http://dx.doi.org/10.3745/KIPSTA.2011.18A.5.177

하드웨어 성능 카운터와 디버깅 기능을 이용한 리코드-리플레이 방법

맹 지 찬[†] · 유 민 수[††]

요 약

본 논문에서는 인터럽트의 기록과 재현을 통해 소프트웨어의 실행을 동일하게 재현하는 리코드-리플레이(record-replay) 기법을 제안한다. 전통적인 리코드-리플레이 방법에서는 경합(race) 현상을 대표적인 비결정적 요인으로 간주하여 임계영역으로의 진입/진출, 공유 메모리 접근, 메시지 교환 등을 기록하고 동일한 순서(order)로 재현하는 방법을 다루어 왔다. 하지만, 인터럽트 역시 프로그램의 실행에 영향을 끼칠 수 있는 중요한 비결정적 요인이며, 게다가 인터럽트의 경우 발생 순서는 물론 정확한 발생 시점을 재현하는 것이 필요하다. 이에 본 논문에서는 프로세서 하드웨어가 제공하는 성능 카운터와 디버깅 기능을 이용하여 인터럽트의 발생 시점을 정확하게 기록하고 재현하는 방법을 제안한다.

키워드 : 리코드-리플레이, 디버깅, 롤백 복구, 비결정적 이벤트, 순서, 인터럽트

An Efficient Record-Replay Mechanism using Hardware Performance Counters and Debugging Facilities

Ji Chan Maeng[†] · Minsoo Ryu[††]

ABSTRACT

In this paper, we present a record-replay technique based on interrupt logging and reproduction. Race conditions have been considered as the main source of nondeterminism in conventional record-replay approaches. However, interrupts are another source of nondeterministic computer system behavior, which must be reproduced at accurate time points, let alone the order of interrupt occurrence. We show that an interrupt-based replayer can be efficiently and effectively implemented by using hardware performance counters and debugging functionality. Experiments also show that the runtime overhead of the interrupt-based replayer is sufficiently low.

Keywords : Record-Replay, Debugging, Rollback Recovery, Nondeterministic Events, Order, Interrupt

1. 서 론

리코드-리플레이(record-replay) 기법은 소프트웨어의 실행을 동일하게 재현하는 방법으로서 디버깅이나 롤백 복구(rollback recovery) 분야에서 매우 유용한 것으로 알려져 있다. 일반적으로 프로그램의 실행을 동일하게 재현하는 것은 쉽지 않은 문제이다. 동일한 데이터를 가지고 동일한 코드를 실행한다 하더라도 경합(race) 등과 같은 비결정적(nondeterministic) 요인으로 인해 매번 프로그램의 실행이 달라지기 때문이다. 이를 해결하기 위한 한 가지 방법이 리코드-리플레이이며, 이는 프로그램의 실행에 영향을 끼치는 비결정적 이벤트들을 적절하게 기록하고 재현하는 방법을 일컫는다.

전통적인 리코드-리플레이 방법에서는 경합(race) 현상을 대표적인 비결정적 요인으로 간주하였다. 경합이란 쓰레드 또는 프로세스의 실행 순서에 따라 처리 결과가 달라지는 현상을 말하며, 이러한 경합 현상을 일으키는 이벤트로는 임계영역으로의 진입/진출, 공유 메모리 접근, 메시지 교환 등을 들 수 있다. 이에 따라 기존의 리코드-리플레이 연구에서는 이러한 이벤트들이 발생한 순서(order)를 정확하게 기록하고 재현하는 방법을 주로 다루어 왔다[1, 2, 3, 4, 7].

반면 [5]에서는 인터럽트 역시 프로그램의 실행에 영향을 끼칠 수 있는 중요한 비결정적 요인임을 지적하였다. 또한 인터럽트의 경우 발생 순서는 물론 정확한 발생 시점을 재현하는 것이 필요하다는 것을 함께 지적하였으며, 이를 위해 인터럽트가 발생한 시점에서 프로세서의 프로그램 카운

※ 이 논문은 2009년 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 연구임(KRF-2009-0077548).
† 준 회 원 : 한양대학교 전자컴퓨터통신학과 박사과정
†† 정 회 원 : 한양대학교 컴퓨터공학부 부교수
논문접수 : 2011년 3월 24일
수 정 일 : 1차 2011년 5월 24일
심사완료 : 2011년 5월 25일

터 값과 명령어 카운터(instruction counter) 값을 기록하는 아이디어를 소개한 바 있다. 하지만 당시에는 명령어 카운터를 제공하는 프로세서가 많지 않았기 때문에 [5]에서는 프로그램의 기본 블록(basic block)을 단위로 그 내부에서의 분기 횟수(branch count)를 측정하는 방법을 대신 제안하였다. 이는 기본 블록(basic block)마다 적절한 코드를 삽입하여 인터럽트가 발생한 시점에서의 분기 횟수를 기록하고, 또한 해당 시점의 프로그램 카운터 값을 함께 기록하는 방법으로 구현이 가능하다. 이후에 소프트웨어를 재실행(replay)할 때는 이전에 기록된 분기 횟수와 프로그램 카운터 값이 모두 일치하는 시점에 인터럽트가 재현되도록 한다. [5]에서 제안한 방법은 인터럽트를 정확한 시점에 재현하는 것을 가능하게는 하지만, 대상 프로그램을 수정해야 하는 동시에 그로 인한 실행 오버헤드가 매우 크다는 문제점을 가지고 있다. 실제로 [5]에서는 제안하는 방법을 구현한 결과 프로그램의 크기가 최대 40.3%, 수행시간은 최대 37.8% 증가하는 것으로 밝히고 있다.

본 논문에서는 프로세서가 제공하는 명령어 카운터(instruction counter)를 직접 이용하여 리코드-리플레이 기법을 효과적으로 구현할 수 있음을 보이고자 한다. 사실 [5]에서 명령어 카운터를 이용하는 아이디어를 처음으로 소개하기는 하였지만 이를 구현하는 구체적인 방법은 제시하지 못하였으며, 본 논문에서는 명령어 카운터를 이용할 때 고려해야 할 문제점과 해결 방법을 함께 제시하려고 한다. 나아가 제안하는 방법을 사용할 경우 구현이 매우 용이할 뿐만 아니라 실행 오버헤드를 크게 줄일 수 있다는 점을 함께 보이려고 한다. 이를 위해 본 연구진은 제안하는 방법을 x86 프로세서와 uC/OS-II 운영체제에 적용해보았으며, 이를 통해 응용 프로그램은 수정하지 않고 운영체제의 인터럽트 핸들러와 예외 핸들러를 수정하는 것으로 충분하다는 점과 또한 무시할만한 수준의 실행 오버헤드가 유발된다는 사실을 알아낼 수 있었다.

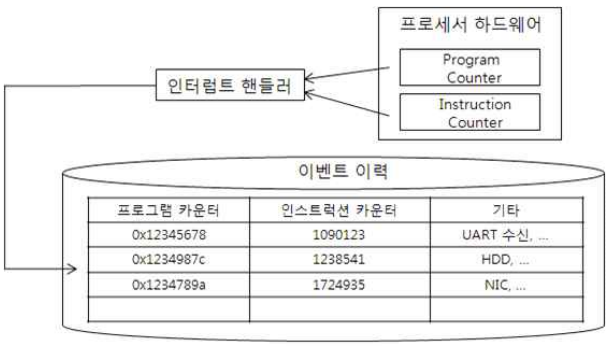
2. 인터럽트 기반 리코드-리플레이 방법

리코드-리플레이 관점에서 컴퓨터 프로그램의 실행은 리코드(record)와 리플레이(replay)의 두 가지 모드(mode)로 구분할 수 있다. 리코드 모드는 컴퓨터 프로그램이 원래의 목적을 달성하기 위해 실행하는 모드이며, 나중에 동일한 동작을 재현하기 위해서 필요한 정보를 수집하고 기록하는 단계에 해당한다. 리코드 모드에서 수집된 정보는 이벤트 이력(event history)이라 불리우는 데이터 구조에 기록되어 리플레이 모드에서 사용된다. 리플레이 모드는 디버깅이나 오류의 복구 등을 목적으로 리코드 모드와 동일한 실행을 재현하는 모드이다.

본 논문의 목적은 인터럽트를 정확하게 기록하고 재현하는 것으로서, 이를 위해 운영체제의 인터럽트 핸들러와 예외(exception) 핸들러를 수정하는 방법을 사용하였다. 인터럽트 핸들러에는 인터럽트가 발생한 시점을 포함하여 리플레이 모드에서 인터럽트를 재현하는데 필요한 정보를 수집하는 코드가 삽입되며, 예외 핸들러에서는 기록된 인터럽트를 에뮬레이션하는 코드가 삽입된다. 수정된 인터럽트 핸들러와 예외 핸들러의 상세한 동작 방법은 다음과 같다.

2.1 리코드 모드에서의 인터럽트 기록

인터럽트 핸들러에서는 인터럽트가 발생한 시점을 프로그램 카운터 값과 명령어 카운터 값으로 표현하여 기록한다. 이론적으로는 명령어 카운터만으로 이벤트의 발생 시점을 정확하게 표현하는 것이 가능하다. 하지만, 프로세서의 내부 구조에 따라 명령어 카운터에 오차가 있을 수 있으며, 또한 리코드-리플레이를 구현하기 위해 삽입되는 코드로 인해 명령어의 개수에 차이가 발생할 수도 있다. 참고로 인텔 IA-32 아키텍쳐 매뉴얼에서는 명령어 카운터를 포함하는 성능 모니터링 카운터(performance monitoring counter)에 오차가 존재할 수 있음을 밝히고 있다. 이에 따라 본 논문에서는 명령어 카운터에 존재하는 오차를 극복하기 위해 이벤트가 발생한 시점의 프로그램 카운터 값을 함께 사용하도록 한다.



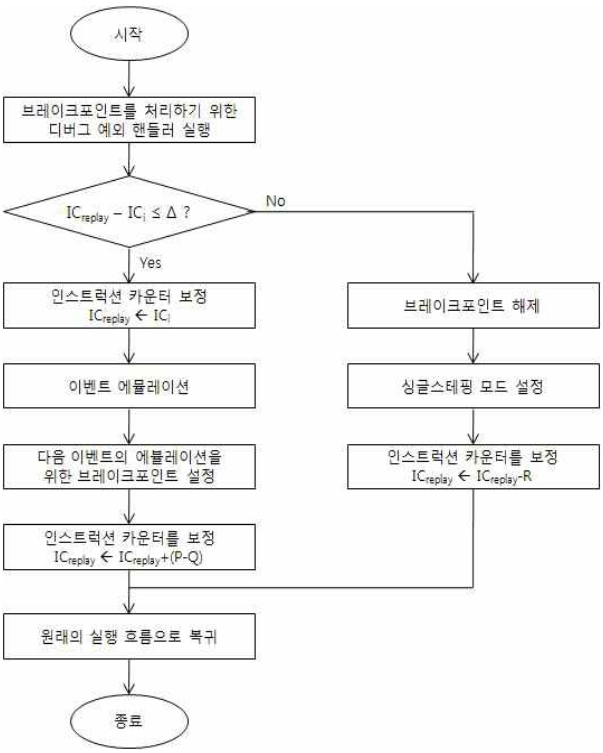
(그림 1) 이벤트 이력

인터럽트 E_i가 발생하면 인터럽트 핸들러에서는 (그림 1)과 같이 <프로그램 카운터 값 PC_i, 명령어 카운터 값 IC_i, 기타 XYZ_i>의 정보를 이벤트 이력에 기록한다. 여기서 기타 정보 XYZi는 이벤트 E_i의 에뮬레이션에 필요한 모든 정보를 포함한다. 예를 들면, 이벤트의 형태 및 종류, 이벤트와 함께 수반되는 데이터 등을 포함할 수 있다.

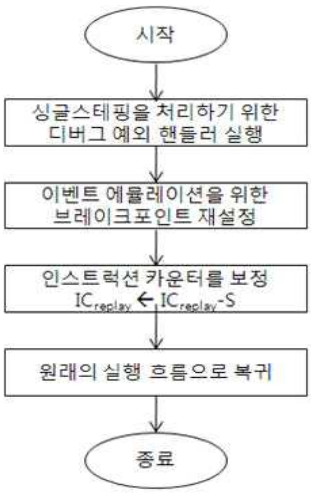
참고로 본 논문에서는 명령어 카운터 값 IC_i가 프로세서의 실행과 함께 항상 증가(increment)하는 값으로 가정한다. 프로세서에 따라서는 명령어 카운터의 값이 감소(decrement)하도록 구현되어 있을 수 있으며, 또는 절대적인 명령어 개수가 아니라 임의의 구간에서 실행된 명령어의 상대적인 개수를 이용하는 방법도 가능하다. 그러나 이러한 경우에 있어서 표현 방법의 차이는 있지만 명령어 카운터의 실질적인 의미는 동일하다. 편의상 본 논문에서는 명령어 카운터 값 IC_i가 항상 증가(increment)하는 경우만을 고려한다.

2.2 리플레이 모드에서의 인터럽트 재현

이벤트 이력에 기록된 인터럽트들은 디버깅 예외를 이용



(그림 2) 브레이크포인트를 이용한 이벤트 재현



(그림 3) 싱글스테핑과 브레이크포인트 재설정

하여 재현된다. 즉, 인터럽트가 재현되어야 할 시점에 디버깅 예외가 발생하도록 미리 준비를 해두고, 예외가 발생하면 이벤트 이력의 내용을 참고하여 인터럽트를 에뮬레이션하도록 한다. (그림 2)와 (그림 3)은 디버깅 예외를 이용하여 인터럽트를 재현하는 상세한 과정을 보여준다.

이벤트 이력에 기록된 인터럽트 E_i를 에뮬레이션하는 과정은 다음과 같다. 우선 E_i의 프로그램 카운터 값 PC_i를 브레이크포인트(breakpoint) 레지스터에 설정하도록 한다. 이후에 프로세서의 실행이 설정된 프로그램 카운터 값에 이르면 예외(exception)가 발생하게 되며, 예외 핸들러에서는 리플레이를 위해 삽입된 코드를 실행한다.

예외 핸들러에서는 예외(exception)가 발생한 시점의 명령어 카운터 값 IC_{replay}와 이벤트 이력에 기록된 명령어 카운터 값 IC_i를 비교하여 이벤트 에뮬레이션의 수행 여부를 결정한다. 만약 명령어 카운터 값에 존재할 수 있는 최대 오차를 Δ라 할 때 $|IC_{replay} - IC_i| \leq \Delta$ 이면 이벤트 에뮬레이션을 실행하도록 한다. 만약 $|IC_{replay} - IC_i| > \Delta$ 이면 아직 이벤트를 에뮬레이션할 시점이 아님을 의미하는데, 이 경우 이후에 프로세서가 다시 PC_i를 방문할 때 동일한 검사가 수행될 수 있도록 준비시켜 두어야 한다. 이를 위해 프로세서를 원래의 실행 흐름으로 복귀시키기 전에 PC_i를 다시 브레이크포인트로 지정하여야 한다.

여기서 유의해야 할 점은 현재의 예외 핸들러에서 PC_i를 브레이크포인트로 지정할 경우 예외가 반복적으로 발생하는 현상이 있을 수 있다는 점이다. 이러한 현상은 프로세서가 브레이크포인트에 지정된 프로그램 카운터 값의 명령어를 실행하기 직전에 예외를 발생시키도록 설계된 경우에 발생할 수 있다. 이러한 문제를 피하기 위해 싱글스테핑(single-stepping) 기능을 활용하는 것이 가능하다. 즉, 현재의 예외 핸들러에서 PC_i를 브레이크포인트로 지정하는 대신 프로세서를 싱글스테핑(single-stepping) 모드로 설정하고, 싱글스테핑 예외를 처리하는 핸들러에서 PC_i를 다시 브레이크포인트로 재설정하도록 한다. 싱글스테핑(single-stepping) 모드에서는 프로세서가 명령어를 수행할 때마다 예외가 발생하며, 따라서 원래의 실행 흐름으로 복귀한 프로세서는 그 다음 명령어를 수행하자마자 예외를 발생시키고 예외 핸들러를 실행한다.

2.3 명령어 카운터의 오차 보정

위에서 밝힌 바와 같이 프로세서가 제공하는 명령어 카운터에는 약간의 오차가 있을 수 있으며, 게다가 리코드 모드에서 실행되는 인터럽트 핸들러와 리플레이 모드에서 실행되는 인터럽트 에뮬레이션 하는 코드에서 실행되는 명령어의 개수가 서로 다르다는 점도 오차의 원인으로 작용한다.

리코드 모드의 인터럽트 핸들러에서 실행되는 명령어의 개수를 P, $|IC_{replay} - IC_i| \leq \Delta$의 경우 예외 핸들러에서 이벤트 에뮬레이션을 포함하여 실행하는 명령어의 개수를 Q, $|IC_{replay} - IC_i| > \Delta$의 경우 예외 핸들러에서 이벤트를 에뮬레이션 하지 않고 싱글스테핑 모드로 설정하고 원래의 실행 흐름으로 복귀하는 동안 실행된 명령어의 개수를 R, 싱글스테핑 예외를 처리하는 예외 핸들러에서 실행되는 명령어의 개수를 S라 하자. 세 가지 경우를 구분하여 오차를 보정하여야 하는데 각각의 경우를 살펴보면 다음과 같다. 우선, $|IC_{replay} - IC_i| \leq \Delta$을 만족하는 경우 먼저 인스트럭션 카운터 IC_{replay}의 값을 IC_i로 설정하여 이전에 발생했을 수 있는 오차를 제거한다. 그런 후에 이벤트 에뮬레이션이 끝나고 원래의 실행 흐름으로 복귀하는 시점에 IC_{replay}의 값을 $IC_{replay}+(P-Q)$으로 설정하도록 한다. $|IC_{replay} - IC_i| > \Delta$이어서 이벤트를 에뮬레이션 하지 않고 싱글스테핑 모드로 설정하고 복귀하는 경우에는 IC_{replay}의 값을 $IC_{replay}-R$으로 설정

한다. 그리고 싱글스테핑 예외 핸들러가 실행되는 경우에는 IC_{replay}의 값을 IC_{replay}-S로 설정하여야 한다.

3. 구현 및 실험 결과

본 연구에서는 위에서 제안한 방법을 Intel Core2 Duo 6600 2.4GHz와 uC/OS-II에 적용하여 구현하였다. 사용된 하드웨어에서 사용하는 I/O 장치로는 타이머, 키보드, FDD 이며, 이 장치들로부터 발생하는 인터럽트를 기록하기 위해 uC/OS-II의 인터럽트 핸들러에 (그림 1)과 같이 동작하는 코드를 삽입하였다. 또한 uC/OS-II의 예외 핸들러에는 (그림 2)와 (그림 3)과 같이 리플레이를 실행하는데 필요한 코드를 삽입하였다. 참고로 x86 아키텍처에서는 명령어 카운터와 같은 성능 카운터를 접근하기 위해 MSR(Model Specific Register)를 제공하며, 또한 디버그 레지스터를 제공하여 브레이크포인트와 싱글스테핑 기능을 이용하는 것이 가능하다.

우선 프로세서가 제공하는 TSC(timestamp counter)를 이용하여 리코드 단계에서 타이머 인터럽트를 기록하는데 걸리는 시간을 측정하였으며, 그 결과 평균 회당 0.44us 정도가 소요되는 것으로 나타났다. 또한 DivX 플레이어에 리코드 기능을 넣은 경우와 그렇지 않은 경우를 실행하여 시간을 측정한 결과를 <표 1>에 정리하였다. 각각 10개의 프레임을 디코딩하는데 소요되는 시간을 측정하였으며, 필요한 파일데이터는 미리 메인메모리에 적재하여 실험하였다. 따라서 별도의 I/O 이벤트가 발생하지 않는 가운데 타이머 인터럽트만을 기록하고 재생하도록 실험이 진행되었다. 그 결과는 <표 1>에 정리되었으며, 이를 통해 타이머 인터럽트의 기록을 포함하여 리코드 작업에 의한 전반적인 오버헤드가 매우 작음을 볼 수 있다.

〈표 1〉 리코드 오버헤드 측정

소프트웨어 구성	실행시간 (ms)
uC/OS-II + DivX	608.2547
uC/OS-II + DivX + Record	608.2595

4. 결 론

본 논문에서는 프로세서 하드웨어가 제공하는 성능 카운터와 디버깅 기능을 이용하여 인터럽트의 발생 시점을 정확하게 기록하고 재현하는 방법을 제안하였다. 제안하는 방법은 기존의 방법에 비해 구현이 매우 용이할 뿐만 아니라 실행 오버헤드도 크게 줄일 수 있다는 점에서 매우 유용할 것으로 판단되며, 실험 결과도 이를 잘 뒷받침하고 있다.

참 고 문 헌

[1] D. Wittie, “Debugging distributed c programs by real time reply,” In Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, pages 57 - 67, 1988.

[2] T. J. LeBlanc and J. M. Mellor-Crummey, “Debugging parallel programs with instant replay,” IEEE Transaction on Computers, 36(4):471 - 482, 1987.

[3] Michiel Ronsse and Koen De Bosschere, “RecPlay: a fully integrated practical record/replay system,” ACM Transactions on Computer Systems, 17(2): 133-152, 1999.

[4] J. Choi and H. Srinivasan, “Deterministic replay of Java multithreaded applications,” In Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pages 48 - 59, 1998.

[5] J. H. Slye and E. Elnozahy. Support for software interrupts in log-based rollback-recovery. IEEE Transactions on Computers, 47(10):1113 - 1123, 1998.

[6] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth, “Accuracy of Performance Counter Measurements,” In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2009.

[7] S. Narayanasamy, G. Pokam, and B. Calder, “BugNet: Continuously recording program execution for deterministic replay debugging,” In Proceedings of the 32nd International Symposium on Computer Architecture, 2005.

맹 지 찬

e-mail : jcmaeng@rtcc.hanyang.ac.kr
2004년 한양대학교 응용화학공학부(학사)
2006년 한양대학교 소프트웨어전공 (공학석사)
2006년～현 재 한양대학교 전자컴퓨터통신학과 박사과정
관심분야 : 실시간 시스템, 운영체제, 소프트웨어 테스팅

유 민 수

e-mail : msryu@hanyang.ac.kr
1995년 서울대학교 제어계측공학과(학사)
1997년 서울대학교 전기공학부(공학석사)
2002년 서울대학교 전기공학부(공학박사)
1999년～2001년 Inus Technology 연구원
2001년～2002년 자동화시스템연구소 연구원
2003년～현 재 한양대학교 컴퓨터공학부 부교수
관심분야 : 실시간 시스템, 운영체제, 멀티코어 프로그래밍