Real-Time Linux 시스템을 위한 재구성 가능한 메모리 할당 모델

심 재 홍[†]·정 석 용[††]·강 봉 직[††]·최 경 희[†††]·정 기 현[††††]

요 약

본 논문에서는 Real-Time Linux를 위한 메모리 할당 모델을 제안한다. 제안 모델은 사용자로 하여금 하나의 응용에 여러 개의 연속된 메모리 영역들을 생성하고, 각 영역마다 별도의 영역 할당 정책을 설정한 후, 원하는 영역으로부터 필요한 메모리 블록을 할당 받을 수 있게 한다. 이를 위해 기존의 단일 메모리 관리 모듈 대신 할당 정책을 구현한 영역 할당자 모듈과 이를 제어하는 영역 관리자 모듈로 세분한 두 단계 분리 구조를 채택했다. 이 구조는 할당 정책을 할당 메커니즘으로부터 분리함으로써, 시스템 개발자로 하여금 필요한 경우 동일한 할당 정책을 서로 다른 알고리즘을 사용하여 구현할 수 있게 한다. 또한 사전에 정의된 인터페이스를 준수할 경우 새로운 할당 정책을 쉽게 구현해 삽입할 수 있고, 불필요한 정책은 시스템에서 제거할 수도 있다. 제안 모델은 다수개의 할당 정책들을 사전에 구현하여 제공함으로써, 시스템 구축자로 하여금 매번 기존 정책들을 새로이 구현할 필요 없이 제공된 정책들 중 해당 응용에 가장 적합한 정책들만을 선택하여 시스템을 재구성할 수 있게 한다.

A Reconfigurable Memory Allocation Model for Real-Time Linux System

Jae-Hong Shim[†] · Suk-Yong Jung[††] · Bong-Jik Kang[††] · Kyung-Hee Choi[†††] · Gi-Hyun Jung[††††]

ABSTRACT

This paper proposes a memory allocation model for Real-Time Linux. The proposed model allows *users* to create several continuous memory *regions* in an application, to specify an appropriate *region allocation policy* for each memory region, and to request memory blocks from a necessary memory region. Instead of using single memory management module in order to support the proposed model, we adopt two-layered structure that is consisted of *region allocators* implementing allocation policies and a *region manager* controlling regions and region allocator modules. This structure separates allocation policy from allocation mechanism, thus allows *system developers* to implement same allocation policy using different algorithms in case of need. In addition, it enables them to implement new allocation policy easily as long as they preserver predefined internal interfaces, to add the implemented policy into the system, and to remove unnecessary allocation policies from the system. Because the proposed model provides various allocation policies implemented previously, *system builders* can also reconfigure the system by just selecting most appropriate policies for a specific application without implementing these policies from scratch.

키워드 : 메모리 할당 모델(memory allocation model), 실시간 메모리 할당 알고리즘(real-time memory allocation algorithm), 실시간 시스템(real-time system), 실시간 리눅스 시스템(real-time linux system)

1. 서 론

실시간 시스템을 연구하는 대부분의 연구자들의 애로점은 해당 응용을 쉽게 구현하고 테스트 할 수 있는 범용 실시간 시스템이 없다는 것이다. 이는 실시간 시스템 특성상 대부분의 시스템이 응용의 특성에 맞게 특별히 설계/제작된 보드에 내장되기 때문이다. 비록 일부 상용 RTOS들이 범용 시스템 보드를 동시에 지원하기도 하지만, 소스가 공개되지 않아 일반 연구자들이 쉽게 접근하기 어려운 문제가 있다. 다행이 최근 들어 Linux를 기반으로 하는 일부 실시간 시스템들이 발표되었다[1-3].

이들 중 Real-Time(RT) Linux [1]는 기존 Linux 커널의 외부에서 그러나 기존 커널의 하부에서 별도의 모듈로 존재하는 조그마한 실시간 커널이다. 그 결과 실시간 응용은 어떠한 Linux 커널 서비스도 제공 받지 못한다. 대신 RT-

† 정 회 원 : 아주대학교 정보통신전문대학원 Post Doc.
†† 정 회 원 : 아주대학교 대학원 정보및컴퓨터공학부
††† 정 회 원 : 아주대학교 정보통신전문대학원 교수
†††† 정 회 원 : 아주대학교 전기전자공학부 교수
논문접수 : 2001년 5월 10일, 심사완료 : 2001년 7월 14일

Linux 상의 실시간 응용과 Linux상의 비실시간 응용간 통신할 수 있는 메커니즘인 lock-free FIFO 큐를 제공하고 있다. 이러한 기법과 인터럽트 가로채기(interception) 기법을 통해 주기적 경성 실시간 태스크들은 기존 Linux의 방해(interference) 없이 엄격하게 스케줄링 될 수 있다. 그러나 Real-Time Linux는 기존 Linux 기반에서 실시간 응용의 시간 제약을 만족할 수 있는 기반만을 제공할 뿐, 극히 제한된 실시간 서비스를 제공한다. 이에 대한 예로 실시간 메모리 할당 및 반환과 관련한 서비스를 제공하지 않는다.

동적 메모리 할당(dynamic storage allocation : DSA)은 사전에 그 크기를 결정할 수 없는 객체를 효과적으로 관리하기 위해 사용하는 유용한 프로그래밍 기술이다. DSA는 또한 태스크나 프로세스보다 생명주기가 훨씬 짧은 객체들을 동적으로 관리함으로써 제한된 메모리 자원의 사용 효율성을 증가시킨다. 실시간 시스템의 적용 대상이 다양해지고 실시간 응용의 특성에 따라 다양한 크기의 메모리 블록이 동적으로 필요함에 따라 실시간 동적 메모리 할당에 대한 요구도 증가되고 있다[4-8]. 그러나 기존에 발표된 실시간 DSA를 바탕으로 새로운 알고리즘을 개발하고, 이의 성능을 비교 분석하기 위해선 기존 알고리즘도 함께 개발해야 하는 고충이 따른다. 또한 연구자들은 자신이 연구하는 분야의 소스 코드도 이해해야 하지만, 이것이 미치는 파급 효과로 인해 시스템 전체를 이해해야 하고 하위 단계의 복잡한 커널 모듈도 함께 수정해야 하는 어려움이 있다[9].

이러한 맥락에서 본 논문에서는 소스가 공개되어 여러 실시간 연구자들이 쉽게 접근할 수 있는 RT-Linux를 위한 실시간 메모리 할당 모델을 제안하고자 한다. 또한 폭 넓은 실시간 응용의 다양한 요구에 적합한 여러 실시간 메모리 할당 정책들을 구현하여 제공하고, 이를 기존 시스템에 쉽게 적용할 수 있는 재구성 방안도 함께 제시한다. 제안 모델을 바탕으로 시스템 설계자는 RT-Linux상에서 제안 모델이 제공하는 다양한 메모리 할당 정책들 중 해당 응용에 가장 적합한 정책들을 선택하여 시스템을 재구성할 수 있고, 또한 새로운 할당 정책을 개발하고 테스트할 수 있는 기반 환경을 확보할 수 있다.

본 논문의 구성은 다음과 같다. 먼저 2절에서 본 연구와 관련이 있는 기존 연구들을 비교 분석한다. 3절에서는 RT-Linux를 위한 재구성 가능한 메모리 할당 모델을 제안하고, 제안 모델이 제공하는 다양한 실시간 메모리 할당 정책들과 이의 구현 방안 및 시스템 재구성 방안에 대해 기술한다. 4절에서는 영역 관리를 위해 필요로 하는 각 할당 정책별 오버헤드에 대해 분석하고, 5절에서는 할당 정책들의 메모리 관리 성능을 실험해 보고, 그 결과를 분석한다. 그리고 마지막 6절에서 향후 연구 계획에 대해 논의한다.

2. 관련 연구

전통적으로 실시간 시스템 개발자들은 메모리 관리를 위한 방안으로 최악의 경우 실행시간을 사전에 예측할 수 있고 메모리 조각화(fragmented)가 발생하지 않는 정적 할당(static allocation) 방법을 주로 사용하였다[4-6]. 이 방법은 시스템 개발 당시 고정된 크기의 메모리 블록을 프로그램상의 전역 변수로 설정한 후, 이를 각 태스크에게 사전에 할당 시키거나 또는 시스템 초기화시 각 태스크가 사용할 메모리 블록을 미리 할당 받는 방식이다. 사전에 각 태스크에 할당된 블록들은 오직 해당 태스크에서만 사용될 수 있고, 다른 태스크에서는 사용할 수 없다. 이 방법은 불필요한 메모리 낭비를 초래할 수 있으며, 프로그래머가 직접 메모리를 관리해야 하는 부담을 가지게 했다[5].

이러한 문제점을 극복하기 위해 일부 상용 RTOS들은 제한적이지만 동적 할당(dynamic allocation) 방법을 지원한다. pSOS [10]는 분할(partition)과 영역(region)이라는 두 가지 추상화된 메모리 객체를 지원한다. 분할 객체는 사전에 지정된 동일한 크기의 메모리 블록만을 할당하는 고정 크기 할당 방식을 지원하고, 영역 객체는 임의의 크기의 메모리 블록을 할당하는 가변 크기 할당 방식을 지원하는 객체이다. 분할 객체는 관리하는 메모리 구역에서 항상 동일한 크기의 블록들만을 할당/반환하므로, 블록의 할당/반환에 소요되는 시간이 항상 일정(constant)하고, 메모리 조각화가 발생하지 않는 장점이 있다. 영역 객체의 경우 최악-적합(worst-fit) 할당 정책을 사용하고, 즉시-합병(immediate coalescing)을 실시한다. 그러나 반환된 블록은 크기 순으로 하나의 리스트에서 관리하므로, 할당된 블록의 반납시 소요되는 최악의 경우 실행시간을 사전에 예측할 수 없는 문제가 있다. 또한 두 객체가 서로 다른 인터페이스를 사용함으로써 상호간 호환성이 없는 문제가 있다.

VRTX [11] 역시 pSOS처럼 고정 크기 할당 방식을 지원하는 분할(partition) 객체와 가변 크기 할당 방식을 지원하는 힙(heap) 객체를 지원한다. 그러나 힙 객체의 경우 참고문헌 [24]와 동일한 범용 시스템용 동적 메모리 할당 알고리즘을 사용한다. 이 알고리즘은 한 페이지(512 bytes) 크기 이하의 프리 블록에 대해서는 2의 지수승 크기별로 그 페이지 내에서 별도로 관리하고, 한 페이지 보다 큰 프리 블록들은 주소 크기별로 하나의 리스트에 보관한다. 이는 한 페이지 이하 크기의 블록 할당/반환에는 일정한 시간을 보장하지만, 이 보다 큰 블록들에 대해서는 실행시간을 예측할 수 없는 문제점이 있다.

경성 실시간 시스템을 위한 가변 크기 블록들의 동적 메모리 할당 알고리즘은 메모리 조각화를 최소화하면서 최악의 경우 실행시간을 예측할 수 있어야 한다. 절반-적

합(half-fit)은 실시간 시스템용으로 개발된 DSA 알고리즘이다[4]. 프리 리스트(half 리스트)의 영역 크기로 2의 거듭제곱을 사용하고, 해당 리스트 내에서 적절한 크기의 블록을 찾기 위한 순차적 리스트 탐색을 지양하기 위해, 영역 (2^{i-1}, 2^i) 사이 크기의 블록 요청의 경우 리스트 (i-1)이 아니라 i 에서 할당하는 round-up 전략을 사용한다. 이 알고리즘은 시간 복잡도가 $O(1)$이고 고정된 실행 시간을 보장하며, 항상 고정된 개수의 프리 리스트들을 가지는 장점이 있다. 그러나 리스트 영역 크기가 2의 거듭제곱 크기이므로 작은 크기의 메모리 요구가 많은 경우 상대적으로 높은 조각화 율을 보임을 참고문헌 [7]에서 확인할 수 있다.

QSHF(quick-segragated-half-fit) [8] 알고리즘은 기본적으로 작은 크기의 메모리 요구에는 워드 크기별로 프리 리스트를 관리하는 빠른-적합(quick-fit) [12] 전략을, 중간 크기의 요구에는 적절한 크기 영역별로 리스트를 관리하는 segregated-fit [13] 전략을, 큰 크기의 요구에는 2의 거듭제곱 크기별로 리스트를 관리하는 절반-적합 전략을 사용한다. 이 알고리즘의 시간 복잡도는 $O(1)$이며, 최악의 경우 실행시간을 쉽게 측정할 수 있고 하나의 알고리즘으로 다양한 메모리 할당 정책을 지원할 수 있는 장점이 있다. 그러나 불필요한 메모리 관리 오버헤드가 정책에 상관없이 일괄적으로 부과되는 문제가 있다.

따라서 본 논문에서는 RT-Linux상에서 나양한 메모리 할당 정책을 지원하면서도 필요에 따라 사용자가 이를 선택적으로 골라 사용할 수 있는 메모리 할당 모델을 제시한다. 이 모델은 다수의 메모리 할당 정책을 지원하면서도 동일한 사용자 인터페이스를 유지함은 물론, 사용자는 하나의 응용에 여러 메모리 영역들을 생성할 수 있고, 각 영역마다 서로 다른 할당 정책을 설정하여 필요한 메모리 블록들을 할당 받을 수 있다.

3. 메모리 할당 모델 및 실시간 영역 할당 정책

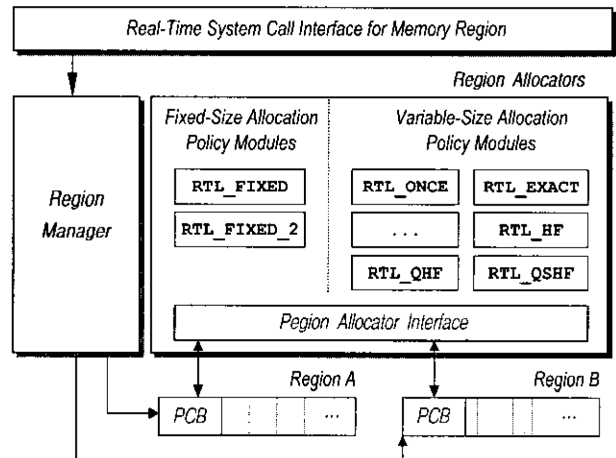
본 절에서는 RT-Linux상에서 나양한 실시간 메모리 할당 정책을 지원하면서 재구성이 가능한 메모리 할당 모델을 제안한다. 먼저 제안 모델의 구성 요소에 대해 간단히 기술하고, 제안 모델이 지원하는 나양한 메모리 할당 정책들에 대해 논의한다. 또한 다양한 정책들을 선택적으로 삽입/삭제할 수 있는 재구성 방안에 대해서도 논의한다.

본 논문에서 제안하는 메모리 할당 모델의 설계 목표는 다음과 같다.

- 하나의 응용에서 여러 개의 메모리 영역을 만들고, 각 영역으로부터 필요한 메모리 블록을 할당 받고 반환할 수 있게 한다.
- 각 영역마다 서로 다른 실시간 영역 할당 정책을 설정할 수 있게 한다. 이를 위해 제안 모델을 기반으로 구현한 다양한 실시간 영역 할당 정책 및 구현 모듈을 제공한다.
- 시스템 구축자로 하여금 사전에 제공된 다양한 영역 할당 정책(구현 모듈)들 중 필요한 할당 정책만을 골라 시스템 구축시 선택적으로 삽입/삭제 할 수 있게 한다.
- 영역과 영역 할당 정책에 상관없이 항상 동일한 사용자 인터페이스를 제공한다.
- 널리 사용되고 있는 RT-Linux에 기반 메모리 할당 모델과 사전에 정의된 내부 표준 인터페이스를 제공함으로써, 시스템 연구자들로 하여금 필요한 경우 새로운 실시간 메모리 할당 알고리즘을 보다 쉽게 구현할 수 있도록 하는 기반 환경을 제공한다.

3.1 메모리 할당 모델

본 논문에서 제안하는 메모리 할당 모델은 영역 관리자(region manager)와 영역 할당자(region allocator)로 구성된다 (그림 1 참조). 제안 모델의 기본적인 메모리 관리 단위는 영역(region)이며, 이는 영역 관리를 위해 필요한 정보를 저장하는 정책 제어 블록(policy control block : PCB)과 데이터 블록들로 구성된다.



(그림 1) 영역 관리자와 영역 할당자로 구성된 메모리 할당 모델

영역은 시작 주소와 크기를 가지는 연속된 메모리 범위를 가진 추상화된 객체이다. 사용자는 영역 관리자가 제공하는 API 함수를 통해 영역의 시작 주소, 크기, 영역 할당 정책 등을 지정함으로써 동적으로 새로운 영역을 만들 수 있다. 이때 시작 주소와 크기를 가진 연속된 메모리는 다른 영역으로부터 동적으로 할당 받았거나 또는 시스템 초기에 프로그램상에서 전역 변수로 설정된 배열이어야 한다. 영역은 해당 구역 내에 속하는 동일 또는 가변 크기의 데이터 블록들을 가지며, 이들 블록들은 이미 사용자에게 할당되어

사용 중인 블록들과 사용 후 반환된 프리 블록들로 구분된다. 블록들은 해당 영역에 설정된 영역 할당 정책에 따라 사용자에게 할당되거나 반환된다.

영역 할당 정책은 해당 영역 내의 프리 블록들을 어떻게 어떤 순서로 관리하며, 사용중인 블록이 반환될 때 어떻게 기존의 인접 프리 블록과 합병할 것이가를 결정하는 규칙들의 집합이다. 이 정책은 영역의 조각화와 할당/반환에 소요되는 실행시간에 상당한 영향을 미친다. 따라서 사용자는 영역별 그 영역에 가장 적합한 정책을 신중하 고려하여 선택해야 한다. 하나의 영역은 하나의 정책만을 지원할 수 있다. 제안 모델은 고정 크기 할당 방식과 가변 크기 할당 방식을 지원하는 여러 할당 정책들을 지원하며, 이는 3.2절에서 상세히 논의된다. 특정 할당 정책을 구현한 모듈을 해당 정책을 지원하는 영역 할당자라 한다. 제안 모델은 또한 재구성 기능도 지원한다. 이를 이용하여 사용자는 기본적으로 제공되는 영역 할당자들(영역 할당 정책들) 중에서 해당 응용에 불필요한 영역 할당자들을 시스템 구성시 삭제할 수 있으며, 필요한 경우 응용에 적합한 새로운 영역 할당 정책을 개발하여 이를(영역 할당자) 삽입할 수도 있다. 이에 대한 구체적인 방안은 3.5절에서 다시 논의된다.

정책 제어 블록(PCB)은 해당 영역 내의 프리 블록들을 관리하기 위해 필요한 정보들을 보관하는 하나의 데이터 구조이다. 정책 제어 블록은 영역의 시작 주소와 크기, 할당 정책과 관련한 관리 데이터, 해당 정책을 구현한 멤버 함수들에 대한 벡터 포인터, 프리 블록들의 리스트 주소 등을 기본적으로 가진다. 정책별로 여러 프리 리스트를 가질 수 있으며, 이 경우 각 프리 리스트에 대한 주소들을 테이블로 저장하여 관리한다. 따라서 정책 제어 블록은 정책별 서로 다른 크기를 가지며, 이는 곧 정책별 영역 관리를 위해 필요한 별도의 메모리 오버헤드가 된다. 각 정책별 정확한 정책 제어 블록의 오버헤드는 4절에서 상세히 논의된다. 정책 제어 블록은 해당 영역 내의 맨 처음에 존재할 수도 있고, 경우에 따라서는 사용자가 지정한 또 다른 영역으로부터 할당 받아 이를 활용할 수도 있다.

영역 관리자는 할당 정책을 구현한 영역 할당자의 등록 및 제거, 새로운 영역의 생성 및 삭제, 정책별 해당 멤버 함수의 호출 등을 담당한다. 시스템 개발자는 필요한 경우 응용에 적합한 새로운 할당 정책을 개발할 수 있으며, 이때 사전에 정의된 영역 할당자 인터페이스 함수들을 구현한 후, 이 함수 테이블을 영역 관리자가 제공하는 API 함수를 통해 등록할 수 있다. 또한 사용자는 사전에 이미 등록된 할당 정책들을 동적으로 제거할 수도 있다. 새로운 영역 생성 요청이 들어 올 경우 영역 관리자는 주어진 할당 정책의 초기화 멤버 함수를 호출하여 해당 정책 제어 블록을 초기화하게 한다. 이 후 각 영역별 메모리 할당/반환 요청이 들어 올 경우 영역 관리자는 해당 할당 정책의 할당/반환 멤버 함수를 호출한다.

3.2 실시간 영역 할당 정책

하나의 메모리 할당 정책으로 다양한 실시간 응용의 메모리 요구를 모두 만족시키기는 어렵다. 이는 실시간 응용별로 요구하는 메모리 크기 분포와 요구 패턴이 다양하며, 이에 따라 다양한 메모리 할당 정책을 필요로 하기 때문이다. 또한 동일한 응용 내에서도 서로 다른 특성을 가진 메모리 요구 패턴을 가질 수 있다. 따라서 제안 모델은 다양한 메모리 요구 패턴에 필요하다고 판단되는 기본적인 영역 할당 정책들을 제공한다. 이들은 사전에 정의된 동일한 크기의 블록들만을 할당하는 고정 크기 할당 정책과 다양한 크기의 블록들을 할당하는 가변 크기 할당 정책으로 분류될 수 있다((그림 1) 참조).

고정 크기 할당 정책으로 **RGN_FIXED**와 **RGN_FIXED_2** 등 두 가지 정책을 지원한다. 이들 두 정책은 고정된 크기의 단위 블록들을 할당하고 반환한다. 단위 블록 크기는 해당 영역 생성시 사용자가 직접 지정해야 한다. **RGN_FIXED**정책은 임의의 블록 크기를 지정할 수 있으나, **RGN_FIXED_2**의 경우 블록 크기가 반드시 2의 지수승 크기여야 한다. 이들 정책은 영역 내의 각 블록들을 하나의 테이블을 이용하여 관리하며, **RGN_FIXED**는 테이블의 프리 인덱스를 찾아 블록 크기를 곱하여 프리 블록의 주소를 찾는 반면, **RGN_FIXED_2**의 경우 단지 shift 명령을 통해 블록의 주소를 찾는다. 따라서 블록의 할당 및 반환은 **RGN_FIXED_2**가 더 빠르다. 사용자는 영역이 지원하는 고정된 블록 크기보다 같거나 작은 블록들을 요청할 수 있다.

고정 크기 할당 정책들은 블록의 할당 및 반환에 소요되는 시간이 항상 일정하고, 메모리의 외부 조각화가 발생하지 않는 장점이 있다. 그러나 영역이 지원하는 블록 크기보다 작은 블록을 요청할 경우 내부 조각화가 발생할 수 있다. 이 정책들은 I/O 버퍼, 제어 블록 등과 같이 블록의 크기가 사전에 알려져 있고, 시스템 실행동안 그 크기가 변하지 않는 객체를 위해 유용하게 사용될 수 있다.

제안 모델은 또한 실시간 응용의 메모리 요구 특성에 따라 선택적으로 골라 사용할 수 있는 다양한 가변 크기 할당 정책들을 제공한다. 다음은 이들의 특성과 장단점을 요약한 것이다.

- **RGN_ONCE** : 한번 할당된 블록은 절대 반환되지 않는 메모리 영역의 할당 정책으로 사용된다. 할당된 블록에 대한 오버헤드가 없으며, 가장 작은 정책 제어 블록 크기와 가장 빠른 응답시간을 가진다. 이 정책은 다목적용 영역에는 적합하지 않으며, 한번 할당 받고 반환하지 않는 사전에 그 용도가 잘 알려진 사용자 데이터를 할당 받을 때 적합하다.
- **RGN_HF**(half-fit) : 절반-적합(half-fit) [4] 전략을 사용

한다. 메모리 요구 패턴을 사전에 예측하기 어렵고 시간 제약이 엄격한 경성 실시간 응용의 메모리 영역 할당 전략으로 적합하다. 작은 크기부터 큰 크기까지 매우 넓은 메모리 요구 분포를 가진 영역에 적합하다. 메모리 관리 효율성은 낮지만 고정된 응답시간을 보장한다.

- **RGN_QHF**(quick-half-fit) : 64 워드(512 bytes) 이하의 작은 크기 메모리 요구가 대부분이지만, 큰 크기의 요구 분포가 매우 넓은 경우에 적합하다. **RGN_HF**와 마찬가지로 중간 크기의 메모리 요구가 많은 경우 메모리 사용 효율성이 저하될 수 있다. 응답시간은 한정된(bounded) 시간을 보장한다[7]. 64 워드 이하의 요구는 빠른-적합(quick-fit)[12] 전략을 사용하며, 그 이상의 요구는 절반-적합 전략을 사용한다.
- **RGN_QSHF**(quick-segregated-half-fit) : 메모리 요구 분포가 매우 넓거나, 또는 이를 예측하기 힘든 메모리 영역의 할당 전략으로 적합하다. 작은 크기부터 큰 크기까지 매우 넓은 메모리 요구 분포를 가진 영역에 적합하며, 메모리 관리 효율성이 뛰어나고 다양한 크기의 메모리 요구를 동시에 만족시킬 수 있다. 64 워드 이하의 요구는 빠른-적합 전략을 사용하며, 64 워드보다 크고 2730.5 KB보다 같거나 작은 중간 크기의 요구에는 적절한 크기 영역별로 프리 리스트를 관리하는 segregated-fit [13] 전략을, 그 이상의 요구는 절반-적합 전략을 사용한다. 최악의 경우 실행시간은 쉽게 예측 가능하고 한정된 시간을 보장한다[8]. 프리 블록 리스트 관리 오버헤드가 다른 정책들에 비해 상대적으로 크다.

이상의 가변 할당 정책들은 각 정책별 서로 다른 최악의 경우 실행시간과 영역 관리 오버헤드를 가지며, 또한 조각화 율에서도 상이한 면을 보인다. 이들은 각각 4장과 5장에서 다시 논의된다.

3.3 구현 전략

본 연구에서 제공하는 고정 크기 할당 정책들(**RGN_FIXED**와 **RGN_FIXED_2**)과 가변 크기 할당 정책 중 **RGN_EXACT**는 비교적 구현 자체가 간단하며, 따라서 단일 테이블 또는 리스트에 의해 간단히 구현될 수 있다. 그러나 나머지 가변 크기 할당 정책들(**RGN_HF**, **RGN_QHF**, **RGN_QSHF**)은 프리 블록들을 관리하기 위해 여러 개의 프리 리스트를 사용한다. 프리 리스트는 사전에 정의된 동일 크기 또는 비슷한 크기의 프리 블록들을 관리한다. 따라서 하나의 프리 리스트는 자신이 저장하여 관리하는 프리 블록들의 크기에 대한 일정 범위를 가지고 있으며, 해당 범위 내의 크기를 가지는 블록들만을 관리한다.

메모리 할당 요구가 있을 경우, 요구된 크기를 포함하는 프리 리스트에서 최초 블록을 선택한다. 만약 요구된 크기를 포함하는 프리 리스트가 빈(empty) 리스트라면, 이 리스트보다 더 큰 범위을 가진 non-empty 리스트들 중 가장 작은 크기 범위을 가진 리스트에서 할당할 블록을 선택한다. 할당을 위해 선택된 블록이 요구된 크기보다 클 경우, 이를 분할하여 할당하고 남은 블록은 다시 이를 수용하는 적절한 프리 리스트에 삽입한다. 반환된 메모리 블록의 인접한 블록이 프리 블록인 경우 반환된 블록과 합병된다. 반환한 메모리 블록이나, 합병한 결과 생성된 새로운 블록, 또는 분할하고 남은 블록들은 다시 적절한 프리 리스트에 삽입되어야 한다. 적절한 프리 리스트와 non-empty 리스트를 한정된(bounded) 시간 범위 내에 찾는 알고리즘과 이의 시간 분석은 참고문헌 [8]에 상세히 기술되어 있다.

3.4 메모리 할당 모델의 재구성

(그림 2)는 영역 관리를 위해 커널이 제공하는 사용자 인터페이스 함수이다. 사용자는 rtl_region_malloc()와 rtl_region_free()를 통해 특정 메모리 영역(rgn_desc)으로부터 원하는 메모리 블록을 할당 받고 반환할 수 있다. 여기서 rgn_desc는 사전에 이미 생성된 특정 메모리 영역에 대한 기술자(descriptor)이다.

```
/* 영역 관리를 위한 주요 커널 인터페이스 */
rtl_region_init (*rgn_desc, int policy, base_addr, region_size,
                 options, *parent_rgn_desc)
rtl_region_deinit (*rgn_desc)
rtl_region_malloc (*rgn_desc, size)
rtl_region_free (*rgn_desc, *ptr)
rtl_region_overhead (policy, length, options)
```

(그림 2) 영역 관리를 위한 주요 커널 인터페이스 함수들

메모리 영역은 rtl_region_init()를 통해 생성할 수 있다. 매개변수 rgn_desc는 새로 생성된 영역을 위한 기술자로서 영역에 대한 기본 정보를 담고 있는 데이터 구조에 대한 포인터이다. policy는 생성된 영역에 적용할 할당 정책이며, 현재 RTL_RGN_FIXED, RTL_RGN_FIXED_2, RTL_RGN_ONCE, RTL_RGN_HF, RTL_RGN_QHF, RTL_RGN_QSHF 등이 지정 가능하다. 각 할당 정책은 정책별 여분의 정보를 사용자로부터 요구할 수 있는데, 이는 options을 통해 전달된다. 예를 들어, RTL_RGN_FIXED나 RTL_RGN_FIXED_2정책의 경우 매 할당 요청시 고정된 크기의 메모리 블록을 할당하는데, options을 통해 이 크기를 지정할 수 있다. 따라서 이들 정책의 경우, rtl_region_malloc()의 size 매개변수는 무시된다. base_addr는 영역으로 사용할 연속된 메모리의 시작 주소이며, 이 주소부터 region_size 바이트가 해당 영역에 포함된다. 이때 지정한 연속된 메모리는 이미 기존의 다른 영역으로부터 할당 받은 메모리 블록이거나 프로그램상에서 전역 변수로 지정된 배열의 주소일 수도 있다. 이는 영역으로 사용할 메모리를 반드시 동적으로 할당받지 않고도 사전에 확보한 메모리의 일정 구역을 동적으

로 관리할 수 있음을 의미한다. 각 할당 정책은 사전에 생성된 특정 영역을 매개변수 parent_rgn_desc로 지정할 경우, 정책 제어 블록으로 사용할 메모리를 이 영역에서 할당받아 사용한다. 그러나 이를 지정하지 않을 경우 (즉, NULL), 해당 영역 내의 메모리 중 맨 처음 부분을 정책 제어 블록용으로 사용한다. 이 경우 사용자는 해당 영역으로 사용할 연속된 메모리의 크기를 정책 제어 블록 크기까지 고려하여 확보하여야 한다. 이를 위해 rtl_region_overhead() 함수를 제공하고 있다. 이는 각 정책별 정책 제어 블록 크기를 사전에 계산해 준다.

Linux의 경우 insmod/rmmod 명령어를 이용하여 동적으로 커널 모듈을 삽입하거나 삭제할 수 있다. 이러한 기능은 RT-Linux에서도 적용할 수 있으며, RT-Linux의 커널 모듈들은 모두 이러한 메커니즘을 통해 시스템에 추가된다. 특정 영역 할당 정책을 구현한 영역 할당자 역시 이 방식을 통하여 정적 또는 동적으로 시스템에 삽입/삭제될 수 있다.

(그림 3)의 영역 할당자 인터페이스는 영역 관리자와 영역 할당자 사이에 준수해야 할 할당 정책 인터페이스이며, 이는 할당 정책 알고리즘을 구현한 영역 할당자가 제공해야 하는 함수 프로토타입들의 집합이다. 영역 할당자는 이 인터페이스를 통해 영역 관리자와 통신할 수 있다.

```
/* 영역 할당자 인터페이스 */
typedef struct {
    void *(*malloc)(*rgn, size) ;
    int   (*free)(*rgn, *ptr) ;
    int   (*init)(*rgn, *base, length, options, *parent_rgn) ;
    void  (*deinit)(*rgn) ;
    unsigned long (*overhead)(length, options) ;
} rtl_alloc_policy_ops_t ;

/*  영역 할당자 등록 함수 */
rtl_register_alloc_policy(policy,rtl_alloc_policy_ops_t *ops) ;
rtl_unregister_alloc_policy(policy) ;
```

(그림 3) 영역 할당자 인터페이스와 관련 등록 함수

영역 할당자는 영역 할당자 인터페이스 함수들(rtl_alloc_policy_ops_t)을 정의한 후, 이를 영역 할당자 모듈 삽입시 관련 등록 함수(rtl_register_alloc_policy())를 통해 시스템에 등록하여야 한다. rtl_register_alloc_policy()의 매개 변수 policy는 등록하고자 하는 영역 할당자가 지원하는 할당 정책을 의미한다. 이외에도 시스템 개발자는 새로운 할당 정책을 필요로 하는 경우 영역 할당자 인터페이스를 준수하여 별도의 정책을 구현하여 추가할 수 있다. 등록 함수들은 기반 영역 관리자에 의해 제공된다. 사용자가 블록 할당 요청을 위해 해당 커널 인터페이스 함수를 호출할 경우, 커널 내의 영역 관리자는 해당 영역의 영역 할당자 인터페이스 함수들을 호출한다.

하나의 시스템에 서로 다른 할당 정책을 구현한 여러 개의 영역 할당자들을 등록할 수 있으며, 사용자는 영역 생성시 해당 영역에 가장 적합한 할당 정책을 선택적으로 골라 사용할 수 있다. 또한 사전에 정의된 영역 할당자 인터페이스를 준수할 경우 새로운 영역 할당 정책을 쉽게 구현해 삽입할 수 있고, 해당 응용에 불필요한 정책은 시스템에서 제거할 수도 있다.

3.5 제안 메모리 모델의 특징

제안 모델은 하나의 응용이 해당 응용의 특성에 맞는 여러 메모리 영역을 가질 수 있게 하고 각 영역별 서로 다른 할당 정책을 사용할 수 있도록 하기 위해, 기존의 단일 할당 정책을 제공하던 기법을 피하고, 다양한 할당 전략을 제공할 수 있는 방안을 제시했다. 이는 사용자로 하여금 각 메모리 영역별로 메모리 요구 특성을 정확히 파악하여 가장 적합한 정책을 선택할 수 있게 함으로써, 메모리 사용 효율성을 극대화하고자 함이다.

이를 위해서 기존의 단일 메모리 관리 모듈을 영역 관리자와 각 정책별 구현 모듈로 분리하는 두 단계 계층적 구조를 채택했다. 이러한 구조는 정책과 메커니즘의 분리(policy/mechanism separation) 기법 [14]을 제공한다. 영역 관리자가 제공하는 영역 할당 정책을 할당 메커니즘(정책을 구체적으로 구현한 영역 할당자)으로부터 분리함으로써, 시스템 개발자는 동일한 할당 정책을 새로이 다른 알고리즘을 사용하여 구현할 수 있다. 또한 이 구조는 시스템 재구성을 보다 유연하게 해준다. 본 연구에서는 다수개의 할당 정책들을 사전에 구현하여 제공하고 있으며, 시스템 구축자는 매번 기존 정책들을 새로이 구현할 필요 없이 이미 제공된 다양한 정책들 중 해당 응용에 가장 적합한 것을 선택하여 시스템을 재구성할 수 있다. 뿐만 아니라, 사전에 정의된 인터페이스를 준수할 경우 새로운 정책을 쉽게 구현해 삽입할 수 있고, 해당 응용에 불필요한 정책은 시스템에서 제거할 수도 있다.

4. 영역 관리 오버헤드(Overhead)

본 절에서는 제안 메모리 모델이 지원하는 다양한 영역 할당 정책별 별도로 추가되는 메모리 오버헤드와 최악의 경우 실행시간을 측정한 결과를 제시하고 이를 분석하고자 한다. 이를 통해 제안 모델의 타당성을 검증하고자 한다. 본 연구에서 제안한 메모리 할당 모델과 다양한 영역 할당 정책들은 Linux 2.2.14를 기반으로 하는 RT-Linux 2.2 상에 구현되었다. 기반 시스템은 Pentium II MMX 233MHz 프로세서와 128MB RAM, 33MHz PCI bus를 탑재한 시스템이다.

4.1 메모리 오버헤드

<표 1>은 제안 모델을 기반으로 각 정책별 이를 구현한 영역 할당자 크기, 해당 정책이 관리하는 프리 리스트의 종류 및 수, 정책 제어 블록을 위한 메모리 오버헤드 등을 보인 것이다.

〈표 1〉 영역 할당 정책별 메모리 오버헤드

할당 정책	RGN_FIXED	RGN_FIXED_2	RGN_ONCE	RGN_HF	RGN_QHF	RGN_QSHF
영역 할당자 모듈 크기 (KB)	2.0	2.1	1.8	4.5	5.2	5.7
리스트 종류	table	table	single pointer	half	quick, half	quick, segregated, half
리스트 수	1	1	1	32	87 = 64+23	139 = 64+62+13
오버헤드 (bytes)	rgn_sz / unit_sz * 2	rgn_sz / unit_sz * 2	12	264	736	1,160

정책 제어 블록은 메모리 영역을 관리하기 위해 필요한 모든 정보들을 보관하는 데이터 구조이다. 이는 정책별로 서로 다른 크기를 가지며, 별도의 메모리 오버헤드로 고려되어야 한다.

표에서 **RGN_HF**, **RGN_QHF**, **RGN_QSHF**와 같은 가변 크기 할당 정책들은 하나 또는 그 이상의 프리 리스트 종류(quick, half, segregated lists)를 가진다. 이들은 할당 요청 크기에 따라 적용되는 세부적인 할당 방법이 서로 다르다는 것을 의미한다. 반면, 고정 크기 할당 정책들은 프리 블록들을 링크로 직접 연결하여 관리하는 것이 아니라, 별도의 테이블을 두고 이 테이블의 인덱스를 활용하여 연결한다. 이는 메모리 관리시 프리 블록들에 대한 불필요한 접근을 삼가고 관리 테이블만 접근함으로써, 메모리 접근 지역성(locality)를 증가시키고 캐시(cache)의 적중율(hit ratio)을 향상시키기 위함이다.

RGN_ONCE를 제외한 가변 크기 할당 정책들이 사용하는 메모리 블록의 헤드는 모두 동일한 구조를 가진다. 블록 헤드는 프리 또는 사용 중인 블록들을 관리하기 위해 필요한 최소한의 데이터 구조로서, 이중 연결 포인터와 두 개의 경계 태그(boundary tag)를 가지며, 총 16 bytes이다. 그러나 사용 중인 블록 헤드는 개선된 경계 태그 기술 [15]을 사용할 경우, 프리 블록 헤드와는 달리 이중 연결 포인터와 하나의 경계 태그만 사용한다 (총 12 bytes). 할당된 블록의 주소는 *double* 실수 값을 고려하여 항상 8 bytes 배수 주소 값으로 조정(alignment)한다. 이러한 블록 헤드 구조체와 경계 태그 오버헤드가 메모리 조각화에 미치는 영향은 5.2절에서 논의된다. 고정 할당 정책들은 블록 헤드 구조체와 경계 태그 기술을 사용하지 않으므로 이로 인한 오버헤드는 없다. 그러나 관리하는 영역의 블록의 수가 증가할수록 정책 제어 블록의 오버헤드가 상대적으로 증가한다. 표에서 rgn_sz는 영역의 크기이며, unit_sz는 사전에 정의된 고정 블록들의 크기이다.

각 정책별 이를 구현한 영역 할당자의 목적 코드 크기는 위의 표에서 보인 바와 같다. 제안 모델은 기존의 단일 모듈로 구성되던 메모리 관리 모델을 지양하고 영역 관리자와 영역 할당자로 분리하는 두 단계 계층적 구조를 채택했다. 사용자와 영역 할당자간의 중간 연결자 역할을 하는 영역 관리자는 영역 할당자의 등록 및 제거, 새로운 영역의 생성 및 삭제, 정책별 해당 멤버 함수의 호출 등을 담당한다. 영역 관리자 모듈은 기존 단일 메모리 관리 모델에 비해 두 단계 계층적 구조를 가진 제안 모델을 위해 별도로 추가된 모듈이다. 따라서 이 모듈은 제안 모델의 오버헤드로 간주될 수 있으며, 그 목적 코드 크기는 약 1.5 KB 정도이다. 그러나 이 정도의 오버헤드는 동일한 사용자 인터페이스를 제공하면서도 다양한 영역 할당 정책을 제공할 수 있는 제안 모델의 유연성과 재구성 측면을 고려할 때 충분히 수용 가능하다고 판단된다.

4.2 최악의 경우 실행시간 (WCET)

실시간 시스템을 위한 메모리 할당 알고리즘들은 관리해야 할 영역의 전체 크기에 상관없이 항상 일정한(constant) 실행시간을 보장하거나 또는 최악의 경우 실행시간을 한정(bounding)시킬 수 있어야 한다. 이러한 특성은 알고리즘 상의 시간 복잡도(time complexity)와 이를 구현했을 때의 WCET이 서로 다른 양상을 보일 수 있기 때문에 매우 중요한 평가 요소이다. 예를 들어, 이진 버디 시스템(binary buddy system) [16]은 알고리즘 복잡도가 $O(1)$일지라도, 관리하는 메모리 영역의 용량이 증가함에 따라 WCET도 함께 증가한다[7]. 이유는 이진 버디 시스템에서 블록의 분할 및 합병시 매번 최악의 경우 32번(워드의 비트 수)의 분할 및 합병을 수행해야 하며, 이 과정에서 각 프리 리스트에 블록의 삽입/삭제를 위해 메모리 전반에 걸친 광범위한 접근이 이루어지기 때문이다. 따라서 본 연구에서는 이러한 영향을 배제하기 위해 고정 크기 할당 정책들은 프리 블록들간의 직접적인 링크를 피하고, 테이블과 이의 인덱스를 이용한 프리 블록 관리 방법을 채택했으며, 가변 크기 할당 정책들은 WCET을 한정시킬 수 있는 알고리즘들을 채택했다[8].

<표 2>는 영역 할당 정책별 기존 단일 메모리 관리 모듈로 구현되었을 경우와 제안 모델을 기반으로 구현되었을 경우의 할당 및 반환에 소요된 WCET을 보인 것이다. 실험에 사용된 시스템은 Pentium II MMX 233MHz 프로세서와 128MB RAM, 33MHz PCI bus를 탑재한 시스템이며,

실행시간은 펜티엄 프로세서가 제공하는 64-bit TSC(time stamp counter)를 이용하여 측정하였다.

〈표 2〉 영역 할당 정책별 최악의 경우 실행시간 (μ sec)

할당 정책		RGN_FIXED	RGN_FIXED_2	RGN_ONCE	RGN_HF	RGN_QHF	RGN_QSHF
기존 단일 모듈로 구현되었을 경우	할당	3.3	3.2	3.0	12.8	15.2	17.7
	반환	3.2	2.8	N/A	10.0	10.3	10.5
제안 모델상에서 구현 되었을 경우	할당	3.8	3.6	3.5	13.2	15.6	16.2
	반환	3.6	3.3	N/A	10.4	10.8	10.9

표에서 고정 크기 할당 정책보다는 가변 크기 할당 정책(RGN_ONCE는 예외)이 상대적으로 실행시간이 길며, 또한 가변 크기 할당 정책 중에서도 복합 정책(RGN_QSHF, RGN_QHF)이 단일 정책(RGN_HF)보다 상대적으로 길게 나타난다. 이는 관리해야 할 리스트 종류와 리스트 수가 증가하기 때문이다. 고정 크기 할당 정책과 RGN_ONCE 정책은 항상 고정된 실행시간을 보이지만, RGN_ONCE를 제외한 가변 크기 할당 정책들은 가변적인 실행시간을 보이지만 최악의 경우 실행시간은 항상 일정하다. 또한 이진 버디 시스템과는 달리 관리해야 할 영역 크기와 상관없이 항상 일정한 최악의 경우 실행시간을 보였다.

전체적으로 기존 단일 모듈로 구현되는 것보다 제안 모듈상에서 구현되었을 경우가 약 0.4~0.5 μsec 정도 더 소요된다. 이는 사용자의 요청을 받은 영역 관리자가 해당 영역 할당자를 호출하는데 소요된 시간으로서 두 단계 계층적 구조를 가진 제안 모델의 실행 오버헤드로 분석된다. 그러나 이 정도의 실행 오버헤드는 빠른 실행시간을 요하고 태스크 집합의 총 프로세서 이용률이 1에 가까운 경성(hard) 실시간 시스템이 아닌 이상 수용 가능하다고 판단된다.

5. 영역 관리 효율성 분석

본 절에서는 제안 모델이 지원하는 다양한 영역 할당 정책별 영역 관리 효율성을 비교하기 위해, 전체 메모리 용량을 제한한 채 인위적으로 생성된 메모리 할당 및 반환 패턴을 사용하여 메모리 조각화 율을 측정했다[4, 12, 17]. 본 절에서는 실험에 사용된 시뮬레이션 방법을 기술하고, 이 방법에 의해 실시된 실험 결과를 제시한 후, 이를 분석한다.

5.1 실험 방법

제안한 모델이 지원하는 정책별 성능을 비교하기 위해 본 연구에서 사용하는 평가 기준은 메모리 할당 및 반환에 소요되는 내부 조각화 율, 외부 조각화 율, 총 조각화 율, 할당 실패 율 등이다.

조각화 율은 DSA 알고리즘(할당 정책)이 적용되는 응용 프로그램, 라이브러리, 운영체제와의 상관 관계에 따라 다양하게 정의될 수 있다[18]. 본 실험에서는 실시간 시스템의 특성에 가장 적합한 참고 문헌 [19]의 방법을 따르기로 했다. 따라서 내부 조각화 율은 $IF = A/R$로 정의한다. 여기서 A는 DSA 알고리즘이 할당한 총 메모리 양이고, R은 사용자가 요청한 총 메모리 양이다. 또한 외부 조각화 율은 할당이 실패할 때마다 측정되며, $EF = M/A$로 정의한다. 여기서 M은 실험에 사용된 총 메모리 용량(사전에 고정)이고, A는 할당이 실패할 때까지 할당된 총 메모리 양이다. 각 실험에서는 여러 번의 할당 실패가 발생할 수 있으며, 그 때마다 외부 조각화 율을 계산하여 이를 평균한 값이 그 실험에서의 외부 조각화 율이다. 총 조각화 율은 내부 조각화 율과 내부 조각화 율로 인해 발생하는 메모리 손실로서, 다음과 같이 정의한다.

$$TF = IF * EF = (A/R) * (M/A) = M/R$$

따라서 총 조각화 율은 총 메모리 용량과 사용자가 요청한 순수한 메모리 양과의 상대적인 비율로 나타난다. 이는 곧 DSA 알고리즘이 사용자가 요청한 메모리 양보다도 메모리 관리를 위해 얼마 만큼의 메모리 용량을 더 필요로 하는가를 의미하며, DSA 알고리즘의 메모리 사용 효율성을 나타내는 지표이기도 하다.

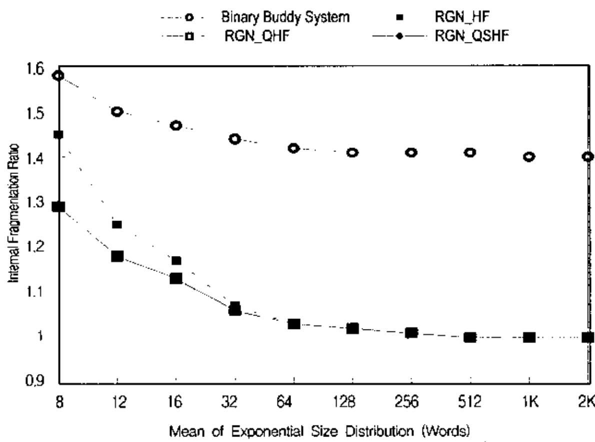
기존의 연구에 따르면 메모리 할당 시뮬레이션은 요구된 블록의 할당 크기, 할당된 블록의 시스템 내의 존속 기간, 그리고 할당 요구가 들어오는 시간 간격 등 세가지 분포에 상당한 영향을 받는다[17]. 본 연구에서는 참고 문헌 [4]와 같이 할당 크기 분포를 위해 지수(exponential) 분포와 균등(uniform) 분포 등 두 가지 확률 분포를 사용한다. 할당 크기 분포의 평균은 8, 10, 12, 14, 16, 32, 64, 128, 256, 512, 1024, 2048 워드로 다양하게 변화시켰다. 그러나 이때 사용되는 총 메모리 용량은 항상 32K 워드로 고정시켰다. 할당된 메모리 블록의 시스템 내의 존속 기간 분포는 균등 분포를 따르며, 5에서 15 사이의 시간 단위 범위 내에서 존속하며, 평균은 10 시간 단위이다. 메모리 요구 도착 시간 간격 분포는 지수 분포를 따른다. 따라서 시뮬레이션은 $M/G/\infty$ 큐잉(queuing) 모델을 따르며, 시스템 내에 존속하는 할당된 메모리 블록의 개수는 평균 값 λ/μ를 가지는 포아송(Poisson) 분포를 따른다. 여기서 $1/\lambda$는 요구 도착 시간 간격 분포의 평균 값이며, $1/\mu$는 할당된 블록의 존속 기간 분포의 평균 값이다. 실험에 사용되는 총 메모리 용량은 항상 고정되어 있으므로, 평균 할당 크기가 결정되면 시스템 내에 존속하는 할당된 메모리 블록의 평균 개수 λ/μ 값을 결정할 수 있다.

λ/μ이 결정되면 평균 존속 기간은 $10(=1/\mu)$으로 항상 고정되어 있으므로 평균 요구 도착 시간 간격 $1/\lambda$를 결정할 수 있다.

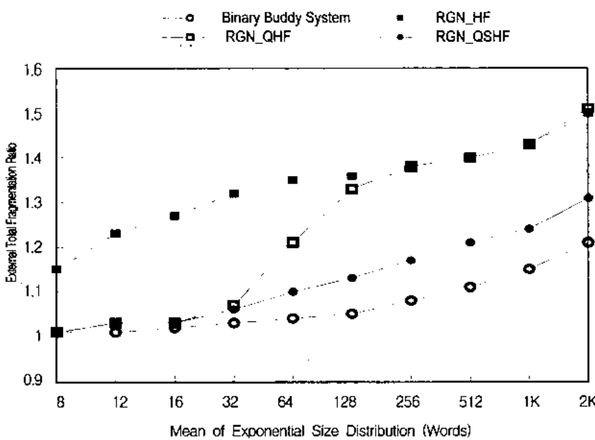
5.2 실험 결과 및 분석

(그림 4)와 (그림 5)는 각각 앞서 기술한 큐잉 모델에 따른 평균 할당 크기별 가변 크기 할당 정책들의 내부 조각화 율과 외부 조각화 율을 보인 것이다.

비교를 위해 이진 버디 시스템 [16]의 조각화 율도 함께 측정하였다. Iyengar는 참고 문헌 [19]에서 가중치(wighted) 버디 [20]와 더블(double) 버디 [21]의 경우 이진 버디 시스템보다 조각 율은 근소한 차이로 낮지만 응답 시간은 상당히 느리다는 실험 결과를 밝혔다. 또한 최악의 경우 실행시간 변동이 심하므로 실시간 시스템에 적용하기 곤란한 문제점을 가지고 있다. 따라서 본 연구에서는 대표적인 이진 버디 시스템만을 고려하기로 한다. 본 실험에서 하나의 워드는 8 bytes이며, 주소 조정과 블록 헤드를 위해 별도로 더 할당된 메모리 공간은 내부 조각화 율 계산에 모두 포함시켰다.

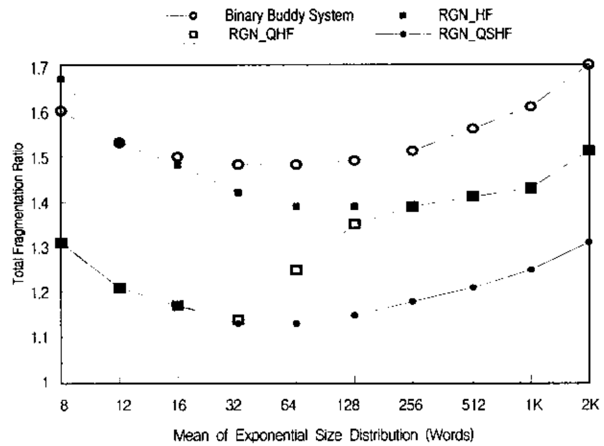


(그림 4) 내부 조각화 율 (지수 분포)

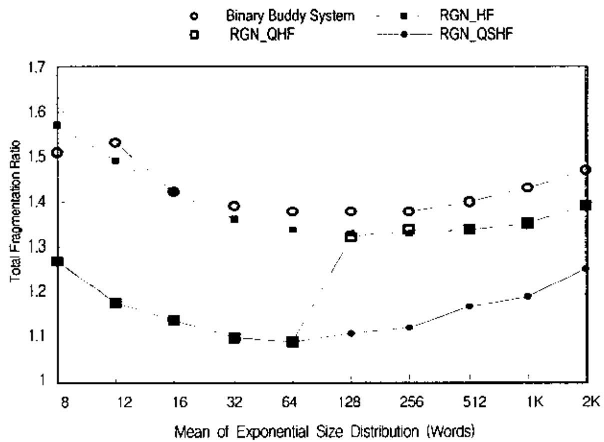


(그림 5) 외부 조각화 율 (지수 분포)

각 그림에서 할당 크기 분포는 지수 분포를 가진다. 이미 예측되는 바와 같이 이진 버디 시스템의 내부 조각화 율은 제안 모델의 정책들에 비해 상대적으로 가장 높지만, 외부 조각화 율은 상대적으로 가장 낮다. **RGN_QSHF**는 가장 낮은 내부 조각화 율을 보인 반면, 외부 조각화 율은 이진 버디 시스템 보다는 높지만 전반적으로 급격한 변화 없이 이와 유사한 패턴을 보인다. 이는 작은 크기의 메모리 요구에 대해 빠른-적합 전략을 사용하고, 중간 크기는 segregated-fit 전략을, 큰 크기는 절반-적합 전략을 사용하는 **RGN_QSHF**의 특성으로 인한 결과이다. 이러한 현상은 총 조각화 율(그림 6)에서 뚜렷이 확인할 수 있다.



(그림 6) 지수 분포 때의 총 조각화 율



(그림 7) 균등 분포 때의 총 조각화 율

(그림 6)에서 **RGN_QSHF**는 전 구간에 걸쳐 이진 버디 시스템 및 **RGN_HF** 보다 상대적으로 낮은 조각화 율을 보인다. **RGN_QSHF**와 **RGN_QHF**가 작은 크기(32 워드 이하)에서 **RGN_HF** 보다 낮은 조각화 율을 보이는 것은 quick-fit 전략의 효과이다. 또한 중간 크기(64 워드 이상)에서부터 **RGN_QSHF**와 **RGN_QHF**의 조각화 율이

차이가 나는 것은 RGN_QSHF의 segregated-fit 전략의 효과이다.

할당 크기 분포가 균등 분포를 가지는 경우에도 유사한 패턴을 보임을 (그림 7)에서 확인할 수 있다. RGN_HF 정책과 이진 버디 시스템의 총 조각화 율은 참고 문헌 [4]의 실험 결과와 유사한 수치를 보인다. RGN_QSHF 정책은 전 구간에 걸쳐 가장 낮은 조각화 율을 보인 반면, 이진 버디 시스템은 가장 높은 조각화 율을 보인다. RGN_HF 정책은 평균 크기가 작을수록 이진 버디 시스템과 유사한 조각화 율을 보인다. RGN_QHF 정책은 평균 크기가 작은 경우(32 워드 이하) RGN_HF와 같고, 평균 크기가 큰(256 워드 이상) 경우 RGN_QSHF와 같아 진다. 평균 크기가 계속해서 커진다면 RGN_QSHF의 조각화 율은 궁극적으로 RGN_HF와 같아 질 것으로 예상된다.

이상의 조각화 율 실험에서 특이한 사항은 평균 크기가 증가하면서 비율이 감소하다가 일정 크기를 지나면서 다시 증가한다는 사실이다. 이러한 현상은 평균 할당 크기가 작을수록 메모리 블록 관리를 위해 사용하는 블록 헤드 구조체의 오버헤드가 상대적으로 증가한 것으로 판단된다. 이러한 블록 헤드 구조체로 인한 부하는 참고 문헌 [18]에서 이미 지적된 사항이기도 하다.

다양한 응용의 메모리 요구 분포가 거의 대부분 30~40 워드 범위 내라는 연구 결과 [12, 13, 18, 19, 22, 23]를 고려해 볼 때, 실시간 응용 프로그램의 메모리 요구 분포도 이 범주에서 크게 벗어나지 않을 것으로 판단된다. 전반적으로 조각화 율에서 RGN_QSHF가 가장 뛰어난 메모리 관리 성능을 발휘하지만, <표 1>과 <표 2>에서와 같이 상대적으로 증가된 메모리 오버헤드와 최악의 경우 실행시간 오버헤드가 따른다. 따라서 실시간 응용의 메모리 요구 분포를 사전에 예측 가능한 경우에는 RGN_HF, RGN_QHF 등의 선택도 고려 할 수 있다. 이러한 영역 할당 정책의 선택은 3.2절에서 제시된 바와 같이 해당 메모리 영역을 사용하는 태스크의 시간 엄격성, 실행시간의 가변성(jitter), 메모리 요구 분포의 예측 가능성 등을 고려하여, 가장 적절한 정책을 선택하여야 한다.

6. 결 론

본 연구에서는 RT-Linux를 위한 메모리 할당 모델을 제안했다. 또한 폭 넓은 실시간 응용의 다양한 요구에 적합한 여러 실시간 영역 할당 정책들을 구현하여, 이를 기존 시스템에 쉽게 적용할 수 있는 재구성 방안도 함께 제시했다. 이를 바탕으로 시스템 설계자는 RT-Linux상에서 제안 모델이 기본적으로 지원하는 다양한 실시간 영역 할당 정책들 중, 해당 응용에 가장 적합한 정책들을 선택하여 시스템을 재구성할 수 있다. 이는 기존의 단일 메모리 관리 모듈을 영역 관리자와 각 정책별 구현 모듈로 분리하는 두 단계 계층적 구조를 채택했기 때문에 가능해졌다. 이러한 구조는 정책과 메커니즘의 분리 기법을 제공한다. 할당 정책을 할당 메커니즘으로부터 분리함으로써, 시스템 개발자는 동일한 할당 정책을 새로이 다른 알고리즘을 사용하여 구현할 수 있다. 또한 사전에 정의된 인터페이스를 준수할 경우 새로운 정책을 쉽게 구현해 삽입할 수 있고, 해당 응용에 불필요한 정책은 시스템에서 제거할 수도 있다.

제안 모델이 정의한 메모리 영역은 사전에 사용자가 지정한 제한된 크기의 연속된 메모리 구역이다. 이는 사용자의 잘못된 메모리 사용량 예측 또는 조각화로 인한 메모리 고갈 문제를 유발시킬 수 있다. 현재 이러한 문제는 예외처리자(exception handler)를 통하여 해결하고 있지만 궁극적인 해결 방안은 될 수 없다. 따라서 자동으로 시스템 메모리를 더 할당 받아 해당 영역의 크기를 확장시키는 방안과 이때 발생할 수 있는 최악의 경우 실행시간의 변동을 어떻게 한정시킬 것인가에 대한 추가 연구를 계속할 계획이다. 또한 본 연구에서 제공하는 다양한 메모리 할당 정책들이 어떠한 응용에 가장 적합한지 실시간 시스템 개발자들이 결정하기 어려울 수 있고, 처음에 결정한 할당 정책이 현재 시스템 상태에 따라 적합성이 달라질 수도 있으므로, 향후 현재의 시스템 상태에 따라 자동적으로 할당 정책을 선택하거나 바꾸어 주는 기법에 대한 연구가 더 이루어져야 할 것이다.

참 고 문 헌

[1] Michael Barabanov, "A Linux-based Real-Time Operating System," Master thesis, New Mexico Institute of Mining and Technology, June 1997.

[2] Yu-Chung Wang and Kwei-Jay Lin, "Implementing a General Real-Time Scheduling Framework in the RED-Linux Real- Time Kernel," *Proc. of 20th IEEE Real-Time Systems Symposium*, Dec. 1998.

[3] R. Hill, B. Srinivasan, S. Pather, and D. Niehaus, "Temporal Resolution and Real-Time Extensions to Linux," Technical Report ITTC-FY98-TR-11510-03, Department of Electrical Engineering and Computer Sciences, University of Kansas, June 1998.

[4] Takeshi Ogasawara, "An Algorithm with Constant Execution Time for Dynamic Storage Allocation," *Proc. of 2nd*

Intn. Workshop on Real-Time Computing Systems and Applications, pp.21-25, 1995.

[5] Kelvin D. Nilsen and Hong Gao, "The Real-Time Behavior of Dynamic Memory Management in C++," *Proc. of Real-Time Technology and Application Symposium*, pp.142-153, 1995.

[6] Ray Ford, "Concurrent Algorithms for Real-Time Memory Management," *IEEE Software*, Vol.5, Issue 5, pp.10-23, 1988.

[7] 정성무, 유해영, 심재홍, 김하진, 최경희, 정기현, "예측 가능한 실행 시간을 가진 동적 메모리 할당 알고리즘", 한국정보처리학회논문지, 제7권 제7호, pp.2204-2218, Jul. 2000.

[8] 정성무, 유해영, 심재홍, 박승규, 최경희, 정기현, "다양한 할당 정책을 지원하는 실시간 동적 메모리 할당 알고리즘", 한국통신학회논문지, 제25권 제10B호, pp.1648-1664, Oct. 2000.

[9] 심재홍, "다양한 실시간 스케줄러를 지원하기 위한 커널 구조화 및 재구성 방안", 공학 박사학위 논문, 아주대학교 대학원 컴퓨터공학과, Feb. 2001.

[10] Integrated Systems, *pSOSystem Programmer's Reference*, pp.2-17~2-20 1997.

[11] Microtec Division, *VRTX User's Guide*, Mentor Graphics Corporation, pp.4-9~4-21, 1998.

[12] Charles B. Weinstock, "Quick Fit : An Efficient Algorithm for Heap Storage Allocation," *SIGPLAN Notices*, Vol.23, No.10, pp.141-148, 1988.

[13] Doug Lea, *Implementation of malloc*, See also the short paper on the implementation of this allocator, Available at http : //g.oswego.edu.

[14] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Sulf, "Policy/Mechanism Separation in HYDRA," *Proc. of Symposium on Operating Systems Principles*, Nov. 1975.

[15] Thomas Standish, *Data Structure Techniques*, Addition-Wesley, Reading, Massachusetts, 1980.

[16] J. L. Peterson and T. A. Norman, "Buddy Systems," *Communications of the ACM*, Vol.20, No.6, pp.421-431, 1977.

[17] J. E. Shore, "Anomalous Behavior of the Fifty-percent Rule in Dynamic Memory Allocation," *Communications of the ACM*, Vol.20, No.11, pp.812-820, 1977.

[18] Mark S. Johnstone and Paul R. Wilson, "The Memory Fragmentation Problem : Solved?," *Proc. of Intn. Symposium on Memory Management*, pp.26-36, 1998.

[19] Arun Iyengar, "Scalability of Dynamic Storage Allocation Algorithms," *Proc. of 6th Symposium on the Frontiers of Massively Parallel Computing*, pp.223-232, 1996.

[20] K. K. Shen and J. L. Peterson, "A Weighted Buddy Method for Dynamic Storage Allocation," *Communications of the ACM*, Vol.17, No.10, pp.558-562, 1974.

[21] David S. Wise, "The Double Buddy-System," Technical Report 79, Computer Science Department, Indiana University, Bloomington, Indiana, 1978.

[22] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles, "Dynamic Storage Allocation : A Survey and Critical Review," *Proc. of Intn. Workshop on Memory Management*, pp.1-126, 1995.

[23] Arun Iyengar, "Parallel Dynamic Storage Allocation Algorithms," *Proc. of 5th IEEE Symposium on Parallel and Distributed Processing*, pp.82-91, 1993.

[24] Marshall Kirk McKusick and Michael J. Karels, "Design of a General Purpose Memory Allocator for the 4.3 BSD UNIX Kernel," *Proc. of the San Francisco USENIX Conference*, pp.295-303, June 1988.

심 재 홍

e-mail : jhshim@cesys.ajou.ac.kr
1987년 서울대학교 전산과학과 졸업(학사)
1989년 아주대학교 컴퓨터공학과 졸업(석사)
2001년 아주대학교 컴퓨터공학과 졸업(박사)
1989년~1994년 서울시스템(주) 공학연구소
2001년~현재 아주대학교 정보통신전문대학원 BK21 전임연구원
관심분야 : 운영 체제, 분산시스템, 실시간 및 멀티미디어시스템

정 석 용

e-mail : syjung@dongyang.ac.kr
1987년 서울대학교 전산과학과 졸업(학사)
1993년 한국과학기술원 정보및통신공학과 졸업(석사)
1987년~1995년 LG정보통신㈜ 중앙연구소
1996년~현재 동양공업전문대학 전산경영기술공학부 조교수
관심분야 : 운영 체제, 분산시스템, 실시간 및 멀티미디어시스템

강 봉 직

e-mail : bjkang@dongyang.ac.kr
1989년 서울대학교 전산과학과 졸업(학사)
1991년 한국과학기술원 전산학과 졸업(석사)
1987년~1994년 포스데이타㈜ 기술연구소
1995년~현재 동양공업전문대학 전산경영기술공학부 조교수
관심분야 : 운영 체제, 분산시스템, 실시간 및 멀티미디어시스템

최 경 희

e-mail : khchoi@madang.ajou.ac.kr

1976년 서울대학교 수학교육과 졸업(학사)

1979년 프랑스 그랑데꼴 Enseeiht대학 졸업(석사)

1982년 프랑스 Paul Sabatier대학 정보공학부 졸업(박사)

1982년~현재 아주대학교 정보 및 컴퓨터공학부 교수

관심분야 : 운영 체제, 분산시스템, 실시간 및 멀티미디어시스템 등

정 기 현

e-mail : khchung@madang.ajou.ac.kr

1984년 서강대학교 전자공학과 졸업(학사)

1988년 미국 Ilinois주립대 EECS 졸업(석사)

1990년 미국 Purdue대학 전기전자공학부 졸업(박사)

1991년~1992년 현대반도체 연구소

1993년~현재 아주대학교 전기전자공학부 교수

관심분야 : 컴퓨터구조, VLSI 설계, 멀티미디어 및 실시간 시스템 등