

**RISC for Graphics: A Survey and Analysis of
Multimedia Extended
Instruction Set Architectures**

Grant Erickson
University of Minnesota
Department of Electrical Engineering
December 10, 1996
Fall Quarter

Electrical Engineering 8362
Professor: Dr. Pen-Chung Yew

**RISC for Graphics: A Survey and Analysis of
Multimedia Extended
Instruction Set Architectures**

Abstract

The recent explosion of computing applications rich in graphics and sound has placed new and taxing demands on today's microprocessors. These processors, designed with the intent of performing operations on integer and floating point numbers, have shown difficulty servicing requests from multimedia applications such as MPEG video on top of servicing requests from the operating system and other processes. Vendors have begun to take advantage of tremendous gate counts on their dies to implement a minimal set of multi-media extensions capable of accelerating media-intensive applications at little cost. In this paper, we survey and analyze several such implementations. We look at Hewlett-Packard's Multimedia Architectural Extensions (MAX-2), Intel's Multimedia Extensions (MMX), MIPS Technologies' MIPS V instruction set and Digital Multimedia Extensions (MDMX), and Sun Microsystem's Visual Instruction Set (VIS). Finally, we look at both a qualitative and quantitative performance analysis of these media enhanced processors by modifying some code for several applications to take advantage of an extension set. We find that while certain types of applications may yield a significant increase in performance with only modest coding effort, applications reliant on system libraries offer little or no opportunity for such improvement. In the latter case, porting system libraries to take advantage of these new instruction sets can offer the greatest benefit to the largest cross-section of applications in a transparent manner. Because of the lack of disadvantages in implementing multimedia extensions, those vendors who have not implemented similar technologies in their microprocessors have placed themselves at a competitive disadvantage in the marketplace.

Table of Contents

Abstract	i
Table of Contents.....	ii
Introduction.....	1
Architectural Enhancements.....	1
SIMD Techniques.....	2
Data Path Considerations.....	3
Instruction Set Overview	4
Arithmetic	5
Conversion and Reordering.....	6
Memory	6
Other.....	7
Software Support.....	7
Extension Summary	8
Performance Analysis.....	9
Test Software.....	9
Methodology	9
Results.....	10
<i>animabob</i>	11
<i>maplay</i>	12
Conclusion.....	14
References	16
Appendix A: Tested Configurations	18

Introduction

The recent explosion of computing applications rich in graphics and sound, such as the world wide web, has placed new and taxing demands on today's microprocessors. These processors, RISC and CISC alike, were designed with the intent of performing operations on integer and floating point numbers as well as character-based data. However, when asked to service the demands of MPEG video on top of servicing requests from the operating system and other processes, few general-purpose microprocessors are able to deliver the 30 frames per second recommended for MPEG video. To address this problem, systems vendors and users have had to employ the aid of dedicated media or digital signal coprocessors.

Rapid advances in semiconductor process technology have created an opportunity for microprocessor manufacturers to reverse this trend. Submicron design rules of 0.35 and 0.1 μm have left designers with exceptionally small die sizes with staggering gate counts. Such gains in gate count have allowed many microprocessor vendors to offer 64-bit versions of their processors in the same or smaller die area as their 32-bit counterparts. Beyond this, vendors have been unable to put this new found abundance of gates to further use. This is particularly true with RISC processors because of their lean logic cores. Hence, the response has typically been to increase the size of the on-chip caches.

Vendors have begun to realize however that this extra chip area and gate windfall can be used to implement a minimal set of multi-media extensions capable of accelerating media-intensive applications at little cost. Currently, four major vendors have shipped or are poised to ship microprocessors which have on-chip support for *media processing*—the processing and manipulation of digital media data types. With the advent of these media enhanced microprocessors comes the promise of significant gains in media-rich applications.

In the remainder of this paper we investigate the architectural opportunities that have allowed vendors to implement media processing on their microprocessors. Next, we cover the intrinsics of the instruction set enhancements that have been made by four vendors to their microprocessors. In particular, we look at Hewlett-Packard's Multimedia Architectural Extensions (MAX-2), Intel's Multimedia Extensions (MMX), MIPS Technologies' MIPS V instruction set and Digital Multimedia Extensions (MDMX), and Sun Microsystem's Visual Instruction Set (VIS). Finally, we look at both a qualitative and quantitative performance analysis of these media enhanced processors by modifying some code for several applications to take advantage of an extension set. We use MIPS's MIPS V and MDMX enhancements to do so.

Architectural Enhancements

The 32- and 64-bit microprocessors available today offer a tremendous level of performance for scientific and business applications. Unfortunately, they exhibit disappointing performance when dealing with digital media-based applications such as audio, video, and telephony.

The disappointing performance is not due to a flawed architecture or lack of raw computational power, but is rather due to the processor's inefficiency in dealing with digital media data types.

For example compact disc-quality digital audio uses 16-bit integer data; far smaller than a 64-bit machine word. As another example, digital video often uses red-green-blue (RGB) triplets or red-green-blue-alpha (RGBA) quadruplets, each component of which is an 8-bit integer. The inefficiency enters into play when we consider that these processors may perform only a single operation on a single set of operands in one instruction, yielding only a single result. As shown in Figure 1, with a 64-bit processor, operations on 16-bit audio samples leave 75% of the operand space unused.

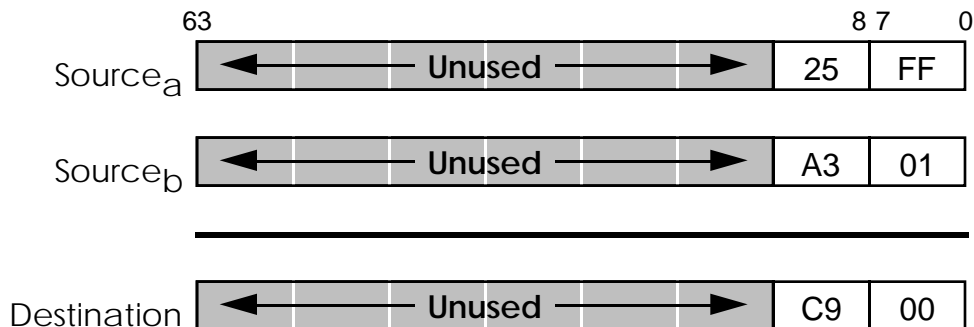


Figure 1. Example of the wasted operand space in current micro-processors. A16-bit integer add operation is shown; the operand values are in hexadecimal.

If the usage of this wasted operand space could be maximized, the processing bandwidth of these processors could increase several-fold when operating on multimedia data types. To accomplish this, processor designers have looked to processing models found in vector processors and massively parallel multiprocessors. These techniques are discussed in the next section.

Another problem undermining the performance of today's processors in media-rich applications is that there is no support at the instruction level for the intrinsic operations which often form the inner loops of a wide range of media-based applications. True to the RISC design of many of these processors, these intrinsics are currently coded in a series of simple instructions on floating point or integer data types.

However, each of these current instructions incurs a latency of at least one cycle. By treating multimedia operations differently from floating point and integer operations the same RISC principles may be applied to derive a minimal set of operations at the instruction level that support multimedia operations. What designers have found is that many of these operations may be implemented with a latency of a single cycle, thereby reducing the amount of work performed in inner loops.

SIMD TECHNIQUES

Microprocessor designers have realized that the underutilized operand space provided by 64-bit processors can be used to achieve a form of parallel processing that is similar to a single-instruction multiple-data (SIMD) multi-processor or a vector processor. In an SIMD multiprocessor there is a

single stream of instruction control with multiple streams of data. In a vector processor, operations are performed on vector registers, where the operation is applied in parallel to every element in the vector. Hewlett-Packard has labeled this *subword parallelism* [8] in it's MAX-2 enhancements. Here we will use the term in a general, non-implementation specific sense.

By looking at Fig. 1, it is easy to see how we could add three additional 16-bit values to each source register, thereby obtaining four results in the destination register. Take again the example of a RGBA quadruplet. Each 8-bit component can be thought of as a subword of a 32- or 64-bit machine word. By implementing subword parallelism, these subwords can be packed into the space of a single word, and subsequent operations may be performed on the entire word. In this manner, utilization and bandwidth increase by an amount proportional to the number of subwords.

In theory, any number of arbitrarily sized subwords could be allowed. However, by keeping the sizes of subwords regular we can limit the number of possible combinations. This serves to keep the hardware as simple and fast as possible. Intel allows 2 32-, 4 16-, or 8 8-bit subwords; Hewlett-Packard allows only 4 16-bit subwords; MIPS allows 4 16- or 8 8-bit subwords¹; and Sun allows 2 32- or 4 16-bit subwords.

DATA PATH CONSIDERATIONS

Modern microprocessors support both floating point and integer operations, each with their own functional units and data path. Often, there exists redundancy in each data path such that multiple instructions can be executing at once. Rather than implementing entirely new sections of logic to implement these multimedia extensions, designers have reasoned that they can be implemented on top of either the integer or floating point data paths. There are various arguments for choosing one data path over the other when deciding on which to implement the multimedia extensions.

Vendor	Extension	Data Path
Hewlett-Packard	MAX-2	Integer
Intel	MMX	Floating point
MIPS Technologies	MIPS V / MDMX	Floating point
Sun Microsystems	VIS	Floating point

Table 1. Data paths chosen by various vendors upon which to implement their multimedia extensions.

Intel has chosen to use the floating point data path of its *Pentium* processors. This was done for two reasons. First, the Pentium is only a 32-bit processor; however, it's floating point unit uses 64-bit registers and has a 64-bit data path. Hence, by using the floating point data path, they can pack more subwords into a machine word. Further, Intel felt that by using the floating point data path,

¹ These are the allowed options for integer subwords. MIPS's MIPS V / MDMX also offers the ability to perform vector arithmetic on two IEEE 754 single-precision floating point numbers.

minimal perturbations would occur in the integer data path thereby ensuring the greatest possible compatibility with existing hardware and software.

MIPS and Sun both implement their multimedia extensions on processors which support a 64-bit architecture and were not encumbered by the restrictions Intel faced. Three additional features of the floating point data path make implementation there attractive [16]. First, it leaves the integer data path and registers free for address and branch computation. Second, many of the operations used in the multimedia extensions are multi-cycle and therefore more suitably map into the control structure of the floating point data path. Finally, altering the integer data path to support multimedia extensions would likely lengthen the critical path length by adding additional gate levels, forcing the designers to increase the processor's cycle time—an ill affordable design and marketing option in today's marketplace.

The importance of the first and third issues is clear. By implementing the multimedia enhancements on the integer side, contention for its functional units would increase greatly because of the frequent address calculations and variable manipulations performed there. Every time the usage of the shared data path and register set changes, the state of the registers must be saved and the integer registers reloaded. This implies a fairly large latency if it must be done frequently. By using the floating point data path and registers, which are utilized significantly less, such latencies can be minimized.

Hewlett-Packard has taken an approach opposite of the other three vendors and has implemented MAX-2 on the integer data path. They did so for several reasons. First, their designers felt that the area savings gained by not replicating integer functional units in the floating point data path outweighed the disadvantage of sharing the integer data path with address and variable calculations [8]. Second, by profiling a variety of multimedia applications, they found that 3D graphics are fairly floating point intensive. Hewlett-Packard then reasoned that it would be an obvious benefit to have both floating point and multimedia instructions executing concurrently.

Instruction Set Overview

In order to decide which new instructions would be most beneficial to the widest range of software, the processor designers profiled a number of applications they felt best represented a typical multimedia work load. Adding support for subword parallelism to their addition and subtraction instructions offered the greatest performance benefit as nearly every application written is able to utilize the power of these instructions to some extent. In addition to these basic arithmetic operations, the vendors have vectorized other instructions such as multiplication, shifts, and logical operations.

Beyond these trivial modifications, vendors have further extended their instruction set architectures to include instructions supportive of primitive operations common to many media-based applications. However, the number and complexity of these instructions vary somewhat between the vendors.

Hewlett-Packard and MIPS have opted to adhere closely to the RISC design principles upon which their processors are based and have included a fairly simple and general set of new instructions. Any of these are applicable to any application performing operations on subword data types. Sun, however, has opted to include a slightly more aggressive set of instructions which include operations for motion estimation, edge boundary processing, and volume rendering.

In general, the instructions implemented in the various multimedia extensions can be divided into four instruction classes: arithmetic, logical, data conversion and reordering, and memory. In the following subsections, we discuss the various classes of instructions and highlight those implementations or instructions that are of particular interest.

ARITHMETIC

All vendors support the addition, subtraction, multiplication, and division of subword operands. Division is only supported by right shift operations, which are equivalent to division by powers of two. Hewlett-Packard only supports subword multiplication through the use of left shifts.

An interesting feature of the addition and subtraction operations offered is that of signed and unsigned saturating arithmetic [6, 8, 14]. In unsigned saturating arithmetic, rather than overflowing bit n into bit $n+1$, the result is clamped to the largest allowable value in the n -bit subword. Similarly, if underflow occurs, the value is clamped to zero in the unsigned case. Saturating arithmetic is useful for 3D rendering algorithms which employ *Gourad* shading or is beneficial in alpha blending operations [12].

Hewlett-Packard also offers an average instruction, commonly used in blending algorithms, which computes the arithmetic mean of two operands and then rounds the results. The rounding is performed to ensure precision over repeated average instructions. Although such an average operation could be easily implemented on other architectures with a vector add followed by a vector right shift, Hewlett-Packard has been able to implement it with a single cycle latency at no increase in overall cycle time.

MIPS's MIPS V and MDMX extensions offer perhaps the most flexible and varied choice of arithmetic operations. In addition to subword addition, subtraction, and multiplication, MIPS V also allows a variety of concurrent operations on two IEEE 754 single-precision floating point numbers in a single instruction. These *paired-single* [8] operations double the potential bandwidth of *all* single-precision floating point computations on MIPS processors supporting the MIPS V ISA. An additional option on subword integer computations is the ability to perform vector/scalar arithmetic. With this option, the first input register contains a vector of subwords. Using the select field of the instruction, a single subword is selected from the second source register and is used as a scalar in the operation on the vector. A similar effect can be accomplished with traditional ISAs through a series of logical operations. In [10] an example is shown in which an 88% reduction in static instruction count is realized by using a vector/scalar multiplication. By using the `permute` instruction in Hewlett-Packard's MAX-2 as discussed below, this effect could be replicated—requiring only an additional instruction over the MIPS approach.

Finally, to prevent the growth of operands from a smaller storage width to a larger computational width during instruction execution, the results of subword integer operations may be optionally saved to a private 192-bit accumulator on the MIPS processor under MDMX. If such an option were exercised, the results of operations on 4 16-bit subwords are promoted to 4 48-bit subwords. The results of 8 8-bit subwords are promoted to 8 24-bit subwords. A special instruction is then required to retrieve the results from the accumulator.

CONVERSION AND REORDERING

In order to move single 8-, 16- or 32-bit data into or out of packed subwords, all vendors support a variety of packing and unpacking operations. In addition, all vendors support some form of subword merge operation. The merge example shown in Fig. 2a begins with the highest order subword of the first source register and merges every other subsequent subword to the right with the corresponding subword from the second source register. By specifying low rather than high the merge would begin with the lowest order subword and progress to every other subword to the left. This operation is useful for interpolation, matrix transpositions, or RGBA quadruplet to plane conversions [14]. Hewlett-Packard adds an additional `permute` operation which takes an arbitrary combination of subwords from a source register and places them in a destination register. An example of the permute operation is shown in Fig. 2b.

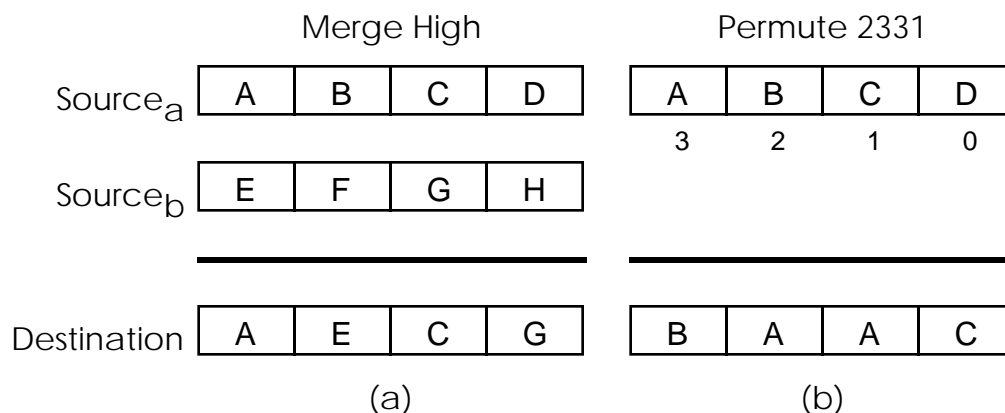


Figure 2. Examples of the `merge high` instruction (a) and Hewlett-Packard's `MAX-2 permute 2331` (b) instruction.

MEMORY

All of the vendors support move or load and store operations intrinsic to their processors. Sun's VIS also offers an instruction which makes possible partial store operations. This instruction stores only those subwords selected through a mask. In addition, a load operation is offered which operates on 8- or 16-bit subword alignments.

An exceptionally powerful operation offered in Sun's VIS is the 64 byte block load or store. This operation transfers 64 bytes between a region of memory and a set of eight consecutive floating point registers [16]. A unique feature of the transfer is that it reads or writes around the cache. Because the transfer avoids the cache, the transfer latency is dictated entirely by the access time of

main memory. This operation is useful in situations in which a series of read-modify-write operations must be performed on a large block of data. In such a case, the transferred data would normally displace useful data in the cache, yet would be never used again. However, with the block transfer no cached data is displaced and the coherency of blocks matching those transferred is transparently maintained.

OTHER

Moving beyond the general instructions offer by Hewlett-Packard, Intel, and MIPS, Sun has added the `edge`, `array`, and `pdist` instructions. With the exception of `edge`, these operations are of little use outside of specific graphics applications. The operation and usefulness of these three instructions are discussed in the following three sections.

Edge. The `edge` operation may be useful in generating masks for the partial store operation, which always begins a store on an 64-bit aligned boundary. Such masks are required in writing to non-aligned memory addresses, a requirement frequently encountered in scan-line-based rendering algorithms [16]. By setting the appropriate mask bits with the `edge` instruction, the partial store instruction is forced to write only to those locations enabled by the mask.

Array. The `array` operation enables efficient caching of and access to 3D data sets such as those used in volume rendering. The operation accomplishes this by converting a set of 3D fixed-point coordinates to a blocked-byte address. The blocked-byte address format attempts to increase the likelihood that a single cache line will contain a data element's nearest neighbors, thereby enhancing the performance of 3D image processing operations.

The 3D fixed-point coordinates are stored in a single 64-bit register. The `array` instruction then takes that register as an input and outputs a memory offset. The offset may then be added to the base address of the data volume, producing the address of the desired 3D data element. However, for `array` instructions, the limitation that the volume be of the dimensions $N \times N \times M$ is imposed, where $N = 2n \times 64$, $M = m \times 32$, $0 \leq n \leq 5$ and $1 \leq m \leq 16$. These size limits are imposed so that subvolumes of an entire volume will fit within an UltraSPARC cache line or memory page.

Pixel distance. The `pdist` instruction implements a pixel distance calculation, such as that frequently used in MPEG video compression. The instruction performs a calculation according to Eq. 1, computing the absolute difference between 8-bit subwords in two registers and accumulating that difference.

$$\begin{aligned} dist_n = dist_{n-1} &+ |a_7 - b_7| + |a_6 - b_6| + |a_5 - b_5| + |a_5 - b_5| \\ &+ |a_4 - b_4| + |a_3 - b_3| + |a_2 - b_2| + |a_1 - b_1| \end{aligned} \quad (1)$$

Software Support

None of the vendors offers any high-level language support for their respective multimedia extensions; support only at the assembly level will be offered. The only exception is MIPS, who will

offer compiler-level support for MIPS V, but not for MDMX. For the programmer who wishes to use the new instructions, he or she must either call the assembly instructions directly, or invoke the instructions through a C preprocessor macro. The latter option is the preferred method, since the macro may contain a series of non-multimedia instructions which emulate the multimedia instruction on processors which do not support the extensions. In addition, the compiler can perform the macro expansion thereby performing the register allocation and code scheduling.

Regardless of whether the programmer chooses to call assembly instructions directly or call them from C macros, he or she must be prepared for a significant amount of additional coding effort. Because no high-level language support is offered, the programmer must assume the role and responsibility of many of the usual compiler tasks. In addition, none of the checks typically performed in a high-level language are available for the multimedia instructions.

Extension Summary

Shown below in Table 2 is a summary of the various features and operations supported by the various vendors' multimedia extensions. The differences between the various implementations are fairly insignificant, although MIPS does offer a wider range of arithmetic operations and Sun offers some specialized graphics operations.

Vendor	MIPS	Intel	Sun	HP
Architecture Extension	MIPS V w/ MDMX	MMX	VIS	MAX-2
Feature				
Operands	3-4	2	3	3
Vector Registers	32 FP	8 FP	32 FP	32 Integer
Integer Storage Sizes				
8 8-bit	x	x	x	
4 16-bit	x	x	x	x
2 32-bit		x		
Integer Computation Sizes				
8 8-bit	8- or 24-bit	x	x	
4 16-bit	16- or 48-bit	x	x	x
2 32-bit		x		
Arithmetic Operations				
Vector/Scalar	x			
Accumulation	x			
Saturation	x	x	x	x
Vector Floating Point	x			
Multiply	x	x	x	
Multiply Add	x			x
Other Operations				
Align/Expand/Pack	A,E,P	E,P	A,E,P	A

Merge/Permute	M	M	M	M,P
Distance			x	

Table 2. Feature summary and comparison of various multimedia-extended instruction set architectures.

What is clear is that regardless of the supported features above, these multimedia extensions provide programmers and users alike with additional performance at little or no cost in terms of cycle time or die area. In Table 3 we show the increase in die area of four different processors due to the addition of support for their respective multimedia extensions. Intel has estimated that the increase in die area due to the addition of MMX technology to all processors across their product line will require the addition of a single new fabrication line to maintain unit production at current volumes.

Vendor	Extension	Processor	Line Width	Die Area Increase
Hewlett-Packard	MAX-2	PA-8000	0.35 μm	0.1%
Intel	MMX	Pentium	0.35 μm	?? %
MIPS	MIPSV / MDMX	N/A ²	N/A	N/A
Sun	VIS	UltraSPARC I	0.35 μm	3.0%

Table 3. Change in die area of various processors due to the addition of support for their respective multimedia extensions.

Performance Analysis

In this section we attempt to both quantitatively and qualitatively assess the potential performance benefits by recoding critical sections of two application's code, taking advantage of the MIPS multimedia-extended instruction set architecture.

TEST SOFTWARE

To quantitatively analyze the benefits of multimedia extensions, we have chosen two different multimedia applications. The first, *animabob*, is a publicly-available interactive volume renderer maintained by the Laboratory of Computational Science and Engineering at the University of Minnesota. The second, *maplay*, is a publicly-available MPEG-I or -II audio stream player written by Tobias Bading at the University of Berlin.

METHODOLOGY

To establish baseline performance numbers for these two applications, both were compiled as specified in Appendix A. Each application was then instrumented for profiling using the *pixie*

² Data is unavailable for MIPS because they are not currently shipping any processors supporting either MIPS V or MDMX.

software available in Silicon Graphics' IRIX 6.2 and were then subsequently profiled with the *prof* software. Note that *animabob* was profiled using the machine `agate.lcse` and *maplay* using the machine `viskan.lcse`. We use the collected *prof* data to facilitate the computation and reporting of the following performance metrics:

- Static instruction count
- Dynamic instruction count
- Floating point operation count
- Integer operation count

For *animabob*, a test volume data set with 16 elements per side was used. The volume data was generated using the `random()` function call available in the C library. The random variates were scaled and multiplied such that the numbers came from a uniform distribution in the range (0, 255). A 0:20.32 second, 44.1 kHz, stereo, MPEG-II audio stream was used as a test data set for *maplay*.

Using these test data sets, Table 2 summarizes the baseline performance metrics obtained by *prof* for each application.

	<i>animabob</i>	<i>maplay</i>
Dynamic Instruction Count	109006104	193661531
Static Instruction Count	30999	17840
Floating point operations	67683	40327120
Integer Operations	38150267	46495628

Table 2. Baseline performance data obtained using *prof* for the two test applications.

As shown in Table 2, *maplay* is far more floating point intensive than *animabob*. As such, we expect that *maplay* will show far greater performance improvements with the MIPS V and MDMX extensions due to the vector floating point operations offered.

Although, the most ideal performance gains might be realized by recoding both applications in their entirety, Amdahl's law as stated in Eq. 2, dictates that the greatest speedup will be realized by optimizing the most frequently executed sections of code.

$$Speedup = \frac{1}{(1 - \%_{enhanced}) + \frac{\%_{enhanced}}{Speedup_{enhanced}}} \quad (2)$$

To ascertain which procedures should be modified, the cumulative procedure call data reported by *prof* were used. In the following two sections, we discuss which procedures were modified and how those modifications were made. Finally, we present the changes in the metrics shown in Table 2 due to the modifications.

RESULTS

animabob

Looking at Table 3, we can see that unfortunately the two most highly utilized routines in *animabob* `__glx_Color4fv()` and `__glx_Vertex3fv()` are within the dynamic shared library `libGL.so`—part of Silicon Graphic's OpenGL for IRIX. As such, the source code for these routines is unavailable and assembly level modifications cannot be made. The routine with the next greatest frequency of usage is `DrawVoxel()`.

Cycles	%	Cum %	Instructions	Calls	Procedure	DSO:File
25559456	26.55	26.55	24760840	798720	<code>__glx_Color4fv</code>	<code>/usr/lib32/libGL.so</code>
22431136	23.30	49.86	23232248	801112	<code>__glx_Vertex3fv</code>	<code>/usr/lib32/libGL.so</code>
9735360	10.11	59.97	11926785	104	<code>DrawVoxel</code>	<code>animabob:draw.c</code>
3204448	3.33	63.30	4005560	801112	<code>glVertex3fv</code>	<code>/usr/lib32/libGLcore.so</code>
3194880	3.32	66.62	3993600	798720	<code>glColor4fv</code>	<code>/usr/lib32/libGLcore.so</code>

Table 3. Top five procedures by number of cycles used in *animabob*.

Because of the relatively small contribution in instructions executed due to `DrawVoxel()`, Amdahl's law dictates that we should expect little performance improvement by optimizing it. Nonetheless, we proceed with the analysis and modifications.

Using *prof*, we can obtain a report of the most heavily used lines of code, correlated to their high-level source statements. Table 4 lists those source lines accounting for at least 0.10% or more of usage.

cycles	%	times	line #	Source code
3115008	3.24%	106496	316	<code>for (i = 0, l = iy; i < nx; ++i, l += 4) {</code> <code> c2[i] = ctab + 4*cd[i];</code> <code> v2[l] = pp;</code> <code>}</code>
898560	0.93%	99840	354	<code>glColor4fv(c1[0]); glVertex3fv(v1);</code> <code>glColor4fv(c2[0]); glVertex3fv(v2);</code>
798720	0.83%	99840	356	<code>glColor4fv(c1[2]); glVertex3fv(v1+ 8);</code> <code>glColor4fv(c2[2]); glVertex3fv(v2+ 8);</code>
798720	0.83%	99840	355	<code>glColor4fv(c1[1]); glVertex3fv(v1+ 4);</code> <code>glColor4fv(c2[1]); glVertex3fv(v2+ 4);</code>
798720	0.83%	99840	357	<code>glColor4fv(c1[3]); glVertex3fv(v1+12);</code> <code>glColor4fv(c2[3]); glVertex3fv(v2+12);</code>
436800	0.45%	99840	358	<code>c1 += 4; v1+= 16; c2 += 4; v2 += 16;</code>
433004	0.45%	6656	110	<code>for (i = j = 0; j < 256; i += 4, ++j) {</code> <code> ctab[i] = cmap[j][0];</code> <code> ctab[i+1] = cmap[j][1];</code> <code> ctab[i+2] = cmap[j][2];</code> <code> ctab[i+3] = cmap[j][3];</code> <code>}</code>
318064	0.33%	26624	232	<code>c2 = cbuf + c2b;</code>
262080	0.27%	99840	353	<code>for (i = nx / 4; i; --i)</code>
226304	0.24%	26624	222	<code>for (j = 0; j < ny; ++j, cd += nx)</code>

225740	0.23%	4992	205	for (j = 1; j < ny1; ++j) { qdraw[j] = face[j-1] face[j]; build[j] = qdraw[j] face[j+1]; }
212992	0.22%	26624	306	switch (build[j])
187080	0.19%	26624	226	i = c1b; c1b = c2b; c2b = i;
156416	0.16%	26624	341	switch (qdraw[j])
99840	0.10%	24960	365	if (nx & 0x1)
99840	0.10%	24960	360	if (nx & 0x2)
99840	0.10%	24960	368	glEnd()

Table 4. Listing of high-level source lines with 0.10% or more of usage from DrawVoxel() in draw.c.

As expected, the calls to the routines from libGL.so account for the greatest amount of usage. The only lines that appear reasonable for optimization are 316, 358, 110, 205, and 226. However, after examining the assembly-level source code, no opportunity for optimizations were found. No optimizations were possible for line 110 as it is only a series of load and store instructions. The optimizations possible in lines 316, 358, 205, and 226 were insignificant due to the lack of data-level parallelism. Hence, we deemed that at the user level, no optimizations were appropriate for DrawVoxel(). We attribute that lack of optimization possible here due the *animabob's* heavy use of library calls.

maplay

As was done with *animabob*, we used *prof* to find those routines which accounted for the largest percentages of instructions executed. A summary of the data for the top five procedures is shown in Table 5.

Cycles	%	Cum %	Instructions	Calls	Procedure	DSO:File
142729377	45.85	45.85	79292182	27972	compute_pcm_sam ples	maplay: synthesis_filter.cc
32458644	10.43	56.27	19496088	755244	put_next_sample	maplay: subband_layer_2.cc
31762206	10.20	66.48	21944034	27972	compute_new_v	maplay: synthesis_filter.cc
27654533	8.88	75.36	18561196	1	main	maplay: maplay.cc
14667015	4.71	80.07	10493991	251748	read_sampledata	maplay: subband_layer_2.cc

Table 5. Top five procedures by number of cycles used in *maplay*.

The potential here for optimization is significantly better than with *animabob*. Collectively, these routines account for just over 80% of all cycles executed. Of that, the compute_pcm_samples() routine accounts for just under 50%. Clearly, the greatest performance benefit would result from optimizing compute_pcm_samples().

The bulk of the code in `compute_pcm_samples()` is comprised of 16 case bodies in a switch statement, all of which share nearly identical code. The code for one of these case bodies is shown in Box 1.

```

vp = actual_v + actual_write_pos;
for (; i < 32; ++i)
{
floatreg = *vp * *dp++;    floatreg += *--vp * *dp++;
floatreg += *--vp * *dp++; floatreg += *--vp * *dp++;
floatreg += *--vp * *dp++; floatreg += *--vp * *dp++;
floatreg += *--vp * *dp++; floatreg += *--vp * *dp++;
floatreg += *--vp * *dp++; floatreg += *--vp * *dp++;
floatreg += *--vp * *dp++; floatreg += *--vp * *dp++;
floatreg += *--vp * *dp++; floatreg += *--vp * *dp++;
floatreg += *--vp * *dp++; floatreg += *--vp * *dp++;

pcm_sample = (int)(floatreg * scalefactor);

if (pcm_sample > 32767)
{
++range_violations;
if (floatreg > max_violation)
max_violation = floatreg;
pcm_sample = 32767;
}
else if (pcm_sample < -32768)
{
++range_violations;
if (-floatreg > max_violation)
max_violation = -floatreg;
pcm_sample = -32768;
}
buffer->append (channel, (int16)pcm_sample);
vp += 15;
}

```

Box 1. Code for a case statement body from the `compute_pcm_samples()` routine.

The main loop body in Box 1 is already unrolled and should be very amenable to vectorization. Using exclusively MIPS V instructions, we were able to reduce loads by just under 50%. The savings were slightly less than 50% due to a realignment instruction that was required. Similarly, the multiplication and addition instructions were nearly halved, due to the doubling of floating point bandwidth by using *paired-single* instructions. A single unpacking instruction was required at the end of the unrolled loop in Box 1, hence preventing the reduction from being an ideal 50%. For the case body in Box 1, the total instruction count was reduced from 108 to 88 instructions, a savings of 18.5%. Looking at *maplay* as a whole, the optimized metrics from Table 2 are shown in Table 6. There were no changes in either the number of floating point operations nor integer operations, although the number of floating point operations per second should increase due to the *paired-single* instructions.

	Baseline	Optimized	% Change
Dynamic Instruction Count	193661531	153753925	20.60%

Static Instruction Count	17840	17520	1.79%
Floating point Operations	40327120	40327120	0.00%
Integer Operations	46495628	46495628	0.00%

Table 6. Comparison of baseline and optimized metrics for *maplay* after MIPS V and MDMX changes were made to the `compute_pcm_samples()` procedure.

Although the change in static instruction count is fairly minimal, the difference in dynamic instruction count changed appreciably. The coding effort, while not insignificant, is justified by these improvements. In [7] the author noted that a tremendous performance increase could be gained by recoding an inverse discrete cosine transform (IDCT) with multimedia instructions. Reviewing the routine `compute_new_v()`, we find that it is essentially an IDCT. Hence, gains greater than 20% could be gained by optimizing this routine as well.

Conclusion

We have shown here four sets of multimedia-extended instruction set architectures. Each employs SIMD processing techniques and includes a variety of new instructions to enhance the performance of media based applications. These extensions have been implemented in various microprocessors at little cost in terms of die area and at no cost in terms of cycle time. Because of the lack of disadvantages in implementing such extensions, those vendors who have not implemented similar technologies in their microprocessors have placed themselves at a competitive disadvantage in the marketplace.

Two different media-based applications were profiled and assessed for their optimizability using the MIPS V and MDMX extensions. The tests on the two different applications verify our assertion that not all multimedia applications will benefit from these extensions. In particular, our results with *animabob* show that offering support for these extensions in system level libraries and drivers offer a large potential for performance increases to a large number of applications in a transparent manner without requiring recoding and recompilation of user programs. The lack of high-level language and compiler support for these extensions further solidifies this argument. At the same time, our results with *maplay* demonstrated that performance improvements of 20% are possible with a modest amount of coding effort. Further, the high usage of floating-point operations in *maplay* suggests that MIPS may have a significant advantage in performance by offering vector floating point operations.

References

1. Dowdell, C. "Scalable Graphics Enhancements for PA-RISC Workstations," *Proceedings of COMPCON*. Volume 37, Number 1. 1992. pages 122-128.
2. Hansen, Craig. "MicroUnity's MediaProcessor Architecture," *IEEE Micro*. Volume 16, Number 4. August 1996. pages 34-41.
3. Hennessy, J.; Patterson, D. *Computer Architecture: A Quantitative Approach*. 2nd Ed. Morgan Kaufmann Publishers, Inc.: San Francisco, CA. 1996. page 30.
4. Kohn, L. "The Visual Instruction Set in UltraSPARC," *Proceedings of COMPCON*. Volume 40, Number 1. 1995. pages 462-469.
5. Lee, Ruby. "64-bit and Multimedia Extensions in the PA-RISC 2.0 Architecture," *Proceedings of COMPCON*. Volume 41, Number 1. 1996. pages 152-160.
6. Lee, Ruby. "Accelerating Multimedia with Enhanced Processors," *IEEE Micro*. Volume 15, Number 2. April 1995. pages 22-32.
7. Lee, Ruby. "Media Processing: A New Design Target," *IEEE Micro*. Volume 16, Number 4. August 1996. pages 6-9.
8. Lee, Ruby. "Subword Parallelism," *IEEE Micro*. Volume 16, Number 4. August 1996. pages 51-59.
9. MIPS Technologies, Inc. *MIPS Digital Media Extension*.
<http://www.mips.com/MDMXspec.ps>.
10. MIPS Technologies, Inc. *MIPS Extensions for Digital Media with 3D*.
http://www.mips.com/ISA_Tech_Brf.ps.
11. MIPS Technologies, Inc. *MIPS V Instruction Set*. <http://www.mips.com/MIPSVspec.ps>.
12. Neider, Jackie. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1*. Addison-Wesley Publishing Company: Reading, MA. 1993.
13. O'Conner, M. "Extending Instructions for Multimedia," *Electronic Engineering Times*. Volume 874. November 1995. page 82.
14. Peleg, Alex. "MMX Technology Extensions to the Intel Architecture," *IEEE Micro*. Volume 16, Number 4. August 1996. pages 42-50.
15. Tremblay, Marc. "UltraSparc I: A Four-Issue Processor Supporting Multimedia," *IEEE Micro*. Volume 16, Number 2. April 1996. pages 42-49.

16. Tremblay, Marc. "VIS Speeds New Media Processing," *IEEE Micro*. Volume 16, Number 4. August 1996, pages 10-20.

Appendix A: Tested Configurations

COMPILERS:

Silicon Graphics IRIX C and C++ compiler, version 6.2

FLAGS:

animabob

bob.o: cc -O2 -n32 -mips3 -r5000 -woff 1164 -I../lib -nostdinc -I/usr/include -DSYSV -DSVR4 -DFUNCPROTO=7 -DNARROWPROTO -DUSEFALLBACK -c bob.c

setup.o: cc -O2 -n32 -mips3 -r5000 -woff 1164 -I../lib -nostdinc -I/usr/include -DSYSV -DSVR4 -DFUNCPROTO=7 -DNARROWPROTO -DUSEFALLBACK -c setup.c

vox.o: cc -O2 -n32 -mips3 -r5000 -woff 1164 -I../lib -nostdinc -I/usr/include -DSYSV -DSVR4 -DFUNCPROTO=7 -DNARROWPROTO -DUSEFALLBACK -c vox.c

draw.o: cc -O2 -n32 -mips3 -r5000 -woff 1164 -I../lib -nostdinc -I/usr/include -DSYSV -DSVR4 -DFUNCPROTO=7 -DNARROWPROTO -DUSEFALLBACK -c draw.c

movie.o: cc -O2 -n32 -mips3 -r5000 -woff 1164 -I../lib -nostdinc -I/usr/include -DSYSV -DSVR4 -DFUNCPROTO=7 -DNARROWPROTO -DUSEFALLBACK -c movie.c

animabob: cc -o animabob -O2 -n32 -mips3 -r5000 -woff 1164 bob.o setup.o vox.o draw.o movie.o ../lib/libgvl.a -lGLw -lGLU -lXm -lPW -lXt -lSM -lICE -lXmu -lGL -lXext -lX11 -lmpc -nostdlib -L/usr/lib32/mips3 -L/usr/lib32 -lm

maplay

maplay.o: CC -c -O2 -n32 -mips3 -r5000 -DIRIX -DIndigo maplay.cc -o maplay.o

ibitstream.o: CC -c -O2 -n32 -mips3 -r5000 -DIRIX -DIndigo ibitstream.cc -o ibitstream.o

header.o: CC -c -O2 -n32 -mips3 -r5000 -DIRIX -DIndigo header.cc -o header.o

scalefactors.o: CC -c -O2 -n32 -mips3 -r5000 -DIRIX -DIndigo scalefactors.cc -o scalefactors.o

subband_layer_1.o: CC -c -O2 -n32 -mips3 -r5000 -DIRIX -DIndigo subband_layer_1.cc -o subband_layer_1.o

subband_layer_2.o: CC -c -O2 -n32 -mips3 -r5000 -DIRIX -DIndigo subband_layer_2.cc -o subband_layer_2.o

synthesis_filter.o: CC -c -O2 -n32 -mips3 -r5000 -DIRIX -DIndigo synthesis_filter.cc -o synthesis_filter.o

obuffer.o: CC -c -O2 -n32 -mips3 -r5000 -DIRIX -DIndigo obuffer.cc -o obuffer.o

crc.o: CC -c -O2 -n32 -mips3 -r5000 -DIRIX -DIndigo crc.cc -o crc.o

ulaw.o: CC -c -O2 -n32 -mips3 -r5000 -DIRIX -DIndigo ulaw.cc -o ulaw.o

maplay:

CC -O2 -n32 -mips3 -r5000 -DIRIX -DIndigo maplay.o ibitstream.o
header.o scalefactors.o subband_layer_1.o subband_layer_2.o
synthesis_filter.o obuffer.o crc.o ulaw.o -o maplay -lm

MACHINE CONFIGURATION³:

Name	agate.lcse.umn.edu
Manufacturer	Silicon Graphics
Product	Power Onyx
Processors	4 194 MHz CPUs: MIPS R10000 Revision: 2.4 FPUs: MIPS R10010 Revision: 0.0
Data cache size	32 KB
Instruction cache size	32 KB
Secondary unified cache size	1 MB
Main memory size	2048 MB, 8-way interleaved
Operating system	IRIX-64
Version	6.2
Initialized state	Multi-user

Name	viskan.lcse.umn.edu
Manufacturer	Silicon Graphics
Product	Indy
Processors	1 150 MHz CPU: MIPS R5000 Revision: 1.0 FPU: MIPS R5000 Revision: 1.0
Data cache size	32 KB
Instruction cache size	32 KB
Secondary unified cache size	N/A
Main memory size	64 MB