트랩 기반의 동적 커널 재구성 기법

손재웅*, 김영필*, 유혁*
*고려대학교 컴퓨터학과
e-mail : jwson@os.korea.ac.kr

Dynamic Kernel Reconfiguration Mechanism based on Trap

Jae-Woong Son*, Young-Pill Kim*, Hyuck Yoo*
*Dept. of Computer Science & Engineering, Korea University

요 약

최근 사용자의 요구가 다양해지는 반면, 기존 운영체제는 정적인 구조를 갖기 때문에 이러한 요구를 효율적으로 반영하기 어렵다. 따라서 운영체제를 동적으로 재구성하여 사용자의 요구를 만족시키는 매커니즘에 관한 연구들이 진행되어 왔다. 그러나 기존의 커널을 재구성하는 기법들은 하드웨어 의존적인 문제점을 가지고 있다. 본 논문에서는 기존 연구들의 문제점을 해결할 수 있는 트랩 기반의 커널 재구성 기법을 제안하고 리눅스 커널에 구현하였다.

1. 서론

오늘날 네트워크 기술의 발전과 컴퓨터의 성능 향상으로 사용자의 요구가 다양해지고 있다. 이러한 요구를 효율적으로 만족시키기 위해서는 컴퓨터 하드웨어와 사용자간의 인터페이스를 담당하는 운영체제의 기능들이 유연성을 갖고 교체 혹은 확장될 수 있어야 한다.

그러나 기존 운영체제는 다양한 사용자의 요구와 하드웨어 플랫폼을 지원하는 기능들이 밀집된 하나의 실행 이미지를 갖고 있다. 따라서 하드웨어 구성이 바뀌었을 때나 사용자의 요구를 반영할 때 커널 소스를 다시 컴파일하여 실행 이미지를 새로 생성해야 한다. 이렇게 커널의 기능들이 밀집되어 복잡한 구조를 갖는 운영체제들은 다양한 사용자의 요구 및 하드웨어의 지원에 대한 추가 및 재구성이 용이하지 못하다.

그래서 커널을 동적으로 재구성하여 기존의 운영체제가 가지고 있는 문제를 해결하려는 연구들이 진행되고 있다. 동적 재구성에 관한 연구인 kernist 와 GILK 는 분기 명령어를 커널 내부 함수에 삽입하여 동적으로 재구성하는 기법을 제안하였다. 그러나 이들 연구에서 제안하는 기법은 하드웨어에 의존적인 문제를 가지고 있다.

본 논문에서는 기존 커널 재구성의 연구들이 가지는 문제점을 해결하기 위하여 트랩을 사용한 커널 재구성 기법을 제안한다.

본 논문의 구성은 다음과 같다. 먼저 2 장에서는 커널을 동적으로 재구성하는 연구들에 관하여 알아보고 3 장에서는 커널 재구성 방법을 제안한다. 4 장에서는 제안된 기법을 구현한 내용에 대하여 설명하고, 5 장에서는 결론을 기술한다.

2. 관련 연구

본 장에서는 실행 중인 커널에 임의의 코드를 특정 커널 코드 영역에 삽입하여 재컴파일 없이 기능을 확장하고 성능을 최적화시킬 수 있는 커널 동적 인스트루먼테이션(Dynamic Kernel Instrumentation) 기법들에 대하여 설명한다.

2.1 Kerninst

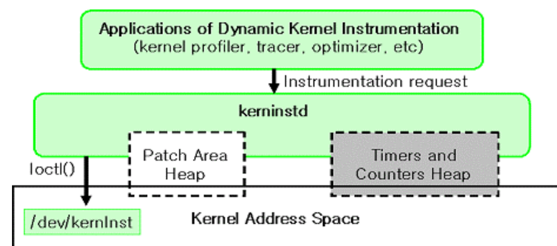

그림 1 kerninst 시스템 구조

Kernist[1]는 솔라리스(Solaris) 커널을 위한 동적 인스트루먼테이션 프레임워크이다. [그림 1]은 kernist 의 구조를 보여준다. 동적 인스트루먼테이션란 동적으로 생성된 일련의 코드를 커널 코드 영역 내의 특정 위치에 삽입(splicing)하는 과정이라고 말할 수 있다.

Kerninstd 는 어플리케이션의 인스트루먼테이션 요청에 따라 kerninst driver 의 도움을 받아 패치할 코드를 patch area heap 에 할당한다. 그리고 Kerninstd 는 어플리케이션의 요청을 수행하기 전에 인스트루먼테이션할 커널 함수들의 상호 의존에 대한 정보를 필요로 하기 때문에 제어 흐름 그래프와 콜 그래프 구성한다. 그 다음, 패치 영역 힙(patch area heap)에 저장된 코드를 바탕으로 코드 삽입(Code Splicing)과 코드 교환(Code Replacement)을 수행한다. 코드 삽입과 코드 교환은 kerninst 에서 제시한 동적 커널 인스트루먼테이션 프리미티브(primitive)이다.

코드 삽입은 인스트루먼테이션 포인트에 분기 명령어를 덮어써서 패치 영역 힙으로 분기하도록 한다. 인스트루먼테이션 포인트는 커널 코드 내에서 코드 삽입이 수행될 위치이다. 패치 코드는 그림 2 에서 보여주는 것들을 포함한다. 코드 교환은 새로운 버전이 시작하는 특정 함수의 진입 점을 인스트루먼테이션 한다.

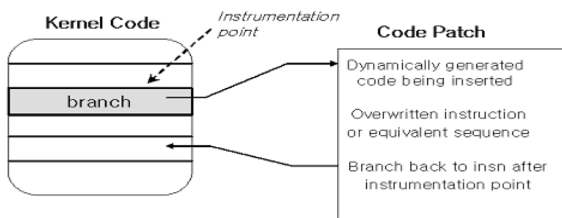

그림 2 코드 splicing

Kerninst 는 사용자 레벨(user-level)에서 커널의 상태를 알아볼 수 있고, 커널의 기능을 바꿀 때 커널을 멈추지 않고 동적으로 바꿀 수 있는 장점을 갖는다. 그러나 단점으로는 Kerninst 는 Sun UltraSpac 구조를 대상으로 구현되었기 때문에 명령어 구조가 다른 Intel X86 환경에서는 사용할 수 없다.

2.2 GILK

GILK[2]는 kernist 의 동적 커널 인스트루먼테이션 기법을 Intel X86 환경에 적용하기 위한 프레임워크이다. Intel X86 구조에서는 명령어 개수가 많고 길이가 가변적이기 때문에 코드 삽입을 하는데 문제가 생긴다. 기본적으로 커널 함수들은 분기하지 않고 실행하는 명령어들의 집합인 베이직 블록(Basic Block)으로 구성되어 있다. 이 베이직 블록들 중 하나가 크기가 4 바이트 이하일 경우 5 바이트 분기 명령어를 사용하게 되면 베이직 블록의 경계를 넘어가 다른 명령어를 덮어쓰기 때문에 문제가 발생한다. 따라서 GILK 는 로컬 바운싱(Local Bouncing) [2]기법을 사용하였다.

다음 그림 3 은 GILK 의 코드 삽입을 보여준다. 이 기법은 앞에서 설명한 kerninst 의 방법과 같다. 다른 점은 명령어의 길이가 가변적인 Intel X86 구조에서 코드 삽입을 할 경우 발생되는 문제를 해결할 수 있는 로컬 바운싱[2] 기법을 사용했다는 점이다.

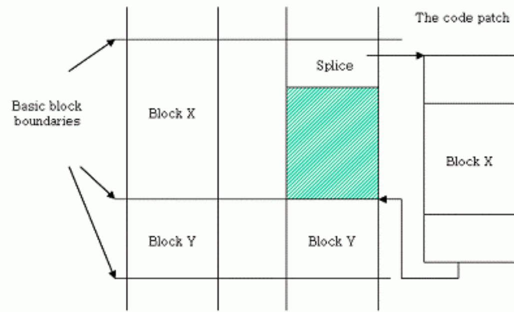

그림 3 GILK 의 Code Splicing

3. 커널 재구성 기법

기존 연구들은 커널 함수에 코드를 삽입하는 기법을 사용한다. 코드 삽입 기법은 새롭게 확장할 코드로 분기(Branching)하는 명령어를 인스트루먼테니션 지점에 덮어쓰는 것이다. 이러한 코드 삽입은 대상 하드웨어 플랫폼의 특성에 따라 다른 기법이 요구된다. 예를 들어 kerninst 인 경우 주 대상 플랫폼은 Sun UltraSpac 이다. Sun 플랫폼은 명령어 길이가 같은 RISC 구조이므로 코드 삽입을 할 때 확장할 코드로 분기하기 위해서 덮어쓰는 분기 명령어의 길이에 대한 고려가 필요 없다. 왜냐하면 모든 명령어의 길이가 일정하므로 삽입하는 분기 명령어가 다른 명령어를 덮어쓰지 않기 때문이다. 그러나 Intel X86 구조에서는 분기 명령어의 크기가 큰 문제가 된다. Intel X86 인 경우 명령어의 개수가 많고 명령어와 데이터의 길이가 일정하지 않기 때문이다. 따라서 Intel X86 플랫폼 기반인 GILK 에서는 로컬 바운싱[2]이라는 별도의 기법이 사용되었다.

본 연구에서는 기존의 동적 커널 인스트루먼테이션 기법의 문제점을 보완한 트랩 기반의 동적 커널 인스트루먼테이션(Trap-based Dynamic Kernel Instrumentation) 기법을 제안한다.

트랩 기반 커널 인스트루먼테이션의 핵심은 코드 삽입에 사용되는 분기 명령어로써 기존의 jmp 가 아닌 트랩 발생 명령어(Trap-generation Instruction)를 사용하는 것이다. 이러한 명령어들로 대표적인 것은 Intel 의 int 명령어가 있다. 본 연구에서 굳이 트랩 발생 명령어를 분기 명령어로써 채택한 이유는 다음의 몇 가지가 있다.

첫 번째는 특정한 하드웨어 플랫폼에 의존적이지 않고 독립성을 보장할 수 있다는 이유이다. 기존 연구인 GILK 에서는 Intel X86 의 명령어 길이가 가변적이기 때문에 인스트루먼테이션 지점의 명령어 길이가 분기 명령어의 길이보다 작을 때 분기 명령어가 다른 명령어까지 덮어쓰게 되는 문제가 발생한다. 따라서 GILK 에서는 하드웨어 플랫폼의 특성 때문에 발생하는 문제를 해결하기 위해 Local bouncing 기법을 사용하였다. 그러나 트랩 발생 명령어는 플랫폼에 무관하게 모두 지원된다.

두 번째는 코드 삽입에 직접적으로 요구되는 분기 명령어의 요구량이 작다는 것이다. 명령어 길이가 다를 수 있는 Intel x86 기반에서 패치 코드로 분기하기 위해 커널 메모리에 덮어쓰는 분기 명령어로써 jmp 를 사용하게 되면 5 바이트 미만의 명령어 코드를 덮어쓸 때 원하지 않는 명령어까지 손상시킬 위험성이 있다. 때문에 별도의 2 바이트 jmp 명령어를 반복적으로 사용하는 변칙적인 방법이 제시되었으나, 이는 불필요한 다수의 분기를 부가적으로 유발하게 된다. 이와는 달리 트랩 발생 명령어 int 는 2 바이트만을 차지하기 때문에 별도의 부가적인 분기가 필요하지 않으며 단 한번의 호출로 분기가 이루어지므로 복잡하지 않고 간단하다.

4. 커널 재구성 기법 구현 및 실험

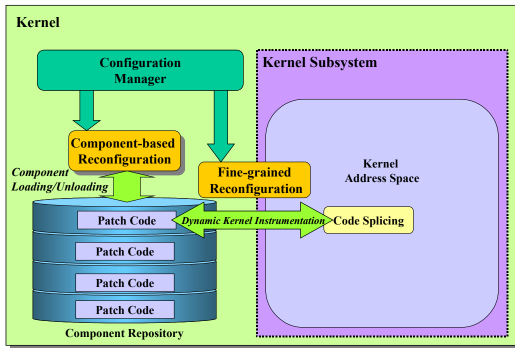

그림 4 커널 재구성의 프레임워크

본 연구에서 제안하는 커널의 재구성 프레임워크는 크게 두 부분으로 구성된다. 첫 번째는 명령어 단위의 재구성에 관련된 FR(Fine-grained Reconfiguration) 부분이다. 이 부분은 패치 코드로부터 얻어진 명령어 코드를 커널로 삽입하여 동적 커널 인스트루먼테이션을 가능하게 한다. 두 번째는 패치 코드와 패치 코드를 커널에 삽입할 때 필요한 정보들을 포함하는 컴포넌트를 관리하는 CR (Component-based Reconfiguration) 부분이다. 필요한 정보들에는 컴포넌트 ID, 인스트루먼테이션 지점, 심볼 테이블(Symbol Table)이 있다.

여러 커널 함수들이 동일한 패치 코드를 사용하려고 할 때나 패치 코드의 내용이 새로운 함수일 경우 이러한 정보들을 참조하여 CR 은 패치 코드의 사용 유무를 결정하고 커널의 런타임 심볼 테이블을 수정한다. FR 은 CR 의 작업이 종료한 후 사용할 수 있는 컴포넌트를 대상으로 커널 인스트루먼테이션을 수행한다.

커널 재구성 프레임워크의 전체적인 구조는 그림 4 와 같다.

4.1 Fine-grained Reconfiguration

트랩 기반의 동적 커널 인스트루먼테이션을 수행하는 FR 은 크게 preprocessing, 코드 삽입, 패치 코드 실행 이 세가지 단계로 진행된다. 다음 그림 5 는 커널 재구성을 수행하는 모듈과 모듈의 실행 흐름을 보여준다.

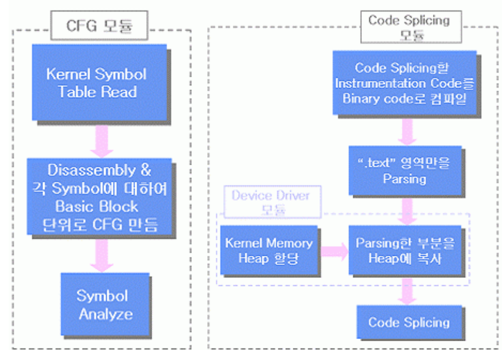

그림 5 트랩 기반 동적 커널 인스트루먼테이션 모듈

처음 단계인 preprocessing 에서는 재구성 하기에 앞서 커널 심볼들(kernel symbol) 사이의 상호 의존 관계를 대한 정보를 얻는다. 커널 심볼들 사이의 상호 의존 관계를 분명하게 보여주는 방법으로 상호 의존 관계를 표현할 수 있는 제어 흐름 그래프(Control Flow Graph)를 사용하였다.

제어 흐름 그래프는 각 커널 심볼을 연속으로 실행되는 어셈블리 명령어들의 집합인 베이직 블록과 에지(Edge)로 표현한다. 이 제어 흐름 그래프의 에지를 보고 커널 심볼들 간의 상호 의존성과 커널 심볼 내부의 베이직 블록간 상호 의존 관계를 알 수 있다. 상호 의존 관계를 분명하게 나타내는 제어 흐름 그래프를 생성하는 제어 흐름 그래프 모듈은 커널 심볼 테이블을 읽어서 각 커널 심볼들을 제어 흐름 그래프로 표현 할 수 있는지를 분석한다. 그런 다음, 각 커널 심볼에 대하여 역어셈블하고 베이직 블록단위로 나누어 제어 흐름 그래프를 생성한다. 다음 그림 6 은 제어 흐름 그래프를 보여준다.

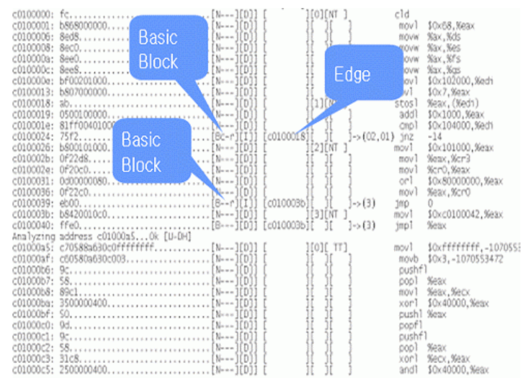

그림 6 제어 흐름 그래프

두 번째 단계인 코드 삽입에서는 앞 단계에서 생성된 제어 흐름 그래프를 기반으로 기존 커널 심볼을 동적으로 인스트루먼테이션한다. 트랩을 발생시킬 명령어 int 를 인스트루먼테이션 포인트에 덮어쓴다.

세 번째 단계로 동적 인스트루먼테이션한 커널 심볼을 수행시키면 트랩이 발생된다. 트랩이 발생하면 트랩 핸들러가 코드 패치의 명령어들을 실행시키고 트랩을 발생시킨 커널 심볼로 리턴한다.

4.2 Component-based Reconfiguration

FR 에서 사용되는 패치 파일은 CR 에 의해 관리되어지고 파일 내용은 커널 코드에서 단지 변수의 값을 바꾸는 것과 같은 간단한 연산을 하는 코드, 새로운 변수를 선언하여 값을 할당하여 조작하는 코드, 특정 기능을 담당하는 함수 전체를 추가하는 코드, 특정 커널 함수 안에서 다른 함수를 호출하는 코드 등 다양하게 구성될 수 있다. 이러한 패치 파일은 컴포넌트 이미지로 메모리에 저장된다.

컴포넌트 이미지는 컴포넌트를 구별하는데 사용되는 컴포넌트 ID, 컴포넌트의 인스트루먼테이션 지점에 대한 정보, 패치 파일의 내용, 컴포넌트가 차지하는 길이에 대한 정보, 새로운 변수나 함수를 추가할 때 커널의 런타임 심볼 테이블(Runtime Symbol Table)을 재배치(Relocation)하는 작업이 필요한데 이를 위한 심볼 테이블을 갖고 있다. 다음 그림 7 은 메모리상의 컴포넌트 이미지를 보여준다.

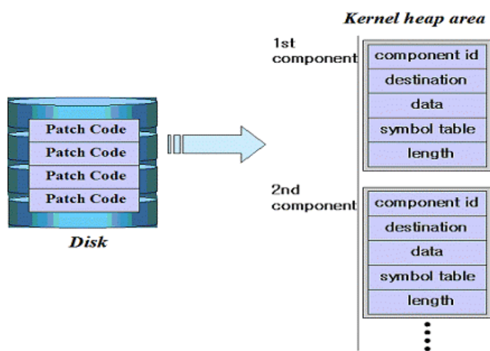

그림 7 메모리 상의 컴포넌트 이미지

5. 결론

앞에서 언급한 바와 같이 기존 커널 재구성 기법에 대한 연구들은 하드웨어 종속적인 문제를 갖고 있다. 이 문제는 커널 재구성 기법에 하드웨어 종속적인 분기 명령어를 사용하기 때문에 발생된다. 따라서 본 논문에서는 트랩을 사용하여 하드웨어 종속적인 문제를 제거하였다.

참고문헌

[1] Ariel Tamches and Barton P.Miller, “Fine-grained dynamic instrumentation of commodity operating system kernels”, In Proceedings of the Third Symposium on Operating Systems Design and Implementation, 1999.

[2] D. Pearce, P. Kelly, U. Harder, and T. Field, “GILK: A dynamic instrumentation tool for the Linux Kernel.”, In Proceedings of the 12th International Conference on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation (TOOLS '02), pages 220--226, London, United Kingdom, 2002.

[3] M Beck, H Böhme, M Dziadzka, U Kunitz, R Magnus and D Verworner: Linux Kernel Internals (second edition). Addison-Wesley, 1997, ISBN: 0-201-33143-8