임베디드환경에서의 검사점 구현

박상준° 국중진 홍지만
광운대학교 컴퓨터공학과
hidRomeo@gmail.com°, {tipsiness, gman}@kw.ac.kr

Implementation of Checkpointing in Embedded Environment

Sangjun Park°, Jungjin Kook, Jiman Hong
Dept. of Computer Engineering, KwangWoon University

요 약

검사점 및 복구 도구는 사용자 응용 프로그램의 상태를 주기적으로 안정된 저장소에 저장을 하고, 결함이 발생하였을 경우 가장 최근의 검사점으로부터 효율적으로 복구하게 하는 도구이다. 특히 검사점 및 복구 도구는 장시간 수행 되는 프로세스를 위해서는 아주 중요한 의미를 지니며, 결함으로 인해 장시간 수행 되는 프로세스에 의해 생성된 중간 결과를 잃어버리지 않게 한다. 본 논문에서는 일반 범용 컴퓨터시스템 상에서 구현된 검사점 및 복구들을 리눅스 기반의 임베디드 시스템에 적용시켜 보고, 그 결과를 통해 임베디드 시스템상에서의 검사점 적용의 가능성을 알아본다.

1. 서 론

후방 에러 복구(backward error recovery)기법으로 알려진 검사점 및 복구 기법은 예상하지 못한 결함을 허용할 수 있는 메카니즘으로 소프트웨어 결함 허용 기법에서 아주 중요한 의미를 갖는다. 검사점 및 복구 도구[2][6][7][8][11]는 시스템의 프로세스들이 결함이 없는 상태로 수행 중일 때 각 프로세스의 상태를 주기적으로 안전한 저장소(디스크)에 저장하여 시스템에 결함이 발생하였을 때 저장된 상태로부터 시스템이 다시 시작할 수 있게 한다.

검사점 및 복구 도구는 시스템에 결함이 발생하더라도 시스템이 잃어버리는 작업을 최소화함으로써 시스템에 결함 허용을 제공하기 때문에 장시간 수행되는 프로그램에 결함 허용을 제공하기 위해 아주 광범위하게 사용되어 왔다. 검사점 작성은 프로세스의 휘발성 상태(레지스커, 메모리 등)를 안전한 저장소에 저장하는 절차라고 할 수 있으며, 검사점은 저장된 프로세스의 상태이다. 그리고 이전의 검사점 상태에서 새롭게 시작하는 절차를 복구(recovery)라고 한다.

일반적으로 프로세스의 검사점을 만들기 위해 대상이 되는 실행 프로그램에 주기적으로 인터럽트를 걸고 프로그램의 상태를 디스크에 저장한다. 유닉스 프로세스의 상태는 메모리의 내용(텍스트, 데이터 그리고 스택 세그먼트), 프로세서 레지스터 그리고 오픈한 파일을 포함한다. 프로세서의 결함을 복구 시에는 메모리의 상태와 레지스터 상태가 재구성되어 검사점의 대상이 되었던 프로그램은 마지막 검사점 지점부터 계속 진행되어야 한다.

소프트웨어 결함 허용을 위한 검사점 및 복구에 대한 연구는 일반 범용 컴퓨터 시스템을 대상으로 행해진 것이 대부분이다. 한번의 검사점 수행시마다 프로세스와 관련한 모든 정보를 새로 저장하는 기본적인 검사점방법을 기본으로 시작하여, 프로세스의 정보를 파일에 기록할때 중복되어 저장되는 정보를 최소화함으로써 오버헤드를 줄이기 위한 방법들이 추가로 개발되고 있다. 그러나 검사점에 저장해야할 정보 중 가장 많은 부분을 차지하는 것은 메모리 관련 정보이기 때문에 이런 메모리에 관련된 정보를 특정크기로 나누어 검사점 수행주기마다 변경된 정보만을 저장하는 점진적 검사점[5], fork()를 이용하여 별도의 프로세스에서 정보를 저장하는 일을 수행하는 검사점[1] 등의 방법이 제시되었다. 본 논문에서는 기존의 이러한 방법들을 리눅스를 기반으로 한 임베디드 타겟보드상으로 이식해봄으로써, 임베디드 환경에서의 검사점 적용의 가능성을 알아보고, 그 성능을 측정, 비교한다.

2. 관련연구

결함허용성에 관한 연구는 크게 커널에서의 검사점 기능지원과[1][5], 사용자 라이브러리에서의 검사점 기능지원[2][3]으로 나누어 볼 수 있다. 커널내 시스템콜의 형태로 구현되어지는 방법을 사용하면 사용자 프로그램의 소스코드를 수정할 필요가 없으며, 사용자에게 투명하게 기능을 제공하는 장점이 있다. 반면 라이브러리를 통한 검사점 기능의 사용은 사용자 프로그램의 소스를 수정해야하는 번거로움이 있지만, 다소 복잡한 커널 리빌드가 필요없는 장점이 있다. 그 외에도 POSIX 멀티스레드 프로세스의 검사점을 작성을 지원하는 유저영역 라이브러리도 개발되어져있다[4].

3. 설계 및 구현

3.1 검사점

본 논문에서 구현한 검사점은 커널내의 시스템콜의 형태로 추가되었으며, 단일스레드로 동작하는 프로세스만을 그 대상으로 한다. 검사점의 기본동작은 다음과 같다. 먼저 <그림 1>에서보듯이 리눅스상에서 태스크를 나타내는 대표적인 자료구조인 task_struct를 파일에 저장하고, task_struct를 가리키는 시그널관련구조체, 메모리관련구조체 등의 정보를 차례대로 저장한다. 하나의 태스크가 가지게 되는 가장 큰 정보는 메모리 내용이므로, 이 메모리내용을 어떠한 방식으로 저장하는가에 따라서 버전별로 상당한 성능상의 차이를 보이게 된다. 이러한 메모리를 저장하는 방식은 일반적인 검사점, 점진적 검사점, 페이지 수준의 검사점 그리고 포크 검사점 등으로 나눌 수 있다. 일반적인 검사점은 프로세스의 메모리공간을 파일에 기록할 때마다 모든 메모리내용을 모두 새로 저장하게 된다. 상당히 직관적이며 구현이 간단하다는 장점을 가지지만, 검사점이 수행될 때마다 중복되어져서 저장되는 메모리내용이 크다는 단점이 있다.

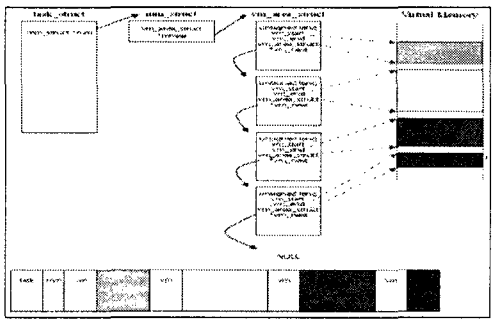

그림1 기본적인 저장정보

점진적 검사점은 최초의 검사점에서는 모든 메모리를 저장한 후에, 두번째 검사점부터는 변경된 VM_AREA 만을 선택적으로 저장함으로써 저장파일의 크기를 줄인다[4]. VM_AREA가 가리키는 영역단위로 메모리내용을 저장하므로, 보다 논리적이고, 그 수가 많지 않아 메모리영역을 파일에 저장하기 위한 탐색시간이 짧은 장점을 가진다.
페이지 수준의 검사점은, 일반적인 검사점과 동일하나 메모리공간의 저장 시에 VM_AREA 단위가 아닌 PAGE_SIZE 크기만큼으로 잘라서 저장하게 된다. 이 방법은 실용성이 없고 동일한 내용의 영역을 잘게 나누어 저장하게 되므로 저장에 필요한 영역탐색시간을 길게만드는 불필요한 오버헤드만을 증가시킨다. <그림 2>에서 보여주듯이, 페이지 수준의 점진적 검사점은, 메모리공간을 PAGE_SIZE 크기로 잘게 나눈 후에, 그 중 변경된 영역만을 저장함으로써 Incremental version 보다 저장되는 정보량를 좀 더 줄임으로써 파일기록 오버헤드를 최소화할 수 있다.

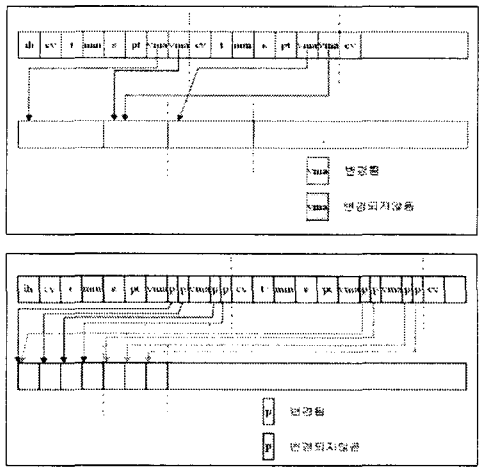

그림2 - 페이지 수준의 점진적 검사점

포크 검사점은 <그림 3>에서 보여주듯이,현재 수행중인 프로세스의 복사본(자식 프로세스) 을 만들고, 그 복사본에서 일반적인 검사점의 검사점을 수행한다. 원래 작업중이던 프로세스(부모 프로세스)는, 스케줄링되는 프로세스의 개수가 늘어나기 때문에 작업종료시간이 조금 길어지기는 하지만, 빠른 시간내에 자신의 작업으로 돌아올 수 있다는 장점을 갖는다.

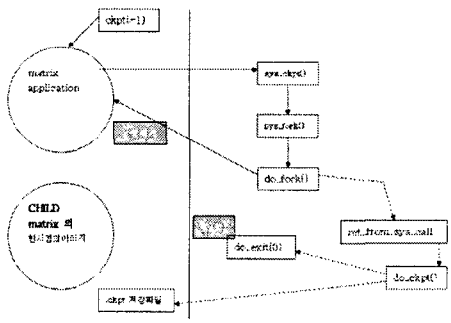

그림3 - 포크 검사점

3.2 임베디드 시스템을 위한 검사점

리눅스 운영체제를 사용하는 PC(Intel CPU 플랫폼)상에 위에서 언급한 다양한 검사점 기법를 모두 구현한 후에, StrongARM을 탑재한 임베디드 시스템 평가 보드에 이를 이식하였다. 각 검사점 기법을 이식 중 발생한 문제와 이를 해결한 방법은 다음과 같다. 첫째로, 메모리의 내용을 가져오기 위해서는 우선 VM_AREA 구조체가 가리키는 가상메모리에 접근할 수 있어야 한다. 그러나, ARM용 리눅스에서는 PC용 리눅스에서 사용했던 방법처럼 메모리로 바로 접근을 할 수가 없다. 따라서 <그림 3>에서 보여주듯이 각 메모리영역의 퍼미션 플래그를 수정하여 접근이 가능하게 하였다.

```
if (tmp->vm_flags == (unsigned long)0x70) {
        tmp->vm_flags = (unsigned long)0x75;
```

그림3 - 퍼미션플래그 추가 코드

두번째는, 읽어온 정보를 파일에 저장하기 위해서 직접 플래쉬 파일시스템에 계속적인 접근을 하게 되면 쓰기 속도가 느려지게 되는 문제점이다. 따라서 플래쉬 파일시스템에 바로 접근하는 것이 아니라, 램 파일시스템에 검사점 결과 파일을 우선 만들어두고, 정보 입력이 완료되면 즉시 영구 저장 장치인 플래쉬 메모리로 복사하는 방식을 취하도록 설계하였다. 이를 위해서는 <그림 4>에서 보여주듯이, 검사점 내부동작에 필요한 llseek 함수를 램 파일시스템의 오퍼레이션 구조체에 추가해서 램파일시스템 상에서 검사점이 수행될 수 있도록 코드를 수정하였다.

```
static struct file_operations ramfs_file_operations = {
    /* ckpt project */
    llseek:     generic_file_llseek,
    /*****************/
    read:       generic_file_read,
    write:      generic_file_write,
    mmap:       generic_file_mmap,
    fsync:      ramfs_sync_file,
};
```

그림4 - ramfs의 file-operation 함수추가 코드

4. 성능평가

테스트를 위해서 사용된 임베디드 시스템 평가 보드는 StrongArm 1110 을 탑재한 Hybus Hyper104이며, 크로스 컴파일을 위한 호스트 개발 환경은 커널버전 2.4.18의 리눅스 PC를 이용하였다. 임베디드 시스템 평가 보드 상에서의 테스트용 프로그램은 행렬곱을 수행하는 응용을 사용했으며, 임베디드 시스템 평가 보드의 메모리의 한계 때문에 50 * 50 의 작은 행렬에 대한 연산을 수행하였다. 성능 측정을 위해서 행렬 연산 프로그램내의 루프에서 검사점 시스템 콜을 수행하도록 설정하여 총 50번의 검사점이 호출되도록 하였고, <표 1>에서 보여주듯이 프로그램이 시작해서 끝날 때까지 시간을 측정하였다.

버전	Basic	Page	Inc	Inc Page	Forked
시간 (S)	0.3366	0.5220	0.2184	1.3647	1.0352
파일크기 (KB)	1121 * 50	1121 * 50	6400 * 1	1935 * 1	1096 * 20

표1 - 임베디드보드에서의 행렬곱 프로그램의 수행시간

또한 <표 2>에서 보여주듯이, 리눅스가 탑재된 PC상에서도 각 검사점 기법별로 검사점 성능을 측정하였다. 이는 측정된 절대속도보다는 버전 간 성능차이가 중요함을 보이기 위해서이다.
실험 결과를 보면 점진적 검사점의 경우, PC에서 수행할 때보다도, 다른 버전의 검사점 기법과 비교해보았을 때, 훨씬 빠른 성능을 보여주었다. 이것은 하드디스크로 접근하는 대신 중간 단계로 램 파일시스템을 사용하였기 때문이다. 이것은 일반 범용 컴퓨터상에서도 점진적 검사점 기법을 사용할때, 추가적인 램파일시스템을 이용한다면 그 수행속도를 훨씬 빠르게 할 수 있다는 것을 의미한다.

버전	Basic	Page	Inc	Inc Page	Forked
시간 (S)	2.2400	2.7600	0.4000	19.6799	0.1233
파일크기 (KB)	1367 * 100	1367 * 100	18384 * 1	4046 * 1	1367 * 100

표2 - x86 PC 에서의 행렬곱 프로그램의 수행시간

포크 검사점의 경우 경우, 검사점을 위해 별도의 프로세스가 생성되고 심지어 원래의 프로세스가 종료한 후에도 메모리상에 남아서 검사점을 수행하는 경우도 있었다. 그만큼 별도의 메모리와 CPU 자원을 소모하기는 하지만, 원래 작업이 수행을 중단하지 않고, 검사점을 수행하기 때문에, 임베디드 시스템 환경에서도 유용하게 사용될 수 있음을 보여준다.

5. 결론

본 논문에서는 여러 방법을 사용해 설계되어진 범용 컴퓨터용 검사점들을 실제 임베디드 시스템 상에 이식해보고 그 성능을 측정해보았다. 일반 범용컴퓨터에서도 검사점파일 생성작업을 위해서 램파일시스템을 도입한다면, 훨씬 빠른 수행을 얻을 수 있을 것이라는 결론을 얻을 수 있었다. 따라서, 임베디드 시스템 환경에서도 결함허용을 위해 검사점 기법을 적용할 수 있음을 보였다.
향후에는 별도로 구현되어진 각 버전의 검사점을 하나의 시스템콜로 통합하여 좀 더 사용하기 쉬운 인터페이스를 제작하고, 점진적 버전 검사점을 사용할 때의 탐색속도 저하를 방지하는 버전을 추가로 구현하여, 임베디드 평가 보드 상에 이식할 생각이다.

6. 참고문헌

[1] Kckpt : An Efficient Checkpoint and Recovery Facility on UnixWare Kernel - ISCA 15th International Conference on Computers and Their Applications Mar. 2000, pp.303-308, New Oreleans, USA.

[2] Libckpt : Transparent Checkpointing under UNIX," USENIX Winter 1995 Technical Conference, 1995.

[3] http://www.cs.utk.edu/~plank/plank/www/libckpt.html

[4] User-level Checkpointing of POSIX Threads

[5] Ickpt : An Efficient Incremental Checkpointing Using Page Writing Fault - Preceedings of The 31th KISS Spring Conference. 2004, Spring, pp.145-147