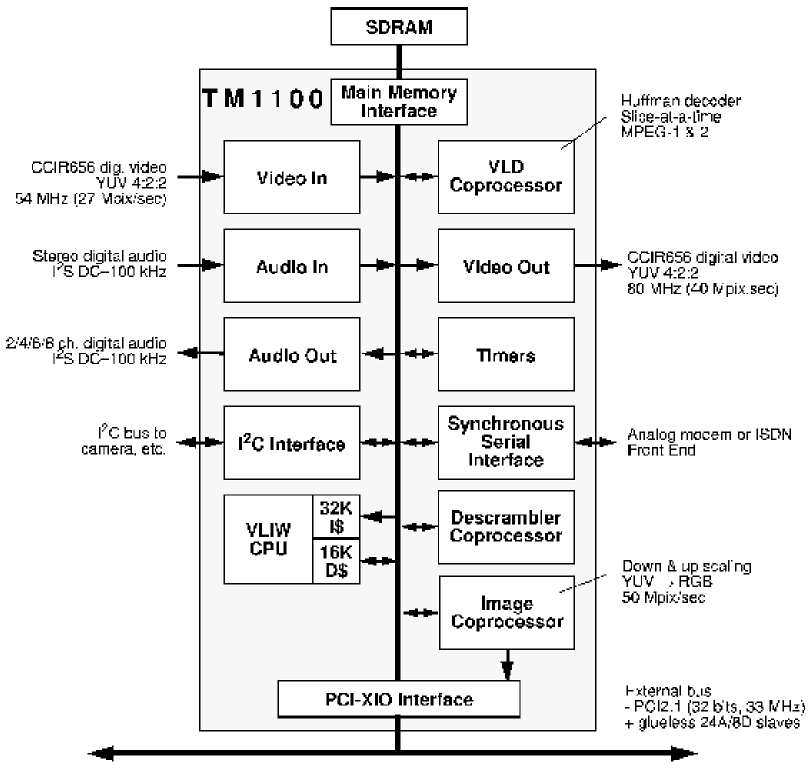


Figure 15: TriMedia TM-1100 Architecture [49]

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 16 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 section 4.1.

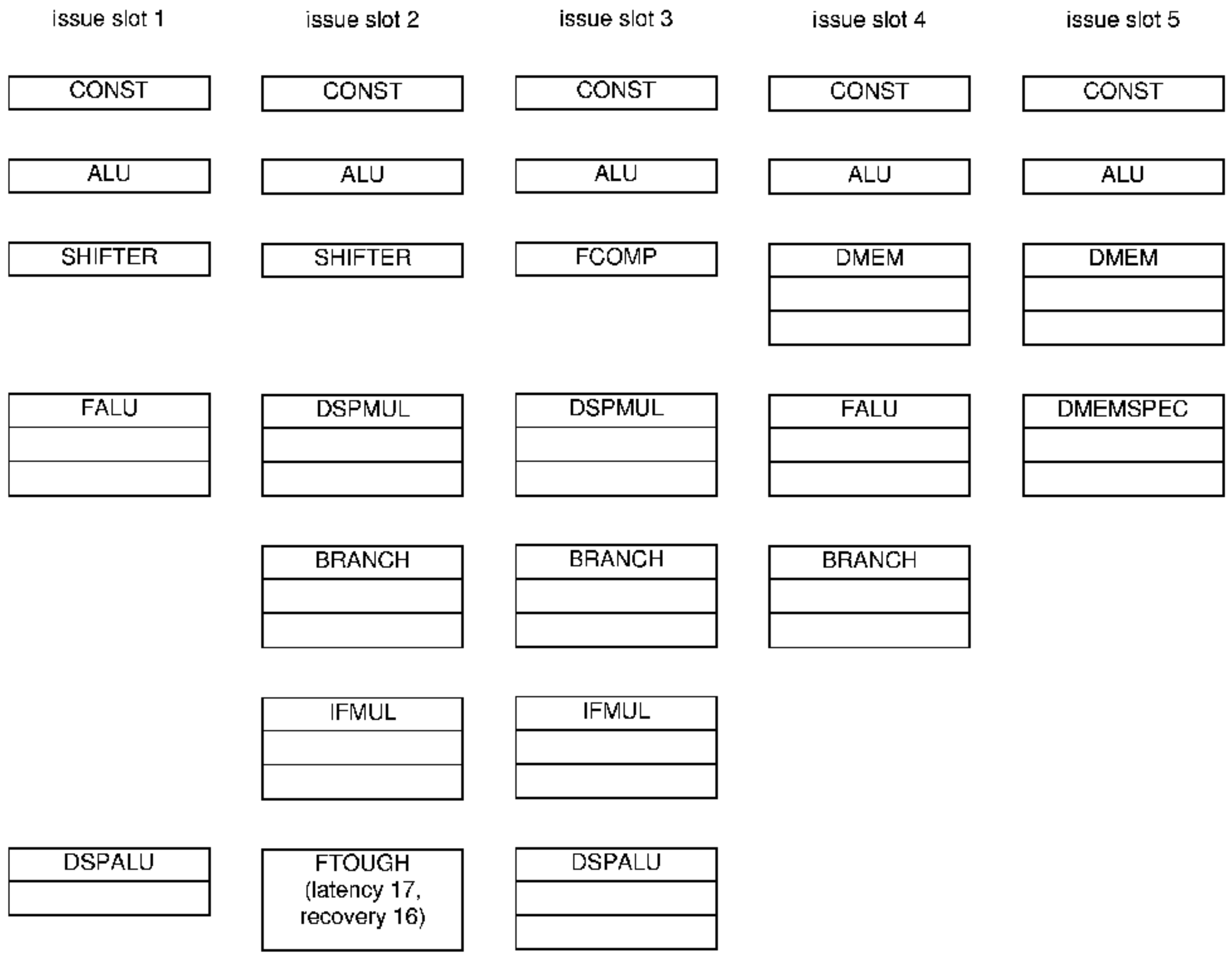


Figure 16: TM-1100 Issue Slots, Functional Units and Latencies [49]

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 comparision, ... 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].

9 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.

9.1 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 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.

10 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 assmbler.

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 sections 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 addtional *inter-decision-tree register allocation* is enabled. Level 3 (O3) adds some further global optimisations.

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 [59, 1, 6], and on specific optimisation of the TriMedia SDE see [50].

11 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 stil 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 17 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 17: *Convolution* program of the DSPstone benchmark suite

12 Pointer Clean-up Conversion

This section 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.

13 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 `biquad_N_sections` in figure 18. 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 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.

Figure 19 shows a loop with explicit array indexes that is semantically equivalent to the previous loop in figure 18. 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 `wi` are involved in several loop independent anti- and input-dependences and that there are no loop carried dependences with respect to `wi` 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

```
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 18: Original pointer-based loop

```
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 19: Loop after conversion of pointer-based accesses into explicit array accesses

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.

14 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 [59] 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].

15 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.

16 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:

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 20 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 20: 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 21 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`.

```
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 21: Legal pointer initialisations

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 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 22). 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 22: 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.

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.

17 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 1 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 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

Algorithm 1 Pointer clean-up conversion for CFG G

```
map ← ∅
L ← preorderList(G);
while L not empty do
  stmt ← head(L);
  removeHead(L);
  if stmt is pointer declaration and initialisation then
    map ← map ∪ (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,*,*) ∈ map then
      map ← map - (pointer,array,*,*)
    end if
    map ← map ∪ (pointer,array,offset,0)
  else if stmt contains pointer reference then
    Look up (pointer,array,offset,*) ∈ map
    if stmt contains pointer-based array access then
      replace pointer-based access by array[initial index+offset]
    else if stmt contains pointer arithmetic then
      map ← map - (pointer,array,offset,*)
      calculate new offset
      map ← map ∪ (pointer,array,new offset,0)
    end if
  end if
end while
```

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 18 as computed by the algorithm and stored in *map* are shown in figures 23 and 24. 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 Δ and local offset δ. 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.

18 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.

18.1 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

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) $\in$ map
    map ← map - (pointer,array,offset,increment)
    calculate new increment
    map ← map $\cup$ (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) $\in$ map
    Look up (pointer,local offset) $\in$ offsetMap
    if (pointer,local offset) $\notin$ offsetMap then
      offsetMap ← offsetMap $\cup$ (pointer,0)
    end if
    if stmt contains pointer-based array access then
      index function ← increment $\times$ 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 $\cup$ (pointer,new local offset)
    end if
  end if
end while
for all (pointer,*,*,*) $\in$ map do
  map ← map - (pointer,array,offset,increment)
  new offset ← increment $\times$ upper loop bound + offset + local offset
  map ← map $\cup$ (pointer,array,new offset,0)
end for
```

`int w, f;`	Contents of data structure *map:*
`int *ptr_coeff, *ptr_wi1, *ptr_wi2;`	$\{(ptr, array, offset, increment), \ldots\}$
`int wi[2 * NumberOfSections];`	
`int coefficients[5*NumberOfSections];`	
`int x,y;`	
`ptr_coeff = &coefficients[0];`	(ptr_coeff,coefficients,0,0)
`ptr_wi1 = &wi[0];`	(ptr_coeff,coefficients,0,0),(ptr_wi1,wi,0,0)
`ptr_wi2 = &wi[1];`	(ptr_coeff,coefficients,0,0),(ptr_wi1,wi,0,0),(ptr_wi2,wi,1,0)
`x = pin_down(x, coefficients, wi);`	
`y = x;`	
`for (f=0;f<NumberOfSections;f++)`	
`{`	
`  w  = y -  *ptr_coeff++ * *ptr_wi1;`	(ptr_coeff,coefficients,0,1),(ptr_wi1,wi,0,0),(ptr_wi2,wi,1,0)
`  w -= *ptr_coeff++ * *ptr_wi2;`	(ptr_coeff,coefficients,0,2),(ptr_wi1,wi,0,0),(ptr_wi2,wi,1,0)
`  y  = *ptr_coeff++ *  w;`	(ptr_coeff,coefficients,0,3),(ptr_wi1,wi,0,0),(ptr_wi2,wi,1,0)
`  y += *ptr_coeff++ * *ptr_wi1;`	(ptr_coeff,coefficients,0,4),(ptr_wi1,wi,0,0),(ptr_wi2,wi,1,0)
`  y += *ptr_coeff++ * *ptr_wi2;`	(ptr_coeff,coefficients,0,5),(ptr_wi1,wi,0,0),(ptr_wi2,wi,1,0)
`  *ptr_wi2++ = *ptr_wi1;`	(ptr_coeff,coefficients,0,5),(ptr_wi1,wi,0,0),(ptr_wi2,wi,1,1)
`  *ptr_wi1++ = w;`	(ptr_coeff,coefficients,0,5),(ptr_wi1,wi,0,1),(ptr_wi2,wi,1,1)
`  ptr_wi2++;`	(ptr_coeff,coefficients,0,5),(ptr_wi1,wi,0,1),(ptr_wi2,wi,1,2)
`  ptr_wi1++;`	(ptr_coeff,coefficients,0,5),(ptr_wi1,wi,0,2),(ptr_wi2,wi,1,2)
`}`	

Figure 23: First pass of clean-up conversion and values of the *map* data structure

```
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:*
$\{(ptr, localoffset), \ldots\}$

(ptr_coeff,0),(ptr_wi1,0),(ptr_wi2,0)
(ptr_coeff,1),(ptr_wi1,0),(ptr_wi2,0)
(ptr_coeff,2),(ptr_wi1,0),(ptr_wi2,0)
(ptr_coeff,3),(ptr_wi1,0),(ptr_wi2,0)
(ptr_coeff,4),(ptr_wi1,0),(ptr_wi2,0)
(ptr_coeff,5),(ptr_wi1,0),(ptr_wi2,0)
(ptr_coeff,5),(ptr_wi1,0),(ptr_wi2,1)
(ptr_coeff,5),(ptr_wi1,1),(ptr_wi2,1)
(ptr_coeff,5),(ptr_wi1,1),(ptr_wi2,2)

$\Longrightarrow$ (ptr_coeff,5),(ptr_wi1,2),(ptr_wi2,2)

Array index functions

`coeff.`	`wi` (1)	`wi` (2)
$5 \times f + 0$	$2 \times f + 0$	
$5 \times f + 1$		$2 \times f + 1$
$5 \times f + 2$		
$5 \times f + 3$	$2 \times f + 0$	
$5 \times f + 4$		$2 \times f + 1$
	$2 \times f + 0$	$2 \times f + 1$
	$2 \times f + 0$	

Figure 24: Second pass of clean-up conversion, values of the *offsetMap* data structure and affine index functions

"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 `E1[E2]` is identical (by definition) to `*((E1)+(E2))`.

Figure 25 shows a function with parameters `*px` and `*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.

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

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

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

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

Figure 25: Example of pointer clean-up conversion of arrays passed as function arguments

18.2 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 26: Pointer clean-up conversion of perfect loop nests

Figure 26 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 26.

```
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 27: General structure of perfect loop nest with pointer-based array access

```
for($i_1 = 0$; $i_1 < N_1$; $i_1 + +$)
{
   for($i_2 = 0$; $i_2 < N_2$; $i_2 + +$)
   {
      ⋮
         for($i_n = 0$; $i_n < N_n$; $i_n + +$)
         {
            ... array[($\sum_{j=1}^{n-1}(i_j \times N_{j+1}) + i_n + \Delta + \delta$)] ...
         }
   }
}
```

Figure 28: Perfect loop nest after pointer clean-up conversion

Resulting index functions are still affine functions, but they are now functions 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 27 and 28 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.

18.3 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 29 shows the structure of a general loop nest.

```
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($i_{22} = 0$; $i_{22} < N_{22}$; $i_{22}++$)
    {
        ⋮
    }
    $S_{13}$
    ⋮
}
```

Figure 29: 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,
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.

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

for (i = 0 ; i < 3; i++)              for (i = 0 ; i < 3; i++)
{                                     {
  p_x = &x[0]  ;

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

  p_y++ ;
}                                     }
```

Figure 30: Example of pointer clean-up conversion of general loop nests

An example is shown in figure 30. Since the pointer `p_x` is re-initialised within the `i`-loop but outside the `f`-loop, the access via `p_x` inside the inner loop body is only dependent on `f` and not on `i`. The opposite is true with the pointer access `p_y`. It is not changed in the `f`-loop, but in the `i`-loop. The access to the array `h` is dependent on `i` as well as on `f`.

18.4 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.

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

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

```

for (i = 0 ; i < N; i++)
{
   if ( ... )
   {
      ...  array[2*i+0] ...
      ...  array[2*i+1] ...
   }
   else
   {
      ...  array[2*i+0] ...
      ...  array[2*i+1] ...
   }
}
```

Figure 31: Example of pointer clean-up conversion of conditional branches

Figure 31 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.

`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.

18.5 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 32 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

```
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;
  }
}
```

```
int a[10][20];



for(i = 0; i < 10; i++)
{

  for(j = 0; j < 20; j++)
  {
     a[1*i+0][1*i+j] = 5;
  }
}
```

Figure 32: Example of pointer clean-up conversion of a 2-dimensional array

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 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.

19 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.

20 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 support Instruction Level Parallelism.

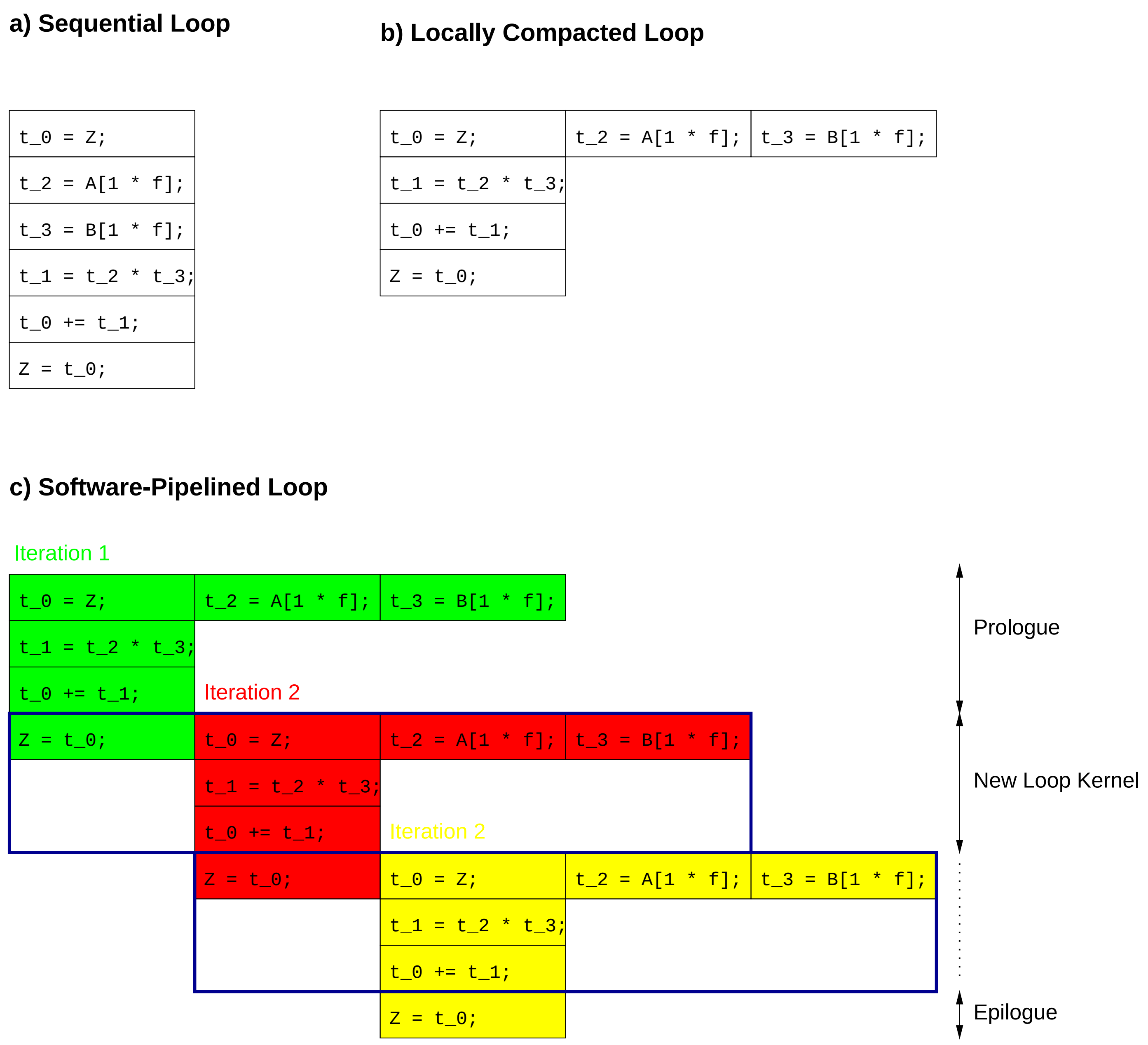
```
for (f  =  0; f < N; f++)
{
  t_0  =  Z ;
  t_2  =  A[1 * f] ;
  t_3  =  B[1 * f] ;
  t_1  =  t_2 * t_3 ;
  t_0  +=  t_1 ;
  Z  =  t_0 ;
}
```

```
t_0  =  Z ;
t_2  =  A[0] ;
t_3  =  B[0] ;
t_1  =  t_2 * t_3 ;
t_0  +=  t_1 ;
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 ;
```

Figure 33: Sequential Loop and Software-Pipelined Loop

Figure 33 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 34 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 [3] 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. 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

[3]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.



```
t_0 = Z;
t_2 = A[1 * f];
t_3 = B[1 * f];
t_1 = t_2 * t_3;
t_0 += t_1;
Z = t_0;
```

```
t_0 = Z;           t_2 = A[1 * f];  t_3 = B[1 * f];
t_1 = t_2 * t_3;
t_0 += t_1;
Z = t_0;
```

```
t_0 = Z;           t_2 = A[1 * f];  t_3 = B[1 * f];
t_1 = t_2 * t_3;
t_0 += t_1;
Z = t_0;           t_0 = Z;           t_2 = A[1 * f];  t_3 = B[1 * f];
                   t_1 = t_2 * t_3;
                   t_0 += t_1;
                   Z = t_0;           t_0 = Z;           t_2 = A[1 * f];  t_3 = B[1 * f];
                                      t_1 = t_2 * t_3;
                                      t_0 += t_1;
                                      Z = t_0;
```

Figure 34: Software-Pipelining of a simple loop

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:

$$\begin{aligned} Cost_{lc} &\geq Cost_{sp} \\ 4N &\geq 3(N-1)+4 \\ N &\geq 1 \end{aligned} \tag{36}$$

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 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 don 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.

21 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.

22 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 $RecMII$:

$$MII = \lceil max(ResMII, RecMII) \rceil \tag{37}$$

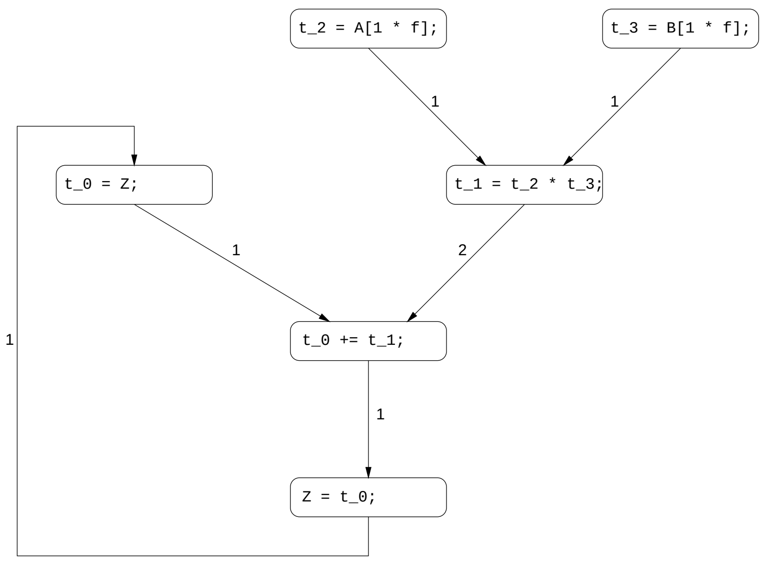
Figure 35 shows the *Data Dependence Graph (DDG)* and a Schedule for the loop of the introductory example. In the DDG the nodes represent instructions of the loop body and the edges dependences between instructions. In this example only true dependences are displayed. However, output and anti dependences 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.

22.1 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

$$ResMII = \max_{i \in \text{FU types}} \left(\frac{\text{\# FU type i required}}{\text{\# FU type i in hardware}} \right) \tag{38}$$

For the example in figure 35 this means to take the number of instructions and divide it by the number of functional units as all functional units are equivalent. Therefore, $ResMII = \frac{6}{4} = \frac{3}{2}$.



Cycle	Functional Unit 1	Functional Unit 2	Functional Unit 3	Functional Unit 4
1	t_0 = Z;	t_2 = A[1 * f];	t_3 = B[1 * f];	
2	t_1 = t_2 * t_3;			
3				
4	t_0 += t_1;			
5	Z = t_0;			

Figure 35: Data Dependence Graph and schedule with resource-constraints

22.2 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, $Delay(c)$ the total sum of latencies along that circuit and $Distance(c)$ 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 $Delay(c)$ later. Or written another way

$$II \times Distance(c) \geq Delay(c) \tag{39}$$

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:

$$RecMII = \max_{c \in \text{dependence circuits}} \left(\frac{Delay(c)}{Distance(c)} \right). \tag{40}$$

The example in figure 35 contains only a single dependence cycle[4]. The sum of latencies $Delay(c)$ on this cycle is 3, the number of different iterations $Distance(c)$ involved in it is 2. Hence, $RecMII = \frac{3}{2}$ holds. With ResMII and RecMII together, the MII for the example is calculated as

$$MII = \lceil max(ResMII, RecMII) \rceil = \lceil max(\frac{3}{2}, \frac{3}{2}) \rceil = 2 \tag{41}$$

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.

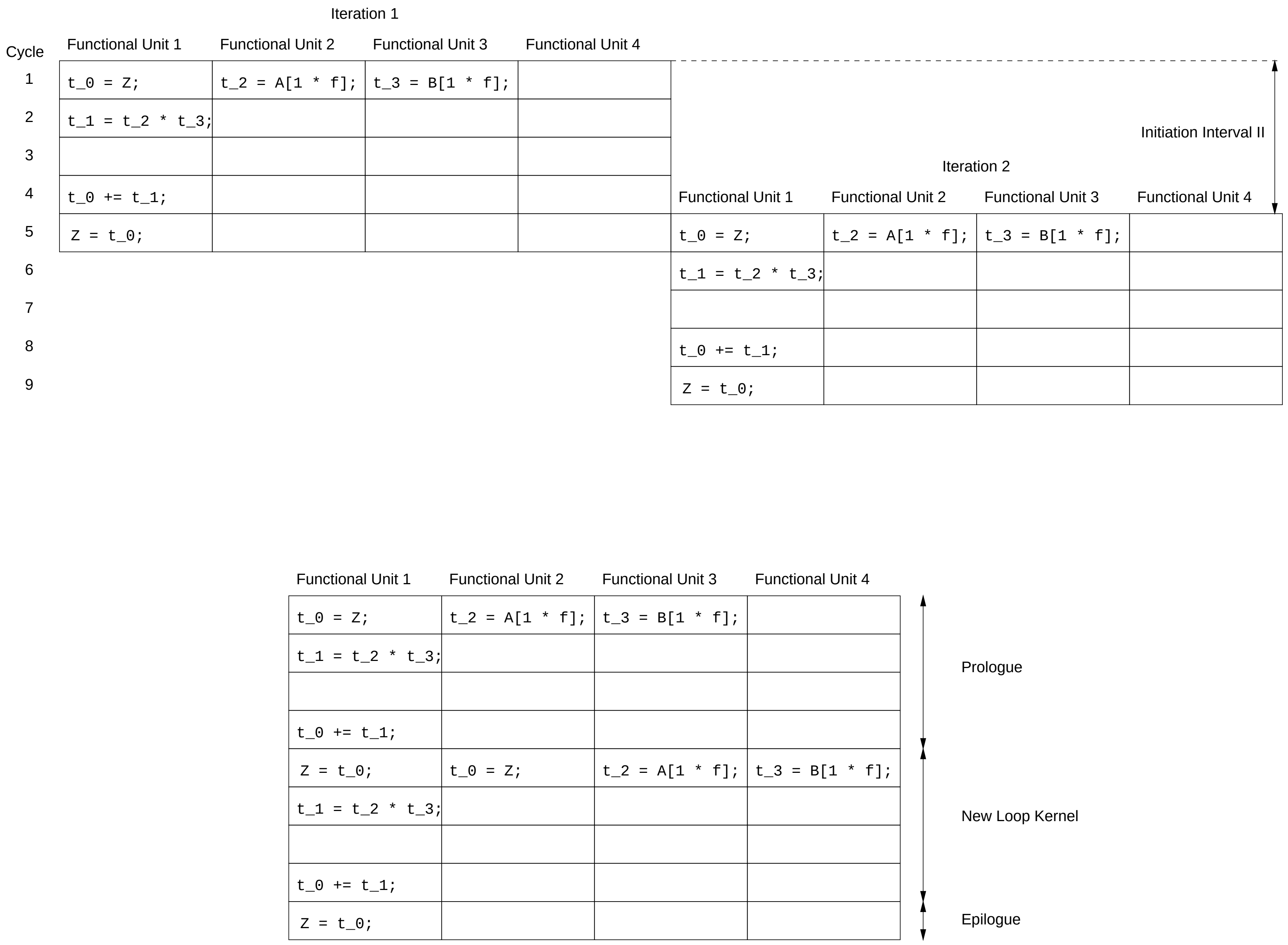
23 Scheduling a New Loop Kernel

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 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 36 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.

[4]Although there are actually more dependence cycles these are not shown in the figure in order to keep the example simple.



Iteration 1

Cycle	Functional Unit 1	Functional Unit 2	Functional Unit 3	Functional Unit 4
1	t_0 = Z;	t_2 = A[1 * f];	t_3 = B[1 * f];	
2	t_1 = t_2 * t_3;			
3				
4	t_0 += t_1;			
5	Z = t_0;			

Iteration 2

Cycle	Functional Unit 1	Functional Unit 2	Functional Unit 3	Functional Unit 4
5	t_0 = Z;	t_2 = A[1 * f];	t_3 = B[1 * f];	
6	t_1 = t_2 * t_3;			
7				
8	t_0 += t_1;			
9	Z = t_0;			

	Functional Unit 1	Functional Unit 2	Functional Unit 3	Functional Unit 4
Prologue	t_0 = Z;	t_2 = A[1 * f];	t_3 = B[1 * f];	
	t_1 = t_2 * t_3;			
	t_0 += t_1;			
New Loop Kernel	Z = t_0;	t_0 = Z;	t_2 = A[1 * f];	t_3 = B[1 * f];
	t_1 = t_2 * t_3;			
	t_0 += t_1;			
Epilogue	Z = t_0;			

Figure 36: Example of Modulo 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 a 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.

24 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 techniquse for handling loops with conditional branches [22] is discussed briefly in next section.

24.1 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 IIs 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 section 8) 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 37 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.

```
if (exp)
{
   Statement 1;
   ...
}
else
{
   Statement 2;
   ...
}
```

```
P = exp;
[P]  Statement 1;
[P]  ...
[!P] Statement 2;
[!P] ...
```

Figure 37: Example of *if-conversion*

Because many DSPs such as the Philips TriMedia (see section 8) 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.

25 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 adaptions 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 section 12 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 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[5]

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)* [] 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 section 6 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[6] 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 with only a few original statements. The *Abstract Syntax Tree* is traversed until a `for`-loop construct is

[5]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.

[6]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.

found. `while`- and `do-while`-loops are ignored, because they are 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 lenth 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 36. 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 transfered 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.

26 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.

27 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.

28 Observations and Results

In this sections 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.

28.1 Software-Pipelining

Figure 38 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 `O0`, `O1`, `O2` and `O3` (compare section 10).

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 `O1`. 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.

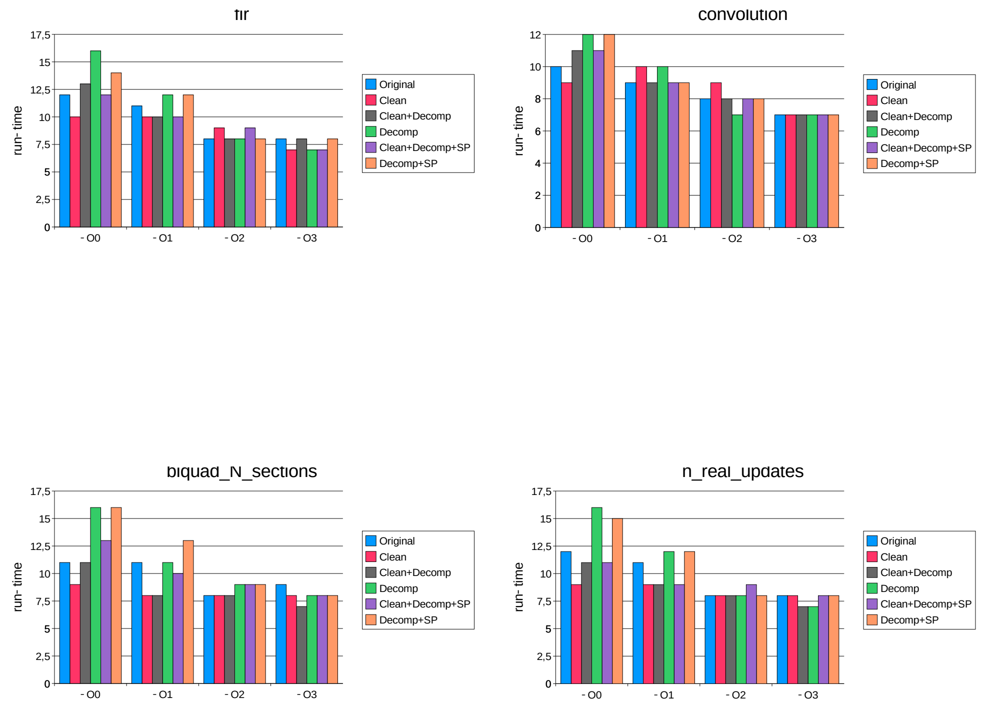


Figure 38: Results for `biquad_N_sections`, `n_real_updates`, `convolution` and `fir`

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 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 `O0` 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 `O0` 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 39. 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.

28.2 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 39, 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, `O1`, 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 40 shows the run-times of the `matrix1`, `matrix2`, `lms` and `mat1x3` programs. The original version is compared to the cleaned version for all available optimisation levels. For the levels `O0` and `O1` all cleaned programs take less cycles than the corresponding original programs. The difference

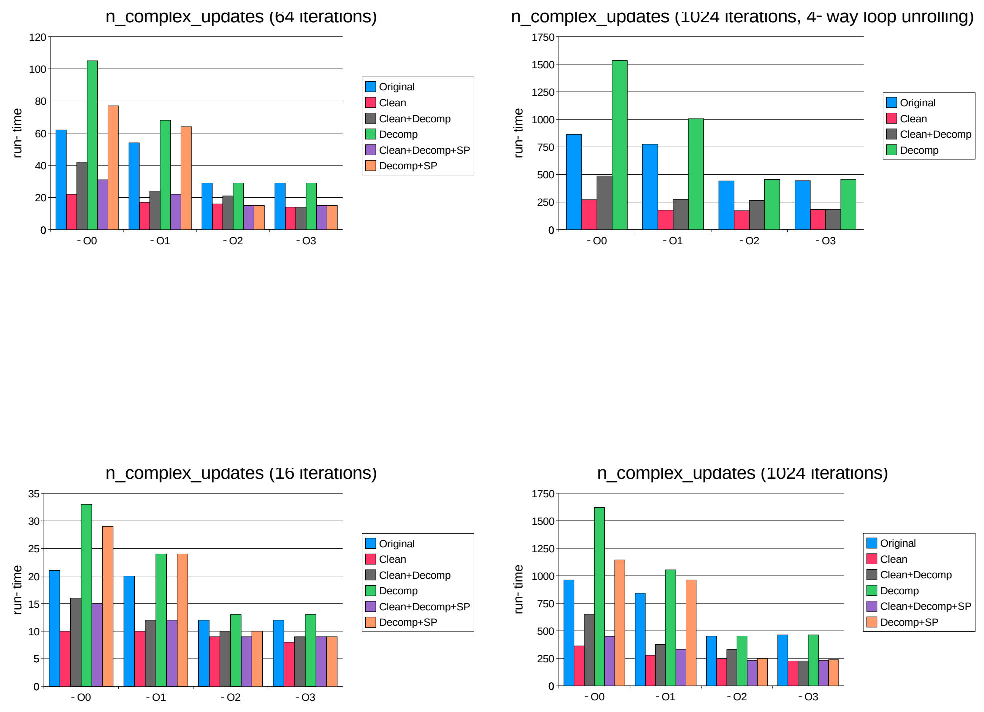


Figure 39: Results for different loop ranges of `n_complex_updates`

can be up to 50%. For higher optimisation levels the differences become smaller. At level `O2`, in particular, the original programs show either better or the same performance. On the contrary, at level `O3` 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 39, for more complex programs the conversion provides some substantial advantage.

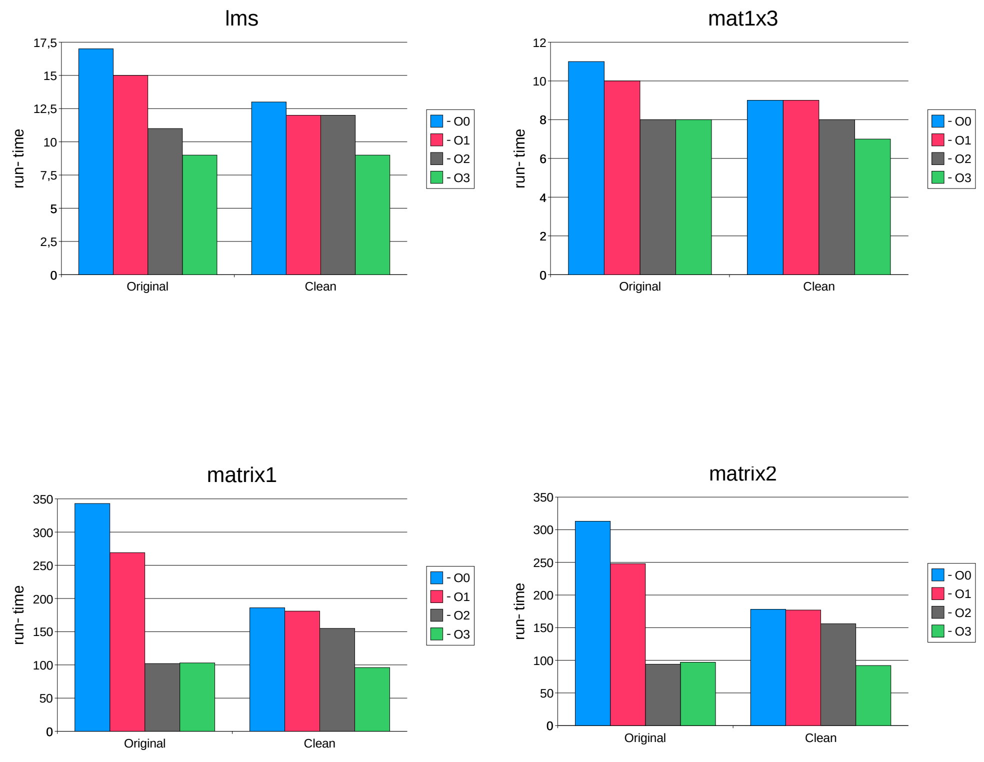


Figure 40: Results of pointer clean-up conversion for `matrix1`, `matrix2`, `lms` and `mat1x3`

28.3 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 section 6) 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, 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.

29 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 sub-optimal 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.

30 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 modell 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.

Part II
Optimization of DSP, Image Processing and 3D Applications on General-Purpose Processors with SIMD ISA Extensions

Abstract

This deliverable analyzes and quantifies the performance of DSP, image and 3D applications on recent general-purpose microprocessors using streaming SIMD ISA extensions (integer and floating point). We have evaluated the AltiVec extensions to PowerPC ISA on a set of 9 benchmarks. Our benchmark suite includes DSP kernels (FIR), multimedia kernels (IDCT, RGB2YCbCr) and a 3D geometry transformations kernel. The benchmark suite includes both integer and single precision floating point kernels in PPC and AltiVec versions. Each benchmark has been optimized for both ILP and memory hierarchy by using loop unrolling, software pipelining, blocked algorithms and data prefetch. We demonstrate that SIMD applications are very sensitive to memory bandwidth and that optimizations can speedup Altivec applications from 1.9 to 7.1.

31 Introduction

The SIMD (Single Instruction Multiple Data) extensions to general purpose microprocessor instruction sets have been introduced in 1995 to enhance the performance of multimedia and DSP applications. The goal for these extensions to general purpose microprocessors instruction sets is to accelerate DSP and multimedia application at a small cost. Traditionally in these processors, better ILP exploitation goes through increasing dispatch logic (reorder buffer size, reservation station size) and the number of functional units available. With SIMD instruction sets only the number of functional units is increased (the width of a functional unit). With constantly increasing available transistors in microprocessors there was room for the additional functional units needed by SIMD instruction sets. With SIMD extensions it's up to the programmer or the compiler to extract the ILP that is traditionally extracted by the out of order core of the microprocessor. Multimedia and DSP applications generally works one data vectors, so SIMD programmation is well adapted to these class of applications.

At the beginning, these extensions were able to process 64 bits integer vectors: Sun with VIS in 1995 [54], HP with MAX in 1995 [51] and Intel with MMX in 1997 [43]. The introduction of SIMD extensions able to process single precision floating point vectors is more recent: AMD with 3DNow! [16] and Intel with SSE [44] in 1998. The multimedia applications use more and more floating point computations. Currently the main target for these FP SIMD extensions is 3D polygonal rendering that is extensively used in CAD applications and games. The need for single precision floating point computing power is very important in this class of applications. Last year Motorola introduced AltiVec [57][4][41], which is the first SIMD instruction set that can handle both integer and floating point vectors.

Several studies have presented the performance of integer SIMD applications (multimedia applications with MMX in [10], image processing with VIS in [18]) and floating point SIMD applications with an ideal memory hierarchy (ILP sutdies on 3D geometry transformations in [18] and DSP and multimedia performance with AltiVec [61])

In this deliverable we study the performance of both integer and floating point applications through DSP, image and 3D kernels on a general purpose microprocessor with SIMD extensions with memory hierarchy. DSP, image and 3D are the 3 main classes of multimedia applications and multimedia is the main workload on home computers.

In the first section we present our baseline architecture and the tools we used for this study. Next we present our workloads: multimedia kernels originally written in C that we have optimized for AltiVec SIMD instruction set. These micro kernels are taken from 3D polygonal rendering, voice recognition, 3D sound effects, MPEG compression and decompression, and 2D image processing.

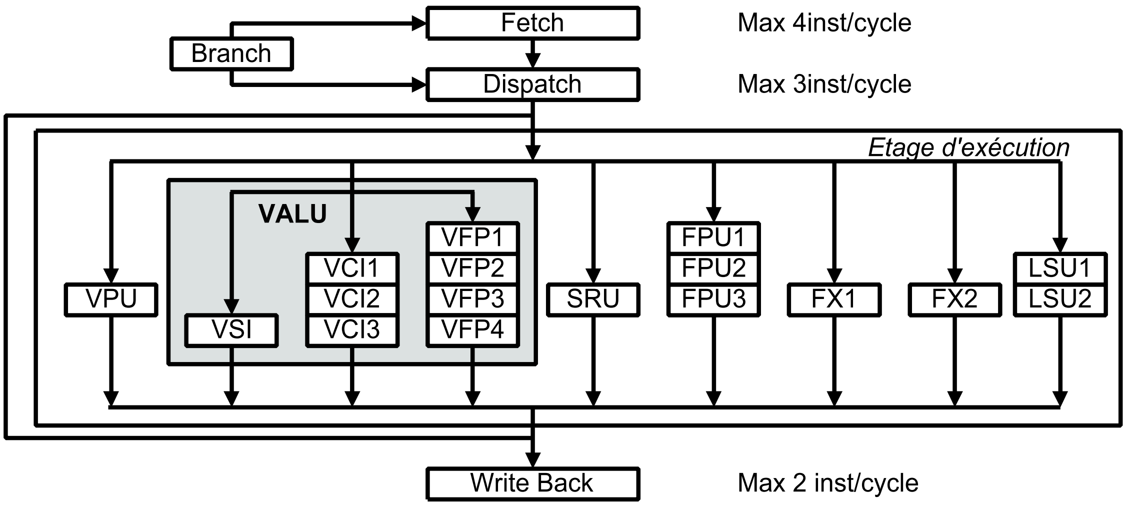


Figure 41: PowerPC G4 pipeline diagram

We detail the optimizations used for a better exploitation of SIMD characteristics and the problems related to them.

In the last section we study the impact of the memory hierarchy on the performance of the set of benchmarks. We will study more precisely the impact of the AltiVec streaming prefetch and the impact of the main memory bandwidth.

32 Methodology

32.1 AltiVec and G4 microarchitecture

AltiVec is a Single Instruction Multiple Data (SIMD) machine comprised of $32 \times 128 - bit$ "vector" registers, 2 fully pipelined processing units (a vector permute and an ALU) and 170 new instructions. These instructions comprise vector multiply accumulation, all kinds of permutations between two vectors, float to int conversions, pixel packing, logical operations, comparisons, selections and extensions from a scalar to a vector.

AltiVec vector registers can hold either 16 8-bits integers, 8 16 bits integers, 4 32 bits integer than 4 IEEE-754 single precision floats. AltiVec provides streaming prefetch instructions. The programmer can use these instructions to specify the amount of data to be prefetched into the first level cache, with an optional stride. The Motorola PowerPC G4 is the first microprocessor to implement AltiVec instruction set. The PowerPC G4 has a 4 stage pipeline (Fetch, Dispatch, Execution, Completion). There are six executions units and one reserving station per unit: 2 integer units (FX1 and FX2), 1 floating point unit (FPU), 1 Load-Store Unit (LSU), 1 Vector Permute Unit (VPU) and 1 Vector Arithmetic-Logic Unit (VALU). The VALU is subdivided into 3 sub-units that can not be used simultaneously: Vector Simple Integer (VSI), Vector Complex Integer (VCI) and Vector Floating Point (VFP). The latency for VSI is 1 cycle, 3 cycles for VCI and 4 cycles for VFP. The PowerPC G4 we used for all the tests is a 450MHz G4 with 32Kb L1 and 1024k L2 on a half speed dedicated bus, the memory bus uses PC100 SDRAM. There is more informations about the PowerPC G4 in [58].

32.2 Tools

32.2.1 The AltiVec programming interface

We have used Apple MrC compiler through a Metrowerks Codewarrior pro 4 plug-in. MrC allows easy Altivec programming through some extensions which enable the use of Altivec functions in a C program. The C programming model for Altivec used in MrC is designed to look like function calls, but each of this function call represents and generates one Altivec instruction. Variables are defined using C types that match Altivec data types and variables declared using these types are all 16-byte aligned. The sample following program performs the sum of two arrays of N float (N multiple of 4):

```
vector float A[N], B[N], C[N];
int i;
for ( i=0 ; i<N ; i++ )
      A[i] = vec_add( B[i] , C[i] );
```

As seen in this sample program, the programmer has not to do the registers allocation, and he can use the C control structures (here *for*).

32.2.2 Trace extraction and simulation environment

We have evaluated the performances of our multimedia kernel by execution on a real PowerPC G4. We have used trace driven simulation to estimate the impact of memory bandwidth impact on these kernel's performances. We couldn't modify the memory bandwidth of our PowerPC G4 (by changing the bus/processor ratio and the memory latency) so we used Motorola G4 simulator v1.1.2 and pitsTT6 v1.4 execution trace generator. PitsTT6 works as a shared library. A call to *startTrace()* launch the trace generation during the execution and a call to *stopTrace()* stops it. It generates a TT6 execution trace which we used as input for the G4 simulator.

The G4 simulator hasn't much parameters but it allows to change the bus/processor ratio and the timings of the SDRAM. We have used these two parameters to simulate increased bandwidth. By dividing the ratio by 2 and multiplying the main memory latency by 2 we obtained a doubled memory bandwidth. We have verified the validity of this model by simulating very simple micro-kernels like memory to memory copies.

33 SIMD optimizations for DSP image and 3D kernels

33.1 Workloads

We have used 9 micro-kernels taken from multimedia and DSP applications. There are 4 single precision floating point kernels and 5 integer kernels.

33.1.1 SOAD

This Sum Of Absolute Difference (SOAD) kernel is found in several reference video coder implementations such as MPEG2. It is used to compare the similarity of two blocks of pixels in motion-estimation algorithms, which is one of the most computationally intensive aspects of video encoding.

33.1.2 RGB2YCbCr

RGB2YCbCr converts from Gamma Corrected RGB to the YCbCr Color Space. RGB is generally used by monitors and represents the Red, Green, and Blue components of a color. YCbCr is used for transmission and storage. Y represents the luminence, with Cb representing the blue color difference, and Cr representing the red color difference. It is used in video compression algorithms.

33.1.3 IDCT

The Inverse Discrete Cosine Transform (IDCT) is used in MPEG2 decompression, it transforms an image from the DCT domain back to the spatial domain for image reconstruction. All calculations are performed in 16 bits fixed point arithmetic.

33.1.4 Image filters

The Image filters corresponds to gamma correction filters on static images. The sparse one works only on one component of the image (the green one) and the dense one works on the 3 components.

33.1.5 FIR

FIR and IIR filtering are two of the fundamental operations in digital signal processing. In FIR/IIR filters (a FIR filter is a subset of an IIR filter), a sequence of input digital samples is convolved with either one or two kernels of filter coefficients to produce a filtered output sequence. There are number of applications that use FIR/IIR filtering in digital signal processing. They are used by example into audio equalizer to filter, boost or attenuate some frequencies. This filter is also used in 3D positional audio effects, to make a sound appear to come from a specified location. This is done by applying different filters to the left and right stereo channels. Finally adaptative FIR/IIR are used in speech compression by the Linear Predictive Coding algorithm.

33.1.6 DAXPY

DAXPY is used in number of numerical algorithms, it is the heart of matrix-matrix products.

33.1.7 Max

The Max micro-kernel is used to find the maximum floating-point element of a vector and the elements corresponding index. The Max algorithm can be used in speech recognition. Finding the maximum or the minimum floating-point element in a vector (an array) is a very important step in the Viterbi algorithm. The Viterbi algorithm is often in speech recognition to find the best sequence of states in a model that best describes a speech sample [68][67].

33.1.8 3D Kernel

The 3D kernel performs the essential steps of the geometry part of the 3D polygonal graphic pipeline. 3D polygonal rendering is used by most of the CAO/CAD applications, virtual reality and a lot of games. Our 3D kernel performs the vertices transformation and perspective correction from 3D object space coordinates to 2D screen space coordinates, the normals transformation and renormalization, backface culling and phong lighting. We perform two steps of phong lighting to simulate a rendering where there is two dynamic lights. The C code we use comes from the Mesa 3D graphics Library [66] and the algorithms for 3D polygonal rendering are described in [28].

33.2 AltiVec features to raise the performances

33.2.1 SIMD processing and easy permutations

AltiVec provides SIMD instructions able to work on 4 to 16 element wide vectors depending on the data types. The programmer can permute each byte element of a vector as he wants. The instruction count should theoretically be reduced by the number of elements a vector contains. So it should be 4 for single precision floating point applications 16 for 8 bit integer applications and 8 for 16 bit applications. The theoretical speedups for programs that use SIMD instructions correspond to the maximum data vector width available for data types that are used. It is the instruction count reduction factor. The speedups won't be so high for the following reasons:

- data need often to be reorganized to be processed efficiently with the SIMD instructions available. This overhead limits the speedup of all the programs that need
- The ILP that can be extracted from each algorithm is variable and often the theoretical speedup isn't reachable because of the instructions dependency graph that limits the ILP.

33.2.2 Specialized instructions

SIMD instruction sets extensions generally contains more than just SIMD instructions. They provide some specialized instruction that accelerates much the processing of most DSP and multimedia algorithms. These instructions, if only present in the SIMD extensions can raise the speedup of the SIMD version of the programs over the theoretical speedup. These specialized instructions are:

- meta-instructions: these instructions perform multiple operation at a time, the multiply accumulation is the most known example. There are instructions that perform $\frac{1}{\sqrt{x}}$ that accelerates much some 3D geometry processing that cant be found in non-SIMD instruction sets.
- *"vector if"* without branch: by using masking and selecting instructions the programmer can perform conditional tests on individual vector elements without any branch instruction. The overhead for these instructions is that all computations are always done on each element of the vector even if it is unuseful, and finally the useful results are selected in the resulting vector by using the comparison mask generated for the *"if"*. By reducing the number of branch in the heart of loops, the average fetched instruction count per cycle is increased.
- Type conversion, vector packing and unpacking. SIMD instruction sets provide float to int and int to float conversions and packing, unpacking instructions with at the user choice saturated arithmetics or not. Saturated arithmetics and type conversions are generally costly operations and are much used in multimedia algorithms that works on floats or 32 bit integers and that need the results to be converted into 8-bits integer to be displayed at screen.

33.2.3 Large register file

The SIMD instruction sets use large register files with currently four times more space than conventional register files (same amount of registers, 4 times wider). This additional register space can be used in algorithms by maximizing the amount of data stored in registers. That reduces the first level cache miss rate and can raise much the performances of some algorithms.

33.3 Optimizations

We have tried to optimize both AltiVec and C versions of our micro-kernels. All the optimizations techniques are commonly used to optimize non-SIMD code. We have used loop unrolling for all the kernels and software pipelining when it was possible. We tried to maximize the number of register used. The compiler did not detect all the loop invariants, so we forced the use of registers to store these loop invariants. When using the AltiVec programming model, we have not to specify explicitly when the data have to be loaded or stored. This is done automatically by the compiler, but not always in the optimal manner. So we always forced the load and store by loading and storing explicitly the data into registers. We did not try to schedule the instructions by hand, the out of order execution does it automatically. The previous sum of two arrays algorithm optimized will look like:

```
vector float A[N], B[N], C[N];
register vector float a0,a1,a2,a3,b0,b1,b2,b3,c0,c1,c2,c3;
int ia,ib,ic;

for ( ia=&(A[0]),ib=&(B[0]),ic=&(C[0]) ;
      i<=&(A[N]) ;
      ia+=64,ib+=64,ic+=64 )
{
/* loading B[i..i+3] */
    b0=vec_ldx(ib,0);b1=vec_ldx(ib,1);
    b2=vec_ldx(ib,2);b3=vec_ldx(ib,3);
/* loading C[i..i+3] */
    c0=vec_ldx(ic,0);c1=vec_ldx(ic,1);
    c2=vec_ldx(ic,2);c3=vec_ldx(ic,3);

    a0=vec_add(b0,c0);a1=vec_add(b1,c1);
    a2=vec_add(b2,c2);a3=vec_add(b3,c3);
/* storing A[i..i+3] */
    vec_stvx(a0,ia,0);vec_stvx(a1,ia,1);
    vec_stvx(a2,ia,2);vec_stvx(a3,ia,3);
}
```

We use blocked algorithms when there was some temporal locality in our algorithms (essentially 3D kernel). We have used two kinds of parallelization to write the SIMD versions of all the micro kernels: the intra-iteration parallelization and the inter-iterations parallelization when it was possible. The inter-iteration parallelization is usable only when there are no inter-iteration dependencies. The instruction dependency graph of an iteration is the same in SIMD when using inter-iterations parallelization than in C, so the performance of this parallelization is highly predictable (assuming a perfect memory hierarchy).

The DSP algorithms generally have these kind of dependencies (all the convolutions by example), so we have used intra-iteration parallelization (FIR). The parallelism extractible by this way is generally lower than by the inter-iteration way.

For the 3D kernel we have tried both kind of parallelization. Traditionally the 3D data are stocked in Arrays Of Structures (AOS). Intel has evaluated in [45] that the Structure Of Arrays (SOA) is a more efficient way to store 3D data for SIMD geometry transformations. When in AOS the parallelization used is intra-iteration and when in SOA inter-iterations is used. One of our goal

Benchmark	Dataset Size (MBytes)	Description
SOAD	1	1 MPEG2 frame
RGB2YCbCr	9	1 TVHD frame
IDCT	8	8 MPEG2 frame
Img sparse	9	1 TVHD frame
Img dense	9	1 TVHD frame
FIR	10	2400000 elements
DAXPY	8	2000000 elements
Max	16	4000000 elements
3D	10	512000 vertices scene

Table 3: Datasets for the 9 benchmarks

is the integration of the AltiVec geometry transformations into Mesa, so we had to use the AOS structures of Mesa. We have evaluated that it is more efficient to convert AOS to SOA at the beginning of the pipeline and back SOA to AOS at the end and use inter-iterations parallelization than using intra-iteration with AOS. The overhead involved by this data transposition is made of vector permutation instructions. These instructions can be executed in parallel with vector computations in the PowerPC G4, so the impact of these transposition on final performance is low.

34 Performance evaluation

34.1 Execution on a PPC G4

We have measured the execution time of the 9 micro benchmarks on a PowerMac G4 450 with 1MByte of L2 cache. We have run each benchmark in its C version and AltiVec version with prefetch on and off. All the results are given as speedups against the C version without prefetch. We have used the datasets presented in Table 3. For all the benchmarks, the L1 misses are only cold misses because multimedia applications exploits mainly spatial locality via data streams.

The speedups range from 1.8 to 4.9 for single precision floating point kernels and from 1.9 to 7.1 for integer ones. The results are shown in figure 42.

Generally we have measured a speedup lower than the theoretical one. It is between 2.75 and 3.5 for floating point applications. FIR has the biggest instruction count reduction because there is no need for data reorganization because of the intra-iteration parallelization. Its speedup does not reach the theoretical because the loop cannot be unrolled so the AltiVec functional units stay unoccupied most of the time. The 3D kernel that uses inter-iteration parallelization needs data reorganizations, that is an overhead in instruction count. The dependency graph between instructions is the same in AltiVec than in C (omitting the Permutations for reorganization) so the speedup is essentially limited by the overhead of the reorganization. AltiVec provides an independent permutation unit so a part of the reorganizations can be done in parallel with the computations. Max and DAXPY are more memory intensive than computation, so the speedup is limited by the memory hierarchy performance (cf. next section). For the integer benchmarks, IDCT is very computational and well parallelizable without any overhead, the number of executed instructions is reduced by a 8 factor and the speedup is 7.1. The sparse filter isn't much efficient because it performs a lot of computations for nothing, just because the data structure is sparse. The AltiVec instruction count is near the one for the dense filter, versus a 4 times reduction for the PPC version. The two last benchmarks have an instruction reduction count of about 16, but

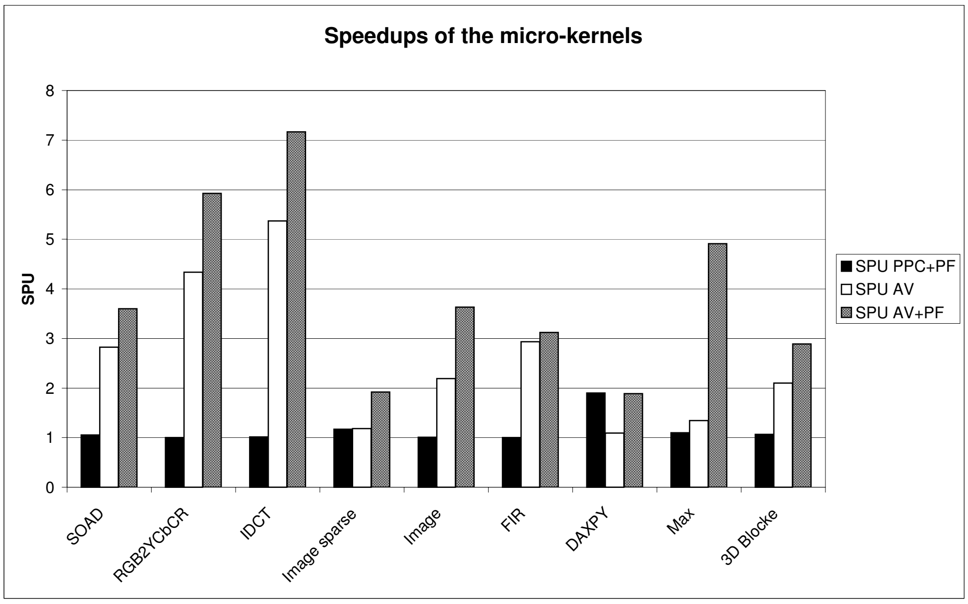


Figure 42: Performance of the 9 kernels on a PowerMac G4 450, speedup over the C version.

they become memory bound faster than computation bound. The 3D kernel uses some AltiVec specialized instructions as $frac1sqrtx$ in the normals transformation process. The clipping step uses intensively masking and selection instructions to perform "vector if". The phong lighting step uses vector packing, float to int conversions and saturated arithmetics. The part of the phong algorithm that uses these instructions has a speedup of 14, and the phong lighting step has a global speedup of 7.4. The functions we have implemented in our 3D kernels represent 85% of the time of the geometry transformations into the Mesa 3D graphic Library. The integration of our AltiVec functions into Mesa should give over a 100% increase in performance.

34.2 Latency problems: Streaming prefetch impact

Our benchmarks are multimedia applications that have streaming data acess. We use blocked algorithm to avoid cache misses other than cold ones. Prefetch is a well known solution to memory latency problems. In all our algorithms we have measured the impact of AltiVec streaming prefetch. We have chosen to launch streaming prefetch each 32Bytes of data.The impact on performance of enabling prefetch is shown in Table 4. The impact of prefetch is much more important for AltiVec programs than PPC one but DAXPY.

DAXPY is memory bandwidth limited in both AltiVec and PPC version, and this limit is reached when activating prefetch (the performance are the same for PPC with prefetch than AltiVec with prefetch) so this explains why the speedup due to prefetch is lower for AltiVec version (the version without prefetch is faster in AltiVec).

Max in its PPC version is limited by the branch mispredictions that reduce the average number of instruction fetched each cycle. We see that the impact of prefetch is not very high because memory latency impact is much lower than branch mispredictions impact. Max has no mispredicted branches in its AltiVec version so the fetch does not limiting the performance. The limiting factor becomes the memory latency, most of the time is spent in waiting for the data to come from main

	C (PPC)	AltiVec
SOAD	1.05	1.27
RGB2YCbCr	1.00	1.37
IDCT	1.01	1.34
Img sparse	1.17	1.62
Img dense	1.01	1.66
FIR	1.00	1.06
DAXPY	1.90	1.72
Max	1.10	3.65
3D	1.07	1.37

Table 4: speedup due to AltiVec streaming prefetch

memory. This explains the great impact of prefetch in the AltiVec version of this Kernel.

There are some kernels on which the activation of the streaming prefetch has a negative impact on the performance. The activation of prefetch generates a heavy load on the Load/store unit and on SOAD and the dense image filter (PPC) it has a negative impact on performance. On the other side, FIR performs a lot of computations per data and it is not much dependent of the memory latency and the impact of prefetch on performance remains low for both versions.

For the PPC version the impact is greater in the sparse version of the image filter than for the dense version. In the dense version the first level cache hit rate is very high without prefetch (1 miss / 32 load), for the sparse version it is only 1/8 so the prefetch increases relatively more the L1 hit rate for the sparse filter. For the AltiVec version the acceleration due to prefetch are very similar because the data are accessed by 128 bit vectors even if they will not be processed (sparse filter). There are a little more computation in the sparse version (masking and selecting the element to be processed) so the impact of prefetch is reduced compared to the dense filter (memory access are less important).

For the rest of the kernels the impact of streaming prefetch for PPC versions is under 10% and between 27% and 66% for AltiVec ones. This is a great improvement in performance for AltiVec programs at a low programmer's effort cost.

34.3 AltiVec and memory bandwidth

We have seen that streaming prefetch has a great impact on AltiVec programs performance. AltiVec programs are much more sensible to memory latency than the PPC ones, but what impact has memory bandwidth? With streaming prefetch enabled is it the main factor limiting the performance of the AltiVec programs. We couldn't change the available memory bandwidth on our G4 platform, so we used Motorola G4 simulator with the same kernels to evaluate the impact of memory bandwidth on our micro-kernels. We have used pitsTT6 execution trace as input to the simulator. We have simulated the various memory bandwidth by changing the processor/memory ratio and the SDRAM timings because there was no other ways of modifying the relative memory bandwidth available. We have simulated a halved, doubled and quadrupled memory bandwidth. We have simulated a perfect memory hierarchy too (all data in L1 cache) to get a upper bound of the performance and see how far of this bound we stay.

All the results are in Figure 43 and Figure 44. By looking to the speedup due to perfect memory hierarchy we can know if the kernel is memory bound or computation bound. SOAD, RGB2YCbCr and 3D are memory bound in their AltiVec version and DAXPY and Max both in PPC and AltiVec

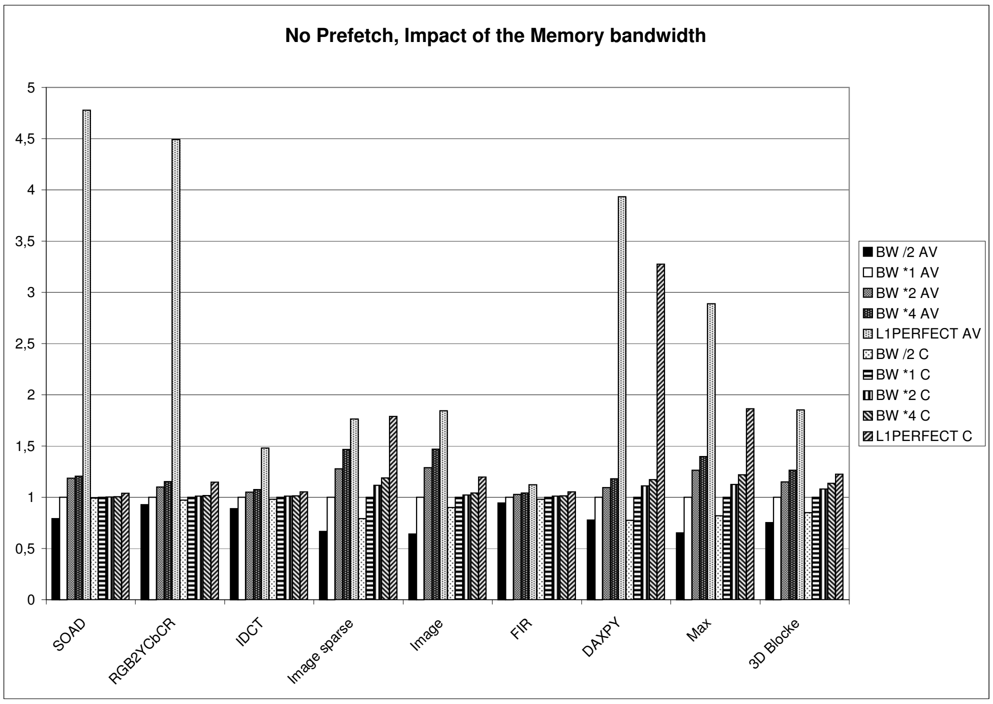


Figure 43: Impact of the memory bandwidth for the 9 kernels, prefetch off.

version.

On most of the benchmark memory bandwidth increase has much more impact on AltiVec performance than PPC. It is not a surprising result because the time spent in computations relatively to the memory accesses in AltiVec programs is much lower than in PPC due to SIMD processing. DAXPY and 3D kernel are sensible to memory bandwidth increase in both C and AltiVec version, but the impact is greater for AltiVec versions. SOAD, RGB2YCbCr, Image filters and Max are sensible to memory bandwidth increase only in their AltiVec versions when prefetch is activated. The two remaining programs IDCT and FIR are essentially computation bound, the impact of memory bandwidth is still greater in AltiVec than in PPC but it is quite negligeable.

The increase of main memory bandwidth has more impact when prefetch is activated for both C and AltiVec versions. When prefetch is used, memory access are more regular and optimized, the computations are accelerated because of the memory latency reduction, so the need for high bandwidth is more important.

With a quadrupled bandwidth the half of the programs have performance comparable to the performance with perfect memory hierarchy. SOAD, RGB2YCbCr, DAXPY, Max, 3D kernel have more requirements than a quadrupled memory bandwidth.

That shows that the main factor limiting the performance of multimedia, DSP and 3D applications with AltiVec is the main memory bus bandwidth. Currently the PowerPC G4 is limited by its 100MHz 64bit with SDRAM main memory bus on the studied applications. The only tested applications that have really no need for an increased memory bandwidth are applications that are only computation bound. The G4 bus is well designed for most of our applications when we don't use AltiVec but does not meet the requirements for the AltiVec versions. The memory technology that offer a doubled memory bandwidth are currently available with DDRSDRAM [73][55] and

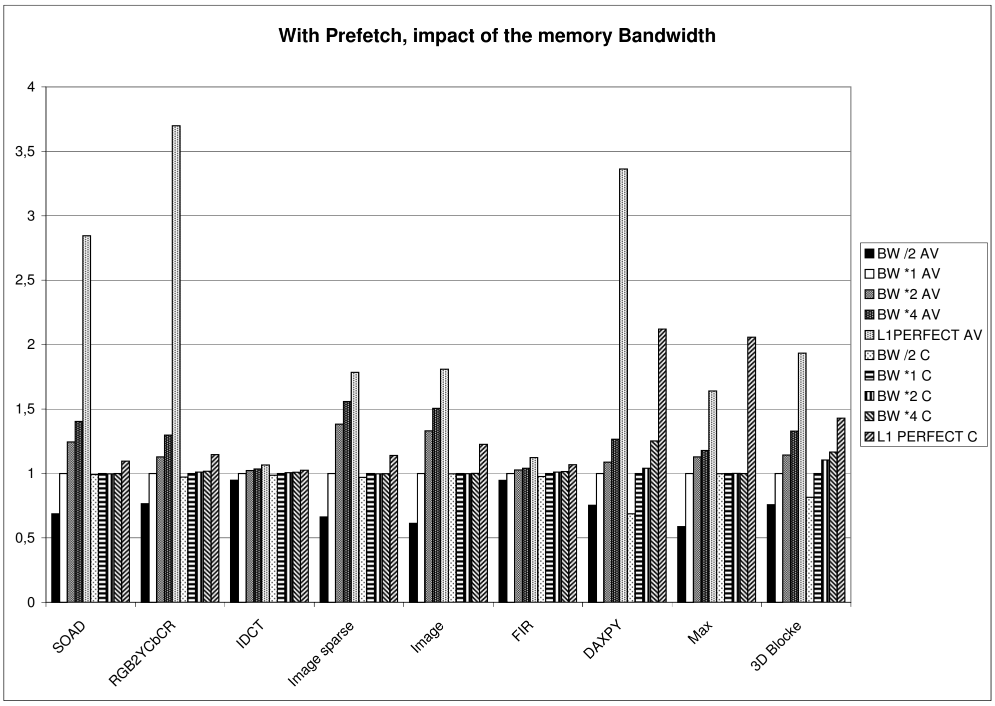


Figure 44: Impact of the memory bandwidth for the 9 kernels, prefetch on.

RAMBUS [24]. By using such memory technology some application performance would increase of up to 40% if the latency stays the same as the SDRAM (it is the case for DDRSDRAM).

35 Conclusion

We have run 9 benchmarks on a PowerPC G4 in both AltiVec and PPC versions. These benchmarks are kernels taken from multimedia applications. The speedups achieved range from 1.9 to 7.1 with prefetch activated. We have explained the optimization techniques that allow this range of speedup and evaluated the impact of memory hierarchy on the performance of our DSP, multimedia and 3D benchmarks. These techniques are well known and commonly used on non-SIMD code but give interesting performance with SIMD multimedia applications. Finally after the software optimizations study we have studied how our programs map on G4 architecture and evaluated that the memory bandwidth available on a PowerPC G4 is sufficient for PPC applications but becomes the main bottleneck for AltiVec versions of our programs. By using higher bandwidth memory like DDRSDRAM or RAMBUS which provide up to a quadrupled bandwidth, the performance could increases of up to 55%.

Part III
CME Driven Optimizations

Abstract

The effectiveness of the memory hierarchy is critical for the performance of current processors. The performance of the memory hierarchy can be improved by means of program transformations such as padding and loop tiling, which are code transformations targeted to reduce conflict and capacity misses respectively. This chapter presents novel approaches to perform near-optimal padding and loop tiling based on Cache Miss Equations and genetic algorithms. The results show that they can remove practically all replacement misses (conflict and capacity) in some kernels from different benchmarks. The reduction of replacement misses results in a decrease of the miss ratio that can be as significant as a factor of 9 for the Swim program and a particular cache architecture by means of padding and a factor of 7 for the matrix multiply kernel by means of loop tiling.

36 Introduction

Memory performance is critical for the performance of current computers. Memory is organized hierarchically in such a way that the upper levels are smaller and faster. The uppermost level typically has a very short latency (e.g. 1-2 cycles) but the latency of the lower levels may be a few orders of magnitude longer (e.g. main memory latency may be around 100 cycles). Thus, techniques to keep as much data as possible in the uppermost levels are key to performance.

In addition to the hardware organization, it is well known that the performance of the memory hierarchy is very sensitive to the particular memory reference patterns of each program. The reference patterns of a given program can be changed by means of transformations that do not alter the semantics of the program. These program transformations can modify the order in which some computations are performed or can simply change the data layout. *Loop Tiling* and *Padding* are examples of these families of techniques respectively. Loop tiling is based on a combination of strip-mining and loop interchange, and padding is based on adding some dummy elements between variables (inter-variable padding) or between elements of the same variable (intra-variable padding).

Loop tiling and padding have a significant potential to remove cache misses. Loop tiling can remove most capacity misses by restructuring the loop and changing the order in which statements are executed. On the other hand, padding can remove most conflict misses by changing the address of conflicting data, and some compulsory misses by aligning data with cache lines. However, finding the optimal loop tiling or padding for a given program is a very complex task, since the options are almost unlimited and exploring all of them is infeasible. For very simple programs, the programmer intuition may help but in general, a systematic approach that can be integrated into a compiler and can deal with any type of program is desirable. This systematic approach requires the support of a locality analysis method in order to assess the performance of different alternatives.

In this chapter, we propose automatic approaches to perform loop tiling and padding in numeric codes. They are based on a very accurate technique to analyze the locality of a program that is known as Cache Miss Equations (CMEs) and a genetic algorithm in order to search the solution space. The proposed genetic algorithm converges very fast and although it does not guarantee that the optimal solution is found, we show that after loop tiling or padding, the replacement miss ratio of the evaluated benchmarks is almost negligible. Therefore, for these programs the results are near-optimal. The rest of this chapter is organized as follows. Section 37 overviews the locality analysis approach. Section 38 presents the padding technique and its performance is evaluated in section 39. Section 40 presents the loop tiling technique and its performance is evaluated in section 41. Section 42 summarizes the main conclusions of this work.

37 Memory Locality Analysis

To effectively transform a code in order to optimize memory performance, a good memory locality analysis is required. In this section, we describe the locality analysis technique used in this work to perform loop tiling and padding.

In particular, we use Cache Miss Equations (CMEs) [31] to represent the cache behavior. Cache Miss Equations are a very accurate analytical model of the cache memory. They describe the cache behavior by means of diophantine equations, which allows us to use mathematical techniques to compute the locality of each memory reference. Unfortunately a direct solution of these equations is computationally intractable due to its NP nature.

37.1 Overview of Cache Miss Equations

Cache Miss Equations are a set of equations[7] that represent all the potential cache misses for the references in a loop nest. They describe the precise relationship among the iteration space, array sizes, base addresses, and the cache parameters for a loop nest. This section presents an overview of the CMEs. For more details about CMEs see [31, 32].

37.2 Finding Cache Misses from CMEs

Deciding whether a reference causes a miss or a hit for a given iteration point is equivalent to deciding whether this iteration point belongs to the polyhedra defined by the CMEs. The points inside each CMEs polyhedron represent the potential cache misses (the number of points is the number of potential cache misses). This leads us to consider several ways for computing them. Our approach builds upon traversing the iteration space to solve the equations. This approach to solve the CMEs allows studying each reference in a particular iteration point independently of all other memory references and iteration points. Details can be found in the M3D1 deliverable.

37.3 CME for multiple convex regions

The implemented technique to count points in polyhedra requires that the iteration space is a convex region, but after tiling n dimensions the iteration space is the union of 2^n convex regions.

Figure 45 shows how the iteration space of a one-dimension loop becomes a two-convex region iteration space after tiling. The shaded regions correspond to the different convex regions before and after tiling.

Because of this, the CME implementation has been modified to deal with multiple convex region by defining the equations for every convex region and solving for every analyzed point the equations corresponding to the convex region in which the point is contained.

38 Padding

This section presents our method for guiding both inter- and intra-variable padding. In this paper we refer to the cache size as C_s. mem_i is the original base address of variable number i (Var_i) and P_Base_i stands for the inter-variable padding between Var_i and Var_{i-1}. dim_{ij} stands for the size of the dimension j of Var_i (D_i is the number of dimensions) and S_i is its size. P_Dim_{ij} is the intra-variable padding applied to dim_{ij}, and P_S_i is the size of Var_i after padding (see Figure 46). We define Δ_i as $P\_S_i - S_i$.

[7]The term equation has been loosely used to refer to a set of simultaneous equalities and inequalities.

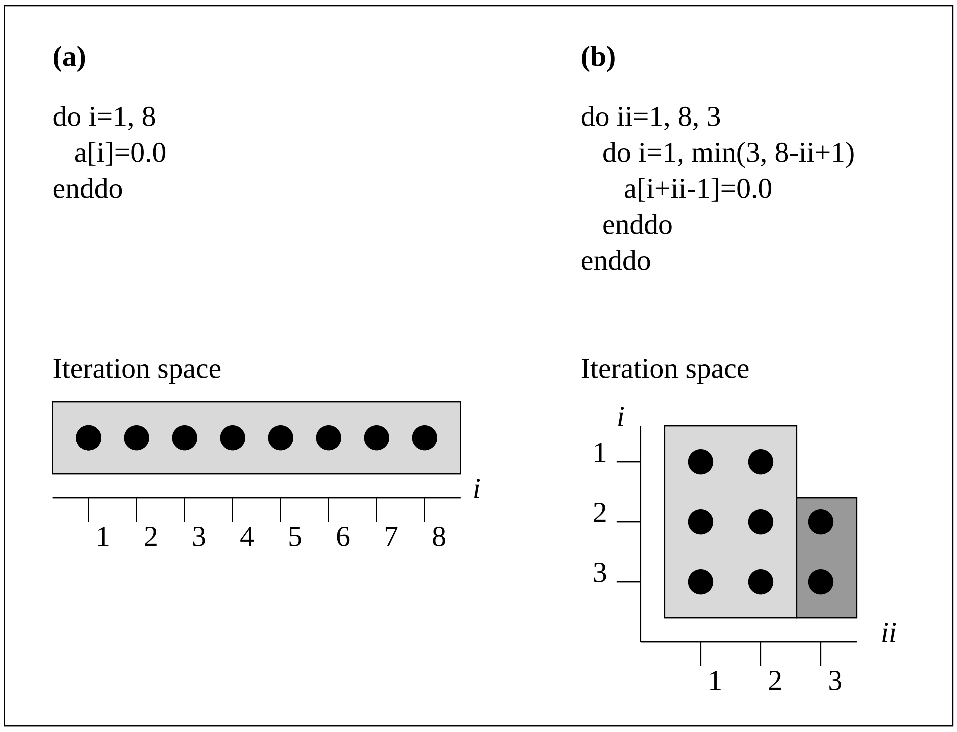


Figure 45: Example of iteration space: (a) before tiling, (b) after tiling

38.1 Inter-variable padding

When inter-variable padding is applied only the base addresses of the variables are changed. Thus, padding is performed in a simple way. Memory variable base addresses are initially defined using the values given by the compiler. Then, we define for each memory variable Var_i, a variable P_Base_i, $i = 0 \dots k$:

$$0 \leq P\_Base_i \leq C_s - 1$$

Note that padding a variable is equivalent to modifying the initial addresses of the other variables (see Figure 46). Thus, after padding, the memory variable base addresses are computed as follows:

$$BaseAddr(Var_i) = mem_i + \sum_{k=0}^{k \leq i} P\_Base_k$$

38.2 Adding intra-variable padding

The result of applying both inter- and intra-variable padding is that all base addresses and sizes of every dimension of each memory variable may change. They are initially set according to the values given by the compiler. For each memory variable $Var_i, i = 0 \dots k$ we define a set of variables $\{P\_Base_i, P\_Dim_{ij}\}$, $j = 0 \dots D_i$

$$0 \leq P\_Base_i, P\_Dim_{ij} \leq C_s - 1$$

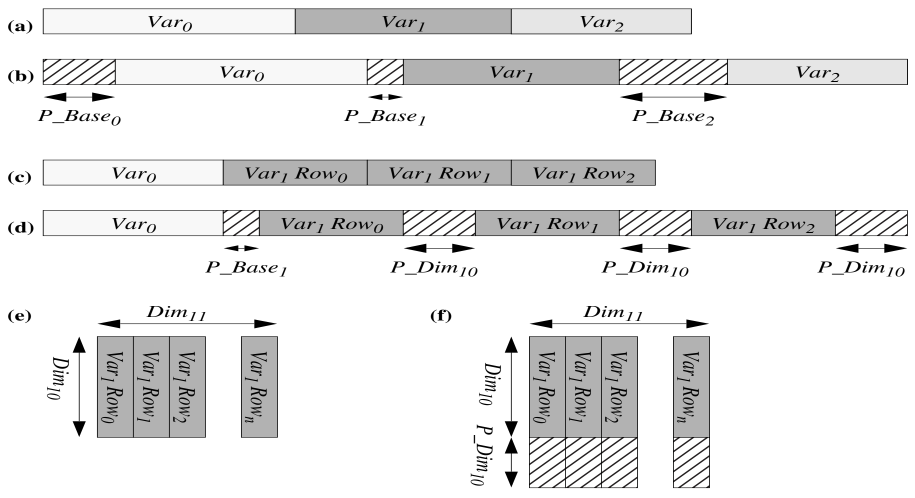


Figure 46: Data layout: (a) before inter-variable padding, (b) after inter-variable padding (c) before padding, (d) after padding, (e) 2-D array, (f) 2-D array after intra-variable padding

After padding, memory variable base addresses are computed in the following way(see Figure 46):

$$BaseAddr(Var_i) = mem_i + \\ + \sum_{k=0}^{k<i} (P\_Base_k + \Delta_k) + P\_Base_i$$

and the size of the dimensions are:

$$Dim_i(Var_j) = dim_{ji} + P\_Dim_{ji}$$

38.3 Model

For the sake of uniformity in the analysis presented here, we assume that both inter- and intra-variable padding are applied[8]. Our work focuses in obtaining the values of the variables

$$\{P\_Base_i, P\_Dim_{ij}\}$$

that minimizes the number of misses.

Let f be the function that represents the number of misses for each possible value of the padding variables:

$$f \quad \longmapsto \quad \#Misses \tag{42}$$

$$f(\underbrace{[0, C_s - 1]}_{P\_Base_0} \times \underbrace{[0, C_s - 1]^{D_0}}_{P\_Dim_{0j}} \times \ldots \times \underbrace{[0, C_s - 1]}_{P\_Base_k} \times \underbrace{[0, C_s - 1]^{D_k}}_{P\_Dim_{kj}}) = \\ = f(P\_Base_0, \underbrace{P\_Dim_{0j}}_{D_0}, \ldots, P\_Base_k, \underbrace{P\_Dim_{kj}}_{D_k})$$

Note that $[0, C_s - 1]^{D_i}$ represents the domain of the different P_Dim_{ij} of the variable Var_i.

[8]To apply only inter-variable padding, set all P_Dim_{ij} to 0

Our problem can be expressed as follows:

$$MIN \quad f(P\_Base_0, \underbrace{P\_Dim_{0j}}_{D_0} \dots , P\_Base_k, \underbrace{P\_Dim_{kj}}_{D_k})$$

$$0 \leq P\_Base_i, P\_Dim_{ij} \leq C_s - 1$$

$$i = 0 \dots k$$

where f is called the *objective function*.

Since f is a pseudo-polynomial function [20], the relationship between padding and the number of misses is nonlinear. P_Base_i and P_Dim_{ij} can take only integer values, thus, our problem can be seen as a nonlinear integer optimization (NLP) one.

Many researchers have studied NLP [7, 33]. A well studied case is the one where the constraints are linear (named *linearly constrained optimization*). A special case is when the objective function is entirely linear; this is called Linear Programming (LP). Algorithms to solve both real (e.g simplex [60]) and integer functions (e.g branch & bound scheme [74]) can be found in the literature.

One of the challenges in NLP is that some problems exhibit local minima. Algorithms that propose to overcome this problem are named *Global Optimization*. Real functions have been studied deeply [75, 40, 33]. Unfortunately, integer functions are hard to optimize. There are some studies based on {0,1} valued integer functions [38], but in general, this is a hard and time-consuming problem. Hence, the use of heuristics is necessary. Tabu search [34] obtains promising theoretical results, but only partial implementations have been reported so far. On the other hand, simulated annealing [46] and genetic algorithms [35, 39] have been used for years with very good results.

Our proposal is based on the use of a genetic algorithm to optimize function f.

38.4 Genetic Algorithm

Algorithms for function optimization are generally limited to convex regular functions. However, there are lots of functions that are not continuous, non differentiable or multi-modal. It is common to solve this problem by means of stochastic sampling. Whereas traditional search techniques use characteristics of the problem to determine the next sampling point (e.g Gradient), stochastic methods use non-deterministic decision rules [27].

Genetic Algorithms (GAs) are a particular type of stochastic methods that have been used to solve hard problems with objective functions that do not meet the properties to use traditional methods [35]. These algorithms search in the solution space of a function simulating the Nature-based process of evolution, that is, the survival of the fittest. Usually, the fittest individuals tend to reproduce more than the inferior individuals, and they survive to the next generation propagating the best genes.

GAs simulate the evolution of a population. Figure 47 shows the simplest GA. It starts from a random generated population, and it makes the population evolve by means of basic genetic operators (selection, mutation and crossover) [35] applied to individuals of the current population, to produce an improved next generation.

38.5 Genetic Algorithm Parameters

The use of GAs requires the determination of the following issues: chromosome representation, selection function, genetic operators, the creation of the initial population and the termination criteria.

```
ALGORITHM:
Supply a population $P_0$
i=1
while (not finish)
    $P_i$=Selection($P_{i-1}$)
    $P_i$=Reproduce($P_i$)
    i=i+1
end
```

Figure 47: Simple Genetic Algorithm

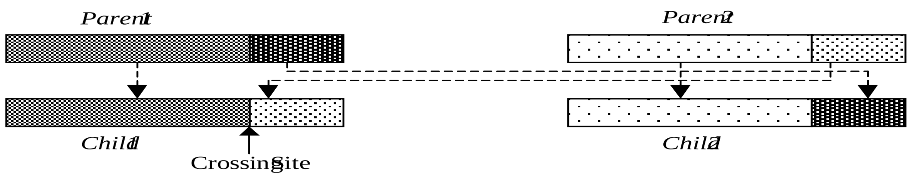


Figure 48: Schematic of simple crossover

Each individual is made up of a set of chromosomes, which represent the variables. In our work, each individual is one configuration of padding (identified by all the inter- and intra-variable padding factors), and the chromosomes represent one single padding factor. The fitness of those individuals is computed using the objective function (eq. 42). The fittest individual is the one that has a set of padding factors that results in less misses.

A chromosome representation is needed to represent each individual in the population. Genetic algorithms require the natural parameter set of the optimization problem to be coded as a finite-length string over some finite alphabet such as alphabet {0,1}. Thus, each chromosome is made up of a sequence of genes from a certain alphabet.

It has been shown that using large alphabets gives better results [53]. We have used the alphabet $\{0, \ldots, 2^k - 1\}$, where k is the greatest divisor of the $log_2 C_s$ that is lower than $log_2 C_s$. This is the largest value of k that guarantees that a single padding factor consists of at least two genes for every cache size. This is not a restriction because the compilers know the cache size. Thus, this computation can be done automatically.

Example. Let us assume a cache of 32 Kbytes. Thus, $log_2(32 \times 2^{10}) = 15$. The set of divisors is $divisors = \{1, 3, 5, 15\}$. Hence, the greatest divisor less than 15 is 5, and we will use the alphabet $\{0, \ldots, 31\}$, representing each single padding with 3 genes. For instance, a padding factor of 10017 is represented by the following three genes:

$$\underbrace{\underbrace{01001}_{gene_0=9}\ \underbrace{11001}_{gene_1=25}\ \underbrace{00001}_{gene_2=1}}_{chromosome}$$

Genetic operators provide the basic search mechanism of the GAs, creating new solutions based on the solutions that exist. The *selection* of individuals to produce successive generations plays an extremely important role. A common selection approach assigns probability of selection to each individual depending on its fitness. Individuals with higher fitness have a higher probability of contributing one or more offsprings to the next generation. Then, individuals are selected depending on this probability. We have adopted one of the selection schemes that gives better results, which is known as *remainder stochastic selection without replacement* [35].

Crossover takes two individuals and produces two new individuals with a given probability, merging the genetic material in a random point (named cross site). In the case they do not crossover, both individuals are added to the new population (see Figure 48). *Mutation* changes one individual to produce a new one by flipping some of its genes. Both crossover probability and mutation probability have to be determined empirically, and are related to the size of the population.

The GA must be provided with an initial population (see Figure 47), that is created randomly. GAs moves from generation to generation, and the usual termination criterion is the number of generations, although other criteria can be used [35].

Our experiments have shown that an initial population of size equal to 30, with a crossover probability of 0.9 and a mutation probability of 0.001, gives near-optimal results after 15 generations.

38.6 Objective Function

If the objective function to optimize is hard to compute, it will be necessary to choose a trade-off between accuracy and time-consumption.

Given a loop nest, our objective function (f in eq. 42) consists of the CMEs generated in a parameterized way, where the parameters are the padding factors.

A genetic algorithm requires to evaluate the objective function multiple times. Since counting the number of solutions to the CMEs is an extremely high time-consuming process, we use the techniques outlined in M3D1 deliverable in order to make this approach feasible.

39 Padding Performance Evaluation

This section evaluates the proposed padding approach. Examples of its use will be given, as well as its accuracy.

39.1 Experimental Framework

CMEs have been implemented for fortran codes through the Polaris Compiler [65] and the Ictineo library [5]. These libraries allow us to obtain all the compile-time information needed to generate the equations.

The evaluation of CMEs has been implemented in C++ following the techniques outlined in M3D1 deliverable and using our own polyhedra representation [9].

Due to CMEs restrictions, only perfectly nested loops in which the array subscript expressions are affine functions of the induction variables are analyzed [31]. The loop nests considered are obtained from the SPECfp95, choosing for each program the most time-consuming loop nests that in total represent between the 60-70% of the whole execution time, using the reference input data. Also some kernels from algebraic codes have been analyzed. Results for different cache architectures are reported, including the cache architecture of the new Pentium 4 processor [13]. Since the main objective of padding is to remove conflict misses, a fully-associative cache has been evaluated as a reference point. This evaluation has been done by means of the CMEs [77].

The difference between the miss ratio of a given cache and that of a fully-associative cache is an estimation of the amount of conflict misses that are not removed by the padding technique. We

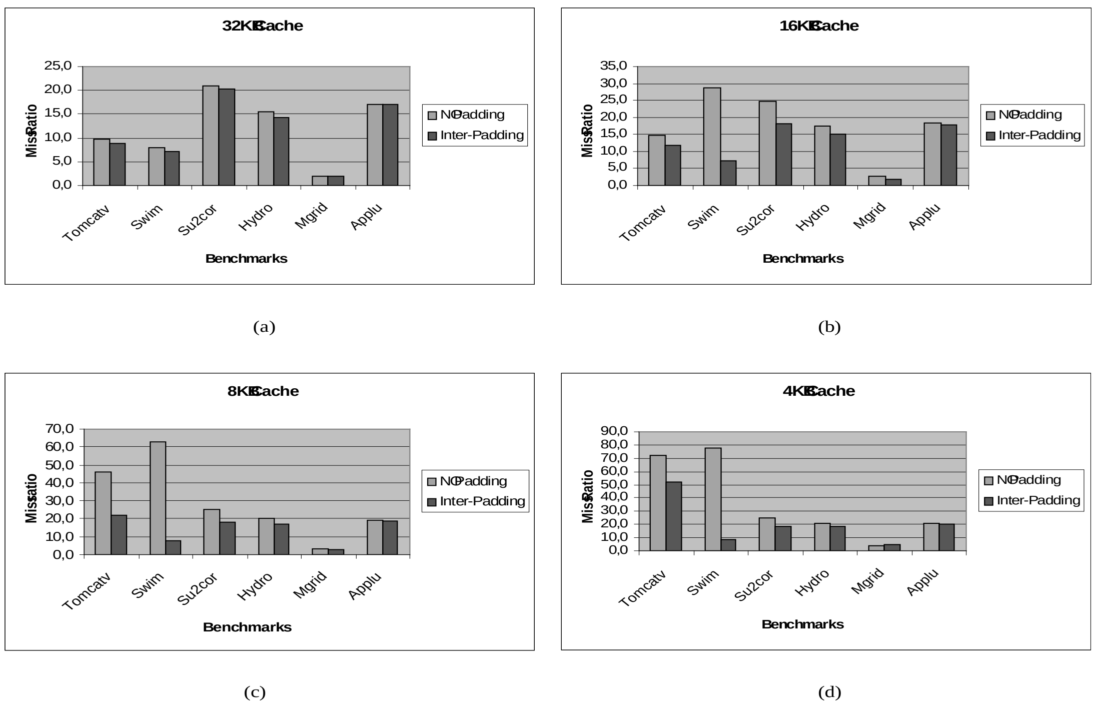


Figure 49: Miss ratio before and after inter-variable padding for different cache sizes.

assume an LRU replacement policy, which is very used in practice, unlike the Belady algorithm [8], which is optimal but only off-line implementations are known. In this case, one may find some pathological cases where the miss ratio of a fully-associative cache is higher than that of a direct-mapped cache. For instance, assume a loop nest that traverses multiple times an array whose size is equal to the cache size plus one line and the line size is equal to one array element. The LRU fully-associative cache will miss for every access whereas a direct-mapped cache will hit for all the accesses excepting those corresponding to the first and the last line.

Four SPECfp95 programs have not been evaluated because the following reasons:

- **125.turb3d and 141.apsi:** The loop nests that represent the 65% of the execution time have not enough iteration points to use sampling, although our method could be applied to these loops by analyzing the whole iteration space, we preferred to analyze some kernels instead.
- **145.fpppp:** The section of the code that represents the main part of the execution time does not contain perfectly nested loops.
- **146.wave5:** The array subscripts are function of other arrays, and thus the CMEs cannot be obtained.

39.2 Examples of Some Kernels

We will first illustrate the effectiveness of the padding approach by means of some Livermore kernels (see Table 5). We have evaluated their miss ratio for a 16-Kbyte cache with 32-byte lines.

Kernel	Description	Problem size
ADI32 (LIV 8)	2D ADI integration	N=10240
DOT256 (LIV 3)	Vector dot product	N=10240
MULT300 (LIV 21)	Matrix multiplication	N=1024

Table 5: Kernels

Program	NO Padding	Inter-Padding	Intra-Padding	Fully-Assoc
ADI32	75%	11.6%	11.6%	11.6%
DOT256	7%	5.2%	5.2%	5.2%
MULT300	21.3%	5.5%	3.2%	3.2%

Table 6: Miss ratio for the evaluated kernels (16KB direct-mapped cache, 32B lines)

Table 6 shows the results. Column 2 shows the miss ratio for a direct-mapped cache before padding, whereas column 3 shows the miss ratio after applying inter-variable padding and column 4 the miss ratio after intra-variable padding. We can see that padding significantly reduces the miss ratio for all of them. Column 5 shows the miss ratio of the fully-associative cache. We can see that after inter-variable padding, the miss ratio of the direct-mapped cache and the fully-associative cache are the same for two kernels and very close for the third one.

This third one has intra-variable conflict misses, so intra-variable padding is required to further reduce the number of misses. We can see that after applying intra-variable padding, the number of misses for the first two kernels does not change, but the third one has now the same miss ratio as the fully-associative cache. This indicates that the proposed padding technique has removed all conflict misses for the three kernels.

39.3 Performance Evaluation for the Specfp95

Figure 49 shows, for the 6 Spec95 programs analyzed, the miss ratio of a direct-mapped cache before and after applying inter-variable padding. Note that the figures for the different cache sizes (32KB, 16KB, 8KB, and 4KB) have different scales. Note also that the Spec95 applications have a relatively small working set with respect to current applications. Thus, the results for the smaller cache sizes may be more representative of what we can expect today for larger caches and bigger applications. Two sets of programs can be distinguished:

- **Set1** is composed of programs Tomcatv and Swim. The miss ratio of this set of programs is highly affected by cache size. In addition many of the misses are due to conflicts [29].
- **Set2** is composed of programs Su2cor, Hydro, Mgrid, and Applu. The miss ratio of this set of programs is quite insensitive to the cache size. In addition all the programs of this set have practically no conflict misses [29].

39.3.1 Inter-Variable Padding

Since the objective of padding is to eliminate conflict misses, for **Set2** we obtain a small improvement when applying inter-variable padding due to the low number of conflicts. Su2cor, which is the program with the highest conflict miss ratio in this set, experiences the highest improvement (e.g 27% miss rate reduction for a 16KB cache). In addition, another source of improvement is

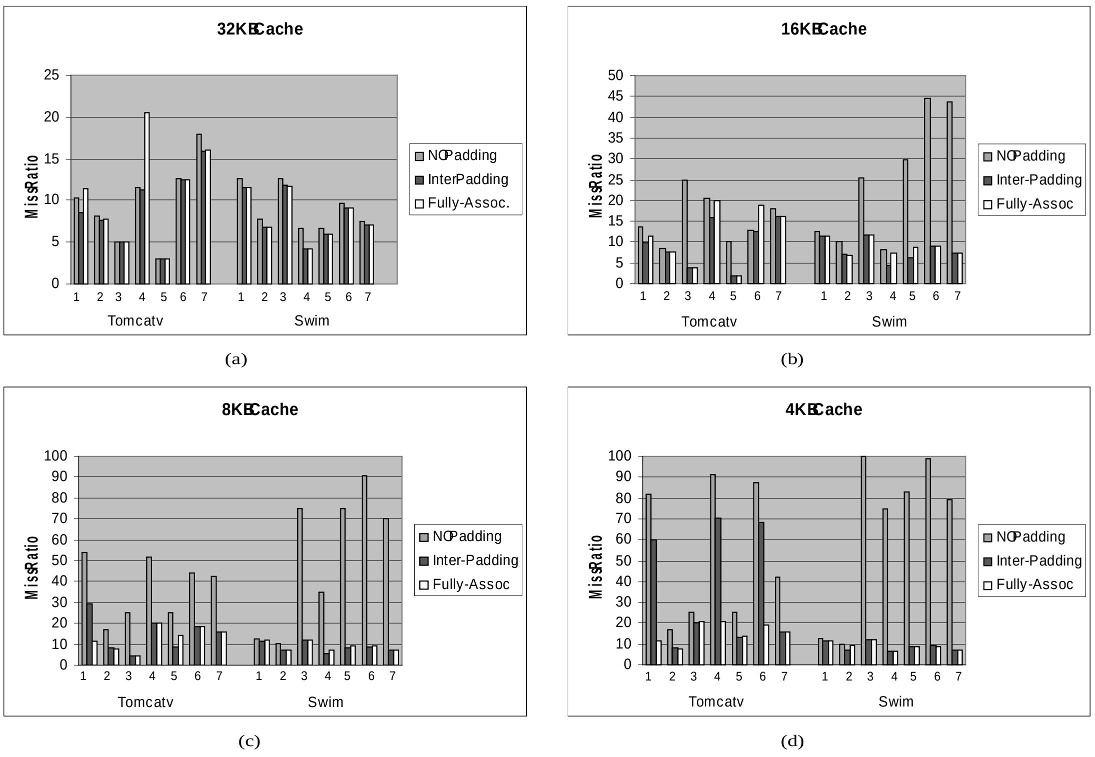


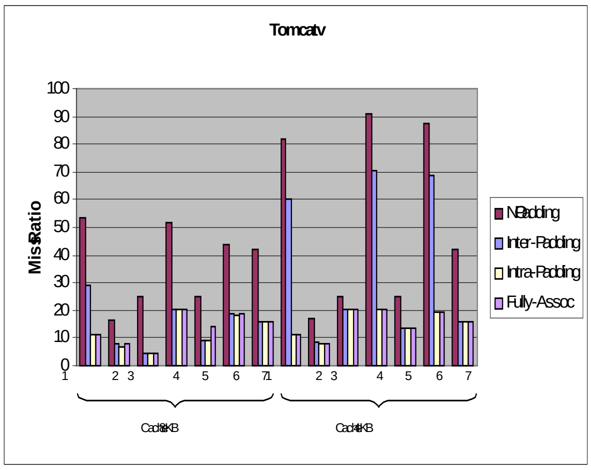
Figure 50: Miss ratio for the Tomcatv and Swim loop nests before and after inter-variable padding for different cache sizes.

that the proposed inter-variable padding technique also aligns the data structures with cache lines, which reduces compulsory misses.

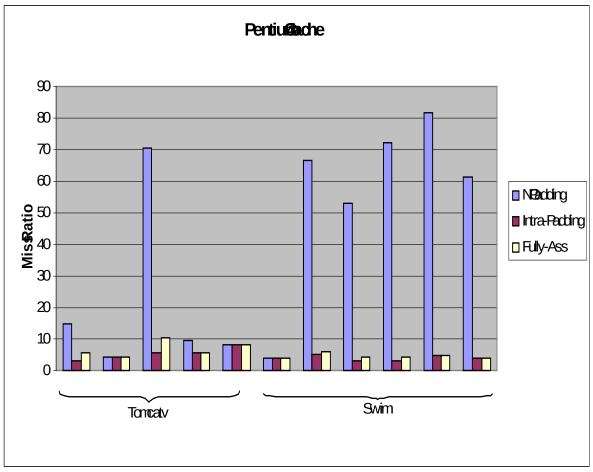
On the other hand, inter-variable padding provides a huge improvement in miss ratio for **Set1**. Note that for both programs, a small improvement is obtained for a 32KB cache (Figure 49.a). This is caused by the fact that almost no conflicts arise for 32KB caches or bigger for these programs due to the relatively small working set of the Spec95 applications. However, the smaller the cache the bigger the miss ratio and the bigger the improvement that inter-variable padding obtains.

For the Swim program, the miss ratio grows from 8.1% to 24.8%, 62.9%, and 77.9% when the cache is reduced from 32KB to 16KB (Figure 49.b), 8KB (Figure 49.c), and 4KB (Figure 49.d) respectively. However, when we apply inter-variable padding, the miss ratio is kept almost constant (7.1%, 7.2%, 7.8% and 8.2% respectively). This is because most of the misses of this program are caused by conflicts between different data structures (inter-variable conflict misses) and the algorithm practically obtains the optimal padding among them.

For the Tomcatv program, the miss ratio also grows significantly when the cache size is reduced (9.5%, 14.8%, 46.0%, and 72.1% respectively for the different cache sizes). In this program, we also obtain a considerable improvement when applying inter-variable padding for caches smaller than 32KB. However, the miss ratio after inter-variable padding varies significantly with the cache size (8.8%, 11.8%, 21.6%, and 52%). This variation is caused by capacity misses that grow when the cache is reduced, and by intra-variable conflict misses (e.g. conflicts among distinct rows and columns of the same array) whose frequency also grows when the cache is reduced. Inter-variable



(a)



(b)

Figure 51: (a) Miss ratio for different Tomcatv loop nests before and after inter- and intra-variable padding (b) Miss ratio for the Tomcatv and Swim loop nests for the Pentium 4 L1 cache

padding does not remove the latter type of conflicts, which are the target of intra-variable padding.

Figure 50 details the miss ratio for the main loop nests of the programs in **Set1** (note again the different scales for the different cache sizes). The figure shows the miss ratio for each loop before and after applying inter-variable padding. It also shows the miss ratio for a fully-associative cache after inter-variable padding.

For the Swim program loop nests 1 and 2 have practically no improvement due to inter-variable padding (excepting a slight improvement due to alignment) because they have no conflict misses. Note also that these two loop nests have almost the same miss rate regardless of the cache size. On the other hand, loop nests 3 to 7 have an extremely large miss ratio due to conflicts for small caches. Note that in these loops, applying inter-variable padding practically removes all the conflict misses since the miss rate after inter-variable padding and the fully-associative miss rate are practically identical. As an extreme case, loop nest 3 has a miss ratio close to 100% for a 4KB cache, which after inter-variable padding is reduced to 11.8%. This is exactly the same miss ratio obtained for a fully-associative cache, which means that all the conflict misses have been removed. Note that inter-variable padding removes all the conflict misses for all Swim loops since the miss rate after inter-variable padding and the fully-associative miss rate are practically identical.

The Tomcatv program has several loop nests that deserve special comments. For the 32KB and 16KB, the proposed inter-variable padding technique practically removes all conflict misses, so we will emphasize the 8KB and 4KB caches. For the 8KB cache, inter-variable padding removes all conflict misses from all loop nests except for loop 1. In this case, inter-variable padding reduces the miss ratio from 53.6% to 29.2% but not all conflict misses are removed since the fully-associative miss ratio is 11.4%. An analysis of this loop shows that in addition to inter-conflict misses there are also intra-conflict misses and this type of misses can only be removed by applying intra-variable padding.

In the case of a 4KB cache, inter-variable padding achieves about the same miss rate as a fully-associative cache for loop nests 2, 3, 5, and 7. As a noticeable case, the miss ratio of loop 7 has been reduced from 42.3% to 15.8%. For the other loop nests there is a significant improvement (from 81.9% to 60.0% for loop 1, from 91.1% to 70.3% for loop 4, and from 87.7% to 68.5% for loop 6) but the miss ratio is still far from that of the fully-associative cache (11.4%, 20.6%, and 19.2% respectively). An analysis of this three loop nests revealed that most of the remaining misses are intra-variable conflict misses so a further intra-variable padding optimization is required.

39.3.2 Intra-Variable Padding

Inter-variable padding cannot remove intra-variable conflict misses, that is, conflicts among different accesses to the same array. The objective of intra-variable padding is to eliminate these conflict misses.

We have shown in the previous section that Tomcatv is the only program of our benchmarks that has a significant intra-variable conflict miss ratio, in particular for caches of 4 and 8 KB. Figure 51.a shows the miss ratio for the different loop nests of the Tomcatv program. The figure shows the miss ratio for each loop after applying inter- and intra-variable padding. It also shows the miss ratio before padding and that of a fully-associative. As we observed before, inter-variable padding does not remove all conflicts misses because there are intra-conflict misses. Intra-variable padding achieves about the same miss rate as the fully-associative cache, which means that the proposed padding algorithm removes practically all conflict misses.

Figure 51.b details the miss ratio for the main loop nests of the programs in **Set1** for a 8KB 4-way set associative cache with 64B lines, which is the cache arquitecture of the new Pentium 4 processor [13]. Intra-variable padding achieves about the same miss ratio as the fully associative cache, reducing the average miss ratio from 62.5% to 4.18% for the Swim program, and from 23.6% to 4.6% for the Tomcatv.

39.3.3 Execution Time

Finally, padding has to be performed in a reasonable amount of time in order to be included as an optimization step of a compiler. In our case, it took about 5 minutes to optimize each program[9]. This amount of time can be significantly reduced if the technique is guided by a locality analysis in order to apply padding only to those loop nests that can benefit from it. The locality analysis developed in this work could easily be extended to provide such information.

40 Loop Tiling

In this section we present our proposal to perform loop tiling. Tiling [72, 25] is a transformation which combines strip-mining [25] with loop interchange to form small [25] tiles of loop iterations which are executed together to exploit data locality. In this section T_i and U_i stand for the tile size and upper bound respectively of loop i in the original loop nest. Figure 52 shows an example of a normalized [69] loop before (a) and after (b) loop tiling.

40.1 Model

The target of our work is to obtain the values of the variables T_i that minimize the number of misses. Note that loop tiling changes only the order in which the original iteration space is traversed, so the number of compulsory misses before and after tiling remains constant. Because of this, our work focuses in minimizing the number of replacement misses.

[9]In an Alpha Server with a 21264 processor at 524 MHZ

(a)	**(b)**

```
do $i_1$=1, $U_1$
   do $i_2$=1, $U_2$
      do $i_3$=1, $U_3$
         A($i_3$,$i_2$,$i_1$) = B($i_1$,$i_3$,$i_2$)
      enddo
   enddo
enddo
```
	```
	do $ii_1$=1, $U_1$, $T_1$
	   do $ii_2$=1, $U_2$, $T_2$
	      do $ii_3$=1, $U_3$, $T_3$
	         do $i_1$=$ii_1$, min($ii_1$+$T_1$-1, $U_1$)
	            do $i_2$=$ii_2$, min($ii_2$+$T_2$-1, $U_2$)
	               do $i_3$=$ii_3$, min($ii_3$+$T_3$-1, $U_3$)
	                  A($i_3$,$i_2$,$i_1$) = B($i_1$,$i_3$,$i_2$)
	               enddo
	            enddo
	         enddo
	      enddo
	   enddo
	enddo
	```

Figure 52: 3D matrix transposition: (a) before tiling, (b) after tiling

Let f be the function that represents the number of replacement misses for each possible value of the tile size variables:

$$f \quad \longmapsto \quad \#Replacement\ Misses \tag{43}$$

$$f : \underbrace{[1, U_1]}_{T_1} \times \ldots \times \underbrace{[1, U_k]}_{T_k} \longrightarrow \mathbb{Z}$$

Our problem can be expressed as follows:

$$MIN \quad f(T_1, \ldots, T_k)$$

$$1 \leq T_i \leq U_i$$

$$i = 1 \ldots k$$

where f is called the *objective function.*

Since f is a pseudo-polynomial function [20], the relationship between loop tiling and the number of misses is nonlinear. T_i can take only integer values, thus, our problem can be seen as a nonlinear integer optimization (NLP) one (see section 38.3).

40.2 Genetic Algorithm Parameters

The representation used to perform loop tiling has some significant differences with the representation used to perform padding. In the case of inter-variable padding, we have that the function to transform representation values to padding factors is the identity function because the padding factor is exactly the integer value of the binary representation of the corresponding chromosome. But in the case of loop tiling we need a representation for the values [1 ... U_i] where U_i can take any integer value greater than zero. The best way we have found to represent this range of values is to use a binary representation of the values [0 ... 2^k-1] where k is $\lceil log_2 U_i \rceil$. Thus, there are more values in the representation range than possible tile size values. Therefore, we need a function to map values [0 ... 2^k-1] onto the range [1 ... U_i].

Let g be the function that represents the tile size for each possible value of the representation:

$$g : [0 \ldots 2^k - 1] \longrightarrow \; [1 \ldots U_i]$$

where

$$k = \lceil log_2 U_i \rceil$$
$$x \in [0 \ldots 2^k - 1]$$

$$g(x) = \lfloor \frac{x * (U_i - 1)}{2^k - 1} \rfloor + 1 \tag{44}$$

Figure 53 illustrates an example of how this function works.

Sample code	**Tile sizes**
do ...	$g(0) = 1$
...	$g(1) = 1$
do i_j=1, 5	$g(2) = 2$
...	$g(3) = 2$
enddo	$g(4) = 3$
...	$g(5) = 3$
enddo	$g(6) = 4$
	$g(7) = 5$
$U_i = 5$	
$k = 3$	
g : [0 ... 7] $\longrightarrow$ [1 ... 5]	

Figure 53: Example of the correspondence between representation values and the tile sizes for a given dimension

It can be easily seen that every possible tile size $t_i \in$ [1 ... U_i] has at least one representation value x such that $g(x) = t_i$. The following proof shows that the lowest x corresponds to $t_i = 1$, the highest x corresponds to $t_i = U_i$ and for every x (except the highest) $g(x+1) \leq g(x) + 1$, so it is not possible to have $g(x) = t_i$ and $g(x+1) > t_i$+1, which would mean that t_i+1 had no representation.

Proof:

$g(0) = 1$ and $g(2^k$-1$) = U_i$ are obvious with equation 44.

We have that for all other values x between 0 and 2^k-1,

$$g(x+1) \leq g(x) + 1$$

thus, applying equation 44 we obtain the following inequality

$$\lfloor\frac{(x+1)*(U_i-1)}{2^k-1}\rfloor + 1 \leq \lfloor\frac{x*(U_i-1)}{2^k-1}\rfloor + 1 + 1$$

and after simplifying, the inequality is as follows

$$\lfloor\frac{x*(U_i-1)}{2^k-1} + \frac{U_i-1}{2^k-1}\rfloor \leq \lfloor\frac{x*(U_i-1)}{2^k-1}\rfloor + 1$$

using that U_i-1 $\leq 2^k$-1, it can be seen that the second fraction on the left of the inequality is smaller than 1, so the inequality is always true.

Our experiments show that an initial population of size equal to 30, with crossover probability of 0.9 and a mutation probability of 0.001, gives near-optimal results in most cases after 15 generations. In the rest of the cases, near-optimal results are obtained after a number of generations between 15 and 25. Figure 54 shows the algorithm to decide the number of generations required before the optimal tile search stops, where *converge()* is a function to decide when the population is homogeneous enough. In our case we consider that a population converges when the best individual has a difference of replacement misses smaller than 2% with respect to the population average of its generation. We have observed in the evaluated loops that this convergence criterion is only achieved if the population is closed to the optimal.

Finally, CME have an exponential cost with respect to the number of dimensions of the loop nest, and loop tiling is a transformation that doubles the number of dimensions of the loop nest and increases the number of equations (see M3.D1). Due to this, 450 evaluations (15 iterations $\times$ 30 individuals) for each loop are quite time consuming. In our case, it took more than 20 hours for some loops using an interval of width 0.05 and a 95% confidence for sampling. Due to this we changed these parameters to 0.1 and 90% respectively. After that, the time required for each loop is between 15 minutes and 4 hours.

41 Loop Tiling Performance Evaluation

This section evaluates the proposed loop tiling approach. Examples of its use will be given, as well as its accuracy.

41.1 Experimental Framework

The experimental framework used to evaluate this technique is the same as that used to evaluate padding (see section 39.1).

The loop nests considered are some kernels from different programs (NAS[10], BIHAR[11], LIVERMORE) and some frequently used kernels (see Table 7). These loops have been chosen because they exhibit high number of capacity misses. Results for different cache architectures are reported. This evaluation has been done by means of the CMEs.

41.2 Examples of some kernels

First of all we show the effectiveness of the loop tiling technique by means of some well-known kernels (see Table 8). We have evaluated their miss ratio for a 8KB direct-mapped cache with 32-bytes lines.

[10]Numerical Aerospace Simulation Facility (by NASA Ames Research Center)

[11]Biharmonic Partial differential equations solver in rectangular geometry

Kernel	Program	Description	Problem size
T2D		2D Matrix transposition	N=100, 200, 500, 1000, 2000
T3DJIK		3D Matrix transposition a[k,j,i] = b[j,i,k]	N=20, 50, 100, 150, 200
T3DIKJ		3D Matrix transposition a[k,j,i] = b[i,k,j]	N=20, 50, 100, 150, 200
JACOBI3D		Partial differential equations solver	N=20, 50, 100, 150, 200
MATMUL		Matrix by vector multiplication	N=100, 200, 500, 1000, 2000
MM	LIVERMORE	Matrix multiplication	N=100, 200, 500, 1000, 2000
ADI	LIVERMORE	2D ADI integration	N=100, 200, 500, 1000, 2000
ADD	NAS	Addition of update to a matrix	N=163
BTRIX	NAS	Block Tri-diagonal solver. Backward block sweep	N=30
VPENTA1	NAS	Invert 3 pentadiagonals simultaneously. Loop 1	N=128
VPENTA2	NAS	Invert 3 pentadiagonals simultaneously. Loop 2	N=128
DPSSB	BIHAR	unnormalized inverse of a forward transform of a complex periodic sequence	N=100
DPSSF	BIHAR	forward transform of a complex periodic sequence	N=100
DRADBG1	BIHAR	backward transform of a real coefficient array. Loop 1	N=100
DRADBG2	BIHAR	backward transform of a real coefficient array. Loop 2	N=100
DRADFG1	BIHAR	forward transform of a real periodic sequence. Loop 1	N=100
DRADFG2	BIHAR	forward transform of a real periodic sequence. Loop 2	N=100

Table 7: Evaluated kernels

Kernel	Problem size	No Tiling		Tiling	
		Total	Replacement	Total	Replacement
T2D	N=2000	63.3%	36.4%	27.7%	0.9%
T3DJIK	N=200	63.4%	36.7%	30.2%	3.6%
T3DIKJ	N=200	34.6%	7.0%	27.9%	0.3%
JACOBI3D	N=200	25.6%	7.2%	19.8%	1.3%

Table 8: Miss ratio for some evaluated kernels (8KB direct-mapped cache, 32B lines)

```
ALGORITHM:
finish := false
iters := 0
while (not finish)
   if (iters<15)
      iters = iters + 1
      create next generation
   else if (iters>=15 and iters<25)
      if (not converge())
         iters = iters + 1
         create next generation
      else finish := true
      endif
   else finish := true
   endif
endwhile
```

Figure 54: Genetic Algorithm used to perform loop tiling

Table 8 shows the results. Column 2 shows the problem size. Columns 3 and 4 show the total and replacement miss ratio before loop tiling respectively, whereas columns 5 and 6 show the total and replacement miss ratio after tiling respectively. We can see that after tiling the replacement miss ratio is near zero for all kernels. This indicates that loop tiling has removed almost all replacement misses. The remaining replacement misses probably are conflict misses which must be removed by means of techniques like padding.

41.3 Performance Evaluation for some kernels

In this section we present the replacement miss ratio of different direct-mapped caches (8KB, 16KB, 32KB and 64KB) before and after applying loop tiling for the kernels of the table 7.

It can be seen that for most programs near-optimal results are obtained after tiling. However, for some kernels (ADD, BTRIX, VPENTA1 and VPENTA2) the replacement miss ratio obtained after tiling is still quite high for all cache sizes. For some others (ADI_1000 and ADI_2000) we can observe the same trend for a 8KB cache. We have analyzed these cases carefully and we have observed that most of the remaining replacement misses are due to conflicts and they cannot be removed by loop tiling. For these programs we have investigated the combination of padding and tiling techniques as discused below.

Table 9 shows the replacement miss ratios obtained for those kernels where the replacement miss ratio is still high after loop tiling. Column 2 shows the original replacement miss ratio, whereas columns 3 and 4 show the replacement miss ratio after padding, and after padding and tiling respectively. The replacement miss ratios obtained for all kernels are near-optimal.

Table 10 shows the percentage of kernels (not considering those which appears in table 9), which have a replacement miss ratio lower than 1%, 2% and 5% respectively. For all these kernels and all cache sizes the replacement miss ratios are lower than 5%.

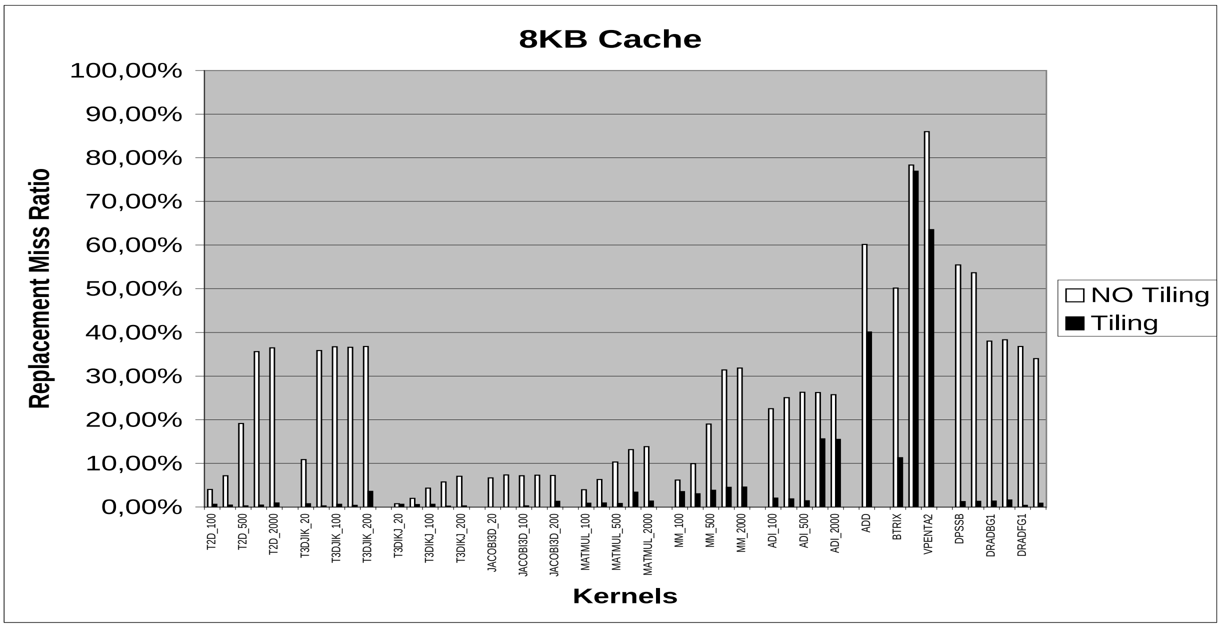


Figure 55: Replacement miss ratio before and after loop tiling for 8KB cache.

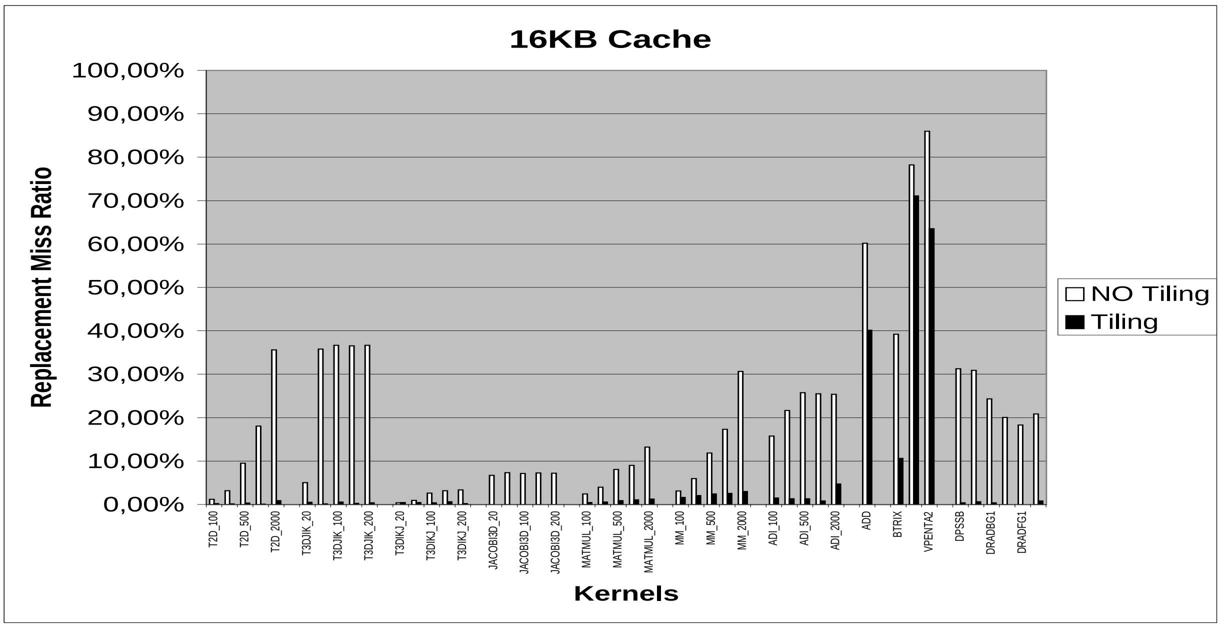


Figure 56: Replacement miss ratio before and after loop tiling for 16KB cache.

We have shown that the proposed loop tiling technique obtains near-optimal results for almost all loops. Since most of the replacement misses which our tiling technique cannot remove are conflict misses, applying padding and tiling together is a promising technique to study in future work.

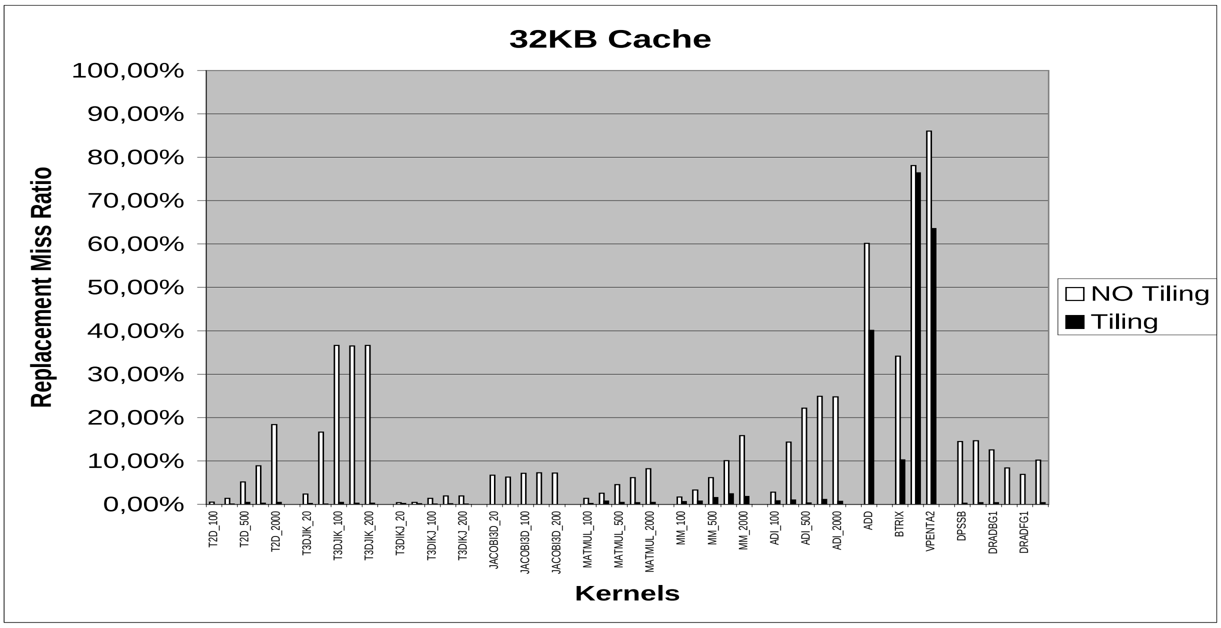


Figure 57: Replacement miss ratio before and after loop tiling for 32KB cache.

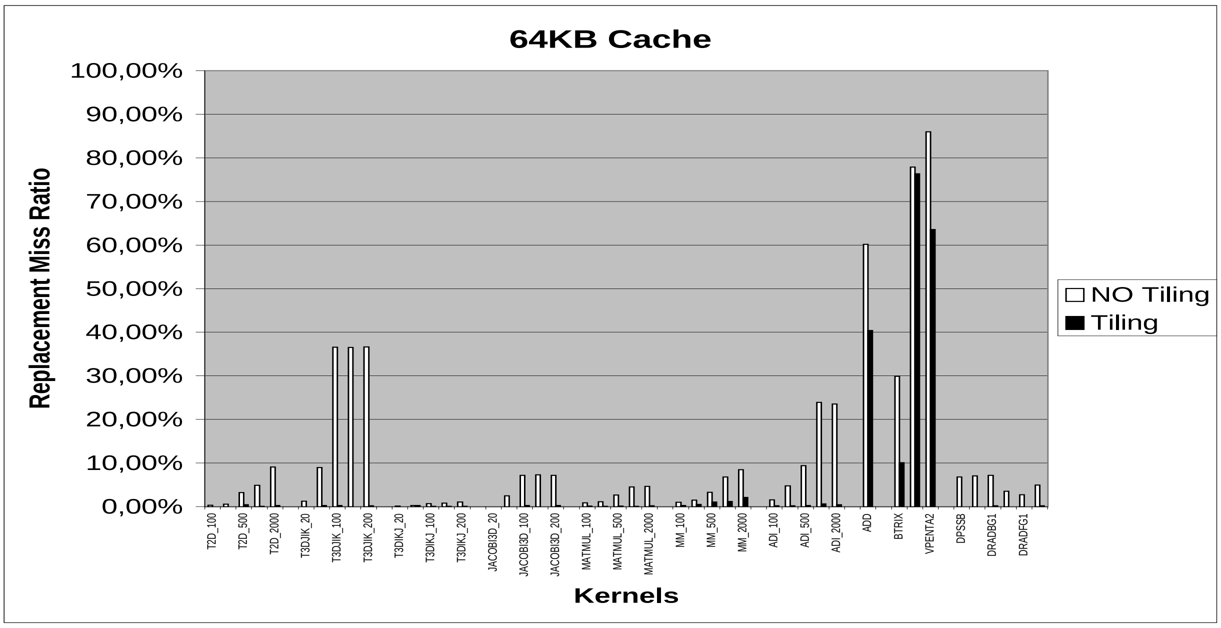


Figure 58: Replacement miss ratio before and after loop tiling for 64KB cache.

42 Conclusions

Cache memory performance is critical for the efficient execution of numerical applications. Padding and loop tiling are program transformations that reduce both types of replacement misses: conflict and capacity misses. In this work, we have proposed the use of genetic algorithms in order to

Kernel	8KB			16KB		
	Original	Padding	Padding + tiling	Original	Padding	Padding + tiling
ADD	60.2%	59.8%	0.5%	60.2%	59.8%	0.0%
BTRIX	50.1%	0.2%	0.2%	39.2%	0.0%	0.0%
VPENTA1	78.3%	52.4%	0.0%	78.2%	38.2%	0.0%
VPENTA2	86.0%	11.9%	0.0%	86.0%	11.5%	0.0%
ADI_1000	26.2%	12.3%	4.1%			
ADI_2000	25.7%	12.4%	3.4%			

Kernel	32KB			64KB		
	Original	Padding	Padding + tiling	Original	Padding	Padding + tiling
ADD	60.2%	59.8%	0.0%	60.2%	59.8%	0.0%
BTRIX	34.1%	0.0%	0.0%	29.9%	0.0%	0.0%
VPENTA1	78.1%	32.9%	0.0%	77.9%	22.6%	0.0%
VPENTA2	86.0%	11.3%	0.0%	86.0%	5.9%	0.0%

Table 9: Miss ratio for some evaluated kernels (8KB direct-mapped cache, 32B lines)

Cache sizes	Replacement miss ratio		
	<1%	<2%	<5%
8KB	56.4%	79.5%	100.0%
16KB	73.2%	87.8%	100.0%
32KB	90.2%	97.6%	100.0%
64KB	92.7%	97.6%	100.0%

Table 10: Replacement miss ratios after tiling of all kernels except those which appear in table 9

perform near-optimal padding and loop tiling.

Genetic algorithms require many evaluations of the objective function, in order to evaluate the miss ratio for different padding or tiling configurations. In order to make the technique computationally tractable and effective, a fast and accurate locality analysis technique to evaluate the miss ratio is required. We have used Cache Miss Equations for their high accuracy. However, solving CMEs directly is computationally intractable. We have proposed several techniques to solve them very fast.

The evaluations of padding and tiling show that, for the programs that have conflict and capacity misses respectively, we achieve a significant improvement. For instance, for a 8KB direct-mapped cache, we can reduce the replacement miss ratio of the 3D matrix transpostion (N=100) from 36.7% to 0.6% and the replacement miss ratio of the kernel Dpssb from 55.5% to 1.25% by means of tiling. For the same cache configuration we can reduce the miss ratio of the Swim program from 62.9% to 7.8% and the miss ratio of the Tomcatv program from 46.0% to 21.6% by means of padding. Besides, padding slightly reduces the compulsory misses due to a better alignment of arrays with cache lines.

Finally, an exhaustive evaluation of the programs and kernels with a high number of conflict and capacity misses reveals that the proposed techniques to perform loop tiling and padding practically remove all the capacity and conflict misses for all loops that have been analyzed.

Part IV
Iterative Compilation

43 Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation

Abstract

Loop tiling and unrolling are two important program transformations to increase locality and expose instruction level parallelism (ILP), respectively. However, these transformations are not independent and each can adversely affect the goal of the other. Furthermore, the best combination will vary dramatically from one processor to the next. In this paper, we therefore address the problem of how to select tile sizes and unroll factors simultaneously. Furthermore, we approach this problem in an architectural adaptive manner by means of iterative compilation, where we generate many versions of a program and decide upon the best by actually executing them and measuring their execution time. We evaluate several iterative strategies based on genetic algorithms, random sampling and simulated annealing. We compare the levels of optimisation obtained by iterative compilation to several well-known static techniques and show that we outperform each of them on a wide range of benchmarks across a variety of architectures. Finally, we show how to quantitatively trade-off the number of profiles needed and the level of optimisation that can be reached. In this way, we can reach high levels of optimisation within 50 iterations.

44 Introduction

Efficient use of the memory hierarchy is essential for good performance due to the ever increasing gap between processor and memory speed. Program transformations such as loop tiling or blocking have been shown to be an effective approach to improving locality and cache exploitation [30, 22, 37, 49, 71]. In this approach one tries to divide the loop into smaller tiles in such a way that the working set of each tile fits in the cache thereby exploiting the available locality.

It is also important to fully utilise the internal parallelism within modern processors which are capable of issuing several instructions per cycle [23]. Loop unrolling is an important transformation for this purpose [59] as it increases the size of the loop body exposing more instructions for Instruction Level Parallelism (ILP) . As we require effective utilisation of the memory hierarchy and internal parallelism, we need to combine both of these transformations. To illustrate the transformation we consider in this paper consider the example given in Figure 59. In the figure, `TJ` and `TK` are the tile sizes for `J` and `K` loop, respectively, and `U` is the unroll factor of the `I` loop. [12]

In this paper we study the combination of these two transformations. and address the problem of determining simultaneously optimal tile sizes and unroll factors for any given loop nest. Combining the best tiling transformation for locality with the best unrolling factor for ILP, however, does not give the best overall transformation as transformation application is not orthogonal in effect. Loop unrolling can adversely affect locality and tiling may restrict the available instruction level parallelism.

The close interaction between tiling and unrolling can be seen in Figure 60 which shows that a small deviation from 'good' tile sizes and unroll factors can cause a huge increase in execution time and even a slow down with respect to the original program. We need to answer the question for which tile sizes and unroll factors we obtain the minimum execution time? A static technique essentially tries to give an analytical expression for this minimum. In [12, 48] we have studied

[12]The variation on loop unrolling that we consider in this paper is unroll-and-jam [2, 15, 14] whereby an outer loop is unrolled and the inner loops are fused. Epilogue code is not shown here for simplicity

the characteristics of optimisation spaces in detail for a variety of benchmarks and platforms. and showed that different platforms give rise to widely varying optimization spaces and that a compiler, using a simplified static cost model, would never be able to predict this minimum. Instead, we propose to actually search this space in a manner that is adaptable for different architectures.. We call this approach *iterative compilation* where. we generate many versions of a program and try to determine their execution time. This can be done by actually executing the program on the target hardware, by employing one or more static models, or by a combination of both techniques. The version that performs best is selected. and thus we determine the best combined tile size and unroll factor. In the present paper, we use real execution time to search the space for a minimal point and by using a generic iterative compilation strategy, we can find excellent optimisations across a range of architectures. Thus, we obtain highly architecture specific optimisations in an architecture independent manner.

This approach is highly attractive in situations which require high performance such as embedded systems where the compilation time can be amortised across the number of products shipped or in the case of vendor supplied library codes for which the same argument holds. It is also useful in contexts where the underlying architecture changes (e.g. additional memory, a new release of the low-level compiler or a completely new processor) as the iterative search strategy has no hardwired system dependent knowledge.

Although, theoretically, iterative compilation can find the optimal version of a program for any architecture by simply considering all possibilities, in practice the search space is extremely large and therefore in this paper we examine techniques that reduce this cost and show that iterative compilation outperforms static techniques across a range of architectures.

In order to assess the efficiency of our approach, we use a collection of small benchmark kernels and three target platforms in section 45. In section 46 we show that we can find good tile sizes and unroll factors by visiting only a tiny fraction of the entire optimization space. We assess the quality of the optimization found by comparing it to two well-known static tile size selection algorithms in section 47 and show that we outperform each of them in almost all cases and can find good solutions in less than 20 minutes on average. After analysing the cost of compilation time in section 48, we limit the number of iterations to a realistic small and fixed amount and construct quantitative trade-off graphs, in section 49, that show the good performance can be reached with little cost. Finally, we discuss related work in section 50, future directions in section 51 and draw some concluding remarks in section 52.

Original

```
DO I = 1,N
 DO J = 1,N
  DO K = 1,N
   A[I,J] = A[I,J] + B[I,K]*C[K,J]
```

Transformed

```
DO JJ = 1,N,TJ
 DO KK = 1,N,TK
  DO I = 1,N,U
   DO J = JJ,MIN(JJ+TJ-1,N)
    DO K = KK,MIN(KK+TK-1,N)
     A[I,J]     = A[I,J]     + B[I,K]     * C[K,J]
     A[I+1,J]   = A[I+1,J]   + B[I+1,K]   * C[K,J]
     ...
     A[I+U-1,J] = A[I+U-1,J] + B[I+U-1,K] * C[K,J]
```

Figure 59: Example Transformation

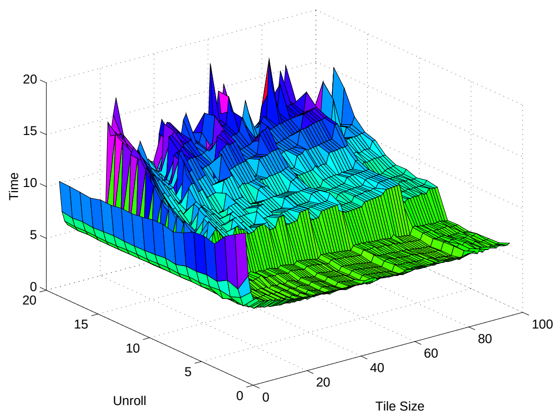


Figure 60: Execution Time MxM on Pentium for Unrolling and Tiling

45 Experiment

In this section we discuss the parameters of our experiment, including the search algorithms employed by the global driver, the benchmarks and the target platforms.

45.1 Search Algorithms

We have implemented several search algorithms, including a genetic algorithm, simulated annealing, pyramid search, window search and random search.

Genetic algorithm Genetic Algorithms are modelled on natural evolution processes and manipulate individuals in a population over several generations to improve their fitness. In GA there are two important operations: crossover and mutation.

crossover: In this phase, new children are created from two existing parents picked from the current population. Stronger parents are more likely to be selected and their children inherit their characteristics. If the crossover probability is set low, weak individuals also get a chance to be selected.

mutation: In this phase, all individuals in the population are checked bit by bit and the bit values are randomly reversed according to a specified rate, the mutation probability. This mutation operator forces the algorithm to search new areas of the search space. If the mutation probability is set high, the search sequence becomes more random.

Simulated annealing SA is modelled on the physical process of annealing which is the process of heating up a solid and then cooling it down slowly until it crystallises. The algorithm consists of a sequence of iterations starting with high temperature. A position of an atom is selected randomly and its neighbouring points are evaluated. With a certain probability, this atom makes a move. Then the temperature is reduced and the same procedure is repeated until this atom comes to a halt.

Pyramid or Grid search We define a top level grid over the search space and evaluate each point on this grid. We order the points in a priority queue. Around the best points we refine the grid.

Window search We define windows over the search space. Initially, the window is the entire space. We take s samples and order them in a priority queue. Around the best points we define a new window with length and width of p%.

Random search We randomly generate 2000 possible sets of parameters.

45.2 Benchmarks

In order to assess the efficiency of iterative compilation for selecting tile sizes and unroll factors, we want to use many small kernel benchmarks from multimedia applications that exhibit a wide variety of memory access behaviour. In this way, we are able to give a statistically relevant analysis of the results. Therefore, we chose the following benchmarks.

- Matrix-Matrix Multiplication (MxM). We use all 6 possible loop orders to generate 6 benchmarks with highly different memory access behaviour. We use data input sizes of $N = 256$, $N = 300$ and $N = 301$.
- Matrix-Vector Multiplication (MxV). We use the two possible loop orders. We use data input sizes $N = 2048$, $N = 2300$ and $N = 2301$.
- Forward Discrete Cosine Transform. This benchmark is one of the most important routines from the low-level bit stream video encoder H263. It contains three initialisation loops and two main loops: the first loop repeatedly calculates innerproducts and the other loop is a matrix-matrix multiplication. These loops are hand optimised in the reference implementation and we undid some of this optimization in order to remove a dependence that would prohibit some transformation. We use both the first main loop in isolation and the entire routine in our experiments. We use data input sizes of $N = 256$, $N = 300$ and $N = 301$ and for the purposes of this paper, restricted our attention to tile sizes from 1 to 100 and unroll factors from 1 to 20.

45.3 Platforms

We have conducted our experiments on three different platforms: Pentium II, Hewlett-Packard Precision Architecture (HP-PA 712/60) and UltraSparc. In order to compile a transformed version of the source program we used the native Fortran77 compiler with full optimization on.

Not every benchmark and data input size has been executed on every platform. Instead, we have considered a total of 82 different experiments.

46 Results

In this section we show how much speedup we obtain as a function of the number of iterations, where we show the best speedup found so far. Due to space limitation, we present only part of the speedup graphs that we have measured. For an exhaustive presentation of all these results the reader is referred to the full version of this paper [47]. We consider both rectangular and square tiles and show that square tiles achieve as much improvement as rectangular tiles in only a fraction of the compilation time. In section 47 we discuss how good these levels of optimization are by comparing them to existing static approaches.

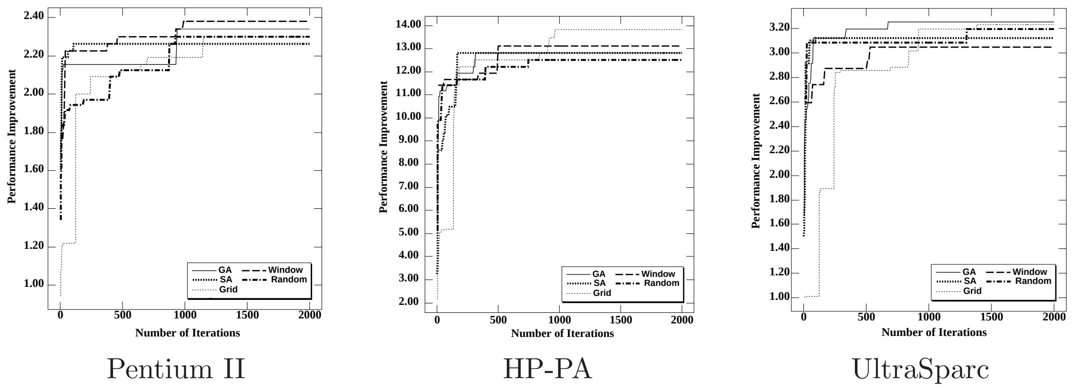


Figure 61: Speedup MxM – ijk version

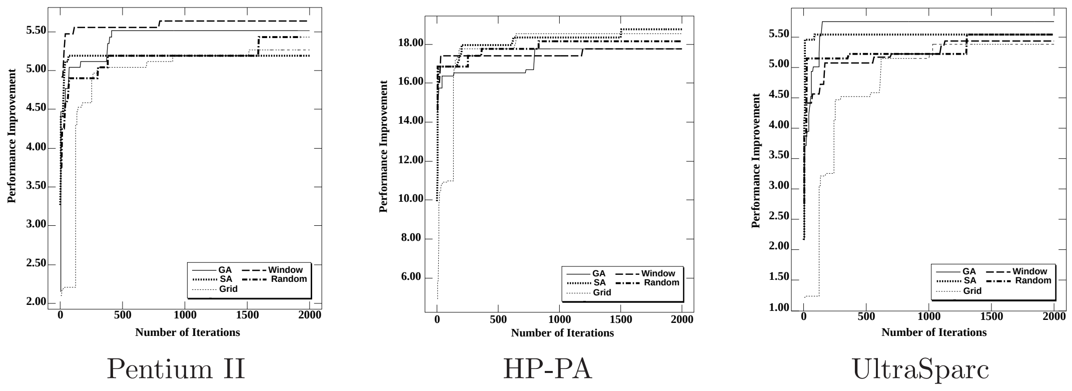


Figure 62: Speedup MxM – ikj version

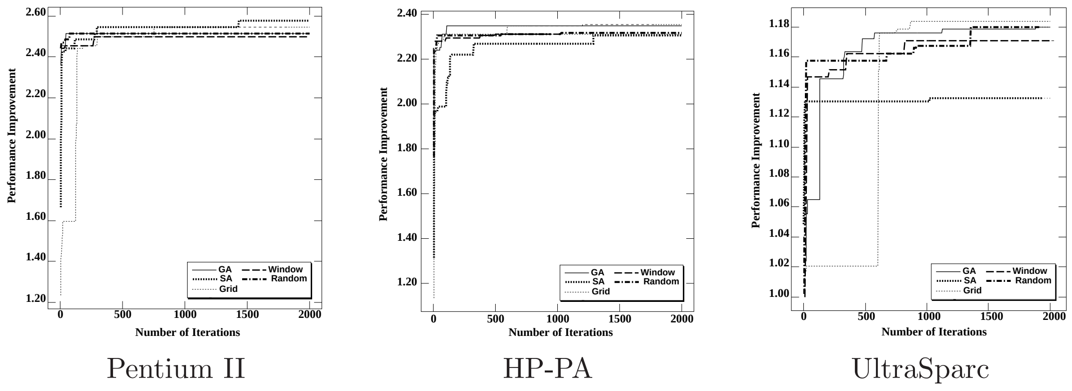


Figure 63: Speedup FDCT

46.1 Rectangular Tile Sizes

In this section we discuss the results when we search for rectangular tile sizes together with unroll factors. In this case, the search space consists of $20 \times 100 \times 100 = 200,000$ points. We let the search algorithm run for 2000 iterations. We did not consider MxV in this experiment since it only contains a double loop and rectangular tiling is not applicable.

46.1.1 Analysis of Search Algorithm Performance

The results of iterative compilation are given in Figures 61 through 63. The x-axis denotes the number of iterations, that is, the number of times a transformed version of the program is generated, compiled and executed. The y-axis denotes the speedup of the fastest version found so far.

The first observation is that iterative compilation indeed yields high levels of optimization. For example, we obtain a speedup of over 18 for MxM (IKJ version) and a speedup of 30 for MxM (KIJ version) on the HP-PA. For these benchmarks, we obtain speedups of over 7, respectively 4, on the Pentium, and 5.5, respectively 3.3, on the UltraSparc. We have shown in [47] that these high levels of optimization are found across all our benchmarks and platforms for all data input sizes. This gives some confidence in the viability of our approach. In section 47 we give a more quantified assessment of this by comparing iterative compilation to two well-known static techniques.

The second observation we can make is that these search algorithms do not differ much in their efficiency. The speedups found by the different search algorithms are within 5% on average of each other [47]. In the table below, we have given the number of iterations required to obtain maximal speedup. We also gave the number of iterations needed to reach 90% of this maximum.

	max.	90% of max.
GA	808.4 its.	76.5 its.
SA	767.6 its.	144.6 its.
Pyramid	1043.9 its.	251.7 its.
Window	835.6 its.	65.6 its.
Random	747.0 its.	162.2 its.

We observe that we on average need roughly between 750 and 1000 iterations in order to obtain the maximum speedup. We also observe, however, that iterative compilation reaches high levels of optimization much earlier and that the last few hundred iterations are used for a small increase in the final outcome. From the table we also observe that SA and Random reach their maximum fastest. It should be noted that SA shows quite random behaviour also, especially in its earlier phases where the probability to accept degradations is high. Also, Window search exhibits random behaviour because we draw random samples from the window that initially covers the entire search space. In our opinion this suggests that the underlying search space indeed is quite irregular and regular searching approaches will not yield acceptable results. Pyramid search is slowest because initially we define a grid over the entire search space that consists of 500 points. Many of these point may be located in regions that perform badly. Therefore, Pyramid search has a high initial overhead. This is also reflected in the high number of iterations needed to reach 90% of the maximal speedup. In general, we see that we need only a fraction of the number of iterations for maximal speedup to reach 90% of this maximum. Although the average number of iterations we need to reach 90% lies between 0.03% and 0.13% of the entire search space, the absolute number is far too high to settle for it. However, this observation enables us to propose a trade-off between the number of evaluations we employ and the level of optimization that we can find. This topic is discussed in more detail in section 49 where we show a quantitative trade-off graph.

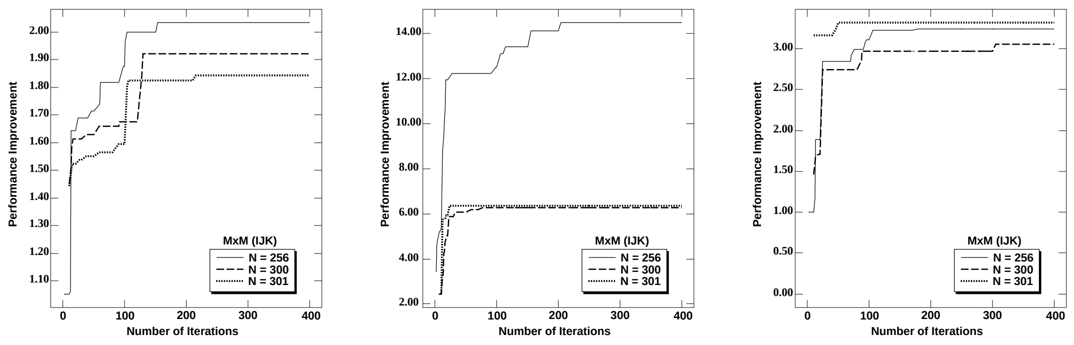


Figure 64: Speedup MxM – ijk version

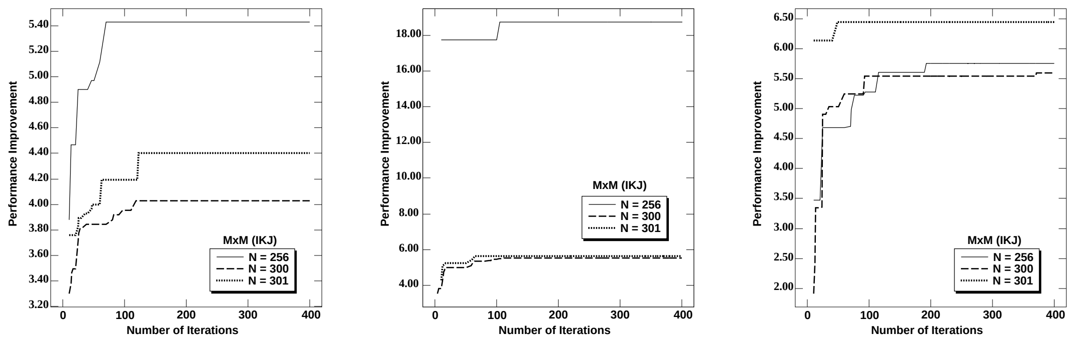


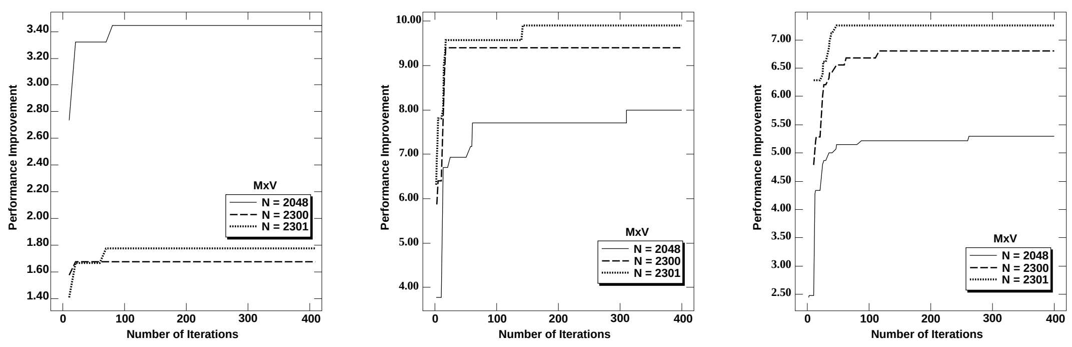
Figure 65: Speedup MxM – ikj version

46.1.2 Conclusion

We conclude that iterative compilation is capable of finding good unroll factors and tile sizes across a wide variety of benchmarks, data input sizes and platforms. Several natural algorithms all perform almost equally well. We reach high levels of optimization visiting between 0.375% and 0.5% of the entire search space. We can reach 90% of maximum optimization by visiting a far smaller fraction of the search space: between 0.03% and 0.13%. However, searching for rectangular tile sizes requires a large search space of 200,000 points. Therefore, in the next section, we consider the possibility to search for square tiles that reduces the size of the search space to 2000 points.

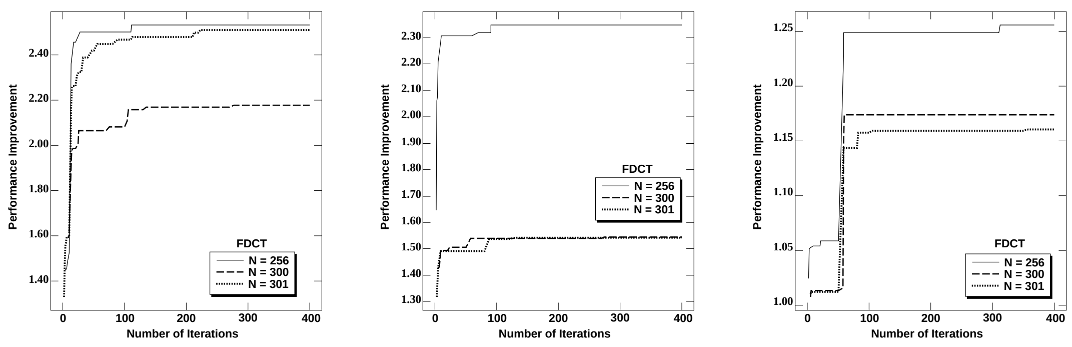
46.2 Square Tile Sizes

In this section we search for square tile sizes, reducing the size of the search space by a factor 100 to 2000 points. We implemented four search algorithms: GA, Pyramid, Random and SA. We compared the search algorithms and found that they reached their maximum improvement in about the same number of steps [47]. In this section we only present the results using Pyramid search. The results are given in Figures 64 through 67. We used 400 iterations to determine the maximal speedup, which is 20% of the number of iterations we used for rectangular tiles.



Pentium II HP-PA UltraSparc

Figure 66: Speedup MxV



Pentium II HP-PA UltraSparc

Figure 67: Speedup FDCT

46.2.1 Analysis of results

Comparing the speedups obtained using square tiles with the speedups obtained using rectangular tiles shows that they are about the same. In one case (MxM-KIJ on HP-PA) square tiling performs about 30% slower than rectangular tiling. In one other case (MxM-IKJ for $N = 300$ on Pentium) square tiling obtains a speedup of 4.03 whereas for rectangular tiling, speedups of 7.22 are obtained. However, in some other cases square tiling outperforms rectangular tiling: for the first loop in FDCT square tiling performs 45% better than rectangular tiling on the UltraSparc. In all other cases, square tiling is within 5% on average from rectangular tiling. In general, the difference between the different search algorithms for rectangular tiling is of the same order of magnitude than the difference between square and rectangular tiling. This shows that square tiles can provide the same speedup as rectangular tiles do and therefore we can restrict attention to square tiles.

The next observation is that iterative compilation reaches high levels of optimization rapidly. In the table below we have shown the average number of iterations needed to find the maximal speedup and 90% of this maximal speedup, for each platform. We see that we improve by a factor of 8 over searching for rectangular tile sizes, both for finding the maximal improvement and for finding 90% of this maximal improvement.

	max.	90% of max.
Pentium	146 its.	31.3 its.
HP-PA	79.9 its.	23.9 its.
Ultra	127 its.	38.2 its.
Average	116.2 its.	30.9 its.

46.2.2 Conclusion

We conclude that the speedup obtained using square tiles is almost as good as the one obtained using rectangular tiles. However, the time needed for iterative compilation is a factor of 8 smaller for square tiles than for rectangular tiles. This provides our first heuristic for managing the complexity of iterative compilation by only considering square tiles.

47 Comparison with Static Techniques

In this section we quantify the efficiency of iterative compilation by comparing the performance improvements to two static tile size selection algorithms. In [71] it has been shown that these tiling algorithms perform as good as or better than a host of other tiling techniques. The first algorithm, TSS by Coleman and McKinley [22], considers the size of the working set in the loop body and requires that this working set is smaller than the cache size. It also takes into account an estimate of the cross interference between different arrays and tries to minimise this cross interference. We unrolled the loop a number of times and computed the tile size using TSS for the unrolled loop. The second algorithm, LRW by Lam, Rothberg and Wolf [49], does not consider the working set nor the cross interference rate. It computes a tile size based only on the size of the cache. We have used this tile size together with different unrolling factors.

We compute the comparison between our approach and the static approaches as follows. Let S_{it} be the speedup obtained by iterative compilation and let S_{TSS} be the speedup obtained by TSS. Then the improvement of iterative compilation over TSS, I_{TSS}, is given by

$$I_{TSS} = \frac{S_{it} - S_{TSS}}{S_{TSS}} \cdot 100\%$$

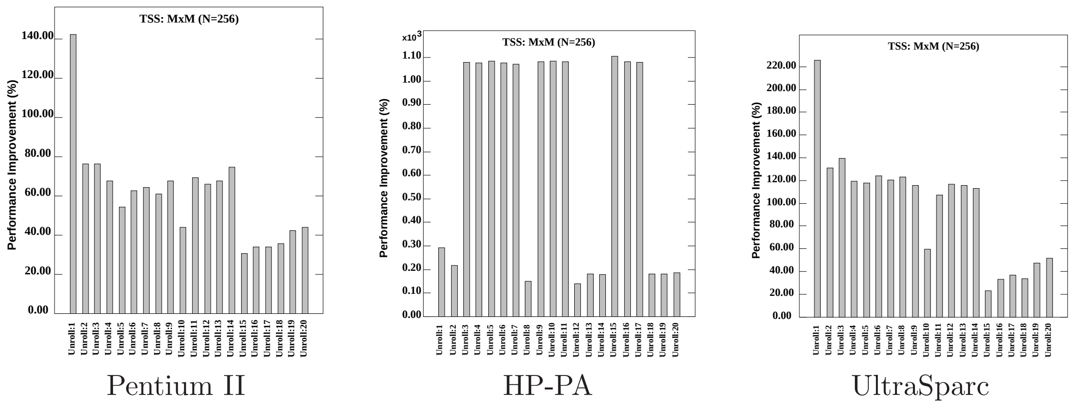


Figure 68: Improvement over TSS for MxM – ijk version, $N = 256$

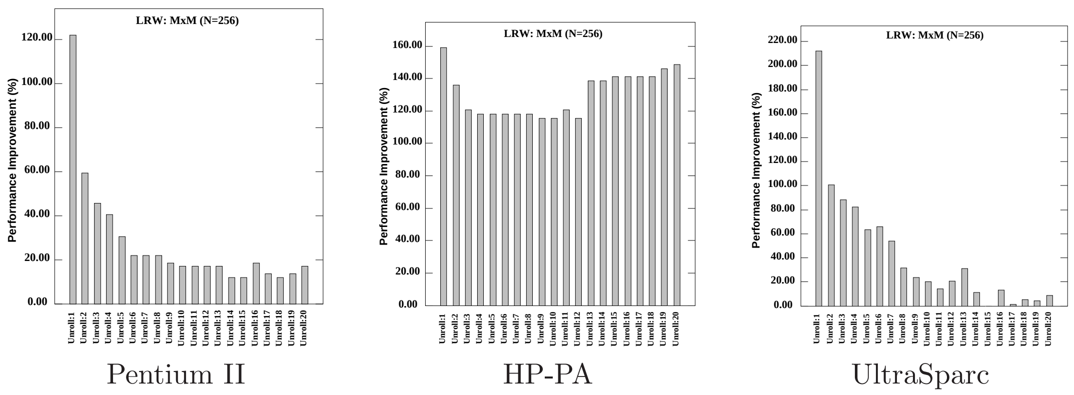
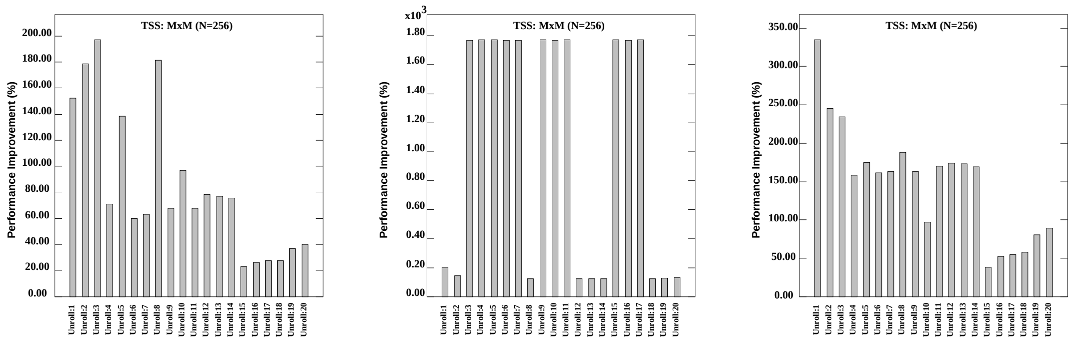


Figure 69: Improvement over LRW for MxM – ijk version, $N = 256$

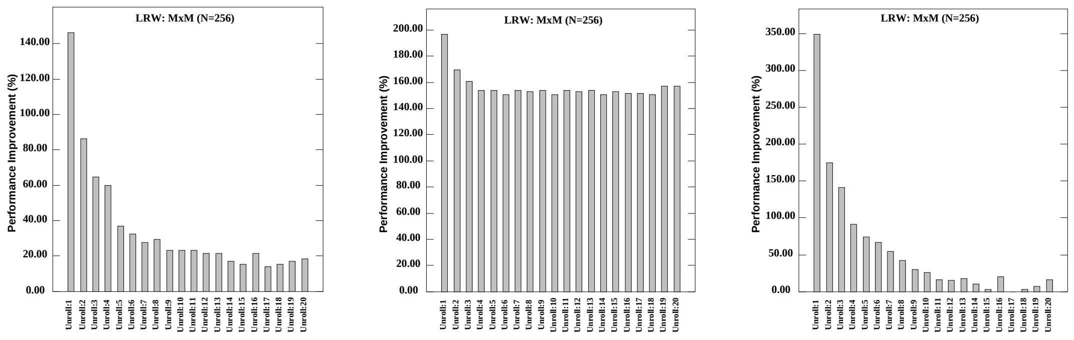
We compute the improvement of iterative compilation over LRW likewise. Note that when iterative compilation produces a lower speedup than TSS, a negative improvement is obtained. The results are given in Figures 68 through 74 (see also [47]).

We immediately observe from these figures that iterative compilation outperforms the other techniques significantly, up to 1800% for MxM-IKJ on the HP-PA. Note that we show that this improvement holds for each unroll factor less than or equal to 20 that the compiler might choose. From the figures it can also be observed that for an unroll factor of 1, which corresponds to no unrolling, improvements over tiling only are large. In the full version of the paper [47] it is shown that the observations given above hold for almost each benchmark, data size and platform. In particular, iterative compilation improves in all cases on Pentium. For HP-PA, there are 20 out of 600 cases where iterative compilation performs 2% less than TSS and 2 cases where iterative compilation performs 3% less than LRW. For UltraSparc, there is 1 case out of 600 where iterative compilation performs 2% less than TSS. It always performs better than LRW. Hence, summing up, in more than 99% of our benchmarks iterative compilation outperforms a static tile size selection algorithm. In the other cases only a very slight degradation of about 2% can be observed. From this data we conclude that iterative compilation is a powerful optimization technique outperforming existing static techniques significantly.



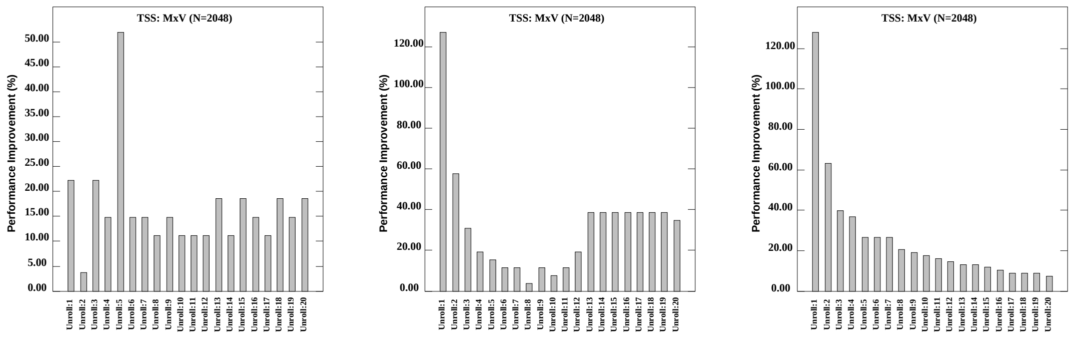
Pentium II HP-PA UltraSparc

Figure 70: Improvement over TSS for MxM – ikj version, $N = 256$



Pentium II HP-PA UltraSparc

Figure 71: Improvement over LRW for MxM – ikj version, $N = 256$



Pentium II HP-PA UltraSparc

Figure 72: Improvement over TSS for MxV, $N = 2048$

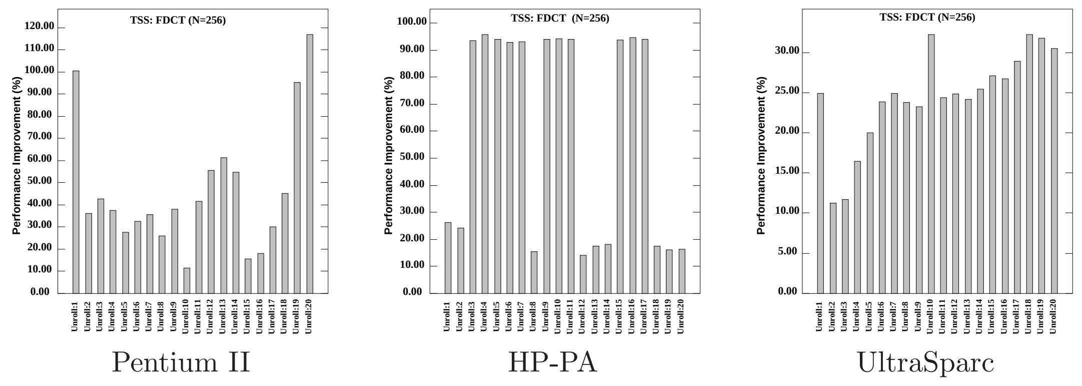


Figure 73: Improvement over TSS for FDCT, $N = 256$

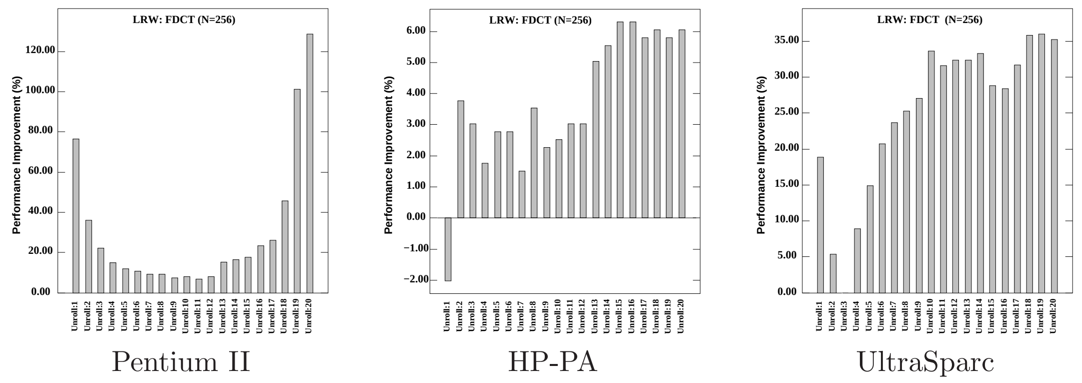
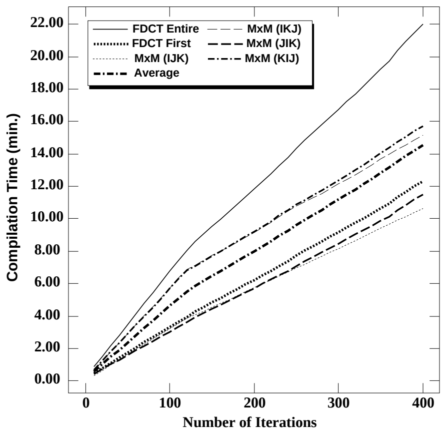
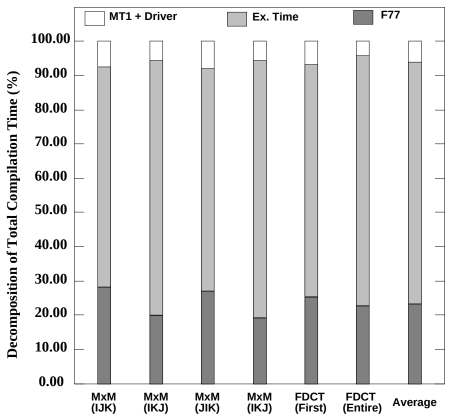


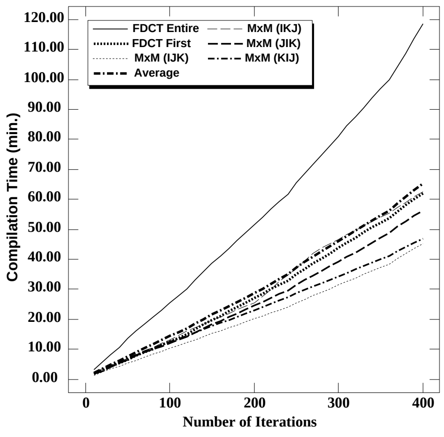
Figure 74: Improvement over LRW for FDCT, $N = 256$



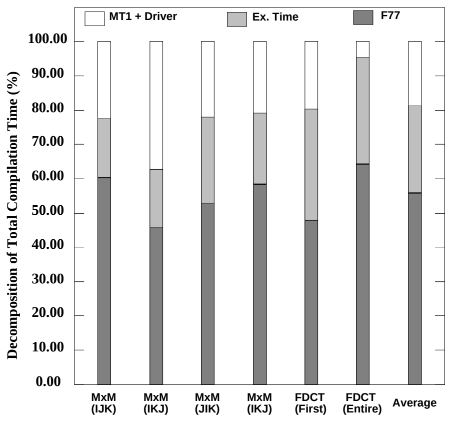
Pentium: Total time



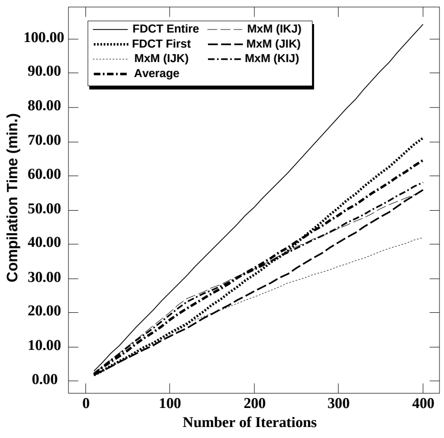
Pentium: Decomposition



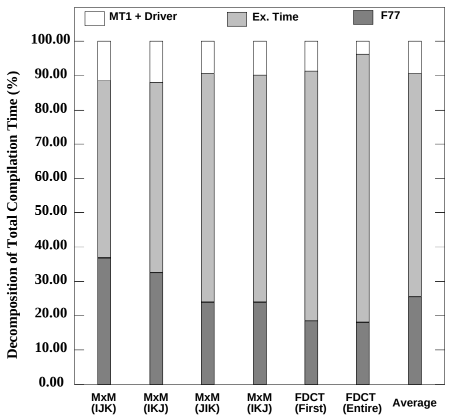
HP: Total time



HP: Decomposition



Ultra: Total time



Ultra: Decomposition

Figure 75: Compilation Time and Decomposition of Compilation Time

48 Compilation Time

In this section we discuss the compilation time required for iterative compilation that we show in Figure 75 as a function of the number of iterations, together with the breakdown in the times required for searching and program transformation in MT1, native Fortran77 compilation and execution. We observe that the relationship is almost linear. For 400 iterations, we need on average on Pentium 14.6 minutes, on HP-PA 65.23 minutes and on UltraSparc 64.49 minutes.

In case of the Pentium and UltraSparc, most time is spent in executing the transformed program. However, for HP-PA, the time required for native Fortran77 compilation is the dominant factor and the time required for the global driver can be larger than the execution time of the transformed program. Hence most reduction in time required for iterative compilation can be obtained from reducing the number of actual executions, that need both native compilation and running the program.

49 Trade-off between Number of Iterations and Level of Optimization

In this section we discuss how much improvement we can obtain from iterative compilation when limiting the number of iterations. Thus we investigate how much performance gain is available for a given compilation time "budget".

49.1 Comparing Improvements

We need to define a metric that we can use to compare improvements. In this subsection we discuss two possibilities. We use both metrics below to construct a trade-off graph between the number of iterations and the levels of optimization that result. First, we want to quantify the effect of a transformation. There are two natural approaches.

Speedup Suppose the execution time of the original program is t_0 and the execution time of the transformed program is t_1. The *speedup* of the transformation S is defined as

$$S = \frac{t_0}{t_1}$$

As we keep track of the best version found so far, the speedup is always greater than or equal to one, $S \geq 1$.

Execution time improvement This quantity is defined as the difference between the original execution time and the execution time of the transformed program, relative to the execution time of the original execution time. The improvement I of the transformation is therefore defined as

$$I = \frac{t_0 - t_1}{t_0} \cdot 100\%$$

If the transformation slows down the original program, $I < 0$. Otherwise, $0 \leq I < 100\%$. If $I = 0\%$, then the transformation has no effect. If t_1 decreases to 0 seconds, I increases to 100%. In the present case of iterative compilation, I lies between 0% and 100%.

Using the two measures above, we can define a metric to quantify how close a transformation comes to another transformation. There are two properties we require this metric to have. First, if transformation T_1 has no effect, but transformation T_2 has a positive effect, we want to be able to say that T_1 reaches 0% of the improvement of T_2. Second, if T_1 results in the same running time as T_2, we want to be able to say that T_1 reaches 100% of the improvement of T_2.

Speedup Since the minimal speedup we obtain from iterative compilation is 1, we define the metric M_S based on speedup by

$$M_S(T_1, T_2) = \frac{S_1 - 1}{S_2 - 1} \cdot 100\%$$

where S_i is the speedup obtained from transformation T_i. It is easy to see that M_S has the two properties discussed above.

Execution time improvement We want the metric M_I to measure how close the improvement I_1 comes to I_2. That is, we want the relation $I_1 = M_I(T_1, T_2) \cdot I_2$ to hold. Therefore, we define

$$M_I(T_1, T_2) = \frac{t_0 - t_1}{t_0 - t_2} \cdot 100\%$$

where t_0 is the original execution time and t_i is the execution time resulting from transformation T_i. Again it is easy to see that M_I has the two properties discussed above. We can rewrite M_I in terms of speedup as follows.

$$M_I(T_1, T_2) = \frac{S_2}{S_1} \cdot \frac{S_1 - 1}{S_2 - 1} \cdot 100\%$$

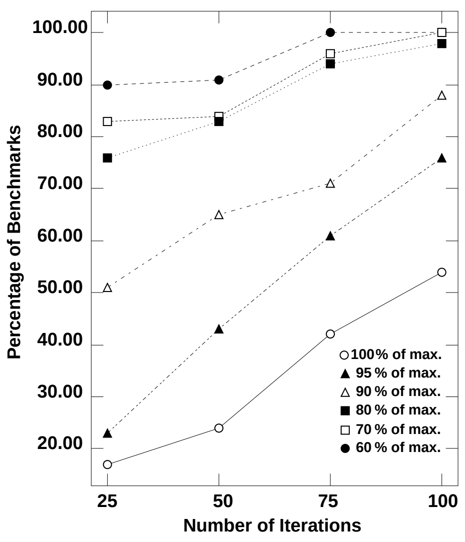
Below, T_2 is the transformation with maximal improvement, hence $S_2 \geq S_1$ and $M_I \geq M_S$.

For example, suppose we have a program that runs in 20 seconds. Suppose transformation T_1 reduces the running time to 4 seconds, and transformation T_2 reduces the running time to 2 seconds. Then $S_1 = 5$ and $S_2 = 10$, and $I_1 = 80\%$ and $I_2 = 90\%$. To compute comparisons, $M_S(T_1, T_2) = \frac{4}{9} \cdot 100\% \approx 44\%$ and $M_I(T_1, T_2) = \frac{20-4}{20-2} \cdot 100\% \approx 88\%$. Hence we see the difference between the two metrics. There is a large difference between speedups and therefore M_S is only 44%. On the other hand, there is not so much difference in running times compared to the original running time. T_1 already reduces the running time to a large extend. The metric M_I records this and says that T_1 reaches 88% of T_2.

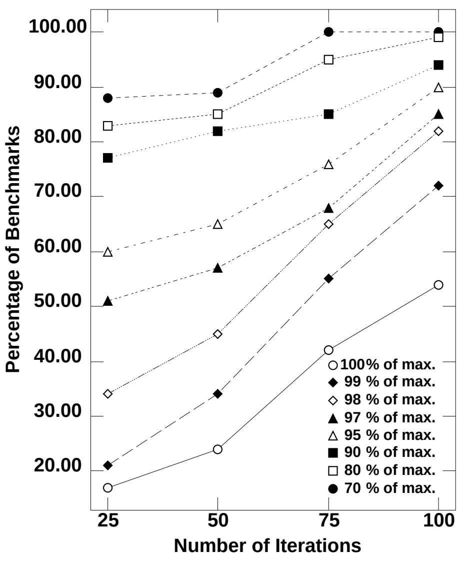
49.2 Quantitative Trade-off Graphs

We now want to compare the transformation T_i found in iteration i to the final transformation T_m with maximal effect that has been found after 400 iterations. We use the results of both the Pyramid and the Random search algorithm since this last algorithm inspects the entire search space in a few iterations and hence delivers high levels of optimization rapidly. We use both metrics discussed above in this comparison. In this way, we are able to focus on both speedup and execution time improvement. We proceed as follows.

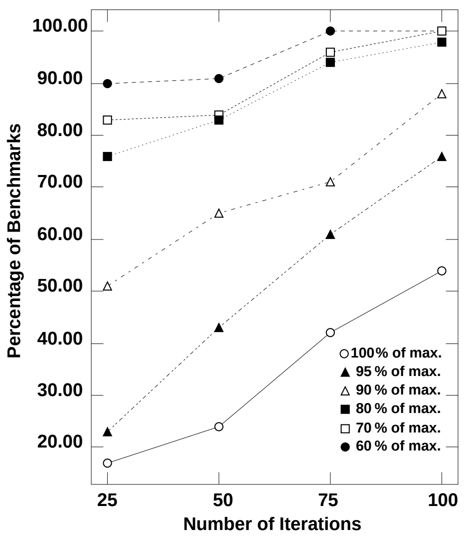
We base our comparison on the transformations using square tiles discussed in section 46.2. We measured the speedup found after 25 iterations in all 82 experiments that we have conducted. We counted the number of cases where we reached 100% of the speedup. This yields a percentage of



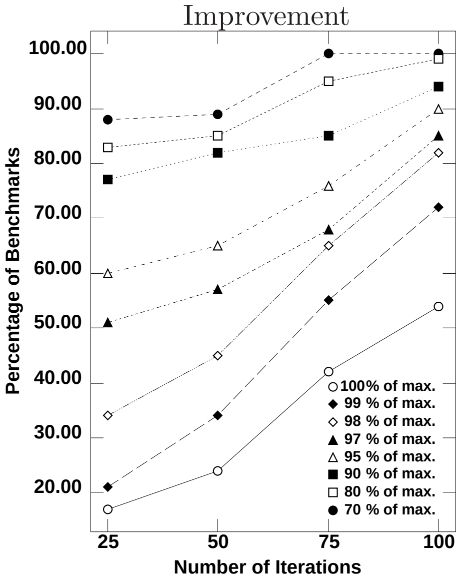
(a) Pyramid: Trade-off Speedup



(b) Pyramid: Trade-off Time Improvement



(a) Random: Trade-off Speedup



(b) Random: Trade-off Time Improvement

Figure 76: Trade-off Graphs for Pyramid search (top) and Random search (bottom)

Compilation time	25 its.	50 its.	75 its.	100 its.
Pentium	1.23 m.	2.37 m.	3.27 m.	4.63 m.
HP-PA	4.10 m.	7.68 m.	10.4 m.	14.34 m.
UltraSparc	4.66 m.	9.02 m.	12.51 m.	17.77 m.

Table 11: Compilation Time for Small Numbers of Iterations

the experiments that reach this maximal result. Likewise, we counted the number of cases where we reached at least 95%, 90%, 80% and 70% of the maximal speedup. We also counted the cases where we reached at least 100%, 99%, 98%, 97%, 95%, 90% and 80% of the maximal execution time improvement.

We followed the same procedure after 50, 75 and 100 iterations. The results are plotted in Figure 76 that provide quantitative trade-off graphs. From these figures we can deduce, for example, that using the Random algorithm after 100 iteration, 58% of the benchmarks were fully optimized and thus reached 100% of the speedup or execution time improvement. Likewise, we see that after 50 iterations, 85% of the benchmarks reached at least 90% of the maximal speedup and 79% of the benchmarks reached at least 97% of the execution time improvement. Conversely, given a budget if N iterations, we can deduce the effect of iterative compilation in terms of how likely it is that a certain level of improvement is reached.

There are a number of observations we can make. First, we see that after 25 iterations most benchmarks reach high levels of optimization: more than 50% reach 95% of the maximal speedup and 45% reach 99% of the maximal execution time improvement for Random search. Likewise, after 25 iterations more than 50% of the benchmarks reach 90% of the maximal speedup and 97% of the maximal execution time improvement for Pyramid search. For 50 iterations, all benchmarks reach at least 70% of their maximal speedup and 80% of their maximal execution time improvement for Random search, and 90% of the benchmarks reach 60% of their maximal speedup and 70% of their maximal execution time improvement for Pyramid search. Conversely, after 100 iterations most benchmarks reach high levels of optimization. For 50 or more iterations, the "equibars" for 70% and 80% of the speedup, and the "equibars" for 80% and 90% of the execution time improvement are close together. This means that there are few programs with an improvement between 70% and 80% in speedup (or between 80% and 90% in execution time improvement). The vast majority reaches a higher level of optimization.

Comparing the trade-off graphs for Pyramid search and Random search we observe that Random search reaches the highest levels of optimization quickest. One reason for this is that Random search takes samples from the entire optimization space, whereas Pyramid search evaluates a 50 point grid first. Therefore, after 25 iterations, Pyramid search only has inspected half of the search space. This shows that if we are only willing to execute a few iterations, Random search can outperform other approaches. The trade-off graph suggests that after taking 50 *random* samples, we have a high *probability* to have found a good optimization.

In Table 11, we have shown the compilation time in minutes required for 25, 50, 75 and 100 iterations. We observe that the times for 25 or 50 iterations can be afforded. In future work we hope to bring down these compilation times even further by including static models in the search.

50 Related Work

There are many paper dealing with tile size selection [22, 30, 49, 59, 71]. All these selection algorithms use static analysis and models to compute tile sizes, in contrast to the present approach that uses dynamic profiling information. Carr and Kennedy [15] and Carr [14] compute unroll-and-jam factors in order to minimise the difference in machine and loop balance. Carr computes how much benefit the unroll-and-jam of a loop has for a range of unroll factors based on static models and searches at compile time to decide which unroll factor has the most benefit. In contrast, the present approach uses actual execution times and moreover considers loop tiling at the same time.

Currently, searching techniques are employed in hardware generation, for example, in design space exploration [26]. In this approach, many implementations of a design are generated and static models are used to estimate for example die size and speed of the circuit. Optimal points in the design space are called Pareto points. For example, one such point signifies that some implementation gives the fastest circuit for a given die size, or alternatively the smallest die for a given speed.

Whaley and Dongarra [78], and Bilmes et al. [11] describe systems for generating highly optimized BLAS routines that probe the underlying hardware to find optimal transformation parameters. They show to be capable of outperforming vendor supplied, hand optimized library BLAS routines. In contrast to the present approach, however, these systems are only able to optimise BLAS routines and are not general purpose compilers.

Wolf, May Dan and Chen [79] have described a compiler that also searches for the optimal optimization by considering the entire optimization space. Han, Rivera and Tseng [37] also describe a compiler that searches for tile and pad sizes using static models. In contrast to the present approach, however, their compilers use static cost models to evaluate the different optimisations. Our approach based on actual execution times will deliver superior performance and can adapt to any architecture, requiring no prior modelling phase.

Chow and Wu [19] apply 'fractional factorial design' to decide on a number of experiments to run for selecting a collection of compiler switches. They, however, focus on on/off switches and do not consider the choice of parameter values that might come from a large range of values.

Over the past years, many proposals have been put forward to use profile information, for example, in the creation of superblocks [42] or hyperblocks [52] to enable efficient scheduling for ILP processors. These techniques are currently being employed in commercial compilers [21]. Profiles are also used to identify runtime constants that can be exploited at compile time [56]. The recently established workshop on Feedback Directed Compilation shows that currently many proposals are being put forward to exploit profile information in the compiler chain. This paper can be seen as taking profiling one step further by using many profiles for deciding between many alternatives.

The present research was started within the Esprit project OCEANS [6]. Within this project, other approaches to iterative compilation are considered. Bodin explores in [76] the interplay between loop unrolling and software pipelining. This approach can be fully integrated with the present approach since Bodin targets a different phase in the compiler, namely, the code generation phase. In [62], Nisbett proposes a genetic algorithm approach to searching.

51 Future Directions

The obvious drawback of iterative compilation is its long compilation time that is required in order to generate many versions of the source program and execute them to obtain profiling information. In this section we discuss a few possible solutions.

The first approach is to use analytical models in the search. We intend to use the static models to guide the search by using them to decide whether one version of the program is better than another. Only if the models predict that some version might be better than the best one found so far, we will actually profile that version. In this way, profiling the transformed program will only be done in situations where static information is insufficient to give reliable predictions. These models should be accurate enough to cover a large portion of the search space or, alternatively, should be accurate for certain aspects of the search space. For example, based on Figure 60 we could include a model that suggests large unroll factors and small tile sizes, and another model that suggests large tiles and small unroll factors. Iterative compilation would then search in both these areas of the optimization space. Currently, we are implementing a cache model to measure the impact of the transformations at compile time on memory access behaviour and a parameterised scheduler to measure the impact on ILP exploitation. We are also implementing Cache Miss Equations [36] and the procedure to statically evaluate the effect of Unroll-and-Jam proposed by Carr [14]. In the long term we envisage a compilation system where the user can trade-off levels of optimization and compilation time by tuning the complexity and accuracy of static models, the number of points that are inspected and the number of programs that are actually executed. In this compilation model, traditional compilers use models of low to medium complexity, visit one point and execute no programs. The present paper discusses the situation where there are no models and many points are visited and evaluated. The compilers by Wolf, Maydan and Cheng [79], and the one by Han, Rivera and Tseng [37] fall in between by using low complexity models, visit many points but do not execute any program.

In the present paper we discuss how to deal with parameters for a given transformation. However, many other transformations can be employed that do not have such a parameter, like loop interchange. In [64] we discuss an approach by considering decision trees for applying a host of transformations, including data transformations [63]. Loop unrolling and tiling is one node in this tree and the present approach to transformation space exploration can be used in this node to determine optimal parameters. However, we need to ensure that good tile sizes and unroll factors can be found very quickly for this approach to be feasible.

52 Conclusion

In this paper we have discussed how to simultaneously select tile sizes and unroll factors using a novel approach to program optimization, namely, iterative compilation. This approach generates many transformed versions of the source program and searches for the best by compiling and executing these programs. We have implemented this approach and we have shown that it is able to find high levels of optimization rapidly, outperforming existing approaches significantly. We have shown how to trade-off compilation time and levels of optimization by limiting the number of iterations. In this way, within 50 iterations, high levels of optimization on average can be found. The compilation time required is then reduced to less than 20 minutes.

Part V
Reordering Methods for Data Locality Improvement

Abstract

Cache memories have been introduced to reduce influence of the main memories slowness on processors performances. This speed gap increases almost as fast as processor frequencies. Recent systems uses several levels of cache memories. But for big data structures as in most of scientific programs, the cache mechanism isn't efficient. Many studies have attempted to modify source program in order to improve cache use. However, known techniques have many limitations. The aim of this report is to suggest a new method which exploits avaible cache management software features and which allows cache memories use on real-time systems.

53 Introduction

The performance gap between every new commodity processor generation and main memories increases constantly. Progress concerning memories is widely insufficient in front of the very fast increase of processors frequencies, and number of operations which they can handle during one clock cycle. This led in the 60s to the introduction of *caches*, intermediate memories with small sizes and closer to processors speed. Purpose was to take advantage of the speed of the fastest memory (level 1 cache) combined to the size and to the price of the slowest (the RAM), by exploiting the *principle of data locality.*

The effective use of this memory hierarchy is one of the keys to obtain the best performance. We obtain them only with a maximal use of data locality. The first attempts to solve this problem were programming crafts, leading to non portable codes, nearly illegible, and error-prone. That is why big efforts were made to find solutions allowing the compiler to do the optimization work. It consisted first of all in integrating already well known program transformation techniques, more recently in studying the data layout transformations, and now in combining these two approaches.

Techniques focusing on cache optimization suffer big limitations. It is difficult to make choices on the program tranformation to be applied, as on the order in which to apply them. Furthermore, these transformations are very sensitive to dependence problems. The data layout tranformations lack flexibility and do not allow an optimal layout for the whole program in realistic cases.

None of the current techniques takes into account the existence of tools allowing the software to participate in cache management. We propose here a new method of locality optimization, based on the use of these tools. The principle is to reorganize the operations of a program to form subsets for which all the referenced data hold in the cache. This technique, called *chunking* takes advantage of the cache memory by being freed from inherent limitations of already existing methods. Furthermore, controling the cache contents at any time allows us to offer guarantees of real-time type.

We will recall first of all what phenomenon is responsible for reduction in cache performance. For it we will define the principle of locality, and show how one take advantage of it and what are its limits. We will then quickly present the various techniques exploiting this principle to optimize programs at compile time. We will show then the limitations of these methods and present chunking, whose aim is to surmount them.

53.1 Principle of data locality

53.1.1 Definition

Memory accesses have been known to be predictable for a long time. The locality principle is a hypothesis made on program memory accesses behavior. We divide it in two suppositions putting in evidence the two dimensions of this principle [Hen90]:

temporal locality: if an item is referenced, it will tend to be referenced again soon,

spatial locality: if an item is referenced, nearby items will tend to be referenced soon.

53.1.2 Exploitation

The principle of locality is the basis of hierarchical memories systems. Their organization and their functioning are simple: the highest level of the hierarchy is the processor registers, the lower levels are slower but bigger (and less expensive) cache memories. The largest but also the slowest (and the least expensive), main memory, takes the last place. One may even consider disks as the lowest level of hierarchy. If an item is present at a level of the hierarchy, it will be present also in all lower levels.

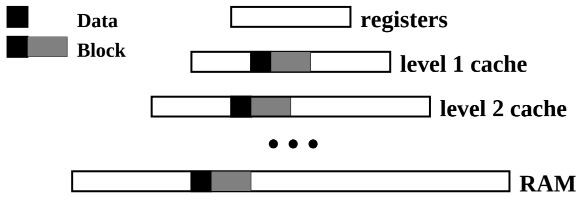


Figure 77: Memory hierarchy

When a processor needs an item, it looks for it in the memory hierarchy from the upper to the lower level. When it is found, it is copied up to the upper level (temporal locality: if the processor wants to re-reference it a short time later, it will be in the highest hierarchy level, and so referenced in the fastest way). Furthermore, some nearby data are copied at the same time (spatial locality: if the processor wants to reference one or some of these data a short time later, they will be in the top of the hierarchy, and so referenced in the fastest way). The data set simultaneously copied is called a *line* or a *block*. Supplementary data to the one that was referenced are generally its neighbors.

More precise information about the functioning of cache memories and more generally about hierarchical memories systems will be found in [Hen90, Tan90].

53.1.3 Limits

In most of cases, the functioning of cache memories effectively allows one to benefit from data locality. But in situations such as repeated treatments on big regular structures, as for matrices or vectors during scientific calculations, we can see that it is unsuitable. In these circumstances, data can be replaced in the hierarchy at each access, in spite of the fact that they are the subject to re-use during the treatment. Bad performance results from this temporal locality loss. Also, the successive use of non consecutive data in memory will lead to a spatial locality loss, and so, to a performance loss.

Here are two examples illustrating these various types of locality loss. Measures were made on an i386 machine provided with an unique 32Kb cache level, and using a 32 bytes length line (except when stated otherwise, all the quantitative data of this document will have this machine for source). The first example presented in figure 78 shows performance loss caused by temporal locality loss on a simple program. This program computes *nb_times* repeated operations on consecutive data of floating type (4 bytes) of an array v of size N. A performance loss logically occurs when the array becomes too big for the cache memory.

```
for $(i = 0; i < nb\_times; i++)$
  for $(j = 0; j < N; j++)$
    $v[j] = v[j] + constant$ ;
```

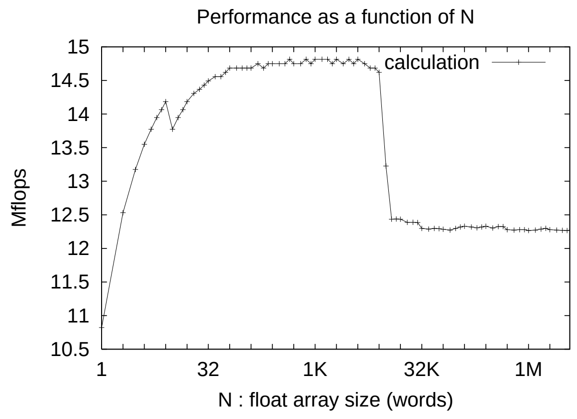


Figure 78: Temporal locality loss

The second example in figure 79 is similar to the first with the difference that calculations made in the j loop do not take place on consecutive data in memory. Program makes jumps of *jump* length in the array, instead of treating cells one after the other. This causes a spatial locality loss and so a performance deterioration which increases with the lenght of the jump. This deterioration stops when the jump length becomes equal to the block length. At this moment, every data can only be found in the main memory.

These limits are known since cache memories introduction. Since then, a lot of work was focused at improving this situation.

53.2 Classical techniques of locality improvement

We can consider two main families of techniques allowing to improve locality: *program transforations* and *data transformations*. All these techniques try to move operations referencing identical data closer. This so that data are still in the cache when we will need it again. We call it transforming re-use into locality. The first technique aims at finding a better execution order for operations by modifying control statements, the second tries to adapt the memory data layout. In the following section, we will outline these two techniques.

53.2.1 Program transformations

Program transformations were the first proposed solutions. They consist of various methods whose applications was the responsability of the programmers [All72]. The resulting code was difficult to understand and non portable. Wolf and Lam first proposed an algorithm based on program transformations that allows automatic optimizations [Wol91]. They so began the big effort of research led to create effective optimizing compilers.

The general idea of program transformation is to modify the iteration space of every loop nest program [Wol91, Wol92]. It means changing the operations order by loop reorganization, to bring

```
for $(i = 0; i < nb\_times; i++)$
    for $(j = 0; j < N; j+ = jump)$
        $v[j] = v[j] + constant$ ;
```

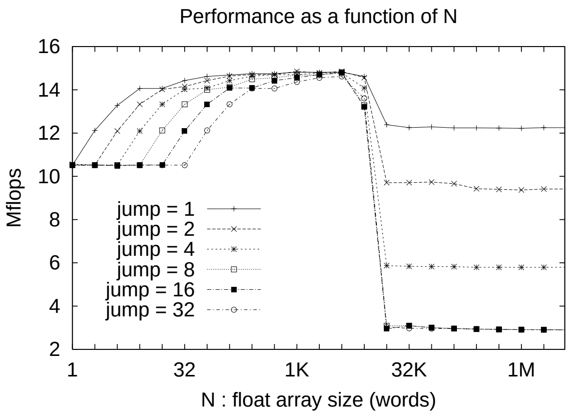


Figure 79: Temporal and spatial locality loss

together the treatments of the same data and so, to take maximum advantage of the locality. An ideal reorganization is not always possible (or *legal*): we have to make sure of the absence of dependence forbidding transformation [Gof91, Kuc81]. But this is not the real problem: analysing dependence is a well known problem in automatic parallelization [Ber66, Ban88]. The real problem, particularly for automation, will not be to know if a transformation is legal, but whether it improves locality or not.

These techniques can have effects in various levels of the hierarchy: on top with scalar replacement and unroll-and-jam (see [Cal90, Car94]), on some levels with tiling (see [WoM87, WoM89, Iri88, Lam91, Wol91]) or on all levels with loop transformations (see [McK96, War84, WoM86]).

53.2.2 Data layout transformation

To optimize locality by using data layout transformation, or more simply data tranformation, is a more recent approach than program transformations. Li put the foundations of it in his thesis [LiW93], expanding Wolf's theory [Wol91]. The basic idea comes from a simple observation: the locality improving way depends on the data layout. For example in the C language, the data lines of a matrix are ordered successively in memory (*row-major*), while this is the case of columns in FORTRAN (*column-major*). To improve locality for a certain code with program transformations would lead to a different result for each of these cases. So, instead of adapting program to a certain layout to improve locality, this approach adapts the layout to the program. Contrary to the program transformations, with which we can obtain an optimized code equivalent to source code, the data transformation take place more profoundly within the compiler and leave code intact. So, they are not limited by dependences. Every data structure of the program can have a different layout in memory.

O'Boyle and Knijnenburg showed how the data transformations improved locality where program transformations could not [OBo96]. Before them, Anderson, Amarasinghe and Lam proposed an algorithm based on this approach to allow the processors of shared memory machines to reference consecutive data [And95].

It was demonstrated that for a given loop nest, if we were able to improve spatial locality by program transformation, we will be able to make this improvement by data transformation [Kan98]. This powerful law also puts in evidence the big limitations of the data transformations. On one hand, they can improve only spatial locality, and on the other hand, we can guarantee their efficiency only for a given loop nest. Indeed, the array layout must be fixed during all the program, so you can be in the case where for a loop nest, a certain data layout is the best, and not for another loop nest of the same program. Other ways were investigated in the direction of the data transformation. One of it advocates dynamic layout change, but compromises have to be made between time lost in changing the layout and time gained by having made it. Another one suggests to calculate new array references in order that data layout corresponds exactly to the data utilisation order [Cla00], but this approach is still limited to simple cases.

53.2.3 Mixed approach

Most of current researches in the locality improvement domain tend to propose a mixed approach. It is natural considering the good complementarity of possibilities and limits of the first. One approach is limited by dependences but can be applied to fragments of program when the other one can be applied to the complete program but can ignore dependences. Cierniak and Li have been the first to use such an approach [Cie95], and were able to obtain in most of cases better performances than separate approaches. Kandemir, Choudhary and Ramanujam proposed a more complete method [Kan97, Kan99, and others . . .], but still suffering some deficiencies (for example it does not take advantage of tiling).

54 Towards a new solution: chunking

Combining the techniques of program transformations and data transformations does not remove their respective limitations. All difficulties: choice of transformations, order of their application as well as dynamic transformation of data layout, remains present, even if clevers combinations may limit their impact.

Researchers restrain themselves in only using these two techniques, only because their basis hypothesis is that cache management is left to a purely hardware mechanism. But there are already methods, although still limited, for controlling cache content (such as the primitive *Write Back Invalidate* WBINVD of the i386 instruction set). More cache control features may appear in new architectures, as IA64. Our technique for locality optimization is based on the availability of these new tools.

Besides releasing us from limits of the usual methods, the choice of a technique using cache controls is going to allow guarantees of real-time type. Indeed, the prediction of the contents of the cache and its stability at the same point in various executions become possible. This solution, not suffering inconveniences of those already existing, and in addition adapted to real-time processing is called **chunking**.

54.1 Preliminaries

54.1.1 Notations

- $\mathcal{P}$ is a program, considered as the set of its statements and its arrays;
- $A \in T_{\mathcal{P}}$: A is an array of $\mathcal{P}$. $\rho(A)$ is the rank of A, i.e. its number of subscripts, and D_A its subscript domain (in most languages, a rectangular parallelepiped);

- if $i \in D_A$, then $A[i]$ is the cell of A with subscripts i. The disjoint union of all D_A is the data set of $\mathcal{P}$, $M_\mathcal{P}$;
- $S \in I_\mathcal{P}$: S is a statement of $\mathcal{P}$. $\rho(S)$ is the rank of S, i.e. the number of surrounding loops, and D_S is its iteration domain.;
- if $x \in D_S$, the statement $S[x]$ is the instance of S with iteration vector x. The disjoint union of all D_S is the set of operations of $\mathcal{P}$, $E_\mathcal{P}$;
- a *reference* is a pair $\langle A, f\rangle$ where A is an array, and f an index function. A statement is considered as the set of its references;
- if $u \in E_\mathcal{P}$, $D(u)$ is the access domain of u, the set of memory cells that u access;
- $<_\mathcal{P}$ is the sequential execution order of $\mathcal{P}$, $<_\mathcal{P} \subseteq E_\mathcal{P} \times E_\mathcal{P}$;
- $\delta_\mathcal{P}$ is the dependence relation of $\mathcal{P}$, $\delta_\mathcal{P} \subseteq <_\mathcal{P}$.

54.1.2 Hypothesis

To allow the functioning of the chunking or temporarily to simplify the study of the problem, some hypothesis were put on the program source and on the target architecture. The source programs are supposed to have static control:

- the C **for** loop is the only control statement, with affine bounds;
- arrays are the only data structures, with affine subscripts;
- affine bounds and subscripts depends only on outer loop counters and structure (or size) parameters;
- no subroutine of function calls.

The target architecture should have:

- a main memory and only one cache level between it and the processor, the main memory being big enough to contain $M_\mathcal{P}$;
- a *fully associative* cache memory, in which a line can occupy any place [Hen90], and where the replacement policy is indifferent (the principle of the method is never to use it);
- an instruction set dedicated to the control of the cache:

Instruction	Description
LOAD CACHED	a memory line is fetched to the cache and to the processor
LOAD NOT CACHED	a memory line is fetched to the processor but not to the cache
STORE CACHED	a cache line is modified from the processor
STORE NOT CACHED	a memory line is modified directly from the processor
INVALIDATE	a line is evicted from the cache and, if modified, written back to memory
FLUSH	generalize INVALIDATE for all the contents of the cache

54.2 Principle

Chunking is a new method for reorganizing the operations of a program. The principle is to partition the set of operations $E_{\mathcal{P}}$ in smaller subsets such that the accessed data fit in the cache: **the chunks**. These subsets must be such that their execution following a definite total order is equivalent to the execution of the original program $\mathcal{P}$. The operations belonging to the same chunk are executed according to $<_{\mathcal{P}}$. A set of chunks answering these requirements is called a **chunk system for $\mathcal{P}$**.

Chunking is a reordering operation: it must be compatible with the dependences of the original program. We say that there is a dependence when references access the same memory cell and when at least one of these accesses is made in writing:

$$S[x]\,\delta_{\mathcal{P}} T[y] \equiv \forall x \in D_S \;\; y \in D_T,\; S[x] <_{\mathcal{P}} T[y] \;\;\&\;\; \exists \langle A, f\rangle \in S, \langle A, g\rangle \in T : f(x) = g(y) \qquad (45)$$

such as $\langle A, f\rangle$ or $\langle A, g\rangle$ or both are writes. Let us construct a graph Γ whose vertices are the chunks. There is an edge from c to c' iff $\exists u \in c, u' \in c' : u\,\delta_{\mathcal{P}} u'$ The chunk system has a valid execution order if Γ has no cycle. The execution order can then be found by topological ordering. In practice, chunks will be numbered and executed in order of increasing numbers. Let $\theta(u)$ be the number of the chunk operation u belongs to. The constraint is:

$$\forall S, T \in I_{\mathcal{P}},\;\; S[x]\,\delta_{\mathcal{P}} T[y] \Rightarrow \theta(S[x]) \leq \theta(T[y]).$$

We call **the footprint** of a chunk all the memory cells accessed by the operations of this chunk. The footprint of a chunk t is:

$$F(t) = \cup_{S \in P} \cup_{\langle A, f\rangle \in S} \{A[f(x)] \mid \theta(S[x]) = t\}. \qquad (46)$$

All chunks must satisfy **the capacity condition**, where C is the cache size:

$$\forall t,\; \text{Card}\, F(t) \leq C. \qquad (47)$$

Schematically, the progress of a program having been submitted to chunking would be made step by step, chunk after chunk, as if there was between each of them a kind of context switch. The new program is constructed from the source program and the chunking function θ. The first step is to reorder the program according to θ, which is a well known Z-polyhedron scanning problem when θ is affine [Anc91]. We next have to write the cache management code. This consists in writing INVALIDATE instructions for all memory cells in $F(t) - F(t+1)$. If this solution proves to be too complicated, a brute force solution is to INVALIDATE all of $F(t)$. This is summarized by the following pseudo-code:

```
FLUSH
for ($t = 0; t < L; t++$) {
    execution of chunk $t$ according to $<_{\mathcal{P}}$ ;
    INVALIDATE $F(t)$ ;
}
```

where L is the number of chunks. At the end of the new program, all the operations of the original have been executed, the only misses being compulsory misses at the beginning of each chunk.

54.3 Quantitative tools

Our present purpose is to know how to build, in a totally automatic way, a chunk system for every program $\mathcal{P}$. The construction should supply an optimal chunk system. It will be the case when every cell of $\mathcal{P}$ appears in as few footprints as possible. This condition is enough to guarantee that one has found the best possible chunk system, and to limit their number L. This section propose tools to describe a chunk system and to estimate its quality. In particular, one will find there the ways to estimate cell redundancy and footprints size, which are the main characteristics of chunk systems.

54.3.1 Computing the memory traffic

Memory traffic $\mathcal{T}$ is the number of comings and goings between main memory and cache memory. In the chunking case, we can estimate it to the double of the total size of chunk footprints. Indeed, every data will be at first read from the main memory then rewritten there after use. This value is an overestimate because only data which were modified are being updated in memory:

$$\mathcal{T} = 2\sum_t \text{Card } F(t).$$

Since we are only considering the order of magnitude of $\mathcal{T}$, we can ignore the factor of 2, and we can then define the traffic as:

$$\mathcal{T} = \sum_{a \in M_{\mathcal{P}}} \text{Card } \{t \mid a \in F(t)\}. \tag{48}$$

Proof Let $i_S(a)$ be the indicator function of a set S, i.e. the function which is equal to 1 if $a \in S$ and zero otherwise. We have:

$$\mathcal{T} = \sum_t \text{Card } F(t) = \sum_t \sum_{a \in M_{\mathcal{P}}} i_{F(t)}(a)$$

inverting the order of summation gives:

$$\mathcal{T} = \sum_{a \in M_{\mathcal{P}}} \sum_t i_{F(t)}(a)$$

which is clearly equivalent to (48). ■

We deduce immediately that the minimum value of $\mathcal{T}$ is obtained when Card $\{t \mid a \in F(t)\} = 1$, or, equivalently, when footprints are disjoint. The value of this minimal traffic, that we will call the *compulsory traffic* is:

$$\mathcal{T} = \text{Card } M_{\mathcal{P}}.$$

A chunk system with this property is said to be perfect. All programs have at least one perfect chunk system which corresponds to $\forall u \in E_{\mathcal{P}} : \theta(u) = 0$. This is useful only if the chunk system satisfies the capacity condition Card $M_{\mathcal{P}} \leq C$, which rarely happens in practice. Most programs do not have any other perfect chunk system. The reason is that a program which has a non trivial perfect chunk system corresponds to L independent programs. Such programs, of the *embarrassingly parallel* variety, are neither frequent nor interesting. In the general case, we should form imperfect chunk systems, where the same cell is accessed by operations belonging to different chunks. The objective in such a case is to minimize the traffic, that is to minimize multiple accesses to identical memory cells.

54.3.2 Decompositions

Let us rewrite the formula of the traffic (48) by introducing **the usage relation**: the set of all pairs ⟨cell,chunk⟩ where "cell" is accessed during the execution of "chunk":

$$U = \{\langle a, t\rangle \mid \exists u \in E_{\mathcal{P}} : \theta(u) = t, a \in D(u)\}. \tag{49}$$

Given the equality

$$\text{Card } \{t \mid a \in F(t)\} = \text{Card } \{\langle a, t\rangle \mid \langle a, t\rangle \in U\}$$

and because the sets $\{\langle a, t\rangle \mid \langle a, t\rangle \in U\}$ are disjoint for different values of a, one has:

$$\begin{aligned}
\mathcal{T} &= \sum_{a \in M_{\mathcal{P}}} \text{Card } \{\langle a, t\rangle \mid \langle a, t\rangle \in U\} \\
&= \text{Card } \cup_{a \in M_{\mathcal{P}}} \{\langle a, t\rangle \mid \langle a, t\rangle \in U\} \\
&= \text{Card } U.
\end{aligned}$$

Hence to find a correct and optimal chunk system, our job is to minimize Card U under the constraints $\forall t :$ Card $F(t) \leq C$.

We can give a more detailed definition of U, knowing that:

- a memory cell $a \in M_{\mathcal{P}}$ is in fact an array cell $A[i],\ A \in T_{\mathcal{P}},\ i \in D_A$;
- an operation $u \in E_{\mathcal{P}}$ is an instance of a statement $S[x],\ S \in I_{\mathcal{P}},\ x \in D_S$;
- $a \in D(u)$ iff $\exists \langle A, f\rangle \in D(S) : i = f(x)$;

this gives finally:

$$U = \cup_{A \in \mathcal{P}} \cup_{S \in \mathcal{P}} \cup_{\langle A, f\rangle \in S} \{\langle A[i], t\rangle \mid \exists x \in D_S : f(x) = i, \theta(S[x]) = t\}.$$

The first observation is that the sets:

$$U_A = \cup_{S \in \mathcal{P}} \cup_{\langle A, f\rangle \in S} \{\langle A[i], t\rangle \mid \exists x \in D_S : f(x) = i, \theta(S[x]) = t\}$$

are disjoint. Hence, when computing the traffic, we can compute their size independently and sum the results. Next, we observe that we can now dispense with A and introduce sets:

$$u_A = \cup_{S \in \mathcal{P}} \cup_{\langle A, f\rangle \in S} \{\langle i, t\rangle \mid \exists x \in D_S : f(x) = i, \theta(S[x]) = t\}.$$

We finally arrive at the traffic decomposition:

$$\begin{aligned} u_{S,A,f} &= \{\langle i,t\rangle \mid \exists x \in D_S : f(x) = i, \theta(S[x]) = t\} \\ &= \{\langle f(x), \theta(S[x])\rangle \mid x \in D_S\}. \end{aligned} \tag{50}$$

The sets $u_{S,A,f}$ are not disjoint. Indeed, an array A can be accessed in various instructions, which means that we should have the union $\cup_{S\in\mathcal{P}}$ so that these sets are separate. Furthermore, the same array can be accessed with several index functions in the same instruction, we should also keep the union $\cup_{\langle A,f\rangle\in S}$ so that they are disjoint. However, for heuristics reasons, one will suppose that they are effectively disjoint. Because at the moment, we limit ourselves to the programs in which every array is accessed only once.

The same decomposition applies also to footprints cardinals. We are lead to the study of the following sets:

$$\begin{aligned} v_{S,A,f}(t) &= \{i \mid \exists x \in D_S : f(x) = i, \theta(S[x]) = t\} \\ &= \{f(x) \mid x \in D_S, \theta(S[x]) = t\}. \end{aligned} \tag{51}$$

54.3.3 Asymptotic Evaluation

Let us first study a set:

$$H = \{Ux \mid Vx = 0, x \in D\}$$

where U and V are arbitrary integral matrices of the right dimension, and where D is a bounded full dimensional domain. Let us first study the dimension of the supporting subspace:

$$K = \{Ux \mid Vx = 0\}.$$

Let $\{v_1, \dots v_k\}$ be a basis for $\ker U \cap \ker V$. This basis can clearly be extended to a basis $\{v_1, \dots v_m\}$ for $\ker V$.

Lemma 54.1 *The vectors $\{Uv_{k+1}, \dots Uv_m\}$ are a basis for K.*

Proof Vectors such that $Vx = 0$ are linear combinations of $v_1, \dots v_m$, and, when multiplied by U, vectors $v_1, \dots v_k$ disappear. Hence, $Uv_{k+1}, \dots Uv_m$ generate H. Suppose they are not linearly independent. There exists $\lambda_{k+1}, \dots \lambda_m$ not all zero such that:

$$\sum_{i=k+1}^{m} \lambda_i U v_i = 0.$$

This implies that the vector $x = \sum_{i=k+1}^{m} \lambda_i v_i$ is in $\ker U$. Since it already is in $\ker V$, it belongs to $\ker U \cap \ker V$, hence is a linear combination of $v_1, \dots v_k$. From this we deduce that the $v_1, \dots, v_m$ are linearly dependent, a contradiction. ■

The dimension of K is thus $l = \dim\ \ker V - \dim\ (\ker U \cap \ker V)$. This means in fact that, given l of the components of $y = Fx$, we can compute the remaining ones by a linear transformation:

$$(y_{l+1}, \dots, y_d)^T = L(y_1, \dots y_l)^T.$$

From this follows:

$$\text{Card}\ \{Ux \mid Vx = 0, x \in D\} = \text{Card}\ \{(Ux)[1..m] \mid Vx = 0, x \in D\}.$$

Suppose now that D is such that the value of each component of x is an integer in a segment of length w. It follows that each component of Ux also is integral and belongs to a segment of length proportional to w. Hence, the size of H if of the order of w^l.

Since dim $\ker V + \text{rank } V =$ number of column V, one has finally

$$\text{Card } H \text{ is of the order of } w^l, \text{ with } l = \text{rank} \begin{pmatrix} U \\ V \end{pmatrix} - \text{rank } V. \tag{52}$$

54.3.4 Application to the estimation of the traffic and footprint sizes

Let us consider first formula (51); because we suppose that the program has static control, the index function is affine:

$$f(x) = Fx + a$$

where F is a matrix of dimension $\rho(A) \times \rho(S)$ and a is a constant vector which can be ignored for size computation. For instance, for the program:

```
for $(i = 0; i < N; i++)$
  for $(j = 0; j < M; j++)$
    S: $A[j+2] += \textit{constant}$ ;
```

the statement S has $\langle A, f \rangle$ where A is an array and f its index function:
$f(i,j) = F \begin{bmatrix} i \\ j \end{bmatrix} + a$, where $F = \begin{bmatrix} 0 & 1 \end{bmatrix}$ and $a = \begin{bmatrix} 2 \end{bmatrix}$.

Similarly, the chunking function is supposed to be of the form:

$$\theta(S[x]) = Tx + b$$

where T is a matrix of dimension $g \times \rho(S)$ and b is a constant vector. We choose as first dimension of T: $g = \max \rho(S)$ to have chunking functions having all the same dimension, given that to add null lines to a matrix does not modify its rank. F can be extracted by analysis of the source code, while T is the unknown of the problem.

In the case of footprints, there are two situations, depending on the value of t:

1. The system of constraints $x \in D_S, \theta(S[x]) = t$ is unfeasible, and $u_{S,A,f}(t)$ is empty. This means that S does not belong to chunk t, and the size of the corresponding footprint is 0. Deciding which statement belongs to which chunk is a problem in program segmenting, and is left for future research.

2. The above system of constraint is feasible. The asymptotic size of the corresponding footprint is directly given by the preceding result, with T as V and F as U. The size is of the order of $O(w^l)$, with $l = \text{rank} \begin{pmatrix} F \\ T \end{pmatrix} - \text{rank } T$.

$$\text{Card } F(t) = O(w^l), \text{ with } l = \text{rank} \begin{pmatrix} F \\ T \end{pmatrix} - \text{rank } T. \tag{53}$$

Consider now the estimation of the traffic as given by (50). In this case there is no constraint on x, which can be represented by saying that V is the null matrix 0. The rank of the null matrix is 0. U is clearly the block matrix $\begin{pmatrix} T \\ F \end{pmatrix}$. Hence, we can finally estimate traffic:

$$\mathcal{T} = O(w^k), \text{ with } k = \text{rank} \begin{pmatrix} T \\ F \end{pmatrix}. \tag{54}$$

54.4 Chunking functions search algorithms

The quality of a chunking operation is quantifiable thanks to the values of traffics (48) and cardinals of footprints (46). Thanks to the asymptotic evaluation we have a way to estimate these values. They result both of the application of chunking functions T_S on the S statement characterized by their index functions. The index functions are given, they directly depend on the source code. The chunking functions are the unknowns which it is necessary to discover.

It is a question here of finding these functions to propose the best chunk system possible. It means a system where footprints respect the capacity condition (47) and where the traffic is minimal. This section is going to propose algorithms which will allow to determine such a chunk system, that is to find the good chunking functions. We will study first of all the simplified case of unique references, to finish by the general case of multiple references.

54.4.1 Unique references case

In this section, we consider the problem of finding chunk system for programs where each statement access only one array. A statement is an unique reference one if it access only a single array: $\forall S \in I_{\mathcal{P}}, \forall c \in D(S) : \exists! A \in T_{\mathcal{P}}, \exists i \in D_A, c = A[i]$. This implies that there is only one index function F_S per statement S.

The principle is to find, for each statement, the function T_S which will generate least traffic, while satisfying the capacity condition. T_S and F_S are the only parameters for traffic evaluations (54) and footprint cardinals (53). Hence, we will search the best possible pair $\left\langle \text{rank}\, T_S, \text{rank} \begin{pmatrix} T_S \\ F_S \end{pmatrix} \right\rangle$ for each statement, and build the matrix T_S which respect these constraints. Furthermore, we will have to guarantee that chuncking function respects dependences. We will do that by reorganizing source program operations. The unique references case algorithm is:

Unique References Algorithm

1. For each statement S of program $\mathcal{P}$:

(a) Find the best pair $\left\langle \text{rank}\, T_S, \text{rank} \begin{pmatrix} T_S \\ F_S \end{pmatrix} \right\rangle$.

(b) Build a T_S matrix with respect to this pair.

In the following, we look over the algorithm operations. We will see first that we can always find a pair in part (1a), and then that the part (1b) always succeeds, provided that rank T_S and rank $\begin{pmatrix} T_S \\ F_S \end{pmatrix}$ respect obvious inequalities.

(1a) In order to find the best pair $\left\langle \text{rank}\, T_S, \text{rank} \begin{pmatrix} T_S \\ F_S \end{pmatrix} \right\rangle$ for a statement S, we have to evaluate traffic (54) and footprint cardinals of generated chuncks (53). We must do it for each possible pair and the solution comes with the one that offers the best results. rank (T) can take values between 0 and the number of lines of T, which is $\rho(S)$. For each possibility, rank $\begin{pmatrix} T_S \\ F_S \end{pmatrix}$ can take its values between $\max(\text{rank}\, F_S, \text{rank}\, T_S)$ and $\min(\rho(S), \text{rank}\, T_S + \text{rank}\, F_S)$. Hence, the algorithm to find the best pair is :

Best Pair Search Algorithm

```
1. $m = \infty$.
2. Best pair $= \emptyset$.
3. For rank $T_S$ from 0 to $\rho(S)$:
    (a) For rank $\begin{pmatrix} T_S \\ F_S \end{pmatrix}$ from $\max(\text{rank}\, F_S, \text{rank}\, T_S)$ to $\min(\rho(S), \text{rank}\, T_S + \text{rank}\, F_S)$:
        i. If $l = \text{rank} \begin{pmatrix} F_S \\ T_S \end{pmatrix} - \text{rank}\, T_S$ respects capacity condition AND
           If rank $\begin{pmatrix} T_S \\ F_S \end{pmatrix} < m$:
            A. $m = \text{rank} \begin{pmatrix} T_S \\ F_S \end{pmatrix}$.
            B. Best pair $= \left\langle \text{rank}\, T_S, \text{rank} \begin{pmatrix} T_S \\ F_S \end{pmatrix} \right\rangle$.
```

This algorithm requires the F_S matrix in order to calculate its rank and use the number of columns: $\rho(S)$. It is also necessary that it knows the order of magnitude of C (47): $C = O(w^l)$, to determine if the capacity condition is respected. For instance, if the reference access a vector which does not fit completely in the cache, we would look for a solution where $C = O(w^0)$. Let us note that there is always a solution. Indeed, the hardest constraint for the capacity condition is $l = 0$, and the last choice will always be the one for rank $T_S = \text{rank} \begin{pmatrix} T_S \\ F_S \end{pmatrix} = \rho(S)$ giving $l = 0$, and where the traffic is maximun: $\mathcal{T} = O(w^{\rho(S)})$.

Example I To illustrate the search for the chunking functions, let us execute the algorithm on an example. This example consists of a program containing only one statement S which reference the array A with index function $F = \begin{bmatrix} -1 & 2 & -1 \\ -2 & 4 & -2 \\ -1 & 2 & -1 \end{bmatrix}$. If this example has nothing realistic, it is going to show the details of the algorithm. We will suppose that the array accessed in this program is too big for the cache, the capacity condition is then $C = O(w^0)$. Given that rank $F = 1$, the possibilities set studied by the algorithm is given in the table below. We can see that the optimal solution is obtained for the pair $\langle 1, 1 \rangle$.

rank T	rank $\begin{pmatrix} T \\ F \end{pmatrix}$	Footprint	Traffic
0	1	$O(w^1)$	$\times$
1	1	$O(w^0)$	$O(w^1)$
	2	$O(w^1)$	$\times$
2	2	$O(w^0)$	$O(w^2)$
	3	$O(w^1)$	$\times$
3	3	$O(w^0)$	$O(w^3)$

(1b) When the optimal pair is found, it remains to build the matrix T from the information which it contains. Let us change this optimal pair of ranks by the optimal pair of the dimensions of kernels $\langle n, m \rangle$, with $m = \rho(S) - \text{rank} \begin{pmatrix} T \\ F \end{pmatrix}$. We know the following facts:

- F is of size $\rho(A) \times \rho(S)$, and $\dim \ker F = \rho(S) - \text{rank } F = p$,
- T is of size $g \times \rho(S)$, and $\dim \ker T = n$,
- $\dim \ker \begin{pmatrix} T \\ F \end{pmatrix} = m$.

To find a matrix T respecting these data, we begin by calculating a basis $e_1..e_p$ of the kernel of F , that we complete in a basis $e_1..e_{\rho(S)}$ of $N^{\rho(S)}$. We form then the matrix M, whose columns are e_i. We calculate the inverse M' of M, possibly multiplying lines by their respective common denominator to stay with integral coefficients. The matrix formed for its $\rho(S) - n$ first lines by the lines of M' numbered of $m + 1$ in $\rho(S) + m - n$ and for the $g - \rho(S) - n$ last lines, of null lines is a valid matrix T . This process is summarized in the following algorithm:

T Building Algorithm

```
1. Calculate $B$, a basis of $\ker F$.
2. Complete $B$ in $B_2$, a basis of $N^{\rho(S)}$.
3. Build matrix $M$:
   (a) For $i$ from 1 to $\rho(S)$:
       i. $i^{th}$ column of $M$ = $i^{th}$ vector of $B_2$.
4. Calculate $M'$, inverse of $M$ which all lines will be multiplied by the PPCM of the denominators
   of their constituents.
5. Build matrix $T$:
   (a) For $i$ from 1 to $\rho(S) - n$:
       i. $i^{th}$ line of $T = (m + i)^{th}$ line of $M'$.
   (b) For $i$ from $\rho(S) - n + 1$ to $g$:
       i. $i^{th}$ line of $T$ = null line.
```

Proof We demonstrate here that algorithm builds a matrix T answering requirements:

$\dim\ \ker T = n$: T is formed of $\rho(S) - n$ linearly independent lines (they come from an invertible matrix), hence $\operatorname{rank} T = \rho(S) - n$ and as by definition $\dim\ \ker T = \rho(S) - \operatorname{rank} T$, we have finally $\dim\ \ker T = n$.

$\dim\ \ker \begin{pmatrix} T \\ F \end{pmatrix} = m$: If M' is the inverse of a matrix M of size $c \times c$, the kernel of any submatrix of M' formed by i of its lines is generated by vectors corresponding to $c - i$ columns of M of additional numbers of those of the i line taken from M. T is formed with the lines of M' from $m+1$ to $\rho(S) + m - n$. The kernel is so generated with e_i for i from 1 to m and from $\rho(S) + m - n + 1$ to $\rho(S)$. Now, the kernel of $\begin{pmatrix} T \\ F \end{pmatrix}$ is the intersection of the kernel of T with the kernel of F. So, intersection is generated by the first m e_i, and is then of dimension m.

■

This algorithm is useful only under three conditions. First of all we must have $m \leq n$ and $m \leq p$. Because m is the dimension of the intersection of kernels, its value is necessarily less or equals of the dimensions of these kernels. The algorithm to find the best pairs guarantees that it is always the case, because it looks for the solutions only in the correct domains. Then, we must have $g \geq \rho(S) - n$. It is a mathematical constraint, because $\operatorname{rank} T = \rho(S) - n$ and because the rank is raised with the number of lines, and for the algorithm, which forms $\rho(S) - n$ lines of T with $\rho(S) - n$ columns of M'. As the choice of g is $\max \rho(S)$, this constraint is respected, and there is always a solution.

Let us resume the example above. The best pair in term of kernel dimensions is $\langle n, m\rangle = \langle 2, 2\rangle$. The various steps of contruction of the chunking function are then:

1. We calculate $\dim\ \ker F = p = 2$ and a basis of $\ker F$: $B = \left\{ \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix} \right\}$.

2. We complete B to form B_2, a basis of N^3:

- we change first of all the vectors of B by combinations so that they have the shape of columns of a lower diagonal matrix:
  $B = \left\{ \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix} \right\}$;
- we add then the vectors of the canonical basis to form the diagonal matrix:
  $B_2 = \left\{ \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right\}$.

3. We form matrix M, whose columns are the vectors of B_2 (pay attention to the order):
$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 2 & 1 \end{bmatrix}$.

4. We calculate M', inverse of M:
$M' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & -2 & 1 \end{bmatrix}$.

5. We form T with $\rho(S) - n = 1$ lines of M' from $m + 1 = 3$ to $\rho(S) + m - n = 3$ and $g - \rho(S) - n = 2$ null lines:

$$T = \begin{bmatrix} 1 & -2 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Note on dependences The chunking is a technique of operations reordering. Hence, it has to respect dependences and not allow the placement of an operation in a chunk of lower number than those containing operations on which it depends. In the case of programs having only statements with unique references, dependences can exist only between the same statement instances. Now in that restricted case, we can not find two operations that do not commute. It means that any reorganization is legal.

We showed that the chunking algorithm in the unique references case gave always a solution. The reorganization of the operations which it proposes is always legal.

Complete example Let us compute a chunking system for the program below, knowing that array v can not fit in the cache.

```
/* original program */
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    S: v[j] = v[j] + constant ;
```

There is only a statement S of which the unique index function is $F = \begin{bmatrix} 0 & 1 \end{bmatrix}$.

1. We calculate rank $F = 1$ to find the optimal pair $\langle 1, 1\rangle$ among all possible solutions:

rank T	rank $\begin{pmatrix} T \\ F \end{pmatrix}$	Footprint	Traffic
0	1	$O(w^1)$	$\times$
1	1	$O(w^0)$	$O(w^1)$
	2	$O(w^1)$	$\times$
2	2	$O(w^0)$	$O(w^2)$

2. We try then to build the matrix T:

- a basis of $\ker F$ is $\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}$;
- we complete the basis of $\ker F$ to form a basis of N^2: $\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$;
- $M = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$;
- we calculate M', inverse of M: $M' = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$;

- we build T from $\rho(S)-n=1$ lines of M' from $m+1=2$ to $\rho(S)+m-n=2$ then until the last line ($g=2$) by null lines:
  $T=\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$.

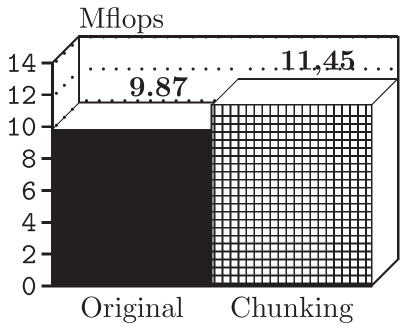
The matrix T thus obtained corresponds to the following chuncking function:

$$\theta(S[i,j]) = j$$

and the final program is:

```
/* Program after chunking */
FLUSH ;
for ($j = 0; j < N; j++$) {
   for ($i = 0; i < M; i++$)
      S: $v[j] = v[j] + constant$ ;
   INVALIDATE &$v[j]$ ;
   }
```

This transformation is the simplest program transformation: loop permutation. Supposing that time taken by instructions FLUSH and INVALIDATE is null, we can simulate the transformation and see the performance evolution, a 15% improvement:



54.4.2 Multiple references case

This section is devoted to the general case of statements with multiple references. A statement S has multiple references if it access various arrays, or several times the same array with different index functions. Difficulty with regard to unique references case is to obtain a correct chunking function from several index functions. Let x_S be the number of index functions in statement S. Our purpose is so to find for every statement S the ideal pair $\left\langle \dim\ \ker T_S, \left\{ \dim\ \ker \begin{pmatrix} T_S \\ F_{S_i} \end{pmatrix} \right\}_{1\le i\le x_S} \right\rangle$, such that every $\left\langle \dim\ \ker T_S, \dim\ \ker \begin{pmatrix} T_S \\ F_{S_i} \end{pmatrix} \right\rangle$ satisfies the capacity condition associated to the array which F_{S_i} indexes, and such as the global traffic generated for S: $\mathcal{T}_S = \sum_{i=1}^{x_S} \mathcal{T}_{S_i}$ is minimal. Besides, this ideal pair should allow the construction of a chunking matrix T_S which respect dependences. An algorithm to search for the chuncking functions for multiple reference statement programs is:

Multiple References Algorithm

1. For each S statement of $\mathcal{P}$ program:

(a) List of pairs $= \{\emptyset\}$.

(b) For rank T_S from 0 to $\rho(S)$:

i. Find all sets $\left\{\dim\ \ker\begin{pmatrix} T_S \\ F_{S_i}\end{pmatrix}\right\}_{1\leq i\leq x_S}$ such that for the array indexed by F_{S_i} we have $l_i = \text{rank}\begin{pmatrix} T_S \\ F_{S_i}\end{pmatrix} - \text{rank}\, T_S$ satisfying the capacity condition, and form for each of them a new pair $\left\langle \dim\ \ker T_S, \left\{\dim\ \ker\begin{pmatrix} T_S \\ F_{S_i}\end{pmatrix}\right\}_{1\leq i\leq x_S}\right\rangle$ in the list.

(c) Classify the pairs in order of increasing $\mathcal{T}_S$.

(d) Look for the first pair in the list for which a matrix T_S exists.

This algorithm requires as data the matrices F_{S_i} and the orders of magnitude for the capacity conditions ((47) and (51)) for every array. The first part is similar to the best pair search algorithm seen in section 54.4.1. In particular, we can prove that there is always at least a pair to be placed in the list because for every F_{S_i}, there is a possibility rank $\begin{pmatrix} T_S \\ F_{S_i}\end{pmatrix} = \rho(S)$ respecting the hardest condition: $l = 0$ and corresponding to the pair $\left\langle 0, \left\{\dim\ \ker\begin{pmatrix} T_S \\ F_{S_i}\end{pmatrix} = 0\right\}_{1\leq i\leq x_S}\right\rangle$.

Example II Let us illustrate the first step of the algorithm by the study of a program consisting of a unique statement containing $x = 2$ references. Each reference access a different array. The two index functions, the capacity conditions and the results of the study which would make the best pair search algorithm for each reference are the following ones:

- $F_1 = \begin{bmatrix} 2 & -1 & -1 \\ 1 & 0 & -1 \end{bmatrix}$ (rank $F_1 = 2$) with capacity condition $\forall t,\ \text{Card}\ F(t) \leq O(w_1^1)$:

$\dim\ \ker T$	$\dim\ \ker\begin{pmatrix} T \\ F_1\end{pmatrix}$	Footprint	Traffic
3	1	$O(w_1^2)$	$\times$
2	1	$O(w_1^1)$	$O(w_1^2)$
	0	$O(w_1^2)$	$\times$
1	1	$O(w_1^0)$	$O(w_1^2)$
	0	$O(w_1^1)$	$O(w_1^3)$
0	0	$O(w_1^0)$	$O(w_1^3)$

- $F_2 = \begin{bmatrix} 1 & -2 & 1 \\ 1 & -2 & 1 \end{bmatrix}$ (rank $F_2 = 1$) with capacity condition $\forall t,\ \text{Card}\ F(t) \leq O(w_2^0)$:

dim ker T	dim ker $\begin{pmatrix} T \\ F_2 \end{pmatrix}$	Footprint	Traffic
3	2	$O(w_2^1)$	$\times$
2	2	$O(w_2^0)$	$O(w_2^1)$
	1	$O(w_2^1)$	$\times$
1	1	$O(w_2^0)$	$O(w_2^2)$
	0	$O(w_2^1)$	$\times$
0	0	$O(w_2^0)$	$O(w_2^3)$

The pairs $\left\langle \text{dim ker}\, T, \left\{ \text{dim ker} \begin{pmatrix} T \\ F_i \end{pmatrix} \right\}_{1 \le i \le 2} \right\rangle$ classified in order of increasing global traffic are: $\langle 2, \{1, 2\} \rangle$, $\langle 1, \{1, 1\} \rangle$, $\langle 1, \{0, 1\} \rangle$, $\langle 0, \{0, 0\} \rangle$.

We have then to find the first pair for which a correct matrix T exists. It is not possible to rule on the existence of a solution from information contained in a pair only, even without taking into account the dependence problem . Indeed, the matrix T is closely linked to the properties of matrices F_i, and in particular to the basis vectors of their kernels. The construction of a matrix T will be tried case by case considering the constraints that are matrices F_i and dim ker $\begin{pmatrix} T \\ F_i \end{pmatrix}$ values. The dependence constraint will be treated last. The algorithm which is going to be presented is a generalization for x index matrices of the T building algorithm presented in section 54.4.1. To build T, we then know:

- matrices F_i, of sizes $\rho(A_i) \times \rho(S)$, and with dim $\ker F_i = \rho(S) - \text{rank } F_i = p_i$,
- of the matrix T: its size $g \times \rho(S)$, and dim $\ker T = n$,
- the dim ker $\begin{pmatrix} T \\ F_i \end{pmatrix} = m_i$.

The major difference between the generalized algorithm and the unique references version is in the construction of the set B. This set is a partial generator of the kernel of T. To have the required m_i, it contains a limited number of linearly independent vectors of the basis of $\ker F_i$. Indeed, to have a solution T such that dim $\ker T = n$, T has to be formed of $\rho(S) - n$ linearly independent vectors. The kernel of such a matrix can be generated only by n linearly independent vectors, hence B has to contain at most n vectors.

The principle of construction of B is to begin with $B = \{\emptyset\}$ and to successively include for each F_i, m_i vectors of a basis of $\ker F_i$. The choice of vectors to be included in B is essential. The order in which we add vectors of $\ker F_i$ is not important. But we sometimes have for the same F_i several choices of vectors. Some choices will lead to exceed the n vectors limit in B or to make impossible the obtaining of a m_j, for any $j \neq i$. We have an interest to take as many vectors as possible from vect $B \cap \ker F_i$ because it limits the number of vectors in B, within interference on a constraint m_j, for any $j \neq i$. We can not foresee the best choice of dim $\ker F_i$ − dim (vect $B \cap \ker F_i$) vectors of $\ker F_i$ remaining to include before having studied every F_i. That's why proposed algorithm uses a tree to retain every set B possible. The T search algorithm in the multiple references case will be:

T Building Generalized Algorithm

1. Tree initialization: root at level 0 and a leaf at level 1 carrying $B = \{\emptyset\}$.

2. For i from 1 to x:
For all the leaf of the tree at level i:
If dim vect $B + m_i - \dim(\text{vect } B \cap \ker F_i) \leq n$:

(a) Find the r combinations of $m_i - \dim(\text{vect } B \cap \ker F_i)$ vectors of a basis of $\ker F_i$ such that for every combinations C we have:
- $\dim(\text{vect } B \cap \text{vect } C) = 0$,
- $\forall j, j \neq i, \dim \text{vect } B + \dim \text{vect } C + \dim\ \ker F_j - \dim(\text{vect } (B \cup C) \cap \ker F_j) \leq m_j$.

(b) Transform the current leaf in node with r leaf of $i+1$ level contening for each a new set B formed with one of the combinations:
$B = \{B, \text{combination}\}$.

(c) If we are on level x, for every created leaf:
   i. Complete B to B_2, a basis of $N^{\rho(S)}$.
   ii. Form the matrix M:
      A. For i from 1 to $\rho(S)$:
         i^{th} column of M = i^{th} vector of B_2.
   iii. Calculate M', inverse of M and multipy it by Det(M).
   iv. Form the T matrix:
      A. For i from 1 to $\rho(S) - n$:
         i^{th} line of T = $(\dim B + i)^{th}$ line of M'.
      B. For i from $\rho(S) - n + 1$ to g:
         i^{th} line of T = null line.
   v. If T respects dependences, or can be modified to respect them, T is a solution.

Proof This algorithm guarantees that the set B is formed for every F_i of exactly m_i vectors of a basis of $\ker F_i$. The vectors of B are linearly independent, which is a consequence of our choice to use every time that we treat one F_i the vectors of vect $B \cap \ker F_i$. Finally, B contains at most n vectors. In these conditions, we can refer to the demonstration of the T building algorithm seen in section 54.4.1 to see that when a set B was found, a correct matrix T is built. ■

Let us resume example II. The first pair to be studied is $\langle 2, \{1, 2\}\rangle$. We have:

- for the matrix $F_1 = \begin{bmatrix} 2 & -1 & -1 \\ 1 & 0 & -1 \end{bmatrix}$, $B_1 = \text{vect } \ker F_1 = \left\{ \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\}$;
- for the matrix $F_2 = \begin{bmatrix} 1 & -2 & 1 \\ 1 & -2 & 1 \end{bmatrix}$, $B_2 = \text{vect } \ker F_2 = \left\{ \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix} \right\}$;
- $\text{vect } B_1 \cap \text{vect } B_2 = \left\{ \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\}$.

The tree until the obtaining of a complete B set is:

$$\{\emptyset\} \quad \rightarrow \quad \left\{ \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\} \quad \rightarrow \quad \left\{ \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix} \right\}.$$

We use then the T construction algorithm:

- we form $M = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 2 & 1 \end{bmatrix}$,
- we calculate $M' = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & -2 & 1 \end{bmatrix}$,
- T is formed by $\rho(S)-n = 1$ lines of M' from the line $\dim B+1 = 3$, and with $g-(\rho(S)-n+1) = 2$ null lines:
  $T = \begin{bmatrix} 1 & -2 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$.

The T search algorithm can not always find solutions which may not exists. However, when solutions exist, it always allows to find one, because it is going to study all the possibilities for B (apart from the fact that there is a multiplicative coefficient by vector). But the algorithm will always find a solution. Indeed, as we saw, the pair $\langle 0, \{m_i = 0\}_{1 \leq i \leq x} \rangle$ is inevitably in the list, and the solution: $T = I$ always exists.

54.4.3 The dependence problem

Contrary to the unique references case, operations involving several references can not always commute. We have to make sure that when there are dependences, reorganization defined by chunking function for a statement respects them. We can distinguish two cases:

- The case where accesses to identical memory cells are caused by operations from the same statement only, that is when each dependence is a self-dependence. It will be in particular the case of programs containing only one statement. In such a case, the dependence vectors are reduced to the kernel basis vectors of the statement index functions, with a lexicopositive shape (that is the first non null element of the vector is positive). For each dependence vector $\vec{d}$, we should have $T\vec{d}$ lexicopositive, i.e. $T\vec{d} \gg 0$. Such a case will often solved by sign changes of some T lines, which do not change the T properties. This case is already solved by chunking.
- The case where several statements make reference to identical memory cells. We should then find a set of matrices T_S satisfying dependences. Let us remember that chunking suggests that if an operation must be executed before an other one, that one has to belong to a chunk of lower or equal number. Each chunking functions being independently determined in relation to the others, nothing forbids an illegal interlacing of operations from different instructions. It is a question of correcting introduced error by modifying the chunking matrices without alteration of their properties that is the n and m_i values. Only the following transformations are possible:
  - multiplication of lines by non null scalars;
  - lines exchange;
  - addition to a line of a linear combination of the other lines.

This means finding an inversible adequate matrix of the same size as T_S and to multiply it by T_S. For each matrix T_S we should look for a correction matrix C_S to have finally the chunking function $\theta_S = C_S T_S$. These matrices C_S are determined by the dependence system resolution. This system is formed in such a way that if an instance of V with iteration vector i_1 must be executed before an instance of W with iteration vector i_2, there is an equation of the shape:

$$C_w T_w i_2 - C_v T_v i_1 \gg 0.$$

The automatic resolution of such a system is not easy, in particular because the "C_S invertible" constraint is non linear. Besides, the existence of a solution can not be guaranteed. So it will be necessary in some cases to choose a different matrix T_S for some statements (the T building algorithm can give all the independent solutions according to the transformations above), even less effective and to envisage compromises. To automate the resolution of this difficulty or to by-pass it are the objects of current reseach.

54.5 Future works

In what was presented here, only the data dependence issue in the general case remains unfinished. Nevertheless, many other tools are needed to make chunking possible and to improve its efficiency. Our objective is to produce a tool able to receive as input a source code, and to return an optimized code which can be used in a real-time framework for machines with caches. We have mentioned here only the main algorithm, which consists in finding the best chunking functions. This central part can be improved, in particular by the use of better footprint size estimation methods, which would allow us to find tiling-like optimization. It will be also necessary to try to minimize the traffic when identical arrays are accessed in various statement by building correlated chunking matrices. Secondly, we still have to design the code generation phase. This part will make operation reorganization, using chunking functions and original code informations such as array sizes. We should find there ways to improve spatial locality. The principle of the respect for the capacity conditions is limited to the presence of a unique cache level, it will be also interesting to study a generalization with several levels.

Once this tasks are complete, we will be able to study other applications of this method. For instance, it could be a good solution for virtual memory management by the user.

55 Conclusion

This report presents, through various techniques and illustrations, the impact of the memory hierarchy on performance. We recall the two major transformation families allowing data locality improvement. Program transformations are well known, and effective tools have been designed; they enable both temporal and spatial locality improvement, and can work on program parts. Data transformations can be applied to more complex programs with dependences, and is well adapted to parallel program optimizations. Their large complementarity generates an interest in mixed solutions, results of which are promising, in spite of only partial integration of program transformations. Nevertheless we were able to see that these techniques, combined or not, suffer serious limitations. The NP-hard characteristics of the choice of program transformations to be applied, as well as the order in which to apply them, imposes heuristics methods. Data layout transformations lack flexibility, forbidding various layouts for the same array, this can be bad for performances.

We argued that the major deficiency of classical optimization techniques was not to take into account software cache controls. We proposed chunking, a new method which takes advantage of cache management tools. Eventually, this technique should allow maximal data locality use, without limitations in existing techniques. More than improving locality, it is one of the first optimization methods for program compilation on real-time systems with caches. Besides, it will be possible to apply it lower in the memory hierarchy, between hard disk and main memory. We will then be able to optimize out of core programs on classical architectures.

The methods presented in this paper are implemented in the "chunky" prototype that takes as input cache size and source code informations, and generate chunking functions which will subsequently be used for code generation. A lot of works should be done before chunking can compete with existing techniques for performance. To solve the dependence problem in the multiple reference case will be the first of them. It will also be a question of allowing tiling-like transformations. It will remain finally to integrate the spatial locality improvement.

Part 1

References

[1] Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman
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.

[3] Allen, Randy and Steve Johnson
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, 1988.

[4] Analog Devices, Inc.
Quad-SHARC DSP Multiprocessor Family AD14160/AD14160L
`http://www.analog.com`, Norwood, MA, USA, 1998.

[5] Analog Devices, Inc.
ADSP-2183 Datasheet
`http://www.analog.com`, Norwood, MA, USA, 1998.

[6] Appel, A.W. and M. Ginsburg
Modern Compiler Implementation in C
Cambridge University Press, Cambridge, United Kingdom, 1998.

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

[8] The American Amateur Radio League
The ARRL Handbook for Radio Amateurs
Newington, CT, USA, 1999.

[9] Bhattacharyya, S.S., R. Leupers and P. Marwedel
Software Synthesis and Code Generation for Signal Processing Systems
Technical Report UMIACS-TR-99-57, Institute for Advanced Computer Studies, University of Maryland, 1999.

[10] Cheng, W.-K. and Youn-Long Lin
Code Generation for a DSP Processor
Proceedings of the 7th Internation Symposium on High-Level Synthesis, Canada, 1994.

[11] DeFatta, D. et al.
Digital Signal Processing: A System Design Approach
Wiley, New York, 1998.

[12] Dingel, S. and Rainer Leupers
LANCE - LS12 ANSI C Compilation Environment, User's Guide
`http://ls12-www.informatik.uni-dortmund.de`, 1999.

[13] Duesterwald, E., R. Gupta and M. Soffa
A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations
Proceedings of the SIGPLAN Conference on Programming Languages Design and Implementation, 28(6), pp. 67-77, Albuquerque, New Mexico, 1993.

[14] Fettweis, G., M. Weiss, W. Drescher, U. Walther, F. Engel and S. Kobayashi
Breaking new grounds over 3000 MOPS: A broadband mobile multimedia modem DSP
Proceedings of ICASSP '98, pp. 31-34, München, Germany, 1998.

[15] Forster,M., Andreas Pick and Marcus Rainer
The Graph Template Library (GTL) Manual
`http://www.fmi.uni-passau.de`, 2000.

[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
Van Nostrand Reinhold, New York, 1994.

[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
`http://www.pixelfusion.com`, Bristol, UK, 2000.

[20] Gebotys, C.H. and Robert J. Gebotys
Complexities in DSP Software Compilation: Performance, Code Size, Power, Retargetability
31st Hawaii International Conference on System Sciences (HICSS '98), 1998.

[21] Horwitz, S., P. Pfeiffer and T. Reps
Dependence Analysis for Pointer Variables
Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, pp. 28-49, ACM Press, 1989.

[22] Hu, Ping
The Techniques for Software Pipelining Loops with Conditional Constructs
`http://www-rocq.inria.fr/~hu/`, 2000.

[23] Iwata, E. and Kunle Olukotun
Exploiting Coarse-Grain Parallelism in the MPEG-2 Algorithm
Stanford University, Computer Systems Lab, Technical Report CSL-TR-98-771, 1998.

[24] Kernighan, Brian W. and Dennis M. Ritchie
The C Programming Language, Second Edition, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[25] Strategies for Realistic and Efficient Static Scheduling of Data Independent Algorithms onto Multiple Digital Signal Processors
Ph.D. thesis, Aalborg University, Institute for Electronic Systems, DSP Research Group, Denmark, 1995.

[26] Lapsley, Phil
DSP processor fundamentals : architectures and features
IEEE Press, New York, 1997.

[27] Lee, C., Miodrag Potkonjak and William H. Mangione-Smith
MediaBench: A Tool for Evaluating and Synthesizing Multimedia Communications Systems
Proceedings of the 30th Annual International Symposium on Microarchitecture, 1997.

[28] Leupers, Rainer
Novel Code Optimzation Techniques for DSPs
2nd European DSP Education and Research Conference, Paris, France, 1998.

[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 is 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
EMMSEC '99, Stockholm, Sweden, 1999.

[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, pp. 321-329, 1997.

[37] Liao, S., S. Devedas, K. Keutzer, S. Tijang and A. Wang
Code Optimization Techniques for Embedded DSP Microprocessors
Design Automation Conference 1995.

[38] Lu, J.
Interprocedural Pointer Analysis for C
Ph.D. thesis, Department of Computer Science, Rice University, Houston, Texas, 1998.

[39] Marwedel, P. and G. Goossens
Code Generation for Embedded Processors
Kluwer Academic Publishers, 1995.

[40] Marwedel, Peter
Compilers for Embedded Systems
`http://ls12-www.informatik.uni-dortmund.de`, Invited Embedded Tutorial, SASIMI, Osaka, 1997.

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

[42] Maydan, Dror.E., John L. Hennessy, and Monica S. Lam.
Effectiveness of Data Dependence Analysis
International Journal of Parallel Programming, 23(1):63-81, 1995.

[43] MediaBench home: `http://www.cs.ucla.edu/~lee/mediabench`.

[44] Muchnick, Steven.S.
Advanced Compiler Design and Implementation
Morgan Kaufmann Publishers, San Francisco, California, 1997.

[45] Oppenheim, A.V., and R.W. Schafer
Digital Signal Processing
Prentice Hall, 1975.

[46] Panda, P.R., N.D. Dutt and A. Nicolau
Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications
Proceedings of the 1997 European Design and Test Conference (ED&TC,1997), March 1997.

[47] Philips Semiconductors
R.E.A.L. DSP Core
`http://www.semiconductors.philips.com`, Eindhoven, Netherlands, 1998.

[48] Philips Semiconductors
TM-1300 Media Processor Data Book
`http://www.semiconductors.philips.com`, Eindhoven, Netherlands, 2000.

[49] TM-1300 Media Processor
`http://www.semiconductors.philips.com`, Eindhoven, Netherlands, 1999.

[50] Philips Semiconductors
Philips TriMedia SDE Reference I, Part 2: Program Development Tools
`http://www.semiconductors.philips.com`, Eindhoven, Netherlands, 1997.

[51] Rajan, S.P., Ashok Sudarsanam and Sharad Malik
Development of an Optimizing Compiler for a Fujitsu Fixed-Point Digital Signal Processor
ACM CODES '99, Rome, Italy, 1999.

[52] Rau, Bob Ramakrishna.
Data Flow and Dependence Analysis for Instruction Level Parallelism
Lecture Notes in Computer Science, Vol. 589, p. 236-250, 1991.

[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 Embedded Systems, CASES '98, 1998.

[56] Siroyan
Product strategy
`http://www.siroyan.com`, 1999.

[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 of Uppsala, December 1998.

[58] Stoodley, Mark G. and Corinna G. Lee
Software Pipelining Loops with Conditional Branches
Proceedings of the 29th annual IEEE/ACM International Symposium on Microarchitecture, pp. 262-273, Paris, France, 1996.

[59] Stotzer, Eric
Modulo Scheduling for the TMS320C6x VLIW DSP Architecture
ACM LCTES '99, Atlanta, GA, USA, 1999.

[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 Paterson University, 1998.

[61] Texas Instruments, Inc.
TMS320C8x System-Level Synopsys
`http://www.ti.com`, Houston, Texas, USA, 1995.

[62] Texas Instruments, Inc.
TMS320C80 Multimedia Video Processor Data Sheet
`http://www.ti.com`, Houston, Texas, USA, 1995.

[63] Tietze, Ulrich, and Christoph Schenk
Halbleiter-Schaltungstechnik
10th Edition, Springer-Verlag, Berlin, 1993.

[64] Wilson, R.P.
Efficient Context-Sensitive Pointer Analysis for C Programs
Ph.D. thesis, Stanford University, Computer Systems Laboratory, December 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
ACM, DAC 98, San Francisco, California, 1998.

[66] Zivojnovic, V., J.M. Velarde, C. Schlager and H. Meyr
DSPstone: A DSP-Oriented Benchmarking Methodology
Proceedings of Signal Processing Applications & Technology, Dallas 1994.

Part 2, 3 and 4

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] F.E. Allen and J.Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[3] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9:491–542, 1987.

[4] Apple. *AltiVec Home Page.* http://developper.apple.com/hardware/altivec, may 1999.

[5] Eduard Ayguadé et al. *A uniform internal representation for high-level and instruction-level transformations.* UPC, 1995.

[6] M. Barreteau, F. Bodin, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. Gurd, J. Hoogerbrugge, P. Hu, W. Jalby, T. Kisuki, P.M.W. Knijnenburg, P. van der Mark, A. Nisbet, M.F.P. O'Boyle, E. Rohou, A. Seznec, E.A. Stöhr, M. Treffers, and H.A.G. Wijshoff. OCEANS: Optimizing compilers for embedded applications. In P. Amestoy *et al.*, editor, *Proc. Euro-Par 99*, volume 1685 of *Lecture Notes in Computer Science*, pages 1171–1175, 1999.

[7] Bazaraa, Shetty, and Sherali. *Nonlinear programming: theory and applications.* Wiley, 1994.

[8] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 1966.

[9] Nerina Bermudo, Xavier Vera, Antonio González, and Josep Llosa. An efficient solver for cache miss equations. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2000.

[10] R. Bhargava, L. John, B. Evans, and R. Radhakrishnan. Evaluating mmx technology using dsp and multimedia applications. In *Micro 31*, 1998.

[11] J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proc. ICS'97*, pages 340–347, 1997.

[12] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proc. Workshop on Profile and Feedback Directed Compilation*, 1998.

[13] Doug Carmean. *Inside the Pentium 4 Processor Micro-Architecture (www.intel.com/pentium4)*, 2000.

[14] S. Carr. Combining optimization for cache and instruction level parallelism. In *Proc. PACT'96*, pages 238–247, 1996.

[15] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[16] Brian Case. 3dnow boosts non-intel 3d performance. *Microprocessor Report*, 12(7):18–21, juin 1998.

[17] I-Cheng K. Chen, John T. Coffey, and Trevor N. Mudge. Analysis of Branch Prediction via Data Compression. *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 128–137, October 1996.

[18] Alvin R. Lebeck Chia-Lin Yang, Barton Sano. Exploiting instruction level parallelism in geometry processing for three dimmensional graphics applications. In *Micro 31*, November 1998.

[19] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizatons. In *Proc. 2nd Workshop on Feedback Directed Optimization*, 1999.

[20] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: applications to analyze ans transform scientific programs. In *ICS96*, 1996.

[21] R. Cohn and P.G. Lowney. Feedback directed optimization in Compaq's compilation tools for Alpha. In *Proc. 2nd Workshop on Feedback Directed Optimization*, 1999.

[22] S. Coleman and K.S. McKinley. Tile size selection using cache organization and data layout. In *Proc. PLDI'95*, pages 279–290, 1995.

[23] H. Corporaal. *Microprocessor Architectures: From VLIW to TTA*. John Wiley, 1997.

[24] R. Crisp. Direct rambus technology: the main memory standard. *IEEE Micro*, 17:18–28, nov 1997.

[25] Susan L.Graham David F.Bacon and Olivier J.Sharp. Compiler transformations for high-performance computing. Technical report, University of California, 1994.

[26] G. de Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[27] Y. Ermoliev and R.J.-B. Wets. *Numerical Techniques for Stochastic Optimization*. Springer-Verlag, 1988.

[28] Alan Watt et Mark Watt. *Advanced Animation and Rendering Techniques*. Addison Wesley, 1992.

[29] Agustín Fernández. A quantitative analysis of the specfp95. Technical Report UPC-DAC-1999-12, Universitat Politècnica de Catalunya, March 1999.

[30] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. *J. Parallel and Distributed Computing*, 5:587–616, 1988.

[31] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: an analytical representation of cache misses. In *ICS97*, 1997.

[32] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *ASPLOS98*, 1998.

[33] Gill, Murray, and Wright. *Practical optimization*. Academic Press, 1981.

[34] Glover and Laguna. *Tabu search*. Kluwer, 1997.

[35] D.E. Goldberg. *Genetic algorithms in search, optimizations and machine learning.* Addison-Wesley, 1989.

[36] S. Gosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tunig memory behavior. *ACM Trans. on Programming Languages and Systems*, 21(4):703–746, 1999.

[37] H. Han, G. Rivera, and C.-W. Tseng. Software support for improving locality in scientific codes. In *Proc. CPC2000*, pages 213–228, 2000.

[38] Hansen, Jaumard, and Mathon. Constrained nonlinear 0-1 programming. *ORSA Journal on Computing*, 1995.

[39] J. Holland. *Adaptation in natural and artificial systems.* The University of Michigan Press, Ann Arbor, 1975.

[40] Host, Pardalos, and Thoai. *Introduction to global optimization.* Kluwer, 1995.

[41] http://www.altivec.org. *AltiVec.org*, 2000.

[42] Wen-mei W. Hwu et al. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1/2):229–248, May 1993.

[43] Intel. Mmx technology architecture overview. *Intel Technology Journal*, sept 1997.

[44] Intel. Streaming simd extensions. *Intel Technology Journal*, jan 1999.

[45] Intel. *Streaming SIMD Extensions - 3D Transformations.* in AP597, January 1999.

[46] Kirkpatrick, Gelatt, and Vecchi. Optimization by simulated annealing. *Science 220*, 1983.

[47] T. Kisuki, P.M.W. Knijnenburg, and M.F.P. O'Boyle. Iterative compilation for tile sizes and unroll factors: Implementation, performance, search strategies. Technical Report TR2000-06, LIACS, Leiden University, 2000.

[48] T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, F. Bodin, and H.A.G. Wijshoff. A feasibility study in iterative compilation. In *Proc. ISHPC'99*, volume 1615 of *Lecture Notes in Computer Science*, pages 121–132, 1999.

[49] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. ASPLOS'91*, pages 63–74, 1991.

[50] Gyungho Lee, Clyde P. Kruskal, and David J. Kuck. An empirical study of automatic restructuring of nonnumerical programs for parallel processors. *IEEE Transactions on Computers*, pages 927–933, 1985.

[51] R. Lee and J. (Hewlett Packard) Huck. *64-bits and Multimedia Extensions in the PA-RISC 2.0 Architecture.* http://www.hp.com/ahp/framed/technology/micropro/architecture/docs/pa2go3.html, 1996.

[52] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proc. MICRO 25*, 1992.

[53] Z. Michalewicz. *Genetic algorithms+Data structures=Evolution Programs.* Springer-Verlag, 1994.

[54] Sun Microelectronics. *The VIS Instruction Set.* http://www.sun.com/microelectronics/vis/, 1995.

[55] Micron. *DDRSDRAM features and options.* http://www.micron.com/msp/ddrsdramfeat.html, 1999.

[56] M. Mock, M. Berryman, C. Chambers, and S.J. Eggers. Calpa: A tool for automating dynamic compilation. In *Proc. 2nd Workshop on Feedback Directed Optimization*, 1999.

[57] Motorola. *Motorola's high-performance vector parallel processing expansion to the PowerPC architecture.* http://www.motorola.com/SPS/PowerPC/AltiVec/, 1999.

[58] Motorola. Mpc7400 risc microprocessor hardware sepcification. Technical report, Motorola, sept 1999.

[59] S.S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[60] S.G. Nash and A. Sofer. *Linear and nonlinear programming.* McGraw Hill, 1996.

[61] H. Nguyen and L. K. John. Exploting simd parallelism in dsp and multimedia algorithm using altivec technology. In *ICS99*, 1999.

[62] A. Nisbet. GAPS: Genetic algorithm optimised parallelization. In *Proc. Workshop on Profile and Feedback Directed Compilation*, 1998.

[63] M.F.P. O'Boyle and P.M.W. Knijnenburg. Efficient parallelization using combined loop and data transformations. In *Proc. IEEE Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 283–291, 1999.

[64] M.F.P. O'Boyle, N.P. Motogelwa, and P.M.W. Knijnenburg. Feedback assisted iterative compilation. Technical Report 012, Division of Informatics, Edinburgh University, 2000.

[65] David Padua et al. *Polaris developer's document*, 1994.

[66] Brian Paul. The mesa 3d-graphic library. http://www.mesa3d.org, Jan 1999.

[67] R. A. Quinnel. Speech recognition: No longer a dream but still a challenge. *EDN*, pages 41–46, jan 1995.

[68] L. R. Rabiner and B. H. Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, page 5, jan 1986.

[69] R.Allen and K.Kennedy. Automatic translation of fortran programs to vector from. In *ACM Transactions on Programming Languages and Systems 9, 4, 491-542*, 1987.

[70] B.R. Rau. Iterative modulo scheduling. *International Journal of Parallel Programming*, 24(1), 1996.

[71] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *Proc. 8th Int'l Conf. on Compiler Construction*, 1999.

[72] R.Schreiber and Jack Dongarra. Automatic blocking of nested loops. Technical report, RIACS, 1990.

[73] Samsung Semiconductors. *Double Data Rate synchronous DRAM.* http://www.intl.samsungsemi.com/Memory/DRAM/Next_Generation/DDR_SDRAM/DDR.htm, 1999.

[74] G. Sierksma. *Linear and integer programming: theory and practice.* M. Dekker, 1996.

[75] Torn and Zilinskas. *Global optimization.* Springer-Verlag, 1989.

[76] P. van der Mark, E. Rohou, F. Bodin, Z. Chamski, and C. Eisenbeis. Using iterative compilation for managing software pipeline – unrolling tradeoffs. In *Proc. SCOPES99*, 1999.

[77] Xavier Vera, Josep Llosa, Antonio González, and Nerina Bermudo. A fast and accurate approach to analyze cache memory behavior. Submitted to European Conference on Parallel Computing (Europar), 2000.

[78] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proc. Alliance 98*, 1998.

[79] M.E. Wolf, D.E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. *Int'l. J. of Parallel Programming*, 26(4):479–503, 1998.

Part 5

References

[All72] F. Allen, J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, Prentice-Hall, Englewood Cliffs, 1972.

[Anc91] C. Ancourt, F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1991.

[And95] J.M. Anderson, S.P. Amarasinghe, M.S. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing*, Kluwer Academic, 1988.

[Ber66] A.J. Bernstein Analysis of programs for parallel processing. In *IEEE Transactions on Electronic Computers*, Vol. 15, No 5, 1966.

[Car94] S. Carr, K. Kennedy.G. Improving the ratio of memory operations to floating-point operations in loops. In *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 6, 1994.

[Cal90] D. Callahan, S. Carr, K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, ACM, New York, 1990.

[Cie95] M. Cierniak, W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation*, ACM, New York, 1995.

[Cla00] P. Clauss, B. Meister. Automatic Memory Layout Transformations to Optimize Spatial Locality in Parameterized Loop Nests. In *INTERACT'4, Fourth Annual Workshop on Interaction Between Compilers and Computer Architectures*, Toulouse, France, 2000.

[Gof91] G. Goff, K. Kennedy, C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, ACM, New York, 1991.

[Hen90] J.L. Hennessy, D.A. Patterson. *Computer Architecture a Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1990.

[Iri88] F. Irigoin, R. Triolet. Supernode partitionning. In *Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ACM, New York, 1988.

[Kan97] M. Kandemir, J. Ramanujam, A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proceedings of the Eleventh ACM Int'l Conference on Supercomputing*, 1997.

[Kan98] M. Kandemir, A. Choudhary, J. Ramanujam, P. Banerjee. Optimizing spatial locality in loop nests using linear algebra. In *Proceedings of the Seventh Workshop on Compilers for Parallel Computers*, 1998.

[Kan99] M. Kandemir, J. Ramanujam, A. Choudhary. Improving cache locality by a combination of loop and data transformations. In *IEEE Transactions on Computers*, Vol. 48, No. 2, 1999.

[Ken93] K. Kennedy, K.S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, Springer-Verlag, Berlin, 1993.

[Kod97] I. Kodokula, N. Ahmed, K. Pingali. Data-centric multi-level blocking. In *Proceedings of the SIGPLAN'97 Conference on Programming Language Design and Implementation*, ACM, New York, 1997.

[Kuc81] D. Kuck, R. Kuhn, D. Padua, B. Leasure, M.E. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the 8th Annual ACM Symposium on the Principles of Programming Languages*, ACM, New York, 1981.

[Kul98] D. Kulkarni. Transformations for improving data access locality in non-perfectly nested loops. In *IEEE PACT'98 International Conference on Parallel Architectures and Compilation Techniques*, 1998.

[Kwo96] O. Kwon, G. Park, T. Han. A compiler optimization to reduce execution time of loop nest. In *Computer Architecture News*, Vol. 24, No 1, 1996.

[Lam91] M.S. Lam E.E. Rothberg, M.E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, New York, 1991.

[Lef97] V. Lefebvre, P. Feautrier. Storage management in parallel programs. In *IEEE PDP'97 Fifth Euromicro Workshop on Parallel and Distributed Processing*, 1997.

[LiW93] W. Li. *Compiling for NUMA parallel machines*, Ph.D. Thesis, Cornell University, 1993.

[McK96] K.S. McKinley, S. Carr, C. Tseng. Improving data locality with loop transformations. In *ACM Transactions on Programming Languages and Systems*, 1996.

[OBo96] M. O'Boyle, P. Knijnenburg. Non-singular data transformations : definition, validity and applications. In *Proceedings of the Sixth Workshop on Compilers for Parallel Computers*, 1996.

[OBo98] M. O'Boyle, P. Knijnenburg. Integrating loop and data transformations for global optimisation. In *IEEE PACT'98 International Conference on Parallel Architectures and Compilation Techniques*, 1998.

[Tan90] A. Tanenbaum. *Structured Computer Organization*, third edition, Prentice-Hall, Englewood Cliffs, 1990.

[War84] J. Warren. A hierarchical basis for reordering transformations. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, ACM, New York, 1991.

[Wol91] M.E. Wolf, M.S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, ACM, New York, 1991.

[Wol92] M.E. Wolf. *Improving Locality and Parallelism in Nested Loops*, Ph.D. thesis, Dept. of computer science, Stanford University, California, 1992.

[WoM86] M. Wolfe. Loop skewing : the wavefront method revisited. In *Int'l Journal of Parallel Programming*, Vol. 15, No. 4, 1986.

[WoM87] M. Wolfe. Iteration space tiling for memory hierarchies. In *Third SIAM Conference on Parallel Processing for Scientific Computing*, 1987.

[WoM89] M. Wolfe. More iteration space tiling. In *Super Computing'89*, 1989.