Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2007

MSc THESIS

Design Considerations for High Performance Vector Microarchitecture

Bogdan Spinean

CE-MS-2007-05

Abstract

In this thesis, we analyze the speedup potentials of media and signal processing software on vector processors by evaluating the impact on performance of several design decisions such as number of vector registers, register length, memory latency and bandwidth, number of vector functional units, and number of memory units. We introduce the basic concepts of vector processing and present an overview of key components of the vector architecture and microarchitecture. More specifically, various configurations and organizations of the vector register file and the vector functional units are analyzed in the context of high performance. We also discuss microarchitectural aspects of the vector processor such as the memory system and the control unit. The basic memory access patterns and different ways to support them efficiently are discussed. The traditional approach of tradeoff between memory bandwidth and latency and a more recent one that uses caches are also presented. A theoretical analysis of the vector control unit is given. Mathematical evaluation suggests that the issue logic is not the performance bottleneck. Techniques for improving vector functional units occupancy by exploiting both instruction level and thread level parallelism are presented and discussed. To quantify the influence of the aforementioned design parameters, we modify SimpleScalar 3.0 by adding new vector instructions, a vector register file, and vector functional units. We simulate several media and signal processing applications. Simulation results indicate that using vectorization we can obtain kernel speedups ranging from 5.36x to 14.84x and application speedups of 1.82x and 1.25x for the MPEG2 encoder and decoder respectively. We envision that these speedups can be further increased by coding style improvements. In respect to coding style, a simple example of synthetic code indicates how interchanging of inner loop and outer loops can improve vector performance by a factor of over 50x. The thesis is concluded with presenting optimal microarchitectural features and coding style suggestions for best vector performance.

Faculty of Electrical Engineering, Mathematics and Computer Science

Design Considerations for High Performance Vector Microarchitecture

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Bogdan Spinean
born in Brasov, Romania

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Design Considerations for High Performance Vector Microarchitecture

by Bogdan Spinean

Abstract

In this thesis, we analyze the speedup potentials of media and signal processing software on vector processors by evaluating the impact on performance of several design decisions such as number of vector registers, register length, memory latency and bandwidth, number of vector functional units, and number of memory units. We introduce the basic concepts of vector processing and present an overview of key components of the vector architecture and microarchitecture. More specifically, various configurations and organizations of the vector register file and the vector functional units are analyzed in the context of high performance. We also discuss microarchitectural aspects of the vector processor such as the memory system and the control unit. The basic memory access patterns and different ways to support them efficiently are discussed. The traditional approach of tradeoff between memory bandwidth and latency and a more recent one that uses caches are also presented. A theoretical analysis of the vector control unit is given. Mathematical evaluation suggests that the issue logic is not the performance bottleneck. Techniques for improving vector functional units occupancy by exploiting both instruction level and thread level parallelism are presented and discussed. To quantify the influence of the aforementioned design parameters, we modify SimpleScalar 3.0 by adding new vector instructions, a vector register file, and vector functional units. We simulate several media and signal processing applications. Simulation results indicate that using vectorization we can obtain kernel speedups ranging from 5.36x to 14.84x and application speedups of 1.82x and 1.25x for the MPEG2 encoder and decoder respectively. We envision that these speedups can be further increased by coding style improvements. In respect to coding style, a simple example of synthetic code indicates how interchanging of inner loop and outer loops can improve vector performance by a factor of over 50x. The thesis is concluded with presenting optimal microarchitectural features and coding style suggestions for best vector performance.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2007-05

Committee Members :

Advisor:	Georgi Gaydadjiev, CE, TU Delft
Advisor:	Georgi Kuzmanov, CE, TU Delft
Chairperson:	Stamatis Vassiliadis, CE, TU Delft
Member:	Stephan Wong, CE, TU Delft
Member:	Rene van Leuken, CAS, TU Delft

To my loving parents

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank professor Gheorghe Toacse for guiding my first steps in computer architecture and for sending me to Delft as an exchange student.

I would also like to thank professor Stamatis Vasiliadis for offering the possibility to further study in Delft. This thesis would not have been possible without my advisors Georgi Gaydadjiev and Georgi Kuzmanov. Thank you for your guidance!

Bogdan Spinean
Delft, The Netherlands
April 24, 2007

1 Introduction

1.1 Background

1.1.1 Vector Processors - Principles of Operation

A vector processor is a CPU design that is able to run mathematical operations on multiple data elements simultaneously. For a scalar processor, each data operation is described using a single instruction. In, contrast, vector processor's instructions operate on entire arrays of elements. For instance, the addition of two vectors element by element can be done by a memory to memory vector processor using a single instruction.

Vector processors are common in the scientific computing domain, where they formed the basis of most supercomputers through the 1980s and into the 1990s. Programs executed on the vector processors are normal programs, written in a high level programming language, usually Fortran. Sections that contain high data level parallelism are run in vector mode, while the rest of the program is run in normal, scalar mode. Consequently, a vector processor contains a scalar unit and a vector unit that work side by side. A vector processor can run programs entirely in scalar mode, but programs cannot be executed entirely in vector mode because control dominated sections (nester if statements, switches, etc) of the programs are not vectorizable.

Two factors affect the success with which a program can be run in vector mode: the structure of the program itself and the capability of the compiler. The first factor is influenced by the algorithms chosen and, to some extent, the way they are coded.

Not all loops can be vectorized. If the iterations within a loop are independent or can be restructured to become independent, then the loop can be vectorized. The following example shows such a loop:

```
for i = 1 to n do
    A[i] = 2 * A[i]
```

In contrast, when the data used in an iteration depends on another iteration, the loop cannot be vectorized. For instance, a loop that computes the Fibonacci sequence cannot be run in vector mode:

```
for i = 1 to n do
    F[i] = F[i-1] + F[i-2]
```

Iteration i depends on the results produced by iterations $i-1$ and $i-2$ thus, these iterations cannot be run in parallel.

The second factor that influences the success with which a program can be run in vector mode is the capability of the compiler. While no compiler can vectorize a loop

Benchmark name	Operations executed in vector mode, compiler optimized	Operations executed in vector mode, hand optimized	Speedup from hand optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	NA
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	NA
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

Table 1.1: Level of vectorization among the Perfect Club benchmarks when executing on the Cray Y-MP

like the one presented above, there is a great variation in the ability of compilers to determine whether a loop can be vectorized. The techniques used to vectorize programs are the same used in superscalar processors for uncovering Instruction Level Parallelism and for determining which instructions ca execute in parallel.

In [15] are shown the various vectorization levels observed for the programs that form the Perfect Club benchmark suite. Table 1.1 shows the percentage of operations executed in vector mode for two versions of code executed on the Cray Y-MP. The first version is obtained with just compiler optimizations on the original code, that was initially written to run on a scalar processor. The second version has been extensively hand optimized for the Cray Y-MP vector processor.

The level of vectorization is not sufficient by itself to determine performance. Alternative vectorization techniques might execute fewer instructions, or keep more values in vector registers, or allow better overlapping of vector operations, and therefore improve performance even if the vectorization level remains the same or drops. So, the hand-optimized versions generally show significant speedups either by increasing vectorization levels or by more efficiently using the machine characteristics.

The vector processors are highly dependent on the ability of the vectorizing compiler (or of the programmer writing assembly code) to extract the data level parallelism present in the executed applications.

1.1.2 Short History of Vector Processors

The pioneering vector processors where the CDC STAR-100 and the TI ASC. They where both announced in 1972 and both had a memory-to memory architecture. However, the first successful vector processor was the Cray-1 introduced in 1976. In contrast to the designs mentioned above, the Cray-1 used a register to register architecture that offered substantial benefits.

The vector processors of the 1980s and mid 1990s where by far the fastest supercomputers available. They targeted utmost performance thus, their prices where enormous. Initially supercomputers where implemented in bipolar ECL technology. This was very fast but power hungry and offered a low integration level. Thus, a single vector processor was build from many ECL chips.

The memory system is one of the major expenses in vector processors. To provide the huge bandwidth required, traditionally SRAM was used as main memory to reduce the number of memory banks required and to reduce the latency. SRAM has an access time much lower than that of DRAM, however, it costs roughly ten times as much per bit.

In an attempt to stay ahead of the ever-improving CMOS microprocessor, supercomputer vendors like Cray Computer Corporation, Convex, or Alliant tried the introduction of gallium arsenide but the costs to introduce a new technology have proven to be too great. Cray eventually built one GaAs-based machine in the early 1990s, the Cray-3, but the effort was so costly that the venture failed.

In the mean time, the small RISC processors underwent significant performance improvements. They where implemented in CMOS technology, that offered a very large scale integration; also, they where produced in very large series offering low prices. In time, the performance gap between the superscalar processors and the vector supercomputers slowly shrinked and the vector supercomputers where overrun by the much cheaper highly parallel machines build from commodity superscalar CMOS processors.

In 1993 Cray Research introduced their first highly parallel machine, the T3D, employing up to 2048 Digital Alpha 21064 microprocessors. Following the same trend, in 1994, Convex, that previously produced vector minicomputers, introduced an entirely new design, known as the Exemplar which was a parallel-computing machine based on off-the-shelf HP-PA RISC chips.

Since there where just a handful of companies producing vector processors, most of which switched to massively parallel machines, this meant the end of the vector processors dominance in the supercomputing area (an extensive history of the vector processors can be found in section G.9 of [15]).

However, the concepts behind the vector processor are still widely used. The vector paradigm relies on extensive data parallelism. Not only scientific calculations but also multimedia applications have an abundance of data level parallelism.

Superscalar processors have vector-like instruction set extensions that use the concepts developed in vector processors. MMX, 3DNow!, SSE and many other ISA extensions use a single instruction to operate on multiple data elements. For an extensive presentation of these extension refer to [5].

A clear sign of the rebirth of vector processors is the Cell processor. It consists of a dual core, modest performance PowerPC processor, that is used to distribute the work to eight mini vector units. It was launched in late 2006 and is expected to perform any type of precision scientific calculations 3 to 12 times faster than any desktop processor at a similar clock speed.

Also, IBM announced plans to produce the world fastest supercomputer using vector units. The current world fastest supercomputer is the Blue Gene/L with a sustained performance of 280 teraflops (280 x 1024 billion floating point operations per second). The new IBM machine, named Roadrunner is expected to reach a full petaflop performance by combining AMD Opteron blade servers and Cell-based accelerator systems.

All these show that the concepts behind the vector processor are more actual than ever. Future microprocessor design will include more and more of the vector techniques with the aim of increasing performance by efficiently utilizing the huge amount of data level parallelism present in today's applications.

1.2 Introduction to Vector Architecture and Microarchitecture

In the following pages we shall present a general picture of a vector processor. This is meant to lay the grounds for the more advanced architectural topics introduced in later chapters. A firm understanding of the basics is the key.

Most of the techniques used in the early vector processors can also be found in scalar processors. These are general techniques adapted to the processing model: working on entire vectors or on single elements.

The first part contains a general presentation of the vector architecture, then the vector instruction set is described. The section concludes with some architectural and micro-architectural particularities of the vector machines.

1.2.1 General Overview of Vector Processors

A vector processor consists of an ordinary scalar unit plus a vector unit. The fetch unit distributes the program instructions to the corresponding scalar or vector unit. These two units work in parallel. A vector unit on its own is not efficient as only parts of applications can be vectorized. The vector unit must work side by side with a powerful scalar unit. The segments of the program rich in branches and conditional statements are best suited for the scalar processor. Also address computations and exception handling are done in the scalar unit. On the other hand uniform, computation intensive, data parallel segments of code are executed by the vector unit. Not all array operations can be executed by the vector unit. There have to be no data dependencies between the elements of the array. If an application contains large sections of vectorizable code the use of a vector unit can greatly speedup the program execution time.

There are several implementations that treat the vector unit as an extension to an already existent scalar processor. This is the case with the IBM System/370 that has the 3090 vector facility. Also the MIPS has a vector extension named VMIPS that is treated like a coprocessor.

There are two primary types of architectures for vector units: vector-register processors and memory-memory vector processors. In a vector-register processor, all vector computations are performed on operands stored in the vector registers. This is a similar approach to the load-store scalar processors.

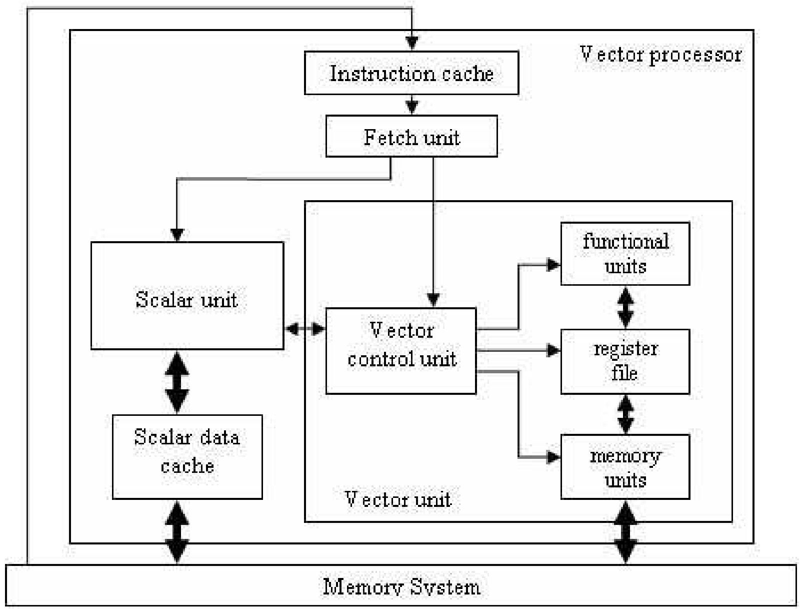


Figure 1.1: General organization of a vector processor

In the memory-memory architecture, as the name suggests, there are no registers, all vector operations take their source operands directly from memory and write results back to memory. The early machines, like the CDC Star-100 and its successors, used this principle. Like in the scalar case this approach has severe disadvantages like incurring the memory latency for every reuse of data and increasing the stress on the memory system. From this point on, we will focus on vector-register architectures only.

The primary components of the vector unit are the following, see also Figure 1.1 illustrating the generic vector architecture:

- **Vector registers**: each vector register contains multiple elements, implementation dependent, power of two, typically 64 or 128. Each register must have several read and write ports in order to allow a high degree of overlap among vector operations using different registers. The addition of read/write ports increases complexity of the register file while the shortage of these ports can cause structural hazards. The regular access pattern of the registers allows several strategies to be implemented that reduce the read/write port requirements.

- **Scalar registers**: these registers are part of the scalar unit, some vector instructions have a scalar operand that comes from a scalar register.

- Vector functional units: each unit is fully pipelined and can start a new operation on every clock cycle. Since an instruction generates tens of operations back to back, throughput is more important than latency. The parallel semantics of a vector instruction allows an implementation using either a deeply pipelined functional unit or by using an array of parallel functional units, or a combination of parallel and pipelined functional units. Each parallel functional unit is called a lane and using lanes changes the way vector registers are designed, again, to be discussed later.

- **Vector Load-Store units**: loads or stores vectors to or from memory. The memory units are fully pipelined, so that words can be moved between the vector registers and memory with a bandwidth of 1 word per clock cycle, after an initial latency. Because vector applications work with large data sets that do not fit into a cache, the vector load-store unit is connected directly to the main memory. The memory latency is mitigated by the large number of words transferred. The memory system is the most critical part of a vector machine design. The memory bandwidth is the primary performance bottleneck for many vectorizable applications.

One of the most clear features of traditional vector processors is an in order execution model of the vector instructions. Until the mid '90s all major vector platforms executed vector instructions in strict program order. Although there was overlapping between scalar and vector instructions a stall in a vector instruction prevented further vector instructions dispatching until the hazard was resolved. The Cray machines: Cray-1, Y-MP, C90 and also the Convex C series execute vector instructions in strict program order. Some of the Japanese models have a VLIW scalar unit that allows several operations to be packed into a single instruction but the vector unit was still in order. Only the NEC SX-4 launched in '96 uses out of order vector and scalar pipelines.

The workload for a scalar unit within a vector machine is somewhat different from that of a vectorless machine. In [16] is shown that many applications with high Instruction Level Parallelism are vectorizable, and that many non-vectorizable applications have low Instruction Level Parallelism. This suggests that the non-vectorizable portion of the workload is best handled by a scalar processor that emphasizes low latency rather than wide parallel instruction issue. Also [24], based on a study of Cray Y-MP performance argues that a less highly pipelined and lower latency scalar unit is best suited for the non-vectorizable portion of the workload. Free of the burden of providing high throughput and wide instruction issue, such a scalar processor should also be simpler to design. Because cycle time is reduced such a scalar processor has a performance advantage over a wide superscalar processor for tasks that offer low Instruction Level Parallelism.

Scalar processors typically employ a hierarchy of data caches to lower average memory latency. Vector memory accesses will usually bypass the upper levels of the cache hierarchy to achieve higher performance on the larger working sets typical for vector code. Extra logic is required to maintain coherence between vector and scalar memory accesses, but this will also usually be required for multiprocessor support or for coherent I/O devices.

1.2.2 Vector Instruction Set

This section presents the instruction set of the Vector MIPS (VMIPS) processor. It contains simple, intuitive instructions that show the semantic power of the vector instructions. The mnemonics of the instructions are similar to those of the MIPS scalar instructions, they have a V appended. The internal organization of the VMIPS machine is not relevant for the instruction set and will not be described. For a complete description please refer to [15].

The vector instructions take as their input either a pair of vector registers (addV) or a vector register and a scalar register (addVS). For the latter instruction, the value in the scalar register is used as the input for all operations. The scalar value is copied to the vector functional unit at issue time. Most vector operations have a vector destination register, although few, like population count, produce a scalar value which is stored in a scalar register. Load instructions have as destination a vector register while Store instructions have the vector register as source. Operands can be two scalar registers for starting address and stride or a vector register for indexed operations. More on this in the next pages.

There are also comparison instructions that compare the elements of a vector register to a scalar value or to another register. These comparison instructions write a mask register, used for conditional execution. In addition to the vector registers two additional special purpose registers are needed: the vector length (VL) and the vector mask register (VMR). The vector length register determines the number of operations the vector instructions perform. It contains an integer value between 0 and the vector register length VLMAX or the section size. The VL register is used for instance during the last iteration of a loop when the vector registers are not completely full and so, operations should not be performed on empty register elements. For example, if we want to add element by element two arrays each containing 1020 elements and the vector register length is 100 (VLMAX) then the loop will have 11 iterations, the first ten VL is set to 100, the last one will process just 20 elements, thus in the last iteration VL is set to:

$$1020 \ mod \ VLMAX = 20 \tag{1.1}$$

The mask register is used for conditional operations. The exact working of the mask register will be described later in the chapter.

1.2.3 Specific Vector Techniques

In this section some specific techniques used in vector processors will be presented. These where introduced in the Cray-1 and have become a standard in all vector implementations threreafter. This section presents chaining, support for conditional operations using the mask register and indexed memory accesses.

In order to make things clearer, in this section several examples will be presented. These examples are based on basic vector unit, invented just for these examples. It has an organization very similar to the Cray-1 and has the following characteristics:

- A single memory port for load/stores
- Two pipelined functional units one for additions and another one for multiplication
- 8 vector registers each 64 elements long
- instructions are executed in program order

Instructions	Operands	Function
addV	rV1,rV2,rV3	Add elements of rV2 and rV3, then put each result in rV1
addVS	rV1,rV2,rS1	Add rS1 to each element of rV2, then put each result in rV1.
subV	rV1,rV2,rV3	Subtract elements of rV3 from rV2, then put each result in rV1
subVS	rV1,rV2,rS1	Subtract rS1 from elements of rV2, then put each result in rV1.
subSV	rV1,rS1,rV2	Subtract elements of rV2 from rS1, then put each result in rV1
mulV	rV1,rV2,rV3	Multiply elements of V2 and V3, then put each result in rV1.
mulVS	rV1,rV2,rS1	Multiply each element of rV2 by rS1, then put each result in rV1.
divV	rV1,rV2,rV3	Divide elements of rV2 by rV3, then put each result in rV1.
divVS	rV1,rV2,rS1	Divide elements of rV2 by rS1, then put each result in rV1
divSV	rV1,rS1,rV2	Divide rS1 by elements of rV2, then put each result in rV1.
loadV	rV1,rS1	Load vector register rV1 from memory starting at address rS1
storeV.	rS1,rV1	Store vector register rV1 into memory starting at address rS1
LVWS	rV1,(rS1,rS2)	Load rV1 from address at rS1 with stride in rS2, i.e., rS1+i x rS2.
SVWS	(rS1,rS2),rV1	Store rV1 from address at rS1 with stride in rS2, i.e., rS1+I x rS2.
LVI	rV1,(rS1+rV2)	Load rV1 with vector whose elements are at rS1+rV2(i), i.e., rV2 is an index.
SVI	(rS1+rV2),rV1	Store rV1 to vector whose elements are at rS1+rV2(i), i.e., rV2 is an index.
CVI	rV1,rS1	Create an index vector by storing the values 0, 1 x rS1, 2 x rS1,...,63 x rS1 into rV1.
S–V	rV1,rV2	Compare the elements (EQ, NE, GT, LT, GE, LE) in rV1 and rV2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0.
S–VS	rV1,rS1	Put resulting bit vector in vectormask register (VM). The instruction S–VS performs the same compare but using a scalar value as one operand.
POP	rS1,VM	(Population count) Count the 1s in the vector-mask register and store count in rR1.
CVM		Set the vector-mask register to all 1s.
MTC1	VLR,rS1	Move contents of rS1 to the vector-length register.
MFC1	rS1,VLR	Move the contents of the vector-length register to rS1.
MVTM	VM,rS1	Move contents of rS1 to the vector-mask register.
MVFM	rS1,VM	Move contents of vector-mask register to rS1.

Table 1.2: The VMIPS Instruction Set

1.2.3.1 Chaining

Chaining is a micro-architectural feature, the equivalent of forwarding used in scalar processors. Let us consider an example, a typical vector operation.

$$Y = a \cdot X + Y \tag{1.2}$$

This would be translated into assembly code like this:

```
1   loadS   rS1, a
2   loadV   rV2, Y
3   loadV   rV1, X
4   mulVS   rV3, rV1, rS1
5   addV    rV1, rV2, rV3
6   storeV  rV1, Y
```

The basic, naive, approach is to execute the instructions one after the other. An instruction starts execution only after the previous one ends. This would mean that the processor's resources (functional units and memory ports) are heavily under-utilized. While a load or store is in progress, no logic or arithmetic operation is performed and vice-versa. In other words, instructions that have data dependencies cannot be overlapped. Since a vector instruction spans over many clock cycles the penalty of a stall in the instruction pipeline is significantly larger than in the scalar case.

For the scalar processors forwarding can improve the resource utilization. This concept has actually been introduced by the Cray-1 vector processor and then adapted to the scalar paradigm. Instead of a single element being forwarded, a contiguous stream of elements is chained from the output of a resource to the input of another one.

The vector instructions proceed element by element. The first result of the multiply instruction is available much sooner than the entire instruction finishes. Thus it can be forwarded to the add instruction in a similar way to the scalar forwarding. Even though this time an array of elements is chained instead of a single one, the complexity is similar to the scalar forwarding. A contiguous stream of elements must be routed from the output of the first functional unit to the input of the second one during the execution of the two instructions.

Let us take a look at how our little example will work using chaining. The scalar load is followed by the vector load of Y. The following instruction, the load of X, cannot start before the load X finishes because of structural hazard, there is only one memory port. The fourth instruction, mulVS, has a data dependency with the previous one, but it can be resolved through chaining. As soon as the first element of array X comes from memory it is sent also to the multiplier and the loadV and the mulVS are overlapped. The addV instruction can also be chained. The output of the multiplier goes to the input of the adder. The storeV instruction, again, can be chained from the addV. However, a structural hazard may appear with the last loadV. If the vectors are long enough by the time the first element of addV is ready to be chained to the memory port, the loadV instruction might still be executing, still chained to the mulV. Thus the storeV might have to stall until the memory port is free.

In our basic vector organization up to three instructions can be overlapped, one for each of the resources: a memory port, an adder and a multiplier. In this example, using chaining, the machine had, for a number of cycles, all of its functional units busy, going at its highest throughput. Without chaining, our general vector architecture, could execute concurrently only an addV and a mulV with no data dependencies. In the following chapters we present techniques to further increase the fraction of time when the processor uses its full potential. Chaining significantly increases the performance but it also increases implementation complexity. In the following pages we shall analyze the costs of possible implementations.

Flexible chaining allows a vector instruction to chain to any other vector instruction if no structural hazard is generated. In a processor without chaining only the following data transfers are supported:

- memory access instructions transfer data between registers and memory
- arithmetic and logic instructions read their source operands only from the registers, the results are written back to the register file

Chaining requires that routing of data between the registers, functional units and memory ports be much more flexible. Additional connections are necessary:

- from output of functional unit to input of any other functional unit (two consecutive arithmetic or logic instructions)
- from memory port to input of any functional unit (load followed by an arithmetic or logic instruction)
- from output of any of the functional units to the memory port (an arithmetic or logic instruction followed by a store)

In the general case of a processor having many functional units and memory ports, the necessary wiring can be quite substantial increasing both the area and delay. Flexible chaining requires simultaneous access to the same vector register by different vector instructions. This can be implemented either by adding more read and write ports or by organizing the vector-register file storage into interleaved banks in a similar way to the memory system. More details will be presented in the chapter presenting the register file.

1.2.3.2 Mask Operations

A program cannot benefit from the vector unit unless it has large segments of vectorizable code. One of the main reasons why higher level of vectorization is not achieved is the presence of if statements inside loops.

Let's take an example

```
for i = 1 to n do
   if (A[i] == 0)
    A[i] = A[i] + B[i]
```

This loop cannot normally be vectorized because of the conditional execution of the body; however, if the inner loop could be run for the iterations for which A[i] == 0, then the addition could be vectorized. In superscalar processors predicated instructions can turn such control dependencies into data dependencies, enhancing the ability to parallelize the loop. This concept was actually introduced by vector processors in the form of vector-mask control. The vector mask is a Boolean vector that controls the execution of a vector instruction just as predicated (conditionally executed) instructions use a Boolean condition to determine whether an instruction writes back its result. When the vector mask mode is enabled, vector instructions operate only on the vector elements whose corresponding entries in the mask register are 1. If the vector mask register is set by the result of a condition, only elements satisfying the condition will be affected.

Using the mask register the code of our example becomes:

```
1   loadV   rV1, A
2   loadV   rV2, B
3   load    rS1, zero
4   seqVS   rV1, rS1
5   addV    rV1, rV1, rV2
5   CVM
7   storeV  rV1, A
```

The fourth instruction (Set on EQual) checks if the elements of A (loaded in rV1) are equal to zero (rS1). If they are zero, the corresponding bit in the vector mask is set to 1, else is reset to 0. The add instruction writes back results only for elements that have a 1 in the corresponding position of the vector mask. An instruction is also provided to set the vector mask to all 1s.

Vector instructions executed with a vector mask take the same amount of cycles as if there where no vector mask at all. The vector mask only influences the write-back of the results thus even if the mask bit of an operation is 0, that particular operation still is executed. Even with a large number of 0s in the mask, using the vector mask may still be significantly faster than using scalar mode.

Vector mask execution can complicate things. The operations that shouldn't be executed might generate exceptions. For instance if the operation in the loop where B = B/A then for the operations where A == 0 we would get a floating point exception since that is a division by 0. Care must be taken when handling exceptions. Precise exceptions are in general complicated to implement, as we shall see in the chapter presenting the control unit of the vector processor.

1.2.3.3 Indexed Accesses

Another important source of limited vectorization is the use of sparse matrices. Sparse matrices are big matrices that have the vast majority of items 0. Thus storing just the nonzero elements occupies significantly less space than storing the entire matrix. This means that the elements are stored in a compacted form and accessed indirectly.

Let us consider a new example:

```
for i = 1 to n do
```

```
A[K[i]] = A[K[i]] x B[i]
```

In this example the K array is used to point to the nonzero elements of A. K is called an index vector.

A primary mechanism for supporting sparse matrices is scatter/gather instructions. A gather operation takes an index vector and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector. In other words, elements that have random positions in memory are brought in the vector register in consecutive positions. After operating on these elements brought into a dense form, they can be stored back to memory using the scatter instruction. All vector machines since the Cray-1 have support for these two instructions. The code in our example becomes:

```
1   loadV      rV1, K
2   loadVI     rV2, (A + rV1)
3   loadV      rV3, B
4   addV       rV1, rV1, rV2
5   storeVI    (A + rV1), A
```

The first instruction loads the index vector. Then, A is gathered using K as the index vector Computation is performed A is scattered using the same K array as the index vector.

A simple vectorizing compiler can not automatically vectorize the source code because the compiler cannot know at compile time that the elements of K are distinct and so, no dependencies exist. Instead, a compiler directive can aid the compiler telling it to run the loop in vector mode.

A more sophisticated compiler can vectorize the loop automatically by adding guard statements to check at runtime the data dependencies. If a dependence is detected, the vector length is set to a smaller value that avoids all dependency violations, leaving the remaining elements to be handled on the next iteration of the loop. This scheme adds considerable software overhead to the loop but this overhead is mostly vectorized for the common case where there are no dependencies and so, the loop runs considerably faster than in scalar mode, but slower than the case when a compiler directive is provided.

1.2.4 Summary

In this section we have seen that the vector processor consists of a scalar unit and vector unit that work side by side. We have seen that control code and bookkeeping is executed in the scalar unit, while the data parallel computations are executed by the vector unit.

We have also seen a general overview of vector instructions. Most of them are arithmetic operations. There are three memory access types: unit stride, non-unit stride and indexed. The vector control instructions are instructions that operate on the vector mask register and on the vector length register.

The final pages have presented specific techniques used in vector processors. Chaining is the equivalent of forwarding in scalar processors. Conditional execution in vector processors is provided by the mask register and provides the same functionality as predicated instructions do in scalar processors. The indexed memory accesses do not have correspondent in scalar processors, these instructions group many accesses to random locations of the address space into a single instruction. We shall see in the chapter dedicated to the memory system that this type of memory access accounts for most of the cost of the vector main memory.

The following chapter analyzes architectural elements of the vector unit while the third chapter looks at microarchitectural components. After understanding the inner-workings of the vector processor we make a step forward an build a simulation model and analyze the impact of certain design decisions of both hardware and software. The fourth chapter presents the simulation environment and results while the last chapter contains our conclusions.

1.3 Thesis Objectives

The high amount of data level parallelism in current applications indicates vector architectures as a strong candidate for increasing performance beyond the current limits of instruction level and thread level parallelism.

The vector architecture is focused on operations on arrays stored at continuous addresses in memory. All computation is very uniform and simple. This is what gives the vector processor's strength: the hardware is simple and fast. To gain full performance the software should be tailored with the same principles in mind: regularity and uniformity.

Most of the current applications run on scalar processors and are coded with the scalar paradigm in mind. Scalar processors can offer high performance even without high regularity of the data structures and computations.

The goal of this thesis is to analyze the potentials of media and signal processing applications for acceleration by vector processors. We want to analyze the impact of both microarchitectural decisions and coding style on performance. We are looking at both software and hardware.

Our objectives are:

- Analyze media and signal processing applications for vectorization potentials
- Vectorize their code and simulate
- Analyze the influence of different design parameters such as register file size, section size, memory latency, memory bandwidth and number of functional and memory units over performance.

We analyze image, sound and video processing software for their very high amount of data level parallelism and adapt these applications to our simulator by vectorizing the selected algorithms. We then run multiple tests by varying the aforementioned parameters of our vector simulator and measure results.

We are giving answers to the following design questions:

- What are the architectural and micro-architectural characteristics of the hardware in order to efficiently run media and signal processing applications?

- What algorithms and coding styles prove to be suitable for vectorization?
- Is data level parallelism enough for good vector performance?
- What are the exact attributes that impact vector performance?

1.4 Thesis Organization

Chapter 1 has presented background information on vector processors.

One of our goals is to find architectural and microarchitectural design considerations for high performance vector computing. To this extent, we provide an extensive theoretical analysis of the elements that are crucial for performance. Chapter 2 describes the major components of a vector processor that are influenced by both the architecture and microarchitecture: the vector register file and the vector functional units. Chapter 3 is dedicated to the components of a vector processor that are influenced only by microarchitectural choices.

Chapter 4 describes the experiments we have performed by presenting the simulation framework, the applications we have studied and the simulation results. Chapter 5 summarizes our work, presents our conclusions and design recommendations for high performance vector processing.

Architectural Components Investigated 2

This chapter is dedicated to the components of the vector processor that are influenced by the architecture, these are the vector register file and the vector functional units.

The number of registers and their size is fixed by the architecture while the organization in banks (if any), number of read/write ports, timing and so on are totally transparent to the architecture and thus, implementation dependent. The same thing holds for the vector functional units. While the architecture does not explicitly state the type of functional units required, the instruction set implicitly determines the needed functionality. However, the organization of the functional units, the number of functional units and many other parameters are implementation dependent and not influenced in any way by the architecture.

2.1 Vector Register File

The vector register file lies at the heart of the vector unit. It provides both temporary storage for intermediate values and the interconnections between vector functional units.

There are two main groups of aspects that will be analyzed in this chapter about the vector register files. The first one is the architectural aspects, meaning what the programmer and the compiler sees. In certain publications (like [2]) this is called the configuration of the register file. The first half of this section includes analysis of the number of registers, their length, a fixed number and size of registers or configurable registers. Also, we shall present code optimizations that compilers can or cannot use, depending on the number of registers. There will be a discussion on the influence on spill code and ways to reduce it. Operating system issues related to context switching are also presented.

The second part of this section is dedicated to the organization of the vector register file. First, we will present the detailed requirements that a register file must fulfill. Some micro architectural techniques may add requirements to the functions the vector register files must provide. For instance, chaining requires more read/write ports while functional units organized in multiple lanes require the partitioning of the vector registers in banks.

Starting from these requirements the complexity of the implementation will be determined. The final sections introduce and discuss several microarchitectural techniques that have been used in commercial systems to reduce the cost of implementing a large high-bandwidth vector register file.

2.1.1 Vector Register Architecture

The number and length of programmer-visible vector registers to provide is a key decision in the design in a vector unit. Table 2.1 shows the configuration of vector register files for

System	Number of vector registers	Vector register length	Element size (bits)	Total storage (KB)
Vector Supercomputers				
Cray 1.2, 3.4	8	64	64	4
Cray X-MP, Y-MP	8	64	64	4
Cray C90, T90	8	128	64	8
Fujitsu VP-200	8-256	1024-32	64	64
Fujitsu VPP500. VPP300	8-256	2048-64	64	128
Hitachi S8 10	32	256	64	64
NEC SX-2	8+32	256	64	80
Unisys Sperry ISP	16	32	72	4.6
IBM 3090 Vector Facility	16	128	32	8
Vector Minicomputers				
Cray J90	8	64	64	4
Convex Cl, C2	8	128	64	8
Convex C4/XA	16	128	64	16
DEC Vector VAX	16	64	64	8
Ardent Titan	8 x 32	32	64	64
Single-chip Vector Coprocessors				
NEC VPP	4+4	64+96	64	5.1
Fujitsu uVP	8-64	128-16	64	8
CM-5 Vector Units	4-16	16-4	64	0.5

Table 2.1: Vector register file configurations for different vector processors

the most successful vector processors. Some machines allow a fixed number of elements to be dynamically repartitioned to form either a large number of short vector registers or fewer, longer vector registers. This dynamic configuration of the vector register file is common to the Japanese vector processors like the NEC SX series or the Fujitsu VP series.

2.1.1.1 Vector register length

Vector processors, in contrast with the scalar processors, operate on array of elements. This is the fundamental concept that gave birth to the vector processors. But how long should the vector register be? This is one of the most important design decisions for vector units. Intuitively, the longer the better. The memory latency is better amortized over longer arrays, because the operation startups (functional unit pipeline latency) are the same for all lengths. Of course there are practical limits, and several constraints to be discussed in the following pages.

We can place a lower bound on the required vector register length if we want that in the average application case all the vector functional units are kept busy. This requires that an instruction can be issued to every functional unit before the first functional unit finishes execution. In other words, the control part of the vector unit must issue instructions at leas as fast as the functional units can execute them. In analogy to the producer-consumer concept, the control part must produce instructions faster than the functional units can consume them.

Let us identify the elements that contribute in the decision. On the consumer part, each instruction keeps a functional unit busy. Since a vector instruction operates on the elements of the vector registers, the execution time of the instruction is dependent on the vector register length. Also, the execution time is dependent on the number of lanes used in the implementation of the functional units. On the producer part, it is important the number of instructions issued per cycle.

We can sum-up these constraints and emphasize on the vector register file length:

$$Vector\ register\ length \geq \frac{Number\ of\ lanes\ \cdot\ Number\ of\ functional\ units}{Number\ of\ vector\ instructions\ issued\ per\ cycle} \tag{2.1}$$

Note that executing a vector instruction takes much longer than the time spent fetching, decoding and issuing the vector instruction. Thus, the number of vector instructions issued per cycle is not the bottleneck. As we will see, one instruction issued per clock cycle is enough for most vector machines.

Also note that on the right hand side of the equation all the factors are architecture independent. The number of functional units, the number of lanes and the width of the issue logic are all implementation dependent. In contrast, the vector register length is most of the times fixed by the Instruction Set Architecture. The IBM System/370 vector architecture changed this and made the vector length an implementation detail, not visible to the ISA. The next subsection presents the way the vector length can be hidden from the ISA. For now, it is interesting to note the tradeoff between architectural and organizational elements.

There are several advantages gained by increasing the vector register length beyond the threshold given by the previous equation:

- Any vector startup overhead is amortized over a greater number of productive cycles. Similarly, energy efficiency can improve because instruction fetch, decode and dispatch energy dissipation are amortized over more datapath operations

- Instruction bandwidth requirements are reduced. Also, because each instruction creates a lot of work for the vector unit, the scalar unit can get further ahead of the vector unit for a given length of vector instruction queue

- Longer vector registers will tend to increase the spacial locality within vector memory reference streams, by taking more elements from one array before requesting elements from a different array

- If the maximum possible iteration count of a vectorizable loop is known to be less than or equal to the maximum vector register length at compile time, then stripmining code can be completely avoided; lengthening the vector registers will increase the probability of this occurring

Of course, there are disadvantages to increasing the vector length over the threshold in the equation:

- For all sources of fixed overhead like functional unit startup cycles, instruction energy consumption and instruction bandwidth, each doubling of vector register length will give ever smaller reductions in the amortized overhead

- Lengthening the register length only improves performance for applications with natural vector length greater than existing vector register length and these become fewer as vector register length grows

- Longer register translate into more memory traffic in certain cases like spill code or at context switches

2.1.1.2 Implementation-dependent Vector Register Length

One of the goals of the Instruction Set Architecture is to make the architecture scalable, so as to take advantage of technology improvements over time and to enable both low-cost and high-performance implementations within the same technology generation. As shown by the equation regarding the vector register length, if we can scale up the number of lanes or the number of functional units, we must either increase vector instruction issue bandwidth or increase the vector register length to sustain peak performance. Increasing vector instruction issue rate allows the same vector length to be maintained across a range of implementations, but this can require expensive issue logic. Rather than fix a vector register length for all implementations, the vector register length can be made implementation dependent. Thus, high performance, many lane vector units can then use longer vector registers than low cost, few lane vector units. There are several ways to make the vector register length hidden from the application.

The first approach to making the vector register length transparent to the programmer is a software one. For compatibility, an efficient run-time scheme is required to allow the same object code to run on machines with different vector register length. A machine's standard application binary interface can define a dynamically-linked library holding constants related to machine vector register length. It requires no hardware support, but does require an operating system capable of supporting dynamically-linked libraries (this is trivial nowadays, but in the early days of vector computers it wasn't). Also, loading this information from memory before using it to make the first control decision for a stripmined loop adds startup overhead.

The IBM System/370 has a different approach. The architecture supports implementations of vector register lengths between 8 and 512 elements and has added a new instruction "load vector count and update" (VLVCU) to control stripmined loops. This instruction has a single scalar register operand which specifies the desired vector length. The instruction reads this scalar register and it writes the Vector Length register and also the source register.

Before the loop is executed the scalar register is initialized with the number of elements that will be processed (this is equal to the number of times the loop would be executed on a scalar processor). The VLVCU instruction works like this:

- If the value of the scalar register is larger or equal to the implementation vector register length, then the Vector Length register is set to the implementation vector register length and the scalar register is updated with the old value minus the Maximum Vector Length. This is a normal execution of the loop.

- If the value of the scalar register is smaller than the implemented vector register length, then the Vector Length register is set to the value of the scalar register and the scalar register is updated with the value 0. This is the last execution of the loop.
- If the value of the scalar register is zero, then the loop exits, all the array elements have been processed.

This instruction has two problems for a modern RISC pipeline. First, it places a comparison and a subtraction in series which is unlikely to complete in a single machine cycle. Second, it requires several additional instructions to determine the vector length currently being processed; this length is often required in more complex vector code to update address pointers or to check for early exits from speculative vector loops.

An alternative scheme, more appropriate for a RISC pipeline is to provide a "saturate to maximum VL" scalar instruction. This *setvlr* $< dest >< src >$ instruction takes a scalar register as the only source operand and writes a destination scalar register with the minimum of Maximum Vector Length register and the scalar source value treated as unsigned integer. The same value is simultaneously written to the Vector Length Register. When the saturated vector length is not needed in a scalar register, the result can be dumped to the scalar zero register. A separate subtract instruction is needed with this scheme. The result of the scalar subtraction is needed only the next time the loop executes and the scalar unit has plenty of time to execute it while the vector unit execute the body of the loop. Using this scheme, the loop would use a branch on not zero instruction. Note that the Instruction Set Architecture should specify that vector instructions perform no operations if the Vector Length register is set to zero.

While writing both the Vector Length register and a scalar register with the same result is straight forward for an in-order machine, for machines that perform register renaming it can be troublesome for an instruction to produce two results. The operation can be split in two parts, one that performs the saturation in a scalar register and a conventional move to Vector Length register instruction. These two instructions have a data dependency and cannot be executed simultaneously adding at least a cycle of latency before the vector instructions can be dispatched.

To manage implementation-dependent vector lengths, it is also desirable to provide instructions to read this implemented register length and also the logarithm of the value and write it to a scalar register.

It is possible to support implementation-dependent vector register length with a reconfigurable vector register file (these will be discussed in the next section). The configuration would first be set by selecting the number of registers then, the above techniques can be used to determine the configuration vector lengths.

2.1.1.3 Number of Vector Registers

Traditional vector machines have had a relatively small number of vector registers, typically 8. All the Cray vector machines and also the mini supercomputers such as Convex

C series and Alliant only have 8 logical vector registers. The limited number of vector registers was initially the result of hardware costs when vector register instruction sets where originally being developed. Today the small number of registers is generally recognized as being a shortcoming.

For architectures that have a limited number of registers at the ISA level a serious problem is spill code. The small number of logical registers available to the programmer or compiler induces extra traffic between the processor and the next memory level because when the compiler runs out of registers it must insert a store instruction to spill some register to the stack and, later, insert a load instruction to reload the data.

Spill code can be very detrimental to performance due to two main reasons: first, when a vector register is spilled it involves a very large amount of data movement and, second, the distance, in terms of processor cycles, between a vector register and main memory can be rather large.

Several situations can force the compile to insert extra load/store instructions:

- Large loops that compute more temporary values the can be held in registers force the explicit introduction of store and load pairs of instructions.

- A register may be loaded from memory, but the data can not be saved for long because the register is needed for some other instruction. Then, when the memory data is needed again, it must be reloaded.

- Variables that are reused between close iterations of a given loop are difficult to keep in the register file due to the high register pressure.

- Call/return sequences, typically to intrinsic math routines, might involve saving and restoring of vector registers.

The number of registers also dictates the optimizations that a compiler can use. As presented in [27], most vector compilers use a scheduling technique known as simple vector scheduling (SVS). With SVS all instructions of one loop iteration are issued before any succeeding iteration begins. Long vector register is the primary mechanism for increasing pipeline utilization. A more advanced scheduling technique is polycyclic vector scheduling (PVS), that executes multiple loop iterations concurrently. PVS, for vector computers, is an adaptation of polycyclic scheduling, PS, used in scalar machines. It generates code interleaving instructions from multiple iterations of a loop, using registers to buffer intermediate results from all concurrently executing iterations. It can be seen as an advanced version of loop unrolling.

By combining code from multiple iterations of the same loop, consecutive instructions with data dependencies can be scheduled further apart so that the first instruction ends execution before the second one starts. Traditionally data dependencies are made tolerable by using chaining. Loop unrolling, on vector machines with many registers, combined with shorter registers lengths can reduce hardware complexity, especially concerning chaining.

Flexible chaining implies that any instruction can chain its results to any other instruction. Remember from the chapter on the traditional vector architectures that there are three types of chaining:

- From a load to a arithmetic or logic instruction
- Between two arithmetic or logic instructions
- From an arithmetic or logic instruction to a store instruction

The freedom of scheduling instructions, offered by a large number of registers, can relax the chaining requirements thus reduce hardware complexity. For instance, there are certain machines, like the Convex C3, that do not support the first type of chaining. Others, like the Cray-2 do not support chaining at all.

So far, we have presented strong arguments for increasing the number of logical registers, for having more than the 8 registers that traditional machines offer. But this is not always possible. One of the most important factors is the code compatibility. All changes to the Instruction Set Architecture must be incremental. That is, extended ISAs must be binary compatible with previous ones and, since more registers are encoded with more bits, increasing the number of registers cannot be done while maintaining backward compatibility.

Hardware techniques can be used to alleviate the small number of registers, while maintaining binary compatibility. We can adopt the tricks used in scalar processors. For instance, the x86 ISA has just a few architectural registers and still is alive and well. The most widely used technique is register renaming. Spill code can be reduced by using a victim cache, a special buffer used only for spill stores. More on these techniques in the chapter that presents the control part of the vector processors.

There are also disadvantages to having more vector registers. As we shall see in the section dedicated to the organization, the register file needs many read/write ports for each register. As we add more registers, the interconnection network grows in size, delay and even complexity, thus the clock period has to be increased or allocate more cycles for transferring data in and out of registers. None of these is desirable. Organizing register files in banks further complicates matters as there are constraints that the interconnection network must satisfy.

Summing up, we can observe two approaches to register files.

The first one, the traditional one, has few, but long vector registers, typically 8 registers each containing 64, 128 or 256 elements. The key to performance is having long registers and chaining is crucial.

The second approach is a more recent trend of having more registers, each register being quite short compared to the traditional lengths, 32 registers, each containing 32 elements. These offer more flexibility, allow more sophisticated compiler optimizations and can reduce hardware complexity. Note that 32 registers, each 32 elements long can be seen as 8 registers, each 128 elements, thus this new approach can 'simulate' the traditional approach with little overhead.

Of course, there is the middle approach, reconfigurable register files, to be discussed next.

2.1.1.4 Reconfigurable Vector Register Files

Most of the Japanese vector machines use a reconfigurable vector register file. The size is fixed, but the number of registers and their lengths is configurable.

Let us take a look at the Fujitsu VPP series. The total register file size is 128 kilo bytes and the machine word is 64 bits wide. This means that it can store in the registers 128KB/8B = 16K elements or machine words. The configurations supported range from 8 vector registers each holding 1024 elements to 256 vector registers, each holding 32 elements. The configuration of the register file is stored in special function registers and contribute to the state of the processor.

Where there are many temporary values, the vector register file can be configured to provide more vector registers to avoid register spills. When there are few temporary values, the register file can be configured to provide longer vector registers to better amortize overheads. This scheme offers more flexibility than the traditional approach but it doesn't come for free.

There are several disadvantages to a reconfigurable vector register file.

- Control logic to manage vector register dependencies, chaining, and vector register file access conflicts is more complex.
- The configuration must be held as a state in the processor, thus extra instructions and hardware must be added to manipulate this state.
- Each subroutine is written for a particular configuration, thus that particular configuration must be set before calling the subroutine. Also, when passing arguments, the change of configurations must be taken into account.
- Context switches must also save/restore the register file configuration.
- Organization in multiple lanes or banks further complicates the routing logic.

2.1.1.5 Context-Switch Overhead

Both lengthening vector registers and adding more vector registers increases the amount of processor state that must be saved and restored on a context switch. Although vector machines have high memory bandwidth to help speed these transfers, they might still represent significant overhead. Several techniques can be utilized to minimize this impact. All of these techniques keep information about the usage of the vector register file in separate hardware bits.

One technique is to add a hardware valid bit for each vector register. The valid bits would be cleared on process startup, and set whenever a vector instruction writes a vector register. At the context switch only the valid registers would be saved and then restored. This approach reduces register save and restore overhead for a process that doesn't need to use all the vector registers. A user-level ”vector state invalidate” instruction can be provided, whose semantics are to make the value of all vector state undefined. This instruction clears all the valid bits to inform the kernel that the user process no longer requires the state in the vector registers. Programming conventions usually classify vector register state as caller-save, and so software ca add a vector state invalidate instruction at the exit of every subroutine.

Further reductions in register save overhead are possible if we also keep track of a hardware dirty bit per vector register which is set whenever a vector register is written.

The operating system kernel can check these bits and only save those registers that have been modified since the last context swap. These enhancements are particularly useful for real-time multimedia applications where high priority processes are invoked repeatedly to process chunks of data but do not need vector register state preserved between activations. To support efficient user-level context switching, the dirty and valid bits should be made available to the user code. Both valid and dirty bits are part of the IBM System/370 vector architecture.

The valid and dirty bits work at the granularity of the whole vector registers, thus are useful mostly for machines that have many registers. In the case of 8 architectural registers, it is very likely that all registers are used by every process. The extra hardware to keep track of the register changes might not be worth the effort unless additional information is also stored. To avoid overhead on processes that use only short vectors we can keep track of the maximum vector register lengths used by a process. We can keep a single maximum length for all vector registers, or some intermediate grouping of vector registers to maximum length values. The kernel can use only the observed maximum length to save and restore a vector register.

2.1.2 Vector Register File Implementation

Let us start by looking at the requirements of a vector register. First we shall determine the parameters required for full performance. We shall see that the required complexity is quite high, costly to implement thus, we shall look at ways to reduce cost with minimum performance loss.

The interface of the vector register file is very similar to the one of a scalar register file. Each vector functional unit requires two read ports and a write port. Each vector memory unit requires a read port for stores, a write port for loads. Vector store and load instructions also have two scalar source operands (the base address and the stride) that come from the scalar register file. We shall not discuss the implications on the scalar register file here. Scatter/gather operations are indirect accesses to memory and require a vector source operand thus another read port is required. Thus, a memory unit needs two read ports and a write port.

Concluding, each resource has the following requirements to the register file:

- A functional unit needs two read ports and a write port
- A memory unit also needs two read ports and a write port

Let us take an example of a machine that has two functional units and a memory unit and we want to support all kinds of combinations of instructions, meaning all the resources can work in parallel and access any of the registers. For this case, the register file must have six read ports and three write ports. Note that the number of read/write ports is independent of the number of registers and their length. If we where to consider a scalar processor with the same number and type of resources, the number of read/write ports would be almost the same, just a read port less because of scatter/gather operations. The similarities stop here.

A vector register contains tens or hundreds of elements, a read/write port will be busy reading/writing all of the elements of the register, one element per cycle. To do this, there has to be some sort of internal control to go from the first to the last element. Not only this, but several instructions may access the same cycle the same register but different elements.

Let us have an example, we shall focus on the vector register rV1:

```
1   mulV        rV2, rV1, rV1
2   addV        rV1, rV2, rV1
```

The mulV instruction reads all the elements of register rV1 starting from the first till the last, one per cycle. After a number of cycles dm equal to the latency of the pipelined multiplier the corresponding element (result of the multiplication) is produced and written to rV2 and at the same time chained to the addV instruction. The addV instruction reads from rV1 the corresponding element, which is located dm positions behind the element that the mulV is reading. After a number of cycles equal to the latency of the pipelined adder, the addV produces the result and writes it back to rV1.

In this example we can see that, each cycle, two different elements of rV1 are read and a third, different one, is written.

Let us take a look at a second example focusing on the same register rV1:

```
1   mulV    rV2, rV1, rV1
2   loadV   rV1, X
3   addV    rV1, rV1, rV1
```

Following the rationale in the previous example we see that, each cycle, one element of register rV1 is read and two different elements are written.

If we want all combinations of instructions to execute in parallel accessing any vector register, we need every read/write port to have access to any of the elements stored in vector register file and all of these accesses to be done in parallel. This is quite a challenge since for our simple machine proposed earlier it would mean eight simultaneously unrestricted accesses to any of the register file elements (we have six read ports and tree write ports but the memory unit will never read/write more than two registers at a time).

A straight forward, but expensive, implementation of a vector register file would be as a single multiported memory. This would give each resource independent access to any element of any vector register at any time. This approach is extremely expensive and not easily scalable, by adding more functional units and/or memory units the complexity of the register file would grow more than linearly. Implementation costs can be reduced considerably by using two orthogonal techniques that take advantage of the regular pattern in which vector instructions access register elements. First, the vector register file storage can be split into multiple banks, with each bank having fewer ports. Second, each port can be made wider to return more than a single element per port.

2.1.2.1 Partitioning the Register File in Banks

There are two orthogonal ways in which the vector register storage can be divided into banks. The first places elements belonging to the same vector register into the same bank, this is called register partitioned. The second places elements with the same

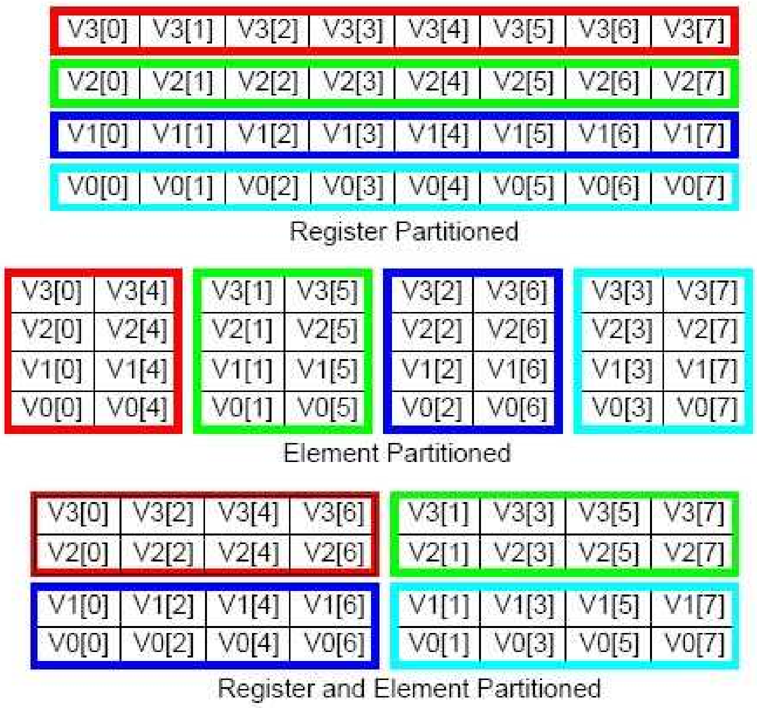

V3[0] V3[1] V3[2] V3[3] V3[4] V3[5] V3[6] V3[7]
V2[0] V2[1] V2[2] V2[3] V2[4] V2[5] V2[6] V2[7]
V1[0] V1[1] V1[2] V1[3] V1[4] V1[5] V1[6] V1[7]
V0[0] V0[1] V0[2] V0[3] V0[4] V0[5] V0[6] V0[7]
Register Partitioned
V3[0] V3[4] V2[0] V2[4] V1[0] V1[4] V0[0] V0[4]
V3[1] V3[5] V2[1] V2[5] V1[1] V1[5] V0[1] V0[5]
V3[2] V3[6] V2[2] V2[6] V1[2] V1[6] V0[2] V0[6]
V3[3] V3[7] V2[3] V2[7] V1[3] V1[7] V0[3] V0[7]
Element Partitioned
V3[0] V3[2] V3[4] V3[6] V2[0] V2[2] V2[4] V2[6]
V3[1] V3[3] V3[5] V3[7] V2[1] V2[3] V2[5] V2[7]
V1[0] V1[2] V1[4] V1[6] V0[0] V0[2] V0[4] V0[6]
V1[1] V1[3] V1[5] V1[7] V0[1] V0[3] V0[5] V0[7]
Register and Element Partitioned

Figure 2.1: A register file partitioned by registers, by elements and by registers and elements

element index in the same bank, this is called element partitioned. Figure 2.1 illustrates both forms of register file partitioning and also a hybrid or register partitioned and element partitioned.

By interleaving vector register storage across multiple banks, the number of ports required on each bank can be reduced. A separate interconnection network connects banks with functional and memory units. Since from the register file point of view read/write ports from the vector functional units and the vector memory units act in the same way, for simplicity, in this section we will use the term functional units for both.

In effect, all of these bank partitioning schemes reduce the connectivity between element storage and functional units ports from a full single-stage crossbar to a more limited two stage crossbar. This reduces implementation complexity but has the drawback of possible element bank access conflicts.

With a register partition scheme, a functional unit reserves a bank port for the duration of an instruction's execution. If another functional unit tries to access the same bank using the same port, it must wait for the first instruction to finish execution.

Because a single functional unit monopolizes a single bank port, opportunities for chaining through the register file are limited to the number of available ports on a single bank plus additional bypass paths around the functional units themselves. Also, complications arise in a register partitioned scheme if there is only a single read port per bank and there is more than one vector register per bank. The hardware will either be unable to execute a vector instruction with both vector register operands in the same bank, in which case software must avoid this contention, or the instruction will run at half speed (due to time multiplexing of the single read port) which will complicate chaining with other instructions. Similarly, if a bank can only support one read or one write, then a vector register cannot be used as both source operand and a destination operand within a single vector instruction. The Cray-1 implementation had a register partitioned scheme with only a single read or write port per vector register.

With an element partitioned scheme, a functional unit executing a single instruction cycles through all the banks. This allows all functional units to perform multiple chained accesses to the same vector register even though each bank has a limited number of ports.

The bank in which element i is found is given by the relation

$$bank\ number = i\ mod\ (number\ of\ banks) \tag{2.2}$$

If there are more elements than banks, then the functional unit will make multiple accesses to each bank. A subsequent vector instruction beginning execution in another functional unit might have to stall while the first functional unit performs an access to the first bank, but this stall will only last for one cycle. After this initial stall, both functional units are now synchronized and can complete further register accesses without conflicts.

One complication with an element partitioned scheme is that some operations (vector-vector arithmetic and indexed stores) require two elements with the same index but from different vector registers each cycle. There are three possible approaches to solving this problem.

The simplest is to have at least two read ports in each bank.

Another solution is to read element 0 of the first vector operand in one cycle and hold this in a buffer, then read element 0 of the second operand in the next cycle while simultaneously reading element 1 of the first operand from the second bank. This adds one cycle of startup latency if there are no other conflicts with executing instructions but after this initial cycle the vector accesses complete at the rate of one per cycle.

If the number of banks is at least equal to the number of vector registers, then a third approach is to place element 0 of each vector register in a different bank. This allows element 0 of both operands to be obtained from element banks with single read ports, without any additional startup latency, and with subsequent accesses cycling through banks as before. For this organization, the formula presented above becomes:

$$bank\ number = (i + reg\ number)\ mod\ (number\ of\ banks) \tag{2.3}$$

The main disadvantage of element partitioning versus register partitioning is that control is more complicated, especially in the presence of functional units that run at different rates, or that experience different patterns of stall cycles. Stalls are usually only generated for memory accesses by memory units, but theses will also cause any chained functional unit to stall.

For systems with a single memory unit, a simple solution is to stall the whole vector unit on any memory stall thereby preserving the relative access patterns of all functional units. This allows all potential vector register access to be resolved once at instruction issue time.

Another simple approach is to use a fixed rotating schedule to connect element banks to functional units. Again, this allows conflicts to be resolved at issue time, and also guarantees good performance when all functional units are active. The main disadvantage of this approach is that it increases the length of any register access conflict stalls. The startup latency increases because a functional unit has to wait until the schedule rotates around to give it access to the first bank, even if no other functional unit was active.

An alternative approach that still allows all conflicts to be resolved at issue time is to disallow chaining of loads into other operations and provide separate dedicated ports on each bank for memory unit access. No memory unit stall can affect the operation of functional units. Stores can still be chained onto other operations.

The most general solution is to resolve access port conflicts cycle by cycle, giving priority to the oldest instruction for any stall. One problem with this scheme is that dynamically assigned port access schedule might be suboptimal, a certain dynamic assignment of functional units to element banks might prevent a further functional unit from beginning execution whereas a different assignment could support all simultaneously. Increasing the number of available bank ports relative to the required number of functional unit ports decreases the probability of hitting a bad port access schedule, as does providing separate read and write ports.

2.1.2.2 Wider Element Bank Ports

Another way of increasing vector register file bandwidth is to make each access port wider, transferring more than one element per cycle. Each functional unit read port needs a buffer to hold the multiple words read each cycle until they are needed, and each functional unit write port needs a buffer to hold the result elements until a whole line of elements can be written to the register file.

The main drawback of this scheme is that it can increase the latency of chaining through the register file, as a vector register read must wait for all elements in a row to be written. Bypass logic can help by allowing a dependent instruction to read a value as it is produced, before it is written to the buffer. This will only work if the chained instruction issues at the right time or the buffer has two elements. A more general approach would be to read the appropriate element from the buffer but this would complicate control even more.

The use of wide access ports can be combined with any bank partitioning scheme. The available read or write bandwidth from a vector register file must be at least equal to the bandwidth required by the attached functional unit.

2.2 Vector Functional Units

This section is dedicated to the particularities of the vector functional units. The presentation is based on the differences from the scalar functional units.

Several algorithms used in computer arithmetic are discussed and for each of them mentioning how suitable they are for implementation in a vector functional unit.

The functional units are tightly linked to the way the register files are designed. The final part of the chapter presents an approach that is different from the traditional one discussed until now. The novelty of the proposed design is not in the functional units themselves but in the way the register file is organized around them.

2.2.1 General Issues

Arithmetic pipelines for a vector processor can be made identical to those for a scalar processor, a low cost design will likely share expensive floating point units between scalar and vector units.

A high performance design, however, provides separate functional units for scalar and vector operations enabling separate optimizations. The scalar functional units are optimized for low latency, while the vector functional units are optimized for maximum throughput per unit area. Starting from a scalar functional unit, there are two orthogonal ways to improve throughput at the cost of increasing the latency. The first saves area by removing latency reduction hardware, increasing the number of cycles required to produce a result at the same clock rate. The second increases the clock rate by deepening the pipeline with extra latches.

Scalar functional units are designed to have a very low latency. Back to back operations are common in scalar workloads thus, scalar microprocessors clock cycle times are generally fixed by the time to propagate through an integer add and forward the result for the next addition. In contrast, vector operations within one instruction are known to be independent and so offer more freedom when designing vector functional units, throughput is far more important than latency.

Current scalar floating point units require considerable resources to reduce the latencies, to as low as 2 clock cycles. As described in [17] Vector functional units can keep the same throughput by increasing the latency and saving a great deal of area. In general, vector functional units implement algorithms that can be easily pipelined and that have very regular structure.

For scalar addition Carry Look-Ahead implementations are very widely used. They are very fast, quite regular but cannot be pipelined. In contrast, Carry Select adders offer the same performance but even better regularity and the great advantage of being easily pipelined. This makes the Carry Select adder the ideal candidate for a vector implementation.

When it comes to multiplications, scalar units typically use Wallace or Dada tree multipliers. They offer very low latencies at the cost of high complexity, high irregularity and large area. The array multiplier has a larger latency, but a very regular structure and low area. It can be very easily pipelined offering a very high throughput. This makes it preferable for vector functional units.

Convergence algorithms are popular in scalar designs. These algorithms are used to execute operations costly to implement such as division or square rooting by replacing them with additions and multiplications, easier to implement. This process is iterative in nature. It starts with an initial approximation and with each iteration the error of the approximation is reduced. These algorithms have typically quadratic or better convergence rates that make them ideal for long operand widths. The iterative nature of the process means that it cannot be pipelined and thus, will not be used in vector functional units.

Scalar floating point units reduce the latency by normalizing results before magnitudes are calculated by predicting leading zeros. Pipelining can eliminate the need for such costly hardware, again trading latency for throughput.

A complete description of all the mentioned arithmetic algorithms is found in [18].

Reducing area occupied by the functional units is crucial for techniques that increase throughput by replicating resources. One of these techniques is the multiple lanes approach discussed in the next section.

The vector functional units due to, deeper pipelining, can run at higher frequencies than the scalar ones. One way to reconcile these opposing clock cycle demands in a tightly coupled system is to use a multiple of the scalar clock frequency in the vector unit. For example, the Cray-2, Fujitsu VP-200, Hitachi S810 run the vector unit at twice the scalar clock rate, while the Hitachi S-3800 runs the vector unit at three times the scalar clock rate. To simplify interfacing with the scalar unit and allow a relaxed timing specification for the vector unit control logic, a vector functional unit running at twice the scalar clock rate can be treated as if there where twice as many lanes running at the normal clock rate.

The IRAM project [6] developed at Berkley and Stanford Universities uses the greater degree of pipelining in the vector unit to maintain the same clock rate while supply voltage is reduced. This approach reduces energy dissipation in the vector unit, while allowing the scalar unit to run with a higher supply voltage to maintain low latency.

While the 64bit floating point arithmetic is a requirement for many supercomputing applications, many multimedia and human-machine interface processing tasks can be performed with 32bit floating point arithmetic, or even 16bit or 8bit fixed point arithmetic. This reduction in data precision enables a corresponding increase in processing throughput if wide datapaths are subdivided to perform multiple lower precision operations in parallel. The narrower subwords can be handled elegantly with a vector instruction set. For operations with narrower elements, the vector unit can be considered to have longer registers. For example, a machine with registers containing 64 elements each of 64bits can also be treated as having 128 elements each of 32bits or 256 elements of 16bits or 512 elements of 8bits. The maximum vector length becomes a function of the element width.

This technique was used in the Control Data STAR-100 vector processor, one of the early supercomputers, to provide higher processing speed for 32bit compared to the 64bit floating point arithmetic. A complete description can be found in [13]. This technique is used in all the multimedia extensions to general purpose scalar architectures. An extensive description of these multimedia extensions can be found in [5].

The functional units and the vector register files are very closely linked and often are co-designed. In the following section we present an approach that carefully optimize the organization for the efficient use of resources.

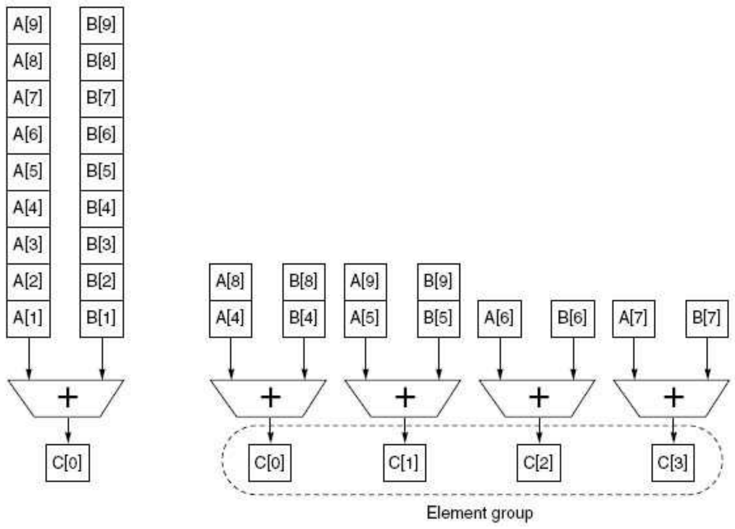


Figure 2.2: Using multiple functional units to improve performance of a single vector add instruction

2.2.2 Multiple Lanes

One of the greatest advantages of a vector instruction set is that it allows software to pass a large amount of parallel work to hardware using only a single, short instruction. The parallel semantics of a vector instruction allows an implementation to execute these element operations using either a deeply pipelined functional unit as discussed until now, or by using an array of parallel functional units, or a combination of parallel and functional units.

Figure 2.2 taken and adapted from [2] illustrates how vector performance can be improved by using parallel pipelines to execute a vector add instruction. On the left side of the picture, the machine has a single add pipeline and can complete one addition per cycle. On the right side, the machine has four pipelines and can complete four additions per cycle. The elements within a single vector add instructions are interleaved across the four pipelines. The set of elements that move through the pipelines together is termed an element group.

This concept is further detailed in the next figure, taken from [15]. It presents a general vector unit with an addition functional unit, a multiplication functional unit and a memory unit. The design is organized in four lanes.

We can see from Figure 2.3 that the functional units and the register file are organized in a clustered manner. The register file is element partitioned (See the chapter on register files) into a number of banks N equal to the number of lanes. The elements of the registers are interleaved N way, in the same manner as the memory address space is mapped to memory banks (described in the next chapter). Each of the vector arithmetic units contains N execution pipelines, one per lane, that work in parallel to complete a single vector instruction.

Summing up, organizing a vector unit in lanes implies replicating each of the execution units N times and having the register file element partitioned in N banks. In our

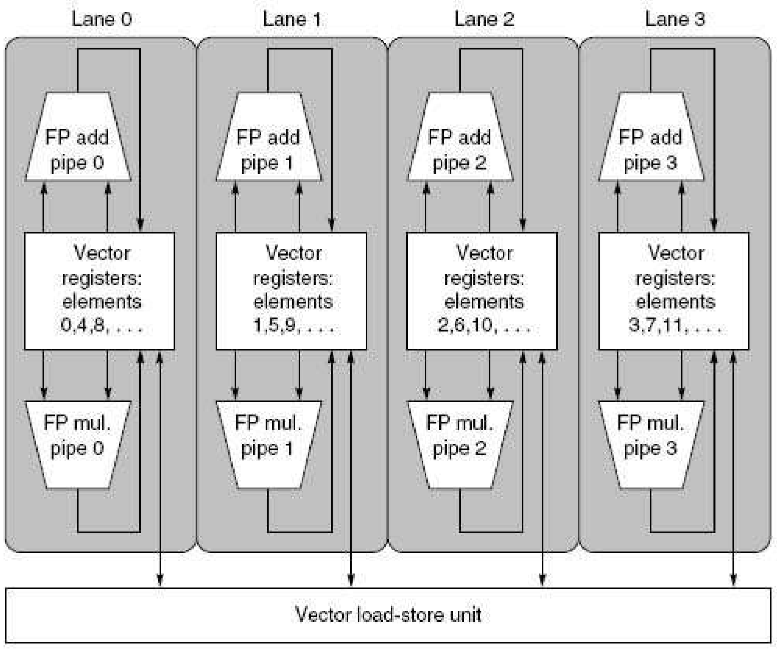

Lane 0
Lane 1
Lane 2
Lane 3
FP add pipe 0
FP add pipe 1
FP add pipe 2
FP add pipe 3
Vector registers: elements 0,4,8, . . .
Vector registers: elements 1,5,9, . . .
Vector registers: elements 2,6,10, . . .
Vector registers: elements 3,7,11, . . .
FP mul. pipe 0
FP mul. pipe 1
FP mul. pipe 2
FP mul. pipe 3
Vector load-store unit

Figure 2.3: Datapath of a vector unit containing four lanes

case, each of the lanes holds and processes every fourth element of each vector register.

Confusions can appear from the similarities of the terms functional and execution units. A vector instruction is executed by a functional unit. If the machine is organized in multiple lanes, the functional unit consists of multiple execution units, one for each lane. We use the term execution unit when we refer to a physical adder or multiplier. All the execution units that work in parallel to carry out an instruction form the functional unit.

Let us take a look at the implications of organizing the vector unit into multiple lanes compared to the traditional, single lane approach. A clear plus for this organization is that the implementation is entirely transparent to the Instruction Set Architecture. The throughput is increased proportional to the number of lanes, and the area is increased by replicating the execution units. Using the methods previously described for choosing execution units that require less area, the increase in chip size can be significantly smaller than the increase in performance, in other words, the performance per unit of area can grow substantially.

We have seen in the previous section that the register file lies at the heart of the vector unit and is the most difficult component to design. Partitioning it in banks simplifies the construction practically having more and smaller register files. The number of read and write ports is reduced since the register file bank has to read or write only the elements stored in that bank. For typical vector instructions there is no inter-lane communication, an element is stored and processed in its own lane. Thus, wires connecting the execution units and register banks are shorter, all computations and data transfers are local. The lanes work in parallel in lock step. The control signals are the same for all of the lanes.

Conceptually, having an N lane organization is similar to having N vector processors, each with shorter register lengths (N times shorter).

Concluding, adding multiple lanes is a popular technique to improve vector performance as it requires little increase in control complexity and does not require changes to existing machine code. Several vector supercomputers are sold as a range of models that vary in the number of lanes installed, allowing users to trade price against peak vector performance. The Cray SV1 allows four two lane CPUs to be ganged together using operating system software to form a single larger eight lane CPU.

2.3 Summary

In this chapter we have analyzed the vector components that are influenced by both the architecture and microarchitecture. The first section presents in the first half architectural aspects of the vector register file, while in the second half implementation issues.

Architectural aspects are related to what the programmer and compiler see. We have started the analysis with the vector register length. We have discussed the implication of the vector length on the instruction issue rate. We have seen that long vectors allow for a single instruction to be decoded and issued per cycle which in turn, allows a simple decode logic. Long registers also amortize the startup costs of vector instructions over many operations, thus providing an efficient mechanism to hide latencies.

The number of registers was also discussed. We have seen that vector machines typically have a small number of registers. This is mainly because the sheer size of the vector register file. It was decided that few longer registers are preferable to more, shorter registers. The small number of registers reduces the optimizations that compilers can perform and even worse, few intermediate values can be kept in the registers and many spill stores and loads must be performed.

We have seen that the Japanese vector machines propose a configurable vector register file. The total size is fixed but the number of registers and their size is configurable. This scheme offers a greater flexibility but increases both implementation and architectural complexity since the state of the processor is augmented with the configuration of the register file.

The overall size of the register file complicates context switching since large amount of data must be transferred. Techniques like the valid and dirty bit have been applied to reduce the memory traffic at context switches.

The second half of the first section has been dedicated to the implementation of the registers files. The discussion started from several examples that show the requirements of register files. We have seen that each functional and memory unit requires two read ports and a write port. Also, each of these ports must be able to access any element of any vector register. Even more, more access ports can access the same register but in different locations. From this initial analysis we can conclude that the vector register file is very complex and that implementing all the above mentioned features is very costly.

There are two methods to reduce the register file complexity with little loss in performance. The first technique is the use of multiple banks. The register file can be organized in banks register-wise (a bank contains several entire registers) or element-wise (a bank contains several elements from all of the registers).

The second technique uses wide register ports, each transfer in and out of the register file transfers several elements, thus accesses can be done more rarely allowing for time multiplexing of the read/write ports. This technique has the disadvantage that it complicates instruction synchronization and chaining.

In the second half of this chapter we have seen the particularities of the vector functional unit. We have seen that throughput is very important while latency is less of a concern. Compared to the functional units used in scalar processors we have seen that the vector ones are simpler, in terms of design complexity and also take up less area. The vector functional units must be very regular and must be pipelined. We have discussed several algorithms used in computer arithmetic and we have shown which are suitable and not for implementation in a vector functional unit.

We have seen that the register file is a neuralgic point of the vector unit. To this extent, we have presented co-designs of the vector register file and the functional units that reduce the complexity of the register file.

One of the most common techniques used to increase throughput of vector units is the multilane organization. We have studied this organization and we have also seen that in contrast to other techniques used to increase performance, the multi-lane organization does not increase overall complexity. The control complexity is very slightly increased since all lanes are identical and work in lock-step, thus there is just one set of control signals which is broadcast to all lanes. Since the data computations are kept inside lanes, and there is no inter-lane communication, the design of the data path is simplified and most importantly the register files are organized in totally independent banks (element wise) and have a substantial lower complexity.

Microarchitectural Components Investigated 3

This chapter is dedicated to the micro-architectural aspects of the vector processor. The first half discusses the memory system that is required to support the high throughput of the vector processor. We shall present the general approach to the memory system and point out the particularities with respect to the memory hierarchy used in scalar microprocessors.

The second half of the chapter is dedicated to the control part of the vector unit. Our goal is to keep the vector functional units busy at all times. For this we need to have enough instructions in flight and we also need to arrange instructions in such a way to hide the memory latency and to avoid stalls due to data dependencies.

First we analyze the front end of the vector unit and we show that a single vector instruction issued per cycle is more than enough even for vector processors with many functional units. We then look at other levels of parallelism: instruction level parallelism and thread level parallelism. We shall see that a combination of those brings the best results.

3.1 The Memory System

We shall start by presenting what the memory system must provide. The types of memory accesses are analyzed and two metrics to evaluate complexity are introduced. The access types will be compared in terms of these metrics. We shall see that some accesses are far more costly than others.

Other studies have already tried to find the frequency at which each access type occurs. The result of these studies are presented and discussed. This analysis is done in order to optimize, in conformance with Amdahl's law, the frequent access types and not spend design time and costly hardware speeding up instructions that will rarely be used.

Then, the common approach to the memory unit implementation and the corresponding memory organization are presented. All the vector supercomputers of the 80's and early 90's did not use data caches, the processor was connected directly to the main memory. The reasons for this design choice will be presented, and also difficulties that the memory system had to face and how they where solved.

Vector processors have more than one memory units to provide the huge operand bandwidth for the fast functional units. The suitable number of memory units are discussed in relation to the number of functional units and other machine elements like the degree of support for chaining and the number of registers. Also application requirements and statistics from benchmarks are taken into account.

Since multiple memory units are used and work in parallel certain rules must be enforced regarding a consistent state of the memory at any given time. Consistency models are presented, differences from the scalar model, impact on hardware complexity, several implementations and required compiler support.

The final section of this chapter discusses a more recent approach to the vector processor's memory system, the use of caches. The reasons that brought this change in design and how it affects the other elements of a vector machine are presenter. Also, the differences compared to a scalar cache are described and a general implementation is presented.

3.1.1 Introduction to the Vector Memory System

We have seen that vector processors have simple and regular functional units that provide a very high throughput. In order for them to work at full speed they require support from an equally fast memory system to transfer data between main memory and registers. The memory system has roughly the same characteristics as the functional units. The memory must provide a very fast flowing stream of data in and out of the processor. The bandwidth is the most important aspect. Large latencies can be tolerated because the startup time is mitigated over a large number of elements transferred. Keeping those elements flowing under any condition is the design goal. We shall see that there are situations when this is very hard to implement. For this reason the memory system represents a big fraction of the total cost of a vector processor.

Before going into any details the types of memory access that a vector unit can generate are presented. All of them cause the transfer of an entire register file:

- Unit stride access: words are read or written to main memory at consecutive addresses. This is the most simple and most common way vector processors access memory

- Non-unit strides: elements are read or written at addresses calculated by adding the multiples of an integer value (the stride) to the base address. The stride is expressed in bytes or words, depending on the implementation. The accessed memory locations have the following addresses:

  $$base,\ base + stride,\ base + 2 \cdot stride,\ base + 3 \cdot stride\ ... \tag{3.1}$$

  When the stride has the value 1, this is a consecutive address access, the unit stride access.

- Indexed access: these are Gather/Scatter instructions. They have as source operands a scalar register (the base) and a vector register (the indexes). The values in the source vector register are addresses relative to the base address where the memory is read or written.

The unit stride access is in fact a burst of data read or written at consecutive addresses. This is the most simple of the three access types in terms of hardware requirements.

The non-unit stride poses some difficulties but it has a very regular access pattern and can be implemented without major complexity increases.

Unlike the two previous memory accesses, the indexed access is a highly irregular one, there is no predictable pattern whatsoever and causes the most problems for implementing high bandwidth, low latency memory systems. In fact, we can consider an indexed access as a sequence of independent scalar memory accesses. The only thing that has to be enforced is the order in which the elements are written since multiple elements might write the same memory location. Thus, nearly all of the advantages of working in vector mode are lost.

In the following, address bandwidth is introduced, a general metric concerning the memory system, the. The rest of this section will be build around this new concept. Vector memory units execute vector memory instructions which move data between vector registers and memory. A single vector instruction is converted into some number of separate requests to the memory system.

Data bandwidth is the number of words that can be transferred per unit of time. Peak bandwidth values are quoted for streams of data stored contiguously in memory. The address bandwidth is defined as the number of non-contiguous memory requests that can be transferred per unit of time. A unit stride access is translated into a single memory request of VLMAX consecutive elements from memory. In this case the data bandwidth is far greater than the address bandwidth. On the other hand, an indexed access is translated into VLMAX memory requests each for a single element. In this case the address bandwidth is equal to the data bandwidth (note that we have considered the data bandwidth expressed in words, not bytes).

Address bandwidth is very expensive to provide because the actual address traffic must be supported by buses and control logic and also each individual memory access suffers the full memory latency. Variations in accesses without a predictable pattern might create contention and stalls. A clear case is the DRAM that naturally have less address bandwidth than data bandwidth, address lines are multiplexed and accesses are first by selecting a row and only afterwards selecting an element on that row.

One method of reducing the address bandwidth used in scalar processors is to use a memory hierarchy. On the top level of the hierarchy the address bandwidth is the same as the data bandwidth: each memory access transfers a single word. Between the L1 and L2 for each access an entire block is transferred, usually 4 words long. Between L2 and main memory even larger blocks are transferred for each access. In contrast, vector supercomputers typically allow a unique address for each word transferred throughout their memory system. The huge address bandwidth and associated crossbar circuitry adds a great deal to the expense of a vector supercomputer's memory system.

Concluding, the implementation complexity of the access types compared from the point of view of the address bandwidth and access pattern can be summarized as follows: The unit stride access has the smallest address bandwidth and the highest regularity: consecutive accesses. It is the simplest of the three access types. The non-unit stride has the address bandwidth equal to the data bandwidth but the access pattern is regular. It has moderate implementation complexity. The indexed access also has the address bandwidth equal to the data bandwidth but, this time the access pattern is totally random. It is the most complicated access type to support.

3.1.2 Analyzing the Frequency of each of the Access Type

In [2] data is gathered from studies on a variety of scientific codes running on several different platforms (Table 3.1). It shows the distribution of vector memory operations braking them into unit stride, non-unit stride and indexed accesses. Several of the studied applications are completely or almost completely unit stride. Some others perform the majority of their accesses using strides greater than one. Many programs don't use indexed accesses; where they occur, indexed accesses are usually a small fraction of the total.

It is interesting to notice that several application from the Perfect Club benchmark suite when compiled for different machines have different distributions of the unit stride, non-unit stride and indexed accesses showing how differences in compilers can affect these statistics.

Choosing the balance between address bandwidth and data bandwidth is an important design decision in a vector processor as it will have a large impact on final system cost and performance. Unit stride accesses dominate in vector supercomputers workloads, even though these systems have full address bandwidth. Apart from vector memory instructions measured in these studies there are other demands on the memory system, mainly I/O transfers that are usually performed as unit stride bursts. Although strided and indexed access patterns are a small fraction of the total memory workload they could dominate execution time if they execute too slowly.

3.1.3 Typical Implementation of a Memory System

The supercomputers of the '80s and early '90s where designed for maximum performance, cost was not an issue. Even though we have seen that indexed accesses are quite rare and that have significant hardware requirements, they have received a lot of attention. The high performance memory systems of supercomputers are built with the indexed access in mind.

The most striking design element is the lack of cache memory. Main memory can provide the bandwidth required by the vector processor but it has a large latency. When reading the memory, the data starts flowing many cycles after the first request. For scalar processors caches drastically improve memory access latency by exploiting spatial and temporal locality. The way vector processors work, crunching long arrays of elements, would put very high stress on the cache and so there are several architectural disadvantages to using caches for vector processors:

- Supercomputer workloads tend to be very large and have working sets much larger than feasible cache sizes, thus yielding very high data miss rates, very small temporal locality.
- Vector strides could exceed the length of a cache line, thus causing a miss for every memory access, not enough spatial locality
- Indexed accesses could have no locality at all, thus also causing a miss for every memory reference

Benchmark Name	Unit Stride (%)	Other Strides (%)	Indexed (%)
NAS Parallel Benchmarks on Cray C90 [25]			
CG	73.7	0.0	26.3
SP	74.0	6.0	0.0
LU	18.0	82.0	0.0
MG	99.9	0.1	0.0
FT	91.0	9.0	0.0
IS	73.4	0.0	26.6
BT	78.0	22.0	0.0
Hand-Optimized PERFECT Traces on Cray Y-MP [24]			
Highly vectorized group	96.9		3.1
Moderately vectorized group	85.3		14.7
PERFECT code on Alliant FX/8 [10]			
arc2d	54.4	44.4	1.2
bdna	7.1	84.5	8.4
adm	22.4	18.3	59.3
dyfesim	37.4	22.4	40.2
PERFECT Traces on Convex C3 [9]			
arc2d	80.0	11.0	9.0
bdna	78.0	17.0	5.0
flo52	72.0	28.0	0.0
trfd	68.0	32.0	0.0
SPECfp92 Traces on Convex C3 [9]			
tomcatv	100.0	0.0	0.0
swm256	100.0	0.0	0.0
hydro2d	99.0	1.0	0.0
su2cor	76.0	12.0	12.0
nasa7	25.0	75.0	0.0
Ardent Workload on Ardent Titan [11]			
arc3d	98.6	1.4	N/A
flo82	74.5	25.5	N/A
bmkl	50.4	49.6	N/A
bmkl la	100.0	0.0	N/A
lapack	100.0	0.0	N/A
simple	55.4	44.6	N/A
wake	99.5	0.5	N/A

Table 3.1: Summary of published analysis of vector memory accesses categorized into unit stride, non-unit stride and indexed

As mentioned before, the vector architecture, is generally tolerant to latencies. Therefore memory latency can be hidden efficiently without seriously degrading performance. To illustrate this, let us have a look at a sample code running on a scalar and also on a vector processor, in both cases without cache:

```
for i = 1 to n do
    A[i] = 2 * A[i]
```

On a scalar processor this would be translated in a loop that would look like this (suppose r1 contains the starting address of A, r2 contains the value r1+n+1):

```
loop:   load    r3, r1          # load A[i]
        add     r4, r3, r3      # perform multiplication by 2
        store   r4, r1          # store the new A[i]
        addi    r1, r1, 1       # increase the address pointer
        bne     r1, r2, loop    #
```

The load instruction suffers the full memory latency, the add instruction cannot proceed until the data comes from memory. The store and following instructions can execute without blocking the pipeline but the next time the loop reaches the load instruction again the processor is stalled by the memory latency. The processor sees n times the memory latency, one for each iteration.

On a vector processor the code will look like this (suppose rS1 contains the starting address of A, rS2 contains the value rS1+n):

```
loop:   loadV   rV3, rS1        # load a portion of A
        addV    rV4, rV3, rV3   # perform multiplication by 2
        storeV  rV4, rS1        # store the new values
        addi    rS1, rS1, VLMAX # increase the address pointer
        blt     rS1, rS2, loop  #
```

This time the load instruction is executed only $\lceil n \ div \ VLMAX \rceil$ times and so the memory latency is perceived significantly less than in the scalar case. Typical loops have many more instructions and thus, careful coding can hide the latencies even better. We can see that the vector machine has a clear advantage over the scalar processor.

Another very important reason for not using caches is related to technology. In order to be efficient a cache must be several hundred times larger than the size of the register file. Let us take a look at the scalar case and consider a typical RISC machine with 32 registers, each 32bits wide. The total size of the register file is 1024bits = 1Kb. A usual size for the Level one cache is 16kB that is 16 times 8 is 128 times larger than the register file size. The Level two cache is tens of times larger than the Level one cache. We could easily say that an efficient cache must be at least a thousand times larger than the register file. Now let us look at the vector case. The Cray Instruction Set Architecture has both few registers, just 8, and short vector lengths, VLMAX is 64. Typically vector processors operate on double precision floating point numbers, thus each element is 64 bits wide. The size of the register file is 8 registers times 64 elements times 64 bits per

	Bank							
Cycle no.	0	1	2	3	4	5	6	7
0		N						
1		busy	N+1					
2		busy	busy	N+2				
3		busy	busy	busy	N+3			
4		busy	busy	busy	busy	N+4		
5			busy	busy	busy	busy	N+5	
6				busy	busy	busy	busy	N+6
7	N+7				busy	busy	busy	busy
8	busy	N+8				busy	busy	busy
9	busy	busy	N+9				busy	busy
10	busy	busy	busy	N+10				busy
11	busy	busy	busy	busy	N+11			
12		busy	busy	busy	busy	N+12		
13			busy	busy	busy	busy	N+13	

Table 3.2: Memory access by bank and time slot for a memory system organized in 8 banks, 5 cycle latency and unit stride access

element. That is 32kbits, 4kBytes. An efficient cache would have around 4Mbytes, a value that in the '80s and early '90s was unconceivable. Concluding, we have seen that for early vector processors caches where not a viable option.

Next, the typical organization of the memory system is presented. It is implemented using SRAM technology and uses a large number of independent memory banks. Consecutive addresses are mapped to consecutive banks, the address space wraps around the number of banks. The mapping of addresses to banks is given by the following equation

$$Bank\ number = address\ (in\ words)\ \mathbf{mod}\ number\ of\ banks \tag{3.2}$$

How many should the banks be? For unit stride accesses a lower bound on the number of banks is determined by the memory latency expressed in processor cycles. A unit stride access will cycle through all of the banks. Before the memory unit returns to a bank, that bank must have completed its previous request and must be ready to accept a new one.

Let us have an example (taken and adapted from [15]). Suppose we want to read from memory a vector stored at consecutive addresses and we want to receive a word every cycle. Suppose also that each memory access takes 5 clock cycles. That means that we need at least 5 banks. We shall choose the closest power of two, thus we shall have 8 banks and we can serve a unit stride access by reading a word each cycle, without any stalls. Table 3.2 shows the timing for the first accesses. For simplicity the address is expressed in words, not bytes.

Non unit stride accesses cause problems. Let us consider the previously described memory organization, this time accessed with a stride of two. We read only every other

	Bank						
Cycle no.	0	1	2	3	4	5	6
0		N					
1		busy		N+2			
2		busy		busy		N+4	
3	N+6	busy		busy		busy	
4	busy	busy	N+8	busy		busy	
5	busy		busy	busy	N+10	busy	
6	busy		busy		busy	busy	N+12
7	busy	N+14	busy		busy		busy
8		busy	busy	N+16	busy		busy
9		busy		busy	busy	N+18	busy
10	N+20	busy		busy		busy	busy
11	busy	busy	N+22	busy		busy	
12	busy		busy	busy	N+24	busy	
13	busy		busy		busy	busy	N+26

Table 3.3: Memory access by bank and time slot for a memory system organized in 7 banks, 5 cycle latency and an access with a stride of two

bank. This situation is equivalent to a unit stride access on four banks. Since the access latency is of five cycles, we would get stalls and we could not read a memory word each cycle. If we use as stride of 8, then only one of the banks is accessed and data bandwidth is reduced to one eighth of the maximum possible bandwidth.

Generalizing, when the number of banks is a multiple of the stride, the number of used banks is reduced to the number of banks divided by the stride. The same situation occurs when the stride is a multiple of the number of banks. A first solution is to have a very large number of banks, much larger than the lower bound dictated by the memory latency. Since usually the strides are small values, in most cases, if sufficient banks are used, words can be read each cycle. This would mean replicating hardware that in many cases will not be used.

We can think of a more cleaver solution. We have seen that the problem occurs when the number of banks is a multiple of the stride. Thus, we can choose the number of banks to be a prime number. This way any accesses with a stride smaller than the number of banks would use all the banks. The only case when not all banks are used is when the stride is a multiple of the number of banks but this is highly unlikely. Very rarely applications need to access memory at addresses (added to the base address) that are multiple of a prime number. Figure 3.3 shows the timing for a stride two access on a memory organized in 7 banks, each having a latency of 5 cycles. Despite the advantages of the number of banks being a prime number, there are practical difficulties in the sense that binary logic is suited for powers of two and any other number of banks would under utilize the selection and routing logic.

Looking at the indexed accesses, since there is no predictable pattern of the addresses, we cannot use any mathematics tricks. The only thing we can do is add as many banks as possible in order to decrease the probability of accessing a busy bank.

Commercial implementations use a large number of banks, always a power of two. This way, both the non-unit stride and the indexed accesses have high probability of providing a result every cycle. For instance, The Cray Y-MP had 3 memory units, each capable of accessing a word per cycle, had 256 memory banks.

The introduction of the Synchronous Dynamic RAM has slightly changed the way vector memory system is organized. It still uses banks but this time within a bank the access time is no longer uniform, banks are divided in rows. Also, due to the dynamic nature of the SDRAM, refresh must also be taken into account when estimating timing. For space reasons this will not be discussed here, a detailed description of a particular SDRAM implementation can be found in [1]. The use of SDRAM chips to form a vector memory unit can be found in section 8.6 of [2].

3.1.4 Number of Vector Memory Units

A vector supercomputer typically has more memory units allowing for multiple parallel accesses to memory. How many should there be it is an important design decision. The memory units should be able to provide sufficient memory accesses to keep all the functional units busy.

First we introduce the memory to computation ratio. It is defined as the ratio of the number of memory units to the number of functional units and it is an implementation dependent characteristic. Most of the Cray vector supercomputers (X-MP, Y-MP, C90, T90) are using two load vector memory units and one store vector memory unit for every multiply and add unit. This gives a 3:2 memory to compute ratio in the implementation. It takes into account the worse case scenario like in the following example:

```
loadV   rV1, X
loadV   rV2, Y
mulVS   rV3, rV1, rV2
storeV  rV3, Z
```

The aforementioned Cray machines can execute all four instructions in parallel, the loads are chained to the multiplication which is in turn chained to the store. This ratio of 3:2 is the same ratio as a memory to memory machine. The use of registers reduces the ratio, the more registers, the smaller the ratio. More registers mean operand reuse while, in contrast, few registers may generate spill code and thus increase memory traffic.

The ratio is applied to programs as well, this time as the ratio of vector memory instructions to other vector instructions. Note that the application's ratio is dependent on the underlying architecture since it is defined as the ratio of assembly instructions and not of high level language constructs. Designers should take into account application requirements and build machines with memory to compute ratios close to average application memory to compute ratios. Of course, it is hard to determine the correct value and so, high performance supercomputers are prepared for the worst case, while mini supercomputers have an implementation ratio of one or less. The memory to compute ratio can only be taken as an estimate to compute the lower bound on the execution time of an application. It does not take into account data dependencies, lack of fully flexible chaining or other possible sources of execution delays.

To get a general idea, [21] reports that most of the applications from the Perfect Club benchmark suite, executed on a Convex C3 vector processor have a memory to compute ratio close to 1. The Convex C3 has 8 registers, two functional units and a single memory unit thus an implementation memory to compute ratio of 1:2. Judging from the ratio we would guess that the memory port is the bottleneck, that is fully utilized at all times and that the functional units are waiting for data from memory. What the same study observes is that, contrary to believes, the memory unit is often idle because compute instructions have numerous stalls. This is mainly because of the lack of support for full chaining and because of in order execution of instructions in the vector unit. We can conclude that the optimal number of memory units is highly dependent on other microarchitectural parameters like those mentioned above.

3.1.5 Vector Memory Consistency Models

As there are multiple memory units that work in parallel certain rules must be enforced regarding a consistent state of the memory at any given time.

A memory consistency model specifies the possible orderings in which concurrent memory accesses can be observed by the application or operating system. Consistency models can be considered at three possible levels [2]: between processors (inter-processor consistency), between instructions (inter-instruction consistency) and between elements within an instruction (intra-instruction consistency). Inter-processor consistency is beyond the scope of this thesis and will not be discussed.

3.1.5.1 Inter-instruction Consistency

Inter-instruction consistency model defines the possible orderings in which different instructions from the same instruction stream can observe each other's memory accesses. Scalar processors almost invariably adopt a strict consistency model for instructions in the same instruction stream, requiring that all memory accesses be visible in program order. In contrast, most vector Instruction Set Architectures adopt a very loose inter-instruction memory consistency model. Typically there is no guarantee on the order in which vector loads and stores are performed by a processor unless there is an intervening memory barrier instruction of the appropriate type.

```
1   storeV  rV1, X
2   loadV   rV2, X    # rV1 might not be equal to rV2

3   storeV  rV1, X
4   mem barrier
5   loadV   rV2, X    #rV1 is guaranteed to be equal to rV2
```

The aim of such a loose inter-instruction model is to allow a sequence of vector memory instructions to be issued to multiple memory units, where they can execute concurrently without the need for hardware to check for collisions between the multiple address streams.

Vectorizing compilers have to perform extensive compile time analysis of memory dependencies to detect vectorizable code, and they can pass on the results of the analysis to hardware by omitting memory barriers when unnecessary. This explicit inter-instruction consistency model provides a large cost saving compared to huge numbers of address comparators required to support the execution of many concurrent memory access that need to be visible in program order.

An important case when barriers can be omitted is between a load and a store to the same location if the store has a data dependency on the value being read. This is illustrated in the following example code:

```
1   loadV   rV1, X
2   mulVS   rV2, rV1, rV1
3   storeV  rV2, X          # no barrier required
```

Scalar memory instructions are also included in the inter-instruction consistency model. Scalar accesses in a vector machine are generally defined to observe other scalar accesses in program order to simplify scalar coding and to remain compatible with a base scalar ISA. But to simplify hardware implementations, the ordering of scalar memory accesses with respect to vector memory accesses can remain undefined except for explicitly inserted barrier instructions. Software can often determine that scalar and vector accesses are to disjoint memory regions and omit unnecessary barriers.

There are four basic types of memory access to consider when specifying barriers to maintain inter-instruction consistency: scalar read, scalar write, vector read, vector write. A possible design for the memory barrier instruction is to take into account all of the combinations of the four memory accesses and have special bits encoding the type of ordering that must be preserved (scalar read after vector write, scalar write after vector read .). Other machines have coarser control. For example, the Cray architecture only allows the user to specify that all vector writes must complete before following instructions, or that all vector reads must complete before following instructions.

Memory barrier instructions need not stall the issue stage immediately. Rather, they set hardware flags to prevent a subsequent memory operation from violating the desired constraint. For example, library routines might frequently end with a memory barrier to ensure that memory has the correct values at the end of the routine, but this will not stall issue unless the caller tries to access memory before the consistency conditions are satisfied. Computation instructions can proceed unrestricted.

Vector machines that execute instructions in program order and have a single memory unit used for both scalar and vector memory access can simply discard memory barriers when encountered in the instruction stream, there is no memory parallelism and all accesses are in program order already.

3.1.5.2 Intra-instruction Consistency

A single vector instruction specifies multiple parallel memory operations which might be executed concurrently across multiple memory banks. The memory unit might reorder these accesses to make better use of the particularities of the memory organization.

For unit stride and constant stride stores, the order in which elements are written is usually not important because the elements do not overlap except for the case of stride equal to zero that will write all elements in the same location or for very large strides that cause addresses to wrap around the address space. But there are algorithms which can only be vectorized if indexed stores write values in element order.

Applications generally do not require that vector loads are retrieved from memory in element order but doing so might prevent chaining. Some memory barriers can be avoided if we define that values that are read are chained to other computations in the exact element order. Guaranteeing element order should not impact implementation performance. The memory system can perform element load out of element order, as long as chaining logic ensures that load values are used in order. For stores, write buffers can drain out of order to memory provided that writes to the same location complete in element order.

3.1.6 Caches

The final pages of this section is dedicated to a more recent approach to the vector processor's memory system, the use of caches. The reasons that brought this change in design are presented and also a general description of the implementation is provided.

As we have seen in earlier sections, most vector supercomputers do not have cache vector data mainly because of the belief that program behavior in vector workloads precludes efficient cache operations and that vector computers can tolerate even high memory latencies.

A technique extensively used for hiding memory latencies is the careful ordering of instructions. This can be achieved by scheduling loads early enough that requested data reach the registers before it is used by arithmetic or logic operations, thus the machine can achieve a sustained performance without the need for a vector cache. This puts heavy stress on the vectorizing compiler and, in some cases, to reach the highest performance, the code must be tuned by hand at assembly level.

In time, the software has become more and more complex and this kind of optimizations has become more and more expensive to support. Also nowadays the accent is on code reusability and ease of programming. Also, in time, the processor frequency has increased steadily while the memory performance could not keep up the pace. This performance gap translates into longer memory latencies in terms of CPU cycle time. As memory latencies increase, the harder it gets to hide them efficiently.

Nowadays processor frequency has hit a wall at around 3GHz and CPU performance is increased by micro-architectural means rather than technological ones. This might give the memory time to reduce the performance gap compared to CPUs. But the rate at which memory access times are improving is reported in [15] to be around 5% which is quite discouraging. It is a too slow improvement rate and so new approaches must be attempted. Most importantly, Moore's law has offered designer each year more and more transistors and so, sufficiently large caches for vector machines can be efficiently implemented using today's technology.

Caches can compensate the large differences in speed between the processor and main memory but this is not enough. Unlike scalar processors, the typical memory behavior of vector processors is not suitable for caches. We have presented these aspects in previous sections but for clarity we shall briefly restate them:

- Supercomputer workloads tend to be very large and have working sets much larger than feasible cache sizes, thus yielding very high data miss rates, very small temporal locality.
- Vector strides could exceed the length of a cache line, thus causing a miss for every memory access, not enough spatial locality
- indexed memory accesses could have no locality at all, thus again causing a miss for every memory reference.

To cope with this problems there are two main solutions. The first is an algorithmic approach aimed at reusing data stored in the cache by splitting the work on more subproblems each using relatively small data sets that can fit in the data cache. The second approach is adding support in the Instruction Set Architecture for cache management. Since only some vector data will benefit from caching, the application can select which of the vector accesses to be cached and which not.

The first solution tackles the introduction of caches by changing the way applications are written to increase the locality of memory accesses. This is not an issue since large numerical applications can almost always be coded to execute efficiently in smaller memories.

Let us take the example of big dense matrix multiplication. Let us further consider that the two matrixes are square and that each line is much larger than the cache size, this is a common case in big scientific applications. We shall take a look at two possible implementations of the matrix multiplication algorithm.

First consider calculating the elements of the resulting matrix one at a time. This would mean reading a row of the first matrix and a column of the second one, multiply them element by element and the result of these multiplications reduced to a single value of the result matrix. Of course, the row and column would be too big to fit into a vector register and the loop would be stripmined, after each multiplication, new elements must be read from memory. Obviously, if we had a cache, using this method would create a cache miss for every access since we supposed that any of the row or columns are larger than the cache.

We can think of a more clever algorithm to solve the problem. We divide the two matrixes in sub blocks such that at least three of these sub blocks fit in the vector cache: the two source sub matrixes and the result sub matrix. In this way while computation is performed on these sections of the matrixes the data resides in the cache and most of the load/store accesses do not go to main memory. Then, the intermediate results must be combined to create the final result.

The algorithm is more complex but it is worth the effort because the time spent waiting for data transfers is significantly reduced. Besides, very similar ways of writing algorithms are used in parallel machines with non-uniform memory access (NUMA) where moving data between local and non-local memory is very expensive in terms of number of cycles. Also algorithms aimed at increasing operand reuse because of cache size have been developed for scalar processors, thus the increase in algorithm complexity has already been explored and solved.

We have stated earlier that optimizations can significantly improve the application execution time by carefully reordering instructions to hide memory latencies for vector memory systems that do not use caches. Note that this is related strictly to compiler techniques and assembly optimizations and is much harder to achieve since it is a customization of the application to the particularities of the underlying machine, changing the machine, requires, besides re-compiling, different optimizations.

The addition of caches has switched the application sensible points from the compiler back-end level to the algorithmic level. The slight increase in algorithmic complexity for vector memory systems using caches yields much better results and provides inter-platform application compatibility. Therefore, it is preferable to the compiler assembly code optimizations used in the past.

Another way to support the efficient use of data vector caches is a hardware approach by providing cache management instructions. These cache management instructions are used also in the real time machines where the execution time must be known precisely. Since the cache is probabilistic in nature, there is no way to accurately predict whether certain data is or not in the cache. In these real time systems, cache instructions can guarantee that certain data is in the cache, this in term guaranteeing an upper bound for execution time of certain critical program sections.

The cache management instructions in vector processors have a different aim, that of storing in the cache only data that will be soon afterwards reused thus, enforcing temporal locality. This prevents the cache from being polluted by data accessed sporadically evicting useful data already in the cache.

A possible scheme for distinguishing accesses that should be cache allocated use a cache hint bit on vector data instructions or some form of dynamic cacheability predictor. Other no-allocate vector accesses will still operate in cache if they are already in the cache but do not allocate data in the cache if there is a miss. We shall not go into any more details since the topic is too specific for the purpose of this thesis. For further description please refer to [2].

This hardware approach requires changes at the Instruction Set Architecture level. Additional bits must be provided for caching hints for loads and stores or even new instructions must be added. It might not be possible to keep backward code compatibility. Also, the instruction decode logic and cache control logic must support these operations, slightly increasing complexity. Therefore the algorithmic approach is generally preferred since it has fewer implications for hardware architecture and organization.

The following paragraphs present the slight changes in implementation details of vector data caches compared to the scalar case. These changes are dictated by the aforementioned memory behavior of vector processors.

Capturing spatial locality is less important than for a scalar data cache, because vector unit-stride or non-unit stride instructions already accurately encode the presence of spatial locality. This reduces the benefit of the long cache lines and speculative prefetching used to exploit spatial locality in scalar caches.

The main benefits of a vector data cache arise from capturing temporal locality. Reusing vector data from caches reduces data bandwidth demands on the lower levels of the memory hierarchy. Large capacity and high associativity, or other techniques to avoid conflicts, are key features of vector data caches that help capture temporal locality.

Vector data caches also improve performance by reducing memory latency, though, as we have seen, this is less important then in scalar processors due to the inherent latency tolerance of vector instructions. In particular, speculative prefetching to reduce latency may be beneficial for latency sensitive scalar processors but fetching additional speculative data can slow down a latency tolerant vector processor.

In [14] the authors compare several vector data cache architectures and simulated them on an Ardent Titan vector processor. They found that the most successful one was the Interleaved Cache Model. We will briefly present the most important characteristics of this approach. The scalar processor has a primary cache that is connected to a larger secondary cache. The primary scalar cache is interfaced to the secondary cache in the same way as the vector memory units.

To provide the full bandwidth that a vector unit requires, the secondary is interleaved N way. This technique is similar to the one presented earlier for traditional memory systems. Practically the cache consists of N single ported caches treated like banks.

This organization increases the potential bandwidth by a factor of N and allows multiple memory access ports to run in parallel, given a suitable interconnect. The crossbar connects on one side the N ports of the interleaved cache and on the other side the primary scalar cache and the vector memory units.

Cache consistency is easily maintained by making the contents of the scalar primary cache a strict subset of the vector secondary cache. This is done by having the scalar cache with a write through policy, writes of the scalar processor update both the scalar cache and the secondary vector cache. Vector stores need to check scalar cache to invalidate any matching lines. The secondary cache is managed using a write back policy to reduce main memory traffic.

A single memory instruction can touch many cache lines and main memory latencies can be very long, and so the secondary cache is non-blocking to allow multiple outstanding cache misses to be in progress simultaneously. Hardware must allow multiple vector accesses to the same cache line to be merged as it is common with merging scalar accesses in scalar processors.

3.2 The Control Unit

This section looks at ways of improving performance by increasing functional unit occupancy rates. We start by analyzing the front end of the vector unit and we show that the issue logic does not represent a bottleneck for performance.

Vector processors inherently exploit data level parallelism. In this section we also explore other levels of parallelism. We first look at instruction level parallelism that can be exploited through dynamic reordering of instructions. A basic method of dynamic scheduling is the decoupled architecture that splits the instruction stream into two other streams: computation instructions and memory access instructions. These two streams of instructions execute out-of-order with respect to each other but, each stream is executed in-order. The natural extension is the complete out-of-order execution of all instructions.

We shall also look at register renaming and a straight forward method of implementing precise exceptions.

Thread level parallelism is exploited in the multithreaded vector organization. We shall discuss the implications with respect to the increase in chip area and routing complexity. We shall also present three ways of combining instructions from the various threads executing in parallel.

The best performance is offered by a microarchitecture that uses all level of parallelism, more precisely an out-of-order multithreaded vector processor which will be discussed in the final pages of this chapter.

3.2.1 Issue Width

The analysis starts by looking at the issue width requirements of vector processors. The issue width is the number of instructions decoded and sent for execution to the functional and memory units. We have already had a preview of this discussion in the chapter dedicated to the register file when we placed a lower bound on the register length related to the issue width. The argument will be restated, this time focusing on the matter of interest here, the issue width.

The instruction decode and issue logic must provide enough instructions to keep all the functional units busy. In the best case scenario, when there are no data dependencies and no stalls of any kind all of the functional units must be kept working at all times. Since a vector instruction operates on the elements of a vector register, the execution time of the instruction is dependent on the vector register length. Also, the execution time is dependent on the number of lanes used in the implementation of the functional units. In general, the execution time of an instruction is given by the following equation:

$$Execution\ time = startup\ time + \frac{Vector\ register\ length}{Number\ of\ lanes} \tag{3.3}$$

The start-up time is implementation dependent and consists mainly of the number of cycles it takes the functional unit to produce the first result. This is due to the deeply pipelined manner in which the functional units are typically implemented and depends on the functional unit used. For example multiply units have longer startup times than addition units and floating point units have longer startup times than integer units.

The startup time must also take into account the number of cycles it gets for the operands to get from the register files to the inputs of the functional units (a large number of registers and a large number of functional units data transfers might require a crossbar with latency of more than one cycle).

Note that there are cases when the stream of elements might be stalled during the execution of an instruction. The most often situation is when a load instruction is chained to an arithmetic instruction and the memory unit cannot provide a new element in each cycle. In this case the arithmetic instruction must also stall, to work in lock step with the load. The above formula is just a lower bound on the execution time of a single logic or arithmetic instruction. When arithmetic instructions are executed back to back on the same functional unit, the startup time will be visible only for the first instruction, the arithmetic pipeline will be full at all times, even if during some cycles it will contain elements from different instructions.

Even in the best case, when there are no stalls at all, each vector instruction takes a large number of cycles to complete. We could say that the functional units 'consume' the instructions at a low rate. This means that the issue logic is not required to issue, or 'produce', the instructions at a very fast rate. Each instruction takes a large number of cycles to complete, there are several functional units to 'feed' thus, a single instruction issued per cycle is enough in most cases. To have precisely one instruction issued per cycle we need the following:

$$\frac{Average\ instruction\ execution\ time}{Number\ of\ functional\ units} \geq 1\ instruction\ issued\ per\ cycle \tag{3.4}$$

Coming back to equation 3.3 and neglecting the startup time, in order to have a single instruction issued per cycle we get:

$$Vector\ register\ length \geq\ Number\ of\ lanes\ \cdot Number\ of\ functional\ units \tag{3.5}$$

On the left hand side of the inequality we have the vector register length which is, in most cases, visible to the programmer and cannot be changed in different implementations of the vector unit.

On the right hand side of the inequality we have the number of lanes and the number of functional units. These numbers are implementation dependent. This will be the point where design decisions will apply. A very high performance implementation has both a large number of functional units and a large number of lanes. If vector register lengths are not sufficient, the issue width might have to be increased.

Note that in the discussion until now we have not included the memory units. We have chosen to do this in order to simplify the formulas. We can consider the memory units as being functional units with the correspondent of multiple lanes is an increased memory bandwidth transferring more words per cycle.

To get a better idea let us look at two examples. The Cray-1 vector processor has two functional units, one memory unit and has a single lane organization. The vector register length is 64 elements. This means that each instruction takes, neglecting startup latency, 64 cycles to complete. Thus, an issue width of one instruction per cycle is more than enough, actually, more than 90% of the cycles the issue logic is idle.

In contrast the Torrent architecture described in [3] has 32 vector registers each containing 32 elements. The T0 implementation has two functional units, one memory unit and is organized in eight lanes. This time each computation instruction, again neglecting startup, takes 4 cycles to complete, significantly less than in the previous example. A memory transfer instruction takes the full 32 cycles, equal to the number of elements in a register. If we consider that of all the vector instructions two thirds are computational instructions and the other third are memory instructions, the average execution time is:

$$\frac{2}{3} \cdot 4\ cycles + \frac{1}{3} \cdot 32\ cycles = 13.3\ cycles \tag{3.6}$$

Thus, we get:

$$\frac{Average\ instruction\ execution\ time}{Number\ of\ functional\ units} = \frac{13.3}{3} = 4.4 \tag{3.7}$$

This means that an instruction issued every 4.4 cycles is enough to keep all the functional units busy.

If we totally neglect the memory instructions and consider that all instructions are computation instructions we get the average execution time of 4 cycles and because we have two functional units we see that an instruction issued every 2 cycles is enough. By leaving out the memory units in our calculations, the constraints on the instruction issue rate have been tightened. Even with these stronger constraints a single instruction issued per cycle is enough for most vector implementation.

Concluding, unlike high performance scalar processors designs, where considerable effort is spent on parallel scalar instruction issue, vector instruction issue bandwidth is not a primary concern. A single vector instruction issue per cycle is sufficient to saturate many parallel and pipelined functional units. This means that designers can focus on other microarchitectural features to improve functional unit occupancy rate, features like the ones presented in the next pages.

3.2.2 Decoupled Vector Organization

Decoupled processors attack the memory latency problem by making the observation that the execution of a program can be split into two different tasks: moving data in and out of the processor and executing all arithmetic instructions that perform the program computations. This concept was introduced for scalar processors in [22]. A decoupled processor has typically two independent processors, the address processor and the computation processor, that perform these two tasks asynchronously and that communicate through architectural queues. Latency is hidden because the address processor is able to slip ahead of the computation processor and loads early in time the data that will be needed. This excess data produced by the address processor is stored in FIFO queues, and stays there until it is retrieved by the computation processor.

Decoupling achieves a limited form of dynamic instruction scheduling: a sequential instruction stream is fetched and split into the two streams mentioned. Each stream proceeds out-of-order with respect to the other, but instructions from a single stream are still executed in order. Decoupling increases the probability that the memory pipeline is always full. The combination of vector execution and decoupling gives a very powerful latency tolerant architecture.

3.2.2.1 Decoupled Vector Implementation

We have chosen to briefly present the decoupled vector architecture described in [20]. This implementation uses a fetch processor to split the incoming, non-decoupled, instruction stream into three different decoupled streams. Each of these three streams goes to a different processor:

- The address processor, AP, performs all memory accesses on behalf the other two processors
- The scalar processor, SP, performs all scalar computations
- The vector processor, VP, performs the vector computations.

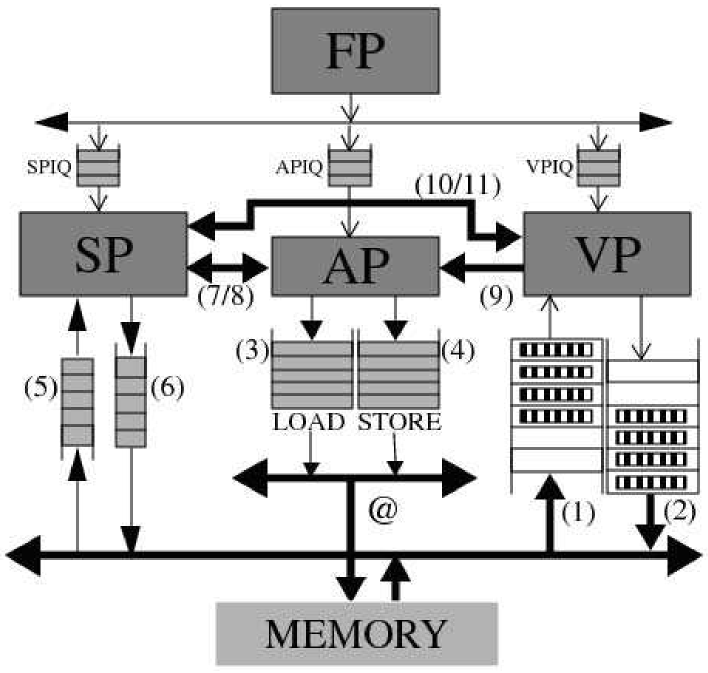


Figure 3.1: The decoupled vector organization

The fetch processor fetches instructions from a sequential instruction stream and translates them into a decoupled version. The translation is such that each processor can proceed independently, synchronized through communication queues. A block diagram is showed in Figure 3.1. In total, there are eleven queues used for inter-processor communication.

The fetch processor makes some modifications to the instructions. All memory accesses are converted into two pseudo-instructions. Also instructions executed in one processor that need operands from another processor are converted into pairs of pseudo-instructions, each going to the corresponding processor.

For instance, vector loads are transformed into:

- a load pseudo-instruction that goes to the address processor. The AP reads the data from memory and places it into the vector load data queue, VLDQ
- a transfer pseudo-instruction that goes to the vector processor. The VP when reaches this instruction reads the first element of the vector load queue, VLDQ, and puts it into the vector register specified by the transfer pseudo-instruction. In case the queue is empty it must wait until an element is written by the AP.

Vector stores and scalar memory accesses work in a similar fashion.

The total hardware added to the original design is the fetch processor and the communication queues. Each of the processors has a private decode unit. The resources inside each processor are the same as in a coupled design. Most of the queues added are scalar queues and, therefore, require a small amount of extra area. However, the vector load data queue and vector store data queue hold full vector registers, each slot contains Kilo Bytes of data (in the implementation presented in [20] each vector register occupies 1KB). One of the key points in this organization is to achieve good performance with relatively few slots in those two queues.

The address processor performs all memory accesses, both scalar and vector, as well as all address computations. Scalar memory accesses go first through a scalar cache that holds only scalar data. Vector accesses do not go through the cache and access main memory directly.

Decoupling improves resource utilization improving overall performance. Particularly by keeping the memory port occupied this organization reduces the impact of memory latency.

3.2.2.2 Memory Traffic Reduction

The decoupled organization has added more 'hidden' vector registers in the form of the Vector Load Data Queue and Vector Store Data Queue. These extra registers do not show in the logical register file space and there is no obvious way to take advantage of them to reduce the amount of spill code that the machine has to execute.

When a vector load instruction is inserted in the address load queue, it is disambiguated against all stores present in the address store queue. If a dependency is found with a certain store, the store queue contents must be sent to main memory before proceeding with the load. An interesting possibility is that the load might be identical to some store waiting in the queue. In such a case, when the data for the store becomes available, the data can by bypassed from the vector store data queue back into the vector load data queue and the load eliminated.

This bypassing is a limited form of data caching and has several advantages. First, the data being bypassed does not suffer any memory latency penalties. Second, during the bypass operation the memory port is idle and can be used by subsequent independent memory operations. On this second case, bypassing gives the illusion of having one extra memory port.

3.2.3 Out of Order Vector Execution

An evolution of decoupled execution is allowing the out of order execution among all types of instructions. Reordering among computation instructions can help in better using the functional units. Reordering among all memory instructions could also improve the memory ports occupation.

Out of order execution has been introduced in the IBM 360/91 scalar processor presented in [7]. The goal of out-of-order execution is to detect instructions that do not have dependencies and execute them in parallel. In order to do so, an out-of-order processor must fetch several instructions at a time and compare all their data requirements to determine a partial execution ordering that guarantees a semantically correct execution and maximizes resource usage. Modern out-of-order processors still use the same concepts developed by R. M. Tomasulo in [23].

A technique commonly used in out-of-order processors is register renaming. The implementation has more physical registers than the number of architectural ones. There is a front-end part of the hardware that maps architectural registers to free physical registers. The rest of the processor operates on the physical registers. The register file keeps the same organization as for in-order processors, the only difference is that it contains more registers.

Register renaming can increase the number of instructions that can be executed in parallel by eliminating read after write and write after write hazards. Also, since there are more physical registers, spill code can be reduced, reducing memory traffic. Both of these benefits lead to increasing resource usage and therefore performance. A detailed description of out-of-order execution and register renaming can be found here [15].

For vector processors, the potential advantages of out of order execution and register renaming can be harnessed at much lower cost than in superscalar processors. It is widely recognized that a major problem in the performance of the superscalar architectures is the bottleneck created at the fetch and issue state by the wide and complex instruction fetchers and decoders.

We have already seen in the beginning of the thesis that, for vector units, the instruction decode and issue phases are not of great concern. On the contrary, a single instruction issued per cycle is enough even for the vector processors organized in multiple lanes. Another thing that is in favor of the vector unit is the fact that the control intensive portions of an application (mainly branches and loop control instructions) are executed in the scalar unit of the vector processor. Thus, out-of-order execution in a vector processor can be implemented at a substantial lower cost compared to scalar processors.

Using out-of-order issue and register renaming techniques in a vector processor, performance can be improved greatly. Dynamic instruction scheduling allows better overlapping of memory latencies and uses the valuable memory resource more efficiently. Renaming enables straightforward implementation of precise exceptions, which in turn provide an easy way to support virtual memory, without much extra hardware and without incurring great performance penalty.

An out of order vector implementation is proposed in [19]. Apart from the vector registers, which are treated like in the scalar approach, the vector machine has two important registers that have to be taken into account: the vector length register and vector mask register. These two registers control de execution of all vector instructions. Renaming is clearly necessary because vector instructions are executing out-of-order and hence need to be able to identify their own vector length register and mask register.

There are two possible commit strategies. When releasing a physical vector register there are two possibilities. In short, a physical register can be freed when an instruction that releases the register finishes execution or, a more aggressive approach, when an instruction that releases the register starts execution.

A vector register contains a large number of elements that are read and written in order, starting from the first to the last. When an instruction that will free the physical vector register (is the last instruction that will read that register) starts execution, it will start by reading the first element of the vector register and read each subsequent element in the following cycles. This means that in the second cycle the first element is free, in the third cycle, the first two elements are free and so on.

When an instruction allocates a physical vector register, it writes the vector elements one by one, again starting from the first element. We can move the granularity of the free/allocate steps from vector register file to register elements. This means that a vector register can be at the same time freed and allocated but at different positions. In practice things are simpler. Once an instruction has started reading a register that will be freed, any instruction that writes to that register (allocates the register) cannot catch up with the reading instructions. Thus, when an instruction frees a vector register, the register can already be considered free in the cycle following the start of the freeing instruction.

This very aggressive commit strategy allows for best performance because registers can be reused faster and less physical registers are needed for maximum performance. However this strategy has a severe disadvantage, it does not allow for precise exceptions.

In contrast, a more cautious approach releases a physical vector register when the releasing instruction completes execution. The release of the register not only waits for reading the last element of the released register, but also waits for the last element of the destination register to be written. This vector register release strategy requires more registers to achieve the performance of the more aggressive previous approach but it offers a straightforward mechanism for precise exceptions.

Precise exceptions are required for virtual memory support and desirable for handling arithmetic faults. However, they are difficult to implement in vector processors. Whenever an operation causes an exception (a page fault or an arithmetic exception) the exact state of the processor must be saved, control transferred to a routine that solves the exception and then the previously saved processor state is restored. If an exception is generated in the middle of an instruction, then, after the exception has been solved, the instruction must be re-executed. However, in the case of the aggressive commit strategy this might not be possible since the first elements of one of the registers used by the instruction might have been overwritten by another instruction.

The loss in performance of conservative approach compared to the aggressive one is estimated by [9] to be approximately 10% which is a small price to pay for the benefits that precise exception can bring.

3.2.3.1 Decoupled vs. Out-of-order Execution

The main difference is the amount of instruction reordering allowed by the out-of-order execution organization. The limitation of the decoupled organization is that it does not allow instructions of the same type to be reordered. The stream of computation instructions is executed in-order, the stream of store instructions is also executed in-order and for the load instruction is the same. In contrast the out-of-order machines can reorder any type of instructions.

Another difference relates to the way in which data fetched in advanced is stored. Both types of organizations improve upon in-order execution by pre advancing instructions. In the Decoupled machines, the memory loads executed ahead of time are stored in queues that are regular and simple. In contrast, the out-of-order model uses the central register file to store this kind of data. Thus the register file size grows and enlarging the register file might have a negative impact on the crossbar complexity connecting vector registers and functional units.

The decoupled organization converts some of the instructions in pseudo-instructions, adding instructions that transfer data between processors. The out-of-order execution does not need this.

Out-of-order execution with register renaming can offer support for precise exceptions while the decoupled organization cannot offer this capability.

Concluding, both organizations improve the resource utilization, reduce memory traffic and increase overall performance. The decoupled version is simpler in construction compared to the out-of-order organization but the latter offers better performance because it allows reordering of all types of instructions and most importantly allows support for precise exceptions.

3.2.4 Multithreaded vector organization

In the previous sections we have seen that dynamic scheduling of instructions in the form of decoupled or out-of-order execution can improve the resource utilization by partially or fully reordering instructions. These two approaches aim at improving the Instruction Level Parallelism, the instructions that are executed in parallel are all from the same instruction stream, the same thread. One of the limitations of the dynamic scheduling approach is that there is a limited amount of ILP that can be extracted by reordering instructions, for details see [26].

However there is a higher level of parallelism available, the thread level parallelism. We can be certain that instructions from one thread are independent from the instructions of another thread. A processor that can execute concurrently instructions from two ore more threads is called a multithreaded processor. This concept was introduced in [12] and its main advantage is that it offers more independent instructions that can be executed in parallel.

There are three strategies for switching between threads. The first is to switch thread only when there is a long-latency operation, such as a cache miss, that threatens to halt the processor for many cycles. The second strategy is to switch threads each cycle or after a fixed number of cycles. In the first case, the switch is triggered when the executed thread fulfills a condition, for example several consecutive stalls, while in the second case the switch is triggered automatically by the processor. This is similar to preemptive and non-preemptive schedulers in operating systems.

The most aggressive multithreaded approach is not to switch at all, but rather have the threads running in parallel. Instructions from the different threads can be executing at the same time in the processor. This requires that the fetch unit be able to fetch instructions from the multiple threads at the same time.

Multithreading is orthogonal to the dynamic scheduling methods, meaning that processors can be implemented using multithreading or any form of dynamic scheduling or a combination of dynamic scheduling and multithreading. We shall see that a combination of multithreading and out-of-order execution provides the best performance.

Note that multithreading is used to improve the global throughput of the machine, it does not speed-up individual threads but rather it interleaves the instructions from these threads in order to hide processor stalls.

A multithreaded vector unit is presented in citeEV96. To implement multithreading the register files must be replicated as many times as supported contexts. This might be a cause of concern because vector register files are very large and complex. We have seen in chapter 2 that vector register files need a large number of read/write ports and those ports must access all registers and all the elements of the registers.

There are two approaches for the register file. Either maintain a single larger, monolithic, register file or add more separate register files. The first approach maintains part of the additional routing wires inside the register file, while the second approach transfers the complexity to the exterior crossbars. In chapter 2 we have already seen that even for few registers the internal routing logic is quite complex and several methods are used to reduce the complexity by using more banks. Thus, the method generally used is to have multiple separate register files and so, with regard to the register file the area grows linearly with the number of contexts supported.

Having more register files (each with the same number of read/write ports as the original single threaded organization) significantly increases the routing logic complexity, which grows super-linearly with the number of contexts. The crossbars connecting the registers to the functional units also grow substantially in size and consequently in delay. They require the introduction of multiple level switches and more cycles for register transfers.

The fetch unit must be able to get instructions from all of the simultaneous threads, from one at a time. To support this, the program counter must also be replicated for each of the contexts. Apart from this, there is no significant complexity increase for the fetch unit.

Concluding, the vast majority of the complexity increase is generated by the need to support data transfers between more pairs of source/destinations by the addition of more registers. This complexity increase is taken over only by the crossbars, the other components of the processor are hardly affected.

Multithreading significantly increases the die area. The vector register file already represents a large portion of the die size. By replicating them several times and by adding the required routing logic, the die size can easily double.

How many simultaneous contexts should be supported for maximum efficiency? The simulations have shown that the increase in performance is sub-linear with respect to the number of contexts. This is also supported by [4], a case study on scalar processors which shows that more than four threads is are not practical. Since vector register files are much larger than in the scalar case, the practical number of contexts that can efficiently be used is reduced even more down to two or three. Scalar processors can accommodate multithreading easier because of the much smaller increase in area determined by replicating register files.

Better increases in performance can be obtained from combining multithreading with dynamic scheduling techniques. In particular multithreading is very suited for out-of-order execution. Combining these two techniques has several advantages since they have some things in common. The out-of-order engine remains unchanged, a multithreaded fetch unit would be the main addition. This improved fetch unit simultaneously gets instructions from all the threads and gives them to the reorder buffer. The reorder buffer is augmented by storing for each instruction the thread it comes from and so, renaming provides a very convenient way to implicitly distinguish instructions from different threads. From this point on, all instructions are treated in the same manner, and thus, with little increase in control complexity, instructions from different threads can be active at the same time in the processor.

3.3 Summary

We have started the chapter by identifying and studying the memory access types. Statistics from benchmarks have been quoted that show the frequency of each of these access types.

We have seen that unit stride accesses are the most common and the most easy to implement memory accesses since memory is accesses at consecutive locations. In contrast, the indexed memory accesses are both rarely used and very costly to support since they group many independent memory access into a single instruction that accesses random memory locations. Nonetheless, vector supercomputers are built with the goal of utmost performance, and thus provide full support for indexed accesses without any degradation in performance when compared to the unit stride accesses. A typical memory system organization has been presented. We have seen that the main memory is organized in many independent banks and that it represents a high fraction of the total cost of a vector processor.

Vector processors have more memory units. The appropriate number of those units is discussed taking into account the number of functional units and the frequency of appearance of memory access instructions in vectorizable applications. Since there are multiple agents accessing the memory, there must be provided a mechanism for keeping the memory consistent. We have seen that the consistency model is quite loose, reducing the hardware complexity passing the disambiguation of memory accesses from hardware to the compiler.

Then, we have presented a more recent approach to the vector memory system, the addition of vector caches. This change in design is motivated by the technological improvements that have allowed efficient production of sufficiently large vector data caches. The use of caches must be supported at the algorithmic level favoring codes that split the work in small sub-problems that can fit in the cache. We have also seen that the emphasis in vector caches is on temporal locality, by nature, most vector memory instructions access memory at consecutive locations.

The second half of the chapter was dedicated to the control part of the vector unit. We have analyzed methods of keeping the vector functional units busy at all times. First we have shown that a single vector instruction issued per cycle is more than enough even for vector processors with many functional units. This means that that the front end of the vector processor is not a bottleneck for performance.

Vector processors inherently exploit data level parallelism. We have also explored other levels of parallelism. We have looked at instruction level parallelism that can be exploited through dynamic reordering of instructions. The most basic form of dynamic instruction reordering is the decoupled organization. The instruction stream is split in two separate streams: computation instructions and memory access instructions. These two streams of instructions execute out-of-order with respect to each other but, each stream is executed in-order.

The next step for increasing performance is the complete out-of-order execution of all instructions. We have seen that register renaming can be easily adapted to the vector model. We have studied two release strategies for the vector registers, an aggressive one that brings very high performance and a more conservatory one that has the great advantage of providing a straight forward method of implementing precise exceptions, generaly very difficult to provide for vector processors.

We have taken another step further by exploiting thread level paralelism through the multithreaded vector organization. We have seen that support for multiple contexts requires the replication of the vector register file which incur a substantial increase in chip area. We have also seen that the interconnection network between the register files and the functional units becomes a neuralgic point. There are three ways of combining instructions from the various threads executing in parallel.

The best performance is offered by a microarchitecture that uses all level of parallelism, more precisely an out-of-order multithreaded vector processor. Register renaming provides a straight forward mechanism that combines the registers from multiple contexts in a single physical register file. Using this scheme, only the front end of the vector machine needs to be changed.

Experimental Results 4

This chapter presents the experiments we have conducted. We present the simulation environment, the analyzed applications and the simulation results.

We start with the process of vectorization. We formally define it and show in detail how a scalar loop is transformed into vector instructions. We then have an example to illustrate how coding decisions can affect vector performance.

The second section presents the simulation environment. We have started from the sim-outorder simulator of the SimpleScalar 3.0 suite. This is an out-of-order, cycle accurate superscalar processor simulator. We have added vector instructions, vector register files, vector functional and memory units. We have also added statistics to follow the execution of the vector unit. We have vector instruction related statistics and vector functional unit related statistics. The section concludes by presenting the experiment flow and the simulation steps.

We analyze media and signal processing software for their vectorization potentials. We start by profiling the application in search of computationally intensive kernels. We inspect their code and if it is vectorizable we proceed through hand vectorization. If the kernels cannot be vectorized we record the reasons that prevent vectorization and take them into account when formulating our coding style suggestions.

We then perform multiple simulations by varying vector unit parameters like the number of vector registers, their size, the number of vector lanes, the memory latency, the memory bandwidth. These simulations are performed on both the individual kernels and the entire application.

4.1 Introduction to Vectorization

The vector unit is a domain specific accelerator. The original code, entirely consisting of scalar instructions, running on a scalar processor is partitioned into scalar code and vector code running on the scalar unit and vector unit respectively. We base our discussion on the fact that vector code running on the vector unit executes significantly faster than equivalent scalar code executing on a scalar processor. Therefore, we want to have as much vector code as possible.

This chapter answers the following questions:

- what kind of code can be transferred to the vector unit
- under which conditions?

We start by describing the process of vectorization, how it is performed and we provide an example and tips for coding. Further in this section we discuss some application requirements for efficient vectorization.

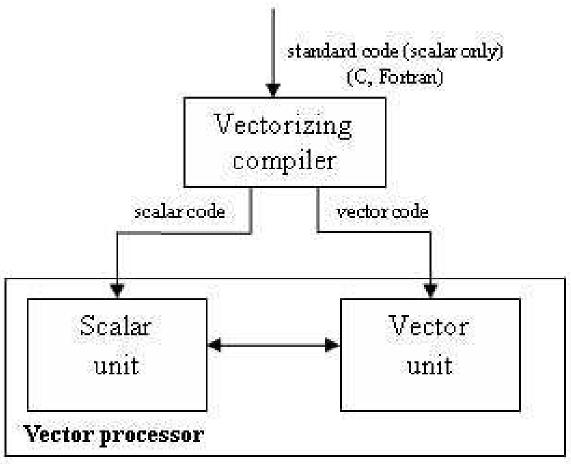


Figure 4.1: The process of vectorization

4.1.1 The process of Vectorization

Vectorization means compacting many iterations of a scalar loop into a single iteration of a vector loop. Ideally an entire scalar loop could be converted into just a few vector instructions. The vectorizing compiler transforms a stream of scalar instructions into two separate stream of instructions that are each executed by different units of the vector processor (see Figure 4.1):

- the scalar instruction stream executed on the scalar unit
- the vector instruction stream executed on the vector unit

Example: We want to add two arrays B and C to produce array A. All three arrays are of length n.

```
for i = 1 to n do
   A[i] = B[i] + C[i]
```

Through strip-mining we can convert this loop to ($m < n$):

```
for i = 1, i < n, i = i + m
   for j = i to i + m - 1 do
      A[j] = B[j] + C[j]
```

Now, the entire inner loop can be written as a single vector instruction that adds m elements of arrays B and C and produces m elements of array C. Thus, the n iterations of the original scalar loop are converted into n/m iterations of vector instructions. The value m is machine dependent and is the length of the vector registers. The value n is application specific and is the length of the scalar loop.

The following advantages are easy to observe.

- The number of instructions is reduced by a factor of n/m thus reducing the instruction fetch bandwidth required and also the instruction issue bandwidth.
- The number of branches is reduced by the same factor, reducing the probability of miss speculation and branch miss prediction.
- The number of address calculations are reduced again by the same factor
- The operations packed together in a vector instruction are known to be independent of each other, so no runtime dependency checks are required further reducing the complexity of the possibly out of order issue logic.

A vector processor must be able to run any software that runs on a scalar processor. Therefore, the input of vectorization is standard scalar code, written in any high level language, like C or Fortran. The vectorizing compiler looks for scalar loops that can be converted into vector instructions. In order to do that, it must make sure that there are no data dependencies whatsoever between the instructions of the original scalar loop. The overall speedup of the vectorized applications is dependent on the ability of the compiler to vectorize the input code. In the following pages we shall look at how the choice of algorithms and coding can impact this process.

We have multiple levels we can look at in order to assert code vectorization:

- the theoretical level of the problem itself
- the algorithm used to solve the problem
- the coding

To illustrate the process of vectorization we shall take a close look at possible implementation of a digital linear filter over an uncompressed image. We will examine several design decision when coding this application and discuss the implications of possible choices. Later in the chapter we shall also present the simulation results for two possible implementation.

For our filter, we are considering a window of 9 pixels. Each pixel in the new image is dependent only on several adjacent pixels of the old image, so there is no data dependency between the pixels of the new image. For all pixels, the computations are exactly the same. This is a suitable application for a vector processor, which potentially gains significant speedups through vectorization.

Considering a scalar approach, the values of the new pixels are computed one at a time. This would mean multiplying the values of the adjacent pixels with the corresponding coefficients and then summing up. The program would look like this:

```
for i = 1 to height do
   for j = i to width do
      loop over adjacent pixels
           multiply with coefficient
      new_val[i][j] = sumup
```

When multiple nested loops are involved, only the innermost loop can be vectorized. In our case, when the window size of the filtering is 9 pixels, it would mean a maximum vector length of 9. In other words, only 9 operations can be packed in a vector instruction. This number is way too small, we can do significantly better.

A simple way to improve the speedup gained through vectorization is to reorder the loops:

```
for i = 1 to height do
   loop over the coeficients
        loop over the pixels of one line and multiply by the
            current coeficient
        add the new values to the previously computed ones
```

This time, the inner loop has the vector length equal to the image width. This is significantly better since the size of the picture usually is much larger than the filter window size.

Loop interchange is a common compiler optimization. However the compiler might detect this improvement or might not. It all depends on the compiler implementation. And even a very smart compiler cannot always reorganize the code in the optimal way. The programmer can always optimize better and he should help the compiler by writing the code in a clearly vectorizable way.

In the following, we shall look at possible choices for the data structures. The color of each pixel is stored as three separate values for red, green and blue. We can either have three different matrixes or a single matrix of structs.

```
int red[][]; int green[][]; int blue[][];
```

or

```
struct pixel
{
    int red, green, blue;
};
struct pixel picture[][];
```

For a scalar processor, the two options would make little difference, the cache hit rate will vary slightly, depending on the implementation. For a vector processor, however, the differences might be substantial.

Using the presented algorithm, we filter one color at a time, ex: first red then green and so on. When we use dedicated matrixes for each of the colors all memory accesses are at consecutive addresses. In contrast, for a matrix of structs, all memory accesses have a stride of 3. As discussed in the chapter dedicated to the vector memory system, strided accesses require more complicated hardware and, depending on the implementation, might offer smaller bandwidth than unit stride accesses thus reducing overall execution time.

From this simple example we can see that in order to fully benefit from the potential speedup offered by vectorization, the software should be tailored in a certain way. We can also note that by coding 'the vector way', we usually do not reduce scalar performance but, by writing code 'the scalar way', vector performance might be drastically reduced.

The following section presents our simulation environment while the last section presents the simulation results.

4.2 Simulation Environment

We have started by modifying the out of order processor model of the SimpleScalar 3.0 simulator suite. Sim-outorder is a cycle accurate simulator that models an out of order superscalar processor. It is written in such a way that it can be easily extended with new functionality. This proved to be the ideal platform to build our vector processor simulator.

For each instruction it can be specified the functional unit it works on and the registers it works with. For the functional units the latency can be varied. New instructions, registers and functional units can be added easily. New instructions are actually annotated existing instructions. This allows addition of new instructions without the need to modify the compiler and assembler.

To the original superscalar processor simulated by sim-outorder, we have added a vector register file with the corresponding bit vectors, vector functional units, memory units and vector instructions.

Since we have started from an out-of-order simulator, our simulator is also capable of executing the vector instructions out-of-order, which is quite advanced because most of the commercial implementations of vector processors are capable only of in-order execution. This should give us an edge.

This section is organized as follows: first we describe the additions to the SimpleScalar simulator: the vector instructions, the vector register files, the vector functional units and the simulator statistics added. The last pages of this section are dedicated to describing the way experiments are conducted, the entire flow: all tools used, order in which they are used, inputs and outputs of each.

4.2.1 Vector Instructions

We have extended the simulator instruction set with vector instructions. These instructions are standard vector instructions, very similar to those described in the chapter dedicated to the basic vector architecture.

We have included support for masked operation. This will be presented in detail in the next section, dedicated to the vector register files.

Our extension to the simulator includes support for the following data type transfers:

- Floating point
    - Double precision
- Integer arithmetic
    - 1 byte signed or unsigned

  – 2 bytes signed or unsigned
  – 4 bytes signed or unsigned

4.2.1.1 Auto-sectioning

An important feature of the vector instruction set is the support for auto-sectioning. This allows the software to be totally independent of the section size which is a parameter in our simulator.

The auto-sectioning mechanism has been conceived to simplify the addition of inline vector assembly into C code. This approach is slightly different than the one described in the chapter dedicated to vector register files. The traditional approach is best suited for code written entirely in assembly language or for code generated by a compiler. There is no vectorizing compiler available so, for our experiments, we have vectorized by hand existing applications replacing loops by inline assembly into the existing C code. This motivates the slight deviation from the traditional approach.

Three instructions have been added to implement auto-sectioning.

```
v.setconf       vl, index
v.updateindex   index
v.update        vl
```

Let us take an example. This loop:

```
for i = 1 to n do
    perform some computation on a[i]
```

would look like this (slightly simplified):

```
int index = 1
int vl = n

while (vl  > 0)
{
    __asm__(
        "v.ld           $1,  %2"
        computation here
        "v.st           %2,  $1"
        "v.updateindex  %0"
        "v.updatevl     %1"
        :
        "=r"    (index),
        "=r"    (vl)
        :
        "r" (&a[index])
}
```

The *__asm__* function does the following replacements:

- %2 is replaced by the register number where the address of a[index] is located. Of course it also makes sure that that address is first loaded into a register.
- %1 is replaced by the register that contains the value of variable index
- %0 is replaced by the register that contains the value of variable vl

The parameters preceded by "=r" represent registers that written (output parameters), while parameters preceded by "r" represent registers that are only read.

The instruction "v.setconf vl, index" initializes the vector length register inside the CPU to the minimum value between the Section Size and the input parameter vl (passed as a register number).

The instruction "v.updatevl vl" also updates the vector length register internal to the CPU but also updates the register passed as parameter with the following value:

- If vl $\leq$ Section Size then vl = 0
- If vl > Section Size then vl = vl - Section Size

This instruction is particularly useful for the control of the while loop that contains the vector code: while (vl > 0).

The instruction "v.updateindex index" updates the value of the register index by adding an amount equal to the Section Size: index = index + Section Size. This is particularly useful when passing the address of the first element to be read by the vector load instruction.

4.2.2 Vector Register Files

We have added two vector register files, one for integers, each element being 32 bits, the other one for floating point operations, an element being 64 bits wide.

The vector length and number of registers are parameters in our simulator. In the experiments we have varied the vector length to range between 1 and 1024 elements, always being a power of two.

Each vector register has an associated bit vector. The vector unit has a configuration bit that indicates whether masked operation is turned on or off. If the masked operation is enabled, the results are written back to a register only for those elements, which have a corresponding one in the vector register's bit vector.

For instance, this piece of code:

```
for i = 1 to n do
    if a[i] > 0 then
        b[i] = a[i] + b[i]
```

is translated into:

```
1.  v.ld          vr1,   a
2.  v.ld          vr2,   b
3.  v.compare.gt  vr2,  vr1,  vs0
```

```
4.  v.maskson
5.  v.add           vr2,  vr2,  vr1
6.  v.masksoff
```

The third instruction does the following: the bit vector associated with the vector register vr2 is set only for those elements where the contents of vector register vr1 is greater than the scalar register vs0 (equal to zero) and cleared otherwise. The fourth instruction turns the mask operation on while the fifth instruction writes back results only for those elements where the bit vector was set by the previous instruction. Masked operation by default is disabled.

It would be desirable to have bit vectors that are not bound to vector registers, so that, each operation can be predicated by any bit vector. The binding of bit vectors to vector registers comes from a limitation of SimpleScalar. Internally, each instruction may have up to 3 input dependencies and 2 output dependencies. However, the PISA instruction set, used by the simulator, only allows an instruction to have 2 input operands and a single output operand. This has forced us to tie each bit vector to a corresponding vector register.

The question of having two separate vector register files, one for integer and one for floating point or a single register file that can contain both integer and floating point data remains open. This will be answered in future work.

4.2.3 Vector Functional Units

In the SimpleScalar implementation, each functional unit can perform a class of operations. For instance, the ALU can perform additions, subtractions and logical operations. One ALU at any given cycle can perform only one of these operations. On the other hand, the simulator can support any number of functional units, of any type i.e multiple operations per cycle.

The functionality of the instruction is built in the instruction and not in the functional unit it uses. Thus, for each operation that a functional unit can perform there are only two simulation parameters:

- The execution latency, expressed in cycles.
- The number of cycles that must pass until a new operation can be started. This is the degree of pipelining, A value of one corresponds to a fully pipelined unit.

Instructions specify the operation they require and not functional units. The issue logic, at issue time for each ready instruction, finds a functional unit that is capable of executing that particular instructions.

Adhering to these conventions, we have added four types of functional units: two types of computational units and two types of memory access units. The vector ALU can perform additions and subtractions in two flavors: between two vector registers or between a vector register and a scalar register. In the latter case, each element of the vector register is added/subtracted to/from the scalar element.

Also, the vector ALU assigns bit vector values as results of comparisons. These comparisons can also be between two vector registers or between the elements of a vector register and a scalar register. The vector multiplication unit can perform multiplications and divisions, both between two vector registers or between a vector register and a scalar register. These functional units have the same execution latency for all of the operations they can execute and is of the following form:

$$Execution\ latency = Startup\ latency + \frac{SectionSize}{Number\ of\ lanes} \tag{4.1}$$

$$Pipelining\ degree = \frac{SectionSize}{Number\ of\ lanes} \tag{4.2}$$

All of these values are simulator parameters:

- Startup latency depends on the operation and in our experiments we have decided to use 2 cycles for addition and 4 cycles for multiplication.
- Section size is the number of elements each vector register has
- The number of lanes is the number of physical units inside a functional unit that work in parallel to perform the operations on the elements of a vector register. For instance in a two lane implementation there are two physical functional units, one performing operations on the first half of the vector register, the other one on the second half of the vector register. For more information refer to the chapter that presents the vector functional units.

A vector instruction is equivalent to a series of back to back operation of the same type. If the functional unit is fully pipelined, then the total execution latency for a vector instruction is the sum of the startup latency and of the number of elements in the vector register (Section Size), thus a new instruction can be issued to a vector functional unit Section Size cycles after a previous instruction started execution.

The vector memory units have very similar parameters. For the load we have:

$$Load\ latency = Memory\ latency + \frac{SectionSize}{Words\ per\ cycle} \tag{4.3}$$

These parameters are:

- Memory latency is the number of cycles it takes for the first word to come from memory after a read has been requested
- Section Size is the number of elements contained in a vector register
- Words per cycle is the number of elements that can be transferred each cycle, it is the memory bandwidth expressed in words.

The vector memory accesses are bypassing the cache, accessing main memory directly. The timing of these accesses (the vector memory latency) is modeled into the functional unit latency. This has reduced the complexity of our simulator eliminating the need to model a vector memory system.

For the store unit the memory latency is not visible from the view of the processor. Thus, a store instruction completes in at most section size cycles after it starts.

$$Store\ latency = \frac{SectionSize}{Words\ per\ cycle} \tag{4.4}$$

For both load and stores the pipelining degree is the same:

$$Pipelining\ degree = \frac{SectionSize}{Number\ of\ lanes} \tag{4.5}$$

4.2.4 Scalar and Vector Memory

4.2.4.1 Scalar Memory

For the scalar memory we have used a traditional memory hierarchy with two levels of cache. The first level has separate instruction and data caches while the second level cache is joint for both instructions and data.

The level one instruction cache has the following parameters:

- 512 sets
- the block size is 32bytes, that means 8 instructions
- associativity is one, thus a direct mapped cache
- replacement policy is Least Recently Used
- total size of L1I cache is 16KBytes
- hit latency is one cycle

The level one data cache has the following parameters

- 128 sets
- the block size is 32bytes, that means 8 32bit words
- associativity is four
- replacement policy is Least Recently Used
- total size of L1D cache is 16KBytes, same as for the level one instruction cache
- hit latency is one cycle

The joint second level cache is

- 1024 sets
- the block size is 64bytes, that means 16 instructions or 16 32bit words
- associativity level is four

- replacement policy is Least Recently Used
- total size of the level 2 cache is 256KB
- hit latency is 6 cycles

We have simulated a pipelined main memory with the scalar bus 8 bytes wide. The initial latency for every access is 18 cycles and for burst accesses, eight bytes (the scalar bus width) are available every other cycle.

4.2.4.2 Vector Memory

The vector memory accesses go to the same main memory which is shared between the vector and scalar unit. However, the vector memory bus is wider than the scalar one, allowing up to 256 bytes transferred per cycle. This value corresponds to four double precision floating point numbers per cycle. This allows the vector datapath to be organized in up to four lanes working on double precision floating point numbers and have the memory bandwidth equal to the data processing rate. However, both the number of lanes and the vector memory bus width are simulation parameters.

Also, the vector memory latency is a parameter in our simulations and is varied from 0 (perfect memory) upto extreme cases where the vector memory latency is significantly higher than the scalar latency, even up to 100 cycles. This might be the case of cheap memory systems that sacrifice latency for increased bandwidth. However, the most relevant case can be considered the case when the vector memory latency is equal to the scalar latency and having roughly 20 cycles.

4.2.5 Simulator Statistics

We have implemented several statistics in the simulator to better evaluate performance. There are two major types of statistics that will be described in the following subsections:

- Instruction related statistics
- Functional unit related statistics

These statistics are adopted from Espasa's work [8].

4.2.5.1 Instruction statistics

The instructions statistics count down the number of vector instructions that are issued. We have statistics for:

- The total number of vector instructions issued
- The percentage of the total amount of instructions occupied by the vector instructions
- The total number of vector computation instruction
  - The number of instructions that use the vector ALU

  - The number of instructions that use the vector multiply unit

- The total number of vector memory instructions
  - The number of vector stores
  - The number of vector loads
- The number of vector control instructions (like those used for turning mask operation on or off, auto-sectioning instructions)

4.2.5.2 Functional unit statistics

These statistics track the usage of the vector functional units on a cycle basis. We count the number of cycles each of the functional unit is busy and we come up with two numbers for each:

- A total number of cycles that each functional unit was busy
- An occupancy rate meaning the percent of all execution cycles that each functional unit was busy

Another class of statistics is the history of occupancy rate for each of the functional units. For instance we can calculate the occupancy rate of a functional unit over each thousand cycles.

Using the history of the occupancy rates for any of the functional units or memory units we can visually follow the evolution of the simulation. Also, these statistics can be a very valuable tool for debugging both the simulator and the process of vectorizing the applications.

4.2.6 Experiment Flow

SimpleScalar supports multiple instruction set architectures. We have used the default one: PISA (Portable ISA), a MIPS based instruction set. The simulator also comes with a compiler, a linker and a loader for this particular architecture. The compiler is gcc-2.7.2.3 with a modified back-end to generate code for PISA. This is quite an old version of gcc and several features of the newer versions are not supported but this was not a major problem in our experiments.

For every selected application, the flow goes like this:

1. Compile the application for the x86 architecture
2. Profile the application in search of computation intensive kernels
3. Inspect the kernels for their vectorization potential
4. Compile the application for the PISA architecture, the architecture used by our simulator
5. Run the application on the simulator to assess the initial performance, before vectorization

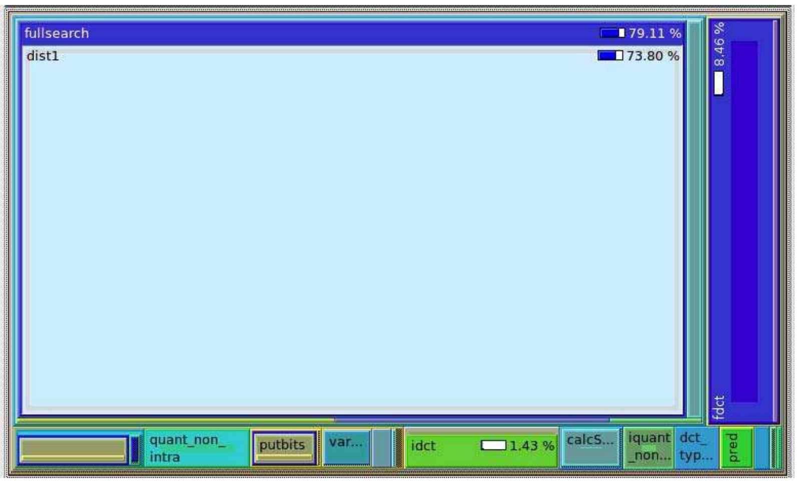


Figure 4.2: Profile of the MPEG2 Decoder using kcachegrind

6. Vectorize by hand the computation intensive kernels identified at the third step
7. Simulate the vectorized applications by varying parameters
8. Analyze results

For profiling the selected applications we have used the Linuz application *callgrind*. The output can be graphically visualized using *kcachegrind*, see Figure 4.2.

There have been situations when the kernels identified at the second step for various reasons could not be vectorized. In these cases we had to restrict the analysis to the software side only, by carefully looking at the reasons that prevented vectorization. This was the case for some of the bio informatics applications that we have looked at. More details on this in the following section, dedicated to the simulation results.

The most important step was the code vectorization by transforming, by hand, scalar loops into vector loops with significantly less iterations. Undoubtedly a vectorizing compiler would do a significantly better job but there is none we can use. Due to time constraints we are forced to search only the kernels that occupy most of the application's execution time and vectorize them manually. Therefore, when analyzing the results we must take into account that with the proper tools (such as a vectorizing compiler and corresponding libraries) the gains from vectorization would be higher, maybe significantly higher.

4.2.6.1 Simulation Flow

An important tool for our experiments was SSIAT (SimpleScalar Instruction Architecture Tool). This is an application that modifies the source code of SimpleScalar by adding new instructions, functional units and register files to the base architecture. SSIAT has been developed at TUDelt, Computer Engineering lab. More information about it can be found here [?]. SSIAT has a configuration file that contains all of the parameters of the new instructions, functional units and register files (as presented earlier). This file contains most of the architectural configuration data needed by our simulator.

The micro-architectural details of the simulation parameters are passed to the SimpleScalar by directly editing the simulator's source code through a bash script.

Each simulation iteration consists of the following phases:

- Run SSIAT to modify architectural parameters of SimpleScalar
- Update the micro-architectural parameters
- Recompile the simulator
- Run the application through the simulator
- Log results

4.3 Simulation Results

In the first part of the thesis, we have described the vector architecture and microarchitecture. In the previous section we have described a simulator that simulates the general features of a vector processor. In this section, based on our vector processor simulator, we evaluate how existing software behaves on vector processors.

Following this goal, we have selected several applications from different fields and we have analyzed the possible degree of vectorization and the speedups obtained. We are looking for limitations and ways to improve or totally remove these limitations.

We have looked at applications from the following domains:

- Video processing
- Image processing
- Sound processing

All of the above are computationally intensive applications with huge amount of data parallelism. These attributes make them potential candidates for vectorization and execution on vector processors. Our initial expectations are that these applications are easily adaptable to the vector paradigm and that they will provide good speedups.

We shall also test the coding component. We have chosen publicly available implementations of these algorithms. The questions we address are:

- These implementations, written with a scalar processor in mind, when vectorized gain satisfactory speedups?
- Is it necessary to rewrite the algorithms, this time with the vector architecture in mind?

We shall investigate the performance gains with respect to the section size of the vector machine and to the problem size.We shall also track the following program attributes:

- Data structures
- Data types
- Array lengths and the average number of iterations the loops have
- The use of pointers
- Coding requirements:
  - to be vectorized, loops must be regular and clearly structured
  - unrolled loops must be re-rolled before vectorization
  - when multiple nested loops are involved, only the innermost loop can be vectorized and this last loop cannot have any function calls

All of the considered applications are written in plain C. We did not consider C++ or any object oriented programming language because it would unnecessarily complicate the vectorization process.

The kernels we have analyzed have proven to have a very simple structure and they can be efficiently executed with a single ALU and one vector multiplication unit. Also, our preliminary tests and code inspection have indicated that the number of vector load instructions is on average twice the number of vector store instructions. Based on these initial results we have decided to run the majority of our simulations using a vector unit with the following configuration:

- one vector ALU unit
- one vector Multiplication unit
- two vector Load units
- one vector Store unit

Having fixed the number of functional units and memory units leaves us with a design space with five dimensions:

- the number of vector registers
- the size of the vector registers
- the number of lanes in the datapath
- the memory latency
- the memory bandwidth

Looking again at the vectorized versions of the applications we have analyzed, we have determined that six vector registers are enough to efficiently execute them. Thus, we have chosen the closest power of two that meets our requirements and fixed the number of registers to eight. Several compiler optimizations like loop unrolling or loop fusion would benefit from more vector registers but these optimizations are beyond the scope of this thesis and have not been explored.

Theoretical analysis indicates that lowest complexity and best performance are obtained when the memory can sustain data transfers roughly at the same rate as functional units process data from registers. In other words, neglecting the instruction startup latencies, a vector memory access instruction should take the same time to execute as a vector computation instruction. When using a balanced number of functional and memory units, this enables a continuous flow of computation without having to wait for the memory and a continuous flow of memory transfers without having to wait for computations to execute. Having both computation and memory accesses working at the same speed also greatly simplifies the synchronization between these two. For example, imagine a load chained to a computation instruction where the datapath is organized in four lanes and the memory unit can transfer only one word per cycle. This would mean that the functional unit could perform computation only one out of every four cycles greatly reducing performance and requiring additional synchronization circuitry.

Following this logic we have chosen to simulate an organization that has the memory bandwidth (in number of words transferred per cycle) equal to the number of lanes in the datapath. We have decided to perform our experiments using four lanes and a memory bandwidth of four words transferred per cycle.

Using these conventions we have reduced the design space to only two dimensions: the section size (length of a vector register) and the vector memory latency. Thus, for our simulations we vary the following parameters, for each of the vectorized applications:

- section size: from 1 to 256 or 1024, powers of 2
- vector memory latency: from 0 cycles to 100 cycles

4.3.1 MPEG2 Encoder Kernels

We have started by looking at an MPEG2 encoder and decoder. These two applications are very similar in nature and several issues will be discussed for both applications in parallel.

For both applications, most of the computation is performed on data stored in integer format, one bit wide. The data is organized in arrays of bytes. The main kernels are written in a vectorization friendly way and can be transformed in vector operations quite easily.

It is well known that pointers, if used improperly, can make vectorization extremely hard. In this case, the programs are sufficiently well written so that pointers do not create difficulties in vectorization.

We shall start by showing the profiling information. Then we shall identify the kernels suitable for vectorization and discuss them. The kernels are quite similar in nature thus, we shall present in detail the first one, for the others, to avoid repetitions, we shall only point out the important particularities and their consequences. As a final step we analyze the speedup of the entire application, consisting of both the vectorized kernels and the non vectorized parts.

Profiling the MPEG2 encoder gives the following information (Table 4.1):

We can observe that the functions dist1, fdct and quant_non_intra occupy more than 85% of the execution time. In the following pages we shall have a closer look at these functions.

4.3.1.1 Function dist1

Function dist1 consists of four groups of nested loops that are very similar. Depending on the input parameters, only one of the four loops is executed. We have chosen to present only one of them:

```
for (j=0; j<h; j++)
{
    for (i=0; i<16; i++)
    {
      v = ((unsigned int)(p1[i]+p1[i+1]+1)>>1) - p2[i];
      if (v>=0)
        s+= v;
      else
        s-= v;
    }
  p1+= lx;
  p2+= lx;
}
```

The inner loop uses two consecutive elements of the array p1. In the scalar case this has good performance because data required by one iteration is already loaded in a register or at least in the cache because it was required by the previous iteration. For vector execution we need to load p1[i] and p1[i+1] in two separate registers in order to add the elements in one instruction. In other words, we have to load from memory the same array twice: in the first vector register we load *vl* elements starting from the address of p1[0] and in another vector register *vl* elements starting from the address of p1[1]. The loss in performance caused by this double loading of data is compensated by the masked operation that is very efficiently executed in vector hardware.

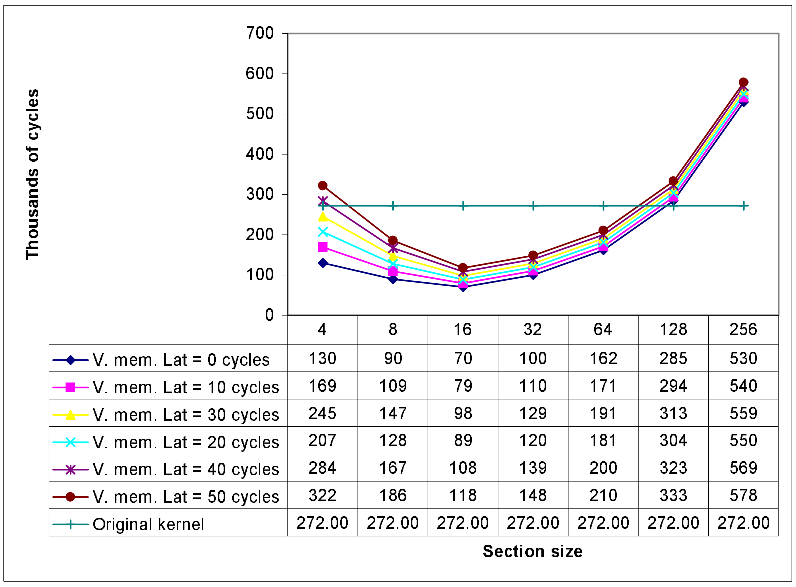
The inner loop through vectorization is translate into:

```
s_tmp = 0;
```

Self	Called	Name
73.81%	4,518,806	dist1
7.84%	31,680	fdct
7.32%	38,720	fullsearch
4.8%	19,320	quant_non_intra
1.82%	9,504	bdist1
1.43%	31,680	idct

Table 4.1: Profiling information for the MPEG2 encoder

```
        vl = 16;
        ld_index = 0;

        __asm__("v.setvconf %0, %1, %2" : : "r" (vl), "r" (0), "r" (0));

        while (vl)
        {
/*Load the data*/
            __asm__ __volatile__ (
            "v.ldstep       $1, %0, 1"
            "v.ldstep       $2, %1, 1"
            "v.ldstep       $3, %2, 1"
            :
            :
            "r" (&p1[ld_index]),
            "r" (&p1[ld_index + 1]),
            "r" (&p2[ld_index])
                                );
/*Perform the computation*/
            __asm__ __volatile__ (
            "v.add      $4, $1, $2"
            "v.sadd     $4, %0, $4"
            "v.shr      $4, $4, 1"
            "v.sub      $5, $4, $3"
            "v.scompare.lt  $5, $5, $0"
            "v.maskson"
            "v.ssub     $5, $0, $5"
            "v.masksoff"
            :
            :
            "r" (1)
                                );
/*write back results and update pointers*/
            __asm__ __volatile__ (
            "v.sumup        %0, $5"
            "v.updateindex  %1"
            "v.updatevl     %2"
            :
            "=r" (s_tmp),
            "=r" (ld_index),
            "=r" (vl)
            :
                                );
        }
        s += s_tmp;
```



	4	8	16	32	64	128	256
◆ V. mem. Lat = 0 cycles	130	90	70	100	162	285	530
■ V. mem. Lat = 10 cycles	169	109	79	110	171	294	540
▲ V. mem. Lat = 30 cycles	245	147	98	129	191	313	559
× V. mem. Lat = 20 cycles	207	128	89	120	181	304	550
✱ V. mem. Lat = 40 cycles	284	167	108	139	200	323	569
● V. mem. Lat = 50 cycles	322	186	118	148	210	333	578
+ Original kernel	272.00	272.00	272.00	272.00	272.00	272.00	272.00

Figure 4.3: MPEG2 Encoder, kernel dist1: the execution time as a function of the section size and memory latency

The clear disadvantage of this loop is its iteration count. The maximum vector length is 16. Thus, when increasing the section size passed 16 elements, this brings no improvements because any vector instruction will be able to process only 16 useful elements. Since the execution time of a vector instruction is (neglecting startup latency) linearly dependent on the section size, by increasing the section size, we actually increase the execution time by processing unneeded elements. Figure 4.3 shows the execution time of the dist1 kernel as a function of the section size. The figure consists of several curved lines that plot how the execution time changes for different memory latencies.

This limitation can be overcome in hardware by execution only the first vector length operations on a vector register's elements. In other words, when the vector register is not completely filled with useful elements, perform operations only on the useful elements and completely discard the rest. The useful elements are always at the beginning of the vector register and the number of these elements is contained in the status register: vector length. This way there would be no penalty when executing vector computation on very short vectors. Unfortunately, the host SimpleScalar simulator does not allow us to model such a behavior. The latencies of the functional units are fixed and cannot be changed at runtime.

Another important conclusion from Figure 4.3 is related to the memory latency. Consider vectorizing a loop with n iterations. The longer the section size, the fewer vector instructions we execute. On the other hand, each instruction processes more elements, therefore takes longer to complete. It might seem that these two statements cancel each other, and that the section size makes no difference. A closer look, however, indicates that the number of instructions is reduced by a factor of

$$\frac{n}{Section\ Size}$$

Each instruction takes

$$Const1 + \frac{Section\ Size}{Const2}$$

where, in the case of a computation instruction, we have:

$Const1$ functional unit startup latency, is around 2-4 cycles. For large section sizes, this term can almost be neglected

$Const2$ number of lanes of the vector unit. We have selected to simulate a configuration with four lanes.

In the case of memory instructions we have:

$Const1$ is:

- For loads is the memory latency, a parameter in our simulations that varies from 0 cycles (perfect memory system) to 100 cycles (slow memory system)
- For store is always 0 because from the point of view of the processor, for stores the memory latency is not visible for the processor

$Const2$ is the memory bandwidth expressed in words transferred per cycle. We have decided to simulate a configuration that goes accordingly to the functional unit execution time, thus our memory system provides four words per cycle

By reducing the number of batches of vector instructions we reduce the amount of times that startup latency of the instructions is encountered and so, we reduce the total execution time.

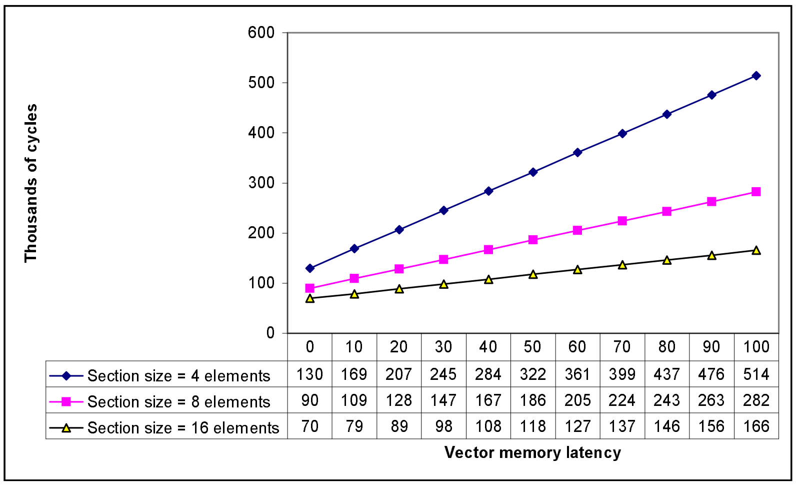
For real implementations, the functional unit startup time is significantly smaller than the memory latency, thus we can almost neglect it in our discussion. To support this we look at the performance curve for the perfect memory system (the bottom one in Figure 4.3). When increasing the section size, the performance increases only by several percents due to fewer functional unit startup latencies. On the other hand, for slow memory the execution time of the application for a small section size is significantly longer than for the same application, with the same long memory latency but with a longer section size.

Summarizing, from Figure 4.3 we can see that memory latency can significantly degrade vector performance especially for short section sizes. As the section size grows the impact of memory latency is becoming less of an issue. This explains why vector processors typically have good performance for long vectors and reduced performance for short vectors.

If we change the perspective on Figure 4.3 we obtain Figure 4.4 where on the X axis is the memory latency and the series are for different section sizes. The latter Figure indicates that for any given section size the execution time is linearly dependent on the memory latency. We can explain this by considering the following equations:

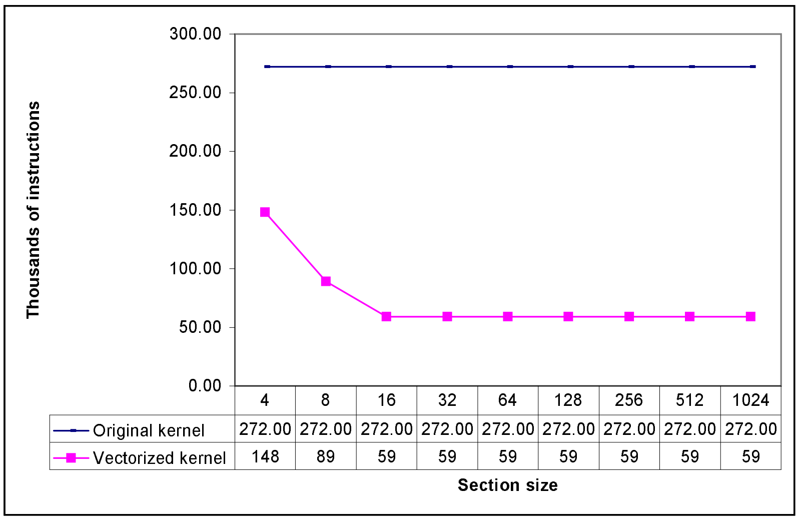
$$Total\ no.\ of\ cycles = Scalar\ cycles + Vector\ cycles \tag{4.6}$$

$$Vector\ cycles = No.\ of\ loads \cdot (Mem.\ latency + \frac{Section\ Size}{Words\ per\ cycle}) +$$



Vector memory latency	0	10	20	30	40	50	60	70	80	90	100
◆ Section size = 4 elements	130	169	207	245	284	322	361	399	437	476	514
■ Section size = 8 elements	90	109	128	147	167	186	205	224	243	263	282
▲ Section size = 16 elements	70	79	89	98	108	118	127	137	146	156	166

Figure 4.4: MPEG2 Encoder, kernel dist1: The total execution time as a function of the memory latency and the section size



Section size	4	8	16	32	64	128	256	512	1024
▬ Original kernel	272.00	272.00	272.00	272.00	272.00	272.00	272.00	272.00	272.00
■ Vectorized kernel	148	89	59	59	59	59	59	59	59

Figure 4.5: MPEG2 Encoder, kernel dist1: The total number of instructions executed as a function of the section size

$$+No.\ of\ stores \cdot Const1 + Computation\ inst \cdot Const2$$

$$\ldots.$$

$$Vector\ cycles = ct1 + c2 \cdot Mem.\ latency$$

$$Total\ no.\ of\ cycles = ct3 + ct \cdot Mem.\ latency$$

Figure 4.5 plots the evolution of the total number of instructions executed for various values of the section size. From this Figure we can see that as long as the program vector length is longer than the section size, when increasing the section size the number of instructions drops. As soon as the section size becomes larger than the vector length the number of instructions gets saturated and no longer drops. Remember, for the studied kernel the maximum vector length is 16 elements.

From the same figure we can note that the vectorized kernel executes significantly less instructions than the scalar only (non-vectorized) kernel, for long section size up to five times less instructions. In the following, we shall investigate the reasons for such a large difference in the number of executed instructions.

Section size	Total no of inst	No of vect inst	Percent of all inst	Memory inst	Computation inst
4	148	54720	36.73	11520	43200
8	89	27840	31.12	5760	22080
16	59	14400	24.12	2880	11520
32	59	14400	24.12	2880	11520
64	59	14400	24.12	2880	11520
128	59	14400	24.12	2880	11520
256	59	14400	24.12	2880	11520
512	59	14400	24.12	2880	11520
1024	59	14400	24.12	2880	11520

Table 4.2: MPEG2 Encoder, kernel dist1: Number of vector instructions

Table 4.2 shows how the number of executed instructions varies as the section size increases. For the section size of four elements we have in total approximately 55 thousands vector instructions and a total of approximately 150 thousands of executed instructions (both scalar and vector). Since in this case the section size is four, we can consider that each vector instruction replaces four scalar instructions. Thus, we can estimate that the scalar version of the kernel would take $150k + 4 \cdot 55k \approx 370k$ instructions.

However, the scalar kernel executes only 270k instructions, 100k fewer instructions than our estimate. It is clear that the extra instructions are all scalar ones. The reason these extra instructions are there is because we use inline assembly to insert the vector assembly instructions into the high level C code of the kernel. For example, to load the contents of array a[i] into a vector register we use the following code:

```
__asm__ __volatile__    (                       \
    "v.ld           $1, %0       \n\t"   \
    :                                           \
    :                                           \
    "r" (&a[i])                                 \
                            );
```

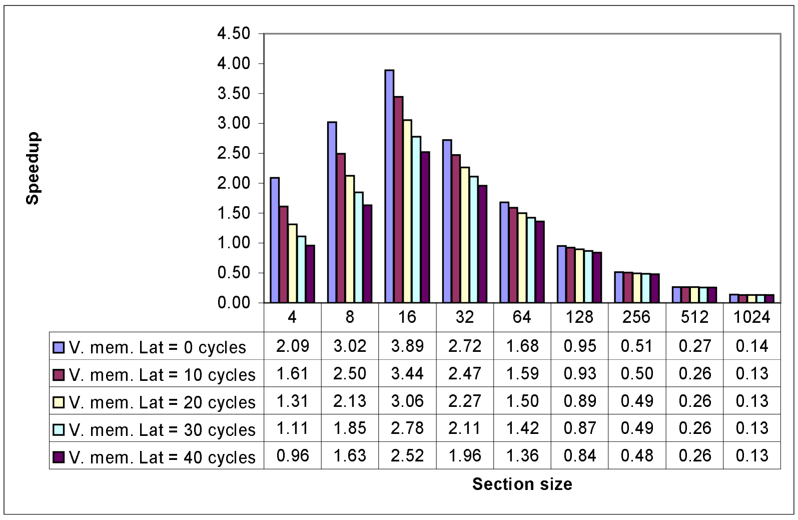
This code loads the array which has the first byte at address &a[i] in the vector register $1. However, in the background, the compiler calculates the address of a[i], loads it into a register and replaces %0 with the name of the register containing &a[i]. Thus, in our example, for every vector load, there are around three additional scalar instructions. The same holds for vector store instructions.

These additional scalar instructions do not appear in real implementations of vector processors. For real implementations, the auto-sectioning instructions update the memory addresses directly, while we update in 'hardware' only the array indexes. In other words, real implementations perform the address computation in vector hardware while we do it through both vector hardware and scalar instructions.

The reason we adopt this scheme is to reduce overall simulation complexity. When writing a vector instruction using inline assembly, we have to pass scalar register names and vector register names. However, the scalar register allocation is done by the C compiler, so we do not know which register will contain a particular variable. This forces us to let the compiler perform these additional scalar instructions instead of computing the address in hardware.

Note that the vector registers in our simulator are not visible to the C compiler, nor are they visible to the assembler (if they where visible, we would have a vectorizing compiler which unfortunately is not the case). We use a standard C compiler, assembler and linker that target the PISA architecture. The vector instructions that we add are left unchanged by both the C compiler and the assembler. Only the simulator recognizes them.

Let us take a look at the speedups obtained for the various section sizes used. These are plotted in Figure 4.6. This graph allows for a better understanding of the influence of application vector lengths (on the software side) and vector section size and vector memory latency (on the hardware side).



	4	8	16	32	64	128	256	512	1024
V. mem. Lat = 0 cycles	2.09	3.02	3.89	2.72	1.68	0.95	0.51	0.27	0.14
V. mem. Lat = 10 cycles	1.61	2.50	3.44	2.47	1.59	0.93	0.50	0.26	0.13
V. mem. Lat = 20 cycles	1.31	2.13	3.06	2.27	1.50	0.89	0.49	0.26	0.13
V. mem. Lat = 30 cycles	1.11	1.85	2.78	2.11	1.42	0.87	0.49	0.26	0.13
V. mem. Lat = 40 cycles	0.96	1.63	2.52	1.96	1.36	0.84	0.48	0.26	0.13

Figure 4.6: MPEG2 Encoder, kernel dist1: Vectorization speedups for the studied section sizes and memory latencies

While the section size is smaller than the vector length, when increasing the section size, the speedup also increases. When the section size becomes larger than the application vector length the speedup begins to drop because the vector unit performs unnecessary operations on vector elements that are not used.

We can note again the influence of the vector memory latency on performance. For small section sizes, the vector memory latency can decrease performance significantly. For instance, with a section size of four elements and a vector memory latency of 40 cycles (the scalar memory latency is 18 cycles), the execution time is twice as long as for the same section size but with a perfect vector memory system with a vector memory latency of 0 cycles.

However, for long section sizes, of 128 elements and more, the impact of vector memory latency on execution time is minimal. Executing the vectorized kernel with a vector memory latency of 40 cycles is just a few percents slower than with a perfect memory system.

Vectorizing this kernel speeds up its execution time by up to four times. We are not satisfied by the results and look at ways to further improve performance. The bottleneck for this limited speedup is the vector length that is simply too short to use the vector processor at it's full potential. The MPEG2 encoder splits the image into small blocks of 8 by 8 pixels and performs the computation on each of these blocks independently. A possible solution to improve performance is to segment the image into larger blocks or blocks with the width larger than the height. This would offer longer vectors and vectorization would provide better results. However, changing the segmentation method might have severe impact on the way the application is written and on the video quality.

4.3.1.2 Function fdct

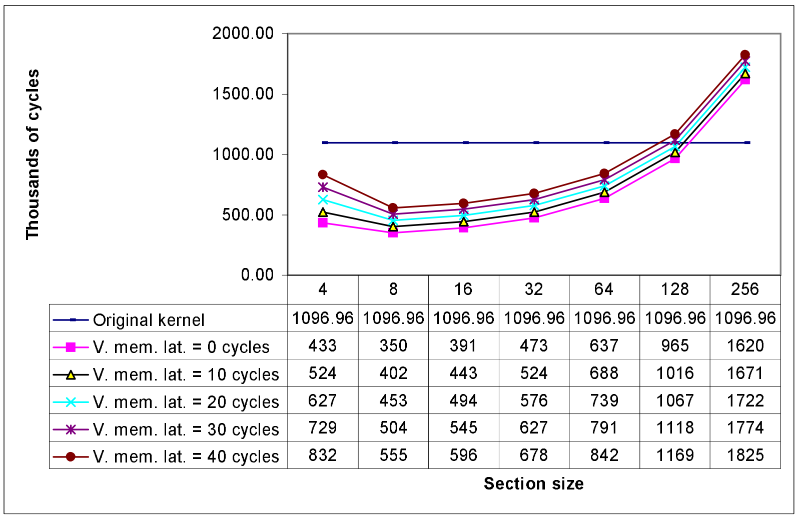
The inner most loop of the Finite Discreet Cosine Transform looks like this:

```
          s = 0.0;

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

          tmp[h*i+j] = s;
```

This vectorized looks like this:

```
          tmp[h*i+j] = 0;
          vl = w;
          ld_index = 0;

          __asm____volatile__("v.setvconf %0, %1, %2"::"r" (vl),"r"(0),"r"(0));

          while (vl)
          {
/*Load the data*/
            __asm__ __volatile__ (                    \
            "v.ldstep       $1, %0, 1       \n\t"   \
            "v.ld.d         $2, %1          \n\t"   \
            :                                         \
            :                                         \
            "r" (&block[h*i + ld_index]),             \
            "r" (&c[j][ld_index])                     \
                                );
/*Perform the computation*/
            __asm__ __volatile__ (                  \
            "v.i.to.d      $1, $1          \n\t"   \
            "v.mul.d       $3, $1, $2      \n\t"   \
            :                                       \
            :                                       \
                                ); 

/*write back results and update pointers*/
            __asm__ __volatile__ (                  \
            "v.sumup.d     %0, $5          \n\t"   \
            "v.updateindex %1              \n\t"   \
            "v.updatevl    %2              \n\t"   \
            :                                       \
            "=r" (s_tmp),                           \
            "=r" (ld_index),                        \
            "=r" (vl)                               \
            :                                       \
                               );
            tmp[h*i+j] += s_tmp;
         }
```

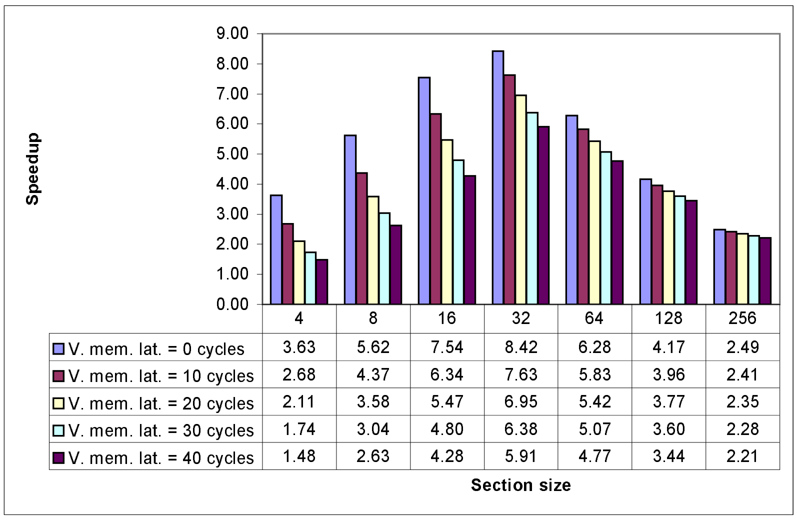
This kernel behaves very similar to the one previously described. We have also run the same tests as for the previous kernel. Figure 4.7 plots the execution time of the kernel for various section sizes and vector memory latencies.



	4	8	16	32	64	128	256
— Original kernel	1096.96	1096.96	1096.96	1096.96	1096.96	1096.96	1096.96
■ V. mem. lat. = 0 cycles	433	350	391	473	637	965	1620
▲ V. mem. lat. = 10 cycles	524	402	443	524	688	1016	1671
× V. mem. lat. = 20 cycles	627	453	494	576	739	1067	1722
✱ V. mem. lat. = 30 cycles	729	504	545	627	791	1118	1774
● V. mem. lat. = 40 cycles	832	555	596	678	842	1169	1825

Figure 4.7: MPEG2 Encoder, kernel fdct: Kernel execution time as a function of the section size

For the fdct kernel, the maximum vector length is 8 since it performs computation only inside the block thus, vectorization does not offer high speedups because the vector are just too short. Figure 4.8 shows the variation of the speedup as the section size increases. As expected, the best speedup is obtained when the section size is equal to the vector length. We can observe similar behavior as in the dist1 case however, even though has shorter vectors, the fdct function offers better speedups. This is because for vectorization, the dist1 kernel required loading an array twice, load that substantially reduced performance (see the previous section for details).



	4	8	16	32	64	128	256
🟦 V. mem. lat. = 0 cycles	3.63	5.62	7.54	8.42	6.28	4.17	2.49
🟥 V. mem. lat. = 10 cycles	2.68	4.37	6.34	7.63	5.83	3.96	2.41
🟨 V. mem. lat. = 20 cycles	2.11	3.58	5.47	6.95	5.42	3.77	2.35
⬜ V. mem. lat. = 30 cycles	1.74	3.04	4.80	6.38	5.07	3.60	2.28
🟪 V. mem. lat. = 40 cycles	1.48	2.63	4.28	5.91	4.77	3.44	2.21

Figure 4.8: MPEG2 Encoder, kernel fdct: Vectorization speedups

A particularity of this kernel is the data types used. We have integer pixel values that are multiplied by floating point coefficients. The integer values are converted to floating point values, then, a floating point multiplication is performed followed by a reduction to a single floating point value (sumup). This is the only kernel we have analyzed that works with both integer and floating point data.

4.3.1.3 Function qna

Function qna is the most complex function that we have analyzed. The loop contains a substantial amount of computation for each iteration. Also, computation is done on all the pixels of a block thus, offering vector lengths of 64, again the highest of all studied kernels. The combination of long vector lengths and high amount of computation per iteration has brought very high speedups.

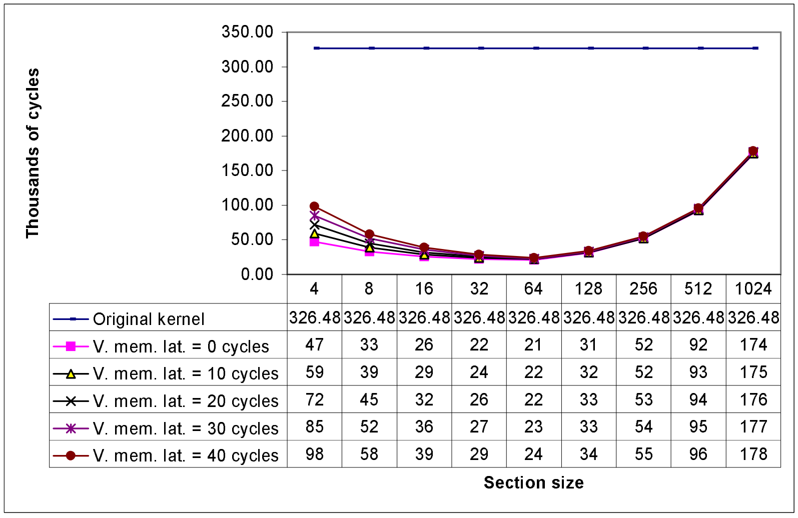
```
for (i=0; i<64; i++)
{
  x = src[i];
  d = quant_mat[i];
  y = (32*(x>=0 ? x : -x) + (d>>1))/d;
```

```
        if ((dst[i] = (x>=0 ? y : -y)) != 0)
          nzflag=1;
      }
```

Vectorized, this turns into:

```
    vl = 64;
    ld_index = 0;

    __asm__ __volatile__("v.setvconf %0, %1, %2"::"r" (vl),"r" (0), "r" (0));

    while (vl)
    {
/*Load the data*/
        __asm__ __volatile__ (                          \
        "v.ldstep       $1, %0, 1       \n\t"   \
        "v.ldstep       $2, %1, 1       \n\t"   \
        :                                               \
        :                                               \
        "r" (&src[ld_index]),                           \
        "r" (&quant_mat[ld_index])                      \
                            );
/*Perform the computation*/
        __asm__ __volatile__ (                          \
        "v.scompare.lt  $3, $1, $0      \n\t"   \
        "v.maskson                      \n\t"   \
        "v.ssub         $3, $0, $1      \n\t"   \
        "v.masksoff                     \n\t"   \
        "v.smul         $4, %0, $3      \n\t"   \
        "v.shr          $5, $2, 1       \n\t"   \
        "v.add          $6, $4, $5      \n\t"   \
        "v.div          $7, $6, $1      \n\t"   \
        "v.maskson                      \n\t"   \
        "v.ssub         $3, $0, $7      \n\t"   \
        "v.masksoff                     \n\t"   \
        :                                               \
        :                                               \
        "r" (32)                                        \
                            );
/*write back results and update pointers*/
        __asm__ __volatile__ (                          \
        "v.sumup        %2, $3          \n\t"   \
        "v.ststep       %3, $3, 1       \n\t"   \
        "v.updateindex  %0              \n\t"   \
        "v.updatevl     %1              \n\t"   \
        :                                               \
        "=r" (ld_index),                        \
        "=r" (vl),                              \
        "=r" (nzflag)                           \
        :                                       \
        "r" (&dst[ld_index])                    \
                       );
    }
```

In the scalar implementation, the check 'if ($x \geq 0$)' is done twice for each iteration. In the vector implementation, we make this comparison once and store it in a bit vector. The next time, we use the same bit vector computed earlier, saving a lot of computation cycles.

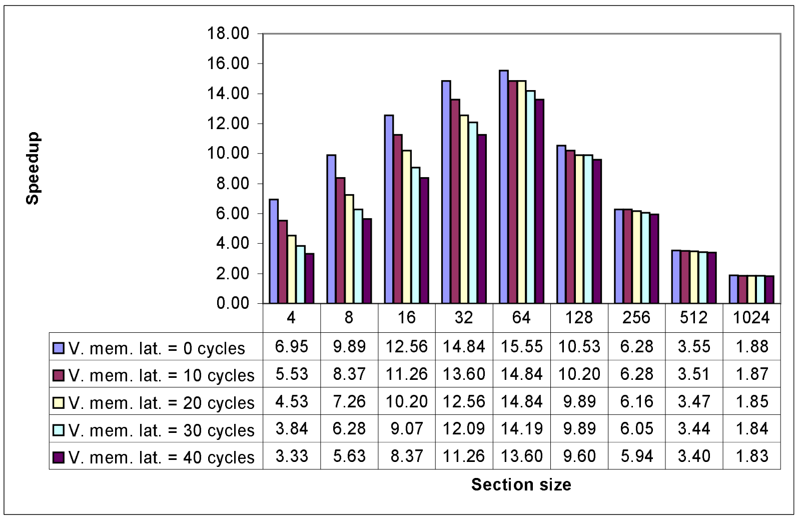


	4	8	16	32	64	128	256	512	1024
Original kernel	326.48	326.48	326.48	326.48	326.48	326.48	326.48	326.48	326.48
V. mem. lat. = 0 cycles	47	33	26	22	21	31	52	92	174
V. mem. lat. = 10 cycles	59	39	29	24	22	32	52	93	175
V. mem. lat. = 20 cycles	72	45	32	26	22	33	53	94	176
V. mem. lat. = 30 cycles	85	52	36	27	23	33	54	95	177
V. mem. lat. = 40 cycles	98	58	39	29	24	34	55	96	178

Figure 4.9: MPEG2 Encoder, kernel qna: Execution time as a function of the section size and the memory latency

Figure 4.9 shows the execution time of this kernel for various section sizes. Each curved line corresponds to a different vector memory latency. What is striking from this picture is the very low execution time of the vectorized version compared to the scalar implementation. Even for small section sizes, the vectorized version greatly outperforms the scalar one. Of course, the best performance is reached for section size equal to the vector length of 64.

This kernel, with its long vectors allows for a larger spectrum of section sizes that brings positive results (speedups). This enables us to analyze the influence of the section size alone on the speedup. The speedups obtained through vectorization are shown in Figure 4.10. From this graph we can see very clearly the sublinear evolution of speedup when increasing the section size. We can note that after a certain point, doubling the section size brings very little increase in performance. For instance for section size of 32 the maximum speedup is 14.84x while for section size of 64 elements the maximum speedup is 15.55x, only slightly bigger than before. This can be explained as follows: each time the section size doubles, the number of vector instructions is halved (this



	4	8	16	32	64	128	256	512	1024
V. mem. lat. = 0 cycles	6.95	9.89	12.56	14.84	15.55	10.53	6.28	3.55	1.88
V. mem. lat. = 10 cycles	5.53	8.37	11.26	13.60	14.84	10.20	6.28	3.51	1.87
V. mem. lat. = 20 cycles	4.53	7.26	10.20	12.56	14.84	9.89	6.16	3.47	1.85
V. mem. lat. = 30 cycles	3.84	6.28	9.07	12.09	14.19	9.89	6.05	3.44	1.84
V. mem. lat. = 40 cycles	3.33	5.63	8.37	11.26	13.60	9.60	5.94	3.40	1.83

Figure 4.10: MPEG2 Encoder, kernel qna: Vectorization speedups

behavior is observed only while the section size is smaller than the vector length). Also, the number of scalar instructions caused by the inline assembly drops. What remains constant is the scalar part of the kernel that cannot be improved. Thus, even with perfect memory systems and infinite vector lengths we get diminishing returns from increasing the section size beyond a certain point. We are interested in finding the section size that offers the best performance - cost tradeoff. For this particular case, the optimum section size is 32 elements.

4.3.2 MPEG2 Decoder Kernels

Profiling the decoder brings the results shown in Table 4.3.

Self	Called	Name
55.61%	476	Decode_Picture
20.75%	685,368	form_component_prediction
8.01%	685,440	clear_block
5.44%	4,617,448	flush_buffer
4.39%	63,687	decode_macroblock

Table 4.3: Profiling information for the MPEG2 decoder

The best candidate for vectorization is the function ”Decode_Picture”. However, the code is not vectorization friendly and cannot be changed. The functions ”form_component_prediction” and ”Clear_Block” account for almost 30% of the computation. These two functions contain loops that are regular enough to have potential for high vectorization rates.

4.3.2.1 Kernel form_component_prediction

This kernel is composed of eight loops. Depending on the input parameters, only one of the eight loops is executed. We have chosen to present only the most representative of these loops:

```
        for (i=0; i<w; i++)
        {
            v = d[i]+s[i];
            d[i] = (v+(v>=0?1:0))>>1;
        }
```

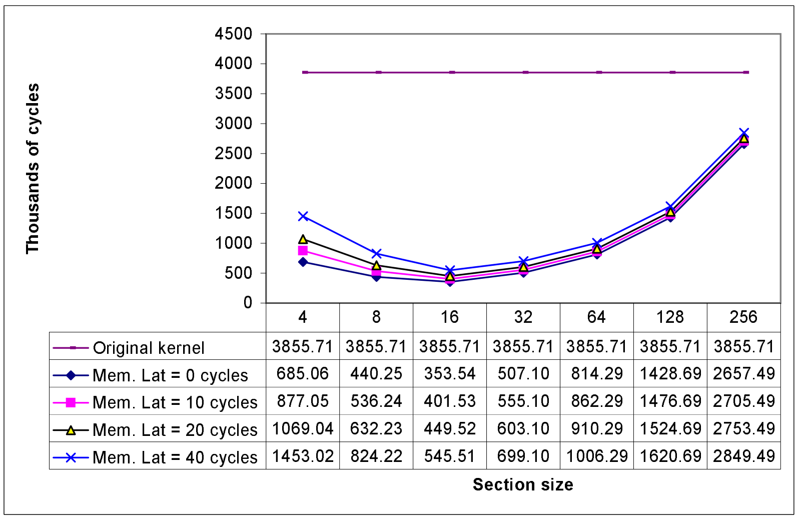
After vectorization, kernel form_component_prediction is coded like this:

```
        vl = w;
        d_index = 0;

        /* initialize the vector length, and load index
        __asm__ ("v.setvconf %0, %1, %2" : : "r" (vl), "r" (0), "r" (0));

        while (vl)
        {
/*read the two vectors*/
          __asm__ __volatile__ (                \
          "v.ldstep     $1, %0, 1       \n\t"   \
          "v.ldstep     $2, %1, 1       \n\t"   \
          :                                     \
          :                                     \
          "r" (&d[ld_index]),                   \
          "r" (&s[ld_index])                    \
                                  );
/*do the computation*/
          __asm__ __volatile__ (                    \
          "v.add            $3, $1, $2      \n\t"   \
          "v.scompare.gte   $3, $3, %0      \n\t"   \
          "v.maskson                        \n\t"   \
          "v.sadd           $3, %1, $3      \n\t"   \
          "v.masksoff                       \n\t"   \
          "v.shr            $3, $3, 1       \n\t"   \
          :                                         \
          :                                         \
          "r" (0),                                  \
          "r" (1)                                   \
                                 );

/*write back results and update pointers*/
          __asm__ __volatile__ (                \
          "v.ststep         %2, $3, 1   \n\t"   \
          "v.updateindex    %0          \n\t"   \
          "v.updatevl       %1          \n\t"   \
          :                                     \
          "=r" (ld_index),                      \
          "=r" (vl)                             \
          :                                     \
          "r"  (&d[ld_index])                   \
                              );
}
```

The vector length for this kernel is 16 which brings fairly good speedups. Also, each iteration contains conditional execution in the form of 'if then else' statements. As we have seen with the qna kernel, this kind of computation can be very well accelerated in hardware through the use of bit vectors. On the other hand, in the scalar version each conditional statement generates branches which add to the instruction count and jumps might be miss-predicted slowing down execution even more.

This kernel uses only integer computation, the element size is one byte and the used data types are arrays of bytes.

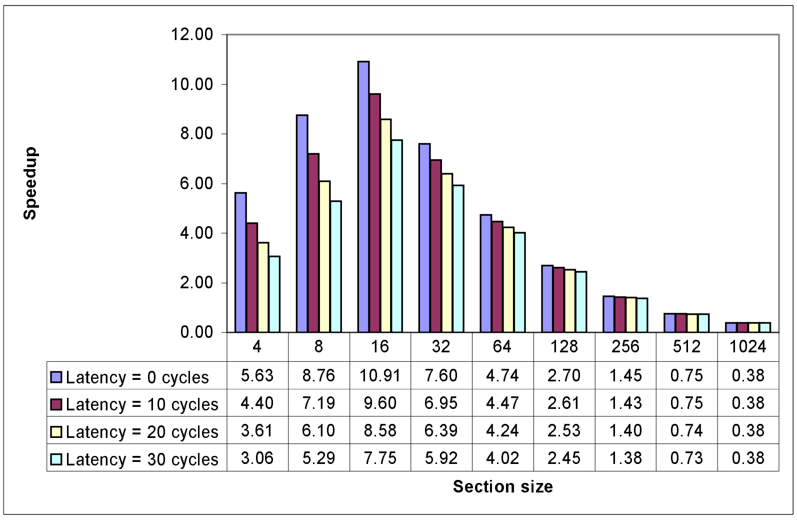
We show the graph for speedups when performing computation on a single block (Figure 4.12) and the comparison of speedups for one, two and four image blocks processed in parallel (Figure 4.11). For this situation the performance increases steadily when increasing the section size up to a maximum value of 64. This kernel also advocates for 64 as being the optimal section size.



	4	8	16	32	64	128	256
— Original kernel	3855.71	3855.71	3855.71	3855.71	3855.71	3855.71	3855.71
◆ Mem. Lat = 0 cycles	685.06	440.25	353.54	507.10	814.29	1428.69	2657.49
■ Mem. Lat = 10 cycles	877.05	536.24	401.53	555.10	862.29	1476.69	2705.49
△ Mem. Lat = 20 cycles	1069.04	632.23	449.52	603.10	910.29	1524.69	2753.49
× Mem. Lat = 40 cycles	1453.02	824.22	545.51	699.10	1006.29	1620.69	2849.49

Figure 4.11: MPEG2 Decoder, kernel comp_prediction: Execution time as a function of the section size and the memory latency

Figure 4.11 shows the execution time of the vectorized kernel when varying the section size an the vector memory latency. Even though the kernel offers vector lengths of only 16 performance gains are quite good, reaching speedups of close to 11x (see Figure 4.12).



	4	8	16	32	64	128	256	512	1024
Latency = 0 cycles	5.63	8.76	10.91	7.60	4.74	2.70	1.45	0.75	0.38
Latency = 10 cycles	4.40	7.19	9.60	6.95	4.47	2.61	1.43	0.75	0.38
Latency = 20 cycles	3.61	6.10	8.58	6.39	4.24	2.53	1.40	0.74	0.38
Latency = 30 cycles	3.06	5.29	7.75	5.92	4.02	2.45	1.38	0.73	0.38

Figure 4.12: MPEG2 Decoder, kernel comp_prediction: Vectorization speedups

We can again notice the reduced impact of the vector memory latency for long section sizes. For vector registers of over 128 elements, we get almost the same performance with both a perfect memory system or a very high latency memory. For this kernel, the best performance is reached with a section size of 16 elements.

4.3.2.2 Kernel clear_block

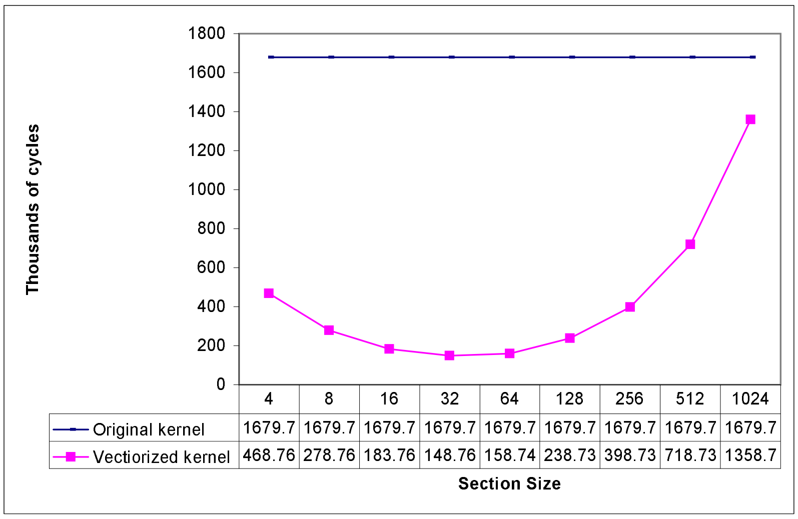
This function initializes the memory before computation takes place. This means that there are no vector loads and no vector computations, only vector stores. This makes the kernel immune to memory latency. The memory is initialized with zero and since in our architecture the register 0 is reserved for value 0, the vectorized kernel is trivial.

The scalar loop looks like this:

```
      for (i=0; i<64; i++)
         *Block_Ptr++ = 0;
```

This vectorized is like this:

```
        vl = 64;
        ld_index = 0;
        /* initialize the vector length, and load index, disable mask operation*/
        __asm__ __volatile__("v.setvconf %0, %1, %2"::"r" (vl), "r" (0), "r" (0));

        while (vl)
        {
/*No computation is required since the vector register 0 contains
        the vector value of zero*/
            __asm__ __volatile__ (                  \
            "v.ststep         %2, $0, 1   \n\t"   \
                "v.updateindex    %0           \n\t"   \
                "v.updatevl       %1           \n\t"   \
                :                                      \
                "=r" (ld_index),                       \
                "=r" (vl)                              \
                :                                      \
                "r"  (&Block_Ptr[ld_index])            \
                                     );
        }
```



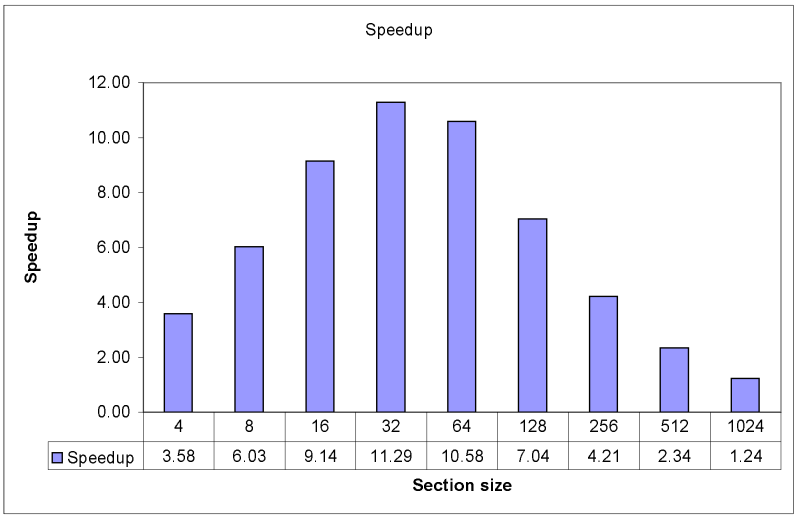
	4	8	16	32	64	128	256	512	1024
Original kernel	1679.7	1679.7	1679.7	1679.7	1679.7	1679.7	1679.7	1679.7	1679.7
Vectiorized kernel	468.76	278.76	183.76	148.76	158.74	238.73	398.73	718.73	1358.7

Figure 4.13: MPEG2 Decoder, kernel clear_block: Execution time when varying the section size

The vector length is 64 and the data type is 1 byte integer. This is by far the simplest kernel studied. Figure 4.13 plots the execution time as a function of the section size. This time the graph depicts a single curved line because, as stated before, this kernel does not have any vector loads, thus, it is immune to vector memory latency. Figure 4.14 shows the speedups obtained through vectorization. Best performance is obtained for section size of 32 elements.

4.3.3 MPEG2 - Entire Application

In the previous section we have seen how the individual kernels perform when changing various simulation parameters. In this section we combine all results and analyze speedups for the entire applications. We have already seen that the best performance is obtained for vector registers 16 or 32 elements wide. The behavior of the studied kernels is illustrated in Table 4.4 for section sizes of 8, 16, 32 and 64 elements. The Table is constructed in the following manner: the second column contains the name of the kernel, the third column shows the percent of cycles spent inside the corresponding kernel with respect to the application execution time. Columns four to seven show the speedup of



Section size	4	8	16	32	64	128	256	512	1024
▨ Speedup	3.58	6.03	9.14	11.29	10.58	7.04	4.21	2.34	1.24

Figure 4.14: MPEG2 Decoder, kernel clear_block: Vectorization speedups

the isolated vectorized kernel over the isolated scalar kernel for section size of 8, 16, 32 and 64 elements respectively.

			Speedup			
	kernel	time	8	16	32	64
Encoder	dist1	73.0%	2.13	3.06	2.27	1.50
	fdct	7.84%	2.42	2.22	1.90	1.48
	qna	4.8%	7.26	10.20	12.56	14.84
Decoder	comp_pred	20.75%	6.10	8.58	6.39	4.24
	clr_blk	8.01%	6.03	9.14	11.29	10.58

Table 4.4: MPEG2: Comparative kernel speedups for section size of 8, 16, 32 and 64 elements

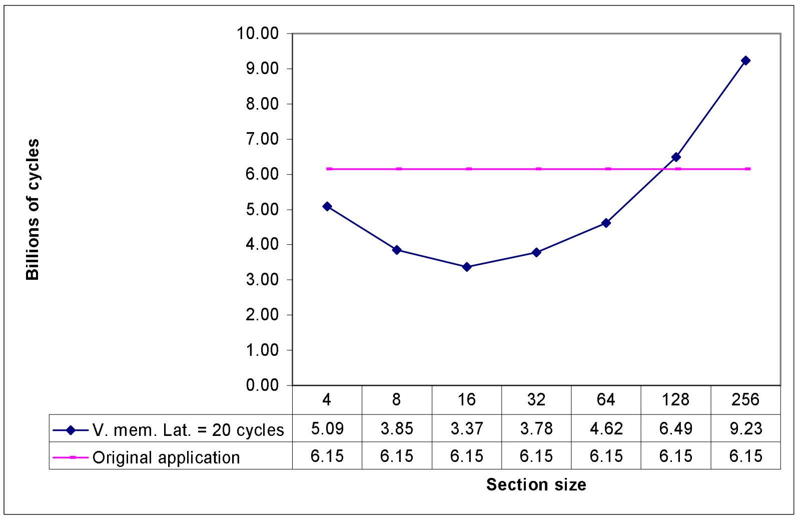
This Table helps us better understand the influence of each kernel over the entire application. It is interesting to note that each kernel has different vector lengths and for a fixed section size, some kernels perform better while others perform not that good. For instance, the MPEG2 Encoder contains kernels $dist1$, $fdct$ and qna that have vector lengths of 16, 8 and 64 elements respectively thus, $dist1$ will perform best for vector registers of 16 elements while for vector registers of 64 elements will perform poorly, $fdct$ performs best for vector registers of 8 elements and so on. The overall application speedup is dictated by the individual kernel speedups weighted by the percent of time spent inside each kernel.

Table 4.5 presents the overall application speedup for various vector register widths (section size). Vectorization level does not mean the percent of vector instructions from all executed instructions but instead, is the cumulative execution time of the considered scalar kernels of the scalar application execution time. For instance, for the scalar version of the MPEG2 Encoder, functions $dist1$, $fdct$ and qna account for 85.64% of the execution time. Only these three functions could be efficiently vectorized. Note that the number of vector instruction is dependent on the section size. The maximum theoretical speedup is considered for infinite kernel speedup. Of course this value can never be reached but is presented here as the upper bound of the achievable speedup. We can estimate the efficiency of our work by comparing our results to this upper bound.

	Vectorization level (relative to execution time)	Maximum theoretical speedup (for kernel speedup of infinity)	Obtained speedup for section size of 16	32	64
Encoder	85.64%	6.9	1.82	1.63	1.33
Decoder	28.76%	1.40	1.25	1.25	1.23

Table 4.5: MPEG2: Overall application speedup for one block processed at a time

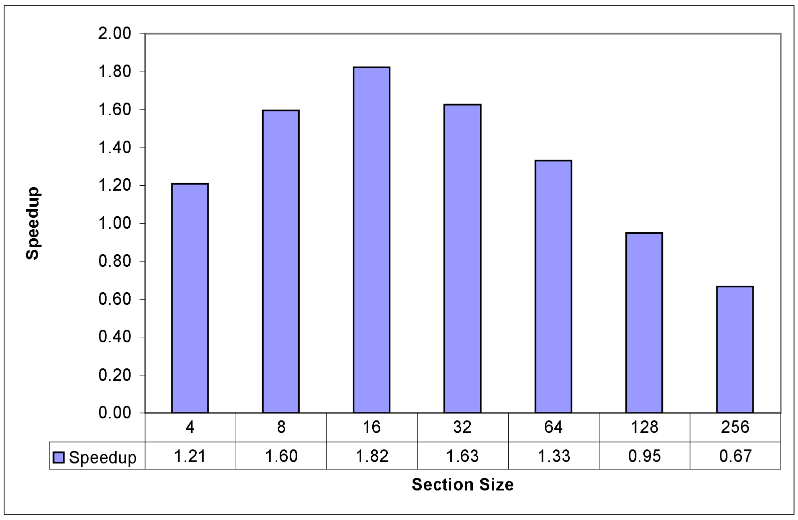
Aside from simulating each kernel independently, we have also simulated the whole vectorized applications consisting of both the vectorized kernels and the scalar part that could not be vectorized. Figure 4.15 shows the overall execution time for the entire vectorized MPEG2 Encoder for the considered values of the section size. Figure 4.16 shows the speedup of the vectorized version over the scalar implementation. For both these graphs, the vector memory latency is 20 cycles. The speedup is determined mostly by the behavior of the *dist*1 kernel since it takes 73.0% of all the scalar execution time. This kernel has vector lengths of 16 elements therefore, the whole application performs best for section size of 16 elements. Even though kernel *qna* reaches speedups of up to almost 15x, it only accounts for less than 5% of the computation time thus, it hardly influences the overall application speedup.



	4	8	16	32	64	128	256
V. mem. Lat. = 20 cycles	5.09	3.85	3.37	3.78	4.62	6.49	9.23
Original application	6.15	6.15	6.15	6.15	6.15	6.15	6.15

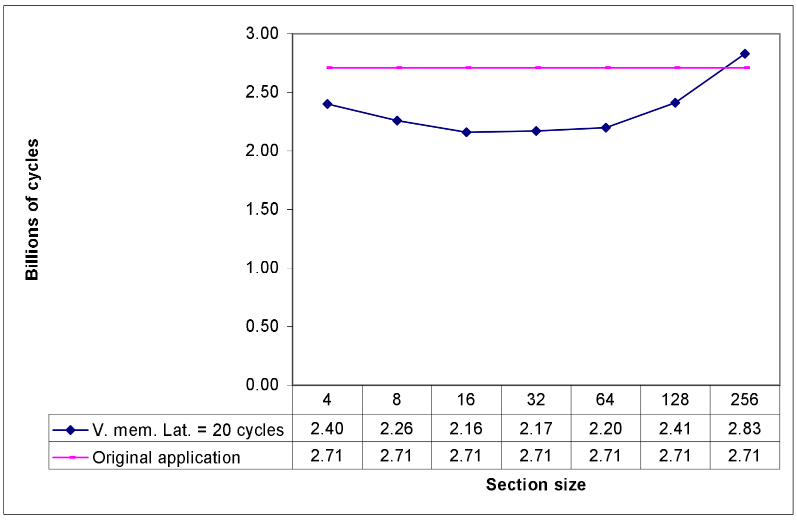
Figure 4.15: MPEG2 Encoder: application execution time for various section sizes

Figure 4.17 shows the overall execution time for the entire vectorized MPEG2 Decoder for the considered values of the section size. Figure 4.18 shows the speedup of the vectorized version over the scalar implementation. Again, for both these graphs the



	4	8	16	32	64	128	256
■ Speedup	1.21	1.60	1.82	1.63	1.33	0.95	0.67

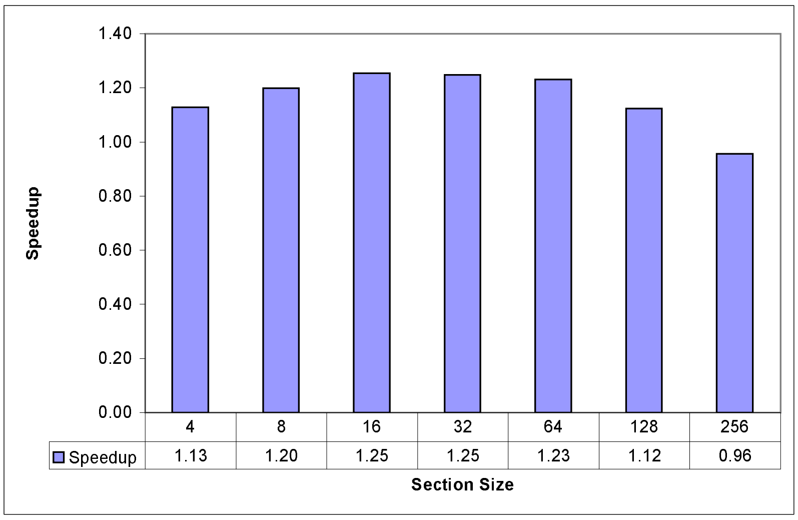
Figure 4.16: MPEG2 Encoder: application speedup for various section sizes



	4	8	16	32	64	128	256
◆ V. mem. Lat. = 20 cycles	2.40	2.26	2.16	2.17	2.20	2.41	2.83
■ Original application	2.71	2.71	2.71	2.71	2.71	2.71	2.71

Figure 4.17: MPEG2 Decoder: application execution time for various section sizes

vector memory latency is 20 cycles. For the MPEG2 Decoder, the kernels that could be vectorized take only 28.76% of the execution time thus the highest possible speedup is 1.40x. The considered kernels, *comp_prediction* and *clear_block* offer good speedups, on average 8x and 10x respectively and have vector lengths of 64 and 16 respectively. Also in this case when one of the kernels has best performance with respect to the section size, the other kernel offers poor speedups. However, because the two kernels take less than 30% of all the application execution time the speedup obtained through vectorization is not spectacular but still is quite close to the theoretical upper bound. What is interesting to note is that for large section sizes, none of the two kernels suffer significant performance degradation. Even for vector registers of 256 elements, both vectorized



	4	8	16	32	64	128	256
▨ Speedup	1.13	1.20	1.25	1.25	1.23	1.12	0.96

Figure 4.18: MPEG2 Decoder: application speedup for various section sizes

functions perform better than their scalar counterparts. This is not the case for *dist*1 and *fdct* which for long registers perform much worse than the scalar version.

4.3.3.1 MPEG2 - Summary

In the previous pages we have looked at both kernels and entire MPEG2 encoder and decoder applications. For these programs the computation is done on small image blocks of 8 by 8 pixels. This limits the possible vector lenghts to the following values:

- vector length of 8 when the computation is performed on the rows or columns of a single bolck
- vector length of 16 when the computation is performed on two adjacent blocks
- vector length of 64 when the computation is performed on each element of a block

We have seen that speedup grows with longer vectors and larger section sizes but only up to a point. If the section size is larger than the vector lengths, the benefits from further increasing the register lengths diminishes rapidly. Therefore, for the MPEG2 the best performance/complexity ratio is obtained with vector registers that are 16 or 32 elements wide.

Also, more complex computation brings greater speedups than simple computation. Especially the use of masked operations greatly accelerates execution since in the scalar case if-then-else statements require the use of branches that might severely impact performance.

Table 4.6 shows the vector lengths for the studied kernels. Note that the fdct kernel has vectors of 8 elements long thus, for section size of 32 elements is too long. For kernels component_prediction and dist1 the section size of 16 corresponds to the highest speedup while for the kernels clear_block and qna 64 is the peak speedup. On average, vector registers of 16 or 32 elements wide bring the best performance.

	cl_block	comp_pred	dist1	fdct	qna
vector length for each block	64	16	16	8	64

Table 4.6: MPEG2: Vector lengths for the studied kernels

4.3.4 Image processing

We have implemented a digital median filter used in image processing to be considered as synthetic benchmark. This application has been described in detail in section 4.1 when presenting how coding decision impact vector performance.

The median filter is used to improve the quality of images by removing noise, pixels that have colors very different from the surrounding pixels. The value of each point in the new image is the weighted average of several adjacent pixels of the old image. The images are stored uncompressed in three separate matrixes, one for each of the color components (red green and blue) and computation is done for each of the color panes. For our implementation we have used a square window of nine pixels.

The main loop of the implementation looks like this:

```
for i = 1 to height do
    for j = i to width do
        loop over adjacent pixels
            multiply with coefficient
        new_val[i][j] = sumup
```

We consider two implementation variations:

- Standard implementation that has short vectors, written the scalar way
- To the standard implementation, mentioned above, we have applied loop interchange to significantly increase the vector lengths

4.3.4.1 Basic implementation of the Median Filter

For the basic implementation, the main loop looks like this:

```
for (i = 1; i < HEIGHT-1; i++)
    for (j = 1; j < WIDTH-1; j++)
        for (k = 0; k < 9; k++)
            out[i][j] += coef[k] * in [i + x[k]][j + y[k]];
```

The innermost loop vectorized is shown in the following lines:

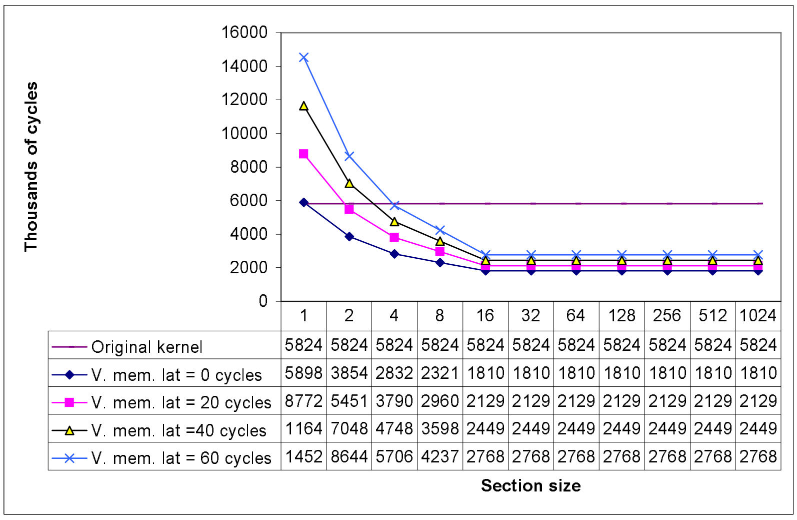
```
/*create a temporary vector to contain the values of the neighboring
pixels*/
            t[0] = in[i-1][j-1];
            t[1] = in[i-1][j];
            t[2] = in[i-1][j+1];
            t[3] = in[i][j-1];
            t[4] = in[i][j];
            t[5] = in[i][j+1];
            t[6] = in[i+1][j-1];
            t[7] = in[i+1][j];
            t[8] = in[i+1][j+1];

            vl = 9;
            ld_index = 0;
/* initialize the vector length, and load index, disable mask
operation*/
            __asm__ __volatile__ ("v.setvconf %0, %1, %2" : : "r" (vl), "r" (0), "r"

            while (vl)
            {
/*read the three vectors*/ /*$1 = t*/ /*$2 = coef*/ /*$3 = in*/
                __asm__ __volatile__ (                  \
                "v.ldstep     $1, %0, 1              "  \
                "v.ldstep     $2, %1, 1              "  \
                "v.ldstep     $3, %2, 1              "  \
                :                                       \
                :                                       \
                "r" (&t[ld_index]),                     \
                "r" (&coef[ld_index]),                  \
                "r" (&out[i][j+ld_index])               \
                                );

/*perform the computation and write back results*/
                __asm__ __volatile__ (                    \
                "v.mul            $4, $1, $2          "   \
                "v.sumup          $4, %2              "   \
                "v.updateindex    %0                  "   \
                "v.updatevl       %1                  "   \
                :                                         \
                "=r" (ld_index),                          \
                "=r" (vl)                                 \
                :                                         \
                "r"  (&out[i][j])                         \
                                  );
            } /*while*/
```

In the first step a temporary array is created that contains the values of the neighboring pixels. Then these values are multiplied with the corresponding coefficients and then added together.

Figure 4.19 plots the execution time of the digital filter when varying the section size and the vector memory latency. Clearly, the vector length is not enough to show the full potential o the vector unit. Nine elements packed in a vector instruction offer speedups of only 3.2 compared to the scalar implementation.

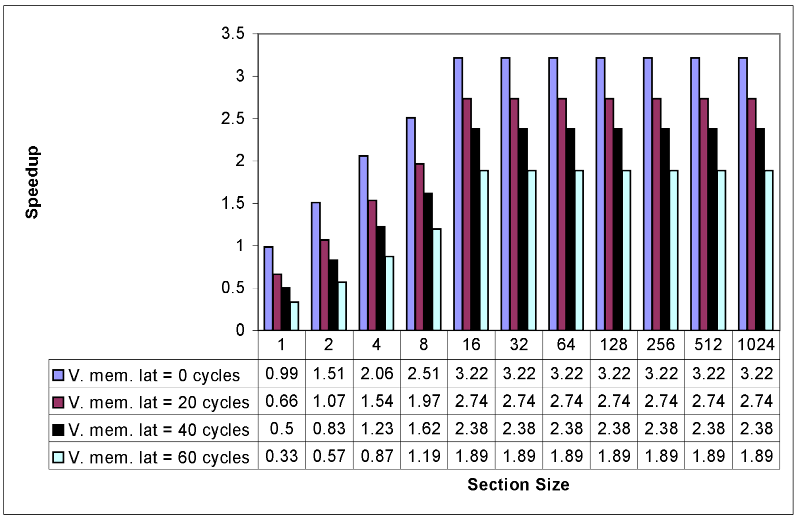
Figure 4.20 shows the speedup of the vectorized version over the scalar only implementation.

Note that the size of the image does not impact the vector length. In the following subsection we shall make a small change to the coding of the algorithm that will enable vector lengths proportional to the image size.



	1	2	4	8	16	32	64	128	256	512	1024
— Original kernel	5824	5824	5824	5824	5824	5824	5824	5824	5824	5824	5824
◆ V. mem. lat = 0 cycles	5898	3854	2832	2321	1810	1810	1810	1810	1810	1810	1810
■ V. mem. lat = 20 cycles	8772	5451	3790	2960	2129	2129	2129	2129	2129	2129	2129
△ V. mem. lat =40 cycles	1164	7048	4748	3598	2449	2449	2449	2449	2449	2449	2449
× V. mem. lat = 60 cycles	1452	8644	5706	4237	2768	2768	2768	2768	2768	2768	2768

Figure 4.19: Digital filter, basic implementation: Execution time as a function of section size



	1	2	4	8	16	32	64	128	256	512	1024
■ V. mem. lat = 0 cycles	0.99	1.51	2.06	2.51	3.22	3.22	3.22	3.22	3.22	3.22	3.22
■ V. mem. lat = 20 cycles	0.66	1.07	1.54	1.97	2.74	2.74	2.74	2.74	2.74	2.74	2.74
■ V. mem. lat = 40 cycles	0.5	0.83	1.23	1.62	2.38	2.38	2.38	2.38	2.38	2.38	2.38
□ V. mem. lat = 60 cycles	0.33	0.57	0.87	1.19	1.89	1.89	1.89	1.89	1.89	1.89	1.89

Figure 4.20: Digital filter, basic implementation: Speedup for various section sizes

4.3.4.2 Optimized implementation of the Median Filter

A significant improvement over the previous method can be achieved by loop interchange. As stated earlier, when the code contains multiple nested loops, only the innermost loop can be vectorized. In our case, the outer loop offers vector length equal to the image width thus, a simple solution is to interchange the loops. In this case, the code would look like this:

```
    for (i = 1; i < HEIGHT-1; i++)
        for (k = 0; k < 9; k++)
            for (j = 1; j < WIDTH-1; j++)
                out[i][j] += coef[k] * in [i + x[k]][j + y[k]];
```

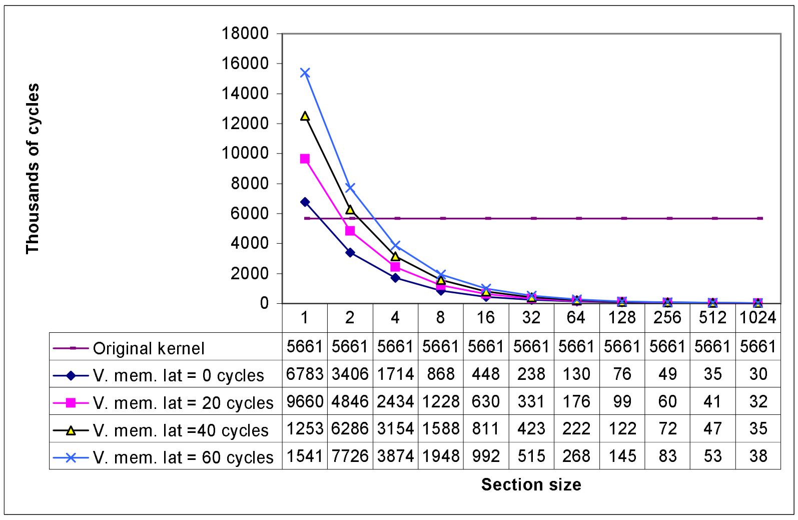
Through vectorization, the innermost loop becomes:

```
            vl = WIDTH - 1;
            ld_index = 1;
/* initialize the vector length, and load index, disable mask
operation*/
            __asm__ __volatile__ ("v.setvconf %0, %1, %2"::"r"(vl), "r" (0), "r" (0)

            while (vl)
            {
/*read the two vectors*/ /*$1 = in*/ /*$2 = out*/
                __asm__ __volatile__ (                      \
                "v.ldstep     $1, %0, 1           "         \
                "v.ldstep     $2, %1, 1           "         \
                :                                           \
                :                                           \
                "r" (&in [i + x[k]][j + y[k] + ld_index ]),\
                "r" (&out[i][j+ld_index])                   \
                                    );

/*do the computation and write back results*/
                __asm__ __volatile__ (                     \
                "v.smul           $3, %2, $1      "        \
                "v.add            $4, $2, $3      "        \
                "v.ststep         %2, $4, 1       "        \
                "v.updateindex    %0              "        \
                "v.updatevl       %1              "        \
                :                                          \
                "=r" (ld_index),                           \
                "=r" (vl),                                 \
                "=r" (coef[k])                             \
                :                                          \
                "r"  (&out[i][j+ld_index])                 \
                                   );
```

```
        }
```

In this case, there is no need to create any temporary array. Figure 4.21 shows the execution time of the new implementation as a function of the section size and vector memory latency. The simulation was run on an image on thousand pixels wide thus, the vector length is one thousand. For this application, the vectors are long enough to exceed all the values for the section size that we have considered in our study. Therefore, the graph for this application has a different shape than for the kernels studied so far. The vector lengths are more than enough for every viable section size. We can again observe how doubling the section size beyond a certain threshold brings diminishing returns. This threshold is application dependent and the average value we have determined for all of the applications studied is 64 elements.

Figure 4.22 plots the speedup over the scalar only implementation. We can clearly see the advantages of having long vector lengths. We can achieves speedups of up to 180 for section size 1024, a significant improvement over the 3.2 speedup of the basic filter implementation.



Section size	1	2	4	8	16	32	64	128	256	512	1024
— Original kernel	5661	5661	5661	5661	5661	5661	5661	5661	5661	5661	5661
◆ V. mem. lat = 0 cycles	6783	3406	1714	868	448	238	130	76	49	35	30
■ V. mem. lat = 20 cycles	9660	4846	2434	1228	630	331	176	99	60	41	32
△ V. mem. lat =40 cycles	1253	6286	3154	1588	811	423	222	122	72	47	35
× V. mem. lat = 60 cycles	1541	7726	3874	1948	992	515	268	145	83	53	38

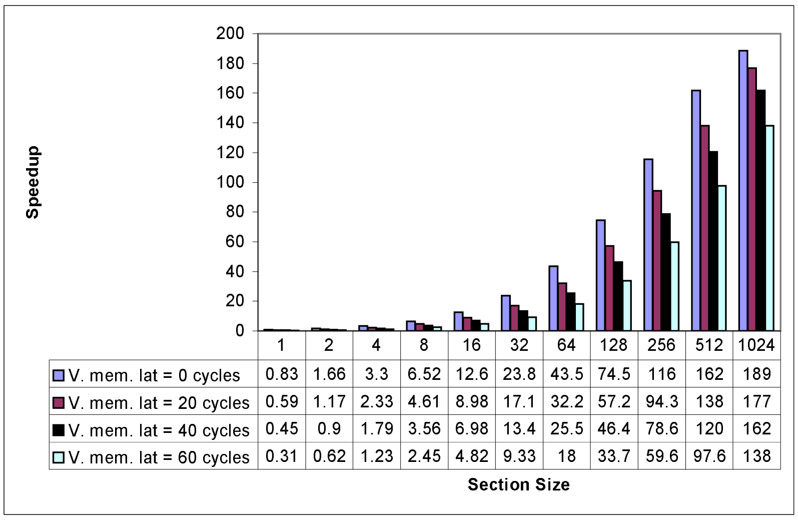
Figure 4.21: Digital filter with loop interchange: Execution time vs. section size

Figure 4.23 shows the speedup of the vectorized interchanged implementation over the vectorized original implementation considering a vector memory latency of 20 cycles, a value close to the scalar memory latency of 18 cycles.

This small example illustrates how coding style can influence vector performance. A very simple optimization, loop interchange, has reduced execution time of the linear filter by over 65 times.

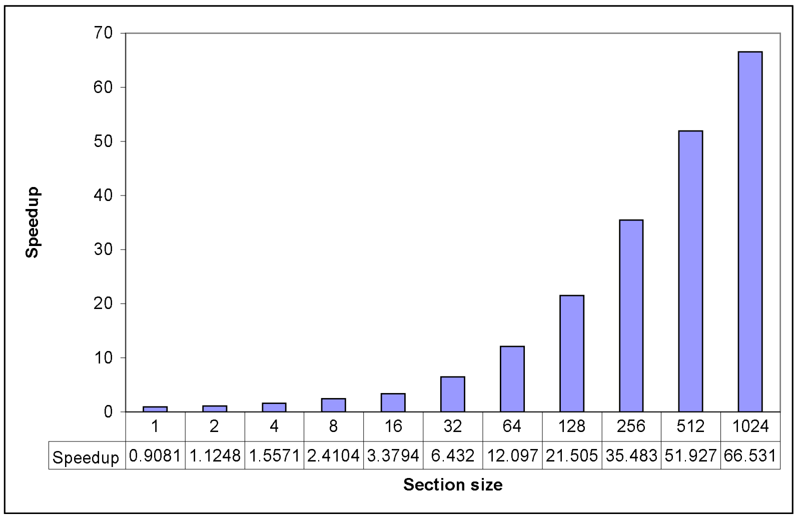
4.3.5 Sound processing

For sound processing we have taken a look at the MAD audio codec. The code profiling looks like this (Table 4.7):



	1	2	4	8	16	32	64	128	256	512	1024
V. mem. lat = 0 cycles	0.83	1.66	3.3	6.52	12.6	23.8	43.5	74.5	116	162	189
V. mem. lat = 20 cycles	0.59	1.17	2.33	4.61	8.98	17.1	32.2	57.2	94.3	138	177
V. mem. lat = 40 cycles	0.45	0.9	1.79	3.56	6.98	13.4	25.5	46.4	78.6	120	162
V. mem. lat = 60 cycles	0.31	0.62	1.23	2.45	4.82	9.33	18	33.7	59.6	97.6	138

Figure 4.22: Digital filter, basic implementation after loop interchange: Speedup for various section sizes



	1	2	4	8	16	32	64	128	256	512	1024
Speedup	0.9081	1.1248	1.5571	2.4104	3.3794	6.432	12.097	21.505	35.483	51.927	66.531

Figure 4.23: Digital filter: Interchanged implementation vs. the original one

Profiling shows that there are some kernels that might be vectorized. However, at a

Self	Name
36.4%	synth_full
27.3%	mad_layer_ii
18.3%	ii_samples
18.2%	mad_bit_read

Table 4.7: Profiling information for the MAD audio codec

close inspection of the code our hopes have been blown up.

The application is written with heavy optimizations in mind, most of the loops are totally unrolled up to the point there are no more loops.

For example, function synth_full calculates 64 coefficients used at a later point. We would expect to see a for loop that for each iteration calculates one coefficient. This is not the case. Each of these coefficients has its own calculus expression written in the code, without any loop. There are hundreds of lines of code that calculate each of the coefficients individually.

The code also uses extensively macros. Summing up, this implementation of MAD audio codec is very difficult to vectorize. We have identified only one function that can be efficiently vectorized. Unfortunately, this function only takes up 4.43% of the total execution time. We have decided that the effort to vectorize this application is not worth the results and we did not proceed.

4.4 Summary

We have started the chapter by describing the vectorization process and by seeing how coding style can significantly influence vector performance.

Then we have presented the simulation environment, the additions we have made to the SimpleScalar simulator: the new instructions, the new functional units and register files.

The most important part of the chapter was dedicated to the experimental results. We have presented the execution time of the vectorized applications and also the speedups compared to the scalar code for various simulation conditions.

The following pages we shall summarize the influence on performance of the following aspects: architectural and microarchitectural features, data types, data structures and tools. For each of the discussed elements we discuss both software and hardware implications.

Architecture and microarchitecture

We shall start our summary by discussing the influence of the vector memory latency and of the section size on the execution time.

We have seen that longer section size translates into fewer instructions needed to execute a loop but, each of these instructions takes longer to execute. However each instruction has a startup latency and, fewer instructions executed translates into less latency and thus faster program execution.

For computation instruction, the startup latency is around several cycles (two to four) while the memory latency is substantially larger, in the order of tens of processor cycles. Thus, the memory latency strongly dominates the overall instruction startup latency.

A big disadvantage of the classic vector processors is the memory system which must provide a very high memory bandwidth at the cost of high latency. There are two ways to cope with this:

- Hiding the memory latency by overlapping memory loads with computation instructions

	Actual vector lengths (average values)	Possible vector lengths
MPEG2 Encoder	16	frame width
MPEG2 Decoder	16	frame width
Median Filter	9	picture width
Optimized Median Filter	picture width	picture width
Bio Informatics App.	very small	Hundreds of elements
MAD codec	very small	64
Linpack	Width of the matrix	width of the matrix

Table 4.8: Vector Lengths of selected applications

- Incurring the memory latency as rarely as possible by having as few instructions as possible thus making them as long as possible

The first method needs many independent instructions to be able to reorder them. This can be achieved by providing support the microarchitectural level for exploiting instruction and thread level parallelism. Both techniques have been extensively used in scalar processors and we have seen that adaptation to the vector paradigm slightly increase on chip routing complexity but may substantially increase chip area by replicating the already large vector register files.

On the software side, most applications that exhibit enough data level parallelism to be worth vectorizing also have large amounts of instruction and thread level parallelism. Thus, for this approach, software does not need to change in any way.

The second method of coping with the long latencies requires long vectors. Let us take a look at vector lengths in the programs we have analyzed (Table 4.8). In the second column we have the average vector length used by the implementation we have used. In the last column we have the largest dimension of the data structured used. If the computation is symmetric, the largest dimension gives the largest possible vector length.

The MPEG2 standard works on blocks of 8x8 pixels. For the implementation we have analyzed, had average vector lengths of 16 elements. Multiple blocks can be processed in parallel so that by slightly rewriting the implementation vector lengths can be substantially increased and thus, vector performance.

The median filter calculates the value of each pixel in the new image based on 9 pixels of the old picture. For the first implementation of the median filter the vector length was only as big as the filter's window size. With a small optimization, just two loops interchanged, the vector length became equal to the width of the image.

Also the bio informatics applications process data in such a way that, in the end, for the innermost loops yields very short vector lengths. With some minor rearrangement the vector lengths can be improved in this case as well.

The MAD audio codec calculates 64 coefficients. However, this calculation is done without even having a loop, each coefficient has his own formula. It is like the loop has been unrolled 64 times up to the point where it has disappeared entirely. For a scalar processor this is beneficial because there is no loop overhead, however it has lost the possibility to be vectorized.

Linpack is based on matrix multiplication that inherently has vector lengths equal to the dimensions of the matrix, therefore Linpack offers very good results when vectorized.

By looking at the previous table we can easily see that the way current applications are written they don't offer vector lengths enough to flex the muscles of traditional vector processors with section sizes of 64 and larger. For long section sizes the registers will hold only few useful elements, the rest will simply not be used.

For processors with long section sizes we must avoid executing dead cycles. Usually a vector instruction executes an operation on all elements of a vector register. It is vital for performance that operations be performed only on those elements that contain useful information. In other words, when the vector length is shorter than the section size, execute only vector length operations and not section size operations. This is especially useful when the section size is significantly larger than the used vector length.

Another approach is to design the vector processor with short section sizes and optimize for this aspect. This is the approach used by the Tarantula vector microprocessor [3]. They have also reached the conclusion that for general purpose applications shorter vector lengths are much more common than long ones. They are in favor of building a vector machine with short vectors but with many registers.

Traditional vector processors transfer huge amounts of data in and out of memory because they can process all this data. Transferring several full vector registers can easily fill up the cache therefore vector memory accesses bypass the cache, going directly into main memory. As a consequence, the vector memory system must provide huge bandwidths at the cost of the high latency already mentioned. Since current applications do not use data in such big chunks, caches might become viable. This is a question that remains to be answered in future work. If this where the case, caches mean lower memory latencies that in turn supports the idea of designing the vector processor with short section sizes (remember the discussion on memory latency and section size).

The software can clearly be written in such a way to allow longer vector lengths. In most cases it is only necessary to reorder the loops. Loop interchanges can make a substantial difference.

Let us take a look at data organized in a matrix with its height much larger than its width. Accessing it the standard way the outer loop goes for the rows and the inner one iterates along the column would yield short vector sizes because the number of columns is smaller than the number of rows. In contrast if the outer loop iterates on the columns and the inner loop iterates on the rows we obtain longer vector lengths. Most of the times, the adaptation from the first type of access to the second one is relatively easy to implement.

In other cases, problems are subdivided into smaller ones like in the mpeg2 where the image is divided into blocks and then the blocks are processed one at a time. If the blocks are independent of each other than the algorithm can be rewritten in a way to process multiple blocks in parallel thus increasing the possible vector lengths.

Data Types

Vector processors where initially built for scientific applications where all data was stored and processed in double precision floating point format. Many of the applications we have analyzed use integer data types. These range from a single byte to 4 bytes.

For vector processors running day to day application is imperative to have support for both floating point and integer arithmetic. The use of integer arithmetic imposes some diversity. There must be support for multiple element lengths corresponding to standard data types like for instance in C:

- 1 byte char
- 2 bytes short
- 4 bytes long
- 8 bytes long long

There must also be support for signed or unsigned operation. The difference is when sign extending. There must be different instructions for loading and storing these types of data.

Two independent hardware decisions need to be made. First, we must decide whether to use dedicated vector registers files for floating point and separate, dedicated, register file for integers or use a single vector register file that can contain both integers and floating point data. Vector register files are quite big, they occupy a lot of area. From this point of view a single unified register file sounds better. However, a single vector register file that contains both integer and floating point data must have some data type checks enforced to prevent treating floating point data as integer data or vice-versa. This type check can be done in hardware or leave the responsibility to the programmer.

Applications rarely use both integer and floating point computation at the same time. All of our analyzed programs use intensively just one type of data type. From this point of view there would be no inconvenient if we use a common register file.

The second decision is whether to use one vector register element for each array element or pack multiple shorter integer data types into a single register element. For instance an array of chars can occupy a register element for each array element or eight chars can be packed into a 64 bit register element in a manner similar to that used for multimedia extensions.

The latter approach practically increases the section size for smaller element types with a small overhead. In addition, the arithmetic operations must specify the data type they operate on. In this direction, many lessons can be learned from the multimedia extensions from scalar processors [5].

On the software side, there are no limitations on the data types to ease the transition of software to the vector processor. The only restriction is a limited use of pointers. A vectorizing compiler looks for dependencies between the iterations of loops. If there are no dependencies, then the loop can be vectorized. However, pointers greatly complicate the dependencies analysis. Since it is very hard to track where a pointer points to, generally compilers totally avoid any optimizations when pointers are involved including the case of a dependency decision. This may reduce the amount of vectorization attained.

In almost any cases the use of pointers can be avoided. They can simplify algorithms in some cases but in others they can rend optimizations impossible to perform. As a general idea, vector processors work with regularity. Pointers can point to any memory address and are big sources of code irregularity thus, the excessive use of pointers can significantly reduce vector performance.

Data Structures

Vector machines handle well data that is regularly arranged in memory. Arrays and arrays of structs are well supported. However, linked lists are totally undesirable.

We have already seen that pointers can degrade vector performance and linked lists are based on pointers. The great advantage of arrays is that by knowing the base address and the index we can calculate at any time the address of any element and even more, we can read in one memory access a big number of consecutive elements of an array. In contrast, when we want to access the elements of a linked list we have to do it sequentially: read the first element to obtain the address of the next element and so on. Using this scheme there is no more parallelism. We could work around this by packing the elements of a linked list into an array, process it vectorially and then unpacking the resulting array to the original linked list. It would generate too much overhead. Another method would be to have an array that contains the memory addresses where the linked list elements can be found. This would allow the use of indexed loads and stores to eliminate the packing and unpacking.

Hardware handles best memory accesses at consecutive addresses. We look again at the implementation of the digital median filter from the chapter dedicated to the introduction to vectorization. The color of a pixel is stored as three values: red, green and blue. It is preferably to use three separate arrays for each of the colors rather than use an array of structs. Accessing one color of the picture would mean in the first case consecutive accesses while in the second case would mean strided accesses.

Tools

The success of vector processors is highly dependent on the existence of a very powerful vectorizing compiler. Transforming a traditional programming language like C or Fortran into vector instructions is no easy task.

Vectorizing the code by hand is no longer an option. Of course the writing directly in assembly language creates code that runs faster than any code produced by a compiler, however, the sheer size of the current applications makes manual vectorization impossible.

Libraries with high performance code must be created by professionals and used in applications. The vectorizing compiler must also be able to re-target existing applications.

5 Conclusions

This thesis presented the high level concept of vector processor: operation on entire arrays of data. We have continued with a short history of vector processors. The first chapter has concluded with the description of a general vector architecture with the typical instruction set, and the specific processing techniques. After the basics had been presented and described we have continued with an in-depth analysis of vector processors. We have considered both architectural and micro-architectural issues.

Several micro-architectural choices are dictated by the architectural features. This is the case of the vector register file. The number of registers and their size is fixed by the architecture while the organization in banks (if any), number of read/write ports, timing and so on are totally transparent to the architecture and thus, implementation dependent. The vector functional units have a similar construction. While the architecture does not explicitly state the type of functional units required, the instruction set implicitly determines the needed functionality. However, the organization of the functional units, the number of functional units and many other parameters are implementation dependent and not influenced in any way by the architecture. Chapter two has described in detail the vector processor's components that are subject to common influences from both the architecture and the micro-architecture.

Other components of the vector processor, such as the memory system and the control unit, are completely transparent to the architecture. Chapter three has been dedicated to these elements that are exclusively shaped by micro-architectural decisions. More specifically, we have analyzed the characteristics of the vector memory system by looking at the memory access types and ways to efficiently support them. We have looked at the tradeoff between memory bandwidth and latency, at the traditional approach to the memory system and also at a more recent approach to vector memory system that uses caches.

We have then presented the control part of the vector unit. We have focused on increasing the occupancy rate of the vector functional units. We have analyzed requirements of the issue logic and then we have presented three methods of rearranging instructions to better hide memory latency and to avoid stalls due to data dependencies. More specifically, we have exploited all levels of parallelism: data level parallelism (inherently present in vector processors), instruction level parallelism (out-of-order execution) and thread level parallelism (multithreading).

The goal of this thesis was to analyze the potentials of media and signal processing applications for acceleration by vector processors. To this extent, we have selected four applications: an MPEG2 encoder, an MPEG2 decoder, a median filter used in image processing and an implementation of the MAD audio codec. We have profiled them, identified computationally intensive kernels and explored them for vectorization potentials. We have reached a level of vectorization of 85% for the MPEG2 encoder, 30%

for the MPEG2 decoder and 80% for the digital image filter. We could only vectorize about 7% of the MAD audio codec. This is mainly because the code was written in such a way that could not be manually vectorized.

An important objective was to evaluate the impact on performance of various architectural and microarchitectural parameters such as the vector register length, the number of vector registers, the number of functional units, the number of memory units, the vector memory latency and bandwidth. We have modified SimpleScalar 3.0 by adding vector instructions, vector register files, vector functional units and vector type main memory accesses. We have run multiple simulations by varying the aforementioned architectural and microarchitectural parameters. Simulation results indicate that through vectorization we can obtain kernel speedups ranging from 5.36x to 14.84x and application speedups of 1.82x and 1.25x for the MPEG2 encoder and decoder respectively. The image filter has reached the highest speedups, the vectorized version being up to 180 times faster than the scalar implementation.

Based on our experiments and analysis we belive the optimal vector processor organization is a multi-lane vector unit tailored for short vectors and low memory latencies. Regarding the particular design features we have considered in this thesis, we can summarize them in the following observations and conclusions:

Vector memory latency can seriously degrade performance. We have seen two methods of reducing the impact of the latency: long vector registers amortize the latency over many elements thus, longer section size translates into less performance degradation caused by latency. The second approach uses instruction reordering such that the memory latency is hidden by performing loads in advance; when data is needed for computation it is already in a vector register. To this extent, we have presented the out-of-order and multithreaded vector processor organizations.

By doubling the **section size**, the performance increases at a sub-linear rate. This is because only the vectorized part of the application runs faster, there is a break even point after which the scalar instructions dominate the execution time. Thus, even for infinite vector lengths, there is this point when doubling the section size increases chip area much more then it increases performance. The simulations we have performed indicate that the optimal section size is 16 or 32. Also, software that we have analyzed offer short vectors mostly from 8 to 64 elements and rarely over one hundred elements. Therefore we recommend short vector registers and a microarchitecture that emphases on high functional unit usage rates by exploiting both instruction and thread level parallelism.

Organization of the datapath in multiple lanes improves performance almost linearly with the number of lanes. Functional units must be replicated for each of the lanes but the organization of the register file is greatly simplified because the big, monolithic register file is split into smaller ones by partitioning in banks, one bank for each lane. The chip area is increased by replicating the functional units but the complexity and delay of the wiring logic is reduced because data is processed closer to the register banks.

Multiple lanes improve performance over the single lane organization in a similar way as processing vectors over processing scalars. By adding more lanes the computation time is reduced linearly with the number of lanes, but the functional unit startup latency remains constant. However, for two, four or eight lanes the computation time is significantly larger than the startup latency and we can consider that performance increases by a factor of 2, 4 and 8 respectively.

High memory bandwidth is required for a multi-lane datapath. Each memory unit should be able to transfer per cycle as many words as a functional unit can process. For instance, for four lanes, the memory bandwidth should be four times the bandwidth of a single lane organization. In chapter 3 we have seen that, for utmost performance, the vector processor should have a memory to compute ratio of 3:2. That means that for every two functional units there should be 3 memory units. The statistics of our simulations indicate that, on average, the number of vector loads is twice the number of vector stores and that the number of vector additions is roughly equal to the number of vector multiplications and vector stores. Thus, *we propose a vector unit that uses two load units, one store unit, a vector ALU and a vector multiplier.*

Data types. Traditionally, vector processors were built with scientific calculations in mind and provided vector support mainly for floating point data. However, media applications extensively use integer data types therefore, support for integer vector register files and vector functional units is an important requirement for media vector processors.

Data alignment. Vector processors work best with data arranged regularly in memory. Unit stride accesses are the fastest memory accesses followed by the non unit stride accesses and indexed accesses. Therefore, it is desirable for software to use data structures that can be accessed through unit or non-unit stride accesses and avoid using vector indexed accesses. Thus, arrays are preferred to arrays of structs and the use of pointers and irregular data structures randomly distributed in memory (such as linked lists) should be avoided.

Vectorization and compilation tools are crucial in performance gains through vectorization. A powerful vectorizing compiler and adequate libraries are one of the first prerequisites for executing applications on vector processors. Without properly vectorized code even a very high performance vector processor offers poor results.

Future directions. In this thesis we have shown that vector processors can significantly reduce execution time for applications with a high amount of data level parallelism. There are many architectural and microarchitectural features that can be further explored. In future research, we wish to develop a more accurate vector processor simulator that can allow us to model more precisely the architectural and microarchitectural parameters. More specifically, we wish to test a new architecture of the vector register file that will allow a significant increase in flexibility through reconfigurability. Another important future goal is to extensively investigate the vector memory system and ways to adapt caches to changing memory access patterns.

Bibliography

[1] *Ddr2 sdram documentation* `http://download.micron.com/pdf/datasheets/dram/ddr2/512`

[2] *Vector microprocessors*, Ph.D. thesis, University of California at Berkley, 1998.

[3] K. Asanovic and D. Johnson, *Torrent architecture manual, revision 2.11*, Tech. report, Computer Science Division, University of California at Berkley, 1997.

[4] B. Spinean C. Ciobanu, *Simultaneous multithreading vs. chip multiprocessing*, Summer Summer 2006.

[5] Catalin Ciobanu, *Vector processor architecture - specialized instructions sets*, September 2006.

[6] N. Cardwell R. Fromm K. Keeton C. Kozyrakis R. Thomas K. Yelick D. Patterson, T. Anderson, *A case for intelligent ram: Iram*, IEEE Micro (1997).

[7] F. M. Tomasulo D. W. Anderson, F. J. Sparacio, *The ibm system/360 model 91: Machine philosophy and instruction handling*, January 1967.

[8] Roger Espasa, *Jinks: a parametrizable simulator for vector arcihtectures*, Tech. report, 1995.

[9] ______, *Advanced vector architectures*, Ph.D. thesis, Universidad Politechnica de Catalunya, February 1997.

[10] J. W. C. Fu and J. H. Patel, *Data prefetching multiprocessor cache memories*, In proceedings of the 18th ISCA (1991), 54–63.

[11] J. D. Gee and A. J. Smith, *Vector processors caches*, Tech. report, University of California at Berkley, September 1992.

[12] S. Nagamine Y. Mochizuki A. Nishizawa H. Hirata, K. Kimura, *An elementary processor architecture with simultaneous instruction issuing from multiple threads*, In proceedings of the 19th Annual International Symposium on Computer Architecture (1992), 136–145.

[13] R. G. Hintz and D. P. Tate, *Control data star-100 processor design*, In proceedings of COPCON (1972), 1–4.

[14] Allan J. Smith Jeffery D. Gee, *The performance impact of vector processor caches*, in proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences (1992).

[15] David A. Patterson John L. Hennessy, *Computer architecture - a quantitative approach*, Morgan Kaufman Publishers, 2002.

[16] C. Lee, *Code optimizers and register organizations for vector architectures*, Ph.D. thesis, University of California at Berkley, May 1992.

[17] T. Mudge R. Brown M. Upton, T. Huff, *Resource allocation in high clock rate processors*, Proceedings of the sixth international conference on Architectural support for programming languages and operating systems (1994), 98 – 109.

[18] Behrooz Parhami, *Computer arithmetic - algorithms and hardware designs*, Oxford University Press, 2000.

[19] James E. Smith Roger Espasa, Mateo Valero, *Out-of-order vector architectures*, In the proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (1997).

[20] Mateo Valero Roger Espasa, *Decoupled vector architecture*, In proceedings of the 2nd Internaltional Symposium on High Performance Computer Arhchitecture (1996), 281–290.

[21] ———, *Instruction level characterization of the perfect club programs on a vector computer*, In XV International Conference of the Chilean Computation Society (1997), 198–209.

[22] James E. Smith, *Decoupled acess/execute computer architectures*, In proceedings of 9th Annual International Symposium on Computer ARchitectures (1982), 112 – 119.

[23] R. M. Tomasulo, *An efficient algorithm for exploiting multiple arithmetic units*, Tech. report, IBM Journal of Research and Development, 11:25-33, January 1967.

[24] S. Vajpeyam, *Instruction-level characteriaztion of the cray y-mp processor*, Ph.D. thesis, University of Wisconsin-Madison, 1991.

[25] J. E. Smith W. C. Hsu, *Performance of cached dram organizations in vector supercomputers*, In proceedings of the 20th Annual International Symposium on Computer Architecture (1993), 327–336.

[26] David W. Wall, *Limits of instruction level parallelism*, in 4th International Conference on Architectural Support for Programming Languages and Operating Systems (1991), 176–188.

[27] Edward S. Davidson William Magione-Smith, Santosh G. Abraham, *Vector register design for polycyclic vector scheduling*, ACM SIGPLAN Notices archive, Volume 26 (1991), 154 – 163.