# REAL-TIME H.264 ENCODING BY THREAD-LEVEL PARALLELISM: GAINS AND PITFALLS

Guy Amit and Adi Pinhas
Corporate Technology Group, Intel Corp
94 Em Hamoshavot Rd, Petah Tikva 49527, PO Box 10097
Israel
{guy.amit, adi.pinhas}@intel.com

**ABSTRACT**
Real-time encoding of video streams with H.264 coding standard is a challenging task for current personal computers. In this study, thread-level parallelism was applied to an optimized H.264 encoder, achieving real-time encoding of high-definition video sequences on a quad-processor machine. The multithreaded encoder combined data decomposition at the macroblcok level with functional decomposition of serial tasks at the frame level. The resulting performance speedup was up to 3.6x on four physical processors. Analysis of the software and hardware factors that limit the speedup of the encoder indicated that the most dominant factors are miss rates of L2/L3 data caches, inter-thread synchronization overhead and the remaining sequential portions of the code. Each of these factors constituted about one third of the overall degradation from the theoretical speedup of 4x. It is concluded that hardware support of multithreading, along with optimized multithreaded software algorithms and data structures lay the foundation for significant performance enhancement of computationally-heavy media applications.

**KEY WORDS**
Video compression, H.264, multithreading, software parallelization

## 1. Introduction

H.264, also known as MPEG-4 part 10, is the latest international video coding standard [1] that addresses applications such as video telephony, storage, broadcast and streaming. Similarly to earlier MPEG and H.26x standards, H.264 is based on modules of block motion-compensation, transform, quantization and entropy coding (Figure 1). The new advanced coding tools [2] collectively provide impressive coding efficiency as well as a significant increase in the algorithmic complexity of both encoder and decoder. The H.264 baseline encoder is estimated to be 5x to 10x more complex than the H.263 encoder [3], while the decoder is 2x to 2.5x more complex than the H.263 baseline decoder [4].

In order to meet the demanding requirements of the standard, three types of solutions have been suggested by previous studies:

(a) Reduced complexity. Low complexity algorithms that are suboptimal in terms of the compressed video quality were described in [3,5]. The H.264 encoder described in [3] achieved real-time encoding of low-resolution CIF video on a Pentium® processor. However, its reduced complexity resulted in a 20% higher bit rate compared to the reference encoder.

(b) Instruction-Level Parallelism (ILP). An optimized implementation using a media instruction set was described in [6,7]. In [6], the time-consuming modules of the H.264 reference code were identified. The execution time of these modules was improved using SIMD instructions, which execute several computations in parallel with a single instruction. As a result, the entire codec was improved more than 3x. Nevertheless, the final conclusion was that H.264 encoder remains too complex to be implemented in real time on a single-core processor of a personal computer.

(c) Thread-Level Parallelism (TLP). Multithreaded implementation on a multiprocessor machine was described in [8]. This study used a non-real-time, one-frame-per-second codec. The multithreading side effects of such a codec were too small relative to the long computation time, and as a result, the encoder showed nearly linear performance speedup with the number of processors.

In the current study, we combine all three solutions mentioned above: We start with an encoder that was already well optimized using reduced complexity and ILP and significantly speed up its performance by TLP. The multithreaded encoder is capable of real-time encoding of 720p24 High Definition (HD) video (progressive 1280x720 images at a frame rate of 24Hz). To the best of our knowledge, this is the first implementation of a real-time H.264 encoder on a PC, in which the distributed video processing does not cause any degradation in either compressed video quality or bit rate.

Furthermore, as we parallelized a well-optimized codec, we were able to reveal and closely examine the side effects of multithreading. These side effects include cache performance, bus bandwidth, Amdahl's law and synchronization overhead.

The development of microprocessor architectures that provide TLP support in hardware makes TLP a very promising approach for speeding up computationally-heavy applications. Two examples are Pentium® 4 processors with Hyper-Threading (HT) technology, which allows a single physical processor to manage data as if it were two logical processors, and Pentium®-D, which is a dual-core processor, where each core supports HT technology, enabling simultaneous execution of four threads.

The remainder of the paper is structured as follows: Section 2 provides an overview of the parallelism options in H.264. The implementation is detailed in Section 3. The performance results and analysis are provided in Sections 4. Section 5 concludes this paper.

## 2. Parallelism options in H.264

### 2.1. H.264 Overview

The modules and flow of a typical H.264 encoder are illustrated in Figure 1: An input frame can be divided into multiple slices. A slice is a portion of the image that is processed independently of other slices, thus providing better recovery from stream corruption. Each slice is processed in units of 16x16 pixel patches, termed macroblcoks (MB). Each macroblock is encoded in either intra or inter mode. In intra mode, a prediction is formed from samples in the current slice that have been previously encoded, decoded and reconstructed. In inter mode, a prediction is formed by motion-compensated prediction from one or more reference pictures. The reference pictures can be selected from past and future pictures that have already been encoded. The prediction is then subtracted from the current block to produce a residual block that is transformed and quantized, to give a set of quantized transform coefficients, which are reordered and entropy encoded. The encoder also decodes (reconstructs) each macroblock to provide a reference for further predictions. A filter is applied to the reconstructed picture to reduce the effects of blocking distortion. The major new features introduced in H.264 include variable block-size motion compensation with small block sizes, quarter-sample motion vector accuracy by sub-pel interpolation, multiple reference picture motion compensation and context-adaptive entropy coding.

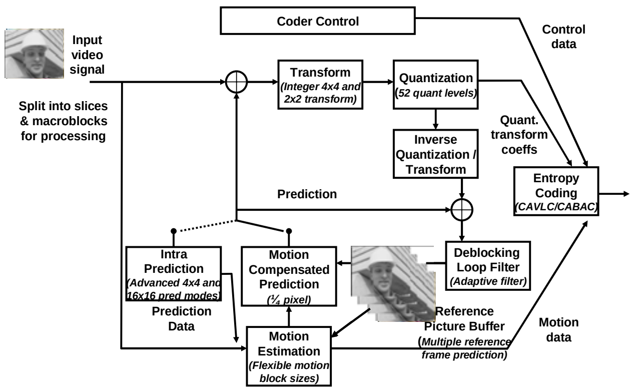

Coder Control
Input video signal
Control data
Transform (Integer 4x4 and 2x2 transform)
Quantization (52 quant levels)
Split into slices & macroblocks for processing
Quant. transform coeffs
Inverse Quantization / Transform
Entropy Coding (CAVLC/CABAC)
Prediction
Intra Prediction (Advanced 4x4 and 16x16 pred modes)
Motion Compensated Prediction (¼ pixel)
Deblocking Loop Filter (Adaptive filter)
Prediction Data
Reference Picture Buffer (Multiple reference frame prediction)
Motion data
Motion Estimation (Flexible motion block sizes)

**Figure 1: The algorithmic modules and data flow of H.264 encoder, incluidng motion estimation/prediction, transform, quantization and entropy coding. New features introduced in H.264 are indicated by italicized script.**

An encoded video sequence is composed of three types of frames: I-type frames, which are encoded in intra mode, P-type frames, which are encoded with inter prediction from previously encoded I or P-type frames, and finally, B-type frames, which use bidirectional prediction from both previous and future frames.

### 2.2. H.264 Decomposition

The execution time of most of the computationally-intensive modules in the H.264 scheme (e.g. motion-estimation, entropy coding) is data dependent and cannot be predicted. Consequently, static scheduling of the encoder's tasks is inefficient: as some areas of the picture might be harder to encode than others, partitioning of the tasks between the threads might be imbalanced, resulting in low system utilization. To better balance the threads, the number of computational tasks that can be executed concurrently should be higher than the number of threads. This way, the maximal waiting time for the last thread to complete the last computational task is reduced, and the overall processor utilization is improved. Partitioning video encoding algorithms to a large number of independent tasks is not trivial. Video-encoding algorithms search for spatial and temporal redundancy in the video stream. Each pixel value is encoded in respect to other pixels in the same picture, in previous pictures or in future pictures that have already been encoded. These dependencies impose restrictions on the parallel-processing scheme.

H.264 partitioning can be attained by using either functional or data decomposition. In functional decomposition, each thread is responsible for executing a distinct module of the encoder. The maximal number of concurrent tasks is limited by the number of functional modules in the algorithm, which is about 10 modules (Figure 1). Therefore, the number of threads is typically low, and load balancing is likely to be inefficient. Furthermore, the bandwidth of data transfer between the threads is typically high. In data decomposition, each thread performs the same operations as the other threads on different data portion. The following describes the various data decomposition options (Figure 2):

**Frame-level decomposition.** The number of frames that can be coded in parallel is determined by the sequence of frames types in the video. A typical sequence of frames is $I_1B_2B_3P_4B_5B_6P_7B_8B_9P_{10}$, (where the subscript of the frame type indicates the frame's serial order). In this sequence, only three frames can be processed concurrently, with the following order of processing: $\{I_1\}$=>$\{P_4\}$=>$\{B_2,B_3,P_7\}$ =>$\{B_5,B_6,P_{10}\}$. In the low-delay sequence $I_1P_2P_3P_4P_5P_6$, only one frame can be processed at any time.

**Slice-level decomposition.** Partitioning of a frame to multiple independent slices enables parallel processing of slices. However, slicing the image and compressing each slice independently reduce the amount of spatial redundancy that can be exploited. Therefore, the more slices in the video, the higher the bit rate of the compressed video, assuming a desirable fixed quality of the compressed video. If the bit rate is kept fixed and the allowed degradation in the compressed video quality is less than 0.3db (in terms of signal-to-noise ratio), then each picture can be divided into 4-8 slices only, resulting in a limited number of threads.

**MacroBlock- (MB) level decomposition.** In standard-definition (SD) and high-definition (HD) video there are thousands of MB in each frame. However, the level of MB-based parallelism is constrained by spatial dependencies between adjacent MBs. Each MB depends on its left, above, above-left and above-right neighbor MBs. These dependencies originate from different components of the encoding scheme: motion-vector prediction, intra-prediction and deblocking filter (Figure 3). Efficient parallel processing of macroblocks requires a scheduling algorithm that will determine the order of MB processing, given that an MB can be processed only after its dependencies have been satisfied.

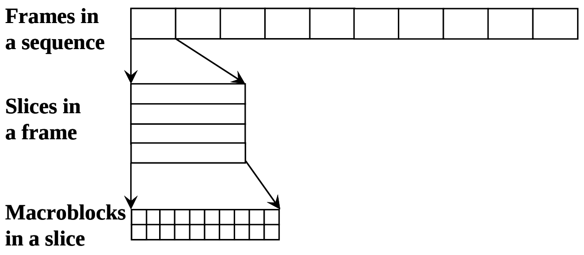


**Figure 2: Data decompoistion options - frame level, slice level and macroblock level**

In this paper, we have chosen to use MB-level decomposition, due to of the advantages of good load balancing and preserved video quality. To ensure that at any time, there is a sufficient number of macroblocks whose dependencies are satisfied, we used a 'wave-front' scheduling scheme, first described in [9]. The scheduling scheme is illustrated in Figure 4: MBs are grouped in a 'wave-front' format, rolling from upper-left corner downward. All macroblocks on the same wave-front are independent (MB with the same number in Figure 4). Their dependencies reside on previous wave-fronts, and they can therefore be processed in parallel. Note that for 4:3 or 16:9 video, this scheme restricts the maximal number of concurrent threads to *(w+1)/2*, where *w* is the horizontal number of macroblocks in a frame. Furthermore, the system utilization will be typically lower at the beginning and at the end of a frame, where the wave-front is shorter.

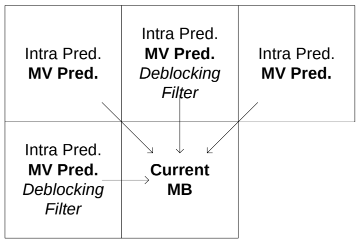


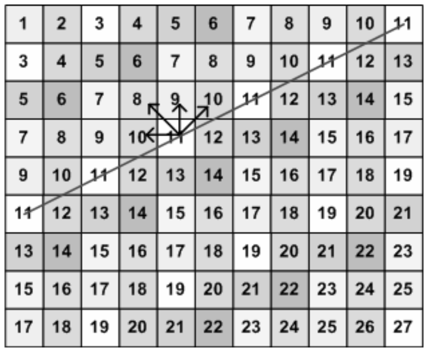
**Figure 3: Macroblock dependencies. A macroblock can be encoded after the macroblocks on its left, above, above-left and above-right have been encoded. The dependeceis are enforced by the intra-prediction, motion-vector prediction and deblocking filter modules.**

## 3. Implementation

### 3.1. Threading framework

The H.264 encoder was multithreaded using Win32 threads, according to the threading framework illustrated in Figure 5. The framework includes a single main thread, a single I/O thread and multiple worker threads. The main thread is responsible for initializing the framework, performing portions of the serial preprocessing and postprocessing logic for each frame and synchronizing with the other threads. The I/O thread performs the rest of the serial code, using a double-buffer mechanism. The worker threads perform parallel macroblock compression. Thread-safe data structures, residing in shared memory are used to coordinate the concurrent work of the worker threads. These data structures include a list of the macroblocks available for processing, counters of the remaining dependencies of each macroblock, a counter of the processed macroblocks and pointers to the current input and output memory buffers. Asynchronous events are used for inter-thread signaling. This framework is used to encode a single video frame by the following execution scenario:

1. The main thread receives an input frame from the I/O thread.
2. The main thread re-initializes the shared data structures (MB list and MB dependency counters), and signals the worker threads to start frame compression.
3. The I/O thread (concurrently) postprocesses and writes the previous output frame and then reads and preprocesses the next input frame.
4. Each worker thread waits for the MB list to be non-empty. When signaled, the worker thread pops its next MB from the list.
5. The worker thread compresses the MB and then updates the effected dependency counters and pushes newly-available MBs to the list. If there are waiting worker threads – they are signaled.
6. When all MBs in the frame have been encoded, the worker thread signals the main thread, and the scenario is repeated.



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |

**Figure 4: 'Wave-front' macroblock scheduling. Rolling from the upper-left corner downward, MBs on the same wave-front can be encoded concurrently. MB numbers indicate the processing order. Note that each MB depends only on previously-processed MBs.**

### 3.2. Macroblock scheduling

The order of macroblcok processing is generally dictated by the dependencies between adjacent macroblocks, as described in Figure 4. However, as the number of independent macroblocks that can be processed concurrently is typically larger than the number of available processors, there are several alternatives for the exact scheduling order of macroblocks. We have implemented two scheduling policies – a FIFO scheduling and locality-based scheduling. The FIFO scheduling policy handles the MB list as a queue, where macroblocks whose dependencies were satisfied are processed according to the arbitrary order of the queue. The locality-based scheduling attempts to improve data locality by letting each worker thread pop the MB that is closest (in the raster order) to the previous MB processed by the thread. In this approach, co-located parts of the picture are more likely to be encoded by the same processor, improving cache coherency.

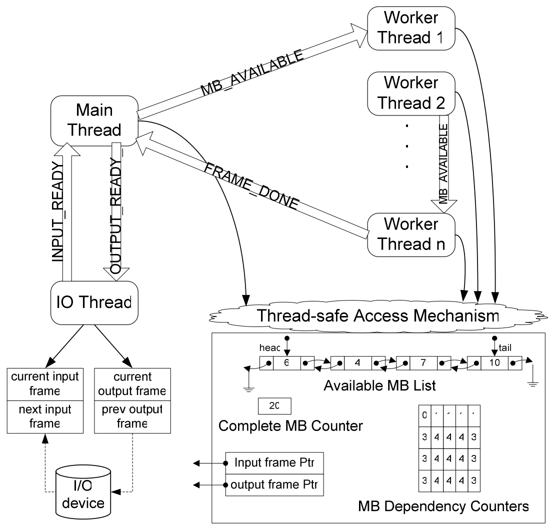


**Figure 5: MB-based multithreading architecture. A main thread, an I/O thread and multiple worker-threads synchronize using events and thread-safe shared data structures**.

### 3.3. Test setup

The performance of the multithreaded encoder was measured on an Intel® Xeon$^{TM}$ system with four processors (IBM xSeries255) running at 2.7 Ghz. Each processor has an 8KB first-level cache (L1), a 512KB second-level cache (L2) and a 2048KB third-level cache (L3) on chip. The frequency of the processors front-side bus (FSB) is 400 Mhz (100 Mhz quad data rate). Hyper-Threading technology was disabled, unless otherwise specified. The operating system was Microsoft Windows® Server 2003. Input files used for the experiments were either SD (720x480 / 640x480) or HD (1280x720 / 1920x1080) resolution, 25/30 FPS, 300-720 frames per stream. Execution time of different code portions was measured by designated functions, using accurate hardware timers. Measurements of cache and bandwidth performance were done using Intel VTune® performance analyzer.

## 4. Results & Discussion

An overall speedup ranging from 3.1x to 3.6x was achieved on the quad-processor system. Speedup results are summarized in Figure 6. The encoder's speedup increased for encoding higher video resolutions (from 3.33x for SD to 3.44x for HD). Scalability was found to be directly related to the complexity of the encoding algorithm, expressed by the presence of B-type frames between P-type frames (from 3.17x for HD-1080p encoding without B frames to 3.44x for encoding with B frames). With the Hyper-Threading feature enabled, 8 worker threads on 8 logical processors achieved higher speedup, up to 3.63x. The average compression time of a frame decreased as the number of worker threads increased (up to the number of logical processors) (Figure 7). Using 4 or 8 worker threads, the real-time boundary of 41.6 milliseconds per frame (24 frames per second) was achieved for SD sequences and HD-720p sequences.

To evaluate the time overhead imposed by the multithreading framework, the performance of the multithreaded encoder on a single-processor system were compared to the baseline single-thread encoder. The multithreaded encoder (with a single worker thread) was found to be 5% slower than the single-thread encoder.

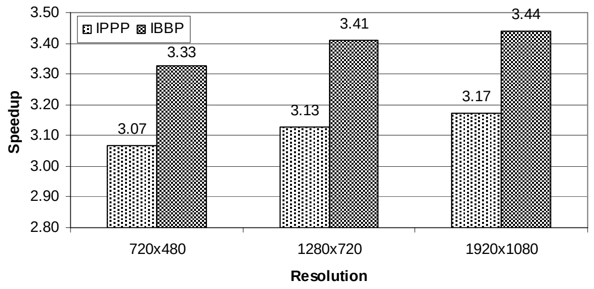


**Figure 6: Average speedup of the multithreaded encoder with four worker threads on a quad-processor system. Speedup was measured for encoding SD- and HD-resolution video, with B-type frames (IBBP sequence) and without B-type frames (IPPP sequence).**

The scalability achieved by the multithreaded encoder was less than the theoretical 4x limit. The following subsections discuss this gap in detail. The factors examined were the serial code, synchronization overhead, cache performance and memory bandwidth.

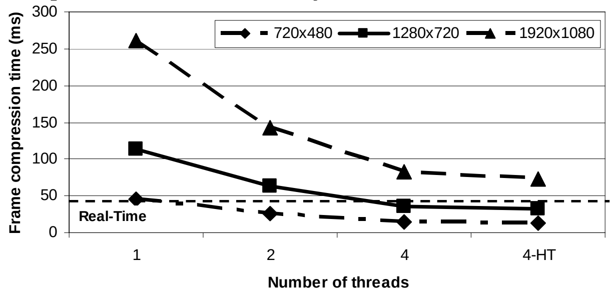


**Figure 7: Average frame compression time as a function of the number of worker threads, for SD- and HD-resolutions. Real-time line is defined as 41.6 ms (24 FPS).**

### 4.1. Serial pipelining

Profiling the code of encoding an SD sequence without B-type frames showed that the execution time of the nonscalable serial code was about 5% of total execution time. To reduce the relative portion of the serial code, it was partitioned between the main thread and a designated I/O thread that work concurrently, as illustrated in Figure 8. On the SD input mentioned above, the serial pipelining reduced the relative execution time of the serial code from about 5% to 3.5%, and the resulting speedup was improved from 2.97 to 3.07. If we denote the parallel portion of the code by *P*, the serial portions by $S=S_1+S_2=1-P$ and the effective number of utilized processors by *N*, then the serial pipelining improves the speedup, according to the Amdahl's law as shown in equation (1).

$$Speedup = \frac{1}{\frac{P}{N} + \max\{S_1, S_2\}} > \frac{1}{\frac{P}{N} + (S_1 + S_2)} \quad (1)$$

These results emphasize the asymptotic nature of the speedup derived from Amdahl's law, yielding a modest speedup improvement despite the significant reduction in the serial portion of the code.

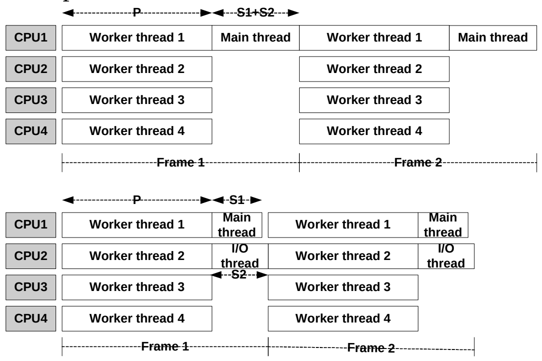


**Figure 8: (top) Execution order of the main thread and four worker threads. (bottom) Execution order of the main thread, four worker threads and the I/O thread.**

## 4.2. Synchronization overhead

Two main components comprise the time overhead imposed by the multithreading framework:

1. The time of updating the shared data structures.
2. The time of the inter-thread signaling mechanism.

When using a single worker thread, the overhead of the shared data structure updating was measured to be 1.6% of the frame compression time. This measurement included the update time, MB-dependency counters and the list of available MBs. As the number of concurrent worker threads grows, the probability of simultaneous access to the shared data from two or more threads increases, and the locking mechanism that guards the consistency of the data structures became a significant source of additional overhead. With four worker threads, this overhead was measured to be as large as 35% of the frame compression time. To avoid this major bottleneck, the number of accesses to the MB list was decreased by allowing the worker threads to 'bypass' the MB list, and independently choose the next MB from the macroblocks made available in the last cycle. As a result, the overhead of updating the shared data structures grew much more gradually with the number of threads, composing about 2.5% of the total compression time, using four worker threads.

The time overhead of the inter-thread signaling mechanism was measured as the time of sending/receiving events plus the context-switch time whenever a thread yields the CPU while waiting for work. Signaling between the main thread and a worker thread occurs once per frame, while signaling between worker threads occurs whenever there are not enough macroblocks for all threads, typically at the beginning and end of a frame. With a single worker thread, there is a single synchronization event per frame. The overhead imposed by this setting was measured to be 1.8% of the total frame compression time. With four worker threads, the average number of inter-thread synchronization events per frame was found to be about 20 for each thread (out of a total of 1200 MB in an SD-resolution frame), and the resulting overhead becomes a significant component of 10.5% of the total frame compression time. The signaling mechanism relies on system calls of the host operating system, and its overhead is therefore strictly related to the efficiency of the inter-thread communication services provided by the operating system.

## 4.3. Cache performance

The miss rates of the data caches were measured for different numbers of worker threads (Table 1). To measure the cache performance that derives from the encoding algorithm and exclude the cache pollution caused by the operating system's thread scheduling, each thread was associated to a specific processor by setting its *affinity* attribute.

The L1 load miss rate did not vary significantly. The relatively-high miss rate of L1 is a result of its small size (8KB) and the intrinsic data-access pattern of the algorithm, at sub-macroblock level. The L2 load miss rate of the multithreaded encoder, compared to the single-thread encoder, was higher by an average of 4.5K misses per frame for each processor. Given that the additional latency resulting from a data miss in L2 is 45 cycles, this difference is insignificant compared to the frame encoding time. The degradation in L3 performance is more prominent, with an average of 7.7K more misses per frame for each processor. As the latency for accessing the external memory is 230 cycles, the increased L3 miss rate can account for up to 5% of the frame encoding time (for an SD-resolution input). Cache performance is therefore a major scalability-limiting factor.

The cache performance with different macroblock scheduling schemes, as described in section 3.2, produced the same hit rates, suggesting that the cache behavior is dominated by low-level functions that process single macroblocks and not by the higher level scheduling at the frame level.

| Worker threads | L1 load miss rate (%) | L2 load miss rate (%) | L3 read miss rate (%) |
|---|---|---|---|
| 1 | 8.4 | 4.1 | 52.6 |
| 4 | 8.6 | 5.2 | 77.1 |

**Table 1: Cache miss rates in L1,L2 and L3 caches, using either one or four worker threads. Miss rates are calculated for each cache level as the number of cache misses divided by the number of cache accesses. Input file is SD resolution.**

### 4.4. Memory bandwidth

To test whether FSB bandwidth is a scalability-limiting factor, the effect of the number of threads on the shared bus bandwidth and on the average bus latency was analyzed. As shown in Figure 9 (bottom curve), the bandwidth utilization of the multithreaded encoder showed sublinear direct relation to the number of threads, with maximal values that are lower than 7% of the total FSB bandwidth. The average latency of bus read operations did not increase with the number of threads (Figure 9, top curve). The memory bandwidth was therefore concluded to have a minor effect on the scalability of the algorithm.

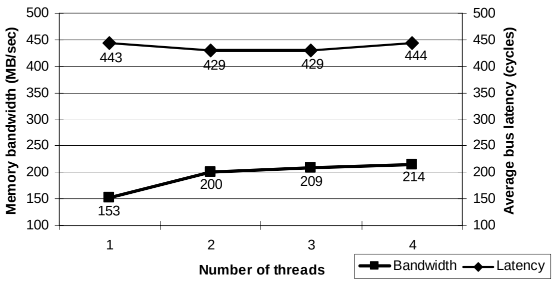


**Figure 9: Memory bandwidth (top curve) and average bus latency (bottom curve) of a multithreaded encoder with 1-4 worker threads**

### 4.5. Analysis of speedup degradation

The scalability of the multithreaded framework, when the number of worker threads is increased from one to four is limited by a collection of factors, each contributing to the overall speedup degradation from the theoretical linear speedup of 4x to the actual speedup of 3x to 3.6x. These factors and their effect on the speedup, analyzed for an SD input, are illustrated in Figure 10. The wave-front scheme imposes submaximal utilization at the corners of the frame. For an SD-resolution input, the maximal speedup is therefore 3.97. When measuring the net time of the core function that compresses a single macroblock (data not presented), the speedup on four processors is 3.62. This decrease, contributing 38% of the total speedup degradation, is due primarily to a lower cache hit rate. The implemented multithreading framework imposes overheads in thread synchronization and shared data structure. These overheads are mainly due to the concurrent processing of the worker threads, with an additional contribution by the synchronization mechanism between the main thread and the worker threads, summing up to 36% contribution, and a resulting speedup of 3.28. The last factor in the graph is simply the effect of Amdahl's law due to the remaining 3.5% of serial code, which accounts for 23% of the speedup degradation.

## 5. Conclusions

In this paper we have shown that thread-level parallelism of H.264 encoder, applied at a fine-grained level of macroblocks, can speed-up performance up to 3.6x, achieving real-time performance on HD video sequences. Nonetheless, as we approach the real-time barrier, the scalability of the algorithm becomes more significantly limited by a combination of hardware and software factors. Our experimental results indicate cache performance, synchronization overhead and serial code fractions as the dominant speedup-limiting factors. Further work is required in order to evaluate potential ways of reducing the effects of these factors either by µ-architecture mechanisms (e.g. cache organization) or by software optimization (e.g. lock-free shared data structures).

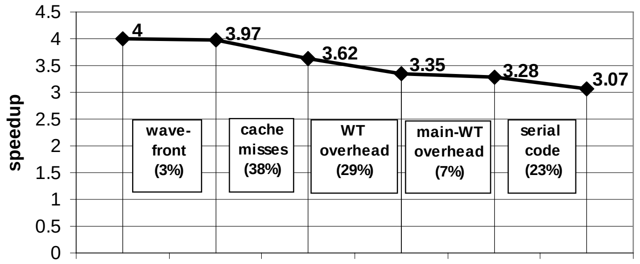


**Figure 10: The relative contribution wave-front scheduling, cache misses, synchronization overhead and serial code to the degradation of the multithreaded encoder's speedup from theoretical 4x to actual 3.07x, on an example SD-resolution input.**

## References:

[1] ITU-T Rec. H.264 | ISO/IEC 14496-10 AVC, Document JVT-D157, *4th Meeting: Klagenfurt, Austria,* July 2002.

[2] T. Wiegand, G.J. Sullivan, G. Bjntegaard, & A. Luthra, Overview of the H.264/AVC video coding standard, *IEEE Transactions on Circuits and Systems for Video Technology, 13(7),* 2003, 560-576.

[3] V. Iverson, J. McVeigh, & B. Reese, Real-time H.264/AVC codec on Intel architectures, *Proc. of the IEEE International Conference on Image Processing, Vol. 2,* 2004, 757-760.

[4] M. Horowitz, A. Joch, F. Kossentini, & A. Hallapuro, H.264/AVC baseline profile decoder complexity analysis, *IEEE Transactions on Circuits and Systems for Video Technology, 13*(7), 2003, 704-716.

[5] C. Kim, & C.J. Kuo, Fast Intra/Inter mode decision for H.264 encoding using a risk-minimization criterion. *Proc. of the SPIE, Vol. 5558,* 2004, 536-546.

[6] Y.K. Chen, E.Q. Li, X. Zhou, & S. Ge, Implementation of H.264 Encoder and Decoder on Personal Computers, *To appear in the Journal of Visual Communication and Image Representation,* 2005.

[7] J. Lee, S. Moon & W. Sung, H.264 decoder optimization exploiting SIMD instructions. *Proc. of the IEEE Asia-Pacific Conference on Circuits and Systems, Vol. 2,* 2004, 1149-1152.

[8] S. Ge, X. Tian. & Y.K. Chen, Efficient multithreading implementation of H.264 encoder on Intel hyper-threading architecture, Proc. of the *IEEE Pacific-Rim Conference on Multimedia, Vol.* 1, 2003, 469-473.

[9] E.B. van der Tol, E.G. Jaspers, & R.H. Gelderblom, Mapping of H.264 decoding on a multiprocessor architecture, *Proc. of the SPIE, Vol. 5022*, 2003, 707-718.