임베디드 소프트웨어 결함 추적을 위한 효율적인 Record and Replay 기법 개발

김우종, 유민수
한양대학교 전자컴퓨터통신학과
e-mail : {wjkim.msryu}@rtcc.hanyang.ac.kr

Efficient Record-and-Replay Technique for Fault Trace on Embedded Software

Woojong Kim and Minsoo Ryu
Dept. of Electronics & Computer Engineering, Hanyang University

요 약

임베디드 시스템이 소형화되면서도 많은 기능들이 요구됨에 따라 여기에 올라가는 임베디드 소프트웨어 역시 점점 복잡해지고 있다. 특히 멀티 쓰레드 환경에서 수행되는 임베디드 소프트웨어의 경우, 실행도중 오류가 발생했을 때 버그의 원인을 찾기가 어려울 뿐 아니라, 버그를 재현하는 것 또한 쉽지 않다. 효과적인 디버깅을 하기 위해서는 프로그램 실행 중에 버그가 발생했던 상황을 그대로 재현해야 한다. 본 논문에서는 프로그램이 실행하는 도중에 이벤트가 발생하는 시점의 이벤트 정보를 record 하고, 나중에 이를 이용하여 버그가 발생한 시점으로 replay 할 수 있는 기법을 개발하였다. VPOS[1] 에 이 기법을 적용함으로써 임베디드 소프트웨어의 결함을 좀더 쉽게 탐지하여 효율적인 디버깅이 가능하도록 하였다.

1. 서론

1.1 배경

소프트웨어 프로젝트에서의 성패를 결정짓는 과정 중 하나가 디버깅이라고 할 만큼, 예전에 비해서 디버깅의 중요성은 점점 강조되고 있다. 프로그램 수행 도중에 버그가 발생했을 때, 서버 또는 데스크 탑 기반의 소프트웨어와는 달리 임베디드 소프트웨어는 제한된 환경에서 디버깅을 해야 하기 때문에 더 많은 시간과 비용이 요구된다. 또한 임베디드 시스템이 소형화되고, 여러가지 기능들이 요구되어짐에 따라 임베디드 시스템 위에서 동작하는 임베디드 소프트웨어는 점점 복잡해지고 있다.

1.2 동기

효과적인 디버깅을 하기 위해서는 버그가 발생했을 당시의 상황을 그대로 재현하여 버그의 원인을 찾아내야 한다.

이전의 coredump 를 이용하는 디버깅 방법의 경우, coredump 가 생성되었던 당시의 정보만을 바탕으로 재현할 수는 있지만, 이 경우 coredump 가 생성되기까지의 이전 정보를 얻을 수 없기 때문에 버그의 원인이 되는 단서를 탐지하는데 한계가 있다. 또한 기존의 record and replay 를 이용한 디버깅 방법의 경우, 프로그램의 실행정보를 얻기 위해 소스코드에 실행정보를 record 하는 코드를 삽입함으로서 실행 결과의 변화를 초래하는 탐침 효과(probe effect) 를 발생시킨다.

대부분의 임베디드 소프트웨어 디버깅의 경우 타겟(on-target) 자체의 디버깅 보다는 UART, JTAG 과 같은 인터페이스를 이용한 호스트에서의 원격 디버깅 방법을 사용한다. 이러한 방법은 호스트와 타겟 간의 시간 차로 인한 지연으로 버그가 발생한 당시의 상황을 재현하기 힘들다.

요즘 들어, 이러한 문제점을 보완하기 위해 고성능의 하드웨어 기반의 디버깅 장비나 소프트웨어 디버깅 도구들이 출시되고 있다. 하지만 가격이 고가이면서도 개발환경(컴파일러, 아키텍처, 운영체제)에 의존적이기 때문에 디버깅에 제한이 있다.

1.3 목적

이러한 문제점을 해결하기 위해서 본 논문에서는 record and replay 기법을 이용한 효율적인 임베디드 소프트웨어 디버깅 기법을 제시한다.

① 소프트웨어가 수행 도중, 이벤트(하드웨어 또는 소프트웨어 인터럽트) 발생시, 로그(log) 데이터를 저장함으로서, 나중에 이벤트가 발생했던 상태를 replay 할 수 있다.
② VPOS[1] 에 포함(built-in) 됨으로서, 타겟 디

버깅이 가능하고, replay 시에 디버깅 명령어를 이용하여 버그의 원인을 탐지할 수 있다.

③ 커널(kernel) 레벨에서 record and replay 기법을 사용함으로서, kernel 에 대한 지식이 없는 개발자나 사용자들도 손쉽게 kernel-aware 를 통한 추상화된 kernel 정보를 얻을 수 있다.

1.4 해결방법

임베디드 시스템 상에서 프로그램이 실행될 때 발생하는 이벤트(하드웨어 또는 소프트웨어 인터럽트)마다, 이벤트 로그 데이터를 메모리의 특정 영역에 저장한다. 이후, 실행 도중 익셉션(exception)이 발생하거나, 프로그램 실행이 종료되면, 메모리의 특정 영역에 저장되어 있는 로그 데이터를 플래시 메모리에 저장한다.

타겟 재부팅(reboot) 후에, replay 모드에서 플래시 메모리에 저장된 이벤트 로그 데이터들이 발생한 순서대로 리스트(list)화 되어 출력되고, 여기서 replay 하고자 하는 이벤트를 선택한다. 선택한 이벤트가 발생한 시점에서 PC 레지스터(register)가 가리키고 있던 메모리 주소에 저장되어 있는 명령어 코드는 메모리의 다른 영역에 백업(backup)되고, 여기에 트랩(trap)을 발생시키는 명령어 코드를 삽입한다. 프로그램이 실행되면서, 삽입된 코드에 의해 트랩이 발생하게 되면, 디버깅 명령어를 사용할 수 있는 디버깅 모드로 진입하게 된다. 여기서 명령어를 통해 선택한 이벤트가 발생한 시점의 여러 가지 상태 정보들을 얻을 수 있음으로써 버그의 원인을 탐지할 수 있다.

1.5 관련 연구

초기의 record and replay 관련 연구들은 프로그램 실행 중에 발생하는 이벤트 정보와 그에 따른 순서와 내용을 모두 저장하는 방식을 사용했다[2,3]. 이 방법의 경우, 비교적 정확한 replay 를 보장할 수 있다는 장점이 있는 반면, 저장되는 정보의 양이 비효율적으로 커진다는 단점이 있다.

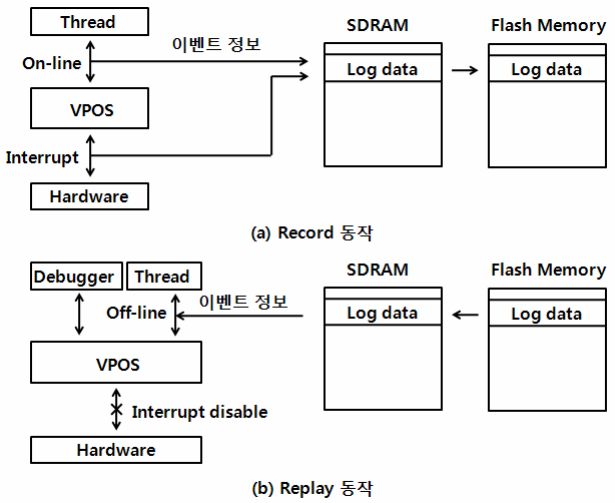
그 이후, 프로그램이 수행 중에 공유 메모리에 접근하는 순서만을 기록하는 방법이나 메시지(message)나 세마포어(semaphore)를 이용한 동기화 이벤트 만을 저장하는 방법이 연구되었다[4,5,6]. 하지만, 프로그램을 정확히 replay 하기 어렵거나, replay 정보를 수집하기 위한 소스코드 삽입으로 인한 탐침 효과가 비교적 커짐에 따라 실제 프로그램 실행과 다른 결과를 초래하였다.

최근에는 소프트웨어적인 접근뿐만이 아니라, 하드웨어 지원을 통해 좀더 정확하고, 효율적인 record and replay 방법이 연구되고 있다[7,8].

본 논문에서는 POSIX API 를 지원하는 VPOS 에 record and replay 기법을 적용하여 소스코드를 변경하지 않고, 임베디드 시스템 환경에서 임베디드 소프트웨어의 결함을 효율적으로 탐지하도록 구현하였다.

2. Overview

개발자가 작성한 프로그램은 VPOS 의 쉘(shell) 쓰레드에 의해서 새로운 쓰레드가 생성됨으로써 실행된다. 그림 1 과 같이 실행 도중 발생하는 이벤트(하드웨어 또는 소프트웨어 인터럽트)에 의해서 스케줄러가 실행되고 context switch 가 일어난다. 이 때, 이벤트 로그 데이터를 메모리의 특정 영역에 저장한다. 프로그램 수행 도중, 익셉션이 발생하거나, 프로그램 실행이 종료되면, 메모리의 특정 영역에 저장했던 이벤트 로그 데이터를 플래시 메모리에 저장한다.



(그림 1) record and replay 기법

타겟 재부팅 후, replay 모드에서는 앞서 플래시 메모리에 저장했던 이벤트 로그 데이터를 메모리로 복사하여, 발생한 순서대로 이벤트 로그 리스트를 출력한다. replay 하고자 하는 이벤트 로그 데이터를 리스트에서 선택하면, 이벤트(하드웨어 또는 소프트웨어 인터럽트)가 발생했던 시점까지의 모든 이벤트 로그 데이터의 location 부분의 명령어 코드를 메모리의 다른 영역으로 백업하고, 트랩을 발생시키는 명령어 코드를 삽입한다. 프로그램을 실행하여, 도중에 트랩이 발생하면, 앞에서 선택했던 이벤트의 카운터 번호와 비교하여 디버깅 명령어를 사용할 수 있는 디버깅 모드로 진입하게 된다. 디버깅 명령어를 실행함으로서, 이벤트(소프트웨어 또는 하드웨어 인터럽트) 또는 익셉션이 발생했을 당시의 상태 정보를 얻을 수 있음으로써 결함의 원인을 탐지할 수 있다.

3. Record Mechanism

record 는 소프트웨어 실행 도중에 이벤트(소프트웨어 또는 하드웨어 인터럽트) 가 발생할 때마다 실행되며, 특정 메모리 영역에 저장된다. 이 때 record 되는 정보는 다음과 같다.

① 이벤트 종류(소프트웨어 또는 하드웨어 인터럽트)
② 이벤트 발생 원인
③ 이벤트 발생 순서
④ PC 레지스터 값

이벤트 로그 데이터는 크게 이벤트 종류와 발생 원인과 순서를 나타내는 Identification 부분과 이벤트가 발생한 지점을 나타내는 Location 부분으로 나눠진다. 각 부분은 32 bit 를 사용하여 나타내고, 하나의 이벤트 로그 데이터는 총 8 byte 를 사용한다.

Identification 부분은 발생한 이벤트의 종류, 발생한 원인과 순서를 나타낸다. 각 bit 는 발생한 이벤트가 소프트웨어 인터럽트 또는 하드웨어 인터럽트인지 구분하기 위해서, 각각 인터럽트마다 특정한 값을 사용하여 메모리에 저장한다. 그림 2 와 그림 3 은 각각 소프트웨어 인터럽트와 하드웨어 인터럽트에 따른 로그 데이터 포맷을 나타낸 것이다.

<표 1> 시스템 콜 목록

시스템 콜 함수	번호
mg_send	1
mg_receive	2
sem_wait	3
sem_post	4
interrupt_enable	5
interrupt_disable	6

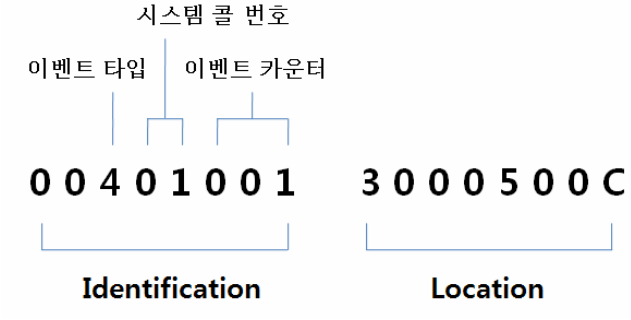
<표 2> 하드웨어 목록

하드웨어	번호
timer	1
uart	2

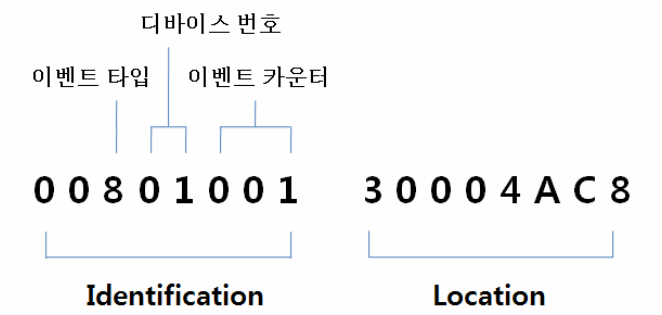
소프트웨어 인터럽트라면, 표 1 와 같이 인터럽트를 발생시킨 시스템 콜에 할당된 번호를 저장한다. 하드웨어 인터럽트의 경우에는, 표 2 와 같이 인터럽트를 발생시킨 하드웨어에 할당된 번호를 저장한다.

Location 부분은 이벤트가 발생한 시점의 PC 레지스터 값을 저장한다. 하지만, PC 레지스터 값 만으로는 재귀함수나 루프문이 실행되는 구조의 프로그램 루틴에서 이벤트가 발생하는 정확한 지점을 탐지하기 어렵다. 이러한 문제점을 해결하기 위해서 이벤트의 발생 순서를 저장함으로서, replay 시에 정확한 이벤트 지점을 찾는데 사용한다.

프로그램이 실행되는 동안에 record 되고, 수행 도중에 익셉션이 발생하거나 프로그램 실행이 종료되면 메모리에 저장된 로그 데이터를 플래시 메모리에 저장한다.



(그림 2) 소프트웨어 인터럽트 로그 데이터 포맷



(그림 3) 하드웨어 인터럽트 로그 데이터 포맷

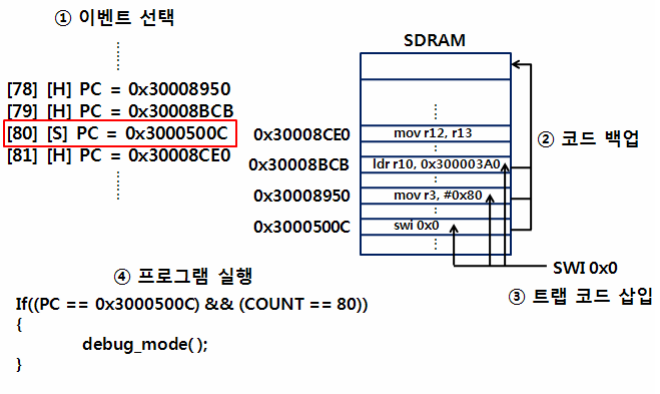
4. Replay Mechanism

타겟을 재부팅하면, 최소한의 초기화(레지스터 설정, 각 모드별 Stack 초기화, GPIO, UART)가 완료된 시점에서 표 3 과 같이 record 모드로 부팅을 할 것인지, 아니면 replay 모드로 부팅할 것인지 선택한다. replay 를 수행하기 위해서는, 타겟을 재부팅 시키고 replay 모드를 선택한다. replay 모드에서는 모든 인터럽트가 disable 된다.

<표 3> 부팅 모드 설명

모드	설명
record	프로그램 동작 시에 발생하는 이벤트를 특정한 메모리 공간에 저장
replay	저장된 로그 데이터를 바탕으로 replay 를 수행

replay mode 를 선택하면, 기존의 플래시 메모리에 저장되어 있던 이벤트 로그 데이터를 특정 메모리 영역으로 복사한다. 그리고 각각의 저장된 이벤트 로그 데이터를 읽어들여서 리스트로 출력한다.



(그림 4) replay 과정

그림 4 와 같이 리스트 번호를 선택하면 해당 이벤트가 발생했던 시점(PC 레지스터 값)까지 발생한 모든 이벤트 로그 데이터의 Location 부분에 저장되어 있는 명령어 코드를 메모리의 특정 영역에 백업한다. 그리고 나서, 트랩을 발생시킬 수 있는 명령어 코드를 삽입한다. 프로그램을 실행 시키면 이벤트가 발생

했던 시점에서 트랩이 발생하게 되고, 이때 이벤트 카운터를 비교한다. 리스트에서 선택했던 카운터 번호와 일치하지 않으면, 계속 실행해 나간다. 카운트 번호가 일치하게 되면, 디버깅 모드로 진입한다.

디버깅 모드에서 지원하는 명령어들은 표 4 에 나와있다.

<표 4> 디버깅 모드에서 지원하는 명령어

명령어	설명
bt	Record 된 시점의 백트레이스 (backtrace) 목록 출력
register	Record 된 시점의 레지스터 목록 출력
thread	Record 된 시점의 쓰레드 상태와 주소값 출력
mem dump <address>	특정 메모리 영역 출력
bl	현재 시점에서 실행할 수 있는 로그 데이터 리스트 출력
go	트랩이 발생하기 전까지 프로그램 수행

go 명령을 실행하면, 이벤트(소프트웨어 또는 하드웨어 인터럽트)를 수행하고 앞에서 메모리 특정 영역에 백업되어 있던 원래의 명령어코드를 다시 삽입하고 이 코드부터 프로그램이 실행된다.

5. 결론

효율적인 디버깅을 위해서는 버그가 발생한 상황을 최대한 그대로 재현해야 한다. 본 논문에서 제시한 record and replay 기법의 경우, 이벤트가 발생하는 시점만을 저장하여, 로그 데이터 저장에 필요한 공간을 줄였고, 프로그램 소스코드에 record 코드를 삽입하지 않음으로써 탐침 효과를 줄였다. 또한 커널에 대한 지식이 부족한 개발자나 사용자들도 디버깅 명령어를 이용하여 손쉽게 kernel aware 디버깅을 통한 추상화된 커널 정보를 알 수 있음으로써 쉽게 프로그램의 결함을 탐지할 수 있도록 하였다.

참고문헌

[1] 김지민, 유민수, “SOC 설계와 검증을 지원하는 실시간 운영체제,” 한국정보처리학회 춘계학술발표회 논문집, 2005 년 5 월.

[2] R.S Curtis and L.D Wittee, “Bugnet: a debugging system for parallel programming environment,” Proceedings of 3rd International Conference of Distributed Computing Systems. pp. 394-399. Oct. 1982

[3] E.T. Smith, “Debugging tools for message-based communicating processes,” Proceedings of 4th International Conference of Distributed Computing Systems. pp. 303-310, May. 1984

[4] T. J. LeBlanc and J. M. Mellor-Crummey, “Debugging parallel programs with InstantReplay,” IEEE Transactions on Computers, C-36(4), pp. 471-482, April 1987

[5] R. H. B. Netzer and B. P. Miller, “Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs,” Proceedings of the International Conference on Supercomputing, pp. 502-511, Nov. 1992

[6] R. H. B. Netzer, “Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs,” Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, pp. 1-11, May 1993.

[7] Thane H. and Hansson H. “Using Deterministic Replay for Debugging of Distributed Real-Time Systems,” In 12th EUROMICRO Conference on Real-Time Systems, pages 265-272 Stockholm , June 2000. IEEE Computer Society.

[8] S. Narayanasamy, G. Pokam, and B. Calder, “BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging,” Proc. 32nd Ann. Int`l Symp. Computer Architecture (ISCA 05), IEEE CS Press, 2005, pp. 284-295.