Hardware Support to Reduce Overhead in Fine-Grain Media Codes

Deependra Talla, Lizy K. John, and Doug Burger
Technical Report – Laboratory for Computer Architecture
The University of Texas at Austin

Abstract - The growing importance of media and media-like codes has caused general-purpose processors to incorporate SIMD-like extensions, such as MMX, SSE, and AltiVec. While these media extensions do improve performance, significant parallelism in these codes remains unexploited. In this paper, we propose and evaluate a programmable loop engine (PLE for short) that executes media codes and kernels efficiently by moving most of the overhead associated with the media programs into hardware. We study a range of media codes to determine which features and characteristics of these programs are sufficiently frequent to merit implementation in hardware. The common classes of operations that we find are *multiple nested loop control*, *address generation, data transformations,* and *streaming memory operations*. We quantify the fraction of instructions that can be eliminated for our benchmarks, showing that it is over 65% on average. We evaluate a design of one PLE, showing how it can be integrated cleanly with a general-purpose pipeline, defining an instruction set interface, and quantifying its area and power overhead with a VHDL implementation. We find that the PLE requires a mere 10% of the area required by the MMX and SSE extensions. Furthermore, the PLE improves performance of a set of media kernels, over that of a 4-way SIMD processor, by more than 7X on average, and accelerates a set of five media applications by an average of 54%. For all kernels and 3 of 5 applications, a 4-way processor with the PLE outperforms an 8-way SIMD processor without the PLE. For all the kernels and 2 of the 5 applications, a 4-way processor enhanced with the PLE outperforms even a 16-way processor.

1 Introduction

The growing importance of audio and video applications has resulted in media-specific support in most major instruction set architectures. Since general-purpose processors are not well suited to executing codes with copious fine-grain parallelism, media extensions permit a higher rate of execution within a mostly unmodified general-purpose pipeline. Example of media extensions are Intel's MMX, SSE1, and SSE2, Sun's VIS, HP's MAX, Compaq's MVI, MIPS's MDMX, and Motorola's AltiVec [1][2]. Such SIMD (single instruction multiple data) extensions 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]. For many codes, however, the overhead associated with the loops themselves limits the parallelism that can be achieved through SIMD-like media extensions. In this paper, we focus on using hardware support to reduce the time spent on executing the instructions that support the SIMD instructions, rather than focusing on accelerating the computational, fine-grain parallel instructions themselves.

We first analyze a range of media workloads to determine the amount of serial overhead in these applications. We find that more than 75% of instructions that are executed in the media instruction stream are not the computation instructions, but are instructions that are managing data and control for the computation. Amdahl's

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.

law dictates that these overhead instructions will limit the fine-grain parallelism that may be extracted from these media codes. We show an example of this phenomenon in Table 1, which shows execution time of a one-dimensional discrete cosine transform (1-D DCT) kernel that is used in the JPEG image standard and the MPEG video standard. The 1-D DCT is essentially a multiply of two 8x8 matrices. In a non-MMX implementation, if all of the compute units were fully utilized, the multiplication would require 512 cycles. With MMX, the full utilization of the functional units would complete the DCT in 128 cycles. We ran this code on a Pentium III processor both with and without prefetching, and with and without MMX. We see that while prefetching and MMX both provide performance boosts, the level of parallelism that is exploited is far from ideal. Without MMX, a potential five-fold speedup remains even after prefetching, and with MMX, more than 13-fold improvement in performance is possible, if the computations were to be executed at the peak computation rate.

Since we are studying media workloads running on general-purpose processors, any added hardware support should support efficient execution of a broad range of media loops. The workload analysis we present in this paper breaks overhead instructions and necessary supporting mechanisms down into four major classes, which cover the vast majority of overhead instructions contained in these media loops: *loop processing, address generation, data transformations,* and *streaming memory accesses*. Loop processing involves maintaining the loop indices and issuing the branches that control the loops. Address generation consists of computing addresses for multi-dimensional arrays with various constant strides for each dimensional index. Data transformations reorder data accessed from memory for fast consumption by the SIMD instructions. The major transformations that we found include packing, unpacking, and multicasting of operands.

We evaluate a hardware design called a Programmable Loop Engine (PLE) intended to reduce the time spent executing general media loops. It is a hardware unit that takes a single 140-byte command. It can then execute up to five nested loops, accessing up to four streams of data (three input and one output) with different strides, transforming the data in ways required by the computation algorithm and writing the result stream back. The PLE contains hardware looping, hardware address generation and address transformation, and fetch mechanisms to stream data in and out. The PLE can be cleanly integrated with a superscalar pipeline, requiring minimal control to initiate and terminate hardware-supported loop execution. The single macro-instruction needs to be fetched and decoded only once per nested loop structure, resulting in a substantial reduction of fetch/decode activity and corresponding power savings.

Table 1. 1-D DCT without and with MMX on a Pentium III processor

	Pentium III		**Pentium III with MMX**	
	Cycles	**IPC**	**Cycles**	**IPC**
Maximum compiler optimizations	3500	1.47	2375	1.04
Perfect memory system/prefetching	2737	1.88	1578	1.56
Full compute utilization	~512	-	~128	-

In section 2 of this paper, we analyze a suite of media applications and kernels to quantify the loop overhead required to support media computation. We decompose that overhead into discrete classes, with the goal of designing a single, simple hardware engine that can provide broad coverage of the applications' media loops. In section 3, we describe our PLE design, and show how it is programmed by a single large instruction. We also describe how that instruction can be incorporated into a conventional out-of-order processor pipeline. In Section 4, we measure the performance improvements that the PLE provides over a pipeline that already uses SIMD ISA extensions. For a 4-way baseline, we show speedups of 13x, 11x, 8x and 2.3x on the four kernels we evaluate, and performance improvements of 3.5x, 8%, 94%, 61%, and 1% on the five applications we measure. Similar or higher speedups are obtained with 8-way and 16-way baseline configurations. In Section 5 we quantify the area and power overhead of the proposed PLE by analyzing its VHDL, as well as to validate the timing assumptions that we used in our simulation environment. In Section 6, we discuss some of the copious related work, and we conclude the paper in Section 7.

2. Analyzing Loop Overheads in Media Code

We list the nine benchmarks that we used to study the characteristics of media code in Table 2. We include the dynamic instruction counts to show that the counts are sufficiently high to amortize cold-start effects in the simulation environment. Our suite includes several common media applications (*g711, aud, jpeg, ijpeg*, and *decrypt*) and four common 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. Several of these benchmarks are also part of media benchmark suites such as MediaBench [10]. For those benchmarks that are not included in the MediaBench suite, we have made the source code available on-line [9].

Table. 2. Description of the multimedia benchmarks

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

To provide hardware support that can accelerate a wide range of media codes with a simple design, we must first understand what is common across these media codes. An analysis of our benchmark suite led to the following four categories of "computational support" operations: (1) *loop processing, (2) address generation, (3) data transformation, and (4) streaming memory accesses*. To show an example of the relative frequencies of these categories in a media loop, we show the assembly code of a one-dimensional discrete cosine transform (1-D DCT) routine in Fig. 1. This assembly code is for Pentium III processor based on the P6 microarchitecture [17], and was compiled using maximum compiler optimizations (including loop unrolling) and intrinsics by Intel C/C++ compiler version 4.5.

In the assembly code, we have marked the *true computation instructions* in boldface. These instructions perform the computational essence of the 1-D DCT, which are the multiply and the accumulate operations. The

Pentium III – MMX code

Assembly instruction		Comment	Category of Overhead
lea	ebx, DWORD PTR [ebp+128]	;load/address overhead	2
mov	DWORD PTR [esp+28], ebx	;load/address overhead	2
$B1$2:			
xor	eax, eax	;address overhead	2
mov	edx, ecx	;address overhead	2
lea	edi, DWORD PTR [ecx+16]	;load/address overhead	2
mov	DWORD PTR [esp+24], ecx	;load/address overhead	2
$B1$3:			
movq	mm1, MMWORD PTR [ebp]	;load overhead	4
pxor	mm0, mm0	;initialization overhead	3
pmaddwd	**mm1, MMWORD PTR [eax+esi]**	**;True Data Parallel Computation**	**Not Overhead**
movq	mm2, MMWORD PTR [ebp+8]	;load overhead	4
pmaddwd	**mm2, MMWORD PTR [eax+esi+8]**	**;True Data Parallel Computation**	**Not Overhead**
add	eax, 16	;address overhead	2
paddw	**mm1, mm0**	**;True Data parallel computation**	**Not Overhead**
paddw	**mm2, mm1**	**;True Data Parallel Computation**	**Not Overhead**
movq	mm0, mm2	;load related overhead	4
psrlq	mm2, 32	;SIMD reduction overhead	3
movd	ecx, mm0	;SIMD load overhead	4
movd	ebx, mm2	;SIMD load overhead	4
add	ecx, ebx	;SIMD conv. Overhead	3
mov	WORD PTR [edx], cx	;store overhead	4
add	edx, 2	;loop overhead	1
cmp	edi, edx	;branch related overhead	1
jg	$B1$3	;loop branch overhead	1
$B1$4:			
mov	ecx, DWORD PTR [esp+24]	;load/address overhead	2
add	ebp, 16	;loop/address overhead	1/2
add	ecx, 16	;address overhead	2
mov	eax, DWORD PTR [esp+28]	;load/address overhead	2
cmp	eax, ebp	;branch related overhead	1
jg	$B1$2	;loop branch overhead	1

Fig. 1. Optimized assembly code for the 1D-DCT routine (essentially an 8x8 matrix multiply).
The major component of discrete cosine transform is an 8x8 matrix multiply. In this example, we have not included the instructions necessary for transposing the second matrix, which would have increased the supporting overhead instructions further.

rest of the instructions in the loop are *overhead/supporting instructions*, such as the instructions necessary to perform address computations or to accomplish multiple levels of looping as required by the algorithm. We have broken them down into the four classes of overhead as defined in the previous paragraph. We note that some instructions may fall into more than one of our defined categories; since an operation on a register may be used for both loop management and address calculation.

It is clear that, in DCT at least, the majority of the instructions in the loop are overhead instructions. The high fraction of these instructions in the loop is partly 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.

If the SIMD units are widened, some of the loop instructions can perform more operations per instruction. In Fig. 1, however, only the instructions in bold can benefit from wider arithmetic units. However, as one sees, the vast majority of instructions in the instruction stream are just supporting the core computation instructions. Hence it is essential to improve their performance if the overall performance is to get better. It is clear from Amdahl's law that, given the fraction of overhead in the loop, wider SIMD units will provide only incremental performance benefits.

We showed the 1-D DCT example to illustrate the breakdown of computation versus overhead. In Fig. 2, we show a similar breakdown for six of the nine benchmarks we studied. We did not include *jpeg*, *ijpeg*, and *decrypt* in these results experiment because the source code for these three benchmarks includes initialization routines and file I/O. We ran the six benchmarks on a Pentium III processor with MMX support. In five of the benchmarks (*g711* was the exception), all core computation was executed using SIMD instructions. In Fig. 2 we decomposed the execution into the four defined classes, except that we were unable to separate address and loop arithmetic. For these codes, the overhead and supporting instructions account for 75%-85% of the dynamic instruction stream; the true computation instructions are a small percentage.

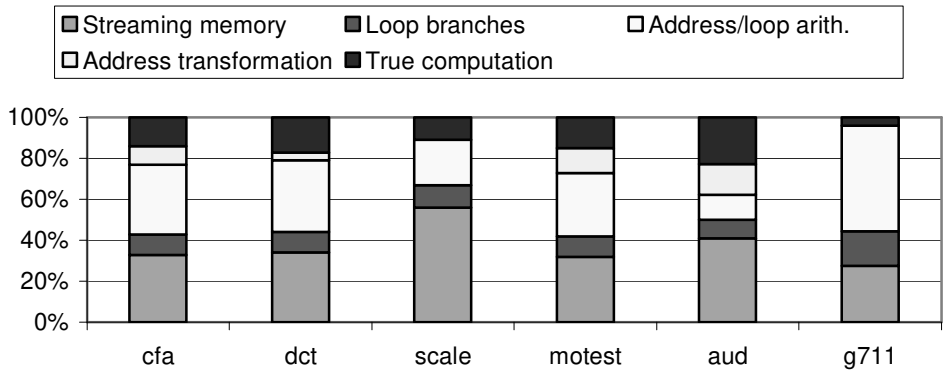

Fig. 2. Breakdown of dynamic instructions into various classes

2.1 Sources of overhead instructions

In this subsection we discuss the reasons that such a high fraction of these media loops are overhead. Essentially, media applications use nested loops and use multiple strides. Current general-purpose processors (GPPs) have little support to handle multiple loops or the abundant address generation efficiently. Hardware to generate multiple address sequences is not overly complicated, but current ISAs require a large number of instructions to produce them, as the available addressing modes are limited. Furthermore, there is not enough support for keeping track of multiple indices/strides efficiently in GPPs. 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 data-level parallelism in multimedia programs, there is a mismatch between the requirements of media applications (for address generation and nested loops) and the capabilities of GPPs to support the nested loops, memory operations, and address generation efficiently. We elaborate on the loop management and address generation overhead below.

Loop management: Desktop/workstation multimedia applications such as streaming video encoding/decoding (MPEG 1/2/4 and Motion JPEG), audio encoding/decoding (ADPCM, G.711, 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. Image and video applications operate on sub-blocks of two-dimensional data at a time. For example, the DCT algorithm operates on 8x8 segments of data in images of sizes like 1600x1200 pixels. These algorithms typically use multiple nested loops with statically known loop boundaries to process the sub-blocks and the data streams. To determine the depth of loop nests, we analyzed our benchmark suite and a number of other common media codes, and show the results in Table 3.

Table. 3. Loop nest depth and common addressing sequences in key media applications

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)	> 5	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

Address generation and stride management: The division of data into sub-blocks results in the data being accessed with different strides at various instances in the algorithm. Managing multiple strides results in numerous software instructions. In general, addressing sequences in media programs may be classified into sequences shown in Fig. 3. We have included the prevalent addressing sequences in the analyzed media applications in Table 3. By supporting sequential accesses, multiple strided accesses, reflected and shuffled data transformations in hardware, we can over the bulk of the addressing and data streaming across this spectrum of media codes.

2.2 Effect of prevalent overhead instructions

The net effect of the overhead instructions is that the SIMD execution units are often idle. In this section, we evaluate what utilization of the SIMD execution units is achieved for a subset of our media applications. Peak throughput can be achieved if the required operands are available as soon as the SIMD execution units are available. In Table 4, we show how well the SIMD units are utilized on the Pentium III baseline with MMX. The utilization achieved by the media extensions is low (1-12%), despite the abundance of data-level parallelism in these codes, because the supporting units are not able to feed the right data, in the right packed form, at the required rate.

The results of this section show both that the properties of the overhead instructions are common across a broad range of media applications, and that significant potential for improved performance exists due to the low execution of the SIMD units. In the next section, we describe the Programmable Loop Engine (PLE), which implements those common overhead operations in hardware, reducing the serial overhead and allowing a greater utilization of the SIMD units.

Given a sequence of length L, if A_m is address m in the range $0 \le m \le L-1$, 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. 3. Typical access patterns in multimedia and DSP kernels [13]

Table. 4. Execution statistics and utilization of media programs

Benchmark	Pentium III – MMX & SSE		
	Inst. Count	Actual Cycle count	Fraction of peak throughput
cfa	404,290,544	231,616,932	5.16 %
dct	188,798,806	123,944,326	6.2 %
scale	2,170,274	20,756,929	2.31 %
motest	156,734,613	113,623,185	3.38 %
aud	220,320,505	150,386,375	11.97 %
g711	59,066,806	64,006,729	1.12 %

3 Hardware Support for Efficient Media Processing

The media application characterization presented in the previous sections, especially Fig 2, presents a compelling case for providing hardware support for multiple levels of looping and address generation. Such facilities common in DSP processors can be integrated in a cost-effective fashion to general-purpose cores. While a variety of implementations are possible, we present an example architecture to study the effectiveness of the proposed hardware acceleration mechanisms in GPPs. We refer to this as the PLE (programmable loop engine) architecture. Fig. 4 illustrates the proposed scheme. The shaded units are the new additions. Essentially, we add a new instruction (called Programmable Loop or PL instruction) equivalent to a multidimensional vector instruction which indicates the multiple nested loops that are required in the computation, the different address strides, loop bounds, etc. Once the PL instruction is encountered, program execution proceeds in the shaded region without using the fetch, decode, rename and issue blocks of the original processor. The details of the added instruction and the added units are provided below. While we have made some choices regarding the number of levels of looping or simultaneous data streams, our objective is only to prove the effectiveness of the general concept of providing hardware support for these functionalities. Simple ASICs can perform these tasks efficiently; however, loss of programmability and flexibility is a weakness of that approach, and hence we do favor a programmable processor. But the point is that programmability does not need to extend to every aspect of the functionality. Programmability can be limited for these structured media applications, to improve efficiency.

The new hardware units (shaded blocks in Fig. 4) are the address generation units, hardware looping, and PL Instruction memory & decoder. Their functions are described below:

- **Loop processing:** To eliminate branch instruction overhead, PLE employs zero-overhead branch processing using dedicated hardware loop control and supports up to five levels of loop nesting. We chose to have five levels since that would be sufficient to handle the most common algorithms and routines used in media proc-

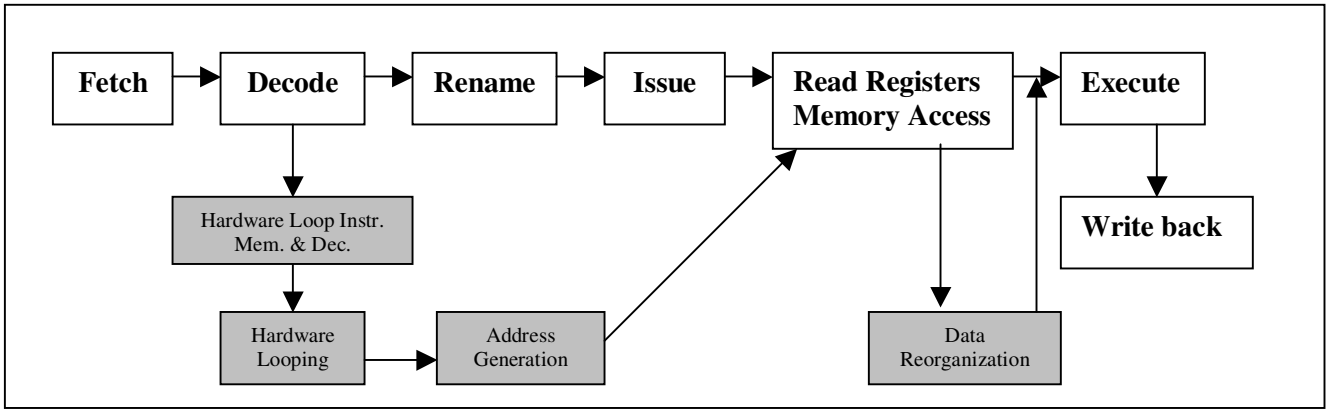

Fig. 4. A superscalar processor enhanced with the PLE Architecture

essing. All branches and instructions related to loop increments are handled by this technique. This approach is fairly simple and straightforward to implement and has been implemented in many conventional DSP processors such as the Motorola 56000 and TMS320C5x from Texas Instruments [18].

- **Address calculation:** The current PLE allows for three input data structures/streams and produces one output structure. The choice was made because many media algorithms can benefit from this capability (current SIMD execution units sometimes operate on three input registers to produce one output value). A dedicated hardware unit where address arithmetic hardware would generate all input and output address streams/data structures concurrently with the computations.
- **Data Reorganization:** 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, at times, a single element (16-bits wide) needs to be broadcast into all the four sub-words of a SIMD register (64-bits wide). PLE efficiently handles the task of reordering data with explicit hardware support. As an example of operations supported, there is multicasting of data into selected registers in programmable patterns. *Multicasting* eliminates 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.
- **Hardware loop instruction storage and decoding:** In order to program/control the hardware units in the PLE architecture; a special instruction called the PL instruction is formulated. The PL instruction is equivalent to a multidimensional vector instruction. The PL instruction memory stores these instructions once they enter the processor.

3.1 PLE implementation details

The major hardware additions, the looping circuitry and address generation circuitry are described below:

Loop support: Figure 5 illustrates the looping circuitry. 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 PL 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 PL instruction ends when the outermost loop (loop1 in Fig. 5) 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

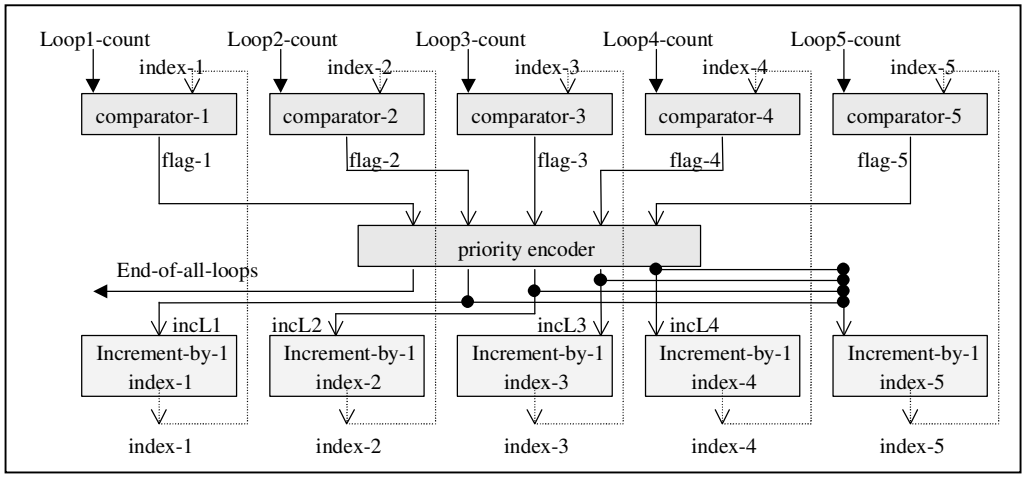

Fig. 5. Block diagram of the hardware looping circuitry

counter is *incremented-by-1* (circuit for incrementing a 32-bit value 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).

Address generation: The PLE 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 PL 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. 6 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 PL 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 PL 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.

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

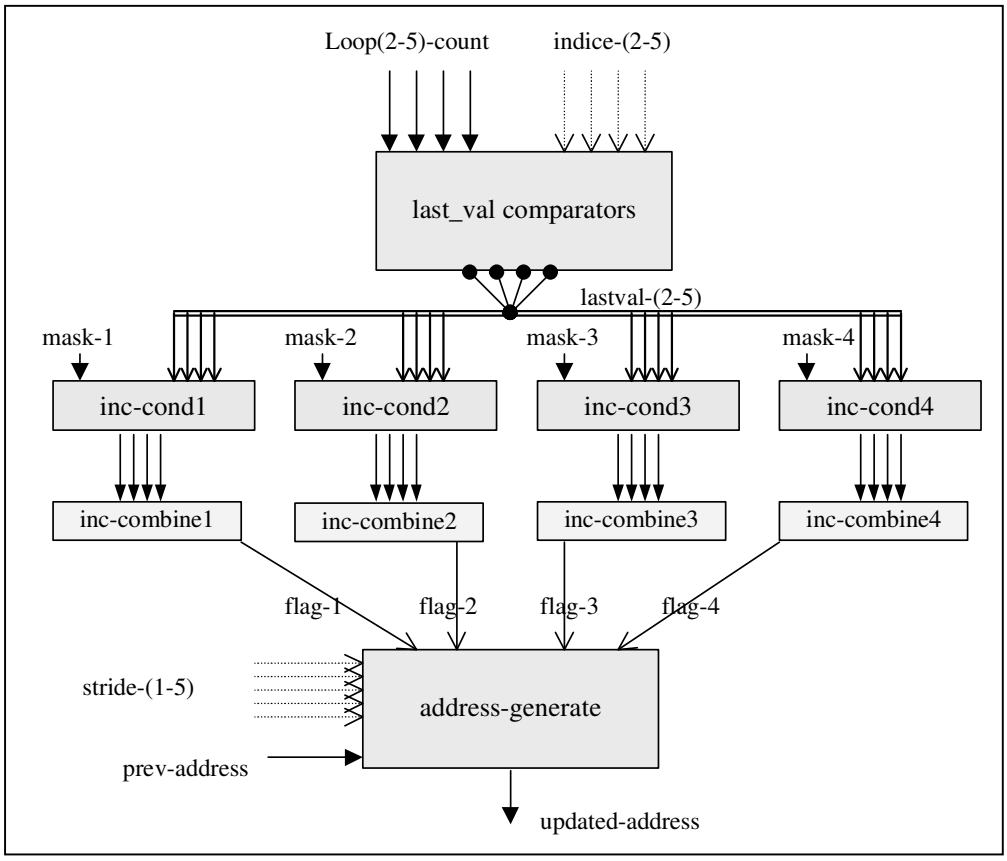

Fig. 6 Block diagram of address generation hardware (per data stream)

3.2 Required ISA support

The major ISA addition is a new instruction, the PL instruction, which conveys information on the various levels of loops and their strides to the hardware. The structure of the added PL instruction is explained in Fig. 7. The PL 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 increment/decrement values based on the loop-nest level. Depending on the mask values for each stream (indicated in the PL 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 PL instruction. 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 PL instruction, special hardware in the PLE performs this function. The PLE performs reduction operations and this is also indicated in the PL 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

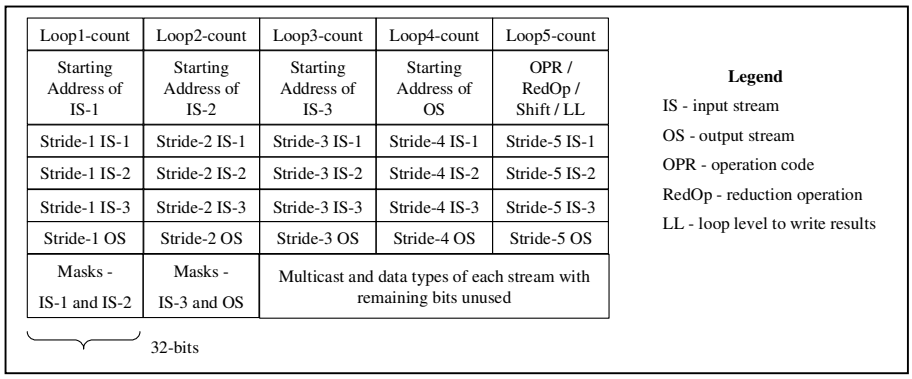

Fig. 7. Structure of the PL Instruction

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

Encoding all the overhead/supporting operations along with the SIMD true/core computation instructions has the advantage that the PL 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 PL 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 PLE architecture advantages similar to ASIC-based acceleration in [20].

It is possible that an exception or interrupt occurs while a PL 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. The PLE has registers to hold the loop parameters for all the loops.

3.2.1 PL instruction encoding example

The PL instruction is a densely encoded instruction and hence most media algorithms can be processed in just a few PL instructions. Fig. 8 illustrates the actions during the execution of a PL instruction using pseudo-code. In a scenario in which all the loop nests and data streams are processed, the PLE executes (in hardware) the following equivalent number of dynamic software instructions (in conventional ILP processors)

- 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

Common kernels such as the DCT, color space conversion, motion estimation, and filtering can be mapped to either one or two PL instructions. Fig. 9 illustrates the PL instruction mapping of the 1-D DCT 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 PL 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. 8. Pseudo-code illustrating operations during execution of a PLE compute instruction

```
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 re-
         quired */

              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. 9. PL instruction mapping of 1D-DCT

4. Performance Evaluation and Results

To measure performance of the PLE architecture, we modified the *PISA* version of Simplescalar-3.0 (sim-outorder) and simulated PL 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. Table 5 shows the speedup obtained for each of the benchmarks using the PLE architecture with a 4-way processor with SIMD extensions as the baseline. The baseline as well as the PLE enhanced architecture incorporate prefetching.

Table 5. Speedups of 4-way, 8-way and 16-way processors enhanced with PLE

	cfa	dct	mot	scale	aud	g711	jpeg	ijpeg	decrypt
4-way baseline	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
4-way + PLE	13.30	11.70	8.30	2.30	3.53	1.08	1.94	1.61	1.01
8-way baseline	1.24	1.29	1.33	1.96	1.47	1.68	1.18	1.40	1.18
8-way + PLE	26.00	22.14	16.00	4.58	5.49	1.83	2.23	2.09	1.19
16-way baseline	1.24	1.66	1.35	3.35	2.11	2.50	1.91	1.77	1.55
16-way + PLE	51.03	42.07	31.00	9.12	6.13	2.74	2.25	2.29	1.56

Incorporation of hardware looping and address generation coupled with savings in repeated fetch/decode activity results in significant speedup in all the kernels and several applications. One application that does not benefit is *decrypt. G711* and *decrypt* are applications that have the smallest fraction of SIMD instructions (i.e., it is the superscalar pipeline that accounts for a bulk of the execution time rather than the PLE pipeline). Considering geometric mean of speedups, incorporating the PLE enhances a 4-way processor by 7.3x in kernels and 1.54x in applications. We also computed the speedups of the 8-way enhanced architecture over the 8-way baseline and the 16-way enhanced architecture over the 16-way baseline architecture. In the case of the 8-way processor, the speedup in kernels is 10x and the speedup in applications is 1.63x (over the 8-way baseline). In the case of the 16-way architecture, the speedup over the 16-way baseline is 15.9x in kernels and 1.37x in applications.

One may also note that the incorporating PLE support for looping and address generation makes a 4-way processor perform better than an 8-way processor in all the kernels and 3 of 5 applications. For all the kernels and 2 of the 5 applications, a 4-way processor enhanced with the PLE outperforms even a 16-way processor. Hence if one is searching for complexity effective enhancements, incorporating hardware looping, address generation, and address transformation is an effective way to achieve it. We discuss the area, time and power savings associated with our proposal in the next section.

Power savings: Since the PL instruction is densely encoded, few PL instructions are needed for any media-processing algorithm. The number of dynamic instructions that need to be fetched and decoded is shrunk tremendously, 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 the PL instruction is interpreted, the instruction fetch, decode, and issue logic in the superscalar processor can be shut down for the duration of the loop nest. An indication of the power savings can be obtained by examining the savings in fetch and decode. As shown in Fig. 10, the use of the vector-style PL 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.

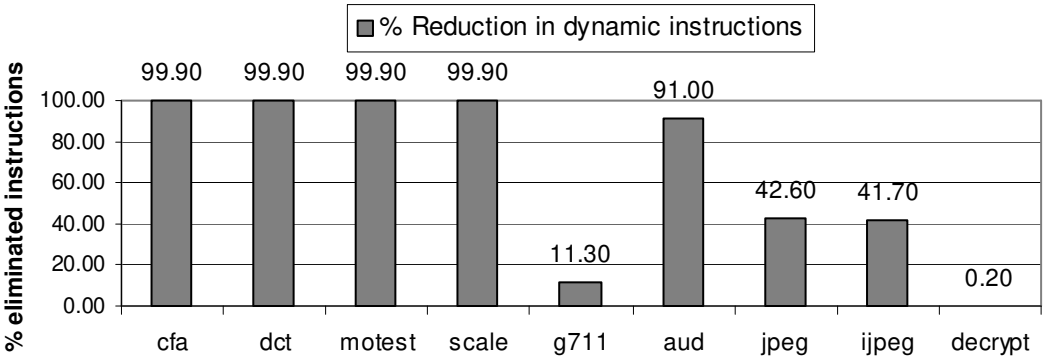

Fig. 10. Reduction in dynamic instructions by using the PLE architecture. Power savings proportional to the instruction count savings can be expected for fetch, decode and renaming energy.

5 Hardware cost of the PLE Architecture

In this section, we describe a VHDL implementation of the PLE, which we developed to estimate the PLE's area and power overhead, as well as to validate the timing assumptions that we used in our simulation environment. 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 6 lists the libraries and technologies used for evaluating the implementation cost.

Table. 6. 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.

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 gathered by the tool from simulation. The area, power, and timing estimates are obtained after performing maximum optimizations for performance in the synthesis tools. More information about the details of these tools can be found elsewhere [21].

Table 7 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 7 correspond to a clock frequency of 1 GHz. The hardware cost of commercial SIMD implementations [25][26] is also shown in Table 7. We discuss each of the three categories below.

Area: The overall chip area required for implementing the hardware loops, address generation (for all four data streams), and the PL instruction decoder (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 added hardware units is less than 10% of SIMD-related hardware and the overall increase in chip area is less than 0.3%.

Table 7. Timing, Area, and Power estimates for hardware looping and address generation
(The 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 PL instruction decoder 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, thus the PLE hardware would increase power by less than 1%. The overall energy consumption of the PLE architecture would be clearly less than that of a superscalar processor with SIMD extensions, since the PL instruction significantly reduces the total instruction count, as explained in Fig. 10.

Timing: The PLE hardware can be incorporated into a high-speed processor without elongating the critical path of the processor (after appropriate pipelining). In Table 7, some of the units in our implementation do not support a 1GHz clock, however, pipelining the hardware looping logic into two stages (in a 0.18-micron technology) would permit gigahertz frequencies. Similarly, the address generation stage needs to be divided into three pipe stages to achieve frequencies greater than 1 GHz.

6 Related Work

Corbal *et al.* [36] proposed to exploit DLP in two dimensions instead of one dimension as in current SIMD extensions. 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 results show that more overhead can be eliminated with 5 loops. 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. But such an architecture can perform well over limited domains only, 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 ap-

plications 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.

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 the PLE approach improves the utilization of the existing computation units by reducing looping and address generation overhead. Another related effort is the reconfigurable PipeRench coprocessor [41].

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]. 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. There also has been research in specialized access processors and address generation coprocessors [13][35] and decoupled access execute processors [30,31,32,19,33,34], which also tried to accelerate the overhead component of the instruction stream.

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, we still have a fully programmable engine.

7 Conclusions

While SIMD extensions have improved media application performance, there are additional bottlenecks that exist in the media instruction stream. . We study a range of media codes to determine which features and characteristics of these programs are sufficiently frequent to merit implementation in hardware. The common classes of operations that we find are *multiple nested loop control*, *address generation, data transformations,* and *streaming memory operations*. We note that:

- Approximately 75-85% of instructions in the dynamic instruction stream of media workloads are only supporting the actual/core computations. These instructions are mostly performing address generation, data rearrangement, loop branches, and loads/stores.
- The utilization of the SIMD computation units in current SIMD extensions is very low because of the copious overhead/supporting instructions. Our measurements on a Pentium III processor with a variety of media kernels and applications illustrate SIMD utilization ranging from 1% to 12%.

Then, we propose and evaluate a programmable loop engine (a PLE) that executes media codes and kernels efficiently by moving most of the overhead associated with the media loops into hardware.

- On the average, 65% of all instructions in the instruction stream can be eliminated with the addition of the proposed hardware. This leads to performance and power savings.
- Incorporating hardware looping and address generation into a 4-way processor with SIMD extensions, results in a speedup of up to 1.54X in applications and 7.3X in kernels.
- For all kernels and 3 of 5 applications, a 4-way processor with the PLE outperforms an 8-way SIMD processor without the proposed hardware. For all the kernels and 2 of the 5 applications, a 4-way processor enhanced with the PLE outperforms even a 16-way processor.
- The cost of adding the PLE hardware to a SIMD GPP is negligible compared to the performance improvements. We find that the PLE 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 solution essentially approximates the performance of hardware solutions, but retains the flexibility of software solutions. 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 right solution is to append simple hardware support for tasks that can be done efficiently and elegantly in hardware.

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 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 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. Link suppressed for BLIND review.

[10] C. Lee, M. Potkonjak and W.H. Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", *Proc. of 30th IEEE/ACM 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. Intl. 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 Intl. 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 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 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 Sym. on Computer Architecture*, pp. 20-27, Jun. 1985.
[33] Wm. A. wolf, "Evaluation of the WM architecture," *Proc. IEEE/ACM 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 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 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. 31st IEEE/ACM 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. 32nd IEEE/ACM 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. 26th IEEE/ACM 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 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 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] Tech report, Link suppressed for blind review.