Ickpt: 페이지 폴트를 사용한 효율적인 점진적 검사점

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

Ickpt: An Efficient Incremental Checkpointing Using Page Writing Fault

Sangho Yi[O1], Junyoung Heo[1], Sangsu Kim[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 검사점인 Ickpt의 설계 내용과 함께 리눅스 커널 2.4.20에서 구현하는 기법에 대해 설명한다. Ickpt는 리눅스 운영체제에서 제공하는 페이지 쓰기 결함을 이용하여, 변경된 페이지만을 새로운 검사점에 저장한다. Ickpt의 실험 결과는 비점진적 검사점을 사용한 것에 비하여 상당히 오버헤드를 줄일 수 있음을 보여준다.

1. 서론

검사점은 시스템이 심각한 장애를 일으키더라도, 프로세스가 처음부터 재시작 하지 않도록 하는 효율적인 방법이다 [1][2][3][4]. 오랜 기간 동작하는 프로세스에 임의 검사점으로의 복원 방법을 제공함으로써, 심각한 장애로 프로세스가 비정상 종료가 되더라도 기록된 검사점으로 해당 프로세스를 쉽게 복원할 수 있다.

특히 점진적 검사점 기법은 페이지 쓰기 보호[3][5]를 사용하여 검사점간에 변경된 페이지만 저장하는 것이다. 그러나 오래된 검사점 파일들의 쓰레기 수집 문제가 있다. 일반 검사점에서는 가장 최근 검사점 파일 하나만 유지하고 나머지 오래된 검사점 파일들을 지울 수 있다. 반면에 점진적 검사점에서는 프로세스의 메모리 페이지들이 여러 검사점에 흩어져 있어 오래된 검사점들을 지울 수가 없다. 점진적 검사점에서는 같은 페이지에 대해 여러 버전이 저장되므로 누적된 검사점의 크기는 점점 늘어나게 된다[5].

본 논문에서는 효율적인 점진적 검사점인 Ickpt의 설계 내용과 함께 리눅스 커널 2.4.20에서 구현하는 기법에 대해 설명한다. Ickpt는 리눅스 운영체제에서 제공하는 페이지 쓰기 결함을 이용하여, 변경된 페이지만을 새로운 검사점에 저장한다. Ickpt의 실험 결과는 비점진적 검사점을 사용한 것에 비하여 상당히 오버헤드를 줄일 수 있음을 보여준다.

본 논문의 구성은 다음과 같다. 2절에서 관련 연구들을 논의하고, 3절에서 Ickpt의 설계와 구현 기법에 대해 설명한다. 4절에서 Ickpt의 실제 성능에 대해 설명하고 마지막으로 6절에서 결론을 맺는다.

2. 관련 연구

검사점 오버헤드를 줄이기 위하여 몇몇 연구들이 실제 시스템 환경에서 사용되기 위하여 제안되었다. Beck 등은 [6]에서 컴파일러의 도움을 받는 메모리 배제 검사점 방법을 제안하였다. 컴파일러는 자료흐름 분석을 통하여, 읽기-전용 메모리와 쓰기-전용 메모리를 구분함으로써, 저장하지 않아도 될 메모리를 배제시켜 검사점의 오버헤드를 줄이게 된다. Plank 등은 [5]에서 Libckpt라 불리는 투명한 점진적인 '쓰기 시 복사' 검사점을 제안하였다. 그러나 Libckpt는 사용자의 소스코드를 변경해야만 했다. 한 예로, main() 함수는 반드시 ckpt_target() 이라고 변경해야 했다.

Hong 등은 [2]에서 Kckpt를 제안하였다. 이것은 포크된 검사점 방법으로, 유닉스웨어와 리눅스 커널레벨에서 구현되었으며, 이 방법을 사용할 때에는 사용자의 소스코드에 아무런 변경이 필요가 없다.

3. Ickpt 의 설계 및 구현

본 절에서는, Ickpt의 리눅스 커널에서의 설계 및 구현에 대하여 보인다. 먼저, 검사점 및 복원을 위한 새로운 두 개의 시스템 호출은 다음과 같다.

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

3.1. 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, 열린 파일, 사용중인 라이브러리와 자신 프로그램 코드등에 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 알 수 있다. 그림 1은 task_struct와 리눅스 커널에서 이와 관련된 다른 자

료구조들의 연관도를 보여준다. 이 그림에서 우리는 task_struct와 열린 파일 정보(files), 신호 구조체(sig), 메모리 구조(mm) 그리고 레지스터 집합(thread)들의 간단한 연결 그림을 볼 수 있다.

해당 프로세스의 검사점의 수행을 결정하기 위하여, 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. 이것은 후에 do_ckpt()가 변수를 참조한 후, 검사점을 수행할 것인지 아닌지를 결정하게 된다. 그림 2는 sys_ckpt()의 흐름을 보여준다.

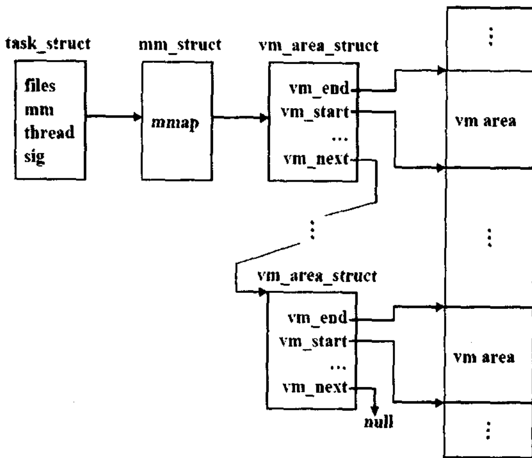

그림 1 task_struct 구조

리눅스 커널은 커널모드에서 사용자모드로 변환을 수행할 때 (ret_from_sys_call), do_ckpt()를 호출한다. 그리고나면 do_ckpt()는 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 sys_ckpt()가 불리게 되면, Ickpt는 [pid].N.ckpt 형식의 이름을 갖는 검사점 용도의 파일을 생성하게 된다. 한 예를 들어서, pid가 1234번 이고, 3번째 점진적 검사점을 갖게 되는 파일의 이름은, 1234.3.ckpt 가 된다. 이 검사점 파일은 헤더와 각각의 vm_area_struct에 대한 버전 헤더를 담는다. 그림 3은 검사점 파일의 포맷을 보여준다. 리눅스 커널에서는 프로세스의 주소공간을 세그먼트 단위로 다루며, 이것의 구조는 vm_area_struct에서 관리된다.

Ickpt는 처음에는 비 점진적 검사점을 버전 0번으로 주고 검사점을 수행하기 시작한다. 이후의 1, 2, 3,..., N 의 검사점은 검사점 사이에서 변경된 정보만 기록하는 방식으로 점진적 검사점을 수행한다. Ickpt는 이 점진적 검사점을 수행하기 위하여, 프로세스의 주소공간을 모두 읽기-전용으로 바꾸게 된다. 따라서, 검사점 사이에서 변경이 일어날 때에는 항상 페이지 폴트가 발생하며, 이는 do_page_fault()에서 수행된다. Ickpt에서는 이 함수를 수정하여, 어떠한 페이지가 변경이 일어났다는 사실을 쉽게 알 수 있고, 이 사실을 바탕으로 점진적 검사점을 수행한다.

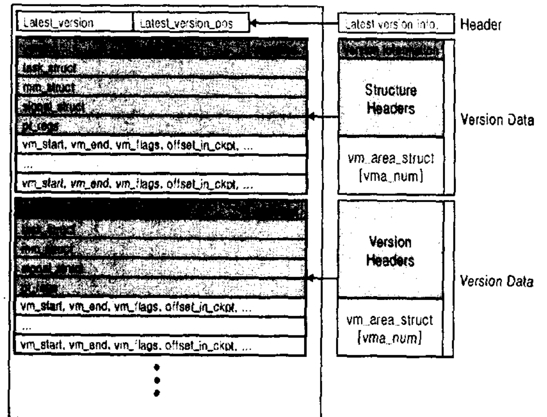

그림 2 검사점 파일 구조

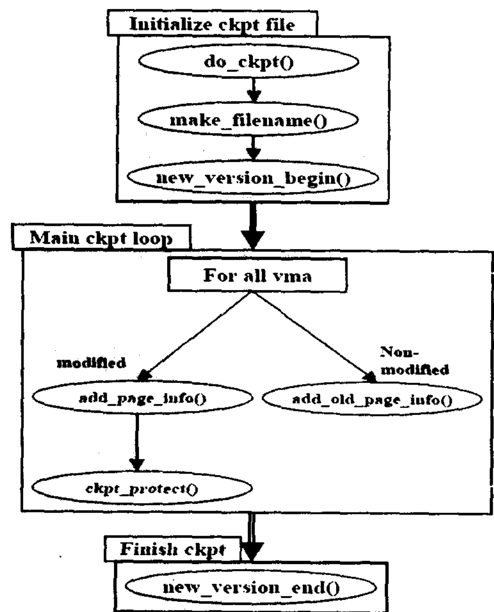

그림 3 sys_ckpt() 흐름도

3.2. 복원

시스템 장애에 의하여 비정상 종료된 프로세스를 복원하는 과정은 execve() 시스템 호출과 상당히 유사하다. 다만 다른점이 있다면, Ickpt의 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 4는 sys_recover()의 흐름을 보여준다.

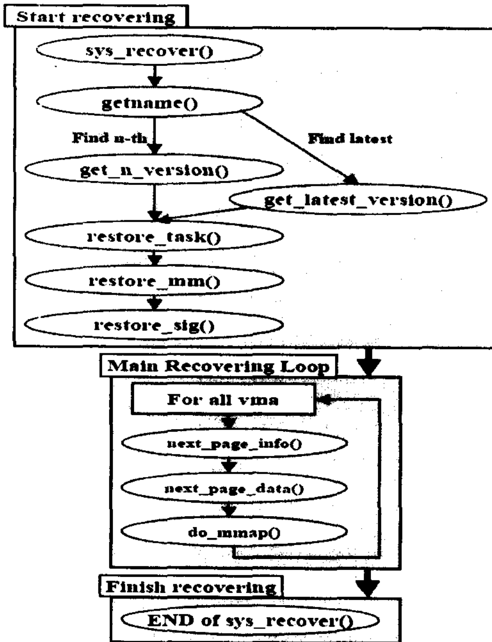

그림 4 sys_recover() 흐름도

4. 성능 평가

이 절에서는, Ickpt의 성능을 보여준다. 성능 평가를 위하여 행렬곱셈 프로그램(matmul), Fast Fourier Transform 프로그램(fft), Discrete Fourier Transform 프로그램(dft) 등의 다양한 프로그램에서 검사점 오버헤드를 비 점진적 검사점을 수행하였을 때와 비교하였다. 우리는 각각의 프로그램에 대하여, 100, 64, 56번의 검사점을 수행하였다. Ickpt의 경우는 단 한번의 비 점진적 검사점을 수행한 후, 나머지 검사점은 모두 점진적 검사점을 수행하였고, normal의 경우는 모두를 비 점진적 검사점으로 수행하였다. 그림 5는 각각의 프로그램에 대하여 검사점 파일의 크기를 보여준다. 이 그림은 Ickpt를 사용할 경우, 기존의 점진적 기법을 사용하지 않는 검사점 방식에 비해, 상당히 오버헤드를 줄일 수 있다는 것을 보여준다.

5. 결론

검사점은 오랜 기간 동작하는 프로세스에 장애 복구 기능을 제공하기에, 매우 중요한 기술이다. 본 논문에서는 리눅스 커널 상에 구현한 점진적 검사점인 Ickpt의 설계 및 구현 방법과 실

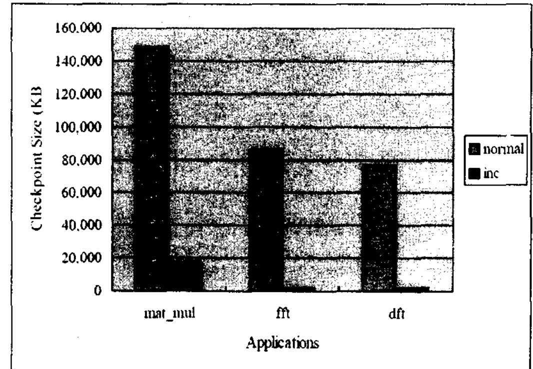

그림 5 검사점 파일 크기 비교

험 결과를 보여주었다. 실험결과는 기존 비 점진적 검사점에 비하여, Ickpt는 검사점 오버헤드를 상당히 줄였음을 보여준다.

[참고 문헌]

[1] E. N. Elnozahy, D. B. Johnson and W. Zwaenepoel, The Pefromance of consistent checkpointing, 11th symposium on reliable distribued systems, pp. 39-47, October 1992

[2] 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

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

[4] 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

[5] 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

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