mRTS:Run-Time System for Reconfigurable Processors with Multi-Grained Instruction-Set Extensions

Waheed Ahmed, Muhammad Shafique, Lars Bauer, and Jörg Henkel
Karlsruhe Institute of Technology, Chair for Embedded Systems, Karlsruhe, Germany
{ahmed.waheed,muhammad.shafique, lars.bauer, henkel} @ kit.edu

Abstract:
We present a run-time system for a multi-grained reconfigurable processor in order to provide a dynamic trade-off between performance and available area budgets for both fine- as well as coarse-grained reconfigurable fabrics as part of one reconfigurable processor. Our run-time system is the first implementation of its kind that dynamically selects and steers a performance-maximizing multi-grained instruction set under run-time varying constraints. It achieves a performance improvement of more than 2x compared to state-of-the-art run-time systems for multi-grained architectures. To elaborate the benefits of our approach further, we also compare it with offline- and online-optimal instruction-set selection schemes.

1. Introduction and Related Work

Reconfigurable computing may be categorized into *coarse-grained* and *fine-grained* [1][2]. A coarse-grained reconfigurable processor uses an array of reconfigurable ALUs coupled with the core processor, which is amenable to data-dominant applications with streaming and (sub) word-level processing (e.g. *add*, *subtract*, *multiply* etc.). Prominent examples are ADRES [3], PACT XPP [4], Montium Tile Processor [9], and ElementCXI's ECA computing fabric [10]. These architectures target specific domains like multimedia and communication etc. However, they are performing inefficiently when it comes to control-dominant applications with bit/byte-level processing. Fine-grained reconfigurable processors (like Chimaera [15], XiSystem [16], and RISPP [6]) deploy an embedded FPGA as an extension to core processor, to accelerate control-dominant applications and bit/byte-level processing (bit shuffling, packing, merging, etc.). However, these fine-grained may architectures underperform for data-dominant applications on word level.

As future embedded applications possesses heterogeneous processing behavior (i.e. both control- and data-dominant), therefore, recently so-called multi-grained reconfigurable architectures (like 4S [7], MORPHEUS [8] and KAHRISMA [5]) have evolved that integrate both fine- and coarse-grained reconfigurable fabrics on a single chip. These processors accelerate both data- and control-dominant applications, thus offering further flexibility and efficiency compared to solely coarse-grained or solely fine-grained architectures. Often real-world applications consist of diverse functional blocks that may contain diverse computational kernels[1]. Instruction Set Extensions (ISEs) for reconfigurable processors are employed to accelerate the kernels of different applications. These ISEs consist of data paths that may be reconfigured on a coarse-grained and/or a fine-grained reconfigurable fabric. A kernel can have different ISEs that offer different performance improvements for different utilizations of the reconfigurable fabric. This brings flexibility but realization of such scheme requires additional computational time and resources.

The challenge is to determine at run time which ISEs should be reconfigured using which type of fabric (coarse- or fine-grained) in order to achieve the highest performance for a particular kernel for a given amount of (coarse- and fine-grained) reconfigurable fabric. State-of-the-art approaches (e.g. [7], [8], [13]) select the ISEs at compile time after performing an extensive evaluation of an applications' processing behavior. However, the decisions undertaken at compile time hamper these approaches to perform efficiently (in terms of performance, for instance) when considering run-time variations of (a) the application execution properties (i.e. execution frequency of ISEs), (b) the available fine- and coarse-grained reconfigurable fabric (shared among various tasks), and (c) input data properties (e.g., in audio or video processing applications).

The scheme in [11] manages coprocessor reconfigurations on a fine-grained reconfigurable fabric at run time. However, this scheme operates at the task level and thus suffers from inefficiency when targeting applications that exhibit adaptivity at a finer level of granularity, e.g. at the functional block level. The fine-grained RISPP architecture [6] offers run-time adaptivity at functional block level but it is less efficient when executing applications that are suited for coarse-grained architectures. Apart from lack of support for the coarse-grained fabrics, another reason for the inefficiency of approaches [11] and [6] when targeting multi-grained ISEs is their cost function. These approaches are aimed to optimize considering the longer reconfiguration time of the fine-grained reconfigurable fabric (in ms), thus they do not provide good results when considering the significantly less reconfiguration time (in µs) of coarse-grained fabrics.

Hence **a run-time system for multi-grained reconfigurable processors is desirable** that can cope with the distinct reconfiguration and the processing nature of heterogeneous reconfigurable fabrics supporting multi-grained ISEs. Such a run-time system needs to determine the selection and execution decisions of multi-grained ISEs for each functional block by jointly considering all of its kernels *at run time*.

To demonstrate the effects of run-time varying scenarios (which have an impact on the available reconfigurable fabric and the kernel execution frequency) on the selection of multi-grained ISEs, we first discuss a case study of a real-world example of the Deblocking Filter from an H.264 video encoder [17].

2. Motivational Case Study

The ISEs for the H.264 Deblocking Filter are composed of two types of data paths with distinctive computational properties:

a) a control-dominant *condition* data path with bit-level operations suitable for fine-grained reconfigurable fabric, and
b) a data-dominant *filter* data path with arithmetic (sub) word-level operations suitable for coarse-grained reconfigurable fabric.

Depending upon the available area of the coarse- and fine-grained reconfigurable fabrics, different ISEs may be realized for the H.264 Deblocking Filter, each providing a certain area vs. performance trade-off. For simplicity, this case study only discusses the following three ISEs:

ISE-1: The *filter* and *condition* data paths are implemented on fine-grained reconfigurable fabric.
ISE-2: The *filter* and *condition* data paths are implemented on coarse-grained reconfigurable fabric.
ISE-3: The *condition* data path is implemented on the fine-grained reconfigurable fabric and the *filter* data path is implemented on the coarse-grained reconfigurable fabric.

Due to varying computational properties and reconfiguration latencies of the two fabrics[2], the execution- and reconfiguration latencies of these ISEs differ from each other, which leads to different performance improvements. Eq.1 formulates the *Performance Improvement Factor (pif)* of each ISE, where *sw_time* represents the time to ex-

[1] the compute-intensive loops, which are executed most often in a program.

[2] reconfiguration time of a single data path in fine-grained reconfigurable fabric is around 1.2ms, while the reconfiguration of the same data path on coarse-grained fabric takes approximately 0.00015ms.

978-3-9810801-7-9/DATE11/©2011 EDAA

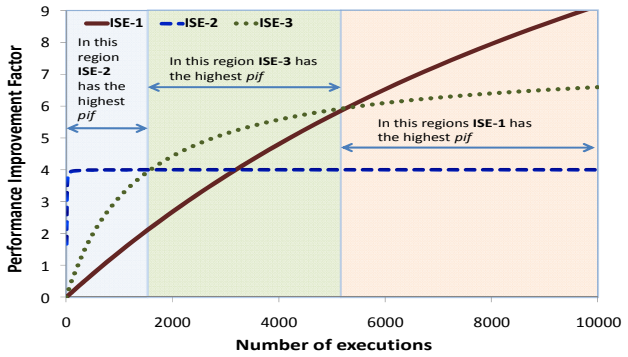

Fig. 1: Performance improvement factor of three different Instruction Set Extensions (ISEs)

ecute the kernel in RISC-mode[3] and *hw_time* is the time to execute the kernel on the reconfigurable fabric using ISEs.

$$pif = \frac{sw\_time * executions}{reconfiguration\_latency + hw\_time * executions} \quad (1)$$

Since the reconfiguration latency represents a fixed overhead, the overall performance improvement depends upon the total number of kernel executions (that may vary at run time, as we will discuss later in this section).

Fig. 1 shows the *pif* of the three ISEs for varying number of kernel executions. Each particular ISE has a higher *pif* value than the other two in a certain range of kernel executions (as highlighted by the three different regions). Due to its lower reconfiguration time, ISE-2 provides a higher *pif* for a lower number of executions when compared to ISE-1 and ISE-3. On the contrary, ISE-1 outperforms the others for a relatively large number of executions, where its longer reconfiguration time can be amortized. ISE-3 is a compromise between the above two scenarios. Fig. 2 shows the excerpts of execution behavior of the *Deblocking* filter over time, where the plotted values correspond to the execution numbers in each subsequently encoded frame (showing different iterations of the kernel). It can be observed that the performance-wise best ISE during one iteration of the kernel does not remain the best option for the next iteration as the number of executions within an iteration may change due to changing workload characteristics of the application. The experimental setup for the case study in Fig. 1 and Fig. 2 is explained in Section 5.1.

Different ISEs require different amount/type of reconfigurable fabric and offer different performance improvements. The problem of selecting the performance-maximizing set of ISEs becomes complex when an application consists of multiple functional blocks where each functional block may have several kernels. Each kernel may be accelerated by using one out of multiple possible ISEs, as each ISE can be composed of different data paths that may be used in different quantities. In the above example, we focused on one functional block of the H.264 encoder. The complete encoder contains in fact three functional blocks where the biggest one contains more than six kernels. Moreover, there are cases where the number of ISEs may reach up to 60 for a single kernel.

Summarizing: From this case study we can conclude that a runtime system is desirable that is able to:

- select a set of ISEs that provides a good trade-off between area and performance by exploring the search space of multi-grained ISEs during run time and
- provide the demanded adaptivity to cope with changing application requirements at run time without influencing the performance improvement procured by ISEs.

Our Novel Contributions: We present a run-time system for multi-grained reconfigurable processors (like [5], [8]). It dynamically selects a set of ISEs of a given functional block under (above-mentioned) run-time changing scenarios while maximizing the overall application performance. In particular, the proposed run-time system enables adaptivity in a multi-grained reconfigurable processor through:

- *dynamically exploring* the search space of multi-grained ISEs and *selecting* a set of ISEs that provide – at run time – a good trade-off between performance and available fine- and coarse-grained reconfigurable fabric under varying constraints,
- *a profit function*, to determine the benefit of each ISE while considering the reconfiguration latency, performance improvement, and the distinct computational properties of the coarse- and fine-grained reconfigurable fabrics, and
- *an Execution Control Unit* that steers the execution of application kernels by using *intermediate ISEs* (of the selected ISE) that require less reconfigurable fabric but still procure considerable performance improvement compared to RISC-mode execution.

We present a detailed evaluation of our approach for diverse reconfigurable processors and provide a comparison with state-of-the-art approaches MORPHEUS [8] and RISPP [6] (see Section 5). The results in Section 5.2 also demonstrate the suitability of our run-time system for different reconfigurable processors.

Before proceeding to our core idea, we present a brief overview of our multi-grained reconfigurable processor [5], which is required to introduce our novel run-time system in Section 4.

[3] executing the kernel using the basic instruction set of the core processor.

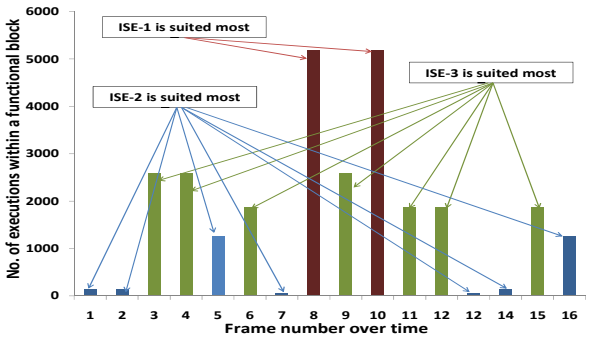

Fig. 2: Execution behavior of the H.264 Deblocking filter

3. The model of a Multi-Grained Reconfigurable Processor

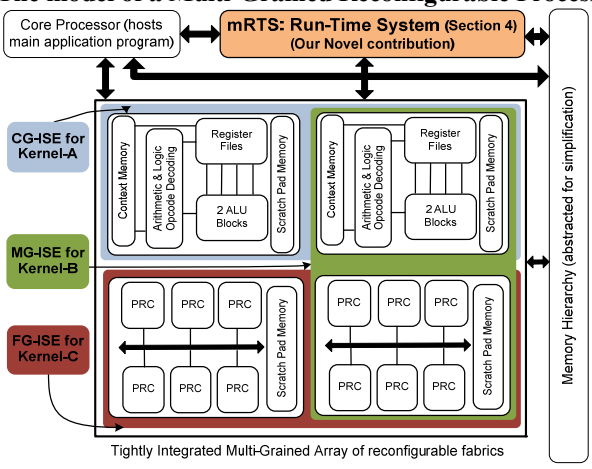

Fig. 3: Multi-Grained Reconfigurable Processor

Fig. 3 presents an overview diagram of our multi-grained (MG) reconfigurable processor [5] integrated with our novel run-time system (mRTS). It constitutes a core processor, memory system, and an array

of tightly coupled fine-grained (FG) and coarse-grained (CG) reconfigurable fabrics. The two ALUs can be used in parallel. The FG-fabric is realized as an embedded FPGA, which consists of partial reconfigurable parts donated as *Partially Reconfigurable Containers* (PRCs). Both FG- and CG-fabrics have dedicated scratch pad memories – connected to the memory hierarchy – to allow for fast data access and to store intermediate results.

A core processor hosts the main application while different FG- and CG- fabrics may be combined at run time to realize CG-, FG- or MG-ISEs of kernels. Fig. 3 presents combination scenarios for three different kernels (Kernel-A, Kernel-B, Kernel-C) to realize their CG-, MG, and FG- ISEs, respectively.

4. mRTS: Run-Time System for Multi-Grained Fabrics

A key to adaptivity in multi-grained reconfigurable processors is our novel run-time system (see Fig. 3), which dynamically selects an ISE for each kernel in a given functional block to maximize the overall performance. The run-time system reacts to different scenarios while considering the system's constraints, the current workload on the system, and the utilized resources. It therefore enables the possibility to operate the system to perform efficiently at any time during execution (better than a static analysis, which do not have accurate run-time information). We will discuss the details on the associated overhead of the run-time system in Section 5.4. In the following, we will focus on its functionality and design flow. Fig. 4 presents the high-level view of the run-time system showing its three main components, i.e. the *Monitoring & Prediction Unit* (MPU), the *ISE selector*, and the *Execution Control Unit* (ECU).

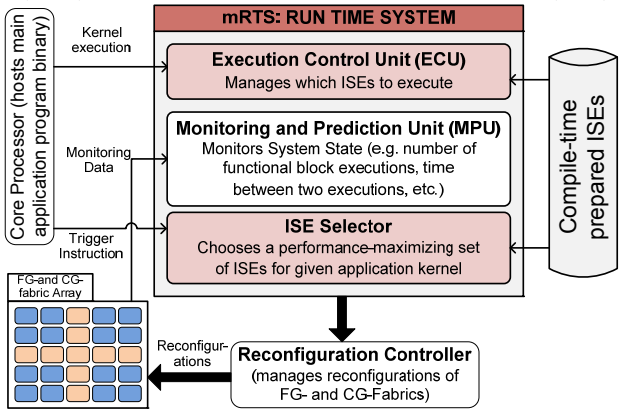

Fig. 4: Overview of our run-time system for multi-grained fabrics

At compile time, different ISEs for each kernel of an application are arranged[4]. We use our proprietary automatic tool chain to generate the CG- FG- and MG-ISE of prepared ISEs by designing their data paths for CG-fabric or FG-fabric. Each of these ISEs require different amounts and type of resources (FG- or CG-fabrics, thus require different reconfiguration latency) and provide different levels of performance improvement over the RISC-mode execution (i.e. kernel executing on the core processor without any accelerating data paths). At compile time, the amount of reconfigurable fabrics is fixed and known. Therefore, all non-fitting ISEs (requiring more fabric than available) are filtered out at this stage.

The application programmer embeds so-called *Trigger Instructions* into the application binary (incorporated as of assembler instructions) to forecast the kernel executions in the upcoming functional block. These trigger instructions contain the IDs of the requested kernels, their corresponding expected/estimated number of executions, and the average time between two consecutive kernel executions. The relative correctness of these numbers affects the quality of the run-time selection decision. They are initially obtained from an offline profiling and at run time the MPU monitors and updates them. Since the number of kernel executions may change at run time (due to, for example, changing input data), we have implemented a lightweight error back-propagation scheme [12] in our run-time system that updates the monitored values. Apart from keeping the track of execution counters, the MPU also keeps the track of the available amount of reconfigurable fabric at any time during execution.

In the following sections, we will discuss the two key components (i.e. ISE selector and ECU) of our run-time system in detail.

4.1. Instruction Set Extension Selector

The objective of the Instruction Set Extension (ISE) selector is to maximize the performance of a given functional block by jointly considering the ISEs of the kernels predicted in the trigger instruction. The core processor activates the ISE selector when it encounters a trigger instruction. The ISE selector uses the information available in the trigger instruction and the amount of available reconfigurable resources to select a set of ISEs, while considering the reconfiguration overhead and performance improvement provided by different ISEs. The number of CG-fabric (N_{CG}) and the total number of PRCs in all FG-fabrics (N_{PRC}) are known to the ISE-Selector while the set of trigger instructions is provided as input to the ISE selector. The trigger instructions are represented as 4-tuples $\{K_i, e_i, t_{fi}, t_{bi}\}$, where K_i is the i^{th} forecasted kernel in the functional block, e_i is the corresponding expected number of executions, t_{fi} is the time until the first execution, and t_{bi} is the average time between two consecutive executions as visualized in Fig. 5 (details will be explained later in this section). The task of the ISE selector is to find the set *'S'* of ISEs considering the following constraints:

- the selected set of ISEs in the set *S* must fit into the available CG- and FG-fabrics
- for each kernel there is exactly one ISE in the selected set *S*

To find the best solution, we need to use an optimal algorithm. It involves a) evaluating all combinations of ISEs, b) calculating the profit of each combination, and c) selecting the combination with the best profit while meeting the resource constraint. An optimal algorithm prunes the combinations of ISEs that do not meet the resource constraints. We learned from our experiments that for six kernels of the H.264 video encoder, there are more than 78 million combinations. Due to its high computational overhead, such an optimal algorithm is not feasible when considering the run-time nature of the ISE selection. Therefore, we use this optimal algorithm merely to evaluate the quality of our proposed ISE selector.

The multi-grained ISE selection is implemented using a heuristic algorithm. We have learned from our experiments that the number of kernel executions differs within each functional block due to changing input data. Consequently, not all of the kernels equally contribute to the overall performance improvement of the functional block. We aim to assign more resources to the kernels that contribute more towards the overall performance improvement. In summary, the following is required:

a) a profit function that calculates the performance improvement contribution of each ISE while considering its reconfiguration time and possible number of executions and
b) a selection algorithm that finds 'good' results and is referred to as the *ISE selection algorithm* throughout this paper.

Analyzing the profit function:

The expected profit of an ISE is actually the performance improvement offered by it in a given functional block. It corresponds to the difference(cycles saved) between the time required by an ISE and the RISC-mode execution of a kernel. Intermediate ISEs are the extensions that are realized using a subset of data paths of a specific ISE. They may become available due to the completed reconfiguration of a subset of data paths of a specific ISE, or due to the completed reconfigurations of other ISEs that share some data paths with the specific

[4] For further details about compile-time instruction-set extensions see the approaches proposed in [18] and [19].

ISE. Therefore, several intermediately available ISEs may be used for the execution of a kernel before the selected ISE is completely reconfigured. Since the reconfiguration of data paths of each ISE is completed at different points in time, the profit is the sum of potential performance improvements by the ISE and its intermediate ISEs. Fig. 5 presents an example scenario of the execution pattern of a kernel. The inputs to the profit function are e, t_{fi}, and t_{bi}, that are obtained from the trigger instruction. The ISE_1, ISE_2, …, ISE_{n-1} are the intermediate ISEs and ISE_n is the ISE with all of its data paths reconfigured.

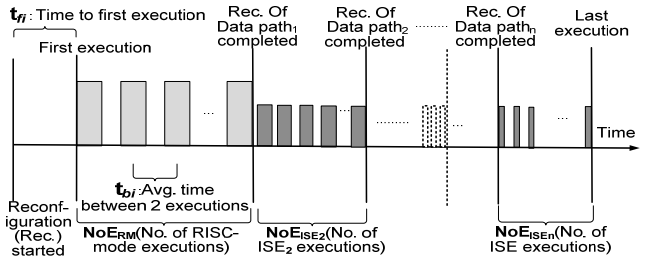

Fig. 5: The execution behavior of an ISE

The sizes of the rectangular boxes in Fig. 5 symbolize the kernel execution time. The rectangular boxes become smaller with every data path configured to symbolize the reduced execution latency. NoE_{RM} denotes the number of the RISC-mode executions. NoE_{ISEi} are the executions using *{Data path$_1$, Data path$_2$, …, Data path$_i$}* of the i^{th} intermediate ISE. Performance improvement (*per_imp*) of the i^{th} intermediate ISE is given by Eq. 2.

$$per\_imp(i) = NoE(i) \times (latency\_RM(ISE_n) - latency(ISE_i)); \quad (2)$$

The function *NoE()* returns the number of executions of the i^{th} intermediate ISE, *latency_RM()* returns the number of cycles if the kernel was executed in RISC-Mode, and *latency()* returns the number of cycles required to execute the i^{th} intermediate ISE. The term *NoE* changes during run-time of an application and it is calculated in Eq. 3.

$$NoE(i) = \begin{Bmatrix} \frac{recT(ISE_{i+1}) - t_{fi}}{latency(ISE_i) + t_{bi}} : recT(ISE_i) \le t_{fi} \le recT(ISE_{i+1}) \\ \frac{(recT(ISE_{i+1}) - recT(ISE_i))}{(latency(ISE_i) + t_{bi})} : recT(ISE_i) \ge t_{fi} \end{Bmatrix} \quad (3)$$

The function *recT()* returns the reconfiguration time of the i^{th} intermediate ISE. Note that the term *NoE* depends on the input parameters t_{fi} and t_{bi}. If the reconfiguration time the of i^{th} intermediate ISE is greater than t_{fi} then it means it will only be executed as long as the $(i+1)^{th}$ intermediate ISE is not available. Therefore, dividing the difference between the reconfiguration time of the i^{th} and $(i+1)^{th}$ intermediate ISE by the latency of a single execution will give the expected number of total executions. Similarly, when the reconfiguration time of the i^{th} intermediate ISE is less than t_{fi} then we take a difference between t_{fi} and the reconfiguration time of the $(i+1)^{th}$ intermediate ISE as that is the time when the i^{th} configured intermediate ISE will be executed. While calculating *NoE,* we have also incorporated the average time between two consecutive kernel executions as a part of denominator. Finally, the profit of an ISE is given by Eq. 4.

$$profit = \sum_{i=1}^{n-1} per\_imp(i) + (latency\_RM(ISE_n)) \times (e - \sum_{i=1}^{n-1} NoE(i)); \quad (4)$$

Eq.4 shows that the total expected profit of an ISE is the sum of performance improvements offered by the ISE and the intermediate ISEs with all its data paths reconfigured, where the number of ISE executions is obtained by the difference between the total executions of intermediate ISEs and the total number of expected executions '*e*'.

ISE selection algorithm:

The goal of our ISE selector is to select a set of ISEs while maximizing the combined profit of the ISEs of all kernels as shown in Eq.5.

$$\text{maximize} \sum_{\forall i, ISE:\{ISE\} \in S \cap Kernel_i} profit(ISE, e_i, t_{fi}, t_{bi}) \quad (5)$$

Fig. 6 shows the flow-chart of our ISE selection algorithm, which is based on the above-mentioned heuristic. The steps are:

Step-1: Make a *candidate list* of the ISEs of all kernels in the TIs.

Step-2: Remove ISEs from the candidate list that (a) require more reconfigurable fabric than available, and (b) are covered by data paths that are available from the already selected ISEs.

Step-3: Compute the profit of each ISE in the candidate list and then select the ISE with the maximum profit.

Step-4: Add the selected ISE to the output set, update the reconfigurable hardware status, and remove all other ISEs of the same kernel from the *candidate list*.

We can see from the steps that the ISE with the maximum profit is selected first; therefore, it obtains the required resources. Once the ISE is selected for a particular kernel, then it is considered as the final selection even though there might be other combinations that could provide a better overall profit.

The complexity of the above algorithm in **Fig. 6** is reduced from $O(N^M)$ (as is the case in the optimal algorithm) to $O(N*M)$, where N is the number of kernels within a particular functional block and M is the number of compile-time prepared ISEs for each kernel. Afterwards, the ISE selector forwards the set of selected ISEs to the *reconfiguration controller* that manages the reconfiguration process and the configuration state of CG- and FG-fabrics.

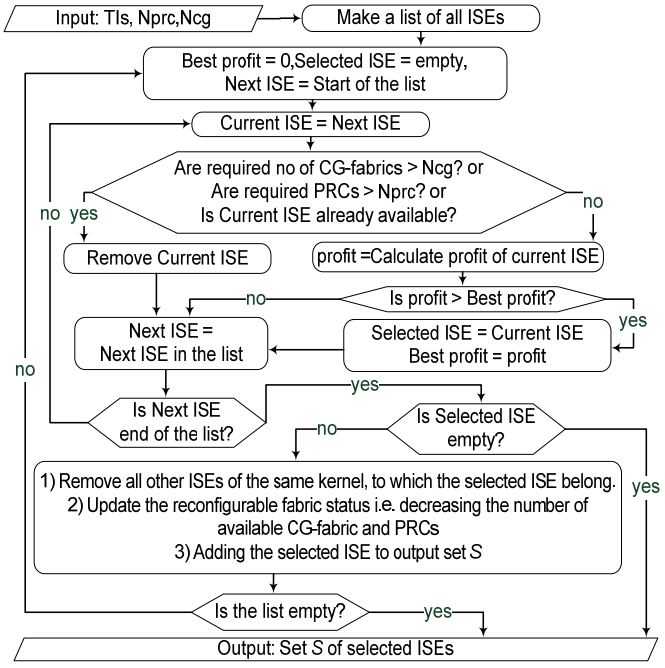

Fig. 6: Flow chart of our ISE selection algorithm

4.2. Execution Control Unit (ECU)

The ECU presents another aspect of adaptivity by steering the execution of kernels. The ECU exploits the flexibility that a kernel may be executed upon using different intermediate ISEs. The FG- and MG-ISEs are only executable when the reconfiguration of their fine-grained data paths is completed. The delay between the RISC-mode execution and the first accelerated execution using either intermediate ISE or selected ISE is significantly larger. To bridge this delay we have introduced a special kind of extension, which implements a full kernel on one of the free CG-fabrics. Such an extension is termed as *monoCG-Extension*. The monoCG-Extension uses both ALUs and register files of CG-fabric to expedite the kernel, which is still faster than a RISC-mode execution. Note, that the reconfiguration time of a CG-fabric is insignificantly low; therefore, it will be readily available

after few RISC-mode executions. Fig. 7 summarizes the operational flow of the ECU.

a) When a kernel is executed in the program binary, the core processor triggers the ECU. It first checks the availability of the selected ISE (i.e. it checks whether all constituting data paths of that ISE are completely reconfigured or not).
b) If the selected ISE is available, the ECU will execute. Otherwise, the ECU checks for the availability of the intermediate ISEs.
c) If no intermediate ISE is available, the ECU checks for a free CG-fabric to realize a *monoCG-Extension.*
d) In case no data path is reconfigured and no CG-fabric is available at a certain point in time during execution, the ECU executes the functional block in RISC-mode.

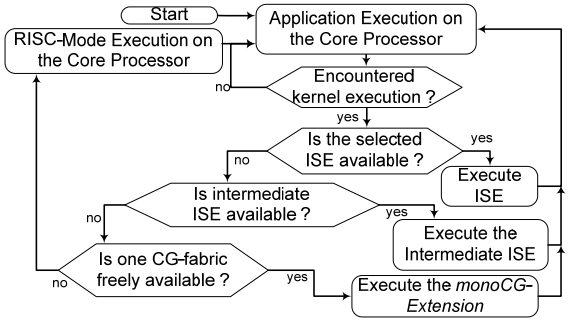

Fig. 7: Flow of the Execution Control Unit

5. Implementation, Experimental Results and Analysis

In this section, we will present our experimental setup and the details of reconfigurable fabrics. Later we will provide the comparison with state-of-the-art and a detailed analysis of mRTS.

5.1. Experimental Setup and Reconfigurable Fabric

The core processor is a LEON processor, a 32-bit synthesizable processor core based on the SPARC V8 architecture and the CG-fabrics are operating at 400 MHz, while the FG-fabrics (i.e. Virtex-4 FPGA) are operating at 100 MHz. The reconfiguration bandwidth of the FG-fabric is 67584 KB/s. Each CG-fabric can store multiple contexts and a context switch takes 2 cycles. The instruction size of the CG-fabric is 80 bits. Instructions can be streamed into the context memory that can store up to 32 instructions. Typical ALU operations like *add, sub, or,* etc. are executed in a single cycle while a *multiply* instruction takes 2 cycles. The *divide* instruction takes 10 cycles. Each CG-fabric provides a zero overhead loop instruction. The 32-bit load/store unit is virtually available to each CG-fabric, while each FG-fabric is provided with 128-bit load/store unit. There are two 32-bit register files with 32 registers per register file in each CG-fabric. Implemented is a point-to-point connection between the CG-fabrics. Communication between them requires 2 cycles. Communication within FG-fabric, i.e. between PRCs requires only a single cycle.

The complete system is simulated on our cycle-accurate instruction-set-simulator. Its inputs (i.e. the data-path latency/reconfiguration cycles for FG- and CG-fabrics) are obtained after place-and-route using Xilinx-FPGA-tools and ASIC-synthesis-flow for TSMC using the same technology node (90nm), respectively. Our mRTS is executing on a dedicated CG-EDPE. For comparisons, we conducted simulations for different state of the art approaches, using an entire H.264 video encoder as it is a complex application and exhibits various compute-intensive kernels with both control- and data-flow dominant processing.

5.2. Comparison to State-of-the-Art

In our comparisons in Fig. 8, the x-axis of the 2D graphs shows different combinations of CG-fabrics and PRCs of FG-fabrics. The first combination represents the execution of the whole application in RISC-mode. The bars in Fig. 8 show the execution time of different approaches. The x-axis comprises 4 bars per combination, where the first bar represents a RISPP-like approach [6], the second bar shows the Offline-optimal selection approach, the third bar shows Morpheus- and 4S-like approaches [8],[7], and the fourth bar presents the results for our novel mRTS approach. For fairness of comparison with state-of-the-art, the set of fabric combination, the application, and the application's input data are the same for each competitor.

Comparison with RISPP-like [6] approach:

The RISPP run time system [6] only deals with FG-fabrics. Therefore, its profit function is more tuned for longer reconfiguration time and computational properties of the FG-fabrics. However, for the purpose of a direct comparison we have extended their selection approach to use CG-fabrics, too. Fig. 8 shows the execution time of RISPP and our approach for different resource combinations. In the best cases, when multi-grained ISEs are used, our approach is up to 1.8x (on average 1.3x) faster than the extended RISPP. The main difference is the ability of our profit function to provide a good selection for the multi-grained ISEs and the use of *monoCG-Extension* by the Execution Control Unit (see Section 4.2). From Fig. 8, we can also see that RISPP and our approach perform similar when no CG-EDPEs are available, which demonstrates the applicability of our approach for RISPP-like [6] reconfigurable processors.

Comparison with Morpheus & 4S project like approach:

Approaches like Morpheus [8] and the 4S project [7] have already proposed solutions for multi-grained reconfigurable fabrics to achieve adaptivity for different application requirements. In both approaches, the decision of *which fabric is assigned to which task or application* is made at compile time. 4S [7] and Morpheus [7] do the selection at task level because they are loosely coupled architectures, i.e. the communication possibilities between the CG- and FG-fabric are limited. To model the loosely coupled approach of 4S [7] and Morpheus [8], we perform a combined offline selection for all functional blocks of an application in such a way that each kernel can be executed on either using CG-fabrics or FG-fabrics, therefore, no multi-grained ISE can be used within a functional block.

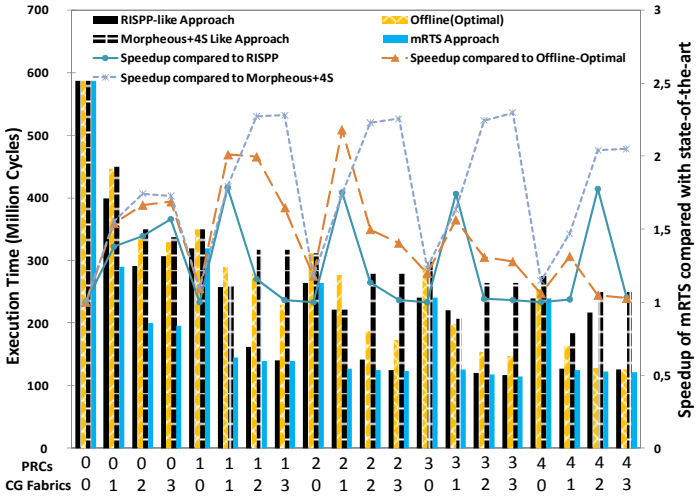

Fig. 8: Comparison with state of the art approaches

Fig. 8 shows an average speedup of 1.78x for our approach. In the best cases, our approach is up to 2.3x times faster. The main reason for the difference is that our mRTS performs selection for multi-grained ISEs at functional block level. The ability to use intermediate ISEs during the execution of functional block is another reason of our approach's edge over state-of-the-art here. It is noted that whenever the available resources are either FG or CG only, the mRTS results are similar to the results of Morpheus or 4S approach, as we can only choose either FG-ISE or CG-ISE , which brings our approach to the paradigm of loosely coupled architectures [7][8] and demonstrates its applicability for loosely coupled architectures as well.

Comparison with offline selection:
We compare against the offline (optimal) selection for tightly coupled multi-grained fabrics. From Fig. 8 we can observe that on average, our approach is 1.45x times faster than the optimal offline selection, and at best it is up to 2.2x times faster. Especially when at least one CG-fabric is available good speedups are noticeable, as then the benefits of *monoCG-Extension* come into effect. On the other hand, we can see diminishing profit as the total amount of resources increases because offline optimal can distribute the reconfigurable fabric judicially among kernels and run-time replacement gets less important.

5.3. In-depth Analysis of mRTS

In this section, we present a detailed analysis of our ISE selection algorithm as explained in Section 4.1.

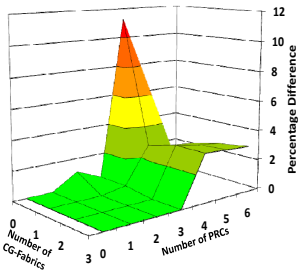

Fig. 9: ISE Selection algorithm vs. optimal algorithm

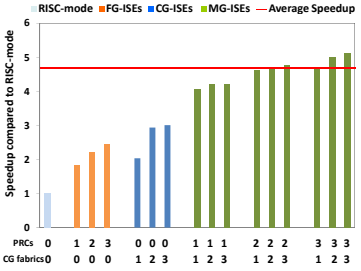

Fig. 10: Speedup compared to the RISC-mode

ISE Selection algorithm vs. optimal (run-time) algorithm:
Fig. 9 shows the percentage difference between the performance improvements offered by the optimal algorithm at run-time and the proposed heuristic based algorithm. We can see in Fig. 9 that in these experiments the ISE selection algorithm performs equally well as compared to the optimal algorithm. Only for certain resource combinations (in the cases when number of available PRCs are 4 or more than 4), the optimal algorithm performs noticeably better, as it can distribute the resources among the kernels of one functional block in a better way. On the contrary, the ISE selection algorithm tends to give more fabric to the kernel that contributes most towards the overall performance. Even for such unfavorable situations, the difference stays within around 3%, if at least one CG-fabric is available. The worst-case error, i.e. 11% difference in this experiment, occurs when only 4 PRCs are available. The reason is that during the execution of this application, the ISE Selection algorithm often assigns 3 out of 4 PRCs to one kernel, while the optimal algorithm shares them equally between the two most important kernels of the functional block.

General Speedup compared to the RISC-mode:
Fig. 10 shows the application speedup compared to RISC-mode execution at different resources combinations, sorted into groups of combinations, where only FG-, only CG- or a mix of both kinds of fabrics are available. The line shows the average speedup.

It is noticeable that when the application is executed using only PRCs, the achieved speedup ranges between 1.8x and 2.2x. However, when multi-grained reconfigurable fabrics are provided, then the mRTS achieves more than 5x application speedup as it starts employing multi-grained ISEs and *monoCG-Extension*. This is demonstrated by the fact that the combination of 1 PRC and 1 CG-fabrics performs significantly better than even 3 PRCs or 3 CG-fabrics only.

5.4. Implementation overhead of mRTS

Our mRTS comprises of two major computational blocks that are fundamental to the ISE selector, i.e. the profit function and the ISE-selection algorithm. The time to calculate the profit function depends on the number of intermediate ISEs, the number of intermediate ISEs that are covered by already available data paths, the number of different data paths, and number of data paths that must be reconfigured for ISE_i after ISE_{i-1} has already been reconfigured. Similarly, the execution time of ISE-selection algorithm depends on the number of kernels in the functional block and number of candidate ISEs.

In our experiments, the mRTS on average takes less than 3000 cycles to select an ISE for each kernel in a functional block, which is about 1.9% of an average execution time of a functional block. This overhead is negligible, especially as it only affects the first selection. As soon the first ISE is selected, the reconfiguration process is started and the mRTS then starts the ISE selection for the next kernel in parallel, i.e. its calculation time is hidden by the reconfiguration process.

6. Conclusion

Our novel run-time system for multi-grained reconfigurable processors provides a dynamic trade-off between performance and available area of multi-grained fabrics. It achieves this trade-off by selecting multi-grained instructions at run-time and steering them for enhanced performance. Our results demonstrate that our approach is equally beneficial for various multi-grained reconfigurable processors. Compared to state-of-the-art, our mRTS provides more than 2x performance improvement at the cost of an insignificant 1.9% performance overhead.

7. Acknowledgments

We thank German Research Foundation (DFG) for funding this project and Manuel Hammerich for his practical contribution in this project as Master's Thesis student.

8. References

[1] H. P. Huynh, T. Mitra, "Runtime Adaptive Extensible Embedded Processors - A Survey", *In Int'l Conf. on Embedded computer Systems, Architectures, Modeling and Simulation* (SAMOS), pp. 215-225, 2009.

[2] S. Vassiliadis, D. Soudris, "Fine- and Coarse-Grain Reconfigurable Computing", Springer, ISBN 978-1-4020-6504-0, 2007.

[3] F-J. Veredas et al., "Custom implementation of the coarse-grained reconfigurable ADRES architecture for multimedia purposes", *In Int'l Conf. on Field Programmable Logic and Applications* (FPL), pp. 106–111, 2005.

[4] PACT XPP Technologies, "Programming XPP-III Processors", http://www.pactxpp.com/main/download/XPP-III_programming_WP.

[5] R. Koenig et al., "KAHRISMA: A Novel Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array Architecture", *In Int'l Conf. on Design, Automation, and Test in Europe* (DATE), pp.819-824, 2010.

[6] L. Bauer et al., "Run-time System for an Extensible Embedded Processor with Dynamic Instruction Set", *In Int'l Conf. on Design, Automation, and Test in Europe* (DATE), pp.752-757, 2008.

[7] G. Smit et al., "Overview of the 4s project", *In Int'l Symp. on System-on-Chip* (SoC), pp. 70–73, 2005.

[8] F. Thoma et al., "Morpheus: Heterogeneous reconfigurable computing", *In Int'l Conf. on Field Programmable Logic and Applications* (FPL), pp. 409–414, 2007.

[9] G. Samit et al., "Lessons learned from designing the MONTIUM – a coarse-grained reconfigurable processing tile", In *Int'l Symp. on System-on-Chip* (SoC), pp. 29–32, 2004.

[10] P. Master, "Reconfigurable hardware and software architectural constructs for the enablement of resilient computing systems", *In Int'l Conf. on Application specific Systems, Architectures and Processors* (ASAP), pp. 50–55, 2006.

[11] C. Huang et al., "Dynamic Coprocessor Management for FPGA-Enhanced Compute Platforms", *Int'l Conf. on Compilers, Architectures, and Synthesis for Embedded System* (CASES), pp.71-78, 2008.

[12] L. Bauer et al., "A Self-Adaptive Extensible Embedded Processor", *Int'l Conf on Self-Adaptive and Self-Organizing Systems* (SASO), pp. 344-347, 2007.

[13] S. Vassiliadis et al., "The MOLEN polymorphic processor", *in IEEE Transactions on Computers (TC)*, vol. 53, no. 11, pp. 1363–1375, 2004.

[14] http://www.xilinx.com/products/v4q/lx.htm

[15] Z. Ye et al., "CHIMAERA: a high performance architecture with a tightly-coupled reconfigurable functional unit", *Int'l Symp. On Computer Architecture* (ISCA), pp. 225–235, 2000.

[16] A. Lodi et al., "A VLIW processor with reconfigurable instruction set for embedded applications", *Journal of Solid-State Circuits*, 38(11): pp.1876–1886, 2003.

[17] M. Shafique et al., "Optimizing the H.264/AVC Video Encoder Application Structure for Reconfigurable and Application-Specific Platforms", *In Journal of Signal Processing Systems* (JSPS), pp.183-210, 2010.

[18] T. Mitra et al., "An efficient framework for dynamic reconfiguration of instruction-set customization", *Int'l Conf. on Compilers, Architectures, and Synthesis for Embedded System* (CASES), pp.135-144, 2007.

[19] L. Pozzi et al, "Introduction of Local Memory Elements in Instruction Set Extensions", *In Design Automation Conference* (DAC), pp.729-734, 2004.