Interprocedural Placement-Aware Configuration Prefetching for FPGA-based Systems

Joon Edward Sim, Weng-Fai Wong
School of Computing
National University of Singapore
Singapore
{*esim, wongwf*}*@comp.nus.edu.sg*

Gregor Walla, Tobias Ziermann, Jürgen Teich
Department of Computer Science
University of Erlangen-Nuremberg
Germany
{*walla, ziermann, teich*}*@codesign.informatik.uni-erlangen.de*

***Abstract*—One of the major impediments to deploying partially run-time reconfigurable FPGAs as hardware accelerators is the time overhead involved in loading the hardware modules. While configuration prefetching is an effective method that can be employed to reduce this overhead, mispredicted prefetches may worsen the situation by increasing the number of reconfigurations needed. In this paper, we present a static algorithm for configuration prefetching in partially reconfigurable FPGAs that minimizes the reconfiguration overhead. By making use of profiling, the interprocedural control flow graph, and the placement information of hardware modules, our algorithm predicts hardware execution and tries to prefetch hardware modules as early as possible while minimizing the risk of mis-predictions. Our experiments show that our algorithm performs significantly better than current-state-of-the-art prefetching algorithms for control-bound applications.**

I. Introduction

Configuration prefetching [1] seeks to address the problem of high run-time reconfiguration overhead of FPGAs through parallelizing the (partial) reconfiguration of the FPGA with an application's execution. However, as we shall show, a misprediction in the prefetch can be very costly because it increases the number of configurations. Therefore, the correct scheduling of configurations is the key to good performance of such accelerators. Ideally, the execution of a hardware module should be predicted *as early as possible* (given the huge configuration latency) and *as accurately as possible* (so as to avoid costly recoveries). In this paper, we present an algorithm that reduces configuration latency for specifications written in interprocedural control flow graphs [3]. Through the use of profile information, the algorithm predicts the execution of hardware modules by computing 'placement-aware' probabilities. These probabilities are in turn used for the generation of prefetching codes that are then inserted into the control-flow graph. Our experiments show that our approach significantly outperforms previous works.

II. Background

A. Architecture Model

We consider the architecture model as shown in Figure 1. The model is based on actual silicon devices such as the Xilinx Virtex family of FPGAs, especially Virtex-II Pro, IV and V. The software code, data, and the bitstreams to be loaded onto the reconfigurable region are stored in memory. The CPU is the main controller of application execution and is also responsible for initiating the reconfiguration of the FPGA. The reconfiguration manager is a hardware module that loads bitstream data from the memory upon requests issued by the CPU.

The reconfigurable region is organized as n slots where hardware modules can be placed. We consider any two placements of the hardware modules with overlapping slots to be in 'physical placement conflict' (or just 'conflict' for the rest of the paper). Conflicting hardware modules cannot be loaded into the reconfigurable region at the same time.

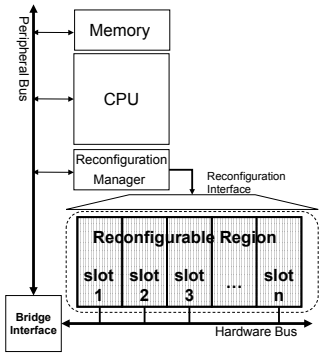
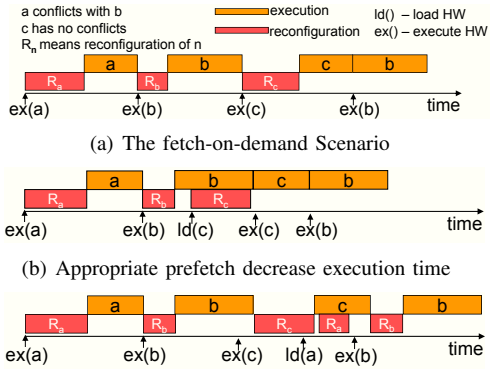


Figure 1. Architecture Model

III. Problem Formulation And Motivation

The aim of this paper is to minimize the reconfiguration delay of a single, sequential program for the platform described above. We have assumed that the placements of the hardware modules are fixed. We represent the program as an interprocedural control flow graph (ICFG), a directed graph $G = (V, E, C, U, HW)$ where every node on the graph is either a basic block or a hardware node (a block of code that invokes hardware execution). V is the set of all the nodes in the graph. E is the set of all the edges in the graph. $head(e)$ and $tail(e)$ refers to the begin and end node of edge e respectively. C is the set of all call sites and $C \subset V$. U is



(a) The fetch-on-demand Scenario

(b) Appropriate prefetch decrease execution time

(c) Misprefetch increases execution time

Figure 2. How prefetching affects overall execution time

the set of exit nodes of all procedures and $U \subset V$. HW is the set of hardware nodes. We denote two conflicting hardware nodes hw_1 and hw_2 as $hw_1 \nsim hw_2$.

Consider the execution sequence **abcb** for hardware modules a, b and c. Hardware modules not yet present on the FPGA by the time of invocation will be loaded prior to execution. Figure 2(a) shows the "fetch-on-demand" schedule of the execution of the hardware modules (i.e., no configuration prefetching). Figure 2(b) shows how the overall execution time is reduced when c is loaded during the execution of b. On the other hand, a mispredicted loading may result in a schedule that is longer than the "fetch-on-demand". In Figure 2(c), the loading of a during the execution of c results in an additional reconfiguration of b later, hence lengthening the original "fetch-on-demand" schedule. This paper aims to ensure that configurations are loaded at appropriate times so that the reconfiguration overhead is minimized.

IV. Interprocedural Placement-Aware Configuration Scheduling

The proposed algorithm has five stages. 1) Obtain the frequency of executing each control-flow edge through profiling and remove all edges that are not executed. The weight function w of each edge is computed using the equation $w(e) = \frac{\text{frequency of edge } e}{\text{total frequency count of node } head(e)}$. 2) Compute *post dominators* [6] for every node, denoting the immediate post-dominator of each v to be $ipdom(v)$. 3) Compute the *intra post dominator paths* (IPDP) information for each node v, which are set of paths that start from v with the following properties: a) There does not exist any node along the path that is a post-dominator of any other node along the path and b) the estimated probability of this path being taken is greater than a threshold value. This threshold value is set to be 0.0005 in our experiments. The estimated probability of taking a path $P_{path}(p)$ is the product of the weightage of the edges on the path. 4) Compute for every node on the graph the estimated *placement-aware probability* (PAP) of reaching each hardware node using the IPDP and post dominator information through a fixed-point iterative method. 5) Insert hardware loading instructions into candidate basic blocks chosen based on the PAP information. The rest of this section focuses on describing steps 4 and 5 in more detail.

Algorithm 1: Iterative Probability Updating

```
Result: Final placement-aware probabilities computed for each node
        ∀v ∈ V
change ← true;
while change do
    change ← false;
    forall v ∈ V do
        if v is an exit node i.e. v ∈ U then
            continue;
        else if v is a call site i.e. v ∈ C then
            tmp_change ← update_probabilities_for_call_site(v);
        else
            tmp_change ← update_general_probabilities(v);
        end
        if tmp_change then
            change ← true;
        end
    end
end
return P;
```

Algorithm 2: update_general_probabilities(v)

```
Input: v
Result: Update probabilities for v based on post-dominator and
        IPDP_Prob information
change ← false;
forall hw ∈ HW do
    if v is hw or conflicts with hw i.e. v = hw ∨ v ≁ hw then
        continue;
    max_prob ← -1;
    forall p ∈ IPDP_v do
        if no nodes in p conflicts with hw (i.e. ∄n : n ∈ p ∧ n ≁ hw)
        then
            new_prob ← P_path(p) × P(end(p), hw);
            new_IPDP_prob ← new_prob − P(ipdom(v), hw);
            if new_IPDP_prob > IPDP_Prob(v, hw) then
                IPDP_Prob(v, hw) ← new_IPDP_prob;
            if new_prob > max_prob then
                max_prob ← new_prob;
    end
    factor ← 1.0;
    forall hw' ∈ HW : hw' ≁ hw do
        factor ← factor − IPDP_Prob(v, hw');
    end
    if factor × P(ipdom(v), hw) > max_prob then
        max_prob ← factor × P(ipdom(v), hw);
    if max_prob < threshold then
        temp_p(hw) ← 0;
    else
        temp_p(hw) ← max_prob;
    end
end
if ∃ hw ∈ HW : temp_p(hw) ≠ P(v,hw) then
    change ← true;
    P(v) ← temp_p;
return change;
```

A. Iterative PAP Estimation and Prefetch Code Generation

Our configuration prefetch algorithm is based on a placement-aware probability computed for every node.

Definition 4.1: **Placement-aware Probability (PAP)** The placement-aware probability (PAP) of a node n of the ICFG reaching hardware node g is the sum of the estimated

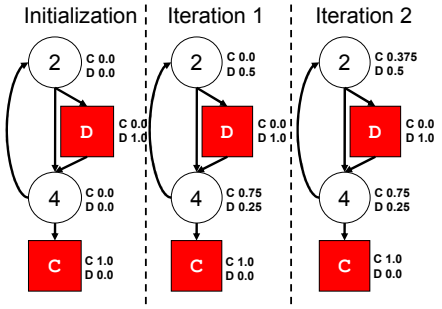


Figure 3. All edges except (D,4) are weighted 0.5. Hardware nodes `C` and `D` conflict with each other.

probabilities of all paths from n to g such that there is no conflicting hardware node of g on the path.

We estimate the PAP of reaching every hardware node for each node in the ICFG through an iterative fixed point method. Algorithm 1 shows a main loop that processes all the nodes in the graph during each iteration and continues doing so until a fixed-point (i.e., the estimated probabilities for each node have stabilized.) is reached. Throughout all iterations, we maintain two two-dimensional vectors *IPDP_Prob* and *P*. $IPDP\_Prob(v,hw)$ are the estimated probabilities that a node v may reach a hardware node hw through its IPDP. $P(v,hw)$ is the estimated PAP that a node v may reach a hardware node hw through all possible paths while $P(v)$ is a vector of all estimated PAPs for node v. Every $P(v,hw)$ is initialized to zeros except when $v = hw$, where $P(v,hw)$ is initialized to 1. A procedure may have multiple callers. Due to the uncertainty of the call context, we do not compute PAPs for the exit nodes of the procedures.

We distinguish between the general case and call sites for the updating of estimated PAPs. Algorithm 2 shows how the estimated PAPs for a general node v (i.e., neither call site nor exit node) is updated, by computing a vector of estimated PAPs $temp\_p$ that will be used to update $P(v)$ if these 2 vectors are different. In the case when $P(v)$ is updated, a change is reported. We estimate the PAP of call sites by taking the maximal value of a) either the weighted sum of the PAPs of all its callee sites or b) the PAP of its own immediate post-dominator.

Example: Figure 3 shows how the estimated PAPs are computed for a simple CFG. During initialization, the PAPs of reaching the hardware nodes are set to 0 except for hardware nodes themselves (e.g., the probability of node `C` reaching `C` is 1). We note that while `C` is a post dominator of `4`, the estimated PAP of reaching `C` is 0.75. This is because there is also a 0.25 probability of reaching `D` through `2` from `4`. While this could be an over-estimation, it is sufficient for our purposes to obtain relative size of probabilities for reaching each hardware module.

After PAP estimation, we proceed to select basic blocks that become candidates for insertion of hardware prefetch instructions. The number of candidates can be reduced by clearing the PAPs for nodes where all its parents have the same PAPs as the node itself. The basic blocks with non-zero PAPs are where we insert hardware prefetch instructions. The exact hardware module loaded, however, depends on run-time conditions. If the most probable hardware module is not yet loaded and not being reconfigured, it will be loaded at the candidate basic block. Otherwise if there is no ongoing loading, the next most probable hardware module that is not yet loaded and not conflicting with the most probable hardware module will be loaded on the FPGA.

V. Experimental Evaluation

A. Experimental Setup

To evaluate the effectiveness of our approach, we performed experiments with two applications, `429.mcf` from the SPEC2006 benchmark suite [7], and `h264enc` from the MediaBench II video benchmark suite [8]. `429.mcf` performs single-depot vehicle scheduling while `h264enc`[9] is a H.264/AVC (Advanced Video Coding) encoder. Through profiling, we identified 6 compute-intensive regions for `429.mcf` and 7 such regions for `h264enc` that are to be implemented in hardware. These compute-intensive regions are either basic blocks or loops in the original program. For our experiments, we assumed that the hardware performance is 5 times faster than the software counter-parts.

We modeled our experimental platform along the lines of ReCoBus [10] which supports complex run-time reconfiguration. The ReCoBus's reconfiguration regions are organized in terms of *reconfigurable slots* that are 6 CLB columns in size. A slot is the smallest granularity that any hardware module can occupy on the FPGA. For our experiments, we assumed a hardware device that has a similar geometry as Xilinx Virtex II Pro XC2VP30 [11] that is organized as a CLB matrix of 80 rows and 56 columns, with the PowerPC CPU operating at 300MHz and the 32 bits wide reconfiguration port at 100Mhz. The overhead of reconfiguring each slot can be calculated based on the data in the datasheet[11]. It is approximately 81,576 PPC cycles.

We performed our experiments using a trace-based simulator that takes in the basic block trace and the execution time information and computes the execution time of the application on the reconfigurable computing architecture we modeled. We compared the performance of our algorithm by comparing it against four other algorithms described below.

Fetch-on-demand (FOD): In the FOD schedule, there is no prefetching of configurations. The hardware modules are loaded if they are encountered during execution, and if it is not already residing on the FPGA. It is reasonable to expect that any prefetching approach should do better than this. We used the execution time of the fetch-on-demand scenario as the baseline for comparison in our experiments.

Optimal prefetching (OPT): Our implementation of OPT relies on the algorithm described in [15]. It can be done only if the entire execution trace is already known beforehand. We do not expect any static approach to be able

to perform better, but the gap between OPT and FOD serves as a useful gauge for the effectiveness of our approach.

Placement-blind probabilistic algorithm (PBP): The implementation of PBP is based on [5]. It should be noted that the PBP was developed for relocatable and defragmentable FPGAs, and not for the Xilinx FPGA architectures. Therefore, this approach does not account for conflicts between the hardware modules.

Conservative analysis (CA): The implementation of CA is based on [4]. Reconfigurations are not preempted in this case. Instead, all previously issued prefetches (maintained in a queue) must be completed before a hardware module that is yet to be configured can execute. The prefetch queue is cleared only at the insertion edges.

B. Experimental Results

The specific placement of the hardware modules affect the conflict relationship between them. Therefore, in order to evaluate the effectiveness of our algorithm, we performed experiments for different placements and Figure 4 shows the various speedups/slowdown results. Each placement is named after the corresponding applications. Hence, placements starting with `h264-` refers to placements for `h264enc` while labels starting with `mcf-` refers to placements for `429.mcf`. The placements that are labeled with 's6' are placements generated for a reconfigurable region of 6 reconfigurable slots while those labeled with 's8' were generated for a reconfigurable region of 8 reconfigurable slots.

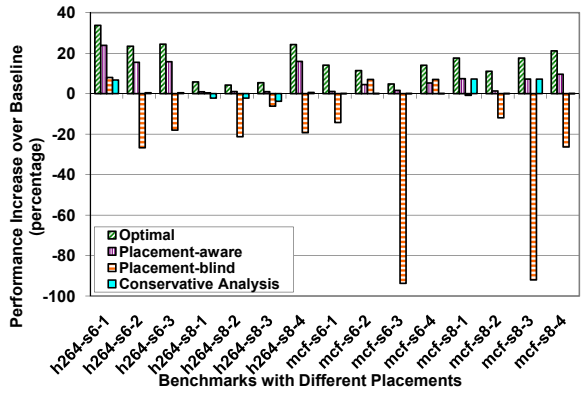


Figure 4. Speedups over baseline

We make the following observation of the results shown: a) Performance degrades seriously when conflicts are not taken into account. The PBP suffers 20% to 90% degradation in performance for most of the placement sets tested in our experiments. b) CA is consistently either very close to baseline or slightly worse than baseline. Being conservative, prefetches inserted in control flow points are very near to where the hardware modules need to execute. c) For the same benchmark, the speedup that can be gained in our approach depends on the placement. In particular, `h264-s6-1` is the best for `h264enc`, achieving a speedup of nearly 30% that of the optimal prefetch algorithm. This shows how placements affect both the overall performance and the opportunities available for configuration prefetching. d) On the whole, our algorithm returned results that fall between 17% and 72% of the OPT results without having to deal with gigabytes of traces needed by the latter.

VI. Conclusion

In this paper, we have described a novel method that statically determines the places in an application's control flow graph where prefetches of hardware modules into the FPGA should be initiated such that the reconfiguration overhead is minimized. Our approach performs consistently better than our baseline and also out-performs the state-of-the-art static prefetching algorithms, coming to 72% of optimal prefetching at its best. As future work, we intend to extend the algorithm such that it will also take into consideration the execution phases of the applications.

References

[1] S. Hauck, "Configuration prefetch for single context reconfigurable coprocessors," in *FPGA '98*, 1998, pp. 65–74.

[2] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, 1970.

[3] S. Sinha, M. J. Harrold, and G. Rothermel, "Interprocedural control dependence," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, pp. 209–254, 2001.

[4] E. M. Panainte, K. Bertels, and S. Vassiliadis, "Interprocedural compiler optimization for partial run-time reconfiguration," *J. VLSI Signal Process. Syst.*, vol. 43, no. 2-3, pp. 161–172, 2006.

[5] Z. Li and S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation," in *FPGA '02*, 2002, pp. 187–195.

[6] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 1, pp. 121–141, 1979.

[7] "SPEC Benchmark Suite 2006. http://www.spec.org."

[8] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf, "Mediabench II video: Expediting the next generation of video systems research," *Microprocessors and Microsystems*, vol. 33, no. 4, pp. 301–318, 2009.

[9] T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, no. 7, pp. 560–576, 2003.

[10] D. Koch, C. Beckhoff, and J. Teich, "A communication architecture for complex runtime reconfigurable systems and its implementation on Spartan-3 FPGAs," in *FPGA '09*, 2009, pp. 253–256.

[11] *Xilinx Virtex-II Pro Platform FPGAs: complete data sheet*, Xilinx Inc., San Jose, CA, United States.

[12] "The Open IMPACT IA-64 Compiler." http://gelato.uiuc.edu."

[13] *Intel Itanium 2 Processor Reference Manual for Software Development*, Intel Corp, June 2002.

[14] *PowerPC 405 Processor Block Reference Guide*, Xilinx Inc., San Jose, CA, United States, July 2005, avaiable at http://www.xilinx.com.

[15] J. E. Sim, W. F. Wong, and J. Teich, "Optimal placement-aware trace-based scheduling of hardware reconfigurations for FPGA accelerators," in *FCCM '09*, Apr. 2009, pp. 279–282.