A Vector-μSIMD-VLIW Architecture for Multimedia Applications

Esther Salamí and Mateo Valero [*]
Computer Architecture Department
Universitat Politècnica de Catalunya, Barcelona, Spain
{esalami,mateo}@ac.upc.es

Abstract

Media processing has motivated strong changes in the focus and design of processors. These applications are composed of heterogeneous regions of code, some of them with high levels of DLP and other ones with only modest amounts of ILP. A common approach to deal with these applications are μSIMD-VLIW processors. However, the ILP regions fail to scale when we increase the width of the machine, which, on the other hand, is desired to achieve high performance in the DLP regions. In this paper, we propose and evaluate adding vector capabilities to a μSIMD-VLIW core to speed-up the execution of the DLP regions, while, at the same time, reducing the fetch bandwidth requirements. Results show that, in the DLP regions, both 2 and 4-issue width Vector-μSIMD-VLIW architectures outperform a 8-issue width μSIMD-VLIW in factors of up to 2.7X and 4.2X (1.6X and 2.1X in average) respectively. As a result, the DLP regions become less than 10% of the total execution time and performance is dominated by the ILP regions.

1 Introduction

As technology evolves, the number of transistors to be included on a single chip will continue increasing [40]. To take benefit of these additional resources, most of the traditional techniques focus on exploiting more *Instruction Level Parallelism* (ILP) [28].

Superscalar processors are the most traditional ILP implementation for the general purpose domain. However, it is widely assumed that current superscalar processors cannot be scaled by simply fetching, decoding and issuing more instructions per cycle [17]. Branches, the instruction cache bandwidth, the instruction window size, the register file and the memory wall are some of the aspects that currently limit the scalability of superscalar processors. And, even if these problems could be overcome with future technology, the performance results would not pay off the amount of chip area and power and the design effort required [27].

Very Long Instruction Word (VLIW) processors are another form of exploiting ILP that requires less hardware complexity. The compiler and not the hardware is responsible for identifying groups of independent operations and packaging them together into a single VLIW instruction [9]. The first generation of VLIW processors were successful in the scientific domain [4, 29], and it has also been the architecture of choice for most media embedded processors [26, 38, 37]. However, some relevant facts, such as code compatibility and non-deterministic latencies, have contributed to the belief that VLIW processors are not appropriate for the general-purpose domain. At present, a revival of the VLIW execution paradigm is observed. HP and Intel have recently introduced a new style of architecture known as *Explicitly Parallel Instruction Computing* (EPIC) [35] and a specific architecture implementation: the *Itanium Processor Family* (IPF) [36]. EPIC retains compatibility across different implementations without the complexity of superscalar control logic.

Another kind of parallelism that can be found in programs is *Data Level Parallelism* (DLP) (or *Single Instruction Multiple Data* (SIMD)) [10]. The DLP paradigm tries to specify with a single instruction a large number of operations to be performed on independent data words. Traditionally, this kind of parallelism has been successfully exploited in the supercomputing domain by vector [31, 3, 39] and array [13, 30] processors. However, during the last decade, the increasing significance of media processing has motivated a great interest in exploiting sub-word level parallelism (also called μSIMD parallelism) [25]. In the general purpose domain, these changes have been very straightforward with the inclusion of multimedia extensions such as SSE [15] or Altivec [24]. This is also a form of DLP in which short data are packed into a single register and opera-

[*]This work has been supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIC2001-0995-C02-01 and TIN2004-07739-C02-01, and by the European HiPEAC network of Excellence. We also acknowledge the Supercomputing Center of Catalonia (CESCA) for supplying the computing resources for our research.

tions are carried out simultaneously on the different register elements. A third way of exploiting DLP comes from the combination of the previous two [6, 18, 20]. These architectures adapt to typical multimedia patterns by extending the scope of vectorization to two dimensions.

In the media domain, μSIMD-VLIW processors have been widely proposed [12, 26, 38, 8], as they are able to exploit DLP by means of the μSIMD operations and ILP by the use of wide-issue static scheduling. But, although media applications are usually characterized by high amounts of DLP, there is also a significant part of code that exhibits only modest amounts of ILP, thus taking little benefit from increasing the processor resources. And, even though VLIW processors are simpler than superscalar designs, very high issue rates require decoding more operations in parallel and complicate the register files, which clearly increases power consumption.

In this paper, we propose and evaluate a new architecture that includes vector operations in a μSIMD-VLIW processor to exploit the DLP typical of multimedia kernels in a more efficient way and with lower fetch bandwidth requirements. Although a quantitative analysis on power consumption is out of the scope of this paper, it is widely assumed that vector architectures contribute to increase power efficiency [2, 20]. Initial results for a reduced number of benchmarks were first presented in [34]. Apart from a more extensive evaluation, additional contributions also include a thorough description of the architecture and of the static scheduling of vector operations.

The rest of the paper is organized as follows. Section 2 defines the concept of scalar and vector regions and evaluates their scalability separately. Section 3 overviews the Vector-μSIMD-VLIW architecture and discusses the main compilation issues. Section 4 describes the modeled architectures and the simulation framework. Next, section 5 presents quantitative data such as speed-up and operation per cycle rates. Finally, the last section summarizes the main conclusions.

2 Scalar and Vector Regions

Most media applications consist on a set of algorithms that process streams of data in a pipeline fashion. Furthermore, the same set of operations are performed over the elements inside the stream. Therefore, media kernels exhibit high amounts of DLP [7]. On the other hand, there is also a significant portion of code that is difficult to vectorize. That is some protocol related processing overhead such as first order recurrences, table look-ups and non-streaming memory patterns with large amounts of indirections. Therefore, a real media program is composed of heterogeneous regions of code with highly variable levels of parallelism: some of them with high amounts of DLP and the other ones with only modest amounts of ILP. We will refer to those regions that can be vectorized with the term of *Vector Regions* and to the remaining non-DLP regions of code with the term of *Scalar Regions*.

Table 1. Vector regions

Benchmark	%Vect	Vector Regions
JPEG_ENC	29.56 %	RGB to YCC color conversion Forward DCT Quantification
JPEG_DEC	18.46 %	YCC to YCC color conversion H2v2 up-sample
MPEG2_ENC	52.29 %	Motion estimation Forward DCT Inverse DCT
MPEG2_DEC	23.11 %	Form component prediction Inverse DCT Add block
GSM_ENC	18.66 %	LTP parameters Autocorrelation
GSM_DEC	0.91 %	Long term filtering

In order to evaluate the scalar and vector regions separately, we have marked the start and end point of the most computational intensive vector regions in the source codes. These regions generally correspond to one or two levels of nested loops plus some previous initializations. Table 1 lists the selected benchmarks, the parts of each program that have been considered as vector regions, and the percentage of the execution time they represent in a 2-issue width μSIMD-VLIW architecture. These benchmarks are representative programs of image, audio and video, all from the UCLA *Mediabench* suite [22].

Figure 1 shows the speed-up of 2, 4 and 8-issue width μSIMD-VLIW architectures over the 2-issue width μSIMD-VLIW (see section 4 for details about methodology and processor configurations). The dashed lines represent the speed-up in the vector/scalar regions over the vector/scalar regions of the 2-issue width architecture. The solid lines refer to the speed-up in the full application.

From the graphs, it can be observed that, except for the *gsm_enc*, the scalar regions fail to scale above 4-issue width. While increasing the width of the architecture from 2 to 4 provides an average speed-up of 1.24X in the scalar regions, moving from 4 to 8-issue only introduces a small 1.03X performance improvement. As far as the vector regions is concerned, they exhibit potential to benefit from wider issue scheduling, but this parallelism could be exploited in a more efficient way by conventional DLP oriented techniques. Furthermore, even though the vector regions scale up to 3.19X for the *jpeg_dec* application (2.49X in average),

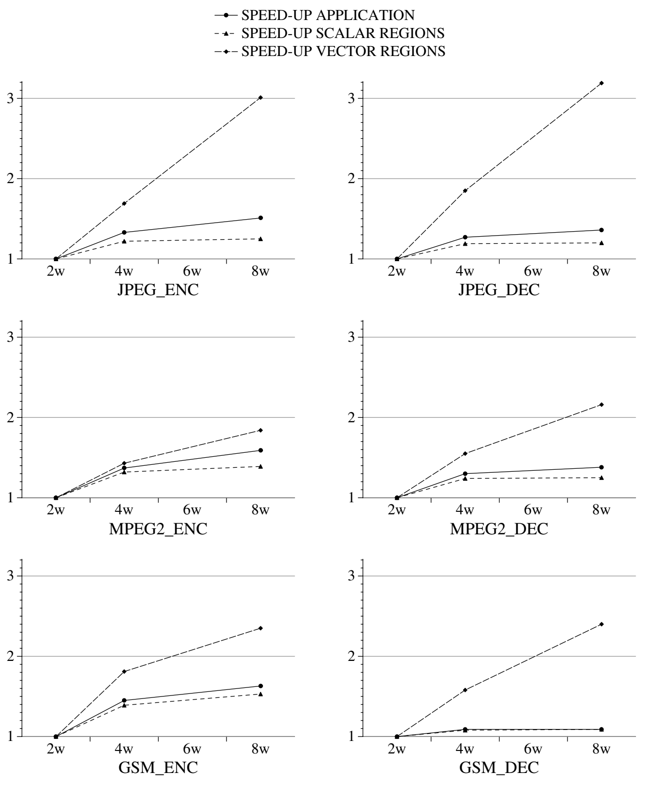


Figure 1. Scalability of scalar and vector regions in μSIMD-VLIW architectures

the vectorization percentage is low (24 % in average) and the lack of scalability in the scalar regions (1.28X in average) limits the performance of the complete application.

In any case, the actual performance achieved is very far from the theoretical peak performance and do not pay off the hardware complexity inherent in very aggressive architectures. We claim that Vector-μSIMD extensions arise as a better candidate to invest in, as they clearly reduce the fetch pressure, simplify the control flow and memory access, and speed-up the performance of the vector regions without detrimental effects over the scalar part.

3 Adding Vector Units to a VLIW processor

3.1 Vector-μSIMD ISA Overview

Our Vector-μSIMD ISA is based on the *Matrix Oriented Multimedia* (MOM) extension [6]. It can be viewed as a conventional vector ISA where each operation is a MMX-like operation. But it does not include costly vector operations, such as conditional execution, gathers or scatters.

It provides vector registers of 16 64-bit words each, vector load and vector store operations to move data from/to memory to/from the vector registers, and a set of computation operations that operate on vector registers. Since each word can pack either eight 8-bit, four 16-bit or two 32-bit items, each vector register can hold a matrix of up to 16x8 elements. The architecture also provides 192-bit packed accumulators similar to those proposed in the MDMX multimedia extension. Additionally, two special registers are required to control the execution of vector operations: the vector length register and the vector stride register.

As far as terminology is concerned, we reserve the term *operation* to refer to each independent machine operation codified into a VLIW *instruction*. Each vector operation executes so many *sub-operations* as the vector length dictates. Finally, as the maximum vector length is 16 and each sub-operation can operate on either eight 8-bit, four 16-bit or two 32-bit items, a vector operation can perform up to 16x8 *micro-operations*.

3.2 The Vector-μSIMD-VLIW Architecture

Figure 2 shows the main components of the proposed architecture. Essentially, it is a VLIW processor with the addition of a vector register file, one or more vector functional units, and a modified cache hierarchy specially targeted to serve vector accesses. Both, the vector register file and the vector functional units can be clusterized in independent vector lanes. This can be achieved with relatively simple logic by replicating the functional units, splitting each vector register across each lane and assigning each functional unit to a certain lane. The different elements of a vector register are interleaved across lanes, allowing all lanes to work independently. From the point of view of implementation, a vector register file scales better than a μSIMD one, due to the organization in lanes, which reduces the number of ports per cluster. For aggressive configurations, a vector register file can provide larger storage capacity with similar area cost and less access time [5]. In this work, we use four independent vector lanes. As our vector lengths are relatively short, a larger number of lanes would not pay off.

The Vector-μSIMD-VLIW architecture also includes a simple accumulator register file and adds limited connection between the lanes to be able to perform the last series of accumulation in a reduction operation. Only one of the lanes needs to read and write the source and destination packed accumulator. This lane is the responsible for performing the last reduction.

We use a *vector cache* [27] in the second level of the memory hierarchy. The vector cache is a two-bank interleaved cache targeted at accessing stride-one vector requests by loading two whole cache lines (one per bank) instead of individually loading the vector elements. Then, an interchange switch, a shifter, and a mask logic correctly align the data. Scalar accesses are made to the L1 data cache, while

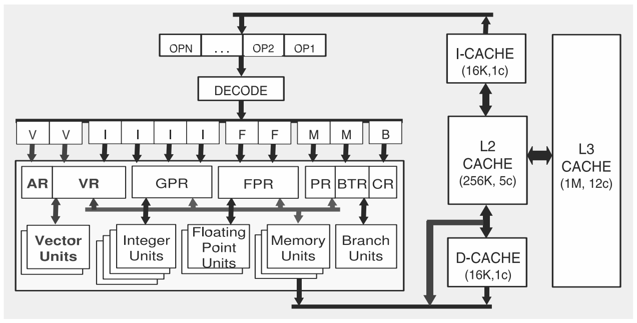
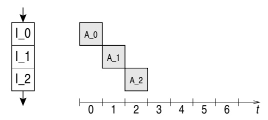
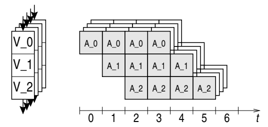


Figure 2. Vector-μSIMD-VLIW architecture



$$\begin{aligned} T_{er} &= 0 \\ T_{lr} &= 0 \\ T_{ew} &= 0 \\ T_{lw} &= L \end{aligned}$$

(a) Scalar operation



$$\begin{aligned} T_{er} &= 0 \\ T_{lr} &= \lfloor (VL-1)/LN \rfloor \\ T_{ew} &= 0 \\ T_{lw} &= L + \lfloor (VL-1)/LN \rfloor \end{aligned}$$

(b) Vector operation

Figure 3. Latency descriptors (*Ter* = earliest read, *Tlr* = latest read, *Tew* = earliest write, *Tlw* = latest write, L = flow latency, *VL* = vector length, *LN* = vector lanes)

vector accesses bypass the L1 to access directly the L2 vector cache. If the L2 port is $B\times$64-bit wide, these accesses are performed at a maximum rate of B elements when the stride is one, and at 1 element per cycle for any other stride. A coherency protocol based on an exclusive-bit policy plus inclusion is used to guarantee coherency.

3.3 Compilation Issues

The Achilles' heel of the proposed architecture is, obviously, the compiler; but nowadays there are compilers that allow basic autovectorization for μSIMD architectures, and the same compiler techniques could be used to generate Vector-μSIMD code. As we do not have a reliable compiler at our disposal yet, we have used emulation libraries to hand-write μSIMD and Vector-μSIMD code to evaluate the approach, and the compiler replaces the emulation functions calls by the corresponding operation.

From the VLIW point of view, new register files and functional units have been added, and some extra considerations must be taken into account by the scheduler, which is the module that needs the most detailed information about the target architecture, as it is responsible for assigning a schedule time to each operation, subject to the constraints of data dependence and resource availability. For every input and output operand, an earliest and a latest read and write latency must be specified respectively [1]. Figure 3.a depicts the execution of a 3 cycles fully-pipelined scalar operation. In this example, the source registers are read sometime during the first cycle after the initiation of the operation, and the result is written at the end of three cycles.

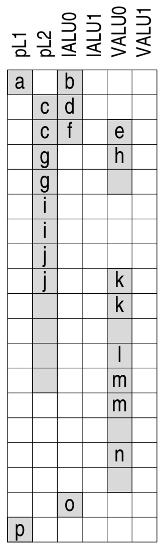
In the case of a vector operation, these values also depend on the vector length (VL) and on the number of parallel vector lanes (LN). As up to LN sub-operations are initiated per cycle, the last input operand will be read at $\lfloor (VL-1)/LN \rfloor$, and the last output will be written at $L + \lfloor (VL-1)/LN \rfloor$, being L the latency of one sub-operation (see Figure 3.b). The number of parallel vector lanes is a fixed parameter from the architecture and it is known at compile time; but the vector length is variable for each operation, and will be dynamically set. Fortunately, the vector length register is usually initialized with an immediate value, and a simple data flow analysis is able to provide the right value to the compiler. In the few cases in which the vector length is not known at compile time, the compiler must assume the maximum vector length (16) in order to ensure correctness. Note that, for a vector unit with four parallel lanes, the penalty to pay would be three extra cycles at worst (that is, if the vector length turns out to be four or less).

The same latency descriptors are taken for vector memory operations, but replacing the number of vector lanes by the width of the L2 port (in elements). As it was mentioned in Section 3.2, in the proposed architecture, the execution time of a vector memory operation also depends on the stride. For simplicity, our compiler schedules all vector memory operations as having a stride of one and hitting in the L2 vector cache, and the processor stalls at run-time if either of the two assertions is not true.

On the other hand, providing a register file which supports concurrent acceses to the same vector register, the compiler can do chaining [31] of two vector operations with a dependence on a vector register operand by just scheduling the second one before the first operation has completed execution.

3.3.1 Code Example

Figure 4 shows the scheduling of a vector code generated by the compiler for a 2-issue width VLIW architecture with two vector units and a wide 4x64 bit port to the vector cache. Latencies are 2 cycles for the vector units and 5 cycles for the vector cache. This code is taken from the *dist1* function in the *mpeg2_enc* application, and computes the *sum of* *absolutes differences* (SAD) between two blocks of $8x16$ pixels. It is assumed that registers $R1$ and $R2$ keep the initial address of the blocks, and lx is the stride between consecutive rows. As the registers are 64 bit wide, and the stride between rows is not one, we need two vector registers to keep each block. The SAD operation is implemented using a packed accumulator that allows parallel execution over the vector elements. Finally, the values packed in the accumulators are reduced and the final result is stored.



cyc		
0	(a) VS=lx	(b) VL=8
1	(c) V1=[R1]	(d) R3=R1+8
2	(e) A1=0	(f) R4=R2+8
3	(g) V2=[R2]	(h) A2=0
4	———	———
5	(i) V3=[R3]	———
6	———	———
7	(j) V4=[R4]	———
8	(k) A1=SAD(V1,V2)	———
9	———	———
10	———	———
11	(l) R5=SUM(A1)	———
12	(m) A2=SAD(V3,V4)	———
13	———	———
14	———	———
15	(n) R6=SUM(A2)	———
16	———	———
17	(o) R5=R5+R6	———
18	(p) [R7]=R5	———

Figure 4. Scheduling of motion estimation for a 2-issue Vector-μSIMD-VLIW processor

It can be observed that this kernel is memory bound and, in fact, the second vector unit is not used at all, as the second SAD operation (m) must wait for the data being loaded from memory and cannot be scheduled earlier. Chaining is performed between the vector loads (g) and (j) and the vector SAD operations (k) and (m) respectively. Note also that the vector loads are scheduled as having a stride of one, that is, as if they will produce four elements by cycle. As this assumption is not true, the processor will be stalled at run-time, thus incurring in a great penalty in performance, as we will see in the evaluation section.

Note that the two innerloops in the scalar version have been totally eliminated, and the Vector-μSIMD architecture only needs to decode 16 operations to process one complete block, in front of the 172 operations required in the μSIMD versions of code.

4 Methodology

4.1 Compilation and Simulation Framework

For our experiments we have used *Trimaran* [21]. Trimaran is a compiler infrastructure for supporting state of the art research in compiling for ILP architectures. The system is currently oriented towards EPIC architectures. To expose sufficient ILP it makes use of advanced techniques such as Superblock [14] or Hyperblock[23] formation. The architecture space is characterized by HPL-PD [19], a parameterized processor architecture.

Our internal release of the compiler also includes *Pcode* Interprocedural Pointer Analysis [11] and Cost Effective Memory Disambiguation [32]. Therefore, our scalar versions of code include memory disambiguation (inherent in the vector versions), which introduces an average performance speed-up of 1.32X (for a 8-issue width architecture) over the same codes compiled with the public release of Trimaran. We have used emulation libraries to hand-write the applications with μSIMD and Vector-μSIMD extensions. The compiler has been modified to detect the emulation functions calls and replace them by the related low level operations. Both, the compiler and the HPL-PD machine description have been enhanced with the new operations, register files and functional units. The simulator has also been extended to include the new ISAs and a detailed memory hierarchy.

4.2 Modeled Architectures

We have evaluated 2, 4 and 8-issue width VLIW and μSIMD-VLIW architectures and two different 2 and 4-issue width Vector-μSIMD-VLIW configurations. Table 2 summarizes the general parameters of the ten architectures under study. In order to support the high computational demand of multimedia applications, our configurations are quite aggressive in the number of arithmetic functional units. Latencies are based on those of the *Itanium2* processor [16].

The μSIMD-VLIW architecture includes 64-bit registers together with functional units able to operate on up to eight 8-bit items in parallel. This extension provides 67 opcodes fairly similar to Intel's SSE [15] integer opcodes. Note that the vector architectures are not balanced against the same issue width VLIW or μSIMD-VLIW architectures because we consider them as an alternative to wider issue processors. For example, the arithmetic capability of the 2-issue Vector2 and the 4-issue Vector1 configurations is comparable to that of the 8-issue μSIMD configuration, not to the 2 or 4-issue μSIMD.

The first level data cache is a 16 KB, 4-way set associative cache with one port for the reference 2-issue width

Table 2. Processor configurations

Resource	VLIW 2 / 4 / 8 w	+μSIMD 2 / 4 / 8 w	+Vector1 2 / 4 w	+Vector2 2 / 4 w
Int regs	64 / 96 / 128	64 / 96 / 128	64 / 96	64 / 96
SIMD regs	–	64 / 96 / 128	20 / 32 x16	20 / 32 x16
Acc regs	–	–	4 / 6	4 / 6
Int units	2 / 4 / 8	2 / 4 / 8	2 / 4	2 / 4
SIMD units	–	2 / 4 / 8	1 / 2 x4	2 / 4 x4
L1 ports	1 / 2 / 3	1 / 2 / 3	1	1 / 2
L2 ports	–	–	1 x4	1 x4

architecture. We consider pseudo-multi-ported caches for the configurations with greater number of ports. There is a 256KB vector cache in the second level and a 1MB cache in the third level. Latencies are 1 cycle to the L1, 5 cycles to the L2, 12 cycles to the L3 and 500 cycles to main memory. We have not simulated the instruction cache since our benchmarks have small instruction working set. The compiler schedules all memory operations assuming they hit in the cache and the processor is stalled at run-time in case of a cache miss or bank conflict.

5 Evaluation

5.1 Speed-up in Vector Regions

Figure 5.a shows the performance speed-up obtained in the vector regions with perfect memory simulation. By perfect memory we consider that all accesses hit in cache, but with the corresponding latency. That is, all scalar accesses are served after 1 cycle of latency and all vector accesses in the Vector configurations go to the L2 and take 5 cycles plus the additional cycles to serve all vector data elements (which slightly favours the VLIW and μSIMD-VLIW configurations). For each architecture, the graph shows the speed-up of the vector regions over the execution time of the vector regions in the 2-issue width VLIW architecture.

As it was to be expected, both μSIMD and Vector architectures clearly outperform the same issue VLIW architecture. The 2-issue width Vector2 architecture outperforms the same width μSIMD architecture in a factor ranging from 3.0X to 6.2X (4.4X in average). Furthermore, the 8-issue μSIMD is outperformed by both, the 2-issue Vector2 in a factor of up to 2.6X (1.7X in average), and the 4-issue Vector2 in a factor of up to 4.0X (2.3X in average).

We also observe that half of the benchmarks do not take much benefit of increasing the number of vector units (that is, when going from Vector1 to Vector2). This is because they have vector regions similar to the *motion estimation* example explained in section 3.3, with very short vector lengths and small loops. Examples of this include the *form component prediction* and the *add block* regions in the MPEG2 decoder and the *calculation of the long term parameters* in the GSM encoder. On the contrary, other benchmarks shuch as the JPEG encoder and decoder, whose vector regions are characterized by larger vector lengths (ex. *color conversions* or *upsampling*) and/or larger loop sizes (ex. *DCT*'s), exhibit a significant improvement in performance when the number of vector units is doubled.

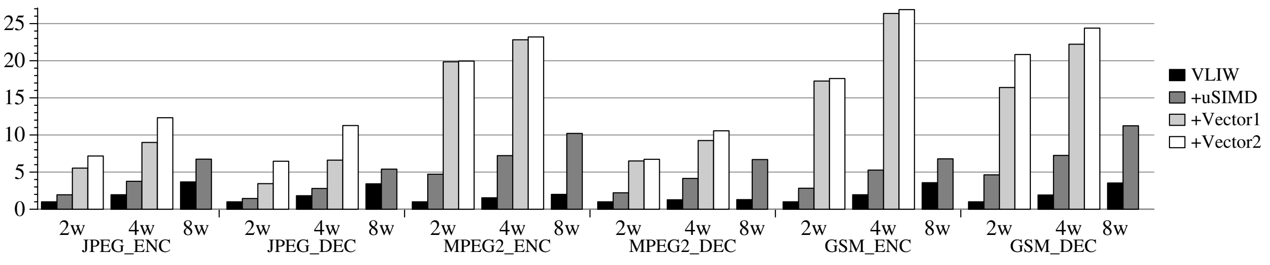
Figure 5.b shows the speed-up of the vector regions again, but with the simulation of the memory hierarchy. We observe that the Vector architectures exhibit the highest performance degradations when considering a realistic memory system. This fact may seem counterintuitive, since vector architectures are well known for their capability to tolerate memory latency. Two reasons explain this behavior. First, the vector lengths are not long enough to take benefit of this characteristic. Second, VLIW architectures are very sensitive to non-deterministic latencies.

As it was explained before, during the scheduling, the compiler assumes that all vector accesses have a stride of one, and the processor stalls at run-time if this assertion is not true. That is what happens in the *mpeg2_enc* benchmark, in which the stride of the main region (the *motion estimation*) is the image width. Moreover, in this kernel, these memory operations represent an important fraction of the overall code, resulting in a high performance degradation (close to 200%). Apart from this, all benchmarks exhibit high hit ratios and very low performance degradation when considering realistic memory.

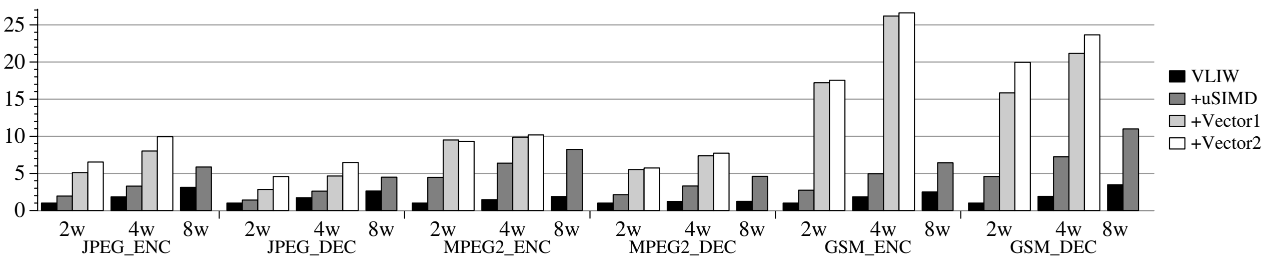
5.2 Speed-up in Complete Applications

Figure 6 shows the speed-up for complete applications. As it was to be expected, the benchmark that exhibits the highest performance improvement is the *mpeg2_enc* (up to 4.74X speed-up for the 4-issue Vector2). Even though there are other benchmarks (such as *gsm_enc*) with similar (or even greater) speed-ups in the vector regions, the impact in the overall performance is not so significant, due to the low vectorization percentage. The 4-issue Vector2 architecture slightly outperforms the 8-issue μSIMD in all the applications (1.03X in average). Note also that the 4-issue Vector1 configuration achieves, in average, the same performance than the 8-issue μSIMD, with only one port to the first level cache and two vector units.

The gap between the different architectures decrease with the issue width of the machine. For example, while the 2-issue Vector2 exhibits a factor of 1.22X of performance improvement (in average) over the 2-issue μSIMD, the 4-issue Vector2 only outperforms the 4-issue μSIMD in a 1.14X. That makes sense, as a wide enough μSIMD-VLIW architecture is able to exploit as ILP the parallelism



(a) Perfect memory



(b) Realistic memory

Figure 5. Speed-up in vector regions

that the Vector-μSIMD-VLIW exploits as DLP.

On the other hand, the vector regions only represent a 40% of the total execution time in the 2-issue VLIW architecture. When most of the available DLP parallelism is exploited via multimedia extensions, the remaining scalar part becomes the bottleneck. In the 4-issue Vector2 architecture, the vector cycles represent less than 10% of the overall execution time (except for the *mpeg2_enc*). By the Amhdal Law, further improvements in the execution of the vector regions would be imperceptible in the complete application.

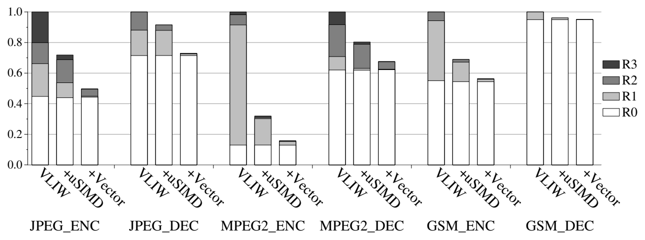


Figure 7. Normalized operation count

5.3 Operations per Cycle

Figure 7 shows the dynamic operation count normalized by the dynamic operation count of the base VLIW architecture. We have distinguished the contribution of each region. Regions from $R1$ to $R3$ are the fractions of code that have been vectorized in the μSIMD and Vector versions in the same order they are listed in Table 1 (for example, in *mpeg2_enc*, $R1$ accounts for the motion estimation and $R2$ and $R3$ for the forward and inverse two dimensional DCT). Region $R0$ always refers to the remaining scalar part.

The results confirm that the μSIMD and Vector-μSIMD versions of code require to execute much less operations than the scalar versions. This may not seem so obvious if we take into account that these versions are sometimes based on algorithms that require to execute much more operations [33].

As can be observed, the Vector architecture executes an average of 84% fewer operations in the vector regions than the μSIMD (19% fewer in the complete application). The obvious reason is that Vector architectures can pack more micro-operations into a single operation (up to 81.10 for the *jpeg_dec* application and 38.78 in average). Moreover, there is an additional reduction on the number of operations involved in the loop-related control. This reduction in the number of operations to fetch and decode also translates into a decrease in power consumption.

Table 3 shows the average number of operations per cycle for the scalar and vector regions of code separately. It confirms our belief that the non-vector regions of code do not benefit from scaling the width of the machine above 4 issue width. Fetching 1.84 operations per cycle does not pay off the hardware complexity of a 8-issue width architecture. For the μSIMD and Vector-μSIMD versions we also show the average number of micro-operations executed per cycle. The Vector-μSIMD ISA obtains the highest speed-ups by exploiting more data parallelism in the vector regions (up to 14.00 micro-operations per cycle) and with the low-

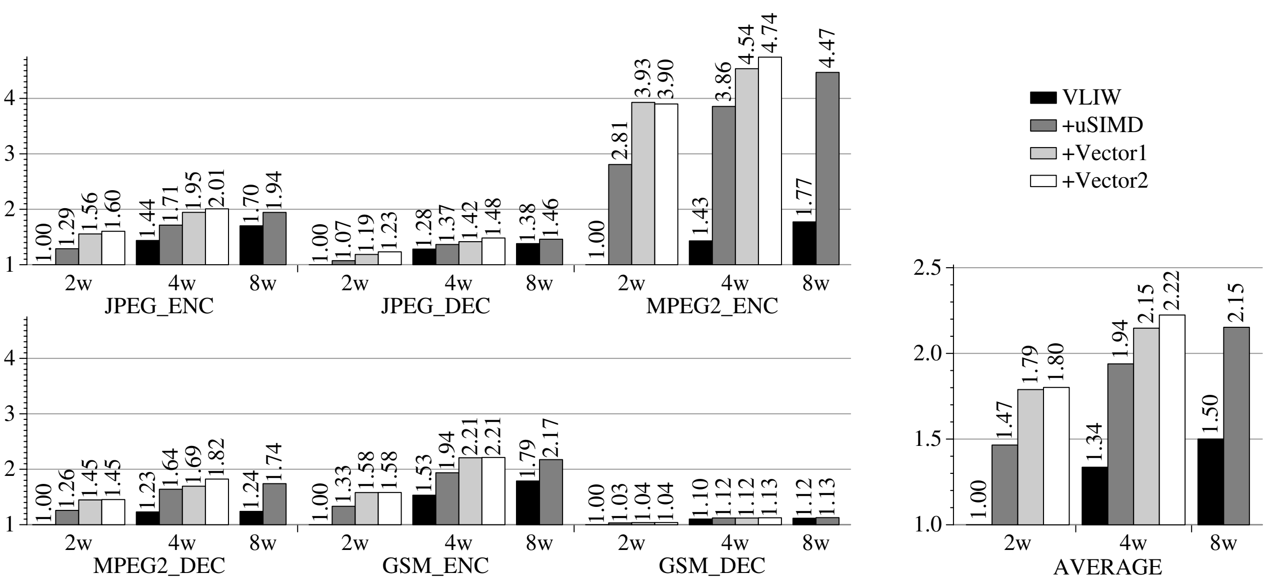


Figure 6. Speed-up in complete applications

est fetch bandwidth requirements (just 1.37 operations per cycle), making it an ideal candidate for embedded systems, where high issue rates are not an option. However, for wide issues, the μSIMD ISA exhibits more flexibility to benefit from wide static scheduling and also reaches significant micro-operations per cycle rates, but at a higher cost.

Table 3. *OPC* **= operations per cycle,** μOPC **= micro-operations per cycle,** *SP* **= speed-up**

	Scalar regs		Vector regs			Application		
	OPC	SP	OPC	μOPC	SP	OPC	μOPC	SP
2w VLIW	1.44	1.00	1.80	1.80	1.00	1.59	1.59	1.00
+μSIMD	1.44	1.00	1.78	4.68	2.88	1.52	2.32	1.47
+Vector1	1.44	1.00	0.87	7.91	9.33	1.36	2.12	1.79
+Vector2	1.44	1.00	0.98	10.10	10.61	1.37	2.15	1.80
4w VLIW	1.77	1.24	3.03	3.03	1.66	2.14	2.14	1.34
+μSIMD	1.78	1.24	2.95	7.80	4.62	1.98	3.05	1.94
+Vector1	1.71	1.20	1.24	11.64	12.87	1.63	2.55	2.15
+Vector2	1.76	1.23	1.37	14.00	14.09	1.69	2.64	2.22
8w VLIW	1.84	1.28	4.54	4.54	2.47	2.42	2.42	1.50
+μSIMD	1.84	1.29	4.47	12.07	6.76	2.18	3.38	2.15

6 Conclusions

The actual performance achieved by very wide issue VLIW architectures is very far from the theoretical peak performance and do not pay off the related hardware complexity. By analyzing the scalability of the scalar and vector regions of code separately, we have shown that the scalar regions do not benefit from increasing the width of the machine above 4-issue width. On the other hand, the kind of parallelism found in the vector regions could be exploited in a more efficient way by means of SIMD execution.

To exploit the data parallelism inherent in the vector regions, we have proposed the addition of one or more vector units together with a vector register filer and a wide port to the L2 that provides the bandwidth required by the vector regions. This extension can be viewed as a conventional short vector ISA where each element is operated in a MMX-like fashion. This enhancement has a minimal impact on the VLIW core and provides high performance in the vector regions for low issue rates.

We have evaluated the proposed architecture for complete applications of audio, video and image processing and compared it againts a VLIW architecture with and without μSIMD extensions. In the vector regions, a 4-issue width Vector-μSIMD-VLIW architecture outperforms the 8-issue μSIMD-VLIW architecture in a factor of up to 4.2X (2.1X in average). Furthermore, a 4-issue architecture with only one port to the first level cache and two vector units achieves, in complete applications, similar performance to that of the 8-issue μSIMD-VLIW.

On the other hand, it has been seen that Vector-μSIMD-VLIW architectures do not perform well in front of non stride-one memory references and exhibit the highest performance degradations when considering a realistic memory system, mainly due to the high sensitivity of VLIW architectures to non-deterministic latencies. Future research must be done to improve the memory hierarchy and to test more flexible scheduling techniques.

References

[1] S. Aditya, V. Kathail, and B. R. Rau. Elcor's machine description system: Version 3.0. Technical Report HPL-98-128, Information Technology Center, 1998.

[2] K. Asanovic, J. Beck, B. Irissou, B. Kingsbury, N. Morgan, and J. Wawrzynek. The t0 vector microprocessor. In *Hot Chips VII*, pages 187–196, August 1995.

[3] V. Bongiorno and G. Shorrel. Cray sv1, sv1e, sv1ex – overview, 2000. http://www.cray.com/products/systems/.

[4] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A vliw architecture for a trace scheduling compiler. *IEEE Trans. on Computers*, C-37(8):967–979, August 1988.

[5] J. Corbal. *N-Dimensional Vector Instruction Set Architectures for Multimedia Applications*. PhD thesis, UPC, Departament d'Arquitectura de Computadors, 2002.

[6] J. Corbal, R. Espasa, and M. Valero. Exploiting a new level of dlp in multimedia applications. In *Proceedings of the 32nd Int. Symp. on Microarchitecture*, pages 72–79, 1999.

[7] K. Diefendorff and P. Dubey. How multimedia workloads will change processor design. *IEEE Computer*, 30(9):43–45, Sept 1997.

[8] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *Proc. of the 27th Int. Symp. on Computer Architecture 2000*, pages 203–213, June 2000.

[9] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. on Computers*, C-30:478–490, July 1981.

[10] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. on Computing*, C–21(9):948–960, 1972.

[11] D. M. Gallagher. *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation*. PhD thesis, University of Illinois, 1995.

[12] L. Gwennap. Majc gives vliw a new twist. *Microprocessor Report*, 13(12):12–15, September 1999.

[13] R. M. Hord. *The Illiac IV, the first supercomputer*. Computer Science Press, 1982.

[14] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *Supercomputing*, 7:229–248, 1993.

[15] Pentium iii processor: Developer's manual. Technical report, Intel, 1999.

[16] Intel. Intel itanium2 processor reference manual for software development and optimization, 2004. http://developer.intel.com/design/itanium2/manuals/.

[17] M. Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[18] B. Juurlink, S. Vassiliadis, D. Tcheressiz, and H. A. Wijshoff. Implementation and evaluation of the complex streamed instruction set. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 73–82, September 2001.

[19] V. Kathail, M. Schlansker, and B. R. Rau. Hpl-pd architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett–Packard Lab., 2000.

[20] C. Kozyrakis. A media-enhanced vector architecture for embedded memory systems. Technical Report CSD-99-1059, UCB, 27, 1999.

[21] H. P. Lab., R.-I. Group, and I. Group. Trimaran user manual, 1998. http://www.trimaran.org/docs.html.

[22] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Int. Symp. on Microarchitecture*, pages 330–335, 1997.

[23] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Int. Symp. on Microarchitecture*, pages 45–54, Dec. 1992.

[24] H. Nguyen and L. K. John. Exploiting SIMD parallelism in DSP and multimedia algorithms using the altivec technology. In *Proceedings of the International Conference on Supercomputing*, pages 11–20, 1999.

[25] A. Peleg and U. Weiser. Mmx technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, 1996.

[26] Trimedia tm-1300. http://www-us3.semiconductors.com/.

[27] F. Quintana, J. Corbal, R. Espasa, and M. Valero. Adding a vector unit on a superscalar processor. In *Proc. of the International Conference on Supercomputing*, pages 1–10, 1999.

[28] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: history, overview, and perspective. *The Journal of Supercomputing*, 7(1-2):9–50, 1993.

[29] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The cydra 5 departmental supercomputer. *IEEE Computer*, 22(1):12–35, January 1989.

[30] S. F. Reddaway. Dap-a distributed array processor. In *Proceedings of the 1st annual symposium on Computer architecture*, pages 61–65. ACM Press, 1973.

[31] R. Russel. The cray-1 computer system. *Comunications of the ACM*, 21(1):63–72, January 1978.

[32] E. Salamí, J. Corbal, C. Alvarez, and M. Valero. Cost effective memory disambiguation for multimedia codes. In *Proc. of the Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 117–126, 2002.

[33] E. Salamí, J. Corbal, R. Espasa, and M. Valero. An evaluation of different dlp alternatives for the embedded media domain. In *Proceedings of the 1st Workshop on Media Processors and DSPs*, pages 100–109, November 1999.

[34] E. Salamí and M. Valero. Initial evaluation of multimedia extensions on vliw architectures. In *Proceedings of the 4th international workshop on Systems, Architectures, Modeling, and Simulation*, pages 403–412, July 2004.

[35] M. S. Schlansker and B. Raw. Epic: Explicitly parallel instruction computing. In *IEEE Computer*, pages 37–45, February 2000.

[36] H. Sharangpani and K. Aurora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, September 2000.

[37] TI. TMS320C62XX family, 1999. http://www.ti.com/sc/-docs/products/dsp/tms320c6201.html.

[38] Introducing tigersharc, 1999. http://www.analog.com/new/-ads/html/SHARC2.

[39] A. van der Steen and J. Dongarra. The nec sx-5, 2001. http://www.top500.org/ORSC/2001.

[40] A. Yu. The future of microprocessors. *IEEE Micro*, 16(6):46–53, 1996.