안정 저장장치의 효율적 사용을 위한 페이지 기반 점진적 검사점 기법

(Page-level Incremental Checkpointing for Efficient Use of Stable Storage)

허 준 영 † 이 상 호 † 구 본 철 †† 조 유 근 ††† 홍 지 만 ††††
(Junyoung Heo) (Sangho Yi) (Boncheol Gu) (Yookun Cho) (Jiman Hong)

요 약 페이지 기반 점진적 검사점은 검사점 오버헤드를 줄이기 위해 프로세스의 메모리 상태 중 변경된 페이지만 저장하는 기법이다. 그러나 점진적 검사점의 누적 크기는 검사점 횟수가 증가함에 따라 서서히 증가하게 된다. 이는 한 페이지가 검사점 작성 이후에 변경되어 검사점 작성시에 검사점에 저장되는 과정이 되풀이 되고, 이후에 삭제되지 않기 때문이다. 복구 시에 프로세스의 저장된 상태를 만들기 위해 검사점들이 모두 필요할 수 있으므로 함부로 검사점을 삭제를 할 수 없다. 본 논문에서는 페이지 기반 검사점 도구인 Pickpt를 소개하고, Pickpt가 검사점의 누적 크기 증가 문제를 해결하는 방법을 설명한다. 실험을 통해 기존 점진적 검사점에 비해 Pickpt가 점진적 검사점의 누적 크기를 현저히 줄임을 보였다.

키워드 : 검사점과 복구, 페이지 기반 점진적 검사점, 결함 허용, 리눅스 커널

Abstract Incremental checkpointing, which is intended to minimize checkpointing overhead, saves only the modified pages of a process. However, the cumulative size of incremental checkpoints increases at a steady rate over time because a number of updated values may be saved for the same page. In this paper, we present a comprehensive overview of *Pickpt*, a page-level incremental checkpointing facility. *Pickpt* provides space-efficient techniques aiming to minimizing the use of disk space. For our experiments, the results showed that the use of disk space using *Pickpt* was significantly reduced, compared with existing incremental checkpointing.

Key words : Checkpoint and Recovery, Page-level Incremental Checkpointing, Fault Tolerance, Linux Kernel

· 본 연구는 2단계 BK21 사업과 숭실대학교 교내연구비 지원으로 이루어졌으며 서울대학교 컴퓨터연구소에서 연구장비를 지원하고 공간을 제공하였음

† 학생회원 : 서울대학교 컴퓨터공학부
jyheo@ssrnet.snu.ac.kr
shyi@ssrnet.snu.ac.kr
†† 정 회 원 : 서울대학교 컴퓨터공학부
bcgu@os.snu.ac.kr
††† 종신회원 : 서울대학교 컴퓨터공학부 교수
ykcho@snu.ac.kr
†††† 종신회원 : 숭실대학교 컴퓨터공학과 교수
jiman@ssu.ac.kr
(Corresponding author임)

논문접수 : 2006년 3월 3일
심사완료 : 2007년 11월 19일

정보과학회논문지: 시스템 및 이론 제34권 제12호(2007.12)

1. 서 론

검사점[1]은 시스템 결함으로 프로세스의 수행이 정지되었을 때, 처음부터 다시 수행하지 않도록 해주는 효과적인 방법이다[1,2]. 주기적으로 검사점을 작성하여 시스템 결함으로 인한 프로세스 정지시에 최근 검사점 상태로 되돌아 갈 수 있기 때문에, 시스템에 결함이 발생할 가능성이 있을 경우에 불필요한 재수행을 방지할 수 있다. 결과적으로 결함 가능성이 있는 시스템내에서 프로세스의 총 수행시간을 줄일 수 있는 유용한 방법이다. 그러나 검사점을 수행하는 과정은 수행과 관련없는 오버헤드를 야기하기 때문에, 검사점 수행을 너무 자주하면 검사점 오베헤드로 인해 오히려 총 수행시간이 길어질 수 있고, 반대로 검사점 작성을 오랫동안 하지 않으면

1) 검사점에는 프로세스의 상태, 메모리 상태, 레지스터 상태 등 프로세스가 수행하는데 필요한 모든 상태 정보들이 포함된다. 따라서 필요한 경우 검사점 작성 시점으로 프로세스를 되돌릴 수 있다.

결함으로 인한 재수행 시간이 길어져 총 수행시간이 길어 질 수 있다. 따라서 검사점 작성 시점을 적절히 정하는 것이 필요하고, 이 시점은 검사점의 오버헤드와 직접적으로 연관이 있다. 즉, 검사점 오버헤드를 줄일 수 있다면 결과적으로 총 수행시간을 단축시킬 수 있다[3-5].

지금까지 검사점 오버헤드를 줄이기 위한 방법들이 다양하게 연구되어 왔다[1,2,6-13]. 이러한 연구들은 크게 두가지 종류로 나누어 볼 수 있는데, 첫째는 검사점 지연 시간을 줄이는 방법이다. 포크 검사점[1], 디스크 없는 검사점[6,7], 압축 검사점[8] 등이 이에 속하는 것이다. 두번째로 검사점 대상의 크기를 줄이는 방법이다. 메모리 배제 검사점[2,9], 점진적 검사점[10-13] 등이 두번째에 속한다. 점진적 검사점과 같은 검사점 대상의 크기를 줄이는 방법은 읽기 전용으로 되어 있는 메모리 영역이나 연속된 검사점 사이에서 변경되지 않는 메모리 영역들을 검사점에서 제외하는 것이다.

이런 검사점 오버헤드를 줄이는 기법들 중, 점진적 검사점이 실제 시스템내에서 널리 사용되는 것 중 하나이다. 점진적 검사점은 변경된 부분들을 찾기 위해 마지막 검사점과 현재 메모리 내용을 모두 비교하거나 페이지 폴트 기법을 사용할 수 있다. 비교하는 방법은 비교 오버헤드가 발생하지만 페이지 폴트 기법은 비교 방법에 비해 훨씬 적은 오버헤드로 변경된 영역을 찾을 수 있다. 단, 페이지 폴트 기법을 제공하지 않는 시스템에서는 사용이 불가능하지만 대부분의 일반 시스템에서는 페이지 폴트 기법을 제공한다.

기존의 점진적 검사점에서 불필요한 검사점 파일의 누적으로 인한 안정 저장장치의 낭비로 인해 점진적 검사점의 실질적 사용에 있어 문제의 여지가 있다[11,14]. 일반 검사점의 복구시에는 마지막 검사점만 사용하여 프로세스의 복구가 가능하지만, 점진적 검사점에서는 프로세스의 마지막 검사점 상태로 복구하기 위해 이전의 검사점들도 필요로 하기 때문에 모든 검사점 파일들을 보관해야 한다. 따라서 검사점을 수행함에 따라 검사점으로 인한 저장장치의 누적 크기가 증가하게 된다. 이는 같은 주소의 페이지가 변경될 때마다 검사점에 남게되어 불필요한 과거 페이지 데이타가 계속 유지되기 때문이다[11,15]. 결국 기존의 점진적 검사점은 안정 저장장치의 낭비를 유발하게 되고 이는 전체 공간 오버헤드에 막대한 영향을 미치게 된다.

본 논문에서는 리눅스 커널 수준에서 구현한 점진적 검사점 도구인 Pickpt를 소개한다. Pickpt는 페이지 폴트 기법을 이용하여 마지막 검사점 이후에 변경된 페이지만 저장하는 페이지 수준 점진적 검사점 기법이다. 또한 앞서 설명한 점진적 검사점의 저장 공간 낭비 문제의 해결방안을 제시하고 Pickpt내에 이를 구현하였다. 실험을 통해 기존의 점진적 검사점 기법에 비해 Pickpt가 저장 공간 오버헤드를 현저히 줄였음을 알 수 있었다.

본 논문의 2절에서는 관련 연구를 설명하고, 3절에서는 Pickpt의 디자인과 구현 내용을 설명한다. 4절에서는 저장 공간을 절약하는 기법을 설명하고 5절에서 기능 및 성능 비교 결과를 설명한다. 마지막으로 6절에서 결론으로 논문을 맺는다.

2. 관련 연구

본 절에서는 지금까지 연구되어온 검사점 기법들에 대해 살펴본다. 검사점 오버헤드를 줄이는 데에 초점을 맞춰 여러 검사점 기법들이 제안되고 구현되었다[2,8,15-18].

Beck 등은 컴파일러를 이용한 메모리 배제 검사점을 제안하였다[2,16]. 이는 소스 코드 내에 프로그래머가 검사점 위치나 특별히 제외할 메모리 영역을 지정하는 지시자(directive)를 사용하게 된다. 컴파일러는 데이타 흐름 분석(data flow analysis)을 통해 쓰지 않는 메모리 영역이나 읽기 전용 메모리 영역을 판별해 내고 이 영역들을 검사점에서 배제하게 된다.

Plank 등은 압축 비교 기법을 이용한 점진적 검사점을 제안하고 이론적으로 성능을 분석하였다[8]. 이 기법은 워드 수준으로 변경된 메모리를 찾으며, 변경된 영역은 압축하여 저장하여 저장시간과 공간을 줄이게 된다.

검사점 구현 기법은 크게 사용자 수준과 커널 수준으로 나눌 수 있다. 사용자 수준은 커널을 수정하지 않아도 된다는 장점이 있지만, 프로그램을 수정해야 하기 때문에 소스 코드 등을 필요로 하는 단점이 있다. 반면에 커널 수준은 커널을 수정하는 반면에 응용 프로그램의 수정 없이 사용이 가능하다는 장점이 있다.

사용자 수준 검사점으로 Plank 등이 개발한 Libckpt[15]와 Litzkow 등이 개발한 Condor 라이브러리[17]가 있다. Plank 등은 Unix환경에서 사용 가능한 사용자 수준의 검사점 도구인 Libckpt를 고안하고 성능을 보였다[15]. Libckpt는 투명한 점진적 검사점과 쓰기 시 복사 기법을 제공한다. Libckpt를 사용하기 위해서는 main() 함수를 ckpt_target()으로 소스 코드를 수정해야 하는 단점이 있다. Litzkow 등은 Condor라는 이름의 검사점 라이브러리를 개발하였다[17]. 이는 사용자 수준에서 구현되었으며 Unix의 시그널 핸들링 기법을 이용하였고 커널 자체의 수정은 필요로 하지 않는다. 그러나 검사점을 사용할 프로세스가 Condor 라이브러리와 링크를 해야 하는 단점이 있다.

커널 수준의 검사점으로 홍 등이 개발한 Kckpt가 있다[18]. Kckpt는 일반 검사점과 포크 검사점을 제공한다. Kckpt는 UnixWare 커널에서 구현되었으며 프로그램의 수정은 일체 필요치 않다.

3. PICKPT의 설계와 구현

본 절에서는 Pickpt의 리눅스[2] 커널내의 구현 방법에 대해 설명한다. 우리는 Pickpt의 구현을 위해 검사점과 복구를 위한 시스템 콜을 다음과 같은 이름으로 추가하였다[19].

int sys_ckpt(pid_t pid);

*int sys_recover(char *name);*

3.1 검사점

보통 검사점은 프로세스의 메모리 내용, 레지스터 내용, 열린 파일 정보 등으로 이루어진다. 복구는 복구용 프로세스를 만들고, 이 프로세스의 내용을 검사점의 메모리 내용, 레지스터 내용 등으로 바꿈으로써 가능하다[11,15,18]. 이 프로세스의 내용은 리눅스 커널 소스에서 include/linux/sched.h 파일의 task_struct라는 구조체를 통해 알 수 있다.

검사점을 지시하는 sys_ckpt 시스템 콜이 불리면 인자로 받은 pid를 사용해 검사점 대상 프로세스를 찾는다. 대상 프로세스를 찾으면 그 프로세스의 task_struct 내에 우리가 추가한 should_ckpt라는 필드의 내용을 1로 변경한다. 이 should_ckpt 필드는 해당 프로세스의 검사점을 작성해야 함을 나타내는 것으로 커널 모드에서 사용자 모드로 전환이 일어나는 ret_from_sys_call()이라는 함수 내에서 필드를 확인하고 실제 검사점 작성 함수인 do_ckpt()를 부르게 된다. 즉, sys_ckpt는 검사점할 대상 프로세스를 찾아 표시만 하고, 실제 검사점은 ret_from_sys_call()이 불릴 때 do_ckpt()에 의해서 이루어진다. 이는 ret_from_sys_call()시에 검사점을 함으로써 커널 스택 내용을 검사점에서 뺄 수 있으므로 가장 안전하고 효율적이기 때문이다.

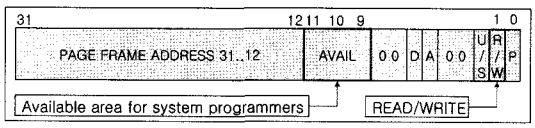

그림 1 i386버전 리눅스 커널에서 페이지 테이블 엔트리

점진적 검사점 구현을 위해 변경된 부분을 찾아야 하는데, Pickpt는 리눅스 커널에서 페이지 폴트 기법을 응용하여 구현하였다. 리눅스 커널 소스에서 arch/i386/mm/fault.c파일의 do_page_fault()를 수정함으로써 가능하다. 페이지 보호를 위해 i386 하드웨어는 페이지마다 페이지 테이블 엔트리가 존재한다. 이 엔트리에서 페이지의 물리 주소와 보호 모드 설정을 알 수 있다. 그림 1은 이 페이지 테이블 엔트리를 나타낸 것이다. 우리는 변경된 부분을 찾기 위해 페이지 테이블 엔트리에서 사용되지 않는 9번째와 10번째 비트를 임의로 사용하였다.

우선, 모든 쓰기 가능한 페이지들을 쓰기 불가능한 페이지로 보호 모드를 변경한다. 이 페이지들에 대해서는 9번째 비트도 1로 변경한다. 이는 변경된 페이지를 찾기 위해 의도적으로 쓰기 보호를 했음을 표시한 것이다. 페이지 폴트 핸들러에서는 쓰기 보호 위반으로 페이지 폴트가 발생하면, 9번째 비트를 검사해본다. 9번째 비트가 1인 경우에는 원래 쓰기 가능한 페이지를 검사점 때문에 쓰기 보호 모드로 바꾼 것이기 때문에, 쓰기 가능 모드로 전환하고 9번째 비트는 0으로 변경하고 10번째 비트를 1로 변경한다. 즉, 이 페이지는 변경되었음을 나타내는 것이다.

다음 검사점을 위해 do_ckpt()함수가 불릴 때, 10번째 비트가 1인 페이지들이 변경된 페이지들이 되는 것이다. 검사점 함수에서 변경된 페이지들을 저장하고, 다시 모든 쓰기 가능한 페이지들을 쓰기 모드로 바꾸고 9번째 비트는 1로, 10번째 비트는 0으로 바꾼다.

3.2 복구

복구는 시스템 콜 execve()와 유사하다. 검사점 파일을 sys_recover()의 인자로 넘겨주면, sys_recover()는 검사점에서 저장된 프로세스의 메모리 영역을 읽어서 현재 프로세스의 이미지를 덮어쓴다. 그리고, 레지스터나 task_struct의 기타 내용들을 바꾼 후, 시스템 콜을 끝내면 복구가 완료된다.

그런데 프로세스의 메모리 영역을 읽기 위해서 이전 검사점들을 모두 확인해야 하는 오버헤드가 발생한다. 이를 줄이기 위해 저장되지 않은 페이지들에 대해 가장 최근 내용이 어느 검사점에 있는지 표시를 해두면 모든 검사점들을 확인해야 하는 오버헤드를 줄일 수가 있다. 그러나 이 방법 역시 최악의 경우 모든 검사점을 확인해야 한다. 따라서, 이를 해결하기 위해서 적당한 시점에 완전 검사점을 하거나 검사점의 저장 장치 사용 공간을 줄일 필요가 있다.

4. 디스크 공간 사용의 최소화

저장 장치의 비효율적인 낭비는 점진적 검사점의 적은 오버헤드 장점을 반감시키는 해결해야 할 문제이다. 일반 검사점에서는 마지막 최근 검사점 파일만 필요하지만, 점진적 검사점에서는 프로세스의 메모리 페이지가 여러 검사점에 퍼져있기 때문에 모든 검사점 파일을 유지해야 한다. 즉, 동일한 주소의 페이지에 대해 여러 버전이 존재하기 때문에 시간이 흐름에 따라 점진적 검사점의 누적 크기는 계속 증가하게 된다[15]. 본 절에서는 점진적 검사점의 저장 장치 비효율성을 개선할 수 있는 두 가지 방법을 제안한다.

2) 리눅스 커널 2.4.20을 사용하였다.

본 절에서 사용할 기호들을 표 1에 정리하였다.

표 1 본 논문에서 사용되는 기호들

C_j	j번째 검사점
$P_{i,j}$	j번째 검사점 파일 안의 i번째 페이지
P_i	프로세스의 현재 상태에서 i번째 페이지
$P_i.count$	프로세스의 i번째 페이지의 현재 count 값
$P_{i,j}.count$	j번째 검사점 파일의 i번째 페이지의 count값
$P_{i,A}.version$	$P_{i,A}$ 페이지가 있는 검사점의 버전
$P_i.AorB$	다음 검사점에서 어느 검사점 파일을 사용할지 나타내는 플래그
C_A	검사점의 그림자 사본 A

4.1 페이지 버전 정보를 이용한 점진적 검사점

저장 장치의 효율적 사용을 위해 검사점을 작성할 때 저장되는 변경된 페이지와 페이지 버전 정보를 같이 저장을 한다. 각 페이지의 버전 정보는 나중에 어떤 검사점이 필요한지 아닌지 판단할 때 사용이 된다. 알고리즘 1에 페이지 버전 정보를 같이 저장하는 방법을 기술하였다.

어떤 검사점 파일이 필요한지 판단하기 위해 *count* 변수를 사용한다. 이 변수는 검사점 내의 해당 페이지가 최신 것인지 판단하는데 사용이 된다. 이 변수는 변경된 페이지가 검사점에 저장될 때 1씩 증가한다. 알고리즘 2에 이 *count* 변수를 이용하여 검사점의 필요 유무를 판단하는 방법을 기술하였다.

알고리즘 1. 페이지 버전 정보를 이용한 점진적 검사점

```
for every page Pi of the process do
    if Pi is modified then
        Pi,j := Pi // checkpoint Pi
        Pi,j.count := Pi.count
        Increment Pi.count
    else
        Pi,j := invalid // skip Pi
    end if
end for
```

알고리즘 2. 제거 가능한 검사점 판별

```
removable := true
for every page Pi,j of checkpoint Cj do
    if Pi,j.count + 1 = Pi.count then
        removable := false
        break
    end if
end for
return removable
```

그림 2는 페이지 버전 정보를 같이 저장한 점진적 검사점의 예를 보여준다. 점선 사각형은 각 검사점 파일을 나타낸다. 그 사각형 안의 원은 검사점 파일내의 저장된 페이지들을 나타낸다. 마지막 검사점 상태로 프로세스를 복구하기 위해서 $P_{1,5}$, $P_{2,5}$, $P_{3,6}$, $P_{4,1}$, $P_{5,4}$, $P_{6,6}$ 페이지들을 모아야 한다. 따라서 C_2, C_3 검사점 파일은 필요하지 않고 삭제가 가능하다. 즉, C_2, C_3 검사점은 알고리즘2에 의해 현재 *count* 값보다 1작은 *count* 값을 갖는 페이지가 전혀 존재하지 않기 때문에 삭제가 가능한 검사점으로 판별이 된다.

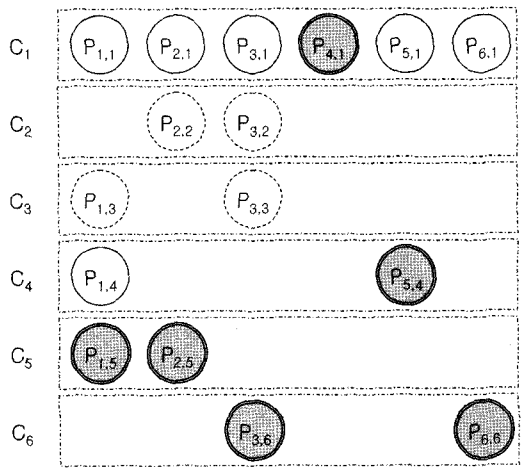

그림 2 페이지 버전 정보를 이용한 점진적 검사점의 예

알고리즘 1은 기존 점진적 검사점 알고리즘에서 페이지 버전 정보를 추가로 저장하기 때문에 수행 시간 $O(N)$에는 변화가 없다. 여기에서 N은 프로세스의 페이지 수이다. 알고리즘 2는 적절한 시점에 수행하여 삭제 가능한 검사점들을 삭제할 수 있다. 이때 검사점 수와 검사점 파일 내의 페이지 수에 따라 수행 시간이 결정된다. 최악의 경우 $O(CN)$이 된다. 여기에서 C는 검사점 개수이고 N은 프로세스의 페이지 수이다. 하지만, 최악의 경우는 점진적 검사점이 전혀 득을 주지 못하는 상황이고, 실질적으로 모든 페이지가 계속 변경된다는 가정이기 때문에 일어날 가능성이 거의 없는 상황이다.

최종 검사점만 필요한 경우라면, 주기적으로 완전 검사점을 수행하여 이전의 점진적 검사점 파일들을 모두 삭제할 수도 있다. 하지만, 분산 시스템 환경에서 여러 버전의 검사점 파일들을 유지해야 하기 때문에 이런 방법이 필요하다[20]. 또한 최종 검사점만 필요한 경우라면 다음의 그림자 사본을 이용한 훨씬 효율적 방법이 가능하다.

4.2 그림자 사본을 이용한 점진적 검사점

마지막 검사점 파일만 필요한 경우, 그림자 사본을 이용한 훨씬 효율적인 방법이 가능하다. 이 방법은 두 검

사점 파일 C_A와 C_B 만을 유지하기 때문에 저장 장치를 낭비하지 않는다. 가장 마지막에 변경된 페이지를 두 검사점 파일 중 하나에 저장하게 된다. 그리고, 이 두 검사점 파일에서 최종 변경된 페이지 만을 선택하여 최종 검사점 상태를 만들 수 있다.

알고리즘 3은 그림자 사본을 이용한 점진적 검사점을 기술한다. 두 검사점 파일 중 하나에 페이지를 저장하기 전에 $P_i.AorB$ 플래그를 먼저 확인한다. 만일 이 플래그가 A라면 페이지를 검사점 C_A에 저장하고, 그렇지 않은 경우 C_B에 저장한다. 그리고 플래그를 A인 경우에는 B로, B인 경우에는 A로 변경한다.

알고리즘 3. 그림자 사본을 이용한 점진적 검사점

```
// Checkpoint j-th version
for every page Pi of the process do
    if Pi is modified then
        if Pi.AorB = A then
            Pi,A := Pi // checkpoint Pi in CA
            Pi,A.version := j
            Pi.AorB := B
        else
            Pi,B := Pi // checkpoint Pi in CB
            Pi,B.version := j
            Pi.AorB := A
        end if
    else
        skip Pi
    end if
end for
Write version number, j in checkpoint
```

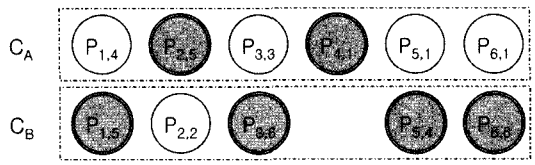

그림 3 그림자 사본을 이용한 점진적 검사점의 예

그림 3은 그림자 사본을 이용한 점진적 검사점의 예를 보여준다. 어두운 원이 최근 페이지를 나타낸다. 최근 검사점 상태를 만들기 위해서는 어두운 원, 즉 최근 페이지들만 모으면 가능하다.

하나의 파일을 사용하여 페이지들을 덮어쓰는 방법은 그림자 사본 방법보다 훨씬 간단하고 저장 장치도 적게 사용하지만, 만일 덮어쓰던 중에 시스템 결함이 발생하면, 복구가 불가능 하게 된다. 따라서, 그림자 사본이라는 또 하나의 검사점 파일을 사용할 필요가 있다. 단, 이때에 덮어쓰던 페이지 이외의 페이지는 데이타를 사용할 수 있다는 가정을 하고 있다. 만일 그렇지 못하다면, 페이지 별로 파일을 작성하는 방법도 가능하다.

앞서 언급했듯이 분산 시스템 환경에서는 노드들간의 데이타 의존성 때문에 마지막 검사점만으로 전역 일치 상태(global consistent state)가 가능해야 이 방법을 사용할 수 있다. 하지만 대부분의 분산 시스템 환경에서는 여러 시점에서의 검사점 파일들이 필요하게 되므로 그림자 사본 점진적 검사점은 사용이 어렵다.

4.3 제안 방법의 안전성

검사점 알고리즘은 결함으로 인한 복구 시에 정상적으로 복구가 가능하도록 프로세스의 모든 메모리 내용을 정확히 저장해 두어야 한다. 일반적으로 검사점 작성이 정상적으로 완료된 후에는 문제가 되지 않지만 검사점 작성 중에 시스템 결함이 발생하면 검사점 자체가 안전하지 않을 수 있다. 이러한 경우에 가장 쉽게 사용하는 방법이 마지막으로 안전하게 저장된 검사점을 복구 지점으로 사용하는 것이다.

제안 알고리즘 1의 경우 검사점 작성 중에 이전에 안전하게 저장된 검사점을 접근하는 일은 없고, 단지 메모리 상에 저장되어 있는 버전 번호만 참조를 한다. 따라서, 검사점 작성시 결함이 발생하더라도 이전에 안전하게 저장된 검사점을 복구 시점으로 사용하는 데에 문제가 없다.

제거 가능한 검사점 판별 알고리즘 2의 경우 제거 가능한 검사점 삭제로 인해 복구 불가능한 상태가 되어서는 안 된다. 마지막 검사점은 여러 점진적 검사점 파일로부터 *count* - 1의 버전을 갖는 페이지들(즉, 가장 최근에 저장된 페이지를 의미함)을 모아서 만들 수 있다. 알고리즘 2에서 이런 페이지들이 하나라도 속해 있는 경우 삭제 불가능으로 판단하기 때문에 이 알고리즘으로 인해 최근 검사점 구성에 필요한 페이지가 삭제되지 않는다. 따라서 알고리즘 2가 최근 검사점 구성에 영향을 주지 않는다.

제안 알고리즘 3의 경우 각 페이지에 대해 가장 최근의 두 버전이 저장되도록 되어 있다. 한 페이지에 대해 새 버전을 저장할 때, 이미 저장되어 있는 두 버전에서 더 오래된 버전을 덮어 쓰게 된다. 따라서 저장하던 중에 결함이 발생하여도 가장 최근 버전에는 문제가 없게 된다. 즉, 최근 검사점을 구성하는 최근 버전의 페이지들이 알고리즘 3으로 인해 영향을 받지 않으므로 안전하게 복구가 가능하다.

5. 성능 평가

본 절에서는 제안 기법인 Pickpt와 기존 기법들과의 기능 측면의 비교와 성능 향상 정도에 대해 살펴본다.

5.1 기능 비교

검사점은 대부분 사용자 수준과 커널 수준 방법으로 구현된다. 사용자 수준은 대부분 사용자 라이브러리 형

태로 커널을 수정할 필요는 없지만 응용 프로그램을 수정해야 하며 검사점 시점이 응용 프로그램 코드 내에서 결정이 되어야 한다[15]. 따라서 구현은 쉽지만 사용하기가 쉽지 않다. 반대로 커널 수준의 구현 방법은 커널을 수정해야 하지만, 응용 프로그램을 수정 없이 바로 사용가능하고, 언제든 필요할 때마다 검사점을 수행할 수 있는 장점이 있다[18]. Pickpt의 경우도 커널 수준에서 구현하여 이런 장점을 갖고 있다.

기존 점진적 검사점 기법으로는 컴파일러를 이용한 기법[2,16]과 마지막 검사점과 현재 프로세스 메모리 내용을 직접 비교하는 방법[8]이 있다. 전자의 경우 컴파일러가 응용 프로그램 코드를 컴파일 하면서 마지막 검사점 이후에 변경되지 않은 내용이나 읽기 전용 영역을 판단한다. 이 정보를 검사점을 작성하는 함수에게 주어 점진적 검사점을 만들 수 있다. 이 방법 역시 사용자 수준의 방법에서의 문제를 가지고 있다. 즉, 사용자가 직접 소스 코드에 검사점 위치를 지정해주어야 하고, 소스 코드가 있는 응용 프로그램에서만 가능하다.

직접 비교하는 방법의 경우 사용자 수준과 커널 수준 모두 구현 가능한 방법이다. 하지만 메모리의 내용을 비교하는 오버헤드가 발생한다. Pickpt의 경우에는 페이지 수준의 점진적 검사점으로 CPU의 페이지 관리 하드웨어를 이용하여 비교 오버헤드 없이 변경된 부분을 찾을 수 있다. 페이지의 크기보다 변경된 부분이 아주 작다면 불필요하게 페이지의 나머지 부분까지 저장해야 하는 단점이 있을 수 있지만, 많은 응용에서 검사점 간격이 너무 짧지 않다면 이런 경우에 불필요하게 저장되는 영역의 크기는 매우 작거나 없는 경우가 대부분이다[21].

5.2 성능 비교

Pickpt의 성능을 측정하기 위해 과학 프로그램에서 많이 사용되고 오래 실행되는 행렬 곱셈 프로그램인 MAT, 빠른 퓨리에 변환 프로그램인 FFT, 이산 코사인 변환 프로그램인 DCT, 퀵 소트인 QSORT 프로그램들을 검사점 대상으로 선택하였다. 각 프로그램들의 수행 시간에 따라 MAT는 100회, FFT는 64회, DCT는 56회, QSORT는 99회 검사점을 수행하도록 하였다.

그림 4는 일반적인 기존 점진적 검사점(IC), 페이지 버전 정보를 이용한 점진적 검사점(IC1), 그림자 사본을 이용한 점진적 검사점(IC2)을 각 대상 프로그램들에게 적용하였을 때의 결과를 보여준다. 각 막대는 해당 프로그램의 모든 버전의 검사점 파일들의 평균을 검사점 파일 크기(단위는 디스크 블록의 수)로 나타낸 것이다. 기대했던 바와 같이 IC1과 IC2는 IC에 비해 모든 프로그램들에 대해 저장 장치를 훨씬 적게 사용하였다. 실험에 사용된 프로그램들의 경우QSORT를 제외하고 IC1과 IC2는 IC에 비해 약 40%정도의 저장 장치를 절약할 수

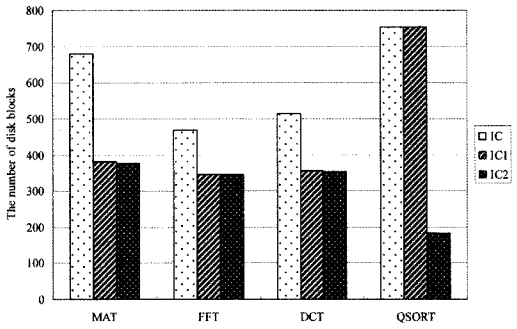

그림 4 각 점진적 검사점 기법의 저장 장치 사용량

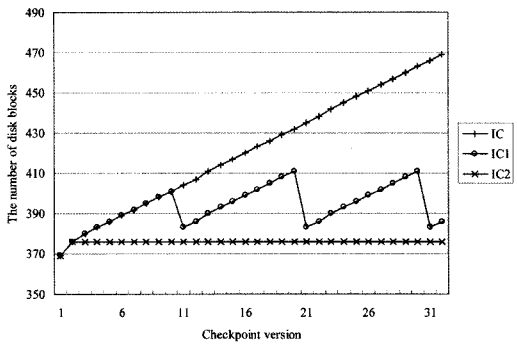

그림 5 행렬 곱셈의 저장 장치 누적 사용량

있었다. QSORT의 경우는 대부분의 검사점 파일들이 서로 다른 메모리 페이지 영역을 가지고 있기 때문에 대부분 지울 수 없는 검사점으로 판별이 되었다. 사용한 QSORT의 경우, 프로그램 특성 상 매번 다른 메모리 영역을 사용하기 때문에 이런 결과가 나타났다.

그림 5는 MAT 프로그램에 대해 각 버전 흐름에 따라 저장 장치 사용의 누적 사용량을 보여준다. 10번째 검사점마다 IC1의 경우 알고리즘 2를 수행하고, 불필요한 검사점 파일들을 삭제한다. 결과적으로 11번째, 21번째, 31번째 검사점에서 누적 크기가 급격히 떨어지게 된다. MAT 프로그램의 경우, 사용하는 메모리 영역이 프로그램 수행 중 거의 비슷하기 때문에 톱과 같은 모양의 그래프가 되었다.

이러한 실험결과를 바탕으로 페이지 버전 정보를 이용한 점진적 검사점과 그림자 사본을 이용한 점진적 검사점이 저장 장치를 효율적으로 사용할 수 있음을 알 수 있다.

6. 결 론

페이지 기반 점진적 검사점은 검사점 오버헤드를 줄이기 위해 프로세스의 메모리 상태 중 변경된 페이지만 저장하는 기법이다. 그러나 점진적 검사점의 누적 크기

는 검사점 횟수가 증가함에 따라 서서히 증가하게 된다. 이는 한 페이지가 변경이 될 때마다 검사점에 저장되고 삭제되지 않기 때문에 검사점의 누적 크기가 증가하게 되는 것이다. 본 논문에서 페이지 기반 검사점 도구인 Pickpt를 소개하였고, Pickpt가 검사점의 누적 크기 증가 문제를 해결하는 방법인, 페이지 버전 정보를 이용한 점진적 검사점과 그림자 사본을 이용한 점진적 검사점을 설명하였다. 실험을 통해 기존 점진적 검사점에 비해 Pickpt의 두 가지 방법이 점진적 검사점의 누적 크기를 현저히 줄임으로써 효율적인 저장 장치 관리가 가능함을 보였다.

참 고 문 헌

[1] J. Hong, S. Kim, and Y. Cho, "Cost analysis of optimistic recovery model for forked checkpointing," *IEICE Transactions on Information and Systems*, E86-D(9), 2003, pp. 1534-1541.

[2] J. Plank, M. Beck, and G. Kingsley, "Compiler-assisted memory exclusion for fast checkpointing," in *IEEE Technical Committee on Operating Systems and Application Environments, Special Issue on Fault-Tolerance*, 1995, pp. 62-67.

[3] A. Ziv and J. Bruck, "An on-line algorithm for checkpoint placement," *IEEE Transactions on Computers*, 46(9), 1997, pp. 976-985.

[4] Nitin H. Vaidya, "Impact of Checkpoint Latency on Overhead Ratio of a Checkpointing Scheme," IEEE Transactions on Computers, Vol.46, No.8, pp. 942-947, 1997.

[5] Andrzej Duda, "The Effects of Checkpointing on Program Execution Time," Information Processing Letters, Vol.16, pp. 221-229, 1983.

[6] J. Plank, K. Li, and M. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, 9(10), 1998, pp. 303-308.

[7] S. Yi, J. Heo, Y. Cho and J. Hong, "Adaptive Mobile Checkpointing Facility for Wireless Sensor Networks," LNCS Vol.3981, pp. 701-709, 2006.

[8] J. Plank, J. Xu, and R. Netzer, "Compressed differences: An algorithm for fast incremental checkpointing," Technical Report CS-95-302, University of Tennessee, August 1995.

[9] J. Plank, Y. Chen, M. B. K. Li, and G. Kingsley, "Memory exclusion: optimizing the performance of checkpointing systems," *Software Practice and Experience*, 29(2), 1999, pp. 125-142.

[10] J. Lawall and G. Muller, "Efficient incremental checkpointing of java programs," in *IEEE Proceedings of the International Conference on Dependable Systems and Networks*, 2000, pp. 61-70.

[11] J. Heo, S. Yi, Y. Cho, J. Hong and S. Y. Shin, "Space-efficient Page-level Incremental Checkpointing," in Proceedings of the 2005 ACM Symposium on Applied Computing, pp. 1558-1562, 2005.

[12] S. Yi, J. Heo, Y.Cho and J. Hong, "Adaptive page-level incremental checkpointing based on expected recovery time," in Proceedings of the 2006 ACM symposium on Applied computing, pp. 1472-1476, 2006.

[13] T. H. Feng and E. A. Lee, "Incremental checkpointing with application to distributed discrete event simulation," in Proceedings of the 38th conference on Winter simulation, pp. 1004-1011, 2006.

[14] J. Heo, S. Yi, J. Hong, Y. Cho, and J. Choi, "An efficient merging algorithm for recovery and garbage collection in incremental checkpointing," in *IASTED International Conference on Parallel and Distributed and Networks*, 2004, pp. 365-368.

[15] J. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt:transparent checkpointing under unix," in *Usenix Winter Technical Conference*, 1995, pp. 213-223.

[16] M. Beck, J. S. Plank, and G. Kingsley, "Compiler-assisted checkpointing," Technical Report, UT-CS-94-269, University of Tennessee, 1994.

[17] M. Litzkow, T. Tannenbaun, J. Basney, and M. Livny, "Checkpoint and migration of unix processes in the condor distributed processing system," Technical Report 1346, Department of Computer Science, University of Wisconsin-Madison, 1997.

[18] J. Hong, T. Park, H. Yeom, and Y. Cho, "Kckpt : An efficient checkpoint facility on unixware," in *International Conference on Computers and Their Applications*, 2000, pp. 303-308.

[19] 조유근, 최종무, 홍지만, "리눅스 매니아를 위한 커널 프로그래밍 Vol. 1", 교학사(주), 2002.

[20] E. N. Elnozahy, Lorenzo Alvisi, Yi-ming Wang and David B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," ACM Computing Surveys, Vol. 34, No. 3, pp. 375-408, 2002.

[21] J. Heo, X. Piao, S. Yi, G. Park, M. Park, J. Hong and Y. Cho, "Impact on the Writing Granularity for Incremental Checkpointing," LNCS Vol. 3614, pp. 1154-1157, 2005.

허 준 영

1998년 서울대학교 컴퓨터공학과 졸업 (학사). 2002년~현재 서울대학교 컴퓨터공학부 박사과정(수료). 관심분야는 시스템 및 운영체제, 무선 센서 네트워크, 결함 허용 시스템, 임베디드 시스템 보안

이 상 호

2003년 고려대학교 전기전자전파공학부 졸업(학사). 2003년~현재 서울대학교 컴퓨터공학부 박사과정(수료). 관심분야는 시스템 및 운영체제, 무선 센서 네트워크, 센서 운영체제, 결함 허용 시스템

구 본 철

2004년 연세대학교 컴퓨터과학과 졸업(학사). 2004년~현재 서울대학교 컴퓨터공학부 박사과정(수료). 관심분야는 시스템 및 운영체제, 무선 센서 네트워크, 임베디드 시스템 및 미들웨어, 소프트웨어 라디오

조 유 근

1971년 서울대학교 졸업(학사). 1978년 미국 미네소타 대학교 컴퓨터 과학 박사 졸업. 1985년 미국 미네소타 대학교 방문 교수. 2001년 한국 정보과학회 회장 1979년~현재 서울대학교 컴퓨터공학부 교수. 관심분야는 운영체제, 알고리즘, 시스템 보안, 결함 허용 컴퓨팅

홍 지 만

2003년 서울대학교 컴퓨터공학부 박사 졸업. 2004년~2007년 광운대학교 컴퓨터공학부 조교수. 2007년~현재 숭실대학교 컴퓨터학부 조교수. 관심분야는 임베디드 운영체제, 결함 허용 컴퓨팅, 무선 센서 네트워크