동적 순환 메모리 할당 기법을 이용한 메모리 누수 검출

(Memory Leak Detection Using Adaptive Cyclic Memory Allocation)

임 우 섭 † 한 환 수 †† 이 상 원 ††
(Woosup Lim) (Hwansoo Han) (Sang-Won Lee)

요 약 메모리 누수 검출을 위한 많은 도구들이 존재한다. 하지만 큰 시간적, 공간적 오버헤드로 인해 규모가 큰 제품의 개발자들은 사용을 꺼리게 된다. 이에 우리는 개발자들이 자신이 개발한 모듈만을 대상으로 유닛 테스트 시에 메모리 누수를 검출할 수 있는 기법을 고안하였다. 우리는 고정 크기 순환적 메모리 할당 기법을 우리의 목적에 맞게 확장함으로써 이것을 달성하였으며 우리의 기법을 평가하기 위해서, 간단한 데이터베이스 관리 시스템을 구현하여 그 중 일부 모듈을 대상으로 테스트 하였다. 실험 결과 우리 기법은 유닛 테스트 시에 적은 시간적, 공간적 오버헤드와 거짓 검출을 가졌다.
키워드 : 메모리 누수, 동적 분석, 순환적 메모리 할당

Abstract There are many memory leak detection tools. However, programmers, who develop very large programs, tend to avoid testing their programs with memory leak detection tools since these tools require runtime and space overheads. Thus, we present a memory leak detection technique which enables programmers to test their modules in their unit test phase with low overheads. To achieve this goal, we extend the existing cyclic memory allocation technique and evaluate our memory leak detection technique on a tiny DBMS. In our experiments, we find our tool has reasonably low runtime and space overheads and it reports only a small number of false positives.
Key words : Memory leaks, Dynamic analysis, Cyclic memory allocation

1. 서 론

소프트웨어 개발에서 가장 어려운 부분은 결함을 발견하고 고치는 일이다. 결함은 시스템 실패로 이어지기 때문에 소프트웨어 개발자는 결함을 줄여 신뢰성 있는 프로그램을 만들기 위해 부단한 노력을 경주하고 있다. 이러한 결함 중에 개발자를 가장 괴롭히는 것은 메모리 누수이다. 이 결함은 개발자의 작은 실수로 일어나서 시스템 전체를 실패시키는 막대한 결과를 일으키지만 일반적인 디버깅 툴을 통해서 검출되지 않아 Valgrind와 같은 부가적인 툴을 통해 검출해야 한다[1]. 하지만 부가적인 툴이 갖고 있는 큰 시간적, 공간적 오버헤드로 인해 개발자가 규모가 큰 제품을 테스트할 때는 많은 시간과 노력을 필요로 한다. 더욱이 제품을 새로이 개발한 것이 아니라 기존 제품에다 일부 새로운 기능의 모듈만 추가한 경우라면 전체 제품에 대한 메모리 누수 검출 테스트는 시간이 오래 걸릴 뿐만 아니라 거짓 검출로 인해 효율성이 감소될 것이다. 이에 우리는 개발자가 전체 프로그램이 아닌 자신이 개발한 모듈만을 대상으로 메모리 누수를 검사할 수 있는 메모리 누수 검출 기법을 고안하였다. 이것을 통해 개발자는 개발한 모듈

· 본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업(NIPA-2010-(C1090-1021-0008))과 서울시의 지원으로 수행한 서울시 산학연 협력사업(PA090903), 교육과학기술부 재원으로 한국연구재단의 지원을 받아 실시한 중견연구자 지원사업(NRF 2009-0084870)의 연구결과입니다.

† 학생회원 : 성균관대학교 임베디드소프트웨어학과
fairy34@skku.edu
†† 종신회원 : 성균관대학교 정보통신공학부 교수
hhan@skku.edu
swlee@skku.edu

논문접수 : 2010년 4월 16일
심사완료 : 2010년 8월 3일

Copyright©2010 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.
정보과학회논문지: 소프트웨어 및 응용 제37권 제10호(2010.10)

에 대한 테스트 시간을 줄일 수 있을 것이다. 이를 위해서 우리는 고정 크기 순환적 메모리 할당 기법을 우리의 목적에 맞게 조정하였다[2]. 이로 인해 두 가지 이득을 얻을 수 있었는데, 첫 번째는 다른 모듈의 메모리 누수에 관계없이 자신이 원하는 모듈에 대한 테스트를 지속할 수 있다는 것이다. 두 번째는 이미 실행되고 있는 서버 프로그램을 대상으로 서비스를 계속 제공하면서 테스트할 수 있다는 것이다. 이는 메모리 누수가 존재하더라도 전체 시스템이 실패하지 않는 기존 기법의 특성 때문에 가능한 것이다. 그러나 기존 기법은 사용 중인 메모리를 재할당할 수 있는 심각한 위험을 가지고 있다. 이 기법은 할당 위치(Allocation site)[1]마다 실행 전에 몇 개의 객체를 가진 버퍼가 필요한지 계산하고, 계산된 개수의 객체로 이루어진 버퍼를 선할당하여 실행 시에는 해당 버퍼가 가진 객체를 순환적으로 할당해 준다. 만약 미리 계산된 버퍼의 객체 개수가 작게 설정된 경우에는 사용 중인 메모리 객체가 버퍼로 반환되면서 그 객체의 데이터가 손실된다. 따라서 우리는 강제적인 메모리 객체 반환을 없애고 메모리 누수 검출을 통해 메모리 반환이 일어나도록 함으로써 이러한 문제점을 개선하였다. 이런 개선을 통해서 우리가 기여한 점은 기존 기법인 고정 크기 순환적 메모리 할당 기법을 더 안전하게 만들었으며, 실행 전에 미리 필요한 버퍼의 객체 개수를 구할 필요가 없도록 한 것이다. 또한 원하는 모듈에만 적용할 수 있도록 하여 프로그래머가 유닛 테스트 시에 활용할 수 있도록 하였다.

본 논문의 구조는 다음과 같다. 2장은 고정 크기 순환적 메모리 할당 기법의 한계, 그리고 우리가 제안하는 기법과 그 이점에 대해서 서술하고 3장은 우리가 메모리 누수 검출을 어떻게 하였는지 설명한다. 4장에서 실험을 통해서 우리 기법의 시간적, 공간적 오버헤드와 검출 능력을 평가하고, 5장에서는 관련 연구가 어떻게 진행되고 있는지 서술한다. 마지막으로 6장에서 우리의 연구를 평가하며 결론을 맺는다.

2. 순환적 메모리 할당 기법

본 장에서는 고정 크기 순환적 메모리 할당 기법[2]에 대해 구체적으로 설명하고, 이 기법이 가진 한계를 우리가 어떻게 극복했는지 서술하겠다. 고정 크기 순환적 메모리 할당 기법은 프로그램 실행 동안 사용할 메모리의 양을 미리 할당 받아 사용함으로써 적은 오버헤드를 가지고 메모리 누수를 제거할 수 있는 방법이다. 하지만 일반적인 메모리 누수 검출 기법과는 달리, 검출된 메모리 누수를 프로그래머에게 알려주는 것이 아니라 메모리 누수가 있어도 프로그램이 적절히 처리하여 지속적으로 실행될 수 있도록 하는 결함 내성(Fault tolerance)에 속한다.

2.1 고정 크기 순환적 메모리 할당 기법

고정 크기 순환적 메모리 할당 기법은 다음과 같은 단계를 통해서 메모리 누수를 제거한다.

① 버퍼의 객체 개수 결정 : 이 단계는 할당 위치마다 프로그램 실행 시에 몇 개의 객체를 가진 버퍼가 필요한지 결정하는 단계이다. 이를 위해서 테스트 실행을 실시한다. 즉, 테스트 입력을 통해 할당 위치에서 할당된 메모리 객체 중에 실제 프로그램 실행 동안 동시에 사용되는 객체의 수를 확인하여 그 최대값을 버퍼의 객체 개수로 결정한다.

② 버퍼 선할당 : 그림 1과 같이 앞에서 구한 버퍼의 객체 개수만큼 할당 위치마다 미리 버퍼를 할당한다. 따라서 실제 실행 간 메모리 할당을 요청하면 새로운 메모리 공간이 아닌 미리 준비된 버퍼의 객체를 할당받게 된다.

③ 순환적 할당 및 강제적 반환 : 실제 프로그램 실행 간 할당 위치마다 메모리 할당을 요청하게 되면 이전 요청에서 할당했던 버퍼 객체의 바로 다음 객체를 할당해준다. 즉 그림 1에서 1번 객체까지 할당이 된 상태이면 2번 객체를 할당해준다. 만약 버퍼의 제일 마지막 객체까지 할당되었다면 버퍼의 제일 처음으로 돌아가 맨 처음 객체를 할당해준다. 그리고 할당과 함께 반환이 일어나게 되는데 할당된 바로 다음 객체는 강제적으로 버퍼에 반환되고 다음 할당 요청 시 재할당된다.

고정 크기 순환적 메모리 할당 기법은 실제 프로그램 실행 시에 메모리 누수 검출을 위한 동적 분석이 필요 없기 때문에 공간적, 시간적 오버헤드가 매우 적다. 또한 매우 간단하여 누구나 쉽게 구현하고 적용해 볼 수

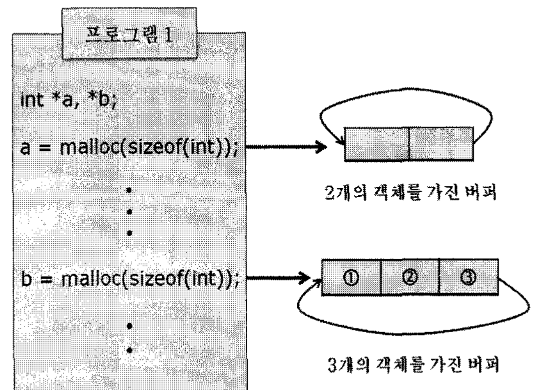

그림 1 고정 크기 순환적 메모리 할당 기법

1) C 표준 라이브러리 함수인 malloc함수가 호출되는 부분

있다는 장점을 갖는다. 그리고 할당 시에 강제적인 메모리 반환이 일어나므로 메모리 누수가 존재할 수 없고 따라서 시스템이 메모리 부족으로 인해 실패하는 일은 없다.

그러나 이 기법이 가진 잠재적인 문제는 강제적으로 버퍼의 객체를 반환하기 때문에 실제 사용 중인 메모리를 반환할 가능성이 있다는 것이다. 이러한 경우는 입력에 따라 필요한 메모리 객체의 수가 달라지는 경우에 쉽게 일어날 수 있는데 그 대표적인 경우가 연결 리스트, 트리, 해쉬 테이블이다. 이 3가지 자료 구조의 특징은 자료구조에 필요한 메모리 객체의 개수가 실행 시에 동적으로 정해진다는 것이다. 따라서 테스트 실행 시보다 큰 입력이 들어오거나 다른 실행 흐름을 가질 경우 실제 사용되고 있는 메모리 객체가 버퍼 객체의 부족으로 버퍼로 반환될 것이고 그 버퍼에 담겨 있던 데이터는 프로그래머의 의도에 맞지 않게 손실될 것이다. 예를 들어 버켓 오버플로우 처리를 위해 체이닝 기법을 적용한 해쉬 테이블을 생각해보자. 강제적인 반환으로 인해 사용 중인 해쉬 엔트리가 버퍼로 반환 된다면, 버켓의 체인 일부가 손실될 것이고 해당 버켓은 사용될 수 없을 것이다. Nguyen과 Rinard는 이러한 문제가 일어나게 되면 프로그램 내의 예외 처리를 통해 다시 시도하거나 해당 기능의 모듈만 실패하기 때문에 메모리 누수의 결과 즉, 메모리 부족으로 인해 시스템 전체가 멈추는 것보다는 훨씬 낫다고 말하고 있다. 그러나 해당 기능의 모듈이 만약 프로그램의 핵심 기능이었을 때, 프로그램은 정상적인 결과를 출력하지 못할 것이고, 사용자나 프로그래머는 정확한 원인을 찾기 힘들 것이다.

2.2 동적 순환 메모리 할당 기법

따라서 우리는 강제적인 메모리 반환을 하지 않는 기법을 제안한다. 우리는 고정 크기 순환적 메모리 할당 기법에서 명시적인 반환 함수[2)]를 무시한 것과는 다르게 명시적인 반환 함수가 호출될 때만 해당 메모리 객체를 버퍼에 반환한다. 즉, 프로그래머의 의도를 반영할 수 있게 한 것이다. 그러나 프로그래머가 명시적으로 반환하는 객체만 버퍼로 반환하게 된다면 여전히 메모리 누수 문제가 존재할 것이고 반복될 경우 메모리 부족으로 시스템이 멈추게 될 것이다. 그래서 우리는 메모리 누수 검출 기법을 추가로 적용하여 검출된 객체에 대해서는 프로그래머에게 알리고 버퍼에 반환하도록 했다. 구체적인 동작 원리는 다음과 같다.

① 가변 크기 순환적 할당 : 할당 위치에서 할당을 요청하면 버퍼 내에 할당되지 않은 객체가 있는지 찾아 그 객체를 할당해 준다. 만약 없다면 버퍼 객체의 개수를 하나 늘려 새로운 객체를 만들어 할당해준다. 즉, 버퍼는 처음 할당 시 한 개의 객체만을 가지다가 추가적인 할당 요청 시마다 빈 객체가 없다면 새로운 객체를 추가로 생성한다.

② 명시적인 버퍼 반환 : 프로그래머가 반환 함수를 통해 명시적으로 객체를 버퍼에 반환하게 되면 반환되었음을 표시하여 다음 할당 시 그 객체가 할당될 수 있도록 한다.

③ 메모리 누수 검출 : 만약 명시적인 반환만 있다면 메모리 누수가 일어난 할당 위치의 버퍼는 계속해서 증가하기만 할 것이고, 메모리가 부족해 시스템이 멈추게 될 것이다. 따라서 우리는 메모리 누수 검출을 통해 메모리 누수로 판단되는 객체는 버퍼에 반환하고 프로그래머에게 알려준다.

이러한 수정을 통해서 다음과 같은 이점을 얻을 수 있다.

- 고정 크기 순환적 메모리 할당 기법보다 안전성이 높다. 앞에서 살펴본 바와 같이 기존 기법은 특정 자료 구조에 대해서 실제 사용 중인 데이터가 사라질 수 있다는 문제점이 있었다. 하지만 우리 기법에서는 메모리 누수로 판단되지 않는 한 사용 중인 데이터가 사라지는 일은 발생하지 않는다.
- 테스트 실행이 필요 없다. 사실 테스트 실행을 위해서 샘플 데이터를 선정하고 그것을 입력으로 하여 필요한 버퍼 객체의 개수를 구한다는 것은 프로그래머 입장에서 어려운 일이다. 하지만 우리가 제안한 기법에서는 버퍼 객체가 동적으로 늘어나고 메모리 누수 검출을 통해 자동으로 적정 선에서 관리되기 때문에 적용하기가 쉽다.
- 명시적으로 메모리 누수가 일어나는 곳을 알 수 있다. 고정 크기 순환적 메모리 할당 기법은 메모리 누수에 대해 프로그래머에게 어떠한 보고도 하지 않는다. 그러나 우리 기법은 실제 메모리 누수로 추정되는 부분에 대해 알려주므로 프로그래머는 프로그램을 검토하여 메모리 누수를 제거할 수 있다. 이것은 신뢰성을 높여준다.

3. 메모리 누수 검출

메모리 누수를 검출할 때 가장 중요한 문제는 어떤 객체를 메모리 누수로 볼 것인가에 대한 기준을 정하는 것이다. 우리는 다음과 같은 직관적인 기준을 활용하여 메모리 누수를 판단한다[3].

누수 검출 기준 : 할당된 메모리 객체가 오랫동안 접근되지 않는다면 메모리 누수로 판단한다.

이 기준의 정확성을 높이기 위해서는 '오랫동안'을 어떻게 판단하느냐가 가장 중요한 문제일 것이다. SWAT[3]

2) C 표준 라이브러리 함수인 free함수

는 1,000,000,000을 기준으로 할 때 가장 적은 거짓 검출을 보였으나 우리는 전체 프로그램이 아닌 모듈만을 대상으로 하기 때문에 더 적은 횟수의 기준이 필요할 것이고 실험을 통해서 다시 평가하였다. 여기서 1,000,000,000을 기준으로 삼겠다는 것은 프로그램 실행 간 메모리 접근이 1,000,000,000번 일어나는 동안 한 번도 접근되지 않은 객체를 메모리 누수로 보겠다는 의미이다. 시간을 기준으로 하지 않는 이유는 프로그램이 사용자와 상호작용을 통해 작동하는 경우 오랫동안 사용자의 입력이 없으면 모든 객체가 메모리 누수로 판단될 수 있기 때문이다. 각 메모리 객체에 대한 접근 정보를 유지하기 위해서는 힙 영역을 모델링 하는 것이 필요하다. 즉, 객체의 주소를 이용해 그 객체에 대한 정보를 효율적으로 찾을 수 있는 자료구조가 필요하다. 기존 논문은 이것을 위해 이진 트리를 사용했지만 우리는 레드-블랙 트리를 사용했다.

3.1 힙 모델링

우리가 세운 기준을 통해 메모리 누수를 판단하기 위해서는 메모리 객체에 대한 추가적인 정보를 남겨야 한다. 즉, 할당되는 모든 객체에 대해서 주소, 크기, 접근 시간 등의 정보를 유지해야 한다. 일반적인 프로그램에서는 수없이 많은 객체들이 생성될 것이고 그 많은 객체 중에 특정 객체를 찾아 정보를 갱신하는 것은 비용이 크다. 따라서 특정 객체에 대한 정보를 쉽게 찾을 수 있도록 자료 구조를 유지하는 것이 힙 모델의 목적이다.

힙 모델을 위해서 몇 가지 자료구조를 사용할 수 있는데 먼저 검색에 있어서 가장 빠른 성능을 보이는 해쉬 테이블이다. 하지만 해쉬 테이블은 객체의 크기를 고려할 수 없다는 제한을 갖는다. 예를 들어 그림 2와 같은 경우 객체의 시작 주소인 0×00만을 이용해 해쉬 테이블을 구성한다면 구조체 멤버인 a나 bit를 접근할 경우 같은 객체임에도 불구하고 해당 엔트리를 찾을 수 없다. 왜냐하면 bit를 접근할 경우 동적 분석을 통해서 얻게 되는 주소는 0×05일 것이고 시작 주소인 0×00에 대해서만 해쉬 테이블이 구성되어 있으므로 다른 객체로 판단되기 때문이다. 이것을 회피하기 위해서는 객체의 시작 주소만이 아닌 객체에 해당하는 모든 주소를 해쉬 테이블로 구성해야 하는데 이는 많은 충돌을 일으켜 해쉬 테이블의 성능을 저하시킨다.

다음으로 SWAT[3]가 사용한 이진 트리이다. 이것은 해당 객체의 주소 비트를 사용하여 이진 트리를 구성한

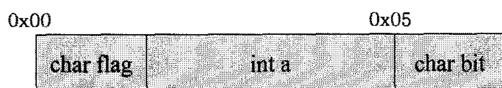

그림 2 여러 멤버들을 가진 구조체

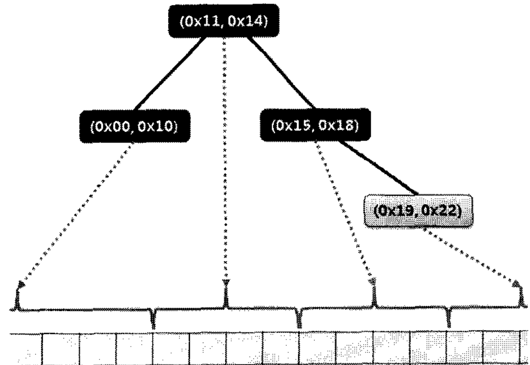

그림 3 레드-블랙 트리를 통해 힙을 모델링한 예

것이다. 중간 노드를 사이즈에 따라 삭제함으로써 앞의 해쉬 테이블이 가졌던 문제를 해결해 주지만 객체 주소의 가장 높은 비트부터 0/1을 판단해서 내려가므로 32비트 컴퓨터의 경우 32의 깊이를 가지게 된다. 이는 많은 검색 비용을 유발한다. 따라서 우리는 그림 3과 같이 레드-블랙 트리를 이용하여 깊이를 최소화할 수 있도록 하였다. 물론 삽입, 삭제에 대해서는 이진 트리보다 큰 비용이 들지만 일반적으로 메모리 생성, 해제보다 메모리 접근이 더 많이 일어나므로 성능의 향상을 기대할 수 있다. 그림 3에서 보는 바와 같이 레드-블랙 트리의 각 노드는 키 값으로 시작 주소와 끝 주소의 범위 값을 갖게 된다. 따라서 우리가 주소로 해당 객체를 찾고자 하면 해당 노드의 키에 해당하는지 확인하고 시작 주소보다 작다면 왼쪽으로, 끝 주소보다 크다면 오른쪽으로 이동하면서 트리를 탐색하게 된다. 그림에는 나와 있지 않지만 각 노드는 노드와 대응하는 객체에 대한 정보 즉, 할당 위치, 접근 시간이 기록되어 있는 구조체에 대한 포인터를 가지고 있다. 따라서 트리 탐색을 통해 노드를 검색하면 대응하는 객체에 대한 정보를 갱신하거나 삽입, 삭제할 수 있다.

3.2 유닛 테스트를 위한 변형

유닛 테스트 시 메모리 누수 검출의 정확성을 높이기 위해서는 할당 위치에서 할당된 객체가 어느 모듈에서 사용되고 있는지를 정확히 알아야 한다. 이 정보를 가장 잘 아는 것은 모듈을 개발한 개발자이므로 우리는 사용자로부터 대상 함수들의 이름을 입력 받도록 하였다. 우리는 PIN을 사용하여 메모리 누수 검출을 구현 하였는데 함수 이름을 통해 해당 함수를 찾아서 동적 분석을 실시할 수 있는 인터페이스 함수가 제공되었다[4]. 따라서 우리는 이 인터페이스 함수를 통해 개발자가 입력한 이름의 함수들 안에서 일어나는 모든 메모리 접근 주소를 동적 분석을 통해 얻고 레드-블랙 트리를 통해 접근 시간 정보를 갱신하였다. 그리고 SWAT는 1,000,000,000

번을 기준으로 메모리 누수 검출을 하였을 때 가장 적은 거짓 검출을 보였지만 우리는 모듈만을 대상으로 하기 때문에 더 적은 기준 횟수를 사용해야 할 것으로 판단하였다. 실험을 통해 평가해보니 35,000번을 기준으로 하였을 때 가장 적은 거짓 검출을 보였다. 이 수치는 대상이 되는 모듈 안에서 일어나는 메모리 접근 횟수가 프로그램마다 다양하기 때문에 서로 다르게 적용되어야 할 것이다.

4. 실 험

우리가 제안하는 기법을 실험적으로 평가하기 위해서 우리는 메모리 누수 검출기를 구현하였다. 가변 크기 순환적 할당 방법은 GNU에서 제공하는 Malloc_hook 함수[5]를 이용하여 구현하였으며, 메모리 누수 검출은 PIN[4]을 이용하여 구현하였다. 우리는 유닛 테스트의 상황을 재현하기 위하여 Redbase정도의 데이터베이스 관리 시스템을 구현하여 시간적, 공간적 오버헤드와 메모리 누수 검출 능력을 평가했다[6]. 우리가 구현한 데이터베이스 관리 시스템은 크게 데이터 저장기와 질의 처리기로 구분된다. 그래서 데이터 저장기 내의 함수 두 개와 질의 처리기 내의 함수 한 개를 선택하여 각각의 컴포넌트를 대상으로 실험을 하였다. 데이터 저장기의 함수를 두 개 선택한 이유는 한 개 함수의 실행 시간 비율이 너무 작았기 때문이다. 우리가 대상으로 삼았던 함수들과 각 함수가 전체 프로그램 실행 시간에 대해 차지하는 비율은 표 1과 같다. 실험 환경은 인텔 i7 2.8GHz(4코어), 6GB의 메모리, 리눅스 커널 2.6(64비트)를 사용하였다. 또한 시간적, 공간적 오버헤드에 대한 실험 값은 다섯 번 반복해서 돌린 후 평균치를 사용하였다.

표 1 적용 함수가 전체 시간에서 차지하는 비율

상위 컴포넌트	적용 함수	전체 실행 시간에 대한 비율(%)	
데이터 저장기	insertTocatalog	0.017[1)]	0.026*
	deleteTocatalog	0.009[2)]	
질의 처리기	qpeSelectExec	0.06	

* 1)과 2)의 합

4.1 오버헤드

시간 오버헤드에 대한 결과가 표 2에 나타나 있다. 시간 측정은 프로그램 전체에 대해 리눅스의 time 명령을 통해서 실시하였고, 첫 번째 줄은 데이터 저장기의 함수를 대상으로 테스트 했을 때, 두 번째줄은 질의 처리기의 함수를 대상으로 테스트 했을 때, 시간 변화를 보여주며 평균적으로 약 180%정도 시간이 증가하였다. 여기서 180%정도 시간이 증가하였다는 것은 실행 시간 증가분을 원래 실행 시간으로 나눈 값으로 실행 시간이 원래 실행 시간의 2.8배가 되었다는 것은 의미한다. 우리는 시간 증가의 주요 원인을 알기 위해 가변 크기 순환적 할당만 적용했을 때와 가변 크기 순환적 할당을 메모리 누수 검출과 함께 적용했을 때의 시간 증가율을 따로 측정하였는데 표에서 보듯이 대부분의 시간 증가는 동적 분석을 이용한 메모리 누수 검출에서 소모되었다. 여기서 가변 크기 순환적 할당이란 것은 2.2장의 1번과 2번 단계만 적용한 것으로 고정 크기 순환적 메모리 할당 기법[2]과는 달리 버퍼의 크기를 고정하지 않고 필요한 만큼 유동적으로 늘려서 할당하는 것을 말한다. 그리고 데이터 저장기와 질의 처리기를 비교해 볼 때, 가변 크기 순환적 할당의 오버헤드가 늘어난 것을 볼 수 있는데 이는 빈 객체를 찾기 위해 버퍼를 순환하면서 검사하는데 이 때 버퍼의 객체 수가 많고 빈 객체가 없는 경우 검사에 소모되는 시간이 늘어나기 때문이다.

표 2 실행 시간 변화

단위 : 초

대상 모듈	정상 실행시간	가변 크기 순환적 할당 (증가율 %)	메모리 누수 검출 (증가율 %)
데이터 저장기	4.17	4.12 (-1.2)	11.92 (185.9)
질의 처리기	4.17	4.59 (10.1)	11.41 (173.6)

공간 오버헤드를 측정하기 위해 우리는 프로그램을 실행시키면서 리눅스 '/proc/pid/status'파일에 남아 있는 힙 영역의 크기를 0.1초 단위로 측정하면서 그 최대값을 구하였다. 표 3에서 보여지듯이, 약 47%정도의 메모리가 더 소모되었다. 이 역시 원래 메모리 소모량이 1이라고 한다면 우리 기법을 적용했을 때 1.47의 메모리가 소모되었음을 의미한다. 우리는 시간적 오버헤드 측정 시와 같이 공간 오버헤드의 주요 원인을 알기 위한 실험을 실시하였는데 표에서 보듯이 비슷한 비율을 보이거나 가변 크기 순환적 할당으로 인한 메모리 증가 비율이 조금 더 컸다. 동적 분석을 이용한 메모리 누수 검출로 인한 오버헤드는 메모리 객체 별 접근 정보 즉,

표 3 메모리 사용량 변화

단위 : MB

대상 모듈	정상 메모리 사용량	가변 크기 순환적 할당 (증가율 %)	메모리 누수 검출 (증가율 %)
데이터 저장기	2,575	3,344 (29.9)	3,807 (47.8)
질의 처리기	2,275	3,217 (41.4)	3,805 (67.3)

할당 위치, 접근 시간, 객체의 시작 및 끝 주소를 유지하기 위해 소모되는 공간이며 가변 크기 순환적 할당으로 인한 오버헤드는 버퍼 안에서 그 객체에 대한 정보 즉, 할당 여부, 할당 위치, 다음 버퍼 객체에 대한 포인터를 유지하기 위해 소모되는 공간이다.

4.2 메모리 누수 검출 능력

우리는 일부러 메모리 누수를 발생시켜 유닛 테스트 시 기준 횟수 별 검출 능력을 평가하였다. 메모리 누수를 발생시키기 위해서 테스트 대상이 되는 함수에 할당 위치 열 개를 삽입하고, 30,000~40,000까지 5,000단위로 기준 값을 변경하면서 참 검출(True positive)과 거짓 검출(False positive)를 측정하였다. 그 결과는 표 4, 5과 같다. 우선 표에서 보는 바와 같이 우리가 발생시킨 모든 메모리 누수를 찾았다. 그리고 기준 값이 30,000일 때나 35,000일 때는 결과의 차이를 보이지 않았지만 40,000일 때는 메모리 누수를 찾지 못했다. 이는 모듈 안에서 메모리 참조 횟수가 기준 값에 못 미치게 일어나서 모든 메모리에 대해서 아직은 누수가 아니라고 판단하기 때문이다. 그리고 질의 처리기에서는 거짓 검출이 두 개 있었다. 해당 객체는 메모리 검출의 대상이 되는 모듈에서 생성이 되어 검출 대상이 아닌 모듈에서 사용되기 때문에 거짓 검출 되었다. 그러나 이런 경우는 사용자가 객체가 사용되는 함수의 이름을 제대로 적어준다면 일어나지 않을 것이다. 실험에서는 적용 모듈에서 생성되어 그 모듈 안에서 사용되는 객체에 대해서만 관심을 갖고 검출했기 때문에 사용 모듈을 적어주지 않았고 거짓 검출이 일어나게 되었다.

표 4 질의 처리기에서 검출된 메모리 누수

검출 기준	총 개수	참 검출	거짓 검출
30000	12	10	2
35000	12	10	2
40000	0	0	0

표 5 데이터 저장기에서 검출된 메모리 누수

검출 기준	총 개수	참 검출	거짓 검출
30000	10	10	0
35000	10	10	0
40000	0	0	0

5. 관련 연구

메모리 누수 검출을 위해 다양한 기법들이 제안 되었다. 이것들은 크게 세 가지로 구분될 수 있으며, 그 중 첫 번째는 정적 분석이다. 프로그램을 직접 실행하지 않고 소스를 이용해서 프로그램을 분석하고 메모리 누수를 검출한다. 실행 시에 정보를 모으지 않기 때문에 오버헤드는 적으나 부정확한 분석으로 인한 거짓 검출과 거짓 미검출이 많다. 대표적인 기법으로 Clouseau[7]는 소유 제약을 추론하고 그 위반을 찾아 누수를 찾으며, Xie와 Aiken[8]은 탈출 분석을 기반으로 하여 메모리 누수를 찾는다. 또한, Cherem은 할당 위치에서 반환 위치까지 프로그램 그래프를 통해 흐름을 분석하는 방법을 제안하였다[9].

두 번째는 동적 분석이다. 이것은 정적 분석과 반대로 실행 시에 메모리에 관한 정보를 모으고 분석하여 메모리 누수를 검출한다. 상대적으로 거짓 검출과 거짓 미검출은 적으나 프로그램 실행 정보를 얻기 위한 부가적인 시간과 공간으로 인해 오버헤드가 크다. 이에 속하는 기법 중 대표적인 것이 SWAT[3]와 Hound[10]이며, 동적 분석으로 인한 오버헤드를 줄이고자 제안되었다. 이들 기법은 우리 기법과 같이 메모리 접근을 추적하여 오랫동안 접근되지 않는 메모리를 메모리 누수로 검출하는 기준을 가지고 있다. 이 중 SWAT는 코드를 샘플링하는 기법으로 만약 샘플 비율이 10%이고 10번 실행하는 코드가 있다고 할 때, 10번 중 1번만 동적 분석을 실시하여 메모리 접근 정보를 얻는 기법이다. 9번은 동적 분석을 실시하지 않기 때문에 샘플 비율을 줄이면 적은 오버헤드를 갖는다. 하지만 샘플 비율만큼 정보를 적게 얻기 때문에 실제로는 지속적으로 접근되고 있는 메모리 객체를 메모리 누수로 보고하게 되어 거짓 검출이 많다. 반면, Hound는 mprotect 함수를 이용해서 페이지 단위로 메모리 누수를 검출하는 기법이다. mprotect 함수는 페이지 보호 레벨을 설정하는 함수로써 해당 페이지에 쓰기, 읽기를 금지시킬 수 있다. 프로그램이 쓰기, 읽기가 금지된 페이지에 접근을 요청하면 SIGSEGV 신호가 발생되고, 따라서 해당 페이지가 접근되고 있음을 알 수 있다. 페이지 단위로 접근 여부를 판단하므로 한 페이지에 누수인 메모리 객체와 누수가 아닌 메모리 객체가 섞여 있을 경우 누수인 메모리 객체를 찾지 못하는 거짓 미검출이 있을 수 있다.

Valgrind[1], Purify[11], RADAR[12]와 같은 툴은 도달 가능성을 분석하여 메모리 누수를 검출한다. 프로그램 실행 동안 포인터 정보를 저장하고, 해당 정보를 이용해 포인터를 통해 도달 불가능한 메모리 객체를 찾아 메모리 누수로 보고한다. 따라서 도달 가능한 메모리 누수는 찾을 수 없다는 단점을 가진다.

세 번째는 결함 내성 컴퓨팅이다. 고정 크기 순환적 할당 기법[2]이 이 결함 내성 컴퓨팅의 일종으로 메모리 누수가 있더라도 시스템이 실패하지 않도록 해준다. 따라서 서비스 중인 서버 프로그램에 대한 테스트를 실시할 때 적합한 기법으로 실제 입력에 대한 테스트를 가능하게 해준다. 이 기법 중에 Melt[13]와 LeakSurvivor

[14]는 오래된 메모리 객체를 고립시키고 압축함으로써 메모리 누수로 인한 영향을 최소화한다.

우리가 제안한 기법은 동적 분석과 결함 내성 컴퓨팅의 장점을 결합하여 실시간으로 서비스가 되고 있는 프로그램의 일부 모듈에 대해서 시스템 실패 없이 테스트를 실시할 수 있도록 하였으며 동적 분석을 사용함으로써 명시적으로 메모리 누수를 검출할 수 있도록 하였다.

6. 결 론

우리는 고정 크기 순환적 메모리 할당 기법의 단점을 개선하여 더 안전하고, 적용하기 쉬운 동적 순환 메모리 할당 기법을 제안하였다. 메모리 누수가 있어도 시스템 전체에 영향을 미치지 않기 때문에 유닛 테스트 시에 우리 기법을 활용하여 메모리 누수를 검사해 볼 수 있다. 기존 기법은 상황에 따라 사용 중인 메모리가 재할당되는 문제를 가지고 있는데 우리는 강제적인 메모리 반환을 없애고 메모리 누수로 판단되는 객체만 강제로 반환함으로써 이것을 최소화 하였다. 특히, 연결 리스트, 트리, 해쉬 테이블과 같이 기존 기법을 적용하기 어려운 자료구조를 테스트 할 때 더욱 유용할 것이다. 우리 기법의 성능을 평가하기 위해서 Redbase정도의 데이터베이스 관리 시스템을 구현하여 시간적, 공간적 오버헤드를 측정하였고 메모리 누수 검출 능력을 평가했다. 실험 결과 동적 분석을 이용하기 때문에 기존 기법보다는 시간적, 공간적 오버헤드가 컸으나 데이터 손실의 가능성이 적고, 일부 모듈에 대해서만 적용하면 오버헤드를 줄일 수 있었으며 우리가 일부러 넣은 열 개의 메모리 누수 전부를 검출하는 능력을 보였다. 따라서 우리의 기법은 프로그래머가 유닛 테스트 시에 자신이 개발한 모듈에 대해 메모리 누수를 검사하거나, 이미 서비스되고 있는 프로그램의 일부 모듈에 대해서 메모리 누수를 검사할 때 유용할 것이다.

참 고 문 헌

[1] Valgrind-project., Available: http://www.valgrind.org

[2] H. H. Nguyen and M. Rinard, "Detecting and Eliminating Memory Leaks Using Cyclic Memory Allocation," In *International Symposium on Memory Management(ISMM)*, pp.15-30, October 2007.

[3] T. M. Chilimbi and M. Hauswirth, "Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling," In *Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, pp.156-164, October 2004.

[4] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," In *Programming Languages Design and Implementation(PLDI)*, pp.190-200, June 2005.

[5] The Linux man-pages project., Available: http://www.kernel.org/doc/man-pages/online/pages/man3/malloc_hook.3.html

[6] RedBase Project., Available: http://infolab.stanford.edu/ ~widom /cs346/project.html

[7] D. L. Heine and M. S. Lam, "A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak detector," In *ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI)*, pp.168-181, 2003.

[8] Y. Xie and A. Aiken, "Context- and Path-Sensitive Memory Leak Detection," In *the 5th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESAC/FSE)*, pp.115-125, 2005.

[9] S. Cherem, L. Princehouse, and R. Rugina, "Practical Memory Leak Detection using Guarded Value-flow Analysis," In *the 2007 ACM SIGPLAN Conference on Programming language design and implementation(PLDI)*, pp.480-491, 2007.

[10] G. Novark and E. D. Berger, "Efficiently and Precisely Locating Memory Leaks and Bloat," In *Programming Languages Design and Implementation(PLDI)*, pp.397-407, 2009.

[11] R. Hastings and B. Joyce, "Fast Detection of Memory Leaks and Access Errors," In *the Winter '92 USENIX conference*, pp.125-136, 1992.

[12] Microsoft TechNet, Microsoft Corporation, "Memory Leak Diagnoser," 2007.

[13] M. D. Bond and K. S. McKinley, "Tolerating Memory Leaks," In the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA), pp.109-126, 2008.

[14] Y. Tang, Q. Gao, and F. Qin, "LeakSurvivor: Towards Safely Tolerating Memory Leaks for Garbage-Collected Languages," In the 2008 USENIX Annual Technical Conference, pp.307-320, 2008.

임 우 섭

2005년 인하대학교 컴퓨터공학(학사). 2009년~현재 성균관대학교 정보통신공학부 석사과정. 관심분야는 컴파일러, 프로그램 동적 분석

한 환 수

1993년 서울대학교 컴퓨터공학(학사). 1995년 서울대학교 컴퓨터공학(석사). 2001년 Univ of Maryland 전산학(박사). 2001년~2002년 Intel 책임연구원. 2003년~2008년 KAIST 전산학과 교수. 2008년~현재 성균관대학교 정보통신공학부 교수 관심분야는 최적화 컴파일러, 고성능 컴퓨팅, 프로그램 분석

이 상 원

1991년 서울대학교 컴퓨터공학(학사). 1994년 서울대학교 컴퓨터공학(석사). 1999년 서울대학교 컴퓨터공학(박사). 1999년~2001년 한국 오라클. 2001년~2002년 이화여대 BK21 계약교수. 2002년~현재 성균관대학교 정보통신공학부 교수. 관심분야는 플래시 메모리 DBMS