Minimization of Average Execution Time Based on Speculative FPGA Configuration Prefetch

Adrian Lifa, Petru Eles and Zebo Peng
Linköping University, Linköping, Sweden
{*adrian.alin.lifa, petru.eles, zebo.peng*}*@liu.se*

***Abstract*—One of the main drawbacks that significantly impacts the performance of dynamically reconfigurable systems (like FPGAs), is their high reconfiguration overhead. Configuration prefetching is one method to reduce this penalty by overlapping FPGA reconfigurations with useful computations. In this paper we propose a speculative approach that schedules prefetches at design time and simultaneously performs HW/SW partitioning, in order to minimize the expected execution time of an application. Our method prefetches and executes in hardware those configurations that provide the highest performance improvement. The algorithm takes into consideration profiling information (such as branch probabilities and execution time distributions), correlated with the application characteristics. Compared to the previous state-of-art, we reduce the reconfiguration penalty with 34% on average, and with up to 59% for particular case studies.**

I. Introduction

In recent years, FPGA-based reconfigurable computing systems have gained in popularity because they promise to satisfy the simultaneous needs of high performance and flexibility [12]. Modern FPGAs provide support for partial dynamic reconfiguration [24], which means that parts of the device may be reconfigured at run-time, while the other parts remain fully functional. This feature offers high flexibility, but does not come without challenges: one major impediment is the high reconfiguration overhead.

Researchers have proposed several techniques to reduce the reconfiguration overhead. Such approaches are configuration compression [6] (which tries to decrease the amount of configuration data that must be transferred to the FPGA), configuration caching [6] (which addresses the challenge to determine which configurations should be stored in an on-chip memory and which should be replaced when a reconfiguration occurs) and configuration prefetch [4], [7], [10], [11], [13] (which tries to preload future configurations, overlapping as much as possible the reconfiguration overhead with useful computation).

This paper proposes a speculative approach to configuration prefetching that implicitly performs HW/SW partitioning and improves the state-of-art[1]. The high reconfiguration overheads make configuration prefetching a challenging task: the configurations to be prefetched should be the ones with the highest potential to provide a performance improvement and they should be predicted early enough to overlap the reconfiguration with useful computation. Therefore, the key for high performance of such systems is efficient HW/SW partitioning and intelligent prefetch of configurations.

II. Related Work

The authors of [3] and [18] proposed a partitioning algorithm, an ILP formulation and a heuristic approach to scheduling of task graphs. In [2], the authors present an exact and a heuristic algorithm that simultaneously partitions and schedules task graphs on FPGAs. The authors of [5] proposed a CLP formulation and a heuristic to find the minimal hardware overhead and corresponding task mapping for systems with communication security constraints. The main difference compared to our work is that the above papers address the optimization problem at a task level, and for a large class of applications (e.g. those that consist of a single sequential task) using such a task-level coarse granularity is not appropriate. Instead, it is necessary to analyze the internal structure and properties of tasks.

The authors of [19] present a hybrid design/run-time prefetch scheduling heuristic that prepares a set of schedules at design time and then chooses between them at run-time. This work uses the same task graph model as the ones before. The authors of [16] propose a scheduling algorithm that dynamically relocates tasks from software to hardware, or vice versa. In [6], the author proposes hybrid and dynamic prefetch heuristics that perform part or all of the scheduling computations at run-time and also require additional hardware. One major advantage of static prefetching is that, unlike the dynamic approaches mentioned above, it requires no additional hardware and it incurs minimal run-time overhead. Moreover, the solutions generated are good as long as the profile and data access information known at compile time are accurate.

The authors of [8] present an approach for accelerating the error detection mechanisms. A set of path-dependent prefetches are prepared at design time, and at run-time the appropriate action is applied, corresponding to the actual path taken. The main limitation of this work is that it is customized for the prefetch of particular error detection modules.

To our knowledge, the works most closely related to our own are [13], [7] and [10]. Panainte et al. proposed both an intra-procedural [10] and an inter-procedural [11] static prefetch scheduling algorithm that minimizes the number of executed FPGA reconfigurations taking into account FPGA area placement conflicts. In order to compute the locations where hardware reconfigurations can be anticipated, they first determine the regions not shared between any two conflicting hardware modules, and then insert prefetches at the beginning of each such region. This approach is too conservative and a more aggressive speculation could hide more reconfiguration overhead. Also, profiling information (such as branch probabilities and execution time distributions) could be used to prioritize between two non-conflicting hardware modules.

Li et al. continued the pioneering work of Hauck [4] in configuration prefetching. They compute the probabilities to reach any hardware module, based on profiling information [7]. This algorithm can be applied only after all the loops are identified and collapsed into dummy nodes. Then, the hardware modules are ranked at each basic block according to these probabilities and prefetches are issued. The main limitations of this work are that it removes all loops (which leads to loss of path information) and that it uses only probabilities to guide

[1]Configuration compression and caching are not addressed here, but they are complementary techniques that can be used in conjunction with our proposed approach.

978-1-4673-2921-7/12/$31.00 ©2012 IEEE

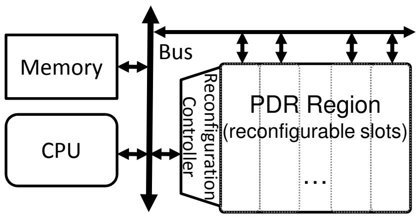


Figure 1: Architecture Model

prefetch insertion (without taking into account execution time distributions, for example). Also, this approach was developed for FPGAs with relocation and defragmentation, and it does not account for placement conflicts between modules.

To our knowledge, the state-of-art in static configuration prefetching for partially reconfigurable FPGAs is the work of Sim et al. [13]. The authors present an algorithm that minimizes the reconfiguration overhead for an application, taking into account FPGA area placement conflicts. Using profiling information, the approach tries to predict the execution of hardware modules by computing 'placement-aware' probabilities (PAPs). They represent the probabilities to reach a hardware module from a certain basic block without encountering any conflicting hardware module on the way. These probabilities are then used in order to generate prefetch queues to be inserted by the compiler in the control flow graph of the application. The main limitation of this work is that it uses only the 'placement-aware' probabilities to guide prefetch insertion. As we will show in this paper, it is possible to generate better prefetches (and, thus, further reduce the execution time of the application) if we also take into account the execution time distributions, correlated with the reconfiguration time of each hardware module.

III. System Model

A. Architecture Model

We consider the architecture model presented in Fig. 1. This is a realistic model that supports commercially available FPGAs (like, e.g., the Xilinx Virtex or Altera Stratix families). Many current reconfigurable systems consist of a host microprocessor connected, either loosely or tightly, to an FPGA (used as a coprocessor for hardware acceleration).

One common scenario for the tightly connected case is that the FPGA is partitioned into a static region, where the microprocessor itself and the reconfiguration controller reside, and a partially dynamically reconfigurable (PDR) region, where the application hardware modules can be loaded at run-time [20]. The host CPU executes the software part of the application and is also responsible for initiating the reconfiguration of the PDR region of the FPGA. The reconfiguration controller will configure this region by loading the bitstreams from the memory, upon CPU requests. While one reconfiguration is going on, the execution of the other (non-overlapping) modules on the FPGA is not affected.

We model the PDR region as a rectangular matrix of heterogeneous configurable tiles, organized as reconfigurable slots where hardware modules can be placed, similar to [17]. Although it is possible for the hardware modules to be relocated on the PDR region of the FPGA at run-time, this operation is known to be computationally expensive [14]. Thus, similar to the assumptions of Panainte [10] and Sim [13], we also consider that the placement of the hardware modules is decided at design time and any two hardware modules that have overlapping areas are in 'placement conflict'.

B. Application Model

The main goal of our approach is to minimize the expected execution time of a program executed on the hardware platform described above. We consider a *structured* [1] program[2], modeled as a control flow graph (CFG) $\mathcal{G}_{cf}(\mathcal{N}_{cf}, \mathcal{E}_{cf})$, where each node in $\mathcal{N}_{cf}$ corresponds to either a basic block (a straight-line sequence of instructions) or a candidate module to be executed on the FPGA, and the set of edges $\mathcal{E}_{cf}$ corresponds to the possible flow of control within the program. $\mathcal{G}_{cf}$ captures all potential execution paths and contains two distinguished nodes, *root* and *sink*, corresponding to the entry and the exit of the program. The function $prob : \mathcal{E}_{cf} \rightarrow [0,1]$ represents the probability of each edge in the CFG to be taken, and it is obtained by profiling the application. For each loop header n, $iter\_prob_n : \mathbb{N} \rightarrow [0,1]$ represents the probability mass function of the discrete distribution of loop iterations.

We denote the set of hardware candidates with $\mathcal{H} \subseteq \mathcal{N}_{cf}$ and any two modules that have placement conflicts with $m_1 \bowtie m_2$. We assume that all hardware modules in $\mathcal{H}$ have both a hardware implementation and a corresponding software implementation. Since sometimes it might be impossible to hide enough of the reconfiguration overhead for all candidates in $\mathcal{H}$, our technique will try to decide at design time which are the most profitable modules to insert prefetches for (at a certain point in the CFG). Thus, for some candidates, it might be better to execute the module in software, instead of inserting a prefetch too close to its location (because waiting for the reconfiguration to finish and then executing the module on the FPGA is slower than executing it in software). This approach will implicitly generate a HW/SW partitioning.

The set $\mathcal{H}$ can be determined automatically [15], [21], or by the designer, and might contain, for example, the computation intensive parts of the application, identified after profiling. Please note that it is not necessary that all candidate modules from $\mathcal{H}$ will end up on the FPGA. Our technique will try to use the limited hardware resources as efficiently as possible by choosing to prefetch the modules with the highest potential to reduce the expected execution time. Thus, together with the prefetch scheduling we also perform a HW/SW partitioning of the modules from $\mathcal{H} \subseteq \mathcal{N}_{cf}$.

For each node $n \in \mathcal{N}_{cf}$ we assume that we know its software execution time[3], $sw : \mathcal{N}_{cf} \rightarrow \mathbb{R}^+$. For each hardware candidate $m \in \mathcal{H}$, we also know its hardware execution time[3], $hw : \mathcal{H} \rightarrow \mathbb{R}^+$. The function $area : \mathcal{H} \rightarrow \mathbb{N}$, specifies the area that hardware modules require, $size : \mathcal{H} \rightarrow \mathbb{N} \times \mathbb{N}$ gives their size, and $pos : \mathcal{H} \rightarrow \mathbb{N} \times \mathbb{N}$ specifies the position where they were placed on the reconfigurable region. Since for all modules in $\mathcal{H}$ a hardware implementation and placement are known at design time, we also know the reconfiguration duration, which we assume it is given by a function $rec : \mathcal{H} \rightarrow \mathbb{R}^+$.

C. Reconfiguration API

We adopt the reconfiguration library described in [14], that defines an interface to the reconfiguration controller, and enables the software control of reconfiguring the FPGA. The

[2]Since any non-structured program is equivalent to some structured one, our assumption loses no generality.

[3]The execution time could be modeled as a discrete probability distribution as well, without affecting our overall approach for configuration prefetching.

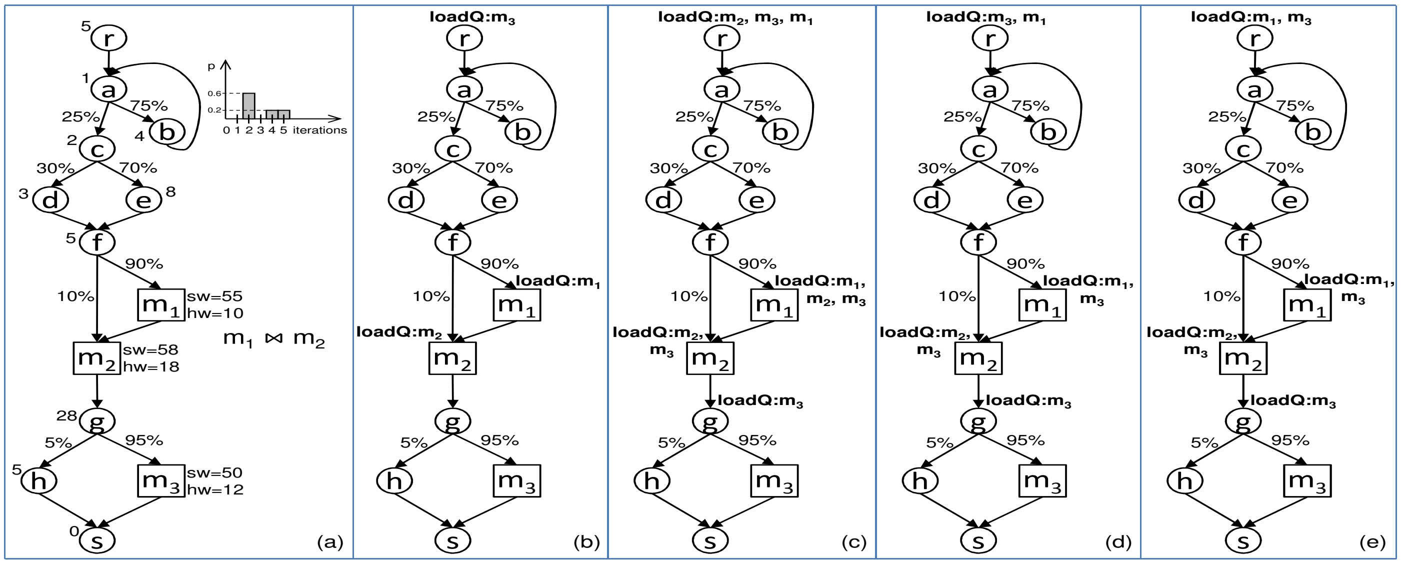


Figure 2: Motivational Example

library defines the following functions to support initialization, preemption and resumption of reconfigurations:

- $load(m)$: Non-blocking call that requests the reconfiguration controller to start or resume loading the bitstream of module m.
- $currently\_reconfig()$: Returns the id of the hardware module being currently reconfigured, or -1 otherwise.
- $is\_loaded(m)$: Returns *true* if the hardware module m is already loaded on the FPGA, or *false* otherwise.
- $exec(m)$: Blocking call that returns only after the execution of hardware module m has finished.

D. Middleware and Execution Model

Let us assume that at each node $n \in \mathcal{N}_{cf}$ the hardware modules to be prefetched have been ranked at compile-time (according to some strategy) and placed in a queue (denoted *loadQ*). The exact hardware module to be prefetched will be determined at run-time (by the middleware, using the reconfiguration API), since it depends on the run-time conditions. If the module with the highest priority (the head of *loadQ*) is not yet loaded and is not being currently reconfigured, it will be loaded at that particular node. If the head of *loadQ* is already on FPGA, the module with the next priority that is not yet on the FPGA will be loaded, but only in case the reconfiguration controller is idle. Finally, if a reconfiguration is ongoing, it will be preempted only in case a hardware module with a priority higher than that of the module being reconfigured is found in the current list of candidates (*loadQ*).

At run-time, once a hardware module $m \in \mathcal{H}$ is reached, the middleware checks whether m is already fully loaded on the FPGA, and in this case it will be executed there. Thus, previously reconfigured modules are reused. Otherwise, if m is currently reconfiguring, the application will wait for the reconfiguration to finish and then execute the module on FPGA, but only if this generates a shorter execution time than the software execution. If none of the above are true, the software version of m will be executed.

IV. Problem Formulation

Given an application (as described in Sec. III-B) intended to run on the reconfigurable architecture described in Sec. III-A, our goal is to determine, at each node $n \in \mathcal{N}_{cf}$, the *loadQ* to be used by the middleware (as described in Sec. III-D), such that the expected execution time of the application is minimized. This will implicitly also determine the HW/SW partitioning of the candidate modules from $\mathcal{H}$.

V. Motivational Example

Let us consider the control flow graph (CFG) in Fig. 2a, where candidate hardware modules are represented with squares, and software nodes with circles. The discrete probability distribution for the iterations of the loop $a - b$, the software and hardware execution times for the nodes, as well as the edge probabilities, are illustrated on the graph. The reconfiguration times are: $rec(m_1) = 37$, $rec(m_2) = 20$, $rec(m_3) = 46$. We also consider that hardware modules m_1 and m_2 are conflicting due to their placement ($m_1 \bowtie m_2$).

Let us try to schedule the configuration prefetches for the three hardware modules on the given CFG. If we use the method developed by Panainte et al. [10], the result is shown in Fig. 2b. As we can see, the $load$ for m_3 can be propagated upwards in the CFG from node m_3 up to r. For nodes m_1 and m_2 it is not possible (according to this approach) to propagate their $load$ calls to their ancestors, because they are in placement conflict. The data-flow analysis performed by the authors is too conservative, and the propagation of prefetches is stopped whenever two $load$ calls targeting conflicting modules meet at a common ancestor (e.g. node f for m_1 and m_2). As a result, since the method fails to prefetch modules earlier, the reconfiguration overhead for neither m_1, nor m_2, can be hidden at all. Only module m_3 will not generate any waiting time, since the minimum time to reach it from r is $92 > rec(m_3) = 46$. Using this approach, the application must stall (waiting for the reconfigurations to finish) $W_1 = 90\% \cdot rec(m_1) + rec(m_2) = 90\% \cdot 37 + 20 = 53.3$ time units on average (because m_1 is executed with a probability of 90%, and m_2 is always executed).

Fig. 2c shows the resulting prefetches after using the method proposed by Li et al. [7]. As we can see, the prefetch queue generated by this approach at node r is $loadQ : m_2, m_3, m_1$, because the probabilities to reach the hardware modules from r are 100%, 95% and 90% respectively. Please note that this method is developed for FPGAs with relocation and defragmentation and it ignores placement conflicts. Also, the load queues are generated considering only the probability to reach a module (and ignoring other factors, such as the execution time distribution from the prefetch point up to the prefetched module). Thus, if applied to our example, the method performs poorly: in 90% of the cases, module m_1 will replace module m_2 (initially prefetched at r) on the FPGA. In this cases, none of the reconfiguration overhead for m_1 can be hidden, and in addition, the initial prefetch for m_2 is wasted. The average waiting time for this scenario is $W_2 = 90\% \cdot rec(m_1) + (100\% - 10\%) \cdot rec(m_2) = 90\% \cdot 37 + 90\% \cdot 20 = 51.3$ time units (the reconfiguration overhead is hidden in 10% of the cases for m_2, and always for m_3).

For this example, although the approach proposed by Sim et al. [13] tries to avoid some of the previous problems, it ends up with similar waiting time. The method uses 'placement-aware' probabilities (PAPs). For any node $n \in \mathcal{N}_{cf}$ and any hardware module $m \in \mathcal{H}$, $PAP(n,m)$ represents the probability to reach module m from node n, without encountering any conflicting hardware module on the way. Thus, the prefetch order for m_1 and m_2 is correctly inverted since $PAP(r,m_1) = 90\%$, as in the previous case, but $PAP(r,m_2) = 10\%$, instead of 100% (because in 90% of the cases, m_2 is reached via the conflicting module m_1). Unfortunately, since the method uses only PAPs to generate prefetches, and $PAP(r,m_3) = 95\%$ (since it is not conflicting with neither m_1, nor m_2), m_3 is prefetched before m_1 at node r, although its prefetch could be safely postponed. The result is illustrated in Fig. 2d (m_2 is removed from the load queue of node r because it conflicts with m_1, which has a higher PAP). These prefetches will determine that no reconfiguration overhead can be hidden for m_1 or m_2 (since the long reconfiguration of m_3 postpones their own one until the last moment). The average waiting time for this case is $W_3 = 90\% \cdot rec(m_1) + rec(m_2) = 90\% \cdot 37 + 20 = 53.3$ time units.

If we examine the example carefully, we can see that taking into account only the 'placement-aware' probabilities is not enough. The prefetch generation mechanism should also consider the distance from the current decision point to the hardware modules candidate for prefetching, correlated with the reconfiguration time of each module. Our approach is to estimate the performance gain associated with starting the reconfiguration of a certain module at a certain node in the CFG. We do this by considering both the execution time gain resulting from the hardware execution of that module (including any stalling cycles spent waiting for the reconfiguration to finish) compared to the software execution, and by investigating how this prefetch influences the execution time of the other reachable modules. For the example presented here, it is not a good idea to prefetch m_3 at node r, because this results in a long waiting time for m_1 (similar reasoning applies for prefetching m_2 at r). The resulting prefetches are illustrated in Fig. 2e. As we can see, the best choice of prefetch order is m_1, m_3 at node r (m_2 is removed from the load queue because it conflicts with m_1), and this will hide most of the reconfiguration overhead for m_1, and all for m_3. The overall average waiting time is $W = 90\% \cdot \overline{W}_{rm_1} + rec(m_2) = 90\% \cdot 4.56 + 20 \approx 24.1$, less than half of the penalties generated by the previous methods (Sec. VI-B and Fig. 3 explain the computation of the average waiting time incurred by m_1, $\overline{W}_{rm_1} = 4.56$ time units).

Algorithm 1 Generating the prefetch queues

```
1: procedure GENERATEPREFETCHQ
2:     for all $n \in \mathcal{N}_{cf}$ do
3:         for all $\{m \in \mathcal{H} | PAP(n,m) \neq 0\}$ do
4:             if $\overline{G}_{nm} > 0 \vee m$ in loop then
5:                 compute priority function $C_{nm}$
6:         $loadQ(n) \leftarrow$ modules in decreasing order of $C_{nm}$
7:         remove conflicting modules from $loadQ(n)$
8:     eliminate redundant prefetches
```

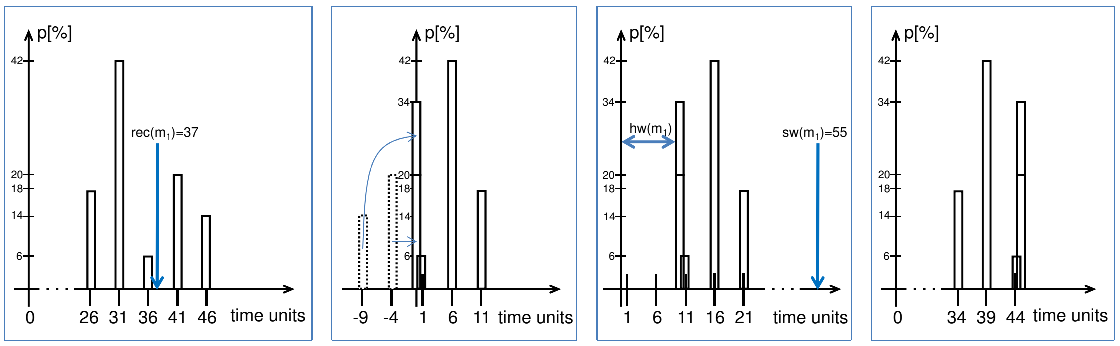
VI. Speculative Prefetching

Our overall strategy is shown in Algorithm 1. The main idea is to intelligently assign priorities to the candidate prefetches and determine the load queue (*loadQ*) at every node in the CFG (line 6). We try to use all the available knowledge from the profiling in order to take the best possible decisions and speculatively prefetch the hardware modules with the highest potential to reduce the expected execution time of the application. The intelligence of our algorithm resides in computing the priority function C_{nm} (see Sec. VI-A), which tries to estimate at design time what is the impact of reconfiguring a certain module on the average execution time. We consider for prefetch only the modules for which it is profitable to start a prefetch at the current point (line 4): either the average execution time gain $\overline{G}_{nm}$ (over the software execution of the candidate) obtained if its reconfiguration starts at this node is greater than 0, or the module is inside a loop (in which case, even if the reconfiguration is not finished in the first few loop iterations and we execute the module in software, we will gain from executing the module in hardware in future loop iterations). Then we sort the prefetch candidates in decreasing order of their priority function (line 6), and in case of equality we give higher priority to modules placed in loops. After the *loadQ* has been generated for a node, we remove all the lower priority modules that have area conflicts with the higher priority modules in the queue (line 7). Once all the queues have been generated, we eliminate redundant prefetches (all consecutive candidates at a child node that are a starting subsequence at all its parents in the CFG), as in [7] or [13] (line 8). The exact hardware module to be prefetched will be determined by the middleware at run-time, as explained in Sec. III-D.

A. The Prefetch Priority Function C_{nm}

Our prefetch function represents the priorities assigned to the hardware modules reachable from a certain node in the CFG, and thus determines the *loadQ* to insert at that location. Considering that the processor must stall if the reconfiguration overhead cannot be completely hidden and that some candidates will provide a higher performance gain than others, our priority function will try to estimate the overall impact on the average execution time that results from different prefetches being issued at a particular node in the control flow graph (CFG). In order to accurately predict the next configuration to prefetch, several factors have to be considered.

The first one is represented by the 'placement-aware' probabilities (PAPs), computed with the method from [13]. The second factor that influences the decision of prefetch scheduling is represented by the execution time gain distributions (that will be discussed in detail in Sec. VI-B). The gain distributions reflect the reduction of execution time resulting from prefetching a certain candidate and executing it in hardware, compared to executing it in software. They are directly impacted by the



(a) X_{rm_1} (b) W_{rm_1} (c) $W_{rm_1} + hw(m_1)$ (d) G_{rm_1}

Figure 3: Computing the Gain Probability Distribution Step by Step

waiting time distributions (which capture the relation between the reconfiguration time for a certain hardware module and the execution time distribution between the prefetch node in the CFG and that module).

We denote the set of hardware modules for which it is profitable to compute the priority function at node n with $Reach(n) = \{m \in \mathcal{H} \mid PAP(n,m) \neq 0 \wedge (\overline{G}_{nm} > 0 \vee m\ in\ loop)\}$. For our example in Fig. 2a, $Reach(r) = \{m_1, m_2, m_3\}$, but $Reach(m_3) = \emptyset$, because it does not make sense to reconfigure m_3 anymore (although $PAP(m_3, m_3) = 100\%$, we have the average waiting time $\overline{W}_{m_3m_3} = rec(m_3)$ and $rec(m_3)+hw(m_3) = 46+12 > 50 = sw(m_3)$). Thus, we do not gain anything by starting the reconfiguration of m_3 right before it is reached, i.e. $\overline{G}_{m_3m_3} = 0$. Considering the above discussion, our priority function expressing the reconfiguration gain generated by prefetching module $m \in Reach(n)$ at node n is defined as:

$$\begin{aligned} C_{nm} &= PAP(n,m) \cdot \overline{G}_{nm} \\ &+ \sum_{k \in MutEx(m)} PAP(n,k) \cdot \overline{G}_{sk} \\ &+ \sum_{k \notin MutEx(m)} PAP(n,k) \cdot \overline{G}^{k}_{nm} \end{aligned}$$

In the above equation, $\overline{G}_{nm}$ denotes the average execution time gain generated by prefetching module m at node n (see Sec. VI-B), $MutEx(m)$ denotes the set of hardware modules that are executed mutually exclusive with m, the index s in $\overline{G}_{sk}$ represents the node where the paths leading from n to m and k split, and $\overline{G}^{k}_{nm}$ represents the expected gain generated by k, given that its reconfiguration is started immediately after the one for m.

The first term of the priority function represents the contribution (in terms of average execution time gain) of the candidate module m, the second term tries to capture the impact that the reconfiguration of m will produce on other modules that are executed mutually exclusive with it, and the third term captures the impact on the execution time of modules that are not mutually exclusive with m (and might be executed after m). In Fig. 2a, modules m_1, m_2 and m_3 are not mutually exclusive. Let us calculate the priority function for the three hardware modules from Fig. 2a at node r (considering their areas proportional with their reconfiguration time). $C_{rm_1} = 90\% \cdot 40.44 + 10\% \cdot 36.96 + 95\% \cdot 38 \approx 76.19$, $C_{rm_2} = 10\% \cdot 40 + 90\% \cdot 22.5 + 95\% \cdot 38 \approx 60.35$ and $C_{rm_3} = 95\% \cdot 38 + 90\% \cdot 1.72 + 10\% \cdot 30.96 \approx 40.74$ (the computation of execution time gains is discussed in Sec. VI-B). As we can see, since $C_{rm_1} > C_{rm_2} > C_{rm_3}$, the correct *loadQ* of prefetches is generated at node r. Note that m_2 is removed from the queue because it is in placement conflict with m_1, which is the head of the queue (see line 7 in Algorithm 1).

Algorithm 2 Computing the average execution time gain

```
1: procedure AVGEXECTIMEGAIN($n$, $m$)
2:     construct subgraph with nodes between $n$ and $m$
3:     build its FCDT (saved as a global variable)
4:     $X_{nm} \leftarrow ExecTimeDist(n, m)$
5:     $W_{nm} \leftarrow max(0, rec(m) - X_{nm})$
6:     $G_{nm} \leftarrow max(0, sw(m) - (W_{nm} + hw(m)))$
7:     for all $x \in \{y \mid g_{nm}(y) \neq 0\}$ do
8:         $\overline{G}_{nm} \leftarrow \overline{G}_{nm} + x \cdot g_{nm}(x)$
9:     return $\overline{G}_{nm}$
```

B. Estimating the Average Execution Time Gain $\overline{G}_{nm}$

Let us consider a node $n \in \mathcal{N}_{cf}$ from the CFG and a hardware module $m \in \mathcal{H}$, reachable from n. Given that the reconfiguration of module m starts at node n, we define the average execution time gain $\overline{G}_{nm}$ as the expected execution time that is saved by executing m in hardware (including any stalling cycles when the application is waiting for the reconfiguration of m to be completed), compared to a software execution of m. Algorithm 2 presents our method for computing the average gain $\overline{G}_{nm}$. The first steps are to construct the subgraph with all the nodes between n and m and to build its forward control dependence tree [1] (lines 2-3). Then we estimate the distance (in time) from n to m (line 4). Let X_{nm} be the random variable associated with this distance. The waiting time is given by the random variable $W_{nm} = max(0, rec(m) - X_{nm})$ (see line 5). Note that the waiting time cannot be negative (if a module is already present on FPGA when we reach it, it does not matter how long ago its reconfiguration finished). The execution time gain is given by the distribution of the random variable $G_{nm} = max(0, sw(m) - (W_{nm} + hw(m)))$ (see line 6). In case the software execution time of a candidate is shorter than waiting for its reconfiguration to finish and executing it in hardware, then the module will be executed in software by the middleware (as described in Sec. III-D), and the gain is zero. If we denote the probability mass function (pmf) of G_{nm} with g_{nm}, then the average gain $\overline{G}_{nm}$ will be computed as: $\overline{G}_{nm} = \sum_{x=0}^{\infty} \big(x \cdot g_{nm}(x)\big)$ (see lines 7-8).

The discussion is illustrated in Fig. 3, considering the nodes $n = r$ and $m = m_1$ from Fig. 2a. The probability mass function (pmf) for X_{rm_1} (distance in time from r to m_1) is represented in Fig. 3a and the pmf for the waiting time W_{rm_1} in

Algorithm 3 Computing the execution time distribution

```
1:  procedure ExecTimeDist($n$, $m$)
2:      $ex_n \leftarrow Exec(n)$
3:      if $n$ has 0 children in $FCDT$ then
4:          $x(ex_n) \leftarrow 100\%$
5:      else if $n.type = root$ then
6:          $x \leftarrow ex_n * FCDTChildrenDist(n, m, l)$
7:      else if $n.type = control$ then                         ▷ 'if' blocks
8:          $(t, f) \leftarrow GetLabels(n)$                     ▷ branch frequencies
9:          $ex_t \leftarrow t \times \big(ex_n * FCDTChildrenDist(n, m, t)\big)$
10:         $ex_f \leftarrow f \times \big(ex_n * FCDTChildrenDist(n, m, f)\big)$
11:         $x \leftarrow ex_t + ex_f$
12:     else if $n.type = loop\,header$ then
13:         $ex_{li} \leftarrow ex_n * FCDTChildrenDist(n, m, l)$
14:         $Truncate(ex_{li}, rec(m))$
15:         for all $i \in Iterations(n)$ do
16:             $ex_i \leftarrow iter\_prob_n(i) \times \big[(*^i)ex_{li}\big]$
17:             $Truncate(ex_i, rec(m))$
18:             $ex_{lb} \leftarrow ex_{lb} + ex_i$                 ▷ the loop body
19:             if $\min\{y \mid ex_i(y) \neq 0\} \geq rec(m)$ then
20:                 $break$                                     ▷ no point to continue
21:         $x \leftarrow ex_n * ex_{lb}$                        ▷ header executed last time
22:     $Truncate(x, rec(m))$
23:     return x
```

Fig. 3b. Note that the negative part of the distribution (depicted with dotted line) generates no waiting time. In Fig. 3c we add the hardware execution time to the potential waiting time incurred. Finally, Fig. 3d represents the discrete probability distribution of the gain G_{rm_1}. The resulting average gain is $\overline{G}_{rm_1} = 34 \cdot 18\% + 39 \cdot 42\% + 44 \cdot 6\% + 45 \cdot 34\% = 40.44$ time units.

C. Computing the Execution Time Distribution X_{nm}

Algorithm 3 illustrates our method for computing the execution time distribution between node n and module m (for a more detailed description please refer to [9]). We remind the reader that all the computation is done considering the subgraph containing only the nodes between n and m and its forward control dependence tree (FCDT) [1]. Also, before applying the algorithm we transform all post-test loops into pre-test ones (this transformation is done on the CFG representation, for analysis purposes only). Our approach is to compute the execution time distribution of node n and all its children in the FCDT, using the recursive procedure $ExecTimeDist(n, m)$. If n has no children in the FCDT (i.e. no nodes control dependent on it), then we simply return its own execution time (line 4). For the root node we convolute its execution time with the execution time distribution of all its children in the FCDT (line 6). This is done because the probability distribution of a sum of two random variables is obtained as the convolution of the individual distributions.

For a control node, we compute the execution time distribution for all its children in the FCDG that are control dependent on the 'true' branch, convolute this with the execution time of n and scale the distribution with the probability of the 'true' branch, t (line 9). Similarly, we compute the distribution for the 'false' branch as well (line 10) and then we superpose the two distributions to get the final one (line 11). For example, for the branch node c in Fig. 2a, we have $ex_t(2 + 3) = 30\%, ex_f(2 + 8) = 70\%$ and, thus, the pmf for the execution time of the entire if-then-else structure is $x(5) = 30\%$ and $x(10) = 70\%$.

Finally, for a loop header, the method computes the execution time distribution of one iteration through the loop, ex_{li} (line 13). Then it uses the distribution of loop iterations ($iter\_prob_n$) to convolute the execution time distribution of the loop body with itself ($(*^i)$ denotes the operation of convolution with itself i times) as many times as indicated by $iter\_prob_n$ (lines 15-18). Let us illustrate the computation for the loop $a{-}b$ in our example from Fig. 2a. Since in this case b is the only node inside the loop, $ex_{li} = ex_a * ex_b$ gives the probability mass function (pmf) $ex_{li}(1 + 4) = 100\%$. Then we convolute ex_{li} with itself two times, and we scale the result with the probability to iterate twice through the loop, $iter\_prob_a(2) = 60\%$, obtaining $ex_2(10) = 60\%$. Similarly, $ex_4(20) = 20\%$ and $ex_5(25) = 20\%$. By superposing ex_2, ex_4 and ex_5 we get the pmf for the loop body, ex_{lb}, which we finally have to convolute with ex_a to get the pmf of the entire loop: $x(10 + 1) = 60\%, x(20+1) = 20\%$ and $x(25+1) = 20\%$. This distribution can be further used in the computation of $\overline{G}_{rm_1}$, for example.

The procedure $FCDTChildrenDist(n, m, label)$ simply convolutes the distributions of all children of n in the FCDT that are control dependent on the parameter edge $label$. In order to speed-up the computation, when computing the pmf for the execution time distributions, we discard any values that are greater or equal than the reconfiguration time of m, because those components of the distribution will generate no waiting time. Procedure $Truncate$ works as follows: if the smallest execution time is already greater than the reconfiguration overhead, we keep only this smallest value in the distribution. This is done because the distribution in question might be involved in convolutions or superpositions (in procedure $ExecTimeDist$), and keeping only this minimal value is enough for computing the part of the execution time distribution of interest (that might generate waiting time). Otherwise, we simply truncate the distribution at $rec(m)$.

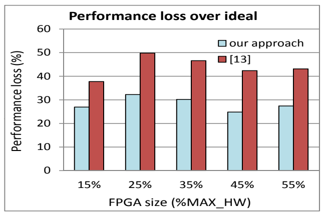
One observation related to the computation of the execution time of a node $m \in \mathcal{N}_{cf}$ (procedure $Exec$ in Algorithm 3, line 2) is that, if m is a hardware candidate ($m \in \mathcal{H}$) we need to approximate its execution time, since the prefetches for it might be yet undecided and, thus, it is not known if the module will be executed in software or on the FPGA. In order to estimate the execution time, we make use of a coefficient $\alpha_m \in [0, 1]$. The execution time for a hardware module m will be computed as: $exec(m) = hw(m) + \alpha_m \cdot (sw(m) - hw(m))$. Our experiments have proven that very good results are obtained by setting the value of α_m, for each hardware module, to the ratio between its own hardware area and the total area needed for all modules in $\mathcal{H}$: $\alpha_m = \frac{area(m)}{\sum_{k \in \mathcal{H}} area(k)}$.

VII. Experimental Results

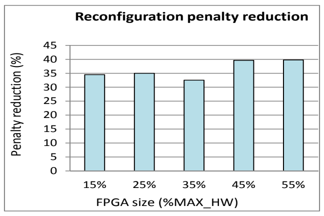
A. Synthetic Examples

In order to evaluate the effectiveness of our algorithm we first performed experiments on synthetic examples. We generated two sets of control flow graphs: Set_1 contains 20 CFGs with $\sim$ 100 nodes on average (between 67 and 126) and Set_2 contains 20 CFGs with $\sim$ 200 nodes on average (between 142 and 268).

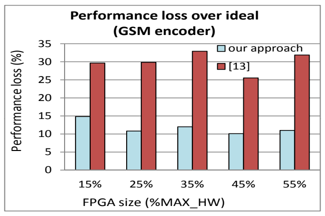
The software execution time for each node was randomly generated in the range of 10 to 100 time units. A fraction of all the nodes (between 15% and 25%) were then chosen to become hardware candidates, and their software execution time was generated β times bigger than their hardware execution time. The coefficient β was chosen from the uniform distribution on the interval [3, 7], in order to model the variability of hardware speedups over software. We also generated the size



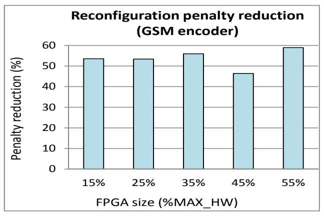
(a) Set_2



(b) Set_2



(c) GSM Encoder



(d) GSM Encoder

Figure 4: Comparison with State-of-Art [13]: Synthetic Benchmarks and Case Study - GSM Encoder

of the hardware modules, which in turn determined their reconfiguration time.

The size of the PDR region available for placement of hardware modules was varied as follows: we summed up all the areas for all hardware modules of a certain application: $MAX\_HW = \sum_{m \in \mathcal{H}} area(m)$. Then we generated problem instances by considering the size of the available reconfigurable region corresponding to different fractions of MAX_HW: 15%, 25%, 35%, 45%, and 55%. As a result, we obtained a total of $2 \times 20 \times 5 = 200$ experimental settings. All experiments were run on a PC with CPU frequency 2.83 GHz, 8 GB of RAM, and running Windows Vista.

For each experimental setting, we first generated a placement for all the hardware modules, which determined the area conflict relationship between them. Then, for each application we inserted the configuration prefetches in the control flow graph. Finally, we have evaluated the result using the simulator described in [9] that produces the average execution time of the application considering the architectural assumptions described in Sec. III-A, III-C and III-D. We have determined the result with an accuracy of $\pm 1\%$ with confidence 99.9% [9].

Due to space constraints, in what follows we present only the results for Set_2, and those for Set_1 can be found in [9]. As a baseline we have considered the average execution time of the application (denoted as $baseline$) in case all the hardware candidates are placed on FPGA from the beginning and, thus, no prefetch is necessary. Please note that this is an absolute lower bound on the execution time; this ideal value might be unachievable even by the optimal static prefetch, because it might happen that it is impossible to hide all the reconfiguration overhead for a particular application.

First of all we were interested to see how our approach compares to the current state-of-art [13]. Thus, we have simulated each application using the prefetch queues generated by our approach and those generated by [13]. Let us denote the average execution times obtained with ex_G for our approach, and ex_{PAP} for [13]. Then we computed the performance loss over the baseline for our approach, $PL_G = \frac{ex_G - baseline}{baseline}$; similarly we calculate PL_{PAP}. Fig. 4a shows the results obtained (averaged over all CFGs in Set_2). As can be seen, for all FPGA sizes, our approach achieves better results compared to [13]: for Set_2, the performance loss over ideal is between 25% and 31.5% for our method, while for [13] it is between 38% and 50% (Fig. 4a). In other words, we are between 28% and 41% closer to the ideal baseline than [13].

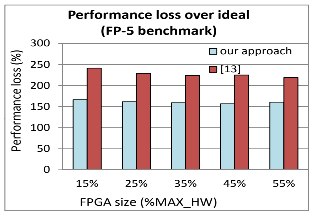
One other metric suited to evaluate prefetch policies is the total time spent by the application waiting for FPGA reconfigurations to finish (in case the reconfiguration overhead was not entirely hidden). One major difference between the approach proposed in this paper and that in [13] is that we also execute candidates from $\mathcal{H}$ in software (if this is more profitable than reconfiguring and executing on FPGA), while under the assumptions in [13] all candidates from $\mathcal{H}$ are executed only on FPGA. Considering this, for each execution in software of a candidate $m \in \mathcal{H}$, we have no waiting time, but we do not execute m on FPGA either. In this cases, in order to make the comparison to [13] fair, we penalize our approach with $sw(m) - hw(m)$. Let us define the reconfiguration penalty (RP): for [13] RP_{PAP} is the sum of all waiting times incurred during simulation, and for our approach RP_G is the sum of all waiting times plus the sum of penalties $sw(m) - hw(m)$ whenever a module $m \in \mathcal{H}$ is executed in software during simulation. Fig. 4b shows the reconfiguration penalty reduction $RPR = \frac{RP_{PAP} - RP_G}{RP_{PAP}}$, averaged over all CFGs in Set_2. As we can see, by intelligently generating the prefetches we manage to significantly reduce the penalty (with up to 40%), compared to [13].

Concerning the running times of the heuristics, our approach took longer time than [13] to generate the prefetches: from just $1.6\times$ longer in the best case, up to $12\times$ longer in the worst case, incurring on average $4.5\times$ more optimization time. For example, for the 15% FPGA fraction, for the biggest CFG in Set_2 (with 268 nodes), the running time of our approach was 3832 seconds, compared to 813 seconds for [13]; for CFGs with a smaller size and a less complex structure we generated a solution in as low as 6 seconds (vs 2 seconds for [13]).

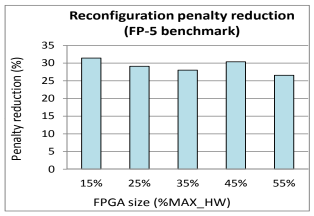
B. *Case Study - GSM Encoder*

We also tested our approach on a GSM encoder, which implements the European GSM 06.10 provisional standard for full-rate speech transcoding. This application can be decomposed into 10 functions executed in a sequential order: Init, GetAudioInput, Preprocess, LPC_Analysis, ShortTermAnalysisFilter, LongTermPredictor, RPE_Encoding, Add, Encode, Output. The execution times were derived using the MPARM cycle accurate simulator, considering an ARM processor with an operational frequency of 60 MHz. We have identified through profiling the most computation intensive parts of the application, and then these parts were synthesized as hardware modules for an XC5VLX50 Virtex-5 device, using the Xilinx ISE WebPack. The resulting overall CFG of the application contains 30 nodes, including 5 hardware candidates (for the CFG and another experimental setting with 9 hardware candidates see [9]). The reconfiguration times were estimated considering a 60 MHz configuration clock frequency and the ICAP 32-bit width configuration interface. Profiling was run considering several audio files (.au) as input.

We have used the same methodology as for the synthetic examples and compared the results using the same metrics defined above in Sec. VII-A, i.e. performance loss over ideal and reconfiguration penalty reduction (presented in Fig. 4c



(a) 25 HW candidates



(b) 25 HW candidates

Figure 5: Comparison with State-of-Art [13]: Case Study - Floating Point Benchmark

and d). As can be seen, the performance loss over ideal is between 10.5% and 14.8% for our approach, while for [13] it is between 25.5% and 32.9% (Fig. 4c). Thus, we were from 50% up to 65% closer to the ideal baseline than [13]. The reconfiguration penalty reduction obtained is as high as 58.9% (Fig. 4d). The prefetches were generated in 27 seconds by our approach and in 11 seconds by [13].

C. Case Study - Floating Point Benchmark

Our second case study was a SPECfp benchmark (FP-5 from [22]), characteristic for scientific and computation-intensive applications. Modern FPGAs, coupled with floating-point tools and IP, provide performance levels much higher than software-only solutions for such applications [23]. In order to obtain the inputs needed for our experiments, we used the framework and traces provided for the first Championship Branch Prediction competition [22]. The given instruction trace consists of 30 million instructions, obtained by profiling the program with representative inputs.

We have used the provided framework to reconstruct the control flow graph (CFG) of the FP-5 application based on the given trace. We have obtained a CFG with 65 nodes, after inlining the functions and pruning all control flow edges with a probability lower than 10^{-5} (the resulting CFG is available in [9]). Then we used the traces to identify the parts of the CFG that have a high execution time (mainly loops).

The software execution times for the basic blocks were obtained by considering the following cycles per instruction (CPI) values for each instruction: for calls, returns and floating point instructions CPI = 3, for load, store and branch instructions CPI = 2, and for other instructions CPI = 1. Similar to the previous experimental sections, we considered the hardware execution time β times smaller than the software one, where β was chosen from the uniform distribution on the interval [3, 7].

We have considered as hardware candidates the top 25 nodes with the highest software execution times (another scenario considering the top 9 nodes is available in [9]). Following the same methodology as described in Sec. VII-A, we compared our approach with [13]. The results are presented in Fig. 5. The performance loss over ideal is between 152% and 160% for our approach, while for [13] it is between 220% and 245% (Fig. 5a). Thus, we are from 32% up to 35% closer to the ideal baseline than [13]. The reconfiguration penalty reduction is as high as 31% (Fig. 5b). The prefetches for FP-5 were generated in 53 seconds by our approach and in 32 seconds by [13].

VIII. Conclusion

In this paper we presented a speculative approach to prefetching for FPGA reconfigurations. We use profiling information to compute the probabilities to reach each candidate module from every node in the CFG, as well as the distributions of execution time gain obtained by starting a certain prefetch at a certain node. Then we statically schedule the appropriate prefetches (and implicitly do HW/SW partitioning of the candidate hardware modules) such that the expected execution time of the application is minimized. One direction of future work is to develop dynamic prefetching algorithms, that would also capture correlations, for the case when the profile information is either unavailable, or inaccurate.

References

[1] A. V. Aho *et al.*, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.
[2] S. Banerjee *et al.*, “Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration,” *Design Automation Conference*, 2005.
[3] R. Cordone *et al.*, “Partitioning and scheduling of task graphs on partially dynamically reconfigurable FPGAs,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 28(5), 2009.
[4] S. Hauck, “Configuration prefetch for single context reconfigurable coprocessors,” *Intl. Symp. on FPGAs*, 1998.
[5] K. Jiang *et al.*, “Co-design techniques for distributed real-time embedded systems with communication security constraints,” *Design, Automation, and Test in Europe*, 2012.
[6] Z. Li, “Configuration management techniques for reconfigurable computing,” 2002, PhD thesis, Northwestern Univ., Evanston,IL.
[7] Z. Li and S. Hauck, “Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation,” *Intl. Symp. on FPGAs*, 2002.
[8] A. Lifa *et al.*, “Performance optimization of error detection based on speculative reconfiguration,” *Design Automation Conference*, 2011.
[9] ——, “Execution time minimization based on hardware/software partitioning and speculative prefetch,” *Technical reports in Computer and Information Science, ISSN 1654-7233*, 2012.
[10] E. M. Panainte *et al.*, “Instruction scheduling for dynamic hardware configurations,” *DATE*, 2005.
[11] ——, “Interprocedural compiler optimization for partial run-time reconfiguration,” *J. VLSI Signal Process.*, 43(2), 2006.
[12] M. Platzner, J. Teich, and N. Wehn, Eds., *Dynamically Reconfigurable Systems*. Springer, 2010.
[13] J. E. Sim *et al.*, “Interprocedural placement-aware configuration prefetching for FPGA-based systems,” *IEEE Symp. on Field-Programmable Custom Computing Machines*, 2010.
[14] J. E. Sim, “Hardware-software codesign for run-time reconfigurable FPGA-based systems,” 2010, PhD thesis, National Univ. of Singapore.
[15] J. Bispo *et al.*, “From instruction traces to specialized reconfigurable arrays,” *ReConFig*, 2011.
[16] P.-A. Hsiung *et al.*, “Scheduling and placement of hardware/ software real-time relocatable tasks in dynamically partially reconfigurable systems,” *ACM Trans. Reconfigurable Technol. Syst.*, 4(1), 2010.
[17] M. Koester *et al.*, “Design optimizations for tiled partially reconfigurable systems,” *IEEE Trans. VLSI Syst.*, 19(6), 2011.
[18] F. Redaelli *et al.*, “An ILP formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures,” *ReConFig*, 2008.
[19] J. Resano *et al.*, “A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware,” *DATE*, 2005.
[20] P. Sedcole *et al.*, “Modular dynamic reconfiguration in Virtex FPGAs,” *IET Comput. Digit. Tech.*, 153(3), 2006.
[21] Y. Yankova *et al.*, “DWARV: Delftworkbench automated reconfigurable VHDL generator,” *FPL*, 2007.
[22] “The 1st championship branch prediction competition,” *Journal of Instruction-Level Parallelism*, 2004, http://www.jilp.org/cbp.
[23] Altera, “Taking advantage of advances in FPGA floating-point IP cores - white paper WP01116,” 2009.
[24] Xilinx, “Partial reconfiguration user guide UG702,” 2012.