List Scheduling in Embedded Systems under Memory Constraints

Paul-Antoine Arras, Didier Fuin, Emmanuel Jeannot, Arthur Stoutchinin, Samuel Thibault

List Scheduling in Embedded Systems under Memory Constraints

Paul-Antoine Arras*†‡, Didier Fuin†, Emmanuel Jeannot*, Arthur Stoutchinin†, Samuel Thibault*‡
* Inria Bordeaux Sud-Ouest, Talence, France, first.last@inria.fr
† STMicroelectronics, Grenoble, France, first.last@st.com
‡University of Bordeaux, France

***Abstract*—Video decoding and image processing in embedded systems are subject to strong resource constraints, particularly in terms of memory. List-scheduling heuristics with static priorities (HEFT, SDC, etc.) being the oft-cited solutions due to both their good performance and their low complexity, we propose a method aimed at introducing the notion of memory into them. Moreover, we show that through appropriate adjustment of task priorities and judicious resort to insertion-based policy, speedups up to 20% can be achieved. Lastly, we show that our technique allows to prevent deadlock and to substantially reduce the required memory footprint compared to classic list-scheduling heuristics.**

***Keywords*—Task graphs, scheduling, memory, system on chip, video decoding.**

I. Introduction

At a time when the convergence of digital terminals is pushing the limits of multimedia integration, including for features once reserved to *ad hoc* devices, it is no longer uncommon to come across mobile phones capable of playing streaming video received wirelessly from the Internet. Nonetheless, it does not mean that the operation consisting in decoding a video stream has become a trivial job suitable for sequential processing by any low-end, general-purpose embedded processor. Actually, the complexity [14] of recent video-coding algorithms, such as the H.264/AVC [20] and its successor HEVC [17], makes the use of a single processing element impractical unless poor-quality reproduction is admissible. Instead, the solution consists in resorting to parallel processing with specialized hardware accelerators for a number of performance-demanding tasks.

In this paper, we study parallel scheduling of video-coding and image-quality-improvement applications in an embedded parallel heterogeneous computing environment. In particular, traditional *list-scheduling* heuristics exhibit good performance while remaining of relatively low complexity, and therefore lend themselves well to the lightweight embedded systems. However, existing parallel scheduling algorithms are mostly geared towards high-performance computing with no particular constraints on memory size, whereas in embedded environments reducing memory footprint is of major concern. That is what motivates our work.

We used a model of an embedded platform from STMicroelectronics called STHORM (formerly P2012) [3], [13] for conducting our study. STHORM is a system on chip (SoC) consisting of a number of general-purpose processing elements and specialized hardware accelerators, all sharing a limited amount of level-one memory. In order to take into account the limited level-one memory size of STHORM, we extended the previously proposed list-scheduling heuristics by introducing additional memory constraints to the scheduling process. The main contribution of the paper is the following: as the raw enforcement of memory constraints yields poor schedules or even deadlocks, we devised a scheme that ensures the absence of deadlock and helps find the best trade-off between memory footprint and makespan.

The remainder of this paper is organized as follows: Section II discusses some related work; Section III describes the computation model being used; Section IV formally defines and discusses the problem; Section V presents the core contribution, which is a method to adapt priority of list-scheduling heuristics accounting for memory consideration; Section VI shows our results using a STHORM simulation environment; and finally, Section VII summarizes our contributions and proposes future directions.

II. Related Work

In embedded systems, the problem of executing an application on a SoC is often modeled by scheduling a *dataflow* graph. However, even recent models derived from *synchronous dataflow* (SDF [11]) like *schedulable parametric dataflow* (SPDF [6]), do not take into account all the dynamics of the application, like varying execution time of tasks. Moreover, most SoC's are heterogeneous with general-purpose processors coupled with accelerators (hardware processing element). Such heterogeneity is not captured by these modern dataflow models of computation.

Scheduling task graphs on parallel machines is NP-hard even in the case of homogeneous parallel machines [10]. This justifies using heuristics to address the problem. List scheduling is a technique that is widely acknowledged for its good trade-off between its complexity and the quality of the solution [1]. The principle is to assign priorities to tasks and to sort them in a list ordered by decreasing priority; thus, among available tasks, the first to be executed is always the one having the highest priority, that is the first in the list. As soon as a task has been scheduled, it is removed from the list. Ties are broken randomly, if any.

In the heterogeneous case, many heuristics have been proposed in the literature (see [5] for a study of around 20 of them). Among those, HEFT [18] is a popular list-scheduling heuristics where task priorities are computed using the average bottom level [9]. SDC [16] is another list-scheduling heuristics

aiming at addressing the resource-scarcity issue when only few resources can execute a given subset of tasks.

Concerning memory constraints, preliminary work dates back to register allocation [15]. There also exists work for optimizing footprint for dataflow graphs [4] or for scheduling jobs in batch schedulers [2]. It is also known that optimizing the makespan under resource constraints is NP-Hard for almost all non-trivial problems [10].

Therefore we see that, to the best of our knowledge, we are lacking studies and solutions for scheduling applications on embedded systems using a fast technique (e.g. list scheduling) and dealing with memory constraints and variable task execution times. The goal of the remainder of this paper is to address this need.

III. Definitions and Models

We here expose in further details the context of our work, and the entailed model of the platform, the execution, and the memory constraints.

A. Computing Environment

In the context of embedded image processing, a homogeneous solution based on general-purpose processors would be too expensive and inefficient, while application-specific integrated circuits (ASICs) exhibit very good performance, but are too specialized and lack flexibility. A heterogeneous platform integrated in a SoC comprising both specialized hardware accelerators and general-purpose processors is therefore a widely accepted solution [7], [8], [19].

The target of our research, the STHORM computing platform, consists of both a number of general-purpose, programmable cores, namely *software processing elements* (SW-PEs), and a number of specialized, hardwired accelerators, namely *hardware processing elements* (HWPEs). Two levels of memory are available. The first level is a local memory tightly coupled to PEs, therefore it is more efficient and more costly, thus present in limited amount expressed here in number of *slots*: it only stores the data being currently processed (e.g. a line of pixels or a macroblock from an image). The second level is an external memory located farther from the PEs, therefore suffering from an increased latency while being cheaper and thus able to accommodate much more data, including those already processed and those yet to be processed. The transfers between these two levels are conducted by a *direct memory access* (DMA) controller.

In order to be able to leverage classical scheduling heuristics such as HEFT, while still being general enough to be applied to most real-world embedded architectures, we consider some simplifications, and come up with the following model:

- The platform is composed of several independent *processing elements* (PEs). For a given task, PEs have differing efficiencies according to their type, or may even not be able to execute it at all. For instance, HWPEs can only execute the task they were designed for, and memory-transfer tasks can only be run by the DMA controller, which cannot execute any other kind of task.
- Data originally lie in the external memory, and have to be transferred to the local memory through DMA in order to be worked on.
- To execute tasks, PEs access the data located in the local memory. The latency and bandwidth costs of this access are assumed to be contentionless, and are comprised in the task duration.

The first assumption is not a simplication: it only states how the STHORM platform works. The second one reflects the way target applications (such as image-processing algorithms) are typically implemented on similar architectures for performance matters. The last assumption is the only real simplification: contentionless accesses to the local memory usually cannot be guaranteed on real platforms. Nevertheless, the overhead incurred by contention can be neglected in most cases. Lifting this assumption is left as future work.

B. Execution Model

In the STHORM environment, applications are usually programmed following the dataflow model of computation. An application is thus represented by a dataflow graph (DFG) made of a set of parallel actors connected via a set of FIFOs used for communicating *data tokens*[1]. An application execution consists of multiple parallel firings of actors. Each actor firing consumes some number of data tokens in the actor's input FIFOs, performs some computation based on these input tokens, and produces some number of tokens on the actor's output FIFOs. To adapt this model for list scheduling, we will assimilate the firing of an actor as a *task*. A single actor thus usually generates multiple tasks, one per firing. This results into a classical *directed acyclic graph* (DAG) to be scheduled over the available PEs.

Transforming a DFG into a DAG consists in unrolling several iterations of the DFG by simulating and building the respective tasks and their dependencies. This is a straightforward technique. How many iterations are instantiated depends on the following factors. On the one hand, the more iterations the larger the DAG and the better our understanding of the application. It is therefore easier to take good scheduling decisions if we have a large graph. On the other hand, the DAG can become very large and therefore the time for scheduling can increase sharply. More importantly, the size of the schedule may exceed the available memory to store it on the embedded system. The solution consists in finding a trade-off between the quality of the schedule and its size. Such decision is left to the decision maker (application designer). But technically it is possible to apply the same schedule window by window as if the DFG were unrolled dynamically.

C. Memory Model

To take memory constraints into account, we introduce a new, dedicated kind of tasks: memory-slot allocation and release. Once a memory slot has been allocated by an allocation task, its reference is passed between actors as a data token, up to the task which releases it. Such kind of tasks can only be

[1] A token is the smallest unit of data that can be processed by a task. It is application specific; e.g. for an image-processing algorithm, it can be a line of pixels.

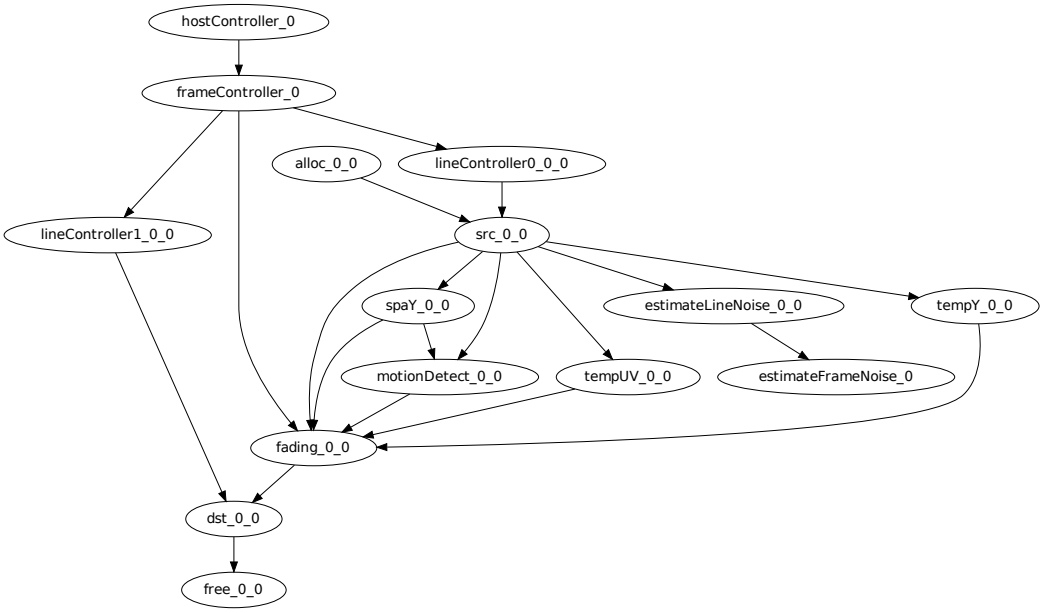


Fig. 1. Example DAG for the TNR algorithm. A single line of pixels is handled. For n lines, double-suffixed tasks have to be run n times. `alloc_0_0` consumes memory while `free_0_0` releases memory. `estimateFrameNoise_0`'s successor is `frameController_1` and is thus not represented on this figure.

run by a SWPE, and their scheduling is more complex than regular tasks.

Indeed, when it is run, each of them can either consume or release a certain amount of local memory expressed as a number of tokens. In order to keep the model simple, we assume—without loss of generality—that one memory slot can accommodate exactly one data token. The token transfer of such a task is expressed as an algebraic cost: positive if it allocates memory or negative if it releases memory. The number of available slots is updated on each task execution by subtracting algebraically its cost; it shall always be nonnegative: when it becomes zero, the scheduler first has to schedule some releaser tasks before being allowed to schedule other allocators.

Figure 1 illustrates the model described above with a DAG representing an image-quality-improvement algorithm that applies a *temporal noise reduction* (TNR) to each line of pixels. The graph comprises only one instance (i.e. task) of each actor because any one of them does the same parallel processing on all pixel lines included in the frames that compose a video sequence[2]. Simple-suffixed nodes (e.g. `frameController_0`) are executed once per frame while double-suffixed nodes (e.g. `tempUV_0_0`) are run once per line; the numbers indicate image and line numbers, respectively.

The TNR application works as follows: `hostController` is run by the host processor of the SoC to introduce a full frame into external memory; `frameController` launches the processing from a SWPE; `lineController0` and `1` program the DMA to, respectively, read and write the data in external memory. The critical part begins with the `alloc` actor which allocates a memory slot for a whole line. This slot is filled by a transfer from the external memory by the `src` actor, and after treatement is transferred back to external memory by the `dst` actor, after which the memory slot can be released by the `free` actor. `estimateLineNoise` and `estimateFrameNoise` evaluate frame n's noise level so as to calibrate the processing for frame $n+1$. Lastly, `spaY`, `tempUV`, `tempY` and `motionDetect` analyze the frame in order for `fading` to be able to apply the appropriate correction.

It should be noted that `src` and `dst` can only run on the DMA. As we have only one DMA controller on the platform, these tasks are serialized during the execution of the graph. This scheme ensures the absence of contention on the DMA: memory transfers are executed one after the other.

[2]Thus, from a processing viewpoint, pixel lines are independent.

IV. Problem Definition

A. Inputs

Based on the models described in Section III, we define the problem we tackled as follows. Let $G = (V, E)$ be a directed acyclic task graph (DAG) modeling the application. Each task $v_i \in V$ corresponds to a firing of an actor and each edge $(v_i, v_j) \in E$ models a dependency between two tasks. We have a heterogeneous environment composed of m heterogeneous processing elements (PEs) being all able to access S memory slots. The duration of task v_i on PE j is noted $p_{i,j}$. When a PE j cannot execute task v_i we have $p_{i,j} = +\infty$. Otherwise, to account for the fact that task durations may depend on the input data, $p_{i,j}$ is a random variable that follows a law in $[0, +\infty[$. We also need to distinguish the *memory tasks*, which allocate or release memory. They have a negligible but

non-zero duration. We call $V_M \subset V$ the set of all memory tasks. The number of memory slots allocated or released by task $v_i \in V_M$ is $\text{cost}(v_i)$, which is positive when the task allocates slots (*consumer* task), or negative when the task releases slots (*releaser* task). Each consumer task is paired with the corresponding releaser task, therefore we have a bijection function pair:

$$\forall v_i \in V_M, \exists! v_j \in V_M, \begin{cases} v_j = \text{pair}(v_i) \in V_M \\ \text{cost}(v_i) + \text{cost}(v_j) = 0 \end{cases} .$$

Lastly, there always exists a path from v_i, $\text{cost}(v_i) > 0$, to $\text{pair}(v_i)$ in G to ensure that the reference of the allocated memory slot is passed from actor to actor, starting from its consumer task, down to its releaser task.

B. Metrics

The goal of the problem is to schedule the tasks on the availables PEs, respecting resource constraints and task dependencies. We have two metrics to optimize: the average makespan $C_{\max}$ (i.e. the finish time of the last task) and the average memory usage $M_{\max}$. We take an average metrics to account for random task durations. The memory-usage metrics is defined as follows. Given a schedule, let $M(t)$ be the memory usage of the schedule at time t. By definition:

$$M(t) = \sum_{v_i \in V_M^{\leq}(t)} \text{cost}(v_i) \ ,$$

where $V_M^{\leq}(t) \subset V_M$ is the set of memory tasks scheduled up to time t. Hence, we have:

$$M_{\max} = \max_{t \in [0, C_{\max}]} M(t) \ ,$$

and the schedule has to respect the available number of slots:

$$M_{\max} \leq S \ .$$

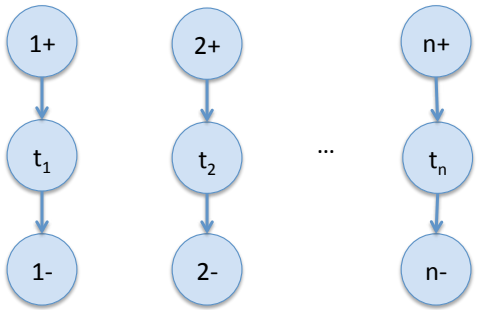
C. Discussion

The above problem is a multi-criteria problem as memory usage and makespan are conflicting objectives. Indeed, as shown in the DAG of Fig. 2(a), if we schedule sequentially each 3-task thread on a homogeneous set of processors, we reach $M_{\max} = 1$ but $C_{\max} = n$, and if we parallelize on n resources we have $C_{\max} = 1$ but $M_{\max} = n$.

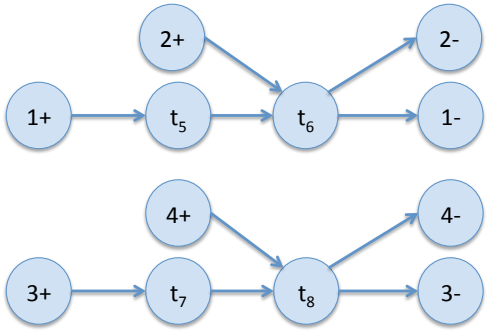
Moreover, it is well known that minimizing $C_{\max}$ alone is NP-hard. But, minimizing $M_{\max}$ alone is NP-hard as well. A possible reduction comes from the following NP-complete version of *pebble game*. It consists in deciding if a DAG can be pebbled with less than K pebbles where 1) pebbles can be placed on the node of the DAG only when all its predecessors are pebbled, 2) pebbles can be removed at any time and 3) nodes can be pebbled only once [15].

D. Motivating Example

Not all scheduling heuristics which respect precedence constraints can produce valid schedules respecting memory constraints. Indeed it may happen, if we do not have enough memory slots, that the scheduling heuristics *deadlocks*. An example of DAG that leads to deadlock is given in Fig. 2(b). On one processor, the scheduling sequence 1+, 3+, t_5, t_7 deadlocks if we have only two memory slots. Indeed, after executing t_5 and t_7, the only task that can be executed consumes memory (2+ and 4+). With two memory slots a solution consists in executing the upper part and the inferior part of the DAG sequentially: the sequence 1+, 2+, t_5, t_6, 1-, 2-, 3+, 4+, t_7, t_8, 3-, 4- is a valid schedule with 2 available memory slots. Therefore, having a scheduling heuristics that takes into account memory constraints is necessary to obtain schedules that do not deadlock.



(a) DAG leading to conflicting objectives for makespan and memory consumption.



(b) DAG leading to a deadlock even with two available memory slots.

Fig. 2. $\text{cost}(i_+) = +1$, $\text{cost}(i_-) = -1$, and duration of computing tasks t_i is 1 on all processors.

V. Solution Description

Definition 1: A *memory set* is a set of DAG nodes located on a path from a consumer to its paired releaser, including those two. Memory sets are clustered into *memory clusters* such that a memory cluster comprises all memory sets which have intersecting nodes.

Following this definition, a memory set which has no vertex in common with any other memory set is also a memory cluster. For instance, in the graph represented on Fig. 1, the memory cluster corresponding to the processing of line 0 from frame 0 consists of `alloc_0_0`, `free_0_0` and those eight tasks located between them. Additionally, Figure 3 shows a more complex graph with several memory clusters. In the remainder, we will only consider memory clusters.

Definition 2: The *achievable lower bound* (ALB) of the memory cost is the maximum number of consumed memory slots by a memory cluster, over all memory clusters.

Then we derive two conditions which permit to achieve this lower bound:

C1: The sets of priorities of consumer tasks from different clusters do not overlap.

C2: Consumer tasks from ancestor clusters have higher priorities.

Proposition 1: Conditions C1 and C2 are sufficient to schedule under the ALB.

Proof: Due to space limitation, we only give the intuition. Let us consider two disconnected clusters A and B. As A and B are disconnected, Condition C1 guarantees that if a consumer from A is scheduled first then all consumers from A will be scheduled before those from B, which will ensure no deadlock due to lack of memory (for instance, in the case of the DAG of Fig. 2(b), this will ensure that 1+ and 2+ are scheduled together, before 3+ and 4+, and thus the whole cluster will be schedulable). Now assume that some node in B has an input dependency from a node in A. Then Condition 2 demands that consumers from A should be scheduled first. Consequently, the dependency will be satisfied when B's consumers are scheduled, ensuring no memory waste. ■

In order to meet these conditions, the scheduling process has to be adapted since the mere counting of memory slots introduces implicit dependencies that do not appear in the initial graph and therefore cannot be accounted for by the usual schedulers. To solve this issue, we devise a new task graph:

Definition 3: The *independence graph* associated with an application is an undirected graph whose vertices represent only memory tasks. The edges are such that two nodes are connected if and only if there exists no path between them in the original DAG.

The idea is to account for the precedence relations between memory tasks that do not appear as data dependencies. Using this graph allows for a priority adjustment so as to bring forward the execution of releaser tasks, since they constitute the main locking point in the schedule.

A. Priority Adjustment

We now introduce an adjustment of priorities for memory constraints, which can be applied to static-priority–based list-scheduling algorithms.

Each releaser task v_r will get a priority bonus P_B equivalent to the total priorities of the set V_C^* of tasks v_c satisfying the following requirements:

1) v_c is adjacent to v_r in the independence graph;
2) cost(v_c) > 0, i.e. v_c is a consumer;
3) one of the following holds:
   a) $P(v_c) < P(\text{pair}(v_r))$ where $P(v)$ gives the priority of any task v,
   b) pair(v_c) is *not* adjacent to pair(v_r) in the independence graph.

$$P_B(v_r) = \sum_{v_c \in V_C^*} P(v_c)$$

Requirement 1 ensures that only tasks with no pre-existent precedence relation are considered, to avoid producing a bonus loop; Requirement 2 prevents releaser tasks from influencing one another; Requirements 3a and 3b respectively aim at meeting Conditions C1 and C2. These adjusted priorities are then propagated to the rest of the DAG through a second pass of the regular task-prioritizing phase.

In some few cases, this priority-adjustment scheme is not sufficient to meet Condition C1. In such cases, we give in Algorithm 1 a way to enforce C1 directly. The rationale behind this algorithm is that the priorities of some consumer tasks may have to be raised in order to avoid overlapping between clusters. To do so, the priority list is traversed backward and each time overlapping clusters are detected the priority of the lower-ranked consumer is raised. Thanks to this scheme, priority forcing does not alter already-traversed tasks and the algorithm requires only one pass.

```
Count the number of consumers in each cluster;
// Traverse the priority list backward
   considering only consumer tasks
foreach new cluster C traversed do
    if all tasks in preceding cluster C' have not already
    been traversed then
        Find task T' from C' with highest priority
        while there are tasks T in C such that
        P(T) < P(T') do
            // Raise priority of task T
            P(T) ← P(T') + 1
        end
    end
end
```

Algorithm 1: Priority forcing

In practice, the extra bonus that this algorithm introduces is very small, i.e. the initial priority adjustment is already very good.

B. Insertion-based Policy

Many scheduling heuristics (e.g. HEFT, SDC) provide insertion mechanisms to schedule tasks in idle-time slots. We here show how to adapt this mechanism for memory tasks, whose insertion also has to respect memory constraints.

Let $s(t)$ be the number of available slots at time t; $s(t)$ represents the state of the local memory at any step of the scheduling process and is supposed to be retrievable for any previous step $t_0 < t$. Let $I(t)$ be the set of idle-time slots at time t. For all $i \in I(t)$, we define start(i) the start time of i and end(i) the end time of i, then we derive the duration of i: $d(i) = \text{end}(i) - \text{start}(i)$ [3]. Let $V_M^=(t)$ be the set of all memory tasks running at time t. Then, a consumer task v_c can be inserted in a given i if the following assertions hold:

- the considered slot has enough memory to accommodate the memory cost of v_c:

$$d(i) \geq d(v_c) \ ,$$

$$\exists (t_0, t_0') \in [\text{start}(i), \text{end}(i)]^2, \begin{cases} \forall t \in [t_0, t_0'], s(t) \geq \text{cost}(v_c) \\ t_0 \leq \text{EST}(v_c) \leq t_0' \end{cases} ;$$

[3] Through a slight abuse of notation, we also use d to denote the duration of tasks.

- insertion will not affect subsequent, already-scheduled tasks:

$$\forall t \geq \text{EST}(v_c), s(\text{EST}(v_c)) + \text{cost}(v_c) + \sum_{v_m \in \bigcup_{t' \in [\text{EST}(v_c), t]} V_M^{\overline{=}}(t')} \text{cost}(v_m) \geq 0 \ ,$$

where $\text{EST}(v)$ denotes the estimated start time of task v.

C. Self-timed Scheduling

To cope with the randomized task durations of the problem, we have modified the list heuristics as follows. First, we compute the priority and a static schedule of each task by using the average of the random variable $p_{i,j}$ that gives the duration of task i on processor j. Then, when we actually execute the application we use this precomputed schedule to allocate and order the tasks: during the real execution each task is executed on the same processor and in the same order as what was computed by the schedule. However, as task durations may diverge from the average value used to compute the schedule, the start times of the tasks change as well. Hence, a task is executed as soon as its dependencies (in the DAG) are satisfied and its preceding task (on its allocated processor) is terminated. For this reason, we call this technique *self-timed* scheduling [12] as only the allocation and the order respect the static schedule while the start time is computed dynamically. By doing this procedure several times, the observed average of the different obtained makespans approaches the expected makespan of the schedule.

VI. Experiments

We implemented our contributions, namely the priority-adjustment method and the insertion-based policy for memory tasks, into HEFT and SDC. It should be noted that they are both compatible with any static-priority–based list-scheduling algorithm.

We carried out experiments on two real-world applications: the TNR presented in Section III-C and the H.264 video coding algorithm [20].

All our experiments consisted in comparing the makespans of schedules with and without our priority-adjustment method for different memory-slot numbers; insertion-based policy is always used. Makespan values are averages on a thousand executions with random task durations.

Random task durations are computed through the following strategy:

1) For each type of actor (`src`, `fading`, etc.), we define a unitary duration per number of pixels.
2) We determine the reference duration w^r for each actor by multiplying the unitary duration by the number of pixels that are processed (line or macroblock).
3) In order for all instances of a given task to get a similar variation, we first set the average random duration $\bar{w}$ of this actor by choosing a dispersion factor $a \geq 1$ such that $\bar{w} \in [\frac{w^r}{a}, aw^r]$. To do so, we use the beta law which has a support on $[0, 1]$ and when $\alpha = \beta$ has a mean 0.5. Here, we use $(\alpha, \beta) = (2, 2)$:

$$\bar{w} = w^r \left(\text{Beta}(\alpha, \beta)(a - 1/a) \ + 1/a\right) \ .$$

Moreover, we impose that:

$$\forall i, a \leq \sqrt{w^r_{i,j_s} / w^r_{i,j_h}} \ ,$$

where w^r_{i,j_s} and w^r_{i,j_h} are reference durations of task v_i respectively on a SWPE and a HWPE. This ensures that a SWPE is never faster than an HWPE.
4) The final duration of each task instance is computed similarly with the same dispersion factor a.

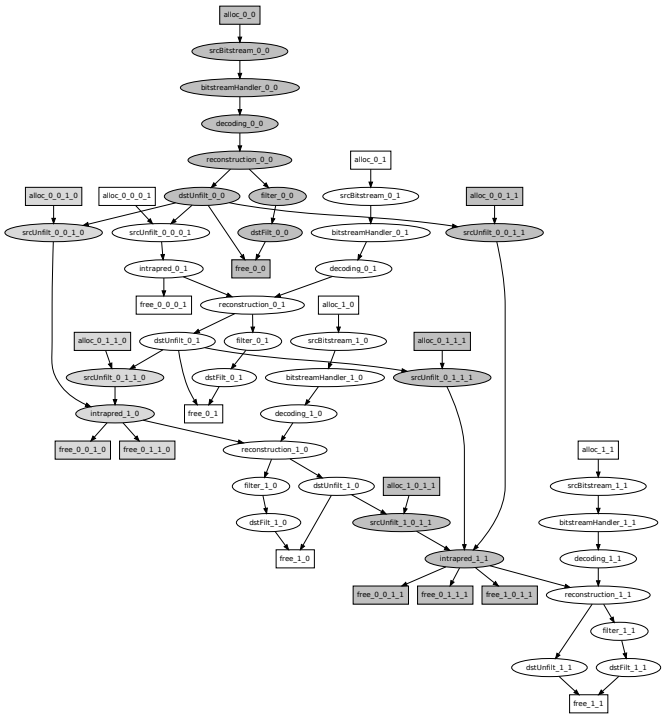
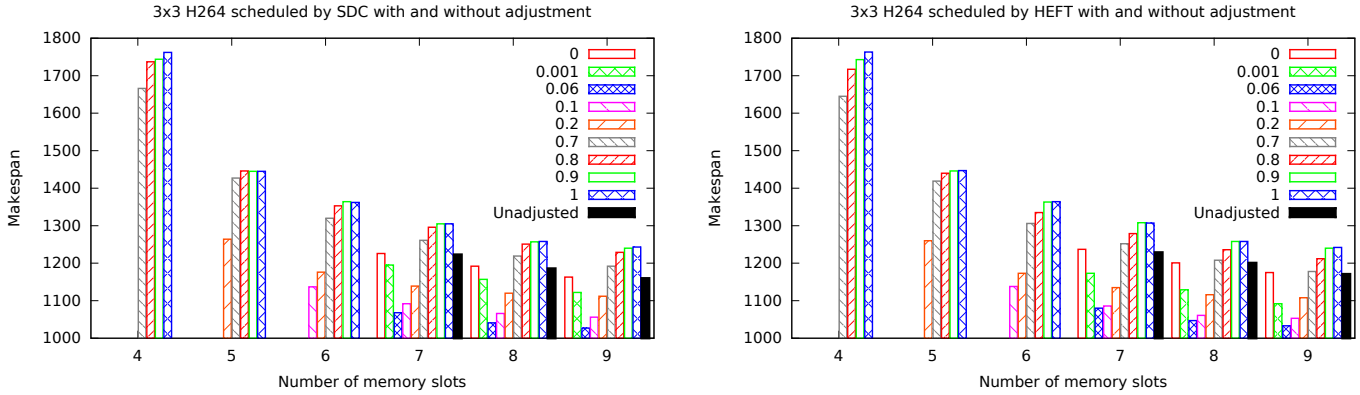


Fig. 3. H.264 task graph for 4 dependent macroblocks. 3 out of 7 memory clusters are shown in different shades of grey. Allocator and releaser tasks appear in square boxes.

A. TNR

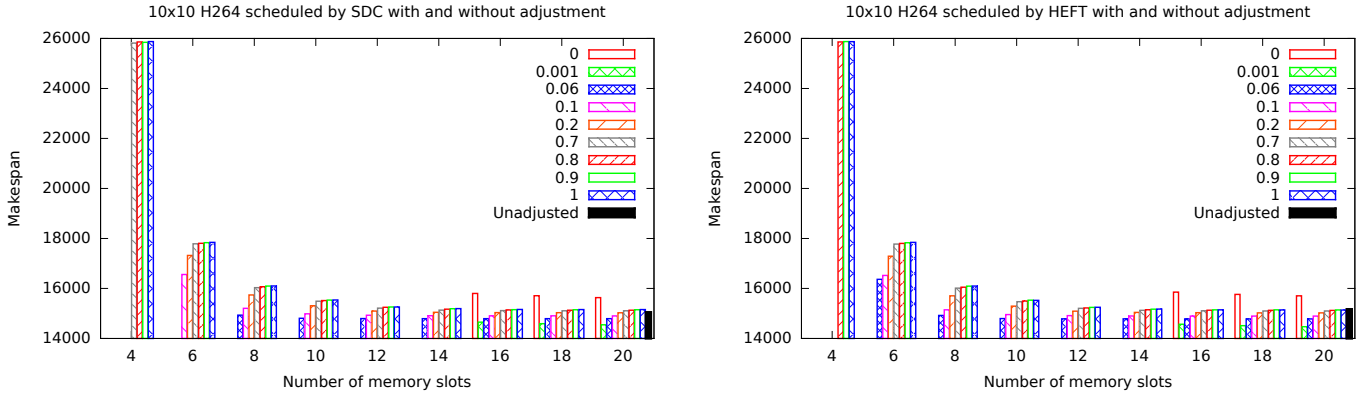
Our heuristics were fed with a DAG describing the processing of 10 lines of 1000 pixels each. Since this simple example has no risk of deadlock, Algorithm 1 of priority forcing is not used. The simulated platform is composed of 1 DMA, 1 SWPE, and 5 HWPE (one per accelerated computation actor). Figure 6 illustrates the results. Schedules with priority adjustment always outperform their unadjusted counterparts; this is true both for HEFT and SDC. Speedups range from 4 % for 1 slot to 20 % for 2 slots, and 10.6 % on average. The low speedup for 1 slot can be explained by the low pipelining potential since only one line can be processed at at time. Conversely, the high speedup for 2 slots is due to the wrong decisions taken by the unadjusted versions which try to schedule all consumers at once since they have the same priority. However this gap vanishes when the amount of memory increases.



(a) SDC heuristics with different bonus factors

(b) HEFT heuristics with different bonus factors

Fig. 4. H.264 with 3x3 macroblocks. The missing bars mean that the version of the heuristics produces a schedule that deadlocks.



(a) SDC heuristics with different bonus factors

(b) HEFT heuristics with different bonus factors

Fig. 5. H.264 with 10x10 macroblocks. The missing bars mean that the version of the heuristics produces a schedule that deadlocks.

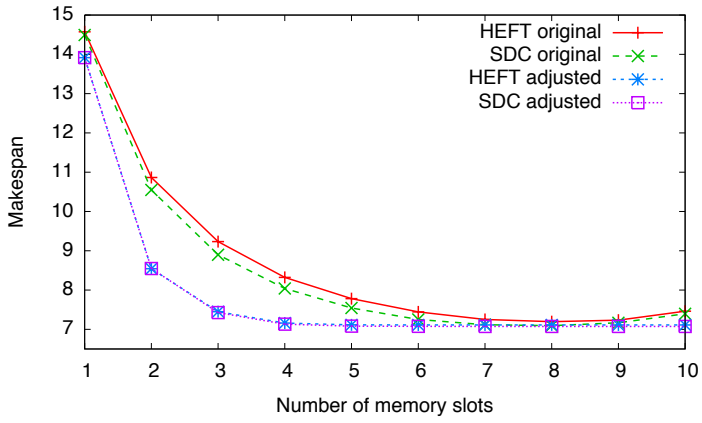


Fig. 6. TNR with 10 lines of 1000 pixels

B. H.264

We used a simplified model of an H.264 decoder illustrated by Fig. 3. The base unit of the decoding process is the macroblock (MB), which is a contiguous set of—typically—16 lines of 16 pixels. Each MB is processed as follows: the first stage is the decoding (entropy, dequantization, etc.) of the current MB; the second step is the intraprediction[4] using at most 4 previously decoded MBs; the third step is the reconstruction of the original MB; the final step is the filtering. Each use of an MB, either as reference or while being decoded, must be preceded by a memory allocation modeled by a consumer task in the DAG and followed by a memory release modeled accordingly. For the sake of simplicity, buffers are not reused, hence the need to systematically reload the MBs required for the computation. Optimizing this scheme is left as future work. Thus, the tasks processing subsequent MBs—in raster-order image scanning—have data dependencies from earlier-MB tasks.

Contrary to the TNR, it is not possible to schedule the H.264 under an arbitrary low number of memory slots, as some tasks need 4 MBs at the same time. The ALB is actually 4. The simulated platform is composed of 1 DMA, 1 SWPE, and 4 HWPE (one per accelerated computation actor).

Schedules with priority adjustment do not outperform the unadjusted counterparts anymore, on the contrary. This is due to the priority adjustment tending to prevent the pipelining of the dataflow instances. We have thus tried to use a *bonus factor*

[4]To keep the model simple, interprediction is not considered.

$BF \in [0,1]$ to mitigate the priority adjustment as follows: $\forall v \in V, P_{\text{adjusted}}(v) = P_{\text{original}}(v) + P_B(v) * BF$.

In the first set of simulations, the schedulers were fed with a DAG describing the processing of 3 lines of 3 MBs (3x3). Figure 4 illustrates the results. When there is no bar, it means that the schedule deadlocks due to lack of memory. We see that the lower the bonus factor the larger the number of memory slots required to produce valid schedules. This is due to the fact that with a low bonus factor the adjusted priority is very close to the original priority (see above formula). With a bonus factor of 0, only priority forcing (see algorithm 1) is performed. Unadjusted schedulers are unable to produce legal schedules below 7 slots while their adjusted counterparts can, but at the cost of a higher makespan. Changing the bonus factor permits to tune the benefits of both aspects, and we can see that a speedup can be reached (around $BF = 0.01$) up to 13 % for 7 slots, 12 % for 8 slots and 11 % for 9 slots. In the worst case, the adjusted version is 6 % slower but ensures the absence of deadlock. However, it is always possible to outperform the original HEFT or SDC with our adjustement technique. Moreover, if we compare Fig. 4(a) with 4(b), we see that there is no real difference between HEFT and SDC in our case. Like for the TNR, makespans and speedups decrease as the memory constraint is loosened since the processing of different MBs can then be further pipelined. Conversely, for 4 slots the makespan is particularly high because most MBs have to be processed sequentially.

In the second set of simulations, the schedulers were fed with a DAG describing the processing of 10 lines of 10 MBs (10x10). Figure 5 illustrates the results. The outcome is similar, except that the original HEFT and SDC algorithms are not able to produce legal schedules with less than 19 slots, while the adjusted variants are able to produce legal schedules with as few as 4 slots.

The overall results show very close performance for HEFT and SDC. This demonstrates the ability of our contributions to be applied to different existing heuristics with equal benefits.

VII. Conclusion

In this paper, we have presented extensions to list-scheduling algorithms for taking into account memory requirements. This is done through a new model featuring *memory tasks* and priority adjustment of the tasks. Moreover, we have shown how to extend task insertion to this case. Experiments on TNR show that we can achieve a makespan gain up to 20%. For complex applications (e.g. H.264), we show that a strong priority adjustment prevents deadlock contrary to unmodified heuristics. Moreover, we have explored the trade-off between makespan and memory consumption and we have shown that we are able to find schedules that outperform original heuristics for both criteria.

Our future work is directed toward dynamic scheduling. We want to study how on-line scheduling is able to better cope with the dynamics of the application: when the structure as well as the duration of the tasks are not fully known in advance. More specifically, we will address the issues stemming from the scheduling of video coding algorithms such as H.264 and HEVC, mainly: hardware/software partitioning, execution model, parameter passing and graph reconfiguration.

References

[1] T. L. Adam, K. Chandy, and J. Dickson, "Comparison of list schedules for parallel processing systems." *Communications of the ACM*, vol. 17, no. 12, pp. 685–690, 1974.

[2] A. Batat and D. Feitelson, "Gang scheduling with memory considerations," in *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*. IEEE, 2000, pp. 109–114.

[3] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," 2012, pp. 983–987.

[4] J. Buck and E. Lee, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," in *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-93)*, vol. 1. IEEE, 1993, pp. 429–432.

[5] L.-C. Canon, E. Jeannot, R. Sakellariou, and W. Zheng, "Comparative evaluation of the robustness of dag scheduling heuristics," in *Grid Computing*, S. Gorlatch, P. Fragopoulou, and T. Priol, Eds. Springer US, 2008, pp. 73–84.

[6] P. Fradet, A. Girault, P. Poplavko *et al.*, "Spdf: A schedulable parametric data-flow moc (extended version)," 2011, inria RR7828.

[7] T. Geng *et al.*, "Parallelization of computing-intensive tasks of the h.264 high profile decoding algorithm on a reconfigurable multimedia system," *IEICE Transactions on Information and Systems*, vol. E93-D, no. 12, pp. 3223–3231, 2010.

[8] G.-A. Jian *et al.*, "A system architecture exploration on the configurable hw/sw co-design for h.264 video decoder," 2009.

[9] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/344588.344618

[10] E. L. Lawler, J. K. Lenstra, A. R. Kan, and D. B. Shmoys, "Sequencing and scheduling: Algorithms and complexity," *Handbooks in operations research and management science*, vol. 4, pp. 445–522, 1993.

[11] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[12] E. A. Lee and S. Ha, "Scheduling strategies for multiprocessor real-time dsp," *IEEE Global Telecommunications inproceedings and Exhibition*, vol. 2, 1989.

[13] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, "Platform 2012, a many-core computing accelerator for embedded socs: Performance evaluation of visual analytics applications," 2012, pp. 1137–1142.

[14] S. Saponara *et al.*, "Performance and complexity co-evaluation of the advanced video coding standard for cost-effective multimedia communications," *Eurasip Journal on Applied Signal Processing*, vol. 2004, no. 2, pp. 220–235, 2004.

[15] R. Sethi, "Complete register allocation problems," *SIAM journal on Computing*, vol. 4, no. 3, pp. 226–248, 1975.

[16] Z. Shi and J. J. Dongarra, "Scheduling workflow applications on processors with different capabilities," *Future Generation Computer Systems*, vol. 22, no. 6, pp. 665 – 675, 2006.

[17] G. Sullivan and J.-R. Ohm, "Recent developments in standardization of high efficiency video coding (hevc)," *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 7798, 2010.

[18] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Task scheduling algorithms for heterogeneous processors," in *8th IEEE Heterogeneous Computing Workshop (HCW'99)*, San Juan, Puerto Rico, Apr. 1999, pp. 3–14.

[19] S.-H. Wang *et al.*, "A software-hardware co-implementation of mpeg-4 advanced video coding (avc) decoder with block level pipelining," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 41, no. 1, pp. 93–110, 2005.

[20] T. Wiegand *et al.*, "Overview of the h.264/avc video coding standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560–576, 2003.