Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 234230, 17 pages
doi:10.1155/2012/234230

Research Article

Evaluation of Runtime Task Mapping Using the rSesame Framework

Kamana Sigdel,[1] Carlo Galuzzi,[1] Koen Bertels,[1] Mark Thompson,[2] and Andy D. Pimentel[2]

[1] *Computer Engineering Group, Technical University of Delft, Mekelweg 4, 2628 CD, Delft, The Netherlands*
[2] *Computer Systems Architecture Group, University of Amsterdam, Science Park 904, 1098 XH, Amsterdam, The Netherlands*

Correspondence should be addressed to Kamana Sigdel, k.sigdel@tudelft.nl

Received 8 May 2011; Revised 20 December 2011; Accepted 30 December 2011

Academic Editor: Viktor K. Prasanna

Performing runtime evaluation together with design time exploration enables a system to be more efficient in terms of various design constraints, such as performance, chip area, and power consumption. rSesame is a generic modeling and simulation framework, which can explore and evaluate reconfigurable systems at both design time and runtime. In this paper, we use the rSesame framework to perform a thorough evaluation (at design time and at runtime) of various task mapping heuristics from the state of the art. An extended Motion-JPEG (MJPEG) application is mapped, using the different heuristics, on a reconfigurable architecture, where different Field Programmable Gate Array (FPGA) resources and various nonfunctional design parameters, such as the execution time, the number of reconfigurations, the area usage, reusability efficiency, and other parameters, are taken into consideration. The experimental results suggest that such an extensive evaluation can provide a useful insight both into the characteristics of the reconfigurable architecture and on the efficiency of the task mapping.

1. Introduction

In recent years, reconfigurable architectures [1, 2] have received an increasing attention due to their adaptability and short time to market. Reconfigurable architectures use reconfigurable hardware, such as Field Programmable Gate Array (FPGA) [3, 4] or other programmable hardware (e.g., Complex Programmable Logic Device (CPLD) [5], reconfigurable Datapath Array (rDPA) [6]). These hardware resources are frequently coupled with a core processor, typically a General Purpose Processor (GPP), which is responsible for controlling the reconfigurable hardware. Part of the application's tasks is executed on the GPP, while the rest of the tasks are executed on the hardware. In general, the hardware implementation of an application is more efficient in terms of performance than a software implementation. As a result, reconfigurable architectures enhance the whole application through an implementation of selected application kernels onto the reconfigurable hardware, while preserving the flexibility of the software execution with the GPP at the same time [7, 8]. The design of such architectures is subject to numerous design constraints and requirements, such as performance, chip area, power consumption, and memory. As a consequence, the design of heterogeneous reconfigurable systems imposes several challenges to system designers such as hardware-software partitioning, Design Space Exploration (DSE), task mapping, and task scheduling.

Reconfigurable systems can evolve under diverse conditions due to the changes imposed either by the architecture, by the applications, or by the environment. A reconfigurable architecture can evolve under different conditions, for instance, processing elements shutdown in order to save power, or additional processing elements are added in order to meet the execution deadline. The application behavior can change, for example, due to the dynamic nature of the application-application load changes due to the arrival of sporadic tasks. In such systems, the design process becomes more sophisticated as all design decisions have to be optimized in terms of runtime behaviors and values. Due to changing runtime conditions with respect to, for example, user requirements or having multiple simultaneously executing applications competing for platform resources, design time evaluation alone is not enough for any kind of architectural exploration. Especially in the case of partially

dynamic reconfigurable architectures that are subject to changes at the runtime, design time exploration and task mapping are inadequate and cannot address the changing runtime conditions. Performing runtime evaluation enables a system to be more efficient in terms of various design constraints, such as performance, chip area, and power consumption. The evaluation carried at runtime can be more precise and can evaluate the system more accurately than at design time. Nevertheless, such evaluations are typically hard to obtain due to the enormous size and complexity of the search space generated by runtime parameters and values.

In order to benefit from both design time and runtime evaluations, we developed a modeling and simulation framework, called rSesame [9], which allows the exploration and the evaluation of reconfigurable systems at both design time and runtime. With the rSesame framework, designers can instantiate a model that can explore and evaluate any kind of reconfigurable architecture running any set of streaming applications from the multimedia domain. The instantiated model can be used to evaluate and compare various characteristics of reconfigurable architectures, hardware-software partitioning algorithms, and task mapping heuristics. In [10], we used the rSesame framework to perform runtime exploration of a reconfigurable architecture. In [11], we proposed a new task mapping heuristic for runtime task mapping onto reconfigurable architectures based on hardware configurations reuse. In this paper, we present an extension of the work presented in [10, 11]. In particular, we present an extensive evaluation and comparison of various task mapping heuristics from the state of the art (including the heuristics we presented in [11]) both at design time and at runtime using the rSesame framework. More specifically, the main contributions of this paper are the following:

(i) a detailed case study using the rSesame framework for mapping different runtime task mapping heuristics from the state of the art (including the runtime task mapping heuristics in [11]). For this case study, we use an extended MJPEG application and a reconfigurable architecture;

(ii) an extensive evaluation of the different heuristics for a given reconfigurable architecture. This evaluation is performed by considering different number of FPGA resources for the same reconfigurable architecture model;

(iii) a thorough comparison of the aforementioned heuristics under different resource conditions using various nonfunctional design parameters, such as execution time, number of reconfiguration, area usage, and reusability efficiency. The comparison is done both at design time as well as at runtime.

The rest of the paper is organized as follows. Section 2 provides the related research. Section 3 discusses the rSesame framework, which is used as a simulation platform for evaluating task mapping at runtime, while Section 4 presents a detailed case study using the different heuristics. In Section 5, a detailed analysis and evaluation of the task mapping at runtime using the rSesame framework is presented. Finally, Section 6 concludes the paper.

2. Related Work

Task mapping can be performed in two mutual nonexclusive ways: at design time and at runtime. The task mapping performed at the design time can generally be faster, but it may be less accurate as the runtime behavior of a system is mostly captured by using offline (static) estimations and predictions. Examples of techniques for task mapping at design time are dynamic programming [12], Integer Linear Programming (ILP) [13], simulated annealing [14, 15], tabu search [16], genetic algorithm [17, 18], and ant colony optimization [19].

In another way of performing task mapping, the reconfigurable system is evaluated for any changes in the runtime conditions and the task mapping is performed at runtime based on those conditions. Under such scenario, the changes in the system are considered and the task mapping is performed accordingly. In [20], the authors present a simple approach for runtime task mapping in which a mapping module evaluates the most frequently executed tasks at runtime and maps them onto a reconfigurable hardware component. However, this work [20] focuses on the lower level and it targets only loop kernels. A similar approach for high-level runtime task mapping is presented in [21] for multiprocessor System on Chip (SoC) containing fine-grain reconfigurable hardware tiles. This approach details a generic runtime resource assignment heuristic that performs fast and efficient task assignment. In [22], the authors define the dynamic coprocessor management problem for processors with an FPGA and provide a mapping to an online optimization based on the cumulative benefit heuristic, which is inspired by a commonly used accumulation approach in online algorithm work.

In the same way, the study in [23] presents runtime resource allocation and scheduling heuristic for the multithreaded environment, which is based on the status of the reconfigurable system. Correspondingly, [24] presents a dynamic method for runtime task mapping, task scheduling and task allocation for reconfigurable architectures. The proposed method consists of dynamically adapting an architecture to the processing requirement. Likewise, the authors in [25, 26] present an online resource management for heterogeneous multiprocessor SoC systems, and the authors in [27] present a runtime mapping of applications onto a heterogeneous reconfigurable tiled SoC architecture. The approach presented in [27] proposes an iterative hierarchical approach for runtime mapping of applications to a heterogeneous SoC. The approach presented in [28] consists of a mapper, which determines a mapping of application(s) to an architecture, using a library at runtime. The approach proposed by authors in [29] performs mapping of streaming applications, with real-time requirements, onto a reconfigurable MPSoC architecture. In the same way, Faruque et al. [30] present a scheme for runtime-agent-based distributed application mapping for on-chip communication for adaptive NoC-based heterogeneous multiprocessor systems.

There are few attempts which combine design time exploration together with runtime management and try to evaluate the system at both stages [21, 31]. However, these methodologies are mostly restricted to the MPSoC domain and do not address the reconfigurable system domain. Unlike existing approaches that are either focused on design time or on runtime task mapping, we are focused on exploring and evaluating reconfigurable architectures at *design time* as well as at *runtime* during early design stages.

3. rSesame Framework

The rSesame [9] framework is a generic modeling and simulation infrastructure, which can explore and evaluate reconfigurable systems at early design stages both at design time and at runtime. It is built upon the Sesame framework [32]. The rSesame framework can be efficiently employed to perform DSE of the reconfigurable systems with respect to hardware-software partitioning, task mapping, and scheduling [10]. With the rSesame framework, an application task can be modeled either as a hardware (HW), or as a 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 based on the design time exploration of the system. At runtime, these tasks are mapped onto their corresponding resources based on time, resources, and conditions of the system.

The rSesame framework uses the Kahn Process Network (KPN) [33] at the granularity of coarse-grain tasks for application modeling. Each KPN process contains functional application code instrumented with annotations that generate read, write, and execute events describing the actions of the process. The generated traces are forwarded onto the architecture layer using an intermediate mapping layer, which consists of Virtual Processors (VPs) to schedule these traces. Along with the VPs, the mapping layer contains a Runtime Mapping Manager (RMM) that deals with the runtime mapping of the applications on the architecture. Depending on current system conditions, the RMM decides where and when to forward these events. To support its decision making, the RMM employs an arbitrary set of user-defined policies for runtime mapping, which can simply be plugged in and out of the RMM. The RMM also collaborates with other architectural components to gather architectural information. The architecture layer in the framework models the architectural resources and constraints. These architectural components are constructed from generic building blocks provided as a library, which contains components for processors, memories, on-chip network components, and so forth. As a result, any kind of reconfigurable architecture can be constructed from these generic components. Beside the regular parameters, such as computation and communication delays, other architectural parameters like reconfiguration delay and area for the reconfigurable architecture can also be provided as extra information to these components.

The rSesame framework provides various useful design parameters to the designer. These include the total execution time (in terms of simulated cycles), area usage, number of reconfigurations, percentage of reconfiguration, percentage of HW/SW execution, and reusability efficiency. These design parameters are described in more detail in the following.

3.1. Execution Time. The execution time is recorded in terms of simulated clock cycles. The SW execution time is the total number of cycles when all the tasks are mapped only on the GPP. The HW execution time is recorded when the tasks are mapped onto the FPGA. The speedup is calculated as a ratio of these two values.

3.2. Percentage of HW and SW Execution Time. The percentage of HW (SW) execution is computed as the total percentage of the execution time contributed by the FPGA (GPP) for HW (SW) execution of an application. Similarly, the percentage of reconfiguration time represents the percentage of the total execution time spent in reconfigurations. This provides an indication on the total time spent in the computation and in the reconfiguration. These values are calculated as follows.

The percentage of SW execution time is given by

$$\text{SW Exec}(\%) = \frac{\sum_{i=1}^{N} \#\text{SWEx}(T_i) \cdot T_{\text{SW}(i)}}{\text{TotalExecTime}} \cdot 100, \quad (1)$$

where $\#\text{SWEx}(T_i)$ is the total number of SW executions counted by the model for task T_i, $T_{\text{SW}(i)}$ is the software execution latency for task T_i, and TotalExecTime is the total simulated execution time.

The percentage of HW execution time is given by

$$\text{HW Exec.}(\%) \leq \frac{\sum_{i=1}^{N} \#\text{HWEx}(T_i) \cdot T_{\text{HW}(i)}}{\text{TotalExecTime}} \cdot 100, \quad (2)$$

where $\#\text{HWEx}(T_i)$ is the total number of HW executions counted for task T_i by the model, $T_{\text{HW}(i)}$ is the hardware execution latency for task T_i, and TotalExecTime is the total execution cycles incurred while running an application onto the given reconfigurable architecture.

Note that, the HW execution percentage can only be given here as an upper bound, since the execution of tasks on the FPGA can be performed in parallel. The metric calculated here is an accumulated value. The simulator, however, can give the actual value. A similar equation holds for the time spent reconfiguring, which is given as a percentage of the total execution time as follows:

$$\text{Recon}(\%) \leq \frac{\sum_{i=1}^{N} \#\text{Recon}(T_i) \cdot T_{\text{Recon}(i)}}{\text{TotalExecTime}} \cdot 100, \quad (3)$$

where $\#\text{Recon}(T_i)$ is the number of times T_i is configured, $T_{\text{Recon}(i)}$ is the reconfiguration delay of T_i, and TotalExecTime represents the total execution cycles incurred while running an application onto the given reconfigurable architecture.

3.3. Number of Reconfigurations. The number of reconfigurations is recorded as the total number of reconfigurations incurred during the execution of an application onto the given architecture. This provides an indication on how efficiently the reconfiguration delay is avoided, while mapping tasks onto the FPGA. For example, the mapping of task A, task B, and then task A again on the FPGA requires 3 reconfigurations, while by changing this mapping sequence to task A, task A and then task B, only 2 reconfigurations are required.

3.4. Time-Weighted Area Usage. The weighted area usage factor is a metric that computes how much area is used throughout the entire execution of an application on a particular architecture. This provides an indication on how efficiently the FPGA area is utilized. This metric is calculated as follows:

$$\text{Area Usage}(\%) = \frac{\sum_{i=1}^{N} \text{Area}(T_i) \cdot T_{\text{HW}(i)} \cdot \#\text{HWEx}(T_i)}{\text{TotalExecTime} \cdot \text{Area(FPGA)}} \cdot 100, \tag{4}$$

where $\text{Area}(T_i)$ is the area occupied by task T_i on the FPGA, $T_{\text{HW}(i)}$ is the hardware execution latency of T_i, $\#\text{HWEx}(T_i)$ is the total number of HW executions counted by the model for task T_i, Area(FPGA) is the total area available on the FPGA, and TotalExecTime is the total execution time of the application.

3.5. Reusability Efficiency. A task execution onto the FPGA has two phases: the *configuration phase*, where its configuration data that represents a task is loaded onto the FPGA, and the *running phase*, where the task is actually processing data. In an ideal case, a task can be configured onto the FPGA only once and it is executed in all other cases. Nonetheless, this is not always possible as the FPGA has limited area. The Reusability Efficiency (RE) is the ratio of the reconfiguration time that is saved due to the hardware configuration reuse to the total execution time of any task. The RE of a task can be defined as follows:

$$\text{RE}_{\text{task}} = \frac{(\#\text{HWEx} - \#\text{Recon}) \cdot T_{\text{Recon}}}{\#\text{HWEx} \cdot T_{\text{HW}} + \#\text{SWEx} \cdot T_{\text{SW}} + \#\text{Recon} \cdot T_{\text{Recon}}}, \tag{5}$$

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 RE of a task indicates the percentage of the total time saved by a task when multiple reconfigurations are avoided or, in other words, a task configuration is reused. The numerator in (5) represents the time that is saved when a mapping of a task is reused, and the denominator represents the total execution time. The total RE for an application can be calculated as the summation of the numerator in (5) for all N tasks divided by the total execution time for the whole application as follows:

$$\text{RE}_{\text{App}} \leq \frac{\sum_{i=1}^{N} (\#\text{HWEx}(i) - \#\text{Recon}(i)) \cdot T_{\text{Recon}(i)}}{\text{TotalExecTime}}. \tag{6}$$

Note that the RE 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 as an evaluation parameter to study the behavior of each task.

4. Case Study

We use the rSesame framework as a simulation platform for performing extensive evaluation of the various task mapping heuristics from the state of the art. In order to perform this case study, we constructed a Molen model using the rSesame framework for mapping an extended MJPEG application (see Section 4.2) onto the Molen reconfigurable architecture [34] (see Section 4.1). The Molen model is used to evaluate the different task mapping heuristics under consideration. We incorporated these heuristics as strategies for the Molen model to perform runtime task mapping of the extended MJPEG application onto the Molen architecture. We conducted an evaluation of these task mapping heuristics based on various system attributes recorded from the model.

The rSesame framework allows easy modification and adjustment of individual components in the model, while keeping other parts intact. As a result, the framework allows designers to experiment with different kinds of runtime task mapping heuristics. The considered heuristics have variable complexity in terms of their implementation and the nature of their execution. In the original context, they were used at different system stages, ranging from the lower architecture level to Operating System (OS), and the higher application levels. These heuristics are used as a strategy to perform runtime mapping decisions in the model. They are taken from literature, and have been adapted to fit in the framework. In the following, we discuss these heuristics in more detail.

4.1. As Much As Possible Heuristic (AMAP). AMAP tries to maximize the use of FPGA resources (such as area) as much as possible, and it performs task mapping based on resource availability. In this case, tasks are executed on the FPGA if the latter has enough resource to accommodate them; otherwise, they are executed on the GPP. This straightforward heuristic can be used as a simple resource management strategy in various domains.

Algorithm 1 presents the pseudocode that describes the functionality of the AMAP heuristic for performing runtime mapping of a task T_i. The heuristic chooses to execute task T_i onto the FPGA if there are sufficient resources (e.g., area in Algorithm 1) for T_i (line 3 to 6 in Algorithm 1). In all other conditions, tasks are executed on the GPP (line 7 to 9 in Algorithm 1).

```
(1) HW ← set of tasks mapped onto the FPGA
(2) SW ← set of tasks mapped onto the GPP
(3) if T_i.area ≤ area then
(4)    {T_i is mapped onto FPGA}
(5)    HW = HW ∪ T_i
(6)    area = area − T_i.area
(7) else
(8)    {Map T_i onto the GPP}
(9)    SW = SW ∪ T_i
(10) end if
```

Algorithm 1: Pseudocode for the As Much As Possible heuristic (AMAP) for mapping task T_i.

```
(1) HW ← set of tasks mapped onto the FPGA
(2) SW ← set of tasks mapped onto the GPP
(3) if T_i.area ≤ area then
(4)   if CB(T_i) > (T_SW(i) − T_HW(i)) then
(5)      {T_i is mapped onto the FPGA}
(6)      HW = HW ∪ T_i
(7)      area = area − T_i.area
(8)   end if
(9) else
(10)   {Not enough area, swap the mapped tasks.}
(11)   while area ≤ T_j.area and j ∈ HW do
(12)    if CB(T_i) - (T_SW(i) − T_HW(i)) > CB(Tj) then
(13)       area = area + T_j.area
(14)    end if
(15)   end while
(16)   if T_i.area ≤ area then
(17)    {T_i is mapped onto the FPGA}
(18)    HW = HW ∪ T_i
(19)    area = area −T_i.area
(20)   else
(21)    {Map T_i onto the GPP}
(22)    SW = SW ∪ T_i
(23)   end if
(24) end if
```

Algorithm 2: Pseudocode for the cumulative benefit heuristic (CBH) for the mapping on task T_i.

4.2. Cumulative Benefit Heuristic (CBH). CBH maintains a cumulative benefit (CB) value for each task that represents the amount of time that would have been saved up to that point if the task had always been executed onto the FPGA. Mapping decisions are made based on these values and on the available resources. For example, if the available FPGA resources are not sufficient to load the current task, other tasks can be swapped if the CB of the current task is higher than that of the to-be-swapped-out set. Huang and Vahid [22] used this heuristic for dynamic coprocessor management of reconfigurable architectures at architecture level.

Algorithm 2 presents the pseudocode that describes the functionalities of CBH for performing runtime mapping of a task T_i. If resources, such as area slices, are available in the FPGA, then T_i is executed onto the FPGA only if the CB of T_i is larger than its loading time defined by the difference between $T_{SW(i)}$ and $T_{HW(i)}$, where $T_{SW(i)}$ and $T_{HW(i)}$ are the software and the hardware latencies of task T_i, respectively (line 3 to 8 in Algorithm 2). In other cases, when the FPGA lacks current capacity for executing the task, the heuristic searches for a subset of FPGA-resident tasks, such that removing the subset yields sufficient resources in the FPGA to execute the current task. The condition, however, is such that all the tasks in the subset must have smaller CB value than the current task (line 9 to 18 in Algorithm 2). If such a subset is not attained, then the current task is executed by the GPP (line 19 to 22 in Algorithm 2).

```
(1) T ← set of all tasks.
(2) while T ! = ∅ and area ≤ Total_area do
(3)    Select T_i with maximum frequency count
(4)    if area + T_i.area ≤ Total_area then
(5)       map T_i onto the FPGA
(6)       area = area + T_i.area
(7)    else
(8)       map T_i onto the GPP
(9)    end if
(10)   Remove T_i from T
(11) end while
(12) Map rest of the tasks from T onto the GPP
```

Algorithm 3: Pseudocode for the Interval Based Heuristics (IBH) for the mapping on task T_i.

4.3. Interval-Based Heuristic (IBH). In IBH, the execution is divided into a sequence of time slices (intervals) for mapping and scheduling. At the beginning of each interval, a task is examined for its execution. In each interval, the execution frequency of each task is counted, and the mapping decisions are made based on the frequency count of the previous intervals, such that tasks with the highest frequency count are mapped onto the FPGA. In [23], this heuristic is used for resource management in a multithreaded environment at OS level.

Algorithm 3 presents the pseudocode that describes the functionalities of the IBH heuristic for performing runtime mapping in each interval for a set T of tasks. Working from the highest to the lowest frequency count, each task $T_i \in T$ that satisfies the current resource conditions is selected for FPGA execution. The area constraint is updated accordingly before considering the next task. This process continues until the FPGA is full or until there is no task left in T (line 2 to 6 in Algorithm 3). If the FPGA current capacity is not enough for executing any task from T, then these tasks are executed with the GPP (line 8 to 12 in Algorithm 3). As it can be seen in Algorithm 3, tasks are executed onto the FPGA based on frequency count, but other mapping criteria, such as speedup, can also be used.

4.4. Reusability-Based Heuristic (RBH). RBH is based on the hardware configuration reuse concept, which tries to avoid

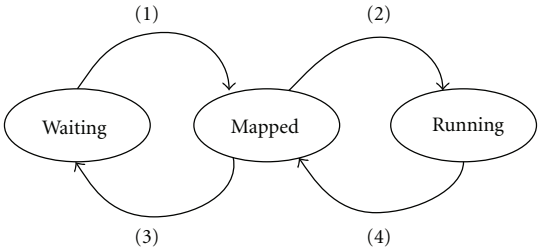

Figure 1: A finite-state machine (FSM) showing the different states of a task.

the reconfiguration overhead by reusing the configurations, which are already available on the FPGA. The basic idea of the heuristic is to avoid reconfiguration as much as possible, in order to reduce the total execution time. Especially in case of application domains, such as streaming and networking, where certain tasks are executed in a periodic manner, for example, on the basis of pixel blocks or entire frames, hardware configuration reuse can easily be exploited. To take advantage of such characteristics of streaming applications, we proposed this heuristic in [11].

For certain tasks that are mapped onto the FPGA, RBH preserves them in the FPGA after their execution. These tasks are not removed from the hardware, so that their hardware configurations can be reused when the task is re-executed. Reusing hardware configurations multiple times can significantly avoid reconfiguration overhead; thus, performance can be considerably improved. Unfortunately, preserving hardware configurations is not possible for all tasks. For this reason, the heuristic tries to preserve hardware configurations for selected tasks. For example, tasks that have higher reconfiguration delay and occur more frequently in the system have priority on being preserved in the FPGA.

We define three states for a task as shown in Figure 1: 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 onto 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. Figure 1 depicts a finite-state machine (FSM) showing the different states of a task, where the numbers 1 to 4 refer to the following state transitions:

(1) area becomes available or task dependency ends;

(2) task execution starts;

(3) other tasks need to be executed;

(4) task execution finishes, but the task may re-execute.

It should be noted that the mapped state has a reconfiguration delay associated with it. If a task transits from a waiting state to a running state, this delay is considered. However, if the task is already in the mapped state, its hardware configuration is saved in the FPGA and this delay is ignored. Thus, when the task needs to be re-executed, it can immediately start processing without reconfiguration. The performance can be significantly improved by avoiding the former transition.

Algorithm 4 presents the pseudocode that describes the functionality of the RBH heuristic for performing runtime mapping of a task T_i. If T_i is already configured, then it starts directly processing data (line 1 to 4 in Algorithm 4). However, if T_i is not currently available in the FPGA, then the task is evaluated for its speedup. If resources are available, T_i is executed onto the FPGA only if there is a performance gain (line 5 to 10 in Algorithm 4). 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:

$$\text{Speedup} = \begin{cases} \dfrac{T_{\text{SW}}}{T_{\text{HW}}} & t = 0, \\ \dfrac{\text{T}_{\text{SW}} \cdot (\#\text{HWEx} + \#\text{SWEx})}{\#\text{SWEx} \cdot \text{T}_{\text{SW}} + \#\text{HWEx} \cdot \text{T}_{\text{HW}} + \#\text{Recon} \cdot \text{T}_{\text{Recon}}} & t > 0, \end{cases} \tag{7}$$

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, and t is the execution time-line. When the application execution starts, $t = 0$. The heuristic maintains a profiling count of HW executions, SW executions, and reconfigurations for all tasks. Each time a task is executed, these counters 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 on 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 is 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). In the first step, a candidate set of tasks from the FPGA is selected, in such a way that these tasks are less beneficial than the current task in terms of speedup (line 12 to 16 in Algorithm 4). The speedup in this case is also calculated by using (7). In 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 21 in Algorithm 4). The RER value for each task is computed as follows:

$$\text{RER} = \frac{T_{\text{Recon}}}{T_{\text{HW}}} \cdot \text{Exec\_Freq}, \tag{8}$$

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 as follows:

$$\text{Exec}_{\text{Freq}} = \frac{\#\text{HWEx}}{\sum_{i=1}^{N} \text{HW}_i\text{Ex}}, \tag{9}$$

```
(1) {Task already mapped onto the 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 onto the FPGA, configure it.}
(8)          configure(T_i);
(9)          T_i.state ← RUNNING;
(10)      end if
(11)   else
(12)    for All tasks T_j onto the FPGA do
(13)          if SpeedUp(T_j) < SpeedUp(T_i) then
(14)             candidateSet = candidateSet ∪ T_j
(15)      end if
(16)   end for
(17)   while area ≤ T_i.area do
(18)      Select T_k ∈ candidate Set with lowest RER
(19)      removeSet = 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 onto the FPGA, configure it.}
(27)         configure(T_i);
(28)         T_i.state ← RUNNING;
(29)         end if
(30)      end if
(31)   end if
```

Algorithm 4: Pseudocode for the Reusability Based Heuristics (RBH) for the mapping on task T_i.

where the numerator represents the number of times a task is executed on the FPGA. The denominator represents the total hardware execution count of the entire application, and N represents the total number of tasks in the application.

A task with a high RER value indicates that it has high reconfiguration-per-execution delay, and it has executed frequently, in its history in the system, making it a probable candidate for future execution. The heuristic makes a careful selection while removing tasks from the FPGA. By preserving tasks with high RER values 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 value computed using (7) is not a constant factor. This value is continuously updated based on the execution profile of the task at runtime. Hence, mapping tasks onto the FPGA based on such value represents the precise system behavior at that instance of time. Note that the RBH is a generic heuristic, and it is not restricted to one type of resource or to one type of architecture. To perform runtime mapping decisions considering multiple resources (such as memory or DSP slices) for different architectural components, the parameters defining the heuristic can be easily customized, hence making it a flexible approach.

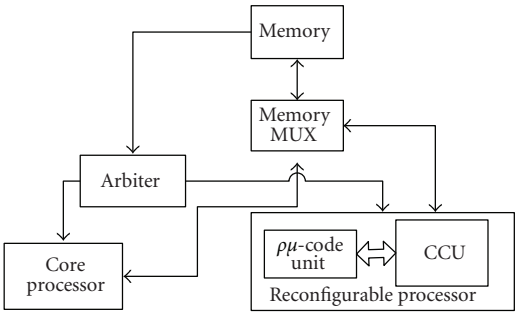

Figure 2: The machine organization of the Molen reconfigurable architecture. The architecture consists of a General Purpose Processor (GPP) and a Reconfigurable Processor (RP), which are coordinated by an arbiter.

4.4.1. The Molen Architecture. Figure 2 depicts the machine organization of the Molen polymorphic processor that is established on the basis of the tightly coupled coprocessor architectural paradigm [34, 35]. It consists of two different kinds of processors: the core processor that is a GPP and a Reconfigurable Processor (RP), such as an FPGA.

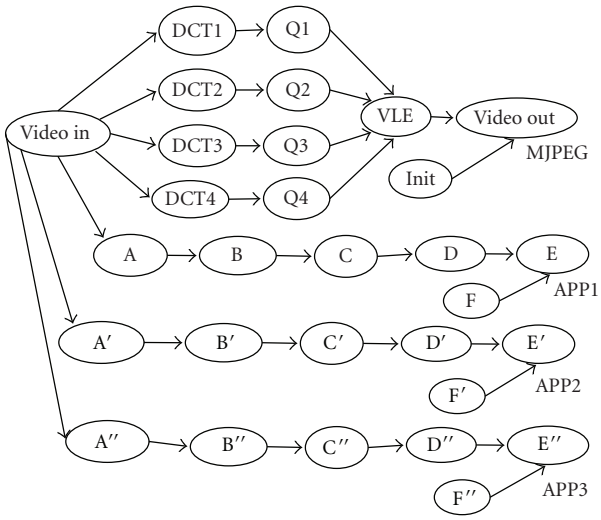

Figure 3: The Motion-JPEG (MJPEG) application model considered for the case study. The MJPEG application is extended by injecting sporadic applications in each frame.

The reconfigurable processor is further subdivided into the reconfigurable microcode ($\rho\mu$-code) unit and a *Custom Computing Unit* (CCU). The CCU is executed on the FPGA, and it supports additional functionalities, which are not implemented in the core processor. In order to speed up the program execution, parts of the code running on a GPP can be implemented on one or more CCUs.

The GPP and the RP are connected to an arbiter. The arbiter controls the coordination of the GPP and the RP by directing instructions to either of these processors. The code to be mapped onto the RP is annotated with special `pragma` directives. When the arbiter receives the `pragma` instruction for the RP, it initiates an "*enable reconfigurable operation*" signal to the reconfigurable unit, gives the data memory control to the RP, and drives the GPP into a waiting state. When the arbiter receives an "*end of reconfigurable operation*" signal, it releases the data memory control back to the GPP and the GPP can resume its execution. An operation executed by the RP is divided into two distinct phases: `set` and `execute`. In the `set` phase, the CCU is configured to perform the supported operations, and in the `execute` phase the actual execution of the operation is performed. The decoupling of `set` and `execute` phase allows the `set` phase to be scheduled well ahead of the `execute` phase and thereby hiding the reconfiguration latency.

4.4.2. The Application Model. We extend a Motion-JPEG (MJPEG) encoder application to use it as an application model for this case study. The corresponding KPN is shown in Figure 3. 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. A random number (0 to 3) of applications (APP1 to APP3) is injected in each frame of the MJPEG application in order to create a dynamic application behavior. These applications are considered as sporadic ones, which randomly appear in the system and compete with MJPEG for the resources. In this case study, we want to evaluate task mapping under different resource conditions; therefore we use only one application as a benchmark for comparing different heuristics. Nevertheless, the rSesame framework allows to evaluate any number of applications, architectures, and task mapping heuristics.

Table 1: Available area (in slices) for different FPGAs from the Xilinx Virtex4 FX family [36].

Hardware	Area (slices)
XC4VFX12	5472
XC4VFX20	8544
XC4VFX40	18624
XC4VFX60	25280
XC4VFX100	42176
XC4VFX140	63168

4.4.3. Experimental Setup. As discussed before, for this case study, we consider a model instantiated from the rSesame framework for the Molen reconfigurable architecture. The model instantiated for this case study consists of 30 CCUs allowing each task to be mapped onto one CCU. Note that the number of CCUs is a parameter that can be defined based on the number of pageable and HW tasks. For this case study, we consider all tasks as pageable to fully exploit the runtime mapping by deciding *where* and *when* to map them at runtime depending on the system condition. The model allows dynamic partial reconfiguration and, therefore, if the FPGA cannot accommodate all tasks at once, the latter can be executed after runtime reconfiguration.

We study and evaluate different task mapping heuristics from various domains by considering, for the same architecture model, different FPGA sizes. We consider six FPGAs from the Xilinx Virtex-4 FX family [36], namely, XC4VFX12, XC4VFX20, XC4VFX40, XC4VFX60, XC4VFX100, and XC4VFX140. These FPGAs have different available area (slices) as shown in Table 1. As a result, they are used to evaluate the runtime task mapping under different resource conditions. Note that, in this case study, we have used area as one dimensional space. Nevertheless, rSesame can evaluate any other types and numbers of architectural parameter. We assume that the Processor Local Bus (PLB) of these FPGAs is 4 bytes wide, and the Internal Configuration Access Port (ICAP) functions at 100 MHz; thus, its configuration speed is considered at 400 MB/sec [37].

We use estimated values of the computational latency, the area occupancy (on the FPGA), and the reconfiguration delay for each CCU. The computational latency values for the GPP model are initialized using the estimates obtained from literature [38, 39] (non-Molen specific).

We estimated area occupancy for each process mapped onto the CCU using the Quipu model [40]. Quipu establishes a relation between hardware and software, and it

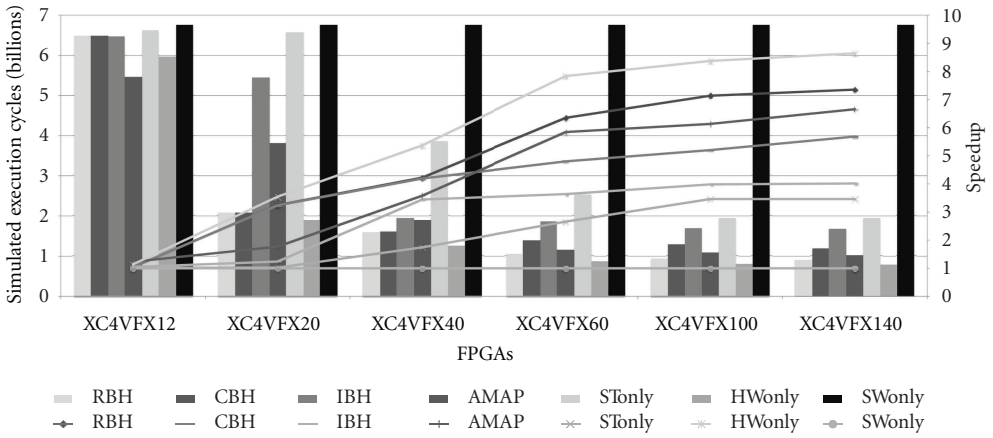

Figure 4: Comparison of the different heuristics tested in the proposed case study under different FPGAs conditions in terms of simulated execution time with corresponding application speedup. The application performance is proportional to the FPGA size. HWonly mapping has the best performance followed by RBH, AMAP, CBH, and IBH. STonly has the worst performance.

predicts FPGA resources from a C-level description of an application using Partial Least Squares Regression (PLSR) and Software Complexity Metrics (SCMs). Kahn processes contain functional C-code together with annotations that generate events such as read, execute, and write. As a result, Quipu can estimate area occupancy of each Kahn process. Such estimations are accepted while exploring systems at very early design stages with rSesame. In later design stages, other more refined models can be used to perform more accurate architectural explorations.

Based on the reconfiguration delay of each FPGA and the estimated area of each Kahn process, we computed the reconfiguration delay of each CCU using the following equation:

$$T_{\text{Recon}} = \frac{\text{CCU\_slices}}{\text{FPGA\_slices}} \cdot \frac{\text{FPGA\_bitstream}}{\text{ICAP\_bandwidth}}, \quad (10)$$

where CCU_slices is the total number of area slices a CCU requires, FPGA_slices is the total number of slices available on a particular FPGA, FPGA_bitstream is the bitstream size in MBs of the FPGA, and ICAP_bandwidth is the ICAP configuration speed. As a final remark, we assume that there is no delay associated with the runtime mapping, such as task migration and context switching.

5. Heuristics Evaluation

In this section, we provide a detailed analysis of the experimental results and their implications for the aforementioned case study. We conducted a wide variety of experiments on the above-mentioned task mapping heuristics with the Molen architecture by considering various FPGAs of different sizes. We evaluated and compared these heuristics based on the following parameters:

(i) the execution time,

(ii) the number of reconfigurations,

(iii) the percentage of hardware/software executions,

(iv) the reusability efficiency.

The detailed description of these parameters has been provided in Section 3. In the rest of this section, we discuss the evaluation results by using these parameters in more detail.

5.1. Execution Time. Figure 4 depicts the results of running different task mapping heuristics for mapping an extended MJPEG application 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. The software-only (SWonly) execution is measured when all the tasks are mapped onto the GPP. Similarly, the hardware-only (HWonly) execution is measured when all the tasks are mapped onto the FPGA. In HWonly, tasks are forced to be executed on the FPGA. However, if the task does not fit on the entire FPGA, the task is executed on the GPP. The static execution (STonly) is measured when only design time exploration is performed. In STonly execution, a fixed set of hardware tasks is considered for the FPGA mapping and this set does not change during the application runtime. For this experiment, tasks considered as fixed hardware are DCT1–DCT4 and Q1–Q4. The secondary y-axis (right) in Figure 4 represents the application speedup for each heuristic compared to the SWonly execution. The x-axis lists different types of FPGAs, which are ranked (from left to right) based on their sizes, such that XC4VFX12 has the smallest number of area slices and XC4VFX140 has the largest number of area slices (see Table 1). Several observations in terms of FPGA resources and speedup for different heuristics can be made from Figure 4.

A first observation that can be noticed from Figure 4 is that the application performance is proportional to the FPGA size to a certain degree: the bigger the available area in the FPGA, the higher the application performance. In the case of XC4VFX12, there is no significant performance gain

Table 2: The performance increase in different heuristics with the corresponding area increase in the FPGA. There is no linear relation between the area and the corresponding performance improvement.

Heuristics	Performance increase (%)	
	XC4VFX12 ⇒ XC4VFX20 (54% slice increase)	XC4VFX100 ⇒ XC4VFX140 (33% slice increase)
HWonly	67.9	3.14
STonly	0.69	0.007
AMAP	30	7.7
IBH	15	0.87
CBH	67	8.5
RBH	70	2.8

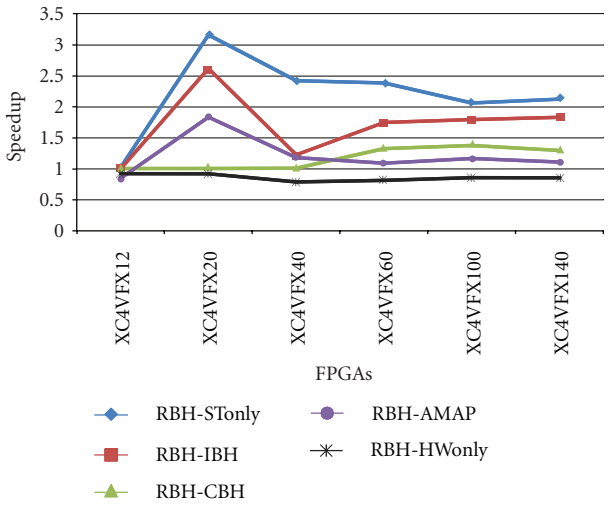

Figure 5: The performance increase of the RBH compared to HWonly, STonly, IBH, CBH, and AMAP. RBH performs better than AMAP under all resource conditions except XC4VFX12. RBH performs better than STonly, IBH, and CBH under all resource conditions.

by using any heuristic compared to the software execution. As there is a limited area, only few tasks can be mapped onto the FPGA; thus, performance is limited. Nevertheless, there is a notable performance improvement with the other FPGAs.

Secondly, while comparing the results of different heuristics for different FPGAs in Figure 4, we observe that there is no linear relation between the FPGA area and the corresponding performance. For instance, although XC4VFX20 has 54% more slices than XC4VFX12, the corresponding increase in the application performance is 67.9%, in the case of HWonly, as shown in Table 2. Similarly, there is 33% increase in area slices while comparing XC4VFX140 with XC4VFX100 in Table 1. Nevertheless, there is considerably lower increase in the performance in this case, as compared to the former case. The performance increase associated with the corresponding area increase in XC4VFX12 and XC4VFX20 as compared to XC4VFX100, and XC4VFX140 respectively, in case of different heuristics is reported in Table 2. The table depicts that there is no linear increase in the performance with area increase. This becomes obvious as the performance increase in an application is bounded by the degree of parallelism in that application. The use of more resources does not always guarantee a better application performance.

Another observation that can be made from Figure 4 is in terms of application performance of each heuristic. As it can be seen from the figure, STonly has the worst application performance, and HWonly has the best application performance. HWonly executes *all* tasks on the FPGA. As a result, it has approximately up to 9 times better performance than SWonly. STonly executes a fixed set of tasks on the FPGA, and mapping optimizations cannot be performed at runtime and, as a result, it has only upto 3 times better performance than SWonly. On the other hand, with runtime heuristics such as AMAP, IBH, CBH, and RBH, the task mapping is performed at runtime. When the application behavior changes due to the arrival of a sporadic application, task mapping is optimized, and better performance can be obtained in latter cases. This can be clearly seen in the figure, where the performance of the other heuristics, such as RBH, CBH, IBH, and AMAP, are bounded by HWonly and STonly.

While comparing the application performance of RBH against the other heuristics, we observe that RBH provides the best performance. RBH outperforms IBH under all resource conditions. RBH performs similar to CBH in the case of XC4VFX12, XC4VFX20, and XC4VFX40, while it performs better than CBH for the rest of the FPGAs. Task mapping is highly influenced by the task selection criteria and the FPGA size. CBH chooses a task with the highest SW/HW latency difference and executes that task in FPGA. RBH also maps tasks based on the speedup factor, but the major difference is in the way this value is calculated. RBH calculates the speedup value at runtime taking into account the past execution history, while with CBH, the SW/HW value is calculated statically. This difference significantly influences the performance of these heuristics. The performance increase of the RBH as compared to HWonly, STonly, IBH, CBH, and AMAP is reported in Figure 5. As it can be inferred from the figure, the performance improvement of the RBH compared to AMAP shows an irregular behavior. The RBH performs 10% worse than AMAP for XC4VFX12. However, the improvement significantly increases for XC4VFX20. For XC4VFX40, the improvement suddenly decreases to 10%. The improvement is regained for XC4VFX60 and stays identical for XC4VFX100 and XC4VFX140. AMAP performs task mapping based on the area availability in an ad hoc manner, in the sense that it tries to map as many tasks as possible at once. However, the RBH performs a selective task mapping based on the task speedup and the hardware configuration reuse. When area is limited, as in the case of XC4VFX12, not many hardware configurations can be preserved in the FPGA. Thus, configuration reuse cannot

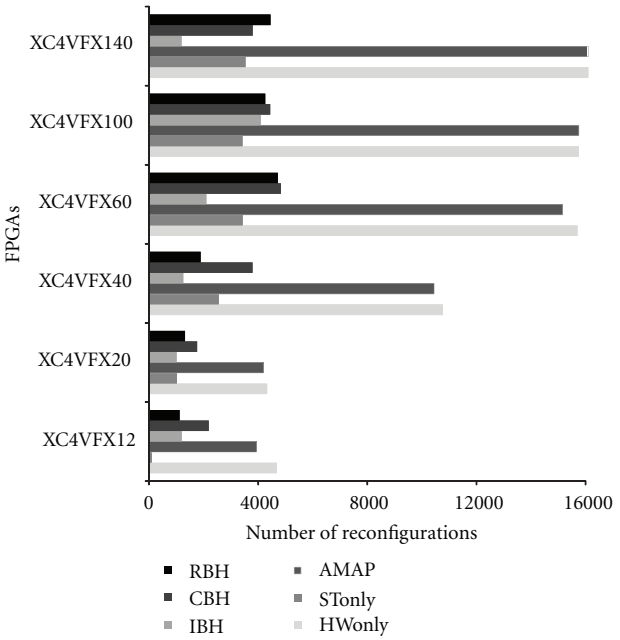

Figure 6: Heuristics comparison under different FPGAs conditions in terms of number of reconfigurations. There is a direct relation between the number of reconfigurations and the FPGA area.

be exploited with the RBH. As a result, AMAP performs better than RBH. With the increase in area, many hardware configurations can be preserved in the FPGA. Consequently, the RBH performs better than AMAP.

5.2. Number of Reconfigurations. Figure 6 depicts an overview of the number of reconfigurations for different heuristics, by considering different FPGAs. Several observations can be made from Figure 6 in terms of FPGA resources and the number of reconfigurations for the different heuristics. Only few tasks can be executed on the FPGA with limited area slices, contributing to the small reconfiguration counts. When the area slices increase, more tasks can be executed onto the FPGA, and, hence, reconfiguration counts increase. Nevertheless, the reconfiguration count is greatly influenced by the mapping strategies used. As it can be inferred from Figure 6, HWonly has relatively higher reconfigurations as compared to other heuristics. With HWonly, *all* tasks are executed to the FPGA, and hence, they are configured frequently. In large FPGAs, there is a possibility for CCUs to save and reuse their configurations and, hence, to avoid their reconfiguration. Therefore, reconfigurations saturate with large FPGAs. Similarly, STonly has a relatively low number of reconfigurations with small FPGAs, such as XC4VFX12 and XC4VFX20. The reconfiguration count increases in case of XC4VFX40 and XC4VFX60, and, then, it stays constant in all other cases. STonly executes a fixed set of HW tasks in all cases; since the number of HW task is constant, the reconfiguration also saturates.

We can observe from Figure 6 that AMAP has significantly higher reconfiguration counts unlike the other heuristics. AMAP performs task mapping based on the area availability in an adhoc manner, in the sense that any task can be mapped onto the FPGA. This leads to a significant increase in reconfiguration counts. It is worth noticing that the application performance in case of AMAP does not decrease drastically with the higher reconfiguration numbers. We may expect a significant performance decrease due to massive reconfigurations. The reconfiguration latency considered for a task is relatively small compared to the HW execution latency. Despite the larger number of reconfigurations, the performance can be considerably improved with the HW execution in such cases. Similarly, in the case of CBH, the reconfiguration counts are lower in smaller FPGAs due to lower hardware executions. This number increases with large FPGAs. There are no significant changes in the reconfiguration counts with the increase in area slices once sufficient area is available.

The number of reconfigurations for IBH is somewhat lower compared to the other heuristics, such as AMAP, CBH, and RBH under all FPGA conditions. This is not due to an efficient algorithm, which tries to optimize the reconfiguration delay, rather it is the effect of limited HW execution. In case of IBH, the mapping decision is changed only in the beginning of each interval, and the mapping behavior is fixed within an interval. Thus, a fixed set of tasks is mapped onto the FPGA during such an interval. This limits the hardware execution percentage, and hence, the reconfigurations. On the other hand, RBH reuses the hardware configurations to reduce the total number of reconfigurations. As a result, we observe a lower number of reconfigurations in case of RBH compared to CBH and AMAP in Figure 6. Note that IBH and STonly have lower reconfigurations than RBH as a consequence of their lower hardware execution. Nonetheless, RBH has a better reconfiguration-to-HW-execution ratio as compared to IBH and STonly, making the former better in terms of performance.

5.3. Percentage of Hardware Execution, Software Execution, and Reconfiguration. Figure 7 shows the comparison between different task mapping heuristics in terms of hardware execution, software execution and reconfiguration measured using (1), (2), and (3), respectively. The x-axis in the graph is stacked as 100%, and it shows the contribution of hardware execution, software execution, and reconfiguration to the total execution time. We observe that in few FPGAs the percentage of execution is greater than 100%. The hardware execution percentage measured in (1) is provided as an upper bound to address the parallel execution possibility of the FPGA. As a result, its value can go beyond the 100% limitation.

A first observation that can be made from Figure 7 is in terms of execution percentage and FPGA area. The limited area slices in the FPGA confines the HW execution percentage in smaller FPGAs. The hardware execution percentage increases considerably with more area slices, but this increase is not linear. As it can be seen from the figure, hardware

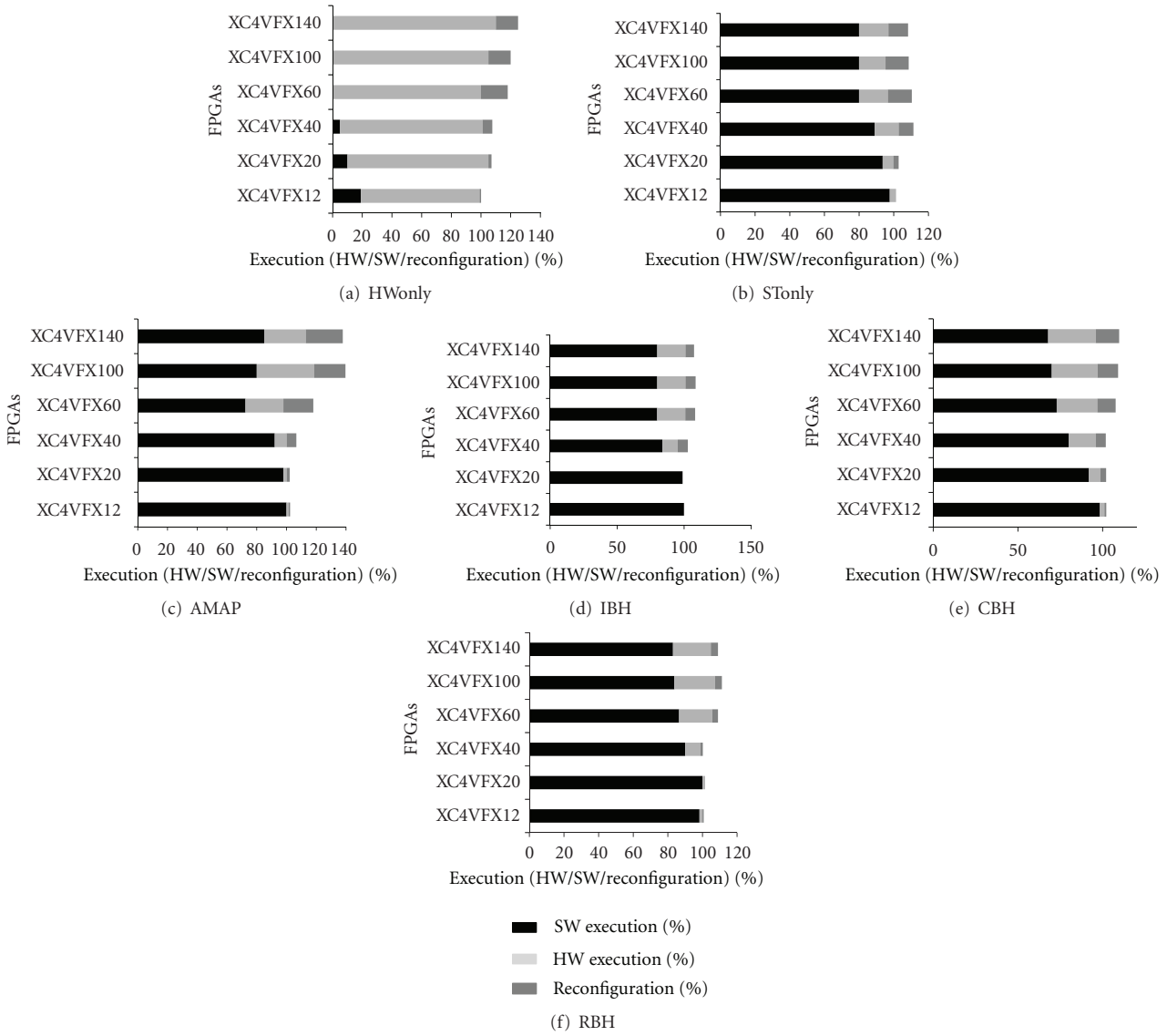

Figure 7: The comparison of different heuristics based on percentage of hardware execution, software execution, and reconfiguration. The hardware execution percentage is low in smaller FPGAs, and it increases considerably with more area slices.

execution percentage somewhat saturates with large FPGAs, such as XC4FX100 and XC4FX140. This observation is valid for all the runtime mapping heuristics including STonly and HWonly.

With HWonly mapping, *all* tasks are forced to be executed to the FPGA. However, if the task does not fit on the entire FPGA, then the task is executed with the GPP. Therefore, in Figure 7, we observe certain percentage of software execution with small FPGAs, but, with larger FPGA, there is only HW execution and the corresponding reconfiguration. With smaller FPGAs, almost no tasks are executed in hardware and, as a result, STonly has very minimal hardware execution (if any) and, therefore, the less reconfigurations. With the larger FPGAs, STonly has a relatively good but constant hardware execution and reconfiguration percentage, since it executes a fixed set of tasks on the FPGA.

While comparing the runtime heuristics, such as AMAP, CBH, IBH, and RBH, we can observe that AMAP has the best hardware execution percentage in larger FPGAs, followed by RBH and CBH. CBH and RBH somehow show similar behavior in terms of hardware execution percentage. However, in case of reconfiguration percentage, they do not follow the same trend. The reconfiguration is somewhat linear to the hardware execution in case of CBH. However, RBH does not show any linear increase in reconfiguration with hardware execution. RBH performs task mapping based on configuration reuse and, as a result, tries to avoid reconfiguration with more hardware executions. This behavior of RBH heuristic is apparent in the figure, especially

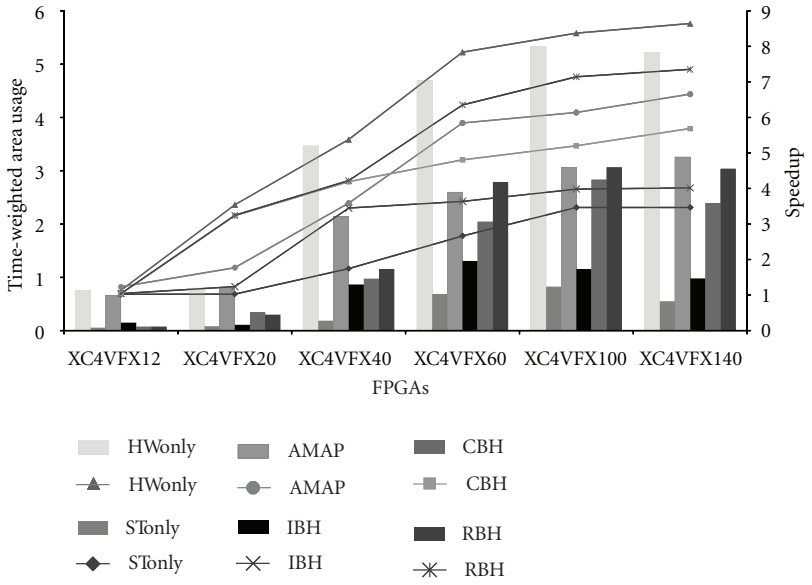

Figure 8: The comparison of different heuristics in terms of time-weighted area usage against speedup under different FPGAs conditions. The HWonly has the most time-weighted area usage, followed by the other runtime heuristics AMAP, CBH, IBH, and RBH.

in the case of moderate to large FPGAs, such as XC4FX60, XC4FX100, and XC4FX140. IBH follows a behavior similar to STonly in terms of software and hardware execution, as it also executes a fixed set of tasks on the FPGA.

By mapping more tasks onto the FPGA, the application can be accelerated, but it also has reconfiguration overhead. The efficiency of the mapping heuristics lies in finding the best mapping while minimizing the number of reconfigurations. Nevertheless, in Figure 7, we see almost a linear contribution of the reconfiguration overhead to the total execution time in all heuristics, except in RBH. This phenomenon is highly influenced by the policy implemented for task mapping. Another observation that can be made from the figure is the contribution of the hardware execution, SW execution, and reconfiguration to the total execution time. The figure shows that the GPP executes most of the application and the FPGA computes only less than 40% of the total application. This is due to the architectural restrictions of the Molen architecture. The GPP and the RP run in a mutual exclusive way, due to the processor/coprocessor nature of the architecture. This influences the mapping decision, which, in turn, contributes to the low hardware execution rates. This significantly increases the total execution time. Another reason for the lower percentage of hardware execution is due to the lower hardware latency for each task. The execution percentage is calculated as the ratio of execution latency of all tasks to the total execution time of an application. The hardware latency is comparatively lower than the SW latency for each task. Therefore, the corresponding hardware execution contribution is always lower compared to the percentage of SW execution.

5.4. Time-Weighted Area Usage. Figure 8 depicts an average time-weighted area usage measured using Equation (4) for different heuristics under different FPGA devices. The primary y-axis (left) in the graph represents the time-weighted area usage measured for each heuristic. The secondary y-axis (right) in the figure represents the application speedup for each heuristic compared to the SWonly execution. Several observations can be made from Figure 8 in terms of FPGA resources and time-weighted area usage of different heuristics. The first observation that can be made from the figure is in terms of time-weighted area usage and the hardware resource. As it can be seen from the figure, the time-weighted area usage is directly impacted by the number of area slices in the FPGA. With the limited area slices in small FPGAs, few tasks are executed in the FPGA, contributing to a smaller number of hardware executions. This, in turn, contributes to the lower area usage. With sufficient area slices, there is a considerable number of hardware executions and, hence, the area usage is high. Nonetheless, there is no linear relation between the time-weighted area usage and the available FPGA area. In XC4VFX140, the area usage is relatively low compared to XC4FX100, despite the fact that area slices are greater in the former. The area usage measured is the time-weighted factor, and it depends on the hardware execution, the total FPGA area and the total execution time, as shown in Equation (4). The increase in the area slices, with no significant increase in hardware executions, contributes to the lower area usage in the former case.

As it can be inferred from Figure 8, HWonly has the highest time-weighted area usage under all FPGA conditions. HWonly executes *all* tasks onto the FPGA and, as a result, the cost of using FPGA in this case is higher than all the other heuristics. STonly, however, has the lowest area usage due to its lower number of hardware executions and, therefore, its corresponding performance is also very poor. Similarly, AMAP has higher area usage compared to

other heuristics, such as CBH, IBH, and RBH, under all FPGA conditions, except XC4VFX60. AMAP performs task mapping based on area availability. As a matter of fact, it has a relatively higher number of hardware executions compared to the other heuristics and, therefore, it consumes additional area. RBH, on the other hand, has less time-weighted area usage. While comparing AMAP and RBH, we can observe that RBH performs somewhat better than AMAP in terms of performance. This implies that RBH reuses the hardware configuration already present in the FPGA to avoid reconfiguration overhead and, as a result, it can give better performance with the same amount of hardware resources as required by AMAP. Likewise, CBH has a comparable percentage of time-weighted area usage, but it lags behind in terms of speedup as compared to RBH. However, IBH has a considerably low percentage of area usage, as it also has lower hardware executions due to the constantly executed HW task set, and hence it also has lower performance. We can summarize that HWonly has the best performance but consumes more hardware resources. STonly has the lowest area usage but straggles behind in terms of performance. A tradeoff in terms of performance and resources can be obtained with task mapping at runtime, which performs selective task mapping onto the FPGA at runtime.

Another compelling observation that can be made from Figure 8 is about the lower value of the time-weighted area usage. The Molen architecture is based on processor/coprocessor paradigm. As a result, the GPP and the RP run in a mutual exclusive. This contributes to the lower number of hardware executions, which consequently increases the total execution time. Thus, these two factors significantly contribute to the low value of area usage. The area usage can be increased either by mapping more tasks onto the FPGA or by operating the RP and the GPP in parallel.

5.5. Reusability Efficiency. Figure 9 depicts the reusability efficiency (RE_{Task}) recorded for all CCUs using Equation (5) for different heuristics under different FPGA conditions. Several observations can be made from the figure in terms of FPGA area and the RE_{Task} of each CCU. Firstly, we observe that the CCU reuse is significantly affected by the number of area slices in the FPGA. Small FPGAs, such as XC4VFX12 and XC4VFX20, have many CCUs with RE_{Task} value zero. A CCU has an RE_{Task} value of zero under the following conditions:

(i) when a CCU is always mapped onto the GPP

(ii) when a CCU is configured every time it is executed on the FPGA.

With few resources in the FPGA, only a limited number of tasks can be executed to the FPGA. Additionally, in such cases, hardware configurations cannot be preserved for future reuse. As a result, CCUs have an RE_{Task} value of zero. Moreover, in this case, CCUs that are reused have a very small size in terms of area. With the increase in number of slices in the FPGA, more CCUs are reused. Medium-sized FPGAs, such as XC4VFX40 and XC4VFX60, reuse more CCUs compared to smaller FPGAs, but, in such cases, the reuse percentage is still low. With the larger FPGAs such as XC4VFX100 and XC4VFX140, more CCUs are reused with large RE_{Task} value.

As it can be inferred from Figure 9, HWonly has the best RE_{Task} for many CCUs in large FPGAs, such as XC4VFX100 and XC4VFX140. HWonly executes all the tasks on the FPGA and, as a result, it has high hardware execution count. However, with small FPGAs, all the tasks are configured, due to area restrictions, and there is no configuration reuse. On the other hand, with larger FPGAs, more configurations are saved and reused and, as a consequence, many CCUs have a considerably high RE_{Task} value. Similarly, STonly always maps a set of fixed tasks onto the FPGA. Out of these tasks, only a few number of small tasks can be reused. We notice that these CCUs have a relatively higher RE value compared to the ones reused with the AMAP heuristic in the figure. AMAP has higher HW execution percentage as compared to IBH and CBH. As a matter of fact, many tasks are reused in case of AMAP, but the reuse percentage of these CCUs is low. AMAP has no fixed pattern for task execution and, as a result, any task can be executed in FPGA. Therefore, the reusability is rather distributed among many CCUs. CBH, on the other hand, follows a specific policy for task execution in FPGA and hence executes a fixed set of selected task. As a result, a set of selected tasks is reused. Similar behavior is observed in the case of CBH. As it also executes a set of specific task within an interval, same tasks are reused (if any).

Likewise, from Figure 9, we observe that the RE_{Task} of RBH is better than that of other runtime heuristics for many CCUs. The impact of this hardware configuration reuse, in case of RBH, can be directly seen in terms of performance gain in Figure 4, where RBH has better speedup than the other heuristics. From Figure 10, we also observe that, for few tasks, RE_{Task} decreases when FPGA resources increase. With larger FPGAs, more tasks can fit onto the FPGA. As a result, these tasks are also mapped onto the FPGA, thus, over writing the saved configurations of other tasks. RE_{Task} for few tasks decreases in the FPGAs with moderate size. With the abundant resources, the hardware configuration can be saved for more tasks, and RE_{Task} increases again.

Note that STonly, AMAP, CBH, and IBH do not map the task based on the hardware configuration reuse. The reuse obtained in the case of STonly, AMAP, CBH, and IBH is a default value determined based on the arrival of the application task. If a CCU is already configured on the FPGA when its corresponding task arrives, the task can be executed without reconfiguration. However, the RBH reuses more hardware configurations than the other heuristics on top of the default value obtained.

Figure 10 depicts the total RE_{app} recorded using Equation (6) for different heuristics under different resource conditions. In the figure, we again observe that the reusability increases when using larger FPGAs. HWonly executes *all* the tasks in FPGA and, therefore, there can be a possibility that many of these tasks are reused when sufficient area is available, resulting into higher RE_{app}. STonly has almost a constant RE_{app} in larger FPGA, since it executes a constant set of tasks in FPGA. While comparing runtime heuristics, such as AMAP, CBH, IBH, and RBH, we can observe that

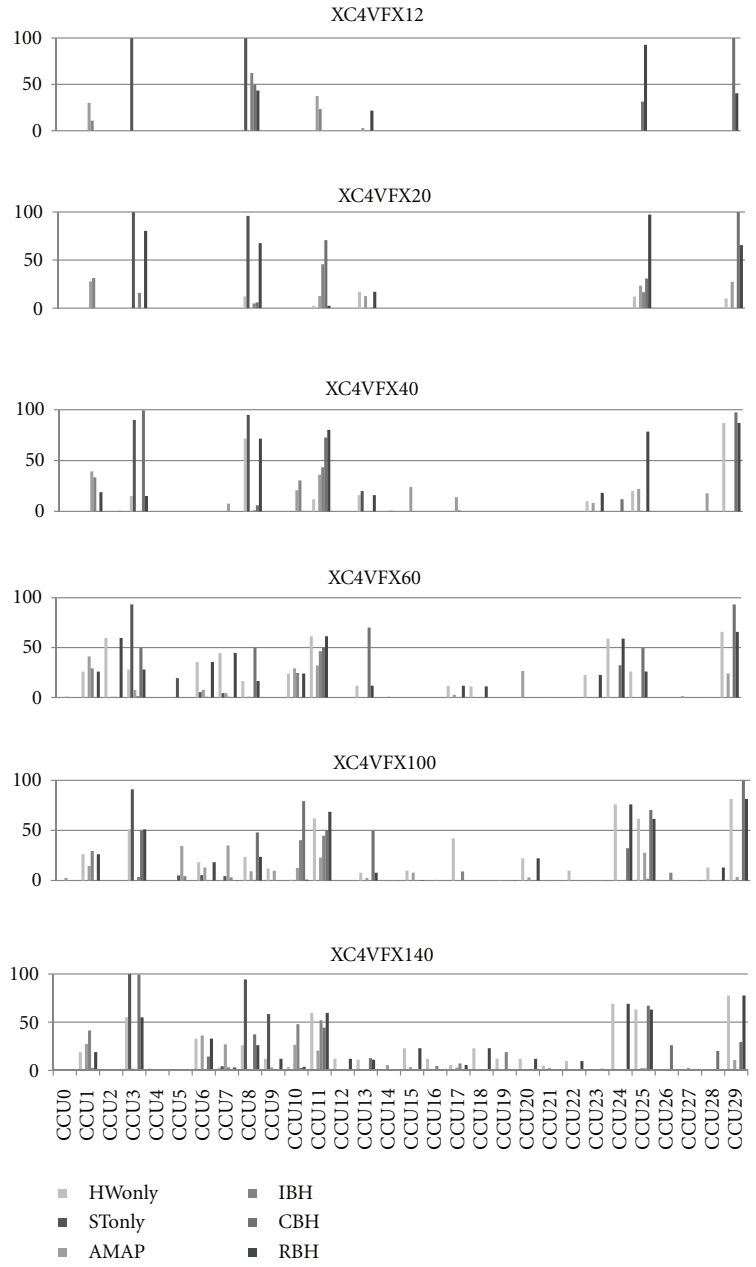

Figure 9: Reusability efficiency (RE_{Task}) of CCUs for different heuristics under different FPGA conditions. The CCU reuse is significantly affected by the number of area slices in the FPGA.

since the RBH has more CCUs reused than other heuristics, as shown in Figure 7(f), RBH has a better RE_{app} value than other heuristics in all resource conditions but XC4VFX12. Since XC4VFX12 has less area, all the heuristics have approximately the same value for RE_{app}. RE_{app} is the accumulation of the time saved due to hardware configuration reuse of each CCU. If all CCUs obtain the same value of the RE for a task mapping heuristic, then the application RE_{app} depends on the corresponding total execution time of that heuristics.

6. Observations and Conclusions

In this paper, we evaluated the task mapping of application(s) onto reconfigurable architectures under different resource conditions. We thoroughly evaluated various task mapping heuristics from the state of the art with the rSesame framework for a reconfigurable architecture with different FPGA resources using an extended MJPEG application. Based on the evaluation discussed in the previous sections, we can summarize the following conclusions.

(i) The comparison of different FPGAs shows that with very limited resources (in case of small FPGAs), the

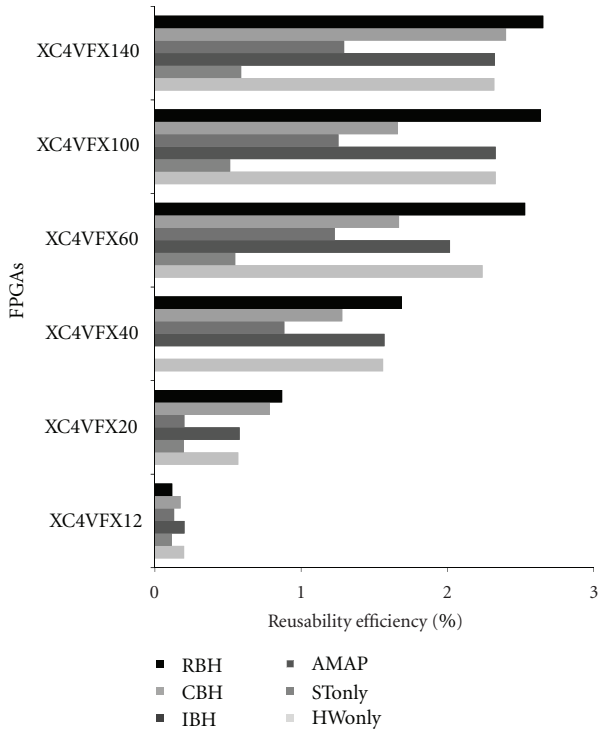

Figure 10: Heuristics comparison under different FPGAs conditions in terms of application reusability efficiency (RE_{app}). RBH has better RE_{app} compared to other heuristics.

number of tasks that can be mapped onto the FPGA is low. Consequently, these tasks are mapped onto the GPP. This leads to a poor application performance.

(ii) More resources (in case of moderate/higher FPGAs) imply more tasks mapped onto the FPGA. Consequently, we can obtain better application performance.

(iii) Runtime mapping provides better performance in case of dynamic application/architecture conditions. If the application behavior is well known in advance, design time mapping can give equal performance.

(iv) Mapping all the tasks onto the FPGA gives better performance, but it consumes more hardware resources. Runtime mapping performs task mapping based on the runtime system conditions. As a matter of fact, with the runtime mapping, a tradeoff can be obtained in terms of performance and resources.

(v) Comparing different heuristics, in case of limited resources conditions (small FPGAs), the adhoc task mapping of AMAP performs better compared to CBH, IBH, and RBH. Due to limited resources, the careful task selection with RBH, CHB, and IBH cannot be fully exploited in such cases.

(vi) The reuse of hardware configurations is better in case of sufficient resource conditions (medium-to-large FPGAs). As a result, the configuration reuse can be well exploited. Additionally, the RBH provides better application performance than AMAP and CBH.

(vii) In case of abundant resource conditions (very large FPGAs), the performance saturates due to application constraints. Under such scenarios, all the heuristics have similar performances.

Acknowledgments

This research is partially supported by Artemisia iFEST Project (Grant 100203), Artemisia SMECY (Grant 100230), FP7 Reflect (Grant 248976).

References

[1] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, 2002.

[2] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung, "Reconfigurable computing: architectures and design methods," *IEE Proceedings—Computers and Digital Techniques*, vol. 152, no. 2, pp. 193–207.

[3] S. Hauck, "The roles of FPGA's in reprogrammable systems," *Proceedings of the IEEE*, vol. 86, no. 4, pp. 615–638, 1998.

[4] R. J. F. Stephen, D. Brown, and J. Rose, *Field-Programmable Gate Arrays*, vol. 180 of *The Springer International Series in Engineering and Computer Science*, Kluwer Academic Publishers, 1992.

[5] Xilinx Corporation, *Coolrunner-II CPLDs Family Overview*, September 2008.

[6] R. W. Hartenstein and R. Kress, "A datapath synthesis system for thereconfigurable datapath architecture," in *Proceedings of the Asia andSouth Pacific Design Automation Conference (ASP-DAC '95)*, pp. 479–484, September 1995.

[7] S. Vassiliadis and D. Soudris, *Fine- and Coarse-Grain Reconfigurable Computing*, vol. 16, Springer, Berlin, Germany, 2007.

[8] N. S. Voros and and K. Masselos, *System-Level Design of Reconfigurable Systems-on-Chip*, Springer, Berlin, Germany, 1st edition, 2005.

[9] K. Sigdel, M. Thompson, C. Galuzzi, A. D. Pimentel, and K. Bertels, "rSesame —a generic system-level runtime simulation framework for reconfigurable architectures," in *Proceedings of the International Conference on Field-Programmable Technology (FPT '09)*, pp. 460–464, 2009.

[10] K. Sigdel, M. Thompson, C. Galuzzi, A. D. Pimentel, and K. Bertels, "Evaluation of runtime task mapping heuristics with rSesame—a case study," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '10)*, pp. 831–836, deu, March 2010.

[11] K. Sigdel, C. Galuzzi, K. Bertels, M. Thompson, and A. D. Pimentel, "Runtime task mapping based on hardware configuration reuse," in *Proceedingsof the International Conference on Reconfigurable Computing and FPGAs (ReConFig '10)*, pp. 25–30, 2010.

[12] P. V. Knudsen and J. Madsen, "PACE: a dynamic programming algorithm for hardware/software partitioning," in *Proceedings of the 4th International Workshop on Hardware/Software Co-Design (Codes/CASHE '96)*, pp. 85–92, March 1996.

[13] M. Kaul and R. Vemuri, "Design-space exploration for block-processing based temporal partitioning of run-time reconfigurable systems," *Journal of VLSI Signal Processing Systems for*

Signal, Image, and Video Technology, vol. 24, no. 2, pp. 181–209, 2000.

[14] B. Miramond and J. M. Delosme, "Design space exploration for dynamically reconflgurable architectures," in *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, pp. 366–371, March 2005.

[15] B. Miramond and J. M. Delosme, "Decision guide environment for design space exploration," in *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '05)*, pp. 881–888, September 2005.

[16] L. Y. Li and M. Shi, "Software-hardware partitioning strategy using hybrid genetic and Tabu search," in *Proceedings of the International Conference on Computer Science and Software Engineering (CSSE '08)*, vol. 4, pp. 83–86, 2008.

[17] B. Mei, P. Schaumont, and S. Vernalde, "A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems," in *Proceedings of the Annual Workshop on Circuits, Systemsand Signal Processing (ProRISC '00)*, pp. 1–8, November 2000.

[18] C. Haubelt, S. Otto, C. Grabbe, and J. Teich, "A system-level approachto hardware reconfigurable systems," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '05)*, pp. 298–301, 2005.

[19] G. Wang, W. Gong, and R. Kastner, "Application partitioning on programmable platforms using the ant colony optimization," *Embedded Computing*, vol. 2, no. 1, pp. 119–136, 2006.

[20] G. Still, R. Lysecky, and F. Vahid, "Dynamic hardware/software partitioning: a first approach," in *Proceedings of the Design AutomationConference (DAC '03)*, 2003.

[21] V. Nollet, P. Avasare, H. Eeckhaut, D. Verkest, and H. Corporaal, "Run-time management of a MPSoC containing FPGA fabric tiles," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 24–33, 2008.

[22] C. Huang and F. Vahid, "Dynamic coprocessor management for FPGA-enhanced compute platforms," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '08)*, pp. 71–78, 2008.

[23] W. Fu and K. Compton, "An execution environment for reconfigurable computing," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, pp. 149–158, April 2005.

[24] F. Ghaffari, M. Auguin, M. Abid, and M. B. Jemaa, "Dynamic and on-line design space exploration for reconfigurable architectures," *Transactions on High-Performance Embedded Architectures and Compilers*, vol. 4050, pp. 179–193, 2007.

[25] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and H. Yajun, "Resource manager for non-preemptive heterogeneous multiprocessor system-on-chip," in *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia (ESTIMEDIA '06)*, pp. 33–38, October 2006.

[26] O. Moreira, J. J. D. Mol, and M. Bekooij, "Online resource management in a multiprocessor with a network-on-chip," in *Proceedings of the ACM Symposium on Applied Computing (SAC '07)*, pp. 1557–1564, March 2007.

[27] L. T. Smit, J. L. Hurink, and G. J. M. Smit, "Run-time mapping of applications to a heterogeneous SoC," in *Proceedings of the International Symposium on System-on-Chip (SoC '05)*, pp. 78–81, November 2005.

[28] L. T. Smit, G. J. M. Smit, J. L. Hurink, H. Broersma, D. Paulusma, and P. T. Wolkotte, "Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '04)*, pp. 421–424, December 2004.

[29] P. K. F. Hölzenspies, G. J. M. Smit, and J. Kuper, "Mapping streaming applications on a reconfigurable MPSoC platform at run-time," in *Proceedings of the International Symposium on System-on-Chip (SOC '07)*, pp. 74–77, November 2007.

[30] M. A. A. Faruque, R. Krist, and J. Henkel, "ADAM: run-time agent-based distributed application mapping for on-chip communication," in *Proceedings of the 45th Design Automation Conference (DAC '08)*, pp. 760–765, June 2008.

[31] C. Ykman-Couvreur, E. Brockmeyer, V. Nollet, T. Marescaux, F. Catthoor, and H. Corporaal, "Design-time application exploration for MPSoC customized runtime management," in *Proceesings of InternationalSymposium on System-on-Chip (SOC '05)*, pp. 66–69, 2005.

[32] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 99–111, 2006.

[33] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress*, vol. 74, 1974.

[34] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. L. M. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, 2004.

[35] S. Vassiliadis, G. N. Gaydadjiev, K. Bertels, and E. M. Panainte, "The Molen programming paradigm," in *Proceeding of the International workshopon Systems, Architectures, Modeling and Simulation (SAMOS '03)*, pp. 1–30, July 2003.

[36] Xilinx Corporation, "Virtex-4 family overview (V3.0)".

[37] Xilinx DS86, "LogiCORE IP XPS HWICAP (v5.00a)," 2010.

[38] H. Nikolov, M. Thompson, T. Stefanov et al., "Daedalus: toward composable multimedia MPSoC design," in *Proceedings of the 45th annual Design Automation Conference (DAC '08)*, pp. 574–579, 2008.

[39] A. D. Pimentel, M. Thompson, S. Polstra, and C. Erbas, "Calibration of abstract performance models for system-level design space exploration," *Journal of Signal Processing Systems*, vol. 50, no. 2, pp. 99–114, 2008.

[40] R. Meeuws, Y. Yankova, K. Bertels, G. Gaydadjiev, and S. Vassiliadis, "A quantitative prediction model for hardware/software partitioning," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 735–739, August 2007.