Implementation of an Algorithm for Fast Down-Scale Transcoding of Compressed Video on the Itanium ™

Sumit Roy
Media Systems Architecture Group
Client and Media Systems Laboratory
Hewlett-Packard Laboratories
Palo Alto, CA 94304
sumit@hpl.hp.com

Bo Shen
Streaming Media Technology Group
Client and Media Systems Laboratory
Hewlett-Packard Laboratories
Palo Alto, CA 94304
boshen@hpl.hp.com

ABSTRACT

With the introduction of the next generation wireless networks, mobile devices access increasingly more media-rich content. However, a key factor that prevents a mobile device from accessing multimedia content is that it does not have enough display real estate to render the content that is traditionally created for the desktop web client. Moreover, the wireless network typically has lower bandwidth compared to a wired network of the same epoch. Therefore, a transcoder is needed somewhere in the network to transform multimedia content to the appropriate form factor and bandwidth requirement. This paper introduces an extremely fast transcoder. Compared to previous methods, additional 30 - 44 % of computing power is saved while the quality of transcoding is maintained. This saving in computing resources maps into the benefit that more concurrent sessions can be supported on one transcoding device. The computational power is saved by exploiting some of the key features of the new Itanium ™ Processor Family.

Categories and Subject Descriptors

C.1.1 [**Single Data Stream Architectures**]: RISC/CISC, VLIW architectures; C.4 [**Performance of Systems**]: Measurement techniques

General Terms

Algorithms, design, measurement, performance

Keywords

Video transcoding, Itanium ™, MPEG-2

1. INTRODUCTION

The Internet brings heterogeneous devices together. For rich media transmission over the wireless Internet, content adaptation is a key issue. The original content may have been coded at higher resolution and higher bit rate, say 720×480 at 2 to 8 Mbps for DVD quality, or 320×240 at 1.5 Mbps for desktop clients connected to the Internet through a T1 line. However, due to the characteristics of mobile wireless communication ie. low bandwidth channel and limited display real estate, a 100 kbps video at a lower resolution is desired. Current 3G wireless communication is trying to provide a 128 – 384 kpbs communication channel. Therefore, a transcoder is needed to adapt the content to the appropriate size and bit rate.

A straightforward method to perform this transcoding is to decode the original stream, down sample the decoded frames to a smaller size and re-encode to a lower bit rate. Considering a typical CCIR601 MPEG-2 video, it takes the full power of a 400 Mhz CPU Pentium class CPU to perform real-time decoding. Encoding is even more expensive, which makes the straightforward method non-practical. From a price performance point of view, it is expected that one transcoding unit should be able to support as many concurrent sessions as possible. This would for example permit a transcoding service to support multiple different streams in near real time scenarios. Therefore, it is extremely worthwhile to develop fast implementations that reduce the CPU load for such kind of transcoding session.

This paper describes our implementation of an optimized video transcoding algorithm using a compressed domain approach. Specifically we show the performance advantages of using the multimedia instruction set of the new Itanium ™ Processor Family (IPF) from Intel, as well as how we used some of the novel features of these processors.

The paper is organized as follows. In Section 2, we present background on the compressed-domain video transcoding algorithm. The IPF features used in the optimized implementation are described in Section 3. Detailed performance results are presented in Section 4. We conclude with directions for future work in Section 5.

2. TRANSCODING SYSTEM

Down-scale transcoding is defined as generating compressed bitstream for down scaled video of size N/n×N/n ($n = (2, 3, 4)$) given the bitstream for original video of size N×N. The conventional transcoding approach requires the original video be decompressed and motion vectors recomputed for the downscaled video followed by re-encoding in a conven-

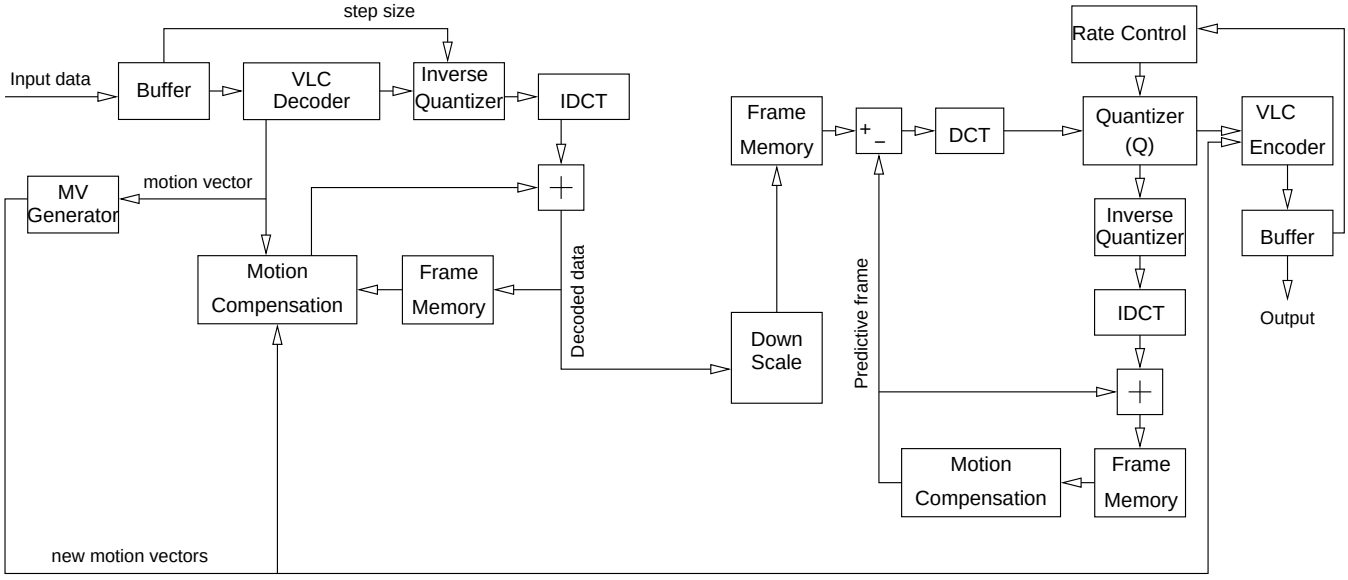

Figure 1: Hybrid Transcoding System.

tional video encoder.

Video compression standards such as MPEG [1], H.26x employ motion compensated prediction to exploit the temporal redundancy to achieve a lower bit rate. Motion estimation is often employed in the motion-compensation process; however, motion estimation process is a compute-intensive operation and typically is at least 60% of the workload of the video encoder. Therefore, recomputing the motion vectors renders the problem of video downscaling from a compressed video bitstream as a computationally intensive task and may place a heavy burden on a transcoding server that has to generate MPEG or H.26x bitstreams for the downscaled video in real time.

Figure 1 illustrates the processing flow of a compressed-domain transcoding system for MPEG video [2]. In the transcoding system, the spatial frames are reconstructed and downscaled in the spatial domain but the motion vectors are estimated directly from the existing motion vectors in the original sequence. Therefore, a costly motion estimation process is saved. Similar systems can also be developed for the downscaling of H.261 and H.263 bitstreams.

The MV Generator module is responsible for generating the new motion vector by averaging the existing motion vectors. In addition, the coding type of the output macroblock is also decided using the coding types of the input macroblocks. For instance, in down-scale-by-two case, there are four input macroblocks involved in generating one output macroblock. If most of the input macroblocks are intra, the output macroblock is coded as intra. On the other hand, if most of the input macroblocks are predicted, the output macroblock is coded as predicted, in which case, the output macroblock is constructed using the new motion vector produced by the MV Generator module.

Moreover, if the output macroblock is decided to be predicted, but there is one intra macroblock, one skipped macroblock among the input macroblocks, we view the intra macroblocks and skipped macroblocks as predicted macroblocks with zero-valued motion vector. Note that the skipped macroblocks in bi-directionally predicted frames in MPEG or H.263 may have non-zero-valued motion vectors.

The bit rate of the output bitstream is controlled by the rate-control module. The rate-control module also uses the compressed domain information existing in the original stream. It first estimates the number of bits available to code the picture then computes a reference value of the quantization parameter based on buffer fullness and target bit rate. Finally, it derives the value of the quantization parameter from the generated reference according to the spatial activity of the macroblock. The spatial activity is derived from the DCT coefficient activity in the input macroblocks. Therefore, there is no need to compute a mean square variance which consumes more CPU cycles.

A simplified version, similar to the transcoding system described above is illustrated in Figure. 2. This system represents a simplified version in which an open-loop approach is used. The open-loop approach may cause an error propagation problem on the receiver. However, the simplified system performs much faster due to the elimination of the one inverse quantization process and one IDCT process per loop. The PSNR results in Section 4 show that the overall noise added with these optimizations is well within 0.6 dB for this system, when compared to a traditional pixel domain transcoding approach.

3. ARCHITECTURAL FEATURES OF THE ITANIUM ™ PROCESSOR FAMILY

The Itanium ™ Processor Family (IPF) is an implementation of the IA-64 architecture [3]. It features a new processor architecture technology called EPIC, or Explicitly Parallel Instruction Computing. Multiple instructions are combined

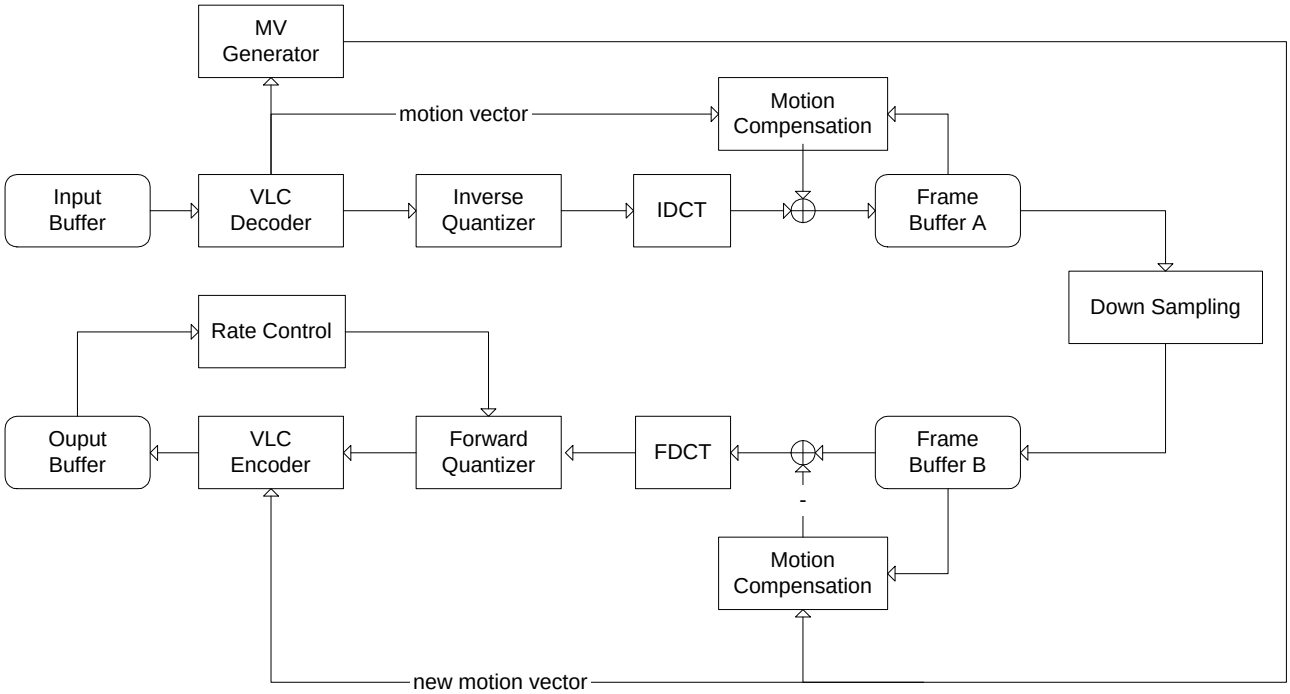

Figure 2: Compressed Video Transcoding system

together into large instruction words called bundles. If the processor implementation has enough functional units and if the instructions in the bundle do not have any dependencies on each other, all of them can be executed at the same time.

3.1 Instruction Bundles

The Itanium ™ is the first implementation of this processor architecture. In this case, each bundle consists of three instructions. If there is a dependent instructions in a stream are separated by stopbits. This guarantees that the processor only issues independent instructions in a cycle. For the Itanium ™, the issue width is two bundles, which means that a maximum of six instructions can be issued per cycle.

To support multiple instructions per cycle, the processor has numerous functional units. In the Itanium ™ implementation, there are two memory units, two integer ALU units, two floating point units, and three branch units. The mapping of instructions in a bundle to a functional unit is indicated via template bits in the bundle. If the sequence of independent instructions in the bundles results in an oversubscription of the functional units eg. three integer arithmetic instructions are issued consecutively, a so called split-issue is cause. The remaining instructions are issued in the next cycle. The processor will not make any attempt to reorder the instruction stream to avoid split issues. The functional units of a specific type are also not completely symmetric [4]. For example, multimedia multiply instructions can only be done on integer ALU unit 0.

3.2 Multimedia Instructions

Intel introduced the MMX instructions in the Pentium family of processors. The technology is based on the Single-Instruction, Multiple-Data concept from parallel processing. Multiple small data items are packed into a 64-bit register and all processed in a single instruction. The initial set of 57 instructions operates on integers that are 8, 16, or 32 bits wide, and are known as the MMX extensions. Early implementations of this architecture multiplexed the MMX registers on the Floating Point registers and state.

The SSE instructions are introduced with the Pentium III family. They allow packed IEEE compliant single precision floating point arithmetic on four operands at a time. Some additional packed integer operations are also introduced.

The IA-64 has multimedia instructions which are semantically equivalent to the MMX and Streaming SIMD (SSE) instructions introduced with the Intel Pentium and Pentium III [5]. However, there is no special multimedia register set on the Itanium ™. All general purpose registers can be treated as eight 8-bit, four 16-bit, or two 32-bit elements. Packed integer arithmetic instructions are particularly useful while processing streaming media, since data tends to be accessed in blocks, and each datum can be processed independently. Most of the multimedia ALU instructions for the Itanium ™ can either be executed on the M-units of the I-units. Exceptions include the shift, pack, and multiply instructions, which can only be dispatched to the I units.

One of the limiting factors in using the MMX and SSE instructions on Pentium class processors was the limited size of the MMX register set, only 8 in the Pentium III implementation. The Itanium ™ on the other hand has a register set of 128 64-bit registers which could be used to completely process an image block without any looping.

4. EXPERIMENTAL RESULTS AND ANALYSIS

The code was compiled using the `gcc` compiler, and results were obtained on a dual processor 800 MHz Itanium ™ workstation, running the RedHat Seawolf release of the Linux operating system. Data was collected for two different reference MPEG-2 sequences, flower-garden and football. The football test stream is of size 704×480 pixels and is coded at 6 Mbps. It has 150 frames. The flower-garden test stream contains 450 frames of size 704×480 pixels and is coded at 4 Mpbs. In the experiments, down-sample-by-two and down-sample-by-four operations also reduce the bit rate by a factor of 4 and 16 respectively.

4.1 Pro£ling Data

The transcoder code was initially compiled without optimization at *-O0* to obtain the baseline performance. The program was then recompiled with the compiler optimization set to *-O3*. Table 1 shows the ten procedures that take the most time as output determined by `gprof(1)` (_divdi3 is a library routine).

Table 1: Output of `gprof(1)` for code compiled with *-O3*

% time	seconds	calls	name
26.78	25.37	1188000	fdct
13.91	13.18		_divdi3
11.21	10.62	5537394	recon_comp
8.31	7.87	17843632	idctcol
4.57	4.33	31550362	flushbits
4.54	4.30	991488	quant_non_intra
4.43	4.20	600	getMBs
4.20	3.98	2230454	addblock
2.97	2.81	17843632	idctrow
2.38	2.25	600	down2PixelPic

Based on the profiling data the following code is replaced with a multimedia instruction optimized version:

1. Forward DCT (fdct) - The optimized FDCT code is adapted from the mjpeg_beta1a package, which is derived from [6].

2. Motion Compensation (recon_comp) - This uses an adapted motion compensation code from Aaron Holtzman's mpeg2dec library [7].

3. Prediction (predict) - This also uses routines from the same Motion Compensation code optimization.

4. Downsampling (down2PixelPic) - The subsampling code was developed in-house.

The initial implementation consisted of mapping the original IA32 optimized assembly code to the equivalent IA64 code. When required, stopbits were inserted in the code sequence to maintain the dependencies between instructions. The results shown only use 10 registers explicitly, *r20 – r30*. The compiler further uses registers *r14 – r19* as address registers.

Table 2: Output of `gprof(1)` for code compiled with *-O3* and Multimedia optimized FDCT, Motion Estimation, Prediction, and Downsampling

% time	seconds	calls	name
19.17	11.91		_divdi3
11.95	7.43	17843632	idctcol
6.93	4.31	31550362	flushbits
6.70	4.17	991488	quant_non_intra
6.59	4.09	600	getMBs
6.37	3.96	2230454	addblock
4.56	2.84	17843632	idctrow
3.96	2.46		forward_dct_row1
3.08	1.91	1761505	MC_put_y
2.44	1.51	792000	var_sblk

Table 2 shows the profiling results after the FDCT, Motion compensation, prediction, and downsampling multimedia optimizations have been applied (forward_dct_row1 and MC_put_y are multimedia optimized routines).

4.2 Computational Speedup

Figure 3 shows the computational time for both down-sample-by-two operation and down-sample-by-four operation on the two test streams. The sppedup is computed with respect to the case when no optimization is used in the compiler, *O0*. The default optimization level for *gcc* is *O1*, and this alone provides a speedup of four in all cases. The speedup for an optimization level of *O2* lies between 4.6 - 5.2, depending on both the input sequence and the down-sample ratio. There is no change in the performance when going from *O2* to *O3*.

Table 3 shows the percentage improvements when adding multimedia optimized routines for the FDCT (F), the Motion Compensation (MC), the Prediction (P), and finally the Down sampling D. Note that the current implementation uses MMX optimized down sampling code for the down-sample-by-two case only.

From the table it can be seen that the overall execution time can be reduced by a factor of 30 - 44 % from the default optimization level of *O1*.

4.3 PSNR Comparison

The use of multimedia instructions can lead to round-off errors, for example when single precision floating point code is replaced with an integer version. Hence we show the results of the PSNR difference for the two sequences in Figure 4 and Figure 5 when each optimization is applied. The reference frames were obtained by downsampling the decoded input stream. In the figures, plot *-O3* has only compiler optimizations, F has multimedia instruction optimized FDCT, $F + MC$ adds optimized Motion Compensation, $F + MC + P$ further adds optimized prediction, and finally, $F + MC + P + D$ adds optimized downsampling. In each case the video sequence is reduced by a factor of two in size, and factor of four in bitrate. It is seen that the additional noise is negligible for all optimizations.

4.4 Itanium ™ Speci£c Optimizations

Figure 6 shows a part of the FDCT MMX code optimized IA32 code derived from [6]. The IA64 version of the code de-

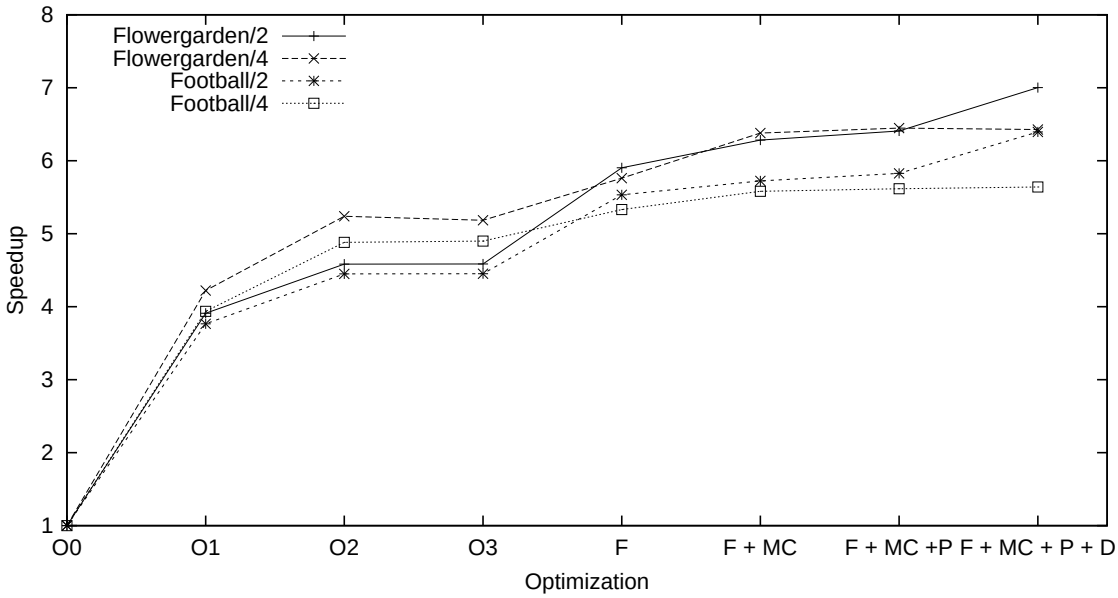

Figure 3: Speedup due to various Multimedia Optimized code blocks on IA64

Table 3: Percentage Reduction in Execution Time with Multimedia optimized FDCT, Motion Estimation, Prediction, and Downsampling

Optimized Procedures	Baseline		
	O0	*O1*	*O3*
F	81.24 - 83.06	26.19 - 33.76	8.13 - 22.32
F+MC	82.08 - 84.32	29.49 - 37.75	12.24 - 27.01
F+MC+P	82.20 - 84.49	29.93 - 38.97	12.79 - 28.43
F+MC+P+D	82.27 - 85.72	30.23 - 44.15	13.15 - 34.51

rived using a simple translation scheme is shown in Figure 7. In the actual implementation, this pass was accomplished by wrapping the original IA32 code into macros and rewriting the macro definitions for IA64.

Line 1. shows that the simpler load/store primitives of the IA64 instruction set require some code expansion when emulating enhanced addressing formats. This is even more apparent in the code expansion for Line 2. For the code fragment shown, 9 IA32 operations were translated to 12 IA64 operations. Due to the dependencies between statements, no fewer than 7 stopbits had to be inserted. This leads to an underutilization of the functional units of the IA64.

The code fragment shown in Figure 8 shows one particular rescheduling of the code. In this case, the following optimizations were possible:

1. Reorder 5. since there is no dependence. This also hides the 2 cycle bypass latency from the load of register *r30* to its use in statement 7.
2. Execute 1a. and 2a. in parallel, since there is no dependence.
3. Execute 1b. and 2b. in parallel, since there is no dependence.
4. Remove 3a. *r22* holds the old value of *r25*, by rewriting 6. we achieve the same semantics and save a statement (*r22* is not used in subsequent code).
5. Remove 7. *r21* holds the old value of *r30*, by reordering statements 8. and 9. we achieve the same semantics with fewer statements and stopbits.

The overall result is that we have reduced the number of instructions to 10 and the number of stopbits to 5. This gives us greater concurrency and shorter code paths. Depending on the bundling of the instructions, one must also consider the latencies between producers and consumers of variables [3].

The optimizations described above were applied to the complete FDCT code. The FDCT code was converted into a micro-benchmark for this experiment, so that the effect of the code rescheduling could be isolated. Table 4 shows the results from three implementations, measured using the Performance Measuring Unit (PMU) on the Itanium ™. The Perfomance Metric uses the nomenclature defined in [3]. *Original* refers to the translated IA32 code, *Code Path* is a first attempt at optimization by reducing the length of the code path. The number of IA64 instructions retired has been reduced by 20 %. A primary source of this reduction is to use the postincrement form of load/store instructions.

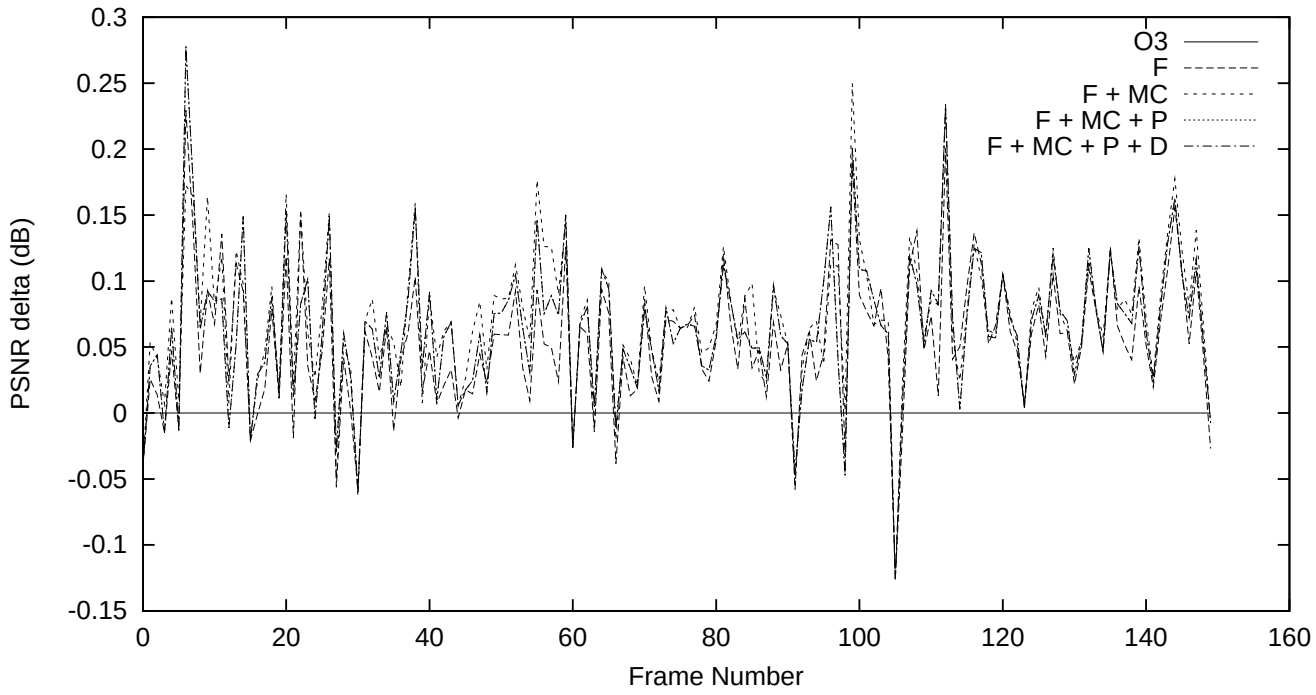

Figure 4: PSNR difference for different levels of optimization (football sequence)

```
      .
      .
      .
1.  movd mm5, [INP+12];

2.  punpcklwd mm5, [INP+8];

3.  movq mm2, mm5;
4.  psrlq mm5, 32;

5.  movq mm0, [INP];
6.  punpcklwd mm5, mm2;

7.  movq mm1, mm0;
8.  paddsw mm0, mm5;

9.  psubsw mm1, mm5;
      .
      .
      .
```

Figure 6: IA32 MMX FDCT code fragment

This save extra address calculation operations like shown in 1a. and 2a. in Figure 8. However, the execution cycles has only reduce by about 10 %. Table 5 shows some metrics derived from the raw PMU data. It is seen that the Instructions Per Cycle (IPC) actually decreases from the *Original* implementation to the *Code Path* implementation. The *Latency* implementation actually considers the latency between dependent instructions while scheduling them. For example, using postincrement load/stores introduces a 2 cycle latency on the use of the new value of the base register. Regular address calculations with *adds* have a one cycle latency to the use of the base register only if they were assigned to an M-unit. By taking such restrictions into account, one can increase the IPC as well as reduce the cycles lost due to issue limits. It is to be noted that a large number of cycles is still lost due to this limit. This would suggest that using more registers for the current code is unlikely to provide a large improvement. The current implementation uses 1287 instructions, with approximately 597 stopbits per 8×8 FDCT. The original implementation used 1719 instructions, with 1160 stopbits.

Table 5: Derived Metrics from Performance Counters

Implementation			Derived
Original	Code Path	Latency	Metric
0.79	0.70	0.93	IPC
6788.4	3804.3	3447.2	Issue Limit cycles

5. CONCLUSION

This paper presents the results of using some Itanium ™ Processor Family architectural optimization on a video transcoding system. Using multimedia instructions provides an increase in the speedup obtained from the compiler alone by about 30 - 44 %. Results on exploiting the Explicitly Parallel Instruction Computing features of the Itanium ™ suggest an additional improvement of nearly 40 % for some of the core routines in the transcoder. Much of the improvement comes from reducing code path lengths and correctly handling instruction pair latencies. It is also found that the number of functional units in the Itanium ™ are a limiting factor for some of the code. Our future work will explore adding these performance enhancing features to other parts

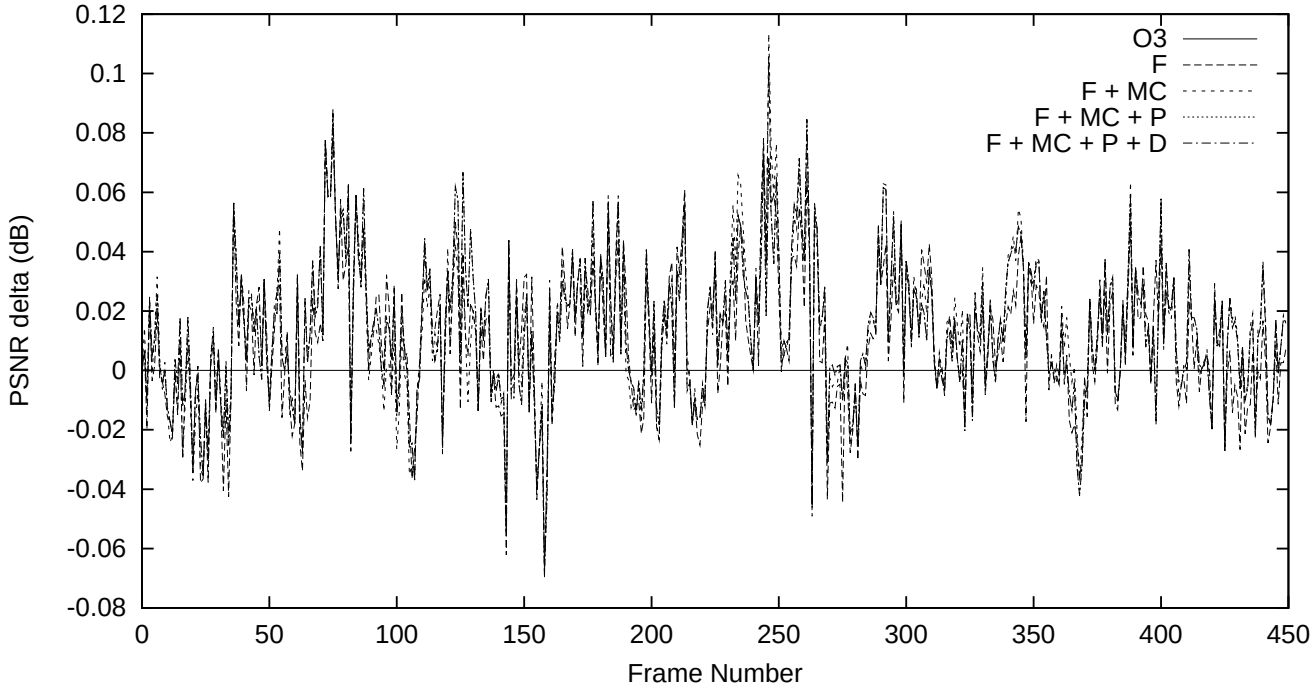

Figure 5: PSNR difference for different levels of optimization (flower-garden sequence)

Table 4: Sample Output from Performance counters on Itanium for 10 iterations of the FDCT kernel

Implementation			Performance
Original	Code Path	Latency	Metric
21684.6	19524.2	13746.9	CPU_CYCLES
17192.3	13832.2	12873.3	IA64_INST_RETIRED
11608.2	8620.1	5974.6	EXPL_STOPS_DISPERSED
13693.3	10329.2	7453.9	ALL_STOPS_DISPERSED
8258.3	5911.5	5058.4	DEPENDENCY_ALL_CYCLE
1469.9	2107.2	1611.2	DEPENDENCY_SCOREBOARD_CYCLE
3105.6	3193.0	3144.8	MEMORY_CYCLE
6348.7	3789.0	3549.1	NOPS_RETIRED
2660.9	2276.2	2338.0	UNSTALLED_BACKEND_CYCLE

of the code.

6. REFERENCES

[1] ISO, *Coded Representation of Picture and Audio Information, Test Model 5*, iso-iec jtc1/sc29/wg11/n0400 ed., 1993.

[2] B. Shen, I. Sethi, and V. Bhaskaran, "Adaptive Motion-vector Resampling for Compressed Video Downscaling," *IEEE Transactions On Circuits and Systems for Video Technology*, vol. 9, pp. 926 – 936, September 1999.

[3] Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual*, July 2000.

[4] intel, *Itanium Processor Microarchitecture Reference*, 245473-002 ed., August 2000.

[5] Intel Corporation, *Intel Architecture MMX Technology Developer's Manual*.

[6] Intel Corporation, *Intel Application Note AP-922 - fast, precise implementation of DCT*, http://developer.intel.com/vtune/cbts/appnotes.htm ed.

[7] A. Holtzman, "http://dara.notbsd.org/ aholtzma/ac3/mpeg2dec.php.".

```
      .
      .
      .
1a.  adds  r14 = 12, r32  # assume r32 = INP
     ;;
1b.  ld8   r25 = [r14]

2a.  adds  r15 = 8, r32
     ;;
2b.  ld8   r29 = [r15]    # r29 = temporary register
     ;;
2c.  unpack2.l r25 = r25, r29
     ;;
3a.  mov   r22 = r25
4.   shr.u r25 = r25, 32
     ;;
5.   ld8   r30 = [r32]
6.   unpack2.l r25 = r25, r22
     ;;
7.   mov   r21 = r30
8.   padd2.sss r30 = r30, r25
     ;;

9.   psub2.sss r21 = r25, r21;
      .
      .
      .
```

Figure 7: Translated IA64 MMX FDCT code fragment

```
      .
      .
      .
5.   ld8   r30 = [r32]
1a.  adds  r14 = 12, r32  # assume r32 = INP
2a.  adds  r15 = 8, r32
     ;;
1b.  ld8   r25 = [r14]
2b.  ld8   r29 = [r15]    # r29 = temporary register
     ;;
2c.  unpack2.l r25 = r25, r29
     ;;
4.   shr.u r22 = r25, 32
     ;;
6.   unpack2.l r25 = r22, r25
     ;;
9.   psub2.sss r21 = r25, r30;
8.   padd2.sss r30 = r30, r25
      .
      .
      .
```

Figure 8: Rescheduled IA64 MMX FDCT code fragment