데이터 플로우 모델로부터 합성 가능한 하드웨어-소프트웨어 인터페이스의 자동 생성

주영표, 양회석 ^O^, 하순회

서울대학교 전기컴퓨터공학부

{youngpyo, hyang, sha}@iris.snu.ac.kr

Automatic Generation of Synthesizable Hardware-Software Interface from Dataflow Model

Young-Pyo Joo, Hoeseok Yang^O^, Soonhoi Ha

EECS, Seoul National University

요약

컴퓨터 시스템의 설계는 알고리즘 수준의 모델링에서부터 시제품 수준까지 시스템을 구체화해 나가는 일련의 과정이다. 시스템 구현의 구체화 과정에는 단순하고 반복적인 구현이 많이 포함되며, 이 과정에서 많은 오류가 발생한다. 이러한 오류는 개발자가 알고리즘 수준에서는 드러나지 않는 복잡하고 아키텍처 의존적인 하드웨어-소프트웨어 동기화 메커니즘의 개발과 같은 시스템 구현의 구체화 과정을 모두 떠안고 있기 때문에 발생하는 것이다. 이 논문에서는, 이러한 문제를 극복하기 위하여, 알고리즘을 데이터 플로우로 모델링하면 이로부터 합성 가능한 하드웨어 플랫폼과 동기화 로직, 그리고 동기화를 위한 드라이버 소프트웨어 일체를 자동 생성하는 설계 과정을 제시하고자 한다. 제시된 설계 과정은 자체 개발한 통합 설계 도구 상에 구현되었으며, 이를 통해서 개발된 H.263 디코더 예제를 상용의 RTL 통합 시뮬레이션 도구인 Seamless CVE와, SoC 프로토타이핑 환경인 Altera Excalibur 시스템 상에서 테스트하여 그 완성도를 검증하였다.

1. 서론

시스템 설계 공간 탐색에 대한 많은 연구들은 그림 1의 Y-차트 접근법[1]을 바탕으로 하고 있다. 이 접근법에서는 시스템 아키텍처를 모델링 하고, 그 위에 알고리즘 수준으로 설계된 어플리케이션을 매핑한다. 이렇게 매핑한 결과는 실제화된 시스템으로 생성되어 그 성능이 평가되고, 이를 토대로 주어진 어플리케이션에 최적화된 아키텍처를 찾아내게 된다. Y-차트 접근법은 이와 같이 최적의 아키텍처를 찾기 위하여 아키텍처 생성, 매핑, 성능 평가 과정을 반복하기 때문에, 그림에서 'Simulator'에 해당하는 성능 평가 기법의 속도 향상이 주요 열쇠가 된다. 하여 최근의 연구들은 이 과정에 RTL보다 높은 수준의 빠른 성능 평가 기법을 도입하고 있다.

하지만 최적의 아키텍처 후보군이 압축되고 나면 상황은 달라진다. 후보 간의 성능 차가 앞서의 성능 평가 기법의 오차 범위 내에 들 가능성이 높고, 생성한 아키텍처에 대한 보다 정밀한 성능 평가와 오류 검증이 요구되기 때문에 앞서의 성능 평가 기법을 계속 활용할 수 없다. 그렇기 때문에 이 과정에서는 하드웨어-소프트웨어 통합 시뮬레이션이 널리 활용된다. 이는 대개 사이클-정밀도를 갖는 마이크로프로세서 시뮬레이터와 하드웨어를 위한 RTL 시뮬레이터로 구성되며, 전통적인 RTL 시뮬레이션이나 실제 프로토타이핑에 비하여 상대적으로 빠르게 구축하고, 검증할 수 있다는 장점을 갖고 있다. 최근의 상용 하드웨어-소프트웨어 통합 시뮬레이션 환경들은 개발자의 편의를 위하여 다양한 아키텍처의 컴포넌트 모델을 제공하는 프론트-엔드 툴들을 포함하고 있으며, 여기에는 아키텍처의 모델링 및 템플릿 제공, 그리고 아키텍처 자동 생성이 포함되기도 한다.

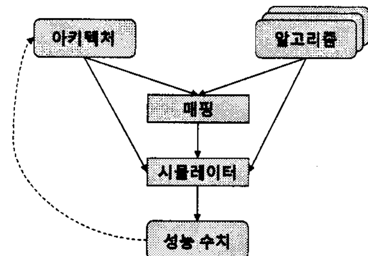


그림 1 설계 공간 탐색을 위한 Y-차트 접근법

위와 같은 시스템 설계 및 설계 공간 탐색 과정 중에서, 알고리즘의 아키텍처 매핑 이후 시스템의 구체화 단계는 그 중요성에 비하여 상대적으로 상용 도구들의 지원이 부족한 부분이다. 알고리즘의 아키텍처 매핑이 끝나고 나면, 우선 사실적인 성능 검증을 위해서 구동이 가능한 수준까지 알고리즘과 아키텍처를 구체적으로 구현하는 작업이 필요하다. 기존의 설계 도구들이 알고리즘 모델링 도구와 아키텍처 모델링 도구들로 나뉘어져 있기 때문에, 그 접점이 되는 이 과정은 대개 개발자의 수작업을 통해 이루어지고 있다. 하지만, 이 과정은 많은 시간을 요구하는 반복 작업이 대부분이며, 발견하기 힘든 에러가 발생할 가능성이 매우 높기 때문에 빠른 설계 공간 탐색에 큰 걸림돌이 되곤 한다.

이 논문에서는 이러한 문제를 극복하기 위하여 시스템의 구체화 과정을 자동화하는 기법을 제시한다. 제시된 기법은 데이터플로우로 설계된 어플리케이션의 시스템 아키텍처 매핑 결과에서 데이터를 추출하여, 데이터 경로와 컨트롤 경로를 자동 생성한다. 자동 생성한 경로들은 시스템 아키텍처와 함께 합성되며, 여기에는 소프트웨어 드라이버도 포함되기 때문에 시스템은 바로 구동이 가능한 수준까지 구체화될 수 있다.

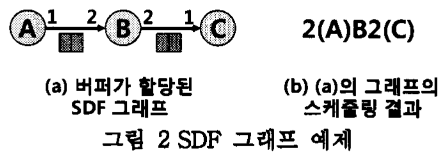
2. 관련 연구

Platform Express™[2]는 Mentor Graphics®의 상용 툴로, 동사의 하드웨어-소프트웨어 통합 시뮬레이션 툴인 Seamless CVE™[3]를 위한 시뮬레이션 모델 생성을 지원하는 GUI 기반의 프론트-엔드 툴이다. 이 툴은 아키텍처 모델링과 생성을 담당하기 때문에, 앞서 설명했던 상용 도구들과 마찬가지로, 모든 시스템 구체화 과정은 물론 시뮬레이션 모델의 디자인도 모두 개발자의 몫이다.

보다 이론적인 하드웨어-소프트웨어 인터페이스의 자동 합성에 대한 연구들은 [4]에서 확인할 수 있다. 이들 연구들은 공통적으로 [5]에서 제안하는 '인터페이스 기반 설계'라는 개념을 따르고 있다. 이는 블록 기반으로 설계한 어플리케이션을 블록 내부 동작부와 블록 간의 통신부로 나누어 설계를 구체화 할 수 있다는 개념이다. 몇몇 연구에서는 IP들 간의 프로토콜을 자동으로 변환하여 일치시킬 수 있도록 하는 통신 프로토콜의 정형적인 표현에 대한 연구가 진행되어 왔다[6][7].

조금 다른 접근법으로, 파라미터화 된 인터페이스 템플릿을 사용하는 방식이 있다[8]. 본 논문은 바로 이 범주에 들어가게 되는데, 이 논문에서 제시하는 설계 환경에서는 알고리즘 블록의 아키텍처 매핑을 위하여 파라미터화 된 send/receive 블록이 사전 정의되어 있고 이들은 파라미터화 되어 있다는 점이 유사하다. 보다 일반화된 최근의 연구로는 소프트웨어 인터페이스 코드의 합성을 포함하는 연구가 있으며[9], 본 논문은 이에 대해서도 다루고 있다.

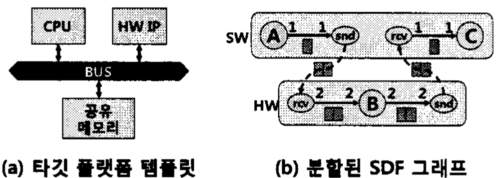
3. 제안하는 디자인 플로우



(a) 버퍼가 할당된 SDF 그래프　　(b) (a)의 그래프의 스케줄링 결과

그림 2 SDF 그래프 예제

알고리즘의 명세를 위해서, 본 연구에서는 그림 2 (a)와 같은 SDF(Synchronous Dataflow) 그래프[10]를 사용하고 있다. 알고리즘의 아키텍처 매핑의 편의를 위하여, 그래프는 블록 단위로 아키텍처 매핑이 이루어진다고 가정한다. 데이터플로우 그래프에서 블록 간 연결은 데이터 샘플들의 스트림을 포함하는 채널로 표현되는데, SDF는 샘플이 머무는 전역 버퍼가 정수 크기로 고정된다. 여기서 버퍼의 크기는 블록의 한 번 수행 시 생산 및 소비되는 샘플의 개수를 토대로 계산할 수 있다. 예를 들어, 그림 2 (a)의 A는 한 번 수행 시 하나의 데이터 샘플을 생성하고, B는 한 번 수행 시 두 개의 데이터 샘플을 소비하기 때문에 둘 사이의 전역 버퍼 크기는 2가 되며, 스케줄은 A가 두 번 수행될 때 B가 한 번 수행되게 된다. 이를 토대로 하여 얻을 수 있는 그래프의 스케줄링 결과는 그림 2 (b)와 같다.



(a) 타깃 플랫폼 템플릿　　(b) 분할된 SDF 그래프

그림 3 타깃 플랫폼 템플릿과 해당 플랫폼에 분할 매핑된 그래프

Y-차트 접근법에서는 알고리즘의 모델링 외에도 아키텍처 모델링이 요구된다. 본 연구에서는 그림 3 (a)와 같은 단순한 아키텍처 템플릿을 모델링 하였다. 알고리즘이 아키텍처 매핑 과정을 거치면서 블록 A와 C는 CPU에, 블록 B는 HW IP에 분할되었다고 가정할 때, 분할된 SDF 그래프는 그림 3 (b)와 같다. 분할된 SDF는 [5]에서 제안된 바와 같이 블록 A~C의 내부 구현은 변하지 않으며, 외부 인터페이스 구현만이 분할 결과에 맞추어 바뀌게 된다. 다른 프로세서로 분할된

블록들 사이에 부가적인 send/receive 블록이 추가되고, 이로 인하여 추가적인 전역 버퍼가 할당된 것을 그림 3 (b)에서 확인할 수 있다. 타깃 플랫폼 템플릿, send/receive 블록, 전역 버퍼, 그리고 데이터 경로와 컨트롤 경로의 자동 생성은 4장에서 소개된다.

4. 하드웨어-소프트웨어 플랫폼의 자동 생성

자동 생성되는 플랫폼의 형태는 아래의 그림과 같다. 여기서 send/receive 블록과 전역 버퍼에 대한 정보는 분할된 SDF 그래프로부터의 코드 생성 과정에서 얻을 수 있는 것으로, 표 1과 같다. 이를 바탕으로 한 생성 과정은 4.1에서 소개된다. 하드웨어의 경우에는 send/receive 블록이 명시적으로 표현되어 있지만, 소프트웨어는 소프트웨어 인터페이스로써 API 형식으로 작성되어 있다.

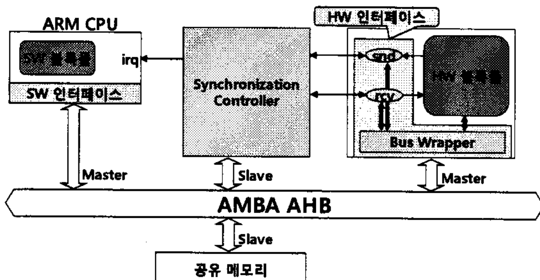


그림 4 생성되는 타깃 플랫폼 템플릿

데이터 경로의 자동 생성은 4.2에서 소개되는데, 하드웨어의 데이터 경로가 자동 생성된 Bus Wrapper 및 send/receive 블록들과 버스 사이의 중계를 의미한다면, 소프트웨어는 공유 메모리의 접근이 CPU의 메모리 인터페이스를 통하여 이루어지므로, 별도의 로직이나 구현이 필요치 않다.

표 1 분할된 SDF 그래프로부터의 코드 생성과정에서 얻어지는 정보

Send	SndPtr	데이터 전송 포인터와 그 현재 위치
	SndSize	한 번의 전송에서 쓰는 데이터의 크기
	SndIntrPtr	Synchronization Controller 상의 인터럽트 레지스터 주소 (소프트웨어로 매핑이 결정된 경우에만 해당)
Receive	RcvPtr	데이터 수신 포인터와 그 현재 위치
	RcvSize	한 번의 수신에서 읽는 데이터의 크기
	RcvIntrPtr	Synchronization Controller 상의 인터럽트 레지스터 주소 (소프트웨어로 매핑이 결정된 경우에만 해당)
Buffer	ID	해당 채널의 ID
	BufferStart	버퍼의 시작 주소
	BufferSize	버퍼의 전체 크기
	InitDelay	버퍼 상에 초기 데이터가 존재하는 경우 그 크기

컨트롤 경로는 Synchronization Controller 위주로 구성되어 있으며, 4.3에서 다루어진다. 하드웨어가 해당 컨트롤러와 wire 수준의 직접적인 연결을 통해 버퍼의 상태를 파악한다면, CPU에 매핑된 블록들은 인터럽트 기반의 드라이버를 통해 그 기능을 수행한다.

마지막으로, 위 그림의 전체적인 윤곽은 타깃 플랫폼의 템플릿을 의미하며, 이 모든 것을 한데 묶어 생성하는 과정이 4.4에서 소개된다.

위의 표에서 *SndSize/RcvSize*, *BufferSize*, *InitDelay*는 분할된 SDF 그래프에서 추가적인 노력 없이 기본적으로 얻을 수 있는 정보이다. 예를 들어 그림 3 (b)에서 A와 B 사이의 send/receive의 경우 *SndSize*는 *{데이터 단위 샘플 크기}x1*, *RcvSize*는 *{데이터 단위 샘플 크기}x2*이며, *BufferSize*는 *RcvSize*와 같다. 이 경우 *InitDelay*는 0이다. *InitDelay*가 존재하는 예제는 뒤의 실험에서 볼 수 있다.

ID, *SndIntrPtr/RcvIntrPtr*, *BufferStart*는 그래프로부터 코드를 생성하는 과정에서 얻을 수 있는 데이터로, 이 논문에서는 위의 값들이 아래와 같이 일련의 배열 형태로 배치되는 것을 가정하였다.

ID[i] = i

SndIntrPtr[i] = 인터럽트기본주소 + i

RcvIntrPtr : SndIntrPtr과 하나의배열로취급됨

$$\text{BufferStart}[i] = \text{버퍼기본주소} + \sum_{k=0}^{i-1} BufferSize[k]$$

4.1 Send/Receive 블록과 전역 버퍼의 자동 생성

분할된 그래프에서 기능 블록들의 내부가 알고리즘의 수행을 의미하는데 비하여, send/receive 블록들은 그 내부 동작 자체가 데이터의 전달을 목표로 하고 있다. 모든 전역버퍼는 각각에 접근하는 send/receive 블록이 유일하므로 각 블록이 각 버퍼에 대하여 하나의 읽거나 쓰는 포인터를 유지할 수 있다. 여기서는 소프트웨어로 매핑된 send 블록과 하드웨어로 매핑된 receive 블록의 pseudo 코드를 예로 보이기로 한다. 이 때 필요한 데이터는 앞서 설명하였듯이 분할된 SDF 그래프로부터 표 1과 같은 정보들을 얻어 활용한다.

```
소프트웨어로매핑된send블록:
1) If 초기화요청
2)  SndPtr = BufferStart + InitDelay
3) End if
4) Wait for SndIntrPtr 인터럽트
5) Write data to SndPtr (size : SndSize)
6) SndPtr + = SndSize
7) Clear SndIntrPtr 인터럽트
```

하드웨어로매핑된receive블록:
1) If 초기화요청
2) RcvPtr= BufferStart
3) End if
4) Wait for RcvIntrPtr시그널
5) Read data from RcvPtr(size : RcvSize)
6) RcvPtr+ = RcvSize
7) Clear RcvIntrPtr시그널

소프트웨어 쪽 send/receive API는 위의 표가 헤더 파일 형태로 선언되기 때문에 이를 참조로 하여 데이터를 주고 받고, 하드웨어 쪽 send/receive 블록은 각자 접근해야 하는 버퍼의 주소가 사전에 결정되어 생성된다.

위와 같은 send/receive 블록의 자동 생성과 함께, 버퍼 기존 주소부터 $\sum_{k=0}^{i-1} BufferSize[k]$ 만큼의 공유 메모리도 생성한다.

4.2 데이터 경로의 자동 생성

하드웨어로 매핑된 알고리즘 블록의 send/receive 블록은 그림 4의 Bus Wrapper를 통하여 데이터를 주고 받는다. Bus Wrapper는 send/receive의 단순한 메모리 접근 프로토콜을 타깃 아키텍처에 맞게 변환하는 역할을 맡고 있으며 파라미터화 된 라이브러리 형태를 갖는다. 소프트웨어의 데이터 경로의 경우 부가적인 설정이 필요치는 않지만, 버퍼 영역에 대한 캐시를 끄는 설정이 필요하다. 이 역시 파라미터화 된 라이브러리 코드를 통하여 동작한다.

4.3 컨트롤 경로의 자동 생성

Synchronization Controller는 별도의 하드웨어 로직이다. 그림 4에서 확인할 수 있듯이, 하드웨어 send/receive 블록들이 CPU의 인터럽트에 직접 연결되는 구조를 택하지 않은 이유는 인터럽트 번호 자원의 남용을 막고, 인터럽트 핸들러의 수행 시간을 일정하게 보장하기 위해서이다. 시스템 상에 널리 흩어져 있는 각각의 send/receive가 인터럽트를 호출한다면, 인터럽트 핸들러가 이들을 접근하여 처리하는 과정에서 접근 경로가 달라짐에 따라 수행 시간이 일정치 않을 수 있기 때문이다. Synchronization Controller의 pseudo 코드는 아래와 같다.

Synchronization Controller :
1) If 초기화요청
2) For 모든send/receive
3) dataSize = InitDelay
4) End for
5) End if
6) For 모든send
7) If send데이터전송완료시그널이올라감
8) dataSize + = SndSize
9) End if
10) If BufferSize - dataSize < SndSize
11) send데이터전송가능시그널을내림
12) Else
13) send데이터전송가능시그널을올림
14) End if
15) End for
16) For 모든receive
17) If receive데이터수신완료시그널이올라감
18) dataSize - = RcvSize
19) End if
20) If dataSize >= RcvSize
21) receive데이터수신가능시그널을올림
22) Else
23) receive데이터수신가능시그널을내림
24) End if
25) End for

위의 코드에서 시그널을 올리거나 내리는 것은 CPU에 매핑된 소프트웨어 send/receive에게는 인터럽트 발생 여부를, 하드웨어 send/receive에게는 wire 시그널을 제어하는 것을 의미한다.

4.4 전체 플랫폼의 자동 생성

프로토타이핑 환경은 시스템의 구성에 맞게 CPU나 메모리를 생성하지 않아야 하기 때문에 여기서는 통합 시뮬레이션을 위한 모델을 생성하기는 것을 예로 들고자 한다. 시뮬레이션 모델은 Seamless CVE에서 구동 가능한 형태이며, 생성되는 컴포넌트는 다음과 같다.

1) ARM926ej-s 프로세서 모델
2) SRAM 메모리 모델
3) AMBA AHB 버스 모델
4) 하드웨어로 매핑된 블록과 send/receive 블록
5) 소프트웨어로 매핑된 블록과 send/receive API
6) Bus Wrapper와 Synchronization Controller
7) 위의 컴포넌트들을 한 데 묶는 시뮬레이션 모델
8) 소프트웨어와 하드웨어 시스템을 각기 빌드하고 시뮬레이터를 구동하는 스크립트

위의 컴포넌트들을 모두 생성하면 그림 4와 같은 결과물을 얻을 수 있으며, 이를 빌드 및 구동하는 스크립트까지 함께 생성된다. 하드웨어의 경우 합성하는 스크립트는 프로토타이핑 과정에서만 필요하므로, 시뮬레이션을 위해서는 타깃 시뮬레이터를 위한 컴파일러를 구동하는 것으로 충분하다. 소프트웨어의

경우, 통합 시뮬레이션 환경에서 OS를 구동하는 것은 시간적인 부담이 되므로, OS가 없는 환경에서 구동할 수 있는 드라이버와 스케줄러를 자동 생성한다. 이 스케줄러는 비선점형 스케줄러이기 때문에 복잡한 알고리즘의 구동에서는 데드락이 발생할 가능성을 완전히 배제할 수 없다. 소프트웨어의 경우에는 합성과 컴파일의 구분이 없으므로 스크립트는 프로토타이핑 환경과 동일하게 생성된다.

5. 실험

실험에 사용된 예제는 H.263 디코더이다. 실험을 위하여 분할 및 매핑은 아래 그림의 Motion Compensation 블록을 하드웨어로, 나머지 모든 블록을 소프트웨어로 매핑하였다. 그림에서 가장 아래 쪽의 세 채널에는 중간에 작은 블록이 포함되어 있는데, 이는 앞서 언급하였던 Initial Delay를 의미한다. 초기값의 삽입은 모든 어플리케이션의 구동 이전에 수행되므로, 어플리케이션의 구동 중에는 아무런 영향을 미치지 않는다. 이 경우 앞서의 InitDelay값을 설정하는 것 외에 별다른 동작은 필요치 않다. 자동 생성되어야 하는 하드웨어-소프트웨어 간 채널은 모두 10개이며, 10개의 채널들이 사용하는 전역 버퍼의 크기는 아래의 표 2와 같다.

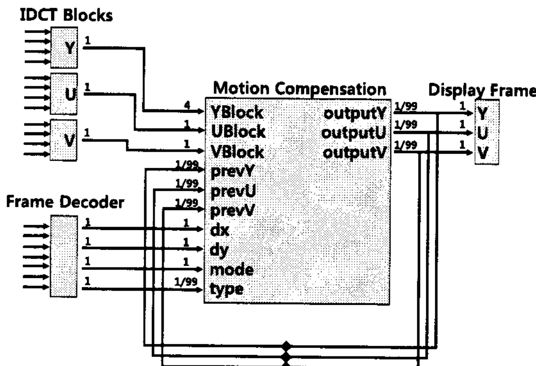


그림 5 H.263 디코더의 서브 그래프

실험은 SeamlessCVE와 Altera Excalibur 시스템 상에서 이루어졌다. Seamless CVE가 사용하는 프로세서 시뮬레이터로는 ARM®의 ARMulator™를 사용하였고, RTL 시뮬레이터로는 Mentor Graphics®의 ModelSim™을 사용하였다. 이 때, 아키텍처 템플릿으로는 ARM926ej-s, AMBA AHB 버스, synchronous SRAM 모델이 사용되었다. Aletra®의 Excalibur™ 시스템은 Huins의 SoCMaster 프로토타이핑 보드를 사용하였으며, Excalibur의 내부는 아래 그림과 같다. 공유 메모리는 80MHz로 구동되는 2차 AHB 버스 상의 SRAM을 사용하였으며, OS 환경은 리눅스 2.4.18 커널을 사용하였다.

표 2 전역 버퍼의 크기

채널명	기본 데이터 샘플 크기 (Bytes)	버퍼 크기 (Bytes)
YBlock	64	256
UBlock	64	64
VBlock	64	64
dx	4	4
dy	4	4
mode	4	4
type	4	4
outputY	50688	50688
outputU	12672	12672
outputV	12672	12672

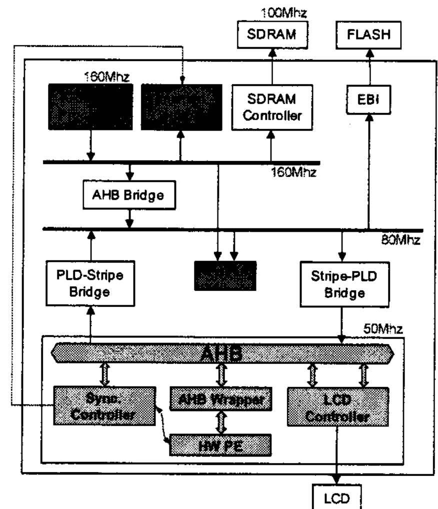


그림 6 Altera Excalibur 프로토타이핑 시스템

5.1 수작업 디자인과의 비교

표 3 드라이버 구현 패턴에 따른 수행 성능 비교

디바이스 드라이버의 구현 스타일	MC 수행 시간 (usec)
매 접근 때마다 read/write 시스템 콜 사용 send/receive : 로컬 메모리에서 공유 메모리로 데이터 복사	9,080,000
매 접근 때마다 mmap 사용 send/receive : 로컬 메모리에서 공유 메모리로 데이터 복사	6,060,000
전체 공유 메모리 영역에 대하여 mmap 한 차례 수행 send/receive : 로컬 메모리에서 공유 메모리로 데이터 복사	1,700,000
전체 공유 메모리 영역에 대하여 mmap 한 차례 수행 send/receive : 공유 메모리에 직접 데이터 기록	1,590,000
전체 공유 메모리 영역에 대하여 mmap 한 차례 수행 send/receive : 공유 메모리에 직접 데이터 기록 디바이스 드라이버에서 데이터 복제 안함	301,000
위와 동일한 구현에, 개발자에 의한 설계	296,000
MC 알고리즘 블록의 수행 시간	256,212

실험의 편의를 위하여 QCIF 크기(176x144)의 작은 동영상 샘플을 사용하였다. 아래 표의 성능 정보는 Altera Excalibur 시스템에서 구동하여 얻은 것이다. 표 3에서 확인할 수 있듯이, 자동 생성된 것 중 가장 빠른

것과 개발자에 의해 이루어진 디자인의 성능 차이가 1.6% 정도로 매우 작은 편이다. 이는 자동화에 사용된 파라미터화 된 디자인 템플릿의 성능이 우수함을 의미한다. 더불어, 디바이스 드라이버의 구현에 따라 성능이 많은 차이를 보임을 알 수 있는데, 이는 충분한 고려와 튜닝을 거쳐 만들어진 경우에만 자동 생성된 코드가 경쟁력이 있음을 의미한다.

5.2 다양한 매핑 결정

구동 가능한 시스템을 자동 생성할 수 있다는 것은 다양한 설계 공간 탐색을 손쉽게 수행할 수 있다는 것을 의미한다. 여기서는 이점을 살려, 앞서의 H.263 디코더 예제에 대하여 1) MC의 하드웨어 매핑, 2) IDCT의 하드웨어 매핑, 3) MC와 IDCT 모두의 하드웨어에 매핑을 수행하였다. 각각의 매핑을 수행하는 데에는 Pentium 4 환경에서 수 초 이내의 시간이 필요하였다.

세 가지 다른 하드웨어 매핑에 대하여, Altera Excalibur를 대상으로 하드웨어를 합성한 결과, 아래와 같은 수치를 얻을 수 있었다. 표에서 확인할 수 있듯이, Synchronization Controller와 Bus Wrapper의 하드웨어 크기는 전체 하드웨어의 크기가 커질수록 그 비중이 작아짐을 확인할 수 있다. 이는 예제의 크기가 커진다고 해서 자동 생성되는 로직의 크기가 급격히 늘어나지 않음을 보여준다.

표 4 다양한 설계 공간 탐색 결과

RTL 블록	MC (LUTs)	IDCT (LUTs)	MC + IDCT (LUTs)
Synchronization Controller	104	145	166
Bus Wrapper	761	1035	1492
알고리즘 블록	4398	21102	28890
플랫폼 전체	4748	22021	29974

[Note]
1. Synplify Pro 7.7.1로 Altera Excalibur EPXA10F1020C2를 타깃으로 하여 합성한 결과임
2. EPXA10F1020C2의 총 LUT 크기는 38400임

6. 결론

이 논문에서는 데이터플로우 모델로부터 합성 가능한 하드웨어-소프트웨어 SoC 플랫폼을 자동 생성하는 플로우를 제시하였다. 플랫폼의 생성을 위해서는 분할된 SDF 그래프가 제공하는 send/receive 블록 및 전역 버퍼에 대한 정보와 부가적인 코드 생성 관련 정보가 필요하다. 생성된 플랫폼은 마이크로프로세서, 하드웨어 블록, 메모리, 버스로 구성되는 기본 플랫폼에, SDF 그래프로 기술된 어플리케이션의 구동을 위한 별도의 데이터 경로와 컨트롤 경로가 포함된다. 제안한 기법을 통해 설계한 H.263 디코더 예제가 Seamless CVE와 Altera Excalibur 프로토타이핑 시스템에서 성공적으로 구동되는 것을 통하여 기법의 정확성을 검증하였다.

또한 추가적인 실험을 통하여, 본 기법을 사용할 경우, 개발자가 많은 노력을 들여 만든 결과물과 비슷한 성능을 갖는 플랫폼을 손쉽게 자동 생성할 수 있음을 확인하였다. 마지막으로 손쉽게 구체화된 어플리케이션을 생성할 수 있다는 이점을 살려 다양한 설계 공간 탐색이 가능함을 확인하였다.

감사의 글

본 연구는 BK21 프로젝트, 과학기술부 도약연구지원사업(R17-2007-086-01001-0)에 의해 지원되었다. 또한 서울대학교 컴퓨터신기술연구소와 IDEC은 본 연구에 필요한 기자재들을 지원해 주었다. 본 연구는 또한 한국전자통신연구원의 SoC 핵심설계인력양성사업에 의해 부분적으로 지원되었다.

참고 문헌

[1] B. Kienhuis et. Al. "An approach for quantitative analysis of application specific dataflow architectures," Proc. Intl. Conf. Application-Specific Systems, Architectures and Processors, pp. 14-16, Jul 1997.
[2] Mentor Graphics, "Platform Express Professional," [Online document], Available HTTP: http://www.mentor.com
[3] Mentor Graphics, "Seamless Co-Verification," [Online document], Available HTTP: http://www.mentor.com
[4] B. A. Rajawat, M. Balakrishnan, and A. Kumar, "Interface synthesis: Issues and Approaches." Proc. Intl. Conf. on VLSI Design, pp. 92-97, 2000.
[5] James A. Rowson and Alberto Sangiovanni-Vincentelli. "Interface-based design," Proc. DAC, Jun 1997.
[6] V. D'silva, S. Ramesh, Arcot Sowmya, "Bridge Over Troubled Wrappers: Automated Interface Synthesis," Proc. VLSI Design, 2004.
[7] R. Passerone, J. A. Rowson, and A. Sangiovanni-Vincentelli, "Automatic Synthesis of Interfaces between Incompatible Protocols," Proc. DAC, pp. 8-13, 1998.
[8] J. Smith and G. D. Micheli, "Automated composition of hardware components," Proc. DAC, 1998.
[9] Adriano Sarmento, et. Al. "Service dependency graph: an efficient model for hardware/software interfaces modeling and generation for SoC design," Proc. CODES+ISSS 2005, pp. 261-266, Sept 2005.
[10] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," IEEE Proceedings, Sep 1987.