Journal of Korea Multimedia Society Vol. 12, No. 5, May 2009(pp. 617-632)

SoC 플랫폼에서 태스크 기반의 조립형 재구성이 가능한 네트워크 프로토콜 스택에 관한 연구

김영만[†], 탁성우[††]

요 약

본 논문에서는 네트워크 프로토콜의 기능 명세를 소프트웨어 및 하드웨어 태스크로 분할한 후에 태스크 단위에서 조립형 재구성이 가능한 네트워크 프로토콜 스택의 설계 기법을 제안하였다. 또한 네트워크 기능을 사용하는 실시간 응용 서비스의 마감시한을 보장하기 위하여 개별 태스크의 마감시한을 보장함과 동시에 각 태스크 간에 교환되는 메시지의 마감시한을 보장하는 기법을 제안하였다. 제안한 기법은 네트워크 프로토콜의 기능을 태스크 단위로 분할한 후에 조립형 재구성이 가능한 소프트웨어 및 하드웨어 기반의 네트워크 프로토콜 태스크로 설계 및 구현할 수 있다. 또한 제안한 실시간 메시지 교환 기법은 마감시한 내에 메시지의 처리를 완료해야 하는 멀티미디어 응용 서비스의 실시간 속성을 만족시킬 수 있다. 본 논문에서는 TCP/IP 프로토콜을 태스크 단위로 분할하여 SoC (System-on-chip) 플랫폼에서 각각 하드웨어 및 소프트웨어 태스크로 구현한 후에 제안한 기법의 성능을 분석한 결과, 응용 서비스가 요구하는 실시간성 만족도를 제공함과 동시에 TCP/IP 프로토콜의 처리 성능도 향상되었음을 확인하였다.

A Study on Reconfigurable Network Protocol Stack using Task-based Component Design on a SoC Platform

Youngmann Kim[†], Sungwoo Tak[††]

ABSTRACT

In this paper we propose a technique of implementing the reconfigurable network protocol stack that allows for partitioning network protocol functions into software and hardware tasks on a SoC (System on Chip) platform. Additionally, we present a method that guarantees the deadline of both an individual task and messages exchanging among tasks in order to meet the deadline of real-time multimedia and networking services. The proposed real-time message exchange method guarantees the deadline of messages generated by multimedia services that are required to meet the real-time properties of multimedia applications. After implementing the networking functions of TCP/IP protocol suite into hardware and software tasks, we verify and validate their performance on the SoC platform. Experimental results indicate that the proposed technique improves the performance of TCP/IP protocol suit as well as application service satisfaction in application-specific real-time.

Key words: SoC(System-on-chip), Hardware-Software Co-design(하드웨어-소프트웨어 통합설계), Real Time Communication Channel between Hardware and Software Modules (하드웨어 및 소프트웨어 모듈간 실시간 통신 채널)

※ 교신저자(Corresponding Author) : 탁성우, 주소 : 부산시 금정구 장전동 산30번지(609-735), 전화 : 051)510-2387, FAX : 051)515-2208, E-mail : swtak@pusan.ac.kr
접수일 : 2009년 1월 15일, 완료일 : 2009년 3월 25일

[†] 정회원, 부산대학교 컴퓨터공학과 석박사통합과정
(E-mail : ssomai@gmail.com)
[††] 종신회원, 부산대학교 정보컴퓨터공학부 부교수

1. 서 론

고성능 프로세서 및 풍부한 하드웨어 자원을 이용하여 다양한 응용 서비스의 실행이 가능한 개인용 컴퓨팅 시스템과는 다르게, 임베디드 시스템은 낮은 사양의 하드웨어 자원을 가지고 있어 복잡한 처리과정이 요구되는 다양한 작업을 동시에 소프트웨어적으로 처리하는 것이 어렵다. 하지만 하드웨어 모듈은 동일한 클록에서도 프로세서보다 최대 100배의 성능을 제공할 수 있을 뿐만 아니라 프로세서와 병렬 수행이 가능하다는 장점을 가지고 있다 [1-2]. 이에 최근 임베디드 시스템의 설계 및 개발 경향은 임베디드 시스템에서 제공하려는 서비스를 SoC (System-on-Chip) 기법으로 통합하고 있다. 그리고 네트워킹 서비스의 성능을 향상시키기 위해서, 네트워크 프로토콜의 일부분 혹은 전체를 하드웨어 모듈로 구현하는 기법에 대한 많은 연구가 이루어지고 있다 [3-6].

임베디드 시스템에서 사용하는 네트워크 프로토콜은 임베디드 시스템의 목적이나 성능, 사용 환경에 따라 다양할 뿐만 아니라, 네트워크 프로토콜의 표준 명세에 대한 변경 또한 지속적으로 이루어지고 있다. 하지만 기존의 임베디드 SoC 시스템을 개발하는 회사 및 연구소에서는 하나의 단일 하드웨어 모듈로 구현하는 기법을 적용하고 있기 때문에, 네트워크 프로토콜 변경 시에 구현된 네트워킹 시스템 전체의 재설계 및 재개발이 요구된다. 이러한 문제점을 해결하기 위해서는 각 네트워크 프로토콜의 기능을 태스크 단위로 분할하여 조립형 재구성이 가능해야 한다. 그리고 멀티미디어 응용 서비스의 경우에는 마감시한 내에 패킷 전달이 완료되어야 하는 실시간 속성을 요구하기 때문에, 네트워킹 시스템은 이러한 실시간 속성을 고려해야 한다.

이에 본 논문에서는 임베디드 SoC에서 소프트웨어 및 하드웨어 모듈간 조립형 재구성이 가능한 네트워크 프로토콜 스택의 설계 기법을 제안한다. 제안한 기법은 소프트웨어 및 하드웨어 모듈간 조립형 재구성을 지원하기 위해서 소프트웨어 및 하드웨어 태스크간 공통 메시지 전달 기능을 제공하며, 태스크 간에 주고받는 메시지에 대한 공통 규격을 정의한다. 또한, 태스크 간에 주고받는 메시지에 대한 실시간 스케줄링을 제공하며, 멀티미디어 서비스와 같은 실시간 응용 서비스의 마감시한을 만족시킨다.

본 논문의 2장에서는 네트워크 프로토콜 스택을 하드웨어 모듈로 구현하는 기법과 관련된 기존 연구와 소프트웨어 및 하드웨어 모듈간 메시지 전달 기법에 대한 기존 연구, 그리고 실시간 스케줄링 기법에 대한 기존 연구를 기술하고, 3장에서는 개발 방법론과 함께 조립형 재구성이 가능한 네트워크 프로토콜 스택의 설계 기법을 설명하고, 4장에서는 제안한 설계 기법을 검증하기 위하여 TCP/IP 프로토콜을 태스크 단위로 분할한 후에 TCP/IP 프로토콜의 동작을 검증하였다. 마지막으로 5장에서 결론을 기술하였다.

2. 관련 연구

이 장에서는 먼저 네트워크 프로토콜 스택의 성능 향상을 위하여 네트워크 프로토콜의 일부나 전체 기능을 하드웨어 모듈로 구현한 기존 연구 사례를 소개한다. 그리고 기존 연구 사례의 문제점을 기술하고, 그 문제점을 해결하기 위하여 제안된 소프트웨어 및 하드웨어 통합 개발 기법에 대해서 기술한다. 그리고 기존의 소프트웨어 및 하드웨어 통합 개발 기법에서 적용되고 있는 소프트웨어 및 하드웨어 태스크간 메시지 전달기법에 대해서 기술한다.

최근 네트워크 프로토콜 스택의 성능을 향상시키기 위해서 네트워크 프로토콜의 일부 또는 전체를 하드웨어 모듈로 구현하고자 한 논문은 다음과 같다. Wu와 Chen [5]은 FPGA (Field-Programmable Gate Array) 기법을 이용하여 TCP/IP 프로토콜 중 IP와 ICMP, ARP 그리고 MAC을 각각 하드웨어 모듈로 구현하였으며, 소프트웨어로만 이루어진 네트워킹 시스템보다 1.5배에서 1.6배만큼 대역폭을 향상시켰다. 그러나 IP나 ICMP, ARP, MAC 중 하나라도 다른 프로토콜을 변환하려고 하면 하드웨어 모듈간 통신 인터페이스를 재설계해야 하기 때문에, 조립형 재구성이 제공되지 않는다. Chung 등 [6]은 저사양 프로세서에서 CPU 이용률을 낮추기 위해 TCP/IP 프로토콜의 전체를 하드웨어 모듈로 구현하였다. 대역폭은 평균 105%만큼 성능이 향상시켰으며, CPU 이용률을 최대 3/4까지 감소시켰다. 그러나 TCP/IP 스택의 기능을 단일 하드웨어 모듈로 구현하였기 때문에, TCP 또는 IP의 기능 변경이나 다른 프로토콜을 추가하기가 어렵다.

소프트웨어 및 하드웨어 시스템 통합 설계에서 소프트웨어 및 하드웨어 태스크간 통신 인터페이스를 설계하는 방법에는 두 가지가 있다. 소프트웨어 및 하드웨어 태스크간 관계와 기능 명세에 따라 통신 인터페이스를 설계하는 방법 [5-6]과 소프트웨어 및 하드웨어 태스크가 공통으로 사용할 수 있는 통신 채널을 라이브러리처럼 설계하는 방법 [7-8]이 있다. Srivastava와 Brodersen [7], 그리고 Fischer 등 [8]은 소프트웨어 및 하드웨어 태스크간 조립형 재구성을 제공하기 위하여 세마포어와 인터럽트 핸들러를 이용한 메시지 큐 기반 소프트웨어 및 하드웨어 모듈간 통신 채널을 제안하였다. 이 통신 채널을 기반으로 Srivastava와 Brodersen [7]은 로봇 제어 시스템을 개발하였고, Fischer 등 [8]은 CAN (Controller Area Network) 모니터 시스템을 개발하였으며, 개발 기간의 단축에도 도움이 되었음을 보여주었다. 그러나 기존의 시스템은 모듈간의 상호 의존성이 존재하기 때문에, 모듈의 독립성을 보장하지 못한다. 또한 기존의 메시지 큐 기반 소프트웨어 및 하드웨어 모듈간 통신 채널은 멀티미디어 응용 서비스에서 요구하는 마감시한과 같은 실시간 속성을 고려하지 않기 때문에 QoS (Quality of Service)를 제공해야 하는 네트워크 프로토콜 스택의 개발에는 적합하지 않다. 그러므로 기능별 독립성을 위한 태스크 분할 기법과 함께 메시지 큐 기반 통신 채널에서 실시간 응용 서비스의 속성을 고려한 실시간 스케줄링 기법이 필요하다.

소프트웨어 및 하드웨어 태스크간 메시지 큐 기반 통신 채널에서 실시간 스케줄링을 고려한 기존 연구를 살펴보면 다음과 같다. Eisenring과 Platzner [9]의 통신 채널인 HASIS (Hardware Software Interface Synthesis)는 Srivastava와 Brodersen [7], 그리고 Fischer 등 [8]과 같이 FIFO (First-In First-Out)방식으로 전달한다. Gopalakrishnan 등 [10]의 iSLIP (iterative SLIP)은 FIFO방식을 기반으로 하며, 기아(starvation) 현상을 피할 수 있도록 공정성을 함께 고려한 메시지 스케줄링을 지원한다. Yiming과 Eisaka [11]의 메시지 스케줄링 채널인 RTCC (Real Time Communication Control)는 EDF (Earliest Deadline First)기반의 메시지 스케줄링 기법을 제안하였다. 하지만 Eisenring과 Platzner [9], Gopalakrishnan 등 [10], 그리고 Yiming과 Eisaka [11]은 메시지의 실시간 속성에 대해서만 고려하고 메시지를 처리해주는 태스크의 실시간 속성에 대해서는 고려하지 않기 때문에 태스크가 메시지의 마감시한보다 늦게 메시지를 처리해줄 가능성이 있다는 공통적인 문제점이 있다.

본 논문에서는 실시간성과 조립형으로 재구성이 가능한 네트워크 프로토콜 스택의 설계 기법을 위해서 다음과 같은 기능을 제안하였다. 네트워크 프로토콜 스택의 기능별 독립성을 위한 태스크 분할, 소프트웨어 및 하드웨어 태스크간 메시지 큐 기반의 통신 채널, 실시간 속성값을 포함한 메시지 공통 규격, 그리고 태스크 및 메시지의 실시간 속성을 고려하는 실시간 스케줄링을 제안하였다.

3. 조립형 재구성이 가능한 네트워크 프로토콜 스택의 설계

3.1 SoC 플랫폼에서 네트워크 프로토콜 스택의 개발 과정

이 장에서는 먼저 네트워크 프로토콜 스택의 성능 향상을 위하여 네트워크 프로토콜의 기능을 하드웨어 모듈로 구현하는 SoC 설계 기법을 비교 분석하였다. 그림 1은 SoC 플랫폼에서 네트워크 프로토콜 스택의 개발과정을 보여준다.

그림 1-(a)에서 보는 바와 같이 일반적인 SoC 플랫폼에서 네트워크 프로토콜 스택의 개발 과정은 주어진 네트워크 프로토콜의 기능 명세를 분석하고, 하드웨어(H/W) 모듈과 소프트웨어(S/W) 모듈로 명확히 구분한 후에 각 모듈간의 세부 기능 및 각 모듈별

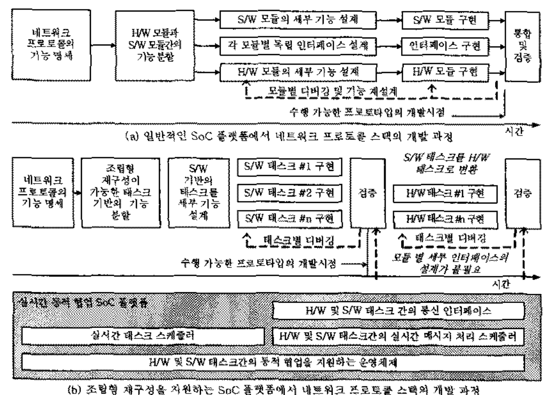


그림 1. SoC 플랫폼에서 네트워크 프로토콜 스택의 개발 과정

인터페이스의 설계 및 구현을 진행한다. 수행 가능한 프로토타입의 개발 시점은 하드웨어 및 소프트웨어, 그리고 인터페이스의 개발이 완료된 후에 최종적으로 통합하는 과정에서 확인 가능하다. 그러나 개발 과정 중에서 하드웨어 및 소프트웨어 모듈에 새로운 기능의 추가나 변경이 요구되는 경우 각 모듈 간 인터페이스의 변경도 발생될 수 있기 때문에 결국 네트워크 프로토콜 스택에 대한 재설계 과정이 필요하게 된다. 또한 하드웨어 모듈의 개발은 사이클 단위의 검증이 필요하기 때문에 많은 시간과 비용이 소요된다. 그리고 하드웨어 모듈의 개발이 지연됨에 따라 이에 의존적인 소프트웨어 모듈의 개발 시간이 길어질 수 있다. 이러한 개발 과정으로는 계속해서 수정되는 네트워크 프로토콜의 표준화 명세에 따른 SoC 기반의 네트워크 프로토콜 스택을 적시에 개발하기가 어렵다.

본 논문에서 제안하는 방식은 기능 명세의 수정이 계속해서 요구되는 네트워크 프로토콜의 표준을 적시에 개발 및 제공할 수 있도록 조립형 재구성이 가능한 SoC 플랫폼이다. 그림 1-(b)는 조립형 재구성을 지원하는 SoC 플랫폼에서 네트워크 프로토콜 스택의 개발 과정을 보여준다. 이러한 개발과정에서는 먼저 네트워크 프로토콜의 명세를 소프트웨어 태스크로 구현하여 네트워크 프로토콜 스택을 검증한 후에 점진적으로 성능 향상이 요구되는 각 소프트웨어 태스크를 하드웨어 태스크로 개선하여 개발할 수 있다. 소프트웨어 태스크 수준에서 네트워크 프로토콜을 개발하여 검증하는 단계에서는 하드웨어 모듈이 없기 때문에 사이클 단위의 하드웨어 검증과 같은 오버헤드가 없다. 또한 개발된 소프트웨어 태스크와 하드웨어 태스크간의 논리적인 구조와 태스크간의 메시지를 주고받는 통신 인터페이스가 동일하기 때문에 태스크간 자유로운 소프트웨어 및 하드웨어 변환이 가능하다. 또한 새로운 프로토콜의 명세가 추가되어도 기본 설계 구조의 변경 없이 해당 프로토콜의 부가적인 태스크를 구현하면 되는 장점이 있다.

이와 같이 조립형 재구성이 가능한 네트워크 프로토콜 스택의 구조를 유지함과 동시에 멀티미디어 응용 서비스의 실시간 속성을 만족시키기 위하여 필요한 4가지 기법은 다음과 같다. 첫째, 네트워크 프로토콜의 기능을 태스크 단위로 분할하여 조립형 재구성이 가능한 소프트웨어 및 하드웨어 태스크로 구분하여 구현을 하는 것이다. 둘째, 하드웨어 및 소프트웨어 태스크간의 메시지 교환을 제공할 수 있는 메시지큐 기반의 통신 채널 기법이 필요하다. 셋째, 메시지큐 기반의 통신 채널을 통하여 하드웨어 및 소프트웨어 태스크간에 교환되는 메시지의 공통 규격이 필요하다. 마지막으로 메시지 교환 기능을 실시간으로 제공하는 실시간 메시지 스케줄러를 제공하여 멀티미디어 응용 서비스의 실시간 속성을 만족시키는 것이다. 본 논문에서는 이러한 4가지 기법을 초기 설계 단계 과정에서 적용하기 때문에, 소프트웨어 태스크와 하드웨어 태스크를 독립적으로 구현하더라도 통합 및 검증 과정에 필요한 부가적인 오버헤드가 없다.

3.2 네트워크 프로토콜 스택의 조립형 재구성을 지원하는 SoC 플랫폼

그림 2는 네트워크 프로토콜 스택의 조립형 재구성을 지원하는 SoC 플랫폼의 구조와 할당된 메모리 주소 영역을 보여준다. 그림 2에서 메모리 주소 영역의 시작점은 0×0000(상위 16비트)_0000(하위 16비트)이다.

SoC 플랫폼의 하드웨어 구성도를 살펴보면 다음과 같다. SoC 칩셋으로 EPXA4 칩셋을 사용하였다 [12]. EPXA4는 ARM922T 프로세서 코어와 함께 부가적인 하드웨어 모듈을 구현할 수 있는 PLD (Programmable Logic Device) 개발 환경을 제공한다. 네트워크 프로토콜의 기능을 소프트웨어 및 하드웨어 태스크 단위로 설계한 후에 소프트웨어 태스크는 외부 SDRAM 기반의 메모리에서 실행되며, 하드웨어 태스크는 PLD에서 구현 및 실행된다. 하드웨어 태스크에서 처리된 메시지는 PLD 내에서 구현된 레지스터에 저장되고, 레지스터에 저장된 메시지는 PLD 브릿지와 AHB1-to-AHB2 (AMBA High-performance Bus) 버스 브릿지, 그리고 메모리 제어부(SDRAM Controller)를 거쳐 SDRAM에서 실행되고 있는 소프트웨어 태스크에게 전달되며, 이에 대한 역방향으로도 메시지가 교환된다. 하드웨어 태스크와 소프트웨어 태스크로 구현된 네트워크 프로토콜에서 생성되는 데이터는 EBI (Expansion Bus Interface)에 물리적으로 연결되어 있는 이더넷 칩셋인 LAN91C111을 통해 전송된다 [13]. 그림 2를 구성하고 있는 나머지 모듈 및 기능에 대해서는 다음 절에서 다루고 있다.

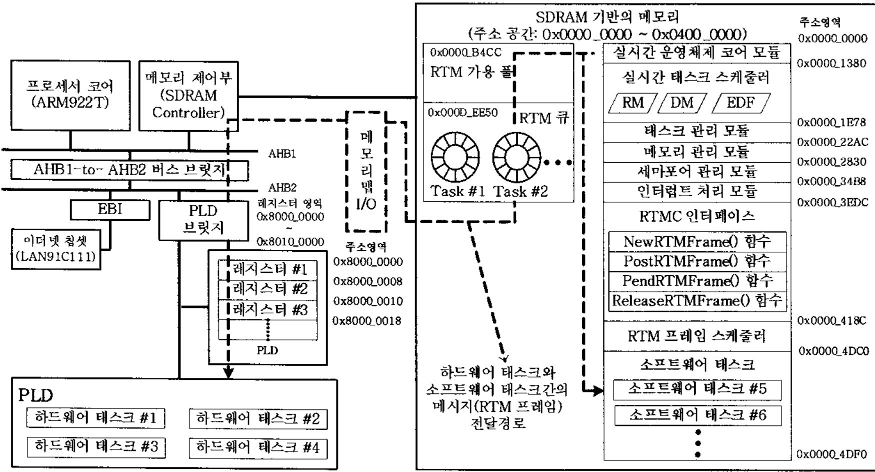


그림 2. SoC 플랫폼의 구조

3.3 네트워크 프로토콜의 조립형 재구성을 지원하기 위한 기능별 구성

먼저, 네트워크 프로토콜의 기능을 태스크 단위로 분할하는 기법에 대해서 살펴보면 다음과 같다. 태스크는 완전히 독립된 수행이 가능하도록 명령어 실행 코드 및 실행에 필요한 데이터, 그리고 태스크의 상태 정보를 저장할 수 있는 공간으로 구성된 기능 단위이다. 소프트웨어 태스크인 경우에는 명령어 실행 코드는 텍스트(TEXT) 영역에 저장되며, 실행에 필요한 정보는 데이터(DATA) 영역에 저장된다. 그리고 멀티태스킹 운영체제 환경에서는 개별 태스크의 상태 정보가 스택(STACK) 영역에 저장된다. 하드웨어 태스크인 경우에는 그림 2에서 보는 바와 같이 명령어 실행코드 및 실행에 필요한 데이터는 산술논리 연산 코어를 담당하는 하드웨어 모듈로 설계되어 PLD 영역에 구현된다. 하드웨어 태스크는 다른 하드웨어 태스크와 독립적으로 병렬 실행이 가능하기 때문에 개별 하드웨어 태스크의 상태 정보를 저장하는 스택 영역이 필요 없다.

소프트웨어 및 하드웨어 태스크간에 메시지 교환을 담당하는 메시지 큐 기반의 통신 채널 기법을 구성하는 요소로는 PLD 영역의 레지스터와 메모리 맵 I/O 기능, RTM (Real-Time Message) 큐, RTM 가용 풀, 그리고 RTMC (Real-Time Message Channel) 인터페이스 함수가 있다. 하드웨어 태스크와 소프트웨어 태스크간의 메시지 교환 단위는 RTM 프레임이다. 메시지 큐 기반의 통신 채널에서

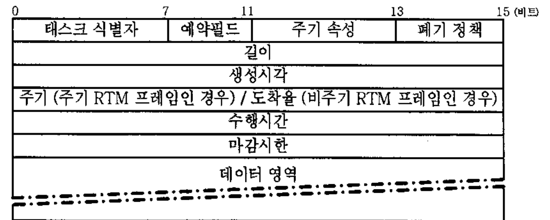


그림 3. RTM 프레임의 구조

교환되는 RTM 프레임의 규격은 그림 3과 같다.

태스크 식별자는 8비트로 구성되며, RTM 프레임을 처리하고 있는 태스크의 식별자를 저장한다. 2비트의 주기 속성 필드는 RTM 프레임이 주기적으로 생성되는 것인지 아니면 혹은 비주기적으로 생성되는 것인지를 구별하는 값이 저장된다. RTM 프레임이 주기 속성을 가지는 경우에는 7번째 필드에 RTM 프레임의 생성 주기 값이 저장된다. RTM 프레임이 비주기 속성을 가지는 경우에는 7번째 필드에 RTM 프레임의 도착률 값이 저장된다. 폐기 정책(Drop Policy) 필드는 RTM 프레임의 마감시한(Deadline)이 초과되는 경우에 RTM 프레임의 폐기 여부를 지정한다. RTM 프레임의 생성시각(Release Time)은 6번째 필드에 저장된다. 8번째 필드인 수행시간(Execution Time) 필드에는 RTM 프레임의 처리시간이 저장된다. 9번째 필드인 마감시한 필드에는 RTM 프레임의 처리 마감시한을 지정한다. 그리고 10번째 필드인 데이터 영역에는 목적지 태스크에게 전달하는 데이터를 저장한다.

하드웨어 및 소프트웨어 태스크는 RTMC 인터페이스 함수를 사용하여 SDRAM 메모리 영역 내에 상주하고 있는 RTM 가용 풀과 RTM 큐를 관리한다. RTM 큐는 모든 하드웨어 및 소프트웨어 태스크에게 할당된다. RTM 큐는 할당 받은 RTM 프레임의 주소를 보관하는 원형 대기열(Circular Queue)로 구현하였다. RTMC 인터페이스 함수의 사용 예를 살펴보면 다음과 같다. 예를 들어 소프트웨어 태스크 #5는 먼저 int *NewRTMFrame (int TID, int RTMFrameSize) 함수를 사용하여 RTM 가용 풀에서 RTMFrameSize 만큼의 새로운 RTM 프레임 A를 요구하여 할당된 RTM 프레임 A의 주소를 반환 받는다. TID는 요구한 태스크의 식별자 번호를 지정한다. 소프트웨어 태스크 #5는 할당 받은 RTM 프레임 A에 정보를 담아 하드웨어 태스크 #1이 관리하는 RTM 큐에 전달하기 위해서 int PostRTMFrame (int TID, int *RTMFrameAddr)을 호출한다. TID는 수신 태스크의 식별자 번호를 지정하며, RTMFrameAddr에는 전달하고자 하는 RTM 프레임 A의 주소를 지정한다. PostRTMFrame() 함수의 동작이 실패하면 -1을 리턴한다. 하드웨어 태스크 #1은 소프트웨어 태스크 #5가 PostRTMFrame() 함수를 통해 자신의 RTM 큐에 전달된 RTM 프레임 A를 수신하기 위하여 int PendRTMFrame (int TID)를 호출한다. TID는 송신 태스크의 식별자 번호를 지정하며, 이 예에서는 소프트웨어 태스크 #5의 식별자 번호에 해당된다. 본 논문에서 제안한 RTM 프레임 스케줄러인 RTP-RTM (Real-time Task Properties inherited to RTM frame)은 소프트웨어 태스크 #5에서 전달한 RTM 프레임 A의 주소를 하드웨어 태스크 #1에서 사용하는 레지스터 #1에 저장하고, 하드웨어 태스크 #1에게 이러한 사실을 알려준다. 하드웨어 태스크 #1은 레지스터 #1에 저장된 RTM 프레임 A의 주소를 획득한 후에 PLD 브릿지와 AHB1-to-AHB2 버스 브릿지, 그리고 메모리 제어부를 거쳐 SDRAM 기반의 메모리에 저장되어 있는 RTM 프레임을 가져온다. 그리고 RTM 프레임 A를 처리한 후에 int ReleaseRTMFrame (int TID, int *RTMFrameAddr)을 호출하여 RTM 프레임 A를 RTM 가용 풀에 반환한다. 여기에서 TID는 하드웨어 태스크 #1의 식별자이며, RTMFrameAddr에는 가용 풀에 반환하는 RTM 프레임 A의 주소를 지정한다.

그림 2의 SoC 플랫폼에서 보는 바와 같이 ARM922T 프로세서 코어는 PLD내에서 구현된 메모리 영역에 대해서도 SDRAM 기반의 메모리 공간과 동일한 메모리 맵 I/O을 사용할 수 있다. 본 논문에서는 하드웨어 태스크에서 사용되는 레지스터 영역의 시작 주소로 0×8000_0000을 할당하였다.

3.4 RTMC 인터페이스 기법

RTM 프레임의 교환과정은 소프트웨어 태스크와 소프트웨어 태스크, 소프트웨어 태스크와 하드웨어 태스크, 그리고 하드웨어 태스크와 하드웨어 태스크 간에 이루어질 수 있다. 먼저 RTMC 인터페이스 함수를 사용하여 하드웨어 태스크와 소프트웨어 태스크간에 RTM 프레임이 교환되는 세부과정은 그림 4와 같다.

하드웨어 태스크는 AHB2 버스에 연결되어 있는 PLD 영역에서 동작하며, 그림 5에서 보여준 RTMC 제어 메시지를 메모리 주소 0×8000_0000부터 0×8000_XXXX까지 맵핑되어 있는 레지스터에 저장한다. 그리고 동기 I/O 방식인 폴링방식 보다 효율적인 비동기 I/O 방식인 인터럽트 핸들러를 통해 레지스터에 저장된 RTMC 제어 메시지를 RTMC-HST (Real Time Message Channel among Hardware/Software Tasks) 모듈에 전달한다. RTMC-HST 모듈은 RTM 제어 메시지의 종류에 따라 NewRTMFrame(), PostRTMFrame(), PendRTMFrame(), 그리고 ReleaseRTMFrame()에 해당되는 기능을 수행한 후에 그 결과를 인터럽트 핸들러를 통하여 다시 PLD 영역에 있는 레지스터에 저장한다. RTMC-HST 모듈은 RTM 프레임 스케줄러, RTM 가용 풀, 소프트웨어 태스크와 하드웨어 태스크용 RTM 큐, RTMC 인터페이스 함수로 구성된다.

예를 들어 하드웨어 태스크가 NewRTMFrame() 함수를 호출한 경우에 그림 4에서 보여준 RTMC 인터페이스의 동작과정을 살펴보면 다음과 같다. 먼저 하드웨어 태스크는 새로운 RTM 프레임의 할당을 요청하기 위해 NewRTMFrame 제어 메시지를 PLD 영역에 있는 레지스터 #1 (그림 2의 0×8000_0000번지에 위치함)에 쓴다 (동작과정 (1)). NewRTMFrame 제어 메시지에는 NewRTMFrame() 함수의 호출을 의미하는 0×01이 저장되고 요청한 RTM 프레임의 길이와 새로운 RTM 프레임을 요청한 하드웨어 태스

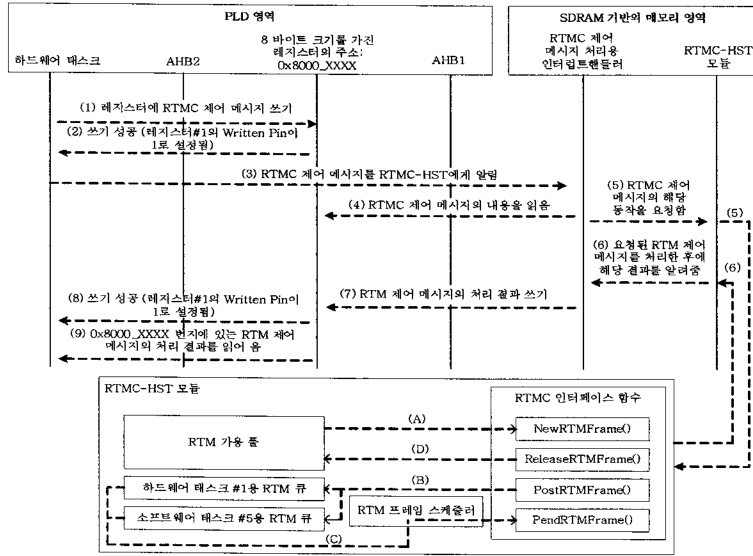


그림 4. RTMC 인터페이스의 동작과정

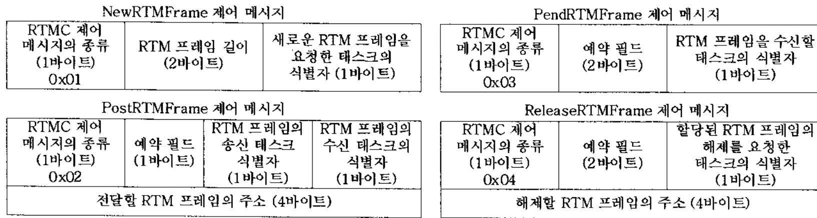


그림 5. RTMC 제어 메시지의 종류

크 #1의 식별자가 저장된다. 동작과정 (1)이 성공적으로 수행되면 레지스터 #1의 Written 핀이 1로 설정되며 (동작과정 (2)),이를 확인한 하드웨어 태스크 #1은 인터럽트를 발생시켜 RTMC-HST 모듈에게 NewRTMFrame 요청 메시지를 레지스터 #1에 기록하였다는 사실을 알린다 (동작과정 (3)). NewRTMFrame 메시지 처리용 인터럽트 핸들러는 레지스터 #1에 기록된 정보를 읽은 후에 RTMC-HST 모듈에 새로운 RTM 프레임의 할당을 요청한다 (동작과정 (4)와 (5)). RTMC-HST 모듈 내의 NewRTMFrame() 함수가 실행을 하여 RTM 가용 풀에서 새롭게 할당한 RTM 프레임의 주소 정보를 NewRTMFrame 메시지 처리용 인터럽트 핸들러에게 알려주고, 인터럽트 핸들러는 RTM 프레임의 주소 정보를 레지스터 #1에 저장한다 (동작과정 (A) 및 (6)과 (7)). 레지스터 #1에 RTM 프레임의 주소 정보가 저장되면, 레지스터 #1의 Written 핀이 1로 설정되고 (동작과정 (8)) 이를 확인한 하드웨어 태스크 #1은 레지스터 #1에 저장되어 있는 RTM 프레임의 주소 정보를 읽어 온다. SDRAM 기반의 메모리 영역에서 동작하고 있는 소프트웨어 태스크가 NewRTMFrame() 함수를 호출하여 새로운 RTM 프레임을 할당 받는 과정에서는 PLD 영역 내의 레지스터와 인터럽트 핸들러의 사용이 필요하지 않다. 소프트웨어 태스크에서 호출되는 NewRTMFrame() 함수는 기존의 소프트웨어 프로그래밍 방식과 동일하게 SDRAM 기반의 메모리 영역에 상주하고 있는 RTMC-HST 모듈의 함수들을 직접 제어하여 새로운 RTM 프레임을 할당 받는다.

한편, 하드웨어 태스크 #1이 소프트웨어 태스크 #5에게 RTM 프레임을 전달하기 위하여 호출하는 PostRTMFrame() 함수의 경우는 앞서 기술한 NewRTMFrame() 함수의 동작과정과 거의 유사하며 차이점만 살펴보면 다음과 같다. 먼저,

PostRTMFrame 제어 메시지가 PLD 영역에 있는 레지스터 #1에 저장된다 (동작과정 (1)). 그림 5에서 보는 바와 같이 레지스터에 저장되는 PostRTMFrame 제어 메시지의 내용은 PostRTMFrame 제어 메시지를 나타내는 값 (0×02)과 RTM 프레임을 전달하는 송신 태스크의 식별자 (즉, 하드웨어 태스크 #1의 식별자), 그리고 RTM 프레임을 받는 수신 태스크의 식별자 (즉, 소프트웨어 태스크 #5의 식별자) 및 전달할 RTM 프레임의 주소로 구성된다. RTMC-HST 모듈 내의 PostRTMFrame() 함수는 인터럽트 핸들러를 실행시켜 레지스터 #1에 저장되어 있는 RTM 프레임의 주소를 소프트웨어 태스크 #5에서 관리하는 RTM 큐에 저장한다 (동작과정 (B) 및 (5)와 (6)). 이 과정이 성공적으로 수행되면, 인터럽트 핸들러는 레지스터 #1에 NULL값을 입력하여 PostRTMFrame() 함수의 동작이 완료되었음을 하드웨어 태스크 #1에게 알린다 (동작과정 (7)과 (8)).

하드웨어 태스크 #1이 소프트웨어 태스크 #5로부터 RTM 프레임을 수신하기 위하여 PendRTMFrame() 함수를 호출한 경우, PendRTMFrame 제어 메시지를 나타내는 값 (0×03)과 RTM 프레임을 받는 수신 태스크의 식별자 (즉, 하드웨어 태스크 #1의 식별자)로 구성된 PendRTMFrame 제어 메시지를 사용한다. RTMC-HST 모듈 내의 PendRTMFrame() 함수는 RTM 프레임 스케줄러를 실행시킨다 (동작과정 (C)와 (5)). RTM 프레임 스케줄러는 레지스터 #1에 저장되어 있는 하드웨어 태스크 #1의 식별자 정보를 사용하여 하드웨어 태스크 #1에서 관리하는 RTM 큐에 접근한 후 한 개의 RTM 프레임을 선출한다 (동작과정 (C)). 그리고 선출된 해당 RTM 프레임의 주소 정보를 하드웨어 태스크 #1에게 다시 되돌려 준다 (동작과정 (6)부터 (8)까지).

하드웨어 태스크 #1이 할당 받은 RTM 프레임을 해제하기 위하여 ReleaseRTMFrame() 함수를 호출한 경우, ReleaseRTMFrame 제어 메시지를 나타내는 값 (0×04)과 RTM 프레임을 할당하는 태스크의 식별자 (즉, 하드웨어 태스크 #1의 식별자)와 할당하고자 하는 RTM 프레임의 주소로 구성된 ReleaseRTMFrame 제어 메시지를 사용한다. RTMC-HST 모듈 내의 ReleaseRTMFrame() 함수는 해당 RTM 프레임을 RTM 가용 풀에 반환한다 (동작과정 (D)). 이 과정이 성공적으로 수행되면, 인터럽트 핸들러는 레지스터 #1에 NULL값을 입력하여 ReleaseRTMFrame() 함수의 동작이 완료되었음을 하드웨어 태스크 #1에게 알린다 (동작과정 (7)과 (8)).

한편, 소프트웨어 태스크간의 RTM 교환 과정을 처리하기 위해서는 PLD 영역 내의 레지스터와 인터럽트 핸들러의 간섭 없이 RTMC-HST 모듈의 함수들을 직접 제어하여 송신 소프트웨어 태스크는 수신 소프트웨어 태스크에게 RTM 프레임을 전달하면 된다.

3.5 RTM 프레임 스케줄링 기법

본 논문에서는 실시간 응용 서비스의 실시간성을 만족시키기 위하여 두 가지 형태의 실시간 스케줄링 기법을 제공한다. 첫째, 분할된 소프트웨어 태스크를 실시간으로 스케줄링 하는 태스크 기반의 대단위(Coarse-Grained) 실시간 스케줄링 기법이다. 본 논문에서는 실시간 스케줄링 기법에서 많이 활용되며, 동적인 우선순위 할당 기법을 제공함과 동시에 응용 서비스의 대표적인 실시간 속성인 마감시한을 고려한 EDF 스케줄링 기법의 개념을 이용한다 [14]. 둘째, 소프트웨어 태스크와 하드웨어 태스크간의 실시간 메시지 교환을 제공하기 위해서는 RTM 큐에 저장된 RTM 프레임의 실시간 속성을 만족시키는 RTM 프레임 기반의 소단위(Fine-Grained) 실시간 메시지 스케줄링 기법이 필요하다. 본 논문에서는 실시간 RTM 프레임 스케줄링 기법으로 RTP-RTM 스케줄러를 제안하였다. 특히 PLD영역에 구현된 하드웨어 태스크는 소프트웨어 태스크와는 독립적으로 동작한다. 따라서 하드웨어 태스크는 RTP-RTM 스케줄러를 사용하여 소프트웨어 혹은 다른 하드웨어 태스크에게 실시간으로 RTM 프레임을 전달 및 처리하여 응용 서비스가 요구하는 실시간 속성을 만족시키고자 한다. 표 1은 2장의 관련연구에서 살펴본 기존의 메시지 스케줄링 기법을 RTM 프레임 스케줄링에 적용할 때 요구되는 RTM 프레임의 선출 기준을 기술하였다.

RTM 프레임 스케줄러는 태스크 스케줄러간의 협업 관계에 대한 분석이 필요하다. 앞서 살펴본 바와 같이 태스크 스케줄링을 위하여 EDF 스케줄러를 사용한다. 식 (1)의 프로세서 이용률에 대한 필요충분 조건이 만족되면, 시각 0에서 동시에 생성되며 주기값과 동일한 마감시한을 가지는 주기 태스크들은 EDF 스케줄러에 의하여 스케줄링이 가능하다 [14].

표 1. RTM 프레임 스케줄러와 RTM 프레임의 선출 기준

RTM 프레임 스케줄러	RTM 프레임의 선정 기준
HASIS-FIFO	가장 일찍 생성된 RTM 프레임을 RTM 큐에서 우선적으로 선출한다.
iSLIP-WFQ	[단계 1: RTM 프레임의 가중치 설정] 주기 RTM 프레임인 경우에는 RTM 프레임 내의 주기 필드 값에 반비례하여 가중치를 주며, 비주기 RTM 프레임인 경우에는 RTM 프레임 내의 도착률에 비례하여 가중치를 준다. [단계 2: RTM 프레임의 선출 기준] RTM 프레임의 가중치에 비례하는 확률에 따라 RTM 프레임을 선출한다.
RTCC-EDF	RTM 프레임의 절대적 마감시한(생성시간 + 마감시한)이 가장 이른 RTM 프레임을 우선적으로 선출한다.

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{1}$$

식 (1)에서 n은 전체 주기 태스크의 수를 나타내며, i는 개별 주기 태스크를 의미한다. C_i는 주기 태스크의 수행시간을 의미하여 T_i는 주기 태스크의 주기를 의미한다. 그리고 태스크의 집합이 주기와 비주기로 구성되는 경우, 식 (2)의 프로세서 이용률에 대한 충분조건이 만족되면, EDF 스케줄링 기법에 의하여 시각 0에서 동시에 생성된 태스크 집합은 스케줄링이 가능하다 [14].

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1 \tag{2}$$

식 (2)에서 D_i는 주기 혹은 비주기 태스크의 절대적 마감시간을 의미한다. 식 (1)과 식 (2)의 개념을 본 논문에서의 태스크 스케줄러와 RTM 프레임 스케줄러에 적용시키면 다음과 같다. 하드웨어 태스크는 소프트웨어 태스크를 처리하는 SoC 플랫폼의 ARM922T 프로세서와 독립적으로 실행되면서 RTM 프레임을 처리하기 때문에 하드웨어 태스크에게 전달되는 RTM 프레임의 하드웨어 태스크 이용률이 1을 넘지 않으면, RTM 프레임의 마감시한을 만족시켜 RTM 프레임에게 실시간성을 제공할 수 있다.

한편, 소프트웨어 태스크는 SoC 플랫폼의 ARM922T 프로세서에서 처리되므로, 모든 소프트웨어 태스크의 프로세서 이용률이 다양한 태스크 스케줄러에서 실시간성을 보장할 수 있는 최대 프로세서 이용률을 넘지 않는 한, 소프트웨어 태스크의 실시간성 만족은 태스크 스케줄러에 의해서 보장된다. 태스크 스케줄러가 EDF이면 식 (1)과 식 (2)의 프로세서 이용률 조건을 만족하는 경우가 된다. RTM 프레임은 RTM 스케줄러에 의해서 선출되지만, RTM 프레임의 처리는 태스크에서 처리된다. 따라서 RTM 프레임의 실시간성 만족을 제공하기 위해서는 RTM 프레임의 프로세서 이용률이 RTM 프레임을 처리하는 태스크의 프로세서의 이용률보다 낮아야 RTM 프레임이 마감시한 내에 처리될 수 있다.

표 2는 표 1에서 살펴본 RTM 프레임 스케줄러의 실시간성 만족을 분석하는데 사용되는 태스크의 속성을 보여준다.

그림 6은 표 2에서 살펴본 RTM 프레임 스케줄러의 실시간성 만족을 분석하는데 사용되는 태스크의 속성을 보여준다. 태스크 스케줄러로는 EDF를 사용하였고, RTM 프레임의 스케줄러는 마감시한을 고려한 RTCC-EDF를 사용하였다. 먼저 $Task_2$가 시각 0에서 실행을 시작한다. 이때의 $Task_2$의 프로세서 이용률을 식 (2)를 이용하여 구하면 0.5가 된다. 따라

표 2. RTM 프레임 스케줄러의 실시간성 만족 분석을 위한 시나리오 예

태스크	시작시간	수행시간	마감시한	RTM	송신태스크	수신태스크	전달요청시간	수행시간	마감시한
$Task_1$	20	12.5	25	$RTMFrame_{2,3}$	$Task_2$	$Task_3$	14	10	15
$Task_2$	0	15	30	$RTMFrame_{4,3}$	$Task_4$	$Task_3$	*(25, 30)	10	45
$Task_3$	14, *(25, 30)	10	35	*로 표시된 값은 RTM 프레임의 스케줄러에 따라 결정된다. 30은 RTP-RTM 스케줄러를 적용하는 경우이며, 25는 RTP-RTM을 제외한 나머지 RTM 프레임 스케줄러를 적용한 경우이다.					
$Task_4$	20	7.5	20						

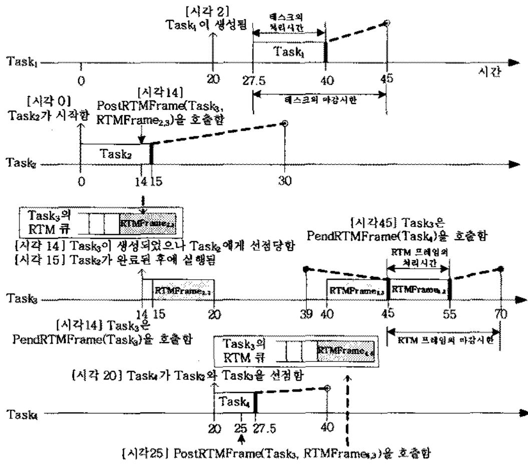


그림 6. EDF기반의 태스크 스케줄링과 RTCC-EDF기반의 RTM 프레임 스케줄링간의 분석

서 식 (2)의 충분조건을 만족하여 EDF 기법으로 스케줄링이 가능하다. $Task_2$는 시각 14에서 PostRTMFrame($Task_3$, $RTMFrame_{2,3}$)을 호출하여 $RTMFrame_{2,3}$을 $Task_3$의 RTM 큐에 전달한다. 시각 14에서 $Task_3$이 생성되지만 $Task_2$의 마감시한이 $Task_3$의 마감시한보다 더 이르기 때문에 $Task_2$의 실행이 계속된다. 시각 15에서 $Task_2$의 실행이 완료되면, $Task_3$의 실행이 시작된다. $Task_3$은 PendRTMFrame($Task_3$)을 호출하여 자신의 RTM 큐에서 $RTMFrame_{2,3}$을 선출 및 처리하기 시작한다. $Task_1$과 $Task_4$가 시각 20에서 생성된다. 시각 20에서 $Task_1$, $Task_3$, 그리고 $Task_4$의 프로세서 이용률은 0.723이 된다. 따라서 식 (2)의 충분조건을 만족하여 EDF 기법으로 스케줄링이 가능하다. 이때 $Task_1$, $Task_3$, 그리고 $Task_4$의 마감시한이 각각 45, 49, 40이다. 따라서 마감시한이 가장 짧은 $Task_4$가 $Task_3$을 선점한다. $Task_4$가 시각 25에서 PostRTMFrame ($Task_3$, $RTMFrame_{4,3}$)를 이용하여 $RTMFrame_{4,3}$을 $Task_3$의 RTM 큐에 전달한다. 시각 27.5에서 $Task_4$의 실행이 완료되면, 마감시한이 짧은 $Task_1$이 수행된다. 시각 40에서 $Task_1$의 실행이 완료되어 $Task_3$의 실행이 다시 시작된다. $Task_3$은 $RTMFrame_{2,3}$의 처리를 계속 수행한다. 그리고 시각 45에서 $RTMFrame_{2,3}$의 처리가 완료되지만, $RTMFrame_{2,3}$의 마감시한인 39를 초과하였다.

그림 6에서 살펴본 바와 같이 태스크 스케줄링이 식 (2)의 조건을 만족하더라도 RTM 프레임 스케줄러와의 협업을 하지 않고 독립적인 동작을 하면 RTM 프레임의 마감시한을 초과하게 되어 RTM 프레임에서 요구하는 실시간성 만족을 제공하지 못하게 된다. 이에 본 논문에서는 이러한 문제점을 해결하는 RTP-RTM 스케줄러 기법을 제안하였다. RTP-RTM 스케줄러에서는 RTM 프레임을 처리하는 소프트웨어 태스크가 RTM 프레임의 실시간성을 만족시켜 주기 위하여 RTM 프레임의 실시간 속성값을 소프트웨어 태스크의 실시간 속성값에 상속시킨다. 표 3에서 기술한 바와 같이 이러한 상속은 태스크 스케줄러의 속성에 따라 달라진다. 본 논문에서는 태스크 스케줄러로 EDF를 사용하므로 이에 해당되는 실시간 속성값의 상속 조건을 따른다.

그림 6에서 RTP-RTM 스케줄러를 적용하는 경우의 동작과정을 살펴보면 다음과 같다. 최초 시각 0부터 15까지는 앞에서 설명한 내용과 동일하게 진행된다. PendRTMFrame ($Task_3$)를 통해서 자신의 RTM 큐에서 $RTMFrame_{2,3}$을 선출 및 처리를 시작한다. 이때 표 3에서 기술한 바와 같이 $RTMFrame_{2,3}$의 절대적 마감시한이 $Task_3$의 절대적 마감시한보다 더 이르기 때문에, $Task_3$은 RTP-RTM 스케줄러에 의해서 $RTMFrame_{2,3}$의 절대적 마감시한인 39를 상속받

표 3. RTP-RTM 스케줄러에서 RTM 프레임의 선정기준과 실시간 속성값의 상속조건

태스크 스케줄러	RTM 프레임의 실시간 속성값을 상속하는 조건	RTM 프레임의 선정 기준
RM (Rate Monotonic)	RTM 프레임의 주기가 RTM 프레임 처리를 담당하는 수신 태스크의 주기보다 짧을 경우	RTM 프레임의 주기가 가장 짧은 RTM 프레임을 우선적으로 선출한다.
DM (Deadline Monotonic)	RTM 프레임의 상대적 마감시한이 RTM 프레임 처리를 담당하는 수신 태스크의 상대적 마감시한보다 짧을 경우	RTM 프레임의 상대적 마감시한이 가장 짧은 RTM 프레임을 우선적으로 선출한다.
EDF (Earliest Deadline First)	RTM 프레임의의 절대적 마감시한이 RTM 프레임 처리를 담당하는 수신 태스크의 절대적 마감시한보다 짧을 경우	RTM 프레임의 절대적 마감시한 (생성시간 + 마감시한)이 가장 이른 RTM 프레임을 우선적으로 선출한다.

는다. $Task_1$과 $Task_4$가 시각 20에서 생성된다. 시각 20에서 $Task_1$, $Task_3$ 그리고 $Task_4$의 마감시한이 각각 45, 39, 40이기 때문에 마감시한이 가장 이른 $Task_3$이 계속 수행하게 된다. $Task_3$이 시각 25에서 $RTMFrame_{2,3}$의 처리를 완료하여, $RTMFrame_{2,3}$의 마감시한을 만족시킨다. 시각 25에서 나머지 태스크 중에서 마감시한이 가장 이른 $Task_4$가 실행된다. 나머지 동작은 이와 유사하게 실행되며. 마지막으로 $Task_3$은 시각 55에서 $RTMFrame_{4,3}$의 처리가 완료되어 모든 태스크와 RTM 프레임의 마감시한은 만족된다. 이와 같이 RTP-RTM 스케줄러는 RTM 프레임의 실시간 속성값을 태스크에 상속하여 RTM 프레임의 실시간 속성을 만족시키고자 한다.

한편, RTP-RTM 스케줄러는 t 시점에서 처리할 RTM 프레임의 수락 검사(Acceptance Test)를 사용하여 다음의 세 가지 경우를 보장하고자 한다. 첫째, t 시점에서 RTM 프레임을 처리하고 있는 태스크를 제외한 나머지 실행준비 상태인 소프트웨어 태스크의 마감시한을 보장해야 한다. 둘째, 시각 t 에 RTM 큐에 도착한 RTM 프레임보다 우선순위가 높으며, 현재 실행준비상태인 RTM 프레임의 마감시한을 보장해야 한다. 셋째, 시각 t 에 RTM 큐에 도착한 RTM 프레임의 마감시한을 보장해야 한다.

식 (3)은 송신 태스크 s가 수신 태스크 d에 전달하는 $RTMFrame_{s,d}$의 수락 조건을 나타낸다.

$$U(t, RTMFrame_{s,d}) = \begin{bmatrix} TU(t) + OldRU(t, RTMFrame_{s,d}) \\ + NewRU\ (RTMFrame_{s,d}) \end{bmatrix} \le 1 \tag{3}$$

$TU(t)$는 t 시점에서 RTM 프레임을 처리하는 태스크를 제외한 나머지 실행준비 상태인 소프트웨어 태스크들의 전체 프로세서 이용률이며, 식 (4)와 같이 계산된다.

$$TU(t) = \sum_{i \in T_R(t) - TRTM_R(t)} \frac{RC_i(t)}{RD_i(t)} \tag{4}$$

$T_R(t)$는 t 시점에서 실행 중인 태스크의 집합을 의미하며, $TRTM_R(t)$는 t 시점에서 RTM 프레임을 처리하고 있는 태스크의 집합을 의미한다. 그리고 $RC_i(t)$는 t 시점에서 실행 중인 태스크 i의 남은 실행시간이며, $RD_i(t)$는 t 시점에서 태스크 i의 남은 마감시한이다.

$OldRU(t, RTMFrame_{s,d})$는 t 시점에서 도착한 $RTMFrame_{s,d}$보다 우선순위가 높고, 현재 실행준비상태인 RTM 프레임의 전체 프로세서 이용률이며, 식 (5)와 같이 계산된다.

$$OldRU(t, RTMFrame_{s,d}) = \sum_{\substack{RTMFrame_{i,j} \in \\ HigherRTMFrame_R(t, RTMFrame_{s,d})}} \frac{RC_{RTMFrame_{i,j}}(t)}{RD_{RTMFrame_{i,j}}(t)} \tag{5}$$

$HigherRTMFrame_R(t, RTMFrame_{s,d})$는 t 시점에서 처리 중인 RTM 프레임 중 $RTMFrame_{s,d}$보다 우선순위가 높은 RTM 프레임의 집합을 의미한다. $RC_{RTMFrame_{i,j}(t)}$는 t 시점에서 처리 중인 $RTMFrame_{i,j}$의 남은 실행시간이며, $RD_{RTMFrame_{i,j}(t)}$는 t 시점에서 $RTMFrame_{i,j}$의 남은 마감시한이다.

$NewRU\ (t, RTMFrame_{s,d})$는 시각 t 에 RTM 큐에 도착한 RTM 프레임의 프로세서 이용률이며, 식 (6)과 같이 계산된다. 식 (6)에서 $C_{RTMFrame_{s,d}}$과 $D_{RTMFrame_{s,d}}$는 $RTMFrame_{s,d}$의 수행시간 및 마감시한을 나타낸다.

$$NewRU\ (t, RTMFrame_{s,d}) = \frac{C_{RTMFrame_{s,d}}}{D_{RTMFrame_{s,d}}} \tag{6}$$

식 (3)에서 $U(t, RTMFrame_{s,d})$가 1보다 작거나 같으면, $RTMFrame_{s,d}$이 처리 가능하다는 것을 의미하여 RTM 프레임의 수락 검사를 통과하게 된다.

4. 네트워크 프로토콜 스택의 구현 및 성능 분석

이 장에서는 3장에서 기술한 소프트웨어 및 하드웨어 모듈간의 조립형 재구성이 가능한 네트워크 프로토콜 스택의 설계 기법을 검증하기 위하여 TCP/IP 프로토콜을 소프트웨어 및 하드웨어 태스크 단위로 분할한 후에 TCP/IP 프로토콜의 동작을 검증하였다. TCP/IP의 기능을 하드웨어로 구현한 태스크는 다음과 같다. TCP 프로토콜은 2개의 하드웨어 태스크로 구현되었으며, TCP 송신 기능을 담당하는 TCP-TX 태스크와 TCP 수신 기능을 담당하는 TCP-RX 태스크로 구성된다. IP 프로토콜은 역시 IP 패킷의 송신 기능을 담당하는 IP-TX 하드웨어 태스크와 IP 패킷의 수신 기능을 담당하는 IP-RX 하드웨어 태스크로 구성된다. 그리고 TCP와 IP 프로토콜에서 전송되는 패킷마다 요구되는 체크섬(Checksum) 계산을 담당하는 CKS-TX 하드웨어 태스크와 수신

패킷의 체크섬 계산을 담당하는 CKS-RX 하드웨어 태스크를 구현하였다. 마지막으로 ARP 프로토콜에서는 ARP Request와 Reply를 처리하는 기능을 ARP-RR 하드웨어 태스크로 구현하였다. 이더넷 칩셋의 디바이스 드라이브는 이더넷 프레임의 송신 기능을 담당하는 ETH-TX 소프트웨어 태스크와 수신 기능을 담당하는 ETH-RX 소프트웨어 태스크로 구성된다.

먼저 구현된 TCP/IP 프로토콜 스택의 성능을 검증하기 위하여 두 대의 SoC 플랫폼에 패킷을 송신 및 수신하는 주기 및 비주기 태스크들을 동작시켰다. 두 대의 SoC 플랫폼은 100Mbps 이더넷 통신을 수행한다. 동작 실험에서 송신 태스크는 주기 태스크의 집합으로 구성하였으며 표 4와 같다. 그리고 패킷의 수신은 네트워크 상황과 시스템의 상황에 패킷의 전달 상황이 다양하게 변화할 수 있기 때문에 수신 태스크는 비주기 태스크의 집합으로 구성하였으며 표 5와 같다.

표 4와 표 5는 멀티미디어 네트워킹 서비스를 이용하는 주기 및 비주기 응용 태스크 집합의 실시간 속성값과 CPU 이용률을 보여준다. 영상 패킷 송신 태스크는 표준문서 ISO/IEC 14496-2:2001 [15]를 참조하여 구현하였다. 최고 비트율(Peak Bit Rate)이 384kbps이며, 초당 30개의 1000바이트 영상 프레임을 주기적으로 생성한다. 영상 패킷 수신 태스크는 초당 30개의 패킷을 수신하는 패킷 도착률이 30인 비주기 태스크로 동작한다. 음성 패킷 전송 태스크는 표준문서 ISO/IEC 13818-3:1998 [16]을 참조하여 구현하였다. 최고 비트율이 64kbps이며, 표본 추출 비율(Sampling Rate)이 44.1kHz이며 초당 40개의 208바이트 패킷을 전송한다. 음성 패킷 수신 태스크는 도착률이 40인 비주기 태스크이다. 주기 패킷 전송 태스크와 비주기 패킷 전송 태스크는 음성/영상 패킷 이외에 생성되는 1450바이트 패킷을 각각 전송한다. 그리고 다양한 CPU 이용률에서 각 태스크의 실시간성 만족도를 검증하기 위해, 표 4와 표 5에 주기 및 비주기 태스크 집합을 포함하여 0.4부터 1.0까지의 CPU 이용률이 되도록 추가적인 더미(Dummy) 태스크 집합을 구성하여 동작시켰다.

표 6은 각 패킷별 지연 시간과 각 패킷이 담긴 RTM 프레임에 대하여 각 네트워크 프로토콜 태스크가 처리해야 하는 마감시한을 보여준다. 영상 및 음성 패킷의 응용 태스크간 최대 수신 패킷의 허용 지연 시간은 Nahrstedt와 Steinmetz [17]을 참조하였으며, 종단 SoC 시스템간 패킷 전송 지연 시간은 130ms라고 가정하였다. 그리고 응용 태스크간 최대 수신 패킷의 허용 지연 시간에서 종단 SoC 시스템간 패킷 전송 지연 시간을 뺀 나머지 시간을 각 네트워크 프로토콜 태스크의 수행 시간에 비례하게 나눈 값을 네트워크 프로토콜 태스크에게로 전달되는 RTM 프레임의 마감시한으로 할당하였다.

표 7은 SoC 플랫폼에서 각 태스크의 개별적인 평균 실행시간을 보여준다. SW-TCP/IP-RX는 TCP/IP 프로토콜의 수신 기능을 소프트웨어 태스크로만

표 4. 주기 태스크 집합

태스크	주기 (ms)	수행시간(ms)	마감시한(ms)	CPU 이용률
영상 패킷 송신 태스크	30	3.57	30	0.119
음성 패킷 송신 태스크	20	1.32	20	0.066
주기 패킷 송신 태스크	25	0.50	25	0.020

표 5. 비주기 태스크 집합

태스크	패킷 도착률 (Packet Arrivals per second)	수행시간 (ms)	마감시한 (ms)	CPU 이용률
비주기 패킷 송신 태스크	50	0.50	20	0.025
비주기 패킷 수신 태스크	50	0.50	20	0.025
영상 패킷 수신 태스크	33	2.31	30	0.077
음성 패킷 수신 태스크	50	0.78	20	0.039
주기 패킷 수신 태스크	40	0.50	25	0.020

표 6. 패킷별 RTM 프레임의 마감시한

네트워크 태스크 \ 응용 태스크	응용 태스크 간 최대 수신 패킷 허용 지연 (ms)	종단 SoC 시스템간 최대 허용 패킷 전송 지연 (ms)	TCP-TX & TCP-RX 태스크 마감 시한 (ms)	IP-TX & IP-RX 태스크 마감 시한 (ms)	CKS-TX & CKS-RX 태스크 마감 시한 (ms)	ARP-RR 태스크 마감 시한 (ms)	ETH-TX & ETH-RX 태스크 마감 시한 (ms)
영상 패킷	150	130	2	1	7	1	9
음성 패킷	250	130	12	6	42	6	54
주기 패킷	3000	130	287	143.5	1004.5	143.5	1291.5
비주기 패킷	3000	130	287	143.5	1004.5	143.5	1291.5

표 7. 소프트웨어 및 하드웨어 네트워크 태스크의 평균 실행시간

네트워크 태스크	TCP-TX	TCP-RX	IP-TX	IP-RX	ARP-RR	CKS-TX	CKS-RX	ETH-RX	ETH-TX
소프트웨어 태스크의 수행시간 (us)	155	153	127	125	133	797	766	8064	4454
하드웨어 태스크의 수행시간 (us)	94	92	88	87	121	162	157		

구현한 것이며 HW-TCP/IP-RX는 ETH-RX 태스크를 제외한 나머지 태스크를 하드웨어 태스크로 구현한 것이다. 그리고 앞서 기술한 내용과 유사하게 SW-TCP/IP-TX와 HW-TCP/IP-TX는 송신 기능을 소프트웨어와 ETH-TX 태스크를 제외한 나머지 태스크를 하드웨어 태스크로 구현한 것이다. 이 실험에서는 각 패킷의 크기 별로 1000번의 평균 전송 지연 시간을 측정해서 그 평균값을 계산하였다. 표 7에서 보는 바와 같이 하드웨어 태스크의 처리 성능이 소프트웨어 태스크보다 약 1.6배 더 빠름을 알 수 있다. 특히, CKS-TX 하드웨어 태스크와 CKS-RX 하드웨어 태스크의 경우에는 약 4.8배 더 빠른데, 이는 하드웨어 태스크에서 AHB의 기능인 Burst I/O를 사용하였기 때문이다. 또한 각각의 하드웨어 태스크는 독립적으로 동시 연산이 가능하여 소프트웨어 태스크보다 우수한 성능을 보여준다.

그림 7은 표 4와 표 5의 주기 및 비주기 태스크 집합이 실행하는 경우에 각 CPU 이용률에 따라 측정한 평균 CPU 여유율 및 초당 평균 문맥 전환 횟수를 보여준다. 평균 CPU 여유율은 HW-TCP/IP-TX&RX가 SW-TCP/IP-TX&RX를 사용하는 경우보다 평균 약 0.06만큼 더 높아졌다. 그리고 초당 평균 문맥 전환 횟수는 HW-TCP/IP-TX&RX가 SW-TCP/IP-TX&RX를 사용하는 경우보다 평균 70회 정도 감소하였다.

그림 8부터 그림 10까지는 RTM 프레임 스케줄러

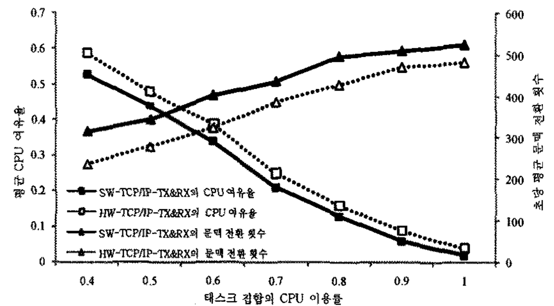
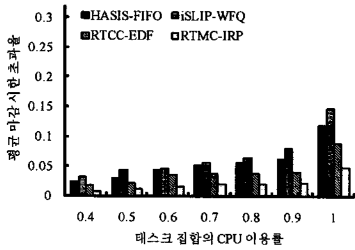
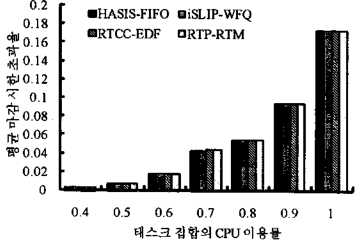


그림 7. 평균 CPU 여유율 및 초당 평균 문맥 전환 횟수

인 RTP-RTM과 기존의 스케줄링 기법인 HASIS-FIFO, iSLIP-WFQ, 그리고 RTCC-EDF간의 성능 비교를 하였다. RM, DM, EDF 태스크 스케줄러를 사용하는 각각의 경우를 고려하여 RTM 프레임의 평균 마감 시한 초과율을 측정하였다. 그리고 RTM 프레임 스케줄러가 실행할 때 발생되는 전체 태스크의 마감시한 초과율도 측정하였다. 표 4와 표 5에서 보여준 태스크 집합을 사용하였다. 각 그림의 (a)에서 볼 수 있듯이 RTP-RTM 스케줄러는 다른 RTM 프레임 스케줄러를 사용했을 때보다 RTM 프레임의 마감시한 초과율이 최소 4.7%, 평균 43.7%만큼 감소하였다. 특히 마감시한과 같은 실시간 속성을 고려하지 않은 HASIS-FIFO와 iSLIP-WFQ기반의 RTM 프레임 스케줄러는 항상 RTM 프레임의 마감시한 초과율이 가장 높았다. 특히 영상 및 음성 패킷이 담긴 RTM 프레임의 마감시한이 많이 초과되었다. 각

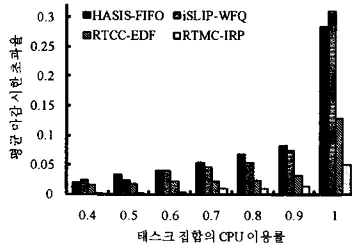


(a) RTM 프레임의 평균 마감시한 초과율

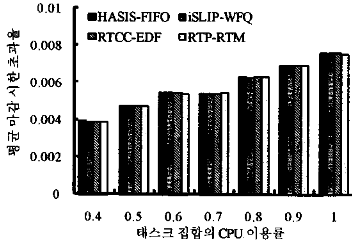


(b) 태스크 집합의 평균 마감시한 초과율

그림 8. RTM 프레임 및 태스크의 평균 마감시한 초과율 (RM 태스크 스케줄러를 적용)

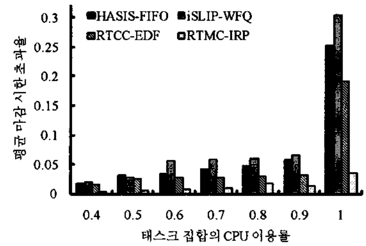


(a) RTM 프레임의 평균 마감시한 초과율

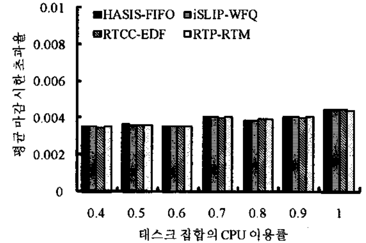


(b) 태스크 집합의 평균 마감시한 초과율

그림 9. RTM 프레임 및 태스크의 평균 마감시한 초과율 (DM 태스크 스케줄러를 적용)



(a) RTM 프레임의 평균 마감시한 초과율



(b) 태스크 집합의 평균 마감시한 초과율

그림 10. RTM 프레임 및 태스크의 평균 마감시한 초과율 (EDF 태스크 스케줄러를 적용)

그림의 (b)에서 보는 바와 같이 태스크 집합의 마감시한 초과율이 RTM 프레임의 스케줄링 알고리즘에 의해 거의 영향을 받지 않았음을 보여주지만, 절대적 마감시한에 따라 태스크의 우선순위를 동적으로 할당하는 EDF 태스크를 사용하는 경우에 태스크 집합의 평균 마감시한 초과율이 감소하였다.

그림 11은 종단 SoC 플랫폼간 패킷의 전송지연 시간을 보여준다. 그림 8부터 그림 10까지 살펴본 바와 같이 EDF 태스크 스케줄러가 가장 낮은 마감시한의 초과율을 보여주었기에 그림 11에서는 EDF 태스크 스케줄러를 사용하였다. 그림 10에서 보는 바와 같이 RTP-RTM 프레임 스케줄러가 RTM 프레임의 가장 낮은 평균 마감시한 초과율을 보여 주었던 종단 SoC 시스템간 패킷의 전송지연 시간에도 많은 영향을 주었음을 알 수 있다. RTP-RTM 스케줄러에서의 종단 SoC 시스템간 패킷의 전송지연 시간이 다른 RTM 프레임 스케줄러를 사용할 때보다 평균 68.1% 정도 감소하였다.

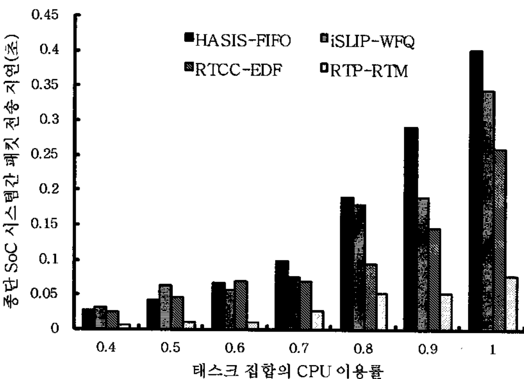


그림 11. 종단 SoC 시스템간 패킷 전송 지연

5. 결 론

본 논문에서는 네트워크 프로토콜의 기능 명세를 소프트웨어 및 하드웨어 태스크로 분할한 후에 태스크 단위에서 조립형 재구성이 가능한 네트워크 프로토콜 스택의 설계 기법을 제안하였다. 또한 네트워크 기능을 사용하는 실시간 응용 서비스의 마감시한을 보장하기 위하여 개별 멀티미디어 응용 태스크의 마감시한을 보장함과 동시에 각 네트워크 태스크 간에 교환되는 RTM 프레임의 마감시한을 보장하는 기법인 RTP-RTM 프레임 스케줄러를 제안하였다. 본 논문에서 제안한 RTP-RTM 프레임 스케줄링 기법은 기존의 스케줄링 기법보다 평균 43.7% 감소한 RTM 프레임과 태스크의 마감시한 초과율을 제공함과 동

시에 평균 68.1% 감소한 종단 SoC 시스템간의 낮은 패킷 전송 지연시간을 제공할 수 있다. RTP-RTM 프레임 스케줄링 기법과 더불어 병렬 수행 및 동시 연산, 그리고 Burst I/O 기법을 적용하여 하드웨어 태스크의 빠른 실행과 CPU에 부과된 처리량을 감소시켰다. 이를 통해 평균 CPU 여유율은 향상되고, 소프트웨어 태스크간의 초당 평균 문맥 전환 횟수는 감소하여 전체 시스템의 성능이 향상된다. 이에 SoC 플랫폼에서 각 네트워크 프로토콜의 기능을 태스크 단위로 분할한 후에 소프트웨어 및 하드웨어 기반의 네트워크 프로토콜 태스크로 재구성이 가능하고, RTP-RTM 프레임 스케줄러를 사용하여 하드웨어 및 소프트웨어 네트워크 태스크 간의 실시간 메시지 교환을 제공할 수 있는 시스템을 설계 및 구현할 수 있음을 확인하였다.

참 고 문 헌

[1] J.P. Durbano, F.E. Ortiz, J.R. Humphrey, P.F. Curt, and D.W. Prather, "FPGA-Based Acceleration of the 3D Finite-Difference Time-Domain Method," IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 156-163, 2004.

[2] M. Gokhale, J. Frigo, C. Ahrens, J.L. Tripp, and R. Minnich, "Monte Carlo Radiative Heat Transfer Simulation on a Reconfigurable Computer," International Conference on Field Programmable Logic and Application, pp. 95-104, 2004.

[3] D.W. Kim, W.O. Kwon, K. Park, and S.W. Kim, "Internet Protocol Engine in TCP/IP Offloading Engine," International Conference on Advanced Communication Technology, Vol. 1, pp. 270-275, 2008.

[4] J.C. Hamerski, E. Reckziegel, and F.L. Kastensmidt, "Evaluating memory sharing data size and TCP connections in the performance of a reconfigurable hardware-based architecture for TCP/IP stack," IFIP International Conference on Very Large Scale Integration, pp. 212-217, 2007.

[5] Z.Z. Wu and H.C. Chen, "Design and Implementation of TCP/IP Offload Engine System over Gigabit Ethernet," Proceedings of 15th International Conference on Computer Communications and Networks, pp. 245-250, 2006.

[6] S.M. Chung, C.Y. Li, H.H. Lee, J.H. Li, Y.C. Tsai, and C.C. Chen, "Design and implementation of the high speed TCP/IP Offload Engine," International Symposium on Communications and Information Technologies, pp. 574-579, 2007.

[7] M.B. Srivastava and R.W. Brodersen, "Rapid-prototyping of hardware and software in a unified framework," IEEE International Conference on Computer-Aided Design, pp. 152-155, 1991.

[8] F. Fischer, A. Muth, and G. Farber, "Towards interprocess communication and interface synthesis for a heterogeneous real-time rapid prototyping environment," Proceedings of the Sixth International Workshop on Hardware/Software Codesign, pp. 35-39, 1998.

[9] M. Eisenring and M. Platzner, "Synthesis of interfaces and communication in reconfigurable embedded systems," IEE Proceedings Computers and Digital Techniques, Vol. 147, No. 3, pp. 159-165, 2000.

[10] S. Gopalakrishnan, M. Caccamo, and L. Sha, "Switch Scheduling and Network Design for Real-Time Systems," Proceedings of 12th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 289-300, 2006.

[11] A. Yiming and T. Eisaka, "A switched Ethernet protocol for hard real-time embedded system applications," Proceedings of 19th International Conference on Advanced Information Networking and Applications, Vol. 2, pp. 41-44, 2005.

[12] Altera, Excalibur EPXA4 device, available at http://www.altera.com/literature/es_epxa4.pdf.

[13] LAN91C111, 10/100 Non-PCI LAN ethernet controller, SMSC Corporation, available at http://www.smsc.com/main/datasheets/91c111.pdf.

[14] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri, Scheduling in Real-time systems, John Wiley, England, 2002.

[15] ISO/IEC, "Coding of audio-visual objects - Part 2: Visual," ISO/IEC 14496-2:2001 Second Edition, 2001.

[16] ISO/IEC, "Information technology - Generic coding of moving pictures and associated audio information - Part 3: Audio," ISO/IEC 13818-3:1998 Second Edition, 1998.

[17] K. Nahrstedt and R. Steinmetz, "Resource management in networked multimedia systems," IEEE Computer, Vol. 22, No. 11, pp. 52-63, 1995.

김 영 만

2007년 2월 부산대학교 컴퓨터공학과 졸업
2007년 3월~현재 부산대학교 컴퓨터공학과 석박사통합과정
관심분야 : SoC설계, 실시간 시스템

탁 성 우

1995년 2월 부산대학교 컴퓨터공학과 졸업
1997년 2월 부산대학교 컴퓨터공학과 석사
2003년 2월 미국미주리주립대학교 ComputerScience 박사
2004년~현재 부산대학교 정보컴퓨터공학부 부교수
2004년~현재 부산대학교 컴퓨터 및 정보통신 연구소 겸임 연구원
관심분야 : 유무선 네트워크, Soc 설계, 실시간 시스템, 위치인식, 최적화 기법, 그래프 이론, 큐잉이론