Journal of Signal Processing Systems, 2009, doi:10.1007/s11265-009-0366-z.

A Low-overhead Scheduling Methodology for Fine-grained Acceleration of Signal Processing Systems

JANI BOUTELLIER

Machine Vision Group, University of Oulu, P.O.Box 4500, 90014 Finland

jani.boutellier@ee.oulu.fi, Tel. +358 8 553 2814, Fax +358 8 553 2612

SHUVRA S. BHATTACHARYYA

Electrical and Computer Engineering Department, University of Maryland, College Park, MD, USA

OLLI SILVÉN

Machine Vision Group, University of Oulu, P.O.Box 4500, 90014 Finland

Abstract. Fine-grained accelerators have the potential to deliver significant benefits in various platforms for embedded signal processing. Due to the moderate complexity of their targeted operations, these accelerators must be managed with minimal run-time overhead. In this paper, we present a methodology for applying flow-shop scheduling techniques to make effective, low-overhead use of fine-grained DSP accelerators. We formulate the underlying scheduling approach in terms of general flow-shop scheduling concepts, and demonstrate our methodology concretely by applying it to MPEG-4 video decoding. We present quantitative experiments on a soft processor that runs on a field-programmable gate array, and provide insight on trends and trade-offs among different flow-shop scheduling approaches when applied to run-time management of fine-grained acceleration.

Keywords: *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 scheduled and synchronized in some manner to keep the results consistent. Most applications that use multiple processors, can handle their communication with a schedule that is static and computed at compile time.

However, in the recent years several applications have emerged that run with underutilized processors, if pre-computed fully static schedules are used. In such applications, the schedules should be computed at run-time, which can lead to high computational overheads if the scheduling algorithm is inefficient.

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 optimization can vary [1], 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.

1.1 The benefits of fine-grained acceleration

In this work we assume that the scheduled co-processors are hardwired accelerators. Hardwired accelerators are important in mobile applications, since their energy-efficiency is up to 50x higher than that of software implementations of the same algorithm [2]. In this paper we assume that the turnaround times of accelerators are short compared to traditional accelerators [3], *i.e.* they are *fine-grained*. Instead of using one *monolithic* accelerator, the accelerated functionality is implemented on several smaller units.

Hardware accelerators can be considered fine-grained when their execution time is around 100-1000 clock cycles, which is a fraction of the runtime of a coarse-grained hardware accelerator. Recently, Silvén *et al.* identified that some important modern-day applications such as video and baseband processing can benefit from fine-grained hardware accelerators, if they are used instead of the traditional coarse-grained ones [3].

This finding has also shown to be very true in the upcoming Reconfigurable Video Coding (RVC) standard [4]. In RVC, existing and future video decoders are implemented by combining a set of standard video coding tools originating from a pre-defined library. These library components are rather fine-grained and if they are translated into hardware accelerators to achieve high performance, their execution times become very short. Also present-day applications such as MPEG-4 can benefit from fine-grained hardware accelerators as we will show later in this work.

Fine-grained hardware acceleration brings up new problems that do not exist with traditional coarse-grained acceleration. Rintaluoma *et al.* showed that synchronization primitives such as interrupts and polling can create prohibitive overheads [5]. Moreover, all scheduling activities performed at run-time can slow down the system tremendously, since the scheduler needs to be invoked much more frequently than in coarse-grained systems.

In this paper we shall introduce some very fast scheduling methods and discuss the areas where they can be used. It will also be shown how to apply run-time scheduling of fine-grained hardware accelerators to MPEG-4 video decoding. Finally, the performance of the shown algorithms is measured on a field-programmable gate array (FPGA) and the results are analyzed. A part of this work has been published previously in [6]. In addition to general improvements, this paper extends the previous work by offering a more theoretical analysis of the results, as well as new experimental results that have been acquired from running the experiments on a soft RISC processor.

1.2 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 100-1000 clock cycles. Figure 1 shows an example of a fine-grain accelerated system architecture with shared memory. Fine-grain accelerated functions can be, *e.g.,* the computation of an 8x8 IDCT or interpolation, both of which can be done in less than 100 clock cycles [7, 8]. MPEG-4 video decoding is an important application for which fine-grained hardware accelerators can yield significant benefits [9]. Also, the fine granularity enables possibilities to accelerate general-purpose functions that can be shared by different applications, just like it is done in RVC [4]. 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, statically scheduled macroblock decoder chip can end up running half-idle because some macroblocks are only partially coded.

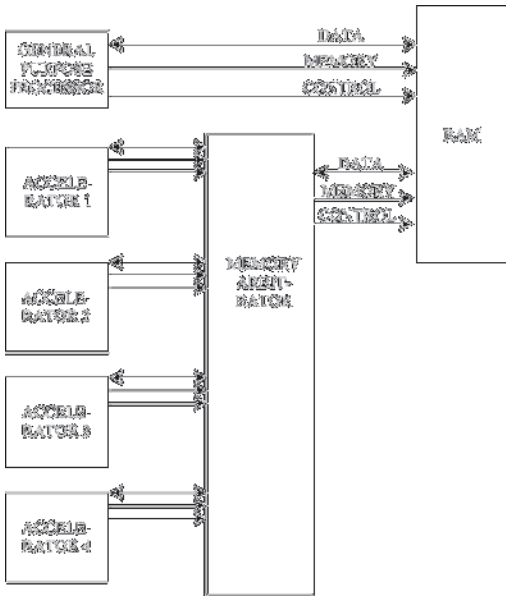

Fig. 1 Architecture of a fine-grain accelerated shared-memory system

As the fine-grained accelerators are intended to be invoked very frequently, the synchronization and scheduling mechanisms need to be efficient, not to cause unreasonable overhead. Since the hardware accelerators are assumed to be application specific, we can assume that their low-level synchronization is handled efficiently.

On the other hand, for the CPU that uses the hardware accelerators, we cannot make this assumption. When a system contains coarse-grained accelerators, it is usually assumed that the accelerator interrupts the CPU when it has finished its task. With fine-grained accelerators this is not possible, since this would practically prohibit the CPU from doing anything else than serving interrupts. Rintaluoma *et al.* noticed this and proposed that with fine-grained hardware acceleration the execution times of accelerators can be assumed to be deterministic [5]. For functions that do not have deterministic execution times, it is possible to consider their worst-case (WC) execution times. If all accelerated functions have known deterministic or WC execution

times, the CPU does not need to be interrupted when an accelerator finishes its task, since it is known when this has happened.

Static and WC execution times solve the overhead problems caused by synchronization primitives, but cannot solve the underutilization problem of accelerators when static schedules are used in dynamic applications. As mentioned previously, this problem can be solved by tailoring the accelerator invocation schedules according to the application needs that are resolved at runtime.

However, the scheduling has to be done efficiently, which can be shown easily by a brief example: Suppose we have an 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 10000 clock cycles, the scheduling alone consumes 75 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 [10]. Wang *et al.* have implemented a MPEG-4 video decoder with pipelined fine-grained accelerators [11]. 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 [13], 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 and 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, Section 5 shows and Section 6 analyzes the measurement results.

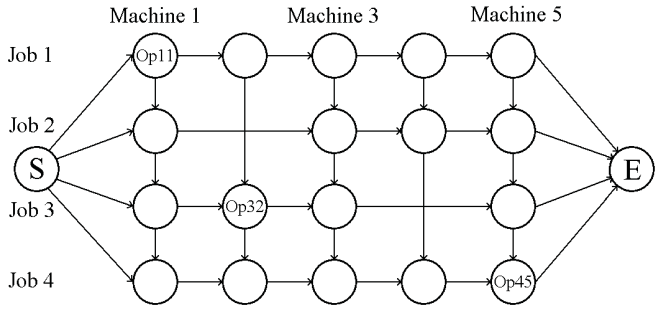

Fig. 2 A permutation flow-shop problem shown as a directed, acyclic graph

2 The flow-shop scheduling problem

One of the aims 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 [14] 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 (flow-shop operations must not be confused with microprocessor operations). 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 [14]. 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 out of scope of this work.

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 [1], 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* [1]. 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*) is started immediately after the previous operation (*n-1*) within the same job has finished. More alternatives are discussed in [1], which is a good source of information for the interested reader.

Generally computer science –oriented scheduling problems are modelled with *directed, acyclic graphs* (DAGs) [15, 16] that model the interdependencies of different computing operations. Figure 2 shows a DAG formulation of a permutation flow-shop problem after the job order has been determined. 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.

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 number 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 number of operations 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. If the decoding functionality is limited to sequences containing intra- and unidirectionally predicted frames with quarter-pixel decoding disabled, the setup produces 13 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.

The number of machines must be defined explicitly, since the division of the decoding process into accelerators is somewhat arbitrary. As an example, the MPEG-4 video decoding process has been divided into seven 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. In Table 1, the presence of accelerators for each job is shown. Letters *a* and *b* indicate that there are two versions of the same accelerated function, whereas an *x* means that only a single version exists. A more detailed and slightly different partitioning can be found in [17].

This is also the stage where the deterministic operation execution time requirement of flow-shop scheduling needs to be taken care of. Several different versions of accelerated tasks (operations) exist and each of them has a characteristic execution time. The execution times of tasks may vary from job to job, but need to be static for different instances of the same job. The partitioning of the program into accelerators was done so that the accelerated code executes in a time that can be considered to be fixed.

Table 1 MPEG-4 video macroblock decoding jobs

Macroblock coding parameters	1	2	3	4	5	6	7
Intra, coded, h263 quant, pred.dir 1		x	x	a	a	x	x
Intra, coded, MPEG quant, pred.dir 1		x	x	a	b	x	x
Intra, predicted, h263 quant, pred.dir 1		x		a	a	x	x
Intra, predicted, MPEG quant, pred.dir 1		x		a	b	x	x
Intra, coded, h263 quant, pred.dir 2		x	x	b	a	x	x
Intra, coded, MPEG quant, pred.dir 2		x	x	b	b	x	x
Intra, predicted, h263 quant, pred.dir 2		x		b	a	x	x
Intra, predicted, MPEG quant, pred.dir 2		x		b	b	x	x

Inter, predicted	x		x				
Inter, coded, h263 quant	x		x		a	x	x
Inter, coded, MPEG quant	x		x		b	x	x
Inter, GMC, h263 quant			x		a	x	x
Inter, GMC, MPEG quant			x		b	x	x

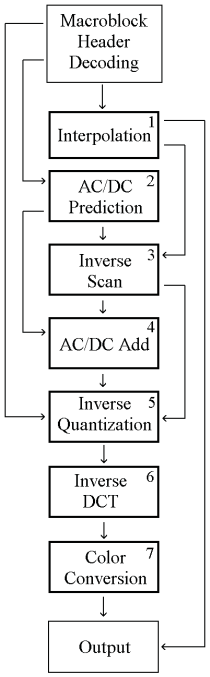

Fig. 3 Various control flows of MPEG-4 block decoding

4 Solving permutation flow-shop problems

We will first look at some conventional approaches to solve flow-shop scheduling problems and then discuss 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 [14, 18, 19] and one that focuses on the permutation flow-shop problem [20]. Since the permutation flow-shop problem is known to be NP-hard [20], 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 [18]. This formulation enables the possibility to use ATSP-solving methods for flow-shop 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 [21] 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 short, should be placed first in the execution order*. The priority 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 [22]. Later, Carpaneto *et al.* extended their method for large-scale (up to 2000 vertices) problems [23]. 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 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 maximally close to each other and therefore the schedule is optimal with the job order A, B, C. Notice how operation B1 cannot 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 number of job types. Thus, the only task left for run-time is timetabling.

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.

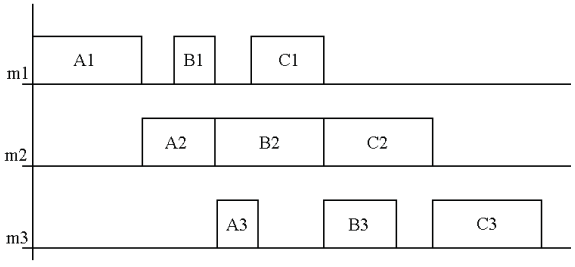

Fig. 4 No-wait timetabling schedule of three jobs

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 cannot be established, since the operation start times within a job cannot be planned in advance. 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.

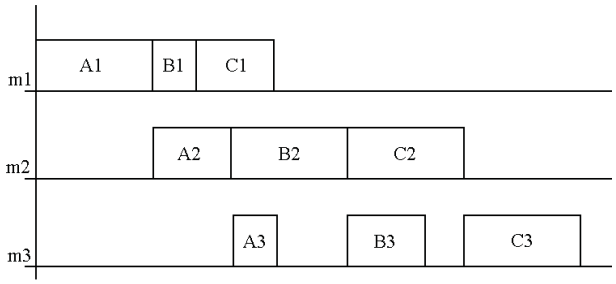

Fig. 5 Semi-active timetabling schedule of three jobs

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 [21] does not mention explicitly how to compute a slope index when machine skipping is allowed. Some alternatives were tried and 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.

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 [18]: "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 number 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 inter-city distance can be interpreted as the optimal offset that is used in no-wait timetabling.

The use of dummy jobs can be done according to [24], 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 inter-city 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. In practice, the complexity of ATSP solving is so high that it is not feasible for run-time scheduling. For this reason, we omitted this approach in our experiments.

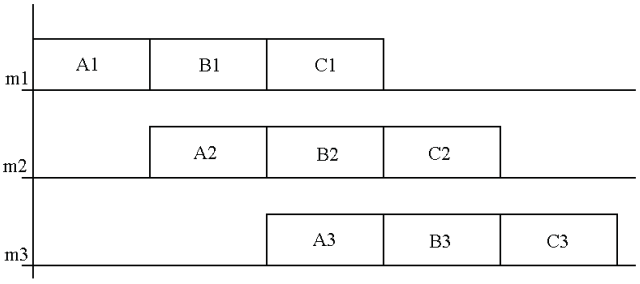

Fig. 6 Optimally balanced pipeline

4.6 Task dispatching

Until now we have only discussed how to compute a schedule for the accelerator invocations. However, there is also another aspect to the problem, namely the dispatching of scheduled operations. If we assume that there is no separate hardware available for the accelerator invocation, the CPU must commit itself partially or fully for this procedure. A fully committed CPU continuously checks if the dispatching time of some task has come and initiates the accelerator once the time comes. This means that the CPU cannot perform any useful work at the same time.

To allow the CPU to work on something else while taking care of dispatching, a regular interrupt must be created (in literature called *timer tick*). When the timer interrupts the CPU while it is working on something else, the CPU goes to the interrupt service routine and checks if any new operations must be dispatched. A higher timer tick frequency means that tasks are dispatched more accurately, but the dispatcher overhead grows.

Solving the task dispatching problem by software is challenging, since there are also situations when multiple tasks need to be started simultaneously on parallel accelerators. This happens for example, when the accelerators have been pipelined and balanced optimally, as shown in Figure 6.

In fine-grained hardware acceleration the task dispatching on a CPU can easily become a problem. If the average execution times of accelerators are, *e.g.*, around 1000 cycles, the CPU is constantly required to dispatch a task, especially if several accelerators run in parallel. These facts clearly show that in fine-grained hardware accelerated systems, hardware-assisted dispatching would be of great help.

5 Experimental results

In our previous publication [6] we measured the performance of the above mentioned four schedulers on a workstation. Now we will elaborate these results by performing them on an Altera Cyclone III FPGA that runs a NIOS II/f soft processor. Compared to the results of [6], the scheduling algorithms were re-written and optimized in the C – language, although their functional behavior remained the same. We will first introduce the experiments that were made and then show the numerical results. In Section 6 we will analyze the results.

Altogether there were four different scheduling method combinations that were tested. We were mainly interested in the trade-off between the makespan of the schedule and the time used in building the schedule. The scheduling procedure consists of two major operations: job ordering and timetabling. Both "no ordering" and Palmer's job ordering heuristic were experimented and timetabling was done with no-wait and semi-active approaches.

In the first experiment we used a generalized model of multi-stream MPEG-4 video decoding (similar to [25]): 20 random job types were defined, some of which could *skip* some of the 6 machines in the system. There were 4 streams with 1 to 6 random jobs each, which resulted in 4 to 24 jobs per scheduling problem. The experiment was conducted on various average operation lengths to see which scheduling factors are independent of the operation length. The standard deviation of operation lengths was about 50% of the average operation length. The scheduling process was iterated 10000 times with different job numbers and –types to get reliable average results. The results in Table 2 show the results for each algorithm. The following notation is used:

T_s average scheduling time,

L_s average sequential schedule makespan,

L_p average parallel schedule makespan,

T_t sum of average scheduling time and average parallel schedule makespan,

L_o average scheduled operation length,

N_o average number of operations to be scheduled for each scheduler invocation,

K_p throughput compared to sequential execution.

Table 2 Execution times and makespan results with machine skipping. Units are CPU clock cycles, except for K_p, which is a plain coefficient

Algorithm	T_s	L_p for $L_o = 100$	L_p for $L_o = 500$	L_p for $L_o = 1000$	K_p
No ord., No-wait	2775	2321	11558	23103	1.93
Palmer, No-wait	4532	2332	11423	22835	1.95
No ord., Semi-A.	3067	1720	8563	17117	2.61
Palmer, Semi-A.	4820	1483	7383	14759	3.02
Sequential exec.	0	4482	22330	44637	1.00

The average number of operations that were scheduled with one scheduler invocation was 45. With this information it was possible to define a *speedup factor* for each scheduling algorithm. The speedup factor tells the best achievable program speedup with the respective scheduling algorithm using parallel machines. The factor expresses the speedup compared to sequential execution of the same program on one machine. Thus, a speedup factor of 2.00 for scheduling algorithm *A* would mean that if *A* is used to schedule a parallel system, it will run twice as fast as a one machine solution. The speedup factor is dependent on the number of parallel machines in the system, as well as average lengths and numbers of operations in the scheduled jobs.

The same experiments were also done with a different set of jobs that did not skip any of the 6 machines. Again, 20 random job types were defined so that there were 4 streams with 6 random jobs each, which resulted in 24 jobs per scheduling problem. 10000 iterations were used in this

experiment also. This time the average number of operations per scheduler invocation was 84. Table 3 shows the results for this experiment.

Table 3 Execution times and makespan results with no machine skipping. Units are CPU clock cycles, except for K_p, which is a plain coefficient

Algorithm	T_s	L_p for $L_o = 100$	L_p for $L_o = 500$	L_p for $L_o = 1000$	K_p
No ord., No-wait	3960	3047	15149	30279	2.76
Palmer, No-wait	5672	2860	14224	28432	2.94
No ord., Semi-A.	4796	2439	12138	24264	3.44
Palmer, Semi-A.	6513	2288	11383	22757	3.67
Sequential exec.	0	8396	41815	83601	1.00

The used NIOS II processor had 2kB of data cache and 2kB of instruction cache. The whole benchmark program was executed from on-chip memory. We also conducted brief experiments to see how the algorithms slow down when there is no on-chip memory available (except caches) and everything is on off-chip SDRAM: the algorithms slowed down at most 29%. Some algorithms were affected less, but the results are not shown to save space.

6 Analysis of results

To help the analysis of the results presented in the previous section, a couple of equations were derived. With the help of these equations it was possible to draw informative graphs that reveal interesting facts about the scheduling algorithms and the problem at hand.

It was decided that the performance of each scheduling algorithm could be analyzed best by comparing it to the performance of sequential uniprocessor execution. For this, we shall define the speedup factor S:

$$S = L_s / T_t. \quad (1)$$

The total time used by parallel execution can also be expressed by

$$T_t = L_p + T_s \quad (2)$$

and the length of the parallel schedule can be defined as follows:

$$L_p = L_s / K_p. \quad (3)$$

K_p is a scheduling algorithm-specific value that depends on the number of operations scheduled and number of machines in the system. However, it is not dependent on the average operation length L_o.

Now the speedup coefficient can be rewritten as

(4) $$S = L_s / (T_s + L_s / K_p).$$

Statistically, it also holds that

(5) $$L_s = L_o N_o.$$

Finally, we can formulate the equation for S in its final form:

(6) $$S = 1 / (T_s / (L_o N_o) + 1 / K_p).$$

With the help of this equation, we can draw two different figures for each scheduling algorithm. As it can be seen from Tables 2 and 3, T_s and K_p are constants when the number of processors (machines) and the set of jobs remain unchanged. The first figure is created from the data set shown in Table 2.

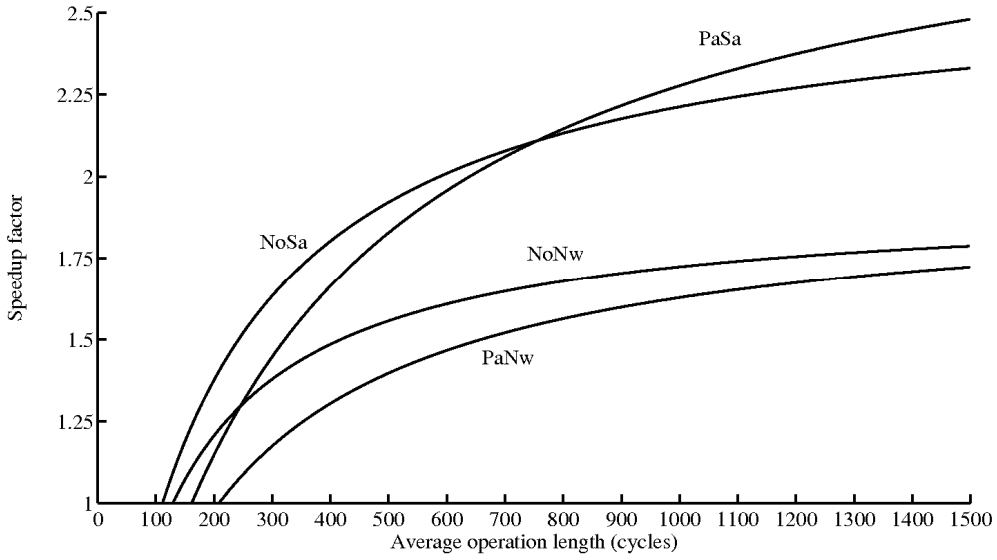

Fig. 7 The speedup provided by run-time scheduled parallel processing units

Figure 7 shows the speedup provided by the presented scheduling algorithms, as a function of average operation length. This result applies to the randomly generated job set that does machine skipping and maps the task to 6 hardware accelerators. The curves in Figure 7 were computed with equation 6 from the Table 2 data set. Figure 7 tells that run-time scheduling starts to provide benefit over sequential uniprocessor execution when the average operation length is more that 150 clock cycles. Also, it can be seen that the best speedup is provided by the "no-ordering, semi-active" algorithm when the average operation length is less than 800. For longer operations the more complex "palmer ordering, semi-active" algorithm provides better results. This is due to the fact that "palmer ordering, semi-active" offers a better throughput, but also produces a higher overhead.

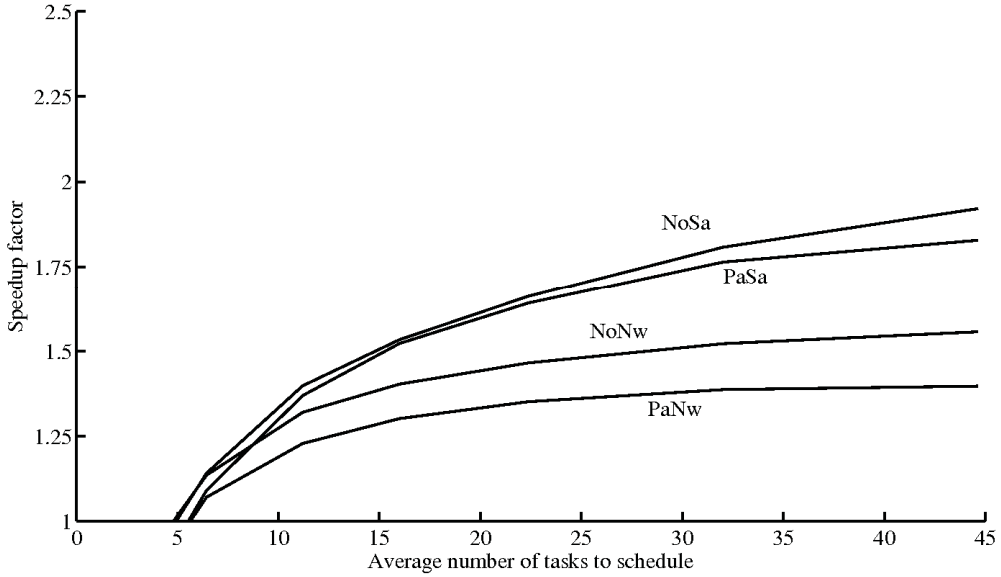

Fig. 8 The speedup provided by run-time scheduled parallel execution as a function of task count

The curves in Figure 8 have also been computed with the help of equation 6. This figure displays the speedup factor as a function of number of tasks to schedule for L_o = 500. As K_p is dependent on the number of tasks (operations) scheduled, the speedup factor had to be computed discretely based on a set of measured K_p values. It can be seen that run-time scheduled parallel execution starts to provide a benefit over uniprocessor execution with already less than 10 tasks. Again, these results only apply for L_o = 500, a system with 6 accelerators and the set of jobs that was used in this experiment.

Similar results were also computed for Table 3. To save space, these results are only explained briefly without figures. The data set of Table 3 differs from the data set of Table 2 only by the different set of jobs, which do not do machine skipping. As a result, the scheduling algorithms had more operations to schedule, per job. That increased the scheduling time, but on the other hand provided better speedup factors by better accelerator utilization. These consequences can also be observed directly from Table 3 by looking at the columns T_s and K_p.

The analysis shows that with our experimental setup, run-time scheduling starts to be beneficial when the scheduled tasks are longer than 150 clock cycles. Unfortunately, some of the fine-grain accelerated functions that were mentioned in Subsection 1.2 are shorter than this. An evident solution to this is to transfer the scheduling functionality to a hardware unit that can perform faster.

7 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 shown how to apply this methodology to MPEG-4 video decoding. Finally, the performance of the presented scheduling algorithms has been measured on a field-programmable gate array and analyzed thoroughly. The results show that run-time scheduling enables the dynamic use of fine-grained hardware accelerators. However, hardware assistance for scheduling and task dispatching could improve the results even further. This is a useful direction for further investigation.

8 Acknowledgments

This work has been partially funded by the Nokia Foundation, the US National Science Foundation (Grant number 0325119), Finnish Graduate School for Electronics, Telecommunication and Automation, and Tekes projects ECUUS and NECST.

9 References

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

[2] Dally, W.J., Balfour, J., Black-Shaffer, D., Chen, J., Harting, R., Curtis, P., V., Park, J., Sheffield, D. (2008). Efficient Embedded Computing, Computer, 41(7), 27–32.

[3] Silvén, O., & Jyrkkä, K. (2005). Observations on power-efficiency trends in mobile communication devices. In Embedded Computer Systems: Architectures, Modeling, and Simulation. SAMOS 2005 Proceedings, Lecture Notes in Computer Science 3553, 142–151.

[4] Lucarz, C., Mattavelli, M., Thomas-Kerr, J., Janneck J. (2007). Reconfigurable media coding: a new specification model for multimedia coders. Proceeding of the IEEE 2007 Workshop on Signal Processing Systems, 481–486.

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

[6] Boutellier, J., Bhattacharyya, S. S., Silven, O. (2007). Low-Overhead Run-Time Scheduling for Fine-Grained Acceleration of Signal Processing Systems. Proceedings of the 2007 IEEE Workshop on Signal Processing Systems, 457-462.

[7] Dang, P. P. (2006). An efficient VLSI architecture for H.264 subpixel interpolation coprocessor. International Conference on Consumer Electronics 2006, Digest of Technical Papers, 87–88.

[8] Chen, T.-H. (1999). A cost-effective 8×8 2-D IDCT core processor with folded architecture, IEEE Transactions on Consumer Electronics, 45(2), 333–339.

[9] de Goede, G. (2005). Accelerating the XViD IDCT on DAMP, Master's Thesis, Delft University of Technology, 2005.

[10] Ma, Z., Wong, C., Yang, P., Vounckx, J., Catthoor, F. (2005). Mapping the MPEG-4 Visual Texture Decoder, IEEE Signal Processing Magazine, 22(3), 65–74.

[11] Wang, S.-H., Peng, W.-H., He, Y., Lin, G.-Y., Lin, C.-Y., Chang, S.-C., Wang, C.-N., Chiang, T. (2005). A Software-Hardware Co-Implementation of MPEG-4 Advanced Video

Coding (AVC) Decoder with Block Level Pipelining, The Journal of VLSI Signal Processing, 41(1), 93-110.

[12] Ling, N. & Wang, N.-T. (2003). A Real-Time Video Decoder for Digital HDTV, Journal of VLSI Signal Processing, 33(3), 295–306.

[13] Cortes, L.A., Eles, P., Peng, Z. (2005). Quasi-static scheduling for multiprocessor real-time systems with hard and soft tasks. Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications 2005, 422–428.

[14] Gupta, J.N.D. & Stafford, E.F. (2006). Flowshop scheduling research after five decades, European Journal of Operational Research, 169(3), 699-711.

[15] Sriram, S. & Bhattacharyya, S.S. (2000). Embedded Multiprocessors: Scheduling and Synchronization, Basel: Marcel Dekker Inc.

[16] Kwok, Y.-K. & Ahmad, I. (1999). Benchmarking and Comparison of the Task Graph Scheduling Algorithms, Journal of Parallel and Distributed Computing, 59(3), 381-422.

[17] Stolberg, H.-J., Berekovic, M., Pirsch, P., Runge, H. (2001). The MPEG-4 Advanced Simple profile - a complexity study. Proceedings of Workshop and Exhibition on MPEG-4, 33–36.

[18] Bagchi, T.P., Gupta, J.N.D., Sriskandarajah, C. (2006). A review of TSP based approaches for flowshop scheduling, European Journal of Operational Research, 169(3), 816-854.

[19] Kis, T. & Pesch, E. (2005). A review of exact solution methods for the non-preemptive multiprocessor flowshop problem, European Journal of Operational Research, 164(3), 592-608.

[20] Framinan, J.M., Gupta, J.N.D., Leisten, R. (2004). A review and classification of heuristics for permutation flow-shop scheduling with makespan objective, Journal of The Operational Research Society, 55(12), 1243-1255.

[21] Palmer, D.S. (1965). Sequencing jobs through a multistage process in the minimum total time: a quick method of obtaining a near optimum, Operations Research Quarterly, 16(1), 101–107.

[22] Carpaneto, G. & Toth, P. (1980). Some new branching and bounding criteria for the asymmetric traveling salesman problem, Management Science, 26(7), 736–743.

[23] Carpaneto, G., Dell Amico, M., Toth, P. (1995). Exact solution of large-scale, asymmetric traveling salesman problems, ACM Transactions on Mathematical Software, 21, 394–409.

[24] Wismer, D.A. (1972). Solution of the flowshop scheduling problem with no intermediate queues, Operations Research, 20, 689–697.

[25] Schumacher, P., Denolf, K., Chilira-Rus, A., Turney, R., Fedele, N., Vissers, K., Bormans, J. (2005). A scalable, multi-stream MPEG-4 video decoder for conferencing and surveillance applications. Proceedings of the IEEE International Conference on Image Processing 2005, II - 886-889.