Hierarchical task scheduler for interleaving subtasks on heterogeneous multiprocessor platforms

Abstract— Nowadays, the System-on-a-chip (SoC) has integrated more processors onto a single chip. Applications are also consisting of multiple tasks that are presented as different source code which can be partly executed concurrently. To efficiently use the multiprocessor platforms, system designers have to divide application tasks into sets of subtasks and map subtasks onto the parallel processors. However, the subtask-level parallelism inside a single task is often too limited to fully utilize all the parallel processors and results in many slacks on processors. To better use the processors, subtasks of multiple tasks will have to be executed in an interleaving fashion. This paper proposes design-time algorithms to interleave subtasks based on the separated schedules of tasks. This can be considered as a hierarchical scheduler to steer the code generation of very complex applications with many tasks. The scheduling experiments show that the execution time can be shortened by 20%–30%by interleaving two tasks and the differences between the solutions given by our scheduling algorithm and the optimal solutions are less than 6% for up to 20 subtasks.

I. Introduction

Nowadays, the merging of computers, consumer and communication in information technology (IT) leads to more and more components integrated onto one chip. This is enabled by the rapid evolution in sub-micron process technology. In an up to date heterogeneous platform, usually one or more programmable components, either general-purpose or DSP processor cores or ASIP's, the analog front end, on-chip memory, I/O and other ASIC's are all integrated into the same chip. However, existing design technologies fall behind these advances in processing technology. This is especially so when dealing with complex (possibly data-dominated) and very dynamic (partly non-deterministic) IT applications. A consistent system design technology that can cope with such characteristics and with the ever shortening time-to-market requirements is greatly needed. It should allow to map these dynamic applications cost-efficiently to the target platform while meeting all real-time and other constraints.

Recent papers[10, 15] have shown that the modern multimedia systems contain multiple concurrent tasks in the original specification. Unfortunately, due to the NP-hardness nature of most scheduling problems, it would take tremendous amount of time to schedule a single huge subtask frame constructed by merging multiple original tasks. This makes it useful to consider a hierarchical scheduling framework. That is, the subtasks in each single task are firstly scheduled separately; then a top-level task scheduler will generate a global schedule for a cluster of single tasks based on their individual schedules. The most obvious way to do that is to run all tasks sequentially. That is, the next task will start when the current task is completed. As a result of this assumption, if the next task is invoked before the completion of the current one, it has to wait for the current task. This assumption is only reasonable for uniprocessor platforms. Since the current heterogeneous multiprocessor platforms have constantly increasing numbers of processors, it is necessary to develop better techniques to keep the on-chip processors busy. In this paper, we propose optimal and fast heuristic algorithms to schedule task clusters based on interleaving subtasks. In other words, our algorithm will take into account the situation when multiple tasks are required to run concurrently and interleave their separated subtasks schedules to generate a new united schedule. The result of our interleaving algorithm will improve the overall performance compared with the sequential case and hence provide larger room for other techniques such as Dynamic Voltage Scheduling(DVS) and Dynamic Power Management (DPM) to further reduce the global energy consumption.

The rest of this paper is organized as below: Firstly, some related work is introduced in Section 2. Then a motivational example is given to illustrate the interleaving problem in Section 3. In Section 4 a problem formulation is given. Section 5 presents the interleaving algorithm. Some experimental results are shown in Section 6 . Finally, the paper is concluded in Section 7.

II. Related work

Task scheduling has been investigated overwhelmingly in the last decades. A good overview of early scheduling algorithms can be found in [12]. In this paper, the terminology *task scheduling* is used for both the ordering and the assignment.

Since more and more embedded systems are targeted at multiprocessor architectures, multiple processor scheduling plays an increasingly important role. Hoang *et al* [5]

try to maximize the throughput by balancing the computation load of the distributed processors. All the partition and scheduling decisions are made at compile time. This approach is limited to pure data flow applications. Yen *et al* [17, 16] try to combine the processor allocation and process scheduling into a gradient-search co-synthesis algorithm. It is a heuristic method and can only handle periodic tasks statically. Busá *et al* [2] try to interleave the coarse-grain operations and the fine-grain operations. But their approach works at the instruction level and only considers interleaving the I/O operations of coarse-grain operations. Recently, a novel dynamic parallel execution technique called Simultaneously Multi-Threading (SMT) has become popular [13]. This technique tries to increase the processor usage by issuing instructions from multiple threads at run-time. The latest Pentium4 processor of Intel has incorporated the Hyper-threading hardware to support SMT. But their technique relies on the special hardware support. In contrast, our technique makes use of design-time scheduling techniques and code generation. So no hardware change is required.

In the above work, performance is the only concern. For embedded systems, cost factors like energy must be taken into account as well. Gruian *et al* [4] have used constraint programming to minimize the energy consumption at system level but their method is purely static and no dynamic policy is applied to exploit more energy reduction.

Power consumption in a multiple processor context is treated in [9] by evenly distributing the workload. However, no manifest power and performance relation is used to steer the design-space exploration. In addition, they also assume a continuously scalable working voltage. In [8], for a multiprocessor and multiple link platform with a given task and communication mapping, a two phase scheduling method is proposed. The static scheduling is based on the slack time list scheduling, and a critical path analysis and task execution order refinement method is used to find the off-line voltage scheduling for a set of periodic real-time tasks. The run-time scheduling is similar to resource reclaiming and slack stealing, which can make use of the variation from the WCET and provide the best-effort service to aperiodic tasks. In [18], a EDF based multiprocessor scheduling and assignment heuristic is given, which is shown better than the normal EDF. After the scheduling, an ILP model is used to find the voltage scaling accurately or approximately by simply rounding the result from a LP solver and the result is claimed within 97% accuracy. The method can be used for both continuous and discrete voltages.

The scheduling technique in this paper is different from the above ones: it works in a hierarchical framework. That is, we assume that schedulers for each individual task , such as the one introduced in [14], have explored the most interesting scheduling tradeoff possibilities and have generated a set of Pareto-optimal schedules for each task.

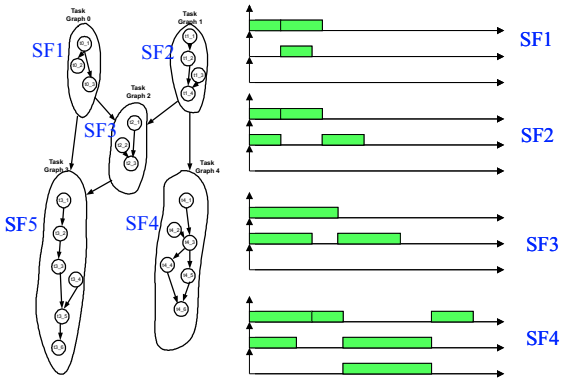

Fig. 1. Subtask frames and their schedules

Our scheduling algorithm will then generate schedules for a cluster of tasks based on those Pareto-optimal single task schedules. This hierarchical scheduling will ensure a performance which is very close to what can be guaranteed by running the conventional task schedulers on the flattened combination of all tasks in the cluster while reducing the computation complexity dramatically. Hence we can effectively deal with much larger task graphs, as shown in the results.

III. Motivational example

Before the design-time scheduling, a system should be modeled as a set of subtask frames. A *subtask frame* is equivalent to a *task* of the conventional models, and these two terms will be used interchangeably in this paper. However, rather than a black-box model as the conventional tasks, a subtask frame contains a set of subtasks and a control-data flow graph defining the control/data dependencies among subtasks. The subtask frame model can express the intra-task parallelism explicitly, which is crucial to obtain a high performance schedule with a low energy budget on multiprocessor platforms. Because the computation complexity is too huge, the single subtask frame scheduler has to work inside the frame, that is, this scheduler will explore most possible schedules for a subtask frame. Fig.1 illustrates the the concept of subtask frames as well as the schedules which will be explored by the design-time scheduler. On the left side, each large bubble represents a subtask frame; the smaller nodes inside subtask frames are subtasks. On the right side, a number of schedules on a three processor platform are shown; each block in the schedules represents a subtask. Note that although we only show one schedule for one subtask frame in the figure, one subtask frame may have many different schedules with different makespans and energy consumptions. In general, a larger time budget will allow the system running with a schedule with lower energy consumption.

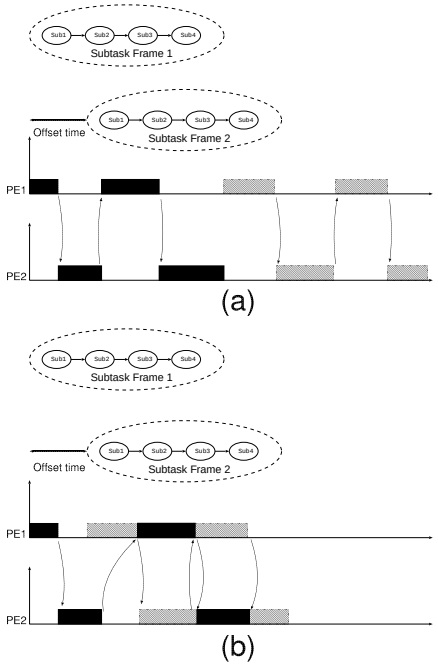

Fig. 2. Interleaving *vs.* non-interleaving

Suppose the next subtask frame can be started before the completion of the ongoing frame, the previous run-time scheduler will have to delay the start of the next subtask frame till the end of the ongoing subtask frame. This is illustrated in Fig.2(a). Since a single subtask frame often does not have sufficient parallelism to fully exploit all the processing resources and therefore has many slacks in its schedules, it is beneficial to interleave two subtask frames and thereby shorten the execution time of the two frames, as indicated in Fig. 2(b). The shorter execution time will then allow other subtask frames have larger time budgets and use schedules with lower energy consumptions.

IV. Problem formulation

A design-time schedule of a single subtask frame on a platform with m processors is defined as an allocation set $S = \{P_1 \cup P_2 \cup ... \cup P_m\}$, where $P_i = \{st_{i1}, st_{i2}, ...\}$ is the set of subtasks st_{ij} which have been mapped onto the ith processor. st_{ij} is a pair (b_{ij}, e_{ij}) which denotes a subtask starting at time $b_{ij}$and with an execution time of e_{ij}. Because control/data dependencies exist among subtasks inside a subtask frame, we have to preserve the precedence constraints inside the schedule such that the control/data dependencies of the subtask frame will not be violated. For a schedule S, the set of precedence constraints is defined as a finite set $Pre = \{(st_{ij}, st_{kl}), ...\}$, where the pair (st_{ij}, st_{kl}) represents the constraint that st_{ij}must be complete before st_{kl}can start.

The subtask frame interleaving problem for the set of schedules $\{S_1, S_2, ...S_n\}$is to assign a start time x_{ij} to the subtask st_{ij}such that the subtask frames can be completed as early as possible and all the precedence constraints between subtasks are satisfied:

$$\forall st_{ij} \in S_1 \cup S_2 \cup ... \cup S_n$$
$$Minimize[(x_{ij} + e_{ij})_{\max}]$$
$$s.t.$$
$$\forall (st_{ij}, st_{kl}) \in Pre_h, h = 1, 2, ...n$$
$$x_{ij} + e_{ij} \leq x_{kl}$$

V. Subtask interleaving technique

Scheduling tasks with non-uniform execution times on multiple processors is well-known for its intractability[3]. In fact, Hoogeveen *et al*[6] have proved that even for three processors, scheduling tasks with fixed processor allocations is a NP-hard problem. Still, for not too large tasks, an exact algorithm can be applied. We have developed a branching-and-bound algorithm for the interleaving problem. This algorithm starts with all subtasks unscheduled. After that, it finds a list of subtasks without precedence constraints and branches into every precedence-free subtask, i.e. it schedules one precedence-free subtask from that list and updates the stauts. Then it invokes itself with the updated status in a recursive way. This recursion continues until all subtasks are scheduled or the partial schedule has a makespan larger than the upper bound. When the algorithm reaches a valid schedule, it stores the makespan as the upper bound. An outline of this branching-and-bound algorithm is given in Algo.1.

Algorithm 1 Branching-and-bound algorithm for interleaving

```
1: BnB()
2: INPUT: status; upper_bound
3: OUTPUT: makespan
4: if makespan of status > upper_bound then
5:    return makespan of status
6: end if
7: if all subtasks are scheduled then
8:    print status
9:    return makespan of status
10: else
11:    schedulable_subtasks ← precedence_free subtasks
12:    for all subtask i in schedulable_subtasks do
13:       new_status ← status
14:       schedule subtask i and update new_status
15:       makespan ← BnB(new_status, upper_bound)
16:       if makespan < upper_bound then
17:          upper_bound ← makespan
18:       end if
19:    end for
20:    return upper_bound
21: end if
```

For larger subtask frames (with more than 20 subtasks

in our experiments) that are too expensive to handle by the straightforwd branching-and bound approach, a divide-and-conquer heuristic strategy can be applied to process partitions separately. That is, the frames are splitted into subframes such that the number of subtasks inside each individual subframe can be handled by the branching-and-bound approach. This comes at the cost of sub-optimality (see Section VI)

It is still interesting though to have fast algorithms that can handle more subtasks. Therefore an effective heuristic algorithm has been developed to interleave multiple subtask frames. This heuristic must be fast to construct a valid schedule so that the designer can evaluate multiple schedules which have been provided by preceding individual task scheduler.

We propose an interleaving heuristic based on the list scheduling algorithm[7]. The basic idea is to keep a list for each processor and add all the subtasks allocated to the corresponding list. Then for each processor, this algorithm will scan the list from left to right. Once a scanned subtask has all of its predecessors completed, it will be added to the ready list and scheduled onto the current processor. We have also adapted the subtasks' priorities in the ready list such that the greedy behaviour of the list scheduling is compensated.

To run the interleaving heuristic, we have created a time tracer for each processor. We also maintain a list of Precedence Constraints (PCs) and an Earliest Start Time (EST) for each subtask. PCs record the number of the subtask's unscheduled predecessors and EST records the latest completion time of the subtask's predecessors. A subtask cannot start unless its PCs are satisfied and it cannot start earlier than the time specified by its EST. During the initialization, the algorithm will set the time tracers to zero, create the PCs according to the original individual schedules of subtask frames and set all EST records to be zero. Then it will start from the first frame (the order of frames is arbitrary). For each processor, The algorithm will scan if any subtask is ready to start. If a subtask does not have unscheduled predecessors and the timer of its allocated processor has a value not less than its EST, then it will be added to the ready list. Then it will be scheduled to start at the time specified by the later one of the time tracer and its EST. The successors of this scheduled subtask will update their EST to the time not earlier than the completion of this scheduled subtask. Also, the time tracer of the processor where the subtask is scheduled will be updated to the earliest EST of all subtasks to be scheduled of this processor. This update can ensure that the subtask with the earliest EST will be scheduled by the next scanning on this processor. The scanning and updating procedure will be repeated until all subtasks are scheduled.

Although experiments have shown that the above heuristic is very effective in most cases, its greedy strategy will make it suboptimal when dealing with subtasks with a very long chain of precedence constraints. That is, if many subtasks are ready to start, the current algorithm does not favor the ones that have more successors. Therefore, a long critical path will lead to a very sequential execution. To tackle this problem, we have incorporated look-ahead mechanism into the current algorithm, i.e. we modified the algorithm such that it gives higher priority to the subtask with more successors. The entire heuristic algorithm is presented in Algo.2. Once the interleaved schedule is generated, we can use it to steer the code generation by using the code merging technique presented in [11]. The resulting code can then be executed on the multiprocessor platform.

Algorithm 2 Interleaving heuristic

```
1: INPUT: st1, st2, ...; pre1, pre2, ....
2: OUTPUT: interleaved schedule of subtasks
3: time_tracer ← 0
4: while unsched_subtasks > 0 do
5:   for all processor i do
6:     for all subtask frame j do
7:       if frame j has unsched sbutask on prcessor i then
8:         if the first subtask of frame j is schedulable then
9:           add the subtask to the ready list on the processor
10:        end if
11:      end if
12:    end for
13:    for all subtasks on the ready list do
14:      priority ← EST + number of successors
15:    end for
16:    EST ← EST of the highest priority subtask
17:    if time_tracer < EST then
18:      time_tracer ← EST
19:    end if
20:    schedule the highest priority subtask at time_tracer
21:    update the schedule
22:    inform this subtask's start time to its successors
23:    unsched_subtasks ← unsched_subtasks − 1
24:  end for
25: end while
```

VI. Experimental results

To evaluate the effectiveness of the interleaving algorithm, we have implemented it in a prototype scheduler with the Python language and firstly conducted the experiments with a set of schedules generated randomly. We then use the Visual Texture Coding (VTC) decoder of the MPEG-4 as our real-life application for the experiment. More experiments have been carried out with the Inverse Discrete Cosine Transform(IDCT) and the Finite Impulse Response (FIR) filter code extracted from the Trimaran benchmarks. We have generated the interleaved source code for IDCT and FIR and simulated the interleaved execution on a heterogeneous multiprocessor simulator.

A. Evaluation of optimality for heuristic interleaving

To evaluate the optimalities of the scheduling algorithms, we have tested the branching-and-bound algo-

		5×2	7×2	10×2
Sequential Makespan		100	100	100
H	Makespan	65.2	66.3	69.1
	Exec. time (ms)	4.0	9.2	19.8
BB	Makespan	62.6	63.3	65.9
	Exec. time (ms)	$\approx 10^4$	$\approx 10^4$	$\approx 10^7$
		15×2	17×2	20×2
Sequential Makespan		100	100	100
H	Makespan	72.5	75.2	74.0
	Exec. time (ms)	9.2	10.1	13.1
BB+	Makespan	71.6	78.2	76.1
DnC	Exec. time (ms)	$\approx 10^5$	$\approx 10^5$	$\approx 10^7$

TABLE I
Optimality comparison results

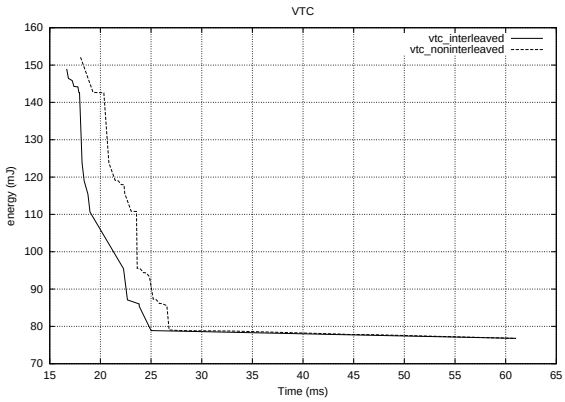

Fig. 3. VTC: interleaved vs. non-interleaved execution

rithm for two subtask frames. Each subtask frame consists of 5 subtasks, 7 subtasks and 10 subtasks, respectively. For larger frames that are too expensive to handle by the straightforwd branching-and bound approach, we have applied the divide-and-conquer heuristic. Using the branching-and-bound combined with the divide-and-conquer strategy, we have conducted experiemnts for two frames, each of which contains 15, 17 and 20 subtasks, respectively. We have also tested the fast heuristic algorithm for all cases. All the scheduling experiments assume a platform with four heterogeneous processors. For each case, we have repeated the experiment for 100 times and report the average numbers. Tab.I shows the results. H, BB and BB+DnC represent the heuristic algorithm, branching-and-bound algorithm and the branching-and-bound algorithm with the divide-and-conquer strategy. For each algorithm, we report the average makespan and the execution time. Note that we have normalized the makespans. All of the experiments are measured on a Linux PC running at 1.7GHz. It is observed that when dealing with small tasks, branching-and-bound is always better than the heuristic algorithm. This is because branching-and-bound is an exact fast algorithm while heuristic is merely an approximation of the optimal result. However, for large cases, our heuristic can give better results than the branching-and-bound with the divide-and-conquer strategy because that also becomes sub-optimal. Therefore, the system-designer should consider both branching-and-bound with the divide-and-conquer and our interleaving heuristic when scheduling very large subtask frames. Since both are performed at design-time, the time overhead of applying both heuristics is acceptable.

B. Comparisons with sequential cases

We have applied the interleaving algorithm on the schedules of the VTC decoder. We have analyzed and made the design-time scheduling of this decoder(see [10] for details). In this paper, we assume that two such decoders will run concurrently for different objects. Fig.3 shows the interleaving results of two VTC decoders running concurrently at different decoding speeds. This figure clearly indicates that the interleaved execution times are on average 20% shorter than the non-interleaved execution times in the time range between 20 and 25 ms, which is the interval of the most frequently imposed speed requirements. We have also conducted more experiments with benchmarks based on IDCT and FIR. Firstly, a 8×8IDCT procedure for a QCIF image was modelled as a subtask and we have considered a subtask frame with four such IDCT subtasks. Then, we modelled a 64-tap FIR filter as a subtask and constructed a subtask frame with eight FIR subtasks. After that, we scheduled each of the two benchmarks on a four-processor platform. Finally, we have conducted interleaving for the following two scenarios: two instances of IDCT frames are scheduled to run in the interleaved way; and one instance of IDCT frame and one instance of FIR frame are scheduled to run in the interleaved way. These system-level interleaving results are shown in the Tab.II.

C. Code generation and simulation of benchmarks

In addition to the system-level scheduling, we have generated the interleaved source code for the IDCT and FIR benchmarks. The generated source code is then compiled and simulated on the multiprocessor compiler and cycle-accurate simulator environment called CRISP[1]. In our experiments, CRISP is configured to emulate a four heterogeneous VLIW processors architecture where the processors are synchronized on a shared memory. Tab.II summarizes the average performance improvements against the non-interleaved execution of those subtask frames.

		2 IDCT
System-level estimation	sequential($10^3 cycles$)	1900
	interleaved($10^3 cycles$)	1320
	improvement	30.5%
Cycle-accurate simulation	sequential($10^3 cycles$)	1876
	interleaved($10^3 cycles$)	1520
	improvement	19.0%
		IDCT+3FIR
System-level estimation	sequential($10^3 cycles$)	1700
	interleaved($10^3 cycles$)	950
	improvement	44.1%
Cycle-accurate simulation	sequential($10^3 cycles$)	1696
	interleaved($10^3 cycles$)	946
	improvement	44.2%

TABLE II
PERFORMANCE IMPROVEMENTS COMPARISON

VII. Conclusions

Task scheduling technique becomes increasingly important to design the real-time embedded systems with the stringent energy budgets. This paper presents a fast scheduling technique to minimize the processor slacks for heterogeneous multiprocessor platforms. This technique can interleave multiple frames of subtasks. We have implemented the technique in a prototype scheduler and conducted experiments for subtasks generated randomly, the execution time can be shortened by a factor up to 37%. The experiments on a MPEG-4 still image decoder show an average execution time improvement of 20% in the most used decoding speed range. More experiments on the benchmarks have also proved the effectiveness of our heuristic algorithm to steer the code generation for heterogeneous multiprocessor platforms.

References

[1] Francisco Barat, Murali Jayapala, Tom Vander Aa, Rudy Lauwereins, Geert Deconinck, and Henk Corporaal. Low power coarse-grained reconfigurable instruction set processor. In *3th International Conference on Field Programmable Logic and Applications*, September 2003.

[2] N.G. Busá, A. van der Werf, and M. Bekooij. Scheduling coarse-grain operations for VLIW processors. In *Proceedings of ISSS 2000*, pages 47–53, 2000.

[3] Michael R. Garey and David S. Johnson. *COMPUTERS AND INTRACTABILITY: A GUIDE TO THE THEORY OF NP COMPLETENESS*. W.H. Freeman and Company, New York, 1979.

[4] F. Gruian and Kuchcinski. Low-energy directed architecture selection and task scheduling. In *Proceedings of 25th EUROMICRO Conference*, volume 1, pages 296–302, 1999.

[5] P. Hoang and J. Rabaey. Scheduling of dsp programs onto multiprocessors for maximum throughput. *IEEE Transactions on Signal Processing*, 41(6):2225–2235, June 1993.

[6] J. A. Hoogeveen, S. L. Van De Velde, and B. Veltman. Complexity of scheduling multiprocessor tasks with prespecified processor allocations, 1992.

[7] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9:841–848, 1961.

[8] J. Luo and N. Jha. Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems. In *Proceedings of the 7th ASP-DAC*, pages 719–726, 2002.

[9] J. Luo and N.K. Jha. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In *International Conference on Computer Aided Design 2000*, pages 357–364, San Jose, USA, November 2000.

[10] Zhe Ma, Chun Wong, Francky Catthoor, et al. Task concurrency analysis and exploration of visual texture decoder on a heterogeneous platform. In *Proceedings of 2003 IEEE Workshop on Signal Processing Systems(SiPS 2003)*, Seoul, Korea, August 2003.

[11] Francesco Poletti, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose Mendias. An integrated hardware software approach for run-time scratchpad-management. In *DAC*, June 2004.

[12] K. Ramamritham and J.A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67, January 1994.

[13] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, 1995.

[14] C. Wong, P. Marchal, P. Yang, A. Prayati, et al. Task concurrency management methodology to schedule the MPEG-4 IM1 player on a highly parallel processor platform. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, pages 170–195, Copenhagen, April 2001.

[15] P. Yang, P. Marchal, C. Wong, et al. Managing dynamic concurrent tasks in embedded real-time multimedia systems (keynote speech). In *Proceedings of ISSS 2002, 15th International Symposium on System Synthesis*, Kyoto, Japan, October 2002.

[16] T.Y. Yen and W. Wolf. Communication synthesis for distributed embedded systems. In *International Conference on Computer-Aided Design*, pages 288–294, 1995.

[17] T.Y. Yen and W. Wolf. Sensitivity-driven co-synthesis of distributed embedded systems. In *Proceedings of the Eighth International Symposium on System Synthesis*, pages 4–9, September 1995.

[18] Y. Zhang, X. S. Hu, and D. Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proceedings of DAC 2002, 39th Design Automation Conference*, 2002.