Rapid Evaluation of Requirements for Vector Micro-Architectures

Milan Stanic

Barcelona School of Informatics (FIB)

Universitat Politècnica de Catalunya

A thesis submitted for the degree of

Master of Information Technology

21th of September, 2011

Advisor: Adrian Cristal

Tutor: Alejandro Ramirez Bellido

Abstract

Power consumption has become one of the dominant issues in processor design, especially important in embedded systems and data centers. One of possible solution that can address this issue and provide higher performance for existing applications and new capabilities for future applications used in hand-held devices and data centers is to use vector processor.

This thesis presents the design and implementation of a vector library that enables the vectorization of the target applications and allows to characterize them.

We also present the ETModel: a simple trace-driven simulator for vector processors. It is used to analyse the micro-architectural requirements of the vectorized applications.

We show that the target applications are highly vectorizable with a degree of vectorization from 62.9% for H264ref to 91% for ECLAT. Detailed instruction level characteristics such as the distribution of vector instructions, the distribution of vector lengths, etc. are also presented in the thesis.

The thesis contains detailed timing analysis of the vectorized applications for different micro-architectural configurations of a vector processor. We measured the execution time for the different configurations of cache hierarchy, main memory latencies, maximum vector lengths and configuration of functional units, as well as the usage of functional units. All these help in understanding the behavior of the vectorized applications and requirements of vector micro-architecture.

Acknowledgements

I would like to thank my supervisors and tutor, for their guidance of this thesis and their support. I am especially grateful to Oscar Palomar for many insightful discussions and for his patience in reading my manuscripts. I would also like to thank all my colleagues in the Microsoft Research Center for their unselfish support.

Special thanks go to my parents, Zivorad and Svetlana, my sister Marija, my girlfriend Suzana and all my family for their love and support. I would not be here today without their dedication and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Work Presentation

1.1 Motivation

Over the last few years, power consumption has become one of the dominant issues in processor design. This issue is especially important in embedded systems such as cellular phones, pagers, PDAs, digital cameras, DVDs, game consoles, etc. in which battery life becomes a major concern, but also in tremendous data centers that consume a large amount of power. At the same time, a new design should provide higher performance for existing applications and new capabilities for future applications that will be used in hand-held devices or data centers.

Vector processors are one possible solution to address this issue because they can express data level parallelism where it exists in a very efficient way. They fetch fewer instructions and therefore reduce the fetch and decode bandwidth requirements.

In order to help us to define a new vector architecture, we should discover which are the characteristics at the instructions level of the target applications (once they have been vectorized) (e.g. degree of vectorization in application, distribution of vector lengths, distribution of instruction types, etc.) and also which new vector instructions we will need in order to vectorize these applications in an efficient way. We also want to know which will be the micro-architectural requirements for implementing such a vector ISA and estimate the execution time of vectorized applications.

Some of these questions are addressed by Espasa [15, 40] using traces from vectorized CONVEX binaries and simulating them. Currently we do not have access to any compiler that performs automatic vectorization of our target applications. To overcome this problem, in this project we have chosen to develop a vector library and a model for execution time that addresses the issues mentioned above.

1.2 Project Objectives

The main goals of the project are:

- To develop a vector library that implements a vector ISA similar to VMIPS[1] [20]. The vector library will also contain some additional instructions that are useful or required to vectorize our target applications. The vector library will be parameterizable: size of vector register file, register length, etc. The library will collect results and statistics at runtime from vectorized applications. It will also provide support to generate instruction and address traces of the vectorized applications to allow further analysis (e.g. a model for execution time). The vector library will also provide support to allow the vectorization of applications written in C, C++ and FORTRAN.

- To choose several modern applications that can be used in handheld devices or data centers, profile them and vectorize them by hand if they are suitable for vectorization. Another sub-goal is to generate statistics that will provide information about instruction level characterization of these applications. These statistics will be generated through traces that will feed the model for execution time.

- To develop a trace-driven model for execution time (ETModel), motivated by work presented by Karkhanis & Smith [23] and Hennessy & Patterson [20]. The model will be parameterizable (e.g. number of lanes, number of ALU units, LD/ST units, start-up latencies, memory bandwidth, etc.).

[1] RISC-like a vector ISA and register based

- To analyze the results. Gathered results and statistics will be used to analyze the micro-architectural requirements to implement the proposed vector ISA. In particular, we wish to study the impact of the parameters of the model on an application's execution time.

1.3 Project Objectives

The following table summarizes the distribution of the time used to develop the different activities in this project.

#	Activity	Time (hours)
1	Vector library implementation.	240
2	Benchmark vectorization.	320
3	ETModel implementation and testing.	200
4	Results collecting.	120
5	Results analyzing.	40
6	Final report.	100
Total		1020

Table 1.1: Project roadmap activities.

Chapter 2

Vector Processors

This chapter presents vector processors in general, the advantages of vector processors and vector instruction set architecture (ISA), reviews existing implementations and gives definitions of some terms used in the rest of the thesis. Patterson & Hennessy [20] gives more details about vector processors as well as Asanovic [3].

Various forms of parallelism have been exploited in computer architecture to provide increases in performance. The three major categories are: **instruction-level parallelism** (ILP), **thread-level parallelism** (TLP) and **data-level parallelism** (DLP). ILP allows simultaneous execution of multiple instructions from one instruction stream (superscalar processors). TLP allows simultaneous execution of multiple instruction streams (Simultaneous Multi Threading - SMT). DLP allows simultaneous execution of the same operations on arrays of elements (Single Instruction Multiple Data - SIMD).

A vector processor implements a type of data-level parallelism. Vector processors typically contain vector registers that hold multiple values instead of single-value registers as in super-scalar processors. They provide vector instructions that operate on all values of the registers, in conceptually simultaneous manner. For example, a scalar addition instruction would take values from two scalar registers A and B, and produce a result stored in scalar register C, as figure 2.1 (a) shows. A vector addition instruction would take two vectors A and B, and produce a resulting vector C, as in figure 2.1 (b), where VL is vector length.

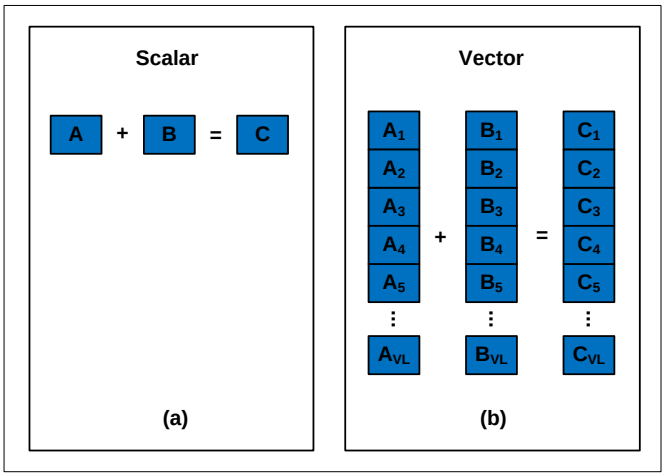

Figure 2.1: Comparison of a scalar instruction and a vector instruction.

2.1 Advantages of Vector Processors

As it is emphasized in previous work [3, 20, 22], vector processors and vector ISAs have several advantages:

- A single vector instruction specifies N operations, where N represents tens or hundreds of operations. It dramatically reduces instruction fetch bandwidth, which is a bottleneck of conventional processors, particularly in terms of power consumption [29, 42].
- These N operations are independent. There is no need for checking data hazards within a vector instruction. It allows simultaneously execution of all operations in an array of parallel functional units, or in a single very deeply pipelined functional unit, or in any intermediate configuration of parallel and pipelined functional units.
- Reduced control logic complexity. Hardware needs only check for data hazards between two vector instructions once per vector operand, not once for every element within the vectors. Therefore, the dependency checking logic required between two vector instructions is approximately the same as that required between two scalar instructions, but now many more elemental operations can be in flight.

- Vector instructions that access memory have a known access pattern. A memory system can implement important optimizations if it has accurate information on the address stream. In particular, a stream of unit-stride accesses can be performed very efficiently using large block transfer. Also in case main memory accessed, the high latency of initiating access versus accessing a cache is amortized, because a single access is initiated for entire vector rather than to a single word.

- Reduced control hazards from loops, because an entire loop can be replaced by a vector instruction whose behaviour is predetermined.

2.2 Relevant techniques and concepts

In this section, we describe techniques and concepts that have been used in vector architectures, relevant for this thesis.

2.2.1 Chaining

Some vector architectures have to complete a vector instruction before starting the next vector instruction. Chaining is a technique that allows overlapped execution of two dependent instructions. It means that next vector instruction can start execution before current vector instruction is completed. Consider the simple vector sequence:

```
addv R1, R2, R3
mulv R4, R1, R5
```

We want to add vector registers R2 and R3 and to store results into vector register R1. After that, we multiply vector registers R1 and R5 and store result into vector register R4. Figure 2.2 shows the timing of chained and an unchained version of the above pair of vector instructions with a vector length of N. In an unchained version two vector instructions are computed serially. We have the start-up time to compute the first element in the first vector instruction and then n cycles to compute the whole vector and then the same for second vector instruction. In a chained version, second vector starts execution when the first

element in the first vector instruction is computed. Generally, chaining allows a vector operation to start as soon as the individual elements of its vector source operand become available.

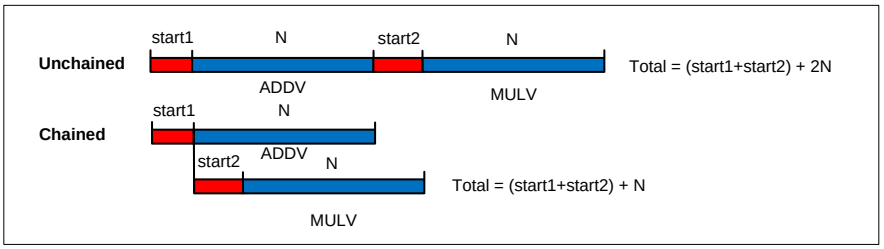

Figure 2.2: Timings for a sequence of dependent vector instructions ADDV and MULV, both unchained and chained.

It is obvious that chaining plays an important role in boosting vector performance. In fact, chaining is so important that every modern vector processor supports chaining [20].

2.2.2 Multiple Lanes

As mentioned above, a vector instruction specifies a number of independent operations that can be executed in parallel. This semantics of a vector instruction allows using an array of parallel functional units, or a single very deeply pipelined functional unit, or any intermediate configuration of parallel and pipelined functional units. Vector performance can be improved by using parallel and pipelined units. For example, figure 2.3 (a) shows a vector unit that has a single pipeline and can complete one addition per cycle. The figure 2.3 (b) shows a vector unit that has four add pipelines and can complete four additions per cycle. The elements within a single vector add instruction are interleaved across the four pipelines.

The construction of parallel vector unit is simple because all vector arithmetic instructions only allow element N of one vector register to take part in operations with element N from other vector registers. Parallel vector unit can be structured as multiple parallel *lanes*. Patterson and Hennessey [20] give one example of a four lane vector unit (figure 2.4). The vector-register storage is divided across

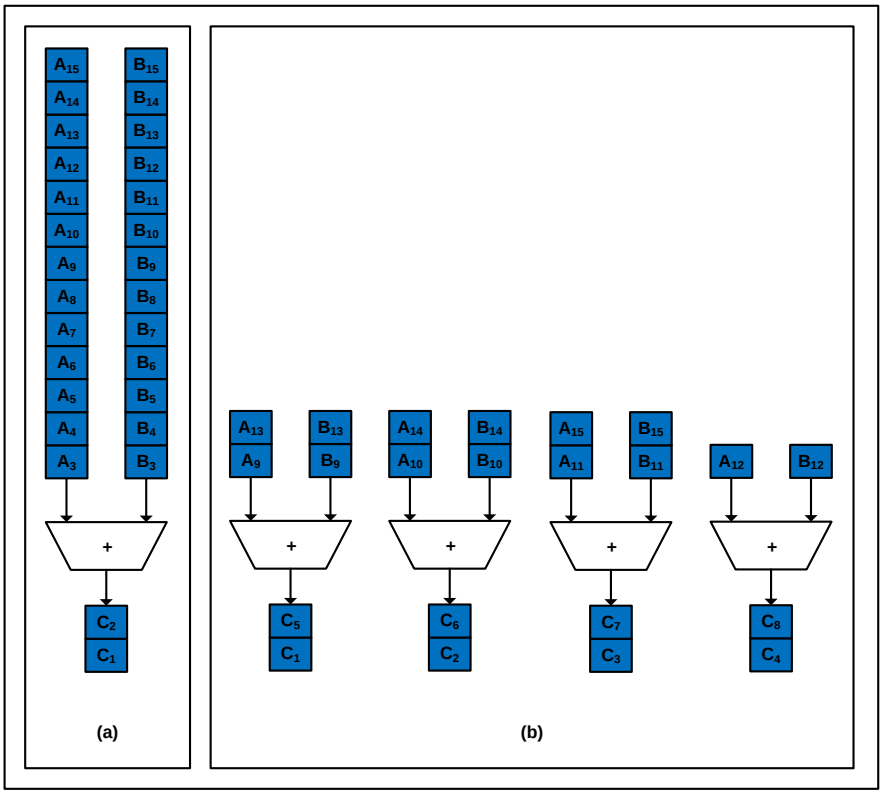

Figure 2.3: Using multiple functional units to improve performance of a single vector add instruction, C = A + B.

the lanes, with each lane holding every fourth element of each vector register. There are three vector functional units shown, an FP add, an FP multiply, and a load-store unit. Each of the vector arithmetic units contains four execution pipelines, one per lane, that act in concern to complete a single vector instruction. Implementation of instructions that require communication across lanes is more complex.

2.2.3 Death time or recovery time

Adding multiple lanes increases performance, but still there are start-up overhead and power consumption concerns. It is possible to reduce start-up overhead allowing the start of one instruction to be overlapped with the completion of preceding vector instructions. It increases the complexity of control logic and

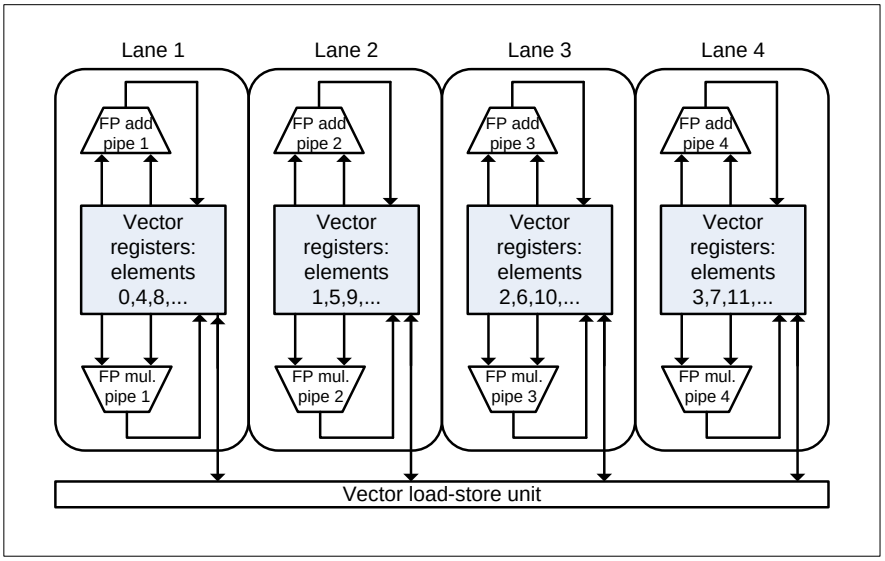

Figure 2.4: Structure of a vector unit containing four lanes.

some vector machines require some *recovery time* or *death time* in between two vector instructions dispatched to the same vector unit.

2.2.4 Masking

Programs that contain conditional (*if*) statements cannot be vectorized using the basic memory, arithmetic and logical instructions which are sufficient for vectorizing many straight-line loops. Consider the following loop:

```
for( i = 1, i <= looplen, i++){
   if (a[i] == b[i])
           c[i] = a[i] + b[i];
}
```

This code cannot normally be vectorized because of the conditional execution of the body. J. E. Smith et al [37] examined a number of vector instruction set alternatives for implementing conditional loops. The paper concludes that "the best approach is to use masked instructions. Masked instruction uses a Boolean vector of maximum vector length (MVL) to control the execution of a vector instruction just as conditionally executed instructions use a Boolean condition to determine whether an instruction is executed. When the vector-mask register

is enabled, any vector instructions executed operate only on the vector elements whose corresponding entries in the vector-mask register are 1".

Masking allows higher level of vectorization and it is critical in achieving the large difference between vector and scalar mode [20].

2.3 Existing implementations

Vector processors have a long and successful history in supercomputers where they are used for large scientific and engineering applications. The first vector architecture were memory based with instructions that operate on memory-resident vectors [21, 41]. Cray [35], register-based vector machines were the first commercially successful supercomputers [15]. They provide arithmetic instructions that operate on vector registers, while separate vector load and store instructions move data between vector registers and memory. Several modest mini-supercomputers [32, 33] were released in the mid 80s.

Vector processors have found their place in microprocessors. Vector microprocessors have been constructed to support vector instructions [4, 31]. Torrent-0 and IRAM are an example of vector microprocessors developed as part of academic research. Espasa et al [16] developed Tarantula, a vector extension to the Alpha architecture.

Vector extensions, such as MAX[26], MMX[30], SSE[39], AVX[9], AltiVec[19], 3DNow![28], etc., are very popular in desktop processors of all of the major vendors. These vector extensions do not implement all traditional vector instructions and operate on much shorter vectors (4-8 elements) than in old vector architectures because one of goals is to minimize additional chip area.

Vector processors are also used in special purpose hardware such as video cards and game consoles. One of example is the Sony Playstation II [38].

"Knight's Corner" is an upcoming massively parallel x86 microprocessor designed by Intel Corporation. It is based on the cancelled Larrabee [36] GPU that contains a 512-bit vector processing unit in each core, able to process 16 single precision floating point numbers at a time.

Vector register architecture has several advantages over memory-memory architectures [3]. A vector register architecture reduces temporary storage requirements, memory bandwidth, and inter-instruction latency compared to vector memory-memory architecture because vector register architectures can keep intermediate results while memory-memory architecture has to write all intermediate results to memory and then must read them back from memory. Also if the result of a vector instruction is needed by multiple other vector instructions, a memory-memory architecture must read it from memory multiple times, whereas a vector machine can reuse the value from vector registers, further reducing memory bandwidth requirements. For these reasons, vector register machines have proven more effective in practice. In the rest of this thesis, I restrict the discussion to vector register architectures.

Chapter 3

Vector library

One of the objectives in this thesis is designing and implementing a vector library that will help us to vectorize and analyse target applications. The vector library is implemented in C++ and has the following features:

- It collects results and statistics at runtime from the vectorized application.
- It implements a configurable vector register file.
- It implements vector ISA similar to VMIPS [20] plus extensions.
- It generates instruction and address traces of the vectorized application, that enable further analysis.
- Provides wrapper functions for applications written in C and FORTRAN.

3.1 Configurable vector register file

The vector library implements a configurable vector register file. The number of vector registers and the maximum number of elements per vector register are parameters of the vector library. This allows us to use different configuration of the vector register file and to specify different maximum vector lengths (MVL) of the vector register.

Each register holds a set of values and they can be one of several data types:

- signed or unsigned integer (16, 32 and 64 bits),
- double (floating-point double precision: 64 bits),
- float (floating-point single precision: 32 bits) or
- char (8 bits).

3.2 Vector ISA

The vector library implements a vector ISA similar to VMIPS [20]. Most instructions are RISC-like and register-based. The implemented vector ISA consists of:

- arithmetic and logical instructions,
- memory instructions,
- reduction instructions and
- bit and element manipulation instructions.

Most implemented instructions are usually found in any register-based vector ISA, but there are some not so common instructions that are useful or required to vectorize our target applications. Following subsections give more details.

Tables 3.1 and 3.2 show the vector instructions implemented in the vector library. The first column contains the type of instruction. The second column is the opcode of instruction. Arithmetic and logical instructions have suffix S in opcode (sixth column gives information if a particular arithmetic or logical vector instruction supports vector-scalar mode) if one operand is scalar, or suffix MASK if it is executed over vector mask register, or SMASK if one of the operands is scalar and it is executed over vector register. Third and fourth columns give information related with source and destination vector registers used by vector instruction. Fifth column contains information related with vector mask register. If an instruction contains x in masking column, it means that there are an available instruction that can be executed over the vector mask register (arithmetic

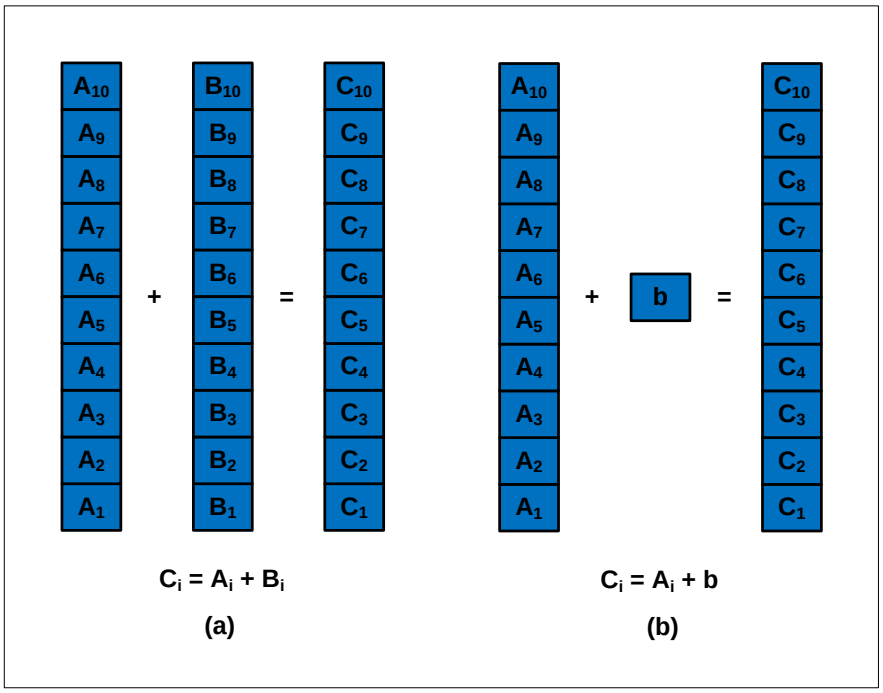

Figure 3.1: Two modes of vector instruction: (a) vector-vector and (b) vector-scalar.

and logical and memory instructions) or one operand is a vector mask register (bit and element manipulation instructions). Availability of an instruction with scalar operand is displayed in the sixth column. The last column contains a short description of the implemented vector instruction.

3.2.1 Arithmetic and Logic Instructions

The vector library implements all common arithmetic and logic instructions such as addition, multiplication, subtraction, logical bitwise operations, etc. Instructions can operate in vector-vector or vector-scalar mode. In vector-vector mode, an arithmetic or logic instruction has two vector source registers and performs arithmetic or logic operations on all elements of vector in a pairwise fashion, as figure 3.1 (a) shows. In vector-scalar mode, an arithmetic or logic instruction has one vector source register and one scalar source value and performs arithmetic or logic operations between all elements of vector register and scalar value, as is

shown in figure 3.1 (b).

The vector library also supports masking. It means that implemented arithmetic or logic instructions can be optionally executed over vector mask register (see Section 2.2.4).

3.2.2 Memory Instructions

Common vector memory instructions such as unit-stride and strided memory instructions as well as indexed memory instructions (scatter and gather) are implemented in the vector library. In unit-stride memory access, consecutive elements are accessed, as in figure 3.2 (a). In strided memory accesses, elements are accessed with a constant stride, as is shown in figure 3.2 (b). With indexed memory access elements accessed randomly using there indices stored in the vector register, as figure 3.2 (c) shows.

Some of these instructions are implemented with support for masking. These instructions are useful in kernels where we have to store or load some elements of a stream depending on some condition. For example, in code below we store only those elements from array b to array c if the corresponding element in array a is greater than constant value *con*.

```
for( i = 1, i <= looplen, i++){
   if (a[i] > con)
           c[i] = b[i];
}
```

If we have memory instructions with support for masking, the code above is vectorized in the following way:

```
ldv VR1, a            // load from array a to
                      // vector register VR1
cmpvsqt VMR, R1, con  // compare vector register VR1 and
                      // scalar value con; store results
                      // in vector mask register VMR
ldvmask VR2, b, VMR   // load over VMR from array b
                      // and store to VR2
stvmask VR2, c, VMR   // store to array c from VR2 over VMR
```

The example above clearly show the importance of memory instructions with support for masking.

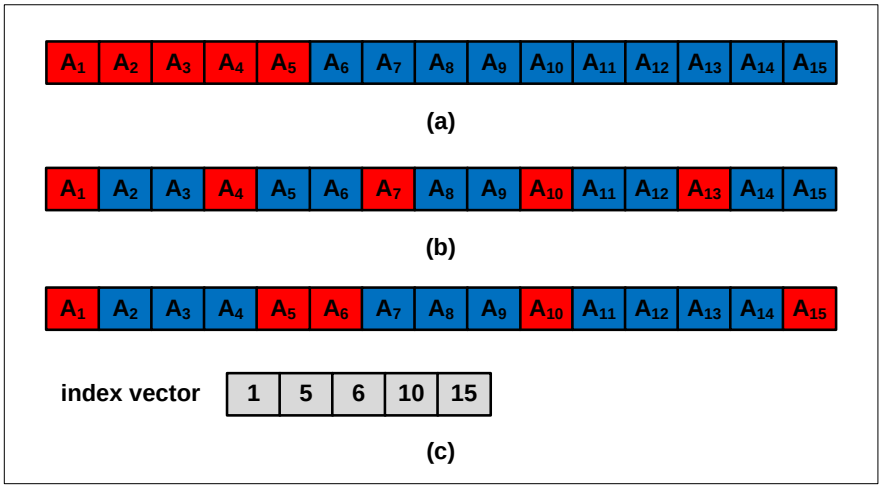

Figure 3.2: Different types of memory instructions: (a) unit-stride, (b) strided, and (c) indexed.

The vector library also implements uncommon memory shape instructions, similar to the one introduced by RSVP [13]. The vector is described by the address of the first element and three scalar values: *stride*, *span* and *skip*. Stride describes the spacing between each loaded/stored element (inclusive of element). Span describes how many elements to access at stride spacing before applying the second-level skip offset. For example, we want to load four elements from the first row, but only every second element (Figure 3.3), then do the same for the second row, etc. Using memory shape load instruction with stride equal 2, span equal 4 and skip equal 3 we can load elements with only one memory instruction.

Memory shape instructions allow vectorization of previously non-vectorized kernels or to increase average vector length of vectorized applications (see Chapter 5) and decrease the number of vector instructions used to vectorize some kernels.

3.2.3 Reduction Instructions

Reductions (such as sum) are often not available in some architectures. The library provides supports for:

- Sum - computes the sum of all elements in a vector register.
- Max - finds maximum element in a vector register.

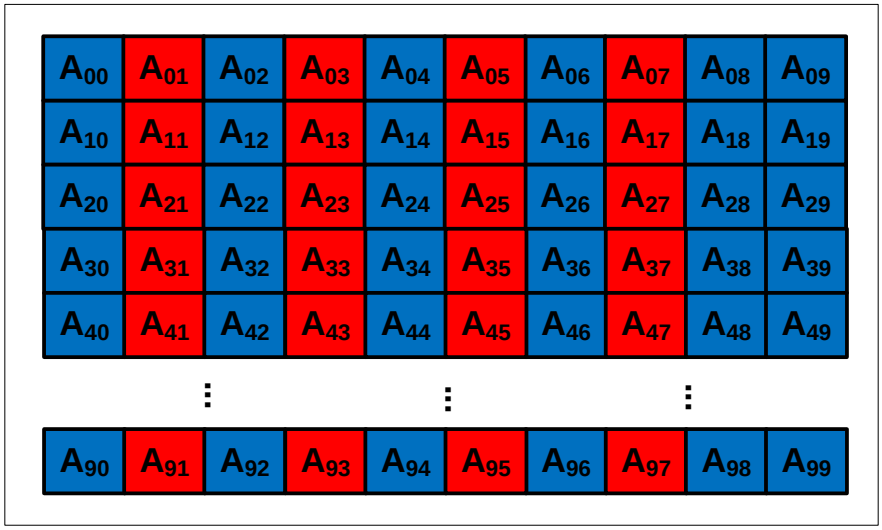

Figure 3.3: An example of vector memory shape instructions where red elements are loaded from matrix.

- Min - finds minimum element in a vector register.

The library also implements new reduction instruction called *sub-reduction add* or *sub-sum*. This instruction performs the sum for sub-sets in a vector register. For example, we want to sum group of 3 elements of an array, the first 3 elements, then next 3, etc. We can do it with the existing sum instruction, but we will have short vectors and several load instructions, as figure 3.4 (a) illustrates. Using sub-sum, we just need two instructions instead: one to load the vector and for another to perform sub-reduction, as it is shown in figure 3.4 (b). With this approach, the number of vector instructions is decreased and the average vector length is increased.

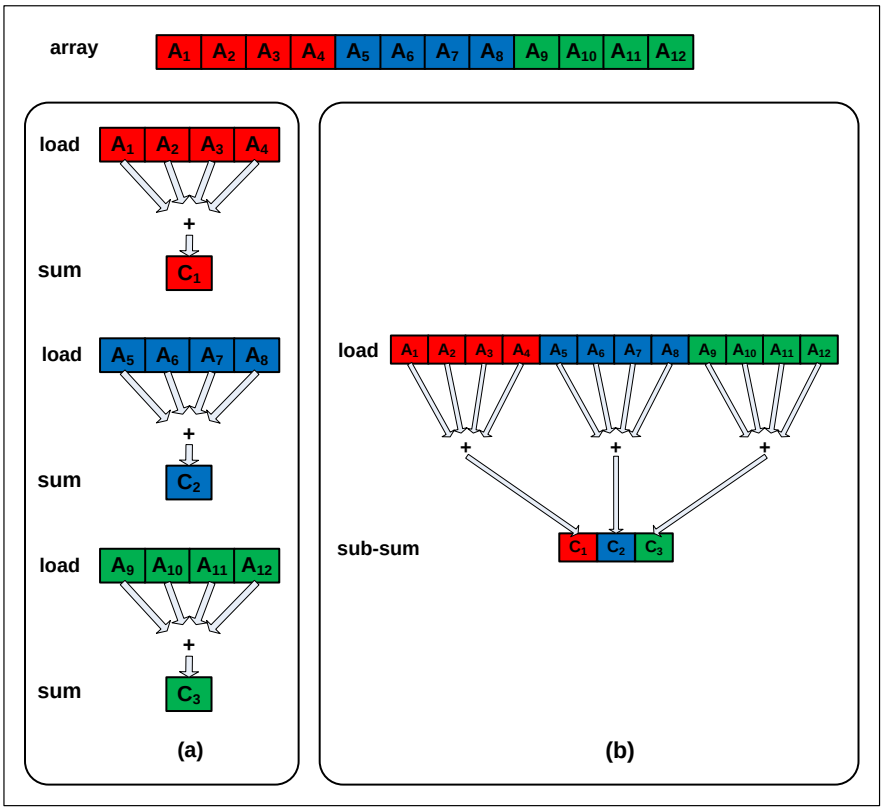

Figure 3.4: An example of performing sum for sub-sets of array using: (a) reduction instruction *sum* and (b) new reduction instrucion *sub-sum*.

3.2.4 Bit and element manipulation instructions

Standard bitwise instructions such as OR, AND, etc. are implemented in the vector library. The library also provides instructions that manipulate individual elements of vector registers such as *getelem*, *setelem*, *select*, *init*, etc. *Getelem* gets a particular element from the specified source register, while *setelem* sets a particular element of vector destination register with specified scalar value. *Select* instruction is related with masking (see Section 2.2.4) and selects elements from one or other source depending on the value in vector mask register. *Init* or iota has two operands, base and stride, and creates the following array:

```
base + j* stride, where j = 0, 1, 2... vl-1.
```

type	opcode	source regs	dest reg	masking	scalar	description
Arithmetic	ADDV	2	1	x	x	V-V or V-S addition optionally over VMR.
	SUBV	2	1	x	x	V-V or V-S subtraction optionally over VMR.
	MULV	2	1	x	x	V-V or V-S multiplication optionally over VMR.
	DIVV	2	1		x	V-V or V-S division.
and	VMOD	1	1		x	Modulo instruction.
	LOGOPV	2	1		x	V-V or V-S logical instructions.
	CMPV	2		x	x	V-V or V-S compare instructions.
Logical	CMPVMR			x		Compare instructions use VMR registers as sources.
Memory	LD		1			Memory load vector instruction.
	LDCAST		1			Memory load vector instruction with cast.
	STV	1		x		Memory store vector instruction, optionally over VMR.
	LDVS		1		x	Strided memory load vector instruction.
	STVS	1			x	Strided memory store vector instruction.
	LDVGI	1	1	x		Indexed memory load vector instruction, optionally over VMR.
	STVGI	2				Indexed memory load vector instruction.
	LDVSHAPE		1		x	Memory load vector instruction with shape.
	STVSHAPE	1			x	Memory store vector instruction with shape.
Reduction	VREDADD	1			x	Reduction sum instruction.
	VSUBREDADD	1	1			Sub-reduction sum instruction.
	VREDMIN	1			x	Find minimum instruction.
	VREDMAX	1			x	Find maximum instruction.

Table 3.1: Implemented vector arithemetic, logical, memory and reduction instructions in the vector library. V-V is vector-vector, while V-S is vector-scalar.

type	opcode	source regs	dest reg	masking	scalar	description
	VSHIFT	1	1		x	Vector shift instruction.
	VINIT		1		x	Vector initialization instruction.
	VSETS	1			x	Sets all elements of vector to value from scalar.
	BITSCANP			x	x	finds position of first set bit in VMR.
	BITSCANN			x	x	finds position of first zero bit in VMR.
Bit	SELECTV	2	1	x	x	Vector-vector or vector-scaler select instruction.
	BWOPV	2	1		x	Vector-vector or vector-scaler bitwise instructions.
	CASTV	1	1			Vector cast instruction.
	GETELEM	1			x	Get element from vector register.
	SETELEM		1		x	Set element to vector register.
	VREPLICATE	1	1		x	Replicate vector coresponding number of times.
and	INSERTVM	1	1	x	x	Copy the content of a vector mask register as one or more elements of a vector register.
	BITSCANV	1	1			Generate indexes of all bits sets in a vector register.
	BITSCANVVM		1	x		Generate indexes of all bits sets in a VMR.
	BITSETVVM	1	1	x		Set bits indicated by an index VMR.
	POPCOUNTVM			x	x	Count the number of set bits in a VMR.
Element	POPCOUNTV	1			x	Count the total number of set bits in a vector register.
	POPCOUNTVV	1	1			Count the number of set bits in a vector register (per element).
	POPCOUNTVVM	1	1	x		Count the number of bits set up to a each position in a vector mask.
	BITCLEARV	1	1			Clear the bits in a vector register as indicated by an index vector.
Manipulation	BITCOMPRESS	2	1			Copy the bits of source vector register 1 according to the set bits of source vector register 2. Copied bits are stored consecutively in destination vector register.
	COMPRESS	1	1	x		Copy the elements of source vector register according to the set bits of vector mask. Copied elements are stored consecutively in destination vector register.

Table 3.2: Implemented vector bit and element manipulation instructions in the vector library.

3.3 Results and Statistics

One of the purposes of the vector library is to generate results and statistics from vectorized applications. The library collects following statistics:

- *Percentage of vectorized code.* It tells us what is the degree of vectorization in an application; number of operations executed in the vectorized code and the number of operations executed in scalar code. The PAPI library [7] is used to count the number of instructions executed in the scalar code (for more details see Section 3.6).
- *Instruction type statistics.* It gives us information about distribution of the vector instructions; how many times each instruction is executed and how many operations are executed per vector instruction.
- *Distribution of vector lengths.* This information tells us how many instructions are executed for every vector length up to maximum vector length and helps to determine the utilization of the vector register file.
- *Algorithmic vector lengths.* Lengths of arrays (vectors) in algorithms are sometimes longer than the maximum vector length. The library collects these statistics optionally and it is done manually during the process of vectorization.
- *Stride distribution information.* It tell us how many memory instructions are executed with the corresponding stride and it helps to determine the dominant memory access patterns.
- *Information related with the vector mask registers.* The vector library supports masking and also collects statistics related with masking. It tells us how many instructions and operations are executed over a vector mask register and how many operations are really executed (operations for which bit in the vector mask register was set).

If more information needs to be harvested, the vector library can be easily extended to collect desired statistics.

3.4 Instruction and address traces

The library has the support to generate a trace of executed vector instructions as well as a trace of addresses for each memory vector instruction. The traces are used as inputs in the ETModel to estimate the execution time of the vectorized application (see Chapter 4). The traces can be generated in binary format, textual or both.

The instruction trace has the following format:

```
[Num_of_scalar_ins] [Set_VL] Block_of_vector_ins
```

Num_of_scalar_ins is the number of scalar instructions between two blocks of vector instructions; *Set_VL* is the vector length used in the following block of vector instructions; and *Block_of_vector_ins* is a block of vector instructions. Each instruction in the block is represented by the *instruction opcode*, *destination* and *source registers* with their *types*. The PAPI library is used to automatically count the number of scalar instructions between two blocks of vector instructions.

Square brackets mean that *num_of_scalar_ins* and *set_VL* are optional. Sometimes there are no scalar instructions between two blocks of vector instructions, they are just executed with different vector lengths or two blocks are executed with the same vector length and between them there is some scalar code.

In order to reduce the size of the address trace, it does not always contain the addresses of all locations in memory accessed by a vector memory instruction. In general, it contains information that are sources of the memory instructions: opcode, type of accessed data, base address, number of accessed elements, etc. Different types of vector memory instructions have different formats.

Unit-stride vector memory instruction has the following format:

```
opcode type start_address num_elems
```

where *opcode* identifies particular instruction which accesses *num_elems* elements of type *type* starting from address *start_address*. These four parameters are enough to generate addresses of all elements accessed by an unit-stride vector memory instruction.

Strided vector memory instruction needs a small addition to the format of unit-stride vector memory instruction:

```
opcode type start_address num_elems stride
```

where *stride* is the distance between two accessed elements. Again this is enough to generate all addresses for strided vector memory instructions.

Vector memory shape instructions need more information than unit-stride or strided vector memory instructions. The format is following:

```
opcode type start_addr num_elems stride 1st_span span skip
```

where *opcode* identifies particular instruction which loads-stores *num_elems elements* of type *type* starting from address *start_addr*. *Stride* describes the spacing between each accessed element. *Span* describes how many elements to accessed at stride spacing before applying the second-level *skip* offset. *1st_span* is related with strip-mining (see section 5.1). Sometimes a memory shape instruction does not access all elements in first *span* group, because some number of elements were accessed in previous stripe of strip-mined loop. This is the reason for adding *1st_span* into the address trace for vector memory shape instructions.

The most problematic instructions are indexed and instructions executed over a vector mask register. For an indexed vector memory instruction the address trace contains the indices of all accessed elements and the base address:

```
opcode type start_address num_elems array_of_indices
```

For vector memory instruction executed over vector a mask register, the trace contains only those addresses for which the corresponding bit in the vector mask register is set:

```
opcode type start_address num_elems array_of_accessed_addr
```

3.5 Wrappers

Some benchmarks were written in C or FORTRAN. The library contains wrapper functions that allow vectorization of applications written in FORTRAN and C.

3.6 Implementation details

Most code in the library is implemented in C++ using templates. Templates are very well suited for our implementation because vector registers and implemented

vector instructions have to support different data types. Several optimization techniques such as in-lining, macros, etc. are applied. It allow us to have compact and optimized code.

As mentioned above, the PAPI library is used to count scalar instructions between blocks of vector instructions. The idea is to count operations between calls to functions of the vector library that simulate vector instructions. If there are two consecutive function calls to the vector library without scalar code between them, the PAPI library should still count some number of operations, which is overhead of calling functions of the vector library. After some experiments, we set a fixed threshold that is used to determine if two vector instructions are from the same block or the obtained number of operations using PAPI represents scalar code between two blocks of vector instructions. The threshold is highly dependent on the host ISA and compiler.

We used Dell sever with four cores as evaluation environment for generating instruction traces and collecting statistics. Each core is Xeon Dual-Core 5160 @ 3.00GHz with 4MB of cache and 16GB of RAM. All applications were compiled with gcc compiler (version 4.4.3), except the FaceRec which was compiled with icc compiler (version 12.0.2).

-O3 optimization flag is used for all applications, except for the FaceRec. *-xSSE3 -fast -no-scalar-rep -unroll1* optimizations flags are used for the FaceRec.

The overhead of the vector library to the original application's execution time depends on the mode in which the library is run. The basic version of vector library just collects results for the instruction level characterization. It can count the scalar operations using the PAPI library and generate instruction and address traces.

For example, the execution time of the original version of the Sphinx3 is sixteen minutes. The basic version is less than four times slower than the original version. The version than counts the scalar operations is 200 times slower because the PAPI library adds a lot of overhead. The version that collects instruction trace is 400 times slower because it has to write trace into a named pipe, while the second process compresses that named pipe.

The vector library adds a lot overhead, but still it is less than a simulator.

Chapter 4

ETModel

The target applications can be vectorized using the vector library and statistics such as instruction level characterisation can be obtained but there is no information related with execution time on a vector architecture. Detailed simulators are very often used to evaluate performance of a processor. Although it is accurate, this method is time-consuming, both to create the simulator and to run the simulations. Our idea was to have fast results and to a perform preliminary evaluation and early parameter exploration.

The ETModel is a simple trace-driven simulator for vector processors based on the work presented by Karkhanis & Smith [23] and Hennessy & Patterson [20]. Karkhanis & Smith [23] propose analytical performance model for superscalar processors, while Hennessy & Patterson [20] describe basic vector architecture as well as techniques and concepts that help in enhancing vector performance. The model consists of a component that models the micro-architecture of the desired vector processor and methods that apply chaining and other implementation features as described in section 2.2. The model uses an instruction trace, optionally an address trace and IPC of scalar code as inputs to estimate execution time of the vectorized application. Instruction and address traces are generated by the vectorized application using the vector library (see Section 3.4).

4.1 Micro-Architecture

The ETModel models a parametrizable in-order vector architecture similar to the architecture presented in [20]. It consists of basic units such as a vector register file, ALU units, vector load/store units and a memory hierarchy (figure 4.1). The model is parametrizable because we can specify how many instances of any unit are available in the model (e.g. number of lanes, number of ALUs, LD/ST units, etc.). All units are also parametrizable:

- Number of vector registers and size of vector register for the vector register file.
- Types of instructions and types of data that each ALU unit supports (e.g. FP multiply unit, logical unit, etc.).
- Types of memory instructions that each vector load/store unit support.

Memory can be modeled in two different ways: simple and detailed. In the simple approach, the memory is simply modeled with parameters such as cache miss rates, bandwidths and latencies for each type of memory instruction. L1 and L2 cache misses are uniformly modeled using cache miss rates. For example if L1 cache miss rate is 10%, every tenth access to L1 cache will be modeled as miss. This approach is fast but has low accuracy.

In the second approach, a more accurate memory model is used. A trace of addresses of vector memory instructions is generated by vector library and a simple cache simulator is implemented based on the memory model of SimpleScalar [8]. For each vector memory instruction from the instruction trace, the address trace contains all necessary information to generate all accessed addresses.

In both memory models, different types of vector memory instructions are modeled separately. For the unit-stride memory instruction, we can load/store a whole L1 cache line with only one access, while for all other types, only one element per access is loaded/stored.

All the parameters mentioned above help us to analyse a broad range of different configurations of vector processors ranging from very simple vector processors with only one lane, one vector load/store unit and a small number of functional

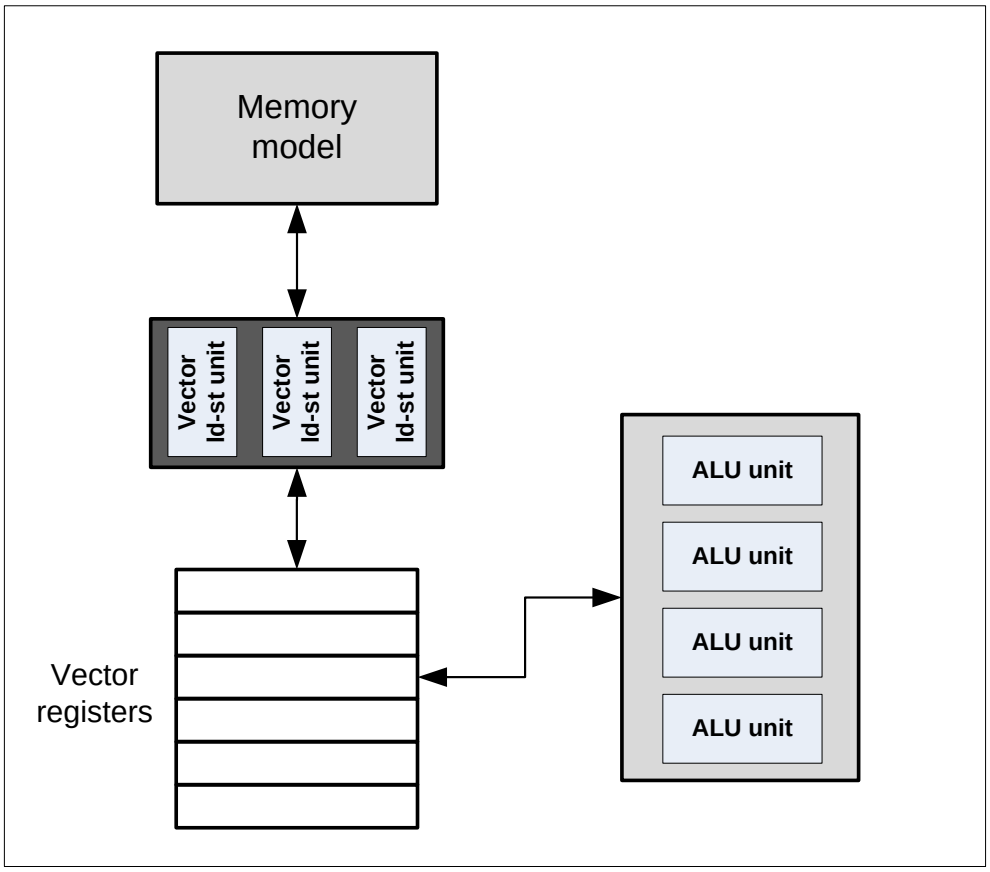

Figure 4.1: The basic structure of model register-based vector architecture.

units, to very complex vector processor with multiple lanes, several vector load-/store units and a rich set of functional units.

Beside the concept of multiple lanes, other important techniques such as chaining and pipelined instruction start-up (dead time or recovery time) are also included into the model.

The ETModel also provides detailed statistics of the resource usage. For the cache hierarchy, the ETModel collects the following:

- Total number of L1 accesses, the number of L1 miss (miss rate) and hit (hit rate) accesses.
- Total number of L2 accesses, the number of L2 miss (miss rate) and hit (hit rate) accesses.

- Total number of cycles spent in the memory hierarchy.
- Average memory access time per access.
- The number of L1 accesses per memory unit.

For every functional unit, there is information about its usage; the number of the cycles that particular functional unit was busy. There is also information about dependent vector instructions: the number of vector instructions executed using chaining or without chaining, as well as the number of cycles waiting for a free ALU or memory unit. The distribution of data types (see section 5.3.4) was collected using the ETModel.

All these statistics help us to better understand the obtained results and the behavior of the vectorized applications.

4.2 Top-Level Model

For reasoning about vector processor operation, we utilized vector execution time and enhancing vector performance models from [20] to create the algorithm of execution in ETModel as shown in figure 4.2. The model reads the instruction trace sequentially. Each opcode is loaded from the instruction trace and fit as one of three possibilities:

1. The number of scalar instructions between two blocks of vector instructions.

   The execution time for these scalar instructions is computed using the input parameter scalar IPC (number of scalar instructions divided by IPC). We assume that there is no overlapping between scalar and vector instructions.

2. A set vector length instruction.

   The VL register is set for following vector instructions.

3. A vector instruction.

   First, operands and type of operands are read from the instruction trace file. After that, the model checks if it is a memory or arithmetic/logic

instruction. In both cases, the model tries to find a free vector load/store or ALU unit. If there isn't a free unit, memory or arithmetic/logic instruction waits for the first free load/store or ALU unit. Finally, when memory or arithmetic/logic instruction finds free load/store or ALU unit it checks for dependency and if there is no dependency the memory or arithmetic/logic instruction is issued. Otherwise, the model checks for chaining and if it is possible it issues chained memory or arithmetic/logic instruction. If it is not possible, the memory or arithmetic/logic instruction is issued in non-chained mode. It means that current instruction will be issued when the previous instruction, from which the current instruction depends, is finished.

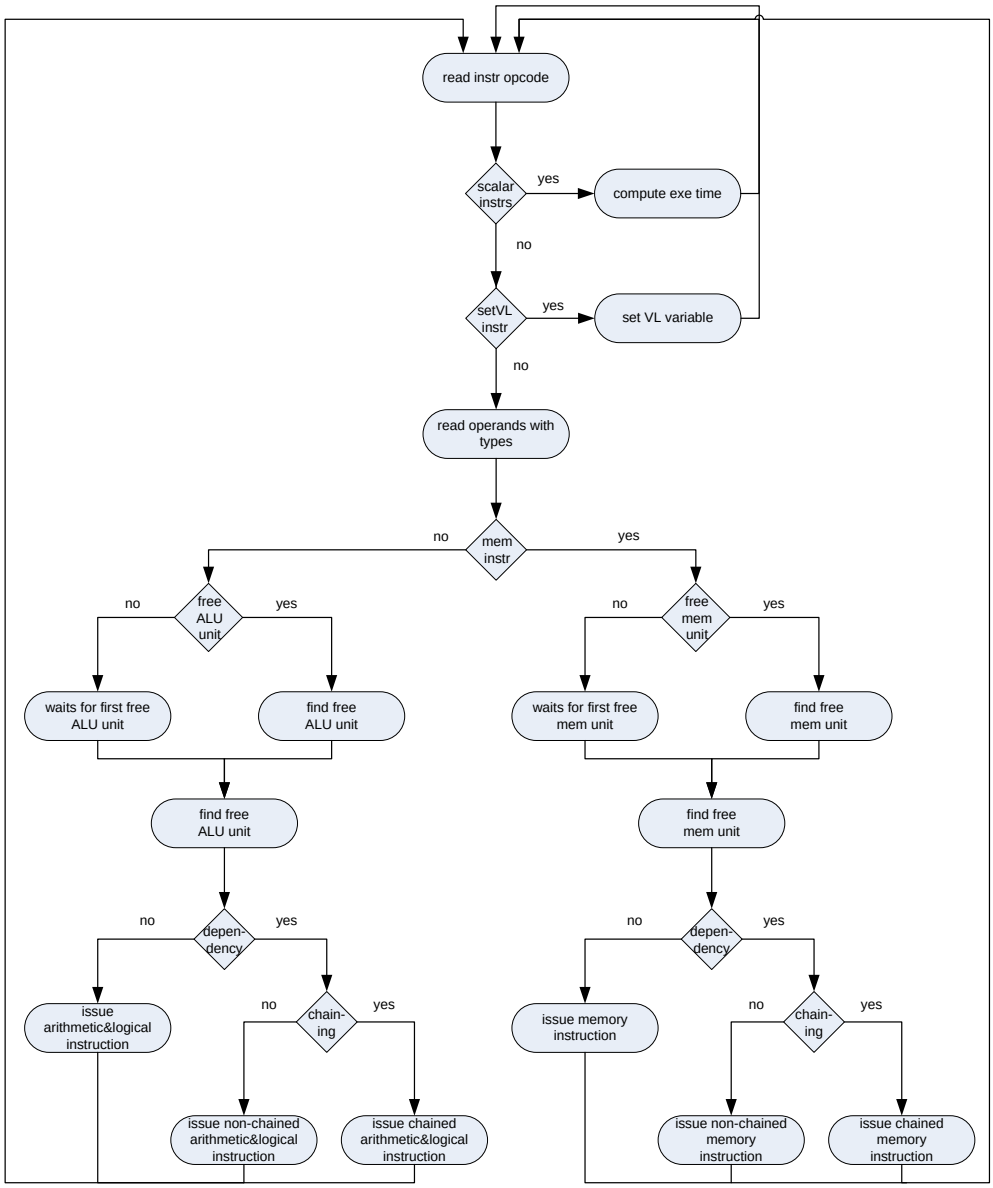

Figure 4.2: Diagram of algorithm used to compute execution time in the ET-Model.

Chapter 5

Vectorization

Our target applications are chosen from a range of applications that are used or will be used in handheld devices or data centers. This range includes computer vision, speech recognition, face recognition, 3D graphics and video media, console games, database management systems, etc. Asanovic et al. [5, 6] identify 13 "dwarfs", which each capture a pattern of computation and communication common to a class of important applications. We also tried to cover several dwarfs choosing the applications that contain different dwarfs.

5.1 Methodology

The process of vectorization contains the following steps:

- Profiling. The goal of profiling is to find kernels that consume the most execution time. Applications are profiled using standard Unix profiler called gprof and/or using Intel's performance analyzer called VTune.
- Kernel testing on vectorization. Kernels that consume the most of execution time are examined for vectorization (e.g. does kernel contain loops, dependency, what is the size of loop, etc.), and if they are suitable for vectorization vectorized pseudo-code is written.
- Vectorization of kernels and applying strip-mining. The kernels are vectorized using functions from the vector library. The actual vector length

required by an algorithm (the number of iterations of a loop) is usually larger than maximum vector length (MVL) supported by the architecture and often unknown at compile time. Strip-mining is a technique that allows operating on "stripes" of the data of length less or equal to MVL. Strip-mining is applied in all vectorized kernels in order to support vectorization of loops that are longer than the size of vector register. It also allows changing the maximum size of vector register in the vector library without modification in the vectorized application. MVL is a parameter of the vector library that is set at compilation time.

- Collecting and analysing results. In this step, the vectorized application is run and statistics are collected. The percentage of vectorized code, average vector register length, etc. are analysed.

- Performing additional modifications of the vectorized code. If the collected results are not satisfying, additional code modifications are performed in order to improve the average vector length (longer vectors) or the percentage of vectorized code, if it's possible.

5.2 Vectorized applications

This section describes applications that have been vectorized. They are chosen from several areas such as speech recognition, face recognition, data-mining, graphical models and video compression. These applications also cover several dwarfs such as dense linear algebra, sparse linear algebra, graphical models and finite state machine.

5.2.1 SPEC2006 Sphinx3 benchmark

Sphinx3 is a widely known speech recognition system from Carnegie Mellon University. It includes both an acoustic trainer and various decoders, *e.g.*, text recognition, phoneme recognition, N-best list generation, etc. Sphinx3 adopted the prevalent continuous hidden Markov acoustic model (HMM) representation

and has been used primarily for high-accuracy, non-real-time recognition. The benchmark is written in C.

5.2.2 SPEC2006 H264ref benchmark

H264ref is a reference implementation of H.264/AVC (Advanced Video Coding), the latest state-of-the-art video compression standard. The standard is developed by the VCEG (Video Coding Experts Group) of the ITU (International Telecommunications Union and the MPEG (Moving Pictures Experts Group) of the ISO/IEC (International Standardization Organization). This standard replaces the currently widely used MPEG-2 standard, and is being applied for applications such as the next-generation DVDs (Blu-ray and HD DVD) and video broadcasting. This benchmark is written in C.

5.2.3 SPEC2006 Hmmer benchmark

Hmmer benchmark searches a gene sequence database. It applies profile Hidden Markov Models (profile HMMs), statistical models of multiple sequence alignments, which are used in computational biology to search for patterns in DNA sequences.

The technique is used to do sensitive database searching, using statistical descriptions of a sequence family's consensus. It is used for protein sequence analysis. It is written in C.

5.2.4 SPEC2000 FaceRec benchmark

This is an implementation of the face recognition system described in [24]. It is an object recognition system based on the Dynamic Link Architecture, which is an extension to classical Artificial Neural Networks. The benchmark is written in FORTRAN.

5.2.5 ECLAT MineBench

ECLAT is an application from the data-mining realm. In particular, it implements a known algorithm for frequent itemset mining. The original implementa-

tion was borrowed from NU-Minebench [27]. The most relevant operation consists in the intersection of large sparse sets. The Minebench implementation has been modified to use another data structure more suitable for vectorization, based on bitmaps. The benchmark is written in C++.

5.3 Vectorized kernels

The vectorization of some kernels is neither obvious nor trivial. It requires a lot of effort, deep understanding of kernels, sometimes algorithm modification or introducing new vector instructions. In this section, we explain the process of vectorization for some kernels that was not trivial.

In all vectorized kernels, the left-most operand is the destination register if an instruction has a destination operand. The letter R represents a vector register, while MV represents a vector mask register.

If a kernel contains a loop with a constant number of iterations during the execution of application, this number is included in the loop condition. In order to simplify the process of vectorization, we do not present strip-mining in the most kernels.

5.3.1 Sphinx3

The Sphinx3 application contains several kernels that are difficult to vectorize.

5.3.1.1 Kernel 1. vector_gautbl_eval_logs3

The source code of this kernel is shown above. This function takes 42.22% of execution time. As we can see the function contains two loops, the inner loop always has 13 iterations and the outer loop has 4,096 iterations for the ref data input set. In the outer loop, three vectors are loaded and in the inner loop it is performed some computation on all the elements in the vectors.

```
float32 *m, *v; // local variables
float64 dval, diff; // local variables
float32 *x, *score; // functions arguments
```

```
double f = log_to_logs3_factor();

for (r = 0; r < 4096; r++){
   m = gautbl->mean[r];
   v = gautbl->var[r];
   dval = gautbl->lrd[r];
   for (i = 0; i < 13; i++){
     diff = x[i] - m[i];
     dval -= diff * diff * v[i];
   }
   if (dval < gautbl->distfloor)
     dval = gautbl->distfloor;
   score[r] = (int)(f * dval)
}
```

The first and obvious approach is to vectorize the inner loop, but in that case, we will have very short vectors (length of thirteen) and reduction that is an expensive instruction. Our approach is to vectorize the outer loop and in this case, we will have vectors with length 4,096, but we will load them with a stride of 13. The code below is the pseudo code of vectorized version. We ignore strip-mining here.

```
ldv R1, gautbl->lrd

for (i = 0; i < 13; i++){
   ldvs R2, gautbl->mean[i], 13;
   subsv R3, R2, x[i];
   mulv R4, R3, R3;
   ldvs R5, gautbl->var[i], 13;
   mulv R6, R4, R5;
   subv R1, R1, R6;
}
cmpvs_gt MR1, R1, gautbl->distfloor;
selectvs R7, R1, gautbl->distfloor, MR1;
mulvs R8, R7, f;
```

```
castv_int32_db R9, R8;
stv R9, score;
```

5.3.1.2 Kernel 2. subvq_mgau_shortlist

In this kernel, the elements of the array *map* are used as indices to access elements from the array *vqdist*. A partial sum of the array *vqdist* is computed for groups of three elements. All the sums are stored in the array *gauscore* and the maximum of them is computed. Finally, for any element from array *gauscore* that is greater or equal to threshold *th*, its index is stored into the array *sl*.

```
int32 *vqdist;
int32 *map;
bv = MAX_NEG_INT32;

for (i=0; i<8; i++){
   v = vqdist[*(map++)];
   v += vqdist[*(map++)];
   v += vqdist[*(map++)];
   gauscore[i] = v;
   if (bv < v)
      bv = v;
}
th = bv + beam; nc = 0;
for (i=0; i<8; i++){
   if (gauscore[i] >= th)
      sl[nc++] = 1;
}
sl[nc] = -1;
```

This kernel can be vectorized using reduction instructions, but the vector length will be very short (just three elements). In order to improve vector length, we introduced a new vector instruction called *subreduction* (see section 3.2.3) that performs the sum of sub-sets in a vector register. With this instruction, we can vectorize the kernel with a vector length of 24. The mnemonic used for this instruction is *vsubredadd*. The vectorized pseudo-code is shown below.

```
ldv R1, map;
ldvgi R2, R1, vqdist;
vsubredadd R3, R2, 3;
vredmax scalar, R3;
stv R3, gauscore;
th = scalar + beam;
cmpvs_ge MR1, R3, th;
vinit R4, 0, 1;
stvmask R4, MR1, vq->mgau_sl;
popcountvm temp, MR1;
sl[temp] = -1;
```

5.3.1.3 Kernel 3. mdef_sseq2sen_active

In this kernel, all elements of the array *sswq* are compared with zero in the outer loop. Then for all non-zero elements of the array *sswq*, their positions are used as indices to access an array of three elements from matrix *mdef-sseq*. After that the three elements of the array are used to index the array *sen*. All accessed elements of the array *sen* are set to one.

```
int16 *sp;
for (ss=0; ss<32639; ss++){
   if (sswq[ss]){
      sp = mdef->sseq[ss];
      for (i=0; i<3; i++)
         sen[sp[i]] = 1;
   }
}
```

In this kernel, the inner loop can be vectorized easily, but vectors will be very short. We decided to vectorize the outer loop. We load elements from array *sswq* and compare with zero. We detected that this array contains a lot of zeros. This is the reason why we count how many elements are different from zero using the instruction *popcountvm*. If this number is equal to zero, we skip the rest of the computation. Otherwise, we vectorize the kernel as it is shown below. *vinit*

instruction creates index vector. The first element is i and every next element is incremented for three. We could use here a different approach: if the *scalar* value is lower than some threshold then execute the loop in scalar mode. To do it, we need some experimental results to choose the proper threshold.

```
vsets R0, 1;
ldv R1, sswq;
smpvs_ne MR1, R1, 0;
popcountvm MR1, scalar;
if (scalar != 0){
   for (i = 0; i < 3; i++){
        vinit R2, i, 3;
        ldvgimask R3, R2, MR1, mdef->sseq[0];
        stvsi R0, R3, sen;
   }
}
```

5.3.1.4 Kernel 4. dict2pid_comsenscr

In the kernel below, the elements of the array *comstate* are used as indices to access the array *senscr*. The inner loop is particularly interesting because we do not know the number of iterations until execution time. It uses the *break* keyword to exit the inner loop when it finds the first negative element in the array *comstate*.

```
int16 *comstate;

for (i=0; i<873; i++){
   comstate = d2p->comstate[i];
   best = senscr[comstate[0]];
   for (j=0; ; j++){
      k = comstate[j];
      if (k < 0)
         break;
      if (best < senscr[k])
```

```
            best = senscr[k];
      }
      comsenscr[i]=best+d2p->comwt[i];
}
```

We decided to vectorize this kernel in the following way. The code below presents a vectorized version of the kernel using strip-mining, because it is easier to understand. We load maximum vector length (MVL) elements from array *comstate* and compare if these elements are greater or equal to zero. Then we try to find the position of the first negative element in the vector (*bitscann* finds position of the first zero in the vector mask register). If this number is equal to MVL, we perform the computation on all elements. Otherwise, we perform the computation just on all positive elements until the first negative element. The pseudo-code below presents the vectorized kernel. We assume that the size of *comstate* is a multiple of MVL and all memory accesses would be valid. If it is not possible, we can check at the beginning if we have to use a smaller VL.

```
VL = MVL;
temp = VL;
for (i=0; i<873; i++) {
   iter = 0;
   temp = VL;
   best = INT32_MIN;
   while(temp == VL){
      setvl VL;
      ldv R1, (d2p->comstate[i] + iter * VL);
      cmpvs_ge MR1, R1, 0;
      bitscann temp, MR1;
      if (temp != 0)
         setvl(temp);
      ldvgi R2, R1, senscr;
      vredmax scalar, R2;
      if (best < scalar)
         best = scalar;
```

```
      iter++;
   }
   comsenscr[i] = best + d2p->comwt[i];
}
```

5.3.1.5 Kernel 5. approx_cont_mgau_frame_eval

The code below is just a part of the kernel *approx_cont_mgau_frame_eval*. For all the elements of array *sen_active* that are different from zero, *best* is subtracted to the corresponding elements in array *senscr*.

```
for (t = 0; t < 6144; t++) {
   if(sen_active[s])
      senscr[s]-=best;
}
```

The vectorized kernel is shown in the code below.

```
ldv R1, sen_active;
cmpvs_ne MR1, R1, 0;
ldv R2, senscr;
subvsmask R2, R2, R1, MR1;
stv R2, senscr;
```

5.3.2 FaceRec

The four most executed kernels for FaceRec application have been vectorized. In the first version of the vectorized code, the average vector length was very short. We spent a lot of time improving the average vector length.

5.3.2.1 Kernel 1. passb4

In the code below, there are two loops. The vectorization of the inner loop is not difficult, but 30% of all instructions are executed with a vector length of four because there are three different combinations for iterations in the outer and inner loops: 1) L1 = 1, IDO = 128, 2) L1 = 4, IDO = 32 and 3) L1 = 16, IDO =

8. For the third case, the vector length for the vectorized version is four because there are only four iterations of the inner loop (the step in the inner loop is two).

```
DO 104 K=1,L1
   DO 113 I=2,IDO,2
      TI1 = CC(I,1,K)-CC(I,3,K)
      TI2 = CC(I,1,K)+CC(I,3,K)
      TI3 = CC(I,2,K)+CC(I,4,K)
      TR4 = CC(I,4,K)-CC(I,2,K)
      TR1 = CC(I-1,1,K)-CC(I-1,3,K)
      TR2 = CC(I-1,1,K)+CC(I-1,3,K)

      ! Some computation

      CH(I,K,2) = WA1(I-1)*CI2+WA1(I)*CR2
      CH(I-1,K,3) = WA2(I-1)*CR3-WA2(I)*CI3
      CH(I,K,3) = WA2(I-1)*CI3+WA2(I)*CR3
      CH(I-1,K,4) = WA3(I-1)*CR4-WA3(I)*CI4
      CH(I,K,4) = WA3(I-1)*CI4+WA3(I)*CR4
   113    CONTINUE
104 CONTINUE
```

In order to increase the vector length for the second and third case, we decided to use vector memory shape instructions. For example, the distance between elements that are used to compute *TI1*, in the two consecutive iterations of the outer loop is constant. Therefore, there is regular access pattern with constant stride across the iterations of the outer loop. We can create vectors of 64 elements using vector memory shape instructions. We do not show vectorized code because it is too long.

5.3.2.2 Kernel 2. gaborTrafo

The code displayed below is an interesting part of this kernel. *FCImage* and *FCTemp* are two two-dimensional arrays of complex numbers with 256 elements in each dimension. *Kernel* is three-dimensional array of floating-point elements

with dimensions of 256, 256, and five. Two consecutive *CSHIFT* statemens shift the first and the second dimension of the three-dimensional array *Kernel* for *ShiftRow* rows and *ShiftCol* columns. The third dimension is fixed.

All elements from the first and second dimension of the three-dimensional array *Kernel* with a fixed third dimension are multiplied with scalar value *DC*. Then all shifted elements of *Kernel* are subtracted with the corresponding elements that are multiplied with *DC*. At the end, resulting elements are multiplied with corresponding elements from the array *FCImage* (they are multiplied with the real and imaginary parts of a complex number) and stored to the array *FCTemp*.

```
Complex(4) :: FCImage(256, 256), FCTemp(256, 256)
Real(4) :: Kernel(256, 256, 5)
FCTemp = FCImage&
&  * (CSHIFT(CSHIFT(Kernel (:, :, Level),ShiftRow,1),ShiftCol,2) &
&  - DC * Kernel (:, :, Level))
```

The critical part is subtraction between the shifted version of *Kernel* and the original version that is multiplied with *DC*, as figure 5.1 shows. We have to subtract the first part of the shifted *Kernel* with the first part of the original *Kernel*, the second part with the second, etc. Consecutive elements in FORTRAN are stored in column-major order.

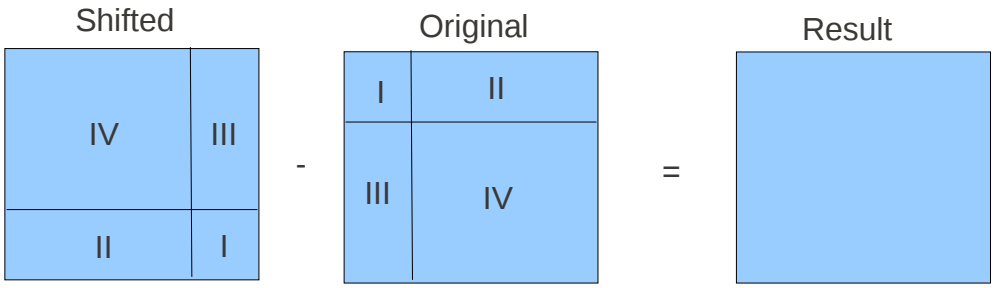

Figure 5.1: Subtraction of shifted version of *Kernel* and the original version.

In the first and simple approach, we load the first column from the first part of the shifted *Kernel* and the first column from the first part of the original *Kernel* and then subtract them. Then we repeat the process for the second columns, etc. The process is the same for the remaining three parts.

It means that the vector lengths are *ShiftRow* for the first and the second part and 256-*ShiftRow* for the last two parts. *ShiftRow* is a small number in a lot of cases and it means that we have short vectors.

We decided here again to use vector memory shape instructions in order to increase the average vector length. Using vector memory shape instructions, we can access all the elements from one part or up to MVL using just one vector instruction.

5.3.2.3 Kernel 3. TopCostFct

This kernel is the most difficult one. This kernel performs some kind of stencil computation. In order to compute one element of a two-dimensional array, it has to access its neighbor elements. The code of this kernel is shown below. It is important to mention that the computation is performed for all the elements of the two two-dimenstional arrays and this is the code for central elements. This computation is little bit different for corner cases and that complicates the vectorization. *CrdX* and *CrdY* are two two-dimensional array with dimensions twelve and nine. This kernel is called inside four nested loops. The two most outer loops iterate over all elements of *CrdX* and *CrdY*. The two most inner loops change current elements of the *CrdX* and *CrdY* from the current value minus eight to the current value plus eight. It means that there are 64 iterations in the two most inner loop. After this kernel computed *OC* and *NC* if is some condition is true, the current element in the arrays *CrdX* and *CrdY* are updated. It means that there is a dependency across iterations.

```
Integer :: EdgeXP, EdgeXM, EdgeYM, EdgeYP
Integer :: V1X, V1Y, V2X, V2Y, OCT, NCT
EdgeXP (V1X, V1Y, V2X, V2Y) = (V2X-V1X-StepX)**2+(V2Y-V1Y)**2
EdgeXM (V1X, V1Y, V2X, V2Y) = (V2X-V1X+StepX)**2+(V2Y-V1Y)**2
EdgeYP (V1X, V1Y, V2X, V2Y) = (V2X-V1X)**2+(V2Y-V1Y-StepY)**2
EdgeYM (V1X, V1Y, V2X, V2Y) = (V2X-V1X)**2+(V2Y-V1Y+StepY)**2

OC=0 +EdgeXM(CrdX(IX,IY),CrdY(IX,IY),CrdX(IX-1,IY),CrdY(IX-1,IY))
NC=0 +EdgeXM(NewX,       NewY,       CrdX(IX-1,IY),CrdY(IX-1,IY))
```

```
OC=OC+EdgeYM(CrdX(IX,IY),CrdY(IX,IY),CrdX(IX,IY-1),CrdY(IX,IY-1))
NC=NC+EdgeYM(NewX,       NewY,       CrdX(IX,IY-1),CrdY(IX,IY-1))
OC=OC+EdgeXP(CrdX(IX,IY),CrdY(IX,IY),CrdX(IX+1,IY),CrdY(IX+1,IY))
NC=NC+EdgeXP(NewX,       NewY,       CrdX(IX+1,IY),CrdY(IX+1,IY))
OC=OC+EdgeYP(CrdX(IX,IY),CrdY(IX,IY),CrdX(IX,IY+1),CrdY(IX,IY+1))
NC=NC+EdgeYP(NewX,       NewY,       CrdX(IX,IY+1),CrdY(IX,IY+1))
```

In the first approach, we changed the code of this kernel in order to vectorize it, but the best what we have were vectors with vector length of 8. After several tests, we realized that the current elements of the *CrdX* and *CrdY* are very rarely updated. Then we decided to vectorize the two most inner loops and increase the length of vectors to 82. If we detect that there is an update after this computation, then we have to recompute this computation using the first approach and vectors of length eight.

5.3.3 ECLAT

My workmate has vectorized this application. I'll not present details of the vectorization here because it is part of unpublished work. In the experiments of this thesis we present the results for only one kernel of ECLAT, but it is the core operation of the algorithm.

5.3.4 Hmmer

The Hmmer application contains only one significant kernel that takes more than 90% of the execution time. The vectorization of two-thirds of this kernel is straightforward, while the vectorization of the remaining part of the kernel is not possible because there is a complex chain of dependencies.

5.3.5 H264ref

In H264ref application, several benchmarks were vectorized. We present only the two kernels that were not straightforward to vectorize.

5.3.5.1 Kernel 1. FastFullPelBlockMotionSearch

The code below is the main part of this kernel which finds the minimum *mcost* for elements from the array *block_sad*. The vectorization of this kernel is not trivial because there is a dependency across iterations if the variable *mcost* is updated. There is also an *if* statement in the loop.

```
#define  WEIGHTED_COST(factor,bits)
                 (((factor)*(bits))>>LAMBDA_ACCURACY_BITS)
#define  MV_COST(f,s,cx,cy,px,py)
 (WEIGHTED_COST(f,mvbits[((cx)<<(s))-px]+mvbits[((cy)<<(s))-py]))
for (pos=0; pos<1089; pos++, block_sad++){
   if (*block_sad < min_mcost){
      cand_x = offset_x + spiral_search_x[pos];
      cand_y = offset_y + spiral_search_y[pos];
      mcost  = *block_sad;
      mcost += MV_COST (lambda_factor, 2,
                        cand_x, cand_y, pred_mv_x, pred_mv_y);

      if (mcost < min_mcost)
      {
         min_mcost = mcost;
         best_pos  = pos;
      }
   }
}
```

We explain the vectorization for this kernel with included strip-mining. The dependency across iterations is solved in the following way. We load MVL elements and compare them with current *min_mcost*. Then we count how many elements are less than current *min_mcost*. If this number is greater than a threshold (in our case MVL/2), then the remaining part of the computation for the current stripe of the array *block_sad* is executed in vector mode. Otherwise, it is executed in scalar mode. The *compress* instruction is used to avoid the problem with *if* statement. This instruction copies the elements of source vector register

according to the set bits of vector mask. Copied elements are stored consecutively in destination vector register. All the elements are loaded, but only the computation is performed just for iterations in which *if* statement is true. The vectorized code is shown below.

```
VL = MVL;
threshold = VL/2;
for (pos=0; pos < 1089; pos+=VL) {
   if ((pos + VL)>max_pos) { VL=max_pos-pos; wrapper_setvl(VL);}
   else wrapper_setvl(VL);
   temp_min = min_mcost;
   ldv R1, block_sad[pos];
   cmpvs_lt MR1, R1, min_mcost;
   popcountvm pop_cnt, MR1;

   if (pop_cnt < threshold){
      if(pop_cnt > 0){
         bitscanvvm R0, MR1;
         for (i=0; i<pop_cnt; i++){
            getelem R0, i, curr_elem;
            curr_cost = block_sad[pos+curr_elem];
            if (curr_cost < min_mcost){
               //--- scalar code ---
         }
      }
   }
   else{
      compress R0, R1, MR1;
      ldv R15, spiral_search_x[pos];
      compress R2, R15, MR1;
      ldv R16, spiral_search_y[pos];
      compress R4, R16, MR1;
      VL = pop_cnt;
      addvs R3, R2, offset_x;
```

```
      addvs R5, R4, offset_y;
      shopvs_l R3, R3, 2;
      shopvs_l R5, R5, 2;
      subvs R3, R3, pred_mv_x;
      subvs R5, R5, pred_mv_y;
      ldvgi R6, R3, mvbits;
      ldvgi R7, R5, mvbits;
      addv R8, R6, R7;
      mulvs R9, R8, lambda_factor;
      shopvs_r R9, R9, LAMBDA_ACCURACY_BITS;
      addv R10, R0, R9;
      vredmin temp_min, R10;
      cmpvs_le MR2, R10, temp_min;
      bitscanp temp_pos, MR2;
      if (temp_min < min_mcost){
         VL = MVL;
         bitscanvvm R11, MR1;
         min_mcost = temp_min;
         getelem R11, temp_pos, curr_elem;
         best_pos  = pos + curr_elem;
      }
   }
}
```

5.3.5.2 Kernel 2. SubPelBlockMotionSearch

In the code below, one part of this kernel is shown. The computation in the inner most loop is the only interesting part in the code for the vectorization. It subtracts four elements with stride of four from matrix *ref_pic* from four consecutive elements from one row of the matrix *orig_pic* and store results to the array *d*. The computation is repeated for the next three rows of matrices *orig_pic* and *ref_pic*.

```
for (y0=0,abort_search=0;y0<blocksize_y && !abort_search;y0+=4){
   ry0 = ((pic_pix_y+y0)<<2) + cand_mv_y;
   for (x0=0; x0<blocksize_x; x0+=4) {
```

```
      rx0 = ((pic_pix_x+x0)<<2) + cand_mv_x;
      d   = diff;

      orig_line = orig_pic [y0  ];    ry=ry0;
      *d++ = orig_line[x0  ] - ref_pic[IP_S*4+ry][IP_S*4+rx0];
      *d++ = orig_line[x0+1] - ref_pic[IP_S*4+ry][IP_S*4+rx0+ 4];
      *d++ = orig_line[x0+2] - ref_pic[IP_S*4+ry][IP_S*4+rx0+ 8];
      *d++ = orig_line[x0+3] - ref_pic[IP_S*4+ry][IP_S*4+rx0+12];

      orig_line = orig_pic [y0+1];    ry=ry0+4;
      *d++ = orig_line[x0  ] - ref_pic[IP_S*4+ry][IP_S*4+rx0];
      *d++ = orig_line[x0+1] - ref_pic[IP_S*4+ry][IP_S*4+rx0+ 4];
      *d++ = orig_line[x0+2] - ref_pic[IP_S*4+ry][IP_S*4+rx0+ 8];
      *d++ = orig_line[x0+3] - ref_pic[IP_S*4+ry][IP_S*4+rx0+12];

      orig_line = orig_pic [y0+2];    ry=ry0+8;
      *d++ = orig_line[x0  ] - ref_pic[IP_S*4+ry][IP_S*4+rx0];
      *d++ = orig_line[x0+1] - ref_pic[IP_S*4+ry][IP_S*4+rx0+ 4];
      *d++ = orig_line[x0+2] - ref_pic[IP_S*4+ry][IP_S*4+rx0+ 8];
      *d++ = orig_line[x0+3] - ref_pic[IP_S*4+ry][IP_S*4+rx0+12];

      orig_line = orig_pic [y0+3];    ry=ry0+12;
      *d++ = orig_line[x0  ] - ref_pic[IP_S*4+ry][IP_S*4+rx0];
      *d++ = orig_line[x0+1] - ref_pic[IP_S*4+ry][IP_S*4+rx0+ 4];
      *d++ = orig_line[x0+2] - ref_pic[IP_S*4+ry][IP_S*4+rx0+ 8];
      *d   = orig_line[x0+3] - ref_pic[IP_S*4+ry][IP_S*4+rx0+12];
   }
}
```

We can vectorize this four parts of the inner loop easily, but then the vector length will be just four. We decided to use vector memory load shape instruction (see section 3.2.2) in order to load all sixteen elements and therefore, increase the vector length to sixteen. The vectorize code is shown below.

```
for (y0=0,abort_search=0;y0<blocksize_y && !abort_search;y0+=4){
```

```
   ry0 = ((pic_pix_y+y0)<<2) + cand_mv_y;
   for (x0=0; x0<blocksize_x; x0+=4) {
      rx0 = ((pic_pix_x+x0)<<2) + cand_mv_x;
      d   = diff;

      VL = 16;
      ldvshape R1, orig_line+x0, 1, 4, 13, 0;
      ldvshape R2, ref_pic[IP_S*4+ry][IP_S*4+rx0], 4, 4, skip, 0;
      subv R3, R1, R2;
      stv R3, diff;
   }
}
```

5.4 Instruction level characterization

Before we define new vector architecture or propose adding some new functionality, it is very important to have detailed knowledge of the low level characteristics of vectorized applications. The same approach used Espasa [15] and Quintatna [34]. They claimed that in order to be able to reason about performance deficiencies of a program it is necessary to know in detail the resource usage made by the dominant parts of the program. Without this resource usage knowledge, it is very difficult to determine whether a performance problem could be easily solved by adding some extra functionality or whether the problem is more complex and requires a significant amount of work.

The CONVEX vector machine was the target platform used in their study, which has a fixed size of vector register, a fixed number of functional and memory units, etc. In our case, the size of vector register or the number of vector registers are parameters of the vector library that can be easily changed. Our vector ISA is also flexible, we can add new vector instructions and generate some new statistics with small changes in the vector library.

In this chapter, we present a detailed characterisation of the vectorized applications. In particular, we are interested in following measurements:

- **Percentage of vectorization :** We can determine the degree of vectorization by counting the number of scalar and vector operations. This degree tells us in general if a vector processor is a suitable choice for vectorized application.

- **Distribution of vector lengths :** An accurate measurement of the vector lengths used in vector computations is crucial to understand the interaction between latencies and performance. As it is known, larger vector lengths help in amortizing all kinds of latencies, particularly memory latency. Vector length also has an impact on the pressure on the fetch and decode unit. With longer vector lengths, lower number of instructions must be decoded and executed.

- **Distribution of Memory and Computation Instructions or Instruction Mix:** This measurement gives us information about the vector instructions executed in the vectorized application. We can determine the most executed instructions, as well as the ratio between memory and computation instructions. This information determines the resources that could be a potential bottleneck when the vectorized application is executed. For example, an application is memory bound if a particular configuration of vector processor has only one memory port and several ALU units and the ratio between memory and computation instructions is one. Memory bound means that memory unit will be the bottleneck when the vectorized application is executed.

- **Memory access patterns :** The most critical part of any vector machine is the memory system. The distribution of unit-stride, strided, and indexed vector memory instructions in vectorized applications has a high impact in micro-architectural design decisions.

- **Impact of vector register length :** The number and length of vector registers is a key decision in the design of a vector unit. Longer vector registers and the increased number of vector registers have several advantages as it is reported in [3], but for applications with natural vector lengths fewer

than the existing vector register length there is no improvements in performance with longer vector register lengths. Using different vector register lengths, we determine what is the optimum length for the set of vectorized applications.

5.4.1 Degree of Vectorization

Table 5.1 presents some basic statistics for the set of vectorized applications. The first column in this table contains the names of vectorized applications. SPEC [2] benchmark suite provides input data sets with different sizes: small (test), medium (train) and large (ref) input data sets. Some applications have several different ref input data sets; Hmmer has two different ref input data sets and H264ref has three different ref input data sets. Table 5.1 presents results for all input data sets. The next two columns contain the total number of executed instructions, broken down into scalar and vector instructions. In this study, we made a distinction between *operations* and *instructions*. A scalar instruction performs a single operation, while a vector instruction performs a varying number of operations, depending on the value of the vector length (VL) register. The next column presents the number of operations performed by the vector instructions (a column for the number for scalar operation is not needed because it is identical to the column that represents the scalar instruction counts). The fifth column is the percentage of vectorization of each application. We used the same metrics as Espasa et al [15]. The degree of vectorization of an application is defined as the ratio between the number of vector operations and the total number of operation performed by the program (i.e., column four divided by the sum of columns two and four). The last column presents the average vector length used by the vector instructions, and is the ratio between vector operations and vector instructions (columns four and three, respectively). Note that these results are obtained for a maximum vector register length of 64.

The first interesting point from table 5.1 is the degree of vectorization. All our applications have a high percentage of vectorization. The degree of vectorization goes from 62.90% for H264ref up to 91.06% for ECLAT. It is important to mention that we have just vectorized the most executed kernels and these numbers can be

Application	# of scalar instructions	# of vector instructions	# of vector operations	% of vectorization	avg. VL
FaceRec	2.1×10^{10}	2.4×10^{9}	9.4×10^{10}	81.81	38.7
Sphinx3	3.6×10^{11}	3.7×10^{10}	1.7×10^{12}	82.47	46
ECLAT	1.7×10^{8}	3.8×10^{7}	1.7×10^{9}	91.06	59.1
Hmmer	2.2×10^{11}	8.5×10^{9}	5.1×10^{11}	70.08	59.9
	5.2×10^{11}	2.1×10^{10}	1.1×10^{12}	66.76	49.8
H264ref	1.5×10^{11}	1.6×10^{10}	2.6×10^{11}	62.90	15.8
	1×10^{11}	9.3×10^{9}	2.1×10^{11}	67.47	22.7
	8.8×10^{10}	7.7×10^{11}	2.1×10^{12}	73.22	23.38

Table 5.1: The degree of vectorization for the set of vectorized applications.

improved by vectorizing other kernels in applications. The degree of vectorization also depends on the input data set. We can observe that Hmmer has different degree of vectorization for its two different input data sets, 70.08% and 66.76%. We investigated Hmmer application and found that we have different number of iterations in the most executed loop for different input data sets. For the first one, the number of iterations is 300, while for the second one is just 100.

The second interesting point from table 5.1 is the average vector length observed in applications. Even though, these applications are highly vectorizable, their average vector length varies a lot. Only Hmmer and ECLAT applications have the average vector length very close to the maximum vector register size. All other applications also have relatively long average vector length, except H264ref application which has a short average vector length.

In our applications, the average vector length depends on the input data set, as it is explained for Hmmer application above. The next section will present a more detailed study about vector lengths.

5.4.2 Distribution of Vector Lengths

The vector length used by the vector instructions in vectorized applications is a very important factor in achieving high performance in vector execution. Different

sizes of vector registers have been used in vector architectures. The higher number of register elements used during the computation is better, because all latencies (memory and functional units latencies and vector start-up costs) are amortized over the length of the vector register, but the utilization of resources can be the problem if the vector register is very large. Simply longer vectors achieve higher performance, because they have a lower associated overhead.

In this section, we present the effective usage of vector registers with 64 elements. In figure 5.2, we plot the cumulative percentage of executed instructions and the cumulative percentage of executed operations in vector instructions for each vector length used in a vector instruction. The X-axis plots the vector length value and the Y-axis plots the cumulative percentage of instructions and operations that have used a certain vector length. For example, for Sphinx3 we can see that about 15% of all vector instructions were executed with a vector length that was lower than 39, about 40% of all instructions were executed with vector length 39, and the remaining 45% of instructions were executed using a vector length equal to maximum, 64. We can also see that more than 60% of all executed operations in vector instructions were executed with maximum vector length of 64, about 35% of all operations were executed using a vector length of 39, and less than 5% of all operations were executed with a vector length that was lower than 39.

From figure 5.2, it is clear that the vector length distribution does not follow any regular pattern. Two applications (ECLAT and HmmerI) have the majority of their vector lengths clustered around 64 and have a small percentage of other vector lengths that are residuals generated due to strip-mining. The H264ref application for all input data sets has a dominant vector length of 16. The other programs use many different vector lengths. For all applications, except H264refI, more than 50% of all operations are executed using the maximum vector length of 64.

All in all, this data suggests that the utilization of the vector registers varies a lot. While some of the applications almost always use the 64 elements, the other could also run at a similar performance using shorter vector registers (H264ref application). It is also obvious that the distribution depends on the input data sets used to run applications (Hmmer and H264ref applications).

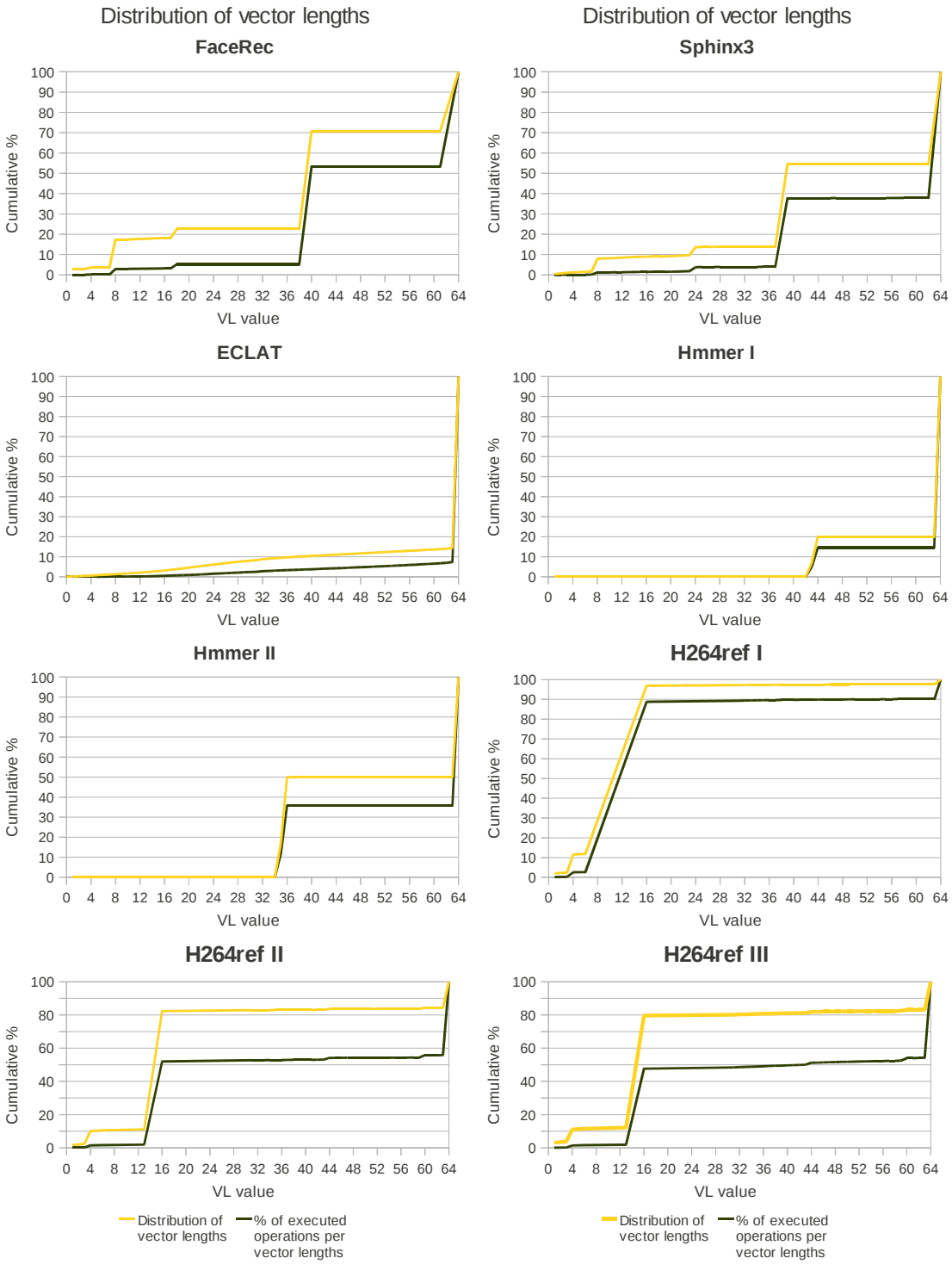

Figure 5.2: Distribution of vector lengths. X-axis is the VL value and Y-axis is a cumulative percentage distribution.

App	# of vector ins	Instructions % A&L	Mem	Red	B&E Man	# of vector ops	Operations % A&L	Mem	Red	B&E Man
FaceRec	2.4×10^{9}	41.0	34.4	13.1	11.5	9.4×10^{10}	43.9	39.9	12.3	3.8
Sphinx3	3.7×10^{10}	46.5	37.8	11.3	4.3	1.7×10^{12}	52.0	36.3	7.3	4.4
ECLAT	3.8×10^{7}	7.3	21.9	0.0	70.8	1.7×10^{9}	7.5	19.8	0.0	73.0
Hmmer	8.5×10^{9}	39.5	42.1	2.6	15.8	5.1×10^{11}	39.5	42.1	2.6	15.8
H264ref	1.6×10^{10}	31.8	48.8	13.4	6.0	2.6×10^{11}	32.5	46.8	13.8	6.9
	9.3×10^{9}	26.6	57.0	11.2	5.1	2.1×10^{11}	18.9	69.0	8.0	4.1
	7.7×10^{11}	26.3	56.0	10.8	6.9	2.1×10^{12}	19.5	76.8	7.6	5.0

Table 5.2: Distribution of vector memory and computation instructions and operations.

5.4.3 Distribution of Vector Memory and Computation Instructions

In table 5.2, we have included the distribution of vector instructions and operations for the set of vectorized applications. We have classified all instructions executed by each application into four categories. These four categories are: *arithmetic&logical*, *memory*, *reduction*, and *bit&element manipulation* instructions. The *arithmetic&logical* category includes all arithmetic and logical vector instructions presented in tables 3.1 and 3.2. The *memory* category includes all types of vector memory instructions. The *reduction* category includes all types of reduction instructions such as sum, sum sub-reduction, minimum and maximum. The *bit&element* manipulation category includes vector instructions that manipulate in the level of bits or elements in vector registers.

What we can see from table 5.2 is that instructions from *memory* category are in the range from 35% to 42% of all vector instructions, except for ECLAT application, in which this percentage is lower, about 22%. It means that if we have a configuration of the vector processor with more functional units than memory ports, it is likely that our applications will be memory bound.

We can also see that the distribution of vector instructions and vector operations for some benchmarks is the same (Hmmer), while for some other is not the same (Sphinx3, FaceRec). The reason why we have different distributions for

some benchmarks is that some vector instructions were executed with different vector lengths. For example, in FaceRec there is a lot of *setelem* and *getelem* instructions that operate on just one element of vector register and it is the reason why we have 11.5% vector instructions from this *bit&element manipulation* category and only 3.8% vector operations from the same category. On the other hand, in Hmmer only one kernel was vectorized and that is the reason why we have the same distribution for both vector instructions and operations.

The dominant computation category is *arithmetic&logical*, except for the ECLAT application where *bit&element manipulation* category is dominant with 70.8% executed vector instructions. During the process of vectorization one goal was to avoid *reduction* instructions wherever is possible, because reduction instructions are expensive, but still reduction instructions are significant in Sphinx3, FaceRec, H264refI and H264refII applications with the range between 10% and 15%.

Table 5.2 presents the distribution of vector instructions and operations just for one input data set for Hmmer application, because we observed that the distribution is the same for the different input data sets. It is not case for the H264ref application. We have almost the same distribution of vector instructions and operations for second and third input data sets, while distribution of vector instructions and operations is different for the first data input set. In the following figures, we didn't show the distribution of vector instructions for the third data input set of H264ref, because it has the same distribution as second data input set.

To understand the distribution of vector instructions better, we present the detailed distribution of vector instruction for each category. The figure 5.3 presents the distribution of vector instructions from *arithmetic&logical* category. The dominant instructions are multiplication and addition (including subtraction) in all applications, except ECLAT which contains only *compare* instruction in this category.

Overall, additions and multiplications appear to be approximately in 1 to 1 proportion, except for the H264ref application where more than 90% are add-like instructions, and for a well balanced architecture (in terms of functional units),

the number of units able to perform a multiplication and the number of units able to perform add-like instructions should be the same.

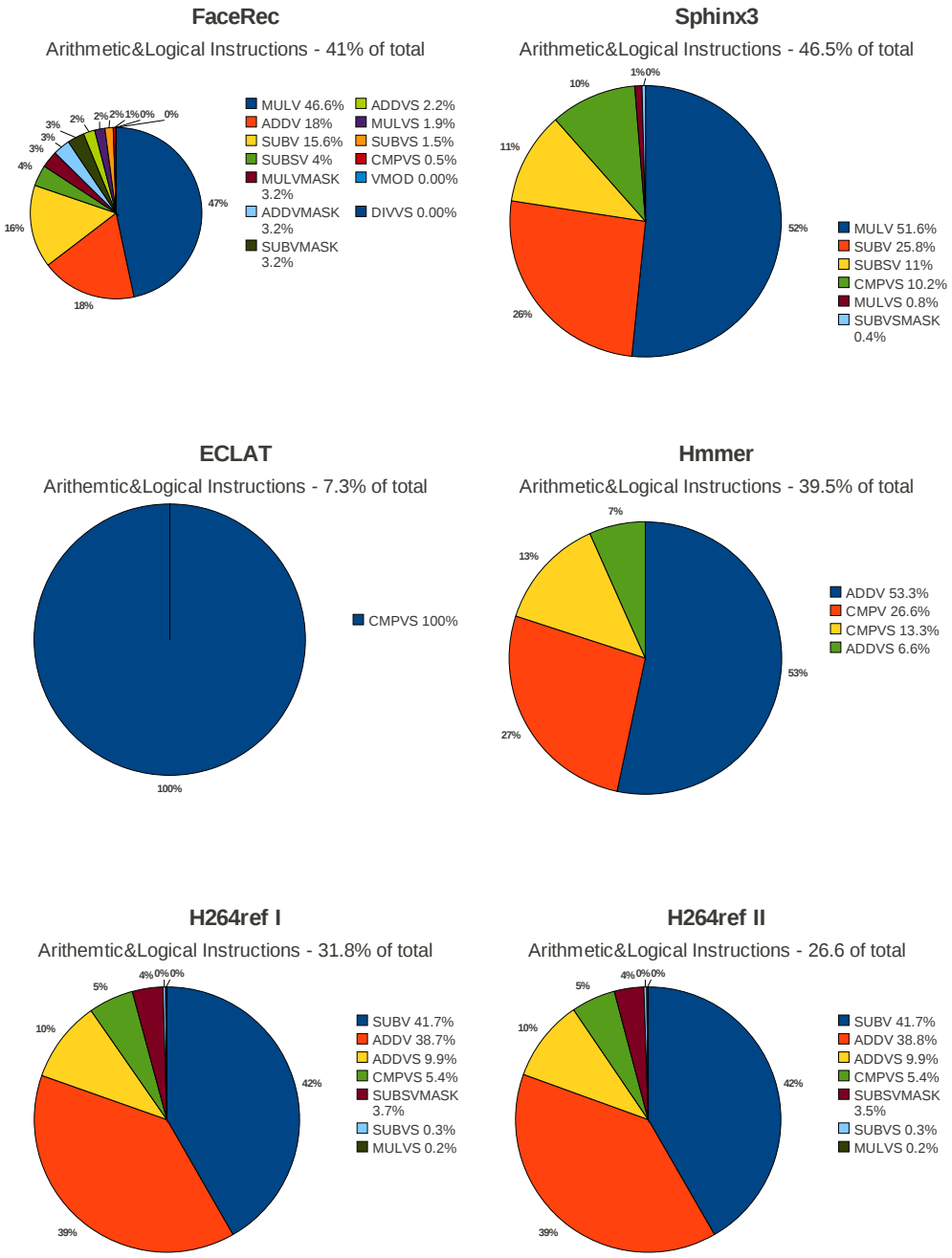

Figure 5.3: Distribution of arithmetic and logical vector instructions.

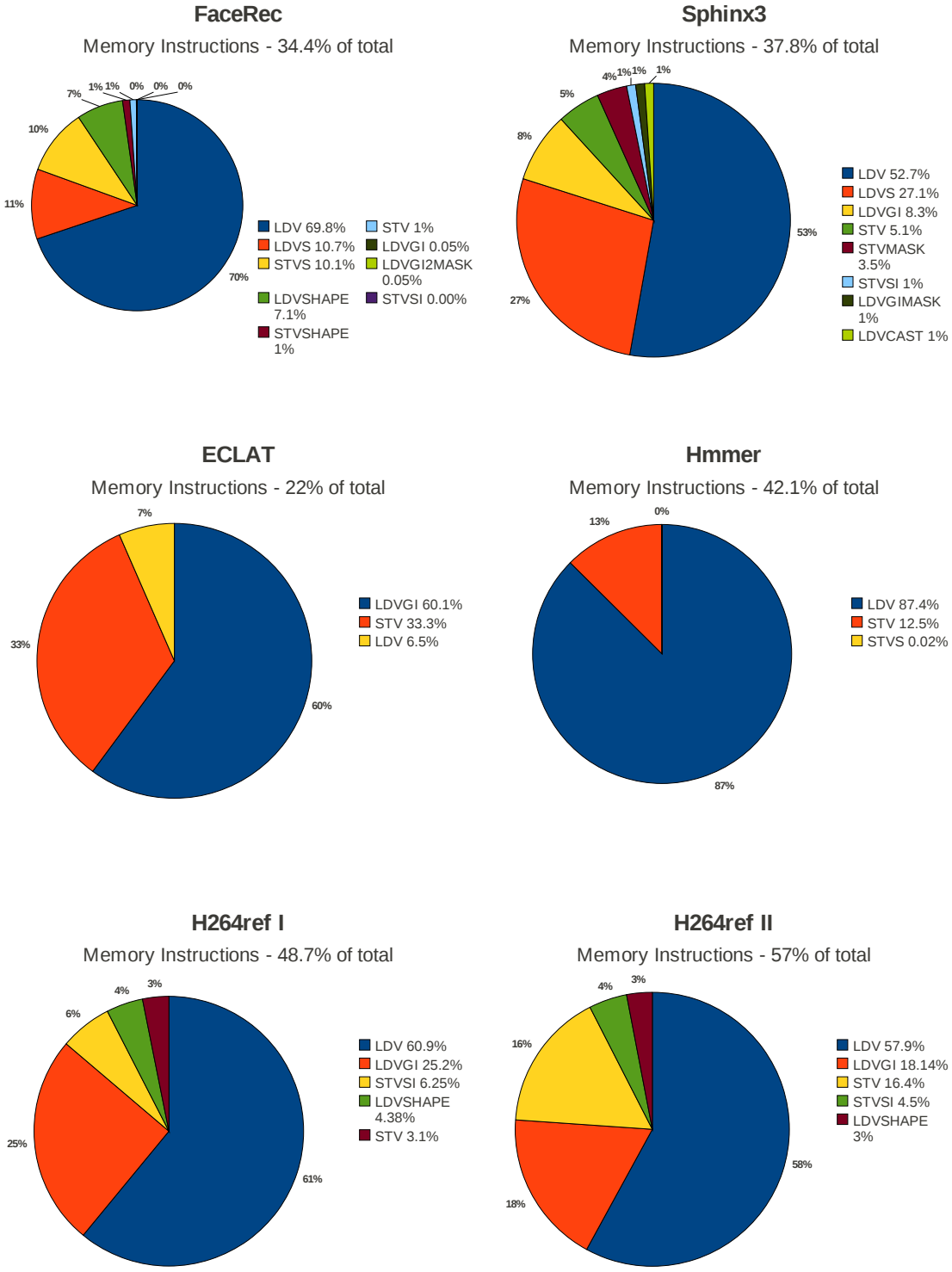

Figure 5.4: Distribution of vector memory instructions.

Figure 5.4 presents the distribution of vector memory instructions. We can see that vector load instructions are dominant compared to vector store instructions. It is interesting that for all applications vector store instructions are in the range from 10% to 15%, except for ECLAT where we have 33.3% of vector store instructions.

Unit-stride vector memory instructions are dominant in all applications, except in ECLAT. In Hmmer application, almost all memory instructions are unit-stride, more than 99%. Strided memory instructions play an important role in Sphinx and FaceRec with 27.1% and 20.8%, respectively.

Memory shape instructions are used in FaceRec, H264refI and H264refII applications with 8.1%, 4.3% and 3%, respectively. They helped in the vectorization of some kernels and the increase of the average vector length (FaceRec).

Indexed vector instruction, *scatter* and *gather*, were also executed in some applications. *Gather* instruction is dominant in ECLAT application with 60% of all memory instructions, and this instruction is also significant in H264refI, H264refII and Sphinx3 with 25.2%, 18.14%, and 9.3%, respectively. *Scatter* instruction is used in H264refI, H264refII and Spihnx3 applications.

Another interesting point in the figure 5.4 is the different distribution of memory instructions for different input data sets of H264ref application. We investigated H264ref and found that function *memcpy* is more significant for the second input data set because it copies larger blocks of data from one to another memory location. Since this function is vectorized using unit-stride vector load and store instructions, it reflects on a higher percentage of unit-stride vector store.

The distribution of reduction instructions is presented in figure 5.5. These instructions are significant in all applications, except in Hmmer with 2.6% and ECLAT which does not use reduction instructions (it does not appear in figure). *Sub-reduction* instruction is dominant in H264ref application with more than 90% and this instruction was also used in FaceRec and Sphinx3. *Sum* instruction was used in the same applications, but this instrucion is dominant in FaceRec and Sphinx3. Instructions that find the maximum or minimum in a vector are also used in Sphinx3 and H264ref.

Figure 5.6 presents the distribution of vector instructions from *bit&element manipulation* category. As we can observe from the figure, the usage and dis-

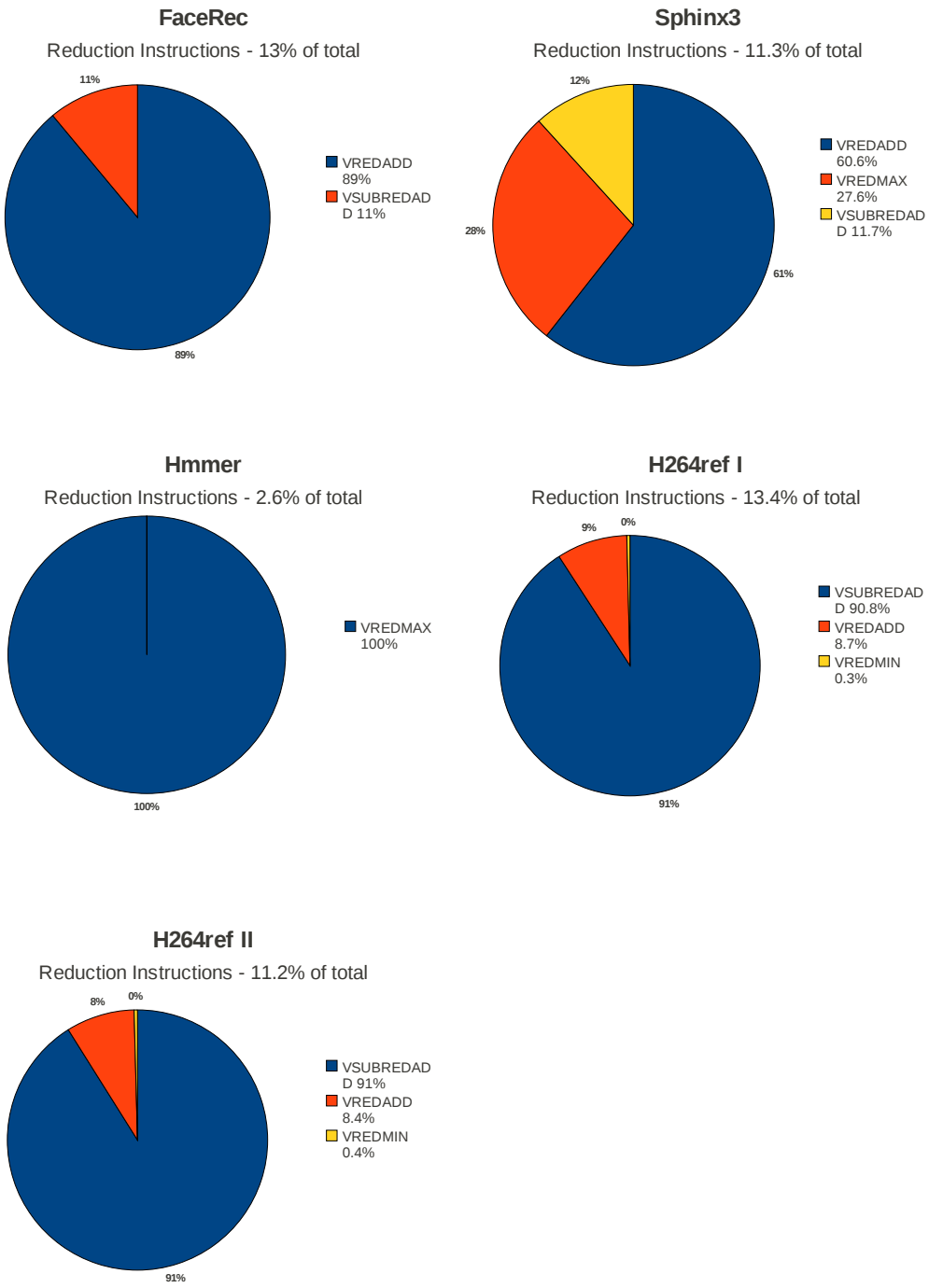

Figure 5.5: Distribution of vector reduction instructions.

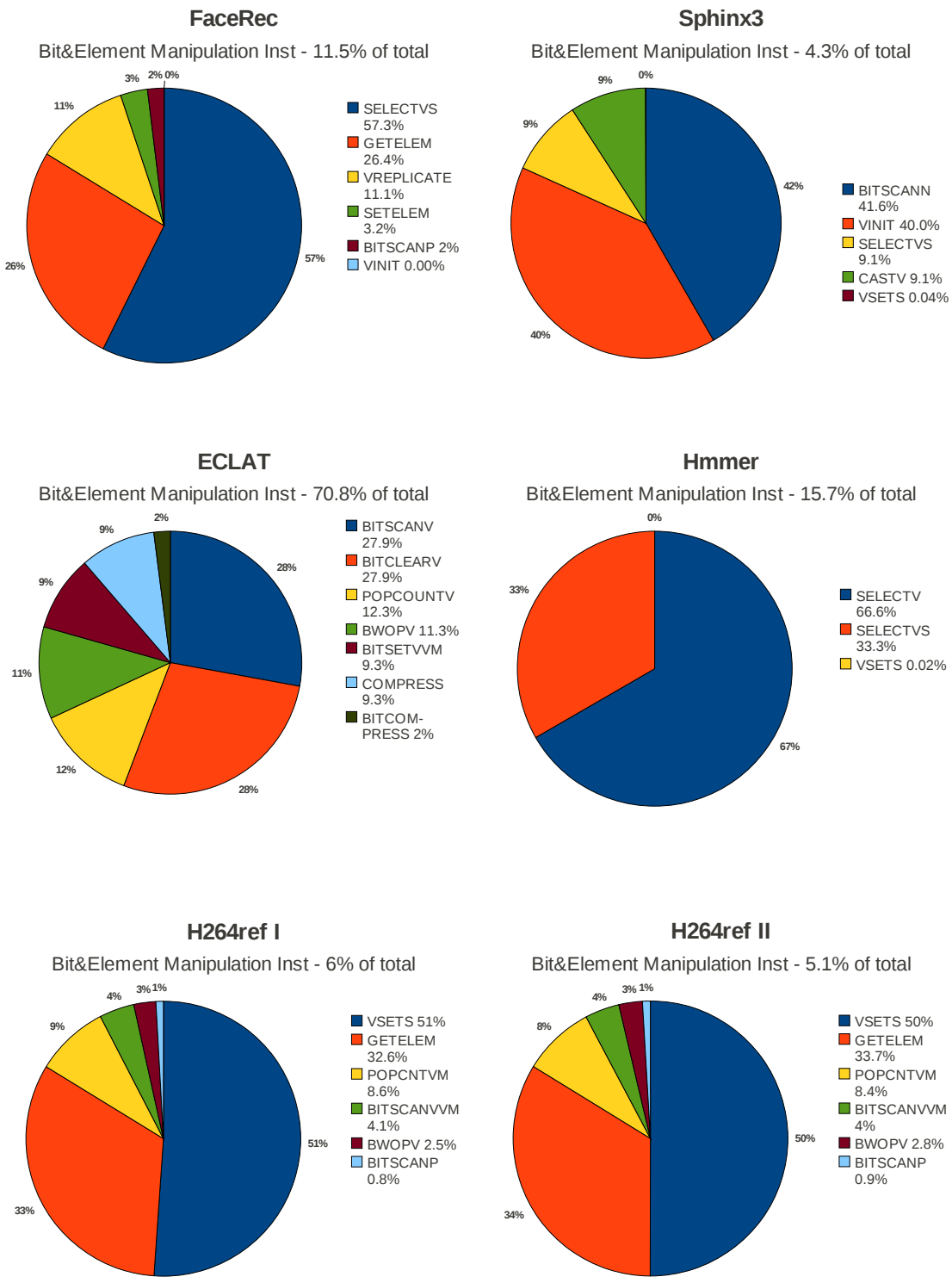

Figure 5.6: Distribution of vector bit and element manipulation instructions.

tribution of vector instructions from *bit&element manipulation* category vary a lot.

Instruction *select* is relevant in FaceRec, Sphinx3 and Hmmer applications. *Bitscan* instruction in different forms is relevant in all applications, except Hmmer, while *vsets* instruction is relevant in Sphinx3, H264refI and H264refII. *Getelem* instruction is relevant in FaceRec, H264refI and H264refII.

5.4.4 Distribution of Data Types

The usage of data types in computation for an application is very important for determining the types of functional units in a processor. Tables 5.3 and 5.4 present the distribution of data types for computational vector instructions in our vectorized applications and memory vector instructions, respectively.

ECLAT and HmmerI use only integer data type of 32 bits. H264ref also uses only integer data types, but of different sizes: 16, 32 and 64 bites. FaceRec and Sphinx3 are floating-point applications. FaceRec only uses 32 bits float and integer data types, while Sphinx3 uses floating-point data types: (64 bits) and float (32 bits); and integer data types of 16 and 32 bits.

Hmmer, ECLAT and H264ref require only ALU units with support for integer data types. FaceRec and Sphinx3 require a lot of support for floating-point data with single precision, while Sphinx3 also needs support for floating-point data with double precision.

We counted the distribution of data types using destination type. The conversion is possible in some vector instructions. That is the reason why the Sphinx3 has 0% of double data in memory instructions, but 63.8% double data in the computational instructions.

5.4.5 Vector Stride Distribution

The vector stride, used in the vector memory access, is an important metric of the vectorized programs. The vector stride is the number of elements that separate two consecutive elements of a vector memory access.

Unit-stride memory access provide the best performance results when memory hierarchy consist of a cache hierarchy because it results in a better utilization

type	FaceRec	Sphinx3	ECLAT	HmmerI	H264refI
int16	0.0%	3.9%	0.0%	0.0%	21.2%
int32	19.3%	11.9%	0.0%	100%	77.5%
uint32	0.0%	0.0%	100%	0.0%	0.0%
uint64	0.0%	0.0%	0.0%	0.0%	1.3%
float	80.7%	20.4%	0.0%	0.0%	0.0%
double	0.0%	63.8%	0.0%	0.0%	0.0%

Table 5.3: Distribution of data types for computational vector instructions.

type	FaceRec	Sphinx3	ECLAT	HmmerI	H264refI
int16	0.0%	8.5%	0.0%	0.0%	57.7%
int32	1.3%	14.4%	0.0%	100%	39.0%
uint32	0.0%	0.0%	100%	0.0%	0.0%
uint64	0.0%	0.0%	0.0%	0.0%	3.3%
float	98.7%	77.0%	0.0%	0.0%	0.0%
double	0.0%	0.0%	0.0%	0.0%	0.0%

Table 5.4: Distribution of data types for memory vector instructions.

of the memory bandwidth. In unit-stride memory access, the full cache line is delivered to the processor, while for strided memory access it is not case. A memory access with stride two only uses half of the elements in the cache line. When the stride is larger than the number of elements in a cache line, each accessed cache line provides only one element. It minimizes the benefits of exploiting the memory bandwidth.

Figure 5.7 presents the vector stride distribution for the set of vectorized application. It does not include scatter/gather vector instructions. Some of the applications, like ECLAT, Hmmer and H264ref, execute the majority of their memory access with stride equal one. In this case, applications directly benefits from the memory bandwidth. In FaceRec, the most significant memory accesses are with stride one and two. This application also has a small percentage of memory accesses with stride 8. In Sphinx3, all memory accesses were performed with

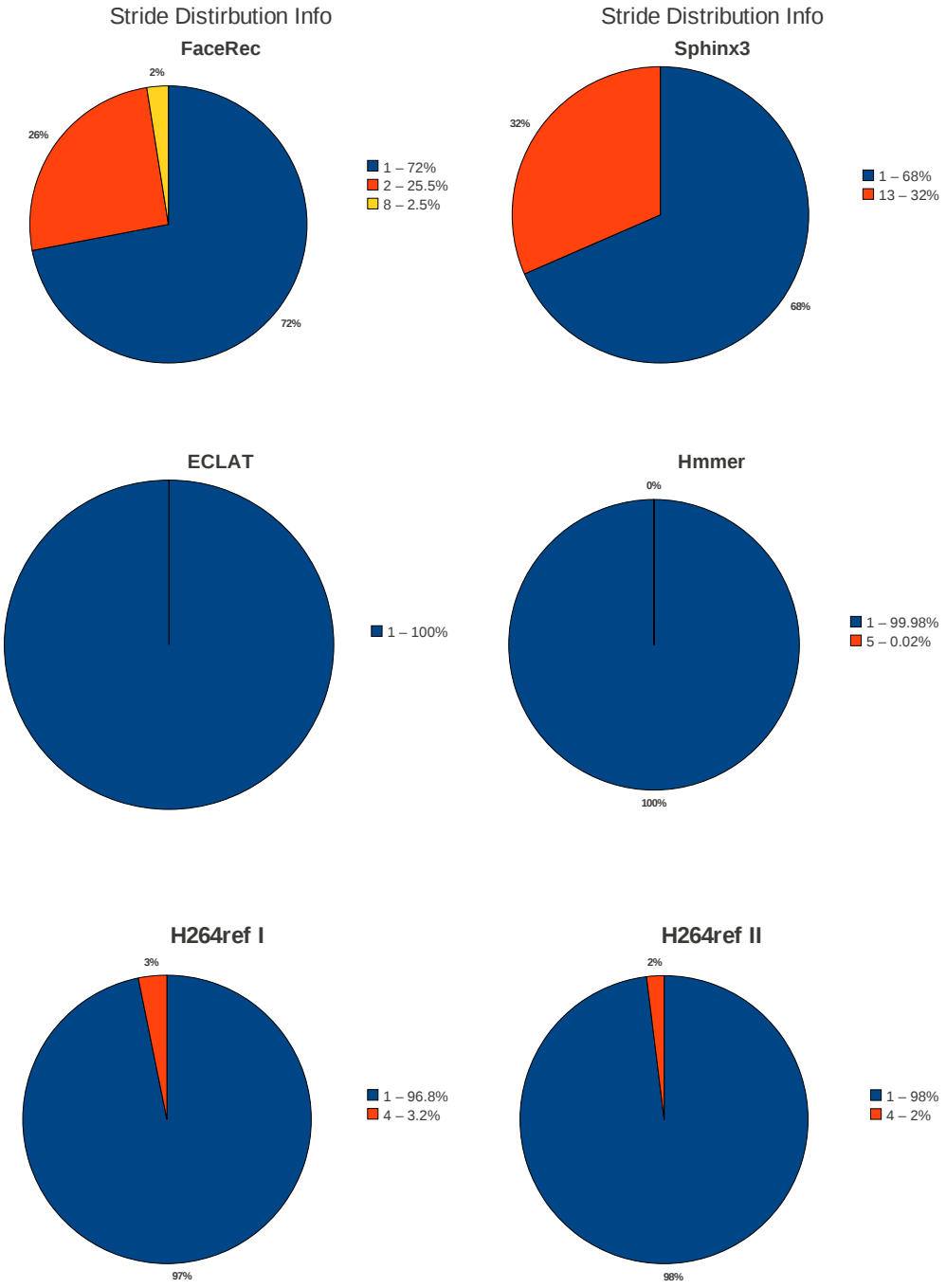

Figure 5.7: Distribution of strides for vector memory instructions.

strides one and 13. Applications that execute memory accesses with unit-stride and strides greater than one, and indexed memory accesses will have performance loss due to memory hierarchy with caches.

5.4.6 Impact of maximum vector register length

As stated at the beginning of this chapter, the maximum length of vector register is a key decision in the design of a vector unit. A larger vector length potentially decreases the number of executed vector instructions and scalar operations, but increases the chip area that is a critical parameter in the design of any microprocessor.

In this section, we make a study about the behavior of the vectorized applications when varying the maximum vector length from 16, to 32, 64, 128 or 256 elements. The study has been made by measuring the degree of vectorization and the number of executed vector instructions and operations with different vector lengths.

As the vector length increases, the total number of executed operation decreases due to the lower number of loop iterations carried out in each vectorized loop where strip-mining is performed. This effect should increase the degree of vectorization. The table 5.5 presents the degree of vectorization (see section 5.3.1) using different maximum vector lengths. As was it expected, the degree of vectorization increases for FaceRec, Sphinx3 and H264ref applications if we increase maximum vector length. The degree of vectorization is the same for all maximum vector lengths in Hmmer and ECLAT applications. We investigated these two applications. We found that only one kernel is vectorized in both applications and the control scalar instructions in vectorized loop were discarded (not included into instruction trace) because the threshold (section 3.6) was high.

Figure 5.8 presents the vector length distribution for the set of vectorized application, when varying the maximum vector length. We plot the cumulative percentage of executed vector instructions for each vector length used in a vector instruction. The X-axis plots the vector length value and the Y-axis plots the cumulative percentage of instructions that have used a certain vector length.

Application	Maximum Vector Length				
	16	32	64	128	256
FaceRec	78.68%	80.83%	81.81%	81.93%	81.98%
Sphinx3	80.54%	81.40%	82.46%	83.57%	85.36%
ECLAT	91.06%	91.06%	91.06%	91.06%	91.06%
Hmmer	71.82%	71.82%	71.82%	71.82%	71.82%
	69.12%	69.12%	69.12%	69.18%	69.18%
H264ref	64.89%	62.76%	62.89%	63.42%	64.54%

Table 5.5: Degree of vectorization for different maximum vector register lengths.

As we can see, the vector length distribution follows several patterns. ECLAT has most of its vector lengths concentrated around maximum vector length, but it does not cause a huge increase in the number of instructions with the maximum vector length, as the maximum vector lengths decreases. It means that usage of vector registers is almost independent of maximum vector length.

HmmerI and HmmerII have a single dominant vector length, which is the number of iteration in the vectorized loop, i.e. 300 and 100, respectively. When the maximum vector length is 256, an instruction with 300 operations in HmmerI must be carried out by using two vector instructions; one with vector length 256 and the other with vector length 44. This is reason for the step in figure 5.8 for VL 256 plot. When the maximum vector length is set to 128, each instruction with vector length 256 is carried out two instructions with vector length 128, and there is one more instruction with vector length 44. This is the explanation of the step in the VL 128 plot. For VL 64, 32 and 16 plots, the same phenomenon happens.

A single vector instruction can carry out 100 operations in a single go in HmmerII for maximum vector length 128 and 256. If we use shorter vector lengths, there is the same phenomenon as for HmmerI.

FaceRec and Sphinx3 have a distribution that follows a staircase, having several dominant vector lengths. As the maximum vector length decreases, the number of vector instructions that use the maximum vector length increases. In

Href264refI, the dominant vector length is 16, although there is a minor use of other values.

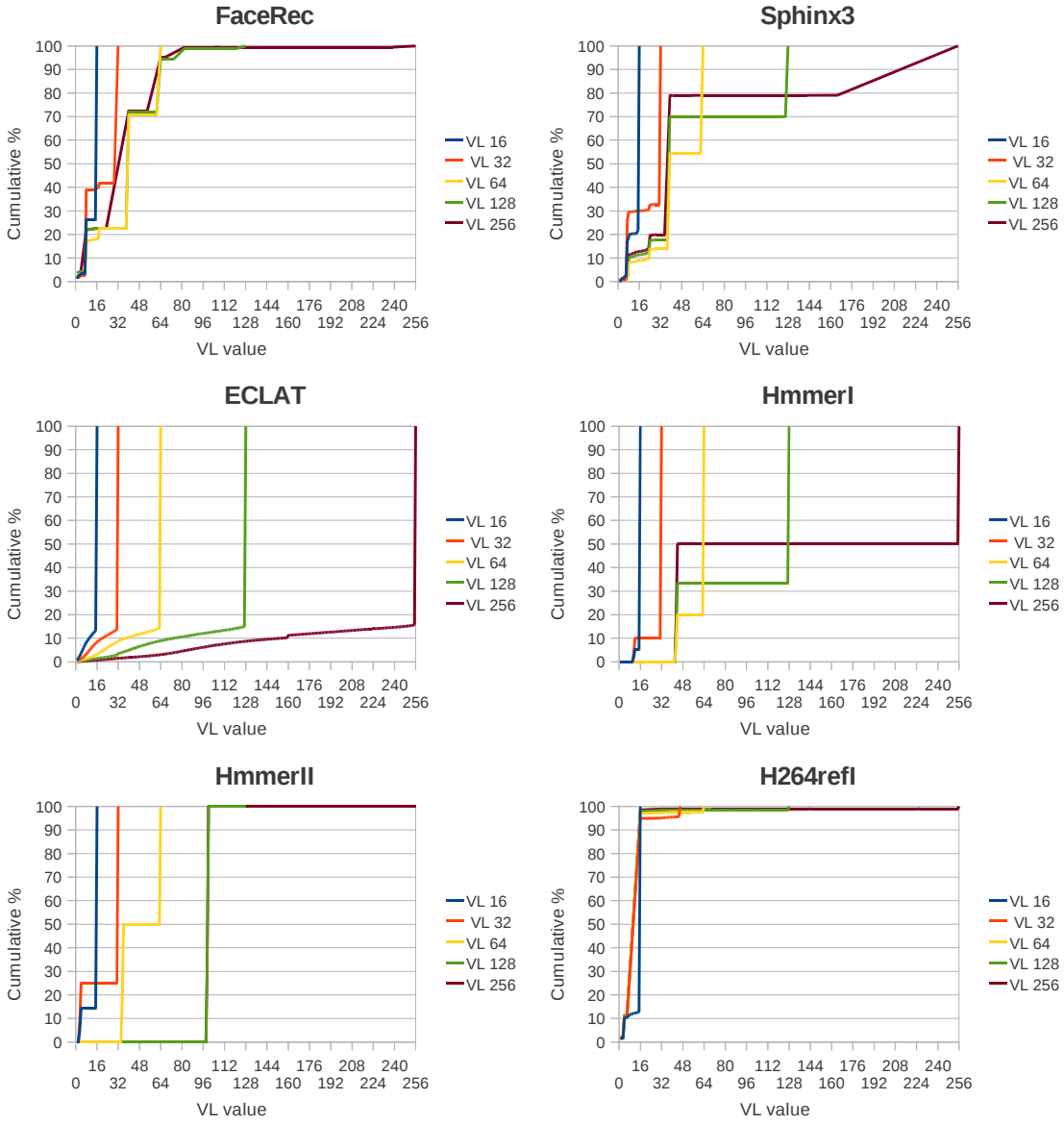

Figure 5.8: Vector length distribution for different maximum vector lengths.

Chapter 6

Timing Analysis

In this chapter, we study the impact of the parameters of the ETModel on an application's execution time. In particular we are interested in the following:

- Impact of memory latency and sizes of L1 and L2 caches. As it is well known, long vectors hide the memory latency. We want to check what is the behavior of the set of vectorized applications using different cache configuration and memory latencies.
- Impact of different configuration of functional units. It is very difficult to determine the best configuration of functional units. We want to see what is the impact on execution time of the set of vectorized application when using different configuration of functional units.

6.1 Memory latency and cache configurations

In this section, we are interested in the impact of memory latency on execution time of the vectorized applications, as well as, impact of different configuration of cache hierarchy.

In the experiments, we use three different memory latencies for main memory: 100, 200 and 400 cycles. Table 6.1 presents different configuration of caches used in our experiments. We use four different configurations, ranging from small caches (L1 2KB, L2 32KB) to very large caches (L1 1MB, L2 16MB). The L2 cache

#	L1 cache	L2 cache	L1 hit latency	L2 hit latency	L1 line size	L2 line size
1	2KB	32KB	1	6	32	64
2	16KB	256KB	1	6	32	64
3	128KB	2MB	2	6	32	64
4	1MB	16MB	4	6	32	64

Table 6.1: Different configurations of cache hierarchy.

is sixteen times larger than the L1 cache for all configurations, while every next configuration has eight times larger L1 and L2 caches than previous configuration. L1 cache hit latency is one cycle for the first two configurations, while two and four cycles are used for third and fourth configurations, respectively. The L2 cache hit latency, associativity, sizes of L1 and L2 cache line are the same for all configurations. All caches are 4-way set associative.

A fixed configuration of functional units with two ALU units and two memory units is used in all experiments. ALU units support all vector arithmetic and logic instructions with any data types, while memory units support all types of vector memory instructions. Instruction and address traces with the maximum vector length (MVL) of 32, 64, 128 and 256 are used as input traces for the experiments.

Figures 6.1 and 6.2 present execution times for the set of vectorized applications using different sizes of L1 and L1 caches and different latencies for main memory. Y-axis displays the number of cycles and X-axis the different configurations of main memory latencies and MVLs. We show the result for the best configuration (the fourth configuration or blue bar) and then stack difference between the third and the fourth, the second and the third, and finally between the first and the second configurations. A smaller number of cycles presents better result.

We can compare several things on these figures:

- Different cache configuration for the same MVL and main memory latency.
- Different MVLs for the same main memory latency and cache configuration.

- Different main memory latencies for the same MVL and cache configuration.

The first observation for all applications is that we have better execution time for larger caches for all MVLs and all main memory latencies. The only exceptions are HmmerI for main memory latency of 100 cycles and all MVLs, and H264refI for all combinations of latency and MVL. In HmmerI, the best execution is obtained for third cache configuration, as it is shown in figure 6.3. We analysed obtained statistics and found that we have very low L1 miss rate in both cases (less than 0.1%), but L1 cache hit latency is two cycles for third configuration and four cycles for fourth configuration. For the other two main memory latencies, fourth configuration is a little bit better because main memory latency has more impact on execution time (L2 miss rate is one order of magnitude smaller for fourth configuration - 0.007%).

In the H264refI application, the third configuration is the best for all latencies and MVLs, as figure 6.4 shows it. We analysed the statistics. The reason is again the same, but for this application, the L1 miss rate is even lower (0.001% for the fourth configuration).

With this very small L1 miss rate, the L1 hit latency has a bigger impact on the execution time than the main memory latency. It is the reason why the third configuration is the best for all latencies and MVLs. The second configuration is also better than the fourth one for a main memory latency of 100 cycles, because the second configuration has one cycle L1 cache hit latency.

The second interesting point for the set of applications is that there is very small difference between the third and fourth configuration. It means that L1 cache of 128KB and L2 cache is 2MB is enough for very good performance, further increasing does not improve so much the execution time.

One expected behavior is to have better execution time if we use the same main memory latency and cache configuration but larger MVL (FaceRec, ECLAT, HmmerI). It is not the case for Sphinx3 and H264refI. In the H264ref application, all vectorized kernels use the vector length of 16 or smaller lengths, except the Kernel1 described in the section 5.3.5 and the kernel that copies an array from one to another memory location. The second kernel does not have an impact on

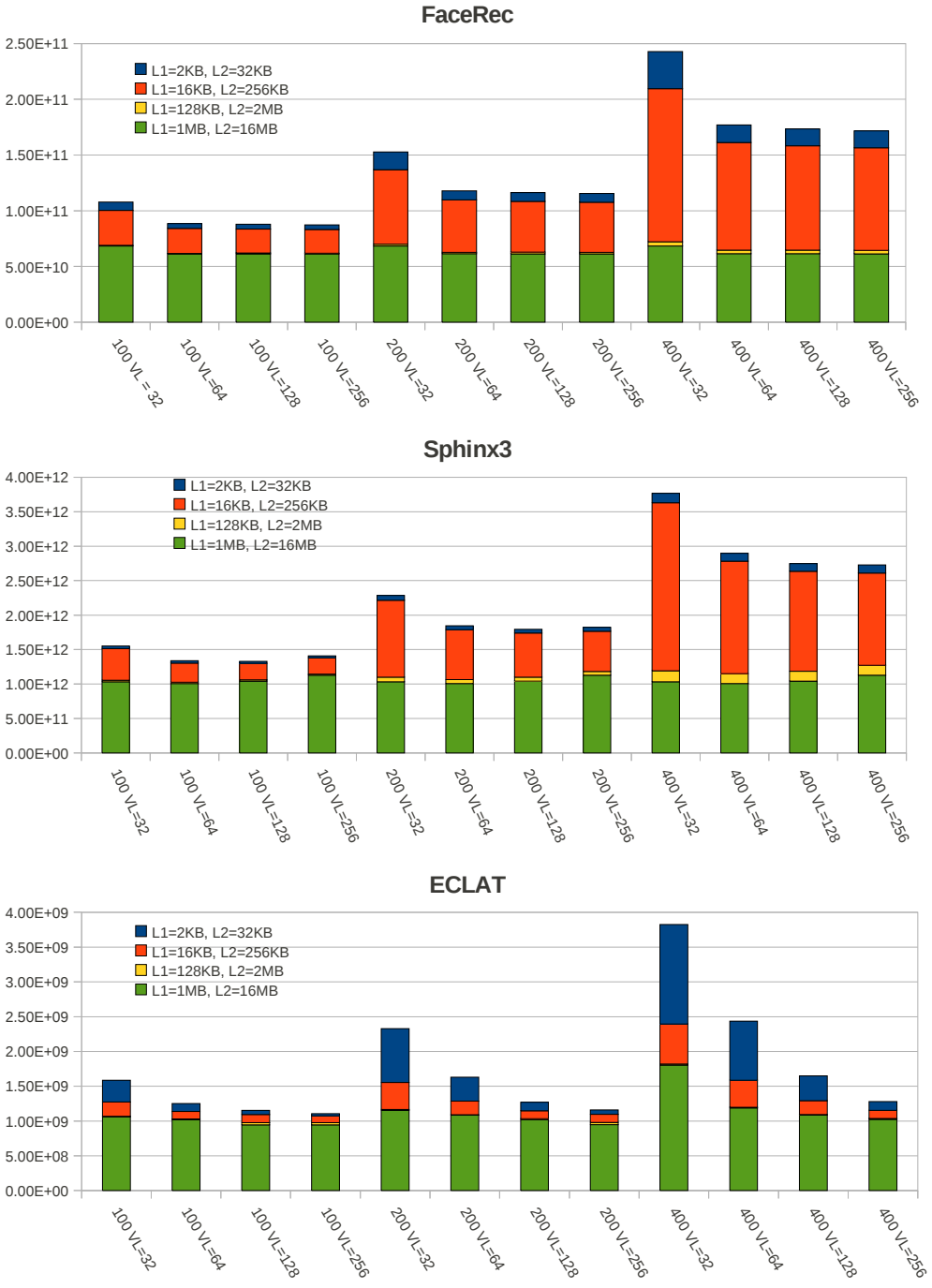

Figure 6.1: Application's execution time for different memory latencies and configurations of cache hierarchy for FaceRec, Sphinx3 and ECLAT.

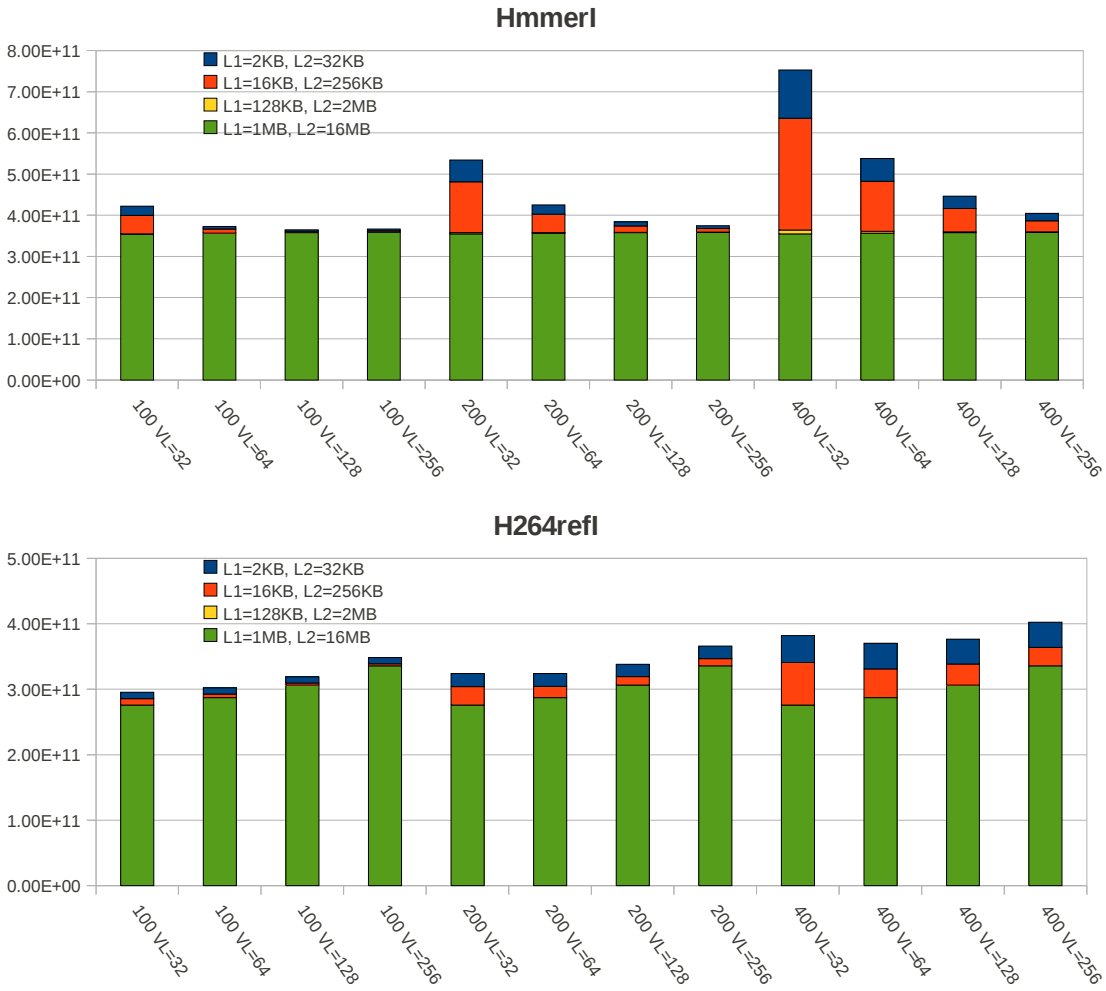

Figure 6.2: Application's execution time for different memory latencies and configurations of cache hierarchy for HmmerI and H264refI.

execution time if different MVLs are used. It means that only the first kernel has an impact on the execution time with different MVLs.

We analyzed the collected statistics of Kernel1 for MVL of 64, 127 and 256. The only difference that we observed is the number of executed vector operations. A larger number of vector operation is executed with longer MVLs. It means that the vectorized kernel causes the execution of useless vector operations. Our conclusion is that the number of useless instructions grows with larger MVL and decreases the performance.

Table 6.2 presents statistics for Sphinx3 using the fourth configuration of table 6.1 with a main memory latency of 100 cycles and three different MVLs: 64, 128 and 256. Two memory units and two ALU units are used in the experiments. It is obvious that chained and non-chained instructions are not the problem, as well as memory units. The ALU units can be the potential problem, because the time spent on waiting for free ALU unit is increasing if we increase the MVL.

After that, we did the same experiments, but with four ALU units. Statistics are presented in table 6.3 and again we have the worst results for the MVL of 256. Our hypothesis that the number if ALU units causes the problem is wrong. Then we noticed that the number of vector operations was increased 8% for the MVL of 128 and 24% for the MVL of 256 over the version with MVL of 64.

We investigated the vectorized kernels and found that two kernels (see sections 5.3.1: Kernel3 and Kernel4) cause this increase in the number of vector operations if we increase the MVL. The approach that we used for vectorization of these kernels in inefficient if we increase the MVL. The number of useless operations rapidly grow for larger MVLs.

The third interesting and expected point for applications is that if we just increase main memory latency and keep the same configuration and MVL, the execution time is also increased. We can also observe that larger caches and MVL longer than 64 very efficient hide main memory latency. There is very small difference in the execution time for different main memory latencies.

In FaceRec, the execution time is almost the same for all MVLs greater or equal to 64 and main memory latencies if large caches are used. There are two reasons for that: very large caches efficiently hide memory latency and just a

	VL 64	VL 128	VL 256
# of vector ins	37,356,562,539	30,379,155,993	26,856,952,827
# of issued chained ins	13,281,546,824 35.5%	11,294,402,511 37.1%	10,299,505,495 38.3%
# of issued non-chained ins	1,072,629,363 2.8%	1,043,396,498 3.4%	1,023,718,732 3.8%
Execution time	1,008,398,908,647	1,043,157,410,086	1,127,667,894,353
# of cycles waiting for non-chained ins	26,179,667,429 2.5%	25,957,119,671 2.4%	25,848,595,780 2.2%
# of cycles waiting for ALU units	393,743,461,782 39.0%	443,172,579,752 42.4%	534,759,004,284 47.4%
# of cycles waiting for mem units	4,043,538,904 0.4%	6,634,680,448 0.6%	10,158,002,143 0.9%

Table 6.2: Statistics for Sphinx3 application with two ALU and two memory units.

	VL 64	VL 128	VL 256
# of vector ins	37,356,562,539	30,379,155,993	26,856,952,827
# of issued chained ins	23,072,615,866 61.7%	17,441,777,540 57.4%	14,626,782,064 54.4%
# of issued non-chained ins	1,714,936,764 4.5%	1,659,553,915 5.4%	1,655,349,130 6.1%
Execution time	828,126,965,262	849,009,773,643	925,992,549,247
# of cycles waiting for non-chained ins	36,606,611,354 4.4%	36,501,315,238 4.2%	36,489,244,465 3.9%
# of cycles waiting for ALU units	9,044,243,498 1.1%	9,217,876,247 1.1%	9,333,152,357 1.0%
# of cycles waiting for mem units	45,434,665,765 5.4%	81,485,034,085 9.5%	100,470,022,047 10.8%

Table 6.3: Statistics for Sphinx3 application with four ALU and two memory units.

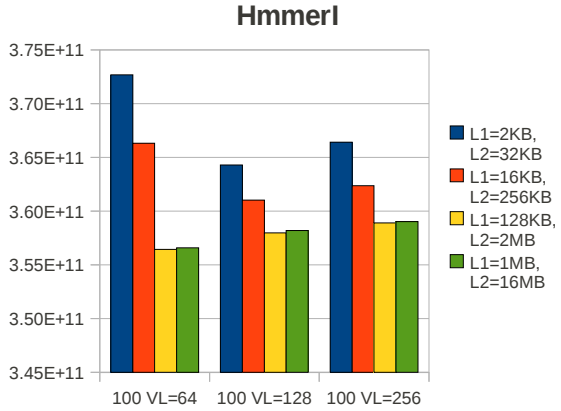

Figure 6.3: Execution time for HmmerI application.

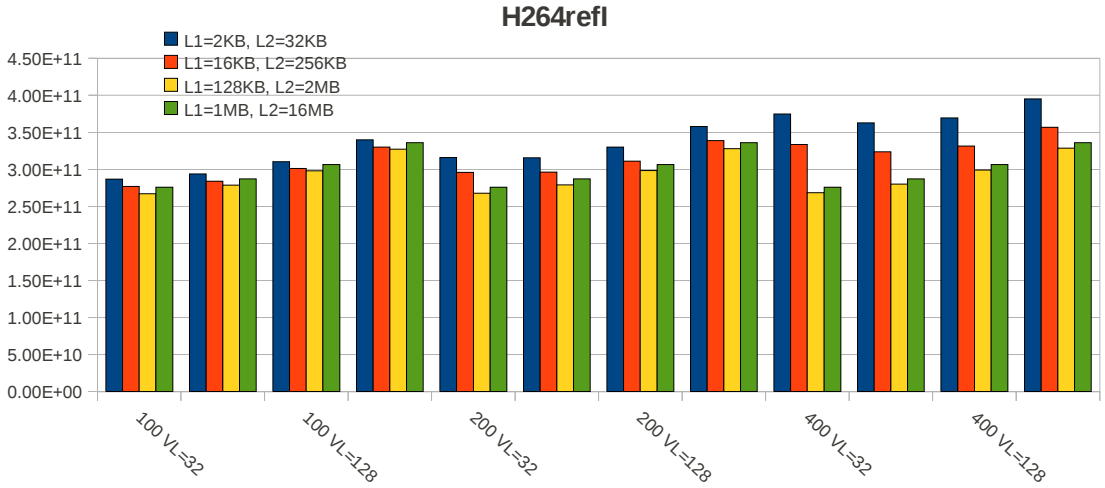

Figure 6.4: Execution time for H264refI application.

small number of instructions are executed with vector length larger than 64 (see figure 5.8).

In Hmmer and H264refI application, the different configurations of the cache hierarchy have a small impact on execution time, except for a main memory latency of 400 cycles.

A general conclusion is that a L1 cache of 128KB and a L2 cache of 2MB are enough to provide sustainable performance and hide main memory latencies for all applications for MVLs larger or equal to 64 elements.

6.2 Functional units

In this section, we explore different configurations of functional units. Table 6.4 presents four different configurations of functional units. For simplicity, in our experiments ALU units support all vector arithmetic and logic instructions with all data types, while memory units support all types of vector memory instructions. Instruction and address traces with the maximum vector length (MVL) of 32, 64, 128 and 256 are used as input traces for the experiments.

In all the experiments, the same configuration of cache hierarchy is used, with a L1 cache of 128KB and a L2 cache of 2MB. It is the third configuration in table 6.1. Again three different memory latencies for main memory (100, 200 and 400 cycles) are used.

Figures 6.1 and 6.2 present the execution time for the set of vectorized applications using different configurations of functional units. The Y-axis displays the number of cycles and the X-axis the different configurations of main memory latencies and MVLs.

The first interesting point is that the fourth configuration with two memory and two ALU units is the best in all applications. That is reasonable because it has more resources than the other configurations. The first configuration is always the worst because it has less resources than the other configurations. The third configuration with one memory unit and two ALU units is better than the second configuration with two memory units and one ALU unit in all applications, except in ECLAT for some configurations, as figure 6.7 shows.

Figure 6.5: Application's execution time for different configurations of functional units for FaceRec, Sphinx3 and ECLAT.

#	ALU units	mem units
1	1	1
2	1	2
3	2	1
4	2	2

Table 6.4: Different configurations of functional units.

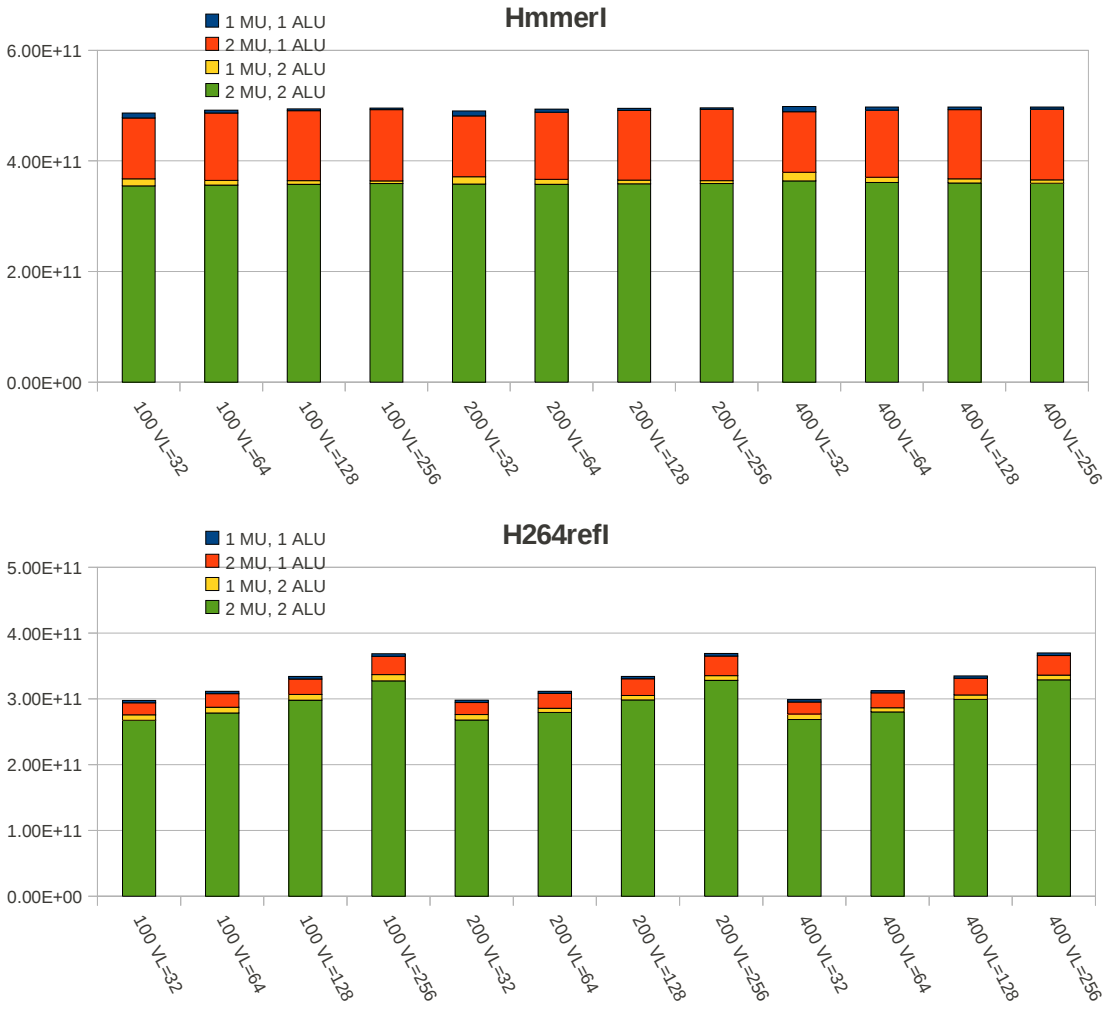

Figure 6.6: Application's execution time for different configurations of functional units for HmmerI and H264refI.

The ECLAT application has a lot of indexed memory instructions. The impact of indexed instructions cannot be hidden when the main memory latency is high and MVL is short. It is the reason why the second configuration performs better than the third for shorter maximum vector lengths and higher main memory latencies.

In Sphinx3, HmmerI and H264refI there is a very small difference between the fourth and the third configuration, as well between the second and the first configuration. It means that these applications are computational bound. An additional memory unit provides very small performance improvements.

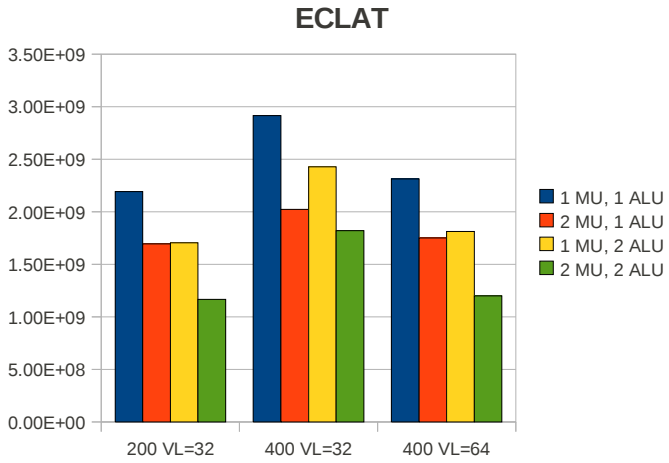

Figure 6.7: ECLAT's execution time for configurations of functional units where the second configuration is betther then the third.

In FacRec, the fourth configuration improves performance over the third configuration. It is also the case in ECLAT for configurations with smaller main memory latencies or large main memory latencies with longer maximum vector lengths.

Another interesting point is that, for the H264ref, there is a small difference between the best and the worst configuration.

Different maximum vector lengths and main memory latencies do not have a big impact for the fixed configuration of functional units, except for ECLAT.

Table 6.5 presents the usage of ALU and memory units for the different configurations of functional units. In all the experiments, the third configuration

from table 6.1 is used as cache hierarchy. Main memory latency is 100 cycles while the MVL is 64.

The first column in this table contains the names of vectorized applications. The second column contains the different configurations of functional units, the same as in table 6.4. The next four columns contain the percentage of usage for functional units used in the tested configurations. If a configuration does not have two ALU units or two memory units, "-" is in the field for the second ALU or memory unit. The sixth column contains the execution time presented in cycles. The last two columns contain the percentage of execution time for which computation or memory instructions wait for free ALU or memory unit.

The first interesting point is that if we use the the configurations with one ALU unit, all applications spend at least 46.6% of the execution time waiting for free ALU unit, except H264refI which spends 27.3% and 29.3% for the first two configurations of functional units, respectively. The second interesting point is that all applications spend less than 10% of the execution time waiting for free memory unit for all configurations of functional units, except for the third configuration in FaceRec, Sphinx3 and ECLAT.

If we use the third configuration from the table 6.4, the execution time is reduced for all applications. In the FaceRec, the third configuration reduces the waiting time from 47.6% for the first configuration to 20.2%. One more memory unit provides very small speed-up over the third configuration, while the waiting time for ALU units is almost the same.

In the Sphinx3, the third configuration causes almost the same behaviour as in FaceRec. Only difference is that the waiting time for free memory unit is dominant now. The fourth configuration removes the waiting time for free memory unit, but does not provide speed-up because the waiting time for free ALU unit is again significant (39.2% of the execution time).

The conclusion is the same for the ECLAT and HmmerI as for the FaceRec, if we use the third configuration. Only difference is that the waiting time for free ALU unit is still significant for the ECLAT (40.2% of the execution time). One more memory unit again provides very small speed-up over the third configuration. It reduces the waiting time for free memory unit but also increases the waiting time for free ALU unit.

The third configuration almost removes the waiting time for free ALU unit and the waiting time for free memory unit in the H264refI. The fourth configuration has very small impact on the execution time.

We can also observe that the second memory unit has the usage less than 5% in the HmmerI and H264refI.

All these data suggest that the vectorized applications are computational-bound. Therefore, the target vector processor should have more ALU units than memory units.

Apps	configu-ration	1st ALU	2nd ALU	1st mem	2nd mem	execution time	wait ALU	wait mem
FaceRec	1alu-1mem	65.0%	-	23.3%	-	9.16×10^{10}	47.6%	9.1%
	1alu-2mem	68.9%	-	15.9%	9.0%	8.63×10^{10}	50.6%	2.9%
	2alu-1mem	47.3%	43.6%	31.7%	-	6.75×10^{10}	20.2%	14.2%
	2alu-2mem	51.1%	48.3%	23.0%	12.1%	6.17×10^{10}	22.1%	5.6%
Sphinx3	1alu-1mem	74.2%	-	28.2%	-	1.56×10^{12}	62.6%	2.4%
	1alu-2mem	74.6%	-	25.6%	2.9%	1.55×10^{12}	63.9%	0.1%
	2alu-1mem	59.0%	55.3%	42.1%	-	1.05×10^{12}	18.7%	23.2%
	2alu-2mem	58.8%	57.4%	29.9%	14.8%	1.02×10^{12}	39.2%	0.5%
ECLAT	1alu-1mem	79.9%	-	28.4%	-	1.70×10^{9}	63.8%	9.5%
	1alu-2mem	85.6%	-	20.6%	11.8%	1.58×10^{9}	74.9%	0.7%
	2alu-1mem	63.5%	53.9%	41.3%	-	1.17×10^{9}	40.2%	21.0%
	2alu-2mem	72.3%	61.4%	32.3%	15.6%	1.03×10^{9}	45.8%	8.4%
HmmerI	1alu-1mem	58.1%	-	12.5%	-	4.91×10^{11}	46.6%	8.0%
	1alu-2mem	58.5%	-	9.4%	3.2%	4.86×10^{11}	53.3%	0.6%
	2alu-1mem	43.0%	36.0%	16.8%	-	3.64×10^{11}	27.0%	11.5%
	2alu-2mem	41.7%	38.7%	12.5%	4.7%	3.56×10^{11}	33.4%	1.2%
H264refI	1alu-1mem	50.3%	-	22.3%	-	3.11×10^{11}	27.3%	3.1%
	1alu-2mem	50.9%	-	17.9%	3.9%	3.07×10^{11}	29.3%	0.0%
	2alu-1mem	33.7%	24.1%	25.1%	-	2.87×10^{11}	4.4%	3.5%
	2alu-2mem	34.8%	24.8%	20.6%	4.4%	2.78×10^{11}	6.2%	0.0%

Table 6.5: The usage of of ALU and memory units for the different configurations of functional units.

Chapter 7

Related Work

7.1 Profiling and characterization of workloads

Espasa [15, 40] and Quintana [34] present a detailed instruction level characterisation of the selected programs from the Perfect Club programs [10], SPECfp92 benchmarks [2] including distribution of operations, average vector lengths and percentage of spill code in each program, etc. Quintana also included several benchmarks for the Mediabench suit [25], modified some benchmarks in order to get them to vectorize, and did manual strip-minig of all programs. They used a trace-driven approach to gather all the data. They compiled programs with the compiler for Convex C3400 [33] vector machine and then the instrumented output of the compiler with a tracing tool called Dixie [17] to produce the traces.

Janin develops vector simulation library for the purpose of his thesis [22]. The library implements many common instructions that are present in a register-based vector architecture and allows simulation of a subset of the features of a vector processor. The library does not simulate the performance of any particular architecture (e.g. cache, memory, chaining, etc.). The library was used to vectorize speech recognition algorithms.

Asanovic [3] presents the design, implementation, and evaluation of the first single-chip vector microprocessor (T0). He also proposes future vector microprocessor designs. He presents the results and statistics for several applications, which have been evaluated on T0. The presented statistics are similar to the

statistics presented in chapter 5 (e.g. distribution of vector lengths, vector register usage, ratio of arithmetic to memory instructions, etc.).

In our thesis, we have vectorized some new, non-traditional vector applications. Unlike most previous work, we didn't have access to any compiler for a vector processor. Therefore, we decided to develop the vector library and vectorize applications manually.

7.2 Vector ISA and vector micro-architecture

One of our goals was to implement common vector instructions. VMIPS [20], CRAY [1] and CONVEX [33] ISAs are used as a base for our vector instruction set. All ISAs are register-based vector instruction sets.

In the first version of the vector library, we implemented common vector instruction that we found in VMIPS [20], CRAY [1] and CONVEX [33] ISAs. When we started with the vectorization of the chosen applications, we also implemented some new instruction in order to vectorize some kernels or to improve some already vectorized kernels.

7.3 Analytical modelling

Karkhanis and Smith [23] proposed a performance model for superscalar processors which uses trace-derived data dependence information, data and instruction cache miss rates and branch miss-prediction rates as inputs. This model consists of a component that models the relationship between instruction issued per cycle and the size of the instruction window under ideal condition, and methods for calculating transient performance penalties due to branch misprediction, instruction cache misses, and data cache misses. The model can arrive at performance estimate that are within 5.8% of detailed simulation.

Eyerman, Eechkhout et al [18] extended the work done by Karkhanis and Smith [23]. In this work, interval analysis focuses on the flow of instructions through dispatch stage that leads to simpler formulation of proposed model, while the focus was on the issue stage in previous work.

Chen [11, 12] proposed techniques to predict the impact of pending cache hits, hardware prefetching, and realistic miss status holding register (MSHR) resources on superscalar performance in the presence of long latency memory systems when employing hybrid analytical models that apply instruction trace analysis, and presented techniques to estimate the performance impact of data prefetching.

Analytical performance modeling is the topic of one chapter in Eeckhout's book [14] in which he discussed three major flavors of analytical modeling: mechanistic modeling, empirical modeling, and hybrid mechanistic-empirical modeling. Mechanistic modeling or white- box modeling builds a model based on first principles, along with a good understanding of the system under study. Empirical modeling or black-box modeling builds a model through training based simulation results (e.g. regression model or a neural network). Finally, hybrid mechanistic-empirical modeling aims at combining the best of worlds: it provides insight (which it inherits from mechanistic modeling) while easing model construction (which it inherits from empirical modeling).

All these analytical models motivated us to develop own analytical model for vector processors, but during the process of development, we decided to create simple trace-driven simulator because we do not have any measurements of a vector micro-architecture that are necessary to create an analytical model. Some parts of our simulator are modeled in an analytical way. One of them is the simple memory model, where we modeled access to memory using parameter such as L1 hit rate, L2 hit rate, etc. Scalar instructions are also modeled in an analytical way using IPC as a parameter of the ETModel.

Chapter 8

Conclusion and Future Work

This thesis presented the vector library and ETModel, tools that help in rapid evaluation of micro-architectural requirements for the target applications, as well as a detailed analysis of the set of vectorized kernels. This section discusses the contributions and conclusions of this thesis.

8.1 Contributions

The main contributions of this thesis are the following:

- Developed a vector library. The library implements an ISA similar to VMIPS and also contains some new instructions such as sub-reduction and memory shape instructions. It also collects statistics of the vectorized applications and generates instruction and address traces of the vector instructions.
- Vectorized applications. We vectorized five non-traditional vector applications and provided a detailed description of their instruction level characteristics.
- Developed ETModel. The ETModel allows us to estimate the execution time and to perform a detailed analysis of micro-architectural requirements of our vectorized applications using traces.

- Detailed timing analysis. We performed a detailed timing analysis with emphasis on different main memory latencies, cache configurations, maximum vector lengths (MVL) and configurations of functional units.

8.2 Conclusions

The achievements and conclusions of this work are:

The vector library can help in the vectorization of desired application and provide detailed information related with vectorizable characteristics of a vectorized application. The library can be easily extended with new instructions if the process of vectorization requires that.

We vectorized five applications using the vector library. The process of vectorization was not trivial for most kernels and required a lot of effort. The results show that these applications are highly vectorizable; the highest degree of vectorization is 91% for the ECLAT while the lowest degree is 62.9% for the H264ref using the first ref input data set. The distribution of vector lengths varies a lot and depends on the input data sets used to at run time.

Memory instructions are in the range from 35% to 42% of all vector instructions, except for the ECLAT application, in which this percentage is lower, about 22%. The dominant computation category is *arithmetic&logical*, except for the ECLAT application where *bit&element* manipulation category is dominant with 70.8% executed vector instructions. Reduction instructions are significant in Sphinx3, FaceRec, H264refI and H264refII applications with the range between 10% and 15%. We will need a lot of bandwidths and a balanced number of functional units.

ECLAT, Hmmer and H264ref just use integer data types, while in FaceRec and Sphinx3 the dominant data types are floating point with single or double precision.

The ETModel uses instruction and address traces of the vectorized applications as inputs to estimate the execution time on the specified micro-architecuture of vector processor. We performed detailed analysis for different cache configuration, main memory latencies, MVLs and configuration of functional units.

The cache configuration with L1 cache of 128KB and L2 cache of 2MB (the third configuration in table 6.1) is enough to provide sustainable perfomance and to hide increasing of main memory latencies for applications for the MVLs larger or equal to 64 elements.

We also found that for Sphinx3 and H264ref we have worse perfomance when increasing the MVL. The reason for that is the approach used to vectorize some kernels. It causes the execution of useless vector operations. The number of these operations grows with larger MVL and decreases the performance.

The configuration with two ALU units and two memory units provide the best results among tested configuration. Different maximum vector lengths and main memory latencies do not a have big impact for the fixed configuration of functional units, except for ECLAT.

8.3 Future Research

This section discusses several directions of future work stemmed from this thesis. The possible directions are:

- Releasing of the vectorized applications. The idea is to release the vectorized applications as benchmark suit for vector processors.
- Real vector instructions. The idea is to automatically substitute the calls to vector library with real vector instructions to feed simulators.
- Scalar version of timing simulator. The idea is to implement simple scalar in-order simulator and estimate the speed-up of the vectorized applications over their sequential version.
- More timing analysis. The idea is to evaluate more parameters related with micro-architecture of vector processor. We would like to access directly to L2 cache with vector memory instructions, while scalar instruction will access to L1 cache.
- Improvement in some kernels. Use a different approach to vectorize kernels from Sphinx3 and H264ref that cause performance loss with longer MVLs.

Bibliography

[1] *Cray Assembly Language (CAL) for Cray X1 Systems Reference Manual*. 84

[2] Spec. The standard performance evaluation corporation. http://www.spec.org/. 51, 83

[3] K. Asanovic. *Vector Microprocessors*. PhD thesis, University of California, Berkeley, May, 1998. 4, 5, 11, 50, 83

[4] K. Asanovic and J. Beck. T0 engineering data. Technical report, Berkeley, CA, USA, 1997. 10

[5] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52:56–67, October 2009. 31

[6] R. Bodik, B. Christopher, C. Joseph, J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. Lester, P. John, S. Samuel, Williams W, K. Asanovic, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, EECS Department, University of California, Berkeley, December 2006. 31

[7] S. Browne, C. Deane, P. J. Mucci, and G. Ho. Papi: A portable interface to hardware performance counters. In *Department of Defense HPCMP Users Group Conference*, Monterey, 1999. 21

[8] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25:13–25, June 1997. 26

[9] M. Buxton, K. Nasri P. Jinbo, and N. Firasta. Intel avx: New frontiers in performance improvements and energy efficiency. Technical report, Intel Inc., 2008. 10

[10] D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Rolo, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, F S. Orszag, Seidl, O. Johnson, R. Goodrum, M. Berry, and J. Martin. The perfect club benchmarks: Efective performance evaluation of supercomputers. *The International Journal of Supercomputer Application*, January, 1989. 83

[11] X. Chen. Analytical modeling of modern microprocessor performance. Master's thesis, The University of British Columbia, 2006. 85

[12] X. E. Chen and T. M. Aamodt. Hybrid analytical modeling of pending cache hits, data prefetching, and mshrs. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 59–70, Washington, DC, USA, 2008. IEEE Computer Society. 85

[13] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi. The reconfigurable streaming vector processor (rsvptm). In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 141–, Washington, DC, USA, 2003. IEEE Computer Society. 16

[14] L. Eeckhout. *Computer Architecture Performance Evaluation Methods*. Morgan & Claypool Publishers, 1st edition, 2010. 85

[15] R. Espasa. *Advanced Vector Architectures*. PhD thesis, Universitat Politecnica de Catalunya, February, 1997. 2, 10, 49, 51, 83

[16] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and André Seznec. Tarantula: a vector extension to the alpha architecture. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 281–292, Washington, DC, USA, 2002. IEEE Computer Society. 10

[17] R. Espasa and X. Martorel. Dixie: a trace generation system for the c3480. Technical report, Universitat Politecnica de Catalunya, 1994. 83

[18] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst.*, 27:3:1–3:37, May 2009. 84

[19] S. Fuller. Motorolas altivec technology. Technical report, Motorola Inc., 1998. 10

[20] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4 edition, 2006. Appendix F. 2, 4, 5, 7, 10, 12, 13, 25, 26, 28, 84

[21] R. G. Hintz and D. P. Tate. Control data star-100 processor design. In *Compcon 72*, pages 1–4, New York, 1972, IEEE Computer Society Conf, 1972, IEEE. 10

[22] A. L. Janin. *Speech Recognition on Vector Architecture*. PhD thesis, University of California, Berkeley, Fall, 2004. 5, 83

[23] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 338–, Washington, DC, USA, 2004. IEEE Computer Society. 2, 25, 84

[24] M. Lades, J. C. Vorbruggen, J. Buhmann, J. Lange, C. von der Malsburg, R. P. Wurtz, and W. Konen. Distortion invariant object recognition in the dynamic link architecture. *IEEE Trans. Comput.*, 42:300–311, March 1993. 33

[25] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. *Microarchitecture, IEEE/ACM International Symposium on*, 0:330, 1997. 83

[26] R. B. Lee. Accelerating multimedia with enhanced microprocessors. *IEEE Micro*, 15:22–32, April 1995. 10

[27] R. Narayanan, B. Ö. Ikylmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *2006 IEEE International Symposium on Workload Characterization*, pages 182–188, 2006. 34

[28] S. Oberman, G. Favor, and F. Weber. Amd 3dnow! technology: Architecture and implementations. *IEEE Micro*, 19:37–48, March 1999. 10

[29] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 206–218, New York, NY, USA, 1997. ACM. 5

[30] A. Peleg and U. Weiser. Mmx technology extension to the intel architecture. *IEEE Micro*, 16:42–50, August 1996. 10

[31] S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, C. E. Kozyrakis, and K. Yelick. Scalable processors in the billion-transistor era: Iram. *IEEE Computer*, September, 1997. 10

[32] R. Perron and C. Mundie. The architecture of the alliant fx/8 computer. *Digest of Papers: Compcon 86*, pages 390–394, Washington, D.C., March, 1986. 10

[33] Convex Press. *CONVEX Architecture Reference Manual (C Series)*. Richardson, Texas, U.S.A., sixth edition, April 1992. 10, 83, 84

[34] F. Quintana. *Aceleradores Vectoriales para Procesadores Superescalaress*. PhD thesis, Universitat Politècnica de Catalunya, 2001. 49, 83

[35] R. M. Russell. The cray-1 computer system. *Commun. ACM*, 21:63–72, January 1978. 10

[36] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski,

T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM. 10

[37] J. E. Smith, G. Faanes, and R. Sugumar. Vector instruction set support for conditional operations. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 260–269, New York, NY, USA, 2000. ACM. 9

[38] Sony. PlayStation web page. http://www.us.playstation.com. 10

[39] S. Thakkar and T. Huff. Internet streaming simd extensions. *Computer*, 32:26–34, December 1999. 10

[40] M. Valero and R. Espasa. Instruction level characterization of the perfect club programs on a vector computer. In *XV International Conference of the Chilean Computation Society*, 1995. 2, 83

[41] W. Watson. The ti-asc, a highly modular and exible super computer architecture. In *AFIPS, 41, pt. 1:221-228*, 1972. 10

[42] R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, A. Fatell, G. W Hoeppner, D. Kruckmeyer, T. H. Lee, P. Lin, L. Madden, D. Murray, M. Pearce, S. Santhanam, K. J. Snyder, R. Stephany, J. Montanaro, and S. C. Thierauf. A 160mhz 32b 0.5w cmos risc microprocessor. In *ISSCC, Slide Supplement*, February, 1996. 5