Optimizing VLIW Architectures for Multimedia Applications

Esther Salamí San Juan

A thesis submitted in fulfillment
of the requirements for the degree of
Doctor in Telecommunications Engineering

Advisor: Mateo Valero Cortés

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

March 2007

To my parents and my husband,
whose love and unconditional support
gave me the strength to carry out this work.

Acknowledgments

First of all, I would like to express my gratitude to my thesis advisor, Mateo Valero, for his invaluable guidance, confidence and support during all these years. I greatly appreciate the amount of time and energy he has devoted to this work despite all those pressing duties taking up his agenda. But most of all, I appreciate his enthusiasm for this thesis.

I would also like to thank Jesús Corbal for introducing me to the department and, particularly, to the multimedia group. His interest and his own work on multimedia processing clearly motivated the beginning of this thesis.

Thanks to Alicia Bustos and Àlex Ramírez for the good times we had in San Francisco. Àlex's comments and suggestions have been very helpful to my research.

Thanks to so many colleagues, friends, and relatives who have contributed to the ending of this thesis with their words of encouragement. I am specially grateful to Beatriz Otero for her friendship and lively support, and to Vanessa Moreno for so many coffees whenever I needed a break.

Finally, I also wish to thank the LCAC people and the administrative staff for their constant assistance, and the *Centre de Supercomputació de Catalunya* (CESCA) for supplying the computing resources for our research.

Abstract

The growing interest that multimedia processing has experimented during the last decade is motivating processor designers to reconsider which execution paradigms are the most appropriate for general-purpose processors. On the other hand, as the size of transistors decreases, power dissipation has become a relevant limitation to increases in the frequency of operation. Thus, the efficient exploitation of the different sources of parallelism is a key point to investigate in order to sustain the performance improvement rate of processors and face the growing requirements of future multimedia applications. We belief that a promising option arises from the combination of the *Very Long Instruction Word* (VLIW) and the *vector processing* paradigms together with other ways of exploiting coarser grain parallelism, such as *Chip MultiProcessing* (CMP).

As part of this thesis, we analyze the problem of memory disambiguation in multimedia applications, as it represents a serious restriction for exploiting *Instruction Level Parallelism* (ILP) in VLIW architectures. We state that the real handicap for memory disambiguation in multimedia is the extensive use of pointers and indirect references usually found in those codes, together with the limited static information available to the compiler on certain occasions. Based on the observation that the input and output multimedia streams are commonly disjointed memory regions, we propose and implement a memory disambiguation technique that dynamically analyzes the region domain of every load and store before entering a loop, evaluates whether or not the full loop is disambiguated and executes the corresponding loop version. This mechanism does not require any additional hardware or instructions and has negligible effects over compilation time and code size. The performance achieved is comparable to that of advanced interprocedural pointer analysis techniques, with considerably less software complexity. We also demonstrate that both techniques can be combined to improve performance.

In order to deal with the inherent *Data Level Parallelism* (DLP) of multimedia kernels without disrupting the existing core designs, major processor manufacturers have chosen to include MMX-like μSIMD extensions. By analyzing the scalability of the DLP and non-DLP regions of code separately in VLIW processors with μSIMD extensions, we observe that the performance of the overall application is dominated by the performance of the non-DLP regions, which in fact exhibit only modest amounts of ILP. As a result, the performance achieved by very wide issue

configurations does not compensate for the related cost. To exploit the DLP of the vector regions in a more efficient way, we propose enhancing the μSIMD-VLIW core with conventional vector processing capabilities. The combination of conventional and sub-word level vector processing results in a 2-dimensional extension that combines the best of each one, including a reduction in the number of operations, lower fetch bandwidth requirements, simplicity of the control unit, power efficiency, scalability, and support for multimedia specific features such as saturation or reduction. This enhancement has a minimal impact on the VLIW core and reaches more parallelism than wider issue μSIMD implementations at a lower cost. Similar proposals have been successfully evaluated for superscalar cores. In this thesis, we demonstrate that 2-dimensional Vector-μSIMD extensions are also effective with static scheduling, allowing for high-performance cost-effective implementations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents the motivations behind this thesis. An overview of the different sources of parallelism usually found in multimedia codes and the most significant trends in their exploitation is also included. The chapter ends up defining the main goals of this work.

1.1 Motivation

There has always been a lively interest in improving the interface between human and machines. In the course of time, advances in microprocessors technology and design have made possible thinking on more ambitious applications that offer a more comfortable and friendly environment to the user, either to aid in work, for personal tasks, or simply for entertainment. As a result, new forms of communication have emerged that integrate multiple information content and processing, including (but not limited to) text, audio, graphics, animation, video, and interactivity. Speech recognition, cryptography, video-conference, web-TV, or the new generation of video games are just a few examples of the great variety of this kind of applications, widely known as *multimedia applications*.

Processors had been traditionally designed for technical and scientific applications. At present, it is widely assumed that the multimedia workload dominates desktop cycles and that it will continue to increase in importance [KP98]. Multimedia workload has significantly different characteristics from other existing applications. Current computers have to face increasing requirements in computational power and memory bandwidth and it is not clear what kind of architecture deals better with present and future multimedia requirements.

During the last three decades, microprocessors have undergone an exceptional increase in performance. The number of transistors on an integrated circuit doubles every 18 months approximately, exceeding Moore's original statement [Moo65]. Furthermore, advances in microarchitecture design provide more aggressive techniques to exploit greater degrees of parallelism. As technology evolves, the number of tran-

sistors to be included on a single chip will continue increasing [Yu96]. Nevertheless, having more and faster transistors does not involve the same performance improvement rates than some years ago.

On the one hand, the available *Instruction Level Parallelism* is limited by the amount of dependences and conditional branches that exists in programs, hence taking little benefit from more aggressive processor implementations. On the other hand, the growing gap between processor speed and memory access time leads to a *memory wall* in which memory accesses dominate code performance [WM95]. Finally, as the size of transistors decreases, there is a significant increase in the concentration of heat, which can even make the chip burn. According to Intel, the power consumption of their chips has doubled approximately every 36 months [MNW+02]. Increasing power dissipation, and particularly, the need to cool regions of local power concentrations, also known as *hot spots*, has become a major problem.

The *Very Long Instruction Word* (VLIW) paradigm provides a promising alternative to traditional superscalar designs, as it requires considerably less hardware complexity, thus reducing power consumption. It has demonstrated to do well in the embedded media domain [Pur98, BLO02, FG00, Ses98, RS96]. Furthermore, in the general-purpose domain, the *Itanium Processor Family* [SA00] has recently arisen as a competitive option against commonly extended out-of-order superscalar processors. Nevertheless, a high degree of *Instruction Level Parallelism* in VLIW architectures still requires decoding more operations in parallel and a large register file, which may affect overall performance due to the increased access time.

Our work concentrates on improving VLIW architectures in the context of multimedia workload. As we will see in next section, the performance of this kind of applications can be improved by exploiting different sources of parallelism. In this thesis, we face two problems. First, we analyze the problem of memory disambiguation, as it imposes a significant restriction on the exploitation of *Instruction Level Parallelism*. Second, we study how to exploit the inherent *Data Level Parallelism* of multimedia applications in a cost effective way, reducing the fetch bandwidth and power requirements of very wide issue architectures.

1.2 Sources of Parallelism in Multimedia Applications

We can distinguish at least three forms of parallelism in multimedia applications: instruction level parallelism, data level parallelism, and thread level parallelism.

1.2.1 Instruction Level Parallelism

The *Instruction Level Parallelism* (ILP) paradigm speeds up execution by causing individual machine operations to execute in parallel [RF93]. The amount of ILP depends on each particular application. Video and imaging codes, for instance, exhibit

more ILP than cryptography applications. Nevertheless, multimedia workloads are in general characterized by larger amounts of ILP than integer ones.

Most of the traditional hardware and compilation techniques focus on exploiting ILP to speed-up execution. *Superscalar* processors are the most extended ILP implementation for the general-purpose domain. The hardware must determine at run-time the dependences between operations and decide at which particular time and on which functional unit and registers the operations must be executed (a detailed analysis of superscalar hardware can be found in [Joh91]). However, it is widely assumed that current superscalar processors cannot be scaled by simply fetching, decoding and issuing more instructions per cycle. Conditional branches, the instruction cache bandwidth, the instruction window size, the register file and the memory wall are some of the aspects that currently limit the scalability of superscalar processors.

Very Long Instruction Word (VLIW) processors are another form of exploiting ILP that requires less hardware complexity. Table 1.1 summarizes the main differences between superscalar and classic VLIW architectures. The compiler and not the hardware is responsible for identifying groups of independent operations, assigning a functional unit to each operation, and packaging them together into a single VLIW instruction [Fis81]. Due to the regularity of multimedia applications, static scheduling arises as a promising option over dynamic scheduling. The first generation of VLIW processors were successful in the scientific domain [CNO+88, RYYT89], and it has also been the architecture of choice for most media embedded processors [Sem99, Dev99, TI99]. However, some relevant facts, such as binary incompatibility across different implementations, the increased code size as a result of aggressive scheduling techniques, and the lack of flexibility in front of non-deterministic latencies, have contributed to the belief that VLIW processors are not appropriate for the general-purpose domain.

Superscalar	Classic VLIW
Requires dependency checking hardware	The compiler is responsible for grouping independent operations
Control logic does not scale well (O(n2))	Simplified hardware for decoding and issuing instructions
Requires routing hardware for assignment of instructions to functional units	Static assignment of operations to functional units
Hardware has full information about dependences	Limited static information available to the compiler
Flexibility in front of variable latency memory operations	Impact of non-deterministic latencies
	Increased code size
Binary compatibility across different implementations	Object code incompatibility across different implementations

Table 1.1. Comparison between superscalar an VLIW architectures

During the last decades, there has been considerable advances regarding these issues and, at present, a revival of the VLIW execution paradigm is observed. The IBM's tree-based VLIW architecture, for example, provides binary compatibility for VLIW implementations of varying width through dynamic binary translation [EFK+98]. Furthermore, each company has developed its own compression scheme to avoid code expansion. The Philips' *Trimedia* architecture [RS96], for example, stores the instructions in a compressed format, and a decompressor unit expands it during the instruction fetch. In the Texas Instruments' *VelociTI* [Ses98], the fetch packets are delimited by parallel instruction link bits in the instruction format.

On the other hand, HP and Intel have recently introduced a new style of architecture named *Explicitly Parallel Instruction Computing* (EPIC) [SR00] (also called *independence architecture* [RF93]). The compiler determines the grouping of independent instructions and communicates this via explicit information in the instruction set, but the hardware makes the final decision of which operations execute on each functional unit at run-time [Smo02]; hence EPIC retains compatibility across different implementations without the complexity of superscalar control logic. The specific instruction set architecture, known either as IA-64 or as *Itanium Processor Family* (IPF) [SA00], includes a large number of registers, predicated execution to reduce control hazards, unbundled branches support, compiler control of the memory hierarchy, and speculative loads support.

Table 1.2 summarizes the evolution of the IPF. The first implementation of the IA-64, the *Itanium* processor (code-named *Merced*), was released in 2001, two years later than originally expected. It was offered at speeds of 733 and 800 MHz, with a choice of 2 or 4 MB off-die L3 cache. Although it was the fastest floating point processor in the market, it was not commercially successful mainly because of the launch delay, the lack of optimized code, and its low performance when running IA-32 applications, among other reasons. Hence, it was replaced in 2002 by the *Itanium2* processor, which is intended for use in high-end enterprise servers. In the first version of the *Itanium2* processor (code-named *McKinley*), Intel shortened the pipeline from ten to eight stages, tripled the system bus bandwidth and moved the L3 cache onto the chip. The *Itanium2* processor can issue up to six operations per cycle in a fixed set of combinations. It includes 128 floating point, 128 integer, 64 predicate and 8 branch registers. As far as functional units, it has six integer, three branch, two floating point, one SIMD, two load, and two store units. In July 2006, Intel released the first dual-core *Itanium2* processor (code-named *Montecito*). Intel reports that it doubles the performance of its single-core predecessor, while reducing power consumption by approximately 20 percent [Int06]. It also features multithreading capabilities, being able to execute two threads per core. From the available information about coming generations, we can envision that future implementations of the IA-64 will relay on multi-core chips, even having as many as 16 cores on the chip die.

Version Processor Date	Clock Speed	Bus Speed Bandwidth	L1 Instr/Data L2 Cache L3 Cache	Technology Transistors Die size Power envelope
Merced Itanium 07/2001	733 or 800 MHz	133 MHz DDR 2.1 GB/s	16 KB / 16 KB 96 KB 2 MB or 4 MB off-die	180 nm 25 (+295) M 300 nm2 116-130 W
McKinley Itanium2 07/2002	900 MHz or 1 GHz	100 MHz QDR 6.4 GB/s	16 KB / 16 KB 256 KB 1.5 MB or 3 MB on-die	180 nm 221 M 421 nm2 90-100 W
Madison Itanium2 06/2003 -07/2005	1.3 to 1.67 GHz	100 MHz QDR 6.4 GB/s	16 KB / 16 KB 256 KB 1.5 MB to 9 MB on-die	130 nm 410-592 M 374-432 nm2 91-130 W
Deerfield Itanium2 08/2003	1 GHz	100 MHz QDR 6.4 GB/s	16 KB / 16 KB 256 KB 1.5 MB on-die	130 nm 221 M 421 nm2 62 W
Fanwood Itanium2 11/2004	1.3 or 1.6 GHz	100 or 133 MHz QDR 6.4 or 8.5 GB/s	16 KB / 16 KB 256 KB 3 MB on-die	130 nm 410 M 374 nm2 99 W
Montecito Itanium2S Dual Core 07/2006	1.4 to 1.67 GHz	100 to 166 MHz QDR 6.4 to 10.6 GB/s	32 KB / 32 KB 2.5 MB 8 to 24 MB on-die	90 nm 1720 M 596 nm2 104 W

Table 1.2. Evolution of the *Itanium Processor Family*

1.2.2 Data Level Parallelism

Another kind of parallelism that can be found in programs is *Data Level Parallelism* (DLP) (or *Single Instruction Multiple Data* (SIMD) [Fly72]). The DLP paradigm tries to specify with a single vector instruction a large number of operations to be performed on independent data elements. As each individual operation is independent of all others, vector instructions are highly parallel and pipelineable, which simplifies the control unit considerably.

One of the main advantages of using vector instructions is the reduction in the overall number of instructions to be executed, as one single vector instruction specifies several scalar instructions. Furthermore, many control operations are also removed, as they are embedded in the semantics of the vector instruction [QEV98]. As a result, the pressure on the fetch unit diminishes significantly.

Other advantages are related to the way the memory system is accessed. As a single vector memory instruction specifies a long sequence of memory addresses, the hardware has advance knowledge regarding memory references. This information can be used to improve the memory system [VLPA95]. Additionally, a vector instruction is able to amortize the start-up latencies of functional units and memory over a potentially long stream of elements.

In the supercomputing domain, DLP has been successfully exploited by vector [Rus78, BS00, vdSD01] and array [Hor82, Red73] processors. During the last decade, the increasing significance of media processing has motivated a great interest in exploiting sub-word level parallelism (also called μSIMD parallelism [Lee99]). DLP is commonly present in multimedia applications in the form of small loops that operate streams of small data elements, such as pixels or audio samples. In the general-purpose domain, μSIMD multimedia extensions such as *SSE* [Int99] or *Altivec* [NJ99] have been a fast and cost effective option to deal with this kind of parallelism: short data are packed into a single register and operations are carried out simultaneously on the different register elements. However, the efficiency of sub-word level implementations is reduced by the effect of unaligned and non-unit stride memory accesses. On the other hand, while traditional vector processors can be easily scaled by just replicating the functional units and widening the paths to the vector registers (with just the limit of the maximum vector length), the scalability of sub-word level implementations is limited by the width of the μSIMD registers.

A third way of exploiting DLP comes from the combination of both traditional vector and sub-word level parallelism [CEV99, JVTW01, Koz99]. These architectures adapt to typical multimedia patterns by extending the scope of vectorization to two dimensions. They overcome some of the limitations of sub-word level implementations and yield better performance than scaling the word size of a sub-word level parallel architecture [SAS+05]

1.2.3 Thread Level Parallelism

As the gap between processor operation frequency and memory access time increases, ILP techniques become insufficient to tolerate memory latency. The hardware complexity and power cost of the structures needed to keep the processor busy during a cache miss are prohibitive. In consequence, there is a growing trend towards exploiting higher levels of parallelism, such as *Thread Level Parallelism* (TLP). A program exhibit TLP if it can be decomposed in different threads, or groups of instructions, that can be executed concurrently. This kind of parallelism is commonly found in commercial server applications, such as databases.

Future media applications are expected to process several media streams concurrently, such as video, audio and encryption, which are controlled by a higher layer of the protocol. We can find an example in the MPEG4 standard [Koe99], an object-based approach to describe and compose interactive audiovisual scenes. Uncorrelated objects are coded, encrypted and transmitted separately in order to be composed

again at reception. These objects may include digital video, still image, audio, speech and even audio synthesis or 3D-graphics. Dealing with multiple concurrent media streams means that we have high levels of coarse level parallelism together with the intra-threaded real time requirements of each media source.

One of the main techniques to exploit TLP is called *simultaneous multithreading* (SMT) [TEL95]. In SMT, instructions from multiple threads can be issued in one processor cycle. The first commercial SMT processor was the Alpha *21464* (*EV8*) [Eme99]. Although the processor was never released, the technology developed for this processor did probably set the bases for later processor designs. The Intel *Pentium 4* [BBH+04] was the first desktop processor to implement SMT (*Hyper-Threading Technology* (HTT) in Intel's terminology).

On the other hand, *interleaved multithreading* consists on issuing multiple instructions from different threads on an interleaved way. We can distinguish different levels of multithreading depending on the frequency of the interleaving. In fine-grain multithreading, for example, instructions from different threads are issued after every cycle. On the contrary, coarse-grain multithreading switches from one thread to another when the current executing thread causes some long latency event.

Another implementation of TLP is *chip multiprocessing* (CMP). It integrates two or more processor cores into one chip, so that different threads can be executed independently. The main manufacturers of high performance processors are following this trend [TDJ+02, SKT+05, Joh05, KAO05, AMD06, GMNR06, MB04, KDH+05]. Nevertheless, different TLP implementations are not exclusive and can be combined to improve performance. Intel's *Montecito* and Sun's *UltraSPARC T1* are examples of coarse-grain multithreading multi-core processors.

1.3 Thesis Overview

1.3.1 Objectives

We can distinguish two main objectives in this thesis. As we will demonstrate, memory disambiguation is a key optimization to exploit ILP, specially in static scheduling implementations, such as classic VLIW architectures. Furthermore, memory disambiguation is also required in order to generate vector code. Even in the case of having hard-to-deal control and data dependences in the computation, typical media kernels usually process disjointed streams of data; nevertheless, common commercial compilers fail to disambiguate them mainly because of the extensive use of pointers and indirect references. One of the main goals of this thesis is to *analyze the problem of memory disambiguation in multimedia codes*. As part of this thesis, we will propose, implement, and evaluate a software memory disambiguation technique based on the memory access patterns of most media kernels.

On the other hand, we think that the combination of the vector and the VLIW paradigms is a promising alternative to exploit the fine-grain parallelism of multimedia codes. Hence, the second main goal of this work is to *evaluate the potential of enhancing a reference μSIMD-VLIW architecture with conventional vector capabilities*. We will show that multimedia applications are composed of heterogeneous regions of code, some of them with high levels of DLP and other ones with only modest amounts of ILP. Vector-μSIMD multimedia extensions have proved to be a good option to exploit the parallelism of the DLP-regions [Cor02], as they adapt well to typical multimedia data structures, providing good performance and overcoming the scaling limitations of existing μSIMD extensions. Furthermore, simplicity and power efficiency are features of both, vector and VLIW architectures, which allows for lower clock rates and lower voltages. We will demonstrate that Vector-μSIMD extensions are also effective with static scheduling, allowing for high-performance cost-effective implementations. Additionally, TLP implementations, such as chip-multiprocessors can be used to exploit coarse-grain parallelism.

1.3.2 Organization of this Document

In this chapter we have exposed the motivation and the main objectives behind this thesis. The rest of this document is organized as follows. Chapter 2 surveys the main implications that multimedia processing is involving in computer architecture and overviews the most significant processor architectures that have been proposed for multimedia.

The working environment is presented in Chapter 3, including the compilation and simulation framework, the extensions built into the original tool set, and the reference architecture used in the evaluations. Next, Chapter 4 analyze the main characteristics of multimedia codes, both at the application and at the loop level. Our set of benchmarks is introduced and characterized for the reference VLIW architecture.

Chapter 5 discusses the problem of memory disambiguation in the context of multimedia codes and proposes a dynamic memory disambiguation technique specially targeted at multimedia loops or any other applications with similar memory access patterns. The proposal is fully described, implemented and evaluated, as well as compared against advanced interprocedural pointer analysis.

Chapter 6 is concerned with our proposal of adding vector capabilities to μSIMD-VLIW processors. We start by performing a scalability study of the DLP and non-DLP regions of the benchmarks in VLIW architectures with μSIMD multimedia extensions. Next, we present the proposed architecture and discuss the main compilation issues. The chapter ends with a performance evaluation of the architecture.

Finally, Chapter 7 concludes the thesis by summarizing the achieved goals and suggesting new directions for future research.

Chapter 2

Processor Architectures for Multimedia

Multimedia processing has motivated strong changes in the focus and design of processors. Current computers have to face increasing requirements in computational power for videoconferencing, image compression and processing, 3D graphic games, encryption, speech recognition and so on. In this chapter, we overview the impact that multimedia processing is having on computer architecture and briefly describe some of the most relevant proposed architectures.

2.1 Architectural Challenges

The importance of multimedia processing has produced a revolution in the design of both embedded and general-purpose processors. In the general-purpose domain, these changes have been very straightforward with the inclusion of MMX-like μSIMD multimedia extensions. These extensions have become the most important change to the basic ISA since the inclusion of the FP units inside the processor core. Nevertheless, the significance that media processing has been taking on during the last years has not been limited to the general-purpose domain. On the contrary, the embedded domain has experimented a revolution based on new and harder demands. Near future applications such as personal mobile computing, Web-TV devices, DVD players or even next generation of game consoles are just a few examples.

Traditional *Digital Signal Processors* (DSPs) were designed to support specific and regular computation-intensive tasks. Most of them included special-purpose operations, complex memory addressing modes, and support for counted loops, among other features. However, such levels of specialization limit the use of high-performance compilers and lack flexibility enough to adapt to variations in the applications. During the last decade, thanks to advances in technology and compilation techniques, and motivated by the evolution of the multimedia market, DSP processors have experimented a change of trend towards simpler and more general load-store RISC-

like architectures. Most of them include μSIMD operations, support for unaligned memory accesses and prefetching, and DMA transfers.

To satisfy the great variety of consumer products, these processors must provide high performance at low cost. At the same time, they must be programmable in order to support the different standards and reduce application development time. Therefore, these processing elements are limited by the trade-offs between performance and flexibility. The increasing importance of these emerging class of processors has deserve its own term: the *media processor*. A media processor is defined as a programmable processor dedicated to simultaneously accelerating the processing of multiple data types, including digital video, digital audio, computer animation, text, and graphics [Kon98].

According to this, Fritts distinguishes three forms of industry support for multimedia: application-specific processors, multimedia extensions to general-purpose processors, and media processors [Fri00]. A similar classification is done by Talla, who distinguishes between general-purpose processors with SIMD extensions, VLIW media processors, and application specific integrated circuits (ASICs) [Tal01]. On the other hand, Dasu proposes a complete categorization of existing microprocessors based on both the evolution of processing architectures and the functionality of the processors (see Figure 2.1) [DP02]. While from an evolution point of view special-purpose programmable processors assimilate features of DSP and RISC architectures, from a functional perspective they are including VLIW and SIMD implementations to exploit parallelism at many levels.

From another perspective, El-Mahdy proposes a taxonomy of multimedia processing based on three architecture models: vector processors, superscalar processors, and multiprocessors [EM01]. DSPs and multimedia approaches are considered as variations of these three architecture models. As we are interested on the architectural point of view, we have also organized the different approaches on three architectural groups: VLIW processors, vector processing, and chip multiprocessors.

The VLIW paradigm has been the architecture of choice for most media processors. Chromatic Research's *Mpact* [Pur98], Equator's *MAP-CA* [BLO02], Analog Devices' *TigerSHARC* [FG00], Texas Instruments' *VelociTI* [Ses98], and Philips' *TriMedia* [RS96] are just a few examples. These architectures rely on the compiler to avoid the overhead of run-time parallelism extraction and become a cost-effective option to provide more flexibility to support the large variety of multimedia standards.

From the supercomputing domain, the vector and systolic paradigms have also influenced new DSP processors. Examples of vector microprocessor designs are the *Torrent-0* [ABI+95] and the *VIRAM* project [KP98]. Additionally, there are projects using streaming SIMD architectures to address 3D graphics processing, such as the *Imagine* processor [RDK+98]. Another research line considers the inclusion of a conventional vector ISA extension [QCEV99] and a matrix ISA extension [CEV99] into a superscalar core.

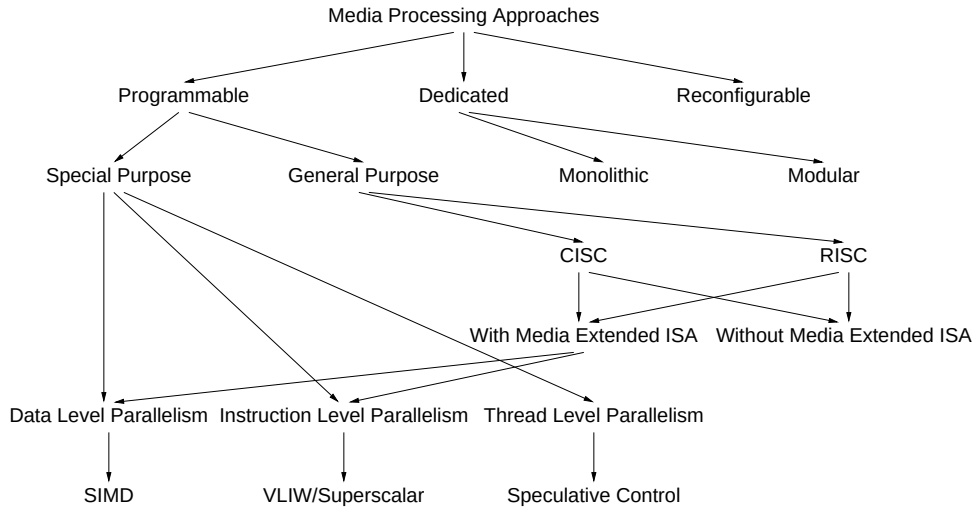


Figure 2.1. Hybrid classification of microprocessors [DP02]

Finally, a natural way of exploiting coarse grain parallelism consists on integrating multiple processor cores into a single chip. In fact, the main manufacturers of high performance processors are following this trend: see for example IBM's *Power5* [SKT+05], HP's *PA-8900* [Joh05], SUN's *UltraSPARC T1* [KAO05], AMD's *Opteron* [AMD06], Intel's *Code Duo* [GMNR06] and *Montecito* [MB04], and the *Cell Broadband Engine* [KDH+05] from Sony, Toshiba and IBM. *Chip MultiProcessors* (CMPs) have the potential to provide high scalability, although they are still limited by the lack of programming tools and their dependency on hand-written libraries [Kon98]. In particular, the combination of the CMP, the VLIW, and the SIMD paradigms appears as a good option to exploit the heterogeneous parallelism found in multimedia applications, being able to provide high performance at low cost. Typical examples of VLIW CMPs are SUN's *MAJC* [Gwe99], Improvisys' *JAZZ* [Imp01], BOPS' *ManArray* [PP99], and HP's *LX* [FBF+00].

2.2 VLIW Processors

As stated before, the VLIW execution paradigm arises as a good candidate to deal with the regular patterns found in multimedia applications. Next, we describe two of the most representative examples of VLIW architectures for multimedia: Texas Instruments' *VelociTI* and Philips' *TriMedia*.

VelociTI

VelociTI [Ses98] is a load-store RISC-like VLIW architecture suitable for multichannel vocoding for telephony and wireless, modems, imaging, and high performance systems in communications and multimedia. It focuses on minimizing design com-

plexity to allow the development of a high performance compiler, with the objective of increasing performance and reducing application development time.

The first implementation of the architecture, the fixed-point TMS320C62x family, has eight independent units, including two multipliers and six ALUs. The TMS320C7x adds floating-point capability to six of the eight units. The processor core is divided into two identical datapaths with four functional units and 16 32-bit registers each. Up to eight operations can be packed into one single VLIW instruction. The instruction set provides saturation and normalize operations, but it does not include μSIMD operations. Almost every operation can be guarded by a predicate register. The TMS320C6201 memory architecture includes 64 Kbytes of on-chip program memory configurable either as mapped memory or as direct mapped cache, 64 Kbytes of interleaved data memory, a DMA controller, and an external memory interface.

The compiler includes classical optimizations such as control-flow simplification, copy propagation, common subexpression elimination, loop-invariant code motion, and so on. In addition, it also performs software pipelining, if-conversion, memory address cloning to allow vectorization and unrolling, memory address-dependence elimination, and memory-bank disambiguation to avoid memory-bank conflicts.

TriMedia

TriMedia is a programmable high-performance VLIW family of processors specially designed for real-time processing of video, audio, graphics and communication data streams. Backward source code compatibility is ensured between the different members of the *TriMedia* family. Nevertheless, the codes must be re-compiled, as binary compatibility is not guaranteed. Unlike the *VelociTI* architecture, it integrates multimedia specific co-processors and μSIMD extensions.

The first implementation of the architecture, the TM1000 [RS96], has 27 functional units and 128 32-bit registers. Up to five operations can be scheduled in parallel into a single VLIW instruction. The instruction set contains load/store operations, arithmetical and logical operations, floating point operations, and μSIMD operations, including special operations to perform convolution and distance computation. The architecture also provides support for guarded execution. The memory architecture includes 32 Kbytes of on-chip instruction cache and 16 Kbytes of on-chip data cache. Two memory requests can be served in parallel provided that they access different banks, but a stall cycle is imposed otherwise. The chip also incorporates two co-processors, an Image co-processor and a Variable Length Decoder co-processor, video input and output, digital audio input and output, and two serial interfaces.

One successor to the *TriMedia* TM1000 is the *TriMedia* CPU64 [vESV+99] architecture, which is targeted for embedded use in electronic devices such as digital televisions and set-top boxes. Improvements over the TM1000 include the extension of the wordsize from 32 to 64 bits and the extension of the instruction set with a

large set of multimedia operations. The data cache maintains the 16 Kbytes size, but changes to a true dual-port design, thus allowing two memory requests to be served simultaneously even if they access the same memory bank.

The latest *TriMedia* processor, the TM3270 [vdWVD+05], is designed to address the performance demands of standard definition video processing. It is typically used as an embedded processor in a *System-on-a-Chip* (SoC). Table 2.1 summarizes the main parameters of the architecture. It must be noted that the data cache has been enlarged up to 128 Kbytes and supports penalty-free non-aligned accesses.

Architecture	5 issue slot VLIW, guarded RISC-like operations
Pipeline depth	7-12 stages
Address width	32 bits
Data width	32 bits
Register file	Unified, 128 32-bit registers
Functional units	31
IEEE-754 floating point	Yes
SIMD capabilities	1 x 32-bit, 2 x 16-bit, 4 x 8-bit
Instruction cache	64 Kbyte, 128-byte lines, 8 way set-associative, LRU replacement
Data cache	128 Kbyte, 128-byte lines, 4 way set-associative, LRU replacement, allocate-on-write miss policy

Table 2.1. Parameters of the TM3270 architecture [vdWVD+05]

One of the main improvements of the TM3270 over previous *TriMedia* processors is the extension of the instruction set with a significant number of new instructions specially targeted to improve performance in video processing kernels. One of these enhancements is the inclusion of *two-slot operations*, that is operations which are executed by two functional units, thus allowing up to four source operands and up to two destination operands. It also includes *collapsed load operations with interpolation* on the retrieved data, specially suitable to reduce the computational complexity of the motion estimation algorithm. Additionally, there are also specific *CABAC operations* for the Context-Based Adaptive Binary Arithmetic Coding (CABAC) algorithm of the H.264/AVC video standard. Finally, it also provides *memory region based prefetching*, which is specially effective for block-based image processing.

2.3 Vector Processing

Vector architectures have traditionally been the most successful way of exploiting DLP in the supercomputing domain for scientific and engineering tasks. As they allow for low-cost implementations, vector architectures also appear as a good alternative to deal with the new computation intensive tasks of multimedia applications.

There are several proposals based on the vector model, ranging from cost-effective implementations of conventional vector processors to stream or n-dimensional vector alternatives. In this section we briefly describe some of the most relevant ones. We

have classified them into four different groups: conventional vector architectures, μSIMD extensions, n-dimensional vector architectures, and stream processors.

2.3.1 Conventional Vector Architectures

Cost-effective implementations of conventional vector microprocessors try to adapt to multimedia data patterns mainly by reducing the maximum vector length and adding sub-word level processing capabilities. Two representative examples of this kind of processors are the *Torrent-0* and the *VIRAM*.

Torrent-0

Torrent-0 (T0) [ABI+95] is a single-chip fixed-point vector microprocessor designed for multimedia, human-interface, neural network, and other digital signal processing tasks. The first use of *T0* was as the core of the *Synthetic Perceptron Testbed II* (SPERT-II) workstation accelerator board [WAK+96], originally designed to accelerate multiparameter neural network training for speech recognition research.

The *T0* architecture consist of a MIPS-II compatible 32-bit integer RISC core, an on-chip 1 KB instruction cache, a high performance fixed-point vector unit co-processor, a 128-bit wide external memory interface, and a byte-serial host interface. The vector unit includes a vector register file, two vector arithmetic functional units, and one vector memory unit. The vector register file contains 16 vector registers of 32 32-bit elements each. The vector arithmetic functional units perform integer arithmetic and logic operations and vector fixed-point operations that include scaling, rounding, and result saturation. Finally, the vector memory unit performs scalar memory operations, vector memory operations, and vector editing operations, and provides support for unit-stride, constant-stride, and indexed addressing modes. As there is only one memory address port, non-unit stride and indexed memory accesses are served at one element transfer per cycle.

All three vector functional units consist of 8 parallel pipelines, with the elements of a vector register striped across them. A vector functional unit accepts a new instruction with a maximum vector length of 32 every four cycles. The *T0* is able to dispatch one 32-bit instruction per cycle to each vector functional units in turn, thus sustaining up to 24 operations per cycle. All vector pipeline hazards are interlocked in hardware.

VIRAM

The *Vector IRAM* (VIRAM) [Koz99] is a vector architecture that combines vector processing with the the concept of *Intelligent RAM* (IRAM), that is the integration of logic an DRAM on a single chip. It was specially designed to match the requirements of the mobile personal environment.

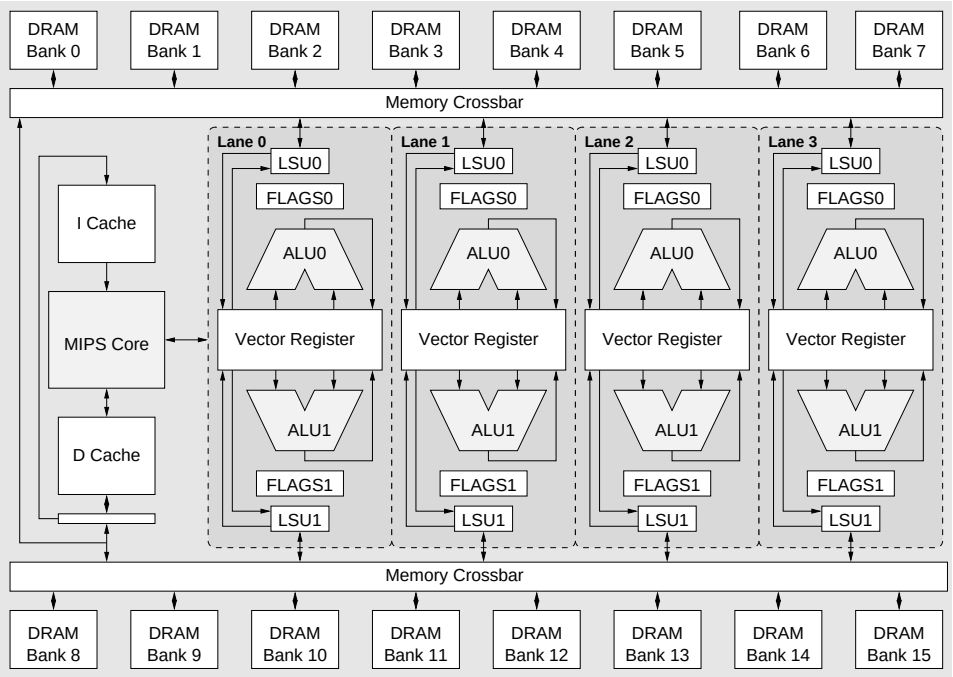


Figure 2.2. Architecture of the *VIRAM* vector processor

Figure 2.2 shows the architecture of the *VIRAM*. It consists of a scalar unit, a vector co-processor, and a network interface, all connected to the on-chip memory system. The scalar unit is based on an in-order, dual-issue superscalar MIPS processor and includes 16 KB instruction and data caches.

The vector unit has six vector functional units: two arithmetic, two flag processing, and two load/store units. It provides support for multimedia data types, short vectors, and other DSP features such as scaling, rounding and saturation. A special bypassing path is also implemented to manage reductions. The vector unit is clusterized into four parallel lanes. The vector register file holds 32 vector registers of 32 64-bit elements each, with the elements of the vector registers distributed along the different lanes. Additionally, vector registers can be subdivided to hold 64 32-bit elements or 128 16-bit elements in order to exploit sub-word level parallelism.

The main memory of *VIRAM* is based on embedded DRAM, which provides high memory bandwidth and low energy consumption, but at the cost of higher memory latency. In order to tolerate the high DRAM latency, the vector pipeline is modified to include the worst case memory access latency. Both memory units support unit-stride memory accesses, but only one can perform strided and indexed operations. Vector memory accesses are not cached, but coherence is maintained between scalar cache and vector accesses.

2.3.2 μSIMD Extensions

Starting in 1994 with the HP's MAX [Lee95] instruction set, and closely followed by SUN's *VIS* [TONL96], MIPS's *MDMX* [SIG97], and Intel's *MMX* [PW96], multimedia extensions have become essential on any general-purpose processor. They appeared with the objective of accelerating the execution of the emerging multimedia kernels while trying to minimize the impact on the overall processor design.

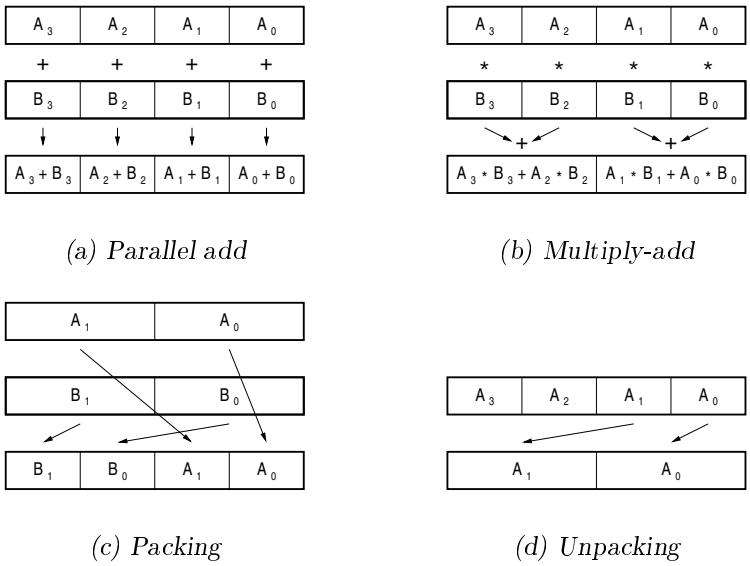
Based on the observation that multimedia applications use to spend a lot of time in loops that process streams of small data types (typically 8 or 16 bits), these ISA extensions exploit SIMD parallelism by packing several elements into a single register and operating simultaneously on the different register elements. In order to differentiate it from traditional SIMD execution, where a vector register is composed by a set of registers but there is only one element per register, some authors call it *microSIMD* (or μSIMD) execution [Lee99].

Initially, most μSIMD extensions included only integer capability. Additionally, to take advantage of the already existing register files, the floating-point register file was typically used to map the new set of μSIMD registers, thus limiting the register width to 64-bit. These μSIMD extensions provide the capacity to operate over two 32-bit, four 16-bit, or eight 8-bit elements in parallel. In the course of time, the increasing significance of the 3D processing domain drove to the inclusion of floating-point μSIMD instructions. Next multimedia extensions, such as AMD's *3DNow!* [AMD00], Motorola's *Altivec* [NJ99], and Intel's *SSE* [Int99], included 32-bit floating-point μSIMD arithmetic and a dedicated register file. Additionally, both *Altivec* and *SSE* are implemented in 128-bit. A summary of the main characteristics of available μSIMD multimedia extensions is given in Table 2.2

Year	Name	Company	Processor	Datapath	Registers	Instructions	FP
1995	Max	HP	PA RISC	64-bit	32 (Int)	8	No
1995	VIS	Sun	Ultra Sparc	64-bit	32 (FP)	121	No
1997	MDMX	MIPS	R1000/PA8000	64-bit	32 (FP)	74	Yes
1997	MMX	Intel	Pentium II	64-bit	8 (FP)	57	No
1999	3DNow!	AMD	K6-2	64-bit	8	24	Yes
1999	Altivec	Motorola	MPC7400	128-bit	32	162	Yes
1999	SSE	Intel	Pentium III	128-bit	8	70	Yes
2000	SSE2	Intel	Pentium 4	128-bit	8	144	Yes
2004	SSE3	Intel	Pentium 4	128-bit	8	157	Yes
2006	SSSE3	Intel	Xeon, Core 2	128-bit	8	173	Yes

Table 2.2. μSIMD multimedia extensions

The extended ISA generally contains a full set of vector instructions, including multiply-add operations, special multimedia instructions such as the sum of absolute differences, and instructions for data reorganization such as packing and unpacking.



(a) Parallel add *(b) Multiply-add*

(c) Packing *(d) Unpacking*

Figure 2.3. Examples of μSIMD instructions

They also provide support for scaling, rounding and saturation. Figure 2.3 shows some examples of common μSIMD instructions.

Nevertheless, the efficiency of this kind of μSIMD extensions is greatly reduced by the overhead to pack/unpack data to/from the μSIMD registers, the effect of unaligned and non-unit stride memory accesses, and the mismatch between the storage and computational formats. It must also be noted that the amount of parallelism that can be exploited is limited by the width of the μSIMD registers. Furthermore, even though there has been a great effort working into compilation techniques, hand optimization is still need to produce efficient μSIMD code.

2.3.3 N-dimensional Vector Architectures

To overcome some of the above mentioned limitations of μSIMD extensions, several approaches try to exploit two or more dimensions of parallelism to adapt to common multimedia data structures in a more efficient way. *MOM*, *CSI*, and *MediaBreeze* are examples of N-dimensional vector architectures.

MOM

The *Matrix Oriented Multimedia* (MOM) extension [CEV99] combines the intra-word parallelism capabilities of μSIMD extensions together with the inter-word parallelism exploitation of traditional vector architectures. Basically, it can be seen as a conventional short vector ISA where each vector sub-operation is a μSIMD one.

The proposed architecture consists of a superscalar core with the addition of a multimedia unit with its own register file. It offers 16 *MOM* registers of 16 64-bit words each to the programmer, vector load and vector store instructions to move data between memory and the *MOM* registers, and a set of computation instructions that operate on *MOM* registers. A *MOM* implementation executes as many μSIMD operations per cycle as the number of parallel lanes in the *MOM* functional unit. Furthermore, the architecture includes two 192-bit packed accumulators to handle reductions. Additional details about the *MOM* extension are given in Chapter 6.

A related proposal but targeting high performance for technical, scientific, and bioinformatics workloads is *Tarantula* [EAE+02]. It is based on adding aggressive vector capabilities to the EV8 processor. It includes two vector units with 16 parallel lanes each, allowing up to 32 double-precision operations per cycle. Vector memory accesses are performed directly to the second level cache, which is able to serve up to 16 words per cycle.

CSI

Complex Streamed Instruction (CSI) [JVTW01] is a memory-to-memory architecture for two-dimensional data streams of arbitrary length. Each stream is specified by six 32-bit stream control registers, containing information which includes the base address, the stream length, the strides in the two dimensions, the size of the stream elements, the scale factor, and the sign and saturation features.

The number of elements is not explicitly codified in the program, instead the hardware is responsible for dividing the data streams into sections which are processed in parallel. Data conversion and rearrangement is pipelined with computation and it is also performed by hardware, thus minimizing the packing/unpacking overhead typical of multimedia extensions. It also includes hardware support for data alignment and loop control.

One of the main differences between *CSI* and *MOM* is that *CSI* allows any stride in both dimensions, while *MOM* allows an arbitrary stride between consecutive rows, but not between consecutive elements inside one row.

MediaBreeze

The *MediaBreeze* [TJ01] architecture was designed to accelerate μSIMD codes by decoupling the true computation from the related overhead instructions, and providing explicit hardware support for processing the overhead instructions, including memory access, addressing arithmetic, loop branches and data reorganization (permute, pack, unpack, and transpose).

In the *MediaBreeze* architecture, the *Breeze unit* fetches and reorganizes input data and transfers them to the input queues in the *Data Station*, which acts as the register file for SIMD computation and is implemented as a set of FIFOs. A conventional

μSIMD unit performs computation and stores back the resulting stream on the output queue of the Data Station.

The Breeze unit is controlled by means of a special multidimensional instruction, called the *Breeze instruction*. This instruction describes the semantics of up to five nested loops and the architecture allows for up to three input and one output data structures. Thus, up to three 5-dimensional input streams can be operated to produce one 5-dimensional output stream. Information specified in the instruction includes the five loop index counts, the start address, stride, multicast and data types of each stream, the operation code, and the sign, saturation and scaling features of the result. Such a complex instruction requires a specific instruction memory to be hold and a specific decoder block inside the Breeze unit.

2.3.4 Stream Processors

The *stream programming model* tries to separate the description of data from the computation. Applications are coded as streams of data and a set of computation kernels that process them. These architectures are usually integrated as a co-processor into a SoC. Examples of stream architectures are *Imagine* from the Standford research group, Sony's *Emotion Engine* and Motorola's *RSVP*.

Imagine

Imagine [RDK+98] is a programmable load/store architecture for one-dimensional streams. It is specially suitable for applications performing many operations on each element in a long stream, such as image processing and 3D rendering

Imagine is organized around a large *stream register file* of 64 KB (see Figure 2.4). The unit of work is the *stream descriptor*, that specifies the base address in the stream register file, the stream length, and the record size of data elements in the stream. The architecture provides load/store operations to move entire streams of data between memory and the stream register file. The memory system consists of four independent SDRAM banks and is able to perform up to two simultaneous stream memory transfers. It provides support for sequential, constant-stride, indexed, and bit-reversed addressing modes. A single micro-controlled handles 8 arithmetic clusters with 6 functional units each (three adders, two multipliers and one divide/square root unit). The arithmetic clusters work in parallel on different elements of the stream and each cluster operate under VLIW control. Intermediate results are kept local to each cluster.

Applications are written in high-level language using a set of library functions and are executed on the host processor. Kernels are written in *Imagine*'s microassembly language using C-like expressions. The kernel compiler applies common high level optimizations such as loop unrolling, iterative copy propagation, and dead code elimination, and generates VLIW microcode instructions that control the arithmetic cluster.

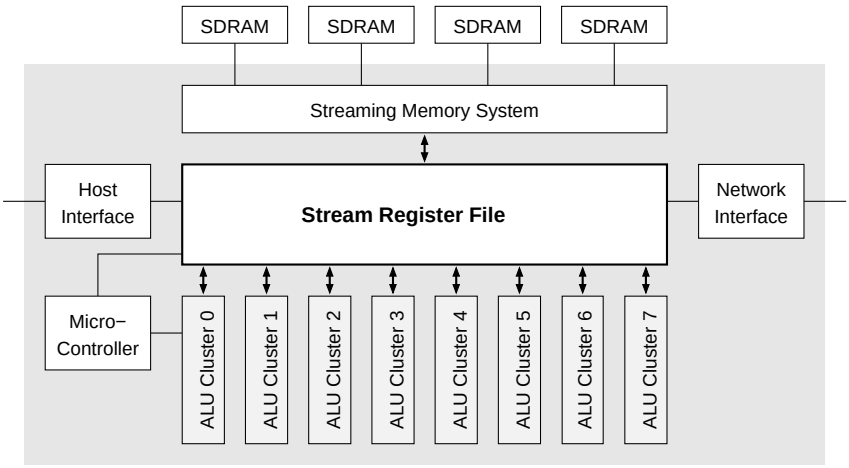


Figure 2.4. Architecture of the *Imagine* stream processor [RDK+98]

Emotion Engine

The *Emotion Engine* [KIea00] is the core of the Sony's *PlayStation 2* video game consoles. It was jointly designed by Toshiba and Sony to support high-quality 3D graphics, especially geometry and perspective transformations. It is basically a 2-way MIPS core with 128-bit μSIMD extensions and 2 vector co-processors connected via a shared 128-bit internal bus.

Each vector unit include four parallel floating-point multiply-accumulate units and a high-speed floating-point division unit, and can operate as a stand-alone 2-way VLIW processor. One of the vector units is mainly used to execute flexible calculations, such as characters movement, in collaboration with the CPU core. The second one has four times more memory than the other one, as it is mainly used as stand-alone processor responsible for conventional 3D graphics calculations, such as processing the background objects of the scene.

RSVP

The *Reconfigurable Streaming Vector Processor* (RSVP) [CEL+03] is a streaming vector co-processor architecture targeted to image and video capture devices and portable computation and communication devices, including handwriting recognition, voice recognition and synthesis, and graphics.

The *RSVP* architecture consists of operand access units, called *vector stream units* (VSUs), which communicate with the processing units via interlocked FIFO queues (see Figure 2.5). Thus, it achieves to decouple and overlap data access and data processing. The number of input and output VSUs depend on the particular implementation, but are defined by the architecture to be between 3 and 64 for input

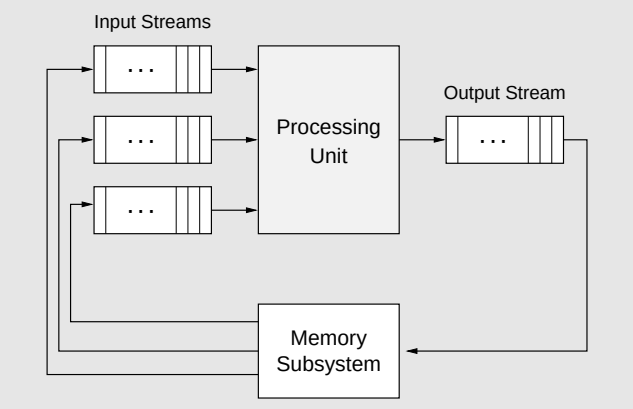


Figure 2.5. Architecture of the *RSVP* [CEL+03]

VSUs and between 1 and 64 for the output ones. It also defines between 2 and 64 64-bit accumulators and between 16 and 64 32-bit scalar registers.

Programming the *RSVP* consists of describing the input and output vectors and scalar values, and describing the computation itself as a data-flow graph. Conditional branches, subroutine calls, and so on are managed by the host processor. A vector is specified by a pointer to the first element and the shape of the vector data in memory, which includes stride, span, and skip values. The *span* describes how many elements to access at *stride* spacing before applying the *skip* offset. Vector operations are expressed as nodes in a data-flow graph where all dependencies are explicitly stated. Each node is specified by the input operands, the operation to be performed, the precision of the output and the sign.

2.4 Chip Multiprocessors

Given current limitations to increase performance by simply increasing the number of transistors, there is a growing trend towards the integration of multiple processors into a single chip. These multiple processors are not tied to be the same. On the contrary, new heterogeneous designs are appearing where general-purpose processor cores are packaged together with special-purpose ones for higher efficiency in processing multimedia and networking. Next we describe the *MAJC* architecture, an example of homogeneous VLIW CMP, and the *Cell*, which is currently the most representative example of heterogeneous CMP for multimedia.

MAJC

SUN's *Microprocessor Architecture for Java Computing* (MAJC) [TCC+00] is a high performance general-purpose microprocessor exceptionally suitable for multimedia computing. Its modular design provides scalability and the ability to exploit parallelism at a hierarchy of levels: at the data level through μSIMD instructions, at the

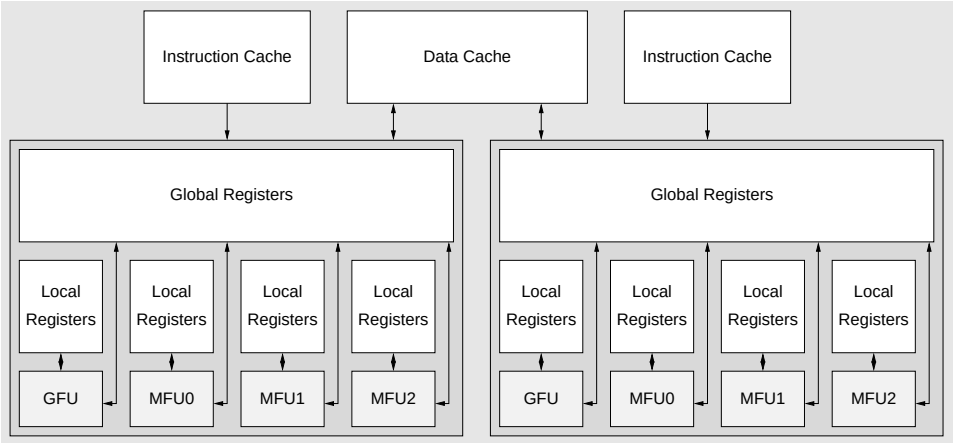


Figure 2.6. Architecture of the *MAJC-5200* processor

instruction level through multiple functional units, at the thread-of-execution level, and at the system level through multiple processor units on a chip.

MAJC supports vertical multithreading inside each processor unit. Vertical multithreading allows another thread to use resources that a stalled thread is not using. The system can hold the state of up to four threads at the same time, so that context switch is very fast. On the other hand, *MAJC* supports processor clusters, each containing multiple processor units, thus allowing different threads to run on separate processor units concurrently. Additionally, *MAJC* also allows speculative threads (future instruction streams) to execute on separate processors. The speculative threads operate in their own memory space and future time. Sun refers to this technique with the term *space-time computing* (STC).

The instruction set includes DSP-like features, such as saturation and μSIMD operations for both integer and floating-point data, powerful instructions for graphic applications, and a set of operations to facilitate byte and bit manipulation.

The first implementation of the *MAJC* architecture, the MAJC-5200 [Sud00], is shown in Figure 2.6. It is a multithreaded dual 32-bit microprocessor with a high input/output bandwidth. The two processors units share a coherent dual-ported 4-way set-associative 16 KB data cache and common external interfaces. Each processor unit is a 4-issue VLIW processor with four functional units: one *General Function Unit* (GFU), which is able to execute memory, flow or arithmetic operations, and three *Media Functional Units* (MFUs) for operations of compute type. Moreover, each processor unit contains its own 2-way set-associative 16 KB instruction cache.

The general-purpose register file is data type agnostic, that is, any register can hold information of any data type. All functional units within a processor unit share 96 registers, which are then called general (or global) registers. Additionally, each

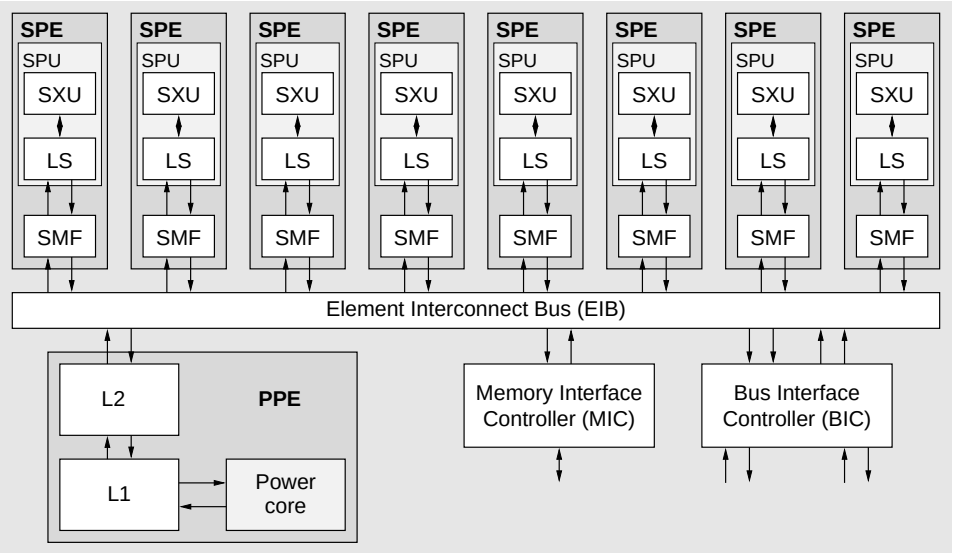


Figure 2.7. *Cell* system architecture

functional unit also has access to a set of 32 private (or local) registers. The 96 global registers plus the 4 sets of 32 private registers allow programs to use a maximum of 224 registers. Logically, the register file has 12 read ports and 5 write ports; physically, it is distributed into 4 register files of 3 read ports and 5 write ports each.

The only use of the MAJC-5200 was as the core of the XVR-1000 and XVR-4000 graphics accelerators. Nevertheless, many of the design ideas, specially in the multi-threading scope, laid the foundations for the design of next UltraSPARC processors.

Cell

The *Cell Broadband Engine Architecture* (CBEA) [KDH+05], also known as the *Cell* or the *CellBE* architecture, is a heterogeneous CMP jointly designed by Sony, Toshiba and IBM (also called the STI alliance). Although it was originally designed for the Sony's *PlayStation 3*, it is suitable to face a wide range of digital applications. Toshiba, for example, plans to incorporate Cell in high definition television sets, and IBM has recently released the *QS20* blade module using double Cell processors [IBM06a]. These modules are also expected to be a part of the IBM *Roadrunner* [IBM06b] supercomputer that will be operational in 2008.

The first implementation of the *Cell* architecture consists of a dual-threaded dual-issue *Power Processor Element* (PPE) (based on a 64-bit Power 970 core) augmented with eight specialized *Synergistic Processor Elements* (SPEs) (based on a novel SIMD architecture), an on-chip memory controller, and a controller for a configurable I/O interface (see Figure 2.7). These units are interconnected with a coherent on-chip *Element Interconnect Bus* (EIB).

Each SPE [GHF+06] consists of a *Synergistic Processor Unit* (SPU) and a *Synergistic Memory Flow Controller* (SMF). The SPU operates on a *Local Store* (LS) memory that contains instructions and data. All the transfers between this local memory and the system memory is performed via a DMA-based interface. It must be noted that the SPU cannot directly access the system memory. The SPU is an in-order dual-issue statically scheduled architecture based on the *pervasively data parallel computing* (PDPC) concept, in which wide datapaths are exploited throughout the system. The execution units are organized around a 128-bit dataflow. There is only one unified register file with 128 128-bit entries, which can be used for scalar data types ranging from 8-bits to 128-bits in size or for μSIMD computations on a variety of integer and floating point formats.

2.5 Summary

The significance of multimedia applications have produce a revolution in a great variety of markets, from the embedded to the high performance general-purpose domain. Technology advances allow current DSP processors to include features that were restricted not far ago to just the general-purpose domain. In fact, the 32-bit embedded processors have already narrowed the gap between embedded and desktop systems.

In order to face the performance, cost, and flexibility trade-offs of constantly changing multimedia applications, processors designers have been compelled to investigate for new processor architectures. Some of them try to accelerate multimedia execution by adding some specific support, such as μSIMD extensions or special-purpose co-processors, to existing microprocessors designs. On the other hand, ideas from the supercomputing domain have also been adapted to exploit the data level parallelism of multimedia codes.

In spite of the variety of existing alternatives, it is widely assumed that the combination of different paradigms is needed to exploit the heterogeneous parallelism of multimedia applications. Most of the current designs provide multicore and/or multithreaded functionality to support thread level parallelism, either static or dynamic superscalar capabilities to exploit instruction level parallelism, and some kind of SIMD support to deal with data level parallelism.

Realizing the computational demands, together with the cost and power consumption requirements of these new applications, it can be easily predicted that even more aggressive approaches are going to be implemented in future media processors.

Chapter 3

Compilation and Simulation Framework

This chapter overviews the compilation and simulation framework used in this thesis, *Trimaran*, and describes the main extensions built into the infrastructure to make it suitable for our work. These extensions include the possibility to extract statistics at the loop or region defined level, the insertion of a new module to perform loop disambiguation, the addition of new Vector-μSIMD units and Vector-μSIMD registers to the HPL-PD architecture, the extension of the compiler and the simulator to recognize, schedule and emulate the new operations, and the development of a simulator of the memory hierarchy. Finally, we summarize the main parameters of the reference architecture used in the evaluations.

3.1 Trimaran Choice

In this work we propose adding vector capabilities to high-performance μSIMD-VLIW processors to improve the performance of multimedia applications. The evaluation of the proposed architecture require developing new tools or adapting existing ones. Specifically, the target framework must allow experimentation in the architecture and in both the compilation and the simulation processes.

All the proposals presented in this thesis have been evaluated using the public domain *Trimaran* compilation and simulation framework [CGH+04]. Trimaran began as a collaborative effort between the Compiler and Architecture Research (CAR) Group (once a member of Hewlett Packard Laboratories), the IMPACT Group at the University of Illinois, and the ReaCT-ILP Laboratory at New York University (now known as CREST, the Center for Research on Embedded Systems and Technology at the Georgia Institute of Technology).

Although there are several compiler infrastructures available to the research community, Trimaran is especially useful for our research for several reasons. First, it is especially geared for ILP research. Second, it provides a rich compilation framework.

The parameterized ILP architecture (HPL-PD) space allows the user to experiment with machines that vary considerably in the number and kinds of functional units and register files and can vary in their instruction latencies. These machine configurations can be described using a machine description facility (MDES). Moreover, the modular nature of the compiler back-end (Elcor) and the intermediate program representation used throughout it allows the construction and insertion of new compilation modules into the compiler.

3.2 Overview of the Trimaran Compiler Infrastructure

Trimaran is a compiler infrastructure for supporting state of the art research in compiling for ILP architectures. The system is currently oriented towards *Explicitly Parallel Instruction Computing* (EPIC) [SR00] architectures, and supports compiler research in what is typically considered to be back-end techniques, such as instruction scheduling, register allocation, and machine-dependent optimizations.

The Trimaran compiler infrastructure is mainly comprised of the following components:

- A parameterized ILP Architecture, called HPL-PD.
- A machine description facility, called MDES, for describing ILP architectures.
- A compiler front-end for C, called IMPACT, which performs parsing, type checking, and a large suite of high-level (i.e. machine independent) classical and ILP optimizations.
- A compiler back-end, called Elcor, parameterized by a machine description, performing instruction scheduling, register allocation, and machine-dependent optimizations.
- A cycle-level simulator of the HPL-PD architecture which is configurable by a machine description and provides run-time information on execution time, branch frequencies, and resource utilization.

Figure 3.1 displays a block diagram of the overall system organization. Each component is described in more detail in the following lines.

3.2.1 Architecture Space

The architecture space targeted by Trimaran is the HPL-PD parametric processor [KSR00]. HPL-PD is a parametric architecture in that it admits machines of different composition and scale, especially with respect to the amount of parallelism offered. The HPL-PD parameter space includes the number and types of functional units, the composition of the register files, operation latencies and descriptors that

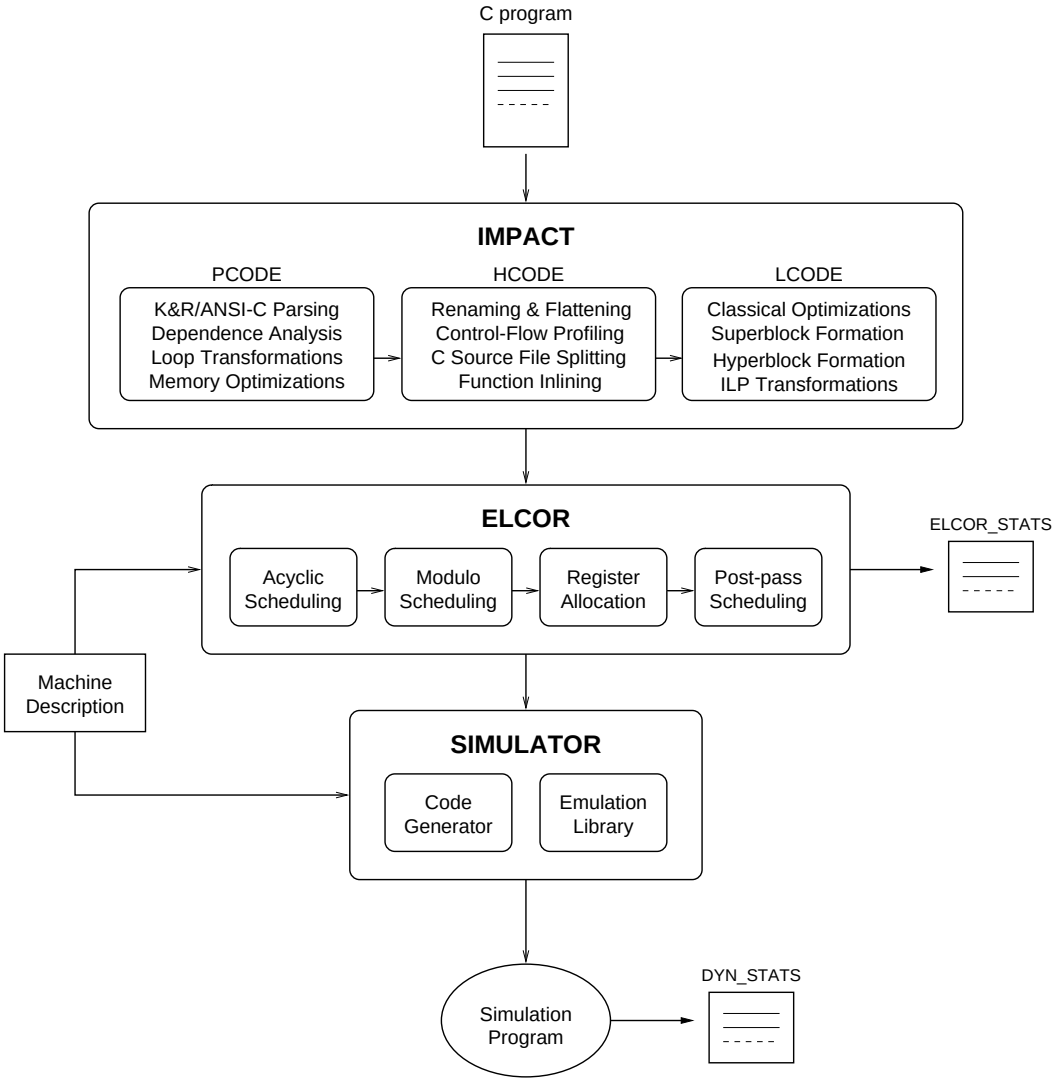


Figure 3.1. Trimaran compiler infrastructure

specify when operands may be read and written, instruction formats, and resource usage behavior of each operation.

The HPL-PD instruction set is similar to that of a RISC load/store architecture, with standard integer, floating point and memory operations. In addition, it provides a number of advanced features for enhancing and exploiting parallelism in programs, such as speculative and predicated execution, compiler exposed memory systems, a decoupled branch mechanism, and software pipelining.

Speculative execution is used to break certain types of dependences between operations. HPL-PD supports two forms of speculation: control speculation for code motion across conditional branches and data speculation for run-time disambigua-

tion. The architecture supports speculative execution of most operations; exceptions are stores and branches. To correctly handle exceptions generated by speculative operations, the architecture provides speculative and non-speculative versions of operations and speculative tag bits on registers.

Predicated or guarded execution refers to the conditional execution of operations based on a boolean-valued source operand, called a predicate. Predicated execution is often an efficient method to handle conditional branches and provides much more freedom in code motion. Predicate execution is also used in software pipelining as noted further on. To support predicated execution, the architecture provides 1-bit predicate register files and a rich set of compare-to-predicate operations which set predicate registers. In addition, most operations have a predicate input to conditionally nullify their execution. The compare-to-predicate operations are unique in that they can define two predicate registers simultaneously, for example, a compare may write the value of a comparison to one predicate, and the complementary value to the other predicate. Furthermore, the architecture permits multiple operations to write into a register simultaneously, provided all producers generate the same value. These write semantics are particularly valuable for the efficient evaluation of boolean reductions as carried out by the compare operations.

The *memory hierarchy* is unusual in that it is visible to the compiler. The ISA includes instructions for managing data across the hierarchy, for saving and restoring registers, and for performing run-time data disambiguation. The architecture provides latency and cache-control modifiers with load/store operations, which permit a compiler to explicitly control the placement of data in the memory hierarchy. the default in the absence of the use of these directives, is the conventional hardware management.

The *branch architecture* permits different pieces of branch related information to be specified as soon as they become available, in the hope the information can be used to reduce the adverse effect of the branch. A prepare-to-branch operation is used to specify the target address and the static prediction. The architecture provides a separate type of register file, called the branch target register file, to store this information. Compare-to-predicate operations are used to compute branch conditions, which are stored in predicate registers. Finally, branch operations test predicates and perform the actual transfer of control. The operation repertoire includes special branch operations to support software pipelining.

Software pipelining [Rau95] is a technique for exploiting parallelism across iterations of a loop. In software pipelining, the loop iterations are overlapped such that new iterations begin execution before previous iterations are complete. The set of instructions that are in flight at steady state constitute the kernel. To reach steady state, a subset of the instruction in the kernel are executed during a prologue stage; similarly, another subset is executed during an epilogue stage to complete the loop. During the prologue and epilogue stages, predication is used to nullify the appropriate subsets of the kernel. The architecture supports rotating registers in integer,

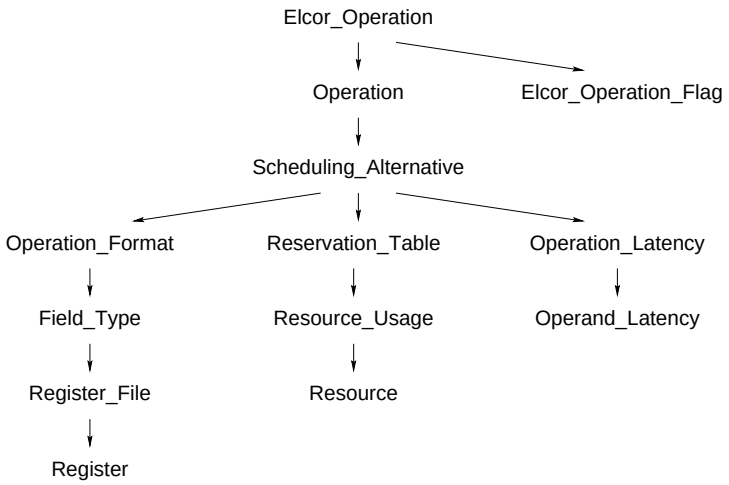


Figure 3.2. HMDES section hierarchy

floating-point and predicate registers in order to generate efficient software pipelined code.

3.2.2 Machine Description Model

HPL-PD adopts an EPIC philosophy whereby the compiler is responsible for statically leading the execution of a program. Thus, a compiler must have exact information pertaining to the particulars of the architecture definition within the HPL-PD space. In Trimaran, a machine-description (MDES) database specifies those particulars which include the register file structure, the operation repertoire, the set of resources in the architecture, the resource utilization patterns for each instruction, and the latency descriptors that define when an operand may be read or written after an instruction is issued.

The architecture is defined using a human-readable, high-level machine description (HMDES) language [GHR96]. The machine structure is described as a hierarchy of types called *sections*. Figure 3.2 shows the hierarchy of sections defined within the database file format. The description is then translated to a low-level language that specifies the same information but in a format that is suitable for a compiler. A MDES Query System (mQS) relays the information to a compiler through a procedural interface. The MDES methodology allows for a retargetable compiler infrastructure and enables experimentation with numerous performance-oriented compiler algorithms as well as architecture-exploration algorithms.

3.2.3 Compiler Front-end

The Trimaran front-end is based on IMPACT, an optimizing C compiler. IMPACT is an acronym for the Illinois Microarchitecture Project utilizing Advanced Compiler

Technology. The front-end is divided into three different modules depending on the level of intermediate representation (IR) used. The first level of IR, called *Pcode*, is a parallel C code representation with loop constructs intact. In Pcode, dependence analysis, parallelization, loop transformations, and memory system optimizations can be performed. Pcode functions are then translated into the *Hcode* format. Hcode is a flattened C representation containing only basic if-then-else and goto control flow constructs. The Hcode module is responsible for basic-block profiling, profile-guided code layout and function inline expansion. Finally, the code is translated to the *Lcode* format. Lcode is a machine-independent assembly like representation similar to many load/store RISC instruction sets. The Lcode module carries out classical code optimizations, Superblock [HMC+93] and Hyperblock [MLC+92] formation and ILP code optimizations. At the end of the process, the resultant code is translated into a bridge code readable for the Trimaran back-end.

3.2.4 Compiler Back-end

Elcor forms the back-end of the Trimaran compiler, and it is mainly responsible for scheduling and register allocation. In the Elcor IR, a program unit consists of a graph of operations connected by edges. This operation graph represents both, a traditional control flow graph and a data flow graph. The edges between operations model different kinds of control flow, data and memory dependences. The Elcor IR provides the necessary infrastructure to build, manipulate and transverse this graph.

The internal representation of the Elcor IR consist of a set of C++ objects. All optimization modules in the Elcor IR use the interface provided by these objects to carry out optimizations. Thus, optimizations are simply IR to IR transformations. The Elcor IR also has a textual representation, known as *Rebel*, with conversion routines between the two. Elcor is designed to allow implementing and testing new compilation modules. These new modules may augment or replace existing Elcor modules.

3.2.5 Simulator

The Trimaran infrastructure also includes an instruction set simulator (ISS). The ISS consumes the output of the Trimaran compiler to generate an executable binary which can simulate the original program.

The *code generator* module generates C files which correspond with the pseudo assembly files used as Elcor's IR. Because the assembly-equivalent files generated are in C, the simulation is completely platform independent. These files contain external variable declarations, global data and a set of emulation tables, which are arrays of HPL-PD machine operations.The main simulation loop processes these tables of operations and for each operation it invokes a function in the *emulation library* that implements the semantics of the opcode. There is a separate emulation function for each HPL-PD operation. The scheduling and latency information is present in the execution stream of instructions.

The simulator also aggregates structures to collect statistics at the block, procedure, and program level. Basically, it gives the scheduling length of each block and operation and cycle count and operations breakdown at the procedure level. In addition, the simulator can also produce an execution trace. The events that are recorded in the trace are: block entry, procedure entry, procedure exit, operation nullification, and memory accesses performed by the loads and stores in the program.

3.3 Extending the Trimaran Compiler Infrastructure

This section overviews the main extensions built into the Trimaran infrastructure. The first one is the addition of a new module into the Elcor back-end to manage loops. This module provides a great range of information about the loops in the compiled code. Our proposal for memory disambiguation [SCAV02] has also been implemented as part of this module. Second, we have extended the HPL-PD architecture with new Vector-μSIMD operations, functional units and register files [SV05b]. The compiler and the simulator have also been modified to recognize, schedule and emulate them. Finally, we have also developed a simulator of the memory hierarchy specially target to VLIW architectures simulation. Figure 3.3 shows the new Trimaran infrastructure with the more relevant additions and modifications.

3.3.1 The *Loops* Module

The main aim of this module is the development of a tool that allows to identify, characterize, and manage the most significant loops in a C program. The new loop driver routine is executed at the beginning of the Elcor compiler main driver. For each loop, it creates an object of a new class, called *Loop_ Region*. Loop detection and general control flow information are taken from the existing *Control* module. On the other hand, some functions in the *Stats* and *Visualize* modules have been adapted to work at the loop level, rather than at the procedure level. The loop driver is called again at the end of the Elcor driver to collect post-scheduling information such as scheduling and operation statistics. The blocks weights obtained from the IMPACT profiling are used to compute dynamic statistics. As the architecture parameters (number of functional units, latencies, and so on) have already been considered in the scheduling, static values does not differ significantly from those obtained dynamically.

An important contribution is the possibility to characterize memory operations. In order to do so, we have extended the concept of induction variable. The Elcor *Control* module identifies as induction variables those register operands which are unique defined in the loop by an addition or subtraction operation, in which the register is both the destination and the first operand and the second operand is loop invariant. These registers are classified as *basic induction variables*. The one related to the loop control branch is given the name of *primary induction variable*. We have defined the *extended induction variables* to be any register operand unique defined in the loop as the addition or subtraction of two operands, in which both operands can be either any induction variable (itself or another) or a loop invariant. Register

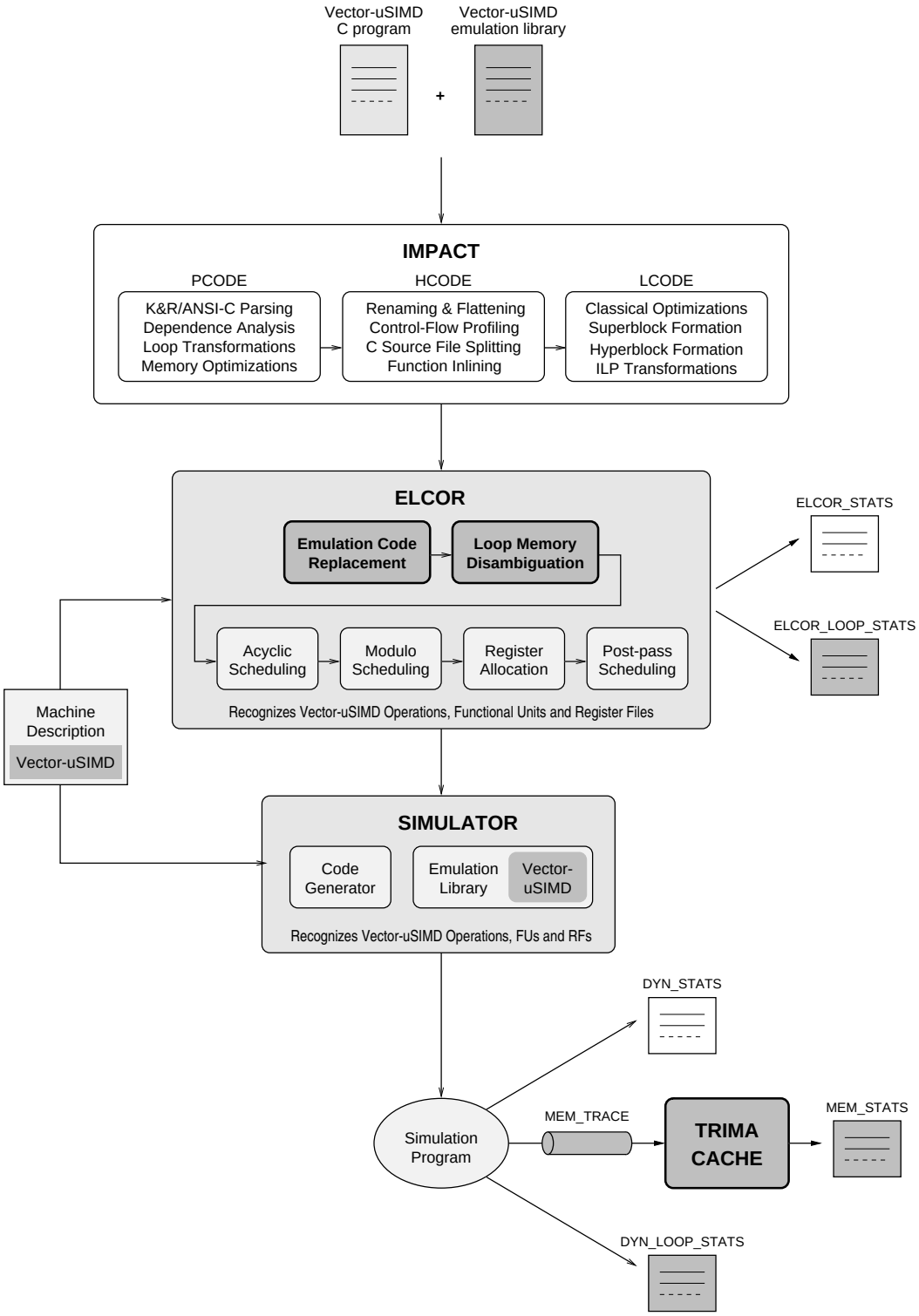


Figure 3.3. Extension of the Trimaran compiler infrastructure

operands which are the result of a shift operation over an induction variable are also considered to be extended induction variables. The step value for each induction variable is computed by tracking the operations performed over each variable and the involved literal values.

For each memory operation, if the address operand is an induction variable, either basic or extended, we can say that it is an strided reference whose stride is the step value of the variable. A deeper data flow analysis provides the compiler the ability to detect references whose address registers differ only in a constant term, that is, accesses to different elements of the same array.

The output produced by the *Loops* module can be controlled by the following flags:

- *print_loop_list*: If this flag is set to "yes", the whole list of loops is written out to a file. This file contains a line for each loop with the following data: source file name, loop name, dynamic cycle count, dynamic operation count, operation per cycle rate, number of invocations, average number of invocations per iteration, nesting level, being innermost or not, category (do-loop or while-loop), being modulo-scheduled or not, containing function calls or not, static number of operations, and the static number of loads and stores. This option is useful to get quick information about all the loops in a program, either to get main trends or to select the most relevant ones.

- *print_loop_info*: This option produces a file with detailed information about the loops in a more pleasant and readable format. A list of the loops to be analyzed can be specified by means of an input file. In addition to the data listed for *print_loop_list*, it provides operations breakdown, scheduling information, induction variables information, and memory operations information.

- *print_loop_mdg*, *print_loop_dfg*, and *print_loop_cfg*: These flags enable drawing the memory dependence graph, the data flow graph and the control flow graph of the loop respectively.

- *do_memory_disambiguation*: If this flag is set to "yes", loop memory disambiguation is performed at the beginning of the Elcor compilation chain. It can perform static memory disambiguation, dynamic memory disambiguation, and/or just delete the memory dependences listed in an input file, depending on the configuration parameters. It is independent of any memory disambiguation performed by the *Impact* front-end, so that both processes are not exclusive and can be used together [SV05a].

Finally, the simulator has also been extended to produce dynamic statistics (mainly cycle count, instruction and operation count and operation breakdown) at the loop level rather than at a procedure level.

3.3.2 Modifying the Architecture and the Instruction Set

Addition of Instruction Set Extensions to a compiler toolchain cannot be considered to be trivial. The first step involves defining the new machine operations and the new resources in the architecture. Specifically, we have extended the machine description with two new kind of register files (Vector-μSIMD registers and packed accumulator registers), two new kind of resources (Vector-μSIMD functional units and second level memory units), and 128 new operations.

The generic definitions of the *Register_File* and *Resource* HMDES sections have been extended with two new properties respectively: the optional *length* property, to specify the number of elements in a vector register, and the optional *lanes* property to specify the number of vector parallel lanes in a vector unit. On the other hand, introducing a new operation also involves defining the operation format, latency, resource usage and reservation table, and the possible scheduling alternatives.

Second, the compiler must be modified to be able to make use of the new operations. As our compiler front-end is not able to generate automatic code for the new architecture, the vector parts of the application C code have been hand-written using a function call for each operation (see an example in Figure 3.4.a). The corresponding emulation functions are defined in an external emulation library to verify code correctness.

At the input of the Elcor back-end, each function call appears in the form of a set of operations performing parameter passing and branch and link (Figure 3.4.b). We have inserted a new module at the beginning of the Elcor toolchain that identifies the branches to the emulation functions and replaces all the related set of operations by a new node in the IR which corresponds to a new Elcor operation (Figure 3.4.c). The source and destination operands of this new operation are obtaining by processing the parameter passing operations. A new virtual register number is assigned to each defined register operand and subsequent source registers are renamed accordingly. The compiler back-end will then treat it as any other standard operation.

The MDES interface has been extended to be able to generate the correct latency descriptors to the compiler. Additional minor modifications, including extending all reader/writer modules, have been performed along the Elcor compiler in order to recognize the new elements of the architecture and consider them in the scheduling and register allocation phases. Finally, the new elements have also been added to the simulator and the new operations semantics have been defined inside the emulation library.

3.3.3 *TrimaCache*

TrimaCache is a cycle-level simulator of the memory hierarchy specially designed for VLIW architectures. It is implemented as a set of layers. Each layer is composed by a set of banks and ports, a write buffer and a miss status holding register (MSHR).

```
...
M_PCK_SS_W(VR1, VR2, VR3);
M_V_ADD_SS_W(VR1, VR1, VR4);
...
```

(a) Vector-μSIMD C source code with emulation function calls

```
...
op 154 (MOVE [m<int_p1>] [i<1>] p<t>)
op 155 (MOVE [m<int_p2>] [i<2>] p<t>)
op 156 (MOVE [m<int_p3>] [i<3>] p<t>)
op 255 (PBRR [r<93:b btr>] [l:g_abs<_$fn_M_PCK_SS_W> i<1>] p<t> ...)
op 157 (BRL [m<ret_addr>] [r<93:b btr>] p<t> ...)
op 158 (MOVE [m<int_p1>] [i<1>] p<t>)
op 159 (MOVE [m<int_p2>] [i<1>] p<t>)
op 160 (MOVE [m<int_p3>] [i<4>] p<t>)
op 256 (PBRR [r<94:b btr>] [l:g_abs<_$fn_M_V_ADD_SS_W> i<1>] p<t> ...)
op 161 (BRL [m<ret_addr>] [r<94:b btr>] p<t> ...)
...
```

(b) Elcor IR before emulation function calls replacement

```
...
op 313 (M_PCK_SS_W [r<129:vx vxr>] [r<127:vx vxr> r<128:vx vxr>] p<t> ...)
op 314 (M_V_ADD_SS_W [r<130:vx vxr>] [r<129:vx vxr> r<120:vx vxr>] p<t> ...)
...
```

(c) Elcor IR after emulation function calls replacement

Figure 3.4. Emulation code replacement

The user can define the model of hierarchy, the number of layers, and the main characteristics of each layer (such as number and type of ports, banks, sets, associativity, block size, write policy, allocate policy, latency, write buffer size, and MSHR size).

At this moment, the simulator admits three possible hierarchy models: conventional superscalar model, a vector cache in the first level of the hierarchy, and a vector cache in the second level of the hierarchy. The *vector cache* has been implemented following the design presented in [QCEV99]. Basically, it is a two-bank interleaved cache targeted at accessing unit-stride vector requests by loading two whole cache lines (one per bank) instead of individually loading the vector elements. Then, an interchange switch, a shifter, and a mask logic correctly align the data (see Figure 3.5). If the port is B elements wide, these accesses are performed at a maximum rate of B elements per cycle when the stride is one, and at 1 element per cycle for any other stride.

Two cache models and three port definitions are implemented. Classical or perfect (always hit) multi-banked cache models can be combined with either a *true* (so many simultaneous memory accesses as the number of ports), a *pseudo* (so many simultaneous memory accesses as the number of ports as long as the references are

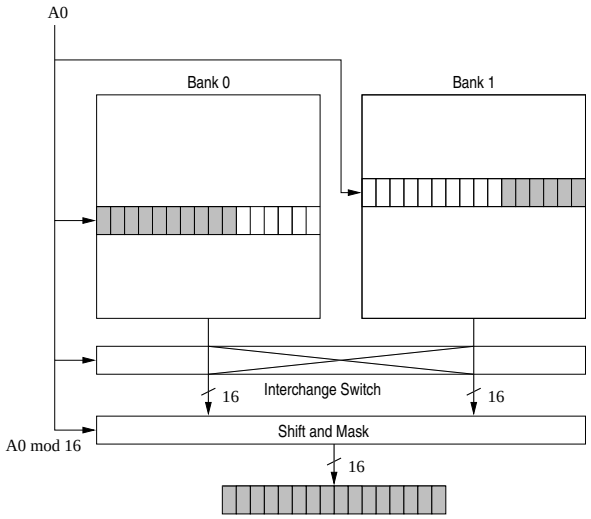


Figure 3.5. The dual bank structure of the vector cache

to different cache banks) or an *ideal* (infinite simultaneous memory accesses) multi-ported system.

The modeled write buffer is a coalescing write buffer of multiple entries, where each entry holds one cache line [SC97]. The retirement order is FIFO, except when a load hits a write buffer entry. In that case, we use a *flush-item-only* policy combined with *data bypass*. The normal retirement follows a *retire-at-X* policy, that is, it is produced when the number of valid entries is greater or equal than X, where X is a user defined parameter (usually half the number of entries).

TrimaCache accepts traces in both textual and binary formats. The trace can be seen as a succession of memory packets. One memory packet is composed by all memory operations issued on the same cycle. Figure 3.6 describes a memory packet in binary format. The first element of each packet is the cycle count in the global clock of the program simulation. The second element indicates the overall number of memory operations issued on that cycle. Next, for each operation, we find the operation type (scalar load, scalar store, vector load or vector store), the size (data width in the case of a scalar operation or the vector length and the vector stride in the case of a vector one), and the initial address being accessed. TrimaCache process each packet and accumulates the extra cycles needed to serve the memory requests.

Additionally, two special commands can be inserted in the trace: the *start_ region* command and the *close_ region* command. These commands are followed by an integer argument which identifies the region of code being entered or exited respectively. TrimaCache will then generate separate statistics for each region and for the full program.

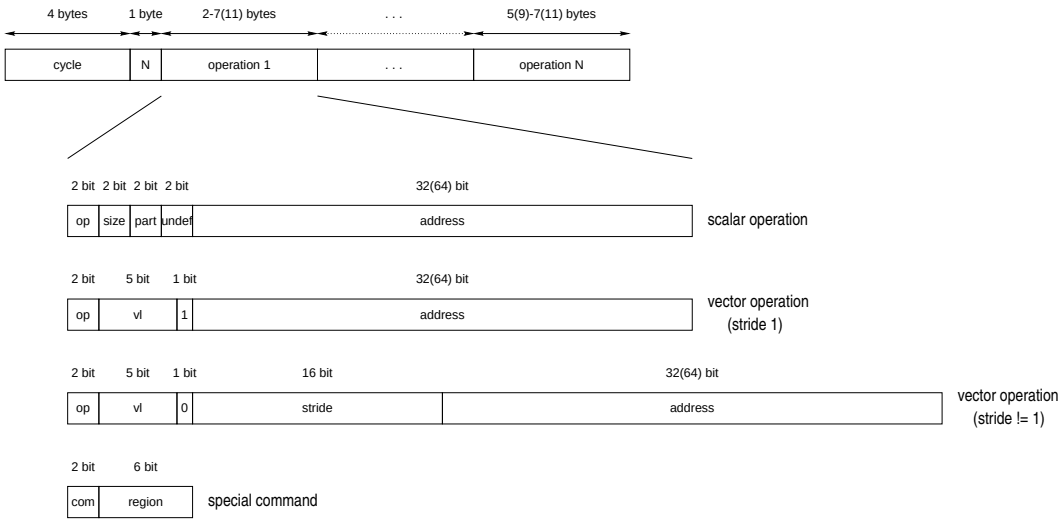


Figure 3.6. Memory trace packet description (binary form)

Two mechanisms have been implemented into Trimaran to take benefit from the possibility of defining multiple regions. The first one allows the user to define the regions directly in the source code by means of explicit calls to two empty emulation functions. These calls are later replaced in Elcor by two pseudo-operations. When these operations are simulated, the corresponding command is written to the memory trace. The second option consists on giving the simulator a list of the basic blocks which are part of each region. This is useful to use in combination with elcor outputs to automatically generate the list of interesting regions.

As a result of the simulation, TrimaCache produce a file with more than seventy statistic parameters for each memory hierarchy layer and for each program region separately, including memory activity cycles, hit rates on each structure, number and cycles of processor stalls due to different reasons, and so on. The Cacti model [SJ01] has also been integrated into TrimaCache in order to estimate time, energy and area cost of the evaluated configurations.

3.4 Reference Architecture

The focus of this thesis in on optimizing general-purpose VLIW processors, rather than extremely specific multimedia architectures. Any desktop computer is already able to execute a wide range of multimedia applications, including videoconference, 3D games, or DVD video. As multimedia workload continues increasing significance, processor designers must offer improved processors with powerful media performance.

Our reference architecture is a generic VLIW processor based in the HPL-PD architecture space, with guarded execution and software pipelining. Neither speculative execution nor exposed memory hierarchy are used, as they are not completely sup-

ported by the compiler. Table 3.1 summarizes the general parameters for 2-, 4-, and 8-issue width configurations. In order to support the high computational demand of multimedia applications, our configurations are quite aggressive in the number of arithmetic functional units. Latencies are based on those of the *Itanium2* processor [Int04].

Functional Units					*Memory Hierarchy*			
	2w	*4w*	*8w*	*latency*		*L1*	*L2*	*L3*
integer	2	4	8	$1, 4\,(\times, /)$	size (bytes)	16K	256K	1M
floating point	1	2	4	4	number of ports	1/2/4	1	1
memory	1	2	4	4	port width (bytes)	8	32	128
branch	1	1	1	2	number of banks	8	2	16
					sets per bank	16	128	64
Register Files					associativity	4	8	8
	2w	*4w*	*8w*		line size (bytes)	32	128	128
integer	96	128	160		write policy	WT	WB	WB
floating point	96	128	160		allocate policy	NWA	NWA	WA
predicate	64	96	128		mshr size	8	8	8
branch target	8	16	24		write buffer size	8	8	8
					retire-at-X	4	4	4
					latency	1	5	12

Table 3.1. Modeled processor configurations

The cache hierarchy is decoupled into three on-chip levels. The first level data cache is a 16 KB, 4-way set associative cache with one port for the 2-issue width architecture. We consider pseudo-multi-ported caches for the configurations with greater number of ports. There is a 256KB cache in the second level and a 1MB cache in the third level. Latencies are 1 cycle to the L1, 5 cycles to the L2, 12 cycles to the L3 and 500 cycles to main memory. We have not simulated the instruction cache since our benchmarks have small instruction working sets. The compiler schedules all memory operations assuming they hit in the first level cache and the processor stalls in case of a cache miss or a bank conflict.

3.5 Summary

To evaluate the architectural improvements and compilation techniques proposed in this thesis, we have used and extended version of the Trimaran compilation and simulation framework. The choice of this tool set was mainly due to its potential to be adapted, not only in the instruction set simulation, but also in the machine description and the compilation process.

The developed tools allows us to characterize entire applications at the loop or region level, evaluate a new loop memory disambiguation technique, experiment with a new Vector-μSIMD-VLIW architecture, and perform a detailed simulation of the memory hierarchy. This chapter has also described the general VLIW architecture used as referenced along this work.

Chapter 4

Workload Characterization

Understanding the behavior of multimedia applications is essential for processor design research. Nevertheless, workload analyses are compromised by the difficult to isolate the effects of the implementation of the algorithm, the compiler optimizations, and the underlying architecture. This chapter is an attempt to verify and quantify main trends and characteristics rather than to perform a thorough characterization. We start by summarizing the main characteristics of multimedia codes. Next, we introduce and discuss the selection of benchmarks used along this study. Finally, we present some experimental results to verify these characteristics in our set of applications. As fine grain parallelism is mainly found in the form of small loops that operate on streams of data, we analyze the behavior of loops and complete applications separately.

4.1 General Characteristics of Multimedia Codes

Typical media programs consist of a set of kernels that process data in a stream-like fashion, with the addition of some protocol related overhead such as header processing and output encoding. As the kernels are invoked over the streams like different stages in a pipeline, the behavior of these kernels in isolation differs from their behavior inside the complete application. This section overviews the main conclusions found in the literature about the characteristics of multimedia codes. First, we describe the general behavior at the kernel level, and then, how these features are modified when they are considered inside the scope of the complete application.

4.1.1 Characteristics of Multimedia Kernels

Most authors in the literature agree that the main characteristics of multimedia kernels are the following [LS96, DD97, CDJ+97]:

- *Small data type sizes.* Multimedia data items often derive from sampling an analog signal in the time domain, such as video or audio. In contrast with

other kind of applications where 32 or 64 bit precision is needed, media data types are usually 8 or 16 bits, since human sense cannot discriminate beyond that range.

- *Significant data parallelism.* Input data streams are frequently large collections of small data elements such as pixels, vertexes or audio samples. Furthermore, the same set of operations are performed over the elements inside the stream. Thus, media kernels exhibit high amounts of data level parallelism.

- *High instruction reference locality.* Media applications often consist of a set of computationally intensive small loops that dominate over the processing time, which results in high instruction cache hit rates.

- *Low data reference locality.* Data is usually loaded, processed, and returned back to memory. As the streams are reused only once, temporal locality is low. On the other hand, as the streams often exhibit a multi-dimensional nature, spatial locality is also difficult to exploit.

- *High memory bandwidth.* The huge working sets of some type of applications, such as 3D imaging, means that processors will need to provide high memory bandwidth and tolerate long memory latencies.

- *Real-time constraints.* Multimedia applications, such as video conferencing, often require real time response and a certain quality of service.

4.1.2 Characteristics of Multimedia Applications

As it has been stated before, kernels process data in a streaming way, and these streams can be sparse across different dimensions. Nevertheless, as these kernels are repeatedly invoked on sets of related data, there is often some kind of overlapping between the different streams. Furthermore, the stream produced in one stage of the pipeline is consumed by the following stage. Thus, these stream-like patterns exhibit *temporal and spatial locality at the scope of the complete program*, which makes the use of cache hierarchies desirable. Several works coincide that data caches do not perform worse for multimedia than for traditional integer and floating point workloads [LPMS97, SS01, RAJ99].

Lee et al. [LPMS97] introduced and analyzed the *MediaBench* suite. They found that the MediaBench programs exhibit higher instruction cache hit rates than the SPECint ones, and that data caches are more effective for reads on MediaBench than on SPECint, although they are less effective for writes. They also noted that the SPECint applications require almost three times more bus bandwidth and achieve lower IPC than the MediaBench ones.

Slingerland and Smith [SS01] analyzed the cache behavior of the *Berkeley Multimedia Workload* [SS02]. They obtained that, except for 3D and document applications, a 32 KB cache is large enough to get extremely low miss ratios.

Ranganathan et al. [RAJ99] provided a quantitative understanding of the performance of image and video processing applications on general-purpose processors, with and without media ISA extensions. They also observed some differences between kernels and complete applications. While the kernels exhibit poor data locality and take benefit from software prefetching techniques, they conclude that software prefetching is not needed for complete applications

It is also widely assumed that *multimedia applications exhibit more parallelism than conventional applications*. Liao and Wolfe [LW97] analyzed the available parallelism in some video applications. They obtained a high amount of ILP ranging from 32.8 to over 1,000 independent instructions per cycle using an idealized execution model (perfect branch prediction, perfect memory disambiguation, infinite resources and infinite scheduling window); whereas Wall [Wal91a] concluded that less than 10 instructions can be issued in parallel for conventional integer applications.

However, Fritts [Fri00] added two extra video processing applications to MediaBench and conducted a set of experiments on an intermediate low-level format. And he found that the basic-blocks in multimedia applications are so small than the parallelism is not within basic-blocks.

On the other hand, although media kernels are characterized by high amounts of data parallelism, complete applications also contain first order recurrences, table look-ups and non-streaming memory patterns with large amounts of indirections, like in the SPECint. Therefore, *there is a significant portion of multimedia codes that is difficult to vectorize* [JVTW01]. Moreover, although most of the algorithms in the standard have a vector nature, there has been a great effort on reducing the overall number of required operations especially oriented towards scalar architectures, hiding in most cases the data parallel nature of the original algorithm.

One representative example is the DCT algorithm. This transformation can be represented as a matrix operation using a 8x8 transform matrix A to obtain the 8x8 transform coefficients matrix C based on a bilinear transformation: $C = A \cdot B \cdot A^T$, where B is the input block and A^T denotes the transpose of A. This would involve 1024 multiplications for each input block. Nevertheless, several fast algorithms have been introduced in the literature aimed at reducing the number of multiplications involved in the transform [Lee84]. The algorithm used in the JPEG standard only needs to perform 192 products to produce one resultant block; but because of this optimization, the new code cannot be directly vectorized.

4.2 Benchmarks Description

The difficulty to capture all of the essential elements of modern multimedia and communication systems is reflected in the lack of any standardized benchmark suite. Parameters that influence the overall application behavior, such as the predominance of each media source, the size of its working set, or the level of protocol overhead

are hard to determine. Even already standardized protocols such as MPEG4 are still slightly ambiguously defined and it is difficult to obtain reliable, non research-oriented source codes. Furthermore, the difference characteristics that we find when we look at a different scopes of media processing, as seen in previous sections, strongly suggest that study of kernels in isolation may bring to misleading conclusions.

As highlighted by its authors, the *MediaBench* is composed of full programs that capture the essential characteristics of media and communication systems, including video, audio, still-image, 3D and encryption standard algorithms. To expedite the next generation of systems research, the *MediaBench Consortium* is developing the *MediaBench II* benchmark suite [FST05], incorporating benchmarks from the latest technologies and providing both a single composite benchmark suite as well as separate benchmark suites for each area of multimedia.

Our methodology is based on selecting a set of multimedia programs from the *MediaBench* suite that approximate the contents of current image, video and audio applications. For every standard, both the encoding and decoding are included. Table 4.1 describes the set of benchmarks selected, together with a brief description and the input sets used for simulation.

Benchmark	Description and input set
jpeg_enc	Descr: JPEG image compression encoder Input: penguin.ppm (ppm file, 24-bit color 1024x739 image)
jpeg_dec	Descr: JPEG image compression decoder Input: penguin.jpg (JPEG file, 1024x739 image)
mpeg2_enc	Descr: MPEG2 digital video encoder Input: mei16v2rec.Y/Cb/Cr (four 24-bit color 352x480 frames Y-Cb-Cr)
mpeg2_dec	Descr: MPEG2 digital video decoder Input: mei16v2rec.mpg (MPEG2 video stream, four 352x480 frames)
gsm_enc	Descr: GSM 06.10 speech encoder Input: clinton.pcm (8KHz sampling rate, 300KB PCM audio stream)
gsm_dec	Descr: GSM 06.10 speech decoder Input: clinton.gsm (13Kb/s GSM audio stream)
epic_enc	Descr: Image compression encoder Input: test_image.pgm (pgm file, gray scale 256x256 image)
epic_dec	Descr: Image compression decoder Input: test_image.pgm.E (EPIC file, 256x256 image)

Table 4.1. Benchmarks description and input sets characteristics

JPEG is a compression standard for either grayscale and color digital images based on the DCT-method [Wal91b]. The codification is performed in three stages: *color space conversion* and *downsample*, *forward DCT transform and quantization*, and *entropy coding*. In color space conversion, each pixel from the source image is converted from the RGB to its YUV representation and then the chrominance components (U and V) are downsampled by a factor of two on both spatial dimensions. The

forward DCT processing step lays the foundation for achieving data compression by concentrating most of the signal in the lower spatial frequencies. Source images samples are grouped into 8x8 blocks and input to the DCT. The output is another block of 64 coefficients with the property that most of them have zero or near-zero amplitude and do not need to be encoded. Afterwards, each coefficient is quantized with the purpose to achieve further compression by representing the coefficients with no greater precision than is necessary to achieve the desired image quality. Finally, all the quantized coefficients are ordered into a zig-zag sequence, so that they can be encoded more compactly based on their statistical characteristics (Huffmann coding). The decoder just performs the inverse operations in the reverse order.

The MPEG2 video compression standard was developed by the *Motion Picture Experts Group* [Sik95]. Video sequences usually contain statistical redundancies in both temporal and spatial direction. Spatial correlation is exploited for each frame in the same way as JPEG, and motion compensated prediction techniques are used to reduce temporal redundancies between frames. *Motion estimation* searches which block of the previous image matches better with the block being compressed (this becomes the most computationally-intensive part of the process), and the resulting displacement between the two blocks is called the motion vector. Usually, the block size is 16x16 pixels for the luminance component (Y) and 8x8 for the chrominance components (U and V). A motion compensated difference block is then formed by subtracting the pixel values of the predicted block from that of the current block. The difference block is then transformed, quantized and entropy coded.

The GSM vocoder is the standard algorithm to perform voice compression for the *Global System for Mobile Communications* or GSM, that is one of the most important second-generation digital mobile phone systems today (especially in Europe) [Tri01]. While there are more than one implementations, this version is the original European vocoder (standard GSM 06.10), which uses residual pulse excitation/long term prediction (RPE-LTP speech encoder) coding at 13 Kb/s blocks of 260 bits (from frames consisting of 160 13-bit samples). The RPE-LTP process is commonly multiplexed by a VAD (Voice Activity Detection) unit, that is responsible for detecting frames of time where the speaker is not talking (so that bandwidth and processing overhead can be saved).

Finally, EPIC is an experimental lossy image compression utility. The compression algorithm is based on a critically-sampled non-orthogonal (imperfect-reconstruction) dyadic wavelet decomposition and a combined run-length/Huffman entropy coder [AS90]. The filters are designed for extremely fast decoding on non-floating point hardware, at the expense of slower encoding and a slight degradation in compression quality (as compared to a good orthogonal wavelet decomposition). This property makes it useful for applications that involve asymmetric computational resources, such as centralized image databases.

4.3 Loop Level Analysis

In order to analyze the main characteristics of media loops, this section presents some quantitative data such as coverage, loop size, operation per cycle rate, data size, and stride and length of array memory references. Results are presented on the scope of each application, but detailed information about each loop in particular can be found in Appendix A.

4.3.1 Coverage

For each application, Table 4.2 shows the number of innermost, do-loops, and modulo scheduling loops together with the percentage of the overall dynamic cycles and operations they represent. Note that each category is a subset of the previous one.

	Innermost			Do-loop			Mod Sched		
Benchmark	#L	%Cyc	%Ops	#L	%Cyc	%Ops	#L	%Cyc	%Ops
jpeg_enc	32	48.49%	61.74%	23	47.97%	61.41%	23	47.97%	61.41%
jpeg_dec	33	83.02%	85.05%	25	82.87%	84.89%	21	25.54%	26.48%
mpeg2_enc	59	63.94%	78.13%	45	63.86%	78.11%	43	60.99%	76.34%
mpeg2_dec	26	36.93%	34.37%	17	31.92%	30.85%	15	10.41%	10.11%
gsm_enc	30	59.53%	76.20%	23	57.72%	74.93%	22	57.29%	74.66%
gsm_dec	13	93.39%	92.82%	8	6.08%	6.74%	7	5.63%	6.24%
epic_enc	38	55.85%	58.37%	15	39.73%	47.10%	15	39.73%	47.10%
epic_dec	32	70.81%	80.75%	23	48.48%	52.75%	23	48.48%	52.75%
sum/average	263	64.00%	70.93%	179	47.33%	54.60%	169	37.01%	44.39%

Table 4.2. Coverage of innermost, do-loops and modulo scheduling loops (number of loops and percentage of the overall dynamic cycles and operations)

In average, the applications spend the 64.00% of the overall execution time in innermost loops. The application with the lowest coverage is the `mpeg2_dec`, in which the innermost loops only represent the 36.93% of the overall execution time. This is mainly due to the high amount of overhead to deal with different input configurations. A different case is the `gsm_dec`. In spite of having a very reduced number of loops, this benchmark exhibits the highest coverage (93.39%). However, the main loop, which means the 80% of the overall execution time, is not a do-loop. As a result, this benchmark exhibits the lowest coverage when considering do-loops or modulo scheduling loops.

4.3.2 Loop Size

To analyze the size of the loops, Table 4.3 shows the average number of static operations, invocations and iterations per invocation for the loops of each application. We have also classified the loops into three categories depending on the number of iterations per invocations: below 16, between 16 and 64, and above 64.

	Stc Ops	Inv	Iter/Inv	Iter/Inv≤16		16<Iter/Inv≤64		64<Iter/Inv	
Benchmark	Avg	Avg	Avg	#	%Cyc.	#	%Cyc.	#	%Cyc.
jpeg_enc	18	6,875	71	24	9.06%	4	22.77%	4	16.65%
jpeg_dec	45	1,529	81	26	57.49%	2	0.01%	5	25.53%
mpeg2_enc	28	135,361	59	48	59.05%	6	4.31%	5	0.58%
mpeg2_dec	48	5,244	112	19	35.76%	4	1.13%	3	0.04%
gsm_enc	32	7,182	42	18	24.14%	6	10.75%	6	24.64%
gsm_dec	31	2,281	57	8	1.47%	2	3.55%	3	88.37%
epic_enc	12	79,118	3,717	33	50.95%	1	0.59%	4	4.31%
epic_dec	59	838	5,193	9	18.12%	16	0.75%	7	51.95%
sum/average	34	29,804	1,166	185	32.00%	41	5.48%	37	26.51%

Table 4.3. Loop-body size (average number of static operations, invocations, and iterations per invocation, and distribution of loops according to the number of iterations per invocation)

It can be observed that we are mainly dealing with small loops (34 static operations per loop in average), with small loop counters, and which are executed a lot of times. A particular case is the EPIC applications. These benchmarks include loops which are executed thousands of times but with only one iteration per invocation, and other loops with thousands of iterations but only one invocation. This leads to confusing results when looking at average numbers. On the other hand, it can be noted that most loops execute less than 16 iterations per invocation, and only `jpeg_enc` and `gsm_enc` have representative loops in the category between 16 and 64.

4.3.3 Memory References

In this section we evaluate the main characteristics of the memory accesses performed in the loops. First, the distribution of the data size and stride values of all memory operations in the loops are shown in Tables 4.4 and 4.5 respectively. Then, array references are analyzed separately in Table 4.6.

Benchmark	1 byte	2 bytes	4 bytes	8 bytes
jpeg_enc	31.52%	38.52%	29.96%	0.00%
jpeg_dec	58.81%	5.65%	35.54%	0.00%
mpeg2_enc	93.35%	2.84%	0.51%	3.29%
mpeg2_dec	54.17%	34.93%	10.90%	0.00%
gsm_enc	0.00%	89.15%	10.85%	0.00%
gsm_dec	0.00%	100.00%	0.00%	0.00%
epic_enc	3.98%	1.85%	92.20%	1.98%
epic_dec	3.14%	5.86%	81.59%	9.41%
average	30.62%	34.85%	32.70%	1.83%

Table 4.4. Data size of memory references

As can be observed, most of the memory accesses (about 75% in average) require 16 bits or less. Moreover, most of the applications have a characteristic data size:

one byte for video applications, two bytes for audio, and four bytes (floating point) for the EPIC applications. Note that, although these are the predominant storage widths, higher data sizes are normally used during computation due to precision requirements. In the JPEG image applications, input and output data are one byte, but intermediate data is stored in two or even four bytes. On the other hand, about 75% of the memory operations have a stride of 1, 2, 3 or 8; the remaining 25% are either invariant or non-strided references. Non-strided references correspond mainly to the use of memory tables to perform computation, such as multiplications or saturation.

Benchmark	Invariant	Stride 1	Stride 2	Stride 3	Stride 8	Non-strided
jpeg_enc	0.00%	43.32%	6.66%	9.98%	10.08%	29.95%
jpeg_dec	0.28%	38.58%	6.62%	9.95%	5.65%	38.92%
mpeg2_enc	0.51%	97.81%	0.00%	0.00%	1.30%	0.38%
mpeg2_dec	10.78%	72.50%	0.00%	0.00%	5.85%	10.87%
gsm_enc	10.93%	88.25%	0.00%	0.20%	0.00%	0.61%
gsm_dec	12.49%	48.53%	0.00%	0.00%	0.00%	38.98%
epic_enc	6.07%	90.93%	0.00%	0.00%	0.00%	2.99%
epic_dec	10.31%	44.82%	19.15%	0.00%	0.00%	25.72%
average	6.42%	65.60%	4.05%	2.52%	2.86%	18.55%

Table 4.5. Stride of memory references

The previous data were obtained considering each memory operation in isolation. However, different memory operations can in fact be referencing elements of the same array, and form what it is call a *reference group* (see Section 5.2.3 in Chapter 5). In color conversion, for example, the input stream contains three bytes for pixel (one for each color component). The innermost loop processes one row of pixels, so that the three components of one pixel are loaded each iteration. If we look at each component load independently, we will see three memory references with a stride of three and length the image width; but in fact we are accessing one single array with a stride of one and length three times the image width.

	Length				Stride		
Benchmark	Avg	Most frequent lengths			1	2	8
jpeg_enc	586	8 (17.45%)	64 (37.18%)	1,024 (26.42%)	89.58%	7.56%	2.86%
jpeg_dec	996	8 (12.12%)	510 (24.30%)	1,024 (36.55%)	90.19%	0.00%	9.81%
mpeg2_enc	17	2 (5.13%)	8 (5.65%)	16 (84.84%)	98.76%	0.00%	1.24%
mpeg2_dec	19	8 (23.34%)	11 (31.94%)	12 (16.09%)	98.47%	0.00%	1.53%
gsm_enc	27	8 (49.11%)	40 (40.05%)	160 (3.95%)	99.38%	0.00%	0.00%
gsm_dec	205	40 (16.09%)	120 (19.31%)	320 (51.49%)	100.00%	0.00%	0.00%
epic_enc	6554	2 (13.07%)	4 (34.26%)	5.041 (24.36%)	100.00%	0.00%	0.00%
epic_dec	24509	90 (19.57%)	5,041 (21.92%)	65,536 (32.87%)	83.39%	16.61%	0.00%
average	4114	8 (14.16%)	16 (10.99%)	1.024 (7.87%)	94.97%	3.02%	1.93%

Table 4.6. Length and stride of array references.

Table 4.6 shows the length and stride of array references considering a reference group as one array reference. For the length, the table shows the average and the three most frequent values, which are different for each application. For example, typical lengths for `jpeg_enc` are 8, 64, and the image width (1024 for the reference input). As far as the stride is concerned, we observe that 95% of the arrays are accessed with a stride of one.

4.3.4 Operations per Cycle

To conclude the loop level analysis, Table 4.7 shows the average operation per cycle (OPC) rates achieved in the innermost loops for the 2, 4 and 8-issue width VLIW architectures. The percentage in brackets indicates the increase over the OPC of the previous issue width.

Benchmark	2-issue	4-issue		8-issue	
jpeg_enc	1.64	2.12	(+29.00%)	2.22	(+ 4.75%)
jpeg_dec	1.64	1.87	(+14.37%)	1.93	(+ 2.70%)
mpeg2_enc	1.72	2.62	(+52.40%)	3.57	(+36.41%)
mpeg2_dec	1.55	1.75	(+12.81%)	1.77	(+ 0.82%)
gsm_enc	1.69	2.56	(+51.52%)	3.28	(+27.84%)
gsm_dec	1.36	1.47	(+ 7.64%)	1.46	(- 0.69%)
epic_enc	0.78	1.05	(+34.71%)	1.06	(+ 0.51%)
epic_dec	1.27	1.40	(+10.17%)	1.42	(+ 1.37%)
average	1.46	1.86	(+26.58%)	2.09	(+ 9.21%)

Table 4.7. Operations per cycle rate in innermost loops for different issue widths

Results confirm that multimedia kernels exhibit more ILP than integer ones. Except for the `epic_enc` application, all benchmarks achieve fair OPC rates in the innermost loops. Nevertheless, for most of the benchmarks, scaling the architecture from 4 to 8-issue is not specially attractive. Only loops in `mpeg2_enc` and `gsm_enc` show a significant improvement when increasing the issue width from 4 to 8.

4.4 Application Level Analysis

This section provides quantitative data about our set of multimedia applications. It includes the analysis of the following topics: static and dynamic code size, operation per cycle rate, operation breakdown, data locality and memory hierarchy.

4.4.1 Static Code Size

Table 4.8 shows the overall number of static operations, blocks (either basic-blocks or hyper-blocks), and functions in each benchmark, together with the number and percentage of them which are in fact executed.

	Operations		Blocks		Functions	
Benchmark	Overall	Touched	Overall	Touched	Overall	Touched
jpeg_enc	46,726	16,886 (36.14%)	3,540	506 (14.29%)	311	106 (34.08%)
jpeg_dec	45,346	21,188 (46.73%)	3,055	533 (17.45%)	266	104 (39.10%)
mpeg2_enc	37,306	30,248 (81.08%)	1,607	662 (41.19%)	93	78 (83.87%)
mpeg2_dec	23,635	13,665 (57.82%)	1,436	348 (24.23%)	112	63 (56.25%)
gsm_enc	31,030	15,954 (51.41%)	1,078	322 (29.87%)	94	57 (60.64%)
gsm_dec	30,795	8,799 (28.57%)	1,265	185 (14.62%)	94	44 (46.81%)
epic_enc	10,476	6,590 (62.91%)	899	254 (28.25%)	46	27 (58.70%)
epic_dec	8,858	6,643 (74.99%)	408	188 (46.08%)	34	14 (41.18%)
average	29,272	14,997 (54.96%)	1,661	375 (27.00%)	131	62 (52.58%)

Table 4.8. Static operation, block and function counts

It can be observed that a significant amount of code is not touched during the execution of the reference inputs. Half of the static operations and functions and two thirds of the basic-blocks are not used during the execution of the program. This low code utilization implies that either the applications contain superfluous code, or their inputs do not exercise many of the control paths. The superfluous code includes functions without any call in the rest of the code, functions that are only used in the opposite codec side, and functions to support options which are not included in the definition of the standard.

A thorough categorization of the unused code can be found in [HH02]. The authors also show that additional inputs often introduce very little variation in the control flow pattern. They claim that these factors must be carefully taken into account, as they can skew a wide variety of experiments, such as the evaluation of techniques whose impact is measured in terms of code size.

4.4.2 Dynamic Code Size

Table 4.9 reports the dynamic operation, block, and function counts. The benchmarks execute a few hundred million operations for the reference inputs. Results confirm the assumption that codecs are designed to allow faster decodification, in clear detrimental of the codification side. This is especially true for MPEG2 and EPIC, where the decoders require to execute about twenty and nine times less operations than the encoders.

The block size (31 operations per block in average) is slightly larger than those reported in the literature. Fritts reports than the average basic block size of multimedia applications is similar to that of integer applications [Fri00]. It must be taken into account that hyperblock formation is performed by the Impact front-end. During hyperblock formation, if-conversion [PS91] is used to form larger blocks of operations, and thus providing a greater opportunity for code motion to increase ILP. We have

		Blocks		Functions	
Benchmark	Operations	Blocks	Ops/Block	Funcs	Ops/Func
jpeg_enc	204,894,494	6,792,840	30	377,714	542
jpeg_dec	171,402,016	2,488,973	69	64,516	2,657
mpeg2_enc	1,677,337,176	172,260,841	10	1,470,927	1,140
mpeg2_dec	86,580,636	4,241,564	20	264,393	327
gsm_enc	235,933,412	4,636,447	51	145,329	1,623
gsm_dec	125,935,930	2,680,835	47	94,074	1,339
epic_enc	75,233,661	16,332,469	5	1,864	40,361
epic_dec	8,912,338	631,708	14	314	28,383
average	323,278,708	26,258,210	31	302,391	9,547

Table 4.9. Dynamic operation, block and function counts

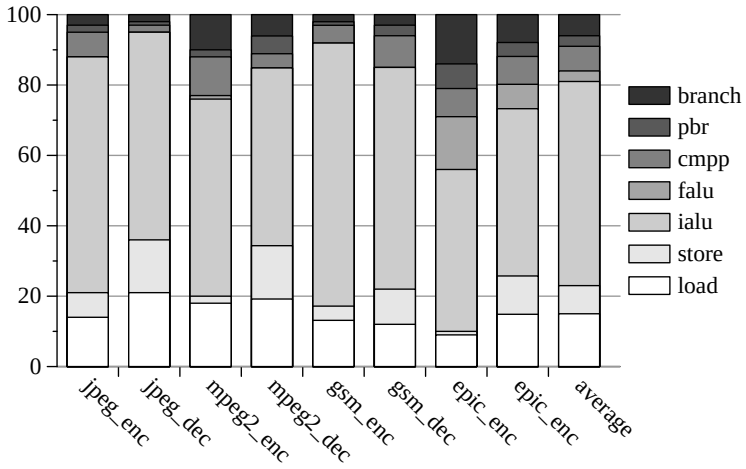


Figure 4.1. Operation breakdown

also noticed that the compiler introduces a high amount of spill code, especially in `jpeg_dec`, `mpeg2_dec`, and `gsm_dec`.

4.4.3 Operation Breakdown

The graph in Figure 4.1 shows the distribution of dynamic operations classified into memory operations (load and store), arithmetic operations (integer and floating point), and control operations (compare, prepare-to-branch, and branch).

The percentage of floating point operations is relatively low, which confirm that multimedia programs are mostly integer. Only `epic_enc` and `epic_dec` use floating point arithmetic. The `mpeg2_enc` application has a minimal floating point operation ratio of 1.17%. These operations are used to compute the forward DCT, which is implemented using the double precision matrix product algorithm instead of a fast scalar algorithm, and to compute some statistics.

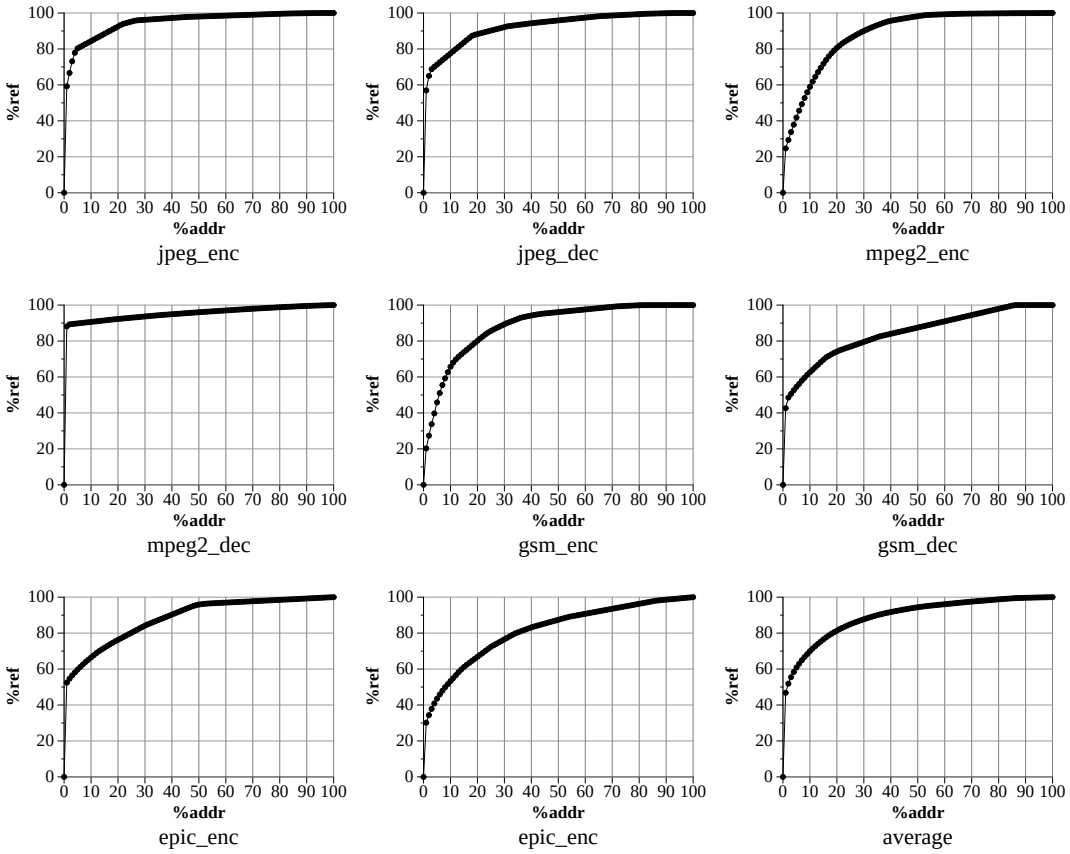


Figure 4.2. Data locality histograms

Load and store operations are relatively higher for video and image processing applications. It is worth noting again that encoders have more computational requirements than decoders. As a result, the percentage of memory operations increases significantly in the decoders.

The branch per operation ratio is 6.01%, which means that only one out of every 17 operations is a branch. The same ratio is reported by Talla [Tal01], who also reports that one out of every 6 instructions is a branch in the SPECint benchmark suite, and one out of 25 instructions is a branch in the case of SPECfp. The low branch ratio fits in with the large block size and the potentially high ILP of multimedia applications.

4.4.4 Data Locality

Figure 4.2 shows the data locality histogram for each benchmark. Horizontal axis represents the percentage of referenced memory locations, and the vertical axis is the accumulated percentage of references. As both axis are sorted, the point (X, Y) indicates that the $Y\%$ of the references are performed over the $X\%$ of the most referenced memory locations.

In general, the benchmarks exhibit very low data reuse: in average, the 90% of all memory references are performed over the 36% of the most referenced addresses. This behavior is closer to that of the SPECfp, which also exhibit low data reuse, than to the SPECint, which are characterized by very high data locality. The exception is the `mpeg2_dec` application, in which the 90% of the references are performed over the 7% of the most referenced memory locations.

4.4.5 Memory Hierarchy

Cache size and memory latency

We have evaluated the memory behavior for different data cache sizes. Table 4.10 shows the obtained hit rates for load and store operations separately. In spite of the low data reuse reported in previous section, very high hit rates demonstrate that data caches are very effective for multimedia applications, even for low cache sizes.

	Load Hit Rate				Store Hit Rate			
Benchmark	16K	64K	256K	1024K	16K	64K	256K	1024K
jpeg_enc	99.37%	99.99%	100.00%	100.00%	94.53%	99.71%	99.89%	99.89%
jpeg_dec	99.72%	99.99%	100.00%	100.00%	95.02%	99.86%	99.92%	99.92%
mpeg2_enc	99.88%	99.90%	99.92%	99.99%	96.65%	96.78%	96.66%	98.04%
mpeg2_dec	99.59%	99.75%	99.85%	99.99%	98.48%	98.97%	99.10%	99.20%
gsm_enc	100.00%	100.00%	100.00%	100.00%	99.99%	99.99%	99.99%	99.99%
gsm_dec	100.00%	100.00%	100.00%	100.00%	99.99%	99.99%	99.99%	99.99%
epic_enc	98.26%	98.59%	99.40%	100.00%	70.25%	69.30%	66.37%	66.02%
epic_dec	94.73%	95.94%	97.71%	100.00%	74.57%	75.57%	74.85%	78.62%
average	98.94%	99.27%	99.61%	100.00%	91.18%	92.52%	92.09%	92.71%

Table 4.10. Hit rate of load and store operations for different cache sizes

This can be explained by the fact that the spatial data locality is more emphasized than the temporal data locality in streaming data access patterns. Spatial data locality is still higher in audio applications (`gsm_enc` and `gsm_dec`), whose main kernels process one-dimensional data structures, and besides, the same data stream but with a small initial offset is processed in consecutive iterations. On the other hand, image and video applications tends to have two-dimensional spatial locality, which is more difficult to exploit by conventional data caches.

As it was stated by Lee et al. [LPMS97], it can also be observed that data caches are more effective for loads than for stores. This makes sense as input streams usually have more temporal locality than the output stream. EPIC exhibit lower store hit rates than the other applications due to the way it is programmed. While other benchmarks, like JPEG, do not need a full-image buffer, EPIC allocates both input and output full-images.

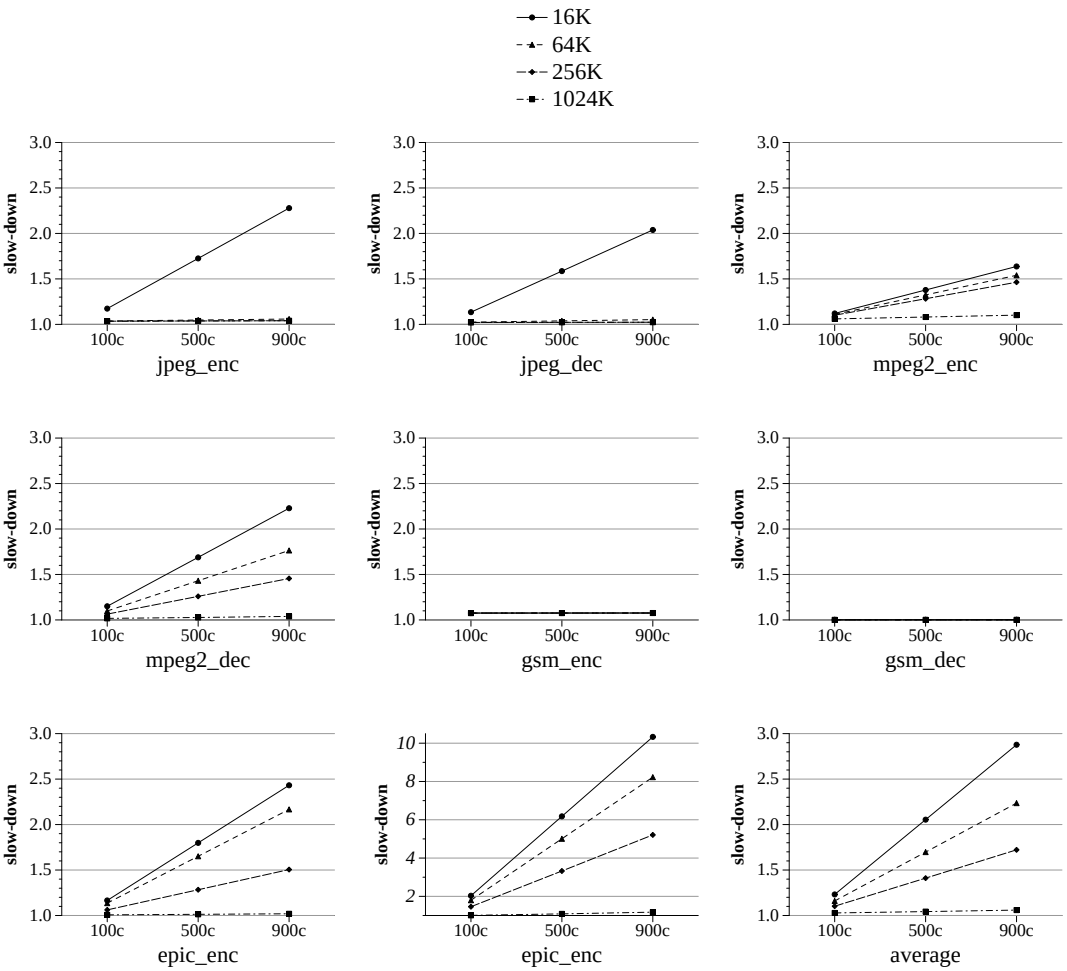


Figure 4.3. Slow-down of a real memory hierarchy vs perfect memory for different cache sizes and memory latencies

Figure 4.3 shows the performance slow-down due to memory stalls for different cache sizes and main memory latencies. For clearness, the vertical scale of the `epic_dec` graph is more than three times greater than for the other benchmarks.

The JPEG and GSM applications exhibit very low cache size requirements, even for long latencies to main memory. The MPEG2 video and EPIC applications require higher cache sizes to compensate for long main memory latencies. In all cases, a 1MB cache is large enough to guarantee very low slow-downs due to memory stalls, even for very long latencies to main memory.

Memory ports

Multi-porting a cache enlarges the overall area of the memory array considerably. It also has a great impact in access time and power consumption. Another alternative to

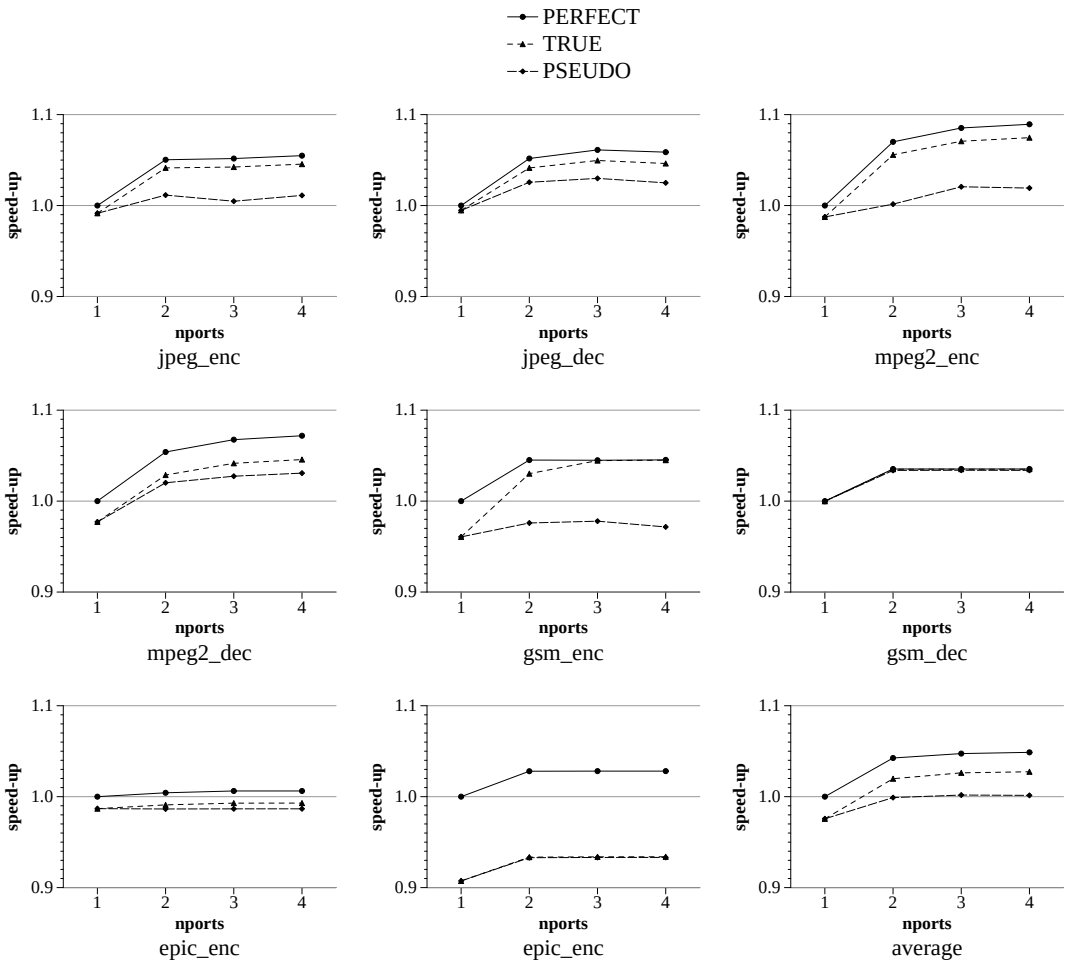


Figure 4.4. Performance speed-up for different memory ports configurations vs 1-port perfect memory

support multiple accesses is to divide the cache in independent banks, each mapping a different address space. This banking model is able to provide simultaneous memory accesses as long as the references are to different banks. However, banking also adds the decoding overhead of routing each address to the right bank and detecting collisions.

The main issue of multi-banked memories are the effect of the *bank conflicts*. While a bank conflict does not necessarily result in a processor slow-down in dynamic scheduling architectures, in our model of VLIW architecture, a bank conflict means one stall cycle of penalty. This effect could be reduced by using scheduling algorithms more sensitive to data storage in memory (like trying not to schedule references to consecutive elements in the same cycle).

Figure 4.4 shows the performance speed-up obtained when the number of ports is increased. The solid line represents the *perfect* case in which there are not memory

stalls (all memory accesses are executed with the latency they were scheduled). The dashed line includes realistic memory hierarchy simulation in a *true* multi-ported system (so many simultaneous memory accesses as the number of ports). Finally, the long dashed line shows performance speed-up for *pseudo* multi-ported caches (so many simultaneous memory accesses as the number of ports as long as the references are to different cache banks). It was assumed the same cycle time can be achieved for all designs. All speed-ups are referred to the one port perfect memory configuration.

Performance trade-offs of true multi-ported caches help to determine cache ports requirements. Results do not show a significant improvement when increasing the number of ports above two. On the other hand, the true multi-ported configuration performs closer to the perfect memory than to the pseudo multi-ported one. This confirms that bank conflicts are an important source of memory performance degradation (more than fifty per cent of the overall memory stalls are due to bank conflicts). Increasing the number of ports also increases the potential for bank conflicts. In `gsm_enc`, for example, the three-ports configuration outperforms the four-ports one because of the negative effect of the increase in the number of bank conflicts.

4.4.6 Operations per Cycle

Finally, Table 4.11 shows the average operation per cycle rates. The OPC rates in the innermost loops have been replicated from Table 4.7 for comparison purpose.

	Loops			Application		
Benchmark	2-issue	4-issue	8-issue	2-issue	4-issue	8-issue
jpeg_enc	1.64	2.12 (+29.00%)	2.22 (+ 4.75%)	1.41	1.71 (+21.47%)	1.74 (+ 1.61%)
jpeg_dec	1.64	1.87 (+14.37%)	1.93 (+ 2.70%)	1.58	1.84 (+16.26%)	1.88 (+ 2.59%)
mpeg2_enc	1.72	2.62 (+52.40%)	3.57 (+36.41%)	1.63	2.35 (+44.17%)	2.92 (+24.38%)
mpeg2_dec	1.55	1.75 (+12.81%)	1.77 (+ 0.82%)	1.57	1.86 (+18.50%)	1.90 (+ 1.85%)
gsm_enc	1.69	2.56 (+51.52%)	3.28 (+27.84%)	1.68	2.32 (+37.79%)	2.56 (+10.38%)
gsm_dec	1.36	1.47 (+ 7.64%)	1.46 (- 0.69%)	1.37	1.47 (+ 7.83%)	1.46 (- 0.56%)
epic_enc	0.78	1.05 (+34.71%)	1.06 (+ 0.51%)	0.83	1.01 (+21.75%)	1.01 (+ 0.50%)
epic_dec	1.27	1.40 (+10.17%)	1.42 (+ 1.37%)	1.12	1.23 (+ 9.98%)	1.24 (+ 1.05%)
average	1.46	1.86 (+26.58%)	2.09 (+ 9.21%)	1.40	1.72 (+22.22%)	1.84 (+ 5.23%)

Table 4.11. Operations per cycle rate in innermost loops and applications for different issue widths

The OPC achieved in the complete applications is slightly lower than in the loops, and exhibit less potential to scale with the way of the architecture. The exceptions are `mpeg2_dec` and `gsm_dec` applications, which are also the benchmarks with lowest coverage of modulo scheduling loops. This shows the relevance of software pipelining techniques like modulo scheduling to exploit the parallelism of loops in VLIW architectures.

4.5 Summary

In this chapter we have evaluated the main characteristics of multimedia applications. These applications are usually composed by a set of kernels that process streams of data like different stages in a pipeline. Results show that most of the benchmarks exhibit low data reuse. However, the streaming data access patterns promote spatial locality, which leads to very high cache hit rates, even for small cache sizes.

Several reasons contribute to the conclusion that caches with a small number of wide ports are preferable to caches with a large number of ports. First, both the percentage of memory operations (23% in average) and port requirements are low, and results do not show a significant improvement when the number of ports is increased above two for perfect memory simulation. On the other hand, multi-porting a cache is more expensive than widen the ports, and alternative feasible multi-banking cache designs entail the issue of bank conflicts. We have observed that bank conflicts are an important source of performance degradation in VLIW architectures, and they are potentially increased with the number of ports. Furthermore, as multimedia memory accesses are mostly unit-stride accesses to short arrays of small elements, wide accesses to memory seems a good option to be included in multimedia architectures. Packing several references to the same array into one wide access reduces both the number of memory access and the potential for bank conflicts.

Results also show that these applications exhibit more parallelism than integer ones. Software pipelining techniques, like modulo scheduling, arise as a key optimization to exploit instruction level parallelism in wide issue architectures. Nevertheless, this parallelism is not so high as it was to be expected from the definition of the algorithms. On the one hand, applications often include a lot of overhead to deal with different options and formats. On the other hand, some algorithms have been implemented with the objective of reducing the number of scalar operations, mainly costly operations such as multiplication, which contributes to hide the existing parallelism. Furthermore, small loop counters also difficult the use of conventional vectorization to exploit data level parallelism. MMX-like μSIMD vectorization arise as a good option to deal with the small data sizes, small loops, and unit-stride memory accesses. The performance of this kind of multimedia extensions will be studied in Chapter 6.

Chapter 5

Memory Disambiguation in Multimedia Applications

This chapter analyzes the problem of memory disambiguation in the context of multimedia applications and proposes a run-time memory disambiguation technique based in the specific behavior of multimedia memory access patterns. We perform a detailed evaluation of the approach, which has been completely implemented into the Trimaran compiler. We also compare it against an advanced interprocedural pointer analysis framework and analyze the possibility of using both of them together to improve performance.

5.1 Relevance of Memory Disambiguation

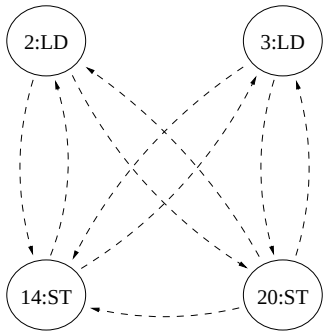
Ambiguous memory dependences often limit the ability of the compiler to detect the existing parallelism, thus preventing it from generating vector code. If there is *any* possibility that two memory operations ever reference the same memory location, the compiler must place dependence arcs between them to ensure they are executed in sequential order.

Multimedia applications share different traits with both numerical and integer applications. As in numerical applications, multimedia programs make extensive use of multi-dimensional data structures with relatively simple patterns. As in integer applications, multimedia applications make extensive use of pointers (since C and C++ are the languages of choice of multimedia code developers), sometimes with several levels of indirection to match the multimedia structures of standardized protocols.

At the same time, multimedia applications differ from these two wide fields in the characteristics of the data processing. As we saw in Chapter 4, multimedia applications are streaming; that is, typical multimedia kernels process one or more input streams of data to produce one or more output streams. Additionally, the input and output streams are typically disjointed regions.

```
h2v2_fancy_upsample (unsigned char **input_data,
                     unsigned char **output_data_ptr, ...)
{
    register unsigned char *inptr0, *inptr1, *outptr;
    register int thiscolsum, lastcolsum, nextcolsum;
    ...
        inptr0 = input_data[inrow];
        if (v == 0) inptr1 = input_data[inrow-1];
        else        inptr1 = input_data[inrow+1];
        outptr = output_data[outrow++];
        ...
        for (colctr = compptr->downsampled_width - 2; colctr > 0; colctr-) {
            nextcolsum = (int)(*inptr0++) * 3 + (int)(*inptr1++);
            *outptr++  = (unsigned char) ((thiscolsum * 3 + lastcolsum + 8) » 4);
            *outptr++  = (unsigned char) ((thiscolsum * 3 + nextcolsum + 7) » 4);
            lastcolsum = thiscolsum; thiscolsum = nextcolsum;
        }
    ...
}
```

(a) C source code



(b) Dependence graph

Figure 5.1. Source code and memory dependence graph of the innermost loop in the h2v2_fancy_upsample function

Techniques to detect aliasing between access patterns of array elements are quite effective for many numeric applications. However, although multimedia codes usually follow very regular memory access patterns, current commercial compilers remain unsuccessful in disambiguating them due mainly to complex pointer references. By way of illustrating, figure 5.1.a shows a code fragment of the upsampling algorithm in `jpeg_dec`. It performs linear interpolation between pixel centers, also known as a triangle filter. The centers of the output pixels are 1/4 and 3/4 of the way between input pixel centers.

Cyc	Op0	Op1	Op2	Op3	Op4	Op5	Op6	Op7
<0>	1:ADD$_1$	4:ADD$_1$	8:SHL$_1$	10:ADD$_1$	19:ADD$_1$	20:ST$_2$	21:ADD$_1$	22:MOV$_1$
<1>	2:LD$_1$	3:LD$_1$	9:ADD$_1$	24:ADD$_1$				
<2>	5:SHL$_1$	11:ADD$_1$						
<3>	6:ADD$_1$	12:SHR$_1$						
<4>	7:ADD$_1$	13:AND$_1$						
<5>	14:ST$_1$	15:ADD$_1$	23:MOV$_1$					
<6>	16:ADD$_1$							
<7>	17:SHR$_1$							
<8>	18:AND$_1$	25:BRF						

(a) Non-disambiguated modulo scheduling

Cyc	Op0	Op1	Op2	Op3	Op4	Op5	Op6	Op7
<0>	1:ADD$_1$	3:LD$_1$	4:ADD$_1$	7:ADD$_2$	10:ADD$_1$	11:ADD$_2$	18:AND$_3$	19:ADD$_1$
<1>	2:LD$_1$	12:SHR$_2$	15:ADD$_2$	20:ST$_3$	21:ADD$_1$	23:MOV$_2$	24:ADD$_1$	
<2>	5:SHL$_1$	8:SHL$_1$	13:AND$_2$	16:ADD$_2$	22:MOV$_1$			
<3>	6:ADD$_1$	9:ADD$_1$	14:ST$_2$	17:SHR$_2$	25:BRF			

(b) Disambiguated modulo scheduling

Figure 5.2. Non-disambiguated vs disambiguated code scheduling of the innermost loop in the h2v2_fancy_upsample function

The assembly code of the innermost loopbody has four memory operations, twenty integer arithmetic and logical operations, and one branch. All the memory operations have a stride of one; however, the initial addresses and the loop count are obtained from complex indirect references. As a result, the independence of the input and output streams cannot be probed at compile time, and the compiler must place memory dependence arcs between the two loads and the two stores. *Trimaran* also fails to disambiguate the two stores (see the memory dependence graph generated by *Trimaran* in Figure 5.1.b). Due to these false dependences, a vector compiler would not generate vector code for this loop.

Ambiguous memory dependences also limit the ability of the compiler to perform ILP-oriented code optimizations, which are crucial to make effective use of VLIW processors. In the example before, the potential loop-carried dependences from the two stores to the two loads prevent the compiler from generating an optimal modulo scheduling [Rau95]. Specifically, the initiation interval for a 8-issue width architecture is nine (you can see the code scheduling generated by *Trimaran* in Figure 5.2.a). However, if the compiler was able to disambiguate them, different iterations could be overlapped in a more efficient way; and, as a result, the same code would be executed more than twice faster (see code scheduling in Figure 5.2.b).

Based in the specific behavior of multimedia memory access patterns, we propose the *Dynamic Memory Interval Test* (DMIT). The DMIT is a run-time memory disambiguation technique that makes sense in the context of multimedia applications, or other kind of programs where input and output data streams are usually disjointed.

Disambiguation can be easily determined by dynamically analyzing the region domain of every load and store before each invocation of a loop. As we will see, significant gains are obtained at nearly no cost and without the inherent complexity of pointer analysis techniques.

5.2 Memory Disambiguation

Both static and dynamic memory disambiguation approaches have been proposed in the literature to determine if dependence actually exists for a pair of ambiguous memory references.

Static dependence analysis attempts to solve the ambiguity at compile time. On the other hand, dynamic memory disambiguation determines at run-time whether two memory operations reference the same location. The compiler provides different execution paths, and at run-time it is determined which one must be followed depending on the existence or not of the dependence.

Whether static, dynamic, or a combination of both is better depends on the particular kind of application being targeted and on the desired trade-off between performance and cost. Gallagher *et al.* investigate the application of both static and dynamic memory disambiguation approaches and provides a quantitative analysis of the trade-offs between the two approaches [GCM+94].

5.2.1 Static Dependence Analysis

Much work has been done to deal with multidimensional arrays and complex array subscripts [GKT91, MHL91, Fea91, PHP98]. However, these techniques are ineffective when the access pattern is non-linear or when some essential information, such as loop bounds, is not known at compile-time.

Pointer dereferencing is also one of the most important impediments to dependence analysis. *Pointer Alias Analysis* attempts to determine at compile-time when two pointer expressions refer to the same memory location. Due to the undecidability of this static analysis [Lan92, Ram94], existing approaches offer a trade-off between efficiency and precision. Although proposed interprocedural analysis techniques provide good pointer disambiguation, especially for pointer-intensive applications such as those of SPECint, they often increase compilation time and memory requirements.

A pointer analysis algorithm can be classified as *flow-sensitive* if it uses control-flow information during the analysis. On the other hand, it is *context-sensitive* if it distinguishes different caller contexts for a common callee. Several approaches are flow-sensitive and context-sensitive [LR92, CBC93, EGH94, WL95]; by contrast, other algorithms are flow-insensitive [And94, Ste96, SH97]. Qualitative comparisons among algorithms are difficult due to varying infrastructure, benchmarks, and per-

formance metric. An empirical comparison of the effectiveness of different pointer algorithms on C programs can be found in [HP00].

The pointer analysis used in this thesis employs a flow-insensitive but context-sensitive interprocedural algorithm which can handle all C features. *Pcode* interprocedural analysis [Gal95] determines what dependences exist with regard to global variables across function boundaries. This analysis also performs intraprocedural pointer disambiguation and dependence analysis, gathers alias and side effect information, and identifies targets of indirect function calls. This information is then merged back into the *Pcode* and is used by subsequent stages of the compilation.

5.2.2 Run-time Dependence Tests

Dynamic data dependence tests can be used to check at run-time whether two references access the same location. Dynamic memory disambiguation techniques usually require significantly less compile-time investment than static approaches, especially in languages such as C which require interprocedural analysis to provide high accuracy. Dynamic approaches are also more accurate than static ones, as they know the exact memory address being accessed by each reference during program execution.

The obvious downside of run-time tests is the overhead they introduce into the program. They usually require the insertion of extra instructions to check dependences. Some approaches also require new instructions and/or additional hardware support.

Nicolau was the first to introduce run-time memory disambiguation [Nic89]. He proposed a software data speculation technique that inserts explicit address comparisons and conditional branch instructions which allow memory flow dependences to safely be removed. Huang *et al.* proposed speculative disambiguation, a combined hardware and software technique to allow aggressive code reordering using predicated instructions [HSS94].

A different point of view is to consider the problem of deciding if a loop is fully disambiguated or not, that is, determining whether or not there is a dependence in any iteration [BCM94]. The *Privatizing DoAll Test* [RP94], for instance, identifies fully parallel loops at run-time and dynamically privatizes scalars and arrays; significant speed-ups were obtained on Fortran loops running on multiprocessor architectures. Other sophisticated approaches exist that produce predicates that may be used either at compile time or at run-time depending on whether there is enough information available [MH99, PW98].

The run-time test proposed in this thesis identifies a type of ambiguous dependences commonly found in multimedia applications. Explicit operations are inserted to compute and check the address space of each memory operation before the execution of the loop. It requires no instructions or hardware support, and thus can be applied to any existing architecture.

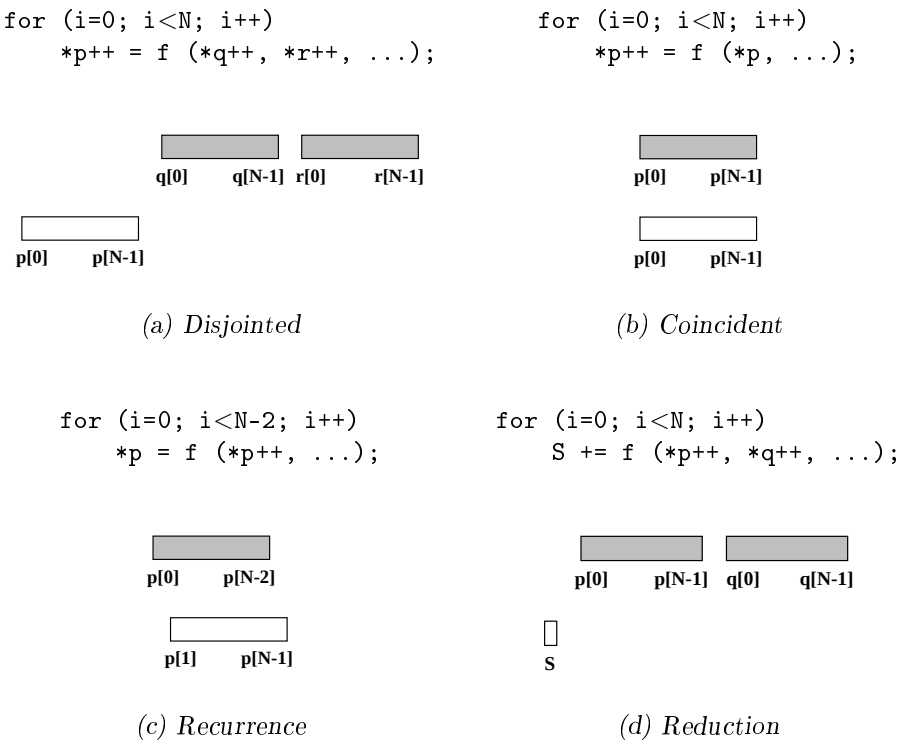


Figure 5.3. Typical multimedia memory access patterns

The concept of calculating non-intersecting data access ranges was probably first explored in [BE94], and later expanded by [PHP98], to handle symbolic array subscripts in scientific applications at compile-time. Our work differs from previous works by observing that, in multimedia loops, the indexing functions are so simple that data access ranges can be easily computed at run-time. The intersecting or non-intersecting of these ranges cannot be determined at compile-time mainly due to the use of pointers, but not because of the complexity of the indexing functions.

5.2.3 The Alias Analysis Problem in Multimedia Loops

As it has been said before, array references in multimedia applications usually follow strided and very simple access patterns. Figure 5.3 summarizes the kind of loops commonly found in these codes. The loop in (a) operates over one or several streams to produce a disjointed one, thus no memory dependence exists. Nevertheless, when these arrays are accessed through pointers, as is usual in multimedia codes, an accurate interprocedural pointer analysis is required to ensure that no aliasing occurs. Such techniques are not generally included in common commercial compilers, so they must be conservative and place dependence arcs between the memory operations to ensure correctness.

Output dependences (dependences between two stores) and *anti dependences* (when a load precedes a dependent store) usually have little impact on the generated code, but *flow dependences* (when a store precedes a dependent load) tend to be a severe restriction for the compiler. In the example, the potential loop-carried dependences from the store in the iteration i to the loads in the iteration $i+1$ would probably restrict modulo scheduling techniques significantly (remember the `h2v2_fancy_upsample` code example in Section 5.1).

In loop (b), the input and output streams coincide. However, loop-carried dependences do not exist in this case either, as loads from iteration $i+1$ never refer the same memory location as stores from iteration i. The opposite case is shown in (c), where there is a recurrence with distance one. In this case, loads from iteration $i+1$ must not precede the stores from iteration i. The last case (d) shows a loop that operates on array elements and accumulates the result on a scalar variable S. A register will probably be assigned to the scalar, and dependences between two loads (*input dependences*) are not a problem, so there are not ambiguous memory dependences in that case.

In numerical applications, the identification of the array elements accessed by a particular reference is important for compiler optimizations. In contrast, we observe that memory references on multimedia loops are always dependent or non-dependent at all. In other words, we have found that in multimedia we have two main kinds of stream behavior: one where all the input and output streams are totally independent, and other one where the streams have recurrences between themselves. A cost-effective approach to perform memory disambiguation would just need to determine, for every loop, which case we are facing. Non-linear array indices or linked lists of data are not common in multimedia loops. The main limitation to our approach is the use of non-streaming (sparse) data structures to perform computations via memory tables.

Reference groups

References with similar array index functions that differ only in the constant term (like $A[i]$, $A[i+1]$ and $A[i+2]$ in the code example in Figure 5.4.a) are also frequent in multimedia loops. These memory access patterns are known in the literature as *uniformly generated references* [WL91] or *reference groups* [CMT94]. Moreover, when the input and the output reference groups are the same (like in the example), we call them *coincident reference groups*.

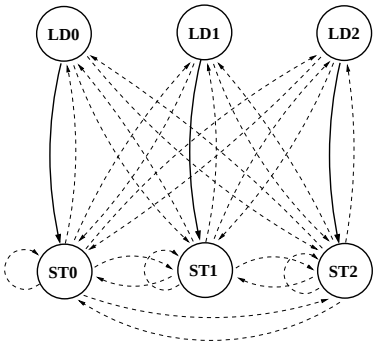
In a reference group, two references with different constant term are independent inside each iteration. On the other hand, if the stride of the variable term is greater or equal to the maximum difference between the constant terms, loop-carried dependences do not exist either. Thus, all dashed arcs in the dependence graph in Figure 5.4.b. can be safely eliminated.

```
for (i=0; i<N; i+=3) {

    ...= A[i];
    ...= A[i+1];
    ...= A[i+2];

    A[i]   = ...;
    A[i+1] = ...;
    A[i+2] = ...;
}
```

(a) C code



(b) Dependence graph

Figure 5.4. Example of coincident reference groups

5.3 The Dynamic Memory Interval Test

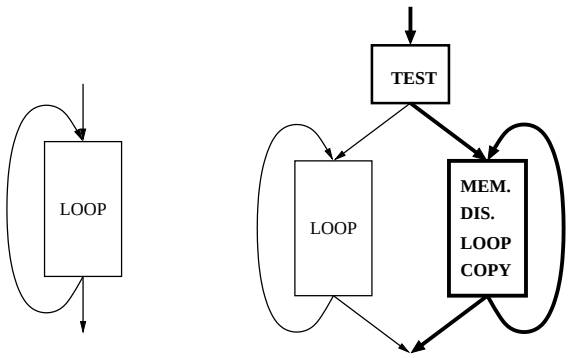
5.3.1 Description

The *Dynamic Memory Interval Test* (DMIT) is a software only mechanism based on the multimedia memory access patterns described in Section 5.2.3. The compiler generates both disambiguated and non-disambiguated versions of the loop, and inserts a simple test block before the loop that decides at run-time which one must be executed (see Figure 5.5). This decision is made by computing and comparing the lower and upper memory addresses that will be accessed by each stream. Complex pointer references or unknown parameters, such as loop bounds, prevent the compiler from making the decision at compile-time.

The test block is executed once on each invocation of the loop. In most cases, reducing the length of the disambiguated loop schedule will compensate for the already low overhead involved in the calculation of the intervals. Otherwise, the penalty introduced by this block (if it turns out that the original loop is executed) is minimal, and has no relevant impact on performance.

5.3.2 Terminology

We define the *Dynamic Memory Interval* (DMI) of a memory reference as the memory space delimited by the lower and upper locations accessed by that operation during one invocation of the loop. Figure 5.6.a shows the DMI of a memory reference with a stride of S inside a loop of N iterations. The shadow boxes represent the memory addresses that are actually accessed. Following this terminology, if we are able to prove before entering a loop that the DMIs of two references do not overlap, we can ensure that they are independent in that invocation. Note that scalar references are also included in this definition, as they are in fact references with a stride of zero.



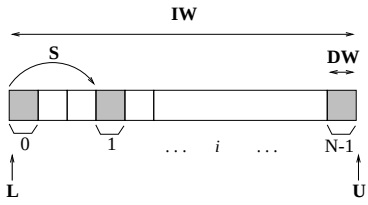
(a) Original control flow *(b) New control flow*

Figure 5.5. Dynamic Memory Interval Test

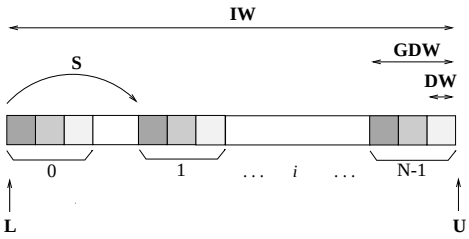
In the case of a reference group, if we were to apply the test between each pair of references inside the group, they would fail, as their individual DMIs overlap. When the compiler detects a reference group, it builds only one DMI for all the group. The stride of the reference group is the same as that of each individual reference. The *group data size* is the size of the memory space traveled by the different references of the group on each iteration (see Figure 5.6.b).

The following notation is used in the figures and algorithm description:

- TB: test block
- LB: loopbody
- Ref: memory reference
- AR_i: address register of Ref_i
- S_i: stride of Ref_i (in bytes)
- DW_i: data size of Ref_i (in bytes)
- N: number of loop iterations in current invocation
- L_i: the lowest location referred by Ref_i in current invocation
- U_i: the next location to the highest one referred in current invocation
- IW_i: size of the memory region between L_i and U_i
- E_{ij}: dependence arc from Ref_i to Ref_j



(a) Single memory reference



(b) Memory reference group

L: lower bound **DW: data width** **N: number of iterations**
U: upper bound **GDW: group data width** **S: stride**
IW: interval width

Figure 5.6. Dynamic Memory Interval representation

5.3.3 Implementation

Main algorithm

This section describes the main features of the implementation. The main algorithm is shown in Figure 5.7. Solving ambiguous memory dependences becomes especially profitable to software pipelining techniques such as modulo scheduling, where just one ambiguous loop-carried memory dependence is enough to prevent the compiler from overlapping different iterations of the loop. In this study, we consider only loops that are targeted with modulo scheduling by the baseline compiler.

The first step consists on building the list of testable memory dependences. Memory dependences in which one of the two references is neither strided nor loop invariant are discarded, as their DMIs cannot be computed before each execution of the loop. Dependences between references which can be statically determined to refer the same location are also excluded, as they are definitely dependent.

The length of the test block should be controlled not only because of performance, but also because it increases the register pressure. For our study, we have used a simple heuristic that limits the maximum number of dependences to be tested. Loop duplication is also avoided if it does not reduce the minimum initiation interval for modulo scheduling.

```
foreach (LB_l) do
  if (is_modulo_scheduling(LB_l)=false) then
    continue
  endif
  foreach (E_ij in LB_l) do
    if (is_strided(Ref_i)=false and is_invariant(Ref_i)=false) then
      continue
    endif
    if (is_strided(Ref_j)=false and is_invariant(Ref_j)=false) then
      continue
    endif
    if (is_static_dep(E_ij)=true) then
      continue
    endif
    if (is_static_indep(E_ij)=true) then
      delete_memdep(E_ij)
      continue
    endif
    test_dep_list += E_ij
  enddo
  if (1 ≤ test_dep_list_size ≤ MAX_SIZE and
    check_MII_reduction(LB_l, test_dep_list)=true) then
      LB_c = create_loop_copy(LB_l)
      TB_l = create_test_block(LB_l, LB_c, test_dep_list)
      foreach (E_ij in test_dep_list) do
        del_memdep(E_ij, LB_c)
      enddo
  endif
enddo
```

Figure 5.7. DMIT. Main algorithm

If any pair of memory references still remains in the list, the compiler duplicates the loop and inserts the test block. This block contains the operations needed to test each of the selected dependences. Finally, dependences in the list are removed in the disambiguated loop version.

The test block

Suppose an ambiguous memory dependence exists between two references whose DMIs are $[L_i, U_i)$ and $[L_j, U_j)$. Then, to ensure they are disjointed intervals, we must test that:

$$L_j \geq U_i \textbf{ or } L_i \geq U_j$$

where L_k and U_k can be computed in this way:

if $(S_k >= 0)$ **then**
  $L_k = AR_k$
  $U_k = AR_k + (N-1) * S_k + DW_k$
else
  $L_k = AR_k + (N-1) * S_k$
  $U_k = AR_k + DW_k$
endif

Note that it handles both positive and negative strides. In case of a reference group, the group data width is used instead of the data width. The group data width can be computed as the difference between the highest and lowest constant terms of the array index functions plus the data width. The stride and the data width are usually known at compile time, while the address register and sometimes the number of iterations are not.

The main steps to create the test block are summarized in Figure 5.8. The *insert_previous_ops* function takes into account the case in which the value of the register AR before entering the loop is not the value it will have when the memory operation is executed in the first iteration. This is the case when the address register is defined inside the loopbody before being used by the memory operation. In that case, the compiler must also insert an equivalent copy of the define operation before the bounds computation in order to get the right value of AR. In this copy, the register is renamed to avoid modifying the real value. Note that *insert_previous_ops* is a recursive function, as it must now ensure data dependences are maintained for each operand of the define operation. The function *insert_interval_computation_ops* creates the low level products and additions to compute L and U, and *insert_compare_intervals_ops* inserts the operations to compare these limits. Finally, *insert_branch_op* inserts the conditional branch.

At first sight, for each pair of intervals to be compared, we would need two products, six adds and two compare operations. Such a quantity of operations could become prohibitive as the number of dependences to be tested increases. However, they are actually reduced if we take into account some trivial considerations. For instance, intervals with the same stride share a single product. Furthermore, if a memory reference must be compared with more than one other, the interval bounds are computed just once, so that only the compare operations are added.

5.3.4 Code Example

As a case of study, we will describe the generation of the test block for the innermost loop of the `h2v2_fancy_upsample` function in Figure 5.1. This case also proves the relevance of detecting reference groups. Without grouping, the two stores would produce two single DMIs with a stride of two and data width one byte, and each one should be compared with the DMIs of all other references. However, if the compiler

foreach *(E_{ij}* **in** *test_dep_list)* **do**
  if *(is_dominator(define_op(AR_i, LB_l), Ref_i)=true)* **then**
    insert_previous_ops(TB_l, LB_l, Ref_i)
  endif
  if *(is_dominator(define_op(AR_j, LB_l), Ref_j)=true)* **then**
    insert_previous_ops(TB_l, LB_l, Ref_j)
  endif
  insert_interval_computation_ops(TB_l, Ref_i)
  insert_interval_computation_ops(TB_l, Ref_j)
  insert_compare_intervals_ops(TB_l, Ref_i, Ref_j)
enddo
insert_conditional_branch_op(TB_l, LB_l, LB_c)

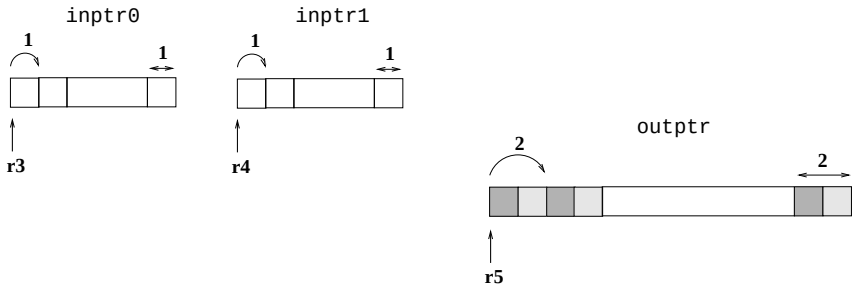
Figure 5.8. DMIT. Test block generation algorithm

detects the group, it will consider just one DMI with a stride of two and group data size two bytes (see Figure 5.9.a), saving an important number of arithmetic and compare operations. The impact is even greater for loops with large reference groups, such as the DCT computation, where the size of the groups is eight. More important is the fact that, in the first case, the DMI of the two stores would be compared with each other, and the test would fail.

The code of the test block created and inserted by the compiler is given in Figure 5.9.b. Let us assume that registers $r3$, $r4$, and $r5$ are the address registers pertaining to *inptr*0, *inptr*1, and *outptr* respectively, and the control register LC (*loop counter*) contains the number of iterations. Then, operations from 3 to 11 are inserted to compute the interval bounds ($r3$, $r4$, and $r5$ are the lower bounds and $r13$, $r14$, and $r15$ the upper ones).

Next, the compiler introduces the operations from 12 to 15 to check whether the DMIs overlap. To support predicated execution, the HPL-PD architecture [KSR00] provides 1-bit predicate register files and a rich set of compare-to-predicate operations which set predicate registers. We make use of these capabilities to generate the code of the test block. In the example, predicate registers are denoted as pn. The OR-compare operations (e.x., $p2 \mid = (r3 < r15)\ if\ p3$) write a 1 into the destination register ($p2$) only if both the predicate input ($p3$) and the result of the comparison are true. Otherwise, they leave the destination unchanged.

The conditional branch is performed in two steps. First, the prepare-to-branch (PBRR) operation loads the target address into a branch-target register (btr$n$). Second, the branch-conditional (BRCT) operation branches to the address contained in the btr$n$ operand if the branch condition (available in the specified predicate) is true. In the example, operation 2 sets the branch-target register $btr2$ to hold the



(a) Dynamic Memory Intervals

```
1:  p2 = 0
2:  btr2 = BB_50
3:  r4 = r2 + r17
4:  LC = LC - 1
5:  r7 = LC « 1
6:  r10 = r3 + LC
7:  r11 = r4 + LC
8:  r12 = r5 + r7
9:  r13 = r10 + 1
10: r14 = r11 + 1
11: r15 = r12 + 2
12: p3 = (r3<r15)
13: p4 = (r4<r15)
14: p2 |= (r5<r13) if p3
15: p2 |= (r5<r14) if p4
16: BRCT btr2 if p2
```

(b) Test block code

Figure 5.9. Test block code generated for the h2v2_fancy_upsample innermost loop

address of the non-disambiguated loop (BB_50), and operation 16 branches to it if the result of the comparisons is true.

5.4 Evaluation

The before described algorithm has been completely built into the *Trimaran* compiler. The original release of the compiler only performs intraprocedural analysis on low level code, which is quite representative of current commercial compilers. We have implemented a new compilation module into the *Elcor* back-end to do loop disambiguation. Loop disambiguation is performed at the intermediate code level just before any scheduling or register allocation (see Figure 5.10).

For the pointer analysis comparison, we have replaced the original *Impact* front-end by an internal release able to perform *Interprocedural Pointer Analysis* (IPA) [Gal95]. Therefore, both techniques are evaluated using the same compilation and simulation framework.

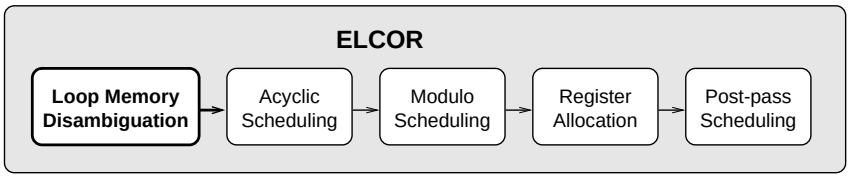


Figure 5.10. Incorporation of the Loop Memory Disambiguation module into the Elcor back-end

5.4.1 Coverage

Current implementation of the DMIT only applies to innermost modulo scheduling loops. This means that it cannot disambiguate multi-dimensional array accesses (except when the innermost loop has been fully unrolled). The algorithm could be extended to work on nested loops. However, this would increase the implementation complexity, which is one of the main advantages of our approach.

There are also some loops that have no potential to be disambiguated, as they contain no store operations. This is the case, for example, of the main loop in the motion estimation algorithm of the `mpeg2_enc`, where the sum of absolute differences is computed for two arrays of 16x16 elements.

Table 5.1 shows the number and fraction of cycles and dynamic operations of innermost, modulo scheduling, and modulo scheduling loops with store operations for each application when they are executed in the 2-issue width architecture. Only loops that account for more than 0.5% of the overall program cycles are included. Loops in the last column are the input candidates to DMIT.

	Innermost			+Mod.Sched.			+Store Ops.		
Benchmark	#L	%Cyc	%Ops	#L	%Cyc	%Ops	#L	%Cyc	%Ops
jpeg_enc	6	47.96%	61.40%	6	47.96%	61.40%	6	47.96%	61.40%
jpeg_dec	4	82.85%	84.87%	2	25.52%	26.46%	2	25.52%	26.46%
mpeg2_enc	15	61.85%	76.47%	13	58.97%	74.71%	5	4.75%	1.91%
mpeg2_dec	11	35.49%	33.23%	7	10.31%	10.04%	6	9.18%	7.69%
gsm_enc	11	56.62%	73.68%	10	55.68%	73.04%	7	42.24%	33.60%
gsm_dec	5	91.92%	91.15%	3	5.11%	5.75%	3	5.11%	5.75%
epic_enc	12	53.25%	55.76%	7	39.20%	45.90%	1	2.18%	1.52%
epic_dec	12	69.22%	78.18%	7	47.73%	51.67%	5	45.64%	44.50%
sum/average	76	62.40%	69.34%	55	36.31%	43.62%	35	22.82%	22.85%

Table 5.1. DMIT. Coverage

Four of the benchmarks show low potential for improvement based on coverage issues. The rest of the benchmarks present a sufficient number of loops to optimize to give good performance improvements as a result of including memory disambiguation. It is important to note that our technique adds near-zero overhead over those codes

that could not benefit from memory disambiguation, thus overcoming the fact that there exist benchmarks without potential for improvement.

5.4.2 Loop Level Analysis

Table 5.2 shows the results of applying DMIT to the candidate loops for the 8-issue width architecture. It includes the coverage of the loop, the operations per cycle rate of the non-disambiguated and the disambiguated loopbodies, the cycle and operation count of the test block, the percentage of times the loop passes the test at run-time, and finally the overall speed-up achieved in the loop (including the test block overhead).

Benchmark	Loop name	%Cyc	LBbase OPC	LBdis OPC	TB Cyc	TB Ops	%Dis	Loop SP
jpeg_enc	_forward_D.9	22.77%	1.53	6.75	9	18	100%	4.28
	_rgb_ycc_c.5	14.07%	2.65	3.16	11	26	100%	1.19
	_forward_D.6	3.38%	1.49	3.20	9	12	100%	1.46
	_h2v2_down.4	2.58%	2.37	6.30	8	17	100%	2.64
	_jpeg_fdct.3	2.73%	4.30	4.55	–	–	–	1.00
	_jpeg_fdct.5	2.44%	4.91	5.90	5	6	100%	1.16
jpeg_dec	_ycc_rgb_c.5	17.12%	1.70	1.98	12	23	100%	1.17
	_h2v2_fanc.8	8.40%	2.46	5.83	9	16	100%	2.36
mpeg2_enc	_iquant_no.5	1.26%	1.08	1.08	9	19	0%	0.99
	_quant_non.3	1.12%	1.21	1.21	9	22	0%	0.99
	_quant_int.6	1.01%	1.10	1.10	9	22	0%	0.99
	_iquant_in.5	0.92%	1.09	1.09	9	19	0%	0.99
	_add_pred_.4	0.44%	1.70	3.67	8	21	100%	1.63
mpeg2_dec	_Add_Block.31	3.51%	1.59	3.64	8	15	100%	1.69
	_form_comp.58	2.18%	1.48	2.71	8	17	100%	1.54
	_Add_Block.36	1.57%	1.32	2.47	8	15	100%	1.50
	_form_comp.18	0.85%	1.96	2.63	8	12	100%	0.98
	_form_comp.38	0.52%	1.77	3.87	8	17	100%	1.76
	_form_comp.73	0.56%	2.05	3.34	8	27	100%	1.43
gsm_enc	_Short_ter.5	20.01%	2.79	3.63	9	14	100%	1.16
	_Autocorre.42	11.33%	1.11	6.87	7	11	100%	6.14
	_Weighting.3	4.62%	1.91	3.97	29	97	100%	1.97
	_Long_term.8	2.95%	1.09	6.04	8	25	100%	5.16
	_Gsm_Coder.5	1.48%	1.74	1.74	8	17	0%	0.98
	_Reflectio.52	1.19%	1.11	4.62	8	23	100%	3.40
	_Calculati.25	0.66%	2.45	7.36	8	12	100%	3.15
gsm_dec	_Gsm_Long_.16	3.03%	2.32	6.84	8	17	100%	2.95
	_Gsm_Long_.24	1.56%	2.28	6.58	–	–	–	1.00
	_Gsm_Decod.5	0.53%	1.19	5.86	8	12	100%	4.57
epic_enc	_quantize_.11	2.18%	0.71	4.75	9	14	100%	6.70
epic_dec	_unquantiz.3	18.00%	0.86	4.63	9	15	100%	5.35
	_main.18	16.46%	0.94	5.67	9	12	100%	6.00
	collapse.9	8.74%	1.84	1.84	–	–	–	1.00
	_write_pgm.3	1.83%	3.50	7.00	8	14	100%	2.00
	collapse.191	0.60%	3.00	6.00	11	15	100%	2.00

Table 5.2. DMIT. Loop level analysis for the 8-issue width architecture

The test results support the assumption that multimedia loops are characterized by high amounts of parallelism. First, a high percentage of the loop candidates disambiguate (only 5 loops out of 32 fail the test). Furthermore, the result of the test is always the same in all invocations of the loops. On the other hand, 3 loops out of the 32 candidates do not require DMIT. In these cases, static disambiguation (coincident memory references and/or store reference groups detection) is enough to determine the dependence or independence of the memory references, without the requirement of a complex array dependence analysis. As the existing parallelism becomes visible to the compiler, the average operation per cycle rate in the loopbodies increases in a 138%.

As can be seen, common sizes of the test blocks range from 11 to 27 static operations, which are usually scheduled in 8 or 9 cycles. This code is executed only once on each invocation of the loop, and it is minimal compared with the reduction in the schedule length of the loopbody. In the `_Weighting.3` loop in `gsm_enc`, the compiler fails to detect a reference group of nine loads and the independence of the store operation is tested for each load operation, resulting in a very large test block of 97 static operations. But even in that case, the overall execution time of the loop is reduced in nearly 50%. On average, as a result of applying the DMIT we obtain a speed-up of 2.60X in the loops.

5.4.3 Applications Analysis

Figure 5.11 shows the performance speed-up obtained in complete applications for different issue widths, both with and without DMIT. All speed-ups are related to the 2-issue width architecture without DMIT.

Results show that memory disambiguation is a key technique to allow an effective exploitation of the available ILP when the architecture is scaled. In the original versions of code, increasing the issue width from 2 to 8 introduces an average performance speed-up of 1.29X. In sharp contrast, the disambiguated versions of code show higher performance improvements when scaling the reference machine, especially for those benchmarks with high coverage, and scaling from 2 to 8 produces an average speed-up of 1.40X. For the 8-issue width architecture, the DMIT exceeds the baseline performance in a factor of 1.13X.

On the other hand, we have observed a degradation of the memory behavior in the disambiguated versions. As an effect of increasing the parallelism, memory pressure also increases, and the number of bank conflicts grows up significantly. Moreover, as processor cycles go down, memory cycles become a greater percentage of the total execution time. For example, in the `epic_dec` benchmark, the 9.32% of the execution time is due to memory stalls in the 8-issue width baseline, and this percentage increases to 14.67% in the 8-issue width disambiguated version. These memory offset cycles make the speed-up decrease from an ideal 1.68X (without processor memory stalls) to the 1.55X shown in the graph.

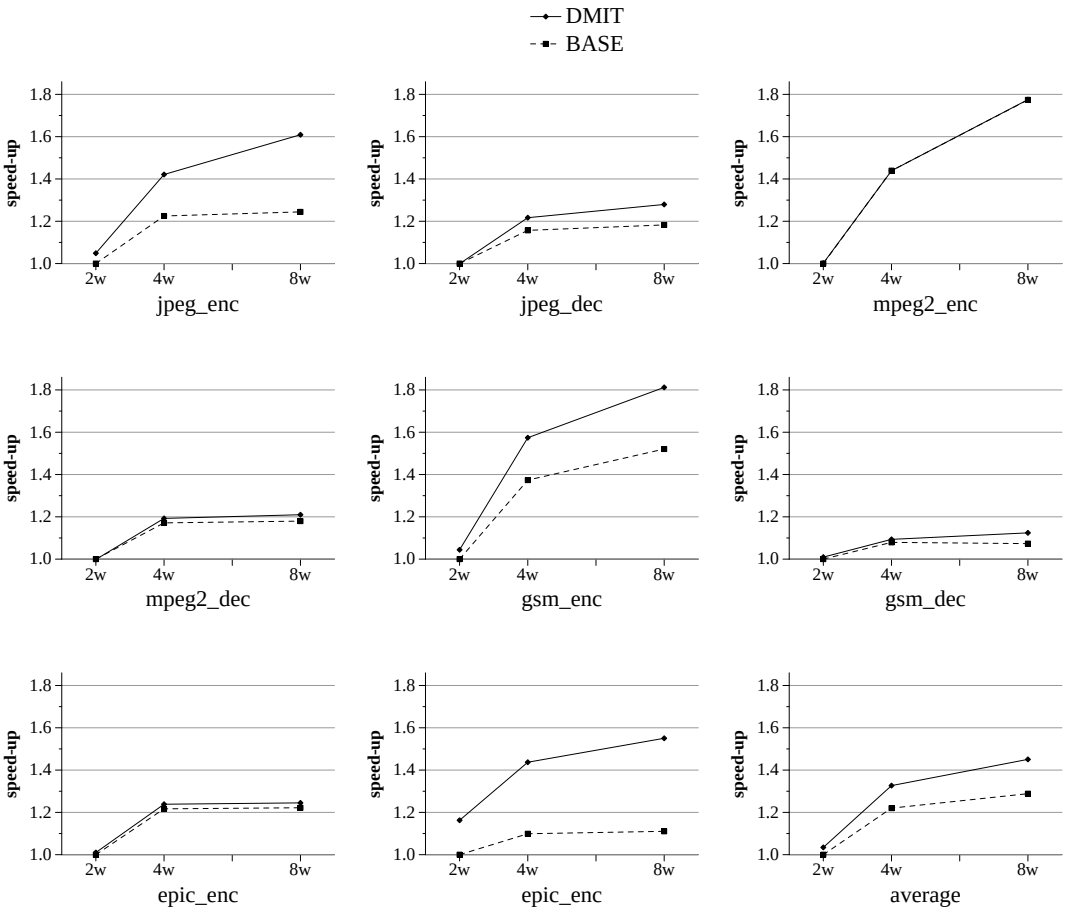


Figure 5.11. DMIT. Performance speed-up of 2-, 4- and 8-issue width architectures over the 2-issue width baseline

5.4.4 Test Block Overhead

The DMIT also involves an overhead in code size and execution time. Nevertheless, this overhead is negligible when compared with the rest of the code. For each application, Table 5.3 reports the average size in both cycles and operations, and what percentage of the loops and applications they mean. Results show that only 0.59% of the overall execution time is spent in test blocks. Even for the `mpeg2_enc`, a benchmark without coverage and in which four of the five candidate loops fail the test, there is not a relevant impact in performance.

As far as static code size is concerned, experimental results show that the duplicated loopbodies and the test blocks have an average size of 34 and 16 static operations respectively. The increase of the overall static code size ranges from 0.24% to 1.97% (0.83% in average).

	Cycles			Operations		
Benchmark	avg	%loop	%appl	avg	%loop	%appl
jpeg_enc	8.45	4.11%	1.20%	12.11	1.37%	0.75%
jpeg_dec	10.00	0.13%	0.03%	19.00	0.09%	0.02%
mpeg2_enc	8.28	1.88%	0.09%	20.86	3.74%	0.07%
mpeg2_dec	8.00	20.52%	1.28%	15.33	13.92%	1.24%
gsm_enc	9.32	7.12%	2.04%	16.39	3.34%	1.16%
gsm_dec	8.00	7.79%	0.07%	14.50	2.48%	0.08%
epic_enc	9.00	0.05%	0.00%	14.00	0.02%	0.00%
epic_dec	9.28	0.03%	0.00%	14.78	0.01%	0.00%
sum/average	8.79	5.20%	0.59%	15.87	3.12%	0.42%

Table 5.3. DMIT. Test block overhead

5.4.5 Comparison with Interprocedural Pointer Analysis

As complex pointer references is the main issue targeted by DMIT, it is of interest to compare it against advanced interprocedural pointer analysis techniques. Moreover, as they are not exclusive techniques, we also report the results obtained when using a combination of both; that is, static Pcode interprocedural analysis is first applied at the front-end, and then DMIT is used before the scheduling to disambiguate those loops that have not been previously disambiguated.

Loop level analysis

Table 5.4 shows the operations per cycle rate and the speed-up achieved at loop level by each compilation model. For the models that include DMIT, we also report the number of loops that require the dynamic test to be disambiguated.

We observe that DMIT achieves in general better results than IPA (1.20X speed-up over IPA in average), even though it requires lower implementation complexity. After interprocedural pointer analysis, the test block is avoided for 16 loops. However, loop duplication is still performed to 16 of the remaining loops, which means that dynamic information is still needed to determine the existence or not of the dependence in those loops. On the other hand, five loops achieve significant gains over pointer analysis without doing the test; these loops are examples of coincident references (case *b* in Figure 5.3) and/or store reference groups.

Furthermore, most of the benchmarks exhibit a beneficial effect when both techniques are used together. In `mpeg2_dec`, for example, the DMIT succeeds in disambiguating four loops (`_form_comp.x`) which pointer analysis does not, while pointer analysis is able to disambiguate another one (`_Add_Block.42`). This loop uses a table to perform saturation, and DMIT is unable to deal with this kind of non-strided references.

	Base	DMIT			IPA		IPA+DMIT		
Benchmark	OPC	T/L	OPC	SP	OPC	SP	T/L	OPC	SP
jpeg_enc	2.23	5/6	4.44	1.91	5.14	2.30	1/6	5.60	2.39
jpeg_dec	1.95	2/2	2.74	1.40	3.48	1.79	1/2	5.15	2.64
mpeg2_enc	1.18	5/5	1.24	1.03	1.19	1.01	5/5	1.25	1.04
mpeg2_dec	1.59	6/6	3.06	1.51	1.91	1.19	4/6	3.16	1.55
gsm_enc	2.04	7/7	3.90	1.75	3.20	1.55	2/7	3.91	1.75
gsm_dec	1.65	2/3	6.34	3.62	2.26	1.37	1/3	6.34	3.62
epic_enc	0.71	1/1	4.75	6.70	4.75	6.70	0/1	4.75	6.70
epic_dec	1.21	4/5	3.45	2.84	1.79	1.47	2/5	3.45	2.84
sum/average	1.57	32/35	3.74	2.60	2.96	2.17	16/35	4.20	2.82

Table 5.4. DMIT vs IPA. Loop level analysis for the 8-issue width architecture

On the other hand, five loops fail the test at run-time with and without pointer analysis. Their dependences were probably proved to be certain at the interprocedural pointer analysis phase, but this information is lost before DMIT, so that it can not differentiate between likely and certain dependences. Maintaining this information would be useful to avoid unnecessary tests.

Complete applications analysis

One advantage of IPA is that it is performed at the beginning of the compilation process, so that it can provide useful information to other phases of code optimization such as loop invariant code removal. On the contrary, DMIT is only applied to a fraction of the code and it only aids the scheduling process.

Figure 5.12 shows the speed-up obtained for the 2, 4 and 8-issue width architectures over the 2-issue width baseline. Although DMIT outperforms IPA in an average 16% in the targeted loops, these loops are only a 23% of the overall execution time. At the scope of the complete applications, the average gains obtained with IPA (1.04X, 1.34X and 1.46X) are very similar to those obtained with DMIT (1.03X, 1.33X, 1.45X). Moreover, the average speed-up increase when both techniques are used together (1.05X, 1.38X and 1.53X).

To facilitate the comparison, Figure 5.13 shows the speed-up achieved by the three options over the original compiler for the 8-issue width architecture. The benchmarks do not show a regular behavior. Although they perform similar in average, DMIT outperforms interprocedural pointer analysis for three of the eight benchmarks, but it does worse in the remaining five. More interesting are the additional gains obtained with the combination of both, specially for the `jpeg_dec` and the `mpeg2_dec` applications.

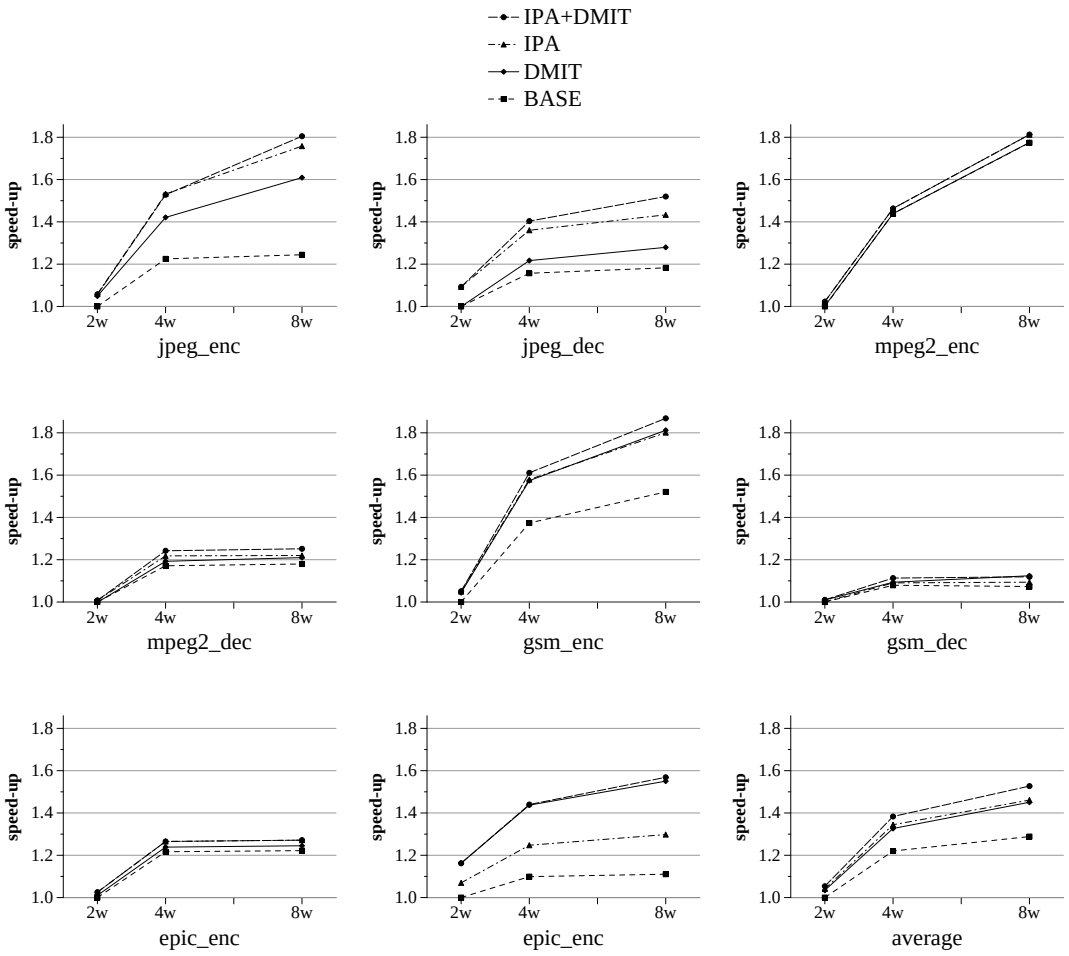


Figure 5.12. DMIT vs IPA. Performance speed-up of 2-, 4- and 8-issue width architectures over the 2-issue width baseline

5.4.6 Effect of DSP Oriented Scalar Optimizations

Due to the intrinsic significance of most multimedia algorithms, there has been a great effort focusing on reducing the overall number of required operations. Unfortunately, this effort has been oriented towards low-end DSP scalar architectures, hiding in most cases the data parallel nature of the original algorithm.

For example, the color conversion function (which stands for a 18% of `jpeg_enc` execution time) uses memory tables to perform multiplications. These table references cannot be disambiguated using DMIT, as they do not have strided patterns. A similar case occurs with saturation (clipping a result to a maximum/minimum value if it exceeds a given range), which is also implemented using memory tables in `jpeg_dec` and `mpeg2_dec`.

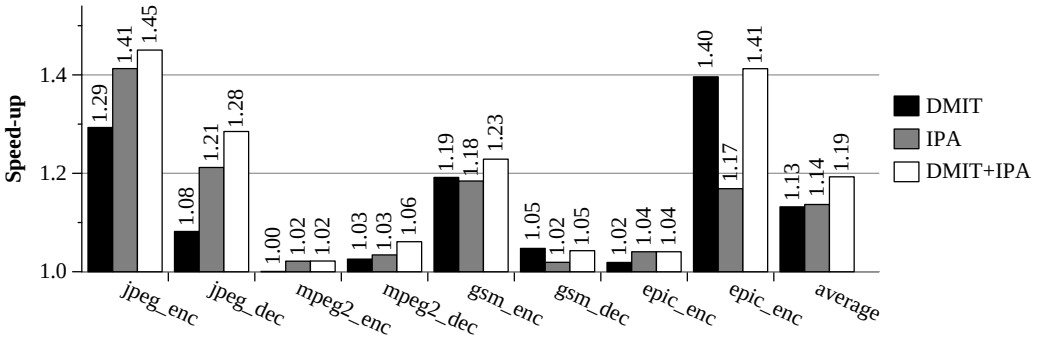


Figure 5.13. DMIT vs IPA. Performance speed-up over the 8-issue width baseline

Another typical scalar optimization is a break condition inside a loop. In the IDCT, for example, computation is avoided for those rows and columns whose elements are all zero. Nevertheless, on machines with fast multiplication, it is possible that the test takes more time than it is worth. Moreover, our compiler does not target the optimized code as modulo scheduling; thus producing worse code scheduling than the same code without the break condition.

We are interested in evaluating the performance of our technique when we revert to the original ways of performing the computation. Thus, we have analyzed the following explicit parallel versions:

- `jpeg_enc_dlp`: uses explicit products to perform color conversion instead of the tables.
- `jpeg_dec_dlp`: inverse color conversion and saturation are implemented without tables, and the zero condition of the inverse DCT has been removed.
- `mpeg2_dec_dlp`: saturation is implemented without tables and the zero condition of the inverse DCT has been removed

Figure 5.14 compares performance of DMIT, IPA, and the combination of both over the base compiler for the 8-issue width architecture. As can be observed, the performance results leveraged by our technique have improved significantly. Especially noticeable are the significant improvement of DMIT over IPA in `jpeg_dec_dlp` and the results leveraged by the combination of both techniques in `mpeg2_dec_dlp`.

5.5 Summary

Memory disambiguation of multimedia applications is compromised by the fact that they are often written in languages that support pointer referencing, such as C or C++. In this chapter, we have evaluated a simple but efficient memory disam-

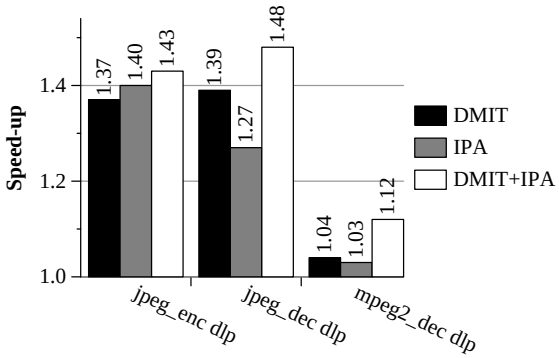


Figure 5.14. DMIT vs IPA. Performance speed-up over the 8-issue width baseline for explicit parallel versions of code

biguation technique specifically targeted at multimedia loops, or any other kind of applications with similar memory access patterns.

Taking into account the disjointed behavior of common multimedia memory streams, our algorithm is able to evaluate at run-time whether or not the full loop is disambiguated and execute the corresponding loop version. By calculating at run-time the dynamic memory intervals of every memory reference in a very efficient way, we avoid having to perform comparisons inside every loop iteration.

In contrast with other dynamic approaches, the Dynamic Memory Interval Test does not require any additional hardware or instructions. It has negligible effects over compilation time and code size, and near-zero cost for all those loops without potential for disambiguation. Nevertheless, one current limitation of this analysis is the inability to deal with non-streaming data structures.

Experimental results also confirm that memory disambiguation is a key technique for exploiting the inherent parallelism of multimedia applications. The Dynamic Memory Interval Test provides significant performance gains in most of our benchmarks. Furthermore, it allows performance scalability of wider-issue machines in sharp contrast with our baseline.

Although the Dynamic Memory Interval Test outperforms Pcode interprocedural analysis at the loop level, they perform similarly when we consider complete applications. This can be explained by the fact that pointer analysis has the advantage of being applied to the complete program code (not only to modulo scheduling loops), and at an earlier stage of the compilation, so that the alias analysis information can be used by further stages of the process. On the other hand, Pcode pointer analysis lacks array dependence analysis, which could be overcome with simple static optimizations (such as the detection of reference groups).

Furthermore, we have shown that a combination of both techniques provides improved results. There is a significant number of loops for which some information is missing at compile time, and they still benefit from Dynamic Memory Interval Test after interprocedural pointer analysis. On the other hand, the test overhead is avoided for those loops that can be statically disambiguated.

Chapter 6

A Vector-μSIMD-VLIW Architecture

In this chapter, we propose and evaluate adding vector capabilities to a μSIMD-VLIW core to speed-up the execution of the regions with data level parallelism, while, at the same time, reducing the fetch bandwidth requirements. We also discuss the main implications in the compilation process, and more specifically in the scheduling process. This enhancement has a minimal impact on the VLIW core and provides high performance with considerably less hardware complexity and power consumption than wider issue μSIMD architectures.

6.1 Scalar and Vector Regions

As it has already been stated, media kernels exhibit high amounts of DLP. Nevertheless, there is also a significant portion of code that is difficult to vectorize. That is some protocol related processing overhead such as first order recurrences, table look-ups and non-streaming memory patterns with large amounts of indirections. Therefore, a real media program is composed of heterogeneous regions of code with highly variable levels of parallelism: some of them with high amounts of DLP and the other ones with only modest amounts of ILP. We will refer to those regions that can be vectorized with the term of *Vector Regions* and to the remaining non-DLP regions of code with the term of *Scalar Regions*.

In the media domain, μSIMD-VLIW processors have been widely proposed [Gwe99, Sem99, Dev99, FBF+00], as they are able to exploit DLP by means of the μSIMD operations and ILP by the use of wide-issue static scheduling. Our claim is that, in media applications, the remaining non-DLP part of code is significant in terms of execution time and it exhibits only modest amounts of ILP, thus taking little benefit from increasing the processor resources. Even though VLIW processors are simpler than superscalar designs, very high issue rates also require decoding more operations in parallel and complicate the register files, which clearly increases access time and power consumption.

In order to evaluate the scalability of scalar and vector regions separately, we have marked the start and end point of the most computational intensive vector regions in the source codes. These regions generally correspond to one or two levels of nested loops plus some previous initializations. Table 6.1 lists the selected benchmarks, the parts of each program that have been considered as vector regions, and the percentage of the execution time they represent in a 2-issue width μSIMD-VLIW architecture.

Benchmark	%Vect	Vector Regions
jpeg_enc	29.56 %	R1: RGB to YCC color conversion R2: Forward DCT R3: Quantification
jpeg_dec	18.46 %	R1: YCC to YCC color conversion R2: H2v2 up-sample
mpeg2_enc	52.29 %	R1: Motion estimation R2: Forward DCT R3: Inverse DCT
mpeg2_dec	23.11 %	R1: Form component prediction R2: Inverse DCT R3: Add block
gsm_enc	18.66 %	R1: LTP parameters R2: Autocorrelation
gsm_dec	0.91 %	R1: Long term filtering

Table 6.1. Vector regions

Figure 6.1 shows the speed-up of 2, 4 and 8-issue width μSIMD-VLIW architectures over the 2-issue width μSIMD-VLIW. The dashed lines represent the speed-up in the vector/scalar regions over the vector/scalar regions of the 2-issue width architecture. The solid lines refer to the speed-up in the complete application.

From the graphs, it can be inferred that, except for the `gsm_enc`, the scalar regions fail to scale above 4-issue width. While increasing the width of the architecture from 2 to 4 provides an average speed-up of 1.24X in the scalar regions, moving from 4 to 8-issue only introduces a small 1.03X performance improvement. As far as the vector regions is concerned, they exhibit potential to benefit from wider issue scheduling, but this parallelism could be exploited in a more efficient way by conventional DLP oriented techniques. Furthermore, even though the vector regions scale up to 3.19X for the `jpeg_dec` application (2.49X in average), the vectorization percentage is low (24 % in average) and the lack of scalability in the scalar regions (1.28X in average) limits the performance of the complete application.

Results state that the actual performance achieved is very far from the theoretical peak performance and do not pay off the hardware complexity inherent in very aggressive architectures. We claim that Vector-μSIMD extensions arise as a better candidate to invest in, as they clearly reduce the fetch pressure, simplify the control

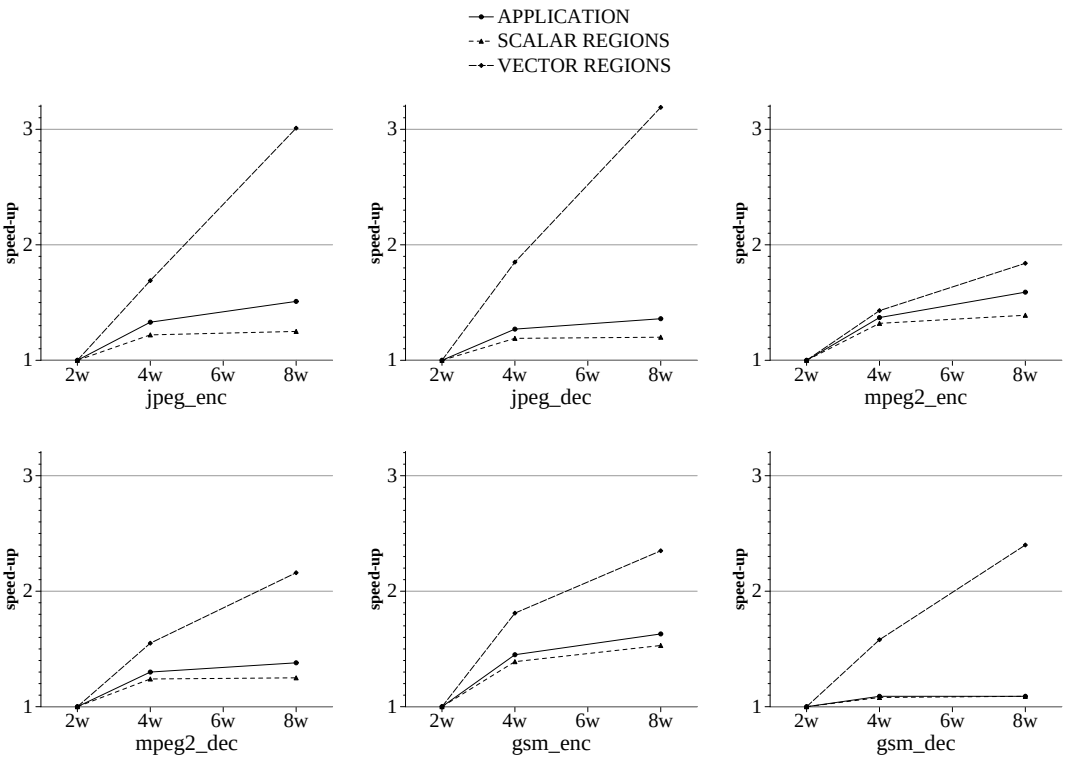


Figure 6.1. Scalability of scalar and vector regions in μSIMD-VLIW architectures

flow and memory access, and speed-up the performance of the vector regions without detrimental effects over the scalar part.

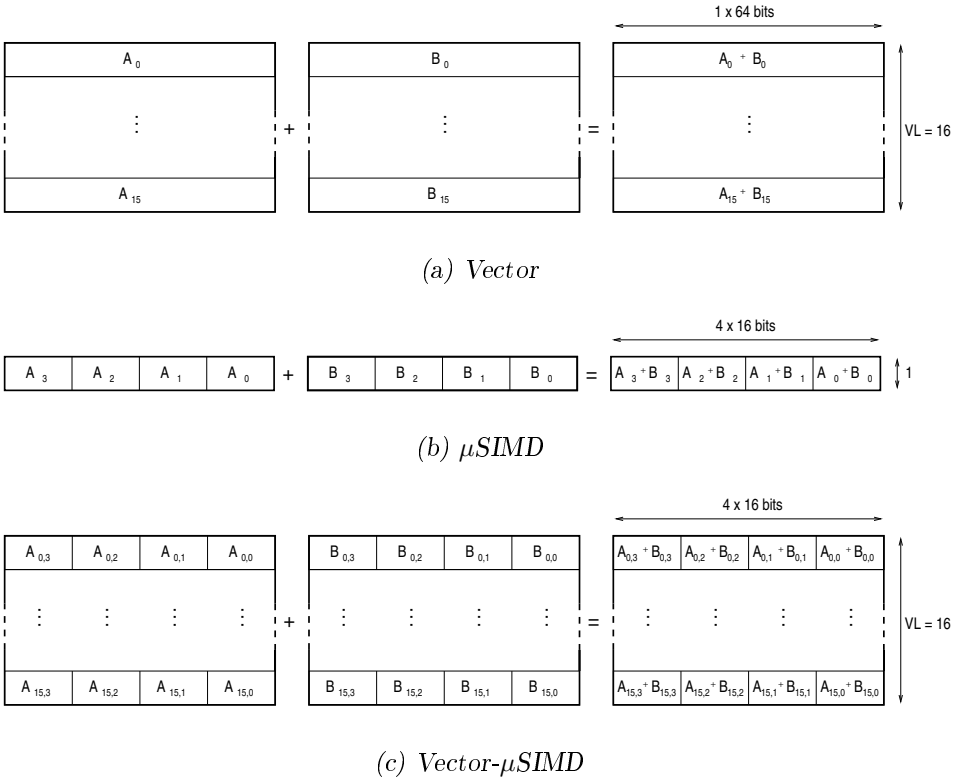
In [Cor02], a superscalar processor is enhanced with MOM, a matrix ISA extension that is basically an hybrid between conventional vector and MMX-like ISAs. We have used the same ISA to enhance our reference μSIMD-VLIW architecture. It must be stressed that additional issues arise mainly in the compiler side, as it must now be able to schedule vector operations.

6.2 Adding Vector Units to a VLIW processor

This section deals with the main implications of adding vector units to a μSIMD-VLIW processor. First, we overview the main features of the Vector-μSIMD ISA extension used for the study. Next, we describe the proposed architecture, including the datapath and the memory hierarchy. Finally, we discuss the main implications in the compilation, and more specifically in the scheduling process.

6.2.1 Vector-μSIMD ISA Overview

Our Vector-μSIMD ISA is based on the *Matrix Oriented Multimedia* (MOM) extension [CEV99]. It can be viewed as a conventional vector ISA where each operation



(a) Vector

(b) μSIMD

(c) Vector-μSIMD

Figure 6.2. Comparison between conventional vector, μSIMD and Vector-μSIMD ISAs

is a μSIMD operation (see Figure 6.2). It was designed to exploit the advantages of both conventional vector architectures (low fetch requirements, simple control logic and strided accesses) and μSIMD ISAs (sub-word level parallelism and multimedia oriented features such as saturation). It does not include costly vector operations, such as conditional execution, gathers or scatters.

It provides vector registers of 16 64-bit words each, vector load and vector store operations to move data from/to memory to/from the vector registers, and a set of computation operations that operate on vector registers. Since each word can pack either eight 8-bit, four 16-bit or two 32-bit items, each vector register can hold a matrix of up to 16x8, 16x4 or 16x2 elements. The architecture also provides 192-bit packed accumulators similar to those proposed in the MDMX multimedia extension [SIG97].

Additionally, two special registers are required to control the execution of vector operations: the vector length register and the vector stride register. The vector length register specifies how many words (out of 16) of the vector register are involved in the vector operation being performed. The vector stride register is specific to vector memory operations and dictates the distance between two consecutive words in the vector register.

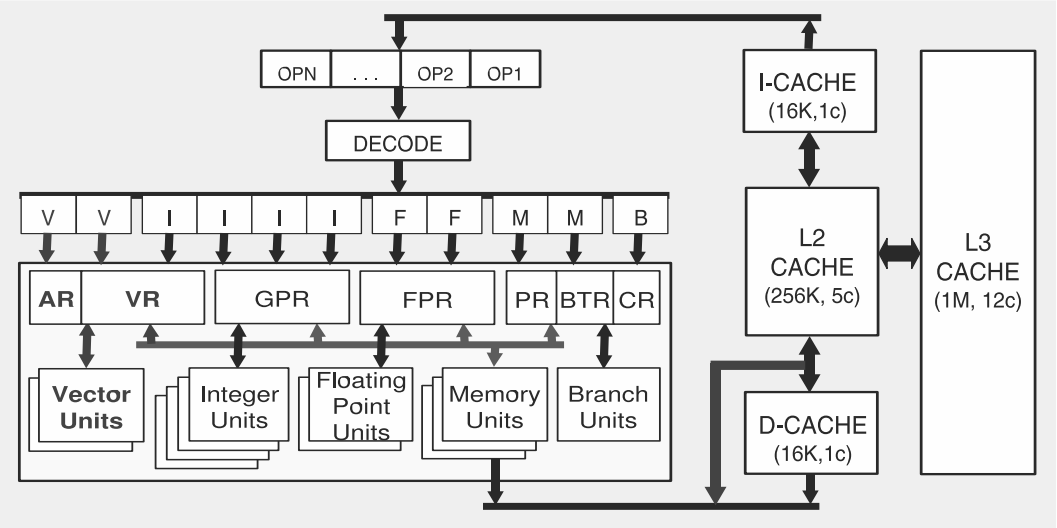


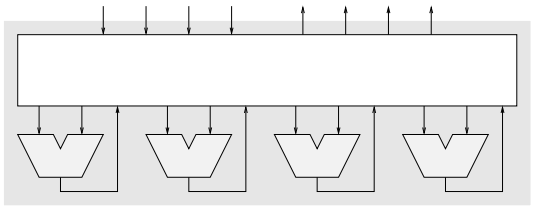
Figure 6.3. Vector-μSIMD-VLIW architecture

As far as terminology is concerned, we reserve the term *operation* to refer to each independent machine operation codified into a VLIW *instruction*. Each vector operation executes so many *sub-operations* as the vector length dictates. Finally, as the maximum vector length is 16 and each sub-operation can operate on either eight 8-bit, four 16-bit or two 32-bit items, a vector operation can perform up to 16x8, 16x4 or 16x2 *micro-operations*.

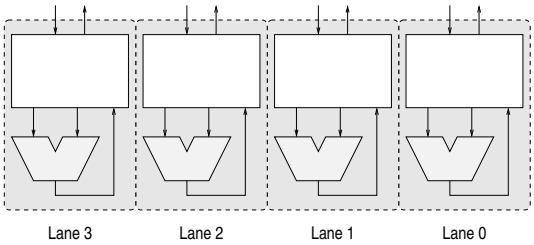
6.2.2 Vector-μSIMD-VLIW Architecture

Figure 6.3 shows the main components of the proposed architecture. Essentially, it is a VLIW processor with the addition of a vector register file, one or more vector functional units, and a modified cache hierarchy specially targeted to serve vector accesses.

Both, the vector register file and the vector functional units can be clusterized in independent vector lanes. This can be achieved with relatively simple logic by replicating the functional units, splitting each vector register across each lane and assigning each functional unit to a certain lane. The different elements of a vector register are interleaved across lanes, allowing all lanes to work independently. The architecture also includes a simple accumulator register file and adds limited connection between the lanes to be able to perform the last series of accumulation in a reduction operation. Only one of the lanes needs to read and write the source and destination packed accumulator. This lane is the responsible for performing the last reduction. In this work, we use four independent vector lanes; as our vector lengths are relatively short, a larger number of lanes would not pay off.



(a) Centralized register file in μSIMD(4 FUs, 4 L1 ports, 20 ports)



(b) Distributed register file in Vector-μSIMD(1 FU, 1 L2 port, 4 lanes, 5 ports/bank)

Figure 6.4. Comparison between centralized and distributed register file organizations

From the point of view of implementation, a vector register file scales better than a centralized one, due to the organization in lanes, which reduces the number of ports per cluster (see Figure 6.4). When scaling a centralized μSIMD register file, the register file storage and communication between arithmetic units become critical factors, dominating in area, cycle time and power dissipation of the processor.

Table 6.2 shows the characteristics of different μSIMD and Vector-μSIMD register files configurations. Register file area, delay and power have been estimated using the models described in [RDK+00]. Register file area is measured in square wire tracks (wt^2). Delays are given in units of fan-out-of-four inverter (FO4) delays[1]. A cycle time of 20 FO4 is assumed, which corresponds to a clock frequency greater than 500 MHz. Normalized values over the 2-issue with μSIMD configuration are also included.

As we can observe, for aggressive configurations, a vector register file can provide larger storage capacity with less area cost and access time. Thus, the proposed architecture appears as a good candidate not only in terms of performance, but also in terms of cost-efficiency.

As far as the memory hierarchy is concerned, we use a *vector cache* in the second level of the memory hierarchy (see Section 3.3.3 for further details). Scalar accesses are made to the L1 data cache, while vector accesses bypass the L1 to access directly

[1] An FO4 delay is less than 100ps for a 0.18μm process.

	μSIMD			Vector-μSIMD			
	2w2u	4w4u	8w8u	2w1v4	2w2v4	4w2v4	4w4v4
SIMD units	2	4	8	1x4	2x4	2x4	4x4
memory ports	1	2	4	1x4	1x4	1x4	1x4
SIMD registers	80	96	128	20	20	32	32
bits per register	64	64	64	16x64	16x64	16x64	16x64
number of lanes	1	1	1	4	4	4	4
banks per lane	1	1	1	1	1	1	2
ports per bank	8	16	32	5	8	8	8
Accumulator registers	0	0	0	4	4	6	6
bits per register	0	0	0	192	192	192	192
ports per bank	0	0	0	2	4	4	4
RF size (bytes)	640	768	1,024	2,656	2,656	4,240	4,240
RF area cost (wt^2)	675,840	2,334,720	10,321,920	1,497,600	2,746,368	4,389,888	4,389,888
RF access time (FO4)	10.31	12.17	15.80	9.71	10.31	11.31	9.86
RF peak power (fJ/FO4)	5,340	18,953	83,044	11,057	21,692	34,257	34,913
RF size (norm)	1.00	1.20	1.60	4.15	4.15	6.63	6.63
RF area cost (norm)	1.00	3.45	15.27	2.22	4.06	6.50	6.50
RF access time (norm)	1.00	1.18	1.53	0.94	1.00	1.10	0.96
RF peak power (norm)	1.00	3.55	15.55	2.07	4.06	6.42	6.54

Table 6.2. Estimated area, delay and power of different μSIMD and Vector-μSIMD register file configurations

the L2 vector cache. A coherency protocol based on an exclusive-bit policy plus inclusion is used to guarantee coherency.

6.2.3 Compilation Issues

The success of the proposed architecture is strongly dependent on the compiler. First, it must be able to generate Vector-μSIMD code. Second, it must perform the scheduling and register allocation for the new operations.

Vector-μSIMD code generation

Nowadays there are compilers that allow basic autovectorization for μSIMD architectures, and the same compilation techniques could be used to generate Vector-μSIMD code. In the case of short nested loops (typical in image and video applications), the vectorization process can be decoupled into two steps: first, generation of μSIMD operations over the inner loop, and second, conventional vectorization of those μSIMD operations over the outer loop. In the case of only one larger loop (such as those of audio applications), the process is in practice the same: first, unrolling the loop in a factor suitable to allow μSIMD vectorization, and second, conventional vectorization of the resulting loop.

As we do not have a reliable compiler at our disposal yet, we have used emulation libraries to hand-write μSIMD and Vector-μSIMD code to evaluate the approach. The compiler has been modified to replace the emulation functions calls by the corresponding operations.

Static scheduling of Vector-μSIMD operations

The scheduler is the module that needs the most detailed information about the target architecture, as it is responsible for assigning a schedule time to each operation, subject to the constraints of data dependence and resource availability. The new register files and functional units have been added to the machine description file. Flow analysis is then used to determine the dependence constraints between operations that define or use the same register. For every input and output operand, an earliest and a latest read and write latency must be specified respectively [AKR98].

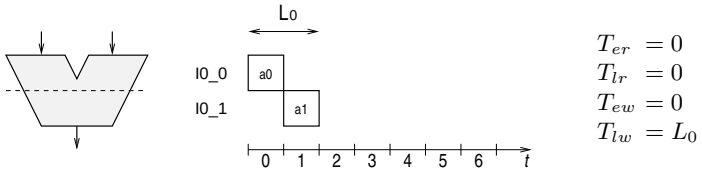
Figure 6.5.a depicts the execution of a 2 cycles fully-pipelined scalar operation. In this example, the source registers are read sometime during the first cycle after the initiation of the operation, and the result is written at the end of two cycles.

In the case of a vector operation, these values also depend on the vector length (VL) and on the number of parallel vector lanes (LN). As up to LN sub-operations are initiated per cycle, the last input operand will be read at $\lfloor (VL-1)/LN \rfloor$, and the last output will be written at $L + \lfloor (VL-1)/LN \rfloor$, being L the latency of one sub-operation (see Figure 6.5.b).

The number of parallel vector lanes is a fixed parameter from the architecture and it is known at compile time. On the contrary, the vector length is variable for each operation, and will be dynamically set. Nevertheless, the vector length register is usually initialized with an immediate value, and a simple data flow analysis is able to provide the right value to the compiler. In the few cases in which the vector length is not known at compile time, the compiler must assume the maximum vector length (16) in order to ensure correctness. Note that, for a vector unit with four parallel lanes, the penalty to pay would be three extra cycles at worst (that is, if the vector length turns out to be four or less).

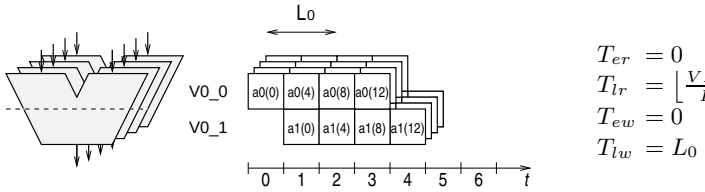
The same latency descriptors are taken for vector memory operations, but replacing the number of vector lanes by the width of the L2 port (in elements). In the proposed memory architecture, the execution time of a vector memory operation also depends on the stride. For simplicity, our compiler schedules all vector memory operations as having a stride of one and hitting in the L2 vector cache, and the processor stalls at run-time if either of the two assertions is not true.

On the other hand, providing a register file which supports concurrent accesses to the same vector register, the compiler can do *chaining* [Rus78] of two vector operations with a dependence on a vector register operand by just scheduling the second operation before the first one has completed execution. Assuming the same number



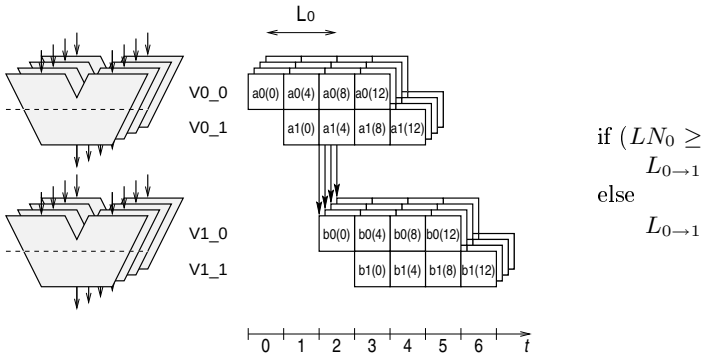
$$T_{er} = 0$$
$$T_{lr} = 0$$
$$T_{ew} = 0$$
$$T_{lw} = L_0$$

(a) Scalar operation



$$T_{er} = 0$$
$$T_{lr} = \left\lfloor \frac{VL_0 - 1}{LN_0} \right\rfloor$$
$$T_{ew} = 0$$
$$T_{lw} = L_0 + \left\lfloor \frac{VL_0 - 1}{LN_0} \right\rfloor$$

(b) Vector operation



if $(LN_0 \geq LN_1)$
$\quad L_{0\to 1} = L_0$
else
$\quad L_{0\to 1} = L_0 + \left\lfloor \frac{VL_0 - 1}{LN_0} \right\rfloor + \left\lfloor \frac{VL_1 - 1}{LN_1} \right\rfloor$

(c) Chaining

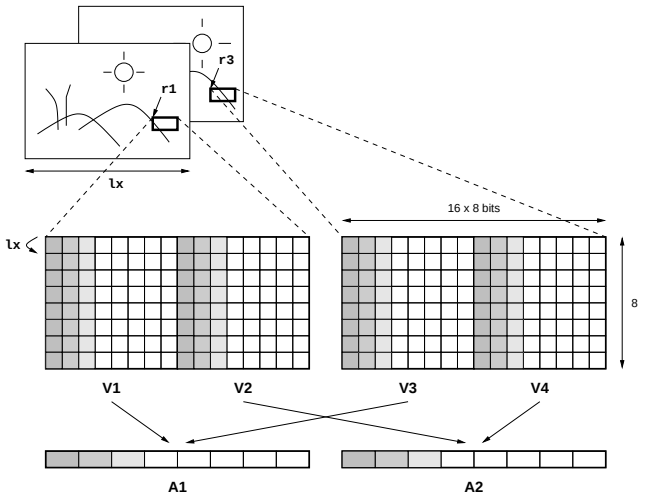
Figure 6.5. Latency descriptors (*Ter* = earliest read, *Tlr* = latest read, *Tew* = earliest write, *Tlw* = latest write, *L* = flow latency, *VL* = vector length, *LN* = vector lanes)

of lanes, the distance between the initiation of these operations must be at least L cycles. It is worth noting that no additional hardware is needed.

6.2.4 Code Example

As a case of study, we show the Vector-μSIMD code of the motion estimation kernel and the scheduling generated by our compiler. Motion estimation is one of the key elements of many video compression schemes. A video sequence consists of a series of frames. To achieve compression, the temporal redundancy between adjacent frames can be exploited. That is, a frame is selected as a reference, and subsequent frames are predicted from the reference using a technique known as *motion estimation*.

In the `mpeg2_enc` implementation of the algorithm, the current frame is divided into macroblocks, typically 16×16 pixels in size for the luminance component and 8×16 for the chrominance components. Each macroblock is compared to a macroblock




```
a: VL_MOV 8
b: VS_MOV lx
c: ADD r2, r1, 8
d: ADD r4, r3, 8
e: V_LD v1, r1
f: V_LD v2, r2
g: V_LD v3, r3
h: V_LD v4, r4
i: LVALACC_B a1, 0
j: LVALACC_B a2, 0
k: V_SAD_B a1, v1, v3
l: V_SAD_B a2, v2, v4
m: SSRCA_S_B r5, a1
n: SSRCA_S_B r6, a2
o: ADD r7, r5, r6
p: S_MOV_REG r7,s
```

Figure 6.6. Vector-μSIMD implementation of the motion estimation algorithm

in the reference frame using the sum of absolute differences (SAD) as error measure, and the best matching macroblock is selected. The search is conducted over a predetermined search area.

Figure 6.7 shows the Vector-μSIMD code of the motion estimation kernel that computes the SAD of two 8×16 blocks. It is assumed that registers $r1$ and $r3$ keep the initial address of each block, and lx (the image width) is the stride between consecutive rows. As the registers are 64 bit wide and the stride between rows is not one, we need two vector registers to hold each block. The SAD operation is implemented using a packed accumulator that allows parallel execution over the vector elements. Finally, the values packed in the accumulators are reduced and the final result is stored.

The corresponding scheduling is given in Figure 6.7. The target architecture is a 2-issue width VLIW processor with two integer units, two vector units with four parallel lanes, one port to the first level cache and a 4×64 bit port to the second level vector cache. Latencies are 1 cycle for the integer units and first level cache, 2 cycles for the vector units and 5 cycles for the vector cache.

As can be observed in the resource usage table, the Vector-μSIMD code of this kernel is memory bound. In fact, the second vector unit is not used at all, as the second SAD operation (l) must wait for the data being loaded from memory and cannot be scheduled earlier. Chaining is performed between two vector loads (g and h) and the vector SAD operations (k and l). Note also that the vector loads are scheduled as having a stride of one, that is, as if they produce four elements per cycle. As this assumption is not true, the processor will be stalled at run-time, thus incurring in a great penalty in performance, as we will see in the evaluation section.

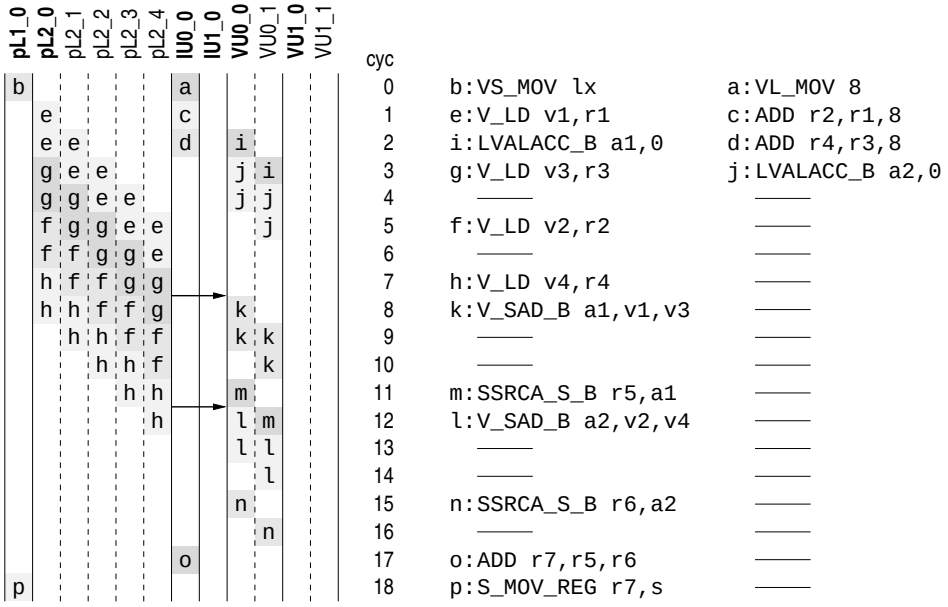


Figure 6.7. Scheduling of motion estimation for a 2-issue Vector-μSIMD-VLIW processor

We must highlight that the Vector-μSIMD code totally eliminates the two innerloops present in the scalar version to scan the blocks. Furthermore, the Vector-μSIMD code only needs to decode 16 operations to process one complete block, in front of the 172 operations required in the μSIMD versions of code.

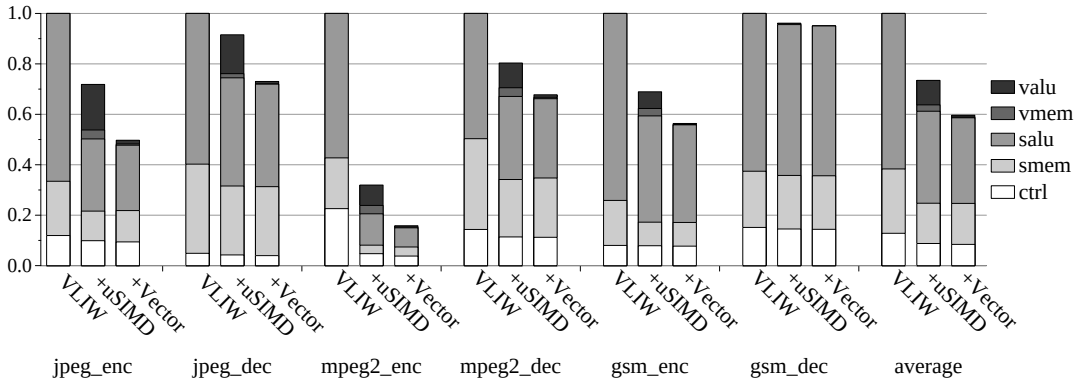
6.3 Evaluation

This section provides quantitative data in order to analyze the behavior of the proposed architecture. Different Vector-μSIMD-VLIW configurations are compared against μSIMD-VLIW and plain VLIW architectures. We must point out that the scalar versions of code include memory disambiguation, both *Pcode* Interprocedural Pointer Analysis and the *Dynamic Memory Interval Test* technique proposed in Chapter 5.

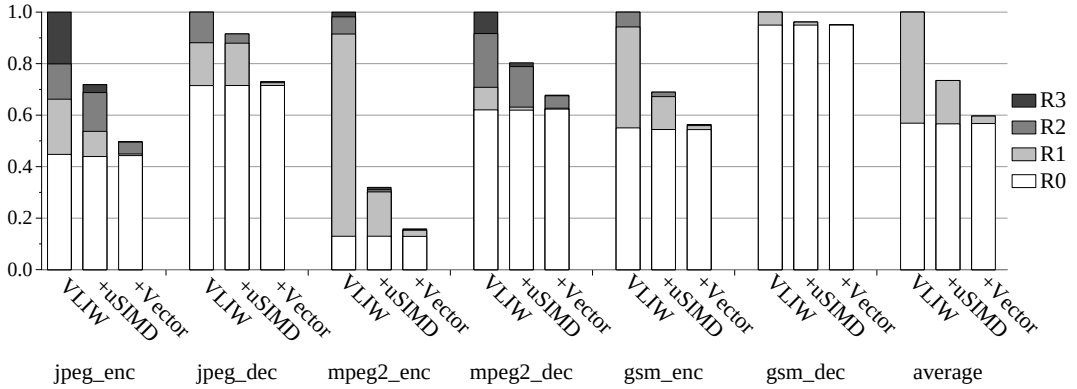
First, we evaluate the impact of the multimedia extensions in the overall number of operations. Next, we present performance results on the vector regions and analyze the influence of the number of vector units and lanes and the impact of the memory hierarchy. To end up, we report the speed-up and operations per cycle rates obtained in the complete applications.

6.3.1 Operation Breakdown

Figures 6.8.a and 6.8.b show the dynamic operation count for the different architectures (VLIW, μSIMD-VLIW and Vector-μSIMD-VLIW) normalized by the dynamic



(a) By type



(b) By regions

Figure 6.8. Normalized operation count

operation count of the base VLIW architecture. The first graph shows the operations classified into five categories: control, scalar memory, scalar arithmetic, vector memory and vector arithmetic. In the second graph, we have distinguished the contribution of each region. Regions from $R1$ to $R3$ are the fractions of code that have been vectorized in the μSIMD and Vector-μSIMD versions in the same order they are listed in Table 6.1 (for example, in `mpeg2_enc`, $R1$ accounts for the motion estimation and $R2$ and $R3$ for the forward and inverse two dimensional DCT). Region $R0$ always refers to the remaining scalar part.

The results confirm that the μSIMD and Vector-μSIMD versions of code require to execute much less operations than the scalar versions. This may not seem so obvious if we take into account that these versions are sometimes based on algorithms that require to execute much more operations [SCEV99]. For example, the μSIMD and Vector-μSIMD versions of the DCT are based in the matrix product, which requires significantly more operations than the optimized scalar algorithm. We must also point out that, in the scalar version of the `mpeg2_dec` benchmark, we are using

the Fast IDCT (a fast scalar algorithm) instead of the Reference IDCT (double precision matrix product algorithm) also included in the standard, as the former is ten times faster and we are interested in comparing against the best scalar version. On the one hand, the semantic richness of the μSIMD and Vector-μSIMD ISAs to perform operations such as the sum of absolute differences or saturation arithmetic contributes to decrease the operation count. Furthermore, there is an additional reduction on the number of operations involved in the loop-related control. This reduction in the number of operations to fetch and decode also translates into a decrease in power consumption.

As can be observed, the Vector-μSIMD architecture executes an average of 84% fewer operations in the vector regions than the μSIMD one (19% fewer in the complete application). The obvious reason is that Vector-μSIMD ISA can pack more micro-operations into a single operation (a maximum of 128 in the Vector-μSIMD in front of a maximum of 8 in the μSIMD). Table 6.3 reports the average vector length for each benchmark. VLx refers to the number of elements packed on one word. VLy corresponds with the vector length register, that is, the number of operations to perform in a vector operation, and it is always one in a μSIMD operation. Finally, $VLxy$ represents the overall vector length in a Vector-μSIMD operation, that is the product of VLx and VLy, or in other words, the number of micro-operations packed in one vector operation. Although most multimedia kernels are characterized by small loop counts, which usually results on low or moderate vector lengths in conventional vector architectures, the Vector-μSIMD ISA leverages quite fair micro-operations per operation rates (an average vector length of 81.10 micro-operations for the `jpeg_dec` application), due to its capability to vectorize two inner nested loops.

	+μSIMD	+Vector		
	VLx	VLx	VLy	Vlxy
jpeg_enc	3.55	3.88	7.59	28.47
jpeg_dec	5.11	5.08	15.96	81.10
mpeg2_enc	6.53	7.43	6.12	46.97
mpeg2_dec	3.57	4.23	3.97	17.46
gsm_enc	2.47	3.99	5.77	22.99
gsm_dec	3.36	3.22	10.61	35.67
average	4.10	4.64	8.34	38.78

Table 6.3. Average vector length

Finally, the reduction in the overall dynamic operation count depends also on the vectorization percentage, which is around 43% of the scalar code in average. As we already saw in Section 6.1, the exceptions are the `mpeg2_enc` and the `gsm_dec` applications. In the first one, the *motion_ estimation* and the DCT transforms account for the 87% of the overall dynamic operation count. On the contrary, `gsm_dec` exhibits a very low coverage (only 5% of the code has been vectorized). As can be seen in Figure 6.8.b, the Vector-μSIMD ISA achieves to reduce the number of operations

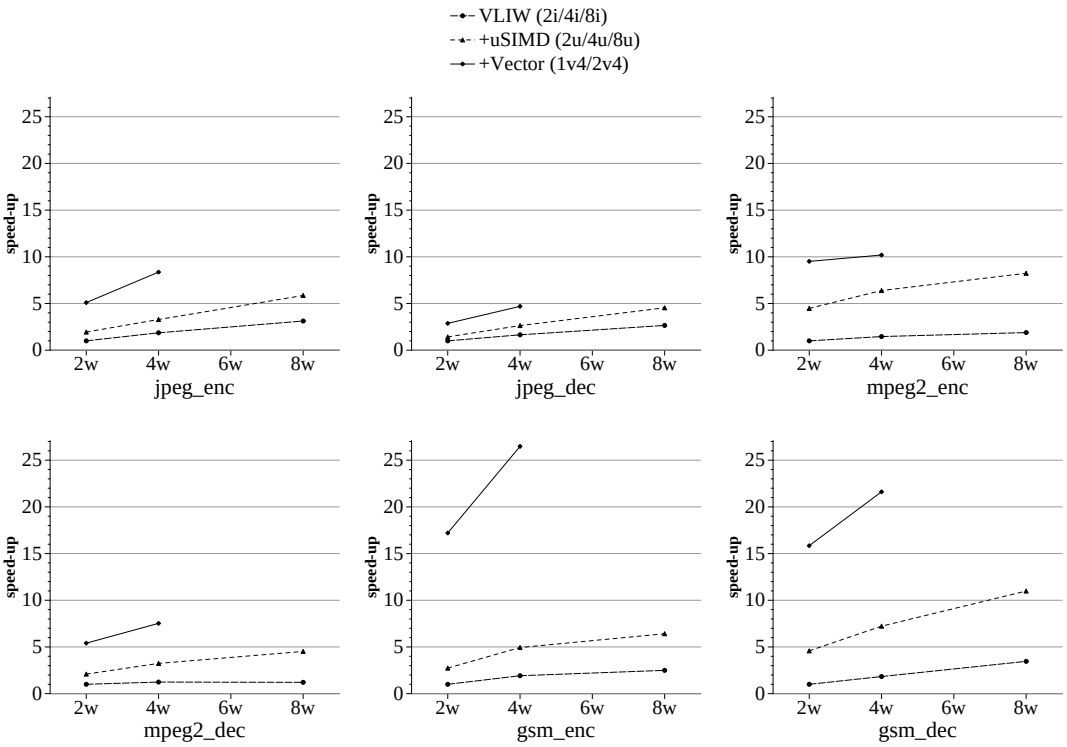


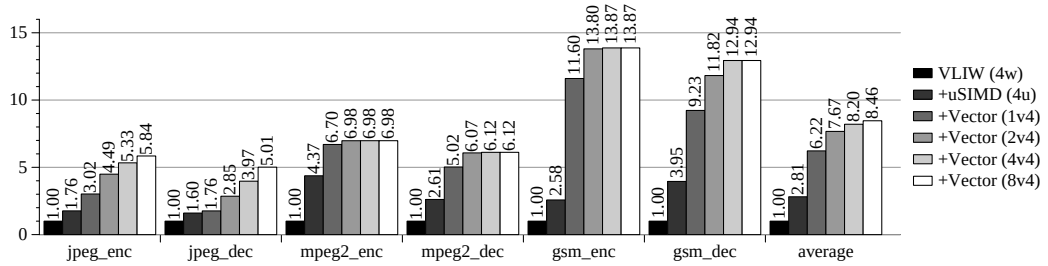
Figure 6.9. Speed-up in vector regions

of the vector regions to a minimum (less than 10% of the total dynamic operation count).

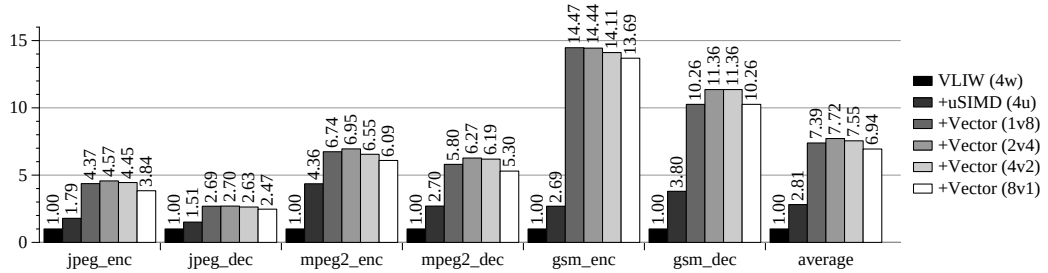
6.3.2 Speed-up in Vector Regions

We have evaluated the performance of 2- and 4-issue width Vector-μSIMD-VLIW architectures with one and two vector units and four vector lanes respectively. Results are compared against 2-, 4-, and 8-issue width VLIW and μSIMD-VLIW architectures with so many integer and μSIMD units as the issue width respectively. Attention must be paid to the fact that the Vector-μSIMD architectures are not balanced against the same issue width VLIW or μSIMD architectures, as we consider them as an alternative to wider issue processors. For example, the arithmetic capability of the 4-issue Vector-μSIMD configuration is comparable to that of the 8-issue μSIMD configuration, not to the 4-issue μSIMD.

For each architecture, Figure 6.9 shows the speed-up of the vector regions over the execution time of the vector regions in the 2-issue width VLIW architecture. As it was to be expected, both μSIMD and Vector-μSIMD architectures clearly outperform the same issue width VLIW. Moreover, the 2- and 4-issue width Vector-μSIMD architectures outperform the same issue width μSIMD in a factor ranging from 2.0X to 6.5X (3.2X in average) and 1.6X to 5.4X (2.8X in average) respectively. On the other hand, the 8-issue μSIMD architecture is outperformed by the 4-issue width



(a) Influence of increasing the number of vector units



(b) Number of vector units vs number of vector lanes

Figure 6.10. Speed-up in vector regions for different number of units and lanes

Vector-μSIMD in a factor of up to 4.1X (1.9X in average), with the same arithmetic capability and considerably less hardware complexity.

It is worth noting that, even the 2-issue width Vector-μSIMD architecture outperforms the 8-issue μSIMD architecture for most of the benchmarks, with half the arithmetic capability and four times less issue width. The exceptions are the `jpeg_enc` and `jpeg_dec` applications, which as we will see next, are characterized by having higher computational demand than other applications.

Number of vector units

To analyze the effect of increasing the number of vector units, Figure 6.10.a shows the performance improvement obtained in the vector regions when increasing the number of vector units from 1 to 8. The graph shows speed-up with respect to the execution of the vector regions in the 4-issue width VLIW. The 4-issue width μSIMD architecture is also included as a reference.

We observe that half of the benchmarks do not take much benefit from increasing the number of vector units. This is because they have vector regions similar to the *motion_estimation* code studied in section 6.2.3, with small loops and very short vector lengths. Examples of this include the *form_component_prediction* and the *add_block* regions in `mpeg2_dec` and the *calculation_of_the_long term_parameters* in `gsm_enc`.

On the contrary, other benchmarks such as the `jpeg_enc` and `jpeg_dec`, whose vector regions are characterized by larger vector lengths (ex. *color_ conversions* or *upsampling*) and/or larger loop sizes (ex. *DCT*'s), exhibit a significant improvement in performance when the number of vector units is increased.

Number of vector units vs number of vector lanes

The Vector-μSIMD architecture can be scaled not only in the number of vector functional units, but also in the number of vector parallel lanes. To analyze the trade-off between them, Figure 6.10.b shows the speed-up in the vector regions for different vector configurations over the 4-issue width VLIW architecture, but now keeping the overall computational capacity constant.

Results confirm that distributing the register file and units in four parallel lanes is a good choice for our set of benchmarks. Apart from increasing the area cost and power, reducing the number of lanes below four also results in performance degradation. This can be explained by the fact that there are not enough vector operations to be executed in parallel to feed a greater number of units. On the contrary, there are data dependences between operations, and a smaller number of lanes translates into a greater execution time for each operation. On the other hand, having more units than lanes benefits those operations that do not depend on the vector length, such as accumulator reductions, as they can then be executed in parallel.

Effect of the memory hierarchy

To analyze the influence of the memory hierarchy, Figure 6.9 shows the performance speed-up obtained in the vector regions with perfect memory simulation for the different architectures. By perfect memory we consider that all accesses hit in cache, but with the corresponding latency. That is, all scalar accesses are served after 1 cycle of latency and all vector accesses in the Vector-μSIMD configurations go to the L2 and take 5 cycles plus the additional cycles to serve all vector data elements (which slightly favours the VLIW and μSIMD-VLIW configurations). The shadow line represents the speed-up obtained with real memory simulation. All speed-ups are referred to the execution time of the vector regions in the 2-issue width VLIW architecture with perfect memory simulation.

We observe that the Vector-μSIMD architectures exhibit the highest performance degradations when considering a realistic memory system. This fact may seem counterintuitive, since vector architectures are well known for their capability to tolerate memory latency. Two reasons explain this behavior. First, the vector lengths are not long enough to take benefit from this characteristic. Second, VLIW architectures are very sensitive to non-deterministic latencies.

As it was explained before, in the scheduling the compiler assumes that all vector accesses have a stride of one, and the processor stalls at run-time if this assertion is

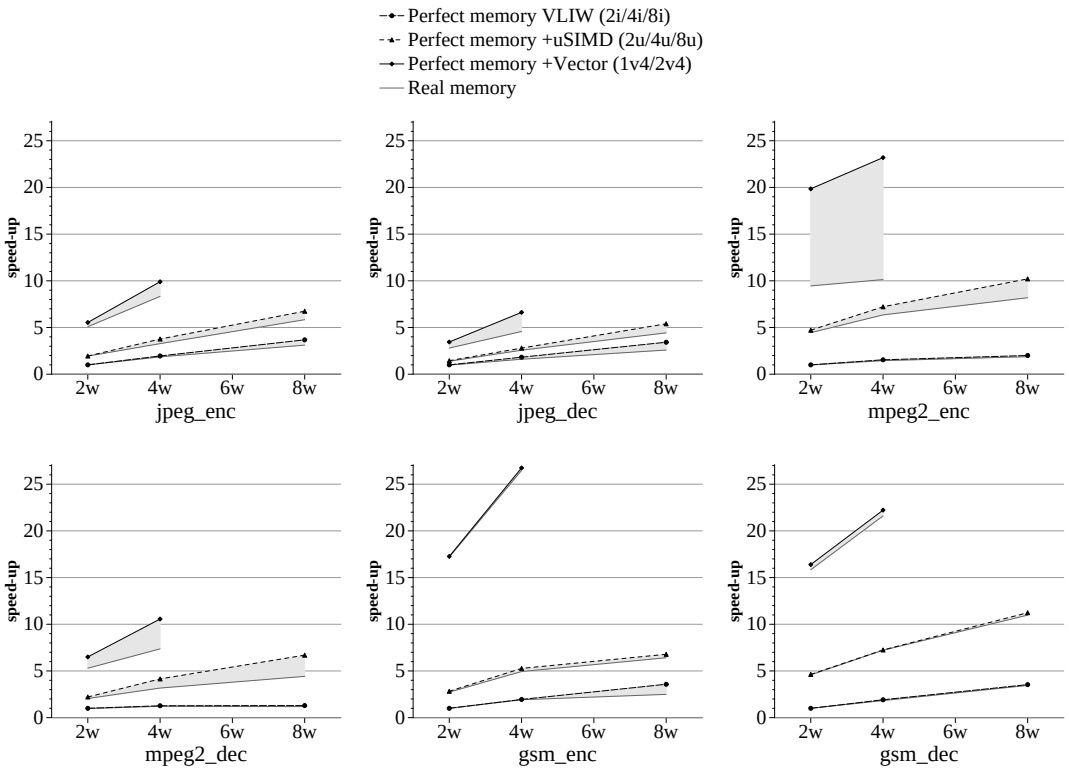


Figure 6.11. Speed-up in vector regions with perfect memory and impact of real memory

not true. That is what happens in the `mpeg2_enc` benchmark, in which the stride of the main region (the *motion_ estimation* code example analyzed in Section 6.2.4) is the image width. Moreover, in this kernel, these memory operations represent an important fraction of the overall code, resulting in a high performance degradation (close to 200%). Apart from this, all benchmarks exhibit high hit ratios and very low performance degradation when considering realistic memory.

6.3.3 Speed-up in Applications

Figure 6.12 shows the speed-up for complete applications. As it was to be expected, the benchmark that exhibits the highest performance improvement is the `mpeg2_enc` (up to 4.2X speed-up for the 4-issue Vector-μSIMD configuration). Even though there are other benchmarks (such as `gsm_enc`) with considerably greater speed-ups in the vector regions, the impact in the overall performance is not so significant, due to the low vectorization percentage. Note also that the 4-issue Vector-μSIMD architecture slightly outperforms the 8-issue μSIMD in all the applications.

It can also be observed that the gap between the different architectures decrease with the issue width of the processor. For example, while the 2-issue Vector-μSIMD exhibits an average factor of 1.22X of performance improvement over the 2-issue μSIMD, the 4-issue Vector-μSIMD only outperforms the 4-issue μSIMD in a 1.14X.

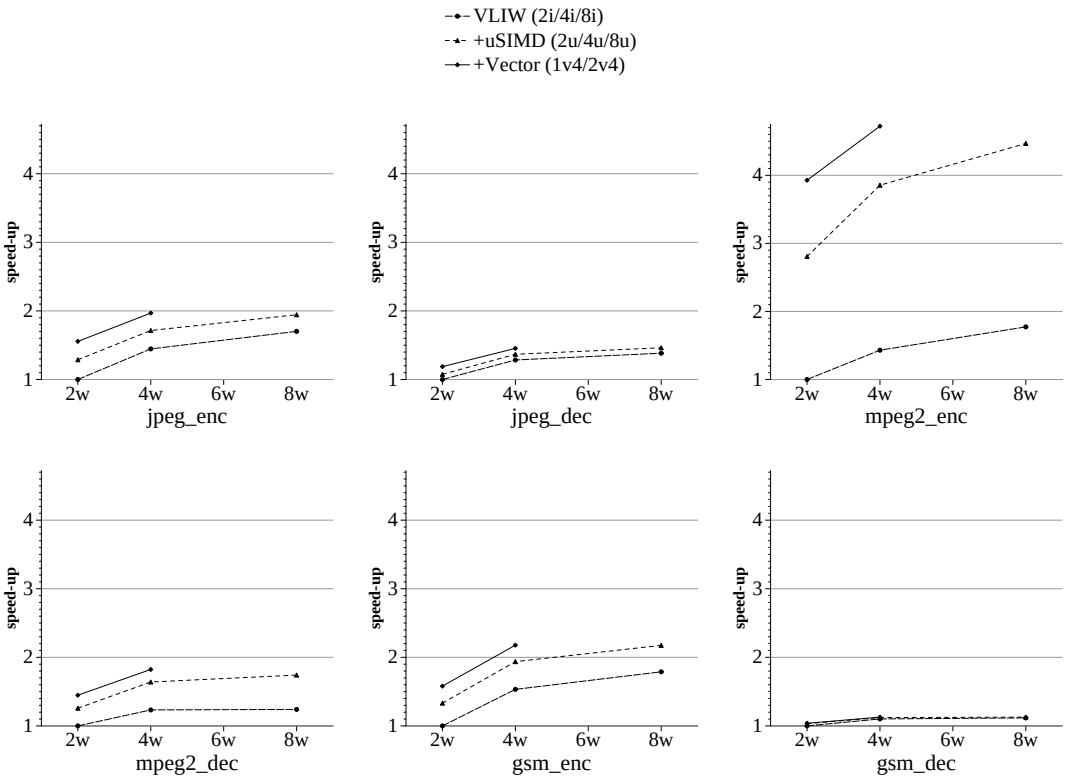


Figure 6.12. Speed-up in applications

This can be explained by the fact that a wide enough μSIMD-VLIW architecture is able to exploit as ILP the parallelism that the Vector-μSIMD-VLIW exploits as DLP.

On the other hand, the vector regions represent less than 40% of the total execution time in the 2-issue VLIW architecture. When most of the available DLP parallelism is exploited via multimedia extensions, the remaining scalar part becomes the bottleneck. In the 4-issue Vector-μSIMD-VLIW architecture, the vector cycles represent less than 10% of the overall execution time (except for the `mpeg2_enc`). By the Amhdal Law, further improvements in the execution of the vector regions would be imperceptible in the complete application.

6.3.4 Operations per Cycle

To conclude the analysis, Table 6.4 reports the average number of operations per cycle in the scalar and vector regions of code separately, and in the complete applications. For the μSIMD and Vector-μSIMD versions, the operations per cycle rate gives information about the fetch bandwidth requirements, but is not representative of the exploited parallelism. To take this into account, the table also include the average number of micro-operations executed per cycle.

		Scalar regions		Vector regions			Application		
		OPC	SP	OPC	μOPC	SP	OPC	μOPC	SP
2w	VLIW	1.44	1.00	1.80	1.80	1.00	1.59	1.59	1.00
	+μSIMD (2u)	1.44	1.00	1.78	4.68	2.87	1.52	2.32	1.47
	+Vector (1v4)	1.44	1.00	0.87	7.91	9.32	1.36	2.12	1.79
	+Vector (2v4)	1.44	1.00	0.98	10.10	10.60	1.37	2.15	1.80
4w	VLIW	1.77	1.23	3.03	3.03	1.66	2.14	2.14	1.34
	+μSIMD (4u)	1.78	1.24	2.95	7.80	4.62	1.98	3.05	1.94
	+Vector (2v4)	1.76	1.23	1.27	11.86	13.14	1.67	2.62	2.21
	+Vector (4v4)	1.76	1.23	1.37	14.00	14.09	1.69	2.64	2.22
8w	VLIW	1.84	1.28	4.54	4.54	2.47	2.42	2.42	1.50
	+μSIMD (8u)	1.84	1.29	4.47	12.07	6.76	2.18	3.38	2.15

Table 6.4. Average operations per cycle (*OPC*), micro-operations per cycle (μ*OPC*), and speed-up (*SP*) in the scalar and vector regions and in the full application

Results confirms our belief that the non-vector regions of code do not benefit from scaling the width of the machine above 4 issue width. Fetching 1.84 operations per cycle does not pay off the hardware complexity of a 8-issue width architecture. The Vector-μSIMD ISA obtains the highest speed-ups by exploiting more data parallelism in the vector regions (up to 14.00 micro-operations per cycle) and with the lowest fetch bandwidth requirements (just 1.37 operations per cycle), making it an ideal candidate for embedded systems, where high issue rates are not an option. However, for wide issues, the μSIMD ISA exhibits more flexibility to benefit from wide static scheduling and also reaches significant micro-operations per cycle rates, but at a higher cost.

6.4 Summary

The actual performance achieved by very wide issue VLIW architectures is very far from the theoretical peak performance and do not pay off the related hardware complexity. By analyzing the scalability of the scalar and vector regions of code separately, we have shown that the scalar regions do not benefit from increasing the width of the machine above 4-issue width. On the other hand, the kind of parallelism found in the vector regions could be exploited in a more efficient way by means of SIMD execution.

To exploit the data parallelism inherent in the vector regions without increasing the way of the architecture, we have proposed the addition of one or more vector units together with a vector register filer and a wide port to the L2 that provides the bandwidth required by the vector regions. This extension can be viewed as a conventional short vector ISA where each element is operated in a MMX-like fashion. This enhancement has a minimal impact on the VLIW core and provides high performance in the vector regions for low issue rates.

We have evaluated the proposed architecture for complete applications of audio, video and image processing and compared it against a VLIW architecture with and without μSIMD extensions. In the vector regions, a 4-issue width Vector-μSIMD-VLIW architecture outperforms the 8-issue μSIMD-VLIW architecture in a factor of up to 4.1X (1.9X in average). Due to the low vectorization percentage, the impact in the complete applications is not so significant, but a 4-issue Vector-μSIMD-VLIW achieves greater or similar performance to that of the 8-issue μSIMD-VLIW with considerably less hardware complexity and power consumption.

On the other hand, it has been seen that Vector-μSIMD-VLIW architectures do not perform well in front of non-unit stride memory references and exhibit the highest performance degradations when considering a realistic memory system, mainly due to the high sensitivity of VLIW architectures to non-deterministic latencies. Future research must be done to improve the memory hierarchy and to test more flexible scheduling techniques.

Finally, we have observed that, once the high performance requirements of the kernels have been satisfied by the use of special DLP-oriented multimedia extensions, multimedia applications become dominated by the scalar performance. To address this problem, other sources of parallelism, such as *Thread Level Parallelism* (TLP) must be exploited together with ILP and DLP to accomplish the real-time constraints and high computational throughput requirements of next generation of media workloads.

Chapter 7

Conclusions

We conclude summarizing the main contributions and some future research options.

7.1 Contributions

We started this thesis realizing the growing interest that multimedia applications have experimented in the desktop and embedded domains and the increasing computational power demands they involve. On the other hand, advances in integration technology do not involve the same performance improvement rates than some years ago, mainly due to the limited available instruction level parallelism, the memory wall and the problem of power dissipation. There is an extended concern about these constraints and whether the next generation of processors will be able to meet with success the increasing requirements of future media applications. Current trends in microprocessor design point to the exploitation of different sources of parallelism, the integration of larger caches on-chip, and a great interest in energy efficient implementations.

We think that *the combination of 2-dimensional vector processing and the VLIW paradigm together with other ways of exploiting coarser grain parallelism, such as simultaneous multithreading and chip multiprocessing, are a promising alternative to face the requirements of future multimedia workload and the emerging technology constraints.* VLIW architectures perform well for multimedia processing, while avoiding the expensive and strongly technology dependent scalability of superscalar processors. On the other hand, 2-dimensional vector extensions are an efficient way of exploiting the inherent DLP of multimedia kernels. They combine the advantages of both conventional vector and sub-word level parallelism implementations, while overcoming the scalability limitations of current μSIMD multimedia extensions. Finally, exploiting thread level parallelism is needed to deal with the processing of multiple concurrent media streams.

Our work has concentrated on improving the exploitation of instruction and data level parallelism in the context of VLIW architectures and multimedia workload.

More specifically, we have addressed two main topics: the problem of memory disambiguation and the problem of exploiting DLP by means of Vector-μSIMD extensions in static scheduling architectures. In order to evaluate the architectural improvements and compilation techniques proposed in this thesis, we have enhanced the Trimaran compilation and simulation framework. The resulting tool set provides new functionalities, such as obtaining a great range of statistics of the loops or regions in the scope of the programs, simulation of the memory hierarchy, loop memory disambiguation, and scheduling and simulation of μSIMD and Vector-μSIMD code. Next, we summarize the main contributions that this work has originated.

Characterization of multimedia applications in VLIW architectures

Understanding the behavior of multimedia applications is essential for our research. Thus, we started our work performing a quantitative analysis of the execution of a set of image, video and audio applications on VLIW architectures. Results have corroborated that the streaming data access patterns promote spatial locality, which leads to very high cache hit rates, even for small cache sizes. We have also observed that bank conflicts are an important source of performance degradation in VLIW architectures. Hence, we concluded that widening the ports is preferable to increasing the number of them; multi-porting a cache is more expensive than widen the ports and alternative feasible multi-banking cache designs produce the non-desired bank conflicts. Packing several unit-stride array references into one wide access would reduce both the number of memory access and the potential for bank conflicts.

Results also confirmed that multimedia codes exhibit more parallelism than integer ones. Nevertheless, this parallelism is not so high as it was to be expected from the definition of the algorithms. One of the reasons that explain this fact is that these applications use to include a lot of overhead to deal with different options and formats. On the other hand, in the course of time, some of the algorithms have gone through a set of optimizations mainly oriented towards reducing the number of instructions in scalar implementations, going so far as to hide the inherent vector nature of the algorithm. Furthermore, we noticed that in most cases the compiler was unable to take benefit from aggressive ILP optimizations, such as modulo scheduling, mainly due to the existence of ambiguous memory references.

Run-time memory disambiguation for multimedia loops

The last observation motivated us to analyze the problem of memory disambiguation in the context of multimedia applications. We realized that one of the main obstacles to memory disambiguation in multimedia codes is that they are often written in languages that support pointer referencing, such as C or C++. The inability of the compiler to demonstrate at compile-time that two pointers are not going to reference the same memory location in any iteration of the loop, forces it to generate conservative code in which different iterations of the loop cannot be overlapped.

Taking into account the disjointed nature of most input and output multimedia memory streams, we have proposed a memory disambiguation technique that dynamically analyzes the region domain of every load and store to evaluate, before entering the loop, whether or not the full loop is disambiguated and execute the corresponding loop version. This technique has been completely implemented into the Trimaran compiler. In contrast with other dynamic approaches, it does not require any additional hardware or instructions. It has negligible effects over compilation time and code size, and near-zero cost for all those loops without potential for disambiguation.

We have also compared our proposal against advanced interprocedural pointer analysis. Results show that, on average, our technique outperforms the later at the loop level (2.60X in front of 2.17X with relative to the non-disambiguated codes), although the average performance achieved is similar at the scope of the complete applications (1.13X on average). Furthermore, it is worth to remark that most of the benchmarks exhibit a beneficial effect when both techniques are used together. This is due to the fact that, while pointer analysis overcomes some limitations of our technique, such as the access to non-streaming data structures, run-time memory disambiguation addresses the cases in which dynamic information is required to determine the independence of two memory references. For the 8-issue width VLIW reference architecture, the combination of the two mechanisms increases the speed-up up to an average of 2.82X in the loops and 1.19X in the complete applications.

Study of scalability of the scalar and vector regions in μSIMD-VLIW architectures

The general characteristics of multimedia kernels, which are basically small loop-bodies that process streams of small data types, have lead to the extended trend of exploiting DLP by means of sub-word level (or μSIMD) multimedia extensions. However, the efficiency of sub-word level implementations is affected by the existence of unaligned and non-unit stride memory accesses and the overhead needed to arrange the elements in the appropriate way.

Another contribution of this thesis is the identification of the scalar and vector regions of each program. The vector regions are those parts of the code that can be vectorized, and the scalar regions are the remaining non-vectorizable parts of code. In order to evaluate the efficiency of aggressive configurations with multimedia extensions, we have separately analyzed the scalability of the scalar and vector regions of our set of benchmarks in μSIMD-VLIW processors. Results confirm our assumption that the scalar regions do not have enough ILP to take benefit from increasing the width of the architecture above 4-issue width. On the other hand, although the vector regions exhibit potential to scale, the vectorization percentage is not high enough, and the actual performance achieved in the complete applications does not compensate the increase in cost of wider issue architectures.

2-dimensional vector extensions in static scheduling architectures

To exploit the DLP in the vector regions without increasing the way of the architecture, we have proposed what stands for the main goal of this thesis: the Vector-μSIMD-VLIW architecture. This architecture is based on the addition of one or more vector units together with a vector register filer and a wide port to the L2 that provides the bandwidth required by the vector regions. This enhancement has a minimal impact on the VLIW core and reaches more parallelism than wider issue μSIMD at a lower cost.

Vector processing has several inherent advantages, such as the reduction in the number of executed operations, a lower pressure in the instruction fetch unit, the simplicity of the control unit, the advance knowledge of the memory accesses, the ability to amortize functional units and memory start-up latencies, and the easiness to be scaled by just replicating the functional units. The union of conventional vector processing with sub-word level vector processing can be seen as a 2-dimensional matrix extension that combines the best of each one.

Given that similar proposals have been successfully evaluated for superscalar cores, the main potential handicaps we could think of are in the compilation side. In our proposal, the assignment of operations to each functional unit, the scheduling, and the register allocation have to be performed at compile-time. Dynamic values, such as the vector length and the vector stride, are potential issues for static scheduling. Nevertheless, these values can be obtained most of the times at compile-time by means of data-flow analysis. In the few cases in which they cannot, the compiler assume default values. The penalty to pay if the assumption fails is acceptable, as we are working with short vector lengths. Nevertheless, the study under a realistic cache hierarchy has evidenced some bottlenecks related to strided memory accesses, mainly due to the high sensitivity of VLIW architectures to non-deterministic latencies.

We have reported performance gains in the vector regions of up to 4.1X (1.9X on average) for a 4-issue width architecture with two vector units of four lanes each with relative to a 8-issue width with eight μSIMD units. Both configurations performs similarly at the the scope of complete applications (the average gain is reduced to 1.02X). Nevertheless, this is specially meaningful taking into account that the Vector-μSIMD configuration has half the fetch bandwidth, the same computational power, and a register file that, even though being four times larger than the centralized μSIMD one, it allows for 70% less access time and 30% less power and area cost.

Overall, the original performance of the non-disambiguated codes running in the reference 8-issue width VLIW architecture has been improved in a factor of up to 2.72X (1.64X on average) by using a 4-issue width Vector-μSIMD processor with two vector units, and even up to 2.26X (1.33X on average) with a 2-issue width Vector-μSIMD processor with only one vector unit.

7.2 Future Work

This research opens several fields for further analysis. Next, we enumerate some future work to be done regarding both the compiler and the architecture.

Vector-μSIMD autovectorization

Compiler support is a key issue to exploit the full potential of the proposed architecture. In this thesis, we have faced the problem of scheduling Vector-μSIMD operations, but we have used emulation libraries to handwrite Vector-μSIMD code. We think that any compiler able to generate code for a μSIMD ISA could be enhanced to vectorize in a second dimension and generate code for a Vector-μSIMD ISA.

The proposed memory disambiguation test could be used to aid in those cases in which ambiguous memory dependences prevent the compiler from generating vector code. The compiler generates both, the scalar and the vector versions of code, and the proposed test evaluates at run-time which version must be executed.

Memory hierarchy

Memory performance is critical for overall performance. The main bottlenecks of the memory hierarchy must be identified in order to suggest possible improvements. In this thesis, we have observed a significant performance degradation in front of strided memory access. Work to be done include the search of both, more flexible scheduling algorithms on the compilation side and alternative designs to the vector cache on the hardware side.

Low-end Vector-μSIMD-VLIW processors

The achieved results suggest that the proposed architecture exhibit a high potential for the embedded domain, as it provides high performance at lower cost and without compromising the cycle time. It would be interesting to evaluate the potential of Vector-μSIMD-VLIW embedded processors. Given the growing interest on cost-effective designs, special attention must be paid to energy and area efficiency.

Vector-μSIMD-VLIW Chip-Multiprocessors

This work has also demonstrated that, once the high performance requirements of the vector regions have been addressed, the low performance of the scalar regions dominate program cycles, resulting into low resource usage. Given the high amount of TLP that seems to characterize current and future multimedia applications, we think that TLP must be exploited together with ILP and DLP to accomplish the real-time constraints and high computational throughput requirements of next generation of media workloads.

Currently, there is a growing trend towards exploiting TLP by means of Chip-Multiprocessors (CMPs). CMPs have the potential to provide high scalability thanks to better cache coherence mechanisms. There exists some commercial systems that combines the VLIW and the Chip-Multiprocessor (CMP) paradigms to provide high performance for multimedia at low cost. We think that Vector-μSIMD-VLIW CMPs are a good match to efficiently exploit the heterogeneous parallelism of multimedia workload.

Alternative application domains

Finally, although this work has been motivated by our interest in improving the performance of multimedia applications, the proposals behind this thesis are not restricted to this area. On the contrary, the ideas presented in this thesis can be extended to other DLP applications. We are currently analyzing the bioinformatic domain. It would be interesting to evaluate the potential of VLIW CMPs with vector extensions to face the computational intensive algorithms of this kind of applications.

Appendix A

Loop Statistics

This appendix provides detailed information about the loops of the eight applications used in this thesis. For each application, we present first a table with the general information of all innermost loops in the benchmark, and second, a more detailed description of the most representative loops.Reported data were obtained compiling the benchmarks with the original compiler for the 8-issue width reference architecture, and simulating them with the reference inputs.

The table of innermost loops is sorted by their contribution to the overall execution time of the application in descending order, and includes the following information:

- *Loop name*: The name of the loop is composed by the first twenty characters of the function it belongs to and the identifier of the header basic-block of the loop. It has been truncated to ten characters for limitation of the table width.
- *Dyn Cyc (%acc)*: Dynamic cycle count. The percentage in brackets indicates the accumulated percentage of the complete application excution time.
- *Dyn Ops (%acc)*: Dynamic operation count. The percentage in brackets indicates the accumulated percentage of the complete application operation count.
- *OPC*: Operations per cycle rate.
- *Inv*: Invocations. Number of times the loop is executed.
- *Iter*: Average number of iterations per invocation.
- *Nest*: Nesting level. The lowest level corresponds to the outer nested loop.
- *Cat*: Category. The loops have been classified into the following categories:
  - While_Loop(W): not counted loops
  - Do_Loop(D): do-loops which are not modulo scheduling
  - Mod_Sched(M): do-loops which are modulo scheduling.

- *Ops*: Static operation count.
- *LDs*: Number of static load operations.
- *STs*: Number of static store operations.

More detailed information is given for those loops which represent more than 1% of the overall execution time. It includes:

- *General information*: source file name, function name, header block, loop blocks, nesting level, category, invocations, iterations per invocations, dynamic operation count and percentage of the complete application, dynamic cycle count and percentage of the complete application, operation per cycle rate, and stall cycles due to memory and percentage of the dynamic cycle count. In the name of the blocks, *BB* stands for basic-block and *HB* for hyper-block.
- *Scheduling*: In the case of modulo scheduling loops, it shows:
  - *RecMII*: minimum initiation interval due to recurrences.
  - *ResMII*: minimum initiation interval due to resource limitation.
  - *II*: resulting initiation interval.
  - *ESC*: epilogue stage counter.

  In the other loops, it shows for each block:
  - *wsl*: weighted scheduling length.
  - *pesl*: scheduling length of the most likely exit.
  - *per*: probability of the most likely exit.
  - *wgt*: weight (number of times the block is executed).

  In both cases, the overall scheduling length of the loop is reported.
- *Operations breakdown*: dynamic and static operation counts classified into eight categories: loads, stores, integer arithmetic and logic, floating point arithmetic and logic, integer compares, floating point compares, prepare-to-branch, and branches. The number in brackets indicates the percentage of each type.
- *Memory operations*: A list of all memory operations with the following information:
  - Name of the operation: composed by the prefix *L* for loads and *S* for stores plus an identifier.
  - *Size*: data size in bytes.
  - *Stride*: distance between elements of consecutive iterations.

  Moreover, for each reference group, that is, uniformly generated references to the same array (see chapter 5), we show:

- *nOps*: number of memory operations in the group.
- *gSize*: data width of the group in bytes.
- *gStr*: stride between consecutive elements of the group.

Spill counts are also included for those loops in which the compiler has generated spill code.

A.1 Jpeg_enc

Innermost loops list

#	Loop name	Dyn Cyc	(%acc)	Dyn Ops	(%acc)	OPC	Inv	Iter	Nest	Cat	Ops	LDs	STs
1	_forward_D.9	26,790,400	(23%)	40,919,950	(20%)	1.53	17,920	64	L2	M	39	2	1
2	_rgb_ycc_c.5	16,549,691	(37%)	43,891,427	(41%)	2.65	739	1,024	L1	M	58	12	3
3	_forward_D.6	3,980,553	(40%)	5,949,440	(44%)	1.49	91,450	8	L2	M	9	1	1
4	_jpeg_fdct.3	3,208,256	(43%)	13,780,480	(51%)	4.30	17,920	8	L1	M	96	8	8
5	_h2v2_down.4	3,038,515	(46%)	7,199,460	(55%)	2.37	740	512	L1	M	19	4	1
6	_jpeg_fdct.5	2,867,200	(48%)	14,067,200	(61%)	4.91	17,920	8	L1	M	98	8	8
7	_encode_on.23	413,292	(48%)	444,975	(62%)	1.08	57,312	1	L2	W	5	0	0
8	_encode_on.5	187,590	(48%)	214,870	(62%)	1.15	15,694	2	L1	W	5	0	0
9	_jpeg_add_.7	4,436	(48%)	4,116	(62%)	0.93	4	64	L1	M	19	2	1
10	_jpeg_make.10	3,723	(48%)	5,220	(62%)	1.40	69	7	L2	W	10	1	1
11	_jpeg_make.13	3,720	(48%)	8,880	(62%)	2.39	6	87	L1	M	17	4	2
12	_jpeg_make.6	2,679	(48%)	4,176	(62%)	1.56	69	7	L2	W	8	1	1
13	_rgb_ycc_s.3	2,313	(48%)	8,705	(62%)	3.76	1	256	L1	M	34	0	8
14	_start_pas.19	975	(48%)	1,731	(62%)	1.78	3	64	L2	M	9	1	1
15	_compress_.53	576	(48%)	896	(62%)	1.56	32	2	L4	W	14	4	1
16	_jpeg_make.37	361	(48%)	235	(62%)	0.65	69	1	L2	W	7	0	0
17	_alloc_sma.14	264	(48%)	227	(62%)	0.86	31	1	L1	W	12	2	0
18	_emit_dqt_.5	201	(48%)	1,170	(62%)	5.82	3	64	L1	M	7	1	0
19	_alloc_sar.12	98	(48%)	225	(62%)	2.30	6	6	L2	M	5	0	1
20	_emit_dht_.9	72	(48%)	328	(62%)	4.56	4	16	L1	M	5	1	0
21	_jpeg_set_.7	51	(48%)	129	(62%)	2.53	1	16	L1	M	8	0	3
22	_per_scan_.22	48	(48%)	75	(62%)	1.56	3	2	L1	W	14	1	2
23	_jpeg_supp.7	40	(48%)	47	(62%)	1.18	1	4	L1	M	13	2	2
24	_select_sc.9	22	(48%)	36	(62%)	1.64	1	3	L1	W	12	2	1
25	_jinit_huf.3	20	(48%)	41	(62%)	2.05	1	4	L1	M	10	0	4
26	_jpeg_supp.3	20	(48%)	30	(62%)	1.50	1	4	L1	M	8	1	1
27	_jinit_c_c.10	11	(48%)	51	(62%)	4.64	1	10	L1	M	5	0	1
28	_write_fra.19	10	(48%)	33	(62%)	3.30	1	3	L1	M	11	2	0
29	_jinit_for.8	5	(48%)	25	(62%)	5.00	1	4	L1	M	6	0	2
30	_jpeg_Crea.9	5	(48%)	25	(62%)	5.00	1	4	L1	M	6	0	2
31	_jpeg_Crea.7	5	(48%)	17	(62%)	3.40	1	4	L1	M	4	0	1
32	_jinit_mem.9	3	(48%)	13	(62%)	4.33	1	2	L1	M	6	0	2

Table A.1. Jpeg_enc innermost loops list

Description of the most representative loops

LOOP_0 _forward_DCT_jcdctmgr.9

Program:	jpeg_enc
File:	jcdctmgr.c
Function:	forward_DCT_jcdctmgr
Header block:	HB_9
Loop blocks:	HB_9
Nesting level:	2
Innermost:	yes
Category:	MOD_SCHED
Invocations:	17920
Iterations:	1146880
Iter/Invoc:	64
Operations:	40919950 (19.97%)
Cycles:	26790400 (22.77%)
Ops/Cyc:	1.53
Stall cycles:	0 (0.00%)
Scheduling	

RecMII	ResMII	II	ESC

	RecMII	ResMII	II	ESC
HB_9	23	5	23	1

Sched length: 1495

Operation breakdown

	Dynamic counts		Static counts	
Load:	2293760	(6%)	2	(5%)
Store:	1146880	(3%)	1	(3%)
iAlu:	32820110	(80%)	32	(82%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	3494400	(9%)	3	(8%)
Pbr:	0	(0%)	0	(0%)
Branch:	1164800	(3%)	1	(3%)
Total:	40919950		39	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_72	2	1				
L_76	2	1				
S_135	2	1				

LOOP_1 _rgb_ycc_convert_jcco.5

Program: jpeg_enc
File: jccolor.c
Function: rgb_ycc_convert_jcco
Header block: BB_5
Loop blocks: BB_5
Nesting level: 1
Innermost: yes
Category: MOD_SCHED
Invocations: 739
Iterations: 756736
Iter/Invoc: 1024
Operations: 43891427 (21.42%)
Cycles: 16549691 (14.07%)
Ops/Cyc: 2.65
Stall cycles: 1400191 (8.46%)

Scheduling

	RecMII	ResMII	II	ESC
BB_5	19	8	20	1

Sched length: 20500

Operation breakdown

	Dynamic counts		Static counts	
Load:	9080832	(21%)	12	(21%)
Store:	2270208	(5%)	3	(5%)
iAlu:	31782912	(72%)	42	(72%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	757475	(2%)	1	(2%)
Total:	43891427		58	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_47	1	3	G_47	3	3	1
L_49	1	3	"			
L_51	1	3	"			
L_56	4	-				
L_59	4	-				
L_63	4	-				
L_70	4	-				
L_73	4	-				
L_77	4	-				
L_84	4	-				
L_87	4	-				
L_91	4	-				
S_67	1	1				
S_81	1	1				

S_95	1	1

LOOP_2 _forward_DCT_jcdctmgr.6

Program:	jpeg_enc
File:	jcdctmgr.c
Function:	forward_DCT_jcdctmgr
Header block:	BB_6
Loop blocks:	BB_6
Nesting level:	2
Innermost:	yes
Category:	MOD_SCHED
Invocations:	91450
Iterations:	731605
Iter/Invoc:	8
Operations:	5949440 (2.90%)
Cycles:	3980553 (3.38%)
Ops/Cyc:	1.49
Stall cycles:	38153 (0.96%)

Scheduling

	RecMII	ResMII	II	ESC
BB_6	5	2	5	1
Sched length:	45			

Operation breakdown

	Dynamic counts		Static counts	
Load:	645120	(11%)	1	(11%)
Store:	645120	(11%)	1	(11%)
iAlu:	3870720	(65%)	6	(67%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	788480	(13%)	1	(11%)
Total:	5949440		9	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_46	1	1				
S_54	2	1				

LOOP_3 _jpeg_fdct_islow.3

Program:	jpeg_enc
File:	jfdctint.c
Function:	jpeg_fdct_islow
Header block:	BB_3
Loop blocks:	BB_3
Nesting level:	1
Innermost:	yes
Category:	MOD_SCHED
Invocations:	17920
Iterations:	143360
Iter/Invoc:	8
Operations:	13780480 (6.73%)
Cycles:	3208256 (2.73%)
Ops/Cyc:	4.30
Stall cycles:	627776 (19.57%)

Scheduling

	RecMII	ResMII	II	ESC
BB_3	14	12	16	1
Sched length:	144			

Operation breakdown

	Dynamic counts		Static counts	
Load:	1146880	(8%)	8	(8%)
Store:	1146880	(8%)	8	(8%)
iAlu:	11325440	(82%)	79	(82%)

fAlu:	0	(0%)	0	(0%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	161280	(1%)	1	(1%)
Total:	13780480		96	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_11	2	8	G_11	8	16	1
L_12	2	8	"			
L_19	2	8	"			
L_20	2	8	"			
L_27	2	8	"			
L_28	2	8	"			
L_35	2	8	"			
L_36	2	8	"			
S_53	2	8	G_53	8	16	1
S_56	2	8	"			
S_64	2	8	"			
S_69	2	8	"			
S_105	2	8	"			
S_110	2	8	"			
S_115	2	8	"			
S_120	2	8	"			

LOOP_4 _h2v2_downsample_jcsa.4

Program:	jpeg_enc
File:	jcsample.c
Function:	h2v2_downsample_jcsa
Header block:	BB_4
Loop blocks:	BB_4
Nesting level:	1
Innermost:	yes
Category:	MOD_SCHED
Invocations:	740
Iterations:	378880
Iter/Invoc:	512
Operations:	7199460 (3.51%)
Cycles:	3038515 (2.58%)
Ops/Cyc:	2.37
Stall cycles:	1555 (0.05%)

Scheduling

	RecMII	ResMII	II	ESC
BB_4	8	3	8	1
Sched length:	4104			

Operation breakdown

	Dynamic counts		Static counts	
Load:	1515520	(21%)	4	(21%)
Store:	378880	(5%)	1	(5%)
iAlu:	4925440	(68%)	13	(68%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	379620	(5%)	1	(5%)
Total:	7199460		19	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_40	1	2				
L_41	1	2				
L_43	1	2	G_43	2	2	1
L_45	1	2	"			
S_54	1	1				

LOOP_5 _jpeg_fdct_islow.5

Program:	jpeg_enc
File:	jfdctint.c
Function:	jpeg_fdct_islow
Header block:	BB_5
Loop blocks:	BB_5
Nesting level:	1
Innermost:	yes
Category:	MOD_SCHED
Invocations:	17920
Iterations:	143360
Iter/Invoc:	8
Operations:	14067200 (6.87%)
Cycles:	2867200 (2.44%)
Ops/Cyc:	4.91
Stall cycles:	286720 (10.00%)

Scheduling

	RecMII	ResMII	II	ESC
BB_5	14	13	16	1
Sched length:	144			

Operation breakdown

	Dynamic counts		Static counts	
Load:	1146880	(8%)	8	(8%)
Store:	1146880	(8%)	8	(8%)
iAlu:	11612160	(83%)	81	(83%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	161280	(1%)	1	(1%)
Total:	14067200		98	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_129	2	1	G_129	8	114	8
L_130	2	1	"			
L_137	2	1	"			
L_138	2	1	"			
L_145	2	1	"			
L_146	2	1	"			
L_153	2	1	"			
L_154	2	1	"			
S_172	2	1	G_172	8	114	8
S_176	2	1	"			
S_184	2	1	"			
S_189	2	1	"			
S_225	2	1	"			
S_230	2	1	"			
S_235	2	1	"			
S_240	2	1	"			

A.2 Jpeg_dec

Innermost loops list

#	Loop name	Dyn Cyc	(%acc)	Dyn Ops	(%acc)	OPC	Inv	Iter	Nest	Cat	Ops	LDs	STs
1	_jpeg_idct.8	36,858,048	(40%)	68,989,748	(40%)	1.87	17,920	8	L1	D	532	114	107
2	_ycc_rgb_c.5	15,618,672	(58%)	26,486,499	(56%)	1.70	739	1,024	L1	M	35	10	3
3	_jpeg_idct.3	15,433,064	(74%)	31,127,431	(74%)	2.02	17,920	8	L1	D	510	110	97
4	_h2v2_fanc.8	7,660,055	(83%)	18,871,480	(85%)	2.46	1,480	510	L3	M	25	2	2
5	_decompres.13	132,352	(83%)	265,088	(85%)	2.00	12,016	1	L4	W	30	7	0
6	_jpeg_make.21	4,809	(83%)	12,215	(85%)	2.54	87	17	L2	M	8	1	2
7	_jpeg_make.10	3,723	(83%)	5,220	(85%)	1.40	69	7	L2	W	10	1	1
8	_build_ycc.4	3,084	(83%)	6,401	(85%)	2.08	1	256	L1	M	25	4	4
9	_jpeg_make.6	2,679	(83%)	4,176	(85%)	1.56	69	7	L2	W	8	1	1
10	_start_pas.25	1,560	(83%)	1,347	(85%)	0.86	3	64	L2	M	7	1	1
11	_jpeg_make.13	1,326	(83%)	1,893	(85%)	1.43	6	16	L1	M	22	4	4
12	_make_funn.9	1,082	(83%)	23	(85%)	0.02	3	1	L2	M	5	1	1
13	_prepare_r.5	385	(83%)	1,537	(85%)	3.99	1	384	L1	M	4	0	1
14	_jpeg_make.57	361	(83%)	235	(85%)	0.65	69	1	L2	W	7	0	0
15	_jpeg_make.42	312	(83%)	720	(85%)	2.31	6	8	L2	D	20	2	0
16	_alloc_sma.14	264	(83%)	227	(85%)	0.86	43	1	L1	W	12	2	0
17	_prepare_r.3	257	(83%)	1,025	(85%)	3.99	1	256	L1	M	4	0	1
18	_make_funn.5	176	(83%)	323	(85%)	1.84	3	13	L3	M	8	1	2
19	_set_wrapa.18	108	(83%)	83	(85%)	0.77	3	1	L2	M	20	4	4
20	_alloc_sar.12	98	(83%)	225	(85%)	2.30	5	8	L2	M	5	0	1
21	_set_wrapa.4	84	(83%)	156	(85%)	1.86	3	1	L2	W	57	14	4
22	_set_botto.10	74	(83%)	105	(85%)	1.42	3	1	L1	W	43	9	2
23	_jinit_mar.3	68	(83%)	129	(85%)	1.90	1	16	L1	M	8	1	1
24	_per_scan_.22	48	(83%)	75	(85%)	1.56	3	2	L1	W	14	1	2
25	_make_funn.7	44	(83%)	99	(85%)	2.25	3	2	L2	M	12	2	2
26	_get_sos_j.46	36	(83%)	36	(85%)	1.00	3	2	L2	D	8	1	0
27	_set_botto.9	34	(83%)	51	(85%)	1.50	3	2	L1	M	6	1	1
28	_get_soi_j.5	17	(83%)	129	(85%)	7.59	1	16	L1	M	8	0	3
29	_jinit_d_c.12	11	(83%)	51	(85%)	4.64	1	10	L1	M	5	0	1
30	_jinit_huf.3	10	(83%)	25	(85%)	2.50	1	4	L1	M	6	0	2
31	_jpeg_Crea.9	5	(83%)	25	(85%)	5.00	1	4	L1	M	6	0	2
32	_jpeg_Crea.7	5	(83%)	17	(85%)	3.40	1	4	L1	M	4	0	1
33	_jinit_mem.9	3	(83%)	13	(85%)	4.33	1	2	L1	M	6	0	2

Table A.2. Jpeg_dec innermost loops list

Description of the most representative loops

LOOP_0 _jpeg_idct_islow.8

Program:	jpeg_dec
File:	jidctint.c
Function:	jpeg_idct_islow
Header block:	HB_8
Loop blocks:	HB_8 HB_17
Nesting level:	1
Innermost:	yes
Category:	DO_LOOP
Invocations:	17920
Iterations:	143360
Iter/Invoc:	8
Operations:	68989748 (40.25%)
Cycles:	36858048 (40.41%)
Ops/Cyc:	1.87
Stall cycles:	157874 (0.43%)

Scheduling

	wsl	pesl	per	wgt
HB_8	254.87	270	0.78	143360
HB_17	10.12	10	0.88	16071
Sched length:	2048			

Operation breakdown

	Dynamic counts		Static counts	
Load:	15092060	(22%)	114	(21%)
Store:	13549800	(20%)	107	(20%)
iAlu:	39615017	(57%)	304	(57%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	143360	(0%)	1	(0%)
Pbr:	302791	(0%)	3	(1%)
Branch:	286720	(0%)	3	(1%)
Total:	68989748		532	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_202	4	1				
L_225	1	-				
L_206	4	8	G_259	8	32	1
L_207	4	8	"			
L_209	4	8	"			
L_211	4	8	"			
L_213	4	8	"			
L_215	4	8	"			
L_217	4	8	"			
L_220	4	8	"			
L_259	4	8	"			
L_333	1	-				
L_340	1	-				
L_347	1	-				
L_354	1	-				
L_361	1	-				
L_368	1	-				
L_375	1	-				
L_382	1	-				
S_228	1	-	G_334	8	8	1
S_230	1	-	"			
S_232	1	-	"			
S_234	1	-	"			
S_236	1	-	"			
S_238	1	-	"			
S_240	1	-	"			
S_242	1	-	"			
S_334	1	-	"			
S_341	1	-	"			
S_348	1	-	"			
S_355	1	-	"			
S_362	1	-	"			
S_369	1	-	"			
S_376	1	-	"			
S_383	1	-	"			

	Dynamic count		Static count	
Spill:	25162322	(36%)	186	(35%)

LOOP_1 _ycc_rgb_convert_jdco.5

Program:	jpeg_dec
File:	jdcolor.c
Function:	ycc_rgb_convert_jdco
Header block:	BB_5
Loop blocks:	BB_5
Nesting level:	1
Innermost:	yes
Category:	MOD_SCHED
Invocations:	739

Iterations: 756736
Iter/Invoc: 1024
Operations: 26486499 (15.45%)
Cycles: 15618672 (17.12%)
Ops/Cyc: 1.70
Stall cycles: 469172 (3.00%)

Scheduling

	RecMII	ResMII	II	ESC
BB_5	20	5	20	1
Sched length:	20500			

Operation breakdown

	Dynamic counts		Static counts	
Load:	7567360	(29%)	10	(29%)
Store:	2270208	(9%)	3	(9%)
iAlu:	15891456	(60%)	21	(60%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	757475	(3%)	1	(3%)
Total:	26486499		35	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_56	1	1				
L_59	1	1				
L_62	1	1				
L_65	4	-				
L_68	1	-				
L_71	4	-				
L_73	4	-				
L_78	1	-				
L_81	4	-				
L_84	1	-				
S_69	1	3	G_69	3	3	1
S_79	1	3	"			
S_85	1	3	"			

LOOP_2 _jpeg_idct_islow.3

Program: jpeg_dec
File: jidctint.c
Function: jpeg_idct_islow
Header block: HB_3
Loop blocks: HB_3 HB_15
Nesting level: 1
Innermost: yes
Category: DO_LOOP
Invocations: 17920
Iterations: 143360
Iter/Invoc: 8
Operations: 31127431 (18.16%)
Cycles: 15433064 (16.92%)
Ops/Cyc: 2.02
Stall cycles: 756401 (4.90%)

Scheduling

	wsl	pesl	per	wgt
HB_3	22.10	23	0.59	143360
HB_15	244.12	244	0.88	47143
Sched length:	819			

Operation breakdown

	Dynamic counts		Static counts	
Load:	6628985	(21%)	110	(22%)
Store:	5542765	(18%)	97	(19%)
iAlu:	18191738	(58%)	296	(58%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	143360	(0%)	1	(0%)

Pbr:	333863	(1%)	3	(1%)
Branch:	286720	(1%)	3	(1%)
Total:	31127431		510	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_24	2	1	G_38	8	114	8
L_25	2	1	"			
L_27	2	1	"			
L_29	2	1	"			
L_31	2	1	"			
L_33	2	1	"			
L_35	2	1	"			
L_38	2	1	"			
L_75	2	1	"			
L_39	4	1	G_39	8	228	8
L_59	4	1	"			
L_63	4	1	"			
L_76	4	1	"			
L_80	4	1	"			
L_98	4	1	"			
L_102	4	1	"			
L_106	4	1	"			
L_110	4	1	"			
S_43	4	1	G_43	8	228	8
S_44	4	1	"			
S_45	4	1	"			
S_46	4	1	"			
S_47	4	1	"			
S_48	4	1	"			
S_49	4	1	"			
S_50	4	1	"			
S_159	4	1	"			
S_163	4	1	"			
S_167	4	1	"			
S_171	4	1	"			
S_175	4	1	"			
S_179	4	1	"			
S_183	4	1	"			
S_187	4	1	"			

	Dynamic count		Static count	
Spill:	9310343	(30%)	173	(34%)

LOOP_3 _h2v2_fancy_upsample_.8

Program:	jpeg_dec
File:	jdsample.c
Function:	h2v2_fancy_upsample_
Header block:	BB_8
Loop blocks:	BB_8
Nesting level:	3
Innermost:	yes
Category:	MOD_SCHED
Invocations:	1480
Iterations:	754800
Iter/Invoc:	510
Operations:	18871480 (11.01%)
Cycles:	7660055 (8.40%)
Ops/Cyc:	2.46
Stall cycles:	853535 (11.14%)

Scheduling

	RecMII	ResMII	II	ESC
BB_8	9	4	9	1
Sched length:	4599			

Operation breakdown

	Dynamic counts	Static counts

Load:	1509600	(8%)	2	(8%)
Store:	1509600	(8%)	2	(8%)
iAlu:	15096000	(80%)	20	(80%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	756280	(4%)	1	(4%)
Total:	18871480		25	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_95	1	1				
L_97	1	1				
S_109	1	2	G_109	2	2	1
S_119	1	2	"			

A.3 Mpeg2_enc

Innermost loops list

#	Loop name	Dyn Cyc	(%acc)	Dyn Ops	(%acc)	OPC	Inv	Iter	Nest	Cat	Ops	LDs	STs
1	_dist1_mot.9	237,589,058	(41%)	968,846,938	(58%)	4.08	6,232,293	16	L2	M	10	2	0
2	_dist1_mot.45	22,955,002	(45%)	103,208,256	(64%)	4.50	318,544	16	L2	M	21	5	0
3	_fdct.5	14,601,432	(48%)	27,844,608	(66%)	1.91	344,899	8	L2	M	11	2	0
4	_fdct.11	14,159,915	(50%)	26,695,680	(67%)	1.89	384,516	8	L2	M	9	2	0
5	_idct.8	12,000,482	(53%)	21,116,739	(68%)	1.76	8,448	4	L1	D	567	133	105
6	_dist1_mot.24	9,118,042	(54%)	37,927,920	(71%)	4.16	164,904	16	L2	M	15	3	0
7	_iquant_no.5	7,236,353	(55%)	7,828,077	(71%)	1.08	4,494	64	L1	M	31	2	1
8	_dist1_mot.37	6,807,065	(57%)	37,524,960	(73%)	5.51	163,152	16	L2	M	15	3	0
9	_quant_non.3	6,402,057	(58%)	7,773,563	(74%)	1.21	4,494	64	L1	M	31	3	1
10	_quant_int.6	5,807,013	(59%)	6,359,104	(74%)	1.10	3,954	63	L1	M	29	3	1
11	_iquant_in.5	5,269,067	(60%)	5,758,632	(75%)	1.09	3,954	63	L1	M	25	2	1
12	_idct.3	4,506,548	(60%)	8,464,749	(75%)	1.88	8,448	3	L1	D	332	66	63
13	_bdist1_mo.4	3,376,753	(61%)	13,111,296	(76%)	3.88	22,528	16	L2	M	37	9	0
14	_add_pred_.4	2,539,391	(61%)	4,325,376	(76%)	1.70	43,112	8	L2	M	14	4	1
15	_calcSNR1_.4	2,509,345	(62%)	6,088,320	(76%)	2.43	1,920	264	L3	M	12	2	0
16	_dct_type_12	2,110,992	(62%)	3,435,520	(77%)	1.63	11,264	16	L4	M	19	5	2
17	_sub_pred_.4	1,311,950	(62%)	2,804,736	(77%)	2.14	43,112	8	L2	M	9	2	1
18	_variance_.4	1,210,878	(63%)	2,635,776	(77%)	2.18	22,528	16	L2	M	7	1	0
19	_pred_comp.49	1,098,653	(63%)	1,359,824	(77%)	1.24	11,376	11	L2	M	10	2	1
20	_bdist2_mo.4	867,816	(63%)	3,316,512	(77%)	3.82	5,776	16	L2	M	36	9	0
21	_var_sblk_.4	856,510	(63%)	3,289,088	(77%)	3.84	57,483	8	L2	M	7	1	0
22	_dct_type_.15	729,545	(63%)	3,066,624	(78%)	4.20	1,408	128	L3	M	17	2	0
23	_pred_comp.44	642,629	(63%)	789,712	(78%)	1.23	4,880	11	L1	M	14	3	1
24	_pred_comp.15	429,506	(63%)	478,704	(78%)	1.11	8,320	11	L2	M	5	1	1
25	_pred_comp.61	363,223	(64%)	633,712	(78%)	1.74	2,968	13	L1	M	16	4	1
26	_dist2_mot.34	307,986	(64%)	1,369,520	(78%)	4.45	4,240	16	L2	M	20	5	0
27	_pred_comp.10	273,000	(64%)	362,272	(78%)	1.33	3,464	11	L2	M	9	2	1
28	_pred_comp.56	260,018	(64%)	443,704	(78%)	1.71	1,536	13	L1	M	21	5	1
29	_dist2_mot.9	242,505	(64%)	789,096	(78%)	3.25	5,368	16	L2	M	9	2	0
30	_dist2_mot.19	236,476	(64%)	883,024	(78%)	3.73	3,840	16	L2	M	14	3	0
31	_clearbloc.7	181,458	(64%)	687,440	(78%)	3.79	10,544	16	L2	M	4	0	1
32	_fullsearc.69	154,566	(64%)	128,805	(78%)	0.83	25,761	1	L2	W	8	0	0
33	_pred_comp.32	129,126	(64%)	186,288	(78%)	1.44	1,304	13	L2	M	10	2	1
34	_border_ex.8	105,097	(64%)	169,088	(78%)	1.61	128	264	L2	M	5	1	1
35	_pred_comp.48	68,064	(64%)	68,064	(78%)	1.00	11,376	1	L2	W	13	0	0
36	_putDC_put.17	50,932	(64%)	59,240	(78%)	1.16	3,528	3	L1	W	5	0	0
37	_dist2_mot.29	48,886	(64%)	239,992	(78%)	4.91	1,032	16	L2	M	14	3	0
38	_clearbloc.25	47,835	(64%)	174,504	(78%)	3.65	5,272	8	L2	M	4	0	1
39	_clearbloc.18	47,592	(64%)	174,504	(78%)	3.67	5,272	8	L2	M	4	0	1
40	_pred_comp.27	38,296	(64%)	50,456	(78%)	1.32	296	12	L2	M	14	3	1
41	_clearbloc.17	31,728	(64%)	15,864	(78%)	0.50	5,272	1	L2	W	7	0	0
42	_clearbloc.24	31,728	(64%)	15,864	(78%)	0.50	5,272	1	L2	W	7	0	0
43	_pred_comp.45	31,219	(64%)	52,145	(78%)	1.67	4,880	1	L1	W	11	0	0
44	_pred_comp.62	19,299	(64%)	34,737	(78%)	1.80	2,968	1	L1	W	12	0	0
45	_stats.12	12,672	(64%)	58,353	(78%)	4.60	1,408	6	L2	M	6	0	0
46	_putseq.84	12,076	(64%)	20,448	(78%)	1.69	1,408	1	L2	W	15	1	0
47	_pred_comp.57	10,060	(64%)	19,652	(78%)	1.95	1,536	1	L1	W	13	0	0
48	_putpict.190	5,939	(64%)	9,633	(78%)	1.62	657	1	L3	W	13	1	0
49	_init_idct.3	3,075	(64%)	10,755	(78%)	3.50	1	1,024	L1	M	12	1	1
50	_init_mpeg.26	3,075	(64%)	10,115	(78%)	3.29	1	1,024	L1	M	11	1	1
51	_putpict.193	2,984	(64%)	9,698	(78%)	3.25	748	3	L3	M	4	0	1
52	_readparmf.17	1,096	(64%)	67	(78%)	0.06	1	3	L1	M	22	5	5
53	_border_ex.7	768	(64%)	384	(78%)	0.50	128	1	L2	W	7	0	0

Table A.3. Mpeg2_enc innermost loops list

#	Loop name	Dyn Cyc	(%acc)	Dyn Ops	(%acc)	OPC	Inv	Iter	Nest	Cat	Ops	LDs	STs
54	_predict.3	384	(64%)	320	(78%)	0.83	64	1	L2	W	8	0	0
55	_calc_actj.43	384	(64%)	192	(78%)	0.50	64	1	L2	W	6	0	0
56	_dct_type_.3	384	(64%)	192	(78%)	0.50	64	1	L2	W	6	0	0
57	_putpict.18	336	(64%)	980	(78%)	2.92	64	3	L3	M	4	0	1
58	_readquant.4	136	(64%)	321	(78%)	2.36	1	64	L1	M	5	1	1
59	_readquant.20	65	(64%)	257	(78%)	3.95	1	64	L1	M	4	0	1

Table A.3. Mpeg2_enc innermost loops (cont.)

Description of the most representative loops

LOOP_0 _dist1_motion_i_1920_.9

Program: mpeg2_enc
File: motion.c
Function: dist1_motion_i_1920_
Header block: HB_9
Loop blocks: HB_9
Nesting level: 2
Innermost: yes
Category: MOD_SCHED
Invocations: 6232293
Iterations: 99716688
Iter/Invoc: 16
Operations: 968846938 (57.76%)
Cycles: 237589058 (41.41%)
Ops/Cyc: 4.08
Stall cycles: 13226582 (5.57%)

Scheduling

	RecMII	ResMII	II	ESC
HB_9	1	2	2	2

Sched length: 36

Operation breakdown

	Dynamic counts		Static counts	
Load:	199433312	(21%)	2	(20%)
Store:	0	(0%)	0	(0%)
iAlu:	545051150	(56%)	6	(60%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	112181238	(12%)	1	(10%)
Pbr:	0	(0%)	0	(0%)
Branch:	112181238	(12%)	1	(10%)
Total:	968846938		10	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_36	1	1				
L_38	1	1				

LOOP_1 _dist1_motion_i_1920_.45

Program: mpeg2_enc
File: motion.c
Function: dist1_motion_i_1920_
Header block: HB_45
Loop blocks: HB_45
Nesting level: 2
Innermost: yes
Category: MOD_SCHED
Invocations: 318544
Iterations: 5096704

Iter/Invoc: 16
Operations: 103208256 (6.15%)
Cycles: 22955002 (4.00%)
Ops/Cyc: 4.50
Stall cycles: 5753626 (25.06%)

Scheduling

	RecMII	ResMII	II	ESC
HB_45	1	3	3	2
Sched length:	54			

Operation breakdown

	Dynamic counts		Static counts	
Load:	25483520	(25%)	5	(24%)
Store:	0	(0%)	0	(0%)
iAlu:	66257152	(64%)	14	(67%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	5733792	(6%)	1	(5%)
Pbr:	0	(0%)	0	(0%)
Branch:	5733792	(6%)	1	(5%)
Total:	103208256		21	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_155	1	1	G_155	2	2	1
L_158	1	1	"			
L_161	1	1	G_161	2	2	1
L_165	1	1	"			
L_170	1	1				

LOOP_2 _fdct.5

Program: mpeg2_enc
File: fdctref.c
Function: fdct
Header block: BB_5
Loop blocks: BB_5
Nesting level: 2
Innermost: yes
Category: MOD_SCHED
Invocations: 344899
Iterations: 2759196
Iter/Invoc: 8
Operations: 27844608 (1.66%)
Cycles: 14601432 (2.54%)
Ops/Cyc: 1.91
Stall cycles: 543960 (3.73%)

Scheduling

	RecMII	ResMII	II	ESC
BB_5	4	2	4	2
Sched length:	40			

Operation breakdown

	Dynamic counts		Static counts	
Load:	4866048	(17%)	2	(18%)
Store:	0	(0%)	0	(0%)
iAlu:	14598144	(52%)	6	(55%)
fAlu:	4866048	(17%)	2	(18%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	3514368	(13%)	1	(9%)
Total:	27844608		11	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_18	8	1				
L_22	2	1				

LOOP_3 _fdct.11

Program: mpeg2_enc
File: fdctref.c
Function: fdct
Header block: BB_11
Loop blocks: BB_11
Nesting level: 2
Innermost: yes
Category: MOD_SCHED
Invocations: 384516
Iterations: 3076128
Iter/Invoc: 8
Operations: 26695680 (1.59%)
Cycles: 14159915 (2.47%)
Ops/Cyc: 1.89
Stall cycles: 372779 (2.63%)

Scheduling

	RecMII	ResMII	II	ESC
BB_11	4	2	4	1

Sched length: 36

Operation breakdown

	Dynamic counts		Static counts	
Load:	5812224	(22%)	2	(22%)
Store:	0	(0%)	0	(0%)
iAlu:	14530560	(54%)	5	(56%)
fAlu:	2906112	(11%)	1	(11%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	3446784	(13%)	1	(11%)
Total:	26695680		9	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_51	8	1				
L_56	8	8				

LOOP_4 _idct.8

Program: mpeg2_enc
File: idct.c
Function: idct
Header block: HB_8
Loop blocks: HB_8
Nesting level: 1
Innermost: yes
Category: DO_LOOP
Invocations: 8448
Iterations: 39163
Iter/Invoc: 5
Operations: 21116739 (1.26%)
Cycles: 12000482 (2.09%)
Ops/Cyc: 1.76
Stall cycles: 250379 (2.09%)

Scheduling

	wsl	pesl	per	wgt
HB_8	300.10	315	0.78	39163

Sched length: 1391

Operation breakdown

	Dynamic counts		Static counts	
Load:	4911481	(23%)	133	(23%)
Store:	3906285	(18%)	105	(19%)
iAlu:	12107748	(57%)	324	(57%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	39157	(0%)	1	(0%)
Pbr:	78314	(0%)	2	(0%)
Branch:	73754	(0%)	2	(0%)
Total:	21116739		567	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_166	2	1	G_232	8	114	8
L_170	2	1	"			
L_173	2	1	"			
L_176	2	1	"			
L_179	2	1	"			
L_182	2	1	"			
L_185	2	1	"			
L_232	2	1	"			
L_302	4	0				
L_306	2	-				
L_308	4	0				
L_312	2	-				
L_314	4	0				
L_318	2	-				
L_320	4	0				
L_324	2	-				
L_326	4	0				
L_330	2	-				
L_332	4	0				
L_336	2	-				
L_338	4	0				
L_342	2	-				
L_344	4	0				
L_348	2	-				
S_307	2	1	G_307	8	114	8
S_313	2	1	"			
S_319	2	1	"			
S_325	2	1	"			
S_331	2	1	"			
S_337	2	1	"			
S_343	2	1	"			
S_349	2	1	"			

	Dynamic count		Static count	
Spill:	7671902	(36%)	206	(36%)

LOOP_5 _dist1_motion_i_1920_.24

Program:	mpeg2_enc
File:	motion.c
Function:	dist1_motion_i_1920_
Header block:	HB_24
Loop blocks:	HB_24
Nesting level:	2
Innermost:	yes
Category:	MOD_SCHED
Invocations:	164904
Iterations:	2638464
Iter/Invoc:	16
Operations:	37927920 (2.26%)
Cycles:	9118042 (1.59%)
Ops/Cyc:	4.16
Stall cycles:	2851690 (31.28%)

Scheduling

	RecMII	ResMII	II	ESC
HB_24	1	2	2	3
Sched length:	38			

Operation breakdown

	Dynamic counts		Static counts	
Load:	7915392	(21%)	3	(20%)
Store:	0	(0%)	0	(0%)
iAlu:	23746176	(63%)	10	(67%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	3133176	(8%)	1	(7%)

Pbr:	0	(0%)	0	(0%)
Branch:	3133176	(8%)	1	(7%)
Total:	37927920		15	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_73	1	1	G_73	2	2	1
L_76	1	1	"			
L_81	1	1				

LOOP_6 _iquant_non_intra.5

Program:	mpeg2_enc
File:	quantize.c
Function:	iquant_non_intra
Header block:	HB_5
Loop blocks:	HB_5
Nesting level:	1
Innermost:	yes
Category:	MOD_SCHED
Invocations:	4494
Iterations:	287616
Iter/Invoc:	64
Operations:	7828077 (0.47%)
Cycles:	7236353 (1.26%)
Ops/Cyc:	1.08
Stall cycles:	244433 (3.38%)

Scheduling

	RecMII	ResMII	II	ESC
HB_5	24	4	24	1
Sched length:	1560			

Operation breakdown

	Dynamic counts		Static counts	
Load:	573696	(7%)	2	(6%)
Store:	286848	(4%)	1	(3%)
iAlu:	5510883	(70%)	23	(74%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	1165320	(15%)	4	(13%)
Pbr:	0	(0%)	0	(0%)
Branch:	291330	(4%)	1	(3%)
Total:	7828077		31	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_24	2	1				
L_34	1	1				
S_48	2	1				

LOOP_7 _dist1_motion_i_1920_.37

Program:	mpeg2_enc
File:	motion.c
Function:	dist1_motion_i_1920_
Header block:	HB_37
Loop blocks:	HB_37
Nesting level:	2
Innermost:	yes
Category:	MOD_SCHED
Invocations:	163152
Iterations:	2610432
Iter/Invoc:	16
Operations:	37524960 (2.24%)
Cycles:	6807065 (1.19%)
Ops/Cyc:	5.51
Stall cycles:	607289 (8.92%)

Scheduling

	RecMII	ResMII	II	ESC
HB_37	1	2	2	3
Sched length:	38			

Operation breakdown

	Dynamic counts		Static counts	
Load:	7831296	(21%)	3	(20%)
Store:	0	(0%)	0	(0%)
iAlu:	23493888	(63%)	10	(67%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	3099888	(8%)	1	(7%)
Pbr:	0	(0%)	0	(0%)
Branch:	3099888	(8%)	1	(7%)
Total:	37524960		15	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_117	1	1				
L_119	1	1				
L_124	1	1				

LOOP_8 _quant_non_intra.3

Program:	mpeg2_enc
File:	quantize.c
Function:	quant_non_intra
Header block:	HB_3
Loop blocks:	HB_3
Nesting level:	1
Innermost:	yes
Category:	MOD_SCHED
Invocations:	4494
Iterations:	287616
Iter/Invoc:	64
Operations:	7773563 (0.46%)
Cycles:	6402057 (1.12%)
Ops/Cyc:	1.21
Stall cycles:	284127 (4.44%)

Scheduling

	RecMII	ResMII	II	ESC
HB_3	21	4	21	1
Sched length:	1365			

Operation breakdown

	Dynamic counts		Static counts	
Load:	860544	(11%)	3	(10%)
Store:	286848	(4%)	1	(3%)
iAlu:	4586861	(59%)	20	(65%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	1747980	(22%)	6	(19%)
Pbr:	0	(0%)	0	(0%)
Branch:	291330	(4%)	1	(3%)
Total:	7773563		31	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_17	2	1				
L_20	1	1				
L_36	4	0				
S_48	2	1				

LOOP_9 _quant_intra.6

Program:	mpeg2_enc
File:	quantize.c
Function:	quant_intra
Header block:	HB_6
Loop blocks:	HB_6

Nesting level: 1
Innermost: yes
Category: MOD_SCHED
Invocations: 3954
Iterations: 249102
Iter/Invoc: 63
Operations: 6359104 (0.38%)
Cycles: 5807013 (1.01%)
Ops/Cyc: 1.10
Stall cycles: 222885 (3.84%)

Scheduling

	RecMII	ResMII	II	ESC
HB_6	22	4	22	1
Sched length:	1408			

Operation breakdown

	Dynamic counts		Static counts	
Load:	749574	(12%)	3	(10%)
Store:	249858	(4%)	1	(3%)
iAlu:	3836728	(60%)	19	(66%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	1269120	(20%)	5	(17%)
Pbr:	0	(0%)	0	(0%)
Branch:	253824	(4%)	1	(3%)
Total:	6359104		29	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_35	2	1				
L_38	1	1				
L_59	4	0				
S_71	2	1				

A.4 Mpeg2_dec

Innermost loops list

#	Loop name	Dyn Cyc	(%acc)	Dyn Ops	(%acc)	OPC	Inv	Iter	Nest	Cat	Ops	LDs	STs
1	_Fast_IDCT.8	7,054,772	(15%)	12,637,191	(15%)	1.79	7,920	3	L1	D	567	133	105
2	_Fast_IDCT.3	2,759,648	(22%)	5,319,957	(21%)	1.93	7,920	2	L1	D	332	66	63
3	_Add_Block.31	1,600,930	(25%)	2,539,680	(24%)	1.59	29,456	8	L2	M	12	4	1
4	_form_comp.58	995,785	(27%)	1,470,272	(25%)	1.48	12,352	11	L2	M	10	2	1
5	_form_comp.50	889,066	(29%)	1,133,312	(27%)	1.27	5,152	11	L1	W	29	3	2
6	_form_comp.10	785,524	(31%)	987,392	(28%)	1.26	5,872	12	L1	W	24	2	2
7	_Add_Block.36	715,310	(32%)	945,120	(29%)	1.32	10,961	7	L2	M	12	3	1
8	_Clear_Blo.3	514,800	(34%)	2,035,440	(31%)	3.95	7,920	64	L1	M	4	0	1
9	_form_comp.18	388,284	(34%)	761,536	(32%)	1.96	12,736	11	L1	M	5	1	1
10	_form_comp.73	254,821	(35%)	522,736	(33%)	2.05	2,544	12	L2	M	16	4	1
11	_form_comp.38	235,644	(35%)	416,592	(33%)	1.77	3,152	13	L1	M	10	2	1
12	_form_comp.65	218,971	(36%)	368,640	(34%)	1.68	928	13	L2	W	38	7	2
13	_form_comp.30	214,640	(36%)	291,840	(34%)	1.36	1,200	12	L2	W	27	3	2
14	_form_comp.19	85,963	(37%)	124,924	(34%)	1.45	12,736	1	L1	W	10	0	0
15	_form_comp.9	37,804	(37%)	57,620	(34%)	1.52	5,872	1	L1	W	10	0	0
16	_form_comp.49	33,064	(37%)	55,672	(34%)	1.68	5,152	1	L1	W	11	0	0
17	_Flush_Buf.15	27,613	(37%)	24,588	(34%)	0.89	1,311	1	L1	M	17	5	2
18	_form_comp.37	20,456	(37%)	34,136	(34%)	1.67	3,152	1	L1	W	11	0	0
19	_Fill_Buff.10	9,310	(37%)	10,270	(34%)	1.10	1	489	L1	M	21	4	4
20	_Initializ.3	3,609	(37%)	10,755	(34%)	2.98	1	1,024	L1	M	12	1	1
21	_Initializ.5	3,105	(37%)	10,115	(34%)	3.26	1	1,024	L1	M	11	1	1
22	_Update_Pi.3	710	(37%)	248	(34%)	0.35	4	3	L1	M	23	5	5
23	_sequence_.17	650	(37%)	1,089	(34%)	1.68	1	64	L1	M	17	4	2
24	_sequence_.7	260	(37%)	577	(34%)	2.22	1	64	L1	M	9	2	1
25	_sequence_.14	260	(37%)	449	(34%)	1.73	1	64	L1	M	7	1	1
26	_Fill_Buff.8	15	(37%)	18	(34%)	1.20	1	2	L1	W	9	1	1

Table A.4. Mpeg2_dec innermost loops list

Description of the most representative loops

LOOP_0 _Add_Block_getpic_i_1.31

Program: mpeg2_dec
File: getpic.c
Function: Add_Block_getpic_i_1
Header block: BB_31
Loop blocks: BB_31
Nesting level: 2
Innermost: yes
Category: MOD_SCHED
Invocations: 29456
Iterations: 235648
Iter/Invoc: 8
Operations: 2539680 (2.93%)
Cycles: 1600930 (3.51%)
Ops/Cyc: 1.59
Stall cycles: 77122 (4.82%)

Scheduling

	RecMII	ResMII	II	ESC
BB_31	6	2	6	1

Sched length: 54

Operation breakdown

	Dynamic counts		Static counts	
Load:	831168	(33%)	4	(33%)
Store:	207792	(8%)	1	(8%)

iAlu:	1246752	(49%)	6	(50%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	253968	(10%)	1	(8%)
Total:	2539680		12	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_175	4	0				
L_176	2	1				
L_177	1	1				
L_180	1	-				
S_181	1	1				

LOOP_1 _form_component_predi.58

Program:	mpeg2_dec
File:	recon.c
Function:	form_component_predi
Header block:	BB_58
Loop blocks:	BB_58
Nesting level:	2
Innermost:	yes
Category:	MOD_SCHED
Invocations:	12352
Iterations:	145792
Iter/Invoc:	12
Operations:	1470272 (1.70%)
Cycles:	995785 (2.18%)
Ops/Cyc:	1.48
Stall cycles:	205065 (20.59%)

Scheduling

	RecMII	ResMII	II	ESC
BB_58	5	2	5	1
Sched length:	64			

Operation breakdown

	Dynamic counts		Static counts	
Load:	291584	(20%)	2	(20%)
Store:	145792	(10%)	1	(10%)
iAlu:	874752	(60%)	6	(60%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	158144	(11%)	1	(10%)
Total:	1470272		10	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_231	1	1				
L_234	1	1				
S_239	1	1				

LOOP_2 _form_component_predi.50

Program:	mpeg2_dec
File:	recon.c
Function:	form_component_predi
Header block:	HB_50
Loop blocks:	HB_50 BB_52 HB_116
Nesting level:	1
Innermost:	yes
Category:	WHILE_LOOP
Invocations:	5152
Iterations:	59648
Iter/Invoc:	12

Operations: 1133312 (1.31%)
Cycles: 889066 (1.95%)
Ops/Cyc: 1.27
Stall cycles: 108490 (12.20%)

Scheduling

	wsl	pesl	per	wgt
HB_50	13.09	13	0.91	59648
BB_52	1.00	1	1.00	0
HB_116	9.00	9	1.00	0
Sched length:	152			

Operation breakdown

	Dynamic counts		Static counts	
Load:	178944	(16%)	3	(10%)
Store:	59648	(5%)	2	(7%)
iAlu:	536832	(47%)	13	(45%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	119296	(11%)	3	(10%)
Pbr:	119296	(11%)	4	(14%)
Branch:	119296	(11%)	4	(14%)
Total:	1133312		29	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_194	1	1				
L_196	1	1				
L_199	1	1				
S_212	1	1				
S_590	1	1				

LOOP_3 _form_component_predi.10

Program: mpeg2_dec
File: recon.c
Function: form_component_predi
Header block: HB_10
Loop blocks: HB_10 BB_12 HB_114
Nesting level: 1
Innermost: yes
Category: WHILE_LOOP
Invocations: 5872
Iterations: 70528
Iter/Invoc: 12
Operations: 987392 (1.14%)
Cycles: 785524 (1.72%)
Ops/Cyc: 1.26
Stall cycles: 74372 (9.47%)

Scheduling

	wsl	pesl	per	wgt
HB_10	10.08	10	0.92	70528
BB_12	1.00	1	1.00	0
HB_114	9.00	9	1.00	0
Sched length:	121			

Operation breakdown

	Dynamic counts		Static counts	
Load:	141056	(14%)	2	(8%)
Store:	70528	(7%)	2	(8%)
iAlu:	352640	(36%)	9	(38%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	141056	(14%)	3	(12%)
Pbr:	141056	(14%)	4	(17%)
Branch:	141056	(14%)	4	(17%)
Total:	987392		24	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_64	1	1				

L_66	1	1
S_76	1	1
S_583	1	1

LOOP_4 _Add_Block_getpic_i_1.36

Program: mpeg2_dec
File: getpic.c
Function: Add_Block_getpic_i_1
Header block: BB_36
Loop blocks: BB_36
Nesting level: 2
Innermost: yes
Category: MOD_SCHED
Invocations: 10961
Iterations: 87694
Iter/Invoc: 8
Operations: 945120 (1.09%)
Cycles: 715310 (1.57%)
Ops/Cyc: 1.32
Stall cycles: 53726 (7.51%)

Scheduling

	RecMII	ResMII	II	ESC
BB_36	7	2	7	1
Sched length:	63			

Operation breakdown

	Dynamic counts		Static counts	
Load:	231984	(25%)	3	(25%)
Store:	77328	(8%)	1	(8%)
iAlu:	541296	(57%)	7	(58%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	94512	(10%)	1	(8%)
Total:	945120		12	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_201	4	0				
L_202	2	1				
L_205	1	-				
S_211	1	1				

LOOP_5 _Clear_Block_getpic_i.3

Program: mpeg2_dec
File: getpic.c
Function: Clear_Block_getpic_i
Header block: BB_3
Loop blocks: BB_3
Nesting level: 1
Innermost: yes
Category: MOD_SCHED
Invocations: 7920
Iterations: 506880
Iter/Invoc: 64
Operations: 2035440 (2.35%)
Cycles: 514800 (1.13%)
Ops/Cyc: 3.95
Stall cycles: 0 (0.00%)

Scheduling

	RecMII	ResMII	II	ESC
BB_3	1	1	1	1
Sched length:	65			

Operation breakdown

	Dynamic counts		Static counts	
Load:	0	(0%)	0	(0%)
Store:	506880	(25%)	1	(25%)
iAlu:	1013760	(50%)	2	(50%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	514800	(25%)	1	(25%)
Total:	2035440		4	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
S_16	2	1				

A.5 Gsm_enc

Innermost loops list

#	Loop name	Dyn Cyc	(%acc)	Dyn Ops	(%acc)	OPC	Inv	Iter	Nest	Cat	Ops	LDs	STs
1	_Short_ter.5	18,440,000	(20%)	51,484,480	(22%)	2.79	147,520	8	L2	MS	47	2	1
2	_Calculati.27	11,152,512	(32%)	88,104,845	(59%)	7.90	3,688	81	L1	MS	310	52	1
3	_Autocorre.42	10,438,884	(43%)	11,632,874	(64%)	1.11	922	152	L1	MS	83	18	9
4	_Weighting.3	4,259,728	(48%)	8,124,664	(68%)	1.91	3,688	40	L1	MS	57	9	1
5	_Long_term.8	2,721,744	(51%)	2,961,464	(69%)	1.09	3,688	40	L1	MS	22	2	2
6	_Gsm_Coder.5	1,360,872	(52%)	2,371,384	(70%)	1.74	3,688	40	L2	MS	18	2	1
7	_Reflectio.52	1,097,180	(54%)	1,219,806	(70%)	1.11	6,454	4	L2	MS	50	6	2
8	_gsm_div .11	868,952	(55%)	1,528,864	(71%)	1.76	7,364	15	L1	W	15	0	0
9	_Calculati.3	634,336	(55%)	2,638,037	(72%)	4.16	3,688	40	L1	MS	20	1	0
10	_Calculati.25	604,832	(56%)	1,478,888	(73%)	2.45	3,688	40	L1	MS	10	2	1
11	_Autocorre.3	598,495	(57%)	2,301,585	(74%)	3.85	922	160	L1	MS	18	1	0
12	_LARp_to_r.4	390,385	(57%)	634,917	(74%)	1.63	3,688	8	L1	D	66	1	4
13	_RPE_grid_.12	376,710	(57%)	800,296	(74%)	2.12	3,688	13	L1	W	17	1	3
14	_Calculati.42	324,544	(58%)	1,342,432	(75%)	4.14	3,688	40	L1	MS	9	1	0
15	_Autocorre.51	276,146	(58%)	434,473	(75%)	1.57	247	160	L1	W	11	1	1
16	_APCM_quan.3	236,032	(58%)	894,977	(75%)	3.79	3,688	13	L1	MS	20	1	0
17	_Autocorre.24	189,336	(59%)	282,436	(76%)	1.49	196	160	L1	MS	9	1	1
18	_Coefficie.4	168,338	(59%)	211,138	(76%)	1.25	922	8	L1	MS	32	3	2
19	_Coefficie.4	167,804	(59%)	211,138	(76%)	1.26	922	8	L1	MS	32	3	2
20	_RPE_grid_.9	154,896	(59%)	435,184	(76%)	2.81	3,688	13	L1	MS	9	1	1
21	_Coefficie.4	84,824	(59%)	135,534	(76%)	1.60	922	8	L1	MS	20	2	1
22	_Autocorre.45	56,242	(59%)	98,654	(76%)	1.75	922	9	L1	W	12	1	1
23	_Autocorre.27	49,266	(59%)	73,491	(76%)	1.49	51	160	L1	MS	9	1	1
24	_Autocorre.38	39,646	(59%)	65,462	(76%)	1.65	922	9	L1	W	8	0	1
25	_Reflectio.18	36,880	(59%)	75,604	(76%)	2.05	922	9	L1	MS	9	1	1
26	_RPE_grid_.30	28,233	(59%)	34,255	(76%)	1.21	1,404	1	L1	W	13	0	1
27	_Reflectio.22	27,660	(59%)	67,306	(76%)	2.43	922	9	L1	MS	8	1	1
28	_Coefficie.4	24,894	(59%)	52,554	(76%)	2.11	922	8	L1	MS	7	1	1
29	_APCM_quan.7	22,801	(60%)	30,745	(76%)	1.35	1,516	1	L1	W	13	0	0
30	_Reflectio.20	22,128	(60%)	52,554	(76%)	2.38	922	7	L1	MS	8	1	1

Table A.5. Gsm_enc innermost loops list

Description of the most representative loops

LOOP_0 _Short_term_analysis_.5

Program:	gsm_enc
File:	short_term.c
Function:	Short_term_analysis_
Header block:	HB_5
Loop blocks:	HB_5
Nesting level:	2
Innermost:	yes
Category:	MOD_SCHED
Invocations:	147520
Iterations:	1180160
Iter/Invoc:	8
Operations:	51484480 (21.82%)
Cycles:	18440000 (20.01%)
Ops/Cyc:	2.79
Stall cycles:	1180160 (6.40%)

Scheduling

	RecMII	ResMII	II	ESC
HB_5	13	6	13	1
Sched length:	117			

Operation breakdown

	Dynamic counts		Static counts	
Load:	2360320	(5%)	2	(4%)
Store:	1180160	(2%)	1	(2%)
iAlu:	41305600	(80%)	39	(83%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	5310720	(10%)	4	(9%)
Pbr:	0	(0%)	0	(0%)
Branch:	1327680	(3%)	1	(2%)
Total:	51484480		47	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_27	2	1				
L_30	2	1				
S_35	2	1				

LOOP_1 _Calculation_of_the_L.27

Program:	gsm_enc
File:	long_term.c
Function:	Calculation_of_the_L
Header block:	HB_27
Loop blocks:	HB_27
Nesting level:	1
Innermost:	yes
Category:	MOD_SCHED
Invocations:	3688
Iterations:	298728
Iter/Invoc:	81
Operations:	88104845 (37.34%)
Cycles:	11152512 (12.10%)
Ops/Cyc:	7.90
Stall cycles:	0 (0.00%)

Scheduling

	RecMII	ResMII	II	ESC
HB_27	2	36	36	3
Sched length:	3024			

Operation breakdown

	Dynamic counts		Static counts	
Load:	15533856	(17%)	52	(17%)
Store:	15388	(0%)	1	(0%)
iAlu:	75325620	(82%)	255	(82%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	309792	(0%)	1	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	309792	(0%)	1	(0%)
Total:	88104845		310	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_111	2	-1	G_111	40	80	1
L_118	2	-1	"			
L_126	2	-1	"			
L_134	2	-1	"			
L_142	2	-1	"			
L_150	2	-1	"			
L_158	2	-1	"			
L_166	2	-1	"			
L_174	2	-1	"			
L_182	2	-1	"			
L_190	2	-1	"			
L_198	2	-1	"			
L_206	2	-1	"			
L_214	2	-1	"			
L_222	2	-1	"			
L_230	2	-1	"			

L_238	2	-1	"
L_246	2	-1	"
L_254	2	-1	"
L_262	2	-1	"
L_270	2	-1	"
L_278	2	-1	"
L_286	2	-1	"
L_294	2	-1	"
L_302	2	-1	"
L_310	2	-1	"
L_318	2	-1	"
L_326	2	-1	"
L_334	2	-1	"
L_342	2	-1	"
L_350	2	-1	"
L_358	2	-1	"
L_366	2	-1	"
L_374	2	-1	"
L_382	2	-1	"
L_390	2	-1	"
L_398	2	-1	"
L_406	2	-1	"
L_414	2	-1	"
L_422	2	-1	"

	Dynamic count		Static count	
Spill:	3600124	(4%)	13	(4%)

LOOP_2 _Autocorrelation_lpc_.42

Program:	gsm_enc
File:	lpc.c
Function:	Autocorrelation_lpc_
Header block:	BB_42
Loop blocks:	BB_42
Nesting level:	1
Innermost:	yes
Category:	MOD_SCHED
Invocations:	922
Iterations:	140144
Iter/Invoc:	152
Operations:	11632874 (4.93%)
Cycles:	10438884 (11.33%)
Ops/Cyc:	1.11
Stall cycles:	0 (0.00%)

Scheduling

	RecMII	ResMII	II	ESC
BB_42	74	11	74	1
Sched length:	11322			

Operation breakdown

	Dynamic counts		Static counts	
Load:	2522592	(22%)	18	(22%)
Store:	1261296	(11%)	9	(11%)
iAlu:	7707920	(66%)	55	(66%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	141066	(1%)	1	(1%)
Total:	11632874		83	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_442	4	0	G_442	9	36	1
L_449	4	0	"			
L_456	4	0	"			
L_463	4	0	"			
L_470	4	0	"			

L_477	4	0	"			
L_484	4	0	"			
L_491	4	0	"			
L_498	4	0	"			
L_440	2	1	G_501	9	18	1
L_452	2	1	"			
L_459	2	1	"			
L_466	2	1	"			
L_473	2	1	"			
L_480	2	1	"			
L_487	2	1	"			
L_494	2	1	"			
L_501	2	1	"			
S_448	4	0	G_448	9	36	1
S_455	4	0	"			
S_462	4	0	"			
S_469	4	0	"			
S_476	4	0	"			
S_483	4	0	"			
S_490	4	0	"			
S_497	4	0	"			
S_504	4	0	"			

LOOP_3 _Weighting_filter_rpe.3

Program:	gsm_enc
File:	rpe.c
Function:	Weighting_filter_rpe
Header block:	HB_3
Loop blocks:	HB_3
Nesting level:	1
Innermost:	yes
Category:	MOD_SCHED
Invocations:	3688
Iterations:	147520
Iter/Invoc:	40
Operations:	8124664 (3.44%)
Cycles:	4259728 (4.62%)
Ops/Cyc:	1.91
Stall cycles:	781944 (18.36%)

Scheduling

	RecMII	ResMII	II	ESC
HB_3	22	8	23	1
Sched length:	943			

Operation breakdown

	Dynamic counts		Static counts	
Load:	1327680	(16%)	9	(16%)
Store:	147520	(2%)	1	(2%)
iAlu:	6195840	(76%)	44	(77%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	302416	(4%)	2	(4%)
Pbr:	0	(0%)	0	(0%)
Branch:	151208	(2%)	1	(2%)
Total:	8124664		57	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_14	2	1				
L_20	2	1				
L_26	2	1				
L_32	2	1				
L_38	2	1				
L_44	2	1				
L_50	2	1				
L_56	2	1				
L_62	2	1				

S_77	2	1

LOOP_4 _Long_term_analysis_f.8

Program:	gsm_enc
File:	long_term.c
Function:	Long_term_analysis_f
Header block:	HB_8
Loop blocks:	HB_8
Nesting level:	1
Innermost:	yes
Category:	MOD_SCHED
Invocations:	3688
Iterations:	147520
Iter/Invoc:	40
Operations:	2961464 (1.26%)
Cycles:	2721744 (2.95%)
Ops/Cyc:	1.09
Stall cycles:	0 (0.00%)

Scheduling

	RecMII	ResMII	II	ESC
HB_8	18	3	18	1
Sched length:	738			

Operation breakdown

	Dynamic counts		Static counts	
Load:	295040	(10%)	2	(9%)
Store:	295040	(10%)	2	(9%)
iAlu:	1917760	(65%)	15	(68%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	302416	(10%)	2	(9%)
Pbr:	0	(0%)	0	(0%)
Branch:	151208	(5%)	1	(5%)
Total:	2961464		22	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_36	2	1				
L_43	2	1				
S_41	2	1				
S_58	2	1				

LOOP_5 _Gsm_Coder.5

Program:	gsm_enc
File:	code.c
Function:	Gsm_Coder
Header block:	HB_5
Loop blocks:	HB_5
Nesting level:	2
Innermost:	yes
Category:	MOD_SCHED
Invocations:	3688
Iterations:	147520
Iter/Invoc:	40
Operations:	2371384 (1.01%)
Cycles:	1360872 (1.48%)
Ops/Cyc:	1.74
Stall cycles:	0 (0.00%)

Scheduling

	RecMII	ResMII	II	ESC
HB_5	9	3	9	1
Sched length:	369			

Operation breakdown

	Dynamic counts		Static counts	
Load:	295040	(12%)	2	(11%)

Store:	147520	(6%)	1	(6%)
iAlu:	1475200	(62%)	12	(67%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	302416	(13%)	2	(11%)
Pbr:	0	(0%)	0	(0%)
Branch:	151208	(6%)	1	(6%)
Total:	2371384		18	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_80	2	1				
L_82	2	1				
S_96	2	1				

LOOP_6 _Reflection_coefficie.52

Program:	gsm_enc
File:	lpc.c
Function:	Reflection_coefficie
Header block:	HB_52
Loop blocks:	HB_52
Nesting level:	2
Innermost:	yes
Category:	MOD_SCHED
Invocations:	6454
Iterations:	25816
Iter/Invoc:	4
Operations:	1219806 (0.52%)
Cycles:	1097180 (1.19%)
Ops/Cyc:	1.11
Stall cycles:	0 (0.00%)

Scheduling

	RecMII	ResMII	II	ESC
HB_52	34	7	34	1
Sched length:	170			

Operation breakdown

	Dynamic counts		Static counts	
Load:	154896	(13%)	6	(12%)
Store:	51632	(4%)	2	(4%)
iAlu:	851928	(70%)	37	(74%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	129080	(11%)	4	(8%)
Pbr:	0	(0%)	0	(0%)
Branch:	32270	(3%)	1	(2%)
Total:	1219806		50	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_194	2	1				
L_195	2	0				
L_203	2	1				
L_224	2	1				
L_225	2	0				
L_232	2	1				
S_220	2	1				
S_249	2	1				

A.6 Gsm_dec

Innermost loops list

#	Loop name	Dyn Cyc	(%acc)	Dyn Ops	(%acc)	OPC	Inv	Iter	Nest	Cat	Ops	LDs	STs
1	_Short_ter.6	70,375,431	(82%)	102,670,232	(82%)	1.46	3,688	320	L1	W	131	19	13
2	_Postproce.4	4,276,812	(87%)	4,867,238	(85%)	1.14	922	160	L1	W	37	1	1
3	_Gsm_Long_.16	2,602,389	(90%)	3,108,984	(88%)	1.19	3,688	40	L1	M	23	2	1
4	_Gsm_Long_.24	1,339,309	(91%)	3,101,608	(90%)	2.32	3,688	120	L1	M	7	1	1
5	_Gsm_Decod.5	453,705	(92%)	1,036,328	(91%)	2.28	3,688	40	L2	M	7	1	1
6	_LARp_to_r.4	390,385	(92%)	634,917	(92%)	1.63	3,688	8	L1	D	66	1	4
7	_RPE_grid_.12	376,710	(93%)	800,296	(92%)	2.12	3,688	13	L1	W	17	1	3
8	_Coefficie.4	168,338	(93%)	211,138	(92%)	1.25	922	8	L1	M	32	3	2
9	_Coefficie.4	167,804	(93%)	211,138	(93%)	1.26	922	8	L1	M	32	3	2
10	_Coefficie.4	84,824	(93%)	135,534	(93%)	1.60	922	8	L1	M	20	2	1
11	_RPE_grid_.30	28,233	(93%)	34,255	(93%)	1.21	1,404	1	L1	W	13	0	1
12	_Coefficie.4	24,894	(93%)	52,554	(93%)	2.11	922	8	L1	M	7	1	1
13	_APCM_quan.7	22,801	(93%)	30,745	(93%)	1.35	1,516	1	L1	W	13	0	0

Table A.6. Gsm_dec innermost loops list

Description of the most representative loops

LOOP_0 _Short_term_synthesis.6

Program: gsm_dec
File: short_term.c
Function: Short_term_synthesis
Header block: HB_6
Loop blocks: HB_6 HB_31
Nesting level: 1
Innermost: yes
Category: WHILE_LOOP
Invocations: 3688
Iterations: 1180160
Iter/Invoc: 320
Operations: 102670232 (81.53%)
Cycles: 70375431 (81.84%)
Ops/Cyc: 1.46
Stall cycles: 753367 (1.07%)

Scheduling

	wsl	pesl	per	wgt
HB_6	56.75	57	0.88	1180160
HB_31	17.95	18	0.97	147520
Sched length:	18878			

Operation breakdown

	Dynamic counts		Static counts	
Load:	13129280	(13%)	19	(14%)
Store:	10178880	(10%)	13	(10%)
iAlu:	64613760	(63%)	82	(63%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	9588800	(9%)	9	(7%)
Pbr:	2655360	(3%)	4	(3%)
Branch:	2504152	(2%)	4	(3%)
Total:	102670232		131	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_31	2	-				
L_34	2	-				
L_190	2	1				
S_114	2	-				

S_122	2	0
S_129	2	1

	Dynamic count		Static count	
Spill:	19325120	(19%)	26	(20%)

LOOP_1 _Postprocessing_decod.4

Program: gsm_dec
File: decode.c
Function: Postprocessing_decod
Header block: HB_4
Loop blocks: HB_4
Nesting level: 1
Innermost: yes
Category: WHILE_LOOP
Invocations: 922
Iterations: 147520
Iter/Invoc: 160
Operations: 4867238 (3.86%)
Cycles: 4276812 (4.97%)
Ops/Cyc: 1.14
Stall cycles: 576 (0.01%)

Scheduling

	wsl	pesl	per	wgt
HB_4	28.99	29	0.99	147520
Sched length:	4638			

Operation breakdown

	Dynamic counts		Static counts	
Load:	147520	(3%)	1	(3%)
Store:	147520	(3%)	1	(3%)
iAlu:	3245440	(67%)	26	(70%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	737600	(15%)	5	(14%)
Pbr:	295040	(6%)	2	(5%)
Branch:	294118	(6%)	2	(5%)
Total:	4867238		37	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_21	2	1				
S_54	2	1				

LOOP_2 _Gsm_Long_Term_Synthe.16

Program: gsm_dec
File: long_term.c
Function: Gsm_Long_Term_Synthe
Header block: HB_16
Loop blocks: HB_16
Nesting level: 1
Innermost: yes
Category: MOD_SCHED
Invocations: 3688
Iterations: 147520
Iter/Invoc: 40
Operations: 3108984 (2.47%)
Cycles: 2602389 (3.03%)
Ops/Cyc: 1.19
Stall cycles: 31853 (1.22%)

Scheduling

	RecMII	ResMII	II	ESC
HB_16	17	3	17	1
Sched length:	697			

Operation breakdown

	Dynamic counts	Static counts

Load:	295040	(9%)	2	(9%)
Store:	147520	(5%)	1	(4%)
iAlu:	2212800	(71%)	17	(74%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	302416	(10%)	2	(9%)
Pbr:	0	(0%)	0	(0%)
Branch:	151208	(5%)	1	(4%)
Total:	3108984		23	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_73	2	1				
L_79	2	1				
S_95	2	1				

LOOP_3 _Gsm_Long_Term_Synthe.24

Program:	gsm_dec
File:	long_term.c
Function:	Gsm_Long_Term_Synthe
Header block:	BB_24
Loop blocks:	BB_24
Nesting level:	1
Innermost:	yes
Category:	MOD_SCHED
Invocations:	3688
Iterations:	442560
Iter/Invoc:	120
Operations:	3101608 (2.46%)
Cycles:	1339309 (1.56%)
Ops/Cyc:	2.32
Stall cycles:	565 (0.04%)

Scheduling

	RecMII	ResMII	II	ESC
BB_24	3	1	3	1
Sched length:	363			

Operation breakdown

	Dynamic counts		Static counts	
Load:	442560	(14%)	1	(14%)
Store:	442560	(14%)	1	(14%)
iAlu:	1770240	(57%)	4	(57%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	446248	(14%)	1	(14%)
Total:	3101608		7	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_103	2	1				
S_106	2	1				

A.7 Epic_enc

Innermost loops list

#	Loop name	Dyn Cyc	(%acc)	Dyn Ops	(%acc)	OPC	Inv	Iter	Nest	Cat	Ops	LDs	STs
1	_internal_.75	24,301,056	(33%)	29,508,864	(39%)	1.21	1,243,968	1	L4	M	11	2	0
2	_internal_.73	7,463,808	(43%)	3,731,904	(44%)	0.50	1,243,968	1	L4	W	7	0	0
3	_quantize_.11	1,617,596	(45%)	1,145,883	(46%)	0.71	13	5,041	L1	M	19	3	1
4	_main.5	1,048,577	(46%)	1,245,184	(47%)	1.19	1	65,536	L1	W	19	5	1
5	_internal_.71	937,344	(48%)	1,093,568	(49%)	1.17	156,224	1	L3	W	15	0	1
6	_internal_.127	920,832	(49%)	849,408	(50%)	0.92	56,096	1	L4	M	11	2	0
7	_internal_.26	920,832	(50%)	849,408	(51%)	0.92	56,096	1	L3	M	11	2	0
8	_encode_st.7	547,930	(51%)	951,819	(52%)	1.74	13,998	3	L2	W	22	3	2
9	_internal_.58	471,552	(51%)	809,088	(53%)	1.72	16,256	4	L4	M	11	2	0
10	_internal_.93	471,552	(52%)	809,088	(54%)	1.72	16,256	4	L4	M	11	2	0
11	_run_lengt.5	438,094	(53%)	398,944	(55%)	0.91	3,072	18	L2	W	7	1	0
12	_quantize_.3	408,868	(53%)	557,019	(56%)	1.36	13	5,041	L1	M	10	1	0
13	_internal_.16	342,722	(54%)	412,808	(56%)	1.20	7,201	3	L2	W	21	2	2
14	_internal_.125	336,576	(54%)	168,288	(57%)	0.50	56,096	1	L4	W	7	0	0
15	_internal_.24	336,576	(55%)	168,288	(57%)	0.50	56,096	1	L3	W	7	0	0
16	_reflect1.108	181,602	(55%)	514,656	(57%)	2.83	6,696	1	L2	M	41	2	1
17	_huffman_e.3	129,321	(55%)	262,145	(58%)	2.03	1	65,536	L1	M	4	0	1
18	_internal_.4	129,168	(55%)	53,820	(58%)	0.42	7,201	1	L2	W	5	0	0
19	_internal_.56	97,536	(55%)	48,768	(58%)	0.50	16,256	1	L4	W	7	0	0
20	_internal_.91	97,536	(55%)	48,768	(58%)	0.50	16,256	1	L4	W	7	0	0
21	_reflect1.36	40,176	(56%)	26,784	(58%)	0.67	6,696	1	L2	W	7	0	0
22	_internal_.54	27,648	(56%)	32,256	(58%)	1.17	4,608	1	L3	W	15	0	1
23	_internal_.89	27,648	(56%)	18,432	(58%)	0.67	4,608	1	L3	W	12	0	1
24	_internal_.123	24,768	(56%)	16,512	(58%)	0.67	4,128	1	L3	W	12	0	1
25	_internal_.22	24,768	(56%)	16,512	(58%)	0.67	4,128	1	L2	W	14	1	1
26	_internal_.11	17,664	(56%)	18,816	(58%)	1.07	992	1	L3	M	11	2	0
27	_internal_.112	17,664	(56%)	18,816	(58%)	1.07	992	1	L3	M	11	2	0
28	_internal_.142	17,664	(56%)	18,816	(58%)	1.07	992	1	L3	M	11	2	0
29	_internal_.41	17,664	(56%)	18,816	(58%)	1.07	992	1	L2	M	11	2	0
30	_internal_.69	15,984	(56%)	17,760	(58%)	1.11	1,776	1	L2	W	20	4	0
31	_insert_in.13	15,605	(56%)	16,320	(58%)	1.05	106	14	L1	W	11	3	0
32	_reflect1.21	10,368	(56%)	49,248	(58%)	4.75	648	15	L1	M	5	0	1
33	_internal_.110	5,952	(56%)	2,976	(58%)	0.50	992	1	L3	W	7	0	0
34	_internal_.140	5,952	(56%)	2,976	(58%)	0.50	992	1	L3	W	7	0	0
35	_internal_.39	5,952	(56%)	2,976	(58%)	0.50	992	1	L2	W	7	0	0
36	_internal_.9	5,952	(56%)	2,976	(58%)	0.50	992	1	L3	W	7	0	0
37	_pack_tree.4	1,035	(56%)	1,449	(58%)	1.40	69	2	L1	M	10	2	2
38	_parse_epi.90	117	(56%)	85	(58%)	0.73	1	4	L1	W	22	3	1

Table A.7. Epic_enc innermost loops list

Description of the most representative loops

LOOP_0 _internal_filter.75

Program:	epic_enc
File:	convolve.c
Function:	internal_filter
Header block:	HB_75
Loop blocks:	HB_75
Nesting level:	4
Innermost:	yes
Category:	MOD_SCHED
Invocations:	1243968
Iterations:	2343360

Iter/Invoc: 2
Operations: 29508864 (39.22%)
Cycles: 24301056 (32.72%)
Ops/Cyc: 1.21
Stall cycles: 0 (0.00%)

Scheduling

	RecMII	ResMII	II	ESC
HB_75	4	2	4	3
Sched length:	20			

Operation breakdown

	Dynamic counts		Static counts	
Load:	4686720	(16%)	2	(18%)
Store:	0	(0%)	0	(0%)
iAlu:	11716800	(40%)	5	(45%)
fAlu:	7030080	(24%)	3	(27%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	6075264	(21%)	1	(9%)
Total:	29508864		11	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_322	4	1				
L_324	4	1				

LOOP_1 _internal_filter.73

Program: epic_enc
File: convolve.c
Function: internal_filter
Header block: HB_73
Loop blocks: HB_73
Nesting level: 4
Innermost: yes
Category: WHILE_LOOP
Invocations: 1243968
Iterations: 1243968
Iter/Invoc: 1
Operations: 3731904 (4.96%)
Cycles: 7463808 (10.05%)
Ops/Cyc: 0.50
Stall cycles: 0 (0.00%)

Scheduling

	wsl	pesl	per	wgt
HB_73	6.00	6	1.00	1243968
Sched length:	6			

Operation breakdown

	Dynamic counts		Static counts	
Load:	0	(0%)	0	(0%)
Store:	0	(0%)	0	(0%)
iAlu:	1243968	(33%)	3	(43%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	2487936	(67%)	2	(29%)
Pbr:	0	(0%)	1	(14%)
Branch:	0	(0%)	1	(14%)
Total:	3731904		7	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr

LOOP_2 _quantize_image.11

Program: epic_enc
File: quantize.c
Function: quantize_image
Header block: HB_11

Loop blocks: HB_11
Nesting level: 1
Innermost: yes
Category: MOD_SCHED
Invocations: 13
Iterations: 65536
Iter/Invoc: 5041
Operations: 1145883 (1.52%)
Cycles: 1617596 (2.18%)
Ops/Cyc: 0.71
Stall cycles: 44420 (2.75%)

Scheduling

	RecMII	ResMII	II	ESC
HB_11	24	3	24	1
Sched length:	121014			

Operation breakdown

	Dynamic counts		Static counts	
Load:	163499	(14%)	3	(16%)
Store:	65536	(6%)	1	(5%)
iAlu:	491179	(43%)	8	(42%)
fAlu:	294571	(26%)	5	(26%)
Cmpp:	65549	(6%)	1	(5%)
Pbr:	0	(0%)	0	(0%)
Branch:	65549	(6%)	1	(5%)
Total:	1145883		19	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_56	4	1				
L_133	8	0				
L_135	8	0				
S_71	2	1				

LOOP_3 _main.5

Program: epic_enc
File: epic.c
Function: main
Header block: HB_5
Loop blocks: HB_5
Nesting level: 1
Innermost: yes
Category: WHILE_LOOP
Invocations: 1
Iterations: 65536
Iter/Invoc: 65536
Operations: 1245184 (1.66%)
Cycles: 1048577 (1.41%)
Ops/Cyc: 1.19
Stall cycles: 0 (0.00%)

Scheduling

	wsl	pesl	per	wgt
HB_5	16.00	16	1.00	65536
Sched length:	1048577			

Operation breakdown

	Dynamic counts		Static counts	
Load:	327680	(26%)	5	(26%)
Store:	65536	(5%)	1	(5%)
iAlu:	589824	(47%)	9	(47%)
fAlu:	65536	(5%)	1	(5%)
Cmpp:	65536	(5%)	1	(5%)
Pbr:	65536	(5%)	1	(5%)
Branch:	65536	(5%)	1	(5%)
Total:	1245184		19	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_88	4	0				
L_90	4	-				
L_99	4	0				
L_100	4	0				
L_686	4	0				
S_96	4	-				

LOOP_4 _internal_filter.71

Program: epic_enc
File: convolve.c
Function: internal_filter
Header block: HB_71
Loop blocks: HB_71
Nesting level: 3
Innermost: yes
Category: WHILE_LOOP
Invocations: 156224
Iterations: 156224
Iter/Invoc: 1
Operations: 1093568 (1.45%)
Cycles: 937344 (1.26%)
Ops/Cyc: 1.17
Stall cycles: 0 (0.00%)

Scheduling

	wsl	pesl	per	wgt
HB_71	6.00	6	1.00	156224
Sched length:	6			

Operation breakdown

	Dynamic counts		Static counts	
Load:	0	(0%)	0	(0%)
Store:	0	(0%)	1	(7%)
iAlu:	624896	(57%)	9	(60%)
fAlu:	156224	(14%)	1	(7%)
Cmpp:	312448	(29%)	2	(13%)
Pbr:	0	(0%)	1	(7%)
Branch:	0	(0%)	1	(7%)
Total:	1093568		15	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
S_343	4	1				

LOOP_5 _internal_filter.127

Program: epic_enc
File: convolve.c
Function: internal_filter
Header block: HB_127
Loop blocks: HB_127
Nesting level: 4
Innermost: yes
Category: MOD_SCHED
Invocations: 56096
Iterations: 61920
Iter/Invoc: 1
Operations: 849408 (1.13%)
Cycles: 920832 (1.24%)
Ops/Cyc: 0.92
Stall cycles: 0 (0.00%)

Scheduling

	RecMII	ResMII	II	ESC
HB_127	4	2	4	3
Sched length:	16			

Operation breakdown

	Dynamic counts		Static counts	
Load:	123840	(15%)	2	(18%)
Store:	0	(0%)	0	(0%)
iAlu:	309600	(36%)	5	(45%)
fAlu:	185760	(22%)	3	(27%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	230208	(27%)	1	(9%)
Total:	849408		11	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_495	4	1				
L_497	4	1				

LOOP_6 _internal_filter.26

Program:	epic_enc
File:	convolve.c
Function:	internal_filter
Header block:	HB_26
Loop blocks:	HB_26
Nesting level:	3
Innermost:	yes
Category:	MOD_SCHED
Invocations:	56096
Iterations:	61920
Iter/Invoc:	1
Operations:	849408 (1.13%)
Cycles:	920832 (1.24%)
Ops/Cyc:	0.92
Stall cycles:	0 (0.00%)

Scheduling

	RecMII	ResMII	II	ESC
HB_26	4	2	4	3
Sched length:	16			

Operation breakdown

	Dynamic counts		Static counts	
Load:	123840	(15%)	2	(18%)
Store:	0	(0%)	0	(0%)
iAlu:	309600	(36%)	5	(45%)
fAlu:	185760	(22%)	3	(27%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	230208	(27%)	1	(9%)
Total:	849408		11	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_151	4	1				
L_153	4	1				

A.8 Epic_dec

Innermost loops list

#	Loop name	Dyn Cyc	(%acc)	Dyn Ops	(%acc)	OPC	Inv	Iter	Nest	Cat	Ops	LDs	STs
1	_unquantiz.3	1,289,469	(18%)	1,114,151	(13%)	0.86	13	5,041	L1	M	21	1	3
2	_main.18	1,179,666	(34%)	1,114,015	(25%)	0.94	1	65,536	L1	M	19	4	3
3	_collapse_.9	626,525	(43%)	1,149,793	(38%)	1.84	236	90	L3	M	54	10	9
4	_collapse_.56	412,965	(49%)	855,097	(47%)	2.07	3,254	1	L6	W	214	50	40
5	_collapse_.147	364,726	(54%)	577,318	(54%)	1.58	235	90	L3	W	212	48	38
6	_internal_.16	345,804	(59%)	412,808	(59%)	1.19	7,201	3	L2	W	21	2	2
7	_collapse_.102	224,085	(62%)	463,910	(64%)	2.07	1,949	1	L4	W	214	50	40
8	_internal_.4	191,592	(65%)	53,820	(64%)	0.28	7,201	1	L2	W	5	0	0
9	_write_pgm.3	131,074	(67%)	458,753	(70%)	3.50	1	65,536	L1	M	7	1	1
10	_collapse_.4	87,628	(68%)	348,164	(73%)	3.97	4	21,760	L2	M	4	0	1
11	_run_lengt.6	62,704	(69%)	290,676	(77%)	4.64	5,711	9	L2	M	5	0	1
12	_collapse_.191	43,014	(69%)	129,027	(78%)	3.00	3	7,168	L2	M	6	1	1
13	_collapse_.263	17,001	(69%)	35,872	(79%)	2.11	236	1	L3	W	165	29	26
14	_collapse_.268	14,632	(70%)	32,096	(79%)	2.19	236	1	L3	W	151	22	26
15	_collapse_.273	14,632	(70%)	32,096	(79%)	2.19	236	1	L3	W	151	22	26
16	_collapse_.278	14,632	(70%)	32,096	(80%)	2.19	236	1	L3	W	151	22	26
17	_collapse_.19	4,849	(70%)	8,976	(80%)	1.85	4	59	L2	M	38	7	6
18	_collapse_.72	4,800	(70%)	7,374	(80%)	1.54	4	59	L2	M	38	7	6
19	_collapse_.125	4,800	(70%)	7,246	(80%)	1.51	4	59	L2	M	38	7	6
20	_collapse_.178	4,800	(70%)	7,230	(80%)	1.51	4	59	L2	M	38	7	6
21	_collapse_.171	4,800	(70%)	7,146	(80%)	1.49	4	59	L2	M	38	7	6
22	_collapse_.65	4,800	(70%)	6,876	(80%)	1.43	4	59	L2	M	38	7	6
23	_collapse_.118	3,840	(71%)	6,913	(80%)	1.80	4	59	L2	M	37	7	6
24	_collapse_.26	3,153	(71%)	8,032	(80%)	2.55	4	59	L2	M	34	7	6
25	_collapse_.33	2,645	(71%)	5,436	(80%)	2.06	4	59	L2	M	23	4	3
26	_collapse_.79	2,640	(71%)	4,611	(80%)	1.75	4	59	L2	M	23	4	3
27	_collapse_.86	2,640	(71%)	4,403	(80%)	1.67	4	59	L2	M	21	4	3
28	_collapse_.40	2,400	(71%)	5,200	(81%)	2.17	4	59	L2	M	22	4	3
29	_collapse_.132	2,400	(71%)	4,618	(81%)	1.92	4	59	L2	M	22	4	3
30	_collapse_.164	2,400	(71%)	4,546	(81%)	1.89	4	59	L2	M	22	4	3
31	_collapse_.111	1,440	(71%)	4,341	(81%)	3.01	4	59	L2	M	23	4	3
32	_collapse_.157	1,200	(71%)	4,105	(81%)	3.42	4	59	L2	M	22	4	3

Table A.8. Epic_dec innermost loops list

Description of the most representative loops

LOOP_0 _unquantize_image.3

Program:	epic_dec
File:	quantize.c
Function:	unquantize_image
Header block:	HB_3
Loop blocks:	HB_3
Nesting level:	1
Innermost:	yes
Category:	MOD_SCHED
Invocations:	13
Iterations:	65536
Iter/Invoc:	5041
Operations:	1114151 (12.50%)
Cycles:	1289469 (18.00%)
Ops/Cyc:	0.86
Stall cycles:	44038 (3.42%)
Scheduling	

RecMII ResMII II ESC

	RecMII	ResMII	II	ESC
HB_3	19	3	19	1
Sched length:	95802			

Operation breakdown

	Dynamic counts		Static counts	
Load:	65536	(6%)	1	(5%)
Store:	65536	(6%)	3	(14%)
iAlu:	458752	(41%)	9	(43%)
fAlu:	327680	(29%)	5	(24%)
Cmpp:	131098	(12%)	2	(10%)
Pbr:	0	(0%)	0	(0%)
Branch:	65549	(6%)	1	(5%)
Total:	1114151		21	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_22	2	1				
S_32	4	1				
S_45	4	1				
S_48	4	1				

LOOP_1 _main.18

Program: epic_dec
File: unepic.c
Function: main
Header block: HB_18
Loop blocks: HB_18
Nesting level: 1
Innermost: yes
Category: MOD_SCHED
Invocations: 1
Iterations: 65536
Iter/Invoc: 65536
Operations: 1114015 (12.50%)
Cycles: 1179666 (16.46%)
Ops/Cyc: 0.94
Stall cycles: 0 (0.00%)

Scheduling

	RecMII	ResMII	II	ESC
HB_18	18	3	18	1
Sched length:	1179666			

Operation breakdown

	Dynamic counts		Static counts	
Load:	262094	(24%)	4	(21%)
Store:	65536	(6%)	3	(16%)
iAlu:	327630	(29%)	5	(26%)
fAlu:	262144	(24%)	4	(21%)
Cmpp:	131074	(12%)	2	(11%)
Pbr:	0	(0%)	0	(0%)
Branch:	65537	(6%)	1	(5%)
Total:	1114015		19	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_390	4	1				
L_392	8	0				
L_652	8	0				
L_654	8	0				
S_398	4	1				
S_403	4	1				
S_408	4	1				

LOOP_2 _collapse_pyr.9

Program: epic_dec
File: collapse_pyr.c

Function: collapse_pyr
Header block: HB_9
Loop blocks: HB_9
Nesting level: 3
Innermost: yes
Category: MOD_SCHED
Invocations: 236
Iterations: 21284
Iter/Invoc: 90
Operations: 1149793 (12.90%)
Cycles: 626525 (8.74%)
Ops/Cyc: 1.84
Stall cycles: 2445 (0.39%)

Scheduling

	RecMII	ResMII	II	ESC
HB_9	28	7	29	1
Sched length:	2644			

Operation breakdown

	Dynamic counts		Static counts	
Load:	212840	(19%)	10	(19%)
Store:	191547	(17%)	9	(17%)
iAlu:	702366	(61%)	33	(61%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	21520	(2%)	1	(2%)
Pbr:	0	(0%)	0	(0%)
Branch:	21520	(2%)	1	(2%)
Total:	1149793		54	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_47	4	1				
L_51	4	2				
L_60	4	2				
L_67	4	2				
L_77	4	2				
L_86	4	2				
L_93	4	2				
L_104	4	2				
L_110	4	2				
L_117	4	2				
S_54	4	2	G_64	3	12	1
S_64	4	2	"			
S_71	4	2	"			
S_80	4	2	G_90	3	12	1
S_90	4	2	"			
S_97	4	2	"			
S_107	4	2	G_114	3	12	1
S_114	4	2	"			
S_121	4	2	"			

LOOP_3 _collapse_pyr.56

Program: epic_dec
File: collapse_pyr.c
Function: collapse_pyr
Header block: HB_56
Loop blocks: HB_56
Nesting level: 6
Innermost: yes
Category: WHILE_LOOP
Invocations: 3254
Iterations: 3996
Iter/Invoc: 1
Operations: 855097 (9.59%)
Cycles: 412965 (5.76%)
Ops/Cyc: 2.07

Stall cycles: 0 (0.00%)

Scheduling

	wsl	pesl	per	wgt
HB_56	103.78	104	0.80	3996
Sched length:	127			

Operation breakdown

	Dynamic counts		Static counts	
Load:	199800	(23%)	50	(23%)
Store:	159840	(19%)	40	(19%)
iAlu:	471528	(55%)	118	(55%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	7992	(1%)	2	(1%)
Pbr:	7992	(1%)	2	(1%)
Branch:	7945	(1%)	2	(1%)
Total:	855097		214	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_466	4	-				
L_475	4	-				
L_482	4	-				
L_492	4	-				
L_501	4	-				
L_508	4	-				
L_519	4	-				
L_525	4	-				
L_532	4	-				
L_3497	4	1				
S_469	4	-	G_479	3	12	1
S_479	4	-	"			
S_486	4	-	"			
S_495	4	-	G_505	3	12	1
S_505	4	-	"			
S_512	4	-	"			
S_522	4	-	G_529	3	12	1
S_529	4	-	"			
S_536	4	-	"			

	Dynamic count		Static count	
Spill:	283716	(33%)	71	(33%)

LOOP_4 _collapse_pyr.147

Program: epic_dec
File: collapse_pyr.c
Function: collapse_pyr
Header block: HB_147
Loop blocks: HB_147 HB_148
Nesting level: 3
Innermost: yes
Category: WHILE_LOOP
Invocations: 235
Iterations: 21284
Iter/Invoc: 90
Operations: 577318 (6.48%)
Cycles: 364726 (5.09%)
Ops/Cyc: 1.58
Stall cycles: 0 (0.00%)

Scheduling

	wsl	pesl	per	wgt
HB_147	8.66	9	0.90	21284
HB_148	97.98	98	0.99	1841
Sched length:	1546			

Operation breakdown

	Dynamic counts		Static counts	
Load:	107811	(19%)	48	(23%)
Store:	69958	(12%)	38	(18%)

iAlu:	266362	(46%)	115	(54%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	42568	(7%)	3	(1%)
Pbr:	46250	(8%)	4	(2%)
Branch:	44369	(8%)	4	(2%)
Total:	577318		212	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_1296	4	1				
L_1300	4	-				
L_1309	4	-				
L_1316	4	-				
L_1326	4	-				
L_1335	4	-				
L_1342	4	-				
L_1353	4	-				
L_1359	4	-				
L_1366	4	-				
S_1303	4	-	G_1313	3	12	1
S_1313	4	-	"			
S_1320	4	-	"			
S_1329	4	-	G_1339	3	12	1
S_1339	4	-	"			
S_1346	4	-	"			
S_1356	4	-	G_1363	3	12	1
S_1363	4	-	"			
S_1370	4	-	"			

	Dynamic count		Static count	
Spill:	123347	(21%)	67	(32%)

LOOP_5 _internal_int_transpo.16

Program:	epic_dec
File:	collapse_pyr.c
Function:	internal_int_transpo
Header block:	HB_16
Loop blocks:	HB_16
Nesting level:	2
Innermost:	yes
Category:	WHILE_LOOP
Invocations:	7201
Iterations:	21752
Iter/Invoc:	3
Operations:	412808 (4.63%)
Cycles:	345804 (4.83%)
Ops/Cyc:	1.19
Stall cycles:	74048 (21.41%)

Scheduling

	wsl	pesl	per	wgt
HB_16	12.33	12	0.67	21748
Sched length:	37			

Operation breakdown

	Dynamic counts		Static counts	
Load:	43504	(11%)	2	(10%)
Store:	21520	(5%)	2	(10%)
iAlu:	195528	(47%)	10	(48%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	65252	(16%)	3	(14%)
Pbr:	43504	(11%)	2	(10%)
Branch:	43500	(11%)	2	(10%)
Total:	412808		21	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_23	4	-				
L_26	4	1				

S_28	4	-
S_30	4	1

LOOP_6 _collapse_pyr.102

Program:	epic_dec
File:	collapse_pyr.c
Function:	collapse_pyr
Header block:	HB_102
Loop blocks:	HB_102
Nesting level:	4
Innermost:	yes
Category:	WHILE_LOOP
Invocations:	1949
Iterations:	2168
Iter/Invoc:	1
Operations:	463910 (5.21%)
Cycles:	224085 (3.13%)
Ops/Cyc:	2.07
Stall cycles:	0 (0.00%)

Scheduling

	wsl	pesl	per	wgt
HB_102	103.87	104	0.89	2168
Sched length:	116			

Operation breakdown

	Dynamic counts		Static counts	
Load:	108400	(23%)	50	(23%)
Store:	86720	(19%)	40	(19%)
iAlu:	255824	(55%)	118	(55%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	4336	(1%)	2	(1%)
Pbr:	4336	(1%)	2	(1%)
Branch:	4294	(1%)	2	(1%)
Total:	463910		214	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_878	4	-				
L_887	4	-				
L_894	4	-				
L_904	4	-				
L_913	4	-				
L_920	4	-				
L_931	4	-				
L_937	4	-				
L_944	4	-				
L_3503	4	1				
S_881	4	-	G_891	3	12	1
S_891	4	-	"			
S_898	4	-	"			
S_907	4	-	G_917	3	12	1
S_917	4	-	"			
S_924	4	-	"			
S_934	4	-	G_941	3	12	1
S_941	4	-	"			
S_948	4	-	"			

	Dynamic count		Static count	
Spill:	153928	(33%)	71	(33%)

LOOP_7 _internal_int_transpo.4

Program:	epic_dec
File:	collapse_pyr.c
Function:	internal_int_transpo
Header block:	HB_4

Loop blocks: HB_4
Nesting level: 2
Innermost: yes
Category: WHILE_LOOP
Invocations: 7201
Iterations: 10764
Iter/Invoc: 1
Operations: 53820 (0.60%)
Cycles: 191592 (2.67%)
Ops/Cyc: 0.28
Stall cycles: 62424 (32.58%)

Scheduling

	wsl	pesl	per	wgt
HB_4	11.67	12	0.67	10764
Sched length:	17			

Operation breakdown

	Dynamic counts		Static counts	
Load:	0	(0%)	0	(0%)
Store:	0	(0%)	0	(0%)
iAlu:	21528	(40%)	2	(40%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	10764	(20%)	1	(20%)
Pbr:	10764	(20%)	1	(20%)
Branch:	10764	(20%)	1	(20%)
Total:	53820		5	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr

LOOP_8 _write_pgm_image.3

Program: epic_dec
File: fileio.c
Function: write_pgm_image
Header block: BB_3
Loop blocks: BB_3
Nesting level: 1
Innermost: yes
Category: MOD_SCHED
Invocations: 1
Iterations: 65536
Iter/Invoc: 65536
Operations: 458753 (5.15%)
Cycles: 131074 (1.83%)
Ops/Cyc: 3.50
Stall cycles: 0 (0.00%)

Scheduling

	RecMII	ResMII	II	ESC
BB_3	2	1	2	1
Sched length:	131074			

Operation breakdown

	Dynamic counts		Static counts	
Load:	65536	(14%)	1	(14%)
Store:	65536	(14%)	1	(14%)
iAlu:	262144	(57%)	4	(57%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	65537	(14%)	1	(14%)
Total:	458753		7	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
L_23	4	1				
S_25	1	1				

LOOP_9 _collapse_pyr.4

Program:	epic_dec
File:	collapse_pyr.c
Function:	collapse_pyr
Header block:	BB_4
Loop blocks:	BB_4
Nesting level:	2
Innermost:	yes
Category:	MOD_SCHED
Invocations:	4
Iterations:	87040
Iter/Invoc:	21760
Operations:	348164 (3.91%)
Cycles:	87628 (1.22%)
Ops/Cyc:	3.97
Stall cycles:	584 (0.67%)

Scheduling

	RecMII	ResMII	II	ESC
BB_4	1	1	1	1
Sched length:	21761			

Operation breakdown

	Dynamic counts		Static counts	
Load:	0	(0%)	0	(0%)
Store:	87040	(25%)	1	(25%)
iAlu:	174080	(50%)	2	(50%)
fAlu:	0	(0%)	0	(0%)
Cmpp:	0	(0%)	0	(0%)
Pbr:	0	(0%)	0	(0%)
Branch:	87044	(25%)	1	(25%)
Total:	348164		4	

Memory operations

	Size	Stride	Group	nOps	gSize	gStr
S_32	4	1				

Bibliography

[ABI+95] K. Asanovic, J. Beck, B. Irissou, B. Kingsbury, N. Morgan, and J. Wawrzynek. The T0 vector microprocessor. In *Hot Chips VII*, pages 187–196, August 1995.

[AKR98] S. Aditya, V. Kathail, and B. R. Rau. Elcor's machine description system: Version 3.0. Technical Report HPL-98-128, Information Technology Center, 1998.

[AMD00] AMD. 3DNow! technology manual. Technical Report 21928G/0, Advanced Micro Devices, Inc, 2000.

[AMD06] AMD. AMD Opteron processor product data sheet, 2006. http://www.amd.com/us-en/Processors/TechnicalResources.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[AS90] E. H. Adelson and E. P. Simoncelli. Subband image coding with three-tap pyramids. In *Proceedings of the Picture Coding Symposium*, pages 3.9.1–3.9.3, 1990.

[BBH+04] D. Boggs, A. Baktha, J. Hawkins, D. Marr, J. Miller, P. Roussel, R. Singhal, B. Toll, and K. S. Venkatraman. The microarchitecture of the Intel Pentium 4 processor on 90nm techonology. *Intel Technology Journal*, 08(01):7–23, February 2004.

[BCM94] D. Bernstein, D. Cohen, and D. E. Maydan. Dynamic memory disambiguation for array references. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 105–111, November 1994.

[BE94] W. Blume and R. Eigenmann. The range test: a dependence test for symbolic, non-linear expressions. In *Proceedings of the 1994 conference on Supercomputing*, pages 528–537, 1994.

[BLO02] C. Basoglu, W. Lee, and J. O'Donnell. The Equator MAP-CA DSP: an end-to-end broadband signal processor VLIW. *IEEE Transactions on Circuits and Systems for Video Technology*, 12(8):646–659, 2002.

[BS00] V. Bongiorno and G. Shorrel. Cray SV1, SV1e, SV1ex – Overview, 2000. http://www.cray.com/products/systems/sv1.

[CBC93] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[CDJ+97] T. M. Conte, P. K. Dubey, M. D. Jennings, R. B. Lee, A. Peleg, S. Rathnam, M. Schlansker, P. Song, and A. Wolfe. Challenges to combining general-purpose and multimedia processors. *IEEE Computer*, 30(12):33–37, December 1997.

[CEL+03] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi. The reconfigurable streaming vector processor (RSVP). In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pages 141–150, Washington, DC, USA, 2003. IEEE Computer Society.

[CEV99] J. Corbal, R. Espasa, and M. Valero. Exploiting a new level of DLP in multimedia applications. In *Proceedings of the 32nd international symposium on Microarchitecture*, pages 72–79, 1999.

[CGH+04] L. N. Chakrapani, J. C. Gyllenhaal, W. W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah. Trimaran: An infrastructure for research in instruction-level parallelism. *17th International Workshop on Languages and Compilers for High Performance Computing. Lecture Notes in Computer Science*, 3602:32–41, 2004.

[CMT94] S. Carr, K. S. McKinley, and C. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the 6th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, 1994.

[CNO+88] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, C-37(8):967–979, August 1988.

[Cor02] J. Corbal. *N-Dimensional Vector Instruction Set Architectures for Multimedia Applications.* PhD thesis, UPC, Departament d'Arquitectura de Computadors, 2002.

[DD97] K. Diefendorff and P. Dubey. How multimedia workloads will change processor design. *IEEE Computer*, 30(9):43–45, Sept 1997.

[Dev99] Analog Devices. Introducing TigerSHARC, 1999. http://www.analog.com/new/ads/html/SHARC2.

[DP02] A. Dasu and S. Panchanathan. A survey of media processing approaches. *IEEE Transactions on Circuits and Systems for Video Technology*, 12(8):633–645, August 2002.

[EAE+02] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: a vector extension to the Alpha architecture. In *Proceedings of the 29th annual International Symposium on Computer Architecture*, pages 281–292, Washington, DC, USA, 2002. IEEE Computer Society.

[EFK+98] K. Ebcioglu, J. Fritts, S. Kosonocky, M. Gschwind, E. Altman, K. Kailas, and T. Bright. An eight-issue tree-VLIW processor for dynamic binary translation. In *International Conference on Computer Design: VLSI in Computers and Processors*, pages 488–495, 1998.

[EGH94] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, 1994.

[EM01] A. H. M. R. El-Mahdy. *A Vector Architecture for Multimedia Java Applications*. PhD thesis, Dept. of Computer Science, University of Manchester, 2001.

[Eme99] J. S. Emer. Simultaneous multithreading: Multiplying Alpha performance, 1999. Microprocessor Forum.

[FBF+00] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture 2000*, pages 203–213, June 2000.

[Fea91] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[FG00] J. Fridman and Z. Greenfield. The TigerSHARC DSP architecture. *IEEE Micro*, 20(1):66–76, 2000.

[Fis81] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30:478–490, July 1981.

[Fly72] M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C–21(9):948–960, 1972.

[Fri00] J. Fritts. *Architecture and compiler design issues in programmable media processors*. PhD thesis, Dept. of Electrical Engineering, Princeton University, 2000.

[FST05] J. E. Fritts, F. W. Steiling, and J. A. Tucek. Mediabench II video: Expediting the next generation of video systems research. *Embedded Processors for Multimedia and Communications II. Proceedings of the SPIE*, 5683:79–93, March 2005.

[Gal95] D. M. Gallagher. *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation*. PhD thesis, Dept. of Electrical and Computer Engineering, University of Illinois, 1995.

[GCM+94] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. *ACM SIGPLAN Notices*, 29(11):183–193, 1994.

[GHF+06] M. Gschwind, H. P. Hofstee, B. Flachs, M. H., Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[GHR96] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau. HMDES version 2.0 specification. Technical Report IMPACT-96-03, University of Illinois,Urbana, IL, 1996.

[GKT91] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 15–29, June 1991.

[GMNR06] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem. Introduction to Intel Core Duo processor architecture. *Intel Technology Journal*, 10(2):89–98, May 2006.

[Gwe99] L. Gwennap. MAJC gives VLIW a new twist. *Microprocessor Report*, 13(12):12–15, September 1999.

[HH02] H. C. Hunter and W. W. Hwu. Code coverage and input variability: effects on architecture and compiler research. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 79–87. ACM Press, 2002.

[HMC+93] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7:229–248, January 1993.

[Hor82] R. M. Hord. *The Illiac IV, the first supercomputer*. Computer Science Press, 1982.

[HP00] M. Hind and A. Pioli. Which pointer analysis should I use? In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[HSS94] A. Huang, G. Slavenburg, and J. Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 200–210, April 1994.

[IBM06a] IBM. IBM BladeCenter QS20 datasheet, 2006. http://www.ibm.com/technology/splash/qs20/pdf/qs20_datasheet.pdf.

[IBM06b] IBM. Press room - 2006-09-06. IBM to build world's first Cell Broadband Engine based supercomputer, 2006. http://www.ibm.com/press/us/en/pressrelease/20210.wss.

[Imp01] Improvsys. Jazz DSP processor datasheet. Technical report, Improvsys, 2001.

[Int99] Intel. Pentium III processor: Developer's manual. Technical report, INTEL, 1999.

[Int04] Intel. Intel Itanium2 processor reference manual for software development and optimization, 2004. http://developer.intel.com/design/-itanium2/manuals/251110.htm.

[Int06] Intel. Dual-Core Intel Itanium 2 processor 9000 series. Product brief, 2006. http://www.intel.com/products/processor/itanium2/-dc_prod_brief.htm.

[Joh91] M. Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Joh05] D. J. C. Johnson. Overview of the HP 9000 rp3410-2, rp3440-4, rp4410-4, and rp4440-8 servers, 2005. http://www.hp.com/products1/servers/HP9000_family_overview.html.

[JVTW01] B. Juurlink, S. Vassiliadis, D. Tcheressiz, and H. A.G. Wijshoff. Implementation and evaluation of the complex streamed instruction set. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 73–82, September 2001.

[KAO05] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, 2005.

[KIea00] A. Kunimatsu, N. Ide, and T. Sato et. al. Vector unit architecture for Emotion synthesis. *IEEE Micro*, 20(2):85–95, March-April 2000.

[Koe99] R. Koenen. Mpeg-4, multimedia for our time. *IEEE Spectrum*, 30(9):26–34, February 1999.

[Kon98] K. Konstantinides. VLIW architecture for media processing. *IEEE Signal Processing Magazine*, 15(2):16–19, 1998.

[Koz99] C. Kozyrakis. A media-enhanced vector architecture for embedded memory systems. Technical Report UCB/CSD-99-1059, EECS Department, University of California, Berkeley, 1999.

[KP98] C. Kozyrakis and D. Patterson. A new direction for computer architecture research. *IEEE Computer*, 31(11):24–32, November 1998.

[KSR00] V. Kathail, M. Schlansker, and B. R. Rau. HPL-PD architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett–Packard Lab., 2000.

[Lan92] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, Dec 1992.

[Lee84] B. G. Lee. A new algorithm to compute the Discrete Cosine Transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-32, 6:1243–1245, December 1984.

[Lee95] R. B. Lee. Accelerating multimedia with enhanced microprocessors. *IEEE Micro*, 15(2):22–32, 1995.

[Lee99] R. Lee. Efficiency of microSIMD architectures and index-mapped data for media processors. In *Proceedings of IS&T/SPIE Symposium on Electric Imaging: Media Processors 99*, pages 34–46, January 1999.

[LPMS97] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 330–335, 1997.

[LR92] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Notices*, 27(7):235–248, June 1992.

[LS96] R. B. Lee and M. D. Smith. Media processing: A new design target. *IEEE Micro*, 16(4):6–9, August 1996.

[LW97] H. Liao and A. Wolfe. Available paralellism in video applications. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 321–329, 1997.

[MB04] C. McNairy and R. Bhatia. Montecito: The next product in the Itanium Processor Family. In *Conference Record of 16th Hot Chips Symposium*, 2004.

[MH99] S. Moon and M. W. Hall. Evaluation of predicated array data-flow analysis for automatic parallelization. In *Proceedings of the ACM Symposium on Principles Practice of Parallel Programming*, pages 84–95, 1999.

[MHL91] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 1–14, June 1991.

[MLC+92] 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 *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, Dec. 1992.

[MNW+02] R. Mahajan, R. Nair, V. Wakharkar, J. Swan, J. Tang, and G. Vandentop. Emerging directions for packaging technologies. *Intel Technology Journal*, 6(2):61–76, May 2002.

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8):114–117, 1965.

[Nic89] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.

[NJ99] H. Nguyen and L. K. John. Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology. In *International Conference on Supercomputing*, pages 11–20, 1999.

[PHP98] Y. Paek, J. Hoeflinger, and D. Padua. Simplification of array access patterns for compiler optimizations. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 60–71, 1998.

[PP99] N. Pitsianis and G. Pechanek. High-performance FFT implementation on the BOPS ManArray parallel DSP. In *Proceedings of the International Symposium on Optical Science, Engineering, and Instrumentation*, 1999.

[PS91] J. C. Park and M. S. Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett–Packard Lab., May 1991.

[Pur98] S. Purcell. The impact of Mpact 2. *IEEE Signal Processing Magazine*, 15(2):102–107, 1998.

[PW96] A. Peleg and U. Weiser. MMX technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[PW98] W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM Transactions on Programming Languages and Systems*, 20(3):635–678, May 1998.

[QCEV99] F. Quintana, J. Corbal, R. Espasa, and M. Valero. Adding a vector unit on a superscalar processor. In *Proceedings of the International Conference on Supercomputing*, pages 1–10, June 1999.

[QEV98] F. Quintana, R. Espasa, and M. Valero. An ISA comparison between superscalar and vector processors. *Selected Papers and Invited Talks from the Third International Conference on Vector and Parallel Processing. Lecture Notes In Computer Science*, 1573:548–560, 1998.

[RAJ99] P. Ranganathan, S. V. Adve, and N. P. Jouppi. Performance of image and video processing with general-purpose processors and media ISA extensions. In *Proceedings of the International Symposium on Computer Architecture*, pages 124–135, 1999.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, Sept 1994.

[Rau95] B. R. Rau. Iterative modulo scheduling. Technical Report HPL-94-115, Hewlett–Packard Lab., 1995.

[RDK+98] S. Rixner, W. J. Dally, U. J. Kapasi, B. K., A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, pages 3–13, November 1998.

[RDK+00] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens. Register organization for media processing. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, January 2000.

[Red73] S. F. Reddaway. DAP-a distributed array processor. In *Proceedings of the 1st annual symposium on Computer architecture*, pages 61–65. ACM Press, 1973.

[RF93] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: history, overview, and perspective. *The Journal of Supercomputing*, 7(1-2):9–50, 1993.

[RP94] L. Rauchwerger and D. Padua. The PRIVATIZING DOALL test: A run-time technique for DOALL loop identification and array privatization. In *Proceedings of the ACM International Conference on Supercomputing*, 1994.

[RS96] S. Rathnam and G. Slavenburg. An architectural overview of the programmable multimedia processor, TM-1. In *Proceedings of the 41st IEEE International Computer Conference*, pages 319–326, Washington, DC, USA, 1996. IEEE Computer Society.

[Rus78] R. Russel. The Cray-1 computer system. *Comunications of the ACM*, 21(1):63–72, January 1978.

[RYYT89] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer. *IEEE Computer*, 22(1):12–35, January 1989.

[SA00] H. Sharangpani and K. Aurora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, September 2000.

[SAS+05] F. Sánchez, M. Alvarez, E. Salamí, A. Ramírez, and M. Valero. On the scalability of 1 and 2-dimensional SIMD extensions for multimedia applications. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 167–176, March 2005.

[SC97] K. Skadron and D. W. Clark. Design issues and tradeoffs for write buffers. *High Performance Computer Architecture*, 00:144–155, 1997.

[SCAV02] E. Salamí, J. Corbal, C. Alvarez, and M. Valero. Cost effective memory disambiguation for multimedia codes. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 117–126, October 2002.

[SCEV99] E. Salamí, J. Corbal, R. Espasa, and M. Valero. An evaluation of different DLP alternatives for the embedded media domain. In *Proceedings of the 1st Workshop on Media Processors and DSPs*, pages 100–109, November 1999.

[Sem99] Philips Semiconductors. TriMedia TM-1300, 1999. http://www-us3.semiconductors.com/trimedia.

[Ses98] N. Seshan. High VelociTI processing. *IEEE Signal Processing Magazine*, 15(2):86–101, 1998.

[SH97] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[SIG97] SIG. Mips extension for digital media with 3d. Technical report, MIPS Technologies, Inc, 1997.

[Sik95] T. Sikora. *MPEG Digital Video Coding Standards*. McGraw W-Hill Book Company, Berlin, 1995.

[SJ01] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Technical Report WRL-2001-2, HP Western Research Labs, 2001.

[SKT+05] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.

[Smo02] M. Smotherman. Understanding EPIC architectures and implementations. In *40th Annual ACM Southeast Conference*, pages 71–78, April 2002.

[SR00] M. S. Schlansker and B. Raw. EPIC: Explicitly parallel instruction computing. In *IEEE Computer*, pages 37–45, February 2000.

[SS01] N. T. Slingerland and A. J. Smith. Cache performance for multimedia applications. In *Proceedings of the 15th International Conference on Supercomputing*, pages 204–217, 2001.

[SS02] N. T. Slingerland and A. J. Smith. Design and characterization of the Berkeley multimedia workload. *Multimedia Systems*, 8(4):315–327, 2002.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 32–41, 1996.

[Sud00] S. Sudharsanan. MAJC-5200: A high performance microprocessor for multimedia computing. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 163–170, London, UK, 2000. Springer-Verlag.

[SV05a] E. Salamí and M. Valero. Dynamic Memory Interval Test vs. Interprocedural Pointer Analysis in multimedia applications. *ACM Transactions on Architecture and Code Optimization*, 2(2):199–219, June 2005.

[SV05b] E. Salamí and M. Valero. A Vector-uSIMD-VLIW architecture for multimedia applications. In *Proceedings of the 2005 International Conference on Parallel Processing*, pages 69–77. IEEE Computer Society, June 2005.

[Tal01] D. Talla. *Architectural Techniques to Accelerate Multimedia Applications on General-Purpose Processors*. Ph.D. thesis, The University of Texas at Austin, 2001.

[TCC+00] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse. The MAJC architecture: A synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12–25, 2000.

[TDJ+02] J. M. Tendler, J. S. Dodson, J. S. Fields Jr., H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[TEL95] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd annual International Symposium on Computer Architecture*, pages 392–403, New York, NY, USA, 1995. ACM Press.

[TI99] TI. TMS320C62XX family, 1999. http://www.ti.com/sc/docs/-products/dsp/tms320c6201.html.

[TJ01] D. Talla and L. K. John. Cost-effective hardware acceleration of multimedia applications. In *Proceedings of the International Conference on Computer Design*, page 415, Washington, DC, USA, 2001. IEEE Computer Society.

[TONL96] M. Tremblay, J. M. O'Connor, V. Narayanan, and H. Liang. VIS speeds new media processing. *IEEE Micro*, 16(4):51–59, August 1996.

[Tri01] A. Triggs. Lecture 11: Global system for mobile communications (GSM). Wireless Cellular & Personal Communications Ericsson Inc., Southern Methodist University, 2001.

[vdSD01] A. van der Steen and J. Dongarra. The NEC SX-5, 2001. http://www.top500.org/ORSC/2001.

[vdWVD+05] J.-W. van de Waerdt, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J.-P. van Itegem, D. Amirtharaj, K. Kalra, P. Rodriguez, and H. van Antwerpen. The TM3270 media-processor. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 331–342, Washington, DC, USA, 2005. IEEE Computer Society.

[vESV+99] J. T. J. van Eijndhoven, F. W. Sijstermans, K. A. Vissers, E.-J. D. Pol, M. J. A. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel, , and H. P. E. Vranken. TriMedia CPU64 architecture. In *Proceedings of the 1999 IEEE International Conference on Computer Design*, pages 586–592, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[VLPA95] M. Valero, T. Lang, M. Peiron, and E. Ayguade. Conflict-free access for streams in multimodule memories. *IEEE Transactions on Computers*, 44(5):634–646, 1995.

[WAK+96] J. Wawrzynek, K. Asanovic, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan. Spert-ii: A vector microprocessor system. *IEEE Computer*, 29(3):79–86, March 1996.

[Wal91a] D. W. Wall. Limits of instruction-level parallelism. *SIGPLAN Notices*, 26(4):176–188, 1991.

[Wal91b] G. K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, April 1991.

[WL91] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 conference on Programming Language Design and Implementation*, pages 30–44, 1991.

[WL95] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[WM95] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[Yu96] A. Yu. The future of microprocessors. *IEEE Micro*, 16(6):46–53, 1996.