A Parallel Configuration Model for Reducing the Run-Time Reconfiguration Overhead

Yang Qu[1], Juha-Pekka Soininen[1] and Jari Nurmi[2]
[1]Technical Research Centre of Finland (VTT), Kaitoväylä 1, FIN-90571 Oulu, Finland
Yang.Qu@vtt.fi
[2] Tampere University of Technology, Korkeakoulunkatu 10, Tampere, Finland

Abstract

Multitasking on reconfigurable logic can achieve very high silicon reusability. However, configuration latency is a major limitation and it can largely degrade the system performance. One reason is that tasks can run in parallel but configurations of the tasks can be done only in sequence. This work presents a novel configuration model to enable configuration parallelism. It consists of multiple homogeneous tiles and each tile has its own configuration SRAM that can be individually accessed. Thus multiple configuration controllers can load tasks in parallel and more speedups can be achieved. We used a prefetch scheduling technique to evaluate the model with randomly generated tasks. The experiment results reveal that in average using multiple controllers can reduce the configuration overheads by 21%. Compared to best cases of using multiple tiles with a single controller, additional 40% speedup can be achieved using multiple controllers.

1. Introduction

To build efficient systems and extend product life, reconfigurable logic is seen as an important type of unit in the design of System-on-Chip (SoC) because they are capable of accelerating computation-intensive tasks while achieving high silicon reusability through run-time reconfiguration (RTR). The RTR means the circuit or a portion of it can be reconfigured while the system is running. Such circuit is referred as dynamically reconfigurable hardware (DRHW).

However, the RTR does not come without a cost. The configuration itself results in extra overhead, e.g. configuration latency and power consumption, which can degrade the overall performance. Novel approaches either at the device level or at the application level have been proposed [1,2] to reduce or hide the configuration latency, but these approaches cannot solve the problem that configuration limits the task parallelism due to the fact that traditional devices use only one configuration controller. The effect is shown in Figure 1(c), where the task 3 has to be delayed because its configuration cannot start earlier. However, if we could use two controllers to reconfigure different portions of the DRHW in parallel, execution of the task 3 can start immediately after its predecessor finishes, as depicted in Figure 1(d). If we assume that the execution time and the configuration time are equal, using two controllers can speedup the system by 25% compared to using only one controller in the case.

In this paper, we present a novel configuration model that uses multiple configuration controllers to reduce the effects of the reconfiguration latency. The initial device model is presented in [3]. The concept is to divide the configuration-SRAM into separate individual sections and use multiple configuration controllers to reconfigure these sections in parallel. Therefore, more than one task can be loaded simultaneously and the effects of configuration latency are reduced. Using extra controllers introduces extra dimensions to the design space and it makes the devices more expensive. In this work, we use a prefetch scheduling technique to explore the design alternatives.

The paper is organized as follows. The related work is summarized in the next section. Section 3 describes the configuration model. The prefetch scheduling is presented in section 4. The design space and a prototype tool to support evaluation are presented in section 5. Different case studies have been carried out, and the results are presented in section 6. The conclusions are in section 7.

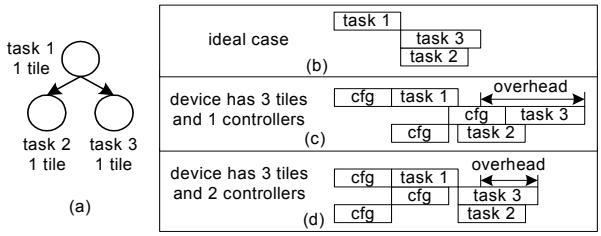

Figure 1. Comparison of schedules

3-9810801-0-6/DATE06 © 2006 EDAA

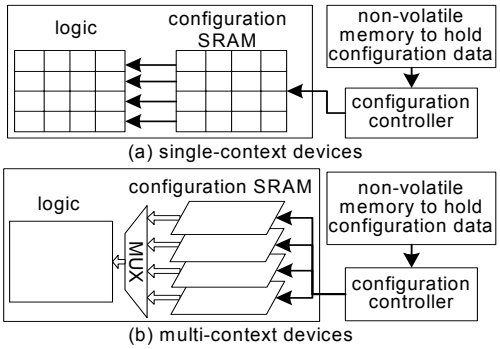

Figure 2. Traditional configuration models

2. Related research

Majority of DRHW devices are using SRAM-based technology. Such devices consist of the circuit and the configuration-SRAM whose outputs are connected to the circuit and whose values continuously control the circuit, as shown in Figure 2(a). Reconfiguration is realized by altering contents of the configuration-SRAM. Multi-context devices [1] have multiple configuration planes and a multiplexer (MUX) controls which plane is used, as shown in Figure 2(b). Switching from one to another takes only one clock cycle, thus configuration latency is significantly reduced. However, such devices require as many times the configuration-SRAM needed in single context device as the number of contexts, and caching tasks consumes lots of energy. Partial reconfigurable devices [6] use a similar configuration model as in single-context devices, but it can selectively change the content of a portion of the configuration-SRAM. Thus, configuration of a task takes much shorter time. However, these models do not support configuration parallelism, namely altering different portions of the configuration-SRAM in parallel. Our model is different because the entire configuration-SRAM is divided into sections and multiple sections can be accessed in parallel by multiple configuration controllers, as shown in Figure 3. The novelty of the model is its ability to enable configuration parallelism, so task parallelism can be more efficiently exploited.

Prefetch scheduling can effectively hide the configuration latency by loading tasks before they are needed. In [2], Hauck presents a prefetch scheduling technique for single-context reconfigurable coprocessors. Scheduling instructions are inserted properly to hide configuration latency from software applications. For partially reconfigurable devices, different prefetching techniques have been developed to reduce the overall reconfiguration overhead. In [4], Resano et al. introduce a hybrid prefetch scheduler, in which the scheduling is computed at design time but the configuration decisions are made at run time. In [5], a two-dimensional, multi-rate cyclic scheduling algorithm is developed. Tasks are scheduled at design time based on real-time constraints and reconfiguration overhead information. Our prefetch scheduling technique is similar to these approaches in a way that the tasks are modeled as a directed acyclic graph (DAG) and scheduling is performed at the design time. The main difference is that our prefetch scheduling technique targets on a different configuration model, which enables different tasks or pieces of a large task to be loaded in parallel.

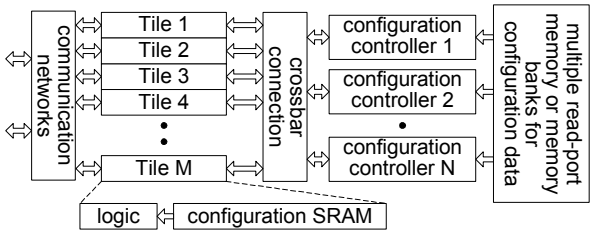

Figure 3. Our configuration model

3. The Device Model

Our DRHW device, as depicted in Figure 3, is based on a commercially available SRAM-based FPGA [6] that supports partial reconfiguration. To enable using multiple controllers to reconfigure different portions of the device in parallel, we propose the following changes. Firstly, the DRHW device consists of a number of continuously connected homogeneous tiles, and each tile consists of the circuit and its own configuration-SRAM that controls the circuit. A task that requires m tiles of resources can use any set of m connected tiles. This is referred as task relocation [7], which can increase the reusability of the device. Secondly, a crossbar connection is used to connect the configuration-SRAMs of the tiles to a number of parallel configuration controllers. The crossbar ensures that any configuration-SRAM can be accessed by any controller but only one at a time. Thus, reconfigurations can be performed in parallel on different tiles. Finally, the memory that holds the configuration data of all tasks should have multiple read ports or consist of a number of separate memories that allow parallel access, so different controllers can read data at the same time.

Our model is not meant for standalone devices like Virtex FPGA platforms. Instead, we assume it is a part of SoC platforms. Data transfers among tiles and between tiles and the rest of SoC are realized through communication networks, which is not a part of the reconfigurable logic, as shown in Figure 3. Thus, routing contention can be avoided when tasks are relocated. In this work, we concentrate mainly on theoretical analysis of the model.

4. Prefetch Scheduling Technique

The principle of the prefetch scheduling is to load tasks whenever there are tiles and configuration controllers available, instead of after tasks become ready. Each task has a priority, which represents the urgency of configuration of the task. The task with the highest priority is scheduled first upon free resources are available. The priority function consists of three elements: the mobility, the gap and the delay. The mobility shows the urgency of execution, and a low mobility value means high priority. The gap shows how much benefit a task can get if its configuration immediately starts. A low gap value means high priority. The delay shows how many additional configurations have to be delayed if configuration of the task cannot start immediately. It is obvious that prefetching successors prior to predecessors does not bring benefits. Therefore, a task with more successors has a high delay value, which means high priority.

The pseudo code of the algorithm is shown in Figure 4. The algorithm iterates starting from the scheduling time (s-time) 1 and stops when all tasks are scheduled, as in (1)-(3). In each iteration, priorities of all the unscheduled tasks are calculated, as in (4). The algorithm searches the DRHW device for any pair of a free tile and a free configuration controller. Scheduling of configurations and executions is continuously performed as long as such a pair exists, as in (5)-(16). Upon scheduling, candidate tiles are selected for the task and configuration of the task is then scheduled, as in (7)-(9). Due to prefetch, a task might not be ready when its configuration has finished. The ready time is then calculated, and execution of the task is scheduled upon that time, as in (10)-(11). The s-time is increased in each iteration, as in (17). Then, free resources might be found at the new s-time.

Brief explanations of the important functions are as follows. The function *Calculate_RTR_Priority* calculates priorities of the tasks and sorts them according to the priority values. The value is calculated as *a/mobility+b/gap+c*delay*, where *a*, *b* and *c* are weights. The mobility is calculated as *(ALAP s-time) – (ASAP s-time) +1*. The gap is calculated with the assumption that configuration of the task starts at the s-time and the task can start to run at the *ASAP s-time*. Its value is equal to *(ASAP s-time) – (configuration_end s-time)*. Offset values are added to the gap values to make all of them positive. The delay value is equal to the normalized value of the total number of successors of the task. The function *Search_for_Free_Res* returns *TRUE* if a pair of a free tile and a free configuration controller is found at the *s-time*. The function *Search_for_Candiate_Tiles(tiles, m)* returns the *m* connected tiles on which the configuration can finish within the shortest time. The function *Schedule_Configuration* uses a resource-constraint ASAP scheduling approach to schedule configuration of the task onto the selected tiles. Its return value represents when configuration is finished. The function *Schedule_Task* sets the task to run at the *run_time*. The function *Insert_Tasks(V,L)* puts the tasks in *V* to the List *L*. The function *First(L)* returns the first task in *L*. The function *Delete(L, T)* deletes the element *T* from *L*. The function *Required_Tiles(T)* returns the number of required tiles of the task *T*. The function *Calculate_Ready_Time* returns the earliest *s-time* at which both configuration of the task and executions of all its predecessors have finished.

```
Insert_Tasks (V,PList);                                  -- (1)
s_time = 1;                                              -- (2)
while(PList ≠ φ ) do                                     -- (3)
  Calculate_RTR_Priority (PList,s_time);                 -- (4)
  while(Search_for_Free_Res(s_time) ≠ 0) do              -- (5)
    task = First(PList);                                 -- (6)
    m = Required_Tiles(task);                            -- (7)
    Search_for_Candidate_Tiles(tiles,m);                 -- (8)
    Schedule_Configuration(tiles,task);                  -- (9)
    run_time = Calculate_Ready_Time(task);               -- (10)
    Schedule_Task(tiles,task,run_time);                  -- (11)
    Delete(PList, task);                                 -- (12)
    if(PList = φ) then                                   -- (13)
      break;                                             -- (14)
    end if;                                              -- (15)
  end while;                                             -- (16)
  s_time = s_time +1;                                    -- (17)
end while;                                               -- (18)
```

Figure 4. Prefetch scheduling algorithm

5. Evaluation Approach

We use the prefetch scheduling technique with randomly generated tasks to evaluate the device model. Independent parameters are defined for the tasks and the devices, so that different design space can be easily explored by simply tuning the parameters.

5.1. Application Model and Design Space

Tasks that are mapped onto the DRHW are modeled as a directed acyclic graph, *G(V,E)*, where $V = \{j_1, j_2, \ldots, j_n\}$ is a set of nodes that represent the tasks and *E* is a set of edges that represent the dependence of the tasks. A task is ready to run when all of its predecessors finish. There are two attributes for a task *j*, execution time, EX_j, and the number of required resources, RR_j.

A device is modeled using a set of independent parameters: 1) *ST*, the size of a tile, 2) *NT*, the number of tiles, 3) *NC*, the number of controllers, and 4) *CL*, the configuration latency of a tile. It can be roughly calculated as: *tile bitstream size/(controller bitwidth * configuration clock frequency)*, where the tile bitstream size is related to

both the parameter *ST* and the granularity of the device. Compared to fine-grain logic, coarse-grain logic suffers less configuration latency because the main part to be reconfigured is the interconnection. Therefore, it is possible to use small *CL* values for coarse-grain logic and large *CL* values for fine-grain logic.

5.2. Prototype Tools for Design Space Exploration

A set of prototype tools has been developed in C++ for design space exploration. It has a graphical front-end tool for creating task graphs. Design parameters are given in a script file. Different scheduling algorithms can be selectively used in the tool, e.g. prefetch and non-prefetch. The toolset performs the scheduling for the tasks with the design parameters, generates text-based results and extracts overall reconfiguration overheads. Some parameters can be set to a number of different values, and the toolset will iteratively run the scheduling algorithms for each alterative. There is also a back-end viewer program to visualize the scheduling results.

6. Experimental Results

In the case studies, 10 DAGs were generated and there were 10 tasks in each DAG. Tasks have from 0 to 3 successors, but in average 1 successor. Two evaluations were carried out. The first one focused on studying the effects of using different number of tiles and different number of controllers. The second one focused on studying how well the configuration parallelism could be exploited when configuration time changed from a dominating factor to a less dominating factor. In the following context, we use (*NT*, *NC*) to refer to the device with *NT* tiles and *NC* controllers.

6.1. Exploration with different number of tiles and different number of controllers

In this experiment, the objective was to evaluate the effects of using more than one controller but with fixed configuration bandwidth per tile. Resource utilization was set as $\lceil RR_j / ST \rceil \in [1,3]$, and the average ratio was 2. We assumed the configuration latency was a dominating factor in this case, and its value was set to about half of the average execution time. Values of other parameters were set to satisfy the constraint:

$$\Sigma_{j=1}^{10}\left[CL*(\lceil RR_j/ST \rceil)/(10*EX_j)\right]=g\,, \qquad (1)$$

where $g = 0.5$.

Different devices were evaluated by setting the number of tiles, *NT*, to iterate from 3 to 10 and the number of controllers, *NC*, to iterate from 1 to 10. Because using more controllers than tiles does not bring benefit, the tool can automatically ignore the settings where *NT*<*NC*. The scheduling results showed that for the test cases speedup was saturated with 5 controllers, and thus results of more than 5 controllers were ignored in analysis.

We use the device (3, 1) as the reference device because this setting is the minimum requirement to map the generated tasks. Comparison of other devices and the reference device is presented in Figure 5. There are 8 sections in the graph with each section representing one setting of *NT* and each index within the section representing one setting of *NC*. As expected, the results show that more speedups could be achieved with using more tiles and more controllers. It is clear from the results that using single controller (*NT*, 1) can bring only limited speedups when compared to using multiple controllers. In single controller case speedups of the overall best case (10, 1) are in the range of [1.45, 2.07] with average value of 1.84. However, when multiple controllers are used, speedups of the overall best case (10, 5) are in the range of [1.55, 2.98] with average value of 2.24, equivalent to an additional 40% improvement in average.

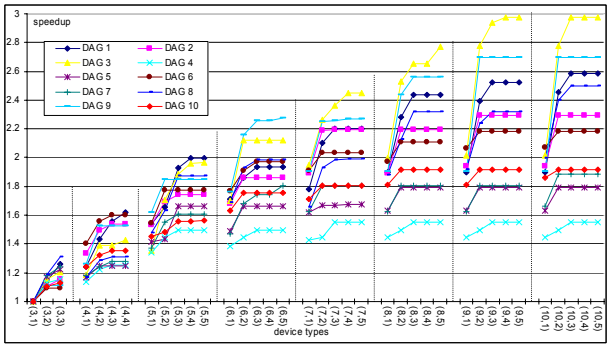

Figure 5. Speedups of tasks on different devices

In order to study how much additional speedup can be achieved with using more controllers, we compared the single controller case (*NT*, 1) with the best speedup (*NT*, 5). For devices with 3 tiles and 4 tiles, the best cases are (3, 3) and (4, 4). The comparison results show that in average 28.5% more speedup can be achievable with configuration parallelism. Although some individual cases show that speedups tend to be irrelevant to the number of tiles, in general configuration parallelism brings more speedups on devices with more tiles. The main reason is that when there are more tiles the resource-constraint scheduling is approaching to the timing-constraint scheduling, thus more configuration parallelism can be exploited using more controllers and so better performance is achieved.

We consider the configuration overhead as the difference between the prefetch scheduling result and the

ideal result, which have zero configuration latency, as shown in Figure 1. Configuration overheads of the DAGs are presented in Figure 6. The results show that the overheads decrease when more tiles are used. This is because when there are more tiles, it is more likely that the configuration of a task can be performed in parallel with the computation of other tasks and thus the effects of the configuration are reduced. When multiple controllers are used, the overheads can be reduced by about 21% in average of the best cases, (*NT*, 5) compared to (*NT*, 1). However, already with 2 controllers, the average reduction is about 16.7% and in some individual cases the overheads are reduced by about 50%.

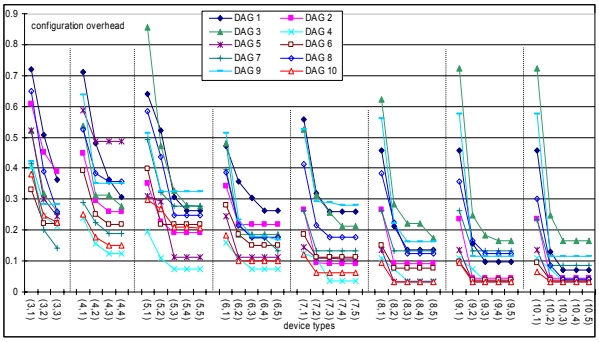

Figure 6. Configuration overheads of tasks on different devices

To justify the speedups, a primitive cost function of the area of the device is constructed. The cost is calculated as:

$$cost = \alpha*ST*NT + \beta*NC + \gamma*NT*NC, \quad (2)$$

where the first part refers to the area of the logic, the second to the area of the configuration controllers, the last to the area of the crossbar, and α, β, and γ are weight values that convert units to equivalent number of gates. These values, as shown in Table 1, are estimated based on the assumption that the basic configurable logic is LookUp-Table (LUT), the configuration controller is implemented as Direct Memory Access (DMA) controller, and each junction in the crossbar costs 26 gates. The configuration controllers support IBM Processor Local Bus interface to make system integration easier, but encryption and compression of bitstream are not supported yet.

Table 1. Values for the cost function

	ST	*α*	*β*	*γ*
values	300	8	2500	26

We use the ratio of *speedup/cost* to present the computation efficiency, and the normalized average results of the 10 DAGs are given in Figure 7. Although devices (3, 1), (4, 1) and (5, 1) provide much higher efficiency, the low speedups make them less attractive. Devices (6, 2), (6, 3) and (7, 2), as marked by the circles in Figure 7, show relatively high efficiency as well as high speedup. If the objective was to find an optimal solution for the 10 test cases, a device from this group could be selected.

It should be noted that our configuration model is not meant for competing with single controller devices that have high configuration bandwidth per tile. For example, a single 4*N*-bit controller tends to provide better performance than four *N*-bit controllers. This is because in former case high configuration bandwidth can always be exploited when configuration starts, but in latter case the configuration overhead can be reduced only if configuration parallelism exists.

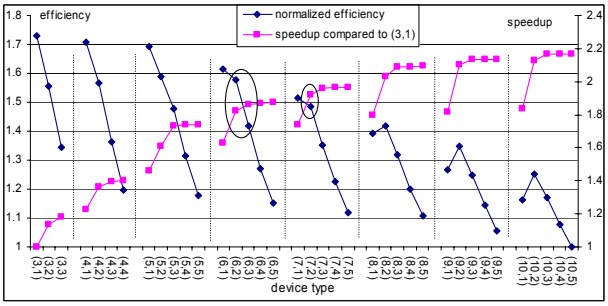

Figure 7. Normalized average efficiency and average speedups

6.2. Exploration on different ratios of execution time to configuration time

In this experiment, we focused on studying how well the configuration parallelism could be exploited when configuration time changed from a dominating factor to a less dominating factor. Five different cases were targeted and we used different *CL* values to distinguish them. The *CL* was set to satisfy the constraint (1), in which the value *g*, the ratio of the average configuration time to the average computation time, was specified to iterate in the set of [0.02, 0.05, 0.1, 0.2, 0.5]. Task models and the rest of settings were the same as described in section 6.1.

The scheduling results of the 10 DAGs are averaged for each setting of the *CL*, and all the averaged values are shown in Figure 8. The results clearly indicate that using multiple controllers is much more useful in the case where configuration time is a dominating factor, as shown in the setting of 0.5. In order to highlight the additional improvements that can be achieved only using multiple controllers, we compare the best of the multiple controller cases with the single controller cases and show the differences in Figure 9.

The 0.5 case shows additional 28.5% speedup can be achieved in average if multiple controllers are used, but in other cases the additional speedups that can be achieved

are well below 12%. In fact, the average of the maximally achievable speedups of the other four settings is only 3.8%. The low speedup is because when the configuration latency becomes less dominating there is more chance that configuration can be performed in parallel with computations of other tasks, thus the effects of reconfiguration are already reduced without using multiple controllers. This experiment reveals that using multiple controllers is much more useful when configuration latency is a dominating factor.

To some extent, this case study imitates the situation of mapping stream-based applications onto devices of different granularities. For example, an 8x8 pipelined IDCT algorithm requires 1581 configuration words when it is mapped onto a coarse-grain device, XPP20 [8]. It uses all the 20 ALU-PAEs. A similar implementation [6], which is capable of delivering the same throughput, can be mapped onto 16 slice columns of a fine-grain logic device, Virtex XC2V3000. The algorithm then occupies all the 2048 slices in those columns. However, the generated partial bitstream has a size of 74879 words, about 46 times larger than that in the coarse-grain device. The difference means much longer configuration time is needed in fine-grain logic devices, and our experiment indicates that using multiple configuration controllers could be more useful in such cases.

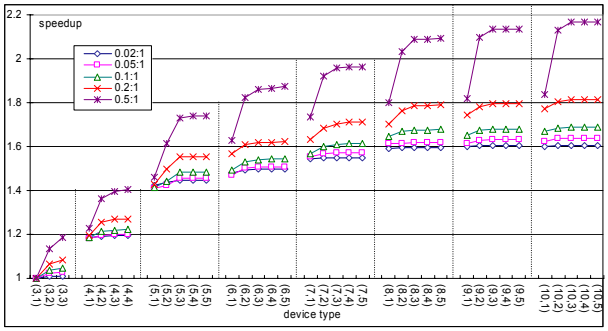

Figure 8. Average speedups for different ratios of configuration time to execution time

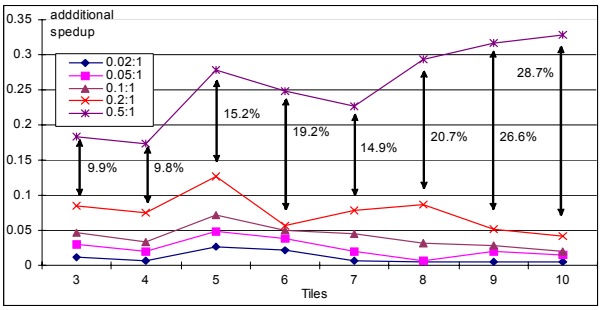

Figure 9. Additional speedups, using multiple controllers compared to using a single controller

7. Conclusions

Reconfigurable logic is an important design alternative because of the low NRE cost and the high silicon reusability. However, the configuration latency can largely degrade the system performance, because configuration usually takes significant amount of time and tasks that are mapped onto the reconfigurable logic can run in parallel but configurations of the tasks can only be performed in sequence.

New devices and scheduling approaches have been developed to solve the first problem. In this work, we have proposed a novel configuration model to solve the second problem. The model consists of multiple homogeneous tiles and each tile has its own configuration-SRAM that can be individually accessed. Thus multiple controllers can load tasks in parallel. Therefore, more task parallelism can be exploited and better performance can be achieved. We have used a prefetch scheduling technique and a set of randomly generated tasks to evaluate the model. The experimental results reveal that in the test cases using multiple tiles with single controller can speedup the system by 1.84 in average, but if multiple controllers are used additional 40% speedup can be achieved. With multiple controllers, configuration overheads can be reduced by 21% in average. The results also show that the approach is more useful when configuration time is relatively large compared to the execution time. In the future, we will continue with real-life tasks and run-time scheduling.

8. References

[1] H. Singh, et al, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications", IEEE Trans. Vol.49, No.5, pp.465-481, 2000.

[2] S. Hauck, "Configuration Prefetch for Single Context Reconfigurable Coprocessors", ACM/SIGDA International Symposium on FPGA, pp. 65-74, 1998.

[3] Y. Qu, et al, "Using multiple configuration controllers to reduce the configuration overhead", IEEE Proceedings of the 23rd Norchip conference, pp. 86-89, 2005.

[4] J. Resano, et al, "A Hybrid Prefetch Scheduling Heuristic to Minimize at Run-Time the Reconfiguration Overhead of DRHW", DATE'05, pp. 106-111, 2005.

[5] S. Li and N.K. Jha, "HW/SW Co-Synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs", ASP-DAC'02, pp. 345-352, 2002.

[6] Xilinx, datasheet and application notes, www.xilinx.com.

[7] O. Diessel, et al, "Dynamic scheduling of tasks on partially reconfigurable FPGAs", IEE Proc.-Comput. Digit Tech, Vol. 147, No. 3, pp. 181-188, 2000.

[8] PACT XPP Technologies, "Reconfiguration White Paper", www.pactcorp.com.