Task Mapping and Scheduling in FPGA-based Heterogeneous Real-time Systems: A RISC-V Case-Study

Sallar Ahmadi-Pour
Institute of Computer Science,
University of Bremen
Bremen, Germany
sallar@uni-bremen.de

Sangeet Saha
University of Essex
Essex, United Kingdom
sangeet.saha@essex.ac.uk

Vladimir Herdt
Institute of Computer Science,
University of Bremen
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
vherdt@uni-bremen.de

Rolf Drechsler
Institute of Computer Science,
University of Bremen
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
drechsler@uni-bremen.de

Klaus McDonald-Maier
University of Essex
Essex, United Kingdom
kdm@essex.ac.uk

***Abstract*—Heterogeneous platforms, that integrate CPU and FPGA-based processing units, are emerging as a promising solution for accelerating various applications in the embedded system domain. However, in this context, comprehensive studies that combine the theoretical aspects of real-time scheduling of tasks along with practical runtime architectural characteristics have mostly been neglected so far.**

To fill this gap, in this paper we propose a real-time scheduling algorithm with the objective of minimizing the overall execution time under hardware resource constraints for heterogeneous CPU+FPGA architectures. In particular, we propose an *Integer Linear Programming* (ILP) based technique for task allocation and scheduling. We then show how to implement a given scheduling on a practical CPU+FPGA system regarding current technology restrictions and validate our methodology using a practical RISC-V case-study. Our experiments demonstrate that performance gains of 40% and area usage reductions of 67% are possible compared to a full software and hardware execution, respectively.

I. Introduction

The increasing demand for computing capabilities, especially in artificial intelligence, multimedia technology and high-performance computing, has led to a paradigm shift towards heterogeneous computing architectures. Such architectures include *Central Processing Units* (CPUs), *Graphics Processing Units* (GPUs), *Application-Specific Integrated Circuits* (ASICs) and even reconfigurable devices such as *Field Programmable Gate Arrays* (FPGAs). Among these, FPGAs allow the adaptation of high-performance application-specific hardware at a lower cost. Meanwhile, CPU+FPGA heterogeneous platforms are also emerging as a promising solution for accelerating various applications in many of today's safety-critical real-time embedded systems, such as automotive systems [1].

For example, in a given complex safety-critical system, CPUs offer satisfying performance over wide-ranging classes of tasks. FPGAs can provide high performance for computation-intensive tasks with hard deadlines by exploiting inherent parallelism. Therefore, efficient execution of hard real-time tasks on CPU+FPGA heterogeneous platforms requires well-defined resource allocation and admission control mechanisms. Such mechanisms should guarantee the satisfaction of all timing constraints, along with a high resource utilization by maintaining the task's execution order.

In a scheduling problem, a real-time application is represented by a set of tasks with a *Directed Acyclic Graph* (DAG), where each node represents a task and each edge represents a data dependency. Each task has an execution time required to be completed within a stipulated deadline. The scheduling objective is to map tasks onto different *Processing Elements* (PEs) to satisfy the deadline constraint and the data dependencies while striving to achieve a minimum overall completion time, called the *makespan*. Such a problem has been proven to NP-hard [2].

A plethora of existing work discussed the scheduling problem for general multi-core computing environments, which involve various software computing modules such as a CPU and a GPU [3], [4]. Such scheduling algorithms consider the different computing speeds of heterogeneous PEs and the inter-core parallelism. In [5], 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 [6]. 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 [7] the authors present a scheduling algorithm for a fixed number of heterogeneous processesing units (CPUs, GPUs) to obtain both a high performance and lower makespan time, while maintaining the system's reliability against any faults.

With the increasing complexity level of high-performance computing and real-time embedded systems, current heterogeneous computing systems are employing FPGAs along with CPUs and GPUs to overcome existing limitations [1], [8]. FPGA-based multi-core systems are composed of multiple software executing PEs (i.e., CPUs and GPUs) and fixed hardware resources (FPGA). Software PEs are suitable for running serial programs. On the other hand, FPGAs are widely used as hardware accelerators by exploiting parallel execution and domain specific hardware implementations. Each task on an FPGA consumes a specific amount of FPGA resources. Therefore, FPGA-based multi-core scheduling should also take hardware resource constraints into account in addition to different computing speeds and inter-core parallelism.

In recent years, the problem of real-time task execution on FPGA-based heterogeneous systems has gathered considerable attention from the research community. The generic problem of real-time scheduling tasks has branched out in different directions primarily based on: i) using optimizing frameworks [9], ii) using heuristic algorithms [10], and iii) using priority-driven algorithms [11]. In [9], the authors proposed a static partitioning-based scheduling strategy for CPU+FPGA systems to minimize energy consumption. In [12], 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-real-time applications and did not consider hardware constraints. Recently, Zhu et al. [13] proposed a real-time task scheduling framework on CPU+FPGA systems, but their work only considered independent tasks. Dependent real-time task scheduling in an FPGA-based multi-core setting to minimize the makespan under hardware resource constraints has been investigated in [2]. However, this technique is only evaluated via software simulations using hypothetical FPGA parameters without considering any practical constraints. Until now, comprehensive studies that combine the theoretical aspects of energy-efficient real-time scheduling of approximated tasks along with runtime architectural characteristics have not been conducted.

To fill the gap, in this paper we propose a real-time scheduling algorithm with the objective of minimizing the makespan under hardware resource constraints for heterogeneous CPU+FPGA architectures. Scheduler decisions rely on task execution times and resource consumption metrics to map the tasks across CPU and FPGA regions. Specifically, we answer the following question: *Given a real-time task graph and heterogeneous PEs, how do we ensure that all tasks of the task graph will be executed on the appropriate PEs while satisfying the deadline and resource constraints?*

The main technical contributions of this paper are:

- We propose an *Integer Linear Programming* (ILP) based technique for task allocation and scheduling, designed for heterogeneous CPU+FPGA systems. In particular defining the domain and problem specific set of constraints for the ILP solver.
- 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.
- We provide a case-study to validate the applicability of our proposed scheduling technique in delivering practical results and demonstrate that performance gains of 40% and area usage reductions of 67% are possible compared to a full software and hardware execution, respectively.

Fig. 1 shows a high-level overview on our proposed approach for task mapping and scheduling in a heterogeneous CPU+FPGA system. The green boxes represent the users specification and inputs and initial artifcats for the system (see top of Fig. 1). The *real-time* (RT) constraints define the deadline for the schedule, how much memory and area are available on the CPU+FPGA system and potential other constraints of the real-time 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 *Software* (SW) implementation for the CPU and as *Hardware* (HW) implementation for the FPGA (i.e. as synthesizable RTL models written in a hardware-description language). Based on the task implementations, relevant execution metrics are obtained by executing the tasks in isolation (right, middle of Fig. 1). Important metrics are the execution time and area usage on the respective FPGA. These metrics are passed together with the RT constraints and task

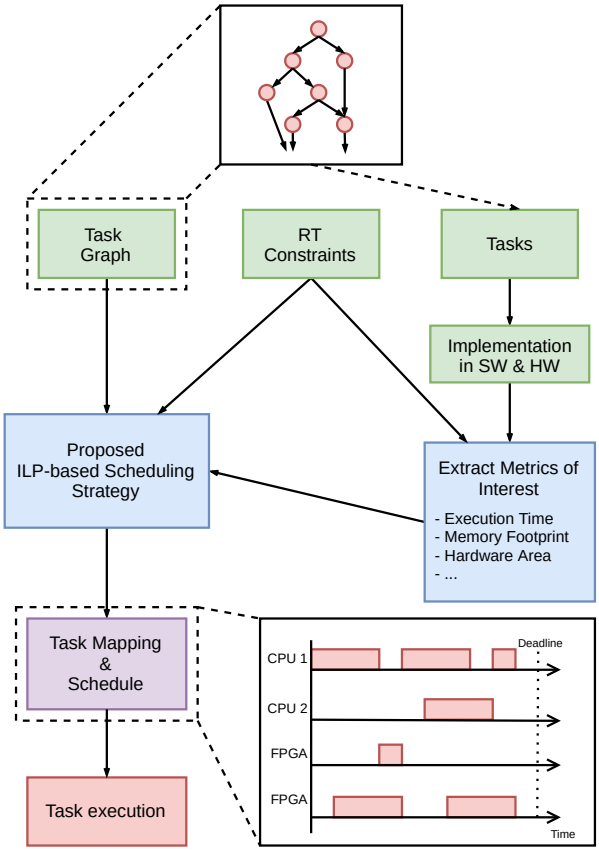

Fig. 1: Overview on our proposed approach.

graph to the ILP-based scheduling algorithm (left, middle of Fig. 1) which derives a task mapping and scheduling. The scheduling is then implemented on the heterogeneous CPU+FPGA system and executed.

Additional metrics can be integrated in order to consider practical implementation constraints such as the communication overhead in moving results between the CPU and FPGA 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.

Following this introduction, which already discussed related work, the remainder of the paper is organized as follows: We present our proposed scheduling strategy in Section II, which covers our ILP-based formulation. Then, we discuss practical constraints and trade-offs in implementing resulting scheduling on a heterogeneous CPU+FPGA system (Section III). Next, we present our RISC-V case-study with an example application task graph on which we apply our methodology and show the obtained results (Section IV). In Section V we further discuss our methodology and provide ideas for future work. Finally, we conclude the paper in Section VI.

II. Proposed Scheduling Strategy

In this section we provide the necessary defnitions (Section II-A and Section II-B) and present the proposed constraint-based formalism to obtain schedulings in this context (Section II-C).

A. Application and Architecture Model

We model a real-time application ($\mathcal{A}$) as a precedence constrained task graph $G = (T, E)$, where T is a set of tasks ($T = \{T_i \mid 1 \leq i \leq |T|\}$) and E is a set of directed edges ($E = \{\langle T_i, T_j\rangle \mid 1 \leq i, j \leq |T|; i \neq j\}$) representing the precedence relations between distinct pairs of tasks. An edge $\langle T_i, T_j\rangle$ refers to the fact that task T_j can begin execution only after the completion of T_i. A source task is a task with no predecessors and similarly, we define a sink task to be the one without any successors. However, being a real-time application, the entire application ($\mathcal{A}$) must meet its deadline, denoted as D_{Dag}, by executing all the associated task nodes within the interval.

B. Problem Description and Challenges

Given a real-time dependent task graph to be executed on a heterogeneous CPU+FPGA system, devise a scheduling strategy to minimize the overall execution time of the task graph. To achieve that, the scheduler must be able to handle all the following requests:

- What task to schedule at which time (*temporal reconfiguration*)?
- Where to place the respective task, in CPU or FPGA (*spatial reconfiguration*)?
- When to start the execution of a task according with its precedence constraints (*temporal scheduling*)?

This setup can be best compared to a multi-processor problem statement, as the platform provides multiple units for the execution of a task. However, due to the particular constraints and the challenges associated with heterogeneous architecture, state-of-the-art multi-processor scheduling strategies cannot be applied for such a platform. The following constraints differ from a pure multi-processor scheduling problem.

- Hardware task execution is non-preemptive.
- Software task execution is preemptive.
- The reconfiguration process for switching between hardware tasks is many times longer than a context switching overhead in software.
- The communication not only has to take place between processors but also between the software and hardware domain to utilize the hardware accelerators.
- Execution times of a task are heavily dependent on the selected execution unit. In general, the execution in hardware is faster as compared to software, however some tasks do not benefit from hardware acceleration (see Table I).

Deciding to place a task computation in software or hardware considering the given constraints is an optimization problem. In the following, we present how to obtain an effective solution to this problem.

C. ILP-based Scheduling

In this part, we present an *Integer Linear Programming* (ILP) solution to the optimal mapping of a DAG application in the heterogeneous CPU+FPGA platform. For this purpose, we define binary decision variables:

- $Z_{i,j,\eta}$ is 1 if task T_i starts execution in η^{th} *Reconfigurable Logic* (RL) at timestep j, 0 otherwise. Here the variables varies in the following ranges, $i = 1, 2, ..., |T|; t = 0, ..., D_{Dag}; \eta = 1...m_{RL}$
- $Y_{i,j,\eta}$ is 1 if task T_i starts execution in η^{th} processor (CPU) at timestep j, 0 otherwise. Here, $\eta \in 1...m_{EP}$ with m_{EP} being the number of *Embedded Processor* (EP).
- $R_{i,j}$ is 1 if reconfiguration for task T_i starts in RL at timestep j, 0 otherwise.

Furthermore, we provide the following symbols that denote specific execution relevant metrics:

- $e_{i,EP}$ denotes the SW execution time of task T_i.
- $e_{i,RL}$ denotes the HW execution time of task T_i.
- LC_i denotes the logic count for task T_i and TLC denotes the total available logic count.

We now present the required constraints on the decision variables to model this task mapping problem before presenting its overall objective function.

1) ***Unique Execution Start Time Constraint:*** Each task must start executing on a particular PE (either on EP or RL) at an unique time step. That is:

$$\forall i : 1 \leq i \leq |T| \mid \sum_{j=0}^{D_{Dag}} \sum_{\eta=1}^{m_{RPL}} Z_{i,j,\eta} = 1 \quad (1)$$

$$\forall i : 1 \leq i \leq |T| \mid \sum_{j=0}^{D_{Dag}} \sum_{\eta=1}^{m_{EP}} Y_{i,j,\eta} = 1 \quad (2)$$

2) ***Uniqueness Constraint:*** Each task can be executed only once using either its software version or hardware version. That is:

$$\forall i : 1 \leq i \leq |T| \mid \sum_{j=0}^{D_{Dag}} \sum_{\eta=1}^{m_{PE}} (Z_{i,j,\eta} + Y_{i,j,\eta}) = 1 \quad (3)$$

The above constraint enforces the following for each task:
- exactly one version (either software or hardware) will be selected for execution.
- start its execution on the processor at an unique time step.
- can be mapped only to one PE.

3) **Resource constraint:** In order to define this constraint, the following situation needs to be described first.
Lemma 1: If a task T_i has still not finished execution at the j^{th} time step, it must have started at most within ($j - e_{i,EP} + 1$) previous time steps. Hence, for this duration our previous derived variable should exhibit 1. i.e.,

$$\sum_{t=\psi}^{j} Y_{i,t} = 1$$

where, $\psi = max(0, j - e_{i,EP} + 1)$. Hence, for all tasks and for all EPs the SW resource constraint can be defined as:

$$\forall j : 0 \leq j \leq D_{Dag} \;\&\; \forall \eta : 1 \leq \eta \leq m_{EP} \mid \sum_{i=1}^{|T|} \sum_{t=\psi}^{j} Y_{i,j,\eta} \leq 1 \quad (4)$$

Equation 4 ensures that at any time step j, a EP can be busy due to the ongoing execution of at most one task.
Similarly for a RL (given as an FPGA), using Lemma 1, the constraint can be enforced as follows:

$$\forall j : 0 \leq j \leq D_{Dag} \;\&\; \forall \eta : 1 \leq \eta \leq m_{RL} \mid \sum_{i=1}^{|T|} \sum_{t=\beta}^{j} Z_{i,j,\eta} \leq 1 \quad (5)$$

where $\beta = max(0,\ j - e_{i,RL} + 1)$.

4) ***Version Conflict Elimination Constraint:*** Corresponding to each task, a task cannot be selected for software execution and hardware execution simultaneously. Hence, at a time-step j, T_i can either be executed on an EP or will start its reconfiguration for its execution on RL. This constraint can be enforced as follows:

$$\forall i : 1 \leq i \leq |T| \mid \sum_{j} \sum_{\eta=1}^{m_{EP}} Y_{i,j,\eta} + \sum_{j} R_{i,j} \leq 1 \quad (6)$$

5) ***FPGA Logic area Constraint:*** The tasks placed at FPGA should satisfy the logic area constraint i.e. the logic requirement of the task should be less than the available logic area budget. This constraint can be represented as:

$$\forall i : 1 \leq i \leq |T| \mid \sum_{j=0}^{D_{Dag}} \sum_{\eta=1}^{m_{RPL}} LC_i \times Z_{i,j,\eta} \leq TLC \quad (7)$$

6) ***Execution Dependency Constraint:*** Corresponding to each directed edge $(T_i, T_i' \in E)$ in the DAG, the execution of task T_i' must commence only after the completion of its predecessor, T_i. This dependency constraint between task T_i and T_i' is symbolically represented as follows:

$$\begin{aligned}\forall (T_i, T_i') \in E \mid \sum_{\eta=1}^{m_{EP}} \sum_{j} j \times Y_{i',j,\eta} + \sum_{\eta=1}^{m_{RL}} \sum_{j} j \times Z_{i',j,\eta} \\ \geq \sum_{\eta=1}^{m_{EP}} \sum_{j} j \times Y_{i,j,\eta} + \sum_{\eta=1}^{m_{EP}} \sum_{j} j \times e_{i,EP} + \\ \sum_{\eta=1}^{m_{RL}} \sum_{j} j \times Z_{i,j,\eta} + \sum_{\eta=1}^{m_{RL}} \sum_{j} j \times e_{i,RL} \quad (8)\end{aligned}$$

7) ***Deadline Constraint:***
In order to ensure that the application $\mathcal{A}$ meets its end-to-end absolute deadline D_{Dag}, the sink node $T_{|T|}$ must complete execution by D_{Dag}. That is:

$$\begin{aligned}\sum_{\eta=1}^{m_{EP}} \sum_{j} j \times Y_{|T|,j,\eta} + \sum_{\eta=1}^{m_{EP}} \sum_{j} j \times e_{|T|,EP} + \\ \sum_{\eta=1}^{m_{RL}} \sum_{j} j \times Z_{|T|,j,\eta} + \sum_{\eta=1}^{m_{RL}} \sum_{j} j \times e_{|T|,RL} \leq D_{dag} \quad (9)\end{aligned}$$

8) ***Objective:*** The objective of the formulation is to choose a feasible solution which minimizes finish time of the sink task. This is formulated as:

$$\begin{aligned}Minimize \ (\sum_{\eta=1}^{m_{EP}} \sum_{j} j \times Y_{|T|,j,\eta} + \sum_{\eta=1}^{m_{EP}} \sum_{j} j \times e_{|T|,EP} + \\ \sum_{\eta=1}^{m_{RL}} \sum_{j} j \times Z_{|T|,j,\eta} + \sum_{\eta=1}^{m_{RL}} \sum_{j} j \times e_{|T|,RL}) \quad (10)\end{aligned}$$

III. Application Case-Study Preliminaries

To evaluate the scheduling strategy an application case-study featuring a realistic heterogeneous real-time system is specified and designed. Especially on the hardware side there exist several choices in building the overall system, which in turn has impact on the task implementation and execution. An important part is the FPGA which has to be chosen. It has to provide sufficient area to fit a processing system like a *System-on-a-Chip* (SoC) and additional hardware tasks. Commercially available FPGAs offer a variety of additional features to the conventional programmable logic blocks and block RAM. For example some FPGAs like the Xilinx Zynq 7000 [14] feature an integrated ARM Cortex-M9 dual core-processor with a multi-channel *Direct Memory Access* (DMA) controller and various SoC peripherals while the programmable FPGA logic contains additional blocks for *Digital Signal Processing* (DSP), high-speed transceivers and more. Other commercial FPGA manufacturers like Intel, Lattice Semiconductor and Microsemi offer similarly broad solutions with different features and integrated processor or an extensive library of *Intellectual Property* (IP) cores. Depending on the choice various aspects of the task mapping and scheduling can change. Our proposed ILP-based scheduling offers to consider technological constraints and considerations as long as they can be formulated in a ILP constraint (e.g. memory access times through various available technologies that have an impact on memory and area usage at the same time).

As we can not cover all possible configurations of various heterogeneous real-time systems, we summarize the relevant practical considerations for various real-time 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 IV.

A. General Practical Considerations

Heterogeneous real-time systems encounter various practical considerations that can not easily be formulated formally in terms of constraints. The following list provides a set of relevant practical constraints which depend on the actual system and focus on technical aspects with regard to communication between software and hardware tasks:

1) What are the capabilities and requirements of the embedded system?
   a) Is there a (special) shared memory?
   b) Is DMA available?
   c) If 1a and 1b are not available, where and how should the task related data be stored?
2) How is data transported or shared between the SW and HW 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 amount and kind of 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 on the system. E.g. there will be a transportation and synchronisation 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 relevant to the scheduling outcome. The technical implementation has also impact on the software memory footprint (e.g. additional code and memory areas to manage DMA or other interfaces to share data and memory).

B. Technical System Considerations

The goal for our application case-study is to evaluate the viability of our approach on an actual heterogeneous real-time system. Our

target system combines an FPGA together with a soft-core CPU based SoC. This SoC provides a basic set of peripherals needed in embedded systems while leaving enough memory space and FPGA fabric area for custom hardware based tasks. Tasks that are implemented as software are stored in the SoCs memory, while tasks that are implemented as hardware are connected to the SoC 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 memory and CPU accessible memory. Moreover, we consider a bare-metal software setting without employing operating systems that might provide preemptive task scheduling capabilities.

IV. Evaluation: A RISC-V Case-Study

This section presents results on the evaluation of our proposed ILP-based scheduling algorithm and shows our proposed task execution and implementation strategy on a concrete heterogeneous CPU+FPGA system using an application case-study. We start with a description on the specific choices with regard to the technical considerations, which consititute the setup of our evaluation (Section IV-A). Then, the example application is introduced and a corresponding implementation sketch is provided (Section IV-B). Next, we present relevant metrics and the obtained scheduling based for the example application based on our proposed methodology (Section IV-C). Finally, we present and discuss the overall results in obtaining the executed scheduling and elaborate how the system choice impacted the realization of the schedule (Section IV-D).

A. Setup

For this case-study we choose the Lattice Semiconductor HX8K FPGA [15] that is capable of containing a SoC whilst offering additional FPGA fabric area for hardware tasks. Compared to other commercially available FPGAs the HX8K does not offer a built-in SoC or slices for DSP tasks like multiply-accumulate. Within the technology of the HX8K, area is mainly determined through *Logic Cells* (LC). These LC contain a four-input look-up table, a D-flip-flop with optional enable and reset controls and carry logic to interconnect with other LCs. Additionally the HX8K FPGA is compatible with the open source toolchain IceStorm [16], which includes the open source synthesis tool Yosys [17].

As a SoC we choose the Murax SoC. The Murax SoC uses a SpinalHDL [18] based RISC-V [19], [20] implementation called VexRiscv [21]. It is known for the high degree of configurability while minimizing the overhead of the generated code, thus resulting in very small FPGA-compatible RISC-V CPUs while suiting the requirements for real-time embedded system tasks. 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 [22] for peripherals. All tasks are implemented in C for the RISC-V processor and in SpinalHDL for the hardware tasks. SpinalHDL is an emerging language for hardware description and generation that can be used to describe hardware 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 hardware descriptions. The SpinalHDL-based descriptions can be used to generate either Verilog or VHDL code. As the hardware tasks are described with SpinalHDL an easy integration into the Murax SoC is ensured. The complete development toolchain is based on open source tools and allows for static and simulation based analysis. The main simulation backend in SpinalHDL is Verilator [23]. Verilator is used to obtain a cycle- and synthesis-accurate RTL simulation to extract the metrics like the execution

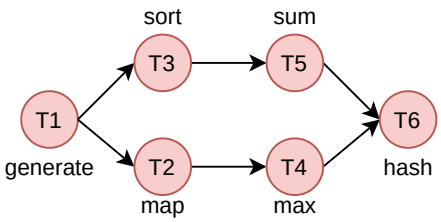

Fig. 2: Task graph for the case-study example application.

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 software in which software and hardware task execution is orchestrated and interleaved. The execution of this schedule is measured on the FPGA and through the cycle- and synthesis-accurate RTL simulation to compare the calculated result with the experimental result. With these results we discuss some of the boundaries of the scheduling strategy with regards to the practical considerations in Section III-A.

Shared memory architectures and DMA for easy data transfer between the CPU and a HW task are not part of Murax SoC. This is due to the goal of Murax SoC being able to fit in small FPGAs such as the HX8K FPGA (and even smaller variants of the same FPGA-family [15] of Lattice Semiconductor). Thus, we have a low-level bare-metal embedded system for our application case-study representing an FPGA-based heterogeneous real-time 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.

For this application case-study each hardware 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 tasks execution. The tasks 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 valid is being asserted, the tasks starts its processing with the current memory content. Once the task is finished, the ready flag will be asserted by the task and the tasks 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 tasks execution the CPU can read all resulting data from the tasks memory. Additional configuration inputs are mapped to memory mapped registers.

B. Application and Implementation

Fig. 2 shows a task graph with six different tasks that represent the example application of this case-study. The tasks represent data flow operations known from functional programming. The task graph combines vector and scalar operations. The first task generates a vector V1 with pseudorandom values based on an initial seed. The next two tasks process V1 into the vectors V2 and V3 by mapping and sorting the values in V2. Then, V2 and V3 are transformed into scalar values S1 and S2 by computing the maximum value and applying a value reduction, respectively. Finally, a unique hash value is obtained by combining S1 and S2 into a single integer.

A directed edge represents a dependency on the output/input of another task. Therefore a task can only be executed if and only

TABLE I: Task metrics of the example application with six tasks.

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		
T_1 (generate)	40.33	80	22.92	-	0.92	18.50	100	805
T_2 (map)	20.83	76	43.58	19.42	0.92	18.50	144	721
T_3 (sort)	79.00	112	49.75	19.42	10.50	18.50	144	840
T_4 (max)	33.67	96	23.67	19.42	0.92	0.17	108	686
T_5 (sum)	24.92	80	23.67	19.42	0.92	0.17	108	653
T_6 (hash)	88.42	144	7.33	0.83	2.25	0.17	72	628

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

```
1  // store all element of the array into the memory
       of the task
2  for (uint8_t i = 0; i < vecSize; i++) {
3    TASK_MAX->MEM_ADDR = i;
4    TASK_MAX->MEM_WDATA = inputData[i];
5    TASK_MAX->MEM_WRENA = 1;
6    TASK_MAX->MEM_WRENA = 0;
7  }
8  // start the task
9  TASK_MAX->VALID = 1;
10 // check ready flag of task until its done
       processing
11 while (!TASK_MAX->READY);
12 // load max value
13 maxVal = TASK_MAX->MAX_VALUE;
```

if the required data is available. For example: Task T2 (map) can only be executed if the data from task T1 (generate) is available. This results in constraints for the order in which the tasks can be executed. At the same time, these tasks can be implemented into a hardware description by hand to evaluate the feasibility of the implementation step of the top level flow from Fig. 1. For each task an implementation in C and SpinalHDL is implemented and measured for their metrics such as execution time, area consumption after synthesis, software memory footprint and transportation time of the data between CPU and FPGA fabric.

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. Passing the data from and to the tasks T2 and T3 have to be handled as part of the scheduling. A fork in that sense means also 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 software is succeeded by a task in hardware and data is moved from the software task to the hardware task.
2) A task in hardware is succeeded by a task in hardware and data is moved from one hardware task to another hardware task.
3) A task in hardware is succeeded by a task in software and data is moved from the hardware task to the software 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.

In general our architecture requires the software 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.

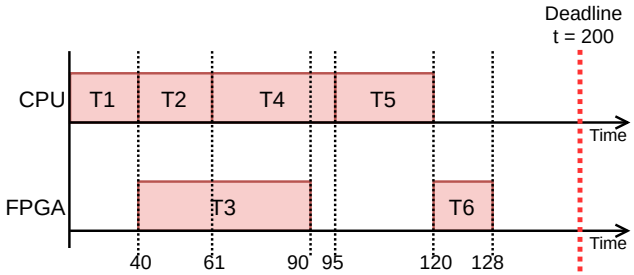

Fig. 3: Scheduling outcome from our proposed algorithm for the example application.

Listing 1 shows such an exemplary transaction between the CPU and the hardware 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 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 on using hardware tasks.

If preemption of tasks is included in the considered properties of our task scheduling strategy, the active checking for the ready flag (see 1 line 11) could be handled through interrupts.

C. Metrics and Scheduling

Table I shows the measured task parameters of our example application. The task parameters from the software and hardware tasks are fed into our ILP formulation from Section II-C. Together with the top-level constraints (e.g. deadline at $200\mu s$, area budget of $1500LC$) the CPLEX [24] solver, which we employ for ILP solving, generates an optimal task mapping and scheduling according to our ILP formulation. In this case we obtain the scheduling as shown in Fig. 3.

Fig. 3 shows the calculated schedule for the tasks with the parameters from Table I. 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 compact way. The tasks T1, T2, T4 and T5 are mapped on the CPU and the tasks T3 and T6 are mapped to be executed as hardware tasks on the FPGA. The total runtime is calculated as $128\mu s$, which is far below the deadline of $200\mu s$. The additional hardware area used is $1468LC$ which also is below the budget of $1500LC$.

With this schedule we can now use the mapping for the software and hardware tasks and implement the top level schedule such that it executes the proposed solution. After the bootcode of the SoC has completed, the proposed schedule is executed. The SW tasks are implemented as C functions, which are called with their parameters

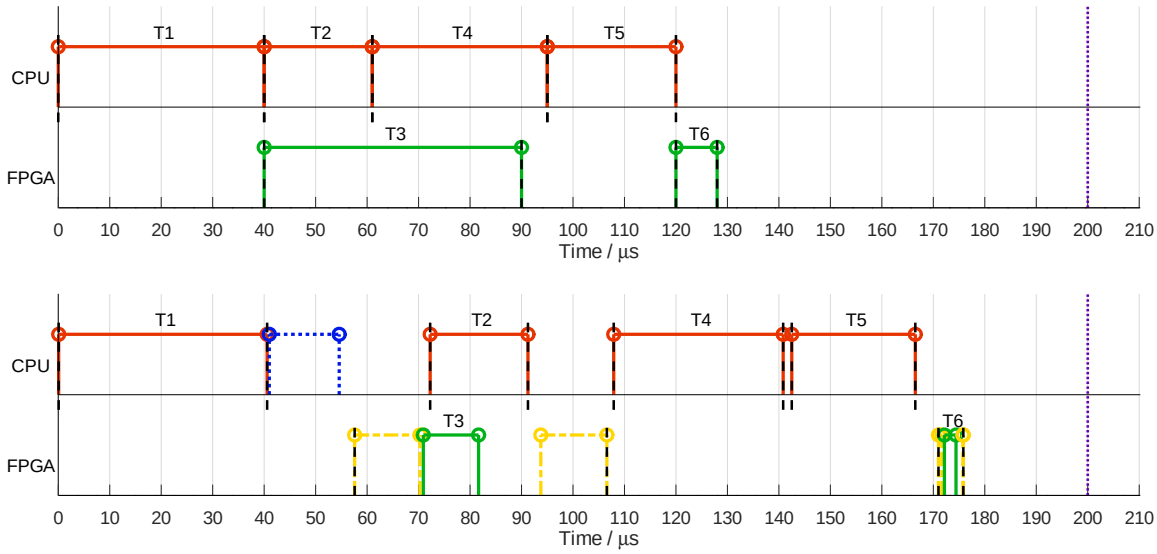

Fig. 4: Compared schedule results for application study. **Top:** calculated schedule (also refer to Fig. 3), **Bottom:** executed schedule implemented in application study.

TABLE II: Baseline of the case-study setup, Lattice Semiconductor HX8K FPGA with VexRiscv, Murax SoC.

Description	Maximum available	Used	Available
Memory usage / Bytes	2048	776	1272
Area usage / LC	7680	2514	5166

and their return value is stored into a variable to be accessed by the next task. For the hardware tasks, the software implementation is used as a specification. Control flow elements from the software task are implemented as finite state machines while the data flow elements represent the data path of the circuit.

D. Results

Initially we look into a comparison between the calculated ILP scheduling with the measured scheduling execution on the embedded system. Fig. 4 shows the respective schedules for comparison. The top plot shows the calculated schedule from our proposed scheduling algorithm. The bottom plot shows the real execution of the schedule on the embedded 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). Each task is delimited with two vertical dashed line (black) and annotated with a task identifier corresponding with I. The top half of the plots show the execution traces of the tasks on the CPU part of the system (red and blue events). For the CPU tasks, the strike through events (red) show the task execution on the CPU, while the dashed event is the execution of housekeeping data. This is required, as for example T3 and T2 both require the same data from T1 and thus it needs to be copied once. The bottom parts of the plots show the execution traces for the hardware tasks on the FPGA fabric (yellow and green events). For the FPGA 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. This is required, as our application case-study leverages an embedded system without shared memory or DMA for devices on the memory bus.

Table II shows the baseline values for the memory usage in Bytes and the area usage in LCs. These values declare the maximum budget of the memory and area that are available. In our application case-study we limited the budget to values lower than the available space.

TABLE III: Proposed schedule in context to executing all tasks in software or hardware.

Property \ Schedule	All Software	Proposed	All Hardware
Memory Footprint (complete) / Bytes	1352	1460	1440
Memory Footprint (no bootcode) / Bytes	576	684	664
Area Usage (complete) / LC	2514	3956	7015
Area Usage (w/o SoC) / LC	0	1442	4501
Total Execution Time / μs	294.24	175.50	189.49

Table III 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 scheduling can be expected. For the schedule *All Hardware* we kept the same sequential order for task execution as for *All Software*. The memory footprint and the area usage are declared twice. In the rows with *(complete)* annotation, the absolute size in terms of Bytes and LCs is shown. The other rows show the values for just the software and hardware solution of the tasks, respectively. These values are calculated as the difference to the baseline values of the embedded system from Table II. As expected the *All Software* schedule requires no additional hardware, while the *All Hardware* requires $4501LC$ to implement all tasks in HW. The *All Hardware* schedule needs 664 Bytes of code, in order to interact with the HW tasks and move the task data around. Our proposed schedule requires 20 Bytes more than the *All Hardware* schedule while requiring much less area of the FPGA fabric.

V. Discussion and Future Work

The results shown in Fig. 4 show differences in how the schedule is executed on the embedded system. We can observe that additional time is spent in setting up the execution of tasks. For example due to the fork in the task graph (in Fig. 2 the task T1 forks to T2 and T3) our architecture needs an additional copy of the data of T1. This can be seen as the time interval marked in blue on the bottom

schedule plot. Additionally, data needs to be transferred from and to the tasks memory if a task is executed in hardware. These transfers are plotted as yellow time intervals in the bottom schedule plot. 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 can refine our constraints to represent our system architecture. For example additional information on the transport duration (see Table I column 5 and 7) can be formulated as part of the hardware tasks that are necessary to contain the execution time. Constraints like these can either be formulated in advance with our given set of ILP constraints or refined in an additional iteration of the methodology in Fig. 1. 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 III-A, Section III-B and Section IV-A) 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 real-time 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.

For future work, we aim to consider further evaluations that involve different heterogeneous real-time systems and different application examples. These systems should contain a range of different features to expand on the general and specific considerations. Through more evaluations we can refine our methodology, for example with a feedback loop, to include 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 real-time system. VPs allow early HW-SW co-design and verification, thus a possible feedback loop in the methodology can be achieved more efficiently.

VI. Conclusion

In this paper we proposed a static scheduling strategy and methodology for mapping and scheduling application tasks for a heterogeneous real-time system. The strategy encompasses an ILP-based optimization of constraints modeling the applications properties. Through these constraints we describe general scheduling properties (such as deadlines or preemption behavior) as well as relevant system architecture and application specific properties (such as hardware area budget or software memory limits). We proposed general practical and technological considerations that help engineers in making decisions and understanding the advantages, disadvantages as well as the limitations of the underlying systems architecture. 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, specific hardware task interfaces, etc.). Our evaluation demonstrates the applicability of our proposed scheduling algorithm in providing practical results for a heterogeneous CPU+FPGA system. 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.

Acknowledgment

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project VerSys under contract no. 01IW19001, and by the Yerun Research Mobility Award (YRMA) scheme, UK Engineering and Physical Sciences Research Council (EPSRC) through grant EP/V000462/1.

References

[1] C. Bobda, J. M. Mbongue, P. Chow, M. Ewais, N. Tarafdar, J. C. Vega, K. Eguro, D. Koch, S. Handagala, M. Leeser *et al.*, "The future of fpga acceleration in datacenters and the cloud," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 3, pp. 1–42, 2022.

[2] J. Xu, K. Li, and Y. Chen, "Real-time task scheduling for fpga-based multicore systems with communication delay," *Microprocessors and Microsystems*, vol. 90, p. 104468, 2022.

[3] A. Dhar, E. Richter, M. Yu, W. Zuo, X. Wang, N. S. Kim, and D. Chen, "Dml: Dynamic partial reconfiguration with scalable task scheduling for multi-applications on fpgas," *IEEE Transactions on Computers*, 2021.

[4] C. Zhang, H. Yu, Y. Zhou, and H. Jiang, "High-performance and energy-efficient fpga-gpu-cpu heterogeneous system implementation," in *Advances in Parallel & Distributed Processing, and Applications*. Springer, 2021, pp. 477–492.

[5] J. Fang, J. Zhang, S. Lu, H. Zhao, D. Zhang, and Y. Cui, "Task scheduling strategy for heterogeneous multicore systems," *IEEE Consumer Electronics Magazine*, vol. 11, no. 1, pp. 73–79, 2021.

[6] Y. Elmougy, W. Jia, X. Ding, and J. Shan, "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, 2021, pp. 1–10.

[7] Z. Deng, D. Cao, H. Shen, Z. Yan, and H. Huang, "Reliability-aware task scheduling for energy efficiency on heterogeneous multiprocessor systems," *The Journal of Supercomputing*, vol. 77, no. 10, pp. 11 643–11 681, 2021.

[8] H. Chen, S. Madaminov, M. Ferdman, and P. Milder, "Fpga-accelerated samplesort for large data sets," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 222–232.

[9] A. Rodríguez, A. Navarro, R. Asenjo, F. Corbera, R. Gran, D. Suárez, and J. Nunez-Yanez, "Exploring heterogeneous scheduling for edge computing with cpu and fpga mpsocs," *Journal of Systems Architecture*, vol. 98, pp. 27–40, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762119300918

[10] T. Zhang, G. Liu, Q. Yue, X. Zhao, and M. Hu, "Using firework algorithm for multi-objective hardware/software partitioning," *IEEE Access*, vol. 7, pp. 3712–3721, 2018.

[11] J. Josephson and R. Ramesh, "A novel algorithm for real time task scheduling in multiprocessor environment," *Cluster Computing*, vol. 22, no. 6, pp. 13 761–13 771, 2019.

[12] S. Yesil and O. Ozturk, "Scheduling for heterogeneous systems in accelerator-rich environments," *The Journal of Supercomputing*, vol. 78, no. 1, pp. 200–221, 2022.

[13] Z. Zhu, "A hardware and software task-scheduling framework based on cpu+ fpga heterogeneous architecture in edge computing," *IEEE Access*, vol. 7, pp. 148 975–148 988, 2019.

[14] Xilinx, "Zynq-7000 SoC," https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html, accessed on 2022-03-24.

[15] L. Semiconductor, "ice40 lp/hx family data sheet," https://www.latticesemi.com/view_document?document_id=49312, accessed on 2022-03-24.

[16] C.Wolf and M.Lasser, "Project icestorm," http://bygone.clairexen.net/icestorm/, 2021, accessed on 2022-03-24.

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

[18] C.Papon, "Spinalhdl," https://github.com/SpinalHDL/SpinalHDL, 2021, accessed on 2022-03-24.

[19] A. Waterman and K. Asanović, Eds., *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, 2019.

[20] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, 2019.

[21] C.Papon, "Vexriscv," https://github.com/SpinalHDL/VexRiscv, 2021, accessed on 2022-03-24.

[22] A. Limited, "Amba 3 apb protocol specification v1.0," https://developer.arm.com/documentation/ihi0024/b/, 2003, 2004, accessed on 2022-03-24.

[23] W. Synder, "Verilator," https://veripool.org/verilator/, 2003-2022, accessed on 2022-03-24.

[24] S. Nickel, "Ibm ilog cplex optimization studio," in *Angewandte Optimierung mit IBM ILOG CPLEX Optimization Studio*. Springer, 2021, pp. 9–23.