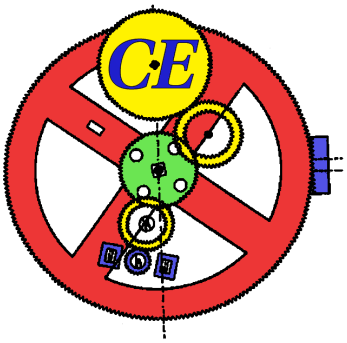


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2007

MSc THESIS

Customizing Vector Instruction Set Architectures

Cătălin Bogdan CIOBANU

CE-MS-2007-04

Abstract

Data Level Parallelism(DLP) can be exploited in order to improve the performance of processors for certain workload types. There are two main application fields that rely on DLP, multimedia and scientific computing. Most of the existing multimedia vector extensions use sub-word parallelism and wide data paths for processing independent, mainly integer, values in parallel. On the other hand, classic vector supercomputers rely on efficient processing of large arrays of floating point numbers typically found in scientific applications. In both cases, the selection of an appropriate instruction set architecture(ISA) is crucial in exploiting the potential DLP to gain high performance. The main objective of this thesis is **to develop a methodology for synthesizing customized vector ISAs** for various application domains targeting high performance program execution. In order to accomplish this objective, a number of applications from the telecommunication and linear algebra domains have been studied, and custom vector instructions sets have been synthesized. Three algorithms that compute the shortest paths in a directed graph (Dijkstra, Floyd and Bellman-Ford) have been analyzed, along with the widely used Linpack floating point benchmark. The framework used to customize the ISAs included the use of the Gnu C Compiler versions 4.1.2 and 2.7.2.3 and the SimpleScalar-3.0d tool set extended to simulate customized vector units. The modifications applied to the simulator include the addition of a vector register file, vector functional units and specific vector instructions. The main results of this thesis can be summarized as follows: overall applications speedups of 24.88X for Dijkstra (after both code optimization and vectorization), 4.99X for Floyd, 9.27X for Bellman-Ford and 4.33X for the C version of Linpack. The above results suggest a consistent improvement in execution times due to the customized vector instruction sets.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Customizing Vector Instruction Set Architectures

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Cătălin Bogdan CIOBANU
born in Braşov, România

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

Customizing Vector Instruction Set Architectures

by Cătălin Bogdan CIOBANU

Abstract

Data Level Parallelism(DLP) can be exploited in order to improve the performance of processors for certain workload types. There are two main application fields that rely on DLP, multimedia and scientific computing. Most of the existing multimedia vector extensions use sub-word parallelism and wide data paths for processing independent, mainly integer, values in parallel. On the other hand, classic vector supercomputers rely on efficient processing of large arrays of floating point numbers typically found in scientific applications. In both cases, the selection of an appropriate instruction set architecture(ISA) is crucial in exploiting the potential DLP to gain high performance. The main objective of this thesis is **to develop a methodology for synthesizing customized vector ISAs** for various application domains targeting high performance program execution. In order to accomplish this objective, a number of applications from the telecommunication and linear algebra domains have been studied, and custom vector instructions sets have been synthesized. Three algorithms that compute the shortest paths in a directed graph (Dijkstra, Floyd and Bellman-Ford) have been analyzed, along with the widely used Linpack floating point benchmark. The framework used to customize the ISAs included the use of the Gnu C Compiler versions 4.1.2 and 2.7.2.3 and the SimpleScalar-3.0d tool set extended to simulate customized vector units. The modifications applied to the simulator include the addition of a vector register file, vector functional units and specific vector instructions. The main results of this thesis can be summarized as follows: overall applications speedups of 24.88X for Dijkstra (after both code optimization and vectorization), 4.99X for Floyd, 9.27X for Bellman-Ford and 4.33X for the C version of Linpack. The above results suggest a consistent improvement in execution times due to the customized vector instruction sets.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2007-04

Committee Members :

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

I dedicate this thesis to my brother, Cristian

Contents

List of Figures

List of Tables

Acknowledgements

I would first like to thank Professor Stamatis Vassiliadis, as he made this thesis possible. I would like also to thank my advisors, Georgi Gaydadjiev and Georgi Kuzmanov for their patience and advice. Special thanks to Prof.dr.ing. Gheorghe Toacşe for his guidance.

A big thank you to all my friends and to my family back home, I couldn't of completed my work without your support.

Cătălin Bogdan CIOBANU
Delft, The Netherlands
June 12, 2007

1 Introduction

Increasing the performance of processors can be achieved at different levels. On the physical level, using smaller devices can lead to a larger transistor budget, and this can lead for example to having more functional units or increasing the cache size. On the micro-architectural level, one can redesign the arithmetic units or the memory hierarchy to gain performance. On the highest level of abstraction, the registers architecture and instruction set (ISA), specialized instruction set extensions that target a specific class of applications can be included in the design.

Processors that use vector instruction sets operate upon multiple elements with a single instruction (for example, two arrays can be added by executing a single vector add instruction). The order in which individual elements are processed is not explicit, and this allows the processor to choose an arbitrary processing sequence in order to obtain maximum performance. The key to high performance of this type of ISA is exploiting the data level parallelism (DLP) present in the application: in the case of scientific software, the data is organized in the form of long arrays of mostly floating point numbers while in the case of multimedia applications, short integer arrays are used.

While other types of parallelism exist, providing vector ISA extensions is critical when seeking high performance in applications with a large inherent data level parallelism.

1.1 The need for Vector ISA extensions

Exploiting different types of parallelism offers the chance of improving performance. Current superscalar CPUs exploit Instruction Level Parallelism (ILP) by executing as many independent instructions in the same time as possible. Chip multiprocessing (CMP) and simultaneous multithreading (SMT) are taking advantage of Thread Level Parallelism (TLP). Data Level Parallelism (DLP) can be exploited by introducing special operations at the ISA level to process certain elements in parallel. The high amount of available DLP in current applications makes the vector ISA extensions a good candidate for increasing the performance beyond the limits of ILP and TLP.

Using a vector ISA provides an efficient way of exposing the available DLP of the application. Applications with inherent DLP range from manipulating large matrixes (scientific software or telecommunication algorithms that process large graph structures) to streaming applications.

While data dependencies reduce the available DLP in a program, this is not the case in streaming applications (such as video encoding / decoding), as no dependencies exist between the loop iterations. This was the reason for creating the multimedia extensions, which are a specialized case of the general vector ISAs. The instructions are designed taking into consideration two main characteristics of the multimedia applications: the usage of mainly integer numbers, usually 8 or 16 bits wide and very short vector lengths

(mostly under 16 elements). Because audio and video is now at its peak, with everything going digital (for example music and internet TV [33]), multimedia vector extensions have been extensively studied and are actually implemented in a range of general purpose processors. The performance improvements in some kernels are impressive, while the hardware cost remains relatively low (e.g. a processor die area increase of around 10% for MMX [28]).

The multimedia extensions are not fundamentally different from the classic vector ISAs. It is possible to use an instruction set similar to the one found in IBM/370 [25] to perform video encoding for example, but the utilization of the functional units would pe poor, because pixel color information is normally stored using 8 bits per pixel while the integer arithmetic units are designed for 32 or 64 bit operands. Taking this fact into consideration, we will examine some existing multimedia extensions in detail as they represent an example of vector instruction sets.

Back in the early days of PC 3D graphics, a lot of articles were written asking an interesting question: which technology will win the market: multimedia extensions to current Instruction Set Architectures or custom 3D accelerator boards? Almost 10 years have passed, and actually we have a mixed answer to the question. 3D graphics accelerators are now providing the levels of performance needed for fluid realistic graphics, and it is clear that the early software 3D rendering engines are long gone. In the same time, most mainstream processors include multimedia extensions in the ISA. However, these extensions are focusing on increasing the CPU performance in audio video codecs found in high definition TV or blu-ray films.

We adopt the following definition of a vector processor from [13]: a vector processor is a machine designed to efficiently handle arithmetic operations on elements of arrays, called vectors. Most of the existing SIMD extensions are similar to vector instructions. However, some differences do exist [23, 15]:

1. Most existing SIMD extensions target multimedia applications by including support for narrow data types. Traditional vector architectures target high-performance computing, so no support for narrow data types is offered.

2. SIMD extensions operate on short, fixed length registers. Usually the SIMD registers are 64 or 128 bits wide. The SIMD operations can operate on up to 4 32-bit or 16 8-bit elements. Traditional vector registers are designed to store hundreds of 64-bit floating point numbers (typically 4096 bits or more). The register lengths are not usually fixed in the traditional vector architecture. Programs can read a control register to determine the length of the registers (the section size). The vector architecture also includes a vector length register which allow the programs to specify a shorter length on which to operate.

3. The memory access model of SIMD extensions is less complex compared to the traditional vector architecture, which includes hardware support for strided and indexed loads and stores.

The SIMD extensions can be regarded as a subset of the full-blown vector architectures. Both approaches capture the DLP by performing multiple operations concurrently

using a single instruction. The operation type is the same for all the elements processed, and the individual data elements are independent.

A well known example of a specialized ISA extension is MMX™ [28]. It targets the multimedia applications, as it was noted that narrow data types used by the multimedia applications to store information such as the pixel color information can be processed more efficiently if several elements are packed into one register and operated upon in parallel. This sub-word level parallelism provides a way to obtain large speedups with minimal hardware costs.

Intel introduced the MMX™technology for the Pentium 1 processor. This meant 57 new instructions providing support for fixed point arithmetic with narrow data types. Unfortunately, poor compiler support and slow adoption by software companies led to mixed results of the technology. AMD stepped up and provided it's own multimedia extension, the 3DNow!. This first showed to the world that improved performance is not impossible to obtain when appropriate of software optimization work is being involved. The software rendering in the popular 3D shooter game Quake2 showed an incredible 30-40% frame rate improvement with 3DNow! compatible processors. Still, if you bought a 3D accelerator, you could enjoy much higher frame rates and much better visual quality.

Basically every major manufacturer provided multimedia extensions to their ISAs. Intel provided MMX™, Intel Streaming SIMD Extensions (SSE), SSE2 and SSE3. The MIPS architecture has the MIPS digital media extensions - MDMX. Sun developed the Visual Instruction Set (VIS). The PowerPC used AltiVec™. Research is still being done in this field, with results being very promising.

The multimedia extensions provide support for a single instruction, multiple data (SIMD) architecture. One instruction performs the same operation on multiple data items. This is being done in parallel, and the performance gain is almost linear with the number of elements stored in one multimedia register. No longer are the functional units under-utilized.

Adding instructions to the ISA has some downsides. Compilers have to be adapted in order to utilize the new operations, and for increased performance gains the data structures used in the programs have to be carefully chosen. Also, modifying the ISA propagates changes to all levels, including the hardware implementation. Operating systems support is also required if extensions use additional register banks.

Multimedia extensions have been extensively studied. However, having a vector unit that operates with medium and long registers (hundreds of elements) has the potential to provide performance improvements in other applications that require full floating point precision or 32 and 64 bit integer values. In scientific applications, classic computers such as IBM/370 [25] provided Vector instructions sets long before the wide adoption of multimedia extensions.

1.2 Thesis objectives

The main objective of this thesis is **to develop a methodology for synthesizing customized vector ISAs**. To fulfill the main goal, the following additional objectives have been pursued:

- **Profile and vectorize applications and algorithms from the telecommunications and linear algebra domains**: The code profiling provides information regarding the most time-consuming sections of the code that are promising candidates for vectorization. Ideally, the applications should be automatically vectorized by a specialized compiler. As this is not the case, each application has to be manually modified in order to make use of the Vector ISA.
- **Synthesize custom Vector instruction sets for the chosen applications**: Each operation of the algorithm can require the addition of a specific class of instructions. A complete ISA will be then created, having both general purpose but also specialized instructions. The correct operation of the instruction set is verified by functional simulations using SimpleScalar.
- **Evaluate the performance gains of the custom Vector ISA over a traditional scalar instruction set**: Simulation results will be used to investigate the performance gains of the proposed instruction set for the applications considered in this work [14].

1.3 Results Summary

We experimented with SimpleScalar 3.0d, which was modified in order to support a set of vector instructions. The speedups obtained are due to the 24 vector instructions introduced (13 for Dijkstra, 11 for Floyd, 18 Bellman-Ford and 6 for Linpack). The targeted applications are three shortest paths in a directed weighted graph algorithms (Dijkstra, Floyd and Bellman-Ford) and the Linpack floating point benchmark. The application level speedups measured can be summarized as follows:

- A peak speedup of 24.88X for Dijkstra compared to the original version implemented with linked lists and pointers, and a speedup of 4.31X compared to the modified (but still scalar) version;
- A maximum speedup of 4.99X for Floyd;
- Speedups of up to 9.27X for Bellman-Ford;
- After vectorization, Linpack was up to 4.33X times faster.

1.4 Thesis organization

The thesis is organized as follows: in Chapter 2 some background information is given about existing multimedia vector extensions. In Chapter 3, the ISA customization framework is presented. The targeted applications are described in Chapter 4. Chapter 5 contains details regarding the experimental results, and Chapter 6 presents the conclusions.

Multimedia extensions examples

2

In this chapter the following multimedia extensions are described: MMX™, Intel Streaming SIMD extensions (SSE), MIPS V and MIPS Digital Media eXtensions, Sun's Visual Instruction Set, AltiVec™, Matrix Oriented Multimedia and Complex Streamed Instructions.

The distinctive features of each vector extension are presented, as well as performance gains in different kernels for each ISA extension. The Matrix Oriented Multimedia and Complex Stream Instructions are not implemented on a real processor, so the performance is estimated by using simulators.

2.1 The Intel extensions

Intel constantly improved the available SIMD extensions available to its x86 line of processors. The first multimedia extensions introduced by Intel are MMX™and SSE, and they are presented in this section. Each new generation of SSE extensions builds on the previous ones therefore the general approach is best understood by analyzing MMX™and SSE.

2.1.1 MMX™

MMX™was designed with specific goals in mind: improve the performance of the Intel architecture by 100% - 300% for multimedia kernels (the parts of the application where most time is spent) and a performance boost of 50% to 100% on complete multimedia applications [28].

When designing MMX™a decision had to be taken between providing a new set of registers for exclusive MMX™use, or share the existing register file. The choice was made to share the floating point register file, consisting of 8 80-bit registers between the floating point unit and MMX. This choice was made for several reasons. First, it didn't require the increase of the processor's die by adding new registers. An additional benefit consisted of no required support from the Operating System: if no new registers are added, the context switching procedure doesn't need to be modified in order to save the new SIMD registers. This meant that any existing operating system that supported the IA32 architecture could work with the new MMX-enabled processors. When a context switch happened, the OS saved the Floating Point registers anyway, so there was no difference between a normal Intel CPU and an MMX enabled one.

Such backwards compatibility meant that several restrictions had to be imposed. When using MMX instructions, it is not recommended to use floating point operations, because they shared the same registers. It was not impossible to do it, but results were unpredictable. The most significant bits of the 80-bit floating point registers were set to

a logical '1' when MMX was in use. This meant that when decoding a register used by MMX as being a floating point number, the data would be interpreted as Not a Number or infinity.

The 64 bit registers used by MMX could handle eight either 8-bi, four 16-bit, two 32-bit or one 64 bit fixed point integer numbers. The values stored in the registers were called "packed data types", because several narrow data types were packed into a single 64-bit wide register.

Instructions added included add, subtract, multiply, compare and shift. And because multiply and accumulate operations are very common in multimedia workloads, MMX provides a special multiply-add instruction.

Another important feature added is the saturation arithmetic support, which is very important in certain graphic processing algorithms. For example, when adding two medium-red pixels, the result should be a dark red not a pixel with a light red color (e.g. if pixel1 has the value of 120 and pixel2 has the value of 144 when the storage format is 8-bit unsigned integer, the result of the addition using saturation arithmetic is 255, while for wrap-around the result would be 9). Normal arithmetic is known to be wrap-around, meaning that the most significant bit is usually truncated. When adding two numbers (positive), you may end up with a result smaller than any of the inputs. This unwanted effect can be avoided by saturating arithmetic that provides the largest or the smallest possible representable number in the data type as the result. Both signed and unsigned saturation are available. Some applications of saturation arithmetic include computing the difference between two consecutive frames and Gouraud shading, in 3D graphics.

The way data is stored has a great impact on the performance benefits of MMX. For example, when storing an image in memory, you can either hold the R, G, B and Alpha values in four separate arrays, or you can store a single array, each component of the array being a record with four components. The best performance when using SIMD extensions is obtained by using separate arrays, as operations that have to deal with two images stored in that format can operate on multiple pixels in parallel. This can be the case when creating a certain "fade" effect between two frames, in which the pixels from the two frames are being multiplied by a fade factor (for the first picture) and (1-factor) for the second one, while the fade factor is being varied from 0 to 1.

When vectorizing code that has data dependant branches, a special boolean register type is used, generally known as bit vectors[25, 26, 4, 18]. The result of a vector compare instruction is stored into a bit vector which is later used to modify the execution of the subsequent instructions (e.g. when copying the date into the result register). Given the fact that MMX doesn't use dedicated bit vector registers, a typical approach when using MMX is to use a compare instruction to create a bit mask as result in a normal MMX register. The data is then copied into the result register using AND / OR operations.

Packing and unpacking is required when the computational data format is different from the memory storage format (e.g. when the data is stored in a 8-bit format in memory but when performing arithmetic operations with the data a 16-bit format is used to avoid overflows). MMX provides such Pack and Unpack instructions. However, placing some of these operations will take place inside the inner-loop of the kernels decreases performance. Packing and Unpacking represent overhead instructions which will increase the time required to complete the computation. Some other multimedia

extensions (like the Complex Streamed Instructions) completely remove the need for such pack/unpack instructions by first selecting the type of data to be used in special configuration registers, and then the pack/unpack is done automatically in hardware, saving time.

The modifications that had to be done to the design of the CPU for implementing the extensions were minimal. The eight 64-bit registers are the same as the eight 80-bit floating point ones. Special OS support is not required. The registers were named MM0 - MM7 for MMX. However, the pipeline of the processor was slightly modified, as one additional stage was provided - a special MMX operands read/write stage. In this stage, operands residing in MMX registers are read, and results are written from MMX registers into memory. Some operations requiring two cycles could execute in a single cycle, as the pipeline was not stalled for the second cycle any more.

From a performance point of view, speedups from 80% to 400% were reported in [28] for some kernels. The binaries that provided both MMX and non-MMX routines for the time consuming operations saw a size increase of approximately 10%. Die area increase is estimated also at about 10% [30]. In [17] it is estimated that 15 mm^2 of the total die size of 141 mm^2 is used for MMX, in the Intel Pentium 1 MMX(P55C), in a 0.29 micron process.

2.1.2 Internet Streaming SIMD Extensions - SSE

Two Pentium generations after MMX^{TM}, Intel introduced a new multimedia extension package: Internet Streaming SIMD (Single Instruction Multiple Data) Extensions (SSE). Unlike MMX, the new SSE provided support for floating point data types. A total of 70 new instructions were added. Instead of following the MMX approach which shared its registers with the Floating Point unit, the engineers chose to provide a separate register file, for SSE exclusive use. Eight 128-bit wide registers are available, named XMM0-XMM7. This increased the parallelism that could be obtained and avoided the increase in complexity of the existing register file. Operating system support is now required for correct functionality. The increase in die size was also about 10%, the same increase the Pentium1 die had when MMX support was added.

Special instructions were created for managing data caching and minimizing data cache pollution. This is caused by the fact that the data being processed could be separated into two categories: some data is being reused in the near future (like for example coefficient matrixes) and some data is being used only one time (usually the streaming data). Having control over which data will be stored in cache can make sure that non streaming data isn't replaced from the cache.

Floating point support is provided for single precision (32-bit) numbers. Also, two models of floating point arithmetic are provided: a normal, full IEEE compliance model, for precision critical computations and a second flush-to-zero (FTZ) mode for real time applications. This FTZ mode returns zero if the exceptions are masked and an underflow occurs. Most real time 3D applications are not sensitive to slight precision degradations. The FTZ mode, however, can execute faster than the normal mode.

The instructions added can be separated in the following categories: arithmetic, logical, comparison, data movement, shuffle, conversions, state, MMX enhancements

and streaming / prefetching. The floating point instructions are of packed type (PS suffix) or Scalar type (SS suffix). The difference is that the packed instructions operate on a all operands in parallel, so for example we can add four 32-bit floating point numbers in parallel with such an instruction, while the scalar instructions operate only on the least significant pair of operands from the two registers.

Special instructions allow control over data placement in the cache hierarchy. The programmer is given the freedom to choose between frequently used data that should stay in the cache and data that is used only used once before being discarded.

The comparison instructions provide a boolean result of all ones or all zeroes, like the MMX variants (SSE doesn't support dedicated bit vector registers either). These masks can be used to perform conditional moves using logic instructions (AND, OR, XOR), or to perform data dependent branches, by first moving four of the most significant bits of each component into an integer register (using MOVMSKPS or PMOVMSKB instructions).

Data organization plays again a major role in obtaining the desired performance. Two formats are possible, the Structure of Arrays (SoA) or the Array of Structures (AoS). When vectorizing the code, the most efficient approach is SoA: store each data component into a separate array (e.g. use three arrays to store the Red, Green and Blue color information) [30].

Conversions include converting between packed integer and packed single precision floating point numbers, or from integer to scalar single precision floats and vice versa. These conversions also perform truncation rounding. If the programmer needs to place the result of a conversion into memory, a separate store instruction is required.

Because most real-time multimedia application require fast computation and slight losses in precision are acceptable as long as the human operator doesn't perceive a loss in quality, two very time-expensive computations have been implemented using hardware look-up tables and limited precision: these are reciprocal of a number (1 divided by that number) and reciprocal square root. The LUTs provide 12 bits of mantissa. These are less accurate than the full precision provided by the IEEE compliant DIV and SQRT instructions, but are much faster. When a greater precision is required, a single Newton-Raphson iteration can provide 22 bits of precision, very close to the full IEEE instructions. A Newton-Raphson iteration requires two multiplications and a single substraction, so it can be performed very fast as shown in [27].

For example, for the computation of the reciprocal with Newton Raphson, the formula for computing the next iteration is presented in Equation 2.1

$$x^{(i+1)} = x^{(i)}(2 - x^{(i)}d) \tag{2.1}$$

The convergence is quadratic. So the number of precision digits doubles with each iteration, but initial convergence is slow. A hardware LUT provides enough precision for the general case, and by one or two Newton Raphson iterations full precision can be obtained in a small number of iterations. This is just another way of computing z/d : compute $\frac{1}{d}$ first and then multiply the result with z. This method is particularly efficient if several divisions by d are to be performed, such as dividing the x, y and z coordinates by the w perspective coordinate in 3D rendering.

An estimated improvement of about 15% is expected when using the approximate instructions for a basic geometry pipeline compared to the full precision IEEE instructions [30].

Unsigned multiplication was missing from the MMX instruction set. This has been fixed with SSE, and more efficient manipulation of unsigned pixel data is possible. Motion compensation in the MPEG2 codecs must use interpolation between the key frames. Operations date use 8 bits data types require rounding, equivalent to a 9 bits precision result. As MMX provided only 8 bits or 16 bits arithmetic, SSE provides a new instruction that performs a 9-bits accurate average operation. This accounts to a 25% speedup of this particular kernel, and 4-5% speedup at application level.

Motion estimation is a time consuming part of video encoding. This operation must find the frame that has the smallest differences compared to the current frame, in order to obtain the highest compression. Two metrics are currently used for that purpose, the Sum of Absolute Differences (SAD) and the Sum of Square Differences (SSD). A new instruction, PSADBW replaces the former seven MMX instructions required to perform the same job. The performance of the motion estimation kernel doubles.

All these new enhancements to the x86 ISA provide support for faster response and better performance with minimal hardware cost. The problem is then shifted to the software developers, which have to take advantages of the new features.

In [22] some examples are given on how to rewrite pieces of code with minimal effort to use the SSE/SSE 2. The following C++ code demonstrates the SSE code used to replace the conditional code $\mathbf{R = (A < B)\ ?\ C : D}$:

```
#include <iostream>
#include <fvec.h>
//Definition of a C++ class interface to Streaming SIMD Extension intrinsics.
//for using F32vec4:    4 packed single precision,
//32-bit floating point numbers

//#include <xmmintrin.h>
//Principal header file for Streaming SIMD
//Extensions intrinsics
//for _mm_andnot_ps and _mm_cmplt_ps
//fvec.h includes xmmintrin.h

using namespace std;

void print4(F32vec4& x)
{
    for (int i = 0; i < 4; i++)
        cout << x[i] << " ";
    cout << endl;
}

int main()
```

```
{
    float av[4] = {0.0, 0.0, -3.0, 3.0};
    float bv[4] = {0.0, 1.0, -5.0, 5.0};
    float cv[4] = {100.0, 200.0, 300.0, 400.0};
    float dv[4] = {-10.0, -20.0, -30.0, -40.0};

    F32vec4 a = (F32vec4 &) av[0];
    F32vec4 b = (F32vec4 &) bv[0];
    F32vec4 c = (F32vec4 &) cv[0];
    F32vec4 d = (F32vec4 &) dv[0];
    F32vec4 r;

    F32vec4 mask = _mm_cmplt_ps(a,b);

//the actual replacement for R = (A < B) ? C : D
    r =  (mask & c) | F32vec4((_mm_andnot_ps(mask,d)));

    cout << "[A]=";
    print4(a);
    cout << "[B]=";
    print4(b);
    cout << "[C]=";
    print4(c);
    cout << "[D]";
    print4(d);

    cout << "[mask]=";
    for (int i = 0; i < 4; i++)
    {
        int tmp;
        memcpy(&tmp, &mask[i], sizeof(float));
        cout << hex << tmp << " ";
    }
    cout << endl;

    cout << "[R]=";
    print4(r);

    getchar();
    return 0;
}
```

The output of the C++ program depicted above is:

```
[A]=0 0 -3 3
[B]=0 1 -5 5
```

```
[C]=100 200 300 400
[D]=-10 -20 -30 -40
[mask]=0 ffffffff 0 ffffffff
[R]=-10 200 -30 400
```

The bit masks created by _mm_cmplt_ps can be clearly seen in the **mask** variable. As no dedicated bit vector registers exist, normal 16 byte SSE registers are used to store the boolean results of the compare instructions.

From a data type point of view, it is shown that with SSE / SSE2 any data types that fit into the 128 bits registers can be used, not only single precision floating point or 16-bit integers. With a single instruction, arithmetic operations on four single precision floats or on two doubles can be performed. In C / C++ programming, the original source code has to be modified to the corresponding Vector Class in order to use SSE. This approach assumes the use of the SIMD friendly Structure of Arrays. Also the SSE / SSE2 instruction set provide data transform operations (shuffles, packs and unpacks) that can be used to restructure the data into a form suitable for SSE if limitations within the programming environment or the legacy code.

So support exists in major programming languages, and several tricks can be employed to obtain better performance. This however doesn't mean much without proper compiler support. In [19], two popular C compilers have been put to the test: the GNU C Compiler (GCC) 4.1 and the Intel C Compiler (ICC) 9.0. Compiled code was benchmarked and compared to a hand-coded assembly version.

All benchmarks were performed on a Pentium4@3 GHz, 2 Gbytes of RAM. The automatic vectorization capability of each compiler was assessed. For each of the selected SSE instructions a C sequence was written, and the code was compiled with both compilers. For example, the PSADBW (sum of absolute differences) test was not vectorized neither by GCC or by ICC. The PADDB test(adding two vectors of 16 bytes) was vectorized by both compilers.

In order to compute the speedup obtained by the automatic vectorization performed by the two compilers, each benchmark was compiled with and without flags that enabled the use of SSE instructions. The speedups obtained by using SSE for the PADDB test was 1.63 for GCC and 3.48 for ICC, while for the PSADBW test speedups were 0.93(GCC) and 0.81(ICC). If the special intrinsics available are used (such as _mm_add_epi8 for the PADDB test), the speedups for PADDB were 6.26(GCC) / 3.53(ICC) while for PSADBW 21.50(GCC) and 8.13(ICC). This suggests that for GCC and ICC using the intrinsics provides much better results compared to the automatic vectorization. The Intel C Compiler does a better job when detecting situations where it's possible to use SSE instructions, but in general GCC generates faster code.

2.2 MIPS V and MIPS digital media extensions (MDMX)

The MIPS V extensions supports floating point operations and it is mainly targeted towards accelerating 3D graphics applications. The MDMX extension provides support for parallel integer applications.

By providing a new format, the paired-single (PS), the MIPS V extensions allow two

single precision floating point numbers to be stored in a double precision floating point register. A lot of 3D applications as well as some scientific ones use single precision data, so a significant speedup could be obtained by this format [16].

MDMX is a set of instructions similar to Intel's MMX. The set of multimedia registers are also mapped onto the Floating Point registers. The new instructions allow parallel operations to take place on 8-bit or 16-bit data types packed in the media registers. One unique feature of MDMX is the presence of a 192-bit wide accumulator, thus allowing integer multiplications and additions to take place without loss of precision or overflow. Significant speedup is expected by the elimination of unnecessary data promotion / demotion instructions, compared to MMX.

MIPS realized the need to support similar extensions for floating point data types by creating MIPS V extensions. Intel introduced such support only in SSE, a few years after MMX became standard on all Intel processors. The operations that support the new PS data format include arithmetic instructions such as ADD, SUB, ABS, MADD (multiply-add) and MSUB. A parallel compare instruction is also present, as well as conditional moves. New instructions include instructions to rearrange PS data or to convert to/from the PS format. The multiplication of two single precision floats yields a single precision result, so when multiplying two paired-single operands a PS product is obtained.

The MDMX register file overlaps with the 32 floating point registers available in the MIPS architecture, four times more then Intel's 8 MMX registers. From a theoretical point of view, a performance boost of 2x-4x is possible, because a 64-bit register can support eight 8-bit values (oct byte - OB) or four 16-bit values (quad half - QH). MDMX includes arithmetic, logical, shift and min/max instructions. The new instructions always operate in saturating mode, clipping the results to minimum / maximum values. This feature is also present in MMX, as shown earlier.

Not only does MDMX support the normal "vector to vector" arithmetic, but a "vector to scalar" mode is also present. It is possible to select a single value from a packed register as the second operand of the instruction. Having a 5-bit immediate value as operand is also possible. This vector to scalar mode is very convenient when multiplying a vector with a constant value.

The PICK instruction can combine the values of two registers depending on the condition bits. If a condition bit is set, the corresponding byte is copied from the first source register, and the value is taken from the second register otherwise. The C.xx parallel instructions are used for setting the condition bits.

The unique wide accumulator found in the MDMX is a completely different approach than the one used in other instruction sets such as MMX. The problem arises when having for example to multiply two 8-bit values. In order to make sure that the result fits, 16 bits of storage are required. If multiple values have to be multiplied and then added together (accumulated), the problem will get worse. MMX provides special opcodes for data promotion. This problem is elegantly solved by the wide accumulator register, which can be partitioned into 24-bit or 48-bit values. A 24 bit value can accumulate the product of up to 256 products of two 8-bit values. The 48-bit mode can ensure that 65536 16-bit values can be multiplied and then accumulated without the danger of having overflows or loss of precision. Special instructions to access the accumulator in

three 64-bit parts are available.

No support for parallel 32-bit operations is available in MDMX. MIPS engineers say that this operation mode was mostly required for having better precision when working with 16-bit samples, to ensure enough precision and the wide accumulator renders this mode unnecessary. However, the architecture supports only a single accumulator and this can be a problem when several values have to be added together as you always have to wait for the previous operation that used the register to finish because you need that value to be present for the current operation, creating a performance bottleneck.

MIPS doesn't provide a hardwired Sum of Absolute Differences instruction. MPEG2 compression spends 70 to 80% of the time performing this operation, so clearly real-time video encoding can benefit a lot from this hardware support. Even the minimalist Motion Vide Instructions (MVI) extensions to the Alpha ISA provide such an instruction, the PERR. A single PERR instruction can replace 9 instructions in the motion estimation inner loop.

2.3 Sun's Visual Instruction Set (VIS)

With SIMD techniques, some algorithms can benefit of a speedup dependent of the ratio between the machine's data path width and the size of the multimedia data types. The hardware cost is low compared to the performance enhancements.

VIS was designed to work with 8, 16 or 32-bit integer or fixed point data types. These are stored in the floating point register files. The instructions can be separated into several classes: conversion, arithmetic/logical, address manipulation, memory access and motion estimation.

In the Sparc 9 architecture provides 32 single precision and 32 double precision floating point registers. The 32 single precision ones are mapped over the first 16 double precision registers. Single precision operands cannot access the last 16 double precision registers. Therefore, VIS provides support for efficiently accessing and updating the low halves of a 64-bit register.

If we look at the conversion instructions, special attention is given to having proper support for data promotion / demotion. These instructions take care of all the necessary scaling, clipping and truncation operations automatically. The arithmetic and logical instruction allow parallel addition, subtraction, etc. A full 16x16 bit multiplier is not present, and all the parallel multiplication are based on small 8x16 bit multipliers.

Resembling the MDMX functionality, distributed multiplication is present, allowing for one of the operands in the multiplication to be the upper half of a 32-bit register. This is multiplied with each of the components of a multimedia register.

Some instructions are specially tailored for accelerating application that process images or video clips. Such an instruction is the pixel compare instruction. A typical use of this instruction is Z-buffering hidden-surface elimination (HSR) [31].

Special instructions handle memory access. The block load and store instructions transfer data in 64 bytes blocks between the memory and 8 floating point registers. Because a lot of multimedia algorithms read the data, perform just a few operations and discard it, the data is never reused. The block load/store instructions do not affect

the data inside the cache. If a cache miss occurs, the data is not copied to the cache, maintaining cache coherency.

Hardware support is provided for computing the sum of the absolute differences, by the pixel distance instruction (pdist). The instruction computes the absolute differences between the corresponding 8-bit components and then adds the error values with the value previously held in the result register. If register src1 holds the byte values a0, a1...a7 and the register src2 holds the values b0, b1...b7 and the dst register has the value c before executing the instruction, the instruction will perform the following computation:

$$c = c+|a_0-b_0|+|a_1-b_1|+|a_2-b_2|+|a_3-b_3|+|a_4-b_4|+|a_5-b_5|+|a_6-b_6|+|a_7-b_7| \quad (2.2)$$

Software and compiler support are discussed in [31]. A programmer can always use the VIS instructions in hand-coded assembler, but this is a very tedious job. Even if a lot of time is spent on optimizations, the results can be much worse compared with a state-of-the art compiler. The Sun C compiler support macros that generate VIS code. These macros look like function calls, but are replaced with the proper sequence of VIS instructions. This allows the programmer to concentrate on the algorithm and leave details such as register allocation and scheduling to be solved by the compiler, automatically.

A common image processing algorithm is separable convolution, used to blur an image using a Gaussian kernel. In [31] the algorithm was described in C, and the code was modified to support the VIS extensions. The mathematical formula is [31]:

$$dst(i,j) = \sum_{l=-1}^{1} \sum_{k=-1}^{1} src(i-k, j-l)h(k)v(l) \quad (2.3)$$

In Equation 2.3 src and dst represent the source and destination images, while h and v represent the horizontal and vertical kernels.

The C code for separable convolution:

```
for (i = 0, i < w, i++) {
    hfilt1 = src[0] * h1 + src[1] * h2 + src[2] * h3;
    //horizontal kernel for row1

    src += slb;
    hfilt2 = src[0] * h1 + src[1] * h2 + src[2] * h3;
    //horizontal kernel for row2

    src += slb


    hfilt3 = src[0] * h1 + src[1] * h2 + src[2] * h3
    //horizontal kernel for row3

    src += slb
```

```
    for (j = 0; j < h; j++) {
        out = hfilt1 * v1 + hfilt2 * v2 + hfilt3 * v3;
        //vertical kernel for column
        out >> 16;
        *dst = (unsigned char) out;
        out = src[0] * h1 + src[1] * h2 + src[2] * h3;
        //horizontal kernel for next row

        hfilt1 = hfilt2;
        hfilt2 = hfilt3;
        hfilt3 = out;
        src += slb;
        dst += dlb;
    }
    sl = src = sl + 1;
    dl = dst = dl + 1;
}
```

Where src, sl, dest and dl represent pointers to the source and destination 8-bit gray scale images, h and w are the height and width of the destination image. Slb and dlb are the widths of the source and destination images in bytes. The horizontal kernel is composed of h1, h2 and h3 while the vertical one is composed of v1, v2, v3. The variables hfil1, hfilt2 and hfilt3 store intermediate results. Loop unrolling was used by performing the multiplications by the three vertical and horizontal filter coefficients explicitly.

In order to minimize the number of loads and stores, data was loaded into the registers 64-bit at a time, so information for eight 8-bit grayscale pixels.

In the inner-most loop, the VIS code was able to output eight pixels per iteration rather than just one pixel for the normal version. The measured speedup was considerable, the modified code being five times faster than the original version. In order to measure the performance of the two versions of the algorithm, a cycle accurate simulator was used, and the average number of cycles spent per pixel was measured.

Another application tested was the sum of absolute differences (SAD). The Pixel distance instruction is used to compute this sum. Each pdist instruction replaces approximately 48 instructions, and 32 instances of pdist are enough to compute SAD for a 16x16 block, compared to about 1500 normal instructions. The speedup in terms of cycles is $2429/441 = 5.5X$.

Trilinear interpolation is the operation of computing the value of a voxel from the values of its eight nearest neighbors. This operation can also benefit from the VIS extensions, with a reported speedup of 2.1X.

As for hardware cost, an increase of about 3% of the die is estimated in [17], or about 4 mm^2, for a 0.29 micron process.

2.4 AltiVec™

The AltiVec™extensions were designed by Motorola to be used in the G4 processor. It consists of 162 new instructions, and it supports both integer and floating point data types. A separate 32 register bank is provided, 128-bits wide. This is twice as wide as Sun's VIS or MDMX. By having 128-bit loads and stores, the memory bandwidth increases by a 4x factor. Also, the programmer can freely mix AltiVec instructions with FP instructions, but when a context switch occurs, the time required to save and restore the CPU state doubles [17]. A special flag is used to signal when these registers have been used, in order to save time when the AltiVec registers were not modified by not saving them.

Special Operating System is required for this extension in order to save and restore the new register bank when a context switch occurs.

AltiVec can therefore perform up to 16 integer operations in parallel, and up to four single precision floating point operations. The instructions have the same format as the regular PowerPC ones, having three registers as operands, as only register to register operations are supported.

Saturation and wrap around arithmetic are supported. For floating point operations, only the "nearest" rounding mode is supported. An even faster mode is also available which is not fully IEEE compliant, in which denormalised numbers (values that, if normalized, would require that the exponent field be less than its minimum value) are ingored. This mode can be appealing especially for 3D games, but some scientific applications that require IEEE compliant single precision floating point operations have to use the non-AltiVec instructions.

Instead of providing support for parallel division or square rooting, AltiVec supports reciprocal estimate and reciprocal square-root estimate, like Intel's SSE. One or two Newton-Raphson iterations can greatly improve precision. Two iterations are required to obtain full 24 bits precision. A special instruction was created to speed-up the Newton-Raphson iteration, the multiply-subtract, performing C - A x B. This new instruction can also be used with VLOGEFP (Base 2 logarithm estimate) and VEXPTEFP (2 to the exponent estimate).

From the data manipulation instructions, the VPKPX (pack into 1/5/5/5 pixels) and VUPKX (unpack 1/5/5/5 pixels) instructions stand out. The 1/5/5/5 pixel format is useful when the true color, 32 bit format is too slow, and provides 1 bit for α and 5 bits for the Red, Green and Blue components.

Parallel compare instructions create bit masks as results. A select instruction (VSEL) is similar to the pick instruction from other extensions, providing an easy way of replacing a data dependant branch from a code resembling the following sequence:

```
unsigned char a[8],b[8],c[8];
for (int i = 0; i < 8; i++)
    if a[i] < b[i]
        c[i] = a[i];
    else
        c[i] = b[i];
```

This can be accomplished by first having a parallel compare instruction executed and then execute a select instruction. The special bounds-check (VCMBFP) determines if $|X| \leq Y$

A set of instructions in AltiVec gives control over the cache hierarchy, from the same reasons explained for all the other multimedia extensions.

The sum of absolute differences instruction is missing from AltiVec. MPEG-2 encoding greatly benefits from a hardwired SAD, as more than 70% of the time spent in the main application performs SAD. Motorola supports the sum of differences squared (SSD) instead, and that could be achieved by the VMSUM instruction.

Hardware cost is estimated at 17 mm^2 in [17], so about 10-15% of the die size.

2.5 Matrix Oriented Multimedia (MOM)

The Matrix Oriented Multimedia extensions are a new type of multimedia extensions that try to exploit the fact that most streams that have to be processed in multimedia applications are 2D streams rather then one dimensional. Usually, small matrixes have to be operated upon inside the multimedia kernels.

It is a register to register architecture, and one of the goals when it was designed was to minimize the number of changes to existing CPU cores [9]. Recent processors include multimedia extensions like MMX or SSE. Increasing the parallelism on these architectures could mean a wider register file, like 256 or 1024 bits. The overhead associated with the packing / unpacking operations could neutralize any performance gain. So instead of trying to obtain more sub-word level parallelism, what MOM tries to do is to apply the advantages of a traditional vector processor ISA as seen in the IBM370 [4], but operating with MMX-style instruction. So Data Level Parallelism (DLP) can be exploited along two dimensions (two nested loops), and the parallelism extracted could be one order of magnitude higher.

The two MOM dimensions are defined in [9]. The X dimension is the one where subworld parallelism is exploited by ISAs like MMX or MDMX. Also, in [9] two restrictions are identified for vectorizing the X dimension: the fixed vector length (which is given by the ratio between the width of the multimedia register and the width of the data type) and the fixed stride (which is actually unit stride when an operation is performed in parallel on all the elements of the multimedia register).

By comparison, a traditional vector ISA vectorizes the outer loop, which is the Y dimension. There, no restriction is imposed for the stride or the vector length. With MOM, the whole matrix can be accessed by a single instruction. When performing MOM vectorization, two steps are performed: generate MMX-style instructions for the inner loop, and vectorize those MMX-like instructions over the outer loop.

So the best of the two worlds can be found in MOM: subword level parallelism for the MMX-style instruction, and the advantages of normal vector architecture, the capability to tolerate long latencies and the possibility to replicate the functional units that process data from a vector register.

The MOM architecture uses one Vector Length register and one Vector Stride register. This is because on the X dimension, the length and stride are fixed. The programmer has 16 matrix registers at his disposal. Each matrix register is composed of 16 64-bit

words. These registers will hold the matrix data. So the maximum number of elements stored in one matrix register is 16x8 elements, when 8-bit wide data is stored.

The memory instructions provided by the MOM ISA consist of instructions like mom_ldq MRi, Rj, Rk. The Vector Length register is used implicitly by the instructions. Rj provides the start memory address, and Rk is the stride. 64 bits are loaded each time before the stride is subtracted from the Vector Length register. The instruction completes when the Vector Length reaches 0.

Arithmetic instructions are similar to MMX instructions. For example, the parallel add instruction has 3 matrix registers as parameters. The Vector Length register controls how many of the 16 words in the two source registers are added together. A special instruction is provided for matrix transposition.

MOM doesn't use the promotion techniques found in MMX [28], but chooses to use packed accumulators similar to the ones found in MDMX [16]. This allows reduction operations to be carried on without loss of precision or overflow. But MDMX had problems when reduction was performed, because each time the accumulator was part of the computation, the current value needed to be known. This led to lower performance compared to the MMX technique that allowed a binary-tree like reduction. This problem is solved by MOM by pipelining the reduction operations over the Y dimension.

Initial performance estimation were carried out in [9], and continued in [12, 10, 11]. In [9], the memory hierarchy was not simulated, as a 1-cycle memory access was considered (equivalent to a perfect L1 cache). Also, no bandwidth restrictions were considered. The Mediabench suite was used, and in [9] only kernel performance was measured. Performance was tested against the MMX and MDMX architectures. All the kernels have been hand-coded for the different multimedia extensions. 67 MMX-like instructions were emulated, 88 for MDMX and 121 for MOM.

One thing worth mentioning is that for MMX in [9, 12, 10, 11] the architecture simulated was not x86, it was a MIPS R10000 like CPU with an Alpha ISA. So MMX got 32 multimedia registers instead of 8. MDMX was simulated with four logical 192-bit accumulators instead of just one. 16 matrix registers were used for MOM, plus two logical accumulators and one VL register.

The simulator used in [9, 12, 10, 11] was Jinks, an out of order simulator capable of executing vector ISAs. The basic architecture is similar to MIPS R10000, with an additional multimedia module (register file and functional units). The compiler used: DEC C V5.8.009 [9].

In all tests in [9], MDMX was usually about 30% faster than MMX, and MOM was faster by a factor of 1.3-4x. Compared to the normal Alpha ISA, performance gain ranged from 1.5x to 15x. One crucial parameter is the average Vector Length on dimension Y computed in the tests. It ranged from 1.83 in the **rgb** test to 11.11 in **ltpsfilt** [9]. This has a large impact on the performance obtained by MOM. When the average vector length is small, the performance advantage of MOM is minimal. When it increases, so does the performance: more work can be done in the same amount of time.

The test runs are summarized with the help of some performance metrics.

The number of cycles as described in [9] is:

$$Cycles = \frac{NI}{IPC} = \frac{NOPS}{IPC \cdot OPI} \tag{2.4}$$

NI = Number of Instructions
IPC = Instructions per Cycle
NOPS = the overall Number of Operations
OPI = Operations per Instruction

Several operations can be performed with one instruction (we are using Single Instruction Multiple Data extensions). So, $NI \cdot OPI = NOPS$.

R is defined as the factor of reduction of the number of operations required. This factor measures how efficient is the ISA in expressing the required computation with a minimum number of instructions. MOM generally has a better R factor because some loop overhead instructions are no longer necessary. The outer loop is included in the MOM instruction if the stream is not longer than the 16 words allowed by the registers. Otherwise several MOM instructions are required to process the Y dimension.

The Operations per Instructions (OPI) can be expressed as a function of the percentage of vector operations (F). If Vl_x and Vl_y are the average vector lengths on the X and Y dimensions,

$$OPI = (1 - F) + F \cdot (Vl_x \cdot Vl_y) \tag{2.5}$$

So the percentage of vector instructions is important because it determines the average operations per instruction achieved.

The Speed-up of a multimedia ISA over the base superscalar ISA can be computed as being:

$$S = \frac{\frac{NOPS_{scalar}}{IPC_{scalar}}}{\frac{NOPS_{new\_isa}}{IPC_{new\_isa} \cdot OPI}} = \frac{NOPS_{scalar}}{NOPS_{new\_isa}} \cdot \frac{IPC_{new\_isa} \cdot OPI}{IPC_{scalar}} \tag{2.6}$$

So if $R = \frac{NOPS_{scalar}}{NOPS_{new\_isa}}$ we get $S = R \cdot \frac{IPC_{new\_isa} \cdot OPI}{IPC_{scalar}}$

From the results in [9], the OPI factor is much larger for MOM than for the other multimedia extensions. It reaches a maximum of 22.27 in **motion2**.

If the Operations per Cycle (OPC) metric is defined as $OPC = IPC \cdot OPI$ the MOM architecture still wins, even if IPC drop dramatically (the MOM instructions are executed over multiple cycles as they operate over a large number of elements).

A more detailed analysis is performed in [12]. The same methodology for simulating the architecture is kept. Again, MOM faces MMX and MDMX. The memory subsystem is more accurately simulated. The register files are 0.5 Kbytes for MMX, 0.78 Kbytes for MDMX and 2.6 Kbytes for MOM. It is interesting that the estimated area of the register file for MOM is of the same order as for the other ISAs. This is due to the fact that many elements influence the die size of the register file, like the number of read and write ports.

When simulating the three multimedia extensions for issue rates ranging from 1-way to 8-way and 1 cycle memory latency, the relative performance advantage of MOM was grater for low issue rates. The greatest speedup over the normal Alpha ISA is for the 4-way CPU. When memory latency is fixed for 50 cycles, the slowdowns for MMX and MDMX range from 4x to 8x, the normal Alpha code slows downs ranging from 3x to 9x. MOM slow downs are in the 2x-4x range. This shows that the vector architecture of MOM has a high tolerance to high memory latency.

A more accurate simulation of the memory hierarchy [12] simulates a memory hierarchy similar to the one of Alpha 21364 processor. 32 Kbytes of direct mapped, write-through L1 cache and 1 Mbyte of 2 way associative write-back L2 cache, with 128 byte lines. 128 Mbytes of Direct Rambus memory are simulated. Also, three different versions of the cache models are tested for MOM [12]. The benchmarks show a 20% advantage of MOM over MMX.

Current processor trends show the popularity of two technologies: the Symmetrical Multi Threaded (SMT) and the Chip Multi Processor (CMP) designs. A SMT CPU is capable of executing multiple threads in the same time. It has the advantage of limited die area increase, and is capable of increasing the CPU's resources utilization. The SMT CPUs usually are build on a aggressive superscalar core, and 2 or 4 threads are usually supported.

CMP designs are multi-core processors. A famous example is the Cell processor, developed by IBM. The main advantage of CMP is that the increasing number of transistors available due to the decreasing of the manufacturing process can be used to put multiple cores on the same chip. The trend is to use simpler cores that are much easier to design and verify, and the chips are just replicated several times to create the CMP processor.

A compromise is sometimes created between CMP and SMT when single-threaded performance is very important. Hybrid designs have appeared, having for example two cores, each core being SMT capable, with two threads executing in parallel.

The multimedia workloads for the new MPEG-4 and H264 codecs demand huge processing power. Combining CMP / SMT with multimedia extensions makes sense, as the current generation of processors have problems providing enough performance for the multimedia tasks.

Data Level Parallelism (DLP) is something most multimedia applications have. Thread Level Parallelism (TLP) will become more and more important, as software is done using multi-threaded approaches, and multiple applications handle different multimedia tasks like video conferencing while 3D rendering is being done, and the operating system in running in background performing various maintenance tasks.

This problem is studied in [10] in order to evaluate the implications of adding MOM extensions to a multi-threaded or multi-core CPU. The preferred choice in [10] is having a SMT design modified to support multimedia extensions. The argument is that even having a small number of threads running, SMT processors can provide sufficient performance to meet the real-time requirements of having a state of the art codec such as MPEG4 running advanced compression algorithms for high definition media, such as having each object in the scene detected individually and then select the best compressing technique for each one, all this in an interactive environment. CMP is still not out of the race, because the implementation problems of out-of-order superscalar designs can be avoided by having many simpler cores on the same die.

The SMT processor simulated is a 8-way superscalar version of the MIPS R10000 CPU [10]. It is capable of fetching up to 8 instructions per cycle. The SMT threads share the same register file. A register renaming technique is employed in order to avoid false dependencies.

The extensions under test for this SMT simulator are an approximation if SSE [30]

and MOM [9]. It's worth mentioning that the SSE extensions simulated are an approximation of the real SSE, because the SMT processor is not based on the X86 ISA, but it is a MIPS R10000-like core. 67 opcodes of the SSE-like extension are implemented, along 121 opcodes for MOM. The SSE extension can use 32 multimedia registers instead of 8, and 16 matrix registers are available for MOM, each being composed of 16 64-bit MMX-like registers. Also, 2 192-bit accumulators are provided for MOM, similar to the ones found in the MDMX ISA. The register width for the SSE style extensions are set to 64 bits.

The memory system for the SMT processor is as follows: 32 Kbytes of L1 cache, direct mapped, write through, 32 byte lines, with interleaving among 8 memory banks. The L2 cache size is of 1 Mbyte, and it's a two way associative write back cache, with 128 byte lines. The Instruction cache consists of a 64 Kbytes, 2 way associative cache with 32 byte lines, and interleaving among 4 memory banks is used.

The basic CPU configuration is able to issue up to 4 integer instructions, up to 4 floating point instructions and up to 4 memory instructions (stores or loads) per cycle. The MMX/SSE processor is able to issue up to two MMX-style instructions per cycle. The SMT-MOM processor has one multimedia functional unit that is composed of two parallel vector pipes, capable of executing two SIMD sub-instructions from the same vector stream every cycle. So the SMT-MOM CPU requires an issue width of 1 for the MOM queue, down from 2 required for SSE/MMX.

The L1 cache has a latency of one cycle and L2 cache latency is fixed to 12 cycles. CPU frequency is 800 MHz, and 128 Mbytes of RDRAM memory are available (8 chips, 3.2 Gbytes / sec, 200 MHz).

SMT designs capable of executing 1, 2, 4 and 8 threads are tested [10]. The tests were part of the Mediabench suite. For each program, profiling was used to identify the most time-consuming function. Hand written assembly was used to vectorize the code, except for the 3D applications where vectorization was not possible because the proposed extensions do not support floating point operations. The tests consisted of the following applications:

- The mpeg2 encode of a four frames 352x480 video stream, 24-bit color
- The mpeg2 decode of four frames 352x480 frames
- The jpeg encode of a 1024x739 pixels, 24 bit color ppm file
- The jpeg decode of a 1024x739 pixels image
- The gsm encode test of a 300 Kbytes, 8KHz sampling rate, PCM audio stream
- The gsm decode test of a 13 Kbit / sec GSM audio stream
- With MESA, a Open GL 3-D graphics synthesis

Instruction breakdown was performed for each test, separating the instructions into the following categories:

- Integer instructions

- Floating Point instructions
- Memory instructions
- Vector instructions

Looking at the break down results, some interesting observations can be made. By Looking at the relative number of instructions (percents), all tests show that the number of integer instructions is by far the largest, over 50% in all tests and up to 90% in the GSM decoder test, with a 65% average for MOM and 62% average for MMX. The GSM decoder test shows poor vectorization opportunities, as only 1% of the instructions executed are SIMD instructions.

The MMX version of the processor usually has a higher percentage of vector instructions, but it must be noted that the work done by a matrix instruction for MOM can be up to 16 times more than for a MMX instruction.

Looking at the absolute number of instructions, MOM provides significant reductions in terms of the total number of instructions executed. This has two main reasons. First, a smaller number of matrix SIMD instructions are required to vectorize a multimedia kernel, because the work being done is much larger (the vectorization of the Y dimension pays of). Second, because the outer loop is being vectorized, a lot of integer instructions are no longer required. The update on the loop counter, the branch instruction for the outer loop can disappear from the MOM code.

One unexpected result is that looking at the relative distribution of the instructions, the integer instructions also dominate the MOM code. This is because the total number of instructions decreased, and as the number of SIMD instructions for MOM is much smaller than for MMX, the integer instructions still represent a large part of the total number of executed code. So the base processor (SMT in this case) has to hide the execution of the memory and SIMD instructions while executing the integer part of the program.

Multimedia applications (in contrast to the multimedia kernels) have a tendency to contain unbalanced regions of code, alternating from highly vectorizable kernels to areas where few or no SIMD instructions are executed. This workload unbalance problem can cause a few problems, performance-wise.

As up to 8 threads have to be busy at all times, a SMT version of the Jinks simulator was used. Throughput was measured rather than execution time. This is normal, as a workload consisting of processing media streams is targeted. A good measure of the throughput is the IPC (Instructions per Cycle). But when measuring two different ISAs, this can be deceiving. A new performance metric, the Equivalent Instructions per Cycle (EIPC) is defined as [10]:

$$EPIC_{MOM} = \frac{instructions_{MMX}}{instructions_{MOM}} \cdot IPC_{MOM} \tag{2.7}$$

This expresses a compensation factor for how semantically reach the evaluated ISA is, compared to MOM, and is computed as the ratio between the number of instructions executed by using the two instruction sets.

Under ideal memory conditions (equivalent to a perfect cache), performance scales well with the increase of threads of the SMT CPU. The SMT/MMX setup has an EIPC of 5.0 for eight threads, while SMT/MOM has an EIPC of 6.19 for eight threads.

If realistic memory conditions are simulated, going from 4 threads to 8 actually hurts performance. However, the MOM processor exhibits lower performance degradation than the MMX one. The explanation for the first effect could be that given the fact that all threads share the same cache, mutual interference increases as the number of threads increases. MOM performs better because it has a higher tolerance to increased memory latency. This shows that under ideal conditions the number of threads can be very high, but in reality a designer has to be careful to get a balanced configuration.

The hit rate of the L1 cache for SMT/MMX drops from 98.7% for one thread to 86.8% for eight threads, while for the same number of threads and SMT/MOM the drop is from 98.4% down to 93.7%. This shows that the thread interference is also lower for MOM, mainly because the stream nature of the memory references that determine several memory accesses for the same thread.

The way instructions are selected for execution can greatly influence the performance of a CPU by offering a balanced mix of scalar and vector (SIMD) instructions. In [10] four fetch policies are tested:

- Round Robin (RR)
- ICOUNT (IC) - give priority to threads that have the lowest number of instructions already decoded but not issued.
- OCOUNT (OC) - same criteria of ICOUNT, but for MOM take into account the value of the Vector Length (Y dimension) and give lower priority to the threads with the highest number of non issued operations.
- BALANCE (BL) - try to give a balanced mix of vector and scalar instructions by giving higher priority to threads that did not fetch any vector instructions if the vector pipes are full, and high priority to the threads that fetched vector instructions last time if the vector pipes are empty. If several threads have the same priority, a Round Robin policy is applied.

The different fetch policies show better results only for a high number of threads (four or eight). Performance can improve with as much as 9%. The best policy for SMT/MMX is ICOUNT, while OCOUNT is best for SMT/MOM. Even if the differences are now smaller, the fastest configuration is still SMT with 4 threads.

In order to take full advantage of the 8 threads configuration (SMT-8), the memory system can be modified [10]. If the SIMD instructions bypass the L1 cache and go directly to the L2 cache, performance of the SMT-8 can be better than the SMT-4 configuration. So instead of providing 4 ports for the L1 cache, 2 memory ports are directly connected to the L2 cache (for the SIMD instructions). Coherence problem have to be taken care of, and a protocol is proposed. The different fetch strategies provide better performance for MOM, up to 7% for the OCOUNT strategy. ICOUNT is still best for MMX.

Bypassing the L1 cache makes sense only for a high number of threads, where the 12 cycle latency of the L2 cache can be better tolerated. With this decoupled cache

hierarchy, SMT/MOM drops by only 15% in performance compared to the ideal case, while SMT/MMX sees a drop of 30% (for SMT-8).

Reduction operations are an important class of SIMD operations. Different approaches have been taken when implementing them. MMX has special instructions for data promotion (storing data to a larger format, for example a 8-bit sample can be stored in a 16-bit slot) and after computations are complete, a data demotion can be issued. This however means that a lot of overhead instructions will be executed. MDMX tries to avoid this by providing a wide 192-bit accumulator, that provides sufficient precision for the multiply and accumulate instructions. But this approach has some disadvantages, because of the data dependencies related to the accumulator register.

The MOM ISA also supports special accumulation registers, but the different vector-like ISA should fix some of the problems with MDMX [11] by using the special matrix registers.

The Mediabench suite has been analyzed after being re-written for MMX support and instructions were separated into the following categories:

- Scalar
- MMX memory
- MMX logic
- MMX arithmetic
- MMX reduction

Even if the reduction operations do not dominate (the scalar instructions do), most of the logic instructions are the overhead instructions, and those account for 10-15% of the total instruction count.

If we want to perform a simple dot product operation, the C code would be [11]:

```
s = 0;
for (int i = 0; i < 16; i++) {
    s += a[i] * b[i];
}
```

The assembly code for performing this loop can be separated into 3 distinct parts:

- Memory access
- Operand multiplication
- Reduction (addition of the multiplication results)

In the MMX-style ISA, all the loads and the multiplications can be done in parallel, no dependencies are present. The reduction stage can be performed by having a reduction tree type of dataflow.

If vector **a** starts at memory address R1, and b starts at address R2, and the multimedia registers are named m0...m31,the MMX assembly for the C code can look like [11]:

```
;Memory access part
ld m0, r1(0)        ;m0 = {a[0], a[1], a[2], a[3]}
ld m1, r1(1)        ;m1 = {a[4], a[5], a[6], a[7]}
ld m2, r1(2)        ;m2 = {a[8], a[9], a[10], a[11]}
ld m3, r1(3)        ;m3 = {a[12], a[13], a[14], a[15]}
ld m4, r1(0)        ;m4 = {b[0], b[1], b[2], b[3],
ld m5, r1(1)        ;m5 = {b[4], b[5], b[6], b[7]}
ld m6, r1(2)        ;m6 = {b[8],  b[9], b[10], b[11]}
ld m7, r1(3)        ;m7 = {b[12], b[13], b[14], b[15]}

;Parallel multiplication part
pmul m0, m0, m4     ;m0 = {a[0] * b[0], a[1] * b[1],
                    ;      a[2] * b[2], a[3] * b[3]}
pmul m1, m1, m5     ;m1 = {a[4] * b[4], a[5] * b[5],
                    ;      a[6] * b[6], a[7] * b[7]}
pmul m2, m2, m6     ;m2 = {a[8] * b[8], a[9] * b[9],
                    ;      a[10] * b[10], a[11] * b[11]}
pmul m3, m3, m7     ;m3 = {a[12] * b[12], a[13] * b[13],
                    ;      a[14] * b[14], a[15] * b[15]}
;Accumulation part
padd m0, m0, m1     ;m0 = {a[0] * b[0] + a[4] * b[4],
                    ;      a[1] * b[1] + a[5] * b[5],
                    ;      a[2] * b[2] + a[6] * b[6],
                    ;      a[3] * b[3] + a[7] * b[7] }

padd m2, m2, m3     ;m2 = {a[8] * b[8] + a[12] * b[12],
                    ;      a[9] * b[9] + a[13] * b[13],
                    ;      a[10] * b[10] + a[14] * b[14],
                    ;      a[11] * b[11] + a[15] * b[15] }

padd m0, m2, m0     ;m0 = {a[0] * b[0] + a[4] * b[4] +
                    ;      a[8] * b[8] + a[12] * b[12],
                    ;      a[1] * b[1] + a[5] * b[5] +
                    ;      a[9] * b[9] + a[13] * b[13],
                    ;      a[2] * b[2] + a[6] * b[6] +
                    ;      a[10] * b[10] + a[14] * b[14],
                    ;      a[3] * b[3] + a[7] * b[7] +
                    ;      a[11] * b[11] + a[15] * b[15]}
```

One more instruction must be executed to add all the components of the multimedia register m0 together.

If N is the number of elements that have to be accumulated (16 for the C example above), W_{MMX} is the width of the MMX registers (for example it is four for 64-bit MMX registers and 16 bit data values), L_{mem} is the latency of the loads, L_N is the length of the packed multiplication and L_R is the latency of the packed addition, we can see that

the critical path is

$$T_{MMX} = L_{mem} + L_N + log_2\left(\frac{N}{W_{MMX}}\right) \cdot L_R \tag{2.8}$$

The binary tree structure of the dataflow graph accounts for the log_2 factor in the critical path. In this example, it is consider that data promotion is not necessary. This shows that MMX can exploit the parallelism, but if we introduce the overhead instructions, things can be quite different.

For the reduction using MDMX-like instructions, we can use the packed accumulator for performing this type of operations. This approach is used mostly in DSPs, where processing narrow data types is very common. Special opcodes exist in the instruction set to specify that the results of an operation have to be added to the current content of the accumulator. Also instructions for adding all the values in the packed accumulator and perform clipping / truncation are available. This solves the precision problem and also saves the overhead found in MMX for data promotion.

The way the accumulator is used however can reduce the performance by inducing a data dependency whenever the accumulator needs its previous value as input. Binary tree structures like the one found for MMX are no longer possible. If the operations performed have long latencies, IPC values will drop. Maximum parallelism imply the use of the associative property of operations, and the accumulator denies in some situation the use of this property.

The MDMX assembly code for the C code above is [11]:

```
;Memory access part
ld m0, r1(0)             ;m0 = {a[0], a[1], a[2], a[3]}
ld m1, r1(1)             ;m1 = {a[4], a[5], a[6], a[7]}
ld m2, r1(2)             ;m2 = {a[8], a[9], a[10], a[11]}
ld m3, r1(3)             ;m3 = {a[12], a[13], a[14], a[15]}
ld m4, r2(0)             ;m4 = {b[0], b[1], b[2], b[3]}
ld m5, r2(1)             ;m5 = {b[4], b[5], b[6], b[7]}
ld m6, r2(2)             ;m6 = {b[8], b[9], b[10], b[11]}
ld m7, r2(3)             ;m7 = {b[12], b[13], b[14], b[15]}
;Multiply and accumulate
pmul&acc acc0, m0, m4    ;acc = {a[0] * b[0],
                         ;       a[1] * b[1],
                         ;       a[2] * b[2],
                         ;       a[3] * b[3]}

pmul&acc acc0, m1, m5    ;acc = {    a[0] * b[0] + a[4] * b[4],
                                 ;   a[1] * b[1] + a[5] * b[5],
                                 ;   a[2] * b[2] + a[6] * b[6],
                                 ;   a[3] * b[3] + a[7] * b[7]}
pmul&acc acc0, m2, m6    ;acc = {    a[0] * b[0] + a[4] * b[4] +
                                 ;                  a[8] * b[8],
                                 ;   a[1] * b[1] + a[5] * b[5] +
```

```
                                ;                  a[9] * b[9],
                                ;   a[2] * b[2] + a[6] * b[6] +
                                ;                  a[10] * b[10],
                                ;   a[3] * b[3] + a[7] * b[7] +
                                ;                  a[11] * b[11]}

pmul&acc acc0, m3, m7   ;acc = {    a[0] * b[0] + a[4] * b[4] +
                                ;   a[8] * b[8] + a[12] * b[12],
                                ;   a[1] * b[1] + a[5] * b[5] +
                                ;   a[9] * b[9] + a[13] * b[13],
                                ;   a[2] * b[2] + a[6] * b[6] +
                                ;   a[10] * b[10] + a[14] * b[14],
                                ;    a[3] * b[3] + a[7] * b[7] +
                                ;   a[11] * b[11] + a[15] * b[15]}
```

The critical path for this code is:

$$T_{MDMX} = L_{mem} + \frac{N}{W_{MDMX}} \cdot (L_N + L_R) \tag{2.9}$$

The logarithmic term disappeared as it is no longer possible to create a binary tree in the reduction stage. And the latencies for multiplication and addition are chained as the operation is of multiply and accumulate type. But because we don't have to worry about precision, we no longer have the overhead instructions for promotion, and after promotion the W factor would become W/2, because the data types used become larger.

One way of improving MDMX would be to provide more than one accumulator.

If the same operation is performed having the MOM ISA at our disposal, the ASM code would look like:

```
;initialize the MOM registers
set_length 4            ;vector length = 4
set_acc acc0, 0     ;acc0 = 0
;memory access part
ld vm0, r1          ;vm0 = a[0..15]
ld vm1, r2          ;vm1 = b[0..15]
;multiply and accumulate part
pmul&acc acc0, vm0, vm1 ;acc0 = a[0]*b[0] + a[1]*b[1] + ... +a[15]*b[15]
```

The latency of the accumulator is hidden by pipelining it. Several shadow accumulators are kept in flight at one time in the pipeline [11].

After initializing the matrix registers, only two loads and one matrix parallel multiply & accumulate instruction over accumulator0 are required for the computation. The pipeline may take advantage of the binary-tree style of schedule similar to the MMX execution of the loop, because all the information about the multiply and accumulate progress is present in one single instruction. One more operation is required over MMX however, the accumulation of the results in the accumulator register.

The length of the critical path is [11]:

$$T_{MOM} = L_{mem} + L_N + \left(log_2 \left(\frac{N}{W_{matrix}} \right) + 1 \right) \cdot L_R \tag{2.10}$$

Where W_{matrix} represents the number of elements packed inside a MMX-style register (four for this example), N is the length of the arrays.Comparing this result to the MMX latency, it is longer only by L_R, so a very small difference that is by far compensated by the advantages of using accumulators.

In order to evaluate the performance benefits the packed accumulators can provide, a more detailed instruction count is performed in [11] for some kernels of the Mediabench suite : mpeg2 encode, mpeg2 decode, jpeg encode, jpeg decode, gsm encode and gsm decode. After manually optimizing each kernel to take advantage of the MMX / MDMX / MOM instructions, the operations were divided in several categories:

- Control
- Scalar memory
- Scalar arithmetic
- Vector memory
- Vector logic
- Vector arithmetic
- Vector reduction

Several observations can be made:

- MOM reduces the number of scalar instructions over MMX and MDMX (both arithmetic and memory), and the reason is the vectorization of the Y dimension. The second loop is therefore removed, and so are the overhead instructions associated with it.
- MOM and MDMX remove a large part of the logical multimedia instructions, which are used for data promotion, sign extensions, matrix transpose, data re-arranging, precision conversions, etc. This reduction is quite dramatic, as usually more than 50% of the MMX instructions are of this type, and for the MDMX / MOM ISAs the numbers drop to 10-20%. This is the result of using packed accumulators.
- MDMX exhibits the highest relative percentage of scalar instructions. This is because the number of logical multimedia instructions is reduced.
- MOM reduces both the number of scalar instructions (because the Y dimension is vectorized) and the number of vector instructions (by using packed accumulators).
- MMX architectures are dominated by logic overhead. MOM/MDMX architectures are dominated by reduction operations because the logic overhead is reduced.

Benchmarking the full applications is performed by simulating a 8-way superscalar MIPS R10000 processor. The MOM unit is composed of four vector pipes, being capable of executing four SIMD instructions per cycle. The MMX/MDMX implementation have four independent memory ports, and the MOM implementation has the ability to perform two independent scalar memory operations / cycle or up to four operations from the same vector memory reference.

The configuration of the register files is different for each multimedia extension. For MMX, 12 read ports and 8 write ports are available. The same holds for MDMX, but four packed accumulators are added, providing 4 read ports and 4 write ports for the accumulator.

The MOM register file is quite different. Because four vector pipes are simulated, 3 read ports and 2 write ports for each lane is provided for each lane. The lanes are provided with a connection between the lines to be able to perform the last stage in the reduction operations. One lane is the Master lane, as only that one is capable of reading and writing from the accumulator (which has 1 read and 1 write port). This ensures that the die size of the register file is about the same size as for MMX / MDMX, even if the size is much larger (4.04 Kbytes for MOM vs. 0.50 Kbytes for MMX and 0.68 Kbytes for MDMX). The reduction of the number of ports also translates into a reduction of the cycle time for the register files (for MOM, the estimated cycle time is 1.362 ns, while for MMX/MDMX is 1.476 ns).

If the number of registers is increased, the performance can be improved. Having the MMX architecture with 32 registers as the base configuration, the number of registers is increased up to 100 for MMX/MDMX and from 16 to 50 for MOM. For every kernel, there's a value after which increasing the size of the register files provides no more benefits, as the performance remains flat.

In the tests where reduction operations are not present, MMX and MDMX perform similarly, as MDMX can be thought of as being a superset of MMX. The speedups range from 2 to 4 for MMX/MDMX and from 3 to 5 for MOM. The number of required registers for obtaining maximum speedup is around 50-60 registers for MDMX/MMX and about 40 for MOM for these tests. However, each matrix register for MOM contains 16 registers, so actually the number of registers for MOM is quite large.

For the tests that include reduction operations, performance gains of MOM range from 1.1X to 2.5X over MMX and from 0.9X to 2.2X over MDMX. MMX can perform up to 4.75X faster than the baseline configuration. In some tests however, MMX clearly outperforms MDMX despite the shortcomings mentioned earlier, because the potential ILP is reduced by the use of the packed accumulator in MDMX. This is most obvious in the **ltppar** test, where MMX is 35% faster than MDMX. MOM performance saturates after 20-30 registers.The number of accumulator registers was not modified during the tests.

The number of available registers has a large impact on performance. But also the latency of the SIMD functional units can influence performance. For the MOM ISA, two different assumptions have been made [11]. One of them is weather to increase the latency of the reduction component or the regular component of the multimedia instructions. The second one is about the last stage of the reduction process (where the vector pipes have to communicate between them to perform the final addition). This

operation can be pipelined or not. For the other extensions (MMX and MDMX), the latency is increased from 1 to 5 cycles.

From the tests in [11], several observations can be made. First, the kernels where reduction operations occur are more sensitive to the increase in latency. For MMX, the average slow down is 2.3X for kernels with reduction operations and only 1.5X for the other kernels. Also, the execution time (in cycles) of MMX grows faster than the one for MDMX. This is again caused by the overhead instructions because of the lack of packed accumulators in MMX. MDMX execution is dominated by scalar instructions, so increased latency for the multimedia instructions can be better tolerated.

For MOM, execution time grows faster when the latency is increased for reduction instructions. Slow-down is 1.4X when the latency is increased for regular instructions and raises to 1.6X when the reduction operations have higher latency. If no pipelining support is given for the intra-lane communication, the MOM execution time rises significantly faster, resembling the patterns of MDMX. If pipelining is allowed, MOM performs much better than the other architectures.

If full application performance is measured, the MMX enabled processor can have slowdowns of up to 40%, much more than MOM which has a maximum performance drop of 5%. This shows the increased tolerance to high latency of the matrix extensions.

2.6 Complex Streamed Instructions (CSI)

The Complex Streamed Instructions [32] is an ISA extension that addresses the problems of other multimedia extensions, like: the limit of the number of elements that can be processed in parallel (which is equal to the multimedia register size, and is visible to the programmers), the overhead for data reorganization (pack / unpack operations, data alignment) and the fact that usually multimedia applications operate on matrix sub-blocks, with gaps between consecutive rows [20].

The CSI instructions process two dimensional streams of data. There is no programmer visible constraint on the length of the streams. The hardware is responsible for dividing the streams into sections that will be processed in parallel. Data packing / unpacking and conversions are also performed automatically by the hardware. The streams are 2D, so loop overhead is also eliminated.

The section size is fixed for current multimedia ISAs, such as MMX or VIS. The multimedia registers have a fixed length (for example 64 bits). If increased performance is needed and the width of the SIMD data path needs to be increased, the ISA has to be changed. This means that the existing code has to be rewritten or recompiled. CSI solves this problem by allowing the processed data streams to have arbitrary lengths. The number of elements actually processed in parallel is not visible to the programmers, and is hardware-implementation dependant.

The execution of most multimedia kernels consists of the following steps [20]:

- Load the operands into registers
- Rearrange data so that elements are stored consecutively
- Convert the data from storage to computational format

- Perform actual computations
- Convert results to storage format
- Rearrange the results
- Store results in the memory

The data streams are 2D, but any stride (distance between to elements) can be used both for the X dimension and for the Y dimension of the stream. The CSI hardware is responsible with the address generations and data alignment. MOM provides similar support, but the stride for the X dimension is fixed to 1.

The different formats used while performing computation represent another source of overhead. MMX has instruction for data promotion / demotion, and MDMX uses packed accumulators to provide sufficient precision.

When the streams processed have different formats (for example one of them uses 8-bit data while the other uses 16-bit data), the packing / unpacking is performed automatically by CSI hardware. This is also the case when the result may not be representable by the format used. Wrap-around and saturation arithmetic can be used. MOM still require data packing/unpacking if the streams use different formats, and the precision problem is solved by using packed accumulators.

Unlike MMX or MOM, CSI is a memory-to-memory architecture. Not using registers means that the section size is not limited by the architecture. Instead of having registers around for data reuse, the L1 cache is considered to be large enough and also the hit rate very high. The memory bandwidth is not stressed by always writing the results to memory [20].

The CSI architecture uses a set of 32-bit Stream Control Registers (SCR set), which are accessed by a number ranging from 0 to 5. These registers are:

0. **Base.** This register contains the base address of the stream.
1. **RLength**. Specifies the number of stream elements in a row.
2. **SLength**. Specifies the total number of elements in the stream.
3. **HStride**. Contains the distance in bytes between consecutive elements in a row.
4. **VStride**. Contains the distance in bytes between consecutive rows of the stream.
5. **S4**. Four fields are present in this register:
    - **Size** is a 2-bit selection of the size of the elements: 00 selects bytes, 01 selects half-words, 10 words and 11 double words.
    - **Sign** is a flag specifying signed or unsigned operands
    - **Saturate** is a flag used to select wrap-around or saturating arithmetic.
    - **Scale factor** determines the number of bits the data is to be shifted left before the Least Semnificant Bits (LSBs) are rounded and discarded. This mechanism is useful for specifying the number of fractional bits.

The CSI instructions are divided into two main categories:

1. Arithmetic and logical instructions

   - Operation SCRSi, SCRSj, SCRSk - two inputs steams are processed and an output stream is produced. Streams are specified by using the SCR sets. Operations include pairwise addition and multiplication.
   - Operation SCRSi, SCRSj, GPRk - similar to the previous format, but one of the operands is a scalar value (for example, the multiplication of a stream with a scalar).
   - Operation GPRi, SCRSj, SCRSk - two data streams are processed and the result is a scalar value (example: dot product of two data streams).

2. Auxiliary instructions. These are used for the initialization of the SCRs.

   - csi_mtscr SCRSi, j, GPRk - mtscr is short for move to stream control register. The instruction loads the jth SCR register in set i with the value of GPRk.
   - csi_mtscr SCRSi, j, imm - same as above, only an immediate value is used instead of GPRk.

Support for exceptions in CSI is similar to the one provided in the IBM System/370 architecture: a stream interruption index register is maintained. When the execution in interrupted, this register indicates which stream element is being processed. The instruction can resume from that point.

Figure 2.1 presents the block diagram of the CSI datapath [20]. The main blocks are the SCR sets, the memory interface unit, the pack / unpack units, the CSI functional units and the accumulator ACC.

The Memory Interface Unit (MIU) transfers data between the memory hierarchy and the stream buffers. It must extract data from the cache (even if it is not stored in consecutive positions) and align it in the proper order.

The **unpack** blocks convert the data from the format in which data is stored to the format used for computations. They use the values from the Sign and Sign fields of SCR S4. The pack blocks perform the reverse procedure, and use the scale factor, sign, saturate and size from SCR S4 to perform saturation and truncation. The CSI functional units perform actual computations, exploiting subword level parallelism. **CSI MULT** performs parallel multiplication and division. The **CSI ALU** performs addition, subtraction, and the sum of absolute differences (SAD) operation. The size of the input registers is n, while the size of the output register is 2n, so overflow is avoided.

The accumulator is 3n bits wide, and is able to perform 2^n $n \cdot n$ products without overflow [20]. It does so without the need of data promotion. It is used in reduction operations such as SAD or dot product (DOTPROD).

One important observation is that the stream unit performs data promotion only when the formats of the two input streams are different, not when the results may become too large. The k-input adder performs reduces the k values that are stored in the accumulator. K can be either n / 8, n / 16, n / 32 or n / 64.

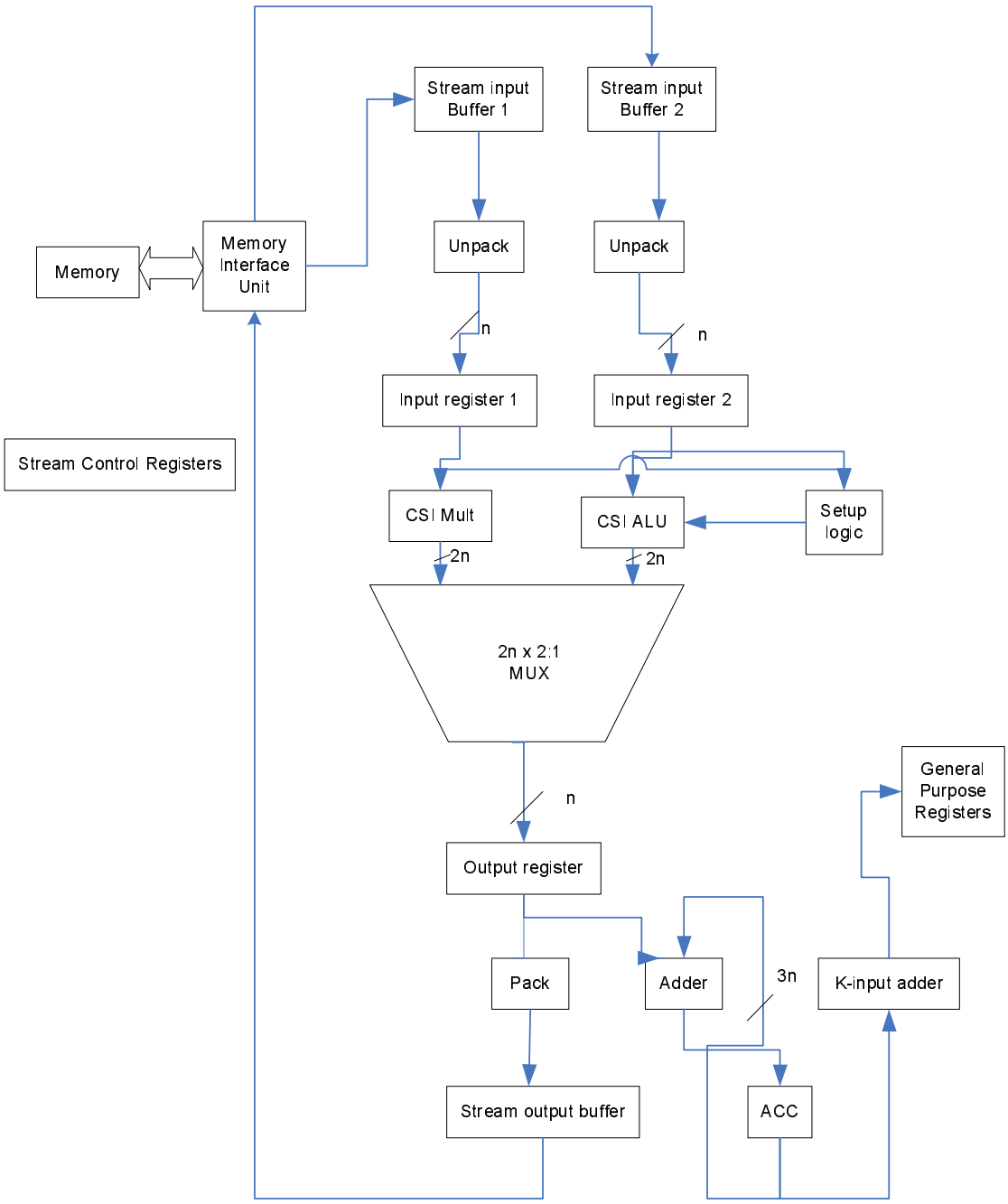


Figure 2.1: CSI data path

The Memory Interface Unit (Figure 2.2) is composed of three Address Generators (AGs) for the two input streams and the output streams, a Load Queue (LQ), a Store Queue (SQ) and the Extract and Insert blocks.

Several possibilities exist to connect the memory interface unit with the cache hierarchy. The MIU can be connected to the L1 cache or directly to the L2 cache. In [20],

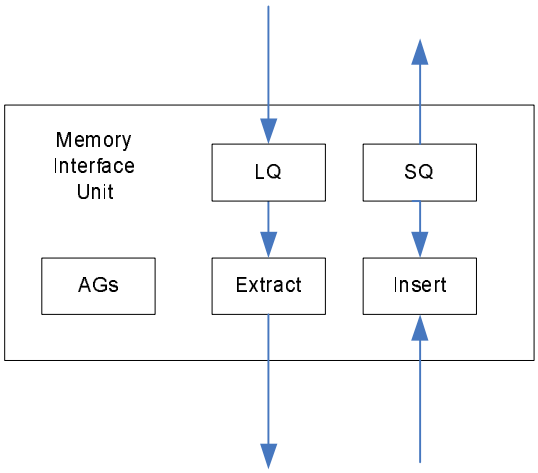


Figure 2.2: CSI Memory Interface Unit

the option is to connect the MIU to the L1 cache for the simulations performs for the following reasons: for multimedia kernels, the hit rate of the L1 cache is very high (up to 99%). Another reason is that the L1 cache is featured on-die for most CPUs, so it is much more easier to widen the data-path between the L1 cache and the CSI unit to arbitrary values.

In order to evaluate the performance of the CSI ISA, the MediaBench suite is used in [20]. The following benchmarks were used: mpeg2enc (MPEG-2 encoder), mpeg2dec (MPEG-2 decoder), cjpeg (JPEG encoder) and djpeg (JPEG decoder). For the MPEG tests, three 128x128 frames were used. For the JPEG tests, a 227x149 image was used.

The simulator used was sim-outorder from the SimpleScalar toolset, version 3.0 [2]. This simulates a superscalar architecture, and support for CSI and VIS [31] was added. The architecture is derived from the MIPS IV ISA.

The routines were optimized for CSI and VIS extensions by hand-coding in assembly. Optimization techniques such as loop unrolling were used. The selected kernels were Add_block (MPEG2 frame reconstruction), Saturate (saturation of 16-bit elements to 12-bit range, in the MPEG decoder), dist1 (Sum of Absolute Differences, used in motion estimation), ycc_rgb_convert(color conversion for JPEG), and h2v2_downsample (downsampling color components for JPEG).

The base system used is a 4-way superscalar processor with out-of-order execution support [20]. Clock frequency is 500 MHz, L1 cache is 32 Kbytes in size, direct mapped, one cycle latency. A 1 Mbyte L2 cache with 6 cycles latency and 2-way associativity is used. Main memory type is SDRAM, 100 MHz frequency and 64 bits bus width. A perfect instruction cache is also assumed.

The multimedia extensions compared are CSI and Sun's VIS. The VIS extensions operate on the floating point register files and have one cycle latency, except for the SAD and packed multiply instructions which have 3 cycle latency. The datapath width for VIS is 128 bits. For CSI, instruction latency is 1 cycle, except for the CSI multiplier which has a 3 cycle latency but is fully pipelined. CSI datapath is 128 bits.

Because of the fact that CSI extensions have a memory to memory architecture, the

pipeline is stalled when a CSI instruction is detected, until all memory instructions have finished executing.

The simulations also included a 2-way superscalar processor, and the speedups are computed with regard to this configuration (without any multimedia extensions).

For the 2-way configuration, the kernel-level speedups range from 1.4X to 5.9X with an average of 3.1X for VIS, and from 4.0X to 22.7X (12.3X on average) for CSI. For the issue width set to 4, the speedups for VIS range from 2.3X to 7.2X (4.4X average), and for CSI they range from 4.2X to 28.3X (15.0X average). This shows CSI holds the performance crown, with the 2-way CSI enabled processor being faster than the 4-way VIS enabled CPU. The largest differences in performance are in the Saturate and Add_Block kernels. By using the CSI extensions, the whole loop for Saturate can be expressed by a single instruction [20].

When looking at the speedups for the whole application for the 2-way CPU, the VIS architecture obtains speedups ranging from 1.17X to 1.93X, while CSI performs 1.28X to 2.28X times faster. For issue width set to four, the speedups for VIS range from 1.59X to 2.37X and for CSI from 1.74X to 2.77X. The good performance of the 2-way processor with CSI extensions suggest a possible use in embedded applications.

Integer performance of CSI is within expectations. However, in 3D graphics applications, the floating point performance is very important. The CSI extensions are extended for supporting single precision IEEE 754 operations in [6]. Support for the new data type (single precision floats) is introduced in [6] by the new FP flag in the S4 configuration register. In order to specify that floating point data is used, S4.Size must be set to 4 bytes, and S4.FP must be set to 1.

Conditional execution is quite common in code that needs to be vectorized. If special support is not given for this type of operation, vectorization is very difficult and many times a compiler will keep a piece of code like the following loop scalar:

```
for (i = 0; i < n; i++) {
        if (A[i] > 0)
            A[i] = A[i] + B[i];
    }
```

Support for conditional execution is provided for CSI by the addition of masked arithmetic support. If the i-th element of the mask vector is set to 1, the operation is actually performed, otherwise a NOP is executed. In order to support this new execution mode, a new type of streams is added: CSI bit streams. Because these are a continuous block of memory, they are completely specified by two values: the Base, which is the memory address of the first stream element (in bytes), and the stream length (the number of bits in the stream). Two 32-bit registers can be used to store the Base and Length, and a pair of registers forms a Mask Stream Control Register (MSCR). 16 such sets are added to the CSI ISA in [6].

Two operating modes are defined for CSI: the masked and the unmasked mode. Also, a new type of operand is supported, the mask operand (a bit stream). A new register, the Stream Status Register (SSR) provides information on the current operating mode. If the masked mode of execution is used, the CSI instructions are processed under the

control of the mask stream operand. Otherwise, the mask operand is ignored and normal execution occurs.

However, masked execution can be less efficient when the mask vector contains a large number of zeroes. This means that many NOPs are executed, but no useful results are computed. This however can be fixed by selecting all the operands for which the corresponding mask bit is set to 1, and form a shorter vector that can be operated upon without execution masks. After the operation is complete, the elements can be copied back to their original positions.

Two instructions provide support for such operations: csi_extract and csi_insert:

- csi_extract SCRSk, SCRSi, MSCRSj executes the following operation: for each element in the SCRSi input stream the corresponding bit in the MSCRSj stream is checked. If it is set to 1, the element is copied (extracted) to the output stream SCRSk.
- csi_insert SCRSk, SCRSi, MSCRSj inserts the elements of the input stream SCRSi in the positions of SCRSk where the MSCRSj bits are set to 1.

If the condition is of if-then type, the csi_insert and csi_extract instructions can solve the problem. However, is the stream is of if-then-else type, two instructions are added to support this case:

- csi_split splits the input data stream by looking at the values of the mask stream into two corresponding streams: one for which the if condition evaluates to false, and the other contains the elements where the condition evaluates to true.
- csi_merge performs the reverse operation of csi_split, and merges the two streams.

In order to evaluate the performance of CSI using the new additions, the SimpleScalar toolset is used. The benchmark used is SPEC Viewperf. TheOpenGL library used was MESA. Three different MESA libraries were created, for the scalar, the SSE-enabled processor and the CSI-enabled one. The kernels were manually rewritten to support the multimedia extensions. The base configuration : a 666 MHz 4-way superscalar processor with out of order execution support, 32 Kbytes of 4-way associative L1 cache, 1 Mbyte of 2-way associative L2 cache, 6 cycles latency. Main memory was of SDRAM type, 166 MHz. 4 floating point single precision multipliers and 4 floating point single precision adders were allocated. The SIMD extensions simulated were SSE and CSI, and each unit was capable of executing 4 floating point operations in parallel.

The CSI execution unit is connected to the L1 cache, because the benchmark used exhibits a high cache hit ratio. The L1 cache is provided with two ports.

In order to identify eventual bottlenecks of the SIMD extensions, a first test is to see the performance impact of the instruction window size. The sizes of the Register Update Unit (RUU) and the Load Store Queue (LSQ) are varied. The baseline system is the superscalar processor with a 32 entries RUU. The RUU scheme uses a Reorder Buffer (ROB) to automatically perform register renaming (in order to eliminate false data dependencies) and hold the results of pending instructions. In each cycle, the ROB retires completed instructions in program order to the register file. The LSQ works

as follows: store values are placed in the queue if the store is speculative. Loads are dispatched to the memory system when the addresses of all previous stores are known. Loads may be satisfied either by an earlier store value that is kept in the queue or by the memory hierarchy [8].

When the RUU size is smaller, CSI has a much larger performance lead compared to SSE. This is because the ability of the CPU to exploit ILP decreases when the RUU size is reduced. CSI ISA can execute a 2-dimensional stream operation by just one instruction, so the performance of the CSI enabled processor is insensitive to changes to the RUU size. This behavior is beneficial for CSI, as the cost of increasing the instruction window grows quadratically with respect to its size [6].

The performance of the CPU with SSE extensions is very close to the one of the standard scalar CPU for the RUU size of 32, and increases by having 64 and 128 entries in the RUU. However, the performance remains very close to that of the scalar CPU. The performance of CSI remains almost constant, and outperforms SSE by a 30-50% margin in the three tests (xform, light, geometry).

When comparing the instruction count reduction, SSE provides a reduction of 1.45, 1.13 and 1.15 but CSI is able to reduce this count by 30, 5.1 and 2.71 for the geometry test. So CSI is able to reduce the instruction traffic much better than SSE. The CSI ISA is able to express the computations by a much smaller number of instructions than SSE, being semantically richer. This behavior has also been observed for MOM, another ISA extension that is able to handle bi-dimensional data streams.

In order to see if the multimedia extensions scale well with an increased amount of execution hardware, the number of floating point numbers that can be operated upon in parallel is increased from 4 to 8 / 16 by adding new SSE execution units and by providing a wider datapath for CSI [6]. Also, the number of ports for the L1 cache is increased from 2 to 4 and the issue width is increased from 4 to 8.

The SSE configuration sees increased performance when having the issue width set to 8. This suggests that the main bottleneck for the SSE enhanced CPU is the issue width, as more ILP needs to be exploited in order to feed the SIMD units. And the 8-way processor can take much better advantage of having 4 cache ports compared to the 4-way processor, and that shows that processor's bottleneck is the number of cache ports. The performance improvements for SSE are most impressive in the geometry benchmark. CSI however doesn't benefit from the increased number of ports of the L1 cache. Some improvement exists for going 8-way, but these can be attributed to the non-SIMD portion of the code, as CSI instructions are capable of replacing entire kernels by a very small number of instructions. CSI can scale well for up to 16 floating point numbers processed in parallel, while SSE scales only to 8 floating point numbers, and that with consistent hardware costs.

Performance gains are high for both SSE and CSI, speedups of up to 3 can be achieved for example in the xform test. This shows the importance of having the right ISA when performing multimedia tasks.

Scalability tests involving integer data were performed in [7], comparing CSI against VIS. The benchmarks used consisted in MPEG-2 and JPEG tests from the Mediabench suite, and some image processing kernels from the VIS Software Development Kit (VSDK). The SimpleScalar toolset is used to simulate a 5-stage pipeline superscalar

processor. After updating the benchmark code to handle the multimedia extensions, test were performed for a CPU with the clock frequency set to 666 MHz, Issue width 4 / 8 / 16, Instruction Window size 16 - 512, Load Store Queue size 8-128. Also the amount of execution hardware was scaled.

The out of order execution of CSI instructions was not allowed, in order to avoid conflicting memory references.

When increasing the instruction window size, the processor using VIS extensions is reported to improve performance, as a result of the exploitation of Instruction Level Parallelism. When processing 2D streams using a VIS enhanced ISA, data must first be loaded into the registers and overhead instructions that control the loops must be placed in the source code. CSI on the other hand, being a memory to memory architecture, can perform the same work with much fewer instructions. It doesn't need to exploit ILP in order to obtain good performance. This means lower hardware costs can be obtained, and the embedded oriented application might choose this type of architecture as it is better suited for simpler designs. This can also be the case for CMP designs, where simpler cores are replicated in the same die, and the cores can be simpler, as a tradeoff can be made between single threaded performance of each cores and the number of functional units included in the chip.

When the issue width remains constant and the number of VIS resources is increased (adders and multipliers), the performance increase is zero. So it makes no sense for example to increase the size of the multimedia registers for VIS if the new CPU will not increase the issue width accordingly. This is partly because the window size was also increased (64 for 4-way, 128 for 8-way and 256 for the 16-way processor). CSI however has a spectacular performance increase when the datapath is increased, resulting in an almost linear speedup in most tests. A performance increased is also present when the issue width is increased, but it is only in the 10-15% range. This result doesn't surprise, as the way CSI is designed, it doesn't have to take advantage of ILP in any way in order to sustain performance.

When looking at the performance advantage of CSI over VIS, the difference decreases when going to the 8-way configuration and is minimum for 16-way. Speedups can be as high as 4.4 are present in the ycc_rgb test.

In order to identify the bottleneck for VIS, in [7] the average Instructions Per Cycle (IPC) is computed and compared to the ideal IPC. The results are very interesting: for the 4-way superscalar VIS processor, the IPC is very close to ideal, in the 80% - 90% range. So there's no much room left for further improvements on this configuration. When looking at the 8-way numbers, the IPC can fall to the 55% in the idct test, while it remains in the 75% - 85% intervals for the other tests. However, when going to the 16-way configuration, the IPC for idct is 40%, a very low percentage for the other kernels, IPC is within 55% to 70% of the ideal IPC. In order to improve the value of the IPC, the need for a very accurate branch predictor is suggested in [7].

The average speedup of CSI over VIS in full applications ranges from 1.08 in cjpeg to 1.54 in djpeg with an average of 1.24, for the issue width of 4. This is quite a large performance gain. For the VSDK kernels, speedup can be as high as 8.0 for the add8 kernel. Because the code executed for multimedia applications is unbalanced, having portions without any SIMD instructions, these large speedups are only accelerating parts

of the execution, and the improvements for full applications are not as impressive as the kernel ones.

In order to improve performance of a processor being able to execute SIMD instructions, two main directions can be followed:

- increase the issue width of the processor, but this can be an area and complexity hungry approach
- increase the width of the multimedia registers - this leads to the need of rewriting / recompiling existing software

CSI can improve performance by increasing the width of the datapath if needed, and software compatibility is maintained, as the number of elements processed in parallel is transparent for the software, and is implementation dependent.

Special purpose instructions can provide huge benefits in certain applications by minimal hardware costs. For example, real time MPEG-2 encoding requires a very fast CPU. Even if a fast CPU is available, this task will hog CPU resources and will prevent the user for having a rich multimedia experience, especially in the desktop systems that run dozens of programs simultaneously. The computer is the media center of the home, slowly replacing several devices in the digital home: the TV, the VCR, the CD player, the radio, etc. Certain tasks have to be executed efficiently in order to be able to have a sufficient reserve in order to make the system responsive.

For MPEG-2 encoding, 70-80% of the CPU cycles are spent computing the Sum of Absolute Differences when performing motion estimation. This simple operation can be performed by a small loop in software, but a hardware implementation is very fast. Such an approach for computing SAD is given in [32].

The SAD for a 16x16 block is defined as:

$$SAD\,(x, y, r, s) = \sum_{i=0}^{15}\sum_{j=0}^{15}|A_{(x+i,y+j)} - B_{((x+r)+i,(y+s)+j)}| \tag{2.11}$$

The coordinates of the current block (A) is given by (x, y). (r, s) is the motion vector (the displacement to the reference block B).

The C code from the MPEG Software Simulation Group:

```
s = 0;
for (j = 0; j < h; j++)
{
    if (( v = p1[0] - p2[0]) < 0 )
        v = -v;
    s+= v;
    if (( v = p1[1] - p2[1]) < 0 )
        v = -v;
    s+= v;
.
    if (( v = p1[15] - p2[15]) < 0 )
```

```
        v = -v;
    s+= v;
    if (s >= distlim)
        break;
    p1 += lx;
    p2 += lx;
}
```

If a single instruction can replace this code, the hardware can pre-fetch the required data so that the functional units are kept busy, hiding the memory latency.

A simple observation can make the hardware implementation much easier [32]:

$$|A_i - B_i| = max\,(A_i, B_i) - min\,(A_i, B_i) \tag{2.12}$$

Three simple steps are then required for obtaining the SAD:

1. Determine $\overline{min(A_i, B_i)}$ and $max(A_i, B_i)$ for the 16 pairs of numbers
2. Use a Carry Save Adder (CSA) tree to reduce the 32 numbers obtained in the previous step to 2 numbers using a Wallace or Dadda tree [27]. An additional compensation LSB needs to be introduced for each of the 16 outputs of the first stage, as $\overline{min(A_i, B_i)}$ is computed instead of $-min(A_i, B_i)$, so the compensation term is 16.
3. Use a fast adder to obtain the actual binary value of the SAD.

The first step gate implementation can look like Figure 2.3. In order to compute the minimum of A_i and B_i, a test subtraction can be carried out followed by an evaluation of the sign bit of the result. Note that computing the carry of an addition is cheaper and faster than performing the actual addition. The first number (A_i) is inverted and the carry generator computes the sign bit of the sum $\overline{A_i} + B_i$. The next step is to invert $min(A_i, B_i)$. If the output of the carry generator is '1', A_i was larger than B_i, so B_i has to be inverted and A_i has to be forwarded to the outputs unchanged. This is accomplished by the XOR gates present in Figure 2.3. The outputs O_1 and O_2 represent $\overline{min(A_i, B_i)}$ and $max(A_i, B_i)$ which are the inputs for the Wallace / Dadda tree present in Step2 of the SAD operation. This approach can be implemented as a 3 cycle operation, and one cycle length is estimated such that a 32-bit multiplication takes 2 cycles [32].

Another special purpose instruction that can be implemented is the Paeth prediction [5]. This is used in the Portable Network Graphics (PNG) image compression standard. The Paeth prediction scheme selects from 3 neighboring pixels a, b, and c that surround the d pixel the pixel that differs the least from the initial prediction p = a + b - c. The selection is called the Paeth prediction for d.

The C code for the Paeth prediction is [5]:

```
Void Paeth_predict_row(char *prev_row,
char *curr_row, char *predict_row, int length)
{
```

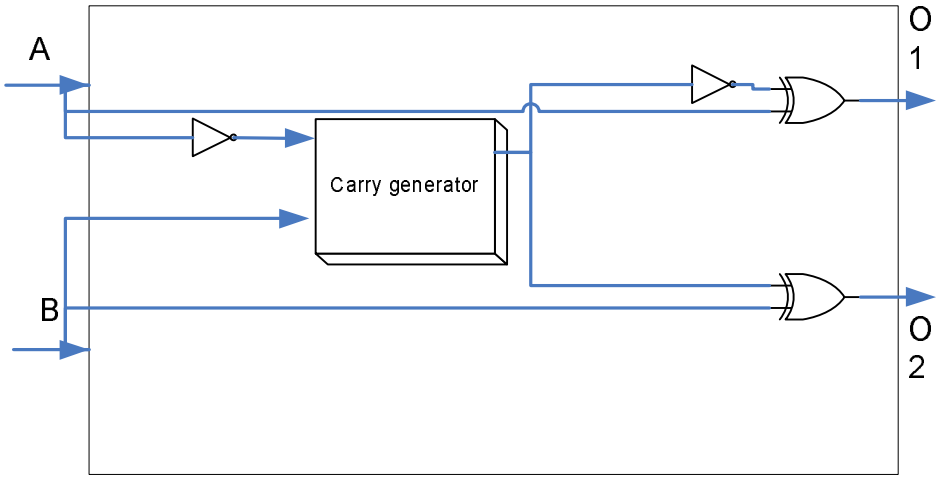


Figure 2.3: CSI:Gate implementation for the first step of SAD

```
char *bptr, *dptr, *predptr;
char a, b, c, d;

short p, pa, pb, pc;
//the initial prediction, and the differences
//for a, b, c

int i;

bptr = prev_row + 1;
dptr = curr_row + 1;
predptr = predict_row + 1;

for (i = 1; i < length; i++)
{
    c = *(bptr - 1);
    b = *bptr;
    a = *(dptr - 1);

    p = a + b - c;

    pa = abs (p - a);
    pb = abs (p - b);
    pc = abs (p - c);

    if ( (pa <= pb) && (pa <= pc)
        *predptr = a;
    else
```

Table 2.1: Paeth prediction: Relative positions of the a, b, c, d pixels

-	-	-	-
-	c	b	-
-	a	d	-
-	-	-	-

```
            if (pb <= pc)
                *predptr = b;
            else
                *predptr = c;
        bptr ++;
        dptr++;
        predptr++;
    }
}
```

The a, b ,c ,d pixels have the following relative positions (Table 2.1):

The special purpose csi_paeth instruction is introduced in [5]. The whole loop can be translated into a single csi_paeth instruction. The latency for this instruction is assumed to be two cycles.

A 1660 MHz processor was simulated using the sim-outorder simulator from the SimpleScalar toolset [2]. The benchmark used was the paeth_predict kernel. Three different configurations were compared: the scalar CPU, scalar + normal CSI and scalar + CSI with added support for Paeth prediction. Masked arithmetic was used for CSI because the kernel used data-depended control. For each CSI unit, two data path widths were tested: 128 bits and 256 bits.

The average instruction count for the kernel dropped from 4838.63 instructions for the superscalar processor to 239.92 instructions for simple CSI and further down to 44.0 instructions for the CSI with hardware support for Paeth prediction. Even if the main loop of the kernel was replaced by just one instruction, the auxiliary CSI instructions used to initialize the CSI registers make up the rest of the instructions.

If the number of cycles spent in the kernel is compared, CSI provides a speedup of 3.46X over the standard superscalar processor while adding the new csi_paeth instruction leads to a speedup of 44.20X, for the 128-bit wide datapath. For the 256-bit datapath, the speedups are 4.69X and 52.25X respectively. By having a wider data path, performance still increases by 35% for simple CSI and by 18% for the complex CSI (that includes csi_paeth). This shows that CSI can scale well with an increased number of hardware resources, even without performing additional changes to the superscalar core.

2.7 Chapter summary and conclusions

In this chapter, seven multimedia vector extensions have been presented. The concept used to design them is not fundamentally different from the classic vector instruction sets such as the one used in IBM/370.

The specialized instruction set extensions can bring high performance benefits at moderate hardware cost(around 10-15% of the die size for the existing implementations). However, having good vectorizing compilers can dramatically influence the execution time of the kernels, as previously shown regarding SSE two well known compilers (GCC and ICC).

The distinctive feature of MDMX is the 192-bit wide accumulator. This indeed solves the need for data promotion and demotion, but because a single accumulator register is supported it can lead to unnecessary data dependencies when performing reduction operations (MMX for example can use a binary tree approach in this case).

The Complex Stream Instructions approach avoids a number of limitations found in the other extensions, such as data packing and unpacking. The implementation details are hidden from the programmer, increasing portability of the code. Also, multiple hardware implementations can be created best suiting the targeted applications, all without changing the CSI enabled source code. Unlike MOM or MMX, CSI is a memory to memory architecture. This decision was based on the fact that there is little or no data reuse in streaming applications.

MMX^{TM}was the first extension introduced by Intel, and currently (as on 2007) plans exist to introduce the fourth version of SSE [29], showing that interest in SIMD extensions didn't slow down in the last 10 years.

3 ISA customization framework

This chapter presents the framework used for building the customized vector ISAs. The considered architecture is described, along with the steps required to customized the vector ISA. Details are given regarding the simulation enviroment and the simulator extensions.

3.1 Steps for customizing the vector ISA

The process of obtaining the custom instruction set for a set of applications is composed of several steps. First, the applications have to be profiled in order to identify the most time consuming operations and loops in the code. The next step is to decide on the portion of code that will be vectorized. The best way is to understand the algorithm behind that code, in order to produce a replacement sequence of instructions that performs the same job but using different (possibly custom) instructions.

The program can be envisioned as a list of operations that need to be performed. Each operation can be implemented in several ways, depending on the number of instructions available. Having to generate a custom instruction set means that a decision has to be taken regarding each operation, and the following factors influence the final set of operations that will be made available to the programmer:

- How often is the operation used?
- How complex are the operations in terms of possible hardware implementations?
- How well do the operations scale when the section size is increased?
- Are the new instructions part of the general purpose or custom categories?

After several applications are processed this way, the new ISA can be produced. The instructions that are used in most of the programs are general purpose, while the ones that are used only in a limited number of cases are the application specific operations.

The procedure for building a custom Vector ISA used here is as follows:

1. Profile the application;
2. Identify the most time consuming kernels of the application;
3. If the kernels are not suitable for vectorization, try rewriting the code. Also, the data structures used in the program can be modified in this step;
4. Vectorize selected parts of the application. In this step, decisions regarding splitting the functionality of a kernel between several instructions is taken;

5. Simulate in order to obtain information regarding the performance of the application after vectorization;
6. Move on to the next kernel;
7. If the performance improvement is not sufficient, return to step 2 and make another iteration.

3.2 Considered architecture

Figure 3.1 presents the block diagram of the considered Vector architecture. The Vector unit is composed of four main blocks: the vector control unit, the vector functional units, the vector registers and the vector memory units. The Fetch unit decides if the current instruction is scalar or it should be executed by the vector unit based on the decoded opcode. Note that the Vector unit is not modeled as a co-processor but it is tightly integrated along the Scalar unit. The main memory system is shared between the scalar and the vector units, but it is allowed for the Vector memory units to have larger memory bandwidth. The Vector unit doesn't include any data caches.

The modeled architecture is of register to register type, so all de data has to be loaded from memory first, and the final results have to be written to the memory. Figure 3.2 presents the architecture of the register file. Separate Integer and Floating point vector register files are modeled. Each Vector Register has an implicit Bit Vector, of the same length, used for implementing masked execution. One reason for choosing this implicit usage of the Bit Vectors was the fact that SimpleScalar [3] doesn't allow more than three operands for each instruction. A special configuration register file is used to store the Vector Length, the Vector Index, and a flag used for enabling or disabling the use of vector masks.

3.3 Simulation enviroment

In order to evaluate the performance improvement due to the vector operations for the targeted applications, a simulation environment was created. The OS used was Ubuntu Linux 6.10, kernel 2.6.17-10. The Simplescalar toolset included a specially modified version of GCC-2.7.2.3, capable of producing Portable Instruction Set Architecture (PISA) binaries for Simplescalar. The Sim-outorder simulator has been compiled with gcc-4.1.2.

We chose sim-outorder from the Simplescalar 3.0 tool set as a starting point. This is the most detail simulator in the SimpleScalar distribution. In order to simulate vector instructions, support for new instructions and functional added to sim-outorder.

In order to simplify the benchmarking process, another tool, SimpleScalar Instruction and Architecture Tool(SSIAT) [21], was used in order to automatically modify the Simplescalar source files.

3.3.1 Inline assembly and GCC

Manually vectorizing the test applications is a time-consuming operation. Fortunately, debugging and readability of the code was vastly improved thanks to the inline assembly

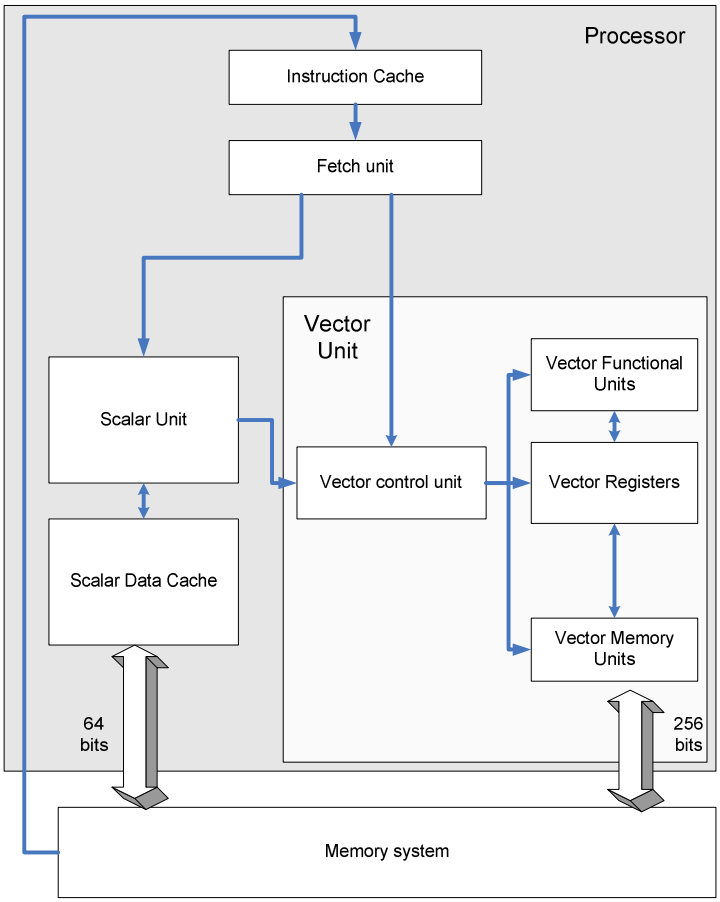


Figure 3.1: General Vector architecture

capabilities of GCC. For example, in order to vectorize a loop that initializes two arrays with the values NONE and chStart, the sequential code is:

```
for (i = 0; i <NUM_NODES; i++)
{
    d[i] = NONE;
    path[i] = chStart;
}
```

The vectorized code inside the C file:

```
__asm__ __volatile__ ("v.setvconf %0, %1, %2" : : "r" (NUM_NODES), "r" (0), "r" (0));
__asm__ __volatile__ ("v.init $1, %0" : : "r" (NONE));
__asm__ __volatile__ ("v.init $2, %0" : : "r" (chStart));
for (i = 0; i <NUM_NODES;)
{
```

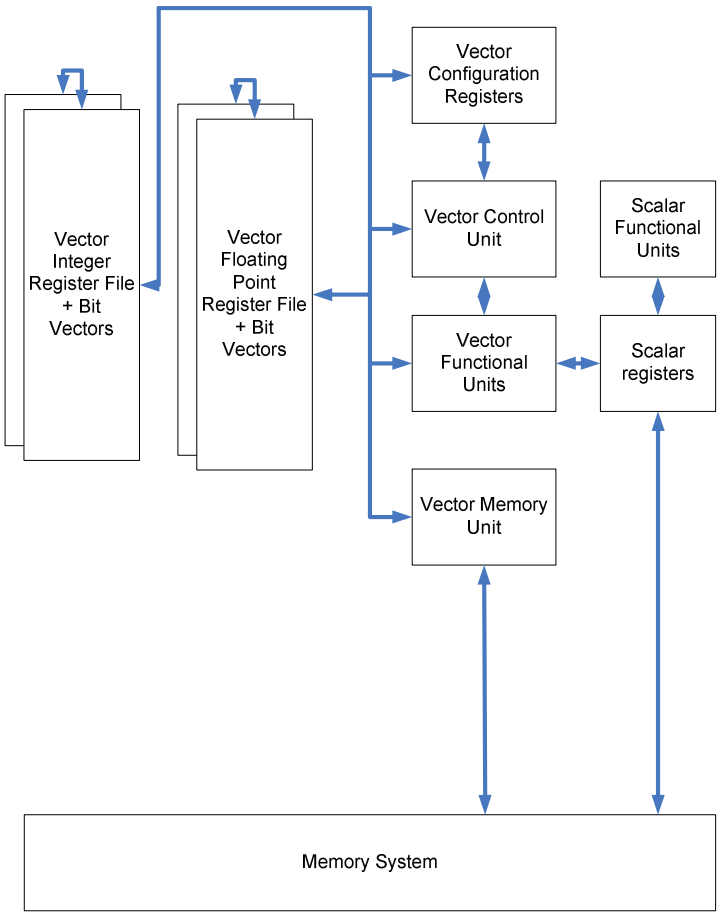


Figure 3.2: Vector Register File architecture

```
__asm__ __volatile__ (          \
"v.st %2, $1\n\t"               \
"v.st %3, $2\n\t"               \
"v.updatevl %0\n\t"             \
"v.updateindex %1\n\t"          \
:                               \
"=r" (kk),                      \
"=r" (i)                        \
:                               \
"r" (&d[i]),                    \
"r" (&path[i])                  \
);
```

}

The *__volatile__* keyword is used in order to prevent GCC for moving around and optimizing the code, because in this case GCC is not aware of the vector capabilities

of the simulator. The __asm__ statement is composed from three sections, thus ensuring that no hidden assumptions are made by the compiler: first, the assembly instructions in which the C variable names are replaced by %0, %1, etc., followed by the list of output variables and the list of input expressions. In the code example shown earlier, the variable **i** is placed in an output register and it is referenced in the **v.updateindex** instruction as **%1**, as it is the second C variable listed in the __asm__ statement. After the first ":", all the outputs have to be listed. Then another ":" marks the start of the input expressions. "r" denotes a floating point register is required for the operation, while "f" denotes the need of a floating point registers. Note that when working with double precision numbers, the floating point registers have to be even numbers. The compiler automatically generates code that moves data to/from the registers.

If the same variable is used both as input as output in an instruction, this has to be indicated in a certain way. If the loop that will be vectorized is:

```
min = INT_MAX;
pmin = -1;
for (j = 0; j <NUM_NODES; j++)
{
        if ((c[j] == 1) && (d[j] <min))
        {
                min = d[j];
                pmin = j;
        }
}
```

The vectorized code will be:

```
min = INT_MAX;
pmin = -1;
__asm__ __volatile__ ("v.setvconf %0, %1, %2" : : "r" (NUM_NODES), "r" (0), "r"
(1));
for (j = 0; j <NUM_NODES;)
{
        __asm__ __volatile__ (                          \
        "v.masksoff \n \t"                              \
        "v.ld $1, %4 \n \t"                             \
        "v.ld.bi $1, %5 \n \t"                          \
        "v.maskson \n \t"                               \
        "v.getmin $1, %2, %3 \n \t"                     \
        "v.updatevl %0 \n \t"                           \
        "v.updateindex %1 \n \t"                        \
        :                                               \
        "=r" (k),                                       \
        "=r" (j),                                       \
        "=r" (min),                                     \
        "=r" (pmin)                                     \
```

```
        :                                   \
        "r" (&d[j]),                        \
        "r" (&c[j]),                        \
        "r" (1),                            \
        "2" (min),                          \
        "3" (pmin)                          \
        );
}
```

The min and pmin variables are both inputs and outputs of the v.getmin instruction, written as "v.getmin $1, %2, %3 \n \t". So by noting in the outputs section "=r" (min) and "=r" (pmin), and in the inputs section "2" (min) and "3" (pmin) the compiler knows that it must preserve the values of the registers where the min and pmin variables are stored.

3.3.2 The benchmarking process

The process of benchmarking a specific application that has already been vectorized has to complete the following steps:

Step1. Obtain the assembly (.s) file from the C source file using the specially modified version of GCC (this version will be referred as gcc-ss for simplicity);

Step2. Write the desired section size parameter to the "section_size.h" header;

Step3. Generate the SSIAT configuration file. This file contains several sections, the most important being %REGISTERS, %FUNCTIONAL_UNITS and %MACHINE.DEF. Using this configuration file, SSIAT will modify the SimpleScalar source files;

Step4. Executing SSIAT. SSIAT modifies both the Simplescalar sources but also the .s file obtained in Step1. The reason is that the opcodes of the new instructions defined in the configuration file defined in Step3 have to be replaced by annotated instructions that can be recognized by a modified version of the GNU assembler (gas) found in the SimpleScalar package.

Step5. Use gcc-ss to obtain the binary from the SSIAT-modified .s file obtained at Step4.

Step6. (Re)Compile the modified version of Simplescalar, in order to support the new registers, functional units and instructions defined in Step3.

Step7. Simulate the binary file obtained in Step5 using the modified Simplescalar version obtained in Step6.

Following these steps make it possible to execute a benchmark that has already been vectorized. Due to the fact that a vectorizing compiler is not available, all the programs had to be vectorized manually, replacing the loops one by one by a block of assembly code.

In order to improve the readability but also the debugging process, the inline assembly feature of GCC has been used extensively. By using the __asm__ keyword, a block of assembly code can be used inside the C program. This provided access to all the variables in the scope of the __asm__ statement. However, the gcc-ss is limited to having a maximum of 10 parameters for each inline __asm__ block.

3.4 Simulator extensions

The simulator is extended by using SSIAT. This allows the addition of new instructions, functional units and register files.

3.4.1 The vector register file

A new register bank is declared using the keyword NEW_REGTYPE. The syntax is :

NEW_REGTYPE <register_bank_name> <number_of_bytes_per_register> <number_of_registers>

Separate integer and floating point vector register banks have been simulated. Each integer or floating point has an implicit bit vector register. This means that all the instructions that support the usage of masks will test the values of the corresponding bit vector of destination register of the instruction.

For 64 integer vector registers and 64 double-precision floating point registers, the %REGISTERS section should look like this:

NEW_REGTYPE vrd, 8, 64
NEW_REGTYPE vri, 4, 64
NEW_REGTYPE vbd, 4, 64
NEW_REGTYPE vbi, 4, 64
NEW_REGTYPE vconf, 4, 3

The proposed architecture includes 3 special configuration registers, declared in the "vconf" register bank. These registers are the Vector Length register, the Index register and the Vmasks register. Before the execution of vector instructions can proceed, initialization of the vconf registers in necessary. The Vector Length registers stores the number of elements that need to be processed in the current array. If the Vector Length is 0, the vector instruction won't process any data from the vector registers. The Index registers stores the index of the current section of the processed array. The Vmask register controls weather vector masks are used in the computation or not.

Because of limitations of de GNU assembler and of Simplescalar, the register banks declared cannot have more than 256 registers, each register having a maximum length of 256 bytes. In order to overcome this limitation, separate register banks have been declared inside sim_outorder.c, and the registers declared in this section have been used mainly to simulate data dependency between the instructions rather than the actual storage of elements.

3.4.2 The vector functional units

In order to simulate the latency of vector instructions, several new functional units have been added to the sim-outorder configuration. In order to declare a new functional unit, the syntax used is:

<name of fu> <quantity> <resource_class_string> <operation_latency_(cycles)> <issue_latency(cycles)>

The operation latency is the number of cycles before the results are ready to use, while the issue latency refers to the number of cycles that must pass before another operation can use this FU.

Five functional units have been declared: VECTOR - it is used mainly for the instructions that access the vconf register bank.

VALU, VMUL - the arithmetic units

VLD, VST - memory access units.

The latency of each unit has been computed taking into consideration the following parameters: the section size of the vector machine, the memory access latency for the vector operations, the number of parallel vector lanes and a constant additive latency of the arithmetic units (this latency will be referred as basic_latency from now on). We assumed those units to be pipelined.

The formulas of used to compute the latency and the issue_latency are the following:

- For the arithmetic units:

  Operation latency = section_size / number_of_lanes + basic_latency

  Issue latency = section_size / number_of_lanes

- For the memory units:

  - Loads:

    Operation latency = section_size / memory_bandwidth(in elements) + memory access latency

    Issue latency = section_size / memory_bandwidth

  - Stores:

    Operation latency = section_size / memory_bandwidth

    Issue latency = section_size / memory_bandwidth

An example section, considering memory access latency = 1 cycle, a single vector lane, a memory bandwidth of one element and section_size = 4, would be:

VECTOR, 1, Vector, 10, 1
VALU, 1, Vadd, 14, 4
VMUL, 1, Vmul, 14, 4
VLD, 1, Vld, 5, 4
VST, 1, Vst, 4, 4

3.4.3 The vector instructions

This section is used for declaring the new instructions. A special macro is used to define a new instruction, the DEFINST macro.

Syntax is: DEFINST (
<instruction name>,
<operands>,
<annotated instruction name>,
<FU class used by the instruction >,
<instruction flags>,
<output dependency reg. 1, output dependency reg. 2>,
<input dependency regs 1,2,3>

)

The way new instructions are added to Simplescalar is the following: the opcode field of the instructions defined with the original Simplescalar has a number of bits that are used for created annotated instructions. This simplifies the whole process of adding new instructions because the assembler used doesn't have to recognize new opcodes. Inside the simulation loop, the programmer checks the field of annotated instructions and can perform the semantics of the annotated instruction.

SSIAT operates in the following way: if new instructions are added in the %MACHINE.DEF section, it will automatically annotate some already defined instructions, according to the <annotated instruction name >field. This is transparent to the user, so while writing an application that uses the new operations, the programmer can just use custom opcodes for the non-standard operations used. SSIAT then modifies the sources of Simplescalar to add support for the annotated instructions. When compiling a new C program, assembly code must be generated first. SSIAT then parses this .s file and replaces the new opcodes with annotated instruction opcodes.

A short example:

The block of code in the .s file before running SSIAT is:

```
#APP
        v.setvconf $10, $11, $12
#NO_APP
```

After running SSIAT on this assembly source, this will become

```
#APP
        add/15:0(4) $10, $11, $12
#NO_APP
```

When the v.setconf instruction has been defined, the "add" instruction was chosen to be annotated in this case. One should always take care so that the annotated instruction chosen has the desired number of operands, and the operands are all either integer or floating point registers.

The definition of "v.setvconf":

DEFINST("v.setvconf", "d,s,t", "add",

Vector, F_ICOMP,

DSSAT_VCONF(VL), DSSAT_VCONF(VINDEX), DGPR(RS), DGPR(RT), DNA

)

Instruction dependency is modeled by the special DGPR macros for the normal scalar registers, and by the DSSAT macros defined by the use of SSIAT. Special macros are defined for accessing the operands inside the instruction. For example, RS will return the second operand of the 3-operand instruction, that has "d,s,t" operands.

Simplescalar is limited to having only 3 input and 2 output dependencies. While this is not ideal in some cases, it didn't affect the accuracy of the simulations performed. Simpler instructions have been defined instead.

Before defining the opcode and dependencies of a new instruction, the functionality of the instruction has to be defined. For the v.setvconf instruction, the implementation used was:

```
#define V_SETVCONF_IMPL {                                    \
X_i = GPR(RD);                                               \
memcpy(X_array1, &X_i, sizeof(int));                         \
SET_SSAT_VCONF(VL, X_array1);                                \
                                                             \
 X_i = GPR(RS);                                              \
memcpy(X_array1, &X_i, sizeof(int));                         \
SET_SSAT_VCONF(VINDEX, X_array1);                            \
                                                             \
X_i = GPR(RT);                                               \
memcpy(X_array1, &X_i, sizeof(int));                         \
SET_SSAT_VCONF(VMASK, X_array1);                             \
}
```

The opcodes can contain dots, and they are replaced by underscores when describing the functionality of the instruction. Syntax:

```
#define <OPCODE >_IMPL {
                    <C implementation>
}
```

3.5 Default parameters used for performance simulations

Unless specified otherwise for a specific test, the default configuration for the tests was:

- The memory system for the Vector Unit features a memory bandwidth of 4 elements / cycle, 2 Load and 2 Store units
- The vector data path is organized in 4 vector lanes, also featuring 2 multipliers and 2 ALUs
- The Vector register file included 64 integer + 64 floating point registers and their corresponding bit vectors
- Default optimizations used for GCC
- Default configuration for the Scalar Unit, as provided with the public version of SimpleScalar simplesim-3v0d

The default configuration of SimpleScalar as provided by the -dumpconfig parameter can be found in Appendix A.2.

3.6 Chapter Summary

The framework developed is flexible and it can be efficiently used to customize the instruction sets and evaluate the performance against scalar ISAs by simulations using the SimpleScalar toolset.

4 Applications description

This chapter presents the applications we have selected for vectorization: the three algorithms which compute the shortest paths in a directed graph (Dijkstra, Floyd and Bellman Ford) and Linpack, the floating point benchmark.

4.1 Dijkstra

The MIBENCH [1] suite has been analyzed in order to find the most promising applications for vectorization. However, not every application is suitable for vectorization. Code containing extensive use of pointers and data dependencies inside the loops often make the use of vector operations inefficient. The benchmarks are split into six categories: automotive, consumer, network, office, security, and telecommunications.

The most promising category of benchmarks was networking. Graph algorithms are often used networking. Dijkstra is a well known algorithm that computes the shortest path from a source node to all the other nodes inside a graph. But actually more algorithms exist to compute the shortest paths in a graph. Having the input graph and the predefined source and destination nodes, we tried to see which is the most vectorization-friendly algorithm and also which is the fastest one. For the Dijkstra and Floyd algorithms use the adjacency matrix to store the input graph, while the input for Bellman-Ford is the edge list of the graph.

The Dijkstra version present in the MIBENCH suite uses a linked list implemented with pointers to store the Candidate nodes in the algorithm. This makes the code virtually impossible to vectorize:

```
while (qcount() > 0) {
    dequeue (&iNode, &iDist, &iPrev);
    for (i = 0; i < NUM_NODES; i++)
    {
        if ((iCost = AdjMatrix[iNode][i]) != NONE)
    {
        if ((NONE == rgnNodes[i].iDist) ||
            (rgnNodes[i].iDist > (iCost + iDist)))
        {
            rgnNodes[i].iDist = iDist + iCost;
            rgnNodes[i].iPrev = iNode;
            enqueue (i, iDist + iCost, iNode);
        }
    }
    }
```

```
}
```

Instead of trying to optimize this code and vectorize it, a rewrite of the dijkstra() function was the chosen approach: arrays were used instead of linked lists, so the optimized version doesn't contain any pointers.

4.1.1 Profile information

After the algorithm was rewritten using arrays instead of linked lists, the application was profiled:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 99.66     13.23    13.23   100000   132.34   132.74  dijkstra
  0.30     13.27     0.04   100000     0.40     0.40  print_path
  0.30     13.31     0.04                             main
```

For each call of the dijkstra() function, a call to print_path() is made. However, it is clear that dijkstra() consumes most of the computation time.

4.1.2 Dijkstra kernel

The main loop of the modified Dijkstra algorithm becomes:

```
for (i = 0; i < NUM_NODES - 1; i++)
{
    //compute the minimum of [D] such as C[j]=1
    min = INT_MAX;
    pmin = -1;
    for (j = 0; j < NUM_NODES; j++)
    {
        if ((s[j] == 0) && (d[j] < min))
        {
            min = d[j];
            pmin = j;
        }
    }
    s[pmin] = 1;
    for (j = 0; j < NUM_NODES; j++)
    {
        //d[j] = MIN(d[j], d[pmin] + AdjMatrix[pmin][j]);
        if ((s[j] == 0) && (d[j] > d[pmin] + AdjMatrix[pmin][j]))
        {
            d[j] = d[pmin] + AdjMatrix[pmin][j];
            path[j] = pmin;
        }
```

```
    }
}
```

Two vectors hold the required information for the algorithm to work: d[] contains the best distances known at the current step in the algorithm from the source node to all the other nodes, while s[] contains a flag for each node, so that at each step the algorithm picks the node that has the smallest distance to the source vertex and that wasn't chosen in the previous steps (Dijkstra is a Greedy algorithm). The s[] array is similar to using a c[] array to mark which nodes were not chosen so far in order to update the d[] array.

The code is composed of one exterior loop and two interior for-loops. The first for loop computes the minimum and its position in the d[] array. At the first glance, two problems seem to appear: inside the loop we have an if clause, and the computation inside the loop doesn't involve elements of the arrays on the left hand side of the assignment operations.

4.2 Floyd

This algorithm computes all the minimum paths inside a graph. If V is the number of nodes in the graph, Floyd has a complexity of $O(V^3)$. While it can be used when only one minimum path is required, its most efficient use is when the graph is very dense and many minimum paths need to be computed. The computation is done only once, and the simplicity of the algorithm makes it a good example of vectorizable code.

4.2.1 Profiling information

Profiling this applications quickly points to the computation-intensive function:

```
 98.90      6.52     6.52     1000     6.52     6.52  floyd_compute
  1.37      6.61     0.09                             main
  0.00      6.61     0.00   100000     0.00     0.00  floyd_print
  0.00      6.61     0.00   100000     0.00     0.00  print_path
```

4.2.2 Floyd kernel

Taking a closer look inside the floyd_compute() function the main computation loop follows a small initialization step:

```
for (k = 0; k < NUM_NODES; k++)
        for (i = 0; i < NUM_NODES; i++)
            for (j = 0; j < NUM_NODES; j++)
                if (d[i][j] > d[i][k] + d[k][j])
                {
                    d[i][j] = d[i][k] + d[k][j];
                    path[i][j] = k;
                }
```

Here we have three nested loops. Only the most interior one can be vectorized.

4.3 Bellman-Ford

Bellman-Ford is an alternative to the well know Dijkstra shortest paths algorithm, which stores the graph as the list of edges and their corresponding weights. The chosen implementation of Dijkstra has a complexity of $O(V^2)$, where V is the number of nodes in the graph, while Bellman-Ford has a complexity of $O(V \cdot E)$, with E being the number of edges. When the graph is very dense, Dijkstra proves to be much faster, while in a sparse graph, the performance of Bellman-Ford improves dramatically.

However, there is another reason why Bellman-Ford is used in some situations: Dijkstras algorithm requires the edges to have non-negative weights, while with Bellman-Ford some of the edges can be negative. It is also guaranteed that after the Bellman-Ford algorithm completes, it can be detected weather the graph contains negative weight cycles. Applications of this algorithm include the use in Routing Information Protocol (RIP).

4.3.1 Profiling information

The profile is similar to the one seen for Dijkstra. Because the computation of the path consumes much more time compared the printing procedure, the profile report shows that 100.27% of the computation time is spent in the main bellman_ford() function.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.27     52.53    52.53    10000     5.25     5.25  bellman_ford
  0.00     52.53     0.00    10000     0.00     0.00  print_path
```

4.3.2 Bellman-Ford kernel

The main loop inside the bellman_ford() procedure processes all the edges in the graph V times (V is the number of nodes in the graph):

```
for (i = 0; i < NUM_NODES; i++) //repeat V times
{
    for (j = 0; j < no_of_edges; j++)       //examine all edges - RELAX
    {
        u = edge_src[j];
        v = edge_dest[j];
        w = edge_cost[j];
        if (d[u] + w < d[v])
        {
            d[v] = d[u] + w;
            path[v] = u;
        }
    }
}
```

The interior loop counts to no_of_edges, which for a very dense graph can be close to V^2. While this is the main reason this algorithm is slower than Dijkstra, it also provides some interesting possibilities of vectorization, because the achieved vector lengths can be much higher than for the other shortest paths algorithms.

But taking a closer look reveals a possible problem when trying to vectorize this loop: indirect addressing is used. Having support for this addressing mode is not the difficult part, but extra care must be taken in this type of situation because the indirect addressing is very similar to pointers: a compiler cannot know at compile-time what value is inside that pointer. It can be a negative value for example, or two indirections done in two iteration of the loop can provide a data dependency that stops the compiler from performing automatic vectorization of this loop.

Some restrictions have to be implemented so that vectorization is still possible and correct results are still guaranteed. One restriction that we imposed was one regarding the **v.stindexed** instruction. If in a certain position of the indirection register, the indirection value is negative, the store is not completed. This is not a major limitation of the usability of the instruction, but provides a mechanism to prevent the generation of incorrect results.

The computation flow of the interior loop is simple: parse all the edges, and perform the so-called edge relaxing operation: the d[] array maintains the best path costs known so far. Each edge is tested to see weather it can do something to improve the values of d[].

When a new value is written to d[], that value might be required as an input in the next iterations of the for loop. For the scalar version of the algorithm, this is not a problem. However, when working with the vectorized version of this algorithm, this Read after Write (RAW) dependency can lead to incorrect results if not handled correctly.

If the data written in d[] is used in another iteration of the vectorized Bellman Ford, then the algorithm produces the correct results. But if, for example, d[v] (from now on, an edge will be referred as the tuple <u, v, w> where u is the source node of the edge, v is the destination node and w is the weight of that edge) is updated in the 5th index of the current vector operation and the data in then read back in the 10th index of the operation, with the section size being larger then 11, the old value of d[v] will be used in the computation instead of the new one.

In order to avoid this situation, a special preprocessing step is proposed: the input file should not contain only the list of edges and their corresponding weight, but it should also contain the index where the destination node of that edge (v) appears for the first time in the edge list after the current position. If v is not found in the edge list, a negative value is stored instead. The **v.st.indexed** instruction has to be implemented in such a way that stores with negative indexes are not performed. This information is sufficient to have the program check during runtime if the updated position of d[] will be used as input values in the if() instruction contained in the inner-most for loop during the same iteration of the vectorized loop.

The time required to compute this extra information depends of how the edges are arranged in the input file. If the edge list is extracted from the adjacency matrix in a row-wise manner, then when searching for a specific v value in the edge list, two independent searches must be performed: one in the array containing all the u values

and one containing the v values. The u values in this case are sorted, and a simple binary search algorithm can be used to find the first occurrence of the value, having the complexity $O(log_2 E)$. The v values are not sorted but the array containing the v values is composed of sequences of sorted numbers. If a linear search is used in this case, the worst case complexity can be $O(E)$, but in average the complexity will be $O(V)$, when the graph is very dense. If the graph is sparse, the number of edges in the graph is close to the number of nodes, so the linear search will complete in $O(V)$ time. It can be noted that this preprocessing step doesn't change the $O(V * E)$ complexity of the whole algorithm.

Another modification to the algorithm in order to have built-in run-time detection of possible RAW hazards is the use of two arrays, each having E elements: possible_conflict[] and conflict_detect[]. Possible_conflict[] contains the extra information obtained in the preprocessing step. In each iteration of the vectorized loop, a section of possible_conflict[] is loaded into a register. Conflict_detect[] is initialized with zeroes before entering the inner loop of the algorithm. At each iteration of the vectorized inner loop, an indexed store is used to place the value (j + 1) at all the positions indicated by the current section of possible_conflict in conflict_detect[], where j is the current value of the index register, part of the **vconf** bank. This operation is masked by the bit vector produced by the v.compare.gt instruction that implements the if() test. The reason for doing the store is that it shows that updating d[] using the v numbers in the current section might produce RAW hazards because if those values will be read in this section, the old content of d[] will be used instead of the modified ones. Two condition must be met in order for a RAW hazard to occur: a modification at position v of d[] must be scheduled, and the first time this data is being read must be in the current section. The first condition is marked true or false by performing the actual comparison if (d[u] + w <d[v]). These comparisons are done in parallel and the results are stored in a bit vector. In order to check for the second condition, a section of conflict_detect[] starting from index j is loaded into a vector register. A test is conducted to compute the positions where this register holds the value (j + 1), as these positions are the ones where data is being read without having the proper version stored into memory yet.

If a RAW hazard has been detected, it means that all the computed results starting from the position of the RAW hazard has to be redone because the results might be corrupted.

4.4 Linpack

Linpack [14] is a well known floating point benchmark used to grade the performance of the supercomputers in the famous Top500 Supercomputer Sites [24]. It uses single or double precision data to perform some linear algebra operations on large matrixes. The version used as a base for the tests was the double precision one.

4.4.1 Profiling information

After profiling the application, it was clear that the daxpy() function would provide the best performance improvements if vectorized:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 75.04      0.03     0.03   106139     0.00     0.00  daxpy
 25.01      0.04     0.01       22     0.45     0.45  matgen
  0.00      0.04     0.00     2079     0.00     0.00  dscal
  0.00      0.04     0.00     2079     0.00     0.00  idamax
  0.00      0.04     0.00       21     0.00     1.40  dgefa
  0.00      0.04     0.00       11     0.00     0.06  dgesl
  0.00      0.04     0.00        1     0.00     0.00  dmxpy
  0.00      0.04     0.00        1     0.00     0.00  epslon
```

4.4.2 Linpack kernel

The main processing loop (which consists of a multiply-with-scalar-and-add operation) of daxpy() is:

```
for (i = 0; i < n; i++)
{
    expected_result[i] = dy[i] + da*dx[i];
}
```

Matgen() accounts for 25% of the total computation time, and it is used to generate the input matrixes. As the useful computation of the benchmark was the target for vectorization, a maximum theoretical speedup of 4X can be achieved in Linpack.

4.5 Chapter summary

The four targeted applications meet the two conditions needed for obtaining good results after vectorization: the profiling shows that a large part of the computation time is spent in a small kernel, and the data structures used make use of arrays to store the data (except for the version of Dijkstra found in the Mibench suite which we optimized by removing the pointers).

5 Experimental results

This chapter presents the customized instruction sets and the performance improvements obtained after vectorization for each targeted application as well as the merged ISA that has resulted.

5.1 The customized vector instruction sets

Tables 5.1, 5.2, 5.3 and 5.4 list the semantics and usage of the general vector instructions, while Table 5.5 lists the special purpose ones. A list of all the instructions sorted alphabetically is given in Tables 5.6 and 5.7.

The following notations have been used in the tables:

- The VR stands for Vector Register, and is followed by I for Integer and D four Double precision floating point;
- The R prefix is used when the operand is a scalar register;
- The VB prefix is used when the operand is a Bit Vector, and it is followed by I for Integer and D for Double to differentiate between the two bit vector banks;
- The D, S, and T suffixes are used to suggest the position of that register in the instruction format (similar to the convention used in the SimpleScalar PISA documentation).

5.1.1 General purpose vector instructions

The general purpose vector instructions are used in most of the applications. They can be categorized as:

- **Arithmetic instructions**: used to perform operations such as add and multiply having vector registers as inputs;
- **Bit vector instructions**: operations having bit vectors as operands, such as a **mov** instruction for bitvectors;
- **Memory instructions**: In order to move data to and from the memory, load and store instructions are used. Indirect addressing is supported;
- **Vector control instructions**: used to set the Vector Length, Vector Index and the usage of vector masks.

The **v.compare.gt VBID, VRIS, VRIT**(Table 5.1) instruction gets one Bit Vector and two integer Vector registers as parameters: VBID, VRIS and VRIT. The bit vector VBID serves both as the output register and as the vector mask (when masked execution is allowed). If the vector masks are enabled, the actual comparison between two elements of VRIS and VRIT is performed only if the corresponding bit from VBID is non-zero, otherwise the result for that position defaults to 0 (FALSE). The result is set to a logic TRUE for an index i only if $VRIS_i > VRIT_i$. The behavioral implementation we used is:

```
#define V_COMPARE_GT_IMPL {                                          \
    memcpy(&X_vl, SSAT_VCONF(VL), sizeof(int));                      \
    memcpy(&X_vmask, SSAT_VCONF(VMASK), sizeof(int));                \
    for (X_i = 0; X_i < MIN(X_vl, SECTION_SIZE); X_i++)              \
    {                                                                \
        if ( (vb_i[SSAT_VBID][X_i] != 0) || (X_vmask == 0) )         \
            if (vr_i[SSAT_VRIS][X_i] > vr_i[SSAT_VRIT][X_i])         \
                vb_i[SSAT_VBID][X_i] = 1;                            \
            else                                                     \
                vb_i[SSAT_VBID][X_i] = 0;                            \
        else                                                         \
            vb_i[SSAT_VBID][X_i] = 0;                                \
    }                                                                \
}
```

The **v.and.bi VBID, VBIS, VBIT** instruction (Table 5.2) performs the logic AND operations between the corresponding positions of VBIS and VBIT. This operations doesn't support vector masks. The implementation used is:

```
#define V_AND_BI_IMPL {                                          \
    memcpy(&X_vl, SSAT_VCONF(VL), sizeof(int));                  \
    for (X_i = 0; X_i < MIN(X_vl, SECTION_SIZE); X_i++)          \
    {                                          \
        if ( (vb_i[SSAT_VBIS][X_i] != 0) && (vb_i[SSAT_VBIT][X_i] != 0) ) \
            vb_i[SSAT_VBID][X_i] = 1;                            \
        else                                                     \
            vb_i[SSAT_VBID][X_i] = 0;                            \
    }                                                            \
```

The **v.ldindexed VRID, RS, VRIT** instruction (Table 5.3) performs indexed loading. VRID is the destination vector register, RS is a pointer to the first element of the source array, and VRIT is the index vector register. The operation performed is $VRID[i] \leftarrow RS[VRIT[i]]$. Note that masked execution mode is available, by using the bit vector corresponding to the VRID integer vector register. The C implementation is:

```
#define V_LDINDEXED_IMPL{
\
```

```
    X_u_k = GPR(RS);                                                \
    memcpy(&X_vl, SSAT_VCONF(VL), sizeof(int));                     \
    memcpy(&X_vmask, SSAT_VCONF(VMASK), sizeof(int));               \
    for (X_i = 0; X_i < MIN(X_vl, SECTION_SIZE); X_i++)             \
    {                                                               \
        if ( (vb_i[SSAT_VRID][X_i] != 0) || (X_vmask == 0) )        \
        {                                                           \
            X_u_j = X_u_k + vr_i[SSAT_VRIT][X_i] * sizeof(int); \
            X_u_l = READ_WORD(X_u_j, my_fault);                     \
            vr_i[SSAT_VRID][X_i] = X_u_l;                           \
        }                                                           \
    }                                                               \
    for (X_i = MIN(X_vl, SECTION_SIZE); X_i < SECTION_SIZE; X_i++) \
    {                                                               \
        vr_i[SSAT_VRIS][X_i] = 0;                                   \
    }                                                               \
}
```

The **v.masksoff** and **v.maskson** instructions (Table 5.4) set the corresponding control register VMASK, enabling or disabling the vector masks usage.

Dijkstra uses the specialized **v.getmin VRID, RS, RT** instruction. It is used to compute the minimum value from an integer array. VRID is the vector register of which the minimum value is computed. RS and RT are scala registers, used as initial values for the minimum and its position, and also served as the outputs of the operations, when a number smaller than the initial value RS is found. Masked execution is available by using the bit vector corresponding to the VRID register:

```
#define V_GETMIN_IMPL {                                             \
    X_min = GPR(RS);                                                \
    X_pmin = GPR(RT);                                               \
    X_needupdate = 0;                                               \
    memcpy(&X_vl, SSAT_VCONF(VL), sizeof(int));                     \
    memcpy(&X_vmask, SSAT_VCONF(VMASK), sizeof(int));               \
    for (X_i = 0; X_i < MIN(X_vl, SECTION_SIZE); X_i++)             \
    {                                                               \
        if ( (vr_i[SSAT_VRID][X_i] < X_min) &&                      \
        ( (vb_i[SSAT_VRID][X_i] != 0) || (X_vmask == 0) ) )         \
        {                                                           \
            X_min = vr_i[SSAT_VRID][X_i];                           \
            X_pmin = X_i;                                           \
            X_needupdate = 1;                                       \
        }                                                           \
    }                                                               \
    if (X_needupdate)                                               \
    {                                                               \
        SET_GPR(RS, X_min);                                         \
```

Table 5.1: Vector arithmetic instructions

Nr.	Instruction	Synopsis	Uses masks	OBS
1	v.init VRIS, RT	VRIS[] ← RT	yes	Used in: Dijkstra, Floyd, Bellman-Ford
2	v.sumup RS, VRIT	RS ← 0, RS += VRIT[i]	yes	Used in: Bellman-Ford
3	v.scompare.eq VBID, VRIS, RT	VBID[i] ← VRIS[i] == RT	yes	Used in: Bellman-Ford
4	v.compare.gt VBID, VRIS, VRIT	VBID[i] ← VRIS[i] > VRIT[i]	yes	Used in: Dijkstra, Floyd, Bellman-Ford
5	v.sadd VRID, RS, VRIT	VRID[i] ← RS + VRIT[i]	yes	Used in: Dijkstra, Floyd
6	v.add VRID, VRIS, VRIT	VRID[i] ← VRIS[i] + VRIT[i]	yes	Used in: Bellman Ford

Table 5.2: Bit Vector instructions

Nr.	Instruction	Synopsis	Uses masks	OBS
1	v.mov.bi VBIS, VBIT	VBIS ← VBIT	no	Used in: Dijkstra, Floyd, Bellman-Ford
2	v.and.bi VBID, VBIS, VBIT	VBID[i] ← VBIS[i] && VBIT[i]	no	Used in: Bellman-Ford

```
        memcpy(&X_i, SSAT_VCONF(VINDEX), sizeof(int));      \
        X_pmin += X_i;                                      \
        SET_GPR(RT, X_pmin);                                \
    }                                                       \
}
```

Table 5.3: Vector memory instructions

Nr.	Instruction	Synopsis	Uses masks	OBS
1	v.ld.bi VBIS, RT	VBIS[i] ← RT[i] == 0 ? 0 : 1	no	Used in: Dijkstra
2	v.ld VRIS, RT	VRIS[i] ← RT[i]	yes	Used in: Dijkstra, Floyd, Bellman-Ford
3	v.ldindexed VRID, RS, VRIT	VRID[i] ← RS[VRIT[i]]	yes	Used in: Bellman-Ford
4	v.ld.d VRDS, RT	VRDS[i] ← RT[i]	yes	Used in: Linpack
5	v.stindexed RD, VRIS, VRIT	RD[VRIT[i]] ← VRIS[i]	yes	Used in: Bellman-Ford
6	v.st RS, VRIT	RS[i] ← VRIT[i]	yes	Used in: Dijkstra, Floyd, Bellman-Ford
7	v.st.d RS, VRDT	RS[i] ← VRDT[i]	yes	Used in: Linpack

Table 5.4: Vector Control instructions

Nr.	Instruction	Synopsis	Uses masks	OBS
1	v.getss RS	RS ← section_size	no	Used in: Bellman-Ford debugging mode
2	v.masksoff	vmask ← 0	no	Used in: Dijkstra, Floyd, Bellman-Ford
3	v.maskson	vmask ← 1	no	Used in: Dijkstra, Floyd, Bellman-Ford
4	v.setvconf RD, RS, RT	vl ← RD, vindex ← RS, vmask ← RT	no	Used in: Dijkstra, Floyd, Bellman-Ford, Linpack
5	v.updatevl RS	vl ← RS ← MAX(0, vl - section_size)	no	Used in: Dijkstra, Floyd, Bellman-Ford, Linpack
6	v.updateindex RS	vindex ← RS ← vindex + section_size	no	Used in: Dijkstra, Floyd, Bellman-Ford, Linpac

Table 5.5: Application specific instructions

Nr.	Instruction	Synopsis	Uses masks	OBS
1	v.getmin VRID, RS, RT	RS ← MIN(VRID), RT ← pMIN(VRID)	yes	RS and RT serve as initial values for searching the minium in the array; if no value lower than RS is found, RS and RT are left untouched. Used in: Dijkstra
2	v.priority RD, VBIS, VBIT	RD ← position of first non-zero value of VBIT, VBIS[i] = 1 ← if i < RD, 0 otherwise	no	Used in: Bellman-Ford
3	v.msadd.d VRDD, FS, VRDT	VRDD[i] += FS * VRDT[i]	no	Used in: Linpack

5.1.2 Application specific instructions

The application specific instructions perform operations that are used in a limited number of situations. If those operations are on the critical path of the algorithm, a custom instruction can accelerate the program execution by replacing a number of general purpose instructions by the new, application specific one.

5.1.2.1 Compute the minimum from an array - v.getmin

A special purpose vector instruction to compute the minimum from an array is proposed, **v.getmin VR, min, pmin**. However, the problem is that normally an instruction operates only with the elements of the vector registers it receives as operands. This isn't an issue for computing the value of the minimum, but when computing its position, only the relative index of the minimum can be returned. Therefore, the index register inside the vconf bank of configuration registers is used. The min and pmin are input-output parameters. Their initial values serve as a reference value when computing the minimum, and if an element smaller than the reference value is found inside the vector register, min and pmin are updated accordingly.

The Dijkstra loop (section 4.1) can be rewritten using only vector instructions:

```
min = INT_MAX; pmin = -1;
```

Table 5.6: Vector instructions sorted alphabetically (1-12)

Nr.	Instruction	Synopsis	Uses masks	Used in
1	v.add VRID, VRIS, VRIT	VRID[i] ← VRIS[i] + VRIT[i]	yes	Bellman Ford
2	v.and.bi VBID, VBIS, VBIT	VBID[i] ← VBIS[i] && VBIT[i]	no	Bellman-Ford
3	v.compare.gt VBID, VRIS, VRIT	VBID[i] ← VRIS[i] > VRIT[i]	yes	Dijkstra, Floyd, Bellman-Ford
4	v.getmin VRID, RS, RT	RS ← MIN(VRID), RT ← pMIN(VRID)	yes	Dijkstra
5	v.getss RS	RS ← section_size	no	Bellman-Ford debugging mode
6	v.init VRIS, RT	VRIS[] ← RT	yes	Dijkstra, Floyd, Bellman-Ford
7	v.ld VRIS, RT	VRIS[i] ← RT[i]	yes	Dijkstra, Floyd, Bellman-Ford
8	v.ldindexed VRID, RS, VRIT	VRID[i] ← RS[VRIT[i]]	yes	Bellman-Ford
9	v.ld.bi VBIS, RT	VBIS[i] ← RT[i] == 0 ? 0 : 1	no	Dijkstra
10	v.ld.d VRDS, RT	VRDS[i] ← RT[i]	yes	Linpack
11	v.masksoff	vmask ← 0	no	Dijkstra, Floyd, Bellman-Ford
12	v.maskson	vmask ← 1	no	Dijkstra, Floyd, Bellman-Ford

Table 5.7: Vector instructions sorted alphabetically (13 - 24)

Nr.	Instruction	Synopsis	Uses masks	Used in
13	v.mov.bi VBIS, VBIT	VBIS ← VBIT	no	Dijkstra, Floyd, Bellman-Ford
14	v.msadd.d VRDD, FS, VRDT	VRDD[i] += FS * VRDT[i]	no	Linpack
15	v.priority RD, VBIS, VBIT	RD ← position of first non-zero value of VBIT, VBIS[i] = 1 ← if i < RD, 0 otherwise	no	Bellman-Ford
16	v.sadd VRID, RS, VRIT	VRID[i] ← RS + VRIT[i]	yes	Dijkstra, Floyd
17	v.scompare.eq VBID, VRIS, RT	VBID[i] ← VRIS[i] == RT	yes	Bellman-Ford
18	v.setvconf RD, RS, RT	vl ← RD, vindex ← RS, vmask ← RT	no	Dijkstra, Floyd, Bellman-Ford, Linpack
19	v.st RS, VRIT	RS[i] ← VRIT[i]	yes	Dijkstra, Floyd, Bellman-Ford
20	v.stindexed RD, VRIS, VRIT	RD[VRIT[i]] ← VRIS[i]	yes	Bellman-Ford
21	v.st.d RS, VRDT	RS[i] ← VRDT[i]	yes	Linpack
22	v.sumup RS, VRIT	RS ← 0, RS += VRIT[i]	yes	Bellman-Ford
23	v.updateindex RS	vindex ← RS ← vindex + section_size	no	Dijkstra, Floyd, Bellman-Ford, Linpack
24	v.updatevl RS	vl ← RS ← MAX(0, vl - section_size)	no	Dijkstra, Floyd, Bellman-Ford, Linpack

```
__asm__ __volatile__ ("v.setvconf %0, %1, %2" : : \
                "r" (NUM_NODES), "r" (0), "r" (1));
for (j = 0; j < NUM_NODES;) {
    __asm__ __volatile__ (          \
    "v.masksoff\n\t"                \
    "v.ld $1, %4\n\t"               \
    "v.ld.bi $1, %5\n\t"            \
    "v.maskson\n\t"                 \
    "v.getmin $1, %2, %3\n\t"       \
    "v.updatevl %0\n\t"             \
    "v.updateindex %1\n\t"          \
    :                               \
    "=r" (k),                       \
    "=r" (j),                       \
    "=r" (min),                     \
    "=r" (pmin)                     \
    :                               \
    "r" (&d[j]),                    \
    "r" (&c[j]),                    \
    "r" (1),                        \
    "2" (min),                      \
    "3" (pmin)                      \
    );

}
```

The **v.setvconf** instruction is required to properly initialize the three configuration registers: the vector length, the vector index, and the flag that turns the usage of bit vectors as masks on or off.

First, a section of the d[] and c[] vectors are loaded into vector registers. Then the custom v.getmin instruction is used to update the min and pmin variables. After the computation is done, the vector length and index are updated. Notice that because min and pmin are also read by the v.getmin instruction, an extra constraint has to be added in the last two positions of the input parameter section of __asm__.

The second interior loop tries to optimize the d[] array by using the best available candidate node. The path[] array which is used to keep track of the actual path in the graph that leads to the minimum cost must also be updated. The loop is then rewritten:

```
__asm__ __volatile__ ("v.setvconf %0, %1, %2" : : \
            "r" (NUM_NODES), "r" (0), "r" (1));
for (j = 0; j < NUM_NODES;) {
    __asm__ __volatile__(           \
    "v.masksoff\n\t"                \
    "v.ld $2, %2\n\t"               \
    "v.ld $1, %3\n\t"               \
    "v.ld $4, %8\n\t"               \
```

```
    "v.sadd $5, %5, $2\n\t"         \
    "v.ld.bi $5, %4\n\t"            \
    "v.maskson\n\t"                 \
    "v.compare.gt  $5, $1, $5\n\t"  \
    "v.mov.bi $1, $5\n\t"           \
    "v.sadd $1, %6, $5\n\t"         \
    "v.mov.bi $4, $5\n\t"           \
    "v.init $4, %7\n\t"             \
    "v.masksoff\n\t"                \
    "v.st %3, $1\n\t"               \
    "v.st %8, $4\n\t"               \
    "v.updatevl %0\n\t"             \
    "v.updateindex %1\n\t"          \
    :                               \
    "=r" (k),                       \
    "=r" (j)                        \
    :                               \
    "r" (&AdjMatrix[pmin][j]),      \
    "r" (&d[j]),                    \
    "r" (&c[j]),                    \
    "r" (d[pmin]),                  \
    "r" (0),                        \
    "r" (pmin),                     \
    "r" (&path[j])                  \
    );
}
```

The presence of the c[] vector inside the if statement is mapped naturally to a bit vector. The comparison (d[j] >d[pmin] + AdjMatrix[pmin][j])) is performed by the v.compare.gt instruction, and a positive result is possible only when c[j] is non zero. c[] acts like a bit mask for the v.compare.gt operation. The path[] is updated with the pmin value only for the indexes where the optimization condition is true. This functionality is obtained by initializing a register with the pmin value on the position where v.compare.gt returned TRUE.

5.1.2.2 Vector multiply with scalar and add - v.msadd

The v.msadd instruction is used to accelerate the daxpy() Linpack loop. The operation performed is of the multiply and accumulate type. The vectorized version of this loop is:

```
    idx = n;
    i = 0;
    __asm__ __volatile__ ("v.setvconf %0, %1, %2" : : \
            "r" (n), "r" (0), "r" (0));
    while(idx)
```

```
{
    __asm__ __volatile__ (              \
        "v.ld.d $0, %2\n\t"             \
        "v.ld.d $2, %3\n\t"             \
        "v.msadd.d $f0, %4, $f2 \n\t"   \
        "v.st.d %2, $0\n\t"             \
        "v.updatevl %0\n\t"             \
        "v.updateindex %1\n\t"          \
        :                               \
        "=r" (idx),                     \
        "=r" (i)                        \
        :                               \
        "r" (&dy[i]),                   \
        "r" (&dx[i]),                   \
        "f" (da)                        \
        );
}
```

The loop doesn't contain any if() statements, so the only thing that has to be added to the instruction set is the **v.msadd.d VRDD, FS, VRDT** which performs the actual multiply and accumulate operation. Even if this operation could have been done with a separate multiply and add operation, the performance gain by having a specialized instruction is justified.

5.1.2.3 Detect the first non-zero element of a bitvector - v.priority

The detection of the RAW hazards in Bellman Ford was needed to vectorize the kernel. Having the modified input file facilitated the runtime checking weather such a hazard is occurring in the current section of the input vectors.

The specialized instruction (**v.priority RD, VBIS, VBIT**) processes the bitvector containing the positions of the detected hazards and extracts the position of the first non zero value and also generates a new bit vector that has the form [1 1 1 ..1 0 0 0], having non zeroes for the positions having correct results. This allows the program to store only the correct results into memory and restart the processing from the first position that used the out of date values of d[]. The processing will be restarted from the index where the first corrupt data was used. This is not the best solution because some correct results will be dropped and redone. However, in terms of hardware complexity it provides a more realistic approach. If unlimited hardware resources could be allocated to this task, a full index comparison could have been performed in order to see exactly which of the results are corrupt and make sure that only those values are corrected. This comparison scales poorly because $O(section\_size^2)$ comparisons must be performed. In the conducted experiments, the need to restart the processing occurred in a small percentage of the cases. If the section size value approaches the number of edges of the graph, instead of processing the inner loop in a single step, it will be restarted over and over again. There is a tradeoff between the speedup generally offered by section sizes that are close to the problem size and the discarding of results because of read after write hazards.

Considering these observations, Bellman Ford can then be vectorized:

```
/*
            %0 = retry
            %1 = &d[0]
            %2 = &edge_src[j]
            %3 = &edge_dest[j]
            %4 = &edge_cost[j]
            %5 = &path[0]
            %6 = &possible_conflict[0]
            %7 = &conflict_detect[0]
            %8 = &conflict_detect[j]
            %9 = j
*/


            __asm__ __volatile (           \
            "v.masksoff\n\t"               \
            "v.ld $5, %2\n\t"              \
            "v.ld $6, %3\n\t"              \
            "v.ld $9, %6\n\t"              \
            "v.ldindexed $1, %1, $5\n\t"   \
            "v.ldindexed $2, %1, $6\n\t"   \
            "v.ld $3, %4\n\t"              \
            "v.add $4, $1, $3\n\t"         \
            "v.compare.gt $4, $2, $4\n\t"  \
            "v.mov.bi $8, $4\n\t"          \
            "v.init $8, %9\n\t"            \
            "v.maskson\n\t"                \
            "v.stindexed %7, $8, $9\n\t"   \
            "v.masksoff\n\t"               \
            "v.ld $10, %8\n\t"             \
            "v.scompare.eq $11, $10, %9\n\t"\
            "v.priority %0, $12, $11\n\t"  \
            "v.and.bi $4, $4, $12\n\t"     \
            "v.mov.bi $5, $4\n\t"          \
            "v.maskson\n\t"                \
            "v.stindexed %1, $4, $6\n\t"   \
            "v.stindexed %5, $5, $6\n\t"   \
            :                              \
            "=r" (retry)                   \
            :                              \
            "r" (&d[0]),                   \
            "r" (&edge_src[j]),            \
            "r" (&edge_dest[j]),           \
```

```
"r" (&edge_cost[j]),            \
"r" (&path[0]),                 \
"r" (&possible_conflict[j]),    \
"r" (&conflict_detect[0]),      \
"r" (&conflict_detect[j]),      \
"r" (j+1)                       \
);
```

Also, the updating of the vector length and the index registers has to be adapted to reflect the possibility to recompute parts of the results:

```
if (retry == -1)
{//continue normally
    __asm__ __volatile__(       \
    "v.updatevl %0\n\t"         \
    "v.updateindex %1\n\t"      \
    :                           \
    "=r" (kk),                  \
    "=r" (j)                    \
    :                           \
    );
}
else    //drop all results starting from
        //and including the retry index
{
    j = j + retry;
    __asm__ __volatile__ ("v.setvconf %0, %1, %2" : : \
            "r" (kk - retry), "r" (j), "r" (0));
    no_of_retrys++;
}
```

In the end, the vectorization of Bellman-Ford was possible, but with some noticeable overhead compared to the other two shortest paths algorithms investigated.

5.2 Simulation results

In this section we compare the performance of the custom instruction sets against a standard scalar ISA. Two micro-architectural parameters have been varied in order to evaluate the performance characteristics of the ISAs: the section size and the memory latency of the vector memory unit.

5.2.1 Dijkstra

The MIBENCH version of Dijkstra was very slow compared to the optimized one. After rewriting Dijkstra with arrays instead of linked lists, a speedup of 5.76X was obtained,

Table 5.8: Custom vector ISA used for Dijkstra

Nr.	Instruction	Synopsis	Uses masks	OBS
1	v.compare.gt VBID, VRIS, VRIT	VBID[i] ← VRIS[i] > VRIT[i]	yes	
2	v.getmin VRID, RS, RT	RS ← MIN(VRID), RT ← pMIN(VRID)	yes	RS and RT serve as initial values for searching the minium in the array; if no value lower than RS is found, RS and RT are left untouched
3	v.init VRIS, RT	VRIS[] ← RT	yes	
4	v.ld VRIS, RT	VRIS[i] ← RT[i]	yes	
5	v.ld.bi VBIS, RT	VBIS[i] ← RT[i] == 0 ? 0 : 1	no	
6	v.masksoff	vmask ← 0	no	
7	v.maskson	vmask ← 1	no	
8	v.mov.bi VBIS, VBIT	VBIS ← VBIT	no	
9	v.sadd VRID, RS, VRIT	VRID[i] ← RS + VRIT[i]	yes	
10	v.setvconf RD, RS, RT	vl ← RD, vindex ← RS, vmask ← RT	no	
11	v.st RS, VRIT	RS[i] ← VRIT[i]	yes	
12	v.updateindex RS	vindex ← RS ← vindex + section_size	no	
13	v.updatevl RS	vl ← RS ← MAX(0, vl - section_size)	no	

an indication of how important are the data structures used. The modified version could also be vectorized much easier. Table 5.8 lists the custom instruction set used for Dijkstra.

After vectorization, the maximum speedup compared to the pointer version is 24.88X for the Section Size of 128, and compared to the modified (but still scalar) version a maximum speedup of 4.31X(Figures 5.2, 5.3, 5.5 and 5.6), again for Section Size 128 and Memory Latency set to 1 cycle. The measurements have been performed for the whole application, not just for the Dijkstra kernel.

The average vector length has a large impact on the expected performance of a vector processor. If the vector length is much lower than the section size of the machine, the arrays will be processed in a single pass, but performance is low because a large section size imply long execution latencies of the vector instructions. Having a vector length much larger than the Section Size keeps the vector units busy, but performance can be further improved by implementing a larger Section Size. The best performance is obtained when the vector lengths are close to the Section Size, but not greater than it.

Because the number of nodes in the graph tested is equal to 100, it was expected to get the best results for section sizes of 64 and 128. If the section size is increased beyond 128, the performance drops, showing how important the relation between the Section Size and the problem size is.

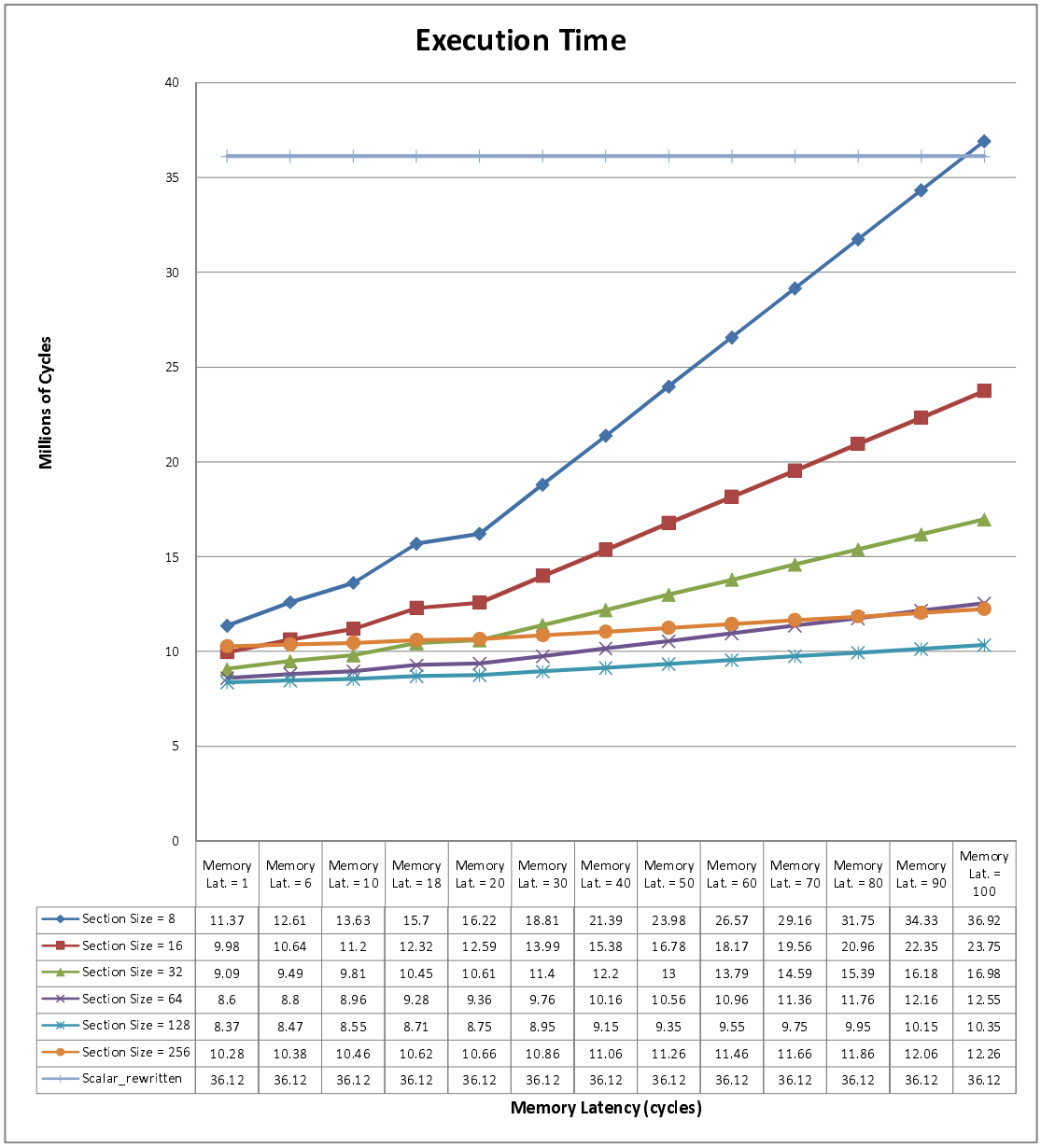
By increasing the Section Size, the impact of increased memory latency is much lower (Figure 5.1). This can assure that by using high bandwidth memory but with slightly larger latency can lead to satisfactory results. If the memory latency is fixed to 100 cycles, having a Section Size of 8 is more than three times slower compared to Section Size 128 and the same latency. For a memory latency of 1 cycle (equivalent to a perfect Level1 data cache), the difference is less than 50%.

As it can be seen in Figure 5.1 the performance starts to drop (the number of executed cycles increases) sharply when the Memory Latency exceeds 18-20 cycles. Also, the difference in execution time between a mem. latency of 1, 6 and 18 cycles is relatively low, and becomes insignificant for large section sizes. For example, a configuration having a Section Size of 128 finishes execution in 8.37 millions of cycles of the Memory Latency set to 1 cycle, and slows down to 8.47 millions cycles for latency equal to 6 cycles and 8.55 millions for 18 cycles latency. The difference is 1.19% respectively 2.15%, within the error margin of the simulator.

Figure 5.4 shows how choosing the Section Size affects the performance. For a fixed Memory Latency of 1 cycle, increasing the Section Size from 8 to 64 improves the execution time by $\frac{11.37-8.37}{11.37} \cdot 100\% = 26.38\%$. However, the same comparison done when the memory latency is 18 cycles gives a difference of 44.5%.

Figures 5.2, 5.3, 5.5 and 5.6 show the speedups obtained by the vectorized Dijkstra over the other versions. The speedup almost doubles by going from a Section Size of 8 to 128 for a memory latency of 18 cycles, and drops from 4.14X to 3.4X if the Section Size is 256, a similar performance of having the Section Size 32 and the same memory latency.

The customized instruction set for Dijkstra provided consistent improvements of the execution time that justified the hardware and software support necessary to actually apply vectorization in a real system.



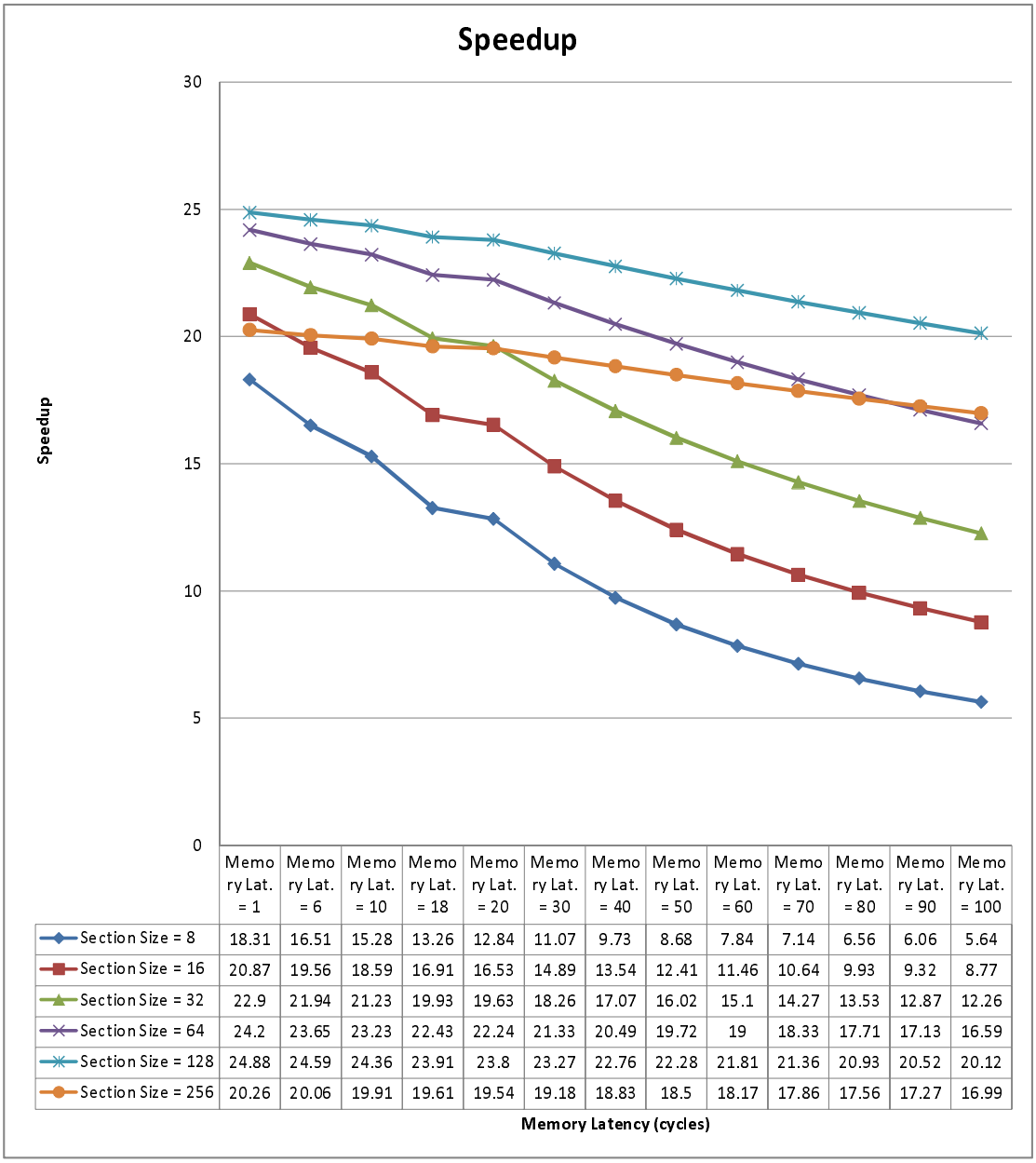
	Memory Lat. = 1	Memory Lat. = 6	Memory Lat. = 10	Memory Lat. = 18	Memory Lat. = 20	Memory Lat. = 30	Memory Lat. = 40	Memory Lat. = 50	Memory Lat. = 60	Memory Lat. = 70	Memory Lat. = 80	Memory Lat. = 90	Memory Lat. = 100
Section Size = 8	11.37	12.61	13.63	15.7	16.22	18.81	21.39	23.98	26.57	29.16	31.75	34.33	36.92
Section Size = 16	9.98	10.64	11.2	12.32	12.59	13.99	15.38	16.78	18.17	19.56	20.96	22.35	23.75
Section Size = 32	9.09	9.49	9.81	10.45	10.61	11.4	12.2	13	13.79	14.59	15.39	16.18	16.98
Section Size = 64	8.6	8.8	8.96	9.28	9.36	9.76	10.16	10.56	10.96	11.36	11.76	12.16	12.55
Section Size = 128	8.37	8.47	8.55	8.71	8.75	8.95	9.15	9.35	9.55	9.75	9.95	10.15	10.35
Section Size = 256	10.28	10.38	10.46	10.62	10.66	10.86	11.06	11.26	11.46	11.66	11.86	12.06	12.26
Scalar_rewritten	36.12	36.12	36.12	36.12	36.12	36.12	36.12	36.12	36.12	36.12	36.12	36.12	36.12

Figure 5.1: Dijkstra execution time when varying the memory latency

5.2.2 Floyd

The Floyd algorithm completed the task of computing the shortest paths in a graph faster than Dijkstra and Bellman Ford. After vectorizing the code using the instructions listed in Table 5.9 the maximum speedup achieved is 4.99X for Section Size = 128, and a Memory Latency of 1 cycle (Figures 5.8 and 5.10). If Memory Latency is set to 18 cycles, reported speedup is 4.86X, very close to the maximum achieved.

The input graph used for Floyd is the same one used also for Dijkstra. Best per-

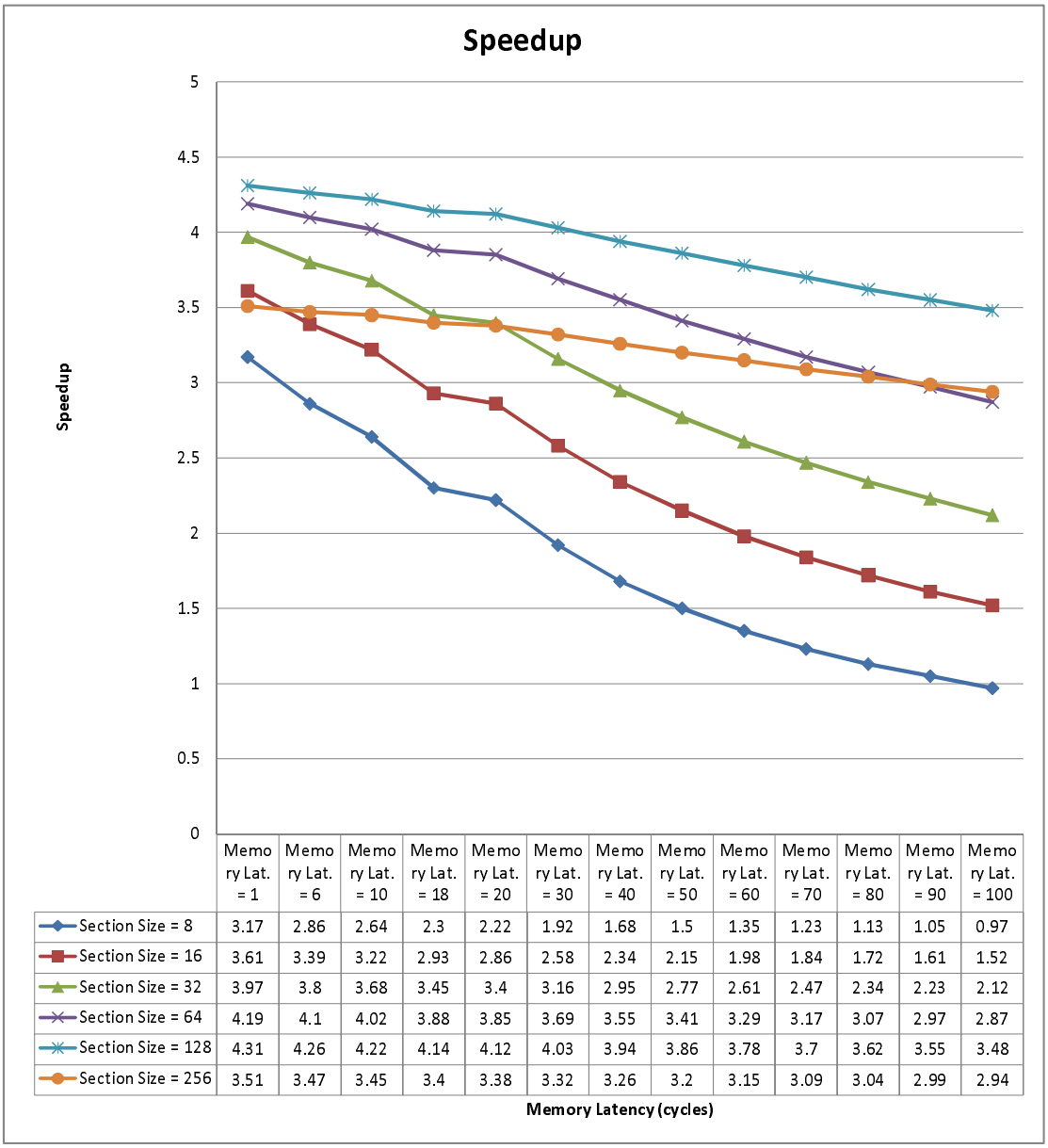


	Memory Lat. = 1	Memory Lat. = 6	Memory Lat. = 10	Memory Lat. = 18	Memory Lat. = 20	Memory Lat. = 30	Memory Lat. = 40	Memory Lat. = 50	Memory Lat. = 60	Memory Lat. = 70	Memory Lat. = 80	Memory Lat. = 90	Memory Lat. = 100
Section Size = 8	18.31	16.51	15.28	13.26	12.84	11.07	9.73	8.68	7.84	7.14	6.56	6.06	5.64
Section Size = 16	20.87	19.56	18.59	16.91	16.53	14.89	13.54	12.41	11.46	10.64	9.93	9.32	8.77
Section Size = 32	22.9	21.94	21.23	19.93	19.63	18.26	17.07	16.02	15.1	14.27	13.53	12.87	12.26
Section Size = 64	24.2	23.65	23.23	22.43	22.24	21.33	20.49	19.72	19	18.33	17.71	17.13	16.59
Section Size = 128	24.88	24.59	24.36	23.91	23.8	23.27	22.76	22.28	21.81	21.36	20.93	20.52	20.12
Section Size = 256	20.26	20.06	19.91	19.61	19.54	19.18	18.83	18.5	18.17	17.86	17.56	17.27	16.99

Figure 5.2: Dijkstra speedup when varying the memory latency compared to the original MIBENCH version

formance is obtained for a Section Size of 128, and the drop in performance for having slower memory (latency of 18 cycles instead of 1 cycle) for this Section Size is 2.65%. As it can be observed in Figures 5.7 and 5.8, performance drops sharply as the Memory Latency increases above 20 cycles, especially for Section Sizes of 8 and 16.

For a fixed Section Size 128, the execution time difference between a Memory Latency of 1 and 100 is 15% (Figures 5.7 and 5.10). Performing the same calculus for Section

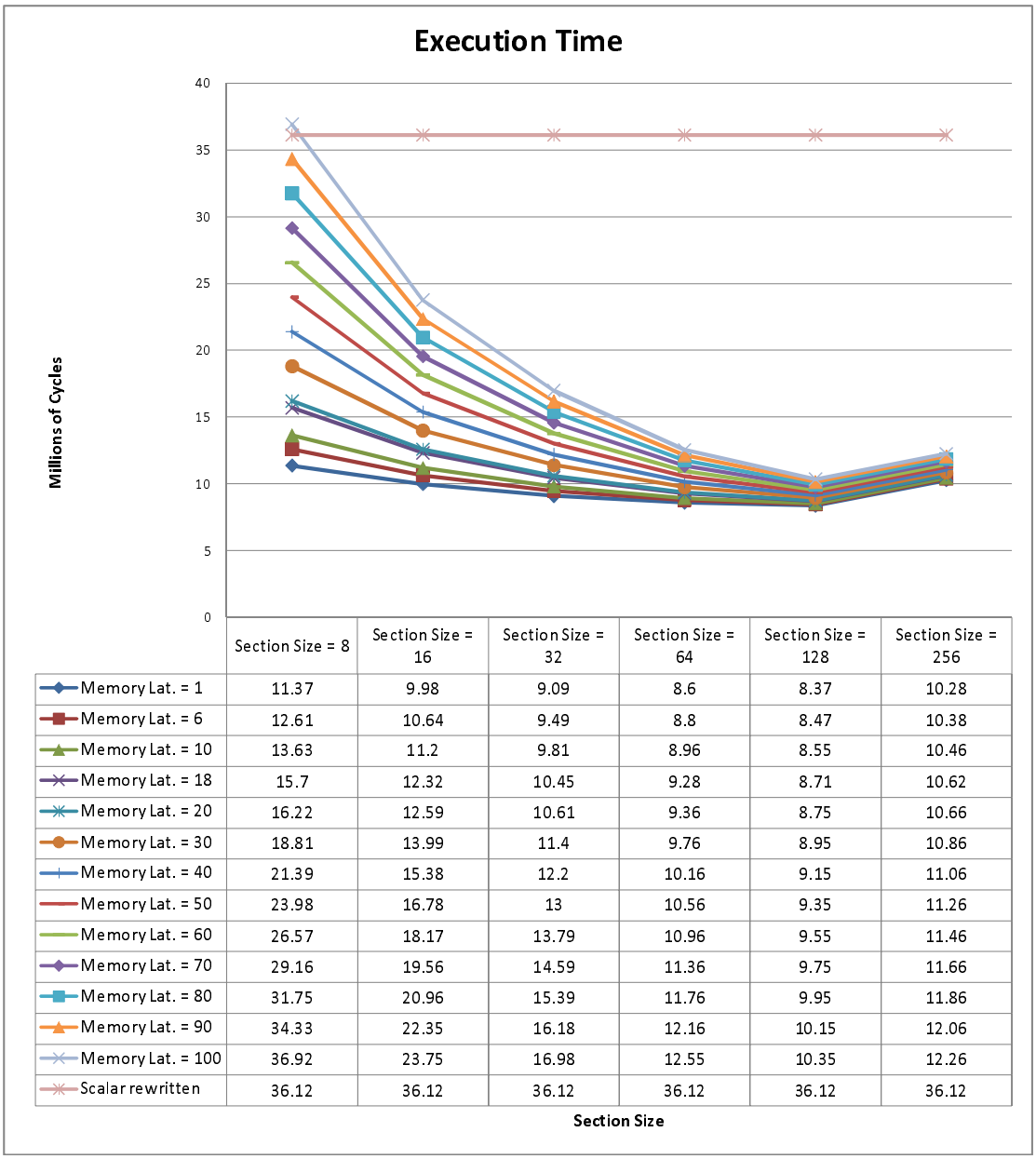


	Memory Lat. = 1	Memory Lat. = 6	Memory Lat. = 10	Memory Lat. = 18	Memory Lat. = 20	Memory Lat. = 30	Memory Lat. = 40	Memory Lat. = 50	Memory Lat. = 60	Memory Lat. = 70	Memory Lat. = 80	Memory Lat. = 90	Memory Lat. = 100
Section Size = 8	3.17	2.86	2.64	2.3	2.22	1.92	1.68	1.5	1.35	1.23	1.13	1.05	0.97
Section Size = 16	3.61	3.39	3.22	2.93	2.86	2.58	2.34	2.15	1.98	1.84	1.72	1.61	1.52
Section Size = 32	3.97	3.8	3.68	3.45	3.4	3.16	2.95	2.77	2.61	2.47	2.34	2.23	2.12
Section Size = 64	4.19	4.1	4.02	3.88	3.85	3.69	3.55	3.41	3.29	3.17	3.07	2.97	2.87
Section Size = 128	4.31	4.26	4.22	4.14	4.12	4.03	3.94	3.86	3.78	3.7	3.62	3.55	3.48
Section Size = 256	3.51	3.47	3.45	3.4	3.38	3.32	3.26	3.2	3.15	3.09	3.04	2.99	2.94

Figure 5.3: Dijkstra speedup when varying the memory latency compared to the rewritten version

Size 8 we get a difference of 135%. When the Section Size is large, the startup cost of the vector operation can be amortized over a larger number of elements. When programming with a vector ISA, a decision must be made: if the vector length is too short, the scalar unit may provide better performance because high startup costs can cancel out any performance enhancement.

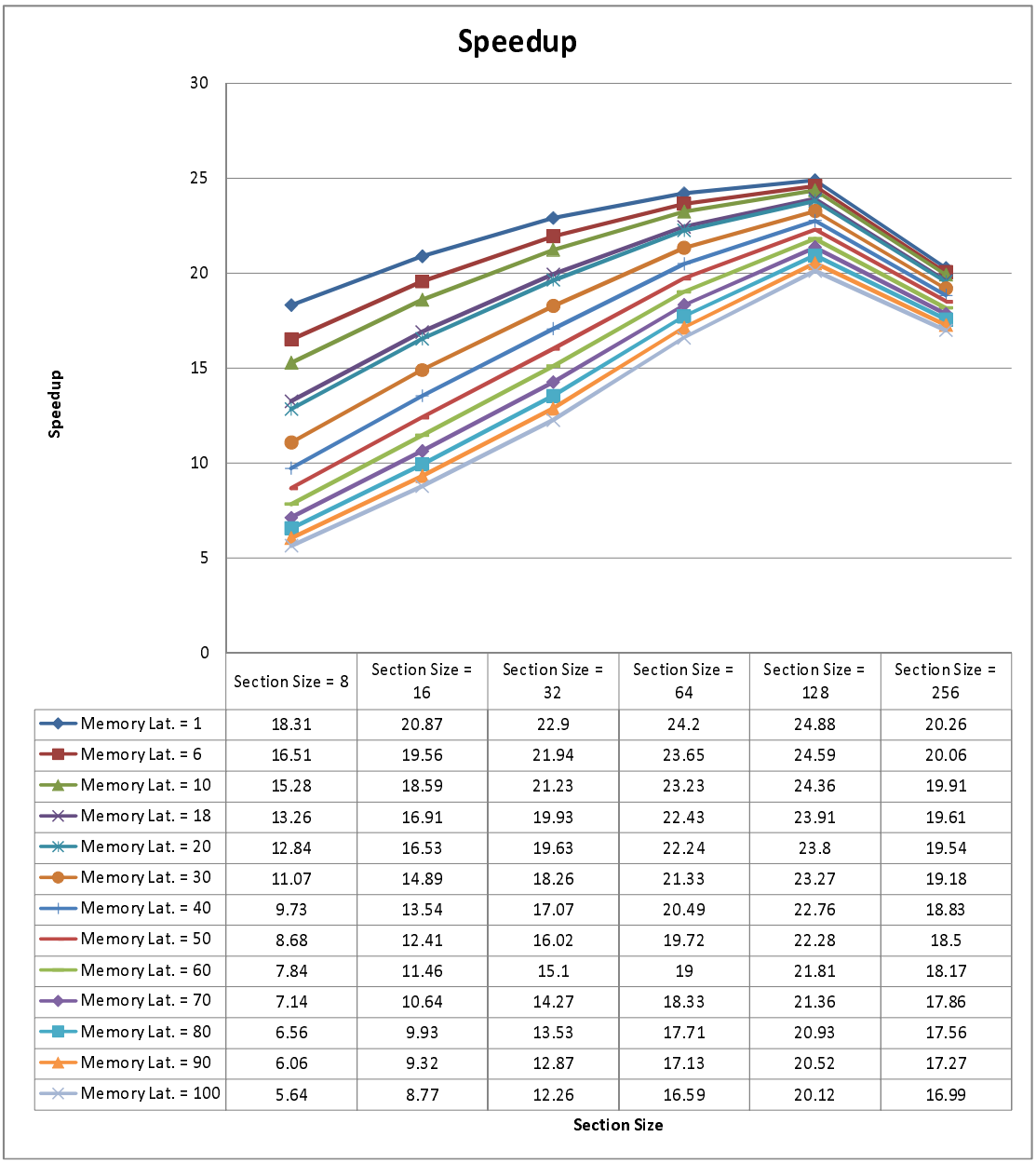
If a perfect Level1 data cache is simulated, a high speedup can be obtained without having a large Section Size (Figures 5.8 and 5.10. The highest speedup for this configu-



	Section Size = 8	Section Size = 16	Section Size = 32	Section Size = 64	Section Size = 128	Section Size = 256
Memory Lat. = 1	11.37	9.98	9.09	8.6	8.37	10.28
Memory Lat. = 6	12.61	10.64	9.49	8.8	8.47	10.38
Memory Lat. = 10	13.63	11.2	9.81	8.96	8.55	10.46
Memory Lat. = 18	15.7	12.32	10.45	9.28	8.71	10.62
Memory Lat. = 20	16.22	12.59	10.61	9.36	8.75	10.66
Memory Lat. = 30	18.81	13.99	11.4	9.76	8.95	10.86
Memory Lat. = 40	21.39	15.38	12.2	10.16	9.15	11.06
Memory Lat. = 50	23.98	16.78	13	10.56	9.35	11.26
Memory Lat. = 60	26.57	18.17	13.79	10.96	9.55	11.46
Memory Lat. = 70	29.16	19.56	14.59	11.36	9.75	11.66
Memory Lat. = 80	31.75	20.96	15.39	11.76	9.95	11.86
Memory Lat. = 90	34.33	22.35	16.18	12.16	10.15	12.06
Memory Lat. = 100	36.92	23.75	16.98	12.55	10.35	12.26
Scalar rewritten	36.12	36.12	36.12	36.12	36.12	36.12

Figure 5.4: Dijkstra execution time when varying the section size

ration is 4.99X for Section Size of 128 and 4.09X for a section size of 16. However, in the Floyd algorithm, after reading the data from the memory, only a few simple operations are performed and the results are stored into the memory. This is a behavior close to the streaming applications, and data caches are not very effective in this cases, unless data can be pre-fetched in time by the compiler.



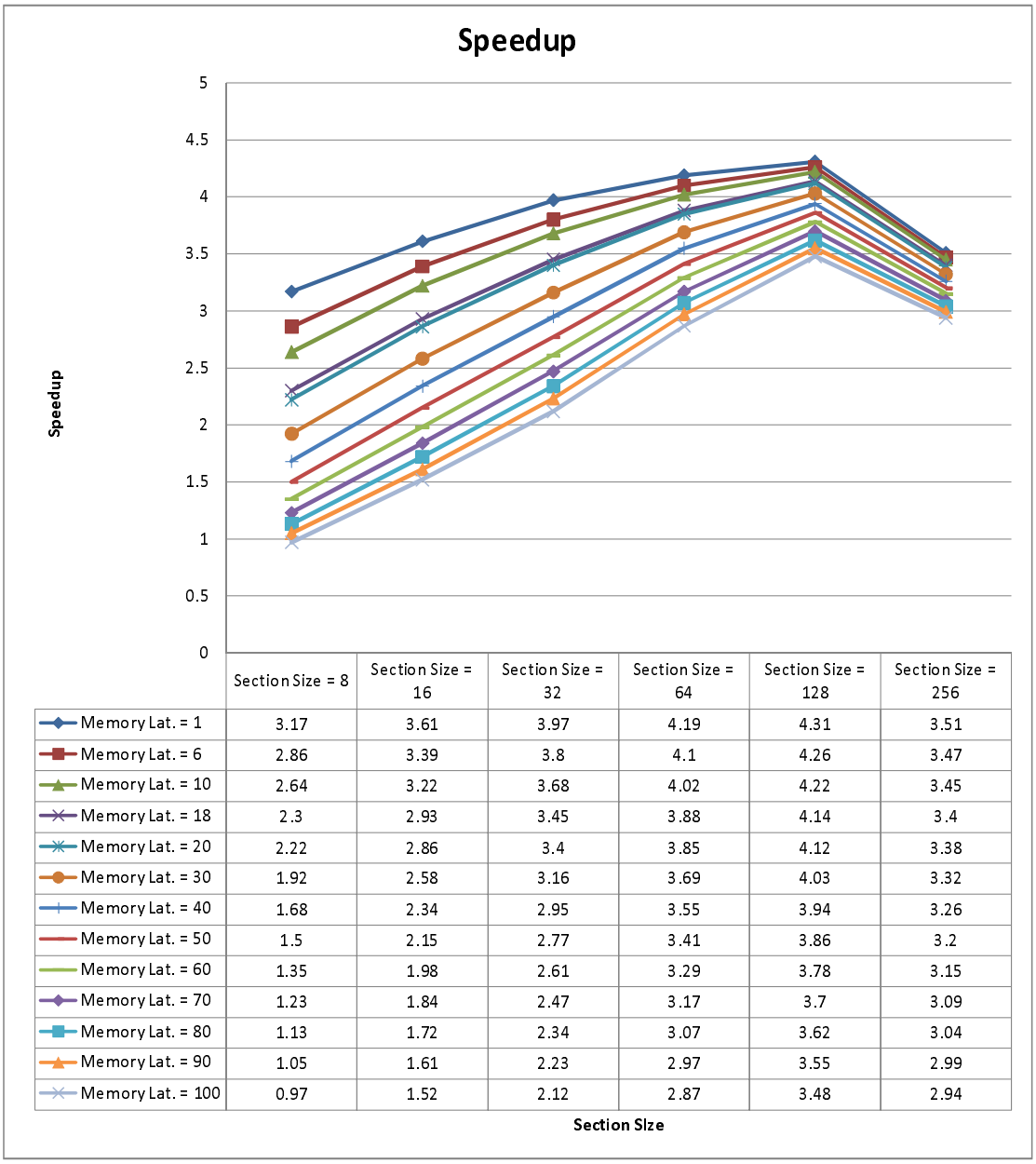
	Section Size = 8	Section Size = 16	Section Size = 32	Section Size = 64	Section Size = 128	Section Size = 256
Memory Lat. = 1	18.31	20.87	22.9	24.2	24.88	20.26
Memory Lat. = 6	16.51	19.56	21.94	23.65	24.59	20.06
Memory Lat. = 10	15.28	18.59	21.23	23.23	24.36	19.91
Memory Lat. = 18	13.26	16.91	19.93	22.43	23.91	19.61
Memory Lat. = 20	12.84	16.53	19.63	22.24	23.8	19.54
Memory Lat. = 30	11.07	14.89	18.26	21.33	23.27	19.18
Memory Lat. = 40	9.73	13.54	17.07	20.49	22.76	18.83
Memory Lat. = 50	8.68	12.41	16.02	19.72	22.28	18.5
Memory Lat. = 60	7.84	11.46	15.1	19	21.81	18.17
Memory Lat. = 70	7.14	10.64	14.27	18.33	21.36	17.86
Memory Lat. = 80	6.56	9.93	13.53	17.71	20.93	17.56
Memory Lat. = 90	6.06	9.32	12.87	17.13	20.52	17.27
Memory Lat. = 100	5.64	8.77	12.26	16.59	20.12	16.99

Figure 5.5: Dijkstra speedup when varying the section size compared to the original MIBENCH version

5.2.3 Bellman-Ford

When comparing the execution time of Bellman Ford with the ones of Dijkstra and Floyd, it can easily be seen that this is the slowest algorithm in this situation (Figures 5.11 and 5.13).

The main reason is that the graph used for benchmarking is very dense, making the complexity of Bellman-Ford $O(V^3)$, much slower than Dijkstra ($O(V^2)$) or Floyd($O(V^3)$



	Section Size = 8	Section Size = 16	Section Size = 32	Section Size = 64	Section Size = 128	Section Size = 256
Memory Lat. = 1	3.17	3.61	3.97	4.19	4.31	3.51
Memory Lat. = 6	2.86	3.39	3.8	4.1	4.26	3.47
Memory Lat. = 10	2.64	3.22	3.68	4.02	4.22	3.45
Memory Lat. = 18	2.3	2.93	3.45	3.88	4.14	3.4
Memory Lat. = 20	2.22	2.86	3.4	3.85	4.12	3.38
Memory Lat. = 30	1.92	2.58	3.16	3.69	4.03	3.32
Memory Lat. = 40	1.68	2.34	2.95	3.55	3.94	3.26
Memory Lat. = 50	1.5	2.15	2.77	3.41	3.86	3.2
Memory Lat. = 60	1.35	1.98	2.61	3.29	3.78	3.15
Memory Lat. = 70	1.23	1.84	2.47	3.17	3.7	3.09
Memory Lat. = 80	1.13	1.72	2.34	3.07	3.62	3.04
Memory Lat. = 90	1.05	1.61	2.23	2.97	3.55	2.99
Memory Lat. = 100	0.97	1.52	2.12	2.87	3.48	2.94

Figure 5.6: Dijkstra speedup when varying the section size compared to the rewritten version

but the kernels runs only once and completes all the shortest paths in the graph in a single pass).

The maximum speedup obtained by using the 18 vector instructions listed in Table 5.10 is 9.27X for Section Size = 64 and Memory Latency = 1 cycle (Figures 5.12 and 5.14). For Memory Latency = 18 cycles, the speedup remains high (8.18X), almost the double of the speedups seen for the other graph algorithms.

Table 5.9: Custom vector ISA used for Floyd

Nr.	Instruction	Synopsis	Uses masks	OBS
1	v.compare.gt VBID, VRIS, VRIT	VBID[i] ← VRIS[i] > VRIT[i]	yes	
2	v.init VRIS, RT	VRIS[] ← RT	yes	
3	v.ld VRIS, RT	VRIS[i] ← RT[i]	yes	
4	v.masksoff	vmask ← 0	no	
5	v.maskson	vmask ← 1	no	
6	v.mov.bi VBIS, VBIT	VBIS ← VBIT	no	
7	v.sadd VRID, RS, VRIT	VRID[i] ← RS + VRIT[i]	yes	
8	v.setvconf RD, RS, RT	vl ← RD, vindex ← RS, vmask ← RT	no	
9	v.st RS, VRIT	RS[i] ← VRIT[i]	yes	
10	v.updateindex RS	vindex ← RS ← vindex + section_size	no	
11	v.updatevl RS	vl ← RS ← MAX(0, vl - section_size)	no	

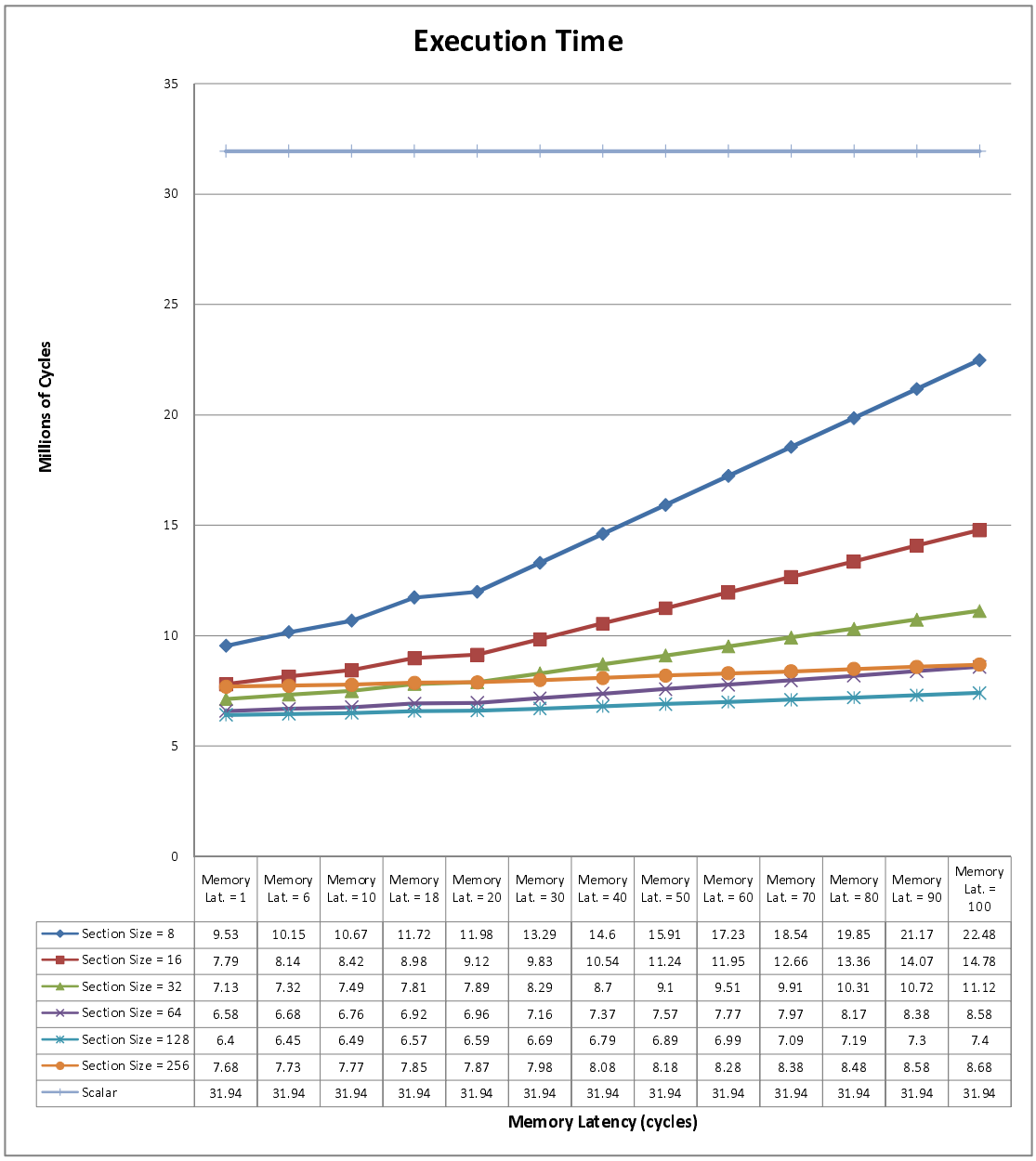
This algorithm proves to be much more sensitive to high memory latency than Dijkstra. The difference in execution time is 128.5% for the Section Size set to 64 and 519.86% for the small Section Size of 8 when increasing the memory latency from 1 cycle to 100 cycles(Figures 5.11 and 5.13).

Having a large Section Size is really important (Figures 5.12 and 5.14). If the memory latency is 18 cycles, the speedup increases from 2.34X for a Section Size of 8 to 8.18X for the Section Size of 64. This is an almost linear increase in performance. The reason for this is that the number of operations inside the vectorized loop is larger compared to the other two graph algorithms considered.

If the problem has be solved by having negative weights present, the vectorized version of Bellman-Ford is very fast. Otherwise, the other graph algorithms perform the computation much faster.

Table 5.10: Custom vector ISA used for Bellman Ford

Nr.	Instruction	Synopsis	Uses masks	OBS
1	v.add VRID, VRIS, VRIT	VRID[i] ← VRIS[i] + VRIT[i]	yes	
2	v.and.bi VBID, VBIS, VBIT	VBID[i] ← VBIS[i] && VBIT[i]	no	
3	v.compare.gt VBID, VRIS, VRIT	VBID[i] ← VRIS[i] > VRIT[i]	yes	
4	v.getss RS	RS ← section_size	no	
5	v.init VRIS, RT	VRIS[] ← RT	yes	
6	v.ld VRIS, RT	VRIS[i] ← RT[i]	yes	
7	v.ldindexed VRID, RS, VRIT	VRID[i] ← RS[VRIT[i]]	yes	
8	v.masksoff	vmask ← 0	no	
9	v.maskson	vmask ← 1	no	
10	v.mov.bi VBIS, VBIT	VBIS ← VBIT	no	
11	v.priority RD, VBIS, VBIT	RD ← position of first non-zero value of VBIT, VBIS[i] = 1 ← if i < RD, 0 otherwise	no	
12	v.scompare.eq VBID, VRIS, RT	VBID[i] ← VRIS[i] == RT	yes	
13	v.setvconf RD, RS, RT	vl ← RD, vindex ← RS, vmask ← RT	no	
14	v.st RS, VRIT	RS[i] ← VRIT[i]	yes	
15	v.stindexed RD, VRIS, VRIT	RD[VRIT[i]] ← VRIS[i]	yes	
16	v.sumup RS, VRIT	RS ← 0, RS += VRIT[i]	yes	
17	v.updateindex RS	vindex ← RS ← vindex + section_size	no	
18	v.updatevl RS	vl ← RS ← MAX(0, vl - section_size)	no	

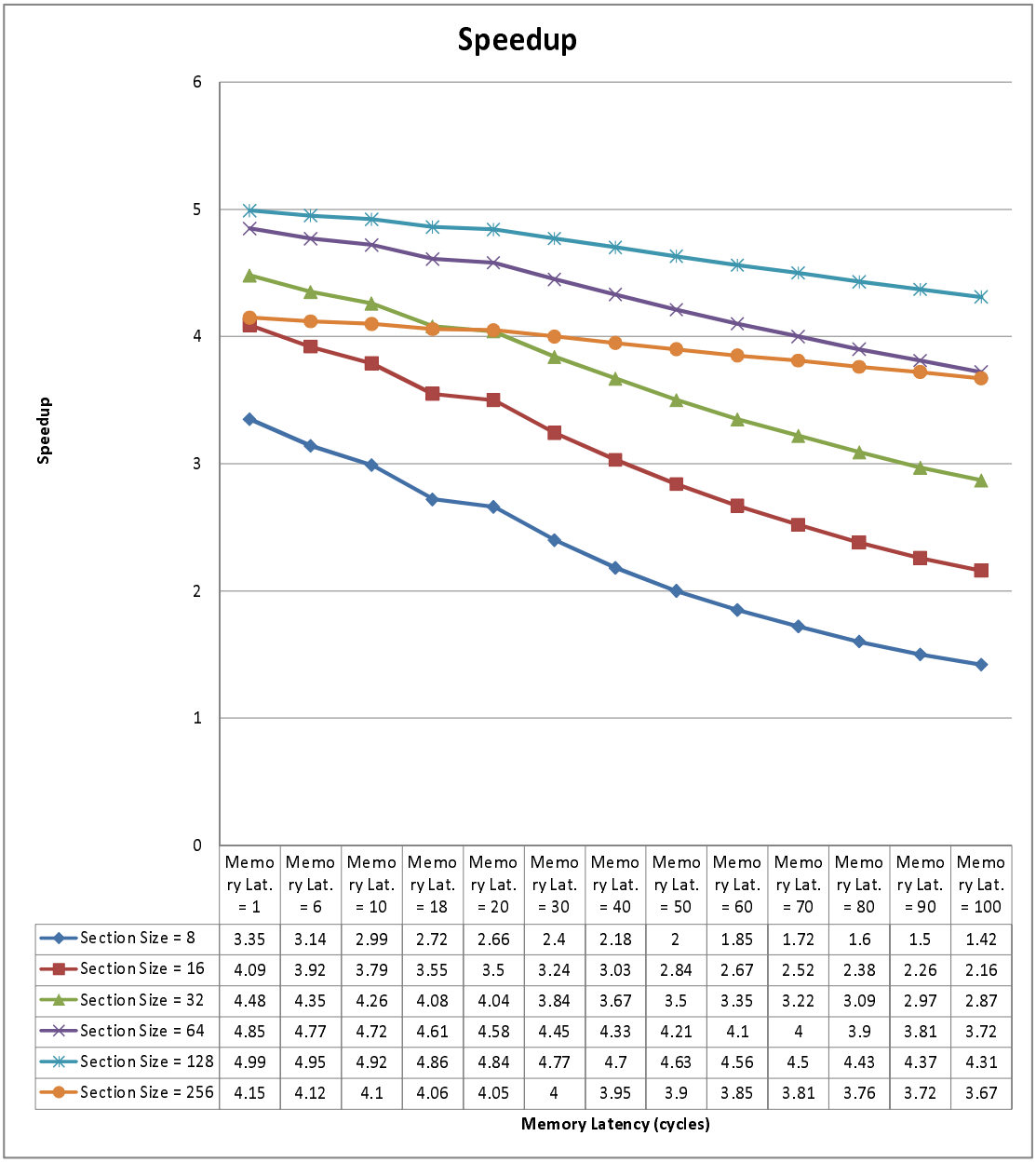


	Memory Lat. = 1	Memory Lat. = 6	Memory Lat. = 10	Memory Lat. = 18	Memory Lat. = 20	Memory Lat. = 30	Memory Lat. = 40	Memory Lat. = 50	Memory Lat. = 60	Memory Lat. = 70	Memory Lat. = 80	Memory Lat. = 90	Memory Lat. = 100
Section Size = 8	9.53	10.15	10.67	11.72	11.98	13.29	14.6	15.91	17.23	18.54	19.85	21.17	22.48
Section Size = 16	7.79	8.14	8.42	8.98	9.12	9.83	10.54	11.24	11.95	12.66	13.36	14.07	14.78
Section Size = 32	7.13	7.32	7.49	7.81	7.89	8.29	8.7	9.1	9.51	9.91	10.31	10.72	11.12
Section Size = 64	6.58	6.68	6.76	6.92	6.96	7.16	7.37	7.57	7.77	7.97	8.17	8.38	8.58
Section Size = 128	6.4	6.45	6.49	6.57	6.59	6.69	6.79	6.89	6.99	7.09	7.19	7.3	7.4
Section Size = 256	7.68	7.73	7.77	7.85	7.87	7.98	8.08	8.18	8.28	8.38	8.48	8.58	8.68
Scalar	31.94	31.94	31.94	31.94	31.94	31.94	31.94	31.94	31.94	31.94	31.94	31.94	31.94

Figure 5.7: Floyd execution time when varying the memory latency

5.2.4 Linpack

Using 6 vector instructions (Table 5.11) Linpack executes 4.33 times faster(Figures 5.15 and 5.17) for a Section Size of 32 and Memory Latency of 1 cycle. The profile showed that the daxpy() function accounted for 75% of the execution time (Section 4.4.1). This 4.33X speedup obtained is more than the theoretical maximum speedup, due to Ahmdal's law. However, the difference is small enough to be in the noise margin of SimpleScalar and in the error margin of the profiler used.

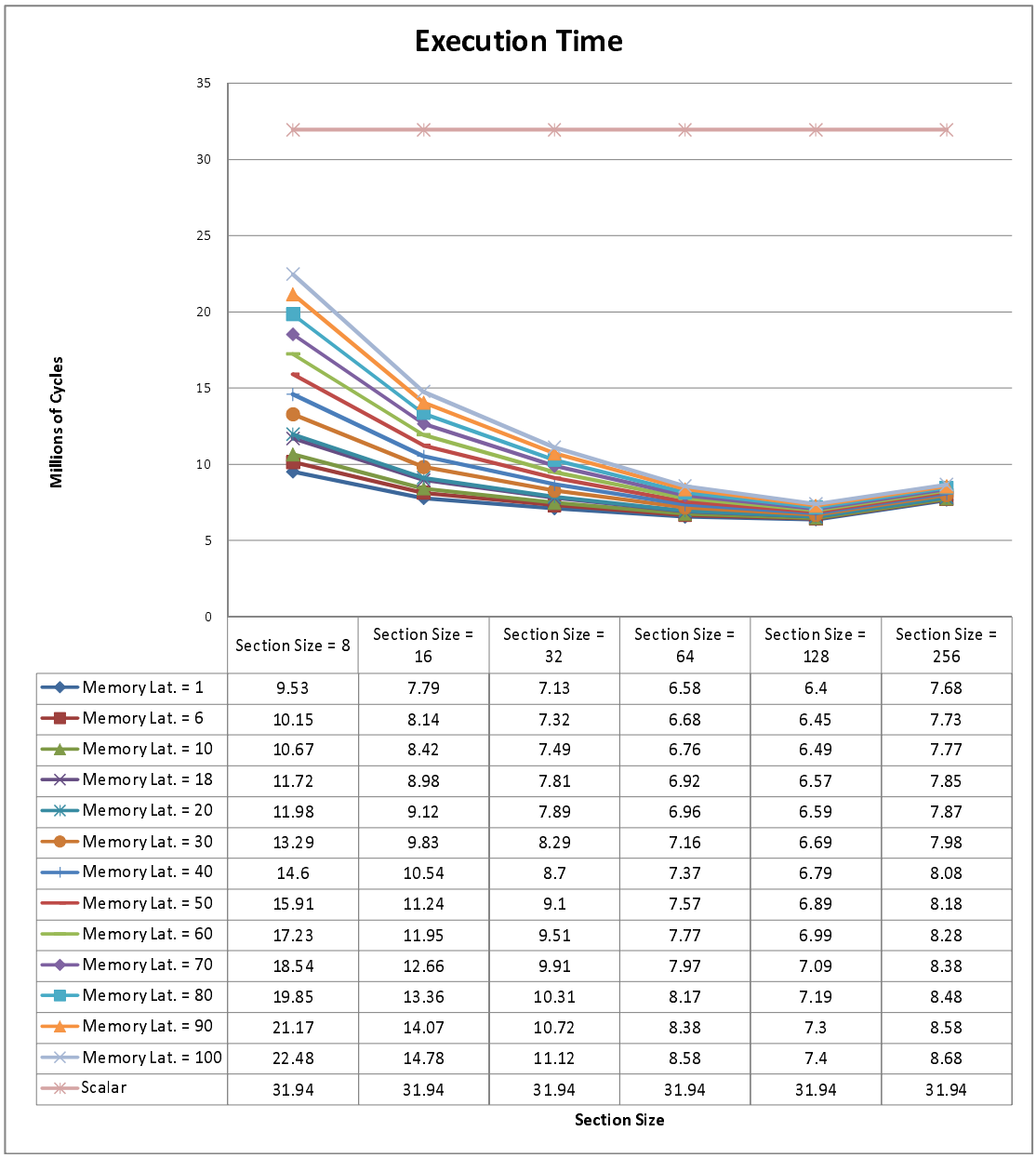


	Memory Lat. = 1	Memory Lat. = 6	Memory Lat. = 10	Memory Lat. = 18	Memory Lat. = 20	Memory Lat. = 30	Memory Lat. = 40	Memory Lat. = 50	Memory Lat. = 60	Memory Lat. = 70	Memory Lat. = 80	Memory Lat. = 90	Memory Lat. = 100
Section Size = 8	3.35	3.14	2.99	2.72	2.66	2.4	2.18	2	1.85	1.72	1.6	1.5	1.42
Section Size = 16	4.09	3.92	3.79	3.55	3.5	3.24	3.03	2.84	2.67	2.52	2.38	2.26	2.16
Section Size = 32	4.48	4.35	4.26	4.08	4.04	3.84	3.67	3.5	3.35	3.22	3.09	2.97	2.87
Section Size = 64	4.85	4.77	4.72	4.61	4.58	4.45	4.33	4.21	4.1	4	3.9	3.81	3.72
Section Size = 128	4.99	4.95	4.92	4.86	4.84	4.77	4.7	4.63	4.56	4.5	4.43	4.37	4.31
Section Size = 256	4.15	4.12	4.1	4.06	4.05	4	3.95	3.9	3.85	3.81	3.76	3.72	3.67

Figure 5.8: Floyd speedup when varying the memory latency

The vector lengths in this version of Linpack varied from 99 to 1 for each pass. This is the reason for having the best performance for small Section Sizes when the Memory latency is low (best performance for Memory Latency = 1 cycle is for Section Size = 32) and when the Memory Latency increases, a Section Size of 128 is the fastest (Figures 5.15 and 5.17).

For a memory latency of 18 cycles, the smallest execution time is achieved by the configuration featuring a Section Size of 64 (Figure 5.15), and the difference in execution

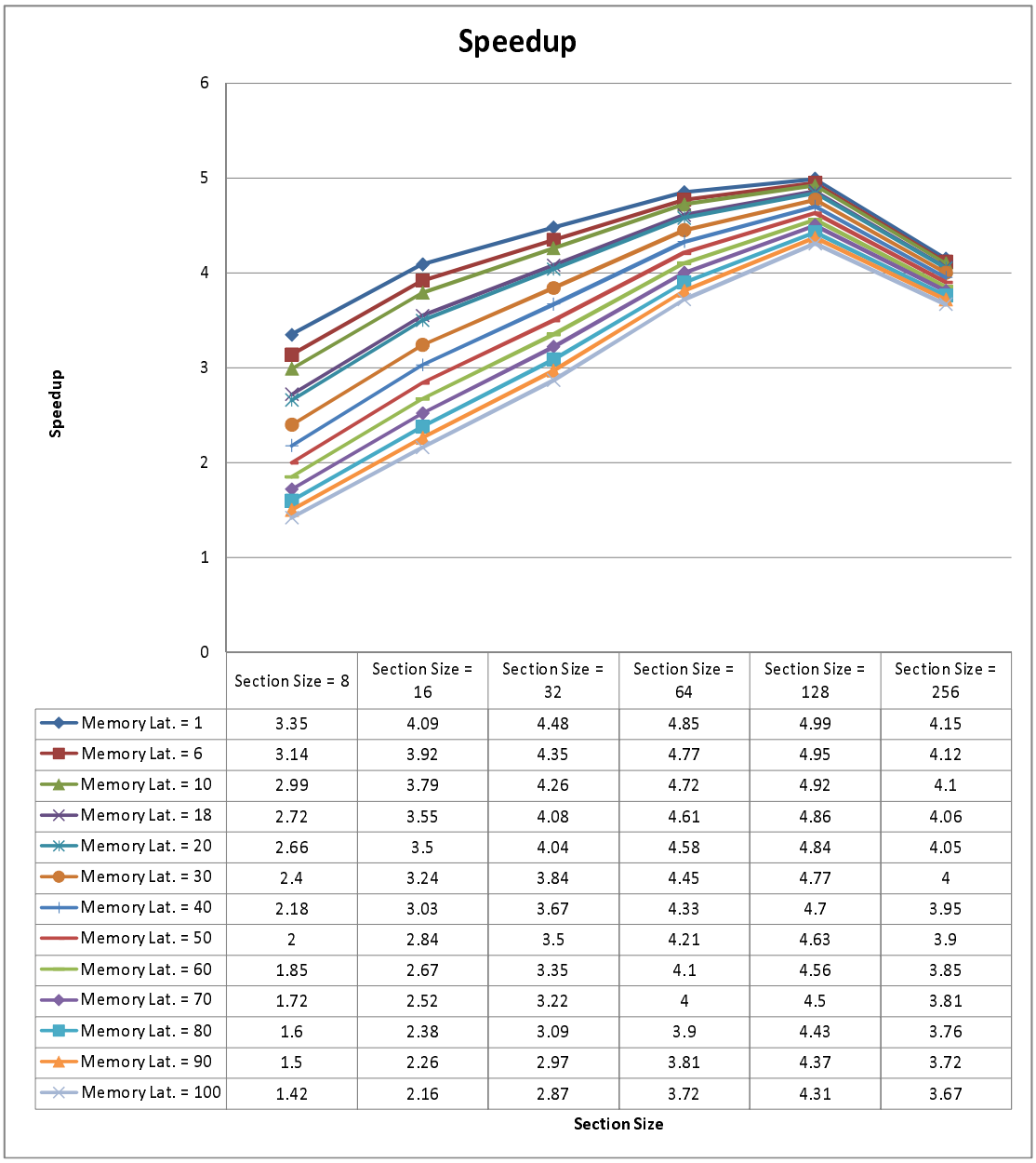


	Section Size = 8	Section Size = 16	Section Size = 32	Section Size = 64	Section Size = 128	Section Size = 256
Memory Lat. = 1	9.53	7.79	7.13	6.58	6.4	7.68
Memory Lat. = 6	10.15	8.14	7.32	6.68	6.45	7.73
Memory Lat. = 10	10.67	8.42	7.49	6.76	6.49	7.77
Memory Lat. = 18	11.72	8.98	7.81	6.92	6.57	7.85
Memory Lat. = 20	11.98	9.12	7.89	6.96	6.59	7.87
Memory Lat. = 30	13.29	9.83	8.29	7.16	6.69	7.98
Memory Lat. = 40	14.6	10.54	8.7	7.37	6.79	8.08
Memory Lat. = 50	15.91	11.24	9.1	7.57	6.89	8.18
Memory Lat. = 60	17.23	11.95	9.51	7.77	6.99	8.28
Memory Lat. = 70	18.54	12.66	9.91	7.97	7.09	8.38
Memory Lat. = 80	19.85	13.36	10.31	8.17	7.19	8.48
Memory Lat. = 90	21.17	14.07	10.72	8.38	7.3	8.58
Memory Lat. = 100	22.48	14.78	11.12	8.58	7.4	8.68
Scalar	31.94	31.94	31.94	31.94	31.94	31.94

Figure 5.9: Floyd execution time when varying the section size

time comparing Section Size of 64 with the Section Size of 8 is 46% and 30.88% compared to a Section Size of 256 elements. If the micro-architecture is able to hide the overhead of having the section size larger than the vector length, the differences in execution time would have been very different. What these numbers suggest is that if you have a machine designed to process very large arrays, with a large section size, but feed it with small input data, the performance will drop significantly.

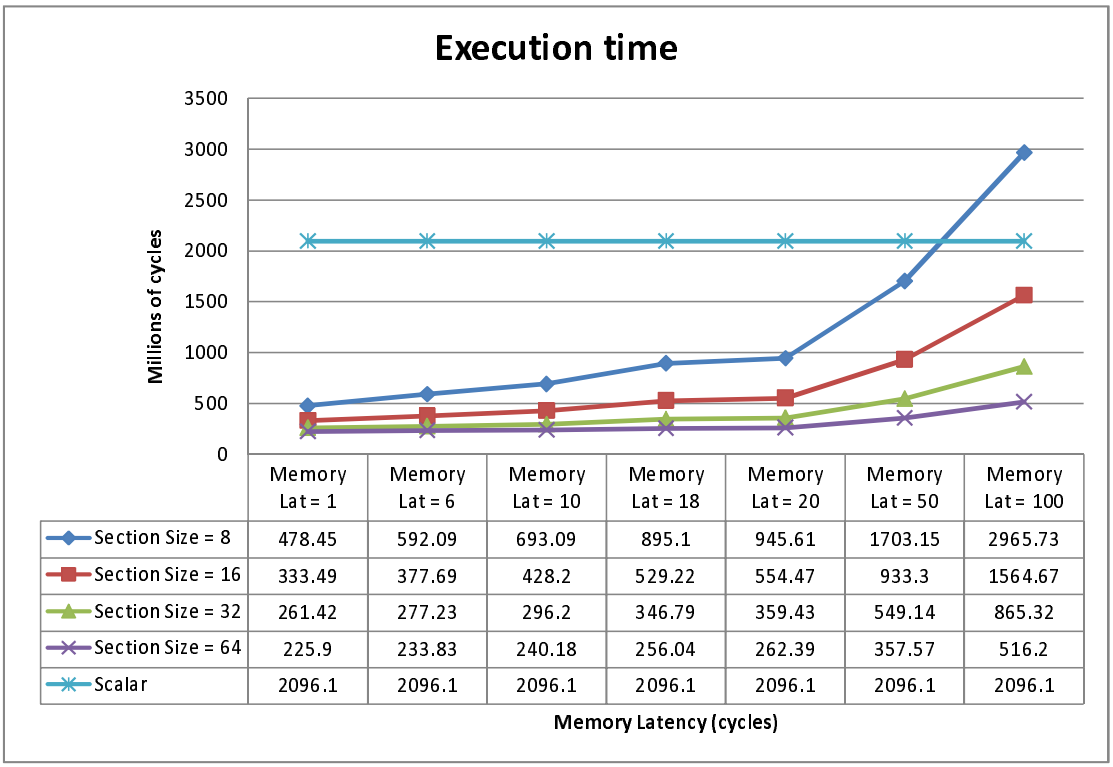
Because the daxpy() loop is very simple(two loads, one multiply-add and one store),



	Section Size = 8	Section Size = 16	Section Size = 32	Section Size = 64	Section Size = 128	Section Size = 256
Memory Lat. = 1	3.35	4.09	4.48	4.85	4.99	4.15
Memory Lat. = 6	3.14	3.92	4.35	4.77	4.95	4.12
Memory Lat. = 10	2.99	3.79	4.26	4.72	4.92	4.1
Memory Lat. = 18	2.72	3.55	4.08	4.61	4.86	4.06
Memory Lat. = 20	2.66	3.5	4.04	4.58	4.84	4.05
Memory Lat. = 30	2.4	3.24	3.84	4.45	4.77	4
Memory Lat. = 40	2.18	3.03	3.67	4.33	4.7	3.95
Memory Lat. = 50	2	2.84	3.5	4.21	4.63	3.9
Memory Lat. = 60	1.85	2.67	3.35	4.1	4.56	3.85
Memory Lat. = 70	1.72	2.52	3.22	4	4.5	3.81
Memory Lat. = 80	1.6	2.38	3.09	3.9	4.43	3.76
Memory Lat. = 90	1.5	2.26	2.97	3.81	4.37	3.72
Memory Lat. = 100	1.42	2.16	2.87	3.72	4.31	3.67

Figure 5.10: Floyd speedup when varying the section size

increasing the memory latency significantly slows down the execution. For the Section Size of 8, the speedup drops from 3.74X (1 cycle memory latency) to 1.07X (100 cycles memory latency). When comparing the times for Section Size 128, the speedup drops from 4.02X to 3.07X (Figure 5.16). If the Memory Latency is fixed to 18 cycles, the fastest configuration is the one with the Section Size of 64, with a speedup of 3.88X, not far from the 4.33X speedup for a Memory Latency of 1 and a Section Size of 8 elements.



	Memory Lat = 1	Memory Lat = 6	Memory Lat = 10	Memory Lat = 18	Memory Lat = 20	Memory Lat = 50	Memory Lat = 100
Section Size = 8	478.45	592.09	693.09	895.1	945.61	1703.15	2965.73
Section Size = 16	333.49	377.69	428.2	529.22	554.47	933.3	1564.67
Section Size = 32	261.42	277.23	296.2	346.79	359.43	549.14	865.32
Section Size = 64	225.9	233.83	240.18	256.04	262.39	357.57	516.2
Scalar	2096.1	2096.1	2096.1	2096.1	2096.1	2096.1	2096.1

Figure 5.11: Bellman-Ford execution time when varying the memory latency

Table 5.11: Custom vector ISA used for Linpack

Nr.	Instruction	Synopsis	Uses masks	OBS
1	v.ld.d VRDS, RT	VRDS[i] ← RT[i]	yes	
2	v.msadd.d VRDD, FS, VRDT	VRDD[i] += FS * VRDT[i]	no	
3	v.setvconf RD, RS, RT	vl ← RD, vindex ← RS, vmask ← RT	no	
4	v.st.d RS, VRDT	RS[i] ← VRDT[i]	yes	
5	v.updateindex RS	vindex ← RS ← vindex + section_size	no	
6	v.updatevl RS	vl ← RS ← MAX(0, vl - section_size)	no	

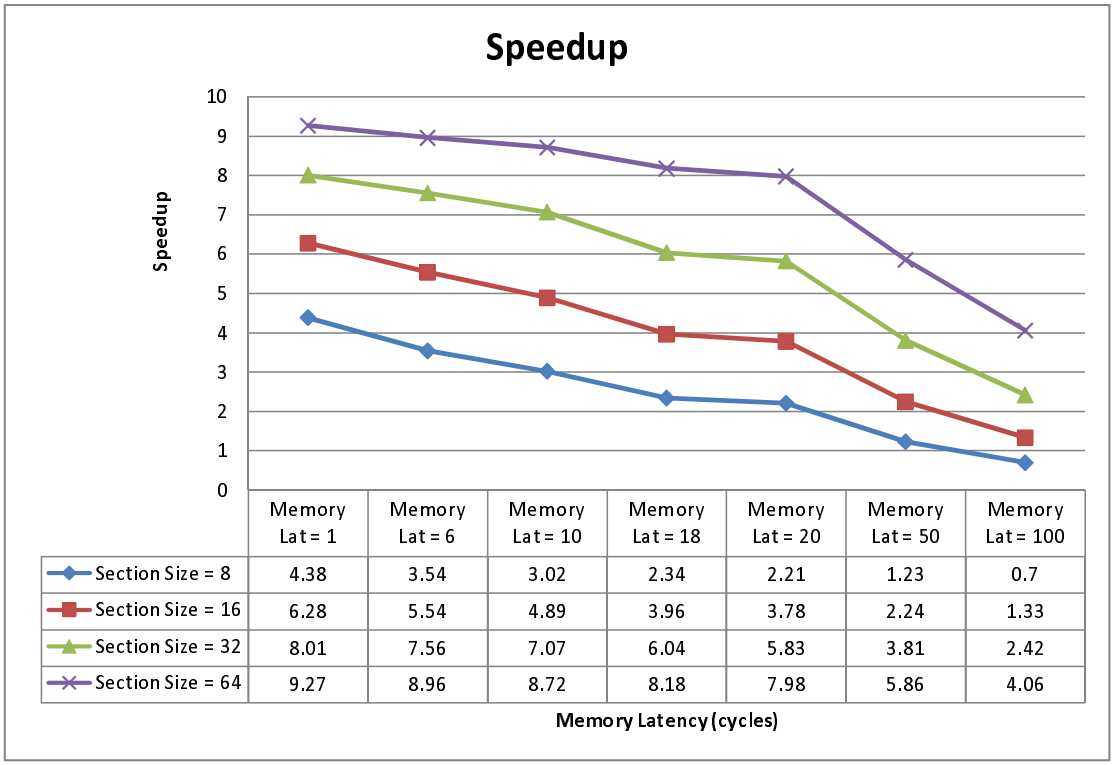


Figure 5.12: Bellman-Ford speedup when varying the memory latency

5.3 Summary of the results

We experimented with the SimpleScalar 3.0d toolset, which was modified in order to support a total of 24 vector instructions (13 for Dijkstra, 11 for Floyd, 18 Bellman-Ford and 6 for Linpack).

The application level speedups can be summarized as follows:

- A peak speedup of 24.88X for Dijkstra compared to the original version implemented with linked lists and pointers, and a speedup of 4.31X compared to the rewritten (but still scalar) version
- A maximum speedup of 4.99X for Floyd
- Speedups of up to 9.27X for Bellman-Ford
- After vectorization, Linpack was up to 4.33X times faster

General observations about the factors that influence performance of the vectorized code:

- Increased Memory Latency drastically reduces the performance of small Section Sizes

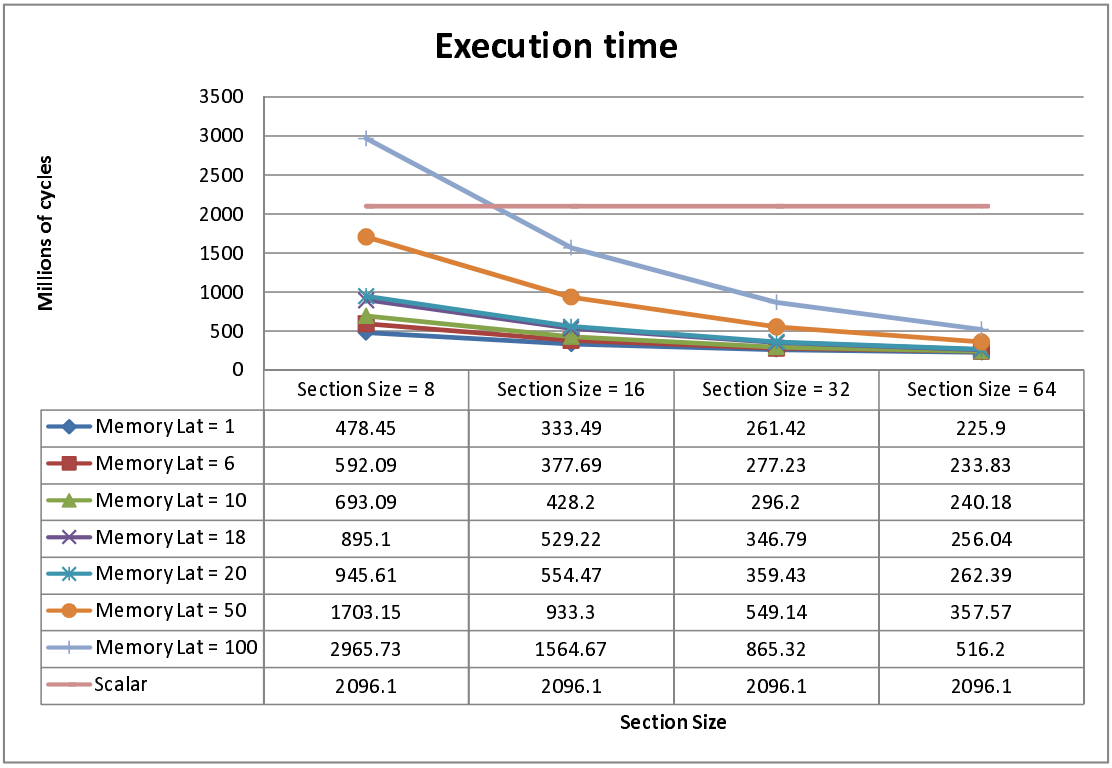
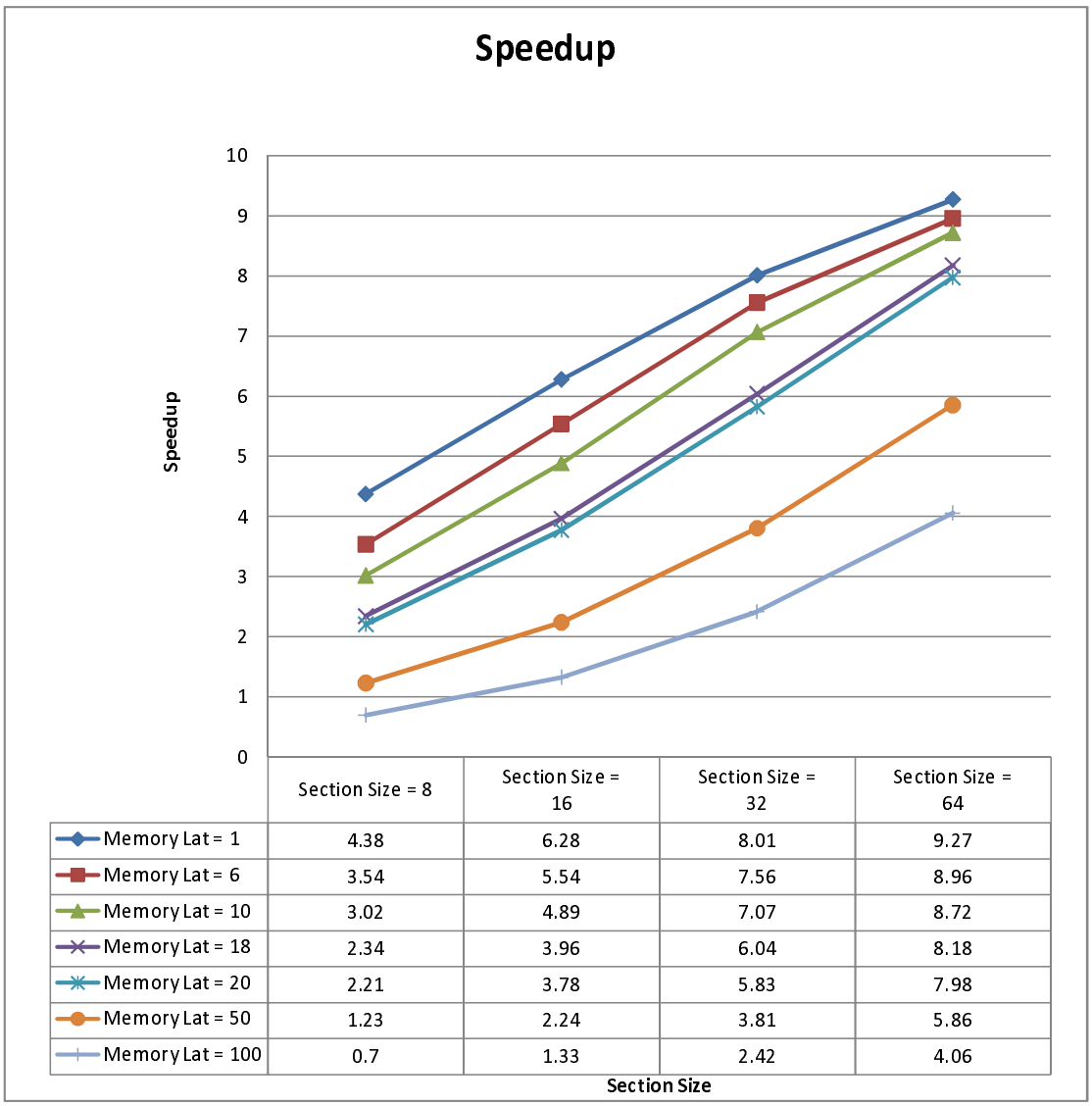


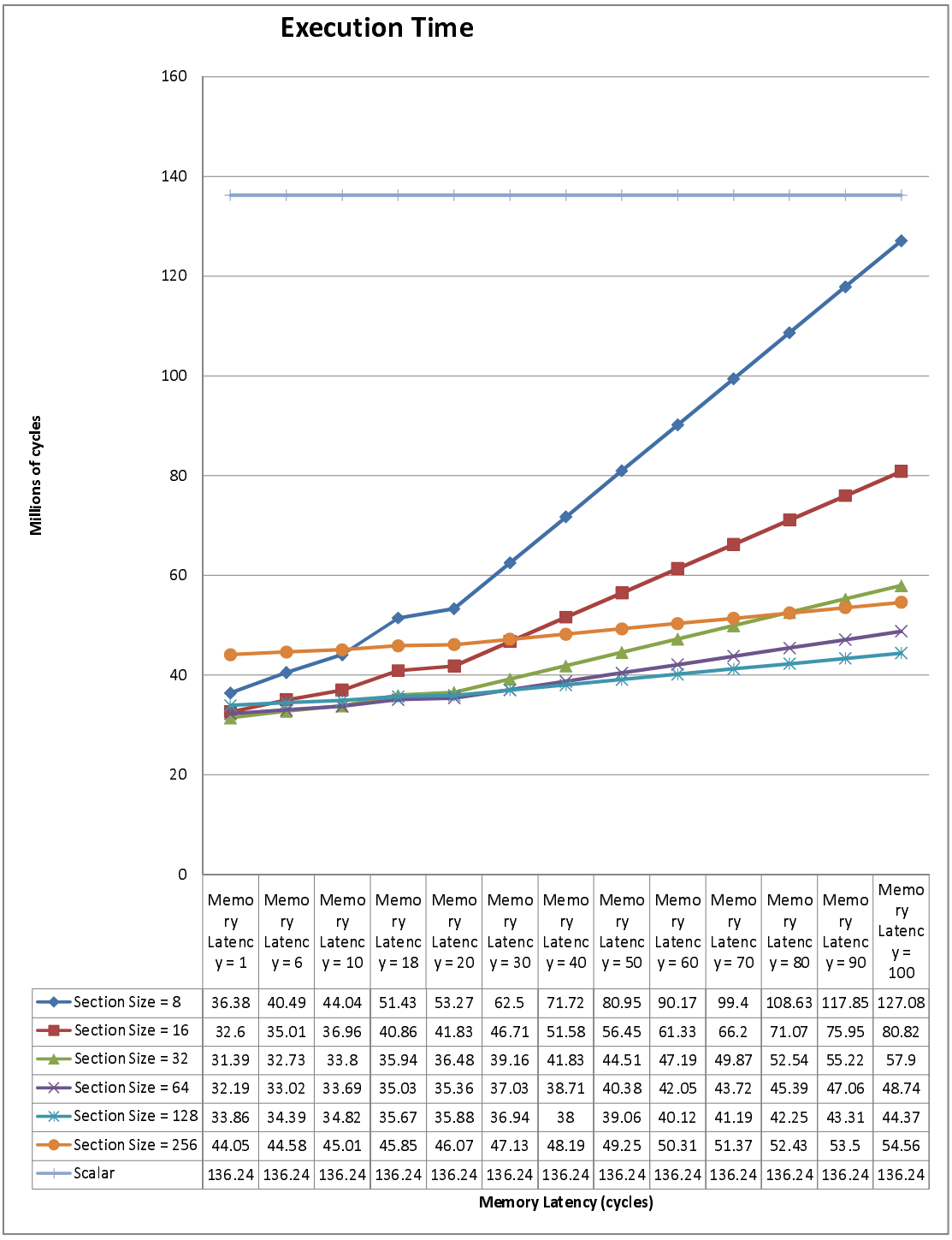
Figure 5.13: Bellman-Ford execution time when varying the section size

- Having a vector length as close to the Section Size as possible is desirable. If the Section Size is smaller than the vector length, the array will have to be split into multiple sections, and better performance can be obtained by having a larger Section Size. If the Section Size is larger than the vector length, cycles will be wasted and performance will be again lower.
- Large Section Sizes compensate for the high startup costs of large Memory Latencies
- The performance doesn't drop significantly for a Memory Latency of 18 cycles compared to the fastest latency of 1 cycle. So an eventual implementation without caches for the vector unit can deliver good performance, as long as enough vector registers are present to hold the intermediate results.



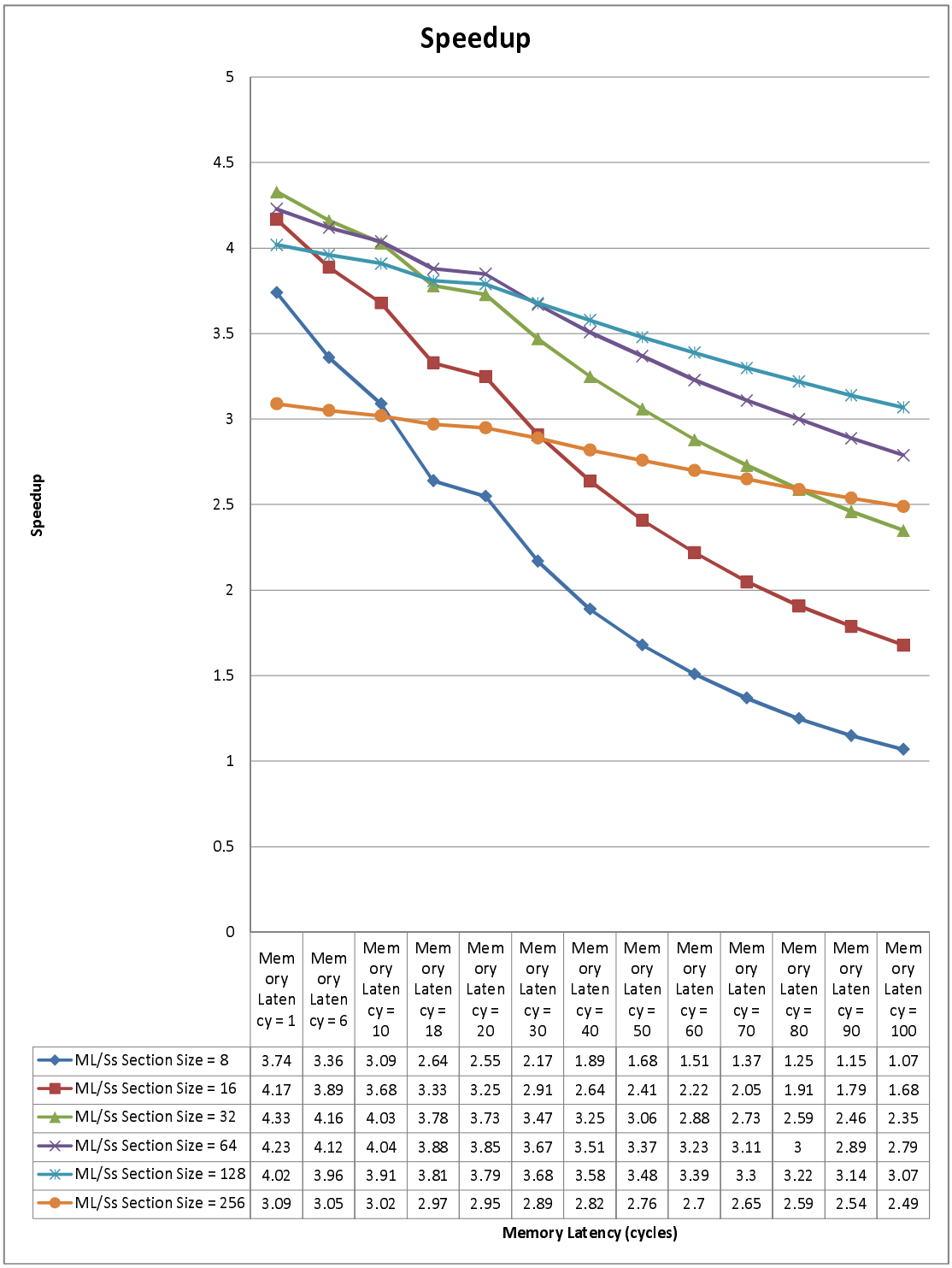
	Section Size = 8	Section Size = 16	Section Size = 32	Section Size = 64
Memory Lat = 1	4.38	6.28	8.01	9.27
Memory Lat = 6	3.54	5.54	7.56	8.96
Memory Lat = 10	3.02	4.89	7.07	8.72
Memory Lat = 18	2.34	3.96	6.04	8.18
Memory Lat = 20	2.21	3.78	5.83	7.98
Memory Lat = 50	1.23	2.24	3.81	5.86
Memory Lat = 100	0.7	1.33	2.42	4.06

Figure 5.14: Bellman-Ford speedup when varying the section size



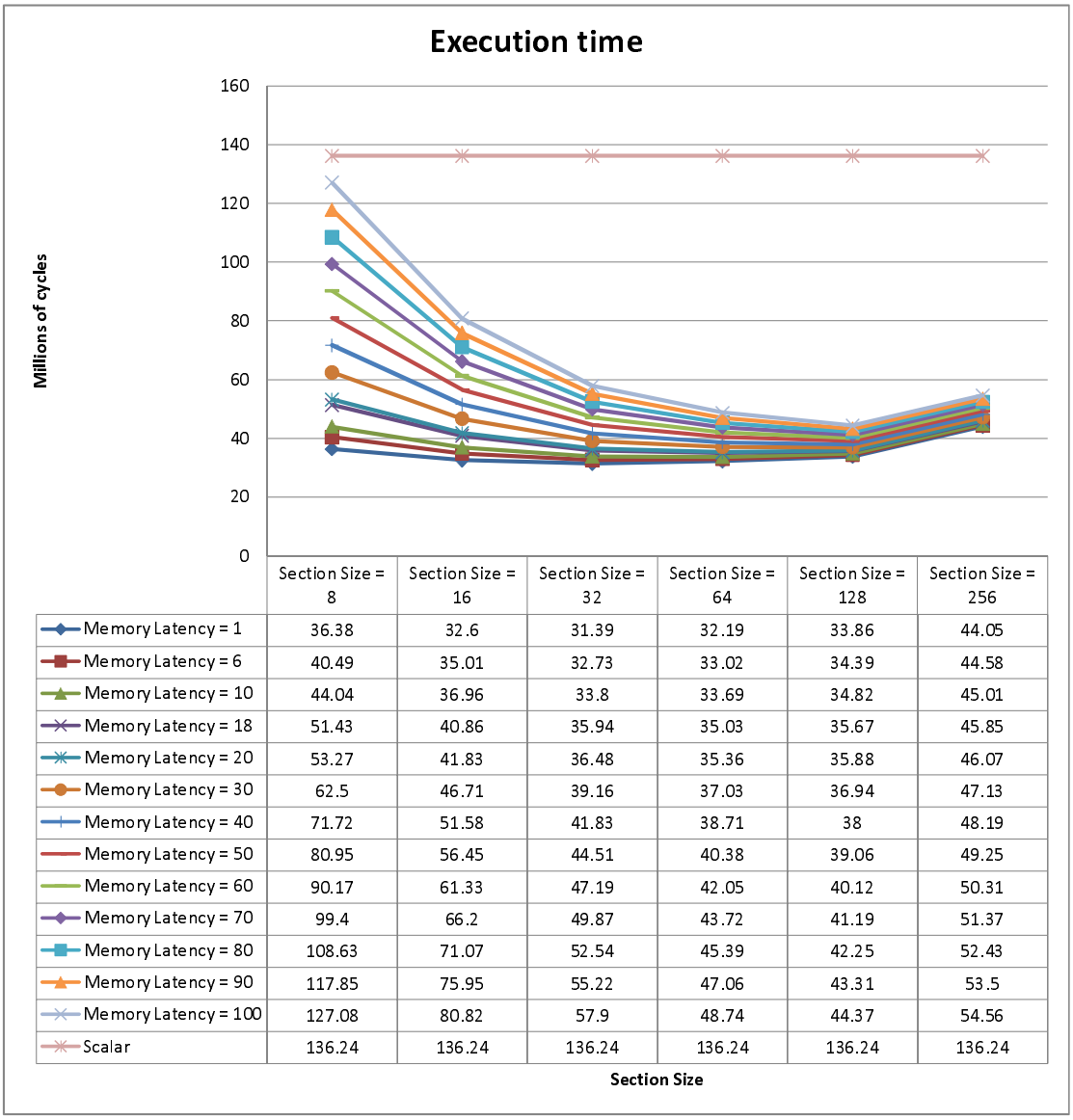
	Memory Latency = 1	Memory Latency = 6	Memory Latency = 10	Memory Latency = 18	Memory Latency = 20	Memory Latency = 30	Memory Latency = 40	Memory Latency = 50	Memory Latency = 60	Memory Latency = 70	Memory Latency = 80	Memory Latency = 90	Memory Latency = 100
Section Size = 8	36.38	40.49	44.04	51.43	53.27	62.5	71.72	80.95	90.17	99.4	108.63	117.85	127.08
Section Size = 16	32.6	35.01	36.96	40.86	41.83	46.71	51.58	56.45	61.33	66.2	71.07	75.95	80.82
Section Size = 32	31.39	32.73	33.8	35.94	36.48	39.16	41.83	44.51	47.19	49.87	52.54	55.22	57.9
Section Size = 64	32.19	33.02	33.69	35.03	35.36	37.03	38.71	40.38	42.05	43.72	45.39	47.06	48.74
Section Size = 128	33.86	34.39	34.82	35.67	35.88	36.94	38	39.06	40.12	41.19	42.25	43.31	44.37
Section Size = 256	44.05	44.58	45.01	45.85	46.07	47.13	48.19	49.25	50.31	51.37	52.43	53.5	54.56
Scalar	136.24	136.24	136.24	136.24	136.24	136.24	136.24	136.24	136.24	136.24	136.24	136.24	136.24

Figure 5.15: Linpack execution time when varying the memory latency



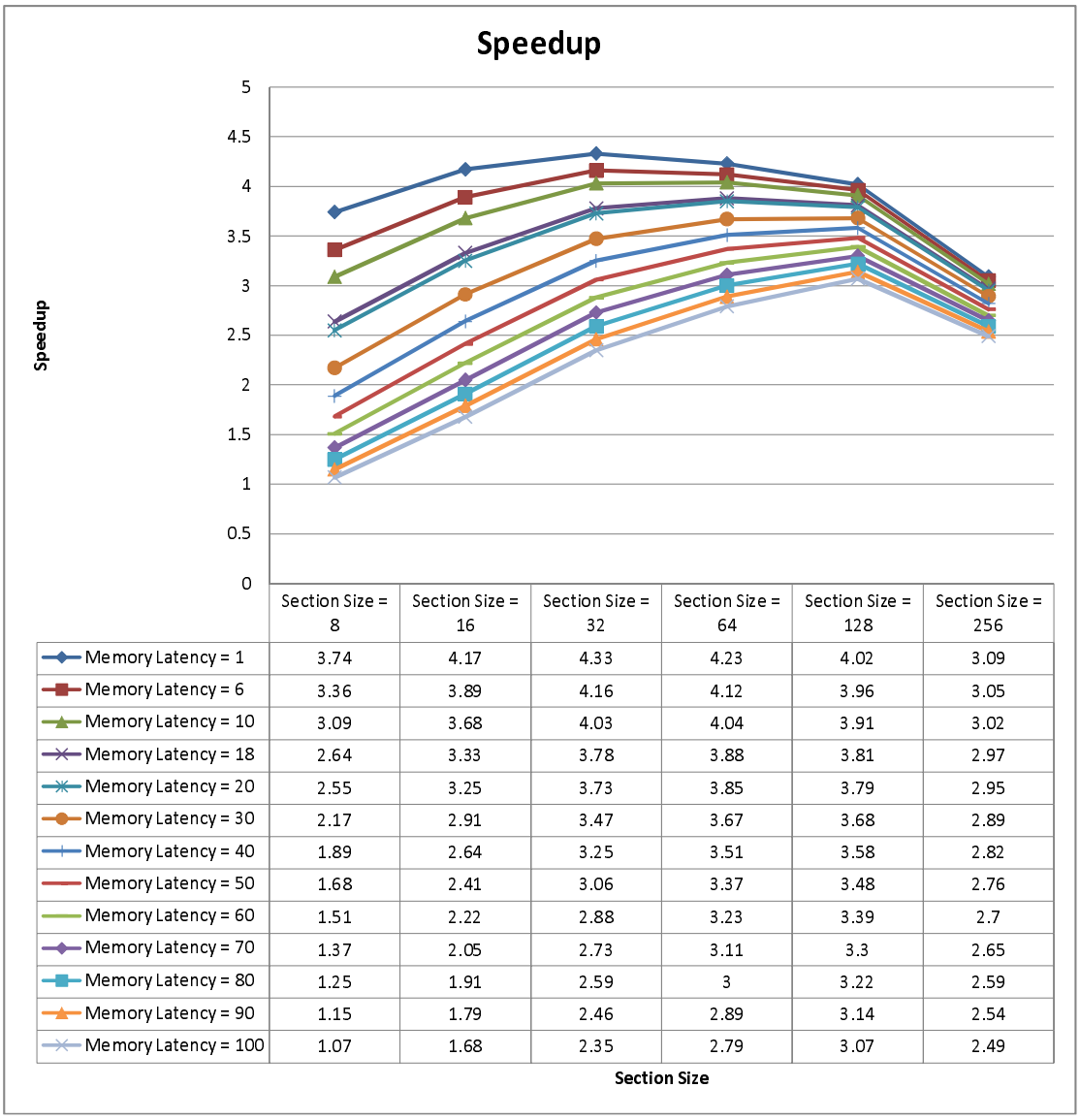
	Memory Latency = 1	Memory Latency = 6	Memory Latency = 10	Memory Latency = 18	Memory Latency = 20	Memory Latency = 30	Memory Latency = 40	Memory Latency = 50	Memory Latency = 60	Memory Latency = 70	Memory Latency = 80	Memory Latency = 90	Memory Latency = 100
ML/Ss Section Size = 8	3.74	3.36	3.09	2.64	2.55	2.17	1.89	1.68	1.51	1.37	1.25	1.15	1.07
ML/Ss Section Size = 16	4.17	3.89	3.68	3.33	3.25	2.91	2.64	2.41	2.22	2.05	1.91	1.79	1.68
ML/Ss Section Size = 32	4.33	4.16	4.03	3.78	3.73	3.47	3.25	3.06	2.88	2.73	2.59	2.46	2.35
ML/Ss Section Size = 64	4.23	4.12	4.04	3.88	3.85	3.67	3.51	3.37	3.23	3.11	3	2.89	2.79
ML/Ss Section Size = 128	4.02	3.96	3.91	3.81	3.79	3.68	3.58	3.48	3.39	3.3	3.22	3.14	3.07
ML/Ss Section Size = 256	3.09	3.05	3.02	2.97	2.95	2.89	2.82	2.76	2.7	2.65	2.59	2.54	2.49

Figure 5.16: Linpack speedup when varying the memory latency



	Section Size = 8	Section Size = 16	Section Size = 32	Section Size = 64	Section Size = 128	Section Size = 256
Memory Latency = 1	36.38	32.6	31.39	32.19	33.86	44.05
Memory Latency = 6	40.49	35.01	32.73	33.02	34.39	44.58
Memory Latency = 10	44.04	36.96	33.8	33.69	34.82	45.01
Memory Latency = 18	51.43	40.86	35.94	35.03	35.67	45.85
Memory Latency = 20	53.27	41.83	36.48	35.36	35.88	46.07
Memory Latency = 30	62.5	46.71	39.16	37.03	36.94	47.13
Memory Latency = 40	71.72	51.58	41.83	38.71	38	48.19
Memory Latency = 50	80.95	56.45	44.51	40.38	39.06	49.25
Memory Latency = 60	90.17	61.33	47.19	42.05	40.12	50.31
Memory Latency = 70	99.4	66.2	49.87	43.72	41.19	51.37
Memory Latency = 80	108.63	71.07	52.54	45.39	42.25	52.43
Memory Latency = 90	117.85	75.95	55.22	47.06	43.31	53.5
Memory Latency = 100	127.08	80.82	57.9	48.74	44.37	54.56
Scalar	136.24	136.24	136.24	136.24	136.24	136.24

Figure 5.17: Linpack execution time when varying the section size



	Section Size = 8	Section Size = 16	Section Size = 32	Section Size = 64	Section Size = 128	Section Size = 256
Memory Latency = 1	3.74	4.17	4.33	4.23	4.02	3.09
Memory Latency = 6	3.36	3.89	4.16	4.12	3.96	3.05
Memory Latency = 10	3.09	3.68	4.03	4.04	3.91	3.02
Memory Latency = 18	2.64	3.33	3.78	3.88	3.81	2.97
Memory Latency = 20	2.55	3.25	3.73	3.85	3.79	2.95
Memory Latency = 30	2.17	2.91	3.47	3.67	3.68	2.89
Memory Latency = 40	1.89	2.64	3.25	3.51	3.58	2.82
Memory Latency = 50	1.68	2.41	3.06	3.37	3.48	2.76
Memory Latency = 60	1.51	2.22	2.88	3.23	3.39	2.7
Memory Latency = 70	1.37	2.05	2.73	3.11	3.3	2.65
Memory Latency = 80	1.25	1.91	2.59	3	3.22	2.59
Memory Latency = 90	1.15	1.79	2.46	2.89	3.14	2.54
Memory Latency = 100	1.07	1.68	2.35	2.79	3.07	2.49

Figure 5.18: Linpack speedup when varying the section size

6 Conclusions

The main objective of this thesis was to develop a methodology for synthesizing a customized vector ISA. We have met that goal by addressing the additional objectives we stated, namely: (1) profile and vectorize a number of applications from the telecommunications and linear algebra domain, (2) synthesize custom vector ISAs for the selected applications, and (3) evaluate the performance of the custom vector ISAs over a traditional scalar instruction set.

Three algorithms that compute the shortest paths in a directed graph (Dijkstra, Floyd, and Bellman-Ford) and the Linpack benchmark have been analyzed. After profiling the selected applications, the critical kernels have been identified. In order to vectorize each kernel, a custom Vector ISA has been synthesized for each application, containing 21 general purpose and 3 application specific instructions.

In order to evaluate the performance of the custom vector ISAs compared to a scalar ISA, we created a framework which facilitated the simulation of a custom vector unit, using a modified version of the SimpleScalar3 toolset, completing objective (3). The vector register file modeled is composed of separate integer and floating point registers, each one being able to store a number of registers equal to the section size. For each vector register, an implicit bit vector register was used to enable masked execution. A special configuration register enables or disables masked execution mode, thus being able to use the same opcodes for both operating modes. When writing code for this vector unit, the programmer doesn't need to know the section size of the machine running the code. Special instructions have been introduced to set and update the Vector Length. Auto sectioning improves portability of the vector code on different hardware implementation featuring different section sizes.

In order to simplify the process of vectorizing the critical kernels, the use of inline assembly has been chosen. The second approach: first compiling the C program in order to obtain an assembly file and then replacing the scalar assembly code with the custom vector instructions was considered as less structured. The chosen method was less error prone and helped speeding up the debugging process, as direct access to all the program variables was available. The disadvantage of our approach consists of the implicit overhead of having the compiler do all the data copying between the C variables and the registers used in the vector instructions.

Simulations were used to compare the performance of the custom vector instruction sets against a scalar ISA. In respect to simulation speed, using SimpleScalar was faster compared to a very detailed simulator written in VHDL or Verilog. The drawbacks are limited flexibility and slight variation in the simulation results between successive test runs. However, the results obtained clearly lead to the conclusion that performance can be improved when using the custom Vector ISAs, with speedups well over the accuracy margin of the simulator. When benchmarking, two critical parameters have been taken

into account: the section size of the vector register file and the memory latency of the vector memory unit.

Overall applications speedups obtained after applying the custom Vector ISAs can be summarized as follows: for Dijkstra, 24.88X (after both optimization and vectorization), 4.99X for Floyd, 9.27X for Bellman Ford and 4.33X for Linpack. The peak speedups have been obtained for a simulated section size close to the average vector length of the processed arrays and by simulating a perfect data cache. Further tests showed that the performance degradation of having the vector memory latency set to 18 cycles is low enough to consider connecting the Vector Unit directly to the main memory using a wide data bus. The vector processor is more sensitive to the available bandwidth than to memory latency. The experimental results suggest that following the proposed methodology for customizing the Vector ISA delivers substantial performance benefits in the targeted application domains. The performance data obtained provided the means to evaluate the performance of the custom instructions sets we built. The framework created is flexible and can be used for future analysis of a wider range of applications.

Based on the experiments completed, the following hardware design choices have a significant impact on the performance of the Vector Processor:

Section size. A tradeoff exists between processor area and the section size. The best results are obtained for a section size equal or slightly larger than the average vector length.

Vector memory latency. Increased memory latency severely degrades performance for small section sizes (up to 32 elements). Experiments show that configurations with large section sizes are less sensitive to increased memory latency.

Vector memory bandwidth. A crucial factor in obtaining good performance with a vector processor is memory bandwidth. If the kernels become bandwidth limited, the advantage of having fast functional units is not fully exploited.

Datapath organization. The use of multiple vector lanes decreases the execution time of the arithmetic and logic operations. The startup cost of the operations remains constant, but the execution time decreases linearly with the number of vector lanes used.

Application specific performance is affected also by the software implementation and the input data sets used. The following performance critical factors have been identified:

The average vector length. In order to efficiently utilize the vector hardware, applications should process long vectors. This is not always possible. In the benchmarks we used, the average vector length was of 100 elements for the shortest paths in a graph algorithms and 50 for Linpack. As expected, top performance was obtained for section sizes close to the average vector length.

Vectorization degree. Scalar code is part of all the applications. The maximum theoretical speedup is limited by Ahmdal's law.

Vectorizing compiler support. Not even the most advanced computer architecture can compensate for the lack of good compilers. Vectorizing compilers always had problems with optimizing data structures containing pointers, but this might improve in the future. Manual vectorization of kernels is not feasible when large applications need to be optimized with a specific architecture in mind, or when the application targeted doesn't have a small number of critical kernels.

All the steps completed in this work contributed to creating a methodology for syn-

thesizing a high-performance custom Vector ISA given a set of applications. The results we obtained indicate that the methodology meets the requirements stated in the introduction of the thesis.

Future research direction. Several problems can be addressed in future work: estimating the performance cost of using inline assembly, analyzing the efficiency of customized vector instruction sets for a wider range of application domains, and improving the vector architecture we used. Improvements of the Vector Register File can have a strong impact on performance. The Vector Memory Unit is suspected to be the major bottleneck of the micro-architecture we used. A more thorough analysis of how to improve the memory performance is needed.

Bibliography

[1] *Mibench* http://www.eecs.umich.edu/mibench/.

[2] Todd Austin, *Simplescalar* http://www.cs.wisc.edu/~mscalar/simplescalar.html.

[3] ———, *Simplescalar* http://www.simplescalar.com/.

[4] W. Buchholz, *The ibm system/370 vector architecture*, IBM Systems Journal (1986), 51.

[5] D. Cheresiz, B.H.H. Juurlink, and S. Vassiliadis, *Performance benefits of special-purpose instructions in the csi architecture*, Proceedings ProRISC 2002, 2002.

[6] D. Cheresiz, B.H.H. Juurlink, S. Vassiliadis, and H. A. G. Wijshoff, *Architectural support for 3d graphics in the complex streamed instruction set*, International Journal of Parallel and Distributed Systems and Networks (2002), 185–193.

[7] ———, *Performance scalability of multimedia instruction set extensions*, Proc. Euro-Par 2002 Parallel processing, 2002.

[8] Sangyeun Cho, Pen-Chung Yew, and Gyongho Lee, *A high-bandwidth memory pipeline for wide issue processors*, IEEE Transactions on Computers (2001), 709 – 723.

[9] Jesus Corbal, Roger Espasa, and Mateo Valero, *Mom: a matrix simd instruction set architecture for multimedia applications*, Proceedings of the ACM/IEEE SC99 Conference, 1999, pp. 1–12.

[10] ———, *Dlp + tlp processors for the next generation of media workloads*, Proceedings of the 7th International Symposium on High-Performance Computer Architecture, 2001, p. 219.

[11] ———, *On the efficiency of reductions in u-simd media extensions*, Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques, 2001, pp. 83–94.

[12] Jesus Corbal, Mateo Valero, and Roger Espasa, *Exploiting a new level of dlp in multimedia applications.*, Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture, 1999, pp. 72 – 79.

[13] Jack Dongarra, *Trends in high-performance computing*, Circuits and Devices Magazine, IEEE (2006).

[14] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet, *The linpack benchmark: Past, present, and future*, http://dx.doi.org/10.1002/cpe.728, July 2002.

[15] Joe Gebis and David Patterson, *Embracing and extending 20th-century instruction set architectures*, Computer, IEEE (2007).

[16] Linley Gwennap, *Digital, mips add multimedia extensions*, MICRODESIGN RESOURCES **10** (1996), no. 15, 1–5.

[17] ———, *Altivec vectorizes powerpc*, MICROPROCESSOR REPORT **12** (1998), no. 6, 1–5.

[18] John Hennessy and David Patterson, *Computer architecture: A quantitative approach*, 3rd edition ed., Morgan Kaufmann, 2002.

[19] Isabelle Hurbain and Georges-André Silber, *An empirical study of some x86 simd integer extensions*.

[20] B.H.H. Juurlink, D. Cheresiz, S. Vassiliadis, and H. A. G. Wijshoff, *Implementation and evaluation of the complex streamed instruction set*, Int. Conf. on Parallel Architectures and Compilation Techniques (PACT) (2001), 73 – 82.

[21] B.H.H. (Ben) Juurlink, Demid Borodin, Roel J. Meeuws, Gerard Th. Aalbers, and Hugo Leisink, *Ssiat* `http://ce.et.tudelft.nl/~demid/SSIAT/`.

[22] Alex Klimovitski, *Using sse and sse2: Misconceptions and reality*, Intel Developer UPDATE Magazine (2001), 1–8.

[23] C. Kozyrakis, *Scalable vector media-processors for embedded systems*, Ph.D. thesis, University of California, Berkeley, 2002.

[24] Hans Meuer, Jack Dongarra, Erich Strohmaier, and Horst Simon, *Top500 supercomputer sites* `http://www.top500.org/`.

[25] Brian Moore, Andris Padegs, Ron Smith, and Werner Buchholz, *Concepts of the system/370 vector architecture*, ISCA '87, 1987, pp. 282 – 288.

[26] Andris Padegs, Brian Moore, Ronald Smith, and Werner Buchholz, *The ibm system/370 vector architecture: Design considerations*, IEEE Transactions on Computers (1988), 509 – 520.

[27] Behrooz Parhami, *Computer arithmetic - algorithms and hardware designs*, 1st edition ed., Oxford University Press, USA, 1999.

[28] Alex Peleg, Sam Wilkie, and Uri Weiser, *Intel mmx for multimedia pcs*, COMMUNICATIONS OF THE ACM **40** (1997), no. 1, 25–38.

[29] R. M. Ramanathan, *Extending the world's most popular processor architecture*, Technology@Intel Magazine (2006).

[30] Shreekant(Ticky) Thakkar and Tom Huff, *Internet streaming simd extensions*, IEEE Computer **32** (1999), no. 12, 26–34.

[31] Marc Tremblay, Michael O'Connor, Venkatesh Narayanan, and Liang He, *Vis speeds new media processing*, IEEE Micro (1996), 10–20.

[32] S. Vassiliadis, B.H.H. Juurlink, and E. A. Hakkennes, *Complex streamed instructions: introduction and initial evaluation*, Proc. 26th Euromicro Conference, 2000.

[33] Niklas Zennstrom and Janus Friis, *Joost* `http://www.joost.com/`.

SimpleScalar

A

A.1 A short introduction to SimpleScalar

When trying to test different ideas and obtain results in a reasonable time frame, a CPU simulator such as the SimpleScalar [52, 53] can be very helpful. A tradeoff between the accuracy of the model, the speed of the simulation and the flexibility always exists. If a simulator is cycle accurate it can provide a very exact model of the target architecture, but long simulation times and usually low flexibility are things that are not always acceptable.

SimpleScalar was written in 1992. It contains several models ranging from simple but very fast simulators to detailed models with features such as dynamic scheduling and a multilevel memory system. When comparing the simulation speed of the fastest and slowest SimpleScalar versions, the difference is of an order of magnitude.

The technique used to run applications is called execution driven simulation. This has several advantages. It provides access to the data produced and consumed during the execution of the program. This is very important for the estimation of dynamic power consumptions, defined by transistor switching activity. It also provides a convenient way of simulating speculative execution models. Drawbacks include difficulties in reproducing experiments and an increased model complexity.

The simulator is open source, and is very popular in the academic world. It supports several platforms, such as Windows, Linux and Solaris. Detailed instructions on obtaining the distribution and setting up the environment can be found in [53]. The package requires the use of some GNU tools, so it can be configured freely and easily.

The most detailed simulator in the tool set is sim-outorder. It is the one usually modified and used when evaluating advanced architectural features (such as hardware support for sparse matrix formats and multimedia extensions). The simulator executes the instructions in a pipelined fashion. The main loop of the simulator has the following structure [53]:

```
ruu_init();
for (;;) {
    ruu_commit();
    ruu_writeback();
    lsq_refresh();
    ruu_issue();
    ruu_dispatch();
    ruu_fetch();
}
```

The execution of the pipeline is done in reverse, and a single pass through each stage is enough to handle inter-stage latch synchronization.

The fetch stage is implemented in ruu_fetch(). In each cycle, it fetches instructions from the Instruction Cache, and places them into the dispatch que.

The dispatch stage is performed in ruu_dispatch(). Instruction decoding and register renaming are performed at this point. The dispatcher takes as many instructions as possible (limited by the dispatch width of the target machine) and places them into the scheduler queue.

The issue stage is simulated in ruu_issue() and lsq_refresh(). In each cycle, the scheduling code locates the instructions for which the register inputs are ready. ruu_issue() also performs the execute stage. A number of instructions that are ready for execution are taken from the scheduler que (up to the issue width). If the functional units are available, the instructions are issued. The routine also schedules write back events, taking into account the latencies of the functional units.

The ruu_writeback() function performs the writeback stage. The event que is scanned, searching for completed instructions. If a completed instruction is found, all the dependencies are checked to mark the instructions that were waiting for the completing instruction. The ones that were waiting only for the current (completing) instruction are marked as ready to be issued. In this stage, checks for branch misspredictions are performed, and any erroneously issued instructions are discarded.

In the ruu_commit() function, instructions coming from the writeback stage are handled. The instructions are commited in-order. When an instruction is commited, the result is placed into the register file.

Cache simulation is also possible with SimpleScalar. The user can choose the cache configuration by setting the size of the L1 and L2 caches, the number of sets in the cache, the associativity and the replacement policy (FIFO, LRU or random) for the caches.

Latency for the memory system can also be specified (the latency in cycles for the L1 and L2 instruction and data cache and also for the memory). The width of the memory bus can also be specified.

Several branch prediction algorithms can also be selected, like a bimodal predictor, a 2-level adaptive predictor, a combined one (bimodal and 2-level adaptive) or a simple always predict (not) taken predictor. A symbolic debugger is available, called DLite!.

A.2 Default SimpleScalar configuration

The default SimpleScalar configuration as provided by the -dumpconfig parameter:

```
# load configuration from a file # -config

# dump configuration to a file # -dumpconfig

# print help message # -h                          false

# verbose operation # -v                          false
```

```
# enable debug message # -d                              false

# start in Dlite debugger # -i                             false

# random number generator seed (0 for timer seed) -seed 1

# initialize and terminate immediately # -q false

# restore EIO trace execution from <fname> # -chkpt <null>

# redirect simulator output to file (non-interactive only) #
-redir:sim                  <null>

# redirect simulated program output to file # -redir:prog <null>

# simulator scheduling priority -nice                                0

# maximum number of inst's to execute -max:inst 0

# number of insts skipped before timing starts -fastfwd 0

# generate pipetrace, i.e., <fname|stdout|stderr> <range> # -ptrace
<null>

# instruction fetch queue size (in insts) -fetch:ifqsize 4

# extra branch mis-prediction latency -fetch:mplat 3

# speed of front-end of machine relative to execution core
-fetch:speed                      1

# branch predictor type {nottaken|taken|perfect|bimod|2lev|comb}
-bpred                         bimod

# bimodal predictor config (<table size>) -bpred:bimod 2048

# 2-level predictor config (<l1size> <l2size> <hist_size> <xor>)
-bpred:2lev            1 1024 8 0

# combining predictor config (<meta_table_size>) -bpred:comb 1024

# return address stack size (0 for no return stack) -bpred:ras 8

# BTB config (<num_sets> <associativity>) -bpred:btb             512
4
```

```
# speculative predictors update in {ID|WB} (default non-spec) #
-bpred:spec_update         <null>

# instruction decode B/W (insts/cycle) -decode:width 4

# instruction issue B/W (insts/cycle) -issue:width 4

# run pipeline with in-order issue -issue:inorder false

# issue instructions down wrong execution paths -issue:wrongpath
true

# instruction commit B/W (insts/cycle) -commit:width 4

# register update unit (RUU) size -ruu:size 16

# load/store queue (LSQ) size -lsq:size                          8

# l1 data cache config, i.e., {<config>|none} -cache:dl1
dl1:128:32:4:l

# l1 data cache hit latency (in cycles) -cache:dl1lat 1

# l2 data cache config, i.e., {<config>|none} -cache:dl2
ul2:1024:64:4:l

# l2 data cache hit latency (in cycles) -cache:dl2lat 6

# l1 inst cache config, i.e., {<config>|dl1|dl2|none} -cache:il1
il1:512:32:1:l

# l1 instruction cache hit latency (in cycles) -cache:il1lat 1

# l2 instruction cache config, i.e., {<config>|dl2|none} -cache:il2
dl2

# l2 instruction cache hit latency (in cycles) -cache:il2lat 6

# flush caches on system calls -cache:flush                  false

# convert 64-bit inst addresses to 32-bit inst equivalents
-cache:icompress              false

# memory access latency (<first_chunk> <inter_chunk>) -mem:lat 18 2
```

```
# memory access bus width (in bytes) -mem:width 8

# instruction TLB config, i.e., {<config>|none} -tlb:itlb
itlb:16:4096:4:l

# data TLB config, i.e., {<config>|none} -tlb:dtlb dtlb:32:4096:4:l

# inst/data TLB miss latency (in cycles) -tlb:lat 30

# total number of integer ALU's available -res:ialu 4

# total number of integer multiplier/dividers available -res:imult 1

# total number of memory system ports available (to CPU)
-res:memport                    2

# total number of floating point ALU's available -res:fpalu 4

# total number of floating point multiplier/dividers available
-res:fpmult                     1

# profile stat(s) against text addr's (mult uses ok) # -pcstat
<null>

# operate in backward-compatible bugs mode (for testing only)
-bugcompat                  false
```