

Using Multiple Configuration Controllers to Reduce the Reconfiguration Overhead

Yang Qu¹, Juha-Pekka Soininen¹ and Jari Nurmi²

¹VTT Electronics, Kaitoväylä 1, Oulu Finland

Yang.Qu@vtt.fi

²Tampere University of Technology, Korkeakoulunkatu 10, Tampere Finland

Abstract:

This work presents a novel run-time reconfiguration model. It uses multiple configuration controllers instead of only one in traditional devices. The configuration SRAM is divided into several individual sections, and controllers can reconfigure different sections in parallel. Therefore, multiple tasks can be loaded simultaneously. Two static task schedulers are developed to evaluate the device. Already with two controllers, the overall configuration overhead can be reduced by about 40% using non-prefetch scheduling. When using prefetch scheduling, another 16.2% reduction can be achieved on average of all results.

1. Introduction

Reconfigurable logic is becoming an important unit in System-on-Chip (SoC) design due to their capabilities of providing higher performance than SW implementation and more flexibility than fixed-HW implementation. The run-time reconfiguration (RTR), which means the circuit or a part of it can be reconfigured while the system is running, is seen as a major advantage, since it can significantly increase the silicon reusability.

However, configuration itself results in extra overheads, e.g. delays and power consumption, which might degrade the overall system performance. Novel devices [1,2] have been developed to relieve the side effects by physically reducing the configuration latency. In [1], multi-context devices are studied. Such devices have multi-layer configuration planes, and switch from one to another takes only one clock cycle. However, tasks should be prefetched into the configuration planes and such devices require much more configuration SRAMs than other structures. Coarse-grain configurable logics [2] require much less configuration bits since the main part to be configured is the interconnection. Therefore, the configuration latency is much shorter compared to fine-grain configurable logics.

At the application design level, researchers have proposed a technique, called configuration prefetch [3] to reduce the side effect. The idea is to load a task before it is needed so the configuration latency is hidden from the overall system performance. The decision of when to fetch tasks is usually made at design time and different static scheduling approaches [3,4,5] have been studied.

In this paper, we present a novel configuration model, which uses configuration parallelism to reduce the

overhead. The idea 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 effect of configuration latency is reduced. In addition to the model, prefetch techniques have been studied to further improve the approach.

This paper is organized as follows. The introduction is presented in Chapter 1. Our configuration model and the related static scheduling techniques are presented in Chapter 2. Validation work and results are presented in Chapter 3. The conclusions are in Chapter 4.

2. Devices with Multiple Tiles and Multiple Configuration Controllers

Reconfigurable logic used in RTR fashion is referred as dynamically reconfigurable hardware (DRHW). In such devices, tasks are loaded at run time. When the size of the DRHW is large enough, more than one task can run simultaneously. However, traditional devices have only one configuration controller, and thus tasks are loaded in sequential fashion, which significantly limits the task parallelism especially when the configuration latency is a significant factor. This motivates us to introduce additional configuration controllers to enable different tasks or different pieces of a large task to be loaded in parallel.

2.1 Device Description

Our DRHW device, as depicted in Figure 1, is based on a commercially available SRAM-based FPGA [6], which has partial reconfigurability. It should be noted that the configuration SRAM refers to the memory cells whose outputs are connected to the circuit and whose values continuously control the circuit. Reconfiguration is realized by altering contents of the configuration SRAM.

In order to enable the use of multiple configuration controllers, we have made 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 configuration controller but only one at a time. Therefore, reconfiguration can be performed in parallel on different tiles. Finally, the memory, which holds the configuration data of all tasks, must have multiple read ports, so different configuration controllers can read data at the same time.

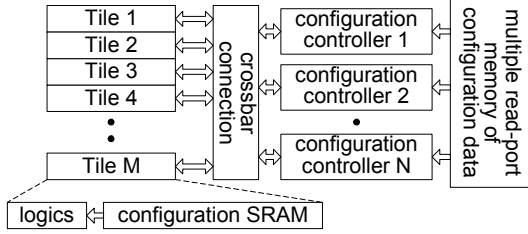


Figure 1. The configuration model

2.2 Scheduling Techniques

Two list-based task scheduling techniques, without and with prefetch, have been developed to evaluate the benefit of using multiple configuration controllers. We assume the tasks that are mapped onto the DRHW can be modeled as a directed acyclic graph (DAG), $G(V,E)$, where $V = \{j_1, j_2, \dots, j_n\}$ 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 cannot start before all of its predecessors finish. Each task has two attributes, execution time and the number of required resources.

The first scheduling technique does not consider task prefetch, and the goal is to evaluate the pure contribution of using more than one configuration controller. The pseudo code of the list-based scheduling algorithm is shown in Figure 2. The algorithm iterates starting from scheduling time (s -time) 1 and stops when all tasks are scheduled, as in (1)-(3). In each iteration, a priority list $PList$ is created for all the ready tasks, and higher priority tasks are scheduled first, as in (4)-(6). A ready task is a node that has no predecessor or all of its predecessors have finished execution at the s -time. Upon scheduling, candidate tiles are selected for a task and configurations of the task are then scheduled, as in (9)-(10). Because the task is already ready before scheduling, execution of the task can then be scheduled immediately after the configurations, as in (11)-(12). The s -time is increased in each iteration, as in (16). Other non-ready tasks may then become ready at the new s -time.

Brief explanations of the important functions are as follows. The function *Calculate_Ready_Priority* searches the unscheduled task list, $NSList$, for the ready nodes, put them into the $PList$, calculates their priorities, and sorts the $PList$. We use mobility [8] as the priority function in that it shows urgency of a task. The value of mobility is the difference between the As-Late-As-Possible (ALAP) scheduling result and the As-Soon-As-Possible (ASAP) scheduling result. The ALAP and ASAP are updated in each iteration via the function *ALAP_ASAP_Schedule(NSList)*. The function *Search_for_Candidate_Tiles(tiles, m)* returns the m connected tiles whose average earliest available time is the smallest. Look-ahead methods can produce more optimized solutions, but they are not included at the

moment. The function *Schedule_Configuration* uses a resource-constraint ASAP scheduling approach to schedule configurations of the *task* onto the *tiles*. Its return value represents when configurations are finished. It is expected that the starting time of configurations will be less delayed when more configuration controllers exist. 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 .

```

Insert_Tasks (V,NSList);           -- (1)
s_time = 1;                         -- (2)
while(NSList  $\neq \emptyset$ ) do       -- (3)
  ALAP_ASAP_Schedule(NSList,s_time); -- (4)
  Calculate_Ready_Priority (NSList, PList,s_time); -- (5)
  while(PList  $\neq \emptyset$ ) do     -- (6)
    task = First(PList);          -- (7)
    m = Required_Tiles(task);      -- (8)
    Search_for_Candidate_Tiles(tiles,m); -- (9)
    conf_time=Schedule_Configuration(tiles,task); -- (10)
    run_time = conf_time + 1;      -- (11)
    Schedule_Task(tiles,task,run_time); -- (12)
    Delete(PList, task);          -- (13)
    Delete(NSList, task);         -- (14)
  end while;                       -- (15)
  s_time = s_time + 1;           -- (16)
end while;                          -- (17)

```

Figure 2. Scheduling algorithm without prefetch

The second scheduling technique takes prefetch into account, and the goal is to further improve the benefit of using multiple configuration controllers. The principle is to load tasks whenever there are tiles and configuration controllers available, instead of after task become ready. Each task has a priority, which represents the urgency of configurations 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 configurations immediately start. If the estimated ready time is close to the end of configurations, the task has a low gap value, which means high priority. The delay shows how many configurations have to be delayed if configurations of the task do not start immediately. The idea is that prefetching successors prior to predecessors does not bring benefits. A task with more successors has a high delay value, which means high priority.

The pseudo code of the algorithm is shown in Figure 3. The algorithm iterates starting from 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 configurations of the task are then scheduled, as in (7)-(9). Due to prefetch, a task might not be ready when its configurations have 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.

Functions that have the same names as in the without prefetch case are explained in earlier sections. Brief description of the rest is 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 the a , b and c are weights. The mobility is calculated as $(ALAP\ s\text{-time}) - (ASAP\ s\text{-time}) + 1$. The gap is calculated with the assumption that configurations 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 $(ASAP\ s\text{-time}) - (configurations\ end\ s\text{-time})$. Offset values are added so all the gap values are 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 *Calculate_Ready_Time* returns the earliest s -time at which both configurations of the task and executions of all its predecessors have finished.

```

Insert_Tasks (V,PList);           -- (1)
s_time = 1;                       -- (2)
while(PList  $\neq$   $\phi$ ) do           -- (3)
  Calculate_RTR_Priority (PList,s_time); -- (4)
  while(Search_for_Free_Res(s_time)  $\neq$  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 =  $\phi$ ) then           -- (13)
      break;                       -- (14)
    end if;                         -- (15)
  end while;                       -- (16)
  s_time = s_time + 1;           -- (17)
end while;                         -- (18)

```

Figure 3. Scheduling Algorithm with Prefetch

2.3 Design Space Exploration and Tools

The overall configuration overhead is related to a set of different parameters: 1) ST , the ratio of the average task size to the tile size, 2) GD , the graph dependence, 3) NT , the number of tiles, 4) NC , the number of controllers, and 5) CL , the configuration latency of a tile.

A set of prototype tools has been developed in C++ to explore the design space. It uses the *dotty* [9] as the front-end tool to receive a task graph. The ST and the GD are specified within the graph, while other design parameters are given in a script file. The toolset performs the two scheduling algorithms for the task graph with the design parameters, generates text-based scheduling 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 alternative. At the back end, there is a viewer program to visualize the scheduling results.

3 Case Study

In this paper, we concentrate on evaluating the benefits of using multiple configuration controllers on devices of different configuration latencies. The complete design space will be studied in later work.

We randomly generated 10 task graphs with 10 tasks in each graph. The ST was set to 2, and the size of a task was randomly generated within the range from 1 tile to 3 tiles. The GD was specified in a way that a task might have from 0 to 3 successors, randomly generated. The NT was set to a fixed value, 6. Because using more configuration controllers than tiles does not bring benefits, the NC was set to iterate from 1 to 6. Five different values were used when setting the CL . The values were adjusted so the ratio of average configuration time of tasks to average execution time of tasks was set to 10, 5, 1, 0.2 and 0.1 respectively.

The 10 task graphs are scheduled for each combination of the CL and the NC , and during the analysis the scheduling results of the 10 task graphs are averaged for each combination. The averaged configuration overheads are depicted in Figure 4. As expected, the overall performance is improved when the number of configuration controllers is increased. Detailed analyses of the overheads are shown in Table 1 and Table 2.

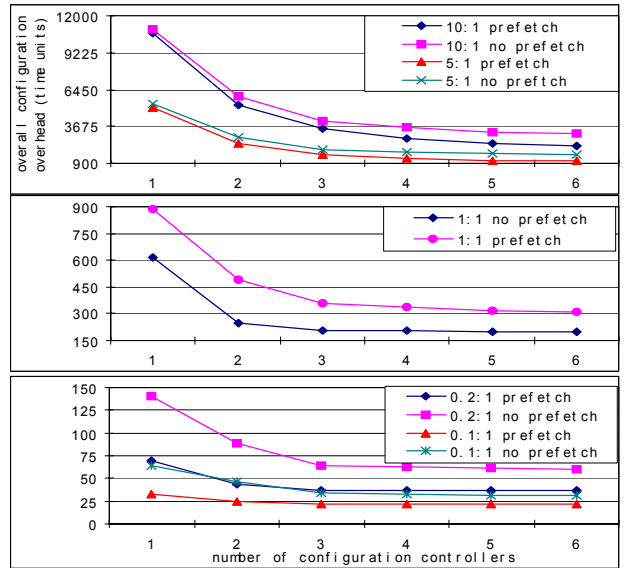


Figure 4. Effects of increasing the number of configuration controllers

The results in Table 1 show the average reductions of the overall configuration overheads when using more than one controller. The prefetch scheduling results are not listed, since they contain the contributions from both the device model and the prefetch algorithm. Most individual results are less than 15% from the average values, but at the settings of 0.1 and 0.2 there are a few results that are clearly far away from the average values. At the settings of 10, 5 and 1, where devices suffer long configuration latency, the results show that the average improvements of increasing controllers are almost irrelevant to the configuration latency. On the other hand, at the settings of 0.1 and 0.2, which refer to short configuration latency, we speculate that using more

controllers becomes less beneficial. This is because when the configuration latency becomes less dominating there is more chance that its effect can be reduced even without using extra controllers, as shown in Figure 5, where the execution of the task 2 can hide more portion of the configuration of the task 3 in the case of short configuration latency.

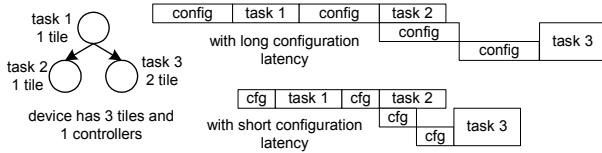


Figure 5. Difference between slow and fast configuration speed (without prefetch)

The significance of using more controllers is clearly shown in Table 1. Already with two controllers, the overall configuration overheads are reduced by about 40%, and particularly the reduction is about half at the settings of 10, 5 and 1. However, it is obvious that the speedups do not increase linearly. There are mainly two reasons of these nonlinear effects. The first is that the tasks require two tiles in average in our case, which limits the benefit when using more than two controllers to load a single task. The second is that in the DAGs most tasks have less than two direct successors, so it is less likely that all the controllers are necessary. It should be noted that the cost is a polynomial function of the number of controllers because of the crossbar connection and the requirement of multiple read-port memories. Therefore, using more controllers might bring more cost than benefit.

Table 1. Average speedups when compared with single configuration controller case (without prefetch)

ctrls	0.1:1	0.2:1	1:1	5:1	10:1
2	27.9%	37.1%	44.6%	46.3%	45.8%
3	46.1%	53.9%	59.5%	63.1%	62.9%
4	47.6%	54.8%	62.1%	67.2%	67.5%
5	49.2%	56.2%	64.5%	69.5%	70.3%
6	49.9%	56.9%	64.6%	70.5%	71.3%

Table 2. Average speedups of using prefetch compared with not using prefetch

ctrls	0.1:1	0.2:1	1:1	5:1	10:1
1	48.7%	50.6%	30.3%	4.7%	2.3%
2	46%	50.6%	49.4%	16.8%	11.5%
3	36.9%	42.6%	42.6%	21.7%	13.6%
4	35%	41%	39.4%	27.3%	21.9%
5	33%	39.1%	36.2%	31.2%	27.3%
6	32%	38.1%	35.9%	31.3%	28.9%

Average speedups of using the prefetch technique compared with not using prefetch have been presented in Table 2. If we further average all the results, the prefetch scheduling achieves about 16.2% reduction of the overall configuration overhead when compared to the non-prefetch scheduling. There are a few individual cases where the speedups are well above 90%. The average results reveal that when the configuration time is a

dominating factor, with more controllers the prefetch scheduling can produce better results, as shown at the settings of 10 and 5. It is also interesting to see the limitations of using only one controller at these two settings, where prefetch scheduling barely produces any speedup. However, on devices of short configuration latency, using prefetch scheduling always tends to bring significant speedups, irrelevant to the number of additional controllers.

The experimental results reveal that using multiple configuration controllers is more beneficial on devices of long configuration latency. In addition, prefetch scheduling should always be used as long as the execution order of tasks is known beforehand.

4. Conclusions

Reconfigurable logic is an interesting design alternative because it can achieve better performance than software implementation and deliver a certain degree of flexibility, which cannot be achieved in fixed hardware. The flexibility comes from the fact that the device can be reconfigured when necessary. However, the configuration latency degrades the system performance.

In this work, we proposed a new configuration model, which contains multiple configuration controllers instead of only one in the traditional devices. The configuration SRAM is divided into several individual sections and the multiple controllers can reconfigure different sections at the same time. A static scheduling technique without prefetch has been developed to evaluate the new model. A number of randomly generated tasks are used. The improvement is significant. With two configuration controllers, the overall configuration overhead can be reduced by about 40%. In addition, a prefetch scheduling technique is developed to further improve the benefit. Compared to the without prefetch case, it achieves another 16.2% reduction of the overall configuration overhead in average. In the future, more studies will be carried out to explore the complete design space.

References

- [1] H. Singh, et al, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications", IEEE Trans. Vol. 49, Issue. 5, pp.465-481, 2000.
- [2] Pact XPP Tech., XPP Datasheets, www.pactcorp.com.
- [3] S. Hauck, "Configuration Prefetch for Single Context Reconfigurable Coprocessors", ACM/SIGDA International Symposium on FPGA, pp. 65-74, 1998.
- [4] 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.
- [5] 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.
- [6] Xilinx, Virtex2 datasheet, 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] D.D. Gajski, et al, "High-level synthesis: Introduction to chip and system design", Kluwer Academic Publishers, 1997.
- [9] S. North and E. Koutsofios, "Applications of graph visualization", Proc. Graphics Interface, pp. 235-245, 1994.