Center for Embedded Computer Systems
University of California, Irvine

A Parallel Transaction-Level Model of H.264 Video Decoder

Xu Han, Weiwei Chen and Rainer Doemer

Technical Report CECS-11-03
June 2, 2011

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{hanx, weiweic, doemer}@uci.edu
http://www.cecs.uci.edu/

A Parallel Transaction-Level Model of H.264 Video Decoder

Xu Han, Weiwei Chen and Rainer Doemer

Technical Report CECS-11-03
June 2, 2011

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{hanx, weiweic, doemer}@uci.edu
http://www.cecs.uci.edu

Abstract

H.264 video decoder is a computationally demanding application. In resource-limited embedded environment, it is desirable to exploit parallelism in order to implement a H.264 decoder. After reviewing a list of technical details of H.264 standard, we have discussed several possiblities of parallelization and developed a TLM model with parallel slice decoders. Extensive experiments are performed to demonstrate the benefit of the our model.

Contents

List of Figures

List of Tables

A Parallel Transaction-Level Model of H.264 Video Decoder

Xu Han, Weiwei Chen and Rainer Doemer

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

{hanx, weiweic, doemer}@uci.edu
http://www.cecs.uci.edu

Abstract

H.264 video decoder is a computationally demanding application. In resource-limited embedded environment, it is desirable to exploit parallelism in order to implement a H.264 decoder. After reviewing a list of technical details of H.264 standard, we have discussed several possiblities of parallelization and developed a TLM model with parallel slice decoders. Extensive experiments are performed to demonstrate the benefit of the our model.

1 Introduction

H.264/AVC video coding standard [6] is widely used in video applications such as internet streaming, disc storage, and television services. H.264/AVC provides high-quality video at less than half the bit rate compared to its predecessors H.263 and H.262. In multimedia services, it can be considered the best compromise between quality and size to combine H.264 video and Mp3 video, as in Figure 1. However, H.264 encoding and decoding also requires more computing resources than its predecessors. In order to implement the standard on resource-limited embedded systems, it is highly desirable to exploit available parallelism in its algorithm.

The rest of the report is organized as follows: several features of H.264 standard is introduced in the next section. Section 3 discusses possible parallelism that we can exploit to build our model. Section 4 describes the parallel H.264 decoder model, followed by some experiments and results showing the benefit of the parallelism. Section 6 concludes the report.

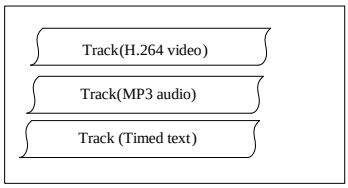

Figure 1: A typical multimedia data unit

2 H.264/AVC Standard Features

This section introduces a list of features of H.264 standard that are most related to our decoder model.

2.1 YUV Color Space and 4:2:0 Sampling

H.264 standard represents color in a picture with YUV format. Component Y (called luma) represents brightness. Components U and V (called chroma) represent color. Because human visual system is more sensitive to brightness than color, H.264 keeps more luma samples than chroma to ensure both picture quality and compression rate. Especially, the most popular sampling structure in which chroma component has one fourth of the number of luma component is called 4:2:0 sampling. Each sample is represented by 8 bits of data.

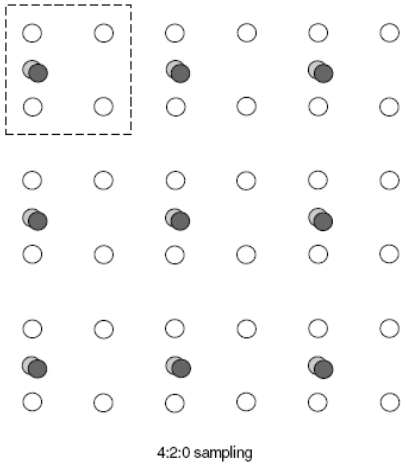

Figure 2: 4:2:0 Sampling

2.2 Macroblocks

A macroblock is the basic encoding and decoding unit in H.264 standard. One macroblock covers a fixed-size rectangular picture area of 16×16 pixels, which contains, with 4:2:0 sampling, 16×16 luma samples and 8×8 of each of the chroma samples, as illustrated in Figure 3. To decode a macroblock, either intra prediction or inter prediction is required.

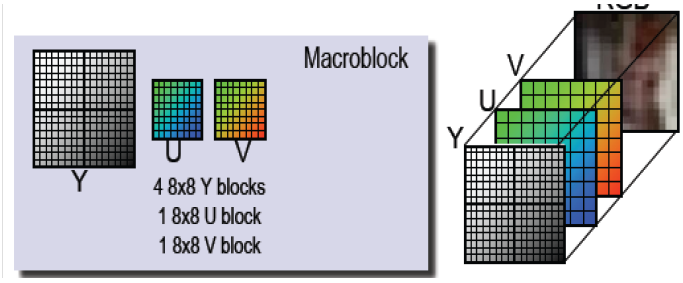

Figure 3: Into a Macroblock

Intra prediction constructs a macroblock by using neighboring samples which are from previously decoded macroblocks in current picture. There are three modes of intra prediction. Intra 4×4 mode predicts a macroblock with smaller blocks of size 4×4. Intra 16×16 mode predicts the whole 16×16 block and is suitable for smooth area of a picture. I_mode simply bypasses the prediction and transform coding of this macroblock to preserve precise video samples. Figure 4 shows how Intra 4×4 prediction can be done by referring to video samples from left and above (which is the previously decoded area) the block to be predicted.

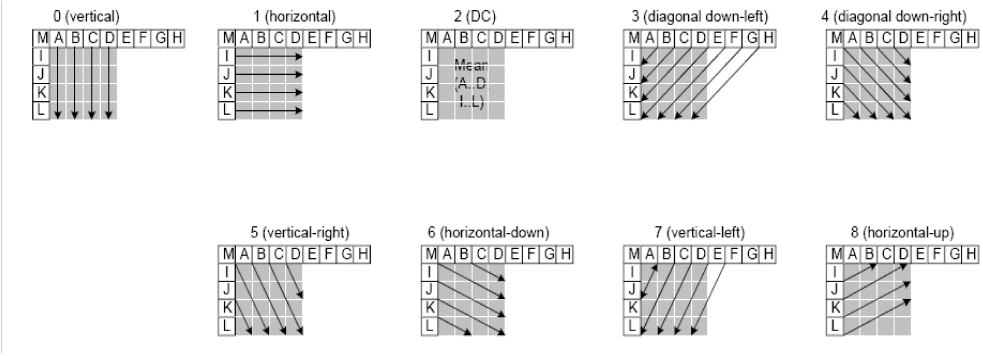

Figure 4: Nine Intra 4×4 prediction modes

Inter prediction, also known as motion compensation, predicts a macroblock by referring to previously decoded pictures. A macroblock can be partitioned into sub-blocks for inter prediction. The option of sub-blocks sizes are 16×16, 16×8, 8×16, and 8×8, among which 8×8 block can be further partitioned into 8×4, 4×8 or 4×4 blocks. Figure 5 illustrates how motion compensation is

done with two references pictures. A picture reference index and a motion vector are required to perform motion compensation.

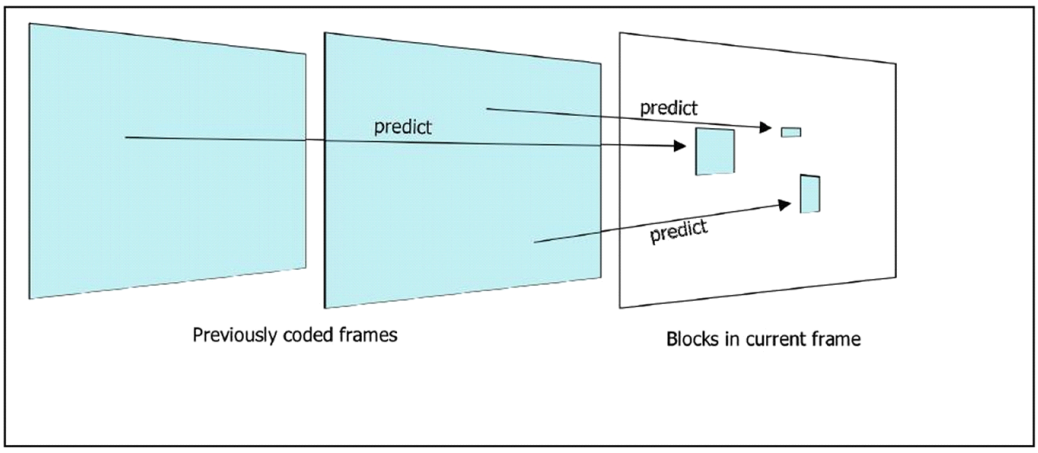

Figure 5: Motion Compensation

2.3 Slices

A slice is a sequence of macroblocks. On the other hand, a H.264 video frame can be split into one or more slices. Figure 6 shows an example of a frame divided into four slices. Note that slices can be of very flexible shape and size with a feature of Flexible Macroblock Ordering(FMO) enabled, as shown in Figure 7, where each color represents a slice group which can contain one or more slices.

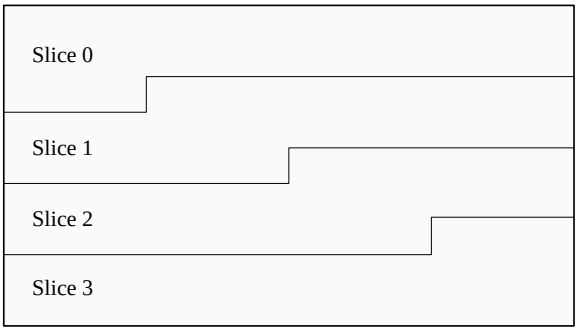

Figure 6: A H.264 video frame divided into four fixed-size slices.

Slices can be classified into three types according to their coding style. A slice with all macroblocks coded using intra prediction is called an *I slice*. In addition to intra prediction, a *P slice* contains inter predicted macroblocks with only one reference frame per prediction block. In addition to coding types in a P slice, a *B slice* can have inter predicted macroblocks with two reference

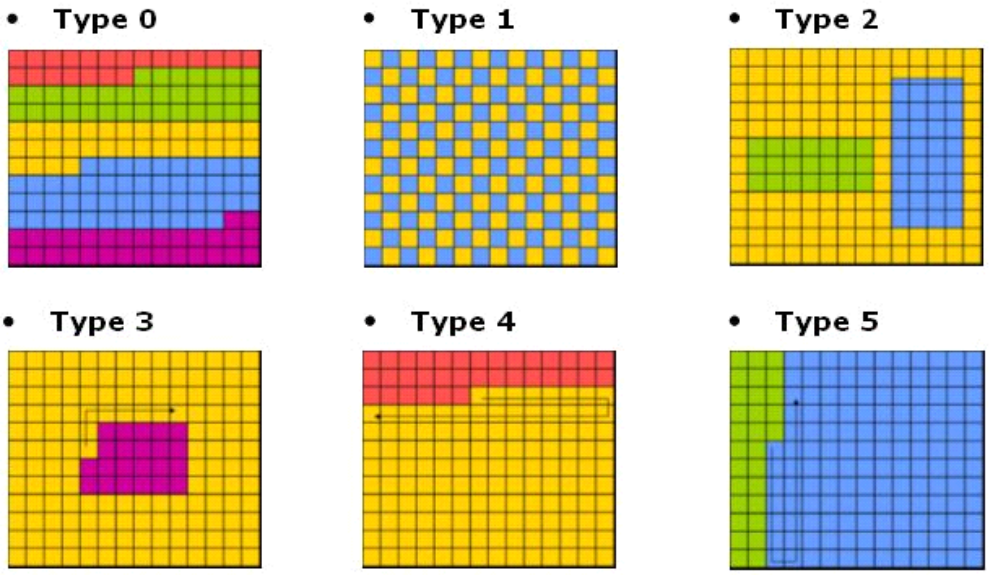

Figure 7: Possible Slice Patterns with FMO enabled

frames per prediction block.

Notably, Slices are *independent* of each other, in the sense that decoding one slice will not require any data from the other slices in the current video frame. However, to correctly decode a frame, reference frames are needed if the frame contains P slices, and information from other slices are still needed when deblocking filter works across slice boundaries.

2.4 H.264 Decoding Algorithm

Our study is based on H.264/AVC JM reference software[4]. This section is to introduce the algorithm of H.264 decoder as implemented in JM version 13.0. JM 13.0 relies heavily on global storage depicted in parallelograms in Figure 8) in the decoding process. Generally, it accesses all input image parameter in structure *img*, temporary output data in structure *dec_picture* and decoded frames in decoded picture buffer *dpb*.

As shown in Figure 8, decoding starts on parsing incoming NAL units, which is a logical data packet containing H.264 syntax structures. It carries both coded video data and information indicating the method to decode the data. After parsing, coded residual data is entropy decoded, inverse quantized, reordered, inverse transformed and result in the decoded residual data stored in array *m7*, which will later build part of a decoded macroblock. On the other hand, parsed information on *MB_types* and *prediction mode info* is used to construct the current macroblock by predicting its content. Depending on how the macroblock is encoded in the first place, either intra prediction or motion compensation will apply to acquired the predicted data, the result of which is stored in an array *mpr* in JM 13.0. At this point, the macroblock has all necessary data decoded and can be

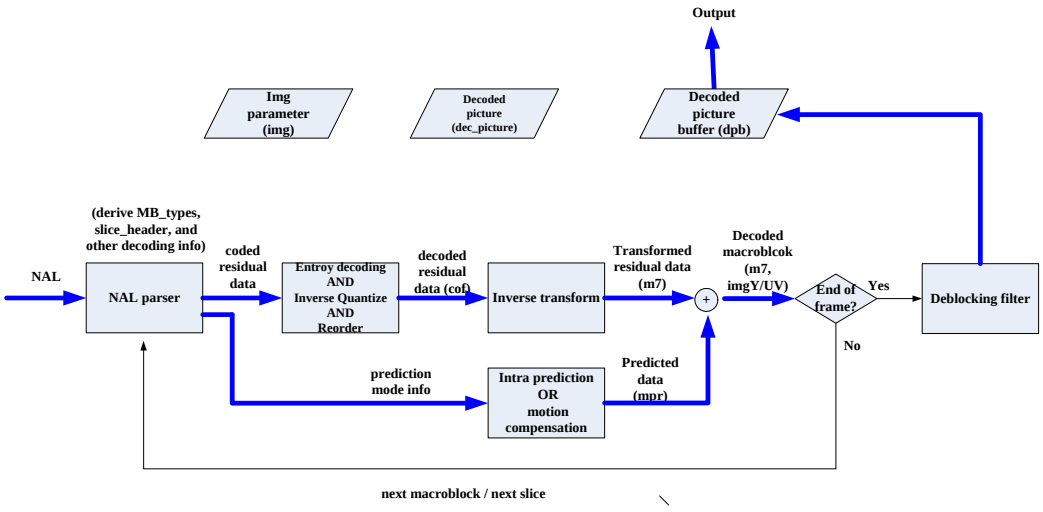

Figure 8: Block Diagram of H.264 Decoder in JM 13.0

constructed by adding transformed residual data and predicted data together. On finish decoding one macroblock, the decoder proceeds to next macroblock in current slice or next slice. After an entire frame is decoded, deblocking filter is applied to remove the blocking artifacts in a frame. The output of the deblocking filter is stored in a decoded picture buffer which uses the buffered frames as reference to decode future frames.

3 Parallelism Exploitation

Various possible parallelisms exist in H.264 standard.

On frame level, decoding a frame can depend on reference frames of very flexible position and number with inter prediction applied. Although part of the frames can be intra predicted only, it is still very difficult to exploit parallelism on frame level.

On macroblock level, we firstly review how a macroblock is constructed in JM 13.0. Illustrated by Figure 9, the decoder firstly acquires the predicted data (*mpr*) by refer

As for the deblocking filter, although it operates across boundaries of macroblocks and slices, deblocking a macroblock only depends on its neighbor macroblocks since its goal is to remove block edges. Therefore, it is possible to filter a frame in parallel, though such parallelism is not easy to exploit.

Due to the nature of slices, it is most promising to exploit parallelism on slice level. We developed a SpecC model with four parallel slice decoders, which is expected to be most efficiency when incoming stream has 4 balanced slices per frame.

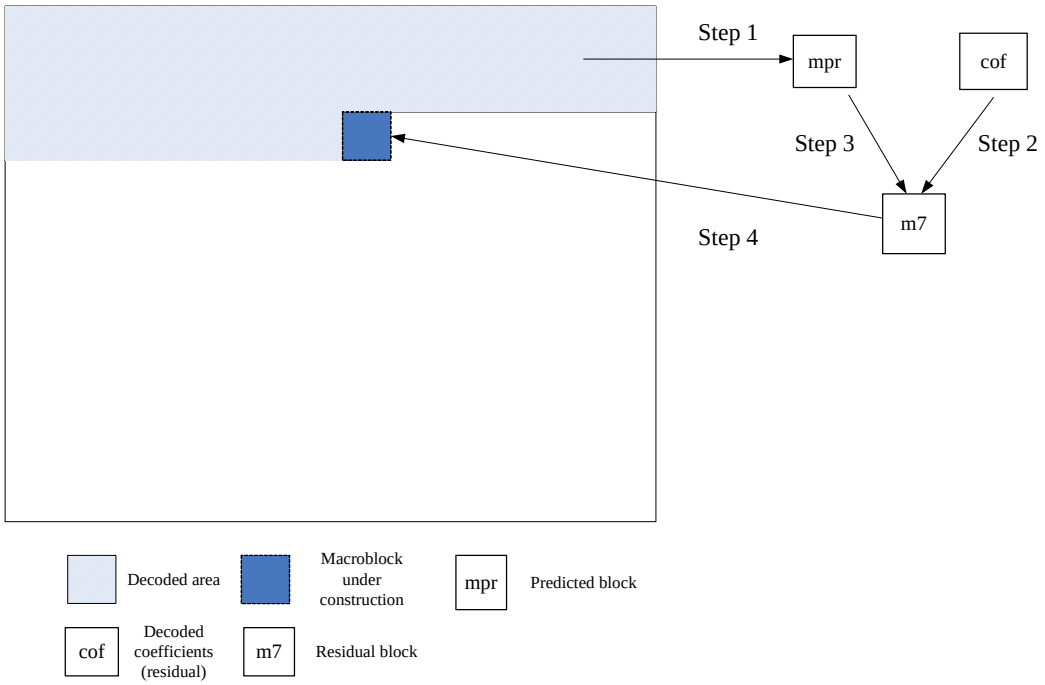

Figure 9: Construction of a macroblock in JM 13.0 H.264 decoder

4 A SpecC Model with Parallel Slice Decoders

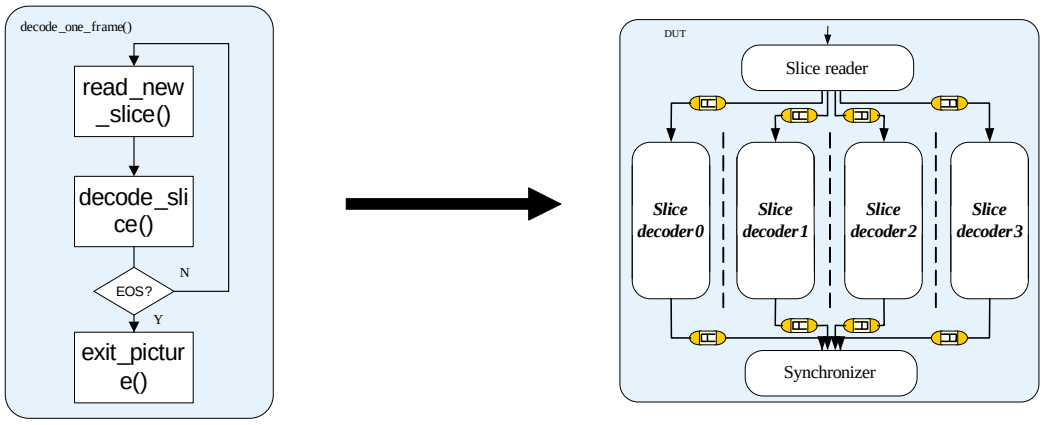

Figure 10: Recoding From JM Reference to SpecC model

An initial system-level model with Simulator, DUT and Monitor structure is designed in [5]. Now after identifying parallelism at slice level, we have extended the existing system-level model

by further recoding JM reference code. The decoding procedure in JM is done by function *decode_one_frame*, which contains subfunctions *read_new_slice*, *decode_slice* and *exit_picture*, as shown in Figure 10. After recoding, *read_new_slice* is recoded into *slice reader*, *decode_slice* into four *slice decoders* and *exit_picture* into *Synchronizer*.

With any input video stream encoded with four slices per frame, Slice Reader reads out four slices every running cycle and dispatch them to Slice Decoders via channel interface. Each slice decoder internally consists of the regular H.264 decoder functions, such as entropy decoding, inverse quantization and transformation, motion compensation, and intra-prediction. Upon each Slice Decoder finishing its work in parallel, Synchronizer completes the decoding by applying a deblocking filter to the decoded frame.

Up to date, this model can be manually adjusted to decode video stream with different number of slices.

5 Experiment and Results

We use the System-on-Chip Environment (SCE) [2] for synthesis and validating of our H.264 decoder model. SCE is a refinement-based framework for heterogeneous MPSoC design. It starts with a system specification model described in the SpecC language [3] and implements a top-down ESL design flow based on the specify-explore-refine methodology.

We partition our parallel H.264 decoder model as follows: the four slice decoders are mapped onto four custom hardware units; the synchronizer is mapped onto an ARM7TDMI processor at 100MHz which also implements the overall control tasks and cooperation with the surrounding testbench. We choose Round-Robin scheduling for tasks in the processor and allocate an AMBA AHB for communication between the processor and the hardware units. Using SCE, we generate the transaction level models (TLM) of our parallel H.264 decoder design at different abstraction levels. They are *spec* for the specification model, *arch* for the architecture mapped model with different kinds of processing elements (PE), *sched* for the model with scheduling decisions made for operation systems on mapped processors, *net* for the model with network connectivities among PEs, *tlm* for transaction level model with communication protocols, and *comm* for pin-cycle accurate model with communication details.

For our first experiment, we use the same stream "Harbour" of 299 video frames, each with 4 slices of equal size. As shown in [1], 68.4% of the total computation time is spent in the slice decoding, which we have parallelized in our decoder model.

As a reference point, we calculate the maximum possible performance gain as follows:

$$MaxSpeedup = \frac{1}{\frac{ParallelPart}{NumOfCores} + SerialPart}$$

For 4 parallel cores, the maximum speedup is

$$MaxSpeedup_4 = \frac{1}{\frac{0.684}{4} + (1 - 0.684)} = 2.05$$

The maximum speedup for 2 cores is accordingly $MaxSpeedup_2 = 1.52$.

Table 1 lists the simulation results for several TLMs generated with SCE when using our multi-core simulator on a Fedora core 12 host PC with a 4-core CPU (Intel(R) Core(TM)2 Quad) at

3.0 GHz, compiled with optimization (-O2) enabled. We compare the elapsed simulation time against the single-core reference simulator (the table also includes the CPU load reported by the Linux OS). Although simulation performances decrease when issuing only one parallel thread due to additional mutexes for safe synchronization in each channel and the scheduler, our multi-core parallel simulation is very effective in reducing the simulation time for all the models when multiple cores in the simulation host are used.

Table 1: Simulation results of H.264 Decoder ("Harbour", 299 frames 4 slices each, 30 fps).

Simulator Par. issued threads:		Reference	Multi-Core						#delta cycles	#threads
		n/a	1		2		4			
		sim. time	sim. time	speedup	sim. time	speedup	sim. time	speedup		
models	spec	20.80s (99%)	21.12s (99%)	0.98	14.57s (146%)	1.43	11.96s (193%)	1.74	76280	15
	arch	21.27s (97%)	21.50s (97%)	0.99	14.90s (142%)	1.43	12.05s (188%)	1.76	76280	15
	sched	21.43s (97%)	21.72s (97%)	0.99	15.26s (141%)	1.40	12.98s (182%)	1.65	82431	16
	net	21.37s (97%)	21.49s (99%)	0.99	15.58s (138%)	1.37	13.04s (181%)	1.64	82713	16
	tlm	21.64s (98%)	22.12s (98%)	0.98	16.06s (137%)	1.35	13.99s (175%)	1.55	115564	63
	comm	26.32s (96%)	26.25s (97%)	1.00	19.50s (133%)	1.35	25.57s (138%)	1.03	205010	75
maximum speedup		1.00	1.00		1.52		2.05		n/a	n/a

Table 1 also lists the measured speedup and the maximum theoretical speedup for the models that we have created following the SCE design flow. The more threads are issued in each scheduling step, the more speedup we gain. The *#delta cycles* column shows the total number of delta cycles executed when simulating each model. This number increases when the design is refined and is the reason why we gain less speedup at lower abstraction levels. More communication overhead is introduced and the increasing need for scheduling reduce the parallelism. However, the measured speedups are somewhat lower than the maximum, which is reasonable given the overhead introduced due to parallelizing and synchronizing the slice decoders. The comparatively lower performance gain for the *comm* model in simulation with 4 threads can be explained due to the inefficient cache utilization in our Intel(R) Core(TM)2 Quad machine[1].

Using a video stream with 4 slices in each frame is ideal for our model with 4 hardware decoders. However, we even achieve simulation speedup for less ideal cases. Table 2 shows the results when the test stream contains different number of slices. We also create a test stream file with 4 slices per frame but the size of the slices are imbalanced (percentage of MBs in each slice is 31%, 31%, 31%, 7%). Here, the speedup of our multi-core simulator versus the reference one is 0.98 for issuing 1 thread, 1.28 for 2 threads, and 1.58 for 4 threads. As expected, the speedup decreases when available parallel work load is imbalanced.

6 Conclusion

In this work, we have discussed options of parallelism in H.264 decoding and developed a SpecC model with four parallel slice decoders. We have refined the model using SCE design flow and per-

[1]The Intel(R) Core(TM)2 Quad implements a two-pairs-of-two-cores architecture and Intel Advanced Smart Cache technology for each core pair (http://www.intel.com/products/processor/core2quad/prod_brief.pdf)

Table 2: Simulation speedup with different h264 streams (spec model).

Simulator		Reference	Multi-Core		
Par. issued threads:		n/a	1	2	4
slices frame	1	1.00	0.98	0.98	0.95
	2	1.00	0.98	1.40	1.35
	3	1.00	0.99	1.26	1.72
	4	1.00	0.98	1.43	1.74
	5	1.00	0.99	1.27	1.53
	6	1.00	0.99	1.41	1.68
	7	1.00	0.98	1.30	1.55
	8	1.00	0.98	1.39	1.59

formed extensive experiments with our multi-core simulator. The results show satisfactory speedup in our parallel decoder model.

Acknowledgment

This work has been supported in part by funding from the National Science Foundation (NSF) under research grant NSF Award #0747523. The authors thank the NSF for the valuable support. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Weiwei Chen, Xu Han, and R. Doemer. Multicore simulation of transaction-level models using the soc environment. *Design Test of Computers, IEEE*, 28(3):20 –31, may-june 2011.

[2] Rainer Dömer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and Daniel Gajski. System-on-Chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design. *EURASIP Journal on Embedded Systems*, 2008(647953):13 pages, 2008.

[3] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, and Daniel D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer, 2001.

[4] H.264/AVC JM Reference Software. http://iphome.hhi.de/suehring/tml/.

[5] Bin Zhang Weiwei Chen, Siwen Sun and R. Dömer. System level modeling of a h.264 decoder. Technical Report CECS-TR-08-10, Center for Embedded Computer Systems, University of California, Irvine, 2008.

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