논문 2010-05-16

실시간 시스템에서 빠른 문맥 전환을 위한 다중 레지스터 파일

(Multiple Register Files for Fast Context Switching in Real-Time Systems)

김 종 웅, 조 정 훈*
(Jong-Wung Kim, Jeoung-Hun Cho)

Abstract : Recently complexity of embedded software cause to be used real-time operating system (RTOS) to implement various functions in the embedded system. And also, according to requirement of complex functions in embedded systems, the number as well as complexity of tasks get increased continuously. In case that many tasks collaborated in a microprocessor, context switching time between tasks is a overhead waisting a CPU resource. Therefore the time of task context switching is an important factor that affects performance of RTOS. In this paper, we concentrate on the improvement of task context switch for reducing overhead and achieving fast response time in RTOS. To achieve these goal, we suggest multiple register files and task context switching algorithm. By reducing the context switch overhead, we try to ease scheduling and assure fast response times in multitasking environment. As a result, the context switch overhead decreased by 8~16% depend on the number of register files, and some task set which are not schedulable with single register file are schedulable due to that decrease with multiple register files.

Keywords : Task context switch, RTOS, Embedded system, Register file

1. 서 론

최근 임베디드 시스템의 다중 작업 및 실시간 처리능력, 멀티미디어 처리 등이 강화되고 유무선 통신 및 네트워크와의 접목으로 전통적인 제조, 유통, 서비스 산업뿐만 아니라 항공, 우주, 국방, 멀티미디어 통신 및 에너지 개발 등의 첨단 분야에 이르기까지 그 사용범위와 영향력이 점점 커지고 있다[1]. 이에 따라 임베디드 어플리케이션은 많은 기능들이 요구되면서 점점 복잡해져가고 있다. 이러한 임베디드 어플리케이션의 복잡도 증가는 RTOS (Real-time Operating Systems)의 사용을 증가시키는 결과를 가져왔다. 임베디드 시스템에서 RTOS들은 현재 우리 주위의 차량, 휴대폰, 통신 장비, 산업 설비 등에 널리 사용되고 있으며 분야에 따라 OSEK[2], REX[3], L4[3], IOS[4] 등을 사용하고 있다. RTOS는 스케줄링, 자원 관리, 동기화, 통신, 정밀한 타이밍 등을 위한 기본적인 지원을 제공하기 때문에 많은 기능을 수행하는데 도움을 주고 있다[5].

하지만, RTOS는 정상적인 태스크 실행 외에 자체 오버헤드 때문에 성능 저하의 부담을 안고 있다. 대부분의 성능 저하는 태스크 스케줄링, 시간 관리, 이벤트 관리와 같은 RTOS의 코어 오퍼레이션에 의해 야기된다. 이러한 오버헤드는 예측성을 저해하고 응답 시간을 늦추는 등의 실시간 시스템에서 민감하게 작용될 수 있다. 그러므로 오버헤드를 줄이는 기법은 중요한 일이라고 할 수 있다.

우리는 RTOS의 여러 가지 오버헤드 중에서 태

* 교신저자(Corresponding Author)
논문접수 : 2010. 08. 11., 수정일: 2010. 08. 20.,
채택확정 : 2010. 09. 02.
김종웅, 조정훈 : 경북대학교 전자전기컴퓨터학부
※ 본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음 (NIPA-2010-C1090-1031-0004)

스크 문맥 전환의 오버헤드에 집중했다. RTOS의 특징 중 멀티태스킹을 위해, 문맥 전환은 하나의 프로세스 또는 스레드로부터 다른 프로세스 또는 스레드로의 CPU사용을 변경하기 위해서 사용된다. 즉, 문맥 전환은 멀티태스킹을 가능하게 하는 방법이다. 동시에, 문맥 전환은 진행하던 태스크의 상태를 저장하고 다른 태스크의 상태를 복원하는 과정에서 피할 수 없는 시스템 오버헤드를 야기한다[6-7].

특히 이 문맥 전환 오버헤드는 RTOS의 중요한 특징인 빠른 응답 시간을 저해하는 문제가 된다. 일반적으로 문맥 전환의 오버헤드는 전체 수행 시간의 약 10~20%정도를 차지한다고 알려져 있다[8]. 따라서 우리는 태스크 문맥 전환 시간을 줄임으로써 빠른 응답 시간을 좀 더 보장하려고 한다.

본 논문에서 우리는 하드웨어에 의한 빠른 태스크 문맥 전환 방법을 소개한다. 그리고 일반적인 RTOS들의 태스크 문맥 전환 과정을 분석하여 오버헤드를 줄일 수 있는 방법을 연구한다. 태스크 문맥 전환의 핵심은 문맥 전환 직전의 태스크 상태와 레지스터들을 저장하고, 새로운 태스크의 상태와 레지스터들을 복원하는 과정이다. 그 중에서 각 태스크가 사용하는 레지스터들을 저장하고 복원하는 과정이 큰 비중을 차지하고 있다. 따라서 우리는 이 레지스터들의 저장 및 복원과정을 없애거나 줄이기 위한 연구를 한다. 그 결과, 이러한 연구를 토대로 기존의 하드웨어 구조에 몇 개의 레지스터 파일을 추가하는 방법을 제안한다. 레지스터 파일의 추가는 레지스터의 변경만으로 문맥 전환을 하게 됨으로서 기존의 저장 및 복원 과정에서 오는 오버헤드를 감소시킬 수 있다. 본 논문에서는 RM 스케줄링 하에서 태스크의 수와 추가된 레지스터 파일의 수의 차이에 따른 문맥 전환 오버헤드 감소를 비교한다. 그 결과 태스크 문맥 전환 오버헤드가 감소되는 것을 확인하고, 이에 따라 스케줄링의 여유를 늘임으로서 기존에 비해 보다 더 빠른 응답 시간을 보장할 수 있게 할 것이다.

본 논문은 다음과 같이 구성된다. 다음 절에서는 문맥 전환 오버헤드 감소와 관련된 연구에 대해서 소개한다. 3절에서는 우리의 개선된 모델에 대해서 설명하고, 4절에서는 RM 스케줄링 하에서의 테스트와 결과에 대해서 설명할 것이다. 마지막으로 5절에서는 결론 및 앞으로 우리의 작업에 대해서 설명할 것이다.

Ⅱ. 관련 연구

멀티태스킹의 실행은 여러 태스크들 사이에서 CPU를 공유하는 것이다. 많은 연구자들은 RTOS에서 주로 스케줄링 메커니즘에 초점을 맞추었었다. 하지만 스케줄링 메커니즘에서 태스크 문맥 전환에는 주의를 덜 기울였다. 그중에서 문맥 전환 오버헤드를 줄이기 위해 선행된 연구들은 두 가지 성능 측정으로부터의 이득을 위해 디자인되어졌다. 그것은 각 문맥 전환에서의 레지스터 이동 횟수와 문맥 전환의 빈도이다. 각 문맥 전환에서의 레지스터 이동을 줄이기 위한 아키텍처와 컴파일러 테크닉에 관한 연구들이 있다[10-11]. 이 아이디어는 실행 시간 동안 실행 중인 프로그램의 레지스터 사용량이 서로 다르다는 것에 착안을 한 것이다. 그러므로 문맥 전환 이벤트는 문맥 전환 포인트를 선택하기 위해 지연되고, 운영체제는 미리 준비된 테이블을 확인함으로서 필요한 레지스터들을 변경할 수 있다. 그러나 문맥 전환 포인트들의 수는 테이블 사이즈, 성능과 응답 시간에 영향을 끼친다. 그리고 이는 실시간 시스템에서 정확한 태스크 스케줄링을 필요로 한다.

다른 방법으로, 문맥 전환의 빈도를 줄이기 위하여, 프로세서는 전환 외에 더 많은 문맥을 포함하기 위한 능력이 필요하게 되어 프로세서는 더 많은 레지스터들을 필요로 한다. 이전의 연구들로부터, 고성능의 멀티스레드 프로세서에서 이들 레지스터를 사용하기 위한 두 가지 방법이 있다. 하나는 하드웨어 스레드가 스스로의 레지스터 파일을 가지는 것이고[12-14], 다른 하나는 다수의 완전히 결합된 레지스터 파일을 공유하는[15] 방법이다. 두 가지 방법이 고성능 멀티스레드 프로세서를 위해 적당함에도 불구하고, 스레드 단위로 레지스터 파일을 추가하거나 결합함으로 인한 제조비용과 파워 소모량 등이 많이 늘어나게 되어 임베디드 프로세서들을 위한 상당한 단점이 된다.

본 논문에서는 일반적인 성능의 프로세서를 대상으로 태스크 단위의 독립된 레지스터 파일들을 추가한다. 각 태스크들이 각 레지스터 파일들을 점유하거나, 혹은 일부 태스크들이 각 레지스터 파일들을 점유하고 나머지 태스크들은 기존처럼 하나의 레지스터 파일을 사용하는 복합적인 방법을 제안한다.

Ⅲ. 본 론

1. 개선

우리는 문맥 전환의 속도를 빠르게 하기 위해서 일반적인 RTOS들의 태스크 문맥 전환 과정을 분석했다. 범용 레지스터들과 상태 레지스터, 스택 포인터를 저장하고 복원하는 과정을 확인 할 수 있었고, 그중에서 가장 많은 명령어를 사용하는 것은 범용 레지스터들을 저장하고 복원하는 과정임을 알 수 있었다. 이러한 문맥 전환 시간을 수식으로 나타내면 다음과 같다[6-7].

$$(m+n)b \times K \ time \qquad (1)$$

(m : 범용 레지스터 수, n : 상태 레지스터 및 스택포인터, b : 저장 명령어의 수, K : 저장 명령어를 수행하는데 필요한 시간)

수식 1은 범용 레지스터와 상태 레지스터 및 스택포인터를 저장할 때 발생하는 시간을 문맥 전환 오버헤드로 표현하였음을 알 수 있다. 또한 수식 1은 저장할 때 걸리는 시간이고 같은 방식으로 복원할 때도 같은 시간이 걸림을 알 수 있다. 따라서 우리는 범용 레지스터들을 저장하고 복원하는 과정을 줄이거나 없앨 수 있다면 문맥 전환의 속도를 향상시킬 수 있다고 판단했다.

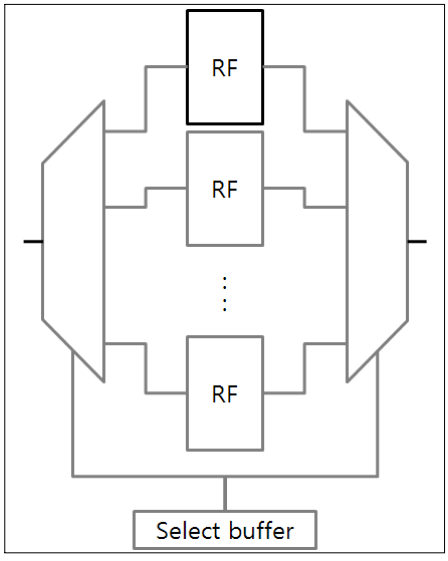

그림 1. 변경된 하드웨어 구조
Fig. 1. Modified architecture

범용 레지스터들의 저장과 복원 과정을 줄이기 위해 우리는 간단하게 레지스터 파일을 추가하는 방법을 생각했다. 레지스터 파일을 추가하여 각 태스크가 이를 사용하게 된다면, 문맥 전환이 일어날 때 범용 레지스터들을 저장하거나 복원 할 필요가 없이 단지 레지스터 파일만을 바꾸어 주는 것만으로 문맥 전환이 가능하게 되므로 오버헤드를 많이 줄일 수 있을 것이다. 따라서 우리는 그림 3과 같이 레지스터 파일을 여러 개 추가하는 구조를 구상하였다. 이 구조는 기존의 프로세서가 1개의 레지스터 파일을 가지는 것에 비해, 새로운 구조는 2개 이상의 레지스터 파일을 가지고 멀티태스킹을 하게 된다. 추가될 레지스터 파일의 개수는 대상 프로세서에 따라 최소 2개부터 더 많은 레지스터 파일을 가질 수 있다. 예를 들면 태스크의 수가 5개가 넘지 않는 소형 임베디드 프로세서의 경우에는 태스크의 수만큼 레지스터 파일을 추가할 수 있고, 일반적인 경우에는 소수의 레지스터 파일만을 추가함으로서 성능을 개선할 수 있다.

그림 1에서처럼, 레지스터 파일을 임의의 수만큼 추가한 뒤, 이들 레지스터 파일을 선택할 수 있는 MUX와 DEMUX를 추가하는 구조를 선택하였다. 그리고 이 MUX와 DEMUX에 레지스터 파일을 선택하는 셀렉트 신호를 위해 별도의 셀렉트 버퍼를 추가하였다. 이를 통해 셀렉트 버퍼에 값을 기록하는 것으로 간단히 사용 할 레지스터 파일을 선택할 수 있게 하였다.

단지 태스크를 대상으로 레지스터 파일 추가만을 고려하였을 때, 우리는 새로운 문제점을 발견하였다. 그것은 일반적인 RTOS에서, 타이머에 의해 주기적으로 발생하는 인터럽트 서비스 루틴에 의해 문맥 전환이 일어날 때, 단순히 태스크의 정보를 저장하고 복원만 하는 것이 아니라, 내부 시간을 계산하는 함수와 스케줄러 함수 등(uC/OS-II[16]의 경우 OSTimeTick(), OSIntExit() 등)도 호출되는 것이다. 이들 과정이 문맥 전환에 포함되기 때문에 단순히 레지스터 파일만을 변경하여 태스크를 변경한다면, 원래 태스크의 정보에 위와 같은 기타 함수 수행에 의해 값이 덮어씌어질 것이고 오류가 발생하게 될 것이다. 따라서 우리는 이들 문맥 전환이 수행되는 동안 사용할 수 있는 임시 레지스터 파일을 추가하는 방법을 생각했다. 하지만 문맥 전환의 수행과 기타 함수의 수행은 지극히 적은 개수의 범용 레지스터만을 사용하게 됨을 확인할 수 있었다. 따라서 새로 추가될 임시 레지스터 파일의 범용 레지스터 수는 기본적인 레지스터 파일의 범용 레지

스터 수에 비해 단지 몇 개만 추가함으로서 부담을 감소시켰다.

결과적으로, 추가할 레지스터 파일의 수 외에 별도로 작은 크기의 임시 레지스터 파일을 추가하게 되었다. 이 임시 레지스터 파일 외에는 사용 목적에 따라 레지스터 파일을 추가하여 문맥 전환에서 발생하는 오버헤드를 줄이게 될 것이다.

2. 상황 별 성능 계산

개선된 구조에서 문맥 전환이 발생할 때 두 가지 경우가 존재한다. 태스크의 수가 레지스터 파일의 수와 같거나 적을 경우, 그리고 태스크의 수가 레지스터 파일의 수보다 많을 경우이다. 각각의 경우에 대해 성능을 계산해보면 다음과 같다.

2.1 태스크의 수 ≤ 레지스터 파일의 수

태스크의 수가 레지스터 파일의 수보다 같거나 적을 경우는 각 태스크가 레지스터 파일을 하나씩 점유하여 사용할 수 있다. 이는 문맥 전환이 발생할 때, 태스크별로 레지스터 파일을 선택해야하는 과정이 필요 없음을 의미한다. 따라서 이 경우의 문맥 전환 시간은 다음 수식 2과 같이 계산 될 수 있다.

$$(nb \times K) + s \quad time \tag{2}$$

(s : 레지스터 파일을 바꾸는 명령어의 수)

범용 레지스터들을 모두 저장하거나 복원하는 대신, 단지 셀렉트 버퍼에 값을 기록하는 간단한 명령어만으로 레지스터 파일을 바꾸어 문맥 전환을 할 수 있으므로, 기존의 문맥 전환 시간인 수식 1보다 빠른 시간에 수행할 수 있음을 알 수 있다. 예를 들면, 일반적인 범용 레지스터의 수가 32개라고 가정할 때, 저장과 복원에 약 64개의 명령어 수행이 필요하지만, 이 경우에는 단지 레지스터 파일을 변경하기 위한 셀렉트 버퍼에 기록하는 2개의 명령어만 필요하게 되어 시간을 단축할 수 있다.

2.2 태스크의 수 > 레지스터 파일의 수

태스크의 수가 레지스터 파일의 수보다 많을 경우에는 앞의 경우와 다르게 몇 개의 과정이 더 추가되게 된다. 이 경우, 문맥 전환이 발생할 때, 각 태스크가 사용될 레지스터 파일을 결정하는 과정이 필요하고, 경우에 따라서는 몇 개의 태스크는 기존의 문맥 전환 과정처럼 범용 레지스터들을 저장하고 복원하는 과정을 선택해야할 수도 있다. 태스크가 어떤 레지스터 파일을 선택하고 사용하는 지는 어떤 스케줄링 기법을 사용하느냐에 따라서 달라질 수 있을 것이다. 본 논문에서는 RM 스케줄링 방법을 사용한다고 가정한다.

RM 스케줄링 방법을 사용한다고 할 때, 발생 빈도가 높은 태스크인 우선순위가 높은 태스크들이 레지스터 파일을 항상 점유하고, 나머지 태스크들은 1개의 레지스터 파일을 이용해 기존의 문맥 전환처럼 저장과 복원을 진행할 경우, 가장 나은 성능을 볼 수 있다. 즉 n개의 레지스터 파일이 있을 경우, 1개의 레지스터 파일을 제외한 (n-1)개의 레지스터 파일에 우선순위가 높은 (n-1)개의 태스크가 각각의 레지스터 파일을 점유하고, 나머지 태스크들이 1개의 레지스터 파일을 기존의 문맥 전환 방법을 사용하는 것이다. 이 경우 문맥 전환 시간은 두 가지 경우로 다음 수식 7, 8과 같이 나타낼 수 있다.

$$(nb \times K) + s + S \quad time \tag{3}$$

$$(m+n)b \times K + s + S \; time \tag{4}$$

(S : 레지스터 파일이 사용되고 있는지 체크하는 시간)

수식 3, 4은 개선된 구조의 시간과 기존 구조의 시간이 함께 나타나고 있다. 추가적으로 두 수식에는 공통적으로 현재 사용할 태스크가 레지스터 파일을 점유하던 태스크인지, 혹은 기존의 방식으로 문맥 전환을 해야 하는 지를 체크하는 시간이 존재한다. 하지만 이 추가된 시간은 단순히 간단한 조건식만으로 수행 가능하므로 큰 오버헤드가 아니다.

RM 스케줄링의 특성상 우선순위가 높은 태스크는 그만큼 자주 발생하고, 우선순위가 낮은 태스크는 드물게 발생하므로 상대적으로 수식 7의 비중이 많이 커지게 된다. 이는 상대적으로 기존의 문맥 전환 시간보다 빠르다는 것을 나타내며, 레지스터 파일의 수가 태스크의 수에 근접할수록 더욱 빨라지는 결과를 얻을 것이다.

IV. 실 험

1. 실험환경 및 조건

빠른 문맥 전환의 실험을 위해 우리는 C언어를 이용해 시뮬레이터를 제작하여 가상환경을 구축하였다. 시뮬레이터는 실제 프로세서와 비슷하게 클럭 단위로 동작을 하며, CPI(Clock per Instruction)가 1이라 가정하여 각 태스크들은 한 클럭당 하나의 명령어를 수행한다. 하지만 본 논문의 실험에서는 각 태스크가 어떠한 명령어를 수행하는 것보다 스케줄링 된 각 태스크들의 수행 시간에 따른 결과가

중요하므로, 각 태스크들은 실행될 때 클럭을 카운팅하는 방법을 취하여 수행 시간을 측정하였다. 그리고 태스크의 수와 레지스터 파일의 수를 조절하여 테스트할 수 있도록 하였다. 또한 기본적으로 RM 스케줄링 방법으로 스케줄링 되고, 문맥 전환 오버헤드에 대해 앞서 제시한 각 상황별로 그 값을 설정할 수 있도록 하였다.

태스크의 수가 레지스터 파일의 수보다 같거나 적을 경우와 태스크의 수가 레지스터 파일의 수보다 많을 경우의 두 가지 환경으로 나눠서 시뮬레이션을 진행하였다. 또한 사용자 태스크는 각각의 스케줄링에 맞게 각자 작업을 수행 하게 된다. 사용자 태스크가 모두 대기상태에 빠졌을 때 IDLE 태스크가 실행되므로, 문맥 전환 오버헤드에 따라 이 IDLE 태스크의 실행량이 달라질 것이다. 따라서 각 경우에 IDLE 태스크가 얼마나 실행되었는가에 따라 문맥 전환 오버헤드를 측정하였다.

실험을 위해 태스크는 IDLE 태스크를 포함하여 6개로 설정하고, 레지스터 파일은 1개에서 6개까지 가변시켰다. 태스크의 수가 레지스터 파일의 수와 같거나 적을 경우에는 레지스터 파일을 6개로 하고 수식 6을 이용하여 문맥 전환 오버헤드를 적용하였다. 또한 태스크의 수가 레지스터 파일의 수보다 많을 경우에는 레지스터 파일을 2개부터 6개까지 가변시키면서, 각 상황에 맞게 수식 7, 8을 이용하여 문맥 전환 오버헤드를 적용하였다.

2. 태스크 스케줄링

일반적으로 RM 스케줄링 기법에서 시간당 CPU 사용률(U, Utilization)을 계산하여 프로세스들이 이상 없이 수행할 수 있는 지를 다음 수식 5를 통해 알아 볼 수 있다[9].

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(\sqrt[n]{2} - 1) \quad (5)$$

(U : CPU 사용률, C : 프로세스의 수행 시간, T : 프로세스의 수행주기, n:스케줄 된 프로세스들의 수)

프로세스들의 사용률을 계산하였을 때, 수식 2를 만족한다면 스케줄링이 가능하다고 할 수 있다. 반면 수식 5를 만족하지 못할 경우, 상대적으로 우선순위가 낮은 프로세스들에서 starvation 등의 상태에 빠질 수 있다. 따라서 프로세스들을 스케줄링을 할 때, 수식 2를 만족할 수 있는 범위에서 스케줄링을 하는 것이 좋다. 하지만 수식 5는 충분조건으로 필요조건이 아니기 때문에 더 높은 사용률을 가진다고 하더라도 스케줄링이 불가능하다고 할 수는 없다.

표 1. 태스크 스케줄링
Table 1. Task scheduling

	Exec. time (clock)	Periodic (clock)	Priority
Task 1	1000	4300	0
Task 2	2000	11000	1
Task 3	3000	28000	2
Task 4	4000	36000	3
Task 5	6000	55000	4

본 실험을 위해 5개의 태스크는 각각 다음 표 1과 같이 스케줄링 하였다. CPI(Clock per Instruction)은 1이라 가정하였다. 그리고 스케줄링이 가능한지를 확인하기 위해 스케줄링 된 태스크의 값을 수식 5의 좌변항에 대입하여 계산한 결과 약 0.7417의 값을 얻었고, 5개의 태스크에 대해 수식 2의 우변항의 계산 결과는 약 0.7435로서 등식이 성립함으로 스케줄링이 가능함을 확인하였다. 또한 RM 스케줄링 방식이므로 수행 주기가 가장 짧은 프로세스에 가장 높은 우선순위를 부여하게 되었다.

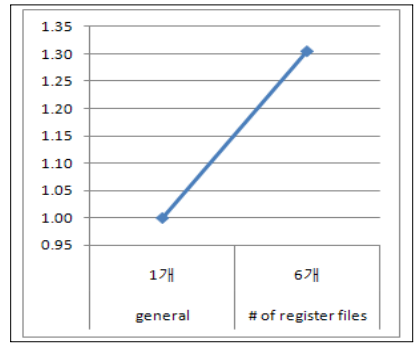

그림 2. 태스크의 수와 레지스터 파일의 수가 같을 경우 IDLE 태스크 수행량 비교
Fig. 2. Compare the two IDLE tasks in the case of the equal number of tasks and register files

실험은 임의적으로 약 160,000,000클럭 동안 작업을 수행한 후 IDLE 태스크의 수행량을 비교하였다.

3. 태스크의 수 ≤ 레지스터 파일의 수

표 1과 같은 태스크 스케줄링 하에서 레지스터 파일을 6개로 하여 실험을 수행하였다. 그 결과 그림 2를 통해 알 수 있듯이, 태스크의 수가 레지스터 파일의 수와 같을 경우, IDLE 태스크가 기존에 비해 약 30%정도 더 많이 실행되었음을 확인할 수 있었다. 이는 각 태스크가 수행되면서 발생하는 문맥 전환에 대한 오버헤드가 그만큼 감소하여 각 태스크가 빨리 작업을 마치고 IDLE 태스크가 진행되는 경우가 늘어났다는 것을 의미한다. 따라서 문맥 전환 오버헤드가 평균적으로 30%정도 감소했다고 볼 수 있다.

4. 태스크의 수 > 레지스터 파일의 수

마찬가지로 표 1과 같은 태스크 스케줄링 하에서 레지스터 파일을 1개씩 추가하며 실험을 수행하였다. 추가적으로 같은 태스크 스케줄링을 적용하되 각 태스크들의 수행 시간 비율을 조절하여 다음 세 가지 상황에서의 실험을 하였다. 첫 번째는 태스크들의 수행 시간을 상대적으로 길게 하여 문맥 전환 오버헤드의 영향을 10%미만으로 한 경우이다. 즉, 오버헤드의 영향이 적은 경우라고 할 수 있다. 두 번째는 일반적으로 알려진 10%~20%정도의 오버헤드를 가지는 경우이다. 이 경우는 일반적인 상황에 적용할 수 있다. 마지막으로 약 30%의 문맥 전환 오버헤드를 가지는 경우이다. 이 경우, 수행 시간이 짧은 태스크들이 실행 될 경우로서 오버헤드의 영향이 큰 경우이다.

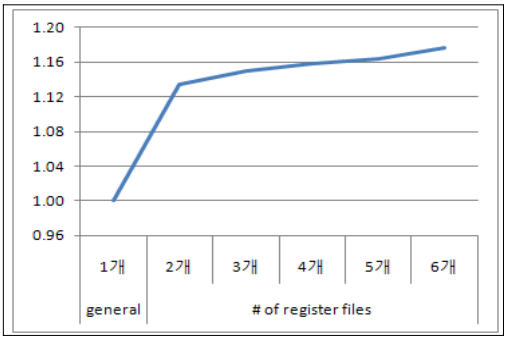

그림 3. 일반적인 문맥 전환 오버헤드를 가질 경우
Fig. 3. General overhead of context switch

태스크의 수가 레지스터 파일의 수보다 많아질 경우, 각 상황에 따라 레지스터 파일을 선택하는 과정이 오버헤드로서 추가된다. 그 결과 그림 3에서 나타난 것처럼 평균적으로 16%정도 오버헤드가 감소된 것을 확인할 수 있었다. 특이할만한 사항은, 레지스터 파일의 수가 늘어날수록 문맥 전환의 오버헤드가 감소하고 있지만 레지스터 파일이 2개일 경우 기존에 비해 가장 큰 효율을 보인다는 것이다.

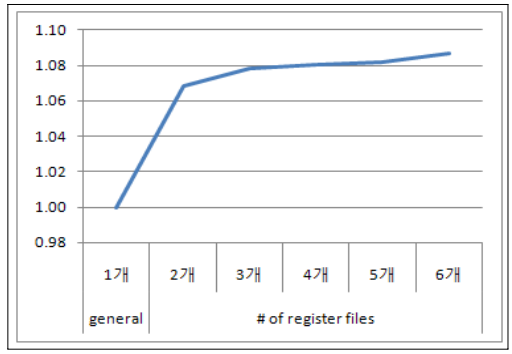

그림 4. 문맥 전환 오버헤드의 영향이 적을 경우
Fig. 4. In case of low overhead of context switch for tasks

하지만 그림 4에서처럼 문맥 전환 오버헤드의 영향이 적은 경우에는 전체적으로 큰 성능 향상을 얻을 수는 없었지만, 마찬가지로 기존의 하드웨어에 비해 전체적으로 약 8%의 오버헤드 감소를 얻을 수 있음을 확인할 수 있다.

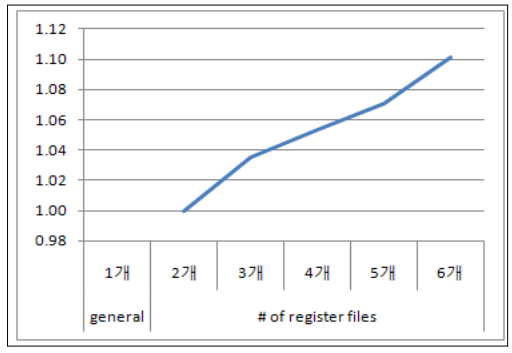

그림 5. 문맥 전환 오버헤드의 영향이 큰 경우
Fig. 5. In case of high overhead of context switch for tasks

다음으로 태스크들의 수행시간이 상대적으로 짧아서 문맥 전환의 오버헤드에 영향이 큰 경우, 그림 5과 같은 결과를 얻을 수 있었다. 레지스터 파일이 1개인 기존의 하드웨어의 경우, 같은 스케줄링 하에서 문맥 전환 오버헤드로 인해 starvation 현상이 발생하여 실행에 오류가 발생했다. 하지만 레지스터 파일이 2개가 될 때부터 starvation 현상이 없어지

고 정상 실행이 가능해졌다. 이는 문맥 전환 오버헤드가 감소하게 되면서, 불가능했던 스케줄링이 가능해졌음을 나타내는 것이다. 이는 성능 향상으로 인해 좀 더 이상적인 환경에 가깝게 스케줄링이 가능함을 나타낸다. 일반적인 상황에서 수행이 불가능했기 때문에 그림 5의 결과는 레지스터 파일이 2개일 경우를 기준으로 하여 성능 향상을 비교하였다. 이 경우 앞의 두 경우보다 레지스터 파일 추가에 따른 성능 향상이 더 높아졌음을 볼 수 있고, 이는 문맥 전환 오버헤드의 영향이 클수록 성능이 더 향상됨을 나타낸다.

5. 정 리

레지스터 파일을 태스크의 수만큼 추가할 경우 문맥 전환 오버헤드가 약 30%정도 감소하였고, 소수의 레지스터 파일만을 추가하더라도 평균적으로 약 8~16%의 오버헤드 감소를 시뮬레이션을 통해 확인할 수 있었다. 특히 태스크들의 수행시간이 상대적으로 짧아 문맥 전환 오버헤드의 영향이 클 경우 레지스터 파일이 추가됨에 따라 더 나은 성능 향상을 확인할 수 있었다. 오버헤드 감소로 인해 IDLE 태스크의 수행 시간이 늘어났으며 이는 좀 더 이상적인 환경에 가깝게 각 태스크들을 스케줄링을 할 수 있음을 확인하였고, 그에 따라 프로세스의 효율을 더욱 높일 수 있음을 나타낸다.

빠른 응답 시간이 요구되면서 태스크의 수를 예측할 수 있는 소규모 RTOS 환경에서는 레지스터 파일을 태스크의 수만큼 추가하여 문맥 전환 오버헤드를 감소시켜 빠른 응답 시간을 보장 할 수 있을 것이다. 반면 일반적인 RTOS 환경의 경우, 태스크의 수가 유동적으로 변할 수 있으므로 레지스터 파일을 적절한 수만큼 추가함으로써 하드웨어 추가에 의한 비용을 최소화하면서 문맥 전환 오버헤드를 줄일 수 있을 것이다.

V. 결론 및 추후 작업

문맥 전환 오버헤드의 감소는 멀티태스킹의 스케줄링을 용이하게 하고, 멀티태스킹 환경에서 빠른 응답 시간을 보장한다. 본 논문에서는 문맥 전환에서 발생하는 오버헤드를 줄이기 위해 기존의 하드웨어 구조에 레지스터 파일을 추가하는 방법을 제안하였다. 레지스터 파일의 추가는 우선순위가 높은 몇 개의 태스크들이 레지스터 파일을 점유할 수 있게 하여 각 태스크들의 문맥을 저장하고 복원하는 과정을 줄여 문맥 전환을 보다 빠르게 할 수 있게 하였다. 실험을 위해 RM 스케줄링과 다양한 환경을 적용한 시뮬레이터를 제작하였고, 제작한 시뮬레이터에 여러 가지 경우의 시나리오를 적용하여 실험을 하였다. 그 결과 문맥 전환의 오버헤드는 레지스터 파일을 태스크의 수와 동일하게 추가할 경우 약 30% 감소하였고, 몇 개의 레지스터 파일만을 추가하여 태스크의 수가 많을 경우에는 평균적으로 약 8~16% 감소하였다.

하지만 이 시뮬레이터는 실제 환경과는 다소 차이가 있을 것이다. 임의의 가상환경이기 때문에 실제 하드웨어에 적용하여 RTOS를 실행하였을 때와는 다소 차이가 있을 수 있다. 그렇기 때문에, 우리는 보다 정확한 실험을 위해 본 논문에서 제안한 레지스터 파일을 추가한 하드웨어를 HDL (Hardware Description Language)을 이용해 설계할 계획이다. 그리고 FPGA를 이용하여 실제 프로세서와 유사한 환경을 만들 것이다. 구현된 하드웨어에 실제 RTOS를 적용하여 RM 스케줄링 하에서 본 실험을 재현해보고 성능을 평가할 것이다.

참고문헌

[1] 임채덕 등, "임베디드 소프트웨어 기술동향 및 산업발전 전망", 정보통신연구진흥 제4권 제3호, 정보통신연구진흥원, 2002.

[2] Operating System Specification 2.2.1, 2003, http://www.osek-vdx.org.

[3] Qualcomm, Operating System REX, L4, http://www.qualcomm.com.

[4] CISCO, Operating System IOS, http://www.cisco.com.

[5] J. A. Stankovic. R. Rajkumar, "Real-time operating system", Real-Time Systems, Kluwer Academic Publishers, Vol.28, pp. 237-253, 2004.

[6] P. A. Laplante, Real-Time systems Design and Analysis: An Engineer's Handbook, Second Edition, 1996.

[7] Hassan Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, 2000.

[8] D. Tsafrir, "The context-switch overhead

inflicted by hardware interrupts (and the enigma of do-nothing loops)", Experimental computer science on Experimental computer science, 2007.

[9] D. B. Stewart and Michael Barr, "Introduction to rate monotonic scheduling", Embedded Systems Programming, 2002.

[10] J. S. Snyder, D. B. Whalley, and T. P. Baker, "Fast context swtiches: compiler and architectural support for preemptive scheduling", Microprocessors and Microsystems, pp. 35-42, 1995.

[11] X. Zhou and P. Petrov, "Rapid and low-cost context-switch through embedded processor customization for real-time and control applications", in Proceedings of the 43rd annual Conference on Design Automation, pp. 352-357, 2006.

[12] R. Alverson, D. Callahan, D. Cummings, B. koblenz, A. Porterfield, and B. Smith, "The tera computer system", in Proceedings of the 1990 International Conference on Supercomputing, pp. 1-6, 1990.

[13] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson, "The next generation of intel ixp network processors", Intel Technology Journal, Vol.6, 2002.

[14] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded sparc processor", Micro, IEEE, Vol.25, No.2, pp. 21-29, 2005.

[15] P. R. Nuth and W. J. Dally, "The named-state register file: Implementation and performance", in IN Proc. 1st Intl Symp. on High-Performance Computer Architecture HPCA, pp. 4-13, 1995.

[16] Jean J Labrosse, uC/OS-II: The Real-Time Kernel, R&D Books[M], 1998.

저 자 소 개

김 종 웅

2009년 : 경북대 전자전기컴퓨터공학부 학사.
현재, 경북대 대학원 전자전기컴퓨터학부 재학.
관심분야 : 임베디드 소프트웨어, Real-time OS.
Email : ber.woong@gmail.com

조 정 훈

1996년 KAIST 전기및전자공학과 학사.
1998년 KAIST 전기및전자공학과 석사.
2003년 KAIST 전자전산학과 박사.
2005년 하이닉스반도체 선임연구원.
현재, 경북대학교 전자공학부 조교수.
관심분야 : 임베디드시스템 최적화, 컴파일러, 소프트웨어 최적화, HW/SW Codesign.
Email : jcho@ee.knu.ac.kr