MESSI: Task Mapping and Scheduling Strategy for FPGA-based Heterogeneous Real-Time Systems

SALLAR AHMADI-POUR, Institute of Computer Science, University of Bremen, Germany
SANGEET SAHA, School of Computer Science & Electronics Engineering, University of Essex, United Kingdom
KLAUS D. MCDONALD-MAIER, School of Computer Science & Electronics Engineering, University of Essex, United Kingdom
ROLF DRECHSLER, Institute of Computer Science, University of Bremen, Germany and Cyber-Physical Systems, DFKI GmbH, Germany

Continuous demands for improved performance within constrained resource budgets are driving a move from homogeneous to heterogeneous processing platforms for the implementation of today's *Real-Time* (RT) embedded systems. The applications executing on such systems are typically represented as a *Precedence Task Graph* (PTG), where a node represents a task or algorithm for one functionality and edges represent the complex interactions between multiple functionalities. Due to RT constraints, the task graph needs to be executed within a specified deadline. Although some existing studies have looked into solving this challenge, comprehensive studies that combine the theoretical features of RT task-graph mapping and scheduling with practical runtime architectural characteristics have mostly been ignored to date. Hence, in this paper, we consider the challenge of scheduling a RT application modelled as a single PTG, with the objective of minimizing the overall execution time under *Hardware* (HW) resource and deadline constraints for heterogeneous *Central Processing Unit* (CPU) + *Field Programmable Gate Array* (FPGA) architectures. First, we introduce an optimal solution using *Integer Linear Programming* (ILP). However, this ILP-based optimal solution suffers from computational complexity and does not scale well even for moderately large problem sizes. Hence, we additionally propose heuristic algorithms for task mapping and scheduling. The efficiency of the proposed scheme, named MESSI, has been evaluated through experiments using PTGs on a practical CPU+FPGA system regarding current technology restrictions. Our experiments demonstrate that performance gains of 55.6 % and area usage reductions of 46.3 % are possible compared to full *Software* (SW) and HW execution, respectively.

CCS Concepts: • **Hardware → Electronic design automation**; **Methodologies for EDA**; **Operations scheduling**; **Hardware accelerators**; **Reconfigurable logic and FPGAs**; • **Computer systems organization → Embedded systems**; **System on a chip**; **Embedded hardware**; **Embedded software**.

Additional Key Words and Phrases: Task Graph, Scheduling, HW SW Co-design, Integer Linear Programming, Heuristic, FPGA, Real-Time

Authors' addresses: Sallar Ahmadi-Pour, Institute of Computer Science, University of Bremen, Bibliothekstrasse 5, Bremen, Bremen, Germany, 28359, sallar@uni-bremen.de; Sangeet Saha, School of Computer Science & Electronics Engineering, University of Essex, Wivenhoe Park, Colchester, Essex, United Kingdom, CO4 3SQ, sangeet.saha@essex.ac.uk; Klaus D. McDonald-Maier, School of Computer Science & Electronics Engineering, University of Essex, Wivenhoe Park, Colchester, Essex, United Kingdom, CO4 3SQ, sangeet.saha@essex.ac.uk; Rolf Drechsler, Institute of Computer Science, University of Bremen, Bibliothekstrasse 5, Bremen, Bremen, Germany, 28359 and Cyber-Physical Systems, DFKI GmbH, Bibliothekstrasse 5, Bremen, Bremen, Germany, 28359, drechsler@uni-bremen.de.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1084-4309/2023/0-ART0 $15.00
https://doi.org/0.0

ACM Reference Format:
Sallar Ahmadi-Pour, Sangeet Saha, Klaus D. McDonald-Maier, and Rolf Drechsler. 2023. MESSI: Task Mapping and Scheduling Strategy for FPGA-based Heterogeneous Real-Time Systems. *ACM Trans. Des. Autom. Electron. Syst.* 0, 0, Article 0 (2023), 29 pages. https://doi.org/0.0

1 INTRODUCTION

Over the years, we have witnessed a drastic shift in the nature of processing platforms employed in RT embedded systems. For example, modern *System-on-Chip* (SoC) platforms contain multicore processors with specialized *Digital Signal Processing* (DSP) cores, *Graphics Processing Units* (GPUs), FPGAs, Application Specific Instruction Set Processor or other Application Specific Integrated Circuit. Processing platforms with such varying types of computing elements are known as heterogeneous platforms. These heterogeneous platforms typically deliver higher performance and better energy efficiency as compared to general-purpose processors [58]. Currently, the NVIDIA Tegra [26]/Jetson [27] with GPUs and the Xilinx Zynq [52] with SoC FPGAs are popular examples. Particularly, FPGA-based heterogeneous systems have drawn considerable interest due to their flexible architecture that can enable efficient HW customization for particular algorithms. For example, FPGA-based systems have been deployed in RT edge computing to accelerate AI-based face tracking, leading to high throughput at low power [22]. Such heterogeneous FPGA-based systems contain a CPU as soft-core *Intellectual Property* (IP) or even hardwired, IP such as in the Xilinx Zynq SoC FPGA. CPU+FPGA systems are not only relevant for prototyping such systems but become the viable realization of embedded systems as their prices decrease. In recent years, heterogeneous embedded systems have been realized through CPU+FPGA systems for their advantages over normal FPGA-based solutions [30]. The flexibility of these platforms makes them popular in fields where FPGAs are predominant, like aerospace [19, 49] and automotive [5, 28], as well as emerging fields like the Internet of Things [14, 18].

Applications in today's heterogeneous embedded systems are often represented by *Directed Acyclic Graphs* (DAGs) or PTGs. In such PTGs, a node represents a task associated with the application, while an edge denotes interdependencies among tasks. When these PTGs (alternatively referred to as task graphs or DAGs in the remainder of this paper) are implemented on heterogeneous platforms [1], (i) the same task may require different execution times on different *Processing Elements* (PEs), and (ii) inter-task data transmission may incur distinct overheads on the different communication channels. Moreover, due to its RT nature, such systems often impose a stringent timing constraints where a specific application has to be executed within the given deadline by executing all the associated tasks. Hence, *given an application modelled as a task graph and a heterogeneous computing platform, the successful execution of all associated tasks within the given deadline while satisfying all resource, precedence-related and architectural constraints is a challenging scheduling problem.*

The problem of RT scheduling of task graphs can be broadly categorized as either static/offline scheduling or dynamic/online scheduling [23]. In the case of static scheduling, decisions, for example, "task-to-processor allocations", execution start times of tasks, etc., are determined offline prior to the system starting its operation. Such offline scheduling is popular for embedded systems because information such as the worst-case execution requirement of each task on every processor, precedence constraints, and communication overheads between task pairs are completely or partially available at design time. However, such partial or complete information about a task graph is not available before execution in the case of dynamic scheduling, and thus all the scheduling decisions can only be taken at runtime.

As many RT systems require a high degree of timing predictability with well-defined workload (e.g., manufacturing robot control, avoinic systems), it is typically preferable to employ offline

scheduling algorithms for such systems, as this allows all timing requirements to be specified offline before runtime operation [4, 41]. Thus, this work also deals with the generation of static schedules for PTGs.

The task graph scheduling problem is usually classified as NP-complete [42]. This means that strategies that try to find the best schedules for PTGs on different types of PEs often come with high computational costs, even for small problem sizes. This is mainly because they necessitate a thorough enumeration of an exponential state space. Therefore, researchers are often focused on designing low-complexity heuristics that can generate a near optimal schedule within a reasonable time [37].

A majority of heuristic scheduling policies attempt to develop the schedule with the objective of minimizing overall schedule length, also known as makespan time. In the context of task graph scheduling for embedded systems, this makespan time minimization could be beneficial and necessary in many ways. For example, in an anti-collision controller for a robot, the obstacle detection application (generally represented as a task graph [48]) is executed repetitively. A lower makespan will provide the actuators with enough time to take action, which in turn could potentially improve stability [44]. Additionally, with the deadline constraint, this lower makespan time will generate slack times that can be used to improve other performance metrics of the system, such as expenditure on QoS enhancement [32, 33], energy consumption [38, 50], and reliability [10, 53].

However, in spite of the practical importance, many existing works [29, 54, 59] that deal with makespan minimization for heterogeneous embedded systems only evaluated their technique via SW simulations using hypothetical parameters without considering any practical constraints. *Until now, studies that combine the theoretical aspects of RT scheduling of task graphs along with runtime architectural characteristics have not been conducted.* Unlike these existing techniques, MESSI, attempts to address the problem of scheduling a RT application modelled as a PTG, which must be scheduled within a stipulated deadline, with the objective of minimizing the makespan time under system-wide constraints. The targeted platform is an CPU+FPGA-based heterogeneous architecture.

The main technical contributions of this paper are:

- Formulation of an ILP-based optimal solution strategy, which can be used to obtain schedules for RT applications represented as a single PTG, executing on a heterogeneous platform. The scalability of the proposed ILP is better than the optimal strategy in the previously presented work [1]. The scalability of our previous ILP was limited due to its explicit dependency on the deadline associated with a given PTG. In MESSI, we used different integer variables to represent the instants at which the task begins and completes execution on the processor to which it has been assigned. In the proposed ILP formulation, we have considered the communication time between two tasks if they were placed in different processing elements. This reflects the possible system's properties more accurately.
- In *Design Space Exploration* (DSE), where numerous rapid design iterations are required, a substantial time overhead is often unaffordable. Therefore, in addition to the optimal solution approach, we propose two distinct heuristic algorithms for task mapping and scheduling. It is observed that the solution qualities delivered by the proposed heuristic is similar to the ILP-based solution. However, the computational overheads associated with ILP are significantly higher than the proposed heuristics.
- We show how to implement a given scheduling on a practical CPU+FPGA system regarding current technology restrictions and discuss the different trade-offs with respect to the system capabilities. This is mostly not considered in related works.

- We provide a case-study to validate the applicability of our proposed scheduling technique in delivering practical results and demonstrate that performance gains of 55.6 % and area usage reductions of 46.3 % are possible compared to a full SW and HW execution, respectively.

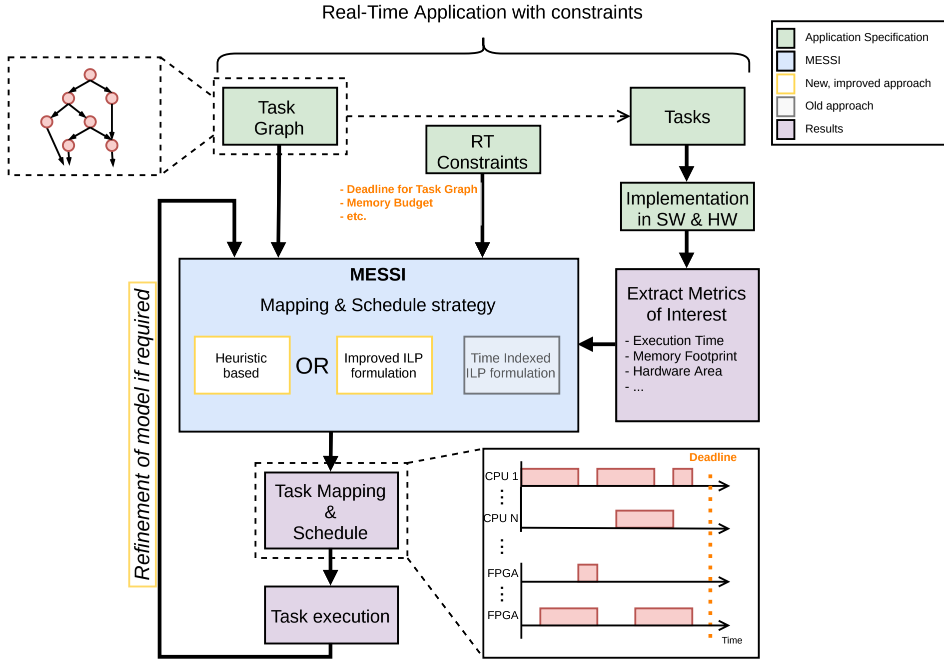


Fig. 1. Overview of our proposed mapping and scheduling strategy.

Fig. 1 shows a high-level overview of MESSI, our proposed approach for task mapping and scheduling in a heterogeneous CPU+FPGA system. The green boxes represent the application's specification, and initial artifacts for the system. The RT constraints define the deadline for the schedule, how much memory and area is available on the CPU+FPGA system, and potential other constraints of the RT system. The task graph represent the tasks with their dependencies and order in which they are executed. The tasks themselves should each be available as SW implementation for the CPU and as HW implementation for the FPGA (e.g., synthesizable *Register-Tranfer Level* (RTL) models written in a HW description language). Based on the task implementations, relevant execution metrics are obtained by executing the tasks in isolation (purple, right, middle of Fig. 1). Possible important metrics are the execution time and area usage on the respective FPGA. These metrics are passed together with the RT constraints and task graph to MESSI, our proposed mapping and scheduling strategy (blue, left, middle of Fig. 1), which derives a task mapping and scheduling. Depending on the requirements and the development cycle, a mapping and schedule can be obtained from MESSI through either ILP or heuristics (yellow boxes within MESSI, right middle of Fig. 1). The scheduling is then implemented on the heterogeneous CPU+FPGA system and executed. If there are differences between the obtained schedule and the executed schedule, a refinement step can be performed to add system specific constraints or conditions to the ILP formulation or the heuristics. Possible differences are expected, as the ILP formulation targets a generalized model for many possible systems. This refinement step is optional in case the mapping & schedule

implemented on the system diverge too much from the ILP solution. Refinements can contain timing specific properties of the CPU or constraints to enable limited preemption between tasks and task communication. New constraints may be added to provide more specialization for a specific CPU+FPGA system. The trade-off in refining either the ILP formulation or the heuristics comes with a more accurate solution (due to specialized constraints) by losing generality for other systems and PTGs. Refining the ILP formulation allows for a more analytical description than a refinement of the heuristic. For the heuristics, refinements consider the requirements as well as decision heuristics for mapping and scheduling, thus possibly affecting the quality of the obtained solution. It has to be noted that in MESSI, the heuristic techniques do not need to be employed in conjunction with the ILP technique; both techniques are independent of each other and can be employed separately.

Additional metrics can be integrated in order to consider practical implementation constraints, such as the communication overhead in moving results between the CPUs and FPGAs accessible memories. However, such metrics are highly system specific and can vary depending on the capabilities of the system. In this work, we focus on bare-metal systems and consider the RISC-V *Instruction Set Architecture* (ISA). Our evaluation case-study demonstrates the applicability of our proposed scheduling algorithm in providing practical results for a heterogeneous RISC-V CPU+FPGA system.

1.1 Structure of the Paper

This journal paper includes and expands upon published material from our previous conference paper [1]. After this outline, we will elaborate on our new contributions of this paper in the next paragraph. Following this introduction, the remainder of the paper is organized as follows: In Section 2 we discuss the related work and the context in which MESSI fits into. We present our proposed scheduling strategy in Section 3, which covers our ILP-based formulation. In Section 4.1 and Section 4.2 we discuss our heuristic approaches for mapping and scheduling, respectively. Then, we discuss practical constraints and trade-offs in implementing the resulting mapping and scheduling on a heterogeneous CPU+FPGA system in Section 5. Next, we present our RISC-V case-study with an example application task graph on which we employ MESSI and show the obtained results in Section 6. In Section 7 we further discuss our proposed methodology, the obtained results and provide ideas for future work. Finally, we conclude the paper in Section 8.

2 RELATED WORK

A plethora of existing work discusses the scheduling problem for general multicore computing environments, which involve various SW computing modules such as CPUs and GPUs [11, 56]. These algorithms take into account the different computation speeds of heterogeneous PEs as well as intercore parallelism. Many works discuss tailored models for application domains and specific heterogeneous system configurations (e.g., CPU+FPGA). Hence, model formulations consider different parameters of the underlying system (e.g., configuration time). Among CPU+FPGA systems, there also exist heterogeneous systems containing CPU+GPU systems and *Multi-Processor System-on-Chips* (MPSoCs). Related works tailoring solutions for systems different from CPU+FPGA systems are still of interest, as formulations in the models or constraints based on the application domain can be related. Hence, we will also discuss works that focus on CPU+GPU systems and MPSoCs as well.

In [16], the authors exploit the advantages of heuristic-based algorithms and also proposed a genetic algorithm-based task allocation strategy to minimize the schedule length. Similarly, a machine learning based online task scheduler for hybrid CPU+GPU systems has been proposed in [13]. However, such computation-intensive methods often raise concerns regarding resource

limitations on real platforms. Thus, some studies propose scheduling methods for systems with limited computing resources. In [10] the authors present a scheduling algorithm for a fixed number of heterogeneous processing units (CPUs, GPUs) to obtain both a high performance and lower makespan time, while maintaining the system's reliability against any faults. Xie et al. [51] introduce a methodology for energy-constrained task scheduling with a primary focus on the power usage of MPSoCs. This work provides insight into the usage of power estimations in the task scheduling across processors, but is only applied to MPSoCs.

With the increasing complexity level of high-performance computing and RT embedded systems, current heterogeneous computing systems are employing FPGAs along with CPUs and GPUs to overcome existing limitations [3, 6]. FPGA-based multicore systems are composed of multiple SW executing PEs (i.e., multiple CPUs and GPUs) and fixed HW resources (FPGA area).

Letras et al. [20] propose an approach to map data flow based applications onto MPSoCs through multi-objective optimization. While this method allows to optimize for high throughput, it does not consider the utilization of custom HW acceleration or FPGA-based heterogeneous systems. In [39] the authors explore the optimization of shared memory architecture for heterogeneous systems with HW accelerators and CPUs. This method enables extensive use of shared memory architectures with increases in performance, but does not particularly deal with the partitioning and scheduling of tasks within the system. Such methods would extend the possibilities in the design space for heterogeneous systems, as Section 5.1 discusses shared memory as a technique to improve performance.

In recent years, the problem of RT task execution on FPGA-based heterogeneous systems has gathered considerable attention from the research community. The generic problem of RT scheduling tasks has branched out in different directions, primarily based on: i) using optimizing frameworks [31], ii) using heuristic algorithms [57], and iii) using priority-driven algorithms [17]. In [31], the authors proposed a static partitioning-based scheduling strategy for CPU+FPGA systems to minimize energy consumption. In [55], the authors measured the speed-up in task execution on an FPGA and by utilizing their *speed-up utilization model*, they determine the appropriate PE (i.e., CPU or FPGA) to assign the tasks to. However, all these works are designed for non-RT applications and do not consider HW constraints. Recently, Zhu et al. [59] proposed a RT task scheduling framework for CPU+FPGA systems, but their work only considered independent tasks. Dependent RT task scheduling in an FPGA-based multicore setting to minimize the makespan under HW resource constraints has been investigated in [54]. However, this technique is only evaluated via SW simulations using hypothetical FPGA parameters without considering any practical constraints.

Ding et al. [12] propose a task partitioning, scheduling and floorplanning approach for partially dynamically reconfigurable systems. The main focus of this work is the realization of a task graph through full HW execution within a single FPGAs resources, which is capable of being partially reconfigured during runtime. While this approach maps a task graph, with its data dependencies for partially reconfigurable FPGAs, it doesn't consider CPUs as part of the system. Hence, a possible execution in SW versus HW is not considered. Furthermore, RT constraints, as discussed in our work, aren't considered either.

In [37] the authors introduce a scheduling approach that is tailored for utilization in embedded systems in which the temperature of the chip cannot exceed a defined limit. This approach minimizes the make span of the task graph with respect to the system's temperature limit. For this, the authors utilize ILP as well as a newly introduced heuristic, but in this work only embedded systems containing a single CPU are considered. In [15] the authors showed how tasks can be mapped onto multiple CPUs utilizing ILP enhanced with logic-based Benders decomposition. While this technique allows to overcome some scalability issues of ILP, the formulation of the problem looks into task

scheduling w.r.t. the makespan alone, however it is not considering any resource constraints or HW acceleration.

The authors of [43] introduce a scheduling framework for FPGA-based heterogeneous systems in order to achieve higher performance. However, the proposed framework does not consider resource constraints and targets to increase high performance computing workloads as well as partial reconfiguration of the FPGA. While the approach of Xu et al. [54] is related to our approach, because it targets task graph scheduling for FPGA-based heterogeneous systems under RT constraints, it solely relies on integer programming techniques. In the work of Xu et al. two approaches are introduced and utilized. Their first algorithm utilizes ILP to solve the task mapping and scheduling, but does not consider the communication between tasks. In order to overcome the lack of the missing communication times, the second approach treats the communication between tasks as part of the optimization problem. Their second algorithm relies on integer non-linear programming, hence potentially requiring more computational effort than ILP-based solutions.

While the work of Xu et al. [54] is related, a fair comparison is challenging due to their reliance on theoretical models without experimental validation on an actual CPU+FPGAs system. Additionally, there appear to be differences in the workloads and benchmarks used across related studies, making it difficult to establish a common ground for comparison, even when similar applications like fast Fourier transform or Gaussian elimination are considered.

In MESSI, we provide an improved ILP-based scheduling generation with communication constraints. We additionally provide two heuristics, in order to overcome the scalability issues that ILP computations can be subject to for bigger task graphs. Finally, we discuss and empirically validate all our theoretical findings in practical implementation with real-life case studies. Until now, studies that consider both, the theoretical aspects of RT scheduling of tasks along with runtime architectural characteristics, have not been conducted. MESSI, our mapping and scheduling algorithms, fills this gap with the objective of minimizing the makespan under HW resource constraints for CPU+FPGA based heterogeneous RT architectures.

3 PROPOSED SCHEDULING STRATEGY

In this section, we provide the necessary definitions (Section 3.1 and Section 3.2) and present the proposed constraint-based formalism to obtain scheduling in this context (Section 3.3).

3.1 Application and Architecture Model

We model a RT application ($\mathcal{A}$) as a PTG, $G = (T, E)$, where T is a set of tasks ($T = \{T_i \mid 1 \leq i \leq n\}$) and E is a set of directed edges ($E = \{\langle T_u, T_v \rangle \mid 1 \leq u, v \leq n; u \neq v\}$) representing the dependency between distinct pairs of tasks. n denotes the number of tasks within the set T. An edge $\langle T_u, T_v \rangle$ refers that the task T_v can begin execution only after the completion of T_u. The source task does not have any predecessors; similarly, the sink task does not have any successors. In the case of multiple sink/source nodes, we add dummy nodes having an execution time of zero, connected to each of the multiple source / sink nodes. This allows us to consider only the PTG formation with one single source and sink task. Being a RT application, the entire application ($\mathcal{A}$) must satisfy its deadline, denoted as D_{DAG}, by executing all the task nodes within the interval.

In a heterogeneous multiprocessor system, PEs of the same type are usually grouped together as a cluster. The individual tasks belonging to an application are assigned to clusters. The architectural model considered in this paper consists of a two type of clusters. One type of cluster contains *Embedded Processors* (EPs) (or CPUs) and another type of cluster consists of *Reconfigurable Logic* (RL) (or FPGA). Let us assume CPU cluster is denoted as Cl_C and there are m_{EP} number of EPs. The FPGA cluster is denoted as Cl_F and the number of RL within the cluster is denoted as m_{PRL}.

344 Each task T_i of the task graph contains a tuple denoting SW execution time (CPU execution time),
345 HW execution time (FPGA execution time) and HW area cost (amount of logic gates requirement),
346 respectively. Additionally, each edge E also includes the communication cost between tasks T_i
347 and T_j . All tasks can be bi-partitioned into HW executable, HW or SW executable, SW, satisfying
348 $HW \cup SW = T$ and $HW \cap SW = \emptyset$. All processors are also assumed to be identical within a cluster,
349 thus a SW task's execution time is identical in any processor.
350

351 ### 3.2 Problem Description

352 Generate a RT schedule with feasible PE assignment, and start time for each task node of a given
353 DAG having a stipulated end-to-end deadline, such that the total completion time is minimized,
354 while ensuring that deadline, precedence, and resource constraints are not violated on a hetero-
355 geneous multiprocessor platform. To achieve this, the mapping and scheduling strategy should
356 answer the following questions:
357
358 (1) What task to schedule at which time (*temporal reconfiguration*)?
359 (2) Where to place the respective task, in CPU or FPGA (*spatial reconfiguration*)?
360 (3) When to start the execution of a task according to its precedence constraints (*temporal*
361 *scheduling*)?

362 This setup can be compared to a multiprocessor task allocation problem (whilst being more
363 abstract by including HW execution), as the platform provides multiple different PEs for the
364 execution of a task. However, due to the constraints (as mentioned below) and the challenges
365 associated with heterogeneous architecture, existing multiprocessor scheduling strategies cannot
366 be applied.
367 The constraints for the given problem description are as follows:
368
369 (1) HW task execution is non-preemptive.
370 (2) The communication not only has to take place between tasks in SW, but also between the
371 SW and HW domain to utilize the HW accelerators.
372 (3) Execution times of a task are heavily dependent on the selected execution unit. In general,
373 the execution in HW is faster as compared to the SW. However, this depends upon the
374 task's characteristics (see Tab. 3).

375 To execute a task in SW or HW, considering the given constraints, is an optimization problem.
376 In the following section, we present how to obtain an effective solution to this problem.
377

378 ### 3.3 ILP-based Mapping and Scheduling

379 We present a scheduling strategy based on ILP. For this purpose, we first introduce an integer
380 decision variable $S_i \in \mathbb{Z}^+$ to capture the start time of each task T_i, where $\mathbb{Z}^+$ denotes the set of positive
381 integers. We further define a binary decision variable, Z_{u,cl_i}, where, $u = 1, 2, ..., n$; $cl_i = Cl_C, Cl_F$;
382 Z_{u,cl_i} is 1, if T_u executes on cluster cl_i, otherwise 0. We define another binary variable Y_{uv}, where
383 $Y_{uv} = 1$, if task T_u starts before T_v, else 0. The variable $ET(u, cl_i)$ denotes the execution time of task
384 T_u if executes on cluster cl_i. Similarly, $EC(u, cl_i)$ denotes the communication/data transfer time
385 needed to transfer the data from cl_i for task T_U.
386 Data communication between two clusters resulting from data dependence between two tasks
387 of an edge ($\langle T_u, T_v \rangle \in E$) is modeled as a binary variable β_{u,v,cl_i,cl_j}. It is "1" when there exists a data
388 dependency between T_u and T_v in task graph G and task u and v are assigned to clusters cl_i and cl_j,
389 respectively.
390 To model our mapping and scheduling strategy, the required constraints on the decision variable
391 are now stated as follows:
392

393 (1) Each task T_u is assigned to exactly one cluster:
394
395
$$\forall u \mid \sum_{cl_i \in CL} Z_{u,cl_i} = 1 \tag{1}$$
396
397 (2) The application $\mathcal{A}$ must meet its end-to-end absolute deadline D_{DAG}. Hence, the sink node
398 T_n should be finished by D_{DAG}, which is represented by the following constraint:
399
400
$$S_n + \sum_{cl_i \in CL} (ET(n, cl_i) \times Z_{n,cl_i}) - 1 \leq D_{DAG} \tag{2}$$
401 (3) **Lemma 1:** For an edge, $\langle T_u, T_v \rangle \in E$, data communication exists between two clusters cl_i
402 and cl_j if and only if there exists data dependence between tasks T_u and T_v, while task T_u is
403 assigned to cl_i and task T_V is assigned to cluster cl_j. This can be modeled as following:
404
405
$$2\beta_{u,v,cl_i,cl_j} - 1 \leq Z_{u,cl_i} + Z_{v,cl_j} - 1 \leq \beta_{u,v,cl_i,cl_j} \tag{3}$$
406 The precedence constraints between the tasks must also be satisfied. The execution of T_v
407 should commence only after the completion of its predecessor T_u and all data communication
408 from its predecessors to itself are finished. Using Eq. (3) this constraint can be formulated
409 as follows:
410
411
$$\forall(\langle T_u, T_v \rangle) \in E \mid S_u + \sum_{cl_i \in CL} [ET(u, cl_i) \times Z_{u,cl_i}] + \sum_{cl_i \in CL} \sum_{cl_j \in CL} [\beta_{u,v,cl_i,cl_j} \times EC(u, cl_j)] \leq S_v \tag{4}$$
412
413 NOTE: Within a single cluster, if two tasks are assigned to different PEs, then the com-
414 munication cost may be associated with that, and those constraints can be formulated
415 similarly. However, in this formulation, we assume the communication time between any
416 two different clusters is higher than that within a cluster.
417 (4) All the tasks selected for execution on CPU or FPGA should satisfy the memory constraint
418 as follows:
419
420
421
$$\sum_{cl_i \in CL} \sum_{u=1}^{n} [MR_{u,cl_i} \times Z_{u,cl_i}] \leq TAM \tag{5}$$
422
423 In the above equation, MR_{u,Cl_i} denotes the memory footprints of individual tasks for the
424 respective cluster and TAM denotes the total available memory budget.
425 (5) The tasks placed in the FPGA cluster should satisfy the logic area constraint, i.e., the sum
426 of the area requirements in logic cells (LC_u) of the tasks (T_u) should be less than the total
427 available logic budget (TLC).
428 This constraint can be represented as:
429
430
$$\sum_{u=1}^{n} [LC_u \times Z_{u,cl_F}] \leq TLC \tag{6}$$
431
432 (6) In order to avoid overlapping between tasks executing at the same PEs, the following
433 inequalities need to be satisfied: $\forall(\langle T_u, T_v \rangle) \in \mathcal{A}$, where $u \neq v$,
434
$$Y_{uv} + Y_{vu} > 0 \tag{7}$$
435
436
$$Y_{uv} + Y_{vu} \leq 1 \tag{8}$$
437
$$S_u + \sum_{cl_i \in CL} [ET(u, cl_i) \times Z_{u,cl_i}] \leq S_v + (1 - Y_{uv}) \times M \tag{9}$$
438
439 Eq. (9) avoids time-wise overlap of any pair of tasks on the same cluster, i.e. T_v should start
440 after completion of T_u, if T_u is the predecessor of T_v. If tasks are executed in reverse order,
441

442 we use big-M nullification for deactivating the constraint. M has been considered as high
443 positive integer.
444 (7) ***Objective:*** The objective of the formulation is to choose a feasible solution which minimizes
445 finish time of the sink task. This is formulated as:

$$Minimize \ [S_n + \sum_{cl_i \in CL} (ET(n, cl_i) \times Z_{n,cl_i})] \tag{10}$$

Table 1. Complexity of ILP formulation constraints

Equation	# Constraints	# Variables Per Constraints
Eq. (1)	$O(n)$	$O(m)$
Eq. (2)	$O(1)$	$O(m)$
Eq. (4)	$O(\|E\|)$	$O(m)$
Eq. (5)	$O(n \times m)$	$O(n \times m)$
Eq. (6)	$O(1)$	$O(n)$
Eq. (9)	$O(n^2)$	$O(m)$

Complexity analysis: We present the complexity analysis for our ILP in Tab. 1. The second column of this table lists the upper bound of the number of constraints for each equation. The unique resource constraint in Eq. (1) should be determined for all n tasks, hence, for a given PTG, overall n constraints will be required. Similarly, the number of variables for this constraint can be represented as $O(m)$, where m denotes the total number of PEs in the system including all clusters. For the deadline constraint in Eq. (2), this condition should be checked for a single sink node, and thus, only $O(1)$ constraints will be required. In this way, the total complexity of ILP (in terms of the number of constraints) can be represented as $O(n^2)$. It may be noted that the complexity of ILP is independent of the number of PEs in a platform and deadline of a PTG.

4 HEURISTIC BASED MAPPING AND SCHEDULING STRATEGY

Due to its drawbacks and scalability issues, obtaining optimal solutions for mapping and scheduling through ILP is not always feasible, especially if the DSE demands multiple quick design iterations. Heuristic based approaches are a common way to trade off the shortcoming of ILP for near optimal solutions. As heuristic based approaches can be tailored for different requirements, we will introduce two heuristics to cover the goals of the ILP based methodology. The first heuristic (discussed in Section 4.1) is utilized to obtain a feasible mapping of tasks between the SW and HW domain, such that the resource constraints are satisfied. In our case study specifically, this means deciding if a task is executed on the CPU as SW, or in the FPGA fabric as HW. The second heuristic (discussed in Section 4.2) is utilized in order to obtain a scheduling in time, such that the task graph is executed within the deadline and the task graph dependencies are satisfied. It should be noted, that our heuristic based approach is tailored towards a mapping and scheduling that utilizes one CPU as the computing element for the SW execution. Furthermore, this approach can be utilized as a template for more complex heuristics with different system properties. Lastly, splitting the heuristic into multiple smaller algorithm is not the only way to obtain a feasible solution for our problem description.

491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539

4.1 HW/SW Task Mapping Heuristic

In multimedia applications, applications are split into several tasks with similar functions to increase the system throughput by executing these tasks concurrently using different cores. Similarly, for the given problem, the DAG represents a stream application with some tasks and a multicore system with one SW and one HW PE.

The partitioning and mapping problem discussed in this paper can be formulated as a minimization problem, aiming to determine maximum throughput subject to area constraint for the FPGA as follows:

$$Minimize\ max(HW_t, SW_t) \tag{11}$$

$$\text{Subject to} : \forall j \mid \sum_{i=1}^{|T|} [LC_i \times x_{ij}] \leq \text{HW resource Budget} \tag{12}$$

$x_{ij} = 1$ if task is T_i is mapped on j^{th} PE and 0, otherwise. Here, HW_t and SW_t denote the execution time, if all the tasks execute on HW and SW respectively, which are formulated in the following equations as follows:

$$HW_t = \sum_{i=1}^{|T|} ET(i, j) \times x_{i,j} \mid j \in Cl_F \tag{13}$$

where j^{th} PE belongs to FPGA. Similarly, we can represent SW_t where the PE represent CPU.

$$SW_t = \sum_{i=1}^{|T|} ET(i, j) \times x_{i,j} \mid j \in Cl_c \tag{14}$$

From Eq. (11), it can be observed that the partitioning and mapping strategy wants to maximize throughput for a given area constraint without exhaustively exploring all possible partitioning. Higher throughput can be achieved if tasks can be executed in parallel across different cores. However, due to data dependency among tasks, they cannot be executed arbitrarily in parallel. Moreover, some tasks must be executed serially due to dependency constraints. Taking a clue from this, our proposed heuristic attempts to minimize this *mandatory serial execution time*, and eventually it will aid in satisfying our objective functions mentioned in the Eq. (11). Our proposed mapping heuristic is shown in Alg. 1.

540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565

Algorithm 1: Partitioning & Mapping Heuristic of MESSI

```
1
   Input  : Set of n tasks of DAG (T);
   ET(i, Cl_C): CPU execution time of T_i;
   ET(i, Cl_F): FPGA execution time of T_i;
   LC_i: Area utilization of T_i;
   TLC: Total area budget;
   Output: τ_s: Task set selected for CPU;
   τ_h: Task set selected for FPGA;
2  Initialize: τ_s = τ_h = ∅
3  Identify the critical path (CP) of the DAG;
4  Calculate the length of CP on CPU: CP_C = Σ_{T_i∈CP} ET(i, Cl_C);
5  Calculate the length of CP on FPGA: CP_F = Σ_{T_i∈CP} ET(i, Cl_F);
6  if CP_F < CP_C AND CP_F ≤ D_DAG AND Σ_{T_i∈CP} LC_i ≤ TLC then
7     τ_h = CP        // Assign all critical path tasks to FPGA
8  else if CP_C < CP_F AND CP_C ≤ D_DAG AND Σ_{T_i∈CP} MR_{i,Cl_C} ≤ TAM then
9  τ_s = CP        // Assign all critical path tasks to CPU
10 for all tasks T_i ∉ CP do
11    if Σ_{T_i∈τ_h} LC_i + LC_i ≤ TLC then
12       τ_h = τ_h ∪ {T_i} // Assign task to FPGA if feasible
13    else if Σ_{T_i∈τ_s} MR_{i,Cl_C} ≤ TAM then
14       τ_s = τ_s ∪ {T_i} // Assign task to CPU
15 Move to scheduling phase;
```

566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588

Tasks connected by edges in the DAG must be executed sequentially, because of the dependencies. Identifying the critical path in the DAG provide us the maximum length of tasks that should be executed in sequence to ensure logical correctness. This critical path determines the total execution time of these processes. Here, the critical path is the longest sequential portion of the task graph, i.e., the number of nodes in that path that require sequential execution. Alg. 1 first determines the total execution time of the critical path by executing all the task nodes in the CPU (Alg. 1 line 3) and then in the FPGA (Alg. 1 line 4). If the total execution time of the critical path is less if we execute all tasks in the FPGA rather than in a CPU, then all the tasks belonging to the critical path will be assigned to the FPGA, provided the area constraint is satisfied. If the total execution time is shorter than its FPGA counterpart, then the critical path will be executed in the CPUs. However, for both cases, it has to be ensured that the length of the critical path is less than the deadline D_{DAG}. Once the task parts are partitioned, it has to be checked whether the total available memory is able to accommodate the individual task's footprint together. Once the condition stands, MESSI will proceed with the scheduling phase.

Note, that even if FPGA tasks commonly might be faster than CPU tasks, this is not always the case. As will be shown in Section 6.3 and Tab. 3, tasks like map can be faster in on the CPU than the FPGA. Alg. 1 handles this situation correctly.

4.2 Scheduling Heuristics

From the partitioning strategy, once we partition tasks for CPU and FPGA, the scheduling algorithm generates a schedule for each task by assigning the tasks to the respective PEs at a particular

Algorithm 2: *Latest Possible Start Time Calculation*

Input:
i.The task graph $G(T, E)$
ii. τ_s/τ_h: selected version of each task T_i (from Alg. 1)
iii. Execution time of T_i
iv. D_{DAG}: The deadline of the task graph.
Output:
i. $LPST_i$: LPST of each task T_i

1 **for** $T_i \in T$ **do**
2   **if** *T_i is a sink task in PTG* **then**
3         $LPST_i = D_{DAG} - ET(T_i, PE)$; // From Alg. 1, we know T_i is placed on which PE
4   **else**
5   Calculate the minimum of the latest start times $min\ (ET(j, PE))\ \forall\ T_j \in Succ(T_i)$;
        `// Let task` T_{sc} `have the minimum value of the latest start times among all successors of` T_i
6         $LPST_i = LPST_{sc} - ET(T_i, PE)$;

time instant while maintaining the data dependency and associative constraint. However, for the scheduling heuristic, the most important question to answer is when to schedule the task by maintaining the inter-task dependency. So, at first, the algorithm needs to derive tasks' execution order. To find the execution order, our proposed heuristic calculates the parameter called *Latest Possible Start Time* (LPST) for each task. The LPST of a particular task T_i implies that T_i must be started at least by that time to avoid a deadline miss. Alg. 2 shows the algorithm for the LPST calculation. Once Alg. 2 returns all the LPST values of each task, our scheduling heuristic will set the priorities for each task based on these values to determine the execution order. From Alg. 2, it is evident that the value of the LPST of a task provides an estimate of the remaining computational demand before the sink task completes its execution (lines 3 & 6). Hence, for any given deadline bound, a relatively lower LPST of a task indicates a higher remaining processing requirement. Taking a clue from this, the proposed heuristic sorts the tasks in ascending order based on their LPST values. This sorted list is the representation of tasks' execution order by maintaining interdependency.

Once the execution is obtained, our scheduling heuristic iterates through each time step until either the deadline is reached or all the tasks have been scheduled, whichever is earlier. The proposed scheduling heuristic is described in Alg. 3. The algorithm can be divided into two parts, the HW task execution and the SW task execution. As we mentioned before, in the similar vein of modern resource-constrained embedded systems, our heuristic attempts to schedule tasks, considering one CPU for SW execution and one FPGA resource for HW task execution. However, it should be noted that scalability will not limit the effectiveness of the proposed algorithm, and it can be applied to higher numbers of PE as well. In the next paragraphs, we will first discuss the part of about HW task execution through the heuristics, followed by the part of the SW task execution.

HW task execution (Alg. 3 lines 2-17): Once the partitioning of tasks is completed by Alg. 1, this part of the schedule only deals with tasks designated for HW executions. However, as we have only one CPU, this CPU acts as the controller which transfers data between the memory of the CPU and the HW task memories. Therefore, at the beginning, the algorithm first checks whether the CPU is

Algorithm 3: Scheduling Heuristic of MESSI

Input:
i. Tasks' characteristic as SW or HW tasks
ii. Tasks' execution order obtained from Alg. 2; Assume tasks' are sorted in set T based on ascending LPST values
Output: Generated schedule

```
1  for t = 0; t ≤ D_DAG AND T ≠ NULL; t++ do
2      // ======== HW TASK EXECUTION ========
3      if CPU is free and T_i is not root node then
4          CBP = C2F_i;
5          /* CBP: an integer variable denoting CPU Busy Period which holds the remaining
              time required to finish the CPU to FPGA data transfer and vice versa */
6      if C2F_i == 0 AND Pred(T_i) data is available upon completion then
7          st_i = t; // assign the time stamp at which T_i started;
8          Execute T_i on FPGA for ET(i, FPGA) duration;
9          ET(i, FPGA)−;
10     if execution on FPGA is over (ET(i, FPGA) == 0) then
11         Ft_i = t; // assign the time stamp at which T_i finished;
12         Remove the task from set T;
13     if T_i's execution is over AND CPU is free then
14         CBP = F2C_i;
15     if (CBP==0) mark the cpu as free at t;
16     else
17         CBP = CBP − 1;
18     // ======== SW TASK EXECUTION ========
19     if CPU is free AND Pred(T_i) data is available upon completion then
20         st_i = t; // assign the time stamp at which T_i started;
21         Execute T_i on CPU for ET(i, CPU) duration;
22         CBP = ET(i, CPU);
23     if execution on CPU is over (ET(i, CPU) == 0) then
24         Ft_i = t; // assign the time stamp at which T_i finished;
25         Remove the task from set T;
26     if (CBP==0) mark the cpu as free at t;
27     else
28         CBP = CBP − 1;
```

free. Once the CPU is free, the scheduling heuristics will assign the task with no predecessors to the FPGA.

In Alg. 3 line 3-5, the heuristic describes that once a HW task T_i is ready for its execution on the FPGA and provided it is not the root node, it will wait until the data transfer (from its predecessors) is completed between the CPU and HW task's memory. We denoted the data transfer time from CPU to FPGA for T_i as $C2F_i$. We also defined an additional variable as CBP, which marks the *CPU Busy Period* (CBP). In line 5, the algorithm assigns data transfer time ($C2F$) as CBP, which in turn

687 provides the remaining data transfer time requirement of T_i and thus CBP becomes zero when the
688 data transfer to initiate T_i finishes. Once the data transfer is completed, the CPU is marked as free
689 and SW tasks can be executed on the CPU.
690 Once the data transfer is complete and if the data from the predecessors is available or if the
691 task doesn't have any predecessors (root node) then the algorithm will start executing the HW
692 tasks. The algorithm will mark the start time and will execute the tasks for the stipulated duration
693 denoted as $ET(i, FPGA)$ (lines 6-8). Once task execution is finished, the finish time will be marked
694 and the task will be removed from the task set (lines 10-12). After the task is completed the data
695 needs to be transferred to the CPU and in line 14, that time is denoted as $F2C_i$. The CPU will remain
696 busy for that period.
697
698 *SW task execution (Alg. 3 lines 18-28):* Once a task T_i is designated as a SW task, Alg. 3 will
699 attempt to schedule it at the proper time instant, utilizing results from Alg. 2. Initially, it will check
700 whether the CPU is available and all the predecessors of T_i are completed, and it has the required
701 data to start its execution. Once these checks has been carried out (line 19), T_i will start its execution
702 and the start time will be marked. T_i will be executed for the stipulated duration $ET(i, CPU)$. The
703 CBP variable will indicate if the CPU is busy for that duration (lines 20-22). Once the execution is
704 completed, the finish time will be marked (line 24) and the task will be removed from the task set.
705 The algorithm will continue its execution with other tasks until the terminating conditions are
706 reached. A few important points on the utilization of Alg. 2 can be stated as follows:
707 • The algorithm continues to consider tasks only when all its predecessor tasks have finished
708 their executions.
709 • Such task to PE assignments enable that the beginning of the task will be the latest finishing
710 time of its predecessors, including communication overhead.
711 • If a task has a single predecessor, then our scheduling algorithm will execute the task right
712 after the finishing time of its predecessor. When a task has multiple predecessors, we will
713 consider the predecessor which has the latest finishing time.
714 • The successor task will be assigned to the same processor that was assigned to its predecessor
715 with the latest finishing time.
716 • The algorithm uses a relative priority order amongst all tasks based on the tasks' LPST start
717 time, considering each task T_i. This priority list based on task's LPST times ensures that
718 inter-task precedence relationships are always satisfied (the LPST time of a predecessor
719 task is always less than the LPST times of all its successors).
720
721 ## 5 APPLICATION CASE-STUDY PRELIMINARIES
722 To evaluate MESSI, an application case-study featuring a realistic heterogeneous RT system is
723 specified and designed. Especially on the HW platform, there exist several choices in building
724 an overall system, which in turn has impact on the task implementation and execution. These
725 considerations are also highlighting the heterogeneity of these systems, as they will be tailored for
726 their use in an embedded system.
727 An important part is the FPGA which has to be chosen. It has to provide sufficient area to fit in a
728 processing system like an SoC and additional HW tasks. Commercially available FPGAs offer a
729 variety of additional features beyond the conventional programmable logic blocks and block RAM.
730 These readily available features (e.g., HW peripheral blocks or interconnects) span a spectrum
731 of possibilities for tailoring application specific computing solutions. For example, FPGAs like
732 the Xilinx Zynq 7000 Series [52] feature an integrated ARM Cortex-M9 dual core-processor with
733 a multichannel *Direct Memory Access* (DMA) controller and various SoC peripherals, while the
734 programmable FPGA logic contains additional blocks for DSP, high-speed transceivers and more.
735

Other commercial FPGA manufacturers like Intel (e.g., Arria V Series [2]), Lattice Semiconductor (e.g., Avant-E Series [36]) and Microsemi (e.g., SmartFusion2 Series [24]) offer similarly broad solutions with different pricing, features and integrated processors or an extensive library of IP cores. Depending on this FPGA choice, various aspects of the task mapping and scheduling can change. The ILP-based mapping and scheduling in MESSI allows for consideration of technological constraints and considerations as long as these can be formulated with an ILP constraint (e.g., different memory access times through various available technologies that impact memory and area usage differently).

As we can not exhaust all possible configurations of the options for various heterogeneous RT systems, we summarize the relevant practical considerations for various RT systems for which our proposed scheduling strategy applies. According to these practical considerations, we select and evaluate a specific configuration for our application case-study in the evaluation Section 6.

5.1 General Practical Considerations

Heterogeneous RT systems encounter various practical considerations that are not always easily formulated formally in terms of constraints. The following list provides a selection of relevant practical constraints, which depend on the actual system considered and focus on technical aspects with regard to communication between SW and HW tasks:

1. What are the capabilities and requirements of the embedded system?
   (a) Is there a shared memory?
   (b) Is DMA available?
   (c) If 1a and 1b are not available, where and how is the task related data be stored?
2. How is data transported or shared between the SW and SW tasks?
3. What interfaces will the HW tasks use? Considering the data transport, what interfaces are required for certain transportation methods?
4. How will tasks be notified to start, respectively how do tasks notify they are done?

This list is not meant to be a complete list of considerations, as the considerations significantly depends on the system and its execution environment. Depending on each of these points, the calculated schedule will deviate from the real execution taking place on the system. For example, there will be a transportation and synchronization overhead in the communication between the CPU and the task in the FPGA fabric that adds to the total execution in the schedule. This deviation can be very small or (depending on the system) being of significant relevance to the scheduling outcome. The technical implementation also has impact on the SW memory footprint (e.g., additional code and memory areas to manage DMA or other interfaces to share data and memory). Next to the method of implementing the SW as well as HW tasks, which itself offers various dimensions of optimization, these considerations can contribute to more strict or loose constraints and thus affect the mapping and scheduling strategy.

5.2 Technical System Considerations

The goal for our application case-study is to evaluate the viability of our approach on an actual heterogeneous RT system. Our target system combines an FPGA together with a soft-core CPU based SoC. This SoC provides a rudimentary set of peripherals required in embedded systems, while leaving sufficient memory space and FPGA fabric area for custom HW based tasks. Tasks that are implemented as SW are stored in the SoCs memory, while tasks that are implemented as HW are connected to the SoCs memory mapped bus system. We consider a bare-metal system that does not provide a DMA controller or dedicated shared memory regions between the soft-core and FPGA, i.e., the soft-core needs to copy the application data explicitly between the FPGA internal

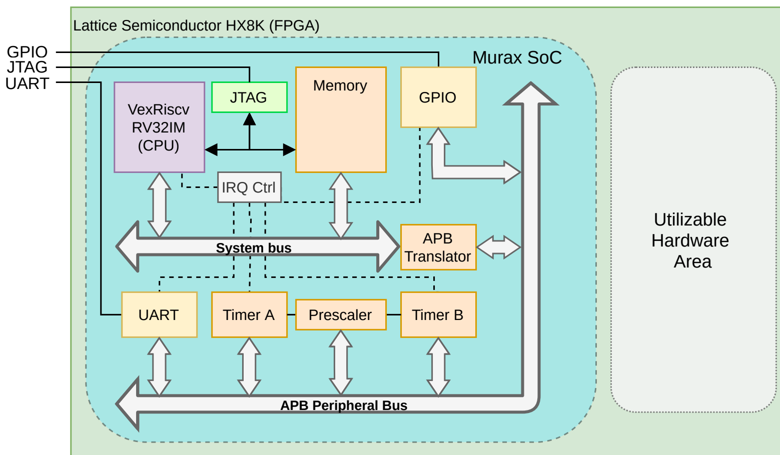
785 memory and CPU accessible memory. Moreover, we consider a bare-metal SW setting without
786 employing operating systems that might provide preemptive task scheduling capabilities. We
787 believe evaluating our proposed strategy on such a heterogeneous RT system is representative of
788 the embedded system domain. Furthermore, through the introduction of other techniques (e.g.,
789 DMA) the system's performance can only be improved.
790
791

6 EVALUATION: A RISC-V CASE-STUDY

792 This section presents results on the evaluation of our proposed scheduling and mapping strategy
793 and shows the achieved task execution and implementation strategy on a concrete heterogeneous
794 CPU+FPGA system, using an application case-study. We start with a description on the specific
795 choices with regard to the technical considerations, which constitute the setup of our evaluation
796 (Section 6.1). Then, the example application is introduced, and a corresponding implementation
797 sketch is provided (Section 6.2). Next, we present relevant metrics and the obtained mapping and
798 scheduling for the example application based on MESSI (Section 6.3). Finally, we present and discuss
799 the overall results in obtaining the calculated as well as executed mapping and scheduling and
800 elaborate how the system choice impacted the realization of the schedule (Section 6.4).
801
802

6.1 Setup

803 For this case-study we choose the Lattice Semiconductor HX8K FPGA [35] which is capable of
804 containing a SoC, whilst offering additional FPGA fabric area for HW tasks. Compared to other
805 commercially available FPGAs, the HX8K does not offer a built-in SoC or slices for DSP tasks like
806 multiply-accumulate. Within the technology of the HX8K, area is mainly determined through *Logic*
807 *Cells* (LCs). These LCs each contain a four-input look-up table, a D-flip-flop with optional enable
808 and reset controls and carry logic to interconnect with other LCs. Additionally, the HX8K FPGA is
809 compatible with the open source tool chain IceStorm [9], which includes the open source synthesis
810 tool Yosys [47].
811



827 Fig. 2. VexRiscv based Murax SoC on HX8K FPGA as baseline system configuration.
828

829 As a SoC, we choose the Murax SoC (see Fig. 2). The Murax SoC uses a SpinalHDL [7] based
830 RISC-V [45, 46] implementation called VexRiscv [8]. The VexRiscv processor is known for its high
831 degree of configurability, while minimizing the overhead of the generated code, thus resulting
832 in very small FPGA-compatible RISC-V CPUs, whilst suiting the requirements for RT embedded
833

system tasks. If required, more powerful configurations to increase the maximum frequency at which the processor runs, as well as features to boot operating systems like Linux. Murax SoC uses a small, pipelined 32 bit RISC-V single core with a lightweight main bus system and an adapter for the APB bus [21] for peripherals, making this SoC is representative for other embedded systems and microcontroller units. All tasks are implemented in C for SW execution on the RISC-V processor and in SpinalHDL as RTL description for HW task execution. SpinalHDL is an emerging language for HW description and generation that can be used to describe HW generators as well as traditional RTL descriptions. Various first-class language elements and language libraries improve the development cycle, thus improving the quality of the HW descriptions. The SpinalHDL-based descriptions can be used to generate either Verilog or VHDL code. As the HW tasks are described with SpinalHDL an easy integration into the Murax SoC is ensured. The complete development tool chain is based on open source tools and allows for static, simulation based, and FPGA emulation based analysis. The main, but not only, simulation backend in SpinalHDL is Verilator [40]. Verilator is used to obtain a cycle- and synthesis-accurate RTL simulation to extract the metrics like the execution times of the tasks. With the extracted metrics, the task graph and the constraints, the scheduling strategy can return a static schedule fulfilling the constraints. This obtained schedule is then realized through a main RISC-V SW, in which SW and HW task execution is orchestrated and interleaved. The execution of the obtained schedule is measured on the FPGA and through synthesis, place and route, and the cycle- and synthesis-accurate RTL simulation to compare the calculated result with the experimental result. With these results, we discuss some boundaries of MESSI with regard to the practical considerations in Section 5.1.

Shared memory architectures and DMA for effortless data sharing between the CPU and a HW task are not part of Murax SoC. This is due to the goal of Murax SoC fitting in small FPGAs such as the HX8K FPGA (and even smaller variants of the same FPGA-family [35] of Lattice Semiconductor). Thus, we have a low-level bare-metal embedded system for our application case-study, representing a FPGA-based heterogeneous RT system. We think this choice is appropriate for a case-study in the embedded system domain. Moreover, our method is also compatible with embedded systems that provide more features (like DMA, more cores, etc.) on the FPGA or the SoC, and can lead to improved results and better usability of the proposed technique. Furthermore, it should be noted that there exist whole bodies of research regarding optimization of SW and HW implementations and the automatic translation from high level specification towards SW and HW. While these topics are compatible with our strategy, their utilization is out of the scope of this work.

For this application case-study, each HW task is designed with its own small memory section, if required. The memory section is multiplexed between the memory mapped bus and the task itself. After storing the initial data in the task memory, the CPU will trigger the task's execution. The task's memory interface provides signals that represent the address, write data, read data and a write-enable. The task is controlled through a valid and a ready signal. If the valid signal is asserted, the tasks will start its processing with the provided parameters and data. Once the task is finished, the ready flag will be asserted by the task and the task's memory is multiplexed back to the memory mapped bus. The ready flag can either be used to trigger an interrupt or it will be read before accessing it. After the task's execution, the CPU can read all resulting data from the task's memory. Additional configuration inputs are mapped to memory mapped registers.

Table 2. Baseline of the case-study setup, Lattice Semiconductor HX8K FPGA with VexRiscv, Murax SoC.

Description	Maximum available	Used	Available
Memory usage / Bytes	4096	904	3192
Area usage / LC	7680	2820	4860

883 With the HX8K FPGA and the VexRiscv based Murax SoC as our heterogeneous system in place,
884 the baseline for the available HW area and memory can be determined. Tab. 2 shows the baseline
885 values for the memory usage in Bytes and the area usage in LCs. These values declare the maximum
886 budget of the memory and area that are available.
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902

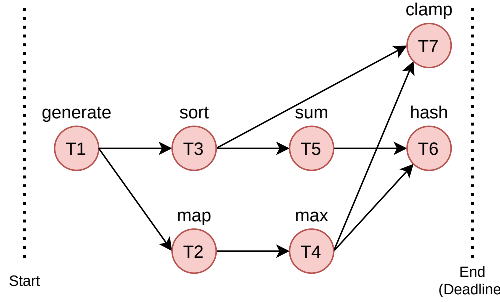


Fig. 3. Task graph for the case-study example application.

6.2 Application and Implementation

903 The task graph for the example application of this case-study is shown in Fig. 3. This task graph
904 contains seven different tasks, with one source task (T1, generate) and two sink tasks (T6, hash
905 and T7, clamp). The tasks represent data flow operations known from digital signal processing and
906 functional programming. Generate (T1) is used to generate vectors with pseudorandom values by
907 utilizing a given seed value. Map (T2) transforms a vector into a vector and applies a function to
908 each value of that vector. Sort (T3) sorts the values of a vector. Max (T4) obtains the maximum
909 value of a vector, thus transforming a vector into a scalar value. Sum (T5) calculates the sum of each
910 element of a vector, thus transforming a vector into a scalar value. Hash (T6) applies a mathematical
911 function to two scalar values and returns a single scalar value. Clamp (T7) takes a vector and clamps
912 each value between two given scalar values. Hence, the task graph combines vector and scalar
913 operations. In this case study, we utilize vectors of the size 16, which could be like the amount of
914 data samples the complete task graph has to process until the deadline. In Fig. 3 the data flow can
915 be described as follows: Generate (T1) is utilized to obtain the data to process, the fork in the task
916 graph means the same data is provided as a copy to Map (T2) and Sort (T3). Along the bottom path
917 of the task graph, Map (T2) transforms the vector, Max (T4) obtains the maximum value in the
918 vector, and the maximum value is used as an input in Hash (T6) as well as the two scalar inputs
919 for Clamp (T7). For Clamp (T7) the maximum value is provided as negative and positive number,
920 respectively. Along the top path of the task graph, Sort (T3) transforms the vector, Sum (T5) obtains
921 the sum of all values to pass it to Hash (T6), while the sorted vector is passed to Clamp (T7).
922 A directed edge in the task graph represents a dependency on the output/input of another task.
923 Therefore, a task can only be executed if and only if the required data is available. For example:
924 Task T2 (map) can only be executed if the data from task T1 (generate) is available. This results in
925 constraints on the order in which the tasks can be executed. At the same time, these tasks can be
926 implemented into a HW description by hand, to evaluate the feasibility of the implementation step
927 of the top level flow from Fig. 1. For each task an implementation, in both C and SpinalHDL, is
928 created and measured for their metrics such as execution time, area consumption after synthesis,
929 SW memory footprint and transportation time of the data between CPU and FPGA fabric. It has to
930 be noted, that tasks HW still require memory for their firmware drivers.
931

The task graph structure already implies requirements with respect to the technical implementation. For example: Task T1 generates data that is used in task T2 and T3 distinctly. Passing the data from and to the tasks T2 and T3 have to be handled as part of the scheduling. A fork in this sense also means that the output data from T1 has to be copied to be available for both tasks independently (e.g., `memcpy()` on an array of data).

Furthermore, a directed edge in the graph can represent three different types of data transactions:

(1) A task in SW is succeeded by a task in HW, and data is moved from the SW task to the HW task.
(2) A task in HW is succeeded by a task in HW, and data is moved from one HW task to another HW task.
(3) A task in HW is succeeded by a task in SW, and data is moved from the HW task to the SW task.

These three cases will look different in the realization of the schedule and their implementation varies based on the features of the embedded system too (e.g., if a DMA is available).

In general, our architecture requires the SW code to access the memory mapped registers via the system bus. This type of access is an essential part of the RISC-V architecture as well as many other embedded systems, thus such transactions as mentioned above don't give rise to additional challenges.

Listing 1. Accessing the task interface through memory mapped registers.

```
// store all element of the array into the memory of the task
for (uint8_t i = 0; i < vecSize; i++) {
  TASK_MAX->MEM_ADDR = i;
  TASK_MAX->MEM_WDATA = inputData[i];
  TASK_MAX->MEM_WRENA = 1;
  TASK_MAX->MEM_WRENA = 0;
}
// start the task
TASK_MAX->VALID = 1;
// check ready flag of task until its done processing
while (!TASK_MAX->READY);
// load max value
maxVal = TASK_MAX->MAX_VALUE;
```

Listing 1 shows such an exemplary transaction between the CPU and the HW task. Lines 2 to 7 move data into the tasks' memory, line 9 starts the task and after line 11 retrieves the ready flag from the task, line 13 reads the result register of the task.

Compared to an approach with a DMA or shared memory, this approach requires manual copying and moving data to and from tasks in order to execute the tasks. It has to be noted that additional features such as DMA will minimize the memory footprint further for the HW tasks.

If preemption of tasks is included in the considered properties of MESSI, the active checking for the ready flag (see Listing 1 line 11) would be handled through interrupts.

6.3 Tasks Metrics, Mapping and Scheduling

In order to collect the aforementioned metrics (execution time, memory footprint, HW area usage), the tasks are implemented in SW and HW respectively. The SW tasks are implemented as C functions, which are called with their parameters and their return value is stored into a variable to be accessed by the next task. For the HW tasks, implemented in SpinalHDL, the SW implementations are used as reference models. Control flow elements from the SW task are implemented as finite state machines, while the data flow elements represent the data path of the circuit (i.e., finite state

machines with data path [34]). The compiler, synthesis, and place & route tools are utilized to determine the memory footprint of SW code, and HW area usage. The various runtimes of the tasks are determined through the simulation trace for cycle accurate timings.

Table 3. Task metrics of the example application with seven tasks and vector size 16. For completeness, we list the forking process of data in this table, as the execution time is relevant for the difference between a calculated and executed schedule. (N/A = Not applicable)

Task	Software (CPU)		Hardware (FPGA fabric)					
	Execution time / μs	Memory footprint / Bytes	Time / µs				Memory footprint / Bytes	Area Usage / LC
			Total execution	Transport CPU to FPGA	Task processing	Transport FPGA to CPU		
– (fork data)	32.33	N/A	N/A	N/A	N/A	N/A	N/A	N/A
T_1 (generate)	80.00	52	33.67	N/A	1.50	27.00	48	1295
T_2 (map)	39.92	36	61.08	29.08	1.50	26.67	108	1275
T_3 (sort)	310.50	76	99.42	29.17	41.50	26.75	88	1383
T_4 (max)	64.33	152	33.08	28.92	1.50	0.08	68	1305
T_5 (sum)	48.08	48	33.17	26.75	1.50	0.08	60	1220
T_6 (hash)	88.92	108	9.75	1.17	4.17	0.50	36	631
T_7 (clamp)	43.25	96	59.67	28.33	1.50	26.75	112	1556

Tab. 3 shows the measured task parameters of our example application. The table is split in three parts: The left column contains the tasks with their designator T_i, the middle column contains the obtained execution time and memory footprint of each task in SW, and the right column contains the obtained execution time, memory footprint and area usage of the HW tasks. The execution time of the HW tasks is further separated into their total execution time (counted from the first instruction that belongs to interacting with the HW task until the last instruction interacting with the HW task), the plain data transport time between the CPU (SW) and FPGA (HW) as well as FPGA (HW) to CPU (SW) and finally the actual task processing time in HW. Note that the columns for the times for each task don't have to add up to the column *Total Execution*, but are contained within these bounds. For further clarity, we added the data forking (task *fork data*) as a row in the table, as it consumes a notable amount of time compared to the tasks.

The task parameters from the SW and HW tasks are fed into our ILP formulation from Section 3.3. Together with the top-level constraints (e.g., deadline at 320 µs, area budget of 4800 LC) the CPLEX [25] solver, which we employ for ILP solving, generates an optimal task mapping and scheduling according to our ILP formulation. Solving the ILP problem for our case-study took around 800 s. Fig. 4 shows the calculated schedule for the tasks with the parameters from Tab. 3. Please note, that the time parameter on the x-axis is not true to scale, but is meant to show the results of the task mapping and scheduling in a compacted way. The tasks T2, T4 and T7 mapped to the CPU and the tasks T1, T3, T5 and T6 are mapped to be executed as HW tasks on the FPGA. The ILP based schedule and mapping calculated a runtime of 183 µs, which is far below the deadline of 320 µs. The additional HW area used is 4529 LC which also is below the budget of 4860 LC.

With this schedule, we can now use the mapping and scheduling for the SW and HW tasks and implement the top level schedule such that it executes the proposed solution. After the boot code of the SoC has completed, the proposed schedule is executed. Furthermore, we obtain a scheduling and mapping from the heuristic described in Section 4.

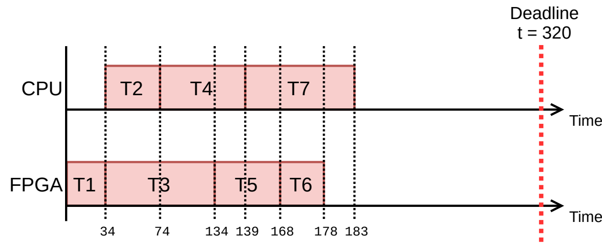


Fig. 4. Scheduling outcome from our proposed approach for the example application utilizing ILP based mapping and scheduling.

6.4 Results

Our results are twofold: First, we discuss the results related to the mapping and schedule obtained from our proposed strategy against the execution on the real system. This result will show, how MESSI applies to an application on a real system. Second, we compare the obtained metrics of the mapping and schedule with those of executing all tasks in SW and HW, respectively. This result puts the obtained solution by our proposed strategy into the context of traditional HW-SW co-design, in which an all SW or all HW solution is the starting point of the optimization.

With the obtained task mapping and scheduling order, shown in Fig. 4, we can execute the mapping and scheduling accordingly on the Murax SoC and HX8K FPGA. Hence, the mapping of

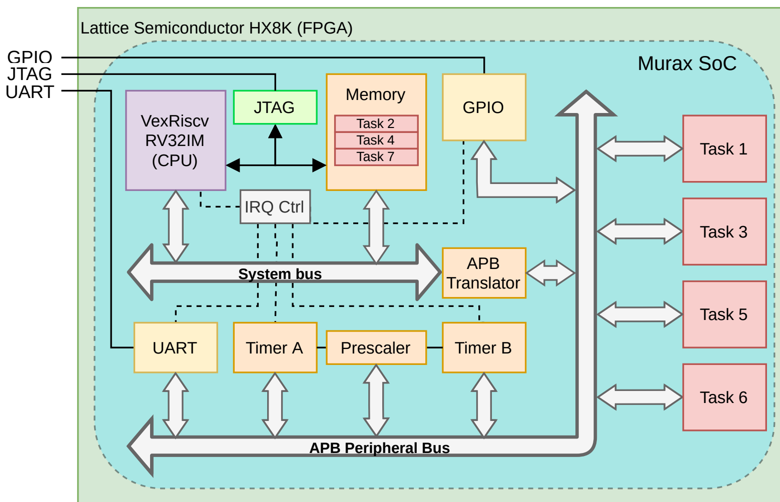


Fig. 5. Partitioned and mapped tasks from task graph for execution in software and hardware.

the tasks is configured and implemented in the system as shown in Fig. 5. The tasks T2, T4 and T7 (in the system's main memory) are executed in SW, while tasks T1, T3, T5 and T6 (attached as peripherals on the APB bus) are executed in HW.

As can be seen in Fig. 4, the calculated schedule would require preemptive task execution with discontinuities during Tasks 4 (at time 134 µs) and 7 (at time 168 µs and 178 µs). Furthermore, the ILP-based mapping and schedule does not contain a consideration for the data forking discussed earlier. As MESSI contains a heuristic to complement the trade-offs ILP-based mapping and scheduling has,

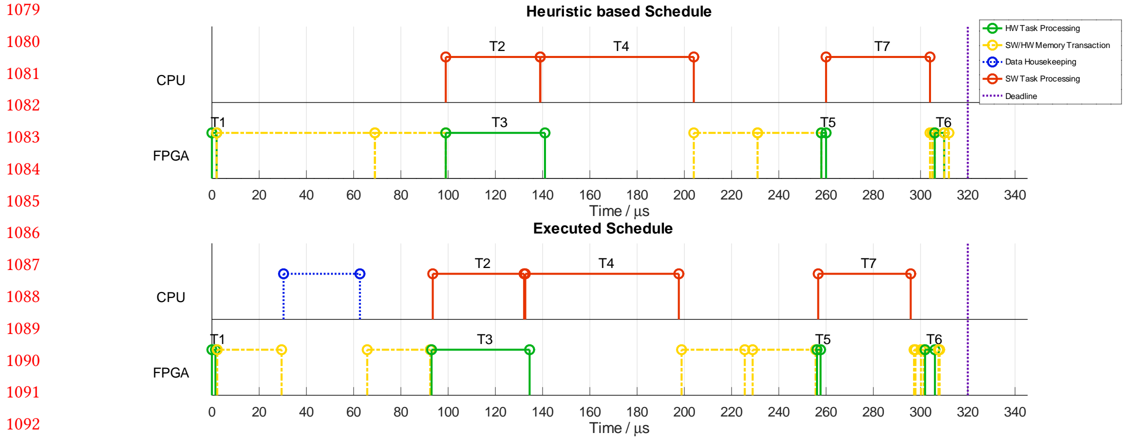


Fig. 6. Compared schedule results for application study. **Top:** calculated schedule (also refer to Fig. 4), **Bottom:** executed schedule implemented in application study.

we will discuss the comparison with the heuristic based solution in further detail. The heuristic will accommodate for the aforementioned differences of the ILP and chosen heterogeneous system.

Fig. 6 shows the schedules for the heuristic based solution and the executed schedule for comparison. The top plot shows the calculated schedule from our proposed scheduling and mapping strategy. The bottom plot shows the real execution of the schedule on the heterogeneous system. The events and their timestamps are reconstructed from a wavetrace, the source code and the disassembly of the implemented schedule. The deadline of the application is marked with a dotted line (purple) at the time 320 µs. Each task is annotated with a task identifier corresponding with Tab. 3. The top half of each plot show the execution traces of the tasks on the CPU part of the system (SW tasks) (red and blue events). For the SW tasks, the strike through events (red) show the task execution on the CPU, while the dashed event (blue) is the execution of housekeeping data (i.e., *fork data*). This is required, as for example T3 and T2 both require the same data from T1, thus it needs to be copied once. The bottom parts of each plot show the execution traces for the HW tasks on the FPGA fabric (yellow and green events). For the HW tasks, the strike through events (green) show the task execution on the FPGA fabric, while the dashed (yellow) events are data transmission for the tasks. Hence, the HW task execution is broken down to data transportation and task processing. This is required, as our application case-study leverages an embedded system without shared memory or DMA for devices on the memory bus. It can be seen from the figure, that the executed schedule differs slightly from the heuristic based solution. Copying the data (i.e., *fork data* task) is visibly executed on the CPU as SW code, while the heuristic considers this as part of the task T1 (generate). This will cause the both tasks to end up requiring similar amount of time. Further differences in the two schedules come from the real code execution on an SoC, in which some portions of the code aren't fully related to a task (i.e., calculating or preparing addresses, saving the register file to the stack, etc.).

Next is the comparison of our obtained mapping and scheduling with all SW and all HW solution in terms of the utilized metrics, respectively. Tab. 4 shows a comparison of three schedules: The column *All Software* and *All Hardware* represent the non-optimal boundaries in which the schedule results of the ILP-based and heuristic based scheduling can be expected. For the schedules *All Hardware* and *All Software*, we kept the same sequential order, both such that they respect the

1128 Table 4. Proposed schedule in context to executing all tasks in software or hardware. Entries marked **bold**
1129 violate the deadline or resource constraints, respectively.
1130

Property \ Schedule	All Software	Proposed	All Hardware
Memory Footprint (complete) / Bytes	1692	1676	1772
Memory Footprint (no boot code) / Bytes	788	772	868
Area Usage (complete) / LC	2820	7351	**11251**
Area Usage (w/o SoC) / LC	0	4531	**8431**
Total Execution Time / μs	**711**	315.25	**325.75**

1131
1132
1133
1134
1135
1136
1137
1138
1139 dependencies on the task graph of the application. The memory footprint and the area usage are
1140 declared twice. In the rows with *(complete)* annotation, the absolute size in terms of Bytes and LCs
1141 is shown. For the memory footprint, this includes the boot code, which would be part of every
1142 SW application. For the area usage, this includes the baseline area usage of the Murax SoC on
1143 the HX8K FPGA. The other rows show the values for just the SW and HW solution of the tasks,
1144 respectively. These values are calculated as the difference to the baseline values of the embedded
1145 system from Tab. 2. The measurements were obtained through the compiler tool chain, synthesis,
1146 and place & route tools. It has to be noted, that differences in the sums of columns *Memory Footprint*
1147 and *Area Usage* in Tab. 3 come from optimizations that the tools can introduce on the SW and HW
1148 design, respectively.
1149 The *All Software* schedule requires no additional HW, while the *All Hardware* requires 8431 LC,
1150 or 11 251 LC for (*complete*), implementing all tasks in HW. Hence, the *All Hardware* schedule is not
1151 realizable with the area budget. Our proposed mapping and schedule requires an area utilization of
1152 4531 LC, or 7351 LC for (*complete*), thus being 1.86× better (46.3 % reduction in area usage) than the
1153 *All Hardware* schedule. For the *All Software* schedule, the memory usage is 788 Byte, or 1692 Byte
1154 for (*complete*). The *All Hardware* schedule requires 868 Byte, or 1772 Byte for *(complete)*, of code, in
1155 order to interact with the HW tasks and move the task data around. Our proposed mapping and
1156 schedule requires a memory usage of 772 Byte, or 1676 Byte for (*complete*). Hence, our proposed
1157 strategy requires 96 Byte less than the *All Hardware* (11.1 % decrease) schedule and 16 Byte less
1158 than *All Software* (2.0 % decrease), while requiring much less area of the FPGA fabric (3900 LC less
1159 than *All Hardware*). The *All Software* schedule executes in 711 µs which is 391 µs more than the
1160 deadline of 320 µs. The *All Hardware* schedule executes in 325.75 µs which is 5.75 µs more than
1161 the deadline of 320 µs. Without any level of parallel execution involved, it can be seen that the *All*
1162 *Hardware* schedule is ×1.18 better (54.2 % reduction) than the *All Software* schedule. Our proposed
1163 mapping and schedule executes within 315.25 µs, which is within the deadline of 320 µs. Comparing
1164 our proposed mapping and scheduling with the *All Software* schedule, our result is ×1.25 better
1165 (55.6 % reduction).
1166 In summary, while the *All Software* fits in terms of memory usage and area utilization, the
1167 deadline of 320 µs is exceeded. Furthermore, the *All Hardware* schedule fits the memory usage as
1168 well, but exceeds the area budget of 4860 LC and the deadline 320 µs. Finally, our proposed schedule
1169 provides improvements in memory usage (11.1 % versus *All Hardware* and 2.0 % versus *All Software*)
1170 and area utilization (46.3 % versus *All Hardware*), while executing the task graph within 315.25 µs
1171 (improving by 55.6 % versus *All Software* and 3.2 % versus *All Hardware*).
1172
1173

7 DISCUSSION AND FUTURE WORK

1174 The results shown in Fig. 4 and Fig. 6 show differences in how the schedule is executed on the
1175 embedded system. First, the ILP creates a solution (Fig. 4) that does not fully cover all aspects of
1176

the system. The utilized ILP constraints don't consider that the data generated by task T1 needs to be copied (see blue time interval in Fig. 6), hence requiring additional time (which is not part of the task itself). Furthermore, the solution would require interrupting the task execution of T4 and T7. Moreover, the ILP is not considering the memory transfers (yellow time intervals in Fig. 6) between the SW and HW domain as separate operation that can be delayed (e.g., the memory transfer of T3 in Fig. 6 from FPGA to CPU occurs after T4 finishes and frees the CPU). Such architectural considerations and constraints are not part of the ILP constraints and thus are not part of the calculated schedule. The advantage of the ILP-based mapping and scheduling is that, at this point, we could refine our constraints to represent our system architecture.

Further refinements can be formulated on the generic set of ILP constraints provided by MESSI, or additional development cycles (as shown in Fig. 1) can employ more specific ILP constraints in order to accommodate the requirements. This might be useful if different task graphs based around the same set of tasks are explored and compared. But such additional ILP constraints are specific to the properties of the underlying embedded system (refer to Section 5.1, Section 5.2 and Section 6.1) as well as the tasks graph and tasks of the application. The set of ILP constraints already provided in this paper deliver a set of common scheduling constraints found in many RT applications. Therefore, the ILP-based mapping and scheduling can provide early estimations independent of the underlying system architecture while being adaptable for refinement due to more specific system details. Hence, ILP-based mapping and scheduling require additional refinement cycles, which can also lead to further issues of scalability (see Tab. 1). Additions to the ILP formulation naturally increase the complexity (see Section 3.3), thus should be done with caution. When we applied the given ILP formulation to our application, our complexity analysis hints towards high computational overhead (in terms of run time of the ILP solver) as the number of nodes in a PTG and/or the number of resources increase. For our task graph, it has been experimentally observed that our proposed ILP formulation takes around 4 hours to find feasible schedules for a PTG with circa 25 nodes on a platform with two heterogeneous PE. Hence, reiterations and small changes will lead to longer DSE, making the heuristic approach more practical than the ILP formulation.

While general task graph scheduling is NP-complete, our approach demonstrates effectiveness for both tractable and complex instances. Firstly, our case study, though hypothetical, reflects the reality of many embedded systems with relatively simple task graphs and fixed processors, where polynomial-time solutions are feasible. This showcases the practical applicability of MESSI in common scenarios. Secondly, the scalability analysis of our ILP formulation (Tab. 1) demonstrates its ability to handle increasingly complex task graphs. The observed trend, though exhibiting non-linearity as expected for ILP, remains manageable due to the formulation's independence from deadline and PE count. This scalability is crucial for tackling NP-complete instances where exhaustive search is impractical. Furthermore, recent work by Senapati et al. [38] highlights the challenges of scheduling multiple periodic DAGs on heterogeneous systems. Their findings emphasize the need for efficient scheduling techniques that can handle complex dependencies and real-time constraints, further validating the relevance of our approach. While acknowledging the NP-completeness of general task graph scheduling, we believe that our approach, with its demonstrated scalability and focus on practical scenarios, provides a valuable contribution to the field. Lastly, ILP solvers are becoming stronger/more effective, so scalability issues can be improved using better ILP solvers for complex tasks.

Our additionally proposed heuristic provides further considerations and overcomes the scalability issues of the ILP-based approach. Comparing the mapping and scheduling generated by the heuristic, we can observe an improvement compared to the mapping and scheduling obtained with ILP. While the task *fork data* becomes part of the task T1 to keep the heuristic approach lightweight, we can see that the remainder of the task graph matches with only minor differences. These differences

come from the real SW execution, in which SW contains pieces of code that aren't directly part of a task.

Lastly, our proposed methodology could also be utilized to handle multiple applications, while not specifically tailored for this. For this, a global task graph consisting of the multiple applications can be created. Consecutive applications' task graphs must be virtually connected through their sink and source node, respectively. By adding up the separate deadlines of the applications to one global deadline, the problem can be mapped to our methodology. With both, the ILP formulation and our proposed heuristics, it is expectable that the complexity of the scheduling problem increases. This is further underlined, when considering that the overall graph structure becomes more complex.

For future work, we aim to consider further evaluations that involve different heterogeneous RT systems and different application examples. These systems should contain a range of different features (e.g., interrupts, DMA) to further investigate and expand on the general and technical considerations. Through more evaluations, we can refine MESSI further, to include more application specific properties and constraints. Additionally, we plan to investigate automating the implementation of tasks through *High-Level Synthesis* (HLS) in order to speed up the development and verification cycles. Using HLS allows for faster design space exploration and can aid in obtaining estimates for task metrics much faster. Lastly, we want to investigate the use of a *Virtual Prototype* (VP) as a reference model of a heterogeneous RT system. VPs allow early HW-SW co-design and verification, thus the possible refinement loop in the methodology can be achieved more efficiently.

8 CONCLUSION

In this paper, we propose MESSI, a static scheduling strategy for mapping and scheduling application tasks for heterogeneous RT systems. The strategy encompasses an ILP-based optimization of constraints, modeling the application's properties, as well as a system specific heuristic approach to overcome the disadvantages of ILP. Through the ILP constraints, we describe general scheduling properties (such as deadlines or preemption behavior) as well as relevant system architecture and application specific properties (such as HW area budget or SW memory limits). We proposed general practical and technological considerations that assist engineers in their decision-making process and in understanding the advantages, disadvantages as well as the limitations of the underlying architecture of the heterogeneous system. With a case-study we provide an evaluation through which we show the consequences that follow from considering specific systems decisions (e.g., no DMA or specific HW task interfaces). Our evaluation demonstrates the applicability of MESSI in providing practical results for a heterogeneous CPU+FPGA system. Furthermore, our evaluation shows how the mappings and schedules obtained through the ILP, a heuristic and the real task execution compare. Additionally, the obtained schedule is compared against the initial system configurations (i.e., *All Software*, *All Hardware*) that span the search space of HW-SW Co-Design. Finally, we provided ideas for future work to further boost our methodology and broaden the scope of our scheduling algorithm to consider more general and application specific constraints, as well as different system architectures.

ACKNOWLEDGMENTS

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project ECXL under contract no. 01IW22002, and by the Yerun Research Mobility Award (YRMA) scheme, UK Engineering and Physical Sciences Research Council (EPSRC) through grant EP/V000462/1 and EP/X015955/1. For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising

1275 ## REFERENCES

1276 [1] Sallar Ahmadi-Pour, Sangeet Saha, Vladimir Herdt, Rolf Drechsler, and Klaus McDonald-Maier. 2022. Task Mapping and Scheduling in FPGA-based Heterogeneous Real-time Systems: A RISC-V Case-Study. In *2022 25th Euromicro Conference on Digital System Design (DSD)*. 134–141. https://doi.org/10.1109/DSD57027.2022.00027

[2] Intel Altera. 2023. Arria V FPGA & SoC FPGA. https://www.intel.de/content/www/de/de/products/details/fpga/arria/v.html. Accessed on 2023-11-28.

[3] Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Miriam Leeser, et al. 2022. The Future of FPGA Acceleration in Datacenters and the Cloud. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15, 3 (2022), 1–42.

[4] Giorgio C Buttazzo and Giorgio Buttanzo. 1997. *Hard real-time computing systems - Predictable Scheduling Algorithms and Applications*. Vol. 356. Springer.

[5] David Castells-Rufas, Vinh Ngo, Juan Borrego-Carazo, Marc Codina, Carles Sanchez, Debora Gil, and Jordi Carrabina. 2022. A survey of FPGA-based vision systems for autonomous cars. *IEEE Access* 10 (2022), 132525–132563.

[6] Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. 2020. FPGA-accelerated samplesort for large data sets. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 222–232.

[7] C.Papon. 2021. SpinalHDL. https://github.com/SpinalHDL/SpinalHDL. Accessed on 2022-03-24.

[8] C.Papon. 2021. VexRiscV. https://github.com/SpinalHDL/VexRiscv. Accessed on 2022-03-24.

[9] C.Wolf and M.Lasser. 2021. Project IceStorm. http://bygone.clairexen.net/icestorm/. Accessed on 2022-03-24.

[10] Zexi Deng, Dunqian Cao, Hong Shen, Zihan Yan, and Huimin Huang. 2021. Reliability-aware task scheduling for energy efficiency on heterogeneous multiprocessor systems. *The Journal of Supercomputing* 77, 10 (2021), 11643–11681.

[11] Ashutosh Dhar, Edward Richter, Mang Yu, Wei Zuo, Xiaohao Wang, Nam Sung Kim, and Deming Chen. 2021. DML: Dynamic Partial Reconfiguration with Scalable Task Scheduling for Multi-Applications on FPGAs. *IEEE Trans. Comput.* (2021).

[12] Bo Ding, Jinglei Huang, Junpeng Wang, Qi Xu, Song Chen, and Yi Kang. 2023. Task Modules Partitioning, Scheduling and Floorplanning for Partially Dynamically Reconfigurable Systems with Heterogeneous Resources. *ACM Trans. Des. Autom. Electron. Syst.* 28, 6, Article 103 (oct 2023), 26 pages. https://doi.org/10.1145/3625295

[13] Youssef Elmougy, Weiwei Jia, Xiaoning Ding, and Jianchen Shan. 2021. Diagnosing the Interference on CPU-GPU Synchronization Caused by CPU Sharing in Multi-Tenant GPU Clouds. In *2021 IEEE International Performance, Computing, and Communications Conference (IPCCC)*. IEEE, 1–10.

[14] Mohammed Elnawawy, Abid Farhan, Ahmad Al Nabulsi, Abdul-Rahman Al-Ali, and Assim Sagahyroon. 2019. Role of FPGA in internet of things applications. In *2019 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*. IEEE, 1–6.

[15] Andreas Emeretlis, George Theodoridis, Panayiotis Alefragis, and Nikolaos Voros. 2017. Static Mapping of Applications on Heterogeneous Multi-Core Platforms Combining Logic-Based Benders Decomposition with Integer Linear Programming. *ACM Trans. Des. Autom. Electron. Syst.* 23, 2, Article 26 (dec 2017), 24 pages. https://doi.org/10.1145/3133219

[16] Juan Fang, Jiaxing Zhang, Shuaibing Lu, Hui Zhao, Di Zhang, and Yuwen Cui. 2021. Task Scheduling Strategy for Heterogeneous Multicore Systems. *IEEE Consumer Electronics Magazine* 11, 1 (2021), 73–79.

[17] Joel Josephson and R Ramesh. 2019. A novel algorithm for real time task scheduling in multiprocessor environment. *Cluster Computing* 22, 6 (2019), 13761–13771.

[18] J Kokila, N Ramasubramanian, and Nagi Naganathan. 2019. Resource efficient metering scheme for protecting SoC FPGA device and IPs in IOT applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 10 (2019), 2284–2295.

[19] George Lentaris, Ioannis Stratakos, Ioannis Stamoulias, Dimitrios Soudris, Manolis Lourakis, and Xenophon Zabulis. 2019. High-performance vision-based navigation on SoC FPGA for spacecraft proximity operations. *IEEE Transactions on Circuits and Systems for Video Technology* 30, 4 (2019), 1188–1202.

[20] Martin Letras, Joachim Falk, Tobias Schwarzer, and Jürgen Teich. 2021. Multi-Objective Optimization of Mapping Dataflow Applications to MPSoCs Using a Hybrid Evaluation Combining Analytic Models and Measurements. *ACM Trans. Des. Autom. Electron. Syst.* 26, 3, Article 18 (dec 2021), 33 pages. https://doi.org/10.1145/3431814

[21] ARM Limited. 2003, 2004. AMBA 3 APB Protocol Specification v1.0. https://developer.arm.com/documentation/ihi0024/b/. Accessed on 2022-03-24.

[22] Xing Liu, Jianfeng Yang, Chengming Zou, Qimei Chen, Xin Yan, Yuao Chen, and Chenran Cai. 2021. Collaborative edge computing with FPGA-based CNN accelerators for energy-efficient and time-aware face tracking system. *IEEE Transactions on Computational Social Systems* 9, 1 (2021), 252–266.

[23] Joshua Mack, Samet E Arda, Umit Y Ogras, and Ali Akoglu. 2021. Performant, multi-objective scheduling of highly interleaved task graphs on heterogeneous system on chip devices. *IEEE Transactions on Parallel and Distributed Systems* 33, 9 (2021), 2148–2162.

[24] Microsemi Microchip. 2023. SmartFusion 2 SoC FPGA. https://www.microchip.com/en-us/products/fpgas-and-plds/system-on-chip-fpgas/smartfusion-2-fpgas. Accessed on 2023-11-28.

[25] Stefan Nickel. 2021. Ibm ilog cplex optimization studio. In *Angewandte Optimierung mit IBM ILOG CPLEX Optimization Studio*. Springer, 9–23.

[26] NVIDIA. 2015. NVIDIA Tegra X1. https://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf. Accessed on 2024-05-31.

[27] NVIDIA. 2022. NVIDIA Jetson AGX Orin Series - Technical Brief. https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf. Accessed on 2024-05-31.

[28] Bikash Poudel, Naresh Kumar Giri, and Arslan Munir. 2017. Design and comparative evaluation of GPGPU-and FPGA-based MPSoC ECU architectures for secure, dependable, and real-time automotive CPS. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 29–36.

[29] Reza Ramezani. 2021. Dynamic scheduling of task graphs in multi-FPGA systems using critical path. *The Journal of Supercomputing* 77 (2021), 597–618.

[30] Juan J Rodríguez-Andina, Maria D Valdes-Pena, and Maria J Moure. 2015. Advanced features and industrial applications of FPGAs—A review. *IEEE Transactions on Industrial Informatics* 11, 4 (2015), 853–864.

[31] Andrés Rodríguez, Angeles Navarro, Rafael Asenjo, Francisco Corbera, Rubén Gran, Darío Suárez, and Jose Nunez-Yanez. 2019. Exploring heterogeneous scheduling for edge computing with CPU and FPGA MPSoCs. *Journal of Systems Architecture* 98 (2019), 27–40. https://doi.org/10.1016/j.sysarc.2019.06.006

[32] Sangeet Saha, Shounak Chakraborty, Sukarn Agarwal, Rahul Gangopadhyay, Magnus Själander, and Klaus McDonald-Maier. 2022. DELICIOUS: Deadline-aware approximate computing in cache-conscious multicore. *IEEE Transactions on Parallel and Distributed Systems* 34, 2 (2022), 718–733.

[33] Sangeet Saha, Shounak Chakraborty, Sukarn Agarwal, Magnus Själander, and Klaus D McDonald-Maier. 2024. ARCTIC: Approximate Real-Time Computing in a Cache-Conscious Multicore Environment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).

[34] P.R. Schaumont. 2012. *A Practical Introduction to Hardware/Software Codesign*. Springer US. https://books.google.de/books?id=dgTx92SrFo0C

[35] Lattice Semiconductor. [n. d.]. iCE40 LP/HX Family Data Sheet. https://www.latticesemi.com/view_document?document_id=49312. Accessed on 2022-03-24.

[36] Lattice Semiconductor. 2023. Avant-E. https://www.latticesemi.com/Products/FPGAandCPLD/Avant-E. Accessed on 2023-11-28.

[37] Debabrata Senapati, Kousik Rajesh, Chandan Karfa, and Arnab Sarkar. 2023. TMDS: Temperature-Aware Makespan Minimizing DAG Scheduler for Heterogeneous Distributed Systems. *ACM Trans. Des. Autom. Electron. Syst.* 28, 6, Article 99 (oct 2023), 22 pages. https://doi.org/10.1145/3616869

[38] Debabrata Senapati, Arnab Sarkar, and Chandan Karfa. 2022. Energy-aware real-time scheduling of multiple periodic dags on heterogeneous systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 8 (2022), 2447–2460.

[39] Mitali Sinha, Gade Sri Harsha, Pramit Bhattacharyya, and Sujay Deb. 2021. Design Space Optimization of Shared Memory Architecture in Accelerator-Rich Systems. *ACM Trans. Des. Autom. Electron. Syst.* 26, 4, Article 30 (mar 2021), 31 pages. https://doi.org/10.1145/3446001

[40] Wilson Synder. 2003-2022. Verilator. https://veripool.org/verilator/. Accessed on 2022-03-24.

[41] Yu-Chu Tian and David Charles Levy. 2022. *Handbook of real-time computing*. Springer Nature.

[42] Jeffrey D. Ullman. 1975. NP-complete scheduling problems. *Journal of Computer and System sciences* 10, 3 (1975), 384–393.

[43] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. 2019. Heterogeneous Resource-Elastic Scheduling for CPU+FPGA Architectures. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies* (Nagasaki, Japan) *(HEART '19)*. Association for Computing Machinery, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/3337801.3337819

[44] Zishen Wan, Bo Yu, Thomas Yuang Li, Jie Tang, Yuhao Zhu, Yu Wang, Arijit Raychowdhury, and Shaoshan Liu. 2021. A survey of fpga-based robotic computing. *IEEE Circuits and Systems Magazine* 21, 2 (2021), 48–74.

[45] Andrew Waterman and Krste Asanović (Eds.). 2019. *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*.

[46] Andrew Waterman and Krste Asanović (Eds.). 2019. *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*.

[47] Wolf, C., Glaser, J., and Kepler, J. 2013. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*.

[48] Yulong Wu, Weizhe Zhang, Nan Guan, and Yehan Ma. 2023. TDTA: Topology-based Real-Time DAG Task Allocation on Identical Multiprocessor Platforms. *IEEE Transactions on Parallel and Distributed Systems* (2023).

[49] Nicholas Wulf, Alan D George, and Ann Gordon-Ross. 2016. A framework for evaluating and optimizing FPGA-based SoCs for aerospace computing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 10, 1 (2016),

1–29.

[50] Guoqi Xie, Junqiang Jiang, Yan Liu, Renfa Li, and Keqin Li. 2017. Minimizing energy consumption of real-time parallel applications using downward and upward approaches on heterogeneous systems. *IEEE Transactions on Industrial Informatics* 13, 3 (2017), 1068–1078.

[51] Guoqi Xie, Hao Peng, Xiongren Xiao, Yao Liu, and Renfa Li. 2021. Design Flow and Methodology for Dynamic and Static Energy-Constrained Scheduling Framework in Heterogeneous Multicore Embedded Devices. *ACM Trans. Des. Autom. Electron. Syst.* 26, 5, Article 36 (jun 2021), 18 pages. https://doi.org/10.1145/3450448

[52] AMD Xilinx. 2023. Zynq-7000 SoC. https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. Accessed on 2023-11-28.

[53] Hongzhi Xu, Renfa Li, Chen Pan, and Keqin Li. 2019. Minimizing energy consumption with reliability goal on heterogeneous embedded systems. *Journal of Parallel and distributed Computing* 127 (2019), 44–57.

[54] Jinyi Xu, Kaixuan Li, and Yixiang Chen. 2022. Real-time task scheduling for FPGA-based multicore systems with communication delay. *Microprocessors and Microsystems* 90 (2022), 104468. https://doi.org/10.1016/j.micpro.2022.104468

[55] Serif Yesil and Ozcan Ozturk. 2022. Scheduling for heterogeneous systems in accelerator-rich environments. *The Journal of Supercomputing* 78, 1 (2022), 200–221.

[56] Chaoyu Zhang, Hexuan Yu, Yuchen Zhou, and Hai Jiang. 2021. High-Performance and Energy-Efficient FPGA-GPU-CPU Heterogeneous System Implementation. In *Advances in Parallel & Distributed Processing, and Applications*. Springer, 477–492.

[57] Tao Zhang, Ganjun Liu, Qianyu Yue, Xin Zhao, and Mengyang Hu. 2018. Using Firework Algorithm for Multi-Objective Hardware/Software Partitioning. *IEEE Access* 7 (2018), 3712–3721.

[58] Xiaofan Zhang, Yuan Ma, Jinjun Xiong, Wen-Mei W Hwu, Volodymyr Kindratenko, and Deming Chen. 2021. Exploring HW/SW co-design for video analysis on CPU-FPGA heterogeneous systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 6 (2021), 1606–1619.

[59] Zongwei Zhu. 2019. A hardware and software task-scheduling framework based on CPU+ FPGA heterogeneous architecture in edge computing. *IEEE Access* 7 (2019), 148975–148988.

Received 00.00.00; revised 00.00.00; accepted 00.00.00