Scheduling DAGs of Multi-version Multi-phase Tasks on Heterogeneous Real-time Systems

Julius Roeder
University of Amsterdam
Amsterdam, The Netherlands
j.roeder@uva.nl

Benjamin Rouxel
University of Amsterdam
Amsterdam, The Netherlands
b.rouxel@uva.nl

Clemens Grelck
University of Amsterdam
Amsterdam, The Netherlands
c.grelck@uva.nl

***Abstract*—Heterogeneous high performance embedded systems are increasingly used in industry. Nowadays, these platforms embed accelerator-style components, such as GPUs, alongside different CPU cores. We use multiple alternatives/versions/implementations of tasks to fully benefit from the heterogeneous capacities of such platforms and due to binary incompatibility. Implementations targeting accelerators not only require access to the accelerator but also to a CPU core for, e.g., pre-processing and branching the control flow. Hence, accelerator workloads can naturally be divided into multiple phases (e.g. CPU, GPU, CPU). We propose an asynchronous scheduling approach that utilises multiple phases and thereby enables a fine-grained scheduling of tasks that require two types of hardware. We show that our approach can increase the schedulability rate by up 24% over two multi-version phase-unaware schedulers. Additionally, we demonstrate that the schedulability rate of our heuristic is close to the optimal schedulability rate.**

I. Introduction

The survey by Akesson et al. [1] shows that industry increasingly makes use of heterogeneous high performance embedded systems. Companies not only use platforms with heterogeneous CPUs (e.g. ARM big.LITTLE architecture) but also employ systems with accelerator-style components such as GPU or FPGA (e.g. Nvidia Jetson, Odroid-XU4, Odroid-N2+). To increase the overall performance, taking full advantage of the available heterogeneity is crucial. Different binary incompatible compute units can be targeted by multiple alternatives/versions/implementations of a task [2]–[4]. Selecting the best task version depends on the schedule history and on the available compute units. Therefore, we propose a multi-version-aware scheduling approach which benefits from more scheduling opportunities and thus creates more performant schedules.

Unlike CPUs, accelerators like GPUs require an external action to branch the application control flow to them, i.e. a CPU core is needed to prepare data and launch the GPU kernel. Hence, a task targeting a GPU has three natural phases: 1) a CPU phase that hands control to the GPU, 2) a GPU phase that does some computations and, 3) a CPU phase that handles post-processing and resuming of the application. A common scheduling solution to multiple phases is to consider the three phases as three individual tasks and schedule them separately. However, when tasks have different versions and each has multiple phases, splitting multi-phase tasks into multiple tasks means that the scheduler may create a sequence of phases from different task versions which would be erroneous.

As an example let us consider two independent tasks that are executed on an architecture with one CPU core and one GPU. *Task 1* starts with a CPU phase, followed by a GPU phase and finalises with a CPU phase. The Worst-Case Execution Time (WCET) of *Task 1* is 6 time units including all phases in sequence. *Task 2* executes on the CPU only and has a WCET of 4 time units. Following a synchronous scheduling approach which does not consider the different phases and schedules the full task in one contiguous sequential block results, as illustrated in Figure 1, in a 10 time units makespan. When scheduling the task in one block, the CPU is stalled

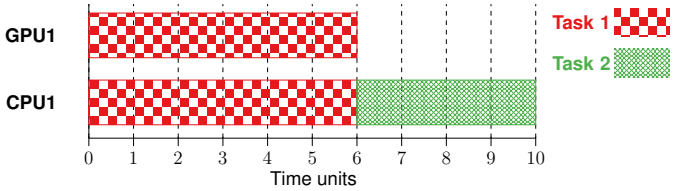

Fig. 1: Motivational example without phases.

while executing the GPU phase. By splitting a task into different phases we enable the asynchronous use of computing resources and schedule phases independently. Our approach is, thus, aware of this phase multiplexing and can rely on it to improve the schedulability rate for Directed Acyclic Graphs (DAG) with high utilisation, as shown in Figure 2 with a makespan of 6.

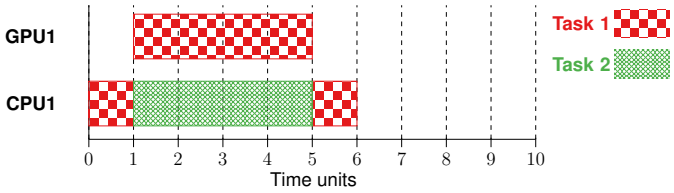

Fig. 2: Motivational example with phases.

We propose a new task model that splits tasks into a finer hierarchy of elements by adding phases, i.e. a version can be split into phases. Phases enable more mapping and scheduling opportunities and increase the hardware utilisation.

In addition we introduce an offline scheduling heuristic to map and schedule a set of tasks onto heterogeneous computing units. Each task has at least one version and each version has at least one phase. The static scheduler chooses which implementation variant of a task is used. The proposed strategy is formulated as a heuristic based on Forward List Scheduling (FLS) to produce schedules fast. According to Davis and Burns [5], our heuristic can be classified as static, partitioned and time-triggered. Additionally, we support fixed preemption points in each task.

The contributions of this work are the following:

- We define a novel task model which splits tasks into versions and phases.
- We propose an offline scheduling heuristic that maps and schedules tasks represented by versions and phases.
- We demonstrate that our method significantly improves the schedulability by up to 11% and 24% in comparison to two multi-version phase-unaware schedulers [3], [6].
- Rouxel et al. [7] showed that no sorting algorithms for heuristics outperforms the others. We empirically show that in our case a HEFT ranking or Breadth-First-Search (BFS) approach is most successful.

This paper builds upon our previous scheduling approach introduced in [3] where we tackled energy-aware scheduling for heterogeneous systems, mainly focusing on Dynamic Voltage and Frequency Scaling (DVFS) and energy modelling. Additionally, the paper followed a synchronous approach where any task requiring the GPU reserved both a CPU core and the GPU for the full length of the task, leading to a CPU core being blocked while using the GPU and vice versa.

II. System Model

We consider an application represented by a Directed Acyclic Graph (DAG), henceforth called task graph. A graph is a tuple $G = (D, \tau, E)$, where D is the deadline, τ represents the set of nodes/vertices, hereafter called tasks, and the set of edges E represents data dependencies between tasks. Hence, a producer task needs to be completed before the corresponding consumer task may start executing. Each task consists of a (non-empty) set of versions V.

The different versions of a task are functionally equivalent (i.e. they implement the same input/output relation), but differ in their structure, their compute units usage, hence in their non-functional properties (energy, time, etc.). Different versions can be the result of: 1) targeting different functional units; 2) different algorithms or implementation variants [3]; 3) using varying compilation flags to, for example, optimise code for energy consumption, binary size, speed or architecture features [8]. Furthermore, each version is split into multiple phases. A phase is a sequential block of code that is executed on one compute unit. Each phase is characterised by a tuple $p = (C, U)$, where C is the Worst-Case Execution Time (WCET) and U is the set of compute units on which it can be executed and on which the WCET holds.

Figure 3 shows a task graph and illustrates different possibilities that our task model enables. The task graph consists of 4 tasks. The first two tasks (*getSensor* and *getImage*) both have one version, which can be executed on either a LITTLE or a big core. The next task (*SobelFilter*) has a total of 4 different versions. Version 1 can only be executed on a big core. Version 2 consists of three phases, where the first phase targets a CPU core (big or LITTLE), followed by a GPU phase and finalised with a CPU phase (big or LITTLE). Versions 3 and 4 have 2 phases each, where the first phase is executed on a CPU and the second phase is executed on either a GPU (V3) or a different accelerator (V4). The *EdgesDetect* task has 3 versions. Lastly, the *UpdateResult* task shows two different versions where the first one is executed on the big core and the second one is executed on the LITTLE core.

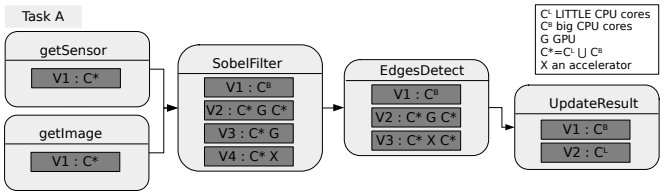

Fig. 3: Example of a task graph.

We target heterogeneous platforms that embed any type of computing units including different types of CPU (e.g. big.LITTLE architectures), GPU, DSP, FPGA, and other accelerators. Our approach is fully platform-independent and can be applied to a wide range of heterogeneous (embedded) system architectures.

III. Interference

A. Cache related delays

Our approach allows for migration. For example, two different CPU phases of the same version can be executed on different CPU cores. We do not allow migration during execution, i.e. when a phase starts on a compute unit, it cannot migrate. Allowing *migration* of phases from the same task induces a non-negligible cost due to cache reloads. If two CPU phases of the same task-version share cache lines spawning them on different cores affects the execution time of the second phase. Figure 4 shows an example of migration where the red part of the last phase of task SF-v2 needs to reload cache lines (instruction or data) that would have been present if executed on the same CPU core as the first phase.

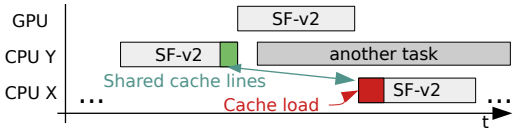

Fig. 4: Example of migration.

Similarly, allowing asynchronous execution between phases has the same impact on cache as migration, depicted in Figure 5. Inserting a phase from another task between two phases of the same task (*entanglement*) can evict shared cache lines (both instruction or data). There are no shared cache lines

between different compute unit types (e.g. CPU and GPU) and thus we assume an empty cache.

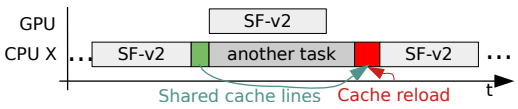

Fig. 5: Example of entanglement.

This interference cost is well-known for preemption and is called *Cache Related Preemption Delay* (CRPD), e.g. [9]. Allowing phases from other tasks to execute in-between two phases of a task is similar to a preemption mechanism. Our scheduling strategy accounts for the CRPD by adding an extra cost to the WCET of a second phase that suffers from it. Computing the CRPD itself can be performed using many techniques, both static analysis [9] and dynamic analysis [10]. Our scheduling method is not linked to any particular CRPD analysis. We only require that an extra cost is added to the WCET in case of migration or entanglement.

B. Shared resources interference

Scheduling tasks for multi-core platforms requires accounting for hardware interference due to shared resources, e.g. bus, memory, etc. One possibility is to isolate the parts of the code that generate interference as phases and ensure that only one of these phases across all tasks can execute at the same time. This approach would be similar to AER [11] or PREM [12]. However, these methods are not suitable for our targeted applications and platforms (Odroid-XU4, Jetson TX2 etc.). The applications we consider (e.g. object detection) often have no predetermined load-compute-write structure as the data might not fit into cache. Therefore, we follow a more portable path and add the interference cost into the WCET to form the Worst Case Response Time (WCRT) of each phase and then schedule phases according to the WCRT [13].

C. Data in memory interference

All data transfer between two compute units (e.g. CPU to GPU) is accounted for in the CPU phases. Thus, we can target both shared memory and dedicated accelerator platforms. We consider memory management outside of this paper. In other words, we assume sufficient memory to be available.

IV. Heterogeneous FLS

Our proposed heuristic is based on Forward List Scheduling (FLS). FLS first orders the tasks, then adds them one by one to the schedule without backtracking. We use three sorting algorithms: Depth First Search (DFS), Breadth First Search (BFS) and HEFT ranking [6]. For DFS and BFS, we use the task WCET as a tie breaking rule (larger WCET to be scheduled first). Furthermore, we introduce one additional tie breaking rule for BFS based on *laxity*. Since it is shown in [7] that no sorting algorithm consistently outperforms the others, we generate four schedules, each resulting from one sorting strategy / tie breaking rule combination, and select the one resulting in the lowest makespan as our heuristic solution. From here on-wards we will refer to our heterogeneous, multi-version task scheduling heuristic as *hFLS*.

Our approach: We schedule all phases of a (task) version one after another. This way we allow for migration between CPU cores or the CPU and an accelerator or even between accelerators. When scheduling a phase, the heuristic checks if there is either migration or entanglement between the current phase and the previous phases of the same task and increases the WCET with the CRPD of the phase. Scheduling decisions are not changed retrospectively.

A. Scheduling Algorithm

Our proposed heuristic is sketched out in Algorithm IV.1. It takes as an input the DAG of the application represented by tasks τ, a set of compute units CU and a deadline D^C. It sorts the tasks and creates a list (Line 2) then it loops over all the tasks in the list (Lines 5 – 37). We try all versions of a task (Lines 8 – 34), where each version starts with a clean schedule (Line 9). All phases of a version are scheduled (Lines 10 – 32) and all compute units CU are tried (Lines 14 – 30).

The scheduler ensures that a phase ph can execute on a given CU (Lines 15 – 16). Then we compute the Release Time (RT) of ph (Line 18). We evaluate if the RT and mapping suffer from either migration or entanglement interference and update the WCET of ph (Line 19). If the specific RT or mapping results in interference, the RT is recalculated (Line 21) with the new WCET. Section IV-B explores the RT calculations. Section IV-C explores the WCET update. After the RT and WCET updating steps ph is added to the schedule (Line 22).

Adding phase ph to the schedule could mean that a previously scheduled phase of a different task now suffers from interference. Thus, Line 24 finds and propagates such interference through the schedule. The interference propagation algorithm is explored in Section IV-D.

The best core mapping for a phase does not only depend on the $makespan$ but also on the release time. An earlier release time of the first phase could lead to a better $makespan$ for the latter phases (Lines 27). The best core for a phase is saved in $bestCoreSchedule$ (Line 28). Once the best core for a phase has been found the $versionSchedule$ is updated (Line 31), which is the base for scheduling the next phase.

After all phases have been scheduled $versionSchedule$ contains the best schedule for a task version. The best version of a task is then saved in the $tmpSchedule$ (Line 34). Once all versions of a task have been tried, the original $schedule$ is updated (Line 35) and the $Qdone$ list is extended (Line 37).

After all tasks in the $Qready$ list have been scheduled we check if the resulting schedule has a $makespan$ lower than the deadline D^C (Line 38).

B. Phase Release Time

Algorithm IV.2 sketches out the release time ρ computation of a phase. The input is the list of scheduled tasks $Qdone$, current task $curTask$, phase ph and target processor $curCU$. ρ has to be after the predecessors of the task it belongs to (Lines 3 – 4). Additionally, ph has to be released after the

ALGORITHM IV.1: Scheduling Algorithm

```
Input  : A DAG composed of multi-version tasks (τ), a set
         of computational units (CU) and a deadline (D^C).
Output : A schedule.
1  Function ListSchedule(τ_x ∈ τ|τ_x = v, CU, D^C)
2      Qready ← TopologicalSortTasks(τ)
3      Qdone ← []
4      schedule ← new Schedule()
5      while t ← Qready.popFront() do
6          tmpSched ← schedule
7          tmpSched.makespan ← ∞
8          foreach v ∈ t.versions do
9              versionSchedule ← schedule
10             foreach ph ∈ v.phases do
11                 bestCoreSchedule ← versionSchedule
12                 bestCoreSchedule.makespan ← ∞
13                 oldReleaseTime ← ∞
14                 foreach u ∈ CU do
15                     if ph does not runs on u then
16                         continue
17                     copy ← versionSchedule
18                     copy.phaseRT(Qdone, t, v, ph, u)
19                     copy.updateWCET(Qdone, t, v, ph, u)
20                     if ph.appliedCRPD() then
21                         copy.phaseRT(Qdone, t, v, ph, u)
22                     copy.addPhase(ph)
23                     ph.setTargetCU(u)
24                     copy.propInterferences(Qdone, t, v, ph,
                       u)
25                     copy.updateMakespan()
26                     newReleaseTime ←
                       copy.releaseTime(ph)
27                     if copy.makespan <
                       bestCoreSchedule.makespan ∨
                       (copy.makespan ≤
                       bestCoreSchedule.makespan ∧
                       newReleaseTime < oldReleaseTime )
                       then
28                         bestCoreSchedule ← copy
29                         bestCoreSchedule.updateMakespan()
30                         oldReleaseTime ←
                           newReleaseTime
31                 versionSchedule ← bestCoreSchedule
32                 versionSchedule.updateMakespan()
33             if versionSchedule.makespan <
               tmpSched.makespan then
34                 tmpSched ← versionSchedule
35         schedule ← tmpSchedule
36         schedule.updateMakespan()
37         Qdone.pushBack(t)
38     if schedule.makespan > D^C then
39         return unschedulable
40     return schedule
```

previous phase of the same task (Lines 5 – 6). We have to ensure that there is no overlap with previously scheduled tasks on the same compute unit $curCU$ (Lines 8 – 14). If there is an overlap ρ of ph is set to the previous tasks completion time (Line 13).

ALGORITHM IV.2: Phase release time

```
Input  : Scheduled tasks (Qdone), task to schedule
         (curTask), phase (ph), processor (curCU).
Output : Computes a phases release time.
1  Function phaseRT(Qdone, curTask, ph, curCU)
2      ρ ← 0
3      foreach x in curTask.predecessors do
4          ρ ← max(x.ρ + x.C, curTask.ρ)
5      y ← ph.previousPhase
6      ρ ← max(ρ, y.ρ + y.C)
7      change ← true
8      while change do
9          change ← false
10         foreach t ∈ Qdone do
11             if t is mapped on curCU then
12                 if t overlaps in time with ph then
13                     ph.ρ ← t.ρ + t.C
14                     change ← true
```

Algorithm IV.2 is guaranteed to terminate because we only postpone ph. In the worst case scenario ph is moved to the end of the schedule.

C. Updating Phase WCET

Algorithm IV.3 sketches out how the WCET of ph is updated in case of interference with respect to one of the previous phases. First, we iterate over all previous phases (Line 2 – 12) and check if ph has migrated with respect to a previous phase (Line 3). A *migration* occurs when the ph is executed on a different compute unit of the same type as the mapping of a previous phase. If migration occurred we add the CRPD to the phase WCET (Line 4) and break out of the loop (Line 5) because the CRPD should only be added once. If ph has not migrated, we still need to verify that there is no *entanglement* with respect to the previous phases (Lines 6 – 12). To do so we check if any of the other scheduled tasks (Line 7) is between ph and a previous phase (Line 10). If *entanglement* occurred we add the CRPD (Line 11) to the WCET and break out of both loops (Line 12).

D. Find and Propagate interference across the schedule

Algorithm IV.4 sketches out how the scheduling of the current phase propagates interference through the already scheduled tasks and phases. First, we iterate over all previously scheduled tasks (Lines 3 – 15). We only check for interference if ph is not part of t (Line 4). Then we iterate over the phases in t twice (Lines 5 and 8). At this point we only need to continue: 1) If the phases are both on $curCU$ (Lines 6 and 9). 2) If the $p1$&$p2$ are different and $p2$ is scheduled after $p1$ (Line 11). Then if ph is scheduled between $p1$ and $p2$ (Line 13), there is *entanglement* and $p2$ has been impacted. At this point we can break out of all three loops (Line 15) because only one previously scheduled phase can be impacted by ph. If a previously scheduled phase has been impacted (Line 16), the interference needs to be propagated through the rest of the already scheduled tasks/phases. This is done in a recursive

ALGORITHM IV.3: Updating the WCET of a phase

```
Input  : Scheduled tasks (Qdone), phase (ph), processor (curCU).
Output : Add the CRPD cost to a phase in case of migration or entanglement.
1  Function updateWCET(Qdone, ph, curCU)
2      foreach pp in ph.previousPhase do
3          if curCU != pp.targetCU() && curCU.type() == pp.targetCU().type() then
4              ph.applyCRPD()
5              break
6          else if curCU == pp.targetCU() then
7              foreach t ∈ Qdone do
8                  if pp == t || curCU != t.targetCU() then
9                      continue
10                 if t between pp and ph then
11                     ph.applyCRPD()
12                     break 2
```

ALGORITHM IV.4: Finding and propagating interference caused by the scheduling of the current phase.

```
Input  : Scheduled tasks (Qdone), phase (ph), processor (curCU).
Output : Check if the scheduling of the current task causes any interference of tasks/phases that have already been scheduled.
1  Function propInterferences(Qdone, ph, curCU)
2      ▷ Find any phase that has a longer WCET due to the scheduling of the current task.
3      foreach t ∈ Qdone do
4          if ph ∉ t then
5              foreach p1 ∈ t do
6                  if curCU ≠ p1.targetCU() then
7                      continue
8                  foreach p2 ∈ t do
9                      if p1.targetCU() ≠ p2.targetCU() then
10                         continue
11                     if p1 == p2 || p2 after p1 then
12                         continue
13                     if ph between p1 & p2 then
14                         impactedPhase ← p2
15                         break 3
16     if impactedPhase then
17         impacted ← [impactedPhase]
18         impacted.findAllImpacted()
19     foreach i ∈ impacted do
20         i.update(impactedPhase.crpd())
```

manner (not shown here due to space limitation), where all successors of the impacted task are added to the $impacted$ list (Line 17). Additionally, all tasks/phases that are scheduled after the impacted phases and on the same compute unit are also added to the impacted list. This is done recursively until all successors or tasks on the same core have been added to the list. After that the CRPD of the original impacted task/phase is used to update all impacted elements (Line 20).

Algorithm IV.4 is guaranteed to terminate as we only increment the release time of the impacted tasks. We never decrement the release time. Hence, in the worst case scenario all release times of all tasks that were released after the original $impactedPhase$ have to be incremented.

V. Evaluation

In this section we first evaluate our hFLS heuristic against an Integer Linear Programming (ILP) formulation and then against two synchronous phase-unaware heuristics. Our experimental results are based on synthetic benchmarks.

A. hFLS vs. ILP

Heuristic algorithms intrinsically generate approximate results but usually scale well. On the other hand ILP formulations give exact results to the problem but do not scale well. We compare schedules generated by the heuristics against the ones generated by the ILP to estimate the heuristics over approximation.

TABLE I: Summary of ILP variables.

γ	Set of compute units
τ	Set of tasks
$version(t)$	Set of versions for task t
$phase(X)$	Set of phases for X where X can be a task or a version
$succ(X)$	Set of successors for X where X can be a task or a phase
$deny(p)$	Set of forbidden compute units for phase p
D_t	Absolute deadline for for task t
$C_{p,c}$	WCET of a phase p on a compute unit c
$a_{t,v}$	1 if version v is selected for task t
$b_{p,c}$	1 if phase p is mapped on compute unit c
$d_{p,q}$	1 if phase p and phase q are on the same compute unit
$e_{p,q}$	1 if phase p is scheduled before phase k
$f_{p,q}$	1 if phase p is scheduled before phase k and both phases are on the same core
ρ_p	Start time of phase p
$\omega_{p,c}$	Augmented WCET for phase p on compute unit c

We summarise all variables and functions in Table I. For conciseness, we use logical operators ($\vee$, $\wedge$) that can be linearised using [14]. Our ILP formulation is as follows.

Task mapping. Eq. 1 assigns a single version a per task t.

$$\forall t \in \tau, \sum_{v \in version(t)} a_{t,v} = 1 \tag{1}$$

Phase mapping. Eq. 2 assigns one compute unit per selected ($a_{t,v} = 1$) phase p.

$$\forall t \in \tau, \forall v \in version(t), \forall p \in phase(v), \sum_{c \in \gamma} b_{p,c} = a_{t,v} \tag{2}$$

Detect identical mapping. Eq. 3 detects if two phases (p & q) are mapped to the same compute unit (c) .

$$\forall (u,t) \in \tau \times \tau, \forall p \in phase(u), \forall q \in phase(t), u \neq t, \quad d_{p,q} = \sum_{c \in \gamma} b_{p,c} \wedge b_{q,c} \tag{3}$$

Mapping restriction. Eq. 4 enforces compute unit mapping restrictions (e.g. p runs on a GPU and not on a CPU).

$$\forall t \in \tau, \forall p \in phase(t), \forall c \in deny(p), b_{p,c} = 0 \quad (4)$$

Order phases in task. Eq. 5 enforces the phase ordering within a version, i.e q only starts after p is complete if q succeeds p.

$$\begin{aligned} \forall t \in \tau, \forall p \in phase(t), \forall q \in succ(p), \\ \forall c \notin deny(p), \rho_p + \omega_{p,c} = \rho_q \end{aligned} \quad (5)$$

Deadline. Eq. 6 guarantees the timing constraints of the graph, i.e. all phases must complete before the deadline D.

$$\forall t \in \tau, q \in phase(t), \forall c \notin deny(q), \rho_q + \omega_{q,c} \leq D_t \quad (6)$$

Task dependency. Similar to Eq. 5, Eq. 7 enforces task dependencies, i.e. task q only starts after all its predecessor phases.

$$\begin{aligned} \forall u \in \tau, \forall t \in succ(u), \forall p \in phase(u), \forall q \in phase(t), \\ \forall c \notin deny(p), \rho_p + \omega_{p,c} \leq \rho_q \end{aligned} \quad (7)$$

Phase ordering. Eq. 8 sets the ordering of two successive phases. If $e_{p,q} = 1$, then phase p is scheduled before phase q. Eq. 9 detects the ordering of two successive phases on the same core ($d_{p,q} = 1$).

$$\forall (u,t) \in \tau \times \tau, \forall p \in phase(u), \forall q \in phase(t),$$

$$e_{p,q} + e_{q,p} = 1 \quad (8)$$

$$f_{p,q} = e_{p,q} \wedge d_{p,q} \quad (9)$$

Prevent overlap. Eq. 10 prevents multiple reservations of a computing unit at the same time. It uses the common big-M ($\mathcal{M}$) notation, where $\mathcal{M}$ is the sum of all WCETs.

$$\begin{aligned} \forall (u,t) \in \tau \times \tau, \forall p \in phase(u), \forall q \in phase(t), \\ \forall c \notin deny(p), u \neq t, \\ \rho_p + \omega_{p,c} \leq \rho_q + (1 - f_{p,q})\mathcal{M} \end{aligned} \quad (10)$$

Entanglement cost. Eq. 11 adds the CRPD cost (Section III-A) if two phases are entangled by augmenting the WCET ($\omega_{r,c}$) of phase r. The entanglement cost ($CRPD_{p,r}$) is only added if phase q is mapped between phase p & r and is on the same compute unit (d).

$$\begin{aligned} \forall (u,t) \in \tau \times \tau, \\ \forall p \in phase(u), \forall q \in phase(t), \forall r \in succ(p), \\ \forall c \notin deny(p) \cap deny(q) \cap deny(r), \\ (d_{p,q} == d_{q,r} == 1 \wedge \rho_p < \rho_q < \rho_r) \\ \implies \omega_{r,c} = C_{r,c} + CRPD_{p,r} \end{aligned} \quad (11)$$

Migration delay. Eq. 12 accounts for additional migration delay mentioned in Section III-A. Equations 11 and 12 are mutually exclusive as either the two phases p and r are mapped on the same core (Eq. 11) or on different core (Eq. 12). The migration delay is only added if the two phases target the same compute unit type but different compute units, e.g. LITTLE core 1 & 2.

$$\begin{aligned} \forall t \in \tau, \forall p \in phase(t), \forall q \in succ(p), \\ \forall c \notin deny(p) \cap deny(q), \\ \omega_{q,c} = C_{q,c} + CRPD_{p,q}(1 - d_{p,r}) \end{aligned} \quad (12)$$

Setup. The target platform for this experiment is a homogeneous quad-core CPU with an on-board GPU. We generate 1000 task graphs using Task Graphs For Free (*TGFF*) [15]. Additionally, TGFF provides the WCET for a simple CPU version and the deadline of the graph. We generate a three phase version which targets the GPU. The WCET of the GPU version is based on the CPU WCET and is either lower or equal to the CPU WCET. The task graphs contain on average 38 tasks. The CRPD of a phase is an arbitrarily chosen 5% of the WCET of the impacted phase. The same CRPD is applied to both hFLS and the ILP. Executing the DAG on the CPU in a sequential manner yields the completion time. And dividing the completion time by the deadline yields the utilisation.

To solve the ILP we use CPLEX [1] v12.10 which was executed on a 16core Intel system with 96GB memory. The heuristic was developed in C++ and was executed on the same machine as the ILP. For the ILP solver, we set up a timeout of 12h and a memory limit to 90GB per DAG.

Results. Due to the time and memory limits, the ILP solver was able to reach a decision in 78.8% of the total DAGs. Among these, 54.8% were unschedulable. All cases marked as unschedulable by the ILP were also decided as unschedulable by the heuristic. In comparison to the ILP the heuristic marked 2.4% more DAGs as unschedulable. Figure 6 compares the schedulability rate per utilisation range of the ILP and hFLS, focusing only on the DAGs where the ILP reached a decision. In Figure 7 we can see that the ILP does not scale well with increasing number of tasks.

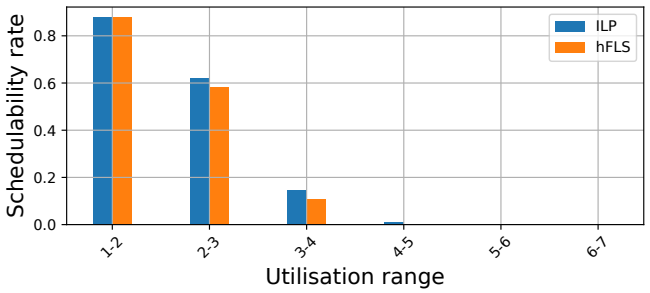

Fig. 6: Performance of hFLS scheduler vs. ILP scheduler. Each utilisation range represents the aggregated schedulability rate for all DAGs within that range, e.g. range 1 to 2 contains 123 DAGs out of which 108 were schedulable.

B. hFLS vs. eFLS vs. HEFT

We compare our approach with our own multi-version energy-aware *eFLS* approach [3] and with HEFT [6]. For a fair comparison we added multi-version capabilities to HEFT and modified the *eFLS* approach to focus on makespan instead of energy consumption. Both comparison heuristics use a synchronous scheduling approach, i.e., a GPU version with multiple phases blocks both the CPU and GPU for the complete execution time of the task.

Setup. We schedule 10,000 task graphs for the Odroid-XU4 [16]. The Odroid-XU4 contains an Arm big.LITTLE CPU [17]

[1]https://www.ibm.com/nl-en/analytics/cplex-optimizer

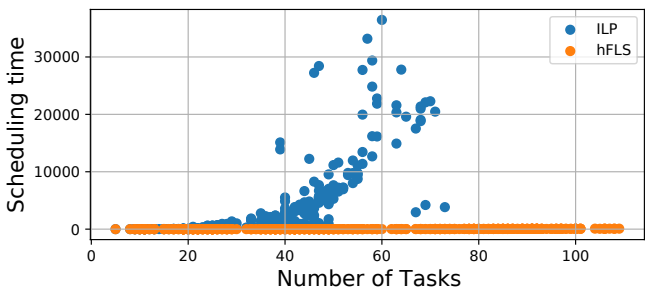

Fig. 7: Scheduling time taken with respect to the number of tasks.

and a Mali GPU [18]. The CPU contains 4 energy-efficient, in-order LITTLE cores and 4 high-performance, out-of-order big cores. CPU and GPU share a common memory.

TGFF [15] provides the structure of each DAG, the WCET of each task on the LITTLE cores and the deadline. The task graphs contain on average 76 tasks. Next to the LITTLE core version there are another 3 versions: one CPU only version for the big cores, one 3-phase GPU version initiated on the LITTLE cores and one 3-phase GPU version initiated on the big cores. The WCET of the additional versions is based on the WCET of the LITTLE version, similar to the approach in Section V-A. The utilisation of each task graph is calculated in the same manner as in the previous section. The only difference is that we use the LITTLE core WCET to compute the sequential schedule.

We calculate the CRPD of each phase based on the WCET of the impacted phase. The CRPD is equal to either the WCET (i.e. 100% of the WCET) or the platforms upper CRPD bound, e.g. fully reloading L2 cache with poor pre-fetching. Thus, for short phases the CRPD cost might be significant, but for long running tasks the CRPD costs might be negligible (e.g. Object Detection takes up to a second on a Jetson [19]).

The Odroid-XU4 has a 2MB big core L2 cache [16] and 64Byte cache lines [20]. Assuming a worst case scenario of 414 cycles difference between L2 and main memory [21] (i.e. having to reload L2 cache from main memory) at a frequency of 2GHz results in an upper bound of 6.8ms (rounded to 7ms). The additional time required for datasets larger than the L2 cache would already be accounted for in the WCET.

Results. The schedulability rate for each utilisation range is shown in Figure 8. The schedulability rate of the *hFLS* heuristic is the same as the schedulability rate of the *eFLS* and *HEFT* approaches until a utilisation index of 4. At utilisation rates above 4 the *hFLS* approach achieves on average 12% higher schedulability rates than the *eFLS* approach. In the utilisation ranges 9-11 the *hFLS* approach results in a 24% higher schedulability rate. *hFLS* outperforms the multi-version HEFT by 2.5% on average and is never worse. The most noteworthy is the significantly better performance (up to 11% higher schedulability rate) in the higher utilisation ranges. In a production environment this could make the difference between a cheaper and a more expensive SoC, as hFLS enables a better utilisation of the hardware. Lastly, *hFLS* performs statistically significantly (McNemar's Test [22]) better than both approaches at a confidence interval of 99.9%. Note that all approaches can schedule DAGs with a utilisation above 8 (i.e. above the number of cores). This is caused by two factors. First, the utilisation is calculated based on the LITTLE core WCET. However, tasks can have a lower WCET on the big cores and on the GPU. Second, the GPU adds an additional "core".

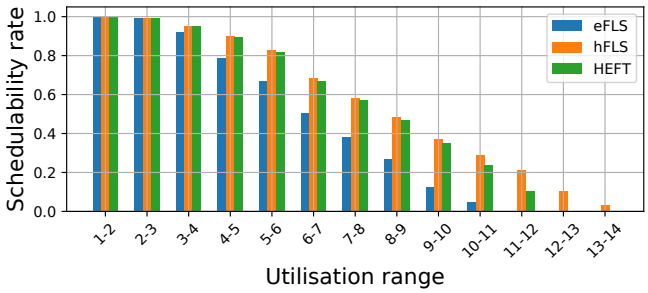

Fig. 8: Performance of multi-phase schedules vs. single-phase schedules. Each utilisation range represents the aggregated schedulability rate for all DAGs within that range, e.g. range 2 to 3 contains 1099 DAGs out of which 1039 were schedulable by HEFT and hFLS.

C. Sorting for hFLS

Rouxel et al. [7] showed that no sorting strategy consistently produces the best schedule for the *eFLS* approach. In our current work on multi-phase scheduling we observe that the HEFT ranking results in the best schedule in 73.6% of the cases. One of the two BFS sorting strategies results in the best schedule in 25.6% of the cases. In the other 0.8% of the cases one of the DFS strategies was better. In the few cases where a DFS strategy resulted in a better schedule than either BFS or HEFT ranking the makespan improved by only 2.6%. We observe that a DFS strategy leads to more entanglement of later phases and propagates delays throughout the schedule, i.e. more CRPD values are added to tasks, thereby resulting in longer makespans. Relaxing the need to try all different sorting algorithms decreases the scheduling time of our heuristic, as the DFS sorting strategies are not necessary.

VI. Related Work

Most previous research in the area of static scheduling for heterogeneous architectures focused on scheduling for heterogeneous CPUs [6], [23]–[26]. Previous work on scheduling incorporating GPUs has either focused on workload balancing between the CPU and GPU [26], [27], introducing real-time capabilities to desktop GPUs [28] or running multiple tasks concurrently on the same GPU [29].

In [30] the authors introduce a run-time system for task and data mapping for embedded systems with a GPU. The aim is to minimise the completion time. However, a task only requires either the CPU or GPU. In contrast to our approach they cannot handle multiple versions.

Multi-version scheduling is explored in [2], [3] and [4]. Houssam-Eddine et al. [2] introduced an online scheduling

approach with an offline component for scheduling multi-version tasks. The target architecture for a task is selected offline and then the task is scheduled online. Their approach does not support executing a task on both CPU and GPU, i.e. a GPU task does not have a CPU phase. Furthermore, in contrast to our offline approach, their approach requires loading binaries for all versions which substantially increases the memory requirements.

Aldegheri et al. [4] also target CPU&GPU scheduling for multi-version tasks. Their approach extends HEFT by improving the ranking of *exclusive* tasks (tasks that only have one implementation). Additionally, tasks can only use either the CPU or the GPU. Our tasks are not limited to using a single compute engine but can make use of multiple compute engines and switch between them.

Lastly, our multi-phase approach extends well-known concepts of multi-phase tasks (AER [11], PREM [12] or LET [31]) by allowing phases to execute on different compute units.

VII. Conclusion

We extend existing task models along with a scheduling strategy to fully benefit from the capacities offered by heterogeneous hardware such as the Odroid-XU4 or the Nvidia Jetson boards. To the best of our knowledge we are the first to propose a task model and a heuristic that interleave CPU and GPU workloads of different tasks using a multi-version multi-phase approach.

We demonstrate that our approach can utilise the hardware better than two synchronous schedulers, improving schedulability by up to 11% and 24% respectively for high utilisation DAGs. Additionally, we show that HEFT ranking or BFS sorting algorithms perform best in our approach, reducing the scheduling time by 40%. Lastly, the solutions given by our hFLS heuristic are close to the optimal with at most 2.4% degradation.

In the future we plan to combine our energy-aware DVFS scheduler [3] with our multi-phase approach. Hence, further improving energy efficiency for high-performance embedded systems.

Acknowledgment

We thank the reviewers for their time and feedback.

This work is supported and partly funded by the European Union Horizon-2020 research and innovation programme under grant agreement No. 779882 (TeamPlay).

References

[1] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, "An empirical survey-based study into industry practice in real-time systems," in RTSS, 2020.

[2] Z. Houssam-Eddine, N. Capodieci, R. Cavicchioli, G. Lipari, and M. Bertogna, "The HPC-DAG Task Model for Heterogeneous Real-Time Systems," IEEE Trans. on Computers, 2020.

[3] J. Roeder, B. Rouxel, S. Altmeyer, and C. Grelck, "Energy-aware scheduling of multi-version tasks on heterogeneous real-time systems," in SAC, 2021, pp. 501–510.

[4] S. Aldegheri, N. Bombieri, and H. Patel, "On the task mapping and scheduling for DAG-based embedded vision applications on heterogeneous multi/many-core architectures," in DATE, 2020, pp. 1003–1006.

[5] R. Davis and A. Burns, "A survey of hard real-time scheduling algorithms for multiprocessor systems," ACM Computing Surveys, 2011.

[6] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," IEEE TPDS, vol. 13, no. 3, pp. 260–274, 2002.

[7] B. Rouxel, S. Skalistis, S. Derrien, and I. Puaut, "Hiding communication delays in contention-free execution for spm-based multi-core architectures," in ECRTS, 2019.

[8] J. Pallister, S. Hollis, and J. Bennett, "Identifying compiler options to minimize energy consumption for embedded platforms," Comput. J., vol. 58, no. 1, 2015.

[9] S. Altmeyer and C. M. Burguière, "Cache-related preemption delay via useful cache blocks: Survey and redefinition," J. of Systems Architecture, vol. 57, no. 7, pp. 707–719, 2011.

[10] V. A. Nguyen, D. Hardy, and I. Puaut, "Cache-conscious offline real-time task scheduling for multi-core processors," in ECRTS, 2017.

[11] C. Maia, L. Nogueira, L. M. Pinho, and D. G. Pérez, "A closer look into the AER model," in ETFA, 2016, pp. 1–8.

[12] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for COTS-based embedded systems," in RTAS, 2011, pp. 269–279.

[13] H. Rihani, M. Moy, C. Maiza, R. I. Davis, and S. Altmeyer, "Response time analysis of synchronous data flow programs on a many-core processor," in RTNS, 2016, pp. 67–76.

[14] G. G. Brown and R. F. Dell, "Formulating integer linear programs: A rogues' gallery," ITE, vol. 7, no. 2, pp. 153–159, 2007.

[15] R. Dick, D. Rhodes, and W. Wolf, "TGFF: task graphs for free," in CODES/CASHE, 1998.

[16] Odroid-XU4. https://wiki.odroid.com/odroid-xu4/odroid-xu4. Accessed: 2019-09-06.

[17] ARM Ltd., "White Paper: big.LITTLE Technology : The Future of Mobile," 2013.

[18] "Exynos 5 Octa 5422 Processor: Specs, Features: Samsung Exynos," Jun 2019. [Online]. Available: https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-5-octa-5422/

[19] N. Tijtgat, W. Van Ranst, T. Goedeme, B. Volckaert, and F. De Turck, "Embedded real-time object detection for a UAV warning system," in ICCVW, 2017, pp. 2110–2118.

[20] Cortex-A15 Technical Reference Manual. https://developer.arm.com/documentation/ddi0438/d/Level-2-Memory-System/About-the-L2-memory-system. Accessed: 2021-05-31.

[21] S. Stepanovic, G. Georgakarakos, S. Holmbacka, and J. Lilius, "Quantifying the Interaction Between Structural Properties of Software and Hardware in the ARM Big.LITTLE Architecture," in IEEE PDP, 2018, pp. 138–144.

[22] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," Psychometrika, vol. 12, no. 2, pp. 153–157, 1947.

[23] L. F. Bittencourt, R. Sakellariou, and E. R. Madeira, "Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm," in IEEE PDP, 2010, pp. 27–34.

[24] H. Zhao and R. Sakellariou, "An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm," in Euro-Par, 2003, pp. 189–194.

[25] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," IEEE TPDS, vol. 25, no. 3, pp. 682–694, 2013.

[26] R. Sakellariou and H. Zhao, "A hybrid heuristic for DAG scheduling on heterogeneous systems," in IPDPS, 2004, p. 111.

[27] D. Grewe and M. F. O'Boyle, "A static task partitioning approach for heterogeneous systems using OpenCL," in CC, 2011, pp. 286–305.

[28] C. Hartmann and U. Margull, "Gpuart-an application-based limited preemptive gpu real-time scheduler for embedded systems," J. of Systems Architecture, vol. 97, pp. 304–319, 2019.

[29] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang, "An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads," in RTAS, 2017, pp. 353–364.

[30] Z. Li, Y. Zhang, A. Ding, H. Zhou, and C. Liu, "Efficient algorithms for task mapping on heterogeneous CPU/GPU platforms for fast completion time," J. of Systems Architecture, vol. 114, p. 101936, 2021.

[31] T. A. Henzinger, C. M. Kirsch, M. A. A. Sanvido, and W. Pree, "From control models to real-time code using Giotto," IEEE Control Systems Magazine, vol. 23, no. 1, pp. 50–64, 2003.