프로시저 단위의 온라인 프로그램 교체
(Procedure-Based On-Line Program Replacement)

김 영 진 † 김 형 곤 †† 김 화 준 ††† 이 인 환 ††††
(Young-Jin Kim) (Hyung-Gon Kim) (Hwajun Kim) (Inhwan Lee)

요 약 본 논문은 프로시저 단위의 온라인 프로그램 교체를 수행하기 위한 일련의 방법들을 제안하고, 제안한 방법을 SUN Solaris 2.6 환경에서 사용자 응용 소프트웨어에 적용하여 검증한다. 구체적으로 본 논문은 동작 중인 소프트웨어의 어드레스 영역 중 교체 대상 프로시저에 해당하는 영역을 직접 새 버전으로 바꾸기 위한 절차와 방법을 제시할 뿐 아니라, 교체 대상 프로시저의 크기가 변화하는 경우에 발생하는 새 버전 프로시저를 위한 공간 할당 문제와 새 버전의 프로시저를 구 버전의 프로세스 어드레스 스페이스에서 올바르게 동작하도록 외부 심벌 리퍼런스를 수정하는 문제에 대한 일관적인 해결책을 제시함으로써, 프로시저 단위의 온라인 프로그램 교체를 가능하게 한다. 제안한 방법은 기존의 운영체제에서 제공하는 서비스만을 이용하여 온라인 프로그램 교체를 수행한다. 또한 제안한 방법에서는 프로시저라는 작은 교체단위를 지원하며 프로세스 어드레스 스페이스상의 필요한 부분만을 직접 수정하므로 온라인 교체에 따른 성능저하가 적다.

Abstract This paper presents a method for procedure-based on-line program replacement for user applications and illustrates the method in the SUN Solaris environment. In addition to developing procedures for directly changing the address space of a running process, the paper provides solutions on how to allocate space for the new version procedure and how to make the new version procedure work correctly in the old version process address space, when the size of new version procedure is different from that of old version, and thus facilitates procedure-based on-line software replacement. The method performs on-line program replacement using services provided by existing operating systems. Also, the method supports a small, procedure-based replacement and directly modifies necessary parts of a process address space, which results in small performance loss.

1. 서 론

소프트웨어 개발에 있어 사용자가 원하는 모든 기능을 갖춘 결함이 없는 소프트웨어를 한번에 개발하는 것은 현실적으로 불가능하다. 그 주된 이유는 소프트웨어 개발 단계에서 소프트웨어가 갖추어야 할 모든 기능을 예측하는 것이 어렵고 또한 소프트웨어에 내재된 모든 결함을 제거할 수 없기 때문이다. 따라서 소프트웨어 산업체에서는 기존의 소프트웨어의 기능을 개선하거나 결함을 제거하기 위해 새로운 버전의 소프트웨어를 필요에 따라 사용자에게 제공한다. 그러나 사용자가 새로운 버전의 소프트웨어를 설치하는 동안에는 통상 해당 소프트웨어 또는 경우에 따라 전체 시스템의 기능이 정지되어야 한다. 이러한 소프트웨어 업그레이드로 인해 발생하는 동작 중단은 광대역 네트워크를 이용한 글로벌 억세스를 전제로 하는 오늘날의 컴퓨팅 환경과 같이 연속운전이 요구되는 환경에서는 대단히 심각한 문제가 된다. 따라서 업그레이드에 의한 동작 정지 시간을 최소화하기 위해 동작중인 소프트웨어의 일부를 그 동작에 영향을 주지 않고 교체하고자 하는 온라인 소프트웨어 교체(on-line software replacement)의 필요성이 대두

· 이 논문은 1997년 한국학술진흥재단의 공모과제 연구비에 의하여 연구되었음
† 비 회 원 : 한양대학교 전자전기공학부
progkim@hitel.net
†† 비 회 원 : 삼성전자 반도체총괄 메모리 개발사업부 연구원
netii@unitel.co.kr
††† 비 회 원 : 삼성전자 정보가전총괄 Storage 사업부 연구원
hjkim99@kmgw.tel.samsung.co.kr
†††† 정 회 원 : 한양대학교 전자전기공학부 교수
ihlee@email.hanyang.ac.kr
논문접수 : 1999년 3월 27일
심사완료 : 2000년 1월 19일

되고 있다[1].

온라인 소프트웨어 교체를 위해서는 다양한 교체 단위와 교체 대상 소프트웨어의 종류에 대응하기 위한 여러 방법들이 필요하다. 그 이유는 한 가지 방법이 모든 교체 상황에 대해 다 효율적일 수 없기 때문이다. 또한 온라인 소프트웨어 교체의 실용화를 위해서는 각각의 교체 방법에 따른 다양한 교체 지원 도구와 절차의 개발이 필요하다. 더불어 온라인 소프트웨어 교체에 적합한 소프트웨어 제작 방식의 개발이 필요하며, 실제 환경에서 온라인 교체를 통한 소프트웨어 유지 및 보수가 널리 활용될 수 있도록 하기 위한 지속적인 노력이 필요하다. 온라인 소프트웨어 교체가 대단히 중요한 문제임에도 불구하고, 지금까지 소프트웨어 업그레이드에 의한 동작 중단이 당연한 것이라는 인식과 이에 따른 제한적인 연구로 인해 온라인 소프트웨어 교체는 실제 소프트웨어 환경에서 널리 활용되지 못하고 있다.

본 논문은 프로시저 단위의 온라인 프로그램 교체를 수행하기 위한 일련의 방법들을 제안하고, 제안한 방법을 SUN Solaris 2.6 환경에서 사용자 응용 소프트웨어에 적용하여 검증한다. 기존의 연구에서는 간접적인 방식, 즉 다수의 버전을 만들어 놓고 각 버전을 가리키는 포인터를 바꿔주는 방법(indirection)을 사용하여 온라인 소프트웨어 교체를 수행하였으나, 본 논문에서는 동작 중인 소프트웨어의 어드레스 영역 중 교체 대상 프로시저에 해당하는 영역을 직접 새 버전으로 바꾸는 방법을 제시한다. 그리고 이러한 직접 교체 방식에서 교체 대상 프로시저의 크기가 변화하는 경우에 발생하는 새 버전 프로시저를 위한 공간 할당 문제와 새 버전의 프로시저를 구 버전의 프로세스 어드레스 스페이스에서 올바르게 동작하도록 외부 심벌 리퍼런스를 수정하는 문제에 대하여 일반적인 RISC 머신에 적용 가능한 일관적인 해결책을 제시함으로써 프로시저 단위의 온라인 프로그램 교체를 가능하게 한다.

본 논문에서 제시한 방법은 운영체제에서 제공하는 서비스들만을 이용하여 온라인 프로그램 교체를 가능하게 한다. 이에 따라 별도의 소프트웨어 도구나 환경을 필요로 하지 않고, 평상시 사용자 프로그램의 성능에 영향을 미치지 않으며, 기존의 소프트웨어에 간단한 시그널 핸들링만을 추가함으로써 이에 대한 온라인 교체를 가능하게 한다. 또한 제안한 방법은 최소한의 교체 단위인 프로시저 단위의 온라인 교체를 지원하고, 기존의 연구와 달리 프로세스 어드레스 스페이스를 직접 수정하기 때문에 다른 교체 방법에 비해 대상 소프트웨어의 동작 및 성능 저하에 미치는 영향이 적으며, 기존 소프트웨어에 내재된 결함을 제거하는 경우와 같이 프로그램 버전간의 변화가 적은 경우에 매우 효율적으로 교체를 수행할 수 있다. 본 본문에서 제안한 방법은 프로시저의 크기나 외부 인터페이스가 바뀔 때 그리고 새로운 프로시저를 추가할 때에도 사용할 수 있다.

다음절에서는 온라인 소프트웨어 교체에 관한 대표적인 연구를 소개한다. 제 3절에서는 동작 중인 소프트웨어의 코드 부분을 이 소프트웨어의 동작에 영향을 주지 않고 프로시저 단위로 교체하기 위한 절차와 방법, 특히 교체 대상 프로시저의 크기가 변화하는 경우에 적용할 수 있는 온라인 프로시저 교체 방법을 제시한다. 제 4절에서는 제안한 방법을 SUN Solaris 환경에서 구체화하고, 제 5절에서는 제안한 방법의 구현 및 검증을 다룬다.

2. 관련 연구

온라인 소프트웨어 교체에 관한 대표적인 연구들은 다음과 같다. 초기의 연구로서 Fabry[2]와 Goullon[3]은 ADT(Abstract Data Type)의 온라인 교체를 제안하였다. ADT는 명세와 구현이 분리된 프로시저들의 집합으로서 온라인 교체를 위해 명세는 그대로 두고 구현만을 교체하는 방법을 이용하였다. Liskov[4]와 Bloom[5]은 클라이언트/서버 환경에서 서버가 제공하는 서비스의 온라인 교체를 연구하였다. 이러한 서비스는 다수의 관련된 ADT로 구성되는데 프로그램은 서버의 서비스를 이용할 때 이를 직접 호출하지 않고 핸들러(handler)를 통하여 간접적으로 호출한다. 따라서 서비스의 교체는 이러한 핸들러가 새로운 버전의 서비스를 호출하도록 핸들러 내의 포인터를 수정하는 것으로 이루어진다. Magee[6,7]는 분산 프로그램을 구성하는 모듈을 온라인으로 교체하기 위한 방법을 제시하였다. 분산 프로그램의 경우 여러 모듈들은 통신 채널을 통해 연결되어 서로 호출한다. 여기에서 통신 채널을 수정하여 이전 버전의 모듈과의 연결을 해제하고 새로운 버전의 모듈과의 연결을 설정함으로써 교체를 수행한다.

Segal[8]은 유닉스 환경에서 프로시저 단위의 온라인 교체를 지원하기 위한 소프트웨어 설계 환경의 개발을 시도하였다. PODUS(Procedure Oriented Dynamic Updating System)라고 하는 이 교체 환경에서는 프로시저를 호출하기 위해 프로시저의 실제 주소가 저장되어 있는 바인딩 테이블을 참조하게 되며, 이 바인딩 테이블의 엔트리를 수정하여 프로시저를 새로운 버전으로 교체한다. PODUS는 이러한 바인딩 테이블을 이용하기

위해 어드레스 스페이스 자체에 프로그램의 버전과 프로시저의 종류를 나타내는 필드를 포함하는 어드레스 스페이스 모델(large sparse address space model)을 적용하고 이를 세그먼트 단위의 가상 메모리 상에서 구현하였다. PODUS는 유닉스 환경에서 사용자 쉘과 온라인 교체 프로세서, 컴파일러, 링커를 포함하는 통합 소프트웨어 설계 환경이며, 새로운 어드레스 스페이스 모델을 사용하므로 이에 따른 추가적인 소프트웨어 레이어와 전용 컴파일러 및 링커를 필요로 한다. 그리고 PODUS에서 사용한 어드레스 스페이스 모델에는 머신 아이디(machine ID)도 포함되어 있어 분산 환경에 적용이 가능하다.

Gupta[9]는 프로세스 상태 전달(process state transfer)을 사용한 온라인 교체 방법을 제시하였다. 여기서는 새로운 버전의 프로그램에 해당하는 프로세스를 생성한 후, 현재 동작하고 있는 구 버전의 프로세스를 정지시키고, 구 버전 프로세스로부터 새 버전의 프로세스로 프로세스의 상태를 복제하여 새로운 프로세스가 이전의 작업을 계속 수행할 수 있도록 함으로써 교체를 수행한다. 구체적으로 상태 전달에 포함되는 프로세스의 상태에는 사용자 프로세스 어드레스 스페이스(코드, 데이타 및 스택), 머신 정보(일반/특수용도 레지스터), 그리고 커널 정보(시스템 콜과 프로세스의 의해 열린 파일에 관련된 정보)가 있다. 제안한 방법을 이용한 온라인 교체는 상태 정보의 재구성을 위한 많은 양의 일을 필요로 하고, 교체 대상 프로세스의 외부 환경에 대한 컨텍스트의 유지, 프로세스에 대한 커널 정보의 수정 및 CPU 레지스터 상태 정보의 처리 등의 복잡한 과정을 필요로 한다. 따라서 교체를 위한 동작 정지 시간이 상당히 길어지고, 이러한 교체 정지 시간은 교체 대상 프로그램의 크기가 증가함에 따라 길어지게 된다. 이 외에도 Gupta는 프로세스가 동적으로 생성한 데이타 구조에 대해서는 명확한 상태 전달 방법을 제시하지 않았고, 프로세스 상태 전달 방법을 사용하기 위해서는 교체 제어 프로세스와 교체 대상 프로세스는 부모-자식(parent-child)관계이어야 한다.

본 논문에서 제안하는 온라인 프로그램 교체 방법은 앞서 언급되었던 Segal의 연구와는 달리 온라인 교체를 지원하기 위한 별도의 소프트웨어 레이어나 전용화된 컴파일러와 링커를 사용하지 않고, 운영체제에서 기본적으로 제공하는 서비스들을 이용하여 온라인 교체를 수행한다. 따라서 제안한 방법은 평상시 사용자 프로그램의 성능에 영향을 미치지 않으며, 기존의 소프트웨어에 간단한 시그널 핸들링만을 추가함으로써 이에 대한 온라인 교체를 가능하게 한다. 또한, 본 논문에서 제시한 방법은 Gupta의 연구와 달리 최소한의 교체 단위인 프로시저 단위의 온라인 교체를 지원하고 또한 프로세스 어드레스 스페이스에서 교체 대상 프로시저 영역만을 직접 수정하므로, 교체 과정이 간단하고 교체를 위한 일과 교체가 대상 소프트웨어의 동작과 성능 저하에 미치는 영향이 적다.

3. 온라인 프로시저 교체의 방법과 절차

일반적으로 프로그램은 시스템의 어드레스 스페이스 상에서 프로세스의 형태로 실행된다. 그리고 하나의 프로세스는 크게 실행 파일로부터 읽혀진 코드 영역과 코드의 실행에 필요한 데이타 영역으로 이루어진다 (그림 1). 본 논문에서는 소프트웨어 업그레이드에 의한 동작 중단을 최소화하기 위한 온라인 소프트웨어 교체의 한 방법으로서 프로시저 단위의 교체 방법을 제시한다. 온라인 프로시저 교체란 이미 실행되고 있는 프로세스의 코드 영역을 프로시저 단위로 새로운 버전으로 교체하는 것이다. 즉, 제안한 방법에서는 그림 1에 나타낸 것과 같이 구 버전의 실행 파일로부터 하나의 프로세스가 생성되어 실행되고 있을 때, 새 버전의 실행 파일로부터 업데이트된 프로시저를 추출하고 이를 이용하여 현재 동작중인 구 버전의 프로세스 코드 영역의 해당 프로시저를 교체함으로써, 이 소프트웨어의 동작을 정지시키지 않으면서 소프트웨어의 업그레이드를 수행한다.

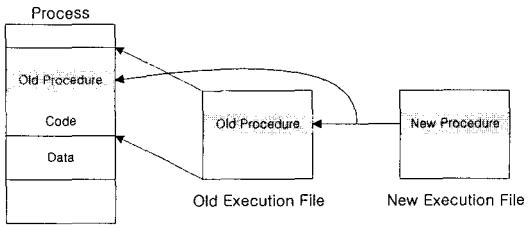

그림 1 온라인 프로시저 교체의 개념

일반적으로 온라인 소프트웨어 교체에서는 교체를 수행하는 과정이나 교체가 끝난 후 교체 대상 소프트웨어의 오동작이 발생하지 않아야 한다. 또한 온라인 교체로 인한 대상 소프트웨어의 성능 저하가 최소화되어야 한다. 이러한 요구 조건을 만족시키기 위하여 본 연구에서 제시한 온라인 프로시저 교체는 크게 교체 시기 선정, 교체 중 오동작 방지, 그리고 프로시저 교체의 세 과정으로 구성된다 (그림 2). 본 절에서는 이러한 세 가지 과정을 수행하는데 필요한 방법과 절차를 제시하고, 특

히 제안한 프로시저 단위의 온라인 교체에서 가장 핵심적인 문제 중의 하나인 교체 대상 프로시저의 크기가 변할 때 어떻게 새 버전 프로시저가 구 버전 프로세스의 어드레스 스페이스에서 올바르게 동작하도록 할 수 있는가하는 문제에 대한 해결책을 제시한다. 본 절에서 제안한 방법은 SUN Solaris 환경에서 구체화되며, 이는 4절에서 다룬다.

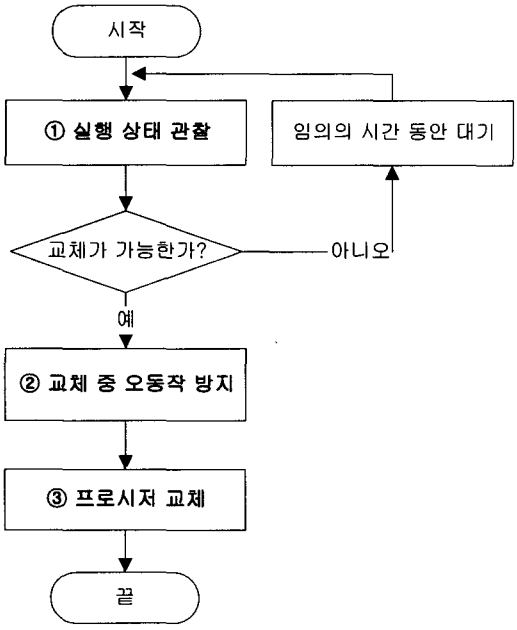

그림 2 교체 절차

3.1 상태 관찰을 통한 교체 시기 선정

온라인 프로그램 교체에서 교체 대상 프로시저가 실행 중일 때 교체를 수행하게 되면 소프트웨어의 정상적인 동작을 보장할 수 없으므로, 교체 대상 프로시저의 실행 상태를 관찰하여 이 프로시저가 실행되지 않고 있을 때 교체를 시작하여야 한다.

프로시저의 실행 상태를 확인하기 위하여 간단하게 PC(Program Counter)의 값을 추적하는 방법을 생각할 수 있다. 즉, PC는 프로세서에 의해 실행되는 명령어의 어드레스 값을 가지고 있으므로, 이 값이 교체 대상 프로시저 어드레스 범위에 포함되는지의 여부를 조사하는 것이다. 그러나 PC는 현재 프로세서가 수행하는 명령어만을 가리키므로, 교체 대상 프로시저가 다른 프로시저를 호출한 경우, 교체 대상 프로시저는 실행 상태에 있지만 PC의 값은 대상 프로시저가 호출한 다른 프로시저의 어드레스 범위에 있게 되어, 교체 대상 프로시저의 실행 여부를 정확히 확인할 수 없다.

따라서 본 논문에서는 프로세스의 런타임 스택(run-time stack)에 저장되어 있는 프로시저 호출 이력(stack trace 또는 procedure activation record)을 관찰하여 교체 대상 프로시저의 실행 상태를 판단한다. 일반적으로 런타임 스택에는 모든 실행 중인 프로시저에 대하여 프로시저 이름, 종료 후 되돌아갈 PC 값, 매개변수(parameters)와 지역 변수의 값 등의 정보가 호출된 순서대로 저장된다[10]. 그림 3은 SUN Solaris에서 실행중인 프로세스의 런타임 스택을 보인다. 이 그림에서 보듯이 스택 트레이스 정보를 이용하면 교체 대상 프로시저가 실행 중인지를 정확히 판단할 수 있다.

온라인 프로시저 교체를 위한 교체 시기 선정에서 교체 대상 프로시저가 실행 중인 것으로 판단된 경우의 대처 방안은 두 가지가 있다. 하나는 임의의 시간 동안 대기한 다음 다시 런타임 스택을 확인하는 것이고, 다른 하나는 스택의 크기가 작아질 때를 감지하여 이 때 다시 런타임 스택을 확인하는 것이다. 후자의 방법이 시간적인 면에서 우수하지만, 이를 위한 추가적인 작업이 필요하고 또한 교체시기의 선정이 일반적으로 분초를 다투는 문제가 아니므로 본 논문에서는 전자의 방법을 사용한다. 그리고 본 논문에서는 "main" 프로시저와 같이 항상 스택에 존재하는 프로시저의 온라인 교체는 고려하지 않는다. 실제로 대규모 소프트웨어에서 이러한 프로시저들의 수효는 대단히 적다.

```
ef7385e8 read      (0, ef7a9668, 400)
ef7385e8 _libc_read (0, ef7a9668, 400, ffffffff, ef7a227c, ef7638ac) + 8
ef7638d4 _filbuf   (ef7a5f18, ef7a9a68, 60a, 33bdd8f, 5b6991e, 0) + b0
ef7678ec getc_unlocked (ef7a5f18, 25, ef7ed34c, ef7a227c, effffcb8, ef7ed34c) + 2c
ef7644cc __doscan_u (64, 1219a, effffcb8, 1, 1, ef7a61e9) + 534
ef763170 _doscan   (ef7a5f18, 12198, effffcb8, ef7a9a7c, 0, 0) + 50
ef768640 scanf     (12198, effffcdc, 12000, 0, 0, 0) + 30
00011554 print_result (0, ef7a7f74, ef7a46c8, 2, ef7a227c, ef718cb0) + 20
00011614 main      (1, effffdc4, effffdcc, 22714, 0, 0) + 1c
00010cc4 _start    (0, 0, 0, 0, 0, 0) + 5c
```

그림 3 Solaris에서의 런타임 스택 정보

3.2 교체 중 실행 방지

교체 대상 프로시저가 실행 중이 아닐 때 교체를 시작하더라도, 교체가 수행되는 도중에 교체 대상 프로시저가 새로 호출되어 실행되면, 이 프로시저는 그 기능상의 일관성이 유지되지 않은 상태에서 실행이 되고 이에 따라 오동작이 발생할 수 있다. 이를 해결하기 위한 가장 간단한 방법으로는 전체 프로세스의 실행을 잠시 정지시키고 프로시저의 교체를 수행하는 것을 생각할 수

있다. 그러나 이러한 경우 교체 과정 동안 이 소프트웨어의 기능이 정지하게 되고 이에 따른 성능저하가 문제가 될 수 있다. 본 논문에서는 이러한 성능저하를 줄이기 위하여 프로세스의 어드레스 스페이스 상에 교체 대상 프로시저에 해당하는 영역을 감시 영역으로 설정하고, 이 영역 내의 코드가 참조될 때 트랩에 의해 프로세스가 정지되도록 함으로써, 교체 대상 프로시저가 교체 과정 중에 새로 호출되는 경우에만 해당 프로세스를 정지시키는 방법을 사용한다 (그림 4). 커다란 소프트웨어 시스템에서 특정 프로시저가 온라인 교체를 수행하는 비교적 짧은 시간 동안 호출될 확률은 상당히 낮으므로 이러한 방법을 사용하면 온라인 프로시저 교체에 따른 소프트웨어의 성능저하를 크게 줄일 수 있다.

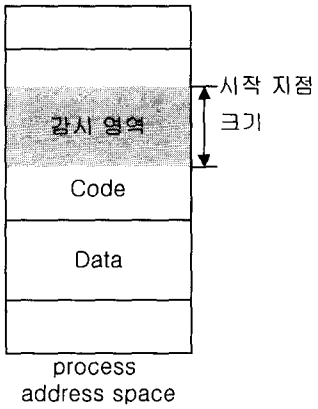

그림 4 감시 영역의 설정

3.3 프로시저 교체

적절한 교체 시기가 선정되고 교체 과정에서의 오동작을 방지할 조치가 완료되면 실질적인 프로시저의 교체가 이루어진다. 본 절에서는 우선 교체 대상 프로시저의 크기와 외부 인터페이스가 변하지 않는 경우의 온라인 프로시저 교체 방법을 다룬다. 이러한 제한 조건 없이 온라인 프로그램 교체를 수행하기 위한 방법은 3.4절에서 제시한다.

하나의 프로세스는 디스크와 메인 메모리, 캐시 메모리의 내용으로 이루어지며, 실행되는 과정에서 캐시 메모리의 내용은 캐시의 하위 저장 장치인 메인 메모리에서, 메인 메모리의 내용은 다시 디스크에서 불러온다. 따라서 온라인 프로시저 교체에서는 교체 대상 프로시저에 해당하는 캐시, 메인 메모리, 디스크의 내용을 모두 새로운 버전의 프로시저로 교체하여야 한다. 그리고 실제 교체는 디스크 수정, 메인 메모리 수정, 그리고 캐시 메모리 수정의 순서로 이루어진다. 그 이유는 상위 저장 장치의 내용을 먼저 수정할 경우 온라인 교체 시스템이 모르는 가운데 수정된 상위 저장 장치의 내용이 아직 수정되지 않은 하위 저장 장치의 내용으로 덮어 쓰여질 수 있기 때문이다.

디스크 수정은, 교체 대상 프로시저의 구 버전 및 신 버전이 같은 크기와 동일한 외부 인터페이스를 갖는 경우에는, 교체 대상 프로시저의 크기와 실행 파일 상에서의 시작 위치를 알아내어 새 버전의 프로시저를 현재 실행 파일의 해당 프로시저의 영역에 직접 쓰는 것으로 완료된다. 메인 메모리의 수정은 두 가지 가능성을 생각할 수 있다. 하나는 디스크 수정과 마찬가지로 프로세스 어드레스 스페이스를 직접 수정하는 방법이고, 다른 하나는 수정될 부분에 해당하는 메인 메모리 상의 페이지를 인위적으로 무효화시키는 방법이다. 무효화된 페이지는 다음에 참조될 때 디스크로부터 다시 읽혀지게 되므로, 디스크 수정이 이루어진 후 페이지를 무효화시키면 메인 메모리의 내용이 새 버전으로 교체되는 효과를 갖는다[11].

메인 메모리 수정에서 어드레스 스페이스를 직접 수정하는 경우에는 교체할 프로시저가 커질수록 수정을 위한 시간이 많이 소요된다. 그리고 교체할 프로시저가 물리적인 메모리 상에 없을 경우 이 프로시저를 디스크에서 새로 읽어 와야 한다. 페이지를 무효화시키는 방법은 상대적으로 교체할 프로시저의 크기가 클 때에는 유리하나, 프로시저의 크기가 페이지 크기에 비해 상당히 작을 경우에는 대상 프로시저 이외의 다른 부분까지 무효화되는 효과가 있다. 따라서 효율적인 온라인 교체를 위해서는 위의 두 방법을 다 지원하고 교체 시스템이 필요에 따라 두 방법 중 하나를 선택하여 적용하는 것이 바람직하다.

캐시 메모리의 수정은 캐시의 유형과 메인 메모리 수정을 위해 어떤 방법을 사용했는지에 따라 그 방법이 달라진다. 메인 메모리가 수정되면 그 내용이 바로 캐시 메모리에도 반영되는 경우에는 따로 캐시 메모리를 수정하지 않아도 되나, 그렇지 않은 경우에는 별도의 캐시 메모리의 수정이 필요하다. 이렇게 캐시 메모리의 내용을 수정하고 앞서 설정한 감시영역을 해제하면 프로시저 교체가 완료된다.

3.4 교체 조건의 확장

제안한 온라인 프로시저 교체 방법의 개발에 있어 가장 핵심적인 문제 중의 하나는 교체 대상 프로시저의 크기가 변할 때 어떻게 새 버전 프로시저가 구 버전 프로세스의 어드레스 스페이스에서 올바르게 동작하도록

할 수 있는가하는 것이다. 본 절에서는 프로시저의 크기와 외부 인터페이스가 바뀌거나 프로시저가 추가 또는 삭제되는 등의 여러 가지 조건에서도 적용 가능한 온라인 프로시저 교체 방법을 다룬다.

3.4.1 프로시저 크기의 변화

교체하려고 하는 프로시저의 크기가 증가하는 경우에는, 교체에 앞서 우선 새 버전의 프로시저를 위한 공간을 확보하여야 한다. 이를 위하여 본 논문에서는 프로세스 어드레스 스페이스에서 프로세스에 의해 사용되지 않는 공간 (비매핑 영역; unmapped area)을 활용한다. 구체적으로 프로세스 어드레스 스페이스에서 프로그램 코드, 데이타, 스택 등을 위해 미리 할당된 영역 이외의 사용되지 않는 공간에 교체 대상 프로시저의 새 버전을 매핑한다. 그리고 구 버전의 프로시저의 첫 부분을 수정하여 이 프로시저가 호출될 경우 새 버전의 프로시저로 다시 호출이 일어나도록 한다(그림 5). 이렇게 하면 구 버전 프로시저가 호출되었을 경우 새 버전 프로시저로 이동한 후에 실행이 일어나므로 결과적으로 프로시저 교체가 이루어진다.

이 경우 중요한 것은, 교체 대상 프로시저의 크기가 증가함에 따라, 새 버전의 실행 파일 내의 심벌들의 위치가 구 버전의 그것들과는 다르게 된다는 것이다. 즉, 새 버전의 교체 대상 프로시저가 이 프로시저 외부의 심벌들을 액세스할 때 이 심벌들의 위치는 구 버전의 그것들과는 다르게 되며, 따라서 새 버전의 프로시저가 구 버전을 이용하여 설정된 프로세스 어드레스 스페이스 내에서 정상적으로 동작하게 하기 위해서는 새 버전 프로시저 내의 이러한 외부 심벌 리퍼런스를 구 버전에 맞게 수정해 주어야 한다.

외부 심벌 리퍼런스란 한 프로시저에서 이 프로시저에 해당하는 어드레스 스페이스의 외부에 위치한 코드나 데이타를 액세스하는 것으로, 외부 심벌 리퍼런스의 대상은 크게 코드 영역상의 위치(예: 프로시저의 시작 위치)와 데이타 영역상의 위치로 분류할 수 있다. 여기서 데이타 영역과 관련된 대상은 전역 변수나 지역 정적 변수와 같은 정적 변수의 위치, 스택 영역, 그리고 힙 영역의 세 가지로 다시 분류된다. 이 중에서 스택 영역은 런 타임 스택의 역할을 하는 것으로 프로시저의 호출 시 리턴 어드레스나 지역 변수 등을 저장하는데, 한 프로시저가 실행 중이 아니면 이 프로시저에 해당하는 영역이 없으므로 외부 심벌 리퍼런스 수정의 대상이 되지 않는다. 또한 힙 영역에 동적으로 생성되는 데이타의 경우에는 데이타의 위치를 가리키는 포인터를 사용하여 액세스하게 되는데, 교체 대상 프로시저가 실행 중이 아닐 때 교체를 수행하므로 이 프로시저에 의해서는 데이타가 새로 생성되지 않으며, 다른 프로시저에 의해 이미 생성된 데이타의 경우에는 그 포인터를 외부에서 받거나 전역 포인터 변수에서 읽어서 사용하게 되므로 외부 심벌 리퍼런스 수정의 대상이 되지 않는다. 따라서 수정이 필요한 외부 심벌 리퍼런스의 대상은 코드 영역상의 위치와 정적 변수에 해당하는 위치의 두 가지로 종합된다.

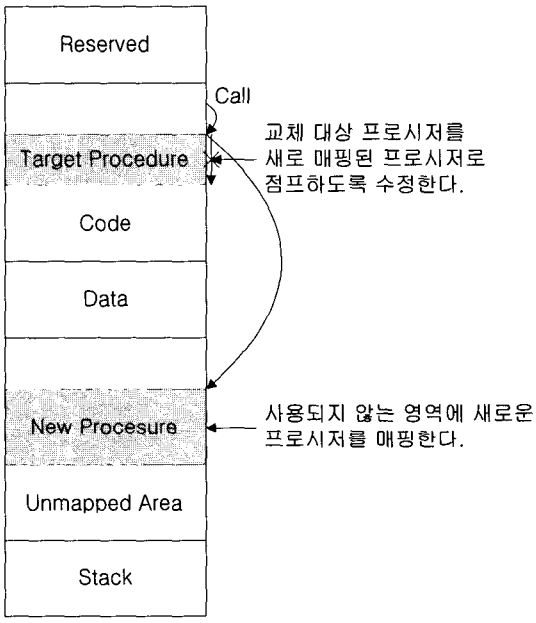

그림 5 새 버전 프로시저의 매핑

이러한 외부 심벌 리퍼런스 대상에 대한 구체적인 수정 방법을 도출하기 위해서는 프로그램에서 외부 심벌을 리퍼런스하는 유형을 고려하여야 한다. 일반적으로 프로그램에서는 두 가지의 형태로 외부 심벌을 리퍼런스한다. 첫째는 일반적인 변수의 사용이나 함수의 호출과 같이 심벌의 이름으로 액세스하는 것이고, 둘째는 포인터 상수를 사용하는 것이다. 포인터 상수는 심벌의 주소값을 나타내는 상수로 C 언어에서 변수명 앞에 &기호가 붙은 경우 또는 함수나 배열의 이름만 사용하는 경우가 포인터 상수를 이용한 외부 심벌 리퍼런스에 해당한다. 제안한 온라인 프로시저 교체를 가능하게 하기 위해서는 이러한 두 가지 리퍼런스 유형에 대한 일관성 있는 수정 방법이 필요하다. 이러한 외부 심벌 리퍼런스의 유형에 따른 구체적인 수정 방법은 CPU 명령어 셋 구조에 관계되므로 이는 4장에서 다룬다.

교체 대상 프로시저의 크기가 변하는 경우의 교체 과정을 단순화하기 위하여 처음 프로그램을 제작할 때 미리 여유 공간을 확보해 두는 방법을 생각할 수 있다. 예를 들어, 미리 프로시저마다 일정한 여유 공간을 확보해 두면 이후에 프로시저의 크기가 커질 경우에도 구 버전 프로시저 영역을 그대로 사용할 수 있으므로 3.3절에서 제시한 방법을 이용하여 교체를 수행할 수 있다. 또 다른 방법은 각 프로시저마다 여유공간을 확보하는 대신, 몇 개의 충분한 크기를 갖는 더미 프로시저를 미리 소스코드에 추가하는 것이다. 이렇게 소스 코드에 미리 여유 공간을 확보하면 온라인 프로시저 교체는 그만큼 수월해지나 코드 부분의 크기는 증가하게 된다.

새 버전의 프로시저가 구 버전보다 크기가 작아지는 경우에는 구 버전 프로시저가 존재하던 공간을 그대로 사용할 수 있으므로 3.3절에서 제시한 교체 방법을 적용할 수 있다. (물론 이러한 경우에도 본 절에서 제시한 방법을 사용해도 무방하다.) 그러나 구 버전과 새 버전 프로시저의 크기가 다른 만큼 코드에서 심벌들의 위치에 차이가 있게 되므로 프로시저의 크기가 증가하는 경우에서와 같이 새 버전 프로시저 내의 외부 심벌을 액세스하는 부분의 수정은 여전히 필요하다. 교체 대상 프로시저의 크기가 변하지 않는 경우에는 외부 심벌의 수정이 필요하지 않다.

3.4.2 프로시저 그룹 교체

위의 3.4.1절에서 제시한 바와 같이 사용되지 않는 텍스트 공간을 사용하면 프로시저의 교체 뿐 아니라 새로운 프로시저의 추가도 가능하다. 그러나 새로운 프로시저를 추가할 경우나 기존의 프로시저를 삭제할 경우, 또는 기존의 프로시저가 새로운 버전으로 교체될 때 프로시저의 외부 인터페이스가 바뀌는 경우에는 하나의 프로시저만을 수정해서는 교체가 이루어지지 않으며 교체와 관련된 여러 프로시저들을 모두 수정해야 한다.

이러한 프로시저 그룹 교체는 앞서 제시한 교체 방법을 반복해서 적용함으로써 가능하다. 즉, 프로시저 P1과 P2가 동시에 교체되어야 하는 경우에는 프로시저 P1과 P2가 모두 실행되지 않을 때로 교체 시기를 선정한다. 그리고 교체 중의 오동작을 방지하기 위하여 이들 두 프로시저의 영역을 모두 감시영역으로 설정하여 교체가 수행되는 동안 이들 프로시저가 새로 호출되어 실행되는 것을 방지한다. 그 다음에 프로시저 P1과 P2의 교체를 각각 수행한다. 이러한 프로시저 그룹의 교체를 통해 교체 대상 프로시저의 외부 인터페이스가 바뀌는 경우의 온라인 교체와 프로시저의 온라인 추가 또는 삭제가 가능하다.

4. Solaris에서의 온라인 프로시저 교체

본 장에서는 3장에서 제시한 온라인 프로시저 교체 방법을 SPARC 프로세서 상에서 동작하는 SUN Solaris 2.6 환경에 적용한다. 이를 위해 우선 Solaris 환경에서의 프로세스 어드레스 스페이스의 구조를 살펴보고 이를 근거로 온라인 프로시저 교체를 위한 구체적인 절차와 방법을 제시한다.

4.1 프로세스 어드레스 스페이스

Solaris 환경에서 프로세스의 어드레스 스페이스는 텍스트, 데이타, BSS, 스택, 비매핑 영역으로 나뉘어지며, BSS와 스택영역 사이에는 프로세스에서 사용하는 라이브러리들이 존재하게 된다[12,13]. 텍스트 영역은 프로세스에서 실제로 실행되는 코드를 담고 있으며, 데이타 영역과 BSS영역에는 전역 변수 중 초기화된 것과 초기화되지 않은 것이 각각 들어가게 된다. 스택영역은 런타임 스택을 나타내는 것으로 프로시저 호출이 일어날 때 주고받는 매개 변수와 해당 프로시저의 지역 변수, 되돌아올 주소 값 등이 호출된 순서대로 저장된다.

각각의 영역은 디스크에 논리적으로 매핑이 되어 있는데, 텍스트 영역이나 라이브러리들은 실행 파일 또는 라이브러리 파일과 같은 이름이 붙여진 파일(named file)들에 매핑되어 있고, 그 외의 나머지 영역들은 스왑 장치에 매핑되어 있다[14]. 그림 6은 프로세스 어드레스 스페이스의 매핑 정보를 출력하는 사용자 명령어인 pmap(1)을 이용하여 실제 어드레스 스페이스의 모습을 살펴 본 것이다.

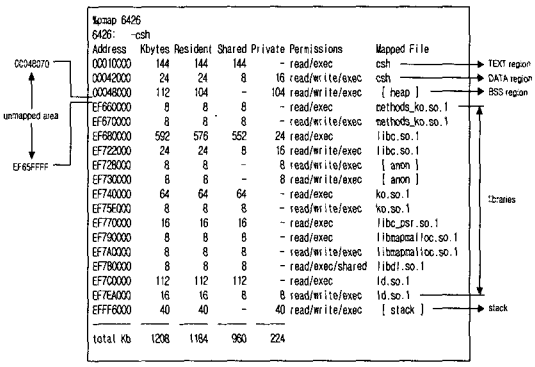

그림 6 프로세스의 매핑 정보

교체 대상이 되는 프로그램의 실제 명령문들은 텍스트 영역에 들어있으며, 텍스트 영역과 데이타 영역은 연속된 공간에 존재하므로 텍스트 영역에는 여유 공간이

존재하지 않는다. 그러나 BSS영역과 스택 사이에는 그림과 같이 비매핑 영역이 존재한다. 이 영역은 프로세스에 의해 사용되지 않는 영역으로, 온라인 프로시저 교체에서 새 버전의 프로시저의 크기가 증가하는 경우 이 프로시저를 새로 매핑하기 위한 공간으로 사용할 수 있다.

4.2 교체 절차

온라인 프로시저 교체에서는 교체 대상 프로시저의 크기가 변하는 경우를 고려하여야 한다. 우선 본 절에서는 프로시저의 크기가 증가하는 경우의 온라인 교체 방법을 그림 7에 표시한 절차에 따라 제시한다. 먼저 새 버전의 프로시저를 독립된 파일로 추출하고, 외부 심벌 리퍼런스 수정에 앞서, 추출된 새 버전 프로시저 파일의 매핑을 수행한다. 이는 외부 심벌 리퍼런스 수정을 위해서는 새 버전 프로시저의 구 버전 프로세스 어드레스 스페이스 상의 위치를 알아야 하기 때문이다. 다음으로 추출한 새 버전 프로시저 파일 상에서 외부 심벌 리퍼런스를 수정하고, 수정된 파일을 이용하여 실제 디스크, 메모리 상의 새 버전 프로시저 영역을 수정한다. 그리고 마지막으로 구 버전 프로시저를 수정하는 것으로 교체는 완료된다. 본 절의 마지막에서는 프로시저의 크기가 감소하거나 변하지 않는 경우에는 교체 절차가 어떻게 단순화될 수 있는지를 다룬다.

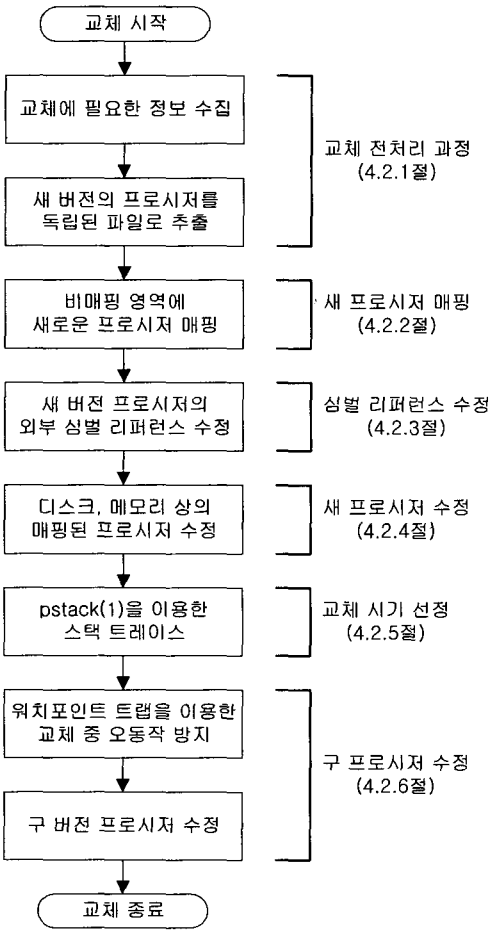

그림 7 교체 절차

4.2.1 교체 전처리 과정

온라인 프로시저 교체를 위한 전처리 과정에서는 구 버전과 새 버전의 교체 대상 프로시저의 프로세스 어드레스 스페이스에서의 위치와 크기 및 실행 파일에서의 위치 정보를 수집하고, 교체 대상 프로세스의 어드레스 스페이스에 새로 매핑할 새 버전의 프로시저를 파일의 형태로 준비한다.

```
                                  :
[70]  0x10af8  116   FUNC  GLOB  0  8  _start
[74]  0x11cd8  72    FUNC  GLOB  0  8  _register_frame_info
[78]  0x10da4  1064  FUNC  GLOB  0  8  getloc
[83]  0x11418  60    FUNC  GLOB  0  8  rsproc
[85]  0x11f70  28    FUNC  GLOB  0  9  _init
                                  :
```

그림 8 심벌 테이블

이를 위해 우선 구 버전과 새 버전의 실행 파일로부터 교체 대상 프로시저의 어드레스 스페이스에서의 위치(가상번지)와 크기를 추출한다. 이러한 정보는 실행 파일의 심벌 테이블에서 추출할 수 있다. 그림 8은 심벌 테이블의 예를 보여준다. 그림에서 각 줄의 마지막 필드는 실행 파일 내부의 심벌들을 나타내고, 두 번째 필드는 해당 심벌의 프로세스 어드레스 스페이스에서의 시작 위치를 나타낸다. 또한 세 번째 필드는 어드레스 스페이스에서 각 심벌이 차지하는 크기를 나타내며, 네 번째 필드는 이 심벌의 종류를 표시한다. 그림에서 질게 표시된 "getloc"이라는 심벌을 살펴보면 네 번째 필드에 "FUNC"라고 표시되어 있으므로 이것이 프로시저를 나타냄을 알 수 있다. 이 프로시저는 프로세스 어드레스 스페이스에서 0x10da4의 위치에 존재하며 1064바이트의 크기를 갖는다. 이러한 정보와 프로세스 어드레스 스페이스에서의 명령어 시작 번지로부터 실행 파일 상에서의 교체 대상 프로시저의 시작 위치를 알 수 있다. 예를

들어, 프로세스 어드레스 스페이스에서의 명령어 시작 번지가 0x10000이라면 "getloc" 프로시저의 실행 파일에서의 시작 번지는 0x0da4 (=0x10da4-0x10000)가 된다.

4.2.2 새 버전 프로시저 매핑

위와 같이 교체 대상 프로시저의 새 버전을 추출하고 이를 파일의 형태로 저장한 다음에는 새 버전 프로시저를 구 버전 프로세스 어드레스 스페이스의 비매핑 영역으로 매핑한다. 새 버전 프로시저의 매핑은 mmap(2) 시스템 콜을 사용하여 수행한다[14]. 그런데 이 시스템 콜은 이것을 호출한 프로세스의 어드레스 스페이스에 대해서만 그 기능을 수행할 수 있기 때문에, 이 시스템 콜을 호출하여 매핑을 수행하는 프로시저가 교체 대상 프로그램의 내부에 포함되어 있어야 한다. 이러한 기능은 시그널(signal)을 이용하여 실현한다 (그림 9).

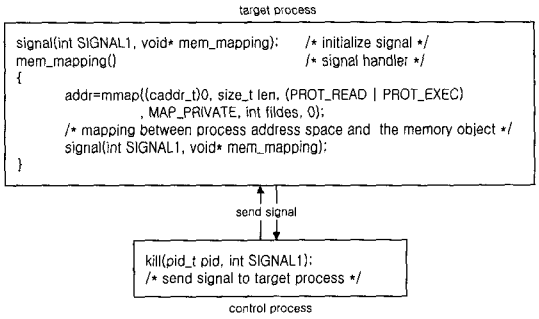

그림 9 시그널을 이용한 프로시저 매핑

우선 signal(3C) 기능을 이용하여 교체 대상 프로그램 내에 시그널을 설정하고, 이 시그널을 받았을 때 실제의 매핑을 수행할 시그널 핸들러(signal handler)를 정의한다. 그리고 교체 시스템의 일부인 외부 제어 프로세스는 매핑이 필요할 경우 교체 대상 프로세스에 시스널을 보낸다. 새 버전 프로시저의 매핑이 끝나면 시그널 핸들러는 다시 시그널을 이용하여 제어 프로세스에 이 사실과 새 버전 프로시저가 매핑된 위치(그림 9 에서 "addr" 값)를 알린다. 이렇게 시그널 핸들러를 이용하여 교체를 수행하면, 이 시그널 핸들러가 실행되는 동안에는 이 프로세스는 다른 기능을 제공할 수 없으며, 이에 따른 성능 저하가 일어날 수 있다. 따라서 시그널 핸들러의 기능을 최소화하는 것이 중요하다. 이러한 시그널 핸들러는 교체 대상 프로세스의 내부에서 수행되고 따라서 기존 프로그램에 이 기능을 추가하여야 한다. 이에 필요한 시그널의 초기화와 실제 매핑을 수행하는 과정은 라이브러리의 형태로 만들어 재사용 가능하도록 하였다.

4.2.3 외부 심벌 리퍼런스 수정

새 버전 프로시저의 매핑이 완료되어 새 버전 프로시저의 위치가 결정되면, 새 버전의 프로시저가 구 버전의 프로세스 어드레스 스페이스에서 올바르게 동작하도록 외부 심벌을 액세스하는 부분을 수정한다. 이 부분은 제안한 온라인 프로시저 교체에 있어 가장 핵심적인 부분 중의 하나로서, 3.4.1절에서 설명한 바와 같이, 이러한 수정은 외부 심벌을 이름으로 액세스하는 경우와 외부 심벌의 포인터 상수가 사용되는 경우를 모두 고려하여야 한다. 본 절에서는 SPARC 머신에서 이러한 두 가지 경우를 고려한 외부 심벌 리퍼런스 수정 방법을 제시한다. 이러한 수정을 위한 구체적인 사항은 머신 종류에 따라 다르나 그 기본 개념은 유사한 명령어 구조를 갖는 다른 RISC 머신에도 적용 가능할 것이다.

먼저 외부 심벌을 이름으로 액세스하는 경우에 외부 심벌 리퍼런스를 수정하기 위해서는 어떤 명령어들이 수정의 대상이 되는지 알아야 한다. 일반적으로 외부 심벌 리퍼런스에 사용되는 명령어에는 데이타 영역과 관련되어 메모리 액세스를 수행하는 명령어와 코드 영역과 관련되어 프로그램의 흐름을 현재의 프로시저 외부로 변경하는 명령어가 있다. SPARC 머신에서는 데이타 엑세스와 관련되는 로드/스토어 명령어에서 그 대상이 로컬 스태틱 데이타이거나 전역 데이타인 경우, 프로시저 호출을 수행하는 명령어, 그리고 간접 레지스터 주소 지정 방식을 사용하여 점프를 수행하는 명령어인 경우가 수정의 대상이 된다[15]. 표 1은 SPARC 머신의 명령어를 간단히 분류하여 나타낸 것으로 이 중 짙게 표시되어 있는 부분이 외부 심벌을 액세스하는, 즉 수정이 필요한 명령어들이다. 표 1에서 보듯이, SPARC 머신에서는 레지스터, 간접 레지스터, 즉치, PC 상대의 네 가지 주소 지정 방식이 있고, 이 중 수정 대상이 되는 명령어들이 사용하는 것은 간접 레지스터와 PC 상대의 두 가지 주소 지정 방식이며, 각각의 방식에 따른 구체적인 수정 방법은 다음과 같다.

먼저 레지스터 간접 주소 지정 방식인 경우에는 수정 대상 명령어가 주소 지정에 사용할 레지스터의 값을 수정해야 한다. SPARC 머신에서 수정 대상이 되는 명령어 중에서 레지스터 간접 주소 지정 방식을 사용하는 명령어들의 형식은 모두 같으므로 스토어 명령어를 이용하여 수정 방법을 설명한다. 예를 들어, "a"라는 전역 변수에 상수값 100000을 넣는 코드(a=100000)는 다음과 같이 컴파일 된다(전역 변수 "a"의 어드레스는 0x21d40번지):

표 1 SPARC 머신의 명령어

Type of instruction	instruction	addressing mode	Comment	
arithmetic inst.	all	register/ immediate	or를 register indirect addressing mode를 위해 사용하는 경우 (synthetic instruction 참조)	
logical inst.	all	register/ immediate	-	
shift inst.	all	register/ immediate	-	
load inst.	all	register indirect	local static data global variable	
store inst.	all	register indirect	local static data global variable	
integer branch inst.	all	PC relative	-	
machine trap inst.	all	register/ immediate	-	
machine control inst.	call	PC relative	모든 경우	
	jmpl	register indirect	rd=%g0	subroutine return
				-
			rd=%o7	subroutine call
				모든 경우
			others	unconditional jump
				모든 경우
	rett	register indirect	-	
	sethi	immediate	register indirect addressing mode를 위해 사용될 경우 (synthetic instruction 참조)	
	save	register indirect	-	
	restore	register indirect	-	
floating-point inst.	load/store	register indirect	local static data global variable	
	others	register	-	
floating-point branch inst.	all	PC relative	-	

r1 ←0x21d40 - ①
r2 ←100000 - ②
store r2,(r1) - ③
(memory[r1] ←r2)

여기에서 ①부분이 ③의 스토어 명령어에서 주소 지정에 사용하는 레지스터 "r1"의 값을 정하고 있는데, 이 값이 외부 심벌에 해당하는 경우 현재 "r1"의 값은 새 버전 프로세스 어드레스 스페이스 상의 심벌 위치를 나타내므로, 구 버전의 프로세스 어드레스 스페이스와 맞지 않는다. 따라서 새 버전 및 구 버전 실행 파일의 심벌 테이블을 이용하여 이 심벌 이름에 대한 구 버전 어드레스 스페이스상의 위치를 추출하고, 이 정보를 이용하여 "r1"의 값이 구 버전 프로세스 어드레스 상의 심벌 위치가 되도록 ①에 해당하는 부분을 수정한다.

SPARC 어셈블러에서는 위의 예에서 보인 ①과 ②처럼 레지스터에 숫자 상수를 넣는 경우, 그 값이 13비트 이상이면 항상 "sethi" 명령어를 사용하여 상위 22비트 값을 결정한다. 그리고 SPARC 머신에서 동작하는 Solaris에서 프로세스 어드레스 스페이스는 항상 0x10000번지에서부터 시작한다. 따라서 주소 지정에 사용되는 레지스터에 주소값을 넣는 부분에는 항상 "sethi" 명령어가 나타난다는 것을 알 수 있다. 즉, ①에 해당하는 부분을 찾기 위해서는 목적 레지스터가 "r1"인 "sethi" 명령어를 찾으면 된다.

SPARC 에서 수정의 대상이 되는 명령어 중 PC 상대 주소 지정 방식을 사용하는 것은 "call" 뿐이며, 이 명령어는 2비트의 "opcode"와 30비트의 "displacement"로 구성된다. 이 때 타겟 어드레스는 "displacement"의 값을 왼쪽으로 2비트 로지컬 쉬프트한 값과 PC값을 더해서 구해진다. 이 경우 새 버전의 프로시저에서 구 버전 프로세스 어드레스 스페이스 상의 심벌을 액세스하도록 해야 하는데, 일반적으로 이 명령어의 위치(PC값)가 구 버전과 새 버전에서 다르므로 "displacement" 부분을 수정함으로써 올바른 심벌 위치를 액세스하도록 한다. 구체적으로 구 버전에서의 심벌 위치에서 새 버전 프로시저 내의 "call" 명령어의 위치를 뺀 다음 오른쪽으로 2비트 로지컬 쉬프트한 값으로 "displacement" 부분을 고쳐서 구 버전 심벌의 위치가 "call" 명령어의 타겟 어드레스가 되도록 하면 수정이 완료된다.

다음에는 프로그램에서 외부 심벌을 액세스하는 두 번째 유형으로서, 포인터 상수를 사용하는 경우에서의 외부 심벌 리퍼런스 수정 방법을 살펴본다. 포인터 상수는 일반적으로 포인터 변수를 특정 심벌의 주소값으로

초기화하거나, 참조에 의한 함수 호출을 위하여 포인터를 매개 변수로 하는 경우에 사용된다. 어느 경우에나 어셈블러 상에서 사용되는 방식은 같으므로 포인터 변수를 초기화하는 경우를 이용하여 외부 심벌 리퍼런스 수정 방법을 설명한다. 예를 들어, "p"라는 전역 포인터 변수에 "a"라는 전역 변수의 어드레스를 넣는 코드(p=&a;)는 어셈블러로 다음과 같이 표현된다("p"의 어드레스는 0x21d40번지, "a"의 어드레스는 0x21d58번지):

```
r1 ←0x21d40 - ①
r2 ←0x21d58 - ②
store r2,(r1)  - ③
(memory[r1] ←r2)
```

여기서 레지스터 "r1"은 ①에서 전역 포인터 변수 "p"의 주소값을 갖게 되고, "r2"는 ②에서 전역 변수 "a"의 주소값을 갖게 되므로 ①과 ②에 해당하는 부분을 모두 수정해야한다. 이러한 방법을 사용하면 레지스터 간접 주소 지정 방식의 명령어에서 주소 지정에 사용되는 레지스터의 값까지 모두 수정되므로, 외부 심벌의 이름을 리퍼런스하는 경우의 레지스터 간접 주소 지정 방식의 명령어에 대해서는 별다른 조치를 취하지 않아도 된다.

그러나 위의 예에서 "r2"의 경우는 ③의 명령어 상에서 주소 지정에 직접적으로 사용되는 것이 아니므로, 어셈블러 코드만으로는 ②에서 "r2"에 넣어지는 값이 수정이 필요한 외부 심벌의 주소값인지, 또는 앞의 레지스터 간접 주소 지정 방식의 경우와 같이 수정이 필요하지 않은 일반 정수 상수인지 알 수 없다. 따라서 이 두 가지의 경우를 구분하여 올바른 수정이 이루어지게 하기 위해서는 교체 대상 프로시저에서 프로세스 어드레스 스페이스의 코드 영역과 데이타 영역의 범위에 해당하는 정수 상수값을 사용할 수 없다는 제한점이 발생한다. SPARC 머신 상의 Solaris에서 프로세스 어드레스 스페이스는 항상 0x10000(=65536)번지에서 시작하므로 이 경우, 교체 대상 프로시저에서는 65536 이상의 상수를 사용할 수 없다. 그러나 실제로 교체 대상 프로시저에서 65536보다 큰 정수 상수의 사용이 필요한 경우에는 65536보다 작은 상수를 이용하여 몇 개의 명령어로 원하는 기능을 합성함으로써 이러한 제한점을 제거할 수 있다.

지금까지 설명한 SPARC 머신에서의 외부 심벌 리퍼런스에 대한 수정 방법은 다음과 같이 간략히 요약된다. 우선 명령어 중 레지스터에 상수값을 넣는 경우, 즉 "레지스터 ← 상수"의 형태에서 상수가 0x10000 이상인 경우에는 이 상수값을 수정한다. 또한 "call" 명령어가 나오면 "call" 명령어의 "displacement" 필드를 수정한다. 구체적인 수정 방법은 위에서 설명한 바와 같다. 제안한 수정 방법은 어셈블러 코드 상에서 외부 심벌의 위치를 나타내는 모든 상수값을 수정하므로 컴파일러 최적화가 사용된 경우에 대해서도 적용할 수 있다.

4.2.4 디스크 및 메모리 내용 업데이트

위의 과정이 완료되면 구 버전 프로세스 어드레스 스페이스에 매핑된 새 버전 프로시저의 실제 수정이 이루어진다. 실제의 수정은 수정할 명령어의 수에 따라 수정이 완료된 새 버전 프로시저로 디스크와 메모리 상의 새 버전 프로시저 영역 전체를 덮어쓸 수도 있고, 수정이 필요한 명령어 부분만을 쓸 수도 있다. 또한, 수정할 명령어의 분포 상황에 따라 필요한 부분만을 블록으로 지정하여 덮어쓸 수도 있다. 이러한 수정은 디스크, 메인 메모리, 캐시 메모리의 순서로 수행된다. 디스크 수정의 경우 write(2) 시스템 콜을 사용하여 수행한다.

메인 메모리의 수정은 2.3절에서 설명한 대로 두 가지 방법이 있다. 우선 어드레스 스페이스를 직접 수정하는 경우에는 Solaris에서 제공하는 프로세스 파일 시스템(process file system, "/proc")의 "as"라는 파일을 이용한다. 이 "as" 파일은 현재 실행되고 있는 해당 프로세스 어드레스 스페이스의 이미지를 가지고 있으며 일반 파일과 같이 취급된다[14]. 따라서 디스크 수정의 경우와 마찬가지로 write(2) 시스템 콜을 이용하여 수정할 내용을 직접 쓰면 메인 메모리의 내용이 수정된다. 다음으로 페이지를 무효화시키는 방법을 위해서는 프로세스 어드레스 스페이스에 대한 여러 가지 제어 기능을 지원하는 memcntl(2) 시스템 콜을 사용한다[14]. 이 시스템 콜을 사용하여 새로운 프로시저가 매핑된 영역을 포함하는 페이지에 대해 페이지 무효화를 수행하면, 이후 해당 영역이 실행될 때 무효화된 페이지를 이미 수정이 완료된 디스크로부터 읽어 오므로 메인 메모리의 내용도 수정된다.

캐시 메모리의 수정에 있어, SPARC 시스템은 명령어 캐시와 데이타 캐시가 분리되어 있고 명령어 캐시에 대해서는 어드레스 스페이스의 수정 내용이 캐시 메모리에 반영되지 않는다. 따라서 메인 메모리의 수정이 이루어진 다음에는 캐시 메모리의 내용을 수정하여야 한다. 이를 위하여 Solaris에서 제공하는 sync_instruction_memory(3C) C 라이브러리 함수를 사용한다[14]. 이 함수는 주어진 어드레스 범위에 해당하는 명령어 캐시의 엔트리를 무효화하는 기능을 가진다. 따라서 메인 메모리 수정이 완료된 후 이 함수를 실행하면,

이후 프로세서에서 명령어들을 실행할 때 다시 메인 메모리로부터 읽어 오게 되므로 캐시의 내용도 새로운 버전의 것으로 변경된다.

페이지 무효화를 수행하는 memcntl(2) 시스템 콜과 캐시 동기화를 수행하는 sync_instruction_memory(3C) 함수는 mmap(2) 시스템 콜과 마찬가지로, 자신을 호출한 프로세스의 어드레스 스페이스에 대해서만 그 기능을 수행할 수 있으므로 4.2.2절에서 밝힌 바와 같이 시그널을 이용해 구현된다.

4.2.5 교체 시기 선정

Solaris의 pstack(1)은 프로세스의 런타임 스택의 내용을 화면으로 출력하는 기능을 갖는 사용자 명령어이다[14]. 따라서 pstack(1)의 결과로 나온 런타임 스택에 교체 대상 프로시저의 이름이 존재하면 이 프로시저가 실행 중임을 알 수 있으며, 이 경우에는 임의의 시간 동안 대기한 다음 다시 런타임 스택의 내용을 점검한다. 그렇지 않은 경우에는 교체 대상 프로시저가 실행 중이 아니므로 교체를 위한 다음 과정으로 넘어 간다.

4.2.6 구 버전 프로시저 수정

교체의 마지막 과정으로 구 버전의 프로시저를 수정해야 하는데 그 과정에서 교체 대상 프로시저가 새로 호출되어 실행되면 오동작이 발생할 수 있다. 이러한 오동작을 방지하기 위하여 3.2절에서 설명한 대로 감시 영역을 설정하는 방법을 사용하며, Solaris에서는 워치포인트 트랩 (watchpoint trap)을 사용하여 이를 구현한다[14]. 워치포인트 트랩은 원래 프로그램의 디버깅을 위해 사용되는 것으로, Solaris 환경에서는 프로세스 파일 시스템(process file system, "/proc")의 "ctl"파일을 이용하여 워치포인트 트랩을 설정할 수 있다[14]. 이 파일에 감시영역의 시작 위치와 크기 정보를 포함하는 정형화된 제어 메시지를 write(2) 시스템 콜을 이용하여 전달함으로써 감시영역을 설정할 수 있고 또한 비슷한 방법으로 이를 해제할 수 있다.

```
open("/proc/pid/ctl");
/* open ctl file to set watchpoint trap
 * ctl
 *      A write-only file to which structured messages are written
 *      directing the system to change some aspect of the process's
 *      state or control its behavior in some way.
 */
write(void* (PCWATCH struct prwatch prwatch_t), size_t size);
/* PCWATCH
 *      set or clear a watched area in the controlled process from
 *      a prwatch structure operand:
 * typedef struct prwatch{
 *      uintptr_t  pr_vaddr; /* virtual address of watched area */
 *      size_t          pr_size; /* size of watched area in bytes */
 *      int             pr_wflags; /* watch type flags */
 *}prwatch_t;
 */
write(void* (PCSFAULT FLTWATCH), size_t size);
/* PCSFAULT
 *      define a set of hardware faults to be traced in the process
 * FLTWATCH watchpoint trap
 */
```

그림 10 워치포인트 트랩을 이용한 프로세스 제어

그림 10은 이러한 제어 메시지를 이용하여 워치포인트 트랩을 설정하는 과정을 의사 코드 형식으로 나타낸다. 먼저 "PCWATCH"라는 제어 메시지를 이용하여 시작 어드레스(pr_vaddr)와 크기(pr_size), 워치 타입(pr_wflags)을 정의하여 감시 영역을 지정하고 "PCSFAULT"라는 제어 메시지를 이용하여 워치포인트 트랩을 설정한다. 이와 같은 방법으로 워치포인트 트랩을 교체 대상 프로시저의 영역에 대하여 설정해 놓으면 대상 프로시저가 실행되려고 할 때 트랩이 발생하고 해당 프로세스가 정지하게 되어 교체 도중의 오동작을 방지할 수 있다.

워치포인트 트랩이 설정되면 구 버전 프로시저의 시작 부분을 새 버전의 프로시저가 매핑된 위치로 점프하는 동작을 수행하도록 수정한다. 이러한 동작은 "sethi"와 "jmpl"명령어의 조합으로 이루어지며 실제 수정은 4.2.4절에서 제시한 과정을 따라 수행한다. 마지막으로 설정된 감시 영역에 대해서 워치 타입의 값을 클리어하여 "PCWATCH" 제어 메시지를 다시 보내어 감시 영역을 해제하는 것으로 구 버전 프로시저의 수정이 완료된다.

4.2.7 프로시저 크기가 증가하지 않는 경우의 교체

지금까지 설명한 방법은 프로세스에서 사용되지 않는 공간에 새로운 프로시저를 매핑하여 교체를 수행하므로 프로시저의 크기 변화에 관계없이 사용할 수 있다. 그러나 프로시저의 크기가 감소하거나 변하지 않는 경우에는 구 버전 프로시저의 영역을 새 버전 프로시저를 위한 공간으로 사용할 수 있으므로, 새 버전 프로시저를 프로세스의 비매핑 영역에 매핑할 필요가 없고 따라서 교체과정을 단순화할 수도 있다. 구체적으로 교체 대상 프로시저의 크기가 감소하는 경우에는 그림 7의 교체 절차 중 "새 프로시저 매핑" 과 "새 프로시저 수정" 과정이 필요하지 않다. 그리고 교체 대상 프로시저의 크기가 변하지 않는 경우에는 "새 프로시저 매핑" 과 "심벌 리퍼런스 수정" , "새 프로시저 수정" 과정이 필요하지 않다.

5. 교체 방법의 구현 및 검증

본 연구에서는 앞서 제시한 연구 결과를 바탕으로

Sun Solaris 2.6 환경에서 프로시저 단위의 온라인 프로그램 교체를 지원하는 소프트웨어를 제작하고, 실제 사용자 응용 프로그램들을 대상으로 온라인 프로시저 교체를 수행함으로써 제안한 방법을 검증하였다.

제작한 교체 소프트웨어는 교체 제어 프로그램과 교체 대상 프로그램에 포함되는 교체 라이브러리로 이루어져 있다. 교체 제어 프로그램인 "chproc"는 사용자에 의해서 실행되어 전반적인 교체 과정을 수행하는 C 프로그램으로서, 소스 코드의 크기는 약 650라인이고 실행 파일의 크기는 약 36KB이다. 이 프로그램의 용법은 그림 11과 같다. 그림 11은 교체 대상 프로시저가 하나인 경우를 나타내며, 교체할 프로시저가 여러 개일 경우에는 교체할 프로시저의 이름을 모두 입력하면 된다.

```
chproc pid procedure_name old_file_name new_file_name

/*  pid : 교체 대상 프로세스 아이디(ID)              */
/*  procedure_name : 교체 대상 프로시저 이름         */
/*  old_file_name : 구 버전 실행 파일 이름           */
/*  new_file_name : 새 버전 실행 파일 이름           */
```

그림 11 교체 프로그램 용법

교체 라이브러리 "chproc.h"는 교체 대상 프로그램에 포함되어 교체 대상 프로그램이 시그널에 관련된 기능을 수행할 수 있도록 해 주는 라이브러리로 시그널 초기화와 시그널 핸들링 함수들로 구성되어 있다. 시그널 핸들링 함수는 새 버전 프로시저의 매핑이나 캐시 동기화같이, 자신을 호출한 프로세스의 어드레스 스페이스에 대해서만 그 기능을 수행할 수 있는 함수들을 사용하는 교체 과정을 수행한다. 교체 라이브러리의 소스 코드 크기는 약 60라인이며, 이 라이브러리를 포함하면 교체 대상 프로그램의 실행 파일의 크기는 포함하지 않은 경우에 비해 약 1.5KB 증가하게 된다.

실제의 온라인 프로시저 교체 환경은 그림 12와 같다. 먼저 교체 라이브러리가 포함된 구 버전의 소스 코드를 컴파일하여 구 버전의 실행 파일을 생성하고 이를 실행시키면, 이 프로그램은 프로세스의 형태로 동작하게 된다. 이후에 이 프로그램의 일부를 변경한 새 버전의 소스 코드를 컴파일하여 새 버전의 실행 파일을 준비하고, 교체 제어 프로세스를 부르면 지정된 프로시저에 대한 온라인 교체가 시작된다.

그림 12에서 보듯이 교체가 시작되면 교체 제어 프로세스는 앞 절에서 제시한 교체 절차에 따라 구 버전과 새 버전 실행 파일의 심벌 테이블 정보를 이용하여, 새 버전 실행 파일에서 새 버전 프로시저를 추출한 다음, 독립된 파일(새 버전 프로시저 파일)로 생성시키고, 이를 교체 대상 프로세스의 어드레스 스페이스에 매핑한 다음, 매핑된 새 버전 프로시저의 외부 심벌 어드레스를 수정한다. 또한, 교체 제어 프로세스는 필요한 경우 교체 대상 프로세스에 시그널을 보내어 교체 과정의 일부를 교체 대상 프로세스에 포함된 교체 라이브러리 내의 함수에서 수행하도록 한다. 교체 과정이 완료되면 교체 제어 프로세스는 종료되며, 이 후 교체 대상 프로세스는 교체 대상 프로시저가 새로운 버전으로 교체된 새 버전의 프로그램으로서 동작하게 된다.

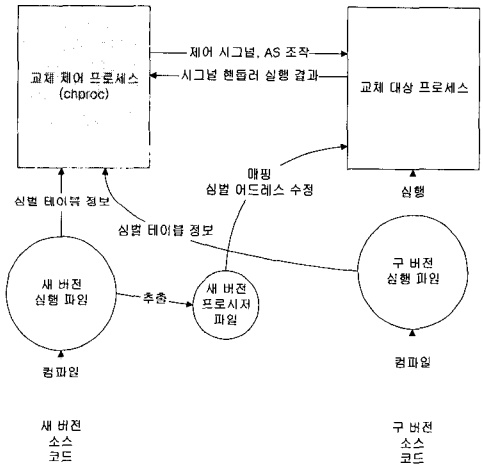

그림 12 교체 실행 환경

6. 결 론

온라인 소프트웨어 교체는 연속 운전이 요구되는 다양한 응용 분야에서 소프트웨어의 업그레이드에 의한 동작 정지 시간을 최소화하기 위한 방법이다. 온라인 소프트웨어 교체의 실용화를 위해서는 다양한 교체 단위와 교체 대상 소프트웨어의 종류에 효율적으로 대응하기 위한 다양한 교체 방법의 개발이 필요하다. 본 논문에서는 프로시저 단위의 온라인 프로그램 교체를 수행하기 위한 일련의 방법들을 제시하고, 제안한 방법을 SUN Solaris 2.6 환경에서 사용자 응용 소프트웨어를 대상으로 적용하여 검증하였다.

프로시저 단위의 온라인 프로그램 교체에서는 이미 실행되고 있는 프로세스의 코드 영역을 이 프로세스의 동작에 영향을 주지 않고 프로시저 단위로 새로운 버전

으로 교체한다. 본 연구에서는 이러한 온라인 프로시저 교체를 실현하기 위한 일련의 방법과 절차를 제시하였다. 제안한 방법에서는 교체 중의 오동작을 막고 성능 저하를 최소화하기 위해 교체 시기의 선정을 위한 상태 관찰, 교체 중의 오동작 방지, 그리고 프로시저 교체의 세 과정을 통해 교체를 수행한다. 특히 본 논문에서는 온라인 프로시저 교체에서의 핵심적인 문제로서 어떻게 새 버전의 프로시저를 구 버전의 프로세스 어드레스 스페이스에서 올바르게 동작하도록 외부 심벌 리퍼런스를 수정할 것인가 하는 문제에 대하여 일반적인 RISC 머신에 적용 가능한 일관적인 해결책을 제시함으로써 프로시저 단위의 온라인 프로그램 교체를 가능하게 하였다.

제안한 방법에서는, 다수의 버전을 만들어 놓고 각 버전을 가리키는 포인터를 바꿔주는 방법을 사용하여 온라인 교체를 수행하는 기존의 연구와 달리, 프로그램의 어드레스 영역을 직접 수정함으로써 교체를 수행한다. 제안한 방법은 최소한의 교체 단위인 프로시저 단위의 교체를 지원하고 교체를 위해서는 대상 프로시저에 해당하는 어드레스 영역만을 수정하면 되므로, 다른 단위의 교체나 다수의 버전을 이용한 교체에 비해 간단하고, 따라서 교체가 해당 소프트웨어의 동작과 성능 저하에 미치는 영향을 줄일 수 있으며, 기존 소프트웨어에 내재된 결함을 제거하는 경우와 같이 프로그램 버전간의 변화가 적은 경우에 매우 효율적으로 교체를 수행할 수 있다. 또한 제안한 방법에서는 기존의 운영체제에서 제공하는 서비스들만을 이용하여 온라인 프로그램 교체가 가능하다. 그리고 이에 따라 별도의 소프트웨어 도구나 환경을 필요로 하지 않고, 평상시 사용자 프로그램의 성능에 영향을 미치지 않으며, 기존의 소프트웨어에 간단한 시그널 핸들링만을 추가함으로써 이에 대한 온라인 교체를 가능하게 한다. 본 논문에서 제시한 교체 방법은 프로시저의 외부 인터페이스가 바뀔 때 그리고 새로운 프로시저를 추가할 때에도 적용할 수 있다.

온라인 프로그램 교체의 실용화를 위해서는 앞으로도 많은 연구가 필요하다. 본 연구와 관련하여 앞으로 데이타의 온라인 교체에 대한 연구가 필요하고, 본 연구에서 제안한 방법을 실제의 대규모 소프트웨어에 적용하여야 하며, 이를 지원하기 위한 버전 관리와 사용자 인터페이스 등을 포함하는 교체 도구의 개발이 필요하다.

참 고 문 헌

[1] M. E. Segal and O. Frieder, "On-The-Fly Program Modification: Systems for Dynamic Updating," *IEEE Software*, Mar. 1993.

[2] R. S. Fabry, "How to Design a System in Which Modules Can Be Changed on the Fly," *Proc. 2nd Int'l Conf. Software Eng.*, 1976.

[3] H. Goullon, R. Isle, and K. Lohr, "Dynamic Restructuring in an Experimental Operating System," *IEEE Trans. Software Eng.*, Vol. 4, No. 4, July 1978.

[4] B. Liskov, "Distributed Programming in Argus," *Comm. ACM*, Mar. 1988.

[5] T. Bloom, "Dynamic Module Replacement in A Distributed Programing System," *Doctoral Dissertation, MIT Press, Cambridge, Mass.* 1983.

[6] J. Magee, J. Kramer, and M. Sloman, "Constructing Distributed Systems in Conic," *IEEE Trans. Software Eng.*, Vol. 15, No. 6, June 1989.

[7] J. Kramer and J. Magee, "Dynamic Configuration for Distributed Systems," *IEEE Trans. Software Eng.*, Vol. 11, No. 4, Apr. 1985.

[8] O. Frieder and M. E. Segal, "On Dynamically Updating a Computer Program: from Concept to Prototype," *J. System Software*, Vol.14, No.2, Sep. 1991.

[9] D. Gupta and P. Jalote, "On-Line Software Version Change Using State Transfer Between Processes," *Software Practice and Experience*, Vol. 23, No. 9, Sep. 1993

[10] W. Richard Stevens, *Advanced Programing in the Unix Environment,* Addison Wesley, 1992.

[11] *System Interface Guide*, SUN Microsystems, Inc., 1997.

[12] *Linkers and Libraries Guide*, SUN Microsystems, Inc., 1997.

[13] M. J. Bach, *The Design of the Unix Operating System,* Prentice Hall, 1990.

[14] *UNIX Reference Manual*, SUN Microsystems, Inc., 1997.

[15] Richard P. Paul *Sparc Architecture Assembly Language Programming, &C,* Prentice Hall, 1994.

김 영 진

1998년 한양대학교 전자공학 학사. 2000년 한양대학교 전자공학 석사. 관심분야는 Reliable and Reconfigurable Computing, Internet Systems.

김 형 곤

1997년 한양대학교 전자공학 석사. 1999년 한양대학교 전자공학 석사. 1999년 ~ 현재 삼성전자 반도체총괄 메모리 개발사업부 Flash 팀 연구원. 관심분야는 Reliable and Reconfigurable Computing, Flash Memory Systems.

김 화 준

1997년 한양대학교 전자공학 석사. 1999년 한양대학교 전자공학 석사. 1999년 ~ 현재 삼성전자 정보가전총괄 Storage 사업부 HDD 팀 연구원. 관심분야는 Reliable and Reconfigurable Computing, Storage Systems.

이 인 환

1979년 서울대학교 전기공학 학사. 1985년 서울대학교 전기공학 석사. 1994년 University of Illinois at Urbana-Champaign 컴퓨터공학 박사. 1979년 ~ 1986년 국방과학연구소 연구원. 1994년 ~ 1997년 Tandem Computers Inc. 연구위원. 1997년 ~ 현재 한양대학교 전자전기공학부 조교수. 관심분야는 Reliable and Reconfigurable Computing, Networked Systems.