THE IMPACT OF SMT/SMP DESIGNS ON MULTIMEDIA SOFTWARE ENGINEERING - A WORKLOAD ANALYSIS STUDY

Yen-Kuang Chen, Rainer Lienhart, Eric Debes, Matthew Holliman, and Minerva Yeung
Microprocessor Research Labs, Intel Corporation

ABSTRACT

This paper presents the study of running several core multimedia applications on a simultaneous multithreading (SMT) architecture and derives design principles for multimedia software engineering. The multimedia workloads range from memory to computational-bounded kernels. A performance metric to evaluate effective SMT performance gain is introduced, and compared to similar metrics on symmetric multiprocessor (SMP) systems. In addition, we analyze and compare SMT versus SMP systems, and highlight the advantages in the studied applications. The results indicate that sharing the cache in SMT processors can provide better cache locality and thus better performance although sharing the cache can introduce cache conflicts and reduce the actual cache size available for each logical processor. We also propose "mutually beneficial prefetching" -- a technique to schedule threads so that they prefetch data for each other in order to reduce cache miss penalty.

1. INTRODUCTION

While processors nowadays are much faster than they used to be, the rapidly growing complexity of such designs also makes achieving significant additional gains more difficult. Consequently, processors/systems that can run multiple software threads have received increasing attention as a means of boosting overall performance. In this paper, we first characterize the workloads of video decoding, encoding, watermarking, and machine learning on current superscalar architectures, and then we characterize the same workloads on simultaneous multi-threading (SMT) architectures. Specially, we use Intel® Xeon™ processors with Hyper-Threading Technology, which is one implementation of the SMT architecture. Our goal is to provide a better understanding of performance improvements on SMT processors.

Figure 1 shows a high-level view of an SMT processor and compares it to a dual-processor system. In an SMT processor, one physical processor exposes two logical processors. Similar to a dual-core or dual-processor system, a SMT processor appears to an application as two processors. Two applications or threads can be executed in parallel. The major difference between SMT processors and dual-processor systems is the different amounts of duplicated resources. In Intel Xeon processors with Hyper-Threading Technology, only a small amount of the hardware resources are duplicated, while the front-end logic, execution units, out-of-order retirement engine, and the memory hierarchy components are shared. Thus, compared to processors without Hyper-Threading Technology, the die-size is increased by less than 5% [7]. Simultaneous multi-threading architectures may increase the latency of some single-threaded applications, but have the benefit of increasing overall throughput of multi-threaded and multi-process applications.

Multimedia applications tend to exhibit large amounts of computation and parallelism. We select a few representative multimedia applications as our workloads and characterize their performance on SMT. Although the workloads are well optimized for Pentium® 4 processors, due to the inherent constitution of the algorithms, almost all multimedia applications cannot fully utilize all the execution units available in the microprocessor. Some of the modules are memory-bounded, while some are computation-bounded.

There is also no common metric to measure the efficiency of multithreaded applications on Hyper-Threading Technology yet. We propose a metric that helps us to clearly identify available performance improvements. By understanding the performance improvements that are possible in multimedia applications, we learn a number of techniques in algorithms and applications to achieve better performance on Hyper-Threading Technology.

The paper is organized as follows. Section 2 describes our workloads, while Section 3 explains the performance speedup of our multi-thread media workloads on SMT processor and dual-processor systems. In Section 4, assuming the amount of parallelism is the same on SMT processors and systems with multiple processors, we formulate a common performance metric to measure effective SMT speedup. In Section 5, after examining the performance numbers, we provide our observations and describe some techniques to increase application performance on processors with Hyper-Threading Technology. Section 6 concludes this work.

®Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

™Xeon processor is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

®Pentium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

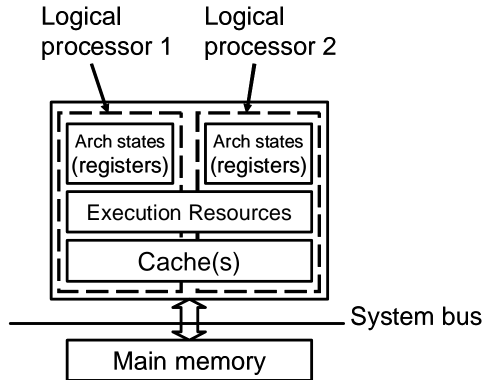
Table 1: MPEG decoding kernel characterization on 2 GHz Pentium 4 processor (9 Mb/s MPEG-2, 720x480)

Kernel	IPC	UPC	MMX, SSE, SSE-2 per instructions	Cond. Branch/ instr.	Mispred. Cond./ Instr.	Mispred. Cond./ Clock	L1 misses/ Instr.	FSB activity
VLD	0.76	0.99	0.074	1/9	1/120	**1/158**	1/92	11.1%
IDCT	0.59	0.89	**0.90**	1/141	1/2585	1/4381	1/193	2.4%
MC	0.24	0.40	0.42	1/17	1/142	1/592	1/11	**30. 3%**

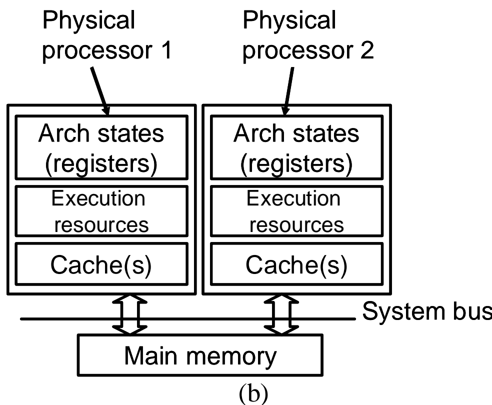
2. WORKLOADS

2.1. MPEG Video Encoding/Decoding

The MPEG decoding pipeline consists of the major operations of Variable-Length Decoding (VLD), Inverse Quantization (IQ), Inverse Discrete Cosine Transform (IDCT), and Motion Compensation (MC) [4], as shown in Figure 2. Table 1 shows a high-level summary of the MPEG-2 decoder's behavior. The first stage of the decoding pipeline, VLD/IQ, is characterized by substantial data dependency, limiting opportunities for instruction, data, and thread-level parallelism. The next stage, IDCT, is completely computation-bound. The kernel is dominated by MMX/SSE/SSE2 (Streaming SIMD Extension) operations. Because 90% of the instructions are executed in the MMX/SSE/SSE2 unit, the integer execution unit is idle most of the time in the IDCT module. The final stage of the decoding pipeline, MC, is memory intensive compared to the other modules in the pipeline. Although the out-of-order execution core in the Intel Xeon processor can tolerate some memory latency, the module shows an equal distribution of time between computation and memory latency because there are too many memory operations. All these modules are well-optimized, but still cannot utilize all of the execution units available in the microprocessors. While the Intel Pentium 4 and Intel Xeon processors can execute multiple uops in one cycle, the uops retired per cycle (UPC) is only 0.74 in our optimized video decoder. After multi-threading these workloads on SMT processors, they show better utilization of the execution units.



(a)



(b)

Figure 1: High-level diagram of (a) an SMT processor and (b) a dual-processor system.

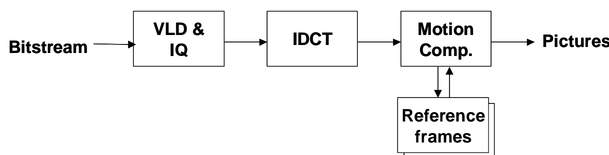


Figure 2: Block diagram of an MPEG decoder.

2.2. Watermark Detection

Another application that we studied is video watermark detection. Our watermark detector has two basic stages: MPEG video decoding (as described in Section 2.1) and image-domain watermark detection. The watermark detection scheme is optimized with the Intel IPL for the image manipulations used during watermark detection.

2.3. Support Vector Machines

Machine learning plays a key role in automatic content analysis of multimedia data [10]. A common task is to predict the output y for an unseen input sample x given a training set $\{(x_i, y_i)\}_{i\in\{1,\dots,N\}}$ consisting of input x_i and its desired output y_i. In other words, the goal is to learn the functional relationship F: $y = F(x)$ between input x and output y.

Predicting qualitative output is called *classification*, while predicting quantitative output is called *regression*. In our investigations, we concentrate on two recent machine learning techniques for classification: Support vector machines (SVMs) [1] and Boosting [3]. Both techniques are quite different. Together, however, they show off execution features of many machine learning algorithms, and are thus good workloads to be analyzed.

The evaluation of trained SVMs is very structured and can, thus, be multithreaded at multiple levels (see Figure

3): On the lowest level, the dimensionality K of the input data can be very large. Typical values of K range between a few hundreds to several thousands. Thus, the vector multiplication in the linear, polynomial and sigmoid kernels as well as the L_2 distance in the radial basis function kernel can be multithreaded. On the next level, the evaluation of each expression in the sum is independent of each other. Finally, in an application several samples are tested and each evaluation can be done in parallel. In the experimental results section we will research the effects of the different level of parallelism.

$$F(\mathbf{x}) = \operatorname{sign}\left[\left(\sum_{i=1}^{N} y_i \alpha_i \Phi(\mathbf{x}, \mathbf{x}_i)\right) + b\right]$$

where $\mathbf{x}, \mathbf{x}_i \in \Re^K$, $y_i \in \{-1,+1\}, \alpha_i, b \in \Re$, and $\Phi(\mathbf{x}, \mathbf{x}_i)$ is one of the following kernels:

- Linear: $\Phi^L(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j$
- Polynomial of degree d: $\Phi^P(\mathbf{x}_i, \mathbf{x}_j) = \left(\mathbf{x}_i^T \mathbf{x}_j + 1\right)^d$
- Sigmoid kernel: $\Phi^S(\mathbf{x}_i, \mathbf{x}_j) = \tanh\left(a \mathbf{x}_i^T \mathbf{x}_j + c\right)$
- Radial basis function:
  $\Phi^{RBF}(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\left\|\mathbf{x}_i - \mathbf{x}_j\right\|^2 / \sigma^2)$

Figure 3: Support Vector Machine (SVM) classification algorithm [1]

Figure 4 and 5 show two different SVM implementations. The first implementation will result in mutually beneficial prefetching, while the latter implementation will make better use of the caches.

```
const int NUM_SUPP_VEC = 1000; // Number of support vectors
const int NUM_VEC_DIM = 24*24; // Feature vector size; 24 by 24
pixel window
// 1D signal scanned by sliding window for faces of size 24 by
24 pixels
const int SIGNAL_SIZE = 320*240;
const int NUM_SAMPLES = SIGNAL_SIZE-NUM_VEC_DIM+1;

Ipp32f supportVector[NUM_SUPP_VEC][NUM_VEC_DIM];
Ipp32f coeffs [NUM_SUPP_VEC];
Ipp32f samples[SIGNAL_SIZE]; // input signal array
Ipp32f result [NUM_SAMPLES]; // stores classification result

// LINEAR KERNEL
float linear_kernel(const Ipp32f* pSrc1, int len, int index)
{
  Ipp32f result;
  ippsDotProd_32f(pSrc1, supportVector[index], len, &result);
  return result * coeffs[index];
}

// non-blocking, mutually beneficial prefechting code
int main() {
  #pragma omp parallel for
  for (int j=0 ; j<NUM_SAMPLES; j++) {
    float sum=0;
    #pragma omp parallel for reduction (+:sum)
    for (int i=0 ; i<NUM_SUPP_VEC; i++) {
       float tmp = linear_kernel(&samples[j], NUM_VEC_DIM, i);
       sum += tmp;
    }
    result[j] = sum;
}}
```

Figure 4: Standard multi-threaded SVM implementation (with mutually beneficial prefetching)

```
// blocking code, block size = supVecBlockSize
int main() {
  int blockSize = ...;
  for (int jj=0 ; jj<NUM_SAMPLES; jj+=blockSize) {
    for (int i=0 ; i<NUM_SUPP_VEC; i+=1) {
      int loopEnd_j = std::_MIN(NUM_SAMPLES, jj+blockSize);
      #pragma omp parallel for
      for (int j=jj ; j<loopEnd_j ; j++) {
        float tmp = linear_kernel(&samples[j], NUM_VEC_DIM,
i);
        result[j] += tmp;
}}}}
```

Figure 5: Re-arranged multi-threaded code with higher cache localities (blocked SVM)

2.4. Boosting

Boosting is a powerful learning concept. It combines the performance of many "weak" classifiers to produce a powerful 'committee' [3]. A weak classifier is only required to be better than chance, and thus can be very simple and computationally inexpensive. Many of them smartly combined, however, result in a strong classifier, which often outperforms most 'monolithic' strong classifiers such as SVMs and Neural Networks.

Different variants of boosting are known such as Discrete Adaboost, Real AdaBoost, LogitBoost, and Gentle AdaBoost [3]. All of them are almost identical from a workload perspective. Therefore, we will look only at the standard two-class Discrete AdaBoost algorithm as shown in Figure 6.

Learning is based on N training examples $\{(\mathbf{x}_i, y_i)\}_{i \in \{1,\ldots,N\}}$ with $\mathbf{x}_i \in \Re^K$ and $y_i \in \{-1,+1\}$. $\mathbf{x}_i$ is a K-component vector. Each component encodes a feature relevant for the learning task at hand. The desired two-class output is encoded as –1 and +1. In the case of face detection, the input pattern x could be a raw bitmap, and an output of +1 and -1 would indicate whether the input pattern does contain a complete face.

1. Given N examples $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$ with $\mathbf{x}_i \in \Re^k$, $y_i \in \{-1,+1\}$.
2. Start with weights $w_i = 1/N$, i = 1, ..., N.
3. Repeat for m = 1, ..., M
   a. Fit the classifier $f_m(\mathbf{x}) \in \{-1,+1\}$ using weights w_i on the training data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$.
   b. Compute $err_m = E_w[1_{(y \neq f_m(\mathbf{x}))}]$, $c_m = \log((1 - err_m)/err_m)$.
   c. Set $w_i \leftarrow w_i \cdot \exp(c_m \cdot 1_{(y \neq f_m(\mathbf{x}))})$, i = 1, ..., N, and renormalize weights so that $\sum_i w_i = 1$.

4. Output the classifier $F(\mathbf{x}) = \mathrm{sgn}(\sum_{m=1}^{M} c_m \cdot f_m(\mathbf{x}))$

Figure 6: Two-class Discrete AdaBoost algorithm: Training (step 1 to 3) and evaluation (step 4) [3]

Each sample is initially assigned the same weight (step 2). Next a weak classifier f_1 is trained on the weighted training data (step 3a). Its weighted training error and scaling factor c_m is computed (step 3b). The weights are increased for training samples, which have been misclassified (step 3c). All weights are then normalized, and the process of finding the next week classifier continues for another M-1 times. The final classifier F(x) is the sign of the weighted sum over the individual weak classifiers f_m (step 4).

All training steps (2 and 3) in Figure 2 can be partitioned to run in parallel:

- Step 2: Every initial weight assignment is independent of each other, and can thus be done in parallel.
- Step 3a: This step is computationally most demanding. Fortunately, most weak classifiers can be trained in parallel. In our research, we use stumps as weak classifiers. A stump is a simple threshold classifier of the form

$$f^k(\mathbf{x}) = \begin{cases} \sum_{x_i^k \le threshold^k} y_i \Big/ \left|\{\mathbf{x}_i^k \le threshold^k\}\right| & if(\mathbf{x}^k \le threshold^k) \\ \sum_{x_i^k > threshold^k} y_i \Big/ \left|\{\mathbf{x}_i^k > threshold^k\}\right| & else \end{cases}$$

  where $\boldsymbol{x}^k$ denotes the k–th component of vector $\boldsymbol{x}$ and |.| the size of the set {…}. The threshold for the best classification performance must be calculated for all K components of $\boldsymbol{x}$ in order to find the best weak classifier:

$$(k_{best}, threshold) = \arg \max_{k, theshold^k} \{E[1_{y=f^k(x)}]\}.$$

  This search over k is independent and can thus be easily parallelized.
- Step 3b: In this step there are only two minor dependencies: the count of the misclassified samples and the final calculation of c_m based on this count. The needed classification of each training sample in contrast is totally independent. Note that the misclassified samples can be counted independently in multiple threads. Only the final accumulation of all partial counts has to be done in serial in order to calculate c_m (reduction operation).
- Step 3c: The picture here is similar to 3b. Weight updating is independent in the samples. Only the accumulation of the new weights must either be performed synchronized or partially in each thread (reduction operation). It is followed by a non-parallel calculation of the inverse of the weight sum as the new normalization factor. Normalization of each weight is independent in the samples.

Evaluation can be multithread at multiple levels. On a sample level, the different weak classifiers can be evaluated and scaled independently. However, often many samples have to be evaluated. Therefore on this higher level, each test sample can be evaluated individually, too. In face detection, for instance, a window slides over the whole image at multiple resolutions. At each position the classifier is evaluated. This can be easily parallelized.

3. PERFORMANCE CHARACTERISTICS

This section shows the performance analysis of our applications on multithreading architectures. In general, our results show that SMT offers an average 16%~25% performance improvement. This is very cost effective, as the 5% area cost for SMT is far less than the cost of doubling the hardware for dual-processor systems.

Our SMT and dual-processor system has two 2.0GHz Intel Xeon processors with Hyper-Threading Technology, running Windows XP. Each processor has a 512KB second-level cache. To contrast the performance with single-thread performance on the system experimentally in lab setting, we disable one physical CPU and the support of Hyper-Threading Technology for the other CPU. To contrast the performance of dual processors, we disable the support of Hyper-Threading Technology for both CPUs.

Table 2 shows our experimental results. We achieve consistently more than 10% higher performance on the processor with Hyper-Threading Technology across several workloads. The speedups reported in the second column are the workload speedups on a single processor with Hyper-Threading Technology. The speedups in the third column are the workload speedups on dual-processor systems. (We will explain the last column in the next section.)

Multithreaded performance is better due to more efficient use of the execution units. To verify that resource utilization is better balanced on a processor with Hyper-Threading Technology, we compare UPC for single-threaded and multi-threaded applications. UPC increases from 1.05 to 1.33 in video encoding, from 0.78 to 0.85 in video decoding, and from 1.01 to 1.21 in watermark detection, confirming the more efficient resource utilization possible with Hyper-Threading Technology, as shown in Table 3.

4. EFFECTIVE SMT SPEEDUP

Before we continue our discussion on SMT speedup, this section formulates our performance evaluation criterion of

the workloads on SMT processors. As mentioned earlier, SMT processors only duplicate a small amount of the resources while dual-processor systems duplicate almost every processor resource. Thus, it makes little sense to compare speedups on SMT processors with those on systems with two processors [8]. In order to have a fair performance evaluation of multi-threaded applications on SMT architectures, we define a metric to measure the effective speedup. This metric can be extended to measure not only the effective speedup of SMT processors, but also those of other multi-threading capable processors or systems, such as chip multi-processor (CMP) architectures.

For example, consider two workloads with a 1.17x and 1.11x speedup on Hyper-Threading Technology (the blocked versions of the support vector machine workload with RBF kernel and linear kernel, as shown in Table 2). The same two workloads show speedups on dual-processor systems of 1.78x and 1.54x, respectively. Obviously the first workload has more parallelism in the code and thus exhibits better speedup on both SMT and SMP systems. However, if we take the ratio of the Hyper-Threading speedup over the dual-processor speedup, we observe that $\frac{1.17}{1.78} < \frac{1.11}{1.54}$, which seems to suggest that the second workload would better exploit the SMT architecture. In fact, counter-intuitively, we observed that the SMT system actually better exploits the parallelism in the *first* workload. Thus comparing ratios of SMT/SMP performance does not necessarily tell the whole story when it comes to assessing the gains achieved via the SMT architecture. This motivates us to define a new metric that is independent of the amount of parallelism available in the code, in order to assess the gains achieved from SMT for a given workload.

According to Amdahl's Law the speedup of a program on a multi-threaded system is:

$$Speedup = \frac{1}{(1 - parallelism) + \frac{parallelism}{ParallelSpeedup}},$$

where *Speedup* is the overall application speedup, *parallelism* is the fraction of code that can be executed in parallel, and *ParallelSpeedup* is the effective speedup achieved for the parallel section. The formulation is true for both SMT and SMP. We would like to use the effective speedup as the metric to compare the performance of an application on different multi-threading architectures.

By measuring the application speedup on an SMT processor and combining that with knowledge of the amount of available thread level parallelism, we can quantify the effective speedup due to SMT, which we denote as $ParallelSpeedup_{MT}$. Since computational resources are shared between the parallel threads, this value is unpredictable and dependent upon the workload itself. We thus determine it as follows.

We assume that *parallelism* is identical for both SMP and SMT systems. Given an application's overall speed-up on an SMP system, we estimate the amount of parallelism achievable under the simplifying assumption that the speedup of the parallel code is approximately a factor of two on dual-processor systems. Under these assumptions, the effective SMT speedup of the workload can be expressed as the following:

$$ParallelSpeedup_{MT} = \frac{2 - \frac{2}{Speedup_{DP}}}{1 + \frac{1}{Speedup_{MT}} - \frac{2}{Speedup_{DP}}}$$

where $Speedup_{MT}$ and $Speedup_{DP}$ are the overall application speedups on the SMT processor and the dual-processor system.

Using the above equation, we measure the effective SMT speedups as shown in the last column in Table 2. We consistently achieve at least 15-30% higher performance on the SMT processor for threaded code across several workloads. For the SVM workloads with mutual prefetching (discussed in the next session), even better results are obtained.

Our metric shows that a multi-threaded workload that has a 1.17x Hyper-Threading speedup and a 1.78 dual-processor speedup (SVM RBF) uses Hyper-Threading more efficiently than another workload that has a 1.11x Hyper-Threading speedup and a 1.54 dual-processor speedup (SVM linear). The first obtains 20% improvement for threaded code, while the latter obtains a 16% speedup for threaded sections. Thus despite the initial intuition that the latter workload benefits more from Hyper-Threading, in fact we see that the former workload shows a greater relative performance gain.

5. DISCUSSION

There are a number of factors that can influence the effective SMT speedup. For example, because two logical processors share one physical SMT processor, the effective sizes of the caches for each logical processor, largely influenced by the cache footprint of each application, seem smaller. Thus, it is important for multi-threaded applications to make judicious use of the cache and comprehend possible thrashing issues. For instance, when considering code size optimization, excessive loop unrolling should be avoided. Please refer to [6] for more details.

While sharing caches may decrease the effective cache sizes seen by some applications running on processors with Hyper-Threading Technology, sharing caches can provide better cache locality between the two logical processors for other applications. As many applications are memory-bound, having good cache locality can provide a significant increase in application performance. For example, in our study we identified two cases in which cache sharing between logical processors on one physical SMT processor produced very beneficial data locality effects. This is the reason why we observe a large speed-up for some threaded workloads on an SMT processor.

5.1. Dynamic Slice Scheduling

As shown in Figure 7, a picture in a video bit stream can be divided into slices of macroblocks. Each slice, consisting of blocks of pixels, is a unit that can be decoded independently. Here we compare two methods to decode the pictures in parallel:

1. Static partitioning: In this method, one thread is statically assigned the first half of the picture, while another thread is assigned the other half of the picture (as shown in Figure 7(a)). Assuming that the complexity of the first half and second half are

Table 2: Speedups of our media workloads on systems with Hyper-Threading Technology & systems with dual-processors, and, the effective Hyper-Threading Speedup.

	Workload Speedup Hyper-Threading vs. Single-Thread	Workload Speedup Dual-processors vs. Single-processor	Effective SMT Speedup
Video Encoder [2, 4]	1.12	1.61	1.16
Video Decoder [2, 4]	1.21	1.61	1.30
Video Watermarking [2]	1.16	1.64	1.21
SVM linear [1] (blocked)	1.11	1.54	1.16
SVM RBF [1] (blocked)	1.17	1.78	1.20
SVM linear [1] (mutally beneficial prefetching)	1.60	1.64	1.92
SVM RBF [1] (mutally beneficial prefetching)	1.58	1.73	1.77
Boosting Training [3]	1.24	1.94	1.25
Boosting Detection [3]	1.11	1.63	1.15

Table 3: The workload characteristics of our applications on single-threaded processors and processors with Hyper-Threading Technology

Event	MPEG encoding		MPEG decoding		Video watermarking	
	Single-thread	Hyper-threading	Single-thread	Hyper-threading	Single-thread	Hyper-threading
Clockticks (Millions)	13,977	11,688	7,467	6,687	23,942	20,162
Instructions retired (Millions)	11,253	11,674	3,777	3,921	17,728	17,821
Uops retired (Millions)	14,735	15,539	5,489	5,667	24,120	24,333
IPC (instructions per clock)	0.80	1.00	0.51	0.59	0.74	0.88
UPC (uops per clock)	1.05	1.33	0.74	0.85	1.01	1.21
Floating-point/SIMD (Millions)	6,226	6,220	1,119	1,120	5,334	5,341
L1 cache misses (Millions)	132	145	132	166	510	638
Front Side Bus utilization rate	8.5%	8.5%	14.7%	16.4%	14.2%	22.3%

Event	SVM (linear) with mutually beneficial prefetching		SVM (RBF) with mutually beneficial prefetching		Boosting Training	
	Single-thread	Hyper-threading	Single-thread	Hyper-threading	Single-thread	Hyper-threading
Clockticks (millions)	12,758	6,901	13,943	8,627	5,995	4,852
Instructions Retired (millions)	3,158	3,162	4,367	4,357	4,077	4,215
Uops Retired (millions)	5,393	5,423	7,563	7,571	5,178	5,677
IPC (instructions per clock)	0.25	0.46	0.31	0.51	0.68	0.87
UPC (uops per clock)	0.42	0.79	0.54	0.88	1.27	1.35
Floating-point/SIMD (millions)	1,778	1,767	2,890	2,888	1,039	886
L1 cache misses (millions)	901	559	686	432	192	234
Front Side Bus utilization rate	47.4%	56.2%	43.3%	43.1%	0.3%	0.1%

similar, these two threads will finish the task at roughly the same time. However, some areas of the picture may be easier to decode than others. This may lead to one thread being idle while the other thread is still busy.

2. Dynamic partitioning: In this method, slices are dispatched dynamically. A new slice is assigned to a thread when the thread has finished its previously assigned slice. In this case, we don't know which slices will be assigned to which thread. Instead, the assignment depends on the complexity of the slices assigned. As a result, one thread may decode a larger portion of the picture than the other if its assignments are easier than those of the other thread. The execution time difference between two threads, in the worst case, is the decoding time of the last slice.

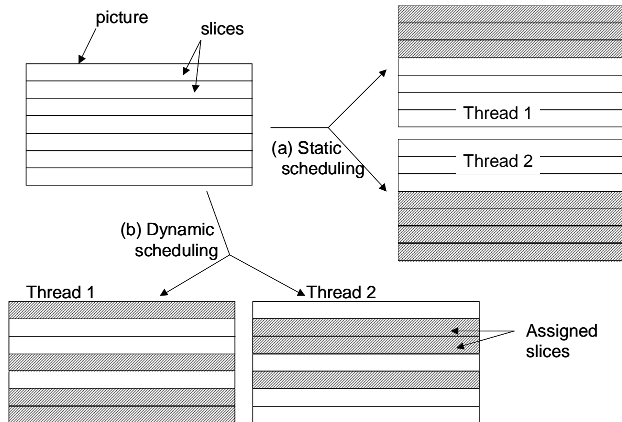


Figure 7: Two slice-based task partitioning schemes between two threads: (a) static scheduling and (b) dynamic scheduling.

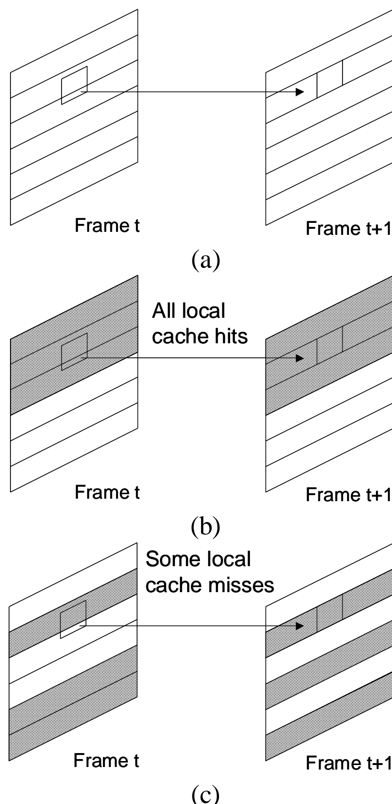


Figure 8: Cache localities, during (a) motion compensation, in (b) static partitioning, and in (c) dynamic partitioning.

The foremost advantage of the dynamic scheduling scheme is its good load balance between the two threads. Because some areas of the picture may be easier to decode than others, one thread under the static partitioning scheme may be idle while another thread still has a lot of work to do. In the dynamic partitioning scheme, we have very good load balance. As we assign a new slice to a thread only when it has finished its previous slice, the execution time difference between the two threads, in the worst case, is the decoding time of a slice.

We now illustrate the advantage of sharing caches in our application. On dual-processor systems, each processor has a private cache. Thus, there may be a drawback to dynamic partitioning in terms of cache locality. Figure 8 illustrates the cache locality in multiple frames of video. During motion compensation, the decoder uses part of the previous picture, the referenced part of which is roughly co-located in the previous reference frame, to reconstruct the current frame. It is faster to decode the picture when the co-located part of the picture is still in the cache. In the case of a dual-processor system, each thread is running on its own processor, each with its own cache. If the co-located part of the picture in the previous frame is decoded by the same thread, it is more likely that the local cache will have the pictures that have just been decoded. Since we dynamically assign slices to different threads, it is more likely that the co-located portion of the previous picture may not be in the local cache when each thread is running on its own physical processor and cache, as shown in Figure 8(c). More cache misses may incur more bus transactions. In contrast, the cache is shared between logical processors on a processor with Hyper-Threading Technology and thus cache localities are preserved. We obtain the best of both worlds with dynamic scheduling: there is load balancing between the threads, and there is the same effective cache locality as for static scheduling on a dual-processor system.

5.2. Mutually Beneficial Prefetching

Although the Intel Xeon processor can execute multiple uops in one cycle, the number of retired uops per cycle (UPC) is only 0.42 and 0.54 for the single-threaded SVM workloads [1]. Low UPC in the single-threaded workload indicates the underutilization of the execution units available in the microprocessor [5] due to the high rate of L1/L2 misses. The large number of L1/L2 misses result

from the large working set of the SVM workloads. The size of the whole set of support vectors is around 2MBytes, which is larger than the L2 caches of most modem microprocessors. Although out-of-order execution can reduce the problem of data dependency on cache misses, memory latencies are too long to be hidden in this particular workload. As a result, execution units are under-utilized. Sharing execution units therefore increases the utilization rate of the resources and thus improves the workload's throughput.

With Hyper-Threading Technology, the UPC increases to 0.79 and 0.88, respectively. Both threads are prefetching data for each other on a single SMT processor. In [9], Wang et al. use one thread to prefetch data for another thread on SMT systems when caches are shared. In their work, the prefetching thread does not generate useful results.

In the SVM workloads, both threads require the same support vectors for different incoming samples at roughly the same time. Thus, when one thread runs faster than the other thread, it will access the L2 or main memory to get the next support vector. So, when the other thread catches up in the execution, the support vector is already in L1. In this case, both threads are prefetching data for each other on a single SMT processor. We call this mutual prefetching.

Because of this mutual prefetching effect, the dual-threaded workload on the SMT processor has fewer L1/L2 misses than the single-threaded workload and its performance is thus much better. In fact, the speed-up is almost the same as that achieved on a dual-processor system (see Table 2).

6. CONCLUSIONS

In this paper, the characteristics of several key multimedia applications have been presented and their performance on a simultaneous multi-threading (SMT) architecture studied. A metric to evaluate the effective speedup due to SMT has been defined, and an example shows that simply comparing workload performance on SMT vs. SMP systems rather than using the metric can give misleading impressions of the relative performance on the SMT architecture. Our results show that the effective speedup achieved on the SMT architecture we studied gives very consistent results across several workloads. The differences between SMT and SMP architectures, and in particular the impact of sharing the cache in SMT architectures, have been discussed. On SMT, sharing the cache provides cache locality between threads. This interesting characteristic has been exploited to reduce the impact of cache misses by scheduling threads to prefetch data for each other. Using this technique, some workloads show speed-ups on SMT systems competitive with those seen on SMP systems, despite the small hardware cost associated with the SMT architecture.

7. ACKNOWLEDGEMENTS

We would like to thank Doug Carmean for his valuable comments to this work. We would like to thank Sergey Zheltov, Alexander Knyazev, Stanislav Bratanov, Roman Belenov, and Valery Kuriakin for their exceptional efforts in developing the multi-threaded media workloads used in this study. Additionally, we thank Mike Upton, Per Hammarlund, Russell Arnold, Shihjong Kuo, and George K. Chen for valuable discussions during this work.

8. REFERENCES

[1] C. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition", *Data Mining and Knowledge Discovery 2*, p. 121-167, 1998.

[2] Y.-K. Chen, M. Holliman, E. Debes, S. Zheltov, A. Knyazev, S. Bratanov, R. Belenov, and I. Santos, "Media Applications on Hyper-Threading Technology," *Intel Technology Journal,* Q1 2002.

[3] Y. Freund and R. E. Schapire, "Experiments with a New Boosting Algorithm," in *Proc. of Int'l Conf. on Machine Learning*, pp. 148-156, 1996.

[4] B. G. Haskell, A. Puri, and A. N. Netravali, *Digital Video: An Introduction to MPEG-2*, MA: Kluwer, 1997.

[5] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal,* Q1 2001.

[6] Intel Corp., *Intel Pentium 4 Processor Optimization Reference Manual*, Order Number: 248966 (also available on-line: http://developer.intel.com/design/pentium4/manuals/24896604.pdf)

[7] D. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading Technology Microarchitecture and Performance," *Intel Technology Journal,* Q1 2002.

[8] E. Palmer, "Hyper-Threading Characterization," private communications, Apr. 2002.

[9] H. Wang, P. Wang, R. D. Weldon, S. Ettinger, H. Saito, M. Girkar, S. Liao, and J. Shen, "Speculative Precomputation: Exploring the Use of Multithreading Technology for Latency," *Intel Technology Journal,* Q1 2002.

[10] www.videoanalysis.org

* Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit http://www.intel.com/procs/perf/limits.htm or call (U.S.) 1-800-628-8686 or 1-916-356-3104.