Configuration splitting to extend the processing capabilities of embedded reconfigurable systems

Christopher Cichiwskyj
University of Duisburg-Essen
Duisburg, Germany
christopher.cichiwskyj@uni-due.de

ABSTRACT
The Elastic node is an embedded hardware platform using an 8-bit MCU and a low-power FPGA, allowing to create IoT applications with more local processing power. Due to the limited amount of resources on low-power FPGAs, very large hardware designs cannot be instantiated, thus limiting the range of algorithms for this platform. To circumvent this issue large designs can be split, using the FPGAs reconfigurability to switch between the smaller parts. This paper outlines the open question regarding such split logic and provides possible solutions.

1 INTRODUCTION
The Internet of Things is a rapidly growing network of everyday objects that are extended with embedded computing devices. With their integrated sensors and actuators they can interact with the physical world and through the internet communicate with other devices and services. Due to their embedded nature these devices typically have to be battery powered and energy efficient to last months or even years. The trend in IoT applications however is that more complex algorithms are required. Machine learning, computer vision, or self-organisation are only some of the concepts, that embedded devices are often not capable of performing locally with their limited resources. A solution to this is offloading these complex tasks e.g. to the cloud. Depending on the type of calculation and application scenario this is not always viable, due to latency, security or privacy concerns.

2 THE ELASTIC NODE
To be able to increase the local performance without requiring larger, more energy intense processors we propose the Elastic Node. The Elastic node [1][2] is an IoT hardware platform, designed to dynamically adapt its hardware capabilities according to the required workload, thus allowing more tasks to be calculated locally. The idea is that it only “grows” as much as is required to be able to solve a given task, and returns back, or “shrinks”, to its low performance state once the complex task is finished.

It is implemented with a low-power 8-bit microcontroller (MCU) and an embedded field-programmable gate array (FPGA), a piece of reconfigurable hardware capable of instantiating arbitrary digital circuits. An application consists of multiple tasks. Simple tasks are executed efficiently on the MCU, complex tasks that require more processing power are offloaded to the FPGA as so called Hardware Functions (HWF). To do so the FPGA is powered up, reconfigured to instantiate the appropriate circuit, performs the necessary calculations, and once finished returns to a sleep state. Preliminary results show that the Elastic Node can perform these tasks more energy efficiently, processing them locally, than when offloading the same tasks through a wireless channel [2].

3 PROBLEM STATEMENT
Unlike the Harvard or Von-Neumann computing architectures an FPGA does not use a classical computer program. Instead an FPGA developer designs the required circuitry using Hardware Description Languages and synthesises them into configuration files. A configuration file describes to the FPGA how its hardware resources on the chip should be interconnected to form the desired circuitry. Larger FPGAs typically have the capabilities to instantiate all logic corresponding to a single HWF in one configuration file. Offering this functionality however, usually requires operating system (OS) abstraction [3][4][5]. Embedded FPGAs do not have enough resources to instantiate such large logic and embedded MCUs often cannot provide full OS abstractions. This limits what HWFs can be implemented on the Elastic Node. A solution to this is to split the larger HWF into smaller subtasks, each small enough to fit into an embedded FPGA configuration. The HWF is then calculated by reconfiguring the FPGA to each subtask and pass the intermediate data between them.

The focus of this research is to determine, if it is possible to execute such large HWFs on an embedded FPGA while remaining more or as efficient as offloading these tasks e.g. to the cloud. Certain key challenges need to be addressed.

Coordinating split HWFs: When certain HWFs are not self-contained logical units, but instead have predefined internal subtask data dependencies, their execution order is not purely defined by the application anymore. Instead certain FPGA configurations need to be loaded and executed in a certain order to ensure the correct overall HWF result.

Reducing reconfiguration overhead: Splitting HWFs into multiple configurations means that more reconfigurations have to be performed to achieve the same result. This means added overhead to the overall execution time of a single HWF. The question is whether this overhead can be reduced.

Intermediate data passing management: As FPGAs are not capable of keeping any data internally throughout reconfigurations, data that has to be passed between subtasks have to buffered and
While the actual execution time of HWFs take nanoseconds \[2\], the is obsolete. Doing so in a static fashion however is not possible, as MCU is scarce any intermediate data has to be freed as soon as it the data management of such intermediate data. As memory on the data across reconfigurations, thus requiring the MCU to perform characters when aligning both sequences.

is mapped to a string, where each character represents an FPGA processing units to distribute the tasks instead of reconfiguring however these focus on splitting the available FPGA area into multi-execution order dictated by the task graph is not violated.

by two. A grouping approach however needs to ensure that the same two HWF sequences reduced the number of reconfigurations be reduced. Grouping subtasks, that use identical configurations, or when the same HWF is called repeatedly, reconfigurations can be done at design time.

Calling multiple HWFs requires a scheduling. A naive approach is to queue them sequentially, as seen in fig. 1a. This schedule requires seven reconfigurations. Each reconfiguration, however, means significant overhead, both in time and energy consumption. While the actual execution time of HWFs take nanoseconds \[2\], the reconfiguration requires multiple milliseconds \[7\]. Assuming that FPGA subtask configurations are reused across different HWFs, or when the same HWF is called repeatedly, reconfigurations can be reduced. Grouping subtasks, that use identical configurations, allows reducing reconfigurations. As seen in fig. 1b grouping the same two HWF sequences reduced the number of reconfigurations by two. A grouping approach however needs to ensure that the execution order dictated by the task graph is not violated.

Similar approaches using task graphs have been proposed \[8\]\[9\], however these focus on splitting the available FPGA area into multiple processing units to distribute the tasks instead of reconfiguring the complete FPGA.

A promising approach is to map the grouping problem onto DNA sequence alignment algorithms such as \[10\]. Each HWF sequence is mapped to a string, where each character represents an FPGA configuration. The goal is to minimise the transitions between characters when aligning both sequences.

As mentioned in section 3 the FPGA cannot store intermediate data across reconfigurations, thus requiring the MCU to perform the data management of such intermediate data. As memory on the MCU is scarce any intermediate data has to be freed as soon as it is obsolete. Doing so in a static fashion however is not possible, as

the time when to free data does not only depend on the data dependency described in the task graph but also the runtime schedule of grouping HWFs.

5 EVALUATION

Due to the remaining open questions regarding possible implementation details the evaluation is not completely defined yet. The main criteria is whether splitting HWFs is viable with regards to latency and energy consumption. To this regard it will be compared to alternative approaches, such as offloading it wirelessly as well as performing all calculation solely on MCUs with different processing capabilities. The reconfiguration reduction heavily depends on the similarity of the HWF sequences that are aligned. Varying the HWFs, the similarity of sequences as well as their number will give an insight into what types of applications benefit from this approach. When splitting HWFs into smaller subtasks the reconfiguration overhead is a significant part of the overall performance. This will answer at what point it becomes more viable to use a larger FPGA, regarding both latency and energy consumption, instead of continuously reconfiguring on a smaller FPGA.

6 CONCLUSION

Extending embedded devices with reconfigurable hardware has the potential to lessen the dependence on offloading complex calculations. By answering the above mentioned questions the insights can be used to create a middleware abstraction to provides developers a simplified way to implement and use HWFs, as well as provide a better understanding as how to design application on reconfigurable systems and what application scenarios can benefit from this type of hardware.

REFERENCES