

Run-time Task Overlapping on Multiprocessor Platforms

Zhe Ma, Daniele P. Scarpazza, Francky Catthoor*

IMEC, Kapeldreef 75, Leuven 3000, Belgium
{mazhe}@imec.be

Abstract

Today's embedded applications often consist of multiple concurrent tasks. These tasks are decomposed into sub-tasks which are in turn assigned and scheduled on multiple different processors to achieve the optimal performance/energy combination. Previous work introduced systematic approaches to make performance-energy trade-offs explorations for each individual task and used the exploration results at run-time to fulfill system-level constraints. However, they did not exploit the fact that the concurrent tasks can be executed in an overlapped fashion. In this paper, we propose a simple yet powerful on-line technique that performs task overlapping by run-time sub-task re-scheduling. By doing so, a multiprocessor system with concurrent tasks can achieve better performance without extra energy consumption. We have applied our algorithm to a set of randomly-generated task graphs, obtaining encouraging improvements over non-overlapped task, and also having less overall energy consumption than a previous DVS method for real-time tasks. Then, we have demonstrated the algorithm on real-life video- and image-processing applications implemented on a dual-processor TI TMS320C6202 board: We have achieved a reduction of 22-29% in the application execution time, while the impact of run-time scheduling overhead proved to be negligible (1.55%).

1 Introduction

The ever increasing functionalities of embedded systems has led to a large number of complex applications that are composed of multiple concurrent real-time tasks. A recent paper [8] has shown that modern multimedia systems contain multiple concurrent tasks in their original specification (usually, in terms of C language functions). These tasks can then be divided into subtasks by system designers by further

analysis (e.g. as in [9]).

On the other hand, the rapid evolution in sub-micron process technologies has enabled the manufacturing of multiprocessor systems-on-chip (MPSoC) with a large number of different processing cores. The current generation of MPSoC platforms usually have one or more programmable components, either general-purpose or DSP processor cores or ASIPs, all on the same chip. Such MPSoC platforms are often referred to as the *heterogeneous* multiprocessor platforms. Existing design technologies fall behind such advances in process technology, and fail to exploit them efficiently, especially in systems with multiple concurrent tasks. Moreover, when deployed as a component in a system such as a multimedia mobile phone, the MPSoC must provide the performance-energy trade-off capability to let the system-level power management choose the optimal speed/energy for it. For instance, when a mobile phone is working near a base station, less energy is needed for the RF transmission while a large energy budget is allowed for running the multimedia applications; If the phone is far away from a base station, much more energy is allocated to the RF transmission and hence a smaller energy budget is allowed for running the applications.

The dynamic voltage scaling (DVS) technique was proposed to make an energy-aware scheduling. Based on DVS, several design-time and run-time scheduling algorithms have been proposed [6, 2, 7, 1]. None of these exploit however the possible task/subtask concurrency in a systematic way. Moreover, they did not consider the trade-offs that are available between performance and energy offered by the heterogeneous multiprocessor.

Wong et al. [11], investigated the multi-objective scheduling problem for MPSoCs at design time, and provided designers with a fast technique to explore optimal performance/energy trade-offs schedules for a single task composed of multiple subtasks. Yang et al. [12] then presented a run-time scheduling algorithm that can apply Wong's design-time exploration results. Fig. 1 illustrates their work, which is partly used as the basis of our work.

As shown in Fig. 1(a), the exploration at design-time can

*Also professor of K.U.Leuven-ESAT

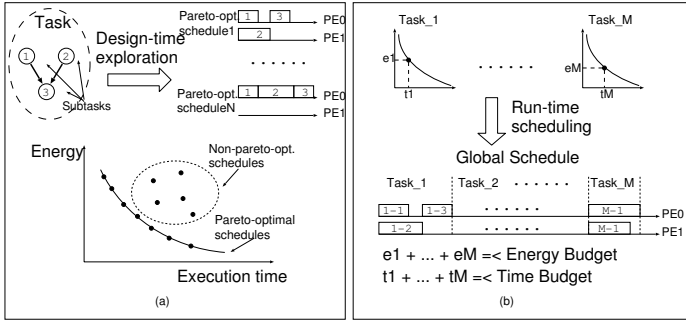


Figure 1. (a)Design-time exploration. (b)Run-time scheduling

generate most of the optimal trade-offs schedules for every single task. Those optimal schedules form the Pareto curve when projected to the energy-versus-time plane. At run-time, the on-line scheduler can then select the appropriate schedules for those tasks that are invoked in parallel. The global schedule generated at run-time can guarantee that both the timing constraint and the energy consumption constraint are met.

Unfortunately, due to the combination of different energy efficiencies of heterogeneous processing elements of MPSoCs and the insufficient subtask-level parallelism, many slacks are present in the Pareto-optimal schedules of a single task. Because Yang’s run-time scheduler can only handle an execution without overlapping between tasks, the slacks would remain in the final global schedule (as illustrated in the global schedule in Fig. 1(b), between subtask 1-1 and 1-3, and also between subtask 1-2 and the start of Task2). Voltage-scaling-based techniques such as [1, 2] can only reclaim the slack between 1-1 and 1-3, but fail to use the slack after 1-2, because of the subtask-level dependencies.

This makes it useful to consider a run-time technique that can efficiently reclaim those slacks and hence better utilize the processing elements of the MPSoCs. In this paper, we propose an on-line algorithm to re-schedule the concurrent tasks based on subtask interleaving. Our algorithm takes into account the case when multiple tasks are required to run concurrently, and interleaves their individual design-time schedules to generate a new global schedule. Our interleaving algorithm proved to provide significant improvements over the previous run-time scheduler while still meet the energy constraints imposed. Moreover, we have also demonstrated that it is feasible to integrate this on-line technique onto an existing RTOS on a real-life multiprocessor platform.

The rest of this paper is organized as follows: Section 2

	st0	st1	st2	st3	st4	st5	st6
PE0 (mJ/ms)	3/3	2/2	2/2	6/6	2/2	6/6	2/2
PE1 (mJ/ms)	8/1	8/1	8/1	16/2	8/1	16/2	8/1

Table 1. Energy consumption & execution time of subtasks on the target PEs

provides a small scheduling problem to illustrate the basic idea of interleaving. Section 3 presents the proposed interleaving algorithm. Experimental results are described in Section 4. Conclusions are drawn in Section 5.

2 Motivational example

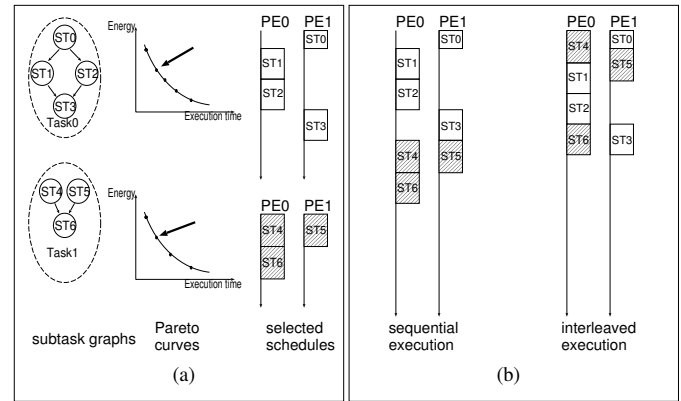
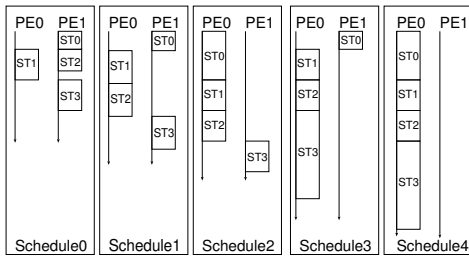


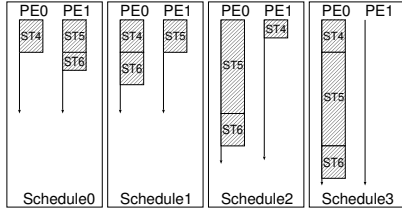
Figure 2. (a) two tasks, their Pareto curves and two specific schedules. (b) comparison between the sequential execution and the interleaved execution

We have employed a event-triggered concurrent task model: a set of concurrent tasks are triggered by a single event, and must be finished within a given period under a certain energy consumption budget. Fig. 2(a) illustrates the subtask graphs of two example tasks, task 0 and task 1. Note that it is unknown at design-time that task 0 and task 1 are invoked simultaneously. The concurrent running situation, the deadline and the energy consumption constraint are dynamic information that is only available at run-time. We consider the problem of scheduling these subtasks on a platform with two heterogeneous processing elements (PE0 and PE1). The energy and execution times of subtasks are given in Tab. 1

A design-time performance-energy trade-off exploration reveals that task 0 has five Pareto-optimal schedules on the



(a)



(b)

Figure 3. Pareto-optimal schedules of task 0 (a) and task 1 (b)

	task 0 (mJ/ms)	task 1 (mJ/ms)
schedule 0	34/5	26/3
schedule 1	28/7	20/4
schedule 2	23/9	16/8
schedule 3	18/11	10/10
schedule 4	13/13	

Table 2. Pareto-optimal schedules on the energy/performance trade-off space

target platform and task 1 has four Pareto-optimal schedules. A schedule is Pareto-optimal if it is the fastest schedule among all schedules that consume the same amount of energy and it consumes the least amount of energy among all schedules that have the same speed. All Pareto-optimal schedules are illustrated in Fig. 3. The energy and timing information of each Pareto-optimal schedule is listed in Tab. 2.

Now suppose that task 0 and task 1 have to be finished within 12 ms and the total energy consumption cannot exceed 48 mJ. The run-time scheduler [12] can choose two schedules from the pools of Pareto-optimal schedules of the two tasks such that they can be finished before the deadline and the total energy consumption is under the constraint. In this specific case, the selected Pareto-optimal schedules are the schedule 1 of task 0 and the schedule 1 of task 1. These two schedules are illustrated in Fig. 2(a).

It is interesting to observe that both Pareto-optimal schedules have significant slacks. In fact, individual task's schedules often exhibit idle slots. This is due to (a) limited subtask parallelism within a single frame, which leads to the insufficient utilization of the processors, and (b) different execution times and energy consumptions caused by running the same subtask on heterogeneous processors: consequently, power-optimizing design-time scheduling strategies tend to under-utilizes the parallelism in favor of the energy efficiency.

Because a traditional re-scheduling of all subtasks would be extremely time-consuming, the previous run-time scheduler can only run over all Pareto-optimal scheduler sequentially and thereby ignore these intra-task slacks. As a result, in the global schedule, these slacks are still present. In contrast, we propose a novel technique named *interleaving* that can efficiently reclaim those slacks. A comparison of the sequential schedule versus our interleaved schedule for the motivation example is shown in Fig.2(b). The interleaved global schedule increases the performance by 27% without using extra energy. The interleaving approach is different from conventional flattening-and-rescheduling (FAR) re-scheduling approaches such as [10] in the fact that the mapping of subtasks to processors and the subtask order are preserved. This means that each subtask is mapped to the same processor as in its original schedule, and that if subtask s_i is scheduled before s_j in the design-time schedule, the same condition holds in the run-time schedule.

Interleaving only operates rigid translations of subtasks along the time axis of the processor where it was allocated at design-time, in order fill in idle slots. It is therefore evident that interleaving speeds up concurrent tasks without increasing the energy consumption with respect to sequential execution. As a consequence, there is no need to verify that the energy constraints are met after interleaving, as long as the sum of the energy consumptions of the individual tasks does not exceed the energy budget.

This hierarchical scheduling adopted by our approach significantly reduces in the exploration space, compared to the conventional FAR schemes and thereby to significantly lower run-time overheads. Actually, it is almost infeasible for FAR algorithms to find a global schedule to satisfy the energy budget within a reasonable time.

3 On-line sub-task interleaving

The problem of scheduling (sub)tasks with non-uniform execution times on multiple processors is well-known for its intractability [4]. In fact, it was proved in [5] that even for three processors, task scheduling with fixed processor allocation is a NP-hard problem. Therefore, we chose to develop a heuristic on-line interleaving algorithm.

Our algorithm basically consists of two steps. The first

step produces an initial interleaved global schedule using the greedy strategy. Starting from this initial global schedule, the second step then does a local search based on a unique neighborhood mapping function.

The greedy search of the first step includes three stages. At the first stage, it creates a subtask set for each processor, containing all the subtasks that have been allocated to that processor in their design-time schedules. Then, at the second stage, each subtask is assigned a priority according to a formula given below, and subtasks are re-ordered according to their descending priority. Finally, the algorithm calculates the earliest starting time (ASAP) for each subtask, preserving the dependency constraints.

Subtask priorities are assigned to each subtask, based on the Earliest Start Time (EST) of the subtask (i.e., the time at which all of the subtask's predecessors are completed in the original subtask frame).

The greedy search algorithm is presented in Algorithm 1. Since the EST of a subtask is known when the design-time schedule is available, at run-time we only need to check for each scanned subtask that if its predecessors are all finished. Consequently, the worst-case time complexity of the proposed algorithm is $O(n^2)$, where n is the total number of subtasks.

Algorithm 1 Greedy search algorithm

```

1: INPUT:  $T_1, T_2, \dots; S_1, S_2, \dots;$ 
2: OUTPUT:  $S_{\text{interleaved}}$ 
3:  $t \leftarrow 0$ 
4: while  $\text{unsched\_subtasks} > 0$  do
5:   for all processor  $i$  do
6:     for all subtask frame  $j$  do
7:       if frame  $j$  has unsched subtask on processor  $i$  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  $s$  on the ready list do
14:      $\text{priority}_s \leftarrow \text{EST}_s$ 
15:   end for
16:    $\text{EST} \leftarrow \text{EST}$  of the highest priority subtask
17:   if  $t < \text{EST}$  then
18:      $t \leftarrow \text{EST}$ 
19:   end if
20:   schedule the highest priority subtask at  $t$ 
21:   update the schedule
22:   inform this subtask's start time to its successors
23:    $\text{unsched\_subtasks} \leftarrow \text{unsched\_subtasks} - 1$ 
24: end for
25: end while

```

After building up the initial global schedule, the interleaving algorithm starts the second step to conduct a local search. The local search can alleviate the greedy behavior of the first step and thereby optimize the final global schedule.

The most crucial part of the local search is the definition of the neighborhood mapping function. That is, how

to define the neighborhood space that we will search for a better solution. We propose the following consecutive-segment-exchanging scheme in our neighborhood mapping function. In a global schedule, two subtasks are consecutive if they are on the same processor and one starts immediately after the other. For a sequence of subtasks, if every pair of subtasks that are next to each other are consecutive, then this sequence of subtasks is a consecutive-segment. It is obvious that if we exchange the position of two subtasks in a consecutive-segment and shift all the subtasks to preserve the subtask dependencies, we can find a new tentative schedule. This tentative schedule, however, could be invalid, because not all pairs of subtasks of a consecutive segment can switch their positions. An exchange that can cause the conflicts of subtask dependencies is called an invalid exchange. Our consecutive-segment-exchange scheme firstly locates all consecutive segments of the global schedule. Then it exchanges every pair of subtasks within each consecutive segment and checks if it is a valid exchange. Finally, it returns all valid exchanges which consequently make up the neighborhood space of the initial global schedule.

The local search first generates the neighborhood space of the initial global schedule and uses this neighborhood as the initial search space. It then enlarges the search space by appending the neighborhood space of each element in the initial search space. This enlarging is repeated until the size of the search space reaches a pre-determined limit. An interesting observation regarding this limit is that if a better solution could be found by the local search, it would often be found within the first 500 explored solutions (extensive experiments have shown that this chance is higher than 80%). Therefore, we chose 500 as the limit in the implementation of our algorithm. The entire local search algorithm is given in Algorithm 2.

Algorithm 2 Local search algorithm

```

1: INPUT:  $S_{\text{interleaved}}$ 
2: OUTPUT:  $S_{\text{initial}}$ 
3:  $S_{\text{interleaved}} \leftarrow S_{\text{initial}}$ 
4:  $\text{search\_space} \leftarrow \{S_{\text{initial}}\}$ 
5:  $\text{space\_size} \leftarrow 1$ 
6: while  $|\text{search\_space}| > 0$  do
7:    $S \leftarrow$  1st schedule of  $\text{search\_space}$ 
8:   if  $S < S_{\text{interleaved}}$  then
9:      $S_{\text{interleaved}} \leftarrow S$ 
10:  end if
11:   $\text{history} \leftarrow \text{history} \cup \{S\}$ 
12:  if  $\text{space\_size} < \text{MAX\_SIZE}$  then
13:     $\text{neighbor} \leftarrow \text{NeighbourhoodMapping}(S)$ 
14:     $\text{neighbor} \leftarrow \text{neighbor} - (\text{neighbor} \cap \text{search\_space})$ 
15:     $\text{neighbor} \leftarrow \text{neighbor} - (\text{neighbor} \cap \text{history})$ 
16:     $\text{space\_size} \leftarrow \text{space\_size} + |\text{neighbor}|$ 
17:     $\text{search\_space} \leftarrow \text{search\_space} \cup \text{neighbor}$ 
18:  end if
19: end while

```

4 Experimental results

To evaluate the effectiveness of the proposed interleaving algorithm, we have performed two sets of experiments. In the first set of experiments, the interleaving algorithm has been compared with the non-overlapped scheduling technique; In the second set of experiments, the interleaving algorithm, together with the design-time/run-time scheduling framework, has been compared with a recent EDF-based DVS technique to evaluate our framework’s effectiveness in terms of reducing energy consumption. Both sets of experiments are conducted based on a set of randomly-generated subtask graphs produced by TGFF [3]. Finally, we demonstrate the applicability of the algorithm in a real-life system: this algorithm has been implemented on a dual-processor board, and applied over a combination of instances of an image-processing benchmark and a H.263 video decoder.

4.1 Experiments with random tasks

In our first set of experiments we have generated 50 random subtask graphs, each composed by 20 subtasks. Each subtask graph has been treated as a single task. Each task has been scheduled on both a four-processor and a six-processor heterogeneous platform, using the design-time scheduler described in [11], and thus received sets of complete Pareto-optimal schedules (remember that a task normally has multiple Pareto optimal schedules, in the sense that each schedule has the lowest energy consumption for its specific speed). Then, we have applied the proposed interleaving algorithm on every set of cardinality 2, 3 and 4 of the above design-time schedules. That is, we simulated three distinct cases in which 2, 3, and 4 tasks are invoked and run concurrently from time 0. We have repeated the above experiments on a four-processor and on a six-processor platform, in order to explore the impact of the increased number of processors. The speed-up has been calculated by comparing the makespans¹ of interleaved schedules and those of running the previous run-time scheduler [12]. Results are shown in Tab. 3.

EDF is a well-known technique for real-time task scheduling. Recently, [7] have combined the EDF technique with the Dynamic Voltage Scheduling (DVS) approach for low-power real-time systems. We have extended their approach to exploit the design-time scheduling results. In our implementation of the EDF scheduler, all tasks are first ordered according to the EDF strategy (since our model assumes that all tasks have the same deadline, a task with the longest makespan is regarded as the one with the earliest deadline). Then starting from the first

¹the *makespan* of a schedule is formally defined as the difference between the start time of the first scheduled subtask and the end time of the last ending subtask.

	2 tasks	3 tasks	4 tasks
4 processors	14.5%	17.2%	18.0%
6 processors	19.2%	23.6%	25.1%

Table 3. Average speed-ups achieved by our interleaving technique on randomly-generated tasks.

	DT [0, 1]	4-processor	6-processor
2 tasks	0.1	3%	5%
	0.5	25%	24%
	0.9	41%	40%
3 tasks	0.1	6%	6%
	0.5	27%	28%
	0.9	40%	41%
4 tasks	0.1	10%	9%
	0.5	31%	30%
	0.9	38%	39%

Table 4. Average energy reduction achieved on randomly-generated tasks

task, we select a Pareto-optimal schedule from each one’s design-time scheduling result such that the selected Pareto-optimal schedule has the lowest energy consumption while all the tasks behind the current one can still meet the deadline. We have performed extensive experiments on random task graphs with different DTs (deadline tightness’s, ranging from 0 to 1). A tightness of 0 means that a deadline equals to the sum of all tasks’ shortest makespans and therefore is the tightest; while 1 means a deadline that equals to the sum of the longest makespans. The comparison given in Tab. 4 shows that our interleaved schedules have consumed less energy than the schedules from the EDF-based DVS approach.

4.2 Experiments with multimedia applications on a dual-processor board

To demonstrate the feasibility of integrating the interleaving technique in existing RTOS and also to measure the performance improvement achieved by interleaving (as well as its run-time overhead on real-life applications), we have performed experiments on a multi-processor board. The platform we have employed is equipped with two Texas Instruments TI TMS320C6202 DSP processors, running the Virtuoso Real-Time Operating System (RTOS). The appli-

cations, our interleaving scheduler code, and additional support code have been written in the C language and compiled with Texas Instruments TMS320C6x C/C++ compiler, version 4.10.

The actual implementation of the proposed methodology first requires two pieces of code to be written: (a) the *scheduling function*: the code that takes as inputs the individual task schedules and calculates one global, interleaved schedule; (b) the *interleaving function*: the code that takes as an input one of the above global schedules and “plays” it on each processor, starting the indicated subtasks at the appropriate time. In addition, in the application code, the third piece of code must be instrumented: (c) the *subtask dispatcher*: a function that takes a subtask identifier as an input and executes the corresponding subtask in the application.

The scheduling and the interleaving functions, and their associated data structures and data-types, are general and application-independent. In order to enforce separation between user and system code, and to facilitate the reuse of the above functions, we gathered them in an object library, against which the application code must be linked.

In our experimental setup, at system startup, the scheduling function is invoked on the first processor (while the second remains idle), and its output schedules are generated. Then, each processor launches its own interleaving function, with the schedule for that processor as an argument. Each interleaving function starts then the application subtasks, according to the schedule, iteratively calling the *subtask dispatcher* provided by the application. The overall structure is depicted in Fig. 4.

We have employed two applications as benchmarks: the image-processing benchmark called ThreeSteps, and the H.263 video decoder.

ThreeSteps (TS) is an image-processing application which performs the following operations: it quantizes an image, applies to it a Sobel filter, and calculates the brightness histogram. The algorithm implementations have been taken from the Texas Instruments IMGLIB library. The benchmark consisted in applying TS on a sample 512×512, 8-bit gray-level image (`peppers.pgm`).

The subtask graph of this example is shown in Fig. 5. Nodes 1 through 4 are four parallel slices of the image quantization process; node 5 is the Sobel filter; nodes 6, 7 and 8 implement the histogram calculation.

Our second benchmark employed a H.263 video decoder, whose canonical block diagram is illustrated in Fig. 6. The above block diagram may suggest a subdivision into subtasks. Unfortunately, this strategy led to poor results. In fact, profiling revealed that the majority of execution time (> 92%, according to our experiments) is spent inside the “store” block. More precisely, the most time-consuming operation is the conversion from the YUV 4:2:0

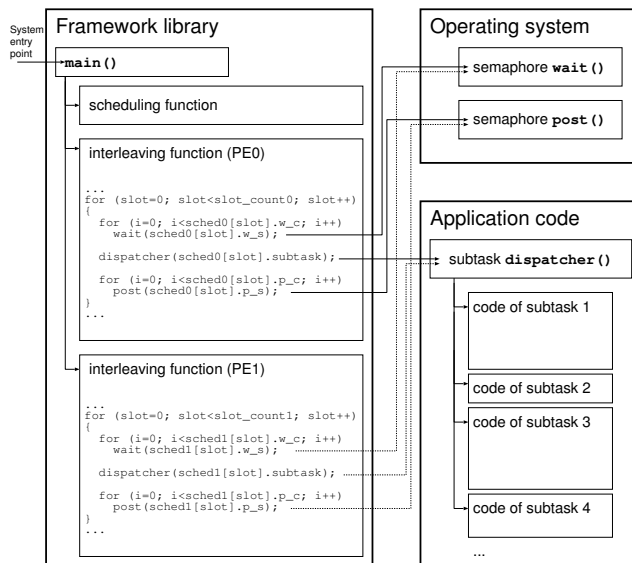


Figure 4. The software architecture of the experiment.

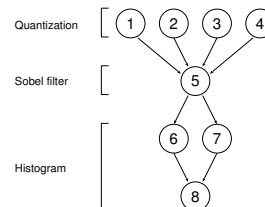


Figure 5. The subtask graph of the ThreeSteps application.

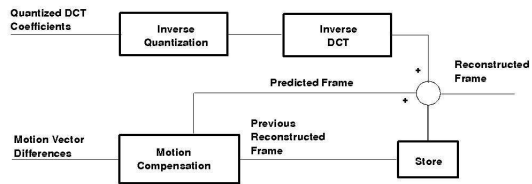


Figure 6. The block diagram of H.263 decoder

encoding to RGB. It is expensive because it requires multiple scans of the frame at full resolution. This clearly suggests to split the store block into multiple subtasks, and to collapse all the other blocks in one subtask. Additionally, we doubled critical buffers in order to increase parallelism, in such a way that even and odd frames can proceed in parallel. The final subtask graph for the H.263 decoder is illustrated in Fig. 7; where: `getpicture` is the collapsed subtask performing the non-store blocks; `copybuffer` performs a buffer copy to increase parallelism; the `U chan` and `V chan` subtasks re-encode chrominance values on the U and V channels; `put RGB` calculates the R, G, and B components of each pixel in the final image; (and 0/1 suffixes indicate the buffer over which the subtask operate).

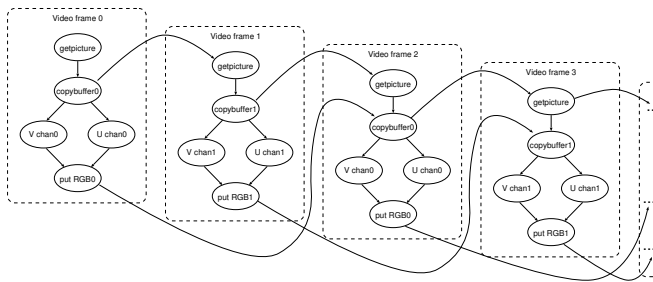


Figure 7. Our subtask graph for the H.263 decoder. Dashed boxes group subtasks associated to the decoding of the same video frame.

The benchmark employed an open-source implementation (tmndec) by Telenor R&D, Norway, and consisted in decoding the first two frames (an intra and an inter frame) of a sample (fmn20.263) video stream, encoded at a QCIF resolution (176×144) with Huffmann compression. The subtask graph associated with the benchmark is depicted in Fig. 8.

We have simulated two scenarios, in which multiple instances of the above benchmarks were executed simultaneously. Each instance is regarded as a task in our experiments. In the first scenario, two TS instances and

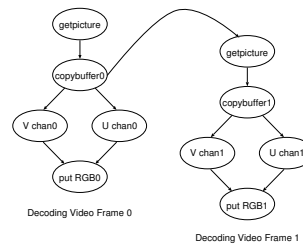


Figure 8. The subtask graph associated to H.263-based benchmark as described.

	$2 \times TS + 1 \times H.263$	$2 \times TS + 2 \times H.263$
Sequential execution	1 403 ms	2 072 ms
Interleaved execution	1 089 ms	1 455 ms
Improvement	22.4%	29.8%
Scheduling overhead	18 ms	24 ms
Relative overhead	1.55%	1.55%

Table 5. Results of interleaving experiments on TI board

one instance of the H.263 decoder were executed concurrently. In the second one, two TS instances and two instances of H.263 were employed. Measurements were repeated multiple times to avoid the spurious influences of non-deterministic behaviors in the RTOS. This was hardly required, because measurements exhibited a relative standard deviation ($\sigma/\mu\%$) less than 0.2%. Experimental results are summarized in Tab. 5. Average speed-ups of 22.4% and 29.8% have been observed for the two scenarios respectively. Additionally, the overhead of the interleaver only accounts for 1.55% of the total run-time.

5 Conclusions

This paper presents a fast, run-time interleaving technique which minimizes idle slots when scheduling multiple concurrent tasks on heterogeneous multiprocessor platforms. The proposed technique can run concurrent tasks in a overlapped fashion and thus improve the performance of the design-time/run-time combined hierarchical system-level scheduling.

Simulations show that our interleaving technique reduces the execution time of randomly-generated subtask schedules by up to 25%; and outperformed the EDF-based DVS method in terms of reducing energy consumptions. We have demonstrated the proposed algorithm on an dual-processor TI TMS320C6202 board, achieving a speedup of

29.8% when two instances of the ThreeSteps and two instances of the H.263 video decoder were interleaved and executed on the board. In the meantime, the measured overhead due to the run-time scheduling execution time was negligible.

References

- [1] A. Andrei, M. T. Schmitz, P. Eles, Z. Peng, and B. M. A. Hashimi. Quasi-static voltage scaling for energy minimization with time constraints. In *Proceedings of the 2005 DATE*, pages 514–519, 2005.
- [2] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. D. Dutt, A. V. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *Proceedings of the Design Automation and Test in Europe*, pages 168–175, 2002.
- [3] R. Dick, D. Rhodes, and W. Wolf. TGFF: Task graphs for free. In *Proceedings of the Sixth International Workshop on Hardware/Software Codesign*, pages 97–101, March 1998.
- [4] M. R. Garey and D. S. Johnson. *COMPUTERS AND INTRACTABILITY: A GUIDE TO THE THEORY OF NP COMPLETENESS*. W.H. Freeman and Company, New York, 1979.
- [5] J. A. Hoogeveen, S. L. V. D. Velde, and B. Veltman. Complexity of scheduling multiprocessor tasks with prespecified processor allocations. Tech. report BS-R9211 ISSN 0924-0659, CWI, the Netherlands, June, 1992.
- [6] N. Jha. Low power system scheduling and synthesis. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD 2001)*, pages 259–263, San Jose, November 2001.
- [7] C.-H. Lee and K. G. Shin. On-line dynamic voltage scaling for hard real-time systems using the EDF algorithm. In *RTSS*, pages 319–327, 2004.
- [8] Z. Ma, C. Wong, S. Himpe, E. Delfosse, F. Catthoor, J. Vounckx, and G. Deconinck. Task concurrency analysis and exploration of visual texture decoder on a heterogeneous platform. In *2003 IEEE Workshop on Signal Processing Systems (SiPS 2003)*, pages 245–250, Seoul, Korea, August 2003.
- [9] R. Stahl, R. Pasko, F. Catthoor, R. Lauwereins, and D. Verkest. High-level data-access analysis for characterisation of (sub)task-level parallelism in Java. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments - HIPS*, April 2004.
- [10] J. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–72, May 1991.
- [11] C. Wong, P. Marchal, and P. Yang. 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, CODES*, pages 170–177, Copenhagen, April 2001.
- [12] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins. A cost-performance tradeoff aware scheduling method for single chip multiprocessor embedded system. *Design & Test of Computers, IEEE*, 18(5):46–58, September 2001.