A Vector Architecture for Multimedia Java Applications

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy in the
Faculty of Science & Engineering

2001

Ahmed H.M.R. El-Mahdy

Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

Multimedia applications written in Java possess different processing requirements from conventional applications for which current microprocessors have been optimised. Multimedia applications are computationally intensive, stretching the capabilities of current microprocessors. Initial multimedia extensions relied on short vector architecture extensions to the instruction sets. While these exploit the wide datapaths in current microprocessors, they introduce misalignment overheads and restrict the addressing mode to sequential access. Java, being a platform-independent language, relies on a stack-based intermediate language, bytecode, needing a translation layer to be executed. The translation has to be fast and this limits the optimisations that can be done in software and opens the question of whether supporting Java bytecode natively is a better idea for high performance processing.

The thesis first demonstrates that a register-based instruction set is more efficient for executing Java bytecode than a stack-based instruction set. It then develops a novel vector architecture that exploits the two dimensional data access patterns of multimedia applications. This results in an instruction set combining the benefits of subword and traditional vector processing. The thesis also develops a novel cache prefetching mechanism to capture two dimensional locality of reference. A prototype translation system is also developed to support the vector instruction set from high-level Java programs.

Throughout the design, high-level analysis tools, such as trace-driven simulation, software instrumentation of Sun's JDK, and simple analytical performance models, are developed to aid the design decisions and verify existing results in the literature. The analytical models are used further to derive an optimal configuration. A detailed simulation of the system is developed and used to analyse the performance of MPEG-2 video encode and decode applications, and two kernels taken from 3D and voice recognition applications. The simulation integrates the vector architecture into a single-chip multithreaded multiprocessor architecture and studies the interactions. The architecture has shown an advantage in terms of reducing the instruction count, cache misses, and overall execution time for current and future technologies.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Dedication

To my parents and grandma

The Author

Ahmed El-Mahdy received the degree of B.Sc. with distinction from the department of computer science and automatic control at Alexandria university in 1995 where he received an award from the faculty of engineering and another from the department for outstanding undergraduate achievements. He was awarded a teaching assistantship from the same department and received the degree of M.Sc. in computer science in 1998. He then joined the computer architecture group at the department of computer science at the university of Manchester in 1998, and was awarded a teaching assistantship. He started reading for the degree of Ph.D. in September 1998, in the JAMAICA project. The project is investigating a single-chip multiprocessor architecture with support for light-weight threads, targeting Java applications. His research interests include architectural support for languages and applications, dynamic compilation techniques, analytical performance modelling, and computer graphics. He is a student member of the IEEE Computer Society and the ACM.

Acknowledgements

Thanks to Prof. Ian Watson for his encouragement and for his constructive feedback and to Greg Wright for many fruitful discussions. Both of them developed parts of the run-time system and Greg developed a JAMAICA simulator which made a more detailed system evaluation possible. Thanks to you both for the inspiring group meetings and for helping me to become familiar with the project rapidly.

Special thanks for my parents, although they were abroad, they managed to keep me going by a constant flow of emails, sharing with me every moment.

Thanks to Mashhuda and Nick for reminding me of the festivals and celebrations. Thanks to Ian Rogers for taking me out for lunches in exotic places, and for his insightful discussions about dynamic compilation.

Thanks to the Amulet group for trying hard to get me out of work to play badminton. Special thanks to Aris for helping to proof read early drafts of the thesis, and to Andrew for his help with Unix.

Thanks to all my friends at Hulme Hall for giving me the sense of a family. Special thanks to David Butler for getting me involved in social activities, Denis for tennis and frisbee breaks, George and Pedro for sharing their spirit of discovery, and to Yan for brain-cracking board games. Thanks also to Hannah, Teresa, and Gareth for improving my knowledge of English culture.

Thanks to Alaa Alameldeen, Mats, and In-Sun for constantly keeping in touch.

Thanks to the department for the teaching assistantship, and thanks for the ORS award. Without these, I would not have been able to travel to the UK in the first place.

Chapter 1: Introduction

There is always a demand for improving the performance of microprocessors. Since their introduction in 1971, the performance of microprocessors has improved at an exponential rate. Technological improvements have allowed the number of transistors per chip to be doubled every 18 months and performance to double every two years. By the mid-1980s, the Reduced Instruction Set Computer (RISC) concept was introduced and architectural innovations thereafter contributed to quadrupling the performance of microprocessors every three years [31].

During the next 10 years, it is expected that the integration technology will continue developing at the same rate, achieving a billion-transistor microprocessor by the year 2010 [67]. Although transistor delays are expected to scale with the shrinking transistor sizes, wire delays are likely to increase and dominate the cycle time [21, 50]. Architectural innovations are thus emphasised for sustaining the performance improvement rate for future microprocessors.

1.1 Architectural Challenges

A major challenge for architecture is to support the relatively new multimedia workloads, as they are likely to dominate the application domain [38]. Multimedia workloads have significantly different characteristics from other existing applications. They are computationally intensive. For instance, an MPEG-2 [33] video encode operation requires in the order of 10 giga operations per second (GOPS), stretching the capabilities of current microprocessors.

Currently, there is a convergence between digital signal processors (DSPs) and general-purpose processors. Both markets are borrowing architectural ideas from the other. DSPs were originally optimised towards processing relatively less complex media tasks such as audio processing. The diversity of multimedia requirements led the DSPs to evolve to media processors, where ideas such as cache, and Very Long Instruction Word (VLIW) architecture are borrowed from the general-purpose processor market.

On the general-purpose processor side, traditionally the architecture is optimised for commercial and scientific workloads. A design shift towards multimedia is starting to take place. Most major microprocessor manufacturers are incorporating multimedia extensions into their architectures. These include Intel x86's MMX [64] and Stream SIMD [76], PowerPC's AltiVec [14], UltraSPARC's VIS [77], PA-RISC's MAX-2 [46], and AMD's 3DNow! [59]. However, this shift is recent and it is not yet clear what are the trade-offs and what an optimal design would be [7, 13].

Another challenge is supporting Java. Java is a new language that is getting more attention as a mainstream programming language. Java relies on having a virtual machine [48] to be executed. The virtual machine is a compilation system that translates, on-line, the Java intermediate format (bytecode) into native machine code. The fact that the compilation is done at the same time as the program is executing, restricts the traditional complex optimisations that assume off-line compilation. Register allocation for instance cannot use extensive register colouring. Although the technique is suitable for off-line static compilation [54], it is not suitable for such fast dynamic translation systems. The architecture might be able to provide support for such a machine.

1.2 Problem Statement

In this thesis, we tackle the problem of designing a general-purpose high performance microprocessor architecture exploiting the new characteristics of multimedia workloads while, at the same time, not penalising the performance of conventional workloads. We restrict the design space to a single-chip multiprocessor as it is a promising approach coping with the future technological constraints. This approach is pursued by our group project Java Machine And Integrated Circuit Architecture (JAMAICA). Moreover, our main assumption is that applications are written in Java. We chose Java as it has many characteristics that are likely to be emphasised in future systems, such as platform independence, object-orientedness, and threading constructs. The latter feature fits nicely with the context of multiprocessing.

So there are two close aims of the thesis:

- Supporting multimedia: Do we need special functional units? How will this affect the design of the instruction set? Will cache work under the multimedia workload?

- Supporting Java: What should be the register file design? How can compilation for multimedia be supported?

1.3 The 2d-vector Approach

We cannot explore all possible design options for supporting multimedia and Java. The approach we have followed is decomposing the design problem into manageable subproblems that will be described below. The main issues within a subproblem are identified, major existing solutions are evaluated and a new solution is developed. Our analysis includes analytical performance modelling as it gives us a wider exploration into the design space. We have also developed trace-driven simulation and software instrumentation for modelling more complex behaviour. We have then integrated these components together and formed the 2d-vector architecture. An analytical model is developed for the 2d-vector architecture, and together with detailed simulation, the architecture is evaluated.

1.3.1 Java Stack Overhead

There are basically two obvious ways to implement a virtual machine; either have it as a software layer, or implement it in hardware. Since we are concerned with architecture, both options are valid for us. Supporting the stack-based execution model of bytecode in hardware seems the most attractive as mapping the stack operands into registers would not be required.

A register-based instruction set, however, has the potential to remove the extra pops and pushes that are required to access stack-operands. We refer to those extra operations as the ‘stack-overhead’. A trivial stack mapping algorithm would not remove these extra instructions and it will effectively generate an instruction for every push and pop. A more clever, and probably more time consuming algorithm, would eliminate many of these

copy instructions. A hardware solution may adopt simple optimisations that fold these instructions on the fly. These optimisation were implemented in Sun's picoJava [73].

So the fundamental design question is whether a simple software register-mapping algorithm will suffice to produce an efficient mapping and remove the stack-overhead. Or whether a simple hardware optimisation will yield better results given that no software overhead analysis is needed. We have explored, in this thesis, this issue and came to the conclusion that a simple register mapping technique can yield significant reduction in the stack overhead (80%). Thus we opted for a register-based instruction set. We have also observed the high percentage of method calls in Java programs and concluded that a register-window approach is appropriate.

1.3.2 Data-Parallelism

Multimedia applications are inherently data parallel; a typical multimedia kernel, as demonstrated in the thesis, would have a tight loop with hardly any loop carried dependencies. Loop iterations can be executed in parallel, moreover the loop control overhead can be eliminated. This suggests a vector instruction set. Indeed all the current multimedia extensions have incorporated vectors in the form of 'subword parallelism' [46]; subwords in a wide register form a small vector, and an operation is performed on each subword in parallel.

Subword parallelism has the cost benefit of utilising existing wide registers and data paths in microprocessors for the small data types in multimedia applications. However, this implementation restricts matrix access (especially when a submatrix is accessed [37]) and misalignment is introduced. A traditional vector architecture with a submatrix addressing mode would not suffer from these overheads, moreover the loop control overhead will be removed.

We developed a simple performance model to take into account the major ways to implement vector execution and applied it to many multimedia kernels. We observed that combining a traditional vector architecture and subword parallelism, with a submatrix addressing mode, yields better performance. The performance is better than a subword parallelism architecture with double the machine word size. That architecture reduces the

limitation introduced by subword parallelism. It would be an extension to the underlying scalar architecture.

1.3.3 Data Locality

Conventional caches have been optimised for temporal and spatial data locality. Multimedia applications tend to have a streaming data access pattern and thus the spatial data locality is more emphasised than the temporal data locality. We have observed that the data access patterns for MPEG-2 encode and decode applications tends to have a two dimensional spatial locality; once an address is accessed in a frame, it is likely that addresses in the same row or nearby rows will be accessed. Conventional caches would only prefetch data within a row (one dimensional).

We have developed a simple cache prefetching (2D cache) optimisation taking that 2D spatial locality into consideration. Using trace-driven cache simulation, the 2D cache eliminated a large percentage of the misses (75%). The behaviour is shown to follow a simple analytical model. The model, together with the instruction model, is combined and used to give an approximation of the overall performance of the 2d-vector architecture. The 2D cache yields a performance advantage over a large range of different memory latencies, making it attractive for the future as memory is getting relatively slower compared to processor speed.

We did not consider software prefetching which is likely to give an advantage. However, software prefetching scheduling is a complex task. Besides many important multimedia kernels have unpredictable memory access patterns making prefetch scheduling more difficult.

1.3.4 Compiler Support

The register windows, and the decision to use a register file instead of a stack, is a particular optimisation for Java scalar applications. The vector instruction set and 2D cache are particularly designed for multimedia applications. In order to evaluate the benefits of that architecture, we developed a media-enabled Java translator. The translator supports both scalar and multimedia features. It implements the Cacao register allocation

algorithm [39] and also enables us to observe the quality of the code and develop optimisations that are tailored to our register-based instruction set. The Java translator supports multimedia through a multimedia class library. The library contains methods for every instruction type. The translator substitutes vector instructions instead of method calls under specific conditions (such as leaf methods).

1.4 Contributions

The thesis makes the following contributions:

- Defying the intuition that a stack-based instruction set is more suited to the execution of bytecode. We have demonstrated that a register-based instruction set, relying on a simple register allocation, is better.
- Developing a Java translator that incorporates a set of novel optimisations for register-window allocation and support for multimedia.
- Designing a novel two dimensional vector instruction set for multimedia.
- Identifying the two dimensional data locality in multimedia video applications and designing a novel cache architecture.
- Developing analytical performance models that help to explore the parameter space of a set of large multimedia designs.

1.5 Thesis Organisation

1.5.1 Outline

The thesis is divided into three parts, reflecting Java design consideration, multimedia support, and the proposed system design and evaluation.

The first part of the thesis is concerned with choosing a good register file design for Java bytecode. Chapter 2 studies bytecode execution models and identifies two major overheads; the stack, and method calling overheads. The chapter examines a selection of programs from the SPECjvm98 [70] benchmark suite and analyses the various ways in which bytecode can be executed and quantifies the resulting overheads which occur. That analysis suggests that a register-window based design would provide more benefit. The

chapter concludes by assessing the feasibility of that design in terms of the required number of registers and register-windows.

Having decided on the register file design, Chapter 3 examines a fast register allocation algorithm for that design. To facilitate such a study, a Java translator is developed and used to analyse and tune the register allocation and a set of optimisations is developed.

The second part of the thesis is concerned with the instruction set and cache design for multimedia. This part starts at Chapter 4, where the characteristics of multimedia applications are presented. Being important multimedia applications, MPEG-2 video encode and decode are described and their characteristics are verified. Chapter 5 surveys existing multimedia architectures, exploring architectural ideas for exploiting multimedia workload characteristics.

Chapter 6 models, analytically, the various ways to implement multimedia specific instruction sets in the context of general-purpose processors. A novel instruction set is developed, modelled, and its advantages over existing instruction sets are presented.

Chapter 7 examines the design space for cache design for multimedia. A two dimensional access pattern is identified and a simple cache prefetching technique is developed. The behaviour of the cache is modelled and its efficiency is assessed.

The third part of the thesis is concerned with the system architecture and performance evaluation. It starts at Chapter 8, where the underlying single-chip multiprocessor is described. The proposed architecture is described together with the compilation support for multimedia. Chapter 9 models, analytically, the processor architecture. The model is used to explore the parameter space. An optimal set of parameters is identified. The performance study is complemented by a simulation study which is presented in Chapter 10. Performance is analysed for both uni- and multiprocessors, and the architecture is justified.

Finally, Chapter 11 concludes the thesis by giving a summary of what was achieved and discusses directions for future work.

1.5.2 Reading Flow Among the Chapters

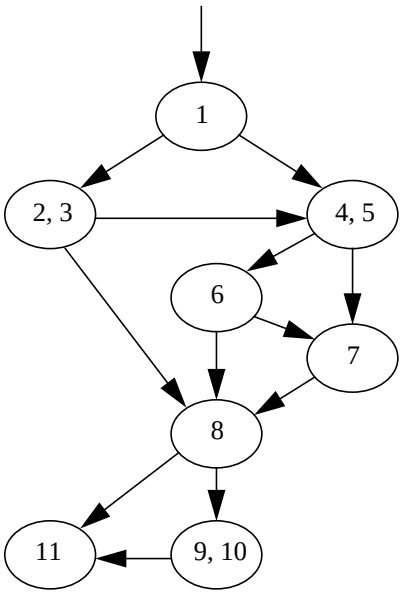

Figure 1.1: Chapter reading flow

There are many reading paths through the thesis depending on the reader's interest. Figure 1.1 shows the reading flow graph between the chapters which we use to provide reading suggestions as follows:

- For Java compilation interest, one may read Chapter 1, then Chapters 2 and 3 as they cover Java execution models and register allocation. Chapter 8 has a section about multimedia compilation support (Section 8.3). Then finally one should read Chapter 11, the conclusion.
- For multimedia instruction set design, one may read Chapter 1, then Chapters 4 and 5 where multimedia kernels and architectures are examined. Then one may read Chapter 6 which focuses on instruction set design, then Chapter 8 and particularly the vector pipeline section (Section 8.2.1). The rest of the chapters should be read in sequence, as they evaluate the interaction of instruction set with other architectural aspects.
- For multimedia cache design, one may read Chapter 1, then Chapter 4 and particularly focus on the memory interface section (Section 5.6) in Chapter 5. Then one may read Chapter 7, where the two dimensional locality is analysed. The rest of the chap-

ters should be read in sequence, as they evaluate the interaction of the cache with other architectural aspects.

Part I: Java Design Aspects

Chapter 2: Register Design for Java

It is tempting to consider a stack-based register organisation for executing Java bytecode. Indeed, Sun's picoJava [51] and Patriot's PSC1000 [62] have used this approach. However, these are embedded processors that emphasise a low-cost memory-footprint design. We believe, for high performance processors, a register-based architecture will outperform a stack-based one. This chapter analyses the trade-offs between register- and stack-based architectures. This is done by implementing a variety of optimisations for these architectures and using them to analyse bytecode execution streams. The results encouraged the design of a register-based architecture using register windows, and have been published in [83].

2.1 The Java Virtual Machine

Java is a platform-independent language. This is achieved by compiling the Java source files into an intermediate instruction set encoded as *bytecodes*. The compiled code is stored into a *class* file. The bytecodes execute over the Java virtual machine [48] (JavaVM). For a platform to execute the bytecodes, it has to implement the JavaVM.

The JavaVM defines many structures for executing the bytecodes. On a method call, a *stack frame* is created where data and temporary results are stored. A stack frame contains an array of variables known as *local variables*. Load and store bytecodes push and pop the values of these variables, respectively, onto the *operand stack*. Parameters are copied from the operand stack to the new stack frame created on a method call. The method return value is copied from the callee operand stack to the caller operand stack.

Associated with every class file is a symbol table called the *constant pool*. It contains various types of constants, such as numeric literals and other references used to resolve methods and classes at run-time. There are bytecodes that load constants onto the operand stack. Also there are other complex bytecodes for creating objects, checking types, and throwing exceptions that use the constant pool.

The intermediate instruction set is a stack-based. This gives flexibility for the underlying JavaVM as no register information is assumed and the JavaVM can thus perform platform specific optimisations. Also, a stack-based instruction set has high code-density, this has the desirable effect of decreasing the network bandwidth required to transmit class files.

2.2 Major Issues

2.2.1 Stack Overhead

A major problem in a stack-based instruction set is that extra instructions are required to push and pop operands onto the operand stack, instead of being specified in one instruction (3-address instruction). Moreover, value communication between the instructions is limited to the operands at the top of the stack. This introduces stack manipulation instructions such as ‘dup’ and ‘swap’.

```
Example 1
a = b - c * d;

              3-address
Bytecodes:    inst. set:
---------     ----------
iload b       mul c, d, tmp
iload c       sub b, tmp, a
iload d
imul
isub
istore a
```

```
Example 2
b = a + b; //line 1
c = a * b; //line 2

                 3-address
Bytecodes:       inst. set:
---------        ----------
iload a          add a, b, b
dup              mul a, b, c
iload b
iadd
dup
istore b
imul
istore c
```

Figure 2.1: Examples of the stack overhead

Figure 2.1 gives an example of the stack overhead. Example 1 shows the extra bytecodes required for popping and pushing operands onto the stack. Example 2 shows the case of limited value communication. The variable ‘b’ is modified at line 1 and its new value has to be communicated to line 2. In the corresponding bytecodes, a ‘dup’ (the second one) is

used to push a copy of the modified variable onto the stack. Moreover, another one is required to overcome the need to reload the variable ‘a’.

A 3-address register-based instruction set does not have that problem as operands are directly addressable from the instructions. However, a compilation step is required to map the operand stack into registers, taking into consideration dependencies that might arise. There is obviously a trade-off between time to carry out such analysis and the success of removing the stack overhead.

2.2.2 Method Calling Overhead

Another major issue in register design, is the fact that Java is an object-oriented language and is likely to have a large number of method calls [6]. A stack-based register organisation has an advantage over a register-based organisation. Local variables can be allocated on the stack and thus a method call does not require copying the locals from the caller operand stack and only one copy, in the opposite direction, will be required for the return value. Moreover, assuming the stack is large enough, no caller data needs to be dumped into memory.

A register file would not be as efficient as a stack register. However, registers can be organised into register windows, where the hardware allocates a new register window on method call from a register windows stack, saving caller register values. The disadvantage will mainly be the extra overhead for maintaining register windows in terms of hardware complexity and the cost of spilling/filling register windows.

2.2.3 Quantifying the Overheads

In order to assess the stack and method calling overheads, we have analysed the bytecode mix of a selection of the Standard Performance Evaluation Corporation’s SPECjvm98 [70] benchmark suite.

The SPECjvm98 suite consists of seven programs. We consider single threaded programs to remove any multithreading overheads. Therefore, we did not consider the _227_mtrt benchmark, as it is a multithreaded program and we are concerned with single thread characteristics. The nature of the benchmarks is described briefly in Table 2.1.

Program	Description
_201_compress	Lempel-Ziv compression
_202_jess	Java Expert Shell System based on NASA's CLIPS expert shell system
_209_db	Performs multiple database functions on a memory resident database
_213_javac	Java compiler from the JDK 1.0.2
_222_mpegaudio	Decompresses ISO MPEG Layer-3 audio files
_228_jack	A Java version of yacc parser generator

Table 2.1: Selected SPECjvm98 programs

Bytecode class	Description
const	Constant loads including constant pool constants
local	Local variable loads and stores
array	Array loads and stores
stack	Stack operations (pop, dup, swap,..., etc.)
ALU	ALU operations
branch	Conditional and unconditional branches including returns
field	Field loads and stores
invoke	Method calls (both virtual and static)
ow	Otherwise (including object creation, synchronisation, and type checking instructions)

Table 2.2: Bytecode classes

In order to obtain the bytecode mix, we have instrumented Sun's JDK JavaVM (version 1.1.7) to count the number of executed bytecodes classified as shown in Table 2.2. The dynamic bytecode mix is shown in Figure 2.2. It is clear from the figure that 'local' has the highest execution frequency. It ranges from 30% to 50%, the average being 40%. Ideally, 'const', 'local', and 'stack' classes can be eliminated if a register-based instruction set is used. These add up to around 50% for all programs in the benchmark suite. O'Connor and Tremblay [60] reported similar results (50.4%).

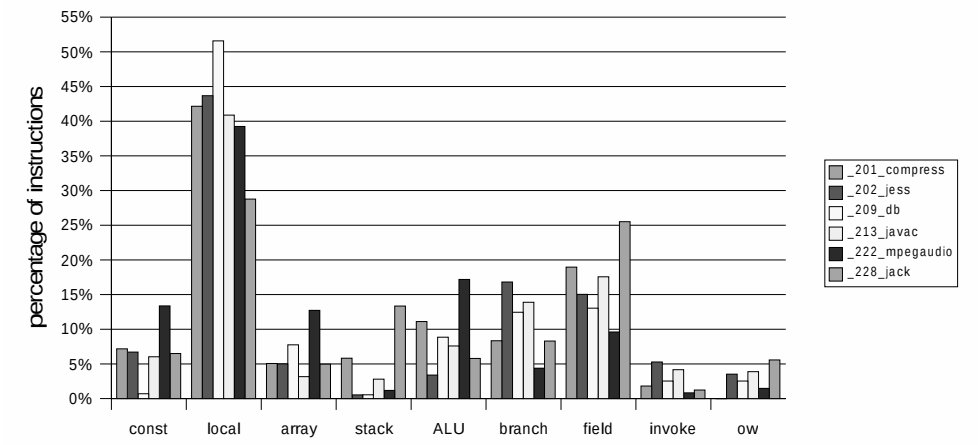

Figure 2.2: Dynamic bytecode mix

It is clear that the stack overhead is a significant problem. This does not necessarily mean that a register-based instruction set is better, observing that method invocation is around 3% on average (and another 3% for method return). The penalty for a method call includes extra instructions for saving and restoring registers. So, method call overhead is also of concern.

2.3 Existing Solutions

Java is a recent language, there is no rigorous coverage for it in the literature. In that sense, we review current Java implementation techniques and non-Java specific optimisations concerning method calling overheads.

There is a wide spectrum for implementing the JavaVM. There are basically two distinct approaches: hardware and software. The hardware approach relies on executing the bytecodes at the architectural level. The software approach adds a layer that either interprets the bytecodes, or compiles them into native host processor machine code.

2.3.1 Hardware-Based Approaches

Patriot's PSC1000 [62] and Sun's picoJava [51], and more recently ARM's Jazelle [49] are Java processors that directly execute bytecodes. PSC1000 and Jazelle do not tackle the stack overhead problem while picoJava does. We thus focus on the picoJava processor.

There are two processors in the picoJava family, picoJava-I [73] and picoJava-II [74]. We use the term ‘picoJava’ to refer to the common characteristics of these processors, and explicitly identify them when referring to an individual characteristic.

```
Bytecodes                    Bytecodes
with out folding             with folding
----------------             ----------------
iload a                      add a, b, c
iload b
iadd
istore c
```

Figure 2.3: Bytecode folding

The picoJava processor implements the operand stack as a register file. The aim is to remove the stack overhead by allowing execution units to have random access to the operands. During execution, consecutive instructions are checked for the possibility of folding as illustrated in Figure 2.3; the add instruction directly accesses the local variables from the register file, removing the loads and stores.

PicoJava-II considers up to four consecutive instructions for folding (picoJava-I considers up to two consecutive instructions only), in which:

- Local loads followed by an ALU instruction and/or
- An ALU instruction followed by a local store.

Instruction folding seems to be a simple technique that assumes a particular pattern of instructions. Applying the following technique to our example in Section 2.2.1, we reduce the number of instructions by 50% and 25% for examples 1 and 2 respectively as shown in Figure 2.4. Minor reductions achieved by carrying out more ambitious folding techniques have been reported [80].

```
Example 1:
a = b - c * d;
                                        3-address
Bytecodes:      Folded bytecodes:       inst. set:
----------      -----------------       --------------------
iload b         iload b                 mul c, d, tmp
iload c                                 sub b, tmp, a
iload d                                 --------------------
                mul c, d, [tof]         ;top of stack register
                                        ;(tof) is preincremented
                sub [tof-1], [tof], a   ;subtraction is carried
imul                                    ;on the top two stack
isub                                    ;operands
istore a

Example 2:
b = a + b;
c = a * b;
                                        3-address
Bytecodes:      Folded bytecodes:       inst. set:
----------      -----------------       --------------------
iload a         iload a                 add a, b, b
dup             dup                     mul a, b, c
iload b         add [tof], c, [tof]
iadd
dup             dup
istore b        istore b
imul            mul [tof-1], [tof], c
istore c
```

Figure 2.4: Examples of the effectiveness of bytecode folding

2.3.2 Software-Based Approaches

There are three main approaches to execute bytecodes at the software-level:

- Interpretation: Each bytecode is translated each time it is executed.
- Dynamic compilation: Bytecodes are compiled at run-time.
- Translation: The entire program is compiled once.

The interpretation approach is the slowest approach and introduces far more overhead than the stack overhead. The dynamic compilation approach overcomes the interpretation overhead by compiling at run-time. Dynamic compilation ranges from compiling a method (or a class) when it is first called (Just-In-Time compilation) to compiling at the basic block level. Dynamic compilation, however, has to be fast and it cannot afford complex run-time optimisations. Full translation has the highest level of optimisation but at the cost of the translation time and a loss of flexibility (dynamic loading of classes is still required).

The dynamic compilation approach is getting more attention now as it combines the benefits from interpretation and translation. From now on we focus on that approach especially Just-In-Time (JIT) compilation as it is described well in the literature. Compilation at the block level has recently been used in Sun's Hotspot compiler [27] and the Dynamite system [66] but not much information is available in the literature about their implementation details.

Commercial JIT compilers have appeared since the introduction of Java. Microsoft, Netscape, Symantec, Sun, Silicon Graphics, and DEC have introduced JIT compilers. In the public domain, Kaffe is available as freeware. Nevertheless, there are few details given for any of these implementations.

Ebcioglu et al. [15] have described the JIT compiler they developed for their Java VLIW processor. Bytecodes are mapped into RISC instructions. However, not many details are given for the register mapping algorithm. Their compiler is similar to the Caffeine translator described elsewhere [32].

Adl-Tabatabai at al. [1] have described their JIT compiler developed for Intel x86 processors. The register mapping is done by introducing a 'mimic' stack that dynamically simulates instruction execution. Many copies due to local loads are removed, however, copies due to local stores are still generated.

Krall [40] introduced the Cacao JIT compiler. It has a fast register mapping algorithm based on a static stack. This is five times as fast as other commercial JIT compilers. While the dynamic stack moves information from an earlier instruction to a later one, a static

stack has the potential to move information in both directions. Accordingly, we have chosen the Cacao register mapping solution in our study. The register allocation details will be given in Chapter 3.

A more recent JIT compiler, LaTTe [87] was introduced at the time we were developing this work. It relies on a more complex register allocation which gives better results than Cacao. However, we are not interested in achieving the best register allocation, rather deciding on whether we should provide support at the hardware-level for Java or rely on the software. So this will not affect our analysis in this aspect.

2.4 Proposed Solution

We believe that using JIT or dynamic compilation techniques on a conventional register-based instruction set would outperform a stack-based instruction set. The latter is against the basic RISC philosophy but particularly suited to embedded applications where it requires minimal software support. However, due to the heavy use of method calling in many Java applications, we thought that the use of register windows might be beneficial.

Register windows are used by SPARC [84] and other processors to reduce the overhead of method calling. Each method has two sets of registers; *ins* and *outs*. *Ins* are used for caller saved registers and *outs* for callee saved registers. During a method call, an *outs* new register window is allocated and the old *outs* becomes the callee *ins*. On method return the *outs* are deallocated, *ins* become *outs* and the old *ins* are restored. Thus the caller state is saved and restored by the hardware.

According to Huguet and Lang [14], register windows have the disadvantage of requiring a large chip area in a Very Large Scale Integration (VLSI) implementation, and they increase the amount of processor context to be saved on context switching. Both of these drawbacks largely depend on the complexity of register windows, but with the availability of a billion transistors, the former constraint is relaxed. Having enough register windows, we can keep the state for more than one context and the context switching overhead can be decreased.

2.5 Modelling and Evaluation

Although it is possible to study Java implementations on real processors which exhibit the various design alternatives, we felt that there was a need to perform an evaluation using a common methodology. We therefore studied a number of different instruction set architectures together with appropriate software support using a variety of Java benchmarks.

In order to produce a useful comparison, it was necessary to postulate some simple cases which represented distinct points in the design space. We chose to compare the following models of execution:

- Bytecode: a direct execution of Java bytecode assuming no optimisation.
- Folding_2: Java bytecode execution assuming the folding optimisations used in the picoJava-I processor (up to two instructions are folded).
- Folding_3: extends the Folding_2 model to consider 3 instruction folding.
- Folding_4: Java bytecode execution assuming the folding optimisations used in the picoJava-II processor (up to four instruction are folded).
- Reg: a simple register-based instruction set together with the Cacao register allocation algorithm.
- RegWin: A register-windows based instruction set again using the Cacao algorithm.

We have implemented these models and added them to Sun’s JDK JavaVM through which the SPECjvm98 benchmarks are analysed. For this early design phase, this analysis would give us bounds on performance rather than accurate results for reg and regWin models (assuming perfect register allocation). That will be enough to decide on the register architecture. The following chapter will study the register-allocation in detail and provide a more detailed analysis demonstrating that the bounds can be nearly achieved.

The following are assumptions that are common for all the models:

- Bytecodes are directly executed by at most one CPU instruction, including complex bytecodes.
- Native methods are not considered in the analysis.

2.5.1 Bytecode Model

Assumptions

- Each bytecode is directly executed by a single instruction.
- There is no overhead for method argument passing.

Implementation Aspects

The main execution loop of Sun's JDK JavaVM is modified so that each time a bytecode is executed, an associated counter is incremented.

2.5.2 Folding Models

Assumptions

The assumptions for this model are the same as the previous model.

Implementation aspects

The following types of instructions are considered for folding_2:

- A constant load or a local load followed by an ALU instruction.
- An ALU operation followed by a local store.
- An array or field access followed by a local store.
- A local load followed by an array or a field store.

Folding_3 considers the same instruction patterns as folding_2 and adds the following patterns:

- A constant load or local load followed by an ALU instruction followed by a local store.
- Two constant and/or local loads followed by an ALU instruction.

Folding_4 considers the same instruction patterns as folding_3 and adds the following:

- Two constant and/or local loads followed by an ALU instruction followed by a local store.

Implementation Aspects

Sun's JDK JavaVM main execution loop is modified to consider the last 2, 3, and 4 instructions depending on the model. A counter is used to count the number of folded instructions.

2.5.3 Register Model

Assumptions

- An infinite register file is available.
- Method calling overhead involves saving all locals and stack temporaries (aside from constant values) upon method call and restoring them on method return as well as copying all arguments on entering a new method.
- Each bytecode is translated to at most one register-based instruction.

Implementation Aspects

The class verification code of Sun's JDK JavaVM is modified so that basic blocks are constructed and sent to the Cacao module. The Cacao module applies the Caco register mapping algorithm and returns the number of register-based instruction. A two dimensional array is used to store these numbers, where one subscript is the method identifier and the second is the instruction offset.

The main execution loop is modified so that whenever a bytecode is executed, a counter is incremented by the amount found in the corresponding entry in the array.

2.5.4 Register Windows Model

Assumptions

- An infinite register file and number of register windows are available.
- Argument variables are preallocated.
- Only non precoloured argument variables and constants, are copied to methods.
- Each bytecode is translated to at most one register-based instruction.

Implementation Aspects

The same implementation technique is used as in reg. The difference is in modifying the method call overhead calculation part of the instruction count.

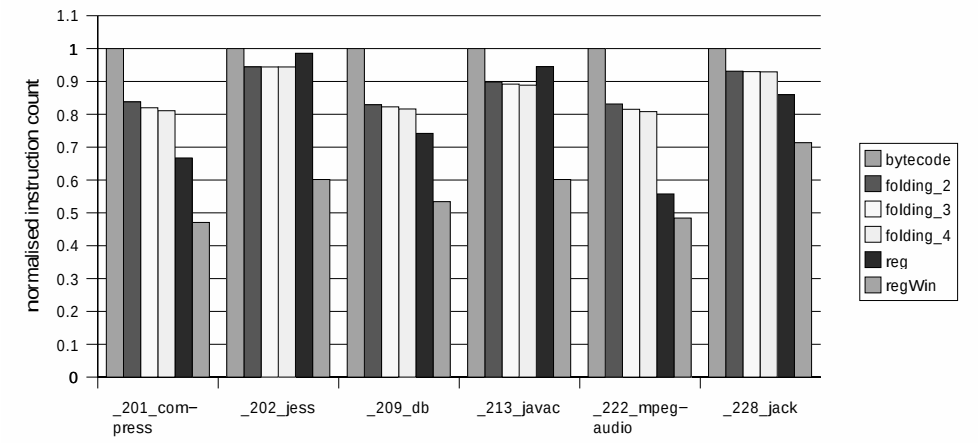

Figure 2.5: Relative instruction counts

2.5.5 Results Analysis

Figure 2.5 shows the number of instructions executed in each benchmark program for the six execution models. Folding_2 reduced the number of instructions by a maximum of 17% (for _209_db) and a minimum of 6% (for _202_jess); the average is 12%. Folding_3 and folding_4 contribute another 3% at most with average of 1%.

Reg worked better than the folding models for four of the programs; reductions ranged from 26% to 44%. For the other two programs (_202_jess, _213_javac) there is a smaller reduction of 14%. As can be seen from Figure 2.2 in Section 2.2.3, these programs have nearly double (2, and 1.7 respectively) the number of method calls of the other programs, increasing the relative call overhead.

RegWin was able to outperform all other models. This was expected observing, again from Figure 2.2 in Section 2.2.3, that method calls account for between 1% and 5% of the executed instructions. RegWin reduced the number of instructions by at least 40% for five programs. The remaining program (_228_jack) showed only a 29% reduction. This is attributed to the relatively frequent stack (at least 8%) and infrequent local and ALU operations. This suggests less sharing of local values and hence the benefit of using registers is decreased.

We have not considered the effect of ‘in-lining’ in our study. This would undoubtedly narrow the gap between the reg and regWin results for some applications. However, there are limits as to which in-lining techniques can be used for deeply nested programs and they are inapplicable in the general cases of both recursion and dynamic binding. It can be argued that most of the SPECjvm98 benchmark programs are not representative of programs written using the full object-oriented style where the above issues will be of increased relevance. We therefore believe that achieving optimum performance on real method calls is important and hence our approach is justified.

2.6 Register and Window Usage

2.6.1 Register Requirements

The previous analysis assumed an unlimited supply of registers and register windows. We re-instrumented Sun’s JDK JavaVM to count the number of local variables accessed. Figure 2.6 shows the accumulated percentage of local variable usage assuming that all local variables are mapped into registers.

The x-axis shows the number of local variables accessed. The most register hungry application (_222_mpegaudio) has a ‘knee’ at 13 registers covering 91% of variable

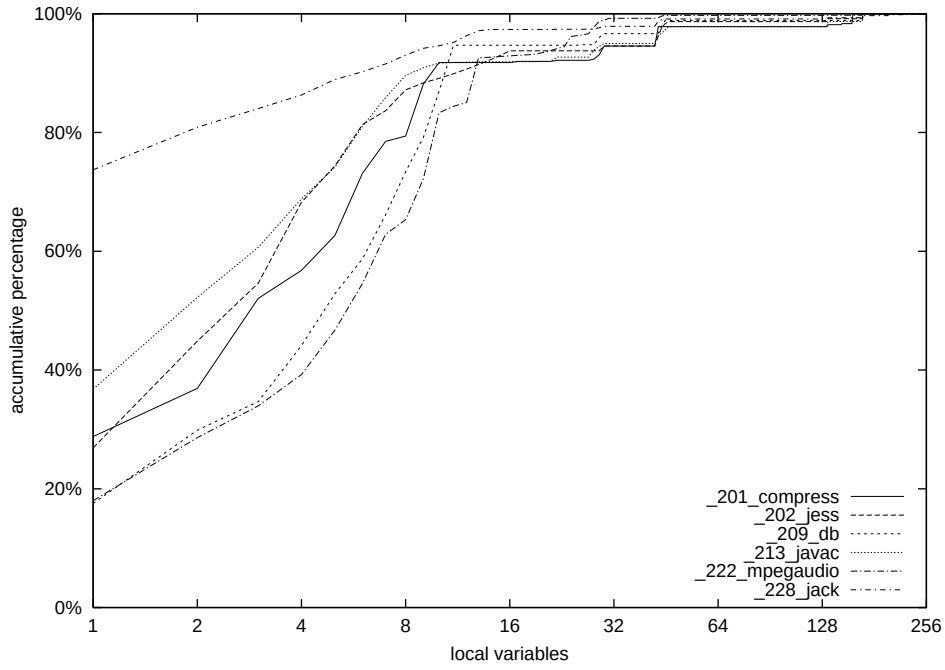

Figure 2.6: Cumulative percentage of register usage

accesses. The next significant ‘knee’ is at 30 registers where several applications achieve 95% usage. This indicates that the decision is between 16 and 32 registers although a more detailed study of dynamic register usage in the presence of dynamic register allocation algorithms is required before a final decision is reached. This is pursued in Chapter 3.

2.6.2 Register Windows Requirements

In order to determine the number of register windows, it is necessary to study the method call depth. We extended our instrumentation to provide this information. Figure 2.7 shows the call depth distribution for the various benchmark programs. The x-axis is the actual call depth, while the y-axis is the percentage of total instructions which get executed at that depth. The absolute value of the call depth is of minor importance, in fact the offset of 13 in the figure is due to a set of ‘wrapper’ methods around the benchmark suite which are executed initially. These will, of course, require register window allocation and register ‘spills’ if the window total is exceeded but this will only occur once. The important characteristic is the width of the profile. Programs such as _209_db

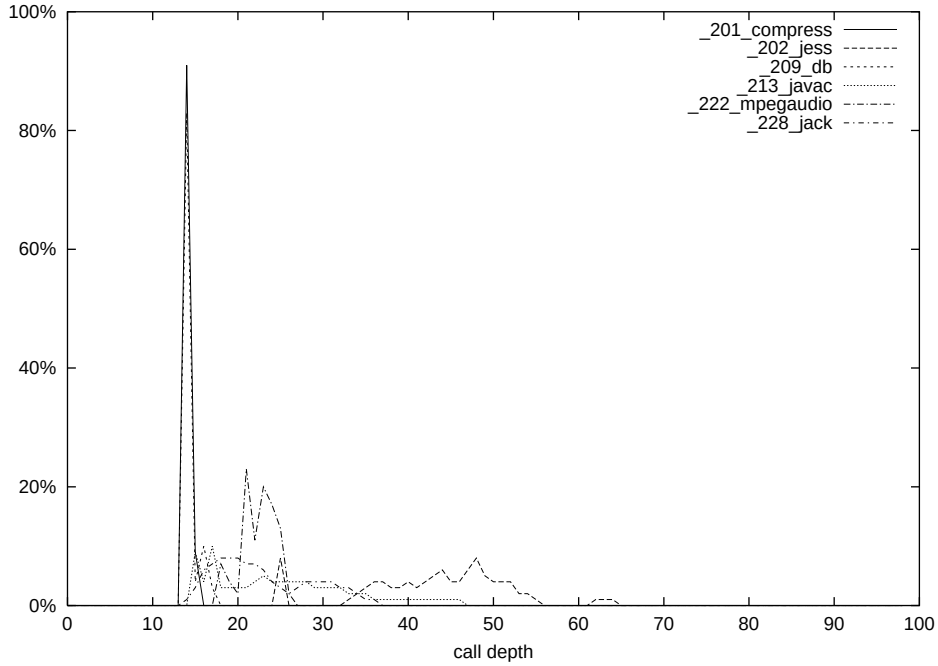

Figure 2.7: Call depth distribution

with a very narrow profile indicate that execution takes place with very shallow nesting while a broad profile like _202_jess indicates deep nesting.

Benchmark	201	202	209	213	222	228
Relative call depth	0.16	5.88	0.54	7.40	1.70	5.67

Table 2.3: Relative call depth

Flynn [20] suggests that a useful measure of the call depth of programs is the relative call depth defined as the average of absolute differences from the average call depth. Table 2.3 shows the relative call depth for the benchmarks used. This clearly distinguishes the different characteristics which are apparent from the graphical profile. However, it does not give an accurate figure for the actual number of windows needed.

A more accurate estimate of the register window requirements is necessary before design decisions can be made. Our instrumentation was yet further extended to study the way in which the provision of windows affected the execution.

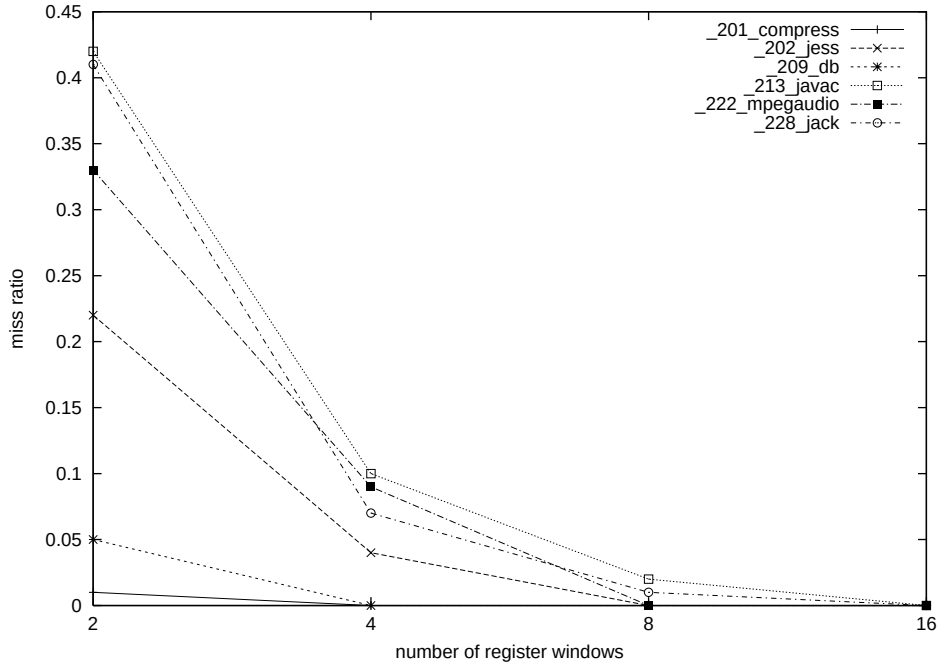

Figure 2.8: Window miss ratios

We simulated the benchmarks with a varying number of windows and measured the miss ratio of the number of window accesses resulting in window overflow or underflow to the total number of window accesses. A window access takes place twice per method call, once on entry and once on exit. The results are shown in Figure 2.8.

As expected from the relative call depth figures, two of the benchmarks have a requirement of only a small number of windows and, for them, two might suffice. However, to achieve a low miss ratio (less than 0.02) for all benchmarks, eight register windows appear to be necessary. To emphasise, at this level over 98% of all method calls would not require register spilling.

From these experiments, it is believed that we have determined that a configuration of eight register windows each containing 16 or 32 visible registers would be sufficient to achieve good performance.

2.7 Summary

In this chapter we tackled the issue of choosing a register design for Java bytecode execution. The stack and method calling overheads are identified to be a major performance impediment that is intrinsic to bytecode. Experiments show that the stack and method calling overheads are around 50% and 10% of the executed bytecodes respectively. Reviewing the literature we identified two main ways to reduce the stack overhead. The first is to dynamically fold instructions at the hardware-level using stack registers. This configuration has the potential to decrease the method calling overhead as well. The second is to rely on a fast register mapping algorithm in the software-level to remove the stack overhead. This approach seems to suffer from intrinsic method calling overhead in register-based instruction sets, especially for an object-oriented language like Java. Therefore we consider using register windows.

In order to assess these approaches, we have implemented these techniques on Sun's JDK JavaVM and used it to run SPECjvm98 benchmark programs. We found that a register-windows instruction set provided the most benefit. Folding only removed 40% of the stack overhead, while register windows removed 80%.

After we identified the significance of register windows, we studied the feasibility of this approach. We experimented with Sun's JDK JavaVM and simulated the behaviour of register windows and counted the frequency of local variable access. We found that 8 windows and 16 visible registers will cover more than 91% of all local access and 98% of method calls. This indicates that a feasible (64-128 registers) number of registers is required, hence the register window approach is of benefit.

Chapter 3: Register Windows Allocation

The previous chapter analysed the various ways that the bytecode instructions can be executed and concluded that a register-windows based instruction set is of benefit. In this chapter, we implement a Java translation system for that architecture and study the generated code and magnify the register allocation details and implementation dependent overheads. We develop register allocation optimisations and evaluate their performance. The optimisations managed to remove a significant fraction of the overheads.

3.1 The Java Translation System

Although full dynamic compilation would ultimately be required, for our preliminary work we perform static compilation. However, we do not feel that using an existing static Java compiler producing C code would provide a realistic use of our system. In particular, we want detailed control of the register allocation, as the performance of the register windows is a crucial issue, so we have developed a Java translation system. The register allocation is based on the Cacao compiler, as it provided a simple but effective mechanism as verified in Chapter 2.

Jtrans[1] is a three-pass translator: in the first pass the source (class) file is loaded and basic blocks are determined. A static stack is constructed and an approximate number of temporaries are tracked in the second pass. Code generation and register allocation are done in the third pass. The details of these passes are given below.

Pass-1: Basic Block Determination

The class file is scanned for occurrences of branch instructions and the target instructions are flagged. The class file is scanned again for the occurrences of these flags and the sequence of instructions that only starts by a flagged instruction is stored into a basic

1. The class loader was developed by Greg Wright

block structure. All basic blocks are inserted into a binary search tree having the offset from the first instruction as the key.

Pass-2: Static Stack Construction

We used the same basic block interface method used in the Cacao JIT compiler. On basic block boundaries, if the operand stack is not empty, the stack entries are allocated to fixed interface registers. The JavaVM specification [48] states that all possible paths of execution to any basic block should result in the same number of stack operands. Thus, in order to determine these entries, we start processing the first basic block then the next possible ones in the pre-order traversal visiting the current block then all its possible successors. During this process, a static stack is constructed.

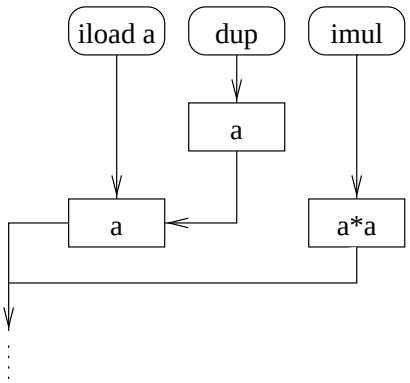

Figure 3.1: Static stack structure

The static stack, as shown in Figure 3.1, keeps track of stack operands before and after the execution of each bytecode instruction. Each entry in the static stack represents an operand in the operand stack. Each entry has a pointer to the predecessor stack entry. Every instruction is associated with a pointer, pointing to the top-most operand in the output operand stack. The input stack for an instruction can be determined by the output operand stack from the previous instruction. For the example shown, the ‘iload a’ instruction loads local variable ‘a’ onto the operand stack. ‘dup’ duplicate the topmost stack entry. This is modelled by creating a new stack entry and linking it to the previous instruction’s top of stack entry. ‘imul’ pops two operands and pushes a new one (the result

of multiplication) onto the stack. The push is modelled by creating a new stack entry for the result. The two pops are modelled by linking the new stack entry to the third stack entry in the output stack of the previous instruction (dup).

An entry in the static stack also contains: the type of operand, the variable identifier, and the variable data type. The operand types are constants, local variables, stack temporaries (stack entries that need to be copied into stack variables), and interface variables.

Ideally no code will be generated in the code generation phase for load and store instructions, if their output and input stack contain the same local variable respectively. The compute instruction directly accesses local variables and thus a later store instruction will change the output stack operand from being temporary to local. However, complications arise because of anti-dependence and output-dependence.

Anti-dependence arises in two cases:

1. A result of a compute instruction is later stored to a local variable. Between these two instructions, another instruction reads that local variable.
2. A store instruction stores a local variable that was loaded early in the program and is still in the stack.

For the first case, a copy instruction is needed and thus output stack operand of the compute instruction is not modified. For the second case, all the occurrences of local variables are replaced by a temporary variable.

Output-dependence occurs when a write to the same local variable occurs between a compute and store instruction. Output-dependence is checked by keeping track of the last instruction (number) to write to each local variable. Whenever a write to a local variable occurs by a store whose number is less than the current one, it is stored into a temporary register and the output stack operand is modified accordingly.

Method arguments are preallocated the same way as locals.

Pass-3: Code Generation and Register Allocation

In this pass, registers are allocated and code is generated. Register allocation is based on the register allocation algorithm used in the Cacao JIT compiler. Local variables are coloured by the Java compiler and thus we believe no advanced register allocation is needed. Temporaries have their live ranges explicit. Any register is allocated upon pushing onto the stack and deallocated upon popping. Complications arise due to the ‘swap’ and ‘dup’ operations, in which case we keep track of the copies in the stack and free the register when the last copy is popped.

Constant loads are combined with the corresponding compute instruction. The Cacao JIT compiler combines only consecutive instructions. On the other hand, jtrans does not impose this restriction.

Complex instructions, such as ‘new’ and ‘ddiv’, are translated into a static method call to the jtrans runtime library.

3.2 Base Register Allocation Algorithm

Figure 3.2 shows the register windows configuration. A set of 32 registers is visible to each procedure. The registers are divided into *globals, ins*, *outs*, and *extras*. *Globals* are used for thread management and global pointers. *Ins* are used for input arguments, *Outs* for passing arguments to the callee method; during a method call, *outs* become the *ins* of the callee and new registers are allocated and become the *outs* of the callee. *Extras* are extra registers that are not saved. Only the *ins* are saved by the hardware.

Registers that have a fixed allocation are the *globals*, Stack and Frame pointers, and the return program counter (PC). We seek an optimal allocation strategy for the rest of the registers. Unsaved registers are considered to be caller saved.

Base Register Allocation Algorithm

Registers are allocated to temporaries within basic block boundaries. Fixed registers act as interface registers between basic blocks. The basic block is extended to include method

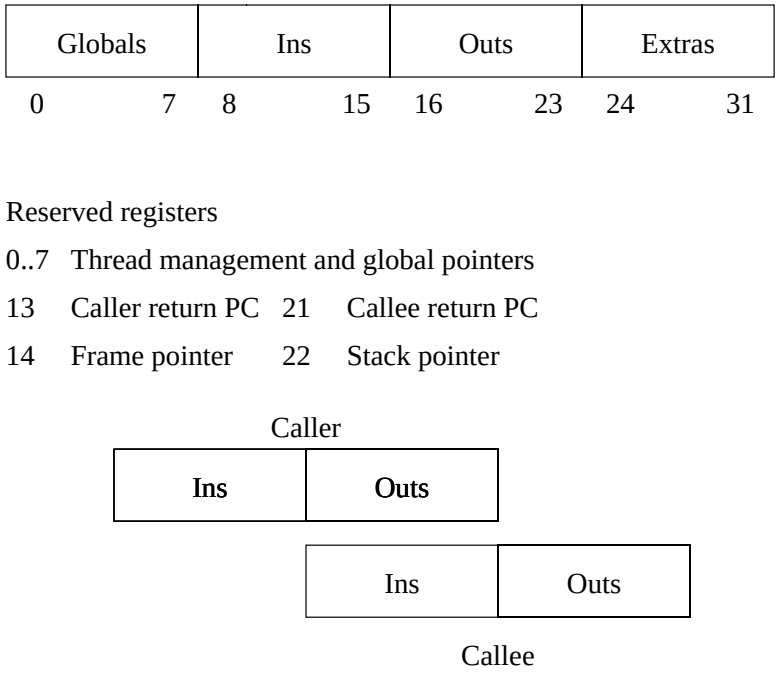

Figure 3.2: Register windows organisation

calls. The following is the basic register allocation algorithm invoked for each method (which we directly use from the Cacao JIT compiler):

1. Allocate input arguments to *ins* and then to memory if overflow.
2. Preallocate the maximum number of output arguments to *outs* and then to memory if overflow.
3. Preallocate temporaries to *ins* and then to *outs* and *extras*. Argument preallocated registers are used by temporaries if they are not currently in use.
4. Allocate the remaining local variables to the remaining *ins*, *outs*, and *extras*.
5. Allocate interface variables to the remaining *ins*, *outs*, and *extras*.
6. When a stack temporary needs creating, allocate the register from the preallocated temporaries set. If the temporary will be alive during method calls then allocate it to *ins* and then to *outs* and *extras*. Otherwise, allocate it to *outs*, *extras*, and then to *ins*.

3.3 Evaluation

Having decided on a base register allocation algorithm, we are now in the position to experiment with applications and examine the generated code. The analysis is concerned with instruction mapping overhead and the register requirements.

3.3.1 Bytecode Mapping Overheads

In order to analyse the generated code, we classify the various overheads into:

- Ideal: the number of instructions required assuming no overheads; every bytecode is mapped to at most one JAMAICA instruction. These instructions are the ones we have counted in the previous chapter.
- Method: extra instructions required for saving locals and temporaries across method calls.
- Complex: extra instructions required for saving locals and temporaries across specific method calls and passing arguments. These methods implement complex bytecodes.
- RegSpill: extra instructions required for handling register spills and fills and moving immediates into registers.
- Other: All other remaining instructions for handing complex bytecodes such as lookupswitch, and double, long, field, and array operations.

Figure 3.3 shows the static instruction count breakdown for a set of kernels of the SPECjvm98 programs. For simplicity, hereafter we will refer to individual programs by their names and will omit their identification number.

We might verify the results obtained in the previous chapter for the regwin execution model by comparing the ideal components for the benchmarks. Table 3.1 compares the normalised static ideal components with the corresponding ones obtained dynamically for the full applications. The results show a similar trend. We do not expect to have identical results due to the inherent differences between static and dynamic analysis and analysing only kernel methods. Nevertheless, the results seems to be quite near (within 17% on average).

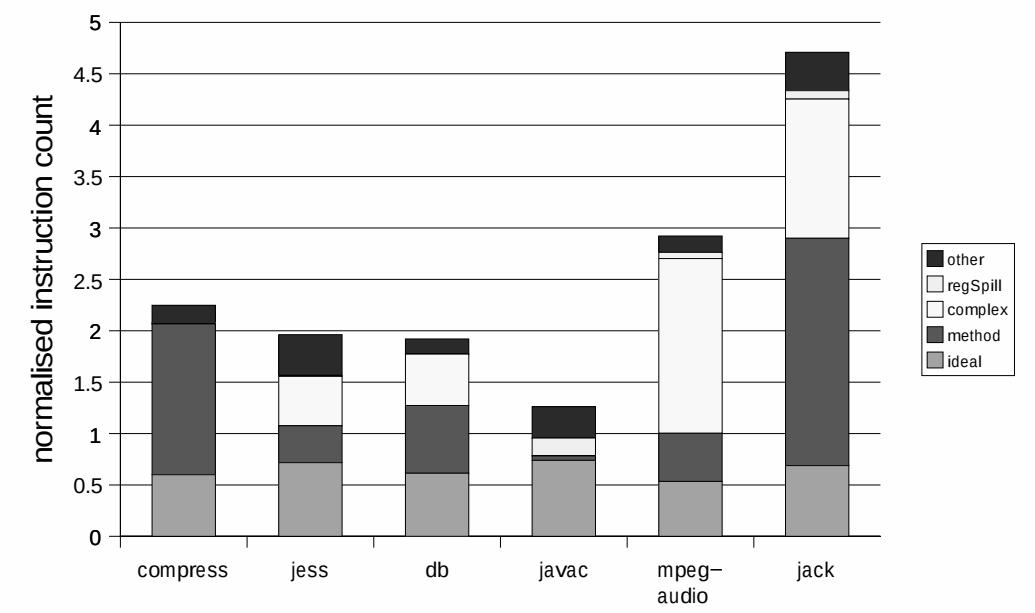

Figure 3.3: Instruction set dependent overheads

Program	com-press	jess	db	javac	mpeg-audio	jack
Ideal (static kernels)	0.6	0.72	0.61	0.74	0.53	0.69
Ideal (dynamic)	0.47	0.6	0.53	0.6	0.48	0.71

Table 3.1: Ideal components using static kernels and full applications

Referring again to Figure 3.3, the most significant overhead is method; on average it ccounts for about 31% of the instructions. The next most significant overhead is complex which is about 23%. Register spills accounted for only 3%. Other accounted for 12% on average, where they are independent of the register allocation algorithm.

3.3.2 Variable Usage

Figure 3.4 shows the distribution of temporaries. The x-axis represents the number of temporaries required to be active and the y-axis represents the percentage of instructions that use them. The compress, db, and javac programs do not require any temporaries for more than 86% of the instructions. Mpegaudio and jack require no arguments for more

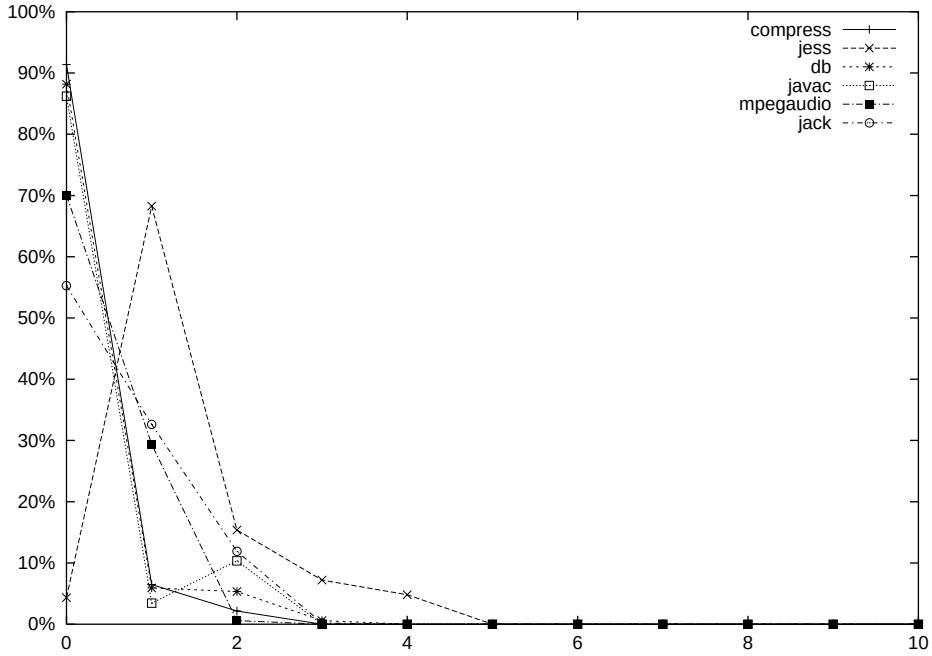

Figure 3.4: Temporaries distribution

than 55% of the instructions, and more than one temporary is only required for less than 12% of the instructions. Jess has the biggest temporaries requirement. However, the average requirement is 1.4 registers, and 2 or more registers are required by less than 16% of the instructions.

Almost no interface registers are used for all the programs examined (less than 0.6% for the compress application and none for all others). Locals are the significant requirement as shown in Table 3.2. Locals, interface register, and temporaries that need saving, are

Program	com-press	jess	db	javac	mpeg-audio	jack
Locals	9	4	9	3	13	14

Table 3.2: Number of local variables accessed in each kernel

saved during method calls. Given that we only have 8 saved registers in the *ins* window, it is essential to have an efficient use of them.

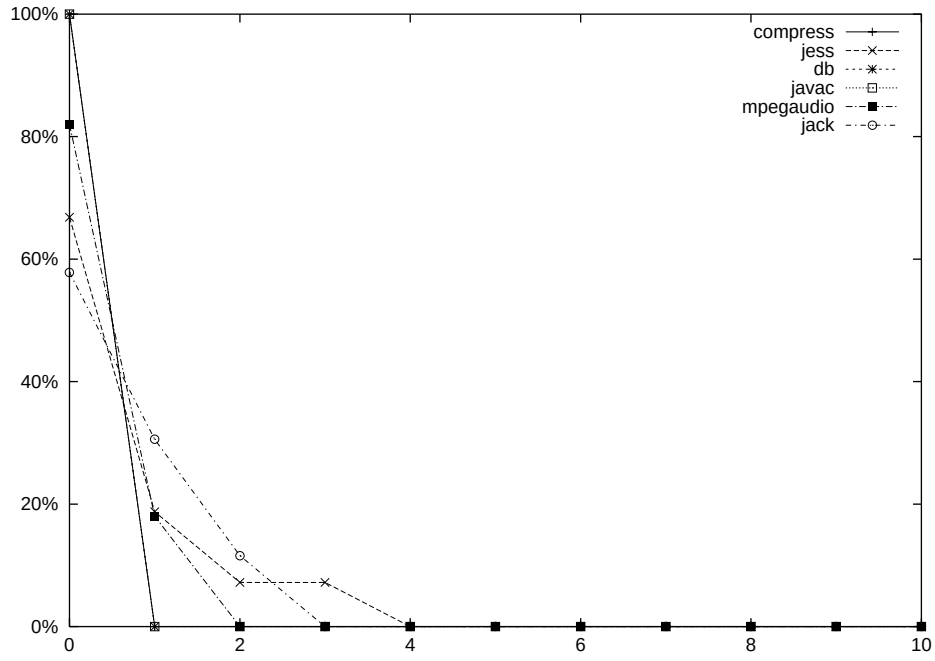

Figure 3.5: Saved temporaries distribution

Figure 3.5 shows the number of temporaries which need saving across method calls. Three of the programs (compress, db, and javac) did not require any saving. The other three required less than one saved temporary on average (0.55). So an efficient allocation would involve storing one or two temporaries into saved registers.

All the kernels considered did not pass more than 6 arguments in any method call. So having six free *out* registers is enough.

3.4 Optimisations

The main points observed from the previous section are:

- Method call overheads are significant.
- Most of the stack temporaries are removed.

We propose three optimisations to decrease the method call overheads. In the original algorithm, the number of stack temporaries and interface variables is set to the maximum operand stack depth. The exact number of temporaries is available at the third pass after the register allocation is done. It is hard to determine it earlier in the second pass. This is because the second pass identifies temporaries quickly and does not track aliases as this might complicate the algorithm. The first optimisation (temp) approximates the number of temporaries by assuming all the temporaries are unique in the operand stack. The maximum number of temporaries is then used to preallocate the temporary registers.

For non leaf methods, the cost for saving and restoring the local variables for method calls/returns is more significant than saving and restoring temporaries. For the previous analysis there are, on average, 8.7 local variables and only 1.4 temporary variables per method. The second optimisation (local) allocates local into memory when there is no space in the *ins* instead of storing them into memory.

On average, one temporary is alive across method calls. The third optimisation (savedtemp) preallocated one temporary into the *ins* before step 4 in the original algorithm (allocating the remaining locals into *ins*, *outs*, and *extras*).

3.5 Results

Figure 3.6 compares the optimisation techniques. ‘all_off’ indicates that all optimisations are inactive. ‘only_<optimisation_name>’ means only that optimisation is active. ‘all_on’ indicates that all optimisations are active.

Only_temp reduced method overhead for most applications (all except javac, and jack). Reductions range from 10% to 79%. This reduction is attributed to the efficient use of the *ins* registers. Table 3.3 gives the maximum operand stack size for a bytecode execution versus the maximum stack size after mapping to the register execution model. It is worth noting that these values match the predicted values. For jack, only_temp increased the method overhead by 14%. This is attributed to the fact that this kernel has the biggest number of local variables (14) and thus decreasing the number of preallocated temporaries increases the number of local variables saved in registers and thus more

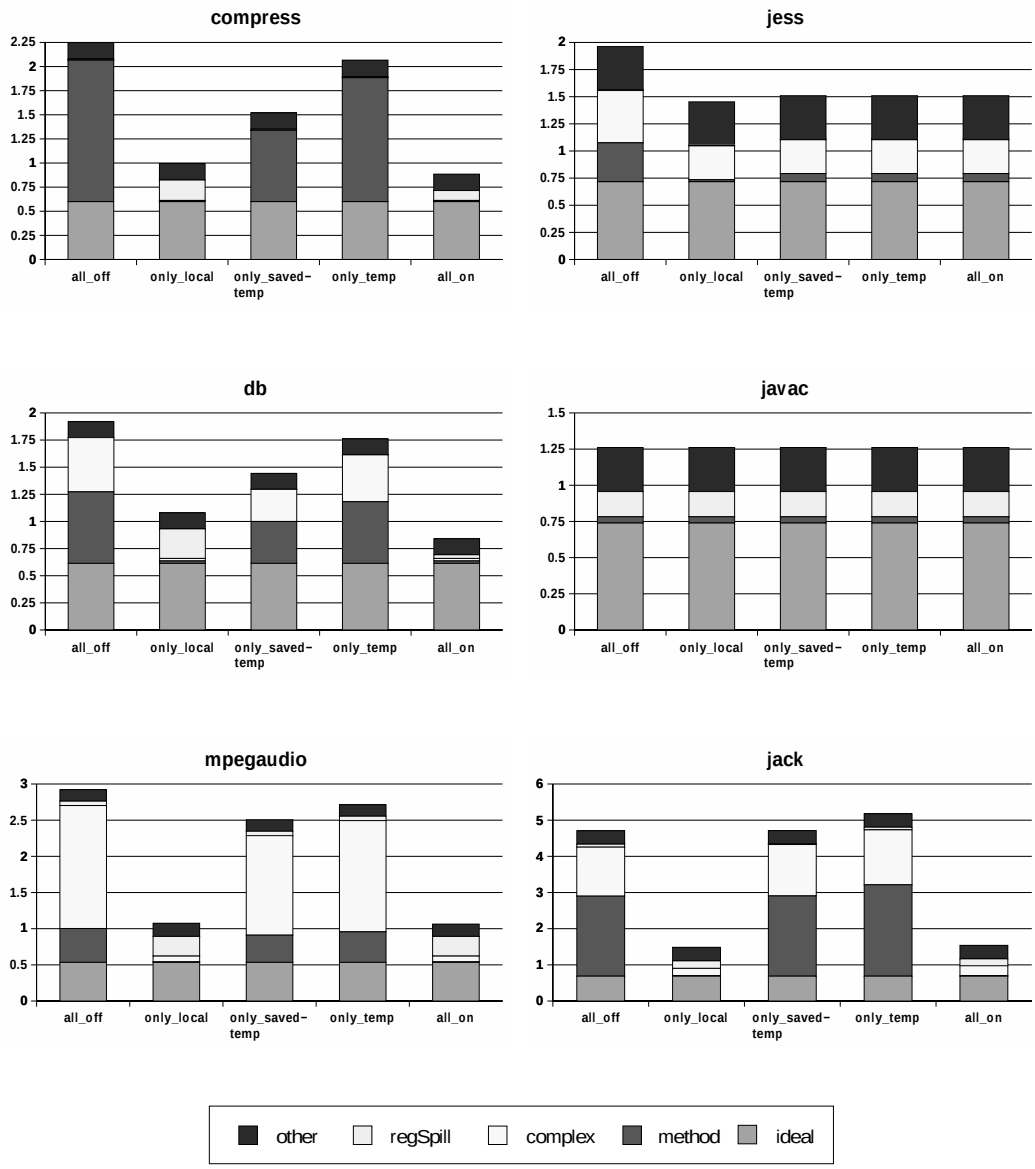

Figure 3.6: Comparing register allocation optimisations

registers need to be saved during method calls. Only_temp did not decrease the overhead in the case of javac as that kernel hardly has any overhead.

Only_savedtemp reduced method overhead slightly, which is better than the only_temp case. The jack case did not suffer like the only_temp case as this case only allocates one temporary in the *ins* so only one local is not allocated in *ins* and needs saving. However, this decreases the overhead of regSpill as now at least one temp is allocated in a register.

Only_local achieved greater reductions of the method and complex overhead than all other optimisations (95% reduction). However, register spills are introduced for four of the programs; compress, db, mpegaudio, and jack. This is attributed to the fact that local variables (for non leaf methods) are stored in memory (observing from Table 3.2 that these applications have greater local variable requirements than other applications).

Max stack	com-press	jess	db	javac	mpegau-dio	jack
Bytecode	6	5	5	3	4	8
Register	4	1	3	2	2	4

Table 3.3: Maximum bytecode and mapped instructions (register) stack

Having all the optimisations on, 73% of the overheads (method and complex) are removed on average. The regSpill overhead in javac is totally due to copying immediates into registers. So register allocation has no effect.

3.6 Summary

In this chapter we have examined the bytecode translation into JAMAICA code in detail. We have developed a Java bytecode translator tool to generate and analyse the JAMAICA code. The translator is based on the Cacao JIT compiler design. Examining the generated code, we found that saving and restoring local variables is a significant overhead. We also found that method arguments and temporaries are few. In order to exploit this, we developed a variety of optimisations and compared their effect. We found that storing local variables in saved registers and memory is better than storing them to non saved registers.

Part II: Multimedia Design Aspects

Chapter 4: Multimedia Video Workloads

Understanding the behaviour of multimedia applications is crucial for determining the design space for hardware support. Processor design for multimedia is a new research area, and few studies have been conducted in the literature. Therefore, this chapter examines a typical multimedia application in detail, identifying its characteristics and verifying other characteristics published in the literature.

The chapter starts by defining application dependent features that describe the behaviour of the software as independently as possible, from the underlying hardware. Known multimedia characteristics in the literature are examined as well as a detailed examination of MPEG-2 multimedia video applications to identify further features.

The chapter concludes by forming a set of processing requirements for the design in later chapters. The video applications also serve as our benchmark for evaluating performance.

4.1 Workload Characterisation

Our goal of workload characterisation is to isolate the behaviour of software from the underlying hardware, thereby forming a set of processing requirements from the software point of view. This leaves scope for the hardware design to exploit these requirements. We view the hardware as a von Neumann architecture with a sequence of instructions specifying operands and operations.

4.1.1 Application Dependent Features

The application requirements can be explained in terms of:

- Operands types: describe the structure, values, sizes of operands, and the way operands are accessed.
- Operation types: describe what the operations do, and the number and types of operands operated on.

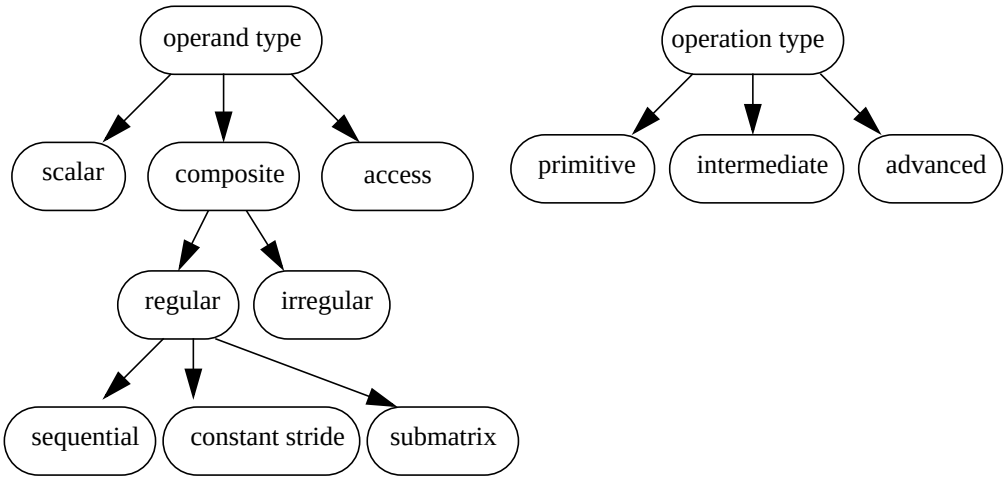

Figure 4.1: Classification of operand and operation types

Figure 4.1 shows our classification of the operand and operation types. The first subtype of operand type is the scalar type. A scalar operand type is a set of numbers or other enumeration such as characters and boolean. These operands are directly accessed in an instruction. This is the most common operand type used in conventional applications. The second subtype is the composite type. In this type an element would be a collection of scalar elements or other composite elements. In this type we distinguish between composite types where all elements are organised and accessed in a regular way and types with irregular organisation and/or access.

In the regular subtype we assume that all elements are organised the same way they are accessed. The first two subtypes of the regular type are sequential and constant stride types. In the sequential type elements are organised in rows, while in the constant stride type, elements are organised in columns and the elements are at a constant distance from each other. These operand types are mostly used in matrix operations. The last subtype of the regular operand type is submatrix, where operands are organised in a small matrix inside a bigger one.

The other subtype of the composite type is the irregular type. This type represents composite types with irregular access and element organisation such as sparse matrices.

The last subtype of the operand type is the access type. This type represents pointers to other data types. We believe that this is not typical in multimedia applications.

Operations are decomposed into primitive, intermediate, and advanced. Primitive operations are integer and floating point operations that are performed in a general-purpose processor. Advanced operations implement an application specific function such as the sum of absolute differences or other media specific operations. Intermediate operations are between the other two and do not require the high complexity of advanced operations but are still more complex than primitive operations. Operations of this type include multiply accumulate which is used in DSP applications.

4.1.2 Workload Measurement Techniques

Identifying these requirements is not trivial. Aside from the problem of selecting a representative workload, there is the problem of isolating the effect of compilation optimisation or the way in which the implementation of the algorithm has been expressed. These problems can be reduced by resorting to statistical techniques where many programs are analysed and some statistical measure is used (such as moments).

One of the basic techniques in measuring workload characteristics is based on counting the frequency of features of interest in a particular implementation. For example, getting operation execution frequencies, data constant sizes, cache hit ratios, and other resource statistics. This is achieved by either using hardware monitors or software techniques like simulation. The problem with this technique is isolating the effect of the underlying architecture on the results. Also this suffers from including a rather mixed behaviour.

Another technique is profiling the program in question and finding out critical sections. These sections are usually inner loops and are referred to as kernels. The critical sections are identified by some cost function which is usually time (but can also be lines of codes, instruction count, or execution times). This technique has the advantage of studying the kernels and identifying kernel characteristics that cannot be identified by the other

technique. The disadvantage of this technique is that it is not automated and every kernel needs to be manually examined.

We thus believe that a combination of these techniques is preferable. As most of the characterisation studies in the literature are based on the first technique, we will complement it with study of the kernels.

4.2 Existing Measurements

Lee et al. [44] proposed the multimedia benchmark MediaBench. It includes audio, image, video processing, encryption, and voice recognition applications. They analysed its execution on a single-issue processor architecture.

Firtts at al. [22] added two extra video processing applications to MediaBench and conducted a set of experiments on an intermediate low-level format that is relatively architecture independent.

Operation type	Percentage of instructions	Comments
ALU	40%	Similar to conventional applications
Load/Store	26-27%	Varies and reaches 35% for video applications
Branch	20%	Highly predictable
Shift	10%	Relatively high
Int mult	2%	Relatively low
Floating point	3-4%	Relatively low

Table 4.1: Operation distribution (Fritts et al. [22])

Both studies showed that the low-level integer operations are similar to SPECint95 (as a representative of conventional applications). The breakdown is shown in Table 4.1. It is worth noting that floating point operations are relatively low and this emphasises that multimedia programs are mostly integer. ALU operations are similar to conventional applications. Load/Store operations are relatively high for video and image processing

applications. Branch operations are similar to conventional applications but predictable. A static prediction achieved 89.5% accuracy.

Also, spatial locality is high, removing 76.1% of data cache misses and 86.8% of the instruction cache misses. This is achieved by using a 64 KB direct mapped cache and varying the cache line size from 8 to 1024 and measuring the relative reduction in the number of total cache misses.

The operand distribution is shown in Table 4.2. These data show that most of the operand sizes require 16 bits or less.

Operand size	Usage frequency
8-bit	40%
16-bit	51%
32-bit	9%

Table 4.2: Operand distribution (Fritts et al. [22])

Property	MediaBench	SPECint95
Instruction cache miss ratio	0.5%	1.5%
Bus utilisation	6%	17%
Instructions per cycle	More than 0.8	Less than 0.8
Data cache	More effective for reads	More effective for writes

Table 4.3: Differences between MediaBench and SPECint95 (Lee et al. [44])

Lee et al. [44] showed that MediaBench is statistically different from SPECint95 in the four properties shown in Table 4.3.

Also, parallelism is higher in multimedia applications than conventional applications; Liao [47] has studied available parallelism in video applications and found large amounts of instruction-level parallelism (ILP) ranging from 32.8 to 1012.7 independent

instructions that can be issued per cycle for an idealistic architecture model. Wall [81] has found about 10 instructions on average for conventional integer applications.

Although these results are low-level, some conclusions can be drawn concerning the high-level aspects of multimedia. Firstly, a relatively low instruction cache miss rate implies that most of the execution happens in tight loops. This is emphasised by the high predictability of the branches. Secondly, high spatial locality and less effective data cache for writing suggests that streaming data is read. Thirdly, bus utilisation indicates that multimedia applications are more computationally bound. Fourthly, small data types of sizes 8 and 16 bits are used frequently. And finally, high degrees of ILP with streaming data access may suggest that data-level parallelism is abundant.

4.3 MPEG-2 Video

In the previous section we presented low-level characteristics of multimedia applications reported in the literature. In this section we complement that study by examining multimedia kernels of video applications.

Video processing is an important application that contains many kernels that are used in other applications. We did not consider audio processing applications as they demand far less processing power than video. For a PAL 25 frame/sec film, the processing requirement to encode it in real-time will be in the order of 10 GOPS, which is indeed very computationally intensive. Also, video is currently receiving major attention as digital video is taking over from analogue video, and standards have been developed.

The Motion Picture Expert Group has defined the digital video MPEG standards. There are currently four standards: MPEG-1 [24] for low quality video mainly used in video conferencing and video CDs, MPEG-2 [33] is a broadcast quality used in digital television, MPEG-4 [55] is designed for the web with lower bit rates than MPEG-2, and MPEG-7 [56, 57] which is concerned with describing the video content to be accessible to user searches. MPEG-2 is a widely used standard in digital television and DVD and the source code is available, so we will consider it in our analysis. MPEG-4 is a superset of MPEG-2, it provides 3D object modelling and speech synthesis. However, being a more

recent standard little has been developed in terms of encoders and decoders, and to our knowledge, no source code is yet available. We thus focus on the MPEG-2 standard [33].

There are two main applications in MPEG-2; mpeg2encode and mpeg2decode for encoding and decoding (compression/ decompression) video respectively.

Figure 4.2 gives an overview of the encode operation. Compression exploits two kinds of redundancy; spatial and temporal. Spatial redundancy occurs within a frame. A technique called discrete cosine transform (DCT) is used to exploit that redundancy. The frame is divided into small blocks of 8×8 pixels and each block is transformed.

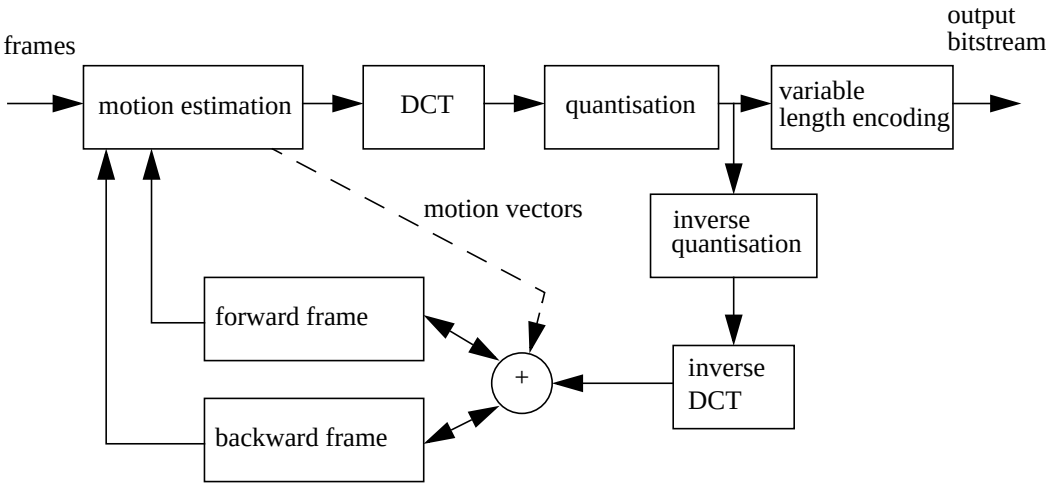

Figure 4.2: Block diagram for MPEG-2 encoding

Temporal redundancy refers to information repeated across the frames (time-wise). For instance, a frame sequence showing a moving object will have the same object information but in different locations. Predictions from an earlier frame and a later one are used to construct that object. To facilitate that and to account for the fact that a video sequence can be randomly accessed, three types of frames are identified; intra, non-intra, and bidirectional frames. An intra frame is only spatially coded. A non-intra frame is forward predicted from the previous intra frame. A bidirectional frame is predicted from a previous intra or non-intra frame and a later non-intra frame. Prediction is done on larger

16×16 pixel blocks called macroblocks. The predicted macroblock is subtracted from the original macroblock and the resulting prediction error values are spatially encoded using DCT. The two reference frames that are used in producing the predictions are reconstructed frames. The DCT blocks are then quantised, variable-length encoded, and stored in a bit-stream file.

Colour information is also used to achieve compression. For every frame, there are three rectangular matrices of 8-bit numbers (pictures) representing colour information (one luminance, and two chroma). The human eye is less sensitive to colour than luminance, so a 4:2:0 format is used. In that format, for every four luminance components, two chroma components are sampled. The chroma frames are thus subsampled in the horizontal and vertical directions.

The decoding process is basically the opposite of the encoding process. The encoding process is more computationally intensive, especially the motion estimation operation as it involves comparing a macroblock with a set of nearby blocks in two other frames (forward and backward frames).

4.3.1 Kernels

This section examines the MPEG-2 encode and decode kernels of the MPEG Software simulation Group [28]. The source code is manually translated from C into Java and is used hereafter as our benchmark. The workload contains two 720×480 frames in the video file. Experimenting with more frames and different content did not affect the execution profile significantly. The other video related parameters such as frame sizes, and bit rates are specified according to the main-level main-profile configuration for MPEG-2. The kernels listed here account for about half the execution time. We present here the profiling information obtained from running the C versions (on a Sun Ultra-5 workstation with an Ultra-SPARC II 333 MHz processor). This profiling information is consistent with the Java versions running under simulation.

The breakdown of mpeg2encode is shown in Table 4.4. The table lists the operation execution percentage over the total execution time of the whole application. Most of the execution time is spent in motion estimation (dist1 kernel).

Operation	% exe time
Motion estimation (dist1)	71%
Discrete cosine transform	9%
Subsampling	2%
Predication	1%
Quantisation	1%
Variable length encoding	1%
Other kernels	14%

Table 4.4: mpeg2encode profiling information

The execution profile for mpeg2decode is shown in Table 4.5. The execution is spread across many kernels, we accordingly increase our profiling granularity to the kernel level.

Figure 4.3 shows the Java source code for the dist1 kernel. This kernel is used in motion estimation. It calculates a first order measure of distance between two macroblocks which is the sum of absolute differences. The extra mask operation (&0xff) is needed as Java does not support unsigned byte data types. The outer loop is exited if the partial sum ‘s’ is greater than some constant ‘distlim’. The kernels also implement the half pixel search operation, where other operations are required to calculate mid-way values in both the horizontal and vertical directions. However, the search window for the latter case is usually very small compared to the former case, and thus it has very little effect (more than 90% of distance operation is done on full pixels).The kernel is repeatedly evaluated for different frame macroblocks spiralling outward. Macroblocks reside inside a frame and are accessed in a submatrix way.

Figure 4.4 and Figure 4.5 show the Java source code for the horizontal and vertical interpolation kernels. These kernels implement a finite-impulse-response filter (FIR) and use it to interpolate the chroma samples and construct the chroma frames. The vertical kernels traverse the image in a row major order which is likely to incur cache misses. The main data type is a large, unsigned byte, matrix (representing the luminance picture), and the main operation is a dot product. These kernels are used in mpeg2decode. For mpeg2encode, there are similar kernels used for subsampling frame data. The kernels use

Operation	Kernels	Description	% exe time
Interpolation filter			**53.41%**
	conv420to422	Vertical 1:2 interpolation filter	22.51%
	conv422to444	Horizontal 1:2 interpolation filter	21.92%
	store_ppm_tga	Yuv2rgb	8.98%
Frame decoding			**13.89%**
	idctcol	IDCT on columns	4.91%
	idctrow	IDCT on rows	3.71%
	saturate	Saturate 8x8 block	3.59%
	clear_block	Write zero in 8x8 block	1.68%
Prediction			**12.10%**
	form_comp_pred	Form prediction	7.43%
	Add_Block	Add prediction to error	4.67%
Variable length decoding			**7.07%**
	Flush_buffer	Advance buffer	2.51%
	Decode_MPEG2_Intra_Block	VLD intra blocks	2.16%
	Show_Bits	Show n bits	1.20%
	Decode_MPEG2_Non_-Intra_Block	VLD non intra blocks	1.20%
Output			**10.66%**
Other			**2.87%**

Table 4.5: Execution profile of mpeg2decode

```
for (j=0; j<16; j++){
  for (i=0; i<16; i++){
    if ((v = (blk1[off_1+i]&0xff)  - (blk2[off_2+i]&0xff))<0)
      v = -v;
    s+= v;
  }
  if(s>=distlim)
    break;
  off_1+= lx; off_2+= lx;
}
```

Figure 4.3: dist1 kernel

```
for (j=0; j<Coded_Picture_Height; j++){
  for (i=0; i<w; i++){
    i2 = i<<1;
    im2 = (i<2) ? 0 : i-2;
    im1 = (i<1) ? 0 : i-1;
    ip1 = (i<w-1) ? i+1 : w-1;
    ip2 = (i<w-2) ? i+2 : w-1;
    ip3 = (i<w-3) ? i+3 : w-1;

    dst[i2+dst_offset] = src[i+src_offset];

    dst[i2+1+dst_offset] =(byte)Clip[Clip_offset+((int)(
              21*( (src[im2+src_offset]&0xff)+
                  (src[ip3+src_offset]&0xff) )+
              52*( (src[im1+src_offset]&0xff)+
                  (src[ip2+src_offset]&0xff) )+
             159*( (src[i+src_offset]&0xff)+
                  (src[ip1+src_offset]&0xff))+128)>>8)];
  }
  src_offset+= w; dst_offset+= Coded_Picture_Width;
}
```

Figure 4.4: conv422to444 kernel

similar operations but their usage percentage is low (less than 1%). The FIR filters are widely used in DSP applications.

```
for (i=0; i<w; i++){
  for (j=0; j<h; j++){
    j2 = j<<1;
    jm3 = (j<3) ? 0 : j-3;
    jm2 = (j<2) ? 0 : j-2;
    jm1 = (j<1) ? 0 : j-1;
    jp1 = (j<h-1) ? j+1 : h-1;
    jp2 = (j<h-2) ? j+2 : h-1;
    jp3 = (j<h-3) ? j+3 : h-1;

    dst[w*j2+dst_offset] = (byte) Clip[Clip_offset +
    ((int)( 3*(src[w*jm3+src_offset]&0xff)
           -16*(src[w*jm2+src_offset]&0xff)
           +67*(src[w*jm1+src_offset]&0xff)
           +227*(src[w*j+src_offset]&0xff)
           -32*(src[w*jp1+src_offset]&0xff)
           +7*(src[w*jp2+src_offset]&0xff)+128)>>8)];

    dst[w*(j2+1)+dst_offset] = (byte) Clip[Clip_offset+
    ((int)(  3*(src[w*jp3+src_offset]&0xff)
             -16*(src[w*jp2+src_offset]&0xff)
             +67*(src[w*jp1+src_offset]&0xff)
             +227*(src[w*j+src_offset]&0xff)
             -32*(src[w*jm1+src_offset]&0xff)
             +7*(src[w*jm2+src_offset]&0xff)+128)>>8)];
    }
    src_offset++; dst_offset++;
  }
}
```

Figure 4.5: conv420to422 kernel

Figure 4.6 shows the Java source for a prediction kernel (form_comp_pred_av). It performs an average operation over four pixels in a macroblock from a reference frame to construct prediction values for the reconstructed macroblock. This kernel is used for macroblocks that are reconstructed from a macroblock that are at half-pixel distances in both horizontal and vertical directions. Kernels that average two pixels are used for macroblocks that are a half-pixel distance in one dimension. For macroblocks with no displacement, a direct copy operation is used. Another prediction kernel (form_comp_pred_cp) is shown in Figure 4.7. Operations similar to these kernels are

```
for (j=0; j<h; j++){
  for (i=0; i<w; i++){
    d[i+d_offset] = (byte)(((s[i+s_offset]&0xff)+
                           (s[i+1+s_offset]&0xff)+
                           (s[i+lx+s_offset]&0xff)+
                           (s[i+lx+1+s_offset]&0xff)+
                           2)>>>2);
  }
  s_offset+= lx2; d_offset+= lx2;
}
```

Figure 4.6: form_comp_pred_av kernel

```
for (j=0; j<h; j++)
  for (i=0; i<w; i++){
    d[i+d_offset] = s[i+s_offset];
  }
  s_offset+= lx2;
  d_offset+= lx2;
}
```

Figure 4.7: form_comp_pred_cp kernel

used in other prediction and frame decoding kernels (saturate, clear_block, and add_block).

```
for (i=0; i<8; i++)
  for (j=0; j<8; j++){
    s = 0;
    for (k=0; k<8; k++)
      s+= c[j][k] * block[8*i+k];

    tmp[8*i+j] = s;
  }
}
```

Figure 4.8: DCT kernel

Figure 4.8 shows the Java source for the DCT kernel (similar to the inverse DCT kernel) which is basically two matrix multiplications and again the matrices are submatrices in a bigger matrix. However, there are optimal algorithms that reduce the number of additions and multiplications significantly [82]. These algorithms are irregular in accessing the data and thus might not benefit from hardware mechanisms.

Also the variable length encoding and quantisation kernels have irregular data access, their execution frequencies are not significant and thus we did not take them as representative kernels.

4.3.2 Findings

Kernel	Main operation	Operand type
dist1 (motion estimation)	$\sum_{i=0}^{15}\sum_{j=0}^{15}\lvert a_{ij}-b_{ij}\rvert$	submatrix
conv422to444 (horizontal interpolation)	$\sum_{k=0}^{5} c_k \cdot s_{i-k-3,\,j}$	constant stride
conv420to422 (vertical interpolation)	$\sum_{k=0}^{5} c_k \cdot s_{i,\,j-k-6}$	sequential
form_comp_pred_av (prediction processing)	$\frac{s_{i,j}+s_{i+1,j}+s_{i,j+1}+s_{i+1,j+1}+2}{4}$	submatrix
form_comp_pred_cp (prediction processing)	$d_i = s_i$	submatrix
DCT/ IDCT	$\sum_{u=0}^{7} a_{ux} \sum_{v=0}^{7} a_{vy} \cdot f_{uv}$	submatrix

Table 4.6: Representative MPEG-2 kernels

All the kernels examined have operands of type composite with regular access. Table 4.6 summarises the operand and operation types. A multiply accumulate instruction would be helpful in implementing the dot product operation used in two kernels. As most data types are 8- and 16-bit wide and overflow is usually clipped, saturation arithmetic might also be helpful. This would help implement the absolute differences operation in dist1 (by two

saturation subtractions and an ‘or’ operation). Supporting the whole operation in hardware is less emphasised as we are not designing an application specific architecture and reusing resources among other instructions in a general-purpose processor is encouraged.

All the kernels are tight loops, with fixed loop boundaries (simple control flow), integer based (no floating point operand type is used) with small data types 8- and 16-bit. Data parallelism is abundant. A submatrix addressing mode would be beneficial for many kernels.

4.4 Summary

In this chapter we studied multimedia workload characteristics. We reviewed the literature for characterisation studies. Most of the studies relied on a low-level measurement of multimedia execution features such as operation usage frequency and cache hit rates. While these studies are useful, they did not provide much information about the high-level requirements. We are interested in finding what high-level operations and data types are used. To enable such a study, we examined the most used kernels of MPEG-2 video encode and decode operations. MPEG-2 operations are important multimedia applications that contains many kernels that are used in other multimedia applications. A large percentage of the kernels followed a submatrix data access pattern. We believe that this form of access can be exploited in hardware in addition to other established characteristics such as small data types and abundance of data-level parallelism. The next chapters will examine how these features can be exploited.

Chapter 5: Multimedia Architectures

The previous chapter identified the multimedia workload execution characteristics, in particular, the submatrix accessing. This chapter surveys the state-of-the-art of current microprocessor architectures and other traditional architectures. The aim of the chapter is to identify a set of candidate architectures for supporting multimedia, and propose new architectural ideas.

The chapter starts by making a taxonomy of the existing architectures and reviews examples of every type. The proposed solution is put in context, and the chapter concludes by recommending the proposed solution and candidate ones from the literature that need to be studied in more detail. That study will be done in later chapters.

5.1 Taxonomy of Multimedia Processing

There are two main approaches followed by state-of-the art processors for processing multimedia workloads:

- DSP-based processors,
- and general-purpose processors.

The DSP approach relies on implementing frequently used multimedia operations in the hardware, such as the pixel distance operation used in the dist1 kernel and inverse discrete cosine transform. On the other hand, the general-purpose approach implements primitive operations. This has the advantage of targeting more general multimedia operations.

The distinction, however, between DSP and general-purpose processors is getting more blurred. For instance MicroUnity's MediaProcessor [30], and NEC's V830R/AV [75] include relatively simple multimedia specific instructions, such as multiply-accumulate, partial absolute difference operations. Moreover they use caches instead of the local memory used in DSP architectures. They support other general-purpose processor functions such as memory management.

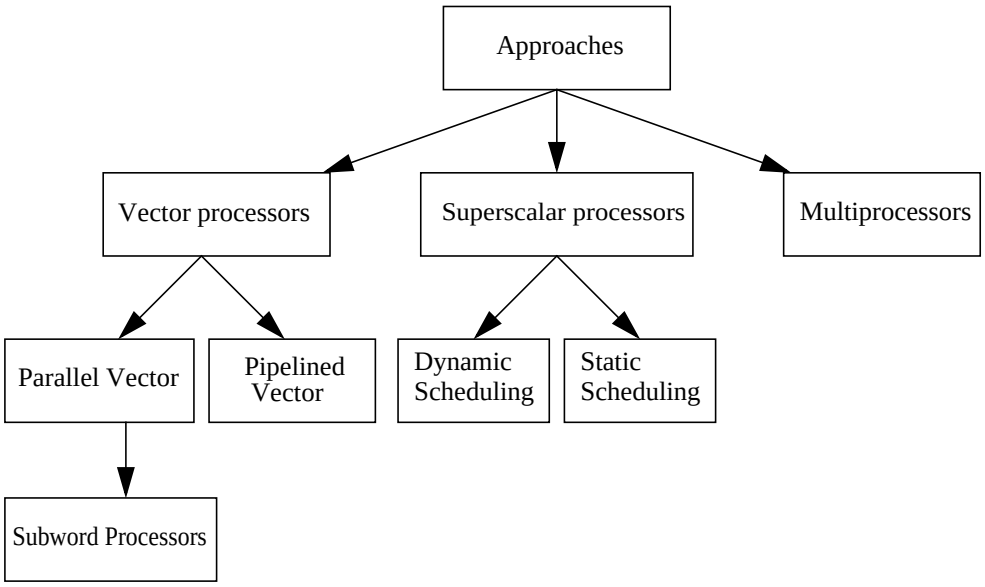

Figure 5.1: Multimedia processing approaches

In this respect, we find it more inspiring to approach our survey from the point of view of established architecture ideas, and view DSP and multimedia approaches as variations of these ideas. Figure 5.1 shows a simple taxonomy of multimedia processing architectures. This taxonomy is based on three well established architecture models; vector processors, superscalar processors, and multiprocessors. These approaches are combined in some architectures. In the following sections we describe every approach in more detail.

5.2 Superscalar Processing

Superscalar architecture [69] is used in most general-purpose processors. The architecture exploits the parallelism at the level of individual instructions. This is termed instruction-level parallelism (ILP). The dependence information between instructions is either handled statically or dynamically. The static approach relies on the compiler to pack independent instructions (from an execution schedule) and the hardware to execute them in parallel. In the dynamic approach, the instruction schedule is done dynamically and dependencies are tracked by the hardware. This increases the hardware complexity but

has the advantage of executing legacy software without recompilation and can cope with unpredictable hardware delays that cannot be determined statically.

The major drawback of dynamic superscalar processors is what is termed diminishing returns. To determine an execution schedule the hardware examines a set of instructions on the dynamic trace called an instruction window. The complexity of designing such a window is of quadratic order in the window size [61]. Also, extra registers (non architecturally visible) are required to remap architecture registers and remove false dependencies between instructions (write-after-write and write-after-read). In multimedia, the processing sequence is highly predictable and few dependencies occur between instructions. Thus this extra hardware complexity is not required. However, the existing superscalar design will obviously improve the processing power.

The static superscalar architecture is more likely to suit a multimedia processor. This approach has been used recently in the MAJC architecture [72] (described in Section 5.4.1), and many DSPs such as Chromatic's Mpact [35] and Texas Instrument's TMS320C82 [26].

A more general limitation of superscalar processors is the limits of ILP. Many studies have considered different design assumptions. Wall [81] has made a highly ambitious assumption and reaches a conclusion that for non scientific applications less than 10 instructions can be issued in parallel. This profoundly limits the potential of that approach for general-purpose applications and raises the question of different architectures.

5.3 Vector Processing

Vector processors are a good candidate for multimedia processing [19]. They were originally designed for scientific applications, such as weather forecasting and physics simulation, that match the data parallel nature of multimedia applications. A large percentage of the execution time in these applications occurs in kernels that involve performing the same operation over large regularly structured operands.

$$a_i = b_i + c_i \tag{1}$$

```
for(i=0;i<64;i++){
  a[i] = b[i] + c[i]
}
```

Figure 5.2: Vector addition

The above vector add operation (Equation 1), for example, can be specified in software as shown in Figure 5.2. Processing that expression will involve instructions for the loop control, address calculation for the arrays, in addition to the add operation. A vector processor would specify a vector add instruction where the other overheads are not incurred.

There are two main ways to implement a vector processor; the first is to replicate the functional units and achieve parallelism by processing all elements of the vector at the same time. However, data needs aligning with the appropriate functional unit at the right time. This requires an interconnection network that introduces extra cost in the design and also many paths will be required from the memory to the processor. This architecture is called a parallel vector processor and was an approach used in many early machines such as the ILLIAC IV and the Burroughs Scientific processors [37].

The second way to implement a vector processor is based on having one or relatively few pipelined functional units. Vector elements are processed in a pipelined fashion. These processors are called pipelined vector processors. This approach has been more widely adopted than the other approach as data alignment is not needed. Moreover, a pipelined design is a natural match for the repeated initiations of the same function invoked by vector instructions. Machines in this class include the CDC STAR-100, the Texas Instruments' Advanced Scientific Computer, and the Cray Research Cray-1 [37].

The memory interface of a vector processor is generally very demanding; the above vector add example would require two memory reads and one write every processor cycle. The required memory bandwidth, during that period, is faster than is available and buffering is needed to cope with this. Cray-1 had a notable advantage of making that buffering visible to the programmer in the form of vector registers. Thus with clever compilation techniques vector values are reused.

With limited memory bandwidth, the vector register approach has a clear advantage over direct memory access. Direct memory access would have benefits if memory bandwidth was greater than the computation as it does not require extra loads into registers and processing could proceed directly. The IRAM project [63] (described in Section 5.3.4) considered building vector processing in memory and thus avoiding vector registers is emphasised. However, since we are not considering that design option, we will focus on the vector register approach.

Vector architecture has been shown [45, 3] to be a cost effective solution for applications with high levels of data-parallelism. This is due to the fact that the register vector organisation does not incur a large area penalty and gives, at the same time, high bandwidth. Also the vector control logic is simple as the instruction scheduling is done explicitly by the compiler.

Given the high data-parallelism of multimedia applications, a vector architecture is a more cost effective way to exploit the data-parallelism in multimedia applications than a superscalar architecture.

5.3.1 Subword Parallelism

Subword parallelism [46] is a form of parallel vector processing. This architecture has been introduced recently as multimedia extensions to major general-purpose microprocessors. These include Intel x86's MMX [64] and Stream SIMD [76], PowerPC's AltiVec [14], UltraSPARC's VIS [77], PA-RISC's MAX-2 [46], and AMD's 3DNow! [59]. A register is viewed as a small vector with elements smaller than the register size. This requires small data types (8- and 16-bit) and large register sizes. Multimedia kernels have such data types and general-purpose processors have wide registers, satisfying these requirements.

The same operations are applied to the subwords at the same time. Figure 5.3 gives an example of an add operation for two 32-bit registers with subwords of size 8-bit. Overflows are handled by either truncation or saturation.

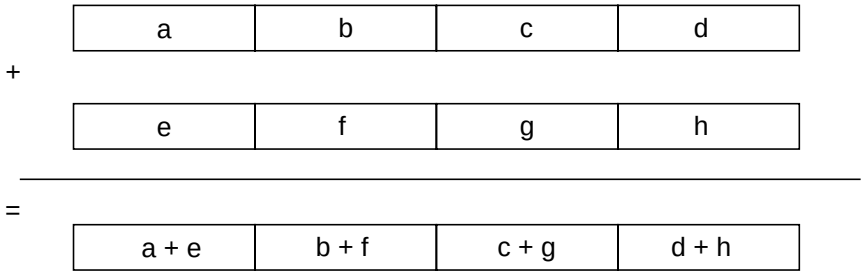

Figure 5.3: Subword addition

Subword parallelism is a cost effective solution to exploit the data-level parallelism of multimedia applications. There is no need to replicate the functional units and the memory port can provide multiple access with no cost.

However there are many overheads introduced. The nature of subword data introduces memory misalignment problems. Accessing data within one word boundary requires extra instructions. Some subword processors introduce extra formatting instructions for aligning operations such as double register logical shift operations.

Another overhead is increasing and decreasing the precision of subwords. These operations are referred to as unpacking and packing respectively. These operations are needed as intermediate results might require increased precision followed by storing them back into memory with low precision. Also since the vectors are of fixed length, increasing the machine word size will increase the chance that some subwords will not be used.

The normal addressing mode is basically stride-1 memory access. This limits the performance for column arranged data. Subword processors include permutation instructions to change the order of subwords within words to overcome such limitations. However, these instructions are very much specialised and may increase the complexity of automatically factorising the code in a compiler.

Also this approach does not remove the loop control and address generation overheads which are reduced in traditional vector processors. Loop unrolling technique may be used to reduce the loop control overhead. However this incurs an increase in the code size.

The scalability of subword processors is not simply achieved by increasing the machine word size. It can be argued that subword processing is a low cost addition to general-purpose microprocessors to exploit the wide datapaths, however reversing the argument may not be valid. Scaling the performance by doubling the machine word size would require increasing the data path size beyond that required for other general-purpose applications. In addition, the addressing mode restrictions, misalignment, and fixed vector sizes overheads would be more emphasised.

5.3.2 Other Subword-Based Approaches

In an attempt to address the stride-1 restrictions of the subword approach, the MOM architecture [8] investigates combining traditional pipelined vector processing with subword processing. Their proposed architecture relies on having a vector register file where every element contains subwords that are processed in parallel. The addressing mode is extended to stride-n access, where every element is loaded separated by an n-byte gap.

This approach did not consider a more general submatrix addressing mode. It put a constraint on having large register sizes to match the row size of the processed data structure. Moreover, it imposes that constraint on future applications that might need a larger row size.

5.3.3 Torrent-0 Vector Processor

The Torrent-0 (T0) microprocessor [3] is a parallel vector microprocessor. It contains 16 vector registers each containing 32 32-bit registers. There are three functional units and each one is replicated 8 times. 8 vector elements can be processed at the same time. It has been demonstrated that the vector processor out performed a superscalar core and the implementation cost does not exceed that of the superscalar. It has been shown that the

control issue is simpler than superscalar and the bandwidth of register organisation is high and justifies the large register file area.

Subword processing is not used. We believe that subword processing has the potential to share the functional units with non multimedia applications, and thus a mix between a vector and a subword model is better.

5.3.4 Intelligent RAM

The Intelligent RAM (IRAM) project [63] investigates having a vector processor in the memory. The approach exploits the high memory bandwidth of RAM technology and builds a vector processor that utilises that bandwidth. The high instruction bandwidth requirement of multimedia is thus optimised.

The main drawback of this approach is that it is highly optimised for multimedia applications and scalar performance does not benefit. The architecture targets mobile multimedia processing by providing high performance and low power.

5.3.5 Simultaneous Multithreaded Vector

The Simultaneous Multithreaded Vector (SMV) project [18] combines vector, superscalar, and simultaneous multithreading. Simultaneous multithreading [16] is a technique where different contexts (threads) are executed at the same time on the same processor core. This technique is implemented on top of a superscalar machine to increase the utilisation of the hardware resources.

In SMV the rationale behind using a vector architecture is to exploit the data-level parallelism of multimedia applications. Superscalar design exploits ILP and also helps to hide the memory latency. Simultaneous Multithreading helps to increase the utilisation of the hardware resources for the underlying superscalar design.

This architecture relies on central hardware resources and increases the complexity of superscalar design. Vector addressing usually results in streaming data access. Simultaneous multithreading keeps different threads at the same time. This is possibly why the SMV does not have a cache. While simultaneous multithreading increases

throughput it does not guarantee fairness between threads, and that is of particular importance for the real time processing requirements of multimedia.

5.3.6 Stream Processors

Another class of vector processors is the stream processors. An example of this class is the Imagine processor [36]. This processor performs compound operations on data streams, using local registers for intermediate results. The data streams are directly read into stream buffers and processed by parallel functional units. The architecture is highly optimised for streaming data and is not a general-purpose design.

A similar design to the Imagine processor is the Emotion Engine [41] used in PlayStation2. The architecture contains two vector units and a subword parallelism-based processor. The vector units process streams of data via local memories and registers for intermediate results. The architecture is optimised for 3D graphics perspective transformations.

5.4 Multiprocessing

The technology is still following Moore's law, doubling the transistor densities and performance every 18 months. This is expected to continue until the end of this decade. However wire delays are believed to be a huge limiting factor and the discrepancy between processing and memory speed is leading in the direction of simple replicated processing cores on a single-chip. Our group project JAMAICA is investigating a single-chip multiprocessor with multithreading to tackle this problem. The Stanford Hydra project [29] and Sun's MAJC architecture [72] also follow the single-chip multiprocessing approach.

The use of multiprocessing in multimedia is applicable, as on a higher level, multimedia processing appears as single-program-multiple-data (SPMD) model. One procedure is generally applicable to a large data set. For example, the motion estimation procedure in MPEG-2 encode is applied to all macroblocks on a frame. This forms a parallel execution schedule. Thus multiprocessing is likely to support multimedia processing.

5.4.1 MAJC Architecture

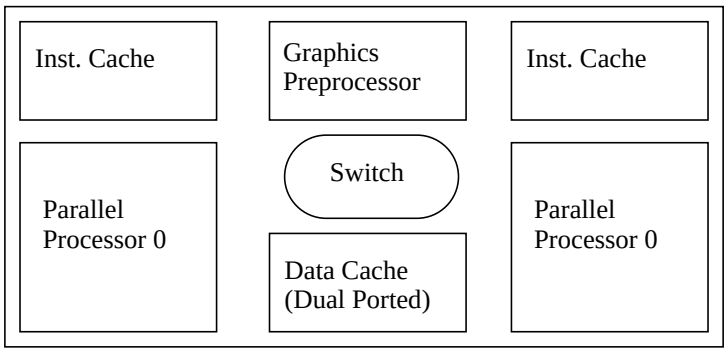

Figure 5.4: MAJC block diagram

Figure 5.4 shows a high-level block diagram for the Microprocessor Architecture for Java Computing (MAJC) processor. The MAJC processor is targeted towards the execution of multimedia benchmarks written in Java. Initial versions include two VLIW processors with a common graphics preprocessor and a data cache. The processor supports thread speculation exploiting the Java features that method boundaries are a natural way to execute methods in parallel as method communication is limited to the return value and heap objects, which makes it easier to track dependencies.

The MAJC processor supports subword processing of packed 16-bit subwords in 32-bit words. The processor also includes a graphics preprocessor that decompresses polygon information in 3D graphics and sends it to the CPU. The processor also supports multithreading by providing a mechanism for fast context switching.

5.4.2 PLASMA Architecture

Another multiprocessing approach based on VLIW is the Partitioned VLiw MultithreAded Superpipelined Multiprocessor Architecture (PLASMA) architecture [52]. The architecture combines multiprocessing, VLIW, simultaneous multithreading, and vector processing, exploiting parallelism at many levels. However, the architecture is optimised for multimedia processing. Stream buffers are used instead

of caches, and DMA units are used to manage reading and writing streams. No subword parallelism is used.

5.4.3 DSP

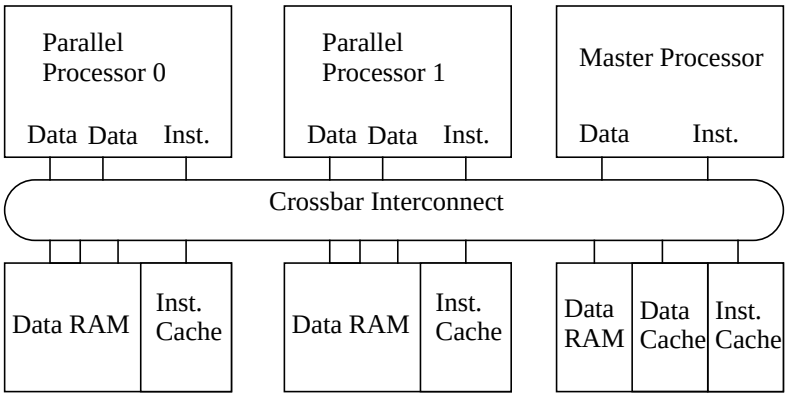

Figure 5.5: Parallel DSP

Multiprocessing is also used in many DSP processors. The Texas Instruments' TMS320C82 uses a single-chip multiprocessor with two DSP processors and one master processor acting as controller. Figure 5.5 shows an overview of the processor. The processor and local on-chip memory are connected by a crossbar network. Each processor has traditional DSP functional units such as multiply accumulate, two memory read ports and one memory write port. Up to five instructions can be executed in parallel.

The DSP is well suited to multimedia processing as most multimedia kernels are typical DSP kernels (such as a finite-impulse response filter) and have predicable execution, such as using memory instead of caches. However, caches are getting more attention in DSP processors. With the increasing complexity of DSP applications, large on-chip memory is required. A large memory is inherently slower than a small memory. This motivated the Texas Instruments' TMS320C6211 [2] to incorporate two levels of caches in the chip (configurable L2 cache).

5.5 Reconfigurable Logic

A different approach is to use reconfigurable logic to synthesise hardware resources optimised to a current kernel. This approach is followed in the PipeRench processor [25] and many others. It has the advantage of decreasing the instruction bandwidth requirement by constructing a parallel implementation of a kernel. The construction is done using a reconfigurable interconnection network. This also helps handling the data alignment better than the subword processors.

However reconfigurable logic is not currently accepted in high performance processor design due to many factors; configuration time ranges from hundreds of microseconds to hundreds of milliseconds. There are also hard constraints on the size of the kernel and the granularity of the logic. PipeRench attempts to solve some of the problems. It overlaps execution with configuration and increases the granularity size.

5.6 Memory Interface

In the previous sections, the architectures varied in the way they organise memory. For vector processors, traditionally, the memory interface is a very expensive part of the architecture. It is required to sustain high data read and write rates. Techniques like memory interleaving were used to achieve that performance. With the growing memory processor speed gap, directly accessing memory is not a good idea. Memory hierarchies are used exploiting the fact that smaller and closer memory is fast and that there is locality in memory access that can be captured in this way.

Multimedia applications are stream oriented. This emphasises the spatial locality rather than temporal locality. Thus a simple streaming buffer may outperform a cache design. That was the rationale used in the IRAM project. However, since the processor is not just executing multimedia workloads, the cache may still be useful and there might be ways to optimise it. In this section we review the literature for techniques to exploit the streaming nature of multimedia applications.

5.6.1 Software Prefetching

Software prefetching involves having a prefetch instruction that instructs the CPU to load from a given address before the data is needed by the program. While this technique accurately specifies the future referenced data, it has two main problems [78, 79]. The first is specifying where to put the prefetch instruction (prefetch scheduling). Scheduling the instruction early makes the data arrive early, which might override other important data in the cache, this is referred to as cache pollution. Scheduling it late, will possibly delay the arrival of the data making the prefetch useless. The position depends on the particular hardware penalties which cannot be determined at compile time.

The second problem is the penalty of executing the instruction itself and generally code transformations are required to avoid unnecessary prefetch instructions, which increase the code size. The prefetch instructions also increase the register requirements (register pressure). This might increase the potential for having spill and fill code that otherwise is not required.

Most of the current multimedia architectures have prefetch instructions. It has been shown that for numeric and regular applications, the programmer can achieve large speedups with prefetching. However, for modifying large amount of code, an automatic compiler would be required which is not as successful as a programmer.

5.6.2 Hardware Prefetching

Hardware prefetching techniques rely on the hardware to prefetch the data. No explicit prefetch instruction is required or executed by the hardware. However, the accuracy of the software approach is better as the required data is precisely specified by the prefetch instruction. The hardware approach either relies on simple prefetch techniques that fetch a fixed pattern of data references or more sophisticated techniques that dynamically approximate memory access patterns. The main advantage of the hardware approach is that there is no penalty executing extra instructions and it does not need the code to be modified for each particular hardware implementation.

Cache Prefetching

Caches employ prefetching on a small scale; on a cache miss, a cache block is fetched. This is done to exploit the spatial locality of reference. Smith [68] has proposed a sequential prefetch technique to enhance the performance of streaming data. The first technique is called prefetch-on-miss; on a cache miss the next sequential cache block is prefetched. The other technique is called prefetch tagging; an extra bit is associated with every cache block to indicate that the block is demand-fetched or prefetched and referenced for the first time. In both cases the next cache line is prefetched. The latter technique achieved 50% to 90% performance improvement and double the performance of the sequential prefetch technique.

Jouppi [34] has proposed a streaming buffer for streaming data with larger spatial locality. This technique overcomes the disadvantage of the previous technique where prefetching is scheduled too close to the required computation and thus no computation and memory overlap occurs. A stream buffer is a queue of sequential cache blocks. On a cache miss the block is fetched and the next sequential ones (up to a fixed number) are prefetched and stored in the stream buffer. On a subsequent cache miss, the data are fetched from the head of the stream buffer queue. There are many modifications to this technique such as having many stream buffers and employing a least-recently-used (LRU) replacement policy among them, however these techniques rely on sequential data access, whereas multimedia employs a submatrix access.

A more relevant technique is proposed by Fu and Patel [23]. They developed a vector cache for vector processors. On a cache miss, if the reference is a scalar or short stride vector, a number of consecutive cache blocks are prefetched. Otherwise, a number of blocks will be prefetched separated by a vector stride. Their architecture utilises the information in the vector instruction and assumes a sequential access for a scalar one.

2D Spatial Locality

It is tempting to view the way multimedia applications access data as a form of 2D spatial locality. Reviewing the literature we found two references to this concept. The existence of 2D spatial locality is hinted by Kuroda and Nishitani [42], though no design was

suggested. Cucchiara et al. [11] have proposed a cache prefetching technique for caching this new type of locality. They developed an adaptive technique that either prefetches stride-1 or stride-n blocks (where n is the width of the image) in both directions, depending on the difference between the last two misses. They simplified this technique later in [12]. Blocks are prefetched on a cache miss and the stride information is kept in a hardware table.

We believe that this form of spatial locality exists for multimedia applications and since MPEG-2 encode and decode have regular data accesses with blocks containing rows that fit entirely in a cache block. We believe a simple sequential prefetch technique using fixed stride information will exploit the 2D spatial locality found in the MPEG-2 applications. This will be investigated in more detail in Chapter 7.

5.7 Summary

Our research project considers a single-chip multiprocessor with multithreading. Multithreading is used to hide the memory latency. The JAMAICA philosophy is to exploit fast on-chip interconnect to support light-weight threading. Our aim is to provide multimedia support for such a system.

In this chapter we surveyed the literature on supporting multimedia. Most of the approaches rely on three different design spaces; vector processing, superscalar, and multiprocessing. The recent interest in vector processing stems from the data-parallel nature of multimedia programs and the wide datapaths in current processor technology. This led to the adoption of a simple vector that fits within a register. We believe that combining traditional vector processing techniques with subword processing is likely to have a benefit. We believe such a mix tackles subword limitations and we will investigate that design and a similar one (MOM) in the literature in Chapter 6. Superscalar design and multiprocessing are used in many proposed and existing architectures. These help exploit the instruction and thread-level parallelism. ILP will be useful for achieving more performance however extra complexity is incurred and using vector processing is more emphasised. The underlying architecture, JAMAICA, combines multiprocessing and multithreading and thus memory and light-weight threading parallelism are more

emphasised. Thus we focus on vector processing in that context. Superscalar techniques are orthogonal to our design and can be used in future work.

The other aspect is memory design. The poor temporal locality is likely to affect the cache performance and data prefetch is thus inevitable. We believe that a two dimensional cache locality is abundant in multimedia and we seek to exploit this in Chapter 7.

Chapter 6: Instruction Set Design for Multimedia

The previous two chapters have examined the characteristics of multimedia workloads and architectural ideas from the literature to support them. In this and subsequent chapters we will investigate novel ideas to exploit those characteristics from the instruction set and cache points of view.

This chapter presents the design of the 2d-vector instruction set. This instruction set is based on utilising subword parallelism found in current multimedia enhanced microprocessors and combining it with traditional vector processing techniques. The design alternatives are modelled and analytically evaluated using the MPEG-2 video encode and decode kernels. Important design parameters such as the number and size of vector registers are determined.

6.1 The 2d-vector Instruction Set Architecture

The 2d-vector instruction set is intended to be an extension to the JAMAICA processor instruction set [85]. It contains vector operations for accelerating multimedia processing. However, scalar operations and branching instructions are not defined and the JAMAICA instructions are used instead.

The instruction set is based on vector registers instead of direct memory access. There are two reasons for choosing this. The first is that memory latency is large and vector registers give the programmer/compiler control over managing the register storage-level, enabling the reuse of register values and avoiding memory access. The second is that the underlying architecture is register-based and many existing architectural features can be reused, such as instruction formats.

Floating-point arithmetic is not supported by the instruction set because it has been shown that multimedia workloads are mostly integer applications. However, if floating-point arithmetic is needed for future multimedia applications (possibly in future 3D

applications) it will not complicate the design, as vector processing was demonstrated to be efficient in floating-point processing in the past [71]. This is mainly due to the simple pipeline control and the operation latency hiding due to repeated operations.

The 2d-vector is a 32-bit instruction set. The first reason behind this is that multimedia data types are small mostly requiring 8- and 16-bit word sizes. A higher word size will benefit subword processing. However there is a trade-off between increasing the word size and having alignment overheads. This issue will be studied in the evaluation section (Section 6.3). The other reason is the underlying architecture is 32-bit and reusing the datapaths is sought.

In this section we will describe and discuss the 2d-vector instruction set but not rigorously define it. Appendix B gives an exact definition. The instruction set is based on that found in the Motorola's AltiVec [53].

6.1.1 Register Aspects

Subword parallelism has the potential to exploit the data-level parallelism in multimedia applications. However, it is limited by the machine word size. Increasing the machine word size is a costly operation as it will affect the datapath widths inside the processor and also has a limited efficiency if the vectors are of variable lengths. Traditional pipelined vector processors do not have these limitations. However their parallelism is limited to pipelining.

The 2d-vector instruction set combines the benefits of these two techniques. The instruction set defines 8 vector registers, each of which has 8 32-bit registers (vector elements). Each vector element can be viewed as a packed register of 8- or 16-bit subwords (The choice of vector register number and sizes will be studied in more detail in Section 6.2). The vector length is variable and determined by a common vector length register ('VL').

Vector operations are performed by 3-operand vector instructions as follows:

$$V_d \leftarrow V_{s1} op V_{s2} \tag{2}$$

The operation is performed on every vector element; The operation is first done on the first element of V_{s1} with the first element of V_{s2} and the result is stored in the first element of V_d, and the same is repeated for all other elements.

V_{s2} can be a scalar register. This is helpful for multiplying a vector by a constant, for example. Another operation that uses a scalar register is the reduction operation, where a vector is reduced to a scalar value. It is implemented by specifying both V_d and V_{s2} registers to be the same scalar register. Previously the scalar register needs to be initialised.

The operations available are:

- Integer addition and subtraction (with and without saturation arithmetic).
- Integer multiplication.
- Integer multiply and accumulate.
- Bitwise boolean AND, OR, and XOR.
- Logical shift left and right, and arithmetic shift right.
- Integer signed and unsigned comparison.
- Pack and unpack operations.

Integer and shift instructions operate on subword operands. So there are three versions for every instruction; 8-bit (byte), 16-bit (short), and 32-bit (word). Integer comparisons are implemented by comparing two operands and storing either zero or the maximum unsigned value into the destination vector for false and true boolean values respectively. The other way to implement comparisons is to use vector masks. A vector mask is a bit vector holding the outcome of a comparison operation. The vector mask can be used later in another vector instruction to specify which elements to operate on. The reason for choosing the first technique is, as shown in Chapter 4, that comparison is not heavily used (mostly for loop control) and techniques like vector masks are more complex to implement.

Vector pack and unpack instructions are used to convert between subword data types. A word pack operation converts a word into a half word and the vector length will be halved.

Another pack operation is used to convert from short to byte. Overflows are either truncated or saturated. Vector unpack does the opposite operation.

These vector packing and unpacking operations mimic those used in the AltiVec instruction set. However they are more efficient as the vectors are of variable length and the operation will only operate on the used part of a vector register.

6.1.2 Addressing Modes

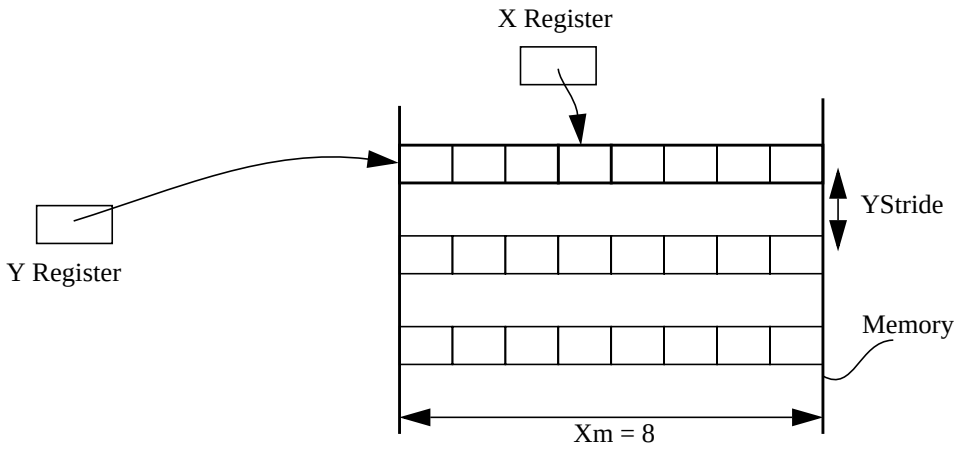

Figure 6.1: Submatrix memory addressing

Multimedia kernels have been shown to operate on data organised in a submatrix (Chapter 4). The 2d-vector instruction set defines a submatrix addressing mode. Data are assumed to be organised in rows with a fixed stride between them. Memory access is done through load and store instructions whose operation is:

$$V_d \leftrightarrow M[X_s + Y_s] \tag{3}$$

Figure 6.1 illustrates the submatrix addressing mode operation. For every vector register, there are two control registers associated with it. The first register is labelled ‘Y’ and is used to hold the base address of the current row. The second register is labelled ‘X’ and is used to hold the word address relative to the start of the row. The row size is determined by a register specified in the memory access instruction. The stride between the base

addresses of the rows is labelled ‘YStride’. The number of elements per row is labelled ‘Xm’. Both YStride and Xm are specified in the memory access instruction.

For a memory load operation, the instruction would be: `vld Vd, Xm, Ystride`

The effective address is calculated by adding X and Y registers. The word at that address is accessed, then X is incremented if the current address is not the last one in the current row. Otherwise, Y is incremented by Ystride and X is reset to zero. The operation is repeated until ‘VL’ elements are loaded.

The reason for keeping Xm and Ystride in the load instruction is that these values are likely to be constant for many vectors and can be reused. X, and Y registers are likely to be different from every vector as every vector will be referring to a different data structure instance with similar dimensions.

This addressing mode is useful in accessing row major data structures. Another addressing mode is defined for column major data structures. This mode is called the transpose mode and essentially the roles of X and Y registers are swapped. In this mode, ‘Xm’ words are accessed with the ‘Ystride’, the base address is then incremented by one and the process is repeated similarly.

Byte (signed and unsigned) load and store instructions are specified as most of the kernels involve reading byte quantities. 16-bit values are usually used for intermediate calculations and not for accessing memory. So we did not implement special instructions for 16-bit operands.

6.1.3 Unaligned Memory Access

Memory alignment is a major concern in the subword aspect of the design. For example for a 32-bit word and 8-bit subwords, every machine word is effectively a vector of 4 byte elements. If it is desired to load a small vector with a base address that does not start at a word boundary, then two loads will be required for loading two consecutive words. Then a combined rotation for the loaded value will be needed.

As word sizes increase, the chance of accessing a vector that starts at unaligned address increases also. Traditional pipelined vector architectures do not have this problem because every element is a word. However, parallel vector architectures suffer from a similar problem which requires a separate aligning network and that is one of the reasons that the pipelined vector systems have an advantage.

The 2d-vector instruction set combines the benefits of both approaches. Two instructions are specified for unaligned vector load and store operations. The repeated nature of vectors is utilised and can save many loads. If all the accesses are sequential then only one extra load will be required.

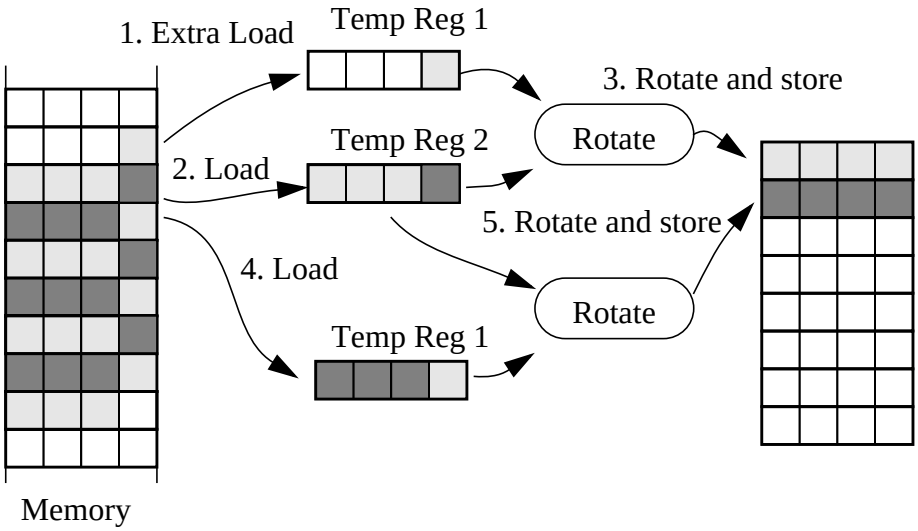

Figure 6.2: Unaligned memory load operation sequence

Figure 6.2 illustrates the unaligned memory load operation for a 2 word vector with byte subwords in each word. Initially, an extra load from the base word is done and stored in a temporary register (Temp Reg 1). Another load is done of the next word and stored in another temporary register (Temp Reg 2). A combined rotate operation is performed on the temporary register to align the word. The aligned word is then stored into the register file. To read the second word, a further load is done from memory and stored into the first temporary register (Temp Reg 1). The initial part of the new value is already in Temp Reg 2, so no extra load is required and the required word is obtained by a combined

rotation as before. Further loads will proceed similarly if the vector words are consecutive. For a stride access, the operation is restarted. So extra loads will be required for every change of row.

6.2 Analytical Performance Modelling

Instruction set design is well understood in the literature. However, combining vector processing with subword processing is new and presents new aspects that need further study. In this section we consider the trade-offs between having a vector and a subword design and analyse their combination. To facilitate such a study, we develop a simple analytical model and use it to examine the multimedia kernels presented in Chapter 4.

The model focuses on the instruction count ignoring the memory effects. The latter will be studied in Chapter 9 where the whole system is defined. The instruction model will give an upper bound on performance and can help us make high-level decisions.

6.2.1 Instruction Performance Model

The total execution time of a program on a machine can be calculated by $I \times CPI \times tc$, where I is the total number of instructions executed, CPI is the average number of cycles per instruction, and tc is the cycle time. I depends on the instruction set architecture and the program, CPI depends also on the instruction set architecture and on the system (CPU and memory) architecture and relative technology speeds between system components, and tc depends on the system architecture and the hardware technology.

Since we are modelling vector instructions, an instruction involves many repeated operations. A better model would be $I \times OPI \times CPO \times tc$, where OPI is the average number of operations per instruction and CPO is the average number of cycles per operation. For this analysis we assume that $CPO = 1$ and $tc = 1$ unit of time. The former assumption is made because all the vector operations are simple. Vector processing does not complicate the control mechanism and we do not expect more than one cycle per operation. The latter assumption is made to isolate the effect of the hardware technology and other aspects of the system architecture from the study. So effectively we calculate the performance in terms of processing cycles.

Java code fragment

```
for(i=0;i<n;i++){
  a[i] = b[i] + 1;
}
```

Assembly code fragment

```
    br L0L002
L0L001:
    s4add  %9, %17, %31  !Address
    ldl    %18, 16(%31)  !Memory
    add    %18, 1, %18   !Compute
    s4add  %9, %11, %31  !Address
    stl    %18, 16(%31)  !Memory
    add    %9, 1, %9     !Loop
L0L002:
    cmplt  %9, %10, %31  !Loop
    bne    %31, L0L001   !Loop
```

Figure 6.3: Instruction breakdown for the scalar instruction set

Multimedia kernels typically apply the same operation over large amounts of data. An example operation is:

$$a_i = b_i + 1 \qquad i = 0, \ldots, n \tag{4}$$

The corresponding Java and assembly code for the loop body are shown in Figure 6.3. Loop initialisation is omitted as usually n is large enough to make it negligible.

There are four different types of instructions in the loop:

- Loop control instructions ($LI_{control}$): These are the instructions that form the loop construct. These are loop index test, conditional branches to exit the loop, and loop index increment. Three instructions are required for such loops.
- Memory access instructions (LI_{memory}): For every array access, there is one associated memory access instruction.
- Address generation instructions ($LI_{address}$): These are the instructions that generate addresses for array addressing. We approximate this number by considering one address calculation per array operation in addition to any array subscript expressions.
- Compute instructions ($LI_{compute}$): ALU instructions excluding address generation.

Due to the simplicity of the algorithm, the quality of the generated code is almost constant and is not sensitive to compiler differences. However there are extra complexities such as register spills and fills and pipelining bubbles that might affect the performance. We ignore these overheads here, but they will be considered, as well as other details, in Chapter 10, where we carry out the simulation study.

The total number of cycles required to execute a single loop iteration would be:

$$CPL = CPL_{control} + CPL_{memory} + CPL_{address} + CPL_{compute} \quad (5)$$

Where for a component t

$$CPL_t = LI_t \cdot CPI_t = LI_t \quad (6)$$

as we assume that each instruction takes one cycle to execute ($CPI_t = 1$). For the above example we will have $CPL = 8$ (3 control, 2 memory, 2 address, and 1 compute).

Java code fragment

```
for(i=0;i<n_8;i++){
  media_ext.vec_load((VectorGeneric)B, Xm, Ystride);
  media_ext.vec_add(B, 1, A);
  media_ext.vec_store((VectorGeneric)A, Xm, Ystride);
}
```

Assembly code fragment

```
  br L1L002
L1L001:
  vld       %v1, %17, %18 !Memory
  vsvaddw   %v1, 1, %v0   !Compute
  vst       %v0, %17, %18 !Memory
  add       %9, 1, %9     !Loop
L1L002:
  cmplt     %9, %10, %31  !Loop
  bne       %31, L1L001   !Loop
```

Figure 6.4: Instruction breakdown for the 2d-vector instruction set

We extended our Java translation system, jtrans (Chapter 3), to generate 2d-vector instructions from the Java code. The 2d-vector instruction functionality is encapsulated into a class library (media_library). For instance, a vector add operation in 2d-vector instructions would be:

`add v1,v2,v3`.

In the Java program it will be

`media_ext.vec_add(v1,v2,v3)`.

Full details of this will be given in Chapter 8 and appendix C.

For the 2d-vector instruction set, Figure 6.4 shows the source code and the corresponding assembly code. A CPL component, for a vector length vl and number of subword per word s can be calculated by:

$$CPL_v = \frac{LI_v \cdot OPI_v \cdot CPO_v}{s \cdot vl} \tag{7}$$

LI_v is the number of vector instructions per loop of component v. OPI_v is the number of vector operations per instruction. CPO is cycles per operation, which is equal to $1/s$. CPL_v is relative to the scalar loop so it is divided by $s \cdot vl$ as a vector loop will effectively execute $s \cdot vl$ scalar loop iterations. Since OPI_v is usually equal to $vl \cdot s$ or a fraction (a_v) of it and CPO_v is $1/s$, Equation 7 can be simplified to:

$$CPL_v = \frac{LI_v \cdot (a_v \cdot vl \cdot s) \cdot (1/s)}{s \cdot vl} = \frac{LI_v \cdot a_v}{s} \tag{8}$$

So for the given example, CPL can be calculated by:

$$CPL = \frac{LI_{control}}{s \cdot vl} + \frac{LI_{vmemory}}{s} + \frac{LI_{vcompute}}{s} \tag{9}$$

Where $LI_{vmemory}$ is the number of vector memory load instructions per loop, and $LI_{vcompute}$ is the number of computation vector instructions per loop. Vector instructions correspond directly to scalar instructions. So the used 2d-vector instructions have scaled all the loop instructions by $1/s$ and further decreased the loop control $LI_{control}$ by vl as the latter is a scalar operation and executed once ($a_v = 1/vl$). For scalar instructions the corresponding CPL is:

$$CPL_{scalarinstructions} = \frac{LI_{scalarinstructions}}{s \cdot vl} \tag{10}$$

For the above example vl is 8, s is 1 and $LI_{vmemory}$ is 2 and $LI_{vcompute}$ is 2 and thus $CPL = 4.375$.

There are other components that are specific to 2d-vector. The first one is the unaligned and aligned memory accesses. Unaligned accesses incur additional memory accesses per row. So the CPL component for unaligned memory accesses, assuming $a_v = 1$, would be:

$$\begin{aligned} CPL_{unalignedmemory} &= \left(LI_{unalignedmemory} \cdot vl + \frac{LI_{unalignedmemory} \cdot s \cdot vl}{\text{kernel row accesses}} \right) \cdot \frac{1}{s \cdot vl} \\ &= LI_{unalignedmemory} \cdot \left(1 + \frac{s}{\text{kernal row accesses}} \right) \cdot \frac{1}{s} \end{aligned} \tag{11}$$

Similarly if we want to model different addressing modes that generate addresses per row/ column change we have:

$$CPL_{address} = \frac{LI_{address}}{\text{kernel col/row accesses}} \tag{12}$$

Finally, we separate pack and unpack instructions into CPL_{pack} and CPL_{unpack}. These components also include instructions to set the VL register before packing and unpacking.

To assess the importance of addressing mode features, unaligned access, and vector length, we model four instruction sets:

- Scalar: This is the base instruction set. It is based on the JAMAICA instruction set [85] without any multimedia extensions. Moreover we assume that no register spilling is done and relax constraints like number of registers, variables not allocated to registers. So it is effectively the vector instruction set without using special addressing modes and with no subword parallelism ($vl = 1$ and $s = 1$).
- Subword: the scalar instruction set and subword style instructions with no special addressing mode ($vl = 1$).
- Matrix: the scalar instruction set and vector/subword based instruction set with fixed stride access. Similar to the 2d-vector except the address calculations would occur for every change of row/col and no alignment.
- 2d-vector: the scalar instruction set in addition to vector/subword based instructions with a submatrix addressing mode with alignment.

6.2.2 Kernel Analysis

In this section we translate the MPEG-2 video encode and decode kernels (Chapter 4) into the 2d-vector instruction set. We then use the instruction performance model to study the effect of changing the parameters of the code. The parameters for the scalar and subword models are obtained from the scalar kernels. The parameters for the matrix and 2d-vector are obtained from the 2d-vector model.

Initially we examine the vectorised kernels and obtain the parameters for the instruction performance model. An analysis will be given in the next section.

dist1 kernel

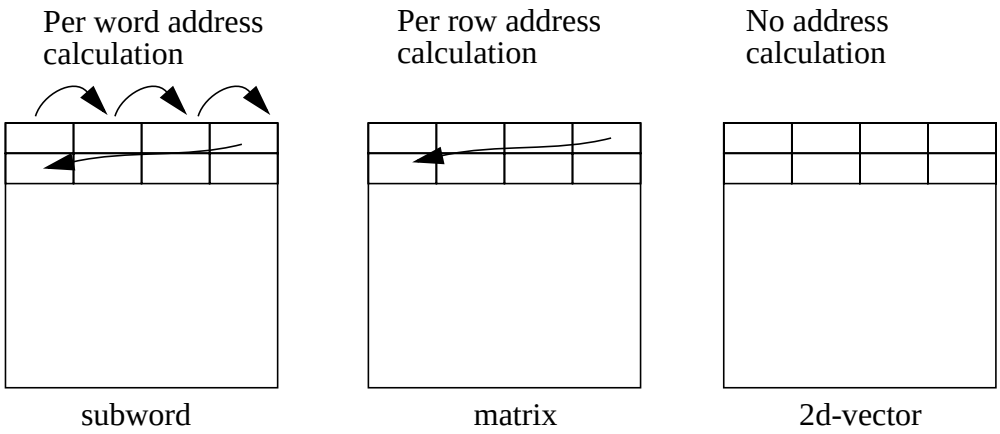

Figure 6.5: The effect of addressing modes in dist1 kernel

This kernel calculates the sum of absolute differences between two 16×16 unsigned byte submartices. Figure 6.5 shows the address calculations of the kernel using different addressing modes. The subword model needs an address calculation instruction after every load. The matrix model, since stride information is supported, needs an address calculation instruction every row change. The 2d-vector model does not need any address calculation as both stride one access (for rows) and stride-n access (for changing rows) are supported.

Figure 6.6 shows the Java code for the vectorised dist1 kernel. An optimisation is used, in the original kernel, to check if the accumulated sum exceeds a certain threshold. This check will introduce increase the penalty of vector initialisation. We ignore this overhead and thus seek for the maximum speedup. However, in later analysis this effect is modelled (Section 9.2).

Table 6.1 shows the parameters for the model. The scalar model uses loop unrolling so in that case the loop overhead (3 for increment, test, and branch) is decreased by a factor of 16. This is needed as one of the 16×16 blocks is not word aligned. The other load is word aligned. The unaligned load requires two memory accesses for subword and matrix models. For 2d-vector, the number of cycles required for unaligned load is calculated from (11) substituting ‘row accesses’ with 16.

The compute part has instructions to compute the absolute difference and the accumulation of the partial sum. The scalar model requires 3 or 4 instructions to calculate the absolute differences (the latter in the case of negative result) and one instruction for the partial sum accumulation. A best case of 4 instructions is assumed. For the 2d-vector model, saturation arithmetic is used to implement the absolute differences; two subtractions and one ‘or’ operation. This is the same for the other models. However another add instruction is required for every loop to accumulate the partial sum.

`media_ext.vec_sum` is a vector reduction operation, where all the vector elements are added together (another instruction is needed to initialise partial sum to zero).

```
for (j=0; j<h; j++){
  media_ext.vec_load_unaligned((VectorGeneric)v1, rM, rLx);
  media_ext.vec_load((VectorGeneric)v2, rM, rLx);

  media_ext.vec_subs(v1, v2, v3);
  media_ext.vec_subs(v2, v1, v2);
  media_ext.vec_or(v2, v3, v1);

  s+=media_ext.vec_sum(v1);
}
```

Figure 6.6: Vectorised dist1 kernel

CPL components	scalar	sub-word	matrix	2d-vector
Loop	$3/16$	$3/16$	$3/(s \cdot vl)$	$3/(s \cdot vl)$
Memory	2			
Memory aligned	n/a	$1/s$	$1/s$	$1/s$
Memory unaligned	n/a	$1 \times 2/s$	$1 \times 2/s$	$1 \times \left(1 + \frac{s}{16}\right)\frac{1}{s}$
Compute	4	$4/s$	$4/s + 2/(s \cdot vl)$	$4/s + 2/(s \cdot vl)$
Pack	n/a	0	0	0
Unpack	n/a	0	0	0
Address	2×2	$2 \times 2/s$	$2 \times 2/16$, 0 for $s = 16$	0
Other	2×2	0	0	0

Table 6.1: Model parameters for dist1 kernel

Address calculations are constant for the matrix model; one load for every row change. This will be zero when the row size is equal to the subword size. Two extra instructions are required in the scalar model for implementing unsigned byte loads for each load. This gives 4 instructions in the ‘other’ category.

The restrictions on s and vl values are $1 \le s \le 16$ and $1 \le s \cdot vl \le 256$.

conv422to444 kernel

This kernel computes a 6 point-FIR filter on the rows of an input frame. Figure 6.7 shows the way the vectors are used to implement the kernel. There are 6 memory loads (vectors v0 to v5) and one memory store (vector vd). The FIR operations occur twice, once on odd numbered columns and again on even numbered columns. Vectors, using the transpose addressing mode, represent 7 columns separated by one column. Every subsequent vector load instruction will load a vector from the next adjacent column. So effectively one iteration loads the odd columns and the next loads the even columns. FIR uses a vector

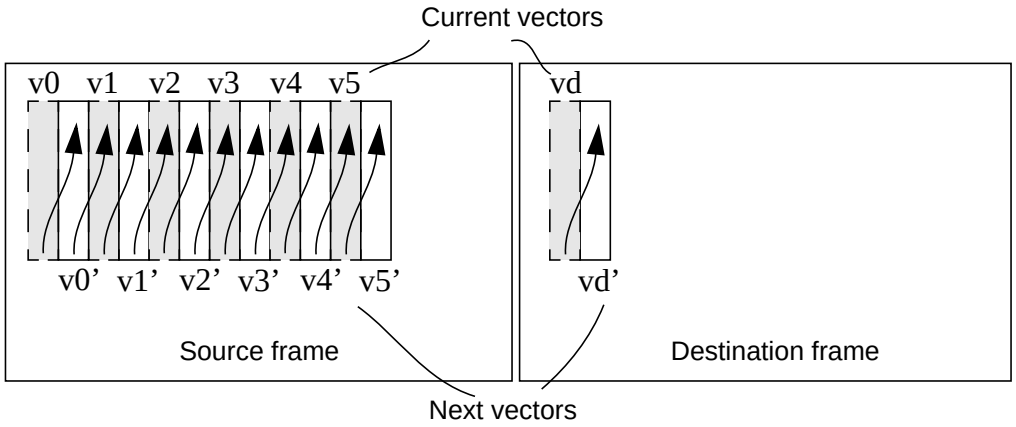

Figure 6.7: Vector layout

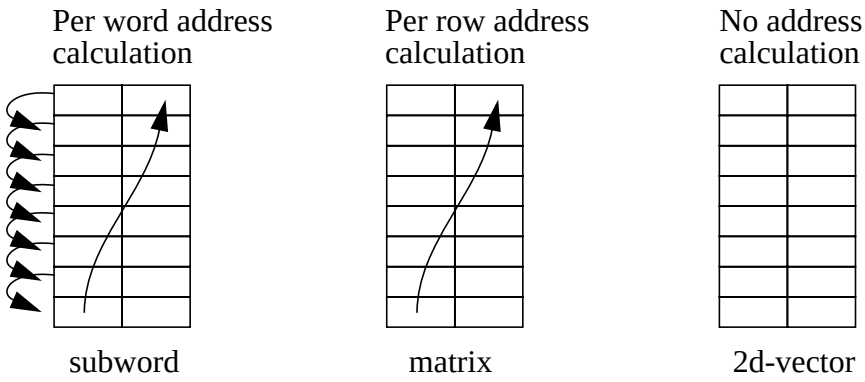

Figure 6.8: The effect of addressing modes in conv422to444 kernel

dot product operation, this is implemented using multiply accumulate instructions. There is also another 1 point-FIR operation used in the kernel, which is effectively a vector copy operation (v3 is copied to vd). Thus two vector stores occur in an iteration.

Figure 6.8 shows the number of address generations required for different addressing modes. The subword model will require a load every word. Moreover subword parallelism will not be used as data are organised in a vertical arrangement, and horizontal vectorisation is not possible as data are not contiguous. The matrix model would require address generation every change of columns. The 2d-vector will not require any address calculation instructions.

CPL components	scalar	sub-word	matrix	2d-vector
Loop	3	3	$3/vl$	$3/vl$
Memory	9			
Memory aligned	n/a	8	8	8
Memory unaligned	n/a	0	0	0
Compute	11	11	9	9
Pack	n/a	0	$1 \times \frac{1}{2} + 1 \times \frac{1}{4}$	$1 \times \frac{1}{2} + 1 \times \frac{1}{4}$
Unpack	n/a	0	$1 \times \frac{1}{2} + 1 \times 1$	$1 \times \frac{1}{2} + 1 \times 1$
Address	9×3	9×3	$9 \times 3/vl$	0
Other	6×2	0	0	0

Table 6.2: Model parameters for conv422to444 kernel

The Java vectorised code is shown in Figure 6.9. Table 6.2 shows the model parameters. There are 3 loop overhead instructions (increment, test, and branch). There are 6 loads and 2 store instructions per loop for all the models and all memory accesses are byte aligned. There is an extra load for the scalar model (for the 1-point FIR). There are 9 compute instructions (1 and, 5 mac, 1 mul, 1 add, and 1 shr). The scalar and subword models require 8 (3 mul, 3add,1 shr, and 1 add) and another 3 for clipping. Clipping is done using an array lookup operation (CLIP[clipoffset + data]) this will cost 2 additions for address generation, and 1 memory load. The clipping is implemented by vector pack and unpack instructions for the matrix and 2d-vector models as it is cheaper. The cycles are calculated using (8).

Address generation is 3 instructions per memory access and since we have 9 memory accesses then we have 27 instructions. This will be the same for the subword model. We have 8 memory accesses for matrix. The address cycles for the matrix model are calculated from (12) substituting col or row access with vl. Two extra instructions are required in the scalar model for implementing unsigned byte loads for each load. This gives 12 instructions in the ‘other’ category.

```
media_ext.SetTransposeMode();

for (i=2; i<w-4; i++){

  media_ext.vec_unsigned_byte_load(vSrc1, vl, w);
  media_ext.vec_unsigned_byte_load(vSrc2, vl, w);
  media_ext.vec_unsigned_byte_load(vSrc3, vl, w);
  media_ext.vec_unsigned_byte_load(vSrc4, vl, w);
  media_ext.vec_unsigned_byte_load(vSrc5, vl, w);
  media_ext.vec_unsigned_byte_load(vSrc6, vl, w);

  media_ext.vec_and( (VectorSignedInt)vSrc3,  (VectorSignedInt)vSrc3,
(VectorSignedInt) vDst1);

  media_ext.vec_byte_store(vDst1, vl, w2);

  media_ext.vec_mul((VectorSignedInt)vSrc1,C1,(VectorSignedInt)vDst2);
  media_ext.vec_mac((VectorSignedInt)vSrc2,C2,(VectorSignedInt)vDst2);
  media_ext.vec_mac((VectorSignedInt)vSrc3,C3,(VectorSignedInt)vDst2);
  media_ext.vec_mac((VectorSignedInt)vSrc4,C3,(VectorSignedInt)vDst2);
  media_ext.vec_mac((VectorSignedInt)vSrc5,C2,(VectorSignedInt)vDst2);
  media_ext.vec_mac((VectorSignedInt)vSrc6,C1,(VectorSignedInt)vDst2);

  media_ext.vec_add((VectorSignedInt)vDst2,128,(VectorSignedInt)vDst2);
  media_ext.vec_shr((VectorSignedInt)vDst2,8,(VectorSignedInt)vDst2);
  media_ext.vec_pack((VectorSignedInt)vDst2,(VectorSignedInt)vDst2,
           (VectorUnsignedShort)vSrc1);
  media_ext.vec_pack((VectorUnsignedShort)vSrc1,
           (VectorUnsignedShort)vSrc1, (VectorUnsignedByte)vDst2);
  media_ext.vec_unpack((VectorUnsignedByte)vDst2,
           (VectorUnsignedShort)vSrc1);
  media_ext.vec_unpack((VectorUnsignedShort) vSrc1,
           (VectorUnsignedInt) vDst1);
  media_ext.vec_byte_store(vDst1, vl, w2);
}
```

Figure 6.9: Vectorised con422to444 kernel

The maximum value of *vl* is the frame height.

conv420to422 kernel

This kernel is similar to conv422to444 except that the calculations are done on columns and four different FIR filters are also used. We will concentrate on one FIR as it will give us an indication of the effect on performance as the others are similar. Address generation requires 4 instructions (compared to 3 in the pervious kernel). This is because the column addressing incurs an extra multiplication. Also the compute component is reduced by one because the first null FIR is not performed (a simple copy).

CPL components	scalar	subword	matrix	2d-vector
Loop	3	$3/s$	$3/(s \cdot vl)$	$3/(s \cdot vl)$
Memory	7			
Memory aligned	n/a	$7/s$ for $s = 1$ $6 \times 3 + 1$ for $s > 1$	$7/s$	$7/s$
Memory unaligned	n/a	0	0	0
Compute	11	$8/s$	$8/s$	$8/s$
Pack	n/a	$2/s$	$(1/2 + 1/4)/s$	$(1/2 + 1/4)/s$
Unpack	n/a	0	0, $6 \times (1/2 + 1/4)/s + 6 \times 1/(s \cdot vl)$ for $s > 1$	0, $6 \times (1/2 + 1/4)/s + 6 \times 1/(s \cdot vl)$ for $s > 1$
Address	7×4	$7 \times 4/s$	0	0
Other	6×2	0	0	0

Table 6.3: Model parameters for conv420to422 kernel

For $s > 1$ vector load and store needs two unpack instructions and one scalar instruction to set the VL register.

```
for (j=8; j<h-8;j+=2){

  media_ext.SetVectorLength(8);
  media_ext.vec_unsigned_byte_load(vSrc1, vl, w2);
  media_ext.vec_unsigned_byte_load(vSrc2, vl, w2);
  media_ext.vec_unsigned_byte_load(vSrc3, vl, w2);
  media_ext.vec_unsigned_byte_load(vSrc4, vl, w2);
  media_ext.vec_unsigned_byte_load(vSrc5, vl, w2);
  media_ext.vec_unsigned_byte_load(vSrc6, vl, w2);

  media_ext.vec_mul(vSrc1, C1, (VectorSignedInt)vDst);
  media_ext.vec_mac(vSrc2, C2, (VectorSignedInt)vDst);
  media_ext.vec_mac(vSrc3, C3, (VectorSignedInt)vDst);
  media_ext.vec_mac(vSrc4, C4, (VectorSignedInt)vDst);
  media_ext.vec_mac(vSrc5, C5, (VectorSignedInt)vDst);
  media_ext.vec_mac(vSrc6, C6, (VectorSignedInt)vDst);


  media_ext.vec_add((VectorSignedInt)vDst, 128, (VectorSignedInt)vDst);

  media_ext.vec_shr((VectorSignedInt)vDst, 8, (VectorSignedInt)vDst);

  media_ext.vec_pack((VectorSignedInt)vDst, (VectorSignedInt)vDst,
(VectorUnsignedShort)vTmp);

  media_ext.vec_pack((VectorUnsignedShort)vTmp,
  (VectorUnsignedShort)vTmp,(VectorUnsignedByte)vDst);

  media_ext.vec_store(vDst, 2, w4);
}
```

Figure 6.10: Vectorised conv420to422 kernel

The maximum value for vl is frame height and the maximum value for s is the frame width.

component_predict_copy kernel

This is a simple kernel. A 16×16 byte block is copied to another 16×16 byte block. The 2d-vector Java source code is given in Figure 6.11. An extra ‘and’ vector operation

```
for(j=0;j<tmp;j++){
 media_ext.vec_load_unaligned(v1, tmp1, lx2_r);
 media_ext.vec_and((VectorUnsignedInt)v1, (VectorUnsignedInt)v1,
 (VectorUnsignedInt)v2);

 media_ext.vec_store(v2, tmp1, lx2_r);
}
```

Figure 6.11: Vectorised form_comp_pred_copy kernel

CPL components	scalar	sub-word	matrix	2d-vector
Loop	$3/16$	$3/16$	$3/(s \cdot vl)$	$3/(s \cdot vl)$
Memory	2			
Memory aligned	n/a	$1/s$	$1/s$	$1/s$
Memory unaligned	n/a	$2/s$	$2/s$	$1 \times (1 + s/16)/s$
Compute	0	0	$1/s$	$1/s$
Pack	n/a	0	0	0
Unpack	n/a	0	0	0
Address	2×2	$2 \times 2/s$	$2 \times 2/16$, 0 for $s = 16$	0
Other	0	0	0	0

Table 6.4: Model parameters for form_comp_pred_copy kernel

is used to move the values across two vector registers. Changing the X and Y registers of the vector register (using one vector register) will be more expensive. This extra instruction will not be needed in the scalar and subword models.

The parameters are shown in table 6.4. Since the data type is byte, the maximum value of s is 16. The restrictions on s and vl values are $1 \le s \le 16$ and $1 \le s \cdot vl \le 256$.

form_comp_pred_av kernel

```
 for(j=0;j<tmp;j++){
  media_ext.SetVectorLength(4);
  media_ext.vec_load_unaligned(v1, tmp1, lx2_r);
  media_ext.vec_load_unaligned(v2, tmp1, lx2_r);
  media_ext.vec_load_unaligned(v3, tmp1, lx2_r);
  media_ext.vec_load_unaligned(v4, tmp1, lx2_r);

  media_ext.vec_unpack((VectorUnsignedByte)v4,
  (VectorUnsignedShort)v5);
  media_ext.SetVectorLength(4);
  media_ext.vec_unpack((VectorUnsignedByte)v3,
  (VectorUnsignedShort)v4);
  media_ext.SetVectorLength(4);
  media_ext.vec_unpack((VectorUnsignedByte)v2,
  (VectorUnsignedShort)v3);
  media_ext.SetVectorLength(4);
  media_ext.vec_unpack((VectorUnsignedByte)v1,
  (VectorUnsignedShort)v2);
  media_ext.vec_add((VectorSignedShort)v2, (VectorSignedShort)v3,
  (VectorSignedShort)v1);
  media_ext.vec_add((VectorSignedShort)v4, (VectorSignedShort)v1,
  (VectorSignedShort)v1);
  media_ext.vec_add((VectorSignedShort)v5, (VectorSignedShort)v1,
  (VectorSignedShort)v1);
  media_ext.vec_add((VectorSignedShort)v1, 2, (VectorSignedShort)v1);
  media_ext.vec_shr((VectorSignedShort)v1, 2, (VectorSignedShort)v1);
  media_ext.vec_pack((VectorUnsignedShort)v1, (VectorUnsignedShort)v1,
  (VectorUnsignedByte)vDst);
  media_ext.vec_store(vDst, tmp1, lx2_r);
}
```

Figure 6.12: Vectorised form_comp_pred_av kernel

The operation of this kernel is similar to the previous kernel in the way data are accessed. However extra computations are required. The kernel calculates the average of four neighbouring points for every element in a 16×16 block and stores the results into a 16×16 block. The elements are of type byte. However, a short type is needed for intermediate calculations. The source code is shown in Figure 6.12. Four vectors are

CPL components	scalar	sub-word	matrix	2d-vector
Loop	$3/16$	$3/16$	$3/(s \cdot vl)$	$3/(s \cdot vl)$
Memory	5			
Memory aligned	n/a	$1/s$	$1/s$	$1/s$
Memory unaligned	n/a	$4 \times 2/s$	$4 \times 2/s$	$4 \times (1 + s/16)/s$
Compute	5	$5/s$	$5/s$	$5/s$
Pack	n/a	$1/s$	$(1/2)/s$	$(1/2)/s$
Unpack	n/a	$4/s$	$4 \times (1 + 1/vl)/s$	$4 \times (1 + 1/vl)/s$
Address	14	14	$4/16$, 0 for $s = 16$	0
Other	4×2	0	0	0

Table 6.5: Model parameters for form_comp_pred_av kernel

defined for every source element used in the average calculations. Pack and unpack instructions are used to extend the sources to 16-bit values for intermediate calculations and to compress them into bytes again.

The parameters are summarised in Table 6.5. Since 16-bit intermediate data types are used, s can read up to 8. The restrictions on s and vl values are $1 \le s \le 8$ and $1 \le s \cdot vl \le 256$.

6.3 Comparing Design Alternatives

In the previous section we have identified the parameters for three instruction set models: subword, matrix, and 2d-vector. These parameters depend on vl and s. In this section we compare the models for the kernels and examine the effect of these parameters.

Figure 6.13 shows the effect of changing the vector length on the 2d-vector model with a 32-bit word length. Vector length reduces the loop overhead and other scalar operations however the loop overhead is the biggest contribution. As seen from the figures, kernels with small loops benefit most from increasing the vector length (dist1, form_comp_pred_cp and form_comp_pred_av). On the other hand, kernels with big

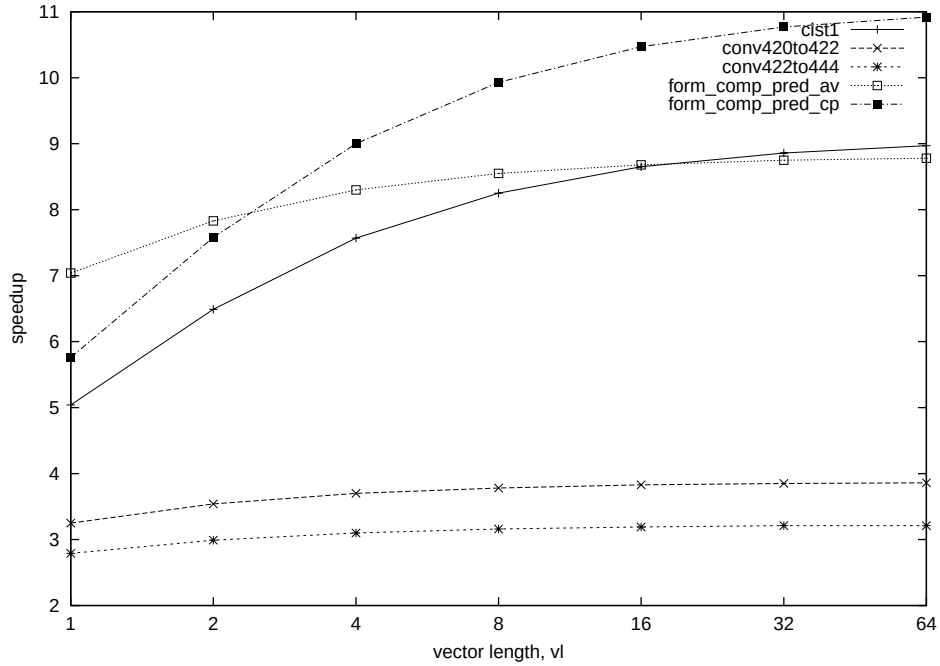

Figure 6.13: The effect of changing the vector length on 2d-vector

loops benefited less (conv420to422 and conv422to444). However, for a vector length of 8 more than 90% of the reduction is achieved with respect to the 64-vector case (the average is 95%). Again, this is because the loop overhead is small (three instructions) and there is not a big start up vector overhead that needs to be hidden. So we justify the choice of 8 for the vector length.

Changing the word size increases the benefits of the vector length as the loop overhead will be relatively larger. However, the effect of changing the vector length will be the same as it will scale proportionally.

Figure 6.14 compares the speedups of the three different models we have analysed. In this comparison we fixed the vector to a maximum of 8 (4 is used for dist1 kernel) and changed the word size. We chose word sizes of 32, 64, and 128-bit. The word size is used as the postfix for the legends.

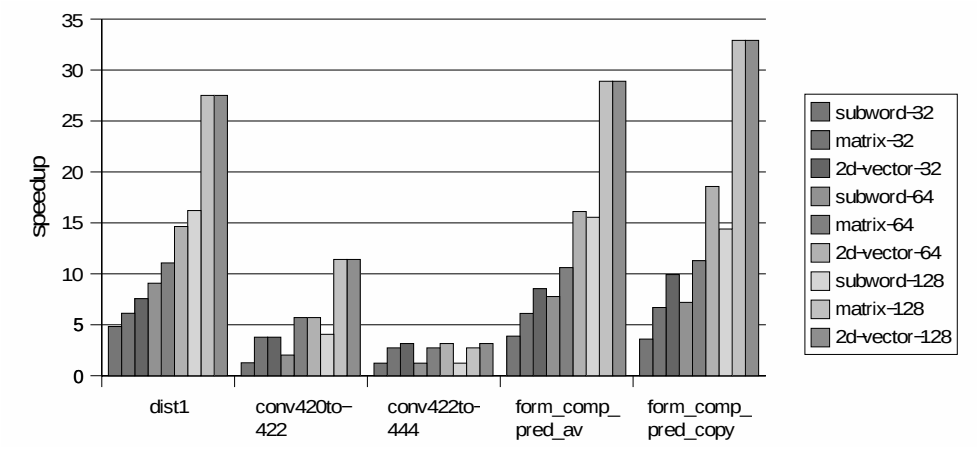

Figure 6.14: The effect of changing the word size on different models

For a 32-bit word size, the 2d-vector achieved 25% better performance on average compared to the closest model (matrix). Performance improvement ranged from zero to 48%. The matrix model achieved 1.98 times the performance of subword.

For a 64-bit word size, the 2d-vector achieved 33% on average compared to the matrix model. This is better than the 32-bit results as the relative address generation penalty is increased. Performance improvement ranged from zero to 65% and 3 of the kernels achieved more than 31% improvement. It is interesting to note that using a 2d-vector instruction set on a 32-bit machine is better than doubling the machine word size and using a subword model (1.54 better on average).

For a 128-bit word size, the overhead for address generation is almost eliminated in most kernels as this word size is enough to accommodate the whole row in a single register. Only conv422to444 showed 15% improvement for the 2d-vector over the matrix model. This is because this is the only kernel where data are accessed column wise and the sub word parallelism is not used. The 2d-vector is again better than the performance of subword with double the word size (1.44 better on average).

Figure 6.15 shows the effect of turning off the alignment hardware. For a 32-bit machine, the advantage is more pronounced. It averaged 10% and ranged from zero to 21%. As the machine word size increase the advantage decreases. For a 64-bit machine, the advantage averaged 6% and ranged from zero to 13%. For 128-bit there was no advantage.

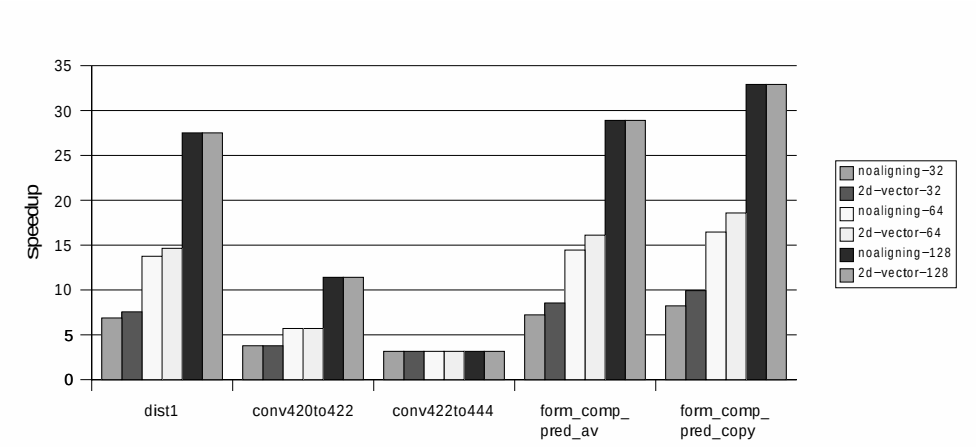

Figure 6.15: The effect of using the alignment hardware

6.4 Summary

In this chapter we have presented the 2d-vector instruction set architecture. The design alternatives are analysed by means of simple analytical models. The models are applied to the MPEG-2 kernels and the choice of the design parameters are justified to be 8 vector elements per vector register with a machine word size of 32-bit. Also 8 vector registers were enough to implement all the kernels with no register spilling. The subword addressing mode and using alignment have shown to have an advantage for the 32-bit word size.

Chapter 7: Cache Design for Multimedia

This chapter focuses on cache design space for multimedia processing. It examines in details the locality of reference of multimedia applications. A two dimensional locality is identified and a simple cache prefetching technique is developed.

7.1 Cache Locality

The principle behind cache operation is exploiting the locality of reference to memory. Programs tend to reference the same data item over a limited period of time as well as referencing nearby data. The former behaviour is called *temporal* locality and the latter is called *spatial* locality.

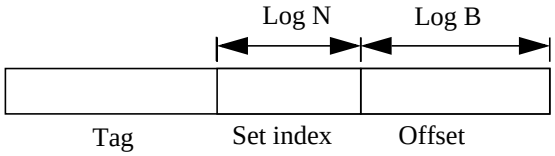

Figure 7.1: Partition of cache search address

The cache is organised as sets of blocks. A cache block is a number of consecutive words. A cache block holds a corresponding memory block. Each set can hold one or more blocks depending on the cache design. Figure 7.1 shows the address partitions of a cache search address. For a cache with block size B (in bytes) and number of sets N, the least $\log B$ bits of the address determine the byte position within a block (offset). The next $\log N$ bits determine the set of the blocks (set index). The remaining bits are used to determine whether the set blocks match the search address (tag). For a direct mapped cache, only one block is associated with a set. Therefore no two blocks can share the same set.

For a fully associative cache, the cache will have one set containing N blocks (for the same cache size). For a cache search all the tags are compared at the same time with the

search tag to determine whether the block is in the cache or not. Searching for a large number of blocks tends to increase the cycle time and decrease the benefits of caching. A compromise solution is to combine the two approaches. This typically makes the cache have 2 or 4 blocks per set.

Increasing the block size increases the chances of capturing spatial locality by fetching more nearby words. Increasing the set associativity increases the chances of capturing the temporal locality as that decreases the chance that a line is evicted.

There are many block replacement policies used in the cache that can affect its operation. The policies commonly used are least-recently-used (LRU), most-frequently-used (MFU) and random replacement. The LRU policy gives the best performance however and can complicate the hardware cost. MFU approximates the LRU by associating a bit with every block and whenever a block is referenced its bit is set to one. At regular time intervals the bits are reset to zero. In choosing which line to replace, a block whose bit is zero is picked. The random replacement is the easiest to implement and has the advantage of not being biased. However, it does not give the same performance as LRU.

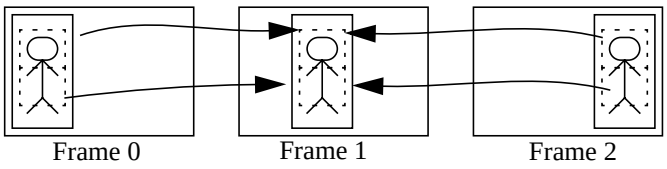

Figure 7.2: 2D spatial locality in motion estimation

Conventional cache design is optimised to exploit the form of locality described above. In multimedia applications, it is more likely that the spatial locality happens in two dimensions. Figure 7.2 illustrates this new kind of locality. MPEG-2 video compression relies on searching a past and a future video frame to reconstruct a current frame (Section 4.3). During the decoding phase a 16×16 block in a typical frame type (bidirectionally predicted) is reconstructed by taking the average of two other blocks in the past and the future frames. It is likely that the blocks next accessed will be nearby in the two dimensional region. Since the frame size is usually large (400 KB) and three

frames are needed, it is unlikely that these regions will be in cache. A cache size of the order of 1 MB is required, which is not practical. Furthermore, higher resolution frames used in the HDTV profiles (high definition TV) would demand even more storage (3-6 MB).

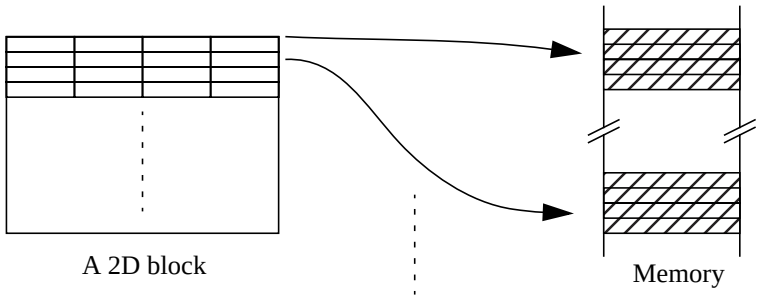

Figure 7.3: Mapping 2D blocks into memory

A block is not stored contiguously in memory. Instead, every row is stored contiguously then there is a gap due to the remainder of the frame width. Figure 7.3 illustrates the block layout in memory. Every row in a block is mapped to consecutive memory locations, then the start address of the next row is mapped to another set of consecutive memory locations but with a stride equal to the frame width.

7.2 Cache Prefetching

Data prefetching can be characterised by considering when the data are prefetched, what is prefetched, and where the prefetched data are stored. As seen in Chapter 5, hardware prefetch techniques rely on some hardware event, such as a cache miss, to initiate a data prefetch. Software prefetch techniques rely on the compiler/programmer to explicitly schedule the prefetch through a prefetch instruction, which also determines what to prefetch. The former is more plausible in the sense that little effort is required from the compiler/programmer. However hardware techniques use various methods to predict the next referenced data and prefetch it, which cannot be as accurate as software prefetching. Prefetched data, for both software and hardware techniques, is generally stored into a separate structure rather than in the cache to decrease cache pollution.

Cache prefetching is a form of data prefetching, where a cache line is prefetched on a cache miss, capturing 1D spatial locality. Exploiting the 2D spatial locality is more naturally viewed as a cache prefetch technique rather than a stand-alone data prefetch technique; prefetching can still be initiated on a cache miss, a set of lines (separated with constant stride) is prefetched and stored in the cache. Polluting the cache this way is less likely as the data is more likely to be used.

The vector instruction set also fits well with exposing the 2D spatial locality as every vector instruction specifies a sequence of references that are going to be used. So we expect that exploiting this type of locality in cache will be beneficial.

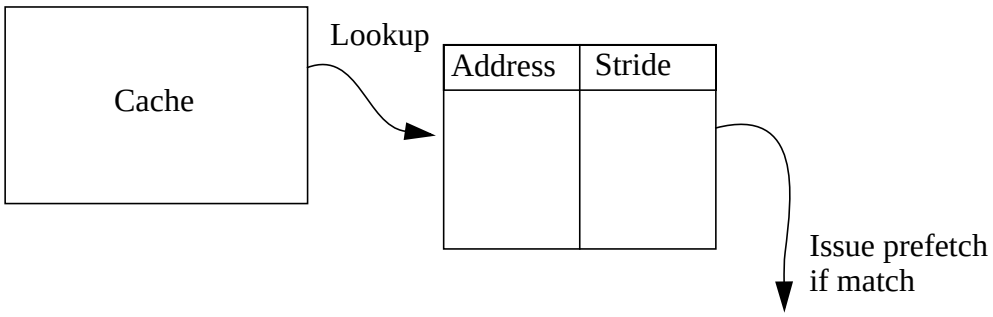

Figure 7.4: Hardware prefetch table

An initial attempt to exploit the 2D spatial locality was investigated by Cucchiara et al. [12]. They proposed a hardware table associated with the cache. The hardware table is filled by the compiler/programmer prior to the program execution and is used to lookup stride information at run-time and dynamically prefetch data on a cache miss using that stride. Up to a certain number of cache lines are prefetched. Figure 7.4 illustrates the operation of the cache.

The technique has shown significant reduction in the number of cache misses (84% for MPEG-2 decode, asymptotically). However an extra lookup per cache reference is incurred that might affect the cache cycle time. We believe a simpler approach is required. We investigate this in the next section.

7.3 A Two Dimensional Cache Design

The design is based on a normal cache and the miss handling is modified to exploit two dimensional access patterns. The cache uses a LRU replacement policy and it is a write-back cache with write allocate. The LRU replacement is likely to decrease the effect of polluting the cache, as if a line is prefetched that is never used, it will not replace a highly used line. The write-back and write allocate features will decrease the cache traffic which is beneficial for the multiprocessor configuration.

On a cache miss (either read or write miss), a cache block is fetched and another b cache blocks are scheduled for prefetching. The blocks are separated by a specified stride. The stride information is either given implicitly in the vector instruction or specified in a control register for scalar misses. The prefetch count b can be set by a hardware control register.

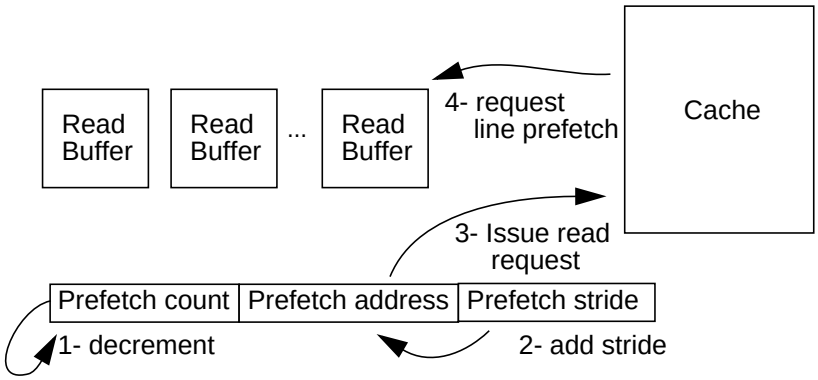

Figure 7.5: Prefetching sequence

Figure 7.5 illustrates how the prefetch is implemented. On a cache miss, the prefetch count is set to b, the miss address is copied into ‘prefetch address’ and stride information either from vector load and store instruction or a control register is copied into ‘prefetch stride’. Every CPU cycle, where the CPU cache interface is idle, the prefetch count is decremented and the prefetch line address is generated by adding ‘Prefetch stride’ to ‘Prefetch address’. The resulting prefetch address is then requested from the cache. The

access proceeds as a normal read cycle except that no data is read from the cache; in the case of a line miss, the request is stored in an appropriate read buffer.

7.4 Evaluation

We have developed a cache simulation using the shade tools [43] and modelled the 2D cache using a 16 KB, 4-way set associative organisation with 32 byte line size. The benchmark programs (written in C) were run on an UltraSPARC-II 333MHz processor and the memory accesses trapped to the cache simulation where the behaviour of the 2D cache was modelled. Our high-level language target is Java, however both C and Java versions have similar data access patterns and we thus opted to use C for this exercise (Chapter 10 verifies this). This enabled us to achieve much higher simulation speed as the programs run in native mode. For other analysis concerning the instruction cycles, we have used Java programs with detailed simulation as described in Chapter 10.

7.4.1 Effect of Prefetching and Set Associativity

Figure 7.6 shows the misses plotted against the prefetch count for 1, 2, and 4-way caches. The data are obtained from running the mpeg2encode program and encoding two 720×480 frames. Figure 7.7 shows the same results for mpeg2decode. The cache associativity is likely to hide prefetches that are not used. The mpeg2encode results show this effect. With no prefetching (prefetch block count equal to zero), the 1-way has about 1.7 times more misses than the 2 and 4-ways. This is not significant in mpeg2decode (less temporal locality). The results for the 2-way and 4-way configuration are almost the same. This indicates that 2-way configuration is enough for MPEG-2 applications. However, generally, the 4-way configuration is beneficial for conventional applications which is still our design target. Therefore from now on, we will focus on the 4-way configuration.

For both applications, as the prefetch count increases the number of misses decrease. There is a slight increase in the 1-way configuration for the mpeg2encode. This is mainly due to polluting the cache.

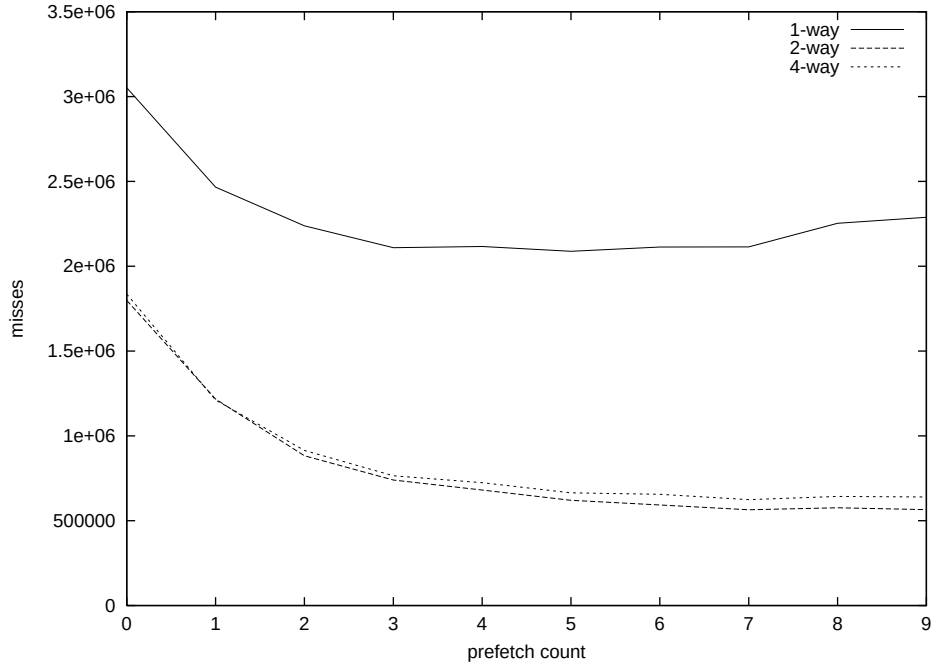

Figure 7.6: mpeg2encode misses

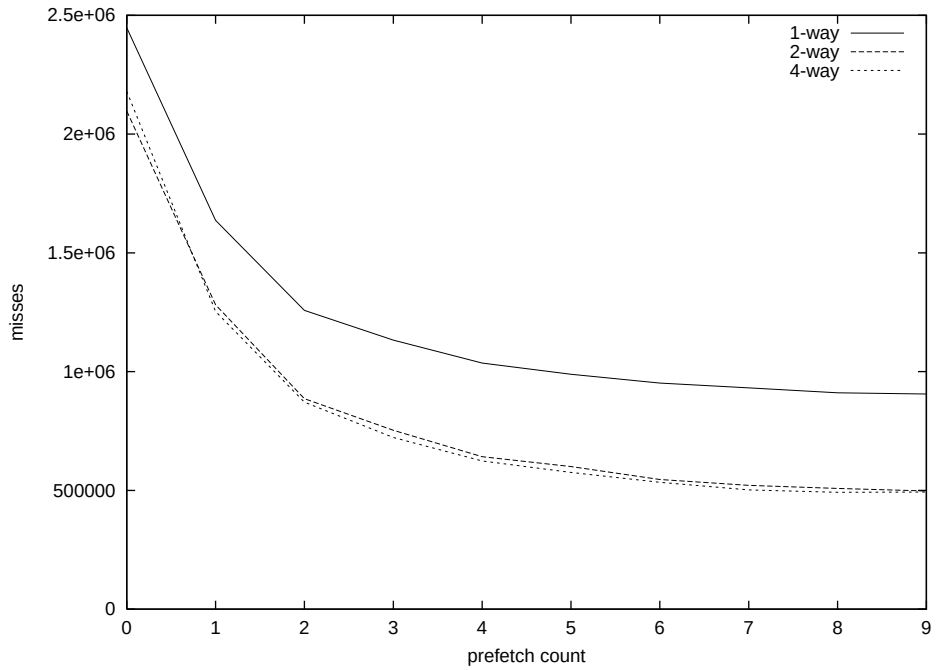

Figure 7.7: mpeg2decode misses

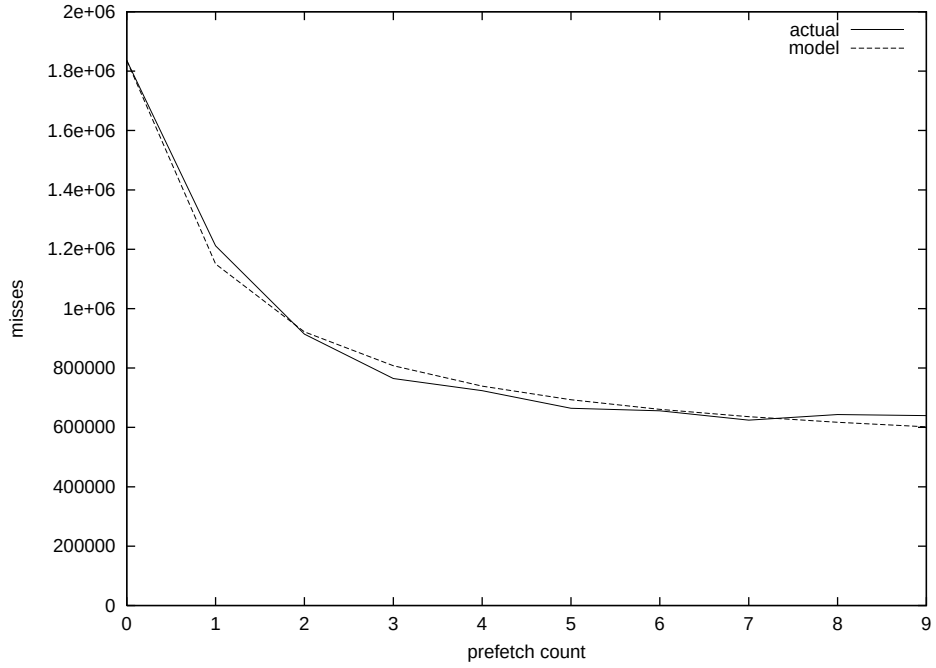

Figure 7.8: Modelling mpeg2encode misses

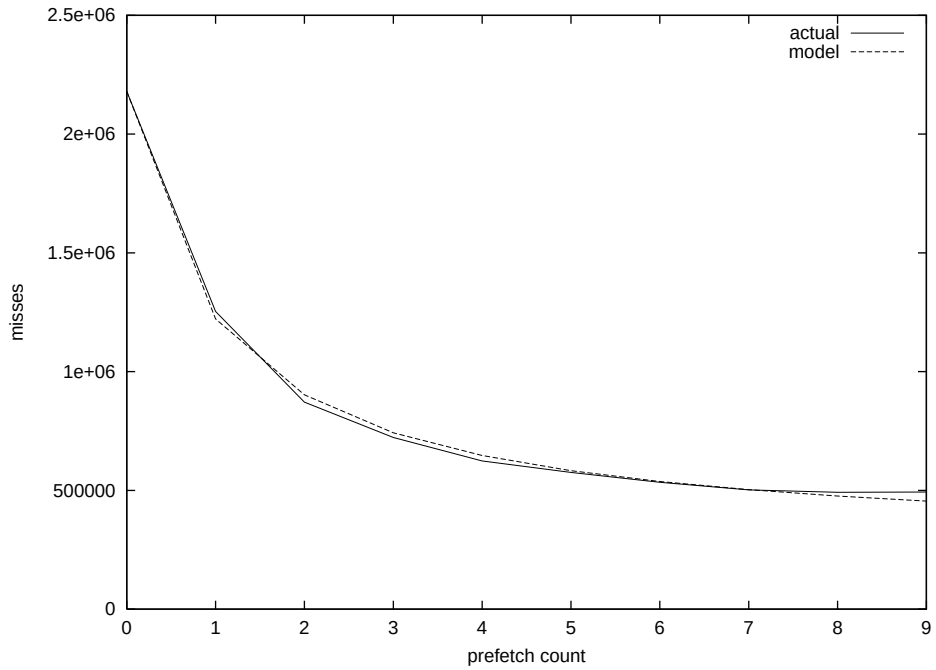

Figure 7.9: Modelling mpeg2decode misses

The behaviour of the cache can be formulated as:

$$misses(b) = \frac{misses(0)}{b+1} \cdot \alpha + misses(0) \cdot (1-\alpha) \quad (13)$$

Where $misses(0)$ is the initial misses (without prefetching), and α is the ratio of the initial misses that have 2D spatial locality (2D access ratio). Since on a cache miss, $b+1$ lines are loaded, b misses can be removed and thus misses are decreased by $b+1$. Fitting Equation 13 to the simulation results, we get α equal to 0.75 and 0.88 for mpeg2encode and mpeg2decode respectively. From the mpeg2decode misses results reported by Cucchiara et al.[12] for the hardware prefetch table approach, we get α equal to 0.84 for a similar cache configuration. Figure 7.8 and Figure 7.9 show the simulation (actual) and misses obtained from the equation (model). The simulated cache behaviour is very close to the model.

Kernel	Miss	Miss ratio	$b = 1$	$b = 2$	$b = 3$
conv422to420	53.33%	9.01%	55.27%	41.85%	29.93%
sub_pred	7.34%	4.27%	77.04%	49.87%	45.16%
read_ppm	4.71%	0.29%	85.64%	87.14%	93.68%
add_pred	3.75%	1.31%	97.57%	50.21%	49.48%
conv444to422	3.59%	0.72%	57.28%	50.62%	46.35%
calcSNR1	3.58%	1.04%	83.43%	80.42%	80.41%
idctrow	3.58%	3.13%	56.16%	48.72%	39.34%
dct_type_estimation	2.60%	1.47%	98.02%	52.26%	51.91%
dist1	2.28%	0.01%	92.75%	59.92%	57.79%
quant_intra	2.07%	1.46%	51.44%	47.44%	32.53%
iquant_intra	2.07%	1.26%	50.51%	48.69%	28.27%
pred_comp	1.80%	1.95%	90.73%	51.76%	48.26%
quant_non_intra	1.52%	0.98%	51.31%	46.04%	32.53%
iquant_non_intra	1.52%	1.51%	50.44%	48.14%	28.57%

Table 7.1: Misses breakdown for mpeg2encode

Kernel	Miss	Miss ratio	$b = 1$	$b = 2$	$b = 3$
variance	1.22%	3.19%	98.09%	49.45%	49.21%
var_sblk	1.20%	1.41%	97.94%	49.53%	49.12%
clearblock	1.06%	2.89%	97.39%	49.80%	49.13%
stats	0.76%	14.46%	99.99%	51.66%	51.76%
Other kernels	2.03%	0.04%	95.00%	92.76%	93.47%

Table 7.1: Misses breakdown for mpeg2encode (Continued)

Table 7.1 and Table 7.2 show the breakdown of the misses among the kernels of mpeg2encode and mpeg2decode respectively. The table lists, for every kernel, the percentage of misses, the cache miss ratio, and the percentage reduction of the misses for prefetch count (b) of 1, 2, and 3. For mpeg2encode, most of the misses (53%) occur in the vertical subsampling filter kernel (conv422to420) where the memory is accessed in column major order. Around 15% of the misses occur in motion prediction kernels (sub_pred, dist1, add_pred, pred_comp) where memory access is not predictable. The remaining 32% of the misses are distributed among other kernels with regular memory access. It is worth noting that prefetching did not increase the number of misses in any kernel and most of the kernels benefited from prefetching.

Kernel	Miss	Miss ratio	$b = 1$	$b = 2$	$b = 3$
conv420to422	88.08%	10.26%	54.81%	36.92%	29.29%
conv422to444	5.97%	0.85%	57.35%	50.68%	46.51%
store_ppm_tga	3.09%	0.88%	98.22%	84.23%	87.72%
form_comp_pred	1.52%	1.98%	90.35%	52.65%	48.77%
Add_Block	0.92%	0.43%	97.17%	50.74%	50.04%
Other kernels	0.42%	0.04%	98.45%	106.19%	123.14%

Table 7.2: Misses breakdown for mpeg2decode

For mpeg2decode (Table 7.2), about 88% of the misses occur in the vertical extrapolation filter kernel (conv420to422). Around 2% of the misses are for the motion compensation kernels (form_comp_pred, Add_Block), where access is unpredictable. All these kernels, in addition to the horizontal extrapolation kernel (conv422to444) benefited from prefetching. However, there were other kernels that showed an increase in misses, but these misses were less than 0.5% of the initial total.

7.4.2 Effect of Prefetching and Cache Size

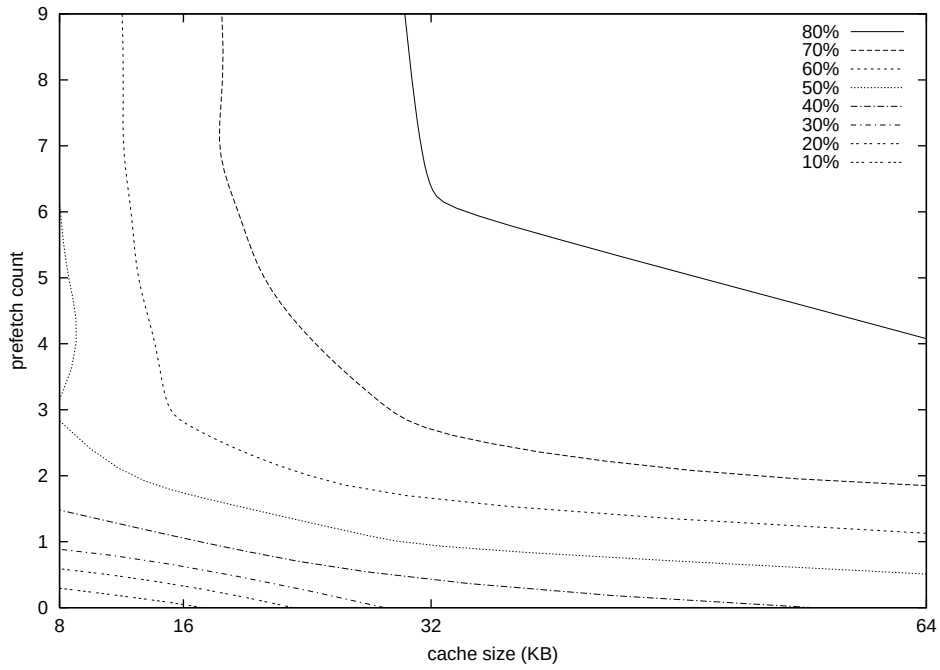

Figure 7.10: Miss contours for mpeg2encode

It is interesting to study the effect of prefetching with increasing cache size. Increasing the cache size with no prefetching reduces the cache capacity misses. However, it does not improve spatial locality. Increasing the cache size and using prefetching has the potential to reduce the effect of cache pollution, at the expense of increasing the cache size. To study this trade-off, we modelled different cache sizes and varied the prefetch count.

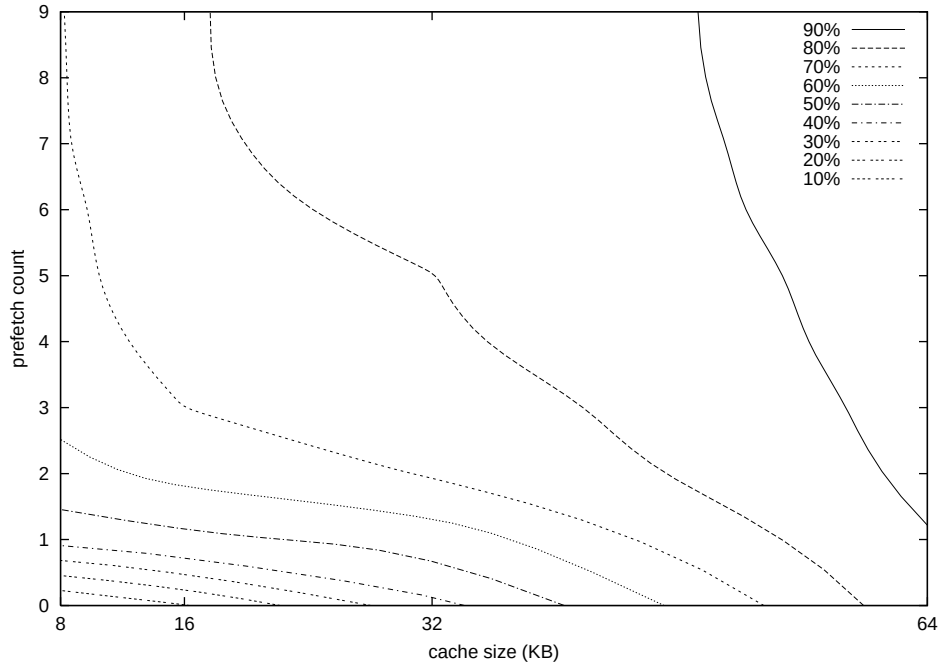

Figure 7.11: Miss contours for mpeg2decode

Figure 7.10 shows the miss contours for mpeg2encode. The x-axis represents the 8, 16, 32, and 64 KB cache sizes. The y-axis represents the prefetch counts. A contour represents a constant number of the metric:

$$\text{miss reduction\%} = \frac{\text{initial misses-current misses}}{\text{initial misses}} \times 100 \qquad (14)$$

The initial misses are obtained for 8 KB cache with no prefetching. Up to 60% reduction of misses can be achieved by the 16 KB cache with a prefetch count of 3 blocks. This is better than a 64 KB cache with prefetch count of one block. The effect of prefetching is greater than increasing the cache size. This is apparent in the flat contours for small prefetch values.

Figure 7.11 shows the miss contours for mpeg2decode. Mpeg2decode benefits more from increasing the cache size than mpeg2encode. This is apparent from the sloping contours for small values of prefetch count. However to remove 60% of the misses, a 16 KB cache with prefetching size of 3 is enough. A 32 KB cache will remove 40-50% with no prefetching.

7.5 Summary

This chapter demonstrates that multimedia video applications possess two dimensional spatial locality that can be extracted by a simple modification to a conventional cache. The modification is based on prefetching a set of cache lines separated by a stride equal to the width of the two dimensional data structure (image in the MPEG-2 workloads). A trace-driven simulation was used to study the 2D cache. An ideal behaviour is modelled analytically and is shown close to the actual results. The model will be used in the third part of the thesis when analysing the parameter space (Chapter 9). Finally the 2D cache matched the performance of a conventional cache with double (and close to quadruple) the cache size.

Part III: System Design and Evaluation

Chapter 8: System Architecture

This chapter describes the a cycle-accurate simulation-level design for the 2d-vector multimedia extensions. The chapter starts by describing the underlying JAMAICA architecture. The 2d-vector multimedia extensions are then described. Finally the software support is described.

8.1 Overview of the JAMAICA System

JAMAICA is a single-chip multiprocessor. The processor is currently defined at a cycle-accurate simulation-level. The number of processors is a simulation parameter, however 8 to 16 processors would be regarded as a typical configuration. The processors follow a shared-memory multiprocessing approach. The processors are connected via a shared bus and a shared L2 cache. Each processor has a private split L1 cache. There is another interconnect for distributing threads among the processors. This interconnect is called the ‘token distribution ring’. Tokens represent idle processor contexts and circulate around the ring. Tokens are managed by a token interface unit in each processor. A thread can query the unit for token availability. On finding a token, a thread can fork a new thread on the idle context. The architecture emphasises fast creation and distribution of threads.

Each processor is multithreaded supporting four hardware contexts. The processor implements a heap-based register-windows mechanism. The mechanism provides fast context switching by storing context data on the windows heap inside the processor.

In this section we will provide details of individual processor design aspects that are relevant to our proposed 2d-vector multimedia extensions. We will focus on the pipeline, caches, and the bus. Further details can be found in [85].

8.1.1 The Pipeline

The processor has a 5-stage pipeline. The pipeline stages are instruction fetch (IF), instruction decode (ID), execute (EX), memory (MEM), and writeback (WB). The IF stage fetches instructions from the instruction cache. It is also responsible for handling

context switching. Context switching is triggered on an instruction cache miss (after retrying), a quantum expiration, or a data miss (as instructed by the memory stage). Further details for handling context switching can be found in [85].

The ID stage identifies the instruction format and fetches the register operands required by the instruction. It also calculates the target address for jump and branch instructions and preforms static branch prediction.

The EX stage preforms integer arithmetic and logical operations. It also generates the target addresses for load/store instructions and resolves conditional branches. All the operations are assumed to execute in one cycle as floating-point arithmetic is not supported.

The memory stage communicates with the data cache for handling load and store instructions and determines a context switch on a cache miss. The pipeline stages are fully bypassed. A pipeline interlock occurs for a memory load operation that is immediately consumed in the EX stage. The pipeline is stalled for one cycle in that case.

The writeback stage commits the instruction by updating the registers and processor context. In the simulation model, the writeback stage is omitted and the memory stage commits the instruction.

8.1.2 Data Cache and the Bus

The JAMAICA processor has a split cache design. Both caches have similar architecture. The data cache is 16 KB 4-way set associative cache with a copy back cache policy. The line size is 32 bytes and the cache replacement policy is pseudo-random replacement. The cache has one port that is shared between the processor and the bus. The bus is 128-bits wide. The bus protocol is split transaction and pipelined. A new bus transaction can start every two bus cycles.

The cache coherence protocol is based on a MOESI cache coherence protocol, further details can be found in [85]. We have not modified the protocol for the multimedia extensions.

8.2 Processor Organisation

The 2d-vector multimedia extensions are incorporated into each JAMAICA processor. Subsequently we will describe the extensions to one processor. Including the extensions with the processor has the advantage of sharing ALU and other processor resources and reusing the same pipeline stages. Even for vector processing, scalar operations are required for overall program control and vector instruction set up.

8.2.1 Vector Pipeline

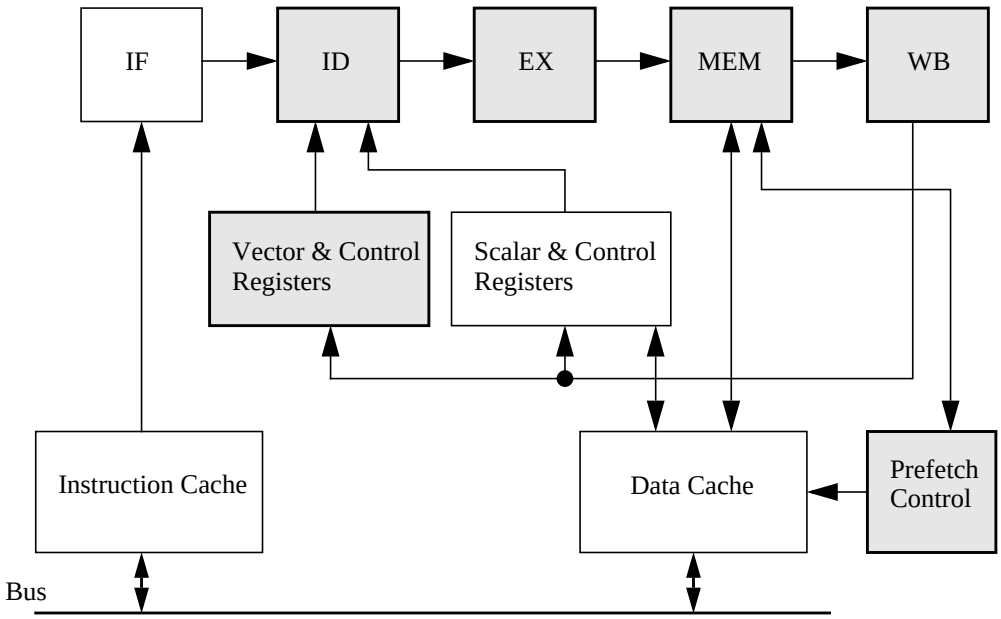

Figure 8.1: Modified pipeline

The vector pipeline uses the underlying JAMAICA pipeline with modifications to the stages. The pipeline is illustrated in Figure 8.1 with the modified parts shaded. The IF stage is unchanged since vector instructions have the same size as the scalar instructions (32-bit) and context switching is allowed to operate in the same way as for scalar execution.

The ID has been modified to recognise the vector instructions. The vector instruction format is similar to the scalar instructions. The stage also generates addressing parameters

to the vector register file and fetches the operands. It stalls the IF stage till the vector instruction is executed. The stage also supports multithreading; in the case of a memory miss triggering a context switch, the decode stage saves the vector addressing states so that the vector instruction is resumed when the context becomes active again. A vector instruction is resumed at exactly the faulting point.

The EX stage executes vector register-to-register instructions in essentially the same way as the scalar mode. However, split arithmetic, shift, and type conversion are performed. The execute stage also calculates the address for loads and stores. Address generation is more complex than the scalar mode as it calculates two dimensional addresses. However, no multiplication is required.

The MEM stage functions similarly to the scalar mode in the sense that operations passing the stage are guaranteed to complete. The stage writes back the results and accesses memory in case of load and store vector instructions. A similar delayed load can occur between a vector load and another vector instruction that consumes that value. This only happens for a vector length of one which is very unlikely.

Vector instructions do not cause any control transfers, scalar instructions are used instead. Updating the control register is bypassed in the EX stage to allow for changing the vector length to take effect for a subsequent vector instruction.

The pipeline control for vector instructions is illustrated by a vector add instruction in Figure 8.2. The left-hand column shows the time steps starting at step t. The next column shows the pipeline stage allocation at a given time. The right-hand column shows the contents of the vector index register (VI) and the vector flag bit (VFlag). VI is used to record the last address at which a vector element is stored. The VFlag bit is set when a vector instruction has committed one of its elements but not yet finished execution. The VFlag is used by the ID stage to know whether the current vector instruction has been interrupted by a context switch and decides whether to resume execution from the current value of VI or to start a new instruction.

The pipeline allocation is the same for all other vector instructions except unaligned memory vector load and store instructions. For unaligned loads and stores an extra access

Cycles	IF	ID	EX	MEM	WB	VI	VFlag
t	vadd					d	d
t+1	I+1	vadd				d	d
t+2	I+1	vadd	vadd			d	d
t+3	I+1	vadd	vadd	vadd		d	d
t+4	I+1	vadd	vadd	vadd	vadd	d	d
t+5	I+2	I+1	vadd	vadd	vadd	0	1
t+6	I+3	I+2	I+1	vadd	vadd	1	1
t+7	I+4	I+3	I+2	I+1	vadd	2	1
t+8	I+5	I+4	I+3	I+2	I+1	3	0

Figure 8.2: Vadd example

is performed at the start of every change of dimension. The first access is stored in a temporary register and is used to align the data.

8.2.2 Data Cache

The 2D cache is implemented by a small modification to the data cache and the addition of a prefetch control unit. The cache-line replacement policy is modified to LRU. The prefetch mechanism is implemented with minimal change to the underlying data cache design. The cache has a data buffer, for each context, to fill and evict cache lines (The buffers are needed as the bus width is smaller than the cache line size). The cache data buffers are extended with prefetch buffers. The prefetch control unit keeps track of outstanding requests for every context and issues prefetch requests. This information is held in the PrefetchInfoTable (one per context). The table contains the prefetch target address, stride, and prefetch count.

The prefetch processing algorithm is shown in Figure 8.3. On a cache miss (MEM stage), the corresponding context entry is initialised with the miss address and stride, either obtained from the vector instruction or from the stride control register. On a subsequent cycle, if no data cache operation occurs, the prefetch control unit accesses the cache as a

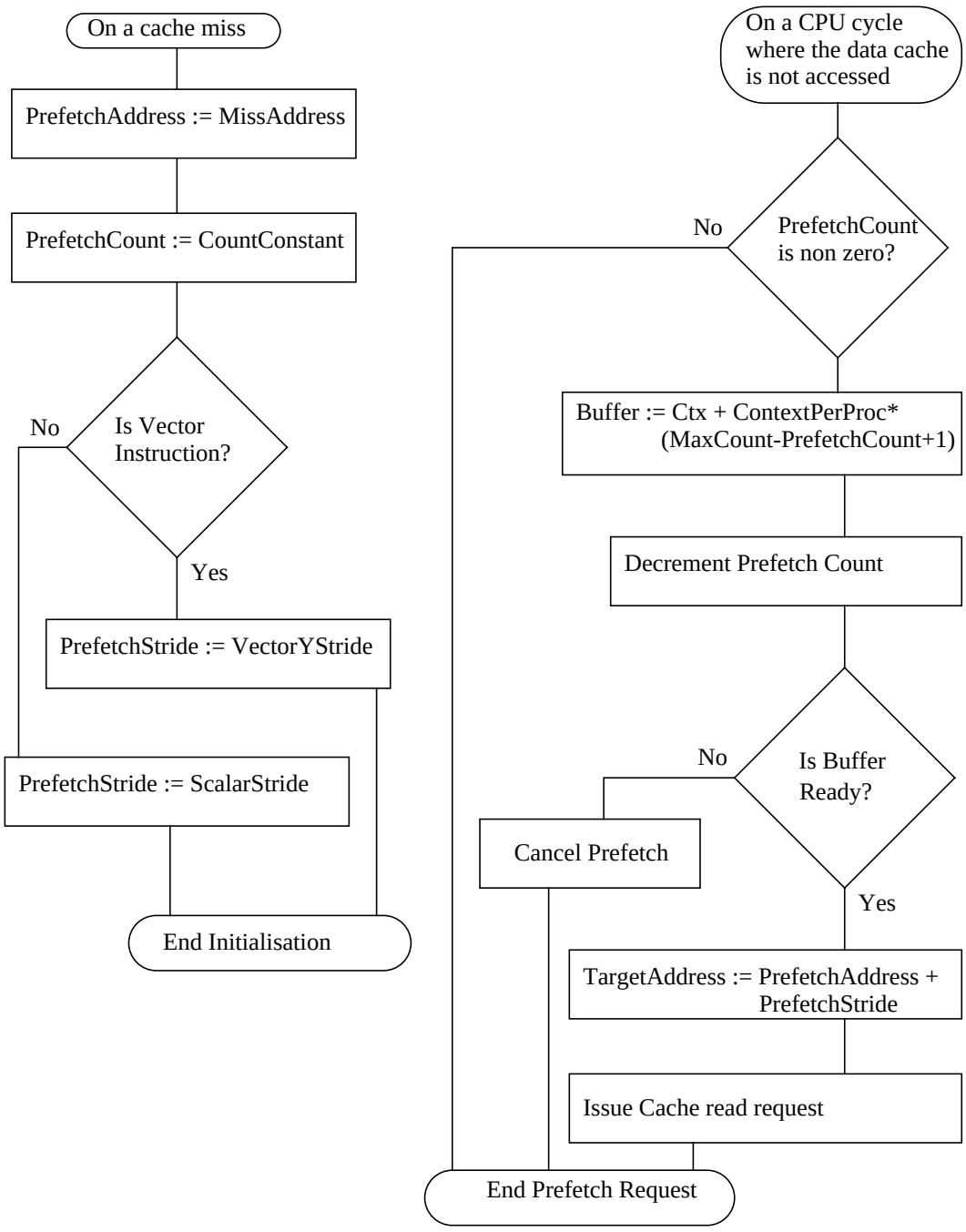

Figure 8.3: Prefetch flowchart

normal cache read cycle with the simple modification that retrieving a cache line does not affect the context logic (no context is signalled as ready to execute).

The prefetch read request is given a lower priority than the normal requests. This is done by allocating the read buffers after the context buffers (the prefetch read buffers, for

different contexts, are interleaved). The cache logic checks the buffers in order starting with the context buffers. The prefetch requests are filled backwards and a prefetch request is cancelled if the corresponding buffer is not free. If the cache is not processing any context requests, then the prefetch requests will be handled as soon as a prefetch buffer is allocated. However, in the case that the cache is busy processing context requests then newer prefetch requests will be serviced first. Servicing older requests, in this case, might not be beneficial as it is likely that the data will be required sooner decreasing the benefit of prefetching.

8.3 Compiler Support for Multimedia

The architecture requires software support to be provided. In this section we describe the compilation route from Java supporting the 2d-vector instruction set. There are four main ways to incorporate multimedia extensions into higher-level languages: the first, which is the straight forward method, is to directly embed vector instructions in the high-level code. This involves a considerable effort from the programmer. The second is to provide a set of library methods that are written in assembly language and optimised. The methods should cover a wide range of typical multimedia operations. Whenever a programmer wants to use the optimised code, the programmer calls the required method. While this approach is easier on the programmer, it is less flexible and the granularity of the operation has to be large to hide the overhead associated with method calls and limiting vector register value reuse (unless a complex inter-method register allocation is used).

The third way is to extended the semantics of the programming language to include constructs to specify vector data types. Also a set of methods are specified that are immediately converted to native assembly instructions.

The fourth approach, which is more general, is to keep the extensions completely transparent to the programmer. At compilation time, or at run-time, sequences of code are analysed and automatically vectorised [5].

The first three approaches are followed by most major microprocessor manufacturers to support their multimedia instruction sets. However, they are targeted at C and C++ rather than Java. Java requires different techniques. No assembly language can be embedded in

Java programs, since Java is translated into an intermediate format that is machine independent. This makes the first approach impossible. The second approach can be implemented by providing native methods. The language can not be extended as this will require changing the bytecode specification so the third approach is not directly applicable. The fourth approach is currently receiving attention. However the focus is on Just-in-time vectorisation which requires extensive analysis.

Our approach is similar to the third way but differs from that in the literature in that the language specification is not extended nor the bytecode, and no native methods are specified. This has the advantage of being more flexible than providing native methods and exploits the fact that most multimedia kernels are simple loops that can be vectorised by the programmer. For a machine to exploit the vector information it needs to provide a special JIT system as described below. If not then the program will still run but with a performance loss.

8.3.1 Multimedia Java Class Library

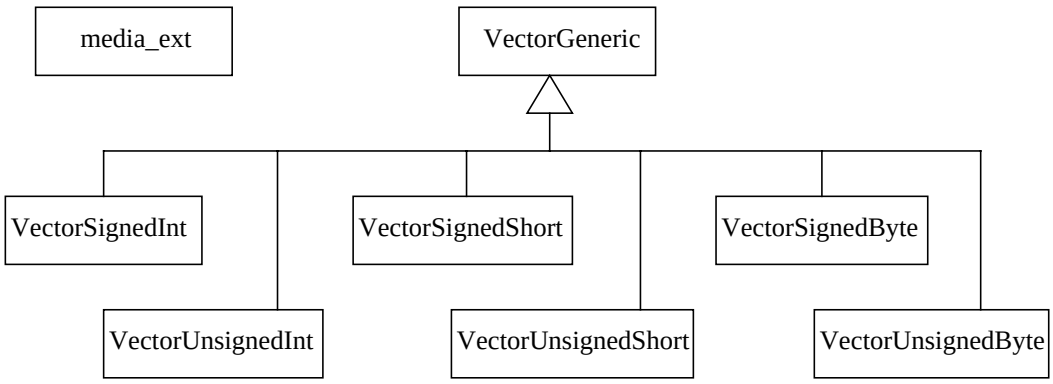

Figure 8.4: Media API class hierarchy

The system provides a multimedia class library called ‘media API’. The class hierarchy for the library is shown in Figure 8.4. The library consists of two main parts. The first models the vector data types, and the second includes the class ‘media_ext’ that provides static methods implementing vector operations.

The class ‘VectorGeneric’ is a generic vector. It is a vector of words of unlimited length. The subclasses of that class provide specific vector type information. The types are associated with the subword data types. The widths are specified by the second name where Int is 32-bit, Short is 16-bit (two packed), Byte 8-bit (four packed). Each type may be signed or unsigned.

The ‘media_ext’ class acts as a module that provides methods to implement the vector instruction set. For every vector instruction there is a method.

A vector is created by media_ext.vec_create() and initialised by vec_init(). For example, a vector of type SignedByte at zero offset is created by the code shown in Figure 8.5.

```
byte ByteArray = new byte[ARRAY_SIZE];
...
VectorSignedByte V = (VectorSignedByte)media_ext.vec_create();
media_ext.vec_init((VectorGeneric)V, ByteArray, 0);
```

Figure 8.5: Vector creation example

The vector `V` is created and its base address is set to point at the element number zero of the byte array `ByteArray`. A subsequent load instruction loads elements from the base address to the vector.

8.3.2 Translation from Bytecodes

The previous specification of the multimedia library allows for efficient JIT compilation. The media API is supported with simple algorithms. The following are restrictions on the code that have to be true for the optimisations to occur:

1. Vector data types are not used in method calls or return types.
2. The vectors are not copied (there is no reference to the vector objects in the stack except for a vector method create).
3. The number of vectors used is less than the number of hardware registers.

4. The vector state is not saved on method call. So optimisations are currently performed for leaf methods.

The first two constraints guarantee that the local variable that holds a reference to a vector object will be the sole local variable to have this reference and it will continue to hold that value throughout the method code. Moreover no method returns a vector except `vec_create`. This is to ease register allocation by guaranteeing that no new vector is created. Another approach is to call a method on a vector object, however this will complicate vector register allocation.

The other two constraints are used to simplify the compilation as all the examined kernels are leaf methods and did not require more vector registers than provided. Accordingly these conditions can be relaxed in a future implementation.

Modification to the second pass

The second pass of jtrans is modified to recognise media_ext method calls and to make sure that condition 1 (partially, only the part that vectors are not used in other method calls) applies. Every static method call is looked up in a hash table of known media API methods. On success, a method call is substituted with a vector_opcode intermediate format.

On load and store and other stack manipulations, whenever an object reference is pushed on the stack or used in other method calls, the reference is flagged as nonvector. This will trigger an error later in the register allocation phase in the third pass if that variable is allocated to a vector register.

Modifications to the third pass

The third pass recognises all conditions. Condition 1 is checked to see if the input parameters contain a vector register. The other conditions are checked when the register allocation is done.

Register allocation is simply extended. The local variables are checked for vector data types. If a variable is assigned to any local variable, it violates condition 2 and the compilation is aborted. Otherwise, the vector registers are assigned in order until overflow. If this occurs then compilation is aborted. No vector register spills and fills are handled.

On generating code, the extended bytecode is recognised and the appropriate code is generated. Any vector arguments will already be assigned a vector register and thus no vector allocation takes place.

8.4 Other System Software

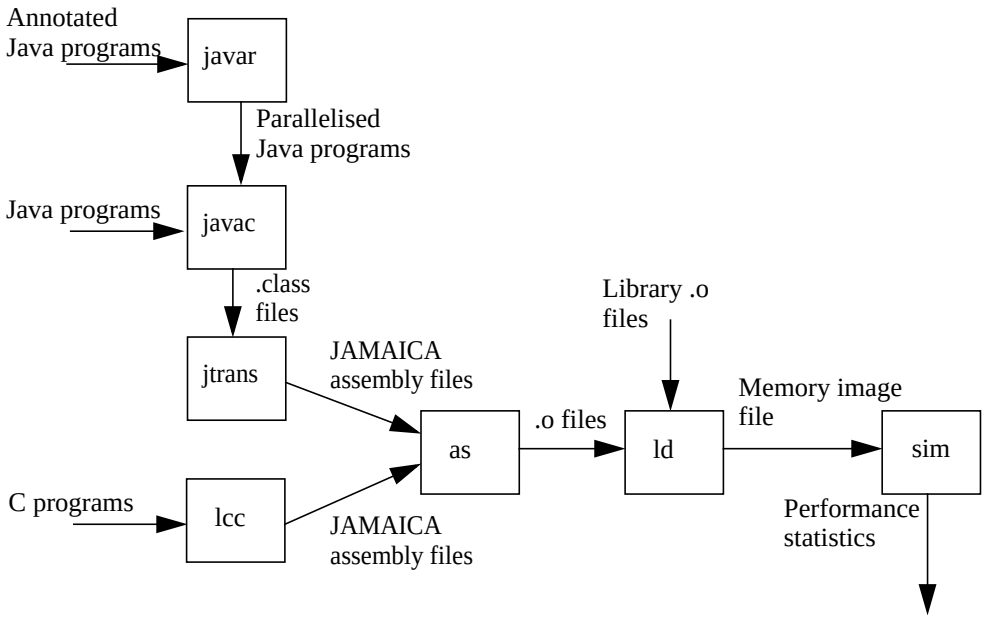

Figure 8.6: Simulation environment

Figure 8.6 illustrates the system software used in the simulation environment. The system supports Java and parallel processing. The latter is supported by the Javar restructuring compiler [4]. Javar was developed at Indiana university and retargeted to the JAMAICA system. Java parallel loops are manually identified and annotated. Javar takes the annotated programs and restructures them to work in parallel. The scheduling is based on

recursive division of the loop iterations. A processor is initially allocated the whole loop iteration. A processor can fork half the iteration to another processor if one is found idle.

The second component of the system is the Java compiler Javac. Javac is Sun's JDK compiler. It is used to compile Java programs into class files. Jtrans is then used to translate class files into JAMAICA assembly files with 2d-vector extensions as described earlier.

Prof. Ian Watson ported the Princeton Compiler to the JAMAICA system and modified Javar. Greg Wright developed the JAMAICA assembler, loader, and simulator. The simulator is based on a register-transfer level, where on every cycle the effect is modelled. We extended the assembler, loader, and simulator to incorporate the 2d-vector multimedia extensions.

8.5 Summary

This chapter describes the system architecture details at the simulation level. The underlying JAMAICA architecture is described and the extensions are highlighted. For vector processing, a simple modification is done to the pipeline to process the vector instructions. Vector pipelining is simple as the pipeline is statically partitioned and only a minor load delay can happen with a vector length of one.

The 2D cache implementation is described. The implementation utilises the existing cache coherence protocol and merely adds a prefetch state per context.

The compiler support for providing a high-level interface for the vector instructions is described. It is based on providing a class library which the programmer uses to code vector routines, and the compiler generates equivalent vector instructions. This is done without extending Java or the bytecode specification.

Finally, other software components of the system are described including simulation and compilation.

Chapter 9: Parameter Space Exploration

The previous chapter described the system architecture. This chapter and the next chapter examine the performance of the whole system. This chapter starts by building an analytical model to capture the performance of the system and limit the simulation space. The model essentially combines the effects of instruction set improvement and the 2d-cache improvement. The model is then used to examine the effect of prefetch count, memory speed and the machine word size. The parameters are obtained from the mpeg2encode and mpeg2decode kernels and bounds on performance are obtained. Moreover a theoretical upper bound on performance is obtained and verified by simulation in the next chapter.

9.1 Analytical Model of the 2d-vector Architecture

The overall performance of a single thread can be decomposed into two components: CPU instruction execution cycles assuming zero memory latency (instruction cycles), and CPU idle cycles waiting for data and instructions (memory access cycles). Data access delays are more significant than instruction delays. The first reason for that is the L1 cache is split for instructions and data, and most kernels have tight loops that fit in the L1 cache. The second reason is that the vector instructions inherently decrease the instruction bandwidth requirement significantly.

The performance can thus be decomposed into instruction execution cycles and memory access cycles for data (ignoring instruction fetch cycles). Figure 9.1-a illustrates this view; on average every h instruction cycle, a cache miss occurs and the instruction execution is stalled for $T1$ cycles while the memory is accessed. It is not necessary that the memory access is uniformly distributed across the total execution time, the aggregation of instruction execution cycles and memory access cycles are the same. Uniform acccess will help formulate performance bounds shortly.

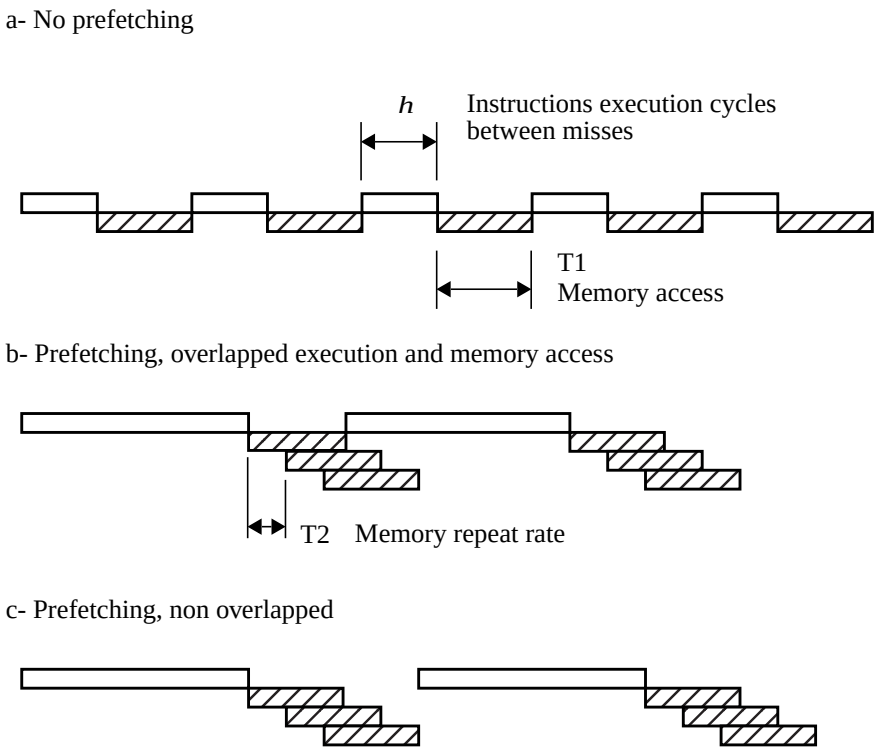

Figure 9.1: Performance components

Taking the 2D cache into consideration, the number of misses should decrease, and thus h should be increased to say, h'. The best execution scenario is that the processor executes instructions for h' cycles, waits for cache line refill and initiates b prefetching requests (pipelined), then resumes instruction execution as soon as the miss line arrives. This is illustrated in Figure 9.1-b. As long as the execution time of instructions between memory accesses is large enough to hide the extra prefetching cycles, prefetching does not introduce any penalty. Otherwise, if the total prefetch cycles is bigger than the cache refill and the instruction execution cycles, prefetching will start to introduce overhead. Thus the minimum overall execution time t_{min} is given by:

$$t_{min} = misses \cdot max(h' + T_1, b \cdot T_2 + T_1) \quad (15)$$

The worst case execution scenario happens when there is no overlap between instruction execution and memory access as shown in Figure 9.1-c. Thus the maximum overall execution time, t_{max} is simply the aggregation of instruction and memory access cycles

$$t_{max} = misses \cdot (h' + T1 + b \cdot T2) \tag{16}$$

We did not account for cache line eviction times and contention on the cache write buffer. However, we believe that the effect of these are minimal due to the high overlap typical in these operations. So the lower bound is not the theoretical lower bound for the system, but rather a measure of worst case prefetch performance.

Note that h' is a function of b. As b increases, the misses decrease and thus h' increases. So:

$$h' = h(b) = \frac{InstCycles}{misses(b)} \tag{17}$$

Where $InstCycles$ is the instruction cycles (with no misses), and $misses(b)$ is the number of misses which is formulated by Equation 13 in Chapter 7. Substituting for $misses(b)$:

$$h(b) = \frac{InstCycles}{misses(0)[(1/(b+1)) \cdot \alpha + 1 - \alpha]} = \frac{h(0)}{(1/(b+1)) \cdot \alpha + 1 - \alpha} \tag{18}$$

Where α is the ratio of misses that benefit from the cache prefetch mechanism.

Figure 9.2 shows the shapes of the overall execution cycles for the bounds for varying b and fixing all other parameters. There are two factors affecting the execution cycles; the improvement in cache misses and the penalty of having extra memory fetch cycles. On the left part of the graph both the lower and upper bound total cycles curves decreases as b increases as the number of cache misses is decreased. Then a point is reached where the prefetch cycle count outweighs the benefit of miss reduction and thus performance degrades with increasing b. The upper bound starts to increase well before the lower bound because there is no overlap between prefetching cycles and execution and so the prefetching penalty is larger.

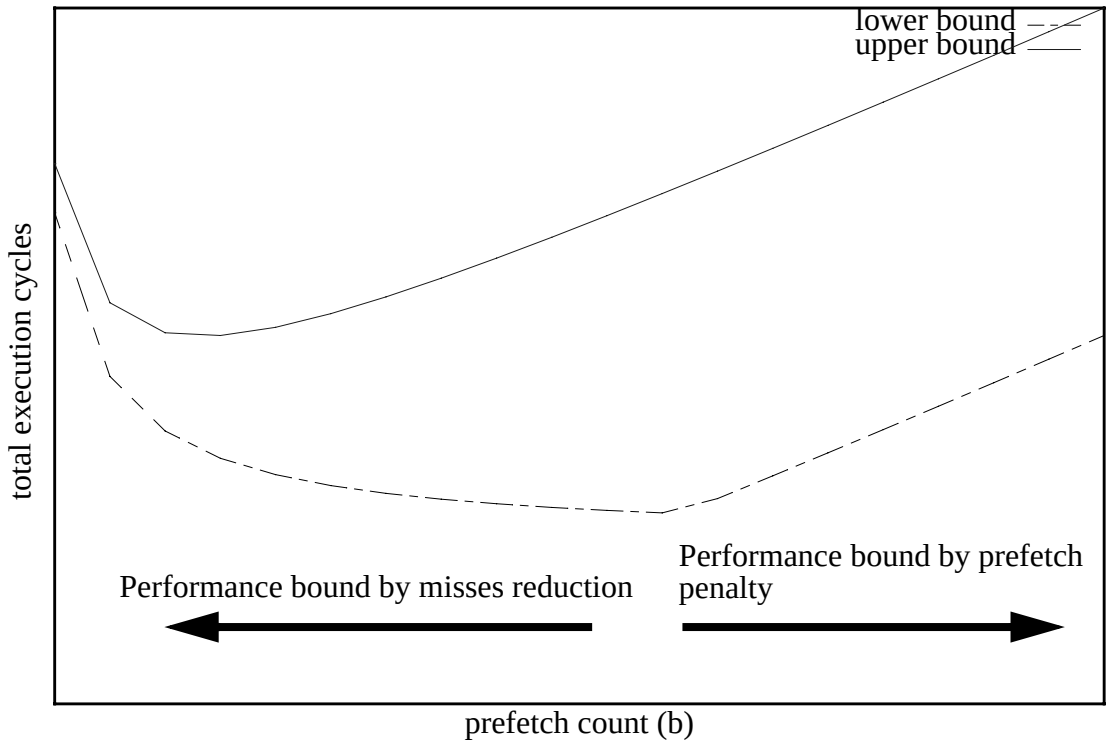

Figure 9.2: Effect of prefetching on total execution cycles

There is a trade-off between decreasing the instruction cycle count and increasing the relative memory latency. To illustrate, assume that the instruction cycle count is improved by a factor a and prefetching is used. The minimum overall cycles is given by:

$$\begin{aligned} t_{min}(a,b) &= misses(b) \cdot max\left(\frac{h(b)}{a} + T1, b \cdot T2 + T1\right) \\ &= \frac{InstCycles}{h(b)} \cdot max\left(\frac{h(b)}{a} + T1, b \cdot T2 + T1\right) \end{aligned} \tag{19}$$

then the maximum speedup, as a function of a and b, can be obtained by:

$$speedup_{max} = \frac{t_{min}(1,0)}{t_{min}(a,b)} = \frac{(h(0) + T1)/h(0)}{\left(max\left(\frac{h(b)}{a} + T1, b \cdot T2 + T1\right)\right)/h(b)} \tag{20}$$

Figure 9.3 plots the equation and shows the shape of the combined effect of instruction set based improvement and memory based improvement. The xy plane has the a and b variables. The z-axis represents the maximum speedup. For a fixed b, increasing a will

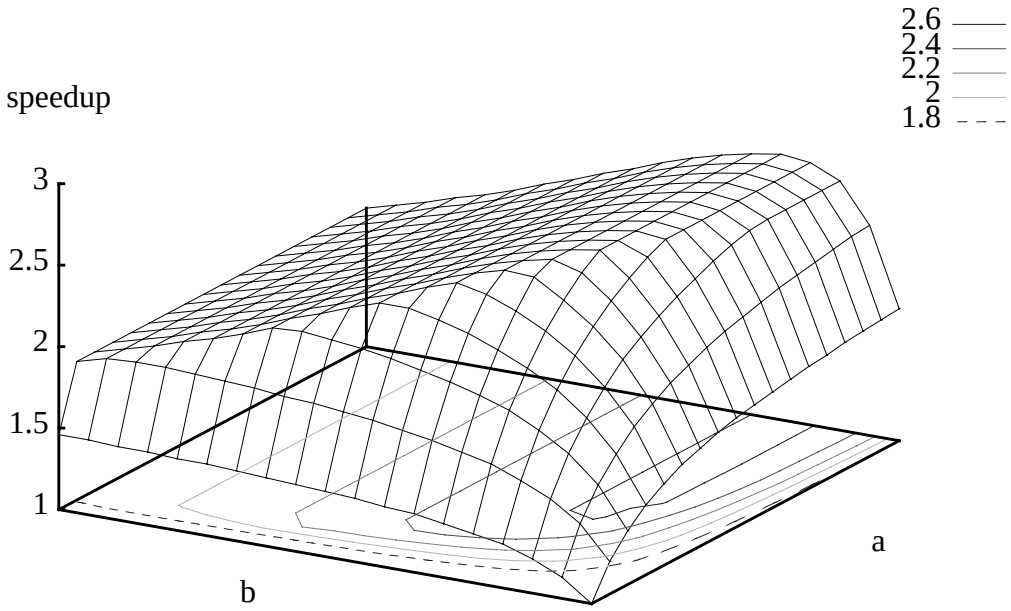

Figure 9.3: The performance effect of instruction cycles reduction and prefetching

increase the speedup to a point where the relative miss penalty increases and halt any further performance improvement. For a fixed a, increasing b will increase the speedup up to a point where prefetch penalty will increase and dominate the performance. As a increases, the optimal value of b decreases, this suggests that b values have to be small. For higher values of b, the effect of a is minimal.

It should be noted that we assume that a does not affect misses. This is reasonable as we have instruction and data caches and the instruction effect is thus isolated. However, increasing the machine word size might affect the misses, and we are not considering this case in the model

9.2 Optimal Parameters

Table 9.1 shows the model parameters. All the parameters, except the prefetch count b, are determined by the architecture and the application. It is interesting to find the optimal prefetch count fixing other parameters.

Parameter	Description	Depends on
$h(0)$	Instruction execution cycles between misses	Depends on total number of instructions, and 2D access ratio
$T1$	Memory access cycle	Depends no technology
$T2$	Memory repeat rate	Depends on technology and memory organisation
α	2D access ratio	Depends on the memory access behaviour
b	Prefetching count	Variable
a	Instruction set speedup	Depends on vector length, machine word size

Table 9.1: Model parameters

The optimal value of the prefetch count b can be obtained by formulating the speedup as a function of b and differentiating with respect to b. Examining Equation 20, only the denominator depends on b, so obtaining the minimum values for the denominator will determine the maximum speedup value. The denominator of Equation 20 can be written as:

$$Denominator = Max\left[\frac{1}{a} + \frac{T1}{h(b)}, \frac{1}{h(b)} \cdot (b \cdot T2 + T1)\right] \tag{21}$$

$$= Max\left[\frac{1}{a} + \left(\frac{1}{b+1} \cdot \alpha + 1 - \alpha\right) \cdot \frac{T1}{h(0)}, \frac{1}{h(0)} \cdot \left(\frac{1}{b+1} \cdot \alpha + 1 - \alpha\right) \cdot (b \cdot T2 + T1)\right] \tag{22}$$

The left-hand side of the Max operator always decreases as b increases. The right side does have a peak.

So the right-hand side is

$$Rightside = \frac{1}{h(0)} \cdot \left[\frac{1}{b+1} \cdot (\alpha T1 + \alpha b T2) + (1-\alpha)T1 + b \cdot (1-\alpha)T2\right] \tag{23}$$

Differentiation with respect to b and equating with zero we get

$$b = \sqrt{\frac{\alpha(T1 - T2)}{(1-\alpha)T2}} - 1 \tag{24}$$

As α increases, b increases. If the prefetch is perfect ($\alpha = 1$) then b becomes infinity. Also as $T2$ decreases b increases as $T2$ determines the prefetch penalty. b increases with $T1$. With higher memory latency, more prefetching is needed. This optimal value for b is obtained when the right-hand side is bigger than the left-hand side. This gives the following inequality:

$$b^2(1-\alpha)T2 + b\left(T2 - \frac{h(0)}{a}\right) - \frac{h(0)}{a} > 0 \tag{25}$$

So the optimal will be the right most root when equating with zero:

$$b_{optimal} = max\left(\sqrt{\frac{\alpha(T1-T2)}{(1-\alpha)T2}} - 1,\ \frac{-T2 + h(0)/a + \sqrt{(T2 - h(0)/a)^2 + 4\cdot(1-\alpha)T2h(0)/a}}{2\cdot(1-\alpha)T2}\right) \tag{26}$$

While the above equation gives the optimal b values, it might result in very large values as the denominator of the speedup equation has the factor $1/b$. Thus the difference between a very large number for b and a relatively small one will result in only a slight decrease in the overall speedup. So to obtain a practical value for b, we use a near optimal speedup which is 95% of the maximum.

For our analysis we consider the mpeg2encode and mpeg2decode programs. $h(0)$ is determined from JAMAICA simulation, α is determined from cache simulation as described in Chapter 7. We vary the machine word size to study its impact. This will affect parameter a. For the 32-bit model it is determined directly from JAMAICA simulation (assuming zero memory latency).

For other word sizes, we use the kernel speedup values obtained in Chapter 6 and modify them to account for implementation overheads and model approximations. It is likely that the model will overestimate the speedup as many details are abstracted, such as register fills/spills and pipelining bubbles due to conditional branch instruction which are less in the vector code. However in the case of early termination of a loop iteration, the overheads of the vector processing, such as extra initialisation instructions might be more significant. Since the model ignores these overheads, the speedup estimate might be lower than actual speedup.

To formulate, let's assume that a scalar kernel is decomposed into scalar overhead execution cycles (S_o) and perfect execution cycles (S_p). A vectorised kernel would have a similar vector overhead (V_o) and perfect execution cycles (V_p). The speedup we achieved in the previous analysis (Chapter 6) is $s_1 = S_p / V_p$.

For the first case the actual speedup is

$$s_2 = \frac{S_p + S_o}{V_p} = \frac{S_p + S_o}{S_p} \cdot \frac{S_p}{V_p} = k \cdot s_1 \tag{27}$$

Where k is a constant. In this case we scale the speedup for the kernels by k.

For the other case the reciprocal of the speedup is given by:

$$\frac{1}{s_3} = \frac{V_p + V_o}{S_p} = \frac{V_p}{S_p} + \frac{V_o}{S_p} = \frac{1}{s_1} + l \tag{28}$$

The memory access time $T1$ for current technology is 70 cycles (1 GHz CPU speed) and it is expected to be 224 cycles (10 GHz CPU speed) in 10 years time. The memory repeat rate $T2$ is set equal to $(T1)/3.5$ in our analysis, which is the current ratio. There are two reasons for fixing this value. Firstly it is the current ratio and seems set to remain constant for the next 10 years; according to the National Semiconductor Industry Association's roadmap [67], by the year 2010, the data rate will be about 2.5 faster than the current rate. Extrapolating for the memory latency in the past, assuming an 8% improvement rate (according to typical values for memory access cycles), gives a memory latency speedup of 2.19, which is close to the improvement in the data rate. The second reason is that since the L2 cache will affect the memory latency, varying the memory latency will help illustrate both the effects irrespective of whether the change in the latency is due to technology or cache behaviour.

The specified values for $T1$ are raw values that are subject to irregularities (bank conflicts) that might increase them. Moreover having L2 cache will decrease this parameter. Thus, for our analysis we will consider a large range of $T1$.

Tables 9.2 and 9.3 summarise the results for the instruction cycle speedup for three machine word sizes over the case where the vector instruction set and prefetching are not used. For every kernel the fraction of total time that the kernel executed is given. This

information is obtained by simulation using a perfect L1 cache. The other columns give the kernel speedups obtained in Chapter 6. The last two rows of the tables show the overall speedup and the scaled one obtained using Equation 27 or 28. Table 9.4 shows the remaining parameters.

kernel	fraction	speedup for 32-bit word	speedup for 64-bit word	speedup for 128-bit word
Dist1 (l=0.1)	0.60	7.57	14.65	27.52
Overall		2.07	2.25	2.35
Scaled Overall		1.84	1.98	2.06

Table 9.2: mpeg2encode speedup breakdown

Kernel	Fraction	Speedup for 32-bit word	Speedup for 64-bit word	Speedup for 128-bit word
conv420to422 (k=1.24)	0.20	3.78	5.71	11.42
conv422to444 (k=1.81)	0.22	3.16	3.16	3.16
form_comp_pred_av (k=1.44)	0.02	8.55	16.12	28.9
form_comp_pred_cp (k=1.44)	0.03	8.55	16.12	28.9
Overall		1.52	1.56	1.61
Scaled overall		1.62	1.67	1.71

Table 9.3: mpeg2decode speedup breakdown

Parameter	mpeg2encode	mpeg2decode
$h(0)$	1980.16	334.8
α	.75	.88

Table 9.4: Other model parameters

9.2.1 Impact of Memory Latency and Machine Word Size

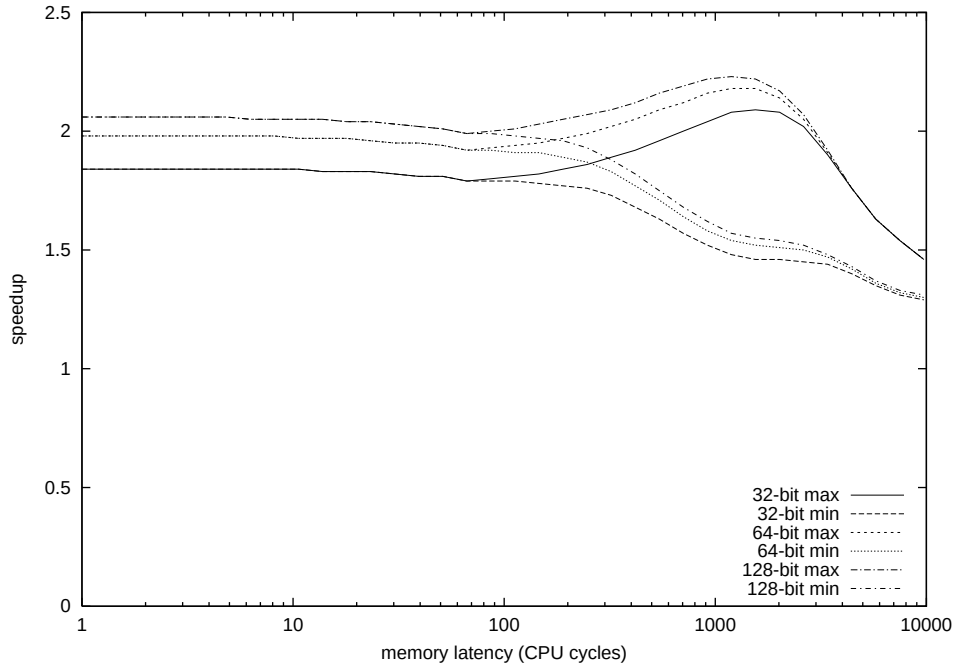

Figure 9.4: mpeg2encode speedup

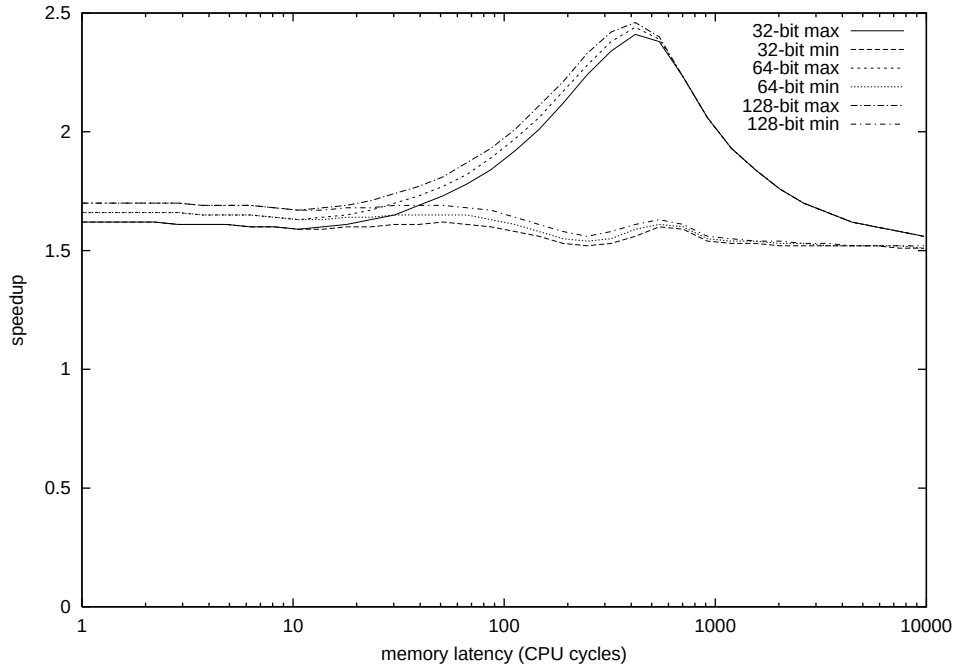

Figure 9.5: mpeg2decode speedup

Figure 9.4 and Figure 9.5 show the speedup bounds for mpeg2encode and mpeg2decode programs. The x-axis is the memory latency and the y-axis is the speedup of the application over the scalar case for the same memory latency. The speedup is calculated as an upper and lower bound for a 32-, 64-, and 128-bit machine word size, 2d-vector architecture. These are denoted by the bit size followed by ‘max’ and ‘min’ respectively. These curves are calculated for 95% optimal prefetch count values. Memory latency is varied from 1 to 1000 CPU cycles.

For fast memory, the upper and lower bounds coincide as there is hardly any memory latency to hide and no prefetching is necessary. For mpeg2encode, as memory latency increases from 40 cycles, the positive effect of prefetch increases, peaking in the region of 400-3000 cycle reaching 14% speedup relative to a zero memory latency configuration.

The mpeg2decode is more memory bound than mpeg2encode. The prefetching has a much more significant effect (up to 60% speedup) and the bounds are shifted to the left accordingly. It is interesting to note that the lower bound does not decrease significantly while the upper bound peaks.

To the right side of the figures, the optimal b values are limited by the relative speeds of memory access and repeat rate and cache efficiency parameter α and thus b stays constant.

The effect of increasing the word size is significant for low memory latency. Beyond the speedup peaks, the word size has no effect on performance as memory becomes a bottleneck.

9.2.2 Optimal Prefetch Count

Figure 9.6 and Figure 9.7 show the 95% optimal prefetch count when the memory latency and machine word size are changed. For both figures, for fast memory, the optimum prefetch count is zero and it increases with the increase of memory latency. It peaks at about 800 and 110 cycles for mpeg2encode and mpeg2decode respectively. Optimal values then decrease as the effect of prefetching increases and stays constant for large memory latency.

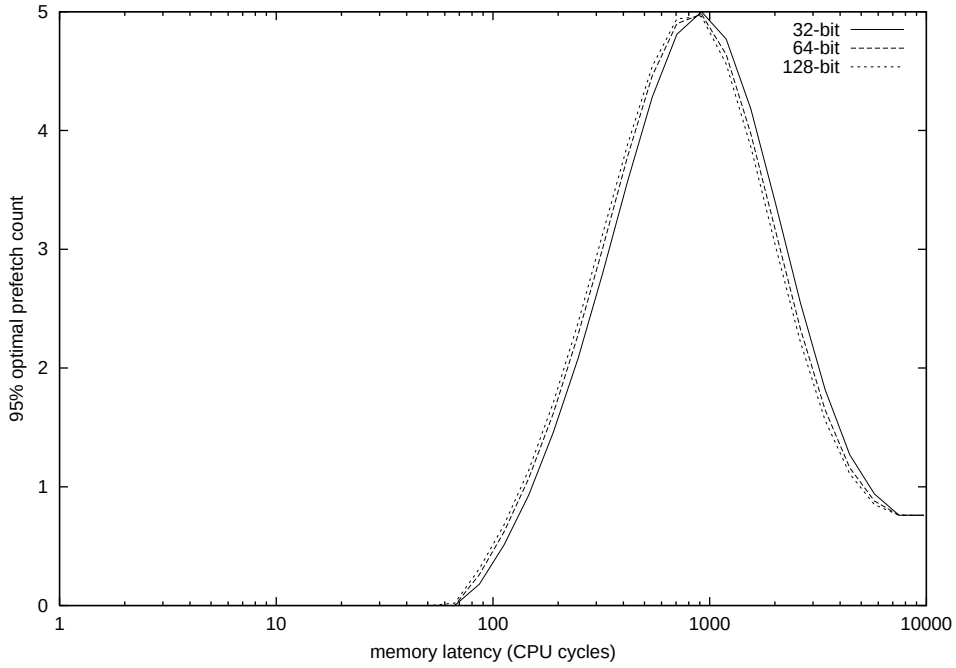

Figure 9.6: Prefetch count curves for mpeg2encode

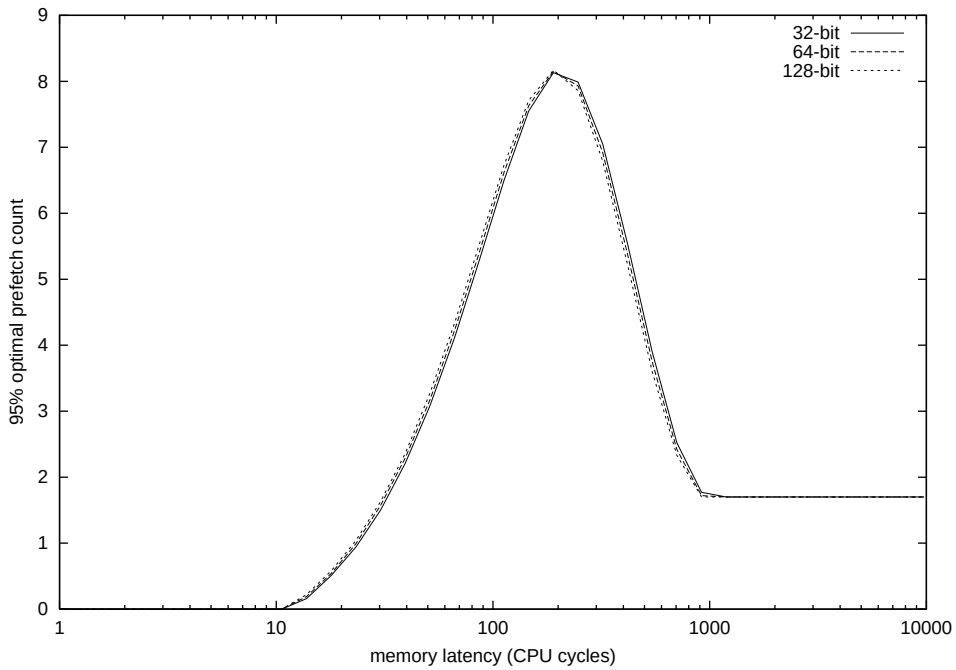

Figure 9.7: Prefetch count curves for mpeg2decode

9.2.3 Bus Utilisation

Bus utilisation is of importance as the system consists of multiprocessors. Bus utilisation can be expressed as:

$$BusUtilisation = \frac{BusCycles}{TotalCycles} \tag{29}$$

The number of bus cycles is given by:

$$BusCycles = misses(b) \cdot (b \cdot T2 + T1) \tag{30}$$

Substituting t_{min} and t_{max} from (15) and (16) for $TotalCycles$ into (29) we get:

$$BusUtilisation_{min} = \frac{b \cdot T2 + T1}{h(b) + b \cdot T2 + T1} \tag{31}$$

$$BusUtilisation_{max} = \frac{b \cdot T2 + T1}{max(h(b) + T1, b \cdot T2 + T1)} \tag{32}$$

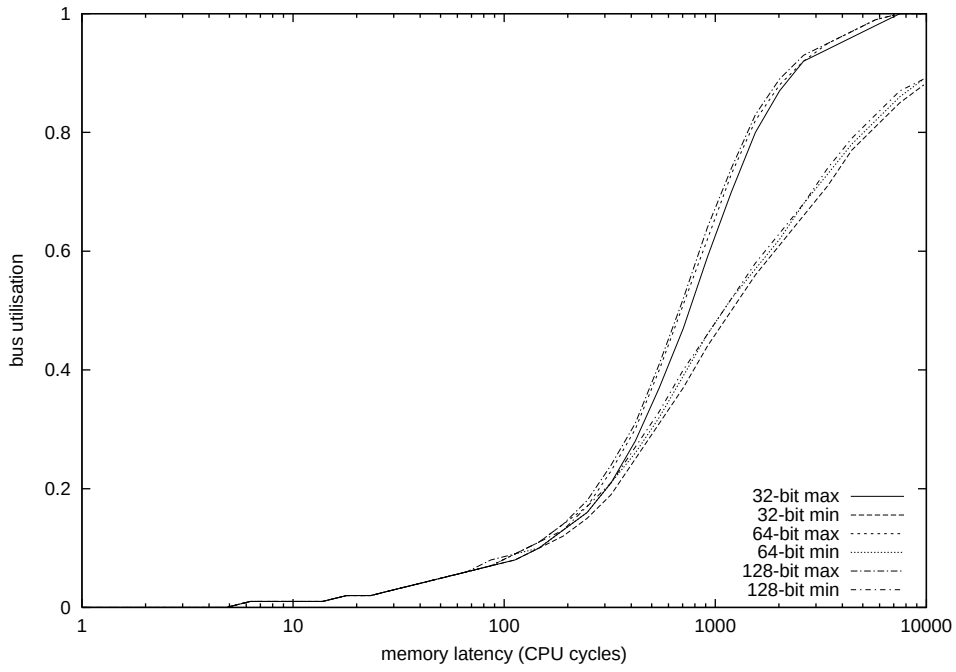

Figure 9.8: mpeg2encode bus utilisation

The Equations are plotted for mpeg2encode and mpeg2decode in Figures 9.8 and 9.9. For mpeg2encode, for current technology (70 cycles and less) bus utilisation is less than 8%

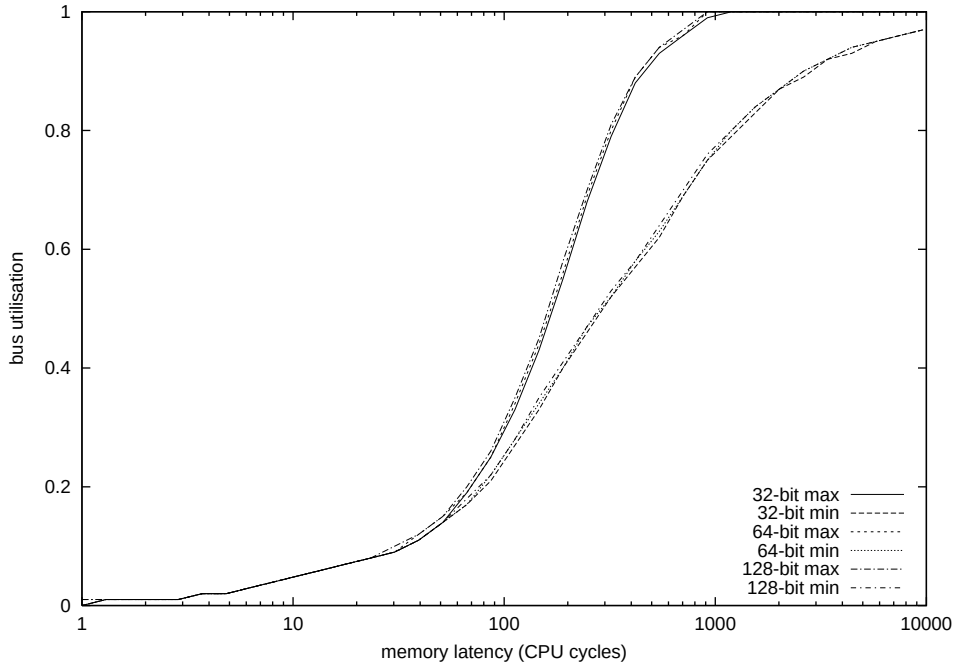

Figure 9.9: mpeg2decode bus utilisation

and for future technology (225) is less then 18%. Utilisation becomes more than 70% for memory latency greater than 900 giving an adequate margin for current configurations.

For mpeg2decode, current technology requires less than 26% bus utilisation. Future technology requires less than 70%. Utilisation increases rapidly after that as mpeg2decode is more memory bound than mpeg2encode.

However we assume an L2 cache connected to the on-chip bus which is expected to be 4 times slower than the CPU. Assuming that the cache has a 50% hit ratio then 220 cycles will become 110. In this case bus utilisation will be less than 33% (for the 32-bit results), this is more likely to provide effective sharing with other processors. Moreover, the analysis is pessimistic as it assumes that the memory read requests, for the original misses, are not overlapped. This is to account for the memory bank contentions. Therefore we expect lower utilisation values from the detailed simulation.

9.3 Summary

This chapter modelled the combined effect of the instruction set improvement and the prefetching improvement. The model is then applied to the mpeg2encode and mpeg2decode applications to study the effect of memory latency, machine word size, and prefetch count. Optimal prefetch counts are derived. The results show that the prefetch has the potential to increase speed by 60%. The potential of prefetching peaked for large memory latency which suits future technology. Bus utilisation is expected to be about 70% (upper bound). However taking into consideration the L2 cache, the maximum utilisation is expected to drop to 33%.

Chapter 10: Simulation Study

The purpose of this chapter is to provide a more detailed analysis of the proposed architecture in a more complete system taking into account more complex interactions that are difficult to model analytically. These include the interaction between cache prefetching and instruction execution, pipeline hazards, prefetching overhead on bus utilisation, multithreading, and multiprocessing.

Simulation is used to study such interactions. The study first analyses in detail the uniprocessor performance, then considers a full multiprocessing multithreaded configuration. Uniprocessor performance is useful in forming a basic configuration that can be used in the parallel system. Moreover, it is used to verify our analytical model (described in Chapter 9) and help assess the efficiency of the simulated system (by deriving an upper limit on performance).

The chapter starts by describing the simulation configuration and measured parameters. Then uniprocessor and multiprocessor-multithreaded configurations are simulated and results are explained. Finally the chapter concludes by verifying the design decisions taken in the early design phase. Parts of the simulation and analytical results are to be published in [17].

10.1 Simulation Setup

Table 10.1 shows current technology parameters and optimistic and pessimistic predictions for technology expected in 10 years time (current, future-op, future-pe respectively). The CPU speeds, bus speed, and data rates are obtained from the Semiconductor Industry Association road map [67]. Memory access time is scaled with the same improvement rate as the data rate (future-op) or kept the same (future-pe).

Table 10.2 summarises the simulation parameters for the base configuration used for every processor. Cache sizes are chosen to be relatively small compared to current processors. This is intended to account for having multiple processors on the chip and the fact that multithreading is used to hide the memory latency. The memory is based on

Parameter	Current (current)	Future-pessimistic (future-pe)	Future-optimistic (future-op)
CPU Clock	1 GHz	10 GHz	10 GHz
Bus clock	1 GHz	2.5 GHz	2.5 GHz
Memory bus data rate	800 MHz	800 MHz	2.5 GHz
Memory access time	70 ns	70 ns	22.5 ns

Table 10.1: Technology parameters

Component	Parameters
L1 Instruction Cache	16 KB 4-way copy back, 32 byte line size, 1 CPU cycle access time
L1 Data Cache	16 KB 4-way copy back, 32 byte line size, 1 CPU cycle access time
L2 Cache	1 MB 4-way copy back, 32 byte line size
Bus	Pipelined, split transaction bus, bus width 128-bit. A bus transaction can start every two bus cycles and takes 8 bus cycles
Memory	One (current) and four (future-op, future-pe) channel Rambus

Table 10.2: Architecture base configuration

RDRAM technology [65]. For the current configuration, a bus cycle transfers 16 bytes, so the maximum bandwidth is 14.9 GB/sec. The current RDRAM memory channel transfers 16 bytes in 8 memory bus cycles with a maximum bandwidth of 1.49 GB/sec. For the most bandwidth hungry application (3d, will be described later in Section 10.2.5), a miss occurs every 306 CPU cycles with an L2 cache miss ratio of 80%. Since a L1 cache line is 32 bytes, this requires a bandwidth of 79.78 MB/sec (19.12 processor loads would saturate the bus).

For the future configuration, the bus bandwidth is increased by 2.5 times while the processor speed is improved by 10 times. The relative bus/CPU speed is therefore decreased by a factor of 4. Assuming optimistic parameters, the memory channel speed is

parameter	Description	Values
prefetch_count	The number of lines prefetched on a cache miss	Varied from 0 to 7
2d_vector_inst	Whether the 2d-vector instructions are used or not	Yes/no
number_of_processors	Number of processors on the single chip	Varied from 1 to 16
number_of_threads	Number of contexts per processor	1 or 4
workload	The name of the benchmark workload	mpeg2encode, mpeg2decode, Viterbi decoding kernel, 3D geometric transformation kernel

Table 10.3: Parameters changed

4 times slower. A four memory channel configuration will thus suffice to provide the bandwidth for the future technology parameters.

The above analysis assumes no channel contention and that memory accesses are randomly distributed, this may severely affect the actual bandwidth. We study this in details by simulation. For the analytical model, in the previous chapter, we assumed a bandwidth of one channel for simplicity and because only one thread performance was considered.

Table 10.3 summarises the other parameters that are changed in the experiments. We use the Java version of MPEG-2 encode and decode programs (mpeg2encode and mpeg2decode), a viterbi decoding kernel [9] used in voice recognition (viterbi), and a 3D geometric transformation kernel [10] (3d). The viterbi and 3d kernels will be described in Section 10.2.5.

10.2 Single Thread Evaluation

10.2.1 Comparing Bytecodes and Native Instructions

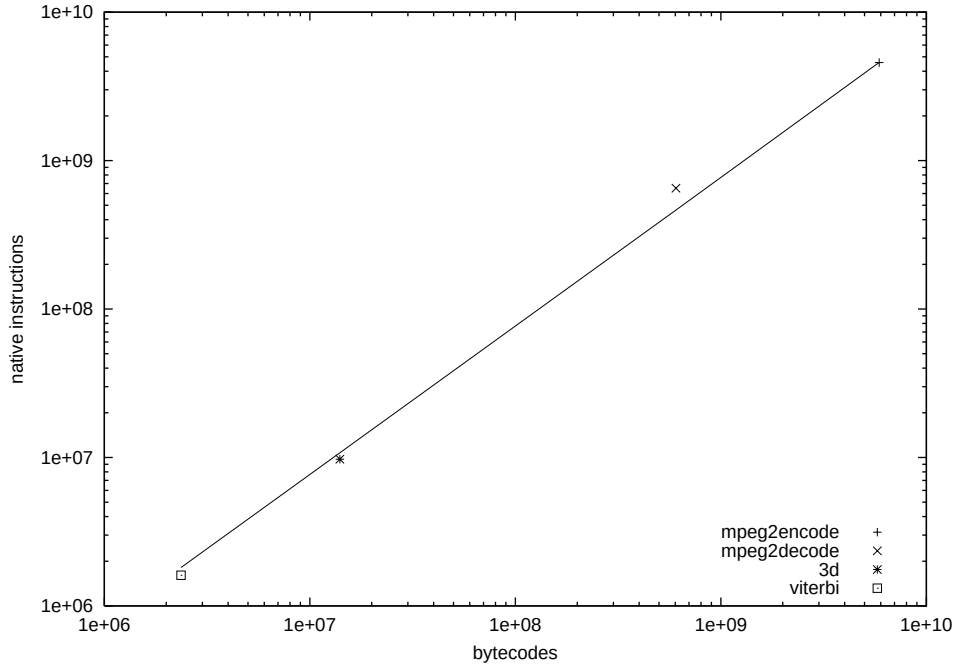

Figure 10.1: Native instructions against bytecodes

Figure 10.1 plots the number of native instructions executed in each application against the number of bytecodes. A straight line can be approximated to the points so that the line slope represents the average number of native instructions generated for every bytecode. This number is 0.77. The 3d and viteribi applications have lower values (0.69, and 0.67 ratios respectively) while mpeg2decode has a higher value (1.08 ratio). This is mainly due to the high register usage in mpeg2decode. This figure does not take into account the compilation overhead. However the compilation is based on a very fast compiler with simple register allocation, which is likely to have a small impact.

Nystrom [58] has compared the bytecode and native instruction counts for a SPARC platform running an interpreter, a JIT compiler, and a static translator. The results, of a set of applications, averaged 15.82, 5.9, and 4.56 for interpretation, JIT compilation and static translation respectively. The minimum ratio among all the applications and

compilation techniques is 1.72. Most other studies in the literature consider overall execution time as a metric. We cannot compare precisely these figures with ours due to different instruction sets, programs, and optimisation techniques. We do not do array bound checking (about 30% extra instructions), nor type checking as for the multimedia workloads, these are easy to identify at compilation time. Optimised code can then be generated. However such a comparison should at least show that our compilation system is no worse than other systems and the overall performance improvement is not due to removal of compilation inefficiencies.

10.2.2 Misses Analysis

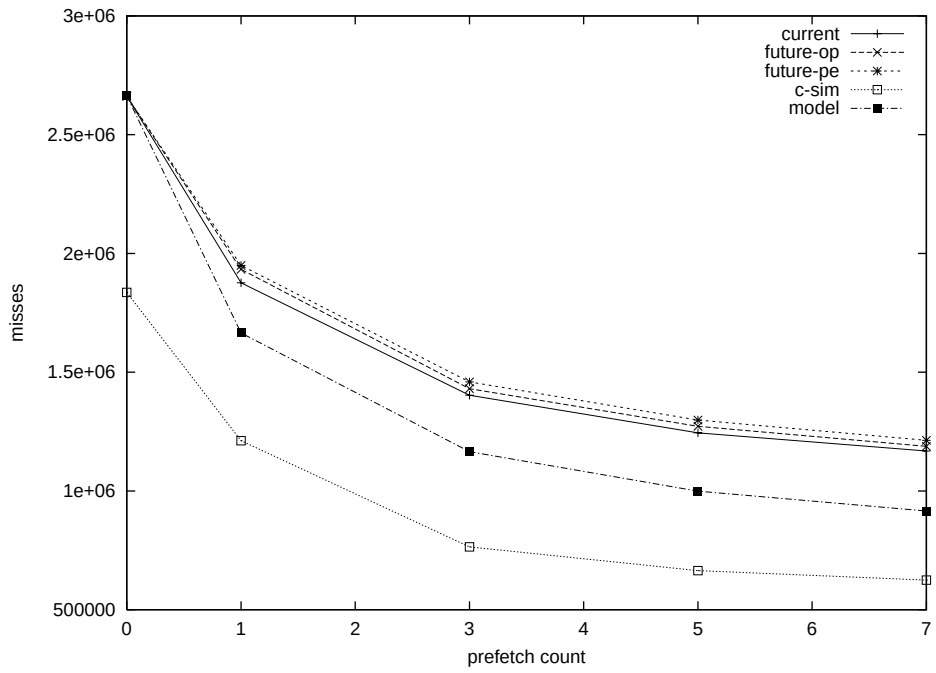

Figure 10.2: mpeg2encode misses

In order to fully relate the analytical study to the simulation study we need to compare C and Java memory access behaviour as trace-driven cache simulation has been done on the C versions of the benchmarks for speed. Figure 10.2 and Figure 10.3 plot the misses against the prefetch count for mpeg2encode and decode respectively. The misses are shown for the simulation results (current, future-op, and future-pe) and the C results (c-sim) obtained by running the same workload on the C trace-driven cache simulation.

The analytical model takes all the parameters from the Java simulation except for α which is obtained from the C trace-driven cache simulation. The initial misses are thus the Java misses and the effect of prefetch count α is taken from the C trace-driven simulation. The misses (model), from the point of view of the analytical model, are plotted in the figure.

For mpeg2encode, the number of misses obtained from the Java simulation is about 48% higher than c-sim. The reason for this is that the misses include memory allocate methods (extra 29%) and other misses due to register allocation differences (19%) where extra variables are used as offsets to implement pointers. However, prefetching has shown a similar improvement in the misses compared to the C simulation. The new α value for the future-pe model is 0.61, for current 0.63 and for future-op 0.62 compared to 0.75 for c-sim.

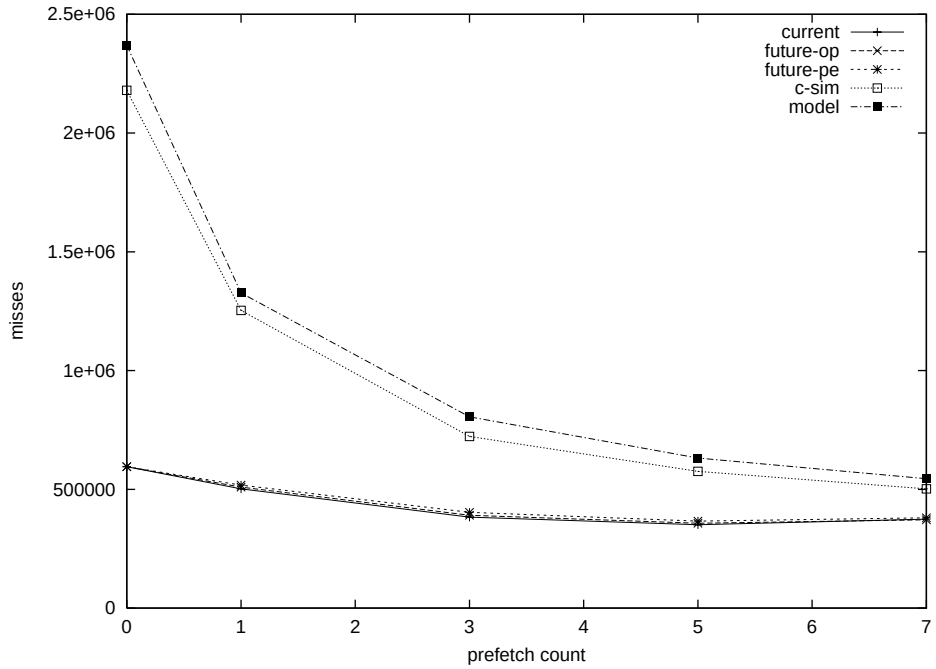

Figure 10.3: mpeg2decode misses

For mpeg2decode, the number of misses, obtained from the Java simulation, are about 28% of the c-sim value with no prefetching. This is because 88% of the misses occur in conv420to422 due to column major access. The vector version changes the way data is

accessed and misses are reduced by a factor of twenty for this particular kernel. Other kernels have similar access patterns. It is interesting to note that using prefetching the gap narrows significantly which decouples the effect of the vector instruction improvement from memory improvement. The new α value for the current model is 0.45, for future-op 0.44, and for future-pe 0.42.

Kernel	Miss	Miss Ratio	$b = 1$	$b = 2$	$b = 3$
conv422to420	35.90%	5.88%	60.25%	50.56%	33.30%
mem_allocate	14.33%	8.93%	100.00%	100.01%	100.01%
sub_pred	8.16%	7.00%	68.23%	53.94%	47.14%
glob_mem_allocate2	5.56%	12.46%	90.67%	88.20%	85.41%
idctrow	5.47%	5.23%	56.58%	44.14%	33.48%
quant_intra	3.19%	0.46%	55.52%	44.03%	31.94%
iquant_intra	3.16%	0.68%	54.42%	42.35%	30.04%
add_pred	2.88%	1.48%	90.32%	74.34%	73.90%
conv444to422	2.44%	0.34%	50.21%	36.53%	26.13%
read_ppm	2.43%	0.12%	100.00%	100.01%	100.01%
quant_non_intra	2.34%	0.61%	54.97%	43.13%	30.94%
iquant_non_intra	2.33%	1.40%	54.48%	42.85%	29.98%
dct_type_estimation	1.82%	0.33%	98.49%	59.87%	59.26%
dist1___BI	1.59%	0.02%	96.80%	76.71%	75.89%
pred_comp	1.23%	0.92%	70.01%	52.03%	47.17%
putAC	0.97%	1.16%	103.49%	107.61%	112.44%
variance	0.85%	0.32%	100.00%	100.00%	100.00%
var_sblk	0.82%	0.15%	94.30%	55.96%	54.51%
clearblock	0.73%	2.94%	96.84%	54.27%	66.49%
frame_ME	0.54%	0.67%	100.03%	100.05%	100.10%
Others	3.26%	0.03%	99.15%	100.84%	101.64%

Table 10.4: Misses breakdown for mpeg2encode

The misses breakdown for the mpeg2encode kernel is shown in Table 10.4. The table lists, for every kernel, the percentage of misses, the cache miss ratio, and the percentage reduction of the misses for prefetch counts (b) of 1, 2, and 3. Most of the kernels benefit from prefetching. We did not use prefetching for kernels that did not show improvement using the 2D prefetching technique. That kernels show a slight degradation in the number of misses due to the prefetching used in the other kernels.

Kernel	Miss	Miss Ratio	$b = 1$	$b = 2$	$b = 3$
conv422to444_vec2_0	21.54%	1.19%	89.32%	77.92%	66.80%
glob_mem_allocate2	13.57%	12.49%	56.02%	49.20%	33.38%
store_ppm_tga	11.29%	0.16%	96.52%	54.26%	54.17%
Decode_MPEG2_Intra_-Block	10.10%	1.09%	100.11%	100.18%	100.27%
form_comp_pred	6.43%	0.78%	78.33%	64.66%	58.40%
Add_Block	5.15%	0.53%	100.15%	100.24%	100.32%
conv420to422_vec2_3	3.65%	1.09%	80.54%	51.74%	46.35%
conv420to422_vec2	3.65%	1.09%	81.64%	52.41%	47.25%
conv420to422_vec2_0	3.64%	1.09%	82.95%	52.56%	47.00%
conv420to422_vec2_2	3.58%	1.07%	83.50%	53.17%	47.50%
mem_allocate	3.47%	8.99%	99.67%	99.62%	99.55%
Decode_MPEG2_Non_Intra_Block	3.05%	0.82%	103.22%	106.54%	109.84%
conv422to444_vec2_serial	1.45%	1.58%	61.10%	36.95%	29.42%
idctcol	1.26%	0.10%	100.88%	101.96%	102.91%
conv420to422_vec2_serial	0.97%	0.18%	97.53%	71.20%	70.16%
Get_Luma_DC_dct_diff	0.89%	14.31%	100.22%	100.28%	100.28%
decode_macroblock	0.83%	0.68%	101.23%	102.09%	103.12%

Table 10.5: Misses breakdown for mpeg2decode

Kernel	Miss	Miss Ratio	$b = 1$	$b = 2$	$b = 3$
Saturate	0.80%	0.23%	103.09%	105.78%	108.62%
readBytes	0.63%	0.48%	100.05%	100.10%	100.21%
Flush_Buffer	0.61%	0.09%	100.11%	100.25%	100.44%
Get_Chroma_DC_dct_diff	0.54%	17.53%	100.00%	100.12%	100.06%
Other	2.90%	0.07%	108.35%	112.26%	125.21%

Table 10.5: Misses breakdown for mpeg2decode

The misses breakdown for mpeg2decode are shown in Table 10.5. Most of the kernels again benefited from prefetching. The conv420to422 kernel is broken down into 5 sub-kernels. This is basically loop diffusion used instead of replicating the vector registers increasing their number, and also coping with boundary irregularities. The same is used for the conv422to444 kernel. Again, for the loops that did not benefit from the 2D prefetching, it is turned off.

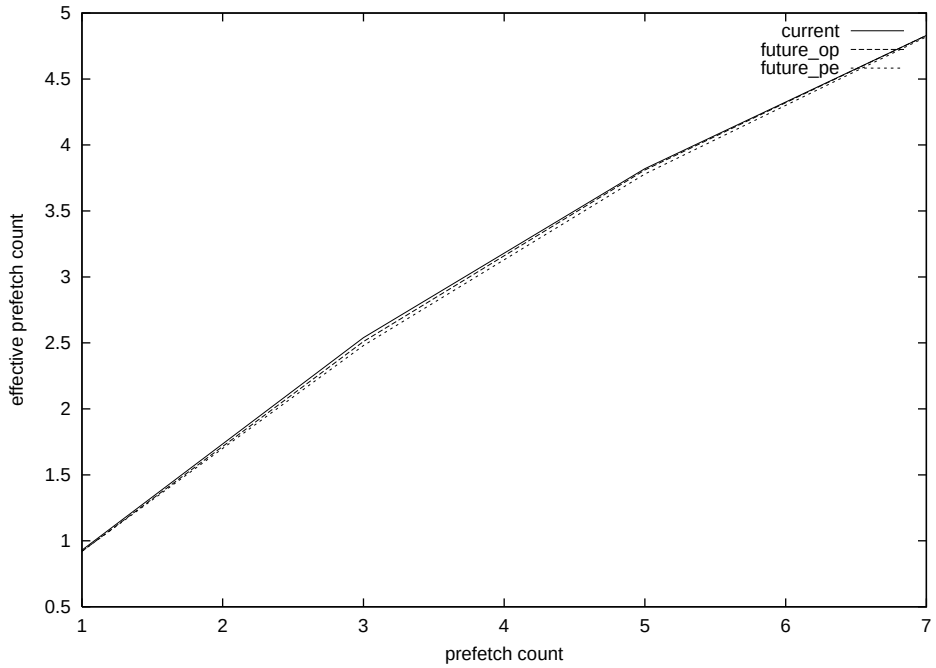

Figure 10.4: Effective prefetch count for mpeg2encode

An important metric is the effectiveness of the prefetch mechanism. We define the effective prefetch count to be:

$$\text{effective prefetch count} = \frac{\text{misses removed}}{\text{prefetches committed}} \times \text{prefetch count} \qquad (33)$$

Values close to the actual prefetch count mean that prefetching is accurate and little cache pollution happens. Figure 10.4 shows these values for mpeg2encode. The effective prefetch count increases with increasing prefetch count. The prefetch efficiency (the ratio of effective prefetch count to prefetch count) is high; more than 0.91 for all memory technologies. It decreases linearly with increasing prefetch count. Eventually the efficiency should be constant for large b as prefetching will be cancelled (increasing b would not increase the number of prefetch requests).

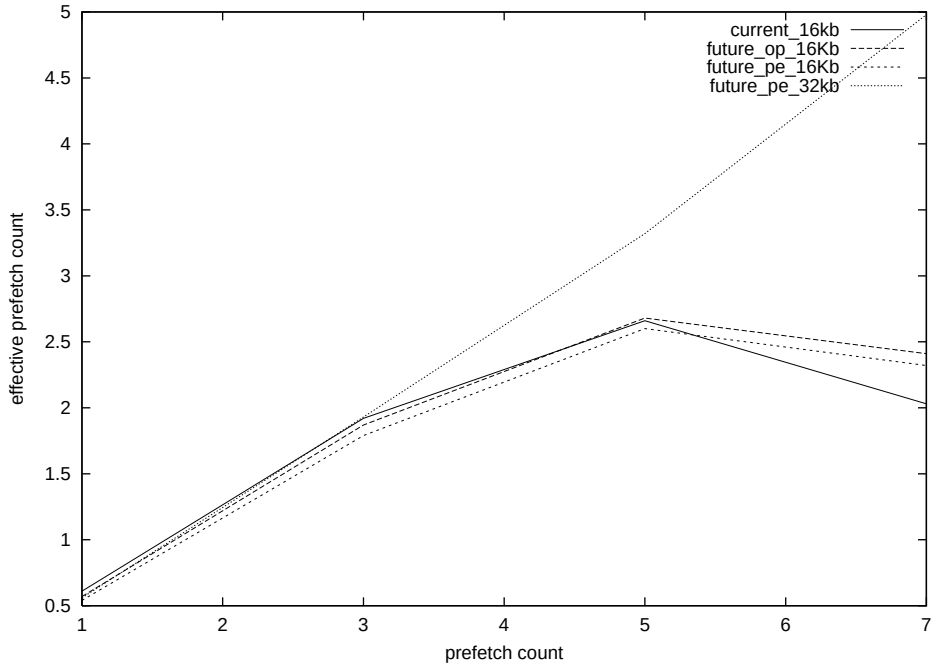

Figure 10.5: Effective prefetch count for mpeg2decode

Figure 10.5 shows the effective prefetch count for mpeg2decode. The effective prefetch count increases until a point is reached at a prefetch count of 5, where the effectiveness of prefetching decreases. The prefetch efficiency is lower than mpeg2encode and starts to degrade after a prefetch count of 3. This is mainly due to cache capacity. The results

shown for a 32 KB cache (future_pe_32kb) show that effect. However, the L2 cache is big enough to hide the effect of these extra misses.

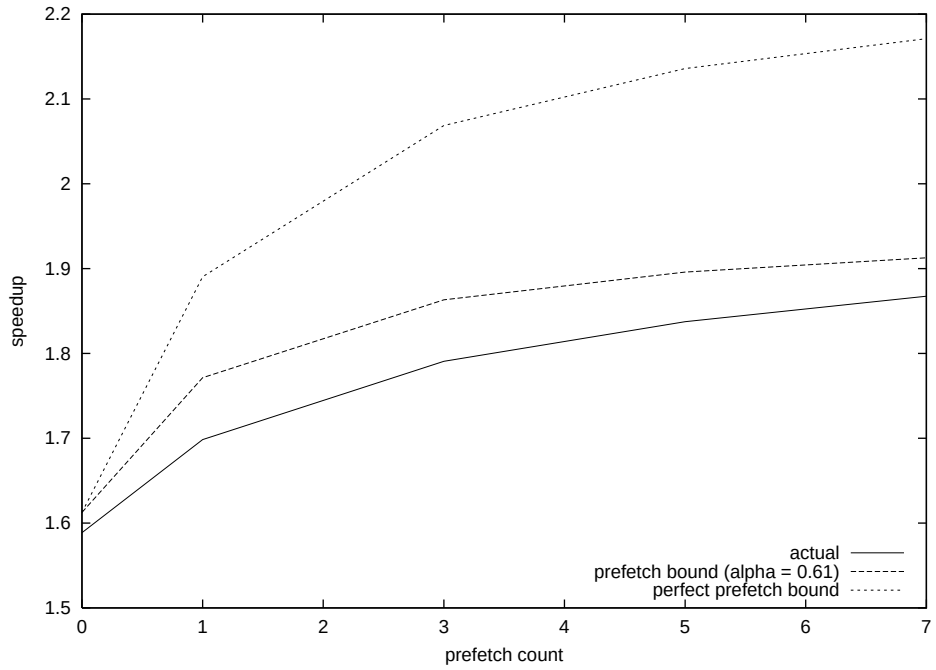

Figure 10.6: mpeg2encode speedup vs. increasing the prefetch count for future_pe

10.2.3 Overall Speedup

We have done three experiments for both benchmarks using different memory technologies. For each experiment we varied the prefetch count parameters from 0 to 7 and plotted the speedup over the scalar case without using any vector instructions. Moreover we plotted an upper bound on performance obtained as follows:

$$speedup_{max} = \frac{ScalarExecutionCycles}{MinVectorExecutionCycles} \tag{34}$$

Substituting $t_{min}(1, b)$ for $MinVectorExecutionCycles$ from Equation 19 in Chapter 9, the speedup is given by:

$$speedup_{max} = \frac{ScalarExecutionCycles}{(InstCycles \times max(h(b) + T1, b \cdot T2 + T1))/h(b)} \tag{35}$$

Where $ScalarExecutionCycles$ is the total execution cycles in the scalar model, and $InstCycles$ is the total instruction execution cycles with no misses and $h(0)$ is the instruction execution cycles per miss in the vector model. These parameters are obtained from simulation (two runs, one for a scalar, and the other for a vector version with no prefetching). Since prefetching is now more accurate than before, and we cancel prefetches if the bus is busy, we define a prefetch upper bound by setting α equal to 1. $T1$ is obtained from the vector simulation by:

$$T1 = (\text{memory read latency} + 2 \times \text{bus transaction cycles}) \times \text{L2 read miss ratio} + \text{bus transaction cycles} \times \text{L2 read hit ratio} \quad (36)$$

And $T2$ is obtained from:

$$T2 = \frac{\text{memory repeat cycles}}{\text{memory channels}} \times \text{L2 read miss ratio} + \text{bus repeat rate} \times \text{L2 read hit ratio} \quad (37)$$

Figure 10.6 shows the speedup results for mpeg2encode for the future_pe memory parameters ($T1$ = 447.89 cycles, $T2$ = 31.86 cycles, $InstCycles$ = 2,862,687,186 cycles, $h(0)$ = 1,075.37 cycles/miss, and $ScalarExecutionCycles$ = 6,538,847,040 cycles). The 2d-vector contributes 59% speedup and the 2D cache contributes 18%. Prefetching obtained 87% out of the perfect prefetch upper bound. Future_op obtained a higher speedup for the 2d-vector (72%) and the prefetch advantage is lower (8%). For current, 2d-vector obtained higher speedup (80%) and prefetching obtained a small improvement (3%).

Figure 10.7 shows the results for mpeg2decode for the future_pe memory parameters ($T1$ = 246.14 cycles, $T2$ = 20.29 cycles, $InstCycles$ = 487,497,774 cycles, $h(0)$ = 819.15 cycles/miss, and $ScalarExecutionCycles$ = 1,089,338,902 cycles). The 2d-vector contributes 50% speedup and the 2D cache contributes 26% (88% of the bound). The 2d-vector improvement increases for other relatively faster memory speeds. For future_op, the 2d-vector contributes 62% and the 2D cache contributes 10.5%. For current, the 2d-vector contributes 62% with a small contribution for the 2D cache of 4%.

Another bound is plotted for the adjusted α. The bound is very close to actual results (within 2%). For mpeg2decode the actual results achieved better than the bound. This is

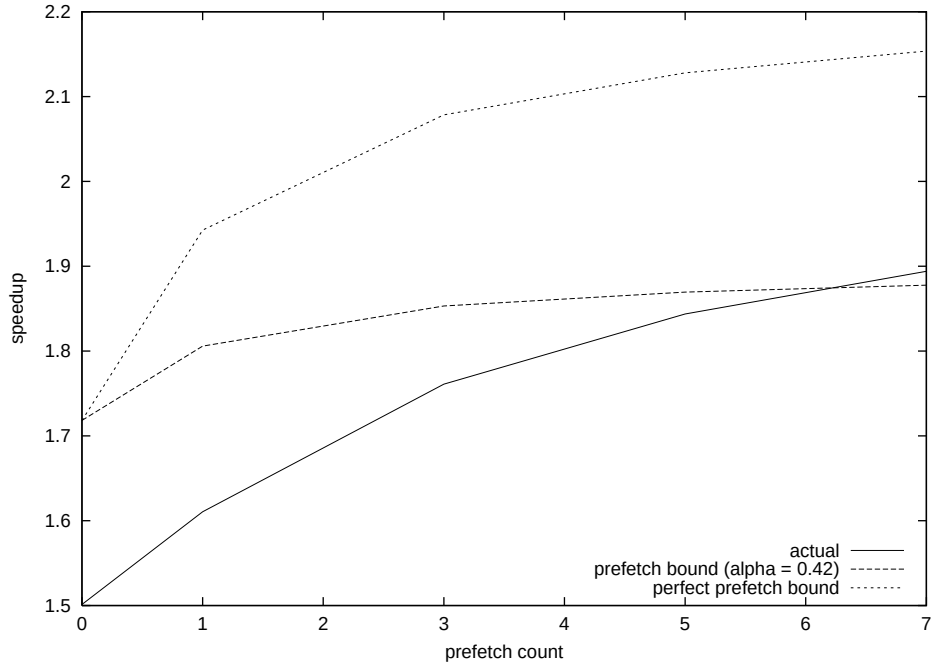

Figure 10.7: mpeg2decode speedup vs. increasing the prefetch count for future_pe

mainly due to the reduction of the misses in the L2 cache which decreases the L1 miss penalty. The bounds for $b = 0$ are higher than the simulation results due to the relatively high instruction cache misses in this application (35% of the misses for mpeg2decode and 1% of the misses for mpeg2encode).

Figure 10.8 and Figure 10.9 compare the simulation speedup results, for the three simulated configurations, with the speedup bounds obtained in the previous chapter. For every configuration, the memory latency and speedup are determined and plotted as a point. The points, from left to right, correspond to the current, future_op, and future-pe configurations respectively. The bounds are closer to the actual results for the mpeg2decode than mpeg2encode (9% and 4% respectively) this is mainly due to slightly better cache performance than the C cache simulation. Results for mpeg2encode show a greater diversion this is mainly due to the slightly larger miss ratios.

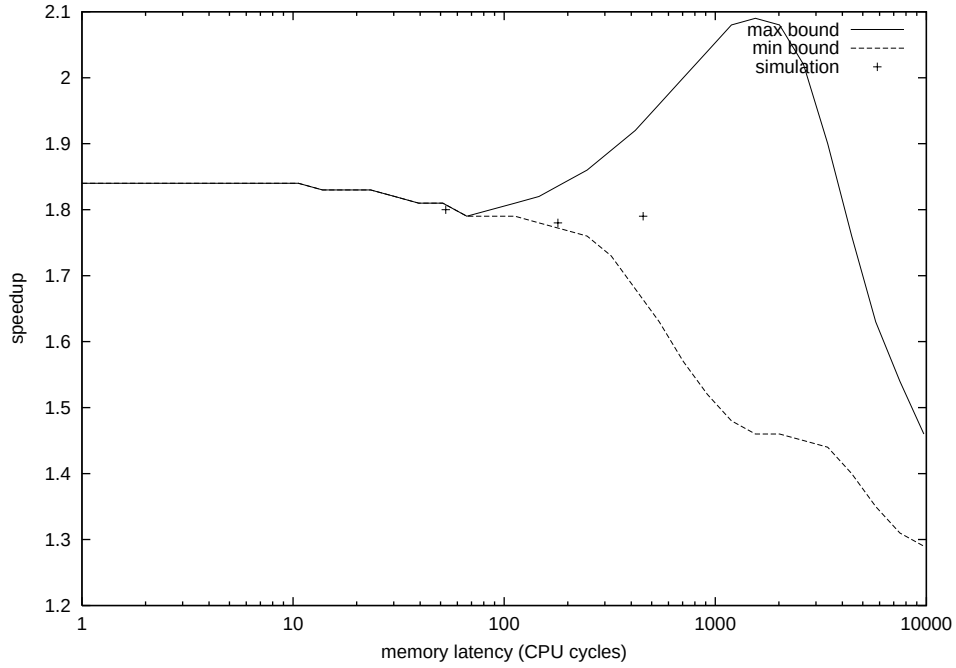

Figure 10.8: Memory latency effect on mpeg2encode speedup

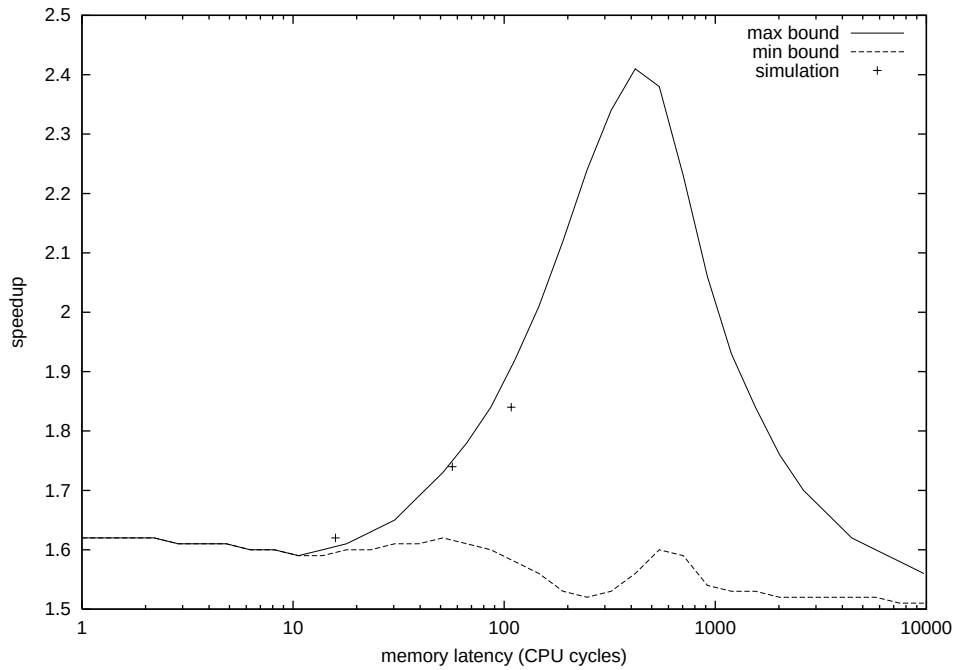

Figure 10.9: Memory latency effect on mpeg2decode speedup

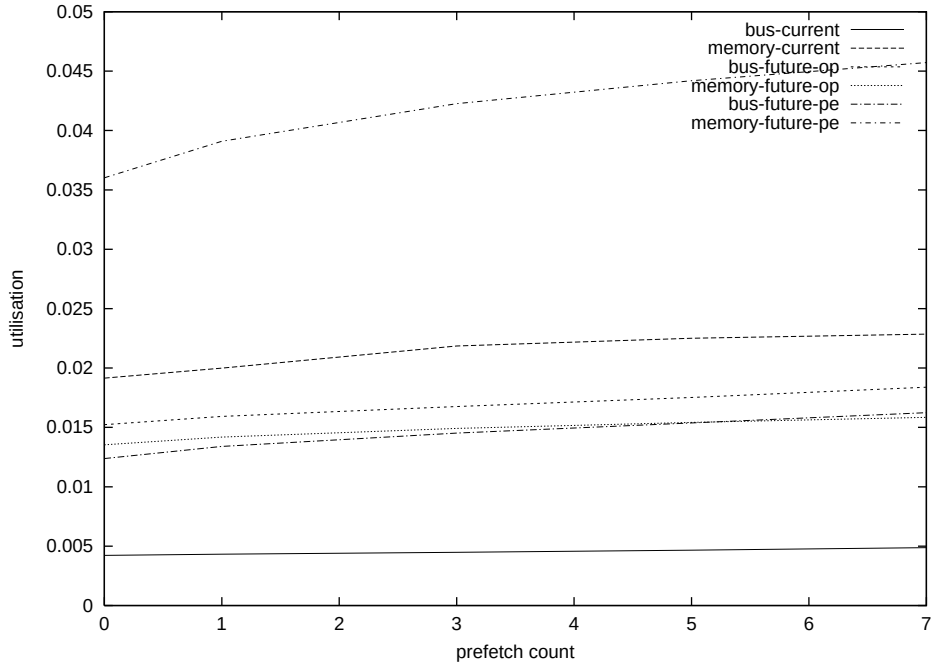

Figure 10.10: mpeg2encode bus and memory channels utilisation

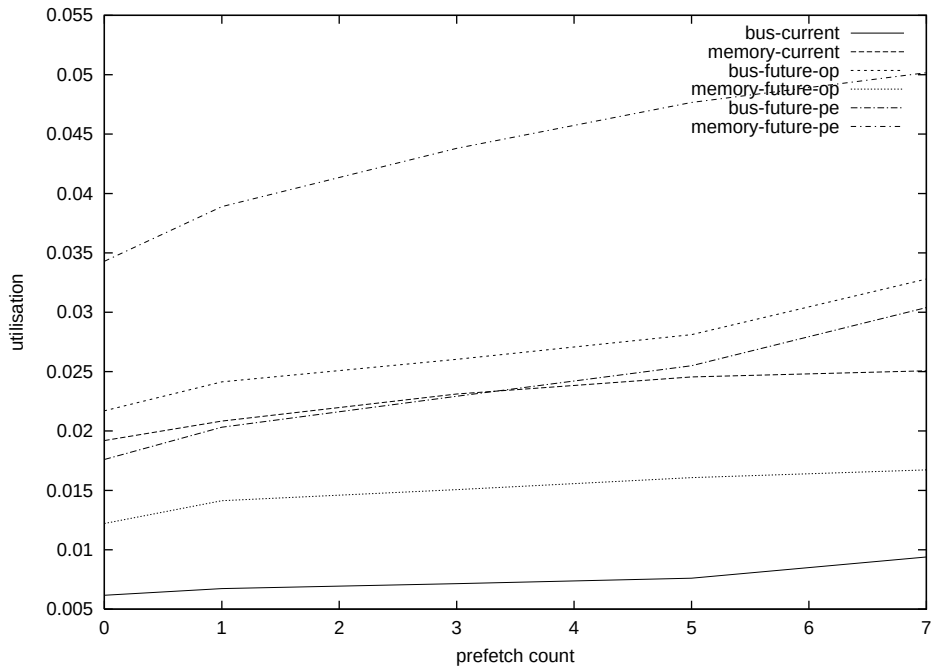

Figure 10.11: mpeg2decode bus and memory channels utilisation

10.2.4 Bus Utilisation

Another important metric is bus utilisation. We show the utilisation for the memory channels and bus in, Figure 10.10, for the mpeg2encode program. Utilisation is less than 4% for bus and memory channels among all technologies. Prefetch count increases the utilisation slightly, however changing the memory technology has a bigger impact (up to 2 times for memory channel utilisation, while prefetch increased the utilisation by a maximum of 27%). Figure 10.11 shows the utilisation data for mpeg2decode. The bus utilisation is slightly higher (50% higher) and follows a similar trend to mpeg2encode.

10.2.5 Other Multimedia Kernels

Although MPEG-2 encode and decode applications have many kernels that exhibit different characteristics, in this section we analyse two kernels from voice recognition and 3d geometric transformation. These kernels are more memory bound than the MPEG-2 applications, and they will thus help us verify the performance for a wide parameter space.

The viterbi kernel implements the following equation:

$$\begin{aligned} Dist(j,t) \;=\; min\{&Dist(j,t-1) + aProb(j,j),\\ &Dist(j-1,t-1) + aProb(j,j-1),\\ &Dist(j-2,t-1) + aProb(j,j-2)\}\\ &+ bProb(j,t) \end{aligned} \tag{38}$$

The variable t is used in an outer loop and j in the inner loop. Only a one-dimensional array is needed for $Dist$ as the previous $t-1$ can be overwritten when the inner loop is reversed; j is set to a maximum value and decremented. Only one outer loop iteration and 40,000 inner loop iterations are executed. These numbers are chosen to stress the cache system and large iterations are typical in this kernel.

The 3D transformation involves a matrix vector multiplication. Points are represented by a vector $V = \left[x\ y\ z\ 1\right]^T$. The transformation is represented by a 4×4 matrix M. If the original vector is V_o and the transformed vector is V_t then

$$V_t \;=\; M \times V_o \tag{39}$$

The calculation is done over 40,000 iterations. A large number of points is typical in 3D applications. We choose this number, again, to be big enough to stress the cache system.

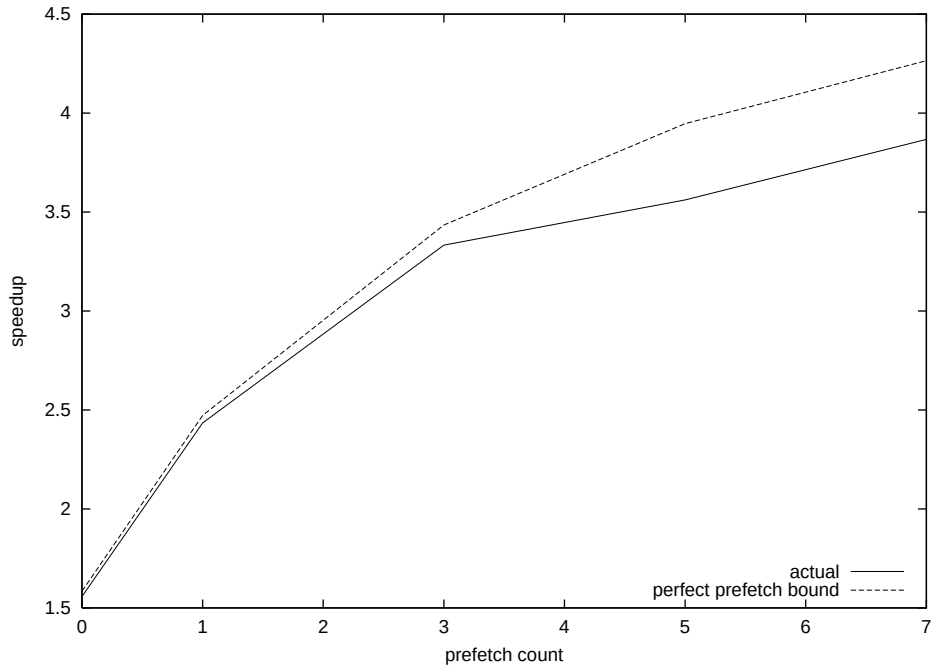

Figure 10.12: viterbi speedup vs. increasing the prefetch count for future_pe

The memory access is highly regular for these applications. Misses are almost perfectly removed by the 2D cache ($\alpha = 0.985$ for viterbi and $\alpha = 0.995$ for 3d calculated for $b = 7$).

Figure 10.12 shows the speedup obtained for the future_pe configuration ($T1$ = 148.42 cycles, $T2$ = 14.68 cycles, $InstCycles$ = 875,293 cycles, $h(0)$ = 58.19 cycles/miss, and $ScalarExecutionCycles$ = 4,923,281 cycles). The 2d-vector contributes 56% and the 2D cache contributes a big speedup of 2.48. The speedup is very close to the upper prefetch bound (91%). For a prefetch count bigger than 3, there is a knee on the speedup. This is due to an increase in the L2 miss ratio (9%).

Figure 10.13 shows the speedup results for the 3d kernel ($T1$ = 634.16 cycles, $T2$ = 42.55 cycles, $InstCycles$ = 2,328,003 cycles, $h(0)$ = 58.18 cycles/miss, and $ScalarExecutionCycles$ = 37,429,646 cycles). The benchmark is more memory bound

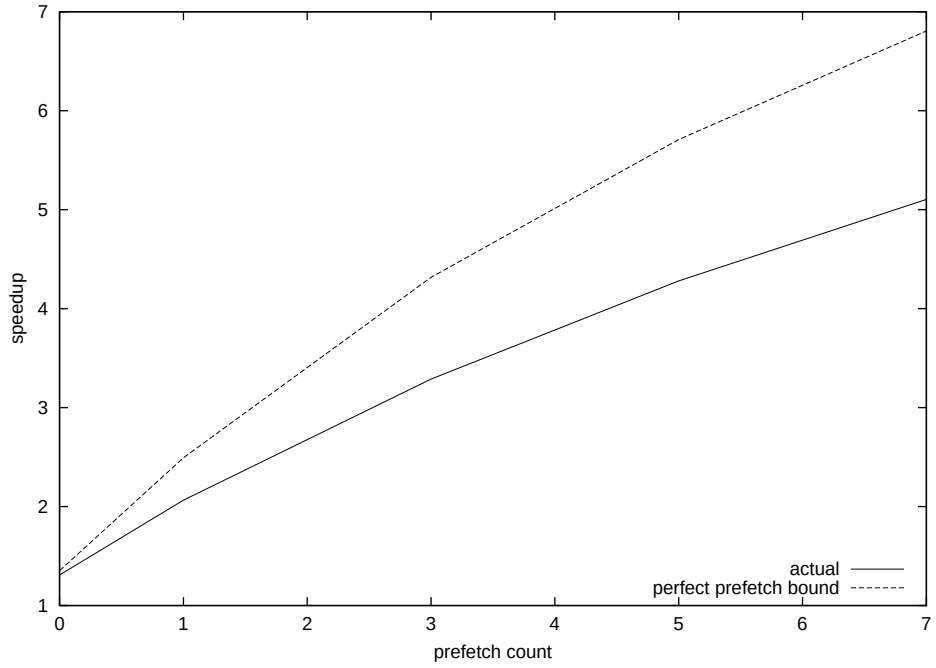

Figure 10.13: 3d speedup vs. increasing the prefetch count for future_pe

than Viterbi and memory channel utilisation reaches 40%. This decreases the effect of prefetching. The 2d-vector contributes a speedup of only 30% while the 2d-vector contributes 3.9 speedup. The results are within 25% of the upper bound.

Figure 10.14 compares the speedup for the three configurations with the speedup bounds obtained in the previous chapter. The model parameters are obtained from a future-pe simulation run. The results are within 7% of the upper bound. The speedup points are shifted to the right compared to the MPEG-2 applications. The future-pe and future-op are close to the peak. The speedup for current is slightly bigger than the bound. This is due to the better misses reduction for this case.

The results for 3d kernel are shown in Figure 10.15. The benchmark is even more memory bound than the viterbi kernel. The speedup points for future-op and future-pe occur in the right-hand region of the curve. In this region the upper bound is limited by the memory bandwidth. The speedup for future-pe is higher than that given by the upper bound. This is due to the fact that the model assumes 1 channel with maximum bandwidth whereas the

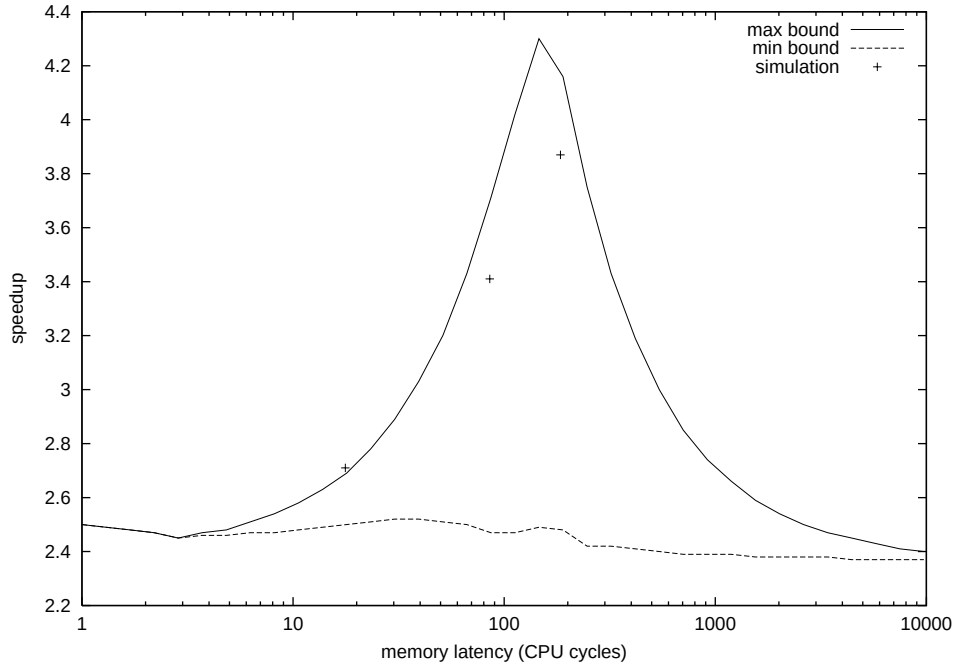

Figure 10.14: Memory latency effect for viterbi

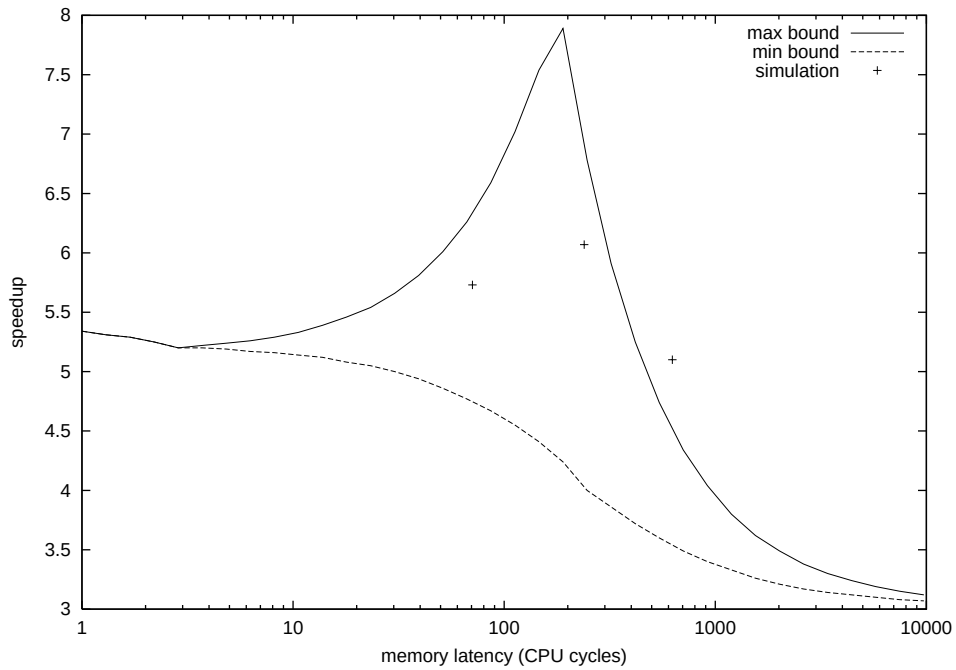

Figure 10.15: Memory latency effect for 3d

simulated configuration achieved 1.49 of the maximum channel bandwidth (4 channels are simulated and the kernel data accesses are distributed evenly among the channels).

10.3 Parallel Threads Evaluation

In the previous section we have analysed the performance of a single processor. A prefetch count of 5 appears to benefit all benchmark applications. In this section we study the effect of a multithreaded multiprocessing environment. We focus on future-op memory parameters as the future-pe is used to approximate the contention on the bus.

It has been demonstrated [86] that the JAMAICA thread support works better than a software-based thread mechanism for fine to medium granularity threads and the finer the grain, the better advantage we get. This was done using mpeg2encode and mpeg2decode programs and without using the vector extensions or the 2D cache. Using the 2d-vector is thus complementary in the sense that it decreases the granularity that can be exploited. Our work in this section extends the analysis to include the proposed instruction set and 2D cache.

The Javar compiler is used to parallelise the conv420to422 and conv422to444 kernels of the mpeg2decode which account for 13% of the execution time. These kernels have the biggest miss ratio and it is interesting to compare the effectiveness of prefetching. For mpeg2encode, the frame_me kernel is parallelised which accounts for 39.4% of the execution time. The kernel is smaller than the one used in the previous analysis for mpeg2encode and is thus likely to achieve better performance in terms of thread management overheads.

10.3.1 Overall Speedup

Figure 10.16 shows the speedup for the mpeg2encode program. The x-axis shows the number of processors and the number of contexts (threads) per processor labelled as <processors#>-<contexts#>. The y-axis is the speedup relative to the case with no prefetching and one processor with one context without the 2d-vector instruction set. The prefetch and noprefetch curves correspond to the case of prefetching switched on (5 prefetches on a cache miss) and off respectively. The number of processors is increased

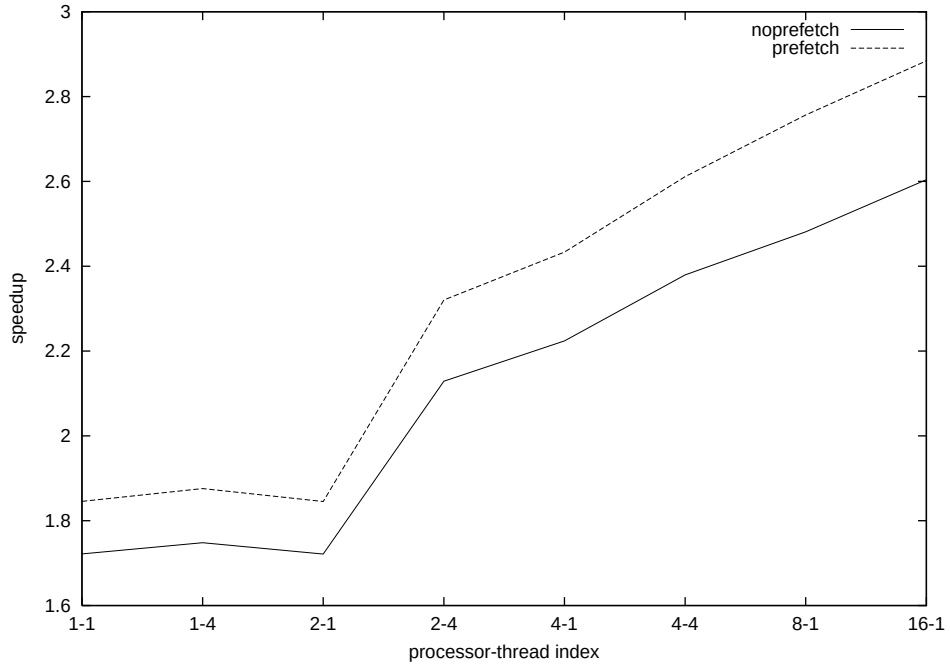

Figure 10.16: Parallel mpeg2encode speedup

by a power of 2. The number of contexts per processor is either 1 or 4. The total number of processors and contexts is constrained to 16 (the closest power of two to the bus capacity which is 19.12 for the most memory bandwidth hungry kernel).

The overhead of parallelism is negligible (less than 0.3%) for both prefetch and noprefetch cases. Prefetching has a gradually improving advantage (ranging from 7 to 11%) with an increasing number of processors and contexts within a processor. Figure 10.17 shows the speedup for the parallelised kernel. Prefetching has a very minor effect. Therefore most of the improvements come from the serial section of the application. The advantage comes from the relative increase in the memory latency.

Figure 10.18 shows the speedup for mpeg2decode. Multithreading has a bigger advantage than the mpeg2encode case as the parallelised kernels exhibit a large miss percentage. Prefetching gave an overall improvement that ranges from 5.8% to 7.4%. The situation here is opposite to the mpeg2encode. The execution time for the parallelised code is smaller (13.38%) but the miss percentage is bigger. The kernel benefited from prefetching (Figure 10.19). Prefetching improves the performance by between 7.1% and 21%.

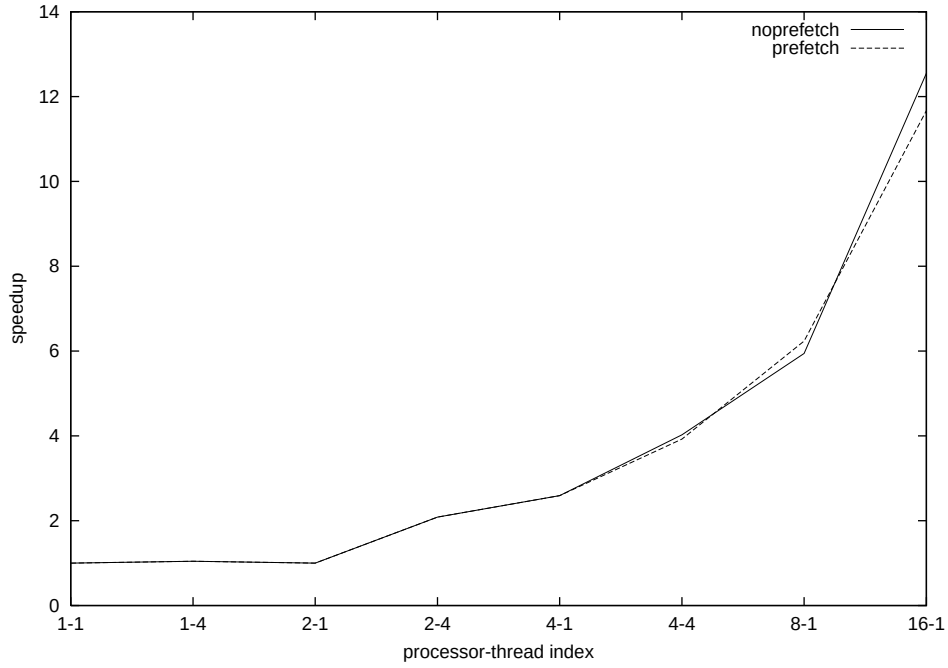

Figure 10.17: Speedup for the parallelised kernel of mpeg2encode

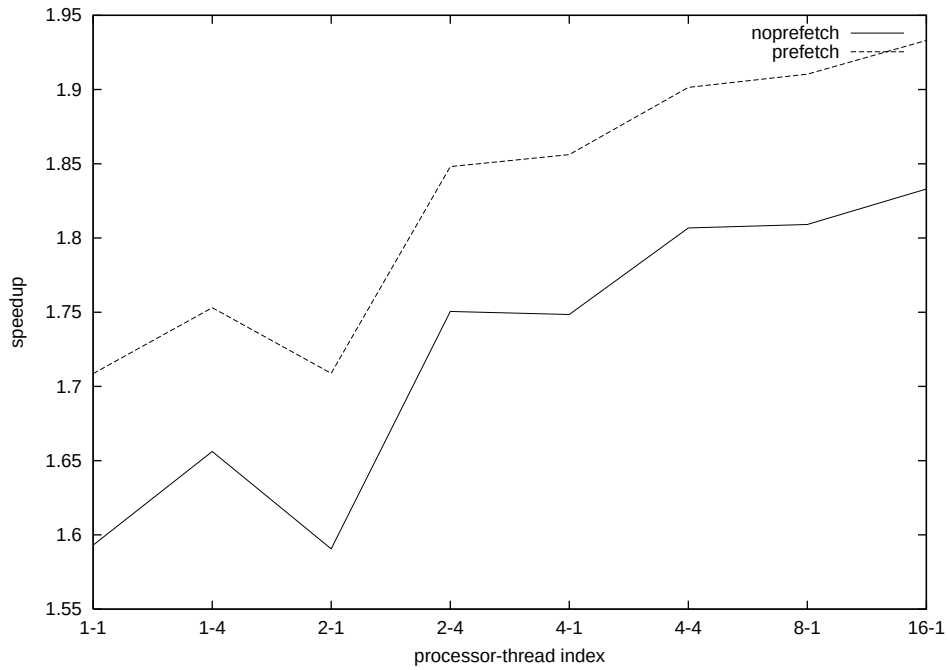

Figure 10.18: Parallel mpeg2decode speedup

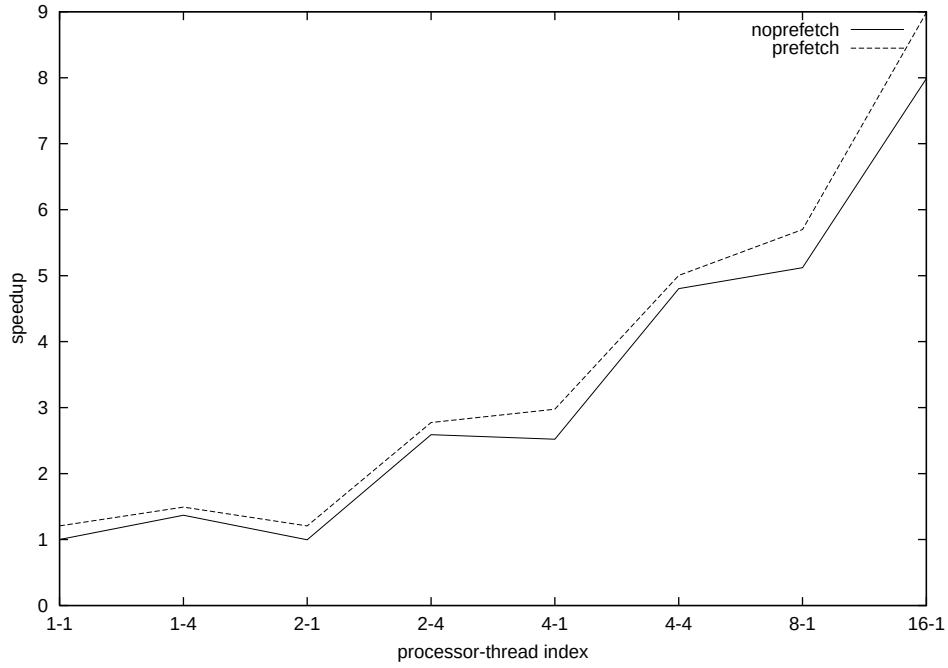

Figure 10.19: Speedup for the parallelised kernel for mpeg2decode

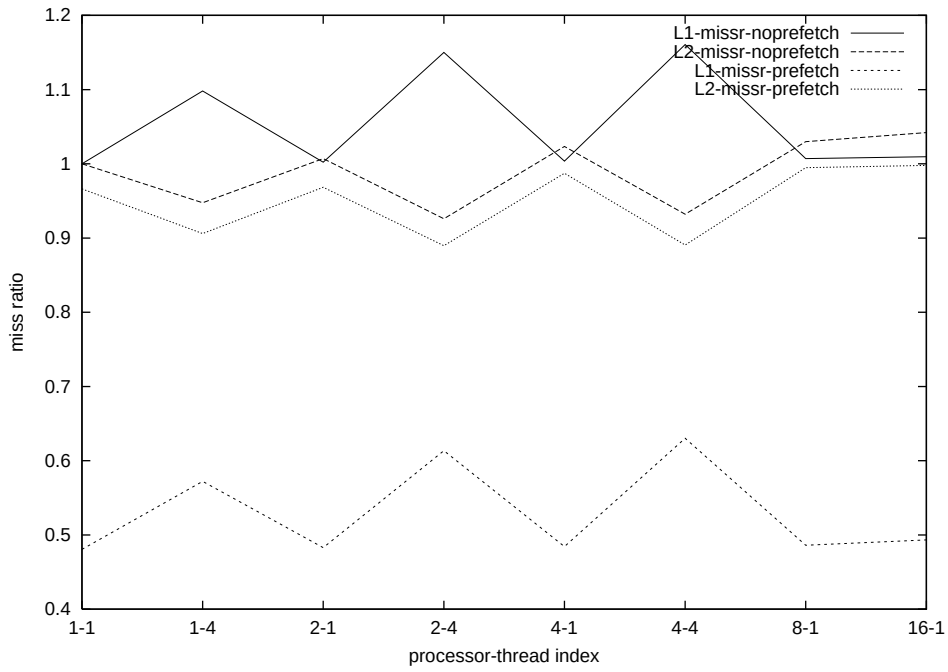

Figure 10.20: mpeg2encode miss ratio

10.3.2 Misses Analysis

The effect of L1 and L2 misses is shown in Figure 10.20 for mpeg2encode The L2 cache misses are important to show the effect of sharing and to account for increased memory traffic due to the change of workload (extra argument passing overheads for parallelisation of the loop). The figure plots the relative miss ratio relative to the noprefetch case at one processor and one context against the processor/context configuration. For the noprefetch configuration, the L1 cache miss ratio increases on using multithreading while the L2 cache miss ratio decreases. L1 misses are likely to be increased due to having multiple threads on the same processor accessing different data sets that are not reused by the other threads. The L2 cache miss ratio decreases as L1 caches are refilled every time a context is switched and basically reusing the same data again from the L2 cache.

For the prefetch configurations, the same effect of multithreading is pronounced. However the L1 cache reduction is shifted by a constant due to the fixed prefetch count used in the experiment and also L2 cache misses benefiting from prefetching.

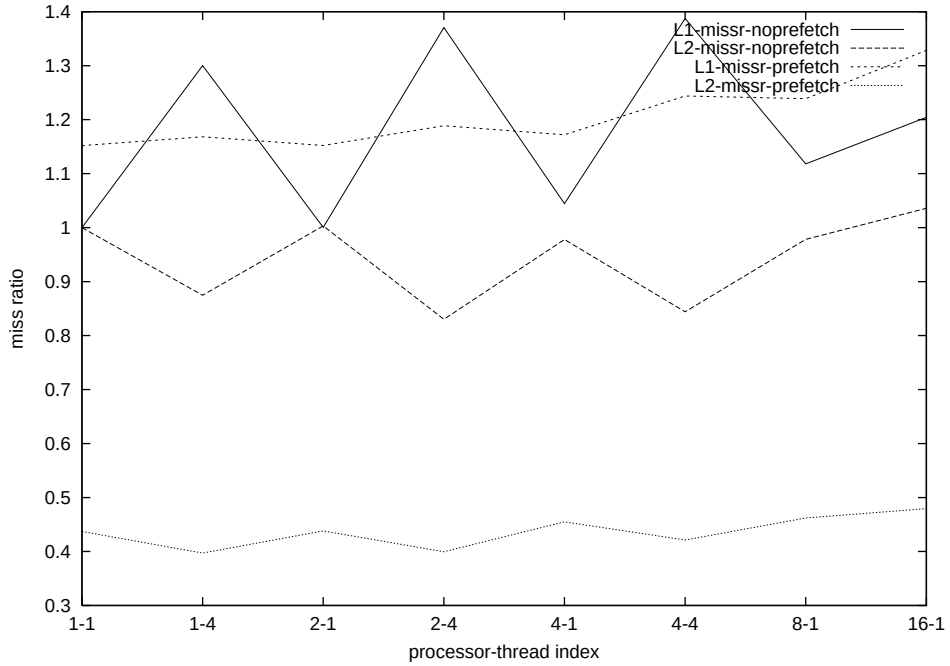

Figure 10.21: mpeg2decode miss ratio

Figure 10.21 shows the misses ratios for mpeg2decode. Without prefetch, multithreading has a similar effect; improving the miss ratio for L2 and degrading the miss ratio for L1. The effect is emphasised here due to the fact that the kernel has a higher percentage of misses than that of the mpeg2encode. For the prefetch case, the multithreading effect is less emphasised, this is mainly due to the fact that misses are already significant for this kernel but are highly optimised by prefetching. Multithreading did not help to decrease the misses. Prefetching tends to space out miss regions and is more likely to overlap the working sets of threads thus increasing reusability and decreasing the extra misses. The L1 miss ratio degrades but this is mainly due to capacity misses (extra argument passing overheads for parallelisation of loops, and low parallelism granularity).

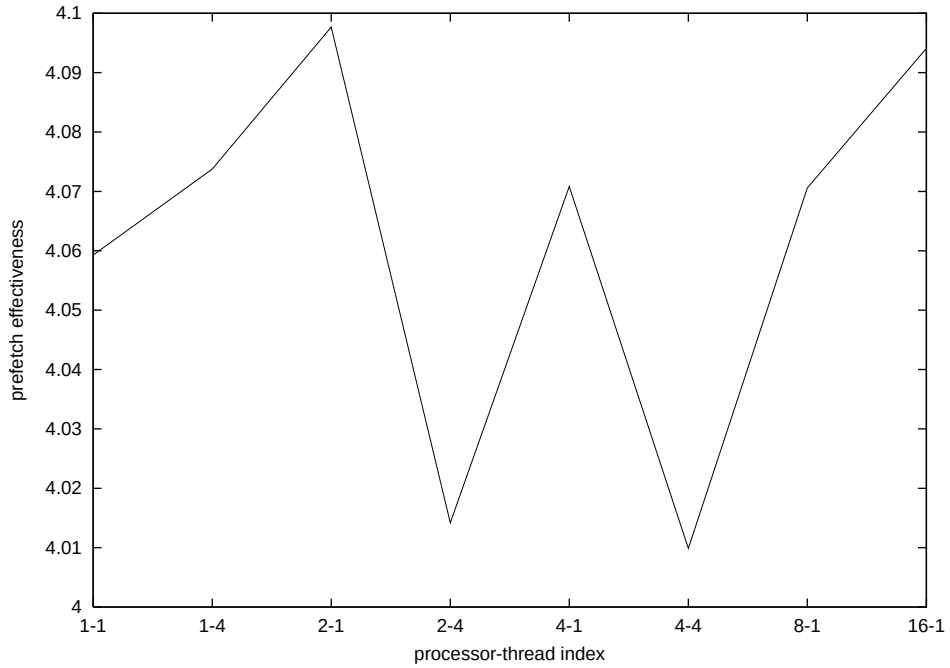

Figure 10.22: mpeg2encode effective prefetch count

Since the L2 cache misses vary with prefetching, we redefine the prefetch effectiveness by substituting ‘misses’ in Equation 33 with the effective misses, such that:

$$\text{effective misses} = \text{L1 cache misses} \times \text{L2 cache miss ratio} \quad (40)$$

The effective prefetch count for mpeg2encode is shown in Figure 10.22. The values are high. They oscillate around 4, given that we are prefetching 5 on a cache miss. Multithreading decreases the effectiveness of prefetching, this is due to the increased misses in multithreading.

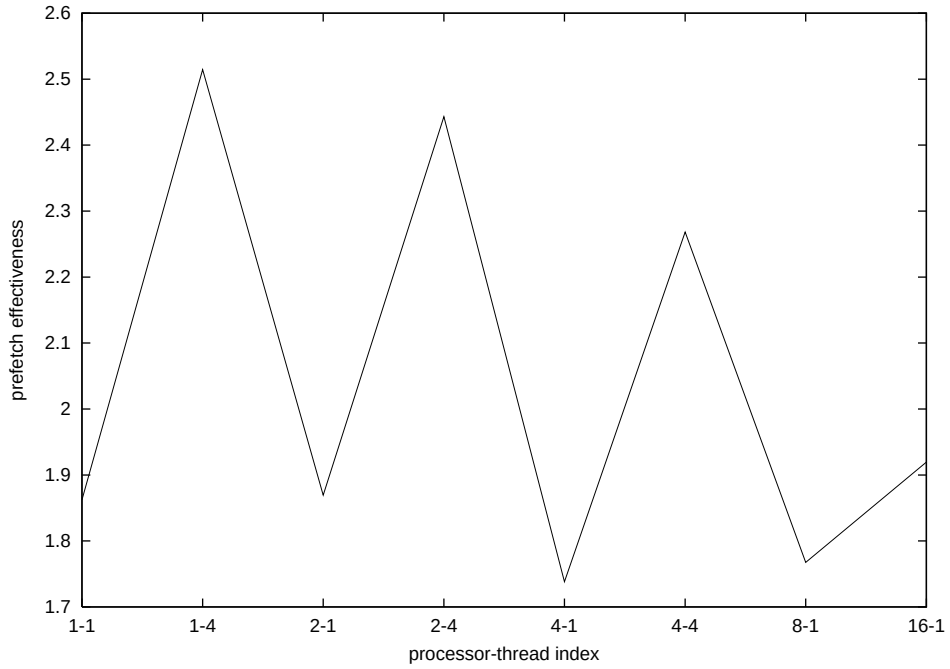

Figure 10.23: mpeg2decode effective prefetch count

Prefetch effectiveness for mpeg2decode is shown in Figure 10.23. The effective prefetching is about 2 (efficiency is around 50%). In contrast to the mpeg2encode case, it is worth noting that effectiveness is increased with multithreading, this illustrates the misses improvement discussed above.

10.3.3 Bus Utilisation

Bus and memory utilisation for mpeg2encode are shown in Figure 10.24. Utilisation is less than 2.8%. Utilisation is higher when multithreading is used, this is due to the high miss ratios. Bus utilisation is higher than memory utilisation for both prefetching and no prefetching. For mpeg2decode (Figure 10.25), the situation is the same, bus utilisation is higher than memory utilisation. Multithreading tends to have little effect on increasing the

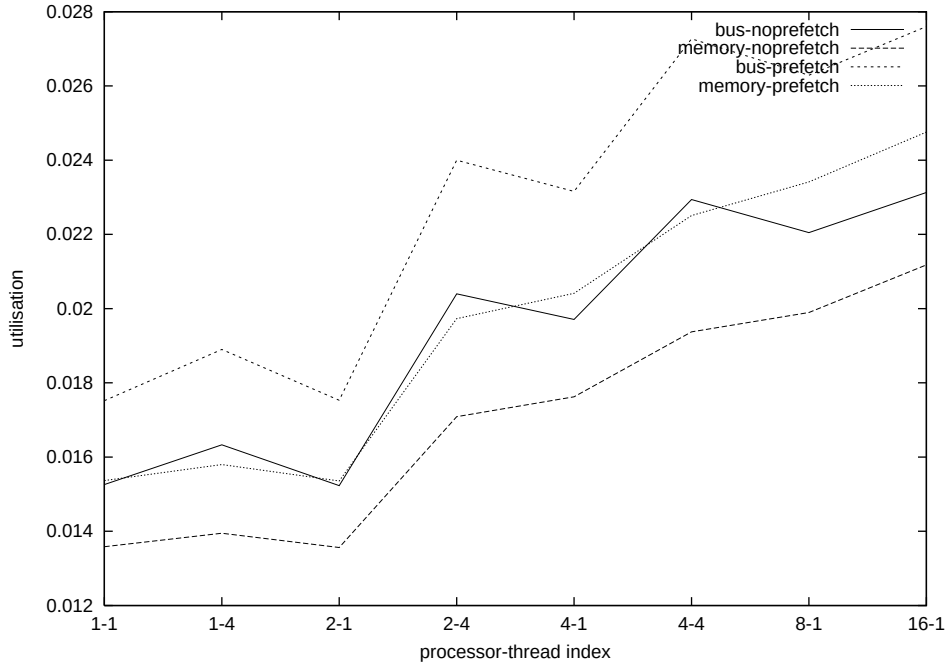

Figure 10.24: mpeg2encode bus and memory utilisation

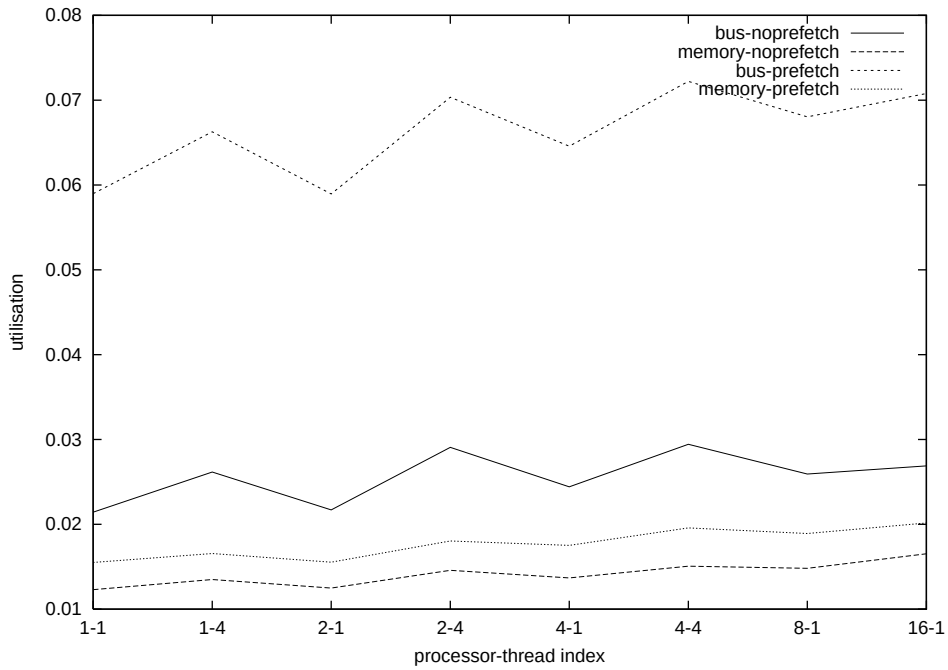

Figure 10.25: mpeg2decode bus and memory utilisation

utilisation for both memory and bus. However with prefetching, multithreading increases both more significantly. This is likely due to less cancellation of the prefetch requests as a context is switched on a cache miss. This gives an opportunity for the prefetch requests of the sleeping context to be issued without further interruption.

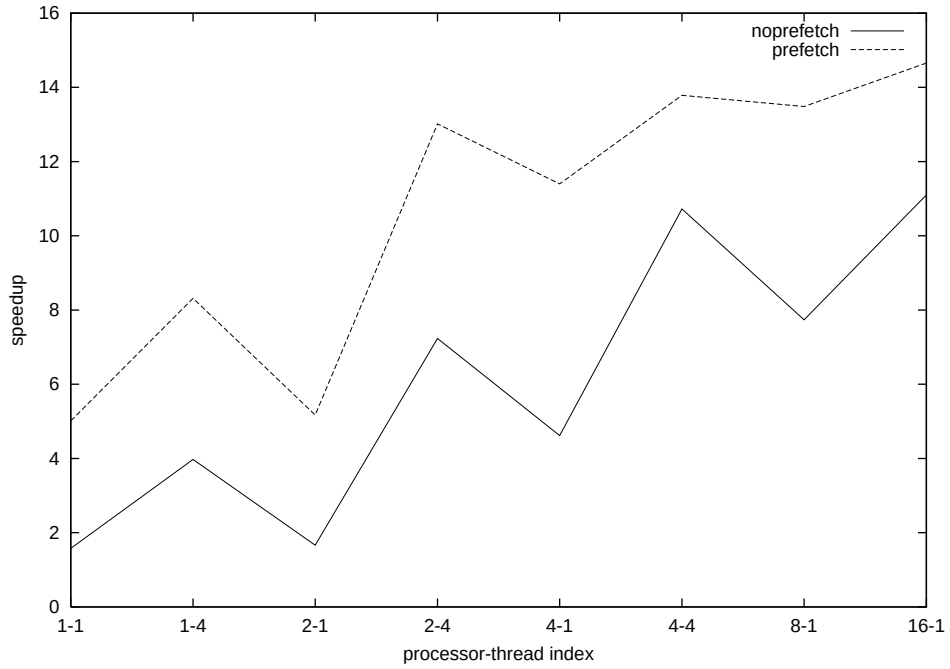

Figure 10.26: Speedup for viterbi kernel

10.3.4 Other Multimedia Kernels

The viterbi kernel has loop carried dependencies which cannot be parallelised using Javar like the MPEG-2 applications. However, to show the effect of bus contention, we produced a multiprogramming workload, where every task was an independent viterbi kernel. We have done the same to 3d, as the overhead of dividing the tasks will decrease the benefit of automatic address generation of the vector instruction set (vector utilisation will be required for every potential parallel section).

Figure 10.26 shows the speedup for the viterbi kernel. Prefetching gave a speedup of 3.2 for 1-1 and degrades to 32% for 16-1. Memory channel utilisation reaches 80-90%

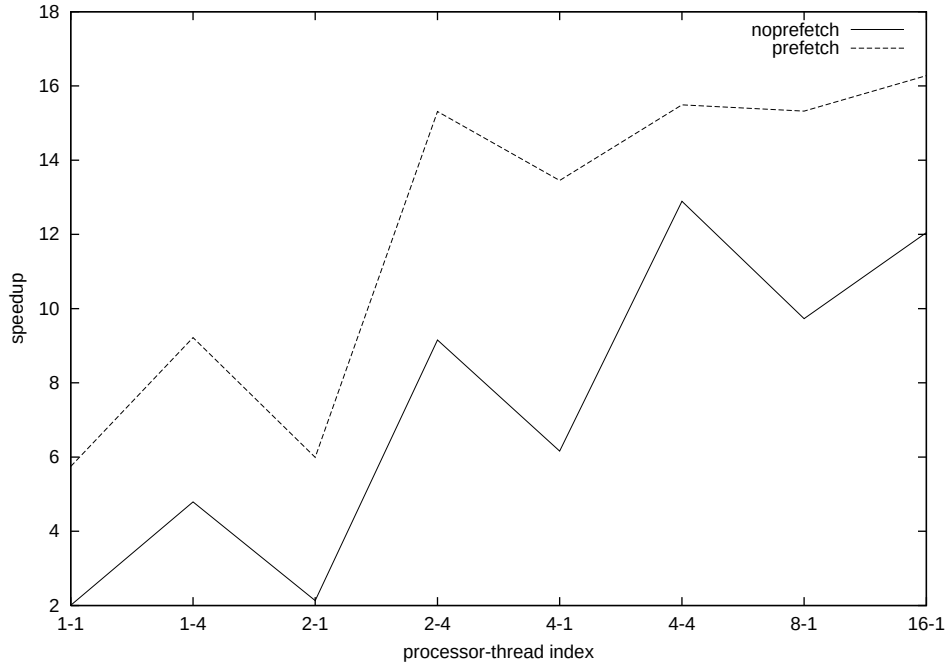

Figure 10.27: Speedup for 3d kernel

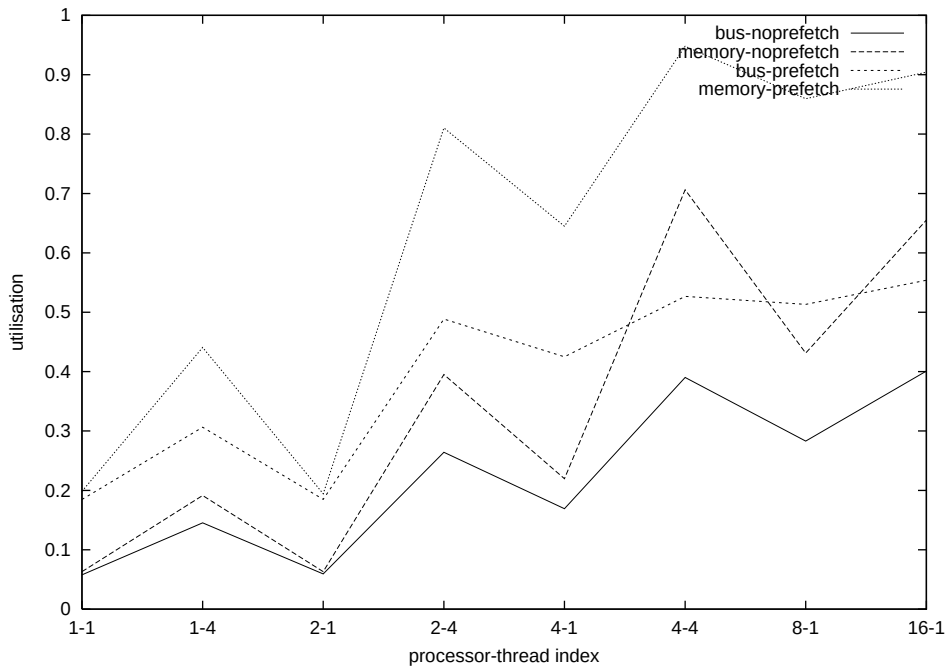

Figure 10.28: viterbi utilisation

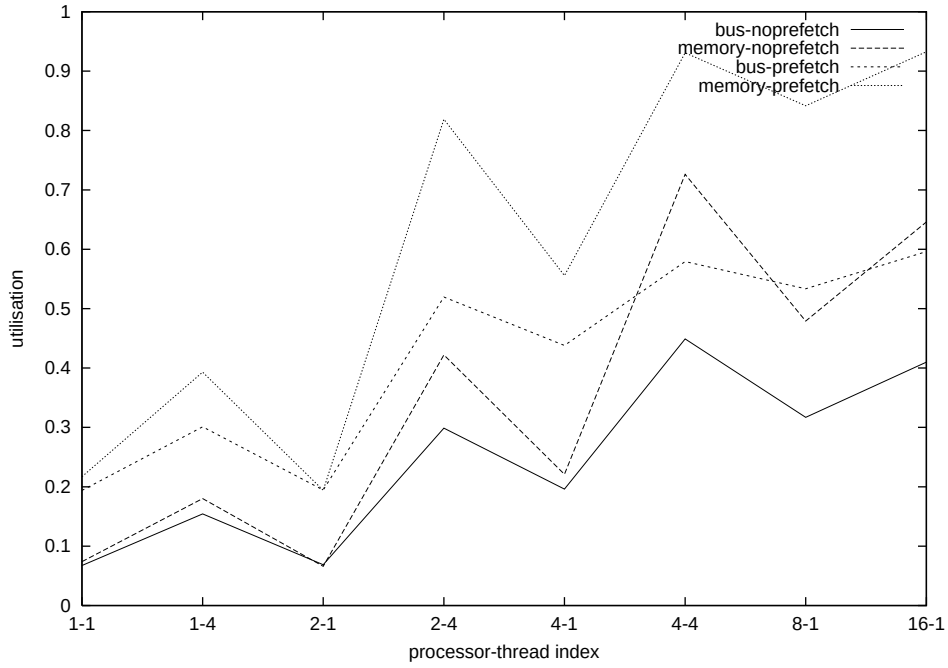

Figure 10.29: 3d utilisation

(Figure 10.28) beyond the 4-4 configuration which limits the benefits of prefetching. The multithreading effect is significant (speedup of 3 on average).

Figure 10.27 shows the speedup for the 3d kernel. The speedup curves shows a similar trend to the viterbi kernel. The prefetch gave a speedup range of 1.2 to 2.9. Memory channel utilisation is high, 80-90% utilisation (Figure 10.29) was reached starting beyond the 4-4 configuration.

10.4 Summary

This chapter studies the overall performance of the JAMAICA system with emphasis on the 2d-vector and the 2D cache techniques. A uniprocessor is first analysed. The effect of prefetching is analysed for three different memory configurations reflecting current, and future technologies (both an optimistic and another pessimistic prediction are used for the future). The 2d-vector and 2D cache improvements are demonstrated and compared to the analytical model developed in Chapter 9. The results seemed in good accordance with the model.

The other half of the chapter studied a multiprocessing configuration. The interaction between multiprocessor, multithreading, and prefetching is examined. Prefetching seemed to aid multithreading to reduce the memory latency.

The overall performance can achieve up to 3 times speedup using all acceleration techniques for an optimistic future configuration.

Chapter 11: Conclusions

11.1 Summary

In this thesis we have investigated hardware support for multimedia Java applications. The combination of Java and multimedia provides different execution characteristics that give an opportunity for hardware support. The thesis developed two main novel architectural ideas to support multimedia. These are the provision to support a submatrix addressing mode in the instruction set, and combining subword processing with vector architecture. The other idea is developing a cache prefetching technique to capture a new type of data access locality (two dimensional locality). Both techniques have shown a considerable benefit for MPEG-2 video applications.

For supporting Java, an important result has been the demonstration that using a register-based instruction set is better than a stack-based one. The latter might be motivated by the stack nature of the Java intermediate language, bytecode. Even with the use of folding techniques, it does not provide the same performance as the register-based instruction set.

The thesis starts by examining the Java support. It considers the Java support independently from multimedia applications, and uses the SPECjvm98 benchmark suite for performance evaluation. Chapter 2 analyses the execution characteristics of the benchmark and observes that the operation stack and method call overheads are significant. A register-based instruction set with register-windows is likely to overcome these limitations. However, there is an extra cost of compilation. That cost will not be required if a stack-based instruction set is used. The chapter models both instruction set models as well as the common dynamic instruction folding techniques used in Java processors, and a register-windows technique for the register-based instruction set. The results have shown that even using a simple register allocation algorithm, a significant portion of the stack-overhead is removed. Moreover, register windows significantly decrease the method calling overhead with modest windows requirements. Chapter 3 goes into more detailed analysis by developing a Java translation system, and tuning the

register allocation to exploit the register-windows mechanism. The results verify the higher-level analysis.

The thesis then examines the multimedia aspects of the design. A set of multimedia kernels from MPEG-2 video applications is examined in Chapter 4. The kernels are shown to be operating on small data types (8- and 16-bits). Data is accessed in a regular submatrix way, and the operations are mostly simple integer ones. This observation motivated the design of a vector instruction set with a submatrix addressing mode.

The literature is surveyed in Chapter 5. Approaches to supporting multimedia range from DSP processors to extensions to general-purpose processors. Also there are special purpose ideas that are highly optimised for multimedia processing such as stream processing. To help explore the wide variety of design ideas, we classified them with respect to three main architectures: superscalar, vector, and multiprocessors. Of these approaches, the vector is commonly used in all multimedia-support hardware. An elegant approach that is used in general-purpose processors (and now in DSP) is subword processing. This exploits the wide datapaths in general-purpose processors and uses registers as small vectors.

It was thought to be interesting to combine subword processing and a vector instruction set with a submatrix addressing mode. This is studied in Chapter 6. To help investigate the trade-offs among changing word sizes, vector length, addressing modes, and alignment, we modelled the performance of the instruction set analytically. The model relies on the simple nature of multimedia kernels and scales the number of instructions according to the model parameters. This shows a significant improvement for our instruction set. The main advantage comes from removing the loop control and address generation overheads.

Another idea that has been cited in the literature is the 2D spatial locality of multimedia applications and the fact that this does not work well with existing cache architectures. This topic is pursued in Chapter 7. A simple cache prefetching technique is developed and shown to remove a significant fraction of the cache misses. The technique also does not require a special hardware structure, and reuses the existing cache architecture.

The design for the architecture is then described in Chapter 8. The architecture is designed at a level of detail suitable for cycle accurate simulation.

In order to limit the parameter space for simulation, Chapter 9 models analytically the interaction between the prefetch mechanism and the CPU execution. The model is used to derive optimal parameters. The analysis is extended, and verified by simulation in Chapter 10. Moreover, the simulation analysed the JAMAICA multiprocessor and multithreading execution. The proposed techniques are shown to contribute to the overall performance.

11.2 Putting Results on Context

The context of the work has been a single-chip multiprocessor with multithreading. This affected some of the underlying design assumptions. The fact that multiprocessors share the bus motivated the inclusion of multimedia extensions inside each processor. The cache prefetching technique is incorporated inside L1 cache rather than L2 cache. Also the size of the L1 cache and its geometry is determined by the fact that a simple processor design is used. The write-back and write allocation cache policy are motivated by the desire to decrease the bus utilisation per processor as this is important in shared-bus multiprocessors.

The fact that the processors are multithreaded requires the vector instructions to be restartable and interruptible. The implementation of such instructions closely followed that of scalar instructions. This limited the implementation of more complex operations, such as the sum of absolute differences which would result in a long pipeline and higher cost of context switching. In addition, this limits the amount of vector state per context. This resulted in having 8 vector registers each containing 8 elements per context. Traditional vector processors usually have more registers; the Cray 1, for instance, has 8 registers each containing 64 elements giving 512 registers in total. JAMAICA has the same number of registers; for each context there are 8 register-windows each containing 8 registers, and there are 64 registers used in the vector register file. This gives a total of 128 registers per context. Since four contexts are supported within a processor, the total number of registers is 512.

Multithreading is a technique used to hide the memory latency by interleaving the execution of many independent threads. It is interesting to show that the cache prefetching technique is complementary to multithreading and it adds to its benefit. This suggests that the technique might be useful in other architectures that issue independent threads such as simultaneous multithreading or even more elaborate superscalar architectures that seek to extract parallelism at run-time.

There are other prefetching techniques that are based on software that have not been investigated. These might help to decrease the memory latency. However scheduling these prefetch instructions might not be useful for unpredictable memory access such as the motion prediction kernels in MPEG-2 video applications. The proposed prefetch technique still benefits from the software determining what to prefetch. Scheduling is done by the hardware with no complexity on the compiler/programmer.

The vector architecture presents opportunities to do operations that have been otherwise expensive in scalar architectures. One worth noting is the alignment operation. This is an expensive operation as two memory accesses are required for every unaligned memory access, together with alignment hardware for merging the values. Vector processing has the benefit that memory access is streamed, and thus only one extra access is required for the whole stream (with no row changes), moreover the cost of alignment can be amortised by the high degree of pipelining in the vector pipeline.

To aid our analysis, we have developed analytical models to explore other design options. The analysis indicated that the vector instruction set has at least the same performance as a subword based architecture with double the word size. Moreover, the cache prefetching technique gave performance similar to a conventional cache with double and sometimes quadruple the cache size.

The choice of MPEG-2 benchmarks covers an important range of multimedia applications. Furthermore, the simulation analysis used a 3D transformation kernel, and a Viterbi decoding kernel used in voice recognition. Based on these applications floating-point processing was not required. Although for future 3D applications, floating-point processing may be needed. The vector architecture has demonstrated in the past its efficiency for floating-point processing in terms of the ease of pipelining control, and

hiding the operation latency by overlapping many floating-point operations. It is probable therefore that the architecture will be helpful, but a large machine word size will be required for subword parallelism. However, the proposed vector architecture would require a smaller word size than a subword-based architecture, as for integer processing, the architecture achieves better performance, on average, than a subword architecture with double the machine word size.

On a higher level, multimedia processing can be supported by having special-purpose hardware components such as 3D acceleration cards. However, we believe that, with the increasing power of microprocessors, the special-purpose solutions will be less emphasised. This happened in the case of supporting floating-point in the past, for instance. Separate coprocessors were developed, and eventually integrated into microprocessors.

11.3 Future Work

This research opens several avenues for further exploration. Future work can be done on both software and hardware.

On the software side, an important topic is the dynamic vectorisation of multimedia kernels. The multimedia kernels examined have shown high regularity of data-access and abundance of data-level parallelism. For the MPEG-2 decode, extrapolating kernel conv422to444 for example, there is of the order of 100,000 independent loop iterations.

The compilation system is currently a prototype system that relies on static compilation. Future work is needed to have a full dynamic compilation system, integrating dynamic parallelism. Also, using method in-lining is likely to decrease the benefit of register-windows for scalar application with large number of method calls.

On the hardware side, supporting floating-point applications is a subject for future work. This would be required for intensive 3D applications. Also, increasing the machine word size would be essential.

Support for real-time scheduling is essential for running different multimedia threads. There is also a chance for the hardware to support this by, for example, employing priorities for the thread distribution mechanism.

The ultimate aim would be to build a VLSI implementation of the processor and analyse its cost and performance.

Appendix A: Java Compilation Details

Code Generation

Jtrans is a prototype compilation system that is designed to provide fast compilation. Most of the bytecode non stack operations are mapped onto native instructions. Complex bytecodes are handled by native functions in the runtime library. The major task of jtrans is removing the stack operations using the Cacao algorithm.

Object and Class Layouts

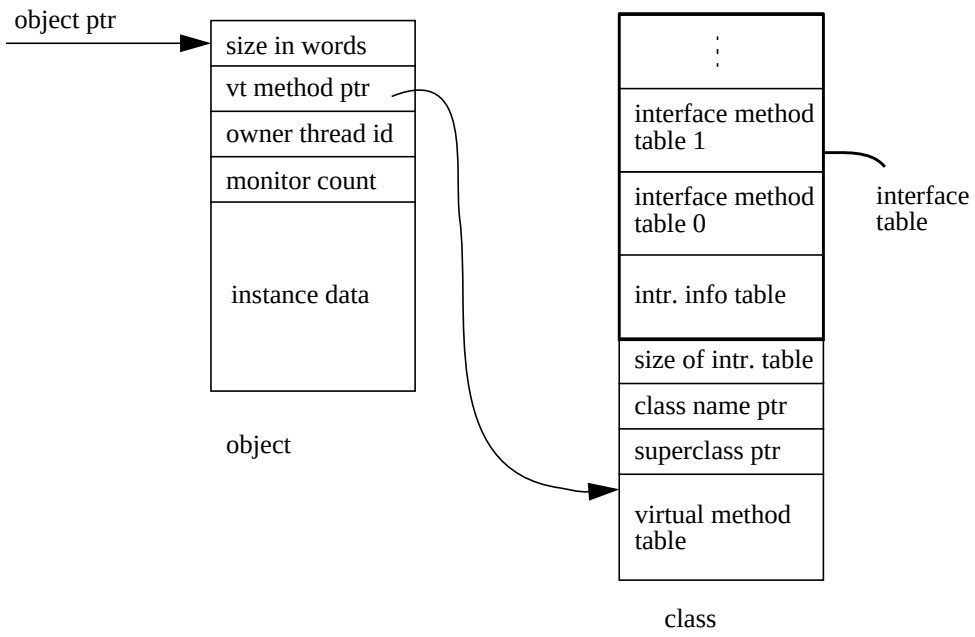

Figure A.1: Object and class layout

Figure A.1 shows the object and class layout used in jtrans. An object pointer is used to directly access an object instead of indirect access via a handle to speedup accesses. However this might affect the performance of garbage collection.

The first word in an object is the size (in words). The second word is a pointer to the virtual method table in the class structure. The virtual method table is an array of pointers to the methods inherited from the ‘Object’ down to the current class. A virtual method call would require two memory indirections.

The ‘owner thread id’ and ‘monitor count’ are used for implementing monitors. When entering a monitor, the owner thread is set to current thread number, and the monitor count is incremented. Threads would wait till the monitor count reaches zero before entering the monitor. On a monitor exit the ‘monitor count’ is reset to zero. The rest of the object holds instance fields.

The class information includes information about the object’s class. Above the virtual method table is a pointer to the superclass structure. This is used for run-time type checking. The next word is a pointer to the class name string, this is mainly used to implement reflection. The rest of the class information holds interface information.

Interface Implementation

A class can only inherit from exactly one other class. However, a class may implement many interfaces. Resolving virtual methods is simple as a method will always have the same location in the creator class and all its subclasses. For interfaces, it might happen that the same method appears in different locations for different classes that implement its interface.

The way jtrans implements interfaces is for every class that implements one or more interfaces to have the methods for each interface kept in a separate interface method table. Each table lists the methods defined by the interface in the order that interface implemented or inherited them from other interfaces. For every entry, the actual method address is stored. If the class does not implement the interface method then the address is the same as its superclass, otherwise, the address is set to the method in the class.

When a method is invoked on an interface, the interface method table is located by searching the interface info table where each entry in the table holds an interface identifier

and an offset to the interface method table. Once located, the order of the method would always be contant and thus can be determined statically.

The pseudo-code for the algorithm is given below.

```
Method name: LoadInterfaces
input:       class
output:      interface table
operation:   constructs the interface table for a given class
{
  for all classes in the class hierarchy starting from the top most {
    for all interfaces in current_class {
      CreateInterface(interface_table, current_class)
    }
  }

  for all methods in class {
    if current method is in any interface_method_list {
      modify that entry address to point to current method
  }
}


Method name: CreateInterface
input:       interface_table
input:       class
output:      updated interface_table
operation:   constructs a interface method list for the given class
{
  for all interfaces in class {
    if the current interface is not in the interface_table {
      CreateInterface(interface_table, current_interface)
      Append current_interface interface_method_list to class
interface_method_list if an entry does not already exist
      Append current_interface tn the interface_table
    }
  }

  for all methods in class {
     Append current_method to the class interface_method_list if new
  }
  Insert class into interface_table
}
```

Appendix B: Vector Instruction Set

Hardware name	Software name	Description
0	VL	Vector length
1	VI	Vector index
2	V0X	Column index for vector 0
3	V0Y	Row base index for vector 0
4	V1X	Column index for vector 1
5	V1Y	Row base index for vector 1
6	V2X	Column index for vector 2
7	V2Y	Row base index for vector 2
8	V3X	Column index for vector 3
9	V3Y	Row base index for vector 3
10	V4X	Column index for vector 4
11	V4Y	Row base index for vector 4
12	V5X	Column index for vector 5
13	V5Y	Row base index for vector 5
14	V6X	Column index for vector 6
15	V6Y	Row base index for vector 6
16	V7X	Column index for vector 7
17	V7Y	Row base index for vector 7
18	VA	Vector access (set to 1 if the current context accesses any vector register)
19	PS	Prefetch stride
20	CF	Control Flags. The LSB turns transpose mode on/off

Table B.1: Control registers

For the following tables, labels starting by ‘V’ are vector registers (such as VA, VB, and

VC), labels starting by ‘S’ are scalar registers (such as SB and SC)

Name	Mnemonic	Syntax	Operation
Vector add unsigned integer [b,h,w] saturate	vaddubs vadduhs vadduws	VA, VB, VC	Split add the first VL elements of VA to the first VL elements of VB and store the results in the first VL elements of VC. The addition is done in saturate mode. If the result is > (2^n-1), saturate to (2^n-1), where n=b,h,w. For b, byte, integer length = 8 bits = 1 byte, add 4 unsigned integers from VA to the corresponding 4 unsigned integers from VB. This is repeated for all the first VL elements in VA and VB. For h, half word, integer length = 16 bits = 2 bytes, add 2 unsigned integers from VA to the corresponding 2 unsigned integers from VB. This is repeated for all the VL elements in VA and VB. For w, word, integer length = 32 bits = 4 bytes, add 1 unsigned integer from VA (word) to the corresponding unsigned integer fromVB. This is repeated for all the first VL elements in VA and VB.
Vector add unsigned integer [b,h,w] modulo	vaddub vadduh vadduw	VA, VB, VC	Same as vector add unsigned integer [b,h,w] saturate except modulo arithmetic is used instead of saturation.
Scalar-Vector add unsigned integer word modulo	svadduw	VA, SB, VC	Add the first VL elements (32-bit) of VA to SB and store the result into the corresponding elements in VC.

Table B.2: Vector integer instructions

Name	Mnemonic	Syntax	Operation
Vector subtract unsigned integer [b,h,w] saturate	vsububs vsubuhs vsubuws	VA, VB, VC	Split subtract the first VL elements of VB from the first VL elements of VA and store the results in the first VL elements of VC. The subtraction is done in saturate mode. If the result is <0, it saturates to 0 corresponding to b, h, w. For b, byte, integer length = 8 bits = 1 byte, subtract 4 unsigned integers from VB from the corresponding 4 unsigned integers from VA. This is repeated for all the first VL elements of VA and VB. For h, half word, integer length = 16 bits = 2 bytes, subtract 4 unsigned integers from VB to the corresponding 2 unsigned integers from VA. This is repeated for all the first VL elements of VA and VB. For w, word, integer length = 32 bits = 4 bytes, subtract the whole element VB (word) from VA. This is repeated for all the VL elements of VA and VB.
Vector subtract unsigned integer [b,h,w] modulo	vsubub vsubuh vsubuw	VA, VB, VC	Same as vector subtract unsigned integer [b,h,w] saturate except modulo arithmetic is used instead of saturation.
Scalar-vector subtract unsigned integer modulo	svsubuw	VA, SB, VC	Subtract SB from the first VL elements of VA and store the result into the corresponding elements in VC.

Table B.2: Vector integer instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector sum across unsigned [b,h,w] modulo	vssumub vssumuh vssumuw	VA, SB, SC	Add the integers of first VL elements from VA to the 32-bit integer in SB and store the sum in SC. For b, byte, integer length = 8 bit = 1 byte, add 4 unsigned integers from VA to the 32-bit integer in SB. This is repeated for all the VL elements in VA. For h, half-word, integer length = 16 bits = 2 bytes, add 2 unsigned integers from VA to the 32-bit unsigned integer in SB. This is repeated for all the first VL elements in VA. For w, word, integer length = 32 bits = 4 bytes, add 1 unsigned integer from VA to the 32-bit unsiged integer in SB. This is repeated for all the first VL elements in VA.
Vector multiply odd signed integer [b,h]	vmulob vmuloh	VA, VB, VC	For each the first VL elements of VA and VB, split multiply odd numbered integers and store the results into VC (double integer size). For b, byte, integer length = 8 bits = 1 byte, multiply 2 odd numbered integers of VA to the corresponding 2 odd numbered integer of VB and store the results as 16 bits = 2 bytes integer length into VC. This is repeated for all VL elements in VA and VC. For h, half word, integer length = 16 bits = 2 bytes, multiply 1 odd numbered integers of VA to the corresponding 1 odd numbered integer of VB and store the results as 32 bits = 4 bytes integer length into VC. This is repeated for all VL elements in VA and VC.

Table B.2: Vector integer instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector multiply even signed integer [b,h]	vmuleb vmuleh	VA, VB, VC	For each the first VL elements of VA and VB, split multiply even numbered integers and store the results into VC (double integer size) For b. byte, integer length = 8 bits = 1 byte, multiply 2 even numbered integers of VA to the corresponding 2 even numbered integer of VB and store the results as 16 bit = 2 byte integer length into VC. This is repeated for all the first VL elements in VA and VC. For h, half word, integer length = 16 bits = 2 byte, multiply 1 even numbered integers of VA to the corresponding 1 even numbered integer of VB and store the results as 32 bit = 4 byte integer length into VC. This is repeated for all the first VL elements of VA and VC.
Vector multiply signed	vmulw	VA, VB, VC	Multiply every element in VA (first VL elements) to the corresponding on in VB, and store the results into the corresponding element in VC.
Scalar-vector multiply signed	svmulw	VA, SB, VC	Multiply every element in VA (first VL elements) to SB, and store the results into the corresponding element in VC.
Vector multiply accumulate signed	vmacw	VA, VB, VC	Multiply every element in VA (first VL elements) to VB, add and store the result into the corresponding element in VC.
Scalar-vector multiply sccumulate signed	svmacw	VA, SB, VC	Multiply every element in VA (first VL elements) to SB, add and store the result to the corresponding element in VC.
Vector logical and	vand	VA, VB, VC	AND the contents of the first VL elements of VA to the corresponding first VL elements of VB and store the result into the corresponding element in VC.

Table B.2: Vector integer instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector logical or	vor	VA, VB, VC	OR the contents of the first VL elements of VA to the corresponding first VL elements of VB and store the results into VC.
Vector logical xor	vxor	VA, VB, VC	XOR the contents of the first VL elements of VA to the corresponding first VL elements of VB and store the result into VC.
Vector logical nor	vnor	VA, VB, VC	NOR the contents of the first VL elements of VA tot he corresponding first VL elements of VB and store the result into VC.
Scalar-vector shift left [b,h,w]	svshlb svshlh svshlw	VA, SB, VC	Split shift left each element of VA (first VL elements) by the number of bits specified in the low-order $\log_2(n)$ bit of SB and store into the corresponding one in VC. For b, byte, integer length = 8 bits = 1 byte, use 4 bytes from each element in VA with SB. For h, half word, integer length = 16 bits = 2 bytes, use 2 half words from each element in VA with SB. For w, word, integer length = 32 bits = 4 bytes, use 1 word from each element in VA with SB.
Scalar-vector shift right [b,h,w]	svshrb svshrh svshrw	VA, SB, VC	Split shift right each element VA (the first VL elements) by the number of bits specified in the low-order $\log_2(n)$ bit of SB and store into the corresponding one in VC For b, byte, integer length = 8 bits = 1 byte, use 4 bytes from each element in VA with SB. For h, byte, integer length = 16 bits = 2 byte, use 2 bytes from each element in VA with SB. For w, byte, integer length = 32 bits = 4 byte, use 1 bytes from each element in VA with SB.

Table B.2: Vector integer instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector compare [gt, lt, ne, eq] [b,h,w]	vcmpgtb vcmpgth vcmpgtw	V_A, V_B, V_C	Each element in V_A (the first VL elements) is split compared with the corresponding element in V_B and the resulting condition is split stored into the corresponding element in V_C. For b, byte, integer length = 8 bits = 1 byte, use 4 bytes from each element in V_A with corresponding element in V_B. For h, half word, integer length = 16 bits = 2 bytes, use 2 half words from each element in V_A with corresponding element in V_B. For w, byte, integer length = 32 bits = 4 bytes, use 1 word from each element in V_A with corresponding element in V_B.

Table B.2: Vector integer instructions (Continued)

Name	Mnemonic	Syntax	Operation
Vector load	vld	VA, SXM, SSY	Loads a pairwise consecutive words from address VAX+VAY, increment VAX by 4 if VAX>SXM then VAX=0 and VAY=VAY+SSY, and store into the corresponding 2 elements in VA. Repeat till the number of loaded words is >= VL.
Vector load unaligned	vldb	VA, SXM, SSY	Same as vector load but the effective address is not necessarily word aligned. Extra load will be incurred for every row change.
Vector unsigned byte load	vubld	VA, SXM, SSY	Same as vector load but unsigned bytes are loaded instead of integers and extended to integers. Address are byte aligned.
Vector store	vst	VA, SXM, SSY	Store a pair of consecutive elements from VA into a pair wise consecutive words to address VAX+VAY increment VAX by 4 if VAX>SXM then VAX=0 and VAY=VAY+SSY Repeat till number of stored words is >= VL.
Vector store unaligned	vstb	VA, SXM, SSY	Same as vector store but the effective address is not necessarily word aligned. Extra store will be incurred for every row change.
Vector unsigned byte store	vubst	VA, SXM, SSY	Same as vector store but unsigned bytes are stored instead of integers and extended to integers. Addresses are byte aligned.

Table B.3: Vector load and store instructions

Name	Mnemonic	Syntax	Operation
Vector pack signed integer [h,w] unsigned modulo	vpackswuh vpackshub	VA, VB, VC	Vector length VL is divided by two. Concatenate the low-order signed integers of VA and the low-order unsigned integers of VB and place into VC using unsigned modulo arithmetic. VA words is placed in the lower half elements and VB is placed in the upper-half elements of VC. For h, half word, integer length = 16 bits = 2 bytes. For w, word, integer length = 32 bits = 4 bytes.
Vector pack signed integer [h,w] unsigned saturate	vpackswuhs vpackshubs	VA, VB, VC	Vector length VL is divided by two. Concatenate the low-order signed integers of VA and the low-order unsigned integers of VB and place into VC using unsigned saturation arithmetic. VA words is placed in the lower half elements and VB is placed in the upper-half elements of VC. For h, half word, integer length = 16 bits = 2 bytes. For w, word, integer length = 32 bits = 4 bytes.
Vector pack unsigned integer [h,w] unsigned saturate	vpackuwuhs vpackuhubs	VA, VB, VC	Vector length VL is divided by two. Concatenate the low-order unsigned integers of VA and the low-order unsigned integers of VB and place into VC using unsigned saturation arithmetic. VA words is placed in the lower half elements and VB is placed in the upper-half elements of VC. For h, half word, integer length = 16 bits = 2 bytes. For w, word, integer length = 32 bits = 4 bytes.

Table B.4: Vector pack and unpack instructions

Name	Mnemonic	Syntax	Operation
Vector pack signed integer [h,w] unsigned saturate	vpackuwshs vpackuhsbs	VA, VB, VC	Vector length VL is divided by two. Concatenate the low-order signed integers of VA and the low-order unsigned integers of VB and place into VC using signed saturation arithmetic. VA words is placed in the lower half elements and VB is placed in the upper-half elements of VC. For h, half word, integer length = 16 bits = 2 bytes. For w, word, integer length = 32 bits = 4 bytes.
Vector unpack signed integer [b,h]	vunpackubh vunpackuhw	VA, VB	Vector length VL is multiplied by two. Every signed integer in the lower half elements of VA is sign extended to a double size integer and stored in order in VB. For h, half word, integer length = 16 bits = 2 bytes. For w, word, integer length = 32 bits = 4 bytes.
Vector unpack unsigned integer [b,h]	vunpackubh vunpackuhw	VA, VB	Vector length VL is multiplied by two. Every unsigned integer in the lower half elements of VA is extended to a double size integer and stored in order in VB. For h, half word, integer length = 16 bits = 2 bytes. For w, word, integer length = 32 bits = 4 bytes.

Table B.4: Vector pack and unpack instructions (Continued)

Appendix C: Media API

media_ext method	Instruction mnemonic
vec_adds	vaddubs
	vadduhs
	vadduws
vec_add	vaddub
	vadduh
	vadduw
	svadduw
vec_subs	vsububs
	vsubuhs
	vsubuws
vec_sub	vsubub
	vsubuh
	vsubuw
	svsuvuw
vec_sum	vssumub
	vssumuh
	vssumuw
vec_mulodd	vmulob
	vmuloh
vec_muleven	vmuleb
	vmuleh
vec_mov	vmov
vec_mul	vmulw
	svmulw
vec_mac	vmacw

Table C.1: Media API to instruction mnemonic cross-reference

media_ext method	Instruction mnemonic
vec_mac	svmacw
vec_and	vand
vec_or	vor
vec_xor	vxor
vec_nor	vnor
vec_shl	svshlb
	svshlh
	svshlw
vec_shr	svshrb
	svshrh
	svshrw
vec_compgt	vcmpgtb
	vcmpgth
	vcmpgtw
vec_complt	vcmpltb
	vcmplth
	vcmpltw
vec_compne	vcmpneb
	vcmpneh
	vcmpnew
vec_compeq	vcmpeqb
	vcmpeqh
	vcmpeqw
vec_load	vld
vec_load_byte	vldb
vec_unsigned_load	vubld
vec_store	vst
vec_store_byte	vstb

Table C.1: Media API to instruction mnemonic cross-reference (Continued)

media_ext method	Instruction mnemonic
vec_unaligned_store	vubst
vec_pack	vpackshub
	vpackswuh
vec_packs	vpackshubs
	vpackswuhs
	vpackuhubs
	vpackuwuhs
	vpackuwshs
	vpackuhsbs
vec_unpack	vunpacksbh
	vunpackshw
	vunpackubh
	vunpackuhw

Table C.1: Media API to instruction mnemonic cross-reference (Continued)

media_ext methods	Description
vec_init(VectorGeneric v1, byte src[], int offset) vec_init(VectorGeneric v1, int src[], int offset)	Sets the corresponding VX and VY control registers associated with vector v1 to point at the integer entry at offset for the arrage src.
SetStride(int value)	Sets the Stride control register to value.
SetVectorLength(int value)	Sets VL control register to value.
SetTransposeMode()	Enables the vector transpose addressing mode by setting the first bit of CF control register to 1.
SetNormalMode()	Disables the vector transpose addressing mode by setting the LSB of CF control register to 0.

Table C.2: Other media_ext methods

References

[1] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *PLDI '98: Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 280–290, Montreal, 1998. ACM.

[2] Sanjive Agarwala, Charles Fuoco, Tim Anderson, Dave Comisky, and Christopher Mobley. A multi-level memory system architecture for high performance DSP applications. In *2001 International Conference on Computer Design*, September 2001.

[3] Krste Asanovic, James Beck, Bertrand Irissow, Brian E. D. Kingsbury, and John Wawrzynek. T0: A single-chip vector microprocessor with reconfigrable pipelines. In *In Proceedings 22nd European Solid-State Circuits Conference*, September 1996.

[4] A. J. C. Bik and D. B. Gannon. Automatically exploiting implicit parallelism in Java. *Concurrency, Practice and Experience*, 9(6):579–619, 1997.

[5] Aart J.C. Bik, Milind Gikar, and Mohammed R. Haghighat. Incorporating Intel MMX technology into a Java JIT compiler. *Scientific Programming*, pages 167–184, 1999. IOS Press.

[6] Zoran Budimlic, Ken Kennedy, and Jeff Piper. The cost of being object-oriented: A preliminary study. *Scientific Computing*, 7(2):87–95, 1999.

[7] Thomas M. Conte, Pradeep K. Dubey, Matthew D. Jennings, Ruby B. Lee, Alex Peleg, Salliah Rathnam, Mike Schlansker, Peter Song, and Andrew Wolfe. Challenges to combining general-purpose and multimedia processors. *IEEE Computer*, 30(12):33–37, December 1997.

[8] J. Corbal, R. Espasa, and M. Valero. MOM: a matrix SIMD instruction set architecture for multimedia applications. In *SC'99 "Supercomputing Conference"*, Oregon, 1999.

[9] Intel Corporation. Using MMX instructions to implement Viterbi decoding. MMX Technology Application Notes, 2000.

[10] Intel Corporation. Using MMX instructions to perform 3D geometry transformations. MMX Technology Application Notes, 2000.

[11] R. Cucchiara and M. Piccardi. Exploiting image processing locality in cache prefetching. In *Fifth International Conference on High Performance Computing*, Madras, India, December 1998. IEEE.

[12] R. Cucchiara, M. Piccardi, and A. Prati. Exploiting cache in multimedia. In *International Conference on Computing and Systems*, Florance, Italy, 1999. IEEE.

[13] Keith Diefendorff and Pradeep K. Dubey. How multimedia workload will change processor design. *IEEE Computer*, 30(9):43–45, September 1997.

[14] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, pages 85–95, March/April 2000.

[15] K. Ebcioglu, E. Altman, and E. Hokenek. A Java ILP machine based on fast dynamic compilation. In *MASCOTS'97 - International Workshop on Security and Efficiency Aspects of Java*, 1997.

[16] Susan J. Eggers, Joel S. Emer, Henry M. Levi, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, September/October 1997.

[17] Ahmed El-Mahdy and Ian Watson. A two dimensional vector architecture for multimedia. In *To be published in the Proceedings of the European Conference on Parallel Computing, Euro-Par 2001*, Lecture Notes in Computer Science. Springer, 2001.

[18] Roger Espasa and Mateo Valero. Exploiting instruction- and data-level parallelism. *IEEE Micro*, pages 20–27, September/October 1997.

[19] Roger Espasa, Mateo Valero, and James E. Smith. Vector architectures: Past, present and future. In *Proceedings of the 1998 International Conference on Supercomputing*, pages 425–432. ACM, July 1998.

[20] Michael J. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones and Bartlett, Boston, 1995.

[21] Michael J. Flynn, Patrick Hung, and Kevin W. Rudd. Deep-submicron microprocessor design issues. *IEEE Micro*, pages 11–22, July/August 1999.

[22] Jason Fritts, Wayne Wolf, and Bede Liu. Understanding multimedia application characteristics for designing programmable media processors. In *Media Processors 1999*, volume 3655 of *Proceedings of SPIE*, pages 2–13, San Jose, CA, January 1999. SPIE.

[23] John W. C. Fu and James H. Patel. Data prefetching in multiprocessing vector cache memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 54–63. ACM, 1991.

[24] Didier Le Gall. MPEG: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, April 1991.

[25] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 28–39. ACM, May 1995.

[26] Jeremiah Golston. Single-chip H.324 videoconferencing. *IEEE Micro*, 16(4):21–33, August 1996.

[27] David Griswold. The Java HotSpot virtual machine architecture. White paper, Sun Microsystems, March 1998. http://java.sun.com/products/hotspot/whitepaper.html.

[28] MPEG Software Simulation Group. *MPEG-2 Encoder/Decoder, version 1.2 July 19, 1996*. http://www.mpeg.org/MSSG.

[29] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE Micro*, pages 71–84, March/April 2000.

[30] Craig Hansen. MicroUnity's MediaProcessor architecture. *IEEE Micro*, 16(4):34–41, August 1996.

[31] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[32] Cheng-Hsueh A. Hsieh, John C. Gyllenhall, and Wen-mei W. Hwu. Java bytecode to native code translation: The Caffeine prototype and preliminary results. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '96)*, pages 90–99. IEEE, December 2–4 1996.

[33] ISO/IEC JTC1/SC29/WG11 N07092. *ISO/IEC 13818-2 video*, 1994.

[34] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associate cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373. IEEE, 1990.

[35] Paul Kalapathy. Hardware-software interactions on Mpact. *IEEE Micro*, 17(2):20–26, Mar./Apr. 1997.

[36] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media processing with streams. *IEEE Micro*, pages 35–46, March/ April 2001.

[37] P. M. Kogge. *The Architecture of Pipelined Computers*. Hemisphere Publishing Corporation, Washington New York London, 1981.

[38] Christoforos E. Kozyrakis and David A. Patterson. A new direction for computer architecture research. *IEEE Computer*, pages 24–32, November 1998.

[39] Andreas Krall. Efficient JavaVM just-in-time compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, October 1998.

[40] Andreas Krall, Anton Ertl, and Michael Gschwind. JavaVM implementation: Compilers versus hardware. In John Morris, editor, *Computer Architecture 98 (ACAC '98)*, volume 20 of *Australian Computer Science Communications*, pages 101–110, Perth, 1998. Springer.

[41] Atsushi Kunimatsu, Nobuhiro Ide, Yukio Endo, Hiroaki Murakami, Takayuki Kamei, Masahi Hirano, Fujio Ishihara, Haruyuki Tago, Masaaki Oka, Akio Ohba, Teiji Yutaka, Toyoshi Okada, and Masakazu Suzuoki. Vector unit architecture for emotion synthesis. *IEEE Micro*, pages 40–47, March/April 2000.

[42] Ichiro Kuroda and Takao Nishitani. Multimedia processors. *Proceedings of the IEEE*, 86(6):1203–1221, June 1998.

[43] Sun Microsystems Laboratories. *Shade V5.33A*. Mountain View, CA 94043, June 1997.

[44] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '97)*, pages 330–335. IEEE, 1997.

[45] Corinna G. Lee and Derek J. DeVries. Initial results on the performance and cost of vector microprocessors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '97)*, pages 171–182. IEEE, December 1997.

[46] Ruby B. Lee. Subword parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, August 1996.

[47] Heng Liao and Andrew Wolfe. Available parallelism in video applications. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '97)*, pages 321–329. IEEE, December 1–3 1997.

[48] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, second edition, April 1999.

[49] ARM Ltd. Jazelle - ARM architecture extensions for Java applications. White Paper.

[50] Doug Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, September 1997.

[51] Harlan McGhan and Mike O'Connor. PicoJava: A direct execution engine for Java bytecode. *IEEE Computer*, 31(10):22–30, October 1998.

[52] Friederich Mombers and Daniel Mlynek. A multithreaded multimedia processor merging on-chip multiprocessors and distributed vector pipelines. In *Proceedings of the International Symposium on Circuits and Systems ISCAS 1999*, pages 287–290. IEEE, 1999.

[53] Motorola. *AltiVec Technology Programming Environments Manual*, 1998. http://www.motorola.cpm/AltiVec.

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

[55] ISO/IEC JTC1/SC29/WG11 N4030. MPEG-4 overview - (V.18-Singapore) version, March 2001. http://www.cselt.it/mpeg/standards/mpeg-4/mpeg-4.html.

[56] Frank Nack and Adam T. Lindsay. Everything you wanted to know about MPEG-7: Part 1. *IEEE MultiMedia*, 6(3):65–77, July/September 1999.

[57] Frank Nack and Adam T. Lindsay. Everything you wanted to know about MPEG-7: Part 2. *IEEE MultiMedia*, 6(4):64–73, October/December 1999.

[58] Nathaniel John Nystrom. Bytecode-level analysis and optimization of Java classes. Master's thesis, Purdue University, August 1998.

[59] Stuart Oberman, Greg Favor, and Fred Weber. AMD 3DNow! technology: Architecture and implementations. *IEEE Micro*, pages 37–48, March/April 1999.

[60] J. Michael O'Connor and Marc Tremblay. PicoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, March/April 1997.

[61] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218. ACM, June 1997.

[62] Patriot Scientific Corporation. *PSC1000 Microprocessor*.

[63] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM. *IEEE Micro*, pages 34–44, March/April 1997.

[64] Alex Peleg. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):34–41, August 1996.

[65] Rambus Inc. *Direct RDRAM 128/144 Mbit (256Kx16/18x32s) (32 Split Bank Architecture)*, 1999. (Preliminary Data Sheet).

[66] Ian Rogers, Alasdair Rawsthorne, and Jason Souloglou. Exploiting hardware resources: Register assignment across method boundaries. In *Workshop on Hardware Support for Objects and Microarchitectures for Java in conjunction with ICCD'99*, Austin, Texas, 1999.

[67] Semiconductor Industry Association. *Roadmap for Semiconductors 1998 Update*, 1998.

[68] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[69] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. *Proc. of the IEEE*, 83(12):1609–1624, December 1995.

[70] Standard Performance Evaluation Corporation. *SPECjvm98 Benchmarks*. http://www.spec.org/osg/jvm98.

[71] Harold S. Stone. *High-Performance Computer Architecture*. Addison-Wesley Publishing Company, third edition, 1993.

[72] Subramania Sudharsanan. MAJC-5200: A high performance microprocessor for multimedia computing. Sun Microsystems, Inc., Palo Alto, CA 94303, USA.

[73] Sun Microsystems. *picoJava I Data Sheet*, December 1997.

[74] Sun Microsystems. *picoJava II Data Sheet*, April 1998.

[75] Kazumasa Suzuki, Tomohisa Arai, Kouhei Nadehara, and Ichiro Kuroda. V830R/AV: Embedded multimedia superscalar RISC processor. *IEEE Micro*, 18(4):36–47, Mar./Apr. 1998.

[76] Shreekant (Ticky) Thakkar and Tom Huff. Internet streaming SIMD extensions. *IEEE Computer*, pages 26–34, December 1999.

[77] M. Tremblay, J. M. O'Connor, V. Narayanan, and K. He. VIS speeds new media processing. *IEEE Micro*, 16(4):10–20, July/August 1996.

[78] Steven P. VanderWiel and David J. Lilja. When caches aren't enough: Data prefetching techniques. *IEEE Computer*, pages 23–30, July 1997.

[79] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 2000.

[80] N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla. Object-oriented architectural support for a Java processor. In E. Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 330–354. Springer, July 1998.

[81] D. W. Wall. Limits of instruction-level parallelism. Research Report 93/6, DEC Western Research Laboratory, 1993.

[82] Zhongde Wang. Fast algorithms for the discrete W transform and for the discrete fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-32(4):803–816, August 1984.

[83] I Watson, G Wright, and A El-Mahdy. VLSI architecture using lightweight threads (VAULT) - choosing the instruction set architecture. In *Workshop on Hardware Support for Objects and Microarchitectures for Java in conjunction with ICCD'99*, pages 40–44, Austin, Texas, 1999.

[84] David L. Weaver and Tom Germond. *The SPARC Architecture Manual: Version 9*. Prentice-Hall, 1994.

[85] G. M. Wright. *A single-chip multiprocessor architecture with hardware thread support*. PhD thesis, Dept. of Computer Science, University of Manchester, January 2001.

[86] Greg Wright, Ahmed El-Mahdy, and Ian Watson. Dynamic Java threads on the Jamaica single-chip multiprocessor. In *Second Annual Workshop on Hardware Support for Objects and Microarchitectures for Java in conjunction with ICCD'00*, pages 1–5, Austin, Texas, 2000.

[87] B. Yang, S. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. Chung, S. Kom, K. Ebcioglu, and E. Altman. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *1999 International Conference on Parallel Architectures and Compilation Techniques*. ACM, October 1999.