임베디드 시스템의 소프트웨어 기반 메모리 테스팅에 관한 연구

노명기*, 김상일*, 류성열**
*숭실대학교 컴퓨터학과
**숭실대학교 컴퓨터학과
e-mail : infinite@selab.ssu.ac.k

A Study on Software-based Memory Testing of Embedded System

Myong-Ki Roh*, Sang-Il Kim*, Sung-Yul Rhew**
*Dept.of Computer Science, Soong-Sil University
**Dept.of Computer Science, Soong-Sil University

요 약

임베디드 시스템은 특별한 목적을 수행하기 위해 컴퓨터 하드웨어와 소프트웨어를 결합시킨 것이다. 임베디드 시스템은 일반 데스크탑보다 작은 규모의 하드웨어에서 운영된다. 임베디드 시스템은 파워, 공간, 메모리 등의 여러 가지 환경적 요소에 제약을 받는다. 그리고 임베디드 시스템은 실시간으로 동작하기 때문에 임베디드 시스템에서 소프트웨어의 실패는 일반 데스크탑에서보다 훨씬 심각한 문제를 발생시킨다. 따라서 임베디드 시스템은 주어진 자원을 효율적으로 사용하여야 하고 임베디드 시스템의 실패율을 낮춰야만 한다. 치명적인 문제를 발생시킬 수 있는 임베디드 시스템의 실패의 원인 중 하나가 메모리에 관련한 문제이다. 임베디드 시스템 특정상 메모리 문제는 크게 하드웨어 기반의 메모리 문제와 소프트웨어 기반의 메모리 문제로 분류된다. 소프트웨어 기반의 메모리에 관련한 문제는 Memory Leak, Freeing Free Memory, Freeing Unallocated Memory, Memory Allocation Failed , Late Detect Array Bounds Write, Late Detect Freed Memory Write 등과 같은 것들이 있다. 본 논문에서는 임베디드 시스템의 메모리 관련에 대한 문제점을 파악하고 관련 툴을 연구하여 그 문제점들을 효율적으로 해결할 수 있는 기법을 점증적으로 연구하고자 한다.

1. 서론

임베디드 시스템은 특별한 목적을 수행하기 위한 컴퓨터 하드웨어와 소프트웨어의 조합이다. [1] 임베디드 시스템은 오직 1개 또는 소수의 임무를 수행하기 위해 프로그램이 작성되기 때문에 임무를 바꾼다는 것은 전체 시스템의 저하를 가져오고 새로 디자인을 해야 한다는 의미이다. 임베디드 시스템은 일반 데스크탑에서 보다 작은 규모의 하드웨어에서 수행된다. 그러므로 임베디드 시스템은 파워, 공간, 메모리 등의 여러 가지 환경적 요소에 제약을 받는다. 파워 제약은 시스템 디자인 결정의 모든 양상에 영향을 미친다. 파워 제약은 프로세서 선택, 속도, 메모리 구조에 영향을 미친다. 최대로 수행할 수 있는 능력은 남은 파워 양에 따라 결정되기 때문이다. 필요한 파워의 양은 CPU 클럭 속도와 동작하고 있는 전자 부품(CPU, RAM, ROM, I/O 장치 등등)의 수에 의해서 결정된다. 임베디드 시스템은 어떤 환경 조건에서도 동작할 수 있어야 한다. 임베디드 시스템은 한정된 메모리 때문에 ROM에 모든 코드를 넣는다. 임베디드 시스템의 코드는 정해진 크기의 ROM에 들어가야 하기 때문에 크기에 매우 제한을 받는다. 임베디드 시스템은 실시간으로 작동해야 한다. 임베디드 시스템은 특별한 임무를 수행하기 위해 프로그래밍 되었기 때문에 그 임무를 수행하고 그 결과를 출력하기 위해서는 실시간으로 작동해야 한다. 임베디드 시스템의 실패는 일반 데스크탑에서보다 심각한 문제를 발생시킨다. 임베디드 실패의 원인 중의 하나가 메모리 문제이다.

현대의 컴퓨터 시스템에서 많은 메모리 디바이스

가 사용 가능하다. 특히 임베디드 시스템에서는 해당 임베디드 시스템 하드웨어에 효율적인 메모리 디바이스를 사용하는 것이 중요하다. 임베디드 시스템에서의 메모리 문제는 크게 두 가지로 분류될 수 있다. 첫째는 하드웨어 기반의 메모리 문제이다. 둘째는 소프트웨어 기반의 메모리 문제이다. 하드웨어 기반의 메모리 문제는 Electrical Wiring Problem, Missing Memory Chip등과 같은 문제가 있다. Electrical Wiring Problem 은 설계 에러나 보드 생산, 그리고 제조 후에 받은 충격으로 발생할 수 있다. Missing Memory Chip은 눈으로 확인할 수 있는 명백한 문제이다. [6] 하드웨어 기술의 발달로 인해 하드웨어 기반의 메모리 문제의 발생은 극히 드물다. 현재의 임베디드 시스템의 메모리 문제는 소프트웨어 기반의 메모리 문제라고 보아도 과언이 아니다. 소프트웨어 기반의 메모리 문제는 대부분이 원시 소스에서 발생한다. 본 논문에서는 하드웨어적인 메모리 문제가 아닌 소프트웨어 기반의 임베디드 시스템의 메모리에 관련한 문제를 연구하고자 한다. 이와 관련된 문제는 다음과 같은 것들이 있다.

■Memory Leak
■Freeing Free Memory
■Freeing Unallocated Memory
■Memory Allocation Failed
■Late Detect Array Bounds Write
■Late Detect Freed Memory Write

본 논문에서는 임베디드 시스템의 심각한 문제를 일으킬 수 있고 메모리의 효율적인 사용을 위해 임베디드 시스템의 메모리에 관련한 문제를 파악하고 이를 테스팅 하는 법을 제시하고자 한다.

2. 임베디드 시스템의 메모리 관련 문제점들

임베디드 시스템에서 소프트웨어 기반의 메모리 문제는 다음과 같은 것들이 있다.

Memory Leak란 프로그래머가 어떠한 함수에 메모리를 할당해 놓고 메모리의 해제 없이 해당 함수를 종료시켰을 때 발생한다. <그림 1-1>은 Memory Leak의 간단한 소스 코드를 보여준다.

```
#include <stdio.h>
int main(int argc, char ** argv ){
    char *p = new char[10];
    p[0] = 'A';
    printf("%c\n",p[0]);
}
```

<그림 1-1> Memory Leak의 예

Freeing Unallocated Memory는 할당되지 않은, 즉 사용 불가능한 메모리를 해제할 때에 발생한다. <그림 1-2>는 할당되지 않은 메모리 해제에 대한 소스 코드를 보여준다.

```
void calcDisplay:: change_sign_double_copy( ){
    char tmp[MAX_CHAR];
    if( display[0] =='0'){
        strcpy(tmp, display+1);
    } else {
        tmp[0] ='-';
        strcpy( tmp+1, display);
    }
    strcpy( display, tmp);
    delete tmp; // tmp is a local variable!!
}
```

<그림 1-2> Freeing Unallocated Memory의 예

Freeing Free Memory는 프로그램이 이미 해제된 메모리를 해제하려고 할 때 발생한다. 하나의 함수가 메모리를 해제하였고 데이터 구조체가 그 메모리에 대한 포인터를 간직하고 있을 때 다른 함수가 해당 메모리를 해제하려고 할 때 나타날 수 있다. <그림 1-3a>와 <그림 1-3b>은 Freeing Free Memory에 대한 소스 코드를 보여준다.

```
calcController::~calcController( ) {
    if ( currentOp) delete currentOp;
    try{
        while(1){
            calcOp * topOp = stack.top();
            delete topOp;
            stack.pop();
        }
    } catch ( StackUnderflow) { }
}
```

<그림 1-3a> Freeing Free Memory의 예

```
void calcStack:: pop ( ){
    if ( _top > -1) {
    calcOp* tmp = top( );
    _stack[_top] =0;
    _top --;
    delete tmp;
    }else {
    throw StackUnderflow();
    }
}
```

<그림 1-3b> Freeing Free Memory의 예

Memory Allocation Failed 는 프로그램의 run-time 에 메모리 할당이 하지 못했을 경우에 발생한다. 예를 들며 증가하는 힙(heap) 영역에 대해 프로그램의 페이징(paing) 공간이 없을 때 발생할 수 있는 문제이다.

Late Detect Array Bounds Write는 프로그램이 할당된 메모리 블록 밖에서 어떠한 값을 사용하자고 할 때 발생하는 문제이다. 사용할 값보다 선언된 배열의 크기가 너무 작게 선언되었을 경우나 배열의 인덱스(index)가 잘 못 사용되었을 경우에 이러한 문제가 발생할 수 있다.

Late Detect Freed Memory Write는 프로그램이 이미 해제된 메모리 영역의 값을 사용하고자 할 때에 발생하는 문제를 나타낸다. Late Detect Freed Memory Write는 이미 해제된 메모리 블록에 대한 포인터를 가지고 있을 때 발생한다. 즉, 포인터를 너무 오랫동안 가지고 있거나 메모리를 너무 빨리 해제 시켰을 때 발생할 수 있다.

소프트웨어 기반 메모리 문제점의 대부분은 프로그래머가 메모리를 할당하고 그에 적절한 대응을 하지 않으므로 발생한다. 함수에서 메모리를 할당한 후에 그 사용한 메모리를 해제하지 않은 경우나 적절하지 않은 문자열의 복사 등과 같은 것들이다. [2]

3. 관련연구

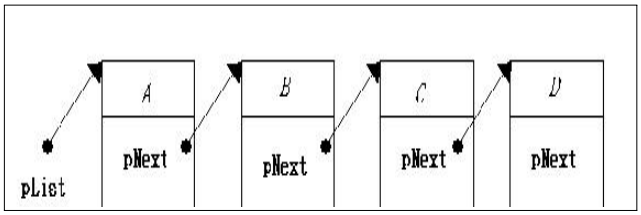
<그림 1-1>의 소스 코드에서 10 characters의 배열을 사용하기 위해 메모리를 할당했다. 그러나 할당된 메모리를 해제하기 위해 'delete [] p;'를 호출하지 않았다.

배열 p는 프로그램 실행 후에 해제되지 않고 남아 있다. 이러한 Memory Leak를 어떻게 탐지할 것인가? 가장 확실한 방법은 오랜 시간 소스 코드에서 *new* 연산자에 대응하는 *delete* 연산자(배열에서는 *new*[ ] /*delete*[ ] 또는 일반 C에서는*malloc*( )/*free*( ))를 사용했는지를 직접 검사하는 것이다. 이렇게 직접 검사하는 방법은 소스 코드가 좀 더 복잡해지거나 길어지면 사용하기 어렵다. 소스 코드를 좀 더 효과적으로 테스팅하기 위한 대안으로서 툴을 사용하는 방법이 있다. 테스팅 툴인 Purify에서 메모리를 테스팅하는 방법을 다음과 같다.

Purify는 테스팅 되는 프로그램 안의 모든 메모리 연산자를 감시한다. 프로그램에 의해 메모리가 할당되었지만 해제되지 않은 메모리, 메모리에 할당되고 정상적으로 해제된 메모리, 그리고 메모리 사용 후 해제되었지만 다른 곳에서 사용하려는 메모리 등 모든 메모리를 감시한다. Purify는 프로그램에서 사용되는 메모리의 각 바이트의 상태를 추적하는 테이블을 가지고 있다. 이 테이블은 메모리의 바이트를 나타내는 두개의 비트로 구성되어 있다. 첫번째 비트 레코드는 대응하는 비트들이 메모리 상에 할당되었는가를 판별하는 기록한다. 두번째 비트 레코드는 메모리가 초기화 되었는지에 대한 정보를 기록한다.

Purify 는 프로그램이 exit( ) 함수를 수행하고 종료되었을 때 Memory Leak에 대한 정보를 생성한다. Purify의 Memory Leak 정보는 실행동안 프로그램에서 누수된(leaked) 메모리의 양을 나타내고, 각각의 메모리 누수의 원인이 되는 함수들을 식별한다. Purify 는 포인터를 가지고 있지 않은 블록을 조사하면서 Memory Leak을 찾아낸다. 이러한 블록들은 접근되지 않기 때문에 해제되지도 않는다.

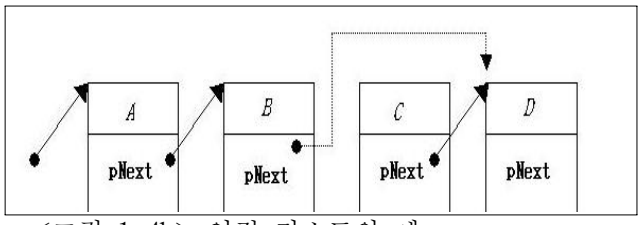
다음과 같은 연결 리스트의 예를 보면,



<그림 1-4a> 연결 리스트의 예

우선, 메모리 블록 A, B, C 그리고 D 모두 자신에 대한 참조(reference)를 가지고 있다. 블록A는 전역 변수인 *pList* 를 통해 참조되어진다. 그리고 블록B, C, D는 *pNext*포인터에 의해 참조되어진다. 그리고 나서 B블록 안에 pNext 포인터를 블록 D에 연결시키고 블록 C를 리스트에서 제거해보면, 리스

트 블록 C는 메모리 상에서 해제되지 않고 남아있을 것이다.



<그림 1-4b> 연결 리스트의 예

블록 C에 대한 다른 포인터가 없다면 블록 C를 해제시킬 방법은 없다(이것이 Memory Leak이다). Purify에서는 위의 예의 연결 리스트에 대한 memory leak 정보를 1 leaked block (블록 C), 3 memory-in-use (블록A, B, C) 그리고 4 allocated blocks(블록 A, B, C, D) 와 같은 형태로 보여준다.

전통적인 leak 탐지 방법은 해제되지 않은 메모리 블록을 찾기 위해 프로그램 종료 전에 메모리 할당과 그에 대응하는 메모리 해제를 단순 비교하는 것을 사용하는 것이었다. 앞의 연결 리스트의 예를 전통적인 malloc-debug leak 탐지를 이용하면 네 개의 모든 블록이 leak으로 탐지될 것이다.

Purify 는 C언어로 작성된 프로그램에서 *malloc* 함수와 *malloc* 함수와 연관된 함수들을 이용하여 Memory Leak을 찾아낸다. 만약 프로그램에서 *malloc* 함수를 사용하지 않았다면 Purify는 프로그램 내의 Memory Leak을 찾아낼 수 없을 것이다. C++ 언어로 작성된 프로그램에서는 *new* 연산자를 이용하여 Memory Leak을 찾아낸다. [5]

4. 결론

임베디드 시스템에서의 메모리 문제도 일반 애플리케이션에서의 메모리 문제와 유사하다. 그러나 임베디드 시스템은 자원과 공간이 제약되어있고 실시간으로 운용되며 특별한 임무를 수행해야 하기 때문에 임베디드 시스템의 메모리 실패는 더욱더 심각한 문제를 발생시킬 수 있다. 임베디드 시스템은 출시 후에는 그것의 수정이 어렵기 때문에 출시 전에 임베디드 시스템의 문제점을 찾아내는 것이 매우 중요하다. 임베디드 시스템은 컴퓨터 하드웨어와 소프트웨어의 조합이기 때문에 이에 따른 문제도 하드웨어적 문제와 소프트웨어 문제로 나뉜다. 소프트웨어 기반 메모리 문제는 해당 임베디드 시스템을 개발하는 프로그래머의 실수로 인해 발생하는 경우가 많다. 이러한 메모리 문제를 찾아내기 위해 툴의 대부분은 메모리 할당을 하는 *new* 연산자와 그에 대응하는 *delete* 연산자(배열에서는*new*[ ] /*delete*[ ] 또는 일반 C에서는*malloc*( )/*free*( ))를 비교하는 방법을 사용한다.

참고문헌

[1] Turley, Jim. “High Integration is Key for Major Design Win.” A paper presented at the Embedded Processor Forum, San Jose, 15 October 1998.

[2] D. Wagner, J. Foster, E. Brewer, and A. Aiken. “A first step towards automated detection of buffer overrun vulncrabilitics.” In symp. On Network and Distributed Systems Security. San Diego, CA, Feb 2000.

[3] Brown, John Forrest. “Embedded Systems Programming in C and Assembly.” New York: Van Nostrand Reinhold, 1994.

[4] Ganssle, Jack G. “The Art of Programming Embedded System.” San Diego: Academic Press, 1992.

[5] R. Hasting and B. Joyce. “Purify : Fast detection of memory leaks and access errors.” In Proceedings of the Winter Usenix Conference, 1992.

[6] Barr, Michael. “Software-Based Memory Testing.” Embedded Systems Programming, July 2000.

[7] S. Yong, S. Horwitz, and T. Reps. “Pointer analysis for programs with structures and casting.” In AGM SIGPLAN Conf. On Programming Language Design and Implementation, May 1999