Reconfigurable SoC Design with Hierarchical FSM and Synchronous Dataflow Model

Sunghyun Lee † Sungjoo Yoo ‡ Kiyoung Choi †

† Design Automation Lab.
EECS, Seoul National Univ.
Seoul 151-742, Korea
{shlee,kchoi}@poppy.snu.ac.kr

‡ SLS Group, TIMA Lab.
46 Avenue Felix Viallet
38031 Grenoble, France
Sungjoo.Yoo@imag.fr

Abstract

We present a method of runtime configuration scheduling in reconfigurable SoC design. As a model of computation in system representation, we use a popular formal model of computation, hierarchical FSM (HFSM) with synchronous dataflow (SDF) model, in short, HFSM-SDF model. In reconfigurable SoC design with the HFSM-SDF model, the problem of configuration scheduling is challenging due to the dynamic behavior of the system such as concurrent execution of state transitions (by AND relation), complex control flow (in the HFSM), and complex schedules of SDF actor firing. Thus, compile-time static configuration scheduling may not efficiently hide configuration latency.

To resolve the problem, it is necessary to know the exact order of required configurations during runtime and to perform runtime configuration scheduling. To obtain the exact order of configurations, we exploit the inherent property of HFSM-SDF that the execution order of SDF actors can be determined before the execution of state transition of top FSM. After obtaining the order information in a queue called *ready configuration queue*, we execute the state transition. During the execution, whenever there is new available FPGA resource, a new configuration is selected from the queue and fetched by the runtime configuration scheduler. We applied the method to an MPEG4 decoder design and obtained up to 21.8% improvement in system runtime with a negligible overhead of runtime (1.4%) and memory usage (0.94%).

1 Introduction

Recently, reconfigurable systems design is gaining more and more attention [1][2][3][4]. Most of research focuses on optimization of reconfigurable resource utilization and architecture adaptability. To increase reconfigurable resource utilization, configuration latency needs to be minimized. To do that, configuration scheduling (and caching) [5][6][7][8][9], configuration compression [10], and coarse-grain reconfigurable architectures [3][11] have been studied.

In terms of design productivity of reconfigurable systems design, to master the ever growing complexity of SoC design, formal models of computation are becoming more and more important since they enable shorter design cycle by formal analysis (e.g. analysis of liveness, deadlock, maximum memory usage, etc.) as well as systematic reuse. In commercial SoC design tools, several formal models of computation are supported: CFSM in Cadence VCC [12], hierarchical FSM with dataflow in Synopsys CoCentric System Studio [13][14], etc. However, for the design of reconfigurable SoCs, the designers still use C or HDL codes without any specific formal models of computation, simple dataflow models [15][8][16], or general models of computation, e.g. process [17], discrete event models [18], etc. There are few formal models or design methodologies well set up for reconfigurability. Thus, in designing the ever increasingly complex future reconfigurable systems, designers will suffer from a severe productivity problem due to the lack of formal analysis and systematic design reuse that could be possible through the usage of formal models of computation.

In our work, we investigate reconfigurable SoC design with a popular formal model of computation, hierarchical FSM (HFSM) with synchronous dataflow (SDF), in short HFSM-SDF.[1] The HFSM-SDF model is well suited to design both complex control (by HFSM) and dataflow computation (by SDF). They also enable useful formal analysis including state reachability test, deadlock analysis with bounded memory, etc. Currently, commercial tools such as Synopsys CoCentric System Studio [13][14] and academic tools such as PtolemyII [19] support the HFSM-SDF model.

In reconfigurable SoC design with the HFSM-SDF model, the problem of configuration scheduling is challenging due to the dynamic behavior of the system such as concurrent execution of state transitions (by AND relation), complex control flow (in the HFSM), and complex schedules of SDF actor firing. Thus, compile-time static configuration scheduling such as [5][6] may not efficiently hide configuration latency.

In this paper, to resolve the problem, we present a method of runtime configuration scheduling. Section 2 gives a review of related work. Section 3 presents preliminaries of our work. Section 4 explains our problem. Section 5 addresses our method. Section 6 gives experimental results. Section 7 concludes the paper.

2 Related Work

Among numerous reconfigurable architectures, an architecture with uni-processor and reconfigurable resource (e.g. FPGA) has been widely studied and commercialized [1][2][3]. In our work, we use a processor/FPGA architecture to implement the HFSM-SDF specification.

For efficient configuration scheduling, configuration prefetch techniques are presented in [5][6][7]. They hide configuration latency by overlapping configuration fetch and useful computation. In [8], configuration reuse is accounted for in determining priorities of task scheduling. In [9], to maximize the reuse of configuration in loop-intensive applications, loop fission is exploited.

As models of computation used in reconfigurable systems specification and configuration scheduling, in [16], temporal partitioning and scheduling algorithms take a dataflow graph as input. In

[1] In the HFSM-SDF model, FSM and SDF can be nested arbitrarily as in PtolemyII and CoCentric System Studio.

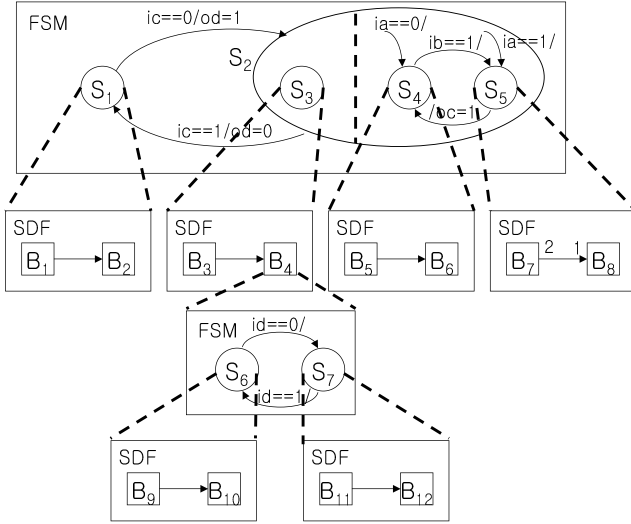


Figure 1: An example of HFSM-SDF model.

[8], task graphs are used in HW/SW partitioning and scheduling for reconfigurable systems design. In [5][6], a control flow graph is used to investigate the opportunity of configuration prefetch. In [15], CSP (communicating sequential process) model is used as an input description of stream computation. In [17] and [18], a process graph and discrete event model are used, respectively. In previous work, these models of computation (CSP, process, discrete event) give little opportunity to reducing configuration latency since they are more general than other specific models of computation such as SDF.

The HFSM-SDF model is a practical model of computation since it is well suited to design both complex control (by HFSM) and dataflow computation (by SDF) and it enables useful formal analysis such as state reachability test. To the best of the authors' knowledge, in reconfigurable SoC design area, there has been no previous work in using the HFSM-SDF model as the system input specification and no previous work in optimizing configuration scheduling of the HFSM-SDF model. Our contribution is to enable an optimized configuration scheduling in reconfigurable SoC design with the HFSM-SDF model.

3 Preliminaries

3.1 Hierarchical FSM and Synchronous Dataflow Model

Figure 1 shows an example of HFSM-SDF model description. In the figure, circles and squares represent states and SDF actors, respectively. An arc between states represents a state transition. A state transition arc is tagged with a guard/action. An arc between SDF actors is tagged with the number of tokens to be consumed (for input) and produced (for output). By default, arcs without numbers have single token production/consumption. In the figure, the rightmost SDF graph has an arc tagged with 2 and 1. In this case, actor B_7 produces two tokens for each firing of the actor and actor B_8 consumes one token when fired. Thus, to balance the number of produced tokens and that of consumed tokens, the SDF graph has a *schedule of actor firing*, in short schedule. There can be several candidates for the schedule. In the above case, we can have $B_7 2B_8$, $2B_7 4B_8$, etc. For each SDF graph, the designer sets one of them, as the schedule of the SDF graph.

At the top of the description, we have a top FSM consisting of two states S_1 and S_2.[2] State S_1 is refined to an SDF graph consisting of two actors, B_1 and B_2. State S_2 has two concurrent sub-FSMs. Inside the circle denoted by S_2, the vertical dashed line represents AND relation between two sub-FSMs. The AND relation makes the two sub-FSMs run concurrently.

State S_3 is refined to an SDF graph consisting of two SDF actors B_3 and B_4. The SDF actor B_4 in the graph is further refined to an FSM consisting of two states S_6 and S_7. Two states S_6 and S_7 are refined to SDF graphs, respectively. In the other sub-FSM of state S_2, states S_4 and S_5 are also refined to SDF graphs as shown in the figure.

The example in Figure 1 may look a little complex. However, in real SoC applications such as MPEG4 decoder, designers face even more complex HFSM-SDF representations. For instance, our implementation of MPEG4 decoder consists of 9 hierarchical FSMs, 31 states, 44 state transitions, 89 SDF actors (10 hierarchical actors and 79 leaf actors).

Details of HFSM-SDF model can be found in [20]. In this paper, to give a brief explanation of HFSM-SDF model execution, we summarize it with three rules and some terminology as follows.

Rule 1: Corresponding to a state transition of parent FSM, a child sub-FSM makes only one state transition.

This is a basic rule of hierarchical construction of FSMs. When there are sub-FSMs that have AND relation with each other, each sub-FSM takes a single state transition for a state transition of their parent FSM.

Rule 2: Whenever a (self or outgoing) state transition is made from a state, if the state is refined to an SDF graph, the schedule of the SDF graph is always executed once.

This rule is needed to conform to Rule 1 when a state is refined to an SDF graph. For instance, in Figure 1, when a (self or outgoing) state transition is made from state S_3, the schedule of SDF graph refining the state, i.e. the schedule of $B_3 B_4$ is always executed once. This rule is necessary to conform to Rule 1, especially when a state is refined to an SDF graph and the SDF graph has an SDF actor which is refined to an FSM. In this case, to conform to Rule 1, for a state transition of upper level FSM, the lower level FSM should also make a state transition. To do that, the schedule of the intervening SDF graph needs to be executed once.

In another word, this rule means that the schedule of child SDF graph is always executed, whichever (self or outgoing) transition is taken from the parent state that the child SDF refines. Thus, only if we can tell current FSM states, we can tell the schedule of SDF actor firings. In our work, we exploit this property to obtain an ordered sequence of required configurations, called a *ready reconfiguration queue*. More details will be given in Section 5.

Rule 3: When an SDF actor is refined to a sub-FSM, the sub-FSM makes a state transition at the last firing of the SDF actor in the schedule of the parent SDF graph.

This rule is needed to conform to Rule 1 when an SDF actor is refined to a sub-FSM. Rule 3 means that there are two types of actor firing for the SDF actor refined to a sub-FSM: one (called *TypeA* firing in [20]) that does not have the sub-FSM make a state transition and the other (called *TypeB* firing) that enables the state transition [20].

For instance, in the example of Figure 1, when the SDF graph consisting of B_3 and B_4 fires its schedule, $B_3 B_4$, the sub-FSM refining the SDF actor B_4 makes a state transition at the last firing of B_4 in the schedule, $B_3 B_4$. In this example, it looks trivial.

[2] At the top of the hierarchy, we can also have an SDF graph. However, even in this case, we can assume that we have a top FSM with only one state refined to the SDF graph and a self state transition as the execution of SDF graph. Thus, in this paper, for the simplicity of explanation, we assume that we have an FSM at the top of the hierarchy.

```
1 TopFSM::Run() {
2     switch(top_cur_state) {
3         case(S1):
4                 S1.RunSDFSchedule();
5                 if( ic==0 ) {
6                         top_cur_state = S2;
7                         if( ia==0 )           S2_sub_cur_state1 = S4;
8                         else                  S2_sub_cur_state1 = S5;
9                         od = 1;
10                }
11               break;
12        case(S2):
13               S3.RunSDFSchedule();
14               switch( S2_sub_cur_state1 ) {
15                       case(S4):
16                               S4.RunSDFSchedule();
17                               if( ib == 1 ) S2_sub_cur_state1 = S5;
18                               break;
19                       case(S5):
20                               S5.RunSDFSchedule();
21                               S2_sub_cur_state1 = S4; oc = 1;
22                               break;
23               }
24               if( ic == 1 ) { top_cur_state = S1; od = 0; }
25               break;
26   }
27 }
```

Figure 2: A code section of implemented HFSM code for the specification of Figure 1.

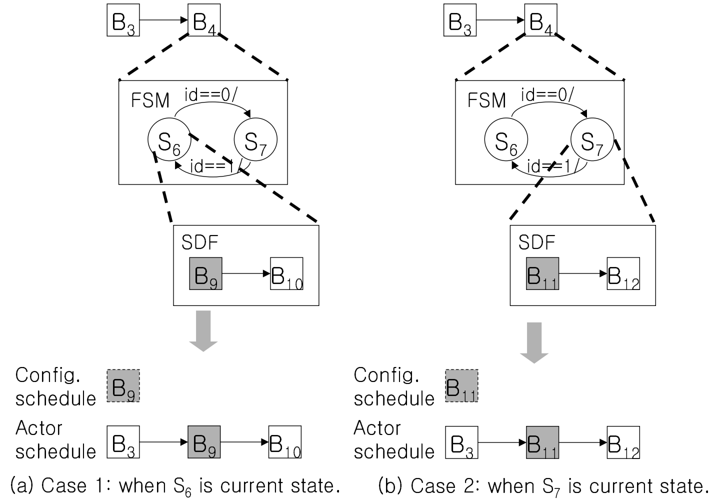


Figure 3: Examples of dynamically determined configuration order and corresponding configuration prefetch.

However, in the case that the SDF actor is fired several times in the schedule. The sub-FSM makes a single transition at the last firing of the SDF actor in the schedule.

The last terminology is *conditional initial transition*. It is needed to determine an initial state when entering a hierarchical FSM. It is denoted with an arc that does not have a source end but a destination end. In Figure 1, state S_4 and S_5 have conditional initial transitions. When entering state S_2, to determine which state (between S_4 and S_5) to enter, we evaluate the guards of conditional initial transitions. In this case, the current state can be S_4 (if ia == 0) or S_5 (if ia == 1). For more details of HFSM-SDF model, more generally, HFSM with concurrency models, refer to [20].

3.2 Target Architecture and Implementation

We implement an HFSM-SDF specification on an SoC target architecture consisting of a processor (ARM7) and FPGA (Xilinx Virtex). The FPGA allows partial runtime reconfiguration. Thus, we can run concurrently both configuration and computation on the FPGA. We can also run FPGA reconfiguration and processor computation concurrently.

Given an HFSM-SDF specification and HW/SW mapping of SDF actors by the designer, we implement the HFSM part on the processor. We implement SDF actors either as co-processors (running mutually exclusively with the computation on the processor) on the FPGA or as functions running on the processor.

From the HFSM model, we implement a sequential code by serializing concurrent execution of FSMs. Figure 2 shows a code section that implements the top HFSM of the example in Figure 1. To execute a state transition of the top FSM, the function TopFSM::Run() is called (line 1 in Figure 2). If the current state of the top FSM is S_1, then according to Rule 2 and 3, the refining SDF graph of state S_1 fires its schedule (S1.RunSDF() in line 4). In this case, the schedule is B_1B_2. Then, the guard of outgoing transition is evaluated (line 5). If the evaluation is true, then the outgoing state transition is performed (lines 6–9). Note that, in this case, since the state to be entered, S_2 has two sub-FSMs one of which has conditional initial transitions, we need to resolve the initial states of the sub-FSM. To do that, we evaluate the guards of conditional initial transitions and set a corresponding initial state (lines 7–8).

If the current state is S_2 (line 12), then both sub-FSMs can make state transitions. In the code of Figure 2, we first make the state transition of state S_3 (line 11). After making the state transition, we make the state transition of the other sub-FSM (consisting of state S_4 and S_5).

If the current state of the sub-FSM is S_4 (line 15), then we run the SDF graph refining the state S_4 (line 16). The guard of outgoing state transition is evaluated and, if true, the next state of the sub-FSM becomes S_5 (line 17). A similar code, for the case that the current state is S_5, is in lines 19–21. Finally, the guard of outgoing state transition of state S_2 is evaluated and, if true, the next state of top FSM becomes S_1 (line 24).

4 Problem

Required configurations are determined dynamically.
Figure 3 shows the SDF graph refining state S_3 and its sub-FSM in the example of Figure 1. In Figure 3, shaded rectangles represent that the corresponding SDF actors are implemented on the FPGA. Thus, configuration fetch is required before their execution. White rectangles are assumed to be executed on the processor as software functions.

In Figure 3 (a), we assume that the schedule of the SDF graph consisting of B_3 and B_4 is B_3B_4, that the sub-FSM refining the SDF actor B_4 has S_6 as its current state, and that the schedule of the SDF graph refining state S_6 is B_9B_{10}. In this case, the total firing order of SDF actors is $B_3B_9B_{10}$ as shown at the bottom of Figure 3 (a). In this case, in terms of configuration scheduling, the configuration latency of B_9 can be hidden (in part or in total) by overlapping the computation of B_3 (on the processor) and the configuration prefetch for B_9. In the figure, the configuration prefetch is represented by the dashed and shaded rectangle denoted by B_9.

Figure 3 (b) shows another case of SDF actor firing. Assuming that the sub-FSM has S_7 as its current state, the total firing order of SDF actors is $B_3B_{11}B_{12}$ as shown at the bottom of the figure. In this case, the configuration latency of B_{11} can also be hidden (in part or in total) by overlapping the computation of B_3 (on the processor) and the configuration prefetch (the dashed and shaded rectangle denoted by B_{11} in the figure.

Configuration Scheduling Problem
In terms of configuration scheduling, the real problem is that the above two cases are determined dynamically during system run depending on the current state of the sub-FSM. Thus, which configuration to prefetch (in the example, the configuration of B_9 or that of B_{11}) needs to be determined dynamically depending on the current state of the FSM.

In such a case, compile-time configuration prefetch can also be tried. However, compile-time solutions should resort to the prediction of next required configuration based on profiling information [5][6]. Moreover, when the prediction is failed, configuration prefetch based on the prediction suffers from penalty (i.e. canceling the previous configuration fetch and launching a new configuration fetch). In terms of system runtime, such a penalty can be prohibitive when the configuration latency is very large.

To reduce or, if possible, eliminate such a penalty, we need to know the exact order of required configurations. In our work, we achieve the goal by dynamically precomputing the order of required configurations. Since the precomputation gives the exact order of required configurations, our method does not suffer from the penalty from which prediction-based methods suffer.

5 Precomputation and Runtime Configuration Scheduling

5.1 Solution Overview

Our solution starts from the observation that if we can tell all the current states of FSMs in the HFSM-SDF specification, we can tell the exact order of all the SDF actor firings which will be made during the present state transition of top FSM.

Precomputation of ready configuration queue

For each state transition of top FSM, we evaluate all the current states of FSMs in the HFSM-SDF specification by traversing the hierarchy of HFSM. After all the current states of FSMs are known, we can build the exact firing order of SDF actors by constructing it with the SDF schedules of child SDF graphs, in a bottom up manner, from leaf SDF graphs. In the example of Figure 3 (a), the SDF graph refining the current state (S_6) gives SDF schedule B_9B_{10}. Then, we go up one level to the SDF graph consisting of B_3 and B_4. In the schedule of the SDF graph, B_3B_4, we substitute the firing of SDF actor B_4 with the schedule of its child SDF graph, B_9B_{10}. Finally, we obtain the schedule of $B_3B_9B_{10}$.

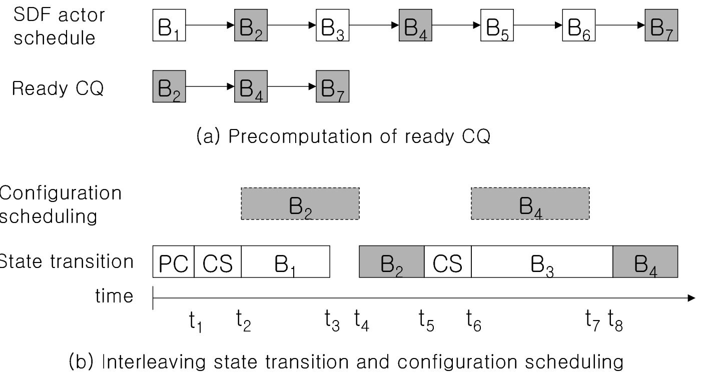
The construction gives only the exact order of all SDF actor firings for the execution of the present state transition of top FSM. Thus, the problem of configuration scheduling is not yet resolved. For configuration scheduling, what matters is the order of required configurations. We can build it by selecting all the SDF actors with FPGA implementation from the exact order of SDF actor firings. In the example of 3 (a), the order of required configurations becomes, trivially, B_9.

Note that since each state transition of top FSM can give different orders of required configurations, before each of state transition of top FSM, we obtain the order of required configurations. Note also that since the order of required configurations is exact, except cases of exception in HFSM-SDF run[3], there is no possibility of canceling configuration prefetch which is often the case in prediction-based methods for configuration prefetch [5][6].

Figure 4 exemplifies the execution of HFSM-SDF model and configuration scheduling. For each state transition of top FSM, a schedule of SDF actors is obtained. In Figure 4 (a), we assume that the schedule consists of eight SDF actors from B_1 to B_8. The arrows represent execution order. For instance, B_1 should be executed before B_2. After obtaining the order, an order of FPGA configurations is extracted. In the figure, shaded rectangles represent SDF actors implemented on the FPGA. Thus, an order of three configurations of actors, B_2, B_4 and B_7 is obtained. We call the ordered configurations *ready configuration queue (ready CQ)*.

Interleaving SDF actor firings and configuration scheduling

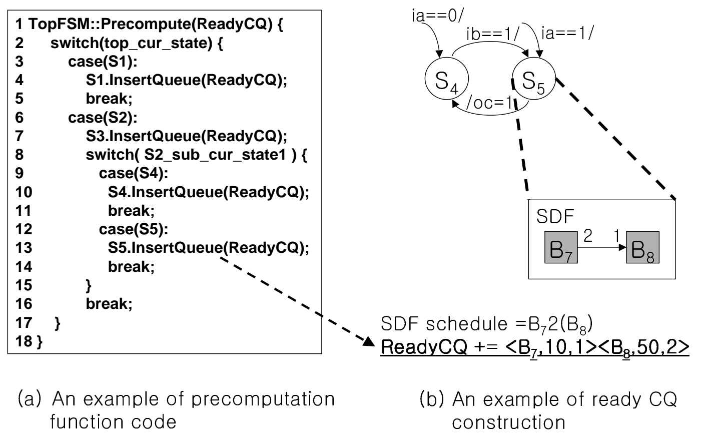
[3] During HFSM-SDF model execution, an exception can be caused by a sub-FSM. If the sub-FSM is refined to an SDF graph, in this case, depending on the levels of exception, the schedule of the SDF graph can be or cannot be executed. If it is not executed, we need to cancel all the fetches of configurations of the schedule. For more details of exception, refer to [20][14].



(a) Precomputation of ready CQ

(b) Interleaving state transition and configuration scheduling

Figure 4: Interleaving SDF actor execution and configuration scheduling.



(a) An example of precomputation function code

(b) An example of ready CQ construction

Figure 5: Examples of precomputation function and ready CQ.

The execution of state transition, i.e. the execution of SDF actors is interleaved with configuration scheduling as exemplified in Figure 4 (b). At the beginning of the state transition, the precomputation (PC) is performed to obtain the ready CQ until time t_1. Then, at time t_2, the configuration scheduler (CS) launches configuration fetch for the first configuration (in this example, the configuration of B_2). At the same time, the first SDF actor (in this example, B_1) starts to execute. As shown in the figure, the configuration fetch is overlapped with the SDF actor execution thereby the hiding configuration latency.

At time t_3, the SDF actor B_1 terminates its execution. The second SDF actor, B_2 needs to be executed. However, since the configuration fetch for the SDF actor has not been completed, the SDF actor waits for the configuration fetch to be completed. At time t_4, the configuration fetch is completed and the SDF actor B_2 starts to execute. At time t_5, since the SDF actor implemented on the FPGA terminates its execution and its configuration is no longer needed on the FPGA, the configuration scheduler is invoked to fetch a next necessary configuration (in this example, the configuration of B_4) onto the FPGA. Interleaving of SDF actor execution and configuration scheduling continues in this way.

In the next two subsections, we will explain the details of precomputation of ready CQ and runtime configuration scheduling.

5.2 Precomputation Function

Figure 5 (a) shows a pseudo code of precomputation function for the case of Figure 1. The pseudo code in Figure 5 (a) has the same

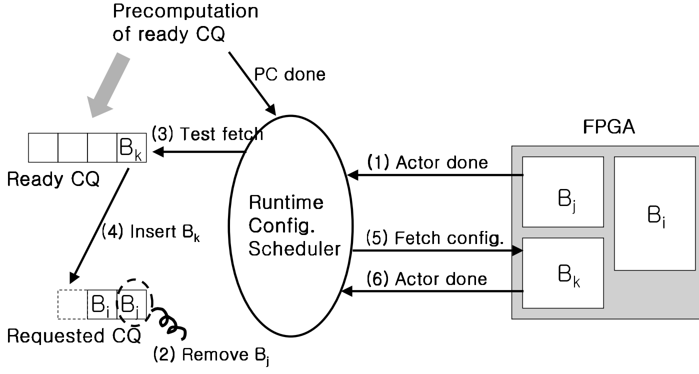


Figure 6: Runtime configuration scheduler.

```
1  int TestFetch( tuple elem ) {
2        if( elem.cost() + cur_cost() <= 100% &&
3           Controller.CheckConflict( elem.config_id ) == ok )
4                 return ok;
5        else
6                 return not_ok;
7  }
```

Figure 7: Testing whether configuration fetch is allowed or not.

control structure with the code of HFSM-SDF example given in Figure 2 since the precomputation function traverses the same hierarchy of HFSM. The traversal is performed in a depth-first manner and started by calling the function TopFSM::Precompute() (line 1 in Figure 5 (a)). During the traversal, all the current states of (sub-)FSMs are identified. It is to evaluate variables indicating current states (lines 3, 6, 9 and 12 in Figure 5 (a)). For each current state that is refined to an SDF graph, the ready CQ is incrementally constructed (lines 4, 7, 10 or 13).

Figure 5 (b) shows an example of obtaining the ready CQ. The figure shows, from the example of Figure 1, the FSM consisting of state S_4 and S_5 and the SDF graph refining state S_5. We assume that the current state of FSM is S_5 and that the SDF actors (B_7 and B_8) in the refining SDF graph are implemented on the FPGA. Thus, their configurations need to be fetched onto the FPGA. As shown in the figure, the schedule of SDF graph is $B_7 2 B_8$. In this case, for the schedule, the function **S5.InsertQueue(ReadyCQ)** in line 13 of Figure 5 (a) inserts into a ready CQ an ordered sequence of two tuples, $< B_7, 10, 1 >< B_8, 50, 2 >$. A tuple consists of $<$ configuration id, configuration cost in percentage, number of configuration requests $>$. Note that, in this case, by using the number of configuration requests in the tuple, we use a single tuple for two times of configuration request for B_8.

After the depth-first traversal, the precomputation function of top FSM gives the entire ready CQ (**ReadyCQ** in Figure 5 (a)). The runtime configuration scheduler (Section 5.3) will use the ready CQ to fetch configurations.

Note that the precomputation does not change the original HFSM. Compared with the code in Figure 2, Figure 5 (a) shows that the precomputation function does not set outputs, nor state variables. From the original code of HFSM, the precomputation function is made by extracting the hierarchy (control structure of the original code), and state variables (to read them).

5.3 Runtime Configuration Scheduler

The configuration scheduler is executed in one of three cases: when the precomputation of ready CQ is done, when an SDF actor terminates its execution on the FPGA, and when a configuration fetch is completed. It treats two queues, ready CQ and a queue called *requested configuration queue (requested CQ)*. The requested CQ is used to maintain the information of current configuration requests.

Figure 6 shows how the configuration scheduler runs. In the figure, an SDF actor B_j terminates its run on the FPGA and sends a done signal to the configuration scheduler (arrow number 1). The configuration scheduler checks to see if the configuration of the terminated SDF actor is still needed (this case will be explained later in this section). If not, it removes the request from the requested CQ (arrow number 2). The configuration scheduler checks the first tuple in the ready CQ to see if its configuration can be fetched onto the FPGA (arrow number 3 in Figure 6).

Figure 7 shows a pseudo code of such a test. Function **TestFetch** checks, first, to see if there exists enough FPGA space for the new configuration (line 2 in the figure). In the code, **cur_cost()** returns the current usage of FPGA in percentage. Although there is enough space (in terms of resource usage percentage) for the new configuration, the new configuration may conflict with existing configurations. The conflict is caused by the limitation of shared resource such as global wires, I/O pins, etc. Thus, we need to check the conflict (line 3).

If the new configuration is allowed to be fetched onto the FPGA, the corresponding tuple is moved from the ready CQ to the requested CQ (arrow number 4 in Figure 6). Then, the new configuration is fetched (in this case, for an SDF actor B_k, arrow number 5). The configuration scheduler runs in this way.

Note that a configuration request can contain multiple times of request for the same configuration. For instance, if the current tuple is $< B_k, 50, 2 >$, then the request contains two times of request for the configuration of SDF actor B_k that consumes 50% of FPGA cost.

In the operation of configuration scheduler, we need to pay attention to the following two cases.

- Case 1: when the requested configuration is already on the FPGA.
- Case 2: when the configuration of terminated SDF actor was requested multiple times.

In Case 1, if there is a new request for an existing configuration on the FPGA, then the request is marked as serviced and no configuration fetch is performed. Thus, we can reuse the existing configuration while preventing redundant configuration fetch.

To deal with Case 2, when an SDF actor terminates its execution on the FPGA, we check to see if the configuration of the terminated SDF actor was requested multiple times. The test is simple since a tuple in the requested CQ contains the number of configuration requests. For instance, if an SDF actor, B_j terminates its execution and if the configuration controller finds a tuple of $< B_j, 20, 2 >$ in the requested CQ, then it should keep the configuration of B_j since the configuration was requested two times and only the first request has been serviced. In this case, the scheduler keeps the configuration and decrements the number of configuration requests by one (2 to 1 in this example). Thus, the scheduler removes a configuration only when the configuration is no longer needed, i.e. the number of configuration requests reaches zero.

6 Experiments

We applied our method of precomputation and runtime configuration scheduling to an HFSM-SDF implementation of an MPEG4 natural video decoder [21]. The MPEG4 decoder is implemented on the SoC architecture consisting of an ARM7 processor and an FPGA (Xilinx XCV50E). We implemented the runtime configuration scheduler on the ARM7 processor. The termination of both SDF actor firing on the FPGA and configuration fetch is signaled to the runtime configuration scheduler on the processor via interrupt.

SDF actors	Gate count	CLB count	Exec_delay	Cfg_latency
IDCT	11980	222	1190	29568
IQUANT	1867	35	570	12672
Reconstruct	12000	223	2000	29568
AddBlockInter	2000	38	2000	12672
AddBlockIntra	2000	38	1500	12672

(a) Statistics of HW implementable functions

Mapping	IDCT	IQUANT	Reconstruc	AddBlockInter	AddBlockIntra
A	HW	HW	SW	SW	HW
B	HW	HW	SW	HW	SW
C	HW	HW	SW	HW	HW
D	HW	HW	HW	HW	SW
E	HW	HW	HW	SW	HW
F	HW	HW	HW	HW	HW

(b) HW/SW mappings

Mapping	Our method	No Prefetch	Gain(%)	OV_pre(%)	OV_sch(%)	OV_total(%)
A	46,840,471	53,691,605	**12.76**	0.66	0.62	**1.28**
B	47,677,649	46,973,684	**-1.4**	0.63	0.73	**1.36**
C	47,970,638	54,642,154	**12.2**	0.67	0.71	**1.38**
D	77,914,479	92,382,532	**15.66**	0.46	0.48	**0.94**
E	77,142,341	91,034,284	**15.26**	0.57	0.6	**1.17**
F	78,207,468	99,978,461	**21.78**	0.59	0.56	**1.15**

(c) Runtime (clock cycles), runtime gain and overhead

Mapping	Original (bytes)	OV (%)
A	41952	0.88
B	41920	0.88
C	41548	0.89
D	40054	0.92
E	40086	0.92
F	39682	0.94

(d) Memory usage overhead

Table 1: Comparison between no configuration prefetch case and our method.

We run 10 frames of MPEG4 decoding with a reference motion picture. The software-only implementation of MPEG4 decoder on the ARM7 processor consumes 82,633,799 clock cycles for the decoding. To have HW/SW mixed implementation, we change HW/SW mapping of five SDF actors. Table 1 (a) shows the execution delay and configuration latency of HW implementable SDF actors. Table 1 (b) shows six cases of HW/SW mapping of the SDF actors. For the other SDF actors of MPEG4 decoder, we map them on SW, i.e. on the ARM7 processor.

Table 1 (c) compares the runtimes of our method and those of no configuration prefetch.[4] Our method gives -1.4% to 21.8% (average 12.7%) performance improvement compared to the cases of no configuration prefetch. Among the six cases of HW/SW mapping, mapping B gives runtime decrease by 1.4% compared to the case of no configuration fetch. In this mapping, SDF actor IDCT, IQUANT, and AddBlockInter are mapped on the FPGA. Configurations of IDCT and AddBlockInter can reside on the FPGA at the same time. Those of IDCT and IQUANT can also reside on the FPGA at the same time. Configurations of AddBlockInter and IQUANT conflict with each other only once. Thus, the configuration fetch for each of the three SDF actors is needed only once. After the configuration, no reconfiguration is needed. Thus, the configuration controller run does not improve the runtime much. Instead, it adds runtime overhead by its runtime (1.4%).

The overhead of our method needs to be evaluated in terms of runtime and memory usage (code and data memory). Table 1 (c) shows the runtime overhead of precomputation function (OV_pre) and runtime configuration scheduler (OV_sch). The total runtime overhead (OV_total) is less than 1.4%. Table 1 (d) shows the memory usage overhead of precomputation function and runtime configuration scheduler. The overhead is less than 0.94%.

7 Conclusion

In this paper, we presented a method of runtime configuration scheduling for the implementation of hierarchical FSM with synchronous dataflow model. For each state transition of top FSM, first, we precompute the exact order of required configurations in a ready configuration queue by traversing the hierarchy of hierarchical FSM. Then, with the queue, a runtime configuration scheduler launches configuration fetches as early as possible during the execution of state transition of top FSM. We applied the presented method to a reconfigurable design of MPEG4 natural video decoder. Experimental results give up to 21.8% improvement in runtime with a negligible overhead of runtime and memory usage.

[4] Even in the case of no configuration prefetch, existing configuration on the FPGA can be reused. That is, when a configuration needs to be fetched, if the same configuration exists already on the FPGA, the existing configuration is reused for the new configuration request while preventing redundant configuration fetch.

References

[1] J. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 24–33, 1997.

[2] R. Maestre et al., "Kernel Scheduling in Reconfigurable Computing", *Proc. Design Automation and Test in Europe*, pp. 90–96, 1999.

[3] "Chameleon systems, inc.", available at http://www.chameleonsystems.com/.

[4] "Adaptive memory reconfiguration and management (amrm) homepage", available at http://www1.ics.uci.edu/ amrm/.

[5] S. Hauck, "Configuration Prefetch for Single Context Reconfigurable Coprocessors", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, 1998.

[6] Z. Li and S. Hauck, "Configuration Prefetch Techniques for Partially Reconfigurable Coprocessors with Relocation and Defragmentation", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2002.

[7] X. Tang, M. Aalsma, and R. Jou, "A Compiler Directed Appraoch to Hiding Configuration Latency in Chameleon Processors", *10th Int'l Confernce on Field Programmable Logic and Applications*, pp. 29–38, 2000.

[8] R. P. Dick and N. Jha, "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems", *Proc. Int'l Conf. on Computer Aided Design*, pp. 62–68, Nov. 1998.

[9] M. Kaul et al., "An Automated Temporal Partitioning and Loop Fission approach for FPGA Based Reconfigurable Synthesis of DSP Applications", *Proc. Design Automation Conf.*, pp. 616–622, June 1999.

[10] Z. Li and S. Hauck, "Configuration Compression for Virtex FPGAs", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2001.

[11] J. Rabaey, "Reconfigurable Computing: the Solution to Low Power Programmable DSP", *ICASSP Conference*, 1997.

[12] Cadence Design Systems, Inc., "Virtual Component Co-design (VCC)", available at http://www.cadence.com/products/vcc.html.

[13] Synopsys, Inc., "CoCentric System Studio", available at http://www.synopsys.com/products/cocentric_studio/cocentric_studio.html.

[14] J. Buck and R. Vaidyanathan, "Heterogeneous Modeling and Simulation of Embedded Systems in El Greco", *Proc. Int'l Workshop on Hardware-Software Codesign*, pp. 142–146, May 2000.

[15] E. Caspi et al., "Stream Computations Organized for Reconfigurable Execution (SCORE)", *Int'l Conf. on Field Programmable Logic and Applications (FPL '2000)*, Aug. 2000.

[16] K. M. Gajjala Purna and D. Bhadia, "Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers", *IEEE Transactions on Computers*, vol. 48, no. 6, pp. 579–590, June 1999.

[17] G.J.M. Smit et al., "Future Mobile Terminals: Efficiency by Adaptivity", *Int'l Workshop on Mobile Communications in Perspective*, Feb. 2001.

[18] J. Noguera and R. M. Badia, "A HW/SW Partitioning Algorithm for Dynamically Reconfigurable Architectures", *Proc. Design Automation and Test in Europe*, pp. 729–734, 2001.

[19] "The Ptolemy Project", available at http://ptolemy.eecs.berkeley.edu/.

[20] B. Lee, "Specification and Design of Reactive Systems", *Ph.D. thesis, Memorandum UCB/ERL M00/29, Electronics Research Laboratory, Univ. of California, Berkeley*, May 2000.

[21] "MPEG4 Industry Forum", available at http://www.m4if.org/.