# Dynamic Configuration Prefetching Based on Piecewise Linear Prediction

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

***Abstract*—Modern systems demand high performance, as well as high degrees of flexibility and adaptability. Many current applications exhibit a dynamic and nonstationary behavior, having certain characteristics in one phase of their execution, that will change as the applications enter new phases, in a manner unpredictable at design-time. In order to meet the performance requirements of such systems, it is important to have on-line optimization algorithms, coupled with adaptive hardware platforms, that together can adjust to the run-time conditions. We propose an optimization technique that minimizes the expected execution time of an application by dynamically scheduling hardware prefetches. We use a piecewise linear predictor in order to capture correlations and predict the hardware modules to be reached. Experiments show that the proposed algorithm outperforms the previous state-of-art in reducing the expected execution time by up to 27% on average.**

## I. Introduction and Related Work

It has been shown that the execution of many modern applications progresses in phases, where each phase may be very different from the others, while still having a homogeneous behavior within a phase [16]. For such systems, it is important to have on-line optimization algorithms, coupled with adaptive hardware platforms, that can adjust to the run-time conditions.

For systems that require both hardware acceleration and high flexibility, FPGA platforms are a popular choice [13]. Recently, manufacturers provided support for partial dynamic reconfiguration [19], i.e. parts of the FPGA may be reconfigured at run-time, while other parts remain fully functional. This flexibility comes with one major impediment: high reconfiguration overheads. Configuration prefetching tries to overcome this limitation by preloading future configurations and overlapping as much as possible of the reconfiguration overhead with useful computation.

This paper proposes a new dynamic optimization approach for configuration prefetching. An on-line piecewise linear predictor is used to capture branch correlations and estimate the likelihood to reach different hardware modules from the prediction point. Timestamps are used to estimate the time to reach a certain module, and based on this the execution time gain resulting from a certain prefetch decision is computed.

Previous work in configuration prefetching can be placed in one of three categories: 1) Static techniques that use profile information in order to decide and fix the prefetches for an application at design-time, and then, regardless of the run-time behavior, the same prefetches are applied. 2) Dynamic techniques that try to take all the decisions at run-time, based on the actual application behavior. 3) Finally, hybrid techniques prepare a set of prefetch schedules off-line, based on profile information, and then they choose on-line which of the schedules to use.

Many static algorithms for prefetching were proposed: [9], [12], [17], [10]. All share one limitation: in case of nonstationary behavior, they are unable to adapt, since the prefetch schedule is fixed based on average profiling information. For applications with inaccurate or unavailable profiling information, and for those that exhibit nonstationary behavior, it is important to have a mechanism that adapts to changes. In [1] and [3] the authors address the problem of scheduling task graphs on reconfigurable architectures. Beside being static (since all decisions are taken at design-time), these works have another disadvantage compared to our work: they do not consider the control flow, thus missing many prefetch opportunities.

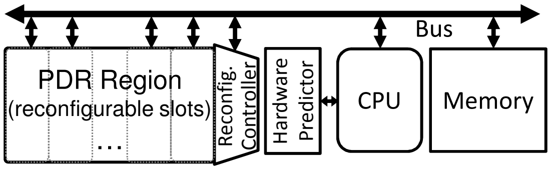


Fig. 1: General Architecture Model

The authors of [11] and [14] present hybrid heuristics that identify a set of possible application configurations at design-time. At run-time, a resource manager chooses among them. These approaches provide more flexibility than static ones, but they are still limited by relying on off-line information.

In [6], the authors propose an on-line algorithm that manages coprocessor loading by maintaining an aggregate gain table for all hardware candidates. For each run of a candidate, the performance gain resulted from a hardware execution over a software one is added to the corresponding entry in the table. When a coprocessor is considered for reconfiguration, the algorithm only loads it when the aggregate gain exceeds the reconfiguration overhead. One limitation of this work is that it does not perform prefetching, i.e it does not overlap the reconfiguration overhead with useful computation. Another difference between all the works presented and ours is that none of the above papers explicitly consider the control flow in their application model. Furthermore, they also ignore correlations.

The authors of [9] propose a dynamic prefetch heuristic that represents hardware modules as the vertices in a Markov graph. Transitions are updated based on the modules executed at run-time. Then, a weighted probability in which recent accesses are given higher weight is used as a metric for candidate selection and prefetch order. The main limitations of this work are that it uses only the weighted probabilities for issuing prefetches (ignoring other factors as, e.g., the execution time gain resulting from different prefetch decisions), and that it uses a history of length 1 (i.e. it predicts only the next module to be executed based on the current module, consequently completely ignoring correlations).

## II. System Model

### A. Architecture Model and Reconfiguration Support

Fig. 1 shows our target architecture composed of a host microprocessor (CPU), a memory subsystem, a reconfigurable FPGA area (used as a coprocessor for hardware acceleration), a reconfiguration controller and a hardware predictor. One common scenario is to partition the FPGA into a static region, where the CPU and the reconfiguration controller reside, and a partially dynamically reconfigurable (PDR) region, where the application hardware modules can be loaded at run-time [4], [15]. The host CPU executes the software part of the application and is also responsible for initiating the reconfiguration of the PDR region. The reconfiguration controller will configure this region by loading the bitstreams from the memory. We assume that the CPU can enqueue modules to be prefetched by the controller, as well as clear this

978-3-9815370-0-0/DATE13/©2013 EDAA

queue. While one reconfiguration is going on, the execution of other (non-overlapping) modules on the FPGA is not affected. The shared memory is used for communication between the CPU and the modules loaded in the PDR region. The predictor module is used to predict the hardware candidates that should be speculatively prefetched on the PDR region (its functionality is described in detail in Sec. V).

We model the PDR region as a matrix of heterogeneous configurable tiles, organized as reconfigurable slots where hardware modules can be placed, similar to [8]. We consider that each hardware candidate has a slot position decided and optimized at design-time using existing methods, such as [5]. This technique will minimize the number of placement conflicts, i.e. candidates with intersecting areas on the reconfigurable region.

### B. Application Model

We model the application as a flow graph $\mathcal{G}(\mathcal{N}, \mathcal{E})$, where nodes in $\mathcal{N}$ correspond to certain computational tasks, and edges in $\mathcal{E}$ model flow of control within the application. We denote with $\mathcal{H} \subseteq \mathcal{N}$ the set of hardware candidates and we assume that all modules in $\mathcal{H}$ have both a hardware and a corresponding software implementation. Since sometimes it might be impossible to hide enough of the reconfiguration overhead for all candidates in $\mathcal{H}$, we decide at run-time which are the most profitable modules to insert prefetches for (at a certain point in the application). The set $\mathcal{H}$ can be determined automatically [2], [20], or by the designer, and typically contains the computation intensive parts of the application.

For each hardware candidate $m \in \mathcal{H}$, we assume that we know its software execution time[1], $sw : \mathcal{H} \to \mathbb{R}^+$, its hardware execution time (including any additional communication overhead between CPU and FPGA), $hw : \mathcal{H} \to \mathbb{R}^+$, and the time to reconfigure it on the FPGA, $rec : \mathcal{H} \to \mathbb{R}^+$. At run-time, once a candidate $m \in \mathcal{H}$ is reached, there are two possibilities: 1) $m$ is already fully loaded on the FPGA, and thus the module will be reused and executed there; 2) $m$ is not fully loaded on the FPGA. Then, we face two scenarios: a) if starting/continuing the reconfiguration of $m$, waiting for it to finish and then executing the module on FPGA results in an earlier finishing time than with software execution, then the application will do so; b) otherwise, the software version of $m$ will be executed.

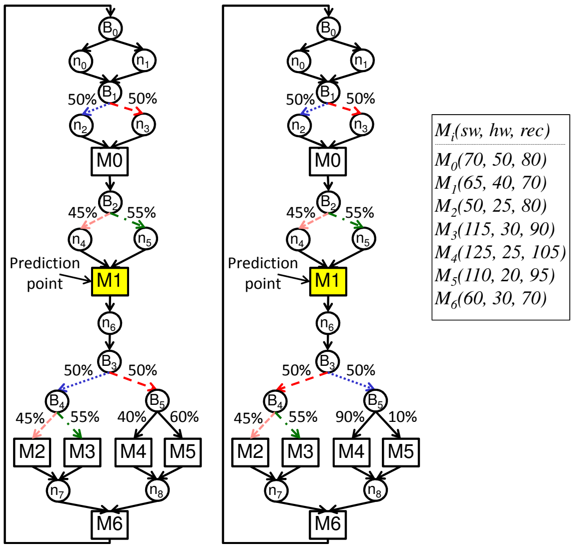
We assume that the application exhibits a nonstationary branch behavior. In order for any prediction algorithm to be applied, we consider that the application passes through an unpredictable number of stationary phases. Branch probabilities and correlations are stable in one phase and then change to another stable phase, in a manner unpredictable at design-time. Since we do not restrict in any way the length of a phase, this model can be applied to a large class of applications.

## III. Problem Formulation

Given an application (as described in Sec. II-B) intended to run on the reconfigurable architecture described in Sec. II-A, our goal is to determine dynamically, at each node $m \in \mathcal{H}$, the prefetches to be issued, such that the expected execution time of the application is minimized.

## IV. Motivational Example

Fig. 2 illustrates two of the stationary phases, each one with a different branch behavior, among those exhibited by an application. Hardware candidates are represented with squares, and software-only nodes with circles. For each phase, the edge probabilities of interest are illustrated on the graph, and they remain unchanged for that phase. We represent branch correlations with different line patterns: in phase 1, conditional branch $B_1$ is positively correlated with $B_3$, and $B_2$ is positively correlated with $B_4$; in phase 2, $B_5$ changes its probability and branches $B_1$ and $B_3$ become negatively correlated. For exemplification purposes we assume that the correlation is perfect; However, our prediction algorithm will capture the tendency of branches to be correlated, even if it is not perfect correlation. The software and hardware execution times for the candidates, as well as their reconfiguration times are also illustrated in Fig. 2. The software-only node $n_6$ has an execution time of at least 105 time units, enough to hide the reconfiguration overhead of any of the modules $M_2 ... M_6$ (the prefetch candidates at $M_1$).



(a) Phase 1

(b) Phase 2

Fig. 2: Motivational Example

We assume that the PDR region of the FPGA is big enough to host only one module at a time. Let us consider a prediction point in the application, module $M_1$, currently placed on the FPGA and executing. After its execution we want to issue a prefetch for a new module. Let us see how a static prefetch approach works, like [17] or [10]. Since it considers profiling information, regardless of the application's phase, it will issue the prefetch based on the average branch probabilities. In our example, considering the reconfiguration overheads and the execution times of the modules, $M_4$ would be always prefetched, having an average probability to be reached from $M_1$ of $50\% \cdot \frac{40\%+90\%}{2} = 32.5\%$ and generating the biggest expected performance improvement of $32.5\% \cdot (sw(M_4) - hw(M_4)) = 32.5$ time units. Notice that $M_6$, although reachable with $100\%$ probability, generates an improvement of only $sw(M_6) - hw(M_6) = 30$ $(< 32.5)$ time units. The static approach is performing very poorly, because in $100\% - 32.5\% = 67.5\%$ of the cases (when module $M_4$ is not reached), the prefetch is wasted and all the other modules (i.e. $M_2$, $M_3$ or $M_5$, depending on the path followed) will be executed in software.

Let us now see how the dynamic technique proposed in [9] works: using a Markov model, the approach estimates the probability to reach the next module from $M_1$, and based on this it issues the prefetches. Assuming that a phase is long enough for the model to learn the branch probabilities, the priorities associated with modules $M_2$, $M_3$, $M_4$ and $M_5$ are equal to their probabilities to be reached, i.e 45%, 55%, 40% and 60% in phase 1, and 45%, 55%, 90% and 10% in phase 2 respectively. Notice that $M_6$ is excluded, since the approach predicts only the next module to be reached. Based on this technique, the generated prefetches at $M_1$ will be: $M_5$ in phase 1 and $M_4$ in phase 2. This approach performs better than the static one, but many prefetch opportunities are still wasted (e.g. when $M_5$ is not reached in phase 1, in $100\% - 50\% \cdot 60\% = 70\%$ of the

[1] Note that this characterization is needed only for the candidates in $\mathcal{H}$.

cases or, similarly, when $M_4$ is not reached in phase 2).

The dynamic technique presented in [6], based on an aggregate gain table (as explained in Sec. I), will issue the following prefetches: $M_6$ in phase 1 (because its normalized long run aggregate gain is the highest, $100\% \cdot (sw(M_6) - hw(M_6)) = 30$ time units), and $M_4$ in phase 2 (for similar considerations, its normalized long run aggregate gain being $50\% \cdot 90\% \cdot (sw(M_4) - hw(M_4)) = 45$ time units). If we could exploit the branch correlation information, then we could issue prefetches better than both the static and the dynamic approaches discussed above. In phase 1, we should prefetch $M_6$ whenever path $B_1 - n_2 : B_2 - n_4$ is followed (because we reach $M_2$ and then $M_6$, and $sw(M_6) - hw(M_6) > sw(M_2) - hw(M_2)$) and $M_3$ whenever path $B_1 - n_2 : B_2 - n_5$ is followed (for similar considerations); Furthermore, if edge $B_1 - n_3$ is followed, then the best prefetch decision is $M_5$ in phase 1 ($60\%(sw(M_5) - hw(M_5)) > 40\%(sw(M_4) - hw(M_4)) > sw(M_6) - hw(M_6)$). Similar reasoning can be used for phase 2.

The big limitation of static prefetching is its lack of robustness and flexibility: the prefetches are fixed based on average profiling information and they cannot adapt to the run-time conditions. For our example, the static approaches always prefetch $M_4$, which is a very restrictive decision. Although the dynamic approaches provide extra flexibility, they still miss prefetch opportunities. One limitation of [9] is that it considers only the next module to be reached from the prediction point, thus excluding $M_6$ in our example. Another limitation exhibited by [9] and [6] is that the approaches rely only on the hardware modules' history and do not exploit the path information (like branch outcomes together with their correlations). As a result, for [9] and [6], the prefetch opportunities are wasted in more than 50% of the cases. We will present an approach that tries to overcome the mentioned limitations.

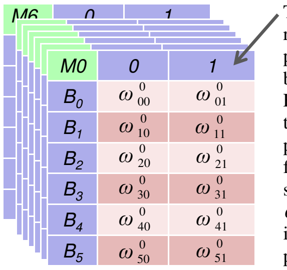
## V. Dynamic Configuration Prefetching

We first describe an idealized version of the algorithm (ignoring the microarchitectural constraints) and then we present a practical alternative in Sec. V-D.

The main idea is to assign priorities to all the hardware candidates and then issue prefetches based on them, at certain points in the application. One natural choice for the prefetch points is after the execution of a module $m \in \mathcal{H}$. It is important to have both an adaptive mechanism and a strategy to prioritize between reachable candidates in a certain phase, to obtain the maximum performance improvement. In our case, adaptation is accomplished by a piecewise linear predictor coupled with a mechanism, based on timestamps, to compute the performance gains associated with different prefetch decisions.

### A. The Piecewise Linear Predictor

*1) Overview:* This concept was previously applied in the context of branch prediction [7]. The idea was to exploit the history of a branch in order to predict its outcome (*taken* or *not taken*). We extend the concept for the case of FPGA configuration prefetching. We associate one piecewise linear predictor with each hardware candidate and we try to capture the correlation between a certain branch history leading to a certain prediction point, and a candidate being reached in the future. We remind that the prefetch points are after the execution of a hardware candidate. The branch history for a prefetch point $m \in \mathcal{H}$ is represented by the dynamic sequence of conditional branches (program path) leading to $m$. The predictor keeps track of the positive or negative correlation of the outcome of every branch in the history with the predicted hardware candidate being reached or not.



| M0 | 0 | 1 |
|---|---|---|
| $B_0$ | $\omega^0_{00}$ | $\omega^0_{01}$ |
| $B_1$ | $\omega^0_{10}$ | $\omega^0_{11}$ |
| $B_2$ | $\omega^0_{20}$ | $\omega^0_{21}$ |
| $B_3$ | $\omega^0_{30}$ | $\omega^0_{31}$ |
| $B_4$ | $\omega^0_{40}$ | $\omega^0_{41}$ |
| $B_5$ | $\omega^0_{50}$ | $\omega^0_{51}$ |

This dimension in the 3D array represents the position on which a particular branch occurs in the branch address history register $A$. For example, if at some point in time, $A = [B_3\ B_4]$, when making a prediction for $M_0$ the prediction function (described in the next section) would use the entries $\omega^0_{30}$ (because $B_3$ is on position 0 in $A$) and $\omega^0_{41}$ (because $B_4$ is on position 1 in $A$) from the 3D array.

Fig. 3: 3D Weight Array

The output of a predictor (associated with a hardware candidate) is a single number, obtained by aggregating the correlations of all branches in the current history, using a linear function (Predict in Algorithm 1). This function induces a hyperplane in the space of all outcomes for the current branch history, used to estimate what is the likelihood to reach the hardware candidate in discussion. We interpret the output $y$ of the predictor as the likelihood to reach the candidate, since the distance of $y$ from the hyperplane (on the positive side) is proportional to the degree of certainty that the candidate will be reached. Since there are many paths leading to a prediction point $m \in \mathcal{H}$, there are also many linear functions used for prediction. Together, they form a piecewise linear surface that separates the paths for which the hardware candidate will be reached in the future, from those paths for which it will not be reached.

*2) Data Structures:* The predictor uses the variables:

- $A$ – the branch address history register. At run-time, when a conditional branch is executed, its address is shifted into the first position of this register.
- $H$ – the branch outcome history register, containing the *taken* or *not taken* outcomes for branches. As for the address register, the outcomes are shifted into the first position. Together, $A$ and $H$ characterize the path history.
- $h$ – the length[2] of both the history registers $A$ and $H$.
- $HW$ – the hardware history register, containing the last $q$ hardware candidates reached.
- $q$ – the length[2] of the hardware history register $HW$.
- $\Omega$ – a 3D array of weights (shown in Fig. 3 for the example in Fig. 2). The indexes are: the ID of a hardware candidate, the ID of a branch and its position (index) in the path history. We can view $\Omega$ as a collection of matrices, one for each hardware candidate. The entries in $\Omega$ keep track of the correlations between branches and hardware candidates. For example, $\Omega[M_i, B_j, p]$, denoted $\omega^i_{jp}$, represents the weight of the correlation between branch $B_j$ occurring at index $p$ in the history $A$ and module $M_i$ being reached. Please note that addition and subtraction on the weights $\omega^i_{jp}$ saturate at $\pm 64$.

*3) Prediction Function:* Algorithm 1 details our approach for computing the likelihoods to reach the hardware candidates from a prediction point $m \in \mathcal{H}$. We use the function Predict to compute the *output* that reflects the correlation between the branch history leading to module $m$ and candidate $k$ being reached in the future. For all the entries in the current history (line 9), if the branch on position $i$ was *taken* then we add to the *output* the weight $\Omega[k, A[i], i]$ (line 10); otherwise we subtract it (line 11). Once the outputs were calculated for all candidates (lines 2-3), we normalize the results (line 6). The result $\widetilde{L}_{mk} \in [0, 1]$ represents the estimated likelihood to reach hardware candidate $k$ from prefetch point $m$, and will be used in computing the prefetch priority function (see Sec. V-C).

*4) Lazy Update Function:* After making a prediction, we need to train the predictor based on the real outcome (i.e. which modules were actually reached). Since this information becomes available only later, we opted for a lazy update: we save the context (history registers $A$ and $H$) based on

[2] The values of $h$ and $q$ reflect the trade-off between the space budget available for the predictor and its accuracy. We obtained good experimental results for relatively small values: $h \in [4, 16]$ and $q \in [2, 8]$.

**Algorithm 1** Prediction Function

```
procedure LIKELIHOODS($m$)
    for all $k \in \mathcal{H}\backslash\{m\}$ do
        $\lambda_{mk} = \text{PREDICT}(k, H, A)$
    $\lambda_{min} = \min_k \lambda_{mk}$; $\lambda_{max} = \max_k \lambda_{mk}$
    for all $k \in \mathcal{H}\backslash\{m\}$ do
        $\widetilde{L}_{mk} = (\lambda_{mk} - \lambda_{min})/(\lambda_{max} - \lambda_{min})$
function PREDICT($k$, $H[1..h]$, $A[1..h]$)
    $output = 0$
    for all $i = 1..h$ do
        if $H[i] == taken$ then $output = output + \Omega[k, A[i], i]$
        else $output = output - \Omega[k, A[i], i]$
    return $output$
```

**Algorithm 2** Lazy Update Function

```
1: procedure UPDATE($m$, $H_S^m[1..h]$, $A_S^m[1..h]$, $HW_R^m[1..q]$)
2:     for all $i = 1..h$ do
3:         for all $j = 1..q$ do update $\Omega[HW_R^m[j], A_S^m[i], i]$
4:         for all $k \in \mathcal{H}\backslash HW_R^m$ do update $\Omega[k, A_S^m[i], i]$
5:     $H_S^m \leftarrow H$; $A_S^m \leftarrow A$
6:     $top \leftarrow HW[q]$
7:     if $m \neq HW[1]$ then push $m$ in history register $HW$
8:     $HW_R^{top} \leftarrow HW$
```

which the prediction is made, and we update the predictor as late as possible, i.e. only when the same prediction point is reached again, and before making a new prediction. The next $q$ candidates that will be reached after the prediction point are accumulated in the hardware history register $HW$.

Algorithm 2 presents our lazy update. It takes as parameters the module $m$ where the update is done, the saved branch outcome register ($H_S^m$), the saved branch address register ($A_S^m$) and the saved hardware history register ($HW_R^m$), containing the first $q$ hardware modules executed after the prediction point $m$. The path information was saved when the last prediction was made at $m$, and the register $HW$ was saved when the $q^{th}$ module after $m$ was reached and $m$ was evicted from the $HW$ register. For all history positions (line 2), for all the $q$ modules reached after $m$ we update the corresponding weights in $\Omega$ by incrementing or decrementing them (saturating at $\pm 64$), depending if the branch on position $i$ was $taken$ or $not\ taken$ (line 3). For all the modules not reached after $m$ we do the opposite: decrement the weights in $\Omega$ for $taken$ branches, and increment them for $not\ taken$ ones (line 4). Next we save the current path that led to $m$ (line 5). Then we pop the $q^{th}$ hardware module ($top$) from the history register $HW$ (line 6) and we push the current module $m$ on the first position of $HW$, but only if $m$ is not already there (line 7). If $m$ is executed in a loop, we want to avoid having repetitions in the $HW$ history register; instead of pushing $m$, we only update its timestamp, used for computing the expected execution time gain (see Sec. V-B). Finally we save the hardware history register containing the $q$ modules executed after $top$ (line 8).

### *B. Estimating the Expected Execution Time Gain*

Let us consider a prediction point $m \in \mathcal{H}$ and one candidate $k \in \mathcal{H}$, reachable from $m$. Given that the reconfiguration of $k$ starts at $m$, we define the execution time gain $\gamma_{mk}$ as the time that is saved by executing $k$ in hardware (including any stalling cycles when the application is waiting for the reconfiguration of $k$ to be completed), compared to a software execution of $k$. Let $\chi_{mk}$ represent the distance (in time) from $m$ to $k$. The waiting time for a particular run of the application is given by $\varpi_{mk} = \max(0; rec(k) - \chi_{mk})$. This time cannot be negative (if a module is present on FPGA when it is reached, it does not matter how long ago its reconfiguration finished). The execution time gain over the software execution is:

$$\gamma_{mk} = \max(0, sw(k) - (\varpi_{mk} + hw(k))) \qquad (1)$$

If the software execution time of a candidate is shorter than waiting for its reconfiguration to finish plus executing it in hardware, then the module is executed in software, and the gain is zero. In order to estimate $\chi_{mk}$ on-line, we use timestamps. At run-time, whenever a candidate $k$ finishes executing, we save the value of the system clock and, at the same time, we compute $\chi_{mk}$ for all modules $m$ present in the current history $HW$ (relative to the timestamps when they finished executing).

In order to give higher precedence to candidates placed in loops, we adjust the value of $\varpi_{mk}$ as follows: First, for every hardware candidate $k \in \mathcal{H}$ we record its frequency, $\varphi_k$, during one run of the application. Then we compute an estimate $\widetilde{F}_k$ for the average frequency over the past runs, using an exponential smoothing formula in order to emphasize the recent history:

$$\widetilde{F}_k^t = \alpha \cdot \varphi_k^t + (1 - \alpha) \cdot \widetilde{F}_k^{t-1} \qquad (2)$$

In Eq. 2, $\widetilde{F}_k^t$ represents the estimate at time $t$ of the expected frequency for module $k$, $\widetilde{F}_k^{t-1}$ represents the previous estimate, $\varphi_k^t$ represents $k$'s frequency in the current application run and $\alpha$ is the smoothing parameter. Given $\widetilde{F}_k$, we adjust the waiting time $\widetilde{\varpi}_{mk} = \frac{\varpi_{mk}}{\widetilde{F}_k}$, and consequently the adjusted gain is:

$$\widetilde{\gamma}_{mk} = \max(0, sw(k) - (\widetilde{\varpi}_{mk} + hw(k))) \qquad (3)$$

This adjustment is done because, for modules in loops, even if the reconfiguration is not finished in the first few loop iterations and we execute the module in software first, we will gain from executing the module in hardware in future loop iterations.

We are interested to estimate the expected performance gain $\widetilde{\gamma}_{mk}$ over several application runs. We denote this estimate $\widetilde{G}_{mk}$, and we use an exponential smoothing formula (similar to Eq. 2) to compute it, emphasizing recent history:

$$\widetilde{G}_{mk}^t = \alpha \cdot \widetilde{\gamma}_{mk}^t + (1 - \alpha) \cdot \widetilde{G}_{mk}^{t-1} \qquad (4)$$

In Eq. 4, $\widetilde{G}_{mk}^t$ represents the estimate at time $t$ of the expected gain obtained if we start reconfiguring module $k$ at $m$, $\widetilde{G}_{mk}^{t-1}$ represents the previous estimate, $\widetilde{\gamma}_{mk}^t$ represents the adjusted gain computed considering the current application run and $\alpha$ is the smoothing parameter. The speed at which older observations are dampened is a function of $\alpha$, which can be adjusted to reflect the application characteristics: if the stationary phases are short, then $\alpha$ should be larger (for quick dampening and fast adaptation); if the phases are long, then $\alpha$ should be smaller.

### *C. The Priority Function $\Gamma_{mk}$*

At each node $m \in \mathcal{H}$ we assign priorities to all the candidates in $\mathcal{K}^m = \{k \in \mathcal{H} | \widetilde{L}_{mk} > 0\}$, thus deciding a prefetch order for all the candidates reachable from $m$ with nonzero likelihood (the computation of $\widetilde{L}_{mk}$ is described in Algorithm 1). Our priority function estimates the overall impact on the average execution time that results from different prefetches being issued after module $m$. Three factors are considered: 1) the estimated likelihood $\widetilde{L}_{mk}$ to reach a candidate $k \in \mathcal{K}^m$ from $m$, obtained from the piecewise linear prediction algorithm (see Sec. V-A3); 2) the estimated performance gain $\widetilde{G}_{mk}$ resulting if $k$ is prefetched at $m$ (see Sec. V-B); 3) the estimated frequencies for candidates, $\widetilde{F}_k$ (see Eq. 2), used to give higher precedence to modules executed many times inside a loop:

$$\begin{aligned}\Gamma_{mk} = {} & \widetilde{L}_{mk}(1 + \log_2 \widetilde{F}_k)\widetilde{G}_{mk} \; + \\ & + \sum_{h \in \mathcal{K}^m \backslash \{k\}} \widetilde{L}_{mh}(1 + \log_2 \widetilde{F}_h)\widetilde{G}_{kh}\end{aligned} \qquad (5)$$

The first term represents the contribution (in terms of expected execution time gain) of module $k$ and the second term captures the impact that $k$'s reconfiguration will produce on other modules competing with it for the reconfiguration controller.

Algorithm 3 presents our overall run-time strategy for configuration prefetching. Once a module $M_i$ is reached, we

**Algorithm 3** Overall Run-Time Strategy for Prefetching

```
1:{candidate $M_i$ is reached
2: increment frequency counter $\varphi_{M_i}$
3: UPDATE($M_i, H_S^{M_i}, A_S^{M_i}, HW_R^{M_i}$)    ▷ Update the predictor using the path history saved when the last prediction was made, and then save the current path history
4: LIKELIHOODS($M_i$)    ▷ Compute $\widetilde{L}_{M_i k}$
5: $\mathcal{K}^{M_i} = \{k \in \mathcal{H} | \widetilde{L}_{M_i k} > 0\}$
6: compute priority function $\Gamma_{M_i k}, \forall k \in \mathcal{K}^{M_i}$, based on the current estimates for likelihoods, frequencies and perf. gains, using Eq. 5
7: sort candidates in $\mathcal{K}^{M_i}$ in decreasing order of $\Gamma_{M_i k}$
8: remove modules that have area conflicts with higher priority modules
9: clear current prefetch queue
10: enqueue for prefetch the top 3 candidates, based on $\Gamma_{M_i k}$ }
    The pseudocode above executes in parallel with the one below
11:{if $M_i$ fully loaded on FPGA then execute $M_i$ on FPGA
12: else if remaining reconfiguration + $hw(M_i) < sw(M_i)$ then
13:         continue reconfiguration and then execute $M_i$ on FPGA
14: else execute $M_i$ in SW
15: save timestamp of $M_i$
    ▷ Compute estimated gains based on current finishing time of $M_i$
16: for all $k \in HW$ do    ▷ For the $q$ candidates reached before $M_i$
17:         compute performance gain $\widetilde{G}_{kM_i}$ with Eq. 4, using the timestamps of $k$ and $M_i$ to compute $\chi_{kM_i}$, $\widetilde{\varpi}_{kM_i}$ and $\widetilde{\gamma}_{kM_i}$ }
```

increment its frequency counter (line 2) that will be used at the end of the application to update the frequency estimates with Eq. 2. Next we perform the update of the piecewise linear predictor (line 3). We use the path history (registers $H_S^{M_i}$ and $A_S^{M_i}$) saved when the last prediction was made at $M_i$, and the $q$ candidates reached after $M_i$, saved in $HW_R^{M_i}$. After the update, we save the current path history (to be used at the next update), as described in Algorithm 2. Next we compute the likelihoods to reach other candidates from $M_i$ (line 4), as illustrated in Algorithm 1. Then, for all candidates reachable with nonzero likelihood (line 5), we compute the priority $\Gamma_{M_i k}$ (line 6) with Eq. 5. Once all $\Gamma_{M_i k}$ have been computed, we sort the candidates from $\mathcal{K}^{M_i}$ in decreasing order of their priority (line 7), giving precedence to modules placed in loops in case of equality. Next we remove from this list all the lower priority modules that have placement conflicts (i.e. intersecting areas on the reconfigurable region) with higher priority modules (line 8). Then, we also remove all the modules that are already fully configured on the FPGA (because they are going to be reused), we clear the current prefetch queue (line 9) and, finally, we enqueue for prefetch the top 3 candidates (line 10).

The execution of the predictor update and the mechanism of generating the prefetches (described above) takes place in parallel with the execution of $M_i$. This observation, coupled with the facts that our algorithm has a worst-case complexity of $O(|\mathcal{H}| \log |\mathcal{H}|)$ and that it runs as a dedicated hardware module (recall Sec. II-A), makes the on-line approach feasible. The execution of $M_i$ is done in hardware or in software (lines 11-14), depending on the run-time conditions. Once the execution of $M_i$ finishes, we save its timestamp and then we compute the estimated performance gains $\widetilde{G}_{kM_i}$ (line 17) for all the modules currently recorded in the hardware history register $HW$ (line 16). After this, the execution of the application continues as normal.

### D. A Practical Predictor

So far we have assumed boundless hardware; In this section we present a practical implementation of the piecewise linear predictor, considering a limited space budget. The most expensive component (with respect to space) is the 3D array $\Omega$ containing the weights used for prediction. It is interesting to observe that, if we look at the 2D matrix corresponding to a hardware candidate $M_i$, we see that not all the weights from the idealized version are actually used for making a prediction.

As an illustration, let us consider the 3D array $\Omega$ from Fig. 3 corresponding to the example from Fig. 2, and assume that the path history has a length of $h = 2$ and the hardware history register $HW$ has a length of $q = 2$. For these values of $h$ and $q$, for module $M_2$ the rows for branches $B_3$, $B_4$ and $B_5$ are never used because $B_3$, $B_4$ and $B_5$ never appear in the path history of any module that would make a prediction for candidate $M_2$. Similar remarks hold for other combinations of candidates and branches. Exploiting this, we can limit the second index of $\Omega$ by taking the branch IDs modulo $N$, where $N$ is chosen considering the space budget for $\Omega$. One consequence of this is aliasing, which for very strict space budgets might affect the prediction accuracy. Fortunately, as we show in the experimental section, the effects of aliasing are not so negative for practical sizes of $N$. The third index of $\Omega$ is limited by choosing an appropriate value for the path history length $h$, while the first index is limited naturally, since the number of hardware candidates is usually not overwhelming. Using a similar reasoning, we can also limit the indexes in the matrix of estimated performance gains $\widetilde{G}$, by taking the hardware IDs modulo an integer $M$.

As a result, the space required by our predictor (including the space for the timestamps and frequency counters $\widetilde{F}$ for candidates) is $|\mathcal{H}| \cdot N \cdot h \cdot 8 + h \cdot (\log_2 N + 1) + M \cdot M \cdot 16 + q \cdot (\log_2 |\mathcal{H}| + 1) + q \cdot 16 + |\mathcal{H}| \cdot 8$ bits. Depending on the choice of parameters, this value can be more or less bigger than the space requirements of the previous dynamic techniques: both [9] and [6] require $|\mathcal{H}| \cdot 16$ bits, where $\mathcal{H}$ is the set of hardware candidates. Nevertheless, as our experiments show, the extra overhead is worth paying, if we consider the performance improvement obtained in return.

## VI. Experimental Evaluation

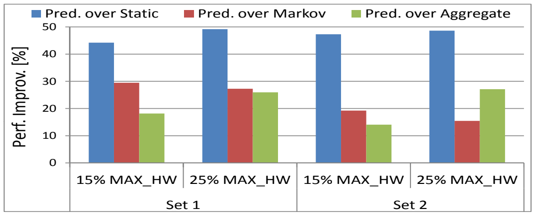
We generated 2 sets containing 25 flow graphs each: $Set_1$ with small graphs (between 48 and 166 nodes, $\sim$100 on average), and $Set_2$ with bigger graphs (between 209 and 830 nodes, $\sim$350 on average). The software execution time for each node was randomly generated in the range of 50 to 1000 time units. Between 15% and 40% of the nodes were selected as hardware candidates (those with the highest software execution times), and their hardware execution time was generated $\beta$ times smaller than their software one. The coefficient $\beta$, chosen from the uniform distribution on $[3, 7]$, models the variability of hardware speedups. We also generated the size of the candidates, which determined their reconfiguration time. The placement was decided using existing techniques (like [5]) that minimize the number of placement conflicts. We generated problem instances where the size of the reconfigurable region is a fraction (15% or 25%) of the total area required by all candidates, $\text{MAX\_HW} = \sum_{m \in \mathcal{H}} area(m)$.

For each application we have evaluated different prefetch techniques using an in-house Monte Carlo simulator (described in [10]) that considers the architectural assumptions described in Sec. II-A. We have determined the results with an accuracy of ±1% with confidence 99.9%.

### A. Performance Improvement

We were interested to see how our approach compares to the current state-of-art, both in static [10] and dynamic prefetching [9], [6]. Thus, we simulated each application using the prefetch queues generated by our approach and those generated by [10], [9] and [6]. The parameters for our predictor were chosen, depending on the application size, from the following ranges: $\alpha = [0.4, 0.6], q = [2, 8], h \in [4, 16], N \in [16, 32], M \in [8, 32]$.

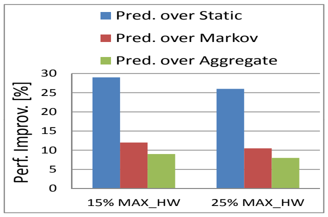
One metric suited to evaluate prefetch policies is the total time spent executing the hardware candidates, plus the time waiting for reconfigurations to finish (in case the reconfiguration overhead was not entirely overlapped with useful computa-



(a) Performance Improvement

|  |  | N |  |  |  |  |
|---|---|---|---|---|---|---|
|  |  | 4 | 8 | 16 | 32 | 64 |
| M | 8 | 1.22 | 1.32 | 1.23 | 1.12 | 1.08 |
|  | 16 | 1.21 | 1.29 | 1.27 | 1.14 | 1.07 |
|  | 32 | 1.19 | 1.27 | 1.24 | 1.13 | 1.05 |
|  | 64 | 1.18 | 1.28 | 1.25 | 1.14 | 1.02 |
|  | 128 | 1.18 | 1.30 | 1.23 | 1.16 | 1 |

(b) Aliasing Effects



(c) MM-1 Case Study

Fig. 4: Experimental Results

tions). If this value is small, it means that the prefetch policy had accurate predictions (many candidates were executed in hardware), and the prefetch was done early enough to have short waiting times.

We denote the average time spent executing hardware candidates, plus waiting for reconfigurations to finish, with $EX_P$ for our approach, with $EX_S$ for the static approach [10], with $EX_M$ for the dynamic Markov approach [9] and with $EX_A$ for the dynamic aggregate gains approach [6]. We compute the performance improvement of our approach over the static, $PI_S^P = \frac{EX_S - EX_P}{EX_S}$; similarly we calculate $PI_M^P$ and $PI_A^P$. Fig. 4a shows the results obtained (averaged over all graphs in $Set_1$ and $Set_2$). The improvements over the static approach [10] are higher because static prefetch lacks flexibility. The improvements over both dynamic approaches ([9], [6]) are also significant, ranging from 14% up to 29% on average.

### B. Aliasing Effects

Recall from Sec. V-D that we take the branch IDs modulo $N$ to index in the 3D array $\Omega$; also, the indexes in the matrix of estimated performance gains $\widetilde{G}$ are limited by taking the hardware IDs modulo $M$. We wanted to investigate the aliasing effects caused by these actions. Table 4b shows the results obtained for an application with 302 nodes (out of which 126 were hardware candidates), and containing 64 branches[3]. The size of the PDR region was set to 25%MAX_HW. The values in Table 4b represent the total time spent executing hardware candidates plus waiting for their reconfigurations to finish, $EX_P$, normalized to the case when $N = 64$ and $M = 128$. As can be seen, we can significantly reduce the space budget needed to implement the predictor with only marginal performance degradation. It is interesting to note that even for the most restrictive budget ($N = 4, M = 8$) the performance of our predictor degrades with only 22% compared to the ideal case ($N = 64, M = 128$), and we still outperform [9] by 21% and [6] by 17%.

### C. Case Study - MultiMedia Benchmark

Our case study was a MultiMedia benchmark (MM-1 from [18]). In order to obtain the inputs needed for our experiments, we used the framework and traces provided for the first Championship Branch Prediction competition [18]. 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 of the MM-1 application based on the given trace and we obtained 549 nodes. Then we used the traces to identify the parts of the application that have a high execution time (mainly loops). The software execution times were obtained considering the following cycles per instruction (CPI) values: 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 $\beta$ times smaller than the software one, where $\beta$ was chosen from the uniform distribution on the interval $[3, 7]$.

We have followed the same methodology as described in the above section to compare our approach with [10], [9] and [6]. The performance improvements obtained ($PI_S^P$, $PI_M^P$ and $PI_A^P$) are presented in Fig. 4c. Our method outperforms [10] by 29%, [9] by 12% and [6] by 9%.

## VII. Conclusions

We presented an approach for dynamic prefetching of FPGA reconfigurations, with the goal to minimize the expected execution time of an application. We use a piecewise linear predictor, coupled with an on-line mechanism, based on timestamps, in order to generate prefetches at run-time. Experiments show significant improvements over state-of-art techniques.

## References

[1] S. Banerjee *et al.*, “Physically-Aware HW-SW Partitioning for Reconfigurable Architectures with Partial Dynamic Reconfiguration,” *DAC*, 2005.

[2] J. Bispo *et al.*, “From Instruction Traces to Specialized Reconfigurable Arrays,” *Intl. Conf. on Reconfigurable Computing and FPGAs*, 2011.

[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] F. Ferrandi *et al.*, “A Design Methodology for Dynamic Reconfiguration: the Caronte Architecture,” *Intl. Parallel and Distributed Processing Symp.*, 2005.

[5] R. He *et al.*, “ISBA: An Independent Set-Based Algorithm for Automated Partial Reconfiguration Module Generation,” *ICCAD*, 2012.

[6] C. Huang and F. Vahid, “Transmuting Coprocessors: Dynamic Loading of FPGA Coprocessors,” *DAC*, 2009.

[7] D. Jimenez, “Piecewise Linear Branch Prediction,” *ISCA*, 2005.

[8] M. Koester *et al.*, “Design Optimizations for Tiled Partially Reconfigurable Systems,” *IEEE Trans. VLSI Syst.*, vol. 19, no. 6, 2011.

[9] Z. Li and S. Hauck, “Configuration Prefetching Techniques for Partial Reconfigurable Coprocessor with Relocation and Defragmentation,” *Intl. Symp. on FPGAs*, 2002.

[10] A. Lifa *et al.*, “Minimization of Average Execution Time Based on Speculative FPGA Configuration Prefetch,” *Intl. Conf. on Reconfigurable Computing and FPGAs*, 2012.

[11] G. Mariani *et al.*, “Using Multi-Objective Design Space Exploration to Enable Run-Time Resource Management for Reconfigurable Architectures,” *DATE*, 2012.

[12] E. M. Panainte *et al.*, “Instruction Scheduling for Dynamic Hardware Configurations,” *DATE*, 2005.

[13] M. Platzner, J. Teich, and N. Wehn, Eds., *Dynamically Reconfigurable Systems*. Springer, 2010.

[14] J. Resano *et al.*, “A Hybrid Prefetch Scheduling Heuristic to Minimize at Run-Time the Reconfiguration Overhead of Dynamically Reconfigurable Hardware,” *DATE*, 2005.

[15] P. Sedcole *et al.*, “Modular Dynamic Reconfiguration in Virtex FPGAs,” *IEE Comput. Digit. Tech.*, vol. 153, no. 3, 2006.

[16] T. Sherwood *et al.*, “Phase Tracking and Prediction,” *ISCA*, 2003.

[17] J. E. Sim *et al.*, “Interprocedural Placement-Aware Configuration Prefetching for FPGA-Based Systems,” *FCCM*, 2010.

[18] “The 1st Championship Branch Prediction Competition,” *Journal of Instruction-Level Parallelism*, 2004, http://www.jilp.org/cbp.

[19] Xilinx, “Partial Reconfiguration User Guide ug702,” 2012.

[20] Y. Yankova *et al.*, “Dwarv: Delftworkbench Automated Reconfigurable VHDL Generator,” *FPL*, 2007.

[3] Similar examples are omitted due to space constraints.