Journal of Korea Multimedia Society Vol. 10, No. 11, November 2007(pp. 1446-1459)

하드웨어 및 소프트웨어 모듈간의 동적 협업을 지원하는 SoC 플랫폼 설계에 관한 연구

이동건[†], 김영만[††], 탁성우[†††]

요 약

본 논문에서는 소형 임베디드 시스템의 처리 성능 향상을 위하여 하드웨어 및 소프트웨어 모듈간의 동적 협업 SoC 플랫폼을 제안하고 성능을 분석하였다. 기존의 소형 임베디드 시스템은 낮은 사양의 하드웨어 자원을 가지고 있어 복잡한 처리 과정을 포함하고 있는 멀티태스킹 환경에 적용하기가 어렵다. 이에 본 논문에서 제안한 하드웨어 및 소프트웨어 모듈간의 동적 협업 플랫폼은 시스템의 기능을 태스크 단위로 모듈화하여 조립형 형태의 세분화된 소프트웨어 및 하드웨어 모듈로 설계 및 구현이 가능하다. 또한 동적 협업이 요구되는 하드웨어 및 소프트웨어 모듈 간의 통신 및 동기화 기법도 제안하였다. 제안한 하드웨어 및 소프트웨어 모듈간의 동적 협업을 지원하는 SoC 플랫폼의 성능을 분석한 결과, 메모리 접근과 계산 복잡도가 높을수록 소프트웨어 태스크로만 구성된 플랫폼보다 우수한 성능을 보여주었다.

A Study on SoC Platform Design Supporting Dynamic Cooperation between Hardware and Software Modules

Donggeon Lee[†], Youngmann Kim[††], Sungwoo Tak[†††]

ABSTRACT

This paper presents and analyzes a novel technique that makes it possible to improve the performance of low-end embedded systems through SoC (System-on-a- Chip) platform supporting dynamic cooperation between hardware and software modules. Traditional embedded systems with limited hardware resources have the poor capability of carrying out multi-tasking jobs including complex calculations. The proposed SoC platform, which provides dynamic cooperation between hardware and software modules, decomposes a single specific system into tasks for given system requirements. Additionally, we also propose a technique for efficient communication and synchronization between hardware and software tasks in cooperation with each other. Several experiments are conducted to illustrate the application and efficiency of the proposed SoC platform. They show that the proposed SoC platform outperforms the traditional embedded system, where only software tasks run, as the number of memory access is increased and the system become more complex.

Key words: SoC (System on Chip), Dynamic Cooperation of H/W and S/W Modules(하드웨어와 소프트웨어간의 동적 협업), Embedded System(임베디드 시스템)

※ 교신저자(Corresponding Author) : 탁성우, 주소 : 부산광역시 금정구 장전동 산 30(607-735), 전화 : 051)510-2387, FAX : 051)515-2208, E-mail : swtak@pusan.ac.kr
접수일 : 2007년 6월 29일, 완료일 : 2007년 10월 16일
[†] 부산대학교 정보컴퓨터공학부 (E-mail : send234h@naver.com)
[††] 부산대학교 정보컴퓨터공학부 (E-mail : ssomai@gmail.com)
[†††] 부산대학교 정보컴퓨터공학부

1. 서 론

최근의 소형 임베디드 시스템은 다양한 멀티미디어 응용 서비스와 고속의 통신 처리 기능 및 복잡한 연산처리를 동시에 제공할 수 있는 시스템으로 발전하고 있다. 그러나 일반적으로 소형 임베디드 시스템에서는 개인용 컴퓨터의 풍부한 하드웨어 자원보다 낮은 사양을 가지고 있어 복잡한 처리과정이 요구되는 멀티미디어 서비스 및 다른 여러 작업을 동시에 소프트웨어적으로 처리하는 것이 어렵다. 소프트웨어 모듈과 하드웨어 모듈간의 동적 협업이 가능하다면, 멀티미디어 코덱과 고속 연산 처리기를 하드웨어 모듈로 구현하여 응용 소프트웨어 모듈과 구현된 하드웨어 모듈간의 고속 동시 작업이 가능한 소형 임베디드 시스템을 개발할 수 있다. 이러한 시스템에서는 낮은 성능의 CPU를 사용하더라도 풍부한 하드웨어 자원을 가진 시스템과 유사한 성능을 제공할 수 있다. 이러한 동적 협업을 지원하는 시스템을 구현하기 위해서는 주어진 시스템 명세에 대하여 하드웨어 및 소프트웨어 모듈의 분할과 통합을 효율적으로 지원하는 SoC(System on Chip) 플랫폼의 설계 과정과 정립이 필요하다. 또한 하드웨어와 소프트웨어간의 효율적인 동적 협업을 지원할 수 있는 운영체제에 대한 연구가 필요하다. 기존의 임베디드 시스템 운영체제는 소프트웨어 태스크만을 고려한 시스템 자원 관리와 태스크 스케줄링을 고려하였다. 따라서 하드웨어로 구현된 태스크와 소프트웨어 태스크간의 효율적인 협업을 지원하기 위해서는 새로운 운영체제 기법이 필요하다. 그리고 주어진 시스템 명세에 따른 응용 시스템과 서비스를 적시에 개발 및 제공하기 위해서는 소프트웨어 및 하드웨어 모듈간의 동적 구성을 제공하는 태스크 단위의 모듈화 기법이 필요하다. 마지막으로 태스크 단위로 모듈화된 소프트웨어 및 하드웨어 태스크간의 동적 협업을 관리하는데 필요한 통신 및 동기화 기법이 요구된다. 이에 본 논문에서는 소형 임베디드 시스템에서 하드웨어 및 소프트웨어 모듈간의 동적 협업을 제공하는 SoC 플랫폼을 설계 및 구현하였다. 본 논문의 구성은 다음과 같다. 2장에서는 동적 협업을 지원하는 SoC 플랫폼의 설계 기술에 대한 기존 연구를 분석하였으며, 3장에서는 하드웨어 및 소프트웨어 모듈간의 동적 협업을 제공하는 핵심 기술을 설명하였다. 4장에서는 제안한 동적 협업 플랫폼의 성능을 분석하였으며, 마지막으로 5장에서는 결론을 기술하였다.

2. 관련연구

현재 하드웨어 및 소프트웨어 모듈간의 공존성을 제공할 수 있는 플랫폼으로는 SoC기반의 임베디드 시스템 플랫폼을 주로 사용하고 있다. 참조 논문 [1]에서는 SoC기반의 임베디드 시스템 플랫폼에서 하드웨어 및 소프트웨어 모듈간의 공존성을 제공하기 위하여 시스템 개발 초기 단계에서 하드웨어 및 소프트웨어 모듈의 기능을 명확히 구분한 후에 각 기능별로 분리된 모듈의 독립적인 개발을 거쳐 통합한다. 그러나 참조 논문 [1]에서 제시한 정적인 소프트웨어 및 하드웨어 모듈 분할 기법은 각 모듈 간의 동적 협업에 대한 개발 환경을 제공하지 않는다. 따라서 개별 모듈의 통합이 완료된 후에는 소프트웨어 및 하드웨어 모듈의 추가 및 삭제로 인한 모듈 간의 협업 관계가 변경되면 설계 초기 단계에서부터 다시 시스템을 재설계해야 한다. 그리고 하드웨어 및 소프트웨어 모듈간의 협업을 위해서는 운영체제의 지원이 필수적이나 이와 관련된 연구는 미흡하다. 참고문헌 [2-5]에서는 FPGA(Field Programmable Gate Array)에 다중 프로세서를 구현한 후에 각각의 프로세서에서 실행되는 소프트웨어 태스크만을 스케줄링 한다. 다중 프로세서로 구현된 하드웨어 모듈은 소프트웨어의 실행을 담당하는 하드웨어 프로세서의 역할만을 담당한다. 따라서 다중 프로세서 기반에서 실행되는 소프트웨어 모듈 간의 협업만을 고려하였다. 주어진 시스템 명세에 따른 응용 시스템과 서비스를 적시에 개발 및 제공하기 위해서는 높은 조립성을 제공하는 모듈화된 시스템 플랫폼이 필요하다. 조립성 기반의 동적인 시스템 재구성은 개발된 소프트웨어 및 하드웨어 모듈의 재사용성을 극대화할 수 있다. 소프트웨어 및 하드웨어 모듈은 수행을 담당하는 기능 부분과 모듈간의 통신을 담당하는 인터페이스 부분으로 구성된다. 일반적으로 기존의 SoC 플랫폼에서 소프트웨어 모듈간의 통신은 운영체제에서 지원하는 IPC(Inter-Process Communication) 기법을 이용하며, 하드웨어 모듈간의 통신은 SoC 시스템에서 정의된 내부 버스 규약을 이용한다. 따라서 기존의 SoC 플랫폼에서 소프트웨어 모듈과 하드웨어

모듈간의 통신을 제공하기 위해서는 새로운 통신 인터페이스의 설계가 요구된다. 특히 하드웨어 모듈인 경우에는 내부 버스 규약뿐만 아니라 소프트웨어 모듈간의 통신도 고려해야 한다. 그러나 소프트웨어 모듈과 하드웨어 모듈간의 동적 협업에 필요한 통신 기법과 관련된 기존의 연구는 미흡한 실정이다. 소프트웨어 모듈간의 통신 기법으로는 파라미터 전달을 이용하여 소프트웨어 모듈 내의 프로시저를 호출하는 방식과 이벤트 기반의 특정 행동을 소프트웨어 모듈에 요청하는 방식이 있다. 그리고 공유 메모리 영역과 메시지 큐를 이용하여 소프트웨어 모듈간의 데이터를 교환하는 방식이 있다[6-7]. 참고 문헌 [8]에서는 메시지 큐 기법을 이용하여 하드웨어와 소프트웨어 모듈간의 통신을 제공한다. 그러나 이 방식에서는 하드웨어 모듈 내에 고정된 크기의 메시지 큐만을 구현할 수 있기 때문에 각각의 하드웨어 모듈과 통신할 수 있는 소프트웨어 모듈의 개수가 제한되는 문제점이 있다. 분석된 기존의 관련 연구를 기반으로 하여 본 논문에서 접근하고자 하는 방향은 다음과 같다. 하드웨어 모듈과 소프트웨어 모듈을 태스크 단위로 설계 및 구현하고자 한다. 태스크는 완전히 독립된 수행이 가능하도록 명령어 실행 코드 및 실행에 필요한 데이터, 그리고 시스템의 상태 정보를 저장할 수 있는 공간으로 구성된 기능 단위이다. 태스크 단위의 하드웨어 모듈에 대한 구현 방식은 다음과 같이 구체화된다. 명령어 실행 코드는 산술논리 연산 코어를 담당하는 하드웨어로 구현되며, 실행 코드 처리에 필요한 데이터를 저장할 수 있는 임시 저장 공간은 하드웨어 버퍼로 구현되고, 시스템의 상태 정보를 저장할 수 있는 공간은 하드웨어적으로 설계된 상태 레지스터로 구현된다. 다음 장에서는 본 논문에서 제안한 핵심 기법에 대하여 기술하였다.

3. 하드웨어 및 소프트웨어 모듈간의 동적 협업 기법

3.1 SoC 플랫폼을 이용한 하드웨어 설계 과정

그림 1은 SoC 기반의 임베디드 시스템 플랫폼에서 하드웨어 및 소프트웨어 모듈간의 공존성만을 제공하는 기존의 SoC 설계 과정과 본 논문에서 제안하는 하드웨어 및 소프트웨어 모듈 간의 동적 협업을 제공하는 SoC 설계 과정을 보여준다.

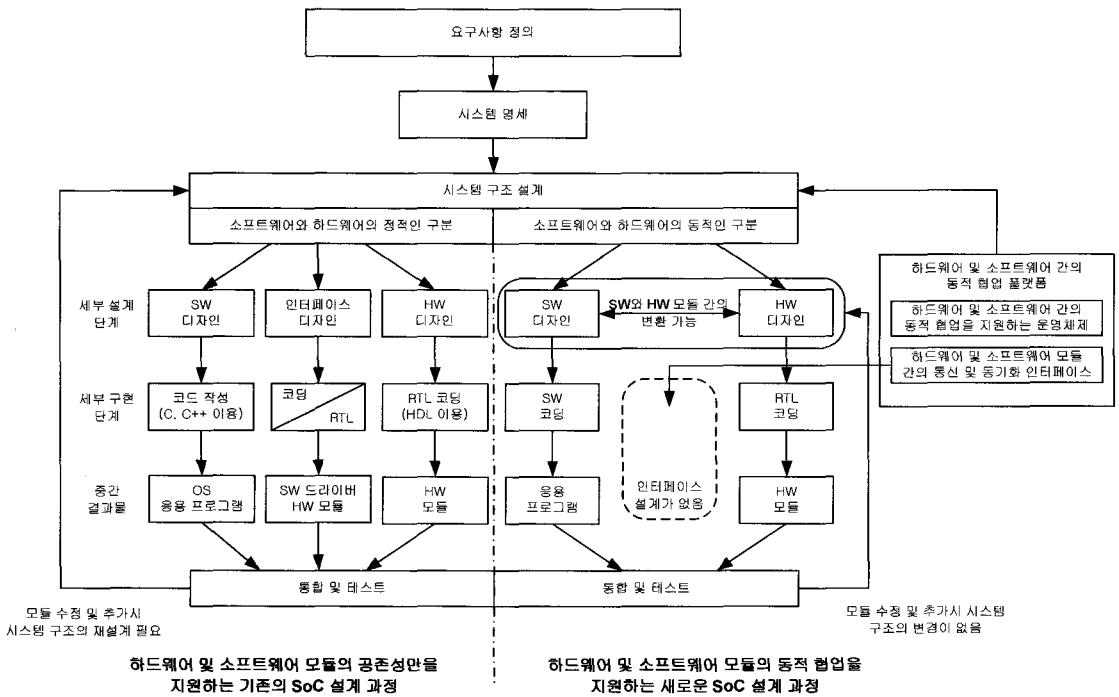


그림 1. 하드웨어 및 소프트웨어 모듈의 공존성과 동적 협업을 지원하는 SoC 설계 과정

기존의 SoC 설계 과정을 살펴보면 다음과 같다. 시스템 개발 초기 단계에서 하드웨어 및 소프트웨어 모듈의 기능을 정적으로 명확히 구분하고, 각 모듈별로 분리된 개발 과정을 거친 후에 통합과정을 수행한다. 만약 소프트웨어 모듈과 하드웨어 모듈간의 분할 작업이 완료된 이후에 두 모듈간의 협업관계가 변경되거나 새로운 기능의 추가 혹은 삭제가 필요하면 설계 초기 단계에서부터 재설계 작업이 필요하다. 또한 재설계 과정에서 이전에 설계된 소프트웨어 모듈과 하드웨어 모듈간의 인터페이스 변경이 요구되면 인터페이스의 재설계뿐만 아니라 두 모듈간의 기존 인터페이스에 대한 재설계 과정도 필요하다.

본 논문에서 제안하는 설계 기법은 초기 설계 단계에서부터 하드웨어 및 소프트웨어 모듈 간의 효율적인 동적 협업을 제공한다. 따라서 세부 설계 단계에서 초기 설계한 소프트웨어 모듈이 하드웨어 모듈로 변경이 요구되더라도 기존에 설계된 다른 모듈의 수정 없이 자유롭게 변환이 가능하다. 그리고 협업 관계가 변경되거나 새로운 기능이 추가되어도 기본 설계 구조의 변경 없이 수정된 부분에 대해서만 세부 설계를 수행하면 되는 장점이 있다. 동적 협업을 지원하는 운영체제와 각 모듈 간의 통신 및 동기화를 지원하는 효율적인 공통 인터페이스를 초기 설계 단계과정에서 제공하기 때문에, 개발자는 제안한 공통 인터페이스에 따라 소프트웨어 모듈과 하드웨어 모듈을 독립적으로 구현하더라도 통합 과정에 필요한 부가적인 오버헤드가 필요 없다.

3.2 동적 협업 SoC 플랫폼 구조

그림 2는 동적 협업 기법을 설계하는데 필요한 SoC 플랫폼을 보여준다. SoC 플랫폼은 ARM9기반의 프로세서 코어와 메모리 및 버스, 그리고 주변장치로 구성된다. 하드웨어 모듈과 소프트웨어 모듈간의 협업을 지원하는 운영체제와 소프트웨어 모듈은 메모리에 탑재된다. 그리고 FPGA로 구현된 하드웨어 모듈은 버스 브릿지를 통해 상호 연결된다. 시스템 버스로 사용되는 AMBA버스는 CPU의 동작 클럭과 동일한 클럭으로 동작하는 AHB1 버스와 CPU 클럭의 1/2로 동작하는 AHB2 버스로 구성된다. AHB1 버스에 연결되어 있는 소프트웨어 모듈은 100MHz 클럭으로 동작되며, 하드웨어 모듈은 AHB2 버스에 연결되어 있어 50MHz로 동작한다.

하드웨어 모듈과 소프트웨어 모듈 간의 동적 협업을 지원하기 위해서는 그림 2-(a)의 하드웨어 모듈과 소프트웨어 모듈 간의 통신 기법과 그림 2-(b)의 하드웨어와 소프트웨어 모듈 간의 동기화 기법, 그리고 그림 2-(c)의 하드웨어 모듈과 소프트웨어 모듈 간의 스케줄링 기법이 필요하다. 또한 하드웨어 모듈과 소프트웨어 모듈 간의 통신 인터페이스와 협업 관리자도 필요하다. 먼저 그림 2에서 기술한 하드웨어 및 소프트웨어 모듈간의 협업을 지원하는 운영체제의 세부 구조를 살펴보면 그림 3과 같다. 그림 3-(a)는 하드웨어 모듈 및 소프트웨어 모듈간의 협업을 지원하지 않는 운영체제의 구조를 보여준다. 그림 3-(a)에서 보는 바와 같이 소프트웨어 태스크간의 통신은

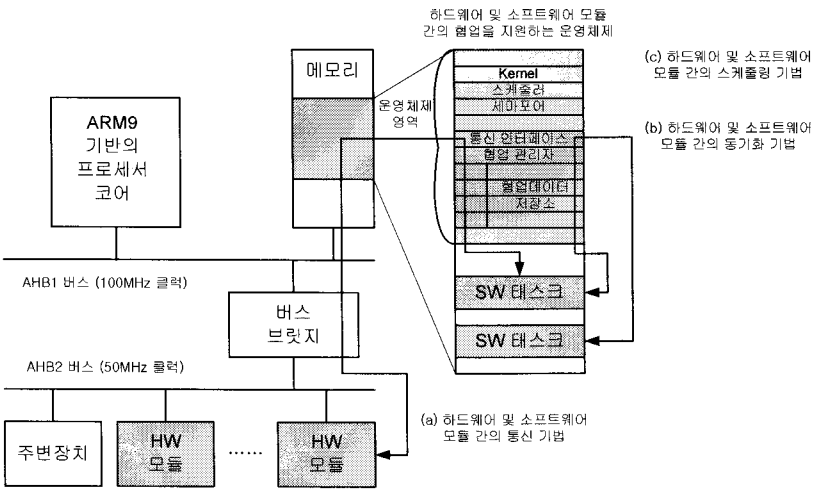


그림 2. 동적 협업 기법의 설계에 필요한 SoC 플랫폼 구조

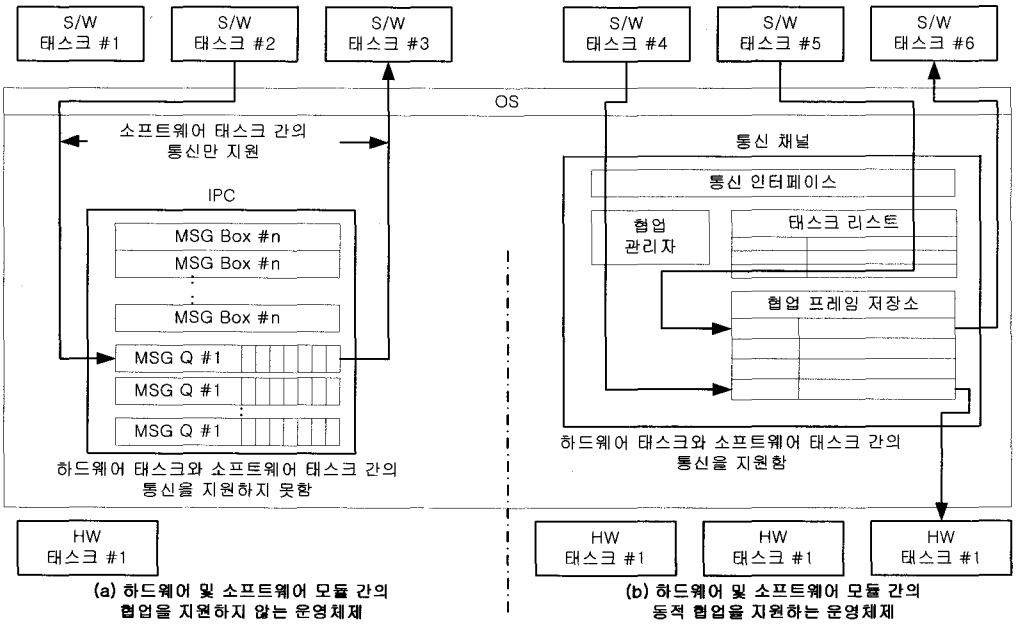


그림 3. 하드웨어 및 소프트웨어 태스크 간의 동적 협업을 지원하는 운영체제 구조

메시지 박스와 같이 기존 IPC를 그대로 사용할 수 있지만, 기존의 IPC 기법만을 사용하여 소프트웨어 태스크와 하드웨어 태스크간의 통신을 제공할 수는 없다.

그림 3-(b)는 하드웨어 모듈과 소프트웨어 모듈을 태스크 단위로 구현한 후에 하드웨어 및 소프트웨어 태스크간의 동적 협업을 지원하는 운영체제의 구조를 보여 준다. 그림 3-(b)에서 보는 바와 같이 하드웨어 태스크 및 소프트웨어 태스크 간의 동적 협업을 지원하는 운영체제에서는 소프트웨어 태스크 및 하드웨어 태스크간의 통신뿐만 아니라 하드웨어 태스크간의 통신과 하드웨어 모듈과 소프트웨어 모듈 간의 공통된 통신 인터페이스를 제공한다. 그림 3-(b)의 세부 동작 기법은 그림 5를 설명하는 부분에서 기술한다.

3.3 동적 협업 So

그림 4는 하드웨어와 소프트웨어 태스크간의 동적 협업 과정에서 상호 교환되는 협업 프레임의 형식을 보여준다.

하드웨어 및 소프트웨어 태스크간의 협업을 위해 교환하는 데이터는 협업 프레임의 형식에 맞게 전달된다. 협업 프레임은 헤더와 데이터 영역으로 구분된다. 헤더는 프레임의 전송과 제어를 위한 정보를 담고 있으며, 제어 필드, 송신 태스크 ID, 수신 태스크 ID, 협업 데이터의 크기, 그리고 협업 프레임 관리 태스크 영역으로 나누어진다. 협업 데이터 필드에는 태스크에 전달되는 데이터가 저장된다. 태스크 ID는 태스크의 고유 식별자로 사용된다. 협업 데이터 크기 필드는 전송되는 협업 데이터의 크기를 바이트 단위로 표현한다. 전송되는 협업 데이터는 협업 프레임

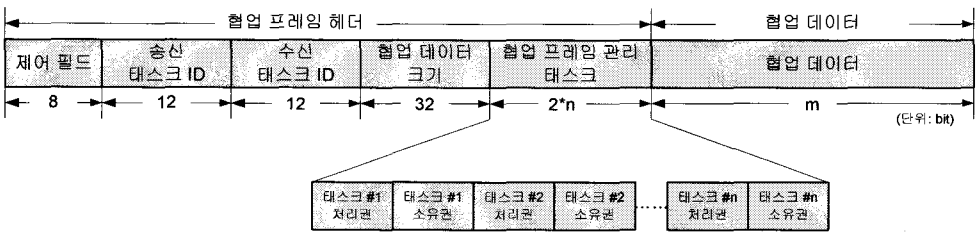


그림 4. 협업 프레임 구조

관리 태스크 필드에서 명시된 태스크들에 의해서 처리되어진다. 협업 프레임 관리 태스크 필드는 태스크의 처리권과 소유권으로 구분되어진다. 처리권은 특정 태스크가 현재 협업 데이터를 처리하고 있는 중임을 나타내어 동시에 여러 태스크가 협업 데이터에 접근하는 것을 막는다. 그리고 소유권은 협업 데이터를 처리해야 할 태스크를 명시한다. 표 1은 협업 프레임에서 사용하는 제어 필드의 속성 값을 보여주고 있다.

표 1. 제어 필드의 속성 값

제어 필드 값	상태 값	설명
0×00	NEW_FRAME	비어 있는 협업 프레임
0×01	READY_FRAME	처리해야 할 협업 프레임이 준비됨
0×02	RUN_FRAME	협업 프레임이 처리 중임
0×03	SUSPEND_FRAME	협업 프레임의 처리가 지연 중임
0×04	TERMINATE_FRAME	협업 프레임의 처리가 완료됨
0×05~ 0×FF	RESERVED	예약된 상태 값

그림 5는 하드웨어 및 소프트웨어 태스크간의 협업 처리 과정을 보여준다. 그림 5-(A)의 협업 프레임 저장소는 협업 프레임을 저장하는 메모리 영역이다. 그림 5-(B)의 태스크 리스트 테이블은 하드웨어 및 소프트웨어 태스크간의 통신 및 동기화를 관리하는데 필요한 정보를 저장한다. 태스크 리스트에서 관리하는 정보는 다음과 같다. 태스크 식별을 위한 태스크 ID와 태스크의 종류 (하드웨어 태스크 혹은 소프트웨어 태스크)를 구분하는 태스크 형태, 그리고 태스크간의 동기화에 사용되는 태스크 세마포어와 마지막으로 태스크 주소 공간에 협업 프레임의 주소를 저장하는 협업 레지스터 주소가 있다. 하드웨어 태스크와 소프트웨어 태스크는 작업시 교환되는 협업 프레임의 주소를 협업 레지스터에 저장한다. 송신 태스크가 전달한 협업 프레임은 협업 프레임 저장소에 저장되며, 협업 스케줄 관리자는 수신 태스크에게 저장되어 있는 협업 프레임이 저장되어 있는 메모리 주소를 전달한다. 협업 프레임 할당 관리자는 협업 프레임 저장소로부터 비어있는 협업 프레임 (즉, 헤더의 제어 필드가 0x00인 협업 프레임)을 찾아 협업

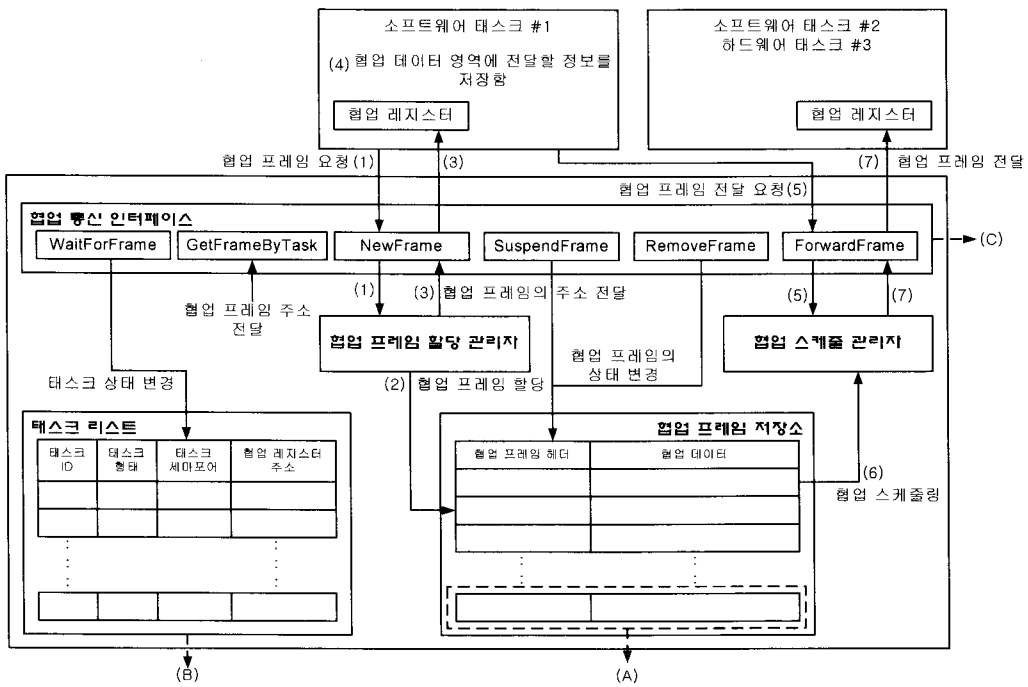


그림 5. 하드웨어 및 소프트웨어 태스크 간의 협업 처리 과정

프레임의 사용을 원하는 태스크에게 전달하는 기능을 수행한다. 그림 5-(C)의 협업 통신 인터페이스는 하드웨어 태스크 및 소프트웨어 태스크간의 통신을 위해 제공되는 인터페이스로써 하드웨어 모듈과 소프트웨어 모듈이 동일한 인터페이스를 제공하기 때문에 협업 대상의 구분 없이 상호 투명한 통신이 가능하다.

표 2는 태스크간의 동적 협업을 위한 통신 인터페이스의 함수를 보여준다. 통신 인터페이스 함수는 태스크나 협업 프레임의 처리 상태를 변화시키고, 새로운 협업 프레임을 요청하거나 전달하는 기능을 제공한다. 소프트웨어 태스크는 협업을 위한 통신 인터페이스 함수에서 제공하는 모든 함수를 호출할 수 있다. 그러나 하드웨어 태스크는 매 클럭마다 협업 레지스터를 검사하기 때문에 WaitForFrame() 함수를 사용할 필요가 없으며, 처리해야 할 협업 프레임이 있는지를 확인한 후에 협업 레지스터에 협업 프레임의 주소가 기록되어 있으면 협업 프레임을 처리한다. 따라서 하드웨어 태스크는 협업 레지스터의 내용을 직접 접근하기 때문에 GetFrameByTask() 함수를 사용할 필요가 없다. 이 두 인터페이스 함수를 제외한 나머지 함수들은 하드웨어 태스크에서도 사용된다.

그림 5-(1)부터 그림 5-(7)까지는 표 2에서 기술한 통신 인터페이스 함수를 기반으로 하여 동작되는 협업 과정을 보여준다. 먼저 그림 5-(1)에서 태스크 #1이 NewFrame() 함수를 호출하여 협업 프레임 저장소에서 빈 협업 프레임을 협업 프레임 관리자에게 요청하면, 그림 5-(2)에서 협업 프레임 할당 관리자는 빈 협업 프레임을 할당하고, 그림 5-(3)에서 협업 프레임 할당 관리자는 할당된 협업 프레임의 주소를 태스크 #1의 협업 레지스터에 저장한다. 그림 5-(4)에서 태스크 #1은 협업 레지스터에 저장되어 있는 주소가 가리키는 협업 프레임의 데이터 영역에 협업 데이터를 저장한다. 저장이 완료되면, 그림 5-(5)에서 태스크 #1은 ForwardFrame() 함수를 호출하여 협업 스케줄 관리자가 협업 프레임의 주소를 태스크 #2에게 전달하도록 요청한다. 전달 요청을 받은 협업 스케줄 관리자는 그림 5-(6)의 협업 스케줄링을 통하여 WaitforFrame() 함수를 호출하여 대기하고 있는 태스크 #2에게 협업 프레임의 주소를 전달한다. 그림 5-(7)에서 태스크 #2는 협업 프레임의 주소를 기반으로 하여 태스크 #1로부터 전달받은 협업 프레임을

표 2. 협업을 위한 통신 인터페이스 함수

구문	인자	반환값	기능
void WaitForFrame (int TID)	TID: WaitForFrame() 함수를 호출한 태스크의 ID	void	태스크는 자신에게 전달되는 협업 프레임이 도착할 때까지 대기함
int* GetFrameByTask (int TID)	TID: GetFrameByTask ()함수를 호출한 태스크의 ID	협업프레임 주소	태스크는 협업 스케줄러에게 협업 레지스터에 저장되어 있는 주소에 해당하는 협업 프레임을 획득함
int* NewFrame (int FrameSize)	FrameSize: 협업프레임의 크기	협업프레임 주소	태스크는 FrameSize만큼의 빈 협업 프레임을 요구하면 협업 프레임 할당 관리자는 협업 프레임 저장소에서 요청한 크기만큼의 협업 프레임을 할당한 후에 해당 주소를 반환함
void ForwardFrame (int DTID, int* FrameAddr)	DTID: 수신 태스크 ID FrameAddr: 협업프레임 주소	void	송신 태스크는 협업 스케줄 관리자가 FrameAddr 주소를 가지는 협업프레임을 수신 태스크에게 전달하도록 요청함
void SuspendFrame (int TID, int* FrameAddr)	TID: SuspendFrame() 함수를 호출한 태스크 ID FrameAddr: 협업프레임 주소	void	현재 시점에서 FrameAddr 주소에 해당하는 협업 프레임의 처리가 완료된 태스크는 향후 해당 협업 프레임을 계속 사용할 수 있음을 협업 스케줄 관리자에게 알려 해당 협업 프레임을 계속 유지할 것을 요청함
void RemoveFrame (int TID, int* FrameAddr)	TID: RemoveFrame() 함수를 호출한 태스크 ID FrameAddr: 협업프레임 주소	void	FrameAddr 주소에 해당하는 협업 프레임의 처리를 완료한 태스크는 협업 스케줄 관리자에게 알려주고 다른 태스크들에 의하여 더 이상 해당 협업 프레임에 대한 접근이 없으면 협업 스케줄 관리자는 해당 협업 프레임을 폐기함

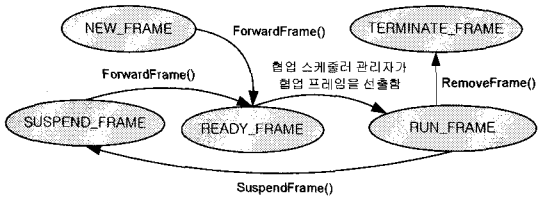


그림 6. 협업 프레임의 상태 천이도

GetFrameByTask() 함수를 호출한 후에 해당 프레임에 접근하여 저장된 정보를 읽을 수 있다.

그림 6은 협업 프레임의 상태 천이도를 보여준다. 협업 프레임 할당 관리자에 의해 새로 생성된 협업 프레임은 NEW_FRAME 상태가 된다. 송신 태스크가 NewFrame() 함수를 호출하면 협업 프레임 할당 관리자는 NEW_FRAME 상태의 협업 프레임의 주소를 송신 태스크에게 전달한다. 협업 프레임의 주소를 전달받은 송신 태스크는 협업 프레임의 소유권 필드에 자신의 소유권을 표시하며, 협업 데이터 영역에 전달하고자 하는 데이터를 기록한 후에 ForwardFrame() 함수를 호출한다. ForwardFrame() 함수가 호출된 후에 협업 프레임은 READY_FRAME 상태로 전이된다. 협업 스케줄 관리자는 READY_FRAME 상태의 협업 프레임을 선출하면, 수신 태스크에 해당 협업 프레임의 주소가 전달되고 협업 프레임은 RUN_FRAME 상태로 전이된다. 수신 태스크는 SuspendFrame() 함수를 호출하여 해당 협업 프레임의 처리를 완료하더라도 태스크는 향후 해당 협업 프레임을 계속 사용할 수 있음을 협업 스케줄 관리자에게 알려서 해당 협업 프레임을 계속 유지할 것을 요청한다. SuspendFrame() 함수가 호출되면 해당 협업 프레임은 SUSPEND_FRAME 상태로 변경된다. 협업 프레임에 대한 모든 처리가 완료되면 RemoveFrame() 함수를 호출하여 해당 협업 프레임을 폐기한다.

4. 성능분석

4.1 설계 방법에 따른 성능 분석

본 논문에서 제안한 하드웨어 및 소프트웨어 모듈간의 동적 협업을 지원하는 플랫폼의 설계 및 구현, 그리고 성능 분석에 사용된 SoC 플랫폼의 개발 환경은 다음과 같다. ARM9 코어를 내장한 Altera사의 Excalibur EPXA4 칩을 탑재한 SoC 칩을 사용하였다[9]. 소프트웨어 모듈은 C언어로 작성하였으며, ADS 1.2 컴파일러를 이용하여 구현하였다. 하드웨어 모듈은 Verilog HDL(Hardware Description Language)과 VHDL(Very-high-speed integrated circuit Hardware Description Language)을 혼용하여 설계한 후에 Quartus II 4.0을 이용하여 설계된 하드웨어 모듈을 구현하였다.

주어진 시스템 명세에서 분석된 전체 기능 모듈 중에서 어떤 모듈을 소프트웨어 및 하드웨어 태스크로 변환하는 것이 효율적인지를 먼저 분석하였다. 표 3에서는 이러한 분석을 위하여 각각의 실험 환경을 구성하고 1개의 태스크를 하드웨어 태스크와 소프트웨어 태스크로 구현하여 1000번의 반복 실행에 대한 평균값을 측정하였다.

먼저 실험 1에 대하여 살펴보면 다음과 같다. 그림 2에서 보는 바와 같이 AHB2 버스에 연결되어 있는 하드웨어 모듈이 AHB1 버스에 연결되어 있는 메모리에 접근하기 위해서는 하드웨어 모듈은 먼저 AHB1 버스의 사용 권한을 획득한 후에 메모리에게 주소를 전달해야 한다. 버스 사용 권한의 획득과 메모리 주소의 전달에 소요되는 오버헤드를 줄이기 위

표 3. 태스크 형태에 따른 성능 분석

	내용	방법	횟수
실험 1	메모리 I/O 방식에 따른 성능 분석	IP 프로토콜 체크섬 태스크의 처리 시간 측정 (하드웨어 타이머의 값 측정)	1000
실험 2	계산 복잡도에 따른 성능 분석	계산 복잡도 O(n)을 가지는 단순 합 태스크의 처리 시간 측정 (1부터 목표 수까지 단순히 더함)	1000
		계산 복잡도 $O(n^2)$을 가지는 단순 곱 태스크의 처리 시간 측정 (1부터 목표 수까지 곱셈표를 생성함)	1000
		계산 복잡도 $O(n^3)$을 가지는 행렬 곱 태스크의 처리 시간 측정 (두 N×N 행렬의 곱을 계산함)	1000

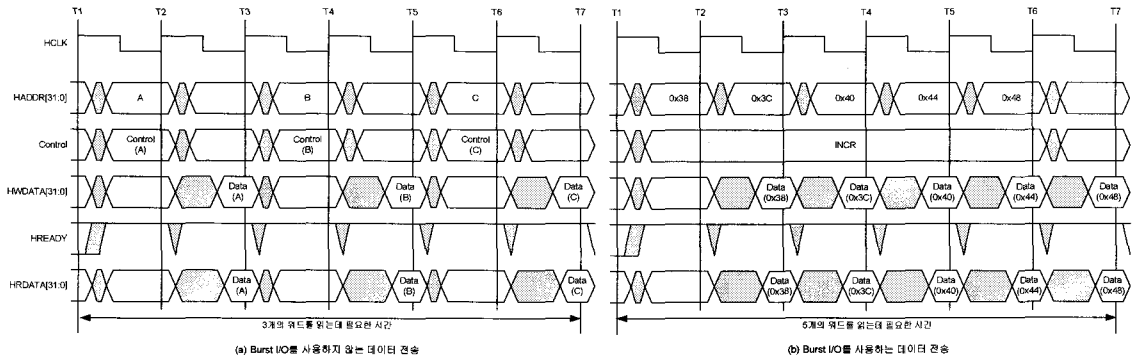


그림 7. 버스트 I/O 사용 유무에 따른 데이터 전송 타이밍도

해서는 버스 사용 권한의 획득 후에 일정 크기의 데이터를 연속적으로 전송할 수 있는 버스트 I/O 기법이 필요하다.

그림 7은 버스트 I/O 사용 유무에 따라서 메모리에 저장되어 있는 데이터를 읽는데 소요되는 시간을 보여준다. 그림 7-(a)에서 보는 바와 같이 버스트 I/O 기법을 사용하지 않는 경우에는 처음 단계에서 메모리에 주소를 전달하고 데이터를 받는데 3클럭이 소요되지만 그 이후부터는 매 2클럭마다 주소와 데이터를 교환하여 7클럭 내에 3워드를 읽을 수 있다. 그러나 그림 7-(b)에서 보는 바와 같이 버스트 I/O 기법을 사용하는 경우에는 매 클럭마다 데이터를 읽을 수 있기 때문에 7클럭 내에 5워드를 읽을 수 있다.

그림 8에서는 소프트웨어 태스크와 버스트 I/O의 사용 유무에 따른 하드웨어 태스크가 IP(Internet Protocol) 패킷의 체크섬을 처리하는데 걸린 시간을 보여준다. 소프트웨어 태스크 및 하드웨어 태스크는 협업 프레임 저장소에 있는 협업 프레임을 4바이트 단위로 읽은 후에 협업 프레임에 저장되어 있는 IP 패킷의 체크섬을 계산한다. 버스트 I/O 기법을 사용

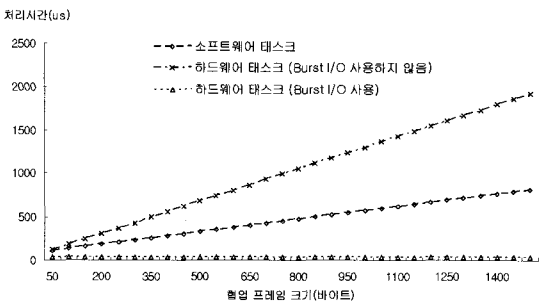


그림 8. 메모리 I/O 형태에 따른 성능 비교

하는 하드웨어 태스크는 버스 사용의 권한을 획득한 후에 8워드(32바이트)를 매 클럭마다 연속적으로 전달받는다. 그러나 버스트 I/O 기법을 사용하지 않는 하드웨어 태스크는 4바이트 단위로 협업 프레임을 읽을 때마다 버스 사용 권한의 획득 및 메모리 주소의 전달을 계속 수행하여 협업 프레임에 저장되어 있는 IP 패킷의 체크섬을 계산한다. 따라서 버스트 I/O를 사용하지 않는 하드웨어 태스크인 경우에는 앞서 기술한 버스 사용 권한의 획득 및 메모리 주소의 전달에 대한 오버헤드가 있기 때문에 그림 8에서 소프트웨어 태스크가 버스트 I/O를 사용하지 않는 하드웨어 태스크보다 성능이 더 우수하다. 그러나 버스트 I/O를 사용하는 경우에는 하드웨어 태스크의 성능이 소프트웨어 태스크의 성능보다 우수함을 보여준다.

그림 9~그림 11에서는 계산 복잡도에 따른 태스크의 성능을 분석하였다. 실험에서 사용되는 태스크는 메모리 접근을 하지 않으며, 랜덤하게 생성된 데이터를 계산한 후에 결과 값을 메모리에 저장하지 않고 버린다. 그림 9에서 보는 바와 같이 계산 복잡도 O(n)을 가지는 태스크의 성능 비교에서는 소프트웨어 태스크가 더 좋은 성능을 보여준다. 반복적으로 더하는 연산인 경우에는 소프트웨어 태스크도 1클럭 내에 계산이 가능하기 때문에 50MHz 클럭에서 동작하는 하드웨어 태스크보다 100MHz 클럭에서 동작하는 소프트웨어 태스크의 처리 시간이 더 빠르다. 그러나 그림 10과 그림 11에서 보는 바와 같이 계산 복잡도가 높아지면 하드웨어 태스크의 처리 시간이 더 빨라짐을 확인할 수 있다. 그림 11에서 다수의 곱셈기를 이용하는 경우에는 행렬의 여러 원소를 1클

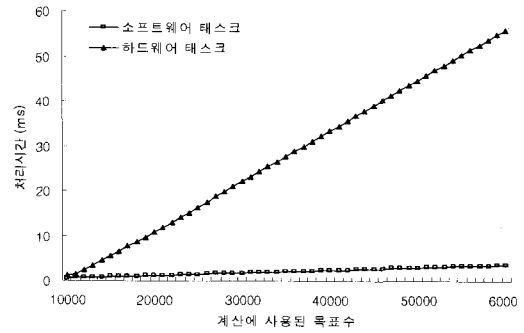


그림 9. 계산 복잡도에 따른 처리 성능 분석 O(n)

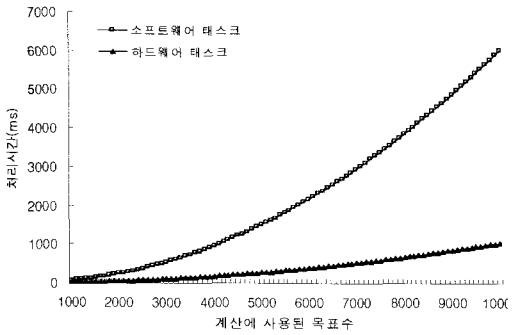


그림 10. 계산 복잡도에 따른 처리 성능 분석 $O(n^2)$

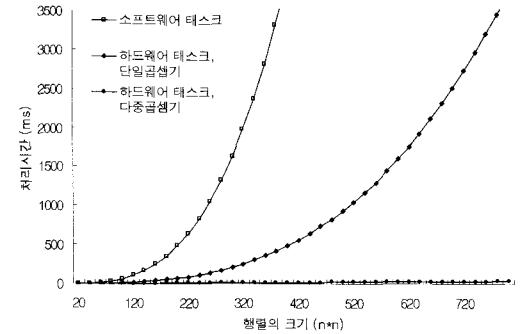


그림 11. 계산 복잡도에 따른 처리 성능 분석 $O(n^3)$

럭 내에 계산이 가능하여 처리 시간이 매우 짧아짐을 확인하였다. 이에 계산 복잡도가 높은 태스크는 하드웨어 태스크로 구현해야 한다.

4.2 모듈간 통신 성능 분석

기존의 운영체제에서 제공하는 소프트웨어 모듈간의 통신 기법과 본 논문에서 제안한 소프트웨어와 하드웨어 모듈간의 통신 기법간의 오버헤드를 분석하였다. 기존 운영체제에서 제공하는 소프트웨어 태스크간의 통신 기법을 분석하기 위하여 초소형 임베디드 운영체제인 uC/OS-II와 uC/OS-II에서 제공하는 세마포어 기법을 SoC 임베디드시스템 보드에 이식한 후에 통신 기법에 대한 실험을 수행하였다[10]. 초소형 임베디드 운영체제인 uC/OS-II는 경량화된 커널 구조와 태스크간의 빠른 통신 기법을 제공한다. 본 논문에서 제안한 동적 협업 플랫폼은 세마포어 기반의 동기화를 지원하므로 공정한 성능 비교를 위해 세마포어 기법을 사용하여 4바이트 단위의 데이터 전송을 비교 대상으로 정하였다. 시간 측정은 CPU 내부에 내장된 하드웨어 타이머를 이용하였다.

표 4에서 보는 바와 같이 소프트웨어 태스크간의 통신에서는 uC/OS-II가 더 짧은 전송 시간을 보여주며, 협업 통신 인터페이스를 사용하는 소프트웨어와 하드웨어 태스크간의 통신이 세 가지 통신 기법 중에서 제일 빠름을 보여준다. 협업 통신 인터페이스를 사용하는 소프트웨어 태스크간의 동기화 및 4바이트 단위의 협업 프레임을 전송하기 위해서는 NewFrame(), ForwardFrame(), WaitForFrame(), 그리고 GetFrameByTask() 함수가 순차적으로 호출되며, 마지막으로 GetFrameByTask() 함수가 호출된 후에 협업 스케줄 관리자는 해당 협업 프레임을 요청한 태스크에게 전달하게 되는 오버헤드가 발생한다. 그러나 협업 통신 인터페이스를 사용하는 소프트웨어 태스크와 하드웨어 태스크간의 통신에서는 NewFrame()과 ForwardFrame() 함수가 호출된 후에 바로 하드웨어 태스크에 전달되기 때문에 오버헤드가 소프트웨어 태스크간의 통신보다 낮음을 알 수 있다.

제안한 하드웨어 및 소프트웨어 모듈간의 동적 협업에서 사용하는 협업 통신 인터페이스의 성능 분석을 위하여 협업 할당 관리자의 협업 프레임 할당 기

표 4. 모듈간의 통신 성능 결과

	uC/OS-II 환경에서 소프트웨어 태스크간의 통신	협업 통신 인터페이스를 사용하는 소프트웨어 태스크 간의 통신	협업 통신 인터페이스를 사용하는 소프트웨어 태스크와 하드웨어 태스크간의 통신
처리시간 (us)	94	130	40

표 5 협업 통신 인터페이스 성능 분석을 위한 태스크 생성

	기능	협업 프레임 생성 간격	태스크 우선순위
태스크 #1	50*50 행렬을 생성하여 태스크 #2 ~ 태스크 #4에 전달함	0.1 ms	1
태스크 #2	전달 받은 50*50 행렬을 곱하여 결과를 태스크 #5에 전달함	해당사항 없음	3
태스크 #3		해당사항 없음	3
태스크 #4		해당사항 없음	3
태스크 #5	처리 시간을 계산하여 통계 정보를 기록함	해당사항 없음	2

법과 협업 스케줄 관리자의 협업 프레임 스케줄링 기법, 그리고 협업 프레임 저장소의 크기에 따른 실험을 하였다.

표 5에서 보는 바와 같이 5개의 태스크를 생성하였으며 태스크 #1은 50*50 행렬을 생성한 후에 태스크 #2부터 태스크 #4까지 협업 통신 인터페이스를 사용하여 전달한다. 태스크 #2부터 태스크 #4는 모든 행렬을 전달받은 후에 그 결과를 태스크 #5에 전달한다. 협업 통신 인터페이스에서 소요되는 최악의 처리 시간을 분석하기 위하여 모든 태스크는 소프트웨어 태스크로 구현하였으며 협업 데이터를 생성하는 간격도 매우 빠르게 설정하였다. 따라서 협업 데이터를 생성하는 태스크 #1의 우선순위가 가장 높으며, 처리 시간을 계산하는 태스크 #5가 다음 우선순위를 가진다. 행렬 곱을 계산하는 태스크 #2부터 태스크 #4까지는 동일한 우선순위를 가진다. 태스크 #1에서 행렬을 생성한 후에 협업 통신 인터페이스를 통해 정보를 전달하기 시작하는 시간부터 행렬 곱 연산을 마치고 최종적으로 태스크 #5에서 정보를 받는 데 소요되는 시간을 측정하였다. 태스크에서 행렬 곱을 수행하는 데 소요되는 시간은 행렬의 크기가 모두 일정하므로 상수 시간으로 가정할 수 있다.

성능 비교에 사용한 협업 프레임의 할당 기법은 최초 적합과 다음 적합 그리고 태스크별 FIFO (First-In First-Out) 큐 기법이다. 최초 적합 기법은 협업 프레임 저장소의 처음부터 순차적으로 검사하면서 최초로 탐색된 비어있는 협업 프레임을 할당하는 기법이다. 다음 적합 기법은 가장 최근에 할당된 협업 프레임 저장소의 위치에서 순차적으로 검사하면서 최초로 탐색된 비어있는 협업 프레임을 할당하는 기법이다. 태스크별 큐 기법은 협업 프레임 저장소를 각 태스크 별로 구분하고 각 부분을 FIFO 큐의 형태로 할당하는 기법이다.

또한 성능 비교에 사용한 협업 프레임 스케줄링 기법은 최초 선출과 공통 FIFO 선출 및 태스크별 FIFO 선출 기법이 있다. 최초 선출 기법은 협업 프레임 저장소의 처음부터 순차적으로 검사하면서 수신 태스크의 협업 프레임 중 맨 처음 발견된 협업 프레임을 스케줄링하는 기법이다. 공통 FIFO 선출은 ForwardFrame() 함수가 호출되어 협업 프레임을 전송받으면, 협업 프레임을 공통 FIFO 큐에 등록하고 공통 FIFO 큐에서 순차적으로 검사하면서 협업 프레임을 선출하는 기법이다. 태스크별 FIFO 선출 기법은 협업 프레임 저장소를 각 태스크별로 구분한 뒤 각각을 FIFO 방식의 형태로 선출하는 기법이다.

표 6은 표 5의 실험 결과에서 협업 프레임 할당 기법과 협업 프레임 스케줄링 기법에 따른 행렬 연산의 처리시간을 분석한 결과이다. 5000개의 협업 프레임을 저장할 수 있는 협업 프레임 저장소를 사용하였다. 협업 프레임 할당 기법에서는 태스크별 FIFO 큐 방식이 다른 방식에 비해 짧은 처리 시간을 보여준다. 또한 협업 스케줄링 기법에서는 태스크별 FIFO 선출 기법이 다른 기법에 비해 짧은 처리 시간을 보여준다.

협업 프레임 저장소의 크기에 따른 성능 분석은 표 5에서 수행한 실험환경과 동일하며, 이에 대한 분석 내용은 다음과 같다. 협업 프레임 저장소의 크기를 500에서 10000으로 변화시키면서 협업 프레임의

표 6. 협업 프레임 할당 기법과 스케줄링 기법에 따른 처리 시간 비교 (단위: us)

할당 기법 / 스케줄링 기법	최초 적합	다음 적합	태스크별 FIFO 큐
최초 선출	24313	30922	23076
공통 FIFO 선출	43726	47571	39331
태스크별 FIFO 선출	30274	23755	23763

할당 기법과 협업 프레임의 스케줄링 기법에 따른 처리 시간을 측정하였다. 그림 12에서 보는 바와 같이 협업 프레임 저장소의 크기가 커질수록 평균 처리 시간이 증가함을 볼 수 있다. 행렬의 크기가 일정하여 행렬 곱을 계산하는 시간은 일정하다. 따라서 증가된 처리시간은 협업 프레임의 할당 관리 기법과 협업 프레임의 스케줄링 기법에 의해 발생한 오버헤드이다. 협업 프레임의 저장소가 증가하면 여러 태스크가 동시에 많은 협업 프레임을 전달할 수 있지만, 협업 프레임의 할당 및 스케줄링에 대한 많은 오버헤드가 발생함을 확인하였다. 그림 12에서 보는 바와 같이 협업 프레임 할당 기법에서는 태스크별 FIFO 큐를 적용하고, 협업 프레임 스케줄링 기법에서는 태스크별 FIFO 선출 기법을 사용하는 경우가 가장 빠른 처리 시간을 보여 주었다. 협업 프레임 할당 기법이 최초 적합이고 협업 프레임의 스케줄링 기법이 최초 선출인 경우도 다른 기법에 비해 우수한 성능을 보여주었다.

4.3 멀티미디어 코덱 처리 성능 분석

최근 대부분의 멀티미디어 응용 시스템에서 사용되는 오디오 코덱인 MP3(MPEG1 Layer 3) 코덱을 구현하여 성능을 분석하였다. MP3 인코더와 디코더는 높은 압축률을 가지면서, 스트리밍 전송이 가능하기 때문에 PDA(Personal Digital Assistant)와 셀룰러폰, 그리고 멀티미디어 플레이어 같은 임베디드 시스템에서 필수적으로 탑재되는 소프트웨어이다. 펄스 부호 변조 기법으로 저장된 오디오 파일은 MP3 인코더를 거친 후에 최종 MP3 프레임으로 생성하게 된다. 펄스 부호 변조 데이터가 MP3 프레임으로 생성하는데 소요되는 시간을 측정하였다.

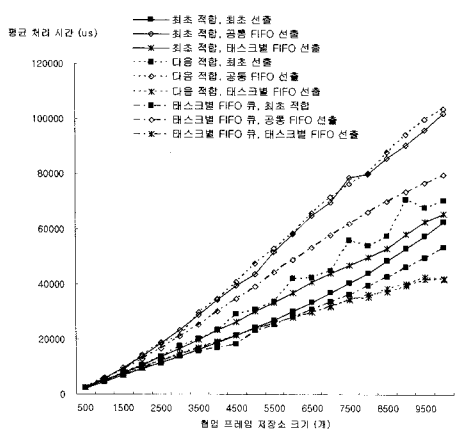


그림 12. 협업 프레임 저장소의 크기에 따른 처리 시간

MP3 인코더의 성능 분석에 사용된 태스크 모듈의 구성은 다음과 같다. 펄스 부호 변조 데이터 획득 모듈은 펄스 부호 변조 오디오 파일을 다른 모듈에게 전달하는 역할을 수행한다. 변형 이산 코사인 변환(Modified Discrete Cosine Transform) 모듈은 펄스 부호 변조 데이터들을 시간 도메인에서 주파수 도메인으로 변환시킨다. 심음향(Psychoacoustic) 처리 모듈은 펄스 부호 변조 데이터의 양자화기와 코딩 모듈을 컨트롤하는데 필요한 데이터들을 생성한다. 본 논문에서는 MPEG1 Layer 3의 오디오 압축 알고리즘 중 가장 유명하고 가장 음질이 좋은 LAME 기법을 따랐다[11-12]. 양자화 루프 모듈은 변형 이산 코사인 변환 모듈과 심음향 처리 모듈에서 생성된 데이터들을 바탕으로 압축된 데이터를 생성한다. 양자화 루프 모듈은 모든 인코딩 과정에서 가장 많은 시간이 소요되는 부분으로 알려져 있다[13]. 양자화 루프 모듈은 소프트웨어 혹은 하드웨어로 구현할 수 있다. 이에 본 논문에서는 양자화 루프 모듈을 소프트웨어 태스크로 먼저 구현하였고, 그리고 하드웨어 태스크로도 따로 구현하여 각각의 성능을 비교하였다. MP3 프레임 생성 모듈은 양자화 루프 모듈에서 생성된 데이터들에 대해서 MP3 헤더를 포함한 실제 비트스트림을 생성한다. 오디오 디바이스로부터 펄스 부호 변조 데이터를 얻어오는 과정이 지연되면 전체적인 성능에 영향을 미칠 수 있으므로 펄스 부호 변조 데이터를 얻어오는 태스크 #1의 우선순위가 가장 높으며, MP3 프레임을 생성하는 모듈의 우선순위는 태스크 #1보다 한 수준 낮은 우선순위로 설정하여 MP3 프레임을 빠르게 처리할 수 있도록 하였다. 그 외의 태스크 모듈은 모두 같은 우선순위로 설정하여 성능을 분석하였다. 성능 분석에 사용한 샘플 파일은 16bit 샘플 크기에 44KHz의 샘플링 속도를 가지는 101MB 크기의 WAV파일을 사용하였다.

그림 13은 다양한 인코딩 비율에 따라 샘플 파일을 인코딩하는데 소요되는 시간을 보여준다. 그림 13에서 보는 바와 같이 인코딩 비율이 증가함에 따라 하드웨어 코덱을 이용한 경우가 소프트웨어 코덱을 이용하는 경우보다 처리시간이 30%정도 감소됨을 보여 주었다.

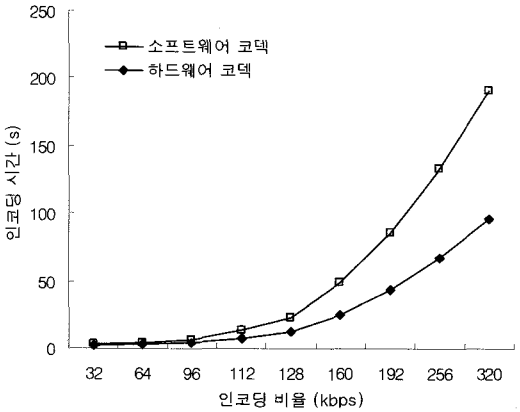


그림 13. MP3 인코더의 성능

오디오 코덱의 일부를 하드웨어 모듈로 구현하여 성능 분석을 하였음에도 성능이 향상됨을 확인하였다. 이에 더욱 복잡한 연산 과정이 필요한 멀티미디어 코덱이 하드웨어 모듈로 구현된다면 더욱더 우수한 성능이 생성될 것이다. 특히 본 논문에서 제안한 태스크기반의 하드웨어 및 소프트웨어간의 동적 협업 플랫폼은 시스템의 각 기능을 조립형 형태로 설계 및 구현할 수 있다. 이에 복잡한 구조를 가지는 멀티미디어 코덱의 일부를 먼저 하드웨어 태스크로 구현하고 나머지 복잡한 기능들을 점진적으로 개별 하드웨어 태스크로 구현 및 조립을 할 수 있기 때문에 전체 하드웨어 모듈의 점진적인 개발을 제공할 수 있다.

5. 결 론

본 논문에서는 소형 임베디드 시스템의 처리 성능 향상을 위하여 하드웨어 및 소프트웨어 모듈간의 동적 협업 플랫폼을 제안하고 성능을 분석하였다. 소형 임베디드 시스템은 PC 시스템에 비해 낮은 CPU의 성능을 가지고 있어 복잡한 연산 과정을 포함하고 있는 멀티태스킹 환경에 적용하기가 어렵다. 하지만 최적화된 프로세서 코어와 하드웨어 프로그래밍이 가능한 FPGA 기술은 하드웨어 및 소프트웨어 모듈간의 동적 협업을 지원하는 SoC 플랫폼의 설계에 활용되어 소형 임베디드 시스템에서 처리 성능의 향상을 제공할 수 있다. 동적 협업 플랫폼을 위한 세부 기술에는 하드웨어 및 소프트웨어 모듈간의 동적 협업을 위한 모듈 설계 기법과 모듈 간의 동적 협업을 지원하는 운영체제, 그리고 태스크 단위의 모듈화된 개발 및 각 모듈 간의 통신 및 동기화 기법이 있다. 이에 본 논문에서는 하드웨어 및 소프트웨어 모듈간의 동적 협업을 위한 세부 기술을 제공하는 동적 협업 플랫폼을 제안하였다. 성능 분석을 통해 각 세부 기술에 대한 시스템의 의존성을 분석하였고, 제안한 동적 협업 플랫폼의 우수성을 보여 주었다.

향후 연구로는 최근 출시된 고성능의 FPGA 칩셋에 적용된 하드웨어 기능을 이용하여 동적 협업 플랫폼의 성능을 향상시키는 방법에 대한 연구와 여러 개의 FPGA 칩이 장착된 고성능 시스템에서 동적 협업 플랫폼을 구성하는 방법에 대한 연구가 필요하다.

제안한 동적 협업 플랫폼을 이용하면, 소프트웨어 및 하드웨어 모듈의 재사용성을 보장하면서 조립화된 고속 소형 임베디드 시스템의 설계 및 개발을 가능하게 한다. 또한 SoC 기술을 이용하여 다양한 기능을 하나의 칩에 구현을 할 수 있기 때문에 저가격 및 고기능을 제공하는 칩셋 개발도 가능하다.

참 고 문 헌

[1] W.H. Wolf, "Hardware-software co-design of embedded systems," *Journal of the IEEE*, Vol.82, No.7, pp. 967-989, 1994.

[2] J. Lee, V.J. Mooney III, A. Daleby, K. Ingstrom, T. Klevin, and L. Lindh, "A comparison of the RTU hardware RTOS with a hardware/software RTOS," *Proceedings of Design Automation Conference*, pp. 683-688, 2003.

[3] V.J. Mooney III and D.M. Blough, "A hardware-softwre real-time operationg system framework for SoCs," *IEEE Design & Test of Computers*, Vol.19, No.6, pp. 44-51, 2002.

[4] M. Vetromille, L. Ost, C.A.M Marcon, C. Reif, and F. Hessel, "RTOS scheduler implementation in hardware and software for real time applications," *IEEE International Workshop on Rapid System Prototyping*, pp. 163-168, 2006.

[5] Y. Cho, S. Yoo, K. Choi, E. Zergainoh, and A. Jerraya, "Scheduler implementation in MPSoC

design," *Asia South Pacific Design Automation Conference*, pp. 151-156, 2005.

[6] N.R. Mehta, N. Medvidovic, and S. Phadke, "Towards a taxonomy of software connector," *Proceedings of international conference on Software engineering*, pp. 178-187, Limerick, Ireland, 2000.

[7] A. Rajawat, M. Balakrishnan, and A. Kumar, "Interface synthesis: issues and approaches," *International Conference on VLSI Design*, pp. 92-97, Calcutta, India, 2000.

[8] J. Furunas, J. Adomat, L. Lindh, J. Starner, and P. Voros, "A prototype for interprocess communication support, in hardware," *Proceedings of 9th Euromicro Workshop on eal-Time Systems*, pp. 18-24, 1997.

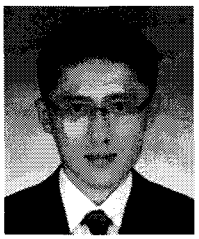
[9] Altera, *Excalibur Devices: Hardware Reference Manual*, Altera, San Jose, 2002.

[10] J.J. Labrosse, *uC/OS-II: The Real-Time Kernel*, CMPBooks, Kansas, 2002.

[11] ISO/IEC, "Information technology - coding of moving pictures and associated audio information: audio," ISO/IEC 11172-3, 1996.

[12] T. Dumitras and R. Marculescu, "On-chip stochastic communication," *Design, Automation and Test in Europe Conference and Exhibition*, pp. 790-795, 2003.

[13] C.Y. Lee, Y.C. Fang, H.C. Chuang, C.N. Wang, and T. Chiang, "A fast audio bit allocation technique based on a linear R-D model," *IEEE Transactions on Consumer Electronics*, Vol.48, Issue. 3, pp. 662-670, 2002.



이 동 건

2005년 2월 신라대학교 컴퓨터교육과 (학사)
2007년 2월 부산대학교 컴퓨터공학과 (석사)
2007년 3월~현재 부산대학교 컴퓨터공학과 박사과정
관심분야 : 무선 네트워크, QoS, SoC 설계, 실시간 시스템

김 영 만

2007년 2월 부산대학교 정보컴퓨터공학부 (학사)
2007년 3월~현재 부산대학교 컴퓨터공학과 석사과정
관심분야 : SoC설계, 실시간 시스템

탁 성 우

1995년 부산대학교 컴퓨터공학과 (학사)
1997년 부산대학교 컴퓨터공학과 (석사)
2003년 미국미주리주립대학교 Computer Science (박사)
2004년~현재 부산대학교 정보컴퓨터공학부 조교수
2004년~현재 부산대학교 컴퓨터 및 정보통신연구소 겸임 연구원
관심분야 : 유무선 네트워크, SoC 설계, 실시간 시스템, 위치인식, 최적화 기법, 그래프 이론, 큐잉 이론