Understanding multimedia application characteristics for designing programmable media processors

Jason Fritts[*], Wayne Wolf, and Bede Liu

Dept. of Electrical Engineering, Princeton University,
Engineering Quadrangle, Princeton, NJ, 08544

ABSTRACT

As part of our research into programmable media processors, we conducted a multimedia workload characterization study. The tight integration of architecture and compiler in any programmable processor requires evaluation of both technology-driven hardware tradeoffs and application-driven architectural tradeoffs. This study explores the latter area, providing an examination of the application-driven architectural issues from a compiler perspective. Using an augmented version of the MediaBench multimedia benchmark suite, compiling and analysis of the applications are performed using the IMPACT compiler. Characteristics including operation frequencies, basic block and branch statistics, data sizes, working set sizes, and scheduling parallelism are examined for purposes of defining the architectural resources necessary for programmable media processors.

Keywords: media processors, processor design, multimedia benchmarks, workload evaluation, MediaBench, MPEG

INTRODUCTION

Digital video and video compression initially generated the extraordinary growth of the multimedia industry, but the industry has since grown to encompass a wide variety of other media, including audio, video, images, computer graphics, speech, and data, or any combination of these. And while compression is still of significant importance, it is now being complemented to a much greater degree by many other forms of media processing. The movement from compression-specific video signal processing to more general media processing will require new means of support for multimedia applications.

Current industry support for multimedia appears in three forms: application-specific processors, multimedia extensions to general-purpose processors, and multimedia co-processors. Application-specific processors offer low-cost alternatives for specific applications, and multimedia extensions to general-purpose processors offer some support for media processing at little additional cost. However, neither method is able to achieve both the flexibility for handling existing and future multimedia applications, as well as the computational capabilities to accommodate the computational intensity and high data rates. Additionally, new applications such as MPEG-4 are arising that have less processing regularity and will be difficult if not impossible to support with application-specific processors[1]. Such applications are also not as amenable to the SIMD parallel processing facilities provided by general-purpose processor multimedia extensions. Instead, media processors should be designed as separate entities, providing high-level language (HLL) programmability and optimizing chip area for the greatest media processing efficiency. These processors should offer the necessary flexibility, be easily programmable, and provide the necessary computing power. While such processors would initially be more costly, the flexibility over a wide range of applications should allow for low cost through mass production.

To design such programmable media processors, it is necessary to understand all characteristics of multimedia applications from both architecture and compiler perspectives. The high degrees of parallelism and large volumes of data within multimedia applications have been well researched[2], but less well understood is how to design a media processor that is both programmable and capable of accommodating these compute and data intensive applications. As with the design of any programmable processor the architecture and compiler designs must be highly complementary in order to achieve the highest efficiency. The designs of the architecture and compiler must be balanced such that the compiler is not constantly saturating

[*] Correspondence: Email: jefritts@ee.princeton.edu; WWW: http://www.ee.princeton.edu/~jefritts; phone: (609) 258-4261

some architecture resources while rarely using others. Achieving this balance requires a design methodology that explores both the technology-driven hardware tradeoffs to determine what architectural features are feasible, and the application-driven architectural tradeoffs to determine which architectural features are desired. The focus of this paper is on the second of these two, examining the application-driven architectural issues from a compiler perspective.

A workload evaluation is performed on a suite of multimedia applications that examines a number of statistics providing insight into various aspects of the media processor architecture. Included among these are operation frequencies, basic block sizes, branch statistics, data sizes, and cache statistics. The operation, basic block, and branch statistics define the functional necessities and branch architecture. Through profiling, the actual integer data sizes (as opposed to the program assigned data sizes) are measured, enabling determination of the size of the datapath. Cache analysis is performed to determine the working set sizes and spatial locality of each application and provide insight into the cache requirements of media processors. And finally, the applications are scheduled onto a simple 8-issue processor for initial determinations of the available parallelism.

This paper is organized as follows. Section 2 describes the multimedia benchmark suite, MediaBench+, and the compiler tools provided by the IMPACT compiler group. Section 3 continues the discussion of workload evaluation and the information the various statistics provide about the architecture. Section 4 provides the results and explains their impact on media processor architectures. Sections 5 and 6 summarize the conclusions and define the direction for our future research.

2. EVALUATION ENVIRONMENT

This multimedia workload evaluation uses the MediaBench benchmark suite recently defined at the University of California at Los Angeles, augmented with H.263 and MPEG-4 applications to make it more representative of future multimedia applications. The study of the characteristics of these applications is performed using the IMPACT compiler tools developed at the University of Illinois at Urbana-Champaign. Use of these compiler tools allows an architecture independent analysis be performed using the compiler's low-level intermediate format (called Lcode), which closely resembles a comprehensive, generic instruction set. Profiling and emulation-based simulation tools are provided by the compiler which enable analysis within the compiler framework.

2.1. MediaBench benchmark suite

Until recently, there were only a few select benchmarks that encompassed small niches of the multimedia industry, such as audio processing, image processing, or small DSP benchmarks. More commonly, small kernels like filters, the discrete-cosine transform (DCT), and motion estimation, would be used in evaluating potential performance. The MediaBench benchmark defined by Lee, Potkonjak, and Mangione-Smith[3,9], is the first combination of multimedia applications to truly represent the entire industry. The benchmark was designed specifically to focus on portable applications written in a high-level language (HLL) that would be representative of the workload of emerging multimedia and communications systems. It incorporates multimedia applications written in C, ranging from image and video processing, to audio and speech processing, and even encryption and computer graphics. Table 1 below gives a detailed description of the MediaBench benchmark suite.

While MediaBench has organized a set of applications that are representative of the multimedia industry as a whole, the benchmark suite is still in its initial stages of development. Some of the applications in the suite, such as ADPCM, G.721, and EPIC, are relatively small programs that are not as representative of more complete applications. To offset this and make the benchmark more representative of emerging and future applications, MediaBench is augmented with the MPEG-4 and H.263 motion video applications. While MediaBench already contains the MPEG-2 application for high-quality video compression, H.263 and MPEG-4 are distinct, with H.263 targeting very low bitrates, and MPEG-4 using the video object model, which entails considerably more video processing and computational complexity. Furthermore, the applications also enable comparison of results with similar trace-driven studies by Wu and Wolf[4,5,6]. Hereafter, the augmented benchmark suite will be referred to as MediaBench+.

The other shortcoming of the MediaBench benchmark suite is the size of the input data sets. Some applications, such as ADPCM, EPIC, JPEG, Mesa, PEGWIT, and RASTA, currently use relatively small input sets, which may skew some workload evaluation results, particularly the cache analysis. While these current studies use the initial input data sets, future studies will likely require modification to the input data sets, or use of the alternative data sets recently provided by Lee, Potkonjak, and Mangione-Smith[9].

ADPCM	A simple adaptive differential pulse code modulation coder (*rawcaudio*) and decoder (*rawdaudio*)
EPIC	An image compression coder (*epic*) and decoder (*unepic*) based on wavelets and including run-length/Huffman entropy coding
G.721	Voice compression coder (*decode*) and decoder (*encode*) based on the G.711, G.721, and G.723 standards
Ghostscript	An interpreter (*gs*) for the PostScript language; performs file I/O but no graphical display
GSM	Full-rate speech transcoding coder (*gsmencode*) and decoder (*gsmdecode*) based on European GSM 06.10 provisional standard
H.263	A very low bitrate video coder (*h263enc*) and decoder (*h263dec*) based on the H.263 standard; provided by Telenor R&D
JPEG	A lossy image compression coder (*cjpeg*) and decoder (*djpeg*) for color and gray-scale images, based on the JPEG standard; performs file I/O but no graphical display
Mesa	A 3-D graphics library clone of OpenGL; includes three demo programs (*mipmap*, *osdemo*, *texgen*); performs file I/O but no graphical display
MPEG-2	A motion video compression coder (*mpeg2enc*) and decoder (*mpeg2dec*) for high-quality video transmission, based on the MPEG-2 standard; performs file I/O but no graphical display
MPEG-4	A motion video compression coder (*mpeg4enc*) and decoder (*mpeg4dec*) for coding video using the video object model; based on the MPEG-4 standard; performs file I/O but no graphical display; provided by the European ACTS project MoMuSys
PEGWIT	A public key encryption and authentication coder (*pegwitenc*) and decoder (*pegwitdec*)
PGP	A public key encryption coder (*pgpenc*) and decoder (*pgpdec*) including support for signatures
RASTA	A speech recognition application (*rasta*) that supports the PLP, RASTA, and Jah-RASTA feature extraction techniques

Table 1 - Description of MediaBench+ benchmark suite

2.2. IMPACT compiler

These media processor research studies use the IMPACT compiler, produced by Wen-mei Hwu's group at the University of Illinois at Urbana-Champaign[7,8,10]. Multimedia applications typically demonstrate large amounts of parallelism which can be exploited on highly parallel processors. However, extracting the parallelism and scheduling the code requires an aggressive ILP (instruction-level parallelism) compiler such as the IMPACT compiler. Years of extensive work and research have gone into designing all aspects of the compiler, particularly the many parallel optimizations. Aggressive ILP compiling requires optimizations such as loop unrolling, procedure inlining, software pipelining, and global scheduling techniques like speculative and predicated execution. The IMPACT compiler provides all these optimizations, but even more importantly, it provides tools, such as profiling and emulation-based simulation, for evaluating their performance and constructing new optimizations, elements critical to researching aggressive processor technologies.

Another important element of the IMPACT compiler is its retargetable nature. The IMPACT compiler was originally designed as a general-purpose compiler and does not support many of the signal processing operations typically seen in DSPs. Because it is anticipated that some of these special operations may also be useful in media processors, the compiler's retargetable back-end can be used to add support for these DSP/media operations as deemed necessary. So, while the workload evaluation currently only examines general-purpose processor operations, additional support may be added later with full advantage of aggressive ILP scheduling available in either case.

For purposes of this workload evaluation study, it is desirable to evaluate the applications independent of any particular instruction set architecture. The IMPACT compiler enables this architecture-independent analysis through their low-level intermediate representation, Lcode. The Lcode representation is essentially a large, generic instruction set of simple operations like those found on most typical RISC architectures, but not biased towards any particular architecture. Of course, one side-effect of this is that some operations or operation formats exist that would likely not be implemented in any reasonable architecture. For example, long immediates are allowed in the immediate field of any operation. This and other such unrealistic operations would need to be broken into multiple operations in a real architecture. However, the effect of these additional operations on code size will typically be minimal.

3. WORKLOAD EVALUATION

To accurately evaluate the intrinsic characteristics of the multimedia applications, the compiler was set to apply only classical optimizations while compiling the benchmark suite. Use of only classical optimizations allows only those optimizations that eliminate redundancies in the code at the assembly level, such as common sub-expression elimination and constant propagation. More aggressive optimizations such as unrolling, procedure inlining, or global scheduling optimizations are specifically disallowed as they can add or remove non-redundant operations, and can also change the size of basic blocks. Such modifications change the characteristics of the workload. Using only classical optimizations and compiling to a generic instruction set architecture provides the most accurate method for measuring multimedia application characteristics.

3.1. Operation frequencies

Defining the correct resource balance in a media processor is of critical importance. Not having enough of the desired resources effects scheduling that lengthens the execution time of applications, while having too many underutilized resources forces extra area, lowers yield, and increases wire length and cycle time. Achieving the proper balance requires consideration of the available parallelism, operation frequencies. While it is currently unknown what degree of parallelism can be obtained from a compiler, knowledge of the operation frequencies can be used to define the appropriate resource ratios.

The IMPACT compiler tools offer a profiling tool which compiles and executes the code and then annotates the execution information back into the code. This profiling may be done at the Lcode level, which is equivalent to the assembly level for a generic instruction set architecture. The profiled Lcode then indicates the execution frequency for each operation in the program. Extracting the dynamic operation frequencies from the profiled code is then easily accomplished. This workload evaluation will extract the operation frequencies for a variety of categories of operations, including integer and floating point, arithmetic versus logic and compares, and even the frequency of calls, returns, and register branches.

3.2. Basic Block and Branch Statistics

Using the same profiling method, it is also possible to extract basic block and branch prediction statistics from the profiled Lcode as well. Information about the basic size provides a good estimation of the initial parallelism obtainable by a compiler. The average size of a basic block defines the maximum amount of local parallelism, i.e. the parallelism available only amongst operations in the same basic block. Of course, achieving the maximum is highly unlikely, as it would effectively require every basic block to issue all its operations simultaneously. Typically, the overall speedup from local parallelism is not greater than 25-35% of that amount. However, the larger the basic block the greater the gain. And as multimedia has demonstrated high degrees of data parallelism[2], it will be interesting to determine whether this translates into large basic blocks.

Often, basic block sizes are small, averaging around five operations, so additional means are necessary for achieving greater parallelism. Extracting more parallelism entails global scheduling of operations beyond the bounds of their original basic block. This requires either speculation or predication. The two methods can have varying degrees of success, but both are dependent upon the ability to accurately predict branches. Speculation is the process where an operation residing on the expected path of control flow executes as soon as its source operands become available, before it is actually known whether it actually needs to execute. Consequently, it is best used across branches that are highly predictable. Predication is the conditional execution of an operation based on the state of a condition associated with the operation. It essentially combines two or more control paths into a single conditional control path, eliminating the dependence of operations on branches. Therefore, it is best used on branches that are more unpredictable. Because both are dependent upon branch prediction, the ability to predict the branches in the program influences the effectiveness of these global scheduling methods. Higher branch prediction accuracy means more effective speculation and predication and greater parallelism. Using the profiling method, the static branch prediction results can be extracted from the profiled Lcode, which will provide a measure of the effectiveness of these global scheduling techniques.

It is also important to evaluate dynamic branch prediction as a potential means of branch prediction as well. Dynamic branch prediction is typically much more accurate than static branch prediction in general-purpose processing application, providing benefits both in decreased branch mispredict penalties as well as improved speculative execution in out-of-order processors. This disparity between static and dynamic branch prediction has warranted the use of dynamic branch predictors in most general-purpose microprocessors. It remains to be seen if such a gap exists for multimedia applications. Simulations of the multimedia benchmarks were performed using a 1024-entry 2-bit counter branch predictor to provide an initial comparison with static branch prediction for media processing.

3.3. Integer Data Sizes

The size of the datapath is another factor that has yet to be determined for media processors. While floating point operations will certainly require 32 bits or more, the integer data sizes for multimedia data are typically believed to require only 8 or 16 bits. If such is true, then it may be possible to design media processors with integer datapaths of smaller widths such as 8 or 16 bits, thereby consuming much less area in the integer computation units and allowing processors with potentially much higher frequencies. Because support of data sizes that do not fit within a smaller datapath can require numerous additional operations and registers, the benefits of a smaller datapath can only be realized if the majority of integer data, including input and output data, intermediates, and temporaries, fit within the smaller datapath.

To determine the effective data sizes for all integer data, the IMPACT simulator monitored every integer operation and kept track of the maximum absolute value produced by each operation. The number of bits required to hold this value defined the data size for all integer results produced by that operation.

While it is obvious that this is not an exact method for computing the largest possible value for a single operation or variable, note that the typical operation executes many thousands to many hundreds of thousands of times. And because the data sizes are scaled according to the execution weight of their operations, the results are expected to be reasonably accurate.

3.4. Cache Statistics

Understanding the memory characteristics of typical multimedia applications is of paramount importance. Not only is it necessary to determine the amount of instruction and data cache necessary for achieving good performance, other characteristics such as spatial and temporal locality are also important factors. Additionally, multimedia applications typically involve streaming data. The memory characteristics of multimedia applications should be examined for evidence that a stream buffer or stride prediction table may provide improved performance instead of, or in conjunction with, the cache memory.

Examination of the memory characteristics involved a cache regression study using the IMPACT simulator. To evaluate the working instruction and data set sizes for each application, instruction and data miss ratios were measured for all base 2 cache sizes between 1 KB and 4 MB, using a line size of 64 bytes. Similarly, spatial locality was evaluated for each application by measuring the instruction and data miss ratios for all base 2 cache line sizes between 8 bytes and 1024 bytes, assuming a 64 KB cache. When measuring the miss ratios, both read and write misses were measured. For a cache that uses a no-write-allocate policy, the write misses would have little effect on the working set size, however, a conservative approach was assumed here to cover both policies. No tests are currently performed for measuring the effectiveness of stream buffers or stride prediction tables, though some initial observations can be drawn about the existence of streaming data from the other results.

3.5. Parallel Performance

As mentioned earlier with regards to basic block size, global scheduling methods are effective at increasing the available parallelism by enabling the compiler to schedule operations outside their original basic block. The two available methods for accomplishing this are through speculative execution and predicated execution. The IMPACT compiler provides two region-based compiling methods for applying these two techniques. The first method creates superblocks[7], larger scheduling regions based on speculative execution, while the second method creates hyperblocks[8], larger scheduling regions based on predicated execution. While a description of these methods is beyond the scope of this paper, the two methods will be applied to examine the additional parallelism they can offer multimedia applications.

The procedure for studying parallelism evaluates five different compilations for each application. The first compilation targets a single-issue processor and allows only classical optimizations for determining the base performance of the applications. This first compilation is the same compilation as used for all the above evaluations. The remaining four compilations all target an 8-issue machine. This processor is a simple machine that provides 8 universal issue slots and uses the same operation latencies as the initial machine (1 cycle integer ALU ops, 2 cycle loads, 3 cycle multiplies and floating-point ops, 10 cycle divides), but only allows one branch per cycle. The first of these four compilations uses only classical optimizations. The second uses classical optimizations as well as procedure inlining. The third compilation uses the superblock method for speculative execution, which also includes loop unrolling. The fourth compilation first applies the hyperblock method for predicated execution, followed immediately by the superblock method to enable some speculative execution and loop unrolling as well.

4. RESULTS

Using the IMPACT compiler and compiling tools, profiling and simulation of the MediaBench+ benchmark suite was performed for purposes of evaluating operation frequencies, basic block sizes, branch prediction rates, data size frequencies, cache statistics, and parallel performance figures. The results and their implications on media processor architecture design are reported here.

4.1. Operation Frequencies

The aggregate results for operation frequencies are reported in Figure 1 and Figure 2. Figure 1 displays the average operation frequencies over all benchmarks in the MediaBench+ benchmark suite, while Figure 2 examines the integer and floating-point load and store operation frequencies for each application. Examining Figure 1 shows the operation frequencies are relatively close to general-purpose processing applications, with the exception that there is little overall floating point usage. Additionally, the percentage of compares is low, but the Lcode instruction set provides compare and branch operations, so that eliminated most of the compares. One other potential surprise is that the multiplier is used less than 2% of the time overall. This is probably because the shifter can perform multiplies (and divides) for powers of 2. This would likely account for the higher usage (10%) of shift operations.

Assuming an integer ALU can perform arithmetic operations, compares, logic operations, and moves, its overall usage frequency would be about 40%. The memory unit must support about 19% of the operations for integer and floating-point loads, as well as supporting 7-8% for stores. The shifter receives 10% of the usage, while the branch unit supports nearly 20% of operations for conditional branches, jumps, calls, and returns. The final 3-4% of operations are used by the multiplier and remaining floating-point operations. From this initial perspective, the ratio of resources might appear as follows:

- (int ALU, load/store, branch, shift, floating-point, int mult) => (8, 5, 4, 2, 1, 1)

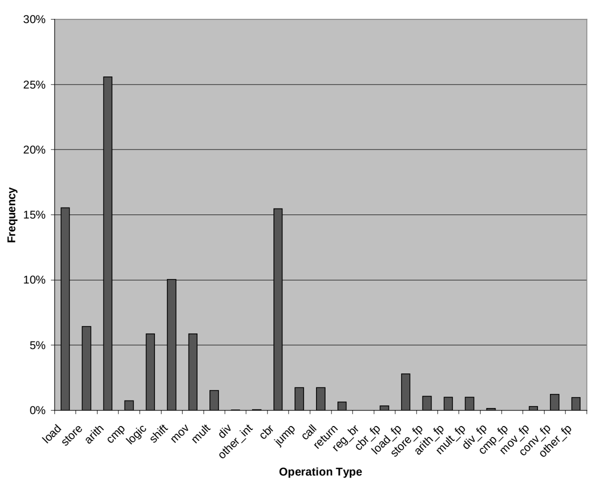

Figure 1 - Overall Operation Frequencies

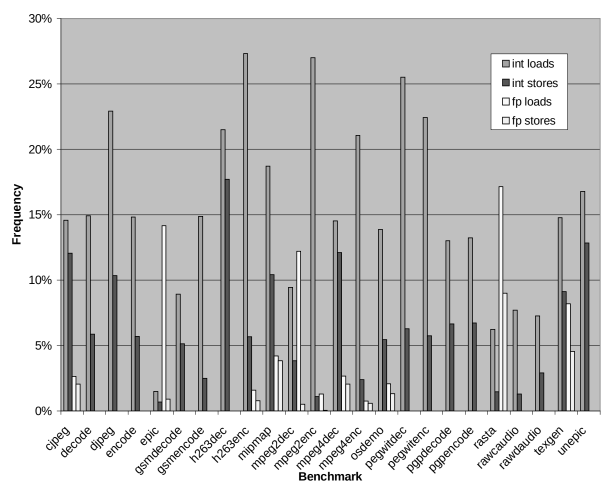

Figure 2 - Load and Store Frequencies

However, from Figure 2, it can be seen that the percentage of loads and stores can vary significantly amongst the different benchmarks. For some applications, particularly video and image processing applications, the frequency of loads and stores can reach 30-35%. Similarly, floating-point operations are heavily utilized in certain applications like *epic*, *mpeg2dec*, *rasta*, and the Mesa applications of *mipmap*, *osdemo*, and *texgen*, where non-load/store floating-point operations can peak 20% usage. With these considerations and the fact that it can be difficult to support more than one branch per cycle, a more reasonable ratio would be as follows:

- (int ALU, load/store, branch, shift, floating-point, int mult) => (3, 2, 1, 1, 1, 1)

One final note is that as more aggressive compiling methods are used, additional operations are introduced through speculation and predication that predominantly increase the usage of the integer ALU unit. So, assuming a more aggressive compiling strategy, the following ratio of resources might prove the most efficient:

- (int ALU, load/store, branch, shift, floating-point, int mult) => (4, 2, 1, 1, 1, 1)

For comparison with separate trace-driven studies performed by Wu and Wolf[4], these resource ratios are similar to their experimental results. For a 32-issue media processor, they found the appropriate ratio of resources for a VLIW VSP to be 24 ALUs, 8 shifters, 16 memory units, and 8 pipelined multipliers (or 16 unpipelined multipliers).

4.2. Basic Block and Branch Statistics

In the section on operation frequencies it was found that overall, approximately 20% of the operations are branch operations. This corresponds with our results for the overall basic block average size of 5.5 operations per basic block. More interesting than this simple average, however, are average basic block sizes for each application as shown in Figure 3. The profile of basic block averages for the various multimedia applications shows enormous variations in the average basic block sizes. The three Mesa applications, the JPEG decoder, and the H.263 encoder all considerably larger basic block sizes than the remainder of the applications, so it is expected that these applications will achieve better parallel performance. It should be noted, however, that some of the applications with larger basic blocks have had critical loops manually unrolled in the C source code by the developers to improve performance. In these cases, the large basic block sizes are not intrinsic properties of the multimedia code.

The overall histogram results for dynamic branch prediction using a 1024-entry 2-bit counter branch predictor are presented in Figure 4. These results were unfortunately lower than expected. Nearly 60% of the branches are predictable 99-100% of the time, but the branch prediction rates drop off quickly after that, for an overall branch prediction average of 90.0%. Fortunately, the static branch prediction fared better, offering nearly the same performance with an average branch prediction accuracy of 89.5%. Figure 5 provides a comparison of dynamic and static branch prediction for the MediaBench+ benchmarks. In all cases except *epic*, static branch prediction performed comparably to dynamic branch prediction. *Epic* is the only benchmark showing a noticeable performance gain from dynamic prediction, which provides an extra 15% accuracy.

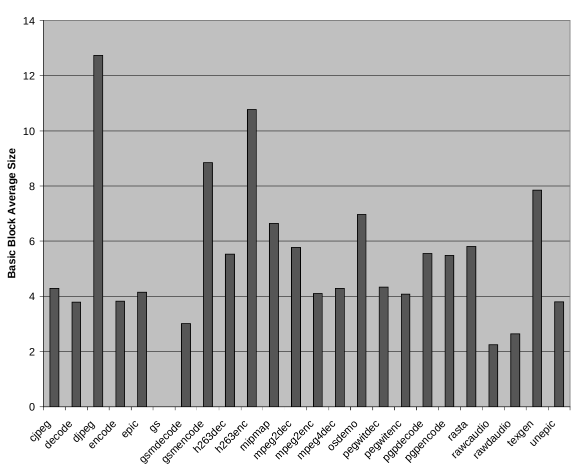

Figure 3 - Average Basic Block Sizes

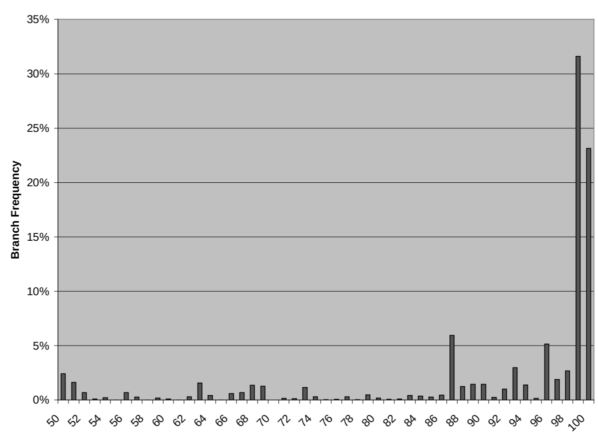

Figure 4 - Overall Branch Prediction Histogram

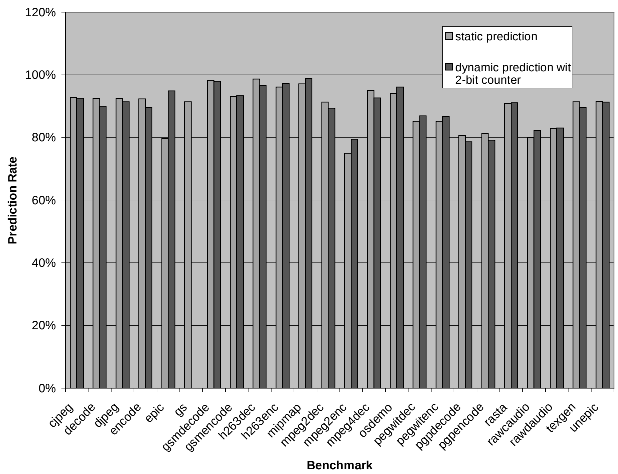

Figure 5 - Dynamic vs. Static Branch Prediction

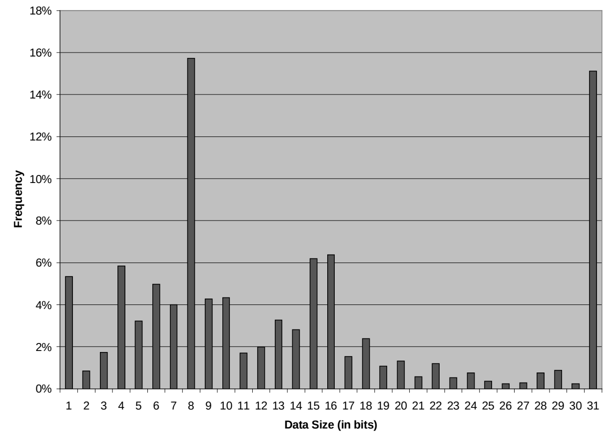

Figure 6 - Overall Data Size Histogram

While these results fare poorly for dynamic branch prediction using the 2-bit counter scheme, the static branch prediction results are better than expected, as compared with typical general-purpose application. The additional static branch prediction efficiency is of considerable benefit as it provides more accurate compile time information about branches. This enables both speculative execution and predicated execution to be applied with greater efficiency, increasing the benefits of global scheduling parallelism.

4.3. Integer Data Sizes

Evaluation of the integer data sizes entailed finding the maximum absolute value produced by each operation and designating the number of bits corresponding to that value as the data size for that operation. The purpose of this evaluation is to determine if the average integer data sizes are sufficiently small enough to warrant using a datapath size smaller than 32 bits. A histogram of the overall results for all the MediaBench+ applications is shown in Figure 6.

While it is evident that there are too many data sizes larger than 8 bits to support an 8-bit datapath, there is a noticeable decrease in the frequency of data sizes larger than 16 bits, with the exception of the 31-bit pointers. The percentage of data values corresponding to pointers is 15%, while the other data sizes larger than 16 bits occur only 9% of the time. The issue then remains as to whether the pointers and other larger data sizes can be sufficiently supported to achieve an overall speedup by using a smaller, and consequently faster, datapath.

An argument can be made for pointers that the lower bits in the pointer change often, but the more significant bits of the pointer change much less frequently. It should therefore be possible to perform computations on pointers typically using only the lower 16 bits. Another possible scheme might provide a single 32-bit unit for pointer manipulation while the remaining units in the datapath are 16 bits. Assuming one of these methods, or potentially another alternative, provides a viable solution for computation on pointers, only the remaining 9% of larger data sizes will require additional operations for computation and use.

4.4. Cache Statistics

Cache regressions were performed to evaluate the variation in cache size and line size for both instruction and data cache. For the cache size regression, cache performance for a direct-mapped cache was simulated for all base 2 sizes between 1 KB and 4 MB, using a line size of 64 bytes. The number of read and write misses were measured and an analysis of the results yielded the working set size for each application. This working set size is identified as the cache size in which the percentage of misses decreased dramatically with respect to smaller cache sizes. This reduction was required to be at least 50%, but in many cases was an order of magnitude. In the absence of a cache size exhibiting such a dramatic decrease, the working set size is defined as the size which reduced the miss ratio to below 3%. The data working set sizes are displayed in Figure 7.

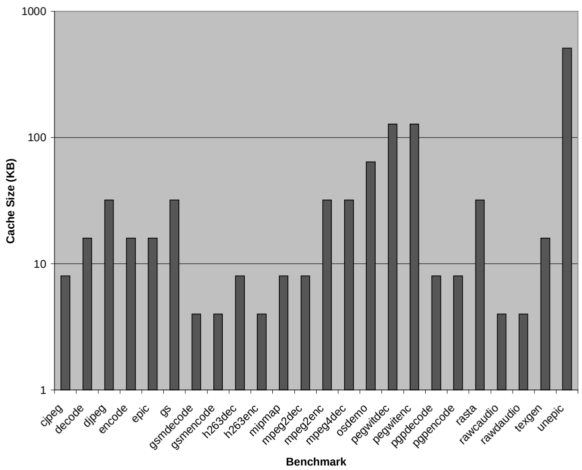

Figure 7 - Data Working Set Sizes

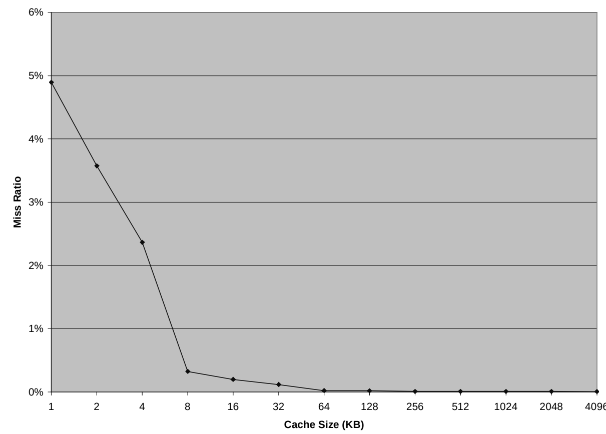

Figure 8 - Overall Instruction Cache Miss Ratios

Based on the statistics, it appears that cache sizes do not need to be very large for most MediaBench+ applications, even in light of the large amounts of data required in multimedia. A data cache size of 32 KB provides an average miss rate of 2.0% on the benchmark suite. This size is sufficient for most of the benchmarks, except for *pegwitenc* and *pegwitdec*, which have miss rates of about 11% at 32 KB. However, its miss ratios remain high unless it has the full 128 KB necessary to contain its working set size. The other applications, *osdemo* and *unepic*, which have working set sizes larger than 32 KB still have respectable miss ratios of 3.9% and 5.5%, respectively, for this cache size. The trace-driven studies of Wu and Wolf[6] yielded similar results, as they concluded a 32 KB 2-way set associative cache with 64 byte line was necessary for good performance on the H.263, MPEG-2, and MPEG-4 motion video applications.

The instruction cache results shown in Figure 8 are even more surprising. A cache size of 8 KB provides the ideal cache size for these applications, with a miss ratio of only 0.3%. Cache sizes smaller than 8 KB decrease miss performance by orders of magnitude, while those larger increase performance only marginally. Even the one application, *gsmencode*, with a larger working set size has a miss rate of only 1.5%. These results are somewhat surprising as many of the applications have relatively large code sizes, with between ten thousand and a hundred thousand and twenty thousand operations. This means that many applications spend the majority of their computation within only a small fraction of the entire code, in some cases less than 3%. This lends considerable credence to the belief that most multimedia applications spend most of their time in a few data processing loops.

Evaluation of the spatial locality performance for instruction and data were based on the cache line size results. As line size increases, performance typically increases because the processor will often use the additional data contained within the cache line without having to generate additional cache misses. The degree to which the processor can use the additional memory within longer line sizes represents the degree of spatial locality for an application. To measure this, a cache line size regression was performed with a 64 KB direct-mapped cache, for all base 2 line sizes between 8 bytes and 1024 bytes. The degree of spatial locality is represented by the average decrease in the miss ratio with respect to the change in line size. The spatial locality for the data cache is shown in Figure 9.

The spatial locality results are quite high with an average spatial locality of 76.1% for data and 86.8% for instructions. Such degrees of spatial locality argue for caches designed with large cache lines. The ideal line size for cache vary with cache size, conclusions cannot be drawn about the ideal line size for multimedia processors. However, for the 64 KB cache used in the cache line size regression, line sizes of 128 or 256 bytes provided the best performance for both data and instruction caches. Wu and Wolf[6] also found longer line size provided the best performance, although they only evaluated line sizes of up to 64 bytes.

While no tests were performed to examine the usefulness of stream buffers or stride prediction tables to help manage the streaming nature of data in multimedia applications, the cache and line size results do provide evidence that such support could be beneficial. Multimedia typically requires very large amounts of data. This is particularly true for video and computer graphics applications, and can hold for data or encryption applications as well. In the operation frequencies results

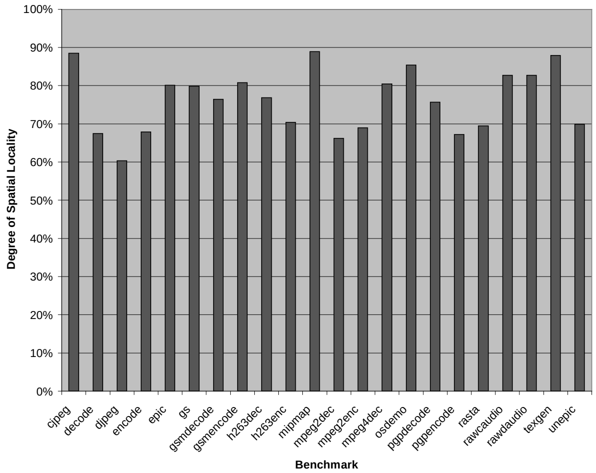

Figure 9 – Data Memory Spatial Locality

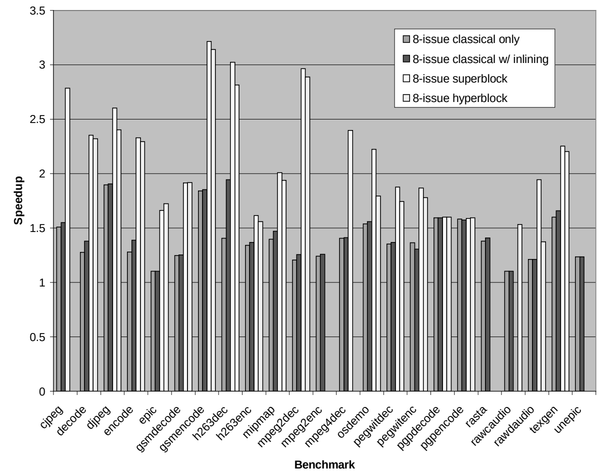

Figure 10 – Parallel Scheduling Performance

a number of applications, including H.263, Mesa, MPEG-2, MPEG-4, and PEGWIT, were found to have very high frequencies for loads and stores. However, with the exception of PEGWIT, none of the data working set sizes for these applications is very large, while the spatial locality results are good in all cases. The large amounts of data coupled with the small working set sizes indicates that the processor typically loads in a small amount of data, processes it, then throws it away. The high frequency of memory accesses and good spatial locality indicate that the many memory accesses are performed to and from the same cache lines, so most of the data is used before it is ejected from the cache. From these two indications, it can be concluded that the processor is constantly loading in small amounts of data, performing all the necessary work on all that data, then throwing the data out, never (or rarely) needing access to it again. This perfectly describes the nature of streaming data. So while, these studies cannot comment on the performance gain from using stream buffers or stride prediction tables, there is substantial evidence to support the existence of a considerable amount of streaming data, so it is likely that performance gains can be obtained from such additional memory support.

4.5. Parallel Performance

Evaluation of the four different compilation methods on an 8-issue processor yields some initial parallel performance results. The four compilation methods involve two compilations with classical only optimizations, the second of which also includes procedure inlining, while the other two compilations provide more aggressive global scheduling optimizations. The first provides speculation through the superblock compiling method, while the second first performs predication with the hyperblock compiling method, following by use of the superblock method. The results for these are found in Figure 10. This study only explores parallel scheduling performance, so an ideal processor model was assumed, excluding any performance penalties from cache and branch effects.

The overall results, while not spectacular, are reasonable and comparable to results found for general-purpose applications. The average results for the four methods are:

- 8-issue classical only - 1.40
- 8-issue classical w/ inlining - 1.44
- 8-issue superblock - 2.22
- 8-issue hyperblock - 2.03

These results are in considerable contrast with the results from Wu and Wolf's trace-driven studies[4,5], which typically found 2-3x better parallel performance. However, these studies used an infinite scheduling window and so represent an upper bound on the potential parallelism.

One positive note is that performance for specific applications can vary considerably from these averages. The video and image applications that are usually more compute intensive, such as JPEG, MPEG-2/4, H.263, and Mesa, typically exceed these averages. One surprising deviation from this trend, however, is *h263enc*, which performs poorly for all scheduling methods. It is particularly surprising in view of the fact that it has the second largest average basic block size. The applications with the largest basic block sizes nearly always exhibited the best parallel performance. *H263enc* must have many sequentially-dependent operations in its basic blocks to defy this general principle. On the whole, however, video and image applications performed better than the other applications from parallel scheduling.

With regards to the specific compilation methods, it is somewhat surprising that the superblock-only compilation provides better performance than the joint hyperblock and superblock compilation method. While the hyperblock has the potential for increased performance, it is still in the development stages within the IMPACT compiler. It is good at providing comparable performance to the superblock, however, without the same degree of code explosion, which can considerably increase instruction cache effects in the absence of a larger instruction cache. For the MediaBench+ benchmark suite, the superblock increased average code size by 95%, while the hyperblock increased it by only 62%.

It is evident from the parallel performance results that the parallelism available in many of these applications, as found from prior research studies[2,4,5], is not easily attainable and additional parallel methods or improvements in these methods will be needed to extract it. This will prove an important area of research necessary for the success of programmable media processors.

5. CONCLUSIONS

This paper presents a workload evaluation of the MediaBench+ multimedia benchmark suite for purposes of investigating the architectural resources necessary for programmable media processors. Using the IMPACT compiler we are able to analyze entire applications and explore the application-driven architectural tradeoffs from a compiler perspective. The workload evaluation examines a variety of characteristics including operation frequencies, basic block sizes, branch prediction rates, data sizes, working set sizes, spatial locality, and scheduling parallelism. From these results, conclusions are made about many aspects of media processor architectures.

The operation frequency statistics define the proper ratio of functional resources as (4, 2, 1, 1, 1, 1) for integer ALUs, load/store units, branch units, shift units, floating-point units, and integer multipliers. Profiling of the integer data sizes indicates that the majority of data sizes are less than 8 and 16 bits. Assuming proper support is available for pointers and longer data sizes, an overall speedup may be obtained using a smaller, faster, 16-bit datapath.

The typically small basic block sizes indicate that the parallelism available in multimedia applications is not available within basic blocks and more aggressive global scheduling optimizations will be needed extract it. While the effectiveness of these scheduling methods depends upon the accuracy of branch prediction, static branch prediction accuracy is relatively high, averaging 89.5%, and bodes well for the obtaining this parallelism. Dynamic branch prediction using a 1024-entry 2-bit counter did not perform as well as expected, performing only marginally better than static branch prediction. More complex dynamic branch prediction methods will be needed if additional branch prediction performance is desired.

Cache analysis concludes that working set sizes for both data and instruction are relatively small, and so cache sizes of 32 KB for data and 8 KB for instruction are sufficient. Spatial locality was excellent, averaging 76.1% for data memory and 86.8% for instruction memory, so longer line sizes should be used in both instruction and data caches. While no tests were performed to evaluate the benefits of stream buffers or stride prediction tables, considerable evidence was found of streaming data, so such additional memory support will likely provide improved memory system performance.

Parallel scheduling performance was only moderate, providing only 2.2 times average speedup even with the most aggressive compiler optimizations. Although, the more compute intensive video applications performed marginally better, the existing methods for parallel scheduling fall well short of the parallelism shown to be available within multimedia applications. Additional parallel methods or improvements to the existing methods will be necessary for extracting this parallelism. Considerable research remains in this area to provide the parallelism necessary for the success of programmable media processors.

6. FUTURE WORK

The evaluation performed in this paper examined characteristics of multimedia applications on a more global scale. Further evaluation shall be conducted at the function level, providing an analysis at the function-level of each application. Such a fine level of characterization conceivably allows definition of the various types of multimedia code segments, such as control code versus data processing code. Even sub-groupings of data processing code may be defined, such as load/stream-input, store/stream-output, or computation code sections. The different characteristics of the various type of code may be found to correspond better with different compiler optimizations. With ability to partition code into different types of code segments and apply the appropriate compiler optimizations to each, it should be possible to increase parallelism within each function, thereby increasing overall performance.

Additionally, a comparison of the performance of static scheduling versus dynamic scheduling will be performed using the MediaBench+ benchmark suite. While the static scheduling model of VLIW architectures has been regarded as the appropriate processor model for media processing, there is no definitive evidence indicating it is the ideal architecture. A comparison of static scheduling and dynamic scheduling will evaluate the benefits of one scheduling model over another, and indicate whether a superscalar or VLIW architecture, or even potentially an amalgam like the EPIC architecture, is fundamentally the better architecture for media processors.

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation under grant number MIP-9408462. The authors would also like to thank all the institutions that provided applications contained within MediaBench+, as well as the members at UCLA that first organized the MediaBench benchmark suite. The authors would particularly like to thank all the members of the IMPACT group at Univ. of Illinois at Urbana-Champaign for the use of their compiler, and especially John Gyllenhaal and Brian Dietrich for their assistance and enduring patience.

REFERENCES

1. V. Michael Bove, Jr., “Multimedia based on object models: Some whys and hows,” IBM Systems Journal, vol. 36, nos. 3 and 4, pp. 337-348, 1996.
2. Heng Liao and Andrew Wolfe, “Avaliable Parallelism in Video Applications,” *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.
3. Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith, “MediaBench: A Tool for Evaluating and Synthesizing Multimedia Communications Systems,” *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.
4. Zhao Wu and Wayne Wolf, “Trace-driven studies of VLIW video signal processors,” *Proceedings of the ACM Annual International Symposium on Parallel Algorithms and Architectures*, ACM, 1998.
5. Zhao Wu and Wayne Wolf, “Parallelism Analysis of Memory System in Single-Chip VLIW Video Signal Processors,” *Proceedings of the SPIE International Symposium on Multimedia Hardware Architectures*, January 1998.
6. Zhao Wu and Wayne Wolf, “Study of Cache Systems in Video Signal Processors,” IEEE Workshop on Signal Processing Systems, IEEE, 1998.
7. W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, “The Superblock: An Effective Technique for VLIW and Superscalar Compilation,” *in Journal of Supercomputing*, Kluwer Academic Publishers, pp. 229-248, 1993.
8. S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, “Effective Compiler Support for Predicated Execution Using the Hyperblock,” *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 45-54, Dec. 1992.
9. MediaBench home: http://www.cs.ucla.edu/~leec/mediabench/
10. IMPACT compiler group: http://www.crhc.uiuc.edu/Impact/