Real-Time Parallel MPEG-2 Decoding in Software

Angelos Bilas, Jason Fritts, Jaswinder Pal Singh
Princeton University, Princeton NJ 08544
{bilas@cs, jefritts@ee, jps@cs}.princeton.edu

Abstract

The growing demand for high quality compressed video has led to an increasing need for real-time MPEG decoding at greater resolutions and picture sizes. With the widespread availability of small-scale multiprocessors, a parallel software implementation may provide an effective solution to the decoding problem.

We present a parallel decoder for the MPEG standard, implemented on a shared memory multiprocessor. Goal of this work is to provide an all-software solution for real-time, high-quality video decoding and to investigate the important properties of this application as they pertain to multiprocessor systems.

Both coarse and fine grained implementations are considered for parallelizing the decoder. The coarse-grained approach exploits parallelism at the group of pictures level, while the fine-grained approach parallelizes within pictures, at the slice level. A comparative evaluation of these methods is made, with results presented in terms of speedup, memory requirements, load balance, synchronization time, and temporal and spatial locality. Both methods demonstrate very good speedups and locality properties.

Keywords: Image processing, MPEG, parallel computing, video compression, real-time, shared memory.

1 Introduction

Recent advances in network and microprocessor technology have placed video applications within our reach. High Definition Television (HDTV), Broadcast Satellite Service, Electronic Cinema, Interactive Storage Media, Multimedia Mailing, Networked Database Services, corporate Internet training and conferencing, Remote Video Surveillance and others are now becoming "practical" applications. The huge amount of data needed to make video available in all these cases has led to the adoption of the MPEG-1 and MPEG-2 standards for motion video compression and decompression. These standards greatly reduce the bandwidth and storage space required. Consequently, MPEG-1 and MPEG-2 are already being used in many video applications.

In this paper, we examine how effectively increasingly popular cache-coherent bus-based shared memory multiprocessors can be used to speed up software MPEG decoding. We present two parallel implementations of the MPEG-2 decoder provided by the MPEG Software Simulations Group[1] [9]. The first version exploits very coarse-grained parallelism across groups of pictures in the video sequence, while the second exploits fine-grained parallelism within each picture. We evaluate their performance and resource requirements for different picture sizes and numbers of processors on a 16-processor Silicon Graphics Challenge multiprocessor, that, though a fairly expensive multiprocessor, uses the same parallel algorithms and techniques as would be used by less expensive desktop servers. Detailed measurements with performance monitoring tools are used to understand the role of potential bottlenecks to good performance. Finally, we use multiprocessor simulation to characterize the spatial and temporal data locality properties of the parallel versions and to understand how they will interact with alternative memory system architectures. Both methods demonstrate very good speedups and locality properties.

Due to space limitations we omit most of the background information on MPEG. An expanded version of this paper can be found in [4]. Section 3 describes the test-bed that was used for implementing and benchmarking the different algorithms. In Section 4 we present the methodology for parallelizing the decoder and the parallel implementations with their results. In the same section we also study the locality properties through simulation. Section 5 discusses related work, and conclusions are drawn in Section 6.

2 MPEG Overview

The MPEG coding standard defines a lossy[2] compression technique which takes advantage of spatial and temporal correlation to achieve high compression ratios. In exploiting spatial correlation, compression is achieved by finding the similarities within each picture and using those similarities to eliminate redundancy. Spatial correlation alone, however, provides only moderate compression, so temporal correlation must also be exploited. Successive pictures within the video sequence are examined for similarities, and these are used to remove temporal redundancies. By using both spatial and temporal correla-

[1] Other sequential software MPEG encoders-decoders (codecs) are publicly available as well [6, 10, 5, 9].

[2] A lossy compression scheme is one in which data are lost. Only partial recovery of the original unencoded data is possible.

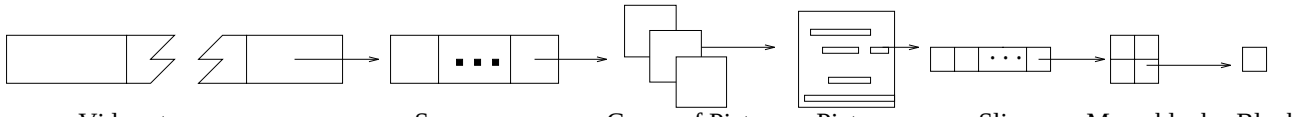

Figure 1. High level bit-stream organization in MPEG

tion, the MPEG standard provides high degrees of compression on video sequences.

An important aspect of the versatility of MPEG is its layered structure [3, 1]. The hierarchy of layers in an MPEG bit-stream is arranged in the following order: *Sequence, Group of Pictures (GOP), Picture, Slice, Macro-block, and Block*. (see Figure 1). The different parts of the stream (except macro-blocks and blocks) are marked with unique, byte aligned codes called start-codes. These start-codes are used both to identify certain parts of the stream and to allow random access into the video stream. The random access ability is vital to parallelization.

The highest level in the layering is the *sequence* level. A sequence is made up of *groups of pictures (GOPs)*. Each GOP is a grouping of a number of adjacent pictures. The purpose in creating such an identifiable grouping is to provide a point of random access into the video stream for play control functions (fast forward, reverse, etc.). Within each GOP are a number of *pictures*. Pictures are further subdivided into *slices*, each of which defines a fragment of a row in the picture. Slices comprise a series of *macro-blocks*, which are 16x16 pixel groups containing the luminance and chrominance data for those pixels in the decoded picture. Macro-blocks are divided into *blocks* (6 to 12 depending upon format). A block is an 8x8 pixel group that describes the luminance or chrominance for that group of pixels. Blocks are the basic unit of data at which the decoder processes the encoded video stream. Macro-blocks and blocks do not have start-codes associated with them; their boundaries are discovered implicitly while decoding.

Pictures in MPEG are encoded into one of three types [3]. All picture types use spatial correlation, but not all use temporal correlation. The first picture type, the *intra* coded picture (I-Picture), uses only spatial correlation. Since their decoding is independent of other pictures, I-Pictures provide access points into the coded stream where decoding can begin. However, using just spatial correlation, they achieve only moderate compression. The second type of picture, the *predictive* coded picture (P-Picture), is coded more efficiently by also using temporal redundancies from a past I or P-Picture. These P-Pictures are then used for reference in further prediction. The final picture type, the *bidirectionally-predictive* coded picture (B-Picture), uses temporal redundancies from both past and future reference pictures, and consequently achieves the highest degree of compression. B-Pictures are never used as references for prediction.

[3]MPEG-1 actually supports a fourth picture type, the DC-coded picture (D-Picture) type. However, this type is little used and was eliminated from MPEG-2

3 System Environment

This section describes the hardware and software environment as well as the testing methodology we used to benchmark the different algorithms.

The Multiprocessor platform: The SGI Challenge multiprocessor is a cache-coherent, bus-based, centralized shared memory multiprocessor. The machine we use has 16 processors connected by a 256 bit-wide bus with peak bandwidth of 1.2 GBytes/sec. Each processor is a 150MHz MIPS R4400 with peak performance of 75 MFlops. Each node has first level data and instruction caches of 16 KBytes each (direct-mapped) and a unified second level cache of 1 MByte (2-way set-associative). The system has 1 GByte of main memory that is 8-way interleaved, out of which we could use up to 500 MBytes for our program. The system can support up to 4 I/O buses, each 320 MBytes/sec peak. The operating system is IRIX 5.3.

Since the machine supports a shared address space programming abstraction, shared data can simply be allocated as such and then referenced directly by any processor. Our parallel programs are written in C, augmented with the `parmacs` parallel programming macros from Argonne National Laboratory. Porting the program to other shared address space architectures is easily achieved by using the proper version of the `parmacs` system for the architecture under consideration.

Stream	Resolution	GOP size	Picture size
1-4	176x120	4,13,16,31	22K
5-8	352x240	4,13,16,31	82.5K
9-12	704x480	4,13,16,31	330K
13-16	1408x960	4,13,16,31	1320K

Table 1. Description of test streams.

Test streams: We tried to be as consistent as possible in choosing the input sequences. Most public domain sequences are small, not consistent, and do not explore the parameter space in any systematic way. We therefore created our own set of test streams. Starting with a small public domain stream, a moving view of a flower garden with 150 pictures and resolution of 352x240 pixels (flowg.mpg, from Stanford), we created larger streams by repeating a number of pictures in a continuous video sequence and scaling each picture using interpolation. Each resulting stream is composed of a total of 1120 pictures, has a 30 pictures/sec display rate and 5 or 7 Mbits/sec bit rate.

The I–P picture distance is 3, thus there are 2 B-Pictures between any two consecutive reference I or P-Pictures. Table 1 shows the characteristics of the streams[4].

We vary only two parameters in our test streams: the resolution and the number of pictures per GOP. These are important because they define the amount of processing required to decode a picture as well as the memory requirements of the system. As seen in Table 1, we use four different resolutions (176x120, 352x240, 704x480, 1408x960)[5] and four different numbers of pictures per GOP (4, 13, 16, 31) for a total of 16 streams.

The public domain MPEG-2 encoder [9] we used to create the streams creates one slice for each row of a picture. Similarly, most public domain video sequences we found also have a small number of slices per picture (usually one per row).

One other parameter of video streams that is of great importance is the bit rate. The bit rate of a video stream provides a measure of both the degree of compression and the relative quality of the video. The streams used in this paper assume a fixed bit rate of 5 Mbits/s for the 352x240 and 704x480 picture sizes and 7 Mbits/s for the 1408x960 picture size. Since bit rates can vary considerably according to the desired degree of compression or video quality, we also examined the effect of different bit rates on parallelism. Using streams of widely varying bit rates, we found that the decoding times for streams of a given picture size are typically within 10%-15% of the time measured for our test streams. This decoding time differential is seen to a proportionate degree with an increasing number of processors, so the speedups we observe are consistent across bit rates.

4 Exploiting Parallelism

The amount of work associated with decoding different pictures, and even with different parts of the same picture, is variable and unpredictable. Maintaining a balanced workload requires that we use some form of dynamic tasking mechanism. Static assignment of tasks to processes is also difficult because tasks are not known ahead of time but are created as the input is read, in parallel with the actual computation. We present two different methods for exploiting parallelism. In both methods the incoming stream is decomposed into tasks that are put in task queues and can be processed in parallel. The difference is in the nature and granularity of the tasks, which affects the performance and characteristics of a parallel implementation.

The possible choices for a task in MPEG are: sequence, group of pictures (GOP), picture, slice, macro-block and block. Given the encoding scheme in MPEG, only a GOP and a slice are reasonable choices, as we shall see.

The first type of parallelism is across pictures. Since P and B-Pictures depend on other nearby pictures, assigning adjacent pictures to different processors leads to many serializing dependencies, and associated synchronization and communication among processors. Parallelizing across either sequences or GOPs might work; however parallelizing across sequences may lead to tasks which are too large and create load imbalance[6]. Therefore, parallelizing across GOPs is a more reasonable choice. Tasks are coarse-grained, but, since GOPs are relatively independent there is essentially no inherent communication in the parallel algorithm except in accessing shared task queues. This forms the first approach, which we call the GOP level implementation.

In the second type of parallelism, parallelism within a picture, the only plausible approach is to use slices as tasks since they are marked with start-codes in the input stream. Macroblocks and blocks would lead to smaller tasks but they do not have start-codes to identify them without actually decoding the input stream. Our other parallel implementation, called the slice level implementation, defines the task unit to be a slice. We shall discuss both these parallel versions and their tradeoffs further in subsequent sections.

4.1 Parallelism at group of pictures level

Since consecutive GOPs[7] may be decoded by different processors, GOPs need to be closed. Although the assumption that all GOPs in the stream are closed is not necessarily true of the streams generated by encoders, it is in fact not very restrictive. One way to overcome it even when the GOPs in the input stream are not closed is by taking advantage of the fact that the stream contains start-codes for pictures and identifies their type using a type field. This parallel design does not require that tasks be GOPs as defined in the input stream, but rather any closed set of pictures that can be decoded independently. The scan process could scan the stream and construct closed tasks.

Figure 2 shows the architecture of the parallel decoder. We dedicate one process, the scan process, to reading the stream from the disk (or network or other source) and identifying the tasks. While reading the stream into memory, it scans the stream for start-codes that mark the beginning of each task. All but one of the other processes are worker processes which dequeue tasks from the task queue and decode the corresponding GOP. The last process is assigned as the display process. It is responsible for displaying the pictures in the correct order, which may have been processed and inserted in the display queue out of order. It is also responsible for dithering the pictures. However, we do not include dithering time in our measurements since it is not a necessary part of decoding. The dithering cost can vary greatly depending on the characteristics of the display device.

The speed at which the scan process is placing pictures in the task queue is shown in Table 2. Here we assume (quite reasonably) that the scan process can be fed with data at the

[4] In MPEG-2 terminology, all the streams have a *main* profile and a *high* level.

[5] The last two streams are more commonly found with pictures sizes of 720x480 and 1440x960. We used the uncommon sizes to maintain consistent picture size ratios.

[6] Recall that in MPEG-2 the GOP level is optional. When the GOP level is used, sequences are typically large, but when it is not used, sequence sizes are usually smaller. Hence, when the GOP level does not exist, the sequence level may be used for parallelization.

[7] A GOP consists of any number of pictures. By definition, a GOP must contain at least one I-Picture. Also, the first picture (in display order) in a GOP must be an I-Picture or a B-Picture, and the last picture in a GOP must be an I-Picture or a P-Picture. If the first picture is an I-Picture or a B-Picture that does not depend on the pictures of the previous GOP, then the GOP is defined as a closed GOP and it can be decoded independently.

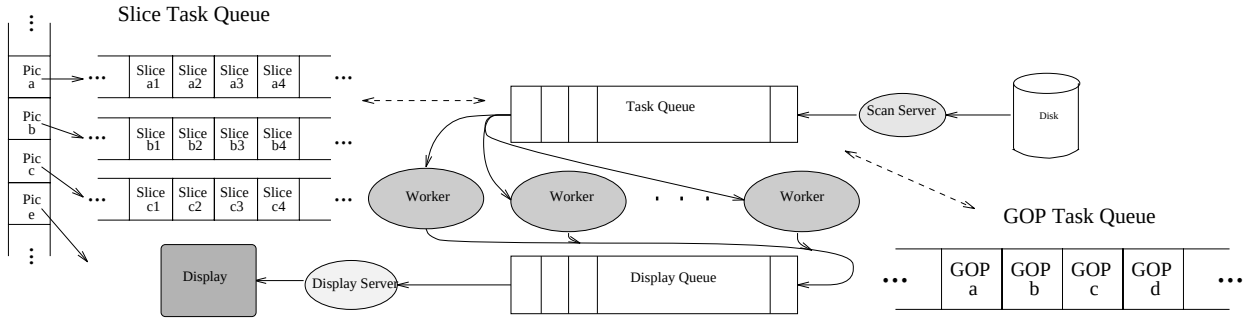

Figure 2. Architecture of the parallel decoder

required bit rate. Doing this under a variety of conditions is a topic of current research in networking and I/O.

Picture size	352x240	704x480	1408x960
File size(MBytes)	25	25	45
Number of pictures	1120	1120	1120
Scan time(sec)	4.5-6.5	4.5-6.5	11-14
Scan rate(pics/sec)	170-250	170-250	80-100
Max pictures/sec	69.9	26.6	7.3

Table 2. Scan rate in the scan process and maximum number of pictures/sec decoded for each picture size.

4.1.1 Results

We tried to capture the behavior of the decoder in terms of speedups, memory requirements, load balance, and the components of execution time including memory overhead. We omit the results obtained for the smallest resolution (176x120) in all cases due to space limitations.

Performance and speedup: We measure speedup as the ratio of the number of pictures per second that P worker processes ($P + 2$ total processes) can decode to the number of pictures per second that are decoded by one worker process (3 total processes). This is different than the speedup obtained over a uniprocessor system, which would multiplex the scan and display processes with the worker process in the uniprocessor baseline and hence likely inflate the speedups. The results show that the speedup is almost linear in all cases. Table 2 gives the maximum number of pictures per second decoded for each picture resolution, using 14 worker processes.

Load imbalance: To capture load imbalance we measured the minimum, maximum and average computing times among the worker processes. The results show that when the number of pictures per GOP is small, the minimum and maximum times are very close to the average. This means that all the worker processes spend approximately the same amount of time computing. As the number of pictures per GOP increases, the load imbalances become more apparent because tasks become larger and fewer. In reality even this is just an artifact of the relatively short input stream, and load imbalance among workers is not likely to be a problem for real streams that contain many GOPs.

Memory subsystem and synchronization overheads: The time spent executing instructions stalled in memory and waiting at synchronization points was measured by `pixie`, `prof` and source level instrumentation. The `pixie` and `prof` timing indicate that in all cases 10%-30% (with an average of 20%) of the time is spent stalled in memory. We shall study cache miss rates and memory system interactions through simulation later.

Synchronization time among the worker processes is minimal. They only need to synchronize when accessing shared resources like the task queue. The time spent on locks was measured to be negligible compared to the processing time in each worker.

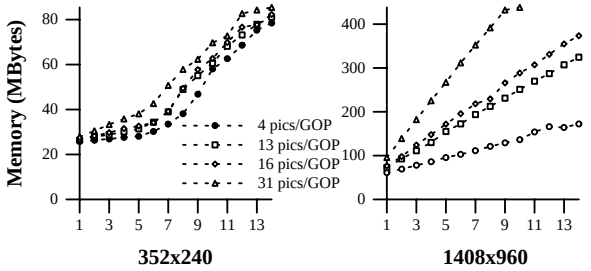

Figure 3. Actual memory requirements for the GOP approach. The x axis is the number of worker processors used.

Memory requirements: The maximum memory required by the system depends on the number of processors and the picture size. Each processor needs to keep up to three decoded pictures in memory (two reference pictures and the one currently being decoded). The speed of the scan server affects the memory requirements as well. It allocates memory to store the data it reads from disk, potentially increasing memory requirements when it out-paces the decoding. In Figure 3 we plot the maximum amount of memory used by the decoder for each test stream. We see that the memory used by the system grows with the size of the GOP, the size of the picture and the number of processors used. In many applications this is not a problem because the size of a GOP is relatively small. For instance, applications that require random access, fast-forward

playback, or fast-reverse playback favor the use of short GOPs.

In addition to the large memory requirements for large picture and GOP sizes this method also has the problem that it has large random access latency for play functions. For example, should the user fast-forward to a later section of the video sequence, decoding must begin anew at that point, with each processor grabbing a different GOP. Because only one processor processes a GOP, the speed at which the video begins to display at that point is dependent upon one processor, not all the processors. As a result, the GOP parallel method is better suited to continuous play.

4.2 Slice level parallelism

While GOP level parallelism is very simple, addressing these problems led us to consider slice level parallelism.

In the most general case, it is not necessary for slices to cover the entire picture. Areas not enclosed in a slice are not encoded. However in all the profiles defined so far by the standard a *restricted slice structure* is used, in which every macro-block in the picture is enclosed in a slice.

The architecture of the decoder is basically the same as in the first approach. However, because of the need of the processors to access picture header information while decoding slices and the need to synchronize at picture boundaries, a 2-D task queue is used (Figure 2). The first level of the task queue holds pictures, while the second level holds the slices within those pictures.

Simple Slice Implementation: In our first implementation (simple slice version), processors synchronize globally at the end of every picture, so parallelism is only exploited within a picture. Also, no attempt is made to preserve locality across the slices from different pictures that are assigned to the same processor.

Two important differences from the GOP level approach are that the memory requirements are much lower and the closed GOP assumption is not necessary. Since all the processors in the system work on the same picture, which is in shared memory, at most three pictures in all need to be in memory at a time (versus at least three pictures per processor as required by the GOP version).

The other benefit of the slice version is that it does not have the random access latency problem for play control functions. When a play control function causes play to begin from a new position in the video stream, all worker processors, not just one, immediately begin decoding the new picture, slice by slice, in parallel.

The disadvantages of the slice approach are synchronization and inherent interprocess communication. Processes communicate as they access the same macro blocks from the reference pictures, particularly if those macro-blocks (slices) were assigned to and written by other processes in the reference picture.

Improved Slice Implementation: Since most test sequences we found used only one slice per row of macro-blocks, each picture usually contains a small number of slices (the vertical resolution divided by 16, the vertical size of a macro-block). This has an important impact on load balance and performance when synchronizing after every picture. For example, a 704x480 picture has 30 slices. If we use 14 workers to decode such a picture, two workers will get three slices while the other twelve only get two and will be idle while the first three are decoding their final slice. Figure 4 shows how this creates a serious problem in speedups. Execution time improves as processors are added only when the load is divided equally between all the processors.

We improve the simple slice implementation by taking advantage of application knowledge, and having the workers synchronize only after certain picture types, not after every picture. The key observation is that all B-pictures in a series use the same reference pictures and are not themselves used as reference pictures. Thus, since the next picture does not depend on the picture currently being decoded, available workers can begin decoding the next picture after completing their tasks in the current picture. Synchronization is needed only at the end of an I or P-picture. This does not exploit the maximum concurrency, but that would require complex synchronization at the slice level.

4.2.1 Results

Results for the slice implementation are presented in the form of speedup, synchronization time (load imbalance) and ideal versus actual time. Compared to the results for the GOP approach, memory requirements are very low and practically independent of the number of processors and the GOP size. We present synchronization wait time results for load imbalance rather than max-min-average results over the whole execution since worker processes synchronize during execution at picture boundaries. Since the effectiveness of the slice approach does not depend on the number of pictures in a GOP, we only vary the size of the pictures and keep the GOP size constant at 13 pictures in this case.

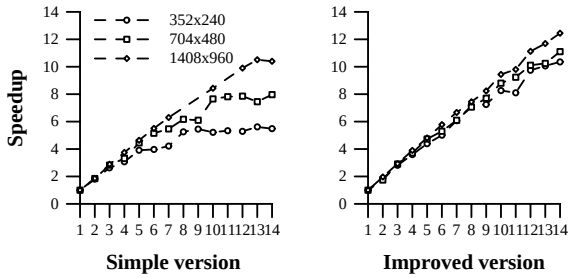

Figure 4. Frames/sec for the slice approach. The x axis shows the number of worker processors.

Performance and speedup: Speedups are measured in the same way as in the GOP approach. Figure 4 shows the performance of both slice implementations. We see that if the processes synchronize at every picture, then speedups are nearly linear only for large pictures (which contain many slices). The knees in the simple version, especially in the 704x480 and 352x240, happen when the integer ratio of the number of slices in a picture over the number of processors, is reduced by one. In the 352x240 case each picture contains 15 slices so performance doesn't increase for more than 8 processors.

The improved version greatly reduces this imbalance. The number of slices processed before global synchronization in-

creases with the I–P distance in the stream. This implementation exposes enough slice level concurrency for the numbers of processors used and achieves very good speedups for all picture resolutions.

Frame size	352x240	704x480	1408x960
Simple version	27.7	15.1	5.6
Improved version	54.7	21.6	6.8
GOP version	69.9	26.6	7.3

Table 3. Maximum number of frames/sec decoded for each picture size.

Table 3 gives the maximum number of frames per second decoded for each picture resolution. From this table we see that the improved slice version approaches the parallel performance of the GOP version without the memory and random access problem. It is slower due to the increased overhead in managing the finer tasks and the additional synchronization time needed at picture boundaries.

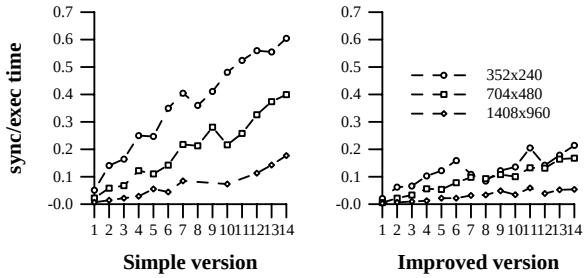

Figure 5. The average (sync time/exec time) of all worker processes versus the number of processors in the slice method. The x axis shows the number of worker processors.

Synchronization overhead: Figure 5 shows the average ratio of synchronization wait time to execution time for all workers as a function of the number of worker processes. It clearly shows that the improved version performs better. The times reported include both the time accessing the task queue and the time spent at synchronization points, though the former is comparatively very small. Thus, although task granularity is much smaller for this version, using a centralized task queue does not constitute a problem for the slice approach either, at these processor counts, which are quite large for decoding.

Memory subsystem: Using `prof` and `pixie` we found the stall time on loads and stores on average less than 5% of the overall execution time, which means that cache misses cost very little in this approach as well.

Let us now examine the memory system interactions of these versions more closely to determine the expected scaling when using larger machines and larger picture sizes, and to see how different cache organizations impact performance.

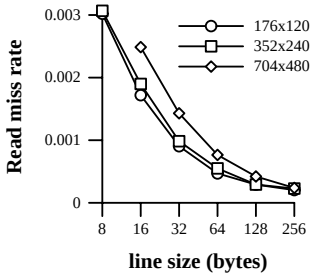

Figure 6. Read miss rate versus line size for an eight-processor execution and 1M , fully associative cache.

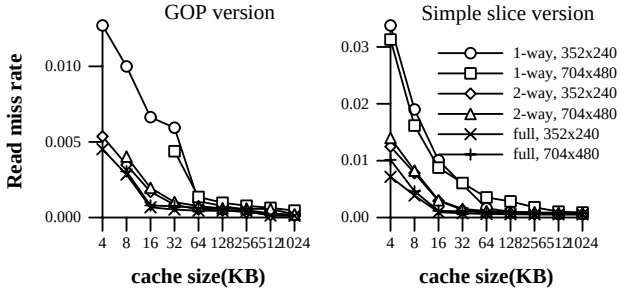

Figure 7. Miss rate versus cache size. Left: GOP version for 1 processor and a 64-byte cache line. Right: Simple slice version for 8 processors and a 64-byte cache line.

4.3 Locality properties

To understand the temporal and spatial data locality, we performed software simulations of the multiprocessor execution for the program. The simulations were done using the Tango-Lite execution-driven reference generator coupled to a memory system simulator. The simulator models a cache-coherent multiprocessor with one level of cache per processor and is flexible in setting architectural parameters.

For spatial locality, Figure 6 shows the read miss rate of the GOP version versus the size of the cache line for a 1 MByte, fully associative, cache. We see that the miss rate halves whenever the cache line size doubles, which indicates that the program has very good spatial locality. The results are for the GOP version, but the slice version has similar behavior.

The size and scaling of a program's working sets (i.e. its temporal locality) are important to understanding how data traffic and performance will scale to larger problems and machines, and for determining what cache sizes will be necessary for good performance. We measure the working sets of the program by plotting the read miss rate versus the cache size (per processor) used in the simulations. Since the GOP approach doesn't have any sharing among the worker processes (which all do similar work) we assume a one processor execution and one-way, two-way and fully associative caches with a 64-byte cache line size. For the slice level approach we present results using eight worker processors. The results for a single

worker processor will be essentially the same as for the GOP approach. The simulation results indicate that the miss rate for realistic second level cache sizes is dominated by cold misses rather than capacity misses even in this case. The number of true sharing misses is small in comparison, and false sharing negligible.

As for capacity misses, we find (Figure 7) that the read miss rate drops dramatically for caches larger than 16K or 32K bytes as long as the caches have some associativity. Direct mapped (one-way associative) caches may need to be larger than 64K bytes to fit the working set. This suggests that the working sets are relatively small, and capacity miss rates and traffic do not constitute a bottleneck for modern caches. The results also show that the working set size does not change with the picture size or the number of processors, even for the slice level version, suggesting that it is determined by the data used for the reconstruction of a single macro-block or set of macro-blocks, which is independent of these parameters.

5 Related Work

Past work on parallel MPEG-2 has focused on message-passing systems, and mostly on the encoding process with its considerably greater computational costs. Reported work has not analyzed the bottlenecks or the important data locality characteristics either.

A parallel MPEG-2 encoder for large scale multiprocessors is presented in [2]. Parallelism is exploited at the block and macro-block level. A parallel decoder that exploits parallelization at the GOP level in a message-passing environment is presented in [7]. This work deals only with MPEG-1 streams. An MPEG-2 video encoder for a LAN of workstations is presented in [11]. They conclude that for their approach the best parallel scheme should be based on slices. Work has also been done in designing hardware or combined hardware-software codecs that achieve real-time performance. A software solution on the Multimedia Video Multiprocessor (TMS320C80) is presented in [8]. They report real-time results for small picture encoding-decoding.

6 Conclusions

We have investigated the behavior of two parallel implementations of MPEG-2 decoding on shared memory multiprocessors. Parallelization was performed at the GOP and slice levels. Both GOP level and slice level approaches give good speedups, though the latter uses much less memory. While the memory requirements of the former increase linearly with GOP size, picture resolution and number of processors, the requirements of the latter depend only on picture resolution. Additionally, the GOP level approach has long random access latency for play control functions. On the other hand, the slice level version has somewhat higher synchronization and communication needs, which reduce its speedup. The results presented were obtained using an SGI Challenge system, but the two implementations are portable across a large number of shared address space platforms. We also used a simulator to investigate the cache behavior of the decoding process, and found excellent spatial and temporal locality. Our results show that communication is small, capacity misses are not a problem and working sets do not grow with picture size.

The good parallel speedups allow us to achieve real time decoding for reasonable sized pictures (352x240, 704x480) on small-scale shared memory multiprocessors. For larger pictures (e.g. 1408x960), close to real-time performance may be achievable with high end systems using the latest processors, and perhaps further optimization of the serial uniprocessor code (we have not tried to optimize the code from the Software Simulation Group other than through the compiler).

7 Acknowledgments

We thank Somnath Ghosh for his help in understanding the structure of the decoder. We are indebted to NEC and particularly to James Philbin for generously providing the Challenge system we used for the measurements, as well as to Kasinath Anupindi and Henry Cejtin for their help in managing the disk space and the cpu time we used. Liviu Iftode helped us in using the Challenge system.

References

[1] I. C. D. 13818-2. *Generic Coding of Moving Pictures and Associated Audio: Recommendation H.262*. ISO/IEC JTC1/SC29 WG11/602, Seoul, November 1993.

[2] S. M. Akramullah, I. Ahmad, and M. L. Liou. A data-parallel approach for real-time MPEG-2 video encoding. *Journal of Parallel and Distributed Computing*, 30(2):129–146, November 1995.

[3] V. Bhaskaran and K. Konstantinides. *Image and Video Compression Standards: Algorithms and Architectures*. Kluwer Academic Publishers, 1995.

[4] A. Bilas, J. Fritts, and J. Singh. Real time parallel MPEG-2 decoding in software. Technical Report TR-516-96, Computer Science Department, Princeton University, Princeton, NJ-08544, 1996.

[5] P. R. Group. *Berkeley MPEG-1 Video Encoder User's Guide*. Computer Science Division, University of California, Berkeley.

[6] A. C. Hung. *PVRG-MPEG CODEC 1.1*. Portable Video Research Group (PRVG), Stanford University, June 1993.

[7] M. K. Kwong, P. T. P. Tang, and B. Lin. *A Real-Time MPEG Software Decoder Using a Portable Message-Passing Library*. Mathematics and Computer Science Division, ANL, Argonne, IL 60439-4844.

[8] W. Lee, R. J. G. J. Golston, and Y. Kim. Real-time MPEG video codec on a single-chip multiprocessor. *Proceedings of the SPIE, Digital Video Compression on Personal Computers: Algorithms and Technologies*, 2187:32–42, February 1994.

[9] MPEG Software Simulation Group. *MPEG-2 Encoder/Decoder, Version 1.1*, 1994.

[10] K. Patel, B. C. Smith, and L. A. Rowe. *Performance of a Software MPEG Video Decoder*. Computer Science Division-EECS, University of California, Berkeley.

[11] Y. Yu and D. Anastassiou. Software implementation of MPEG-2 video encoding using socket programming in LAN. *Proceedings of the SPIE, Conference on Digital Video Compression on Personal Computers: Algorithms and Technologies*, 2187:229–240, February 1994.