Coarse Grain Parallelization of H.264 Video Decoder and Memory Bottleneck in Multi-Core Architectures

Ahmet Gürhanlı, Charlie Chung-Ping Chen, and Shih-Hao Hung

Abstract—**Fine grain methods for parallelization of the H.264 decoder have good latency performance and less memory usage. However, they could not reach the scalability of coarse grain approaches although assuming a well-designed entropy decoder which can feed the increasing number of parallel working cores. We would like to introduce a GOP (Group of Pictures) level approach due to its high scalability, mentioning solution approaches for the well-known memory issues. Our design revokes the need to a scanner for GOP start-codes which was used in the earlier methods. This approach lets all the cores work on the decoding task. Our experiments showed that the memory initialization operations may degrade the scalability of parallel applications substantially. The multi-core cache architecture appeared to be a critical point for getting the desired speedup. We observed a speedup of 7.63 with 8 processors having separate caches, and a speedup of 13.35 using 16 processors when a cache is shared by 2 processors.**

Index Terms—**video compression, H.264 decoder, parallel processing, high-performance computing, image processing.**

I. Introduction

Video encoding and decoding is among the tasks demanding very high computing performance. This demand keeps increasing when we consider the recent developments in video technologies like 3D TV and Ultra High Definition Video. The computation power of a home appliance may become insufficient to decode the huge amount of data needed by those applications.

After Instruction Level Parallelism approached to saturation, Thread Level Parallelism gained more and more importance day by day. Today chip multi-processors (CMPs) are wide-spread, and the number of cores in a CMP is expected to be doubled in every 3 year [1]. This phenomenon increases the importance of data- parallel algorithms for applications needing high computing power.

Recent works on video decoder parallelization preferred fine grain methods, but they could not reach the scalability of coarse grain schemes. In [3], authors declare a linear speedup for GOP approach almost in all cases. GOP approach has no dependency problem, so in an ideal computation environment it has an ideal scalability. Fine grain methods lead to dependencies among data partitions, and the entropy decoding section cannot be processed in parallel in fine-grain approaches. So, even with unlimited memory resources we cannot reach the ideal scalability. For detailed information about the data dependencies in different parallel processing methods [7] is a good reference.

In this paper, we introduce a GOP-level approach for H.264 video decoding. The main point of our design is that; it does not need a GOP start-code scanner. We accept that GOP level parallelism needs more memory resources and has long latency problem. We will address the memory issues in this paper. Latency problem can be resolved by employing hybrid methods as in [4].

Our work showed that memory bandwidth in multi-core architectures is the main bottleneck for highly scalable applications. Programs processing large amount of data has to deal with a huge amount of memory load-store operations. When each core has not a direct path to the memory for write and read operations the bandwidth becomes insufficient for parallel working cores. We will see the importance of efficient memory usage in the results section. In a data-intensive program, choosing a function inefficiently may cause the program run 2 times slower.

Manuscript received September 26, 2010 ; revised January 24, 2011.

Authors are with the Electronics Engineering,Collage of Electrical Engineering and Computer Science, National Taiwan University, Taiwan, R.O.C

II. Structure of the H.264 Decoder and Parallelization Opportunities

The system first entropy decodes the coming stream. There are two choices for entropy coding: CAVLC and CABAC. CABAC is available only in the main profile and uses more computations for achieving a better compression. Entropy decoding is mostly a sequential computation and hard to parallelize.

After entropy decoding the pictures are reordered and passed through inverse quantization. This is followed by inverse integer transform. Then, intra prediction or motion compensation is performed according to the picture type. Finally the picture is passed through a deblocking filter.

There are several alternatives for data level parallelism offered by the structure. Figure 1 shows the data structure in a video sequence. The sequence is first divided into GOPs, each having a certain number of frames. Frames are divided into slices of variable sizes. Slices consist of macro-blocklevel is a candidate for partitioning the data to be decoded.

Ref. [7] gives a good summary of the parallelization methods. The highest level is the Group of Pictures level parallelism. The video stream is divided into groups of pictures for enabling video control functions and synchronization. GOP level parallelism provides high scalability but requires more memory resources. Since there are no dependencies between GOPs, the threads do not need to wait for synchronization operations in this approach.

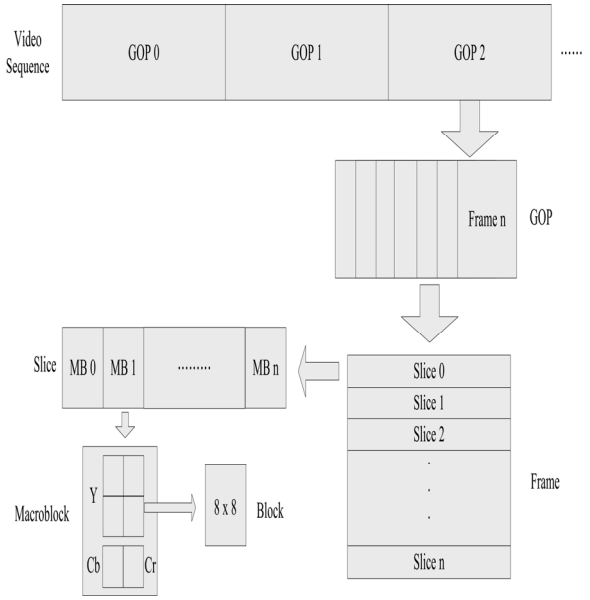

Figure 1: Data structure of H.264 video sequence and partitioning alternatives

In frame-level parallelism, each frame is handled by one thread. The main problem of this approach is the frame referencing. In H.264 all I, P, and B frames might be referenced by others. This flexibility makes frame level parallelism hard to implement.

Another choice is slice level parallelism. In order to ensure error resilience, frames are divided into slices, which are independent from each other. So slices might be processed in parallel. However, scalability of this approach is limited, because the number of slices is decided by the encoder. Besides increasing slice number results in increase in the bit rate.

Recently, most popular approach is the macro-block level parallelism. In this approach macro-blocks are processed in parallel after dependencies have been resolved. Major problem of this approach is its dependence on a high performance entropy decoder.

Finally, the finest approach is block level parallelism. Block level parallelism might be used with SIMD

instructions, for operations like deblocking, IDCT and interpolations which are done at block level.

III. Parallelization Strategy

In a highly scalable model, dependencies between parallel threads must be as low as possible and sequential section of the algorithm must be minimized.

nother important issue is efficient resource usage. Manager-worker architectures allocates one core in the system for task scheduling, and only P-1 of P processors can work on the parallel decoding task. So, in a 4 core CMP we may only expect a speed up of 3.

Most of the parallel decoders we mentioned in related work part, uses a scheduling component. Besides, allocating one core for scheduling issue, this adds a sequential section to the overall decoding task. The scheduler may easily become insufficient for feeding the increasing number of parallel working cores.

Scalability of a fine grain scheme is limited due to the dependencies among the macro-blocks. Before starting to decode a macro-block the system needs to wait for all other referenced macro-blocks be decoded. Another limit for the fine grain methods is the speed of the entropy decoder. A separate core should be employed in entropy decoding, and this sequential decoding must be very fast for feeding lots of parallel macro-block decoders.

Here we present a GOP level scheme that doesn't need a start-code scanner: Scanless GOP. In GOP level parallelism each group of pictures is handled in a processor as shown in figure 2. We employ closed GOPs in our evaluation for the sake of simplicity, but methods for open GOP structures are also introduced in previous works like [1] and [2]. In a closed GOP structure there aren't any references between two GOPs. So each GOP can be decoded in a separate core without any dependency..

However there is another issue limiting the scalability of the system: Before starting to decode a GOP, a process must know the start point of it. In previous approaches a separate process scans the input stream for GOP start-codes, cuts the stream into segments and places these in a task queue.

Actually, we do not need to search for these start codes every time we decode the video. We know their positions in the stream during the encoding process. So these start points can be written in the header of the video stream or in a separate file. For our evaluation, we produced the start points file by means of the decoder, but the same job may easily be done by the encoder as well. Before staring to decode, each process can read these start points into an array, and decode its own portion without waiting for anything.

After this observation we may suggest the following parallel decoding algorithm:

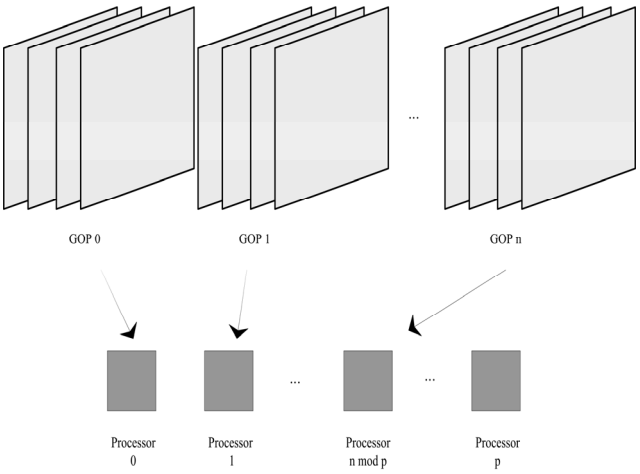

Figure 2: GOP-level parallelization

1. Read the start points into an array.
2. i = process id number
3. While (not end of the stream)
4. Decode (i)th GOP
5. i = i + (total process number)

The forth step of the algorithm includes writing the decoded frames into a disk or a display device or a collector process, but we will exclude this time in our performance calculations.

IV. Experimental Methodology

In our performance evaluations, we used the H.264/AVC

reference software "JM" as the starting point. After parallelizing the program according to our method, we tested our parallel algorithm using a cluster of 5 IBM x3550 machines. Each machine has two 4-core Xeon 5500 series processors. We have tested 3 different situations using 3 different GOP sizes. GOP size affects the performance since it has a direct effect on the memory usage. In order to see the performance on a bounty of computing resources, first we ran just one process in each machine. Then we performed a test on one machine only, to see the performance on a shared-memory platform. And then, we checked the performance using all of the 40 cores. We used Intel's VTune to profile the processor events during the program execution. Profiling results has been very helpful for finding out the bottlenecks of the parallel program. Finally, we tested the run-time and speedup in different machines having different multi-core cache architectures to see the effect of cache structure on scalability and to see the performance in personal computers.

V. Speedup Improvement When the Start-code Scanner is Removed

The speedup gain in coarse grain approaches may be limited in CMP architectures due to memory issues. If we also add some sequential scheduling part to the overall parallel program we might encounter with very disappointing results like in figure 3. Here we see that in a 2 x 4 core machine, the run-time of the algorithm using a start-code scanner hardly decreases from 13.286 sec to 11.057 sec in the minimum case when using 6 cores and than starts to increase again.

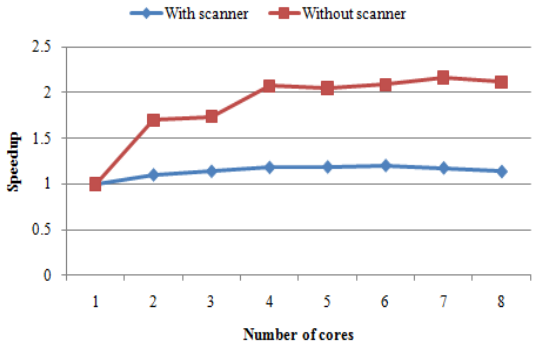

Figure 3: Speedup comparison of the algorithm with start-code scanner vs. the algorithm without a start-code scanner

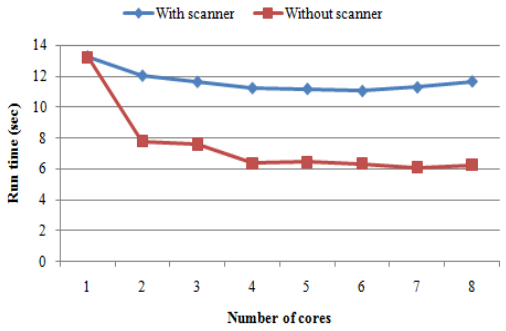

Figure 4: Run-time comparison of the algorithm with start-code scanner vs. the algorithm without a start-code

In figure 3 we see that when we remove the scanner from the algorithm by using ready start codes, we can get the run time decrease from 13.233 sec to 6.105 sec in the minimum case when using 7 cores.

Removing the GOP start-code scanner makes our algorithm purely parallel. Now each process is decoding its own part of data without sharing data with other processes. Besides there is not any scheduler or manager processes assigning tasks to others. So ideally, this algorithm should let each process work without any synchronization.

We can see in figure 3 that maximum speedup jumps from 1.2 to 2.16 when we remove the scanner.

VI. Performance in Multi Machines

We have seen that the speedup in 1 machine is bounded at 2.16. That is not compatible with the expected ideal performance of this algorithm. In order to be sure we can check the results in figure 5. Here we see the speedups for different GOP sizes when we run only 1 process in each machine. This figure is very close to what we expect: we see a very beautiful, linear, one-to-one speedup!

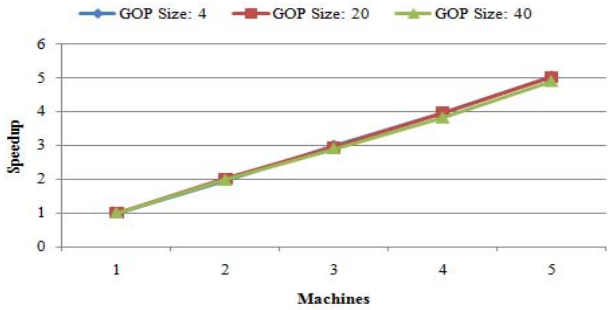

Figure 5: Speedup when only one process is run in each machine

To be sure, we may also check what would happen if we used all the available 40 cores in the 5 machines. Figure 6 shows that we can get a maximum of 11 times speedup when we keep the GOP size very small. GOP size affects the speedup, because it affects the memory usage. A speedup of 11 is reasonable. It approximately equals the speedup in 1 machine, 2.16, times the number of machines, 5.

But, what is the bottleneck in shared-memory platforms?

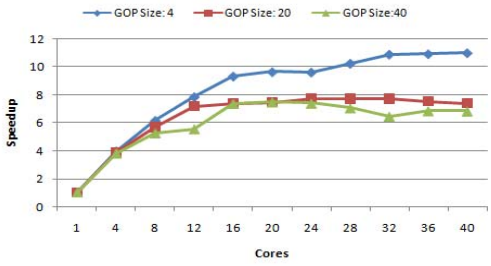

Figure 6: Speedup when we use all the 40 cores in 5 machines

We see the impact of parallelization on branch mispredictions in figure 7. There is a slight increase in mispredicted branches. This must be due to the increase in instruction count. The maximum number of branch mispredictions is just about 420, which is a low number.

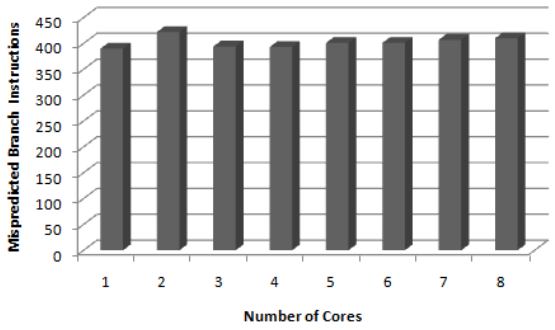

Figure 7: Parallelization impact on branch mispredictions

A reason of performance degradation might be register allocation table stalls. We can see how RAT stall number changes as we increase the number of parallel working processors. There is a general slight increase as we see in figure 8. The increase in RAT stall number may be counted as normal, since we make all the idle cores in the machine start to work.

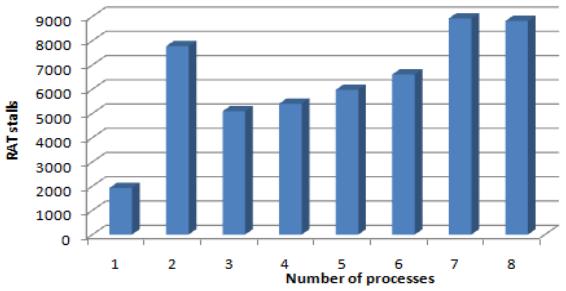

Figure 8: Register allocation table stalls

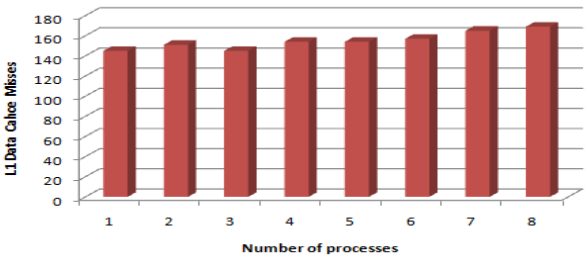

Figure 9: Level 1 data cache misses

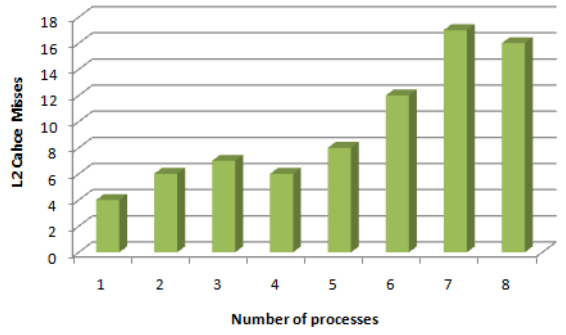

Figure 10: L2 cache misses

So, what causes the speedup degradation? Figure 11 takes us close to the answer. 189,072 resource stalls occur when 1 core is working; this number increases as we increase the parallel working core number and reaches 736,559 when we employ 8 cores. This may explain the matter. Now, we know that there is a resource shortage, but which resource is not sufficient

We see a surprising result in figure 9 and figure 10. Major result of the cache pollution is that it increases the cache misses. The data which is written into the cache newly causes replacement of the data that is already in the cache. So when the program needs the replaced data it has to reload it from the memory system. The amount of data processed by a video decoder is very large. It is expected that when each core processes 1 GOP its cache be filled quickly. So we expect a big number of cache misses. However, we see a pretty good L1 cache performance and an almost ideal L2 cache Performance: 17 misses at most.

VII. The Bottleneck in Shared Memory Platforms

We have seen that the speedup came to saturation in a single machine having 8 cores in 2 processors, even though it produces a very good plot in multiple machines. As we mentioned in our conference paper [8] cache pollution is thought to be the main factor affecting the performance in chip multiprocessors. To find out the major problem we profiled the events during program execution by Intel'vTune.

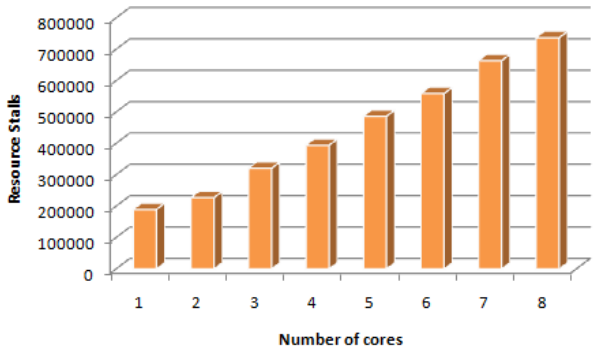

Figure 11: Effect of parallel processing on resource stalls

Figure 12 gives the answer. We see that almost all of the resource stalls are because of load-store operations. We cannot even see the plot for other stalls since their number is too small when compared with the resource stalls due to the load-store operations. When 1 core is working there are 177,117 stalls due to load-store operations. The stall number increases to 732,792 when all the 8 cores are working in parallel. We have seen that cache performance is quite good. So we should suspect the performance of store operations.

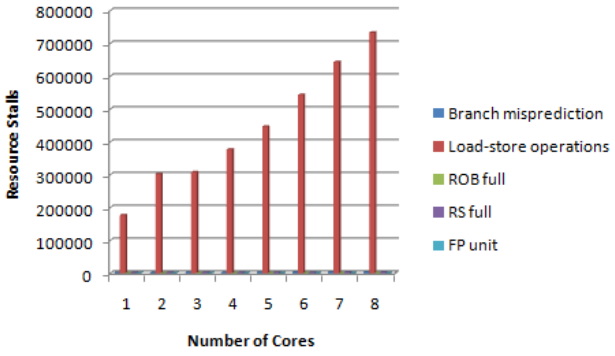

Figure 12: Effect of parallel processing on different kinds of resource stalls

VIII. Most Active Functions

It is the time now to check the most active functions in the system. Figure 13 is very interesting. Function "memset" takes the 70.47% of the whole system for serial execution. This means that most of the time is spent for setting memory locations to a specific value. Basically, we expect store operations to be faster than the load operations. Because during a store the processor can continue to execute other instructions after sending it to the memory system, it does not need to wait for the store operation to finish. But store operations are also done via cache levels. They affect the cache bandwidth and take longer than calculation instructions. When too many stores are executed one after another, the store buffer of the processor fills up, and this causes the load-store resource stalls.

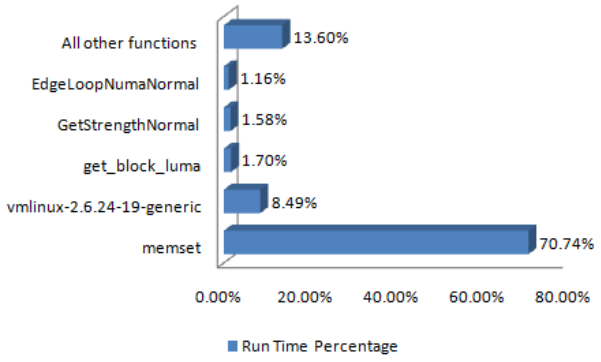

Figure 13: Most active functions in the system during serial run

IX. Impact of Memory Initializations

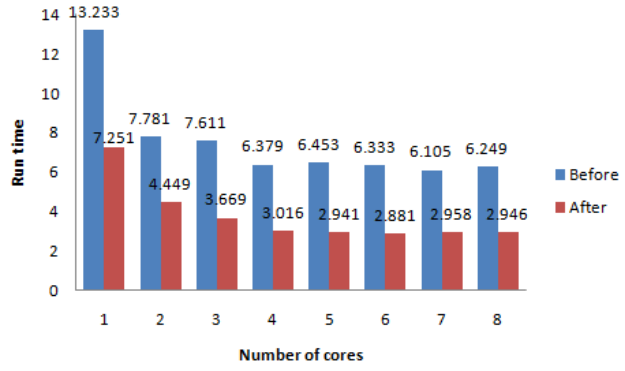

Figure 14: Run time improvement after removing unnecessary memsets

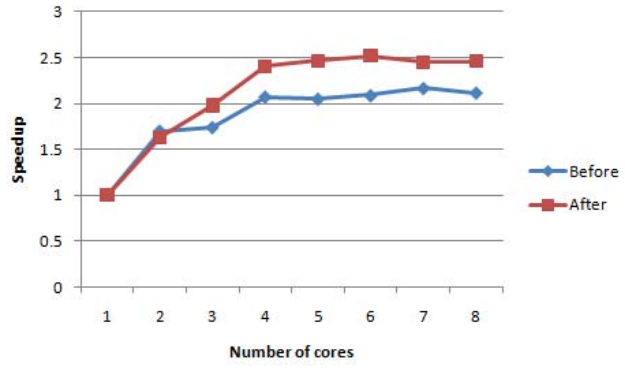

Figure 15: Speedup improvement after removing unnecessary memsets

We noticed that most of the calls to *memset* function is done by the *calloc* function which allocates a location in the memory and initializes these locations to zero. But are all these initializations needed? The answer is system dependent. For portability reasons, in the reference software, all the memory allocations are done with an initialization. Not affecting the correct program execution for linux, we have changed most of the *calloc* functions to *malloc*, which allocates memory, but does not initialize the allocated addresses. We can see the great difference in run time and speedup in figures 14 and 15.

We see that the runtime changed from 6.105 to 2.881, that is we got a speedup of 2.119 just by changing the function *calloc* to *malloc* in suitable places. This shows the impact of memory operations on the performance. The scalability of our algorithm also improved. The speedup due to parallel processing was 2.167 at most before, now we can see a speedup of 2.516.

Figure 16 shows the most active functions in the system after the modification . We see that the percentage of the *memset* function decreased to 60.03%.

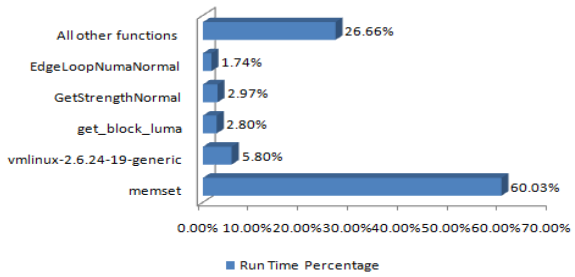

Figure 16: Most active functions in the system during serial run after removing unnecessary memset functions

We have seen that resource stalls due to the store operations is the main bottleneck of this application. Most of the execution time is used by *memset* function which sets a memory area to a specific value. This is done when resetting a data-structure before starting to use it for some different calculation. But this resetting gets very expensive when it is done again and again. It consumes 60% of the runtime even after we removed the unnecessary ones for linux operating system.

The processors repeat the store operation of the same value for the number of items to be initialized. When we have some big amount of data to be initialized this results in a long latency. A possible way to make these memory initializations faster might be using the non-temporal stores which are offered by SSE instruction set. These instructions store the data directly to memory without cache allocation. So in the first glance, we may think that they might have a better scalability, because the cores will not wait for each other for writing to the same cache. However, they will bypass the cache system, only if the related address is not cached. For the case of memory initializations, almost all the data we deal with is already in the cache, because we are resetting some data that we have already used. So we should not be so hopeful about non-temporal store instructions for our problem. Indeed, when we changed the *memset* function with a function using non-temporal stores, we did not observe any speedup.

X. Performance in Shared-memory, Separated-cache Architectures

Finally, it is time to see some good results. Till now, we used Intel Xeon 5500 Series processors having 4 cores. These processors have a "shared" L2 cache of 8MB. We may suspect that a shared cache may degrade the performance of data-intensive parallel processing. As we increase the parallel working core number the cache size and bandwidth available for each core decreases! Furthermore, during the time consuming data initializations, cores are trying to write to the same cache at the same time, causing many resource stalls.

The results we obtained up to this point reveal that shared cache architectures are not suitable for highly scalable data-intensive parallel applications. Even though we haven't encountered a misprediction problem due to enough cache size, the store operations caused the cores wait for each other to write to the same cache. The performance of shared cache structures might be enhanced by providing a separate read-write port for each core in the environment. In our case, we saw that even though the cores are working on totally different data, they had to wait for using the shared resources. We measured 177,117 stalls due to load-store operations when working with 1 core. The stall number increased to 732,792 when all the 8 cores joined the calculation.

We ported our program to IBM System P5 595, a 64-core SMP. In the system there are 8 MCMs (multichip modules) operating on a shared memory. Each MCM has 8 cores, shared 7.6 MB L2 and 144 MB L3 caches. So up to 8 processes this system will behave like a separate cache architecture. After 8 processes we will see the affect of cache sharing. We see the result in figure 17.

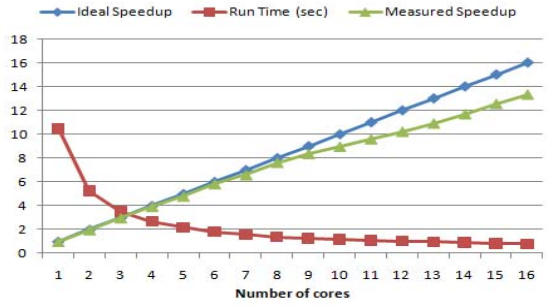

Figure 17: Speedup and run time plot for an SMP with 8 multichip modules having 8 cores in each and one cache system per multichip module

XI. Performance in a Personal Computer

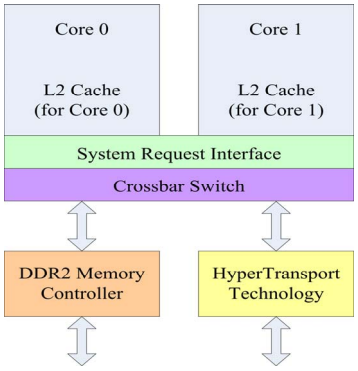

Figure 17: The architecture of AMD Turion 64 X2 mobile technology processor

We saw a pretty good performance in the SMP server. But we should also check the performance in a machine having less computing resources. It has been claimed that GOP level parallelization may be suitable for high performance computing platforms, but it is hard to apply in personal devices.

Therefore, we tested the performance of our algorithm in a different platform. We used a laptop computer having AMD Turion 64 X2 dual-core mobile technology processor. The architecture of this processor is given in figure 18. Here we compare this machine with the IBM x3550 machine. We see that in this processor there are two separate level 2 caches for each core. Each cache is 512KB. We should not be surprised that the machine with AMD dual core processor is slower because it is a notebook having 1.8 GHz core speed and a total of 1 MB L2 cache, while the machine with the Intel processor is a server having 2.5 GHz core speed and a total of 8 MB L2 cache.

Figure 19 shows the impact of cache architecture on the scalability. We see that we can get a speedup of 1.962 with 2 cores in a shared memory platform if we use a separate cache for each core. This is almost an ideal result! This result shows that separated caches are more suitable for multi-core machines in order to get a better scalability. The result also shows the efficiency of GOP level partitioning without a start-code scanner in environments having humble computing resources.

	Server		PC	
	1 core	2 cores	1 core	2 cores
Run Time (s)	13.233	7.781	21.438	10.925
Speedup	1.701		1.962	

Figure 18: Speedup comparison of a separate cache PC with a server, showing the efficiency of the algorithm in personal devices

XII. Related Work

First GOP-level parallel video decoder was introduced a long time ago in [2]. It is a real-time MPEG-1 decoder consisting of parallel processing 16 nodes having distributer and collector components. The task of the distributer is to cut the video sequence into segments. An other GOP level parallelization of MPEG-1 encoder for MIMD multiprocessors was presented by Shen [9].

We see a real-time parallel MPEG-2 decoder in [3]. Both GOP-level and slice-level approaches are evaluated. This system also has scan and display processes. Scan process is responsible for reading the encoded video from the disk and placing encoded GOPs into a task queue. For GOP approach they observed almost linear speedup in all cases. The bottleneck of the design is that, the memory requirement increases with the size of the GOP, size of the picture and number of parallel processors used.

Various slice level parallelization methods have been suggested. Lee introduced a slice level parallel MPEG-2 decoder for HDTV [10]. A manager-worker style parallel H.263 decoder was implemented by Lehtoraanta [11] using 1 manger and 3 worker DSP cores.

A hierarchical parallelization approach for H.264 encoder is introduced in [4]. In this paper authors suggest that a GOP-level scheme and a slice-level scheme might be used

together for overcoming the latency problem. Another hierarchical method was proposed by Chen [14]. In this study frame level and slice level parallelism employed together. When the frame level saturates, slice level parallelization is used for further partitioning. They declare a 4.5X speedup in a machine having 8 cores.

A task level decomposition method has been introduced by Gulati [12]. His system both encodes and decodes H.264 video sequences in real time by means of a control processor and 3 DSPs. Schoffmann [13] also suggests a pipeline model, but at macro-block level.

Among fine grain methods, 2D-wave approach [5], and 3D-wave technique [6] declares pretty high scalabilities. However, fine-grain approaches depend on a well designed CABAC accelerator, since entropy decoding of a single slice or frame is mostly sequential.

XIII. Conclusions

We have introduced a GOP level parallelization method for the H.264 video decoder. Our method revokes the need for a start-code scanner, thus lets all the processors in the environment contribute to the decoding task. This technique also lets the processors work without waiting for a new task assignment. So in the ideal computation environment it has perfect scalability.

We have observed a one-to-one linear speedup in parallel working machines. This is because the memory resources do not change when we increase the parallel working processor number. So we can observe a speedup close to ideal.

The speed of memory store operation degrades the speedup in shared cache platforms. As the parallel working processors increase, the number of simultaneous store operations also increase. This causes a lot of resource stalls due to fullness of the store buffer. We saw a maximum speedup of 2.516 when working with 6 processors, and the speedup got close to saturation after 4 processors.

We saw that memory load-store operations are very expensive in shared cache platforms, and they should be carefully utilized. When we replaced the *calloc* function with *malloc*, both of which are doing memory space allocation but *calloc* is also doing initialization of allocated memory, we got a run-time decrease from 6.105 sec to 2.881 sec, and maximum parallel processing speedup increased from 2.167 to 2.516.

Finally, we saw the effect of multi-core cache architecture on scalability. Our program performed very close to the ideal line up to 8 processes, in an environment having 8 L2-L3 cache systems, after 8 processes we observed a deviation from the ideal line and saw a speedup of 13.35 using 16 cores. Our test on a personal computer showed that the algorithm is also applicable in platforms having not-so-strong computing resources.

References

[1] P. Stenström, Chip-multiprocessing and Beyond, Proc. Twelfth Int.Symp. on High-Performance Computer Architecture, 2006, pp. 109 - 109

[2] 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, 1995

[3] A. Bilas, J. Fritts, and J. Singh, Real-time parallel mpeg-2 decoding insoftware, Parallel Processing Symposium, 1997. Proceedings., 11th International,pp. 19703, 1-5 Apr 1997

[4] A. Rodriguez, A. Gonzalez, and M. P. Malumbres, Hierarchical parallelization of an h.264/avc video encoder, Proc. Int. Symp. on Parallel Computing in Electrical Engineering, 2006, pp. 36368

[5] Mauricio Alvarez Mesa, Alex Ramírez, Arnaldo Azevedo, Cor Meenderinck, Ben Juurlink, Mateo Valero, Scalability of Macroblock-level Parallelism for H.264 Decoding, icpads, pp.236-243, 2009 15th International Conference on Parallel and Distributed Systems, 2009.

[6] A. Azevedo, B.H.H. Juurlink, C.H. Meenderinck, A. Terechko, J. Hoogerbrugge, M. Alvarez, A. Ramirez, M. Valero, A Highly Scalable Parallel Implementation of H.264, Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC), September 2009

[7] Cor Meenderinck , Arnaldo Azevedo , Ben Juurlink , Mauricio Alvarez Mesa , Alex Ramirez, Parallel Scalability of Video Decoders,Journal of Signal Processing Systems, v.57 n.2, p.173-194, November 2009

[8] Ahmet Gurhanli, Charlie Chung-Ping Chen, Shih-Hao Hung, GOP-level Parallelization of the H.264 Video Decoder without a Start-code Scanner, International Conference on Signal Processing Systems, July 2010

[9] Shen, K., Rowe, L. A., Delp, E. J., Parallel implementation of an MPEG-1 encoder: Faster than real time. In Proc. SPIE, digital video compression: Algorithms and technologies 1995 (Vol. 2419, pp. 40718).

[10] Lee, C., Ho, C. S., Tsai, S.-F., Wu, C.-F., Cheng, J.-Y., Wang, L.-W., Implementation of digital hdtv video decoder by multiple multimedia video processors., In International conference on consumer electronics, 1996 (pp. 98, 5 June)

[11] Lehtoranta, O., Hamalainen, T., Saarinen, J., Parallel implementation of h.263 encoder for cif-sized images on quad dsp system. In The 2001 IEEE international symposium on circuits and systems, ISCAS 2001 (Vol. 2, pp. 209 12), 6 May

[12] Gulati, A., Campbell, G. Efficient mapping of the H.264 encoding algorithm onto multiprocessor DSPs. In Proc. embedded processors for multimedia and communications II, 5683(1), 9403, March 2005.

[13] Klaus Schöffmann, O. L., Fauster, M., Böszörmenyi, L., An evaluation of parallelization concepts for baseline-profile compliant H.264/AVC decoders. In Lecture notes in computer science. Euro-Par 2007 parallel processing, August.

[14] Chen, Y., Tian, X., Ge, S., Girkar, M., Towards efficient multi-level threading of h.264 encoder on intel hyper-threading architectures. In Proc. 18th int. parallel and distributed processing symposium, 2004.

Ahmet Gurhanli is a Ph.D. candidate in National Taiwan University, Graduate Institute of Electronics. He graduated from Hacettepe University, Electronics Engineering Department in 2003. He received his MS degree from National Taiwan University in 2006. His research interests are Computer Architecture, Parallel Computing, Electronic Design Automation and Video Processi

Charlie Chung-Ping Chen is a full professor at National Taiwan University, Graduate Institute of Electronics Engineering. He received his Ph.D. from the University of Texas at Austin, USA. His research interests are VLSI CAD, Microprocessor Design, and RF Mix/Signal Circuit Design.

Shih-Hao Hung is an assistant professor at National Taiwan University, Graduate Institute of Networking and Multimedia. His research interests are Co-Optimization of Applications and Architectures, Computer Architecture and Parallel Computing, Computer Performance Characterization and Optimization, Commercial Servers and Applications, and Design of Embedded Syste