In Proceedings of the IEEE Workshop on Signal Processing Systems, pages 457-462,
Shanghai, China, October 2007.

LOW-OVERHEAD RUN-TIME SCHEDULING FOR FINE-GRAINED ACCELERATION OF SIGNAL PROCESSING SYSTEMS

Jani Boutellier[1], Shuvra S. Bhattacharyya[2], Olli Silvén[1]

[1]Machine Vision Group, University of Oulu, 90014 University of Oulu, Finland
[2]Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742

ABSTRACT

In this paper, we present four scheduling algorithms that provide flexible utilization of fine-grain DSP accelerators with low run-time overhead. Methods that have originally been used in operations research are implemented in a way that minimizes the amount of run-time computations. These low overhead scheduling methods can be used for synchronization in multi-processor systems, especially when dedicated co-processors implement tasks with low turnaround times. We demonstrate our methods by an application to MPEG-4 video decoding. In this demonstration, MPEG-4 macroblock decoding is modeled as a permutation flowshop problem and our proposed algorithms are applied to schedule co-processors that implement MPEG-4 block decoding operations. Experimental results demonstrate the effectiveness of our scheduling approach.

Index Terms— Scheduling, parallel processing, digital signal processors

1. INTRODUCTION

When a data processing system consists of multiple computing units that run in parallel, their mutual communication needs to be synchronized in some manner to keep the results consistent. The synchronization method can be very simple, like polling. However, in complex systems, more advanced methods are required. The most sophisticated method of synchronization is *scheduling*. In our context, scheduling is the task of computing a timetable for the processing units in the system, so that all units receive and transmit their data in a timely manner. Scheduling involves always some kind of optimization. The objectives of the optimization procedure can vary [7], but most often the purpose is to minimize the *makespan*. Makespan minimization refers to the procedure of finding a schedule that completes all assigned tasks in a minimal time period.

Scheduling has been researched for decades and therefore a vast amount of scheduling methods exist. However, scheduling is often used only for synchronizing the transactions of large-scale units because the computation of a good schedule can take a relatively long time. Small-scale transaction synchronization has been most often handled with interrupts. However, recent studies have shown that the overheads of interrupt-based synchronization can become high and better alternatives should be developed [15]. In some cases, low-overhead run-time scheduling can be a feasible alternative [2].

In this paper we shall look into some methods to construct very fast scheduling methods and discuss the areas where the algorithms can be used. Finally, an application to MPEG-4 video decoding is presented along with measurement results.

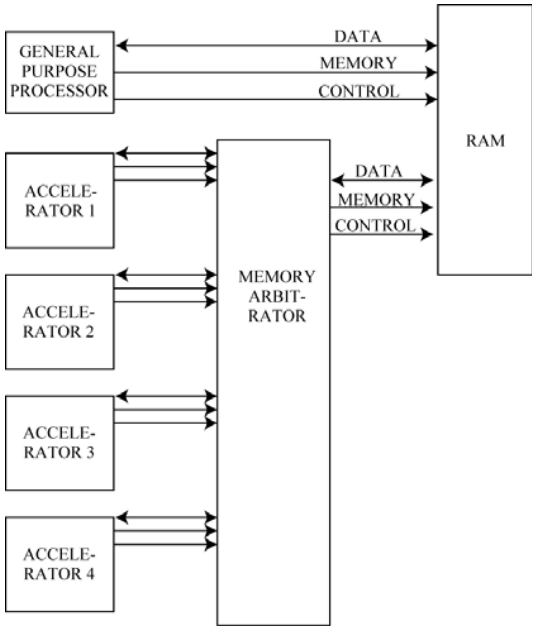

Fig. 1. Architecture of a fine-grain accelerated shared-memory system.

1.1. MPEG-4 video decoding with fine-grained acceleration

Fine-grained hardware acceleration is a variant of conventional hardware acceleration, where the accelerators implement only small-scale tasks with latencies of 500-5000

clock cycles. Figure 1 shows an example of a fine-grain accelerated system architecture. Fine-grain accelerated functions can be *e. g.* the computation of an 8x8 DCT or a bilinear interpolation of the same size. MPEG-4 video decoding is an important application for which fine-grained hardware accelerators can yield significant benefits [8]. This approach enables possibilities to accelerate general-purpose functions that can be shared by different tasks. An evident example would be a multi-standard video decoder chip, since different video codecs have plenty of common functions. Fine-grained hardware acceleration also enables new possibilities for better hardware utilization. For example in MPEG-4 video the contents of a macroblock can vary greatly, and a monolithic macroblock decoder chip can end up running half-idle because some macroblocks are only partially coded.

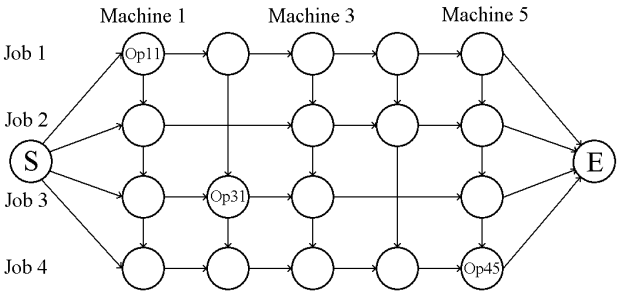

Fig. 2. A permutation flow-shop problem shown as a directed, acyclic graph.

Since the fine-grained accelerators are intended to be invoked very frequently, the synchronization mechanism needs to be efficient, not to cause unreasonable overhead. For this reason, interrupt-based synchronization is out of question, whereas low-overhead scheduling is much more appealing [2, 15]. However, the scheduling has to be done efficiently, which can be shown easily by a brief example: Suppose we have a MPEG-4 -compressed video sequence of resolution 320x240 at 25 frames per second and we schedule the decoding operations of each macroblock. Also, let us suppose that each macroblock decoding involves an average of 30 operations that have to be scheduled. Then, we have to compute 7500 schedules for 30 operations each second in addition to the actual decoding. If we suppose that the schedule computation takes 100000 clock cycles, the scheduling alone consumes 750 MHz computing power. To cope with such frequent scheduler calls, the scheduling algorithm needs to be very efficient.

There has been some research in fine-grained hardware accelerator scheduling in the last few years. Ma *et al.* have developed a scheduling method that tries to minimize both the energy consumption and makespan in schedules [13]. Wang *et al.* have implemented a MPEG-4 video decoder with pipelined fine-grained accelerators [17]. Ling *et al.* have implemented a real-time HDTV decoder with static scheduling of fine-grained accelerators [12]. Also, a similar approach to ours has been used by [5], where schedules are partially computed off-line leaving only minor tasks to be conducted at run-time.

The rest of this paper is organized as follows: the flow-shop scheduling problem is introduced in Section 2, Section 3 shows how MPEG-4 video decoding can be modelled as a flow-shop problem, Section 4 discusses the efficient implementation of appropriate scheduling algorithms and Section 5 shows practical test results.

2. THE FLOW-SHOP SCHEDULING PROBLEM

The aim of this paper is to model the process of MPEG-4 video decoding as a flow-shop problem, since it enables us to use efficient scheduling methods. We shall first look into the definition of the flow-shop problem and agree on some terms.

Flow-shop scheduling was initially formulated by Johnson in 1954 [9] and has since been actively studied. The flow-shop problem involves the processing of *N jobs* on *M machines.* Each job consists of a set of *operations*, so that each operation has to be processed on exactly one machine. The processing times of operations are known beforehand and each job passes through the machines in a prescribed order, although it is possible for jobs to skip machines [9]. Throughout this paper we assume that we are minimizing the makespan instead of other possible scheduling objectives. For some applications, energy minimization would be a more appropriate objective, but it is not studied in this paper due to space limitations.

The *permutation flow-shop problem* is a more restricted version of the general flow-shop problem. In permutation flow-shops the job order for each machine is the same [7], which means that a certain operation *x* must be performed to job *n-1* before it can be performed on job *n* (assuming we have to complete the jobs 1… *n* in ascending order). A schedule is completely specified by a permutation sequence (1, 2, 3, …, n) that describes the job order, if we agree ourselves upon the method of *timetabling* [7]. Timetabling is the process of constructing actual start and end times for each operation and job based on the order of jobs and technological restrictions.

Semi-active timetabling is a method to unambiguously derive schedules from ordered job sequences. In semi-active timetabling each operation is started as soon as possible, so that no technological constraints are violated. Another possibility is to use *no-wait timetabling*. In no-wait timetabling each operation (*n*) within one job is started immediately after the previous operation (*n-1*) has finished. [7] discusses these alternatives thoroughly and should be consulted for more specific information.

Generally computer science –oriented scheduling problems are modelled with *directed, acyclic graphs* (DAGs) [16, 11], that model the interdependencies of different computing operations. Figure 2 shows a DAG

formulation of a permutation flow-shop problem. Each row of vertices represents one job and each column represents a machine. One vertex represents an operation of a job on a particular machine. Notice that in the figure two jobs skip some machines. It can be seen that the execution order of operations is very restricted. After operation (1,1) has executed, only operations (1,2) and (2,1) can be executed. When the method of semi-active timetabling is used, operations (1,2) and (2,1) start at the same time, whereas no-wait timetabling requires that (2,1) is executed immediately after (1,1). (1,2) may only execute when it is sure that (2,2), (3,2) and (4,2) can follow (1,2) instantly without waiting. Since the graph does not allow much freedom in schedule optimization, the remaining variable to be optimized is the order of jobs, as it was stated in the problem formulation of permutation flow-shop scheduling.

In our scheduling algorithms we assume that we know beforehand the job types that can be expected to arrive to the scheduler. This assumption is not a part of the (permutation) flow-shop definition, and can appear very restrictive at first glance, since the amount of different jobs may seem huge in some applications. However, we have to remember that the flow-shop model itself restricts the essence of jobs: *each job may use each machine only once*, which sets the maximum job length equal to the number of machines. Practically this means that if a system produces long jobs that use one machine several times, we have to split the jobs to shorter pieces so that the assumptions of the flow-shop are not violated.

3. ACCELERATION OF MPEG-4 VIDEO DECODING AS A FLOW-SHOP SCHEDULING PROBLEM

MPEG-4 macroblock decoding can be fitted into the permutation flow-shop model by substituting the machines with decoding operations and jobs by coded 8x8 pixel blocks that will form the uncompressed video frame. Notice that the whole MPEG-4 video decoding process is not fitted into this model. The stream-, frame- and macroblock-header reading operations are left outside this model and we focus on macroblock decoding.

When all the possible variants of MPEG-4 macroblock coding are considered, the number of different decoding processes (job types) of one block can be limited to a reasonable number. Our testing method was limited to the decoding of sequences containing intra- and unidirectionally predicted frames with quarter-pixel decoding disabled. This setup produced 12 different job types. A 16x16 pixel MPEG-4 macroblock consists of six 8x8 blocks (in YUV 4:2:0), some of which might not be coded. We used a modified version of the XViD-codec (www.xvid.org) that is able to decode multiple streams simultaneously [2]. Thus, there were a maximum of 6 x *number of streams* jobs in each scheduling problem. The number of operations in the job types ranged between one and five.

The amount of machines is not as clearly defined, since the division of the decoding process into accelerators is somewhat arbitrary. We divided the MPEG-4 video decoding process into six accelerated functions that can be seen in Figure 3 as boxes with thick outlines. The numbers in the boxes indicate the corresponding machine index in the flow shop model. This was also the stage where the deterministic operation execution time requirement of flow-shop scheduling needed to be taken care of. The code division into accelerators was done so that the accelerated code executes in a time that can be considered to be fixed. Interpolation and AC/DC prediction are operations, which are not used for all macroblocks (in fact, they are exclusionary) and they are simply skipped when not required. The hardware accelerators were simulated by program code during these tests, although an FPGA-implementation is under development.

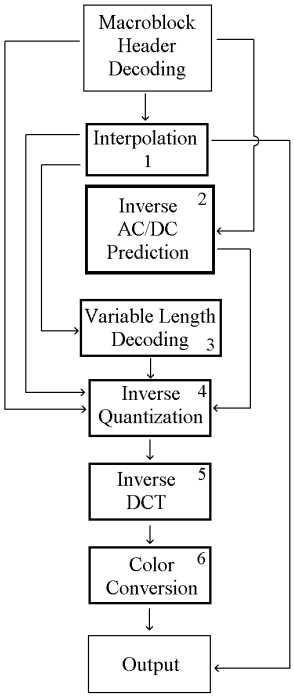

Fig. 3. Various control flows of MPEG-4 block decoding.

4. SOLVING PERMUTATION FLOW-SHOP PROBLEMS

We will first look at some conventional approaches to solve flow-shop scheduling problems and then discuss about the efficient implementation of particular algorithms.

4.1. Common approaches

Recently, many review papers have been published, that cover the area of flow-shop scheduling methods [1, 9, 10] and one that focuses on the permutation flow-shop problem [6]. Since the permutation flow-shop problem is known to

be NP-hard [6], there are heuristic algorithms as well as some methods that find optimal solutions for some restricted cases.

The permutation flow-shop problem can be solved as such, or by formulating it as an asymmetric travelling salesman (ATSP) problem [1]. This formulation enables the possibility to use ATSP-solving methods for scheduling, which is very interesting since there are a couple of methods that can find an exact solution to the ATSP-problem when the problem size is limited. However, most of the methods to solve permutation flow-shop problems (and ATSP-problems) are heuristics that look for a *good* solution in a reasonable time.

The procedure of permutation flow-shop scheduling consists essentially of the task to find the optimal order of jobs to minimize the makespan of the whole schedule. D. S. Palmer proposed a simple heuristic [14] to find a good order of jobs for the flow-shop problem. The method can be put simply in one clause: *those jobs, whose first operations tend to be long, should be placed first in the execution order*. The shape of each job is determined by computing a *slope index*. The order of the jobs in the schedule is then decided by sorting the jobs to an order of ascending slope indices.

G. Carpaneto and P. Toth published a breadth-first branch-and-bound method to solve ATSP problems optimally [3]. Later, Carpaneto *et al.* extended their method for large-scale (up to 2000 vertices) problems [4]. The source code of the large-scale algorithm is also available in public, but was left out from this study.

4.2. No-wait timetabling with no job ordering

This approach is the fastest of all scheduling approaches that are handled in this paper. Practically this approach means that we schedule the jobs to be executed in the order as they arrive to the scheduler. Figure 4 shows the essence of no-wait timetabling: the jobs are rigid sequences, which the scheduler has to slide as close to each other, as possible without overlapping any two operations. We can see from Figure 4 that operations A2, B2 and C2 are as close to each other as possible and therefore the schedule is optimal with the job order A, B, C. Notice how operation B1 can not be executed immediately after A1, because the delay between B1 and B2 has to be zero.

Since we assumed that we know all possible job types beforehand, we can build a look-up table at compile time that includes optimal inter-job distances for all job combinations. The size of this look-up table is obviously *N*x*N*, where *N* is the amount of job types. Thus, the only task left for run-time is timetabling. We want that the scheduler produces a list of operations that is ordered according to start times. This can be implemented efficiently by an array that effectively is a linked, ordered list. When we build a schedule, we loop through all jobs and operations and add each operation to the list, while maintaining the ascending start-time order by pointers.

While being the fastest scheduling method, this approach also produces the worst makespans, because the use of the look-up table limits the inter-job offset optimization to happen between two jobs only. There are some situations when a job does not use all machines that produce imperfectly optimized inter-job offsets. Generally semi-active timetabling is a better choice, but no-wait timetabling can also be interesting when some practical aspects are considered. If we assume that an operation passes information to the next operation within the same job, the no-wait approach may reduce the amount of data transfers and the need of buffer memory.

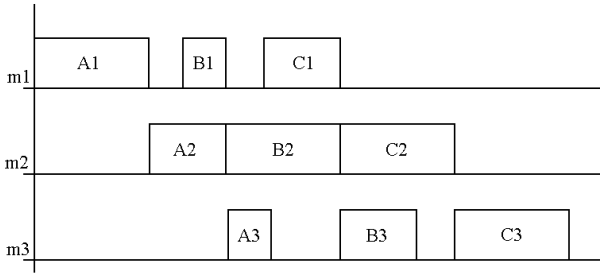

Fig. 4. No-wait timetabling schedule of three jobs.

4.3. Semi-active timetabling with no job ordering

In this approach we still do not do any job ordering. Now the *N*x*N* look-up table of the previous approach is not required, since the scheduler ensures at run-time that no operation overlap happens. Thus, the amount of run-time computations is also slightly higher. Figure 5 shows the same jobs as Figure 4, except that the schedule has been constructed with semi-active timetabling. Operation B1 can now start immediately after A1, since the waiting time between operations within a job need not to be zero anymore. Notice, that this has not affected B3. B3 can not immediately follow A3, since B2 must be finished before B3 can start.

4.4. Palmer job ordering

The job ordering heuristic of Palmer is essentially a slope-index sorting operation. The amount of run-time processing can be minimized by computing the slope indices to a look-up table at compile time. Again, this is possible because we assume to know all job types beforehand. When using the no-wait approach, timetabling can be done as swiftly as in Subsection 4.2, by the *N*x*N* look-up table.

The paper of Palmer [14] does not mention explicitly how to compute a slope index when machine skipping is allowed. Some alternatives were tried, but the best result was obtained by using Palmer's slope formula as usual and by setting the execution times of skipped operations to zero. The implementation of Palmer job ordering along with

semi-active timetabling is self-explanatory.

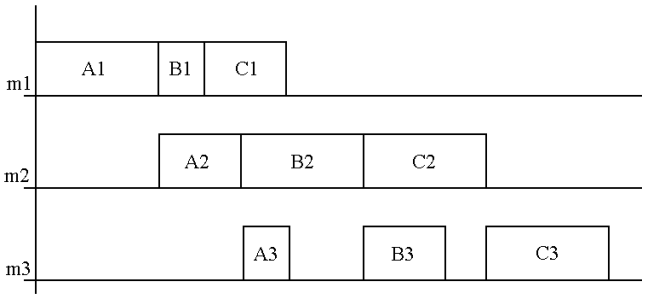

Fig. 5. Semi-active timetabling schedule of three jobs.

Method	Comp. Time (cycles)
No ordering	3489 (1.00)
Palmer ordering	5751 (1.65)

Table 1. Computation times of job ordering algorithms.

Method	Makespan (cycles)
No ordering, semi-active	12616 (0.80)
Palmer, semi-active	11836 (0.75)
No ordering, no-wait	15758 (1.00)
Palmer ordering, no-wait	14804 (0.94)

Table 2. Makespans, no machine skipping.

Method	Makespan (cycles)
No ordering, semi-active	9007 (0.79)
Palmer, semi-active	7775 (0.69)
No ordering, no-wait	11336 (1.00)
Palmer ordering, no-wait	11266 (0.99)

Table 3. Makespans, with machine skipping.

Method	Makespan	Comp. Time
No ord., semi-active	71864 (0.89)	17584 (1.25)
Palmer, semi-active	67178 (0.83)	22027 (1.57)
No ordering, no-wait	80999 (1.00)	14041 (1.00)
Palmer, no-wait	77906 (0.96)	18665 (1.33)

Table 4. Average makespan and computation time for scheduling the decoding of three MPEG-4 macroblocks (in cycles). Relative values are marked in parentheses.

4.5. Optimal job ordering by ATSP solving

The formulation of the permutation flow-shop problem into an asymmetric traveling salesman problem has two phases: (1) determining the respective parts of inter-city distances in the flow-shop problem, (2) the use of dummy jobs. The first phase can be done according to [1]: "intercity distance $d_{k,\,k+1}$ … is the time duration between the start of job k on machine M_1 and the earliest time at which job $k + 1$ can start on that machine by respecting the no-wait restriction when job $k + 1$ is scheduled next after job k". Since the amount of job types is known and fixed, it is possible to calculate the inter-city distances upon compile time and thus save on run-time computations. Notice that the intercity distance is not the same as the optimal offset that is used in no-wait timetabling.

The use of dummy jobs can be done according to [18], which results in two dummy jobs for each scheduling problem. Therefore a flow-shop problem with X jobs results in a $(X+2)*(X+2)$ ATSP matrix. This matrix is a subset of the intercity distance matrix. When the optimal job ordering has been acquired from the ATSP algorithm, no-wait or semi-active timetabling can be used as previously discussed.

5. TEST RESULTS

Altogether there were four different method combinations to test. We were mainly interested in the trade-off between the makespan of the schedule and the time used in building the schedule. The scheduling task consists of two major operations: job ordering and timetabling. Both "no ordering" and Palmer's job ordering heuristic were tested and timetabling was done with no-wait and semi-active approaches. The timetabling methods produce an ordered list of operations with ascending execution times.

The first test measured the computation times of the job ordering methods. We used a generalized model of multi-stream MPEG-4 video decoding: 20 random job types were defined, some of which skipped machines. There were 4 streams with 1 to 6 random jobs each, which resulted in 4 to 24 jobs per scheduling problem. The scheduling process was iterated 10000 times with different job numbers and –types to get reliable average results. The results in Table 1 show only the time used in job ordering by each algorithm. The job ordering duration of the "No ordering"-method is simply the time that was used to copy the job data to the correct data structures for timetabling. With proper arrangements the time used in this task can be made negligible.

The second test was performed with the same testing application. 20 random job types were defined so that every job used every machine (no machine skipping). There were 4 streams with 6 random jobs each, which resulted in 24 jobs per scheduling problem. The makespans of operations were random numbers between 1 and 1000. 10000 iterations were used in this test also. Table 2 shows the results for this test. The third test was similar to test number two, with the exception that machine skipping was now allowed. Results can be seen in Table 3.

The fourth test was performed by including somewhat optimized versions of the algorithms into a real software MPEG-4 video decoder. The decoder is a modified version of the XViD codec and can decode multiple streams simultaneously. The decoder was commanded to decode three 320x240 streams for the first 20 frames. Each stream was read macroblock by macroblock and decoded in the order that was determined by the scheduler. The makespans

and computing times provided by the algorithm combinations can be seen in Table 4. Notice that here the computing times include the timetabling operation, unlike in Table 2.

From Table 4 we can see that the makespan of the best-quality schedule is 13821 clock cycles shorter than the worst schedule. On the other hand, computing the best schedule takes 7986 cycles more than computing the worst schedule. Despite the drawback, the use of the best-quality scheduler is advisable here, since it improves the throughput by 13821-7986=5835 clock cycles per macroblock. The difference is only 6-7% of the overall throughput, so it can not be assumed that the combination of the Palmer heuristic and semi-active ordering would always provide the best throughput.

To find a bottom line for scheduling speed, a further optimized version of the no ordering, no-wait method was created inside the XViD decoder. This scheduler created schedules for a group of four macroblocks (= 4*6*(1 … 5) operations) in 17064 clock cycles (average). The schedule computing time was mostly consumed in timetabling in this case. The sum of the durations of the actual decoding operations was 477003 clock cycles by average, which means that the scheduler consumed 3.6% of the clock cycles. The time to decode a single macroblock (nearly 120000 cycles) is quite high because of some inefficiency in the modified decoder code, but on the other hand, the scheduler could probably also be optimized further.

6. CONCLUSIONS

In this paper, we have motivated and studied the problem of low overhead dynamic scheduling for fine grained acceleration of signal processing systems. We have drawn correspondences between this problem and classical flow-shop scheduling methods, and we have presented ways to efficiently implement such scheduling methods that lead to effective mechanisms for coordinating fine-grained acceleration. We have demonstrated our methods on an MPEG-4 video decoding application. Based on test results, it can be seen that the proposed scheduling methods can be used as a substitute of interrupt-based synchronization in fine-grained hardware accelerator synchronization.

7. ACKNOWLEDGEMENT

We would like to thank Tamas Erdelyi for testing and implementing several scheduling methods. We are also grateful to professor Johan Lilius of Åbo Akademi for several constructive remarks regarding the scheduling methods.

8. REFERENCES

[1] T.P. Bagchi, J.N.D. Gupta and C. Sriskandarajah, “A review of TSP based approaches for flowshop scheduling”, European Journal of Operational Research , Volume 169, Issue 3 , 16 March 2006, Pages 816-854.

[2] J. Boutellier, O. Silvén, T. Erdelyi, “Restructuring a Software based MPEG-4 Video Decoder for Short Latency Hardware Acceleration”, In: Proceeding of SPIE Electronic Imaging 2007, volume 6507, 2007.

[3] G. Carpaneto and P. Toth, “Some new branching and bounding criteria for the asymmetric traveling salesman problem”, Management Science 26, 1980, Pages 736–743.

[4] G. Carpaneto, M. Dell Amico and P. Toth, “Exact solution of large-scale, asymmetric traveling salesman problems”, ACM Transactions on Mathematical Software 21, 1995, pages 394–409.

[5] L. A. Cortes, P. Eles, Zebo Peng, “Quasi-static scheduling for multiprocessor real-time systems with hard and soft tasks”, Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on 17-19 Aug. 2005, Pages: 422 – 428.

[6] J.M. Framinan , J.N.D. Gupta and R. Leisten, “A review and classification of heuristics for permutation flow-shop scheduling with makespan objective”, Journal of The Operational Research Society, Volume 55, 2004, Pages 1243-1255.

[7] S. French, *Sequencing and Scheduling: an Introduction to the Mathematics of the Job-Shop*, Ellis Horwood Ltd, Chichester, 1982.

[8] G. de Goede, *Accelerating the XViD IDCT on DAMP*, Master’s Thesis, Delft University of Technology, 2005.

[9] J. N. D. Gupta and E. F. Stafford, “Flowshop scheduling research after five decades”, European Journal of Operational Research Volume 169, Issue 3, 16 March 2006, Pages 699-711.

[10] Tamás Kis and Erwin Pesch, “A review of exact solution methods for the non-preemptive multiprocessor flowshop problem”, European Journal of Operational Research, Volume 164, Issue 3, 1 August 2005, Pages 592-608.

[11] Y.-K. Kwok and I. Ahmad, “Benchmarking and Comparison of the Task Graph Scheduling Algorithms”, Journal of Parallel and Distributed Computing, Volume 59, Issue 3, December 1999, Pages 381-422.

[12] Nam Ling and Nien-Tsu Wang, “A Real-Time Video Decoder for Digital HDTV”, Journal of VLSI Signal Processing, Volume 33, 2003, Pages 295 – 306.

[13] Z. Ma, C. Wong, P. Yang, J. Vounckx, F. Catthoor, “Mapping the MPEG-4 Visual Texture Decoder”, IEEE Signal Processing Magazine, Volume 22, Issue 3, 2005, Pages 65 – 74.

[14] D. S. Palmer, “Sequencing jobs through a multistage process in the minimum total time: a quick method of obtaining a near optimum”, Operations Research Quarterly, Volume 16, 1965, Pages 101–107.

[15] T. Rintaluoma, O. Silvén, J. Raekallio, “Interface Overheads in Embedded Multimedia Software”, Lecture Notes in Computer Science 4017, 2006, Pages 5-14.

[16] S. Sriram, S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, Marcel Dekker Inc., Basel, 2000.

[17] S.-H. Wang, W.-H. Peng, Y. He, G.-Y. Lin, C.-Y. Lin, S.-C. Chang, C.-N. Wang and T. Chiang, ”A Software-Hardware Co-Implementation of MPEG-4 Advanced Video Coding (AVC) Decoder with Block Level Pipelining”, The Journal of VLSI Signal Processing, Volume 41, 2005, Pages 93 - 110.

[18] D.A. Wismer, “Solution of the flowshop scheduling problem with no intermediate queues”, Operations Research 20, 1972, Pages 689–697.