Exploiting a New Level of DLP in Multimedia Applications.

Jesus Corbal, Mateo Valero*
Departament d'Arquitectura de Computadors,
UPC. Universitat Politècnica de Catalunya–Barcelona, Spain
e-mail: {jcorbal,mateo}@ac.upc.es

Roger Espasa
Compaq Computer Corp.,
Shrewsbury, MA
e-mail: espasa@vssad.hlo.dec.com

Abstract

This paper proposes and evaluates MOM: a novel ISA paradigm targeted at multimedia applications. By fusing conventional vector ISA approaches together with more recent SIMD-like (Single Instruction Multiple Data) ISAs (such as MMX), we have developed a new matrix oriented ISA which efficiently deals with the small matrix structures typically found in multimedia applications. MOM exploits a level of DLP not reachable by neither conventional vector ISAs nor SIMD-like media ISA extensions. Our results show that MOM provides a factor of 1.3x to 4x performance improvement when compared with two different multimedia extensions (MMX and MDMX) on several kernels, which translates into up to a 50% of performance gain when measuring full applications (20% in average). Furthermore, the streaming nature of MOM provides additional advantages for executing multimedia applications, such as a very low fetch pressure or a high tolerance to memory latency, making MOM an ideal candidate for the embedded domain.

1 Introduction

While advances in microprocessor design over the past years were primarily targeted at scientific and integer applications, it is widely accepted that *media applications* will be taking on more and more significance, becoming one of the most significant computing workloads in the next years [1].

In reaction to this trend, major vendors of general-purpose microprocessors have included SIMD extensions to their instruction-set architectures to tackle these types of applications. Examples are Intel's *MMX* [2], SUN's *VIS* [3] and Mips *MDMX* [4]. All these ISA extensions offer new packed data types, fixed-point arithmetic and, typically, 64-bit multimedia vector registers. The goal is to execute between 4 and 8 parallel fixed-point operations over small data. Although initially the new data types did not include single precision floating point numbers, more recently, the importance of *3D* graphic applications has been recognized. As a consequence, new instructions have been added to deal with floating point SIMD parallelism (Motorola's *Altivec* [5], AMD's *3DNow!* [6] and INTEL's *SSE* [7]).

While all these multimedia extensions provide considerable performance improvements, intense research still remains to be done. Future media processing (such as videoconferencing, 3D animation, or MPEG-4 video and audio streams) will require high computational demands. Additionally, the real-time constraints plus the multi-threaded orientation of future standards (such as MPEG-4 [8]), leads us to believe that further research on new ISAs or architecture paradigms for media processing is a matter of great interest.

Recent works dealing with the DLP exploitation of *multimedia* applications can be divided into two different groups: those evaluating the performance of conventional vector ISAs on multimedia codes [9, 10], and those evaluating the performance of current multimedia extensions [11, 12].

In this paper, we propose MOM. MOM stands for *Matrix Oriented Multimedia* extension and is targeted at exploiting a level of data-level parallelism not reachable by any conventional SIMD paradigm. MOM merges in a single ISA the intra-word parallelism capabilities of MMX together with the inter-word parallelism exploitation of traditional vector architectures. We will show that most of the main multimedia kernels exhibit several levels of limited parallelism. While MMX-like extensions are able to exploit parallelism from only one single level, MOM is able to exploit parallelism from up to two levels of parallelism by means of its matrix instructions.

2 Rationale for a matrix ISA

In this section we qualitatively argue why the MOM 2D parallelism exploitation is particularly well suited to multimedia applications due to the limited parallelism found in any single parallel level (loop). Additionally, we will discuss

*This work has been supported by Direccio General de Recerca de la Generalitat under grant 1998FI-00260, by the Ministry of Education of Spain under contract CICYT TIC98-05110C02-01 and by the CEPBA.

```
int dist1(blk1, blk2, length)
unsigned char *blk1,*blk2;
int length;
{
int i,j,s;
unsigned char *a,*b;
     s = 0; a = blk1; b = blk2;
     for(j=0; j<16; j++) {
          for(i=0; i<16; i++) {
               s += abs(a[i]-b[i]);
          }
          a += length;
          b += length;
     }
     return s;
}
```

Figure 1. MPEG2 Motion Estimation, pixel distance function.

```
int fullsearch(org, blk, length, i0, j0, win)
unsigned char *org, *blk;
int length, i0, j0, win;
{
int l, d, i, j, dmin, imin, jmin, k;
  ...

  for (l=1; l<=win; l++) {
    i = i0 - l; j = j0 - l;
    for (k=0; k<8*l; k++) {
        d = dist1(org+i+length*j,blk,length);

        if (d<dmin) {
          dmin = d; imin = i; jmin = j;
        }

      if       (k<2*l) i++;
      else if  (k<4*l) j++;
      else if  (k<6*l) i--;
      else             j--;
    }
  }
}
```

Figure 2. MPEG2 Motion Estimation, full search function.

the basic characteristics of our implementation of a matrix ISA.

2.1 Available data level parallelism

Figures 1 and 2 show a simplified fragment of code extracted from the motion estimation algorithm in a MPEG-2 encoder. It has been chosen as a representative example of what can be typically found in many multimedia applications.

The MPEG2 *motion estimation* algorithm detects movement of objects along different video frames in order to be able to express one single video frame as a function of the others. The program scans the reference image so that it can find which block of the reference image matches better with the block being compressed, by finding the minimal *sum of absolute differences* between the pixels of the two blocks (operation performed by the function `dist1`). Function `fullsearch` just calls function `dist1` following a spiral-like path across a window inside the reference image.

Analyzing the code shown in figures 1 and 2 we can see that there are up to three different levels of Data Level Parallelism to be exploited. The first two levels are the nested loops j and i located in function `dist1`. The third level is in function `fullsearch` where multiple independent (and, therefore, parallelizable) calls to function `dist1` are made. How would the ISA extensions under consideration (MMX and MOM) and a traditional vector compiler exploit this parallelism?

Traditional Vectorization

A conventional vector compiler would detect DLP parallelism over the inner loop i in function `dist1` and would vectorize it, generating 16-word length vector instructions. This would result in every *row* of matrices '`a`' and '`b`' (composed of 16 8-bit elements) being loaded into 16 64-bit positions of a vector register. Unfortunately, as graphically shown in figure 3 (a), there would be a waste of potential data-level parallelism because we would be loading only 8-bits of useful data into 64-bit registers.

Several vector compilers use loop interchange techniques in order to increase the effective vector length by vectorizing outer loops. Note that, in this case, this technique cannot be applied in any of the outer levels of parallelism. First, loop interchange over loop j would not provide higher vector lengths. Second, while the third level of parallelism, located at function fullsearch, could be easily exploited as thread level parallelism (TLP), it is not possible to vectorize it due to the lack of regularity (that is, no constant stride).

MMX-like Vectorization

On the other hand, a MMX-like compiler would detect parallelism in the inner loop i and would look for two necessary conditions before triggering 'vectorization': first, the data size of each element should allow packing multiple elements in a single multimedia register; second, all elements should be arranged consecutively in memory (that is, using stride one). Since both conditions are true for our example, the MMX-like compiler would generate a multimedia instruction packing groups of 8 elements in every *row* into a single 64-bit multimedia register (see figure 3 (b)). Then, all 8 elements in a multimedia register could be operated on in parallel generating eight absolute differences as shown in the figure.

Comparing to the traditional vectorization approach, we can see that the MMX-approach is fairly competitive. The vector machine would need at least eight pipes to match the performance of the MMX-approach and would most likely not be able to take advantage of specialized opcodes as the MMX-like ISAs do. Moreover, the MMX approach would achieve similar or better results with much less hardware.

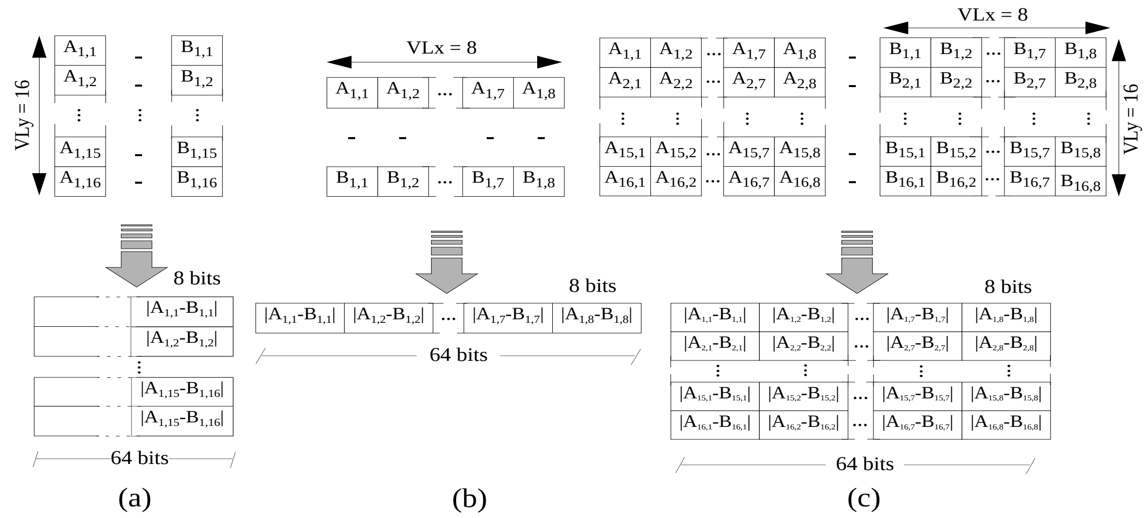

Figure 3. Comparison between (a) conventional vector, (b) MMX-like and (c) matrix oriented ISAs

MOM Vectorization

Clearly, the maximum parallelism exploitable by the two previous approaches is restricted by the reduced vector length of the inner loop. MOM overcomes this severe limitation by observing that the two nested loops in `dist1` can be *simultaneously* vectorized. MOM performs the 2D vectorization process in two steps. First, it analyzes the inner loop and generates MMX-like code for it. Then, it analyzes the outer loop (corresponding to the row index of the matrices), and vectorizes this MMX-like code generating matrix instructions (see figure 3 (c)).

Note that we allow *any* stride between two consecutive rows of the matrix, and that is a key difference between MOM and MMX-like extensions. It could be argued that we could arbitrarily enlarge the multimedia register (as Altivec does) to achieve a similar effect to what MOM does. However, enlarging the register would not provide any benefits because (a) the rows in the matrix are *not* layed out in memory in consecutive locations and (b) the MMX scope is restricted to a single loop (corresponding to a single matrix row). Therefore, in our example, using an arbitrarily large register (a la Altivec) would only allow packing up to 16 8-bit consecutive elements (that is, restricted to a single matrix row), while MOM is able to pack 128 8-bit elements (half of the matrix), as seen in figure 3.

MOM can be viewed as a conventional vector ISA where each of its computation operations are SIMD MMX-like instructions. A MOM implementation executes as many SIMD MMX-like computation operations per cycle as the number of vector pipes of the MOM functional unit. The interesting point is that a MOM register is holding 2D array of data and MOM instructions are doing matrix operations between them. Our claim is that these matrix operations occur frequently enough in multimedia applications to warrant their implementation.

Accumulators and MOM

MMX-like ISAs tend to have problems handling reduction operations. Reduction operations naturally arise in dot products, for example, where all the results from several products must be added together. The problem appears if the product is performed in parallel (using sub-word parallelism) since the result *does not* fit into a normal register. As an example, figure 4 shows that trying to multiply four 16-bit quantities yields a result that only fits in a 128-register. Since these results must be added together *before* truncating the result (or, otherwise, the loss of precision could become unacceptable), MMX-like ISAs end up using *data promotion* to maintain the required precision (i.e. promotion of data to larger data sizes by using pack/unpack operations). Unfortunately, *data promotion* causes a large instruction overhead and reduces the potential sub-word level parallelism by a factor of two.

A very efficient way to deal with reduction operations is the technique introduced by MDMX (Mips). MDMX proposes using packed accumulators which are wide registers that successively accumulate the results produced by operations done with multimedia vector registers (see figure 4(a)). The results from the accumulator are truncated, rounded and conveniently clipped into a conventional MDMX register. Unfortunately, MDMX accumulators introduce artificial recurrences due to the fact that any accumulator operation needs its previous value as an input. For long latency operations, this translates into low IPC. By contrast, MOM can take great advantage of the multimedia accumulators. Since any MOM instruction over one accumulator serializes several operations (see figure 4(b)), we can pipeline the ac-

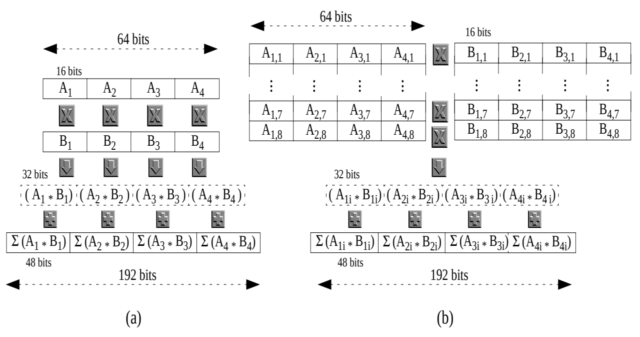

Figure 4. Example of parallel dot product performed with (a) MDMX, (b) MOM accumulators

cumulation in order to avoid the recurrence problem. This technique is very common in conventional vector machines: the vector functional unit manages a number of partial accumulations equal to the latency of the functional unit; the final result is obtained by adding together all these partial accumulations.

2.2 MOM ISA Overview

The basis of the MOM Instruction Set Architecture heavily borrows from the MDMX multimedia extension set. Therefore, most of MOM instructions can be seen as vector versions of MDMX ones. MOM is a load/store architecture composed of vector memory instructions and a set of computation instructions that operate on MOM registers. Each MOM register is composed of 16 SIMD vector elements of 64-bit each. The execution of *every* MOM instruction is controlled by the *Vector Length* register (VL), which dictates how many words (out of 16) of the MOM register will be actually operated. Additionally, the *Vector Stride* register dictates the distance in bytes between two consecutive SIMD vector elements for any MOM memory instruction. The MOM instruction set is divided into four categories (see [13] for an in-depth description of MOM):

MOM packed arithmetic and logical operations. Computation instructions always take as inputs two MOM registers and generate a MOM register as output. These instructions are straightforward matrix translations of MDMX arithmetic and logical instructions.

MOM memory instructions. MOM supports memory instructions of the form $Momldq\ MRi$ `<--` $Rj,\ Rk$, where MR_i is one of the MOM logical registers, R_j is the base address where the load starts and R_k is the *vector stride*. The semantics of the instruction are as follows: starting at address R_j, load a 64-bit word into the first position of MOM register i. Then, add the stride register R_k to the base address, decrement the VL register and repeat the operation until VL reaches 0. MOM store instructions work in a similar fashion.

MOM matrix operations. MOM includes very powerful matrix instructions such as *Matrix per vector* or MPEG-2 *Matrix sum of quadratic differences*. These instructions take one or two MOM registers as inputs and one MOM accumulator as the output. More noteworthy, however, is the capability of doing a transpose on a MOM register. This operation is especially useful to switch vector dimensions without using pack/unpack operations (as would be necessary in an MMX-like ISA).

MOM auxiliary operations. These include a set of instructions to manage the VL register and the logical accumulators (basically, to unload and restore values from them).

3 Methodology

We have studied six different programs from the *Mediabench* suite [14]: `mpeg2 encode`, `mpeg2 decode`, `jpeg encode`, `jpeg decode`, `gsm encode` and `gsm decode`. Note that these programs are representative examples of video, image and audio applications. From the six programs, `gsm decode` had a very low vectorization percentage and therefore was dropped from this study. We have used the *mei16v2rec* bit stream (four 352x480 frames) for the `mpeg2` benchmarks, the image *penguin.ppm* (1024x739) for the `jpeg` benchmarks, and the standard PCM file *clinton.pcm* for the `gsm encode` application

3.1 Emulation Libraries and Code Generation

We have developed three different emulation libraries that contain all the multimedia instructions found in MMX, MDMX and MOM. It is important to note that the libraries do not exactly model MMX and MDMX but, rather, a fair approximation of each one. For instance, we have extended all instruction sets with additional instructions such as *vector average* or *conditional move*. Also it is very important to stress that, in all cases, the baseline ISA is the Alpha ISA. Thus, although we use the name MMX, it has to be clear that we are *not* modeling an x86 ISA with multimedia extensions. Rather, we have added the MMX opcodes to the Alpha ISA. The same argument applies to MDMX and MOM.

The *MMX emulation library* contains 67 instructions and assumes an independent multimedia register file with 32 logical registers (as opposed to only 8 registers in the real MMX). We have included enhanced reduction operations and we have extended the ISA to allow up to three logical source/destination registers instead than two. The *MDMX emulation library* contains 88 instructions and assumes 32 logical multimedia registers and 4 logical accumulators. We have modeled most of the features of MDMX but the subword selector field (which allows to operate all the elements of a MDMX register with one single element of some other

MDMX register). Finally, the *MOM emulation library* contains 121 instructions and assumes 16 logical matrix registers (of 16 words each), 2 MOM accumulators and the VL register.

To the best of our knowledge, there is no available compiler able to generate either MMX or MDMX (let alone MOM). Therefore, we identified those functions with potential DLP and manually rewrote them using stylized subroutine calls to our emulation libraries. In order to maximize the performance of MMX and MDMX, we used loop-unrolling and software pipelining techniques. The correctness of the output was verified to ensure no visually perceptible losses in accuracy. Finally, we modified our *Jinks* simulator [10] to be able to filter the input instruction stream provided by *ATOM* [15] and correctly simulate the emulated instructions.

3.2 Modeled Architecture

Our modeled architecture closely follows a MIPS R10000 processor with the addition of a multimedia unit with its own register file. We have an additional register file for accumulators for MDMX/MOM, while the MOM *VL* register is renamed through the integer register pool.

Table 1 shows the processor configurations used in our simulations. We assume that simple functional units are only able to perform logical/shift and add operations. Complex functional units are able to perform multiplication and division as well. Note that for the 8-way machine, the MOM version does not have 4 multimedia functional units but 2 multimedia units of width 2 (that is, every functional unit is composed of two parallel lanes and is able to perform 2 vector operations per cycle). We have assumed the same for the memory ports. For the 8-way machine, each MOM memory port is able to leverage two vector elements per cycle (but only one element if we deal with scalar data).

We have done preliminary simulations in order to determine the number of multimedia physical registers necessary to maintain processor performance (see table 2). Note that the size of the MOM register file is 5 times the size of the MMX register file. However, there are a lot of parameters that influence the overall area of a register file (such as the number of read and write ports, or the number of physical registers). We have used the model described in [16] in order to estimate the area cost of each register file. As it can be seen in table 2, while the MOM matrix register file (plus the accumulator register file) is 5 times bigger that the MMX multimedia register file, the area costs are of the same order. The reduction of complexity of vector register files, (due basically to the fact that we can interleave the elements of every vector register among several banks) has already been highlighted in previous works [17, 18].

	way-1	way-2	way-4	way-8
ROB size	8	16	32	64
Load/Store queue	4	8	16	32
Bimodal predictor	512	2K	4K	16K
BTB entries	64	256	512	1024
INT simple/complex	0/1	1/1	2/1	2/2
FP simple/complex	0/1	1/1	2/1	2/2
MED simple/complex	0/1	1/1	2	4 - (2x2)
memory ports	1	1	2	4 - (2x2)
INT log/ph registers	32/40	32/48	32/64	32/96
FP log/ph registers	32/40	32/48	32/64	32/96

Table 1. Processor configurations.

	MMX	MDMX	MOM
MEDIA log/ph registers	32/64	32/52	16/20
ACCUMULATOR log/ph reg.	-	4/16	2/4
MEDIA rd/wr ports	6/3	6/3	2/1 (8-b)
ACCUMULATOR rd/wr ports	-	4/2	2/1
Register File Size	0.5 K	0.78 K	2.6 K
Normalized Area Cost	1	1.19	0.87

Table 2. Multimedia register file configurations for a 4-way machine.

4 Performance Evaluation

This section evaluates the performance of our matrix ISA with the set of benchmarks described in section 3. The evaluation will be decoupled in two different steps: a kernel-level analysis and a complete program-level analysis.

4.1 Kernel-level analysis

We have selected the most time-consuming kernels from our applications: `idct` (an inverse discrete cosines transform), `motion1` and `motion2` (sum of absolute differences and sum of quadratic differences algorithms for the MPEG motion estimation), `rgb2ycc` (RBG to YCC color space conversion), `compensation` and `addblock` (MPEG2 motion compensation algorithms), `ltpparameters` (calculation of the parameters of the GSM long term filter), and finally `h2v2upsample` (image zoom algorithm). We have simulated between 5 and 10 million graduated instructions for the plain superscalar version and for each kernel.

Figure 5 shows the speed-up attained by the three multimedia ISAs evaluated when compared with *Alpha* code, for different wide machines. We have considered an idealized memory system with no bandwidth constraints and a fixed memory latency of one single cycle (that is, an equivalent model of a perfect cache).

The results show that MMX and MDMX exhibit performance gains ranging from 1.5x to 15x over a pure superscalar architecture, and that MDMX slightly outperforms MMX for most of the kernels (up to a 30% of improvement). MOM

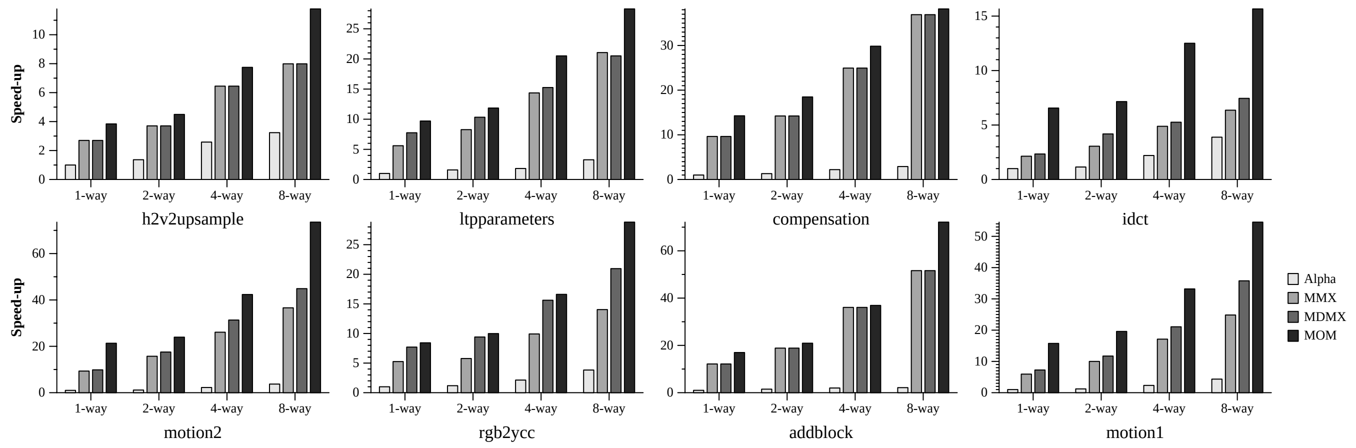

Figure 5. Speed-up of evaluated multimedia ISAs for different issue-rate machines (with respect to 1-way *Alpha* ISA performance).

clearly outperforms both MMX and MDMX with additional performance gains ranging from 1.3x to 4x. The only case where MOM is not much more effective than MDMX is in `rgb2ycc`. The reason is that vectorization happens along the color space (Red, Green and Blue) dimension, yielding a vector length of only 3.

As expected, MOM achieves higher *relative* performance for low-issue rates (for example, for `idct`, 1-way MOM is 7 times faster than a 1-way Alpha, but 8-way MOM is only about 4 times faster than the 8-way Alpha). This is due to the fact that MOM greatly reduces the fetch pressure by packing an order of magnitude more operations per instruction than MMX or MDMX, making it an ideal candidate for embedded systems where high issue rates and out-of-order execution are not even an option. The exception to this is `addblock` where all three ISAs achieve higher relative performance as we increase the fetch rate of the machine. The reason is that the original version of `addblock` uses a memory table to perform the data saturation (limiting severely the potential ILP), whereas the studied multimedia extensions include the saturation as a feature. As a result, the pure *Alpha* version becomes memory-bounded for wider processors.

Furthermore, we have done the same simulations again but with 50 cycles of memory latency (trying to approximate the effects of streaming-like memory references). Results obtained show that MOM exhibits a high tolerance to increases of the memory latency, which is a very well-known capability of vector instructions. When increasing the latency from 1 to 50 cycles, MMX/MDMX observe slow-downs ranging from 4x to 8x and common *Alpha* code observes slow-downs ranging from 3x to 9x. In sharp contrast, MOM slow-downs only range from 2x to 4x.

	Conv/MA		VC/COL	
	4-way	8-way	4-way	8-way
L1 #ports	2	4	1	2
L1 #banks	4	8	1	2
L1 latency	1 cyc	2 cyc	1 cyc	1 cyc
L2 #ports	-	-	1x2	1x4
L2 #banks	(1)	(1)	(2)	(2)
L2 latency	6 cyc	6 cyc	8/10 cyc	8/10 cyc

Table 3. Port configuration of the different memory models: (Conv) Conventional cache, (MA) Multi-Address Cache, (VC) Vector cache and (COL) Collapsing buffer cache

4.2 Complete program-level analysis

Now we address the evaluation of our proposed ISA implementation with complete programs and with realistic memory systems. For this study, we will focus only on the evaluation of MMX-like and MOM, as MDMX exhibits similar behavior to MMX.

4.2.1 Cache Hierarchy

We have included in our processor simulator a highly detailed memory hierarchy model, similar to the one found in the Alpha 21364 [19] where both L1 and L2 cache levels are located on-chip. The L1 cache is a 32 KB, direct mapped, write-through cache with 32-byte lines. The L2 cache is a 1MB, 2-way associative, write-back cache with 128-byte lines. Both levels of cache have 8 MSHRs and a 8-depth coalescing write buffer with selective flush policy. We have assumed that the L1 cache is not able to service unaligned accesses. Therefore, we assume that each memory port decouples any unaligned access into two aligned

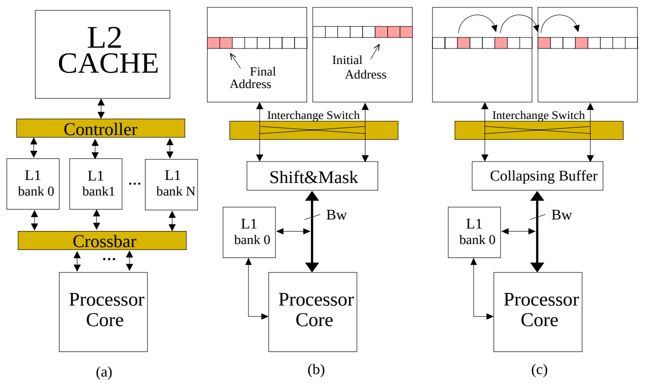

Figure 6. The different cache models evaluated: (a) Conventional/Multi-Address cache, (b) the Vector Cache, and (c) The Collapsing Buffer cache

accesses and then uses special logic to re-construct the desired data. We have modeled a 128MB *Direct Rambus* main memory system which contains a *DRDRAM* controller driving 8 *Rambus* chips and leveraging up to 3.2 GB/s with a 128-bit wide, bi-directional 200Mhz main bus. We have not simulated the instruction cache since our benchmarks have small instruction working sets.

In [10], we studied the design of cost-effective cache hierarchies to leverage high-bandwidth for out-of-order vector processors. In the same way as conventional vector instructions, MOM memory patterns have the potential to allow a smart exploitation of the spatial locality intrinsic in multimedia codes. We have evaluated three different memory models for the MOM processor: a *multi-address* cache, a *vector cache*, and a vector cache with a *collapsing buffer* (see figure 6).

A multi-address cache is simply a conventional multi-banked cache where a MOM memory access is decoupled among all available memory ports. So, if we have two independent memory ports, a MOM memory request will reserve both ports so that the first will access the odd vector elements while the other will access the even vector elements. This model has the advantage of fully taking benefit from all the port resources, even if we have only one single memory request.

The vector cache was proposed in [10] and heavily borrows from the ideas introduced in [20]. As it can be seen in figure 6, the vector cache is targeted at accessing stride-one vector requests by loading two whole cache lines (one per interleaved bank) instead of individually loading the vector elements. Then, an interchange switch, a shifter, and mask logic correctly align the data (allowing even byte-wise alignments).

The collapsing buffer [20] is a more complex version of the vector cache that is able to access several vector elements along two consecutive cache lines, even it they are not consecutively allocated. Instead of the *shift&mask* logic, the collapsing buffer logic groups the requested elements together.

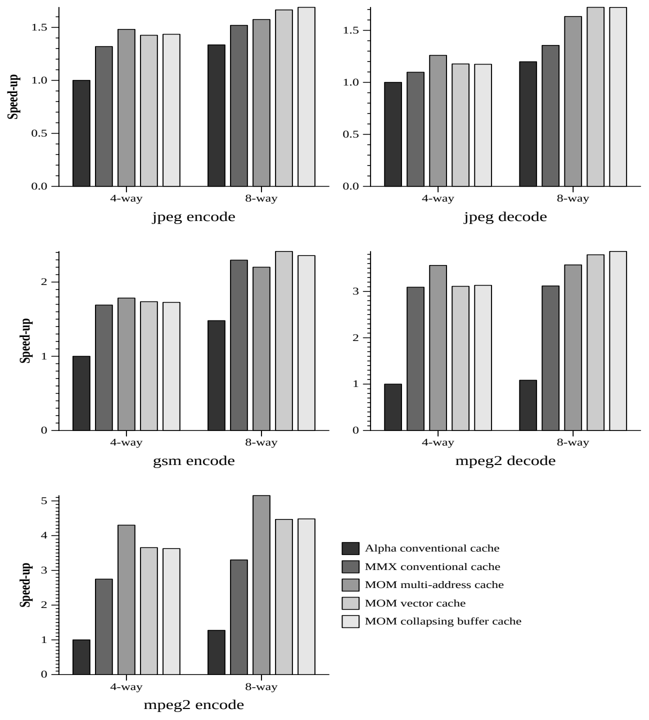

Figure 7. Speed-up of evaluated multimedia ISAs for different issue-rate machines

Note that all MOM memory accesses bypass L1 cache and go straight to the vector/collapsing buffer cache. We believe that this approach (a) avoids jeopardizing the L1 cycle time, (b) effectively decouples the vector working set from the scalar working set and (c) is not detrimental to performance due to the latency tolerance properties of MOM memory accesses. A coherence-protocol (based on an exclusive-bit policy plus inclusion between L1 and L2) has been included. Table 3 shows the port configuration for the different cache models.

4.2.2 Performance results

Figure 7 shows the performance results for the five evaluated benchmarks and for all cache models described. Because of Amdahl's law, the speedups achieved when running full applications are clearly smaller than those presented for the kernels alone. MMX shows speedups over pure *Alpha* code ranging from 1.1X to 3.1X while MOM delivers performance achievements ranging from 1.5X to 4.3x (20% of performance gain over MMX in average).

While looking at the performance of the different cache models, we realize that for the 4-way processor, the Multi-address cache outperforms both the vector and the collapsing buffer caches. The reason is that, in sharp contrast with numerical applications, the set of benchmarks under study have working sets that fit in L1 cache, and, therefore, there is no direct benefit from bypassing the vector workload to a larger level of cache. On the other hand, for the 8-way machine, we observe that both vector and collapsing buffer caches achieve better performance than the multi-address cache. This is due to the fact that for more aggressive architectures, we take advantage of the potential spatial locality exploitable by the vector/collapsing cache which delivers high effective bandwidth. In sharp contrast, a conventional cache scheme based on interleaved banks provides poor performance due to bank collisions and increased complexity of the interconnection network. The only exception to this is `mpeg2 encode` where the vector/collapsing caches present modest performance due to the large values of the strides of most MOM vector accesses. These large strides cause individual words in a MOM access to lie in far apart cache lines and neither the even-odd banking scheme of the vector cache nor the collapsing buffer can capture and compress this far-apart words into a single memory access.

5 Summary

In this paper we have proposed a novel ISA paradigm based on matrix SIMD instructions in order to leverage a new level of performance improvement when comparing with current multimedia extensions. Matrix ISAs are able to exploit a level of DLP not reachable by neither conventional vector ISAs nor current multimedia ISA extensions.

By fusing the sub-word level parallelism approach together with the sequential/streaming-like conventional vector approach, we have developed an ISA able to efficiently deal with the small matrix structures typically found in several multimedia kernels. A side benefit of our proposed ISA is that it matches very well with having accumulators, which provides us with both precision and parallelism.

We have evaluated five benchmarks from the *Mediabench* suite and we have reported local performance improvements ranging from 1.3x to 4x relative to MMX and MDMX multimedia extensions. This improvements translate in up to a 50% of performance gain for complete programs and a 20% of improvement in average. We have demonstrated that all these performance gains have been achieved without adding much extra hardware to a regular out-of-order core, since (a) our proposed register file is relatively small and (b) we have used the same number of functional units as any conventional multimedia extension implementation.

Furthermore, MOM appears as a suitable alternative for multimedia embedded systems as exhibits a high tolerance to memory latency, a very low fetch pressure, and a high potential to exploit spatial data locality with smart cost-effective cache devices.

References

[1] K. Diefendorff and P.K. Dubey. How multimedia workloads will change processor design. *IEEE Micro*, pages 43–45, Sep 1997.

[2] Alex Peleg and Uri Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, pages 42–50, August 1996.

[3] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS Speeds New Media Processing. *IEEE Micro*, August 1996.

[4] Mips extension for digital media with 3D. Technical Report http://www.mips.com, MIPS technologies, Inc., 1997.

[5] Altivec Technology. Technical Report http://www.mot.com/SPS/PowerPC/AltiVec/, Motorola, Inc., 1998.

[6] 3DNOW! technology manual. Technical Report http://www.amd.com, Advanced Micro Devices, Inc., 1999.

[7] Pentium III processor: Developer's manual. Technical Report http://developer.intel.com/design/PentiumIII, INTEL, 1999.

[8] Rob Koenen. MPEG-4, multimedia for our time. *IEEE Spectrum*, pages 26–34, February 1999.

[9] Mark G. Stoodley and Corinna G. Lee. Simple Vector Microprocessors for Multimedia Applications. In *MICRO 31*, December 1998.

[10] Francisca Quintana, Jesus Corbal, Roger Espasa, and Mateo Valero. Adding a vector unit on a superscalar processor. *13th International Conference on Supercomputing*, June 1999.

[11] Huy Nguyeni and Lizy Kurian John. Exploiting SIMD parallelism in DSP and multimedia algorithms using the altivec technology. *13th International Conference on Supercomputing*, 1999.

[12] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. Performance of image and video processing with general-purpose and media ISA extensions. *ISCA 26*, May 1999.

[13] Jesus Corbal, Roger Espasa, and Mateo Valero. MOM: Instruction set architecture. Tech. report, Universitat Politècnica de Catalunya, 1999.

[14] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *MICRO 30*, 1997.

[15] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[16] David Lopez, Josep Llosa, Mateo Valero, and Eduard Ayguade. Resource widening versus replication: limits and performance-cost trade-off. *12 International Conference on Supercomputing* July 1998.

[17] Derek J. DeVries and Corinna G. Lee. Initial results on the performance and cost of vector microprocessors. In *MICRO 32*, December 1997.

[18] Krste Asanovic. Vector microprocessors. Phd thesis, University of California at Berkeley, 1998.

[19] Peter Bannon. Alpha 21364: A Scalable Single-chip SMP. Technical Report http://www.digital.com/alphaoem/microprocessorforum.htm, Compaq Computer Corporation, 1998.

[20] Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, and Burzin A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *ISCA 22*, June 1995.