효율적인 페이지 단위 점진적 검사점의 설계 및 구현

이상호[O1] 허준영[1] 조유근[1] 홍지만[2]
서울대학교 컴퓨터공학부[1] 광운대학교 컴퓨터공학부[2]
{shyi[O], jyheo, cho}@ssrnet.snu.ac.kr[1] gman@daisy.kw.ac.kr[2]

Design and Implementation of an Efficient Page-level Incremental Checkpointing

Sangho Yi[01], Junyoung Heo[1], Yookun Cho[1] Jiman Hong[2]
School of Computer Science and Engineering, Seoul National University[1]
School of Computer Science and Engineering, Kwangwoon University[2]

요 약

점진적 검사점은 검사점 사이의 변경된 상태만을 기록하는 방식으로 검사점 오버헤드를 줄이는 기법으로 알려져 있다. 본 논문에서는 효율적인 점진적 검사점의 설계 내용과 함께 리눅스 커널 2.4.20에서 구현한 기법에 대해 설명한다. 이 논문에서 설명하는 점진적 검사점은 리눅스 운영체제에서 제공하는 페이지 쓰기 결함을 이용하여, 변경된 페이지만을 새로운 검사점에 저장한다. 이 점진적 검사점의 실험 결과는 비점진적 검사점을 사용한 것에 비하여 상당히 오버헤드를 줄일 수 있음을 보여준다.

1. 서론

후방 에러 복구 기법으로 알려진 검사점 기법은 예상하지 못한 결함을 허용할 수 있는 메카니즘으로 소프트웨어 결함 허용 기법에서 아주 중요한 의미를 갖는다. 검사점은 시스템의 프로세스들이 결함이 없는 상태로 수행 중일 때 각 프로세스의 상태를 주기적으로 안전한 저장소(하드디스크)에 저장하여 시스템에 결함이 발생하였을 때 저장된 상태로부터 시스템이 다시 시작할 수 있게 한다.

특히 점진적 검사점 기법은 인접한 두 개의 검사점 간에 페이지들을 비교 분석하여, 변경된 페이지만 저장하게 된다. 이는 비 변경 페이지를 저장하지 않게 되므로, 검사점의 수행에 있어서 시간 및 공간적으로도 상당한 성능 향상을 가져온다. 그러나 이 점진적 검사점 기법은 오래된 검사점 파일들의 쓰레기 수집 문제가 있다. 일반 검사점 기법에서는 가장 최근의 검사점 파일 하나만 유지하고 나머지 오래된 검사점 파일들을 지울 수 있다. 반면에 점진적 검사점 기법에서는 프로세스의 메모리 페이지들이 여러 검사점에 흩어져 있어 오래된 검사점들을 지울 수가 없다. 점진적 검사점 기법에서는 같은 페이지에 대해 여러 버전이 저장되므로 누적된 검사점의 크기는 점점 늘어나게 된다.

본 논문에서는 효율적인 페이지 단위 점진적 검사점인 PICKPT의 설계 내용과 함께 리눅스 커널 2.4.20에서 구현하는 기법에 대해 설명한다. PICKPT는 리눅스 운영체제에서 제공하는 페이지 폴트 핸들러와 페이지 쓰기 보호기능을 이용하여, 변경된 페이지만을 새로운 검사점에 저장한다. PICKPT의 실험 결과는 비점진적 검사점 기법을 사용한 것에 비하여 상당히 오버헤드를 줄일 수 있음을 보여준다.

본 논문의 구성은 다음과 같다. 2절에서 관련 연구들을 논의하고, 3절에서 PICKPT의 설계와 구현 기법에 대해 설명한다. 4절에서 PICKPT의 실제 성능에 대해 설명하고 마지막으로 5절에서 결론을 맺는다.

2. 관련 연구

검사점 오버헤드를 줄이기 위하여 몇몇 연구들이 실제 시스템 환경에서 사용되기 위하여 제안되었다. Beck 등은 컴파일러의 도움을 받는 메모리 배제 검사점 방법을 제안하였다[6]. 컴파일러의 자료흐름 분석을 통하여, 읽기-전용 메모리와 쓰기-전용 메모리를 구분함으로써, 저장하지 않아도 될 메모리를 배제시켜 검사점의 오버헤드를 줄이는 방법이다. Plank 등은 Libckpt라 불리는 점진적인 '쓰기 시 복사' 검사점을 제안하였다. 그러나 Libckpt는 사용자의 소스코드를 변경해야만 했다. 한 예로, main() 함수는 반드시 ckpt_target() 이라고 변경해야 했다. Hong 등은 Kckpt를 제안하였다[2]. 이것은 자식 생성 검사점 방법으로, 유닉스웨어와 리눅스 커널레벨에서 구현되었으며, 이 것은 사용자에게 검사점의 투명성을 보장하게 되므로, 이 방법을 사용할 때에는 사용자의 소스코드에 아무런 변경이 필요가 없게 된다.

3. 페이지 단위 점진적 검사점의 설계 및 구현

본 절에서는, 본 논문의 핵심이 되는 페이지 단위 점진적 검사점인 PICKPT의 리눅스 커널에서의 설계 및 구현에 대하여 보인다. 이 PICKPT는 리눅스 커널 2.4.20에 구현하였고, 이것은 단일 프로세서를 갖는 환경에서 개개의 프로세스에 대한 검사점 및 복원 방법을 제공한다. 구현된 전체 소스코드는 http://ssrnet.snu.ac.kr/~shyi 의 웹 페이지에서 다운로드 할 수 있다.

먼저, PICKPT의 점진적 검사점 기본 단위는 32비트 x86시스템의 페이지 크기(4KB)이며, 검사점 및 복원을 위한 새로운 두 개의 시스템 콜을 추가하였다. 이 시스템 콜들을 이용하여 사용자가 원하는 시점에 사용자 프로그램에서 이를 호출하여, 검사점을 만들 수 있게되고 복원을 수행할 수 있게 된다.

또한, 각 페이지 단위로 쓰기 보호 비트를 켜고 끄기 위하여 페이지 단위로 접근 권한을 바꿀 수 있는 ckpt_set() 와 ckpt_clear() 라는 함수를 제작하였다. 그리고 페이지의 변경 유무를 알기 위하여, 페이지 폴트가 발생 시 이를 처리하는 함수인 do_page_fault()를 변경하였다. 이제 각각의 설계 및 구현 내용을 아래에 보인다.

3.1 추가된 두 개의 시스템 콜

앞에서 이야기한 대로, 리눅스 커널 2.4.20에 두 개의 시스템 콜을 추가하였다. 그것들은 아래와 같으며, 각각 검사점 및 복원을 수행하기 위하여 사용자 측에서 커널에 요청하게 되는 사용자 진입점이 된다.

```
int sys_ckpt(pid_t pid, int incremental);
int sys_recover(char *name);
```

위의 sys_ckpt()는 검사점을 수행할 대상 프로세스의 ID와 점진적 검

사점 여부를 가리게 되는 incremental을 인자로 받는다. 만약 incremental이 0 이라면 이는 비점진적 검사점을 만들게 되고, incremental이 0 이외의 값일 경우에는 점진적 검사점을 만들게 된다. 그리고 sys_recover()는 검사점 파일 이름을 인자로 받게 된다. 이 파일로부터 해당 시점의 프로세스의 상태 정보를 읽어 와서 복원하게 된다.

3.2 검사점

검사점의 목적은 프로세스의 모든 상태를 기록한 후, 시스템 장애 시 기록된 프로세스의 상태 정보를 바탕으로 프로세스를 검사점을 수행한 시점으로 복원시키는 것이다[2][3]. 일반적으로, 프로세스의 검사점은 레지스터 집합, 사용중인 라이브러리와 자신 프로그램 코드 등에 관련된 메모리로 구성된다[2]. 검사점으로부터 한 프로세스를 복원하기 위해서는, 복원될 프로세스의 주소공간과 레지스터 집합의 복원을 수행해야 한다. 이러한 프로세스의 정보의 참조는 리눅스 커널에서, task_struct (include/linux/sched.h)자료구조를 보면 쉽게 알 수 있다. 그림 1은 task_struct와 리눅스 커널에서 이와 관련된 다른 자료구조들의 연관도를 보여준다. 이 그림에서 우리는 task_struct와 열린 파일 정보(files), 시그널 구조체(sig), 메모리 구조(mm) 그리고 레지스터 집합(thread)들의 간단한 연결 그림을 볼 수 있다.

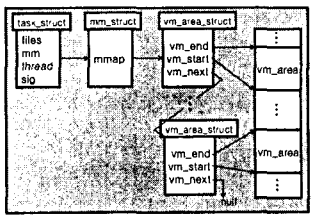

그림 1 task_struct 자료구조

우리는 해당 프로세스의 검사점의 수행을 결정하기 위하여, 리눅스 커널의 task_struct 구조체에 should_ckpt 변수를 추가하였다. 만약 특정 pid(프로세스 ID)가 sys_ckpt()에 인자로 전달되어 시스템 호출을 일으켰다면, sys_ckpt()는 단지 should_ckpt 변수를 세팅하게 된다. 이후에 do_ckpt() 함수가 should_ckpt 변수를 참조한 후, 검사점을 수행할 것인지 아닌지를 결정하게 된다. 그림 2는 sys_ckpt()의 흐름을 보여준다.

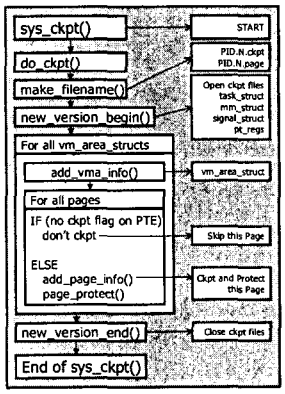

그림 2 sys_ckpt() 흐름

리눅스 커널은 커널모드에서 사용자모드로 변환을 수행할 때(ret_from_sys_call), do_ckpt()를 호출한다. do_ckpt() 함수는 해당 프로세스의 task_struct의 검사점 변수인 should_ckpt를 확인 후, 검사점을 수행한다. 검사점 변수는 3개의 상태를 가질 수 있으며, 그 3가지는 아래와 같다.

0 : 검사점 안함
1 : 비 점진적 검사점 수행
2 : 점진적 검사점 수행

만약 do_ckpt()가 불리게 되면, lckpt는 [pid].N.ckpt 와 [pid].N.page 형식의 이름을 갖는 검사점 용도의 파일을 생성하게 된다. 한 예를 들어서, pid가 1234번 이고, 3번째 점진적 검사점을 갖게 되는 파일의 이름은, 1234.3.ckpt 와 1234.3.page 가 된다. 여기에서 1234.3.ckpt 는 해당 프로세스의 자료구조(task_struct, mm_struct, signal_struct, ...)와 vm_area_struct 및 페이지들에 대한 메타 데이터를 담는 파일이고, 1234.3.page 는 해당 프로세스의 주소공간에서, 페이지 단위로 쪼개진 실제 메모리 데이터를 담는 파일이다. 그림 3은 검사점 파일의 포맷을 보여준다. 리눅스 커널에서는 프로세스의 주소공간을 세그먼트와 페이지의 2단계로 다루며, 이것의 구조는 vm_area_struct에서 관리된다. 각 vm_area_struct는 여러 개의 페이지로 구성되며, 이 페이지들은 vm_area_struct 자료구조의 vm_start 주소와 vm_end 주소 사이에 존재하게 된다. 즉, 32비트 x86 시스템이라면, 페이지의 크기가 4096 바이트로 고정되므로, 하나의 vm_area_struct 자료구조에 포함되는 페이지의 개수는 아래와 같은 식을 따르게 된다.

Number of Pages = (vm_end - vm_start) / 4096

PICKPT는 처음에는 비점진적 검사점을 버전 0번으로 주고 검사점을 수행하기 시작한다. 이후의 1, 2, 3,..., N 의 검사점은 검사점 사이에서 변경된 정보만 기록하는 방식으로 점진적 검사점을 수행한다. lckpt는 이 점진적 검사점을 수행하기 위하여, 프로세스의 주소공간을 모두 읽기-전용으로 바꾸게 된다. 따라서, 검사점 사이에서 변경이 일어날 때에는 항상 페이지 폴트가 발생하며, 이는 리눅스 커널의 페이지 폴트 핸들러인 do_page_fault() 함수에서 수행된다. PICKPT에서는 어떠한 페이지가 변경이 일어났다는 사실을 쉽게 알 수 있게 하기 위하여 이 함수를 수정하였고, 페이지 단위로 쓰기 방지 비트를 세팅해주는 함수인 page_protect()와 이를 해제하는 page_unprotect()를 새로이 구현하였다. 그림 3은 변경된 do_page_fault()의 핵심코드를 보여준다.

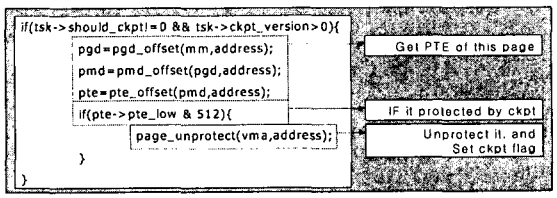

그림 3 do_page_fault() 의 변경된 코드

32비트 x86 시스템은 2단계 페이지 테이블을 갖고 있다. 전체 주소 32비트 중, 첫 10비트는 페이지 글로벌 디렉토리(PGD), 그 다음 10비트는 페이지 테이블 엔트리(PTE), 마지막 12비트는 각 페이지의 4096 바이트 영역을 나타내게 된다. 여기에서, 각 페이지 별 읽기 쓰기 권한 플래그는 PTE에 존재하며 이것의 구조는 아래의 그림 4와 같다.

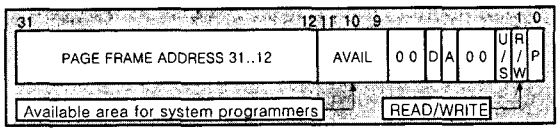

그림 4 32비트 x86 시스템의 Page Table Entry

위의 그림에서 1번 비트는 읽기/쓰기 혹은 읽기 전용으로 권한을 설정하는 비트이며, 9번부터 11번 비트까지는 시스템 프로그래머가 사용하도록 허용해놓았다. 현재의 리눅스 커널에서는 이 비트들을 사용하지 않기에, 우리는 검사점의 구현을 위하여 9번과 10번 비트를 사용하였다. 아래는 9번과 10번 비트의 상태에 따른 의미이다.

9번 비트:
1 : 검사점을 수행함
0 : 검사점을 수행하지 않음
10번 비트:
1 : 검사점을 잡기 위한 목적으로 쓰기 방지 권한을 세팅했음
0 : 원래 쓰기 방지 자료임

PICKPT는 이러한 정보를 바탕으로, 페이지 폴트가 발생 시에 해당 페이지가 검사점 되어야 할지, 혹은 그렇지 아니할 지를 쉽게 알아낼 수 있게 되고, 이 사실을 바탕으로 점진적 검사점을 수행한다.

3.3 복원

시스템 장애에 의하여 비정상 종료된 프로세스를 복원하는 과정은 execve() 시스템 호출과 상당히 유사하다. 다만 다른 점이 있다면, PICKPT의 sys_recover() 시스템 호출에서는 저장된 검사점 정보를 바탕으로 복원된 프로세스를 생성한다는 것이다. sys_recover() 는 파일이름을 단 하나의 함수 호출 인자로 받는다. 인자로 받는 파일은 검사점 정보를 기록하고 있는 파일이다. 그림 5는 sys_recover() 의 흐름을 보여준다.

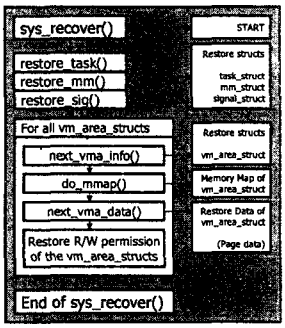

그림 5 sys_recover() 흐름

3.4 검사점 파일 포맷

앞에서 보인 바와 같이 매 번의 검사점 파일은 ckpt 확장자의 프로세스 자료구조 및 메타 데이터 파일과, page 확장자의 실제 프로세스 주소공간의 페이지 내용이 담긴 파일로 나뉜다. 그림 6은 이 둘 파일의 포맷을 보여준다.

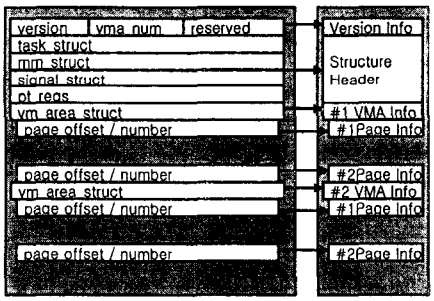

그림 6 검사점 파일 포맷

4. 실험 및 결과

본 절에서는, PICKPT의 점진적 검사점과 비점진적 검사점의 상대적 성능을 평가하기 위하여 여러 가지 사용자 응용 프로그램을 통해 실험하였고 결과 자료를 분석하였다. 실험을 수행한 시스템은 아래와 같다.

System Specification	
Intel Pentium 2.4Ghz	Redhat 9.0
512MB RAM	gcc 2.9.6
5400RPM HDD	Linux Kernel 2.4.20

위의 실험 환경에서 다음과 같은 응용 프로그램들을 수행하였다. 먼저, Matrix Multiplication(MATMUL), Fast Fourier Transform(FFT), Discrete Cosine Transform(DCT) 등의 사용자 응용 프로그램의 소스 코드에 PICKPT의 검사점 시스템 콜을 부르도록 코드를 추가하였고, 이것의 점진적 검사점과 비 점진적 검사점을 수행한 결과를 생성된 검사점 파일의 크기로 비교하였다. MATMUL 프로그램의 경우에는 총 101번의 검사점을 수행하는데, 이 때 비 점진적 검사점을 101번 수행한 경우와, 1번의 비 점진적 검사점을 수행 한 후에 100번의 점진적 검사점을 수행한 것의 차이를 아래의 표로 보였다. FFT의 경우는 총 64번의 검사점을, DCT는 총 56번의 검사점을 수행하였다. 아래의 표는 그 결과를 보여준다.

Performance Evaluation (Checkpoint File Size: Bytes)		
Applications	Full Checkpoint	Inc. Checkpoint
Matrix Multiplication	153,067,520	2,789,376
Fast Fourier Transform	89,391,104	1,921,024
Discrete Cosine Transform	80,052,224	2,109,440

위의 결과에서 우리는 두 검사점 사이에서 변경된 페이지 정보만 찾아 저장하는 점진적 검사점 방식이 월등한 성능을 보인다는 것을 볼 수 있었다. 이는 다시 말하자면, 사용자 주소 공간의 많은 양을 차지하는 라이브러리, 프로그램 스택 등의 자료 변경이 심하게 일어나지 않는다는 것을 말해준다. 또한 점진적 검사점의 작성 시, 저장해야 할 정보가 적어지므로, 디스크 I/O 시간이 줄어서 수행 시간에 있어서도 상당히 나은 결과를 나타내었다.

5. 결론 및 향후 계획

앞에서 보인 바와 같이 PICKPT의 점진적 검사점의 성능은 상당히 좋은 결과를 보여주었다. 이 PICKPT 구현의 기본 철학은 점진적 검사점의 수행 시간을 최소한으로 줄이는 것이었다. 따라서, 검사점의 수행은 빠르게 되지만, 이것의 복원 시간은 그에 비해 상당히 길어지는 것이 단점이 되었다. 검사점의 수행 시간과 복원 시간은 서로 상충 관계에 있어서 둘 중 하나가 양보하거나 절충안을 찾아야 하는 문제가 있다. 만약, 특정한 시스템에서 복원을 여러 번 해야 한다는 가정을 한다면, 복원 시간이 긴 것은 좋지 않은 결과를 가져다 줄 것이다. 앞으로는 이러한 문제를 풀고, 시스템의 특성 및 환경에 따른 최적의 시나리오를 찾아 검사점 및 복원 기법을 맞춤형으로 제공할 수 있도록 해야 할 것이다. 또한 모바일 디바이스 혹은 임베디드 시스템을 위한 비용 효율적인 검사점에 대하여 연구해야 할 것이다.

[참고문헌]

[1] Jiman Hong, Taesoon Park, H.Y Yeom and Yookun Cho, Kckpt : An Efficient Checkpoint Facility on UnixWare, 15th International Conference on Computers and Their Applications, pp. 303-308, March 2000

[2] Julia L. Lawall and Gilles Muller, Efficient Incremental Checkpointing of Java Programs, IEEE Proceedings of Networks, pp. 61-70, June 2000

[3] James S. Plank, Micah Beck and Gerry Kingsley, Compiler-Assisted Memory Exclusion for Fast Checkpointing, IEEE Technical Committee on Operating Systems and Application Environments, Special Issue on Fault-Tolerance, pp. 62-67, December 1995

[4] James S. Plank, Micah Beck and Gerry Kingsley, and Kai Li, Libckpt: Transparent Checkpointing under Unix, Usenix Winter Technical Conference, pp. 213-223, January 1995

[5] M. Beck, J. S. Plank and G.Kingsley, Compiler-Assisted Checkpointing, Technical Report of University of Tennessee, UT-CS-94-269, 1994

[6] Sangho Yi, Junyoung Heo, Yookun Cho, Jiman Hong, Jongmoo Choi and Gangil Jeon, Ickpt: An Efficient Incremental Checkpointing Using Page Writing Fault - Focusing on the Implementation in Linux Kernel, Proceedings of the ISCA 19th International Conference on Computers and Their Applications, pp. 209-212, Mar 2004