論文2003-40SD-8-11

Hierarchical FSM과 Synchronous Dataflow Model을 이용한 재구성 가능한 SoC의 설계

(Reconfigurable SoC Design with Hierarchical FSM and Synchronous Dataflow Model)

李 成 賢 *, 劉 承 周 **, 崔 起 榮 *

(Sung-Hyun Lee, Sung-Joo Yoo, and Ki-Young Choi)

요 약

본 논문은 최근에 많이 사용되는 정형 계산 모델 중 하나인 hierarchical FSM (HFSM)과 synchronous dataflow (SDF) 모델(줄여서 HFSM-SDF)을 이용한 재구성 가능한 SoC 설계에서 실시간 구성 스케줄링(configuration scheduling) 방법을 제시한다. HFSM-SDF 모델을 이용한 재구성 가능한 SoC 설계에서는 HFSM이 갖는 동적인 특성들(예를 들면, AND 관계에 의해 동시에 일어나는 state transition, HFSM이 갖는 복잡한 control flow, 그리고 그에 따른 SDF actor firing의 복잡한 스케줄 등)로 인해 구성 스케줄링이 어려운 일이 된다. 그리고 이러한 동적인 특성들로 인해 정적인 구성 스케줄링 방법을 이용해서는 구성에 의한 지연(configuration latency)을 적절히 감추는 것이 어렵다. 본 논문에서는, 이 문제를 해결하기 위해, 실시간에 정확한 구성 순서를 찾은 후, 이를 이용한 동적인 구성 스케줄링 방법을 제안한다. 우선, 실시간에 필요한 구성 순서를 찾기 위해서는, HFSM-SDF 모델이 갖는 특징, 즉, SDF actor들의 실행 순서(firing schedule)는 최상위 FSM의 state transition 직전에 알 수 있다는 점을 이용할 수 있다. 이렇게 최상위 FSM의 매 transition마다 SDF actor들의 구성 순서를 찾아, ready configuration queue(ready CQ)에 저장한 후에, 전체 시스템의 state transition을 수행하며, 이 과정에서 FPGA에 (기존에 FPGA를 점유하고 있던 SDF actor의 종료 등으로 인해) 공간이 남으면, 실시간 구성 스케줄러는 ready CQ를 살펴보고, 필요한 구성을 다운로드한다. 본 논문에서 제시한 실시간 구성 방법을 MPEG4의 natural video decoder와 IS95의 modem 예제에 적용해 본 결과, 수행 시간이 최대 21.8%까지 향상되었으며 메모리 사용의 부담은 무시할 수 있을 정도였다.

Abstract

We present a method of runtime configuration scheduling in reconfigurable SoC design. As a model of computation, 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 HFSM-SDF model, the problem of configuration scheduling becomes challenging due to the dynamic behavior of the system such as concurrent execution of state transitions (by AND relation), complex control flow (HFSM), and complex schedules of SDF actor firing. This makes it hard to hide configuration latency efficiently with compile-time static configuration scheduling. 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 executing the state transition of top FSM. After obtaining the order information and storing it in the ready configuration queue (ready CQ), we execute the state transition. During the execution, whenever there is FPGA resource available, a new configuration is selected from the ready CQ and fetched by the runtime configuration scheduler. We applied the method to an MPEG4 decoder and IS95 design and obtained up to 21.8% improvement in system runtime with a negligible overhead of memory usage.

Keyword : formal model, FSM, SDF, reconfigurable, SoC, configuration prefetch, scheduling

* 正會員, 서울大學校 電氣컴퓨터工學部
(School of Electrical Engineering and Computer Science, Seoul National University)
** 正會員, SLS Group, TIMA, France

※ 본 연구는 한국과학재단 목적기초연구(98-0101-04-01-3)지원으로 수행되었음.
接受日字:2002年10月31日, 수정완료일:2003年8月5日

Ⅰ. 서 론

최근에 IC 설계의 비용이 급증함에 따라, 그에 대한 하나의 대안으로서, IC 제조 이후에도 재구성을 통해 새로운 기능을 추가할 수 있는 재구성 가능한 시스템 설계 관련 연구가 주목을 받고 있다[1-4]. 이들 연구의 초점은 대부분 재구성 가능한 자원의 효율적 이용과 아키텍처 수준의 유연성 등에 맞추어져 왔다. 특히, 재구성 자원을 효과적으로 이용하기 위해서는, 구성에 의한 시스템 수행 시간의 지연(configuration latency)을 최소화해야 하며, 이를 위해 구성 스케줄링(configuration scheduling)[5-9], 구성 압축(configuration compression)[10]. 큰 단위의 재구성 아키텍처(coarse grained reconfigurable architecture)[3, 11] 등이 연구되어 왔다.

재구성 시스템을 비롯해 SoC 설계의 복잡도가 점차 증가함에 따라, 설계 시간을 줄이고 설계 생산성을 높이기 위해, 정형 분석(formal analysis : 예를 들면, liveness, deadlock, maximum memory usage 분석 등)이 가능한 정형 계산 모델(model of computation)을 이용하는 것이 점점 중요해지고 있다. 이에 따라 CFSM (Cadence VCC)[12], hierarchical FSM with dataflow (Synopsys CoCentric System Studio)[13-14] 등, 상용 SoC 설계 환경에서도 여러 계산 모델을 지원하게 되었다. 그러나, 재구성 시스템을 설계하는 데에는, 아직 C나 HDL에서 시작하는 기존의 설계 방법[8, 15-16], 또는 process[17], discrete event[18]와 같은 일반적인 계산 모델 정형 계산 모델에 기반을 둔 설계 방법론은 전무한 실정이다. 따라서 설계자는 정형 분석 방법이나 시스템 차원의 재사용과 같은 설계 생산성을 높이기 위한 설계 방법론을 이용하지 못하므로, 앞으로 점차 복잡한 재구성 시스템을 설계할 때 심각한 설계 생산성 문제를 겪을 것으로 예상된다.

본 논문에서는 일반적으로 많이 사용되는 정형 계산 모델의 하나인 hierarchical FSM(HFSM)과 synchronous dataflow(SDF) (간단히, HFSM-SDF) 모델을 이용한 재구성 시스템의 설계 방법을 다룬다. HFSM-SDF 모델은 HFSM 모델을 사용함으로써 복잡한 control 시스템 설계에 적합함과 동시에 SDF 모델을 사용함으로써 dataflow 시스템의 설계에도 적합하다. 또한, state reachability, memory 크기 제약 조건하의 deadlock 분석 등과 같은 유용한 정형 분석 방법도 적용이 가능하다. 현재는, 상용 툴인 Synopsys의 CoCentric System Studio[13-14] 와 연구용 툴인 Ptolemy II[19] 등이 HFSM-SDF 모델을 지원한다.

HFSM-SDF 모델을 이용한 재구성 시스템 설계에서 구성 스케줄링 문제는 HFSM이 갖는 동적인 특성들 (AND 관계에 의한 state transition의 동시 수행, 복잡한 control flow, 복잡한 SDF actor 수행 스케줄 등)로 인해 어려워진다. 이러한 동적인 특성들로 인해 컴파일 시에 가능한 정적 스케줄링 방법[5-6]은 구성 지연을 효과적으로 감출 수 없다.

본 논문에서는 이를 해결하기 위해 실시간 구성 스케줄링 방법을 제시한다. 2장에서는 관련된 연구를 제시한다. 3장에서는 본 논문에서 사용하는 계산 모델을 간단히 설명한다. 4장에서는 그러한 계산 모델하에서의 구성 스케줄링 문제를 제시하며, 5장에서 이의 해결 방법을 제시한다. 그리고 6장에서 실험 결과를 보이며, 마지막으로, 7장에서 결론을 맺는다.

Ⅱ. 관련 연구

지금까지 제시된 여러 재구성 아키텍처들 중, 하나의 프로세서와 재구성 자원 (FPGA)으로 구성된 아키텍처가 가장 많이 연구되었으며, 상용화되었다[1-3]. 따라서 본 논문에서도 HFSM-SDF 모델로 기술된 시스템을 구현하기 위해 이러한 프로세서/FPGA 아키텍처를 가정한다.

효과적인 구성 스케줄링을 위해서 configuration prefetch 방법이 제시되었다[5-7]. 이 방법에서는 구성에

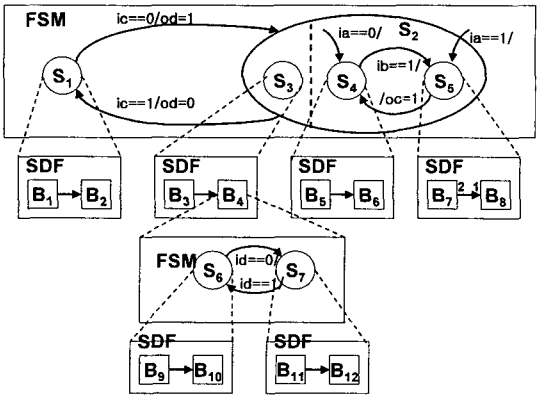


그림 1. HFSM-SDF model의 예
Fig. 1. An example of HFSM-SDF model.

의한 지연 (configuration latency)을 감추기 위해 시스템의 구성 시간과 유용한 연산 시간을 가능하면 겹치도록 한다. [8]에서는 현재 FPGA의 구성을 최대한 재사용하기 위해, FPGA의 구성 정보를 태스크 스케줄링의 우선순위 정보로 활용한다. [9]에서는 이와 관련해서 loop이 중요한 애플리케이션을 위해 loop fission 방법을 제시했다.

재구성 시스템의 기술과 관련해서는, [16]의 경우, 간단한 dataflow 그래프를 입력으로 시간적 분할 및 스케줄링 알고리즘을 제시했다. [8]에서는 task 그래프를 입력으로 하드웨어/소프트웨어 분할을 행했으며, [5][6]에서는 configuration prefetch 가능성을 찾아보기 위해 control flow 그래프를 이용했다. [15]에서는 stream 연산을 기술하기 위해 CSP(communicating sequential process) 모델을 이용했다. [17]과 [18]에서는 process 그래프와 discrete event 모델을 이용했다. 이전의 연구에서 사용된 계산 모델(CSP, process, discrete event)들은 본 논문에서 사용하는 SDF 모델과 달리 그 범용성으로 인해 구성 스케줄링의 여지가 거의 없다.

저자들은 [25]에 HFSM-SDF 모델을 이용한 재구성 가능한 SoC의 설계를 제시했다. HFSM-SDF 모델은 복잡한 control을 갖는 시스템과 dataflow 시스템을 모두 기술할 수 있으며, 또한 여러 유용한 정형 분석 방법을 적용할 수 있기 때문에 재구성 시스템을 기술하기에 매우 실용적인 모델이다. 본 논문은 [25]의 내용을 좀 더 자세히 설명하며, 현재까지 HFSM-SDF 모델을 이용한 재구성 시스템 설계 방법이 전무했으며, 특히 이를 기반으로 최적화된 구성 스케줄링 방법을 제안한다는 점에서 의미가 있다.

III. HFSM-SDF 모델 및 타겟 아키텍처

1. Hierarchical FSM with synchronous dataflow model

<그림 1>은 HFSM-SDF 모델을 이용한 시스템 기술의 예를 보여준다. 그림에서 원과 사각형은 각각 FSM의 state와 SDF actor를 의미한다. State들 사이에 존재하는 곡선은 state transition을 나타내며, transition이 일어나기 위한 조건(guard)과 transition시에 수행할 일(action)이 함께 기록되어 있다. SDF actor들 간의 연결은 dataflow를 의미하며, 각 actor에서 소모하거나 생산하는 토큰들의 수가 함께 기록되어 있다. 이러한 토큰 수가 없는 연결의 경우는 각 actor에서 소모 또는 생성하는 토큰의 수가 1임을 의미한다. 예를 들어, 그림의 가장 오른쪽에 있는 SDF 그래프는 2와 1이 기록된 연결을 갖고 있으며, 이것은 actor B7이 한번 수행 시에 2개의 토큰을 생산하며, actor B8이 한번 수행 시에 1개의 토큰을 소모함을 의미한다. SDF 그래프는 생산되는 토큰과 소모되는 토큰 간의 균형을 유지하기 위해 SDF actor 수행 스케줄을 가지며, 이를 간단히 스케줄이라 부른다. 이러한 스케줄은 하나의 SDF 그래프에 대해 여럿이 존재할 수 있다. 예를 들어, 위의 경우 $B_7 2B_8$, $2B_7 4B_8$ 등이 모두 유효한 스케줄이다. 설계자는 이러한 유효한 스케줄 중 하나를 선택해 SDF 그래프의 스케줄로 사용한다.

<그림 1>에 보인 HFSM-SDF 그래프의 가장 윗부분은 S_1과 S_2의 두 state를 갖는 FSM이다. State S1은 B1, B2 두 actor로 이루어진 SDF 그래프로 refine되었다. State S_2는 내부에 2개의 병렬적인 sub-FSM을 갖는다. S_2의 내부에 수직으로 그려진 점선은 이와 같은 AND 관계를 의미한다. 즉, AND 관계는 두 FSM이 동시에 수행되도록 만든다.

State S_3은 B_3, B_4의 두 actor로 이루어진 SDF 그래프로 refine되며, 이 중 B_4는 S_6, S_7의 두 state를 갖는 FSM으로 refine된다. 그리고 S_6, S_7은 각각 두 actor를 갖는 SDF 그래프로 refine된다. AND 관계에 있는 S_4, S_5로 구성된 또 다른 FSM은 각각 두 actor를 갖는 SDF 그래프로 refine된다.

<그림 1>의 HFSM-SDF 모델의 동작은 아래와 같이3가지 규칙에 의해 설명될 수 있다. HFSM-SDF 모델과 관련된 보다 자세한 내용은 [20]에서 찾아볼 수 있다.

규칙 1 : 부모 FSM의 state transition에 의해 자식 FSM은 오직 한번만의 state transition을 갖는다.

이 규칙은 FSM을 hierarchical하게 조직하기 위한 기본적인 규칙이다. 예를 들어, AND 관계를 갖는 자식 FSM의 경우, 부모 FSM의 state transition에 따라, 각각의 자식 FSM이 한번씩의 state transition을 갖는다.

규칙 2 : State가 SDF 그래프로 refine된 경우, 이 state로부터의 self 또는 outgoing transition에 의해 SDF 그래프의 스케줄이 한번 수행된다.

이 규칙은 FSM의 state가 SDF 그래프로 refine되었을 경우, 규칙 1을 만족시키기 위해 필요하다. 예를 들어, <그림 1>에서 S_3에서 state transition이 일어나는 경우, S_3에서 refine된 SDF 그래프의 스케줄, B3B4가 한번 수행된다. 이것은, B_4가 또 다른 FSM으로 refine되었으므로, 규칙 1을 만족시키기 위해서 SDF 그래프의 스케줄 B_3B_4가 한번 수행되어야 하기 때문이다. 즉, 일반적으로 SDF actor는 또 다른 FSM으로 refine될 가능성이 있기 때문에 규칙 1을 만족시키기 위해서는 중간에 위치한 SDF 그래프를 한번 수행하는 것이 필요하게 된다.

규칙 2는 자식 SDF 그래프는 부모 state가 self 또는 outgoing transition 중 어떤 transition을 행하더라도 한번씩 수행됨을 의미한다. 따라서 만약 현재 FSM의 state를 알 수 있다면 transition 시 수행되어야 하는 SDF 그래프의 스케줄을 알 수 있다. 본 논문에서는 필요한 구성 순서를 찾아내기 위해 HFSM-SDF의 이러한 특성을 이용한다. 이에 관한 자세한 논의는 5장에서 다룬다.

규칙 3: SDF actor가 자식 FSM으로 refine된 경우, 그 자식 FSM은 SDF 그래프의 스케줄 상에서 해당 SDF actor가 갖는 마지막 수행 시에만 transition을 행한다.

규칙 3은 SDF actor가 자식 FSM으로 refine된 경우, 규칙 1을 만족시키기 위해 필요하다. 규칙 3에 따르면, SDF actor의 수행은 자식 FSM이 state transition을 행하는 수행(type-A firing[20])과 transition을 행하지 않는 수행(type-B firing[20])의 두 가지로 나뉜다.

규칙 3에 따르면, <그림 1>에서 B_3, B_4로 이루어진 SDF 그래프의 스케줄 B_3B_4가 수행될 때, B_4가 갖는 자식 FSM의 state transition은 스케줄 B_3B_4 중에서 B_4의 마지막 수행 시에 일어난다. 이 예의 경우, B_4가 스케줄 상에서 한번밖에 수행되지 않으므로 당연한 것이지만, 경우에 따라 하나의 SDF actor가 스케줄 상에서 여러 번 수행되는 일이 존재한다. 이 경우, 규칙 1을 만족시키기 위해서, 이 SDF actor의 마지막 수행 시에 자식 FSM의 transition이 일어난다.

마지막으로, HFSM-SDF에는 조건부 초기 transition(conditional initial transition)이 존재한다. 이것은 hierarchical FSM에 들어갈 때, 초기 state를 결정하기 위해 필요하다. 이것은 source state는 없고 destination state만 존재하는 transition에 의해 표시된다. 예를 들어, <그림 1>에서 S_4와 S_5는 조건부 초기 transition을 갖는다. 즉, S_1에서 S_2로 진입할 때 S_4와 S_5 중 어떤 state를 초기 state로 사용할 것인가를 결정하기 위해 조건부 초기 transition이 갖는 조건을 확인해보고, 그 결과에 따라 S_4 또는 S_5가 초기 state가 된다. 이와 관련된 더 자세한 설명은 [20]에 언급되어 있다.

<그림 1>에 보인 HFSM-SDF 예제는 복잡해 보이지만 이를 이용한 실제 시스템의 기술은 더 복잡한 구조를 갖는 것이 일반적이다. 예를 들어, 본 논문에서 예제로 사용하는 MPEG4 natural video decoder의 경우, 총 31개의 state, 44개의 state transition을 갖는 9개의 hierarchical FSM과 총 89개의 actor(이 중 10개는 hierarchical actor이며, 나머지 79개는 leaf actor)를 갖는 여러 SDF 그래프들로 기술되었다.

2. 타겟 아키텍처와 HFSM-SDF의 구현

본 논문에서는 HFSM-SDF로 기술된 애플리케이션을 reconfigurable SoC의 실용적이며 간단한 형태중 하나인 프로세서와 reconfigurable logic으로 구성된 아키텍처에 구현한다. 이러한 형태는 Chameleon processor[3, 21] 등에서 찾아볼 수 있다. 실험을 위해서는, 실제적인 reconfigurable SoC 아키텍처를 모델링할 수 있도록, ARM7 프로세서 [26]와 Xilinx Virtex FPGA [28]로 구성된 시스템을 가정한다. Virtex FPGA는 Chameleon processor의 reconfigurable logic과 같이 부분적 재구성이 가능하다. 따라서 FPGA위에서 구성과 연산을 병렬적으로 수행할 수 있다고 가정했다. 또한, 프로세서 위의 연산과 FPGA 구성을 병렬적으로 수행하는 것도 가능하다. 하지만, 이와 같이 FPGA를 reconfigurable logic으로 사용하는 것은 구성의 단위가 작으므로, 구성 오버헤드가 커지는 단점이 있을 수 있다.

구현하려는 시스템의 HFSM-SDF 기술이 주어지고, 또한 이 기술에 대해 SDF actor들 간의 하드웨어/소프트웨어 분할이 결정되었을 때, HFSM 부분은 소프트웨어로 구현해 프로세서 위에서 수행하며, SDF actor 부분은 분할 결과에 따라 소프트웨어로 구현, 프로세서 위에서 수행하거나 하드웨어로 구현, FPGA 위에서 수행한다.

HFSM 부분의 구현과 관련해서는, FSM이 갖는 병렬적인 transition 들을 직렬화한 후 순차적인 코드로

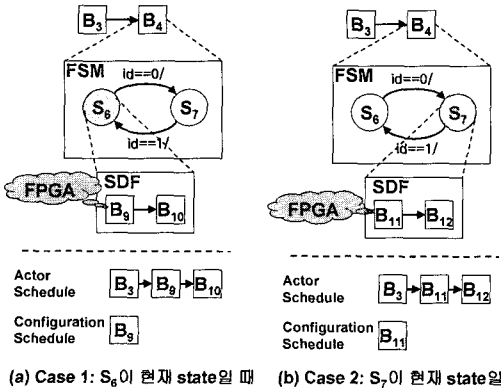
```
1:  TopFSM::Run ()
2:      switch ( top_cur_state ) {
3:          case S1:
4:              S1.RunSDFSchedule ();
5:              if ( ic==0 ) {
6:                  top_cur_state = S2;
7:                  if ( ia==0 )    S2_sub_cur_state1 = S4;
8:                  else            S2_sub_cur_state1 = 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: }
```

그림 2. <그림 1>의 HFSM-SDF model 구현 예
Fig. 2. A code section of implemented HFSM code for the specification of <fig. 1>.

구현한다. <그림 2>는 <그림 1>에 보인 HFSM-SDF 기술에서 top FSM 부분의 구현 예를 보여준다. Top FSM의 state transition을 위해서는 TopFSM::Run() 함수를 호출한다 (1번째 줄). 이 때, 만약 top FSM의 현재 state가 S_1이면 규칙 2와 3에 따라 S_1에서 refine된 SDF 그래프가 S1.RunSDF() 함수에 의해 수행된다 (4번째 줄). 이 경우, 수행되는 SDF 그래프의 스케줄은 B1B2이다. 그리고, S_1에서 나가는 transition의 조건을 테스트해보며 (5 번째 줄), 만약 그 결과가 참일 경우, S_2로의 state transition이 일어난다 (6-9 번째 줄). 이 예에서, S_2는 조건부 초기 transition을 갖는 자식 FSM을 포함하고 있으므로, 이 자식 FSM이 가질 state를 결정하기 위해 조건부 초기 transition이 갖는 조건을 테스트해보며, 그 결과에 따라 자식 FSM의 state를 결정한다 (7-8 번째 줄).

만약 top FSM의 현재 state가 S_2라면 (12번째 줄), AND 관계에 묶인 두 자식 FSM이 모두 state transition을 한다. <그림 2>에 보인 코드에서는 S_3으로 구성된 FSM이 먼저 transition을 수행한다 (11번째 줄). 그리고 S_4와 S_5로 구성된 FSM이 transition을 수행한다.

만약 이 자식 FSM의 현재 state가 S_4라면 (15번째 줄), S_4가 갖고 있는 SDF 그래프를 수행하며 (16번째 줄), outgoing transition의 조건을 테스트해본 후, 그



(a) Case 1: S_6이 현재 state일 때 (b) Case 2: S_7이 현재 state일 때

그림 3. 동적으로 결정되는 구성 순서의 예
Fig. 3. Examples of dynamically determined configuration order and corresponding configuration prefetch.

결과에 따라 이 FSM의 다음 state를 S_5로 결정한다 (17번째 줄). 자식 FSM의 현재 state가 S_6인 경우도 비슷한 형태로 구현된다 (19-21번째 줄). 그리고 마지막으로 S_2의 outgoing transition의 조건을 테스트해보고, 만약 참이면 다음 state를 S1으로 결정한다 (24번째 줄).

Ⅳ. 구성 스케줄링 문제

1. 필요한 구성 순서의 동적 결정

<그림 3>은 <그림 1>에서 state S_3을 refine하는 SDF 그래프와 이 그래프가 갖고 있는 자식 FSM을 보여준다. <그림 3>에서 노랑색으로 칠해진 사각형은 해당 SDF actor가 FPGA로 매핑되었음을 의미한다. 따라서 이 SDF actor의 수행을 위해서 FPGA의 구성이 필요하다. 투명한 사각형은 소프트웨어로 매핑된 SDF actor를 의미하며, 이것은 프로세서 위에서 수행된다.

<그림 3(a)>에서 B_3, B_4로 이루어진 SDF 그래프의 스케줄은 B_3B_4이며, B_4로부터 refine된 FSM의 현재 state는 S_6임을 가정한다. 또한, S_6을 refine한 SDF 그래프의 스케줄은 B_9B_{10}임을 가정한다. 이 경우, SDF actor들의 전체 수행 스케줄은 <그림 3(a)>의 아래에 보인 것과 같이 B3B9B10이 된다. 여기서, B_9를 위한 FPGA 구성 시간은 프로세서 위에서 수행되는 B_3의 수행 시간과 겹치도록 함으로써 부분적으로 또는 완전히 감출 수 있게 된다.

<그림 3(b)>는 SDF actor 수행의 또 다른 예를 보여준다. 만약 자식 FSM의 현재 state가 S_7이라면 SDF actor의 전체 수행 스케줄은 <그림 3(b)>의 아래에 보

인 바와 같이 B3B11B12가 된다. 이 경우에도 앞서와 같이 B_{11}의 구성 시간을 B_3의 수행시간과 겹치도록 함으로써 부분적으로 또는 완전히 감추는 것이 가능하다.

2. 구성 스케줄링 문제

구성 스케줄링에 있어, 가장 큰 문제는 앞서 설명한 것과 같이 자식 FSM의 state에 따라 필요한 구성 순서가 시스템의 수행 중에 동적으로 결정된다는 것이다. 따라서 현재 어떤 구성을 FPGA에 다운로드할 것인가 (<그림 3>의 예에서는, B_9 또는 B_{11})하는 문제는 각 FSM의 현재 state에 따라 동적으로 결정되어야 한다.

이 경우, 컴파일 시의 정적인 구성 스케줄링도 시도해볼 수 있다. 그러나 컴파일 시의 스케줄링 방법은 결국 프로파일링한 결과에 따라 다음에 필요한 구성을 예측할 수밖에 없다[5-6]. 그리고 이러한 예측이 틀렸을 경우, 현재 FPGA에 다운로드된 구성을 지우는 등의 페널티를 갖게 된다. 이러한 페널티는 구성 시간에 비례해서 커지게 되며, 만약 시스템에서 구성 시간이 갖는 비중이 상당히 큰 경우, 컴파일시의 구성 스케줄링 방법 자체를 적용할 수 없게 된다.

따라서, 이러한 페널티를 줄이거나, 혹은 완전히 없애기 위해서는, 현재 시스템이 필요로 하는 정확한 구성 순서를 알아야 한다. 본 논문에서는 시스템의 수행 중에 동적으로 precomputation을 이용해 현재 필요한 구성 순서를 찾아낸다. 여기서 사용되는 precomputation은 필요한 구성 순서를 정확하게 찾아내므로, 본 논문에서 제시하는 구성 스케줄링 방법은 앞서와 같은 예측에 기반한 구성 스케줄 방법이 갖는 페널티를 갖지 않는다.

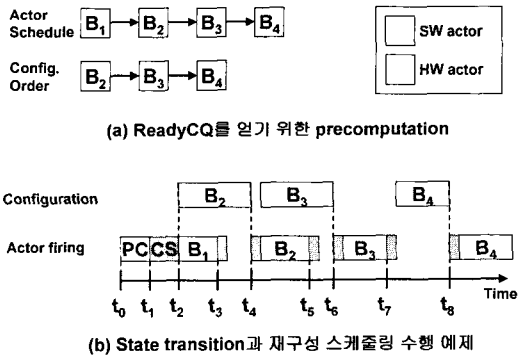


그림 4. SDF actor의 수행과 구성 스케줄러의 교차 수행 예

Fig. 4. Interleaving SDF actor and configuration scheduling.

V. Precomputation과 실시간 구성 스케줄링

1. 해결 방법에 대한 overview

본 논문에서 제시하는 구성 스케줄링 문제에 대한 해결 방법은 HFSM-SDF 모델에서 각 FSM에 대해 현재 state를 알 수 있다면, 현재 수행되어야 하는 SDF actor들의 수행 순서도 역시 알 수 있다는 발견에서 시작한다. 이것은 top FSM의 state transition시에 일어난다.

a) Ready configuration queue (ready CQ)를 얻기 위한 precomputation

Top FSM의 각 state transition에 대해, 현재 시스템이 갖는 FSM을 모두 방문해봄으로써 각 FSM의 현재 state를 알아낸다. 이런 방법으로 모든 FSM의 현재 state가 알려지면, leaf SDF 그래프부터 방문하기 시작해 점차 위로 올라가며 각 SDF 그래프의 스케줄을 모음으로써 현재 수행되어야 할 SDF actor들의 정확한 수행 순서를 알아낼 수 있다. 예를 들어, <그림 3(a)>에서 현재 state S_6을 refine한 SDF 그래프의 스케줄은 B_9B_{10}임을 알 수 있다. 이로부터 한 단계 위로 올라가 B_3, B_1로 이루어진 SDF 그래프를 만나면, B_4의 자식 SDF에 해당하는 스케줄인 B_9B_{10}을 이용해 전체 SDF actor의 스케줄 B3B9B10을 얻을 수 있다.

이렇게 해서 얻은 SDF actor들의 수행 스케줄은 단지 현재 수행되어야 하는 SDF actor들의 수행 순서에 불과하며, 아직 구성 스케줄 문제는 남아 있다. 구성 스케줄링 문제에 있어 의미가 있는 것은 단지 구성이 필요한 SDF actor들 간의 순서이다. 이것은 앞서 얻은 SDF actor들의 수행 순서를 앞에서 뒤로 검색하면서 FPGA에 매핑된 SDF actor들만을 선택해냄으로써 찾아낼 수 있다. <그림 3(a)>의 예에서 필요한 구성 순서는, 따라서, B_9가 된다.

여기서 주의해야 할 것은, top FSM의 state transition에 따른 구성 순서는 각 FSM의 현재 state에 따라 달라질 수 있으므로 각 top FSM의 transition 직전에 구성 순서를 얻어내야 한다는 점이다. 또한, HFSM-SDF의 수행 중에 발생할 수 있는 예외 상황[1]

1) HFSM-SDF 모델의 수행에 있어, sub-FSM에 의해

을 제외하고는 이렇게 얻어낸 구성 순서는 정확하므로 예측 기반의 구성 스케줄링과 달리 configuration prefetch를 취소할 필요가 전혀 없다는 점도 주목해야 한다[5,6].

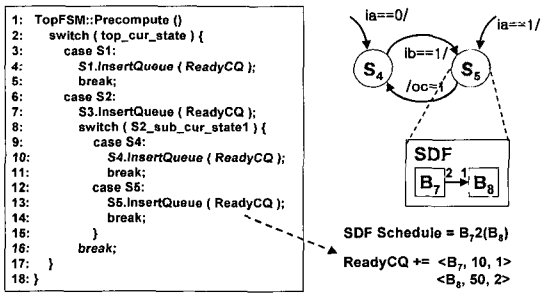
<그림 4>는 HFSM-SDF 모델과 구성 스케줄의 수행 예를 보여준다. Top FSM의 각 state transition에 대해, SDF actor들의 수행 순서가 얻어진다. <그림 4(a)>에서는 그 수행 순서가 B_1부터 B_4에 이르는 SDF actor들로 이루어져 있다고 가정한다. 그림에서 화살표는 수행 순서를 가리킨다. 예를 들어, B_1은 B_2 보다 먼저 수행되어야 한다. 이렇게 모든 SDF actor들의 수행 순서를 얻은 후에는 FPGA위에 구성되는 순서를 찾아낸다. 그림에서 노란색으로 칠해진 B_2, B_3, B_4 actor들이 이에 해당하며, $B_2B_3B_4$의 구성 순서를 얻는다. 본 논문에서는 이렇게 얻어진 구성 순서 정보를 이후에 설명될 구성 스케줄러와 공유하기 위해 ready CQ에 저장한다.

b) SDF actor의 수행과 구성 스케줄링의 교차 삽입

State transition의 수행, 즉, SDF actor들의 수행은 <그림 4(b)>에 보인 것과 같이 구성 스케줄링과 교차 삽입되어 있다. 그림에서는, top FSM의 state transition이 시작할 때, precomputation (PC)이 t_1까지 수행되며, 그 결과로 ready CQ를 얻는다. 그 후, 구성 스케줄러는 이 ready CQ를 이용해 t_2에서 필요한 구성 (B_2)을 FPGA에 다운로드하기 시작한다. 그리고 이와 동시에 프로세서로 매핑된 첫 번째 SDF actor B_1이 수행을 시작하며, 그 결과 그림과 같이 B_2의 구성 시간과 B_1의 수행 시간이 겹치게 된다.

시간 t_3에서 B_1의 수행이 완료되면 SDF actor B_2의 수행을 시작해야 하지만, 아직 B_2의 구성이 끝나지 않은 상태이므로 이를 시작할 수 없고, 구성이 완료되기를 기다린다. 그리고 시간 t_4에서 B_2의 구성이 끝나면 B_2의 수행을 시작하며, 이와 함께 현재 FPGA에 B_3을 구성할 수 있는지 테스트한 후 B_3의 구성을 시작한다. 시간 t_5에서 FPGA위에서 수행하는 actor B_2가 종료하면 B_2는 현재 transition에서 더 이상 사용되지 않으므로 FPGA 위에 B_2가 남아있을 이유가 없게 되며, 구성 스케줄러가 깨어나 그 다음에 필요한 B_4 actor의 구성을 시도한다. 그러나, 시간 t_5에서는 FPGA에 B4의 구성을 위한 공간이 부족하므로, 구성을 시작할 수 없으며, 후에 B_3의 수행이 완료되면 B_2와 B_3가 점유하고 있던 공간을 이용해 B_4의 구성을 시작한다. 이러한 방법으로 HFSM-SDF 모델의 수행은 SDF actor의 수행과 구성 스케줄러가 교차 삽입되며 진행된다.

다음 두 장에서 ready CQ의 precomputation 방법과 실시간 구성 스케줄링 방법에 대해 더 자세히 다룬다.



(a) Precomputation function 예제 (b) ReadyCQ 구성 예제

그림 5. Precomputation function과 ready CQ 예제
Fig. 5. Examples of precomputation function and ready CQ.

2. Precomputation 함수

<그림 5(a)>는 <그림 1>을 위한 precomputation 함수의 가상 코드를 보여준다. 그림에 보인 precomputation 함수는 <그림 2>에 보인 HFSM-SDFM 구현 예와 같은 구조를 갖는다. 이것은 precomputation 함수가 HFSM 구현 방법과 같이 각 FSM을 모두 방문하기 때문이다. 이렇게 각 FSM을 방문하는 것은 depth-first 방법으로 이루어지며, TopFSM::Precompute() 함수를 호출함으로써 시작된다 (<그림 5(a)>의 1번째 줄). 이 과정에서 모든 자식 FSM들의 현재 state를 파악하며, 이것은 각 자식 FSM의 현재 state를 저장하고 있는 state 변수들의 값을 읽음으로써 가능하다 (<그림 5(a)>의 3, 6, 9, 12번째 줄). 그리고 만약 현재 state가 SDF 그래프로 refine된 경우, <그림 5(a)>의 4, 7, 10, 13번째 줄에 보인 것처럼 점증적으로 ready CQ를 구성해나가게 된다.

<그림 5(b)>는 ready CQ를 얻는 방법을 보여준다. <그림 5(b)>는 <그림 1>에서 S_4, S_5 state와 S_5를

---

*exception*이 발생할 수 있다. 만약 이러한 sub-FSM이 SDF 그래프로 refine된 경우, exception의 중요도에 따라, SDF 그래프가 수행되거나, 수행되지 않을 수 있다. 만약, SDF 그래프가 수행되지 않을 경우, 앞으로 남은 구성 스케줄을 모두 취소해야 한다. Exception과 관련한 더 자세한 내용은 [14][20]에 기술되어 있다.

refine한 SDF 그래프를 보여준다. 여기서 이 FSM의 현재 state가 S_5이며, SDF actor B_7, B_8이 모두 FPGA에 매핑되었다고 가정한다. 따라서 B_7, B_8은 모두 수행 전에 FPGA 위에 구성 되어야 한다. 그리고 그림에 보인 바와 같이 이 SDF 그래프의 스케줄은 $B_7 2B_8$이다. 이 경우, <그림 5(a)>의 13번째 줄에 보인 것처럼 S5.InsertQueue(ReadyCQ)함수는 <B7, 10, 1>, <B8, 50, 2> 튜플을 순서대로 readyCQ에 삽입한다. 각 튜플은 <구성 번호, 구성 면적, 구성 요구 회수>을 의미한다. 예를 들어, B_8의 경우, SDF 그래프의 스케줄에 따라 2회 연속 수행되어야 하므로, 이것이 구성 요구 회수에 반영되어 있다.

이와 같이 depth-first 방법으로 모든 자식 FSM을 방문한 후, top FSM의 precomputation 함수는 모든 SDF 그래프의 스케줄을 담고 있는 ready CQ를 얻는다 (<그림 5(a)>의 ReadyCQ). 이후에 구성 스케줄러는 필요한 구성을 FPGA에 다운로드하기 위해 이 ready CQ를 이용하게 된다.

여기서 주의할 것은, precomputation 함수가 원래 HFSM의 구조를 바꾸지는 않는다는 점이다. <그림 2>에 보인 코드와 비교해서 <그림 5(a)>의 precomputation 함수는 어떤 값을 출력하거나 state 변수를 바꾸지 않는다. Precomputation 함수는 원래 HFSM 코드로부터 HFSM이 갖는 구조 (hierarchy) 정보와 state 변수 정보만을 이용해 만들어진다.

3. 실시간 구성 스케줄러

구성 스케줄러는 다음의 세 가지 경우에 수행된다.

- Ready CQ의 precomputation이 끝났을 때
- FPGA상에서 SDF actor의 수행이 끝났을 때
- Configuration prefetch가 끝났을 때

구성 스케줄러는 Ready CQ와 구성 요청 queue (requested configuration queue)의 두 가지 queue를 이용한다 (구성 요청 queue는 현재 FPGA위에 구성하도록 요청된 구성들을 담고 있으며, 프로세서 외부에 HW로 존재하는 configuration controller는 이를 이용, 필요한 구성의 다운로드를 시작한다).

<그림 6>은 구성 스케줄러의 수행 방법을 보여준다. 우선, precomputation이 완료되면, Ready CQ에 Bi, Bj, Bk actor가 FPGA에 구성을 필요로 한다는 정보가 저장되며, 이를 바탕으로 FPGA의 크기 등에 따라 구성이 가능한 actor들을 FPGA에 구성한다 (이 경우, Bi, Bj). 그 후에 예를 들어, SDF actor Bj가 FPGA 상에서 수행을 끝내면, 구성 스케줄러에게 완료되었음을 알린다 (화살표 (1)). 그러면, 구성 스케줄러는 현재 완료된 SDF actor의 구성이 앞으로 더 필요한지를 살펴본

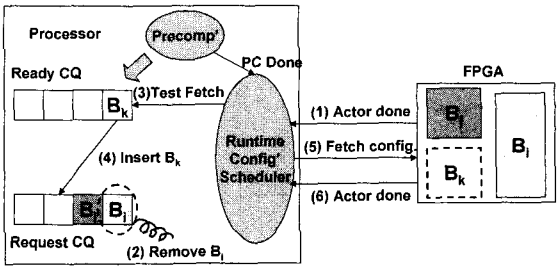


그림 6. 실시간 구성 스케줄러
Fig. 6. Runtime configuration scheduler.

```
1:  int TestFetch ( tuple elem )
2:      if ( elem.cost() + cur_cost() <= 100% &&
3:           Controller.CheckConflict ( elem.config_id ) == OK )
4:          return OK;
5:      else      S1.InsertQueue ( ReadyCQ );
6:          return not_OK;
```

그림 7. Configuration fetch 테스트
Fig. 7. Testing whether configuration fetch is allowed or not.

후 (이 부분은 이 장의 뒷부분에서 좀 더 자세히 다룬다), 그렇지 않다면, 구성 요청 queue에서 Bj에 해당하는 구성 요청을 지운다 (<그림 6>에서 화살표 (2)). 그리고 ready CQ에서 첫 번째 튜플을 보고, 현재 FPGA에 구성이 가능한지 여부를 파악한다 (<그림 6>에서 화살표 (3)).

<그림 7>은 <그림 6>의 화살표 (3)에 해당하는 테스트 관련 가상 코드를 보여준다. TestFetch() 함수는 먼저 현재 FPGA 내에 새로운 구성을 다운로드할 수 있는 여유 공간이 있는지를 살펴본다 (2번째 줄). 여기서 cur_cost() 함수는 현재 FPGA를 점유 정도를 백분위로 알려준다. 그러나 이렇게 FPGA에 여유 공간이 남아 있다 하더라도, 새로운 구성은 기존에 FPGA에 존재하는 구성들과 충돌을 일으킬 수 있다. 이러한 충돌은 각 구성이 사용하는 전역 연결들, 입출력 핀들 등에 의해 발생한다. Controller. CheckConflict() 함수는 이러한 충돌이 발생하는지의 여부를 파악한다 (3번째 줄).

이렇게 일련의 테스트를 거쳐, 새로운 구성이 FPGA에 다운로드가 가능하게 되면, 이 구성은 ready CQ에서 구성 요청 queue로 옮겨진다 (<그림 6>에서 화살표 (4)). 그리고, 구성 요청 queue에 있는 구성들은

configuration controller 하드웨어에 의해 FPGA에 다운로드 된다. 이러한 방법으로 구성 스케줄러가 동작한다.

여기서 주의할 것은 하나의 구성 요청이 같은 구성에 대한 여러 개의 요청을 포함할 수 있다는 점이다. 예를 들어, 튜플 <Bk, 50, 2>는 FPGA의 50%를 차지하는 Bk에 대한 구성 요청이 연속해서 2회라는 것을 의미한다.

이와 관련해서 구성 스케줄링 과정에서 다음의 2 가지 경우를 구분해야 한다.

- 경우 1: 요청된 구성이 이미 FPGA에 존재하는 경우
- 경우 2: 현재 완료된 SDF actor의 구성 요청이 여러 번 이루어진 경우

경우 1에서는 단순히 이미 요청된 구성이 이미 FPGA에 존재하므로 이 요청을 무시하며, 새로운 configuration prefetch는 일어나지 않는다. 즉, 이 경우 현재 FPGA에 존재하는 구성을 재사용할 수 있게 된다.

경우 2에서는, SDF actor가 수행을 완료했을 때 구성 요청 queue를 살펴봄으로써 현재 구성에 대한 또 다른 요청이 있는지의 여부를 파악한다. 예를 들어, 임의의 SDF actor Bj가 그 수행을 완료했을 때 구성 스케줄러는 구성 요청 queue에서 <Bj, 20, 2>라는 튜플을 찾게 되며, 이 경우 2번의 요청이 있지만, 현재 1번만 수행되었으므로 이 구성을 FPGA에 지우지 않고 그대로 두며, 요청 횟수만을 1만큼 감소시킨다. 따라서 구성 스케줄러는 어떤 구성이 더 이상 필요하지 않게 되었을 때, 즉, 구성 요청 횟수가 0이 되었을 경우에만 구성 요청 queue와 FPGA에서 해당하는 구성을 지우게 된다.

VI. 실험 결과

본 논문에서 제시한 precomputation과 실시간 구성 스케줄링 방법을 HFSM-SDF로 기술된 MPEG4 natural video decoder [22]와 IS95 [23][24] modem의 reverse link 부분에 적용했다 (MPEG4 natural video decoder 부분의 HFSM-SDF 기술은 appendix에서 간단히 설명한다). 타겟 아키텍처로는 ARM7 [26]프로세서와 Xilinx FPGA [28] (MPEG4는 Xilinx XCV50E, IS95는 XCV100E)를 가정했다. HFSM-SDF 모델에서 소프트웨어에 매핑된 부분은 C++ 코드로 구현되었으며, ARM7 simulator (ARM사의 armsd [27]) 위에서 수행한 후, 수행 시간 등을 얻었다. FPGA로 매핑된 부분은 VHDL 코드로 구현해 수행 cycle 수를 얻었으며, Xilinx사의 ISE Foundation [29]툴로 합성한 후, CLB 개수 등을 얻었다. FPGA 구성 다운로드를 위해서는 Virtex FPGA가 frame 단위로 구성되므로, 각 FPGA에 구현된 부분이 가장 적은 수의 frame에 매핑되도록 했으며, 이에 따라 필요한 frame수와 각 frame이 요구하는 구성 byte수에 의해 구성 시간 (*configuration latency*)이 결정된다[30]. 전체 시스템의 시뮬레이션을 위해서는, FPGA로 매핑된 부분의 동작을 C++ 코드로 모델링한 후 cycle 정보를 반영했으며, armsd의 메모리와 인터럽트 인터페이스를 통해 소프트웨어 부분과 통신하도록 구현했다. 타겟 아키텍처에서 SDF actor의 수행 완료와 FPGA로의 구성 다운로드 완료 등의 정보는 interrupt를 이용해 프로세서 위에서 interrupt handler로 구현된 구성 스케줄러에게 전달된다.

a) MPEG4 실험 결과

MPEG4 natural video decoder 예제는 HFSM-SDF로 모델링한 후, 이를 기반으로 해서 손으로 decoder를 구현했다. 실험의 입력 데이터로는 MPEG4의 QCIF 포맷의 reference 동화상 10 프레임을 이용했다. 이 경우, SW 만으로 구현된 decoder는 82,633,799 cycle의 수행 시간이 소요됐다. 본 논문에서 제시한 구성 스케줄링을 적용하기 위해, <표 1(b)>에 보인 것처럼 5개의 SDF actor들 (IDCT, IQUANT, Reconstruct, AddBlockInter, AddBlockIntra)의 HW/SW 매핑을 바꿔가며 실험했다. <표 1(a)>는 해당하는 SDF actor들의 크기, 수행 시간 및 구성 시간을 보여준다. 나머지 SDF actor들은 모두 SW로, 프로세서 위에 매핑되었다.

<표 1(c)>는 본 논문에서 제시한 구성 스케줄링 방법으로 얻은 decoder 수행 시간 (our method)과 configuration prefetch를 전혀 하지 않는 decoder의 수행 시간 (no prefetch), 제시한 구성 스케줄링 방법의 오버헤드 (OV_pre : prefetch 코드의 오버헤드, OV_sched : 구성 스케줄러의 오버헤드, OV_total : 전체 오버헤드)를 보여준다. Configuration prefetch를 하지 않은 경우는 각 FPGA에 매핑된 actor의 수행이 실제로 필요할 때에만 구성을 시작한다는 것을 의미한다. 본 논문에서 제시된 방법은 평균 약 12.7%, 최대 21.78%의 성능 향상을 얻었다. 그러나 <표 1(b)>에 보인 6가지

표 1. MPEG4 구성 스케줄링 실험 결과
Table 1. MPEG4 configuration scheduling results.

SDF actors	Gate counts	CLB counts	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) HW에 매핑된 actor의 HW 구현 결과

Mapping	IDCT	IQUANT	Reconstruct	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 매핑 결과

Mapping	Our method	No prefetch	Gain (%)	OV_pre (%)	OV_sched (%)	OV_total (%)
A	46,740,471	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) 수행시간(cycle)과 수행시간 이득 및 오버헤드

Mapping	Original (bytes)	OV (%)
A	41,952	0.88
B	41,920	0.88
C	41,548	0.89
D	40,054	0.92
E	40,086	0.92
F	39,682	0.94

(d) 메모리 오버헤드

매핑 중 매핑 B의 경우, 오히려 1.4% 가량 성능이 나빠지는 결과를 얻었다. 매핑 B에서는 IDDT, PIQUANT, AddBlockInter actor가 HW로 매핑되었다. 이 중, IDCT와 AddBlockInter는 FPGA에서 동시에 수행될 수 있다. 그리고 IDCT와 IQUANT도 역시 FPGA에서 동시에 수행될 수 있다. 그러나 AddBlockInter와 IQUANT는 FPGA에서 동시에 수행될 수 없으며, decoder의 수행 중에 이러한 conflict는 단 한번 발생한다. 따라서 매핑 B의 경우, 각 actor들의 구성 다운로드는 오직 한번이면 충분하며, 일단 FPGA에 다운로드된 이후에는 재구성할 필요가 없다. 이 경우, 구성 스케줄러는 시스템의 수행 시간을 효과적으로 줄일 수 없으며, 오히려 불필요한 오버헤드만을 주게 된다. 그러나, 이러한 오버헤드는 실험 결과에서 보듯이 무시할 만한 양이다.

b) IS95 실험 결과

IS95 modem중 mobile station에서 base station으로 데이터를 전송하는 reverse link 부분을 HFSM-SDF로 기술하고, 손으로 해당하는 reverse link modem을 구현했다. SW만으로 구현된 reverse link modem은 총 396,036,715 cycle이 소요된다. HW 구현과 비교해보면 성능이 매우 좋지 않은데, 이것은 modem 부분은 SW로 구현하기에 적합하지 않은 bit 수준의 연산과 bit 단위의 메모리 접근이 매우 빈번하기 때문이다. 그리고 구성 스케줄링을 적용하기 위해서 <표 2(a)>에 보인 SDF actor들의 HW/SW 매핑을 <표 2(b)>와 같이 4가지로 바꿔가며 실험했다. <표 2(c)>는 구성 스케줄링을 적용해서 얻은 수행 시간과 configuration prefetch를 전혀 적용하지 않았을 때의 수행 시간을 보여준다. 본 논문의 구성 스케줄링 방법을 적용함으로써 평균 7.3%, 최대 9.7%의 수행 시간 향상을 얻었으며, 실시간 스케줄링과 관련된 수행 시간 및 메모리 사용의 오버헤드는 각각 1% 미만, 10% 미만이다. Reverse link modem에서 MPEG4와 비교해 상대적으로 적은 수행 시간 향상을 얻은 이유는, reverse link modem의 경우, 구성 시간이 전 메모리 사용의 오버헤드는 단순히

표 2. IS95 구성 스케줄링 실험 결과
Table 2. IS95 configuration scheduling results.

SDF actors	Gate counts	CLB counts	Exec delay	Cfg latency
Repeat	5022	5	633	12480
Block	15276	15	576	37440
Walsh	11891	12	6758	29952
PNgen	13486	13	27033	32448
IQgen	31486	30	30800	74880

(a) HW에 매핑된 actor의 HW 구현 결과

Mapping	Repeat	Block	Walsh	PNgen	IQgen
A	HW	HW	SW	SW	HW
B	HW	HW	SW	HW	SW
C	HW	HW	SW	HW	HW
D	HW	HW	HW	HW	SW

(b) HW/SW 매핑 결과

Mapping	Our method	No prefetch	Gain (%)	OV_pre (%)	OV_sched (%)	OV_total (%)
A	1,844,156	2,009,457	8.23	0.06	0.39	0.45
B	1,715,591	1,899,057	9.66	0.08	0.56	0.64
C	1,641,003	1,778,591	7.74	0.08	0.59	0.67
D	1,857,758	1,928,303	3.66	0.09	0.64	0.73

(c) 수행시간(cycle)과 수행시간 이득 및 오버헤드

Mapping	Original (bytes)	OV (%)
A	11384	6.92
B	10380	8.24
C	11880	8.01
D	10864	8.80

(d) 메모리 오버헤드

modem 예제의 크기가 작기 때문이며, 이러한 오버헤드는 시스템이 복잡해져도 크게 달라지지 않는다.

Ⅶ. 결 론

본 논문에서는 hierarchical FSM과 synchronous dataflow (HFSM-SDF) 모델을 이용한 재구성 시스템의 설계와 관련해서 실시간 구성 스케줄링 방법을 제시했다. 최상위 FSM의 각 transition에 대해 각 hierarchical FSM의 현재 state를 파악한 후, 이로부터 현재 수행해야 하는 구성 스케줄을 ready CQ에 저장한다. 그리고 이 queue를 기반으로 실시간 구성 스케줄러가 현재 최상위 FSM의 수행 시에 필요한 각 구성을 가능한 한 빨리 FPGA에 다운로드한다. 본 논문에서는 제시한 방법을 MPEG4 natural video decoder와 IS95 modem의 reverse link 예제에 적용해보았으며, 그 결과, 최대 21.8%의 수행 시간 향상을 얻었다. 그리고 실시간 구성에 의한 수행 시간 및 메모리 사용의 오버헤드는 무시할 수 있는 양임을 보였다.

참 고 문 헌

[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] 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://www.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 Approach to Hiding Configuration Latency in Chameleon Processors," 10th Int'l Conference 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)," avaiable 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 Simualation 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 Communication Organized for Reconfigurable Execution(SCORE)," Int'l Conf. on Field Programmable Logic and Applications, Aug. 2000.

[16] K. M. Gajjala Purna and D. Bhadia, "Temporal Partitioning and Scheduling Data Flow 그래프s 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, Berkely, May 2000.

[21] B. Salefski and L. Caglar, "Re-Configurable Computing in Wireless," Proc. Design Automation Conference, pp. 18~22, June 2001.

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

[23] TIA/EIA-95A, "*Mobile Station-Based Compatibility Standard for Dual-Mode Wideband Spread Spectrum Cellular Systems*," 1995.

[24] Qualcomm, Inc., "CDMA System Engineering Training Book," *1993*.

[25] S. Lee, S. Yoo, and K. Choi, "Reconfigurable SoC Design with hierarchical FSM and synchronous dataflow model," Proc. Int'l Symp. on Hardware/Software Codesign, pp. 199~204, May 2002.

[26] ARM Ltd., "ARM7TDMI Technical Reference Manual," available at http://www.arm.com/arm/TRMs?OpenDocument.

[27] *ARM Ltd.*, "*ARM Software Development* Toolkit v. 2.51," available at http://www.arm.com/devtools/SDT.

[28] Xilinx, Inc., "Virtex-ETM 1.8V Field Programmable Gate Arrays," available at http://direct.xilinx.com/bvdocs/publications/ds022.pdf.

[29] Xilinx, Inc., "ISE Foundation," available at http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=ISE+Foundation.

[30] Xilinx, Inc., "Virtex Series Configuration Architecture User Guide," available at http://www.xilinx.com/xapp/xapp151.pdf.

저 자 소 개

李 成 賢(正會員)

1998년 : 서울대학교 전기공학 학사. 2000년 : 서울대학교 전기공학 석사. 2000년 3월~현재 : 서울대학교 전기 컴퓨터 공학부 박사과정 재학 중

崔 起 榮(正會員)

1978년 : 서울대학교 전자공학 학사. 1980년 : KAIST 전기전자공학 석사. 1989년 : Stanford Electrical Engineering 박사. 1980년~1983년 : 금성사 근무. 1989년~1991년 : Cadence 근무. 1991년~현재 : 서울대학교 전기컴퓨터공학부 교수

劉 承 周(正會員)

1992년 : 서울대학교 전자공학 학사. 1995년 : 서울대학교 전자공학 석사. 2000년 : 서울대학교 전기공학 박사. 2000년 4월~현재 : 프랑스 TIMA 연구소 SLS 그룹에서 근무중