임베디드 시스템의 메모리 보호에 관한 연구

임도연*, 박익수**, 오병균**, 권오봉*
*전북대학교 전자정보공학부 컴퓨터공학전공
**목포대학교 정보공학부 정보보호전공
e-mail: haha@mokpo.ac.kr

A Study on Memory Protection for Embedded Systems

Do-Yeon Im*, Ik-Su Park**, Byeong-Kyun Oh**, Oh-Bong Gwun*
*Div. of Electronics and Information Engineering, Chon-Buk University
**Div. of Information Engineering, Mok-Po University

요 약

일반적으로 응용 프로그램의 메모리 요구를 배치 전에 평가하는 것은 많은 어려움이 따르기 때문에 주기억장치의 부족을 초래한다. 또한 임베디드 시스템의 디스크와 가상 메모리의 결핍은 out-of-memory 에러가 발생할 때 응용이 확장되기 위한 swap 공간이 없어 시스템이 붕괴되고 가상 기억장소로부터의 보호가 없어 세그먼트가 그 바운드를 초과했다는 것조차 발견되지 않으므로 붕괴 전의 교정 동작이 불가능하게 한다. 시스템 붕괴가 치명적인 손실이 될 수 있는 임베디드 시스템에서 Out-of-memory 에러는 비신뢰성을 보이는 중요한 원인이 된다. 본 논문에서는 컴파일러에 의한 런타임 조사 코드를 사용함으로써 out-of-memory 에러들이 발생하기 바로 전에 발견하는 런타임 조사와 out-of-memory 이후 죽은 변수 같은 사용되지 않는 공간과 살아있는 변수의 압축으로 자유롭게 된 공간으로 스택이나 힙 세그먼트를 확장시키는 공간 재활용과 데이터 압축 기법으로 시스템 신뢰성을 개선하는 방법을 연구하였다.

1. 서론

임베디드 시스템은 대부분 하드디스크를 가지고 있지 않으며 가상 기억장치 또한 지원하지 않는 경우가 많기 때문에 out-of-memory 에러는 시스템을 심각한 위험에 빠트릴 수 있다. 임베디드 시스템을 디자인할 때 병행적으로 실행되는 모든 응용들의 최대 기억 요구 보다 큰 사이즈의 물리적 기억장치를 선택하여야 하지만 컴파일 시간에 응용의 최대의 기억 요구를 정확히 평가하는 것은 어렵다.

out-of-memory의 가능성은 임베디드 시스템들의 신뢰성을 낮춘다. 가상 메모리에 의한 보호가 없는 임베디드 시스템은 세그먼트가 그의 공간을 초과하는지 조사하지 않기 때문에 붕괴가 발생하기 전에 교정 동작을 취할 수 없다. 하드웨어 보호가 부족한 임베디드 시스템들의 문제와 그 결과로 인한 비신뢰성이 널리 인정되고 있다.[1]

본 논문은 임베디드 시스템의 신뢰성을 개선하기 위하여 안전 런타임 조사, 공간 재활용, 데이터 압축이라는 세 겹의 해결 방법을 취하는 소프트웨어에 의한 메모리 보호와 메모리 재활용 기법을 연구한다. 시스템 신뢰성을 개선하기 위한 첫 번째 테크닉은 컴파일러에서 모든 out-of-memory 조건들에 대해 소프트웨어 조사들을 끼워 넣어 응용 코드를 수정하는 것이다. 두 번째 테크닉은 메모리가 부족한 세그먼트들이 시스템의 불연속적인 비활용 공간으로 확장되도록 함으로써 out-of-memory 에러를 미루거나 피할 수 있게 하는 것이다. 세 번째 테크닉은 살아있는 데이터를 압축하고 그 결과로 자유롭게 된 공간을 스택이나 힙이 넘칠 때 그것들이 확장되도록 사용하는 것이다.

논문의 구성은 2장에서는 관련 연구에 대해 살펴보고, 3장에서는 안전 런타임 조사와 공간 재활용 및 데이터 압축에 의한 메모리 오버플로 보호 기법에 대해 기술하며, 4장에서 결론을 맺는다.

2. 관련 연구

몇몇의 임베디드 시스템에서는 메모리 보호를 제공하지만 스왑 공간이 아닌 가상 메모리의 제한된 형태가 가능하다.[2] 가상 메모리가 있더라도 모든 임베디드 시스템은 하드 디스크와 그에 따른 스왑 공간의 전형적인 부족 때문에 물리적 메모리의 크기가 한정된다.[3] 그 결과 심지어 메모리 관리 하드웨어와 가상 메모리가 있는 임베디드 시스템에서도 공간이 부족할 수 있다. 그러므로 이용 가능한 메모리의 제한된 크기에서 공간을 복구하는 테크닉이 필요하다. 가상 메모리를 제공하는 하드웨어 메모리 관리 유닛들(MMUs)은 세그먼트나 페이지 테이블들 그리고 관련 로직을 포함해야하고 그것들은 영역과 런타임, 전력 면에서 비용이 든다. 그러므로 낮은 비용으로 숨은 공간을 재생시키는 소프트웨어에 의한 메모리 보호가 요구된다.

프로그램에 이용할 수 있는 공간의 크기를 증가시키는 다른 하나의 방법은 가비지 콜렉션이다. 최근, 전통적인 가비지 콜렉션 테크닉들이 임베디드 환경들에 응용되었다.[4, 5] 하지만 가비지 콜렉션과 다른 본 연구의 특징은 전역 세그먼트로부터 공간을 복구하기 위한 여러 가지 시도와 신뢰성을 위한 런타임 조사를 제공하는 것이다. 또한, 본 연구의 전반적인 목표는 메모리 부족의 경우에 재활용 모드로 매끄럽게 전환함으로써 시스템의 신뢰성을 증가시키는 것이다.

3. 메모리 오버플로 보호 및 공간 재활용

3-1. 오버플로 보호를 위한 안전 런타임 조사

안전 런타임 조사는 out-of-memory 에러들을 발견하기 위한 소프트웨어만의 간단한 스킴이다. 스택은 프로시저 호출에서만 확장되고 힙은 malloc()과 같은 동적 메모리 할당 루틴에서만 확장된다. 따라서 기본 스킴은 각 프로시저 호출과 각 malloc() 호출에 오버플로에 대한 런타임 조사를 간단히 끼워 넣는 것이다.

힙 조사에서는 만약 malloc()이 사용가능한 적절한 크기의 어떤 자유로운 영역도 없다는 것을 발견한다면 out-of-memory 에러가 보고 된다. 이것은 malloc()의 대부분의 버전들에 디폴트로 존재하므로 어떤 오버헤드도 덧붙이지 않는다.

스택 조사는 각 프로시저 호출에 끼워 넣어지고 이것은 런타임 오버헤드를 더한다. 그것은 컴파일러가 각 함수의 초입에 코드를 삽입하고, 새로운 값과 업데이트된 스택 포인터, 그리고 스택에 현재 허용되는 경계를 비교한다. 여기서 스택의 경계는 힙이 스택의 확장 방향에 인접해 있다면 힙 포인터일 것이고 다른 작업의 스택이 스택의 확장 방향에 인접해 있다면 인접해 있는 스택의 베이스일 것이며, 스택 끝이 메모리의 끝에 있다면 메모리의 끝이 될 것이다. 그림1은 스택 조사에 대한 안전 런타임 조사 코드를 보여준다.

```
1  PER-PROCEDURE SAFETY CHECK CODE
2  if(Stack-Ptr < ORIGINAL BOUND)
3  { /*Stack Overflow */
4     call routine to handle out-of-memory condition
5  }
```

[그림 1] 안전 런타임 조사를 위한 Pseudo-code

3-2. 스택과 힙의 확장을 위한 공간 재활용

스택과 힙의 공간이 부족하다는 것이 발견될 때 불연속적인 공간 안으로 확장되게 하는 것은 다음과 같이 구현된다. 첫째, 컴파일 시간의 liveness 분석이 확장하기 위한 후보자들로서 코드의 각 포인트에서 죽은 전역변수를 발견한다. 이 liveness 정보는 런타임 데이터 구조에 저장된다. 데이터 구조의 크기를 줄이기 위해 liveness 정보는 명령이 아니라 지역마다 저장된다. 범람 공간이 죽은 변수가 살려지기 전에 해제된다는 것을 보장할 수 있다면 다음 지역 내에서는 살게 되는 죽은 변수들 또한 세그먼트가 확장되기 위해 사용될 수 있다. 둘째, 앞서 기술된 런타임 조사가 스택이나 힙이 메모리 부족이라는 것을 발견하면 범람하는 세그먼트가 사용하지 않는 공간 안으로 불연속적으로 확장되도록 하는 특별한 코드가 실행된다. 사용하지 않는 공간은 스택이나 힙이 확장되기 위한 죽은 globals일 수 있고, 스택이 확장되기 위한 힙 안의 자유 홀일 수도 있다.

스택이 확장되기 위한 global을 선택하는 방법은 다음의 세 단계로 이루어진다. 첫째, 컴파일러는 프로그램을 몇 개의 지역으로 분할한다. 그리고 각 지역에 대하여, 그 지역 내에서 죽어 있고 그 지역으로부터 직·간접적으로 호출되는 모든 함수 내에서도 죽어있는 전역 배열의 목록을 만든다. 여기서 스택 함수가 전역 배열을 접근하지 않는다는 것을 보장하여 전역 배열이 재활용될 수 있도록 한다. 둘째, 재활용 후보자 목록은 큰 배열들에 재활용 선호도를 주기위해 컴파일 시간에 크기의 내림차순으로 정렬된다. 셋째, 런타임에서, 프로그램이 메모리 부족일 때 그 지역을 위하여 재활용 후보자 목록을 조사하고 스택을 확장하기위해 목록의 머리에 있는 전역

변수를 선택한다. 목록이 컴파일 시간에 내림차순으로 정렬되기 때문에, 확장되기 위해 가장 큰 죽은 global을 선택한다. 이러한 지역별 재활용 코드와 그 지역에 대한 재활용을 수행하는 코드가 추가된 안전 런타임 조사 코드가 그림 2에 보여진다.

```
PER-REGION REUSE CODE
Current-Region ←CURRENT REGION CONSTANT ID

SAFETY CODE AUGMENTED WITH REUSE CODE FOR THAT REGION
if ((Stack-Ptr < ORIGINAL BOUND + Space needed by reuse routines)
    OR (Reuse-Started))
{
    if (!Reuse-Started)
    {
        Reuse-Started ← 1
        Current-candidate ← Head of
        Reuse-Candidate-List[Current-Region]
        Stack-Ptr ← Current-candidate→base-address +
        Current-candidate→size
    }
    else
    {
        if (Stack-Ptr <
            Current-candidate→base-address + Space needed by reuse routines)
        { /* Stack Overflow */
            Current-candidate ← Next element of
            Reuse-Candidate-List[Current Region]
            Stack-Ptr ← Current-candidate→base-address +
            Current-candidate→size
        }

        if (Stack-Ptr >
            (Current-candidate→base-address + Current-candidate→size))
        { /* Stack Underflow */
            if (Current-candidate ==
                Head of Reuse-Candidate-List[Current Region])
            {
                Reuse-Started ←0
            }
            else
                Current-candidate
                    ← Previous element of Reuse-Candidate-List[Current Region]
        }
    }
```

[그림 2] 재활용이 추가된 안전 런타임 조사 Pseudo-code

위의 스킴은 힙을 위해 전역 변수를 재활용하기 위하여 확장되는데 다음의 세 가지 부가 작업이 필요하다. 첫째, 재활용 후보자 목록들은 컴파일 시간에 크기만이 아니라 next-time-of-access와 크기에 의해 정렬된다, 그러므로 미래에 가장 나중에 살아나게 되는 죽은 전역 배열이 목록의 머리에 놓여진다. 만약 동시에 살아나게 되는 두 배열들이 있다면, 더 큰 것이 먼저 목록 안에서 놓여진다. 둘째, malloc() 라이브러리 함수는 할당 요청을 만족시키기 위해 이용할 수 있는 자유 영역이 없을 때 특별한 Out-of-Heap 함수를 호출하기 위하여 수정된다. 셋째, 컴파일러는 코드 안에 Out-of-Heap 함수를 삽입한다. 이 함수는 현행 지역의 재활용 후보자 목록의 머리에서 후보자를 선택하고 그것을 힙 free-list에 추가한다.

재활용 후보자 목록들이 죽은 전역 배열들의 next-time-of-access를 기반으로 재정렬되야하는 이유는 힙에 대한 liveness 분석은 어렵기 때문이다. 확장된 힙에 의해 점령된 죽은 global이 살려지는 경우에 global이 살아나기 바로 전에 그 힙이 해제되었는지 알기위해 런타임 조사를 한다. 확장된 힙이 비어있다면 프로그램은 성공적으로 실행되고 확장된 힙이 비어 있지 않다면 out-of-memory라고 선언한다. 재활용 후보자 목록들에 있는 죽은 globals가 next-time-of-access의 내림차순으로 정렬되는 이유는 더 나중에 살려지는 globals를 먼저 선택한다면 런타임 조사가 성공할 가능성이 더 크기 때문이다.

프로그램에 스택 공간이 부족할 때, 다른 하나의 가능성은 힙 안쪽의 자유 홀들 안으로 스택을 확장하는 것이다. 이것은 그림 2에서 스택이 out-of-memory 인지 조사하는 코드에 추가 코드를 끼워 넣음으로써 구현된다. 스택이 out-of-memory 일 때 코드는 먼저 죽은 globals안으로 스택을 확장하려고 하고 그것들이 가득 찬 뒤에만 스택이 힙 안의 자유 홀들 안으로 확장된다. 힙 안으로 확장되기 위해서 특별한 malloc() 호출이 만들어지고 그 후에 반환된 영역 안으로 스택이 확장된다. 이 malloc() 호출은 가장 큰 이용 가능한 크기, 또는 남아 있는 스택의 크기보다 더 큰 자유 홀을 반환한다.

3-3 스택과 힙의 확장을 위한 데이터 압축

프로그램이 스택 또는 힙에서 오버플로 될 때, 살아있는 전역 변수들을 압축함으로써 많은 공간을 자유롭게 할 수 있고, 스택 또는 힙을 그 자유 공간으로 확장시킬 수 있다. 살아있는 데이터는 모든 이용 가능한 죽은 공간이 재활용된 후에만 압축되고 그것이 접근되기 전에 분해 된다. 좋은 성능을 위해 압축할 global은 압축과 분해가 자주 일어나지 않도록 오랫동안 사용되지 않을 것이어야 한다.

압축에 의해 자유롭게 된 공간을 스택을 확장하는데 사용하는 방법은 3-2절에서 기술된 죽은 globals안으로 스택을 확장하는 방법과 유사하나 다른 점은 다음과 같다.

첫째, 재활용 후보자들은 살아있는 전역 배열들을 포함하도록 확장된다. 배열이 그 지역 전체에서 접근되지 않고 그 지역으로부터 직접 또는 간접적으로 호출된 어떤 함수에서도 접근되지 않는다면 전역 배열은 그 한 지역을 위한 재활용 후보자이다. 이 no-access 제약은 압축된 global이 다시 접근되기 전까지는 오버플로 되었던 스택이 제거된다는 것을 보장하기 때문에 인플레이스 분해를 할 수 있다.

둘째, 런타임에서 스택이 전역 세그먼트안의 특별한 후보자안으로 확장되려고 할 때 선택된 후보자가 그 포인트에서 살아있다면, 그것은 압축되고 나

중에 배열이 접근될 때 그것이 복구될 수 있도록 저장된다. 지역의 재활용 후보자들이 일단 결정되었다면, 필요할 때 압축이 개시된다. 이것을 구현하기 위하여, 재활용 후보자 목록들은 그것이 죽었는지 살아있는지를 가리키기 위한 별도의 필드가 각 후보자에 추가된다.

셋째, 모든 지역의 시작에서 컴파일러에 의해 삽입된 코드는 재활용이 시작될 때 다음의 지역에서 접근되는 압축된 전역 배열들 모두 그들 본래의 위치들안에 분해 될 수 있도록 수정된다.

```
1 ADDITIONAL PER-REGION CODE WITH COMPRESSION
2 if (Reuse-Started)
3 {
4     for (each global array GA used in region CURRENT
5     REGION CONSTANT ID and that is currently compressed)
6     De-compress GA in its original location
7 }
```

[그림 3] [그림 2]에 추가되는 압축 Pseudo-code

그림 3은 그림 2의 코드에 추가되는 코드를 보인다. 여기서 재활용이 시작 되고나면, 그 때 다음 지역에서 접근되는 모든 압축된 전역 배열들이 그들 본래의 위치 안에 분해 된다. 이 코드는 압축이 사용될 때만 덧붙여진다.

살아있는 전역 변수들을 압축함으로써 자유롭게 된 공간 안으로 힙을 확장하는 방법은 위에서 기술한 죽은 globals안으로 힙을 확장하는 방법, 그리고 압축된 살아있는 globals안으로 스택을 확장하는 방법을 결합함으로써 구현된다.

그것은 위에서 기술된 것처럼 전역 배열들의 next-time-of-access에 따라 정렬되는 재활용 후보자 목록들을 사용한다. 시스템에 힙 공간이 부족하게 되면, Out-of-Heap 함수를 호출하는데. 그것은 압축을 지원하기 위하여 약간 수정된다. 후보자가 살았는지 먼저 검사하고 만약 후보자가 살아있다면 제자리에 전역 배열을 압축하고, 압축에 의하여 자유롭게 된 공간을 가리키는 포인터를 가진 free 라이브러리 함수를 호출한다. 그리고 모든 지역 시작 전에 재활용이 시작됐는지 조사되고, 시작됐다면 모든 압축된 globals는 앞에서 기술한 것처럼 분해 된다. 분해 전에 오버플로 됐던 힙이 비어 있는지 조사되고, 비어있지 않다면 out-of-memory 에러가 선언된다.

이 스킴은 기존의 기술들을 조합한 것이기 때문에 어떤 새로운 데이터 구조도 사용하지 않으며 스택을 위해 globals를 압축하는 스킴과 같은 런타임 오버헤드를 갖는다.

4. 결론

본 논문에서는 임베디드 시스템의 신뢰성을 개선하기 위한 메모리 관리 방법을 연구하였다. 소프트웨어에 의한 런타임 조사 시스템은 메모리 보호가 없는 임베디드 시스템들에서 요구될 것이며 오버헤드가 매우 낮다. 한편, 죽은 공간을 재활용하고 살아있는 데이터를 압축하는 방법들을 위하여 추가 루틴을 수행하는데 스택에 약간의 공간을 요구한다. 하지만, 오버헤드 루틴이 빠져나가고 그들의 스택 프레임들은 응용 프로그램으로 돌아가는 시간까지 제거되므로 그 후에 공간을 재활용할 수 있기 때문에 그들의 스택 공간은 마지막 분석에서 낭비되지 않는다. 그리고 임베디드 시스템들에서 보통 ROM안에 저장될 재활용 후보자 목록들은 일반적으로 프로그램 코드 크기를 많이 변경하지 않는다.

참고문헌

[1] David Kleidermacher and Mark Griglock. Safety-Critical Operating Systems. Embedded Systems Programming, 14(10), September 2001.

[2] Windows CE.NET. Microsoft Corporation. http://www.microsoft.com/embedded/ce.net/default.aspx.

[3] George V. Neville-Neil. Programming Without A Net. ACM Queue: Tomorrow's Computing Today, 1(2):16-23, April 2003.

[4] Rafael C. Krapf, Jlio C. B. Mattos, Gustavo Spellmeier, and Luigi Carro. A Study on a Garbage Collector for Embedded Applications. In 15th Symposium on Integrated Circuits and Systems Design, pages 127-134, Porto Alegre, Brazil, September 2002. IEEE.

[5] C.D. Lo. The Design of a Self-Maintained Memory Module for Real-Time Systems. In The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, Alberta, Canada, July 2003. IEEE.