# On the Efficiency of Reductions in $\mu$-SIMD Media Extensions

Jesus Corbal, Roger Espasa and Mateo Valero
Departament d'Arquitectura de Computadors,
Universitat Politècnica de Catalunya–Barcelona, Spain*

## Abstract

*Many important multimedia applications contain a significant fraction of reduction operations. Although, in general, multimedia applications are characterized for having high amounts of* Data Level Parallelism, *reductions and accumulations are difficult to parallelize and show a poor tolerance to increases in the latency of the instructions. This is specially significant for $\mu$-SIMD extensions such as MMX or AltiVec. To overcome the problem of reductions in $\mu$-SIMD ISAs, designers tend to include more and more complex instructions able to deal with the most common forms of reductions in multimedia. As long as the number of processor pipeline stages grows, the number of cycles needed to execute these multimedia instructions increases with every processor generation, severely compromising performance.*

*This paper presents an in-depth discussion of how reductions/accumulations are performed in current $\mu$-SIMD architectures and evaluates the performance trade-offs for a near-future highly aggressive superscalar processors with three different styles of $\mu$-SIMD extensions. We compare a MMX-like alternative to a MDMX-like extension that has Packed accumulators to attack the reduction problem, and we also compare it to MOM, a matrix register ISA. We will show that while packed accumulators present several advantages, they introduce artificial recurrences that severely degrade performance for processors with high number of registers and long latency operations. On the other hand, this paper demonstrates that longer SIMD media extensions such as MOM can take great advantage of accumulators by exploiting the associative parallelism implicit in reductions.*

## 1 Introduction

Given the increasing importance of media applications for the desktop market, general-purpose microprocessor designers keep including new and better $\mu$-SIMD (sub-word level) ISA extensions in every processor generation. In fact, media ISA extensions have become as natural as floating-point or integer instructions. Since the first generation of media extensions, with registers of limited width (32-64 bits) and only integer arithmetic (MMX [1]), new extensions have introduced wider $\mu$-SIMD registers (128 bits in Altivec [3] and SSE [5]) and $\mu$-SIMD floating point arithmetic (as in Motorola's *AltiVec*, AMD's *3DNow!* [4] and INTEL's *SSE* and *SSE2* [6]).

Despite all these efforts, little has been made to solve the problem of accumulations and other reduction operations. Reduction and accumulation operations are commonly found in most media applications and severely limit the available data level parallelism. Typical examples of media algorithms involving reduction operations are the *DCT*, the MPEG2 *Motion Estimation*, *FIR filtering* or Viterbi's *Speech Recognition*.

Reduction processes have been a field of research for a long time in the supercomputing domain, specially for the general case of linear recurrences. Linear recurrences, while not obvious, contain parallelism that can be exploited by both vector and multiprocessor architectures (using solvers based on linear bi-diagonal systems). Previous works dealing with this problem are [14, 15, 16].

Two main problems arise when dealing with reductions/accumulations in $\mu$-SIMD architectures. First, when using sub-word level parallelism (as in MMX), operating together parts of a register requires overhead instructions to unpack the different bit fields. Second, *multiply&accumulate* instructions require higher precision for their intermediate results (even if the final result is eventually going to be truncated, rounded and clipped). To accommodate this extra precision, data types are promoted to larger sizes and, as a consequence, $\mu$-SIMD parallelism is reduced. Most solutions to the reduction problem proposed so far for $\mu$-SIMD architectures are based on including certain reduction/accumulate instructions useful for some few known algorithms. For instance, MMX introduced PMADDWD (Packed Multiply and Add) specifically dedicated for dot products, while *SSE* introduced PMIN/PMAX (Packed Min/Max of elements) for Viterbi's speech recognition. Other instructions such as the *Sum of Absolute Differences* (used for the MPEG2 *Motion Estimation*) can be found in more than one multimedia ISA extension (namely, INTEL's *SSE*, SUN's *VIS* [7] and Motorolas's *AltiVec*).

There are three main problems related to reductions and accumulations in $\mu$-SIMD architectures:

*This work has been supported by the Ministry of Education of Spain under contract CICYT TIC98-05110C02-01 and by the CEPBA.



0-7695-1363-8/01 $10.00 © 2001 IEEE

- *logical overhead*: the amount of additional complex $\mu$-SIMD instructions required to manage reductions
- *parallelism exploitation*: the way the ISA hides/shows the potential parallelism in any given reduction process
- *latency sensitivity*: the tolerance to increases in the latency of the $\mu$-SIMD complex instructions

Interestingly, these three factors become even more relevant if we take into account current trends in processor design. Analyzing the evolution of several well-known families of processors (such as INTEL's Pentium, AMD's Kx, or Compaq's Alpha 21x64 families) we can identify two common characteristics. First, an increase in the number of in-flight instructions than can be executed (thanks to bigger windows and higher number of registers). Second, increasing working frequencies helped partially by the over-segmentation of the machine pipeline (which usually produces an increase of the latency for the more complex instructions). Under this scenario, the ideal $\mu$-SIMD ISA would be one which had both high tolerance to increases in the latency of instructions and the property of making explicit the potential parallelism to the increasingly wider issue engine of the processor.

This paper presents an in-depth discussion of how reductions/accumulations are performed in current $\mu$-SIMD architectures and discusses two alternatives that tackle the accumulation and reduction problem: the *packed accumulators* introduced by MIPS in its MDMX extension and the matrix registers with accumulators introduced in the MOM ISA [8].

In particular, we will show that packed accumulators allow to reduce a large amount of logic overhead instructions that have a big impact on performance for long latencies, but, at the same time, this advantage is overridden by the artificial recurrences introduced by them. Therefore, neither of the two alternatives (extensions with accumulators and extensions with no accumulators) seem to fit very well in a scenario with increasing number of registers and raising latencies. On the other hand, we will show that when accumulators are combined with longer (vector-like) matrix registers, we can take full benefit of the advantages of the packed accumulators while sidestepping their drawbacks.

## 2 Reductions in $\mu$-SIMD architectures

### 2.1 Are reductions really that important?

The first obvious approach to determine the relevance of reduction operations for multimedia would be to perform a basic count of every kind of instruction in a set of representative media benchmarks. Figure 1 shows the dynamic instruction breakdown for five benchmarks of the *mediabench* suite [10] re-written using MMX-style $\mu$-SIMD instructions. Those MMX-style instructions have been divided into four different categories: (a) reduction instructions (that is, $\mu$-SIMD instructions directly involved in reductions), (b) arithmetic instructions, (c) logical/overhead instructions (related to


**Figure 1. Instruction breakdown for several mediabench programs (percentage).**

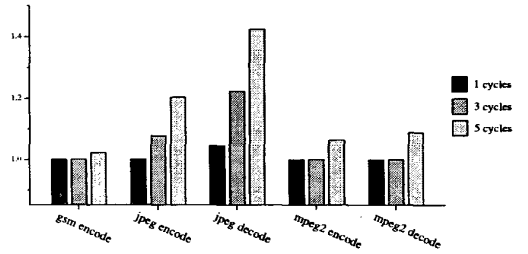


**Figure 2. Increase of the number of execution cycles (normalized) when increasing the latency of MMX non-memory instructions.**

$\mu$-SIMD overhead such as data promotion or sign conversion), and (d) $\mu$-SIMD memory instructions. Looking at figure 1, it might seem that reductions are not very relevant for overall performance, as they only account for 5 % of the overall number of instructions (in average).

However, reductions are actually a critical factor, due to two main reasons. First, they introduce a high amount of logic overhead (most of the MMX-like logic instructions in the figure, which account for 10-15% of the overall number of instructions). Second, they are highly sensitive to increases in the latency of the instructions.

We have evaluated the performance of the same set of benchmarks increasing the latency of all non-memory $\mu$-SIMD instructions by 1, 3 and 5 cycles. The normalized execution time can be seen in figure 2. The baseline architecture is described later in section 4.1. Results show significant performance slow-downs for most of the programs when the latency offset exceeds 1 cycle.

Our claim is that the latency sensitivity of reduction operations will show up as a significant concern in designing media processors. Recently, the 1GHz frequency barrier has been broken, partly thanks to super-pipelining (which helps alleviate the critical paths of the architecture). While doing so, great care has been taken in trying not to increase the latencies of operations critical for performance such as memory or integer basic arithmetic instructions. This is not the case, however,

| operation | int | int SIMD | | | fp SIMD | |
|---|---|---|---|---|---|---|
| | Alu | Padd | Pmul | Pmadd | Padd | Pmul |
| Pentium II (MMX) | 1 | 1 | 3 | 3 | - | - |
| Pentium III (SSE) | 1 | 1 | 3 | 3 | 4 | 5 |
| Pentium IV (SSE2) | 1 | 2 | 8 | 8 | 4 | 6 |
| AMD K6 (3DNow!) | 1 | 1 | 2 | 2 | 2 | 2 |
| AMD K7 (3DNow!) | 1 | 2 | 3 | 3 | 4 | 4 |

**Table 1. Instruction latency evolution: (Alu), basic arithmetic; (Padd), parallel SIMD add; (Pmul), parallel SIMD multiply; (Pmadd), parallel multiply& add.**

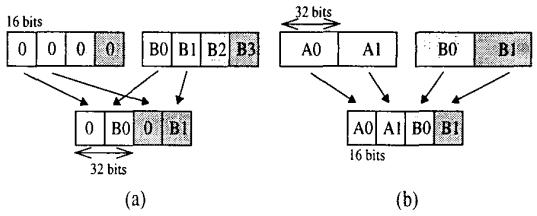


**Figure 3. Data promotion (a) and demotion (b) of 16-bit unsigned integer data.**

for the more complex $\mu$-SIMD instructions, and, as a result, their latencies increase with every processor generation.

Table 1 shows latencies for different processor generations and for different kinds of instructions. As we may observe, basic arithmetic instructions (such as *integer add*) do not increase their already short latency. On the other hand, most of $\mu$-SIMD instruction latencies tend to increase. Compare for instance, SIMD instruction latencies for the AMD K6 processor and for AMD K7. The most evident case, though, is the Pentium IV, as their extremely depth pipeline translates into increases in the latency of up to 5 cycles.

In this paper we will demonstrate that by carefully dealing with reduction processes we can greatly alleviate the performance degradation associated with the latency increases of the most complex $\mu$-SIMD instructions.

## 2.2 Reductions in conventional $\mu$-SIMD ISAs

As already mentioned, three major issues arise in $\mu$-SIMD reductions: overhead, parallelism and sensitivity to the latency. The first one, the overhead caused by reductions is tightly related to two different aspects:

1. Maintaining precision of intermediate results
2. Intra-register dependences

The first problem typically arises when performing series of additions over a large amount of input values or when multiplying small bit quantities, because the result of the operation may not fit within the allocated bits in the $\mu$-SIMD register. For example, multiplying two registers that hold eight 8-bit quantities requires 128 bits of storage in the result register if full precision is desired. Since each 8-bit product yields a 16-bit result, the sub-word element slots of the $\mu$-SIMD register are not large enough and precision is lost if the overall size is kept at 64 bits. A very common example is the dot product, where the results of several products must be added together. Unless the output of each product is kept in its fully expanded form (i.e., using twice the number of bits of the source operands) and accumulated to the previous products using this expanded form, the loss of precision may become unacceptable. Only when all additions have been performed using double precision (128-bits), the final result can be truncated, clipped and stored back into the register file using the usual register size (64-bits).

Although some partial solutions exist to this problem, for the general case programmers have to maintain the required precision using *data promotion* and *data demotion*. *Data promotion* stands for promoting the data to larger data sizes using pack/unpack operations (see figure 3-(a)), while *data demotion* stands for the inverse process (see figure 3-(b)). The drawback of data promotion is that, after doubling the size of the data, we halve the number of slots per SIMD register and, hence, we also halve the number of operations executed per SIMD instruction. This, coupled with the instruction overhead that pack/unpack operations represent, ends up reducing overall performance.

The second problem, operating parts of the same $\mu$-SIMD register, also requires using pack/unpack and other logic instructions to extract from the $\mu$-SIMD register the desired bit slices and store them into general purpose registers. Once the required bits have been extracted, they can be operated upon using normal instructions.

Certain reduction/accumulation patterns are so common that most designers have included some instructions that alleviate the difficulties intrinsic to them. For example, figure 4-(a) illustrates the semantics of instruction PMADDWD from MMX. This instruction is extremely useful for dot products because it solves the precision problem and because it adds together slots from the same register in parallel. Another example targeted at the intra-register operation problem, is illustrated in figure 4-(b). The instruction shown is the *Sum of Absolute Differences*, very useful for the MPEG2 motion estimation.

Another of the relevant issues in $\mu$-SIMD reductions is the sensitivity to increases in the latency. While conventional $\mu$-SIMD ISAs suffer from the impact of the logical overhead, significant parallelism can be achieved by taking advantage of the associativity and commutativity properties that reductions are characterized for. In order to illustrate so, we can study a MMX-like implementation of a media-typical dot product code:

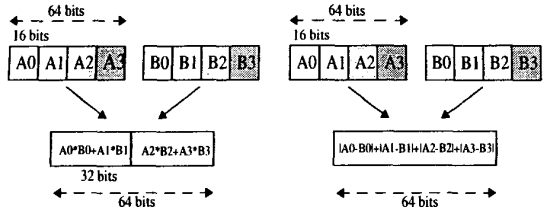


**Figure 4. Special $\mu$-SIMD instructions to manage intra-dependences: (a) Multiply and Add (used for DOT products, $\sum a_i \times b_i$); (b) Sum of Absolute Differences (used for MPEG2 Motion Estimation, $\sum |a_i - b_i|$).**

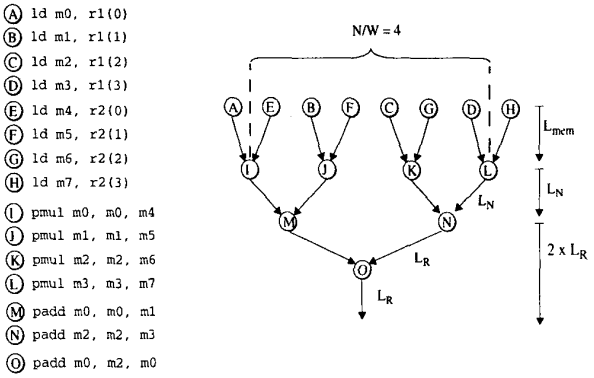


**Figure 5. Multiply&Accumulate reduction process in a MMX-like (no accumulators) architecture .**

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

Figure 5 shows the implementation of the previous code using a MMX-like ISA with 32 logical registers, as well as the dependence graph of the instructions. In order to simplify the example, we assume that no data promotion is required for the operands. We may note that the execution of the code can be divided into three different stages: (a) accessing memory, (b) non-reduction stage (the operand multiplication), and (c) reduction stage (the accumulation of the results of each multiply). The first two stages can be executed totally in parallel as there are no dependences between instructions. On the other hand, the reduction stage is performed following a binary-tree like dependence graph. Note that the last accumulation of the elements in the last $\mu$-SIMD register must be performed with scalar instructions.

Therefore, if we assume that $N$ is the number of individual elements that have to be accumulated down to a single value (16 results from the multiplies in the example above), that $W_{mmx}$ is the number of packed elements per $\mu$-SIMD register (assuming 4 in the example), and that $Lmem$ is the latency of the memory instructions, $L_N$ is the latency of the non-reduction instructions (the packed multiply) and $L_R$ the latency of the reduction instruction (the packed add), then the critical path is:

$$T = L_{mem} + L_N + log_2(\frac{N}{W_{mmx}}) \times L_R$$

Note that this minimum execution time is extremely optimistic. First, it does not take into account limits in the number or resources (load ports and functional units). Second, it does not account for the time required to perform data promotion (which adds more instructions to the critical path) as well as the time required to perform the last set of reduction operations via scalar instructions. Finally, due to data promotion, $W_{mmx}$ would be typically half the potential $W_{mmx}$, given the fact that data promotion doubles the size of the packed elements.

Therefore, we have seen that conventional $\mu$-SIMD ISAs such as MMX are able to exploit a fair amount of parallelism from reductions if used efficiently, but suffer from the impact of the high amount of the logic overhead involved in processes such as data promotion.

## 2.3 Reductions using Packed Accumulators

A specific solution for the problems outlined in the previous section has been in use in DSP processors for many years. Since DSP processors are usually targeted at a relatively narrow range of applications and have historically been characterized for their small-length registers (8-16 bits), DSP architects found that it pays off to offer specific hardware support for reductions in the form of accumulators. These accumulators, with extra bits to maintain precision correctly, are the base for the multiply&accumulate instructions that almost every DSP in the market offers today.

MDMX, the extension proposed by MIPS, adopted the idea of accumulators. MDMX introduces a relatively standard set of media instructions that operate on special $\mu$-SIMD registers and extends this set with one special 192-bit wide register called the "Packed Accumulator". The *Packed Accumulator* is used to successively accumulate the results produced by operations done on conventional $\mu$-SIMD registers. Figure 6 shows a block diagram illustrating how the accumulator works. When performing normal MDMX operations, the result produced by the ALU is simply sent back to the multimedia register file. However, the user can specify with a special opcode that the output of the ALU will be added to the current contents of the accumulator (in general, the user could select among a set of several accumulators, although MDMX

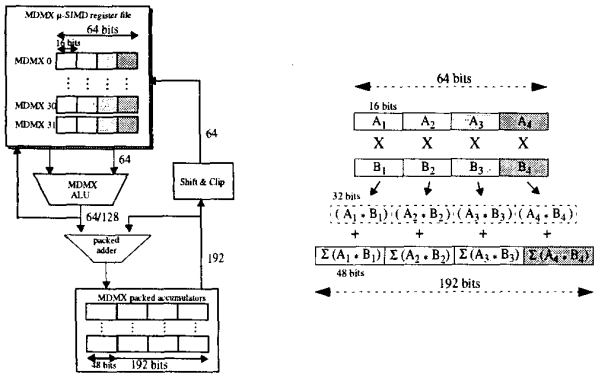


**Figure 6. A block diagram of a hypothetical MDMX implementation, and an example of application.**

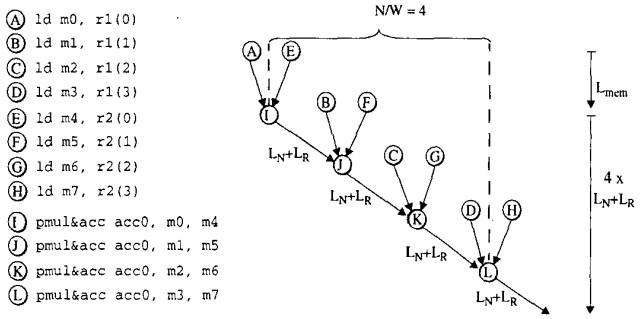


**Figure 7. Multiply&Accumulate reduction process in a MDMX-like (with accumulators) architecture.**

in particular only offers a single one). After performing several accumulations, there are instructions that allow us to take the values contained in the packed accumulator(s), accumulate them down to a single value (if desired), truncate it, round it and/or clip (saturate) it, and finally write it back into a normal MDMX register.

By having a wide accumulator, MDMX avoids using data promotion, as the accumulator is wide enough to hold the outputs of the ALU with the required precision. The packed accumulator also deals partially with intra-register dependences, as MDMX also introduces special instructions to perform accumulation of the different sub-word packed elements inside the accumulator. The approach chosen by MDMX shows up as a suitable alternative to overcome the problem that exhibit more conventional $\mu$-SIMD ISAs, that is, the huge overhead produced by the need to perform data promotion and to manage intra-register dependences.

Nevertheless, if we turn now to the problem of extracting parallelism, MDMX faces the problem of introducing artificial recurrences as any packed accumulator operation needs its previous value as an input. For long latency operations, this translates into low IPC. Therefore, the nature of the accumulator itself hides the associative property of reductions, failing to exploit the maximum parallelism available.

Figure 7 shows the implementation of the same example of code using a MDMX-like ISA with 32 logical registers and a single accumulator. As we can observe, thanks to the *parallel multiply&add* instruction that can be performed over the packed accumulator, the number of instructions drop considerably. On the other hand, the latency to execute a whole *parallel multiply&add* instruction is higher ($L_N$ cycles for the multiply and $L_R$ for the addition). Therefore, if we assume again that $N$ is the number of individual elements that have to be reduced/accumulated down to a single value, and $W_{mdmx}$ is the number of elements per MDMX register, then the critical path of the graph can be expressed as:

$$T = L_{mem} + \frac{N}{W_{mdmx}} \times (L_N + L_R)$$

Compared to the previous model, we can observe that the MDMX accumulator introduces several factors that decrease potential ILP: first, due to the serialization forced by the behavior of the accumulator, we have lost a "$log_2$" factor in the number of elements to be reduced; second, we have increased the proportional latency factor due to joining together the non-reduction and the reduction operations. However, we now benefit from being $W_{mdmx}$ up to two times $W_{mmx}$, from avoiding data promotion, and from having additional support to perform the last series of accumulations over the same register.

Summarizing, MDMX-style of execution brights where other more conventional $\mu$-SIMD ISAs fail; that is, in reducing the number of instructions required to perform the reduction. However, it is unable to extract the potential ILP available, thus being hardly a suitable choice for aggressive out-of-order processors.

A possible solution that might offset the drawbacks of the MDMX-like style of execution would be to implement more than one *packed accumulator* and then split the reduction process between them. Note however, that we would be forced to finish the reduction from the results of every accumulator using scalar instructions.

## 2.4 Reductions using Matrix ISAs

In [8] we proposed MOM, a matrix ISA that is basically a hybrid between traditional vector and $\mu$-SIMD ISAs, able to exploit up to two different dimensions of parallelism (parallel loops). MOM can be viewed as a conventional vector ISA where each of its computation operations are $\mu$-SIMD MDMX-like instructions. In fact, every MOM register holds 16 $\mu$-SIMD registers of 64-bits each (as shown in figure 8).

The rationale and the benefits of matrix ISAs are beyond the scope of this paper and we refer to [8] for an in-depth

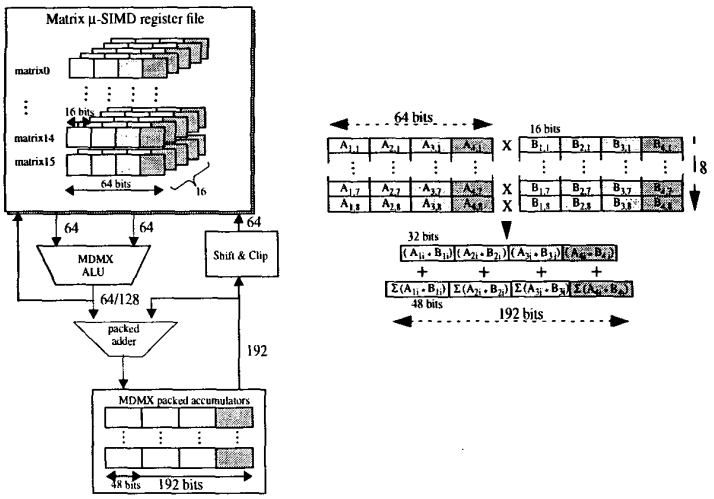


**Figure 8. A block diagram of a hypothetical matrix ISA implementation, and an example of application.**

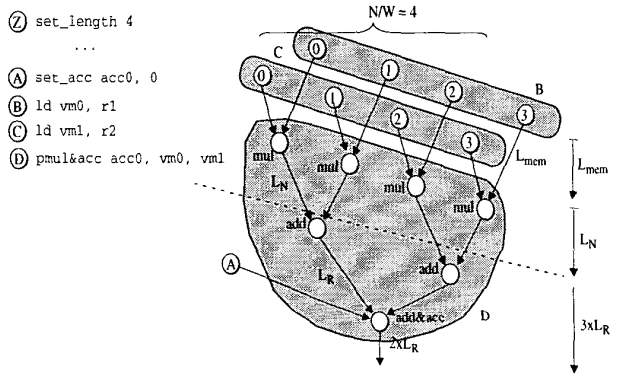


**Figure 9. Multiply&Accumulate reduction process in the matrix ISA architecture.**

discussion. The focus of this paper is on showing how matrix ISAs, by using packed accumulators, can solve the latency/ILP problem.

When a MOM matrix instruction that uses one accumulator is launched, it will perform up to 16 accumulations over the same accumulator register (8 in figure 8). Because the accumulation is associative, we can hide the latency of the accumulator by pipelining it and maintaining several shadow accumulations in flight simultaneously in the pipeline. This is a very common technique used in traditional vector architectures. As a result, the set of operations described in a matrix reduction instruction is executed following a perfect scheduling that minimizes execution time.

Figure 9 shows a scheme of the whole accumulation process using matrix $\mu$-SIMD instructions. As we may observe, we only need three instructions to perform the whole dot product (plus an instruction to set the matrix length to four and another to reset the accumulator) : two matrix loads and one matrix *parallel multiply&add* over accumulator 0. Again, we assume $N$ elements contained in every matrix register to be reduced, and $W_{matrix}$ packed elements in every sub-register within the matrix register. Thanks to the fact that all the information about the reduction process is contained in a single instruction, the pipeline may take advantage of the same binary-tree style parallelism that our MMX-like example exploited before. As a result, the minimum execution time required to finish the whole reduction process in a matrix pipeline would be:

$$T = L_{mem} + L_N + (log_2(\frac{N}{W_{matrix}}) + 1) \times L_R$$

which is only slightly longer than the critical time for the MMX case due to the writing of the final result over the accumulator.

Overall, the matrix architecture is able to take benefit from the advantages of the two previous alternatives: it is able to extract parallelism almost as well as common $\mu$-SIMD ISAs while at the same time exhibit the same reduction of logic overhead typical of other $\mu$-SIMD ISAs with packed accumulators such as MDMX.

## 3 ISA comparison

In the previous sections we have seen analytical models that show the critical path of a given reduction process for three $\mu$-SIMD philosophies. These previous analytical models are useful to understand the advantages and the drawbacks of every alternative. However, several factors have been overlooked in these easy examples that actually affect the real per-

formance of the different architectures: the limit in the number of resources, the use of special instructions, the impact of data promotion, and the opportunities to overlap reduction processes with other instructions concurrently.

In this section we will make a more exhaustive study of the three styles of ISA under the scenario of a near-future aggressive out-of-order superscalar processor. Particularly, we will analyze the impact of increasing the number of registers (as a measurement of potential parallelism exploitation) and the impact of increasing the latency of the $\mu$-SIMD instructions.

## 3.1 Benchmarks and Code Generation

We have selected six full applications from the *Mediabench* suite [10] that are representative of video, image and audio applications: `mpeg2 encode`, `mpeg2 decode`, `jpeg encode`, `jpeg decode`, `gsm encode` and `gsm decode`. As discussed in [8], use of full applications to evaluate proposals in the multimedia domain is paramount due to the large difference in behavior between kernels and full applications. However, we will start by using a set of kernels and an idealistic memory system. The reason is that we want to help highlighting the differences in latency tolerance, ILP extraction and register usage between the three ISAs under study and, thus, we want to study the effect of reductions in isolation. Of course, section 4.5 will present final performance results for the *full* applications and realistic memory models.

The most relevant parts of each application have been identified using profiling and have been rewritten in assembly language using the three styles of ISA under study: conventional (MMX-like), with accumulators (MDMX-like), and matrix-ISAs with accumulators (MOM). The resulting programs are a mixture of plain Alpha instructions and the special multimedia instructions defined in each of the three architectures. To achieve this effect, we have used the emulation libraries described in [8] running under ATOM [11]. We have taken a few liberties in the definitions of the SIMD ISAs (i.e., they are not exact replicas of MMX and MDMX). The most important difference is that, in order to allow aggressive scheduling optimizations in the assembly programming, we assumed 32 logical MMX registers (instead of 8). We also assumed 4 logical accumulators for MDMX (instead of one in the real definition) and 16 matrix registers and 2 logical accumulators for the MOM ISA. For the MMX-like version of code we have used the re-association optimization similar to the one found in figure 5. For The MDMX-like version, whenever possible, we have mapped independent reduction operations onto different logical accumulators.

Of all the pieces re-written in assembly in each application, we have selected a few to be analyzed in-depth in the following sections. These are: `idct`, which performs an *Inverse Discrete Cosine Transform* over 8x8 matrices of data; `motion1`, which performs a *Sum of absolute differences* between two 16x16 image chunks (found in the MPEG2 motion estimation) and `motion2`, which performs the *Sum of quadratic differences*; `addblock` and `compensation`, which are used for saturated blending of images in the Motion Compensation algorithm; `ltpparameters` and `ltpfiltering`, which are special dot products used to calculate the long term filter parameters and the filtering itself in the gsm encoding/decoding process; `h2v2` which performs a 2x2 zoom over the whole image in JPEG decoding; and lastly, `rgb2ycc`, which performs an image conversion from RGB to YCC format.

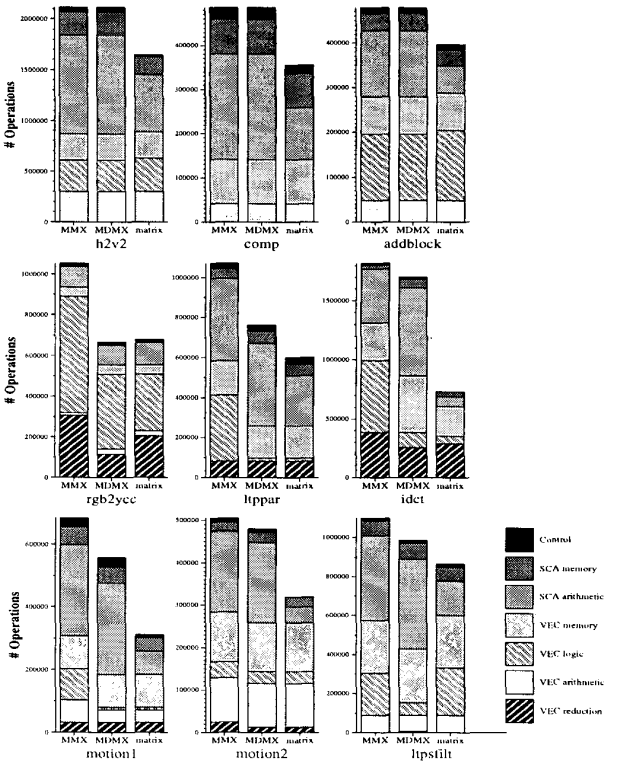


**Figure 10. Operation count of the ISAs under study.**

## 3.2 Operation count

Figure 10 shows the dynamic operation count for every ISA under study (note we present operations rather than instructions because every MOM instruction can encode up to 16 different $\mu$-SIMD MDMX-like instructions). Operations are decoupled into two main groups: scalar (conventional) instructions, and vector ($\mu$-SIMD) operations. Each group is further divided into memory and non-memory operations. In order to study the characteristics of every kernel, we have also broken down non-memory vector operations into three main groups: logic operations, arithmetic operations and reduction operations.

From data on the figure, we can observe that the number of scalar instructions executed for the MMX-like and MDMX-

| operation | MMX | | | MDMX | | | matrix | | |
|---|---|---|---|---|---|---|---|---|---|
| perc % | log | ari | red | log | ari | red | log | ari | red |
| motion1 | 49 | 36 | 15 | 10 | 50 | 40 | 10 | 50 | 40 |
| motion2 | 23 | 61 | 15 | 20 | 72 | 9 | 20 | 72 | 9 |
| rgb | 64 | 1 | 35 | 72 | 5 | 23 | 55 | 5 | 40 |
| ltpsfilt | 70 | 30 | 0 | 41 | 55 | 4 | 73 | 27 | 0 |
| ltppar | 80 | 0 | 20 | 17 | 0 | 83 | 17 | 0 | 83 |
| idct | 61 | 0 | 39 | 33 | 0 | 67 | 18 | 0 | 82 |
| h2v2 | 50 | 50 | 0 | 50 | 50 | 0 | 52 | 48 | 0 |
| comp | 0 | 100 | 0 | 0 | 100 | 0 | 0 | 100 | 0 |
| addblock | 75 | 25 | 0 | 75 | 25 | 0 | 76 | 24 | 0 |

**Table 2. Percentage of the different types of non-memory vector operations (namely, vector logic overhead, vector arithmetic operations, and vector reductions operations).**

like codes are fairly similar. However, the matrix ISA reduces the number of both memory and non-memory scalar instructions (45% less instructions on average). The reason is that, as the matrix ISA can pack several $\mu$-SIMD instructions into a single matrix instruction (thus, replacing multiple instances of a loop) there is an elimination of instructions involved in loop control (i.e., branches, loop indexes, and address arithmetic).

Accumulators are extremely useful in reducing the overall number of non-memory vector operations. Note that five of the kernels under study have implicit reduction operations (MDMX uses accumulators in `ltpsfilt` but not because there is a real reduction properly) while four do not. For those kernels where accumulators are not used (`addblock`, `comp`, `h2v2`, and `ltpsfilt` for the matrix ISA), the overall number of non-memory vector operations is the same. On the other hand, for the rest of the benchmarks, MDMX-like and MOM codes have an average of 50% less non-memory vector operations. This overhead reduction is due to the elimination of almost all data promotion/demotion instructions and other related logic overhead.

MDMX versions of code, in sharp contrast with the other two ISA styles, are dominated by scalar instructions. This is due to the fact that *Packed Accumulators* reduce the overall number of vector instructions. On the other hand, MOM is not dominated by scalar instructions (despite having also *Packed Accumulators*) because the it also reduces the overall number of scalar instructions thanks to its second dimension in the matrix registers that allows getting rid of loop overhead instructions.

In order to gain further insight into which kind of operations have been removed by MOM and MDMX, we can look at table 2. Table 2 shows the predominance of the three different types of non-memory vector operations, namely, vector logic operations, vector arithmetic operations and vector reduction operations. Vector logic operations are those involved in $\mu$-SIMD managing overhead (such as data promotion/demotion, matrix transpose, sign conversion or data re-arranging), as well as those operations involved in precision conversion for fixed-point arithmetic (basically, packed shifts). Vector arithmetic operations are those that process data in a purely SIMD style, performing operations such as alpha-blending, image average or offset adds. Lastly, reduction operations are those involved in reduction processes (that is, characterized for having sub-word level intra-dependences and for computing a scalar value from a set of vector data).

As we can observe, MMX-like architectures tend to be dominated by this logic overhead (around 50% by average of the overall number of non-memory vector instructions). On the other hand, the use of accumulators allows to greatly reduce the amount of logic operations required for the MDMX-like and MOM ISAs (35% on average), while the number of arithmetic operations remains similar. As a result, the ISAs that use *packed accumulators* end up being dominated by reduction operations.

As a conclusion, we have seen that non-memory vector operations in $\mu$-SIMD ISAs without packed accumulators (as our model of MMX) are dominated by the logic overhead involved in reduction processes, while non-memory vector operations in $\mu$-SIMD ISAs with packed accumulators (as our MDMX-model or the matrix ISA) are dominated by the real vector reduction operations. Attending to the number of operations to execute, MDMX-style ISAs should outperform their MMX-style counterparts; however, as we will see in the next section, the way potential parallelism is exploited has a great impact on the overall performance.

# 4 Performance Comparison

In this section we will evaluate how the different ISA styles exploit parallelism in reduction processes. First, we are going to describe the architectures used to perform our evaluations. Then, we will evaluate performance versus the number of $\mu$-SIMD physical registers available. Finally, we will analyze the effects of increasing the latency of the $\mu$-SIMD functional units.

## 4.1 Modeled Architecture

The three architectures under study model a hypothetical 8-way MIPS R10000 processor with the addition of a multimedia unit that has its own register file. Table 3 summarizes the major parameters for the three architectures. The reader will notice that the configurations are fairly aggressive. The reason is that we wish to stress the ILP available in each kernel to clearly show which architecture can extract more parallelism from its ISA.

Note that the MOM SIMD pipeline has one single functional unit, composed of 4 vector pipes, so that it is able to execute up to 4 $\mu$-SIMD MDMX-like instructions per cycle from the same MOM instruction. Additionally, the matrix SIMD processor memory sub-system is able to either provide 2 independent scalar memory references or up to 4 elements from the same vector memory reference per cycle (in contrast

| | MMX | MDMX | matrix |
|---|---|---|---|
| Fetch rate | 8 | 8 | 8 |
| graduation window | 128 | 128 | 128 |
| INT issue/# FUs | 4 | 4 | 4 |
| SIMD issue | 4 | 4 | 1 |
| SIMD FUs | 4 | 4 | 1x4 |
| memory issue | 4 | 4 | 2 |
| memory ports | 4 | 4 | 2 / 1x4 |
| Load/Store queue | 32 | 32 | 16 |

**Table 3. Processor configurations.**

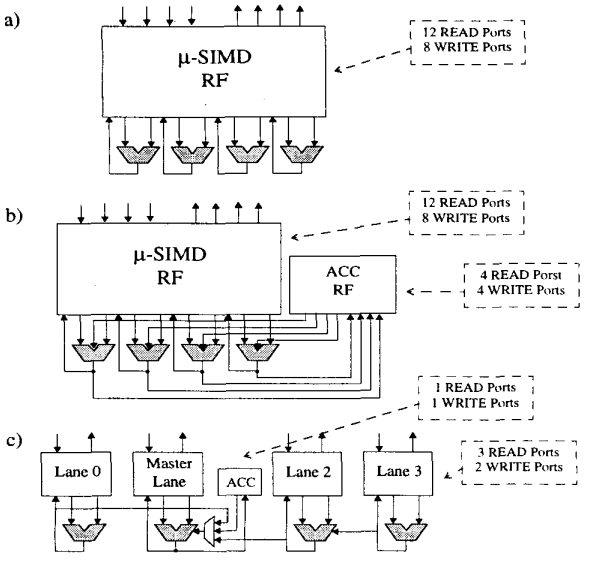


**Figure 11. The register file and FU configuration of our models of (a) MMX, (b) MDMX and (c) MOM architectures.**

with the 4 independent memory ports for both the MMX-like and MDMX-like processor models).

## 4.2 Register file organization and cost

A common misconception that often arises when describing the MOM matrix registers is that they require a much larger chip area than their MMX counterparts. The purpose of this section is to show that thanks to the organization of the MOM register file into separate lanes, we heavily reduce the number of ports required in each slice of the MOM register file. This reduction in number of ports translates into a smaller die area and a faster cycle time than its MMX and MDMX rivals.

Figure 11 shows the SIMD register file and FU configuration for our different models. The MDMX-like architecture is similar to the MMX-like one, plus the addition of a *packed accumulator* register file. Note that we have 4 read and 4 write ports for the Accumulator register file because we can perform

| | MMX | MDMX | MOM |
|---|---|---|---|
| $\mu$-SIMD RF ports | 20 | 20 | 4 x 5 |
| ACC RF ports | 0 | 8 | 2 |
| Overall RF storage (KB) | 0,50 | 0,68 | 4.04 |
| Estimated RF area cost ($wt^2$) | 2.260.992 | 2.463.744 | 2.382.336 |
| Estimated RF access time (ns) | 1,476 | 1,476 | 1.362 |

**Table 4. $\mu$-SIMD register file configurations and estimated cost.**

up to 4 operations over 4 different accumulators (assuming 4 logical *packed accumulators* and 1-cycle latency operations). Note that MOM architecture has a limited connection between lanes to perform the last series of reductions. Only one of the lanes needs to read and write the *packed accumulator* register file and thus fewer ports are required. This lane is responsible of performing the last reduction operation and writing back to the source/destination accumulator.

Table 4 shows the number of ports of the $\mu$-SIMD and *packed accumulator* register files, the overall storage size (in Kbytes), an estimation of the overall area cost and a lower bound of the access time. We have used the models described in [12] to estimate the area (in square wire tracks) and time (in ns), assuming a $0.18\mu$m CMOS process and assuming twice the number of physical registers than logical registers. From the results of the table, we observe that vector pipes are a very efficient way of providing high bandwidth at a very reduced cost. Despite having up to 8 times more storage, the matrix model overall register file size is similar to our models of MMX and MDMX, and slightly faster. As seen in [12], this is due to the special relevance that the number of ports has over the area and delay of a register file. MOM is able to distribute the register elements in a smart way, minimizing communications between clusters.

## 4.3 Influence of the number of physical registers

In this section we want to establish how many physical registers are required to support high ILP in each of the three architectures under study. We take as baseline the performance of the MMX architecture with 36 physical registers and evaluate the performance of MMX, MDMX and MOM as we increase the number of available physical registers. Note that we are *not* varying the number of physical accumulators available. Although not shown here, we did several studies and found that for MDMX, 8 physical packed accumulators were enough to support the maximum ILP. For MOM, simulations showed that 4 physical packed accumulators were also sufficient.

Figure 12 shows the Speed-up of the different ISAs under study as a function of the number of $\mu$-SIMD physical registers. Note that the graphs on the left are related to the kernels where we can find reduction operations while the graphs on the right are related to the kernels without any reduction pro-

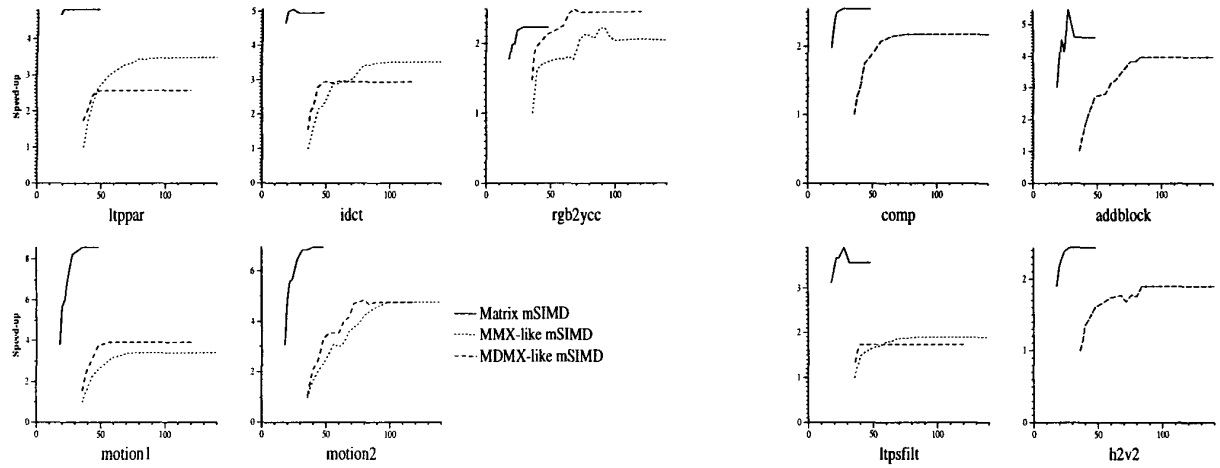


**Figure 12. Speedup of the different ISAs under study as a function of the number of physical registers (baseline: MMX-like performance with 36 physical registers).**

cess.

The results in figure 12 show that our MMX-like model achieves increases in performance between 1.9X and 4.75X when increasing the number of physical registers from 36 to 80-100. Beyond that number, performance saturates. Interestingly, our MDMX model has a behavior completely different for those programs where it can take advantage of the *packed accumulators* (for the rest, the performance is identical to the MMX one, since our MDMX ISA model is a superset of the MMX one). In the 40 to 60 range, MDMX clearly outperforms MMX. However, past 64 physical registers, MMX wins in three benchmarks, MDMX in two and `motion2` can be declared a draw. This is very surprising since in the previous section we saw that MDMX executes much fewer operations than MMX. For example, in `ltppar`, MDMX executes 28% less instructions but MMX achieves a performance 35% higher. The reason was already discussed back in section 2. The use of accumulators (no matter how many physicals back it up) severely limits the amount of ILP available to MDMX kernels. Despite eliminating most of the logic overhead, the packed accumulators hide the potential ILP inside reduction processes.

By contrast, MOM is able to take advantage of the overhead reduction capabilities of *Packed Accumulators* while, at the same time, exploiting effectively the potential parallelism of a whole reduction process. As a result, the matrix ISA outperforms the other two ISAs with performance gains ranging from 1.1X to 2.5X for MMX and from 0.9X to 2.2X for MDMX (1.5X on average for both ISAs). The only exception is `rgb2ycc` where the MDMX-like architecture outperforms the matrix ISA for aggressive configurations due to short vector lengths in MOM second dimension. We can observe that the matrix ISA saturates beyond 20-30 registers. We must realize that every matrix register contains 16 MDMX-like $\mu$-SIMDregisters. Therefore, the matrix ISA is effectively exploiting between 160 and 480 MDMX-like registers on average, clearly stressing the theoretical parallelism limits of the kernels.

## 4.4 Influence of instruction latency

Figure 13 shows the performance impact of increasing the latency of the vector functional units for every instruction and each ISA under study. The x-axis shows the offset added to the original instruction latencies while y-axis shows execution time. Again, the graphs on the left relate to kernels with reduction operations while the graphs on the right relate to kernels with none. For the MOM architecture, four different cases have been evaluated. These cases reflect the effect of two different assumptions that can be made about the matrix pipeline:

- Whether the latency increase applies to the regular or the reduction component of the instruction (for instance, in a multiply&accumulate operation, whether we increase the latency of the multiplication or rather the latency of the addition).
- Whether we fully pipeline or not the last stage of the reduction operation, where the vector pipes communicate together, reducing the last series of values following a binary-tree organization. If no pipelining support is given to this stage, an additional functional unit contention of $log_2(P) \times L_R$ cycles are introduced (where $P$ is the number of lanes and $L_R$ is the reduction latency).

The first main conclusion that may be inferred from the figure is that $\mu$-SIMD architectures are *very* latency sensitive in the presence of reductions. The MMX average performance

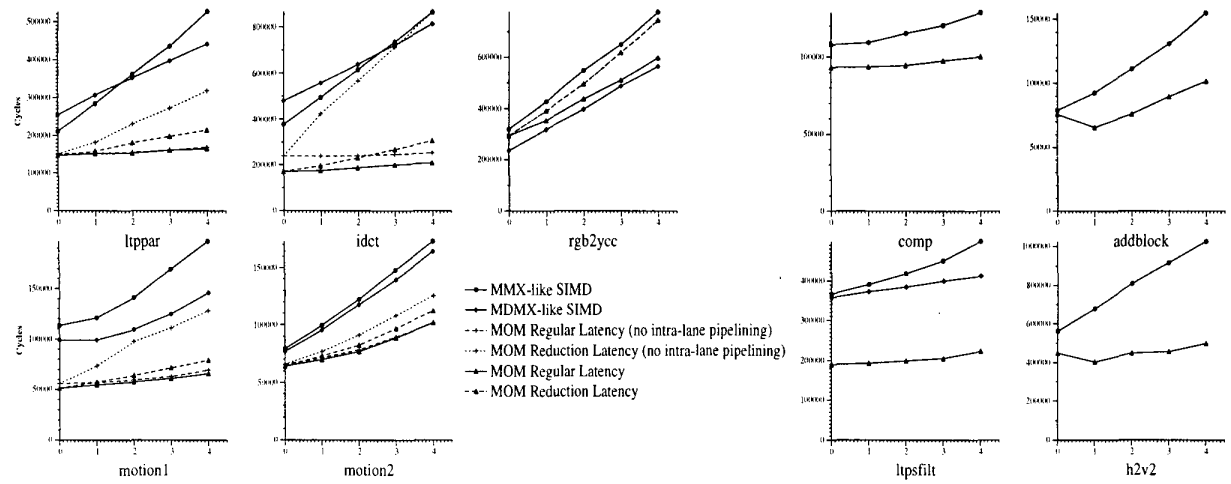


**Figure 13. Speedup of the different ISAs under study as a function of the vector functional unit latency**

slow-down for the kernels with reduction operations is 2.3X while the performance slow-down for the kernels without reductions is only of 1.5X. This enforces our claim that reduction operations are a sensible issue for performance under a scenario of increasing functional unit latencies.

Another interesting fact that can be observed from figure 13 is that MMX execution time grows slightly faster than MDMX execution time. The sensitivity to the latency is dependent on mainly two factors: the percentage of instructions to which we are increasing the latency, and the ability to overlap the latency by exploiting instruction level parallelism. While MMX-style of programming can take higher parallelism from the reduction process, the additional chain dependences produced by operations needed to perform data promotion and such, end up dominating execution. On the other hand, MDMX is dominated by scalar instructions (in sharp contrast with MMX, which is dominated by the $\mu$-SIMD instructions). Therefore, as scalar instructions become the bottleneck for MDMX, it becomes more likely that increased latencies over vectors instructions can be tolerated.

In sharp contrast with the previous results, MOM is clearly characterized for having a high tolerance to increases in the latency of the instructions. If we look at performance slow-downs when the intra-lane communication stage is fully pipelined, we may observe that MOM execution time grows slower than MMX and MDMX execution times.

This better behavior is observed not only when we increase the regular component of the latency but also when we increase the reduction component of the latency. Nevertheless, as already observed in our previous mathematical model (section 2.4), the execution time grows faster when we increase the reduction component than when we increase the regular component of the latency. For example, those benchmarks that do indeed have reduction operations see an average increase in execution time of only a factor of 1.4X when we increase the regular component of the latency in 4 cycles. For `idct`, `motion1` and `ltppar` the increase is almost flat. On the other hand, this factor raises to 1.6X when the component being increased is the reduction latency. However, this higher slow-down still compares positively with the MDMX (1.9X slow-down factor) and MMX (2.3X slow-down factor) counterparts. The only exception to this trend is, again, `rgb2ycc`. Due to the low matrix register lengths achieved in this kernel, the advantages of MOM when performing reductions cannot be exploited at their full potential.

When we assume that no pipelining support is provided to the intra-lane communication hardware of the MOM functional units, we can observe a radically different behavior when we increase the reduction component of the latency. In that case, MOM execution time grows at a similar rate than MDMX –even higher for two of the benchmarks, *motion1* and *idct*. This is a clear proof of the effect of functional unit contention over the final performance. While MOM is still able to tolerate latencies, for higher enough reduction latencies, the contention ($log_2(P) \times L_R$) exceeds the number of elements inside a MOM register, and, thus, the functional unit usage drops radically.

## 4.5 Full application performance comparison

In the previous sections we have studied kernels in isolation to better analyze the effect of reductions. Now we address the issue of evaluating the performance for full applications and with highly accurate memory model simulation (a 32 Kb multi-ported, multi-banked direct-mapped L1 data cache, backed up with a 1Mb, 2-way set associative L2 cache). Figure 14 shows the evolution of normalized execution cycles when we increase the regular latency of the $\mu$-SIMD instructions 1, 3 and 5 cycles for both MMX-like and MOM architectures. For the latter, we have assumed non-pipelined intra-lane communication. Clearly, from the results in the figure, we can

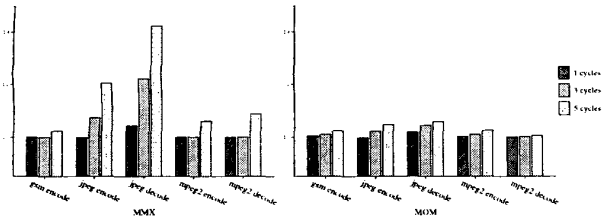

**Figure 14. Increase of the number of execution cycles (normalized) when increasing the regular component of the latency of MMX and MOM non-memory instructions (no intra-lane pipelining).**

corroborate that the observed behaviors at kernel level are also shown in complete programs.

As we can observe, while the MMX-style ISA can suffer severe increases in the execution time (up to a 40% of slow-down) the matrix $\mu$-SIMD architecture exhibits very low performance slow-downs (less than 5%) even for increases in the latency exceeding 5 cycles. Therefore, MOM appears as a suitable alternative for next-generation of media processors as it overcomes the problem of the latencies of complex $\mu$-SIMD style instructions. MOM shows a high robustness to increases in both the regular and reduction components of the latency of any instruction, provided that full pipelining support is given to all the stages of the matrix functional unit.

## 5 Summary

In this paper, we have studied the problems associated with reductions found in current multimedia applications. Reductions pose a particular problem for $\mu$-SIMD architectures such as MMX or AltiVec, where issues such as the size of the sub-word data types or intra-register dependencies are critical for performance.

Data presented have shown that although reductions account for a small percentage of total instructions (less than 5%), their impact on final application performance can be much larger (up to 40% degradation in `jpeg decode`) when latencies are increased.

Given the current trend towards ever-increasing clock frequencies and hyper-pipelining, we believe latencies of complicated multimedia instructions are bound to increase. Hence, this paper has studied two potential solutions to the reduction problem.

First, we have described and studied the performance achieved by *packed accumulators* (as found in MDMX). Packed accumulators solve the packing and unpacking problem typically associated with reductions and thereby reduce instruction overhead by as much as 30%. Unfortunately, they introduce artificial recurrences that hide potential ILP, providing diminishing returns for aggressive configurations. We have seen that even with the fact that MDMX-like ISAs can execute up to 30% less instructions, MMX-like versions can outperform them by a up to 35%.

Second, we have described and evaluated a combination of the packed-accumulator idea and the matrix ISA described in [8]. The longer registers present in the matrix architecture allow using packed accumulators without incurring into the recurrence penalties. By properly scheduling operations onto the accumulator, the matrix ISA can hide the intrinsic latency of the accumulator and exploit operation parallelism. Our simulations show that the matrix ISA outperforms the other two ISAs under study by an average of 1.5X. Furthermore, as we advance into the future and longer latencies become the norm, matrix ISAs turn even more attractive: for example, when increasing latency by 4 cycles, our model of matrix ISA speedups increase to 2.2X over MDMX and MMX.

## References


[1] A. Peleg and U. Weiser. MMX technology extension to the intel architecture. *IEEE Micro*, pages 43–45, August 1996.

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

[3] K. Diefendorff, P.K. Dubey, et. al. Altivec extension to powerPC accelerates media processing. *IEEE Micro*, March-April 2000.

[4] 3DNow! technology manual. Technical Report http://www.amd.com, Advanced Micro Devices, Inc., 1999.

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

[6] http://developer.intel.com/design/processor/index.htm. Willamette Architecture Software Developer Manuals. Intel, 2000.

[7] M. Tremblay, J.M. O'Connor, V. Narayanan, and L. He. VIS speeds new media processing. *IEEE Micro*, August 1996.

[8] Jesus Corbal, Roger Espasa, and Mateo Valero. Exploiting a new level of DLP in multimedia applications. *MICRO*, 1999.

[9] Jesus Corbal, Roger Espasa, and Mateo Valero. MOM: Instruction set architecture. Technical report, UPC, 1999.

[10] C. Lee, M. Potkonjak, and W.H. Magione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communication systems. *MICRO 30*, 1997.

[11] A. Srivastava and A. Eulace. Atom: A system for building customized program analysis tools. PLDI, ACM SIGPLAN'94

[12] S. Rixner, W.J. Dally, B. Khailany, P. Mattson, U. Kapasi, and J.D. Owens. Register organization for media processing. *High Performance Computer Architecture, HPCA-5*, pages 375–386, 2000.

[13] R.E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, pages 24–36, March-April 1999.

[14] James E. Smith. Notes on First Order Linear Recurrences. Technical report, Cray Supercomputers, 1993.

[15] Y. Tanaka, K. Iwasawa, et. al. Compiling techniques for first-order linear recurrences on a vector computer. *Supercomputing'88*

[16] D.J.Kuck and R.A.Stokes. The Burroughs Scientific Processor (bsp). *IEEE Transactions on Computers*, pages 363–376, May 1982.

[17] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley Publishing Company, 1991.

[18] A. Kunimatsu, N. Ide, and T. Sato et. al. Vector unit architecture for emotion synthesis. *IEEE Micro*, pages 85–95, March-April 2000.