Bottlenecks in Multimedia Processing with SIMD style Extensions and Architectural Enhancements

Deepu Talla, *Member, IEEE,* Lizy Kurian John, *Senior Member, IEEE,* and Doug Burger*, *Member, IEEE*

Laboratory for Computer Architecture, Department of Electrical and Computer Engineering
*Computer Architecture and Technology Laboratory, Department of Computer Sciences
The University of Texas Austin
Austin, TX 78712
{deepu, ljohn}@ece.utexas.edu, dburger@cs.utexas.edu

Abstract — Multimedia SIMD extensions such as MMX and AltiVec speedup media processing, however, our characterization shows that the attributes of current general-purpose processors enhanced with SIMD extensions do not match very well with the access patterns and loop structures of media programs. We find that 75-85% of the dynamic instructions in the processor instruction stream are supporting instructions necessary to feed the SIMD execution units rather than true/useful computations, resulting in the underutilization of SIMD execution units (only 1-12% of the peak SIMD execution units' throughput is achieved). Contrary to focusing on exploiting more data level parallelism (DLP), in this paper, we focus on the instructions that support the SIMD computations and exploit both fine- and coarse-grained instruction level parallelism (ILP) in the supporting instruction stream. We propose the MediaBreeze architecture that uses hardware support for efficient address generation, looping and data reorganization (permute, packing/unpacking, transpose, etc). Our results on multimedia kernels show that a 2-way processor with SIMD extensions enhanced with MediaBreeze provides a better performance than a 16-way processor with current SIMD extensions. In the case of application benchmarks, a 2-/4-way processor with SIMD extensions augmented with MediaBreeze outperforms a 4-/8-way processor with SIMD extensions. A first-order approximation using ASIC synthesis tools and cell-based libraries shows that this acceleration is achieved at a 10% increase in area required by MMX and SSE extensions (0.3% increase in overall chip area) and 1% of total processor power consumption.

Preliminary versions of parts of this paper appeared in the Proceedings of IEEE International Conference on Computer Design, Sep. 2001 and ACM Computer Architecture News, Dec. 2001. This research was supported in part by a State of Texas Advanced Technology program grant. L. K. John is also partially supported by the National Science Foundation under grants EIA-9807112 and ECS-0113105 and by Dell, Intel, Microsoft, Tivoli, Motorola and IBM Corporations. D. Burger is supported by a grant from the Intel Research Council, an NSF CAREER Award, an IBM University partnership Award, and a Sloan Foundation Fellowship.

Index Terms — Media processing, subword parallelism, bottlenecks in SIMD extensions, workload characterization, performance evaluation, hardware address generation, low-overhead looping, data reorganization, and superscalar general-purpose processors.

1 Introduction

Contemporary computer applications are multimedia-rich, involving significant amounts of audio and video compression, 2D image processing, 3D graphics, speech and character recognition, communications, and signal processing. While dedicated media-processors and media-tailored ASICs are used in small, low-power embedded devices such as PDAs, cell-phones, and set-top boxes, augmenting the general-purpose processor with media-tailored enhancements has been the course of action in general-purpose computing such as in desktop PCs and workstations. Hardware solutions such as ASICs give advantages of high performance and low power, however, their flexibility and adaptability to new applications is very limited. The necessity to run a variety of workloads including desktop, database, media, Java, scientific and technical applications justifies not abandoning the aggressive general-purpose core in favor of a media-specific solution. The most popular solution in the past five years has been the enhancement of the general-purpose processor with multimedia extensions such as Intel's MMX, SSE1, and SSE2, Sun's VIS, HP's MAX, Compaq's MVI, MIPS's MDMX, and Motorola's AltiVec [1][2]. The key idea in these extensions is the exploitation of subword parallelism in a single instruction multiple data (SIMD) fashion. Four, eight or sixteen data elements of 32-, 16-, or 8-bits width can be operated simultaneously in a single register (128-bits wide). Such techniques have been implemented not only in commercial general-purpose processors, but also in DSP processors such as the TMS320C64 processor from Texas Instruments [3] and the TigerSharc processor from Analog Devices [4].

Obviously the microprocessor design community has embraced the SIMD paradigm for media extensions. Although compiler capabilities to automatically exploit the SIMD extensions have been meager [50][51][52][53], media-rich applications have exploited the paradigm through the use of assembly libraries and compiler intrinsics and have shown significant performance benefits [5][6][7][8]. While the improvement in performance has been encouraging and exciting, we notice that performance does not scale with increasing the SIMD execution resources. Hence, we embark on a study to understand the behavior of multimedia applications on SIMD extensions and the nature of the data level parallelism (DLP) in multimedia applications. More specifically, we attempt to answer the following:

- SIMD enhanced general-purpose processors (GPPs) typically exploit the sub-word parallelism between independent loop iterations in the inner loops of multimedia programs. Where does DLP in media applications reside? Does most of the DLP reside in the inner loops, or is there significant DLP in the outer loops?

- Nested loops are required for processing multimedia data streams and this necessitates the use of multiple indices while generating addresses. GPPs contain limited support to compute addresses of elements with multiple indices. How many levels of nesting are required in common media algorithms? Are the addressing sequences primarily sequential?
- While SIMD extensions are capable of performing multiple computations in the same cycle, it is essential to provide data to the SIMD computation units in a timely fashion in order to make efficient use of the sub-word parallelism. Providing data in a timely fashion requires supporting instructions for address generation, address transformation (data reorganization such as packing, unpacking, and permute), processing multiple nested loop branches, and loads/stores. Are these supporting instructions a dominant part of the instruction stream?
- What percentage of the peak computation rate is achieved for the SIMD execution units in GPPs? If the computation rate is low, what are the reasons that prevent the SIMD execution units from achieving a good computation rate?
- What are effective techniques to further enhance the performance of media applications on SIMD enhanced GPPs?

This paper has two major contributions. The first contribution of the paper is the characterization of media workloads from the perspective of support required for efficient SIMD processing. Typically, studies have focused on the true/core computation part of the algorithms, whereas we show that significant additional performance enhancements can be achieved by focusing on the supporting instructions. The second contribution of the paper is the MediaBreeze architecture, which illustrates how characterization studies can be used to design cost-effective architectural enhancements. The major focus of the proposed architecture is on the instructions that support the true/core computations, rather than on the true/core computations themselves.

The rest of the paper is organized as follows. In section 2 we describe the benchmarks used in the study. Section 3 performs sensitivity experiments on the scalability of conventional instruction level parallelism (ILP) and DLP techniques. In section 4 we describe studies to detect bottlenecks in the execution of SIMD programs on GPPs. In section 4.1, we describe the loop nesting and access patterns in multimedia applications and their mapping onto GPPs with SIMD extensions. In section 4.2, we classify dynamic instructions into two fundamental categories, the true/core computation instructions and the overhead/supporting instructions and analyze their mix in media benchmarks. In section 4.3, we measure the percent of peak computation rate achieved for the SIMD execution units in GPPs by conducting experiments on two different superscalar processors. Section 4.4 identifies additional bottlenecks in conventional ILP processors that limit the computation rate of the SIMD execution units. Based on the understanding of the behavior of multimedia applications and the bottlenecks in GPPs with SIMD extensions, in section 5 we propose the MediaBreeze architecture that incorporates explicit hardware support for processing the overhead/supporting instructions efficiently. The cost of incorporating the MediaBreeze hardware support to a SIMD enhanced GPP is evaluated in section 6. Section 7 discusses related work, and the paper is summarized in section 8.

2 Description of Benchmarks

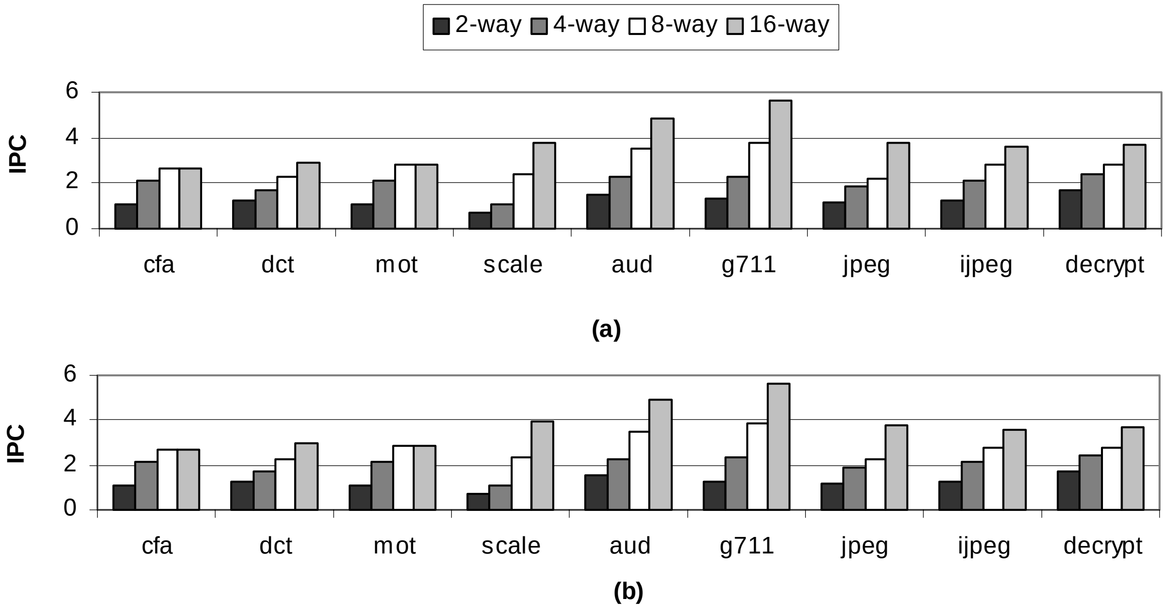
We use nine multimedia benchmarks to study the architectural implications of GPPs with SIMD extensions. Table 1 lists the benchmarks along with a small description and the dynamic instruction count. Sample source code for each of the benchmarks is provided in [9]. SIMD version of the benchmarks was created for two processors, namely, Pentium III and Simplescalar based superscalar processor. The Pentium III MMX code was generated using assembly and compiler intrinsics, while the Simplescalar SIMD code was generated using instruction annotations and assembly code. The code was compiled by Intel C/C++ compiler version 4.5 and gcc version 2.6 respectively, using maximum compiler optimizations (including loop unrolling). Our suite includes applications (*g711*, *aud*, *jpeg*, *ijpeg*, and *decryp*t) and kernels (*cfa*, *dct*, *motest*, and *scale)*. The kernels used are major components in image and video processing standards such as JPEG, MPEG, H.263, etc. Many of these benchmarks are also part of media benchmark suites such as MediaBench [10].

Table. 1. Description of the multimedia benchmarks

Benchmark	***Description***	**Instruction count**
	Kernels	
cfa	Color filter array interpolation of a 2 million pixel image with a 5x5 filter (16-bit data)	349,447,420
dct	2-D discrete cosine transform of a 2 million pixel image (16-bit data)	160,050,834
motest	Motion estimation routine on a frame of 2 million pixels (8-bit data)	136,801,609
scale	Linear scaling of an image of 2 million pixels (8-bit data)	3,129,815
	Applications	
g711	G.711 speech coding standard (A-law to u-law and vice-versa) conversions on 2 million audio samples (8-bit data)	63,360,233
aud	Audio effects on 2 million audio samples (echo, signal mixing and filtering) (16-bit data)	283,199,976
jpeg	JPEG image compression on a 800-by-600 pixel image	208,940,079
ijpeg	JPEG image de-compression resulting in a 800-by-600 pixel image	136,173,916
decrypt	IDEA decryption on 192,000 bytes of data	125,683,876

3 A Scalability test

Media applications are known to contain significant amount of DLP and a logical approach to improve performance is to scale the processor resources to extract more parallelism. To understand the ability of wide out-of-order superscalar processors to increase performance of multimedia programs, we performed experiments scaling the various resources of the processor (using a modified Simplescalar-3.0 simulator [11] enhanced with 64-bit SIMD execution units). All components are scaled as in Table 2. Each of the nine benchmarks was modified to incorporate SIMD code using assembly and instruction annotations of the modified Simplescalar simulator. Fig. 1(a) shows the instructions per cycle (IPC) for different processor configurations for each of the benchmarks. We, incidentally, also note that almost the same performance can be achieved even if the SIMD execution units were not scaled; i.e. the non-SIMD components are scaled up to the 16-way processor keeping the SIMD component constant as a 2-way processor (i.e. 2 SIMD ALUs and 1 SIMD multiplier). The IPC for this case is depicted in Fig. 1(b). The percentage increase in IPC when scaling both the SIMD and non-SIMD resources over the case of scaling only the non-SIMD resources is shown in Fig. 1(c).



	cfa	dct	mot	scale	aud	g711	jpeg	ijpeg	decrypt
4-way	< 1 %	< 1 %	< 1 %	< 2 %	< 4 %	< 1 %	< 1 %	< 1 %	< 1 %
8-way	< 1 %	< 1 %	< 1 %	< 3 %	< 1 %	< 1 %	< 1 %	< 1 %	< 1 %
16-way	< 1 %	< 1 %	< 1 %	< 1 %	< 1 %	< 1 %	< 1 %	< 1 %	< 1 %

(c)

Fig. 1. (a) IPC with both the SIMD and non-SIMD resources scaled, (b) IPC with non-SIMD resources scaled, but SIMD resources are constant (same as 2-way processor configuration), and (c) performance improvement of (a) over (b)

Table. 2. Processor and memory configurations

Parameters	2-way	4-way	8-way	16-way
Fetch width, Decode width, Issue width, and Commit width	2	4	8	16
RUU Size	32	64	128	256
Load Store Queue	16	32	64	128
Integer ALUs (latency/recovery = 1/1)	2	4	8	16
Integer Multipliers (latency/recovery = 3/1)	1	2	4	8
Load/Store ports (latency/recovery = 1/1)	2	4	8	16
L1 I-cache (size in KB, hit time, associativity, block size in bytes)	16, 1, 1, 32	16, 1, 1, 32	16, 1, 1, 32	32, 1, 1, 64
L1 D-cache (size in KB, hit time, associativity, block size in bytes)	16, 1, 4, 32	16, 1, 4, 32	16, 1, 4, 32	16, 1, 4, 32
L2 unified cache (size in KB, hit time, associativity, block size)	256, 6, 4, 64	256, 6, 4, 64	256, 6, 4, 64	256, 6, 4, 64
Main memory width	64 bits	128 bits	256 bits	256 bits
Main memory latency (first chunk, next chunk	65, 4	65, 4	65, 4	65,4
Branch Predictor – bimodal (size, BTB size)	2K, 2K	2K, 2K	2K, 2K	2K, 2K
SIMD ALUs	**2**	**4**	**8**	**16**
SIMD Multipliers	**1**	**2**	**4**	**8**

The observation suggests that SIMD execution units are already underutilized and bottlenecks are concealed elsewhere in the non-SIMD portion of the application.

4 Identification of Bottlenecks

It is evident that there are bottlenecks in SIMD style media processing and that it is not possible to get significant amounts of additional performance improvements by merely increasing the SIMD resources. We investigate characteristics of media programs that point towards the bottlenecks in current SIMD architectures.

4.1 Nested loops in multimedia applications

In this section we investigate the nature of multimedia loops to understand the levels of nesting, stride patterns, and the location of the parallelism. Desktop/workstation multimedia applications such as streaming video encoding/decoding (MPEG 1/2/4 and Motion JPEG), audio encoding/decoding (ADPCM, G.7xx, MP3, etc), video conferencing (H.323, H.261, etc), 3D games, and image processing (JPEG, filtering) typically operate on sub-blocks in a large 1- or 2-dimensional block of data. Audio applications operate on chunks of one-dimensional data samples at a time (for example, the MP3 codec operates on “frames” which are smaller components of the complete audio signal that last a fraction of a second). Image and video applications operate on sub-blocks of two-dimensional data at a time (for example, the DCT algorithm operates on 8x8 pieces of data in a large image such as 1600x1200 pixels). Such a division of data into sub-blocks results in the data being accessed with different strides at various instances in the algorithm. Fig. 2 depicts a 2-dimensional block of data that is accessed with four different strides — two in the vertical direction and two in the horizontal direction.

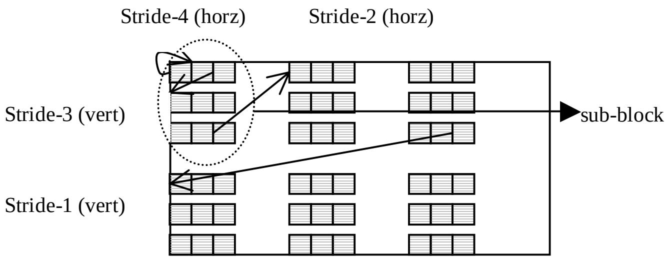


Fig. 2. A 2-D data structure in which sub-blocks of data are processed. The data elements surrounded by the dotted ellipse form one sub-block. Each sub-block requires two strides (one each along the rows and columns of the sub-block, namely stride-4 and stride-3). Additional two strides (stride-2 and stride-1) are required for accessing different sub-blocks in the horizontal and vertical direction.

Source code for the aforementioned algorithms involves the usage of multiple nested loops (commonly ‘for’ loops in C language) to process the data streams. Much of the available parallelism in multimedia applications is seen to be DLP that resides at the various levels of nesting. The dimensions of each sub-block for most multimedia algorithms are small (filtering typically uses 3x3 or 5x5 or 7x7 sub-blocks, DCT operates on 8x8 sub-blocks, and motion estimation operates on 16x16 sub-blocks) resulting in limited parallelism in the innermost loop [12]. However, the number of sub-blocks themselves is large since the size of the data stream can be on the order of several MB. Consequently, a significant part of the DLP in multimedia applications resides outside the innermost loop; the way applications are coded currently.

Existing GPPs with SIMD extensions exploit DLP between independent loop iterations in the innermost loops leading to significant untapped available DLP in multimedia applications. Fig. 3 shows the SIMD C-code implementation of the discrete cosine transform (the DCT is a major component in JPEG image and MPEG video coding) which operates on 8x8 sub-blocks in an image of a given height and width. The second matrix is transposed before doing the computation because accessing the second matrix in column-major order results in a significant amount of overhead. This is particularly true when using SIMD instructions because a SIMD register needs to be packed with an element from different rows (and hence not contiguous). If a SIMD register holds eight elements, then all eight rows of a matrix need to be loaded into the cache and then elements belonging to the same column are packed into the register. It is possible to eliminate one of the transpose operations (either from row or column 1D-DCT) if a transposed version of the DCT coefficients is available. In Fig. 3, there are a total of five nested for-loops for the DCT routine. Current SIMD instructions exploit data level parallelism (DLP) in the innermost for-loop (variable ‘m’). The number of iterations would be scaled down according to the width of the available SIMD datapath (currently 64 or 128 bits wide) and size of each element (8-bit, 16-bit, or 32-bit).

```
void 2D_DCT(IMAGE[IMAGE_WIDTH][IMAGE_HEIGHT])
{
  for(i = 0; i < IMAGE_HEIGHT/8; i++)
    for(j = 0; j < IMAGE_WIDTH/8; j++)
    {
      1D_ROW_DCT (DCT_COEF [8][8], BLOCK [8][8]);
      1D_COL_DCT (DCT_COEF [8][8], BLOCK [8][8]);
    }
}
```

```
void 1D_XXX_DCT(DCT_COEFF[8][8], BLOCK[8][8])
{
  Transpose (BLOCK [8][8]);
  for(k = 0; k < 8; k++)
  {
    for(l = 0; l < 8; l++)
    {
      temp = 0;
      for(m = 0; m < 8/SIMD_WIDTH; m++)
          temp +=
SIMD_MUL (DCT_COEFF [k][m], BLOCK [l][m];
      output[k][l] = SIMD_REDUC (temp)
} } }
```

Fig. 3. C-code for 2D-DCT implementation

Next, we analyze the access patterns in media applications. Analysis of media and Digital Signal Processing (DSP) applications unveils invocation of several address patterns, often multiple simultaneous sequences [13]. Fig. 4 shows the typical access patterns in media and DSP kernels. Table 3 lists several key multimedia and DSP kernels and the typical number of nested loops required along with their corresponding primitive addressing sequences. Hardware to generate multiple address sequences is not overly complicated, but supporting them using general-purpose instruction sets is not very efficient, as the available addressing modes are limited. Furthermore, there is not enough support for keeping track of multiple indices/strides efficiently in GPPs. Similarly, keeping track of multiple loop nests/bounds involves a combination of several addressing modes and instructions. Thus, even though GPPs are enhanced with SIMD extensions to extract DLP in multimedia programs, there is a mismatch between the requirements of media applications (for address generation and nested loops) and the ability of GPPs with SIMD extensions. Simple ASICs can perform these tasks efficiently; however, loss of programmability and flexibility is a weakness of that approach.

Given a sequence of length L, if A_m is address m in the range $0 \leq m \leq L-1$, most multimedia and DSP kernels can be considered to be composed of primitive addressing sequences such as the following:

(i) Sequential addressing: $A_0, A_1, A_2, \ldots A_{N-1}$

(ii) Sequential with offset (k)/stride addressing: $A_{0+k}, A_{1+k}, A_{2+k}, \ldots, A_{N-1+k}$

(iii) Shuffled addressing (base r, N/r = p): $A_0, A_p, A_{2p}, \ldots, A_1, A_{p+1}, A_{2p+1}, \ldots, A_2, A_{2p+2}, \ldots, A_{2p+2}, \ldots, A_{N-1}$

(iv) Bit-reversed addressing (e.g. N = 8): $A_0, A_4, A_2, A_6, A_1, A_5, A_3, A_7$

(v) Reflected addressing: $A_0, A_{N-1}, A_1, A_{N-2}, \ldots, A_m, A_{N-m}, \ldots, A_{N/2-1}, A_{N/2}$

Fig. 4. Typical access patterns in multimedia and DSP kernels [13]

Table. 3. Summary of key media algorithms and the required nested loops along with their primitive addressing sequences

Multimedia/DSP algorithm	Nested loops	Addressing Sequences
Discrete Cosine Transform (JPEG & MPEG coding)	5	Sequential and sequential with multiple offsets/strides
Motion Est./Comp. (MPEG, H.263, etc)	5	Sequential and sequential with multiple offsets/strides
Wavelet Transform (JPEG2000)	2 - 6	Sequential and sequential with multiple offsets/strides
Color Space Conversion (JPEG, MPEG, 3D graphics)	> 4	Sequential, sequential with offsets, and shuffled
Scaling and matrix operations (image/video)	3	Sequential and sequential with multiple offsets/strides
Fast Fourier transform	> 3	Shuffled and bit-reversed
Color Filter Array, median filtering, correlation	2 – 5	Sequential and sequential with multiple offsets/strides
Convolution, FIR, and IIR filtering	3 – 4	Sequential, sequential with offsets, and reflected
Edge detection, alpha saturation (image/video)	2 – 5	Sequential and sequential with multiple offsets/strides
Up/Down sampling, 3-D transformation (graphics)	3 – 5	Sequential and sequential with multiple offsets/strides
Quantization (JPEG, MPEG)	2 – 4	Sequential and sequential with multiple offsets/strides
ADPCM, G.711 (speech)	2 – 3	Sequential and sequential with multiple offsets/strides

4.2 Overhead/Supporting instructions

The discussion in the previous section points to the need of several instructions to compute addresses and otherwise support the core SIMD computations. In this section, we analyze the media instruction stream by focusing on the two distinct sets of operations: the **true/core computations** as required by the algorithm and the **overhead/supporting instructions** such as address generation, address transformation (data movement and data reorganization such as packing and unpacking), loads/stores, and loop branches. Consider the DCT code in Fig. 3. The true/core computation instructions for the DCT routine are the multiply (of DCT coefficients and data) and the accumulate operations (addition of multiplied values). This is shown in bold in Fig. 3. All the other instructions are denoted as **overhead**; their sole purpose is to aid in the execution of the true/core computation instructions. Many of them arise due to the programming conventions of general-purpose processors, abstractions and control flow structures used in programming, and mismatch between how data is used in computations versus the sequence in which data is stored in memory. A similar kind of classification of instructions into access and execute instructions was performed in decoupled access-execute (DAE) processors [14][15]. In our classification, the overhead component includes loop branches and reduction operations [16] that are specific to multimedia applications (e.g. packing/unpacking, and permute) in addition to the memory access task. The instructions contributing to the overhead are:

- Address generation – considerable processing time is dedicated in performing the address calculations required to access the components of the data structures/arrays, which is sometimes called address arithmetic overhead.
- Address transformation – transforming the physical pattern of data into the logical access sequence (transposing the matrix in Fig. 3, packing/unpacking data elements in SIMD computations, and reorganizing data in other ways.
- Loads and Stores – data is not always available in registers and has to be fetched from memory or stored to memory, the so-called access overhead.
- Branches – performing control transfer (for each of the 5 nested for-loops in the example).

Fig. 5 shows the assembly code classified into true/core computation and overhead instructions (for two processors) for the 1D-DCT routine from Fig. 3 excluding the transpose function, i.e. the three inner level nested loop structure. Transposing the second matrix before multiplication will necessitate additional overhead instructions for address transformation. The first processor is a Pentium III processor based on the P6 microarchitecture [17], and the second processor is based on a modified Simplescalar processor enhanced with SIMD extensions. The SIMD registers in the case of Simplescalar are aliased to the floating-point registers. From Fig. 5, it can be seen that a significant number of overhead/supporting instructions are necessary to feed the SIMD computation units.

In order to quantify the amount of overhead/supporting instructions in multimedia programs, we evaluated the performance of six of the nine benchmarks listed in Table 1. *Jpeg, ijpeg,* and *decrypt* are not used in this experiment because the source code for these three benchmarks includes initialization routines and file I/O. Five of the six benchmarks (except *g711)* were mapped in such a way that the SIMD execution units perform every true/core computation. Fig. 6 shows the breakdown of dynamic instructions into various classes (memory, branch, integer, SIMD overhead, and SIMD/true computation). It is seen that the overhead/supporting instructions that are required to assist the SIMD computation (true/core computations) instructions dominate the dynamic instruction stream (75-85%). A significant number of instructions are required for processing the loop branches and computing the strides for accessing the data

Pentium III – MMX code

```
lea      ebx, DWORD PTR [ebp+128]   load/address overhead
mov      DWORD PTR [esp+28], ebx    load/address overhead
$B1$2:
xor      eax, eax                   address overhead
mov      edx, ecx                   address overhead
lea      edi, DWORD PTR [ecx+16]    load/address overhead
mov      DWORD PTR [esp+24], ecx    load/address overhead
$B1$3:
movq     mm1, MMWORD PTR [ebp]      load overhead
pxor     mm0, mm0                   initialization overhead
pmaddwd mm1, MMWORD PTR [eax+esi]
                                    True Computation
movq     mm2, MMWORD PTR [ebp+8]    load overhead
pmaddwd mm2, MMWORD PTR [eax+esi+8]
                                    True Computation
add      eax, 16                    address overhead
paddw    mm1, mm0                   True Computation
paddw    mm2, mm1                   True Computation
movq     mm0, mm2                   load related overhead
psrlq    mm2, 32                    SIMD reduction
overhead
movd     ecx, mm0                   SIMD load overhead
movd     ebx, mm2                   SIMD load overhead
add      ecx, ebx                   SIMD conv. Overhead
mov      WORD PTR [edx], cx         store overhead
add      edx, 2                     address overhead
cmp      edi, edx                   branch related
overhead
jg       $B1$3                      loop branch overhead
$B1$4:
mov      ecx, DWORD PTR [esp+24]    load/address overhead
add      ebp, 16                    address overhead
add      ecx, 16                    address overhead
mov      eax, DWORD PTR [esp+28]    load/address overhead
cmp      eax, ebp                   branch related
overhead
jg       $B1$2                      loop branch overhead
```

Simplescalar-SIMD – gcc code

```
move       $11,$0           address overhead
l.d        $f6,$LC1         load overhead
$L33:
move       $10,$0           address overhead
move       $9,$5            address overhead
$L37:
mtc1       $0,$f4           initialization overhead
mtc1       $0,$f5           initialization overhead
move       $8,$0            address overhead
move       $7,$9            address overhead
move       $3,$4            address overhead
$L41:
l.simd     $f0,0($3)        SIMD load overhead
l.simd     $f2,0($7)        SIMD load overhead
mul.simd   $f0,$f0,$f2      True Computation
addu       $8,$8,1          address overhead
add.simd   $f4,$f4,$f0      True Computation
slt        $2,$8,2          branch related
overhead
addu       $7,$7,8          address overhead
addu       $3,$3,8          address overhead
bne        $2,$0,$L41       loop branch overhead
redu.simd  $f4,$f4,$f6      SIMD reduction
overhead
addu       $9,$9,16         address overhead
addu       $10,$10,1        address overhead
slt        $2,$10,8         branch related
overhead
s.simd     $f4,0($6)        SIMD store overhead
bne        $2,$0,$L37       loop branch overhead
addu       $6,$6,16         address overhead
addu       $4,$4,16         address overhead
addu       $11,$11,1        address overhead
slt        $2,$11,8         branch related
overhead
bne        $2,$0,$L33       loop branch overhead
```

Fig. 5. Optimized assembly code for the 1D-DCT routine shown in Fig. 3 (excluding matrix transpose)

organized in sub-blocks. The Pentium III processor has more memory references than the Simplescalar based processor because the x86 ISA has fewer logical registers (8 versus 32 in conventional RISC processors).

4.3 SIMD throughput and efficiency

In this section, we evaluate the throughput of the SIMD units to understand the impact of the overwhelming number of instructions needed to support the SIMD computations. We define SIMD efficiency as the ratio of the execution cycles **ideally** necessary for the true/core computation instructions to the overall execution cycles **actually** consumed. In other words, SIMD efficiency indicates what fraction of the peak throughput of the SIMD units is actually achieved. The actual execution cycles are

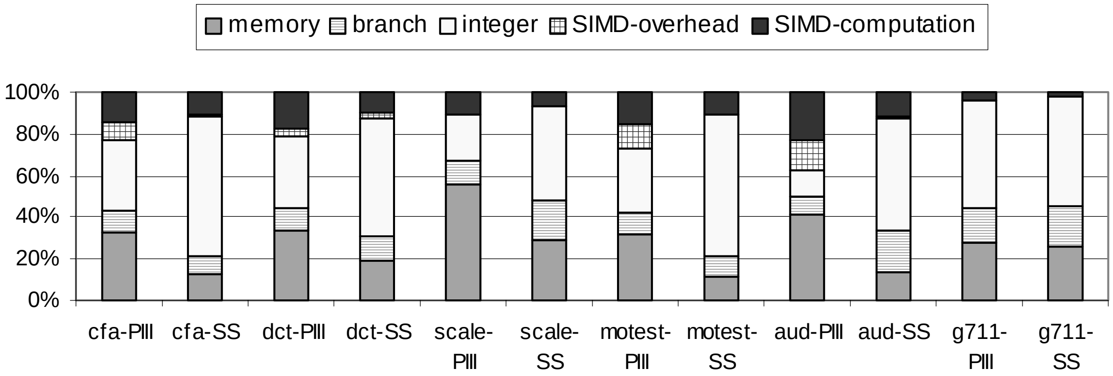


Fig. 6. Breakdown of dynamic instructions into various classes

obtained by measurement with processor performance counters or by simulation, while the ideal cycles are computed assuming that the overhead instructions can be perfectly overlapped with the true/core computation instructions. In the ideal case, overhead instructions such as address generation, memory access, data reorganization, and loop branches do not consume additional processor cycles. The number of ideal execution cycles depends on the amount of SIMD resources in a machine. For example, consider a matrix multiplication algorithm of two *NxN* matrices, with computational complexity $O(N^3)$. This is assuming that there is one multiplier and it is pipelined and the addition/accumulation can take place in parallel. Thus, an 8x8 matrix multiply should take 512 cycles on a machine with one multiplier (in the pure dataflow model), and take 128 cycles on a machine with 4 multipliers (assuming that there are at least 4 adders for the accumulation). If this algorithm were to take 2500 cycles on a real machine with one multiplier, then the efficiency of computation is 20% (512/2500). The important thing to note here is that if efficiency achieved is low, it suggests opportunities for further enhancement.

We measure the SIMD efficiency on two platforms, a Pentium III machine and a 2-way Simplescalar simulator, for each of the first six benchmarks described in Table 1. The MMX extensions in the Pentium III processor provide fixed-point SIMD capability with two 64-bit MMX ALUs and one 64-bit MMX multiplier. The SSE extensions provide floating-point SIMD capability. The Simplescalar processor execution core is similarly configured to contain two 64-bit SIMD ALUs and one 64-bit SIMD multiplier. Table 4 shows the execution statistics and SIMD efficiency for each of the benchmarks. The ideal number of execution cycles is computed by identifying the number of required true/core computation operations and the available SIMD execution units (2 ALUs and 1 multiplier in both the processors).

Table. 4. Execution statistics and efficiency of media programs

Benchmark	Pentium III – MMX & SSE			Simplescalar - SIMD		
	Inst. Count	Actual Cycle count	Efficiency	Inst. Count	Actual Cycle count	Efficiency
cfa	404,290,544	231,616,932	5.16 %	349,447,420	338,685,938	3.53 %
dct	188,798,806	123,944,326	6.2 %	160,050,834	131,587,103	5.84 %
scale	2,170,274	20,756,929	2.31 %	3,129,815	4,626,696	10.36 %
motest	156,734,613	113,623,185	3.38 %	136,801,609	129,364,679	5.94 %
aud	220,320,505	150,386,375	11.97 %	283,199,976	191,516,819	9.40 %
g711	59,066,806	64,006,729	1.12 %	63,360,233	49,302,976	1.45 %

SIMD efficiency ranges from 1% to 12% and 1.5% to 10.5% for the Pentium III and Simplescalar based processor respectively. The SIMD efficiency is alarmingly low because the overhead/supporting instructions dominate the dynamic instruction stream. The execution time is also increased because of conventional architectural limitations such as cache misses, misalignment issues, resource stalls, BTB misses, TLB misses, and branch mis-speculations. The efficiency of the Pentium III processor is slightly higher than the Simplescalar based processor in four of the six benchmarks because it is able to issue three micro-ops (equivalent to 2.7 x86 CISC instructions for the benchmarks above) while the Simplescalar processor issues two instructions per cycle. The L1 cache latency of the Pentium III processor is 3 cycles, while that of the Simplescalar configuration is 1 cycle. Hence two memory-intensive benchmarks (*scale* and *g711*) achieve a better efficiency for the Simplescalar configuration. We also measured similar statistics for the Pentium III and the Simplescalar based processor without SIMD extensions. We found that the execution time is worse than SIMD enhanced processors, but the efficiency is higher for non-SIMD processors (2.5% - 16.5%). This is because a 64-bit SIMD execution unit counts towards a peak rate of either 4 or 8 computations per cycle (16-bit or 8-bit data), whereas the scalar execution unit counts toward a single computation per cycle. While it is true that SIMD enhancements were not added to improve efficiency of processing but to speedup multimedia programs, our characterization highlights the gap between peak rate and achieved rate for SIMD programs and points to ample opportunities for performance improvement.

4.4 Memory access and branch bottlenecks

Memory latency prevents processors from fetching data in a timely fashion to achieve peak throughput. Also, supporting wide issue processors requires the ability to fetch across multiple branches. In this section, we investigate how memory latency and branch prediction impact the performance of these media kernels and applications. Table 5 shows the IPC with unit cycle memory access (i.e. a perfect L1 cache) and perfect branch prediction for the 2-, 4-, 8-, and 16-way processors with SIMD extensions.

Table. 5. Performance (IPC) with unit cycle memory accesses and perfect branch prediction

	cfa	***dct***	***mot***	***scale***	***aud***	***g711***	***jpeg***	***ijpeg***	***decrypt***
Unit cycle memory access									
2-way	1.04	1.26	1.06	1.43	1.57	1.59	1.33	1.34	1.75
4-way	2.19	1.78	2.14	2.84	2.50	3.10	2.00	2.30	2.52
8-way	2.71	2.30	2.85	5.56	3.66	5.22	2.37	3.21	2.95
16-way	2.71	2.95	2.86	9.54	5.27	7.76	5.10	4.07	3.89
Perfect branch prediction									
2-way	1.75	1.60	1.79	0.68	1.62	1.29	1.24	1.42	1.70
4-way	3.44	3.09	3.59	1.05	2.69	2.29	1.92	2.60	2.40
8-way	6.47	5.91	7.03	2.35	4.35	3.79	2.46	3.99	2.86
16-way	10.49	11.19	11.61	3.91	6.37	5.55	5.45	6.66	3.79

It is seen that different programs vary in their sensitivity to memory latency and branch prediction. *Scale* and *g711* benchmarks are memory bound programs and improve significantly due to a unit cycle memory access but show negligible increase in IPC due to perfect branch prediction. *Cfa*, *dct*, and *mot* are benchmarks that operate on sub-blocks in a 2-D structure requiring five levels of loop nesting and benefit the most from perfect branch prediction and the ability to fetch across multiple branches in a single cycle. A unit cycle memory access has negligible performance impact on these three benchmarks. The remaining four benchmarks (*aud*, *jpeg*, *ijpeg*, and *decrypt*) benefit equally from both perfect branch prediction and unit cycle memory access. It is evident from this experiment that it is extremely important to provide low latency memory access and excellent branch prediction extending over multiple branches in order to achieve good performance.

5 Hardware Support for Efficient SIMD Processing

5.1 Decoupling Computation and Overhead

The characterization of media applications presented in the previous sections showed that supporting or overhead related instructions dominate the instruction stream. Obviously, overhead/supporting instructions need to be either eliminated, alleviated, or overlapped with the true/core computations for better performance, i.e. the higher the overlap of overhead/supporting instructions, the higher the SIMD efficiency. We exploit the observed characteristics of the media programs and propose to augment GPPs (having SIMD execution units) with specialized hardware to efficiently overlap the overhead/supporting instructions. We refer to this as the MediaBreeze architecture. Fig. 7 illustrates the block diagram of the proposed architecture.

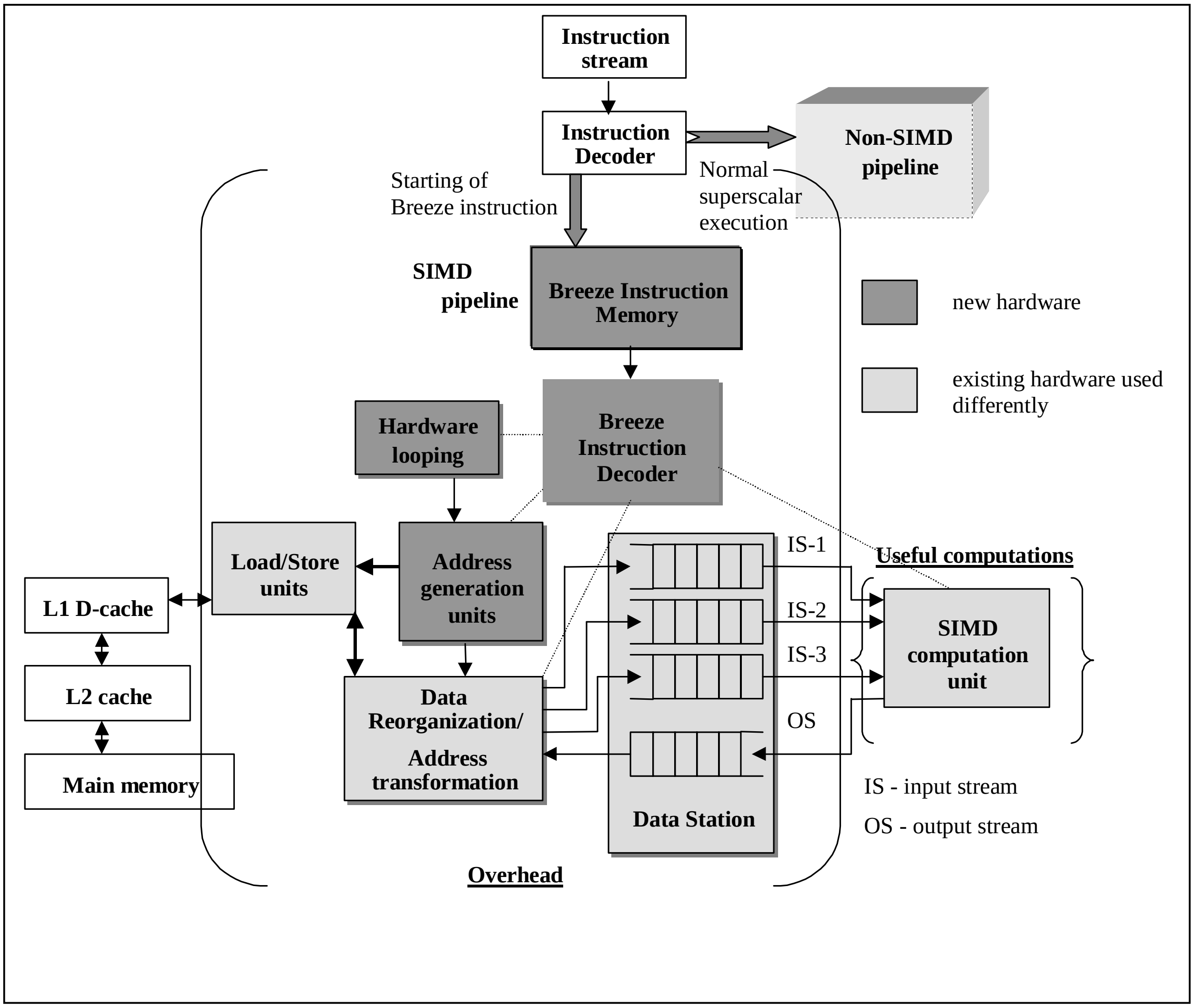


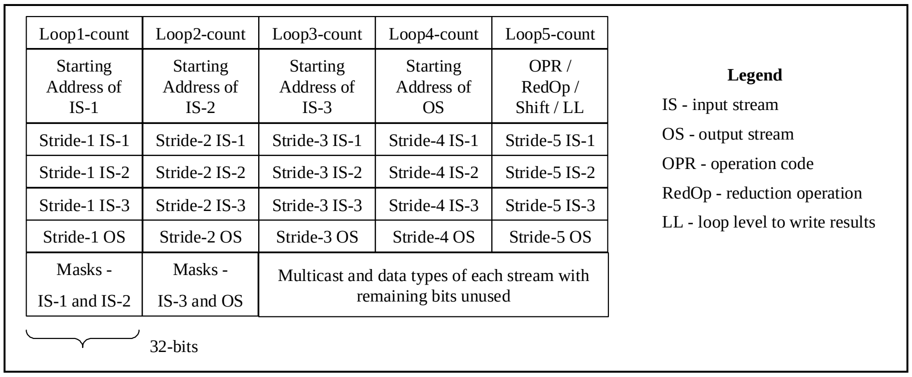
Fig. 7. The MediaBreeze Architecture

In order to perform the SIMD operations, the MediaBreeze architecture introduces new hardware units as well as uses existing hardware units. The new hardware units (darkly shaded blocks in Fig. 7) are the address generation units, hardware looping, and Breeze instruction memory & decoder. The existing hardware units used (lightly shaded blocks in Fig. 7) are load/store units, SIMD computation unit, data reorganization/address transformation, and the data station. The SIMD computation unit handles the true/core computation part while the remaining units handle the overhead/supporting instructions. The hardware units that process the overhead/supporting instructions are:

- Address calculation: address arithmetic functions are moved from the execution unit subsystem in current processors to a dedicated hardware unit where address arithmetic hardware would generate all input and output address streams/data structures concurrently with the SIMD computations. The CPU in current ILP processors performs address calculations explicitly. Dedicated address arithmetic hardware would allow for the SIMD computation unit to stream at the peak rate.

- Address transformation: In many algorithms, the logical access sequence of data is vastly different from the physical storage pattern. Various permute operations including pack, unpack instructions are used. For example, the first element in eight columns of a matrix needs to be packed into a single row (or SIMD register). Similarly a single element (16-bits wide) needs to be unpacked into all the four sub-words of a SIMD register (64-bits wide). MediaBreeze efficiently handles the task of reordering data with explicit hardware support.
- Loads and stores: The same load/store units present in conventional ILP processors are used for this purpose.
- Branch processing: To eliminate branch instruction overhead, MediaBreeze employs zero-overhead branch processing using dedicated hardware loop control and supports up to five levels of loop nesting. All branches related to loop increments (based on indices used for referencing data) are handled by this technique. This is done in many conventional DSP processors such as the Motorola 56000 and TMS320C5x from Texas Instruments [18].
- Data Station: This is the register-file for the SIMD computation and is implemented as a queue. Dedicated register-files are present in conventional machines for SIMD either as a separate register file (as in AltiVec) or aliased to the floating-point register file (as in MMX).
- Breeze instruction memory and decoder: In order to program/control the hardware units in the MediaBreeze architecture, a special instruction called the Breeze instruction is formulated. The Breeze instruction is a multidimensional vector instruction. The Breeze instruction memory stores these instructions once they enter the processor. Fig. 8 illustrates the structure of the Breeze instruction.

Five loop index counts (bounds) are indicated in the Breeze instruction to support five level nested loops (in hardware) [18][42]. None of our benchmarks required more than five nested loops. The MediaBreeze architecture allows for three input data structures/streams and produces one output structure. This was chosen because some media algorithms can benefit from this capability (current SIMD execution units sometimes operate on three input registers to produce one output value). Each data structure/stream has its own dedicated address generation unit to compute the address every clock cycle with the base address specified in the Breeze instruction. Due to the sub-block access pattern in media programs, data is accessed with different strides at various points in the algorithm (as described in section 4.1). The Breeze instruction facilitates multiple strides (one at each level of loop nesting, i.e., a total of five strides) for each of the three input streams and one output stream. The strides indicate address



Loop1-count	Loop2-count	Loop3-count	Loop4-count	Loop5-count
Starting Address of IS-1	Starting Address of IS-2	Starting Address of IS-3	Starting Address of OS	OPR / RedOp / Shift / LL
Stride-1 IS-1	Stride-2 IS-1	Stride-3 IS-1	Stride-4 IS-1	Stride-5 IS-1
Stride-1 IS-2	Stride-2 IS-2	Stride-3 IS-2	Stride-4 IS-2	Stride-5 IS-2
Stride-1 IS-3	Stride-2 IS-3	Stride-3 IS-3	Stride-4 IS-3	Stride-5 IS-3
Stride-1 OS	Stride-2 OS	Stride-3 OS	Stride-4 OS	Stride-5 OS
Masks - IS-1 and IS-2	Masks - IS-3 and OS	Multicast and data types of each stream with remaining bits unused		

Legend

IS - input stream

OS - output stream

OPR - operation code

RedOp - reduction operation

LL - loop level to write results

Fig. 8. Structure of the Breeze Instruction

increment/decrement values based on the loop-nest level. Depending on the mask values for each stream (indicated in the Breeze instruction) and the loop-nest level, one of the five possible strides is used to update the address pointer. If an application does not need five levels of nesting, non-constant strides () can be generated with the extra levels of looping [19].

Data types of each stream/structure are also indicated in the Breeze Instruction. Depending on the size of each element in the data structures, the SIMD parallelism is computed. For example, if one data stream is 8-bit data (16-way parallelism for a 128-bit wide execution unit) and the other is 16-bit data (8-way parallelism), the SIMD processing achieves only 8-way parallelism. The maximum achievable SIMD parallelism is the minimum of all the data structures (all commercial SIMD extensions have this limitation). Current SIMD extensions provide data reorganization instructions for solving the problem of having different element sizes across the data structures (packing, unpacking, and permute) and introduce additional instruction overhead. By providing this information in the Breeze Instruction, special hardware in the MediaBreeze performs this function. The MediaBreeze performs reduction operations and this is also indicated in the Breeze Instruction (for example, multiple independent results in a single SIMD register are combined together in dot product which require additional instructions in current DLP techniques). Support for signed/unsigned arithmetic, saturation, shifting/scaling of final results is all indicated in the Breeze Instruction. This eliminates additional instructions that are otherwise needed for conventional RISC processors.

With the support for multiple levels of looping and multiple strides, the Breeze Instruction is a complex instruction and decoding such an instruction is a complex process in current RISC processors. MediaBreeze instead handles the task of decoding of the Breeze Instruction. MediaBreeze has its own instruction memory to hold a Breeze instruction. Two additional 32-bit instructions are also added to the ISA of the general-purpose processor for starting and interrupting the MediaBreeze. These 32-bit instructions (fetched and decoded by the traditional instruction issue logic) indicate the start and the length of the Breeze Instruction. Whenever a Breeze instruction is encountered in the dynamic instruction stream, the dynamic instructions prior to the Breeze instruction are allowed to finish after which the MediaBreeze instruction interpreter decodes the Breeze instruction. In our current implementation, we halt the superscalar pipeline until the execution of the Breeze instruction is completed because MediaBreeze uses existing hardware units. Otherwise, arbitration of resources is necessary to allow for overlap of the Breeze instruction and other superscalar instructions.

Encoding all the overhead/supporting operations along with the SIMD true/core computation instructions has the advantage that the Breeze instruction can potentially replace millions of dynamic RISC instructions that have to be fetched, decoded, and issued every cycle in a normal superscalar processor. SIMD instructions in GPPs themselves reduce the number of instruction fetches because one instruction operates on multiple data. The Breeze instruction additionally captures all the overhead operations along with the SIMD computation operations thereby drastically reducing repeated (and unnecessary) fetch and decode of the same instructions. This results in giving the MediaBreeze architecture advantages similar to ASIC-based acceleration in [20].

It is possible that an exception or interrupt occurs while a Breeze instruction is in progress. The state of all five loops, their current counts, and loop bounds are saved and restored when the instruction returns. This is similar to the handling of exceptions during move instructions with REP (Repeat Prefix) in x86. MediaBreeze has registers to hold the loop parameters for all the loops and parts of the operating system might have to be modified similar to the Pentium III SSE extensions. Code development for the MediaBreeze architecture is currently done by hand and the programmer has to schedule the dependencies in the code. Compiler technology for SIMD extensions is still in its infancy [50][51][52][53]. Similar to developing code for SIMD extensions, compiler intrinsics have to be employed to utilize the MediaBreeze architecture. We do not underestimate the challenge of compiling for the MediaBreeze architecture; however, the effort will be slightly higher to that of compiling for SIMD extensions. In spite of the lack of adequate compiler support for SIMD extensions, it has been clear that SIMD extensions still enhance media application performance.

5.2 Multicast: A technique to aid in data transformation

The MediaBreeze uses a technique called *Multicast* to eliminate the need for transposing data structures, to allow for reordering of the computations, and to increase reuse of data items soon after fetch by exploiting DLP in outer level loops. Multicasting means copying one/many data items into several registers or buffers at the same item. For example, a data value A may be copied into 8 registers (or 8 sections of a big SIMD register) resulting in a pattern A,A,A,A,A,A,A,A or two items A and B may be copied to 8 registers in the pattern A,A,B,B,A,A,B,B or A,B,A,B,A,B,A,B or another such pattern. The usefulness of multicasting can be illustrated by the well-understood matrix-multiply routine. In a matrix-multiply routine, usually the first matrix is traversed in row-order and the second matrix in column-order. Spatial locality can be exploited in the first matrix due to multiple data elements in each cache block, while the second matrix incurs a compulsory miss on each column the first time; assuming that two consecutive rows do not fit in a cache-block. In a machine with no SIMD execution units, during each iteration for the second matrix, a new cache-line has to be loaded as data belongs to the same column but different cache-line. However, for the case of SIMD processing, multiple cache-lines need to be loaded and data belonging to the required column needs to be reorganized from a vertical to a horizontal direction (packing). This involves substantial overhead and usually, the second matrix is transposed prior to the computation to eliminate the column-access pattern.

The transposing overhead can be eliminated using the Multicast technique. Instead of using column-access pattern, row-order access pattern is used for matrix B, while for matrix A, a single element is multicast to all eight sub-element locations in the SIMD register. Then instead of doing the eight multiplications to generate the first element C1,1 of the result matrix, all eight multiplications using A1,1 (i.e. the first partial product of each of the result terms in the first row) are performed. The sequence of multiplications in a normal SIMD matrix multiply and a multicast matrix multiply are illustrated in Fig. 9. After 64 multiplications, all eight result terms of the first row of the result matrix will be simultaneously generated. The algorithm using the multicast technique is always operating on multiple independent output values, while traditional techniques compute one result term at a time. This eliminates the need for transposing the second matrix. It also increases the reuse of items that were loaded, thus improving the cache behavior of the code. The MediaBreeze architecture provides hardware support for multicasting. This allows the use of cache-friendly algorithms to perform many media algorithms. In this example, broadcast rather than multicast was employed, because one element is transmitted to all eight registers. However, in several applications such as horizontal/vertical downsampling/upsampling, and filtering, several elements are multicast into the sub-element locations, many-to-many mapping as opposed to one-to-many mapping and hence the name multicast. The multicast technique is a superset of existing data reorganization instructions in current SIMD extensions such as AltiVec’s splat [2] and MDMX’s packed accumulators [6][16].

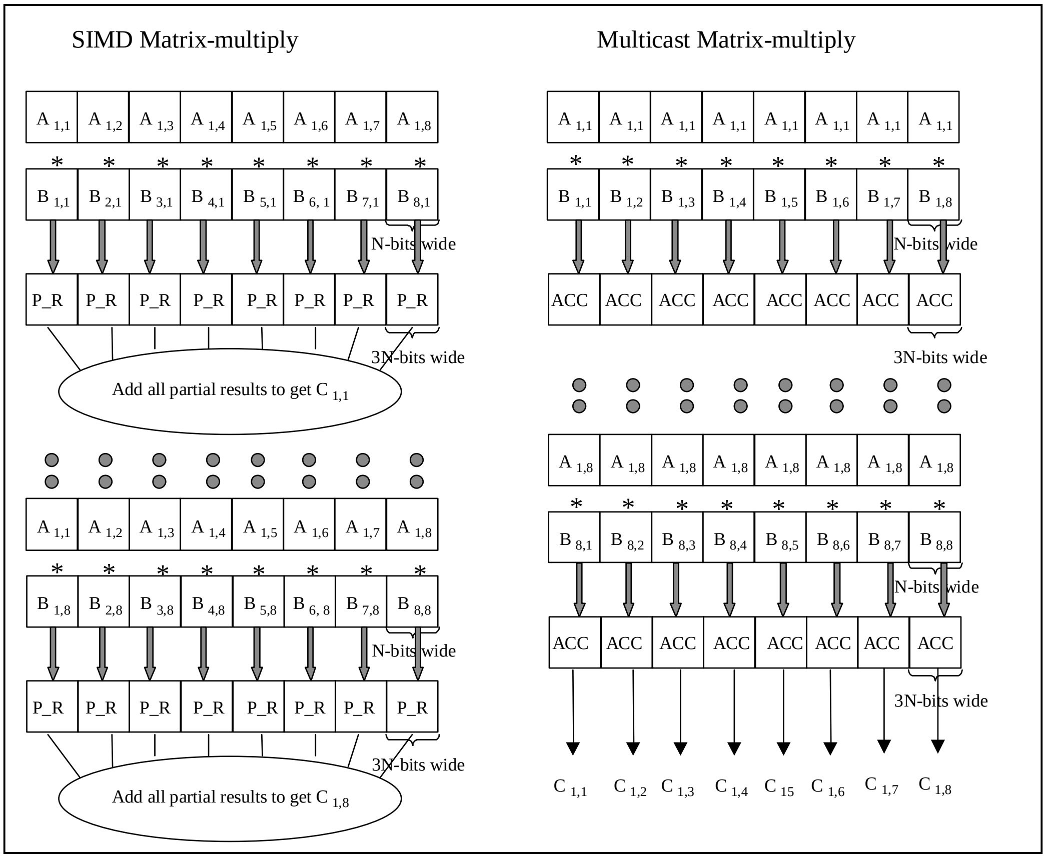

SIMD Matrix-multiply
Multicast Matrix-multiply
$A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{1,4}$ $A_{1,5}$ $A_{1,6}$ $A_{1,7}$ $A_{1,8}$
$B_{1,1}$ $B_{2,1}$ $B_{3,1}$ $B_{4,1}$ $B_{5,1}$ $B_{6, 1}$ $B_{7,1}$ $B_{8,1}$
N-bits wide
P_R P_R P_R P_R P_R P_R P_R P_R
3N-bits wide
Add all partial results to get $C_{1,1}$
$A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{1,4}$ $A_{1,5}$ $A_{1,6}$ $A_{1,7}$ $A_{1,8}$
$B_{1,8}$ $B_{2,8}$ $B_{3,8}$ $B_{4,8}$ $B_{5,8}$ $B_{6, 8}$ $B_{7,8}$ $B_{8,8}$
N-bits wide
P_R P_R P_R P_R P_R P_R P_R P_R
3N-bits wide
Add all partial results to get $C_{1,8}$
$A_{1,1}$ $A_{1,1}$ $A_{1,1}$ $A_{1,1}$ $A_{1,1}$ $A_{1,1}$ $A_{1,1}$ $A_{1,1}$
$B_{1,1}$ $B_{1,2}$ $B_{1,3}$ $B_{1,4}$ $B_{1,5}$ $B_{1,6}$ $B_{1,7}$ $B_{1,8}$
N-bits wide
ACC ACC ACC ACC ACC ACC ACC ACC
3N-bits wide
$A_{1,8}$ $A_{1,8}$ $A_{1,8}$ $A_{1,8}$ $A_{1,8}$ $A_{1,8}$ $A_{1,8}$ $A_{1,8}$
$B_{8,1}$ $B_{8,2}$ $B_{8,3}$ $B_{8,4}$ $B_{8,5}$ $B_{8,6}$ $B_{8,7}$ $B_{8,8}$
N-bits wide
ACC ACC ACC ACC ACC ACC ACC ACC
3N-bits wide
$C_{1,1}$ $C_{1,2}$ $C_{1,3}$ $C_{1,4}$ C_{15} $C_{1,6}$ $C_{1,7}$ $C_{1,8}$

Fig. 9. Multicast technique versus traditional SIMD matrix multiply

If the dimension of the matrices to be multiplied is large, then the multicast method needs temporary registers or an accumulator to store the accumulated results. However, multimedia applications operate on sub-blocks in huge matrices as opposed to processing the entire matrix as a whole. A SIMD parallelism of 8 or 16 is quite adequate to capture most media sub-block rows/columns. Other common operations where multicast is extremely useful include 1-D and 2-D filtering, and convolution. For example, when using MMX for implementing a finite impulse response (FIR) filter, multiple copies of the filter coefficients are needed (equal to the SIMD parallelism) to reduce considerable overhead due to misalignment of coefficient data.

5.3 Example encoding using the Breeze instruction

The Breeze instruction is a densely encoded instruction and hence most media algorithms can be processed in just a few Breeze instructions. Fig. 10 shows the pseudo-code for the implementation of the Breeze instruction. Given a start address for each of the data streams, each address is incremented based on the stride and the loop level during each cycle. Common kernels such as the DCT, color space conversion, motion estimation, and filtering can be mapped to either one or two Breeze instructions. Fig. 11 illustrates the Breeze instruction mapping of the 1-D DCT routine assuming an 8-way SIMD for 16-bit data. For the 1-D DCT routine, only four of the five possible loop nests are needed with the loop boundaries indicated in the Breeze instruction. The starting address of each stream is represented by the starting address of each of the arrays. The third input stream is not used for this algorithm. The value of the strides is computed based on the loop indices and the value of the address pointer in the previous cycle. The address pointer is updated each clock cycle choosing one stride depending on the nesting level of the loops.

```
IS1 = start_address_IS1; IS2 = start_address_IS2;
IS3 = start_address_IS3; OS1 = start_address_OS;

increment_address (level) {
  if (mask_IS1 [level] )  IS1 += stride_IS1[level];
  if (mask_IS2 [level] )  IS2 += stride_IS2[level];
  if (mask_IS3 [level] )  IS3 += stride_IS3[level];
  if (mask_OS [level] )   OS += stride_OS[level];
}

            if       ( (i_5 + 1) = loop1_count)      increment_address(4);
            elseif   ( (i_4 + 1) = loop2_count)      increment_address(3);
            elseif   ( (i_3 + 1) = loop3_count)      increment_address(2);
            elseif   ( (i_2 + 1) = loop4_count)      increment_address(1);
            else     increment_address(5);

SIMD_data_reorganization (R1, R2);
SIMD_compute (MAC, R1, R2, R4);
SIMD_data_reorganization (R4);
```

Fig. 10. Pseudo-code representing a MediaBreeze compute instruction

In a scenario in which all the loop nests and data streams are processed, MediaBreeze executes (in hardware) the following equivalent number of dynamic software instructions (in conventional ILP processors) during each cycle -

- five branches
- three loads and one store
- four address value generation (one on each stream with each address generation representing multiple RISC instructions)
- one SIMD operation (2-way to 16-way parallelism depending on each data element size)

- one accumulation of SIMD result and one SIMD reduction operation
- four SIMD data reorganization (pack/unpack, permute, etc) operations
- shifting & saturation of SIMD results

```
1D_DCT( image[1200][1600], dct_coef[8][8], output[8][8] )
{
 for ( i = 0;  i < 1200/8; i++)
   for ( j = 0; j < 1600/8; j++)
     for (k = 0; k < 8; k++) {
      temp_simd_vector = 0;
       for (l = 0; l < 8; l ++)

        /* Since there is 8-way SIMD parallelism, the innermost loop folds into one
        iteration and is not required */

 temp_simd_vector += multicast(dct_coef[ k ][ l ] * image[ i*8+k ][ j*8+l ]);
            output[ i*8 ][ k*8 ] = temp_simd_vector >> s_bits;
```

0	1200/8	1600/8	8	8

Starting
Address of
image
	Starting
	Address of
	dct coeff
		------------- 	
		NONE
		------------- 	
			Starting
			Address of
			output
				OPR = MAC
				Shift = s_bits
				LL = 4

NONE	16 bytes	-22384 bytes	-22400 bytes	3200 bytes
NONE	-126 bytes	-126 bytes	2 bytes	2 bytes
NONE	NONE	NONE	NONE	NONE
NONE	-22384 bytes	3200 bytes	NONE	NONE

IS-1 = 01111
IS-2 = 01111
	IS-3 = 00000
	OS = 01100
		Multicast is used for dct coefficients
		data types of each stream is set to 16-bit data		

Fig. 11. Breeze instruction mapping of 1D-DCT

5.4 Performance Evaluation and Results

To measure the impact of the MediaBreeze architecture, we modified the *PISA* version of Simplescalar-3.0 (sim-outorder) to simulate Breeze instructions using instruction annotations. We use the same SIMD execution units' configuration as in a Pentium III processor (two 64-bit SIMD ALUs and one 64-bit SIMD multiplier). The memory system for the MediaBreeze architecture is modified to allow for cache miss stalls and memory conflicts (i.e., the SIMD pipeline stalls in the event of a cache miss) since

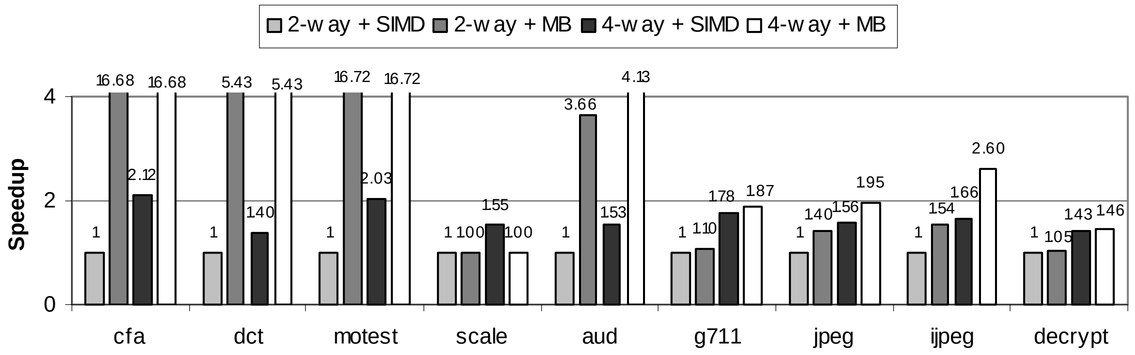


Fig. 12. Performance of MediaBreeze (MB) versus SIMD

the MediaBreeze operates in an in-order fashion. Fig. 12 shows the speedup obtained for each of the benchmarks using the MediaBreeze architecture with a 2-way processor as the baseline.

The speedup of the 2-way MediaBreeze architecture over a 2-way SIMD enhanced processor ranges from 1.0x to over 16x. In four of the nine benchmarks (*cfa*, *dct*, *mot*, *scale* – which are kernels) all of the benchmark code translates into one or two Breeze instructions with no other superscalar instructions necessary. The remaining five benchmarks (*aud*, *g711*, *jpeg*, *ijpeg*, and *decrypt* – which are applications) require scalar superscalar instructions along with Breeze instructions. G711 and decrypt are applications that have the least amount of SIMD instructions (i.e., it is the superscalar pipeline that accounts for a bulk of the execution time rather than the MediaBreeze pipeline) and a 2-way MediaBreeze architecture is only slightly faster than a 2-way SIMD processor. On the other hand for the remaining three benchmarks (*aud*, *jpeg*, and *ijpeg*), a 2-way MediaBreeze architecture is significantly faster than a 2-way SIMD processor.

The MediaBreeze pipeline is susceptible to memory latencies because it operates in-order. Thus MediaBreeze is unable to achieve maximum SIMD efficiency on three of the four kernels (*cfa*, *dct*, and *scale*) in spite of them being mapped completely to one or two Breeze instructions. To reduce the impact of memory latencies on the MediaBreeze architecture, we introduced a prefetch engine to load future data into the L1 cache. Since the access pattern of each data stream is known in advance based on the strides, the prefetch engine does not load any data that is not going to be used. The regularity of the media access patterns prevents the risk of superfluous fetch very commonly encountered in many prefetching environments. The prefetch engine 'slips' ahead of the loads for computation and the computation itself to gather data into the L1 cache. Table 6 shows the speedup of the MediaBreeze architecture with prefetching for the 2-way and 4-way configurations (prefetching was also incorporated into the baseline architecture). We observe that prefetching in the MediaBreeze architecture achieves unit cycle memory access performance in the Breeze instruction portion of the program.

Table. 6. Performance of the MediaBreeze architecture with prefetching

	cfa	***dct***	***mot***	***scale***	***aud***	***g711***	***jpeg***	***ijpeg***	***decrypt***
2-way	1	1	1	1	1	1	1	1	1
2-way + MB	27.92	16.52	16.84	2.14	3.6	1.21	1.44	1.61	1.05
4-way	2.12	1.46	2.03	1.98	1.59	1.78	1.56	1.66	1.43
4-way + MB	27.92	16.52	16.84	4.54	5.61	2.22	2.02	2.68	1.46

The geometric mean of the speedup of the 2-way MediaBreeze processor over a 2-way SIMD processor for the five applications (not including the kernels - *cfa*, *dct*, *mot*, and *scale*) is 1.73 while that of a 4-way SIMD processor over a 2-way SIMD processor is 1.59. Therefore, on average, a 2-way GPP with SIMD extensions augmented with the MediaBreeze hardware achieves a performance slightly better than a 4-way superscalar SIMD processor on media applications. A similar trend is observed for the case of a 4-way GPP with SIMD extensions augmented with the MediaBreeze hardware being slightly superior to an 8-way superscalar SIMD processor.

Since the Breeze instruction is densely encoded, few Breeze instructions are needed for any media-processing algorithm. The number of dynamic instructions that need to be fetched and decoded is shrunk tremendously (as shown in Fig. 13), leading to a reduced use of the instruction fetch, decode, and issue logic in a superscalar processor. The instruction fetch and issue logic are a significant consumer of power in speculative out-of-order processors. Once a Breeze instruction is interpreted, the instruction

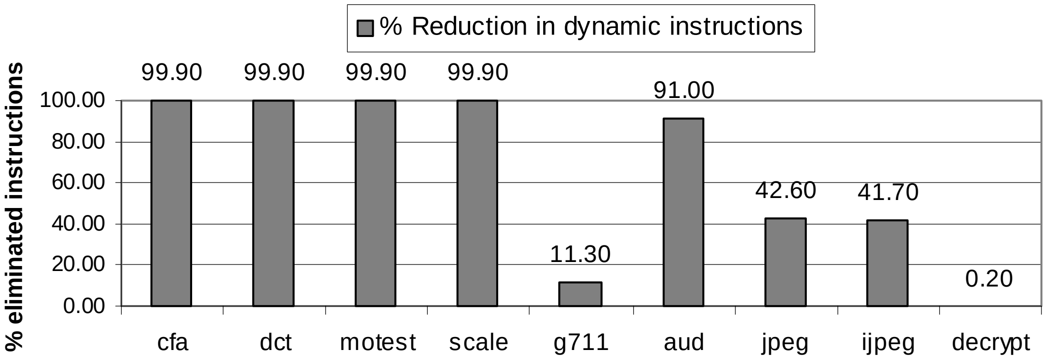


Fig. 13. Reduction in dynamic instructions by using the MediaBreeze architecture. Power savings proportional to the instruction count savings can be expected for fetch, decode and renaming energy.

fetch, decode, and issue logic in the superscalar processor can be shut down for the duration of the loop nest. The use of the vector-style Breeze instruction can eliminate more than half of the instructions from the original program (65% on average). The instructions required to implement looping, address computations, and transformations are removed. Each eliminated instruction results in energy savings in the fetch, decode and register renaming stages.

6 Hardware cost of the MediaBreeze Architecture

6.1 Implementation methodology

To estimate the area, power, and timing requirements of the MediaBreeze architecture, we developed VHDL models for the various components. Using Synopsys synthesis tools [21], we used a cell-based methodology to target the VHDL models to two ASIC cell-libraries from LSI Logic [22][23]. Table 7 lists the libraries and technologies used for evaluating the implementation cost.

Table. 7. Cell-based Libraries (LSI Logic) used in synthesis

Library name	Description
lcbg12-p (G12-p)	A 0.18-micron L-drawn (0.13-micron L-effective) CMOS process. Highest performance solution at 1.8 V with high drive cells optimized for long interconnects associated with large designs.
lcbg11-p (G11-p)	A 0.25-micron L-drawn (0.18-micron L-effective) CMOS process. Highest performance solution at 2.5 V.

The Synopsys synthesis tools estimate area, power, and timing of circuits based on information provided in the ASIC technology library. The ASIC technology library provides four kinds of information.

- Structural information. This describes each cell’s connectivity to the outside world, including cell, bus, and pin descriptions.
- Functional information. This describes the logical function of every output pin of every cell so that the synthesis tool can map the logic of a design to the actual ASIC technology.
- Timing information. This describes the parameters for pin-to-pin timing relationships and delay calculation for each cell in the library.
- Environmental information. This describes the manufacturing process, operating temperature, supply voltage variations, and design layout. The design layout includes wire load models that estimate the effect of wire length on design performance. Wire load modeling estimates the effect of wire length and fanout on resistance, capacitance, and area of nets.

We use the default wire load models provided by LSI Logic's ASIC libraries. The Synopsys synthesis tools compute timing information based on the cells in the design and their corresponding parameters defined in the ASIC technology library. The area information provided by the synthesis tools is prior to layout and is computed based on the wire load models of the associated cells in the design. Average power consumption is measured based on the switching activity of the nets in the design. In our experiments, the switching activity factor originates from the RTL models as the tool gathers this information from simulation. The area, power, and timing estimates are obtained after performing maximum optimizations for performance in the synthesis tools. The hardware cost results obtained by this technique is only a first order approximation based on the accuracy of the synthesis tools and cell-based libraries. The interested reader is referred to [21] for further information regarding the capabilities and limitations of the synthesis tools.

6.2 Hardware implementation of MediaBreeze units

Address generation – The MediaBreeze architecture supports three input and one output data structures/streams. Each of the four data streams has a dedicated address generation hardware unit. Address arithmetic on each stream is performed based on the strides and mask values indicated in the Breeze instruction. For each clock cycle, depending on the mask bits and loop index counts, one of the five possible strides is selected. The new address value is then computed based on the selected stride and the previous address value. Fig. 14 depicts the block diagram of the address generation circuitry for a single data stream/structure.

The *last_val comparators* determine which of the four inner level loop counters have reached their upper bound. The outermost loop comparison is not necessary because the Breeze instruction finishes execution at the instant when the outermost loop counter reaches its upper bound. The *inc-cond* and *inc-combine* blocks generate *flag* signals based on the output from the *last_val comparators* and mask values from the Breeze instruction. If none of the *flag* signals are true, then *stride-5* is used to update the *prev-address;* otherwise, the appropriate *stride- (1–4)* is selected depending on *flag- (1–4)*. The *address-generate* block uses a 32-bit adder to add the selected stride to the previous address. On either an exception or a stall, only the *prev-address* value needs to be stored as the loop counters are stored by the hardware looping circuitry. For each of the four data structures/streams, the *last_val comparators* portion of the logic is shared, but the remaining hardware needs to be replicated.

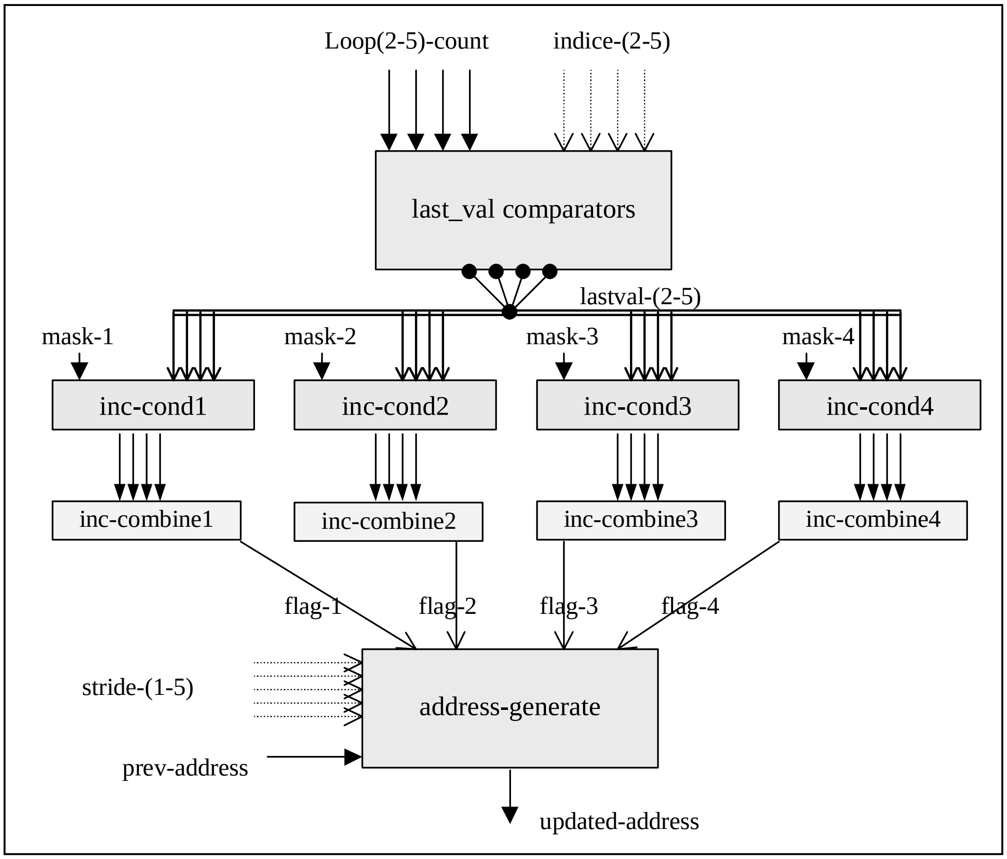


Fig. 14. Block diagram of address generation hardware (per data stream)

Looping – The MediaBreeze architecture incorporates five levels of loop nesting in hardware to eliminate branch instruction overhead for loop increments. A similar mechanism was commercially implemented in the TI ASC [24] (two levels of do-loop nesting in addition to a self-increment loop). Conventional DSP processors such as the Motorola 56000 and the TMS320C5x from TI also use such a technique for one or more levels of loop nesting. Fig. 15 shows the block diagram of the looping hardware. Loop index values are produced every clock cycle based on the loop bound for each level of nesting (bounds for each of the five loops are specified in the Breeze instruction). The value of a loop index varies from 1 (lower bound) to the corresponding loop bound (upper bound), and resets to its lower bound once the upper bound is reached in the previous cycle. The execution of the Breeze instruction ends when the outermost loop (loop1 in Fig. 15) reaches its upper bound. On encountering either an exception or a stall, the loop indices are stored and the increment logic is halted; the counting process is started once the exception/stall is serviced. Each of the five *comparators* (32-bit wide) operates in parallel to generate *flag* (1-bit wide) signals that are *priority encoded* to determine which one of the five loop counters to increment. When a loop counter is *incremented-by-1* (circuit for incrementing a 32-bit value

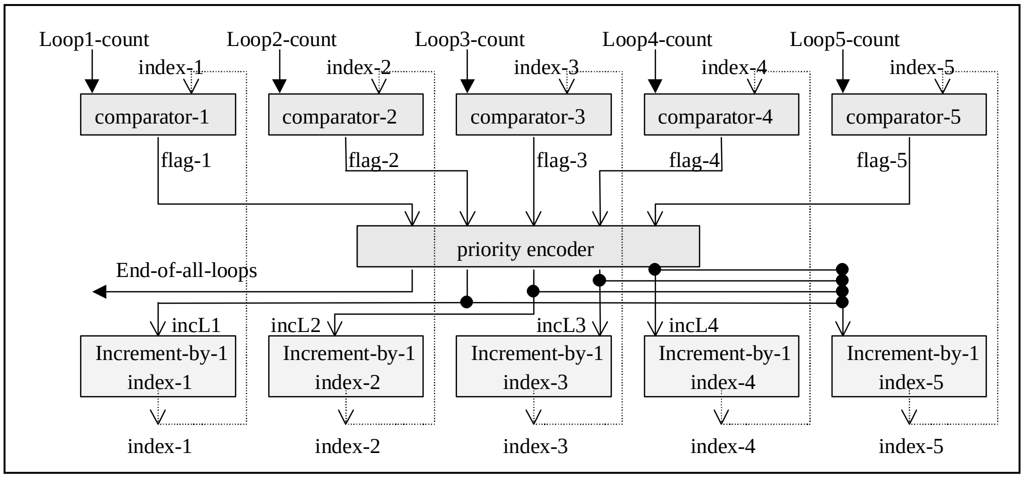


Fig. 15. Block diagram of the five hardware loops

by 1), all the loop counters belonging to its inner level are reset (for example, if loop3 is *incremented-by-1*, then loop4 and loop5 are reset to their lower bound).

Breeze instruction decoder – A stand-alone instruction decoder for the Breeze instructions eliminates the need to modify the conventional instruction decoder of current GPPs. A Breeze instruction needs to be decoded only once since various control parameters are stored in hardware registers after the decoding process. The implementation of the Breeze instruction decoder was merged into the address generation and looping circuitry.

Breeze instruction memory – The Breeze instruction memory stores the Breeze instruction once it enters the processor. We do not estimate the cost of this storage because the ASIC libraries are not targeted for memory cells. However, the area, power, and timing estimates of the Breeze instruction memory are similar to an SRAM structure. One Breeze instruction occupies 120 bytes. The Breeze instruction memory holds one or more Breeze instructions.

Existing hardware units – The remaining hardware units that are required for the operation of the MediaBreeze architecture are the SIMD computation unit, data reorganization, load/store units, and data station. These hardware units are already present in commodity SIMD GPPs. However, the Breeze instruction decoder controls the operation of these units as opposed to the conventional control path. This mandates an extra multiplexer to differentiate between control from the conventional control path and the Breeze instruction decoder. We do not model any of the existing hardware units.

6.3 Area, power, and timing results

Table 8 shows the composite estimates of timing, area, and power consumption for the hardware looping and address generation circuitry when implemented using the cell-based methodology. The power and area estimates in Table 8 correspond to a clock frequency of 1 GHz. The hardware cost of commercial SIMD implementations [25][26] is also shown in Table 8.

Area – The overall chip area required for implementing the hardware loops, address generation (for all four data streams), and the Breeze instruction interpreter (merged into looping and address generation logic) is approximately 0.31 mm^2 in the 0.18-micron library. In a 0.29-micron process, the increase in chip area for implementing the Visual Instruction Set (VIS) hardware into the Sparc processor family was 4 mm^2, MMX into the Pentium family was 15 mm^2, and AltiVec into the PowerPC family was 30 mm^2 [25]. In a 0.25-micron process, the AltiVec hardware was expected to occupy 15 mm^2. In a 0.18-micron technology, the die size of a Pentium III processor was 106 mm^2 with the MMX and SSE execution units requiring approximately 3.6 mm^2 [26]. Thus, the increase in area due to the MediaBreeze units for SIMD-related hardware is less than 10% and the overall increase in chip area is less than 0.3%.

Table .8. Timing, Area, and Power estimates for hardware looping and address generation (the Breeze instruction decoder was merged into the looping and address generation)

	Hardware Looping (5 loops)			Address Generation (per stream)		
	Time (ns)	Area (μm^2)	Power (mW)	Time (ns)	Area (μm^2)	Power (mW)
G12-p (0.18μ)	1.00 ns	72830 μm^2	88.57 mW	1.74 ns	57398 μm^2	85.16 mW
G11-p (0.25μ)	1.49 ns	273249 μm^2	249.30 mW	2.60 ns	165099 μm^2	193.20 mW

Area of commercial SIMD and GPP units for comparison [25][26]

VIS – 4 mm^2 in a 0.29-micron process
MMX – 15 mm^2 in a 0.29-micron process
AltiVec – 15 mm^2 in a 0.25-micron process
Pentium III processor – 106 mm^2 in a 0.18-micron process
MMX + SSE in a Pentium III processor – 3.6 mm^2 in a 0.18-micron process

Power – The power consumed by the looping, address generation (all four streams), and the Breeze instruction interpreter is approximately 430 mW in the 0.18-micron library. General-purpose processors with speeds over 1 GHz typically consume a power ranging from 50 W to 150 W and MediaBreeze hardware increases power by less than 1%. We believe that the overall energy consumption of the MediaBreeze architecture would be less than that of a superscalar processor with SIMD extensions because the Breeze instruction reduces the total dynamic instruction count (0.2 to 40% in our media applications not including kernels). The instruction fetch and issue logic are expected to consume greater than 50% of the total execution power (not including the clock power) in future speculative processors [27]. Once a Breeze instruction is interpreted, the instruction fetch, decode, and issue logic in the superscalar processor can be shutdown to save power.

Timing — Pipelining the hardware looping logic into two stages (in a 0.18-micron technology) would allow for incorporating it into current high-speed superscalar out-of-order processors with over 1 GHz clock frequency. Similarly the address generation stage needs to be divided into three pipe stages to achieve frequencies greater than 1 GHz. The timing results show that incorporating the MediaBreeze hardware into a high-speed processor does not elongate the critical path of the processor (after appropriate pipelining). The Breeze instruction decoder multiplexers that control the hardware units introduce an extra gate delay in the pipeline. However, using a cell-based methodology gives a conservative estimate while custom design (typically used in commercial GPPs) would allow for greater clock frequencies for the added MediaBreeze hardware. In spite of adding five pipeline stages, the overall pipeline depth of a processor is not affected because the looping and address generation stages bypass the conventional fetch, decode and issue pipeline stages.

7 Related Work

The proposed solution combines the advantages of SIMD, vector, DAE, and DSP processors. The DAE concept present in the IBM System 360/370, CDC 6600 [30], CDC7600, CRAY-1, CSPI MAP-200, SDP [31], PIPE [32], SMA [19], WM [33], DS [34], etc demonstrated the potential of decoupling memory accesses and computations [14][15]. There also has been research in specialized access processors and address generation coprocessors [13][35]. The concept of embedding loops in hardware was implemented commercially in the TI ASC [24] (do-loop in this case). The SMA architecture [19] provided similar flexibility in accessing matrices. This concept was seen to be successful in all these machines as well as many DSP processors [18]. Typically all these techniques were successful only for a limited class of applications. This work extends beyond past work to create an integrated environment in which both media and general-purpose workloads can excel.

Previous media characterizations have concentrated on measuring the performance benefits of media extensions [5][6][7][8]. There are a few research efforts in identifying the bottlenecks in exploiting sub-word parallelism using SIMD extensions. Fridman discusses approaches to data alignment for sub-word parallelism in the TigerSharc processor using four sub-word MAC units in [28]. Thakkar and Huff discuss the need for data alignment for SSE extensions in [29]. We perform a comprehensive detection of bottlenecks in SIMD-style extensions.

The proposed Breeze instruction captures all the overhead/supporting operations in addition to capturing the DLP in the true/core computation and has some similarities to the vector parameter file in the TI ASC machine. Compiling for SIMD extensions is still in its infancy [50][51][52][53][54][55]; a MediaBreeze compiler is a challenge. Given the ability of the TI ASC vectorizing compiler to handle the vector parameter file leads us to believe that programming with Breeze instructions is going to be an achievable hurdle in exploiting the MediaBreeze architecture.

Corbal et al. [36] proposed to exploit DLP in two dimensions instead of one dimension as in current SIMD extensions. A 20% performance improvement was achieved using their Matrix-oriented architecture named MOM. However, the overhead factor is not significantly reduced. Vassiliadis et al. [37][38] have concurrently proposed the Complex Streamed Instruction set (CSI) that can exploit two levels of looping. Though they are able to eliminate some overhead because each of their complex instructions can eliminate two loops, our solution is more comprehensive. Lee and Stoodley [39] proposed simple vector microprocessors for media applications, but they used in-order simple processors for scalar processing and vectors for media processing. While we commend the approach, such an architecture cannot achieve good performance over several application domains because the scalar processor is in-order. Ranganathan et al. [5] observe that out-of-order execution is beneficial to media applications. There are several components in many multimedia applications that cannot exploit DLP, but require good branch prediction and speculation to exploit ILP, and hence we also favor the use of the out-of-order processor. It is important to have a general-purpose processor achieve sustained performance on different domains of workloads.

Rixner et al. [40] developed the Imagine architecture for bandwidth-efficient media processing. This architecture is based on clusters of ALUs processing large data streams and is built as a co-processor for a high-end multimedia system. The methodology adopted is to put additional computation units, while our approach is to improve the utilization of the existing computation units by reducing the overhead. Another related effort is the PipeRench coprocessor that is reconfigurable [41]. The Burroughs Scientific Processor (BSP) [42] was a pure-SIMD array processor that had special-purpose hardware (called Alignment networks) for packing and unpacking data. In addition, they have powerful SIMD instructions of which many are being used in current SIMD extensions. Vermuelen et al. [20] described how DCT, Reed-Solomon code and other similar media oriented operations could be enhanced with a hardware accelerator that works in conjunction with a GPP. However, the accelerator has to be designed for each algorithm. Retargeting the accelerator to another algorithm incurs significant effort, while, in our case, only Breeze instruction encoding needs to be performed.

8 Conclusion

This paper analyzes multimedia workloads and proposes architectural enhancements for improving their performance on general-purpose processors. Based on an investigation of loop structures and access patterns in multimedia algorithms, we find that significant amount of parallelism lies outside the innermost loops (between loop levels 3 and 6 as indicated in Table 3), and it is difficult for SIMD units to exploit the parallelism. The characteristics preventing SIMD computation units from computing at their peak rate are analyzed. The major findings of the bottleneck analysis are:

- Approximately 75-85% of instructions in the dynamic instruction stream of media workloads are not performing true/core computations. They are performing address generation, data rearrangement, loop branches, and loads/stores.
- The efficiency of the SIMD computation units is very low because of the overhead/supporting instructions. Our measurements on a Pentium III processor with a variety of media kernels and applications illustrate SIMD efficiency ranging only from 1% to 12%.
- Increasing the number of SIMD execution units does not impact performance positively leading us to conclude that resources for overhead/supporting instructions need to be scaled. We observe that a significant increase in scalar resources is required to increase the SIMD efficiency using conventional ILP techniques. An 8-way or 16-way integer processor is necessary to process the overhead instructions for the SIMD width in current processors.

The paper then addresses the issue of executing the overhead instructions efficiently. Many recent enhancements such as increasing the SIMD width have targeted exploiting additional parallelism in the true/core computation while the MediaBreeze architecture proposed in the paper focuses on the overhead instructions and the ability of the hardware to eliminate, alleviate, and overlap the overhead. MediaBreeze exploits the nature of the overhead instructions to devise simple hardware by combining the advantages of SIMD, vector, DAE, and DSP processors. The major findings are:

- Eliminating and reducing the overhead using specialized hardware that works in conjunction with state-of-the-art superscalar processor and SIMD extensions can dramatically improve the performance of media workloads without deteriorating the performance of general-purpose workloads. On multimedia kernels, we find that a 2-way processor with SIMD extensions augmented with hardware support significantly outperforms a 16-way processor with SIMD extensions.
- On multimedia applications, a 2-way processor with SIMD extensions with the supporting MediaBreeze hardware outperforms a 4-way superscalar processor with SIMD extensions. Similarly a 4-way processor with SIMD extensions added with MediaBreeze hardware is superior to an 8-way superscalar with SIMD extensions.
- The cost of adding the MediaBreeze hardware to a SIMD GPP is negligible compared to the performance improvements. Using ASIC synthesis tools and libraries, we find that the MediaBreeze

hardware units occupy less than 0.3% of the overall processor area, consumes less than 1% of the total processor power, and on appropriate pipelining does not elongate the critical path of a GPP.

Our analysis shows that increasing the number of SIMD execution units to get more parallelism is not the right approach. But if any media processor designer decides to exploit more parallelism just by scaling the current architectures, they should scale the non-SIMD part much more aggressively than the SIMD part.

Acknowledgments: We thank members of the Laboratory for Computer Architecture for their comments and suggestions that improved several drafts of this paper.

References

[1] R. B. Lee, "Multimedia extensions for general-purpose processors," *Proc. IEEE Workshop on Signal Processing Systems*, pp. 9-23, Nov. 1997.

[2] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales, "AltiVec extension to PowerPC accelerates media processing," *IEEE Micro*, vol. 20, no. 2, pp. 85-95, Mar/Apr 2000.

[3] TMS320C64x DSP Technical Brief. Available:
http://www.ti.com/sc/docs/products/dsp/c6000/c64xmptb.pdf.

[4] J. Fridman and Z. Greenfield, "The TigerSHARC DSP architecture," *IEEE Micro*, vol. 20, no. 1, pp. 66-76, Jan/Feb. 2000.

[5] P. Ranganathan, S. Adve, and N. Jouppi, "Performance of image and video processing with general-purpose processors and media ISA extensions," *Proc. IEEE/ACM Int. Sym. on Computer Architecture*, pp. 124-135, May 1999.

[6] E. Salami, J. Corbal, M. Valero, and R. Espasa, "An Evaluation of different DLP alternatives for the embedded domain," *Proc. Workshop on Media Processors and DSPs in conjunction with Micro-32*, Nov. 1999.

[7] R. Bhargava, L. K. John, B. L. Evans, and R. Radhakrishnan, "Evaluating MMX technology using DSP and multimedia applications," *Proc. IEEE/ACM Int. Sym. on Microarchitecture*, pp. 37-46, Dec. 1998.

[8] H. V. Nguyen, and L. K. John, "Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology," *Proc. ACM Int. Conf. on Supercomputing,* pp. 11-20, Jun. 1999.

[9] Sample source code for the Benchmarks. Available:
http://www.ece.utexas.edu/projects/ece/lca/mediabenchmarks/

[10] C. Lee, M. Potkonjak and W.H. Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", *Proc IEEE/ACM Int. Sym. on Microarchitecture,* pp. 330-335, Dec 1997.

[11] D. Burger, and T. M. Austin, "The SimpleScalar tool set," Version 2.0. *Technical Report 1342*, Univ. of Wisconsin-Madison, Comp. Sci. Dept, 1997.

[12] J. Fritts, and W. Wolf, "Dynamic parallel media processing using speculative broadcast loop (SBL)," Proc. *Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (held in conjunction with IPDPS'01)*, Apr. 2001.

[13] P. T. Hulina, L. D. Coraor, L. Kurian, and E. John, "Design and VLSI implementation of an address generation coprocessor," *IEE Proc. on Computers and Digital Techniques*, vol. 142, No. 2, pp. 145-151, Mar. 1995.

[14] J. E. Smith, “Decoupled access/execute computer architectures,” *ACM Trans. on Computer Systems*, vol. 2, No. 4, pp. 289-308, Nov. 1984.

[15] J. E. Smith, S. Weiss, and N. Y. Pang, “A simulation study of decoupled architecture computers,” *IEEE Trans. on Computers*, vol. C-35, No. 8, pp. 692-701, Aug. 1986.

[16] J. Corbal, R. Espasa, and M. Valero, "On the efficiency of reductions in micro-SIMD media extensions,” *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2001.

[17] Intel Architecture Optimization Reference Manual. Available: *http://developer.intel.com/design/pentiumii/ manuals/245127.htm.*

[18] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee. *DSP Processor Fundamentals: Architectures and Features*, Chapter 8, IEEE Press series on Signal Processing, ISBN 0-7803-3405-1, 1997.

[19] A. R. Pleszkun, and E. S. Davidson, “Structured memory access architecture,” *Proc. IEEE Int. Conf. on Parallel Processing*, pp. 461-471, 1983.

[20] F. Vermeulen, L. Nachtergaele, F. Catthoor, D. Verkest, and H. De Man, “Flexible hardware acceleration for multimedia oriented microprocessors,” *Proc. IEEE/ACM Int. Sym. on Microarchitecture*, pp. 171-177, Dec. 2000.

[21] Synopsis Sold Documentation, version 2000-0.5-1. Distributed with Synopsys CAD tools.

[22] LSI Logic ASIC technologies. Available: http://www.lsilogic/products/asic/technologies/index.html.

[23] LSI Logic ASKK Documentation System. Distributed with LSI Logic CAD tools.

[24] H. G. Cragon, and W. J. Watson, “The TI advanced scientific computer.” *IEEE Computer Magazine*, pp. 55-64, Jan. 1989.

[25] L. Gwennap, “AltiVec vectorizes PowerPC,” *Microprocessor Report*, vol. 12, no. 6, May 11, 1998.

[26] Pentium III implementation (IA-32). Available: http://www.sandpile.org/impl/p3.htm.

[27] K. Wilcox and S. Manne, “Alpha processors: A history of power issues and a look at the future,” *Cool Chips Tutorial in conjunction with IEEE/ACM Int. Sym. on Microarchitecture*, Nov. 1999.

[28] J. Fridman, “Sub-word parallelism in digital signal processing,” *IEEE Signal Processing Magazine*, pp. 27-35, vol. 17, no. 2, Mar. 2000.

[29] S. Thakkar and T. Huff, “Internet streaming SIMD extensions,” IEEE Computer Magazine, pp. 26-34, vol. 32, no. 12, Dec. 1999.

[30] J. E. Thornton, “Parallel operation in the Control Data 6600,” *Fall Joint Computers Conference*, vol. 26, pp. 33-40, 1961.

[31] R. R. Shively, “Architecture of a programmable digital signal processor,” *IEEE Trans. Computers*, vol. C-31, pp. 16-22, Jan. 1978.

[32] J. R. Goodman, T. J, Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, “PIPE: A VLSI decoupled architecture,” *Proc. IEEE Int. Sym. on Computer Architecture*, pp. 20-27, Jun. 1985.

[33] Wm. A. wolf, “Evaluation of the WM architecture,” *Proc. IEEE/ACM Int. Sym. on Computer Architecture*, pp. 382-390, May 1992.

[34] Y. Zhang, and G. B. Adams, “Performance modeling and code partitioning for the DS architecture,” *Proc. IEEE/ACM Int. Sym. on Computer Architecture*, pp. 293-304, Jun. 1998.

[35] A. S. Berrached, P. T. Hulina, and L. D. Coraor, “Specification of a coprocessor for efficient access of data structures,” *Proc. Ann. Hawaii Int. Conf. on System Sciences*, pp. 496-505, Jan. 1992.

[36] J. Corbal, M. Valero, and R. Espasa, “Exploiting a new level of DLP in multimedia applications,” *Proc. IEEE/ACM Int. Sym. on Microarchitecture*, pp. 72-79, Nov. 1999.

[37] S. Vassiliadis, B. Juurlink, and E. A. Hakkennes, “Complex streamed instructions: introduction and initial evaluation,” *Proc. IEEE Euromicro Conf*., vol. 1, pp. 400-408, Sep. 2000.

[38] B. Juurlink, D. Tcheressiz, S. Vassiliadis, and H. Wijshoff, "Implementation and evaluation of the complex streamed instruction set,” *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2001.

[39] C. G. Lee, and M. G. Stoodley, “Simple vector microprocessors for multimedia applications,” *Proc. IEEE/ACM Int. Sym. on Microarchitecture*, pp. 25-36, Dec. 1998.

[40] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens, "A bandwidth-efficient architecture for media processing," *Proc. IEEE/ACM Int. Sym. on Microarchitecture*, pp. 3-13, Dec, 1998.

[41] S. C. Goldstein, H. Schmit, M. Moe, M. Nudiu, S. Cadambi, R. R. Taylor, and R. Laufer, "PipeRench: A coprocessor for streaming multimedia acceleration," *Proc. IEEE/ACM Int. Sym. on Computer Architecture*, pp. 28-39, May 1999.

[42] D. J. Kuck, and R. A. Stokes, "The Burroughs scientific processor (BSP)," *IEEE Trans. on Computers*, vol. 31, no. 5, pp. 363-376, 1982.

[43] T. M. Conte, P. K. Dubey, M. D. Jennings, R. B. Lee, A. Peleg, S. Rathnam, M. Schlansker, P. Song, and A. Wolfe, "Challenges to combining general-purpose and multimedia processors," *IEEE Computer Magazine*, pp. 33-37, Dec. 1997.

[44] P. Ranganathan, S. Adve, and N. Jouppi, "Reconfigurable caches and their application to media processing," *Proc. IEEE/ACM Int. Sym. on Computer Architecture*, pp. 214-224, Jun. 2000.

[45] S. A. Mckee, "Maximizing memory bandwidth for streamed computations," *Ph.D. Thesis*, School of Engineering and Applied Science, University of Virginia, May 1995.

[46] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit," *Proc. IEEE/ACM Int. Sym. on Computer Architecture*, pp. 225-235, Jun. 2000.

[47] H. Lieske, J. Wittenburg, W. Hinrichs, H. Kloos, M. Ohmacht, P. Pirsch, "Enhancements for a Second Generation Parallel Multimedia-DSP," *Proc. Workshop on Media Processors and* DSPs in conjunction with Micro-32, Nov. 1999.

[48] D. Talla and L. K. John, "Cost-effective hardware acceleration of multimedia applications," *Proc. IEEE Int. Conference on Computer Design*, pp. 415-424, Sep. 2001.

[49] D. Talla, "Architectural techniques to accelerate multimedia applications on general-purpose processors," Ph.D. thesis, Dept. of Electrical and Computer Engineering, The University of Texas, Austin, Aug. 2001. Available:
http://www.ece.utexas.edu/projects/ece/lca/ps/deepu_talla_dissertation.pdf.

[50] N. Sreraman and R. Govindarajan, "A vectorizing compiler for multimedia extensions," *International Journal of Parallel Programming*, vol. 28, no. 4, pp. 363-400, Aug. 2000.

[51] G. Pokam, J. Simonnet, and F. Bodin, "A retargetable preprocessor for multimedia instructions," *Proc. Workshop on Compilers for Parallel Computers*, Jun. 2001.

[52] A. Bik, M. Girkar, P. Grey, and X. Tian, "Experiments with automatic vectorization for the Pentium 4 processor," *Proc. Workshop on Compilers for Parallel Computers*, Jun. 2001.

[53] G. Cheong and M. S. Lam, "An optimizer for multimedia instruction sets," *Proc. SUIF Compiler Workshop*, Stanford University, Aug. 1997.

[54] S. P. Amarasinghe, "Parallelizing compiler techniques based on linear inequalities," Ph.D. thesis, Computer Systems Laboratory, Stanford University, Jan. 1997.

[55] M. Wolfe. *High performance compilers for parallel computing*, Addison-Wesley Publishing Company, Reading, MA, 1996.

[56] D. Rice, "High-Performance Image Processing Using Special-Purpose CPU Instructions: The UltraSPARC Visual Instruction Set," Master's thesis, Stanford University, 1996.

[57] D. Talla and L. K. John, "MediaBreeze: A decoupled architecture for accelerating multimedia applications," *ACM Computer Architecture News*, ACM Press, ISSN 0163-5964, vol. 29, no. 5, Dec. 2001.

[58] D. Talla, L. K. John, and D. Burger, "Hardware support to reduce overhead in fine-grain media codes," *Technical Report*, Laboratory for Computer Architecture, Dept. of Electrical and Computer Engineering, The University of Texas, Austin, Nov. 2001.

Figures
=======

Tables
=====

Table. 4. Execution statistics and efficiency of media programs

Table. 5. Performance (IPC) with unit cycle memory accesses and perfect branch prediction

Table. 6. Performance of the MediaBreeze architecture with prefetching

Table. 7. Cell-based Libraries (LSI Logic) used in synthesis

Table .8. Timing, Area, and Power estimates for hardware looping and address generation (the Breeze instruction decoder was merged into the looping and address generation)

Biographies of the authors (Pictures provided as postscript files)

Deepu Talla received his Ph.D. in Computer Engineering from The University of Texas at Austin in August 2001. He is currently a System Architect in the Worldwide Imaging and Audio Group at Texas Instruments, Inc. in Dallas. His research interests are in computer architecture, workload characterization, performance evaluation and benchmarking, multimedia processing, and ASIC/FPGA design. He is a member of the IEEE, IEEE Computer Society, ACM, and ACM SIGARCH.

Lizy Kurian John is an associate professor in the Department of Electrical and Computer Engineering at the University of Texas at Austin. She received her PhD degree in computer engineering from the Pennsylvania State University. Her research interests include high performance microprocessor architecture, memory systems, computer performance evaluation and benchmarking, workload characterization, and optimization of architectures for emerging workloads. She has published papers in the IEEE Transactions on Computers, IEEE Transactions on VLSI, ACM/IEEE International Symposium on Computer Architecture (ISCA), ACM International Conference on Supercomputing (ICS), IEEE Micro Symposium (MICRO), IEEE High Performance Computer Architecture Symposium (HPCA), etc., and has a patent for a Field Programmable Memory Cell Array chip. Her research is supported by the US National Science Foundation (NSF), the State of Texas Advanced Technology program, DELL Computer Corporation, Tivoli, IBM, AMD, Motorola, Intel, and Microsoft Corporations. She is the recipient of an NSF CAREER award and a Junior Faculty Enhancement Award from Oak Ridge Associated Universities. She is a senior member of the IEEE and a member of the IEEE Computer Society and ACM and ACM SIGARCH. She is also a member of Eta Kappa Nu, Tau Beta Pi, and Phi Kappa Phi.

Doug Burger has been an Assistant Professor of Computer Sciences and Electrical & Computer Engineering at the University of Texas at Austin since 1999. He received his Ph.D. in Computer Sciences from the University of Wisconsin-Madison, and his B.S. from Yale University in 1991. His main research area is computer architecture, and his interests span compilers, operating systems, emerging technologies, and distance running. He is co-leader of the TRIPS project at UT-Austin, a 2000 NSF Career Award recipient, an IBM Center for Advanced Studies Fellow, and a Sloan Foundation Research Fellow.