ARCHITECTURAL SUPPORT FOR THE ORCHESTRATION OF FINE-GRAINED MULTIPROCESSING FOR PORTABLE STREAMING APPLICATIONS

Jani Boutellier

Machine Vision Group
University of Oulu
90014 Oulu, Finland

Alessandro Cevrero, Philip Brisk, Paolo Ienne

Processor Architecture Laboratory
École Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland

ABSTRACT

Handheld devices are expected to start using fine-grained ASIC accelerators to meet energy-efficiency requirements of increasingly complex applications, *e.g.*, video decoding and reconfigurable radio. To avoid overhead, static multiprocessor schedules are preferable for orchestrating fine-grained accelerators. However, as modern applications use accelerators in irregular patterns, static scheduling leads to low hardware utilization.

Run-time scheduling for fine-grained accelerators solves the utilization problem, but easily produces significant overhead. We propose an efficient Accelerator Management Unit (AMU), implemented in hardware. *E.g.*, in video decoding, the AMU takes 3 to 18 cycles to compute a macroblock decoding schedule. The CPU may perform useful work, as the AMU does independent task dispatching.

Two experiments are performed, where the AMU is integrated into an FPGA-based multiprocessing prototype system. One experiment does AMU-orchestrated MPEG-4 video decoding and the other demonstrates that the AMU enables low-overhead dynamic scheduling and produces a significant performance advantage over static scheduling.

Index Terms— Scheduling, parallel processing, signal processing, application specific integrated circuits

1. INTRODUCTION

Portable consumer devices are continuously adopting new functionality, while consumers demand long battery lifetimes and low prices. Software execution even on high-end DSP processors cannot meet the demands of these current-generation systems. Instead, system designers have integrated *coarse-grained ASIC accelerators* to provide the necessary application speedup and energy-efficiency.

An example of such a system is shown in Figure 1(a). The CPU is responsible for offloading the computation to an accelerator and may work on something else as the accelerator executes. However, such a system is unscalable for future devices that need to support a plethora of video or radio standards. As an example let us consider the upcoming Reconfigurable Video Coding [1] standard. In RVC, previously standardized (MPEG-2, MPEG-4, ...) and future decoders are implemented from a combination of common *functional units* (FUs) used by most decoders. As hardware acceleration is required for efficient decoding, there should be one coarse-grained accelerator for each decoding standard, which would require a tremendous amount of chip area. The same applies for reconfigurable radio systems.

Silvén and Jyrkkä [2] proposed the use of fine-grained rather than coarse-grained accelerators, as shown in Figure 1(b); the functionality of the coarse-grained accelerator is distributed to several, separately accessible fine-grained accelerators (equivalent to the shared FUs in RVC). However, Rintaluoma *et al.* [3] noted that the high software overheads of traditional interrupt based synchronization would become prohibitive for fine-grained accelerators: context switches would occur with greater frequency than in the coarse-grained case, and cache pollution would ensue from each switch.

This paper advocates quasi-static run-time scheduling and orchestration of hardware accelerators. Sriram and Bhattacharyya [4] say that "a quasi-static scheduling technique tackles data-dependent execution by localizing run-time decision making to specific types of data dependent constructs such as conditional, and data dependent iterations." A schedule is constructed from static parts so that the decision making in scheduling is minimized. When a schedule has been computed at run-time, it stays fixed until all tasks have been dispatched. Pre-emption is not allowed. Our paper presents a circuit that performs the quasi-static scheduling and dispatching with negligible area and execution time overhead.

The runtime of a fine-grained accelerator, such as *inverse discrete cosine transform (IDCT)*, typically, is either deterministic or has a tight upper bound. *E.g.*, the IDCT ASIC by Rambaldi *et al.*, takes 80 cycles to compute [5]. If the sequence of accelerator invocations can be determined statically, then static schedules can be computed offline; however, if the sequence is data dependent, then the best a static schedule [6] can do while maintaining correctness, is to plan for the worst-case execution scenario; this will lead to poor acceler-

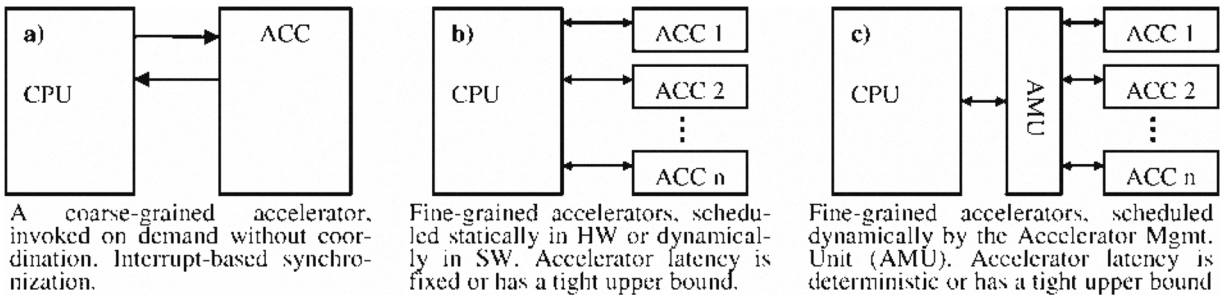

Fig. 1. Management of coarse- and fine-grained hardware accelerators.

ator utilization if the average execution sequence is far from the worst case. Dynamic scheduling is necessary.

Boutellier *et al.* [7] studied algorithms by which an operating system could dynamically schedule the accelerators. This fixes the utilization problem but forces the scheduler to compete for cycles on the CPU; the scheduler is invoked with high frequency and the performance of the scheduling heuristic becomes a limiting factor. The frequency with which the scheduler is invoked justifies its implementation as a decoupled hardware unit.

Figure 1(c) illustrates the solution advocated by this paper: the quasi-static scheduler is realized in an Accelerator Management Unit (AMU) in hardware. The benefits of the AMU are twofold: (1) the number of cycles required to schedule a task is reduced significantly compared to software; (2) task dispatching requires no CPU attention. A prototype AMU has been built and tested in an FPGA-based multicore soft processor system that performs MPEG-4 video decoding.

2. VIDEO DECODING OVERVIEW

Throughout the remainder of this paper, MPEG-4 Simple Profile video decoding will be used as an example. A video is an ordered sequence of *frames*, i.e., still images, which are displayed to the user. Video coding standards typically employ three types of frames: an Intra coded frame (I-frame) is coded like a JPEG image; a Predicted frame (P-frame) is computed from the previous frame and residual data; and a Bi-directionally predicted frame (B-frame) from two neighboring frames and residual data.

Each frame is divided into smaller units called *macroblocks*. Similarly as with frames, there are three kinds of macroblocks: I, P, and B. An I-frame contains only I-macroblocks; a P-frame contains I- and/or P-macroblocks; and a B-frame contains I-, P-, and/or B-macroblocks. A macroblock consists of six *blocks* that can be decoded in parallel. From the decoder's perspective, a video stream is a sequence of macroblocks, whose types are only discovered when the header data of each macroblock has been read.

I-, P-, and B-macroblocks (and their subtypes, not covered here) each require a different sequence of function invocations for decoding. Therefore, a static decoding function invocation schedule must dispatch functions based on worst-case assumptions in terms of function usage. Quasi-static scheduling, in contrast, adapts to the real function demand and schedules only those functions that are required by the macroblock(s) currently being decoded; the drawback is the runtime overhead of computing the schedule.

3. RELATED WORK

The real-time service providing co-processor Real-Time Unit (RTU) [9] supports static and priority-based scheduling. The SSCoP real-time scheduling co-processor [8] supports both static and online scheduling with different policies; it assumes that tasks are non-preemptable and have precedence constraints. Like the AMU, RTU and SSCoP target multiprocessor systems.

Al-Kadi *et al.* [10] have proposed a hardware task scheduler for fine-grained tasks. Their scheduler takes 15 cycles for scheduling and synchronization of tasks. Their solution is designed for homogenous multiprocessing systems and it also selects at run-time, to which processor the tasks are mapped. Our system, in contrast, is designed for heterogeneous multiprocessing systems, where the tasks have been mapped to processing elements (PEs) at design time. To our best knowledge, this paper is the first to propose a hardware scheduler for the orchestration of heterogeneous fine-grained PEs.

Table 1 summarizes the key points of the hardware implementations of the AMU and similar solutions. Many of the designs, including the AMU, are scalable and the values provided represent only one configuration; it is difficult to directly compare the different designs because different evaluation platforms have been used (*e.g.*, standard cells in different technologies).

4. THEORETICAL BACKGROUND

Boutellier *et al.* [7] have modeled the problem studied here as a permutation flow-shop (PFS) scheduling problem, which is NP-complete [11]. The PFS problem involves mapping N jobs onto M machines (accelerators) in the time domain.

Table 1. Comparison of scheduling co-processors.

Implementation	Policy	PEs	Tasks	Announced size	Features
SSCoP (0.80μm) [8]	Many	32	32	84372 transistors	schedule feasibility check
RTU (1.00μm) [9]	Many	3	64	25000 gates	multitasking kernel functionalities
Al-Kadi *et al.* (45nm) [10]	N/A	4	N/A	$0.048mm^2$	task synchronization
AMU	(E)PFS	4	4x32	3300 gates + 512B RAM	task dispatcher

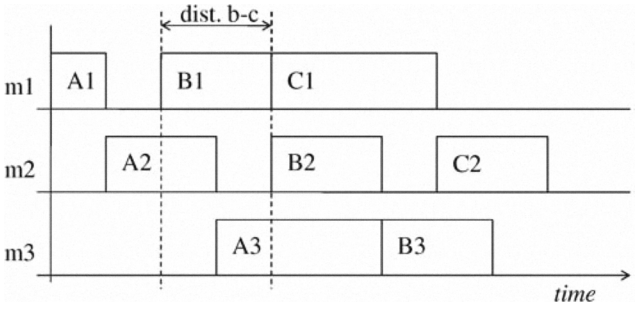

Fig. 2. Three jobs scheduled on three machines (accelerators).

Each accelerator is a highly customized ASIC, and all accelerators are unique; each job consists of a sequence of non-preemptable tasks that execute on dedicated accelerators. Figure 2 depicts three jobs (A, B and C) that have been assigned to three machines. Job C does not use machine three.

All tasks in all jobs use the machines in the same sequence; no job may use a machine more than once; however, no jobs are required to use all machines. Also, task m of job n has to finish before task $m+1$ of job n can start. Although these restrictions may seem strict, they characterize many industrially relevant applications such as video decoding. A solution to the PFS problem is an order in which jobs are processed by accelerators [12].

Timetabling describes the policy of determining the starting times of individual tasks within each job. *No-wait* timetabling ensures that each task starts its execution exactly as the preceding task within the same job terminates. *Semi-active* timetabling, in contrast, schedules operations using the *as soon as possible (ASAP)* heuristic. Previous work [7] showed that semi-active timetabling produces shorter makespans than no-wait timetabling; however, the software overhead of running semi-active timetabling repeatedly eliminated these gains. The AMU implements in hardware the *No Job Ordering, No Wait* scheduling heuristic [7] that also requires smaller buffer sizes between accelerators.

Additionally, the AMU supports the *extended permutation flow-shop* (EPFS) model that was introduced in [13]. EPFS extends PFS by allowing two or more tasks of the same job to execute simultaneously. With this extension, it is possible to model static multiprocessor *synchronous data flow* [4] schedules as EPFS jobs. Quasi-static EPFS schedules are used in one of the experiments of this paper.

5. THE ACCELERATOR MANAGEMENT UNIT

When the operating system performs run-time scheduling, both the schedule computation and task dispatching phases consume processor execution time. The scheduling procedure, which computes the activation timetable, is application-specific. A task dispatcher, in contrast, is common to all scheduling algorithms that do not allow schedule changes in the dispatch phase: a counter, which measures the passing of time, is compared against a timestamp of the next task to start. In software, performing this task every cycle is prohibitive without a customized counter; instead, a system service must periodically check for new tasks to start.

The AMU performs both schedule computation and task dispatching. After an initial reset, the CPU begins to execute; almost immediately, it starts issuing jobs for the AMU to schedule on the accelerators. Jobs may arrive at the AMU at arbitrary times. The AMU can schedule the job within three CPU clock cycles. Once the schedule has been computed, the AMU returns the schedule makespan to the processor. The AMU then begins to dispatch tasks, and the CPU may perform useful work.

5.1. Implementation environment

A set of Altera NIOS II soft processors [14] were synthesized on our Altera Cyclone III FPGA testbed, along with the AMU (all sharing a 50MHz clock). One of the processors acted as the CPU while the others mimicked hardware accelerators under control of the AMU. Altera's general I/O interface facilitated communication between the CPU and the AMU. The signal lines from the AMU to each accelerator delivered a wake-up signal and 8 bits of application-specific data, sent to the AMU from the CPU along with the scheduled jobs.

The AMU's scheduling algorithm was also written in C (optimized for minimum execution time) and executed on the Nios II processor. In software, the scheduling of one job took more than 340 clock cycles; the AMU, in contrast, requires just *three* cycles for this, and also performs independent task dispatching. Lastly, we synthesized the AMU using a UMC 180nm CMOS process and Faraday standard cell library. The AMU, supporting four accelerators and 32 job types (see Section 5.2), required 3300 standard equivalent gates, ignoring the 512 byte RAM. This is considerably smaller than the RTU scheduler [9], whose area, as shown in Table 1 is 25000 gates. The AMU's maximum operating frequency was 526 MHz.

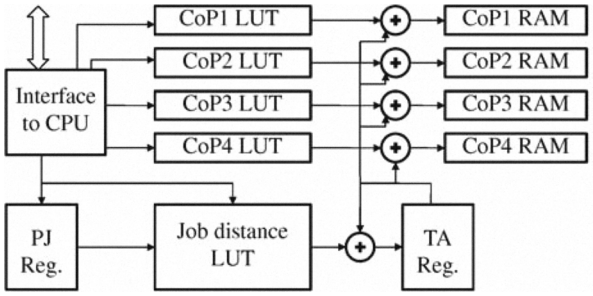

Fig. 3. The scheduler part of our AMU.

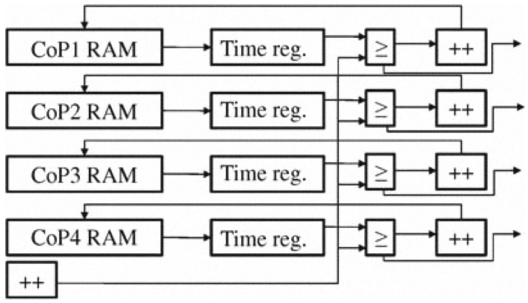

Fig. 4. The dispatcher part of our AMU.

5.2. Detailed AMU functionality

Figure 3 provides an overview of the AMU's scheduler. Each job is a set of tasks; the set of different job types in an MPEG-4 video coding system is fixed and relatively small. What is unknown at design time, is the specific types of jobs required to decode each macroblock. The tasks' functionalities are hard-coded in the accelerator circuits, whereas their possible invocation patterns (job types) are determined offline and stored in a LUT, called *CoPx LUT* in Fig. 3. The *CoPx LUT* contains the accelerators' execution modes for each job type and the time offsets of accelerator invocations. *E.g.*, for task A2 in Figure 2, the *CoP2 LUT* would provide the operation mode of A2 ("write output to A3") and the starting time offset with respect to the starting time of A1.

The *Job distance LUT* contains the time offsets for scheduling a pair of jobs in the system in isolation; this is permitted because no-wait timetabling has been used; each LUT entry only contains an inter-job distance (dist. b-c in Figure 2) that is a single integer value. *E.g.*, for jobs j_1 and j_2, if j_1 starts executing at time t, starting time of j_2 is t + JobDistanceLUT[j_1, j_2]. Here, j_1 stands for the previous job and j_2 for the present job. The type of the previous job is maintained in a register named *PJ*. The use of the *Job distance LUT* significantly reduces the AMU latency.

The top-left corner of Figure 3 depicts the CPU interface to the AMU. The CPU provides data words that carry job identification numbers (IDs), control signals and some application specific data that the AMU redirects to the accelerators

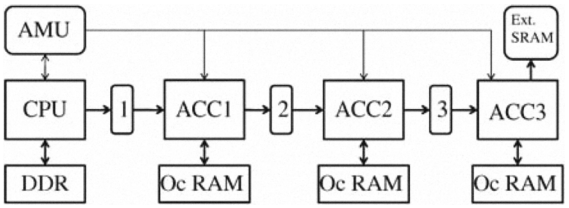

Fig. 5. The system used for experiment 1.

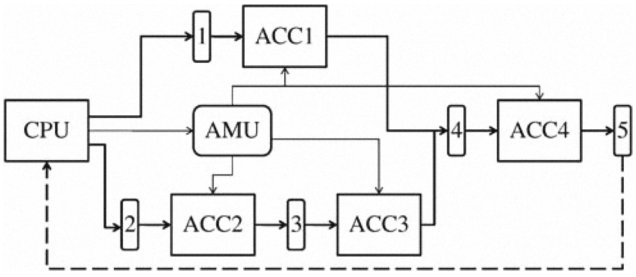

Fig. 6. The system used for experiment 2.

upon dispatching. When the AMU receives a data word, the job ID is extracted from it and used to access the *CoPx LUT* related to each accelerator. This provides the starting time offset for the present task (*e.g.*, A3) with regard to the previous task (*e.g.*, A2) in the same job (A).

Concurrently, the AMU reads a value from the *Job Distance LUT* and updates it to the *TA* register that accumulates the job starting times; this value is added to the task offset read from the *CoPx LUT*. This value is then written to the RAM that is depicted in Figure 3. The latency between the arrival of a data word at the AMU and a write to the RAM is three clock cycles.

Figure 4 shows the dispatcher. Once the application on the CPU has sent all the jobs to the AMU, the RAM contains a list of tasks to execute on each accelerator, sorted in ascending order of starting times. When the CPU gives the command to start dispatching, a time counter is started in the dispatching unit and each accelerator-specific comparator register is loaded with the first value in its RAM. The time counter is synchronous with the system clock. Every clock cycle the comparators compare the current task timestamp to the time counter value; this is done in parallel for all accelerators. When a match occurs in a comparator, the output lines of the AMU are activated and the necessary information is transmitted to the appropriate accelerator. Several tasks can be dispatched concurrently.

6. EXPERIMENTS

6.1. Parallel MPEG-4 decoding

In Experiment 1, four NIOS II soft processors were instantiated on the FPGA in a configuration that is depicted in Fig-

Table 2. Duration of statically and dynamically scheduled MPEG-4 Simple Profile decoding efforts.

Experiment 1, fully static schedule	141 Mcycles
Experiment 1, quasi-static schedule	47 Mcycles
Experiment 2, fully static schedule	1.13 Mcycles
Experiment 2, quasi-static schedule	0.78 Mcycles

ure 5. One of the Nios II processors acted as the CPU and the other ones as dedicated accelerators. The processors were organized as a pipeline interconnected by 1kB on-chip buffer memories (numbers 1, 2, 3 in Figure 5). The last accelerator in the pipeline wrote the results directly to the frame buffer that was located on an external SRAM. The first accelerator (labeled ACC1) was responsible for dequantization of image data, whereas the second processor was responsible for 8x8 pixel IDCT (ACC2). The last accelerator (ACC3) was used for adding together predicted image data and residual data, followed by color value clipping.

ACC1, ACC3 and the CPU were NIOS II/s processors with a 512 byte instruction cache. The IDCT processor was a more powerful NIOS II/f processor with 2kB of data cache and 2kB of instruction cache. The IDCT processor was given more computational performance because the IDCT operation had the longest execution time and was the bottleneck in the pipeline. The processor performing dequantization took 2290 (I) or 2450 (P) clock cycles for processing 64 pixels, the IDCT processor took 4510 cycles and the clipping processor 1470 (I) or 2070 (P) cycles. Dequantization and clipping were mode-dependent: I- and P-blocks require different numbers of cycles. The task latencies were rather long, because the dedicated accelerators were performing the computations by running software code. The partial quasi-static flow-shop schedules had been computed for the system at system design time, and embedded into the AMU.

The system was used to decode MPEG-4 Simple Profile (SP) video. A clip of 45 frames of the well-known *foreman* sequence (176x144 resolution) was chosen for decoding. We measured the number of CPU clock cycles that was required to perform the computation of the *parallelized decoder part*. This part encompassed the dequantization, IDCT and image reconstruction actions. For a comparison, it was determined how many clock cycles the same part would take with a fully static run-time schedule; the results are shown in Table 2. The run-time scheduled decoding of this part takes 67% less time (in CPU clock cycles) to complete the same tasks, including the negligible communication between the CPU software and the AMU. The difference is due to the fact that in MPEG-4 macroblocks, the number of blocks that contain texture (coded) data can be anything between zero to six. Fully static scheduling assumes the worst case situation, *i.e.*, that all blocks contain residual data; in practice, this is rarely the case. Run-time scheduling can adapt to the changing number of blocks to decode. Depending on the number of coded blocks, the scheduling of tasks to decode one macroblock took 3 to 18 cycles from the AMU.

This experiment shows that the AMU can be used in practical applications and that it can help improving on the system throughput. The accelerators of this experiment were coarse grained due to their long execution times. In the following experiment, the AMU is used in conjunction with a soft processor system that emulates a functionally equivalent system that replaces processors with fine-grained ASIC accelerators.

6.2. Orchestration of fine-grained accelerators

In Experiment 2, the AMU was placed on the FPGA along with five NIOS II/s processors. One of the processors acted as a CPU and the four other processors emulated accelerators. The interconnection is depicted in Figure 6, results in Table 2.

To show the potential of the AMU for scheduling of fine-grained hardware accelerators, the system was adapted to model MPEG-4 video decoding. Except for the Nios II labeled ACC4, the other processors mimicked the functionality of real-world fine-grained hardware accelerators. ACC1 was responsible for motion compensation and takes 85 clock cycles [15]. ACC2 mimicked a AC/DC prediction accelerator that takes 136 clock cycles to complete [16]. Finally, the IDCT operation on ACC3 lasted 80 clock cycles [5]. As shown in Figure 6, the ACC4 input buffer (4) has two inputs. Depending on the actual block type, ACC4 either sums the results of the two data paths, or only selects and forwards one of them to buffer 5.

Since the computation speed of the soft processors was not enough for real operations with the desired latencies, the processors were programmed to communicate data according to the commands of the AMU, without modifying the data. When the AMU wake-up signal starts one of the ACCs 1...3, the ACC reads a data value from the input buffer, waits in an idle loop, and writes the data to the output buffer after a fixed time. The time from reading the input buffer to finishing the buffer write takes 85, 136 and 80 clock cycles, respectively.

ACC4 does perform some computations. When the AMU initializes ACC4, it reads the input data from the input buffer, and, depending on the execution mode, outputs the data produced by ACC1, ACC3, or the sum of the two values. ACC4 has a latency of 32 clock cycles for outputting the combined result and 24 cycles otherwise. The EPFS model [13] was used to compute quasi-static schedules that needed also to take care of shared buffer (3, 4 in Figure 6) access arbitration.

The accelerators were invoked in a pattern that was taken from real MPEG-4 SP video decoding. The software XViD video codec[1] was modified [7] so that the usage of the functions modeled by ACC1, ACC2 and ACC3 was recorded in a data file. This recorded data file was used in the FPGA system to provide a hardware accelerator invocation pattern.

[1] http://www.xvid.org/

The signals coming from the AMU were observed by Altera's SignalTap logic analyzer, and no jitter was detected in the AMU output. On a larger scale, the reliability of the AMU was verified by computing checksums of the data that had been orchestrated through the datapath shown in Figure 6. The experiments confirmed that the AMU is suitable for orchestrating the functionality of fine-grained accelerators.

7. CONCLUSION

The use of fine-granularity DSP hardware accelerators has been severely limited by traditional synchronization methods that cause considerable overheads. In this paper we present a scheduling co-processor that computes execution schedules with such small overhead that fine-grained hardware acceleration can be used in dynamic applications with a considerable efficiency benefit over static schedules or dynamic schedules computed by software. The presented accelerator management unit has been tested on a field-programmable gate array and the chip size overhead of the device has been found to be negligible.

Acknowledgments This work has been partially funded by the Nokia Foundation, Finnish Graduate School GETA, and the Tekes project ECUUS. The authors would also like to thank Panagiotis Athanasopoulos.

8. REFERENCES

[1] C. Lucarz, M. Mattavelli, J. Thomas-Kerr, and J. Janneck, "Reconfigurable Media Coding: a new specification model for multimedia coders," in *Proc. IEEE 2007 Workshop on Signal Processing Systems*, Shanghai, China, 2007, pp. 481–486.

[2] O. Silvén and K. Jyrkkä, "Observations on power-efficiency trends in mobile communication devices," in *Lecture Notes in Computer Science 3553*, Samos, Greece, 2005, pp. 142–151.

[3] T. Rintaluoma, O. Silvén, and J. Raekallio, "Interface overheads in embedded multimedia software," in *Lecture Notes in Computer Science 4017*, Samos, Greece, 2006, pp. 5–14.

[4] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, Marcel Dekker, NY, USA, 2000.

[5] R. Rambaldi, A. Ugazzoni, and R. Guerrieri, "A 35 μw 1.1 V gate array 8x8 IDCT processor for videotelephony," in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing*, 1998, vol. 5, pp. 2993–2996.

[6] N. Ling and N.-T. Wang, "A real-time video decoder for digital HDTV," *Journal of VLSI Signal Processing*, vol. 33, no. 3, pp. 295–306, March 2003.

[7] J. Boutellier, S. S. Bhattacharyya, and O. Silvén, "Low-overhead run-time scheduling for fine-grained acceleration of signal processing systems," in *Proc. IEEE 2007 Workshop on Signal Processing Systems*, Shanghai, China, 2007, pp. 457–462.

[8] W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J. Stankovic, G. Wallace, and C. Weems, "The Spring scheduling co-processor: A scheduling accelerator," *IEEE Transactions on VLSI Systems*, vol. 7, no. 1, pp. 38–48, 1999.

[9] J. Stärner, J. Adomat, J. Furunäs, and L. Lindh, "Real-time scheduling co-processor in hardware for single and multiprocessor systems," in *Proc. 8th Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy, 1996, pp. 164–168.

[10] G. Al-Kadi and A. S. Terechko, "A hardware task scheduler for embedded video processing," in *Proc. High Performance Embedded Architectures and Compilers*, Paphos, Cyprus, 2009, pp. 140–152.

[11] 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*, vol. 55, no. 12, pp. 1243–1255, 2004.

[12] S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Ellis Horwood Limited, Chichester, UK, 1982.

[13] J. Boutellier, V. Sadhanala, C. Lucarz, P. Brisk, and M. Mattavelli, "Scheduling of dataflow models within the reconfigurable video coding framework," in *Proc. IEEE Workshop on Signal Processing Systems*, DC, USA, 2008, pp. 182–187.

[14] Altera Corporation, "Nios II processor ref. handbook: www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf," 2008.

[15] T. Reinikka, "Transport triggered architecture based motion compensation of H.264 video," M.S. thesis, University of Oulu, Finland, 2009.

[16] C.-C. Lin, H.-C. Chang, J.-I. Guo, and K.-H. Chen, "Reconfigurable low power MPEG-4 texture decoder IP design," in *Proc. IEEE Asia-Pacific Conference on Circuits and Systems*, 2004, vol. 1, pp. 153–156.