Optimal Placement-aware Trace-based Scheduling of Hardware Reconfigurations for FPGA Accelerators

Joon Edward Sim, Weng-Fai Wong
School of Computing
National University of Singapore
{esim, wongwf}@comp.nus.edu.sg

Jürgen Teich
Department of Computer Science
University of Erlangen-Nuremberg, Germany
teich@cs.fau.de

Abstract

Modern use of FPGAs as hardware accelerators involves the partial reconfiguration of hardware resources as the application executes. In this paper, we present a polynomial time algorithm for scheduling reconfiguration tasks given a trace of actors (invocations of hardware kernels) that is both provably optimal and placement-aware. In addition, we will propose a dependence analysis to determine whether for each actor instance, a reconfiguration task is needed prior to its execution in hardware. A case study using the H.264 encoder is presented to compare our algorithm against the state-of-the-art heuristics.

1 Introduction

One of the key challenges in achieving real speedups using in FPGA-based reconfigurable architectures is that hardware reconfiguration of today's massive FPGAs can be very costly. The configuration cost need to be be amortized, so that all benefits of hardware acceleration may not be lost as the application has to wait for reconfiguration to complete. *Configuration prefetching* [7] seeks to address this problem by overlapping (partial) reconfiguration with the execution of the application in FPGA. However, a prefetch miss is costly because of the additional reconfigurations that may be needed to recover from the miss. Therefore, the scheduling of reconfiguration is crucial.

In this context, this paper solves the following problem. Given a sequence (trace) of *actors* (an invocation of a hardware module):

- Determine whether for a given actor in the trace, it is necessary to schedule a reconfiguration task before it.
- Compute the earliest possible time a required reconfiguration task may be scheduled. For the current technology, at most one reconfiguration task is typically allowed at any time.

In essence, we will present a *polynomial-time* (in terms of the length of the trace and the number of distinct hardware modules) algorithm that schedules all the required reconfigurations such that the overall execution time (latency) of the given actor trace is provably minimized. To the best of our knowledge, this is the first time an algorithm of this nature has been proposed.

2 Preliminaries

2.1 Architecture model

We consider an architecture with one micro-processor that receives a trace of actors. For each actor, we assume a corresponding hardware accelerator module that may be loaded into the FPGA for subsequent execution by means of partial hardware reconfiguration.

2.2 Scheduling model

Example 2.1 *Figure 1 shows an example of a given application consisting of a sequence of five actors (corresponding to four tasks) with data dependencies, and a given conflict relation concerning the shared use of FPGA resources. For example, when task B conflicts with C, this would mean that they share some common hardware resources on the FPGA which may be either I/O pins, memory resources (such as block rams), or slices.*

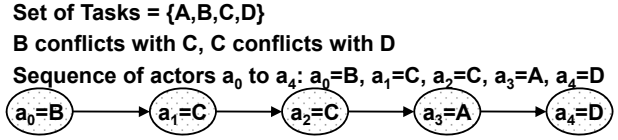

Figure 1. Example of actor trace

Assume for now that this conflict relation is given statically, i.e., no module relocation is allowed. Thus we know the conflicts between every pair of actors at compile time. More formally, we define an actor trace and the corresponding conflicts as follows:

1. *Trace of actors*: $S_a = (a_0, a_1, a_2, \ldots, a_n)$ with $a_i \in T, i \neq 0$. T is a set of tasks, and $\|T\| = N$, where N is number of tasks.

2. *Resource conflicts*: The relation $C = \{(T_i, T_j) | T_i \nsim T_j\}$ denotes that the placement of T_i conflicts with placement of T_j.
3. Any actor $a_i \in S_a$ can only be scheduled for execution on the FPGA if all its preceding tasks have completed execution. Furthermore, if the corresponding module is not in the FPGA, it needs to be loaded, i.e., the corresponding resources reconfigured, prior to execution.

3 Precedence Analysis

Before defining the scheduling problem, we need to distinguish three different types of dependencies: The first one, data dependencies, is obvious. The second is the conflict relation introduced above that is due to the sharing of FPGA resources among the hardware modules. Finally, the third kind of dependencies arise because some actors cannot begin execution until its corresponding configuration task is completed. In order to compute this, we first need to discuss the problem of reconfiguration task generation.

3.1 Generation of reconfiguration tasks

Definition 3.1 (True dependence) *Given a sequence of actors S_a. a_i is called truly dependent on a_j, written $a_j \prec a_i$ iff*

$$\nexists k, j < k < i : (a_k \nsim a_i) \wedge (\forall k', k < k' < i : a_k \neq a_i)$$

True dependence is based on the intuition that, for an actor a_i of task $t \in T$, not every occurrence of conflicting predecessors in the trace matters. It is the conflicting predecessor a_k that is closest to a_i that will have an impact on the reconfiguration decision for a_i. Furthermore, a_i must be the first actor of task type t in the trace subsequent to a_k.

Example 3.1 *In Figure 1, a_1 is truly dependent on a_0 but a_2 has no true dependence because it executes after another actor, a_1, of the same task.*

Now, each first appearance of a task in a trace will also necessitate exactly one reconfiguration task. Hence, the set of required reconfiguration tasks $S_r = (r_0, \ldots, r_l)$ may be found by inspecting the given trace once.[1]

Theorem 3.1 (Reconfiguration task instantiation) *For an actor a_i in a given trace S_a, there needs to be a corresponding reconfiguration actor (task) r_i if, and only if, $\exists a_j \in S_a : a_j \prec a_i$. In other words, if there exists a predecessor a_j in S_a on which a_i is truly dependent.*

[1]Note that the subscripts of reconfiguration tasks in S_r are in sequence but not necessary running as they correspond to the subscript of the associated actor, and not all actors need reconfiguration.

For each reconfiguration task r_i, two additional dependencies must be created. First, each r_i must complete before the corresponding actor a_i starts executing. Second, for a_j such that $a_j \prec a_i$, reconfiguration task r_i for a_i cannot start earlier than the completion of a_j on which a_i is truly dependent on because r_i affects the execution of a_j. The two dependencies are shown by adding an outgoing edge from r_i to a_i and one incoming edge from a_j to r_i.

Example 3.2 *Figure 2 shows both the set of reconfiguration tasks generated for the running example as introduced in Example 2.1 and the additional scheduling dependencies.*

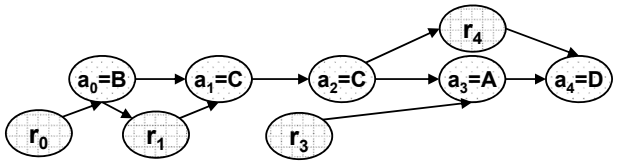

Figure 2. Dependence Relations

In summary, we have to consider the following three types of dependencies for scheduling after having all the required reconfiguration tasks generated:

- *Sequential precedence*: $P_s = \{(a_i, a_j) | (0 \leq i \leq n-1) \wedge (j = i+1)\}$;
- *Conflict (resource) precedence*: $P_c = \{(a_j, r_i) | a_j \prec a_i\}$; and
- *Reconfiguration precedence*: $P_r = \{(r_i, a_i) | (\exists r_i \in S_r)\}$.

The complete dependence relation is thus $P = P_s \cup P_r \cup P_c$.

3.2 Minimizing the schedule length

Given the above, we are now in a position to state the scheduling problem formally. The following notation will be used throughout the paper:

- $l(a_i)$: latency of actor a_i
- $s(a_i)$: the start time of actor a_i
- $f(a_i)$: the end time of actor a_i
- $l(r_i)$: latency of reconfiguration task r_i
- $s(r_i)$: the start time of reconfiguration task r_i
- $f(r_i)$: the finishing time of reconfiguration task r_i

Definition 3.2 (Feasible schedule) *A feasible schedule is an assignment of end times $f(a_i)$ and $f(r_i)$, respectively, to every actor $a_i \in S_a$ and reconfiguration task $r_i \in S_r$ such that all the above mentioned precedence constraints are satisfied, i.e., $\forall j$ such that $(X_i, X_j) \in P$ then $s(X_j) \geq f(X_i)$.*

The aim of a scheduling algorithm for this problem is to find a feasible schedule where $f(a_n)$ is minimized for a trace of actors $S_a = (a_0, a_1, a_2, \ldots, a_n)$.

4 Algorithm MLS

Algorithm 1: MLS Algorithm

```
Input: Trace of actors: S_a;
       Set of Conflicting Hardware Modules: C;
       Set of tasks: T;
Result: Optimal Schedule Length
ForAll (f_t, prev_t: t ∈ T) f_t ← true; prev_t ← −1;
for a_i ← a_0 to a_n : a_i ∈ S_a do
    if f_{a_i} is true then
        CreateReconfigurationTask (r_i);
        (r_i).TimeRemaining ← l(r_i);
        if prev_{a_i} ≠ −1 then AddEdge (a_{prev_{a_i}}, r_i);
        AddEdge (r_i, a_i);
        ForAll (t ∈ T) if (t, a_i) ∈ C then f_t ← true; prev_t ← i;
        if r_i has no preceding tasks then Insert (H, r_i);
if TaskReady (a_0) then current_A ← a_0; else current_A ← empty;
length ← 0;
while current_A ≠ a_n do
    if current_A is empty then
        r ← ExtractMax (H);
        length ← length + (r).TimeRemaining;
        current_A ← NextTask (r);
    else
        length ← length + l(current_A);
        Time ← l(current_A);
        while H not empty ∧ Time ≠ 0 do
            r ← ExtractMax (H);
            if l(r) < T then Time ← Time − l(r);
            else
                r.TimeRemaining ← r.TimeRemaining − T;
                Time ← 0;
                Insert (H, r);
        ForAll (r ∈ DependsOn (current_A)) Insert (H, r);
        if TaskReady (NextTask (current_A)) then
            current_A ← NextTask (current_A);
        else current_A ← empty;
length ← length + l(a_n);
return length;
```

We shall now present the main result of this paper, namely a polynomial time, latency-optimal scheduling algorithm for actors and reconfiguration tasks that we call *Modified List Scheduling* (MLS). The algorithm assumes that reconfiguration tasks can be pre-empted. This is based on the way frame-based reconfigurable devices operate. Configuration for frame-based devices such as Xilinx FPGAs is achieved by writing a set of frames into the SRAM configuration memory of the device. It does not matter whether the reconfiguration process is carried out in 1, 2, or more phases as long as the affected area is not again rewritten by other module configurations in between. Also, the algorithm prioritizes reconfiguration tasks by the order of appearance of their corresponding actors in the actor trace.

The MLS algorithm is shown in Algorithm 1. It consists mainly of 2 passes through the actor trace. In the first pass, the algorithm finds true dependences between the actors and generate the corresponding reconfiguration tasks S_r. To do this, we maintain a flag f_t for each task $t \in T$ and an index $prev_t$. We traverse the trace from a_0 to a_n. Assume that a_i is the current actor. If flag f_{a_i} is true, a corresponding reconfiguration task r_i will be created, and if $prev_t \neq -1$, r_i is to be preceded by actor a_{prev_t} (i.e. truly dependent on a_{prev_t}). $prev_t = 1$ when the reconfiguration task created is needed for the first occurrence of a_i. Furthermore, we record all ready reconfiguration tasks in a heap data structure H, ordered by the relative appearance order of the associated actor in the actor trace. In order to perform preemptive scheduling of reconfiguration tasks, we maintain a **TimeRemaining** attribute for each of the tasks and this is initialized to the full reconfiguration latency required.

The second pass through the trace computes the actual scheduling time using preemptive scheduling of reconfiguration tasks. **current_A** is the current ready actor. If there are no ready actors, we schedule a ready reconfiguration task r whose associated actor has the earliest appearance order in the actor trace. Otherwise, we schedule actor **current_A**. In the time $l($**current_A**$)$, we schedule as many reconfiguration tasks sequentially as possible to configure the FPGA in parallel with the execution of **current_A**. However, the space given by the scheduled actor may not be enough for the **TimeRemaining** of r to fill up. Such r's are inserted back into H with updated **TimeRemaining**. The algorithm terminates when the last actor a_n is scheduled.

5 Case Study

5.1 H264-encoder case study

We use a H.264 [10] encoder application as a case study of the effectiveness of our algorithm. Based on profiling, we identified 15 loops that take up most of the computation time in the application. The hardware implementation of these loops were synthesized using Xilinx's ISE.

Table 1 shows the characteristics of the application using two actor traces obtained with the 15 loops. It shows the length of the actor traces and the number of unique patterns occurring within the trace. A *pattern* is a maximal acyclic sequence of actors that occurs repeatedly in the trace. Two patterns are considered different if they differ in at least one actor. In the shorter trace, we encode one frame while in the longer trace we encode two consecutive frames. The frames are 704 by 576 pixels in size. All the hardware modules are assumed to be running at a frequency of 50 MHz.

Trace	Num. of Frames Encoded	Num. of Actors	Num. Of Unique Patterns
Short	1	35,622,092	52
Long	2	185,232,537	100

Table 1. Characteristics of the two traces

5.2 Experiment Setup

To demonstrate the effectiveness of our approach, we compared it against three algorithms: two different online Least Mean Square Predictor, and a simple scheduler.

Simple Scheduler Instead of prefetching, the Simple Scheduler maintains a record of the current FPGA configuration and only schedules a reconfiguration on demand if the actor to be executed is not yet in the FPGA. It is reasonable to expect that any prefetching approach should do no worse than the Simple Scheduler. We therefore used the schedule length computed by the Simple Scheduler as the baseline for our comparisons.

Least Mean Square Online Predictor A (LMSA-a) This is an online predictor that is similar to that described in [7, 2]. The Least Mean Square Filter is used as the predictor function. However, because the target FPGA architecture considered in our paper is different (their architecture [3] supports relocation and defragmentation), our approach does not use the priority function that is based on the configuration sizes and the different eviction policies. Rather, the hardware module evicted are those in conflict with the module currently being prefetched.

Least Mean Square Online Predictor B (LMSA-b) This is a modification of LMSA-a. Instead of predicting the next hardware task, the algorithm predicts and attempts to prefetch the next task that conflicts in placement with the currently scheduled task.

5.3 Experimental Results

In order to show the effect of increasing configuration overhead on the schedule length, we ran experiments by varying the reconfiguration speed from between 1μsec to 20μsec per CLB column.

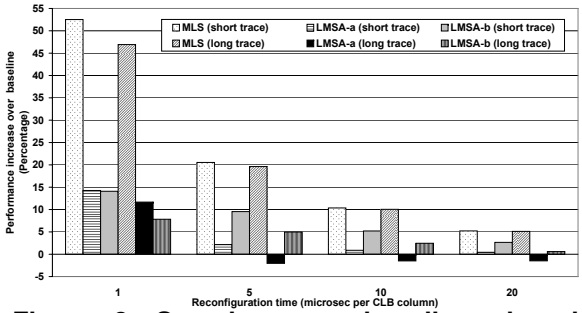

Figure 3. Speedup over baseline plotted against increasing reconfiguration time.

Figure 3 shows the performance increase of the different approaches over the schedule produced by the Simple Scheduler. The threshold of the minimum average execution cycles between two conflicting hardware module is set to 1000 cycles for this experiment. We observe that as reconfiguration speed decreases, the performance gain achieved by all the approaches decreases. With a high reconfiguration overhead, execution just has to wait till reconfiguration completes. The single reconfiguration port also becomes a bottle-neck. Over the range of reconfiguration overheads we considered, the schedule produced by MLS outperforms the others in every case. At best, it can be 30 percent better than those produced by the other schemes.

6 Related works

Configuration prefetching [5, 6, 7, 2, 4] is one of the techniques proposed to reduce the reconfiguration overhead In [7], the author described a prefetching technique for partially reconfigurable FPGAs, exploiting the overlap between hardware execution and reconfiguration. In particular, a Markov predictor was introduced for deciding on the next reconfiguration operation. An extension of the work was presented in [2]. Morphosys[8] presented a heuristic context scheduling for its coarse-grained reconfigurable architecture.

In most FPGAs and partially reconfigurable FPGA-based platforms such as the Erlangen Slot Machine (ESM) [1, 9], the reconfiguration interface may be considered as just another resource. Hence, for applications that have static reconfiguration needs, resource-constrained scheduling techniques may be used to schedule FPGA resources and reconfiguration interface simultaneously [4]. In this paper, we consider traces of *actors* which are requests for hardware activations of complex tasks. Therefore, our work goes beyond earlier ones done mainly on simple unconditional data-flow graphs.

7 Conclusions

In this paper, we presented an algorithm for the scheduling of reconfiguration tasks for FPGA-based hardware acceleration at the electronic system level. A realistic case study using the H.264 encoder has been provided to show the benefits and sensitivity of the results.

References

[1] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich. Increasing the flexibility in fpga-based reconfigurable platforms: The erlangen slot machine. In *IEEE 2005 Conference on Field-Programmable Technology (FPT)*, pages 37–42, Singapore, dec 2005.

[2] Y. Chen and S. Y. Chen. Cost-driven hybrid configuration prefetching for partial reconfigurable coprocessor. In *IPDPS*, pages 1–8, 2007.

[3] K. Compton, Z. Li, J. Cooley, S. Knol, and S. Hauck. Configuration relocation and defragmentation for run-time reconfigurable computing. *IEEE Trans. Very Large Scale Integr. Syst.*, 10(3):209–220, 2002.

[4] S. P. Fekete, J. C. van der Veen, J. Angermeier, C. Göhringer, M. Majer, and J. Teich. Scheduling and communication-aware mapping of HW/SW modules for dynamically and partially econfigurable SoC architectures. In *ARCS '07 - 20th International Conference on Architecture of Computing Systems 2007*, pages 151–160. VDE-Verlag, Berlin, 2007.

[5] S. Hauck. Configuration prefetch for single context reconfigurable coprocessors. In *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 65–74, New York, NY, USA, 1998. ACM.

[6] Z. Li, K. Compton, and S. Hauck. Configuration caching management techniques for reconfigurable computing. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 22, Washington, DC, USA, 2000. IEEE Computer Society.

[7] Z. Li and S. Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 187–195, New York, NY, USA, 2002. ACM.

[8] R. Maestre, F. J. Kurdahi, M. Fernández, R. Hermida, N. Bagherzadeh, and H. Singh. A framework for reconfigurable computing: task scheduling and context management. volume 9, pages 858–873, Piscataway, NJ, USA, 2001. IEEE Educational Activities Department.

[9] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda. The Erlangen Slot Machine: A dynamically reconfigurable FPGA-based computer. *Journal of VLSI Signal Processing Systems*, 47(1):15–31, March 2007.

[10] T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560–576, 2003.