내장형 시스템 동적 메모리 할당 기법의 시스템 수준 성능에 관한 정량적 분석

(Quantitative Analyses of System Level Performance of Dynamic Memory Allocation in Embedded Systems)

박 상 수 † 신 현 식 ††
(Sangsoo Park) (Heonshik Shin)

요 약 내장형 컴퓨터의 규모가 커지고 기능이 복잡해짐에 따라 동적 메모리 할당 기법은 전체 시스템의 성능을 좌우하는 중요한 요인으로 등장하였다. 본 논문의 목적은 내장형 시스템에서 동적 메모리 할당 기법을 사용할 때 하드웨어, 소프트웨어 구성에 따른 성능을 측정하는데 있다. 기존의 연구가 운영체제를 탑재하지 않은 단일 스레드의 단일 메모리 주소 공간을 갖는 시스템을 대상으로 한 반면 본 논문은 실제 환경과 같이 리눅스 운영체제를 탑재한 내장형 시스템을 사용한다. 이러한 시스템 기반에서 소프트웨어의 각 계층과 하드웨어 설계 인자의 변화에 따른 동적 메모리 할당의 수행시간을 실험적으로 분석하였다. 본 논문의 정량적인 성능분석 결과는 시스템 설계자에게 유용한 데이터를 제공함으로써 보다 효율적인 고성능 저전력 내장형 시스템의 구현을 가능하게 할 것이다.

키워드 : 내장형 시스템, 동적 메모리 할당, 운영체제, 시스템 설계

Abstract As embedded system grows in size and complexity, the importance of the technique for dynamic memory allocation has increased. The objective of this paper is to measure the performance of dynamic memory allocation by varying both hardware and software design parameters for embedded systems. Unlike current performance evaluation studies that have presumed the single threaded system with single address space without OS support, our study adopts realistic environment where the embedded system runs on Linux OS. This paper contains the experimental performance analyses of dynamic memory allocation method by investigating the effects of each software layer and some hardware design parameters. Our quantitative results can be used to help system designers design high performance, low power embedded systems.

Key words : Embedded system, dynamic memory allocation, operating system, system design.

1. 서 론

내장형 시스템과 일반 컴퓨터 시스템은 모두 응용에서 수행하고자 하는 기능성을 만족해야 하지만 내장형 시스템은 특히 응용에 종속적인 실시간성, 성능, 전력 소모, 제조비용, 제조시기(time-to-market) 등의 부가적인 제약을 갖는다. 과거 대부분의 내장형 시스템은 엄중한 시간 제약성을 갖는 실시간 시스템으로 수행 태스크들의 시간적인 특성만이 아니라 각 태스크가 시작되어 종료될 때까지 필요한 모든 메모리 공간이 사전에 명시되어 있는 응용을 수행하였다[1]. 이러한 내장형 시스템은 비교적 단순한 메모리 관리 기법을 사용하여 시스템 초기화 시에 필요한 메모리 공간을 정적으로 할당하는 것이 가능하였다. 그러나 내장형 시스템이 점점 복잡해지고 규모가 증대됨에 따라 이러한 메모리 할당 명시가 불가능한 응용이 많아졌다. 이러한 응용에서는 정적 메모리 할당 외에 태스크가 수행되는 동안에 필요한 메모리 공간을 할당 받고 사용 후에는 반납하는 동적 메모리 할당 기법이 필요하다[2].

일반적인 컴퓨터 시스템에서의 동적 메모리 할당 기법은 평균적으로 짧은 메모리 할당 시간과 높은 메모리 공간 효율성을 목표로 한다. 또한, 엄중한 시간 제약성을 갖는 실시간 시스템의 경우 평균적인 응답 시간이나 공간 효율성 보다는 최악 경우의 메모리 할당 시간이 예측 가능해야 하고 항상 메모리 할당이 가능해야 하는

† 학생회원 : 서울대학교 전기·컴퓨터공학부
sspark@cslab.snu.ac.kr
†† 종신회원 : 서울대학교 전기·컴퓨터공학부 교수
shinhs@snu.ac.kr
논문접수 : 2005년 3월 30일
심사완료 : 2005년 9월 5일

제약을 가지고 있다. 기존의 연구에서는 이러한 동적 메모리 할당 기법을 위한 다양한 알고리즘과 대응되는 성능분석을 수행하였다[2-5]. 그러나 이러한 연구들은 대부분 운영체제의 영향을 배제하고 하나의 메모리 주소 공간을 갖는 시스템에서 하나의 응용이 수행되는 것을 가정하였다.

최근의 내장형 시스템은 하드웨어와 소프트웨어 기술의 비약적인 발전으로 멀티미디어, 네트워크 등의 보다 복잡한 다중의 응용을 동시에 수행하는 것이 일반화되어 효율적인 운용을 위해 운영체제를 탑재하고 있다. 또한, 이러한 내장형 시스템은 과거의 엄격한 시간 제약성보다는 높은 처리량, 낮은 제조비용 등을 요구하고 있으며 이러한 제약은 시스템 설계 인자에 의해 좌우된다[6,7]. 따라서, 동적 메모리 할당 및 반납의 성능에 대한 운영체제에 의한 영향을 분석하는 것과 더불어 내장형 시스템에서 다양한 하드웨어 설계 인자와 이에 대응하는 소프트웨어 컴포넌트를 변경하면서 여러 내장형 응용에 대한 성능을 분석할 필요가 있다.

본 연구에서는 실제 환경과 같은 운영체제를 탑재한 시스템에서 설계 인자를 변화시켜가면서 소프트웨어와 하드웨어의 전체 시스템 성능에 대한 동적 메모리 기법의 영향을 정량적으로 측정한다. 이러한 실험 결과와 그에 따른 성능분석의 데이타를 이용하여 효율적인 하드웨어와 소프트웨어의 설계 인자를 결정하는데 중요한 단서를 제공한다. 즉, 현재 각종 내장형 기기에서 널리 사용되고 있는 환경을 가정하고 이들의 시스템 설계 시에 고려되는 보편적인 소프트웨어 컴포넌트 및 하드웨어 설계 인자들을 다루고 있기 때문에 비용 효율적인 시스템 설계를 위한 설계 공간 탐색(design space exploration)이나 소프트웨어의 수정을 통한 성능 향상에 필요한 정량적인 데이타를 제공할 수 있다. 예를 들어, 휴대용 비디오 재생기에서 캐쉬 크기를 늘이거나 CPU 클록 주파수를 높일 경우 재생 성능이 향상 된다는 것은 쉽게 예측할 수 있다. 그러나, 각각의 설계 인자를 변화시켰을 때 개별적인 개선 비율을 알 수 있다면 캐쉬 크기를 늘일 것인지 혹은 CPU 클록 주파수를 높일 것인지 결정할 수 있다. 또한, 이러한 성능분석 결과는 소프트웨어 설계 시에도 도움이 될 수 있다. 실제로 하드웨어 설계 인자 중 MMU 탑재 여부에 따라 문맥교환 성능을 측정하는 기존의 벤치마크 프로그램을 사용한 측정 결과로 소프트웨어 설계에서 MMU를 제외한 개발 사례가 있다[8]. 따라서, 논문에서 제시하는 분석 결과는 최근 유비쿼터스 디바이스 등 이동성과 휴대성이 강조되는 내장형 시스템에 효과적으로 적용되어 고성능 및 저전력 특성을 만족시켜야 하는 차세대 내장형 시스템을 체계적으로 설계하는데 도움이 될 것이다.

본 논문의 구성은 다음과 같다. 2장에서 관련 연구에 대해 기술하고, 3장에서 본 연구의 시스템 모델을 소개한다. 4장에서는 실험 결과 및 성능을 분석한다. 마지막으로 5장에서는 본 논문의 결론을 기술한다.

2. 관련 연구

동적 메모리 할당 기법의 성능분석에 관한 일반적인 연구 방법은 알고리즘 자체의 분석을 통해 알고리즘의 복잡도를 구하고 벤치마크 응용을 수행하여 실험적으로 검증한다. Puaut[2]는 대표적인 동적 메모리 할당 알고리즘들에 대해서 각각 최악 경우의 시간 복잡도를 구하고 벤치마킹 응용을 수행하여 분석적으로 구해진 최악 경우의 수행시간과 실제 관찰된 수행시간을 비교하였다. 한편, 각각의 동적 메모리 할당 알고리즘이 시간 제약성을 가지는 실시간 시스템에서 적용될 때의 문제들에 대해서도 기술하였다. Donahue et al.[3]은 동적 메모리 할당 알고리즘의 하나인 버디 알고리즘(Buddy system algorithm)의 성능을 분석하고 실시간 시스템에 적합하도록 개선하였다. Masmano et al.[4,5]은 기존의 동적 메모리 할당 기법들의 성능을 비교하고 비트맵을 이용한 실시간 알고리즘을 제안하고 성능분석을 수행하였다.

그러나, 이러한 연구들은 알고리즘 자체의 성능분석에만 국한되어 있기 때문에 하나의 응용이 하나의 메모리 주소공간을 사용하는 단일 스레드(single-thread) 환경을 가정한다. 따라서, 다중 스레드 (multi-thread)를 지원하는 시스템에서 여러 스레드가 동적 메모리 할당을 요청할 경우 필요한 동기화에 대한 부가적인 분석이 필요하다. 이 문제와 관련해서 Dice and Garthwaite[9]는 이러한 스레드 간 동적 메모리 할당에서 발생하는 동기화 문제에 대해 기술하고 동기화에 따른 수행시간 지연 문제를 최소화하도록 동적 메모리 할당 알고리즘을 수정하고 실험을 통하여 검증하였다. 또한, 내장형 시스템의 경우 하드웨어 구성에 따라서 시스템의 성능(수행시간, 소비 전력)이 매우 크게 변할 수 있는데[7], Grunwald[10]는 이러한 점을 이용하여 하드웨어의 캐쉬 구성에 따른 효율적인 동적 메모리 할당 기법을 제안하였다.

이와 같이 이미 기존의 여러 연구에서는 일반적인 서버 시스템이나 데스크탑 시스템 환경에서 동적 메모리 할당 및 반환 관련 함수들의 성능을 제시하고 있고 이러한 성능의 변화 추이는 쉽게 유추 가능하다. 그러나, 성능 변화 추이를 예측할 수 있다고 하더라도 내장형 시스템 환경에서 내장형 응용의 정량적인 연구 방법은 여전히 필요하다.

본 연구에서는 기존 연구에서 제시하는 것과 유사한 성능 평가 척도를 사용하여 운영체제를 탑재한 내장형

시스템에서 동적 메모리 할당 기법이 사용되었을 때 발생할 수 있는 문제인 운영체제가 탑재됨에 따라 발생하는 소프트웨어 계층 및 동기화에 따른 성능분석과 하드웨어 구성에 따른 성능분석에 초점을 맞춘다.

3. 시스템 모델

3.1 하드웨어 모델

내장형 시스템의 하드웨어는 목표 시스템의 제약 사항을 만족시킬 수 있도록 설계 되어야 한다. 실제로 시스템의 제조비용과 성능은 CPU 코어의 선택에 의해 크게 영향을 받기 때문에 ARM, MIPS 등 대부분의 내장형 시스템을 위한 마이크로프로세서들은 다양한 사양의 CPU 제품 군을 제공하고 있다[7]. 하드웨어 설계를 위한 대표적인 인자로는 CPU 및 메모리 클록 주파수, 캐쉬 크기, MMU(Memory Management Unit) 탑재 유무 등이 있다. 시스템 설계자는 목표 시스템의 성능, 장치 크기, 전력 소모량 등의 제약이 주어졌을 때 이러한 설계 인자를 적절히 선택하여 비용을 최소화 한다.

본 논문의 하드웨어 플랫폼으로는 대표적인 내장형 시스템용 플랫폼인 ARM[11]을 채택하였다. 또한, 운영체제를 구동하기 위한 최소한의 컴포넌트인 CPU 코어, RAM, 인터럽트 제어기, 타이머를 사용하며, CPU 및 메모리 클록 주파수, 캐쉬 크기 및 MMU 탑재 여부를 하드웨어 설계 인자로 하고 이러한 인자들을 변화하며 실험하기 위해 ARM9E를 목표 플랫폼으로 사용한다. 그림 1은 본 연구의 하드웨어 시스템 모델을 나타낸다. MMU가 있는 CPU의 경우 ARM926EJ-S를 사용하고 MMU가 없는 CPU의 경우 ARM946E-S를 사용한다. 타이머는 100Mhz의 해상도를 갖는다.

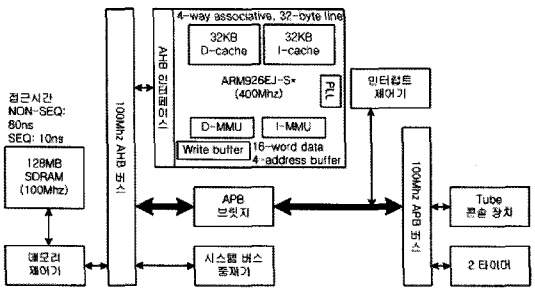


그림 1 하드웨어 시스템 모델(MMU가 없는 CPU의 경우 ARM946E-S를 사용)

3.2 소프트웨어 모델

본 연구의 소프트웨어 플랫폼으로는 리눅스(Linux)를 사용하였다. 리눅스는 소스 코드가 공개되어 있고 다양한 하드웨어 플랫폼과 입출력 장치를 지원하고 있기 때문에 일반 컴퓨터 시스템뿐만이 아니라 내장형 시스템의 운영체제로도 각광 받고 있다. 또한, 비록 소프트웨어 구조는 MMU의 유무에 따라 다르지만 MMU가 있는 CPU와 MMU가 없는 CPU 모두 동작 가능하다.

그림 2는 내장형 응용 수행 시의 동적 메모리 관련 소프트웨어 컴포넌트를 도식한 것이다. 그림 2와 같이 운영체제는 크게 커널, 라이브러리, 그리고 응용 프로세스의 세 계층으로 구성된다. 내장형 시스템은 일반 컴퓨터 시스템과 같이 시스템 초기화, 응용 수행, 시스템 종료의 순으로 수행되지만 초기화와 종료가 매우 드물게 발생한다. 그러므로 본 연구의 성능분석은 실제로 시스템이 응용을 수행하는 기간인 그림 2의 점선 안의 부분에 한정한다.

소프트웨어 플랫폼에 따라서 메모리 관리 시스템의 구현은 상이하지만 대개 유사한 기본구조를 갖는다. 그림 3은 본 연구의 소프트웨어 플랫폼에서 동적 메모리 관련 함수의 소프트웨어 계층별 흐름도이다. 그림 3과 같이 응용이 수행되는 과정에 메모리 공간이 필요할 경우 응용은 라이브러리의 동적 메모리 할당 인터페이스인 *malloc()*을 통해 동적 메모리 공간을 요청한다. 커널

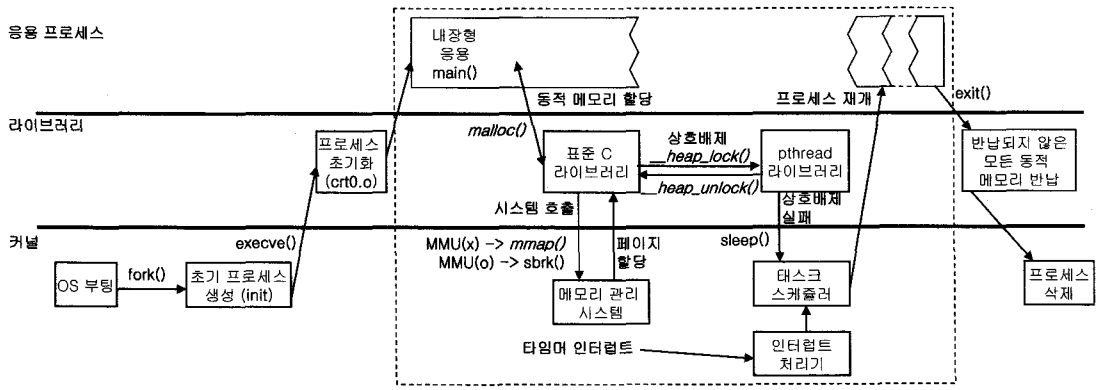


그림 2 내장형 응용 수행 시의 소프트웨어 컴포넌트

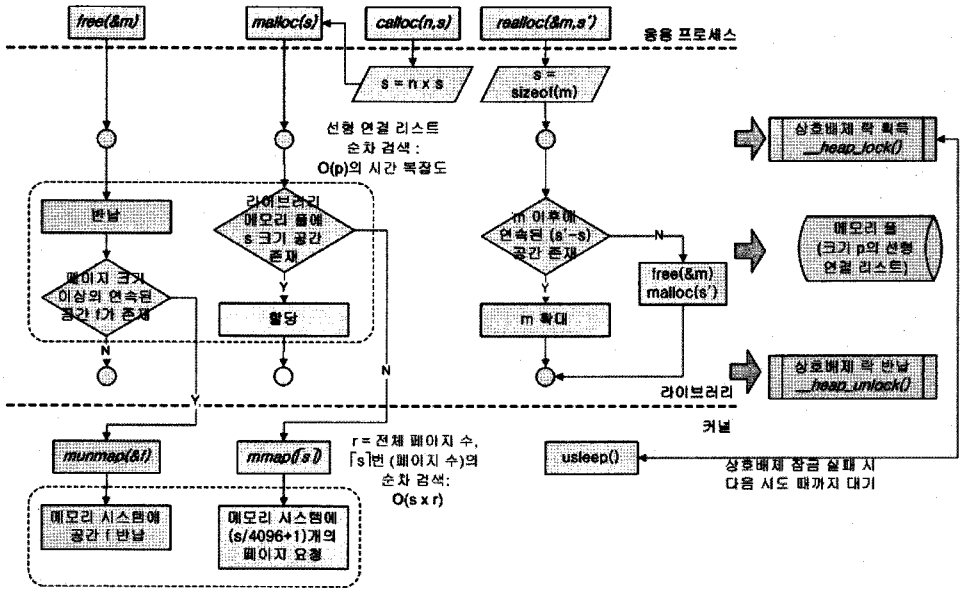


그림 3 동적 메모리 관련 함수의 소프트웨어 계층(linux-2.4.21-uc0, uClibc-0.9.19)

은 시스템의 물리적인 메모리 공간의 효율적인 관리를 위해서 메모리 공간을 페이지(4096바이트)라고 하는 기본 단위로 관리하는데, 라이브러리는 응용으로부터의 동적 메모리 할당을 처리하기 위해 커널에 시스템 호출인 *mmap()*을 사용하여 요청 받은 크기 이상의 물리적인 메모리 공간을 할당 받고 이 공간에서 응용으로부터의 동적 메모리 할당 요청에 필요한 공간을 재할당해준다.

이후에 발생되는 응용의 동적 메모리 할당 요청에서는 커널로부터 할당 받은 물리적인 메모리 공간(메모리 풀)에 여분의 공간이 있을 경우 이 공간에서 요청을 처리한다. 따라서 라이브러리 계층에서는 이러한 동적 메모리 할당 상태를 저장할 수 있는 자료 구조가 필요하고 리눅스는 다중 스레드를 기반으로 하기 있기 때문에 상호배제 락(mutex lock) 획득을 통해 자료 구조의 일관성을 유지해야 한다. 다중의 스레드들이 동적 메모리 할당을 요청할 경우 상호배제에 의해 봉쇄된 스레드는 수행이 중지되고 특정 시간 동안 대기 후 태스크 스케줄러에 의해 깨어나 다시 시도한다.

또한, 동적 메모리를 반납하는 경우에도 마찬가지로 응용이 라이브러리 인터페이스인 *free()*를 통해 할당 받은 메모리를 반납하고 라이브러리에서는 반납 받은 메모리 공간과 현재 유지하고 있는 메모리 풀에서 기존에 반납된 공간을 결합시킨다. 만약 결합된 공간에서 페이지 크기 이상의 공간이 확보되었을 때 시스템 호출인 *munmap()*을 사용하여 커널에 물리적인 메모리 공간을 반납한다.

4. 실험 결과 및 성능분석

4.1 실험 환경

목표 플랫폼이 준비되기 전에 시스템의 성능분석을 하기 위해서는 시스템 수준의 시뮬레이션이 필요하다. 또한, 성능을 분석하고자 하는 내장형 응용뿐이 아니라 운영체제까지 동시에 시뮬레이션이 가능해야 한다. 이를 위해 RTL 수준의 시뮬레이터[7]나 트랜잭션 수준의 시뮬레이터[12]가 초기 시스템 설계 단계에서 성능분석 도구로 종종 이용되지만, 시뮬레이션 시간이 매우 긴 단점이 있다. 본 연구에서는 ARM사의 ISS(Instruction Set Simulator)인 ARMulator[13]를 사용한다. ARMulator는 ARM에서 제공하는 대부분의 CPU 모델을 제공하며 API 형태로 메모리, 버스, 주변 장치 등의 모델을 구현할 수 있는 인터페이스를 제공한다. 또한, 사이클 별로 시스템의 상태를 확인 및 수정이 가능하기 때문에 소스 코드 수준의 디버깅을 하거나 성능 프로파일링 하는데 유용하다. 시스템의 성능분석 면에서 100%의 사이클 정밀도를 가지고 있진 않지만 성능분석에 충분한 정밀도와 빠른 시뮬레이션 시간을 제공한다. 이러한 여러 장점들로 ARMulator는 하드웨어 설계 공간 탐색[14]과 하드웨어/소프트웨어 상호 검증[15] 등에 널리 사용되고 있다. 또한, 확장 가능한 구조를 갖고 있기 때문에 성능분석 외에 저전력 연구[16]에도 활용되고 있다.

리눅스 운영체제에서 MiBench 내장형 응용 벤치마크를 수행하기 위해서 우선 리눅스를 ARMulator에 이식을 하였고 이식된 리눅스 상에서 응용을 수행하였다.

그림 4는 ARMulator 상에서 리눅스와 응용이 수행되고 있는 예를 보여준다.

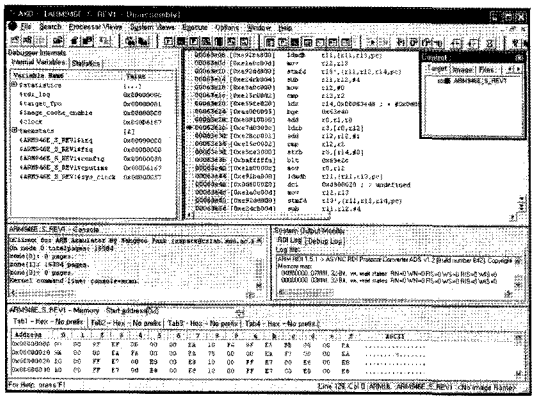


그림 4 ARMulator 상에서 리눅스와 벤치마크 응용 수행 예

또한, 표 1은 소프트웨어 개발 환경을 요약한다.

표 1 소프트웨어 개발 환경

CPU	커널	라이브러리	컴파일러	컴파일 옵션
ARM926E-S	linux-2.4.21-rmk1	glibc-2.2.3	gcc-2.95.3	-O3
ARM946E-S	Linux-2.4.21-uc0	uClibc-0.9.19	gcc-2.95.3	-O3

소프트웨어 개발 환경에서 MMU의 탑재 여부에 따라 서로 다른 라이브러리를 사용한다. uClibc는 GNU C 라이브러리(glibc)와는 달리 MMU가 탑재되지 않은 CPU에서도 동작하도록 개발된 C 라이브러리이다. 또한, 리눅스가 동작하기 위해서는 수행할 응용의 수행 파일 등을 저장할 수 있는 파일 시스템이 있어야 하는데 RAM의 일부를 MTD(Memory Technology Device)를 사용하여 블록 장치로 사용하고 리눅스 기본 파일 시스템(ext-2)으로 초기화 하였다.

내장형 시스템의 동적 메모리 할당 기법의 성능분석을 위해서는 내장형 시스템에서 수행될 대표적인 내장형 응용이 정의되어야 한다. 이를 위해 내장형 마이크로프로세서 벤치마크 컨소시엄의 EEMBC[17], 미시간 대학의 MiBench[18] 등의 내장형 시스템을 위한 벤치마크 도구가 이미 개발되어 있다. 본 연구에서는 리눅스를 기반으로 개발 되었고 소스 코드가 공개되어 목표 시스템에 이식 가능한 MiBench를 성능분석을 위한 응용으로 채택하였다. MiBench는 내장형 응용을 크게 자동차, 공업용 제어, 가전 제품, 네트워크, 사무, 보안, 통신 등으로 분류하고 각 분류 별로 이에 적합한 벤치마크 응용을 제공한다.

그림 5는 내장형 응용에서 동적 메모리 사용 특성을 보이기 위해 전체 수행시간에서 동적 메모리 관련 수행시간의 비율을 보여주고 있다. 많은 벤치마크 응용에서 동적 메모리 할당이 1% 미만의 수행시간 비율을 나타내지만 3-10%의 비율을 가지는 응용도 다수 포함되어 있으며 최대의 경우 38%의 비율을 가지는 응용(typeset)도 존재한다.

4.2 운영체제에 의한 영향

4.2.1 소프트웨어 계층별 시간 복잡도에 따른 성능분석

동적 메모리와 관련된 많은 연구에서는 일반적으로 할당 크기에 따라 메모리 할당 및 반납의 수행시간을 측정함으로써 알고리즘의 복잡도를 분석한다. 이 때 운

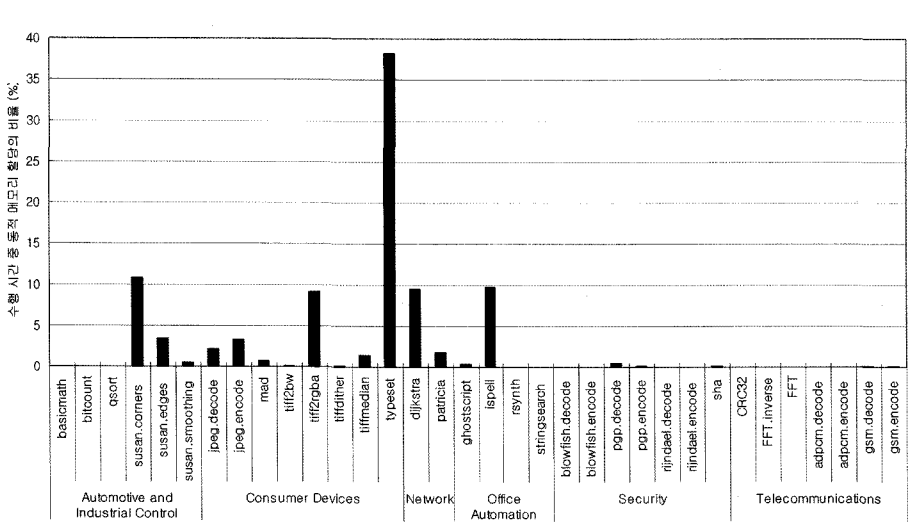


그림 5 MiBench의 각 응용별 수행시간 중 동적 메모리 할당의 비율

영체제 탑재에 따른 영향을 살펴보기 위해 응용 외에 추가된 소프트웨어 계층에 대한 복잡도를 구하고 실험을 통해 실제 측정된 수행시간과 비교하였다. 측정 결과 동적 메모리 할당의 수행시간은 커널의 페이지 할당 크기에 정비례하였고 동적 메모리 반납의 수행시간은 동기화 오버헤드를 제외하면 일정하였다.

응용에서 대표적으로 사용되는 함수로는 동적 메모리 할당을 담당하는 *malloc(s)*과 반납을 담당하는 *free()*가 있다(s: 바이트 단위의 메모리 할당 크기). 이 두 함수는 그림 3에서 볼 수 있듯이 (1) 라이브러리 계층에서 크기 p인 메모리 풀의 선형 연결 리스트 순차 검색을 하고 (2) 커널 계층에서 전체 페이지 수가 r인 물리 메모리의 선형 연결 리스트 순차 검색을 라이브러리 계층에서 요청한 페이지 수 "(s/4096) + 1"번을 반복 수행한다.

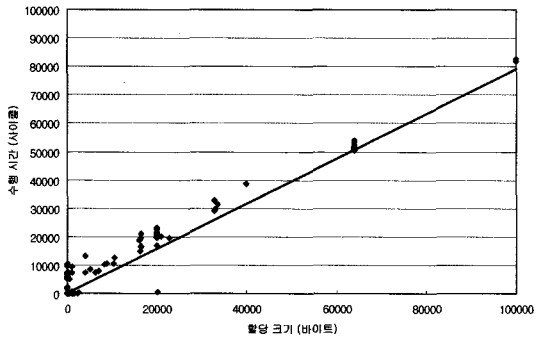
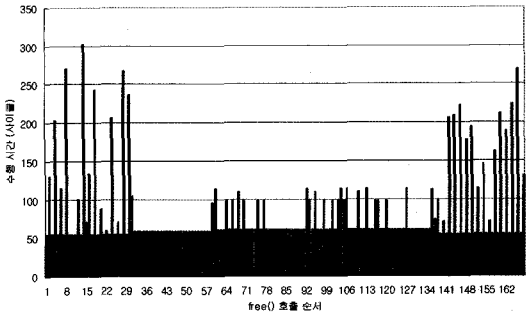


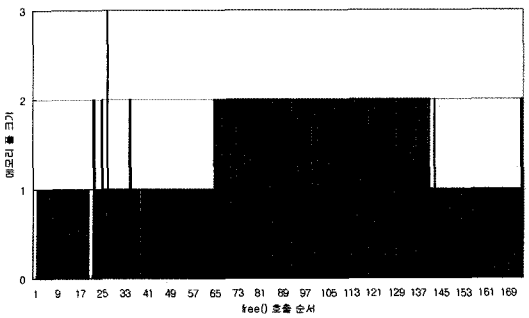
그림 6 *malloc()* 함수의 크기 인자 값에 따른 수행시간 분포

따라서, *malloc()*의 시간 복잡도는 O(p) + O(s · r) + C와 같다(C: 상호배제 락 오버헤드). 이때, 내장형 시스템의 물리 메모리 크기는 일반 컴퓨터 시스템보다 작고 한정되어 있으므로 전체 페이지 수인 r은 상수로 가정할 수 있다. 그림 6은 *malloc()* 함수의 크기 인자 값에 따른 수행시간 분포를 나타낸 것으로 수행시간은 크기 인자 값에 비례하는데 이는 인자 p 혹은 s가 동적 메모리 할당 알고리즘의 수행시간에 지배적인 영향을 준다는 것을 알 수 있다. 그러나 아래의 *free()*의 성능 분석을 보면 라이브러리의 메모리 풀의 크기 p는 대부분의 경우 수 개 이하이다. 따라서, 동적 메모리 할당 시 커널의 페이지 할당 크기 s에 성능이 좌우되는 것을 확인 할 수 있다.

그림 7(a)는 pgp.decode 벤치마크 응용이 수행될 때 호출되는 *free()*의 수행시간을 순서대로 도식한 것이다. 대부분의 호출에서 수행시간이 50 사이클이지만 수 개의 호출에서 수행시간이 스윙(swing)하는 것을 볼 수 있다. 또한, *free()*는 인자로 반납하고자 하는 메모리 공간의 크기를 명시적으로 주지 않지만 프로파일링을 통해 추적해보면 *malloc()*의 경우와는 다르게 수행시간과 반납하고자 하는 메모리 공간의 크기와는 상관 관계가 뚜렷하지 않은 것으로 나타난다.



(a) 수행시간



(b) 메모리 풀의 크기

그림 7 *free()* 함수의 수행시간 및 대응되는 메모리 풀의 크기(pgp.decode)

그 원인을 분석해보면 표 2는 MiBench 벤치마크의 동적 메모리 관련 각 함수 별 호출 빈도를 나타내는데, 대부분의 내장형 응용들이 할당 받은 동적 메모리 공간을 응용 수행 중간에 반납하는 경우가 상대적으로 적다는 것을 알 수 있다. 표 2에서 볼 수 있듯이 응용이 할당 받는 메모리 공간을 반납하는 경우는 1/3에 불과하며 실제로 시스템 호출을 통해 커널의 물리 메모리로 반납하는 경우는 극소수인 것을 확인할 수 있다. 또한, 그림 7(b)는 그림 7(a)의 일련의 *free()* 호출 시에 라이브러리의 메모리 풀의 크기 p를 추적한 결과이다. 그림 7(b)에서 볼 수 있듯이 메모리 풀의 크기가 최대 3개이고 대부분 1개 혹은 2개일 경우가 대부분이므로 메모리 풀에 대한 선형 연결 리스트의 순차 검색 오버헤드는 매우 작다고 할 수 있다. 따라서, *free()* 수행시간이 스윙하는 경우는 시스템 호출을 통해 물리 메모리로 반납하는 경우이거나 상호배제 락 획득에 실패하여 시간이 지연된 경우이다.

표 2 함수 별 호출 빈도

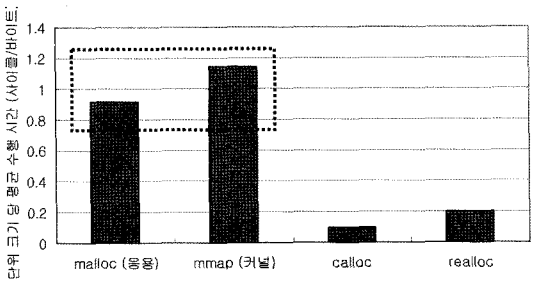
함수	*malloc()*	*free()*	*mmap()*	*munmap()*
빈도	49852	16046	1262	58

위의 실험 결과 및 성능분석에서 응용 수준의 오버헤드에 비해 운영체제 수준의 오버헤드가 현저한 것을 알 수 있다. 따라서, 본 논문에서는 시스템 수준에서 보면 운영체제가 병목점이 될 수 있다는 점을 제시한다. 즉, 기존 연구들에서는 시스템 수준이 아닌 응용 수준의 동적 메모리 알고리즘의 성능분석 및 개선에 초점을 맞추고 있기 때문에 실제 시스템의 성능과 큰 차이가 날 가능성이 있다. 내장형 시스템에서 이러한 문제점에 대한 개선 방안으로는 시스템 초기화 시에 운영체제로부터 충분한 메모리 공간(heap)을 할당 받고 목표 내장형 응용에 특화된 메모리 할당기(customized memory allocator)를 사용하는 방법이 존재한다[19].

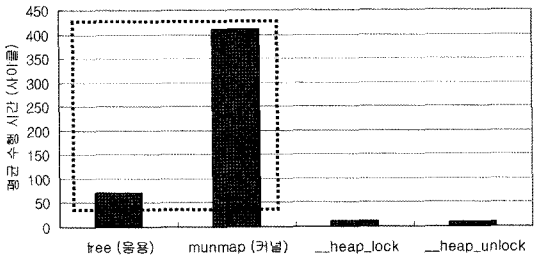
4.2.2 커널과 응용의 함수 별 성능분석

4.2.1절의 성능분석 결과인 응용 수준의 오버헤드에 비해 운영체제 수준의 오버헤드가 현저하다는 점을 확인하기 위해서 커널과 응용의 함수 별 성능을 비교하였다. 이를 위해 각 함수 별 수행시간을 측정하였는데 크기 인자를 받는 경우는 그림 6의 *malloc()*과 같이 할당 받고자 하는 메모리 공간의 크기에 따라서 수행시간이 크게 차이가 날 수 있기 때문에 각 함수가 호출될 때 수행시간을 크기로 나누어 단위 크기당 수행시간을 계산하여 함수 별 성능을 비교하였다.

그림 8에서 볼 수 있듯이 *calloc()*과 *realloc()*은 실제 동적 메모리를 할당하는 알고리즘이 수행되지 않고 다른 함수들을 이용하여 구현되기 때문에 상대적으로 수



(a) 크기 인자를 받는 경우



(b) 크기 인자를 받지 않는 경우

그림 8 함수에 따른 수행시간 비교

행시간이 짧다. 또한, 라이브러리 계층의 함수인 *malloc()*, *free()*에 비해 커널 계층의 함수인 *mmap()*, *munmap()*의 수행시간이 훨씬 길다.

그림 8의 점선 내의 측정 결과에 의하면 커널에서 동작하는 페이지 할당 함수인 *mmap()*의 단위 크기당 수행시간이 라이브러리에서 동작하는 동적 메모리 할당 함수인 *malloc()*의 수행시간 보다 약 24% 길었다. 또한, 동적 메모리 반납의 경우에 커널의 함수가 라이브러

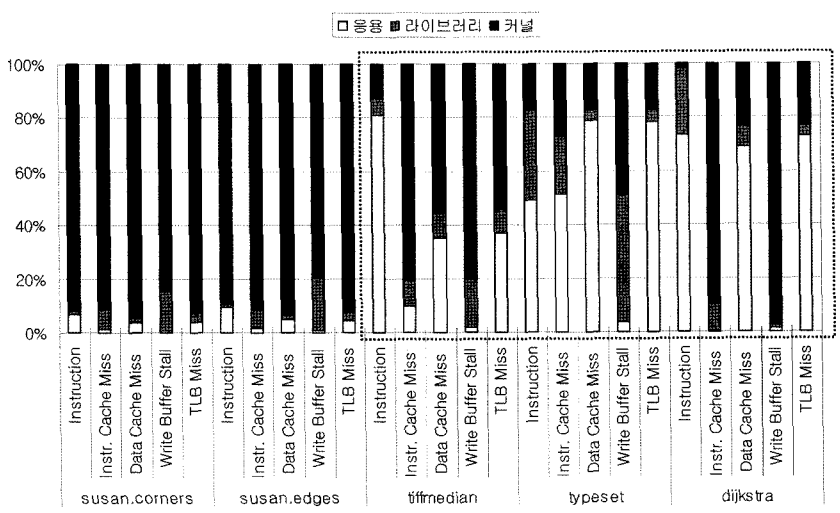


그림 9 소프트웨어 계층 별 메모리 성능

리의 함수보다 수행시간이 약 5.8배 길었다.

커널 계층의 메모리 관리가 슬라브 할당기(Slab allocator), 페이지 캐쉬, 페이지 할당기 등의 다중 구조[20]로 구성 되었기 때문에 수행 코드 수가 많고 일반적으로 운영체제 커널의 캐쉬 효율이 나쁘기 때문에 메모리 입출력 오버헤드에 의해 수행시간이 늘어날 수 있다[21]. 이를 본 논문의 목표 시스템에서 검증하기 위해 MiBench의 일부 벤치마크 응용에 대한 메모리 성능을 측정하는 추가적인 실험을 수행하였다. 실험 결과를 보면 그림 9의 점선 내의 응용과 같이 동적 메모리 할당 및 반납이 빈번한 여러 응용에서 소프트웨어 계층 별 캐쉬, MMU, 쓰기 버퍼(write-buffer) 등의 메모리 성능에서 커널의 수행코드 비율에 비해 메모리 입출력 비율이 큰 것을 알 수 있다.

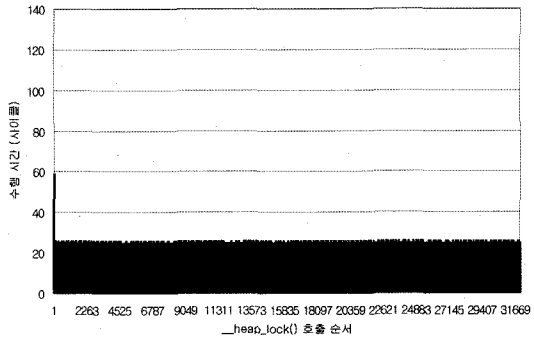
MiBench 벤치마크 응용에서는 커널 수준의 페이지 메모리 관련 함수가 호출 되는 경우는 응용 수준의 동적 메모리 관련 함수가 호출되는 경우의 2-4%에 불과하다. 그러나, 커널의 코드들은 일반적으로 타임머 인터럽트나 입출력과 같은 외부 이벤트나 라이브러리의 시스템 호출과 같은 간헐적인 요청에 의해서 수행되기 때문에 운영체제의 메모리 성능이 낮은문제점과 응용과 커널 간의 전환 오버헤드가 큰 문제점을 가지고 있기 때문에 내장형 응용을 작성할 때 페이지 단위로 할당 및 반납을 하는 커널의 동작 특성을 고려하여 작성하여야 한다. 또한, 커널의 서비스에 크게 의존적인 경우 응용을 프로세스로 수행하는 방법이 아닌 커널의 스레드로 수행하여 응용과 커널간 전환 오버헤드를 최소화하는 방안도 존재한다[22].

4.2.3 다중 스레드를 위한 동기화 성능분석

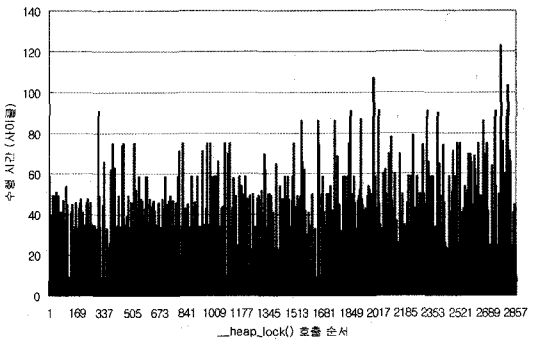
4.2.1절의 실험 결과인 그림 7(a)에서 동적 메모리 관련 함수의 수행시간이 상호배제 관련 함수에 의해 스윙하는 현상을 분석하기 위해 응용에 따라 상호배제 락 획득 시의 수행시간을 측정하였다. 측정 결과에 의하면 응용에 따라 일정한 경우, 수 십 사이클 사이에서 스윙하는 경우, 10-20ms로 크게 증가하는 경우 등 세가지 서로 다른 경우가 발견되었다.

그림 10은 서로 다른 내장형 응용 벤치마크의 상호배제 락 획득 함수인 *__heap_lock()*의 수행시간을 순서대로 나타낸 것이다. 그림 10(a)의 경우 수행시간이 26사이클 이하로 일정하지만 그림 10(b)의 경우 10사이클에서 120사이클 사이에서 스윙하는 것을 볼 수 있다. 또한, MiBench의 모든 벤치마크 응용에서 측정한 대부분의 상호배제 함수의 수행시간은 140사이클 미만이지만 최대 지연 시간은 약 20ms로 이러한 경우는 간헐적으로 발견된다.

uClibc의 경우 동적 메모리를 위해 스레드 상호배제



(a) patricia



(b) typeset

그림 10 *__heap_lock()* 함수의 수행시간 비교

```
int pthread_mutex_lock (pthread_mutex_t *mutex)
{
    while (mutex->__m_lock.__spinlock == 0) {
        usleep(10000);
    }
    --(mutex->__m_lock.__spinlock);
    return 0;
}
```

그림 11 *pthread_mutex_lock()*의 구현

(pthread mutex)를 사용하는데 그림 11는 해당 함수의 소스 코드를 나타낸다. 소스 코드를 보면 상호배제 진입 시 실패할 경우 10ms를 대기 한 후 다시 시도하게 되는데, 간헐적으로 발견되는 지연 현상은 상호배제 락 획득에 실패할 경우에 발생한다.

또한, 그림 11의 구현에서 볼 수 있듯이 상호배제 락 획득에는 mutex, __m_lock, __spinlock의 세 변수에 대한 참조를 하는데 상호배제 관련 함수가 호출되는 간격이 충분히 길어서 관련 변수 및 함수 코드가 캐쉬에서

제거되게 되면 그림 10(b)와 같이 상호배제 락 획득 수행시간이 140사이클 미만에서 스윙하게 된다.

이러한 현상은 최근 내장형 시스템에서 수행되는 멀티미디어와 같은 연성 실시간(soft real-time) 응용에서 비디오 재생 시 재생 속도가 일정하지 않게 되거나 끊김 현상이 발생하는 등 동적 메모리 수행 시간의 지터(jitter)가 커짐에 따라 발생할 수 있는 문제들이 존재한다. 이러한 문제는 2장의 관련 연구에서 살펴보았던 내장형 응용에 특화된 락 없는 메모리 할당기(lock-free memory allocator)를 사용하는 방법으로 개선할 수 있다[9].

4.3 하드웨어 설계 인자에 의한 영향

빈번한 동적 메모리 할당 및 반납이 발생하는 내장형 응용에서 각 함수별 호출 빈도가 정해졌을 때 성능을 최적화 할 수 있는 하드웨어 설계 인자를 결정할 필요가 있다. 이를 위해서 각 함수별 성능 지표와 각각의 하드웨어 설계 인자를 변화시켰을 때 수행시간이 얼마나 향상 혹은 저하되는지 등의 하드웨어 설계 인자를 변화시켰을 때의 성능에 대한 데이타를 확보해야 한다.

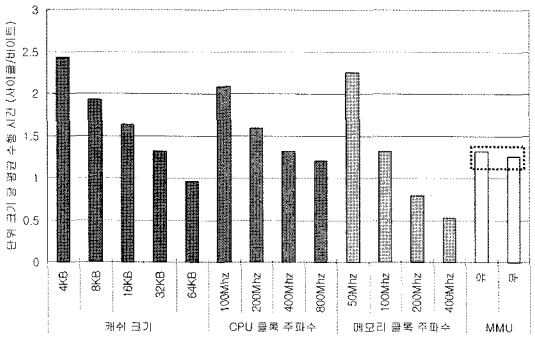


그림 12 CPU 구성에 따른 *malloc()* 함수의 수행시간 비교

그림 1의 하드웨어 시스템 모델에서 CPU 구성에 따른 동적 메모리 관련 함수의 성능분석을 위해 기본 CPU 구성을 CPU 클록 주파수 400Mhz, 메모리 클록 주파수 100Mhz, 캐쉬 크기 32KB로 설정하고 각각의 인자를 변화시켜가며 실험을 수행하였다. 그림 12은 동적 메모리 할당 함수인 *malloc()*의 CPU 구성에 따른 수행시간을 비교한 것으로 모든 인자의 변화에 대해서 성능 차이가 현저한 것을 알 수 있다. 단, 점선 내의 결과와 같이 MMU의 경우 MMU를 탑재하지 않은 경우 약 5%의 성능 향상이 있다. 또한, 상호배제 함수인 *__heap_lock()*, *__heap_unlock()*를 제외한 다른 함수들은 *malloc()*과 동일한 양상을 보인다.

그림 13은 CPU 구성에 따른 상호배제 당 지연시간을 비교한다. MiBench 벤치마크 응용의 경우 상호배제에 의한 지연 현상을 간헐적으로만 관찰할 수 있었기 때문에 IBM DeveloperWorks[23]의 벤치마크 중의 하나인 *sync6*를 사용하여 상호배제 당 지연 시간을 측정하였다. 측정 결과 그림 12의 *malloc()*과는 달리 CPU 클록 주파수를 제외한 캐쉬 크기, 메모리 클록 주파수 변화에 대해선 성능 차이가 거의 없고 점선 내의 결과와 같이 MMU의 경우 MMU를 탑재하지 않은 경우 약 70%의 성능 향상이 있다. 이러한 결과는 동적 메모리 할당 함수의 성능 향상과는 큰 차이가 있다.

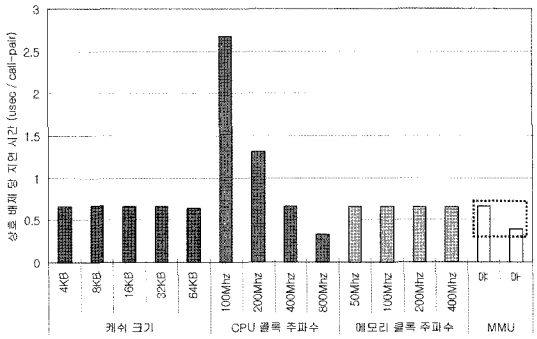


그림 13 CPU 구성에 따른 상호배제 당 지연시간 비교

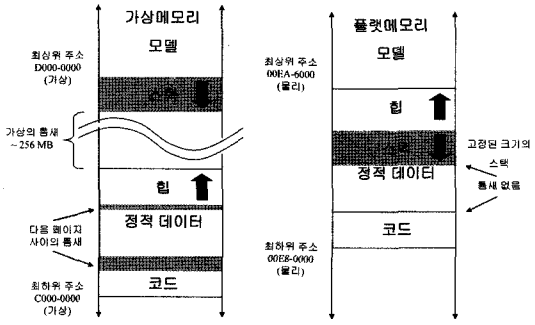
위의 실험 결과에서와 같이 CPU 구성에 따른 동적 메모리 할당 성능의 비교에서는 인자의 변화 따라서 2-4배의 성능 차이가 있는 것을 확인하였다. 이러한 실험 결과는 시스템 설계 시에 다음과 같이 활용한다. 동적 메모리 할당 성능은 캐쉬 크기와 CPU 클록 주파수를 변경함으로써 향상을 될 수 있지만 소비 전력은 주파수에 비례하므로 저전력 시스템의 설계에는 큰 크기의 캐쉬 및 낮은 주파수를 우선 고려 대상으로 한다. 반면, 동기화 성능은 캐쉬 크기에 영향을 거의 받지 않고 CPU 클록 주파수에만 크게 좌우되기 때문에 동기화 성능의 향상에 CPU 클록 주파수도 함께 고려해야 한다.

그림 12, 그림 13의 실험 결과에서 특이할 점은 점선 내의 결과에 나타난 MMU 탑재 유무에 따라 동기화 성능은 70%의 비교적 큰 성능 차이가 있지만 동적 메모리 할당 성능은 5% 미만의 미미한 차이를 보인 점이다. 이러한 현상은 MMU를 탑재하지 않은 운영체제는 응용 프로세스 별 가상메모리 관리를 위한 페이지 테이블과 페이지 보호에 필요한 오버헤드가 없기 때문이다. 그러나, 동기화 성능은 문맥교환이나 응용 프로세스 관리에 크게 의존하기 때문에 성능 개선 효과가 크지만 동적 메모리 할당 성능 개선 효과는 미미하다.

MMU를 탑재하지 않은 시스템은 가상 메모리 주소를 지원하지 않기 때문에 그림 14(b)와 같은 플랫 메모리

모델을 사용해야 한다. 이로 인해서 MMU를 탑재하지 않은 운영체제는 *fork()* 등의 일부 시스템 호출과 서비스를 사용할 수 없다. 또한, 메모리 보호 기능이 없기 때문에 내장형 응용 프로그램 작성자는 응용 프로그램 간 메모리 영역이 침범하지 않도록 주의해야 한다. 특히 스택의 크기가 고정되어 있기 때문에 내장형 응용의 코드를 생성할 때 응용이 수행 되는 기간 동안 스택의 최대 크기를 예측해서 정해야 한다. 그렇지 않으면 스택 데이타가 정적 데이타 혹은 코드 영역을 침범하여 시스템이 오동작하게 된다.

이와 같이 MMU를 탑재하지 않은 운영체제의 경우 일부 성능 개선효과가 있는 장점과 구조적으로 메모리 보호를 지원하지 않고 여러 기능과 제약이 존재하는 단점이 존재하기 때문에 내장형 시스템 설계 시 목표 내장형 응용에 따라 MMU 탑재 유무를 결정한다.



(a) MMU 유 (b) MMU 무

그림 14 MMU 탑재 유무에 따른 메모리 모델

5. 결 론

본 논문에서는 운영체제를 탑재한 내장형 시스템에서 내장형 응용을 위한 대표적인 벤치마크인 MiBench의 응용들에 대해서 각 소프트웨어 계층과 하드웨어 설계 인자에 따른 동적 메모리 기법의 성능을 분석하였다.

기존의 연구들에서 가정하고 있는 운영체제를 탑재하지 않은 단일 스레드의 시스템과는 달리 본 논문에서는 운영체제를 탑재한 시스템을 사용하였다. 이러한 시스템에서는 라이브러리와 커널의 두 계층에 걸쳐서 동적 메모리 관리가 이루어지므로 보다 다양한 실험 결과와 성능분석이 가능하다. 본 논문은 기존의 연구와 유사한 성능 평가 척도를 제시하고 있다. 그러나, 응용 수준의 알고리즘의 성능에 초점을 맞추고 있는 기존의 연구들과는 다르게 본 논문은 시스템 수준의 성능에 대한 운영체제에 의한 영향과 하드웨어 인자 변화에 의한 영향에 초점을 맞추고 있다.

실험 결과 각 응용의 동적 메모리 할당과 반납 패턴에 따라 라이브러리 계층에서 관리되는 메모리 풀의 크기가 대부분 수십 개 미만으로 관찰되었다. 이러한 결과와 유사하게 동적 메모리 할당의 경우 그 수행시간이 할당 크기에 비례하였고 반납의 경우 수행시간이 대부분 일정하였다. 단, 동기화 시에 상호배제 진입에 실패할 경우 커널의 태스크 스케줄러에 따른 지연시간은 간헐적으로 크게 증가하는 경우가 발생하였다. 또한, 하드웨어 설계 인자를 변화시켰을 때 상호배제 관련 함수는 CPU 클록 주파수에 대해서만 성능의 차이가 크게 나타났지만 다른 동적 메모리 관련 함수들의 경우는 모든 하드웨어 설계 인자에 대해 성능의 차이가 두드러졌다.

본 연구는 실제와 같은 실험 환경에서 동적 메모리 기법을 적용할 때 각각의 소프트웨어 및 하드웨어 컴포넌트가 전체 시스템 성능에 미치는 영향을 정량적으로 분석함으로써 운영체제 탑재에 따른 동적 메모리 할당 및 반납의 수행시간을 예측할 수 있도록 하였다. 또한, 빈번한 동적 메모리 할당 및 반납이 발생하는 내장형 응용에서 각 함수 별 성능 지표와 하드웨어 설계 인자를 변화시켰을 때의 성능을 비교할 수 있는 데이타를 제공하였다.

내장형 시스템은 주어진 제약 조건을 만족하는 경우 하드웨어설계 인자뿐이 아니라 소프트웨어 컴포넌트도 특성화 될 수 있는 특징을 갖고 있기 때문에, 성능분석 결과의 제시된 적절한 활용 방안을 바탕으로 내장형 시스템 설계에 도움이 될 것이다. 따라서, 성능분석 결과와 데이타를 이용하여 내장형 시스템 설계자는 목표 시스템의 제약이 주어졌을 때 이를 만족할 수 있는 최적의 하드웨어 설계 인자 혹은 소프트웨어 컴포넌트에서의 개선점을 구할 수 있고 이를 이용하여 보다 고성능, 저전력의 내장형 시스템의 구현할 수 있을 것이다. 특히 최근 유비쿼터스 디바이스 등 이동성과 휴대성이 강조되는 내장형 시스템의 체계적인 설계 기술에 효과적으로 적용될 수 있을 것이다.

참 고 문 헌

[1] Wolfgang A. Halang, and Alexander D. Stoyenko, *Real-Time Computing*, Springer-Verlag, 1992.

[2] Isabelle Puaut, "Real-Time Performance of Dynamic Memory Allocation Algorithms," *Proc. of Euromicro Conference on Real-Time Systems (ECRTS'02)*, Vienna, Austria, Jun. 2002.

[3] Steven M. Donahue, and Matthew P. Hampton, Morgan Deters, Jonathan M. Nye, Ron K. Cytron, and Krishna M. Kavi, "Storage Allocation for Real-Time, Embedded Systems," *Proc. of International Workshop on Embedded Software (EMSOFT'01)*, Tahoe City, CA, USA, Oct. 2001.

[4] M. Masmano, I. Ripoll, and A. Crespo, "Dynamic

storage allocation for real-time embedded systems," *WIP Session International Real-Time Systems Symposium (RTSS'03)*, Cancun, Mexico, Dec. 2003.

[5] M. Masmano, I. Ripoll, A. Crespo, and J. Real, ""TLSF: a new dynamic memory allocator for real-time systems," *Proc. of Euromicro Conference on Real-Time Systems (ECRTS'04)*, Catania, Italy, Jun. 2004.

[6] W. Wolf, *Computers as Components: Principles of Embedded Computing System Design.* Morgan Kaufmann, 2001.

[7] J. A. Darringer, R. Bergamaschi, S. Bhattacharya, D. Brand, A. Herkersdorf, J. Morell, I. I. Nair, P. Sagmeister, and Y. Shin, "Early analysis tools for system-on-a-chip design," *IBM Journal of Research and Development*, vol. 6, no. 6, pp. 20 - 38, 2002.

[8] Hyok-Sung Choi, and Hee-Chul Yun, "Context Switching and IPC Performance Comparison between uClinux and Linux on the ARM9 based Processor," *Proc. Of SAMSUNG Tech. Conference*, 2005.

[9] Dave Dice, and Alex Garthwaite, "Mostly Lock-Free Malloc," *Proc. of International Symposium on Memory Management (ISMM'02)*, Berlin, Germany, Jun. 2002.

[10] Dirk Grunwald, Benjamin Zorn, and Robert Henderson, "Improving the Cache Locality of Memory Allocation," *Proc. of ACM SIGPLAN Conference on Programming language design and implementation*, Albuquerque, New Mexico, Jun. 1993.

[11] Steve Furber, *ARM System-on-Chip Architecture.* Addison-Wesley, 2000.

[12] S. Swan, "An introduction to system-level modeling in SystemC 2.0," *Open SystemC Initiative*, Tech. Rep., 2001.

[13] ARM, "Benchmarking with armulator," Application Note, Mar. 2002.

[14] R. Klein, K. Travilla, and M. Lyons, "Performance estimation of MPEG-4 algorithms on arm based designs using co-verification," *Proc. Embedded Systems Conference*, San Francisco, USA, 2002.

[15] Tim Hopes, "Hardware/Software Co-verification, an IP Vendors Viewpoint," *Proc. IEEE International Conference on Computer Design (ICCD'98)*, Austin, TX, Oct. 1998.

[16] Francesco Menichelli, Mauro Olivieri, Luca Benini, Monica Donno, and L. Bisdounis, "A Simulation-Based Power-Aware Architecture Exploration of a Multiprocessor System-on-Chip Design," *Proc. Design, Automation and Test in Europe Conference and Exposition (DATE'04)*, Paris, France, Feb. 2004.

[17] http://www.eembc.org/

[18] Mathew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," *Proc. IEEE Annual Workshop on Workload Characterization*, Austin, TX, Dec. 2001.

[19] Emery D. Berget, B. G. Zorn, and K. S. McKinley, "Reconsidering Custom Memory Allocation," *Proc. ACM Conference on Object-Oriented Programming Systems (OOPSLA'02)*, Seattle, WA. Nov. 2002.

[20] Daniel P. Bovet, and Marco Cesati, *Understanding the Linux Kernel 2nd Ediotion.* O'Reilly, 2003.

[21] J. B. Chen, and B. N. Bershad, "The impact of operating system structure on memory system performance," *Proc. ACM Symposium on Operating System Principles (SOSP'93)*, Asheville, NC, Dec. 1993.

[22] C. Lever, M. Eriksen, and S. Molloy, "An Analysis of the TUX Web Server," *CITI U. of Michigan Technical Report*, 2000.

[23] E. G. Bradford, *Runtime: High Performance Programming Techniques on Linux and Windows 2000.* IBM Developer Works, 2001.

박 상 수

1998년 KAIST 전산학과 학사. 2000년 서울대학교 컴퓨터공학과 석사. 현재 서울대학교 전기.컴퓨터공학부 박사과정 관심 연구분야는 실시간 내장형 시스템, 시스템 소프트웨어, 성능 평가 및 분석

신 현 식

1973년 서울대학교 응용물리학과 공학사 1980년 미국 텍사스 대학교 의공학과 공학석사. 1985년 미국 텍사스 대학교 전기.컴퓨터공학과 공학박사. 1986년~현재 서울대학교 전기.컴퓨터공학부 교수. 관심 연구분야는 실시간 계산, 분산 시스템, 모바일 컴퓨팅, 입출력 처리