A Framework for Effective Exploitation of Partial Reconfiguration in Dataflow Computing

Riccardo Cattaneo*, Xinyu Niu†, Christian Pilato*, Tobias Becker†, Wayne Luk†, Marco D. Santambrogio*
* Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
† Department of Computing
Imperial College of London

***Abstract*—The exploitation of high-performance architectures based on reconfigurable hardware to build power efficient supercomputing clusters is becoming more and more common. Indeed, large speedups have already been demonstrated in several high-performance computing (HPC) applications. On the other hand, partial reconfiguration (PR) has the potential to further increase performance and power efficiency in many applications; however, there is currently very limited support for transforming a traditional design into a reconfigurable one. In this work, we introduce a design methodology for PR designs that combines application analysis, partitioning, mapping and scheduling, and supports fast exploration of various design options. These steps are integrated in an automated toolchain which allows a designer to implement reconfigurable designs with simple guidance through a graphical interface. We demonstrate our approach by applying the methodology to an image processing application, implementing the proposed design on a Maxeler MaxWorkstation.**

I. Introduction and Related Work

With the advancements of the miniaturization processes of field programmable gate arrays (FPGAs) it is nowadays possible to attain unprecedented levels of high components density and low processing power consumption [**?**], [**?**]. This, in turns, allows designers for more complex solutions in the field of hardware acceleration, ASIC prototyping and network interconnects.

One of the traditional issues with heterogeneous systems, especially in FPGAs, is the effort required by the designer to implement them. This is due to many factors: vastly different toolchains and development environments across different FPGA vendors, device programming languages (VHDL and Verilog paradigm is largely different than that of imperative or procedural languages like C or Java), resource usage planning, meaningful application partitioning in large designs, crisp knowledge of both the hardware and software sides of the overall system. An additional specific peculiarity of FPGAs is its *intrinsically parallel* computing capabilities: as long as there is enough available area to instantiate new hardware accelerators, all computations across all accelerators may run in parallel [**?**]. This is particular interesting in the light of some specific models of computation like dataflow computing (DC) [**?**]. These models of computation describe the target application as an appropriate graph where nodes represent portion of computations and links represents the flow of intermediate results from input to output. Using these paradigm, it is possible to automatically extract both coarse and fine grained parallelism from the topology of the graph, up from the task level down to the operator level. For these reasons, many toolchains and platforms have been developed – both by academia [**?**], [**?**], [**?**] and industry [**?**], [**?**] – to ease these burdens and make it easier to develop reconfigurable hardware based heterogeneous systems, in particular aimed at high performance computing [**?**]. As of today, most of these toolchains mainly focus on the design of solutions which allow reconfiguration between *different* computations, and take little to no advantage of the possibility to reconfigure the device during the *same* computation, a feature called Partial Reconfigurability (PR) [**?**]. In this case the application is represented as a sequence of operations that do not need (or cannot) overlap their execution, thus allowing for the *multiplexing* of the FPGA's limited resources to occur: each operation corresponds to a distinct configuration that can be loaded into the FPGA at run-time, possibly while other operations are still running and without interrupting them.

Given the relative novelty of this technique, it is still difficult for designers to decide whether to come up with a reconfigurable solution or not. In this work, we propose a framework able to automatically devise the conditions that allow an application executed onto a reconfigurable architecture to fully benefit from PR. The main contributions of this work are:

- ASAP, a GUI aimed at guiding the designer through the necessary steps for designing a reconfigurable architecture,
- Application Analysis, a component meant to analyze the application's source code and translate it into a hierarchical task graph and data flow graph
- High Level Analysis, a component to estimate performance metrics about tasks' implementations
- Reconfiguration-aware Mapping and Scheduling, a component able to optimally map and schedule tasks onto a reconfigurable architecture, with the goal of minimizing overall execution time

The paper is structured as follows: Section **??** presents the overall framework and the motivation of our work. Section **??** describes the methodology employed for analyzing the input application. Section **??** presents the high level analysis tool employed to provide estimates to the toolchain. In Section **??** we discuss the mapping and scheduling of the application's tasks, with a focus on partial reconfiguration. Section **??**

discusses some preliminary results validating the overall approach. Finally, Section **??** concludes the paper and presents the future steps to improve our work.

II. Proposed Framework

The overall framework presented in this work is composed of four interacting components, as in Figure **??**.

The designer interacts only with ASAP, a graphical user interface guiding the user through the various steps of the design process, asking her for the data required by the underlying components in a structure and user friendly manner. To ease the development process, the user never interacts directly with the underlying tools, even though they might be invoked as separate components.

The first step in ASAP is to specify the application under examination. It must be written in C and decorated with appropriate OpenMP `#pragma` instructions to delimit and describe the functions that will be the target of the application analysis, high level analysis and – possibly – acceleration on hardware. The hardware implementations of these functions might be generated using an high level synthesis (HLS) tool or might be manually specified. The pragmas allow to specify task level parallelism, dataflow relationships between tasks and amount of data required by the functions to operate. Additional information (such as memory access patterns and data bitwidth) can be also specified by means of custom `#pragmas`.

After specifying the source code of the application, the **Application Analysis** component is invoked on the source code to extract informations about its structure. It does so by analyzing the call graph of the application and the sequence of read/write operations, finally producing the task graph of the application itself and, for each of those tasks, the corresponding data flow graph.

Afterwards, **High-Level Analysis** component is invoked on each element of the task graph in order to produce the estimates of the resource consumption associated to each of the marked functions. To more extensively explore the design space, different trade offs between resource consumption and performance goals are extracted, so that multiple hardware implementations of the same tasks are subsequently available to the mapper.

Subsequently, the designer must specify an architectural template, which is the description of the elements that are present in the system such as generic processors, static hardware accelerators, reconfigurable regions, memory elements and interconnects. Each of these components can be further specified as appropriate. In particular, the number of reconfigurable regions is considered an upper bound to the number of regions effectively used in the resulting design: it is up to the mapping and scheduling component to decide whether to actually use all of the regions or just a subset of them. This aspect is fundamental to guarantee the feasibility of the resulting design: if we allowed more regions than the number that the device can accustom for, a violation of resource constraints would occur, making the solution unfeasible. On the other hand, specifying an upper bound allows us to suggest the usage of at most N regions, which might not be effectively used in the resulting design in order to forbid unfeasible solutions.

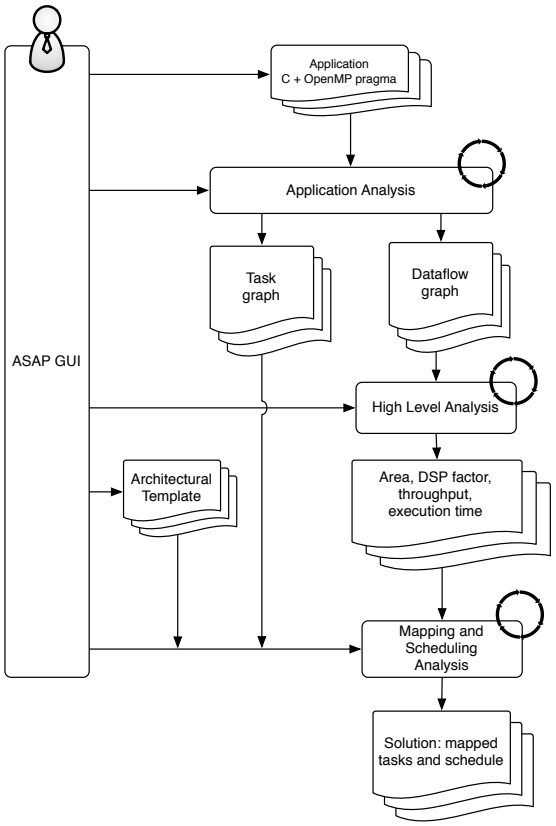


Fig. 1: Overview of the framework. Circles indicate semi-automatic toolchains, agent indicate manual steps.

Finally, the **Mapping and Scheduling** tool is invoked. This component is in charge of analyzing the input task graph, the available implementations (and related resource consumption estimates) for implementing tasks in hardware or software and the architectural template to find the best way to assign each task to the various components in the system. It does so while taking into account the presence of PR as an added degree of freedom to – possibly – overcome resource constraints (by time multiplexing resources) and meet or exceed performance goals.

III. Application Analysis

The first analysis operated on the source code (written in C and decorated with custom and OpenMP `#pragmas`) is done by the *Application Analysis* component. This is divided in three separate phases: *task extraction*, *code analysis* and *DFG generation*.

First, in the **task extraction**, the initial source code is analyzed to identify which are the tasks to be ported in hardware and their interconnections (i.e., data exchange). As output, it produces an XML file containing the description of the resulting task graph with respect to the data streams. More details about this phase can be found in Section **??**.

Then, in the **code analysis**, the initial source code of each marked task is analyzed with a C-to-C compiler step that extracts relevant information specified by means of custom pragmas. As output, it produces an XML file containing a

representing this information (e.g., the list of variables, along with their memory access patterns). More details about this phase can be found in Section **??**.

Finally, the **DFG generation** starts from the initial source code of the kernel and the information identified in the previous step. Another compiler step is adopted to automatically translate the C-based description into the corresponding DFG. This new representation can be then used by the *High-Level Analysis* step or translated into the corresponding HDL through wrappers to the proper HLS tools. More details about this phase can be found in Section **??**.

A. Task Extraction

In this phase, the initial source code is analyzed by means of a compiler step (implemented inside the Mercurium compiler [**?**]) for the analysis of OpenMP `#pragma`s. For example, considering an edge detection application, based on the Canny algorithm, and the source code of the corresponding `main` function reported in the following.

```
loadImage("lena.ppm", &inImage, &width, &height);
for(v=1;v<width-1;v++)
  for(h=1;h<height-1;h++)
     #pragma omp task
     Scale(inImage, scaled, height, h, v);
for(v=1;v<width-1;v++)
  for(h=1;h<height-1;h++)
     #pragma omp task
     Blur(scaled, gaussed, dimh, h, v);
for(v=1;v<width-1;v++)
  for(h=1;h<height-1;h++)
     #pragma omp task
     Edge(gaussed, edged, dimh, h, v);
for(v=1;v<width-1;v++)
  for(h=1;h<height-1;h++)
     #pragma omp task
     Threshold(edged, outImage, height, h, v, 10);
writeImage("edge.ppm", outImage, width, height);
```

Based on the analysis of the `#pragma omp task` annotations, it is possible to identify four tasks: *Scale*, *Blur*, *Edge* and *Threshold*. Then, based on the information about the memory elements passed through the functions (e.g., the image arrays), it is possible to identify the interconnections among them. Finally, the remaining data structures represent the information exchanged with the software counterpart. The resulting task graph is thus reported in Figure **??**.

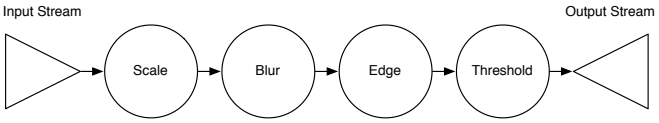


Fig. 2: Task graph extract for the Edge detection application. The four tasks are then further decomposed into their respective DFGs.

Note that, analyzing the information about the induction variables of the loops where the tasks are invoked, it is also possible to infer the number of repetitions for the kernel and the size of the data to be transferred. This will be very important to determine the proper implementation for the application, as described in the following.

B. Code Analysis

This section describes a C-to-C compiler step (implemented inside the Mercurium compiler [**?**]) that has been adopted to analyze the source code of each task and extract relevant information specified with custom pragmas. At the time being, it is possible to specify the following information:

- *memory access patterns*: it specifies a memory access to a block of data, along with specific information about the element to be accessed (i.e., the offset with respect to the current position within the data stream).
- data bit-witdh: it specifies the dimension of the data (in bit) which the algorithm has to operate on.

In particular, `#pragma asap access` is defined as:

```
#pragma asap access variable(<var>) \
                    offset(<index>) \
                    alias(<name>)
```

where the first parameter represents the name of the stream to be accessed, with the offset specified by the second one. Note that, since it represents streaming accesses, the offset is with respect to the current position and thus it can be either positive or negative. Then, it is also specified a alias name to be used to identify this access in the following transformation steps. For example, the following piece of code:

```
#pragma asap variable(array) \
             offset(-1) alias(array0)
#pragma asap variable(array) \
             offset(0) alias(array1)
int var = array[i-1] + array[i];
```

is translated as follows:

```
int array0 = array[i-1];
int array1 = array[i];
int var = array0 + array1;
```

and the corresponding relationship with stream variables and offsets are reported in an XML file.

On the other hand, the `#pragma asap bitwidth` is defined as follows:

```
#pragma asap bitwidth variable(<var>) \
                      size(<num>)
```

where the first parameter represents the name of the variable which the pragma refers to and the second one specifies the size in bits that the designer wants to implement.

For example, considering the source code of the *Scale* function, reported in the following, it is possible to specify information about the accesses to the streams as follows.

```
#pragma omp task
void Scale(
  unsigned char* original1,
  unsigned char* result,
  unsigned int dimh,
  unsigned int h,
  unsigned int v)
{
   unsigned char res;
```

```
    #pragma asap access variable(original1) \
                offset(-1) alias(original1_gs_1)
    #pragma asap access variable(original1) \
                offset(0) alias(original1_gs_2)
    #pragma asap access variable(original1) \
                offset(+1) alias(original1_gs_3)
    res = (original1[3*(v*dimh+h)-1]
           + original1[3*(v*dimh+h)]
           + original1[3*(v*dimh+h)+1]) / 3;
    #pragma asap access variable(result) \
                offset(0) alias(result_gs)
    result[v*dimh+h] = res;
}
```

C. DFG Generation

Finally, we implemented a transformation on the top of the LLVM compiler [**?**] (version 3.2) to generate the Data Flow Graph (DFG) of each task. In particular, it analyzes the internal representation produced by the compiler to identify memory operations, as well arithmetic ones and constant values. Then, dependencies are extracted and specified into the resulting graph. Information about input and output streams is represented as well.

Finally, this DFG representation is enriched with information about data bit-width, if specified by the designer. Considering the example cited above, an example of DFG representation is shown in Figure **??**.

IV. High-Level Analysis

High-level analysis enables the tool chain to estimate the design properties without going through the time-consuming synthesis flow for FPGAs, which normally takes hours to days to finish. Specific mapping between application representation, i.e., hierarchical Data-Flow Graphs (DFGs) and hardware implementations are built. The hierarchical DFGs for application shown in Figure **??** are shown in Figure **??**, where function nodes represent function tasks of applications, and DFGs inside the function nodes indicate the function operations. Graph nodes inside a function node can be divided into three categories: I/O nodes, data access nodes and arithmetic nodes.

Fig. 3: Configuration Data Flow Graph.

Arithmetic nodes are mapped as fully pipelined data-paths in hardware. Given required data, a fully-pipelined data-path generates one result every clock cycle. As data streamed into the data-path, all involved arithmetic operations are active, ensuring the theoretical peak performance of used resources. Resource consumption can be calculated by accumulating resources consumed by each arithmetic operator. For the arithmetic operators, bit-width optimization and arithmetic operation transformation are supported by the high-level analysis to optimize the implemented data-paths. Bit-width optimization refers to the operation to reducing resource consumption by increasing the finite-precision error. The relationship between the resource consumption and the finite-difference error for arithmetic operations in Figure **??** is demonstrated in Figure **??**, where results calculated from a single-precision CPU design are used as the reference. As the number of mantissa bits reduces, resource consumption decreases while the finite-precision error increases. The normalized resource usage factor a is operator-specific, and therefore depends on how many addition and multiplication operations a function node has. Arithmetic operation transformation refers to the optimization

Fig. 4: Configuration Data Flow Graph.

process where an arithmetic operator can be mapped either onto DSP blocks or LUTs/FFs pairs. When hardware design is limited by one resource type, arithmetic operations can be transformed into other types to release more limited resources. The same transformation ratio b is applied to all arithmetic operations inside a function node, with transformation ratio 1, 2 and 3 indicating respectively scenarios where all operators are mapped into LUTs/FFs pairs, multiplication operations are mapped into DSP blocks, and all operators are mapped into DSP blocks. Therefore, resource consumption for the mapped

Fig. 5: Configuration Data Flow Graph.

data-paths can be calculated with the bit-width optimization ratio a, the transformation ratio b and the number of arithmetic operations.

$$Ls = Par \cdot \sum_{i \in \odot} (N_{\mathrm{i}} \cdot a_{\mathrm{L},i} \cdot b_{\mathrm{L,i}}) \quad \odot = \{+, -, \bullet, \div\} \tag{1}$$

where N_{i} denotes the number of arithmetic operation i in the function, b_{L,i,r_b} refers to resource consumption of LUTs for arithmetic operation i represented with b mantissa bits, and $a_{\mathrm{L,i,r_a}}$ express different resource consumption ratio at different transformation ratio a. Similarly, resource consumption for FFs (Fs) and DSPs (Ds) can be estimated. Par indicates the design parallelism, i.e, how many data-paths are replicated. As full-pipelined data-paths cannot share resources with each other, the resource consumption for data-paths increases linearly when Par increases.

Data access nodes indicates the data access pattern of a function. In a hardware implementation, this refers to the data required by a data-path, at each clock cycle. These nodes can be directly linked to off-chip memory. However, the memory bandwidth requirements then increases linearly with the number of data access nodes and design parallelism, as overlapped data accessed at different cycles and by replicated data-paths cannot be shared. A on-chip memory architecture can be built to buffer and shared the access data, with data access nodes accumulated into mem. Therefore, when design parallelism Par increases, resource consumed by the memory architecture can be calculated as:

$$Bs = \bigcup_{i=1}^{Par} \mathrm{mem_i} \cdot R_{\mathrm{B}} \tag{2}$$

where $\mathrm{mem_i}$ is the accumulated memory architecture for data-path i, and R_{B} indicates the memory resources consumed for each buffered data. Moreover, as off-chip data are required at

each cycle to update the on-chip memory, bandwidth requirements can be calculated as difference between on-chip data at consecutive cycles.

$$Bw = \bigcup_{i=1}^{Par} \mathrm{mem_{i,j+1}} - \mathrm{mem_{i,j}} \cap \mathrm{mem_{i,j}} \quad (3)$$

where $\mathrm{mem_{i,j+1}}$ indicates the on-chip data for data-path i, at clock cycle $j+1$. With resource consumption and on-chip memory architecture estimated, the design parallelism Par can be calculated based on available resources for one reconfigurable region. The execution time for a function task thus can be calculated as:

$$T = \frac{ds}{f_{req} \cdot Par} \quad (4)$$

where ds is the data size for the function task and f_{req} is the operating frequency. Meanwhile, reconfiguration overhead can be estimated based on configuration bitstream size and reconfiguration interface throughput θ.

$$O_r = \frac{N_r \cdot \phi}{\theta} \quad : \quad N_r = \frac{R_I + R_{knl} \cdot P}{R_U} \in (1, 2, 3...) \quad (5)$$

where R_U indicates the resource consumption for one reconfiguration unit, ϕ accounts for the bitstream file for each consumed resource unit and N_r is the number of reconfiguration units used. For different reconfigurable platforms, the reconfiguration unit can be one reconfigurable slice, one clock region, or the entire chip. In this work, the reconfiguration unit is one clock region.

V. Mapping and Scheduling Analysis

After producing the estimates for the input application, these are integrated with the outputs of the application analysis and task graph extraction (refer to Figure **??**). The subsequent step of the toolchain is to feed these data into the mapper, which maps the tasks onto the heterogeneous reconfigurable architecture. In order to let the mapper know the structure of the target architecture, the reconfigurable architecture template chosen by the designer is fed into the mapper as well.

The aim of this work is to integrate the PR in the toolchain as a first class citizen, a degree of freedom which is effectively and automatically exploited by the framework to improve the performance of the resulting system. To do so, the mapper explores many different mapping combinations, extract execution times by means of simulation and employs a metaheuristic optimization technique to evolve the system towards better mappings, i.e. those which results in lower overall execution time.

More specifically, as already stated in Section **??**, the HLA allows the designer to estimate the resource consumption of each task in the task graph when different trade offs (for example between area and execution time) are sought. This allows to choose the right implementation for a task given differing global design objectives (like total execution time, peak power or average throughput). For example, one implementation of a given application might allow for a high throughput at the cost of large area consumption, while another might aim for area savings at the cost of limited throughput (due to different HLS strategies, for example). Unfortunately, the limited resources of reconfigurable devices forces the designer to limit – potentially, in a severe manner – the performance of the resulting system. In other situations, and in the general case, the whole of the application might not be entirely ported to the reconfigurable device. The aim of this work is to further exploit the limited availability of resources by *reusing* part of them. We demonstrate that the introduction of the PR allows the designer to augment the possibilities of the target device by specifying different trade offs between area usage while meeting otherwise non attainable performance goals.

The mapper considers the information contained in the architectural template, which specifies the number of available generic processors in the system as well as an upper bound to the number of reconfigurable regions that can be instantiated in the resulting architecture solution. During exploration, the mapper instantiates any number of reconfigurable regions that is appropriate to improve the resulting execution time, finding the best tradeoff between the time saved executing tasks in hardware and the time spent reconfiguring these tasks onto the corresponding reconfigurable regions. In doing so, the mapper enforces a constraint on the available resources, so that no resource consumption (for example, BRAMs or LUTs) exceeds the availability of the target platform. The resulting solution is therefore admissible by construction (while it might not necessarily be floorplannable). Moreover, if the resulting solution results in the assignment of only one task to a specific region, that region is automatically converted to a static hardware function (i.e. non reconfigurable).

More into details, the mapping process is divided in two steps. During the first one, the mapper uses a constructive approach to iteratively build the mapping trace. The solution is built by analyzing the set of currently ready tasks in the task graph (line **4,6**) and generating possible mapping choices (line **7-9**). A mapping choice is a 3-tuple composed of a task, a processor and implementation. This choice represents the execution of a task on a specific processor using a specific implementation of that task on that processor. Then, one of these choices (i.e. possible mappings for this task) is made (line **10-11**) and the dependences satisfied by the execution of the chosen task are resolved (line **12**). In order to guarantee that the computed solution is admissible, we introduce a constrain on the resources such that no choice is considered at any step of the algorithm if it would corresponds to a violation of any resource limit. To limit the complexity of the algorithm (the solution of the mapping problem is NP-hard, in general) we make a local exploration first (local step) and an iterative construction afterwards (iterative construction of the final solution). We don't consider all the possible combinations of all the possible meaningful mappings: while the computed solution is not guaranteed to be theoretically optimal, it is experimentally good enough for the exploration to be fast and meaningful.

After all the tasks have been scheduled exactly once, the mapping trace is done. During the second step (line **13**), this solution is evaluated against any number of metrics. In this work we focus on lowering the overall execution time of the input workload. The time taken to execute the mapped application is a reconfiguration aware scheduler for streaming applications. This component scans the mapping

trace to extract the list of task that are mapped onto every reconfigurable region and remodels the task graph by adding reconfiguration tasks as needed, i.e. between successive tasks scheduled on the same reconfigurable region. Communication tasks are introduced, too, and considered during start and end times assignments. After this, it analyzes the dependencies between tasks and simulates their execution taking into account the partial overlap of the execution of these tasks due to their streaming/dataflow nature as well as communication related considerations. After assigning the proper start and end time, the total execution time is devised and the mapping solution is ranked (the lower the execution time, the better). This information is used to guide the mapper towards an improved solution during successive iterations of the optimization algorithm (line **14-16**).

Algorithm 1: Overview of the mapping algorithm

```
   input : A task graph and a reconfigurable architecture
   output: A mapping trace
 1 forall the iterationsNum do
 2     currentBestSolutions ← ∅
 3     forall the iterSolutionsNum do
 4         tSet ← tasksWithoutPredecessors ()
 5         sSet ← ∅
 6         mappingTrace ← ∅
 7         while tSet ≠ ∅ do
 8             forall the T ∈ tSet, I ∈ iSet,P ∈ pSet do
 9                 pChoice ← generateChoice (T, I, P)
10             mappingChoice ← roulette (pChoice)
11             mappingTrace.add (mappingChoice)
12             tSet ← resolveDependencies ()
13         solution.runtime ← simulate (currentMapping)
           allSolutions.add (solution)
14     currentBestSolutions.add (selectBest (allSolutions))
15     improveNextIteration (currentBestSolutions)
16 solution = selectSingleBest (allSolutions)
17 return solution.trace
```

The framework allows the designer to choose from many different kinds of optimization algorithms to implement the constructive approach, and for any desired metric to be employed to evaluate the solution constructed in the first phase. While we employed an optimized and reconfiguration aware algorithm known in literature as Ant Colony Optimization (ACO) [**?**] (a technique known to be successfully used in the context of mapping problems [**?**]), it is possible to employ any other kind of optimization algorithm. In the same way, the framework allows the designer to specify any kind of metric over the computed solutions and any kind of objective function over those metrics in order to rank them.

As already stated, an interesting behavior resulting from the introduction of the degree of freedom of PR is the possibility to explore mapping solutions which lead to designs whose performance metrics would otherwise be not obtainable. Specifically, consider a highly data parallel application, which can be decomposed into stages. This is a really common case: signals and text processing applications, multimedia codecs or financial modeling are all among the common cases of highly data parallel applications and algorithms. In this case, if the stage can process more data at once, at the cost of increased area consumption, the throughput is theoretically limited only by the amount of reconfigurable resources at disposition. The idea here is that by adding PR, even though the entire pipeline might not be completely implemented in hardware, it might be possible to instantiate, instead, larger single stages onto reconfigurable regions. After executing the single, more performant stage and stored the intermediate results, PR can occur and new stages instantiated on the reconfigurable device. Mapping this way finds more interesting trade offs between resource consumption and execution time, as Section **??** confirms.

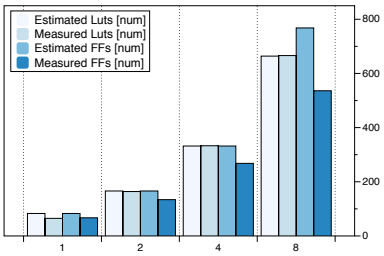
VI. Experimental Validation

We experimentally validate the overall framework on a typical dataflow application, an image analysis algorithm, namely Canny edge detection. The structure of the task graph of this application is made of four sequential tasks as reported in Figure **??**. Our reference architectural template is a commercial supercomputing platform based on reconfigurable hardware, a Maxeler® MaxWorkstation, featuring a Virtex 6 FPGA. The proposed method is evaluated in terms of efficiency of the high-level analysis and improvements of mapped hardware designs.

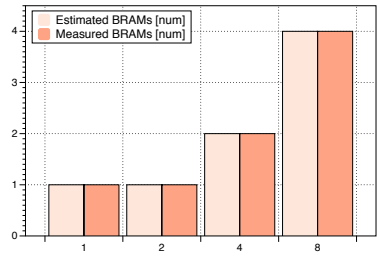
The high-level analysis efficiency is expressed as accuracy of the estimations. Analyzed results for the Canny edge detection are shown in Figure **??**. To simplify the comparison with measured results, the transformation ratio b (see Section **??**) in this application is set to 1, i.e., all arithmetic operations are mapped into LUTs/FFs pairs. As the graphic input data range is bounded between 0 and 255, input and output data are represented with 8-bit fixed-point data, and the bit-width optimization ratio a are correspondingly calculated for operators in each data-path. As shown in Figure **??**, the high-level analysis captures the properties of implemented data-paths, with more than 90% accuracy. For the analyzed data access operations, the constructed on-chip memory architectures buffer and share the data accessed at different cycles, and the calculated off-chip bandwidth requirements align with the measured memory bandwidth usage. In our work, the memory usage is calculated as the ratio between access data and the overall execution time.

In Figure **??** resource consumption, execution time and reconfiguration time overhead are listed, when design parallelism increases from 1 to 8 within a reconfigurable clock region. When design parallelism (Par) increases, resources consumed by data-paths increase linearly to process multiple input data at a cycle. The memory resource consumption, on the other hand, stays at the same level as overlapped data accessed by the replicated data-path are shared in the same memory architecture, increasing the data reuse factor. The consumed BRAMs increase from 1 for $Par = 1$ to 4 for $Par = 8$, as the constructed memory architecture needs more memory ports to support parallel data access from replicated data-paths. As shown in Figure **??**, the estimated execution time can accurately approximates the measured results. The execution time reduces linearly from $72.4ms$ to $9.1ms$, when design parallelism increases from 1 to 8. The reconfiguration overhead, on the other hand, stays almost the same when design parallelism increases, as the reconfiguration unit is limited to 1 clock region, and the 4 function modules can fit into 1 clock region individually when design parallelism is 8.

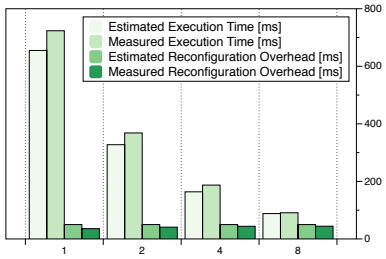
In order to demonstrate the utility of PR, we devised the following experiment. We created four different implementa-



(a) Estimated and measured number of FFs and LUTs



(b) Estimated and measured number of BRAMs



(c) Estimated and measured execution and reconfiguration times

Fig. 6: Experimental results. Notice the similarity between the estimated results and the measured ones.

tions, one per each of the four tasks in the task graph. These four implementations process 1, 2, 4 and 8 words of the input data at once, respectively, at the cost of a proportionally increased amount of area and resource consumption (refer to Figure **??**). We could provide larger implementations, but due to long synthesis time we preferred to scale the problem, constraining the actual available area and synthesizing relatively smaller implementations. Even though the problem is scaled, the same conclusions apply when all the FPGA is employed and larger and more performing implementations are used.

With the high-level analysis module properly capturing design properties for various data size and design parallelism, the mapper can accurately predict the outcome of various mapping solutions.

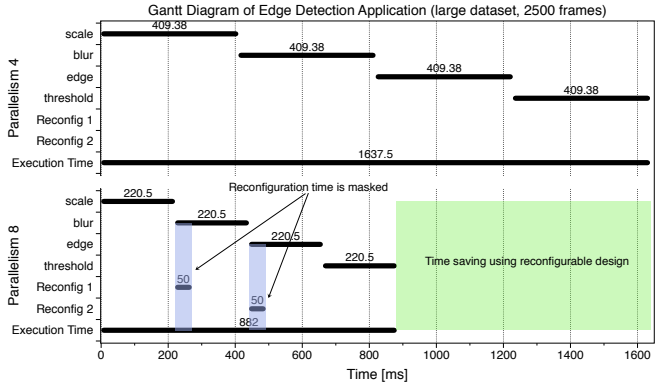


Fig. 7: Comparison between execution times of two reference implementations. Note how the reconfigurable design (bottom) performs better than the static design (above) under an area constraint of $10,000$ luts.

Then, we defined two architectural templates. The first one is the static one, featuring 4 static regions. These are automatically mapped with exactly one of the four tasks in the task graph. The area consumption is the sum of the resource consumptions of the implementations mapped on the respective static areas. Given that we constrained the area of the FPGA dedicated to processing to $10,000$ LUTs, the pipeline with the highest throughput that we could implement is the one which processes 4 pixels at once: the area consumption of all the four stages is $332+3,840+3,688+32 = 7,892$ (in fact, the pipeline processing 8 pixels at once surpasses $15,000$ LUTs). The total execution time of this pipeline is a function of the dimensions of the dataset. Using an input of size 100, $1,000$ and $10,000$ frames of $256 * 256$ pixels, we obtain a total execution time of 65.5, 655 and $6,550$ ms, respectively.

On the other hand, the architectural template of the reconfigurable system features only 2 reconfigurable regions (R0 and R1) available for tasks mapping. In this preliminary example, the mapper explores in a few iterations all possible mappings and determines that the best one is R0:<scale, threshold> R1:<blur, edge>. This mapping requires 664 LUTs for R0 $(max(LUTs(scale), LUTs(threshold)) = max(664.64))$ and 7680 for R1 $(max(LUTs(blur), LUTs(edge)) = max(7680, 7376))$, a resource consumption comparable to the static architecture totaling $8,344$. The execution time is computed as time to execute A, time to reconfigure R0 to C, execution time of B, reconfiguration of R1 to task D, execution of C an D. Considering the same inputs sizes of 100, $1,000$ and $10,000$ frames, the corresponding execution times are 135.28, 452.8 and $3,628$ ms, respectively.

The results show that reconfigurable designs do perform better than static designs in specific cases. Specifically, we see that the reconfigurable designs perform better than the static ones when the execution time of a stage in the pipeline is long enough to mask the reconfiguration time of any reconfigurable region (which is not the case for input size 100), and when the adoption of more resource hungry implementations lead to reduced execution times. In this case the adoption of PR allows the FPGA to be virtually larger than what it really is, allowing for more performing implementations to be used.

Referring to Figure **??** it is easy to see how with parallelism 8, during the execution of `blur` and `edge` the reconfiguration time is completely amortized, given that tasks are sequential and other tasks cannot execute in parallel. Moreover, the second condition indicates that using a more resource hungry – and more performant – implementation, we can execute the application on a device virtually larger than the actual one, potentially leading the design to be more performant than the static design – which is the case in Figure **??**. The conclusion is that under the imposed area constraint of 10.000 LUTs, the reconfigurable design is the most performant.

VII. Conclusions and Future Work

In this work we presented a novel framework aimed at designing reconfigurable hardware based computing systems. We propose a toolchain that, starting from the input, translates the application into the corresponding task graph with explicit producer/consumer relationships marked on the graph. Afterwards, the task graph is analyzed so that estimates of resource consumption for all the tasks are extracted. Finally, these data and an architecture template are fed into the reconfigurable aware mapper and scheduler to generate the final design solution. We expect to validate all the approach and the generated solutions by targeting an actual device, like the Zynq Board, which will require us to extend the toolchain with an automatic generator of a runtime manager. Moreover, we expect to introduce means to make it possible to integrate the generation of the implementations with an High Level Synthesis (HLS) tool, like Xilinx's Vivado Suite [**?**].

Acknowledgment

This work was supported by the European Commission in the context of FP7 FASTER project (#287804).