동적 코드 변환 기법을 이용한 디바이스 드라이버의 고장 분리 기술

임병홍, 김지홍, 엄영익
성균관대학교 정보통신공학부
e-mail : {dd8562, ezjjilong, yieom}@ece.skku.ac.kr

Device Driver Fault Isolation using Binary Translation Technology

Byoung Hong Lim, Jeehong Kim, Young Ik Eom
School of Information and Communication Engineering, Sungkyunkwan University

요 약

디바이스 드라이버는 커널의 대부분을 차지하기 때문에 디바이스 드라이버에 문제가 발생하면 시스템에 심각한 영향을 미치게 된다. 따라서 디바이스 드라이버의 고장 분리 기술은 운영체제의 신뢰도 향상을 위해서 매우 중요하다. 동적 코드 변환 기법(Binary Translation)은 기계어 코드의 수준에서 기존의 디바이스 드라이버의 명령어 집합을 다른 명령어 집합으로 변환하여 실행하도록 하는 기법이다. 이 기법을 통해 우리는 각 명령어의 변환 과정에서 디바이스 드라이버의 모든 행위를 감시할 수 있다. 따라서 동적 코드 변환기법은 디바이스 드라이버의 고장을 분리하며 악의적인 메모리 접근을 제한하는 장점을 가지고 있다. 또한 커널 코드의 수정과 디바이스 드라이버의 수정이 요구되지 않는다. 이 논문에서 우리는 동적 코드 변환 기법을 설계하고 구현하였다. 그리고 동적 코드 변환 기법을 이용한 몇 가지 실험을 통해, 디바이스 드라이버를 수행 시 발생하는 오버헤드와 고장 분리 가능 여부를 평가해 보았다.

1. 서론

대부분의 모노리딕 운영체제는 디바이스에 쉽고 빠르게 접근하기 위해 디바이스 드라이버를 커널 내에 포함하고 있다. 이러한 디바이스 드라이버는 운영체제의 대부분을 차지하기 때문에 디바이스 드라이버의 고장은 운영체제에 큰 영향을 준다.

실제로 Linux 코드의 70% 이상이 디바이스 드라이버로 이루어져 있으며, Windows XP 내에서 발생하는 에러의 85% 이상이 디바이스 드라이버의 고장 때문이라는 연구 결과도 있다[1]. 따라서 디바이스 드라이버의 고장이 운영체제에서 분리된다면, 운영체제의 안정성과 신뢰성은 많이 향상될 것이다.

본 논문에서 우리는 동적 코드 변환 기법을 소개한다. 동적 코드 변환 기법은 기계어 코드의 수준에서 특정 명령어 집합을 다른 명령어 집합으로 변환하여 새로운 주소 공간에서 실행하는 기법이다. 이러한 변환 과정에서 우리는 특정 명령어를 삽입하거나 대체하여 디바이스 드라이버의 고장이나 잘못된 메모리 접근이 발생하면 디바이스 드라이버의 실행을 중지한다. 동적 코드 변환 기법은 C 언어로 구현되었으며, 디바이스 드라이버의 수정이나 커널 코드의 수정이 요구되지 않는다.

우리는 몇 가지 실험을 통하여 동적 코드 변환 기법을 평가하였다. 우선 고장의 종류를 정의 후 문자 디바이스 드라이버에 고장을 삽입하여 동적 코드 변환 기법이 삽입된 고장을 모두 발견하고 분리했는지 실험하였다. 그리고 디바이스 드라이버를 기존 커널과 동적 코드 변환 기법에서 각각 실행하여 발생하는 성능의 오버헤드를 측정한 후 차이를 비교해보았다. 그 결과, 동적 코드 변환 기법에서는 정의된 모든 고장을 발견하고 분리할 수 있음을 발견하였다. 그리고 동적 코드 변환 기법에서 디바이스 드라이버를 수행하였을 때, 전체적인 성능의 오버헤드는 기존 커널보다 약간 증가하였지만 큰 차이를 보이지 않았다.

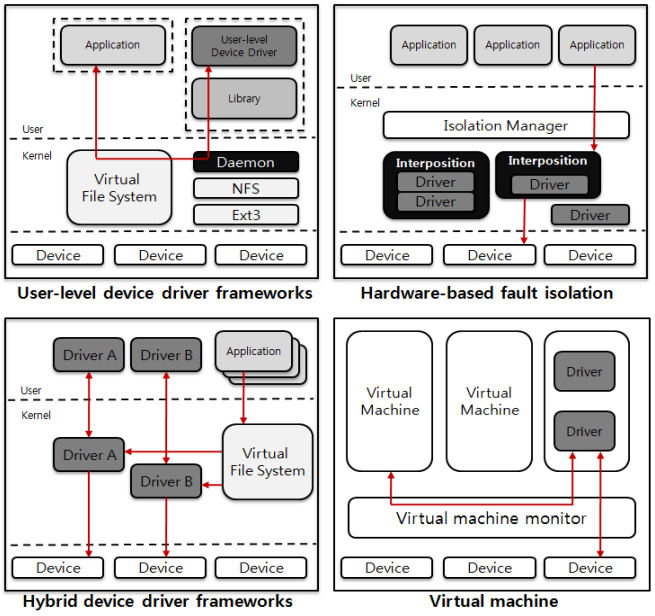
본 논문의 구성은 다음과 같다. 2 장에서는 기존의 디바이스 드라이버 고장 분리 기술들에 대해 알아보고, 3 장에서는 본 논문에서 제안하는 동적 코드 변환 기법의 구성에 대해서 기술한다. 그리고 4 장에서는 동적 코드 변환 기법의 구현 과정을 설명한다. 5 장에서는 디바이스 드라이버의 고장 감지 기능 및 기존 커널과 부분 동적 코드 변환 기법의 성능 차이를 평가하며, 6 장을 통해 결론을 맺는다.

2. 관련연구

디바이스 드라이버의 고장 분리를 위한 연구는 유

본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT 연구센터 지원사업의 연구결과로 수행되었음" (NIPA-2010-(C1090-1021-0008))

저 레벨 디바이스 드라이버 프레임워크, 하드웨어 기반 분리 기술, 하이브리드 디바이스 드라이버 프레임워크와 가상 머신이라는 4 개의 큰 카테고리로 나눌 수 있다. 그림 1 은 관련 연구들의 시스템 구조와 실행 흐름을 보여준다.



(그림 1) 유저 레벨 디바이스 드라이버 프레임워크, 하드웨어 기반 분리 기술, 하이브리드 디바이스 드라이버 프레임워크와 가상 머신의 시스템 구조와 실행 흐름

2.1 유저 모드 디바이스 드라이버 프레임워크

유저 레벨 디바이스 드라이버 프레임워크 방식은 디바이스 드라이버를 커널 레벨이 아닌 유저 레벨에서 실행한다[2][3][4]. 이러한 방식에는 디바이스 드라이버의 고장을 분리한다는 점뿐 아니라 유저 레벨에서 디바이스 드라이버의 구현이 가능하게 하며, 쉽게 디버깅이 가능하다는 장점이 존재한다. 그러나 유저 레벨 디바이스 드라이버 프레임워크는 커널 코드의 수정과 디바이스 드라이버의 수정을 필요로 한다. 게다가 유저 레벨과 커널 레벨간의 지속적인 데이터 교환으로 인해 성능의 오버헤드가 심각하게 증가한다는 문제점이 존재한다.

2.2 하드웨어 기반 분리 기술

하드웨어 기반 분리 기술은 유저 레벨 디바이스 드라이버로 인한 성능의 저하를 해결하기 위해 연구되었다[5][6][7]. 하드웨어 기반 분리 기술은 디바이스 드라이버의 고장을 분리하며 성능 오버헤드를 줄인다. 하지만 커널의 수정을 필요로 하며, 하드웨어에 의존한다는 문제점을 가지고 있다

2.3 하이브리드 디바이스 드라이버 프레임워크

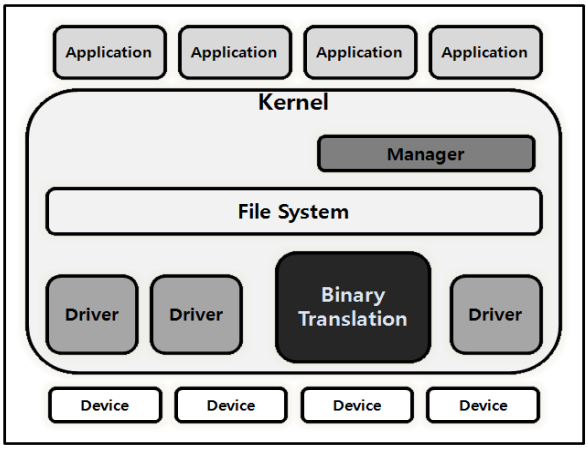
하이브리드 디바이스 드라이버 프레임워크는 유저 레벨과 커널 레벨의 디바이스 드라이버를 모두 사용하여 각각의 장점을 이용하는 것이 목적이다[8][9]. 하이브리드 디바이스 드라이버 프레임워크는 성능의 저하가 적고, 유저 레벨에서의 디바이스 드라이버 구현과 디버깅이 가능하다는 장점을 가지고 있다. 하지만 디바이스 드라이버를 유저 레벨과 커널 레벨로 나누어야 하며, 그에 따른 수정이 필요하다.

2.4 가상머신

가상 머신을 이용한 방식은 기존의 유저 레벨 또는 커널 레벨을 이용한 고장 분리 기술과는 다른 방식을 사용한다[10][11]. 이 방식에서 모든 디바이스 드라이버들은 하나의 독립된 가상 머신에 설치되고 실행된다. 따라서 가상 머신 내에서 발생한 디바이스 드라이버의 고장은 다른 가상 머신에게 아무런 영향을 주지 않는다. 이 방식은 디바이스 드라이버의 고장을 완벽히 분리할 수 있다는 장점이 있다. 그러나 디바이스 드라이버들을 위한 새로운 가상 머신을 생성해야 하며, 커널과 디바이스 드라이버의 수정이 필요하다는 단점이 있다.

3. 설계

이번 장에서는 본 논문에서 제안하는 동적 코드 변환 기법에 대해서 설명한다. 디바이스 드라이버의 고장 분리를 위해서 동적 코드 변환 기법은 크게 Manager 와 Binary Translation 의 두 개의 컴포넌트로 구성된다. 그림 2 는 Linux 내부에 동적 코드 변환 기법을 구성하는 각 컴포넌트들을 보여준다.



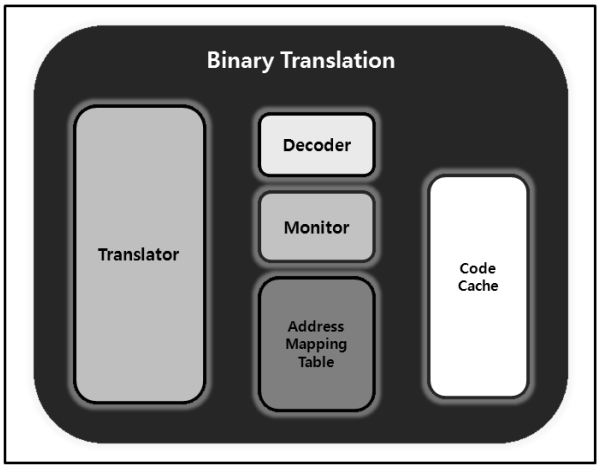
(그림 2) 동적 코드 변환 기법을 구성하는 Manager 와 Binary Translation

3.1 Manager

Manager 는 Binary Translation 을 사용하여 디바이스 드라이버를 실행할 것인지 아닌지 제어하는 역할을 한다. 만약 어플리케이션이 특정 디바이스 드라이버의 고장 분리 기능을 Manager 에게 요청한다면, Manager 는 어플리케이션에서 요청한 디바이스 드라이버의 정보를 저장한다. 그리고 어플리케이션이 해당 디바이스 드라이버의 수행을 커널에 요청하면, Manager 는 Binary Translation 을 호출하여 요청된 디바이스 드라이버를 수행하도록 한다.

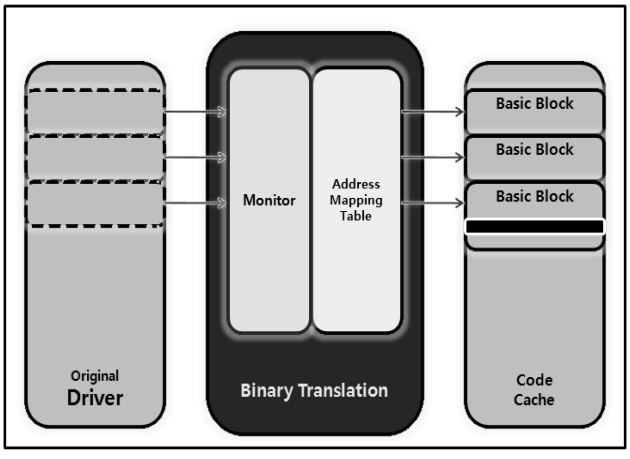
3.2 Binary Translation

Binary Translation 은 기존의 디바이스 드라이버의 명령어 집합을 다른 명령어 집합으로 변환하여 실행하기 위해서 그림 3 과 같은 5 개의 컴포넌트로 구성된다.



(그림 3) Binary Translation 을 구성하는 5 개의 컴포넌트

Translator 는 기계어 수준의 기존 디바이스 드라이버 명령어 집합을 Basic Block 단위로 Decoder 로 분석하여 Code Cache 로 복사한다. 이 때 실제 디바이스 드라이버 실행 시, 기존 디바이스 드라이버의 명령어가 아닌 Code Cache 의 명령어가 수행될 수 있도록 Address Mapping Table 에 매핑 주소를 기록한다. 이렇게 하나의 Basic Block 이 Code Cache 로 복사되면 해당 코드를 실행한다. Monitor 는 기존 디바이스 드라이버의 코드가 복사되는 과정에서 의심되는 특정 명령어나 특정 메모리 주소 접근 시, 기존 디바이스 드라이버의 코드에 새로운 코드를 삽입하거나 기존 코드를 대체하여 디바이스 드라이버의 고장을 발견하고 분리하도록 한다. 그림 4 에서는 Monitor 의 동작 과정을 보여준다.



(그림 4) 특정 명령어를 감시하기 위해 Monitor 에 의해 감시 코드가 삽입되는 과정

4. 구현

우리는 Linux Kernel 2.6.31 에서 동적 코드 변환 기법을 구현하였다. 이러한 동적 코드 변환 기법의 구현에는 커널의 수정이나 디바이스 드라이버의 수정은 필요하지 않았다.

우리는 동적 코드 변환 기법을 위해서 Manager 와 Binary Translation 컴포넌트를 각각 구현하였다. 매니저에는 어플리케이션에서의 고장 분리 기능을 요청받기 위해서 새로운 시스템 콜이 추가되었다. 그리고 Manager 는 Binary Translation 의 실행을 위해서, 요청된 디바이스 드라이버의 정보를 저장한다.

Binary Translation 은 기존 디바이스 드라이버의 명령어 집합을 다른 명령어 집합으로 변환하기 위해 Decoder 와 Translator, Address Mapping Table 로 구현되어 있다. 또한 Monitor 는 Code Cache 로 복사되는 모든 명령어들을 감시하여 고장을 발견한다. 그리고 메모리를 읽고 쓰는 행위를 감시하여 잘못된 메모리 접근을 발견한다. 만약 고장이나 잘못된 메모리 접근을 발견하면, Monitor 는 Binary Translation 의 동작을 멈추고 디바이스 드라이버를 요청한 어플리케이션에게 에러 메시지를 보낸다.

위와 같은 동적 코드 변환 기법에서는 한 가지 문제점이 존재한다. 동적 코드 변환 기법이 실제 디바이스 드라이버 내의 코드가 아닌 디바이스 드라이버가 호출하는 커널 함수의 코드까지 모두 Code Cache 로 복사하고 수행하여, 성능 저하를 유발한다. 커널 함수는 이미 검증되어 있는 코드로써 Monitor 의 감시를 받을 필요가 없다. 따라서 우리는 이 문제점을 개선하기 위해 디바이스 드라이버에서 호출하는 커널 함수의 코드가 Code Cache 로 복사되지 않고 바로 실행될 수 있도록 하였다. 그리고 동적 코드 변환 기법에서는 호출된 커널 함수의 반환 값만을 받아서 디바이스 드라이버의 다음 명령어를 계속 수행하도록 하였다.

5. 실험 및 평가

5.1 평가 환경

우리의 평가 시스템은 Intel Core2 Quad 2.40GHz CPU 와 2 GB 메모리로 구성된다. 현재 동적 코드 변환 기법은 문자 디바이스 드라이버의 고장 분리를 가능하게 한다. 따라서 우리는 /dev/mem 와 /dev/random 문자 디바이스 드라이버를 통해 실험 및 평가를 하였다.

5.2 디바이스 드라이버의 고장

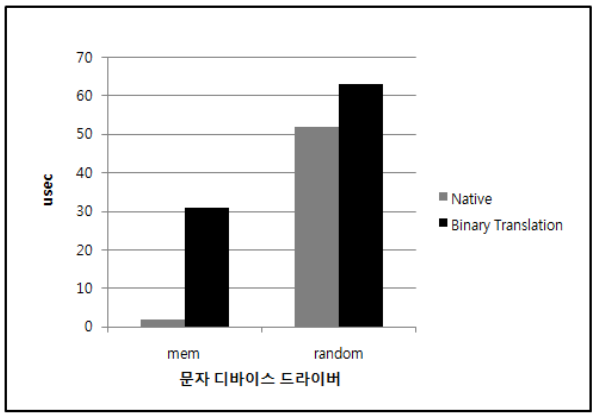
우리는 동적 코드 변환 기법을 평가하기 위해서 기존의 디바이스 드라이버에 고장을 주입하는 연구들을 참고하여 고장의 종류를 정의 하였다[6][12]. 고장의 종류는 메모리 접근 위반, 실행 코드에서 발생하는 오류로 나눌 수 있다. 표 1 은 정의된 고장들의 종류와 분리 가능 여부를 보여준다.

<표 1> 정의된 고장의 종류와 고장에 따른 행위

고장 종류	활동	분리 가능여부
실행 코드 에러	Null pointer	√
	Divide by zero	√
	Infinite loop	√
메모리 접근 위반	Illegal memory access	√
	Source & Destination	√

5.3 성능 평가

우리는 문자 디바이스 드라이버를 실행하면서 기존 커널과 동적 코드 변환 기법 간의 성능 차이를 측정하고 비교해보았다. 일정한 크기의 데이터를 mem 과 random 문자 디바이스 드라이버를 통해 읽으며, 각각의 시간을 측정하였다. 그 결과는 그림 5 를 통해서 볼 수 있듯이 큰 성능의 차이를 볼 수 없었다.



(그림 5) 기존 커널(Native)과 동적 코드 변환 기법(Binary Translation)에서 문자 디바이스 드라이버를 실행한 결과

6. 결론

본 논문에서는 디바이스 드라이버의 고장을 분리시켜 운영체제의 신뢰성과 안전성을 향상하기 위해서 동적 코드 변환 기법을 제안하였다. 동적 코드 변환 기법은 운영체제 내에서 디바이스 드라이버의 고장을 분리시켰다. 또한 디바이스 드라이버의 모든 메모리 접근을 감시하여 잘못된 접근을 방지함으로써 운영체제의 신뢰성과 안전성을 향상시켰다. 이러한 동적 코드 변환 기법은 커널의 수정과 디바이스 드라이버의 수정을 필요로 하지 않는다. 그리고 성능상의 오버헤드가 거의 없다.

본 논문에서는 동적 코드 변환 기법의 성능을 평가하기 위해서 고장의 종류를 정의하였으며, 실제로 고장들을 문자 디바이스 드라이버에게 주입하여 실험하였다. 그 결과, 주입된 모든 고장들은 동적 코드 변환 기법에 의해 발견되었으며, 해당 디바이스 드라이버는 정지되었다. 그리고 시스템 전체에는 아무런 영향을 주지 않아 운영체제의 신뢰성과 안전성의 향상이 입증되었다.

향후 과제로 우리는 문자 디바이스 드라이버만이 아니라 블록과 네트워크 디바이스 드라이버에서 동적 코드 변환 기법이 적용될 수 있도록 할 것이다. 그리고 더 많은 고장을 정의하여 발견 및 분리될 수 있도록 할 것이다. 또한, 우리는 디바이스 드라이버의 고장이 발생하면, 고장을 분리하는 것뿐 아니라 디바이스 드라이버의 실행을 복구하는 기능을 동적 코드 변환 기법에 추가할 예정이다.

참고문헌

[1] V. Orgovan and M. Tricker, “An introduction to driver quality,” Microsoft WinHec 2004 Presentation DDT301, 2003.

[2] J. Elson, “FUSD: A Linux framework for user-space devices,” User manual for FUSD 1.0, 2004.

[3] P. Chubb, “Linux kernel infrastructure for user-level device drivers,” Linux Conference Adelaide, Jan. 2004.

[4] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone and G. Heiser, “User-level device drivers: Achieved performance,” Journal of Computer Science and Technology, 2005.

[5] E. Witchel, J. Rhee, K. Asanovic, “Mondrix: Memory Isolation for Linux using Mondriaan Memory Protection,” Proc. The 20th ACM Symposium on Operating Systems Principles, 2005.

[6] M. M. Swift, B. N. Bershad, H. M. Levy, “Improving the reliability of commodity operating systems,” ACM Transactions on Computer Systems, vol. 23, Feb. 2005.

[7] M. M. Swift, M. Annamalai, B. N. Bershad, H. M. Levy, “Recovering device drivers,” ACM Transactions on Computer Systems, vol. 24, pp. 333-360, 2006.

[8] V. Ganapathy, A. Balakrishnan, M. M. Swift, S. Jha, “Microdrivers: A new architecture for device drivers,” Proc. Hot Topics in Operating Systems (HotOS), San Diego, CA, May 2007.

[9] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, S. Jha, “The design and implementation of microdrivers,” 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’08), Seattle, WA, March 2008.

[10] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, M. Williamson, “Safe hardware access with the Xen virtual machine monitor,” Conf. 1st workshop on Operating System and Architec-tural Support for the on-demand IT Infrastructure (OASIS), Boston, MA, Oct. 2004.

[11] J. LeVasseur, V. Uhlig, J. Stoess, S. Gotz, “Unmodified device driver reuse and improved system dependability via virtual machines,” Conf. Operating Systems Design and Implementation (OSDI), San Francisco, CA, Dec. 2004.

[12] W. T. Ng and P. M. Chen, “The design and verification of the rio file cache,” IEEE Transactions on Computers, vol. 50, pp. 1-16, Apr. 2001.