Customized Vector Instruction Set Architecture

Cătălin Ciobanu, Bogdan Spinean, Georgi Kuzmanov, Georgi Gaydadjiev
{catalin, bogdan}@ce.et.tudelft.nl, {g.k.kuzmanov, g.n.gaydadjiev}@tudelft.nl
Computer Engineering Laboratory, Electrical Engineering Department,
Delft University of Technology,
Postbus 5031, 2600 GA Delft, The Netherlands
Tel (+31) 15 27 82 226, Fax (+31) 15 27 84 898

Abstract— **This paper presents a methodology for synthesizing customized vector ISAs for various application domains targeting high performance execution. 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 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.**

Keywords—**Vector processors, DLP, ISA, Vector Architecture, Performance**

I. Introduction

Processors that use vector instruction sets operate upon multiple elements with a single 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. Applications with inherent DLP range from manipulating large matrixes (scientific software or telecommunication algorithms that process large graph structures) to streaming applications.

In streaming applications (such as video encoding/decoding), no dependencies exist between the loop iterations. This was the reason for creating the multimedia SIMD (Single Instruction, Multiple Data) extensions [14], [15], [10], [8], [6], [16], [7], [3]. 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). The Multimedia SIMD extensions have been extensively studied and are actually implemented in a range of general purpose processors. 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 vector computers such as IBM/370 [12], [13], [2] provided Vector instructions sets long before the wide adoption of multimedia extensions. Adding instructions to the ISA has some disadvantages. 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.

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 optimized (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.

The paper is organized as follows: in Section II the

ISA customization framework is presented. The targeted applications are described in Section III. Section IV contains details regarding the experimental results, and Section V presents the conclusions.

II. Design framework

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 approach 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. After several applications are processed this way, the new ISA can be produced.

A. Considered architecture

Figure 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 the Vector memory units have larger memory bandwidth. The Vector unit doesn't include any data caches.

The modeled architecture is of register to register type, so all the data has to be loaded from memory first, and the final results have to be written to the memory. Figure 2 presents the architecture of the register file. Separate Integer and Floating point vector register files are modeled. For each Vector Register, we provide a Bit Vector register, 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 [1] 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.

B. Simulation environment

In order to evaluate the performance improvement due to the vector operations for the targeted applica-

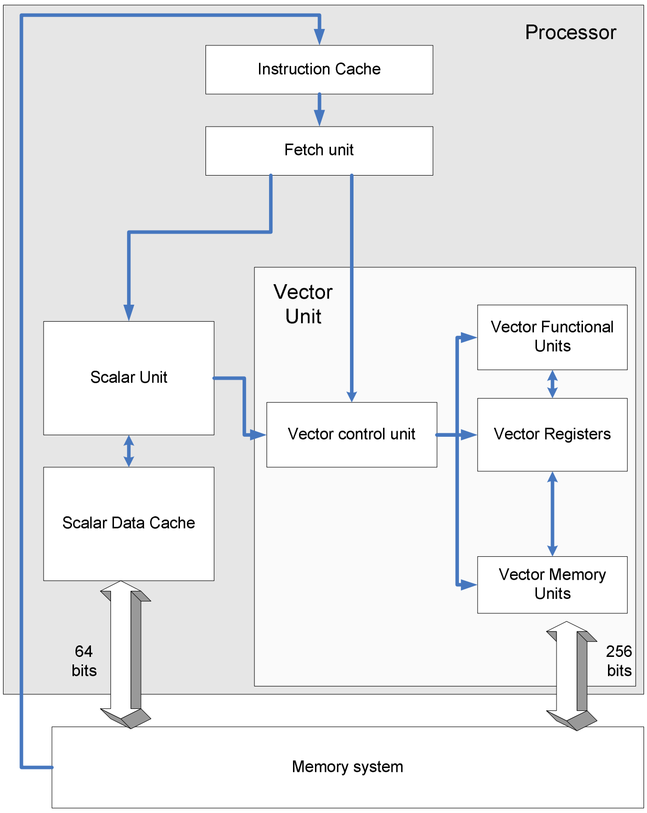

Fig. 1. General Vector architecture

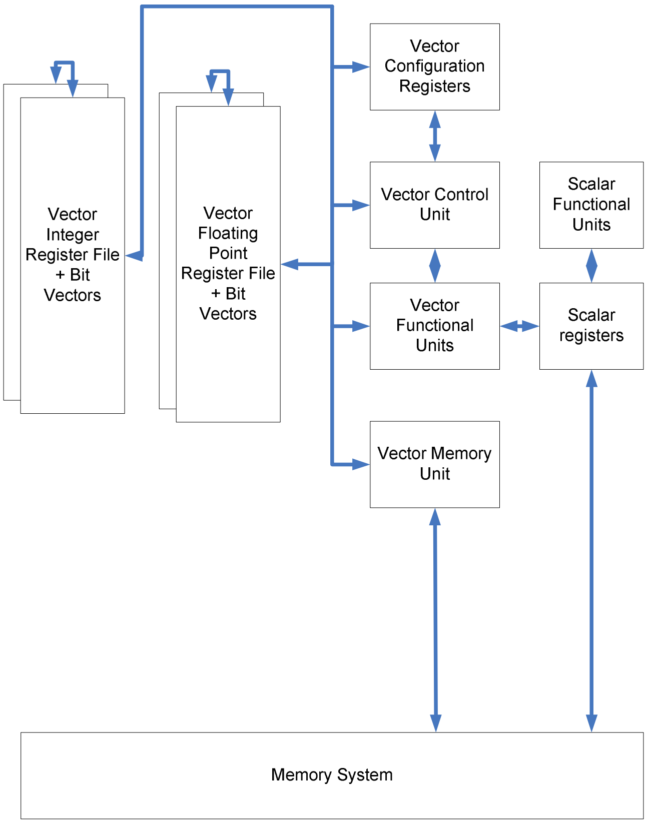
Vector Configuration Registers
Vector Integer Register File + Bit Vectors
Vector Floating Point Register File + Bit Vectors
Vector Control Unit
Scalar Functional Units
Vector Functional Units
Scalar registers
Vector Memory Unit
Memory System

Fig. 2. Vector Register File architecture

tions, 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 detailed simulator in the SimpleScalar distribution. In order to simulate vector instructions, support for new instructions and functional units was added to sim-outorder. The SimpleScalar Instruction and Architecture Tool(SSIAT) [9] was used in order to automatically modify the Simplescalar source files. The vector instructions were introduced in the C code by using the inline assembly features of GCC.

C. Simulator extensions

The simulator is extended by using SSIAT. This allows the addition of new instructions, functional units and register files. However, most micro architectural details had to be manually added to SimpleScalar.

C.1 The vector register file

Separate integer and floating point vector register banks have been simulated. Each integer or floating point register is linked to 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.

The proposed architecture includes 3 special configuration registers, declared in the "vconf" register bank. These registers are the Vector Length , Index register and Vmasks registers. Before the execution of vector instructions can proceed, initialization of the vconf registers is necessary. The Vector Length register 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 register stores the index of the current section of the processed array. The Vmask register controls whether vector masks are used in the computation or not.

Each vector register stores a fixed number of elements. This micro-architectural parameter is called the section size of the vector processor. In order to increase the portability of the vectorized code, we are using auto sectioning: special instructions update the Vector Length and Index registers, without having to know the section size of the target processor at compile time.

C.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. Two latencies are defined for each functional unit: 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. Four vector functional units have been defined:

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 used to compute the latency and the issue_latency are the following:

- **For the arithmetic units:**

$Operation\ latency = \frac{section\ size}{number\ of\ lanes} + basic\ latency$

$Issue\ latency = \frac{section\ size}{number\ of\ lanes}$

- **For the memory units:**
  - Loads:

$Operation\ latency = \frac{section\ size}{memory\ bandwidth(inelements)} + memory\ access\ latency$

$Issue\ latency = \frac{section\ size}{memory\ bandwidth}$

  - Stores:

$Operation\ latency = \frac{section\ size}{memory\ bandwidth}$

$Issue\ latency = \frac{section\ size}{memory\ bandwidth}$

C.3 The vector instructions

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, this field is checked and the semantics of the annotated instruction can be implemented.

SSIAT operates in the following way: if new instructions are added, it will automatically annotate some already defined instructions. This is transparent to the programmer, so while writing an application that uses the new operations the custom opcodes can be used directly. SSIAT modifies the sources of Simplescalar to add support for the annotated instructions. When compiling a new C program, assembly (.s) code is generated. SSIAT then parses this .s file and replaces the new opcodes with annotated instruction opcodes.

For example, the code

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

becomes

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

after runing SSIAT.

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.

D. Default parameters for performance simulations

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

- 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 vector registers and their corresponding bit vectors
- Default optimizations used for GCC (O0)
- Default configuration for the Scalar Unit, as provided with the public version of SimpleScalar simplesim-3v0d, including a latency of 1 cycle for the L1 cache, 6 cycles for the L2 cache and 18 cycles for the main memory

III. Applications

Four applications have been chosen for vectorization: three algorithms which compute the shortest paths in a directed graph (Dijkstra, Floyd and Bellman Ford) and Linpack, the floating point benchmark.

A. Dijkstra

Graph algorithms are often used in networking. Dijkstra is a well known algorithm that computes the shortest path from a source node to all the other nodes inside a graph. Other popular shortest path algorithms are Floyd and Bellman Ford. 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 [5] suite uses a linked list implemented with pointers to store the Candidate nodes in the algorithm. This makes the code virtually impossible to vectorize. 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. The chosen implementation has a complexity of $O(V^2)$, where V is the number of nodes in the graph. Profiling information shows that 99% of the time is spent in the main computation loop of this algorithm. The code is composed of one exterior loop and two interior for-loops. The first for loop computes the minimum path cost and its position in the current solution, while the second one tries to improve the current solution by using the minimum value.

The custom vector ISA created for Dijkstra is composed of 13 instructions (Table I). 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).

B. 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. Profiling shows that 98% of the execution time is spent in the main kernel of this algorithm. The main kernel is composed of three nested loops. Only

TABLE I
Custom vector ISA used for Dijkstra

Instruction	Synopsis	Uses masks
v.compare.gt VBID, VRIS, VRIT	VBID[i] ← VRIS[i] > VRIT[i]	Y
v.getmin VRID, RS, RT	RS ← MIN(VRID), RT ← pMIN(VRID)	Y
v.init VRIS, RT	VRIS[] ← RT	Y
v.ld VRIS, RT	VRIS[i] ← RT[i]	Y
v.ld.bi VBIS, RT	VBIS[i] ← RT[i] == 0 ? 0 : 1	N
v.masksoff	vmask ← 0	N
v.maskson	vmask ← 1	Y
v.mov.bi VBIS, VBIT	VBIS ← VBIT	N
v.sadd VRID, RS, VRIT	VRID[i] ← RS + VRIT[i]	Y
v.setvconf RD, RS, RT	vl ← RD, vindex ← RS, vmask ← RT	N
v.st RS, VRIT	RS[i] ← VRIT[i]	Y
v.updateindex RS	vindex ← RS ← vindex + section_size	N
v.updatevl RS	vl ← RS ← MAX(0, vl - section_size)	N

the most interior one can be vectorized. The custom vector ISA synthesized for Floyd is listed in Table II and is composed of 11 instructions.

C. 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. 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: Dijkstra's 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).

The profiling information shows that the main kernel consumes almost 100% of the execution time. The main loop processes all the edges in the graph V times (V is the number of nodes in the graph). Using each edge, the relaxing operation is performed - testing if it can improve the current best known solution. The interior loop counts to the number of edges (E), which for a very dense graph can be close to V^2. This is the main reason this algorithm is slower than Dijkstra. The graph we used for benchmarking was very dense, which meant that the simulation time for this algorithm was an order of magnitude larger than for Dijkstra and Floyd. This prevented us from completing the tests for configurations featuring section sizes larger than 64 elements, as the time required to simulate vectorized version of Bellman increased proportionally with the section size.

Because indirect addressing is used, extra care must be taken 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 iterations of the loop can provide a data dependency that stops the compiler from performing automatic vectorization of this loop. The custom vector ISA for Bellman-Ford is presented in Table III and contains 18 instructions.

D. Linpack

Linpack [4] is a well known floating point benchmark used to grade the performance of the supercomputers in the famous Top500 Supercomputer Sites [11]. 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. Profiling shows that 75% of the computation time is spent in the daxpy() kernel which consists of a vector multiply-with-scalar-and-add operation kernel and 25% is spent in the matgen() function which generates the input matrixes. The custom vector ISA created for Linpack contains 6 instructions (Table IV).

TABLE II
Custom vector ISA used for Floyd

Instruction	Synopsis	Uses masks
v.compare.gt VBID, VRIS, VRIT	VBID[i] ← VRIS[i] > VRIT[i]	Y
v.init VRIS, RT	VRIS[] ← RT	Y
v.ld VRIS, RT	VRIS[i] ← RT[i]	Y
v.masksoff	vmask ← 0	N
v.maskson	vmask ← 1	Y
v.mov.bi VBIS, VBIT	VBIS ← VBIT	N
v.sadd VRID, RS, VRIT	VRID[i] ← RS + VRIT[i]	Y
v.setvconf RD, RS, RT	vl ← RD, vindex ← RS, vmask ← RT	N
v.st RS, VRIT	RS[i] ← VRIT[i]	Y
v.updateindex RS	vindex ← RS ← vindex + section_size	N
v.updatevl RS	vl ← RS ← MAX(0, vl - section_size)	N

TABLE III
Custom vector ISA used for Bellman Ford

Instruction	Synopsis	Uses masks
v.add VRID, VRIS, VRIT	VRID[i] ← VRIS[i] + VRIT[i]	Y
v.and.bi VBID, VBIS, VBIT	VBID[i] ← VBIS[i] && VBIT[i]	N
v.compare.gt VBID, VRIS, VRIT	VBID[i] ← VRIS[i] > VRIT[i]	Y
v.getss RS	RS ← section_size	N
v.init VRIS, RT	VRIS[] ← RT	Y
v.ld VRIS, RT	VRIS[i] ← RT[i]	Y
v.ldindexed VRID, RS, VRIT	VRID[i] ← RS[VRIT[i]]	Y
v.masksoff	vmask ← 0	N
v.maskson	vmask ← 1	Y
v.mov.bi VBIS, VBIT	VBIS ← VBIT	N
v.priority RD, VBIS, VBIT	RD ← position of first non-zero value of VBIT, VBIS[i] = 1 ← if i < RD, 0 otherwise	N
v.scompare.eq VBID, VRIS, RT	VBID[i] ← VRIS[i] == RT	Y
v.setvconf RD, RS, RT	vl ← RD, vindex ← RS, vmask ← RT	N
v.st RS, VRIT	RS[i] ← VRIT[i]	Y
v.stindexed RD, VRIS, VRIT	RD[VRIT[i]] ← VRIS[i]	Y
v.sumup RS, VRIT	RS ← 0, RS += VRIT[i]	Y
v.updateindex RS	vindex ← RS ← vindex + section_size	N
v.updatevl RS	vl ← RS ← MAX(0, vl - section_size)	N

TABLE IV
Custom vector ISA used for Linpack

Instruction	Synopsis	Uses masks
v.ld.d VRDS, RT	VRDS[i] ← RT[i]	Y
v.msadd.d VRDD, FS, VRDT	VRDD[i] += FS * VRDT[i]	N
v.setvconf RD, RS, RT	vl ← RD, vindex ← RS, vmask ← RT	N
v.st.d RS, VRDT	RS[i] ← VRDT[i]	Y
v.updateindex RS	vindex ← RS ← vindex + section_size	N
v.updatevl RS	vl ← RS ← MAX(0, vl - section_size)	N

IV. Experimental results

The 24 vector custom vector instructions in Table V have been derived as a union of Tables I-IV. They can be divided into general purpose and application specific instructions. If complex 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.

We propose three application specific instructions: **v.getmin VRID, RS, RT**(Compute the minimum from an array) - used in Dijkstra, **v.msadd VRDD, FS, VRDT** (Vector multiply with scalar and add) - used in Linpack and **v.priority RD, VBIS, VBIT** (Detect the first non-zero element of a bitvector) - used in Bellman-Ford. 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.

After vectorization, the maximum speedup obtained for Dijkstra compared to the original version is 24.88X for a Section Size of 128 (Figure 3). 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 the importance of the relation between the Section Size and the problem size. 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 penalized because a large section size implies higher 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. Theoretically, the best performance is obtained when the vector lengths are almost equal but not greater than the Section Size. The speedup for Dijkstra 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.

By increasing the Section Size, the impact of increased memory latency is much smaller. This suggests that by using high bandwidth memory but with slightly higher 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 (Figure 3). For a memory latency of 1 cycle (equivalent to a perfect Level1 data cache), the difference is less than 50%. We observe that the performance starts to drop sharply when the Memory Latency exceeds 18-20 cycles while the difference in execution time between a memory latency of 1, 6 and 18 cycles is relatively low, becoming 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.

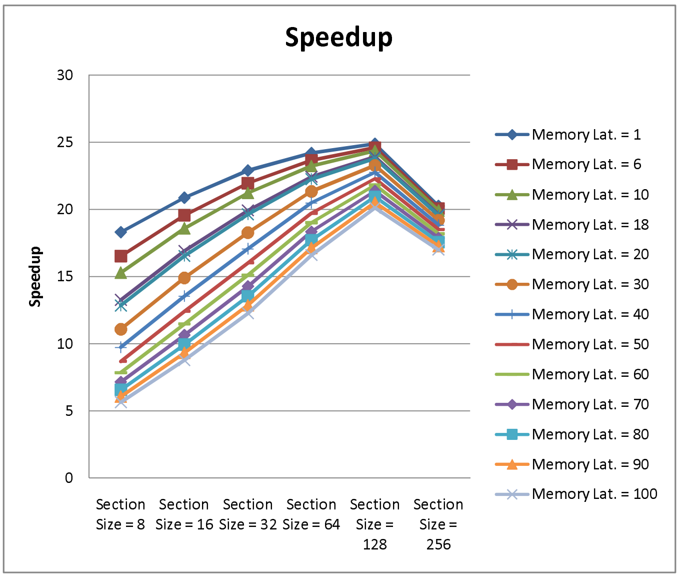

Fig. 3. Dijkstra speedup when varying the section size compared to the original MIBENCH version

Similar results were obtained for Floyd (Figure 4. After vectorizing the code using the instructions listed in Table II the maximum speedup achieved is 4.99X for Section Size = 128, and a Memory Latency of 1 cycle. If Memory Latency is set to 18 cycles, reported speedup is 4.86X, very close to the maximum achieved. As the input graph used for Floyd is the same one we have used also for Dijkstra, best performance is obtained for a Section Size of 128. The drop in performance for having slower memory (latency of 18 cycles instead of 1 cycle) for this Section Size is 2.65%. Performance drops sharply as the Memory Latency increases above 20 cycles, especially for small Section Sizes (8 and 16 elements).

The maximum speedup obtained by using the 18 vector instructions listed in Table III is 9.27X for Section Size = 64 and Memory Latency = 1 cycle for

TABLE V
Vector instructions sorted alphabetically

Instruction	Synopsis	Uses masks
v.add VRID, VRIS, VRIT	VRID[i] ← VRIS[i] + VRIT[i]	Y
v.and.bi VBID, VBIS, VBIT	VBID[i] ← VBIS[i] && VBIT[i]	N
v.compare.gt VBID, VRIS, VRIT	VBID[i] ← VRIS[i] > VRIT[i]	Y
v.getmin VRID, RS, RT	RS ← MIN(VRID), RT ← pMIN(VRID)	Y
v.getss RS	RS ← section_size	N
v.init VRIS, RT	VRIS[] ← RT	Y
v.ld VRIS, RT	VRIS[i] ← RT[i]	Y
v.ldindexed VRID, RS, VRIT	VRID[i] ← RS[VRIT[i]]	Y
v.ld.bi VBIS, RT	VBIS[i] ← RT[i] == 0 ? 0 : 1	N
v.ld.d VRDS, RT	VRDS[i] ← RT[i]	Y
v.masksoff	vmask ← 0	N
v.maskson	vmask ← 1	Y
v.mov.bi VBIS, VBIT	VBIS ← VBIT	N
v.msadd.d VRDD, FS, VRDT	VRDD[i] += FS * VRDT[i]	N
v.priority RD, VBIS, VBIT	RD ← position of first non-zero value of VBIT, VBIS[i] = 1 ← if i < RD, 0 otherwise	N
v.sadd VRID, RS, VRIT	VRID[i] ← RS + VRIT[i]	Y
v.scompare.eq VBID, VRIS, RT	VBID[i] ← VRIS[i] == RT	Y
v.setvconf RD, RS, RT	vl ← RD, vindex ← RS, vmask ← RT	N
v.st RS, VRIT	RS[i] ← VRIT[i]	Y
v.stindexed RD, VRIS, VRIT	RD[VRIT[i]] ← VRIS[i]	Y
v.st.d RS, VRDT	RS[i] ← VRDT[i]	Y
v.sumup RS, VRIT	RS ← 0, RS += VRIT[i]	Y
v.updateindex RS	vindex ← RS ← vindex + section_size	N
v.updatevl RS	vl ← RS ← MAX(0, vl - section_size)	N

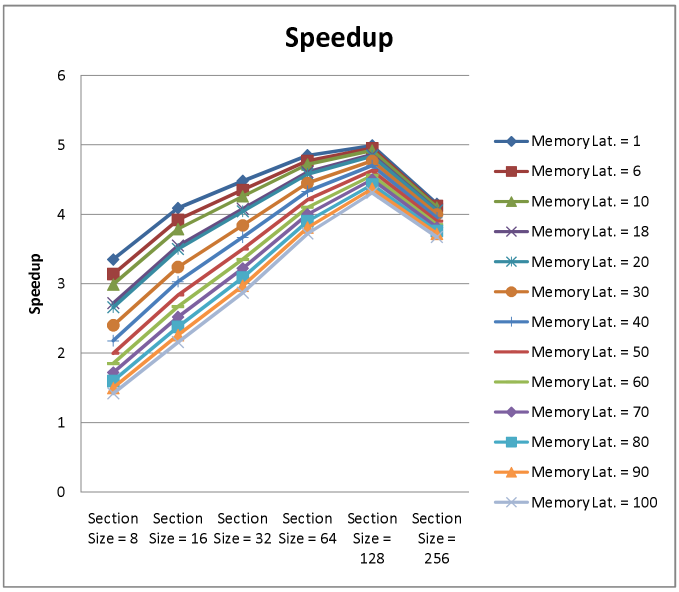

Fig. 4. Floyd speedup when varying the section size

Bellman-Ford. (Figure 5). For Memory Latency = 18 cycles, the speedup remains high (8.18X), almost the double of the speedups seen for the other graph algorithms.

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(Figure 5). As a consequence, having a large Section Size is really important. 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.

The customized instruction set for Dijkstra provided consistent improvements of the execution time that justified the hardware and software support nec-

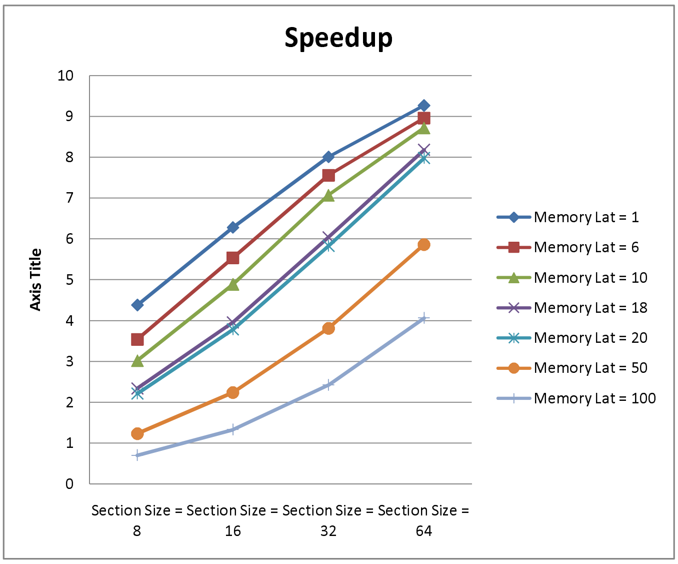

Fig. 5. Bellman-Ford speedup when varying the section size

essary to actually apply vectorization in a real system. 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.

Using 6 vector instructions (Table IV) Linpack executes 4.33 times faster(Figure 6) for a Section Size of 32 and Memory Latency of 1 cycle. 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. For a memory latency of 18 cycles, the smallest execution time is achieved by the configuration featuring a Section Size of 64, and the difference in execution 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.

Because the daxpy() loop is very simple(two loads, one multiply-add and one store), increased 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. 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, close to the 4.33X speedup for a Memory Latency of 1 and a Section Size of 8 elements.

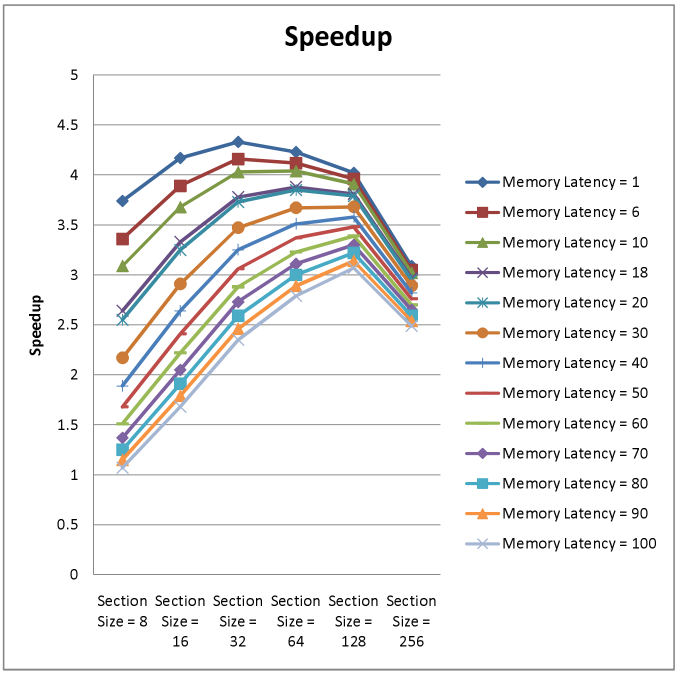

Fig. 6. Linpack speedup when varying the section size

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

We also provide the following general observations about the factors that influence performance of the vectorized code:

- Increased Memory Latency drastically reduces the performance of small Section Sizes
- Having a vector length as close to the Section Size as possible is desirable.
- Large Section Sizes compensate for the high startup costs of large Memory Latencies
- The performance doesn't decrease significantly for a Memory Latency of 18 cycles compared to the fastest latency of 1 cycle. Thus, an implementation without caches for the vector unit can potentially deliver good performance, as long as enough vector registers are

present to store the intermediate results.

V. Conclusions

Three algorithms that compute the shortest paths in a directed graph (Dijkstra, Floyd, and Bellman-Ford) and the Linpack benchmark were analyzed and a custom Vector ISA was synthesized for each application, containing 21 general purpose and 3 application specific vector instructions. Simulations were used to compare the performance of the custom vector instruction sets against a scalar ISA. When benchmarking, two critical parameters were considered: the section size of the vector register file and the memory latency of the vector memory unit. The experimental results suggest that the customized Vector ISAs deliver substantial performance benefits in the targeted application domains. The design framework created is flexible and can be used for future analysis of a wider range of applications.

Future research directions. We identify the following problems which can be addressed for a 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. By investigating the micro architecture of the Vector Register File, performance can be potentially improved. The Vector Memory Unit is expected to be the major bottleneck of the micro-architecture. Therefore, a more thorough analysis of how to improve the memory performance is needed.

Acknowledgements

This work was partially supported by the Dutch Ministry of Education, Culture and Science through the HSP Huygens Programme; the Dutch Technology Foundation STW, applied science division of NWO, the Technology Program of the Dutch Ministry of Economic Affairs (project DCS.7533); and the European Commission in the context of the Scalable computer ARChitectures (SARC) integrated project #27648 (FP6).

References

[1] Todd Austin. Simplescalar http://www.simplescalar.com/.

[2] W. Buchholz. The ibm system/370 vector architecture. *IBM Systems Journal*, page 51, 1986.

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

[4] 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.

[5] Matthew Guthaus, Jeff Ringenberg, Todd Austin, Trevor Mudge, and Richard Brown. Mibench http://www.eecs.umich.edu/mibench/.

[6] Linley Gwennap. Digital, mips add multimedia extensions. *MICRODESIGN RESOURCES*, 10(15):1–5, 1996.

[7] Linley Gwennap. Altivec vectorizes powerpc. *MICROPROCESSOR REPORT*, 12(6):1–5, May 1998.

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

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

[10] Alex Klimovitski. Using sse and sse2: Misconceptions and reality. *Intel Developer UPDATE Magazine*, pages 1–8, March 2001.

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

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

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

[14] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel mmx for multimedia pcs. *COMMUNICATIONS OF THE ACM*, 40(1):25–38, January 1997.

[15] Shreekant(Ticky) Thakkar and Tom Huff. Internet streaming simd extensions. *IEEE Computer*, 32(12):26–34, December 1999.

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