CVM 환경에서 임베디드 가비지 컬렉터의 성능 평가

차 창 일[†] · 김 상 욱[††] · 장 지 웅[†††]

요 약

가비지 컬렉션은 자바 가상 머신의 핵심적인 기능으로서 개발자들이 겪는 메모리 관리의 어려움을 줄여준다. 본 논문에서는 임베디드 자바 가상 머신을 위한 가비지 컬렉터인 GenGC와 GenRGC의 성능을 평가하고 분석한다. 성능 평가를 위하여 썬 마이크로시스템즈사에서 개발한 실제 임베디드 자바 가상 머신인 CVM을 플랫폼으로 사용하며, SpecJVM98을 벤치마크 프로그램 집합으로 사용한다. 첫째, GenGC와 GenRGC의 성능을 비교하기 위하여 힙의 크기 및 각 영역의 크기를 변화시키면서 가비지 컬렉션 수행 시간 및 지연 시간을 비교한다. 둘째, GenRGC의 성능을 보다 세밀하게 분석하기 위하여 힙의 구성 요소 중 블록과 프레임의 크기를 변화시키면서 가비지 컬렉션 수행 시간 및 지연 시간을 측정하여 분석한다. 셋째, GenRGC를 사용하기 위하여 필요한 저장 공간의 크기를 분석하고, GenRGC가 제한된 메모리를 가지는 임베디드 환경에서 적합하다는 것을 보인다. CVM은 가장 대표적인 임베디드 자바 가상 머신이므로, 이와 같은 성능 연구는 실제 응용 환경에서가비지 컬렉터의 성능을 보다 정확히 예측할 수 있다는 점에서 큰 의미를 갖는다.

키워드 : 자바, 자바 가상머신, J2ME, 메모리 관리, 가비지 컬렉션, CVM, 성능 평가

Performance Evaluation of Embedded Garbage Collectors in CVM Environment

Chang-Il Cha[†] · Sang-Wook Kim[††] · Ji-Woong Chang[†††]

ABSTRACT

Garbage collection in the Java virtual machine is a core function that relieves application programmers of difficulties related to memory management. In this paper, we evaluate and analyze the performance of GenGC and GenRGC, garbage collectors for embedded Java virtual machines. For performance evaluation, we employ CVM, a real embedded Java virtual machine developed by Sun Microsystems, Inc., as a platform, and also use a widely-used SpecJVM98 as a set of benchmark programs. To compare the performance of GenGC and GenRGC, we first evaluate the time of garbage collection and the delay time caused by garbage collection. Second, for more detailed performance analysis of GenRGC, we evaluate the time of garbage collection and the delay time caused by garbage collection while changing the sizes of a block and a frame. Third, we analyze the size of storage space required for performing GenRGC, and show GenRGC to be suitable for embedded environment with a limited amount of memory. Since CVM is the most representative one of embedded Java virtual machines, this performance study is quite meaningful in that we can predict the performance of garbage collectors in real application environments more accurately.

Key Words : Java, Java virtual machine, J2ME, memory management, garbage collection, CVM, performance evaluation

1. 서 론

최근, 스마트폰, PDA, 셋탑 박스 등 임베디드 기기의 사용이 급증하고 있다. 자바(Java) 언어는 객체지향성, 안전성, 유연성이 뛰어나 임베디드 기기에서 많이 사용하는 프로그래밍 언어의 하나이다. 자바 언어로 작성된 응용 프로그램이 동작하기 위한 자바 플랫폼은 J2EE, J2SE, J2ME로 구분된다. 이 중 J2ME(Java2 platform, micro edition)는 임베디드 환경을 위한 자바 기술로서 스마트폰, PDA, 스크린폰, 셋탑 박스, 넷 TV와 같이 네트워크로 연결된 임베디드 기기에서 사용된다[13].

자바 언어로 작성된 소스코드는 컴파일 되면 자바 바이트코드로 변환된다. 자바 바이트코드는 자바 가상머신에 의해 해석되고 실행된다. 자바 가상머신(Java virtual machine)이란 컴파일된 자바 바이트코드와 실제로 프로그램의 명령어를 실행하는 마이크로프로세서 또는 하드웨어 플랫폼 간에 인터페이스 역할을 담당하는 소프트웨어를 의미한다[11]. 자

※ 본 논문은 제주대학교를 통한 정보통신부 및 정보통신진흥원의 대학 IT연구센터 지원사업(IITA-2005-C1090-0502-0009)의 연구비 지원을 받았습니다.
†정 회 원 : (주)에이록스 연구원
††종신회원 : 한양대학교 정보통신대학 정보통신학부 교수
†††정 회 원 : 한국산업기술대학교 게임공학과 조교수
논문접수 : 2006년 11월 1일, 심사완료 : 2007년 4월 9일

바 바이트코드는 별도의 변환 작업을 하지 않아도 자바 가상머신이 동작하는 모든 플랫폼에서 동작이 가능하다.

자바를 개발한 썬 마이크로시스템즈사는 CDC 플랫폼을 위한 가상머신인 CVM과 CLDC 플랫폼을 위한 가상머신인 KVM 등 두 가지 종류의 임베디드 자바 가상머신을 제공하고 있다. 임베디드 자바 가상머신은 메모리 용량이 작고, 전력 자원이 제한된 임베디드 환경에 적합하도록 설계된 자바 가상머신이다. 특히 메모리 용량이 제한되어 있으므로, 자바 객체를 효율적으로 생성하고 관리하는 것이 매우 중요하다[4]. 메모리를 동적으로 할당 하고, 반환 하는 방법 중의 하나가 가비지 컬렉션이다.

가비지 컬렉션(garbage collection)이란 더 이상 사용되지 않는 메모리 영역을 자동적으로 수집하여 재사용할 수 있도록 하는 작업을 의미한다[9]. 가비지 컬렉션을 수행하기 위해서는 응용 프로그램에 의해 사용 중인 객체와 이미 사용이 종료된 객체를 식별하는 것이 필요하다. 본 논문에서는 응용 프로그램에 의해 사용 중인 객체를 살아있는 객체(live object)라 하고, 이미 사용이 종료되어 더 이상 사용되지 않는 객체를 죽은 객체(dead object)라고 부르기로 한다.

본 논문에서는 특별히 임베디드 자바 가상머신에서 사용되는 가비지 컬렉션 방법을 간략히 임베디드 가비지 컬렉션(embedded garbage collection)이라 부른다. 썬 마이크로시스템즈사의 대표적인 임베디드 자바 가상머신인 CVM에서는 가비지 컬렉션 방법으로 세대 가비지 컬렉터를 사용한다[14]. 본 논문에서는 이를 GenGC라 부른다. 그러나 GenGC는 큰 영역에 대한 가비지 컬렉션을 한 번에 처리하기 때문에 가비지 컬렉션으로 인한 지연 시간이 커서 임베디드 환경에서 요구하는 실시간 요구 사항을 만족하기 어려운 문제가 있다. 이러한 문제점을 해결하기 위하여 실시간 요구 사항을 고려하는 GenRGC가 제안되었다[3].

본 연구에서는 CVM 상에서 실험을 통하여 GenGC와 GenRGC의 성능을 평가하고, 각 방법의 특성을 분석한다. GenRGC는 [3]에서 아이디어를 제안하고 성능 평가를 수행한다. 그러나 [3]에는 GenRGC의 특성만 제안할 뿐 성능 평가의 비교 대상인 GenGC의 특성에 대한 분석이 없으며, 간단한 성능 평가만 수행하여 임베디드 환경에 적합한지를 보이는 데는 부족 하다. 따라서 본 연구에서는 GenGC와 GenRGC의 특성을 설명하고, 더 많은 벤치마크 프로그램으로 GenRGC가 지원하는 패라미터를 다양하게 변경하면서 다양한 실험을 수행한다. 기존에 제안된 다양한 가비지 컬렉터들에 대한 성능 평가가 이루어진 바 있으나, 대부분의 경우 실제 자바 가상 머신이 아닌 시뮬레이션 환경에서 수행되어 왔다. CVM은 가장 대표적인 임베디드 자바 가상머신이므로, 가비지 컬렉터를 CVM 상에서 구현하고 성능을 분석하는 것은 실제 응용 환경에서의 성능을 미리 예측할 수 있다는 점에서 큰 의미를 갖는다.

첫 번째 실험에서는 GenGC와 GenRGC의 성능을 비교하기 위하여 힙의 크기 및 각 영역의 크기를 변화시키면서 가비지 컬렉션 수행 시간 및 지연 시간을 비교한다. 두 번째 실험에서는 GenRGC의 성능을 보다 세밀하게 분석하기 위하여 힙의 구성 요소 중 블록과 프레임의 크기를 변화시키면서 가비지 컬렉션 수행 시간 및 지연 시간을 측정하여 분석한다. 또한, GenRGC를 사용하기 위하여 필요한 저장 공간의 크기를 분석하고, GenRGC가 제한된 메모리를 가지는 임베디드 환경에서 적합하다는 것을 보인다.

본 논문의 구성은 다음과 같다. 제 2장에서는 관련 연구로서 GenGC과 GenRGC에 대해서 설명한다. 제 3장에서는 GenGC와 GenRGC의 성능을 비교 평가하기 위하여 다양한 실험을 수행하고 결과를 논의한다. 마지막으로 제 4장에서는 본 논문을 요약하고 결론을 내린다.

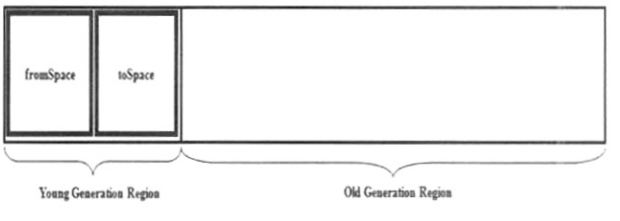
2. 관련 연구

2.1 GenGC

GenGC는 썬 마이크로시스템즈사에서 개발한 CVM[14]에서 동작하는 임베디드 가비지 컬렉션 방법으로서 세대 가비지 컬렉션(generational garbage collection)[10, 18]을 기반으로 한다[3]. 세대 가비지 컬렉션은 힙을 둘 이상의 세대로 분할하여 관리하는 방법으로서 대다수의 객체가 생성된 후 오래 지나지 않아서 소멸되는 응용 프로그램에서 효과적으로 동작하는 것으로 알려져 있다[1, 2, 19].

(그림 1)은 GenGC에서 힙을 관리하는 구조를 나타낸 것이다. GenGC는 힙을 젊은 세대(young generation)와 늙은 세대(old generation)의 비대칭적인 두 개의 세대로 분할한다. 새로운 객체는 상대적으로 크기가 작은 젊은 세대에 먼저 할당된다. 새로운 객체의 할당으로 인하여 젊은 세대가 가득 채워져서 메모리가 부족해지면 젊은 세대에서 가비지 컬렉션을 수행하여 메모리를 확보한다. 이 때, 살아있는 객체는 가비지 컬렉션의 대상에서 제외되는데, 젊은 세대 영역에서 오랜 시간 동안 살아있는 객체는 늙은 세대로 옮겨진다. 젊은 세대 가비지 컬렉션을 수행해도 더 이상 메모리를 할당할 수 없는 경우에는 늙은 세대 가비지 컬렉션을 수행하여 메모리를 할당한다.

가비지 컬렉션을 수행할 때에 살아있는 객체를 식별하기 위하여 도달 가능(reachable)하다는 개념을 이용한다. CVM에서는 루트 셋(root set)이라고 하는 살아있는 객체들에 대한 참조 값들의 집합을 관리한다. 그러므로 루트 셋으로부터 객체들의 참조 값을 추적하면 모든 도달 가능한 객체를 식별할 수 있다. 즉, 모든 살아있는 객체를 식별할 수 있다.



(그림 1) GenGC의 힙의 구조

대부분의 객체는 할당된 후 짧은 시간 동안만 사용되는 특성을 가지기 때문에 많은 수의 가비지 객체가 젊은 세대에서 회수된다. 따라서 상대적으로 크기가 작은 젊은 세대에서의 가비지 컬렉션이 보다 빈번하게 발생하고, 소수의 객체만이 늙은 세대로 이동한다. 그 결과, 가비지 컬렉션에 의한 지연 시간이 적고 전체 프로그램의 실행시간이 감소한다는 장점을 갖는다[9].

늙은 세대 가비지 컬렉션은 크기가 비교적 큰 늙은 세대 영역에 대해서 살아있는 객체를 식별하는 작업이 수반되어야 하므로 오버헤드가 매우 큰 작업이다. 따라서 가능하면 늙은 세대 가비지 컬렉션의 수행 횟수를 줄이는 것이 좋다. 이를 위하여 GenGC에서는 늙은 세대가 가득 찼을 때 곧바로 늙은 세대 가비지 컬렉션을 수행하는 것이 아니라 먼저 젊은 세대 가비지 컬렉션을 수행하여 메모리를 할당받는다. 젊은 세대 가비지 컬렉션을 수행해도 더 이상 메모리를 할당받을 수 없게 되면 비로소 늙은 세대 가비지 컬렉션을 수행한다. 즉, 젊은 세대에서 더 이상 객체를 할당할 수 있는 공간이 없는 경우에만 늙은 세대 가비지 컬렉션을 수행한다. 본 논문에서는 이러한 GenGC의 정책을 늙은 세대 가비지 컬렉션의 지연 수행 정책(delayed execution strategy)이라고 부르기로 한다. 늙은 세대 가비지 컬렉션의 지연 수행 정책은 늙은 세대 가비지 컬렉션의 수행 횟수를 감소시키는 효과가 있으나, 반면에 젊은 세대 가비지 컬렉션을 빈번하게 수행하여야 하는 문제점이 있다.

젊은 세대와 늙은 세대의 특성이 다르므로 가비지 컬렉션 방법도 서로 다른 방법을 사용한다. 젊은 세대에서는 구현이 쉽고, 수행 부하가 적은 세미 스페이스 복사 컬렉터(semi-space copying collector)[5]를 사용하고, 늙은 세대에서는 객체 참조의 지역성(locality of reference)이 높은 마크 콤팩트 가비지 컬렉터(mark-compact garbage collector)[6]를 사용해 가비지 컬렉션을 수행한다.

세미 스페이스 복사 컬렉터는 젊은 세대 영역을 fromSpace와 toSpace로 공간을 이등분하여 관리한다. fromSpace는 동적으로 객체를 할당하는 영역으로서 fromSpace가 가득 차게 되면 젊은 세대 가비지 컬렉션이 수행된다. toSpace는 fromSpace에 대한 가비지 컬렉션을 수행할 때에 사용 중인 객체를 복사하여 저장하는 공간으로 사용되는 영역이다.

젊은 세대의 가비지 컬렉션은 먼저 루트 셋으로부터 도달 가능한 모든 객체들을 검색하여 toSpace로 복사한다. 모든 도달 가능한 객체들이 toSpace로 복사되면 fromSpace를 비우고 가비지 컬렉션을 완료한다. 가비지 컬렉션이 완료되면 fromSpace와 toSpace의 역할이 바뀌게 되어 이후에는 toSpace가 객체의 할당을 담당한다.

세미 스페이스 복사 컬렉션은 객체의 연속적인 할당을 보장한다. 즉, 주어진 공간의 한 쪽 끝에서부터 객체들이 빈틈없이 채워지므로 단편화 현상이 없다. 또한, 함께 할당된 객체들은 함께 사용될 가능성이 높으므로 객체를 연속적으로 할당하는 방법은 참조의 지역성(locality of reference)을 높일 수 있다[2]. 또한, 구현이 용이하고, 젊은 세대 영역에 있는 도달 가능한 객체만을 복사하면 되므로 동작에 부하가 적다는 추가적인 장점을 갖는다[9].

늙은 세대에 대한 가비지 컬렉션은 젊은 세대에서 가비지 컬렉션을 수행해도 더 이상 메모리를 할당할 수 없는 경우에 수행된다. 먼저, 루트 셋으로부터 도달 가능한 객체들을 검색하여 마크를 하고, 마크된 객체들을 늙은 세대 영역의 앞쪽으로 차례로 복사함으로써 데이터를 압축한다. 마지막으로 객체들이 압축으로 인해 이동되었으므로, 이 이동된 객체들을 참조하는 객체 및 루트 셋의 해당 참조 값을 수정하고 가비지 컬렉션을 완료한다.

GenGC는 가비지 컬렉션을 수행할 때에 힙의 전체 영역을 대상으로 하지 않고 분할된 일부 영역만을 대상으로 한다. 이 경우, 살아있는 객체를 모두 식별하는 것이 불가능하다. 왜냐하면 한 영역의 객체가 다른 영역의 객체를 참조하는 경우가 있기 때문이다. 그러나 살아있는 객체를 식별하기 위하여 가비지 컬렉션이 수행될 때마다 힙 상의 모든 객체를 추적하는 일은 매우 오버헤드가 크다.

이러한 문제를 해결하기 위하여 쓰기 장벽(write-barrier)을 사용한다[20]. 쓰기 장벽이란 메모리상의 특정 영역에 쓰기 연산을 수행할 때에, 이를 감지하여 특정 메모리 관리 연산을 수행시키는 일련의 메카니즘이다[20]. 어떤 영역의 객체가 다른 영역의 객체를 참조하게 되면 쓰기 장벽을 사용하여 해당 객체에 표시를 함으로써 가비지 컬렉션을 효율적으로 수행할 수 있다.

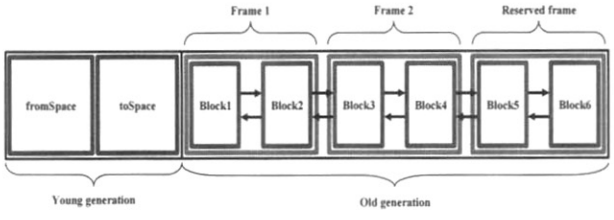
GenGC는 젊은 세대와 늙은 세대에서 모두 객체들을 연속적으로 할당하기 때문에 객체의 빠른 할당이 가능하고, 객체의 참조 지역성을 높이며, 단편화가 발생하지 않는다. 또한 늙은 세대 영역에 대해서 마크 콤팩트 가비지 컬렉션을 수행할 때 데이터를 압축하므로 공간 활용도도 우수하다.

그러나 늙은 세대 가비지 컬렉션은 크기가 큰 늙은 세대 영역 전체에 대하여 가비지 컬렉션을 수행하기 때문에 가비지 컬렉션에 의한 지연 시간이 매우 길다. 또한, 가비지 컬렉션을 수행하기 위해 이동된 객체들을 참조하는 모든 객체 및 루트 셋의 참조 값을 수정해야 하므로 오버헤드가 크다. 따라서 빠른 응답 시간을 요구하는 임베디드 환경에서 적합하다 할 수가 없다.

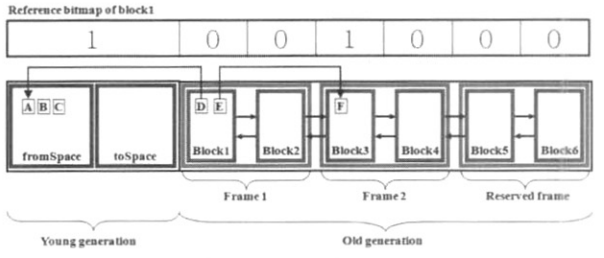
2.2 GenRGC

GenRGC는 GenGC가 늙은 세대에 대하여 수행하는 가비지 컬렉션 방법의 단점을 개선한 방법이다[3]. GenRGC에서는 늙은 세대의 영역을 분할하여 가비지 컬렉션을 부분적으로 수행하되, 이를 점진적으로 수행함으로써 실시간 요구사항을 충족시킨다.

(그림 2)는 GenRGC에서 힙을 관리하는 구조를 나타낸 것이다. GenRGC에서는 늙은 세대를 균등한 크기의 블록이라는 단위로 분할하여 관리한다. 블록들은 이중 연결 리스트로 상호 연결되어 있으며 객체 저장의 단위가 된다. 이 블록들을 여러 개 묶어서 프레임을 형성하는데 이 프레임은 가비지 컬렉션의 단위가 된다. 프레임을 구성하는 블록의



(그림 2) GenRGC에서 힙의 구조



(그림 3) 쓰기 장벽과 참조 비트맵

개수를 프레임의 크기라고 한다.

각 블록은 할당이 발생한 순서대로 나이를 가진다. 즉, (그림 2)에서 블록1에서부터 블록4의 순서로 할당이 발생한다면 가장 나이가 많은 블록은 블록1이 된다. 전체 프레임 중 한 프레임은 객체 할당을 위하여 사용하지 않고 가비지 컬렉션 수행을 위한 예비 공간으로 보존된다.

젊은 세대의 가비지 컬렉터는 지정된 나이가 된 객체를 늙은 세대로 이동시킨다. 이 때 늙은 세대에서의 메모리 할당은 블록의 한 쪽 끝에서 연속적으로 할당된다. 할당 요청된 객체의 크기가 현재 할당 중인 블록의 남은 여유 공간보다 큰 경우에는 다음 번호의 새로운 블록에 할당된다. 응용 프로그램이 시작해서 가비지 컬렉션을 위해 예약된 프레임을 제외한 늙은 세대의 모든 블록에 객체가 가득 차기 전에는 가비지 컬렉터가 동작하지 않는다.

늙은 세대의 가비지 컬렉션은 지연 수행 정책을 채택하는 GenGC에서와는 달리 젊은 세대로부터 일정한 나이에 도달한 객체를 늙은 세대로 이동할 때 그 공간이 부족하면 수행된다. 컬렉션의 대상으로 선정되는 프레임은 가장 나이가 많은 프레임이다. 그 이유는 나이가 가장 많은 프레임이 가장 많은 가비지 객체를 포함하고 있을 가능성이 높기 때문이다 [8, 15, 16, 17]. 가비지 컬렉터는 대상이 되는 프레임에서 루트 셋으로부터 도달 가능한 객체를 미리 예약된 빈 프레임으로 복사시킨다. 모든 도달 가능한 객체들을 옮긴 후 프레임은 비워지고 다음 가비지 컬렉션을 위한 예비 공간으로 사용된다. 복사가 완료된 프레임의 블록의 나이는 가장 어린 나이로 설정되어 젊은 세대로부터의 다음 할당을 기다린다. 한 프레임의 가비지 컬렉션이 완료되면 멈춰있던 응용 프로그램의 수행이 재개된다.

GenRGC에서도 GenGC와 같이 젊은 세대와 늙은 세대를 분할하여 가비지 컬렉션을 수행하기 때문에 도달 가능한 객체를 효율적으로 파악하기 위하여 쓰기 장벽을 사용한다. 더구나 늙은 세대 내에서도 영역을 프레임으로 분할하여 가비지 컬렉션을 수행하기 때문에 이 문제를 해결하기 위하여 새롭게 고안한 2단계 쓰기 장벽을 사용한다[3].

2단계 쓰기 장벽 기법은 도달 가능한 객체를 식별하기 위해 2단계 필터링을 사용한다. 제 1차 필터는 가비지 컬렉션의 대상이 각 블록에 대하여 그 블록내의 객체들이 참조하는 블록들을 식별한다. 제 1차 필터링을 위하여 각 블록의 헤더는 (그림 3)과 같이 참조 비트맵을 가진다. 비트맵 내의 각 비트는 힙 내의 블록과 1대1 대응되며, 따라서 각 블록내의 비트맵은 블록 수만큼의 비트수를 갖는다. 한 블록A내의 한 객체가 다른 세대 혹은 다른 블록B 내의 객체를 참조하게 되면 블록A의 참조 비트맵은 참조가 되는 세대 또는 블록B와 대응되는 비트를 1로 설정한다. 가비지 컬렉터는 블록 헤더의 참조 비트맵을 검사함으로서 현재의 블록을 참조하는 객체를 가지는 모든 블록들을 검출해낼 수 있다. (그림 3)의 예에서, 블록1은 블록3과 젊은 세대 내의 객체들을 참조함을 볼 수 있다. 참조 비트맵은 한 블록A에서 다른 블록B로의 참조가 발생하면, 블록A내의 비트맵에서 블록B에 해당하는 비트가 즉시 1로 설정된다. 그러나 블록A의 참조 비트맵의 초기화는 블록A를 대상으로 가비지 컬렉션이 수행될 때 이루어진다. 제 1차 필터에 의해 추적해야 할 블록이 결정되더라도 하나의 블록에는 많은 수의 객체가 존재할 수 있으므로 이 객체들을 모두 추적하는 것은 역시 상당한 시간이 소요된다. 따라서 블록 내에서 다른 블록을 참조하는 객체를 구체적으로 식별하기 위해 제안된 기법에서는 제 2차 필터를 둔다. 제 2차 필터는 힙을 페이지라는 물리적 단위로 다시 세분화한 후 각 페이지마다 참조 상태를 저장하여 참조 상태에 따라 객체 추적을 달리하는 방법이다[1].

GenRGC는 쓰기 장벽을 사용하여 가비지 컬렉션의 대상이 되는 블록의 도달 가능한 객체를 찾기 위하여 전체 힙에 존재하는 객체 추적의 부하를 응용 프로그램의 실행 중에 분산시킴으로써 가비지 컬렉션의 지연 시간을 줄인다. 이를 위해 2단계 쓰기 장벽을 사용하기 때문에 추가적인 공간이 소요된다. 본 논문에서는 실험을 통해 2단계 쓰기 장벽을 위한 추가적인 저장 공간의 크기를 측정하고 이 공간이 충분히 수용할 수 있을 만큼 작음을 보인다.

3. 성능 평가

본 장에서는 실험을 통하여 실제 CVM상에서 GenGC와 GenRGC의 성능을 비교, 평가하고 각 방법의 특성을 분석한다. 그리고 실험 결과 더 우수한 것으로 나타난 GenRGC에 대하여 다양한 추가 실험을 통해 성능을 분석한다. 제 3.1절에서는 실험 환경에 대하여 설명하고, 제 3.2절에서는 실험 결과를 제시하고 분석한다.

3.1 실험 환경

먼저, GenGC와 함께 GenRGC를 구현하여 CVM에 포팅

1) 제 2차 필터는 CVM[13]을 위한 쓰기 장벽 기법으로서 채택하고 있다.

하였다. 또한, 성능 측정을 위한 벤치마크 프로그램으로는 SpecJVM98[12]을 사용하였다. SpecJVM98은 파서를 생성하는 프로그램(_228_jack), 퍼즐을 해결하는 프로그램(_202_jess), 레이 트레이싱을 하는 프로그램(_205_raytrace), 멀티쓰레드 환경에서 레이 트레이싱을 하는 프로그램(_227_mtrt), MPEG-3 디코더 프로그램(_222_mpegaudio), 자바 컴파일러 프로그램(_213_javac), 데이터베이스 관리 프로그램(_209_db), 파일 압축하고 복원하는 프로그램(_201_compress)의 8가지 응용 프로그램 집합이며, 가비지 컬렉터의 성능 측정을 위하여 널리 사용된다.

메모리 할당 및 가비지 컬렉션과 관련된 각 프로그램의 특징은 다음과 같다.

- _228_jack: 파서가 지워질 때에 사용하던 대부분의 객체들이 가비지 객체가 되기 때문에 힙의 크기가 작더라도 동작할 수 있다.
- _202_jess: 수행 중에 일정한 수준의 메모리를 항상 유지하며, 하나의 퍼즐을 해결하는 동안에는 많은 메모리를 할당한다.
- _205_raytrace: 파일에서 읽어 들인 그림을 베껴서 그리는 작업을 수행한다.
- _227_mtrt: 파일에서 읽어 들인 그림을 여러 섹션으로 나누고 각 섹션 당 하나의 쓰레드가 수행이 되어 그림을 베껴서 그리는 작업을 수행한다.
- _222_mpegaudio: 수행 중 매우 적은 양의 메모리만 사용하기 때문에 가비지 컬렉션이 수행 되지 않는다.
- _213_javac: 컴파일 수행 때, 많은 메모리를 할당하여 사용하며, 컴파일 완료 후에는 _228_jack과 유사하게 컴파일 과정에서 사용된 대다수의 객체들은 가비지 객체가 된다.
- _209_db: 트랜잭션을 수행하는 동안에 일정한 메모리를 지속적으로 사용한다.
- _201_compress: 파일을 메모리로 읽어 들이기 위한 바이트 배열과 압축 및 복원 결과를 저장하기위한 바이트 배열의 2개 큰 바이트 배열을 메모리에 할당한다.

실험은 크게 두 부분으로 구성된다. 첫 번째는 GenGC와 GenRGC의 성능을 비교하는 실험이고, 두 번째는 실험 결과 더 우수한 것으로 보여진 GenRGC에 대하여 실험에 필요한 여러 요소를 변화시키면서 성능을 분석하는 실험이다.

먼저, 실험 1에서는 GenGC와 GenRGC의 성능을 비교 분석한다. 본 실험은 세 부분으로 구성된다. 실험 1-1에서는 힙의 크기를 변화시키면서 응용 프로그램의 수행 시간과 전체 가비지 컬렉션 수행 시간을 측정하여 성능을 비교한다. 전체 가비지 컬렉션 수행 시간은 프로그램이 동작하는 동안 수행된 모든 가비지 컬렉션 수행 시간의 합을 의미하는 것으로 가비지 컬렉션 수행 횟수와 가비지 컬렉션 수행 시간에 의하여 결정된다. 전체 가비지 컬렉션 수행 시간이 작을수록 동일한 프로그램을 동작시키기 위하여 필요로 하는 가비지 컬렉션의 오버헤드가 작다는 것을 의미한다.

실험 1-2에서는 힙의 크기가 고정되어 있을 때 젊은 세대 영역의 크기를 변화시키면서 전체 가비지 컬렉션 수행 시간을 측정하여 성능을 비교한다. 젊은 세대 영역의 크기가 작으면 젊은 세대 가비지 컬렉션을 한번 수행하는데 소요되는 시간은 작으나 자주 가비지 컬렉션을 수행하게 된다.

실험 1-3에서는 가비지 컬렉션에 의한 최대 지연 시간을 성능 평가의 척도로 하여 실험을 수행한다. 최대 지연 시간은 가비지 컬렉션 방법의 실시간 요구 사항에 대한 적합성을 평가하는 중요한 지표가 된다. 즉, 지연 시간이 클수록 실시간 요구 사항을 만족시키지 못하는 것을 의미한다. 실험 1-3에서는 힙의 크기를 변화시키면서 각 방법의 최대 지연 시간의 변화를 측정하고, 힙의 크기가 고정되어 있을 때에 응용 프로그램의 수행 시간에 따른 지연 시간의 분포를 살펴본다. 지연 시간이 고르게 분포될수록 실시간 응용에 더욱 적합한 방법이라고 할 수 있다.

실험 2에서는 GenRGC 방법의 주요 구조인 블록과 프레임의 크기에 따른 성능을 측정한다. 실험 2-1에서는 블록 및 프레임의 크기와 전체 가비지 컬렉션 수행 시간의 관계를 살펴보고, 실험 2-2에서는 블록의 크기와 최대 지연 시간의 관계를 살펴본다. 또한, 실험 2-3에서는 GenRGC를 사용하기 위하여 필요한 저장 공간의 크기를 측정하여 이 방법이 임베디드 환경의 제한적인 메모리 내에서 충분히 수용 가능 하다는 것을 보인다.

실험은 RedHat Linux 2.4.20-8 버전의 운영체제에서 펜티엄II 400 MHz의 CPU, 256 MB의 메모리, 32 KB의 L1 Cache, 512KB의 L2 Cache를 가지는 시스템으로 수행하였다.

3.2 실험 결과

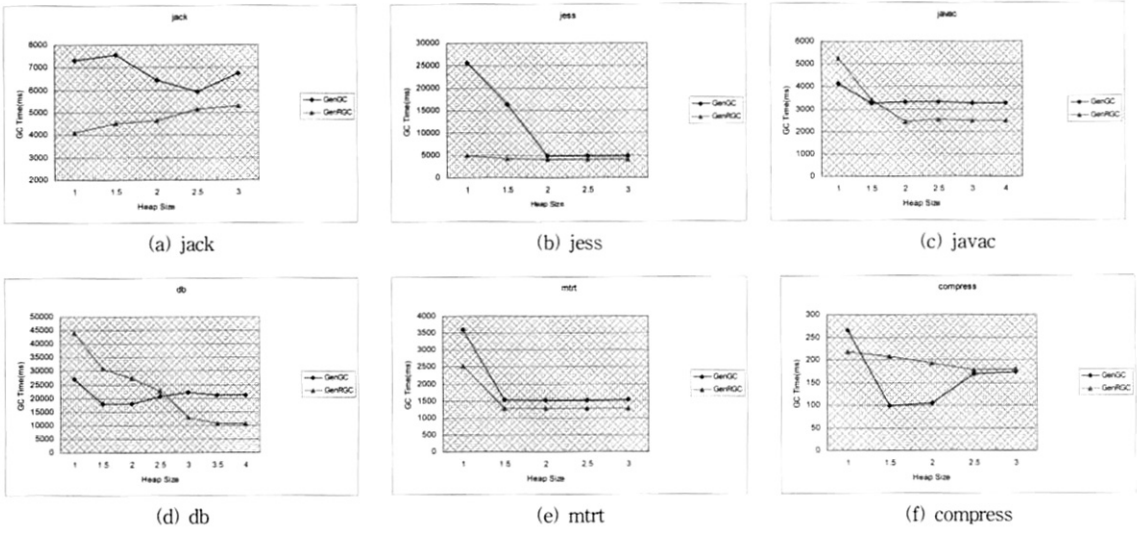
실험 1. GenGC와 GenRGC의 비교

실험 1-1. 힙의 크기에 따른 전체 가비지 컬렉션 수행 시간

본 실험에서는 젊은 세대 영역의 크기는 고정된 상태에서 힙의 크기를 변화시키면서 응용 프로그램의 수행 시간과 전체 가비지 컬렉션 수행 시간을 측정한다. <표 1>은 프로그램에 따른 젊은 세대 영역의 크기, 블록 크기, 그리고 힙의 기본 크기에 대한 설정값을 나타낸 것이다. 본 실험에서 젊은 세대의 크기는 메모리를 적게 사용하는 _222_jack과 _202_jess의 경우에는 256KB으로 설정하고, 나머지 프로그램의 경우에는 1MB로 설정하였다. 블록 크기는 최소 128KB에서 각 벤치마크 프로그램이 가지는 가장 큰 객체를 할당할 수 크기 내에서 설정하였다. 힙의 기본 크기는 각 프로그램을 수행할 때 늙은 세대 가비지 컬렉션이 필요한 상황이 발생하는 힙의 크기 중 가장 작은 크기를 의미한다.

〈표 1〉 각 벤치마크 프로그램의 수행을 위한 환경 설정

프로그램 명	젊은 세대 크기(KB)	블록 크기(KB)	힙의 기본 크기(KB)
_228_jack	256	128	1,408
_202_jess	256	128	1,408
_227_mtrt	1,024	256	7,168
_213_javac	1,024	128	11,776
_209_db	1,024	2,048	8,704
_201_compress	1,024	4,096	8,704



(그림 4) 힙의 크기에 따른 전체 가비지 컬렉션 수행 시간

(그림 4)는 힙의 크기를 힙의 기본 크기의 1, 1.5, 2, 2.5, 3배로 증가시키면서 각 프로그램을 수행하였을 때, 전체 가비지 컬렉션 수행 시간의 변화를 나타낸 것이다. (그림 4)에서 거의 모든 실험 영역에서 GenRGC의 수행 시간이 GenGC보다 적음을 볼 수 있다. 그 이유는 첫째, 늙은 세대 가비지 컬렉션이 점진적으로 수행되면서 가비지에 의한 젊은 세대 내의 객체로의 참조가 줄어들었기 때문이며, 둘째, 본 논문에서 제안한 2단계 쓰기 장벽의 효과로 인해 젊은 세대로의 참조를 가진 객체를 검색하는 시간이 줄어들었기 때문이다.

전체 가비지 컬렉션 수행 시간은 일반적으로 힙의 크기가 증가함에 따라 감소하는 경향을 보인다. 힙의 크기가 증가하면 힙에서 수용할 수 있는 객체의 수가 증가하여 가비지 컬렉션의 수행 횟수가 감소하게 되며, 이로 인하여 전체 가비지 컬렉션의 수행 시간이 감소하게 된다.

그러나 jack의 경우, GenRGC에서 힙의 크기가 증가함에 따라 수행 시간이 증가하는 경향을 볼 수 있다. 그 이유는 다음과 같다. jack이 생성하는 객체들은 jess, mtrt, javac, compress에 의해 생성되는 객체들과 비교하여 다른 세대 혹은 다른 블록으로의 참조를 많이 가지는 특징이 있다[7]. 그러므로 힙의 크기가 증가하면 가비지 컬렉션 대상이 되는 영역의 살아있는 객체를 식별하는데 소요되는 시간도 증가하게 된다.

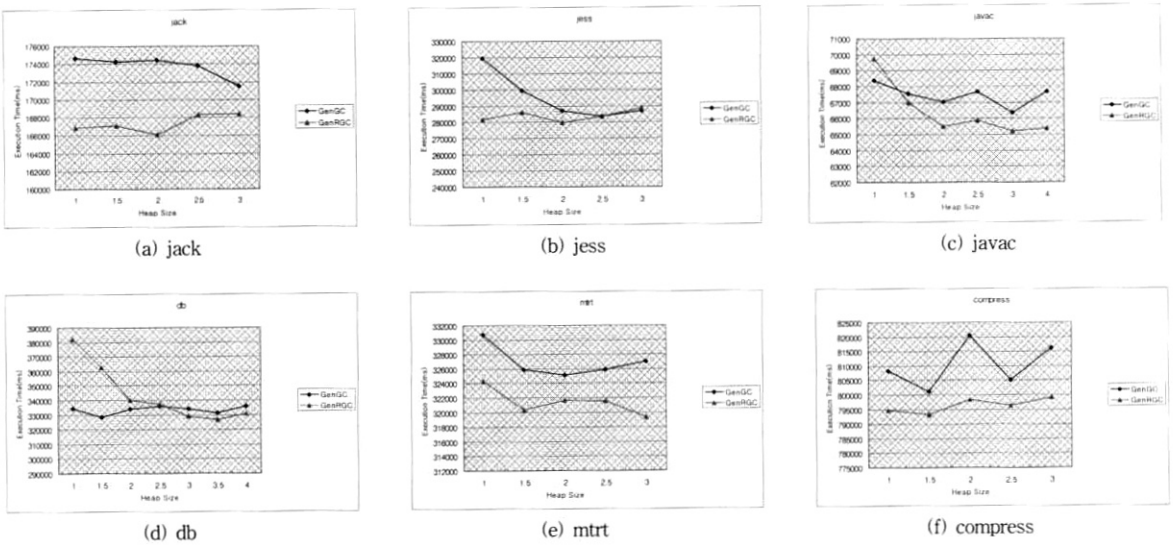
javac와 db의 경우, 힙의 크기가 작을 때에는 GenGC가 GenRGC보다 더 좋은 성능을 보이다가 힙의 크기가 커짐에 따라 성능이 역전되는 것을 알 수 있다. 이러한 현상은 두 프로그램이 모두 상당한 시간동안 객체들을 살아있는 채로 유지하기 때문에 생기는 현상이다. javac는 컴파일을 수행할 때 많은 메모리를 할당하여 컴파일이 완료될 때까지 유지하고, db는 트랜잭션을 수행할 때 레코드 파일을 읽어 들여 메모리에 오랜 시간 동안 유지한다. 그 결과, 힙의 크기가 매우 작은 경우에는 객체를 할당할 영역을 확보하기 위하여 젊은 세대뿐만 아니라 늙은 세대에 대한 가비지 컬렉션이 수행되어야 한다. GenRGC의 경우, 하나의 프레임에 대한 가비지 컬렉션만으로 필요한 공간을 확보하지 못하는 경우에는 다른 프레임에 대하여 연속적으로 가비지 컬렉션을 해야 하는 현상이 발생한다. 이러한 현상으로 인하여 힙의 크기가 작은 경우 GenGC에 비해 GenRGC의 성능이 낮은 것으로 나타난다. 그러나 이러한 문제는 힙의 크기가 비정상적으로 작은 경우에 나타나는 문제로서 힙의 크기가 충분히 커지면 GenRGC가 더 좋은 성능을 보인다.

(그림 5)는 힙의 크기를 힙의 기본 크기의 1, 1.5, 2, 2.5, 3배로 증가시키면서 각 프로그램을 수행하였을 때, 응용 프로그램의 수행 시간을 나타낸 것이다. 응용 프로그램의 수행 시간은 (그림 4)에서 나타난 전체 가비지 컬렉션 수행 시간과 비교적 유사한 경향을 보이는 것을 알 수 있다. 이는 가비지 컬렉션 수행 시간이 전체 프로그램 수행 시간에 많은 영향을 주는 것을 의미한다.

실험 1-2. 젊은 세대 크기에 따른 가비지 컬렉션 수행 시간

본 실험에서는 힙의 크기를 32MB로 고정하고[2], 젊은 세대의 크기를 32KB에서 16MB까지 2배씩 늘려가면서 성능을 측정하였다. (그림 6)은 젊은 세대의 크기에 따른 전체 가비지 컬렉션 수행 시간을 나타낸 것이다. 전반적으로 젊은 세대의 크기가 증가함에 따라 전체 가비지 컬렉션의 수행 시간이 감소하는 경향을 보인다. 이는 젊은 세대의 크기의 증가하면 새로 할당되는 객체를 위한 공간을 쉽게 할당 받을 수 있으므로, 젊은 세대 가비지 컬렉션 수행 횟수가 감소하기 때문이다. 반면에 가비지 컬렉션을 수행할 대상 영역의

2) 힙의 크기는 젊은 세대 영역과 늙은 세대 영역을 모두 포함할 수 있도록 충분히 큰 값으로 설정하였다. 또한, 실험에서 GenRGC의 경우 블록의 크기는 4MB로 설정하고, 하나의 프레임의 크기는 1 블록으로 설정하였다. 본 실험은 젊은 세대의 크기가 전체 가비지 컬렉션 수행 시간에 미치는 영향을 측정하기 위한 것이므로 늙은 세대 영역에 해당하는 블록과 프레임의 크기는 실험 결과에 영향을 미치지 않는다.



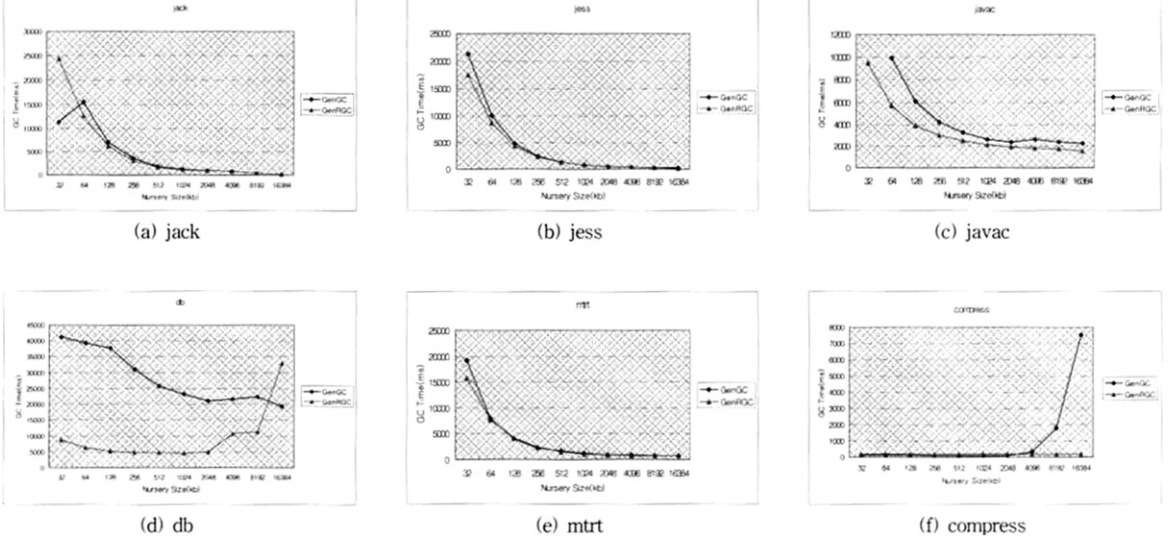
(그림 5) 힙의 크기에 따른 응용 프로그램 수행 시간

크기가 증가하므로 젊은 세대 가비지 컬렉션을 1회 수행하는 시간도 더 커진다. 즉, 젊은 세대 영역이 증가하면 가비지 컬렉션을 수행하는 횟수는 감소하고, 1회 수행할 때 소요되는 수행 시간은 증가한다.

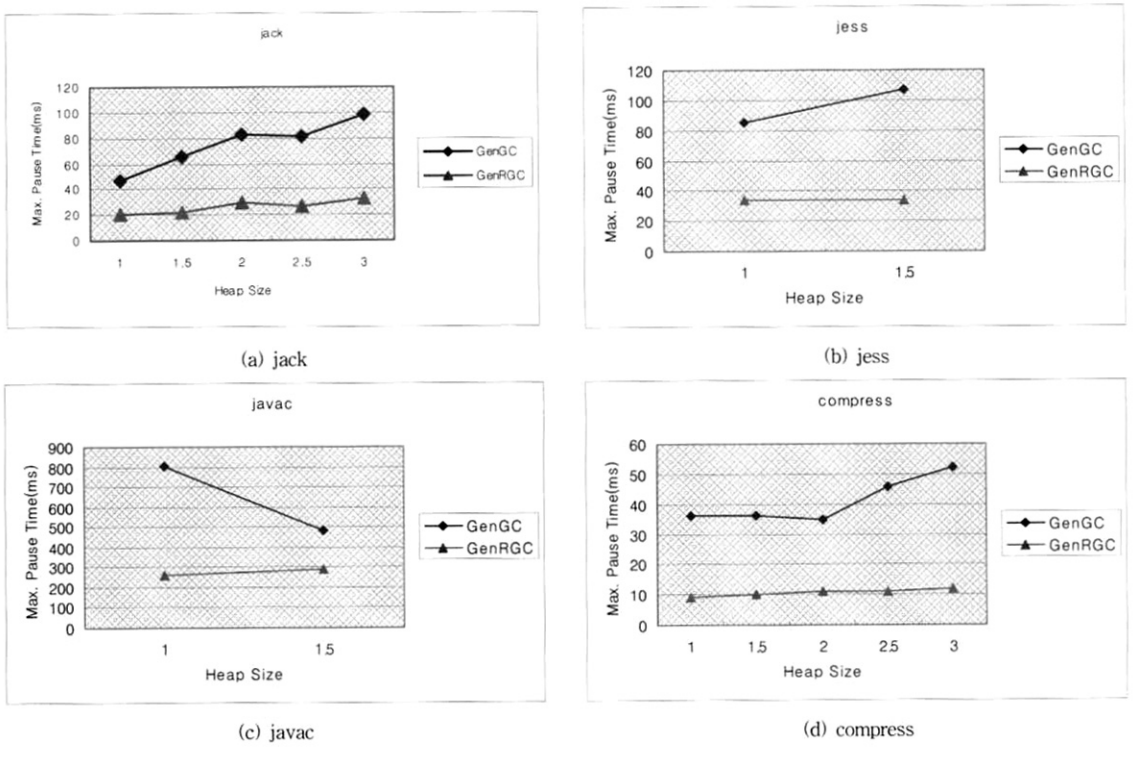
가비지 컬렉션을 위해 복사되는 데이터 양을 비교해 보면, 큰 메모리 영역에 대해 적은 횟수의 가비지 컬렉션을 수행하는 경우가 작은 메모리 영역에 대해 여러 번의 가비지 컬렉션을 수행하는 경우보다 복사되는 데이터 양이 더 작다. 왜냐하면 가비지 컬렉션이 여러 번 수행되는 경우 반복적으로 복사되는 데이터가 존재하기 때문이다. 즉, 젊은 세대의 크기를 크게 설정하여 증가되는 가비지 컬렉션 수행 시간보다 젊은 세대 가비지 컬렉션을 자주 수행함으로 인해 야기되는 반복 작업 수행 시간이 더 크게 된다.

db에서 GenRGC의 경우 젊은 세대 크기가 2,048KB 이상일 때 전체 가비지 컬렉션 수행 시간이 급속하게 증가한다. 이는 힙의 크기가 고정되어 있기 때문에 젊은 세대 영역의 크기가 증가함에 따라 상대적으로 늙은 세대 영역의 크기가 감소하게 되고, 이로 인해 늙은 세대 가비지 컬렉션이 일찍 수행되기 때문이다. 반면에 GenGC의 경우에는 늙은 세대 영역의 크기가 감소하더라도 늙은 세대 가비지 컬렉션 지연 수행 정책에 의해 늙은 세대 가비지 컬렉션이 수행되지 않고 젊은 세대 가비지 컬렉션이 수행되므로 일시적으로 GenRGC에 비해 좋은 성능을 보인다.

그러나 늙은 세대 가비지 컬렉션 지연 수행 정책이 항상



(그림 6) 젊은 세대 영역의 크기에 따른 전체 가비지 컬렉션 수행 시간



(그림 7) 힙의 크기에 따른 최대 지연 시간

좋은 결과를 보이는 것은 아니다. compress에서 GenGC의 경우 젊은 세대 영역의 크기가 4096KB 이상일 때 전체 가비지 컬렉션 수행 시간이 급격히 증가하는 것을 볼 수 있다. 이는 GenGC의 늙은 세대 가비지 컬렉션 지연 수행 정책에 의하여 늙은 세대 가비지 컬렉션 수행 횟수는 약간 감소하였으나 반면에 젊은 세대 가비지 컬렉션의 수행 횟수가 급격하게 증가하였기 때문이다. 늙은 세대 가비지 컬렉션을 지연시키고 대신 젊은 세대 가비지 컬렉션을 수행하는 경우 매우 작은 양의 메모리를 확보하게 되면 반복적으로 젊은 세대 가비지 컬렉션을 수행하게 되어 결과적으로 전체 성능을 저하시키는 요인이 된다.

실험 1-3 힙의 크기에 따른 최대 지연 시간

본 실험에서는 먼저 젊은 세대 영역의 크기와 블록의 크기를 <표 1>과 동일하게 설정하고, 프레임의 크기를 1로 고정한 상태에서 힙의 크기를 힙의 기본 크기의 1, 1.5, 2, 2.5, 3배로 증가시키면서 각 방법의 최대 지연 시간의 변화를 측정한다. 그리고 힙의 크기가 고정된 상태에서 응용 프로그램의 수행 시간에 따른 지연 시간의 분포를 살펴본다.

(그림 7)은 힙의 크기를 변화시키며 각 프로그램을 수행하였을 때, 늙은 세대 가비지 컬렉션의 1회 수행에 의한 최대 지연 시간을 나타낸다[3]. (그림 7)에서 늙은 세대 가비지 컬렉션에 의한 최대 지연 시간은 GenGC의 경우 힙의 크기가 증가함에 따라 증가하는 경향을 보이는 반면, GenRGC의 경우는 거의 일정함을 볼 수 있다. 이는 GenRGC의 경우 항상 정해진 크기의 영역을 점진적으로 가비지 컬렉션하기 때문이다.

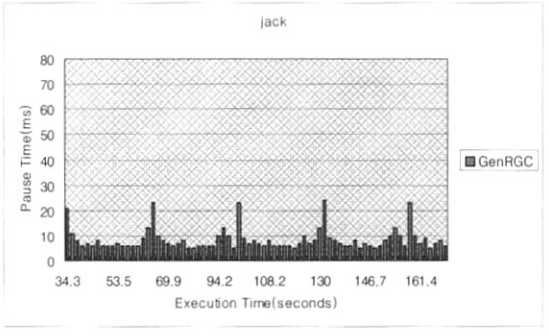
(그림 8)은 힙의 크기를 기본 크기로 고정하고 프로그램을 수행 하였을 때, 늙은 세대 가비지 컬렉션에 의한 지연 시간만을 나타낸 것이다. 그림에서 GenRGC는 가비지 컬렉션에 의한 지연 시간이 비교적 고르게 분포됨을 볼 수 있다. 반면, GenGC는 가비지 컬렉션에 의한 지연 시간이 매우 크게 변화한다. 따라서 실시간 요구 사항을 가지는 임베디드 환경에서는 GenRGC가 더 적합하다고 할 수 있다.

실험 2. GenRGC의 성능 평가

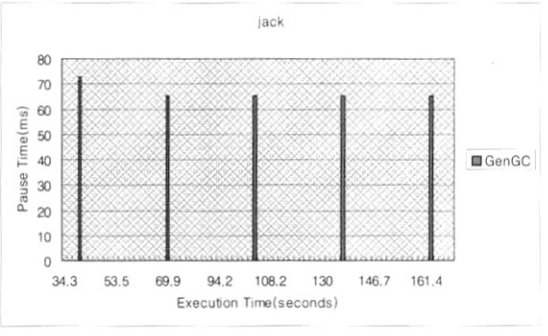
실험 2-1 블록 및 프레임의 크기에 따른 가비지 컬렉션 수행 시간

본 실험에서는 젊은 세대 영역의 크기는 실험 1-1에서와 동일하게 설정하였다. 힙의 크기는 기본 크기로 고정한 상태에서 먼저 블록의 크기를 128KB에서부터 1MB까지 2배씩 증가시키면서 GenRGC의 가비지 컬렉션 수행 시간을 측정하였다. 또한, 프레임의 크기를 1부터 8까지 2배씩 증가시키며 GenRGC의 가비지 컬렉션 수행 시간을 측정하였다. 응용 프로그램은 실험 1의 6가지 응용 프로그램 중 db와 compress를 제외한 나머지 프로그램만을 수행하였다. 왜냐하면 db와 compress는 크기가 1MB 이상인 객체를 생성하는데, 이 객체를 늙은 세대 영역에 할당하기 위해서는 블록

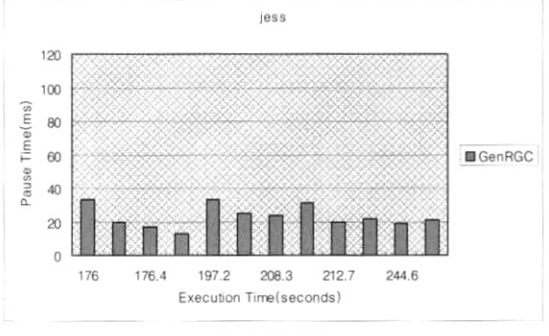
3) 젊은 세대 가비지 컬렉션에 의한 지연 시간은 임베디드 환경에서 실시간성을 보장할 수 있을 만큼 충분히 작기 때문에 실험 결과에서 제외하였다. 또한, jess와 javac의 경우 힙의 크기가 기본 크기의 2배 이상인 경우에는 늙은 세대 가비지 컬렉션이 수행되지 않았으므로 실험 결과에서 제외하였다.



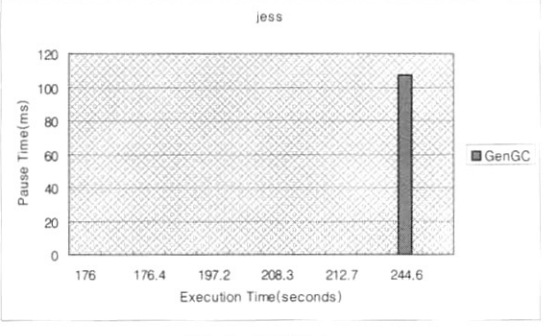
(a) GenRGC에서 jack



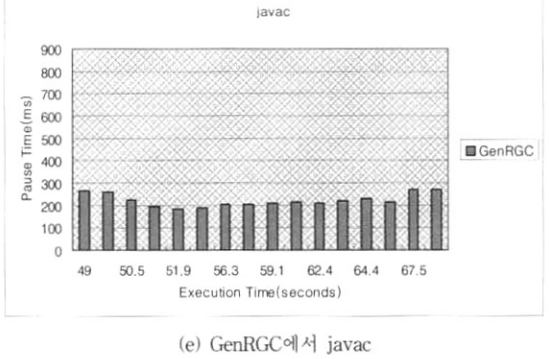
(b) GenGC에서 jack



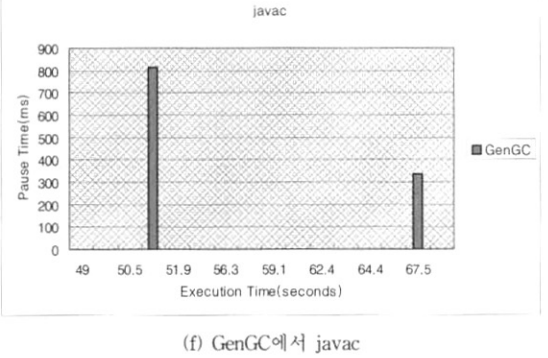
(c) GenRGC에서 jess



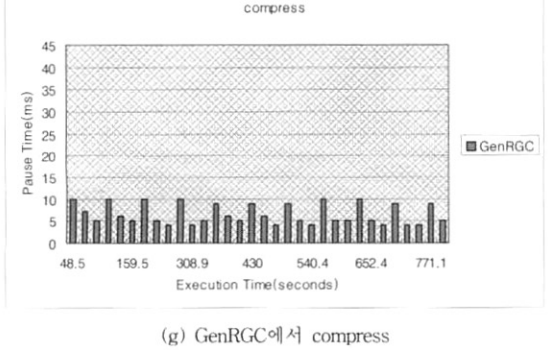
(d) GenGC에서 jess



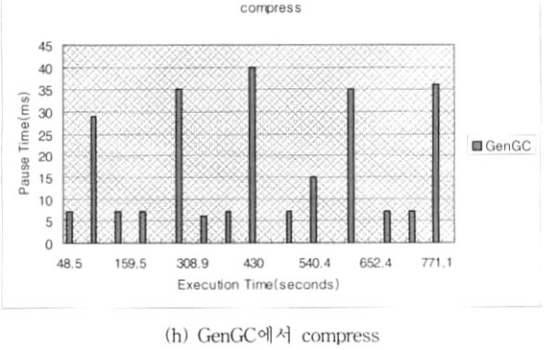
(e) GenRGC에서 javac



(f) GenGC에서 javac



(g) GenRGC에서 compress

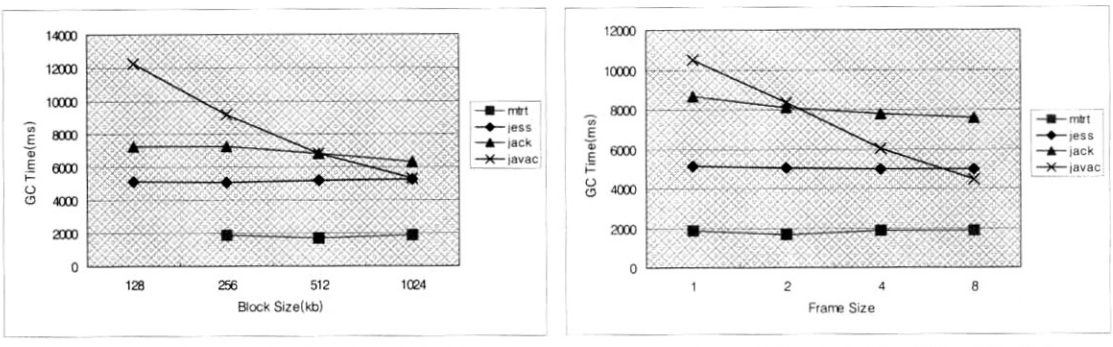


(h) GenGC에서 compress

(그림 8) 지연 시간 분포

의 크기를 1MB 이상으로 설정해야만 하기 때문이다. 임베디드 환경에서 블록의 크기를 1MB 이상으로 설정하는 것은 적합하지 않다.

(그림 9(a))는 GenRGC에서 블록의 크기를 변화시키며 각 프로그램을 수행 하였을 때 전체 가비지 컬렉션 수행 시간의 변화를 나타낸다. (그림 9(b))는 프레임의 크기를 변화시키며 각 프로그램을 수행 하였을 때 전체 가비지 컬렉션 수행 시간의 변화를 나타낸다. 그림에서 전체 가비지 컬렉션 수행 시간은 블록과 프레임의 크기를 증가시킴에 따라 감소하는 경향을 보인다. 이는 블록 또는 프레임의 크기가 증가



(a) 블록 크기에 따른 가비지 컬렉션 수행 시간 (b) 프레임 크기에 따른 가비지 컬렉션 수행 시간

(그림 9) 블록 및 프레임 크기에 따른 가비지 컬렉션 수행 시간

하면 가비지 컬렉션 수행 횟수가 감소하기 때문이다. 실험 1-2에서 논의한 바와 같이 가비지 컬렉션 수행 횟수의 감소는 전체 가비지 컬렉션 수행 시간을 감소시킨다.

jess의 경우는 블록의 크기나 프레임의 크기에 대해서 거의 영향을 받지 않는다. jess는 비교적 가비지가 많이 생성되는 프로그램이기 때문에 늙은 세대 가비지 컬렉션을 수행할 때, 대부분의 객체가 회수된다. 이 경우, 블록 혹은 프레임의 크기가 증가하더라도 가비지 컬렉션을 위해 복사되는 객체가 매우 적으므로 전체 가비지 컬렉션 수행 시간에는 별 영향을 미치지 않는다. 앞에서의 실험에서도 jess는 힙의 크기나 젊은 세대의 크기를 증가시키더라도 가비지 컬렉션 수행 시간은 크게 변화하지 않았다.

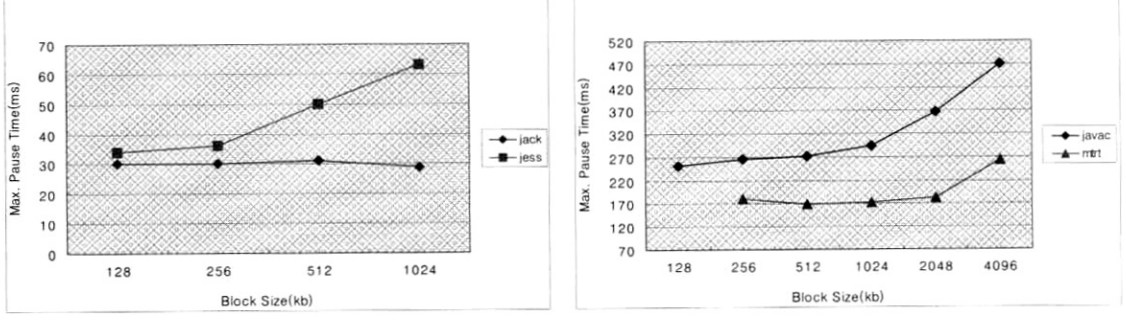
mtrt의 경우도 블록의 크기나 프레임의 크기에 대해서 거의 영향을 받지 않는다. mtrt는 256KB이상의 크기가 큰 객체를 생성하며 파일에서 그림을 읽어 들여 메모리에 계속 유지한다. 그 결과 하나의 블록 및 프레임에 대한 가비지 컬렉션만으로 필요한 공간을 확보하지 못하여 다른 블록 및 프레임으로 연속적인 가비지 컬렉션이 발생하게 된다. 이 경우, 블록 및 프레임의 크기가 증가하더라도 가비지 컬렉션 할 영역의 크기가 비슷하게 되어 전체 가비지 컬렉션 수행 시간이 일정하게 나타난다.

실험 2-2. 블록의 크기에 따른 최대 지연 시간

(그림 10)은 힙을 기본 크기로 설정하고 블록의 크기를 변화 시켰을 때, 늙은 세대 가비지 컬렉션에 의한 응용 프로그램의 최대 지연 시간을 나타낸다. 프로그램에 따라 실험을 통해 측정된 최대 지연 시간 값이 크게 차이가 나므로 최대 지연 시간의 크기에 따라 3개의 그래프로 나누어 나타냈다. 그림에서 블록의 크기가 증가하면 GenRGC에 의한 최대 지연 시간도 증가한다. 이는 블록의 크기가 증가하면 가비지 컬렉션 해야 할 대상 영역의 크기가 증가하기 때문이다. 그러나 jack의 경우 최대 지연 시간이 블록의 크기에 영향을 받지 않았다. 실험 1-1에서 설명한 바와 같이 jack의 경우는 가비지 컬렉션 하고자 하는 영역의 도달 가능한 객체를 식별하는데 많은 시간을 소요한다. 블록의 크기가 증가하면 검색의 대상이 되는 영역의 크기가 감소하기 때문에 도달 가능한 객체를 식별하는 시간이 감소한다. 그 결과, jack의 경우 가비지 컬렉션 해야 할 영역의 크기 증가로 인한 최대 지연 시간 증가폭과 도달 가능한 객체를 식별하는데 소요되는 시간의 감소폭이 비슷하여 최대 지연 시간이 블록 크기의 영향을 받지 않는 것이다. 본 실험에서는 블록의 크기를 조절함으로써 최대 지연 시간의 조절이 가능함을 보였다. 이것은 GenRGC가 블록의 크기 조정을 통하여 응용 프로그램에서 원하는 실시간 요구 사항을 만족시킬 수 있음을 의미하는 것이다.

실험 2-3. 저장 공간 사용량

GenRGC는 2단계 쓰기 장벽을 위한 추가적인 저장 공간



(그림 10) 블록 크기에 따른 최대 지연 시간

〈표 2〉 힙에서 GenRGC의 추가 공간이 차지하는 비율

프로그램 명	힙의 기본 크기와 설정된 힙의 크기의 비				
	1	1.5	2	2.5	3
_228_jack	2.167 %	2.167 %	2.167 %	2.167 %	2.170 %
_202_jess	2.167 %	2.167 %	2.167 %	2.167 %	2.170 %
_227_mtrt	2.158 %	2.159 %	2.159 %	2.161 %	2.161 %
_213_javac	2.173 %	2.176 %	1.800 %	2.185 %	1.808 %
_209_db	2.150 %	2.150 %	2.150 %	2.150 %	2.150 %
_201_compress	3.034 %	2.751 %	2.605 %	2.516 %	2.456 %

이 요구된다. 메모리에 제한을 가지는 임베디드 환경에서는 저장 공간 사용량이 중요한 성능 평가 척도가 된다. 본 실험에는 GenRGC의 추가 요구 공간의 크기를 측정하고 평가한다.

<표 2>는 힙의 크기를 기본 크기의 1, 1.5, 2, 2.5, 3배로 증가시키면서 각 프로그램을 수행하였을 때, GenRGC의 2단계 쓰기 장벽을 위해 추가 소요된 공간의 비율을 나타낸다. <표 2>에 나타난 바와 같이 모든 실험 영역에서 GenRGC의 추가 저장 공간은 항상 전체 힙의 3% 이내였다. 이 공간은 임베디드 환경의 제한적인 메모리 내에서 충분히 수용할 수 있을 정도로 작은 것이다.

4. 결 론

본 논문에서는 실험을 통하여 실제 CVM 환경에서 GenGC와 GenRGC의 성능을 비교, 평가하고 각 방법의 특성을 분석하였다. GenRGC는 [3]에서 아이디어를 제안하고 성능 평가를 수행 하였지만, GenRGC의 특성만 제안할 뿐 성능 평가의 비교 대상인 GenGC의 특성에 대한 분석이 없었다. 또한 간단한 성능 평가만 수행하여 임베디드 환경에 적합한지를 보이는데 부족 하였다. 따라서 본 논문에서는 GenGC와 GenRGC의 특성을 설명하고, 더 많은 벤치마크 프로그램으로 GenRGC가 지원하는 파라미터를 다양하게 변경하면서 다양한 실험을 수행하였다.

첫 번째 실험에서는, GenGC와 GenRGC의 성능을 비교하기 위하여 힙의 크기와 젊은 세대 영역의 크기를 변화시키면서 가비지 컬렉션 수행 시간 및 지연 시간을 비교하였다. 실험 결과 힙의 크기와 젊은 세대 영역의 크기가 증가할수록 감소하는 경향을 보였다. 특히, 대부분의 응용에서 GenRGC가 GenGC 보다 가비지 컬렉션 수행 시간이 작아 더 우수한 성능을 보였다. 또한, 지연 시간의 측면에서도 GenRGC의 최대 지연 시간이 GenGC의 최대 지연 시간보다 작은 것으로 나타났으며, 고른 지연 시간 분포를 보임으로써 임베디드 환경 및 실시간 요구에 더욱 적합한 방법임을 보였다.

두 번째 실험에서는, GenRGC의 성능을 보다 세밀하게 분석하기 위하여 힙의 구성 요소 중 블록과 프레임의 크기를 변화시키면서 가비지 컬렉션 수행 시간 및 최대 지연 시간을 측정하였다. 실험 결과 블록과 프레임의 크기가 증가할수록 가비지 컬렉션 수행 시간은 감소하고, 최대 지연 시간은 증가하는 경향을 보였다. 이는 응용에 따라 블록 및 프레임의 크기를 조절함으로써 가비지 컬렉션 수행 시간과 최대 지연 시간을 조절할 수 있음을 의미한다. 또한, GenRGC를 사용하기 위하여 필요한 저장 공간의 크기를 측정하였다. 실험 결과 GenRGC를 위하여 추가로 소요되는 저장 공간은 전체 힙의 3% 이하로서 GenRGC가 제한된 메모리를 가지는 임베디드 환경에서 충분히 동작 가능하다는 것을 보였다.

참 고 문 헌

[1] D. Barrett and B. Zorn, "Using Lifetime Predictors to Improve Memory Allocation Performance," In Proc. Int'l. Conf. on Programming Languages Design and Implementation, SIGPLAN PLDI, Vol. 24, No. 7, pp. 187-196, 1993.

[2] S. Blackburn, P. Cheng, and K. McKinley, "Myths and Reality: The Performance Impact of Garbage Collection," In *Proc. Int'l. Conf. on Measurement and Modeling of Computer Systems*, SIGMETRICS, pp. 25-36, 2004.

[3] C. Cha et al., "Garbage Collection in an Embedded Java Virtual Machine," In *Proc. Int'l. Conf. on Knowledge-Based Intelligent Information & Engineering Systems*, KES, pp.443-450, 2006.

[4] G. Chen et al., "Tuning Garbage Collection in an Embedded Java Environment," In *Proc. Int'l. Symp. on High-Performance Computer Architecture*, HPCA, pp.92-103, 2002.

[5] C. Cheney, "A Non-Recursive List Compacting Algorithm," *Communications of the ACM*, Vol. 13, No. 11, pp.677-678, 1970.

[6] J. Cohen and A. Nicolau, "Comparison of compacting algorithms for garbage collection," *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 4, pp.532 - 553, 1983.

[7] S. Dieckmann and U. H"olzle, "A Study of Allocation Behavior of the SPECjvm98 Java Benchmarks," In *Proc. European Conf. on Object-Oriented Programming*, ECOOP, pp.92-115, 1999.

[8] L. Hansen and W. Clinger, "An Experimental Study of Renewal-Older-First Garbage Collection," In *Proc. Int'l. Conf. on Functional Programming*, ACM SIGPLAN

ICFP, pp.247-258, 2002.
[9] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons, 1996.
[10] H. Lieberman and C. Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects," *Communications of the ACM*, Vol. 26, No. 6, pp.419-429, 1983.
[11] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1996.
[12] Standard Performance Evaluation Corporation, SPECjvm98 Documentation, Release 1.04 Edition, 2001.
[13] Sun Microsystems, Java2 Platform Micro Edition Technologies, http://java.sun.com/javame/technologies/index.jsp, 2006.
[14] Sun Microsystems, Connected Device(CDC) and the Foundation Profile, http://java.sun.com/products/cdc/wp/CDCwp.pdf, 2006.
[15] D. Stefanović, Properties of Age-Based Automatic Memory Reclamation Algorithms, Ph. D. Dissertation, University of Massachusetts, Amherst, MA, 1999.
[16] D. Stefanović et al., "Age-based Garbage Collection," In *Proc. Int'l. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, ACM SIGPLAN OOPSLA, pp.370-381, 1999.
[17] D. Stefanović et al., "Older-First Garbage Collection in Practice: Evaluation in a Java Virtual Machine," *ACM SIGPLAN Notices*, Vol. 38, No. 2, pp.25-36, 2003.
[18] D. Ungar, "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm," *ACM SIGPLAN Notices*, Vol. 19, No. 5, pp.157-167, 1984.
[19] B. Zorn, Comparative Performance Evaluation of Garbage Collection Algorithms, Ph. D. Dissertation, University of California at Berkeley, 1989.
[20] B. Zorn, Barrier Methods for Garbage Collection, Technical Report CU-CS-494-90, University of Colorado, 1990.

차 창 일

e-mail : charose@zion.hanyang.ac.kr
2005년 2월 성결대학교 컴퓨터공학과 학사
2007년 2월 한양대학교 대학원 전자컴퓨터 통신공학과 석사
2007년 3월~현 재 (주)에이록스 연구원
관심분야 : 데이터베이스 시스템, JVM, 메모리 관리

김 상 욱

e-mail : wook@hanyang.ac.kr
1989년 2월 서울대학교 컴퓨터공학과 학사
1991년 2월 한국과학기술원 전산학과 석사
1994년 2월 한국과학기술원 전산학과 박사
1991년 7월~8월 미국 Stanford University, Computer Science Department 방문 연구원
1994년 2월~1995년 2월 KAIST 정보전자연구소 전문연구원
1999년 8월~2000년 8월 미국 IBM T.J. Watson Research Center Post-Doc.
1995년 3월~2000년 8월 강원대학교 컴퓨터정보통신공학부 부교수
2003년 3월~현 재 한양대학교 정보통신대학 정보통신학부 교수
관심분야 : 데이터베이스 시스템, 저장 시스템, 트랜잭션 관리, 데이터 마이닝, 멀티미디어 정보 검색, 공간 데이터베이스/GIS, 주기억장치 데이터베이스, 이동 객체 데이터베이스/텔레매틱스, 사회 연결망 분석, 웹 데이터 분석

장 지 웅

e-mail : jwchang@kpu.ac.kr
1993년 2월 연세대학교 전산과학과 학사
1995년 2월 한국과학기술원 전산학과 석사
2001년 8월 한국과학기술원 전자전산학과 박사
2001년 9월~2002년 2월 한국과학기술원 첨단정보기술 연구센터 연구원
2004년 3월~현재 한국산업기술대학교 게임공학과 조교수
관심분야 : 데이터베이스 시스템, 주기억장치 데이터베이스, 저장시스템, 동시성 제어, 공간 데이터베이스, 게임 데이터 관리