Context-Aware Speculative Prefetch for Soft Real-Time Applications

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

***Abstract*—Dynamically reconfigurable computing devices have the ability to adapt their hardware to application demands, providing the performance of hardware acceleration, as well as high flexibility, at competitive costs. For these reasons, FPGA-based reconfigurable systems are becoming popular in many application domains, including soft real-time computing. Unfortunately, one of their biggest limitations is the high reconfiguration overhead. One method to overcome this problem is configuration prefetching, which tries to reduce the reconfiguration penalty by preloading modules on the FPGA before they are needed, and overlapping the reconfiguration with useful computation. In this paper we present a speculative approach to context-aware inter-procedural configuration prefetching that provides statistical guarantees by minimizing the α-percentile of the execution time distribution of a soft real-time application. Our method uses profile information and takes into account the calling context of a procedure in order to generate better prefetch solutions. We also propose a middleware needed to apply the context-dependent prefetches at run-time. Our experiments show that the developed algorithm outperforms the previous state-of-art.**

I. Introduction

In recent years, FPGA-based reconfigurable systems have been employed for a large class of applications, because they provide both the performance of hardware acceleration, and high flexibility and adaptability [13]. Modern FPGAs provide support for partial dynamic reconfiguration [19], which means that parts of the FPGA may be reconfigured at run-time, while other parts remain fully functional. Unfortunately, one of their main drawbacks is the high reconfiguration overhead.

One technique to cope with this problem is configuration prefetching[1] [4], [7], [11], [12], [17], [8]. The main idea is to preload configurations on FPGA before they are needed, overlapping as much as possible the reconfiguration overhead with useful computation. All of the previous work on configuration prefetching tries to minimize either the average execution time of an application (statistical *mean*), or the case that happens most frequently (statistical *mode*). For a large class of applications (e.g., soft real-time, multimedia) it is not enough to have a short execution time on average, or in the most probable cases, but instead it is desirable to have some statistical guarantees on the execution time.

In this paper we propose a speculative approach to inter-procedural configuration prefetching that minimizes the α-percentile of the execution time distribution of a soft real-time application. Even in the simpler cases when the mean or the mode of the execution time are minimized, the high reconfiguration overheads make 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, intelligent prefetch scheduling is the key to high performance of such systems, especially when statistical guarantees are required.

[1]Configuration compression and caching [6] are two complementary techniques that can be used in conjunction with prefetching, but they are not explicitly addressed in this paper.

II. Previous Work

One line of previous work related to ours was done in the area of partitioning and statically scheduling task graphs for FPGA-based architectures. Several articles proposed either exact solutions (based on integer linear programming), or heuristic static approaches to solve the problem [3], [2]. The authors of [14] present a hybrid design/run-time prefetch scheduling heuristic that prepares several schedules (corresponding to different scenarios) at design-time, and then, at run-time it chooses one of them based on the actual conditions. There are two main differences between such solutions and the one we are proposing in this article: The first difference is that all 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 second difference is that none of these papers offer any statistical guarantees and, thus, they are not appropriate for soft real-time systems.

The author of [6] proposes a dynamic heuristic to minimize the execution time, based on a markov predictor, that performs all the prefetch scheduling computations at run-time. The same author also developed a hybrid heuristic, that aims at complementing the run-time predictor with static profile information for the target application. One disadvantage of these methods is that they use only probabilities to guide prefetching.

Unlike the dynamic approaches mentioned above, static prefetching has one major advantage: It requires no additional special-purpose hardware and it generates minimal run-time overhead. Moreover, the solutions generated are good as long as the profile information known at compile time is accurate.

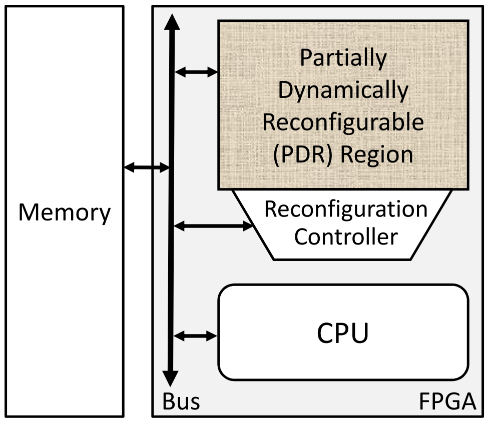
To our knowledge, the works most closely related to the one we present in this paper are [11], [12], [7], [17] and [8].

Panainte et al. proposed both an intra-procedural [11] and an inter-procedural [12] static approach to prefetch scheduling, taking into account FPGA area placement conflicts. In order to hide the reconfiguration overhead they try to anticipate the hardware reconfigurations, either in the intra-procedural control flow graph [11], or in the inter-procedural call graph [12]. Based on data-flow analysis, 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 solution is too conservative and a more aggressive speculation could hide more reconfiguration overhead. Also, since the main goal of the approach is to minimize the number of reconfigurations, execution time minimization is not explicitly modeled in the optimization goal (it is only indirectly modeled as they try to overlap the reconfiguration with useful computation).

Based on the pioneering work of Hauck [4] in configuration prefetching, Li et al. proposed in [7] a method to compute the probabilities to reach any hardware module in an application, based on profiling information. Their algorithm is applied on the control flow graph, but only after all the loops are identified and collapsed into dummy nodes. The probabilities to reach the hardware modules are used to rank them, and prefetches are issued at each basic block based on this ranking. As a result, the execution time of the most frequent trace through the application is minimized (the statistical *mode*). One limitation of this method is that it works on the intra-procedural control flow graph and it removes all loops, which leads to loss of path information. Also, since this approach was developed for FPGAs with relocation and defragmentation, it does not account for placement conflicts between modules.

In [17], Sim et al. propose an algorithm for static configuration prefetching for partially reconfigurable FPGAs. Their goal is to minimize the overall execution time of an application, taking into account FPGA area placement conflicts. They work on the inter-procedural control flow graph and compute, based on profiling information, 'placement-aware' probabilities (PAPs). They represent the probability to reach a hardware module from a certain basic block without encountering any conflicting hardware modules on the way. Similar to [7], the authors use the 'placement-aware' probabilities to rank the modules and generate prefetch queues, which are inserted by the compiler in the inter-procedural control flow graph of the application. One limitation of this work (also common for the approaches mentioned above), is that it uses only probability information to guide prefetch scheduling. As it was shown in [8], it is possible to generate better prefetches (and, thus, further reduce the execution time of the application) if the execution time distributions, correlated with the reconfiguration time of each hardware module, are also taken into account.

In [8], the authors proposed 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. In order to choose the configurations that will provide the highest performance improvement, the method also considers the execution time distributions, beside probabilities. The hardware candidates are ranked according to a cost function at each node in the control flow graph of the application. The cost function tries to estimate the overall impact of a certain prefetch decision on the average execution time.



Memory
Partially Dynamically Reconfigurable (PDR) Region
Reconfiguration Controller
CPU
Bus
FPGA

Fig. 1: Architecture Model

One drawback, common to all the related works mentioned in this section, is that none of them offers any guarantees on the execution time. In other words, since their goal is to minimize either the average execution time (statistical *mean*), or the most common case (statistical *mode*), or they do not even model the execution time minimization as an optimization goal, they are not appropriate in the case of soft real-time applications. Such applications require statistical guarantees on their execution time, and in this paper we will present an inter-procedural approach that tries to minimize the α-percentile of an application's execution time distribution.

III. Contributions

The main contributions of this paper are the following:

- We propose a static inter-procedural configuration prefetching technique that provides statistical guarantees on the execution time of soft real-time applications.
- We develop a method to take into account the calling context of a procedure in the process of designing the configuration prefetch, in order to obtain better solutions.
- We present a new middleware that is needed in order to apply our context-aware inter-procedural prefetch technique at run-time.

IV. System Model

A. Architecture Model

Architectures that use FPGAs as hardware accelerators (coprocessors) become more and more common in modern systems. Due to the recent advances in FPGA technology, one scenario often employed is the following (illustrated in Fig. 1): The FPGA is partitioned into a static region and a partially dynamically reconfigurable (PDR) region. On the static part, a microprocessor is instantiated, along with a reconfiguration controller (that takes care of reconfiguring the PDR region), and other application modules that need not change at run-time (if any). The PDR region is usually partitioned into slots (the granularity varies) where the application's hardware modules will reside. The reconfiguration controller can load one single module at a time on the PDR region, but the execution of the modules not affected by the current reconfiguration can go on in parallel. This is a realistic model that supports modern FPGAs (like, e.g., the Xilinx Virtex or Altera Stratix families).

Given such an architecture and an application to run on it, one solution would be to partition the application into a software part (to be executed by the CPU) and a number of hardware modules (that are computationally expensive in software and are suitable for hardware implementation). Since modern embedded systems have tight cost constraints, it is reasonable to assume that the PDR region on the FPGA is limited and cannot host all the application modules simultaneously. Nevertheless, since the application does not need all its hardware modules at the same time, it is interesting to consider reusing the FPGA area and dynamically loading at run-time the modules that the application currently needs. The reconfiguration controller will configure the PDR region by loading the bitstreams from the memory, upon CPU requests.

We model the PDR region as a rectangular matrix of configurable logic blocks. Each hardware module occupies a contiguous rectangular area of this matrix. 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 [16]. Thus, similar to the assumptions of Panainte [11] and Sim [17], 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

We model our application as a program composed of several procedures. We consider only *structured* programs [1], and since any non-structured program is equivalent to some structured one, our assumption loses no generality. Each procedure is modeled as a control flow graph (CFG) that captures all potential execution paths and contains two distinguished nodes, *entry* and *exit*, corresponding to the entry and the exit of the procedure[2]. The entire program is modeled as an inter-procedural control flow graph (ICFG) [18], $\mathcal{G}_{icf}(\mathcal{N}_{icf}, \mathcal{E}_{icf})$, and we augment it with two special nodes, *root* and *sink*, corresponding to the single entry and the single exit of the program[3]. The set of nodes in the graph is defined as $\mathcal{N}_{icf} = \mathcal{C} \cup \mathcal{I} \cup \mathcal{O} \cup \mathcal{R} \cup \mathcal{S} \cup \mathcal{H}$, where $\mathcal{C}$ represents the call sites, $\mathcal{I}$ the procedure entry nodes, $\mathcal{O}$ the procedure exit nodes, $\mathcal{R}$ the nodes where control returns after procedure calls (there exists one return node corresponding to each call site), $\mathcal{S}$ the regular basic blocks (defined as straight-line sequences of instructions) and $\mathcal{H}$ represents the set of hardware candidates (to be potentially executed on the FPGA). The set of edges in the ICFG, defined as $\mathcal{E}_{icf} = \mathcal{Y} \cup \mathcal{X} \cup \mathcal{E}$, corresponds to the possible flow of control within the program. We distinguish between procedure entry edges ($\mathcal{Y}$), procedure exit edges ($\mathcal{X}$) and regular edges ($\mathcal{E}$).

The function $prob : \mathcal{E} \cup \mathcal{Y} \rightarrow [0, 1]$ represents the probability of each regular and procedure entry edge in the ICFG to be taken, and it is obtained by profiling the application. Please note that procedure exit edges have probability one. In case call sites have only one single outgoing edge, then the corresponding procedure entry edges will also have probability one. Nevertheless, if pointers to functions are used, then a call site can have several outgoing edges with different probabilities (that together sum up to one).

We denote the set of hardware candidates with $\mathcal{H} \subset \mathcal{N}_{icf}$ 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 ICFG). 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). The set $\mathcal{H}$ can be determined automatically (or by the designer) and might contain, for example, the computation intensive parts of the application, identified after profiling.

For each loop header n we denote with $iter\_prob_n : \mathbb{N} \rightarrow [0, 1]$ the probability mass function of the discrete distribution of loop iterations. For each node $n \in \mathcal{N}_{icf}$ we assume that we know its software execution time, given by the function $sw : \mathcal{N}_{icf} \rightarrow \mathbb{R}^+$. For each hardware candidate $m \in \mathcal{H}$, we also know its hardware execution time, given by the function $hw : \mathcal{H} \rightarrow \mathbb{R}^+$. 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 all hardware candidates are synthesized and placed on the FPGA at design time, we also know their reconfiguration time, given by the function $rec : \mathcal{H} \rightarrow \mathbb{R}^+$.

C. Reconfiguration Support

The architecture described in Sec. IV-A supports preemption and resumption of hardware reconfigurations. In order to enable the software control of the FPGA configurations, we adopt the reconfiguration library described in [16]. The library defines an interface to the reconfiguration controller, and contains the following functions to support initialization, preemption and resumption of dynamic reconfigurations:

- $load(m)$: Non-blocking call that requests the controller to start or resume loading the bitstream of module m.
- $currently\_reconfiguring()$: Returns the ID of the 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.

[2]If a procedure contains several return statements, then we add a dummy exit node (with zero execution time), and the corresponding edges such that the single dummy exit node is the successor of all the original return nodes.

[3]In case of multiple exit points, we proceed similar to the case of multiple procedure return statements.

- $execute(m)$: 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 $n \in \mathcal{N}_{icf}$ the hardware modules to be prefetched have been ranked at design time (according to some strategy) and placed in a table with entries of type *context:loadQ(n)*. Here, the label *context* consists of a sequence of c call sites that have led to the current node n. The field *loadQ* contains a list of candidate hardware modules, ordered according to their impact on the α-percentile of the execution time distribution. As we explain below, 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.

In order to perform the context-dependent prefetches at run-time, the middleware manages a call stack CTX. When a call site is reached, it is pushed onto the CTX stack, and on every return edge the corresponding call site is popped off the stack. At every node n that has associated a prefetch table, the middleware will apply the prefetches from the *loadQ* whose *context* label matches the top c call sites registered in the CTX stack.

For a certain *context*, 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 if 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*).

Once a hardware module $m \in \mathcal{H}$ is reached during the execution, the middleware checks whether m is already fully loaded on the FPGA, and in this case it will be executed there. Otherwise, m will be reconfigured, and then executed on the 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.

V. Motivation

Sim et al. propose in [17] a method for inter-procedural prefetch, but their approach has two limitations: First of all, it uses only the 'placement-aware' probabilities (PAPs) to guide configuration prefetching, and secondly, it does not prefetch inside a procedure the hardware modules from outside it, due to the uncertainty of the call context. In [8] it was shown, for the case of minimizing the average execution time, that better results can be obtained by a method that also takes into account the execution time distributions, correlated with the reconfiguration time of each hardware module. In this paper we will extend the previous work to the inter-procedural case of minimizing the α-percentile of the execution time distribution of a soft real-time application. We will improve on the method presented in [17] for computing the 'placement-aware' probabilities in order to make it context-aware, and we will propose a new middleware capable of handling context-dependent prefetches.

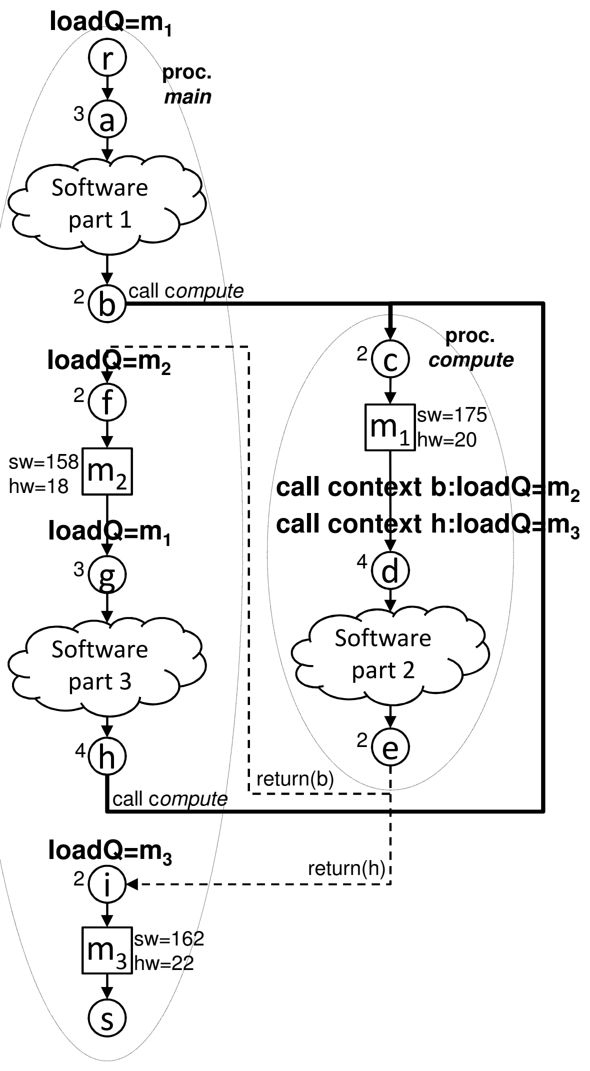
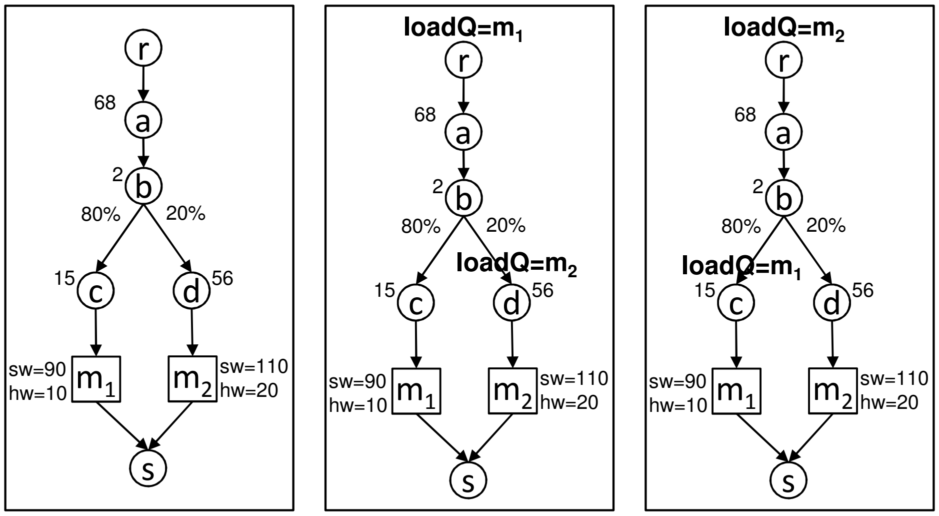


Fig. 2: Motivational Example 1

A. Context Awareness

Let us first illustrate the importance of context-aware inter-procedural configuration prefetching. For this, consider the inter-procedural control flow graph (ICFG) in Fig. 2, where software nodes are represented with circles and candidate hardware modules with rectangles. Call edges are represented with thick lines and return edges with dashed lines. This is a very simple example composed of only two procedures, the *main* program, and procedure *compute*, which is called twice from the *main* procedure. The software and hardware execution times are illustrated on the figure. The reconfiguration times are $rec(m_1) = 184, rec(m_2) = 192, rec(m_3) = 188$, and modules m_1 m_2 and m_3 are all in a placement conflict (i.e. $m_1 \bowtie m_2$, $m_1 \bowtie m_3$ and $m_2 \bowtie m_3$).

If we generate prefetches using the method from [17], the prefetch for m_1 will be anticipated at the root of the ICFG, and its reconfiguration will be overlapped with the computation done in software part 1 (which we assume is



(a) CFG (b) Average Minimized (c) 95-percentile Minimized

Fig. 3: Motivational Example 2

long enough to hide the reconfiguration overhead). Due to the way in which the authors compute the 'placement-aware' probabilities[4] (PAPs), the probability to reach m_2 from any node before m_1 is zero (since m_1 and m_2 are conflicting). Furthermore, the authors will consider that the probability to reach m_2 from inside procedure *compute* is zero (due to the uncertainty of the call context[5]). As a result, module m_2 will be prefetched at node f, and almost no reconfiguration can be hidden for it. As can be seen from the example, if we would account for the call context, we could safely prefetch m_2 at node d, and, thus, overlap all of its reconfiguration overhead with the execution of software part 2. For the second call of procedure *compute*, a similar reasoning applies for module m_3.

Both module m_2 and module m_3 are reachable with nonzero probability from node d. If we do not distinguish the call context at run-time, the best we can do is to prefetch either m_2 or m_3 at node d, thus missing important prefetch opportunities. A context-aware prefetch technique could do better: i.e. prefetch m_2 at node d when procedure *compute* is called from node b, and prefetch m_3 at node d when procedure *compute* is called from node h.

B. Statistical Guarantees

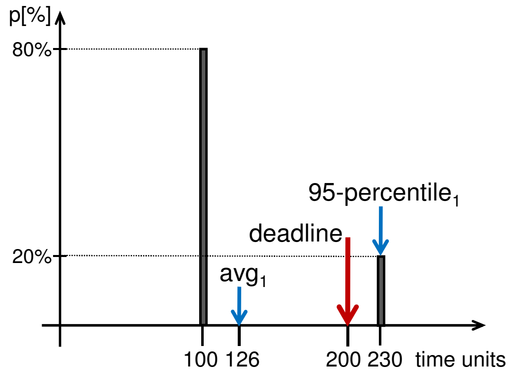
We will now discuss the problem of configuration prefetching in the context of soft real-time systems. To our knowledge, this paper is the first to address this issue. Most of the previous papers focused on minimizing the average execution time of a program [8], [17] or the most common case [7]. Unfortunately, while suitable for the area of high performance computing, these methods are not appropriate for soft real-time applications, since in this case statistical guarantees are required. Minimizing the average execution time might actually lead to more deadline misses for such applications, because of an unwanted stretch in the tail of the execution time distribution. We will illustrate the issue on the example presented in Fig. 3. Let us consider that the control flow graph from Fig. 3a corresponds to a soft real-time application that will provide the required quality of service as long as at least 95% of its executions satisfy a given deadline. In such a case, instead of minimizing the average, we are interested in minimizing the 95-percentile of the execution time distribution, because this will maximize the chances to satisfy the deadline and fulfill the required guarantees.

As in the previous example, the software nodes are represented with circles and the hardware candidates with rectangles. Software and hardware execution times, as well as the edge probabilities, are illustrated on the figure. Reconfiguration times are $rec(m_1) = 90$ and $rec(m_2) = 140$, and the two modules are in placement conflict, $m_1 \bowtie m_2$. The deadline that the application is required to satisfy in at least 95% of executions is 200 time units. For this example, by using either the method described in [17] or the one proposed in [8], the generated prefetches are identical (presented in Fig. 3b): module m_1 is prefetched at node r, and module m_2 at node d. This will generate the execution time distribution shown in Fig. 4a, with an average $avg_1 = 126$ time units, but a 95-percentile of 230 time units (i.e. 95% of executions will complete in 230 time units or less). As can be seen, the deadline is missed in 20% of the cases (when at most a 5% deadline miss rate was allowed).

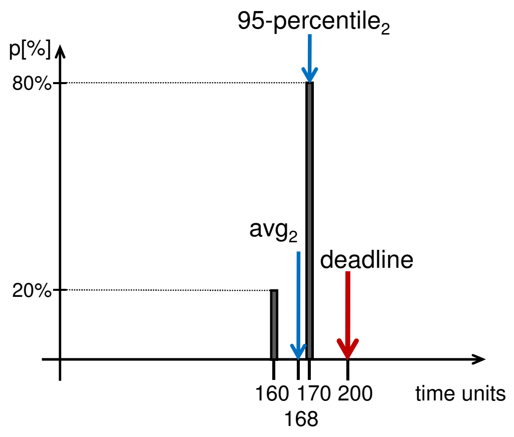
Considering the above discussion, we notice that in this case it is actually better to issue the prefetches illustrated in Fig. 3c: module m_2 at node r and module m_1 at node c. The resulting execution time distribution is shown in Fig. 4b. Although its average is $avg_2 = 168$ time units (bigger than avg_1), the 95-

[4]We remind the reader that these are the probabilities to reach a hardware module from a certain point, without encountering any conflicting modules on the way (whose placement intersects with the module in discussion).

[5]Please note that the reachable modules from inside the procedure *compute* depend on the call context: for the first call, m_2 is reachable, and for the second call, m_3 is reachable.



(a) Average Minimized



(b) 95-percentile Minimized

Fig. 4: Execution Time Distributions

percentile is only 170 time units. In this case, the deadline is always met, and the soft real-time requirements are satisfied.

In this paper we will propose a method to perform speculative inter-procedural configuration prefetch, taking into account the soft real-time requirements of applications.

VI. Problem Formulation

Given a soft real-time application composed of multiple procedures (as described in Sec. IV-B), intended to run on the reconfigurable hardware platform described in Sec. IV-A, our main goal is to minimize the α-percentile of the execution time distribution of the application. In order to do this, we aim to determine, at each node $n \in \mathcal{N}_{icf}$, the context-dependent *loadQ* to be used by the middleware (as described in Sec. IV-D).

VII. Inter-Procedural Configuration Prefetching

As we have shown in the Motivation section (V), it is important to consider the calling context of procedures when performing inter-procedural prefetching. In this paper we will extend the previous work on configuration prefetching [8], in which the main idea is to rank all the reachable hardware candidates from a certain point in the program according to a cost function. This cost function takes into account both the 'placement-aware' probabilities (PAPs) to reach the hardware candidates, as well as the execution time distributions from the prefetch point up to the candidates. The goal is to prefetch those hardware modules that will minimize the α-percentile of the execution time distribution of the application.

A. Soft Real-Time Constraints

As we have mentioned before, we address the real-time constraints by minimizing the α-percentile of the application's execution time distribution. Given a distribution and a number $\alpha(0 \leq \alpha \leq 100)$, the α-percentile of the distribution is defined as the value below which $\alpha\%$ of the observations fall. Thus, for a soft real-time application, the α-percentile of its execution time distribution is the value that gives a statistical guarantee (on the steady-state behavior) that no more than $(100-\alpha)\%$ of the executions will exceed it. Similar approaches to modelling of the soft real-time constraints can be found, for example, in [9] or [15].

Our objective in this paper is to choose the hardware candidates with the highest improvement potential and to schedule their prefetches such that the α-percentile for the obtained execution time distribution is as small as possible. As an example, let us consider a GSM codec application, for which it is necessary that at least 95% of the frames will be decoded before a given deadline, in order to provide a certain quality of service (no major disruptions in the voice signal). In such a case, we will minimize the 95-percentile of the execution time distribution, in order to maximize the chance of meeting the deadline (for at least 95% of the frames) and delivering the required quality of service.

B. Configuration Queues

At any node $n \in \mathcal{N}_{icf}$, we try to intelligently assign priorities to the reachable candidate modules and determine the context-dependent table *context:loadQ(n)* to be used by the middleware (as described in Sec. IV-D). One important observation here is that the set of reachable hardware modules from a certain location inside a procedure depends on the call context of that procedure. As a result, we have a context-dependent *loadQ* at every node n, where the *context* label previously mentioned consists of a sequence of c call sites that have led to node n.

The number of *context* labels at a location is (potentially) exponential in the number of nested call sites for a procedure. For example, if procedure *A* is called from 5 locations inside another procedure *B*, and *B* is itself called 5 times from the *main* program, then the number of *context* labels for the nodes inside procedure *A* is $5 \times 5 = 25$. In order to deal with this problem, we have resorted to two solutions. First of all, our experiments have shown that we can limit the length of the *context* labels to at most $c = 2$ call sites, without sacrificing too much the performance of the prefetch algorithm. The intuition behind this is that it is usually not necessary to start a prefetch too early before a module will be used (but only early enough to overlap its reconfiguration overhead with useful computation). Another empirical observation was that many *context* labels had the same reachable modules with the same

Algorithm 1 Generating context-dependent prefetch queues

```
1: procedure GENERATEPREFETCHQ
2:     for all $n \in \mathcal{N}_{icf}$ do
3:         for all $l \in ContextLabels(n)$ do
4:             for all $\{m \in \mathcal{H} | PAP^l(n,m) \neq 0\}$ do
5:                 $SG_{nm} \leftarrow$ subgraph between $n$ and $m$
6:                 $MG_n \leftarrow MG_n \cup SG_{nm}$      ▷ merge subgraphs
7:             end for
8:             add a $sink$ node to $MG_n$
9:             for all $\{m \in \mathcal{H} | PAP^l(n,m) \neq 0\}$ do
10:                $C^l_{nm} \leftarrow ComputeCostFct(n, m, l, MG_n)$
11:            end for
12:            $l : loadQ(n) \leftarrow$ modules sorted ascending by $C^l_{nm}$
13:            remove modules with area conflicts
14:        end for
15:    end for
16:    eliminate redundant prefetches
17: end procedure
```

probabilities. In this case, it is possible to merge those labels, and, thus, reduce their overall number.

Our overall strategy to generate the prefetch queues is shown in Algorithm 1. Given the reachable modules from node $n \in \mathcal{N}_{icf}$, we want to rank them and prepare the *prefQ* to be used at run-time (line 12). In order to do this, for each *context* label l at node n, we compute the cost function C^l_{nm} (line 10), which tries to estimate at design time what is the impact of reconfiguring a certain module m on the α-percentile of the execution time distribution. After the *context:loadQ(n)* has been generated for a certain node and a certain *context*, we remove all the lower priority modules that have area conflicts with higher priority ones (line 13). Once all the prefetch queues have been generated, we eliminate redundant prefetches, similar to [7], [17] or [8]. This means that all consecutive candidates at a child node that are a starting sub-sequence at all its parents in the ICFG, for the same *context* label, are eliminated (line 16). The exact hardware module to be prefetched will be determined at run-time, as explained in Sec. IV-D.

In order to compute our cost function, we first construct the subgraph (MG_n) containing the nodes between n and all the reachable candidate modules from node n (lines 4-6). In line 4, $PAP^l(n,m)$ represents the 'placement-aware' probability to reach module m from node n, given the context l, without encountering any conflicting hardware modules on the way (the computation of $PAP^l(n,m)$ is discussed in Sec. VII-D). Please note that we add a $sink$ node (line 8) to MG_n, as a successor of all candidate modules m in the subgraphs SG_{nm}; as a result MG_n is a control flow graph with entry node n and exit node $sink$. Next, we want to estimate what is the execution time distribution of this merged subgraph (MG_n). Depending on the candidates that we decide to prefetch, the hardware modules will have different execution times and, thus, the execution time distribution of MG_n will depend on the prefetch decisions. We are interested to estimate the α-percentile of the execution time distribution of MG_n, and we will prefetch those hardware candidates that generate the minimum α-percentile for it.

Algorithm 2 Computing the cost function

```
1: procedure COMPUTECOSTFCT($n$, $m$, $l$, $MG$)
2:     $X_m \leftarrow \begin{cases} sw(m) \ : sw(m) < W_{nm} + hw(m) \\ W_{nm} + hw(m) \ : otherwise \end{cases}$
3:     for all $\{k \in MutEx(m) | PAP^l(n,m) \neq 0\}$ do
4:         $X_k \leftarrow \begin{cases} sw(k) \ : sw(k) < W_{sk} + hw(k) \\ W_{sk} + hw(k) \ : otherwise \end{cases}$
5:     end for
6:     for all $\{k \notin MutEx(m) | PAP^l(n,m) \neq 0\}$ do
7:         $X_k \leftarrow \begin{cases} sw(k) \ : sw(k) < W^k_{nm} + hw(k) \\ W^k_{nm} + hw(k) \ : otherwise \end{cases}$
8:     end for
9:     $fcdt_{MG} \leftarrow$ forward control dependence tree of $MG$
10:    $dist_{nm} \leftarrow$ compute execution time distribution of $MG$,
       based on $fcdt_{MG}$ and using the execution time estimates $X_i$
       for hardware candidate modules
11:    $C^l_{nm} \leftarrow \alpha$-percentile of $dist_{nm}$
12:    return $C^l_{nm}$
13: end procedure
```

C. The Prefetch Cost Function C^l_{nm}

The detailed method to compute the cost function is presented in Algorithm 2. In the above algorithm, X_i represents the random variable associated with the execution time of hardware candidate module i; W_{nm} denotes the random variable expressing the waiting time for module m, given its reconfiguration is started at node n (this is the time when the processor must stall waiting for the reconfiguration of m to finish, before the application could continue its execution with m); $MutEx(m)$ denotes the set of hardware modules that are executed mutually exclusive with m; the index s in W_{sk} represents the node where the paths leading from n to m and k split; and W^k_{nm} represents the random variable expressing the waiting time for module k, given that its reconfiguration is started immediately after the one for m. Sec. VII-E discusses the computation of the waiting time distributions W_{nm}.

As we said earlier, in order to compute the execution time distribution of the merged subgraph MG, we first need to estimate the execution time distributions (represented by X_i) of the hardware candidates, given that reconfiguration for module m is started at node n. For those hardware modules that will be executed mutually exclusive with m we estimate their potential waiting time considering that we start their reconfiguration at the node (denoted with s) where their paths split (lines 3-4). For the modules that are not mutually exclusive with m, we estimate their potential waiting time considering that their reconfiguration starts immediately after the one for m finished (lines 6-7). Then, given these waiting time distributions, we compute the execution time distributions by adding the waiting times with the hardware execution times ($hw(m)$); in those cases when the software execution time ($sw(m)$) is smaller, we use that instead (see lines 2, 4 and 7). This is done because sometimes it might be better to execute a candidate in software, when its reconfiguration overhead could not be overlapped with useful computation.

The next step is to construct the forward control dependence tree (FCDT) [1] for the merged subgraph MG (line 9). Then, based on $fcdt_{MG}$ and using the execution time estimates X_i

for hardware candidate modules, we compute the execution time distribution of MG (line 10). In order to do this, we apply the algorithm described in [8]. In summary, the basic idea is to apply a recursive procedure that convolutes the execution time of n with the execution time distributions of all its children in the FCDT. For nodes with no children in the FCDT, the procedure returns the node's execution time. For control nodes, the approach computes the execution time distribution for all the children control dependent on the 'true' branch and scales this distribution with the probability of the 'true' branch. Similarly, it computes the distribution for the 'false' branch and then it superposes the two distributions. Finally, for loop headers, the method computes the execution time distribution of one iteration through the loop. Then it uses the distribution of loop iterations ($iter\_prob_n$) to convolute the execution time distribution of the loop body with itself as many times as indicated by $iter\_prob_n$.

Once the execution time distribution of MG is computed, the cost function C^l_{nm} is the α-percentile of this distribution (line 11). The cost function tries to capture the contribution of the candidate module m, as well as the impact that its reconfiguration will produce on the other reachable modules and, implicitly, on the α-percentile.

D. Computing Context-Dependent Probabilities $PAP^l(n, m)$

The authors of [17] have presented a method to compute the inter-procedural 'placement-aware' probabilities (PAPs), but their approach has an important limitation: Inside a procedure, it does not compute the probabilities to reach the candidates after the procedure exit node (or, to be precise, it considers that these probabilities are zero), due to the uncertainty of the call context. Unfortunately, such an approach has an important negative implication (as we have shown in section V): many prefetch opportunities are lost. One straight-forward way to deal with this issue would be to ignore the call context and to associate the corresponding probabilities to the procedure exit edges. Thus, inside the procedure, we will issue the same prefetches, regardless of the calling context. We consider such an approach to be too naive, and in this paper we propose a new context-aware prefetch technique and a middleware to apply it.

Our approach extends the method presented in [17], which computes the probabilities to reach the hardware candidates using a fixed point iterative algorithm that starts from the hardware nodes. In order to compute context-dependent probabilities ($PAP^l(n, m)$), we annotate each procedure return edge with the corresponding call site. Then, in the iterative process of estimating the probabilities, whenever we consider a return edge, we propagate upwards on this edge only those modules whose *context* label contains less than $c = 2$ call sites. Otherwise, we consider their probability zero. In case a module's probability is propagated from the return node of a procedure to its exit node, we add the annotated call site from the return edge to the *context* label. When we propagate probabilities from procedure entry nodes to call sites, we remove the call site from all the *context* labels. After the fixed point method has stabilized, if there exist nodes in a procedure that have identical reach probabilities for all candidate modules and all the call contexts of the procedure, then we merge all the *context* labels into a single one (containing a "don't care" in place of the procedure's call site).

E. Estimating the Waiting Time Distributions W_{nm}

Let us consider that we want to prefetch a candidate module $m \in \mathcal{H}$ at node $n \in \mathcal{N}_{icf}$. If the reconfiguration of m does not finish before the execution reaches it, then the application might have to wait for the reconfiguration to be completed, before continuing execution with m. We are interested to compute the waiting time distribution (W_{nm}) that might be incurred if the reconfiguration of module m is started at node n.

In order to estimate the waiting time distribution we use the method presented in [8]. The main idea is to first compute the distance (in time) between n and m. We are only interested in that part of the distribution that is smaller than the reconfiguration time of m (because only those components of the distribution will generate waiting time). This computation is performed on the forward control dependence tree [1] of the subgraph containing only the nodes between n and m (the method is briefly summarized in the end of Sec. VII-C and details are available in [8]). Once we have the distribution of execution time (distance) between n and m, it is straightforward to compute the waiting time distribution (W_{nm}) as the difference between the reconfiguration time of m and the distance in time between n and m.

VIII. Experimental Evaluation

A. Monte Carlo Simulation

1) Sampling: In order to evaluate the quality of our prefetch solutions we have used an in-house developed Monte Carlo simulator that produces the execution time distribution of an application considering the architectural assumptions described in Sec. IV-A, IV-C and IV-D. Each simulation generates a trace through the inter-procedural control flow graph, starting at the *root* node, and ending at the *sink* node (and we record the length of these traces). Whenever a branch node is encountered, we perform a Bernoulli draw (based on the probabilities of the outgoing edges) to decide if the branch is taken or not. At loop header nodes we perform random sampling from the discrete distribution of loop iterations ($iter\_prob_n$) to decide how many times to loop.

For control nodes, if correlations between two or more branches are known, then they could be captured through joint probability tables. In such a case, whenever we perform a draw from the marginal Bernoulli distribution for a branch, we can compute the conditional probabilities for all the branches correlated with it, based on the joint probability table. Later in the simulation, when the correlated branches are reached, we do not sample their marginal distribution, but instead we sample their conditional distribution based on the outcome of the first branch.

2) Accuracy Analysis: We stop the Monte Carlo simulation once we reach a satisfactory accuracy for the α-percentile of the execution time distribution. We describe the desired accuracy in the following way: *"The α-percentile of the output distribution should be accurate to within $\pm\epsilon$ with confidence κ"*. The accuracy can be arbitrarily precise at the expense of longer simulation times. We will next present an analysis based on confidence intervals [5], to determine the number of samples to run in order to achieve the required accuracy.

We determine the percentile P_z of the output distribution associated with a value z by determining what fraction of the samples fell at or below z. Let us assume that z is the actual α-percentile of the true output distribution. Then, for the Monte Carlo simulation, the value generated in each run independently has an $\alpha\%$ probability of falling below z: this is a binomial process with probability of success $p = \alpha\%$. Thus, if we have N samples so far and S of those have fallen at or below z, the uncertainty associated with the true percentile we should associate with z is described by the distribution $Beta(S+1, N-S+1)$[5].

For large enough number of samples N, we can use the Normal approximation to the Beta distribution:

$$Beta(S+1, N-S+1) \approx Beta(S, N-S) \approx$$
$$\approx Normal\left(\frac{S}{N}, \sqrt{\frac{S(N-S)}{N^3}}\right) = Normal\left(\widehat{P}_z, \sqrt{\frac{\widehat{P}_z(1-\widehat{P}_z)}{N}}\right)$$

In the above equation $\widehat{P}_z = \frac{S}{N}$ is the best guess estimate for P_z. By considering the required accuracy for our percentile estimate and performing a transformation to the standard Normal distribution (i.e. with mean 0 and standard deviation 1), we can obtain the following relationship [5]:

$$\epsilon = \sqrt{\frac{P_z(1-P_z)}{N}}\Phi^{-1}\left(\frac{1+\kappa}{2}\right)$$

The function $\Phi^{-1}(\bullet)$ is the inverse of the standard Normal cumulative distribution function. By rearranging the terms and considering that we want to achieve at least this accuracy we obtain the minimum value for the number of samples N:

$$N > P_z(1-P_z)\left(\frac{\Phi^{-1}\left(\frac{1+\kappa}{2}\right)}{\epsilon}\right)^2$$

For example, in order to obtain the 99-percentile with an accuracy of $\pm 1\%$ with confidence 99.9%, we need to perform at least $N > 1072$ iterations of the Monte Carlo simulation.

B. Synthetic Examples

To study the improvement that can be achieved by our algorithm, we first used the CFG generator tool from [10] to generate 40 inter-procedural control flow graphs with ~ 200 nodes on average (between 134 and 247). The software execution time for each node was generated randomly from the interval 10 to 100 time units. A fraction of all the nodes (between 15% and 25%) were chosen as hardware candidates, and their software execution time was generated approximately five times bigger than their hardware execution time (similar to the assumptions of [17]). The size and the reconfiguration time for the hardware modules was also generated.

We varied the size of the partially dynamically reconfigurable (PDR) region available for placement of hardware modules, taking fractions of 15%, 35%, and 55% from the total area needed for all the hardware candidates in the application, $MAX\_HW = \sum_{m\in\mathcal{H}} area(m)$. As a result, we obtained a total of $40 \times 3 = 120$ experimental settings.

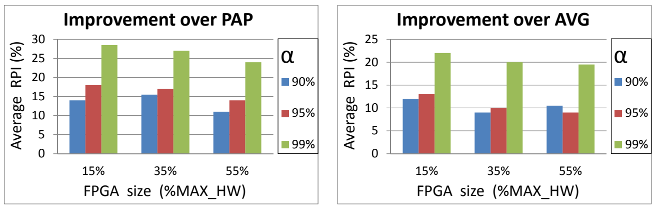
For each experimental setting, we generated a placement for all the hardware modules, which determined the area conflict relationship between them. Then, for each application we have produced prefetches using three methods: The first is the one from [17] (denoted with PAP), that performs inter-procedural placement-aware prefetch for minimizing the average reconfiguration overhead. The second is the method described in [8] (denoted with AVG), that minimizes the average execution time. Finally, we used the method proposed in the current paper (denoted with α-P) to generate prefetches that minimize the α-percentile of the execution time distribution.

For each method, we have performed Monte Carlo simulation (as described in Sec. VIII-A) in order to estimate the α-percentile of the resulting execution time distribution, accurate to within $\pm 1\%$ with confidence 99.9%. We have evaluated three percentiles: $\alpha = 90\%$, $\alpha = 95\%$ and $\alpha = 99\%$. Please note that the methods PAP and AVG generate the same prefetches, regardless of the required statistical guarantees represented by α. Thus, we only perform one Monte Carlo simulation, and then read the different percentiles from the obtained distribution. On the other hand, our α-P algorithm generates prefetches customized to the particular value of α.

Let us denote the values of the obtained α-percentiles for the three methods with Z_{PAP}, Z_{AVG} and $Z_{\alpha-P}$ respectively. We are interested to evaluate the relative percentile improvement (RPI) achieved by our current method compared to the two previous ones. Thus, we compute $RPI_{PAP}^{\alpha-P} = \frac{Z_{PAP}-Z_{\alpha-P}}{Z_{PAP}}$ and, similarly, $RPI_{AVG}^{\alpha-P}$. The vertical axis in Fig. 5 shows the relative improvements averaged over all applications considered. For each hardware fraction (illustrated on the horizontal axis), we present the improvements achieved by α-P over PAP and AVG, considering different α-percentiles.

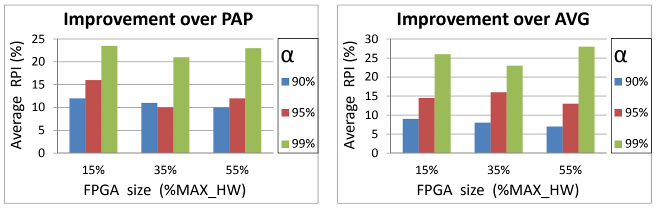
As can be seen, the improvements over PAP are in the range 11%-28%, and those over AVG are between 9%-22%, for different hardware fractions. An interesting observation is that the improvements obtained for high percentiles ($\alpha = 99\%$) are constantly higher than the impovements obtained for lower percentiles. This happens because the methods optimizing for the average execution time (PAP and AVG) easily neglect the far tail of the execution time distribution, while our method (α-P) generates prefetches such that the required α-percentile is minimized. Please note that soft real-time applications typically require high guarantees (i.e. high α) on the execution time. For $\alpha = 99\%$ we obtain significant improvements, of up to 28% over PAP and up to 22% over AVG.

All experiments were run on a Windows Vista PC with CPU frequency 2.83 GHz and 8 GB of RAM. Concerning the running times of our method, it incurred on average 2.2× more optimization time than AVG and 4.1× more than PAP. Nevertheless, the running times are not prohibitive ($\approx$ 1.5 hours in the worst case, for the biggest ICFG) and are justifiable considering the improvements achieved over existing solutions.



(a) Comparison with PAP (b) Comparison with AVG

Fig. 5: Relative Improvement of α-P over PAP and AVG



(a) Comparison with PAP (b) Comparison with AVG

Fig. 6: Relative Percentile Improvement: Case Study

C. Case Study – GSM Decoder

We also tested our approach on the inter-procedural control flow graph (ICFG) derived from a real-life example, a GSM decoder, which implements the European GSM 06.10 provisional standard for full-rate speech transcoding. The application contains 8 procedures: Init, GetAudioInput, Decode, RPE_Decoding, LongTermSynthesis, ShortTermSynthesisFilter, Postprocessing, Output. We estimated the execution times using the MPARM cycle accurate simulator, considering an ARM processor running at 60 MHz. We have identified through profiling the computation intensive parts of the application, which were considered as hardware candidates (to be synthesized for a Virtex-6 device). In order to estimate the reconfiguration times, we considered a 60 MHz clock frequency and the ICAP 32-bit configuration interface.

We have computed $RPI_{PAP}^{\alpha-P}$ and $RPI_{AVG}^{\alpha-P}$ using the same methodology as in Sec. VIII-B. The results are presented in Fig. 6. As can be seen, the improvements are similar to the ones achieved for the synthetic cases. For example, with a hardware fraction of only 15%, our method reduced the 99-percentile with 24% compared to PAP and with 26% compared to AVG. The reduction (of around 15%) for the 95-percentile is satisfactory as well. The running time of our algorithm was 18 minutes in the worst case, compared to 12 minutes for AVG and 7 minutes for PAP.

IX. Conclusion and Future Work

The contribution of this paper is threefold: First of all, to our knowledge, this is the first work that addresses the problem of static inter-procedural configuration prefetching, with the goal to provide statistical guarantees on the execution time of soft real-time applications. Secondly, we have developed a context-aware method to generate the inter-procedural speculative prefetches at design time. Finally, we have proposed a new middleware that is needed to perform the context-aware inter-procedural prefetching at run-time.

In summary, we first compute the context-dependent 'placement-aware' probabilities to reach each hardware module from every node in the inter-procedural control flow graph. Next, we determine the distributions of waiting time incurred by starting a certain prefetch at a node. Then, we use this information in order to statically schedule the appropriate prefetches, such that we minimize the α-percentile of the execution time distribution of a soft real-time application. At run-time, the middleware decides which prefetches to issue based on the actual context.

For future work, we plan to extend the ideas presented here in order to develop dynamic prefetching algorithms, that will also capture correlations. The main advantage of using such techniques would be that the prefetch will not rely on profile information anymore, but the approach would dynamically react and adapt to changes in the application behavior.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.
[2] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "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.*, vol. 28, no. 5, 2009.
[4] S. Hauck, "Configuration prefetch for single context reconfigurable coprocessors," *Intl. Symp. on Field Programmable Gate Arrays*, 1998.
[5] M. Hollander and D. A. Wolfe, *Nonparametric Statistical Methods, 2nd Edition*. Wiley-Interscience, 1999.
[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 Field Programmable Gate Arrays*, 2002.
[8] A. Lifa, P. Eles, and Z. Peng, "Execution time minimization based on hardware/software partitioning and speculative prefetch," *under submission*, 2012.
[9] S. Manolache, P. Eles, and Z. Peng, "Schedulability analysis of applications with stochastic task execution times," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 4, 2004.
[10] C.-F. Neikter, "Cache prediction and execution time analysis on real-time MPSoC," 2008, MSc thesis, Linköping Univ., Sweden.
[11] E. Panainte, K. Bertels, and S. Vassiliadis, "Instruction scheduling for dynamic hardware configurations," *Design, Automation, and Test in Europe*, 2005.
[12] E. Panainte, K. Bertels, and S. Vassiliadis, "Interprocedural compiler optimization for partial run-time reconfiguration," *The Journal of VLSI Signal Processing*, vol. 43, no. 2, 2006.
[13] M. Platzner, J. Teich, and N. Wehn, Eds., *Dynamically Reconfigurable Systems*. Springer, 2010.
[14] J. Resano, D. Mozos, and F. Catthoor, "A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware," *Design, Automation, and Test in Europe*, 2005.
[15] N. R. Satish, K. Ravindran, and K. Keutzer, "Scheduling task dependence graphs with variable task execution times onto heterogeneous multiprocessors," *Intl. Conf. on Embedded Software*, 2008.
[16] J. E. Sim, "Hardware-software codesign for run-time reconfigurable FPGA-based systems," 2010, PhD thesis, National Univ. of Singapore.
[17] J. E. Sim *et al.*, "Interprocedural placement-aware configuration prefetching for FPGA-based systems," *IEEE Symp. on Field-Programmable Custom Computing Machines*, 2010.
[18] S. Sinha, M. J. Harrold, and G. Rothermel, "Interprocedural control dependence," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, 2001.
[19] Xilinx, "Early access partial reconfiguration user guide UG208," 2006.