점진적 검사점에서 복구와 쓰레기 수집을 위한 효율적인 병합 알고리즘

허준영°*, 이상호*, 조유근*, 홍지만**
*서울대학교 컴퓨터공학부
**광운대학교 컴퓨터공학부
{jyheo°, shyi, gman, cho}@ssrnet.snu.ac.kr

An Efficient Merging Algorithm for Recovery and Garbage Collection in Incremental Checkpointing

Junyoung Heo°*, Sangho Yi*, Yoogun Cho* and Jiman Hong**
*School of Computer Science and Engineering of Seoul National University
**School of Computer Science and Engineering of Kwangwoon University

요 약

점진적 검사점은 페이지 쓰기 보호를 사용하여 검사점에서 변경된 페이지만을 저장한다. 점진적 검사점을 사용하면 검사점 오버헤드가 줄어드는 반면에 프로세스의 메모리 페이지들이 여러 검사점에 걸쳐있기 때문에 오래된 검사점들을 병합하거나 지울 수 없다. 본 논문에서는 점진적 검사점에서 복구와 쓰레기 수집을 위한 효율적인 병합 알고리즘을 제안한다. 제안한 알고리즘으로 점진적 검사점들을 병합하여 복구를 위한 완전 검사점을 만들고 불필요한 검사점들을 지울 수 있다.

1. 서론

검사점 오버헤드를 줄이기 위한 여러 연구[1,2,3,4,5,6,7]들이 진행되어 왔다. 이러한 연구들은 두 가지로 분류 할 수 있는데, 하나는 쓰기 시 복사[1], 디스크 없는 검사점[6], 포크 검사점[5], 압축 검사점[5] 같이 검사점 지연을 최소화하는 것과 다른 하나는 메모리 배제 검사점[7]과 점진적 검사점[3,5]과 같이 검사점마다 저장해야 할 데이터의 양을 줄이는 방법이다. 후자에서 중요한 개념은 읽기 전용 메모리와 수정되지 않은 메모리 페이지들이 식별되어 검사점에서 제외되는 것이다.

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]. 따라서 기존의 점진적 검사점의 연구에서는 검사점 오버헤드의 감소에만 주된 관심사가 있었고, 잠재적인 검사점의 오버헤드가 되는 저장장치의 성능에 대한 고려는 거의 없었다.

따라서 본 논문에서는 복구와 불필요한 검사점의 쓰레기 수집을 위한 효율적인 병합 알고리즘을 제안한다. 제안한 알고리즘에서는 프로세스의 메모리 페이지마다 버전과 관련된 추가 정보를 유지한다. 각 페이지는 그 페이지가 삭제되었는지, 수정되었는지 그대로인지를 나타내는 표시(flag)를 갖는다. 제안한 알고리즘은 점진적 검사점이 사용되는 모든 곳에 적용가능하고 단일 프로세서, 다중 프로세서 환경뿐 아니라 분산 컴퓨팅 환경에서 검사점 오버헤드를 줄일 수 있다. 또한 분산 환경의 낙관적 검사점에서 검사점 유지와 병합에 사용될 수도 있다.

본 논문의 구성은 다음과 같다. 2절에서 논문에서 사용되는 표기와 연산자에 대해서 설명한다. 3절에서는 검사점 병합을 위한 개선된 점진적 검사점 알고리즘을 기술하고 4절에서는 점진적 검사점을 위한 효율적 병합 알고리즘과 검사점 병합 예를 보인다. 마지막으로 5절에서 결론을 맺는다.

2. 표기 및 연산 기호

2.1 표기

본 논문에서 사용되는 표기들을 표 1에서 나타내었다.

표 1 논문에서 사용되는 표기들

기호	정의
C_i	i_{th} 점진적 검사점
C_i'	i_{th} 병합 검사점
$P_{p,i}$	C_i에 속한 페이지 정보, p는 페이지 인덱스, i는 버전 번호 $(1 \le i \le n)$
$P_{p,i}'$	병합 검사점의 페이지 정보
$P_{p,i}.data$	$P_{p,i}$의 페이지 내용
$P_{p,i}.flag$	$P_{p,i}$의 변경 표시

2.2 연산 기호

알고리즘을 기술하기 위해 몇몇 표기와 점진적 검사점을 병합하는 연산자를 설명한다. 각 점진적 검사점 C_i는 페이지 정보 자료구조 $P_{p,i}$의 집합이다. $P_{p,i}$에서 p는 페이지 번호이고 i는 버전 번호이다. 다시 말해, i번째 점진적 검사점 C_i는 $\{P_{1,i}, P_{2,i}, \ldots, P_{n,i}\}$이다. 각 페이지의 정보 자료구조 $P_{p,i}$에는 수정 여부를 나타내는 표시($flag$)를 가지고 있다. 이 $flag$는 1, 0, -1중 하나의 값을 갖는데, 1은 이전 검사점 이후에 페이지가 변경되었음을, 0은 페이지가 변경되지 않았음을, -1은 페이지가 삭제되었음을 나타낸다. Ü은 점진적 검사점을 병합하

는 연산자이고 C_i'는 $C_{i-1}\ddot{\cup}C_i$의 결과라고 본다. 연산자 $\ddot{\cup}$는 다음과 같이 정의된다.

- $P_{p,i-1} \in C_{i-1}$ *and* $P_{p,i} \in C_i$ 이고, $P_{p,i}.flag = 0$ 이면 이전 페이지 정보인 $P_{p,i-1}$를 C_i'에 넣고 $P_{p,i}.flag = 1 or -1$ 이면 새 페이지 정보인 $P_{p,i}$를 C_i'에 넣는다.
- $P_{p,i-1} \in C_{i-1}$ *and* $P_{p,i} \notin C_i$ 이면 이전 페이지 정보인 $P_{p,i-1}$를 C_i'에 넣는다.
- $P_{p,i-1} \notin C_{i-1}$ *and* $P_{p,i} \in C_i$ 이면 새 페이지 정보인 $P_{p,i}$를 C_i'에 넣는다.

3. 점진적 검사점 알고리즘

이 장에서는 오래된 검사점을 효율적으로 병합하기 위한 점진적 검사점 알고리즘을 설명한다. 점진적 검사점에서 검사점을 기록할 때, 이전 검사점과 바뀐 부분만 저장한다. 변경되지 않은 부분은 이전 검사점에 저장되어 있기 때문에 복구할 수 있다. 점진적 검사점[1,5]에서 변경되지 않은 부분을 찾기 위해서 페이지 보호 하드웨어를 사용한다. 변경된 부분만을 저장함으로써 각 검사점의 크기와 검사점 오버헤드를 줄일 수 있다.

효율적인 복구와 쓰레기 수집을 위한 검사점 병합을 위해, 각 페이지마다 변경 정보$P_{p,i}.flag$를 유지한다. 기존의 점진적 검사점과 달리 변경된 페이지뿐만 아니라 삭제되거나 변경된 페이지의 정보도 관리한다. 다만 변경된 페이지의 내용만 저장되고, 그 페이지의 변경 정보는 1로 표시한다. 효율적 병합을 위한 점진적 검사점 알고리즘은 알고리즘1에 기술되어 있다.

알고리즘 1. 개선된 점진적 검사점

Input : Current Page Information

FOR every page of process
  IF A page is modified after making C_{i-1}
    set $P_{p,i}.flag = 1$ and $P_{p,i}.data = pagecontents$
  IF A page is unmodified after making C_{i-1}
    set $P_{p,i}.flag = 0$ and $P_{p,i}.data = invalid$
  IF A page is removed after making C_{i-1}
    set $P_{p,i}.flag = -1$ and $P_{p,i}.data = invalid$
  add $P_{p,i}$ into C_i

return C_i

4. 복구와 쓰레기 수집을 위한 병합 알고리즘

이 장에서는 점진적 검사점에서 복구와 쓰레기 수집을 위한 병합 알고리즘을 설명한다. 다음과 같은 점진적 검사점들이 있다면

$$C_1, C_2, \ldots, C_i, \ldots, C_k, \ldots, C_n$$

$C_n\ddot{\cup}C_{n-1}\ddot{\cup}\ldots\ddot{\cup}C_1$를 계산하여 완전 검사점 C_n'를 얻을 수 있고, 시스템 장애 시에 복구를 위해 사용할 수 있다. 또한 $C_k\ddot{\cup}C_{k-1}\ddot{\cup}\ldots\ddot{\cup}C_i$를 계산하여 C_i부터 C_k까지의 검사점들을 병합하여 하나의 검사점 C_k'를 얻을 수 있고, C_i부터 C_k까지의 검사점들을 지울 수 있다. 알고리즘2에서 병합 연산 $(C_n\ddot{\cup}C_{n-1}\ddot{\cup}\ldots\ddot{\cup}C_k)$의 과정을 보여주고 있다.

점진적 검사점들을 병합하기 위해, 알고리즘은 여러 점진적 검사점들에 흩어져 있는 모든 메모리 페이지들을 병합해야한다. 병합 알고리즘은 우선 변경 표시 정보($P_{p,i}.flag$)를 검사한다. 만일 어떤 페이지의 변경 표시 정보가 1이라면, 그 페이지의 내용은 병합 결과 검사점에 포함시킨다.

병합 알고리즘을 최적화하기 위해, *count*변수를 사용하였다. *count*변수는 필요한 모든 최근 페이지들이 결과에 포함되었는지를 판별할 수 있기 때문에 불필요하게 이전 검사점들을 모두 확인하지 않아도 된다. 각 페이지의 변경 정보가 0이 아니면 $(P_{p,i}.flag \neq 0)$ *count*변수를 증가 시킨다.

*count*변수가 점진적 검사점 C_n의 페이지 수와 같다면 병합 알고리즘은 나머지 검사점들의 페이지에 대해서는 할 필요 없이 멈출 수 있다. 이 병합 알고리즘은 알고리즘2에 기술되어 있다.

알고리즘 2. 점진적 검사점들의 병합

Input : Incremental Checkpoints(from C_n to C_k)

$C_n' = C_n$
$count := 0$

FOR $p := 1$ to $|C_n'|$
  IF $P_{p,n}'.flag \neq 0$
    $count := count + 1$

FOR $i := n-1$ to k
  FOR every page info $P_{p,i}$ in C_i
    IF $P_{p,n}'.flag = 0$
      IF $P_{p,i}.flag = 1$
        $P_{p,n}'.flag = 1$
        $P_{p,n}'.data = P_{p,i}.data$
        $count := count + 1$
      IF $count = |C_n'|$
        return C_n'

IF $count = |C_n'|$

C_n' **can be used for recovery**

return C_n'

4.1 예제

그림1은 점진적 검사점 C_5부터 C_1까지 병합하는 예를 보여준다. 여기에서 점선 원은 삭제된 페이지, 실선 원은 변경되지 않은 페이지, 어두운 원은 변경된 페이지를 나타낸다. $C_3 \ddot{\cup} C_2 \ddot{\cup} C_1$를 계산하여 C_3'를 얻고, C_3를 C_3'로 대체하면 C_1, C_2, C_3를 삭제할 수 있다.

같은 방법으로 완전 검사점 C_5'를 만들 수 있는데, 우선 $C_5 \ddot{\cup} C_4$를 계산하고, $C_5 \ddot{\cup} C_4 \ddot{\cup} C_3$를 계산한다. 여기까지만 계산하면 C_1, C_2는 계산하지 않고 병합 알고리즘을 멈출 수 있는데, $count$변수가 $|C_5| = 5$와 같기 때문이다. 병합 후 C_1부터 C_5까지의 점진적 검사점을 지울 수 있다.

그림1은 제안한 알고리즘으로 점진적 검사점을 병합하여 하나의 점진적 검사점이나 완전 검사점을 만들 수 있음을 보인다. 또한 오래된 검사점들을 쓰레기 수집하여 저장장치의 낭비를 막을 수 있다. 즉, 모든 점진적 검사점들을 유지해야 하는 비용을 줄일 수 있다.

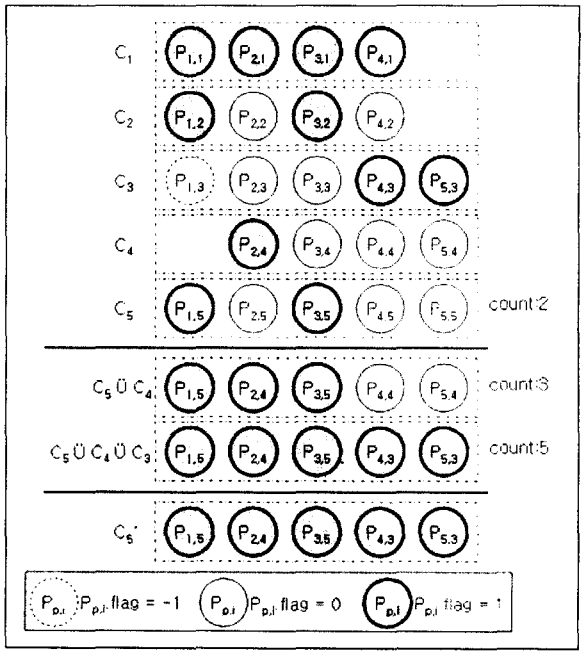

그림 1 예제

5. 결론

점진적 검사점 기법은 페이지 쓰기-보호를 사용하여 검사점들 간에 변경된 부분만 저장하는 것이다[3,5]. 그러나 점진적 검사점은 오래된 검사점들을 지울 수 없기 때문에 저장장치의 낭비를 초래할 수 있다[8]. 점진적 검사점들의 주요 목표가 검사점 오버헤드를 줄이는 것이지만, 영구 저장장치의 효율성에 있어서는 여전히 오버헤드 문제가 있다. 본 논문에서는 점진적 검사점에서 복구와 쓰레기 수집을 위한 효율적인 병합 알고리즘을 제시하였다. 제안한 알고리즘은 몇몇 점진적 검사점을 합쳐서 하나의 완전 검사점을 만들어 복구에 사용될 수 있고 오래된 검사점들을 지울 수 있다. 현재 논문에 제시한 알고리즘을 리눅스 커널 상에서 구현하고 있으며 실험을 통하여 제안한 알고리즘의 실질적 효율성에 대한 연구를 할 계획이다.

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 the International Conference on Dependable Systems and 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] James S. Plank, Kai Li and Michael Puening, Diskless Checkpointing, IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 10, pp. 303-308, October 1998

[7] James S. Plank, Yuqun Chen, Kai Li, Micah Beck and Gerry Kingsley, Memory exclusion: optimizing the performance of checkpointing systems, Software Practice and Experience, Vol. 29, No. 2, pp. 125-142, Feburary 1999

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