MOM: a Matrix SIMD Instruction Set Architecture for Multimedia Applications

Jesus Corbal Roger Espasa Mateo Valero

Departament d'Arquitectura de Computadors,
Universitat Politècnica de Catalunya–Barcelona, Spain
e-mail: {jcorbal,mateo,roger}@ac.upc.es

Abstract

MOM is a novel matrix-oriented ISA paradigm for multimedia applications, based on fusing conventional vector ISAs with SIMD ISAs such as MMX. This paper justifies why MOM is a suitable alternative for the multimedia domain due to its efficiency handling the small matrix structures typically found in most multimedia kernels. MOM leverages a performance boost between 1.3x and 4x over more conventional multimedia extensions (such as MMX and MDMX), which already achieve performance benefits ranging from 1.3x to 15x over conventional Alpha *code. Moreover, MOM exhibit a high relative performance for low-issue rates and a high tolerance to memory latency. Both advantages present MOM as an attractive alternative for the embedded domain.*

1 Introduction

It is widely accepted that *media applications* will become one of the most significant computing workloads in the next years [1, 2]. Realizing this trend, the major vendors of general-purpose processors have included multimedia extensions based on the SIMD (Single Instruction Multiple Data) paradigm. Examples of this are: Intel's *MMX* [3], Motorola's *Altivec* [4], SUN's *VIS* [5] or Mips *MDMX* [6]. Also, more recently, as 3-D graphic applications have become more important, additional extensions have been included to deal with FP SIMD parallelism (*Altivec*, *3DNow!* [7] and *KNI* [8]).

The goal of these architectural extensions was to take advantage of the high levels of data level parallelism found in typical multimedia codes yet, at the same time, minimize the number of changes required to the existing processor cores. Because most of the multimedia kernels operate on very small data types (typically 8 or 16 bits), a conventional 64-bit register can hold a number of these smaller data items. Thus, a conventional register can be thought of as implementing a very short vector register (between 4 and 8 elements per vector, depending on data size). The initial approach taken by most vendors is to exploit this intra-word (or intra-register) parallelism by mapping these short vectors onto conventional registers (such as the FP registers as in Intel's *MMX*) and including to the functional units features to understand the concept of sub-word parallel operation. Figure 1 shows a typical example of MMX operation (parallel add). A parallel add operation can be easily configurated in a basic adder by simply not propagating the carry bits between sub-word element boundaries. More recently, however, the Altivec extension has taken a more aggressive approach, by providing completely separate multimedia registers and making them 128 bits (twice the size of ordinary registers).

As more demanding multimedia applications reach the market, how can we keep improving the performance of the present multimedia units ? Clearly, the fact that current efforts exploit only sub-word parallelism limit the available exploitable parallelism to a factor of 8 (and then, only for the smallest 8-bit data size). Therefore, if we are to break beyond this limit, we need to take a similar approach to Altivec and increase the number of bits available in a register. However, going to a register file where each *single* register has 256, 512 or 1024 bits does not seem a plausible alternative. MMX-like extensions are restricted by the stride-one constraint (that is, all the elements of the vector must be consecutively arranged in memory). Therefore, as long as we increase the width of the multimedia register, we find that we are not enable to load so many arranged sub-word elements as the size of the multimedia register, thus achieving diminishing returns.

Since the current multimedia extensions are a particular case of a vector architecture, could traditional vector ISAs help us in breaking this parallelism limit? Traditional vector architectures have been used for many years for high performance numerical applications [9, 10, 11, 12] and, more recently, have also been used in some neural and DSP applications [13]. The current multimedia extensions are

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

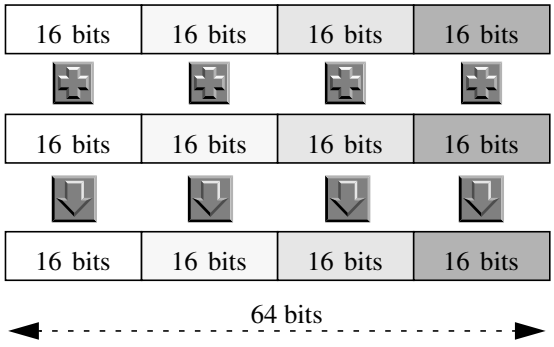
nothing more than a somewhat limited ISA vector extension with a fixed vector length (limited by the size of the data types) and a fixed vector stride (always consecutive memory locations).

There have been some studies evaluating the performance of conventional vector ISAs on typical multimedia codes. Lee et. al. [14] evaluated for a set of kernels the performance of an out-of-order superscalar processor with very short vector instructions (length 8) against a highly parallel, simple in-order conventional vector machine (length 64), achieving better performance with the vector architecture. Despite these results were encouraging for simple *kernels*, when looking at full-blown applications (such as the complete `mpeg` decoder, not just its `idct` kernel) the following problems arise [15]:

- The vector length of real multimedia applications is typically small (between 8 and 16). In [14], this problem was not encountered because the authors applied loop interchange techniques to increase the effective vector length in each kernel. However, this technique is not effective when dealing with full applications.
- Conventional vector ISAs are not well suited to handle sub-word level parallelism as MMX-like vector ISAs. Therefore, a major loss of performance is produced due to the overhead produced by operations such as data pack/unpack, saturation, and/or data promotion.

As a conclusion, while performance benefits may arise from the use of conventional vector ISAs, MMX-like multimedia extensions are much more well suited for achieving higher performance at lower cost (as reported in [16] or in [17]). As an example, a MPEG encode application is evaluated in [17] using the VIS instruction set with a reported performance gain of 3.5x approximately. On the other hand, the same application is evaluated in [15] with an out-of-order vector processor with CONVEX ISA, and for the most aggressive architectures (up to 16 parallel vector pipes) only a 2x performance gain is reported.

Our claim is that significant levels of Data Level Parallelism (i.e. that parallelism where the same operation is performed over several data) still remains to be exploited in typical multimedia codes. MMX-like and conventional vector ISAs are only able to exploit DLP along one single dimension (that is, one single loop). We will show that by fusing both of best worlds, we will be able to exploit DLP from two different dimensions (i.e. two nested loops) extracting a parallelism of an order of magnitude higher. The most interesting point is that matrix operations occur frequently in multimedia applications. and our claim is that matrix ISAs are the best alternative to efficiently exploit this available parallelism.



(A)

Figure 1: MMX Packed ADD example.

2 Rationale of a matrix ISA

MOM stands for *Matrix Oriented Multimedia* extension, and has been designed trying to exploit the advantages of both conventional vector and MMX-like ISAs. In order to understand the basis of this novel ISA, we may look at figure 2 where a comparison between the three different paradigms under study, traditional-vector, MMX-like and MOM-like, are applied to a very simple code example. Despite being simple, those kind of kernels are commonly found in most multimedia codes (specially in image processing).

Let's define *dimension X* as a dimension where sub-word level parallelism is exploited as in MMX-like ISAs, and let's define *dimension Y* as a dimension where parallel operations are expressed in a sequential way (typical of conventional vector ISAs). From the figure, we can see than the MMX-like ISA approach vectorizes the inner loop (j) over *dimension X*. Note that this vectorization is done under two restrictions: fixed vector length and fixed vector stride (consecutive accesses). On the other hand, the conventional vector ISA approach vectorizes the inner loop over *dimension Y*, without any restriction of vector length and stride. Finally, MOM approach vectorizes the inner loop (j) over *dimension X* and vectorizes the outer loop (i) over *dimension Y*. Note that, since columns are usually located at consecutive locations and since any vector stride is allowed under *dimension Y*, we can access in a single instruction a whole matrix. So. the MOM vectorization process is decoupled into two main steps: (a) generation of MMX-like operations over the inner loop, and (b), vectorization of those MMX-like operations over the outer loop.

Therefore, dealing with the two kinds of vectorization

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

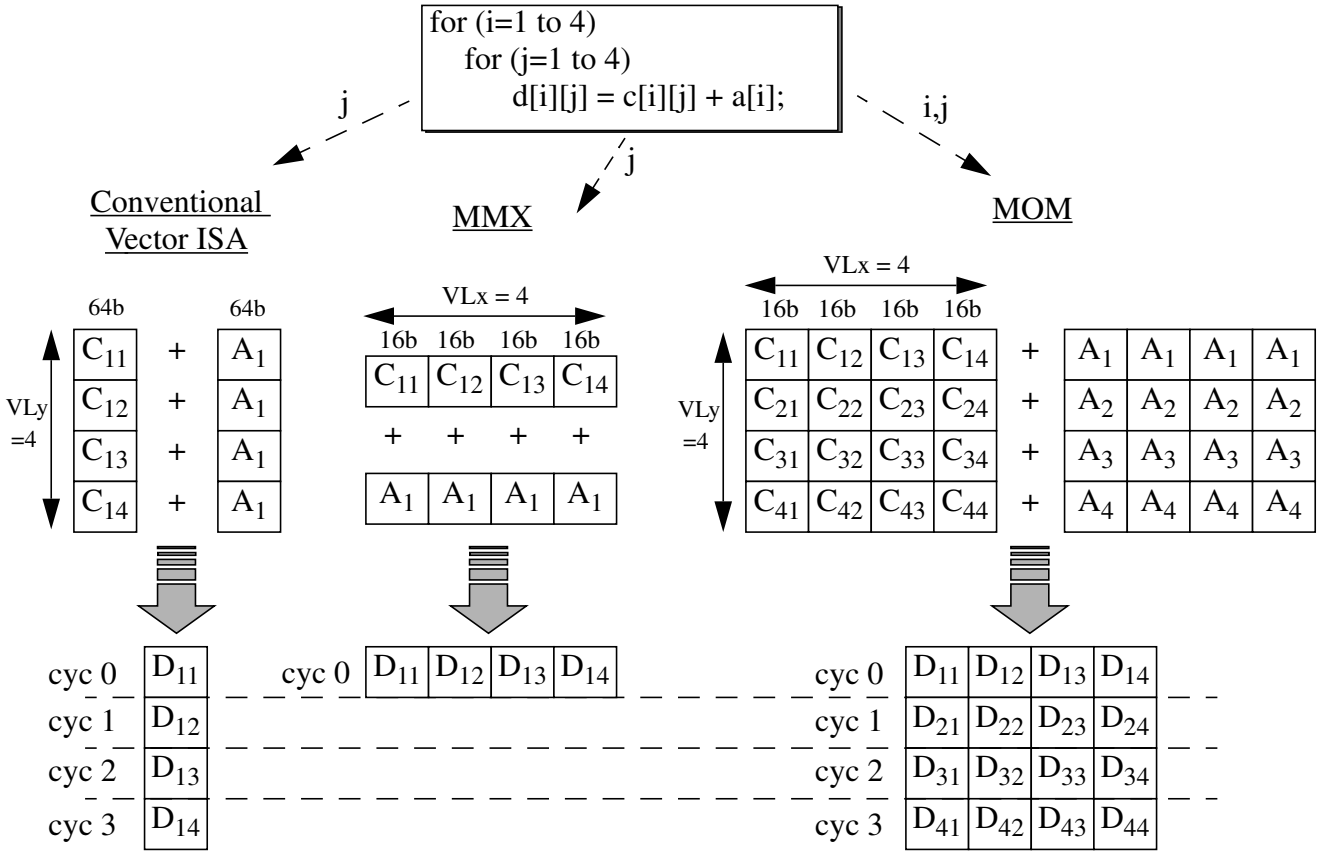


Figure 2: Comparison between conventional vector, MMX-like and matrix oriented ISAs

over each dimension allows MOM to add the advantages of conventional vector ISAs to the advantages of MMX-like instructions. The MMX-like operations allow MOM to take benefit from the sub-word level operation capabilities plus the multimedia oriented features (such as saturation), while the streaming nature of the vectorization of those MMX-like operations provides: a reduction of the pressure over the fetch unit, a capability of tolerating long latencies over long streams of MMX-like instructions, and the possibility of further parallelism by replicating the number of functional units that are fed from the same vector register.

Our claim is that the capability of MOM of dealing with matrix rather than with simple vectors allows it to pack an order of magnitude more elements than common vector ISAs. This is due to the fact that multimedia codes commonly deal with matrix structures of limited size (no longer than 8/16 element per row).

Of course, matrix ISAs are not a new idea. There is a large body of research on array multiprocessors, some of which made it into products for the high-end supercomputer market [18] and there is also current research looking in matrix-based ISAs to be used in floating point applications [19]. The key difference between these previous proposals and MOM is that, in the context of floating-point engineering-like applications, a matrix ISAs is not that much of an advantage over a traditional vector ISA. Thus, all products based on array processors eventually disappeared from the market due to the prevalence of vector supercomputers. However, in the multimedia domain, traditional vector ISAs are not good to exploit the fine grain parallelism available, yet MMX-like ISAs are not able to exploit the inter-word parallelism that traditional vectors offer. Hence, in the multimedia domain, a matrix-oriented ISA, resulting from the fusion of inter- and intra-word parallelism, offers many advantages over other alternatives.

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

3 An Overview of MOM

MOM is a load/store architecture where its memory accessing operations closely follow those available in traditional vector computers, while its computation operations resemble MMX-like instructions. Basically, a matrix ISA should provide 4 architectural registers: one *Vector Length* and one *Vector Stride* register per each matrix dimension. The Vector Length determines the number of elements for that dimension, while the Vector Stride determines the distance between consecutive elements. This constraints are simplified in MOM due to the fact that the MMX-like dimension has restricted vector length (limited by the data size, for instance 8 elements of 8 bits) and vector stride (always consecutive memory locations). Therefore, MOM needs only to provide one Vector Length architectural register and one Vector Stride (which in this case is mapped as a conventional integer register). The size of the matrix is specified by this vector length register and by the data size of the MMX elements inside each vector element.

MOM provides to the programmer 16 matrix registers, each register holding 16 64-bit words. Since each word in the vector register can be thought of holding either eight 8-bit items or four 16-bit items or 2 32-bit items, we view these registers as holding matrices of data. The maximum size of the matrix is, thus, of 16x8 elements.

Memory instructions

MOM offers a vector load and vector store instruction to move data in and out of the MOM registers. These instructions have the following form:

```
mom_ldq  MRi <-- Rj, Rk
```

Where MR_i is one of the 16 MOM registers, R_j is the base address where the load starts and R_k is the *vector stride* register. As in traditional vector ISAs, the execution of this instruction is controlled by a fourth implicit register, VL or vector length register. The semantics of the instruction is 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 stores work in a similar fashion. Therefore, if we have a memory port of wide N, every cycle, N elements separated by R_k bytes are loaded onto N consecutive vector elements. So, the number of cycles required to perform a whole memory operations is Vl/N.

Arithmetic and logic instructions

These instructions are straightforward conversions from typical MMX-like instructions to matrix versions of them (that is, vector/stream versions of MMX instructions, where each single operation of a vector instruction is independent from the others). A typical example would be:

```
mom_paddb  MRi <-- MRj, MRk
```

which does a parallel byte add between each element of the MOM registers j and k leaving the result in MOM register i. The length of the operation is controlled, again, by the VL register. Thus, this example could result in anywhere from 1 element to 16 elements of the involved MOM registers being added. Therefore, if we have a vector functional unit with N vector pipes or lanes, N MMX-like vector elements would be operated each cycle, so that, a whole arithmetic/logic vector instruction would be performed in Vl/N cycles.

Matrix special instructions

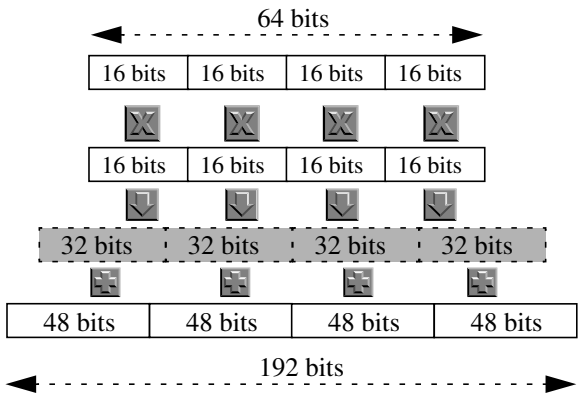
MOM semantics allow very powerful matrix operations. Those instructions are characterized by the fact that there are inter-dependences between elements of the same matrix register in the same way that some MMX-like instruction have inter-dependences between its sub-word level elements. Usually, those inter-dependences are related to reduction operations. MOM deals with this kind of instructions using packed accumulators, which is feature that we will address later in the paper. These accumulators allow to define very powerful operations to be performed in a single matrix instruction (such as a matrix per vector operation).

Another kind of MOM instructions are related to matrix management. For instance, MOM has available a *Matrix Transpose* instruction that allows to transpose a 8x8 matrix with only $8 + C$ cycles of latency and relatively simple logic (despite the fact that this operation is non pipeleneable). *Matrix Transpose* is a powerful mechanism to switch vector dimensions (while MMX/MDMX require several pack/unpack instructions in order to do the same).

3.1 Accumulators in MOM

MMX-like ISAs tend to have difficulties 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 is that, if one tries to do the product in parallel (using intra-word parallelism) the result *does not* fit into a normal register. As an example, figure 3 shows that trying to multiply four 16-bit quantities yields a result that only fits in a 128-register.

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE



(B)

Figure 3: MDMX Packed Multiply&Add with accumulator.

Since these results must be added together *before* the truncation of the result is done (or, otherwise, the loss of precision is unacceptable) MMX-like ISAs end up using data promotion techniques (i.e. promotion of data to larger data sizes by pack/unpack operations) to maintain the required precision. While this solution handles the precision problem, it incurs in a clear overhead and reduces the available sub-word level parallelism.

A very efficient way to deal with reduction operations is the technique introduced by MDMX (Mips). MDMX proposes using packed accumulators (similar to what several DSP architectures do), which are wide registers that successively accumulate the results produced by operations done with multimedia vector registers (see figure 3). Finally, the results from the accumulator are truncated, clipped and conveniently rounded into a conventional MDMX register. Note that the use of accumulators rotate the dimension where the reduction operation is done. In MMX code, a dot product reduction would be done over the *dimension X*, while in MDMX code, the dot product reduction would be done over the *dimension Y* in sequential steps (exploiting thus several independent dot products in *dimension X*).

Therefore, the use of the MDMX model of computation presents two clear advantages: does not reduce the potential sub-word level parallelism and provides high precision. Unfortunately, the accumulators present a critical problem: introduce an artificial recurrence due to the fact that any operation over one accumulator needs the previous value as an input. So, when we face long latency operations, we find poor ILP, thus reducing overall performance. Therefore, it can be argued that by not using packed accumulators (such as is MMX) we allow higher scalability when we increase the number of SIMD functional units and registers.

By contrast, MOM can take great advantage of the multimedia accumulators, since it can provide parallelism over *dimension Y* (the dimension where MDMX has troubles). Therefore, in a similar way that *Sum* operations are performed in conventional vector machines, we can pipeline the reduction operation to avoid the problem of the recurrence though adding some additional cycles of latency (which stream instructions can tolerate anyway).

4 Performance Results

In this section we will present some performance results in order to evaluate the potential performance gains achievable by MOM when compared with MMX and MDMX performance. We will study topics as the scalability of each ISA (by simulating out-of-order processors of different issue rates) and the memory latency tolerance. We will also try to decouple the overall Speed-up into three different factors of performance in order to evaluate the advantages of MOM.

This is the first paper evaluating MOM. So, we are more interested in the implicit characteristics of the ISA rather than in micro-architectural issues. Therefore, as a preliminary study, this paper is going to concentrate mostly on kernels, because it is easier to understand the performance gains of MOM on these shorter sequences of code. Of course, working on kernels raises a number of issues about the broad applicability of MOM to real applications. We have strived to *not* artificially enlarge the vectorization dimension of the kernels under study (dimension Y in the terminology of section 2). This ensures that whenever these kernels are plugged back into real applications, the speedup obtained should be the speedup reported here.

Additionally, we are not going to model the memory hierarchy, but an idealized memory system with no bandwidth restrictions and fixed latency. We understand that memory issues are a key factor in the final performance of the system, as well as the fraction of the program that is vectorizable (due to Amdahl's Law). We plan to incorporate both into our framework in the near future.

4.1 Benchmarks and Simulation Tools

We have studied six different programs from the *Mediabench* [20] suite: `mpeg encode`, `mpeg decode`, `jpeg encode`, `jpeg decode`, `gsm encode` and `gsm decode`. For each program, we identify the most important functions using profiling and we implement them using the three styles of ISAs under study: MMX-like, MDMX-like and MOM. For this study, we will study the performance

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

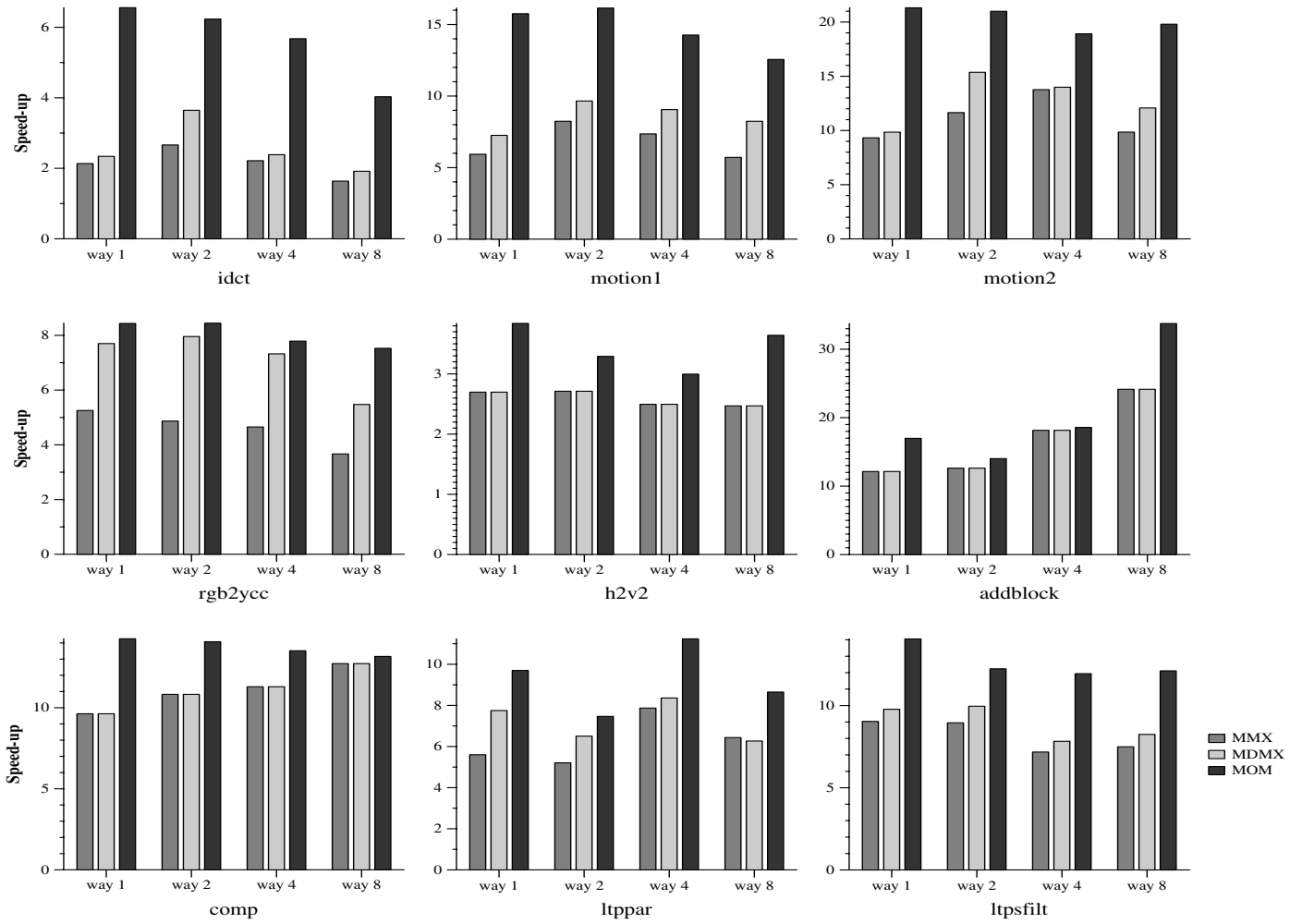


Figure 4: Speed-up of evaluated multimedia ISA for different issue-rate machines (with respect to *Alpha* ISA performance).

of nine of the different kernels so identified: `idct`, `motion1`, `motion2`, `rgb2ycc`, `h2v2upsample`, `compensation`, `addblock`, `ltpparameters` and `ltpfiltering`. The `idct` kernel performs a Inverse Discrete Cosines Transform over 8x8 matrices of data, `motion1` performs a *sum of absolute differences* between two 16x16 image matrices to perform a MPEG2 motion estimation operation while `motion2` performs a *sum of quadratic differences*. The kernels `addblock` and `compensation` are basically special forms of saturated blending between image matrices used in the Motion Compensation algorithm, while `ltpparameters` and `ltpfiltering` are special dot products used to calculate the long term filter parameters and the filtering itself in the gsm encoding/decoding process. Finally, `h2v2upsample` performs a 2x2 zoom over the whole image for the *JPEG decode* benchmark while the `rgb2ycc` kernel transforms a RGB image into YCC format.

Since there is no available compiler that can generate either MMX or MDMX (let alone MOM), each function was manually optimized by including function calls to routines that represent MMX, MDMX or MOM instructions. Then, a library was written to emulate each such instruction (67 MMX instructions emulated, 88 MDMX instructions emulated and 121 MOM instructions emulated). Once the program was written using these subroutine calls, the correctness of the output was verified to avoid losses in accuracy visually perceptible.

We note that we have not considered absolutely exact models of MMX and MDMX, but fair approximations of each ISA. Additionally, we have enhanced these ISAs by providing independent register files and an increased number of logical registers (32 logical vector registers for MMX, plus 4 logical accumulators for MDMX, and 16 logical matrix registers plus 2 logical accumulators and one logical *Vector Length* register for MOM). The maxi-

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

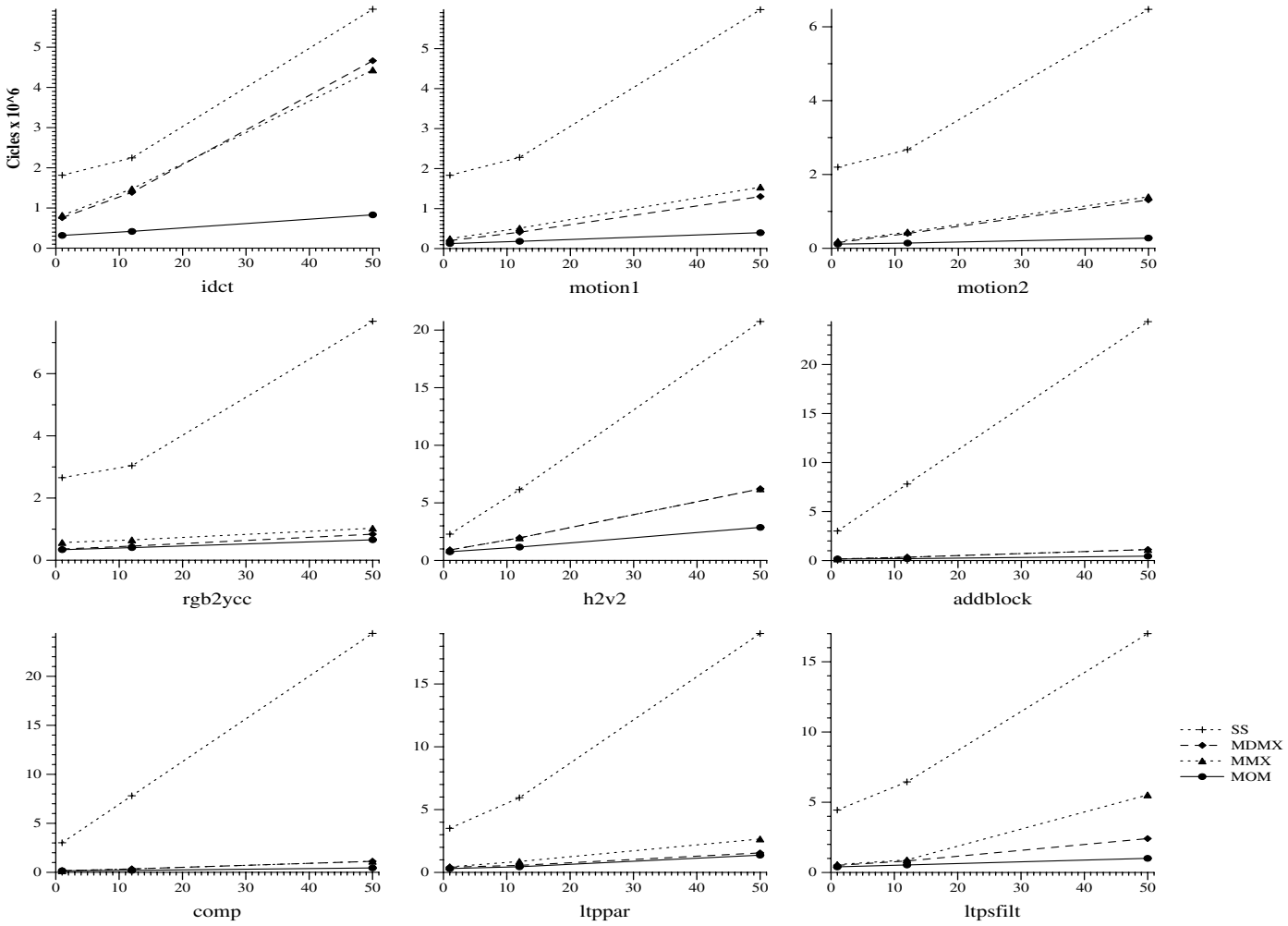


Figure 5: Impact of the memory latency on performance.

mum vector length on *dimension Y* has been set to 16. In order to maximize the performance of MMX and MDMX, we have used loop-unrolling and software pipelining techniques. These changes were done in order to avoid an unfair comparison with MOM, whose streaming nature provides a natural mechanism of instruction scheduling.

Once the applications are tested using the emulated instruction subroutines, the question is how to feed this application into a simulator such that the subroutine calls that emulate an MMX/MDMX/MOM instruction disappear and, instead, we only model the effect of the instruction itself. What we did was to compile the applications using the emulation libraries at only -O1 level, instrument with *ATOM* [21] and feed the output to *Jinks* [22]. Jinks is an out-of-order simulator with capability of executing vector ISAs. The basic architecture closely resembles that of the MIPS R10K, with the addition of a MMX/MOM register file and dedicated functional units. Instructions are fetched and sent to the decode stage where they are renamed. There are three different rename tables: one for integer registers, one for floating point registers and one for multimedia (MMX/MDMX/MOM) registers. Once renamed, the instructions go to the appropriate queue where they wait until their operands are ready and then arbitrate for a free functional unit. Instruction in every queue can execute out-or-order. Jinks is able to detect the subroutine calls, filter them out and translate them into a single instruction. For the baseline case (the pure superscalar code with no multimedia extensions), we used -O4 optimization. In all cases, the compiler was the DEC C V5.8-009.

Each kernel has been simulated a certain number of times in a loop so that we simulate between 5 and 10 million graduated instructions for the plain superscalar version. This assures minimal impact of the initialization code for every kernel.

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

4.2 Performance and scalability

Figure 4 shows the speed-up attained by the three multimedia ISAs evaluated when compared with *Alpha* code, for different wide machines and for the nine kernels studied. We have considered an idealized memory system with no bandwidth constraints and a fixed 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 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. 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. As the fetch and issue rate increase, the out-of-order exploitation of ILP partially compensates the advantage of MOM.

4.3 Tolerance to memory latency

Although we have not modeled a cache hierarchy, we have performed some experiments trying to approximate the behavior of our kernels under realistic conditions. The experiments vary memory latency from 1 cycle (as in the previous section) to 12 and to 50 cycles. These values are chosen trying to mimic what would be an application that misses to the L1 cache but hits in the L2 (12 cycles) and an application that is simply streaming through data and, therefore, without prefetching, is always missing in the on-chip caches and going to main memory (50 cycles latency). In these experiments, we only use the 4-way processor core.

Figure 5 shows the results of these experiments for the nine kernels studied. The results obtained show that MOM exhibits a high tolerance to increases of the memory latency. When we increase the memory latency from 1 to 50 cycles, we observe slow-downs ranging from 4x to 8x for MMX/MDMX and ranging from 3x to 9x for common *Alpha* code. In sharp contrast with these results, MOM only suffers a slow-down ranging from 2x to 4x. Clearly, MOM is better suited for pure streaming applications than its counterparts. In fact, this is not a surprising feature of MOM, since vector ISAs are very well known for their latency tolerance capabilities.

4.4 Sources of performance gain

Under architectural assumptions, MOM doubles MMX/MDMX performance with hardly any additional hardware: we only need a bigger register file (about 6 times more, and without increased complexity, since, as we have observed in our evaluations, MOM requires a much more low number of physical registers). In order to evaluate why MOM achieves such performance improvements, we will try to decouple the execution time into three different parameters: (a) the IPC (Instructions per Cycle), (b) The OPI (Operations per Instruction), and (c) the NOPS (the overall Number of Operations).

The execution time in cycles can be easily described as the number of instructions divided by the IPC rate. Nevertheless, while in conventional ISAs one instruction is equivalent to one operation, in SIMD ISAs, we can perform several operations per instruction. Therefore, the number of cycles could be described as follows:

$$Cycles = \frac{NI}{IPC}$$
$$Cycles = \frac{NOPS}{IPCxOPI}$$

Taking the overall number of operations of the *Alpha* code as a reference, we could define R as the factor of reduction of the number of operations required. The higher the value of R, the higher the semantic richness of a certain ISA (that is, the better the ISA is reducing unnecessary operations). MMX-like codes use to have higher values of R due to small-data manipulation facilities such as saturation or data promotion. MOM adds another level of overhead reduction due to the elimination of loop control instructions and operations on address registers (well-known characteristic of any vector ISA).

Therefore, the Speed-up of any ISA (compared with *Alpha* ISA) can be calculated as follows:

$$S = \frac{NOPS_{alpha}/IPC_{alpha}}{NOPS_{isa}/(IPC_{isa}xOPI)} = \frac{RxIPC_{isa}xOPI}{IPC_{alpha}}$$

As a consequence, it can be seen that the Speed-up is a direct function of the IPC, the factor of reduction of operations (R), and the number of operations per cycle (OPI). If we define F as the percentage of vector instructions, OPI could be calculated as follows:

$$OPI = (1-F) + Fx(Vl_x * Vl_y)$$

where Vl_x and Vl_y are the average vector length of dimension X and dimension Y, respectively.

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

Tables 1 to 9 show the previous parameters for a 4-way machine for all the kernels under study (assuming latency 1). the parameters of the table are: (IPC) instructions committed per cycle, (OPI) operations performed per instruction, (R) reduction of the number of overall operations (related to *Alpha* code), (S) Speed-up, (F) Percentage of vector instructions, (Vl_x) average vector length over dimension X (that is, number of sub-word level elements packed by average), and (Vl_y) average vector length over dimension Y (that is, the average vector length of MOM operations).

Some IPC results from the tables may seem somewhat surprising. For some of the kernels, MMX and MDMX IPCs are even higher than those of the plain superscalar version. Initially, this may seem counterintuitive, since MMX/MDMX are packing instructions exploiting DLP and thus they are limiting the number of operations exploitable as ILP. Nevertheless, we should take into account that we are comparing compiled code against highly tuned hand-written code. Compilers always generate not optimized code due to their inability to handle features such as memory disambiguation and register allocation in a optimal way. Additionally, MMX/MDMX code take advantage of the loop-unrolling plus software pipelining techniques applied, carefully tuned for a 4-way machine. On top of everything, the latency of MMX/MDMX instruction (specially multiply operations) are much shorter than the full 64-bit superscalar instructions, factor that may limit the parallelism in the plain superscalar code.

Another interesting point is that the reduction of the number of required operations is a very significant factor of performance improvement. There are many factors that explain this reduction. Firstly, the MMX-like vectorization reduces many integer instructions related to control overhead (that is, managing of addresses and loop induction variables). This phenomena is also exploited by the loop unrolling techniques applied over the inner loop. Additionally, some special instructions of MMX/MDMX allow to reduce the overall number of operations (typical examples are the saturated arithmetic and the multiply&accumulate features). MOM leverages an additional reduction of the number of overall operation required due to the elimination of control overhead (that is, address registers and loop induction variables managing) over the outer loop.

From the results of the table, if we define OPC (Operations per Cycle) as the product of the IPC and the OPI, we can argue that MOM achieves higher performance due to two main reasons:

- A better overall instruction schedule (since it is able to execute the highest number of operations per cycle) even if loop-unrolling and software pipelining techniques are used in MMX/MDMX codes.
- The highest reduction of overhead operations by combining the advantages of MMX-like and conventional vector ISAs.

Note that these performance results do not take into account the fact that additional performance could be achieved under MOM by simply replicating the number of parallel functional units which execute a matrix instruction. By contrast, if MMX or MDMX tried to do similarly they would become limited by the fetch unit, as seen in [15], and, therefore, could not take advantage of these extra functional units.

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 (such as MMX or MDMX).

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 small matrices structures typically found in several multimedia kernels. Our proposed ISA handles very efficiently the accumulators proposed by the MDMX multimedia extension, providing both precision and parallelism. The implementation of this matrix ISA would not have a huge impact on an out-of-order core with any kind of conventional multimedia extension, as only a bigger multimedia register file would be required.

We have evaluated nine kernels corresponding to some of the main functions of five programs of the *Mediabench* suite and improvements between 1.3x to 4x have been observed for a 4-way machine with working sets fitting in L1 cache. Additionally, we have shown that MOM presents two clear advantages that present it as a suitable alternative for multimedia embedded systems: a high relative performance for low-issue architectures and a high tolerance to memory latency. On top of everything, further performance may be achieved by simply replicating the functional units that are fed from the same vector register, increasing the number of operations done per cycle without any need of increasing the fetch/issue rate.

References

[1] T.M. Conte et. al. Challenges to combine general-purpose and multimedia processors. *IEEE Computer*, pages 33–37, Dec 1997.

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

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

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

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

[5] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS Speeds New Media Processing. *IEEE Micro*, pages 10–20, August 1996.

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

[7] 3dnow! technology manual. Technical Report http://www.amd.com, Advanced Micro Devices, Inc., 1999.

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

[9] R.M.Russell. The cray-1 computer system. *Communications of the ACM*, 21:63–72, January 1978.

[10] W.Oed. Cray y-mp c90: System features and early benchmark results. *Parallel Computing*, 18:947–954, August 1992.

[11] et. al. Katsuyoshi Kitai. Distributed storage control unit for the hitachi s-3800 multivector supercomputer. *International Conference on Supercomputing (ICS)*, pages 1–10, July 1994.

[12] *CONVEX Architecture Reference Manual (C Series)*. Convex Press, Richardson, Texas, U.S.A, 1992.

[13] Krste Asanovic et. al. The to vector microprocessor. *Hot Chips*, VII:187–196, August 1995.

[14] Corinna G. Lee and Derek J. DeVries. Initial results on the performance and cost of vector microprocessors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 171–182, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[15] Francisca Quintana, Jesus Corbal, Roger Espasa, and Mateo Valero. Adding a vector unit on a superscalar processor. *International Conference on Supercomputing*, Available at http://www.ac.upc.es/homes/roger/papers/list.html, June 1999.

[16] Huy Nguyeni and Lizy Kurian John. Exploiting simd parallelism in dsp and multimedia algorithms using the altivec technology. *International Conference on Supercomputing*, 1999.

[17] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. Performance of image and video processing with general-purpose and media isa extensions. *International Symposium on Computer Architecture*, May 1999.

[18] D.J.Kuck and R.A.Stokes. The burroughs scientific processor (bsp). *IEEE Transactions on Computers*, pages 363–376, May 1982.

[19] A. Beaumont-Smith, M. Liebelt, C.C. Lim, and K. To. A digital signal multi-processor for matrix applications. *14th Australian Microelectronics Conference*, October 1997.

[20] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[21] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. *SIGPLAN Notices*, 29(6):196–205, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[22] Roger Espasa, Mateo Valero, and James E. Smith. Out-of-order Vector Architectures. In *MICRO-30*, pages 160–170. IEEE Press, December 1997.

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

	IPC	OPI	R	S	F	VL_x	VL_y
Alpha	2.12	1.00	1.0	1.0	0.00	1.00	1.00
MMX	2.68	4.63	1.99	11.6	0.56	7.48	1.00
MDMX	3.05	4.30	2.04	12.6	0.53	8.00	1.00
MOM	0.82	22.27	2.21	18.9	0.36	8.00	7.51

Table 1: Breakdown of the Speed-up into IPC, OPI and R parameters for `motion2`.

	IPC	OPI	R	S	F	VL_x	VL_y
Alpha	2.27	1.00	1.0	1.0	0.00	1.00	1.00
MMX	2.74	3.80	1.60	7.3	0.45	7.23	1.00
MDMX	2.75	4.15	2.20	11.1	0.33	8.00	1.00
MOM	1.22	12.97	2.60	18.1	0.19	8.00	8.00

Table 2: Breakdown of the Speed-up into IPC, OPI and R parameters for `motion1`.

	IPC	OPI	R	S	F	VL_x	VL_y
Alpha	2.16	1.00	1.0	1.0	0.00	1.00	1.00
MMX	1.76	2.66	0.90	2.0	0.72	3.31	1.00
MDMX	2.23	2.53	0.90	2.4	0.51	4.00	1.00
MOM	0.81	9.94	1.50	5.6	0.53	4.00	4.47

Table 3: Breakdown of the Speed-up into IPC, OPI and R parameters for `idct`.

	IPC	OPI	R	S	F	VL_x	VL_y
Alpha	2.07	1.00	1.0	1.0	0.00	1.00	1.00
MMX	1.85	3.48	1.40	4.4	0.89	3.79	1.00
MDMX	1.84	3.74	2.20	7.3	0.83	4.30	1.00
MOM	1.26	6.03	2.10	7.7	0.71	4.42	1.83

Table 4: Breakdown of the Speed-up into IPC, OPI and R parameters for `rgb`.

	IPC	OPI	R	S	F	VL_x	VL_y
Alpha	2.50	1.00	1.0	1.0	0.00	1.00	1.00
MMX	2.30	2.47	1.10	2.5	0.41	4.58	1.00
MDMX	2.30	2.47	1.10	2.5	0.41	4.58	1.00
MOM	1.12	5.42	1.30	3.2	0.11	4.58	9.00

Table 5: Breakdown of the Speed-up into IPC, OPI and R parameters for `h2v2`.

	IPC	OPI	R	S	F	VL_x	VL_y
Alpha	2.03	1.00	1.0	1.0	0.00	1.00	1.00
MMX	2.38	3.03	3.20	11.4	0.29	8.00	1.00
MDMX	2.38	3.03	3.20	11.4	0.29	8.00	1.00
MOM	1.37	5.92	3.50	14.0	0.09	8.00	6.96

Table 6: Breakdown of the Speed-up into IPC, OPI and R parameters for `comp`.

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

	IPC	OPI	R	S	F	VL_x	VL_y
Alpha	1.79	1.00	1.0	1.0	0.00	1.00	1.00
MMX	2.87	3.12	2.90	14.5	0.58	4.66	1.00
MDMX	2.87	3.12	2.90	14.5	0.58	4.66	1.00
MOM	0.89	9.83	3.10	15.2	0.25	4.54	8.00

Table 7: Breakdown of the Speed-up into IPC, OPI and R parameters for `addblock`.

	IPC	OPI	R	S	F	VL_x	VL_y
Alpha	1.24	1.00	1.0	1.0	0.00	1.00	1.00
MMX	2.40	1.70	2.40	7.9	0.54	2.29	1.00
MDMX	1.82	2.02	2.80	12.6	0.34	4.00	1.00
MOM	1.22	3.71	3.80	13.9	0.11	4.00	6.40

Table 8: Breakdown of the Speed-up into IPC, OPI and R parameters for `ltppar`.

	IPC	OPI	R	S	F	VL_x	VL_y
Alpha	2.09	1.00	1.0	1.0	0.00	1.00	1.00
MMX	1.94	2.23	3.80	7.9	0.52	3.36	1.00
MDMX	1.92	2.31	4.10	8.7	0.44	3.98	1.00
MOM	0.79	6.82	4.70	12.1	0.17	3.17	11.11

Table 9: Breakdown of the Speed-up into IPC, OPI and R parameters for `ltpsfilt`.

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE