MediaBench II Video: Expediting the next generation of video systems research

Jason E. Fritts[*a], Frederick W. Steiling[a], and Joseph A. Tucek[b]

[a]Dept. of Computer Science and Engineering, Washington Univ., St. Louis, MO 63130;
[b]NCSA, Univ. of Illinois at Urbana-Champaign, Champaign, IL 61820

ABSTRACT

The first step towards the design of video processors and video systems is to achieve an accurate understanding of the major video applications, including not only the fundamentals of the many video compression standards, but also the workload characteristics of those applications. Introduced in 1997, the MediaBench benchmark suite provided the first set of full application-level benchmarks for studying video processing characteristics, and has consequently enabled significant research in computer architecture and compiler research for multimedia systems. To expedite the next generation of systems research, the MediaBench Consortium is developing the MediaBench II benchmark suite, incorporating benchmarks from the latest multimedia technologies, and providing both a single composite benchmark suite as well as separate benchmark suites for each area of multimedia. In the area of video, MediaBench II Video includes both the popular mainstream video compression standards, such as Motion-JPEG, H.263, and MPEG-2, and the more recent next-generation standards, including MPEG-4, Motion-JPEG2000, and H.264. This paper introduces MediaBench II Video and provides a comprehensive workload evaluation of its major processing characteristics.

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

1. INTRODUCTION

The last decade has seen significant growth in the use of motion video for communication, entertainment, and archival purposes. While ten years ago it was impractical and generally infeasible to play/decode video clips on even the most powerful desktop computers, video decoding and encoding now occur on a regular basis in a variety of video systems, ranging from desktop computers, video conferencing systems, and video databases to PDAs, digital video cameras, home theater systems, portable DVD players, and even cellular phones. While the video quality across these systems currently varies considerably, with cheaper and less powerful systems having progressively lower quality video, in the future we can expect that (1) users will continue to demand higher quality video at lower prices, (2) research in information theory will continue to push the envelope in maximizing video compression, and (3) designers of video processors and video systems will continue to face the challenge of providing the greatest video capabilities for the lowest dollar.

The first step towards the design of video processors and video systems is to achieve an accurate understanding of the major video applications, such as the many video compression standards, and the workload characteristics of those applications. Knowledge of video applications is a necessary component in video system design for deciding: (1) whether the system will support one or more video standards, and consequently requires hardware and/or software for separate standards (if the standards aren't sufficiently similar), (2) whether the design should employ more dedicated hardware, hence providing less flexibility, or more software, thereby requiring more powerful and costly processors, and (3) what video processing and system control software requirements must be served by the system processor(s). Depending upon how much of the video processing will be done in dedicated hardware versus software, it is necessary to understand the workload characteristics of the video applications in selecting and/or designing an appropriate video processor. Numerous books and papers are available that thoroughly cover the fundamentals of the many video standards, so this paper focuses on the latter category, providing engineers a solid understanding of video application workload characteristics to enable them to more effectively design video processors and video systems.

[*] fritts@wustl.edu; phone 1 314 935-4963; fax 1 314 935-7302; www.ccrc.wustl.edu/~jefritts/

2. MEDIABENCH: THEN AND NOW

Achieving an understanding of the workload characteristics of an application area, as well as studying new computer architectures and compiler optimizations for that application area, requires a benchmark suite representative of that applpication area. Founded in 1997 by Lee, Potkonjak, and Mangione-Smith, MediaBench[1,4] has served as the dominant system-level benchmark suite for multimedia applications over the last seven years. Lee et al. designed the MediaBench benchmark suite with a focus on full applications representative of the workload of emerging multimedia and communications systems (at that time). It incorporated applications written in C, ranging from image and video processing, to audio and speech compression, and even encryption and computer graphics, and proved an invaluable research tool for computer architecture and compiler design for video systems.

Of course, like SPEC[5] and other benchmark suites, MediaBench has aged over the last seven years and is no longer as representative of the *emerging* workloads for multimedia and communications, but is more representative of the *current and past* workloads. So, in order to maintain and improve upon the mission originally set forth for MediaBench, the MediaBench Consortium was founded to provide for, in a manner similar to the SPEC organization, the continuing development and refinement of the MediaBench benchmark suite[6].

In designing the next generation of MediaBench, one of the goals was to expand the utility of the benchmark suite into distinct application areas. While the original MediaBench included applications from the areas of video, image, audio, speech, computer graphics, and security, many of these applications areas were represented by a single application. And in some cases, that application may not have been as representative of its area as desired. Consequently, the next generation of MediaBench will include both not only a composite benchmark suite, MB_{comp}, which includes the most advanced applications from all video areas, but also separate benchmark suites for each of the distinct media types (e.g. audio, video, speech, computer graphics, etc.). These area-specific media benchmarks suites will include the corresponding media benchmark(s) from the composite benchmark, but will complement the suite with additional representative applications from that area. In particular, the composite benchmark suite will serve as the flagship in defining the set of *emerging* multimedia and communications workloads, while the benchmark for each specific media type will include the emerging applications as well as the current popular applications from that area.

While the definition of the full suite of composite and media-specific benchmark suites is still in the early stages, definition of the next generation of MediaBench has been completed for the audio and video benchmarks. For video applications, the MB_{video} subgroup of MediaBench includes both the popular mainstream video compression standards, such as Motion-JPEG, H.263, and MPEG-2, and the more recent next-generation standards, including MPEG-4, Motion-JPEG2000, and H.264, as shown in Table 1. This suite of application-level video benchmarks, will help take researchers and system designers into the next generation of video systems research and design.

H.263	A video coder (*h263enc*) and decoder (*h263dec*) based on the ITU H.263 standard targeting video compression for transmission over ISDN networks. Source code produced by Telenor R&D.
H.264	A video coder (*h264enc*) and decoder (*h264dec*) based on the forthcoming joint ISO/ITU H.264 standard (also known as MPEG-4 part 10) for very low bitrate video coding. Source code is the test model produced by the H.264 working group.
Motion-JPEG	A video coder (*jpegenc*) and decoder (*jpegdec*) based on the ISO JPEG standard for image compression. Source code produced by the Independent JPEG Group.
Motion-JPEG2000	A video coder (*jp2Kenc*) and decoder (*jp2Kdec*) based on the recent ISO JPEG-2000 standard for wavelet-based image compression. Source code is the JasPer library for JPEG-2000.
MPEG-2	A video coder (*mpeg2enc*) and decoder (*mpeg2dec*) based on the ISO MPEG-2 standard for high-quality video coding. Source code produced by the MPEG Software Simulations Group (MSSG).
MPEG-4	A video coder (*mpeg4enc*) and decoder (*mpeg4dec*) based on the recent ISO MPEG-4 standard for object-based and very-low bitrate video coding. Source code is the ffmpeg library for audio/video coding.

Table 1: Description of the MB_{video} benchmark suite.

3. WORKLOAD EVALUATION

In introducing the MediaBench Video (MB_{video}) benchmark suite, we chose to let it speak for itself. As such, presented below is comprehensive workload evaluation of MB_{video}, including of a discussion of the characteristics that distinguish video from general-purpose applications

Video applications are generally understood to have certain characteristics distinct from typical general-purpose applications. Such characteristics include heavy computational loads, large amounts of streaming data, significant processing regularity, extensive data parallelism, real-time constraints, and a tendency towards small integer data types. Additional research on video processors further contends that video also has considerable control complexity in the less computationally intensive program sections[1,2].

This workload evaluation provides a quantitative understanding of these well-known video qualities as well as other intrinsic characteristics. Among the properties being evaluated are instruction frequencies, basic block and branch statistics, data types and sizes, memory characteristics such as working set size and spatial locality, loop statistics, and instruction level parallelism. From these properties we can determine many of the necessary architecture features for video processors, including the type and ratio of functional units, the branch architecture, the cache memory structure, and the data sizes that need to be supported for subword parallelism. This information can aid video processor and system designers in significantly narrowing the design space.

3.1. Evaluation Methodology

The study of the characteristics of these applications is performed using the IMPACT compilation and simulation environment developed at the University of Illinois at Urbana-Champaign[7,8]. The IMPACT environment includes a trace-driven simulator and an ILP compiler. The simulator provides both statistical and cycle-accurate simulation of a variety of parameterizable architecture models, including VLIW, in-order superscalar, and out-of-order superscalar datapaths. The compiler supports many aggressive compiler optimizations including procedure inlining, loop unrolling, speculation, and predication. With its generic RISC instruction set and highly-parameterizable simulator, the IMPACT environment enables architecture-independent analysis, so is ideal for workload evaluation studies.

To accurately evaluate the intrinsic characteristics of video applications, the IMPACT compiler was set to apply only classical optimizations while compiling the MB_{video} benchmark suite. Using solely classical optimizations, the compiler applies 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 loop unrolling, procedure inlining, or global scheduling are specifically disallowed as they can add or remove non-redundant instructions and can also change the size of basic blocks. Such modifications change the characteristics of the workload. Using only classical optimizations and compiling to Lcode, the IMPACT compiler's generic instruction set architecture provides the most accurate method for measuring inherent video application characteristics.

This section examines the workload characteristics of video applications, including instruction frequencies, basic block and branch statistics, data types and sizes, memory characteristics such as working set size and spatial locality, loop statistics, and instruction level parallelism. In the course of these experiments, we will study the variations in the workload characteristics both between the different standards and for different video input data sets. The base video input data set has a 4CIF video resolution (704x576) with a moderate degree of motion and is compressed to a bitrate providing medium/average video quality using a search window of +/- 16 pixels for motion estimation (where applicable). Additional input data sets have also been designed, which explicitly test different video resolutions (QCIF, CIF, and 16CIF), varying degrees of motion, degrees of compression, and search window sizes. Experimenting with different input data sets as well as different standards provides a thorough understanding of the computing needs across a wide range of video systems.

3.2. Instruction Frequencies

Defining the correct resource balance in a video processor is of critical importance. Not having enough of the necessary resources increases execution time, while having too many underutilized resources entails extra area, power, and cost, lowers yield, and increases wire length and cycle time. Achieving the proper balance requires consideration of the instruction frequencies and instruction level parallelism.

Via profiling, this workload evaluation extracted the instruction frequencies for various classes of instructions, including integer and floating point instructions, arithmetic instructions, logic instructions and compares, and all control flow instructions. Figure 1 displays the average instruction frequencies and their standard deviations[†] across all video applications in the MB_{video} benchmark suite.

Examination of the results in Figure 1 indicates the vast majority of video instructions are either memory (load/store), ALU, or control flow instructions. Nearly 40% of the instructions are load or store instructions, over 35% are ALU instructions (add/sub arithmetic, moves, and logic instructions), 14% are control flow instructions (conditional branches, jumps, and calls/returns), and the remaining instructions are primarily shifts and multiplies.

Overall, instruction frequencies for video workloads bear many similarities to general-purpose applications, but there are a few significant differences. One obvious and expected difference is video's minimal use of floating-point. While general-purpose applications often have significant floating-point usage, video applications have negligible floating-point usage. The more recent video applications do occasionally utilize floating-point for such functions as computing rate versus distortion during rate control, but the requisite floating-point used for such functions is still minimal.

One surprise is that the multiply instruction is used an average of only 2.5% of the time. While this is still 2-3x the frequency of multiplies in general-purpose processors, it was expected that the multiply instruction would be more highly used since DSP processors rely heavily on the MAC (multiply-accumulate) instruction. The use of strength reduction by the compiler accounts for some of this difference, since this compiler optimization converts many of the multiply (and divide) instructions for powers of 2 into shift instructions. This conversion of multiplies and divides into shifts also accounts for the higher frequency of shift instructions (7% for video versus 2% for general applications).

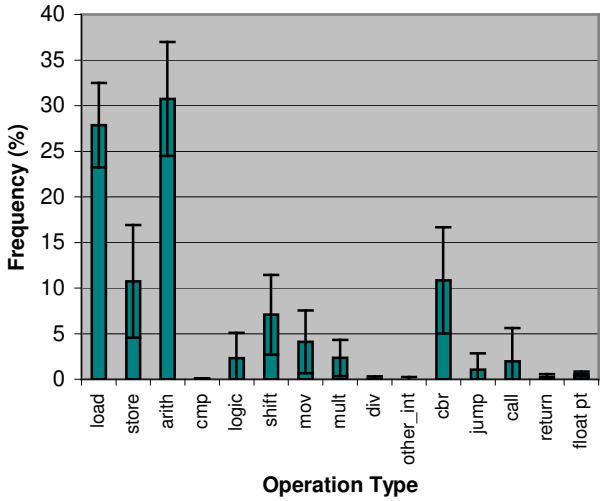

Figure 1. Average instruction frequencies for video applications in MB_{video}.

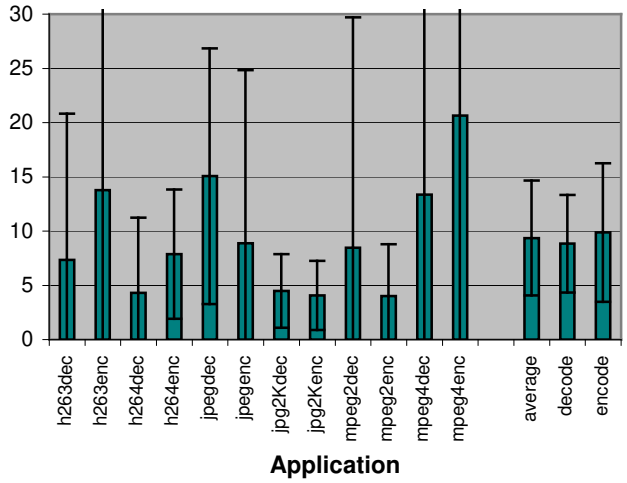

Figure 2. Average basic block sizes for each benchmark.

[†] The standard deviation is represented by the error bars in the figure. Note that standard deviations given for values corresponding with a single application represent the standard deviation of all measured values from the applications computed average value, whereas standard deviations given for values corresponding with a group of applications represent the standard deviation of each application's computed average value from the group's overall average value.

Other less significant differences include the frequency of control flow operations and compares. There is a little more control flow in general-purpose applications, with control flow instructions accounting for 16% of the dynamic instructions on SPECint2000[9] to well over 20% on SPECint92[10], versus video's frequency of less than 14% for control flow. Regarding the frequency of compares, these results indicate that compares account for 0% of the dynamic instruction mix for video applications (versus 5% for general-purpose application). In actuality, IMPACT's instruction set provides 'compare and branch' instructions, which eliminate the need for most compare instructions.

To define the appropriate ratio of functional resources, we assign the instructions to six basic functional unit groups: an integer ALU (IALU), a memory unit (LS), a branch unit (BR), a shifter (SH), a floating-point unit (FPU), and a multiplier (MULT). There could also be a divider, but based on the minimal use of divide instructions, a software implementation is likely sufficient. Given the instruction frequencies from above, one ratio of resources could be:

- (IALU, LS, BR, SH, MULT, FPU) => (5, 5, 2, 1, 1, 1)

However, as more aggressive compiling methods are used, additional instructions are introduced through speculation and predication that tend to increase the usage of the integer ALU unit. Also, considering the fact that it can be difficult to support more than one or two memory instructions per cycle and more than one branch per cycle, a more reasonable ratio under these constraints is:

- (IALU, LS, BR, SH, MULT, FPU) => (3, 2, 1, 1, 1, 1)

Finally, since the frequency of floating-point instructions is so negligible, depending upon the video applications and implementations being used, some processors may eliminate floating-point support altogether.

3.3. Basic Block Sizes

Basic block statistics are similarly extracted using IMPACT's profiling tools. Information about basic block size provides an estimation of the potential instruction level parallelism (ILP) obtainable with a compiler. As mentioned earlier, for these experiments the compiler perform only classical compiler optimizations, since we are measuring the inherent application characteristics. This level of optimization performs only local scheduling, executing in parallel only those instructions from the same basic block. As a result, the average basic block size represents the maximum amount of inherent potential ILP for the application. Of course, achieving this maximum is highly unlikely, as it would effectively require every basic block to execute all of its instructions simultaneously. Since dependencies are common among instructions in the same basic block, the overall speedup from local parallelism is typically not greater than 25-35% of that amount. Regardless, larger basic blocks still provide the potential for greater ILP.

Since research has shown that video has high degrees of parallelism[11], it is to be expected that video has large basic blocks. This workload characterization quantitatively validates that fact. As shown in Figure 2, the results demonstrate an average basic block size of 9.4 instructions per basic block, which is over 50% larger than that than the average in general-purpose applications of 5-6 instructions per basic block[9,10]. Consequently, video shows much greater potential for high ILP than general-purpose applications. It will be interesting to see whether the larger basic block sizes for these applications translates into high ILP when measuring ILP below in section 3.8.

3.4. Branch Prediction

Even with an average of 9.4 instructions per basic block, video applications are still limited by intra-basic-block instruction dependencies and will likely realize ILPs of fewer 2 instructions per cycle (and even less on high frequency processors). Obtaining higher ILPs necessitates global scheduling by the compiler. Global scheduling uses methods such as speculation and predication to search for parallelism beyond the bounds of a single basic block. Speculation is the process whereby an instruction residing on the expected path of control flow executes as soon as its source operands become available, before it is actually known whether it needs to be executed. Consequently, it is best used on critical paths with branches that are highly predictable. Predication is the conditional execution of an instruction based on the state of a condition operand associated with the instruction. It essentially combines two or more control paths into a

single conditional control path, eliminating control dependencies, i.e. the dependence of instructions on branches (though it does create additional data dependencies). Therefore, it is best used across basic block boundaries that have more unpredictable branches. Because both speculation and predication rely on static branch prediction, the compiler's ability to predict branches influences the effectiveness of these methods. Higher branch prediction accuracy means more effective speculation and predication, resulting in greater ILP.

Figure 3 compares the branch prediction miss rates for static, profile-based, branch prediction and a simple dynamic branch predictor. The results indicate that video applications display exceptional static branch prediction performance, with an average branch prediction miss rate of 8.2%, or 6.5% when excluding the MPEG-2 encoder which has abnormally high miss rates. In fact, video's static branch prediction is comparable to the performance of the 1024-entry 2-bit counter dynamic predictor, which has an average miss rate of 7.5%. Such low miss rates from static branch prediction indicate regular, predictable control flow in video, which can be effectively supported without requiring dynamic branch prediction (except in the most aggressive ILP architectures that employ dynamic out-of-order scheduling). With the exception of MPEG-2 encoding, the average miss rate of 6.5% for video is much better than the miss rates for most general-purpose applications. Video's static branch prediction miss rate is nearly two and a half times better than SPECint92's miss rate of 15% and about 40% better than SPECfp92's miss rate of 9%[10].

Video's high static branch prediction rates are indicative of the processing regularity that exists in video applications. While there are certainly many dynamic branches that are difficult to predict, these branches occur infrequently, while the predictable branches dominate video's execution time. The resulting static branch prediction efficiency is of considerable benefit as it provides more accurate compile-time information about control flow. This enables speculative execution and predicated execution to be applied with greater efficiency, increasing the benefits of global scheduling, which enables higher ILP than generally achieved for general-purpose applications.

3.5. Data Types and Sizes

Data type and size is another important issue in video. As opposed to conventional microprocessor applications, video applications typically use small integer data types of 16 bits or less. The data characteristics used by video applications are important because subword parallelism is most effective for small data types, allowing the application to achieve high levels of SIMD parallelism.

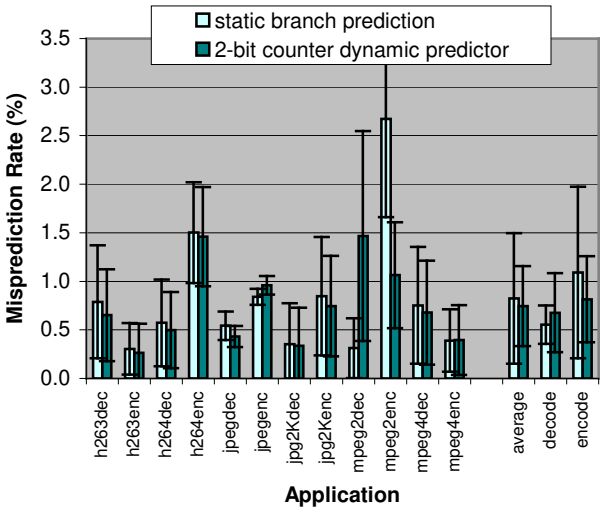

Figure 3. Misprediction Rate of static branch prediction vs. a 1024-entry 2-bit counter dynamic branch predictor.

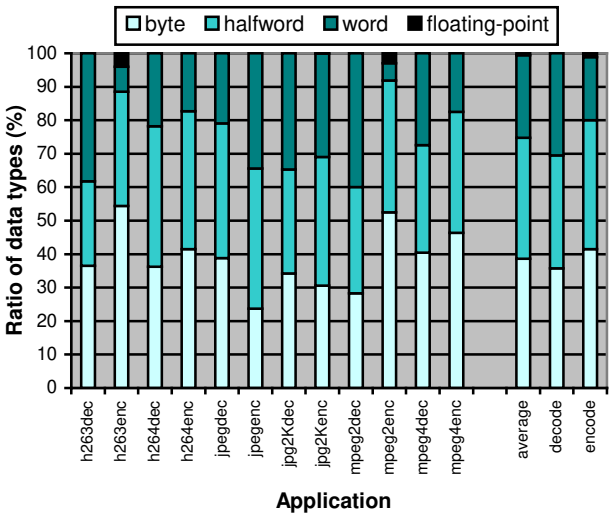

Figure 4. Ratio of data types according to video type.

To determine the effective data sizes for all integer data, the profiling tool for the IMPACT compiler was modified to dump the value of each integer instruction into the execution trace. The high-level simulator was then able to monitor the actual value for each integer instruction and keep track of its maximum absolute value (i.e. determine the maximum number of bits for specifying magnitude) and whether it takes on negative values (i.e. determine whether a sign bit is needed). The number of bits required to hold this value defines the effective data size required for that instruction. While this is not an exact method for computing the maximum value for an instruction or variable, the results are scaled according to the execution weight of the instructions. Instructions executed more frequently are expected to be more accurate and will contribute more to the results, while those instructions executed less frequently will be more prone to error, but will impact the final results minimally. Figure 4 shows the average ratio of data types in MB_{video}.

From the results it is apparent that there is indeed a tendency toward small integer data types. Overall, nearly 40% of the instructions in the benchmark suite require only byte integer data types, and another 35% require halfword (16-bit) data types. The remaining 25% of the instructions are primarily devoted to addressing, so in a 32-bit CPUs these instructions would all use word (32-bit) data types, whereas in 64-bit CPUs they would be split between word and double-word (64-bit) data types, with double-words receiving the majority. So we can conclude that video applications, as characterized by the MB_{video} benchmark, use 16-bit or smaller data types 75% of the time on average, offering ample opportunity for subword parallelism.

In comparison with general-purpose applications, the difference in the frequencies of data types and sizes is significant. As reported by Hennessey and Patterson[10], the percentage of byte, halfword, word, and double-word data sizes is approximately 10%, 5%, 85%, and 0% for the SPECint2000 general-purpose applications. While these results were not obtained in the same manner as the video results, there is still a stark contrast between the two application areas.

3.6. Memory Statistics

Understanding the memory characteristics of typical video applications is of paramount importance. Not only is it necessary to determine the amount of data memory necessary for achieving good performance, other characteristics such as spatial and temporal locality are also important factors. Additionally, video applications typically involve streaming data. The memory characteristics of video applications should be examined for evidence that memory prefetching structures, such as stream buffers or stride prediction tables, may provide improved performance.

Examination of the memory characteristics involved a cache regression study using the IMPACT simulator. For each application, the data working set size, which is defined as the amount of cache memory needed to achieve good memory performance, was determined by measuring the data cache miss ratios for all base-2 cache sizes between 1 KB and 256 KB, using a line size of 32 bytes. Spatial locality was similarly measured from the data cache miss ratios for all base-2 cache line sizes between 8 and 128 bytes. When measuring the miss ratios, both read and write misses were measured.

3.6.1. Working Set Size

To evaluate working set size, a cache regression was performed for all base-2 sizes between 1 KB and 256 KB, using a line size of 32 bytes and a write-back/write-allocate cache write policy. To also understand the impact of cache associativity on working set size, we tested cache associativites from 1-way set associative (direct-mapped) to 8-way set associative. The number of read and write misses were measured and an analysis of the results yielded the working set size for each application. With respect to this study, the working set size is defined as the cache size that reduces the data cache miss ratio to 2.5% or less.

The data working set sizes for the MB_{video} benchmarks are displayed in Figure 5. Based on the results, it appears that cache sizes do not need to be very large for typical video applications, particularly when using 2-way, 4-way, or 8-way set associative caches. For caches with associativity of 2-way, 4-way, or 8-way, an 8 KB cache is sufficient for most of the applications, providing an average miss rate of 2.5% across all the benchmarks. JPEG-2000 and the MPEG-2 decoder are the only two applications that have poor miss rates with an 8 KB cache, requiring 64 KB and 32 KB, respectively, before they achieve reasonable miss rates. For direct-mapped caches, a somewhat larger data cache of 32 KB is needed for good memory performance. A 32 KB direct-mapped cache provides an average miss rate of 2.0%

across all the benchmarks, and is sufficient for most of the benchmarks except JPEG-2000, whose miss rates remain high until it has the full 128 or 256 KB necessary to contain its working set. Overall, the cache sizes required for both direct-mapped and 2-way, 4-way, or 8-way associative are fairly small. Even though video applications involve large amounts of data, the amount of computation performed per pixel is sufficiently large that only a small amount of data needs to be maintained in the data cache in order to achieve good memory performance.

The data working set sizes shown in Figure 5 were only computed for one input data set, but we expect data working set sizes will remain less than 32 KB for most video inputs. While variations in input data set size or application operating parameters are known to affect data working set size, as will be seen in section 4, the results of most common variations have small to moderate effect on data working set size. Consequently, we expect cache sizes of 8KB for 2-way, 4-way, and 8-way associative caches and 32 KB for direct-mapped caches will prove sufficient for most video processors.

In comparison with the video results, general-purpose applications typically exhibit poorer memory performance. According to Hennessey and Patterson[9], the average performance for the SPEC CPU2000 general-purpose applications on a 32 KB direct-mapped data cache is 4.2%, which is more than 2 times worse than the average video miss rate. To achieve the same average miss rate as video, the SPEC general-purpose applications would require a 128 KB data cache. Similarly, for other cache sizes and associativities, general-purpose applications, as characterized by the SPEC CPU2000 benchmark, require cache sizes at least 2 times, and often an order of magnitude, larger than video applications to achieve comparable miss rates.

3.6.2. Spatial Locality

A memory characteristic common to many computer applications is spatial locality, which is the memory property where access to a given memory location is likely followed by subsequent accesses to nearby memory locations in the near future. To evaluate the spatial locality of data memory in video, a cache line regression was performed that evaluated the memory performance for all base-2 line sizes between 8 and 128 bytes in a direct-mapped cache with a write-back/write-allocate cache write policy. To examine the impact of cache size on spatial locality as well, we tested cache sizes of both 16 KB and 32 KB. As line size increases, performance may increase because the processor will often use the additional data contained within the cache line without having to generate additional cache misses. However, increasing the line size while maintaining the same cache size effectively decreases the number of sets, so increasing line size can also decrease performance. The degree to which the processor benefits from the additional memory within longer line sizes represents the degree of spatial locality for an application.

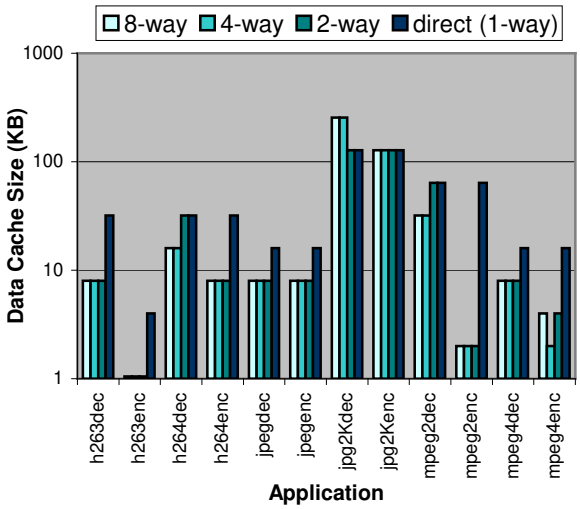

Figure 5. Data memory working set sizes for different cache associativities.

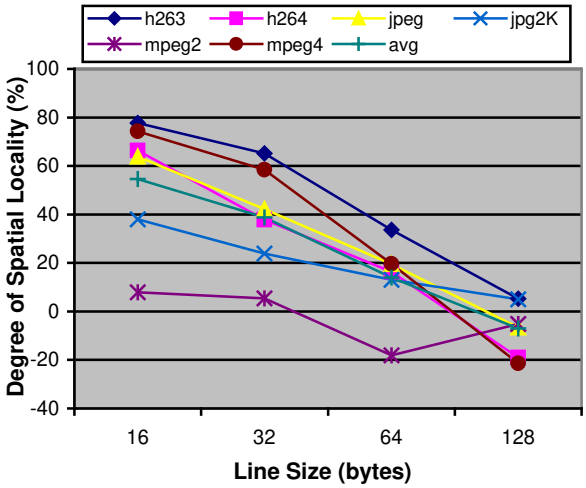

Figure 6. Data memory spatial locality for 32 KB direct-mapped cache. Indicates spatial locality between indicated line size and previous line of half the size.

An equation was defined to quantitatively compute the relative spatial locality from using longer line sizes. This equation assumes 100% spatial locality represents the ideal decrease in cache misses relative to the change in line size. Based on this assumption, perfect (100%) spatial locality corresponds with the case where an increase in line size by a factor of x causes a decrease in the number of cache misses by a factor of $1/x$, i.e. the number of cache misses is exactly inversely proportional to the ratio of line sizes. From this base case, the degree of spatial locality is then defined as the ratio of the actual decrease in cache misses compared to the ideal decrease in miss rate. Assuming m_a is the miss rate for the longer line size, m_b is the miss rate for the shorter line, and r is the ratio of the longer line size to the short line size, i.e. $r = l_a / l_b$, where l_a and l_b are the line sizes for the longer and shorter lines, respectively, then the equation becomes for spatial locality is:

$$spatial\ locality_r = \frac{m_b - m_a}{(m_b / r)}$$

Using this equation, the spatial locality results for data memory in a 32 KB cache are shown in Figure 6. For a given line size, the spatial locality results are relative to the next shorter line size, e.g. spatial locality for the 64 byte line is relative to the 32 byte line. As evident in the figure, the spatial locality for data memory is very good for small line sizes, up to 32 bytes, with average spatial localities of 55%, 39%, and 14% for line sizes of 16, 32, and 64 bytes, respectively. The benefit of spatial locality quickly decreases after 32 bytes, to the point of even becoming negative for many of the video standards with 128 byte line sizes, at which point the smaller number of sets causes many additional cache misses due to cache conflicts. Consequently it appears that the best cache line size for data memory is either 32 or 64 bytes in a 32 KB cache.

The spatial locality results for data memory in a 16 KB cache were measured in similar fashion. Because a smaller cache incurs more cache conflicts from the smaller number of sets, as the cache line size progressively increases, the spatial locality results for the 16 KB cache are more modest, with average spatial localities of 43%, 28%, and 7% for line sizes of 16, 32, and 64 bytes, respectively. Therefore, line sizes of 16 or 32 bytes are more appropriate for 16 KB caches. Likewise, cache sizes smaller than 16 KB can expect even poorer spatial locality benefits from increasing line size and will need their line sizes adjusted accordingly. However, it is also true that greater spatial locality benefits will be achieved from increasing cache line size in larger caches of 64 KB or greater. Therefore, it is necessary to balance the benefits of spatial locality from increasing line size versus the cache conflicts from fewer sets when selecting the cache configuration for a particular cache size in video processors.

In comparison, Hennessey and Patterson present results of a similar experiment examining the cache performance of general-purpose applications from SPEC92 for different line sizes and data cache sizes[10]. While their experiments did not quantitatively compute spatial locality, the results showed only a small performance improvement from increasing line sizes. Overall, it appears line sizes of 32 bytes or less are likely the best option for general-purpose applications.

3.6.3. Streaming Data

While no tests were performed to directly examine the value of stream buffers or stride prediction tables for video applications, the cache and line size results do provide evidence that such support could be beneficial. Video typically requires very large amounts of data. This is particularly true of video and computer graphics applications. In the instruction frequency results, video applications demonstrate very high frequencies of loads and stores. However, with the exception of JPEG-2000, none of the data working set sizes for these applications are 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 cast out of 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 that data, then throwing the data out, never (or rarely) needing to access it again. This perfectly describes the nature of streaming data. So there is significant evidence indicating the existence of considerable amounts of streaming data, so it is likely that performance gains can be obtained from memory prefetching support such as stream buffers, stride prediction tables, or a stream cache. For additional details and experimental data on stream-based prefetching for video applications, see Zucker et al.[12] and Struik et al. [13].

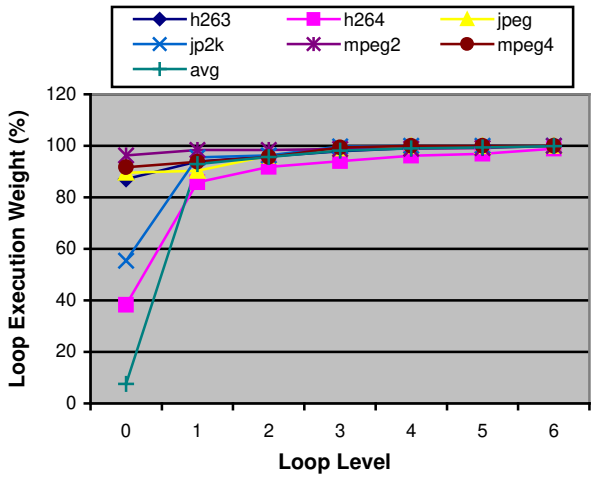

Figure 7. Percentage of loop execution weight by loop level.

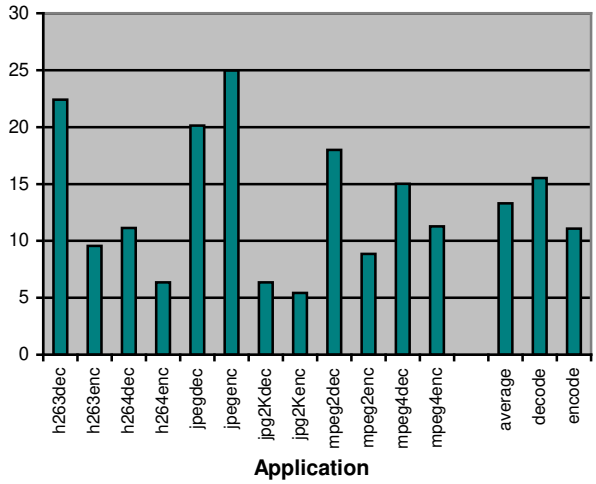

Figure 8. Average number of iterations for each benchmark.

3.7. Loop Statistics

Video applications are commonly know to spend the vast majority of their time executing over small sections of the program code. To more fully understand the processing characteristics within these frequently executed code sections, we again utilize the profiling tools to examine the loop characteristics of video applications, including the execution weight per loop level and the average number of iterations per loop invocation. These statistics will enable a greater understanding of the degree of processing regularity in video applications.

3.7.1. Loop Level Execution Weight

One mechanism for understanding the processing regularity of an application is to examine its loop behavior. Within programs, the innermost levels of loops are typically small and involve regular, straightforward computation, while the outer loops are generally much larger and contain more control code. Consequently, those applications that spend more of their time in inner loops generally have greater processing regularity, while those applications that spend significant portions of their execution time in the outer loops are more control-oriented and have much less processing regularity.

To measure the execution weight for each loop level, a depth-first search is first performed over all functions in the application, assigning a loop level to each loop in the application. The loop level is defined as the number of levels from an innermost loop. Innermost loops are assigned level 0, their parent loops are level 1, and so on. When a parent loop has multiple child loops with different loop levels, the loop level of the parent is defined as one greater than the maximum loop level of the child loops. In this definition for loop level, function boundaries are ignored so all loop levels are global.

The results, presented in Figure 7, indicate that most video applications spend 80-90% or more of their processing time within just the inner loops of the programs. Including the first level of outer loops below the inner loops, video applications aggregately spend nearly 95% of the execution time within the two innermost levels of loops. From these statistics it is evident that video applications have high processing regularity.

3.7.2. Loop Iteration

While the above information regarding execution weight per loop level indicates video applications have a tendency toward highly regular processing, it does not quantitatively define the degree of processing regularity. To quantitatively define the degree of processing regularity, it is necessary to determine how frequently loops iterate. We examine that characteristic in this section by measuring the average number of loop iterations per loop invocation.

The average number of loop iterations per loop invocation is calculated by taking the sum over all loops of the average number of iterations, i_m, multiplied by the number of invocations for that loop, e_m, and dividing by the total number invocations over all loops. The equation is as follows:

$$i_{avg} = \frac{\sum_{m=1}^{N} i_m * e_m}{\sum_{m=1}^{N} e_m}$$

Notice that the average number of loop iterations is weighted by the number of invocations for each loop as opposed to the loop execution weight for each loop, since weighting by loop execution weight would unfairly benefit those loops with more iterations per invocation.

Results on the average number of loop iterations per loop are shown in Figure 8. These results indicate that typical loops have a large number of iterations, about 13 iterations per loop on average. We can expect some variation in the average number of loops when using different data sets, but a comparison of the average number of iterations per loop on a separate input data set demonstrated variations that were within 5% of the results in Figure 8 in most cases.

The results also indicate that video decoding applications typically entail more iterations per loop invocation than encoding. This makes sense when you consider the nature of video decoding versus encoding. While video decoding decompresses the compressed video data in a deterministic fashion, video encoding entails significant decision making in the process of finding the redundant and least important data and eliminating it in order to achieve quality lossy compression. The control flow for this decision making usually entails more control code and loops that iterate less frequently, accounting for the majority of the difference in loop iterations per loop invocation for video decoding and encoding. In a similar fashion, the results also show that JPEG-2000 and H.264 have a relatively small number of iterations per loop. Whereas the other video coding applications perform entropy coding in a context-independent fashion, JPEG-2000 and H.264 both employ context-based adaptive entropy coding and rate-distortion optimization. These mechanisms enable more effective video compression, but require significant decision making control code that decreases the average number of iterations per loop and the loop level execution weight. Context-based adaptive coding is particularly control intensive, making coding decisions based on the correlation between neighboring pixels.

The results of the loop statistics indicate that video applications are highly loop-oriented. Nearly 95% of all execution time is spent within the two innermost loop levels, and loops have about 13 iterations per loop invocation on average. Overall, these results validate the significant processing regularity in video applications.

3.8. Instruction Level Parallelism

The last major characteristic of interest is instruction level parallelism (ILP), which is the amount of instruction parallelism achievable from simultaneously executing independent instructions in parallel in the CPU's datapath. There are two major categories of ILP: static ILP and dynamic ILP. Static ILP is the instruction level parallelism that can be found statically at compile time. Conversely, dynamic ILP is the instruction level parallelism that can be found by an out-of-order scheduler at run time. We shall examine both static and dynamic ILP subsequently in this section.

3.8.1. Static ILP

In measuring static ILP, we compile the video application code and examine its performance when executed on a generic statically-scheduled processor. To understand the variations in static ILP performance, in addition to measuring static ILP using only classical local compiler optimizations, we also need to examine the performance for more aggressive compiler optimizations that perform global scheduling in an attempt to maximize the static ILP. So in addition to the first compilation, which uses only classical local optimization, each application is compiled two additional times with more aggressive compiler optimizations. The second compilation the superscalar optimization level, performing speculation via the superblock optimization[14] and other optimizations such as loop unrolling. The final compilation also performs predication (a.k.a. conditional execution) via the hyperblock optimization[15].

Performance evaluation of the three levels of compilation is performed on a generic 4-issue[‡] VLIW processor. This processor is a simple statically-scheduled VLIW (very-long instruction word) architecture that supports 3 integer ALUs, 2 memory load/store units, 1 branch unit, 1 shifter, 1 multiplier, and 1 floating-point unit, which corresponds to the functional resource requirements of (3, 2, 1, 1, 1, 0) determined in section 3.2. The instruction latencies of these functional units are assumed to be 1 cycle for integer ALU instructions, 2 cycles for loads, 3 cycles for multiplies and floating-point instructions, and 10 cycles for divides. These instruction latencies are consistent with processors whose clock frequencies are on the order of 500 MHz to 1 GHz, which is the frequency range common to many video and media processors. The branch architecture of the processor uses only static branch prediction, and the memory hierarchy of the processor has a 16 KB instruction cache, a direct-mapped 32 KB data cache with a write-back/write-allocate write policy, and a 256 KB unified on-chip L2 cache that is 4-way set associative. Finally, the software architecture model supports a large number of registers (64 integer registers and 64 floating-point registers) in order to exclude spill and fill code while measuring the workload characteristics.

Figure 9 displays the static ILP results in terms of the average number of instructions executed per cycle (IPC) for the three compilation methods on each of the MB_{video} applications. Comparing the various compilation methods, it is somewhat surprising that the superscalar compilation path provides better speedup than the hyperblock compilation path. Because it incorporates both hyperblock and superscalar optimizations, the hyperblock has the potential for better performance than superscalar optimization. It is also important to note that having a higher ideal ILP does not necessarily indicate better performance. The hyperblock enables better branch prediction and has better memory performance, so it often wins out over superscalar compilation under realistic architecture assumptions. The hyperblock is also effective at providing performance comparable to the superscalar optimization level without the same degree of code explosion, which can considerably increase instruction cache effects in the absence of a larger instruction cache.

Overall, the results indicate that video applications contain a moderate degree of static instruction level parallelism. While the degree of static ILP is greater than that typical of general-purpose applications, the results are not as optimistic as one might hope. The reason is that instruc-tions along the control flow path in video applications often correspond to a series of computations being performed on a single pixel or set of pixels, and this series of computations is composed of instructions that are largely sequentially dependent upon each other. So even though video code does not have as much control code as general-purpose code, it offers only a little more ILP than general-purpose applications since there are longer instruction dependence chains.

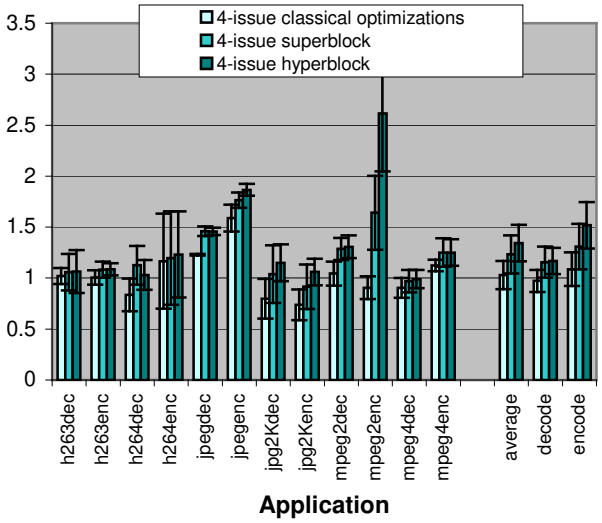

Figure 9. IPCs of a 4-issue architecture for various compilation methods.

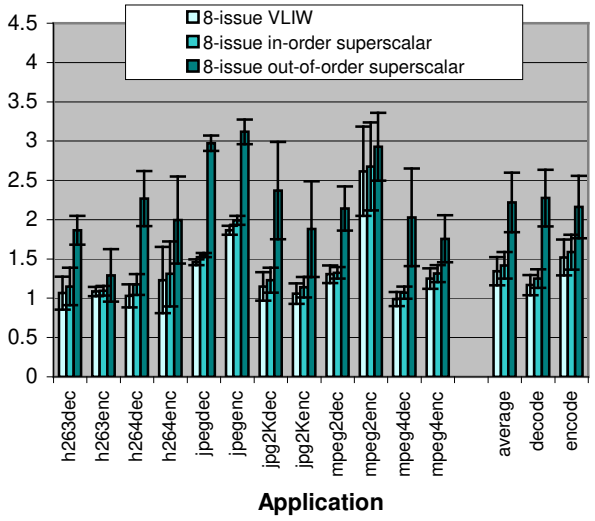

Figure 10. IPCs of a 4-issue architecture for various architecture models.

[‡] Our previous research has demonstrated that there is little benefit to more than 4 issue slots for static scheduling[16].

In reality, the majority of the parallelism that exists in video applications is data parallelism – the parallelism that exists between data elements that have little or no processing dependency between them. In video, this corresponds with the parallelism that exists between independent pixels, blocks of pixels, pictures/frames, video sequences, and so on. Unfortunately, this level of parallelism is of a coarser granularity than instruction level parallelism, so compilers are not able to target it very well. Currently, the most effective means for taking advantage of data parallelism are the subword parallelism instructions in the various multimedia ISA extensions, such as Intel's MMX[17] or Motorola's AltiVec[18], which enable the processor to efficiently perform SIMD processing at the level of individual instructions[19]. While this method is effective, allowing significant speedups, there are currently no compiler methods for subword parallelism, so programmers must manually write subword parallelism code in assembly or via library macros in high-level languages.

3.8.2. Static vs. Dynamic ILP

A second experiment was performed to demonstrate the effectiveness of dynamic scheduling for video processing. The MB_{video} benchmarks were run on three different ILP processor architecture models with progressively greater degrees of dynamic scheduling: (a) an 8-issue VLIW processor, (b) an 8-issue in-order superscalar processor, and (c) an 8-issue out-of-order superscalar processor. Shown in Figure 10 are the results comparing the average IPCs for these three architectures. These results indicate two important conclusions. First, static scheduling performs nearly as well as dynamic in-order scheduling for media processing. With average IPCs of 1.34 and 1.42 for the VLIW and in-order superscalar, respectively, there is only 6% difference in performance. We were expecting a much higher differential since dynamic in-order scheduling enables instructions to continue issuing on non-dependent memory stalls. Second, it is apparent that dynamic out-of-order scheduling, with an average IPC of 2.22, provides much better performance than static scheduling. The out-of-order super-scalar processor enables 66% better performance on average than a VLIW processor over the MediaBench video applications. While the out-of-order superscalar processor entails much greater complexity and power than VLIW and in-order processors, for high-performance solutions, it may be desirable.

While prior research studies have found considerable parallelism in video applications[11], it is evident from this experiment that such levels of parallelism are not likely to be attained with just instruction level parallelism. ILP provides respectable parallelism, with typical scheduling performance of about 2 IPC, but achieving high degrees of parallelism is critical to the success of programmable video processors. Consequently, it is necessary to explore additional avenues for parallelism.

4. WORKLOAD VARIABILITY ACROSS INPUTS

With any workload, it is important to understand how the workload characteristics will vary with different input data and input parameters. For video applications, the most common variations across different inputs are due to frame size, target bit rate, degree of motion, and search window size (for encoding). Aside from those input variations, the only other major differences in the application characteristics arise from the different algorithms and methods that the software designers use in implementing the applications. Fortunately, there is significant variation across the implementations of the many benchmarks in MediaBench Video, so MB_{video} demonstrates those variations effectively.

With respect to resolution of the video frames, experiments were performed on CIF (352x288 pixels) and 16CIF (1408x1152 pixels) video sequences as well as the base 4CIF (704x576 pixels) video sequence. Overall, little variation was found across the various processing characteristics, with the exception of execution time, of course. The execution time was found to increase nearly linearly with the number of pixels. The dynamic instruction count increased by an average of 4.25x with each doubling of frame height and width (i.e. quadrupling frame size).

Varying the target bit rate has a moderate effect on the dynamic instruction count and L1 data cache miss rates. The base video sequence was designed for moderate bit rate, so two input configurations (of the same raw video) with low and high bit rates, corresponding to half and double the target bit rate of the base sequence, respectively, were to tested to examine the variations in processing characteristics due to bit rate. Each doubling of the target bit rate served to increase the dynamic instruction count (i.e. approximate execute time) by an average of 11% for decoding and 3% for encoding. Each doubling of the bit rate also resulted in decreases in the L1 data cache miss rate by 3-5% for decoding and 0.1-1.5% for encoding.

The variations due to different degrees of motion in the video sequences were also examined. The base video sequence had a medium degree of motion, so two additional video sequences with low and high motion, respectively, were also tested. It was found that varying the degree of motion varies the dynamic instruction count by an average of only 4-5% for low and medium degrees of video motion. Conversely, high degrees of motion resulted in significantly greater execution times, with a 24% increase in dynamic instruction count for encoding high motion video (decoding was unaffected). Varying the degree of video motion also caused variations in the L1 data cache miss rate of 2-3%, but otherwise, none of the other processing characteristics was significantly impacted by the degree of motion.

Finally, experiments were performed to vary the search window size and determine its impact of the workload characteristics. Since the various codecs had different implementations for motion estimation, it was not possible to explicitly vary the search window size on all codecs. For the MPEG-2 encoder, the search window size was varied from +/-16 for the base encoder configuration to +/-8 and +/-32 for the two alternate encoder configurations. For the H.263 encoder, the search window size could only be varied for +/-16 and +/-8. The ffmpeg codec used for MPEG-4 dynamically varies search window size, and the version of H.264 used did not yet support different search window sizes, so only MPEG-2 and H.263 could be tested for variations in search window size.

Both MPEG-2 and H.263 showed similar results with respect to variations in search window size. As expected, for the decoding process, the search window size had negligible impact on the workload characteristics. Conversely, for encoding there were significant variations in dynamic instruction count and L1 data cache miss rates. Increasing the search window size from +/-8 to +/-16 approximately doubled the dynamic instruction count and reduced the L1 data cache miss rate by approximately 40%. In the MPEG-2 codec, increasing the search window size again to +/-32 further increased dynamic instruction count by nearly 2.6x over the +/-16 search window execution time, and also reduced the L1 data cache miss rate by 50% over the +/-16 search window miss rate. This increase in execution tie and decrease in cache miss rate is to be expected from increasing search window size is to be expected. Execution time for motion estimation is known to increase nearly linearly with the area of the search window (for full search, as performed by MPEG-2 and H.263). Likewise, since motion estimation has good cache performance, the average L1 data cache miss rate will similarly increase when more time is spent in the motion estimation kernel.

Overall, we can conclude that, aside from the anticipated variations in execution time and cache miss rate, there was little impact on the workload characteristics from different video inputs and execution parameterizations.

5. SUMMARY

This paper presents MB_{video}, the MediaBench video benchmark suite, as the new video benchmarking tool for enabling video systems research and design for the next generation of video. First introduced in 1997, the first generation of MediaBench has proven invaluable to the research community in understanding multimedia applications and designing multimedia systems. Recently, the MediaBench Consortium was founded to continue the mission originally set forth by MediaBench, and its goal is the continued development and refinement of the benchmark suite. Among its immediate aims is the goal to enable both a composite benchmark, similar is scope to the original MediaBench benchmark, as well as area-specific benchmarks for each individual media type. While currently only in the early stages of design of the new benchmarks, the audio and video area-specific benchmarks have been completed. The video benchmark, MB_{video}, contains both the current popular video standards such as H.263, JPEG, MPEG-2 and MPEG-4, as well as the recent and emerging standards, including JPEG-2000 and H.264.

The paper also presents a comprehensive workload evaluation of the new MediaBench Video benchmark suite, providing a quantitative evaluation of many application characteristics, including instruction frequencies, basic block sizes, branch prediction rates, data sizes, working set sizes, spatial locality, loop characteristics, and ILP scheduling performance.

In particular, the loop characteristics demonstrate the impact that the recent and emerging standards of JPEG-2000 and H.264 have on the computational requirements of video systems. These standards employ greater degrees of analysis to enable higher compression rates, utilizing such new methods as rate-distortion optimization, context-adaptive entropy coding, and intra-block/macroblock prediction. In addition to the much longer encoding and decoding times these

standards entail, the low average loop iteration measurements and loop level execution weights for the JPEG-2000 and H.264 encoders demonstrate the increasing amounts of control code and lower processing regularity that result from these advanced compression tools.

REFERENCES

1. Peter Pirsch, Hans-Joachim Stolberg, Yen-Kuang Chen, and S. Y. Kung, "On Implementation of Media Processors," *IEEE Signal Processing Magazine*, vol. 14, no. 4, pp. 48-51, July 1997.
2. S. Y. Kung and Yen-Kuang Chen, "On Architectural Styles for Multimedia Signal Processors," *Proceedings of the IEEE Workshop on Multimedia Signal Processing*, Princeton, NJ, June 1997.
3. Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Video Communications Systems," *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.
4. MediaBench website: http://cares.icsl.ucla.edu/Mediabench/
5. Standard Performance Evaluation Corporation (SPEC) website: http://www.spec.org/
6. Jason Fritts, "MediaBench II," presented at the *2004 Workshop on Media and Signal Processors for Embedded Systems and SoCs (MASES; held in conjunction with CASES 2004)*, September 2004.
7. P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: an architectural framework for multiple-instruction-issue processors," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 266-275, Toronto, Canada, May 1991.
8. IMPACT compiler research group website: http://www.crhc.uiuc.edu/Impact/
9. John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach, 3rd edition*, Morgan Kaufmann Publishers, Inc., 2003.
10. John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach, 2nd edition*, Morgan Kaufmann Publishers, Inc., 1996.
11. Heng Liao and Andrew Wolfe, "Avaliable Parallelism in Video Applications," *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.
12. Daniel F. Zucker, Ruby B. Lee, and Michael J. Flynn, "Hardware and Software Cache Prefetching Techniques for MPEG Benchmarks," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 10, no. 5, pp. 782-796, August 2000.
13. Pieter Struik, Pieter van der Wolf, and Andy D. Pimentel, "A Combined Hardware/Software Solution for Stream Prefetching in Multimedia Applications," *SPIE Photonics West, Multimedia Hardware Architecture*, pp. 120-130, January 1998
14. 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," *Journal of Supercomputing*, Kluwer Academic Publishers, pp. 229-248, 1993.
15. 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 International Symposium on Microarchitecture*, pp. 45-54, December 1992.
16. Jason Fritts, "Architecture and Compiler Design Issues in Programmable Media Processors," Ph.D. Thesis, Dept. of Electrical Engineering, Princeton University, 2000.
17. A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," *IEEE Micro*, vol. 16, no. 4, pp. 42-50, August 1996.
18. K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales, "AltiVec Extension to PowerPC Accelerates Media Processing," *IEEE Micro*, vol. 20, no. 2, pp. 85-95, March 2000.
19. R. B. Lee, "Accelerating Multimedia with Enhanced Microprocessors," *IEEE Micro*, vol. 15, no. 2, pp. 22-32, April 1995.