List Scheduling in Embedded Systems Under Memory Constraints

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

International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

List scheduling in embedded systems under memory constraints

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

Received: date / Accepted: date

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 adequate adjustment of task priorities and judicious resort to insertion-based policy, speedups up to 20% can be achieved. We also show that our technique allows to prevent deadlock and to substantially reduce the required memory footprint compared to classic list-scheduling heuristics. Lastly, we propose a methodology to assess the appropriateness of dynamic scheduling in this context.

Keywords Task graphs · scheduling · memory · system on chip · video decoding

1 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 [22] of recent video-coding algorithms, such as the H.264/AVC [29] and

P.-A. Arras · E. Jeannot · S. Thibault
Inria Bordeaux Sud-Ouest, Talence, France
E-mail: first.last@inria.fr

P.-A. Arras · D. Fuin · A. Stoutchinin
STMicroelectronics, Grenoble, France
E-mail: first.last@st.com

P.-A. Arras · S. Thibault
University of Bordeaux, France

its successor HEVC [26], 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.

In this work, we used a model of an embedded platform from STMicroelectronics called STHORM (formerly P2012) [4, 21] 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 very limited amount of level-one memory (typically 256 KiB). 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 to find the best trade-off between memory footprint and makespan. We also present a method to help position the scheduling strategy within the spectrum from static to dynamic.

The remainder of this paper is organized as follows: Section 2 discusses some related work; Section 3 describes the computation model being used; Section 4 formally defines and discusses the problem; Section 5 presents the core contribution, which is a method to adapt priority of list-scheduling heuristics accounting for memory consideration; Section 6 shows our results using a STHORM simulation environment; and finally, Section 7 summarizes our contributions and proposes future directions.

2 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 [17]) like *schedulable parametric dataflow* (SPDF [8]), 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 elements). 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 [16]. 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 [6] for a study of around 20 of them). Among those, HEFT [27] is a popular list-scheduling heuristics where task priorities are computed using the average *bottom level*[1] [15]. SDC [24] is another list-scheduling heuristics aiming at addressing some additional issues, including resource scarcity—when only few resources can execute a given subset of tasks—and descendant effect—considering scheduling a task on a less powerful processor if it cuts communication costs. Moreover, there exist list-scheduling heuristics based on non-static priorities: for instance, Dynamic-Level Scheduling (DLS) [25] has priorities varying during the scheduling process. This class of heuristics is excluded from our study, in spite of good performance, due to their huge complexity and running times: DLS's time complexity is $O(v^3 \times q)$, where v is the number of tasks and q the number of processors, and it has been shown that, compared to a number of heuristics with static priorities, it is the slowest [27].

Concerning memory constraints, preliminary work dates back to register allocation [23]. There also exists work for optimizing footprint for dataflow graphs [5] or for scheduling jobs in batch schedulers [3]. It is also known that optimizing the makespan under resource constraints is NP-Hard for almost all non-trivial problems [16]. For some application-specific research, there exists work aimed at minimizing the memory footprint. This is the case for direct sparse matrix solvers [11,19]. Recent work [12,20] has studied the case of parallelizing tree-shaped task graphs targeting memory usage and makespan. Their model is slightly different from ours as the memory cost is associated with each task. Our model is somehow more general as we can express the fact that memory slots are shared across different tasks (in this case, when two independent tasks share the same slot, the memory cost does not depend on whether they are executed sequentially or in parallel). This work has been recently extended to arbitrary structures in [13]. In all cases, minimizing the memory footprint is NP-hard. Interestingly [20] shows that for tree-shaped applications each criteria (makespan and memory constraints) can be optimized optimally in polynomial time, but the multi-criteria problem (minimizing makespan under a given memory bound) is NP-hard.

Lastly, as regards real-time scheduling, [2] presents a scheme that takes memory constraints into account, but it is more geared toward *hard* real-time tasks with deadlines, which is not compatible with our model.

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.

[1] The bottom level is also sometimes referred to as the *upward rank*.

3 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.

3.1 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 [9, 14, 28].

The target of our research, the STHORM computing platform, consists of both a number of general-purpose, programmable cores called *software processing elements* (SWPEs) executing generic software such as the runtime system and software implementations of filters, and a number of specialized, hard-wired accelerators, called *hardware processing elements* (HWPEs) which execute hardware-implemented filters. 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 available 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[2] while being cheaper and thus able to accommodate much more data, including those already processed and those yet to be processed. 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 simplifying assumptions, 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 actually not a simplification: 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 per-

[2] The orders of magnitude of the latency for local and external memories are respectively 1 cycle and 100 cyles.

formance 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.

3.2 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*[3]. An application execution consists of multiple parallel *firings* of actors. Each actor firing consists of three ordered and indivisible steps: consuming some number of data tokens in the actor's input FIFOs, performing some computation based on these input tokens, and producing 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 with a *task*. A single actor thus usually generates multiple tasks, one per firing. This results in 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 application designer. Technically it is however possible to apply the same schedule window by window as if the DFG were unrolled dynamically.

3.3 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 that releases it. Such kind of tasks can only be 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 given 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

[3] 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.

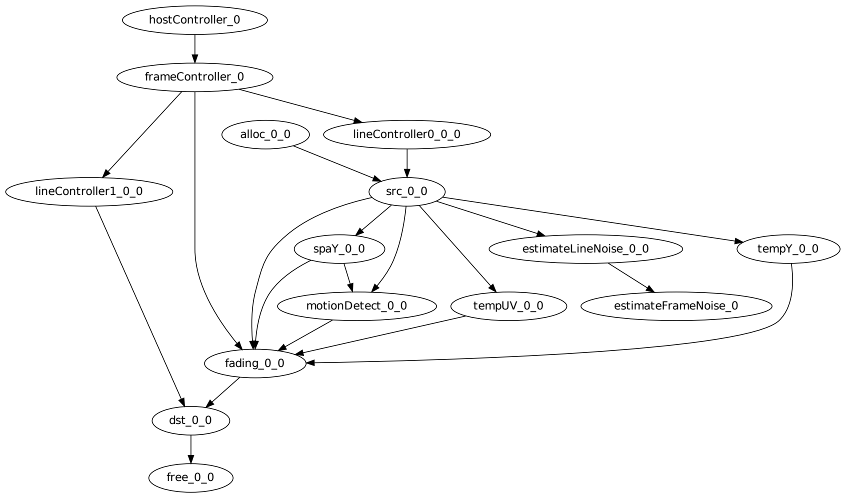


Fig. 1 Example DAG for the TNR algorithm. A single line of pixels is handled. For n lines, double-suffixed tasks have to be run n times. `alloc_0_0` consumes memory while `free_0_0` releases memory. `estimateFrameNoise_0`'s successor is `frameController_1` and is thus not represented on this figure.

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[4]. 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 the external memory; `frameController` launches the processing from a SWPE; `lineController0` and `1` program the DMA to, respectively, read and write the data in the external memory. The critical part begins with the `alloc` actor which allocates a memory slot for a whole line in the local memory. This slot is filled by a transfer from the external memory by the `src` actor, and after treatment (described below) is transferred back to the external memory by the `dst` actor, after which the memory slot in the local memory 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`,

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

`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 data races on the DMA: memory transfers are executed one after the other.

4 Problem Definition

Based on the models described in Section 3, we define the problem we tackled as follows.

4.1 Inputs

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 in the local memory. The duration of task v_i on PE j is noted $w_{i,j}$. When a PE j cannot execute task v_i we have $w_{i,j} = +\infty$. Otherwise, to account for the fact that task durations may depend on the input data, $w_{i,j}$ is a random variable that follows a law in $[0, +\infty[$.

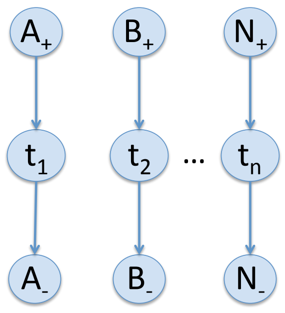
We also need to distinguish the *memory tasks*, which allocate or release memory. They have negligible but non-zero durations. We call $V_M \subset V$ the set of all memory tasks. The number of memory slots allocated or released by task $v_i \in V_M$ is $\text{cost}(v_i)$, which is positive when the 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 called pair:

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

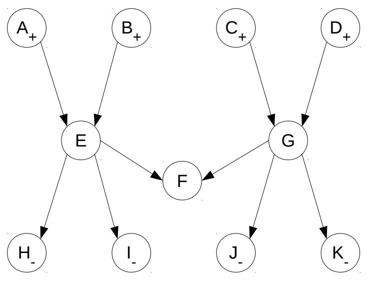
Lastly, there always exists a path from v_i, $\text{cost}(v_i) > 0$, to $\text{pair}(v_i)$ in G to ensure that the reference of the allocated memory slot is passed from actor to actor, starting from its consumer task, down to its releaser task.

4.2 Metrics

The goal of the problem is to schedule the tasks on the available PEs in compliance with 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.



(a) DAG leading to conflicting objectives for makespan and memory consumption. Duration of tasks t_i is 1 on all processors.



	Original priority
A_+	8
B_+	12
C_+	8
D_+	12
E	6
F	1
G	6
H_-	1
I_-	2
J_-	1
K_-	2

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

Fig. 2 DAG examples with memory-slot allocations and releases: $\text{cost}(i_+) = +1$, $\text{cost}(i_-) = -1$.

Given a schedule, let $M(t)$ be the memory usage of the schedule at time t. By definition:

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

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

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

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

$$M_{\max} \le S \quad .$$

4.3 Discussion

The above problem is a multi-criteria problem as memory usage and makespan are conflicting objectives. Let us take the DAG of Fig. 2(a). Tasks with a subscripted "+" allocate *one* memory slot (they are consumers) , tasks with a subscripted "-" release *one* slot (they are releasers) and task i_+ is paired with task i_-. Hence $\forall i \; \text{cost}(i_+) = +1$ and $\text{cost}(i_-) = -1$ and $i_- = \text{pair}(i_+)$. Moreover, the duration of all memory tasks is 0 and the duration of all the n other tasks (i.e. $t_1 \dots t_n$) is 1. In this case, if we schedule sequentially each 3-task thread we reach $M_{\max} = 1$ but $C_{\max} = n$, and if we parallelize on n resources we have $C_{\max} = 1$ but $M_{\max} = n$.

4.4 Motivating Example

Not all scheduling heuristics that 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 a deadlock is given in Fig. 2(b). Here again, tasks with a subscripted $+$ allocate one memory slot and tasks with a subscripted $-$ release one memory slot. Moreover, we can note that the number of machines is of no importance for the memory usage. Indeed, the memory is shared by the nodes and hence, the memory usage is only influenced by the order in which memory is allocated or released. Following the priorities shown in Fig. 2(b)'s table, on one processor, the scheduling sequence B_+, D_+ deadlocks if we have only two memory slots. Indeed, after executing B_+ and D_+, the only tasks that can be executed consume memory (A_+ or C_+). Therefore HEFT and SDC, whatever the number of available resources, will deadlock on this example. With two memory slots, a solution consists in executing the left part and the right part of the DAG one after the other: the sequence $A_+, B_+, E, H_-, I_-, C_+, D_+, G, J_-, K_-, F$ 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.

4.5 NP-hardness

It is well known that minimizing $C_{\max}$ alone is NP-hard, but minimizing $M_{\max}$ alone is NP-hard as well. The above problem is similar to the register allocation problem which is known to be NP-hard [7] by a reduction from graph coloring. However it is not trivially reducible to our problem. To show the NP-hardness of our problem we show that the associated decision problem is NP-complete with a reduction from the pebble game defined as follows.

The Pebble(K) problem

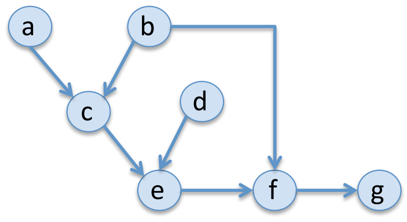
As input we have a DAG H and an infinite set of *labeled* pebbles. Pebbles will be put on the nodes of H. Let us define the game with the following allowed moves:

1. Pick a pebble from a node, if there is one.
2. If there are pebbles on every direct predecessor of a node x, then place a pebble on x (thus a node without predecessor can be pebbled at any time).

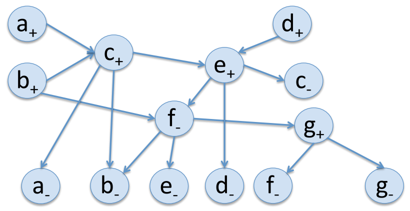
Labels can be sequential numbers used to count the number of pebbles put on H at any time of the game in constant time.

The goal of the game is, starting from a graph without any pebble, to find a sequence of moves such that every node is pebbled exactly once. In [23], Sethi shows that finding a sequence of moves using less than K pebbles is NP-complete[5]. More

[5] If a node can be pebbled more than once, then the problem is PSPACE-complete (and hence NP-Hard), but probably not in NP [10]



(a) A Pebble problem input graph example.



(b) Reduction of the Pebble input on the left to a $M_{\max}$ input.

Fig. 3 Input example for the Pebble to $M_{\max}$ problem.

precisely, the author proposes a third move that allows to slide a pebble from one predecessor of node v to v, if all predecessors of v are pebbled. However, it is proved in [10] that not allowing pebble sliding always increases by exactly one the number of required pebbles, whatever the input graph H. Hence both versions are NP-complete.

Minimizing M_{max} is NP-hard

First, we recall that, due to shared-memory model, the number of machines where the input graph G is scheduled is of no importance: only the order in which memory is allocated or released is relevant. In the example of Fig. 2(b), the one-machine schedule A_+, B_+, E, H_-, I_-, C_+, D_+, G, J_-, K_-, F has a memory usage of 2, while the schedule A_+, C_+, B_+, D_+, E, G, F, H_-, I_-, J_-, K_- has a memory usage of 4.

Second, we call $M_{\max}(M)$ the associated decision problem for minimizing $M_{\max}$: given an integer M and an input graph G, is there a one-machine schedule of the tasks such that $M_{\max} \leq M$? If we show that $M_{\max}(M)$ is NP-complete, it follows that minimizing $M_{\max}$ is NP-hard.

Theorem 1 *$M_{max}(M)$ is NP-complete.*

Proof Given an input of Pebble(K) we build an entry $G = (V, E)$ of $M_{\max}(K)$ as follows:

- For each node i of H, create a vertex i_+ and i_-. We put i_+ in V_+ and i_- in V_-.
- The set of memory nodes is composed of the nodes in V_+ and V_- only: $V_M = V_+ \cup V_-$.
- We pair these memory nodes: $i_- = \text{pair}(i_+)$.
- Costs are unitary: $\text{cost}(i_+) = \text{cost}(i_-) = 1$.
- We only have memory nodes: $V = V_M$.
- For each edge (i, j) in H, we build an edge $e_{i+j+} = (i_+, j_+)$ and an edge $e_{j+i-} = (j_+, i_-)$. We add e_{i+j+} and e_{j+i-} to E.
- If i has no successors in H, we build an edge $e_i = (i_+, i_-)$ and add e_i to E.

It is clear that this reduction is polynomial in the size of H.

In Fig. 3(b), we show how the input of Fig. 3(a) is reduced to an input of $M_{\max}$. For instance, the edge (a, c) is transformed into two edges: (a_+, c_+) and (c_+, a_-). As g has no successor we have only one edge (g_+, g_-).

Any solution σ_G of $M_{\max}(K)$ is a total order $(v_1,\ldots,v_n)$ of the vertices of G that respects the precedence constraints. From such a solution, we build a solution of Pebble(K). We consider the vertices v_i according to the total order of σ_G (from v_1 to v_n). We have two cases:

1. if $v_i \in V_+$, it means that according to the reduction, it has the form i_+: we place a pebble on node i of H;
2. if $v_i \in V_-$, it means that according to the reduction, it has the form i_-: we pick the pebble from node i.

Therefore, from these two cases, it follows that if $M_{\max} = K$ then the number of pebbles used in the solution of the Pebble game is K. Indeed, by definition, the memory usage of σ_G is the maximum of $M(t)$ which is equal to the number of vertices of V_+ minus the number of vertices of V_- executed at time t.

For example, the sequence $a_+,b_+,c_+,a_-,d_+,e_+,c_-,d_-,f_+,b_-,e_-,g_+,f_-,g_-$ respects the precedence constraints of G and uses 4 memory slots. It can be transformed in polynomial time in a solution that pebbles the graph of 3(a) using 4 pebbles: you pebble node i when you have "i_+" and you un-pebble it when you read "i_-".

Moreover, we obey all the rules of the Pebble(K) game (i.e. the solution is correct):

– all nodes will be pebbled exactly once because the nodes of V_+ are executed exactly once in the schedule of G;
– all nodes without predecessor can be pebbled at any time;
– if a node has a predecessor then its predecessors will be un-pebbled only after this node has been pebbled. Indeed, if i is a predecessor of j in H, i_- is a successor of j_+ in G and hence un-pebbling i (executing i_-) can be done only after pebbling j (executing j_+), because σ_G respects the topological order;
– it is correct to pick a pebble from a node i in H as this pebble has been placed before: indeed i_+ is always a predecessor of i_- in G.

From the above, it follows that if we can solve $M_{\max}(K)$ in polynomial time then we can solve Pebble(K) in polynomial time. $\square$

5 Solution Description

We now describe our solution proposal, which mainly consists in modifying the priorities used in list-scheduling heuristics. We first introduce some definitions and propositions that will be used, then describe the priority adjustments that we propose. We also introduce a modification of insertion heuristics typically used in list scheduling, to cope with memory constraints, and eventually explain the self-time scheduling which will be used in experiments.

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

Following this definition, a memory set that 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 nine tasks located between them. Additionally, Figure 9 shows a more complex graph with several memory clusters. In the remainder, we will only consider memory clusters.

Definition 2 Given two memory clusters A and B, A is an *ancestor* of B if there is a directed path from some node v_A in A to a node v_B in B.

Definition 3 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 that 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.[6]

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

Proof First, let us consider two disconnected clusters A and B, i.e. such that there is no path between nodes of A and nodes of B. Let $P : V \to \mathbb{N}$ that maps a node onto its priority. Then Condition C1 guarantees that:

$$\forall (v_A, v_B) \in A \times B, \begin{cases} \exists (v'_A, v'_B) \in A \times B, P(v'_A) > P(v'_B) \implies P(v_A) > P(v_B) \\ \exists (v'_A, v'_B) \in A \times B, P(v'_A) < P(v'_B) \implies P(v_A) < P(v_B) \end{cases} .$$

In terms of schedule, it means that if a consumer from A (resp. B) is scheduled first then all consumers from A (resp. B) will be scheduled before those from B (resp. A), 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+, or the converse, and thus the whole cluster will be schedulable.

Now assume that some node in B has an input dependency from a node in A, which makes A an ancestor of B. Let A^C (resp. B^C) denote the set of nodes from A (resp. B) that consume memory. Then Condition 2 demands that:

$$\forall (v_A, v_B) \in A^C \times B^C, P(v_A) > P(v_B) .$$

Thus consumers from A will be scheduled first. As a result, 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:

[6] As a reminder, the bigger its priority value, the earlier a task is scheduled.

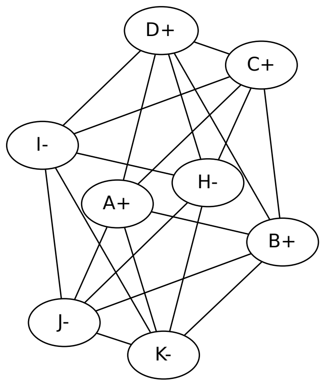


Fig. 4 Independence graph corresponding to Fig. 2(b)'s DAG.

Definition 4 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 memory-constraint 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.

Figure 4 illustrates what an independence graph looks like based upon Fig. 2(b)'s DAG. Let the following memory tasks form consumer-releaser pairs: (A_+, H_-), (B_+, I_-), (C_+, J_-) and (D_+, K_-). The original task graph features two memory clusters: $C_0 = \{A_+, B_+, E, H_-, I_-\}$ and $C_1 = \{C_+, D_+, G, J_-, K_-\}$. In C_0, there exist paths both from A_+ and B_+ to both H_- and I_-. Similarly, in C_1, both J_- and K_- are reachable from both C_+ and D_+. All other memory nodes are disconnected in the DAG, and thus adjacent in the independence graph.

5.1 Priority Adjustment

We now introduce an adjustment of priorities for memory constraints, which can be applied to static-priority–based list-scheduling algorithms. It is assumed that original priorities have been computed using any such pre-existing heuristics.

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

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

	Original priority	Bonus	Adjusted priority
A_+	8	8	16
B_+	12	8	20
C_+	8	8	16
D_+	12	8	20
E	6	8	14
F	1	0	1
G	6	8	14
H_-	1	0	1
I_-	2	8	10
J_-	1	0	1
K_-	2	8	10

Table 1 Priorities before and after adjustment in Fig. 2(b).

This formal framework can be thought of more intuitively in terms of lifetimes.

Definition 5 A *lifetime of memory* (or just *lifetime*) is a portion of schedule spanning from the start time of a consumer until the end time of its paired releaser.

The rationale behind adjusting priorities is thus to prevent lifetimes from overlapping so as to limit the overall memory footprint.

To illustrate these requirements, we consider the independence graph depicted in Fig. 4 and the original priorities mentioned in Table 1. Let us suppose H_- is candidate for priority adjustment since it is a releaser. The following tasks are adjacent to H_- in the independence graph and thus satisfy Requirement 1: C_+, D_+ I_-, J_- and K_-. Among them, only consumers fulfill Requirement 2: $C+$ and $D+$. Then, $P(C_+) = P(A_+)$ and $P(D_+) > P(A_+)$, and A_+ is adjacent to both J_- and K_- in the independence graph, so H_- will not get any priority bonus. Similarly, if I_- is considered for priority adjustment, both C_+ and D_+ satisfy Requirements 1 and 2. On the other hand, $P(C_+) < P(B_+)$ so C_+ also meets Requirement 3a (but not 3b). As a result, I_- will get a bonus equal to $P(C_+) = 8$. Priority adjustment for the other two releasers can be derived through analogous reasoning. After propagating the bonuses to the whole graph, we come up with the new priorities shown in Table 1.

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. More specifically, Requirement 3a tends to prevent memory lifetimes from overlapping by getting bonuses from lower-ranked consumers to releasers in clusters with higher-ranked consumers, but it sometimes happens to be insufficient as shown in Section 5.2; and Requirement 3b means that there is a path in the original task graph from a consumer in the cluster getting the bonus to a releaser in the cluster giving the bonus, so as to ensure that upstream tasks always have higher priorities. These adjusted priorities are then propagated to the rest of the DAG through a second pass of the regular task-prioritizing phase.

5.2 Priority forcing

In some few cases, this priority-adjustment scheme is not sufficient to meet Condition C1. Figure 2(b) gives an example of such situation. The graph comprises two disconnected memory clusters C_0 and C_1, and the ALB is 2. As explained in Section 5.1, I_-'s priority will get a bonus from C_+ and K_-'s a bonus from A_+, resulting into the adjusted priorities shown in Table 1. Then, A has the same adjusted priority as C_+, and B_+ the same as D_+. Therefore B_+ and D_+ will be scheduled before A_+ and C_+ which would cause the scheduler to use at least 3 memory slots instead of 2 available. Hence the need to force the priorities such that A_+ and B_+ will be together scheduled before C_+ and D_+: the smallest priority of one of these clusters must be greater than the largest priority of the other cluster..

To ensure priority forcing, we use Algorithm 1 that enforces condition 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 $P(T) \leq P(T')$ do
            // Raise priority of task T
            $P(T) \leftarrow P(T') + 1$
        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. For instance, in Fig. 2(b)'s example, since the two memory clusters have overlapping lifetimes, the priorities of either A_+ and B_+ or C_+ and D_+ have to be forced. Let us suppose that, due to the topological order, C_0 is the last cluster to be traversed by the algorithm, and thus A_+ and B_+ priorities will be shifted. As the highest adjusted priority of C_1 is $P(D_+) = 20$, A and B will then both have their priorities raised to $20 + 1 = 21$. Therefore, while HEFT and SDC will deadlock with two memory slots with the original priorities, the proposed mechanism ensures that these new priorities enable a deadlock-free schedule.

Lastly, attracting though this algorithm may seem, due to its simplicity and the guarantees it brings, it shall be used only as a last resort and in combination with the priority-adjustment scheme detailed in Section 5.1. In fact, alone it is not sufficient to achieve Condition C2 and thus may not prevent all deadlocks. Furthermore, applied without prior adjustment, it results in a priority packing, which is harmful to the

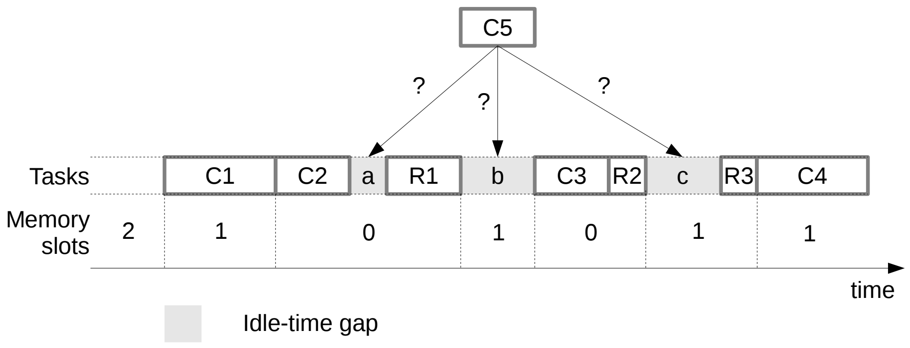


Fig. 5 Attempt to insert a consumer task in an idle-time gap. Only memory tasks are represented: C1 to C5 embody consumers, R1 to R3 stand for releasers. Gaps are denoted *a*, *b* and *c*.

schedule quality in terms of makespan, since it cuts pipelining, even though it does not break memory constraints. These two points will be illustrated in Section 6.2's experiments.

5.3 Insertion-based Policy

Many scheduling heuristics (e.g. HEFT, SDC) provide insertion mechanisms to schedule tasks in idle-time *gaps*. 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 gaps at time t. For all $i \in I(t)$, we define $\text{start}(i)$ the start time of i and $\text{end}(i)$ the end time of i, then we derive the duration of i: $d(i) = \text{end}(i) - \text{start}(i)$. Let $\text{EST}(v)$ and $\tilde{d}(v)$ denote the estimated start time and the duration of task v, respectively. 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 gap has sufficient duration:

$$d(i) \geq \tilde{d}(v_c) \ \ ;$$

- there is enough memory at the insertion point:

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

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

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

Figure 5 exemplifies how insertion works when memory tasks are involved. Suppose that consumer C5 is considered for insertion and three idle-time gaps are candidates for accommodating it. Requirement 1 states that C5 does not fit into gap *a* since $d(a) < \tilde{d}(C5)$. Requirement 2 allows C5 to be inserted either into gap *b* or *c* because both have one memory slot available. As per Requirement 3, inserting a task shall never make the number of available memory slots negative. To enforce this requirement, available memory slots after the insertion point have to be recomputed. In this example, inserting C5 into gap *b* would prevent C3's execution due to lack of memory, so this is not allowed. Finally, the only option is to schedule C5 in gap *c*.

5.4 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 $w_{i,j}$ 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 [18] as only the allocation and the order do 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.

The resort to self-timed scheduling will be justified in Section 6.1.

6 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 3.3 and the H.264 video coding algorithm [29]. As real hardware was not available, we only simulated schedules, as described in Section 5.4, without effectively running them.

All our experiments consisted in comparing the makespans of schedules resulting from our priority-adjustment technique against their unadjusted versions; 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. First, we set the reference duration w^r of each actor as follows:

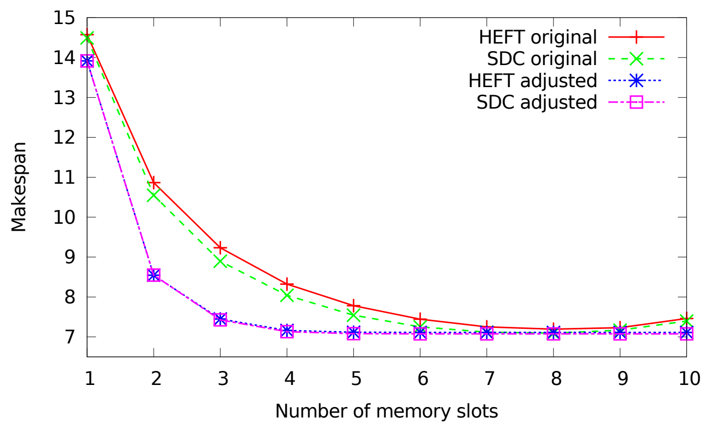


Fig. 6 TNR with 10 lines of 1000 pixels

   (a) For each type of actor (`src`, `fading`, etc.), we define a unitary duration per number of pixels.
   (b) 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).
2. Then we create different task-graph instances where task durations vary around this reference value.
   (a) 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 whose average is 0.5 when $\alpha = \beta$. Here, we use $(\alpha, \beta) = (2,2)$:
$$\bar{w} = w^r \left(\text{Beta}(2,2)(a - 1/a) + 1/a\right) .$$
   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.
   (b) The final duration of each task instance is computed similarly with the same dispersion factor a.

6.1 TNR

First, 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;

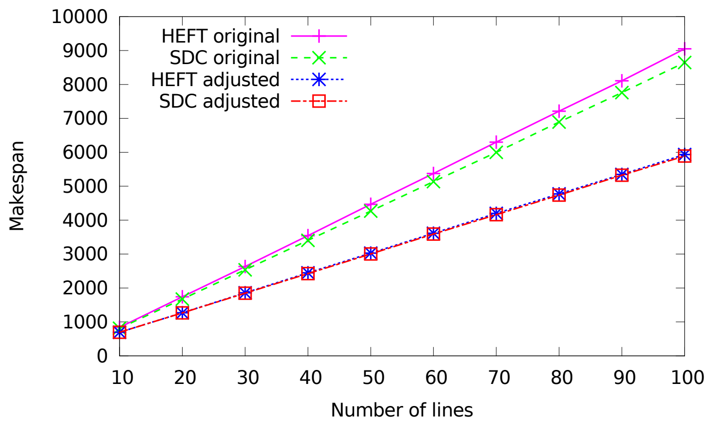


Fig. 7 TNR with 2 memory slots

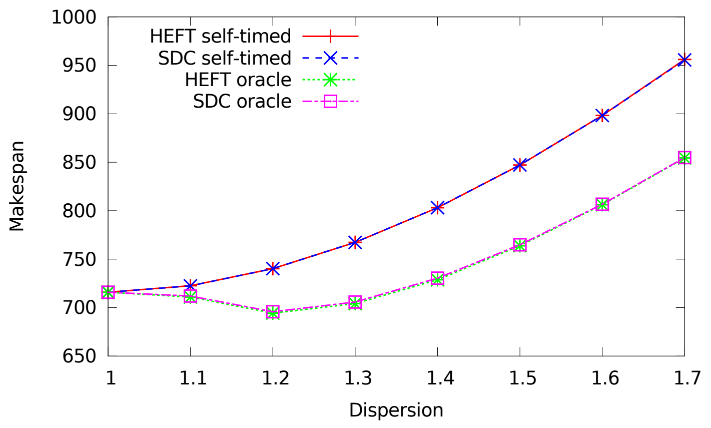


Fig. 8 TNR with 10 lines of 1000 pixels

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.

In a second row of experiments, the number of memory slots is fixed to 2 and the number of input pixel lines ranges from 10 to 100, while all other parameters remain unchanged. This allows to assess the impact of the application size on the makespan, as depicted by Figure 7. The results show that the makespan increases linearly with the number of pixel lines and that the slope is about 20% lower in the adjusted case, as expected, for both HEFT and SDC.

Finally, we present a methodology aimed at assessing the benefits of self-timed scheduling, i.e. whether or not the ordering of tasks on each processing element should be left as a run-time decision. To this end, three kinds of schedules are considered:

- a *reference* schedule, fully static, using reference task durations as described above[7];
- a *self-timed* schedule defined as described in Section 5.4, which models the execution of an application such that:
  - exact task durations are only known at run time,
  - mean task durations are known at compile time;
- an "*oracle*" schedule knowing all exact task durations at compile time.

The difference of makespans between the last two schedules allows to measure the potential gain brought by a partially dynamic in the case of a real execution. So the oracle can be seen as a lower bound for the makespan.

This methodology was applied to the TNR algorithms with the same parameters as above to compare makespans obtained through either a selft-timed or an oracle schedule against the dispersion factor. Figure 8 illustrates the results. It can be observed that the gap widens when task durations vary more, until reaching 10.5%. This is consistent with the oracle's ability to make up for these variations by reassigning a time-consuming task to a different processing element, or by leveraging idle-time slots to insert it. In this case, the conclusion would be that it is worth considering a dynamic adjustment of the schedule if the variations observed with the application correspond to a dispersion factor greater than 1.2.

6.2 H.264

We used a simplified model of an H.264 decoder illustrated by Fig. 9. 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[8] 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, MBs are not cached, 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

[7] The sole purpose of this reference schedule is to serve as a basis to construct the self-timed schedule. Thus, the makespans resulting from it are not meaningful for our study and, as such, are not represented.

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

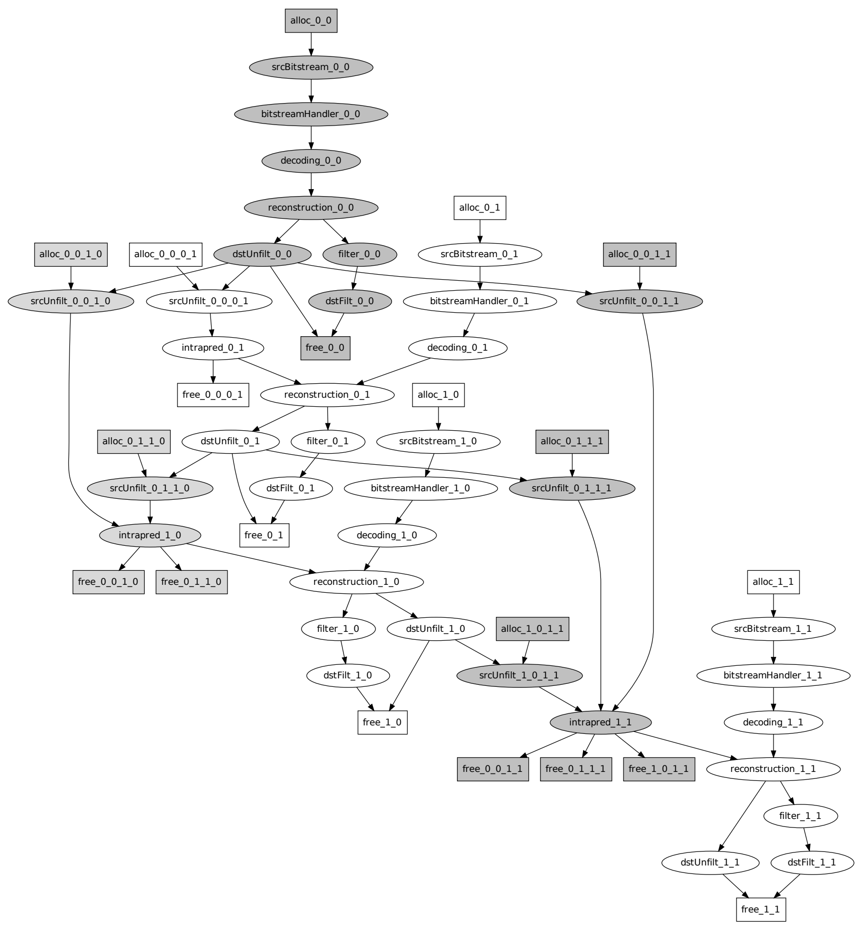
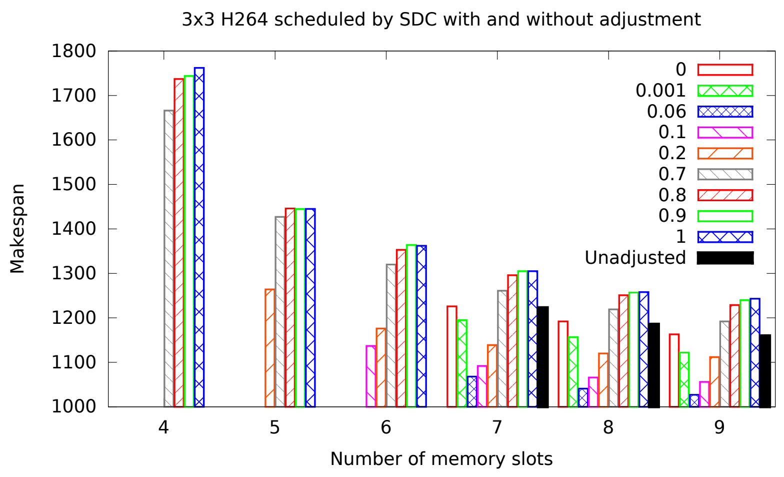


Fig. 9 H.264 task graph for 4 dependent macroblocks. 3 out of 7 memory clusters are shown in different shades of grey. Allocator and releaser tasks appear in square boxes.

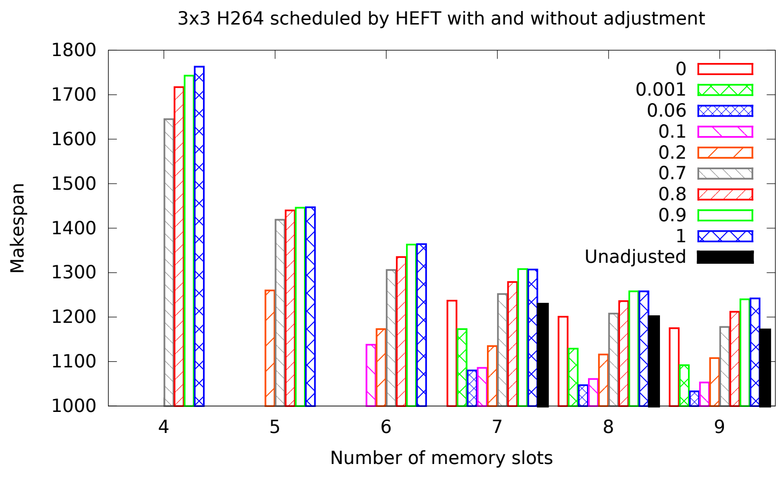
is actually 4. The simulated platform is composed of 1 DMA, 1 SWPE, and 4 HWPE (one per accelerated computation actor).

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* $BF \in [0,1]$ 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 10 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



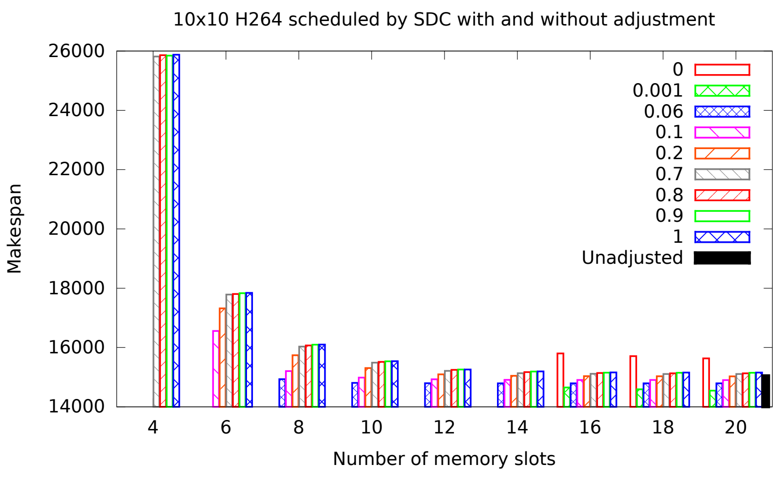
(a) SDC heuristics with different bonus factors



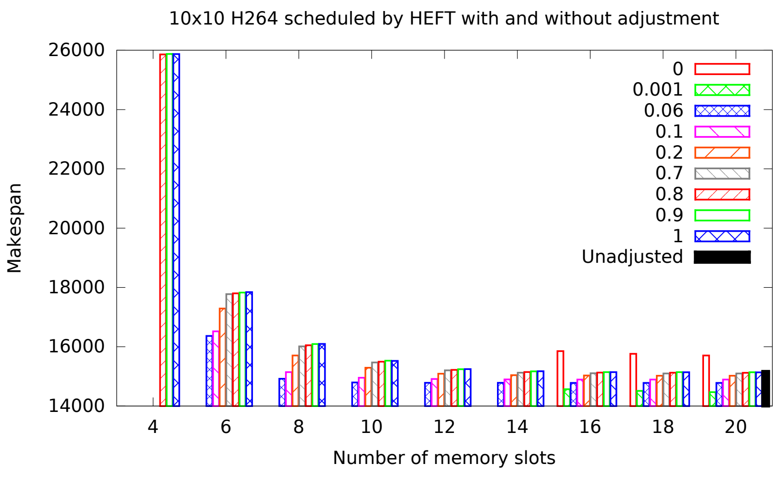
(b) HEFT heuristics with different bonus factors

Fig. 10 H.264 with 3x3 macroblocks. The missing bars mean that the version of the heuristics produces a schedule that deadlocks.

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. In general, for bonus factor lower than 1, condition C2 is not systematically met, hence the absence of solution for lower memory-slot numbers. 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



(a) SDC heuristics with different bonus factors



(b) HEFT heuristics with different bonus factors

Fig. 11 H.264 with 10x10 macroblocks. The missing bars mean that the version of the heuristics produces a schedule that deadlocks.

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. 10(a) with 10(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.

Bonus factor	0	0.001	0.06	0.1	0.2	0.7	0.8	0.9	1
Number of schedules with forced priorities	5454	5476	3048	1491	508	702	996	0	2

Table 2 Resort to priority forcing against bonus factor among 10,000 schedules.

Table 2 indicates, for each bonus-factor value, the total number of schedules that required priority forcing, over 10000 samplings. The results for higher BF values (0 and 2) confirm the overall quality of the priority adjustment rendered even before forcing comes into play. The measure when BD tends toward 0 only considers when the memory capacity is to sufficient to avoid a deadlock.

In the second set of simulations, the schedulers were fed with a DAG describing the processing of 10 lines of 10 MBs (10x10). Figure 11 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.

7 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 have also shown that unmodified heuristics are not able to provide schedules without deadlocks when memory requirements are important. Only a strong priority adjustment prevents deadlocks. 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. As shown in Fig. 8, dynamic scheduling can be beneficial when the dispersion of the duration is important. Hence, 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. Adam, T.L., Chandy, K., Dickson, J.: Comparison of list schedules for parallel processing systems. Communications of the ACM **17**(12), 685–690 (1974)

2. Baker, T.P.: Stack-based scheduling for realtime processes. Real-Time Syst. **3**(1) (1991)
3. Batat, A., Feitelson, D.: Gang scheduling with memory considerations. In: Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International, pp. 109–114. IEEE (2000)
4. Benini, L., Flamand, E., Fuin, D., Melpignano, D.: P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2012, pp. 983–987 (2012)
5. Buck, J., Lee, E.: 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, pp. 429–432. IEEE (1993)
6. Canon, L.C., Jeannot, E., Sakellariou, R., Zheng, W.: Comparative evaluation of the robustness of dag scheduling heuristics. In: S. Gorlatch, P. Fragopoulou, T. Priol (eds.) Grid Computing, pp. 73–84. Springer US (2008). DOI 10.1007/978-0-387-09457-1_7
7. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. Computer Languages **6**, 47–57 (1981)
8. Fradet, P., Girault, A., Poplavkoy, P.: SPDF: a schedulable parametric data-flow MoC. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012, p. 769774 (2012)
9. Geng, T., 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 **E93-D**(12), 3223–3231 (2010)
10. Gilbert, J., Lengauer, T., Tarjan, R.: The pebbling problem is complete in polynomial space. SIAM Journal on Computing **9**(3), 513–524 (1980)
11. Guermouche, A., L'Excellent, J.Y.: Memory-based scheduling for a parallel multifrontal solver. In: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, p. 71. IEEE (2004)
12. Herrmann, J., Marchal, L., Robert, Y.: Model and complexity results for tree traversals on hybrid platforms. In: Euro-Par 2013 Parallel Processing, pp. 647–658. Springer (2013)
13. Herrmann, J., Marchal, L., Robert, Y.: Memory-aware list scheduling for hybrid platforms. Rapport de recherche RR-8461, INRIA (2014). URL `http://hal.inria.fr/hal-00944336`
14. Jian, G.A., Chu, J.C., Huang, T.Y., Chang, T.C., Guo, J.I.: A system architecture exploration on the configurable hw/sw co-design for h.264 video decoder. In: Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on, pp. 2237–2240 (2009). DOI 10.1109/ISCAS.2009.5118243
15. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv. **31**(4), 406–471 (1999). DOI 10.1145/344588.344618
16. Lawler, E.L., Lenstra, J.K., Kan, A.R., Shmoys, D.B.: Sequencing and scheduling: Algorithms and complexity. Handbooks in operations research and management science **4**, 445–522 (1993)
17. Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE **75**(9), 1235–1245 (1987)
18. Lee, E.A., Ha, S.: Scheduling strategies for multiprocessor real-time dsp. IEEE Global Telecommunications inproceedings and Exhibition **2** (1989)
19. Liu, J.W.: On the storage requirement in the out-of-core multifrontal method for sparse factorization. ACM Transactions on Mathematical Software (TOMS) **12**(3), 249–264 (1986)
20. Marchal, L., Sinnen, O., Vivien, F.: Scheduling tree-shaped task graphs to minimize memory and makespan. In: Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, pp. 839–850. IEEE (2013)
21. Melpignano, D., Benini, L., Flamand, E., Jego, B., Lepley, T., Haugou, G., Clermidy, F., Dutoit, D.: Platform 2012, a many-core computing accelerator for embedded socs: Performance evaluation of visual analytics applications. In: Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE, pp. 1137–1142 (2012)
22. Saponara, S., et al.: Performance and complexity co-evaluation of the advanced video coding standard for cost-effective multimedia communications. Eurasip Journal on Applied Signal Processing **2004**(2), 220–235 (2004)
23. Sethi, R.: Complete register allocation problems. SIAM journal on Computing **4**(3), 226–248 (1975)
24. Shi, Z., Dongarra, J.J.: Scheduling workflow applications on processors with different capabilities. Future Generation Computer Systems **22**(6), 665 – 675 (2006). DOI 10.1016/j.future.2005.11.002
25. Sih, G.C., Lee, E.A.: Compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. IEEE Transactions on Parallel and Distributed Systems **4**(2), 175–187 (1993)
26. Sullivan, G., Ohm, J.R.: Recent developments in standardization of high efficiency video coding (hevc). Proceedings of SPIE - The International Society for Optical Engineering **7798** (2010)

27. Topcuoglu, H., Hariri, S., Wu, M.Y.: Task scheduling algorithms for heterogeneous processors. In: 8th IEEE Heterogeneous Computing Workshop (HCW'99), pp. 3–14. San Juan, Puerto Rico (1999)
28. Wang, S.H., 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 **41**(1), 93–110 (2005)
29. Wiegand, T., et al.: Overview of the h.264/avc video coding standard. IEEE Transactions on Circuits and Systems for Video Technology **13**(7), 560–576 (2003)