Online Scheduling for Multi-core Shared Reconfigurable Fabric

Liang Chen, Thomas Marconi and Tulika Mitra
School of Computing
National University of Singapore
{chenliang,marconi,tulika}@comp.nus.edu.sg

***Abstract*—Processor customization in the form of application-specific instructions has become a popular choice to meet the increasing performance demands of embedded applications under short time-to-market constraints. Implementing the custom instructions in reconfigurable logic provides greater flexibility. Recently, a number of architectures have been proposed where multiple cores on chip share a single reconfigurable fabric that implements the custom instructions. Effective exploitation of this reconfigurable fabric requires runtime scheduling of the tasks on the cores and allocation of reconfigurable logic for custom instructions. In this paper, we propose an efficient online scheduling algorithm for multi-core shared reconfigurable fabric and show its effectiveness through experimental evaluation.**

I. Introduction

The emergence of multi-cores in recent years has created the opportunity for innovative application-specific instruction set processors (ASIPs) architectures. In the simplest form, the architecture can contain a set of homogeneous base cores where each core has a dedicated reconfigurable fabric (see Figure 1(a)) for implementation of custom instructions [1]. The user can thus create a heterogenous multi-core at runtime by simply extending the instruction-set architecture (ISA) of each core with different custom instructions. However, sharing a single reconfigurable fabric among limited number of cores (between 2 to 4) can achieve better utilization of resources and hence improved acceleration of the application (see Figure 1(b)). Moreover, a combined shared fabric can accommodate much larger custom instructions for individual tasks than is possible with fragmented private fabrics. Recently, researchers have proposed a number of architectures [2], [3], [4], [5] where the reconfigurable fabric is shared among a group of cores. A crucial issue in such architectures is the online scheduling of the tasks.

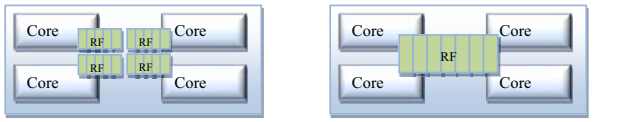
Online scheduling for both multi-cores and reconfigurable logic have been studied extensively in literature [6], [7], [8], [9], [10]. In our context, however, a task requires both the base processor core and the reconfigurable logic (implementing custom instructions) to execute. In multi-core architectures with dedicated reconfigurable fabric, the scheduling problem is exactly identical to that of a homogeneous multi-core architecture as the reconfigurable logic does not impose any additional constraint. On the other hand, in traditional partially reconfigurable FPGAs, the schedulers only need to place hardware accelerator tasks in the fabric without any consideration for the processor availability [7]. Scheduling both the software component on processor core and the hardware component in reconfigurable logic presents a difficult challenge. Moreover, any online scheduler should minimize its runtime overhead. Our goal is to design an efficient online scheduler for multi-core shared reconfigurable logic.

Previous research on shared reconfigurable fabric in multi-cores have either focused on offline scheduling [4] or task scheduling in the reconfigurable fabric [5]. To the best of our knowledge, ours is the first approach that considers runtime scheduling of real-time tasks accelerated with custom instructions on multi-core platforms with shared reconfigurable logic. Our online scheduler overcomes the dual constraint of the availability of the processor and the fabric by intelligently planning ahead and making resource reservation for the future. Our extensive experimental evaluation with tasks extracted from real-world applications confirms the effectiveness of our online scheduling algorithm.

Related work: Sharing the reconfigurable fabric has become a trend with the emergence of resource sharing for multi-core processors. In [4], the authors compare the shared and private reconfigurable fabric architectures, and show significant acceleration achieved by using shared reconfigurable fabric. Similar architectures for multi-core system with shared reconfigurable fabric have been proposed in [2], [3], [5]. However, [2], [3], [4] consider offline task scheduling. The only work that considers runtime scheduling in this context is [5]; however, their approach using minority game theory does not consider underlying realistic reconfigurable fabric model.

II. Problem formulation

We assume a multi-core architecture where the cores share a single large reconfigurable fabric to implement the custom instructions as shown in Figure 1(b). This fabric can be partially reconfigured at run-time.



a) Multi-core with private reconfigurable fabrics b) Multi-core with shared reconfigurable fabric

Fig. 1. Multi-core system with reconfigurable fabric (RF).

We assume that a sequence of independent tasks arrive in the system. A task may or may not require support for custom instructions. If a task employs custom instructions, it requires

978-3-9810801-8-6/DATE12/©2012 EDAA

both the processor core and part of the reconfigurable fabric for execution. A task without any custom instructions simply requires a free processor core. Thus we can define a task T_i as a 4-tuple (a_i, e_i, d_i, w_i) where

- a_i: the arrival time of the task
- e_i: the expected execution time of the task
- d_i: the deadline constraint for the task
- w_i: the area required for custom instructions

We assume hard deadlines, i.e., a task that cannot meet its deadline is considered to be rejected by the system. In this work, we assume 1D area model for the reconfigurable fabric. That is, the reconfigurable fabric is divided into equal sized columns and a column is the basic unit for partial reconfiguration. Thus w_i denotes the number of columns required to implement the custom instructions corresponding to task T_i. If a task T_i does not require custom instructions, then $w_i = 0$. We further assume that we have P processor cores and the reconfigurable fabric contains W columns. We employ non-preemptive scheduling policy to avoid repeated reconfigurations of the fabric due to preemptions.

The job of the online scheduler is to schedule the tasks and place the the custom instructions for that task in the reconfigurable fabric. Thus the scheduler determines the starting time s_i of execution of a task T_i and the starting column x_i where the custom function units (CFUs) of T_i are placed in the reconfigurable fabric. The scheduler needs to satisfy the following constraints.

- **Arrival time and deadline constraint:** Arrival time a_i and deadline d_i determine the time bounds for execution of task T_i. $\forall T_i : \ s_i \geq a_i \text{ AND } s_i + e_i \leq d_i$
- **Processor constraint:** At any point in time, we can have at most P tasks in execution because there are P processor cores: $|S_t| \leq P$ where $S_t = \{T_i | s_i \leq t \leq s_i + e_i\}$
- **Boundary constraint:** The CFUs corresponding to a task T_i should be placed inside the reconfigurable fabric.

$$\forall T_i : \ x_i + w_i \leq W$$

- **Disjoint placement constraint:** The placement of CFUs of different tasks that execute in parallel should not overlap with each other. That is, $\forall T_i, T_j$ where $j \neq i$

$$[x_i + w_i \leq x_j] \vee [x_j + w_j \leq x_i] \vee [s_i + e_i \leq s_j] \vee [s_j + e_j \leq s_i]$$

If the scheduler cannot find a suitable schedule and placement of a task that satisfy the above constraints, then the task is considered rejected. *The goal of the online scheduler is to minimize the rejection rate of the tasks.*

III. Online Scheduler

A. Drawbacks of priority-based scheduling

Most online scheduling algorithms employ some form of task priority for scheduling. A widely used priority function is based on deadline and the scheduling strategy is called **earliest-deadline-first (EDF)** in which the task in the queue with the closest deadline is scheduled first. This works well in a system with only processor constraint (such as multi-core without reconfigurable logic) because the highest priority task will always be able to execute once a processor becomes free. Unfortunately, enforcing the highest-priority task to be scheduled first can have negative effect on the scheduling quality in our system. When a task leaves the system, it may not be possible to schedule the highest priority task T in the queue due to the unavailability of reconfigurable logic. Now T will prevent all the lower priority tasks in the queue from being scheduled even if they are eligible.

A well-known fix to this problem in reconfigurable computing is the **EDF-NF (Next-Fit)** scheduling policy [7], [8], [9]. Here, if the highest priority task cannot be scheduled immediately (due to reconfigurable logic constraint), the scheduler scans through the priority queue till it finds a task that can be scheduled at current time (or fails to find any schedulable task in the queue). However, EDF-NF scheduling strategy has a different problem. Once a lower priority task starts execution, it may prevent the higher priority task to be scheduled by occupying a processor and reconfigurable logic.

B. Co-scheduling

Instead of simply scheduling one task at a time, we can schedule all the tasks in the queue together to obtain locally optimal scheduling solutions. We call this strategy of scheduling multiple tasks at a time **co-scheduling**. Rather than simply scheduling and placing the tasks in the reconfigurable fabric in a pre-defined order (e.g., EDF), we plan in advance the schedule and placement of the tasks in the queue and reserve resources accordingly.

Finding the optimal schedule (i.e., the one with minimal rejection rate) for all the tasks in the queue can be modeled as **2D rectangular strip packing problem (2D-SPP)** with additional processor and deadline constraints. In 2D-SPP, N rectangles need to be packed in a rectangular bin with limited height and width such that the rectangles do not overlap. In our context, the width corresponds to reconfigurable logic area and the height corresponds to the time.

Unfortunately, 2D-SPP is NP-hard problem and finding the optimal solution for 2D-SPP (known as *exact packing*) with large number of rectangles (tasks) is not computationally feasible specially for online scheduling [11], [12]. Moreover, even with 2D-SPP, we only get a local optima for the current tasks in the queue as the behavior of the future tasks are unknown to us. Thus, there is not much incentive to apply 2D-SPP for our problem. Nevertheless, 2D-SPP can provide a good target for our online scheduler.

C. Sliding window-based scheduling framework

We now present our sliding window-based scheduling framework that employs co-scheduling. The arriving tasks are queued according to their priorities (EDF). Every time a new task arrives, the queue is updated. Our scheduler focuses on the first K entries in the queue, which we call the **window**. The co-scheduler attempts to schedule all the tasks in the window together. The scheduling decisions (starting time s_i

and the starting column x_i for task T_i) for all the tasks in the window are maintained in a reservation list. The execution list maintains information about the currently executing tasks. Once the clock reaches the starting time s_i of a task T_i in the reservation list, T_i can start execution.

The co-scheduler is triggered only when the window gets updated. The window is updated either when a new task arrives and gets queued inside the window or when a task within the window starts execution (and the window slides to include the next higher priority task in the queue). In the worst case, co-scheduling is triggered two times per task. We now describe two co-scheduling algorithms: a heuristic-based approach and an exact solution based on 2D-SPP.

D. Co-scheduling heuristic

Algorithm 1: Sliding window scheduler with heuristic

```
triggered by window_update = True
Co_schedule(execution_list, window)
1  K := number of tasks inside window; reservation_list := NULL;
2  for i ← 1 to K do
3      MER := MER_list_creation(execution_list, reservation_list);
4      if (res := search_list(MER, window[i]))! = NULL then
5          insert_reservation_list(res);
6  window_update := False;

triggered by clock reaching starting time of T in reservation list
Start_execution(T)
1  delete_reservation_list(T); insert_execution_list(T);
2  if (queue − window) != NULL then
3      update_window(); window_update := True;

triggered by departure of executing task T
Finish_execution(T)
1  delete_execution_list(T);

triggered by arrival of task T
Insert_queue(T)
1  insert_queue(T);
2  if T ∈ window then
3      window_update := True;
```

Algorithm 1 presents a sliding window-based approach where we employ a heuristic for co-scheduling. Once the window gets updated due to arrival of a task inside the window or starting of execution of a task within the window, the co-scheduler is triggered. The co-scheduler generates future resource reservation for all the tasks in the window. It goes through the tasks in the window in order of their priorities (EDF). For each task T in the window, the algorithm identifies all the maximal empty rectangles (MERs) available for scheduling T taking into consideration the currently executing tasks and the tasks for which reservations already exist. The MERs are generated using the algorithm proposed in [10]. If multiple MERs are available that satisfy all constraints, then we can employ different strategies such as first fit, best fit or worst fit to choose one MER.

E. Co-scheduling with 2D-SPP

To evaluate our heuristic, we also design a co-scheduling algorithm based on 2D-SPP that provides locally optimal schedule for the window. The optimal packing is identified with essentially a branch and bound algorithm as described in [11] and [12]. Algorithm 2 presents the 2D-SPP based co-scheduling algorithm. As the updates of the window are identical to Algorithm 1, we do not include them again. The goal is to find the solution with minimal rejection rate. As the complexity grows exponentially with the number of tasks (rectangles to be packed), we limit the window size to 6 tasks.

Algorithm 2: 2D-SPP based co-scheduling

```
triggered by window_update = True;
Co_schedule(exec_list, window)
1  K := number of tasks inside window; reserv_list := NULL;
2  Global_min_reject = K;
3  DS_Exp(exec_list, reserv_list, window);

DS_Exp(exec_list, reserv_list, window)
4  MER := MER_list_creation(exec_list, reserv_list);
5  if |MER| = 0 then
6      min_reject := number of tasks not scheduled in reserv_list;
7      if min_reject < Global_min_reject then
8          record reserv_list; Global_min_reject = min_reject;
9      return;
10 for i ← 1 to K do
11     for j ← 1 to |MER| do
12         if feasibility(window[i], MER[j])! = 0 then
13             res := create_reservation(window[i], MER[j]);
14             insert_reservation_list(res);
15             DS_Exp(exec_list, reserv_list, window - window[i],
               min_reject);
16             delete_reservation_list(res);
```

Basically, the algorithm performs a depth first search of the design space (procedure DS_Exp) for all the tasks in the window. In DS_Exp procedure, the algorithm finds all the feasible MERs and generates all combinations of MERs and tasks. The algorithm returns when all feasible combinations have been explored. If, for a particular combination, the rejection rate in minimal, it is maintained. So the algorithm can identify the solution with the minimum rejection rate.

IV. Experimental evaluation

We use 17 kernels from MiBench and MediaBench to create our task set. For each kernel, we manually generated custom instructions for Stretch platform [13] by using Stretch C language. Stretch processor incorporates Xtensa processor and the Stretch Instruction Set Extension Fabric (ISEF). The ISEF is software-configurable datapath based on programmable logic. It consists of arithmetic/logic elements (ALU) and multiplier elements (MU) interlinked in a programmable routing fabric. A set of programmer defined custom instructions can be implemented in this fabric. Once we define the custom instructions, the profiler in Stretch provides us the execution time and the hardware area (in terms of ALUs and MUs) for each kernel.

However, Stretch platform currently does not support partial reconfiguration of the ISEF. To emulate partial reconfiguration, we assume that the ISEF is organized as expression-grain reconfigurable architecture (EGRA) [14]. Four ALUs and two MUs are combined together to form a cell. A cell is the atomic unit that is reconfigurable. Thus the reconfigurable fabric is simply an 1D array of cells. We assume that a kernel requires

Kernel	Number of cells	Execution time (ms)
CRC32	10	2
GETMB	60	6
FIR	2570	38
DES	266	49
DCT	1509	280
ZZQ	253	1258
RGB2CMYK	85	1313
DJPEG	3107	1377
YCC2RGB	2082	1440
Filter	1593	1505
AUTOCOR	480	1878
DEQUANTIZE	1025	3370
ADPCM_DEC	190	4428
Quantize	2880	4443
MDCT	1265	4464
CJPEG	2662	6103
RGB2YCC	712	8336

TABLE I
AREA REQUIREMENT (IN CELLS) AND EXECUTION TIME OF KERNELS.

	Shared EDF	Shared EDF-NF	Shared heuristic	Shared 2D-SPP
running time (ms/task)	0.001	0.26	0.31	2.7

TABLE II
RUNNING TIME FOR SHARED ONLINE SCHEDULING ALGORITHMS.

consecutive cells to implement its custom instructions. Table I shows the execution time and the area requirements obtained from Stretch profiler for the 17 kernels. The area requirement is presented in terms of cells. Note that the reconfiguration time for each task is included in its execution time.

We generate task sequences by randomly selecting the kernels from Table I. The task arrival times are generated using Poisson distribution, which assumes no relationship between any two consecutive task requests. The parameter λ in Poisson distribution directly affects the workload. The larger λ is, the heavier is the workload. We carry out experiments for different workload by increasing λ and we stop when the acceptance rate drops below 0.1 for our approach. The laxity (difference between deadline and execution time plus arrival time) is generated using uniform distribution in the range [0, 10000].

We assume a 4-core platform where the cores share reconfigurable fabric. The shared reconfigurable fabric consists of 3600 cells, which is sufficient to accommodate the kernel with largest area requirement. We apply four different scheduling algorithms: *shared EDF*, *shared EDF-NF*, *shared heuristic* and *shared 2D-SPP*. The window size is set to 20 for *shared heuristic*. However, as *shared 2D-SPP* has long running time, we set the window size to 6. We confirmed experimentally that increasing the window size further does not bring any benefit. We attempted three different strategies (best-fit, worst-fit, and first-fit) for choosing the maximal empty rectangle (MER) in *shared heuristic*. However, we noticed that the different strategies achieve quite similar acceptance rate. So we employ best-fit strategy in all the experiments.

Figure 2(a) shows the experiment results for four scheduling algorithms. The X-axis represents the workload in terms of the parameter λ in Poisson distribution of arrival time of the tasks and the Y-axis is the acceptance rate of the tasks. As expected, the acceptance rate decreases as the workload increases.

We notice that the window-based algorithms (*shared heuristic* and *shared 2D-SPP*) perform on an average 22% better compared to those without windows (*shared EDF* and *shared EDF-NF*). What is surprising, though, is that *shared heuristic* is comparable to *shared 2D-SPP* even though the former is a much simpler algorithm. Finding the local optima using 2D-SPP does not help in global scheduling. *Shared EDF* is the worst among the four algorithms. While *shared EDF-NF* could bring around 8% benefit, it still has very low acceptance rate.

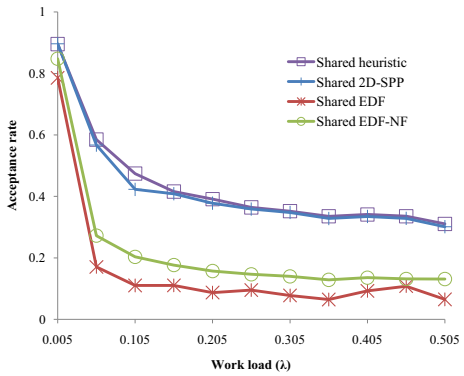


Fig. 2. Comparison of different scheduling algorithms.

The runtime of the algorithms as measured on a 300MHz Stretch processor is presented in Table II.

V. CONCLUSION

In this paper, we consider the online scheduling for multi-core shared reconfigurable fabric. Compared to previous research work, we consider both processor resources and reconfigurable resources. Our scheduling approach leads to significantly better utilization of the shared reconfigurable fabric compared to simple priority based scheduling.

ACKNOWLEDGMENTS

This work was partially supported by Singapore Ministry of Education Academic Research Fund Tier 2 MOE2009-T2-1-033.

REFERENCES

[1] Z. Chen, R. N. Pittman, and A. Forin, "Combining multicore and reconfigurable instruction set extensions," in *FPGA*, 2010.
[2] M. Watkins and D. Albonesi, "ReMAP: A reconfigurable heterogeneous multicore architecture," in *MICRO*, 2010.
[3] P. Garcia and K. Compton, "Kernel sharing on reconfigurable multiprocessor systems," in *FPT*, 2008.
[4] L. Chen and T. Mitra, "Shared Reconfigurable Fabric for Multi-core Customization," in *DAC*, 2011.
[5] M. Shafiue, L. Bauer, W. Ahmed, and H. Jorg, "Minority-Game-based Resource Allocation for Run-Time Reconfigurable Multi-core Processors," in *DATE*, 2011.
[6] T. P. Baker, "Multiprocessor EDF and Deadline Monotonic Schedulability Analysis," in *RTSS*, 2003.
[7] P. Mahr, S. Christgau, C. Haubelt, and C. Bobda, "Integrated Temporal Planning, Module Selection and Placement of Tasks for Dynamic Network-on-Chip," in *IPDPS*, 2011.
[8] K. Danne and M. Platzner, "An EDF schedulability test for periodic tasks on reconfigurable hardware devices," in *LCTES*, 2006.
[9] ——, "A heuristic approach to schedule periodic real-time tasks on reconfigurable hardware," in *FPL*, 2005.
[10] Y. Lu, T. Marconi, G. Gaydadjiev, and K. Bertels, "An efficient algorithm for free resources management on the FPGA," in *DATE*, 2008.
[11] S. Martello, D. Pisinger, and D. Vigo, "The Three-Dimensional Bin Packing Problem," *Operations Research*, vol. 48, no. 2, 2000.
[12] S. Martello, M. Monaci, and D. Vigo, "An Exact Approach to the Strip-Packing Problem," *Informs Journal on Computing*, vol. 15, no. 3, 2003.
[13] R. E. Gonzalez, "A software-configurable processor architecture," *IEEE Micro*, vol. 26, no. 5, 2006.
[14] G. Ansaloni, P. Bonzini, and L. Pozzi, "Design and architectural exploration of expression-grained reconfigurable arrays," in *SASP*, 2008.