논문 00-25-10B-2 한국통신학회논문지 '00-10 Vol.25 No.10B

다양한 할당 정책을 지원하는 실시간 동적 메모리 할당 알고리즘

정회원 정 성 무*, 유 해 영**, 심 재 홍*, 박 승 규*, 최 경 희*, 정 기 현*

A Real-time Dynamic Storage Allocation Algorithm Supporting Various Allocation Policies

Sung-Moo Jung*, Hae-Young Yoo**, Jae-Hong Shim*, Seung-Kyu Park*, Kyung-Hee Choi*, Gi-Hyun Jung* ***Regular Members***

요 약

본 연구에서는 다양한 메모리 할당 정책을 지원하는 실시간 동적 메모리 할당 알고리즘 QSHF(quick-segregated-half-fit)를 제안한다. 제안된 알고리즘은 메모리 할당 정책별로 차이는 있지만, 기본적으로 작은 크기의 메모리 요구에는 워드 크기별로 프리 리스트를 관리하는 exact(quick)-fit 전략을, 중간 크기의 요구에는 적절한 크기 영역별로 프리 리스트를 관리하는 good(segregated)-fit 전략을, 큰 크기의 요구에는 2의 거듭제곱 크기별로 프리 리스트를 관리하는 half-fit 전략을 사용한다. 따라서 사용자는 해당 실시간 응용 프로그램의 특성에 가장 적합한 메모리 할당 정책을 선택할 수 있다. 제안된 알고리즘의 시간 복잡도는 O(1)이며, 최악의 경우 실행 시간(worst case execution time: WCET) 역시 쉽게 측정 가능하다. 또한 본 연구에서는 요구된 메모리 크기에 적절한 프리 리스트를 예측 가능한 시간 내에 찾고, 찾은 리스트가 비어 있을 경우 다음 이용 가능한 프리 리스트를 고정 시간 내에 찾는 방안도 함께 제시한다.

제안된 알고리즘의 실용성을 확인하기 위해 각 정책별 메모리 사용 효율성을 측정하였다. 실험 결과 각 정책은 메모리 크기에 상관없이 고정된 WCET을 보장하지만, 메모리 사용 효율성과 리스트 관리 부하에서는 서로 상반되는 이점을 가진다는 사실을 확인하였다.

ABSTRACT

This paper proposes a real-time dynamic storage allocation algorithm, QSHF(quick-segregated-half-fit), that provides various memory allocation policies. The proposed algorithm adopts exact(quick)-fit policy that manages a free block list per each word size for memory requests of small size, good(segregated)-fit policy that manages a free list per proper range size for medium size requests, and half-fit policy that manages a free list per each power of 2 size for large size requests. The proposed algorithm has the time complexity O(1) and makes us able to easily estimate the worst case execution time (WCET). This paper also suggests two algorithms that finds the proper free list for the requested memory size in predictable time, and if the found list is empty, then finds next available non-empty free list in fixed time.

In order to confirm efficiency of the proposed algorithm, we simulated the memory utilization of each memory allocation policy. The simulation result showed that each policy guarantees the constant WCET regardless of memory size, but they have trade-off between memory utilization and list management overhead.

* 아주대학교 정보및컴퓨터공학부 ** 단국대학교 전산통계학과
논문번호 : 00237-0627, 접수일자 : 2000년 6월 27일

Ⅰ. 서 론

동적 메모리 할당(dynamic storage allocation: DSA)은 C나 C++와 같은 고 수준 프로그래밍 언어에서 사전에 그 크기를 결정할 수 없는 객체를 효과적으로 관리하기 위해 사용하는 유용한 프로그래밍 기술이다. 동적 메모리 할당은 또한 태스크나 프로세스보다 생명주기가 훨씬 짧은 메모리 객체를 동적으로 관리함으로써, 메모리의 사용 효율성을 증가 시킨다.

실시간 시스템에서 시간 제약(time constraints)을 어기지 않고 동적 메모리 할당을 효과적으로 지원하는 것은 매우 어려운 일이다. 실시간 태스크의 스케줄링 가능성을 측정하기 위해서는 사전에 최악의 경우 실행 시간(worst-case execution time: WCET)을 정확히 계산해야 하며, 이를 위해서는 실시간 DSA 알고리즘 역시 WCET을 손쉽게 예측할 수 있어야 한다.

대부분의 경성(hard) 실시간 시스템 개발자는 동적 메모리 할당이 실시간 시스템에서는 부적합 하다는 인식을 가지고 있다. 이는 시스템의 실행이 지속되면서 이용 가능한 메모리 공간이 점점 조각화(fragmented) 되어, 충분한 메모리가 있음에도 불구하고 궁극적으로는 더 이상의 메모리 할당을 진행할 수 없는 사태가 발생할 수 있기 때문이다. 또 다른 이유는 메모리 영역을 할당하거나 반환하는데 소요되는 최악의 경우 실행 시간(WCET)을 사전에 예측할 수 없다는데 있다[1-3].

그러나 본 연구팀은 실시간 시스템에서 동적 메모리 할당은 중요한 시스템 구성 요소 중의 하나라 판단하고, 다양한 메모리 할당 정책을 지원하면서도 예측 가능한 실행 시간을 가진 DSA 알고리즘을 제안하고자 한다. 동적 메모리 할당에서 조각화 또는 잘 못된 메모리 사용량 예측으로 인해 발생하는 메모리 고갈 문제는 일반적으로 오류 복구를 위한 예외 처리자(exception handler)를 통하여 해결하거나, 이를 시스템 관리자에게 통보하여 해결하게 한다[4, 5]. 그러나 본 연구에서는 동적 메모리 할당의 근본적인 문제인 메모리 조각 율을 낮추고, 메모리 할당 알고리즘의 시간 복잡도(time complexity) 및 WCET을 최소화하는데 초점을 둔다.

본 논문에서 제안하는 DSA 알고리즘은 다양한 메모리 할당 정책(policy)들을 제공하여, 사용자로 하여금 실시간 응용 프로그램의 특성에 가장 적합한 정책을 필요에 따라 선택할 수 있게 해 준다. 제안된 알고리즘은 할당 정책별 차이는 있지만, 기본적으로 작은 크기의 메모리 요구를 위해서는 워드(word) 크기별로 프리 리스트(free block list)를 관리하고, 중간 크기의 요구를 위해서는 적절한 크기 영역별로 프리 리스트를 관리하며, 큰 크기의 요구를 위해서는 2의 거듭제곱 크기별로 프리 리스트를 관리한다. 뿐만 아니라, 상대적으로 많은 프리 리스트를 관리하면서도, 요구된 메모리 크기에 가장 적절한 리스트를 예측 가능한 시간 내에 찾고, 찾은 리스트가 비어 있을 경우 다음에 이용이 가능한 리스트를 고정 시간 내에 찾는 방안도 함께 제시하였다. 따라서 다양한 메모리 할당 정책을 지원하고 특성이 다른 여러 리스트를 관리하면서도 쉽게 WCET을 예측할 수 있게 하였다.

본 논문의 구성은 다음과 같다. 2절에서는 실시간 시스템에 적용 가능한 기존 DSA 알고리즘들을 분석하고, 이들을 실시간 시스템에 적용하는데 있어 문제점을 논의한다. 3절에서는 다양한 정책을 지원하는 실시간 DSA 알고리즘을 제안한다. 4절에서는 제안된 알고리즘의 구현상의 문제와 관련한 몇 가지 전략을 제안하고, 이에 따른 시간 복잡도 및 WCET을 분석한다. 5절에서는 제안된 알고리즘의 다양한 정책별 메모리 관리 성능을 실험해 보고, 그 결과를 분석한다. 마지막 6절에서 향후 연구 계획에 대해 논의한다.

Ⅱ. 연구 배경

컴퓨터의 출현 이후 지금까지 고가의 시스템 자원 중의 하나인 메모리를 효과적으로 관리하기 위한 다양한 DSA 알고리즘들이 많은 연구자들에 의해 제안되었다[8,9]. 단일 프리 리스트(*single free list*)는 하나의 프리 리스트만을 가지고 전체 프리 블록들을 관리하는 방식이다 (이후 별도의 설명 없이 리스트, 블록이라 함은 각각 프리 리스트, 프리 블록을 의미함). 이 방식은 리스트를 순차적으로 탐색하는 *first-fit*, *next-fit*, *best-fit* 등과 같은 널리 알려진 메모리 할당 정책(policy)을 사용하였다[13,14]. 기존의 다목적용 동적 메모리 할당 연구들은 DSA 알고리즘이 평균적으로 얼마나 빨리 그리고 얼마나 효율적으로 메모리 블록들을 할당할 수 있느냐에 초점을 두었다. 그 결과 단일 프리 리스트에서는 평균 리스트 탐색 시간을 최소화하기 위해 트리 구조로 관리하는 구현 방법 [18-20]과 메모리 캐싱

(caching)을 이용하는 구현 방법[21-23] 등이 제안되었다. 그러나 여전히 최악의 경우 시간 복잡도가 $O(n)$[19], $O(log_2\ n)$[18], 또는 $O(log_2\ w)$[20]이므로, 실시간 시스템에 적용하기에는 WCET을 예측하기가 쉽지 않다. 여기서 n은 메모리내의 프리 블록의 수이고, w는 할당된 가장 큰 메모리 블록의 크기이다.

프리 블록들을 관리하기 위한 또 다른 방식은 여러 개의 *segregated* 리스트들을 사용하는 다중 프리 리스트(*multiple free lists*)이다. 각 segregated 리스트에는 사전에 정의된 동일 크기 또는 비슷한 크기의 블록들을 함께 관리한다. Segregated 리스트는 자신이 관리할 수 있는 블록들의 크기에 대한 일정 범위, 즉 리스트 영역(range)을 가진다. 메모리 요청 시 해당 요청 크기를 포함하는 영역을 가진 리스트에서 적절한 블록을 선택하여 할당해 준다.

이진 버디 시스템(*binary buddy system*)은 메모리 블록의 분할(splitting)과 합병(coalescing) 작업을 제한적이지만 효과적으로 지원하는 다중 프리 리스트의 변형된 알고리즘이다[13,15,16]. 이진 버디 시스템(그림 1. (a) 참조)의 경우 리스트 개수가 한 워드의 비트 수로 사전에 고정되므로, 시간 복잡도(time complexity)는 $O(1)$이다[1]. 문제는 다른 DSA 알고리즘과 비교 해 볼 때 내부 조각화(internal fragmentation) 율이 상대적으로 매우 높다(25%)는 것이다[9,13,16]. 따라서 이러한 문제를 해결하기 위해 fibonacci 버디[24], weighted 버디[25], double 버디[26] 등이 제안되었으나, 여전히 상대적으로 높은 조각화 율을 보였다.

*Quick-fit*은 다중 프리 리스트와 단일 프리 리스트가 복합된 알고리즘이다[10,11]. *MaxQL* 보다 작은 각각의 블록 크기 s(워드 배수)에 대해 상응하는 하나의 *quick* 리스트를 가진다. 따라서 quick 리스트의 영역 크기는 워드 배수이고, 동일 크기의 블록만 저장한다. *MaxQL*보다 큰 블록들은 크기에 상관없이 하나의 단일 프리 리스트로 관리된다 (그림 1. (b) 참조).

*MaxQL*보다 큰 블록들을 하나의 단일 프리 리스트로 관리하는 quick-fit의 문제점을 해결하기 위해, 단일 프리 리스트를 다시 여러 개의 segregated 리스트로 나누어 관리하는 *MFLF*(*multiple free list fit*)[6,7], *dlmalloc*(*Doug Lea's malloc*)[12] 알고리즘 등이 제안되었다. 이들 알고리즘은 *MaxQL*보다 큰 블록들을 영역 크기가 다른 연속된 여러 개의 segregated 리스트들에 나누어 관리하는 전략을 사용했다. 또한 실시간 제약성 측면에서 볼 때, MFLF는 즉시 합병이 아닌 지연 합병 방식을 사용하는 문제점이 있고, dlmalloc은 메모리 할당시 리스트 내의 가장 적절한 블록을 순차적 리스트 탐색(round-up이 아닌best-fit 전략)을 이용하여 찾아내는 문제점이 있다. 그러나 이들은 최악의 경우 여전히 단일 프리 리스트와 동일한 시간 복잡도를 가진다. 그러나 거의 대부분의 메모리 할당 및 반환이 quick 리스트에서 수행되기 때문에 평균 응답 시간과 메모리 사용 효율성 측면에선 우수한 성능을 보인다. 이러한 전략의 타당성은 다양한 응용 프로그램의 메모리 요구 크기를 조사한 결과 거의 대부분이 30-40 워드 크기 이하라는 최근의 여러 연구 결과[6-12]가 이를 뒷받침하고 있다.

*Half-fit*은 실시간 시스템용으로 개발된 DSA 알고리즘이다[1,17]. 각 리스트(*half* 리스트)의 영역 크기로 2의 거듭제곱을 사용하므로, 크기가 $[2^i, 2^{i+1})$ 사이인 블록들은 i를 색인으로 사용하는 리스트에 관

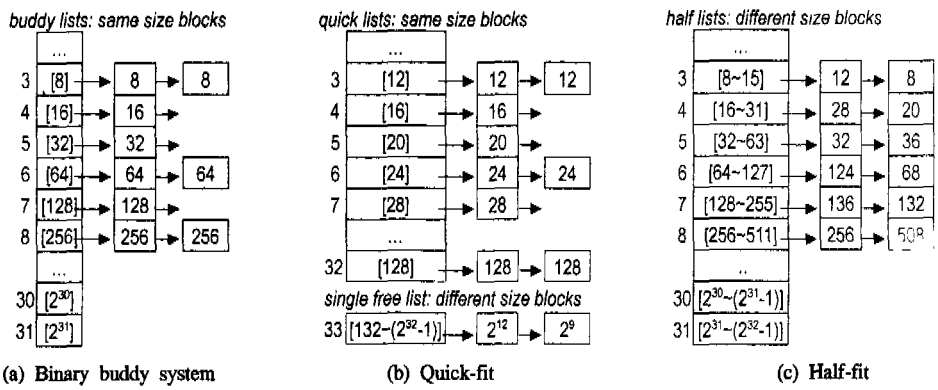


그림 1. 기존 동적 메모리 할당 알고리즘

리된다. 여기서]는 끝점을 포함하며,)는 끝점을 포함하지 않는다. Half-fit은 해당 리스트 내에서 적절한 크기의 블록을 찾기 위한 순차적 리스트 탐색을 지양하기 위해, 영역 (2^{i-1}, 2^i) 사이의 메모리 요청이 있을 경우 리스트 (i-1)에서 할당하는 것이 아니라, 리스트 i에서 할당하는 *round-up* 전략을 사용한다 (그림 1. (c) 참조). Half-fit 알고리즘에서는 시간 복잡도가 $O(1)$이면서 고정된 실행시간을 보장하며, 항상 고정된 개수(한 워드의 비트 수)의 리스트들을 가지는 장점이 있다. 그러나 리스트 영역 크기가 2의 거듭제곱 크기이므로 작은 크기의 메모리 요구가 많은 경우 상대적으로 높은 조각화 율을 보임을 본 연구의 실험에서 확인할 수 있다.

Ⅲ. QSHF (Quick-Segregated-Half-Fit) 동적 메모리 할당 알고리즘

앞 절에서 나열한 quick-fit, MFLF, dlmalloc, half-fit 알고리즘들의 단점을 보완하고 장점을 활용한다면, 각 알고리즘별 고유한 메모리 할당 정책(policy)을 제공할 수 있다. 본 절에서는 실시간 DSA 알고리즘 QSHF가 제공하는 다양한 메모리 할당 정책을 제시하고, QSHF의 메모리 할당 및 반환 전략을 기술한다. 또한 이러한 정책들을 지원하기 위한 프리 리스트들의 종류와 특성에 대해 기술하고, 이들 프리 리스트들의 특성에 따른 다양한 프리 리스트 그룹과 이들 그룹을 특징짓는 요소들에 대해 논의한다.

3.1 메모리 할당 정책(policies)

QSHF는 실시간 응용 프로그램의 메모리 요구 특성에 따라 선택적으로 골라 사용할 수 있는 다양한 메모리 할당 정책을 제공한다.

표 1은 QSHF가 제공하는 다양한 메모리 할당 정책과 이들의 특성을 보여 준다. 여기서 **DSA_QF**와 **DSA_HF**를 제외한 나머지 정책들은 두개 이상의 정책들이 결합된 복합 정책이다. 하나의 DSA 알고리즘으로 다양한 실시간 응용 프로그램의 메모리 요구를 모두 만족시키기는 어렵다. 일반적으로

표 1. QSHF가 지원하는 다양한 메모리 할당 정책

메모리 할당 정책	특 성
DSA_QF (quick-fit)	· 할당 가능한 최대 블록 크기 제한 · 응용 프로그램의 메모리 요구 패턴을 사전에 알고 있고, 모든 메모리 요구가 *MaxQL* 이하일 때 적합 · 시간 제약이 엄격한 경성(hard) 실시간 응용 프로그램에 적합 · 빠르고 고정(bounded)된 응답(response or execution time) 시간 보장 · 리스트 관리 부하(overhead)가 적고, 메모리 관리 효율성 높음
DSA_HF (half-fit)	· 할당 가능한 최대 블록 크기 제한 없음 · 메모리 요구 패턴을 사전에 예측하기 어렵고, 시간 제약이 엄격한 경성 실시간 응용 프로그램에 적합 · 작은 크기부터 큰 크기까지 매우 넓은 메모리 요구 분포를 가진 경우 · 메모리 관리 효율성은 낮지만, 고정된 응답 시간 보장
DSA_QSF (quick-segregated-fit)	· 할당 가능한 최대 블록 크기 제한 · 작은 크기의 메모리 요구가 대부분인 경우 · 응용 프로그램의 메모리 요구 패턴을 사전에 알고 있지만, **DSA_QF** 전략을 사용하기에는 리스트 관리 부하가 클 경우 · 시간 제약이 엄격한 경성(hard) 실시간 응용 프로그램에 적합 · 빠르고 고정된 응답 시간 보장
DSA_QHF (quick-half-fit)	· 할당 가능한 최대 블록 크기 제한 없음 · 작은 크기의 메모리 요구가 대부분이지만, 큰 크기의 요구 분포가 매우 넓은 경우 · 중간 크기가 많은 경우 메모리 사용 효율성 저하 · 응답 시간 가변, WCET은 쉽게 예측 가능함
DSA_QSHF (quick-segregated-half-fit)	· 할당 가능한 최대 블록 크기 제한 없음 · 메모리 요구 분포가 매우 넓거나, 또는 이를 예측하기 힘든 경우 적절 · 뛰어난 메모리 사용 효율성을 요하는 경우 · 다양한 응용 프로그램의 메모리 요구를 동시에 만족 · 응답 시간 가변, WCET은 쉽게 예측 가능함 · 리스트 관리 부하가 큼

실시간 응용 프로그램별로 요구하는 메모리 크기 분포와 요구 패턴이 다양하며, 이에 따라 다양한 메모리 할당 정책을 필요로 하기 때문이다. 따라서 본 연구에서는 다양한 메모리 할당 정책을 제공하되 필요에 따라 응용 프로그램의 특성에 맞는 적합한 정책을 선택할 수 있게 했다. 실시간 시스템 개발자는 해당 실시간 응용 프로그램의 메모리 요구 사항을 정확히 분석하여 가장 적절한 정책을 선택하고 시스템 구축시 이를 설정하여야 한다. 따라서 하나의 시스템에서는 하나의 메모리 할당 정책만을 사용할 수 있다.

지원되는 할당 정책들은 각각의 장점과 단점을 가지고 있다. 정책들은 표에서 보인 순서대로 점점 메모리 요구 분포가 다양한 응용에 적합하지만 관리 부하가 증가하면서 응답 시간도 함께 증가한다. 따라서 할당 가능한 최대 블록 크기가 제한되지만, 메모리 요구 패턴을 사전에 알 수 있으며 빠르고 고정된 응답 시간을 요하는 경성 실시간 응용 프로그램의 경우 **DSA_QF**를 사용할 수 있다. 반면, 메모리 요구 분포가 매우 넓거나(다양한 유형의 메모리를 요구) 또는 이를 예측하기 힘든 응용 프로그램의 경우, 관리 부하가 다른 정책에 비해 상대적으로 증가되지만 여전히 뛰어난 메모리 사용 효율성을 제공하고 WCET을 쉽게 예측할 수 있는 **DSA_QSHF**를 사용할 수 있다.

3.2 프리 블록(free blocks)의 할당 및 반환

QSHF는 프리 블록들을 관리하기 위해 메모리 할당 정책에 상관없이 여러 개의 프리 리스트를 사용한다. 리스트는 사전에 정의된 동일 크기 또는 비슷한 크기의 블록들을 저장한다. 따라서 리스트는 자신이 저장하여 관리하는 블록들의 크기에 대한 일정 범위, 즉 리스트 영역(range)을 가지고 있으며, 해당 영역 내의 크기를 가지는 블록들만을 관리한다. (이후 영역 크기에 대한 단위는 byte임) 리스트 개수, 리스트별 영역 크기 결정 및 메모리 분배 등은 다음 절에서 상세히 기술된다.

메모리 할당 요구가 있을 경우, 요구된 크기를 포함하는 영역을 가진 리스트에서 할당할 적절한 블록을 선택한다. 만약 요구된 크기를 포함하는 영역을 가진 리스트가 빈(empty) 리스트라면, 이 리스트의 영역보다 더 큰 영역을 가진 non-empty 리스트들 중 가장 작은 영역을 가진 리스트에서 할당할 블록을 선택한다. 할당을 위해 선택된 블록이 요구된 크기보다 클 경우, 이를 분할하여 할당하고, 남은 블록은 다시 이를 수용하는 적절한 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. 이러한 리스트와 non-empty 리스트를 찾는 알고리즘과 이의 WCET 분석은 4장에서 논의된다.

3.3 프리 리스트(free lists)와 프리 리스트 그룹(groups)

QSHF가 관리하는 모든 리스트들의 영역은 연속적이다. 즉, 임의의 연속된 리스트 l_i (여기서, i 1)와 l_{i+1}에 저장될 수 있는 블록의 가장 작은 크기를 각각 low_i, low_{i+1}라 할 때, 리스트 l_i의 영역은 [low_i, low_{i+1} - 1] 또는 [low_i, low_{i+1})로 표현된다. 여기서 (low_{i+1} - low_i)를 리스트 l_i의 영역 크기 또는 영역 증가 양이라 하며, 이를 inc_i로 표현한다. 따라서 각 리스트의 영역 크기와 영역 증가 양은 항상 동일하므로, 연속된 리스트는 항상 연속된 영역을 가진다.

QSHF가 관리하는 리스트들은 연속된 리스트들간의 영역 관계에 따라 크게 *quick* 리스트, *segregated* 리스트, *half* 리스트 등 3가지로 분류할 수 있다. Quick 리스트의 영역 크기는 항상 워드 크기이며, 리스트 영역 내의 가장 작은 크기(워드 크기 배수)의 블록들만을 저장한다. 즉, 동일한 크기의 블록들만을 관리하고, 연속된 quick 리스트들은 연속된 워드 크기 배수의 블록들을 저장한다 [그림 1. (b) 참조]. 이러한 quick 리스트들만 모아 놓은 리스트 그룹을 *quick* 리스트 그룹(또는 간단히 *quick* 그룹)이라 하고, 이를 L_{QL}로 표현한다. Quick 그룹은 주로 작은 크기의 블록을 관리하기 위해 사용되며, quick-fit 정책을 지원한다. 메모리 초기화시 사용자는 메모리 할당 정책과 함께 quick 그룹의 리스트 수(N_{QL})도 함께 설정하여야 한다.

Segregated 리스트는 리스트 영역에 속하는 다양한 크기의 블록들을 저장할 수 있다. 각 리스트의 영역 크기와 영역 증가량은 동일한 값이므로, 연속된 리스트들간의 영역은 서로 연속적이다. 그러나 모든 segregated 리스트가 동일한 영역 크기를 가지는 것은 아니다. 따라서 리스트들을 영역 크기에 따라 다시 여러 개의 그룹으로 나눌 수 있으며, 이들 각 그룹을 *segregated* 리스트 그룹(또는 간단히 *segregated* 그룹)이라 하고, 이를 L_k(여기서, k 1)로 표현한다. 각 그룹 내의 모든 리스트들은 동일한 영

역 크기를 가진다. Segregated 그룹들은 주로 quick 그룹의 영역보다 더 큰 크기의 블록을 관리하기 위해 사용되며, segregated-fit 정책을 지원한다. 따라서 이 그룹은 메모리 할당 정책에 따라 항상 quick 그룹과 연계되어 관리된다.

문제는 몇 개의 segregated 그룹을 사용하며, 이 때 각 그룹의 리스트 수와 영역 크기를 얼마로 할 것인가이다. 본 연구에서는 이에 대한 방안으로 참고 문헌 [12]와 유사한 방식을 사용하기로 했다. 즉, segregated 그룹의 개수를 n, 각 그룹의 리스트 수를 N_k, 각 그룹의 리스트 영역 크기를 inc_k라 할 때, 이들은 다음과 같이 결정된다. 여기서 N_{QL}은 quick 그룹의 리스트 수이며, 2^w는 워드 크기이다.

$$n = \lfloor log_2 N_{QL} \rfloor$$

$$N_1 = \lfloor N_{QL} / 2 \rfloor,\ N_k = N_{k-1} / 2 = 2^{n-k},$$
$$\text{where } 2 \le k \le n$$

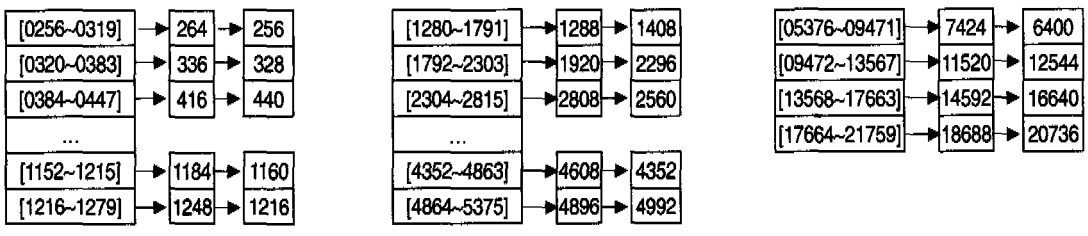
$$inc_k = 2^{w(k+1)},\ \text{where } 1 \le k \le n$$

즉, 연속된 각 그룹의 리스트 수는 반씩 감소시키는 대신, 리스트 영역 크기는 워드 크기를 곱해서 증가시킨다. 이는 작은 크기의 메모리 요구에는 리스트 영역을 보다 세분화 시켜 대처하는 반면, 큰 크기의 메모리 요구에는 리스트 수를 줄이면서 영역 크기를 상대적으로 증가 시킴으로써, 급격한 성능 저하 없이 유연하게 대처하기 위함이다. 예를 들어, 워드 크기가 8 bytes이고(즉, $w = 3$), quick 리스트 수(N_{QL})가 32일 때, 각 segregated 그룹의 리스트 영역 크기는 2^6(64), 2^9(512), 2^{12}(4KB), 2^{15}(32KB), 2^{18}(256KB) 순으로 증가한다. 반면, 각 그룹의 리스트 개수는 16, 8, 4, 2, 1 순으로 감소한다. 그림 2는 이 경우의 세 개의 연속된 segregated 그룹을 보인 것이다. 괄호 안은 리스트 수이다.

Half 리스트는 segregated 리스트와 마찬가지로 리스트 영역에 속하는 비슷한 크기의 블록들을 저장하고, 연속된 리스트들은 연속된 영역을 가진다. 그러나 연속된 리스트들간의 영역 크기 및 영역 증가 양은 고정된 상수가 아니라, 2의 거듭제곱 크기 순으로 증가한다 [그림 1. (c) 참조]. 이러한 half 리스트들만 모아 놓은 리스트 그룹을 *half* 리스트 그룹(또는 간단히 *half* 그룹)이라 하고, 이를 L_{HL}로 표현한다. Half 그룹은 주로 quick 그룹 또는 segregated 그룹 영역 보다 더 큰 크기의 블록을 관리하기 위해 사용되며, half-fit 정책을 지원한다. 따라서 이 그룹은 메모리 할당 정책에 따라 quick 그룹 또는 segregated 그룹과 연계되어 관리되며, 경우에 따라 단독으로 사용될 수도 있다.

표 2는 QHSF가 관리하는 리스트 그룹의 종류와 주요 특성들을 보인 것이다. 표에서 $low_{k,i}$는 각 리스트 영역의 최소값이고, $low_{k,1}$는 각 그룹 영역의 최소값이며, N_k는 각 그룹의 리스트 수이다. *Min?L / Max?L*은 각 그룹의 최소/최대값이며, 특히 segregated 그룹인 경우 전체 segregated 그룹의 최소/최대값을 의미한다. *Max?L*은 메모리 할당 정책과 상관없이 주어진 공식에 의해 결정되지만, *Min?L* 은 메모리 할당 정책에 따라 달라진다. 메모리 할당 정책별 생성되는 그룹과 그룹간의 관계는 달라진다.

표 3은 메모리 할당 정책별 생성되는 리스트 그룹과 그들의 연계관계를 보여 주고 있다. 기본적으로 작은 크기의 블록은 quick그룹으로 관리하고, 중간 크기는 segregated 그룹으로, 큰 크기는 half 그룹으로 관리하고 있다. 이는 실시간 응용 프로그램의 다양한 메모리 요구 분포에 급격한 성능 변화 없이 유연한 메모리 관리 효율성을 제공할 수 있기 때문이다. QSHF의 모든 리스트들간의 영역은 서로 연속되어 있으며, 그룹들간의 그룹 영역 또한 서로



(a) 영역크기 64(16개) (b) 영역크기 512(8개) (c) 영역크기 4096(4개)

그림 2. 영역 크기가 서로 다른 세 개의 연속된 segregated 리스트 그룹 (단위: bytes)

표 2. QHSF가 관리하는 프리 리스트 그룹의 종류와 특성

리스트 종류 / 리스트 특성	Quick 그룹, L_{QL} $2 \le i \le N_{QL}$	Segregated 그룹, L_k $1 \le k \le n,\ 2 \le i \le N_k$	Half 그룹, L_{HL} $2 \le i \le NHL$
리스트 영역 크기	$inc_{QL} = 2^w$: 고정	$inc_k = 2^{w(k+1)}$: 고정	$inc_{HL,i} = 2^{base+(i-1)}$: 가변
리스트 내의 블록 크기	동 일	가 변	가 변
리스트 영역 최소값	$low_{QL,1} = MinQL$ $low_{QL,i} = MinQL+(i-1) \times 2^w$	$low_{1,1} = MinSL$ $low_{k,1} = low_{k-1,1}+N_{k-1} \times inc_{k-1}$ $low_{k,i} = low_{k,1}+(i-1) \times inc_k$	$low_{HL,1} = MinHL$ $low_{HL,i} = 2^{base+(i-1)}$
그룹 영역 최소값	$MinQL$	$MinSL$	$MinHL$
그룹 영역 최대값	$MaxQL = MinQL+N_{QL} \times 2^w-1$	$MaxSL = low_{n,1}+N_n \times inc_{n-1}$	$MaxHL = 2^{word\_bits}-1$
비 고	2^w: 워드 크기	$n = \lfloor log_2 N_{QL} \rfloor$: 그룹 수	$base = \lfloor log_2 MinHL \rfloor$ $word\_bits$: 워드 비트 수

표 3. 메모리 할당 정책별 생성 그룹 및 그룹간 연계관계

할당 정책	생성 그룹	리스트 개수	그룹 영역 최소값, *Min?L*	그룹 영역 최대값, *Max?L*
DSA_QF	L_{QL}	N_{QL}	0	$MaxQL$
DSA_HF	L_{HL}	$word\_bits$	0	$2^{word\_bits}-1$
DSA_QSF $1 \le k \le n$	L_{QL}	N_{QL}	0	$MaxQL$
	L_k	2^{n-k}	$MaxQL+1$	$MaxSL$
DSA_QHF	L_{QL}	N_{QL}	0	$MaxQL$
	L_{HL}	$word\_bits-base$	$MaxQL+1$	$2^{word\_bits}-1$
DSA_QSHF $1 \le k \le n$	L_{QL}	N_{QL}	0	$MaxQL$
	L_k	2^{n-k}	$MaxQL+1$	$MaxSL$
	L_{HL}	$word\_bits-base$	$MaxSL+1$	$2^{word\_bits}-1$

연속되어 있다. 따라서 메모리 할당 정책별로 각 그룹의 최소값은 연계된 앞 그룹의 (최대값+1)을 가진다.

DSA_QSHF 정책의 경우, quick 그룹의 리스트 수 N_{QL}이 사용자에 의해 주어지면, QSHF는 segregated 그룹의 수와 각 그룹의 리스트 수를 결정한다. 또한 *MaxQL*, *MinSL*, *MaxSL*, *MinHL* 등을 순서적으로 결정하고, 최종적으로 half 그룹의 리스트 수 N_{HL}를 결정한다. 다른 정책들도 이와 유사하다.

Ⅳ. 구 현

본 절에서는 QSHF의 구현에 대해 논의한다. 먼저 다양한 종류의 리스트를 관리하는 방안과 특정 리스트를 찾는 방법에 대해 기술하고, 구현 측면의 메모리 할당 및 반환, 블록의 분할 및 합병에 대해서 논의한다. 또한 상대적으로 많은 리스트를 관리하면서도, 요구된 메모리 크기에 가장 적절한 리스트를 예측 가능한 시간 내에 찾고, 찾은 리스트가 비어 있을 경우 다음 이용 가능한 리스트를 고정 시간 내에 찾는 방안에 대해 논의한다. 마지막으로 QSHF의 WCET을 분석한다.

4.1 프리 리스트의 관리

QHSF는 모든 프리 리스트들을 하나의 긴 색인용 배열을 통하여 관리한다. 이 배열의 각 요소(element)는 상응하는 프리 리스트의 첫번째 블록에 대한 주소 값을 가지고 있다. 따라서 원하는 리스트를 참조하기 위해서는 배열의 상응하는 색인 값을 알아야 한다. 배열의 첫번째 요소(또는 제일 첫 프리 리스트)의 색인은 0인 것으로 가정한다.

색인용 배열에 QHSF의 리스트들을 배치하는 방식은 각 정책별로 표 3에 보인 그룹 순서대로 리스

트들을 배치하고, 각 그룹의 리스트들은 리스트 $l_{k,i}$의 첨자 i의 순으로 배치한다. 이 경우 n개의 그룹 $L_1, L_2, \cdots, L_{n-1}, L_n$이 임의의 정책에 대해 생성된 그룹들이라 가정하고, 각 그룹의 특정 리스트 $l_{k,i}$의 배열 색인을 $idx_{k,i}$라고 할 때, 다음의 관계가 성립한다.

$$idx_{k,i} = idx_{k,1} + i - 1, \text{ where } 1 \le k \le n,\ 2 \le i \le N_k$$

$$idx_{k,1} = \sum_{j=1}^{k-1} N_j, \text{ where } 2 \le k \le n,$$

$$idx_{1,1} = 0$$

반환한 메모리 블록이나, 합병한 결과 생성된 새로운 블록, 또는 분할하고 남은 블록들은 다시 프리 리스트에 삽입되어야 하며, 이를 위해선 그 블록을 포함하는 해당 리스트의 색인을 알아야 한다. 만약 s를 삽입하고자 하는 블록의 크기라고 가정할 때, 이를 포함하는 영역을 가진 리스트의 색인 $idx_{k,i}$는 다음의 색인 찾기 전략에 의해 구할 수 있다.

1) 먼저 s를 포함하는 그룹 L_k를 찾는다. 이를 위해선 s를 각 그룹의 첫번째 리스트 영역의 최소값인 $low_{k,1}$과 비교해 본다. 최악의 경우 $(n-1)$번 비교해야 한다.
2) 해당 그룹 내에서 s를 포함하는 리스트 $l_{k,i}$를 찾는다. 즉, 리스트의 첨자 i를 구한다. 그룹 L_k 내의 모든 리스트는 연속된 영역을 가지므로, 이는 다음과 같이 구할 수 있다.
   2.1) 그룹 L_k가 quick 또는 segregated 리스트 그룹인 경우, 모든 리스트가 동일한 영역 크기(inc_k) 및 영역 증가량을 가지므로 리스트의 첨자 i는 다음과 같다.
   $$i = (s - low_{k,1}) / inc_k + 1$$
   2.1) 그룹 L_k가 half 리스트 그룹(L_{HL})인 경우, 각 리스트의 영역 크기는 2의 거듭제곱 크기로 증가하므로 리스트의 첨자 i는 다음과 같다.
   $$i = \lfloor \log_2 s \rfloor - base + 1$$ [표 2 참조]
3) 리스트의 첨자 i를 이용하여 색인 $idx_{k,i}$를 구한다. 즉,
   $$idx_{k,i} = idx_{k,1} + i - 1$$

이를 위해 메모리 초기화시 설정된 메모리 할당 정책에 따라 각 그룹의 첫번째 리스트의 색인 $idx_{k,1}$과 이 리스트의 최소 영역 값 $low_{k,1}$을 계산하여 그룹별로 저장해 두어야 한다. Quick 리스트의 색인 값은 log 계산을 통해 구해야 하는데, 이를 위한 알고리즘을 비트 탐색 전략이라 하며, 이는 4.3절에서 논의한다.

4.2 프리 블록의 할당, 반환, 분할, 합병 전략

프리 리스트 $l_{k,i}$는 리스트 영역 $[low_{k,i}, low_{k,i+1})$ 범위 내의 다양한 크기의 프리 블록을 가진다. 따라서 이 영역의 메모리 할당 요구가 있을 경우, 리스트에서 적절한 블록을 선택해야 하며, 이를 위해선 *first-fit, best-fit, next-fit* 등과 같이 리스트를 순차적으로 탐색해야 한다. 그러나 QSHF는 이러한 순차적 탐색을 지양하기 위해 *round-up* 선택 전략을 사용한다. 즉, 리스트 $l_{k,i}$는 $[low_{k,i}, low_{k,i+1})$ 범위의 메모리 요구를 서비스하는 것이 아니라, $(low_{k,i-1}, low_{k,i}]$ 범위의 메모리 요구를 서비스한다. 이 경우 리스트 $l_{k,i}$가 가진 어떠한 블록도 이 요구를 만족할 수 있다. 따라서 사용자가 요구한 메모리 크기를 s라 하고, *get_index*()를 색인 찾기 함수라고 할 때, 메모리 할당시의 해당 리스트 색인 $idx_{k,i}$는 다음과 같이 구한다.

$$idx_{k,i} = get\_index(s - 1) + 1$$

이렇게 찾은 리스트가 빈(empty) 리스트가 아니면 그 리스트의 첫번째 블록을 할당한다. 그러나 이 블록의 크기가 요구된 크기 보다 클 경우, 내부 조각화를 방지하기 위해 분할작업을 수행한다. 이 경우 남은 블록은 다시 적절한 리스트에 삽입한다.

만약 찾은 리스트가 빈 리스트 일 경우, 이 리스트 보다 큰 영역을 가진 non-empty 리스트들 중 가장 작은 크기의 블록을 가진 리스트를 찾아야 한다. 이를 위해선 $(idx_{k,i}+1)$부터 순차적으로 리스트가 빈 리스트인지 아닌지 비교해야 한다. 그러나 QSHF는 순차적 탐색을 지양하기 위해 *non-empty* 리스트 탐색 전략을 사용한다. 이 전략에 대해서는 4.4절에서 논의한다.

QSHF는 실시간 특성을 고려하여 블록 반환시 반환된 블록과 인접한 프리 블록들은 즉시 합병 방식을 사용하여 합병한다. 즉시 합병을 위해 인접한 블록을 고정된 시간 내에 찾고, 또한 그 블록이 프리 블록인지 아닌지 판단하기 위해, QSHF는 참고문헌 [18]에 제시된 개선된 경계 태그(boundary tags) 기술을 사용하였다. 또한 합병될 블록을 기존의 리스트에서 바로 제거할 수 있게 이중 연결(double linked) 구조를 사용한다.

4.3 비트 탐색(bit search) 전략

앞서 언급한 색인 찾기 전략에서 블록의 크기 s를 포함하는 리스트 그룹이 half 그룹일 경우, 해당

리스트의 첨자 i를 구하기 위해선 log 계산을 해야 한다. 사실 $\lfloor log_2 s \rfloor$는 블록 크기 s를 이진수로 표현했을 때, 제일 왼쪽 1의 비트 위치(LSB의 위치를 0이라 가정)이다.

만약 시스템 CPU가 특정 레지스터의 MSB(most significant bit)에서 LSB(least significant bit) 방향으로 제일 처음 1의 비트 위치를 찾아 주는 비트 탐색(bit search) 명령어를 지원한다면, 한 명령어 주기 내에 $\lfloor log_2 s \rfloor$를 구할 수 있다. 그러나 모든 CPU가 비트 탐색 명령어를 지원하는 것이 아니고, 또한 이 명령어를 사용할 경우 H/W 의존적이 될 수 있으므로, QSHF는 S/W적인 방법도 제공한다. 이를 비트 탐색 전략이라 한다.

이 전략은 색인 값을 이진수로 표현했을 때, 제일 왼쪽 1의 비트 위치를 저장하고 있는 비트 위치 테이블을 사용한다. 표 4는 8 비트 색인용 비트 위치 테이블이다. 만약 블록의 크기를10이라 가정하고, 이를 색인으로 사용하여 이 테이블을 참조하면, $\lfloor log_2 10 \rfloor$은 3이라는 것을 쉽게 찾을 수 있다. 그러나 이 방법은 블록 크기를 표현하는 비트 수가 많아 질 경우, 비트 위치 테이블을 위한 과도한 메모리 부하(overhead)가 발생한다.

따라서 QSHF에서는 8 비트 색인용 비트 위치 테이블을 사용하고, 블록의 크기를 표현하는 비트들(비트 벡터)을 한 바이트(8bits)씩 순서적으로 처리하는 방법을 사용한다. 비트 벡터의 매 바이트별로 그 값이 0인지 아닌지 비교하고, 0인 경우 기준 위치(초기값 0) 값에 8을 더하고, 비트 벡터의 다음 바이트를 비교한다. 만약 비교하는 바이트 값이 0이 아닌 경우, 이를 색인으로 사용하여 비트 위치 테이블에서 해당 비트 위치를 구한다. 이 값을 기준 위치 값에 더해서 $\lfloor log_2 s \rfloor$를 구한다. 한 워드의 크기가 32 비트(4 bytes)인 경우 최악의 경우(블록의 크기가 256보다 작은 경우) 3번의 비교가 필요하다.

표 4. 8비트 색인용 비트 위치 테이블

색인(이진수)	비트위치	색인(이진수)	비트위치
0 (00000000)	N/A	…	
1 (00000001)	0	84(01010100)	6
2 (00000010)	1	…	
3 (00000011)	1	248 (11111000)	7
4 (00000100)	2	249 (11111001)	7
5 (00000101)	2	250 (11111010)	7
6 (00000110)	2	251 (11111011)	7
7 (00000111)	2	252 (11111100)	7
…	…	253 (11111101)	7
10 (00001010)	3	254 (11111110)	7
…	…	255 (11111111)	7

4.4 Non-empty 리스트 탐색 전략

블록 할당시 요청된 크기에 가장 적합한 리스트를 찾았으나, 이 리스트가 빈 리스트일 경우 이 보다 큰 블록을 가진 non-empty 리스트들 중 가장 작은 크기의 블록을 가진 리스트를 찾아야 한다. 이를 위한 방법으로 리스트가 빈 리스트인지 아닌지 오름차순으로 순차적으로 검색해 볼 수 있다. 그러나 이 방법은 리스트의 개수가 많아 질수록 수행시간이 리스트 수에 비례하여 증가될 뿐 아니라, 수행시간 편차(execution time jitter) 역시 커지는 문제가 있다.

따라서 QSHF는 다단계 비트 벡터(layered bit vector) 구조와 맵핑 테이블을 사용함으로써, 고정된 WCET을 보장하는 non-empty 리스트 탐색 전략을 사용한다. 이 전략은 각 리스트별로 이에 상응하는 단계별 비트를 두고, 해당 리스트가 빈 리스트인지 아닌지를 표시한다. 이들 비트들을 비트 벡터(또는 벡트)라 한다. 이 전략을 그림 3과 표 5를 이용하여 설명하면 다음과 같다. 그림 3은 64개의 리스트를 가지는 시스템에서 사용되는 2 단계 비트 벡터 구조이며, 표 5는 이때 사용되는 맵핑 테이블이다. 이 테이블은 색인 값을 이진수로 표현했을 때, 가장 오른쪽 1의 비트 위치(LSB의 위치를 0이라 가정)를 저장하고 있다.

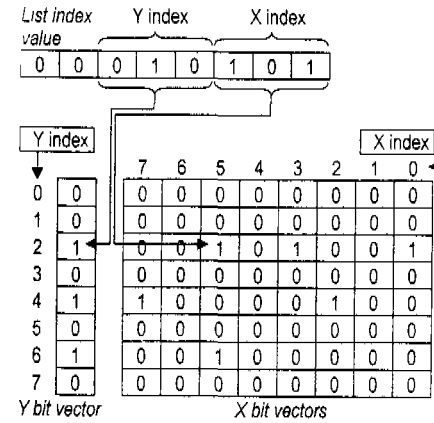


그림 3. 2 단계(two layered) 비트 벡터 구조

그림 3에서는 이진수로 표현된 임의의 리스트의 색인 값(21)과 8 비트로 구성된 하나의 Y 비트 벡터, 그리고 이 벡터의 각 비트별로 상응하는 8개의 X 비트 벡터(8 bits)들이 존재한다. 리스트의 색인

표 5. 가장 오른쪽 1의 비트 위치를 찾기 위한 맵핑 테이블 (8 bits 색인용)

색인	비트위치	색인	비트위치
0 (00000000)	0	…	…
1 (00000001)	0	84(01010100)	2
2 (00000010)	1	…	…
3 (00000011)	0	248 (11111000)	3
4 (00000100)	2	249 (11111001)	0
5 (00000101)	0	250 (11111010)	1
6 (00000110)	1	251 (11111011)	0
7 (00000111)	0	252 (11111100)	2
.	…	253 (11111101)	0
41(00101001)	0	254 (11111110)	1
.	…	255 (11111111)	0

값은 0에서 63까지 가질 수 있으므로 6 비트만 사용되며 (총 리스트가 64개이므로), 이중 상위 3 비트를 *Y* 벡터 색인이라 하고, 하위 3비트는 *X* 벡터 색인이라 한다. 그림에서 임의의 리스트의 *Y* 색인 2는 *Y* 벡터의 3번째 비트에 상응하고, *X* 색인 5는 위로부터 3번째 *X* 벡터의 6번째 비트와 상응한다. 각 리스트가 빈 리스트에서 non-empty 리스트로 바뀔 때, *Y* 벡터와 *X* 벡터의 상응하는 비트를 1로 설정하고, non-empty에서 empty로 바뀔 때는 *X* 비트 벡터의 상응하는 비트만 다시 0으로 설정한다. *Y* 벡터의 비트는 이 비트의 상응하는 *X* 벡터 값이 0일 경우(즉, 상응하는 *X* 벡터의 모든 비트가 0일 경우)에만 0으로 설정한다. 그림의 2단계 *X*, *Y* 벡터는 현재 16, 19, 21, 34, 39, 53을 색인으로 하는 리스트들이 non-empty 리스트로 설정되어 있음을 보여주고 있다.

문제는 이 중에서 가장 작은 크기의 블록을 가지는 non-empty 리스트 16을 어떻게 찾아 내느냐이다. 우선 현재의 *Y* 벡터 값 84(01010100)를 색인으로 사용하여 표 5의 맵핑 테이블에서 해당 맵핑 값 2를 찾는다. 이는 곧 찾고자 하는 리스트의 *Y* 색인이 2라는 것을 의미한다. 이를 이용하여, *Y* 벡터의 색인 2에 상응하는 *X* 벡터의 값 41(00101001)을 색인으로 사용하여 맵핑 테이블에서 해당 맵핑 값 0을 얻는다. 이는 곧 찾고자 하는 리스트의 *X* 색인이 0이라는 것을 의미한다. 따라서 *Y* 색인 2(010)와 *X* 색인 0(000)을 조합하여 가장 작은 크기의 블록을 가지는 non-empty 리스트의 색인 16 (00.010.000)을 얻을 수 있다.

이상에서 기술한 예제는 non-empty 리스트들 중에서 가장 작은 크기의 블록을 가지는 리스트를 찾는 방법이다. 그러나 QSHF에서는 블록 할당 시 해당 리스트가 빈 리스트일 경우, 이 보다 큰 블록을 가진 리스트들 중 가장 작은 크기의 블록을 가진 non-empty 리스트를 찾는 경우이다. 이를 위해서는 상기 설명한 방법을 응용하여 *X*, *Y* 벡터를 *X*, *Y* 색인의 상응하는 매스크(mask)로 매스킹(masking)한 후, 맵핑 테이블을 참조하면 된다.

위의 예제는 64개의 리스트를 위한 2단계 벡터 구조를 사용하였는데, 사실 리스트의 수가 8의 거듭제곱 순으로 증가할 때마다 벡터 단계가 한 단계 증가한다. 그러나 리스트 수의 증가에 따른 색인용 배열의 추가 메모리 부하를 감안하여, 현실적으로 최대 512개의 리스트를 지원하는 3단계 비트 벡터 구조가 가장 타당하다. Non-empty 리스트 탐색 전략은 별도의 비트 벡터 및 맵핑 테이블을 필요로 하지만, 고정된 WCET을 보장하는 장점을 제공한다.

4.5 최악의 경우 실행 시간(WCET) 분석

일반적으로 실시간 시스템에 DSA 알고리즘을 적용하기 위해서는 알고리즘의 시간 복잡도가 $O(1)$이어야 하고, 이를 구현했을 때 실행(응답) 시간의 변화(jitter)가 적고, WCET을 쉽게 측정할 수 있어야 한다. 그러나 이진 버디 시스템의 경우 비록 시간 복잡도는 $O(1)$일지라도 WCET 계산이 쉽지 않다. 이유는 블록 분할 및 합병시 매번 최악의 경우 워드의 비트 수 만큼 분할 및 합병을 수행해야 하며, 이 과정에서 각 리스트에 블록의 삽입/삭제를 위한 메모리 전반에 걸친 광범위한 접근(access)이 이루어지기 때문이다. 이는 곧 빈번한 캐시(cache) 및 TLB misses를 유발한다[1]. 이러한 측면에서 QSHF는 합병할 블록이 기껏해야 3개이고, 분할도 한번만 수행한다. 그리고 리스트 색인용 배열과 다단계 비트 벡터 및 이와 관련한 맵핑 테이블만 참조하므로, 이진 버디 시스템과는 달리 지역성(locality)이 유지된다.

QSHF 알고리즘은 선택된 메모리 할당 정책과는 상관없이 동일한 코드를 이용하여 블록의 분할, 합병, 삽입, 삭제, non-empty 리스트 찾기 등을 수행한다. 다만, 색인 찾기 전략과 비트 탐색 전략만이 메모리 할당 정책과 밀접한 연관을 가진다.

메모리 할당시의 최악의 경우는, 색인 찾기 전략(half 리스트면 비트 탐색 전략 수행)을 이용하여 적절한 리스트를 찾고, 해당 리스트가 빈 리스트일 경우 non-empty 리스트 탐색 전략을 수행하며, 찾

은 리스트에서 첫 블록을 빼낸 후 이를 분할하고, 남은 블록은 다시 적절한 리스트에 삽입하는 경우이다. 반환시의 최악의 경우는, 반환된 블록의 전후 블록이 모두 프리 블록인 경우 이들을 해당 리스트에서 삭제하고, 합병하여 다시 적절한 리스트에 삽입하는 경우이다. 두 경우 모두 블록 삽입을 위한 적절한 리스트는 색인 찾기 전략(역시 half 리스트면 비트 탐색 전략 수행)을 이용한다.

이 과정에서 WCET에 영향을 미치는 부분은 색인 찾기 전략 뿐이며, 나머지는 항상 고정된 수행시간을 보장한다. 색인 찾기 전략은 각 리스트 그룹의 영역 비교를 위해 최대 (그룹의 개수 1)번의 비교가 필요하다. 만약 리스트 그룹이 half 그룹일 경우 비트 탐색 전략을 수행하는 과정에서 (워드 크기 1)번의 비교가 더 필요하다. 워드 크기는 고정된 상수이고, 최대 리스트 그룹의 개수도 사전에 결정되므로, 알고리즘의 시간 복잡도는 궁극적으로 $O(1)$이고, 각 정책별 WCET도 쉽게 구할 수 있다. 또한 리스트 관리를 위한 추가 메모리 부하를 고려해 볼 때, 최악의 경우 리스트 그룹 개수는 10개 이내이다. 따라서 이러한 최악의 경우를 가정한다면, 메모리 할당 정책과는 무관하게 알고리즘의 시간 복잡도는 $O(1)$이고, 고정된 WCET을 구할 수 있다.

V. 시뮬레이션

QSHF의 다양한 메모리 할당 정책별 메모리 사용 효율성을 비교하기 위해, 본 연구에서는 전체 메모리 용량을 제한한 채, 인위적(synthetic)으로 생성된 메모리 할당(allocation) 및 반환(deallocation) 패턴을 사용하여[1,11,27], 최장 수행 시간 및 조각화(fragmentation) 율, 그리고 메모리 할당 실패 율 등을 측정하는 몇 가지 실험을 실시하였다. 본 절에서는 실험에 사용된 시뮬레이션 방법을 기술하고, 이 방법에 의해 실시된 실험 결과를 제시한 후, 이를 분석한다.

5.1 시뮬레이션 방법

본 연구에서 QSHF의 정책별 성능을 비교하기 위해 사용하는 평가 기준은 메모리 할당 및 반환에 소요되는 최장 수행(응답) 시간, 내부 조각화 율, 외부 조각화 율, 총 조각화 율, 할당 실패 율 등이다. 최장 수행 시간은 실제 구현된 시스템에서의 정책별 최악의 경우 실행 시간이다. 이는 시간 복잡도 $O(1)$을 만족하는 알고리즘이 실제 구현된 시스템에서 일정하게 WCET을 유지하는가를 판단하기 위함이다.

조각화 율(fragmentation ratio)은 DSA 알고리즘이 적용되는 응용 프로그램, 라이브러리, 운영체제와의 상관 관계에 따라 다양하게 정의될 수 있다[9]. 본 실험에서는 실시간 시스템의 특성에 가장 적합한 참고 문헌 [7]의 방법을 따르기로 했다. 따라서 내부 조각화 율(internal fragmentation ratio)은 $IF = A / R$로 정의한다. 여기서 A는 DSA 알고리즘이 할당한 총 메모리 양이고, R은 사용자가 요청한 총 메모리 양이다. 또한 외부 조각화 율(external fragmentation ratio)은 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 율(total fragmentation ratio)은 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.

할당 실패 율(allocation failure ratio)은 $AF = FN / RN$로 정의한다. 여기서 RN은 사용자가 요청한 총 할당 회수이고, FN은 할당에 실패한 총 회수이다.

기존의 연구에 따르면 메모리 할당 시뮬레이션은 요구된 할당 크기(sizes of allocation requests), 할당된 블록의 시스템 내의 존속 기간(lifetimes), 그리고 할당 요구가 들어오는 시간 간격(arrival intervals) 등 세가지 분포(distributions)에 상당한 영향을 받는다[27]. 실제 응용 프로그램의 메모리 요구 패턴과 가능한 유사한 패턴을 반영하는 실험 결과를 도출하기 위해서, 이러한 분포를 신중하게 선택하는 것이 중요하다.

본 연구에서는 참고 문헌 [1]과 같이, 할당 크기

분포를 위해 지수(exponential) 분포와 균등(uniform) 분포 등 두 가지 확률 분포를 사용한다. 크기 분포의 평균은 8, 10, 12, 14, 16, 32, 64, 128, 256, 512, 1024, 2048 워드(words)로 다양하게 변화 시켰다. 그러나 이때 사용되는 총 메모리 용량은 항상 32K 워드로 고정 시켰다. 할당된 메모리 블록의 시스템 내의 존속 기간 분포는 균등 분포를 따르며, 5에서 15 사이의 시간 단위(time units) 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 용량(32K)은 항상 고정되어 있으므로, 평균 할당 크기가 결정되면 시스템 내에 존속하는 할당된 메모리 블록의 평균 개수 λ/μ값(총 메모리 용량 / 평균 할당 크기)을 결정할 수 있다. 따라서 평균 존속 기간은 $1/\mu$(10)로 고정되었으므로, 평균 요구 도착 시간 간격 $1/\lambda$를 결정할 수 있다.

5.2 시뮬레이션 결과

앞서 기술한 큐잉 모델에 따라 각 평균 크기 분포별 메모리 할당/반환 패턴을 생성한 후, QSHF의 주요 정책(**DSA_HF**, **DSA_QHF**, **DSA_QSHF**) 및 이진 버디 시스템의 최장 실행 시간(WCET), 내부 조각화 율, 외부 조각화 율, 총 조각화 율, 할당 실패 율 등을 측정하였다. 각 알고리즘이 사용하는 메모리 블록의 헤드는 모두 동일한 구조를 가진다. 블록 헤드는 블록을 관리하기 위해 필요한 최소한의 데이터 구조로서, 이중 연결 포인터와 두 개의 경계 태그(블록 크기 값)를 가지며, 총 16 bytes이다. 그러나 본 연구에서는 개선된 경계 태그 기술[18]을 사용하므로, 할당된 블록의 헤드는 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. 본 실험에서 하나의 워드는 8 bytes이며, 주소 조정과 블록 헤드를 위해 별도로 더 할당된 메모리 공간은 내부 조각화 율 계산에 모두 포함시켰다.

각 실험에서 QSHF의 **DSA_HF**를 제외한 각 정책이 사용하는 quick 그룹의 리스트 수(N_{QL})는 64개이며, *MaxQL*은 511 bytes, *MaxSL*은 2730.5 KB이다. **DSA_HF** 정책과 이진 버디 시스템이 관리하는 리스트 개수는 각각 32개이다. 표 6은 이 경우 (N_{QL} = 64)의 각 정책별 리스트 관리를 위한 메모리 부하를 보인 것이다. 여기서 각 색인용 배열의 요소 크기 및 그룹별 관리 정보는 각각 8 bytes이고, half 그룹인 경우 비트 탐색 전략을 위한 비트 위치 테이블(256 bytes)이 포함되었다.

그림 4와 5는 총 메모리 용량별 QSHF의 세 정책과 이진 버디 시스템의 메모리 할당 및 반환에 소요된 최장 실행 시간을 보인 것이다. 실험은 UltraSPARC 2(200MHz) 프로세서를 탑재한 SUN UltraSPARC 워크스테이션에서 실행되었으며, 실험 결과는 1,000,000을 수행하여 측정한 최장 실행 시간들의 평균 값을 보인 것이다. 실험에서 테스트 프로그램의 코드 및 데이터 영역을 *mlock()* 함수를 통해 locking 함으로써, 가상 기억 장치의 영향력을 배제시켰다. 그러나 캐시(cache)가 미치는 영향은 실험 결과에 반영되지 않았으므로, 실제의 WCET은 실험 결과치 보다 더 커질 것으로 예상된다. 따라서 이 실험은 메모리 할당/반환의 진정한 WCET을 측정하기 보다는 비교하고자 하는 각 정책이 메모리 크기에 상관없이 동일한 최장 실행 시간을 보이는 가를 측정하기 위함이다.

두 그림에서 QSHF의 각 정책은 총 이용 가능한 메모리의 용량에 상관없이 동일한 최장 실행 시간을 보인 반면, 이진 버디 시스템은 메모리 용량이 증가함에 따라 최장 실행 시간도 함께 증가함을 볼

표 6. 메모리 할당 정책별 리스트 관리를 위한 메모리 부하(overhead), N_{QL} = 64

할당 정책	DSA_QF	DSA_HF	DSA_QSF	DSA_QHF	DSA_QSHF
리스트 그룹	quick	half	quick + segregated	quick + half	quick + segregated + half
리스트 그룹 수	1	1	7 = 1 + 6	2	8 = 1 + 6 + 1
리스트 수	64	32	128 = 64 + 64	87 = 64 + 23	139 = 64 + 64 + 11
리스트 관리 부하 (bytes)	520	512	1080	968	1432

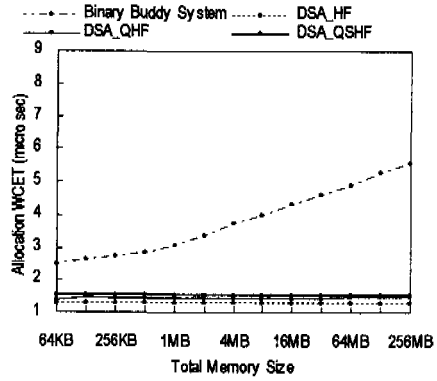


그림 4. 메모리 할당에 소요된 최장 실행 시간

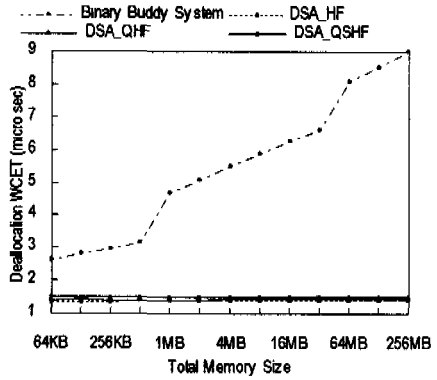


그림 5. 메모리 반환에 소요된 최장 실행 시간

수 있다. 이유는 이진 버디 시스템에서 블록 분할 및 합병 시 매번 최악의 경우 32번(워드의 비트 수)의 분할 및 합병을 수행해야 하며, 이 과정에서 각 리스트에 블록의 삽입/삭제를 위해 메모리 전반에 걸친 광범위한 접근(access)이 이루어지기 때문이다. 이는 또한 알고리즘의 시간 복잡도가 비록 $O(1)$일지라도 알고리즘이 실제 구현된 시스템에서 WCET은 일정하게 유지되지 않는다는 것을 보이고 있다. Iyengar는 참고 문헌 [7]에서 wighted 버디, double 버디의 경우 이진 버디 시스템보다 조각 율은 근소한 차이로 낮지만 응답 시간은 상당히 느리다는 실험 결과를 밝혔다. 또한 WCET의 변동이 심하므로 실시간 시스템에 적용하기 곤란한 문제점을 가지고 있다. 따라서 본 연구에서는 대표적인 이진 버디 시스템만을 고려하기로 한다.

표7은 각 정책별 메모리 할당 및 반환에 소요된 최장 실행 시간을 보인 것이다. 표에서 복합 정책이 단일 정책보다 상대적으로 높게 나타나는 것은 관리해야 할 리스트 그룹과 리스트 수가 증가하기 때문이다. 즉, 색인 찾기 전략에서 비교(그룹 최소값)하는 횟수가 증가하고, half 그룹인 경우 비트 탐색 전략을 수행해야 하기 때문이다. 그러나 전체적인 각 정책별 최장 실행 시간은 큰 차이가 없다.

표 7. 메모리 할당 정책별 최장 실행 시간

할당 정책	DSA_QF	DSA_HF	DSA_QSF	DSA_QHF	DSA_QSHF
할당(μ sec)	1.20	1.29	1.43	1.46	1.53
반환(μ sec)	1.31	1.39	1.49	1.48	1.51

그림 6은 할당 크기 분포가 지수 분포를 가질 때, 그림 7은 균등 분포를 가질 때, 평균 할당 크기별 할당 실패 율을 보인 것이다. 두 그림 모두 할당 크기 분포에 상관없이 유사한 할당 실패 율을 보임을 확인할 수 있다. **DSA_HF** 정책(half-fit)과 이진 버디 시스템의 할당 실패 율은 참고 문헌 [1]의 실험 결과와 유사한 수치를 보인다. **DSA_QSHF** 정책은 전 구간에 걸쳐 가장 낮은 실패 율을 보인 반면, 이진 버디 시스템은 가장 높은 실패 율을 보인다. **DSA_HF** 정책은 평균 크기가 작을수록 이진 버디 시스템과 유사한 실패 율을 보인다. **DSA_QHF** 정책은 평균 크기가 작은 경우(32 이하) **DSA_HF**와 같고, 평균 크기가 큰(256 이상) 경우 **DSA_QSHF**와 같아 진다. 평균 크기가 계속해서 커진다면 **DSA_QSHF**의 실패 율은 궁극적으로 **DSA_HF**와 같아 질 것으로 예상된다. 본 실험에서 **DSA_HF**에 비해 **DSA_QHF**과 **DSA_QSHF**의 성능 향상을 비교 해 봄으로써, quick 리스트 그룹과 segregated 리스트 그룹의 기여도를 확인할 수 있다.

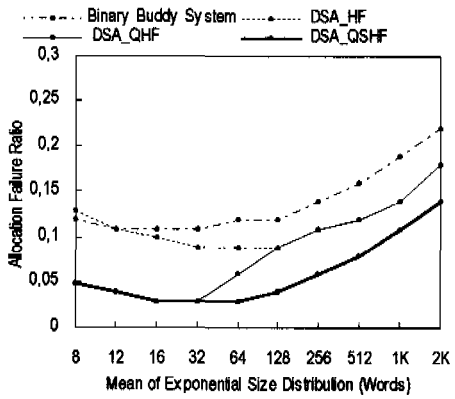


그림 6. 지수 분포 때의 할당 실패 율

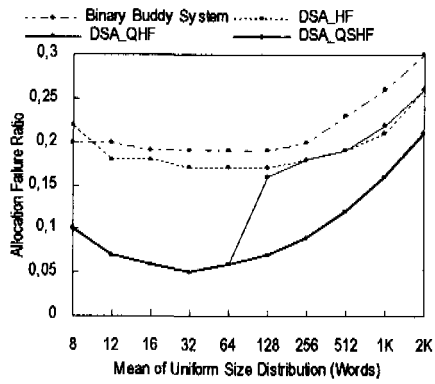


그림 7. 균등 분포 때의 할당 실패 율

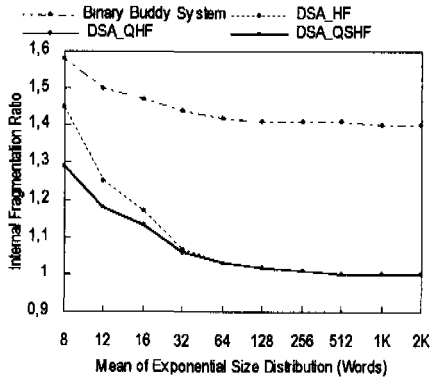


그림 8. 내부 조각화 율 (지수 분포)

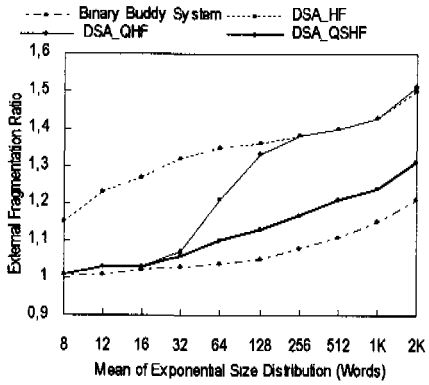


그림 9. 외부 조각화 율 (지수 분포)

그림 8, 9, 10은 각각 평균 할당 크기별 각 QSHF 정책의 내부 조각화 율, 외부 조각화 율, 총 조각화 율을 보인 것이다. 각 그림에서 할당 크기 분포는 지수 분포를 가진다. 이미 예측되는 바와 같이 이진 버디 시스템의 내부 조각화 율은 QSHF의 정책들에 비해 상대적으로 가장 높지만, 외부 조각화 율은 상대적으로 가장 낮다. **DSA_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 exact-fit(quick-fit) 전략을 사용하고, 중간 크기는 segregated-fit 전략을, 큰 크기는 half-fit 전략을 사용하는 **DSA_QSHF**의 특성으로 인한 결과이다. 이러한 현상은 총 조각화 율(그림 10)에서 뚜렷이 확인할 수 있다. 그림 10에서 **DSA_QSHF**는 전 구간에 걸쳐 이진 버디 시스템 및 **DSA_HF** 보다 상대적으로 낮은 조각화 율을 보인다. **DSA_QSHF**와 **DSA_QHF**가 작은 크기(32 이하)에서 **DSA_HF** 보다 낮은 조각화 율을 보이는 것은 quick-fit 전략의 효과이다. 또한 중간 크기(64 이상)에서부터 **DSA_QSHF**와 **DSA_QHF**의 조각화 율이 차이가 나는 것은 segregated-fit 전략의 효과이다.

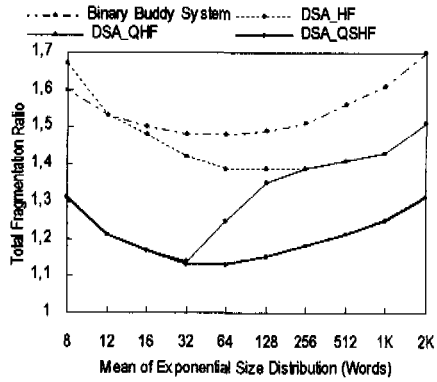


그림 10. 지수 분포 때의 총 조각화 율

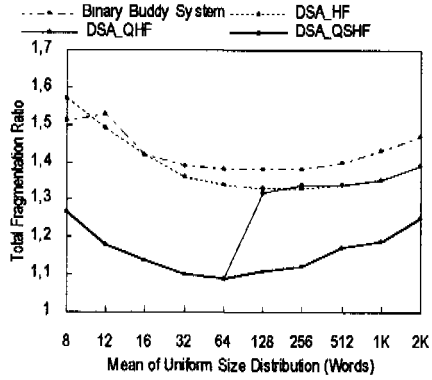


그림 11. 균등 분포 때의 총 조각화 율

할당 크기 분포가 균등 분포를 가지는 경우에도 유사한 패턴을 보임을 그림 11에서 확인할 수 있다.

그림 10과 그림 11의 총 조각화 율은 그림 6과 7의 할당 실패 율과 비슷한 양상을 보인다.

할당 실패 율과 조각화 율에서 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 부하(overhead)가 상대적으로 증가한 것으로 판단된다. 이러한 블록 헤드 구조체로 인한 부하는 참고 문헌 [9]에서 이미 지적된 사항이기도 하다.

다양한 응용 프로그램의 메모리 요구 분포가 거의 대부분 30-40 워드 범위 내라는 연구 결과 [6-12]를 고려해 볼 때, 실시간 응용 프로그램의 메모리 요구 분포도 이 범주에서 크게 벗어나지 않을 것으로 판단된다. 전반적으로 조각화 율 및 할당 실패 율에서 **DSA_QSHF**가 가장 뛰어난 메모리 관리 성능을 발휘하지만, 표 6과 7에서와 같이 상대적으로 증가된 최장 실행 시간 및 리스트 관리 부하가 따른다. 따라서 실시간 응용 프로그램의 메모리 요구 분포를 사전에 예측 가능한 경우에는 **DSA_QF**, **DSA_HF**, **DSA_QHF** 등의 선택도 고려 할 수 있다. 이러한 메모리 할당 정책의 선택은 표 1에서 제시된 바와 같이 해당 응용 프로그램의 시간 엄격성, 실행 시간의 가변성(jitter), 메모리 요구 분포의 예측 가능성 등을 고려하여, 가장 적절한 정책을 선택하여야 한다.

Ⅵ. 결 론

본 논문에서 다양한 메모리 할당 정책을 지원하면서도, 알고리즘 복잡도가 $O(1)$이고, 예측 가능한 실행 시간을 가진 실시간 동적 메모리 할당 알고리즘을 제안했다. 제안된 알고리즘은 할당 정책별 차이는 있지만, 기본적으로 작은 크기의 메모리 요구에는 워드 크기별 리스트를 관리하는 quick(exact)-fit 전략을, 중간 크기의 요구에는 적절한 크기 영역별 리스트를 관리하는 segregated-fit 전략을, 큰 크기의 요구에는 2의 거듭제곱 크기별 리스트를 관리하는 half-fit 전략을 사용한다. 따라서 사용자는 실시간 응용 프로그램의 특성에 가장 적합한 메모리 할당 정책을 선택할 수 있다.

또한 상대적으로 많은 프리 리스트를 관리하면서도, 요구된 메모리 크기에 가장 적절한 리스트를 예측 가능한 시간 내에 찾는 비트 탐색 전략과, 찾은 리스트가 비어 있을 경우 다음 이용 가능한 리스트를 고정 시간 내에 찾는 non-empty 리스트 탐색 전략도 함께 제시하였다. 따라서 다양한 메모리 할당 정책을 지원하고, 특성이 다른 여러 리스트를 관리하면서도, 쉽게 WCET을 예측할 수 있게 하였다.

제안된 알고리즘의 실용성을 확인하기 위해, 주요 정책들과 이진 버디 시스템과의 메모리 사용 효율성을 비교하였다. 실험 결과 모든 면에서 QSHF가 이진 버디 시스템보다 우수하다는 것을 확인하였다. 전반적으로 조각화 율 및 할당 실패 율에서 **DSA_ QSHF**가 뛰어난 메모리 관리 성능을 발휘하지만, 상대적으로 최장 실행 시간 및 리스트 관리 부하가 증가되었다. 따라서 모든 메모리 할당 정책들은 이러한 관점에서 서로 상반된 면을 가지고 있다. 이는 결국 설계자가 각 정책별 장단점을 고려해야 하고, 실시간 응용 프로그램의 특성에 맞는 정책을 선택해야 하는 부담감을 가지게 된다. 이를 위해서 향후 본 연구팀은 사용자가 원하는 메모리 할당 정책을 손쉽게 선택할 수 있는 구체적인 선택 기준을 마련할 예정이다.

본 연구에서 실시한 실험은 전체 메모리 용량을 제한한 채, 인위적(synthetic)으로 생성된 메모리 할당/반환 패턴을 사용하였다. 이러한 시뮬레이션 방법은 실제 사용하는 응용 프로그램에 따라 상당히 다른 양상을 나타낼 수 있다. 따라서 향후 실제 실시간 응용 프로그램의 메모리 할당/반환 패턴을 조사하고, 이를 기반으로 한 실험을 계속할 계획이다. 또한 segregated 그룹의 수와 영역 증가 양을 사용자가 임의로 조정하게 하여, 응용 프로그램 특성에 맞게 보다 세분화 시킬 수 있는 방안을 연구 중에 있다.

참 고 문 헌

[1] 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.

[2] 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.

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

[4] Doug Smith, "Ada Quality and Style: Guidelines for Professional Programmers,

Version 02.01.01," Technical Report SPC-91061-CMC, Software Productivity Consortium, Inc., 1992.

[5] David B. Stewart, R. A. Volpe, and Pradeep K. Khosla, "Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects," *IEEE Transactions on Software Engineering*, Vol. 23, No. 12, pp. 759-776, 1997.

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

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

[8] 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.

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

[10] Charles B. Weinstock, "Dynamic Storage Allocation Techniques," Ph.D. thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1976.

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

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

[13] Donald E. Knuth, *The Art of Computer Programming, Volume1: Fundamental Algorithms*, Addition-Wesley, Reading, Massachusetts, 1973.

[14] C. Bay, "A Comparison of Next-fit, First-fit, and Best-fit," *Communications of the ACM*, Vol. 20, No. 3, pp. 191-192, 1977.

[15] Kenneth C. Knowlton, "A Fast Storage Allocator," *Communications of the ACM*, Vol. 8, No. 10, pp. 623-625, 1965.

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

[17] J. S. Fenton and D. W. Payne, "Dynamic Storage Allocation of Arbitrary Sized Segments," *Proc. of IFIPS*, pp. 344-348, 1974.

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

[19] C. J. Stephon, "Fast Fits: New Method for Dynamic Storage Allocation," *Proc. of the 9th Symposium on Operating Systems Principles*, pp. 30-32, 1983. Also published as *Operating Systems Review*, Vol. 17, No. 5, 1983.

[20] R. P. Brent, "Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation," *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 3, pp. 388-403, 1989.

[21] G. Bozman, "The Software Lookaside Buffer Reduces Search Overhead with Linked Lists," *Communications of the ACM*, Vol. 27, No. 3, pp. 222-227, 1984.

[22] R. R. Oldehoeft and S. J. Allan, "Adaptive Exact-Fit Storage Management," *Communications of the ACM*, Vol. 28, No. 5, pp. 506-511, 1985.

[23] D. Grunwald and B. Zorn, "CustoMalloc: Efficient Synthesized Memory Allocators," *Software Practice and Experience*, Vol. 23, No. 8, pp. 851-869, 1993.

[24] D. S. Hirschberg, "A Class of Dynamic Memory Allocation Algorithms," *Communications of the ACM*, Vol. 16, No. 10, pp. 615-618, 1973.

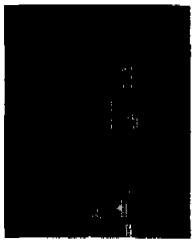
[25] K. K. Shen and J. L. Peterson, "A Weighted Buddy Method for Dynamic Storage Allocation," *Communications of the ACM*, Vol. 17, No. 10, pp. 558-562, 1974.

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

[27] 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.

정 성 무(Sung-Moo Jung)



1981년 : 충남대학교 전자공학과 졸업(학사)
1988년 : 한양대학교 전자공학과 졸업(석사)
2000년 : 아주대학교 컴퓨터공학과 박사과정 (수료)
1981년~1989년 : 잠실중, 경기공고 교사
1989년~1997년 : 한국교육개발원 부연구위원
1997년~현재 : 한국교육학술정보원 연구위원
<주관심 분야> Authoring Systems, 시스템 프로그램, 교육정보화 등
e-mail : smjung@keris.or.kr

유 해 영(Hae-Young Yoo)

1979년 : 단국대학교 수학과 졸업(학사)
1981년 : 단국대학교 수학과 졸업(석사)
1994년~현재 : 아주대학교 컴퓨터공학과 박사과정
1983년~현재 : 단국대학교 전산통계학과 교수
<주관심 분야> 소프트웨어 공학, 시스템 프로그램 등
e-mail : yoohy@dankook.ac.kr

심 재 홍(Jae-Hong Shim)

1987년 : 서울대학교 전산과학과 졸업(학사)
1989년 : 아주대학교 컴퓨터공학과 졸업(석사)
1989년~1994년 : 서울시스템 공학연구소
1995년~현재 : 아주대학교 컴퓨터공학과 박사과정 (수료)
<주관심 분야> 운영 체제, 분산시스템, 실시간 및 멀티미디어시스템
e-mail : jhshim@cesys.ajou.ac.kr

박 승 규(Seung-Kyu Park)

1974년 : 서울대학교 응용수학과 졸업(학사)
1976년 : 한국과학원 전산학과 졸업(석사)
1982년 : Institute National Polytechique de Grenoble 전산학과 졸업(박사)
1976년~1992년 : KIST, KIET, IBM왓슨 연구소, ETRI 연구위원
1992년~현재 : 아주대학교 정보 및 컴퓨터공학부 교수
<주관심 분야> 컴퓨터구조, 멀티미디어, 실시간컴퓨터시스템, 이동컴퓨터 등
e-mail : sparky@madang.ajou.ac.kr

최 경 희(Kyung-Hee Choi)

1976년 : 서울대학교 수학교육과 졸업(학사)
1979년 : 프랑스 그랑메꼴 Enseeiht대학 졸업(석사)
1982년 : 프랑스 Paul Sabatier 대학 정보공학부 졸업 (박사)
1982년~현재 : 아주대학교 정보 및 컴퓨터공학부 교수
<주관심 분야> 운영 체제, 분산시스템, 실시간 및 멀티미디어시스템 등
e-mail : khchoi@madang.ajou.ac.kr

정 기 현(Gi-Hyun Jung)

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
<주관심 분야> 컴퓨터구조, VLSI 설계, 멀티미디어 및 실시간 시스템 등
e-mail : khchung@madang.ajou.ac.kr