Instruction fetch characteristics of media processing

Jason Fritts[*a] and Wayne Wolf[b]

[a]Dept. of Computer Science, Washington University, St. Louis, MO 63130
[b]Dept. of Electrical Engineering, Princeton University, Princeton, NJ 08544

ABSTRACT

This paper presents the results of a quantitative evaluation of the instruction fetch characteristics for media processing. It is commonly known that multimedia applications typically exhibit a significant degree of processing regularity. Prior studies have examined this processing regularity and qualitatively noted that in contrast with general-purpose applications, which tend to retain their data on-chip and stream program instructions in from off-chip, media processing applications are exactly the opposite, retaining their instruction code on-chip and commonly streaming data in from off-chip. This study expounds on this prior work and quantitatively validates their conclusions, while also providing recommendations on architectural methods that can enable more effective and affordable support for instruction fetching in media processing.

Keywords: media processing, instruction fetch, workload characterization, performance analysis, architecture evaluation

1. INTRODUCTION

With the success of the Internet and World Wide Web, and the growing feasibility of image/video compression and computer graphics, the multimedia industry has been growing at a tremendous rate. Multimedia now defines a significant portion of the computing market, and this is expected to grow considerably. As a consequence, the processing demands for multimedia applications are rapidly escalating as users desire new and better applications. Many multimedia applications are already beyond the limits of today's microprocessors, and the next generation of multimedia promises a wider range of applications and considerably greater processing demands.

As the workloads in many computing markets continue to shift more towards multimedia, computer architecture is constantly evolving to better meet the needs media processing applications. In 1997, Diefendorff and Dubey first anticipated this process[1]. However, they predicted this shift solely with respect to general-purpose processors, indicating their belief that general-purpose processors would continue to provide improved support for media applications, and would eventually win out over specialized DSPs in becoming the dominant means for supporting multimedia. Their prediction has been largely true, with two exceptions. The first is that the shift has occurred in both general-purpose processors and DSPs. Both processor architectures have come to provide better support (typically using similar methods) for multimedia. The second exception is that specialized media DSPs (as opposed to multi-purpose DSPs) are still important for media processing, primarily because specialized application-specific processors better target those applications and markets where low power and/or low cost is a critical constraint.

One of the many aspects that Diefendorff et al. discussed is that media applications tend to exhibit significant processing regularity, often consisting of small loops or kernels that dominate the overall processing time[1]. This argument was expounded in a more recent article by Stokes[2] discussing the design of the Emotion Engine[3], the processor for Sony's Playstation 2, who likened the memory access characteristics of media applications to be in direct contrast with the memory access patterns of general-purpose applications. He indicated that general-purpose applications typically retain the majority of their data in on-chip data cache/memory and commonly stream program instructions in from off-chip, whereas media processors stream much of their data memory on and off chip and retain their instruction code in on-chip instruction memory/cache. So, whereas general-purpose applications are characterized by streaming instruction memory

[*] Correspondence: Email: jefritts@cs.wustl.edu; WWW: http://www.ccrc.wustl.edu/~jefritts; phone: (314) 935-4963

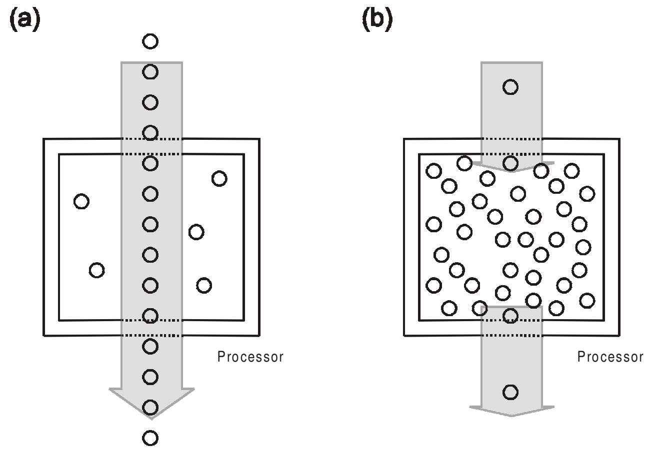

Figure 1 – Comparison of streaming vs. non-streaming memory patterns[2].

and static data memory, whose properties are illustrated in Figure 1a Figure 1b, respectively, media applications are characterized by static instruction memory and streaming data memory, which are conversely illustrated by Figure 1b and Figure 1a, respectively.

This study builds on the prior qualitative discussions by Diefendorff et al.[1] and Stokes[2], quantitatively validating their conclusions and showing the relatively idealistic nature of instruction fetch characteristics in media processing. Additionally, we also provide recommendations for architectural methods that can enable more effective and affordable support for instruction memory access in media processing.

This evaluation first examines the instruction fetch problem from an architecture-independent perspective, evaluating workload-dependent characteristics that are common to any generic RISC-style processor. We have found that many of the instruction fetch characteristics, including various characteristics of loops, program control, and instruction memory, are common among media applications. Based on the commonality of these characteristics, we conclude that multimedia applications have relatively idealistic fetch characteristics: the most critical program code sections tend to fit easily in most on-chip memory, and the control characteristics tend to be fairly predictable.

The second half of the evaluation examines various architecture-dependent aspects of instruction fetch for media processing. In existing media processors, the architectures tend to rely on static architecture support for instruction fetching. While this is reasonable based on the processing regularity and predictability, we have found some relatively minor dynamic architecture features can be added that significantly improve instruction fetch performance. This paper evaluates the performance of various dynamic architecture methods for media processing, identifying the utility of each method. In particular, we found that a small dynamic branch predictor can substantially enhance instruction prediction accuracy. And surprisingly, we found instruction buffers to be of little benefit in improving execution performance.

The remainder of this paper is organized as follows. Section 2 describes the evaluation environment, including the media benchmark and the compilation/simulation environment used for this study. Section 3 then evaluates the architecture-independent, workload-dependent characteristics of instruction memory in media applications. Section 4 continues our evaluation of instruction fetch characteristics, examining various architecture-dependent instruction fetch mechanisms, including a variety of dynamic architecture structures. Section 5 then closes with the conclusions.

2. EVALUATION ENVIRONMENT

This multimedia workload evaluation uses the MediaBench benchmark suite, augmented with H.263 and MPEG-4 to make it more representative of current and future multimedia systems. The experimental evaluation of application characteristics is performed using the IMPACT compilation/simulation tools. Use of this evaluation environment enables us to study both the architecture-dependent and architecture-independent characteristics of media applications.

2.1. MediaBench benchmark suite

The MediaBench benchmark, introduced by Lee, Potkonjak, and Mangione-Smith[4,5], is the first combination of multimedia applications to truly represent the overall multimedia industry. The benchmark was designed specifically to focus on portable applications written in a high-level language that are representative of the workload of emerging multimedia and communications systems. It incorporates multimedia applications written in C, ranging from image and video processing, to audio and speech compression, and even encryption and computer graphics.

This benchmark provides applications covering six major areas of media processing: video, graphics, audio, images, speech, and security. We also augmented MediaBench with additional video applications, MPEG-4 and H.263. MediaBench already contains MPEG-2, but H.263 and MPEG-4 are distinct in that H.263 targets very low bit-rate video, and MPEG-4 uses an object-based representation. These applications are believed to make MediaBench more representative of emerging and future media applications.

Finally, shown below in Table 1 are some statistics about the applications in MediaBench. The first column indicates the number of static instructions in the assembly code (Lcode) when compiled onto a single-issue processor using classical-only optimizations. The last four columns provide statistics for the two input data sets provided with MediaBench. For each input, the first column indicates the input data file size, while the second indicates the number of dynamic instructions for executing the application with that data set. Of the two data sets, we use input 2 (which typically has the larger dynamic traces) as the training data set for profiling. The benefit of this is that simulation is significantly more time consuming than profiling, so simulating with the smaller input data enables more reasonable simulation time. Additional information and analysis on MediaBench can be found in Fritts[6].

Program	*# Static Instrs*	*Input 1*		*Input 2*	
		File Size	*# Dynamic Instrs*	*File Size*	*# Dynamic Instrs*
cjpeg	15,883	101,484	10,845,202	968,046	86,499,438
djpeg	19,397	5756	3,032,372	31,074	25,080,826
epic	4737	65,595	38,577,696	65,610	39,366,149
gs	226,348	78,519	68,400,275	488,795	70,201,200
g721dec	1826	73,760	226,268,757	116,798	348,661,608
g721enc	1817	295,040	239,657,034	467,192	379,687,769
gsmdecode	11,365	30,426	60,794,843	48,180	96,223,492
gsmencode	11,077	295,040	135,010,575	467,192	216,712,453
h263dec	8721	20,364	60,291,153	19,338	66,026,217
h263enc	17,750	1,438,272	1,498,418,843	5,702,400	1,557,457,063
mipmap	116,668	-	20,479,889	-	-
mpeg2dec	9520	34,906	99,657,160	1,593,409	723,990,481
mpeg2enc	14,136	506,880	992,775,122	1,555,200	805,664,583
mpeg4dec	108,273	39,213	1,432,505,735	503,060	505,360,524
osdemo	112,605	-	6,272,117	-	-
pegwitdec	8576	91,537	14,516,495	53,665	12,569,444
pegwitenc	8522	91,503	24,883,764	53,631	21,830,683
pgpdecode	75,756	20,167	497,765,011	11,905	494,735,842
rasta	55,547	17,024	8,590,189	33,024	6,475,968
rawcaudio	220	295,040	5,760,303	467,192	9,388,421
rawdaudio	211	73,760	5,091,778	116,798	8,275,859
texgen	115,316	-	54,692,716	-	-
unepic	3767	7432	5,273,016	10,129	5,743,473

Table 1 – MediaBench input data set characteristics**.**

2.2. IMPACT compiler

The compilation and simulation tools for this evaluation were provided by the IMPACT compiler, produced by Wen-mei Hwu's group at the University of Illinois at Urbana-Champaign[7,8]. The IMPACT environment includes a trace-driven simulator and an ILP compiler. The simulator enables both statistical and cycle-accurate trace-driven simulation of a variety of parameterizable architecture models, including both VLIW and in-order superscalar datapaths. We have also expanded the simulator to simulate an out-of-order superscalar datapath as well. The IMPACT compiler supports many aggressive compiler optimizations including procedure inlining, loop unrolling, speculation, and predication. IMPACT organizes its optimizations into three levels:

- *Classical* – performs only traditional local optimizations
- *Superscalar* – adds procedure inlining, loop unrolling, and speculative execution
- *Hyperblock* – adds predication (conditional execution)

The impact of each of these optimization levels on instruction fetch characteristics is described later in this paper.

For purposes of this workload study, it is desirable to evaluate both the architecture-independent application characteristics as well as the architecture-dependent characteristics. The IMPACT compiler enables an architecture-independent analysis through their low-level intermediate representation, Lcode. The Lcode representation is essentially a large, generic instruction set of simple operations like those found on typical RISC architectures, but not biased towards any particular architecture. Such a generic instruction set enables architecture-independent evaluation.

3. WORKLOAD-DEPENDENT INSTRUCTION FETCH CHARACTERISTICS

To accurately evaluate the intrinsic instruction fetch characteristics of multimedia applications, the compiler was set to apply only classical optimizations while compiling the MediaBench benchmark suite. Using just classical optimizations utilizes 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, and global scheduling are specifically disallowed as they change the characteristics of the workload, allowing potentially significant variations to code size and control flow. Using a generic instruction set architecture and classical-only compilation provides the most accurate method for performing an architecture-independent workload evaluation.

This architecture-independent workload evaluation examines those application characteristics pertinent to instruction memory. These aspects include branch statistics, instruction memory working set size, instruction memory spatial locality, and loop characteristics. Each of these is evaluated in turn below.

3.1. Branch Statistics

This section examines the information on branch statistics such as frequency of branches and static branch prediction performance[†]. This information is important to ascertain first because it provides an indication of the importance of instruction fetch characteristics to the overall execution time. For example, in general-purpose applications, the frequency of branches tends to be quite high, so that all basic blocks (a basic block is a group of operations that all reside on the same path of control flow) tend to be quite small, averaging around 5 operations per basic block[10]. This, coupled with the fact that static branch prediction performance is usually fairly low (average of only about 70-80% hit rate) and working set sizes are often quite large, indicates that instruction fetch characteristics of general-purpose applications can affect execution performance significantly.

For media processing, using the MediaBench benchmark suite, it was found that on average about 20% of operations are branch operations. Similarly, it was found that the average basic block size is 5.5 operations per basic block. Overall, since the frequency of branches in media applications is similar to that for general-purpose applications, the impact of instruction fetch may potentially have significant effect upon execution performance.

[†] These results were first presented in Fritts et al.[9], and more thorough coverage appears in Fritts[6].

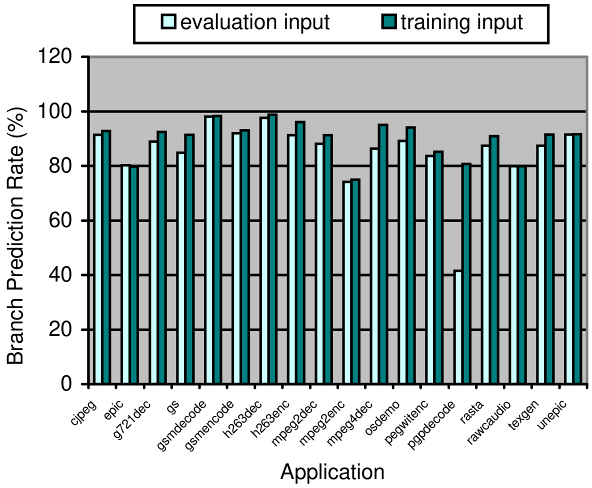

Figure 2 – Static branch prediction performance.

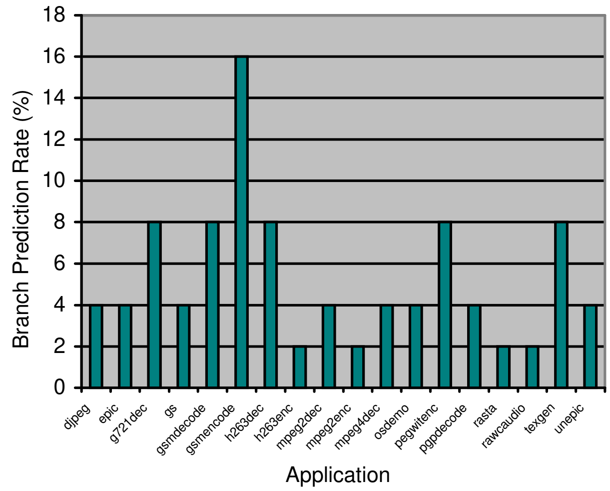

Figure 3 – Instruction memory working set size.

Unlike general-purpose applications however, the static branch prediction performance for media applications was found to be fairly high. In evaluating static branch prediction performance, we simulated static branch prediction for both the training input and a separate evaluation input. Because the code is optimized using the training input, simulation on the training input yields optimal static branch prediction performance for that input. Other input sets will typically yield lower performance. The performance of the evaluation input data set represents the realistic static branch prediction performance, which will typically, but not always, be lower than the performance of the training input.

The results of static branch prediction on the training and evaluation inputs are shown in Figure 2. Overall the performance was very good. The average static branch prediction accuracy is 89.5% with the training input and 85.9% for the evaluation input. For the most part, the realistic performance, as represented by the evaluation input, is only moderately less than the ideal performance. In comparison with general-purpose applications, the branch misprediction rate is around 2-3x lower. Consequently, the higher static branch prediction rate indicates dynamic branch prediction is less important for media applications than for general-purpose applications.

3.2. Working Set Size

The instruction memory working set size is displayed in Figure 3[‡]. To evaluate working set size, a cache regression was performed using a direct-mapped cache with a line size of 64 bytes for all base-2 cache sizes between 1 KB and 4 MB. The number of read and write misses were measured and an analysis of the results yielded the working set size for each application. The working set size is defined by the "knee" on the cache regression graph where the miss rate decreased dramatically with respect to smaller cache sizes. An example of such a knee is illustrated by the 8KB cache in Figure 4. In the absence of a knee, the working set size is defined as the cache size that reduces the miss ratio to below 3%.

Based on the statistics, it appears that instruction cache sizes can be quite small for MediaBench applications. As shown in Figures 4, a cache size of 8 KB provides the ideal cache size for these applications, with an overall miss ratio of only 0.3%. Cache sizes smaller than 8 KB increase miss rates significantly, while larger instruction caches increase performance only marginally. Even the one application, *gsmencode*, with a larger working set size than 8 KB still has a miss rate of only 1.5% at 8KB. These results are somewhat surprising since many of the applications have relatively large code sizes, shown previously in Table 1, with between 10,000 and 120,000 dynamic instructions. This means all applications spend the majority of their processing time within only a small fraction of the code, in some cases less than 3%. The small instruction working set size provides another indication of the processing regularity in media processing.

[‡] Once again, these results were first presented in Fritts et al.[9], and more thorough coverage appears in Fritts[6].

It is also important to note that the IMPACT compilation/simulation environment assumes that each instruction is encoded using a 32-bit format. However, on processors that utilize shorter formats, such as Tensilica's Xtensa processor[11], the code size, and similarly the instruction set working size, may be substantially smaller.

In contrast with the multimedia results, general-purpose applications typically have much larger instruction working sets. To achieve the same miss ratios as multimedia applications, general-purpose applications would require a 32 KB instruction cache[10]. Consequently, media applications demonstrate their tendency to spend the majority of their processing time in very small fractions of the program code.

3.3. Spatial Locality

Spatial locality is another important instruction fetch characteristic. If an application has high spatial locality, then there is less likelihood that a given instruction memory access will result in a cache miss. To evaluate the spatial locality for instruction memory, we performed a cache line size regression on a 64 KB direct-mapped cache for all base-2 line sizes from 8 to 1024 bytes. As line size increases, performance typically increases because the processor will often use the additional data contained within the cache line without having to generate additional cache misses. The degree to which an application can use the additional memory within longer cache line sizes represents its degree of spatial locality.

An equation was defined to quantitatively describe the spatial locality for increasing line size. 100% spatial locality is represented by a perfect decrease in cache misses relative to the change in line size. So if line size doubles, the number of cache misses would halve. The degree of spatial locality is then defined as the ratio of the actual decrease in cache misses compared to the ideal decrease. Assuming A is the number of cache misses for the longer line size, B represents the number of misses for the shorter line, and l_a and l_b are the line sizes for A and B, the equation becomes:

$$spatial\ locality = \frac{(A-B)}{\left(\frac{A}{l_a / l_b}\right)}$$

Measuring the spatial locality between subsequent cache line sizes in the cache regression, the equation becomes:

$$spatial\ locality\,(from\ doubling\ line\ size) = \frac{(A-B)}{(A/2)}$$

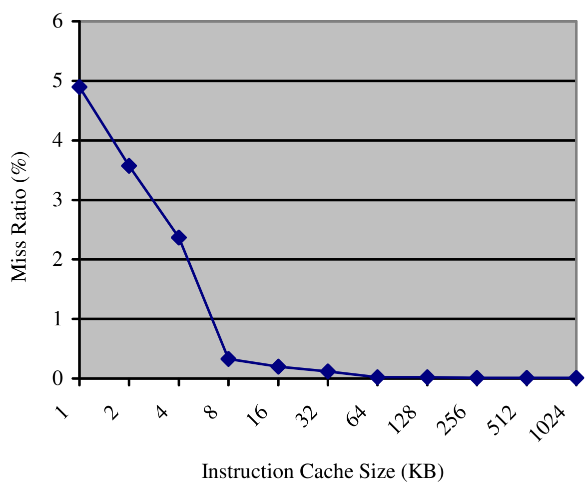

Figure 4 – Average miss rate vs. instruction cache size.

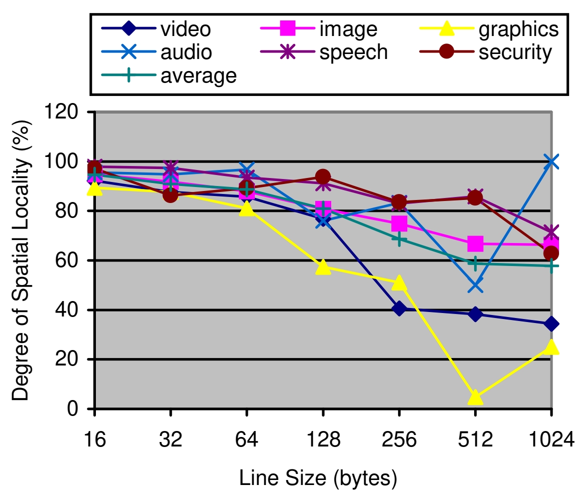

Figure 5 – Instruction memory spatial locality.

Using this equation, the spatial locality for instruction memory is shown in Figure 5. For a given line size, the spatial locality is relative to the next shorter line size, e.g. spatial locality for the 256 byte line is relative to the 128 byte line. As evident in the figure, the spatial locality for instruction memory remains very high for line sizes up to 1024 bytes in most media types except video and graphics. Overall, the average instruction memory spatial locality is 84.8% for line sizes up to 256 bytes, and still remains as high as 77.2% for line sizes up to 1024 bytes.

3.4. Loop Statistics

In the previous sections it was found that multimedia applications have small instruction working set sizes and high instruction memory spatial locality, spending the majority of their time processing over small sections of the program code. To more fully understand the processing characteristics within these frequently executed program sections, we also evaluated the loop characteristics of multimedia applications. Included among the characteristics examined are loop execution weight (by loop level) and the average number of iterations per loop. These statistics will enable a greater understanding of the degree of processing regularity in media processing.

The first characteristic evaluated is loop execution weight by loop level. Using a depth-first search through the functions in each application, we assign a loop level to each loop in every function. The loop level is defined as the number of levels from an innermost loop. Innermost loops are assigned level 1, their parent loops are level 2, and so on. When a parent loop has multiple child loops of different loop levels, the loop level of the parent is defined as one greater than the maximum loop level of all child loops. This loop level definition ignores function boundaries, so loop levels are global.

The results, given in Figure 6, indicate that most multimedia applications spend 80-90%, or more, of their processing time just within inner loops. The second level loops have an even higher execution weight with nearly 95% of the execution time spent within the first and second level loops. The two exceptions that spend a much greater portion of their execution time in lower loop levels are the G.721 applications (*g721dec* and *g721enc*), and the Ghostscript application (*gs*). The G.721 applications have large 4th level loops in which they spend over 30% of their execution time. Ghostscript spends a significant portion of its execution time in outer loops because it is an especially large application with 15 loop levels, much larger than most other applications, which average of 4-8 loop levels.

From the loop execution statistics, we come to the conclusion that instruction memory working set sizes can be so small because such a significant portion of the execution time is spent processing over the two innermost loop levels. However, while this indicates processing regularity, it does not indicate the degree of processing regularity. This requires an understanding of how often these loops iterate.

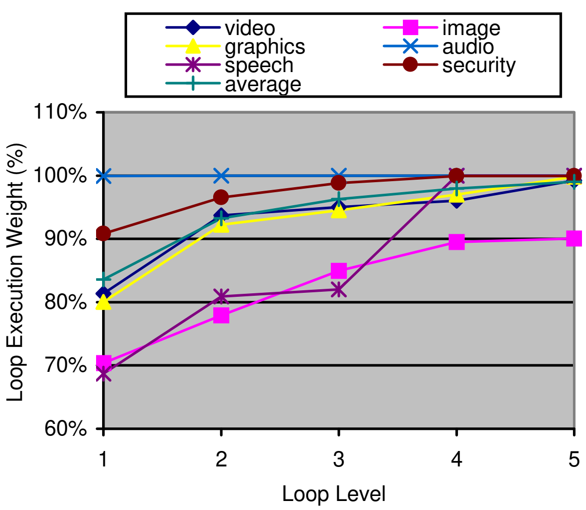

Figure 6 – Loop execution weight vs. loop level.

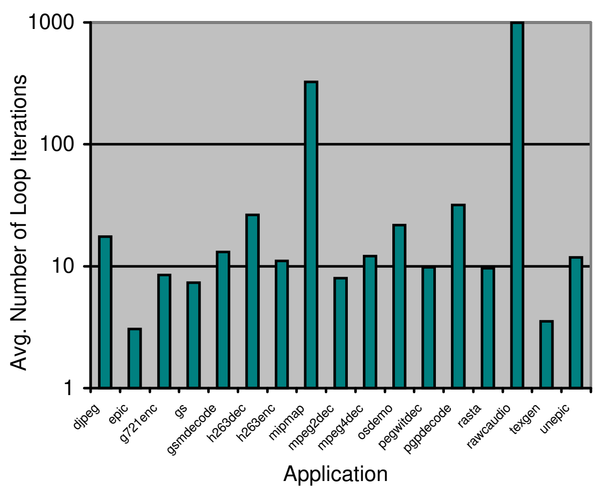

Figure 7 – Average number of loop iterations.

Results on the average number of loop iterations per loop are shown in Figure 7. The average number of loop iterations is weighted according to the number of invocations of each loop. It is calculated by taking the sum for all loops of the average number of iterations multiplied by the number of invocations for that loop, and dividing by the total number of invocations for all loops. This average is weighted by the number of invocations instead of loop execution weight, since weighting by loop execution weight would benefit those loops with more iterations. The results indicate typical loops have a large number of iterations, about 10 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 the training data set versus the evaluation data set indicates these variations are typically within 5% for each application.

Among specific applications, one graphics application, *mipmap*, and the audio applications are all particularly prone to large numbers of iterations, each averaging many hundreds of iterations per loop. The only applications that have few iterations on average are the Epic encoder, *epic*, and the graphics application, *texgen*. Overall, these results enable us to assert that there is a high degree of processing regularity because of the large average number of iterations per loop.

The results of the architecture-independent workload evaluation enable us to conclude that the instruction fetch characteristics of media processing applications are highly regular. Furthermore, they likely do not contribute significantly to the typical execution time of the application. The loop statistics indicate that multimedia applications are highly loop-oriented and each loop iterates an average of 10 times per loop invocation. Coupled with the small working set size and the high spatial locality of instruction memory, we can conclude that an extraordinary fraction of the processing time is spent executing the two innermost loop levels in media applications, so processing is highly regular. In addition, the static branch prediction performance is quite high, providing about 2-3x lower branch misprediction rates than general-purpose applications, so control flow is fairly predictable. Even though the frequency of branches is on the same order as general-purpose applications, we expect that branch mispredictons will not significantly affect the average execution time.

4. ARCHITECTURE-DEPENDENT INSTRUCTION FETCH CHARACTERISTICS

In our architecture-independent evaluation above, we concluded that media processing applications have highly regular instruction fetch characteristics, and that we expect instruction fetch penalties, from both instruction cache misses and branch mispredictions, will not significantly affect the average execution time. However, that does not necessarily mean that additional instruction fetch support is unneeded. If instruction fetch performance can be significantly improved with minimal additional instruction fetch hardware, then it may well be worth the cost of that hardware.

The analyses performed in the previous section primarily evaluate the average workload-dependent instruction fetch characteristics. However, when instruction fetch performance varies widely from the measured average, then execution performance may be significantly affected. Consequently, this section examines the dynamic aspects of instruction fetch by evaluating three dynamic architecture methods: aggressive versus conservative fetch mechanisms, dynamic versus static branch prediction, and the length of the pre-execution pipeline. Based on the results from our results from this evaluation, we can determine what additional instruction fetch hardware may be beneficial for media processing.

4.1. Base Processor Configuration

Systematic evaluation of different architectures requires a base processor model for performance comparisons. We defined an 8-issue base media processor targeting the frequency range from 800 MHz to 1.2 GHz, with instruction latencies modeled after the Alpha 21264, as shown in Table 1. The processor uses a cache memory hierarchy with separate L1 instruction and data caches, a 256 KB unified on-chip L2 cache, and an external memory bus that operates at 1/6 the processor frequency. The L1 instruction cache is a 16 KB direct-mapped cache with 256-byte lines and a 20 cycle miss latency (assuming a hit in L2). The L1 data cache is a 32 KB direct-mapped cache with 64-byte lines and a 15 cycle miss latency. It is non-blocking with an 8-entry miss buffer, and uses a no-write-allocate/write-through policy with an 8-entry write buffer. The L2 data cache is a 256 KB 4-way set associate cache with 64-byte lines and a 50 cycle miss latency. It is non-blocking with an 8-entry miss buffer, and uses a write-allocate policy with an 8-entry write buffer. Further details are available in Fritts[6].

Operation	*Number of Units*	*Latency*
ALU	8	1
Branches	1	1
Load/Store	4	loads - 3, stores - 2
Floating-Point	2	4
Multiply/Divide	2	multiply - 5, div - 20

Table 2 – Number of parallel functional units and their operation latencies**.**

4.2. Aggressive vs. Conservative Fetch

A popular method for reducing the impact of stall penalties from instruction cache misses is to decouple the instruction fetch pipeline from the execution pipeline using an instruction buffer. The instruction buffer works in conjunction with branch prediction to prefetch the predicted instruction control stream. Prefetched instructions are placed in the instruction buffer, from which the execution pipeline is able to access instructions as needed. The buffering between the fetch and execute pipelines enables each to continue operation when the other is stalling. This decoupled fetch-execute method is commonly used in superscalar architectures.

An experiment was performed to evaluate the benefit of aggressive, decoupled fetch versus conservative, un-buffered instruction fetch. Using an instruction buffer of three times the processor issue width, the performance of aggressive and conservative fetch is compared on four different processor architectures, shown in Figure 8. As evident, the aggressive fetch method provides negligible benefit for VLIW architectures, which are forced to issue complete groups of parallel operations. The aggressive fetch mechanism is more beneficial to superscalar architectures, which are able to issue operations atomically. However, the out-of-order superscalar processor has an issue-reorder buffer, which serves in a similar capacity, so an instruction buffer provides minimal additional benefit. Even the in-order superscalar architecture achieves only a moderate performance improvement. Because multimedia applications are highly loop-oriented and only execute over small sections of code, the decoupled fetch engine provides minimal gain for media processors.

4.3. Dynamic Branch Prediction

Branch prediction is a necessary mechanism for reducing the penalties associated with changes in the direction of the instruction control stream. Static branch prediction is most important from the perspective of static scheduling because its accuracy dictates the effectiveness of global scheduling techniques such as speculation and predication, which move operations across branches or combine branches, respectively. However, even though the predictable nature of multimedia code enables good static branch prediction in media processing, its accuracy is still limited. For example, the average branch hit rate is 85.9% for static branch prediction, whereas the branch hit rate of a 1024-entry 2-bit counter dynamic branch predictor is 91.1%, an improvement of 58% in branch miss rate.

For further reduction of the penalties from mispredicted branches, a comparison was made of two types of dynamic branch predictors: a dynamic uncorrelated 2-bit counter predictor and a dynamic branch history table predictor. For the 2-bit counter predictor, predictor table sizes from 4 to 8192 entries were examined. With regards to dynamic branch history table predictors, there exist a variety of alternatives. Yeh and Patt organized the major alternatives into nine different categories[12]. On a cost-performance basis, they found that the PAs(6,16) predictor, which is a per-address history table with 6 bits of branch history and 16 pattern history tables, provides the best performance on general-purpose code. However, they also found additional bits of branch history provide improved performance. Consequently, we chose to evaluate a variety of per-address history tables to determine the appropriate number of branch history bits and pattern history tables. Specifically, we examined the per-address history table, PAs(k,p), with the number of entries varying from 4 to 2048 entries, the number of history bits ranging from 1 to 10, and the number of pattern history tables ranging from 1 to 16.

From the results of these comparisons, two observations were immediately obvious. First is that media processors do not require a large number of entries in dynamic branch prediction tables. Figure 9 shows the results for various

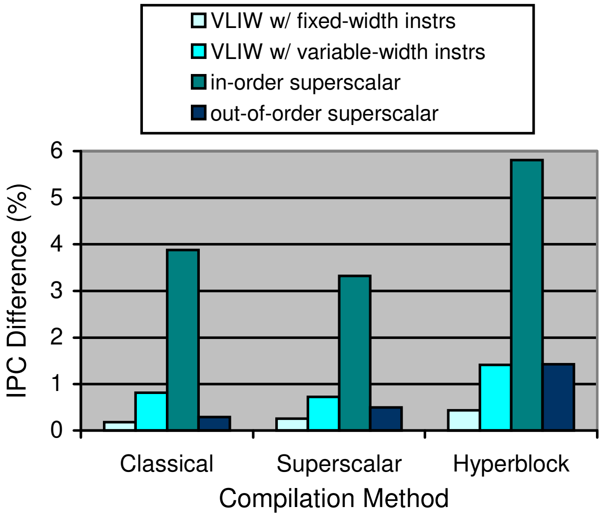

Figure 8 – Aggressive vs. conservative fetch performance.

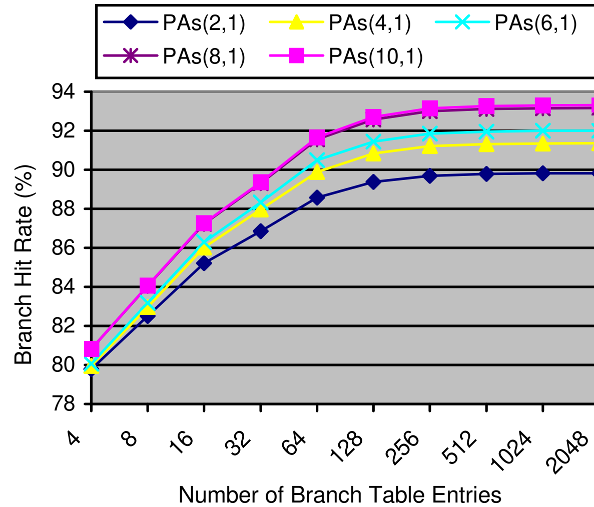

Figure 9 – Hit rate for various branch history tables.

numbers of branch table entries for the PAs(2,1) through PAs(10,1) dynamic branch predictors. The results show that branch predictors with 128 to 256 entries are sufficient for media applications. And though it is not shown, the same is true of the uncorrelated 2-bit counter predictor. However, there is significant variation between applications with respect to the ideal number of entries in the branch predictor. Figure 10 examines the results on a per-application basis. Many applications require as few as 4-16 entries, while other applications require as many as 256-512 entries for best performance. However, in many of the applications requiring more than 128 or 256 entries, relatively good performance is still achieved with 128 or 256 entries, so branch predictors with 128 or 256 entries are sufficient for media processing.

The other observation that was immediately obvious is that media applications do not benefit much from multiple pattern history tables. For example, comparing the difference between the PAs(6,1) predictor and the PAs(6,16) predictor, which has 16x times the number of pattern history tables as the PAs(6,1), it was found that the average branch hit rate was only 0.4% higher for the PAs(6,16) predictor. Only in couple of the applications did the branch hit rate difference become as large as 1-2%. The difference was most prominent for very small branch predictors, those with less than 64 entries, but still remained fairly small in general.

Another important aspect of dynamic branch prediction is that the branch hit rate of the per-address history table predictor varied considerably based on the compilation optimization method used. For example, using a 256-entry PAs(8,1) predictor, the branch hit rate for classical and hyperblock optimized code were about 93.5%, while for the superscalar optimized code it was only 91.2%. The reason for this is that superscalar optimization performs loop unrolling and control speculation that decrease the predictability of control flow. These optimizations are also applied to the hyperblock-optimized code, but hyperblock optimization also performs predication, which converts many of the more unpredictable branches from control dependencies to data dependencies using if-conversion. Consequently, predication helps raise the branch hit rate.

With regards to selecting the appropriate branch predictor, the size of the branch predictor is an important consideration. With regards to the uncorrelated 2-bit counter predictor, not counting the bits needed for the address fields, the number of bits needed is $2b$, where b is the number of entries in the predictor. For the per-address history tables, Yeh and Patt[12] determined the total number of history and prediction bits for the PAs(k,p) according to the equation given below. In this formula, the size of the PAs(k,p) predictor is defined by b, k, and p, where b is the number of entries in the branch history table, k is the number of history bits per entry, and p is the number of pattern history tables in the branch history table.. This cost metric ignores the bits for address fields, as these fields will be necessary for all methods.

$$size\_PAs(k,p) = bk + 2^{k+1}p$$

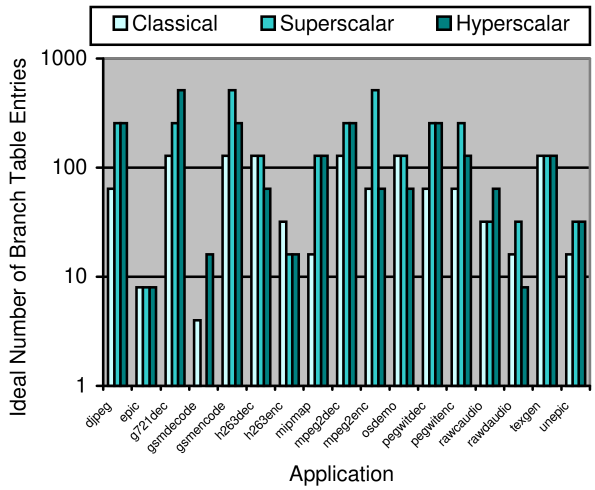

Figure 10 – Ideal entry sizes for each application.

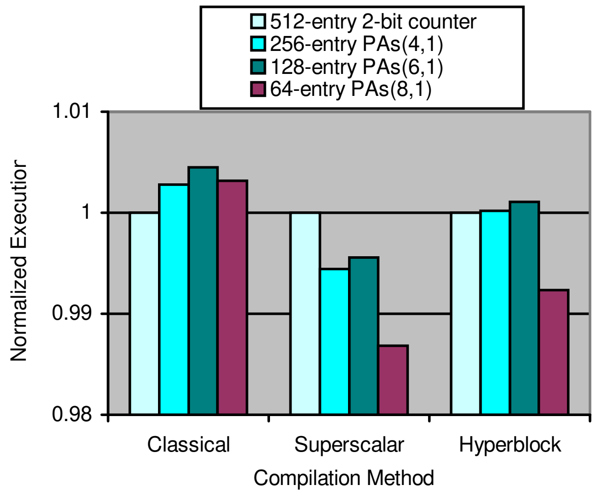

Figure 11 – Dynamic branch prediction performance.

Using this cost metric, we can compare the execution time performance of four possible 1K-bit predictors in Figure 11. The four dynamic predictors which are all approximately 1K-bit in size are a 512-entry uncorrelated 2-bit counter, the 256-entry PAs(4,1), the 128-entry PAs(6,1), and the 64-entry PAs(8,1). All results in the figure are normalized to the performance of the uncorrelated 2-bit counter predictor. We can see from the results that, except for the superscalar-optimized code, the 128-entry PAs(6,1) performs the best. However, the degree of difference between the various results is minimal. The results tend to indicate that the number of history bits in branch history tables is slightly more important than the number of entries when there are at least 128 entries in the table, but again, the difference is minimal.

Dynamic branch prediction performance can be improved with much larger branch predictors, but even with branch predictors as 5-10x the size of the 128-entry PAs(6,1) predictor, the difference in IPC is still only 2-3%. Consequently, small dynamic branch predictors, whose branch misprediction rates 2x lower than static branch prediction, are sufficient for media processing.

4.4. Pre-Execution Pipeline Length

The length of the pipeline prior to the first execution stage is of importance in determining the cost of mispredicted branch penalties. A branch typically resolves during the first execution stage of the pipeline, so the cost of mispredicting a branch is equal to one plus the number of pre-execution pipeline stages. In most existing processors the number of pre-execution pipeline stages is usually between three and five. There are 1-2 instruction fetch stages, 1-2 decode stages, and a register fetch stage. Superscalar processors typically have longer pre-execution pipelines than VLIW processors. We evaluated pre-execution pipelines with lengths from two to six stages on all three processor models. We also evaluated the performance using both conservative and aggressive fetch mechanisms, and fixed and variable-width VLIW instruction formats. Surprisingly, there was minimal variation in performance across the different architectures. For each additional pre-execution pipeline stage added, performance dropped by only 2%. Furthermore, this performance degradation was always within the range of 1.5-2.5%, irrespective of architecture style, compilation method, or fetch mechanism. Consequently, we can definitively expect a 2% performance degradation for each additional pre-execution stage in media processor design.

Our examination of the dynamic aspects of instruction fetch characteristics in media processing has yielded two important results. First, even using various dynamic architecture methods, they all only provide minimal performance gains with respect to overall execution time. This is because the instruction fetch characteristics of media processing are effectively ideal and contribute only minimally to application execution performance. Secondly, branch prediction performance can be improved significantly using a small dynamic branch predictor, which provides at least a 2x decrease in branch misprediction rates.

5. CONCLUSIONS

In the course of this study we have shown that media processing applications embody nearly idealistic instruction fetch characteristics. The analysis of the architecture-independent workload characteristics proved that media applications are highly loop-centric. They spend nearly 95% of their time processing within the two innermost loop levels of their programs. Consequently, their instruction memory working set sizes are very small, typically less than 8KB, and their instruction memory spatial locality is very high. In addition, loops in media applications iterate at least 10 times on average. In effect, media applications display high degrees of processing regularity.

Additionally, multimedia is characterized by a fairly regular, predictable control flow. While the frequency of branches is similar to general-purpose applications, the static branch misprediction rate is 2-3x lower than general-purpose applications. With this degree of control flow predictability, we anticipated that instruction fetch penalties from cache misses and branch mispredictions will not significantly affect the average execution time.

In our examination of the dynamic architecture-dependent instruction fetch characteristics, we found this to be true. There is little benefit from using instruction buffers to enable decoupled fetch and execute pipelines. Also, variations in the length of the pre-execute pipeline only impact performance by 2% per additional pipeline stage. The primary dynamic aspect of note is branch prediction. While static branch prediction is high in media applications, it is still possible to obtain a decrease of at least 2x in the branch misprediction rate using dynamic branch prediction. This can be accomplished using a variety of small dynamic branch predictors. In our evaluation of per-address history tables, we found a 128-entry PAs(6,1) predictor to provide the best performance among the predictors of its size.

All these results lead us to conclude that media applications do indeed have nearly idealistic instruction fetch characteristics. The penalties associated with instruction fetch simply do not contribute much to the overall execution time of media applications. This is true of all programs evaluated in the MediaBench benchmark, and so we expect that this characteristic is true of most media applications in the current generation of multimedia.

REFERENCES

1. Keith Diefendorff and Pradeep K. Dubey, "How Multimedia Workloads Will Change Processor Design," *IEEE Computer*, September 1997, pp. 43-45.
2. John Stokes, "The Playstation2 vs. the PC: a System-level Comparison of Two 3D Platforms," Ars Technica, April 2000.
3. Masaaki Oka and Masakazu Suzuoki, "Designing and Programming the Emotion Engine," *IEEE Micro*, Nov. 1999.
4. Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia Communications Systems," *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.
5. MediaBench home: http://www.cs.ucla.edu/~leec/mediabench/
6. Jason Fritts, "Architecture and Compiler Design Issues in Programmable Media Processors," Ph.D. Thesis, Department of Electrical Engineering, Princeton University, 2000.
7. P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: an architectural framework for multiple-instruction-issue processors," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, Toronto, Canada, May 1991, pp. 266-275.
8. IMPACT compiler group: http://www.crhc.uiuc.edu/Impact/
9. Jason Fritts, Wayne Wolf, and Bede Liu, "Understanding multimedia application characteristics for designing programmable media processors," *SPIE Photonics West, Media Processors '99*, San Jose, CA, January 1999.
10. John L. Hennessy and David A. Patterson, "Computer Architecture: A Quantitative Approach (Second Edition)," Morgan Kaufmann Publishers, Inc., 1996.
11. Ricardo E. Gonzalez, "Xtensa: A Configurable and Extensible Processor," *IEEE Micro*, March 2000, pp. 60-70.
12. Tse-Yu Yeh and Yale N. Patt "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History," *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 257-266, 1993.