Runtime Task Mapping Based on Hardware Configuration Reuse

Kamana Sigdel, Carlo Galuzzi, Koen Bertels
Computer Engineering Group, TU Delft, The Netherlands
Email: {K.Sigdel, C.Galuzzi, K.L.M.Bertels}@tudelft.nl

Mark Thompson, Andy D. Pimentel
Systems Architecture Group, UvA, The Netherlands
Email: {M.Thompson, A.D.Pimentel}@uva.nl

***Abstract*—In this paper, we propose a new heuristic for runtime task mapping of application(s) onto reconfigurable architectures. The heuristic is based on hardware configuration reuse, which tries to avoid the reconfiguration overhead of few selected tasks, by reusing the hardware configurations already available in the reconfigurable hardware. We evaluate our heuristic by performing a mapping of an extended Motion-JPEG application onto a reconfigurable architecture. A large variety of experiments have been conducted on the proposed algorithm for the same reconfigurable architecture model with different FPGA sizes. The obtained result shows up to 45% performance gain by reusing the hardware configurations as suggested by the proposed heuristic, compared to well-known approaches from the state-of-the-art, which do not take into consideration the hardware configuration reuse.**

I. Introduction

More than a decade long, we have witnessed an increasing popularity of reconfigurable architectures due to their capability of providing high execution performance for an application by tuning the architecture towards the specific requirements of the application. These architectures are typically formed with a combination of a General Purpose Processor (GPP) and a reconfigurable hardware such as an FPGA. Reconfigurable architectures are subject to numerous constraints and design objectives, such as cost, resource constraints, power consumption, timing constraints, and dependability. The design of heterogenous reconfigurable systems imposes several challenges to system designers, such as hardware-software partitioning, Design Space Exploration (DSE), task mapping and scheduling.

The applications that will run in parallel and their respective user requirements are not known at the design time. As a result, the design time evaluation alone is not enough for any kind of architectural exploration. Due to changing runtime conditions with respect to e.g. user requirements or having multiple simultaneously executing applications competing for platform resources, it is necessary to perform runtime evaluation for better accuracy. Runtime evaluation enables a system to be more efficient in terms of various design constraints, such as performance, chip area and power consumption. In case of partially dynamic reconfigurable architectures which are subject to changes at runtime, the design time task partitioning, mapping and exploration are inadequate and cannot address the changing runtime conditions.

[0]This research is partially supported by Artemisia iFEST project (grant 100203), Artemisia SMECY (grant 100230), FP7 Reflect (grant 248976).

Towards this goal, in this paper, we present a new heuristic for runtime task mapping onto reconfigurable architectures. The heuristic is based on hardware configurations reuse. By reusing hardware configurations already available on the hardware, multiple reconfigurations can be avoided. As a consequence, the total execution time of the application can be significantly reduced. The paper presents a task mapping strategy that exploits the hardware configuration reuse to reduce the reconfiguration overhead. The main contributions of this paper are the following:

- the presentation of a new heuristic for runtime task mapping of application(s) onto reconfigurable architectures. The proposed heuristic exploits hardware configuration reuse to avoid multiple reconfigurations in order to increase the performance gain.
- the evaluation of the proposed heuristic for a given reconfigurable architecture. This evaluation is done by considering for the same reconfigurable architecture model different number of resources.
- the comparison of the proposed heuristic with the well known methodology from the current state-of-the-art, under different resource conditions. Obtained results suggest up to 45% increase in the performance by reusing hardware configurations.

The rest of the paper is organized as follows: Section II provides the related research. Section III discusses the proposed task mapping heuristic based on the hardware configurations reuse, while Section IV presents a case study using the proposed heuristic. In Section V, a detailed analysis and evaluation of the overall performance is presented and a comparison with existing algorithms from the state-of-the-art is presented. Finally, section VI concludes the paper.

II. Related Work

Task mapping can be performed in two mutual non-exclusive ways: at design-time and at runtime. At design time task mapping is performed under static conditions without taking into account any change of the system. Such examples are: dynamic programming [7], Integer Linear Programming (ILP) [6], simulated annealing [9], tabu search [8], genetic algorithm [3] and ant colony optimization [16]. When task mapping is performed at runtime, changes in the system are considered, and the mapping is performed accordingly.

In [14], authors present a simple approach for runtime task mapping, which evaluates the most frequently executed tasks

at runtime, and maps them onto a reconfigurable hardware. This work, however, has a focus on the lower level, and it targets only loop kernels. A similar approach for high-level runtime task mapping for multiprocessor SoC containing fine-grain reconfigurable hardware tiles is presented in [10]. [4] presents a runtime mapping based on a cumulative benefit heuristic which is based on a commonly used accumulation approach.

In the same way, the study in [1] presents runtime resource allocation and scheduling heuristic for the multi-threaded environment based on the status of the reconfigurable system. Correspondingly, [2] presents a dynamic method for runtime task mapping, task scheduling and task allocation for reconfigurable architectures. Authors in [12] present a runtime optimization which targets the speedup of applications running onto a reconfigurable platform. In this context [12], an online adaptive decision algorithm to determine whether a task should be executed as hardware or software has been proposed. Likewise, the approach of [13] consists of a mapper which determines a mapping of application(s) to a heterogeneous reconfigurable tiled SoC architecture by using a library at runtime.

The above mentioned techniques for runtime task mapping perform mapping based on various functional and non-functional parameters. However, none of these techniques exploit the hardware configuration reuse. In order to improve the performance gain by avoiding the reconfiguration overhead, in our approach, we perform runtime task mapping based on hardware configurations reuse, which avoids multiple configurations.

III. Task Mapping Heuristic

Reconfiguration overhead has always been a serious concern for reconfigurable architectures, as it can drastically limit the performance of reconfigurable systems. In an ideal case, a task is configured on the reconfigurable hardware only once, and it is reused to accelerate the application in all other cases. The reuse of the hardware configuration avoids multiple configurations, and as a result, reconfiguration overhead can be significantly reduced. In case of application domains, such as streaming applications and networking applications, where certain tasks are executed in a periodic manner or per frame basis, the hardware configuration reuse can be easily exploited.

The proposed runtime task mapping heuristic tries to avoid the reconfiguration overhead by reusing the hardware configurations which are already available on the FPGA. The basic idea of the heuristic is to avoid multiple reconfigurations such that the execution time is reduced. Certain tasks mapped onto the FPGA are preserved in the FPGA after their execution. These hardware configurations can be reused when the task is re-executed. Reusing hardware configurations multiple times avoids reconfiguration overhead, thus, performance can be considerably improved. It is not possible for all tasks to preserve their configurations in the hardware. For this reason, the heuristic tries to preserve hardware configurations for selected tasks. For example, tasks which have higher reconfiguration delay and occur very frequently in the system have priority on being preserved onto the FPGA.

We define three states for a task: a *waiting* state, a *mapped* state, and a *running* state. A task is in the waiting state if it waits to be mapped. A task is in the mapped state if it is already configured on the FPGA but it is not being executed, however, it may be re-executed later. A task is in the running state when the task is actually processing data. It should be noted that when a task is in the mapped state, its hardware configuration is saved in the FPGA, thus, when the task needed to be re-executed it can directly start processing without reconfiguration.

Algorithm 1 Pseudocode for the proposed task mapping heuristic based on hardware configuration reuse.

```
1:  {Task already mapped on FPGA, do not configure.}
2:  if T_i == MAPPED then
3:      T_i.state ← RUNNING;
4:  else
5:      if area ≥ T_i.area then
6:          if SpeedUp(T_i) > 1 then
7:              {Task not mapped on FPGA, configure it.}
8:              configure(T_i);
9:              T_i.state ← RUNNING;
10:         end if
11:     else
12:         for All tasks T_j on the FPGA do
13:             if SpeedUp(T_j) < SpeedUp(T_i) then
14:                 candidateSet = T_j
15:             end if
16:         end for
17:         while area ≤ T_i.area do
18:             Select T_k ∈ candidateSet with lowest RER
19:             removeSet = T_k
20:             area = area + T_k.area;
21:         end while
22:         if T_i.area ≤ area then
23:             for All task T_m ∈ removeSet do
24:                 T_m.state = WAITING;
25:             end for
26:             {Task not mapped on FPGA, configure it.}
27:             configure(T_i);
28:             T_i.state ← RUNNING;
29:         end if
30:     end if
31: end if
```

Algorithm 1 presents the pseudocode describing the functionality of the proposed heuristic for runtime mapping of a task T_i. If resources are available in the FPGA, T_i is mapped onto the FPGA only if there is a performance gain (line 5 to 10 in Algorithm 1). The performance gain in this case is measured in terms of speedup. The speedup for each task is measured at runtime by using the following equation:

$$\text{Speedup} = \frac{\text{TSW} \cdot \left(\#\text{HWEx} + \#\text{SWEx}\right)}{\#\text{SWEx} \cdot \text{T\_SW} + \#\text{HWEx} \cdot \text{T\_HW} + \#\text{Recon} \cdot \text{T\_Recon}} \quad (1)$$

where #HWEx, #SWEx and #Recon are the number of HW executions, SW executions and reconfigurations of a task respectively. Similarly, T_HW, T_SW and T_Recon are the corresponding hardware, software and reconfigurable latencies. The heuristic maintains a profiling count of HW

executions, SW executions and reconfigurations for all tasks. Each time a task is executed, these counts for that task are updated. For instance, if a task is executed with the GPP, its SW count is incremented and if the task is executed in the FPGA, its HW count is incremented. Similarly, the reconfiguration count of a task is incremented when a task is (re)configured. These count values for each task are accumulated from all the previous executions. As a result, they reflect the execution history of a task. The speedup calculated with these count values indicates the precise speedup of a task up to that point of execution.

If the available resources are not enough in the FPGA, a set of tasks from the FPGA are swapped to accommodate T_i in the FPGA. The task swapping, in this case, is done based on two factors: a) speedup and b) Reconfiguration-to-Execution Ratio (RER). At the first step, a candidate set of tasks from the FPGA, which are beneficial than the current task in terms of speedup, is selected (line 12 to 16 in Algorithm 1). The speedup in this case is calculated by using the Equation 1. At the second step, the candidate set is examined for its RER ratio, such that tasks with the lowest RER values are swapped first (line 17 to 13 in Algorithm 1). RER for each task is computed as follows:

$$\text{RER} = \frac{\text{T\_Recon}}{\text{T\_HW}} \cdot \text{Exec\_Freq} \tag{2}$$

where Exec_Freq is the average execution frequency of the task in its past history. The execution frequency of a task can be simply computed from the execution profile of each task with respect to the total execution count of that application. The task with a high RER value indicates that the task has high reconfiguration per execution delay and it has executed very frequently in the system in its history, making it a probable candidate for future execution. The heuristic makes a very careful selection while removing tasks from the FPGA. By preserving tasks with higher RER value as long as possible in the FPGA, we try to avoid the reconfiguration of the frequently executed tasks. We would like to stress the fact that the speedup and RER are not constant factors. These values are constantly updated based on the execution profile of the task at runtime. Hence, mapping tasks onto the FPGA based on such value precisely represents the system behavior at that particular instance of time. Note that the proposed heuristic is generic and is not restricted to one type of resources or one type of architecture. In order to consider multiple resources (such as memory, DSP slices) at runtime mapping decision for different architectural components, the parameters in the heuristic can be easily customized, hence making it a flexible and platform independent approach.

IV. Case Study

We use the rSesame framework [11] as a simulation platform for evaluating the proposed runtime task mapping heuristic. The rSesame is a generic modeling and simulation framework which can explore and evaluate reconfigurable systems at runtime. The framework can be efficiently employed to perform DSE with respect to hardware-software partitioning, task mapping, task allocation and scheduling. For the application modeling, the framework uses Kahn Process Network (KPN) [5] at the granularity of coarse-grain tasks. With the rSesame framework, an application task can either be modeled as a *hardware* (HW), *software* (SW) or as a *pageable* task. A HW(SW) task is always mapped onto the reconfigurable hardware component(microprocessor), while a *pageable* task can be mapped on either of these resources. Task assignment to the SW, HW and pageable categories is performed at design time. At runtime, these tasks are mapped onto their corresponding resources based on time, resources and conditions of the system. For this case study, we constructed a model using the rSesame framework for mapping an extended MJPEG application onto the Molen reconfigurable architecture [15] (see Section IV-A). The proposed heuristic is used as a strategy to perform runtime mapping decision in the model.

A. Application and Architecture Model

We extend a Motion-JPEG (MJPEG) encoder application to use it as an application model for this case study. The corresponding KPN graph is shown in Figure 1. The frames are divided into blocks and each task performs a different function on each block as it is passed from task to task. MJPEG operates on these blocks (partially) in parallel. In order to evaluate the runtime task mapping with the proposed heuristic, a random number (0 to 3) of applications (APP1 to APP3) are injected in each frame of the MJPEG application to create a dynamic behavior. These applications are considered as sporadic ones which appear in the system randomly and compete with MJPEG for the resources.

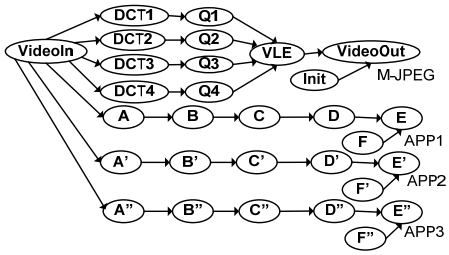

Figure 1. The Motion-JPEG (MJPEG) application model considered for the case study. The MJPEG application is extended by injecting sporadic applications in each frame.

The architecture model considered in this case study is the Molen reconfigurable architecture. The Molen [15] is an established norm for the polymorphic processor paradigm incorporating a GPP and a Reconfigurable Processor (RP), such as an FPGA. The RP is used to accelerate code fragments from the applications in a processor/co-processor paradigm. The RP consists of one or more Custom Computing Units (CCUs), each representing a hardware implementation of a task. The tasks to be accelerated on the RP are mapped onto these units. Application tasks can be executed either on the

Task	DCT1	DCT3	DCT2	F'	DCT4	Q1	Q2	Q3	Q4	C	B	F	Init	VideoIn	A
CCU	CCU0	CCU1	CCU2	CCU3	CCU4	CCU5	CCU6	CCU7	CCU8	CCU9	CCU10	CCU11	CCU12	CCU13	CCU14
Task	VLE	D	Vout	E	E'	D'	B'	A'	C'	E"	D"	C"	B"	A"	F"
CCU	CCU15	CCU16	CCU17	CCU18	CCU19	CCU20	CCU21	CCU22	CCU23	CCU24	CCU25	CCU26	CCU27	CCU28	CCU29

Table I
MAPPING OF TASKS ONTO CCUS

GPP, on the RP, or on both. Tasks run on the GPP as regular (compiled) microprocessor code or on the RP as a hardware IP core.

B. Experimental Setup

We instantiated a model using the rSesame framework for the Molen reconfigurable architecture running the application model depicted in Figure 1. The model consists of 30 CCUs, thus each task is mapped onto one CCU. The mapping of the tasks onto the CCUs is given in Table I, where the first row lists all the tasks and the second row lists their corresponding CCUs. For this experiment, we consider all tasks as pageable ones; as a result, for all the tasks, mapping decisions are made at runtime. The model allows dynamic partial reconfiguration. In case all CCUs do not fit at the same time in the FPGA, they can be executed after reconfiguration at runtime.

Hardware	Area (Slices)
XC4VFX20	8544
XC4VFX40	18624
XC4VFX60	25280
XC4VFX100	42176
XC4VFX140	63168

Table II
AREA AVAILABLE FOR DIFFERENT FPGAS FOR THE XILINX VIRTEX4 FX FAMILY [17].

In this case study, we studied the performance trade-off and the task mapping/re-mapping behavior of an extended MJPEG application by considering, for the same architecture model, different hardware resources. We considered, for this case study, five FPGAs from the Xilinx Virtex-4 FX family as given in Table II. The considered FPGAs have different available area (slices). As a result, they are used to evaluate the runtime task mapping under different resource conditions. The computational latencies for the GPP and the CCUs for the application model are initialized using estimated values. We also used estimated area occupancy for each CCU. Note that the CCU size corresponds to the size of the task mapped onto it. As a result, with different task size in the considered MJPEG application, the size of the CCU also differs. Based on the reconfiguration delay of each FPGA, we computed the reconfiguration delay of each CCU using the following equation:

$$\text{T\_Recon} = \frac{\text{CCU\_slices}}{\text{FPGA\_slices}} \cdot \frac{\text{FPGA\_bitstream}}{\text{ICAP\_bandwidth}} \quad (3)$$

where CCU_slices is the total number of area slices a CCU requires, FPGA_slices is the total slices available on particular FPGA. FPGA_bitstream is the bitstream size in MBs of that FPGA and ICAP_bandwidth is the ICAP configuration

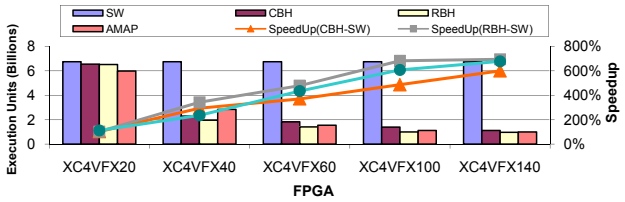

Figure 2. Comparison of the RBH against CBH and AMAP under different FPGAs conditions. The RBH performs better than AMAP and CBH under all FPGA conditions.

speed. We assume that the Processor Local Bus (PLB) is 4 bytes wide and that the ICAP functions at 100 MHz, thus its configuration speed is considered at 400 MB/sec [18]. As a final remark, we assume that there is no delay associated with the runtime mapping such as task migration, context switching, delay associated with the RMM and the RM.

V. RESULTS AND EVALUATION

We conducted a large variety of experiments on the proposed heuristic for the Molen reconfigurable architecture with various FPGAs of different sizes. We also compared the results of the proposed runtime task mapping heuristic with two other task mapping heuristics taken from the current state-of-the-art. As Much As Possible (AMAP) is a very simple heuristic which tries to map tasks based on area availability. It is often used for runtime resource management in various systems especially in OS level. The Cumulative Benefit Heuristics (CBH) [4] uses the cumulative benefit as a factor for runtime task mapping. We evaluated and compared these task mapping heuristics based on two parameters: *execution time* and the *reusability efficiency*.

A. Execution Time

Figure 2 depicts the results of running different task mapping heuristics for mapping an extended MJPEG application model onto the Molen architecture with various FPGAs of different sizes. The primary *y-axis* (left) in the graph represents the application execution time measured for each heuristic - CBH, AMAP and the proposed hardware configuration Reuse Based Heuristic (RBH). The software only execution is measured when all the tasks are mapped onto the GPP. The secondary *y-axis* (right) represents the application speedup compared to software only execution. The *x-axis* lists the different FPGAs which are ranked (from left to right) based on their sizes - XC4VFX20 has the smallest number of area slices and XC4VFX140 has the largest number of area slices (see Table II).

A first trivial observation from Figure 2 is that the application performance is proportional to the FPGA size:

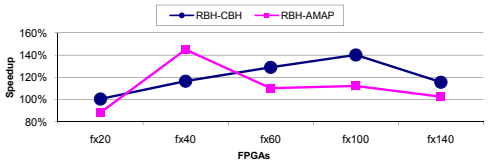

Figure 3. Performance improvement of the proposed heuristic (RBH) as compared to CBH and AMAP. The RBH performs better than AMAP and CBH under all FPGA conditions.

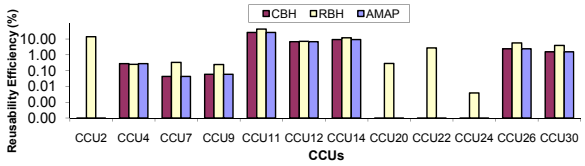

Figure 4. RE_{task} of CCUs mapped onto XC4VFX100 based on the proposed heuristics. Tasks other than shown in the figure are either mapped onto the software or have RE value zero.

the bigger the available area in the FPGA, the higher the application performance. In case of XC4VFX20, there is no significant performance gain by using any heuristic as compared to the software only execution due to the limited area. Nevertheless, there is a notable improvement in the performance with other FPGAs. Furthermore, comparing the results of the RBH for different FPGAs in the figure, we notice that there is approximately 69% improvement in the application performance in case of XC4VFX40 compared to XC4VFX20, while there is 54% increase in the area in the former (see Table II). Nevertheless, comparing XC4VFX100 with XC4VFX140, there is only 1.8% improvement in the application performance with 33% increase in the area. The performance increase is bounded by the parallelism in the application. The use of more resources doesn't give better performance in all cases. The same conclusion applies for the other two heuristics as well.

Another observation that can be made from Figure 2 is that the RBH outperforms the other two heuristics in terms of performance under all resource conditions. The performance improvement of the RBH compared to CBH and AMAP under different resource conditions is depicted in Figure 3. The best improvement of RBH compared to CBH is obtained in case of XC4VFX100 which is approximately 40%. Moreover, when comparing RBH against AMAP, we can see that the best improvement is obtained when using XC4VFX40, which is approximately 45%. Nonetheless, the performance improvement of the RBH compared to AMAP shows an irregular behavior. The RBH performs 10% lower compared to AMAP in XC4VFX20. However, the improvement significantly increases in XC4VFX40. In XC4VFX60, the improvement suddenly decreases to 10% and stays identical in XC4VFX100 and XC4VFX140 (see Figure 3). AMAP performs task mapping based on the area availability in an ad-hoc manner However, the RBH performs a selective task mapping based on the task speedup and the hardware configuration reuse. When the area is limited as in case of XC4VFX20, configuration reuse cannot be exploited with the RBH as many configurations cannot be saved. As a result, AMAP performs better than the RBH. With the area increase, many configurations can be saved in the FPGA, and the RBH performs better.

B. Reusability Efficiency

A CCU execution on FPGA has two phases: the *configuration phase*, where the bitstream of tasks is loaded onto the FPGA, and the *running phase*, where the CCU is actually processing data. In an ideal case, a CCU is configured on the FPGA only once and executed in all other cases. We define, the Reusability Efficiency (RE) as the ratio of the reconfiguration overhead that is saved due to the hardware configuration reuse to the total execution time of any task. The RE of a CCU can be defined as follows:

$$RE_{task}\ (\%) = \frac{(\#HWEx - \#Recon) \cdot T_{Recon}}{\#HWEx \cdot T_{HW} + \#Recon \cdot T_{Recon} + \#SWEx \cdot T_{SW}} \cdot 100 \quad (4)$$

where #HWEx, #SWEx and #Recon are the number of HW executions, SW executions and reconfigurations of a CCU respectively. Similarly, T_{HW}, T_{SW} and T_{Recon} are the corresponding hardware, software and reconfigurable latencies.

RE indicates the percentage of the total time saved by a CCU when multiple reconfigurations are avoided, or in other words, a CCU is reused. In the above expression, the numerator represents the time that is saved by a task when the mapping of a CCU is reused and the denominator gives the total execution time. The total RE for an application can be calculated as the summation of the numerator in Equation 4 for all N tasks divided by the total execution time for the whole application as follows:

$$RE_{app}\ (\%) = \frac{\sum_{i=1}^{N} (\#HWEx - \#Recon) \cdot T_{Recon}}{\text{Total Execution Time}} \cdot 100 \quad (5)$$

Note that the RE_{app} calculated in this way for the whole application can only be given here as an upper bound, since the execution of tasks on the reconfigurable hardware can be performed in parallel. A higher RE can obtain a higher speedup. To study this relation, we use the RE_{task} as an evaluation parameter to study the behavior of each CCU. We also use RE_{app} to study this relation at the application level.

Figure 4 depicts the RE_{task} recorded using Equation 4 for a selection of CCUs for the experiment using XC4VFX100. The RE_{task} is zero when the task is always mapped onto the GPP (#HWEx=0 in Equation 4) or when it has to be reconfigured every time it is mapped onto the FPGA (#HWEx = #Recon in Equation 4). For comparison purposes, Figure 4 only shows the CCUs that are mapped onto the FPGA according to the RBH. AMAP and CBH have different set of CCUs mapped onto the FPGA. In case of AMAP, all CCUs are mapped onto the FPGA: the ones not shown in Figure 4 have RE_{task} value zero because they are always configured.

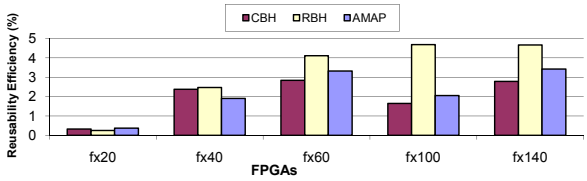

Figure 5. RE_{app} computed for different heuristics for different FPGAs. The proposed heuristic (RBH) has better better RE compare to CBH and AMAP.

The same is true for CBH for the following CCUs: CCU2, CCU6, CCU8, CCU10 and CCU24.

As it can be clearly seen from Figure 4, the RE_{task} of the RBH is better than the RE_{task} of AMAP and CBH for all CCUs that are shown in the figure. In case of AMAP and CBH, several CCUs are configured every time they are executed, and they have RE value zero, such as CCU2, CCU20, CCU22 and CCU24. Note that, AMAP and CBH do not perform task mapping based on the hardware configuration reuse. The reuse obtained in case of AMAP and CBH is due to the resource management of the rSesame framework. The resource management in the rSesame framework is implemented in such a way that it avoids the reconfiguration of CCUs every time they are executed on the FPGA. The RBH actually reuses more hardware configurations on top of the default resource management provided by the framework. The reusability obtained in case of CBH and AMAP is the default reusability of the framework. This can also explain the same value for RE_{task} in case of AMAP and CBH. While evaluating these heuristics stand-alone or with other frameworks, the RE value for all CCUs in case of AMAP and CBH will be zero.

Similarly, Figure 5 depicts the total RE_{app} recorded using Equation 5 for different heuristics under different resource conditions. As it can be inferred from the figure, the reusability increases when using larger FPGAs. The RBH has better RE_{app} value than AMAP and CBH in all resource conditions except XC4VFX20 where all the heuristics have approximately same value for RE_{app}. In case of AMAP and CBH, all CCUs have the same value for RE_{task}. This implies, the time saved because of avoiding reconfiguration is same for CBH and AMAP. As a result, the total RE_{app} value in case of AMAP and CBH depends on the corresponding total execution time (see Figure 5).

Based on the above evaluation, we can summarize the following observations:

- in case of limited resource conditions (small FPGAs), many hardware configurations cannot be saved, thus, the configuration reuse cannot be fully exploited. In such cases, AMAP performs better than RBH.
- the configuration reuse can be well exploited in case of sufficient resource conditions (medium to large FPGAs). In such cases, many hardware configurations can be preserved in the FPGA, thus, the reuse of the hardware configuration is better, and the RBH provides better application performance than AMAP and CBH.
- in case of abundant resource conditions (very large FPGAs), the performance saturates due to application constraints. Under such scenarios, all the heuristics have similar performance.

VI. Conclusions

In this paper, we proposed a new heuristic for the runtime task mapping of application(s) onto reconfigurable architectures. The heuristic is based on hardware configuration reuse, which tries to avoid reconfiguration overhead of some selected tasks by efficiently reusing the hardware configurations already available in the FPGA. We evaluated the heuristic for a reconfigurable architecture with different FPGA resources using an extended MJPEG application. We also compared the heuristic with two heuristics from the state-of-the-art. The results suggest that the proposed heuristic outperforms existing heuristics for all the FPGAs considered with an increase in performance up to 45%.

References

[1] W. Fu et al., "An execution environment for reconfigurable computing," in *Proc. of the FCCM'05*, 2005.
[2] F. Ghaffari et al., "Dynamic and on-line design space exploration for reconfigurable architectures," *Trans. on HiPEAC*, pp. 179–193, 2007.
[3] C. Haubelt et al., "A system-level approach to hardware reconfigurable systems," in *Proc. of the ASP-DAC'05*, 2005.
[4] C. Huang et al., "Dynamic coprocessor management for fpga-enhanced compute platforms," in *Proc. of the CASES'08*, 2008.
[5] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. of the IFIP Congress*, 1974.
[6] M. Kaul et al., "Design-space exploration for block-processing based temporal partitioning of run-time reconfigurable systems," *Journal of VLSI Signal Processing Systems*, vol. 24, pp. 181–209, 2000.
[7] P. V. Knudsen et al., "Pace: A dynamic programming algorithm for hardware/software partitioning," in *CODES*'96, 1996.
[8] L. Lanying et al., "Software-hardware partitioning strategy using hybrid genetic and tabu search," in *Proc. of the CSSE'08*, 2008.
[9] B. Miramond et al., "Design space exploration for dynamically reconfigurable architectures," in *Proc. of the DATE'05*, 2005.
[10] V. Nollet et al., "Run-time management of a mpsoc containing fpga fabric tiles," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 16, no. 1, pp. 24–33, 2008.
[11] K. Sigdel et al., "A generic system-level runtime simulation framework for reconfigurable architectures," in *Proc. the FPT'09*, December 2009.
[12] V. M. Sima et al., "Runtime decision of hardware or software execution on a heterogeneous reconfigurable platform," in *Proc. of the IPDPS'09*, 2009.
[13] L. T. Smit et al.,"Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture," in *Proc. of the FPT'04*, 2004.
[14] G. Still et al., "Dynamic hardware/software partitioning: A first approach," in *Proc. of the DAC'03*, 2003.
[15] S. Vassiliadis et al., "The molen polymorphic processor," *IEEE Transactions on Computers*, pp. 1363– 1375, November 2004.
[16] G. Wang et al., "Application partitioning on programmable platforms using the ant colony optimization," *Journal of Embedded Computing*, vol. 2, no. 1, pp. 119–136, 2006.
[17] Xilinx Corporation, "Virtex-4 family overview (V3.0)."
[18] ——, "LogiCORE IP XPS HWICAP (v5.00a)," July 2010.