Hardware Task Scheduling and Placement in Operating Systems for Dynamically Reconfigurable SoC

Yuan-Hsiu Chen and Pao-Ann Hsiung

National Chung Cheng University, Chiayi, Taiwan–621, ROC
pahsiung@cs.ccu.edu.tw

Abstract. Existing operating systems can manage the execution of software tasks efficiently, however the manipulation of hardware tasks is very limited. In the research on the design and implementation of an embedded operating system that manages both software and hardware tasks in the same framework, two major issues are the dynamic scheduling and the dynamic placement of hardware tasks into a reconfigurable logic space in an SoC. The distinguishing criteria for good dynamic scheduling and placement methods include the total schedule length and the amount of fragmentation incurred while tasks are dynamically placed and replaced. Existing methods either do not take fragmentation into consideration or postpone the consideration of fragmentation to a later stage of space allocation. In our method, we try to reduce fragmentation during placement itself. The advantage of such an approach is that not only the reconfigurable space is utilized more efficiently, but the total schedule length is also reduced, that is, hardware tasks complete faster. Experimental results on large random tasks sets have shown that the proposed improvement is as much as 23.3% in total fragmentation and 2.0% in total schedule time.

Keywords: Operating System for Reconfigurable SoC, Hardware Scheduling, Placement, Dynamic Partial Reconfiguration.

1 Introduction

The advent of reconfigurable technologies in hardware design is making a significant impact on the design and the architectures of embedded systems and Systems-on-Chip (SoC). The hardware technologies for supporting reconfiguration are rapidly maturing, however the related software tools and design environments are relatively immature. For instance, we already have dynamic partial reconfiguration hardware such as Xilinx Virtex-II Pro FPGA; however the supporting software tools and runtime environments such as the support from embedded operating systems is still very much limited. In this work, we try to bridge this gap further by proposing scheduling and placement methods for hardware tasks in an SoC with dynamic partially reconfigurable logic.

In the design and implementation of an operating system for dynamically reconfigurable SoC, as shown in Fig. 1, hardware tasks can be dynamically scheduled and placed. For scheduling and placing hardware tasks, besides the desired efficiency, other important issues include the fragmentation of reconfigurable logic space and the total schedule time for a set of tasks. Existing methods either do not consider the issue of

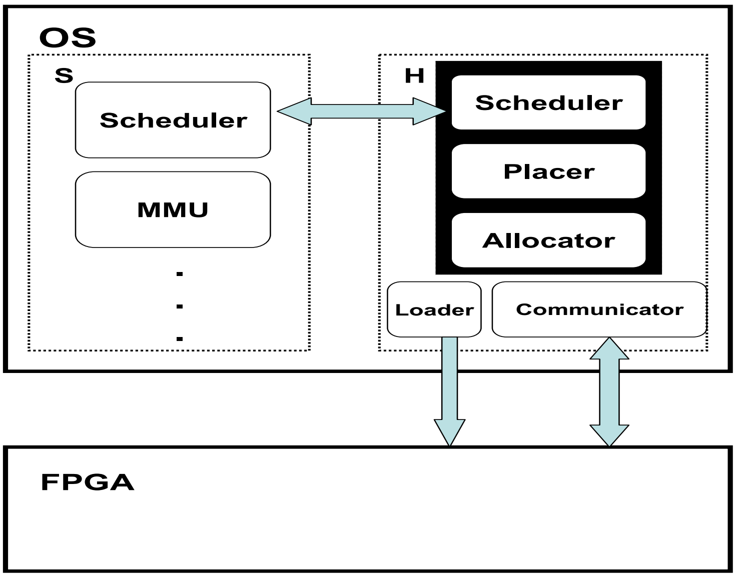

Fig. 1. Operating system for dynamically reconfigurable systems

fragmentation or postpone it to some later stage of space allocation. In this work, we will try to consider the fragmentation of reconfigurable space during scheduling and placement. The goal here is to reduce the amount of unused space that is wasted. As a result not only is the fragmentation reduced but the overall response and total schedule length of the tasks are also reduced.

FPGA is the most widely used reconfigurable logic nowadays. We will model the FPGA space and also the hardware tasks so that they can be used in the proposed scheduling and placement methods. Through experiments, we will also show how much reduction we obtained in fragmentation and in total time through a very simple task classification scheme that nearly does not incur any overhead during dynamic scheduling and placement. The techniques developed in this work will thus be very useful in the design of an operating system for dynamically reconfigurable systems.

The rest of this article is organized as follows. Section 2 presents some previous work in scheduling and placement of hardware tasks for dynamically reconfigurable systems. Section 3 describes the models for FPGA and for hardware tasks, which are used in our proposed method. Section 4 presents the two different scheduling algorithms that we used for scheduling the hardware tasks. Section 5 describes the proposed placement method for hardware tasks. Experimental results are given in Section 6. The article is concluded with future research directions in Section 7.

2 Previous Work

Scheduling hardware tasks is almost similar to scheduling software tasks. Scheduling methods can be classified into static and dynamic depending on when the scheduling is done. Static scheduling can spend more time to find an optimal schedule, but it is fixed before run-time and thus cannot be changed once a system is running, making it quite inflexible. In contrast, dynamic scheduling must make scheduling decisions quickly so that the FPGA is not kept idle for too long without executing any hardware tasks; however the schedules might not be optimal. For static scheduling, variants of list scheduling have been proposed and used in several works, which differ mainly in their assignment of task priorities. Random priority list scheduling was used in the reconfigurable environment scheduling model [3] and dynamic priority list scheduling was proposed in [4].

For dynamic scheduling, the following methods have been proposed or used. Priority-based scheduling was used in the task and context schedulers [5]. Non-preemptive methods such as First-Come First-Serve (FCFS) and Shortest-Job First (SJF) scheduling were used in the online scheduler of [7]. Preemptive methods such as Shortest-Remaining Processing Time (SRPT) and Earliest Deadline First (EDF) were used also used in [7]. Steiger et al proposed a horizon and a stuffing method for dynamic scheduling and placement of hardware tasks [6].

The FPGA space has been modeled differently in the previous works. For example, in [5], the FPGA space was divided into slots that could accommodate only specific types of dynamic reconfigurable logics. In [7], the FPGA space was divided into slots of several fixed sizes and the placer selected the most suitably sized block for a task at hand, thus resulting in a smaller fragmentation. In [6], the FPGA space is considered as either a 1-dimensional or 2-dimensional area that can configured during scheduling and placement, for which two methods were proposed, namely horizon and stuffing.

Because our goal is to perform scheduling and placement of hardware tasks in an operating system, we focus on dynamic scheduling and placement methods. After a thorough survey we found that the 1-dimensional stuffing method [6] approximately meets our needs; however there are some issues such as significant fragmentation and schedule length so we decided to improve on this method such that we can decrease the resulting fragmentation and reduce the total execution time when hardware tasks are dynamically scheduled and placed in a reconfigurable SoC.

3 FPGA and Task Modeling

The FPGA configurable space is considered to be a set of columns, where a column spans the height of a chip and the set of columns spread across the width of the chip. The basic unit of configuration in this model is a column, which itself is a set of CLBs. This model fits the current technology of partial dynamically reconfigurable Xilinx Virtex II Pro FPGA chips. In this model, the reconfiguration is a 1-dimensional problem.

The hardware to be configured on an FPGA is modeled as a set of hardware tasks, where each task t has a set of attributes including arrival time $A(t)$, execution time $E(t)$, deadline $D(t)$, and area $C(t)$ (in terms of the number of FPGA columns required). These numbers can be obtained by synthesizing a hardware function using a synthesis tool such as Synplify or XST.

Given a set of hardware tasks and an FPGA space, our target problem is to schedule and place the tasks in the FPGA space such that the task attributes are all satisfied and the time length of the schedule and the amount of FPGA space wasted (unconfigured) are minimized. As a solution to this problem, we proposed a *classified stuffing* method which improves on the original stuffing technique [6]. The scheduling and placement techniques in classified stuffing will be discussed in Sections 4 and 5, respectively.

4 Hardware Task Scheduling

The hardware tasks to be configured on FPGA may either have deadlines or not, according to which, we use different scheduling algorithms. For tasks without deadlines we use the *Shortest Remaining Processing Time* (SRPT) algorithm [1]. For tasks with deadlines we use the *Least Laxity First* (LLF) algorithm [2].

4.1 Hardware Scheduling Without Deadline

SRPT [1] is an optimal algorithm for minimizing mean response time and it has also been analyzed that the common misconception of unfairness in SRPT to large tasks is unfounded. For hardware tasks without deadline, we thus employ SRPT scheduling algorithm. The task which needs the least execution time will be assigned the highest priority. In the scheduler there is a queue to place hardware tasks that are ready for execution. The scheduler will insert the tasks into the ready queue according to their priorities.

4.2 Hardware Scheduling with Deadline

For scheduling real-time hardware tasks with deadlines, we can use either EDF or LLF. However, because hardware tasks are non-preemptive and truly parallel, we decided to use LLF, which degrades a little more gracefully under overload than EDF, extends relatively well to scheduling multiple processors, which are similar to parallel hardware tasks, and approaches the schedulability of EDF for non-preemptive tasks.

In LLF, the priority $P(t)$ of a task t is assigned as given in Equation (1), where now is the current time. Intuitively, the priority is the *slack* time for a task.

$$P(t) = D(t) - E(t) - now \tag{1}$$

5 Hardware Task Placement

In the 1D FPGA model, the smallest placement unit is one column of CLBs. Hence, we just need to know how wide a task is, that is, the number of columns required. If we only consider the task width for placing, it is a 1-dimensional problem. However, to more effectively utilize FPGA space, we also consider the task execution time, which makes the hardware task placement a 2-dimensional problem.

The main difference between our classified stuffing and the original stuffing method [6] is the classification of hardware tasks. In our method, we classify all hardware tasks into two types and the placement location of these two types of tasks are different, while in the stuffing method there is no such distinction. An example is shown in Fig. 2.

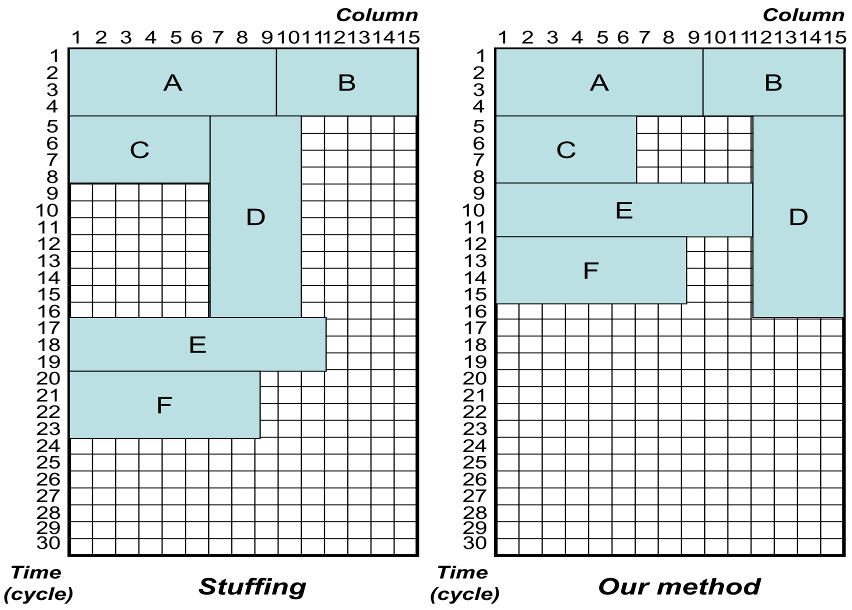

Fig. 2. The difference between our method and stuffing

Suppose tasks A, B, and C are already placed as shown in Fig. 2 and the placements are the same in our method and in stuffing. Next, task D is to be placed, however the location it is placed will be different in our method and in stuffing. In stuffing, it is placed adjacent to task C in the center columns of the FPGA space. In contrast, in our method task D will be placed from the rightmost columns. As a consequence, in our method tasks E and F can be placed earlier than that in stuffing, resulting in a shorter schedule. Finally, the fragmentation in our classified stuffing will also be lesser than that in stuffing because the space is used more compactly.

5.1 Task Classification

For a task t, we call the ratio $C(t)/E(t)$ the *Space Utilization Rate* (SUR) of t. Given a set of hardware tasks, they are classified into two types, namely *high SUR* tasks with *SUR*$(t) > 1$ and *low SUR* tasks with *SUR*$(t) \leq 1$. The classification of a task determines where it will be placed in the FPGA as follows. High SUR tasks are placed starting from the leftmost available columns of the FPGA space, while low SUR tasks are placed starting from the rightmost available columns. This choice of segregating the tasks and their placements is to reduce the conflicts between the high and low SUR tasks. A conflict often results in unused FPGA space, which in turn increases the total execution time. Because the classification scheme does not take the number of tasks of each type into account, one may wonder if this classification will not gain much when there is an imbalance between the number of high and low SUR tasks. However, after thorough experiments, we found the best results (least fragmentation, shortest execution time) are

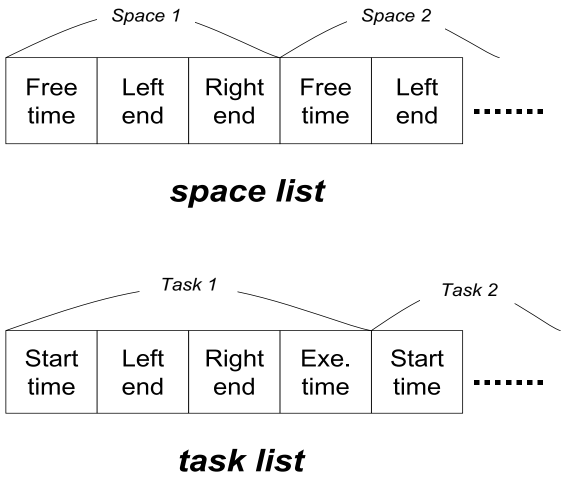

Fig. 3. Data structures used for recording placement and scheduling information

always obtained when we divide the task set considering only their SUR values. The results are described in Section 6 and Table 2.

5.2 Recording Placement Information

During placement, two lists are maintained, namely a *space list* for recording the free spaces and a *task list* for recording the task locations in FPGA after placement. For every free space in the space list we record three data including release time, the leftmost and the rightmost columns of this free space in FPGA. When the placer places a task into FPGA, the task list will store four information about the task, namely (1) the starting execution time of the task, (2) the leftmost column occupied by the task, (3) the rightmost column occupied by the task, and (4) the execution time of the task. The space list supplies information for the placer to find a fit free space for tasks, and the task list allows the placer to detect whether there is any conflict among the tasks. Figure 3 illustrates the two data structures. How to find a fit space and how to detect placement conflicts will be discussed in Section 5.3.

5.3 Placement Method

In the proposed classified stuffing method, placement of hardware tasks in FPGA space is performed as described in the following steps.

1. *Select a task:* The placer chooses a highest priority task from the ready queue according to our scheduling policy as described in Section 4. The task is classified into either high or low SUR as described in Section 5.1.
2. *Find a fit space:* Based on the width of a task, the placer will find a best fit free space from the space list. The best fit space is the earliest released of all free spaces which have enough columns of CLBs for the task. If there are more than one space

released at the same time, the leftmost space will be selected for placing a high SUR task, while the rightmost space will be selected for a low SUR task.

3. *Decide placement location in a free space:* At the second step, the placer found a best fit free space for a task, but this space maybe wider than that required by the task. In this case, according to the task type we determine which columns of CLBs of the best fit space will be used. A *high SUR* task will be placed from the leftmost column and a *low SUR* task will be placed from the rightmost column.
4. *Checking conflict:* At this step we already know where a task is to be placed, but we need to check if there is any conflict between the newly placed task and an existing already placed task. If a conflict exists, we must choose another fit space for the new task by going back to Step (2).
5. *Modify space list:* After a task is placed into a best fit space, not only the information of this fit space need to be modified, but also some other free spaces in the space list may be affected and thus need to be updated.

6 Experiment Results

To evaluate the advantages of the classified stuffing method compared with the original stuffing technique [6], we experimented with 150 sets of randomly generated hardware tasks, one-third of the sets had 50 tasks in each set, one-third had 200 tasks in each set, and one-third had 500 tasks in each set. The input task sets were also characterized by different *set ratios*, where the set ratio of a task set is defined as $|S_{>1}| : |S_{\leq 1}|$, where $S_{>1} = \{t \mid SUR(t) > 1\}$, $S_{\leq 1} = \{t \mid SUR(t) \leq 1\}$ so as to find out the kind of task sets for which our classified stuffing works better than stuffing. We experimented with three sets of tasks with set ratios 4:1, 1:1, and 1:4. The experiments were conducted on a P4 1.6 GHz PC with 512 MB RAM.

In the two-dimensional area of FPGA space versus execution time as illustrated in Fig. 2 each unit of placement is a *column-cycle*, where the space unit is a column and the time unit is a cycle. Table 1 shows the results of applying our method and the stuffing method to the same set of tasks, where the total column-cycles is the amount of column-cycles used by all tasks, the total fragmentation is the number of column-cycle not utilized, the total time is the total number of cycles for executing all tasks, the average fragmentation is the quotient of total fragmentation by total time, and the number of rejected tasks is the number of tasks whose deadlines could not be satisfied. For each task set, the data in Table 1 are the overall averages of the results obtained from experimenting with 50 different sets of randomly generated tasks.

From Table 1, we can observe that for both the methods and for all the sets of tasks the set ratio of 1:4 results in a shorter schedule and more compact placement as evidenced by the smaller fragmentation, which means that it is desirable to have hardware tasks synthesized into tasks with low SUR. The intuition here is that a greater number of high SUR tasks would require wider FPGA spaces resulting in larger fragmentation and thus longer schedules.

From Table 1, we can observe that for all the cases our method performs better than stuffing in terms of both lesser amount of fragmentation and lesser number of cycles. The total fragmentation is reduced by 5.5% to 23.3% and the total execution time is

Table 1. Experimental Results and Comparison

Task Set (S)			**Original Stuffing**				**Classified Stuffing (reduction)**				**S&P Time** (ms)
$\|S\|$	SR	TC	TF	TT	AF	TR	TF	TT	AF	TR	
50	4:1	1,711	505	105	4.8	3	477 (-5.5%)	104 (-1%)	4.6 (-4.2%)	3	3
	1:1	1,707	405	100	4	5	380 (-6.2%)	99 (-1%)	3.8 (-5%)	4	2
	1:4	1,550	418	93	4.5	2	392 (-6.2%)	92 (-1%)	4.3 (-4.4%)	2	2
200	4:1	6,780	1,237	381	3.25	3	1,119 (-9.5%)	376 (-1.3%)	2.97 (-8.6%)	3	5
	1:1	6,478	887	350	2.53	1	760 (-14.3%)	344 (-1.7%)	2.21 (-12.6%)	1	11
	1:4	6,133	823	330	2.49	0	739 (-10.2%)	326 (-1.2%)	2.27 (-8.8%)	0	12
500	4:1	17,145	2,246	922	2.43	5	2,007 (-10.6%)	911 (-1.2%)	2.20 (-9.5%)	5	47
	1:1	16,080	1,619	842	1.92	9	1,242 (-23.3%)	825 (-2.0%)	1.50 (-21.9%)	9	77
	1:4	15,077	1,341	782	1.71	5	1,185 (-11.6%)	775 (-0.9%)	1.53 (-10.5%)	5	98

SR: set ratio, TC: total column-cycles, TF: total fragmentation, TT: total time, AF: average fragmentation (TF/TT), TR: #tasks rejected, S&P: Scheduling and Placement

reduced by 0.9% to 2.0%. The reduction becomes more prominent in the case with set ratio 1:1 because there is an increased interference between the placements of the low SUR tasks and the high SUR tasks resulting in a larger fragmentation and longer schedules in the original stuffing method. These kind of interferences are diminished in classified stuffing. Comparing the task sets with different cardinalities, namely 50, 200, and 500, one can observe in Fig. 4 that the larger the task set is the better is the performance of our method. This shows that our method scales better than stuffing to more complex reconfigurable hardware systems.

As far as the performance of the scheduler and placer is concerned, the time taken by the classified stuffing method and the original stuffing method are almost the same. Thus, we have only one column for the S&P Time in milliseconds in Table 1. This is because the added classification technique does not also expend any significant amount of time, while it does improve the FPGA space utilization and total task execution time.

To support our choice of task classification using SUR value (see Section 5.1) and not considering the number of tasks of each type, we performed several experiments by using different classification ratios (CR) for a task set. The results are tabulated in Table 2, where we can observe that the best results are usually obtained when the classification

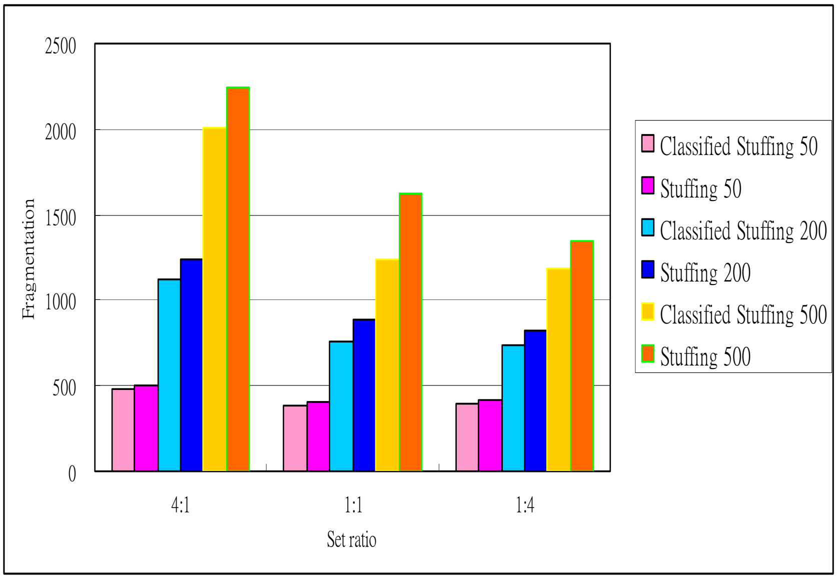

Fig. 4. Classified Stuffing vs. Original Stuffing

Table 2. Finding the Best Classification Ratio

Set Ratio	Scheduling Results	Classification Ratio		
		4:1	1:1	1:4
4:1	Total Fragmentation	**638**	667	667
	Total Time	**191**	192	192
1:1	Total Fragmentation	456	**435**	471
	Total Time	173	**171**	173
1:4	Total Fragmentation	458	438	**432**
	Ttotal Time	166	165	**165**

ratio is the same as the set ratio, that is, the threshold for classification can depend only on the SUR values.

7 Conclusions

A classified stuffing technique was proposed in this work showing significant benefits in both a shorter schedule and a more compact placement, but with little overhead. This was demonstrated through a large number of task sets. The current method focuses only on hardware scheduling. In the future, we will investigate hardware-software coscheduling.

References

1. N. Bansal and M. Harchol-Balter. Analysis of SRPT scheduling: investigating unfairness. In *Proc. of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 279–290. ACM Press, 2001.
2. J. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4:209–219, 1989.
3. S.M. Loo and B. E. Wells. Task scheduling in a finite resource reconfigurable hardware/software co-design environment. *INFORMS Journal on Computing*, 2005. to appear.
4. B. Mei, P. Schaumont, and S. Vernalde. A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems. In *Proc. of the 11th ProRISC Workshop on Circuits, Systems, and Signal Processing*, 2000.
5. J. Noguera and R. M. Badia. System-level power-performance trade-offs in task scheduling for dynamically reconfigurable architectures. In *Proc. of the 2003 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 73–83. ACM Press, October 2003.
6. C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1392–1407, November 2004.
7. H. Walder and M. Platzner. Online scheduling for block-partitioned reconfigurable devices. In *Proc. of the Design Automation and Test, Europe (DATE)*, volume 1, pages 290–295, March 2003.