On the Scalability of 1- and 2-Dimensional SIMD Extensions for Multimedia Applications

Friman Sánchez, Mauricio Alvarez, Esther Salamí, Alex Ramirez, Mateo Valero
Computer Architecture Department
Universitat Politecnica de Catalunya
Barcelona, Spain
{fsanchez, alvarez, esalami, aramirez, mateo}@ac.upc.es

Abstract

SIMD extensions are the most common technique used in current processors for multimedia computing. In order to obtain more performance for emerging applications SIMD extensions need to be scaled. In this paper we perform a scalability analysis of SIMD extensions for multimedia applications. Scaling a 1-dimensional extension, like Intel MMX, was compared to scaling a 2-dimensional (matrix) extension. Evaluations have demonstrated that the 2-d architecture is able to use more parallel hardware than the 1-d extension. Speed-ups over a 2-way superscalar processor with MMX-like extension go up to 4X for kernels and up to 3.3X for complete applications and the matrix architecture can deliver, in some cases, more performance with simpler processor configurations. The experiments also show that the scaled matrix architecture is reaching the limits of the DLP available in the internal loops of common multimedia kernels.

I. Introduction

Multimedia applications have become one of the most important workloads, both in general purpose and embedded application domains. Devices like set-top boxes, DVD and MP3 players, mobile phones and many others demand high computational power in order to support applications like digital video decoding, digital image processing or speech compression and decompression, that need to be done sometimes in real time and under several power constraints. [1].

In microprocessors for desktop computers, the common approach for dealing with the requirements of multimedia applications has been the extension of the Instruction Set Architecture with SIMD instructions tailored to the operations and data types present in multimedia kernels. Extensions like Intel's MMX and SSE, Sun's VIS, HP's MAX, MIPS's MDMX and Motorola's Altivec exploit the data level parallelism (DLP) available in multimedia applications by performing multiple sub-word operations in a single instruction [2].

In the embedded domain, ASICs, media processors, high performance DSPs or some combinations between them are the most common solutions for dealing with the computational and power requirements of multimedia and communications applications. But the changing nature of multimedia and communication protocols suggests the use of programmable processors instead of custom ASICs or highly specialized application specific processors. In this way DSPs manufacturers are including SIMD extensions to their base architectures. VLIWs DSPs like Analog Devices Tiger Sharc, Motorola and Lucent's StarCore and Texas Instruments TMS320C6xx and superscalar-based DSPs like TriCore also include SIMD capabilities [3].

On the other hand, some research proposals suggest the use of vector processors [4], [5] as an effective way of exploiting DLP present in multimedia applications. An alternative approach comes from the combination of vector registers and sub-word computation in such a way that registers can be seen as matrices [6]. This matrix architecture, referred as MOM (Matrix Oriented Multimedia), can be seen as a 2-dimensional SIMD extension to a superscalar or VLIW processor.

As multimedia standards become more complex, processors need to scale their SIMD multimedia extensions in order to provide the performance required by new applications. Scaling these extensions not only need to address performance issues, but also power consumption, design complexity and cost, especially for embedded processors.

The focus of this paper is to develop an scalability analysis of a 1-dimensional SIMD extension, like Intel MMX, and a 2-dimensional extension, like MOM. For the two kinds of SIMD extensions an scaling in the width of registers and the number of execution units was performed. We will show that making the matrix registers bigger it is possible to pack almost all the data available in the inner loops of common multimedia kernels with a lower complexity processor than a 1-dimensional SIMD extension. Further extensions in width or length of matrix SIMD registers would not result in significant performance improvements. By the

0-7803-8965-4/05/$20.00 ©2005 IEEE

use of a scaled matrix architecture, the dynamic instruction count is reduced drastically while still allowing the execution of more operations in parallel. The final result is a significant increment of performance in the evaluated multimedia applications, executing fewer instructions and with a complexity-effective hardware organization.

This paper is organized as follows: in chapter 2 an analysis of scalability alternatives is presented. In chapter 3 the applications, simulator and modeled microarchitectures used in the experiments are detailed. In chapter 4 results for both kernels and complete applications are described; and finally in chapter 5 some conclusions are presented.

II. Scaling SIMD Extensions

The amount of parallelism that can be exploited using SIMD extensions is a function of three conditions. The first one is the number of SIMD instructions that can be executed in parallel, which is related with the number of SIMD functional units and the hardware resources necessary for continuous processing multiple SIMD instructions. The second one is the number of subwords that can be packed into a word, which depends on the size of registers. Packing more data into a single register allows to perform more operations in parallel for each instruction. The last one is the presence of combined instructions that allow the execution of different types of instructions (integer, floating-point, SIMD) concurrently; this condition depends on the application and the code generated by the compiler [7].

The first two techniques are related with microarchitecture and architecture features that can be modified to scale SIMD extensions. Next we are going to analyze the requirements and possibilities of implementing these techniques for scaling 1- and 2-dimensional SIMD extensions.

A. Scaling 1-Dimensional SIMD Extensions

The first approach for scaling SIMD extensions consist of adding execution units to the SIMD pipeline. The advantage of this approach is that it could improve the performance at no programming cost. But, adding more functional units to the pipeline implies an increasing in register file ports, execution hardware and scheduling complexity [8]. Such modifications could have a big impact in area, timing, power and complexity of the processor. But even if an aggressive processor with many SIMD functional units could be developed, performance gains could not be as good as expected. Recent studies [9], [10] have shown that there are some bottlenecks in the microarchitecture that do not allow to obtain better performance by scaling the SIMD resources. These bottlenecks are related with overhead and supporting instructions necessary for address arithmetic, data transformation (packing, unpacking, transposing), access overhead and limitations in the issue width.

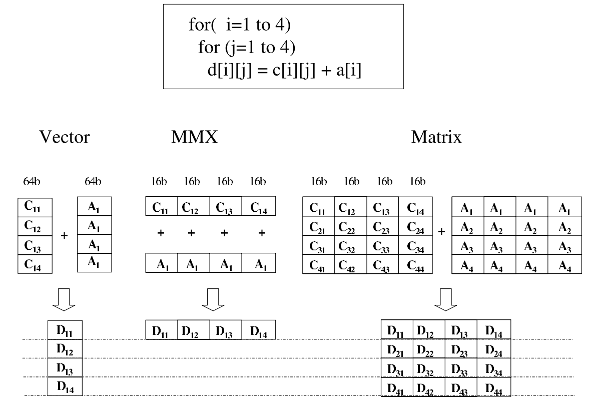
The other way of scaling is to increase the width of SIMD registers, i.e. from current 128-bit registers (in SSE2 and Altivec) to 256-bit, 512-bit or more. However, this option has two main disadvantages. The first one is the hardware implementation cost, that can be significant, taken into account the required increase in the width of interconnect buses, the doubling of execution units, and, more important, the increase in the memory bandwidth necessary to provide enough data to larger SIMD registers [8]. Even if such a processor could be implemented, having 256-bit or 512-bit registers could only be useful if applications have memory data patterns that match the match the hardware organization; that is, have enough operands arranged in memory to fill the SIMD registers. This can be true for some applications, but the majority of image, video an audio applications have small arrays or matrices sometimes non-contiguous in memory. For example the JPEG and MPEG standards define 8×8 or 16×16 pixel blocks. For this kind of applications, making registers bigger than the basic data structures may incur a big overhead for taking data from memory and/or storing back the results [11].

From the previous discussion we can conclude that an scalable SIMD extension need to provide some features in the ISA and in the microarchitecture that allow the exploitation of DLP taken into account the data patterns present in multimedia applications without increasing the complexity of critical structures in the processor

As a part of this study, the benefits and limitations of scaling a MMX-like extension with 64-bit registers (referred here as MMX64) to a 128-bit extension like Intel SSE2 (referred as MMX128) are evaluated.

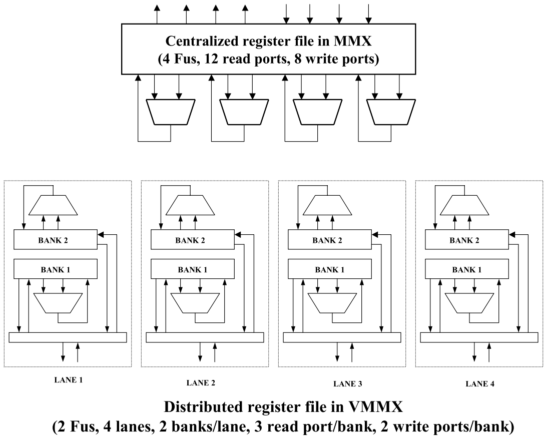
B. Scaling 2-dimensional Extensions

MOM is a matrix-oriented ISA paradigm for multimedia applications based on fusing conventional vector ISAs with SIMD ISAs, such as MMX. MOM is a suitable alternative for the multimedia domain due to its efficiently handling of the small matrix structures typically found in most multimedia kernels [6]. Figure 1 describes how vector, MMX-like, and MOM ISAs exploit DLP using a very simple code example taken from a multimedia kernel. Vector processors, Figure 1(a), perform multiple operations by using long vectors; MMX ISAs, Figure 1(b), perform multiple sub-word operations within a single register; and



```
for( i=1 to 4)
  for (j=1 to 4)
    d[i][j] = c[i][j] + a[i]
```

Fig. 1. Comparison between (a) conventional vector, (b) MMX-like and (c) matrix oriented ISAs



Centralized register file in MMX
(4 Fus, 12 read ports, 8 write ports)
BANK 2
BANK 1
LANE 1
LANE 2
LANE 3
LANE 4
Distributed register file in VMMX
(2 Fus, 4 lanes, 2 banks/lane, 3 read port/bank, 2 write ports/bank)

Fig. 2. Register File and Datapath Comparison.

MOM, Figure 1(c), combines both approaches using vector registers with subword capabilities.

As with 1-dimensional SIMD extensions, a 2-dimensional architecture, like MOM, can be scaled in the width of registers and the number of execution units. Additionally a vector architecture can be scaled in the number of parallel lanes and the maximum vector length.

The original MOM architecture provides the programmer with 16 matrix registers, each one holding sixteen 64-bit words [12]. In this work we study how the vector register file in MOM architecture can scale from 64-bit width to 128-bit, adding to the original proposal more capacity and instructions. A 128-bit matrix register can hold an 8×8 16-bit matrix or a 16×16 8-bit matrix. Like other vector architectures, MOM vector load and store operations supports two basic access methods: unit stride and strided [4]. By using a strided access to memory, a matrix register can be filled with data that it is not adjacent in memory and with almost zero overhead for looping and address calculation. In this way with the strided access using vector registers is possible to overcome part of the overhead associated with reorganization instructions of SIMD extensions. This instructions represent a significant part of the SIMD version of common image and video applications, in which full images or frames are divided into small blocks that are non contiguous in memory.

Multimedia applications on vector architectures are characterized by having small vector lengths [5], [13], for that reason the maximum vector length of MOM architecture is not going to be increased. As we will show in section IV, matrix registers of 128-bit with a maximum vector length of sixteen adapts well for most common multimedia applications. Scaling the number of processing units is different in vector architectures than in SIMD extensions like MMX. MOM register file and datapath are organized in a distributed way in which the total register file is partitioned into banks across lanes, and these banks are connected only to certain functional units in their corresponding vector lane. Figure 2 shows an specific organization of two functional units divided into four vector lanes in which the register file is divided into two banks per lane. With the figure it is possible to compare the distributed vector register file to a centralized SIMD register file used in MMX. The distributed organization of the datapath in MOM provides an effective mechanism to scale performance. By adding more parallel lanes MOM can execute more operations of a vector instruction each cycle without increasing the complexity of the register file. This can be obtained by dividing the register file inside a lane into sub-banks. The limit for including more lanes is the vector length that can be achieved in the vectorization of multimedia applications [6].

Additional to a bigger register file, the scaled MOM architecture includes new instructions to support partial data movement between registers and memory. These instructions are necessary for applications which data patterns do not fill well in 128-bit matrix-registers and they are similar to those ones that were included by Intel in the SSE2 and SSE3 extensions [14].

For simplification purposes, we are going to refer to MOM architecture with 64-bit registers as VMMX64 (64-bit VectorMMX) and to MOM architecture with 128-bit registers as VMMX128 (128-bit Vector-MMX).

C. Hardware cost of scaling

When scaling SIMD extensions, the register file storage and communication between arithmetic units become critical factors, dominating in area, cycle time

Processor configuration	4WAY				8WAY			
	mmx64	mmx128	vmmx64	vmmx128	mmx64	mmx128	vmmx64	vmmx128
Logical registers	32	32	16	16	32	32	16	16
Physical registers	64	64	36	36	96	96	64	64
Lanes	1	1	4	4	1	1	4	4
Banks per Lane	1	1	2	2	1	1	4	4
Read ports per Bank	3	12	12	3	3	24	24	3
Write ports per Bank	8	8	2	2	16	16	2	2
Overall RF storage, KB	0.5	1.0	4.6	9.12	0.77	1.54	8.19	16.3
RF Area (Normalized to mmx64 versions)	1	2.00X	1.41X	2.63X	5.14X	10.29X	2.10X	4.20X

TABLE I

SCALING REGISTER FILES FOR SIMD EXTENSIONS

and power dissipation of the processor [15]. Table I resumes the parameters of capacity, complexity and area of the registers files for a 4-way and 8-way superscalar processors with 4 different SIMD extensions. Area estimations are relative to the MMX64 configuration.

Register file area has been estimated assuming a 0.18 μ CMOS process technology based on the models described in [15]. It is very important to note that these models are just approximative and useful to give upper bounds and determine trends, but cannot be translated directly to reality, because several full custom VLSI optimizations could be done.

As Table I shows, the VMMX configurations have more capacity in the register file and can support more functional units that the MMX ones. This resource capability is reflected in terms of the necessary area for implementing VMMX extensions. The approach followed here is similar to the Altivec extension to the PowerPC architecture [16], in which a considerable silicon area is invested in the implementation of the SIMD extension, but in such a way that processor cycle and complexity are not affected. By using a vector organization in parallel lanes, these objectives can be achieved when scaling the VMMX extension [5].

On the other hand, when MMX64 extension is scaled to MMX128, the register file complexity becomes a predominant factor in terms of area and cycle time. Table I shows that the ratio of area increase are lower in VMMX than in MMX, so that for a 8-way processor configuration the VMMX128 register file has less area cost with less complexity than the MMX128.

The VMMX and MMX pipelines are not balanced in terms of functional units and register file capacity, but what we want to argue is that the VMMX processors have these bigger resources because they can map effectively the hardware structure to the DLP available in common multimedia applications without a significant increment in complexity. In the VMMX configuration the number of lanes and functional units can be adjusted in order to fulfill different design constraints in terms of power and area without compromising the binary compatibility or the complexity of the register file.

D. A case of study: motion estimation

Figure II-D shows different versions of a fragment of code taken from the motion estimation routine that is part of the MPEG-2 encoder application. Code is taken from function *dist1* that computes the Sum of Absolute Differences (*SAD*) between two blocks of *h*x16 pixels (*h* is typically 8 or 16) pointed by *p1* and *p2*. There is a stride *lx* between rows.

Figure II-D(a) shows the scalar version: there are two nested loops, one for intra row elements (i) and the other one for the different rows (j). In the MMX versions the inner loop is eliminated. The MMX64 version, illustrated in Figure II-D(b), operates over arrays of 1×8 pixels, being necessary to divide the data array in two regions. For each of these regions, it is necessary to use and update pointers and to accumulate the partial results of each sub-block in one register. The MMX128 version, shown in Figure II-D(d), operates over 1×16 pixels arrays that are contiguous in memory, allowing a single load to be performed for each row, and requiring less pointer overhead than the 64-bit version.

In the VMMX versions, both loops can be eliminated because it is possible to pack the two dimensional array into vector registers. For loading the data into vector registers, it is necessary to define the vector length (VL=*h*), and for each load, it is necessary to specify, as a part of load instruction, the vector stride *lx*. In the VMMX64 version, shown in Figure II-D(c), it is necessary to divide the array into 2 blocks of *h*x8 pixels, thus being necessary two vector registers to store the data array. Finally, in VMMX128, as shown in Figure II-D(e), it is possible to pack all the pixel array in a single vector register, reducing drastically the number of instructions used. Specially a lot of overhead instructions used for looping and address calculation are eliminated, and *SAD* is implemented using a packed accumulator that allows a parallel execution

```
for (j=0; j<h; j++){
  for (i=0; i<16; i++){
    if ((v = p1[i] − p2[i]) < 0)
      v= −v;
    s+= v;
  }
  p1+= lx;
  p2+= lx;
}
```

(a) scalar

```
for(j=0; j<h; j++){
  VR1 = MEM[p1];
  VR2 = MEM[p2];
  VR1 = VR1 >> 1;
  VR2 = VR2 >> 1;
  VR3 = MEM[p1+8]; p1 += lx;
  VR1 = VR1 − VR2;
  VR4 = MEM[p2+8]; p2 += lx;
  VR1 = Sum(|VR1|);
  VR3 = VR3 >> 1;
  VR4 = VR4 >> 1;
  VR15 += VR1;
  VR3 = VR3 − VR4;
  VR3 = Sum(|VR3|);
  VR15 += VR3;
}
s = VR15;
s = s<<1;
```

(b) mmx64

```
ACC1 = 0;
ACC2 = 0;
VL   = h;
R2   = lx;
VR1  = MEM[p1]    (Vs=R2);
VR2  = MEM[p2]    (Vs=R2);
ACC1 = Sum(|VR1 − VR2|);
VR3  = MEM[p1+8]  (Vs=R2);
VR4  = MEM[p2+8]  (Vs=R2);
ACC2 = Sum(|VR3 − VR4|);
R5   = Sum(ACC1);
R6   = Sum(ACC2);
R5   = R5 + R6;
s    = R5;
```

(c) vmmx64

```
for(j=0; j<h; j++){
  VR1 = MEM[p1];   p1 += lx;
  VR2 = MEM[p2];   p2 += lx;
  VR1 = VR1 >> 1;
  VR2 = VR2 >> 1;
  VR1 = VR1 − VR2;
  VR1 = Sum(|(VR1|);
  VR15 += VR1;
}
s = VR15;
s = s<<1;
```

(d) mmx128

```
ACC1 = 0;
V1   = h;
VR1  = MEM[p1]  (Vs=lx);
VR2  = MEM[p2]  (Vs=lx);
ACC1 = Sum(|VR1 − VR2|);
R5   = Sum(ACC1);
s    = R5;
```

(e) vmmx128

Fig. 3. Motion estimation code example

of the operation over the vector registers [17].

III. Experimental Methodology

A. Workload

In order to evaluate the different architectures under study we have selected six applications from the Mediabench suite [18] that are representative of video, still image and voice processing applications. For each application we have selected the most computational intensive kernels with potential DLP and evaluated them in isolation. Table II describe the kernels and benchmarks and their characteristics.

B. Simulation Framework

Emulation libraries containing the multimedia instructions have been used for the evaluated extensions: MMX64, MMX128, VMMX64 and VMMX128. Most of the functionality of MMX and SSE ISAs have been implemented into the MMX64 and MMX128 emulation libraries respectively, although it is important to note that the modeled extensions use more logical registers and they are based on the Alpha ISA, not on the IA32. Using these emulation libraries versions of the mentioned kernels were developed. To maximize performance, optimization techniques like loop-unrolling and software pipelining were applied. All codes have been compiled using gcc 2.95.2 with

Application	Description	Kernel	Description	Data size
jpegenc	JPEG still image encoder	*rgb*	RGB to YCC color conversion	RGB triads
		fdct	Forward Discrete Cosine Transform	8 × 8 16-bit
jpegdec	JPEG still image decoder	*h2v2*	Image up-sampling	Image width
		ycc	YCC to RGB color conversion	(Y,Cb,Cr)× Image_width 8-bit
mpeg2enc	MPEG2 video encoder	*motion1*	Sum of Absolute Differences	16 × 16 8-bit
		motion2	Sum of Quadratic Differences	16 × 16 8-bit
		idct	Inverse Discrete Cosine Transform	8 × 8 16-bit
		fdct	Forward Discrete Cosine Transform	8 × 8 16-bit
mpeg2dec	MPEG2 video decoder	*comp*	Motion compensation	8 × 4 8-bit
		addblock	Picture decoding	8 × 8 8-bit
		idct	Inverse Discrete Cosine Transform	8 × 8 16-bit
		fdct	Forward Discrete Cosine Transform	8 × 8 16-bit
gsmenc	GSM 06.10 speech encoder	*ltppar*	Parameter calculation for LTP filtering	40 16-bit
gsmdec	GSM 06.10 speech decoder	*ltpfilt*	Long term parameter filtering	120 16-bit

TABLE II

BENCHMARK SET DESCRIPTION

Parameter	MMX 2/4/8 way	VMMX 2/4/8 way
Physical SIMD registers	40/64/96	20/36/64
Fetch, Decode, Grad.	2/4/8	2/4/8
Integer FUs	2/4/8	2/4/8
FP FUs	1/2/4	1/2/4
SIMD issue	2/4/8	1/2/3
SIMD FUs	2/4/8	1×4/2×4/3×4
Mem FUs (L1 ports)	1/2/4 (x64b)	1/1/2 (x64b)
L2 ports	-	1x(64b/128b/256b)

TABLE III

MODELED PROCESSORS

	L1	L2
size	32KB	512KB
number of ports	1/2/4	1
port width (bytes)	8	16/32/64
number of banks	8	2
sets per bank	32	2048
associativity	4	2
line size (bytes)	32	128
latency	3	12
Main Memory Latency (cycles)	500	

TABLE IV

MEMORY HIERARCHY CONFIGURATION

the -O2 flag.

The simulation tool used in this work was an improved version of Jinks Simulator [19], that is a parametrizable simulator targeted at evaluating out-of-order superscalar architectures with a special focus on vector extensions. A combination of trace-driven and execution-driven approaches based on ATOM [20] were used for generating the input trace stream for the simulator.

C. Processor Models

The baseline processor is a 2-way out-of-order superscalar core similar to MIPS R10000 [21] with the addition of a MMX64 SIMD extension. We have evaluated four different configurations that include MMX and VMMX approaches for 64 and 128-bit registers:

- 2/4/8-way superscalar processor + MMX64
- 2/4/8-way superscalar processor + MMX128
- 2/4/8-way superscalar processor + VMMX64
- 2/4/8-way superscalar processor + VMMX128

Table III shows the processor configurations used for the simulations. The 8-way superscalar processors are too aggressive configuration that are obviously not suitable for embedded systems and are nowadays unfeasible in a high performance general purpose processor at current clock frequencies, but they can be used as a guide of the potential performance that could be obtained (and complexity problems that could be found) when scaling processor resources.

D. Memory hierarchy model

A detailed memory hierarchy model with two levels of on-chip cache and a Direct RAMBUS main memory system have been included in the simulator. Table IV shows the configuration parameters for caches and main memory. Parameters are similar to those found in some recent microprocessors with multimedia extensions like PowerPC970. For VMMX versions a *vector cache* was used [22]. 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 conventional data cache, while vector accesses bypass the L1 to access directly the L2 vector cache. This bypass is somewhat similar to the bypass implemented in Itanium2 processor for the floating point register file [23]. If the L2 port is $B\times$64-bit wide, these accesses are performed at a maximum rate of B elements per cycle 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.

As shown in Table IV the latency value for the 2 cache levels and the main memory are relative high, this is done because we want to determine the ability of the proposed extensions to tolerate high latencies in the memory subsystem.

IV. Simulation results

A. Kernel speed-up analysis

Figure 4 shows the kernels speed-up for the different multimedia ISAs under study. The baseline is the 2-way superscalar processor with a MMX64 extension. Scaling from MMX64 to MMX128 does not result in great performance increment taking into account that register and functional units are twice the size of the MMX64 ones. The speed-up goes up to 1.47X for *idct*, 1.43X for *ycc*, 1.25X for *addblock* and 1.19X for *h2v2*. These kernels have a regular data pattern and they adapt well to 128-bit wide registers.

VMMX versions of kernels exhibit bigger speed-ups than the MMX ones in all the cases and produce significant speed-ups when going from VMMX64 to VMMX128 versions, except for *ltppar* and *addblock* kernels. The bigger speed-ups (4.10X for *idct*, 2.71X for *ycc*, 2.43X for *motion2* and 2.29X for *motion1*) are due to the better matching between the data organization and the matrix registers structure. The speed-up in *h2v2* is due to the large size of the input data set and the regular data patterns in memory, allowing a vector stride of one and the use of the maximum VL available (16). In *ltppar* and *ltpsfilt*, computation is done over two short segments of data (40 16-bit and 120 16-bit respectively). This limits the parallelism that could be exploited when going from VMMX64 to VMMX128 and is reflected in the small difference of speed-ups. The small speed-up obtained by *comp* and *addblock* in all versions is related with the parallel data available (8x4 pixels in *comp* with a stride of 800), that represents a small fraction of the matrix registers in VMMX64 and incurs in some arithmetic overhead in VMMX128.

In the *idct* VMMX128 version, it is possible to pack the 8x8 16-bit input data set and coefficients in matrix registers and then performing a multiply-accumulate operation between them. *Idct* exhibits the biggest speed-up due to the use of vector registers as a cache. In VMMX versions we use a 2D-matrix algorithm that need to multiply the input matrix and its transpose with the coefficients matrix. In VMMX128, due to the fact that we can store the whole matrices in vector registers, we can maintain the matrix coefficients in a vector register during the two matrix products and perform all the operations inside a vector register and only go to memory to store the final result. This saves a lot of redundant load operations and allows to apply software pipelining over the packed accumulator that would be extremely difficult to implement in a scalar or MMX versions.

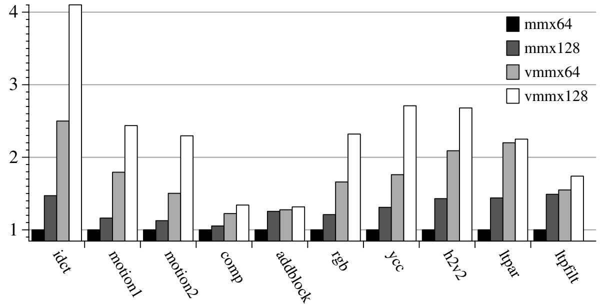


Fig. 4. Kernels speed-up (2-way)

B. Complete applications speed-up

Speed-up of kernels only shows the potential of the evaluated architectures on a highly data parallel code, but these kernels are used in bigger applications where there is a lot of scalar code associated to protocol overhead (MPEG, JPEG) or file manipulation that cannot use the SIMD units. Then, the multimedia extensions not only need to exploit efficiently the data parallelism available but not to degrade the performance of the scalar portion of the application.

Mpeg2enc is the application which takes more benefit from the use of matrix registers. Figure 5 shows that VMMX versions of the application scales better than MMX ones. VMMX128 version has the biggest speed-up due to the good matching of data in the *motion* and *idct* kernels to the 128-bit matrix registers. These kernels account for more than 25% of the total execution time of the MMX-64 version of the application running on the 2-way processor. *Mpeg2dec*, instead, shows a significant speed-up but the difference between MMX and VMMX versions is smaller than in *mpeg2enc*. In this application motion compensation routines are not so much significant of the total execution time and their data parallelism is not so big. Furthermore *mpeg2dec* presents a lot of scalar code in picture decoding that can not be vectorized.

In *jpegdec* application, VMMX versions show a greater capacity to exploit DLP compared with MMX versions. This is due to the fact that *h2v2* and *ycc* kernels operate over data that has a good pattern in memory and the vector length used is high (8 and 16 in both cases). On the other hand, VMMX64 version of *jpegenc* obtains a better performance than MMX versions for less aggressive schemes, that is 2 and

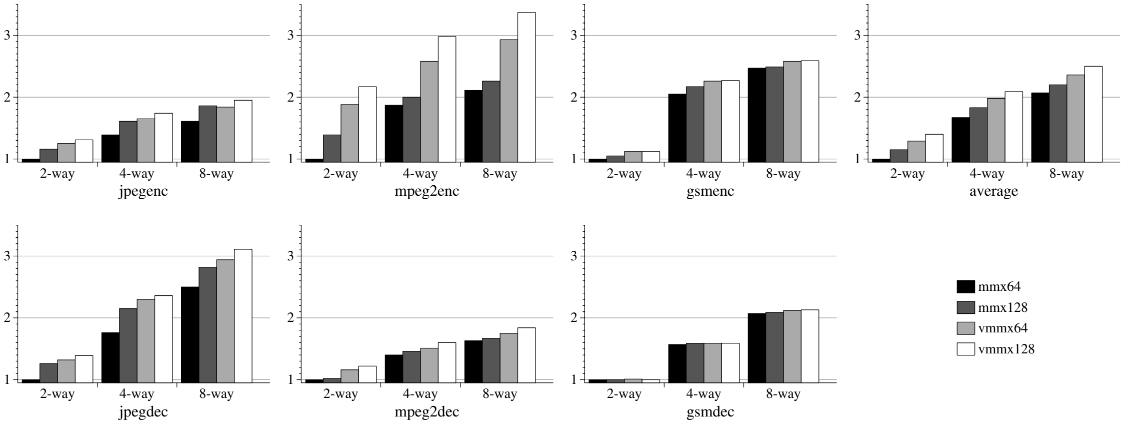


Fig. 5. Full applications speed-up

4-way. However, in 8-way configuration, MMX128 version outperforms the VMMX64 version, this is due to the behavior of *rgb* kernel. The vectorization happens along the color space (red, green and blue) dimension, yield a vector length of only 3. Additionally, the order in which results must be stored in memory does not benefit the VMMX64 version. However, the VMMX128 version overcomes this limitation by allowing to pack more sub-word data into the matrix register.

As showed in figure 5, for *mpeg2enc*, it is possible to obtain a similar performance with a 2-way VMMX128 processor instead of a 8-way MMX128 one. The same behavior can be seen in *jpegenc* and *mpeg2dec* between 4-way VMMX versions and 8-way MMX ones. In those cases, scaling the 2-dimensional register file of a simpler processor is much more effective than scaling the complete resources of a processor with 1-dimensional registers. In *gsmenc* and *gsmdec* applications, the percentage of parallelization is small (less than 10%) and therefore the applications does not benefit so much from SIMD extensions.

C. Cycle Breakdown

Figure 6 shows the dynamic cycle distribution for the *jpegdec* application. The remaining of the benchmarks are not included for room reasons, but they exhibit a similar behavior. The shadow part of each column represents the dynamic cycles used in vector operations, while the white part comes from the scalar code. Results are normalized by the dynamic cycle count of the reference 2-way MMX64 superscalar processor.

As it was expected, scaling the MMX64 extension provides a significant drop in the number of cycles to execute the vector part of code. Scaling in both dimensions (width and length) achieves the maximum reduction: for the 2-way architecture, the VMMX128 extension reduces the execution time of the vector code in a 85% over the MMX64 extension.

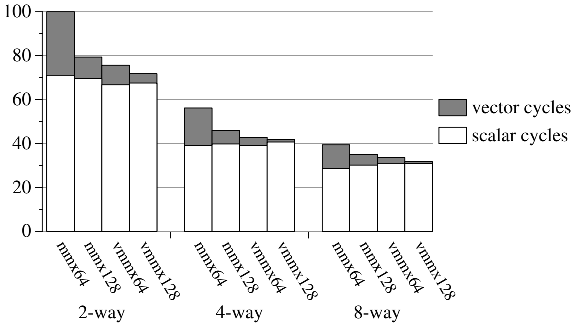


Fig. 6. Cycle count distribution (jpegdec)

However, it can be observed that, when most of the available DLP parallelism is exploited via multimedia extensions, the remaining scalar part of the code becomes the main bottleneck. For the 8-way VMMX128 architecture, the vector cycles represent only the 2.7% of the overall execution time. By the Amhdal Law, further improvements in the execution of the vector region would be imperceptible in the full application.

D. Dynamic Instruction Count

Figure 7 shows the dynamic instruction count for the benchmarks under study. Again, results are normalized by the dynamic instruction count of the MMX64 architecture. The operations have been classified into five categories: scalar memory, scalar arithmetic, control, vector memory and vector arithmetic. We observe that the VMMX architectures execute about 30% fewer instructions than the MMX64, and the MMX128 an average of 15% fewer instructions. This is obviously due to the capability of these extensions to pack more operations into a single instruction.

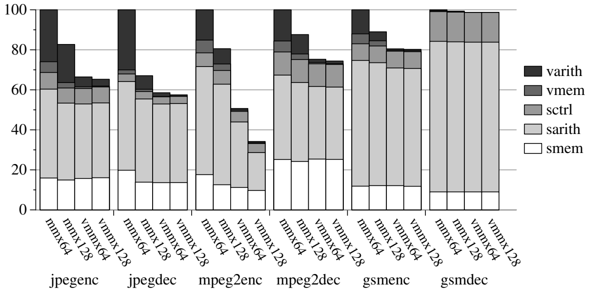


Fig. 7. Dynamic instruction count

As seen in figure 7, the biggest instruction reduction is achieved by the *mpeg2enc* application. This reduction comes from the commented elimination of scalar instructions used for address computation and loop manipulation. In any way, note that the limit of packing data seems to be reached, and scaling further over, either in width or in length, would not provide any noticeable benefit. At this point, a full reorganization of the code is required if we want to expose more parallelism to the processor.

V. Conclusions

In this paper we have presented an scalability analysis of SIMD extensions for multimedia applications. Scaling current 1-dimensional SIMD extensions was compared to scaling a 2-dimensional architecture. The comparison was made using both kernels and complete multimedia applications. Scaling was made in the width of SIMD registers and in processor resources. The matrix architecture with 128-bit registers has shown the best performance improvements compared to a 64-bit matrix architecture and to 1-dimensional (64-bit and 128-bit) SIMD extensions.

The benefits of a Matrix architecture come from the elimination of some of the bottlenecks of current SIMD extensions. Multimedia data structures fit very well into matrix registers, the matrix nature of the ISA eliminates a lot of pointer and loop code overhead, and the combination of vector length and vector stride eliminates the constraints in the contiguous data distribution in memory. In some cases, additional performance improvements can be obtained by the use of matrix registers as a cache for intermediate results, reducing also the pressure on the memory hierarchy.

By applying all of these optimizations, the Matrix architecture is reaching the limits of available DLP in the inner loops of common multimedia applications. Further scaling on the width or length of matrix registers can no deliver significant performance improvements because the execution time is now dominated by the scalar portion of the code. Extracting more parallelism in the analyzed applications requires special code transformations in order to execute multiple instances of the optimized functions in parallel or dedicated hardware to extract parallelism beyond the inner loops.

It was demonstrated also that, for video and image applications, a simple processor with a VMMX128 extension can delivery more performance than a processor with a MMX extension and more resources. This feature and the reduced complexity in some critical structures of the matrix pipeline, like the register file, makes the matrix enhanced processor a suitable choice for embedded applications.

Acknowledgment

This work has been supported by the Ministry of Science and Technology of Spain, the European Union (FEDER funds) under contract TIC2001-0995-C02-01 and TIC2004-07739-C02-01, by the HiPEAC European Network of Excellence, and by IBM. We acknowledge the European Center for Parallelism of Barcelona (CEPBA) for supplying the computing resources for our research.

References

[1] K. Diefendorff and P. Dubey, “How multimedia workloads will change processor design,” *IEEE Micro*, vol. 30, no. 9, pp. 43–45, September 1997.

[2] N. T. Slingerland and A. J. Smith, “Multimedia instruction sets for general pupose microprocessors: A survey,” UCB, Tech. Rep. CSD-00-1124, December 1999.

[3] I. Kuroda and T. Nishitani, “Multimedia processors,” *Proceedings of the IEEE*, vol. 86, no. 6, pp. 1203–1221, June 1998.

[4] K. Asanovic, J. Beck, B. Irissou, B. Kingsbury, N. Morgan, and J. Wawrzynek, “The t0 vector microprocessor,” in *Hot Chips VII*, August 1995, pp. 187–196.

[5] C. Kozyrakis and D. Patterson, “Scalable vector processors for embedded systems,” *IEEE Micro*, vol. 23, no. 6, pp. 36–45, Nov–Dec 2003.

[6] J. Corbal, R. Espasa, and M. Valero, “Exploiting a new level of dlp in multimedia applications,” in *32nd international symposium on Microarchitecture*, 1999, pp. 72–79.

[7] R. B. Lee, “Subword parallelism with max-2,” *IEEE Micro*, vol. 16, no. 4, pp. 51–59, August 1996.

[8] S. Thakkar and T. Huff, “The internet streaming simd extensions,” *IEEE Computer*, vol. 32, no. 12, pp. 26–34, December 1999.

[9] D. Talla, L. K. John, and D. Burger, “Bottlenecks in multimedia processing with simd style extensions and architectural enhancements,” *IEEE Transactions on Computers*, vol. 52, no. 8, pp. 1015–1031, August 2003.

[10] D. Cheresiz, B. Juurlink, S. Vassiliadis, and H. A. G. Wijshoff, “Performance scalability of multimedia instruction set extensions,” in *Proceedings of Euro-Par 2002 Parallel processing*, September 2002, pp. 849–861.

[11] S. Larsen and S. Amarasinghe, “Exploiting superword level parallelism with multimedia instruction sets,” in *Proceedings of the SIGPLAN ’00 Conference on Programming Language Design and Implementation*, June 2000.

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

[13] E. Salamí, J. Corbal, R. Espasa, and M. Valero, "An evaluation of different dlp alternatives for the embedded media domain," in *1st Workshop on Media Processors and DSPs*, November 1999.

[14] D. Boggs, A. Baktha, J. Hawkins, D. Marr, J. Miller, P. Roussel, R. Singhal, B. Toll, and K. S. Venkatraman, "The microarchitecture of the intel pentium 4 processor on 90nm techonology," *Intel Technology Journal*, vol. 08, no. 01, pp. 7–23, February 2004.

[15] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, , and J. D. Owens, "Register organization for media processing," in *Tenth International Symposium on High Performance Computer Architecture*, January 2000.

[16] K. Diefendorff, P. Dubey, R. Hochsprung, and H. Scales, "Altivec extension to powerpc accelerates media processing," *IEEE Micro*, vol. 20, no. 2, pp. 85–95, April 2000.

[17] J. Corbal, R. Espasa, and M. Valero, "On the efficiency of reductions in micro-simd media extensions," in *International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, September 2001.

[18] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *30th International Symposium on Microarchitecture*, 1997, pp. 330–335. [Online]. Available: citeseer.nj.nec.com/lee97mediabench.html

[19] R. Espasa, "Jinks: A parametrizable simulator for vector architectures," Universitat Politécnica de Catalunya, Technical Report UPC-CEPBA-1995-31, 1995.

[20] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, vol. 29, no. 6, pp. 196–205, June 1994.

[21] K. C. Yeager, "The mips r10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, April 1996.

[22] F. Quintana, J. Corbal, R. Espasa, and M. Valero, "Adding a vector unit on a superscalar processor," in *International Conference on Supercomputing*, June 1999, pp. 1–10.

[23] T. Lyon, E. Delano, C. McNairy, and D. Mulla, "Data cache design considerations for the itanium 2 processor," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2002, pp. 11–20.