Developing High-Performance Multimedia Applications

An Integrative Paper Based On:

- S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets.
- W. Ooi, B. Smith, S. Mukhopadhyay, H. H. Chan, S. Weiss, and M. Chiu. The Dalí Multimedia Software Library.
- P. Ranganathan, S. Adve, and N. P. Jouppi. Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions.

Chris Brooks
April 17, 2001

1 Introduction

As applications such as image processing, audio playback, and video decompression become more popular on commodity desktop systems, multimedia applications are expected to become a dominant computing workload. Multimedia processing involves the manipulation of images, audio, and video and represents a shift from today's most common workloads. Since today's computer systems were designed for text processing and scientific applications, but not for tasks such as audio and video compression and playback, developing high-performance, responsive applications can be difficult. While performance may be an obvious concern, the difficulty involved in developing these applications is also important. The complex data compression and encoding schemes used in many systems make it difficult to develop high-quality systems quickly. Finding good solutions to the performance and software development problems in multimedia systems is critical to the introduction of many highly anticipated technologies. However, far too often developers find themselves sacrificing the complexity of application development for program performance or vice versa. In this paper, we review three techniques for improving performance of multimedia applications and examine their impact on the complexity of software development.

1.1 Overview of Techniques

One common technique for improving multimedia performance is for general-purpose processors to include specialized instructions that are optimized for typical multimedia applications. By taking advantage of the full width of the processor's datapath, the instructions allow multiple small data items to be processed simultaneously by one instruction. These short single-instruction stream, multiple-data stream (SIMD) style instructions operate on *packed data types*, wide operands which are composed of smaller individual elements. Since the most computationally intensive sections of multimedia programs typically include a large number of independent operations on small data types, this type of parallelism can be extremely advantageous. Assuming a 64-bit wide packed unit composed of 8-bit elements, the media extensions have the potential to speedup critical portions of applications by a factor of eight.

Almost all major processor manufacturers have developed their own set of media instructions. Examples include Hewlett-Packard's MAX-1 [15] and MAX-2 [16] extensions to the PA-RISC instruction set, Sun's VIS [11, 25] extensions to the SPARC v9 instruction set, DEC/Compaq's MVI extensions to the Alpha instruction set [4], Motorola's AltiVec [6] extensions to the PowerPC instruction set, and Intel's MMX [19], SSE [2], and SSE2 [9] extensions to the x86 instruction set. The extensions differ in the datapath width, number and type of registers provided, as well as the availability of specific operations. Table 1 summarizes a few of the major media instruction sets, but a more complete survey can be found in [24].

	HP MAX-1	HP MAX-2	Sun VIS	DEC MVI	Motorola AltiVec	Intel MMX	Intel SSE	Intel SSE2[1]
Release Date	1994	1995	1995	1997	1999	1996	1999	2000
Datapath Width	64 bits	64 bits	64 bits	64 bits	128 bits	64 bits	128 bits	128 bits
Number of Registers	31	32	32	32	32	8	8	8
Register Type	Existing Integer	Existing Integer	Existing Float	Existing Integer	New Media	Existing Float	New Media	New Media
Number of Instructions	9	8	121	13	162	57	70	144
Floating Point Support?	No	No	No	No	Yes	No	Yes	Yes

Table 1: Features of Major Media Instruction Sets
(1: The Intel SSE2 registers are the same as those added for SSE, and 68 instructions are wider MMX/SSE instructions.)

Ranganathan, Adve, and Jouppi [22] have shown that media instruction sets provide significant performance improvements for a wide range of multimedia applications. However, taking advantage of the new instructions can be a difficult and time-consuming process. Since there is very little compiler support for the media extensions, developers are limited to using in-line assembly macros or specialized library calls. To take advantage of the new instructions, developers must rewrite critical portions of their applications. Since there are significant variations among instruction sets, this may have to be done many times to target multiple architectures. Although macros and library calls make this slightly more tolerable than assembly language programming, developers still have to be very aware of the underlying instruction sets. Both developer productivity and software portability suffer due to the added complexity.

Ideally, compilers would support the new media features of modern architectures as transparently as they do other processor features. This would allow developers to simply recompile their high-level language source code and have the compiler generate the code to utilize the media instructions effectively. Many commercial compilers support backend code generation for the media extensions, but none are able to analyze the source code for portions that may benefit from them. A research system developed by Larsen and Amarasinghe [13] is working toward this goal. Their compiler analyzes the program for *superword-level parallelism*, short groups of statements performing the same operation that could be grouped and replaced by a single media instruction. They show that their technique is able to utilize the multimedia instructions to obtain a significant performance improvement in multimedia benchmarks.

Given the complexity of many multimedia applications and the pressure to release software quickly, using a compiler and high-level language may not be sufficient for many developers. Like many other application domains, multimedia algorithms and encoding schemes are complicated which can make application development both time consuming and error prone. High-level libraries can be used to hide some of the complexities of the system, but the abstractions found in many libraries may make it difficult to develop efficient programs. While the abstractions help developers manage the complexity of their systems, they may also make it difficult to control the machine's resources and to exploit non-standard optimizations. Ooi, Smith, et al. [20] have developed a high-

performance library called Dalí which attempts to balance the level of abstraction with performance. The library provides "thin" primitives that hide most of the details of the multimedia algorithms and encoding from the developer without sacrificing control over the machine.

1.2 Organization

This paper will examine the trade-offs between performance and development complexity found in each of these three techniques. It will focus primarily on the following issues.

- What makes multimedia applications unique (Section 2)?
- How do the techniques improve performance (Section 3)?
- To what degree is performance improved (Section 4)?
- What impact does each technique have on software development complexity (Section 5)?

2 Characteristics of Multimedia Workloads

Modern computer architectures were designed with non-media applications in mind. Multimedia processing presents a major shift in the characteristics of the workloads running on the processor. Applications such as video conferencing, which can involve encoding and decoding multiple video and audio streams, can require up to tens of billions of operations per second. This is an enormous amount of processing for today's desktop systems, so identifying the characteristics that make multimedia applications unique is important when designing multimedia architectures and software systems.

Few studies present quantitative results characterizing multimedia workloads, but some have noted their key properties [1, 3, 12]. Diefendorff and Dubey [3] identify the following eight characteristics of multimedia workloads that make them unique from non-media workloads:

- *Strict real-time constraints* – The primary determinant of application correctness may include qualitative factors such as the users' perception of its responsiveness. Even in the presence of multiple media streams, the application must guarantee minimum levels of performance to satisfy users. Establishing such performance guarantees is complicated by the variability of the time required for multimedia algorithms to process individual media frames [7].
- *Small, continuous data types* – Since they typically take advantage of users' perceptual systems, the data types used in multimedia applications generally require as little as 8 or 16 bits of precision. This property motivates the use of short vector-style media extensions to utilize the full 32 or 64-bits of precision available in most processor datapaths. Also, most multimedia applications process continuous data that represent the variation of samples over time as opposed to the static data types processed in many traditional applications. Since the data are typically processed once, displayed, and then discarded, media applications may lack much of the temporal locality found in the data references of non-media applications [12].

- *Significant fine-grained data parallelism* – The data processed in media applications are typically a large collection of pixel values, vertices, or audio samples. Since these values are largely independent and often undergo the same set of operations, many applications exhibit single-instruction stream, multiple-data stream (SIMD) parallelism. When compared to the complex features found in most superscalar processors to extract instruction-level parallelism, processors can achieve SIMD execution with relatively simple hardware.
- *Extensive data reorganization* – In addition to the SIMD nature of multimedia processing, most applications also need to be able to reorganize the individual data components efficiently to adjust for various datastream layouts. Therefore, multimedia applications are not well suited for traditional SIMD architectures where data reorganization can be expensive. Reorganization can be accomplished more efficiently in a wide media datapath, but the instruction sets must provide proper support for data rearrangement operations.
- *Significant coarse-grained parallelism* – Since multiple threads may be required to process separate data streams with strict timing constraints, the applications lend themselves well to multiprocessor systems. For example, a video conferencing system may require encoding and decoding of multiple audio and video streams each with rigid real-time constraints. Applications structured in this fashion can be simpler to parallelize across multiple threads of execution than many integer or scientific applications.
- *Dominated by small, tight loops* – The instruction stream in media applications exhibits even greater locality than in non-media applications, so instruction caches can be very effective. Since these loops dominate application performance, optimizing them can provide a significant performance improvement for the entire application. Hand optimization of these critical loops is more feasible than in other application domains.
- *High memory bandwidth* – The applications process large data sets, putting a severe strain on the memory system. Even the largest caches fail to capture the poor locality exhibited by the applications' data references. This may cause unnecessary replacements of secondary data structures that are still accessed frequently by the applications. Cache techniques such as bypassing and prefetching allow the application to avoid these unnecessary replacements as well as long miss latencies when accessing the main data set.
- *High network bandwidth* – Because of the amounts of data the applications process through the network subsystem, new architectural features may be required to process network data in real-time.

Benchmark suites for multimedia have not been standardized like non-media applications, but several have been developed to reflect the unique characteristics of applications in this domain. Developing a set of representative multimedia benchmarks has been difficult because of the diversity of multimedia applications. The UCLA MediaBench [14] is a collection of high-level language multimedia applications including image and video compression and decompression, voice and speech coding, speech recognition, 3-D graphics, and data encryption. The Berkeley Multimedia Workload [23] contains many of the same applications as MediaBench, but evaluates them for larger data sets. Other benchmarks such as the Intel Media Benchmark [10] were designed to illustrate specific media instruction sets and the performance of individual architectures.

Because most multimedia applications are computationally intensive with strict real-time constraints, application performance is one of the most significant challenges faced by multimedia developers. The computer

architectures, programming languages, and compilers used today were designed considering different workloads. To make it easy for developers to create high-performance multimedia applications, the impact that these technologies have on multimedia performance must be reevaluated. In the next section, we explore some emerging techniques that take advantage of the characteristics described above to improve multimedia performance.

3 Techniques for Improving Multimedia Performance

3.1 Multimedia Extensions

The first technique that we consider is extending the instruction set of general-purpose processors with operations specialized for multimedia processing. These media extensions take advantage of the small data types and fine-grained data parallelism found in many multimedia applications. By providing operations on packed data types, the extensions allow multiple small precision operations to be performed in parallel on a larger precision word. Most instruction sets support addition, subtraction, multiplication, and logical operations on packed groups of 8-bits, 16-bits, 32-bits, or 64-bits. While the media instructions can be added with only a minor impact on the processor's architecture, they can significantly enhance multimedia performance. Most modern architectures already have datapaths that are at least 64-bits wide and the arithmetic logic unit can be partitioned easily. However, the media instructions provided on most architectures allow up to eight operations to be performed in parallel which can have a significant impact on performance.

In addition to the performance gains provided by the parallel operations, many instruction sets also include features that are designed to perform common multimedia tasks more efficiently. One common feature is support for *saturating arithmetic*. Saturating arithmetic limits the result of an operation to a maximum or minimum value instead of allowing it to wrap around. This type of arithmetic is used frequently in multimedia operations, such as shading, to limit the range of operation results. Most instruction sets also provide partitioned comparison instructions that generate bit masks to indicate the result in each position. Not only can these bit masks be used as conditions for branch instructions, but they can also be used to select portions of a packed register to be written to memory in a *partial store*. Such partial stores allow multimedia applications to deal with edge conditions by writing only sections of a register to memory. Memory instructions are also provided to access an entire packed word and optionally bypass the cache. Additionally, many instruction sets include special-purpose operations designed to optimize specific, more complex tasks. For instance, Sun's VIS provides an instruction for calculating pixel distances, which is used to estimate motion in video encoding.

Much like vector processing, the main source of the performance improvement provided by the media extensions is a reduction in the program's dynamic instruction count. A majority of the reduction comes from replacing multiple arithmetic or logical instructions with a single packed instruction. The speedup due to performing these operations in parallel depends on the width of the datapath and operands as well as the amount of overhead due to operand packing and unpacking. Since the instruction sets also include memory access instructions for the wider packed data types, blocked loads and stores can replace individual memory references as well as their

address calculations. The total number of loop iterations can be reduced if the grouped operations come from different loop iterations. Therefore, the modified program will require fewer loop overhead instructions. Saturating arithmetic eliminates the need for the hard to predict branches used to check for overflow and underflow. Also, partial stores eliminate the need to test for boundary conditions and perform selective writes. Special purpose operations can replace large blocks of sequential instructions. For example, the pixel distance computation mentioned above for VIS replaces an equivalent sequence of forty-eight individual instructions.

3.2 Compiler Technique

For the media extensions to be fully utilized, the compiler must be able to detect opportunities to execute operations in parallel. The second technique that we review is a research system developed by Larsen and Amarasinghe [13] that performs this analysis. They refer to the type of parallelism used by the media extensions as *superword-level parallelism* (SLP), which is the parallelism available in a program when individual operations are combined into operations on a larger precision "superword." To detect SLP in a program, the compiler must locate independent, isomorphic statements and group them together for parallel execution. Two statements are *isomorphic* if they perform the same operation in the same order. An example set of isomorphic statements is shown on the left side of Figure 1. Once identified, isomorphic statements undergo statement packing, which groups their operands into a single, larger unit. This is shown on the right side of Figure 1, where the operands have been combined into packed units and the operations have been replaced with partitioned SIMD operations. At that point the statements are ready to be replaced with an equivalent media instruction.

The algorithm developed analyzes each basic block separately. The first step is to unroll loops to expose opportunities for parallelism. The unroll factor depends on the width of the target's datapath and the size of the operands used within the loop body. Standard optimizations such as constant propagation, copy propagation, and dead code elimination are applied to the program along with scalar renaming in preparation for parallelization. Starting with adjacent memory references, statements are packed to form groups of independent, isomorphic operations. The packed groups of statements are then checked for dependences, scheduled, and the equivalent media instructions are emitted.

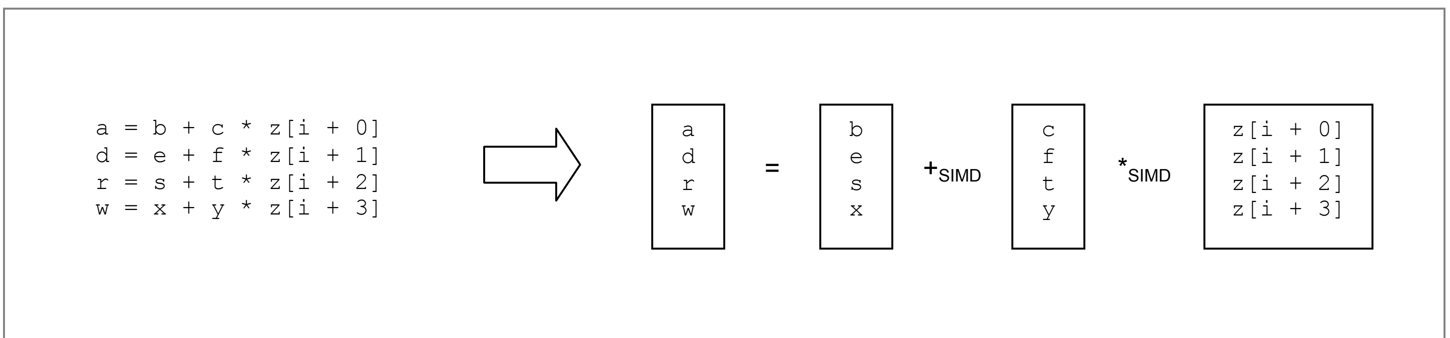

Figure 1: Isomorphic Statements Before and After Statement Packing

Other successful compiler optimizations, such as unrolling loops to expose instruction-level parallelism (ILP), have been adopted quickly in compilers for general-purpose processors. For the SLP analysis to do the same, it must minimize the overhead due to packing, unpacking, and rearranging data. Otherwise the cost of organizing the operands before and after the operation could outweigh any benefit provided by executing the operation in parallel. The algorithm described above accomplishes this by packing statements starting with adjacent memory references. Since they are essentially pre-packed in memory and can be accessed using blocked loads and stores, the overhead due to packing is kept to a minimum. Also, when looking for extensions to the groups of statements it attempts to reuse the results of one packed operation as the source of another. Another complication for the compiler is accounting for the differences between media instruction sets across architectures. Floating-point support, datapath width, and the set of available operations can vary greatly across systems. These factors complicate code generation for the multimedia instruction sets.

3.3 High-Performance Library

While compiler support for new architectural features benefits developers greatly by allowing them to write applications in a high-level language, application development can still be a time consuming and error prone process. One solution is for developers to use a software library to hide the details of the application, such as the compression or encoding scheme. Multimedia libraries such as ooMPEG [21] and the Independent JPEG Group (IJG) system [8] allow developers to focus on their applications and set aside the underlying organization of multimedia data. However, the interfaces that these libraries provide can make non-standard optimizations difficult because too many of the details are hidden from the programmer. The libraries can also make it difficult to use multiple media encodings in the same applications since many are specific to a single multimedia format. The third technique that we review, the Dalí [20] multimedia library, attempts to deal with both of these problems. Dalí's abstractions fall between those found in a high-level language and those provided by a standard multimedia library. This design allows developers to ignore many of the details of the underlying encoding scheme without completely sacrificing application performance. The library is also designed to support multiple data formats so that interoperability is no longer an issue.

Dalí's performance improvements are driven by careful design of the library's abstractions and primitives. While most multimedia libraries attempt to hide the structures of the underlying encoding scheme, Dalí exposes them to the programmer and provides primitives to manipulate them. Access to these structures allows for optimizations based on the data organization that would not be available in most systems. For example, instead of providing one primitive to decode an MPEG frame, Dalí breaks the process down into three stages so that developers may perform application specific optimizations. Also, if special case operations can be implemented in a more efficient manner, they are separated from general-purpose operations. For example, scaling an image by a factor of two or four can be much faster than scaling an image by some other factor. To take advantage of this specialization, the library provides fast implementations for each of these along with a slower implementation for the general case.

One of the key principles of Dalí is giving the developer control of the machine's resources. With better control over the machine, the developer can more easily evaluate and adjust the performance of applications. Instead of having input, output and memory allocation performed implicitly by the library, Dalí forces the programmer to perform these operations explicitly. This makes it easier to analyze the real-time properties of the program, quantify its performance, and evaluate design alternatives. The library's input and output primitives operate on only one abstraction, a BitStream, which is used by other primitives as a data source and target. This allows for non-standard organizations such as multithreaded implementations that may read and process data concurrently. Since the developer explicitly controls the memory utilization, multiple abstractions can often be optimized to share the same memory. This technique can be used when accessing regions of images or individual audio or video frames to avoid excessive data copying. The programmer must also explicitly allocate all memory for the abstractions they are using as well as temporary buffer space. This makes application performance easier to predict and gives the programmer explicit control over the machine's memory resources.

4 Performance Improvements

4.1 Multimedia Extensions

Ranganathan, Adve, and Jouppi [22] used simulation to quantify the impact of multimedia extensions on the performance of a set of multimedia benchmarks. Using the RSIM simulator [18], they modeled in-order single issue, in-order multiple issue (four-way), and out-of-order multiple issue (four-way) processors with and without media extensions. The Sun VIS instruction set from the SPARC v9 architecture was used as the representative media extensions and modeled as found in the UltraSPARC-II. The VIS media extensions are a fairly typical set of SIMD extensions. VIS supports fixed-point operations on 64-bits organized as eight bytes, four words, or two double words.

The authors selected twelve benchmarks to measure the impact of the multimedia instruction sets on the performance of multimedia applications. The benchmarks are similar to those found in the Intel Media Benchmark [10] and the UCLA MediaBench [14]. Six image processing benchmarks were selected from the VIS Software Development Kit (VSDK), including image addition, alpha blending, convolution, dot product computation, scaling, and thresholding. Four image source-coding benchmarks were also selected which perform progressive and non-progressive encoding/decoding of the Joint Photography Experts Group (JPEG) standard, based on an implementation by the Independent JPEG Group. The images used for both the image processing and source-coding kernels were 1024 by 640 with three color channels. The final two benchmarks were an MPEG-2 encoder and decoder implemented by the MPEG Software Simulation Group. The test bit stream was a sequence of three band images that are 352 by 240.

All applications were compiled using the SPARC SC4.2 compiler, optimized for an in-order UltraSPARC. After profiling the applications, the authors manually examined the applications for opportunities to utilize the VIS extensions. Where possible, they replaced entire routines with Sun VSDK Kit or MediaLib library calls that are

optimized for VIS. In the other situations, they unrolled loops and replaced the existing operations with VIS inline assembly macros if doing so would be profitable.

The authors found that the VIS instructions provided a significant performance improvement across all three simulated architectures. On average VIS improved the single-issue architecture's performance by a factor of 2.0x, the multiple-issue by a factor of 2.1x, and the multiple, out-of-order issue by a factor of 1.8x. Also the traditional enhancements to take advantage of instruction level parallelism (ILP) such as multiple and out-of-order issue were still effective when using VIS. The authors found that the performance improvement was due to a decrease in the dynamic instruction count in the benchmarks. Since SIMD operations replaced individual operations, the main reduction in instruction count was from functional unit instructions. As discussed above, there was also a reduction in loop overhead instructions, branch instructions, and memory instructions. Since many of the eliminated branches such as saturation checks and selective writes are difficult to predict, the branch misprediction rates dropped for many of the benchmarks.

4.2 Compiler Technique

Larsen and Amarasinghe [13] implemented their compiler system as a part of the Stanford University Intermediate Format (SUIF) [26] infrastructure. They evaluated the performance of the compiler system on five multimedia benchmarks: a finite impulse response filter, infinite impulse response filter, vector-matrix multiplication, matrix-matrix multiplication, and RGB to YUV conversion. The first four benchmarks use 32-bit floating-point values as the main data type while the final benchmark uses 16-bit integers. To measure the availability of SLP in the benchmarks, they instrumented the basic blocks to count the number of dynamic executions. Using this and the static instruction counts for each basic block, they calculated the dynamic instruction count of the benchmarks on a hypothetical media datapath. They assumed that the datapath can perform any operation on any standard data type and that it had a variety of widths. They found that the dynamic instruction count was reduced by 36-87% on a 128-bit datapath and by 49-98% on a 1024-bit datapath. The greatest reduction came from the RGB to YUV conversion benchmark because it uses the smallest data types and can run almost completely in parallel. These results indicate that the five benchmarks exhibit significant amounts of available SLP that the compiler is able to locate and potentially exploit.

They also measured the performance of the compiled benchmarks on a Motorola MPC7400 with AltiVec extensions. The AltiVec extensions are one of the more powerful media instruction sets, providing support for 128-bit floating point and integer operations. It provides thirty-two 128-bit registers and the corresponding memory access instructions. They configured the compiler to produce C code augmented with AltiVec macros and used a modified version of gcc as a back end. The benchmarks were compiled with and without the use of AltiVec and were run on a 450MHz G4 PowerMac. Most of the benchmarks exhibited speedups of 1.24 to 1.79. However, the RGB to YUV conversion exhibited a speedup of 6.70 due to the reasons discussed above.

The speedups in this section for the AltiVec extensions are not as significant as those discussed in Section 4.1 for the VIS extensions. However, the AltiVec instruction set is much more flexible. Not only is its datapath twice as wide (128-bits), but it provides floating-point support. This discrepancy may be partially explained by

considering that the benchmarks used were different and that one was run in simulation while the other on real hardware. However, based on datapath width alone, one would expect the AltiVec applications to demonstrate a more significant performance improvement. Probably the most significant reason for the inconsistency is that the applications examined in the VIS study were able to take full advantage of the instruction set because they were hand optimized to use the Sun library routines. Meanwhile the AltiVec applications only benefit where the compiler can detect opportunities for parallelism. For example, the VIS applications removed a significant amount of code required to perform saturation checks and selective writes. Another example is the special purpose operations that are provided by some instruction sets, such as VIS's pixel distance computation used in the MPEG encoding benchmark. The hand-optimized applications examined in the VIS study were able to take advantage of these features, while the compiler algorithm has no mechanism to utilize them effectively. The compiler may also impose some additional overhead since it unpacks and packs all values at the basic block level. As the compiler's analysis matures, it may be able to remove some of this overhead and generate slightly better performing code.

4.3 High-Performance Library

Dalí [20] was implemented as a C runtime library with approximately 50,000 lines of code. It is divided into packages based on functionality and data type and currently supports the PNM, GIF, JPEG, WAV, MPEG-1, and AVI formats. The authors measured the performance of the library by implementing three representative benchmarks: image scaling and manipulation, an MPEG decoder, and a JPEG encoder.

The first benchmark program converts a GIF image to a gray scale image and performs scaling. For the measurements, the input was a 1600 by 1200 GIF image and the output was a 320 by 240 gray scale image. The straightforward way to perform this task on a Unix system is to pipe the commands *giftopnm*, *ppmtopgm*, and *pnmscale*. A Sparc 20 system can perform the conversion in 2.6 seconds using these commands. The Dalí program consists of 110 lines of code and takes 1.0 second to perform the conversion. The Dalí version performs better because the three Unix commands have not been optimized to perform the conversion and there is overhead from piping the data between them. A more efficient version could be constructed using the source code from the three commands, but Dalí makes further optimizations simple. For instance, if the color to gray conversion is performed on the GIF color table instead of the image, performance is improved by 13%.

The second program converts an MPEG bitstream into a sequence of PPM images and consists of 150 lines of Dalí code. Its performance was compared to the Berkeley MPEG decoder on a Sparc 20. When tested on a variety of MPEG streams, the Dalí implementation ran about 10% faster. The performance improvement is due to Dalí's specialized primitives for decoding the three types of MPEG frames.

The third program is a JPEG encoder written using Dalí, and was tested with a 1600 by 1200 PPM source image. The Independent JPEG Group's encoder takes about 1.0 second of CPU time on a Pentium II 266 MHz machine with 64 MB of RAM running Windows NT. The straightforward implementation of Dalí takes about 20% longer. After modifying the Dalí implementation to process the image in horizontal strips for better cache performance, its running time is 1.0 second also.

Since Dalí is a software solution to the performance problem, it is quite different from the hardware and compiler based solutions discussed in the previous two sections. The library is limited by the performance of the compiler and the platform that the tests are being run on. The major opportunities for performance improvements are higher-level, non-standard optimizations, but these could be exploited in a hand coded C program. The focus of the library is fundamentally different. Instead of concentrating on improving application performance, the library's developers were mainly concerned with making multimedia development simple while minimizing the impact on application performance. The next section discusses how the developer utilizes each of these performance improvements and how they affect the complexity of writing multimedia applications.

5 Software Development Complexity

5.1 Development Complexity to Utilize Each Technique

In the previous sections, we examined three techniques for improving the performance of multimedia applications. When evaluating these techniques, one must also consider the effort required for programmers to use them during software development. A technique for improving application performance will only be widely accepted if it is not overly burdensome on software developers. The complexity of the development process can influence the development time, error rate, debugging time, final cost, and final quality of the software. In this section we examine the development complexity of each of the techniques and how it can be minimized.

The main drawback of the multimedia instruction sets is that they suffer from a lack of compiler support. This limits developers to using in-line assembly macros and low-level library calls. Macros and library calls do have some advantages over hand-coded assembly. Standard compiler optimizations can still be applied to the code and the new code can be integrated into existing code fairly easily. However, the developer must constantly be aware of how their programs fit into the framework provided by the macros or library calls. Programmers must also be familiar with the details of the multimedia instruction sets. This is complicated by the variations in precision, floating-point support, and available operations across architectures. Requiring programming at the assembly language level can make application development much more complicated. Development time, error rates, and debugging time are all much higher than for applications written in a high-level language. The most devastating impact is on the portability of the applications, which must now be customized individually for each multimedia instruction set.

Except for developers who value performance above all else, most do not typically tolerate having to program at such a low level. This has limited the impact of media instruction sets to specific application domains where developers are willing to sacrifice development complexity for performance. If compilers supported media instruction sets as well as they do other processor features, developers could write their entire programs in a high-level language. While they focus on larger issues, multimedia developers could potentially benefit from the performance improvements provided by multimedia instruction sets. To date there has been very poor compiler support for the media extensions, mostly because compilers have difficulty analyzing operations at the subword

level. The compiler technique described above is a step toward full compiler support for multimedia instruction sets. However, the applications using this compiler still have to be mindful of where the compiler can and cannot utilize the multimedia instruction sets. Only the "analyzable" subset of the C programming language will benefit from the new instructions. Ideally, the compiler would be able to detect opportunities for using the extensions across a much wider subset of the language. The design of the language itself plays a key role in determining where the compiler is able to detect these opportunities. The C programming language is a poor way to express many multimedia operations. For example, there is no direct way to specify saturating arithmetic in the C language. Instead, programmers must express it as a series of conditions for each operation, making it difficult for the compiler to determine the programmer's intent. Also, the data types found in most high-level languages are not adequate for multimedia programs, making it difficult to represent the small values that are often used and the packed words provided by the media extensions.

While high-level languages such as C are a poor way to express multimedia programs, the abstractions and primitives provided by Dalí were designed specifically for that purpose. They hide much of the complexity of the applications, allow development to proceed more rapidly, and simplify application maintenance. By providing "thin" abstractions on top of the high-level language, it allows programmers to ignore the details of multimedia encoding schemes while still providing opportunities for optimizations and detailed control of the application. Despite the advantages of developing applications using Dalí, it is still dependent on the underlying implementation language, compiler, and architecture. To minimize the impact of the tradeoff between application performance and development complexity each of these levels must be designed such that they work together well.

5.2 Composing the Techniques

The three techniques described above approach the problem of multimedia performance from different angles. The multimedia extensions address the performance problem by supplementing general-purpose processors with instructions specifically designed for media processing. Building on these instructions, the compiler technique analyzes a high-level language program for opportunities for parallelism. The multimedia library expands on the gains of the compiler to make multimedia programs simpler to express, which makes application development less complicated. Each of the techniques provides a way to express a program, but they fall at different levels of abstraction at or above the hardware. Therefore, the degree of resource control and development complexity varies. Each level provides a programming interface that is more abstract than the previous, but further removed from direct control of the machine's resources. As one moves from the bottom layers of abstraction to the top, developers are generally forced to give up performance as they simplify development. If this tradeoff could be minimized, programmers could develop high-performance multimedia applications in a simple fashion.

The most promising way for developers to maximize application performance while minimizing development complexity is for the three techniques to be used together. By composing the levels, each could build on the services and abstractions provided by the lower levels to create high-performance applications without sacrificing development complexity. To simplify application development, programmers could write their applications using a high-performance library. The main role of the library should be to encapsulate multimedia

compression and encoding schemes. This library could be written in a high-level language and compiled using a compiler that is able to take advantage of any multimedia features provided by the target architecture. The high-level language must be designed so that the compiler is able to extract sufficient information from the program to utilize the media instruction sets effectively. However, it must also allow programmers to express multimedia algorithms in a natural manner so that the library and applications can be developed easily. The media instruction sets should be designed so that they can efficiently execute a wide range of multimedia workloads and are an easy target for code generation. Careful design of both the high-level language and media instructions will simplify the analysis necessary to detect parallelism in the program and allow compilers to utilize the media instructions more effectively. As a temporary substitute for a fully functioning compiler, high-level libraries like Dalí could be developed using macros to utilize the media instructions, allowing programmers to benefit from them without complicating application development.

5.3 Designing the Abstractions and Primitives

The benefits of composing the techniques into a complete system are significant, but to be most effective, their designs must take the other levels into consideration. We claim that as currently designed, the three techniques would not realize their full potential as an integrated system. In this subsection, we examine how changes to the abstractions and primitives used to represent the program at each level could be used to achieve greater interoperability among them.

The main goals of the media instruction sets should be to allow for efficient execution of multimedia workloads and to provide an interface that promotes automatic code generation. Larsen and Amarasinghe [13] identify five ways that media instruction sets can provide better application performance while simplifying automatic code generation.

- *Reduced operation complexity* – Many media instruction sets provide operations to perform special purpose tasks very efficiently, such as computing pixel distances or matrix transposes. While they may be efficient and easy to use in a hand-coded assembly language program, the complexity of these operations makes code generation difficult. To allow for automatic code generation for the media instruction sets, designers must provide thinner, RISC-like primitives that simplify code generation.
- *Wider set of available operations* – Although there is a significant variation in the types of operations provided, many instruction sets fail to include all of the operations that would be useful in compiling a high-level language program. Lack of support for floating-point operations and full support for multiplication and division places unnecessary limits on the applicability of the instruction sets.
- *Interoperability with other features* – Another problem with media extensions as currently designed is that they are often disjoint from other processor features. For example, the AltiVec instruction set lacks register-register transfer instructions to move data between the media register file and other registers. Forcing register move operations to go through the memory system increases the overhead involved in using the media extensions.
- *Data organization instructions* – Most media instruction sets assume that the data is already packed correctly, and they lack full support for organizing the data to or from other layouts. With better support for data

organization instructions, the compiler would be able to group operands more freely and reorganize them more efficiently.

- *Relaxed alignment constraints* – Since multimedia applications need to access memory on boundaries as small as single bytes, memory alignment constraints can be detrimental to program performance and burdensome on the compiler. The inclusion of efficient unaligned memory accesses would give the compiler more freedom to organize operands effectively.

While the media instruction sets should be designed as a target for code generation, the higher layers must also be designed with developers in mind. A high-level language used for multimedia development should allow developers to unambiguously express their programs while simplifying the compiler's analysis of the source code. Since high-level languages like C are a poor way to express multimedia programs development can become complex and the compiler's analysis of the program can become difficult. Modifying the programming language to express applications more clearly will allow the compiler to extract more information from the program and use the underlying architecture more effectively. Fisher and Dietz [5] have developed a C-like language and compiler that provides an expanded type system and operators to allow developers to more accurately express multimedia programs and to simplify compilation. An expanded set of operators allows developers to perform more traditional SIMD operations while the type system allows them to specify both the overflow semantics and minimum precision of variables. Similarly, Conte et al. [1] identify two areas where the features of languages can provide improved support for multimedia applications:

- *Overflow semantics* – High-level languages implicitly use modulo arithmetic, which forces developers to use a series of conditions to express saturating operations. This makes saturating arithmetic difficult for the compiler to detect and results in poor utilization of the underlying media instructions. If data types were augmented with additional information allowing developers to specify saturating operations, code generation would be simplified and application performance could be improved.
- *Data type sizes* – The compiler must have knowledge of the data type sizes in order to select the appropriate media instructions. Additionally, the programmer must be able to express the size of the data types precisely so that the compiler can exploit as much parallelism as possible. Since data type sizes in languages such as C depend on the underlying machine's architecture, development for multiple platforms can become more complicated. With more precise, machine-independent type systems, compilers can interpret the programs more accurately and use the media instruction sets more effectively.

A general-purpose high-level language cannot support all of the features needed by multimedia programmers to simplify application development. By providing multimedia specific abstractions and primitives, libraries can hide much of the complexity of developing media applications. Libraries can supplement any multimedia support found in programming languages with operations designed for specific media formats. Since these libraries can be updated easily to reflect new encoding and compression techniques, developers can use the latest multimedia technologies in their applications. Although the libraries can provide a significant benefit by

simplifying application development, they can also have a negative effect on application performance. To minimize this impact on application performance the abstractions and primitives must be designed so that programmers do not give up control over the machine's resources. By providing "thin" primitives and exposing the structure of multimedia data to developers, high-performance libraries such as Dalí [20] can simplify programming without sacrificing application performance.

6 Conclusion

Multimedia applications are fundamentally different from most of the programs running on today's desktop systems. Characteristics such as strict real-time constraints, fine-grained data parallelism, and high memory bandwidth distinguish them from traditional scientific and text processing workloads. We have reviewed three emerging techniques for dealing with the unique performance problems presented by multimedia systems. Each provides a significant performance improvement, but to realize this improvement, developers are forced into more complex development practices. This has a negative impact on the final quality, cost and development time of the applications. Composing the techniques into an end-to-end development model provides the best opportunity for developers to minimize the impact that this tradeoff between performance and usability has on their applications and on their productivity. However, as currently designed, the individual techniques will not live up to their potential as a fully integrated system. We have shown how the techniques interact and how the design of each could be improved so that developers do not have to choose between productivity and application performance.

References

[1] 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*, 30(12):33-37, Dec. 1997.

[2] K. Diefendorff. Pentium III = Pentium II + SSE. *Microprocessor Report*, 13(3):1,6-11, Mar. 1999.

[3] K. Diefendorff and P. K. Dubey. How Multimedia Workloads will Change Processor Design. *IEEE Computer*, 30(9):43-45, Sep. 1997.

[4] Digital Equipment Corporation. *Alpha Architecture Handbook*. Accessed at http://www.support.compaq.com/alpha-tools/documentation/current/alpha-archt/alpha-architecture.pdf on April 13, 2001.

[5] R. J. Fisher and H. G. Dietz. Compiling for SIMD Within a Register. In *Languages and Compilers for Parallel Computing: 11th Annual Workshop, LCPC'98*, number 1656 in Lecture Notes in Computer Science, pages 290-304. Springer, 1998.

[6] L. Gwennap. AltiVec Vectorizes PowerPC. *Microprocessor Report*,12(6):1,6-9, May 1998.

[7] C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the Execution of Multimedia Applications and Implications for Architecture. To appear in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 2001.

[8] Independent JPEG Group software. Available at ftp://ftp.uu.net/graphics/jpeg/jpegsrc.v6b.tar.gz.

[9] Intel Corporation. *IA32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Accessed at http://developer.intel.com/design/pentium4/manuals/245470.htm on April 12, 2001.

[10] Intel Corporation. *Pentium Pro Processor Performance Brief*. Accessed at http://developer.intel.com/design/pro/perfbref/242768.htm on April 12, 2001.

[11] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner. The Visual Instruction Set (VIS) in UltraSPARC. In *Proceedings of the 40th Annual IEEE Computer Society International Conference*, pages 462-469, Mar. 1995.

[12] C. E. Kozyrakis and D. A. Patterson. A New Direction for Computer Architecture Research. *IEEE Computer*, 31(11):24-32, Nov. 1998.

[13] S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language and Design*, pages 145-156, May 2000.

[14] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330-335, December 1997.

[15] R. B. Lee. Accelerating Multimedia with Enhanced Microprocessors. *IEEE Micro*, 15(2):22-32, Apr. 1995.

[16] R. B. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51-59, Aug. 1996.

[17] R. B. Lee and M. D. Smith. Media Processing: A New Design Target. *IEEE Micro*, 16(4):6-9, Aug. 1996.

[18] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*, Feb. 1997.

[19] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for Multimedia PCs. *Communications of the ACM*, 40(1):25-38, Jan. 1997.

[20] W. Ooi, B. Smith, S. Mukhopadhyay, H. H. Chan, S. Weiss, and M. Chiu. The Dalí Multimedia Software Library. In *Proceedings of SPIE Multimedia Computing and Networking*, pages 164-275, Jan. 1999.

[21] ooMPEG: Object-oriented MPEG Decoder. Accessed at http://www.cs.brown.edu/software/ooMPEG on April 12, 2001.

[22] P. Ranganathan, S. Adve, and N. P. Jouppi. Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 124-135, May 1999.

[23] N. T. Slingerland and A. J. Smith. Design and Characterization of the Berkeley Multimedia Workload. In Technical Report CSD-00-1122, University of California at Berkeley, Dec. 2000.

[24] N. T. Slingerland and A. J. Smith. Multimedia Instruction Sets for General Purpose Processors: A Survey. In Technical Report CSD-00-1124, University of California at Berkeley, Dec. 2000.

[25] M. Tremblay, M. O'Connor, V. Narayanan, and L. He. VIS Speeds New Media Processing. *IEEE Micro*, 16(4):10-20, Aug. 1996.

[26] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjanyg, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12):31-37, Dec. 1994.