메모리 프로파일 검사점

허준영[O] 김상수 홍지만 조유근
서울대학교 컴퓨터 공학부
{jyheo, sskim[O], gman, cho}@ssrnet.snu.ac.kr

Memory Profiling Checkpoint

Junyoung Heo[0] Sangsu Kim Jiman Hong Yookun Cho
School of Computer Science and Engineering, Seoul National University

요 약

검사점(checkpoint) 오버헤드를 최소화하는 검사점 주기(interval)를 결정하는 것은 결함 허용 시스템(fault tolerant system)에서 검사점 알고리즘과 관련된 많은 연구들의 중요한 목표 중의 하나이다. 검사점을 드물게 하게 되면 실패 후에 재수행 하는데 너무 많은 시간이 필요하게 된다. 반면에, 너무 자주하게 되면 검사점 오버헤드가 커지고 프로그램의 총 실행시간에 영향을 주게 된다. 이 논문에서는 적응성 있는(adaptive) 시계열(time series) 분석을 사용하여 검사점 간격을 동적으로 조절할 수 있는 메모리 프로파일 검사점 간격 알고리즘을 제안한다. 트레이스에 기반한 시뮬레이션 실험 결과에서, 제안한 동적인 검사점 주기 알고리즘이 고정적인 검사점 주기 알고리즘보다 전체 검사점 오버헤드가 더 작고, 최적의 검사점 간격에 훨씬 근접했음을 알 수 있었다.

1. 서 론

검사점은 갑작스런 장애를 견딜 수 있게 하는 후진 에러 복구(backward error recovery) 기법이다. 특히 검사점이 장애 때문에 프로세스가 생성한 중간 결과들을 잃어버리지 않도록 해주기 때문에 장시간 동작하는 프로세스의 장애 감내에 사용되었다. 검사점을 함으로써, 프로세스는 최근 검사점 상태부터 실행을 재개할 수 있고, 실패에 따르는 재수행을 줄일 수 있다. 즉, 검사점은 장애가 있을 수 있는 상황에서 프로세스의 예상 실행시간을 줄여줄 수 있다.

보통 검사점 주기와 간격을 결정하는 것을 최적의 검사점 간격 문제라고 부른다. 이것의 목표는 검사점 알고리즘이 장애가 존재하는 상황에서 프로세스의 예상되는 전체 실행 시간을 최소화하는 검사점 주기와 간격을 찾게 하는 것이다. 검사점 비용이 없다면, 예상되는 프로세스의 전체 실행시간은 검사점 수에 비례하여 단조 감소할 것이다. 하지만, 대개는 검사점 오버헤드가 있고, 따라서 검사점 주기와 간격의 결정은 최소 복구 시간과 최소 실행 시간 같은 여러 문제들에 관해서 어떤 타협점과 연관이 있다[9]. 검사점을 더 자주하게 되면, 예상되는 전체 실행 시간과 검사점 오버헤드는 증가하게 될 것이다. 반면에, 너무 드물게 하게 되면, 장애가 발생했을 때, 재수행에 너무 많은 시간을 버리게 될 것이다.

최적의 검사점 일정을 정하고 검사점 오버헤드를 줄이는 것에 대한 연구가 많이 이루어져 왔다. 고려할만한 이론적 연구 결과 [3, 5, 7, 9]들은 검사점 오버헤드가 낮은 경우에 검사점 간격을 분석하고 결정하는 것이다. [2]에서 Brock은 검사점이 있을 때와 없을 때에 예상 실행 시간을 분석하고 추정하였다. [3]에서 Duda는 고정 크기의 프로세스 실행이 끝날 때까지 걸린 시간을 최소화할 수 있는 검사점 간격의 일정을 고려하였다. Duda는 검사점을 하는 중에 발생하는 장애의 경우 이전 검사점으로 되돌아가지 않고 실행 중이던 검사점을 다시 시작하기만 한다고 가정하였다. [5]에서 L'Ecuyer는 장애 빈도가 일정하지 않을 때, 동적 검사점 간격 알고리즘의 가용성에 대해 수학적 접근을 유도하였다. [7]에서 Toueg와 Babaoglu는 소수(小數)의 검사점 장소가 있을 때, 오프라인에서 검사점 간격을 계산하는 최적 알고리즘을 유도하였다. [9]에서 Ziv는 마코프 체인 모델을 사용하여 검사점이 메모리상태와 가장 유익한 프로그램의 부분을 찾았다. 그러나 Ziv는 메모리 상태가 두 개 뿐이고 메모리 상태의 천이가 마코프 체인 모델에 의해 결정된다고 가정하였다. 그러므로 Ziv의 알고리즘은 일반적이고 실질적인 메모리 사용 모델에 적용될 수 없다.

그러나 이전의 작업들은 메모리 사용량이 검사점 비용과 관련 있음을 고려하지 않았다. 검사점 오버헤드를 증가시키는 주 원인들은 장애 빈도, 검사점 비용과 장애 발생시 복구비용들이다. 대게 시스템의 장애 빈도는 미리 알 수 있고, 장애 비용과 복구비용은 프로세스의 메모리 사용량과 비례한다. 검사점과 복구비용의 가장 큰 부분은 각각 프로세스의 메모리 내용을 안전한 저장장치에 쓰는 시간과 메모리로 올리는 시간이다.

메모리의 프로세스 상태가 메모리 할당과 해제로 인해 실행 중 동적으로 변하기 때문에 메모리 사용량을 미리 아는 것은 어렵다. 그러나 프로세스의 메모리 사용은 반복 순환과 특별한 함수 호출에 의해 특별한 패턴을 이룬다.

이 논문에서, 우리는 검사점의 유무에 따른 프로세스의 예상 실행 시간의 식을 유도하고 검사점 간격에 결정적인 요소를 보일 것이다. 또한, 적응성 있는 시계열을 사용하는 효율적인 검사점 알고리즘을 제시할 것이다. 이 알고리즘은 프로세스의 메모리 사용량을 추적하여 앞으로의 메모리 사용을 예측한다. 그리고 이 알고리즘은 예상 실행 시간과 검사점 오버헤드를 고려하여 검사점 간격을 동적으로 정한다.

이 논문은 다음과 같이 구성된다. 2장에서 검사점 오버헤드와 간격간의 관계를 보이고, 3장에서는 적응성 있는

시계열 분석을 기반으로 메모리 프로파일 검사점 간격 알고리즘을 제안한다. 4장에서 성능 평가 결과를, 5장에서 결론을 맺는다.

2. 검사점 오버헤드와 검사점 간격

프로세스의 검사점을 작성하기 위해서, 검사점을 만드는 프로세스는 주기적으로 프로세스의 실행을 중지시키고 안정한 저장장치(예: 디스크)에 프로세스의 상태를 저장한다. 보통 검사점을 만드는 프로세스는 검사점을 만들고 이를 디스크에 저장하기 위해서 부가적인 실행 시간을 필요로 한다. 그리고 이러한 실행 시간은 유용한 실행에 포함되지 않는다. 그러므로 예상 실행시간과 검사점 오버헤드는 일반적으로 검사점 요구에 비례하여 증가하게 된다.

검사점의 효율을 측정하기 위한 좋은 단위로 검사점 오버헤드 비율 R을 사용한다.

$$R = \frac{\overline{T} - T}{T}$$

여기서 $\overline{T}$는 예상 전체 실행 시간이고 T는 프로세스가 실행한 시간을 나타낸다.

본 연구의 목표는 검사점 오버헤드 비율 R을 어떻게 최소화할 것인가이다. 검사점을 만들 수 있는 장소의 수가 유한하고, 각 장소마다의 검사점 비용을 미리 알 수 있을 때, 최적의 검사점 간격을 알 수 있다. 오프라인 검사점 간격 방법에서는, 검사점 오버헤드를 최소화 하는 최적의 검사점 간격 t가 대략 $\sqrt{\frac{2\bar{c}}{\lambda}}$ [3,9] 이다. 여기서 $\bar{c}$는 전체 프로세스의 평균 검사점 비용이다.

2.1 검사점 오버헤드를 최소화하는 검사점 간격

검사점 오버헤드를 증가시키는 주 원인은 결함 발생 빈도, 검사점 비용과 실행 중 결함 발생 시 복구비용이다. 이 장에서는 적당한 t, c 값을 정함으로써 검사점 간격을 정하여 프로세스의 예상 실행 시간을 최소화 하는 것이다. 검사점 비용의 대부분은 메모리 내용, 열린 파일 정보, 네트워크 I/O 상태를 안정한 저장장치에 쓰는데 필요하다.

보통 시스템의 결함 발생 빈도는 미리 알 수 있고, 예상 실행시간에 영향을 주는 검사점과 복구비용은 검사점 오버헤드에 비례한다고 할 수 있다. 검사점 비용의 대부분은 안정한 저장장치에 쓰기 위한 시간이고, 복구의 대부분은 검사점을 읽어서 주 메모리에 올려 재수행을 하는 시간이다.

프로세스의 메모리 사용량은 메모리 블록의 할당과 해제로 인해 동적으로 바뀌기 때문에, 미리 예측하는 것은 힘들다. 그러나 만약 프로세스의 메모리 사용을 정량화하고 적응성 있는 시계열 데이터로서 시간 t와 $t+1$에서의 양을 사용할 수 있다면, 메모리 사용량을 예측하여 검사점 비용을 미리 추정할 수 있고 검사점 오버헤드를 최소화하는 적당한 곳에 검사점을 함으로써 예상 실행 시간을 최적화 할 수 있다.

3. 메모리 프로파일 검사점 간격 알고리즘

메모리 사용 모델에서, 프로세스의 메모리 사용이 시계열 데이터의 일종으로 정의 가능하기 때문에, 미래의 사용을 예측하기 위한 다양한 시계열 분석 기법을 사용할 수 있다. 그래서 선형이고 비정상(非定常, non stationary) 특성을 갖는 메모리 사용을 예측하기 위해 적응성 있는 시계열 모델을 선택하였다. 특히 ARIMA 모델은 이론적으로 비정상적인 시계열 데이터를 예측하기 위한 가장 일반적인 모델로서, 과거의 값, 과거의 오류 그리고 다른 시계열 데이터의 현재와 과거의 값들의 일차 결합으로 결과 값을 예측할 수 있다. ARIMA 과정은 일변량(univariate) 시계열 모델 확인, 인자 추정, 그리고 예측을 위한 종합적인 툴을 제공한다. 일반적으로, ARIMA 모델에서 시간 t에서부터 시간 l에 대한 식은 다음과 같다.

$$Y(t+l) = \sum_{j=0}^{\infty} \psi_j a_{t+l-j}$$

여기에서 a_t는 입력으로써 백색 소음이고, ψ는 가중치 인자이다.

가장 좋은 메모리 사용 예측을 계산하기 위해, 정확도를 정하는 것이 중요하다. 예측의 정확도는 각 예측의 어느 한 쪽에서 확률 극한을 계산하여 나타낼 수 나타낼 수 있다. ARIMA 모델에서 경과 시간 l에서 예측 오류는 다음과 같이 주어진다.

$$e_t(l) = a_{t+1} + \psi_1 a_{t+l-1} + ... + \psi_{l-1} a_{t+1}$$

$E_t[e_t(l)] = 0$이면, 예측은 공정하다고 할 수 있다 [1]. 또한 예측 오류의 분산은 다음과 같이 주어진다.

$$V(l) = var[e_t(l)] = (1 + \psi_1^2 + \psi_2^2 ... + \psi_{l-1}^2)\sigma_a^2$$

오류 e_t가 정규 분포를 따른다고 가정하면, 예측 오류 $e_t(l)$은 평균 0인 정규분포이고, 분산 $e^2(l)$은 $\sigma_e^2 \sum_{j=0}^{l-1} \psi_j^2$이다. 그리고 $1 \le t \le n$이라고 가정하면 Y_{n+1}의 조건부분포는 평균 $Y_n(l)$이고 분산 $\sigma^2(l)$인 정규분포를 따른다.

$$P(-y_{\frac{a}{2}} \le \frac{Y_{n+l} - Y_n(l)}{\sigma(l)} \le y_{\frac{a}{2}}) = 1 - a$$

이므로, Y_{n+l}에 대한 확률 극한 $(1-a)$는 $Y_n(l) \pm y_{\frac{a}{2}} \sigma(l)$로 주어진다.

따라서 시간 t에서 메모리 사용량을 예측 할 수 있고, 예측 결과는 확률 극한 범위 내에서 이전 메모리 사용과 오류치 $e(t)$에 종속적이다. 이 극한은 계수(係數) a_t에 영향을 받는다.

제안한 시계열 분석을 사용하는 알고리즘은 다음과 같이 동작한다. 메모리 사용 증가가 감지될 때 검사점을 작성하고, 그렇지 않으면 최대 대기 시간 전까지 검사점의 작성을 미룬다. 최대 검사점 작성 대기 시간은 복구 시간을 최소화하는 검사점 오버헤드 비율로 정해 질 수 있다. 이 경우, 검사점과 대기 시간($\sqrt{\frac{2\bar{c}}{\lambda}}$),이 증가하기 직전에 적은 비용으

로 검사점을 작성하는 것이 검사점의 오버헤드를 줄일 수 있다. 그리고 검사점 오버헤드는 적응성 있는 시계열 분석과 메모리 사용 추이에 의해 추정된 검사점 오버헤드로 재계산된다. 현재 검사점 오버헤드 비율이 재계산된 오버헤드 비율과 예측된 오버헤드 비율보다 작다면 검사점 알고리즘은 즉시 검사점을 실행한다. 쿨 경우에는 검사점 타임아웃을 예측한 검사점 오버헤드 비율로 맞춘다. 검사점 알고리즘은 주어진 기간 내에 만약 프로세스의 메모리 사용량이 실행 중에 급격하게 변하는 것을 예측하여 검사점을 작성한다.

4. 성능 평가

이 장에서는, 제안한 검사점 알고리즘의 성능 평가를 설명한다. 제안한 알고리즘의 성능을 평가하기 위해, 우선 리눅스 계산(computation) 서버에서 X 폰트 서버와 X 서버의 메모리 사용을 추적한 프로파일의 일부에서 X 폰트 서버와 X 서버의 메모리 사용 프로파일을 각각 추출하였다.

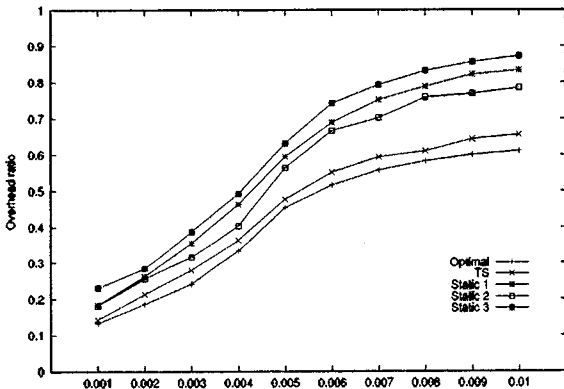

그림 1. X 폰트 서버에서 검사점 오버헤드 비율

그림 1은 장애 비율이 0.001부터 0.010까지 변할 때 평균 검사점 오버헤드 비율 R을 3 종류의 정적인 검사점 주기와 제안한 동적 검사점 주기 알고리즘에 대해 나타낸 것이다. 4가지 모두 X 폰트 서버의 메모리 사용 프로파일에서 최적 검사점 주기가 사용될 때 검사점 오버헤드 비율을 사용하였다.

최적의 오프라인 검사점을 위해, 먼저 X 폰트 서버의 메모리 사용 프로파일을 검사하고, n개의 태스크 실행(n task computation)에 대한 최적 검사점 주기 계산하기 위해 Toueg의 오프라인 동적 프로그래밍 알고리즘을 사용한다 [7]. 정적으로 검사점 주기를 결정하는 경우, 여러 검사점의 주기에 대해 검사점 오버헤드 비율을 측정하였다. 검사점 오버헤드 비율의 공정한 비교를 위해 최적이고 동적인 검사점 주기와 같은 검사점 수를 정적 주기에 대해 사용하였다.

그림 1은 제안한 동적으로 검사점 주기를 조정하는 검사점 알고리즘을 보여준다. 결함 발생 비율과 상관없이 정적 검사점 주기 알고리즘에 비해 훨씬 낮은 검사점 오버헤드 비율을 보이고, 제안한 알고리즘의 검사점 오버헤드가 최적의 오프라인 검사점 알고리즘에 더 근접함을 알 수 있다. 그림 1에서 Static 3의 검사점 오버헤드 비율이 Static 1이나 Static 2보다 높음을 보이는데, 이를 통해 검사점 오버헤드 비율이 검사점 수뿐만 아니라 복구비용과도 관련이 있음을 알 수 있다. Static 3의 경우, 복구비용은 Static 1이나 Static 2보다 훨씬 높다. 안정적이고 일률적인 메모리 사용 형태를 갖는 장시간 동작하는 프로세스에 제안한 알고리즘을 적용한다면, 더 나은 성능을 얻으리라고 확신한다.

5. 결론

이 논문에서, 우리는 프로세스의 메모리 사용량을 예측하기 위해 적응성 있는 시계열 분석을 통한 동적으로 검사점 주기를 결정하는 알고리즘을 제안하였다. 제안한 알고리즘은 검사점 오버헤드 비율과 메모리 사용의 통계적 특성을 사용하여 예측한 메모리 사용을 통해 검사점의 주기를 동적으로 결정한다. 한 프로세스의 메모리 사용은 동적으로 변하기 때문에, 메모리 사용을 미리 예측하는 것은 어려운 일이다. 그러나 실제적으로 장시간 실행되는 응용 프로그램의 경우 실행 중 메모리 사용은 특정 형태를 따르고, 이 것은 메모리 사용의 예측에 사용될 수 있다.

참고문헌

[1] George E. P. Box, Gwilym M. Jenkins and Gregory C. Reinsel, Time Series Analysis, Forecasting and control, Prentice Hall, 1999

[2] A. Brock, "An Analysis of Checkpointing", ICL Technical Journal, Vol. 1, 1979

[3] A. Duda, "The E_ects of Checkpointing on Program Execution Time", Information Processing Letters, Vol. 16, pp. 221-229, July 1983

[4] Jiman Hong, Taesoon Park, H.Y Yeom and Yookun Cho, "Kckpt : An E_cient Checkpoint Facility on UnixWare", 15th International Conference on Computers and Their Applications, 2000

[5] P. L'Ecuyer and J. Malenfant, "Computing Optimal checkpointing for Rollback and Recovery Systems", IEEE Transactions on Computer, Vol. 37, No. 4, pp. 491-496, Apr. 1988

[6] James S. Plank, M. Beck and G. Kingsley, and K. Li, "Libckpt:Transparent Checkpointing under Unix", Proc. Usenix Winter 1pp5 Technical Conference, 1995.

[7] Sam Toueg and Ozalp Babaoglu, "On the Optimum Checkpoint Selection Problem", SIAM Journal Computer, Vol. 13, No. 3, August 1984

[8] N.H. Vaidya, "Impact of Checkpoint latency on Overhead Ratio of a Checkpointing Scheme", IEEE Transactions on Computers, vol. 46, no. 8, 1997

[9] Avi Ziv and Jehoshua Bruck, "An On-Line Algorithm for Checkpoint Placement", IEEE Transactions on Computers, vol .46, no. 9, 1997