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

Jason E. Fritts[a], Frederick W. Steiling[b], Joseph A. Tucek[c], Wayne Wolf[d]

[a]*Department of Mathematics and Computer Science, Saint Louis University, St. Louis, MO*
[b]*Alcotek, Inc., St. Louis, MO*
[c]*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL*
[d]*School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA*

Abstract

The first step towards the design of video processors and systems is to achieve an understanding of the major applications, including not only the theory, but also the workload characteristics of the many image and video compression standards. Introduced in 1997, the MediaBench benchmark suite provided the first set of full application-level benchmarks for multimedia, and has consequently enabled significant research in computer architecture and compiler research for media systems. To expedite the next generation of multimedia systems research, we are developing the MediaBench II benchmark suite, incorporating benchmarks from the latest multimedia technologies, and providing both a single composite benchmark suite ($MB2_{comp}$) as well as separate sub-suites for each area of multimedia. For video, MediaBench II Video ($MB2_{video}$) includes both the popular mainstream video compression standards, such as JPEG, H.263, and MPEG-2, and the more recent and emerging standards, including MPEG-4, JPEG-2000, and H.264. This paper first discusses the goals for MediaBench II and the design of the $MB2_{video}$ sub-suite. The paper then presents the results of a comprehensive workload evaluation of $MB2_{video}$. In particular, while the workload evaluation demonstrates the high processing regularity of video workloads, as compared with general-purpose workloads, it also illustrates how the growing complexity of the emerging video standards is beginning to negatively impact video workload characteristics.

Key words: multimedia benchmarks, workload evaluation, media processors, processor design, MediaBench

1. Introduction

The last decade has seen significant growth in the use of video for communication, entertainment, and archival purposes. While little more than ten years ago it was impractical and generally infeasible to play or decode video clips on even the most powerful desktop computers, both decoding and encoding now occur on a regular basis in a variety of computing systems, ranging from desktop computers, video conferencing systems, and video databases to PDAs, digital cameras, home theater systems, portable DVD players, and even cell phones. While the video quality and resolution across these systems currently varies considerably based on the computing capabilities of the systems, we can expect two trends to continue indefinitely into the future: (1) research in information theory will continue to develop increasingly sophisticated methods for maximizing video compression, and (2) users will continue to demand higher quality video at lower prices. The net result is that designers of video processing systems will continue to face the challenge of providing ever greater 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 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 whether it requires hardware and/or software for separate standards, (2) whether the design should employ either more application-specific hardware, providing more computational capability but less flexibility, or more software, providing greater flexibility but necessitating 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 is performed in dedicated hardware versus software, it is necessary to understand the workload characteristics of the video applications in selecting or designing a video processor. Much literature is available that covers the algorithms and information theory behind the many video standards, so this paper focuses on the latter category, providing engineers a comprehensive understanding of video workload characteristics.

Achieving an understanding of the workload characteristics of an application area requires a benchmark suite representative of that area. For applications in multimedia and communications, the MediaBench benchmark suite, introduced in 1997 by Lee et al. [10], has served as the dominant full-application benchmark suite for research and development of multimedia systems over the last decade. However, like any benchmark suite, MediaBench has aged in that time, progressively becoming less representative of the emerging workloads, instead becoming more indicative of the current and classic video workloads. To continue to serve as an effective benchmark suite for media systems, it is necessary to update and refresh the benchmark periodically.

In order to maintain and improve upon the mission originally set forth for MediaBench, we are spearheading the development of the second generation of MediaBench. The ultimate goal is the development of an organization to provide for the continuing development and refinement of the MediaBench benchmark suite. The goal of this organization is to provide regular updates to the MediaBench benchmarks so that they will not only continue to serve as an effective tool for multimedia workload characterization, but will also continue to grow and meet the changing and diversified needs of future media systems.

This paper presents the goals and design of MediaBench II ($MB2$), the next generation of benchmarks for multimedia. This paper first outlines the goals for the design of MediaBench II, and then presents the specific details for MediaBench II Video ($MB2_{video}$), the first media-specific benchmark sub-suite in the $MB2$ benchmark suite. Finally, the bulk of the paper presents the results of a comprehensive workload evaluation of $MB2_{video}$, including comparisons between video workloads and general-purpose workloads, as well as a comparative analysis of the differences between the more recent and emerging video applications and the older classic applications.

The remainder of this paper is organized as follows: Section 2 presents MediaBench, including MediaBench I, the goals for MediaBench II, and the full details for $MB2_{video}$, the video benchmark sub-suite of MediaBench II. Section 3 presents the results of a high-level workload evaluation, performed using the OProfile profiling tool. Section 4 presents the results of a lower-level workload evaluation, performed using the IMPACT compiler/simulation environment. Section 5 examines the workload variability due to different input data sets and execution parameters. Finally, Section 6 concludes the paper.

2. MediaBench: Then and Now

We first present an overview of the original MediaBench benchmark suite, and then discuss the design goals for the next generation of MediaBench, the MediaBench II benchmark suite. Finally, we provide a detailed discussion of the design and implementation of $MB2_{video}$, the video-specific benchmark sub-suite within MediaBench II.

2.1. MediaBench I

Founded in 1997 by Lee et al. [10], MediaBench was designed with a focus towards 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 coding, to audio and speech processing, and even encryption and computer graphics. It proved to be an invaluable research tool for computer architecture and compiler design of media systems.

The original MediaBench benchmark suite was composed of 11 application packages covering six distinct media areas: video, image, graphics, audio, speech, and security. The video area contained only one video codec, MPEG-2, for encoding and decoding video sequences. Audio was similarly characterized by a single codec, ADPCM, for encoding and decoding audio streams. The image media type comprised three applications, including JPEG and EPIC for coding standard color images, as well as Ghostscript for Postscript transcoding. The speech area contained three applications, including GSM and G.721 for encoding and decoding speech, as well as Rasta, a speech recognition application. Security contained two applications, PGP and Pegwit, for encrypting and decrypting messages. Finally, computer graphics used Mesa, a set of computer graphics libraries similar to OpenGL, which included three demo programs that served as the benchmarks for graphics.

MediaBench has served the computing industry well for many years, and is still commonly used today. However, like any benchmark suite, MediaBench has aged over the last decade. When used today, research groups commonly augment MediaBench with additional applications that are more relevant to today's media systems. And while this produces more representative results, it makes reproducibility of results and systematic comparison of methods by different research groups problematic, since none of these additional applications have been standardized.

Both to facilitate effective experimental comparison and to ensure that the workload is appropriately representative of the target application area, it is necessary to standardize the set of applications, the input data sets, and the parameterizations of the applications for each input data set. This includes defining the workload based not only on a representative set of applications defining the target area, but also utilizing typical data sets and parameters for those applications. And finally, assuming an implicit or explicit weighting method is used to coalesce the individual statistics into an aggregate set of statistics characterizing the entire workload, the number and type of applications must also be balanced to ensure that each benchmark appropriately impacts the final aggregate results.

For future multimedia systems research and development, what is needed is a systematically designed and appropriately balanced set of benchmarks that effectively characterize multimedia workloads across typical input data sets. This is a fundamental goal in the development of the MediaBench II benchmark suite.

2.2. Goals for MediaBench II

In designing the initial benchmark suite for MediaBench II, a key goal was the expansion of the benchmark suite into distinct application areas. Consequently, the next generation of MediaBench will include not only a composite benchmark suite, $MB2_{comp}$, which includes a balanced set of the most recent and emerging applications from all media areas, but also separate benchmark suites for each of the distinct media types (e.g. audio, video, speech, security, etc.). These area-specific media benchmark sub-suites will not only include the corresponding media benchmark(s) from $MB2_{comp}$, but also complement them with additional popular and classic applications from that area. In particular, the composite benchmark suite will serve as the flagship representing the *emerging* multimedia and communications applications, while the area-specific benchmark sub-

suites will serve as the benchmarks that comprehensively represent each specific media area.

The initial set of media-specific applications will include 5 of the 6 media areas from the original MediaBench suite, including *video* (which combines both video and image applications, since they have very similar theoretic foundations), *audio*, *speech*, *security*. We currently do not plan to develop a computer graphics media-specific benchmark, since there are a number of other strong computer graphics benchmarks that already serve the needs of that community. We also plan to add new media areas, such as *vision*, since the fields of image analysis and computer vision are progressively moving into mainstream commercial use, and they employ a wealth of image processing, analysis, and machine learning algorithms that are not currently covered by other media areas. Finally, there is also some interest in having a *kernel* benchmark that provides implementations of many of the common kernels in media processing, so that benchmark may also be developed at some point in the future.

Finally, a potential long-term goal is to develop parallel implementations of the media benchmarks, in a fashion similar to ALPBench, the *All Levels of Parallelism* benchmark developed by Li et al. [11], which targets exposing the instruction, thread, and data level parallelism within an application, and parallelizing the application accordingly. However, developing parallel implementations of existing media applications is an intensive undertaking, so assistance from the research community will likely be needed to achieve this goal.

2.3. MediaBench II Video

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. We elected to group the image and video applications together into the *video* media type since the majority of video methods are based on the same theoretical foundations as imaging. Aside from JPEG-2000's uniqueness as a wavelet-based codec, the primary difference between most image and video coding methods is the inter-frame redundancy elimination provided by motion estimation and compensation.

As with any benchmark suite, in designing the $MB2_{video}$ benchmark suite it was important to consider:

1. application selection,
2. input data set selection and design, and
3. application parameterization

2.3.1. Application Selection

For application selection, the foremost goal is selecting a set of applications that effectively represent the common applications in that area. For image and video processing, by far the most common application is compression, which includes both decoding and encoding. The other primary application areas are image and video editing and enhancement, which are more GUI-related applications. They are predominantly constrained by user interaction, and as such are difficult to model for benchmarking. Moreover, editing and enhancement are generally not resource intensive and are largely independent of real-time deadlines since most of the application time is spent waiting for user input. Consequently, the $MB2_{video}$ benchmarks, like the original MediaBench, focuses on image and video coding.

Since encoding and decoding of images and video are the primary functions being benchmarked, the second major goal is selecting a set of applications that covers the full range of popular and emerging video coding methods. In doing so, it is important both to balance the selection of recent and emerging methods with popular and classic methods, and to balance video with imaging.

Because we are targeting the design of a benchmark suite for computer architecture, compiler, and systems research and development, it is crucial to use open source applications. This distinction is key for compiler research, but also important for computer architecture research since many popular performance analysis tools require access to the source code. Systems-level research does not necessitate source code, but it can still be advantageous for such work.

The end result of the application selection for $MB2_{video}$ is a set of six image and video codecs, as detailed in Table 1. Listed in chronological order, they include JPEG, H.263, MPEG-2, MPEG-4, JPEG-2000, and H.264. The first three benchmarks are the older, classic applications from the first generation of image and video coding, and will be henceforth be called the *Gen-1* codecs. The more recent and emerging applications from the current generation of image and video codecs include MPEG-4, JPEG-2000, and H.264, and will similarly be called *Gen-2*, the second generation codecs[1]. Two of the benchmarks, JPEG and JPEG-2000, are predominantly imaging methods, but may also be employed for video coding, while the other four benchmarks are strictly video standards. Each codec has both an encoder and a decoder, so there are twelve distinct benchmarks, with encoding and decoding each comprising one-half of the workload.

Among the six applications, the two most recent methods, JPEG-2000 and H.264, are the most theoretically distinct from the others. H.264 employs a host of new video coding technologies over prior video methods, including rate-distortion optimization, intra-prediction, and context-adaptive entropy coding. JPEG-2000 is potentially the most distinct of the methods, since it uses wavelets or sub-band coding for intra-frame image compression. None of the other applications currently employ such methods, though video methods based on wavelets are imminent.

The MPEG-4 codec is also quite distinct from the others in its own right. While the MPEG-4 is the oldest of the *Gen-2* codecs, with foundations fundamentally very similar to MPEG-2, the distinction of this MPEG-4 codec lies in its implementation. The the source code for the MPEG-4 codec, which is provided by the FFMpeg group, has been co-developed by a large community of developers over many years time to support a variety of video coding standards. In contrast, most of the other

[1] While MPEG-4 really falls between the two generations, the implementation of this particular MPEG-4 codec is more indicative of second generation video codecs.

Table 1: Description of the $MB2_{video}$ benchmark suite.

Video Generation	*Application*	*Description*
1st Generation	JPEG	A video coder (*cjpeg*) and decoder (*djpeg*) based on the ISO JPEG standard for image compression. Source code produced by the Independent JPEG Group.
	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.
	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).
2nd Generation	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 produced by FFMpeg Multimedia Systems.
	JPEG-2000	A video coder (*jpg2000enc*) and decoder (*jpg2000dec*) based on the recent ISO JPEG-2000 standard for wavelet-based image compression. Source code produced by The JasPer Project.
	H.264	A video coder (*h264enc*) and decoder (*h264dec*) based on the 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.

applications are software reference codecs that are largely unoptimized[2].

Two important benefits arise from the design of the FFMpeg MPEG-4 codec. The first is that it has been been highly optimized for speed and efficiency. Among its many optimizations is the use of the EPZS motion estimation algorithm [17], which is a particularly fast and effective approach to motion estimation. Likewise, it has been adapted to support the multimedia ISA extensions for a number of common processors[3]. We will utilize this feature to demonstrate the benefit of subword parallelism on execution time in Section 3.4. The other advantage is that its functions have been organized and modularized to efficiently support a wide variety of video coding standards across many different platforms. This modularity is particularly indicative of video applications produced by industry, and as well shall see, has distinct implications with respect to the video workload characteristics.

Throughout the workload evaluations in this paper we shall be highlighting the differences between the first and second generation video codecs, but even before we begin examining the dynamic characteristics of these applications one difference is immediately apparent. As shown in Table 2, there is a significant difference in code size between the *Gen-1* and *Gen-2* codecs. Whereas the *Gen-1* codecs average fewer than 60 files and 23 thousand lines of code per codec, then *Gen-2* codecs have nearly 160 files and 110 thousand lines of code per codec. The large size of the *Gen-2* codecs is somewhat skewed by the FFMpeg MPEG-4 implementation, which is especially large because it supports a wide variety of video standards across many platforms, but the JPEG benchmark similarly skews the size of the *Gen-1* codecs, since it supports conversion between a wide range of image file types.

Table 2: Numbers of files and thousands of Lines of Code (KLOC) in the $MB2_{video}$ benchmark.

Gen	*Standard*	*App*	*Files*	*KLOC*
Gen-1	JPEG	*cjpeg*	85	35
		djpeg		
	H.263	*h263dec*	25	7.5
		h263enc	22	8.1
	MPEG-2	*mpeg2dec*	22	9.8
		mpeg2enc	24	7.4
Gen-2	MPEG-4	*mpeg4dec*	263	200
		mpeg4enc		
	JPEG-2000	*jpg2000dec*	89	36.7
		jpg2000enc		
	H.264	*h264dec*	55	32.4
		h264enc	67	57.9

The code size differences between the *Gen-1* and *Gen-2* codecs are indicative of the growing theoretic and algorithmic complexity between the first and second generation video standards. The second generation video standards, in addition to supporting a much wider range of resolutions, employ many additional image processing methods to maximize the compression while minimizing the degradation at high levels of compression. Included in such methods are major additions

[2]The JPEG codec is the exception, since it is the version used in most Linux distributions. However, it was developed by a smaller team and is not as highly optimized as the FFMpeg MPEG-4 codec.

[3]This feature is architecture-dependent and so is turned off by default for the $MB2_{video}$ input data sets.

like rate-distortion optimization, context-adaptive entropy coding, intra-prediction, and/or wavelet coding, as well as smaller changes like arithmetic coding and loop filters. And as discussed in Sections 3 and 4, this growing complexity of second generation video codecs is apparent in many aspects of the workload characteristics as well.

2.3.2. Data Set and Parameters Selection

Selecting the applications is just the first step in the design of a benchmark suite. Each benchmark must also be designed to execute in a fashion indicative of standard uses of that application. Furthermore, when a benchmark suite contains a group of applications that serve a common purpose, such as encoding and decoding in image and video coding, the input data sets used in characterizing that workload should be employed across all related applications, thereby enabling direct comparison between them. Such use of data sets across multiple applications requires that the input data sets be representative of the common uses over the full set of applications.

Similar to data set design, parameterization selection is also important. The execution parameters for an application should be defined to be representative of the typical settings users employ when running that application. For parameters common to all applications serving a similar purpose, such as bit rate and resolution in image and video coding, those parameters should be set identically across all relevant applications. But for parameters unique to an application, they should be defined according to the common settings employed by the user community.

Finally, when selecting data sets and execution parameters, it is also important to provide an understanding on how common variations in the data set and execution parameters impact workload. In Section 5, we present the results of an experiment examining the workload variability due to common variations in video data sets and execution parameters. The experiments show the workload variations for different bit rates, degrees of motion, resolutions, and search window sizes (in encoding).

Given these considerations, the current input data set for the $MB2_{video}$ benchmarks is a 4CIF (704x576) resolution sequence containing 9 frames of video, which corresponds to about $\frac{1}{3}$ of a second of video. The raw video for this sequence was originally a much higher resolution sequence that was downsampled to 4CIF. This choice was made to facilitate comparison with 16CIF as well as CIF resolutions in Section 5. The video has moderate motion, featuring a woman talking in the foreground and another person walking up a flight of stairs in the background. A compression rate of 96:1 was selected for encoding and in generating the input data sets to the decoders. The organization of the frames in the 9-frame video input is IBBBPBBBP[4].

Mid-range levels of resolution, bit rate[5], and degree of motion have purposefully been selected for the video sequence, enabling experimental comparison with lower and higher levels in Section 5, while also remaining indicative of common video data sets. As shall be seen in Section 5, the variations in workload characteristics for lower or higher resolutions, bit rates, and degrees of motion are fairly minimal beyond the expected variations in execution time.

The length of this initial input data set was designed for computer architecture simulation. For systems- and compiler-level research, a larger data set will be forthcoming in the future. The larger sequence will target 900 frames, or about 30 seconds of video.

3. High-Level Workload Evaluation

Video applications are understood to have certain characteristics distinct from typical general-purpose applications, such as 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. The purpose of Sections 3 and 4 is to provide a comprehensive workload evaluation quantifying those characteristics using the $MB2_{video}$ benchmark suite. This section will focus on a high-level workload characterization of the video applications on an Intel Pentium processor, while Section 4 provides a low-level workload evaluation on a generic RISC architecture.

The purpose of the high-level workload evaluation is to provide a first-order understanding of video workload characteristics using profiling and timing. The profiling experiments determine the relative execution time spent in each of the major procedures, while the timing experiments illustrate both the typical execution times and the common variations in memory access time. A final experiment is also performed with the MPEG-4 codec that demonstrates the benefit of using multimedia ISA extensions.

These experiments were performed on a PC-based Linux machine using the OProfile profiling tools and the Linux `time` command. While the results for such an experiment are architecture-dependent, they do provide a first-order perspective on execution time, as well as the relative time spent in the application procedures, library procedures, and the kernel. In particular, the profiling and execution time results will begin to demarcate some of the differences between the first and second generation video workloads.

3.1. Experimental Environment

The execution profile and timing results were statistically computed from 30 runs of the $MB2_{video}$ benchmark suite. The benchmarks were executed in round-robin order on an IBM ThinkPad T43 laptop with a 1.86 GHz Intel Pentium M and 1 GB RAM, running Linux Fedora Core 4 with the 2.6.15 kernel. The execution time and profiling measurements were made using the Linux `time` tool and the OProfile tools, respectively.

The benchmarks were run on an unloaded system, with every precaution taken to minimize or eliminate warm start effects. The experiments were run immediately following a hard system reset to ensure that the first run of each application was a cold

[4] In benchmarks where B-frames are not supported, P-frames are used instead. Likewise, when P-frames are not supported, I-frames are used.

[5] While mid-range for video applications, the 96:1 compression rate is actually a bit high for JPEG and JPEG-2000.

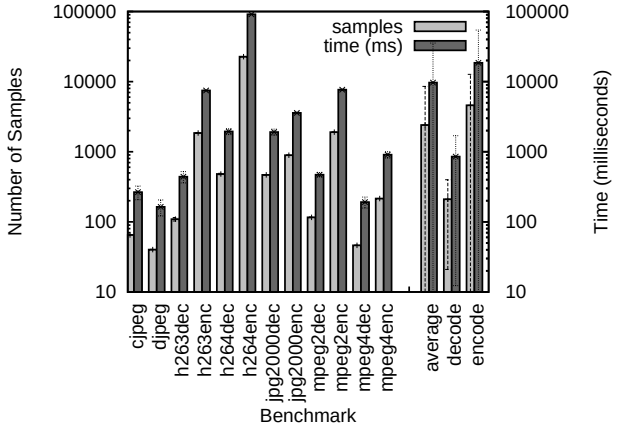

Figure 1: Average execution time and number of OProfile samples for each video benchmark in $MB2_{video}$.

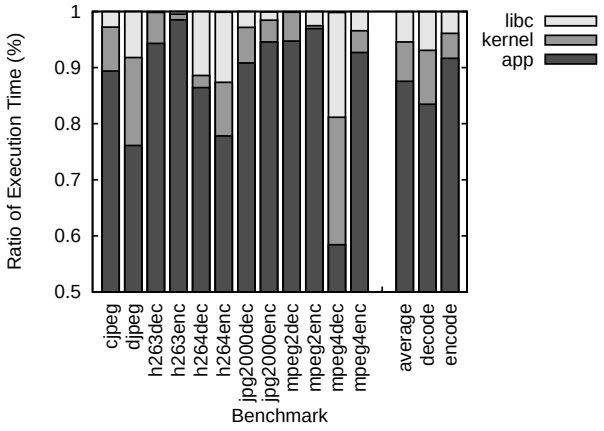

Figure 2: Percent of execution time spent in application, kernel, and libraries for each benchmark in $MB2_{video}$.

run. To minimize the warm start effects in subsequent runs of each benchmark, the benchmarks were executed in round-robin order. Each of the twelve benchmarks was executed once in succession, and this sequence was repeated 30 times. Finally, each benchmark has its own copy of the input data files, so no file or memory sharing occurs between benchmarks.

The profiling results were measured using the OProfile tools [21]. OProfile enables application profiling in a manner similar to `gprof`, but with the advantage of being able to profile existing binaries without having to first instrument and recompile the code. Furthermore, it provides profiling of both application and system code, with a low overhead of only 1-8%. The primary drawback is that OProfile currently only works in Linux systems using kernels 2.2, 2.4, and 2.6.

The OProfile tools provide a few different methods for sampling the code during profiling, but the Intel Pentium M processor only supports interrupt timer sampling at present. This profile sampling method uses the Linux kernel timer interrupt, which operates at the rate of 250 Hz for this system[6]. In other words, in these experiments the OProfile tools sample the process once every 4 ms. Each sampling instance examines the procedure currently being executed, and based on procedure address ranges determines to which procedure the sampled instruction belongs.

The results from the 30 runs were coalesced to generate the average, standard deviation, and 95% confidence intervals for the sample counts and execution times for each procedure in the application, libraries, and kernel.

3.2. Profiling Results

The profiling results are summarized in Figures 1, 2, and 3. For reference, the full OProfile results are available on the MediaBench II website [20].

Figure 1 indicates the number of OProfile samples[7] measured for each of the video benchmarks (as well as the execution time measurements taken using the Linux `time` command, which are discussed in Section 3.3). Examination of the profile sample counts points out two things. First, as anticipated, the average difference in decoding time versus encoding time is at least an order of magnitude. The decoding benchmarks require 141 samples on average while the encoding benchmarks require 4467 samples on average. Second, the second generation video codecs typically take much longer to execute than the first generation codecs. The *Gen-2* codecs average 332 samples for decoding versus only 88 samples for decoding in *Gen-1*. Likewise, encoding averages 7927 samples for the *Gen-2* codecs but only 1272 samples for *Gen-1*.

MPEG-4 is the exception to this trend. As noted in Section 2.3.1, the FFMpeg MPEG-4 codec has been highly optimized, and as a result only requires 215 samples for encoding and 46 samples for decoding. Given MPEG-4's optimized speed, if we instead compare only the two latest codecs, JPEG-2000 and H.264, with the four older codecs, the execution time contrast is even more significant. The decoding and encoding times are 474 and 11782 samples for JPEG-2000 and H.264 versus 78 and 1008 samples for the other applications, which demonstrates that the emerging video applications are approximately an order of magnitude slower, taking 6.1× and 11.7× longer to perform decoding and encoding, respectively.

Figure 2 breakdowns down the sample counts according to the fraction of time spent in the application, libraries, and kernel. On average, nearly 90% of the execution time is spent in the actual application, while the remaining time is divided roughly evenly between the libraries and the kernel. The execution time within the libraries is primarily spent in memory functions such as `memcpy`, `memset`, `malloc`, and `free`, with file functions such as `getc` and `putc` accounting for most of

[6]The sampling event and frequency is fixed in the interrupt timer sampling method, but is much more parameterizable in processors supporting one or both of the other sampling methods.

[7]While the video applications use 9 frames, JPEG and JPEG-2000 only use one frame, so for comparison the reported times and number of samples for these two applications is the measured value multiplied by 9 to give the expected results for 9 frames.

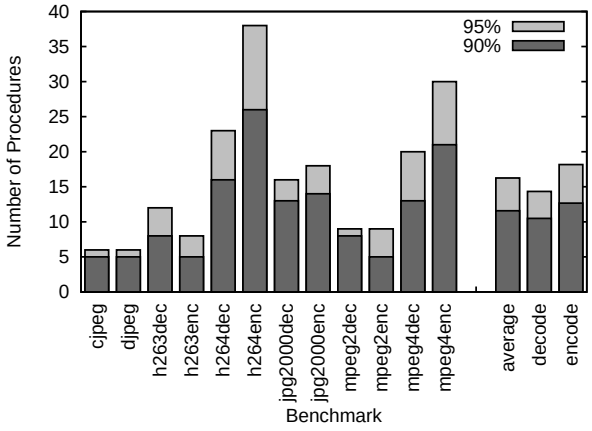

Figure 3: Number of procedures constituting 90% and 95% of the execution time, respectively, for each benchmark in $MB2_{video}$.

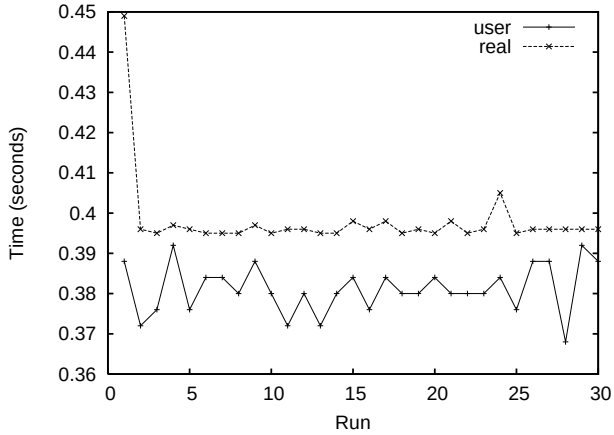

Figure 4: Execution time per run for JPEG-2000 encoding.

the remaining time. OProfile is unfortunately unable to give a breakdown of the kernel time for our platform, but clearly the kernel is not a significant source of execution time in most benchmarks.

Breaking down the sampling results further, we also examined the percentage of time spent in the major procedures in each application. Considering the old adage "90% of execution time is spent in 10% of the code", we focused on the minimal set of procedures comprising 90% of the execution time, and discovered that the first generation codecs spend the majority of their time in a much smaller set of procedures.

Figure 3 more clearly delineates this trend, showing the number of procedures that constitute 90% and 95% of the execution time for each of the applications. While the *Gen-1* codecs spend 90% of their time in only 7 procedures for decoding and 5 procedures for encoding, the *Gen-2* codecs require 14 and 20 procedures for decoding and encoding, respectively. In other words, there are $2-4\times$ as many procedures comprising the key set of functions in the second generation video codecs.

This trend can be attributed to two factors. The first is the greater algorithmic complexity of the more recent video applications. JPEG-2000 and H.264 both include a number of technologies not found in earlier standards, including alternate spatial-to-frequency transform methods, rate-distortion optimization, context-adaptive entropy coding, and intra-prediction, among others.

The second factor, which occurs most significantly in the MPEG-4 codec, is modular design. Recall from Section 2.3.1 that the FFMpeg source code used for MPEG-4 has been co-developed by a community of developers in order to support a variety of video coding standards. Consequently, its functions have been organized and modularized so as to efficiently support the many methods used in the various coding standards. This modularity is particularly indicative of industry-quality applications. And as well shall see, this trait carries distinct implications with respect to the video workload characterization.

3.3. Timing Experiments

For comparison with the OProfile sampling results, it is also desirable to examine the absolute execution times. Consequently, in our second set of experiments we used the Linux `time` command to measure the execution statistics for each of the benchmarks. These timing measurements were taken during the same round-robin experiments during which the OProfile sampling was performed.

The timing results are shown alongside the OProfile sample counts in Figure 1, back in Section 3.2. As expected from a 250 Hz timer interrupt, the sample counts and time measurements are proportional, such that the average execution time per sample is 4 ms.

A more detailed examination of the timing results across runs shows that the first of the 30 runs for each benchmark typically required greater execution times than the subsequent runs. While every precaution was taken to minimize warm start effects in memory and cache, it is only possible to completely eliminate these effects by clearing memory and cache after each run. As a result, the first run of each application had statistically greater execution time than the majority of the subsequent runs. Figure 4 shows an example of this phenomena for the JPEG-2000 encoding application. This sample was typical of the results for the majority of the applications.

The disparity between the first run and subsequent runs is predominantly due to the memory access time necessary to fetch the program and input data from the hard drive and bring it into main memory[8]. After the first run performs this memory fetch, in subsequent runs this data is often already resident in main memory, so the later runs are faster by the amount of time necessary to retrieve the video data from permanent storage.

To determine the extent to which these warm memory and cache effects impact the first run versus the subsequent execution times, we contrasted the execution time of the first run with the median execution time for each benchmark, and found the average time difference across all the benchmarks to be about

[8] In some cases though, it may be due in part to some program and data remaining in the outer levels of the cache hierarchy as well.

0.4 seconds. Because the execution time difference between the first run and subsequent runs is predominantly due to the video data already being resident in main memory, this average time difference is indicative of the time necessary to access the video data and bring it into main memory. However, the large standard deviation between the time differences indicates that this is a rough estimate at best.

3.4. Multimedia ISA Extensions

We conclude the high-level workload evaluation with a demonstration of the impact of subword parallelism on execution time. As discussed in Section 2.3.1, the FFMpeg MPEG-4 codec is the one codec in the $MB2_{video}$ benchmark suite that includes support for multimedia ISA extensions, such as Intel's MMX, SSE, and SSE2 ISA extensions [14], or Motorola's AltiVec [3]. Foremost among the instructions in these multimedia ISA extensions are subword parallelism instructions, which enable the processor to efficiently perform SIMD processing at the level of individual instructions [9]. So employing MPEG-4's ability to use the Intel MMX/SSE/SSE2 ISA extensions, we perform a final experiment examining the speedups attainable from using subword parallelism.

Within the FFMpeg MPEG-4 codec, select procedures have alternate versions that have been optimized to use the multimedia ISA extensions for specific processors. When a supported processor is used, the compiler can be directed to use the ISA extensions if desired. Based on the compile-time parameters, the compiler determines which version of a procedure is to be used, and sets up a function pointer to that procedure so that the appropriate version of the procedure is called at run time.

The execution time results from the Linux `time` command for MPEG-4 decoding were 152 ms with the MMX/SSE/SSE2 optimizations and 185 ms without, for a speedup of 1.22×. For MPEG-4 encoding, the average execution times over 30 runs were 280 ms with the MMX/SSE/SSE2 optimizations and 856 ms without, for a speedup of 3.06×.

While measuring the execution times via the Linux `time` command, we simultaneously performed OProfile profiling. As expected, the speedups as measured by OProfile are nearly identical those from the time measurements. The average sampling results for MPEG-4 encoding are 70.3 and 215.0 samples with and without MMX/SSE/SSE2, respectively, which gives a speedup of 3.06×. Likewise, the sample results for MPEG-4 decoding are 38.13 and 46.2 samples with and without MMX/SSE/SSE2, respectively, for a speedup of 1.21×.

More importantly, the OProfile results enable us to examine the speedup of individual procedures in the MPEG-4 codec. To compare the sample results for the optimized versus original (architecture-independent) versions of a procedure, we had to determine the original procedure name corresponding to each of the optimized procedures. Fortunately, this was easily accomplished by examining the compile-time assignment of the function pointers. For example, the version of the `pix_abs16_xy2` procedure optimized for MMX/SSE/SSE2 is `sad16_xy_mmx2`. Likewise, the version of `ff_jpeg_fdct_islow` optimized for MMX/SSE/SSE2 is `ff_fdct_sse2`.

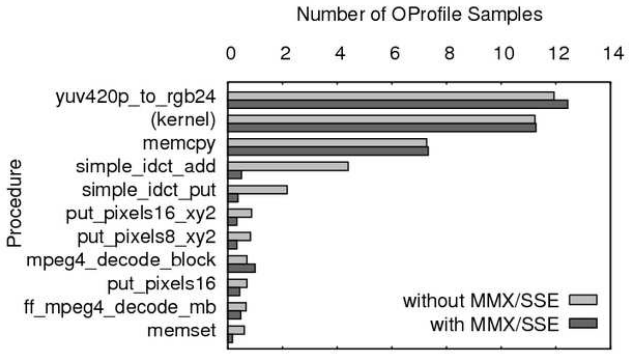

Figure 5: Execution profiles for MPEG-2 decoding with and without MMX/SSE/SSE2 optimization.

Figures 5 and 6 show the individual procedure sample results for decoding and encoding, respectively. Each of these figures lists the top 11 procedures dominating execution time in the unoptimized runs, and shows the average number of samples required for both the MMX/SSE/SSE2-optimized and original unoptimized executions. For decoding, as visible in Figure 5, the three most significant sources of execution time were unaffected as those procedure were not targeted for the ISA optimization. However, the fourth source, `simple_idct_add`, was targeted and achieved a significant speedup of 8.8×. Conversely, for encoding, the two most significant sources of execution time were indeed targeted for optimization and achieved significant speedups. The first procedure, `pix_abs16`, demonstrated a speedup of 10.0×, and the second procedure `pix_abs16_xy2`, had a speedup of 8.0×, together trimming 88.3 samples off the execution time, which is 41.1% of the original unoptimized execution time. Over all the optimized procedures, the execution times had speedups ranging from 2.7× to 10.8×[9]. Some of the original unoptimized procedures also demonstrated minor variations in execution time, but this was due to sampling error.

As evident from Figures 5 and 6, the encoder was able to achieve a much greater overall speedup than the decoder because its main sources of execution time were significantly optimized using the multimedia ISA extensions. In contrast, the decoder was only able to optimize a few of its main sources of execution time, thereby realizing a much more modest speedup overall.

In summary, the high-level workload characteristics, as measured through profiling and timing experiments, delineated the typical execution times, procedure profiles, and speedups from subword parallelism. In particular though, it highlighted two important workload differences between the *Gen-1* and *Gen-2* video standards, including an order of magnitude greater execution time, and greater complexity and modularity in second generation video applications.

[9] Actually, one procedure achieves 21.7× speedup, but this speedup is artificially inflated due to sampling error.

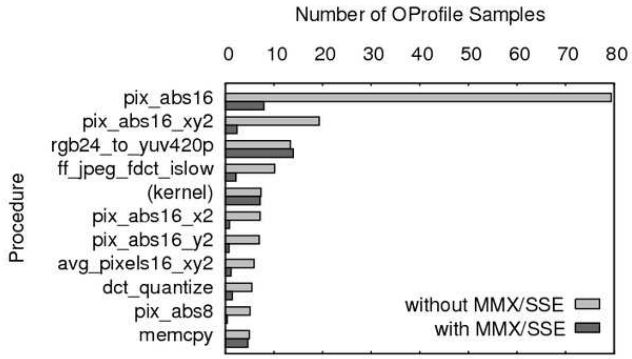

Figure 6: Execution profiles for MPEG-2 encoding with and without MMX/SSE/SSE2 optimization.

4. Low-Level Workload Evaluation

Section 3 began exploring video workloads using profiling and timing experiments on an Intel Pentium M to achieve a high-level perspective of the characteristics of video applications. This section provides a deeper understanding, performing a lower-level workload evaluation of the MediaBench II Video benchmark suite on a generic RISC architecture, using the IMPACT compilation and simulation environment [2, 19].

The purpose of this low-level workload evaluation is two-fold. First, it provides a quantitative understanding of many workload properties on a generic RISC processor, such as instruction frequencies, basic block and branch statistics, data types and sizes, memory characteristics like working set size and spatial locality, loop statistics, and instruction level parallelism (ILP). From these properties we can determine many of the desired processor architecture features for supporting video applications, 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 system engineers in selecting or designing processors for their systems. Second, from examining the low-level workload characteristics we can continue to identify the workload differences between the first and second generations of video codecs.

4.1. Evaluation Methodology

This low-level workload evaluation is performed using the IMPACT compilation and simulation environment developed at the University of Illinois at Urbana-Champaign [2, 19]. 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/EPIC and in-order superscalar architectures. We also extended the simulator environment to model an out-of-order superscalar architecture. The IMPACT compiler is a profile-based ILP compiler that 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 research and development.

The target software architecture for the IMPACT compiler is a highly-parameterizable ISA that enables the compiler to target ISAs with varying resources, including variable numbers of registers, functional units, and issue-widths. Based on the specified ISA configuration, the IMPACT compiler transforms C source code into a low-level intermediate representation called Lcode, which is effectively an assembly code representation. The Lcode representation is essentially a large, generic instruction set of simple instructions, similar to those found in most typical RISC architectures, but not biased towards any particular architecture. The primary distinction between Lcode and many RISC ISAs is that it supports *compare and branch* instructions, which compares operands and then branches accordingly, all within a single instruction. Conversely, many RISC ISAs have separate instructions for comparing and branching. Otherwise, the ISA is quite similar to a standard RISC ISA like MIPS or SPARC. The combination of IMPACT's parameterizable software architecture (ISA) and generic RISC instruction set make it perfect for an architecture-independent workload evaluation.

The base architecture model for all the low-level experiments in Sections 4 and 5, except for the ILP the experiments in Section 4.8, is a single-issue processor using IMPACT's standard generic RISC instruction set, i.e. no instructions were added to or excluded from the default ISA. The base architecture supports a large number of registers (64 integer registers and 64 floating-point registers[10]) in order to exclude spill and fill code while measuring the intrinsic application characteristics.

To accurately evaluate the intrinsic characteristics of video applications, the IMPACT compiler was set to apply only classical optimizations while compiling the $MB2_{video}$ 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 speculation are specifically disallowed as they can add or remove non-redundant instructions and can also change the size of basic blocks. Such optimizations change the characteristics of the workload. Using Lcode's generic ISA and applying only classical optimizations provides the most accurate method for measuring the 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. Then in Section 5 we will examine the variability of these characteristics across different video input data sets.

4.2. Instruction Frequencies

Defining the correct resource balance in a video processor is of critical importance. Not having enough of the necessary re-

[10]The floating-point registers are minimally used though, as demonstrated in Section 4.2

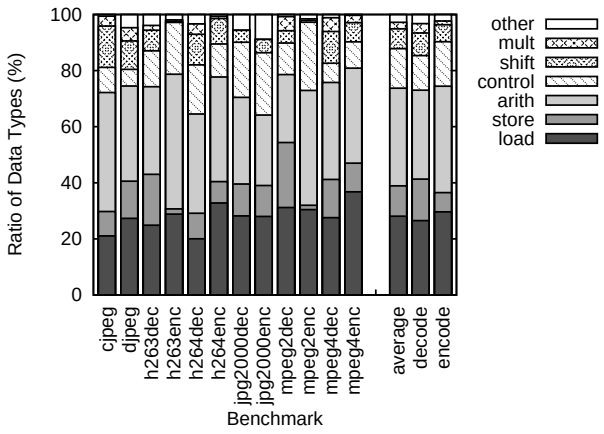

Figure 7: Dynamic instruction frequencies for the video codecs in $MB2_{video}$.

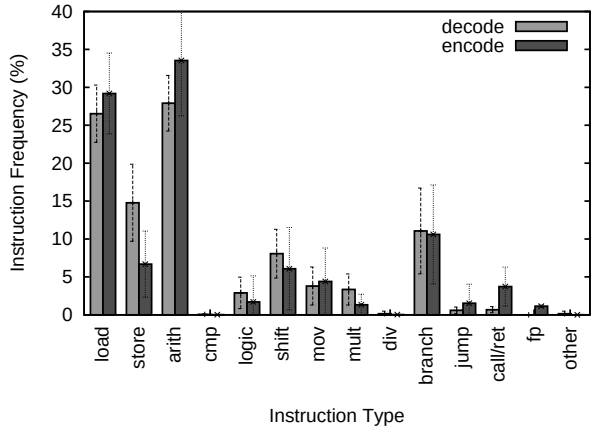

Figure 8: Comparison of the average instruction frequencies encoding and decoding.

sources 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 ILP.

Via profiling, this workload evaluation extracted the dynamic frequencies for the major classes of instructions, including integer and floating point instructions, arithmetic, relational, and logic instructions, and all control flow instructions. Figure 7 displays the average instruction frequencies for the video codecs in the $MB2_{video}$.

As evident from Figure 7, the vast majority of video instructions are either memory, 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 remainder are primarily shifts and multiplies.

Overall, the instruction frequencies for video workloads bear many similarities to general-purpose applications, as characterized by SPEC92 and SPEC CPU2000 [6, 5, 22]. There are a few significant differences though. One obvious and expected difference is video's minimal use of floating-point. While some general-purpose applications use considerable floating-point, video applications typically have minimal floating-point. The more recent video applications may use floating-point for functions such as rate-distortion optimization, but the requisite floating-point for such functions is generally small.

A notable result is the frequency of multiply instructions, which are only used about 2.5% of the time on average. While this is still $2-3\times$ the frequency of multiplies in general-purpose processors, it seems quite low in comparison with DSPs, which rely heavily on the MAC (multiply-accumulate) instruction. In fact, while the multiply instruction is used frequently in some video kernels, such as the DCT and IDCT, it is used negligibly in the more computationally intensive kernels like motion estimation. Furthermore, the use of strength reduction by the compiler converts many of the multiply (and divide) instructions by powers of 2 into shift instructions. This also accounts for the higher frequency of shift instructions in video (7% for video versus 2% for SPECint2000).

Other differences include the frequency of compares and control flow operations. The negligible frequency of compares is easily explained by IMPACT's *compare and branch* instructions, which eliminates the need for most compare (without branch) instructions. In contrast, video applications generally do have lower frequencies of control flow instructions than general-purpose applications. Control flow instructions account for less than 14% of video workloads on average, which is a little lower than the averages of 15% found in SPECint2000 [6] and 20% in SPECint92 [5]. Examining the control flow instruction frequencies in the individual video codecs, we see that MPEG-4 and JPEG in particular have especially low frequencies, accounting for only 8% of the instructions on average. This is indicative of the optimization that has been performed within these applications. All of the other applications, being primarily reference coders, have received little to no optimization. As shall be seen shortly, this has significant implications with respect to basic block sizes and potential ILP.

Further comparison of instruction frequencies for the individual codecs show that while there are no significant differences in the instruction frequencies between the first and second generation video codecs, there are some differences between encoding and decoding. Figure 8 more clearly illustrates these differences, showing the average instruction frequencies for decoding versus encoding. The primary differences arise in the frequencies of arithmetic and memory instructions. For arithmetic instructions, since encoding entails much more data processing per pixel than decoding, it is unsurprising that encoders have higher frequencies of arithmetic instructions. By a similar token the memory differences make sense as well. If you consider that decoders read in a small file containing compressed video, decompress it, and then store enormous volumes of raw video into one or more files, whereas encoders do exactly the opposite, it is not surprising that decoders have higher frequencies for loads and much lower frequencies for stores. In particular, stores occurred more than $2\times$ more frequently in decoding than encoding.

From the instruction frequencies from above, we can determine an appropriate ratio of computing resources within the datapath. Given the six major types of functional units: integer ALU (ALU), memory unit (MEM), branch unit (BR), shifter (SH), floating-point unit (FP), and multiplier (MUL), one ratio of resources could be[11]:

- $(ALU, MEM, BR, SH, MUL, FP) \Rightarrow (5, 5, 2, 1, 1, 1)$

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. If we also consider the difficultly in supporting multiple branches and more than one or two memory instructions per cycle, a better ratio would be:

- $(ALU, MEM, BR, SH, MUL, FP) \Rightarrow (3, 2, 1, 1, 1, 1)$

Finally, since the frequency of floating-point instructions is negligible, some systems may forego floating-point support altogether.

4.3. Basic Block Sizes

Another important characteristic of applications is basic block size. Information about basic block size provides an estimate of the potential instruction level parallelism (ILP) attainable by a compiler. The size of basic blocks in an application effectively defines an upper limit on the maximum amount of potential ILP within the application. Of course, achieving this maximum is highly unlikely, as it would 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 no greater than 25% to 35% of that amount. Regardless, larger basic blocks generally provide greater potential for ILP.

To measure the inherent basic block (BB) size within an application, our experiments use only classical compiler optimizations. More aggressive optimizations such as speculation and predication generally increase BB size, so such optimizations are specifically disallowed when measuring inherent BB sizes.

There are a few meaningful statistics to be considered when identifying the BB sizes of an application. If individual BB execution counts are available, then the most accurate statistics are the average, median, and standard deviation results. The median results in particular are often quite important. Since BB sizes often vary considerably within an application and generally do not follow a standard distribution, the average BB size is often inflated by a small number of very large frequently-executing BBs. In such cases, the median is effective for denoting the typical BB sizes within the application.

Lacking exact BB execution counts, BB sizes can be approximated from the dynamic frequency of control flow instructions. Since control flow instructions can only occur as the last instruction in a BB, their frequency is inversely proportional to the typical size of BBs. So for video, which was seen above to have control flow instructions ranging in frequency from 5.9% (i.e. 1 out of every 17 instructions) for the JPEG decoder to 24.3% (i.e. 1 out of every 4.1 instructions) for the MPEG-2 encoder, the approximate BB sizes will range from 17 to 4.1 instructions per BB.

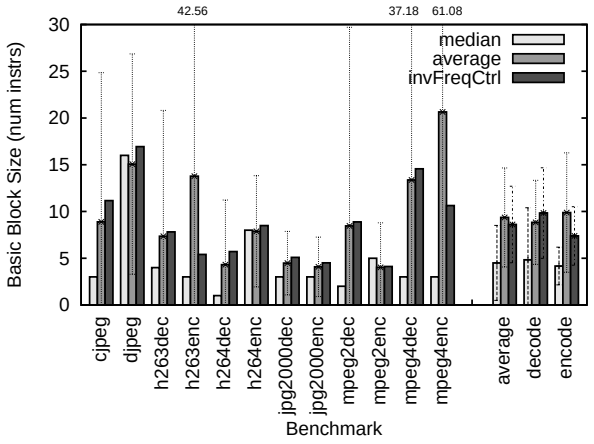

Figure 9: Basic block statistics for the benchmarks in $MB2_{video}$.

Since the IMPACT simulation environment does record the number of times each BB within an application is executed, we compute the average, median, and standard deviations for BBs, and show those results alongside the approximations based on control flow instruction frequencies in Figure 9. Consequently, three sets of BB statistics are shown for each video codec in Figure 9. The first bar, *median*, indicates the median BB size. The second bar, *average*, illustrates the average and standard deviation, and the third, *invFreqCtrl*, gives the approximation based on the dynamic frequency of control flow instructions. Overall, the video benchmarks demonstrate median, average, and *invFreqCtrl* estimated sizes of 4.5, 9.4, and 8.6 instructions per BB, respectively.

In comparison with general-purpose applications, video workloads generally have a lower frequency of control flow instructions, and so offer slightly greater potential for ILP than general-purpose applications. As measured by Hennessy and Patterson [6], five of the SPECint2000 benchmarks demonstrate control flow instruction frequencies ranging from 12.6% to 24.1%, resulting in *invFreqCtrl* BB size estimates with between 7.9 and 4.1 instructions, or 6.7 instructions per BB on average. The average *invFreqCtrl* approximation for video is 8.6 instructions per BB, so this measure indicates video workloads have BB sizes that are 28% larger on average. It will be interesting to see whether the larger BB sizes for video workloads translates into higher ILP when measuring ILP below in Section 4.8.

Examining the results of the individual video codecs, we can immediately see that in most cases the *invFreqCtrl* approximation serves as a good estimate. The H.263 encoder and MPEG-4 encoder are the major two exceptions. Both of these applications have one or more very large BBs that are executed frequently (but not frequently enough to dominate the *invFreqC-*

[11] There could also be a divider, but based on the minimal use of divide instructions, a software implementation will likely serve best.

trl estimate). The H.263 encoder has one BB with 98 instructions that accounts for over 10% of the BBs executed, while the MPEG-4 encoder has a few BBs ranging from 62 to 200+ instructions that aggregately comprise over 13% of the executed BBs. Consequently, the average is significantly inflated by these large frequently-executing BBs.

We also see that in most cases the median BB size is much smaller than the average and *invFreqCtrl* estimated sizes, with typical median BB sizes of only 3 or 4 instructions. In these cases, while there are a few very large BBs that offer significant opportunities for high ILP, the majority of the BBs are fairly small and offer minimal potential for ILP. The two major exceptions are the JPEG decoder and H.264 encoder. Both of these benchmarks have a few large BBs that account for between 45% and 85% of the dynamic BBs, thereby resulting in median BB sizes of 16 and 8 instructions. Such cases are fairly atypical, except when manual loop unrolling of key loops has been applied by the programmers, as is likely the case here.

Finally, while there are no significant differences between the characteristics of decoding versus encoding, there are some noticeable differences between the first and second generation video codecs, and between the optimized and unoptimized codecs. In first examining the optimized versus unoptimized codecs, we see that the JPEG and MPEG-4 codecs have the largest average basic blocks sizes, with 14.5 instructions per BB, versus only 6.8 instructions per BB for the unoptimized codecs. Again, the likely explanation is that is that manual loop unrolling has been applied for key loops in those applications.

Similarly, comparing the *Gen-1* and *Gen-2* codecs, we see that H.264 and JPEG-2000 generally have smaller basic blocks. As mentioned above, the MPEG-4 codec has been optimized for speed, and as such has much larger basic blocks. Consequently, if we consider H.264 and JPEG-2000 versus the four older applications, H.264 and JPEG-2000 have average BB sizes of only 5.2 instructions, whereas the four older codecs average sizes of 11.4 instructions. Or if we only consider the older two unoptimized applications (MPEG-2 and H.263), their average BB sizes are 8.4 instructions. It is evident that the two most recent applications, H.264 and JPEG-2000, are displaying the effects of considerably greater theoretic and algorithmic complexity. This entails more decision making and results in higher frequencies of control flow instructions and smaller BB sizes.

4.4. Branch Prediction

Even with an average of 9.4 instructions per BB, video applications are still limited by intra-BB instruction dependencies and will likely realize IPCs of fewer than 2 instructions per cycle (and even less on high frequency processors). Obtaining higher ILP 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. Since speculation is most effective across highly predictable branches, while predication is most effective for combining dynamic paths with unpredictable branches, both speculation and predication rely on the accuracy of static branch prediction. So, 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.

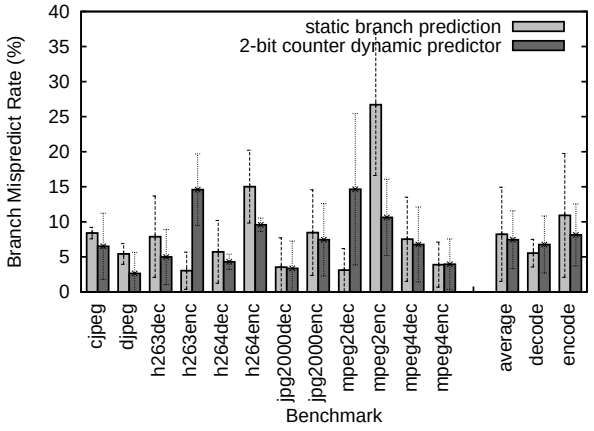

Figure 10: Misprediction rate of static branch prediction vs. a 1024-entry 2-bit counter dynamic branch predictor.

Figure 10 compares the branch prediction miss rates for profile-based static branch prediction and a simple 2-bit counter dynamic branch predictor. The results show 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 achieves 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 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 2.5× better than SPECint92's miss rate of 15% and about 40% better than SPECfp92's miss rate of 9% [6, 5]. Video's static branch prediction accuracy is even better than the accuracy of basic dynamic branch predictors in general-purpose applications. Using a 1024-entry 2-bit dynamic branch predictor, SPECint92 still only achieved a mispredict rate of 11.6% [15].

Examining the results for the individual video codecs, aside from the MPEG-4 encoder, it is clear that the decoding benchmarks achieve better prediction accuracy than the encoding benchmarks. Considering the nature of video decoding versus encoding, this is to be expected. Whereas video decoders decompress the compressed video data in a deterministic fashion, video encoding must perform a lot of decision making in the process of finding and eliminating the redundant and least important data in order to achieve quality lossy compression. Such decision making entails much more unpredictable control flow than the deterministic control flow in decoding.

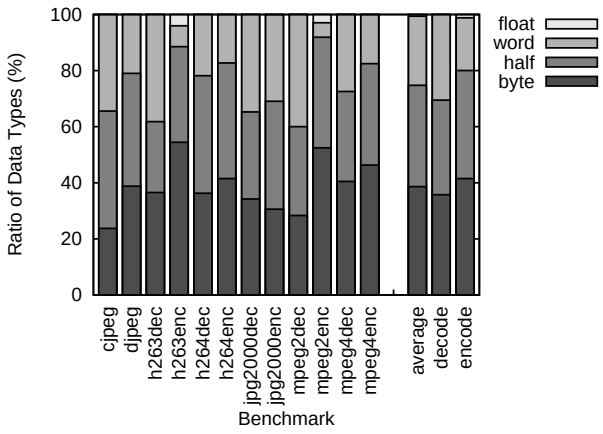

Figure 11: Ratio of data types according to video type.

Overall, video's high static branch prediction rates are indicative of the processing regularity that exists in video applications. While there are still a number of dynamic branches that are difficult to predict, these branches occur much less frequently than the predictable branches. 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 and enabling greater potential for ILP than available in general-purpose applications.

4.5. Data Types and Sizes

Data type and size is another important issue in video. As opposed to general-purpose applications, video codecs typically use small integer data types of 16 bits or less. The data type and size characteristics used in video are important because they impact the degree of parallelism achievable from subword parallelism. Because register width is fixed, smaller data sizes mean more elements can be packed into a single register, so subword parallelism is most effective for small data types.

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 simulator was then able to monitor the actual value produced by 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 effectively scaled according to the execution weight of the instructions. Instructions executing more frequently are expected to be more accurate and will contribute more to the results, while instructions executing less frequently are more prone to error, but will impact the final results minimally.

Figure 11 shows the average ratio of data types in $MB2_{video}$. 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 CPU these instructions would all use word (32-bit) data types, whereas in 64-bit CPUs they would predominantly be double-word (64-bit) data types. So we can conclude that video applications, as characterized by $MB2_{video}$, use 16-bit or smaller data types 75% of the time on average, offering ample opportunity for subword parallelism.

In comparison, general-purpose applications tend to use much larger data types. For the SPECint92 general-purpose applications running on a 32-bit architecture, Hennessy and Patterson [5] reported average frequencies of 7%, 19%, and 74% for byte, halfword, and word data sizes, respectively. They later repeated the experiment for SPECint2000 on a 64-bit architecture, and measured average frequencies of 10%, 5%, 26%, and 59% for byte, halfword, word, and double-word data sizes, respectively [6]. While these results were not obtained in the same manner as the video results, and it is clear that the data size in general-purpose applications is dominated by the data size matching the datapath width, there is still a very significant difference in the frequency of small data types between the video and general-purpose application areas; on average video applications use 16-bit or small data types as much as 75% of the time, whereas general-purpose applications use 16-bit or smaller data types as little as 20% of the time.

4.6. Memory Statistics

Since memory instructions are the primary source of processor stalls in most applications, understanding the memory characteristics is of paramount importance for any application area. 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, so 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 of video workloads involved a cache regression study over the $MB2_{video}$ benchmarks 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 L1 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 L1 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.

4.6.1. Data Working Set Size

To evaluate working set size, a cache regression was performed for all base-2 L1 data cache 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 associativities 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 corresponding to a prominent "knee" in the cache regression results (i.e. a significant drop in cache miss rate), or lacking that, as the smallest cache size that reduces the data cache miss ratio below 2.5%.

The data working set sizes for the $MB2_{video}$ benchmarks are displayed in Figure 12. 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 2-way or greater associativity, 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 MPEG-2 decoding are the only two applications that achieve poor miss rates with an 8 KB cache, requiring 128 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.

In comparison with the video results, general-purpose applications typically exhibit poorer memory performance. According to Hennessy and Patterson [6], 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 larger, and often an order of magnitude larger, than video applications to achieve comparable miss rates.

In examining the results for individual codecs, while there are no distinctive differences between first and second generation video codecs, there is a clear difference between encoding and decoding. The encoding benchmarks, particularly those that employ inter-frame redundancy elimination (i.e. motion estimation), which include H.263, MPEG-2, MPEG-4, and H.264, typically have much smaller working set sizes than their decoder counterparts. This is essentially because the working set size of the encoder is heavily influenced by the motion estimation kernels. The motion estimation kernel is easily implemented as a tight loop with high temporal and spatial locality, and since motion estimation consumes a considerable fraction

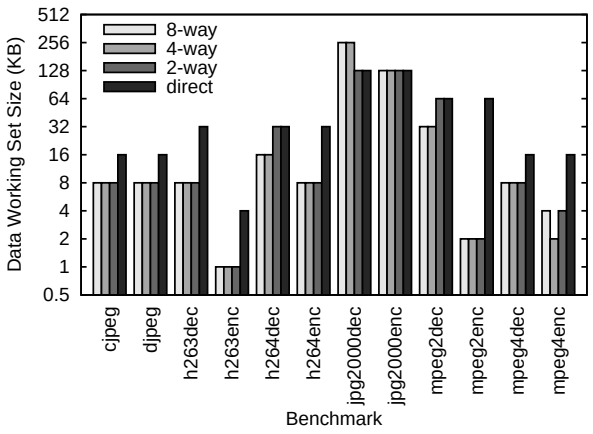

Figure 12: Data memory working set sizes for different cache associativities.

of the execution time, its working set characteristics strongly influence the overall application's working set size. In contrast, for the two encoders that do not employ motion estimation, JPEG and JPEG-2000, the working set sizes for their encoders and decoders are identical.

The only other significant difference is JPEG-2000's large data working set, which we do not believe to be truly indicative of the working set size for wavelet-based coders. By restructuring a few key loops, we expect that its data working set size could be decreased dramatically.

4.6.2. 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 also reduces 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.

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 to the case where an increase in line size by a factor of x results in 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 maximum spatial locality, the

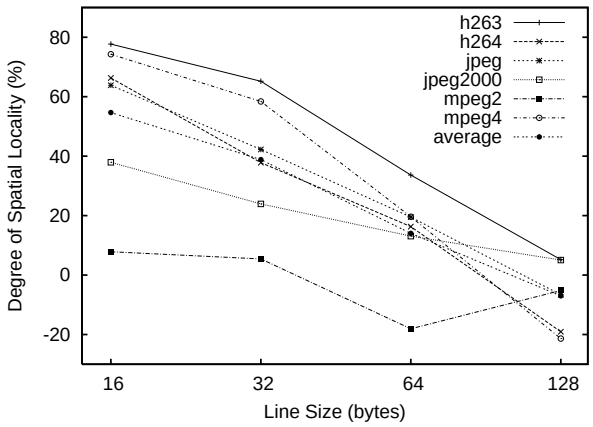

Figure 13: Data memory spatial locality for 32 KB direct-mapped cache. Indicates spatial locality between indicated line size and previous line of half the size.

degree of spatial locality linearly decreases according to the ratio of the actual decrease in miss rate versus the ideal decrease in miss rate. As a result, the spatial locality will be positive for line sizes that result in a decrease in miss rate and negative for line sizes that actually increase the miss rate.

Defining m_a as the miss rate for the longer line size, m_b as the miss rate for the shorter line, and r as 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 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 13. 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, with a 32 KB direct-mapped cache, cache line sizes of 32 or 64 bytes enable the most effective spatial locality performance for $MB2_{video}$.

In comparison, Hennessy 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 [5]. While their experiments did not quantitatively compute spatial locality, the results showed only a small performance improvement from increasing line sizes. Overall, the results indicate line sizes of 32 bytes or less are likely the best option for general-purpose applications.

4.6.3. Streaming Data

While no tests were performed to directly quantify streaming data, the cache and line size results do provide evidence that such memory prefetching support would be beneficial.

Video applications typically entail very large amounts of data, and as the instruction frequency results demonstrated, they perform frequent loads and stores. However, with the exception of JPEG-2000, the data working set sizes for these applications are not very large, while the spatial locality results are good in most cases. The large amounts of data, coupled with the small working set sizes, indicate that the processor typically loads in a small amount of data, processes it, then throws it away. Furthermore, 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. Based on this evidence of streaming data in video applications, 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. [18] and Struik et al. [16].

4.7. Loop Statistics

As found in Section 3, video applications spend the vast majority of their time executing over relatively small sections of the program code. To more fully understand the processing characteristics within these frequently executed code sections, we 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.

The first method for dynamically measuring loop statistics was presented by Kobayashi [8]. Through minor extensions of the IMPACT profiling tools, we were similarly able to extract loop statistics using the IMPACT compilation/simulation environment.

4.7.1. Loop Level Execution Weight

While static branch prediction provides one indicator for processing regularity, another 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, whereas 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.

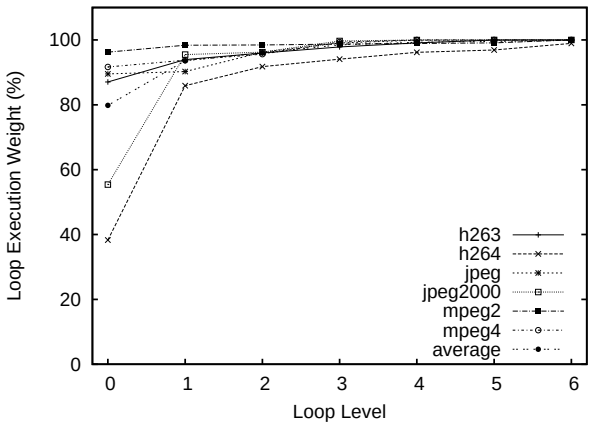

Figure 14: Percentage of loop execution weight by loop level.

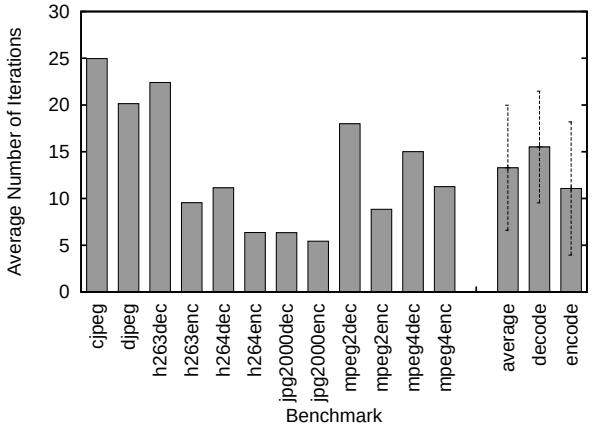

Figure 15: Average number of iterations per loop invocation for each benchmark.

To quantify loop behavior, we 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. Note, our loop level numbering scheme is reversed from the system used by Kobayashi [8] and de Alba and Kaeli [1], both of whom use breadth-first loop level numbering.

The results, presented in Figure 14, 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 nesting level below the innermost loop, 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.

4.7.2. 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 measure this 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 then 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 15. These results indicate that typical loops have a large number of iterations, with 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 separate input data sets demonstrated variations that were within 5% of the results in Figure 15 in most cases.

In comparison with general-purpose applications, video workloads demonstrate much higher average numbers of iterations per loop. As characterized by de Alba and Kaeli [1], the average number of loop iterations in six of the benchmarks from SPECint2000 range from 3.4 to 8.9 iterations per loop invocation. The average over all six benchmarks is only 5.4 iterations per loop invocation, which is more than 2× smaller than the average iterations per loop invocation found in video.

Examination of the results for individual codecs indicates 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 dynamic paths for this decision making usually entail 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.

Finally, the results also show significant differences between the first and second generation video codecs. The *Gen-1* codecs average 17.3 iterations per loop invocation, whereas the *Gen-2* codecs average only 9.3 iterations per invocation. Even worse, since MPEG-4's loop statistics are more indicative of the *Gen-1* codecs, if we instead compare the two most recent codecs, JPEG-2000 and H.264, versus the four older applications, the average number of loop iterations per invocation drops to only

7.3 iterations per invocation, versus 16.3 for the older codecs. Further evidence of the difference between these two codecs and the older codecs is also evident in Figure 14, which shows that they both spend much less of their time in the innermost loops than the other four codecs.

The smaller numbers of iteration per invocation and lower innermost loop execution times are once again explained by the growing complexity of the emerging video coding standards. JPEG-2000 and H.264 both employ a number of new technologies designed to enable greater compression, but at the expense of more control code for making decisions, which decreases both the number of iterations per loop and the time spent in the innermost loops. Context-based adaptive entropy coding, which makes coding decisions based on the correlations between neighboring pixels, is particularly control intensive.

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.

4.8. 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.

Note, in the results of all the ILP experiments presented below, no significant differences were found in the workload characteristics between either the first and second generation codecs, or encoders versus decoders.

4.8.1. Static ILP

In measuring static ILP in video applications, we compile the source 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 from 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 level, superscalar optimization, performs speculation via the *superblock* optimization [7] and other optimizations such as loop unrolling. The final compilation also performs predication (a.k.a. conditional execution) via the *hyperblock* optimization [13].

Performance evaluation of the three levels of compilation is performed on a generic 4-issue[12] VLIW processor. This processor is a simple statically-scheduled VLIW (Very-Long Instruction Word) architecture that supports 3 integer ALUs, 2 memory 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, 1), as determined in section 4.2. The instruction latencies of these functional units are assumed to be one cycle for integer ALU instructions, two cycles for loads, three 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 media and video processors. The branch architecture of the processor uses only static branch prediction, and the memory hierarchy of the processor has a 16 KB L1 instruction cache, a 32 KB direct-mapped L1 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 ISA once again supports a register file with 64 integer registers and 64 floating-point registers.

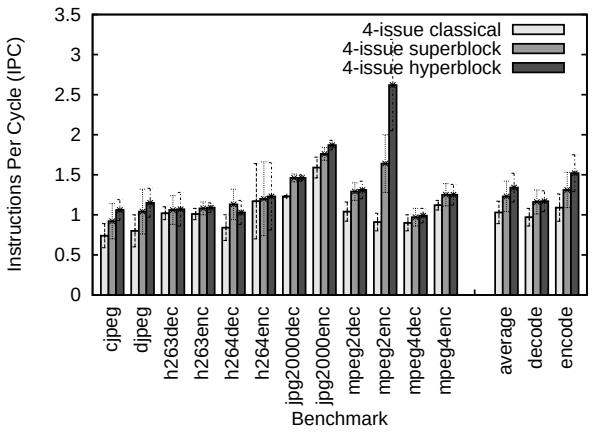

Figure 16: IPCs of a 4-issue architecture for various compilation methods.

Figure 16 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 $MB2_{video}$ applications.

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 instructions 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 frequently sequentially dependent upon each other. So even though video codecs do not have as much control code as general-purpose applications, video offers only slightly more ILP than general-purpose applications since there are longer in-

[12] Our previous research found that media benchmarks achieve little benefit from more than 4 issue slots for static scheduling [4], even with aggressive ILP compiler scheduling.

struction dependence chains.

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 to the parallelism that exists between independent pixels, blocks of pixels, pictures/frames, video sequences, and so on. Unfortunately, this parallelism is a coarser granularity of parallelism than ILP. There are a variety of means for taking advantage of data parallelism, including process-, thread-, and subword-level parallelism, as demonstrated by ALPBench, the *All Levels of Parallelism* benchmark developed by Li et al. [11], and as illustrated for subword parallelism on the MPEG-4 codec in Section 3.4. Unfortunately, compilers are generally ineffectual at targeting these coarser levels of parallelism, so programmers must explicitly parallelize the programs using the appropriate parallel programming methods.

4.8.2. Static vs. Dynamic ILP

A second experiment was performed to demonstrate the effectiveness of dynamic scheduling for video processing. The $MB2_{video}$ benchmarks were run on three different ILP processor architectures 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.

Figure 17 shows 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 superscalar processor enables 66% better performance on average than a VLIW processor for the 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 preferable.

While oracle experiments (which examine parallelism using infinite resources and perfect branch prediction) have demonstrated considerable parallelism in video applications [12], 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 pursue alternate avenues for parallelism.

In summary, this section examined the low-level characteristics of video workloads as they would appear on a generic RISC ISA. We evaluated instruction frequencies, basic block sizes, branch prediction, data types and sizes, memory statistics, loop

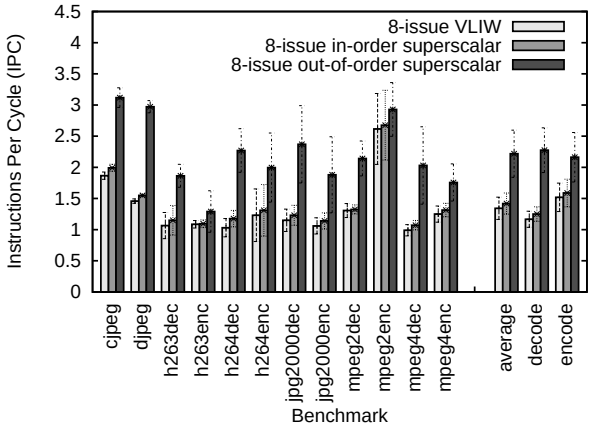

Figure 17: IPCs of various 8-issue architecture models.

characteristics, and static and dynamic ILP. For video workloads on the whole, it was found that video applications have strong processing regularity with high static branch predictability and large average numbers of loop iterations per loop invocation. The memory statistics found good memory performance and spatial locality with reasonably small caches sizes, as well as indirect evidence of streaming data. Unfortunately, the basic block sizes were found to be only slightly larger than in general-purpose workloads, and similarly the ILP experiments found little more ILP from video workloads than commonly found in general-purpose workloads.

More specifically, we also found some significant differences between the first and second generations of video codecs, as well as between optimized and unoptimized codecs. In comparing the optimized codecs, MPEG-4 and JPEG, with the four unoptimized codecs, the optimized applications demonstrated much larger basic block sizes than codecs. For the JPEG codec, this translated into greater ILP results than the other applications, but the MPEG-4 codec must contain long dependence chains in its basic blocks, because it did not realize ILPs any greater than the unoptimized codecs.

The differences between the first and second generation video codecs were most apparent in the two most recent codecs, JPEG-2000 and H.264. Compared with the older codecs, JPEG-2000 and H.264 demonstrated the effects of the growing algorithmic and theoretic complexity of their standards in many ways, including smaller basic blocks, more frequent control instructions, potentially larger working set sizes, and in particular, much less idealistic loop characteristics, with fewer iterations per loop invocation, and much less time spent in the innermost loops.

5. Workload Variability across Inputs

With any workload, it is important to understand how the workload characteristics will vary across different input data sets and execution parameters. For video, the most common variations across inputs are due to frame size, bit rate, degree of motion, and search window size (for encoding). Aside from

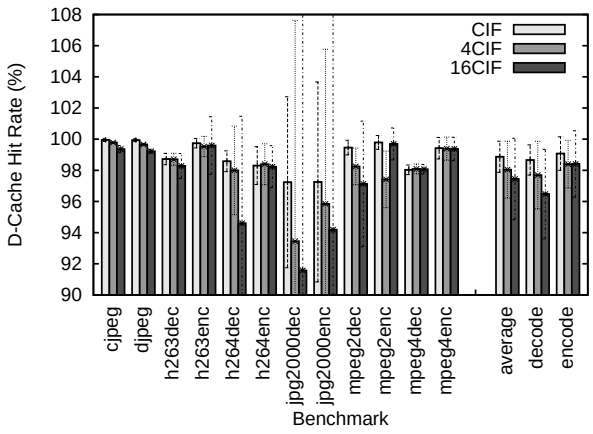

Figure 18: Data cache hit rates for different video frame sizes.

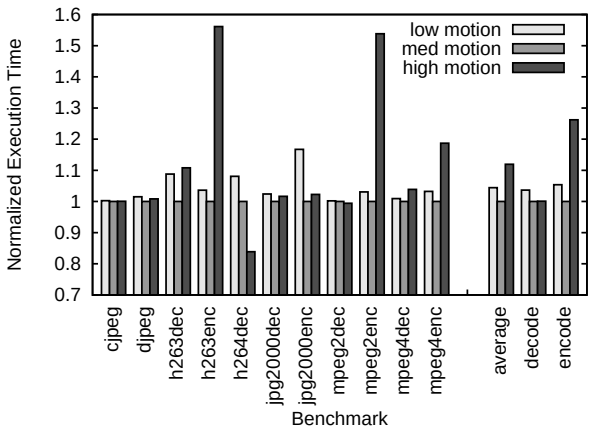

Figure 19: Comparison of execution times for varying degrees of motion in the video.

e input variations, the primary other differences in application characteristics arise from the different algorithms and methods that the software designers use in implementing the applications. Fortunately, there are a variety of implementations in the benchmarks in MediaBench II Video, so $MB2_{video}$ demonstrates such variations effectively.

We first experimented with variations in frame size. Using video sequences with frame sizes of CIF (352x288 pixels) and 16CIF (1408x1152 pixels), their low-level workload characteristics were compared with the base 4CIF (704x576 pixels) video input data set. Aside from the anticipated differences in execution time, little variation was found across the various processing characteristics. The execution time increased nearly linearly with the number of pixels, such that the dynamic instruction count increased by an average of 4.24× with each doubling of frame height and width (i.e. quadrupling frame size). H.264 decoding was the one exception, with its dynamic instruction count increasing by 6.8× from doubling the frame size from 4CIF to 16CIF. Each doubling of the frame size also resulted in decreases in L1 data cache hit rates, as illustrated in Figure 18. These reductions were particularly true for JPEG-2000, and H.264 and MPEG-2 decoding.

Similarly, we examined the effect of varying the target bit rate. The base video sequence targets a moderate bit rate, so two alternate execution parameterizations were used, which corresponded to half and double the target bit rate of the base sequence. While each doubling of the target bit rate served to increase the dynamic instruction count by an average of 11% for decoding and 3% for encoding, the impact was actually minimal in all the benchmarks except H.263 and MPEG-4 decoding, which had increases of 20-30% from each doubling of the bit rate. Doubling the bit rate also resulted in increases in L1 data cache hit rates of 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. As shown in Figure 19, 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 workload characteristics were significantly affected.

Finally, an experiment was performed to vary the search window size and determine its impact on 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 from +/-16 to +/-8. Only the MPEG-2 and H.263 encoders could be included in this experiment since the version of H.264 used did not yet support different search window sizes, and the FFMpeg MPEG-4 codec uses the EPZS motion estimation algorithm, which dynamically varies search window size.

As show in Figure 20, both MPEG-2 and H.263 demonstrated similar results for variations in search window size. As expected, for the decoding 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 2.6×, and also reduced the L1 data cache miss rate by 50% over the +/-16 search window miss rate. The increase in execution time and increase in cache hit rate are to be expected from increasing the search window size. 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

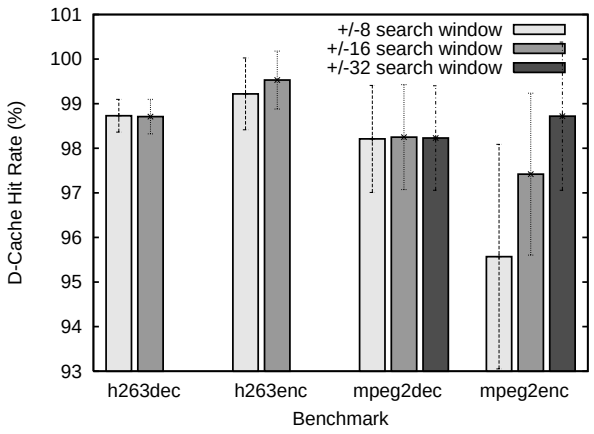

Figure 20: Data cache hit rates for different search window sizes.

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, common variations across input data sets and execution parameters had little effect on the workload characteristics.

6. Conclusions

This paper presents $MB2_{video}$, the MediaBench II 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. To maintain and improve upon the mission initially originally set forth by MediaBench, we recently began developing MediaBench II, an updated set of benchmarks for multimedia. Among the goals for MediaBench II is the development of both a composite benchmark, similar in scope to the original MediaBench benchmark, as well as a set of area-specific benchmarks for each media type. While currently in the early stages of design, the audio and video benchmarks have been completed. The video benchmark, $MB2_{video}$, contains both the first generation video codecs, such as H.263, JPEG, and MPEG-2, as well as the recent and emerging second generation video standards, including MPEG-4, JPEG-2000, and H.264.

The paper presents the results of a comprehensive workload evaluation of the new MediaBench II Video benchmark suite, including both a high-level workload characterization on an Intel Pentium M processor, and a low-level workload evaluation on a generic RISC ISA, as modeled by the IMPACT compilation/simulation environment. The workload evaluation covers the high-level characteristics of execution time and procedure profiles, as well as a variety of lower instruction-level features, including instruction frequencies, basic block sizes, branch prediction rates, data sizes, working set sizes, spatial locality, loop characteristics, and ILP performance. Finally, the paper demonstrates that common variations in the execution parameters and input data set, such as bit rate, frame size, degree of motion, and search window size, have little effect on the workload characteristics, beyond the expected impact on execution time and cache miss rates.

Most importantly, the workload evaluation demonstrates that while video workloads have high processing regularity as compared with general-purpose applications, the growing algorithmic and theoretic complexity in the emerging video standards, such as JPEG-2000 and H.264, are negatively impacting the characteristics of video workloads. The most recent codecs employ a number of new methods such as rate-distortion optimization, context-adaptive entropy coding, and intra-block prediction, that are designed to maximize compression while minimizing degradation. Unfortunately, in addition to the longer encoding and decoding times, these applications are (1) becoming larger and spending more of their time across a wider array of procedures, and (2) becoming less regular and predictable. In particular with respect to processing regularity, the complex analysis and decision making inherent in many of the newer technologies is making the code more control intensive and reducing the potential for ILP. Likewise, the emerging applications are spending less time in the innermost loops and are executing fewer loop iterations per loop invocation. However, as demonstrated by the more optimized codecs, MPEG-4 and JPEG, optimizing code can have significant beneficial impacts on the workload characteristics, which can alleviate some of the negative effects arising with the newer coding technologies.

References

[1] M. de Alba, D. Kaeli, Runtime Predictability of Loops, in: Proceedings of the 4th IEEE Workshop on Workload Characterization, 91–98, 2001.

[2] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Warter, W.-M. Hwu, IMPACT: an architectural framework for multiple-instruction-issue processors, in: Proceedings of the 18th Annual International Symposium on Computer Architecture, 266–275, 1991.

[3] K. Diefendorff, P.K. Dubey, R. Hochsprung, H. Scales, AltiVec Extension to PowerPC Accelerates Media Processing, IEEE Micro 20 (2) (2000) 85–95.

[4] J.E. Fritts, Architecture and Compiler Design Issues in Programmable Media Processors, Ph.D. Thesis, Princeton University, 2000.

[5] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, second ed., Morgan Kaufmann Publishers, 1996.

[6] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, fourth ed., Morgan Kaufmann Publishers, 2006.

[7] W.-M. 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, D.M. Lavery, The Superblock: An Effective Technique for VLIW and Superscalar Compilation, Journal of Supercomputing 7 (1) (1993) 229–248.

[8] M. Kobayashi, Dynamic characteristics of loops, IEEE Transactions on Computers 33 (2) (1984) 125–132.

[9] R.B. Lee, Accelerating Multimedia with Enhanced Microprocessors, IEEE Micro 15 (2) (1995) 22–32.

[10] C. Lee, M. Potkonjak, W.H. Mangione-Smith, MediaBench: A Tool for Evaluating and Synthesizing Video Communications Systems, in: Proceedings of the 30th Annual International Symposium on Microarchitecture, 330–335, 1997.

[11] M.-L. Li, R. Sasanka, S.V. Adve, Y.-K. Chen, E. Debes, The ALPBench Benchmark Suite for Complex Multimedia Applications, in: Proceedings

of the IEEE International Symposium on Workload Characterization, 34–45, 2005.
[12] H. Liao, A. Wolfe, Avaliable Parallelism in Video Applications, in: Proceedings of the 30th Annual International Symposium on Microarchitecture, 321–329, 1997.
[13] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, R.A. Bringmann, Effective Compiler Support for Predicated Execution Using the Hyperblock, in: Proceedings of the 25th International Symposium on Microarchitecture, 45–54, 1992.
[14] A. Peleg, U. Weiser, MMX Technology Extension to the Intel Architecture, IEEE Micro 16 (4) (1996) 42–50.
[15] D. Pnevmatikatos, G. Sohi, Guarded Execution and Branch Prediction in Dynamic ILP Processors, in: Proceedings of the 21st International Symposium on Computer Architecture, 120–129, 1994.
[16] P. Struik, P. van der Wolf, A.D. Pimentel, A Combined Hardware/Software Solution for Stream Prefetching in Multimedia Applications, in: Proceedings of SPIE Photonics West, Multimedia Hardware Architecture, 120–130, 1998.
[17] A.M. Tourapis, Enhanced Predictive Zonal Search for Single and Multiple Frame Motion Estimation, in: Proceedings of Visual Communications and Image Processing 2002, 1069–1079, 1998.
[18] D.F. Zucker, R.B. Lee, M.J. Flynn, Hardware and Software Cache Prefetching Techniques for MPEG Benchmarks, IEEE Transactions on Circuits and Systems for Video Technology 10 (5) (2000) 782–796.
[19] IMPACT compiler research group, http://www.crhc.uiuc.edu/Impact/
[20] MediaBench II, http://euler.slu.edu/ fritts/mediabench/
[21] OProfile, http://oprofile.sourceforge.net/
[22] Standard Performance Evaluation Corporation (SPEC), http://www.spec.org/