공학석사학위논문

동적 바이너리 계측을 이용한 임베디드 펌웨어 시스템 상에서의 정확한 성능 프로파일링

Accurate Performance Profiling on Embedded Firmware System using Dynamic Binary Instrumentation

2015년 2월

서울대학교 대학원

전기 컴퓨터 공학부

김 선 규

동적 바이너리 계측을 이용한 임베디드 펌웨어 시스템 상에서의 정확한 성능 프로파일링

Accurate Performance Profiling on Embedded Firmware System using Dynamic Binary Instrumentation

지도교수 문 수 묵

이 논문을 공학석사 학위논문으로 제출함

2014년 11월

서울대학교 대학원

전기 컴퓨터 공학부

김 선 규

김선규의 석사 학위논문을 인준함

2014년 12월

위 원 장 백 윤 흥 (인)

부위원장 문 수 묵 (인)

위 원 윤 성 로 (인)

국문초록

성능 프로파일링은 소프트웨어 테스트의 일종으로, 프로그램의 어느 부분이 실행시간을 많이 소모하는 구간인지를 판단하게 된다. 이를 바탕으로 프로그램의 병목을 해소하여 최적화 하는 데에 도움을 준다.

하지만 리소스가 극도로 부족한 임베디드 펌웨어 환경에서 성능을 프로파일링 하기는 어렵다. 기존에 사용하던 방법들은 소스 코드 변경, 리소스 사용 증가, 운영체제 필요 등의 한계점으로 펌웨어에서는 그대로 사용하기 어렵다.

이에 따라, 본 논문에서는 위와 같이 적용이 불가능한 방법의 대안으로, 동적 바이너리 계측을 이용하여 소스 코드의 변경 및 코드 사이즈의 증가 없이 펌웨어에 구현된 함수의 실행 시간을 측정할 수 있었다. 하드웨어 디버거를 펌웨어가 실행되는 도중에 함수의 시작 및 종료 지점에 소프트웨어 인터럽트 명령을 삽입하여 프로그램을 정지시키는 방법으로 해당하는 문제를 해결하였다. 또한 이 과정에서 발생하는 오버헤드와 이로 인한 오차를 줄이기 위해서 소프트웨어 인터럽트를 branch and link 명령으로 대체하는 방법을 제안하여, 실제 벤치마크에서 오차가 줄어 더 정확한 결과가 나옴을 확인할 수 있었다.

주요어 : 소프트웨어 테스트, 소프트웨어 프로파일링, 동적 바이너리 계측

학 번 : 2013-20762

목 차

표 목 차

그 림 목 차

제 1 장 서론

소프트웨어의 구조가 복잡해지면서, 소프트웨어를 분석하는 작업은 더 어려워졌으며, 그 중요성 또한 더욱 커지고 있다. 그 중에서도 동적 프로그램 분석(dynamic program analysis)은 소프트웨어를 실제로 실행시키고, 이를 관측하여서 어떠한 결과가 나오는지를 분석하는 방법을 일컬으며, 실행시키지 않고서는 예상하기 힘든 부분을 계측하기 위해서 이루어진다.

성능 프로파일링(performance profiling)은 동적 프로그램 분석의 한 형태로, 프로그램을 실행시켰을 때 실행 시간 등의 프로그램의 전체 성능이나, 프로그램 내에서 특정한 함수나 명령의 호출 빈도와 소요 시간 등을 측정하게 된다. 이러한 결과는 주로 프로그램의 각 부분이 성능에 어느 정도 영향을 끼치는지 파악하는 데에 사용되며, 이를 통해 실제로 코드에서 차지하는 부분은 작지만 프로그램의 실행을 크게 지연시키는 병목(bottleneck)을 파악하여 제거하는 등의 방법을 통해 성능 향상을 지향할 수 있다.

이러한 성능 프로파일링은 일반적인 PC 환경 뿐만이 아니라 플래시 메모리와 같은 환경에서 작동하는 펌웨어 소프트웨어에도 유효하다. 다만 이러한 환경에서는 PC 환경과는 다른 제약이 존재한다. 우선 가상 시뮬레이션을 통해 테스트하는 것은 한계가 있다. 메모리의 온도를 측정하거나 플래시와 연결된 호스트와의 상호작용 같이 하드웨어에 직접적으로 연관되는 경우를 온전히 시뮬레이션 하기는 어렵기 때문이다. 또한 사용 가능한 리소스가 매우 제한적이기 때문에, 프로파일링 방법에 따라서 적용이 어려운 경우가 생기게 된다.

성능 프로파일링을 위해서는 함수의 실행 시간과 같이 프로그램 내에 정해진 구간 별 실행 시간을 측정하게 된다. 이를 측정하기 위한 가장 간단하고 정확한 방법은, 소스 코드에 시간을 측정하는 코드를 직접 삽

입하는 방법이다. 하지만 이 방법은 소스 코드를 수정하게 되어 실행 시킬 바이너리 코드를 새로 빌드해야 하며, 코드 삽입으로 인해 바이너리 코드가 커지게 된다. 따라서 메모리 리소스가 제한되어 있는 펌웨어 환경에서는 이 방법을 적용하기 어렵다.

이 외에는 OProfile[1], gprof[2], DynInst[3], Pin[4] 등과 같이 일반적인 PC 환경에서 사용하는 프로파일링 툴도 여럿 있다. 하지만 기존의 툴들은 직접 프로그램을 실행시키더라도 그 과정에서 운영체제의 API를 이용하게 된다. 따라서 운영체제가 존재하지 않는 펌웨어 환경에서는 이들을 그대로 적용하기는 어렵다. 보드 점검을 위한 하드웨어 디버거에서도 원하는 구간이 실행된 시간을 측정하는 기능을 제공하기는 하나, 사용 가능한 범위가 제한적이며, 디버거에서 시간을 측정을 하므로 측정 결과의 정밀성과 정확성에 한계가 있다.

이런 상황에서 사용할 수 있는 방법이 동적 바이너리 계측(Dynamic Binary Instrumentation)이다. 동적 바이너리 계측은 프로그램이 실행하는 도중에 바이너리 코드를 수정하는 방식으로 프로그램의 행동을 제어하는 방법으로, 이를 통하여 원하는 바를 관찰할 수 있다. 이를 이용하면 위와 같은 문제점 없이 정확한 성능 프로파일링이 가능하다.

본 논문에서는 위 문제를 해결하는 새로운 소프트웨어 프로파일링 방법을 제안하고자 한다.

- 동적 바이너리 계측을 이용하여 소스 코드의 수정 및 추가적인 메모리 사용 없이 펌웨어에서 원하는 함수의 실행 시간을 정확하게 측정할 수 있는 새로운 프레임워크를 제안한다.
- 펌웨어의 실행 시간을 정확하게 측정하여, 실제 임베디드 벤치마크 대상으로 측정 결과가 신뢰할 만함을 확인하였다.

앞으로 2절에서는 기존 툴에서 사용된 방식의 한계점을 설명하고 3절에서는 제안하고자 하는 성능 프로파일링 측정 방법을 설명할 것이다. 4절에서는 이 측정 방법의 정확성을 향상시키기 위한 방법을 제시하고, 5절에서는 정확성을 평가할 것이다. 그리고 6절에서 결론을 내고 향후 방향을 논의할 것이다.

제 2 장 성능 프로파일링 프레임워크

본 연구에서 제안하는 프레임워크는 펌웨어에 구현된 코드의 함수 단위 실행 시간을 측정하는 것을 목적으로 한다. 이를 위하여 함수가 시작하는 부분과 끝나는 부분에서 프로그램을 일시 정지 시킨 뒤, 두 지점에서의 시간차를 계산하여 함수의 실행 시간을 구하는 방법을 사용하였다. 이 때 함수가 시작하는 지점의 기준은 함수가 호출되었을 때에 이동하는 callee의 첫 주소로 정하였고, 종료하는 지점은 callee의 수행이 종료되고 돌아가게 되는 caller의 주소를 사용하였다. 또한 정확한 실행 시간을 측정하기 위하여 펌웨어를 실행시키는 타깃 보드에서 지원하는 시스템 타이머를 이용하여 시간을 측정하였다.

이를 구현하기 위해, 전체 프레임워크는 크게 세 부분으로 구성되어 있다. 우선 타깃 보드는 직접 펌웨어를 실행하게 되는 측정 대상이며, 시스템 타이머의 정보를 제공하게 된다. 실험에서 사용한 ARM Cortex-M3의 경우 보드에서 시스템 클락 단위로 시간을 측정하는 SysTick 타이머를 지원한다. 두 번째로는 호스트가 있으며, 호스트는 전체 프로파일링 과정을 제어하는 역할을 하게 된다. 필요한 정보를 가지고 타깃 보드의 실행를 제어하게 되며, 타깃 보드에서 넘어온 정보를 취합하여 함수의 실행시간을 계산하는 역할도 하게 된다. 마지막으로 하드웨어 디버거로, 호스트와 타깃 보드를 연결하며, API를 통하여 호스트가 타깃 보드를 제어하거나 필요한 정보를 얻어올 수 있도록 해준다.

함수의 실행 시간을 측정하는 과정은 크게 3 단계로 나눠진다. 가장 먼저 소스 코드를 스캔하여 어떠한 함수가 있는지를 조사한다. 다음 단계로 소스 코드를 빌드했을 때 나오는 디버깅 정보에서 함수들이 실제 메모리의 어느 주소에 구현되어 있는지를 조사하고, 앞의 정보와 취합하여 함수의 매핑 테이블을 작성한다. 마지막 단계로 이 매핑 테이블을 토대로 프로그램을 실행시켜 실시간으로 성능 프로파일링을 수행한다.

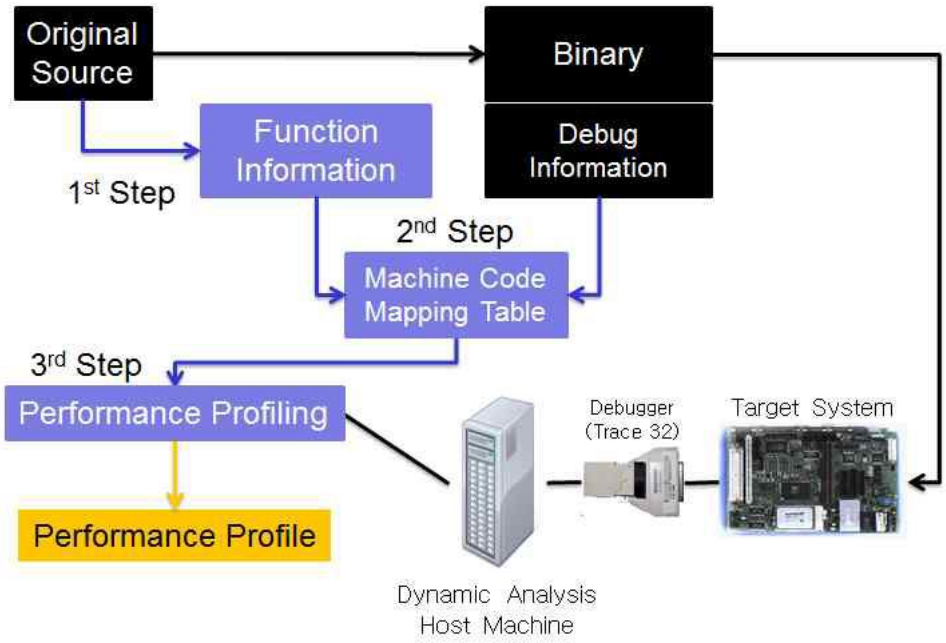

그림 2-1 성능 프로파일링 프레임워크의 개관

2.1 소스 코드 분석을 통한 함수 정보 기록

함수의 실행시간을 측정하기 위해서는 우선 프로그램 내에 어떠한 함수가 구현되어 있는지를 파악해야 한다. 이를 위하여 소스 코드를 스캔하여 어떠한 함수가 실제로 소스 코드의 어느 위치에 구현되어 있는지를 파악하였다. 각 소스 파일 별로 실제로 함수가 구현된 코드를 확인하면, 해당하는 함수의 이름이나 리턴 타입 등 함수에 대한 정보를 파악할 수 있다. 이렇게 이 단계에서는 소스 코드 상에서 얻을 수 있는 정보들을 수집하여, 어떠한 함수들이 실제로 구현되어 있는지를 확인하게 된다.

2.2 함수의 바이너리 주소 조사

두 번째 단계는 첫 번째 단계에서 얻었던 함수들의 정보와 바이너리 코드 주소 정보를 취합하여 매핑 테이블을 구성하는 단계이다. 첫 단계에서 소스 코드를 분석하여 프로그램 내에서 구현되어 있는 수들의 정보를 얻었지만, 이 정보만으로는 실시간으로 함수의 실행 여부를 확인하기 어렵다. 이를 위해서는 실제 실행 환경에서 함수가 구현되어 있는 부분이 포함된 메모리의 바이너리 주소를 알아야 한다. 이 정보는 프로그램 빌드 과정에서 얻을 수 있는 DWARF[5] 디버그 정보에서 가져온다.

DWARF 디버그 정보에는 소스 코드를 컴파일한 뒤 링킹 과정을 위해서 문자 형태의 심볼과 해당하는 심볼들이 대응되는 바이너리 주소를 정리해 놓은 심볼 테이블이 저장되어 있다. 함수에 대한 정보 또한 심볼 테이블에 저장되어 있으며, 앞에서 얻은 함수의 이름으로 심볼 테이블을 조회하면, 해당하는 함수가 실제 메모리의 어느 위치에 구현되어 있는지, 즉 함수가 호출되었을 때 처음 실행되는 주소를 파악할 수 있다. 따라서 DWARF 정보를 통하여 앞에서 정의한 함수의 시작 지점을 파악할 수 있다.

이러한 과정을 통하여 함수 명을 통해 함수가 시작하는 바이너리 주소를 가지는 매핑 테이블을 구성한다. 이 정보를 통해 원하는 함수의 시작 주소가 실행되었는지 여부를 확인하여 해당 함수가 실행된 것을 실시간으로 확인하게 된다.

2.3 실행 및 실시간 분석

함수에 대한 사전 정보가 다 갖춘 후에는 실제로 프로그램을 실행하여 함수의 실행 시간을 측정하게 된다. 우선 프로그램이 시작하기 전에 시간을 측정하고자 하는 함수들의 시작 지점에 도달하면 타깃 보드가 실행을 일시 정지하도록 미리 설정해 놓는다. 그리고 프로그램을 실행 시

키다 보면 함수의 시작 부분에서 정지하게 된다. 이 때 타깃 보드의 시스템 카운터를 이용하여 함수의 시작 시간을 측정한다. 그리고 함수의 종료 지점에서 타깃 보드가 다시 정지하도록 조치한다. 함수의 종료 지점은 일반적으로 레지스터에 저장되기 때문에, 함수 시작부에서 정지한 순간 이를 조회하여 확인할 수 있다. ARM의 경우 주로 LR(link register)이라 불리는 14번 레지스터에 이 값이 저장되어 있다.

이렇게 조치하면 함수가 종료하는 지점에서 타깃 보드가 다시 정지하게 된다. 함수의 시작 시간을 측정한 것과 동일한 방식으로 시스템 타이머에서 함수 종료 시간을 측정한 뒤, 함수의 시작 시간과의 차이를 계산하여 함수의 실행 시간을 계산한다. 그리고 해당하는 함수가 다시 실행되었을 때 실행 시간을 측정해주기 위하여 함수의 시작 주소에서 타깃 보드가 다시 정지하도록 조치하여 준다.

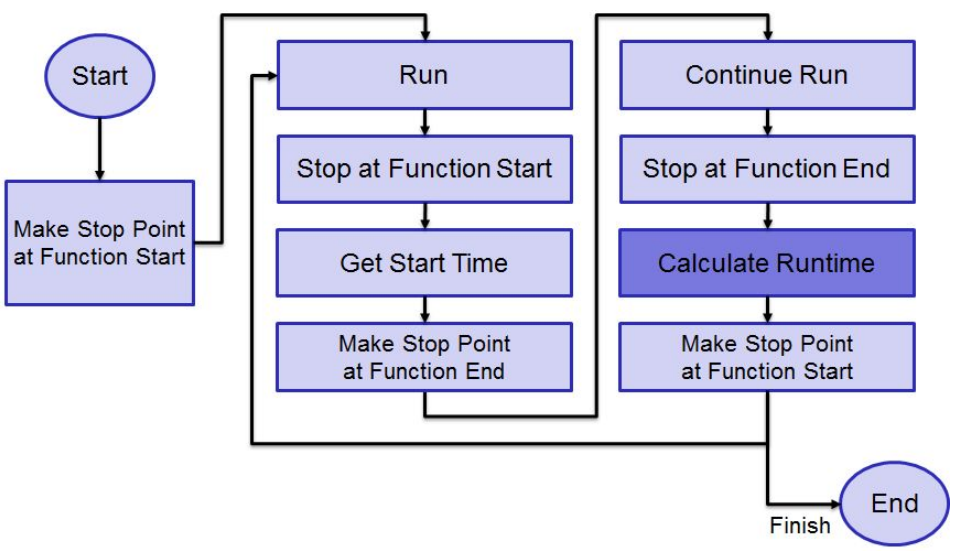

그림 2-2 함수 실행 시간 측정을 위한 실시간 분석 과정

여기서 타깃 보드의 실행을 일시 정지시키기 위하여 소프트웨어 인터럽트 코드를 동적으로 덮어씌우는 방법을 사용하였다[6]. 하드웨어 디버거로 타깃 보드의 exception handler에 breakpoint를 설정한 뒤에 프로그

램을 정지시키고자 하는 지점에 원하는 주소에 소프트웨어 인터럽트 명령을 덮어 씌워 놓으면, 해당하는 주소를 실행하면 인터럽트가 발생하여 인터럽트 핸들러로 이동한 뒤 breakpoint에 의하여 타깃 보드의 실행이 정지된다. 그리고 이렇게 프로그램이 정지하면, 소프트웨어 인터럽트 코드를 원래 코드로 되돌린 뒤에 실행을 계속하면 된다. 이 방법을 사용하면 추가적인 코드의 사용 없이 타깃 보드를 일시 정지 시킬 수 있다.

제 3 장 시간 측정 신뢰도 향상

3.1 현재 방법의 단점

현재 제안한 방식은, 소프트웨어 인터럽트 명령을 삽입한 뒤, exception handler에 breakpoint를 설정하여 프로그램을 정지시키는 방법을 사용한다. 이 방법을 사용하면 하드웨어 디버거가 실제로 타깃 보드에 설정하는 breakpoint는 한 개 뿐이기 때문에, 여러 개의 breakpoint를 설정했을 때 발생하는 오버헤드를 피할 수 있다[6]. 하지만 이 방법을 사용하여 실행 시간을 측정하는 데에는 몇몇 문제점을 내포하고 있다.

우선 exception handler에서 무조건 정지한다는 점이다. 실제 프로그램을 실행시키게 되면 다양한 예외 처리 상황이 발생할 수 있으며, 이 경우 exception handler 주소를 실행시키게 된다. 이에 따라 함수의 실행 시간 측정과 전혀 무관한 상황에서 타깃 보드가 여러 번 정지하는 상황이 발생하게 된다. 이렇게 되면 프로그램의 실행시간이 늦어지거나, 최악의 경우 의도하지 않은 동작을 할 가능성이 있다.

또한, 소프트웨어 인터럽트 명령이 추가로 수행됨에 따른 오차가 존재한다. 소프트웨어 인터럽트가 발생하면 현재 가지고 있는 레지스터 값을 스택에 백업하게 되며, 프로그램을 재개하기 위해서는 이 값들을 다시 레지스터로 복원시키는 과정이 필요하다. ARM Cortex-M3의 경우 0, 1, 2, 3, 12, 14, 15번 레지스터 값이 메모리에 백업되었다가 복원된다. 앞에서 제안한 방식으로 함수의 실행 시간을 측정하면 소프트웨어 인터럽트가 발생한 다음에 exception handler로 이동하여 정지하게 되므로, 인터럽트 발생 시간이 오류로써 측정값에 더해지게 된다.

3.2 Branch-Link 명령을 통한 개선

이러한 문제점을 최소화하기 위하여 소프트웨어 인터럽트 대신 branch-link 명령을 사용하는 방법을 시도하였다. 소프트웨어 인터럽트 명령을 대신하여 LR 레지스터 값을 타깃의 스택에 백업하는 push 명령과 branch-link 명령을 같이 덮어씌우는 방식으로 구현하였다. Branch-link가 발생하면 branch가 끝나고 돌아올 주소가 기존의 LR 값을 덮어씌우기 때문에 push LR 명령으로 기존 LR 값을 백업하였다. 이렇게 구현하면 새로 저장된 LR 값을 이용하여 어디에서 branch가 발생했는지, 즉 타깃 보드를 정지시키려 한 주소가 어디인지를 확인할 수 있다.

정지한 이후 실행을 재개하기 위해서는 소프트웨어 인터럽트를 사용했을 경우와 마찬가지로 저장한다. 다만, 소프트웨어 인터럽트는 종료하고 원래 작업으로 돌아가는 명령이 있는 반면, 이 경우에는 필요한 값을 수동적으로 되돌려줘야 한다. 타깃이 정지한 상태에서 호스트에서 하드웨어 디버거를 이용하여 수동적으로 LR, PC(program counter), 스택을 branch가 발생하기 전의 시점으로 복원시켜 줘야 한다.

기존에 백업한 LR 값을 스택에서 가져와서 복원시키고, 스택에서 LR 값을 제거해야 한다. 이다. 이렇게 하면 한 뒤에, branch-link 명령을 삽입하는 방법이다. 백업문제는 덮어씌누는거야 멍청아! 이 방법은 LR 레지스터를 으로 기존의 방법에서 정지시키고자 하는 지점에 software interrupt instruction을 삽입하는 것 대신에 push LR in-struction과 branch-link instruction을 같이 삽입한다. 이렇게 하면 software interrupt 대신에 branch-link를 통해 이동하게 기존의 LR 값이 스택에 백업되며, 새로운 LR 값을 이용하여 정지시키고자 하던 주소를 알아낼 수 있다.

소프트웨어 인터럽트를 사용했을 때와 비교했을 때의 장점은 우선 프로그램 정지에 걸리는 오버헤드를 줄여 오차를 줄일 수 있는 점이다. 소프트웨어 인터럽트를 사용하면 여러 번의 메모리 읽기/쓰기가 필요하나,

이 방법을 사용하면 메모리 연산은 레지스터 값 하나만 메모리에 쓰면 된다. 따라서 소프트웨어 인터럽트 발생에 의한 측정값의 오차를 줄일 수 있다.

또 다른 장점은 하드웨어 디버거로 breakpoint를 설정하는 지점이 자유롭다는 것이다. 소프트웨어 인터럽트가 발생하면 이동하는 지점은 무조건 exception handler로 정해져 있다. 따라서 다른 예외 상황이 발생하더라도 같은 지점으로 이동하게 되어, 일단 타깃 보드가 정지하게 되는 문제가 발생한다. 하지만 이 방법을 사용하면 branch 명령을 통해 이동하는 주소를 자유롭게 설정할 수 있다. 따라서 함수의 실행 시간을 측정하기 위한 경우 exception handler가 아니며 실제로 실행되지 않는 적당한 주소로 이동하도록 설정하면, 일반적인 예외를 처리하는 경우 프로그램이 정지하지 않게 된다.

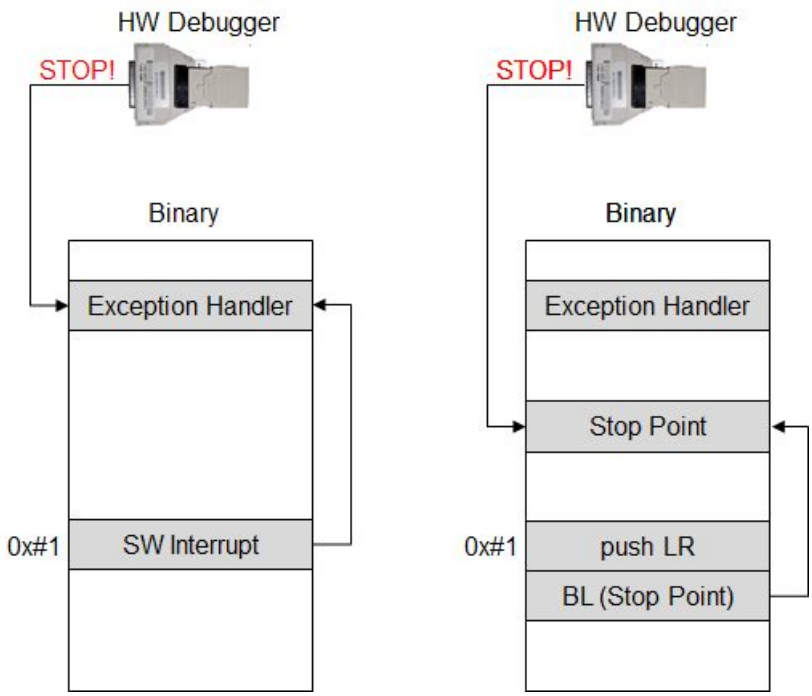

그림 3-1 소프트웨어 인터럽트를 사용한 경우(좌)와 branch-link 명령을 사용한 경우(우)

제 4 장 관련 연구

성능 프로파일링에 대한 연구는 다양하게 진행되어 있으며, 이를 수행하기 위한 툴들도 다양하게 만들어져 있다. 현재까지는 주로 일반적인 컴퓨터 소프트웨어를 대상으로 한 성능 프로파일링 위주로 연구되어 있으며, 프로파일링 툴의 경우에도 주로 사용되는 윈도우나 리눅스 커널 환경에서 수행되는 경우가 많다. 성능 프로파일링에는 주로 소스 코드를 수정하는 소스 코드 계측(source code instrumentation)과, 소스 코드의 수정이 없는 동적 바이너리 계측(dynamic binary instrumentation)의 두 가지 기법이 많이 사용된다.

4.1 소스 코드 계측

```
void foo() {
        clock_t _start, _end, _runtime;
        _start = clock();
        // Function body
        _end = clock();
        _runtime = _end - _start;
        return 0;
}
```

그림 4-1 소스 코드 계측을 이용한 함수 실행 시간 측정

소스 코드 계측은 프로파일링을 위해 소스 코드에 필요한 코드를 삽입하는 방법이다. 가장 단순한 예제로 각 함수에 위와 같이 시간을 측정하는 코드를 삽입하여 실행 시간을 측정하는 방법이 있다. 리눅스 커널에서 이 원리를 이용한 Ftrace를 통해 커널에 구현된 함수의 실행을 프로파일링 할 수 있다[7]. gprof[2]는 함수의 실행 순서를 특정하기 위해 부분적으로 소스 코드 계측을 사용하기도 한다.

이 방법은 구현이 간단하며, 프로그래머가 의도한 지점을 정확히 지정하여 측정할 수 있다는 장점이 있다. 하지만 이런 방법을 사용하면 일반적으로 코드 크기가 증가하기 때문에 메모리가 부족한 환경에서는 사용이 제한적이며, 수정된 코드를 가지고 실행용 바이너리 코드를 다시 빌드해야 한다는 불편함도 존재한다.

4.2 동적 바이너리 계측

동적 바이너리 계측은 바이너리 파일을 실행하면서 실행 중인 바이너리 코드를 조작하는 방법 또한 가능하다. 일반적으로 DWARF[5] 등의 디버그를 위해 생성되는 정보에서 바이너리 코드의 주소 정보를 얻은 뒤, 실행을 위해 메모리에 불러온 코드를 수정하여 원하는 지점의 실행되는 것을 확인하는 방법이 사용된다.

foo:	0x400113d4	~~push %ebp~~ **jmp 0x41481064**
	0x400113d5	~~mov %esp, %ebp~~
	0x400113d7	~~push %edi~~
	0x400113d8	~~push %esi~~
	0x400113d9	push %ebx
	…	
Tool:	**0x41481064**	**… // Tool code**
	…	
	0x414827fe	**call 0x50000004 // Call orig func**
	…	
Copy of foo entry:	0x50000004	push %ebp
	0x50000005	mov %esp, %ebp
	0x50000007	push %edi
	0x50000008	push %esi
	0x50000009	jmp 0x400113d9

그림 4-2 Pin 툴에서 함수가 호출되면 Tool code를 실행하도록 한 예시

DynInst[3]나 Pin[4] 등의 프로파일링 툴에서는 프로그램의 특정한 주소가 실행되면 사전에 지정한 위치를 수행하도록 jump 명령을 삽입하는 방식을 지원한다. 그림 2의 예시 코드를 보면 foo 함수가 호출되면 Tool 위치로 jump하라는 명령이 삽입되어 있으며, 그 위치에 있던 원래 명령은 다른 주소로 백업되어 Tool이 종료된 뒤에 call이 발생하여 실행된다. 성능을 측정하는 것은 아니지만, 이 원리를 통하여 호출 경로를 프로파일링하는 연구도 수행된 바 있다[8]. 또한 gdb[9] 등의 디버깅 툴에서 프로그램을 정지시키는 것 또한 동적 바이너리 계측을 이용한다.

이러한 방법을 사용하면 소스 코드의 수정이 필요 없기 때문에 원래 바이너리 파일을 그대로 사용할 수 있다는 장점이 있다. 또한 바이너리 코드의 크기가 변하지 않기 때문에, 구현에 따라서는 추가적인 코드 증가 없거나 최소화시키는 방법으로 구현하는 것 또한 가능하다.

하지만 기존의 프로파일링 툴들은 ptrace 등과 같이 운영체제가 지원하는 system call API를 사용하여 구현되어 있다. 따라서 운영체제가 없는 환경에서는 이를 그대로 사용할 수 없으며, 펌웨어 환경에서는 운영체제가 없기 때문에 기존의 툴들을 그대로 사용하는 것은 불가능하다.

제 5 장 프로파일링 정확성 평가

5.1 실험 환경

실험 대상이 되는 CPU는 ARM Cortex-M3를 사용하였다. 컴파일러는 ARM사의 RVDS[10] 4.0을 사용하였으며, -O3, -Ospace의 최적화 옵션을 사용하였다. 하드웨어 디버거 장치로는 Trace32[11]를 사용하였다. 벤치마크로는 C언어로 변환된 CaffeineMark[12]와 WhetStone 벤치마크를 사용하였으며, 해당하는 벤치마크의 주요한 함수를 대상으로 실행 시간을 측정하였다. 실행 시간은 Cortex-M3 보드에서 지원하는 시스템 카운터, SysTick으로 측정한 시스템 클럭 단위로 측정하였다.

5.2 정확성 비교 대상

우선 함수의 실행 시간의 정확성을 비교하기 위해서는 그 기준값이 필요하다. 소스 코드에 직접 SysTick을 측정하는 코드를 삽입하여 측정한 값을 기준으로 삼았다(이하 SRC). 이 방법은 측정 대상이 되는 프로그램이 종료될 때 까지 하드웨어 디버거로 타깃 보드에 간섭을 하지 않는다. 따라서 이 값과 비교하면 제시한 방법에서 하드웨어 디버거를 이용한 조작 과정에서 생기는 오차를 확인할 수 있다.

여기에 비교하기 위하여 두 가지 방법을 사용하여 함수의 실행 시간을 측정하였다. 우선 처음에 제시한 소프트웨어 인터럽트를 사용한 방법으로 측정하였다(이하 SWI). 또한 소프트웨어 인터럽트 대신에 branch and link 명령을 이용한 방법으로 측정하여 어떻게 개선이 되었는지를 확인해 보았다(이하 BL).

5.3 측정 결과 및 논의

표 5-1 CaffeineMark 벤치마크 내 함수의 실행 시간 측정결과

Function name	#	SRC	BL	BL-SRC	(%)	SWI	SWI-SRC	(%)
float_execute	1	1144024	1144050	26	0.0023	1144069	45	0.0039
float_initialize	1	4438	4420	-18	-0.406	4439	1	0.0225
logic_execute	1	220956	220987	31	0.014	221006	50	0.0226
loop_execute	1	90060	90080	20	0.0222	90099	39	0.0433
method_execute	1	310233	314149	3916	1.2623	321770	11537	3.7188
notInlineableSeries	100	186400	187000	600	0.3219	188900	2500	1.3412
arithmeticSeries	100	121200	121700	500	0.4125	123600	2400	1.9802
sieve_execute	1	149822	149844	22	0.0147	149863	41	0.0274

우선 CaffeineMark 벤치마크의 실험 결과를 정리하자면, SRC와 비교하였을 때 평균적으로 SWI가 0.9%, BL이 0.7% 더 큰 결과값을 보였다. 그 오차값은 전반적으로 매우 작았으며, 거의 정확한 결과가 나왔다고 할 수 있다.

다만 거의 모든 경우에서 SRC에 비하여 SWI와 BL 둘 다 측정값이 큼을 확인할 수 있다. 이것은 크게 두 가지 종류의 오버헤드로 설명할 수 있다. 하나는 소프트웨어 인터럽트나 branch and link 명령을 수행함으로써 발생하는 오버헤드이다. 메모리에 레지스터 값을 쓰고 읽는 과정이나 실행을 위해 연속된 코드가 아닌 곳으로 점프가 일어나는 것에 의한 오버헤드라고 볼 수 있다. 다른 하나의 원인은 하드웨어 디버거가 타깃 보드를 정지시키고 다시 실행시키는 과정에서 생기는 오버헤드이다. 타깃 보드를 정지시킨다고 하더라도 프로세서의 파이프라이닝 구조나 신호가 전해지는 지연시간을 고려해볼 때 SysTick 카운터가 정지할 때 까지 시간이 걸릴 수가 있다.

SWI와 BL의 차이를 보자면 BL이 SWI보다 값이 적은 것을 확인할 수 있다. 이는 실제로 branch and link 명령으로 발생하는 오버헤드가 소프트웨어 인터럽트에 의해서 발생하는 오버헤드에 비해 적다는 것을

보여주고 있다.

오차가 큰 경우들을 살펴보면 우선 nonInlineableSeries 함수와 arithmeticSeries 함수는 다른 함수들에 비해서 1회당 평균 실행 시간이 짧다. 다른 함수들은 1회 당 실행시간이 최소 5000정도에서 최대 22만의 실행시간을 가지는 데 비해서, 이 두 함수는 1회당 2000 미만의 실행 시간을 가지는 것을 알 수 있다. 따라서 상대적으로 오차가 더 크게 보인 것이다. 그리고 method_execute는 함수 안에서 nonInlineableSeries와 arithmeticSeries 함수를 100번씩 호출하게 되는데, 이 과정에서 타깃 보드를 400번 정지시키고 다시 실행시키면서 발생하는 오차가 누적되어 다른 경우에 비해서 눈에 띄는 오차를 보이고 있다.

표 5-2 WhetStone 벤치마크 내 함수의 실행 시간 측정결과

Function name	#	SRC	BL	BL-SRC	(%)	SWI	SWI-SRC	(%)
P3	899	824383	846858	22475	2.7263	863939	39556	4.7983
P0	616	33264	41888	8624	25.926	53592	20328	61.111
PA	14	177816	178166	350	0.1968	178432	616	0.3464
whetstone_main	1	1157725	1168803	11078	0.9569	1226726	69001	5.9601

WhetStone의 경우 평균적으로 BL이 7.0%, SWI가 16%정도 더 큰 값을 보였다. SRC < BL < SWI의 경향성 자체는 CaffeineMark와 다름이 없지만, 상대적으로 오차가 큰 것을 확인할 수 있다. 이는 벤치마크에서 측정한 함수들이 CaffeineMark에 비해서 짧기 때문이다.

P0 함수는 배열의 값 3개를 복사하는 아주 간단한 함수로, SRC 측정값 기준으로 1회 실행 시간이 54로 매우 짧다. 며, P3 함수는 간단한 if 문으로 구성되어 있는 함수로, 1회 실행 시간은 917로 P0보다는 길지만 CaffeineMark의 함수들에 비해서는 짧다. 이렇게 실행 시간이 짧으면 오버헤드가 전체 함수 실행 시간에서 차지하는 비율이 올라가서 오차율이 커질 수 밖에 없다. 다만 BL과 SWI를 비교해 보면 P0의 경우 오차율이 61%에서 26%로, P3의 경우 4.8%에서 2.7%로 오차가 크게 줄어드는 것 또한 확인할 수 있었다.

제 6 장 결론

본 논문에서는 리소스가 제한된 펌웨어 환경에서, 코드의 변경이나 메모리의 증가 없이 정확하게 프로그램의 함수의 실행 시간을 측정하여 소프트웨어 프로파일링을 수행할 수 있는 새로운 프레임워크를 제안하였다. 이 과정에서 기존 툴들이 운영체제의 API를 이용한 것 대신에 하드웨어 디버거를 사용하는 방법을 통해 기존의 한계점을 해결하였으며, 실제 벤치마크에서 신뢰할 만한 결과를 얻을 수 있었다.

다만 매우 짧은 길이의 함수에 대해서는 오차가 발생하는 것을 막기 힘들었다는 문제점은 존재했다. 하지만, 측정 결과에서 문제를 나타낸 함수들은 극단적으로 짧은 함수들이었으며, 이러한 함수들의 경우 어느 방법으로 측정하던 간에 측정으로 인한 오버헤드에 의한 오차를 막기 힘든 것을 감안해 볼 때 충분히 납득할 만한 결과라고 할 수 있다.

하지만 추가적인 연산과 프로세서 디버거에 의해서 오차가 발생하는 것은 사실이며, 추가적으로 발생하는 오차에 대해서 더 효과적으로 분석하여 오차의 규칙성을 찾고 이를 보정할 수 있는 방법이 있다면 더 정확한 프로파일링이 가능할 것이다.

참 고 문 헌

[1] OProfile, http://oprofile.sourceforge.net/

[2] gprof, http://sourceware.org/binutils/docs/gprof/

[3] DynInst, http://www.dyninst.org/

[4] Pin, http://www.intel.com/software/pintool/

[5] DWARF, http://www.dwarfstd.org/

[6] 정은지 외, 동적 바이너리 계측을 이용한 최적화된 프로그램의 정확한 분기 커버리지 측정, 한국정보과학회 2012년 가을 학술발표논문집, ol. 39, No. 2(B), p58-60, 2012

[7] Tim Bird, Measuring Function Duration With Ftrace, Proceedings of the Linux Symposium, p47-54, 2009

[8] Milind Chabbiet al., Call Paths for Pin Tools, Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, p76-86, 2014

[9] GDB: The GNU Project Debugger, http://www.gnu.org/s/gdb/

[10] ARM Compiler, http://arm.com/products/tools/arm-compiler.php

[11] Trace32, http://www.lauterbach.com/

[12] CaffeineMark, http://www.benchmarkhq.ru/cm30/

Abstract

Accurate Performance Profiling on Embedded Firmware System using Dynamic Binary Instrumentation

Sungyu Kim

Electronic Computer Engineering

The Graduate School

Seoul National University

Performance profiling is a kind of software test to analyze performance of software in part and which part of program is time consuming. Analyzed result helps programmer to resolve bottleneck effect to optimize the software.

However, it is hard to commit performance profiling on embedded firmware environment, with extremely restricted resource. Existing methods cannot be utilized in the environment by several limiation, like source code change, increased resource usage, or OS dependance.

For this reason, this paper introduces an alternative to measure function runtime without source code change nor code size increasement using dynamic binary instrumentation, The limitations can be resolved by hardware debugger by patching software interrupt instruction at function start/end point. In addition, to reduce error

from overhead, better method using branch and link instead of software interrupt is introduced and it was able to fiigure out that thiis method could get better results with less error.

keywords : software test, software profiling, dynamic binary instrumentation
Student Number : 2013-20762