PERFORMANCE EVALUATION OF A PRELOADING MODEL IN DYNAMICALLY RECONFIGURABLE PROCESSORS

Kyprianos Papademetriou * *Apostolos Dollas* †

Department of Electronic and Computer Engineering
Technical University of Crete
GR73100 Chania, Crete, Greece
email: {kpapadim, dollas}@mhl.tuc.gr

ABSTRACT

Dynamic reconfiguration allows for the reuse of the same hardware by different tasks of an application at different stages of its execution. However, reconfiguring the hardware at run-time incurs a configuration delay causing performance degradation of the application. This paper evaluates a preloading model that hides the configuration overhead. An existing preloading model is augmented according to the physical constraints of the system. A reduction of 6% up to 86% in execution time has been obtained with the new model.

1. INTRODUCTION

In our first work [1] it was shown that it is well worth investigating whether a preloading model leverages the performance of an application designed on a partially reconfigurable hardware. The contributions of the present work vs. [1] include a new experimental framework that better models a reconfigurable processor, examination of the impact of the proposed model to the overall execution length, and discussion of the problems incurred by the proposed model.

A variety of preloading models exist that attempt to reduce reconfiguration overhead [2]. Most of them do not take into account resource constraints. Banerjee et al [3] consider reconfiguration overhead and configuration prefetching, while selecting a suitable task granularity. Then, simultaneous scheduling and columnar placement are performed, where the scheduling integrates prefetch to reduce reconfiguration overhead. Our work augments the results in [2] which describes the static prefetching algorithm. It is also related to [3] that schedules tasks according to the physical resource constraints. The difference is that present work examines specific places in the code, i.e., branches, to select the task to be transformed according to the resource constraints.

*Funded with a Ph.D fellowship by the Greek Ministry of National Education and Religious Affairs under the program Heraklitus, EPEAEK II

†Also at ITRI, Wright State University, Ohio

2. CONFIGURATION IN MODERN DEVICES

Dynamic reconfiguration is applied on reconfigurable processors combining a fixed processing unit (FPU) with a reconfigurable processing unit (RPU) on a single chip. In a realistic scenario FPU initiates RPU configuration and continues undisturbed its execution, i.e., FPU execution is not stalled waiting for the configuration data to be loaded. The instruction inserted into FPU's code resembles any other instruction, consuming a single slot in the pipeline.

The configuration memory of Xilinx Virtex-II is arranged in 1-bit width vertical frames. They are the smallest addressable segments of the device configuration memory space and they configure a narrow vertical slice of many physical resources. A pad frame is added at the end of the configuration data which flushes out the reconfiguration pipeline [4].

Although Virtex-II devices have heterogeneous physical resources this work assumes a homogeneous device model wherein application tasks are placed on CLB columns only. Furthermore, for sake of simplicity we consider that a task's circuit is placed in multiples of one CLB column and not in multiples of one frame. As a consequence, reconfiguration is performed in CLB column level only.

3. PROBLEM DESCRIPTION - PROPOSED APPROACH

This section describes the problem that this work deals with as well as the proposed model by presenting the modifications that have been made to the original prefetching model [2]. Figure 1 shows an example application that comprises of five tasks running on a reconfigurable processor. Figure 2 represents the reconfigurable processor with the parts occupied by the FPU and the RPU along with the partitioning of tasks. Tasks t0, t1 and t3 run on the FPU and t2, t4 run on the RPU. In task t0, among other instructions, a decision is made regarding which one of tasks t1, t3 should be followed. Then, the corresponding RPUOP (RPU operation) is called. Given that the available hardware allows for both RPUOPs

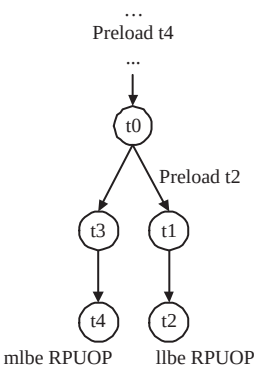

Fig. 1. Insertion of preload instructions according to the original model.

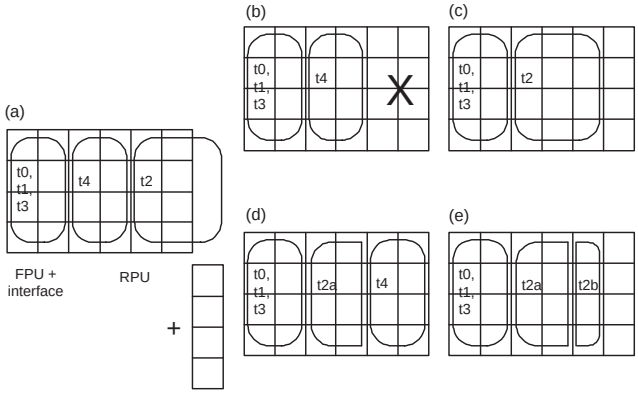

Fig. 2. (a) shows that not all tasks fit into the platform. (b) and (c) correspond to the original model, (d) and (e) correspond to the augmented model.

to be simultaneously placed onto the RPU, no reconfiguration delay is incurred during transition of execution from t0 to the selected RPUOP.

On the contrary, if a resource-constrained RPU is employed that can at most hold one of the t2, t4 a delay might be incurred. In Figure 2(a) we assume that t4 corresponds to the most likely to be executed (mlbe) RPUOP, whereas t2 corresponds to the least likely to be executed (llbe) RPUOP. One more CLB column would be required to place both RPUOPs. In Figure 2(b), as t4 has been preloaded onto the RPU according to the prefetching algorithm [2], in case the outcome of branch in t0 requires t2 after the intervening task t1 the execution might be stalled. The second preload instruction of Figure 1, reconfigures the RPU with t2 which illustrated in Figure 2(c). If the system supports concurrent FPU execution and RPU reconfiguration, t1 execution will hide part or even all of the reconfiguration time. The amount of time that can be hidden depends on the execution length of t1 and the configuration latency of t2.

The static prefetching algorithm of [2] considers that since the total size of the reachable RPUOPs for a certain node could exceed the capacity of the chip, only highly probable prefetches under the size restriction of the chip are generated. The rest of the reachable RPUOPs are ignored. In our model given an area constraint, transformations to the task graph are performed to employ a more aggressive preload that utilizes all the physical resources. This is illustrated in Figure 2(d) and (e). If t4 is the mlbe RPUOP, it is selected for preloading. RPUOP t2 is then transformed; it is split into two subtasks in that t2a fits on the remaining portion of the hardware. Task t2a is preloaded before t0. Therefore, in Figure 1 if the outcome of the branch requires t2, only subtask t2b is required to be loaded after t0.

In this work it is assumed that the RPUOPs selected for split are divisible and recombinable. The idea is along with the placement of the mlbe RPUOP to automatically break down the llbe RPUOP into non-functional tasks according to the physical constraints. Then one portion is placed on the RPU and in case the llbe RPUOP is called, the remaining portion is loaded by displacing the mlbe RPUOP that was not finally executed. The cost of disconnecting the displaced RPUOP when loading the remaining portion of the split RPUOP is not examined. In addition, as illustrated in Figure 2(d) the proposed model fully utilizes the available area. An issue arisen at this point is the limitation to the placement options of the RPUOPs compared to the original model. To effectively exploit the augmented model, the first subtask should be placed on an appropriate location where the second subtask would be adjacently placed by replacing the mlbe RPUOP as shown in Figure 2(e). The original model does not deal with such restrictions, i.e., llbe RPUOP is loaded only when the mlbe RPUOP is not executed. The trade-offs between the two models regarding this issue is an interesting study but present work does not deal with this.

4. EXPERIMENTAL SETUP

The experimental setup consists of an application scenario and the attributes that represent the physical resources of a reconfigurable processor as well as the time and area required to carry out the tasks of the application. The application is represented by a task graph where each node corresponds to a task. This graph can be extracted from a functional specification in a high-level language like Verilog, VHDL, C etc. In order to generate different problem instances the TGFF tool [5] was used. It generates pseudo-random task-graphs while users have parametric control over a number of attributes for tasks, processors, and communication resources. Correlations between attributes may be parametrically controlled.

In Figure 2, on the left part of the device an FPU is placed, e.g., Microblaze, along with the interface with the

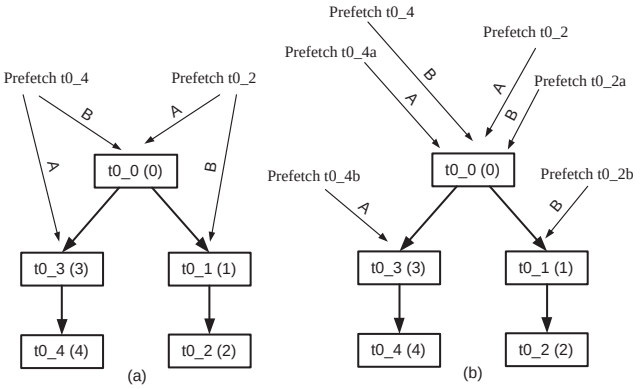

Fig. 3. (a) and (b) have the preloads according to the original and the augmented model respectively. Two different scenarios regarding the insertion and sequence of preload instructions are denoted by the labels A and B at the thin arrows.

RPU, e.g., ICAP. The rest of the device implements the RPU. Figure 3 has the examined task graph as generated by TGFF. Notice that regarding the task names a number on the left of the low dash exists indicating the graph's ID. This ID is used if more that one graphs are generated and as we examine one graph only we eliminate it, e.g., t0_2 will be referred as t2.

The left graph of Figure 3 is carried out with the original model and the right graph is carried out with the augmented model. In Figure 3(a), when the total size of t2 and t4 is larger than the available hardware only the mlbe RPUOP is preloaded. For example, in scenario B it is the t4 that is preloaded before t0. If the decision matches the preload instruction no new preload is executed (the demanded configuration data are already loaded or are being loaded to the RPU). On the contrary, if t2 is the outcome of the branch, the corresponding preload instruction located after t0 should be executed, incurring a greater configuration overhead to the process (the demanded configuration data are not contained onto RPU when execution reaches t2). It is this case we examine. In Figure 3(b) in scenario B, task t4 is preloaded and then a preload instruction of a portion of the task t2 is inserted. The latter was split into two subtasks, t2a and t2b, by statically transform the given task graph. The split was performed in that the total size of t4 and t2a equals to the available hardware. In case the branch's outcome requires t2 the preload instruction for t2b is executed.

A testbench is constructed consisting of 50 systems executing the same task graph. Each task node is unique. This is denoted by the values in the parentheses of Figure 3. The platform contains 24 CLB columns of 32 CLBs each which resembles the Virtex-II XC2V500 device. The FPU with the interface occupies 6 CLB columns which is roughly realistic compared to the area required for Microblaze and ICAP. It is assumed that each task carried out by the FPU takes an average execution of $300 \pm 250\mu s$. The RPU is implemented with the remaining 18 CLBs columns. The average number of CLB columns required by a task is chosen to 10 ± 8. The tasks on the RPU are assumed to be executed in an average of $200 \pm 180\mu s$. The only correlated attributes used in the experiments are the CLBs needed for each RPUOP and their execution time. Configuration time is also needed for the experiments. A similar to ICAP interface is considered with 8-bit of data running at 66 MHz. This is used for the computations of Table 1. Configuration time is proportional to the number of frames to be loaded. This is used to extract the configuration time of the CLB columns to be loaded.

Table 1. Configuration attributes for XC2V500.

Data from Xilinx's data-sheet:	
Device	XC2V500
Number of CLB Col./chip	24
Number of frames/chip	928
Conf. time/chip	$4.85ms$
Number of frames/CLB col.	22
Simple computations give:	
Conf. time/frame	$4.85ms \div 928 = 5.22\mu s$
Conf. time/CLB col.	$22 * 5.22\mu s = 115\mu s$
Conf. time/CLB col. w. pad	$115\mu s + 5.22\mu s = 120.22\mu s$

This setup considers column-based reconfiguration and compared to [1] which examines reconfiguration per CLB unit it is more realistic. Except of the configuration latency and overhead that were examined in the first work as well, present work examines more parameters such as how does the utilization of remaining CLBs after preloading the mlbe RPUOP affect the application execution length.

5. EXPERIMENTAL RESULTS AND DISCUSSION

In this section the experimental results showing the performance gained by the augmented model are discussed. Figure 4 has the execution length of the overall process for the two models for different values of remaining CLB columns after preloading the mlbe RPUOP. A set of 50 experiments were conducted for the same task graph and the average for each CLB column was used to construct the chart graph. In some cases the RPUOPs were completely preloaded, i.e., their total size was smaller than the available hardware. These cases are not included. The results concern llbe RPUOP execution. It is observed that as the volume of CLB columns that can be utilized for preloading the llbe RPUOP increases the execution length of the augmented model decreases compared to the original model.

To evaluate the improvement in execution length, the equation $100 \times (EL_{orig} - EL_{augm}) \div EL_{augm}$ was used

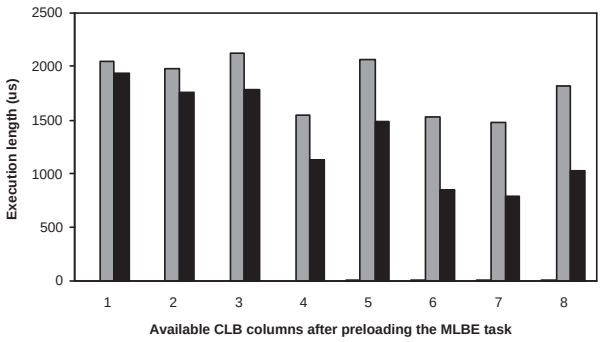

Fig. 4. Execution lengths for the original and augmented model for different values of remaining CLB columns after preloading the mlbe RPUOP. The llbe RPUOP is chosen for execution.

where, EL_{orig} and EL_{augm} are the execution lengths for the original and the augmented model respectively. For 1 available CLB column the decrease was 6.16%, whereas for 7 available CLB columns the biggest improvement was obtained, equal to 86.55%.

Table 2 shows the reconfiguration overhead for the augmented model in contrast to the original model. It consolidates the worst cases with respect to the length of configuration latency for the augmented model, i.e., it corresponds to the cases in which for a specific number of remaining CLB columns after preloading the mlbe RPUOP, the second portion of the llbe RPUOP to be loaded is the largest. *FPU task* column has the execution time of the task (t1 or t3) before which the preload instruction is inserted. *CLOM* (Configuration Latency for the Original Model) refers to the original model and is the time needed to load the whole llbe RPUOP. *CLAM* (Configuration Latency for the Augmented Model) refers to the augmented model and is the time needed to load the second portion of the llbe RPUOP. *CLB col* is the number of remaining CLB columns after preloading the mlbe RPUOP. Reconfiguration overhead corresponds to the amount of time that can(if positive)/can not(if negative) be hidden by overlapping reconfiguration with processor execution. *ROOM* (Reconfiguration Overhead for the Original Model) is the overhead caused by loading llbe RPUOP before FPU task (after the branch). *ROAM* (Reconfiguration Overhead for the Augmented Model) is the overhead caused by loading the second portion of llbe RPUOP before FPU task (after the branch).

The above results illustrate the relation between configuration latency and reconfiguration overhead and whether reconfiguration can be hidden by the processor's execution. In a system where the FPU task executes concurrently with the RPU reconfiguration, depending on the FPU's and RPU's tasks execution time and the number of remaining CLB columns after preloading the mlbe RPUOP, the designer can decide whether it is worthwhile trying to hide the llbe RPUOP's configuration latency by applying an appropriate split operation.

Table 2. Performance gain of the augmented algorithm. Worst cases regarding CLAM time are shown. All times are in μs.

FPU task	CLOM	CLAM	CLB col	ROOM	ROAM
167	1890	1775	1	-1722	-1607
335	1752	1522	2	-1417	-1187
335	1616	1271	3	-1281	-936
125	1195	735	4	-1070	-610
525	1285	710	5	-759	-184
227	815	125	6	-698	-8.4
236	923	118	7	-686	118
333	1116	196	8	-783	136

6. CONCLUSIONS

The experimental results showed significant improvements in reconfiguration overhead over the original prefetching model. The main advantage of the proposed model is the increase in the utilization of the available hardware achieved by splitting the least likely to be executed task. A problem arisen is the limitation to the placement options due to the restriction of the area where the task can be placed. This might cause degradation in task's execution speed. Moreover, unless the less likely to be executed task is called, an overhead is paid for the configuration data of the first portion of the less likely to be executed task. The trade-offs between these limitations and keeping the system in an acceptable performance level is a matter of further research.

7. REFERENCES

[1] K. Papademetriou and A. Dollas, "A Task Graph Approach for Efficient Exploitation of Reconfiguration in Dynamically Reconfigurable Systems," in *Proc. of the IEEE Symposium on Field Programmable Custom Computing Machines*, April 2006.

[2] Z. Li, "Configuration Management Techniques for Reconfigurable Computing," Ph.D thesis, Northwestern University, June 2002.

[3] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Physically-aware HW-SW Partitioning for Reconfigurable Architectures with Partial Dynamic Reconfiguration," in *Design Automation Conference*, June 2005, pp. 335–340.

[4] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan, "A Self-reconfiguring Platform," in *Proc. of the International Conference on Field Programmable Logic and Applications*, September 2003, pp. 565–574.

[5] R. Dick, D. Rhodes, and W. Wolf, "TGFF: Task Graphs For Free," in *Proc. of the International Workshop on Hardware/Software Codesign*, April 1998, pp. 97–101.