UnixWare 커널 수준의 효율적인 검사점 및 복구 도구

홍지만° 한상철 윤진혁 박태순† 염헌영 조유근
서울대학교 컴퓨터공학과 † 세종대학교 정보과학과
{gman, schan, jhyoon@ssrnet.snu.ac.kr}
tspark@cs.sejong.ac.kr, yeom@arirang.snu.ac.kr, cho@comp.snu.ac.kr

An Efficient Checkpoint and Recovery Facility on UnixWare Kernel

Jiman Hong° Sangchul Han Jinhyuk Yoon
Taesoon Park† Heon Y. Yeom Yookun cho
Dept. of Computer Engineering, Seoul National University
† Dept. of Information Science, Sejong University

요 약

검사점 및 복구 도구는 사용자 응용 프로그램의 상태를 주기적으로 안정된 저장소에 저장을 하고, 결함이 발생하였을 경우 가장 최근의 검사점으로부터 효율적으로 복구하게 하는 도구이다. 특히 검사점 및 복구 도구는 장시간 수행되는 프로세스에게는 아주 중요한 의미를 지니며, 결함으로 인해 장시간 수행되는 프로세스에 의해 생성된 중간 결과를 잃어버리지 않게 한다.

본 논문에서는 UnixWare 커널 수준의 검사점 및 복구 도구인 Kckpt의 설계 및 구현 내용을 제시하고, Kckpt의 성능을 사용자 수준에서 구현한 검사점 라이브러리와 비교한 결과를 제시한다. Kckpt를 사용함으로 해서 UnixWare는 소스 코드 수준에서 사용자가가 초기화한 장소에서 검사점을 만들 수 있을 뿐만 아니라 실행 가능한 파일만을 가진 사용자에게도 완전한 투명성을 제공할 수 있다. 또한 Kckpt를 사용자 라리브러리 수준에서 구현한 검사점 도구와 성능을 비교한 결과 오버헤드가 훨씬 줄어들었음을 관찰할 수 있었다.

1. 서론

후방 에러 복구 기법으로 알려진 검사점 및 복구 기법은 예상하지 못한 결함을 허용할 수 있는 메카니즘으로 소프트웨어 결함 허용 기법에서 아주 중요한 의미를 갖는다. 검사점 및 복구 도구[1,5]는 시스템의 프로세스들이 결함이 없는 상태로 수행 중일 때 각 프로세스의 상태를 주기적으로 안전한 저장소(디스크)에 저장하여 시스템에 결함이 발생하였을 때 저장된 상태로부터 시스템이 다시 시작할 수 있게 한다.

단일 프로세서 시스템 환경에서 검사점 및 복구 도구는 운영체제의 커널 수준에서 구현할 수도 있으며, 사용자 라이브러리 수준에서 구현[1,3]할 수도 있다. 사용자 라이브러리 수준에서 구현하는 검사점 및 복구 도구는 사용자 프로그램과 라이브러리를 링크시킴으로 응용 프로그램의 상태를 저장할 수 있게 한다. 이 기법은 사용자나 프로그래머에게 많은 융통성을 제공하는 반면 라이브러리라는 특성으로 인해 다음과 같은 여러 가지 제약점이 있다.

먼저, 사용자 수준의 검사점 라이브러리는 커널 내의 모든 주소 공간을 읽고 쓸 수 있는 권한을 가질 수가 없기 때문에 접근할 수 있는 프로세스와 관련된 모든 정보를 검사점으로 만들지 못한다는 것이다. 즉 응용 프로그램은 운영체제 관리하는 메모리 영역에서 매우 제한된 접근 권한을 가지며 복잡한 작업을 통해 상태 정보를 얻어야 한다. 현존하는 대부분의 검사점 라이브러리들은 *popen()*이나 시그널을 제대로 지원하지 않을 뿐만 아니라[3] 프로세스의 데이터 세그먼트의 주소 공간의 크기를 변경시키는 등의 시스템 콜도 제대로 지원하지 못한다.

두 번째로, 사용자 수준의 검사점 라이브러리는 사용자 수준의 라이브러와 링크시키고 재컴파일 하기 위해 응용 프로그램의 소스코드를 필요로 한다는 것이다. 응용 프로그램의 소스 코드를 구하는 일은 어려울 뿐만 아니라, 재컴파일 과정은 상당히 시간을 요하며, 어떤 경우에는 특정한 컴파일 환경을 필요로 하는 경우가 많다[5].

본 논문에서 제시하는 커널 수준의 검사점 및 복구 도구는 *ckpt()*과 *recover()*라는 시스템 호출을 UnixWare 시스템에 추가하여 구현하였다. 커널 수준에서 구현한 검사점 및 복구 도구를 사용함으로 해서 UnixWare는 소스 코드 수준에서 사용자가 직접 명시한 지점에 검사점을 만드는 사용자 초기화 검사점을 제공할 뿐만 아니라, 응용 프로그램의 사용자에게 완전히 투명한 검사점(totally user-transparent checkpoint)을 제공한다. 기존에 이미 구현된 사용자 수준의 검사점 및 복구 도구인 Libckpt[2]는 사용자에게 투명한 검사점을 제공하기 위해 응용 프로그램의 *main()*을 *ckpt_target()*으로 변경시켜 검사점 라이브러리와 링크를 시켜야 하는 반면에 본 논문에서 제시하는 커널 수준의 검사점 및 복구 도구는 소스 코드의 아무런 변경이 불필요할 뿐만 아니라 소스 코드 자체가 필요 없는 완전한 사용자 투명성을 제공한다.

본 논문의 구성은 다음과 같다. 먼저 2절에서 이전에 수행되었던 관련 연구들을 살펴보고 3절에서 UnixWare 커널 수준의 검사점 및 복구 도구의 구현 방법과 구현 상의 특징에 대해 설명을 한다. 4절에서는 기존의 사용자 라이브러리 수준에서 구현한 Libckpt와의 검사점 오버헤드를 비교 분석한다. 그리고 마지막으로 5절에서 결론을 맺는다.

2. 관련 연구

유닉스와 윈도우 NT 운영체제 기반 하에서 여러 검사점 및 복구 메카니즘이 설계되고 구현되었다. 대부분의 검사점 및 복구 도구는 사용자 수준에서 구현되었으며, 본 절에서는 이것들에 대해 간단하게 설명한다.

- Libckpt : Libckpt[2]는 forked 검사점, 증가(incremental) 검사점과 사용자에게 투명한 검사점을 제공한다. 그러나 이 도구는 검사점을 위한 사용자 수준의 라이브러리와 검사점을 만들 응용 프로그램을 항상 링크 시켜 재컴파일 해야 하기 때문에 사용자에게 완전한 투명성을 제공하지 못한다. 또한 검사점을 위한 라이브러리가 커널의 모든 주소 공간을 제어할 능력이 없기 때문에 일부 시스템 호출과 시그널이 포함된 사용자 응용 프로그램의 검사점을 만들지 못한다. 그리고 사용자 라이브러리 수준에서 구현을 하였기 때문에 이식성이 높을 것으로 생각되지만 운영체제 시스템 마다 메모리를 제어하는 방법이 다르고 가상 주소 메카니즘이 다르기 때문에 이식도 쉽지 않다.
- CosMic : CosMic은 Libck, Libfcp, Libft 그리고 Libst와 같은 네 개의 검사점 라이브러리로 구성된 결함 허용 도구이다[1]. Libck는 투명한 검사점 라이브러리이며, Libfcp는 파일 검사점 라이브러리이며, Libft는 주요 데이터 검사점 라이브러리이며, Libst는 강한 형태의 검사점 라이브러리이다. CosMic은 사용자 수준의 프로세스 이주에 중점을 두어 구현되었으며, 일반적인 사용자 응용의 검사점을 위해 간단한 해결책은 제시하지 못하고 있다.
- A Transparent Checkpoint Facility on NT : 이것은 윈도우 NT상에서 구현된 최초의 검사점 도구[3]로 사용자에게 투명한 검사점을 제공한다. 이 도구는 NT 시스템 API 호출과 라이브러리 API 호출을 조화시켜 구현하였다.

3. UnixWare 커널 수준의 검사점 및 복구 도구 (Kckpt)

본 절에서는 프로세스의 검사점을 만들고 결함이 발생 시에 이를 복구하는 방법에 대해 설명한다. 프로세스의 검사점을 만들기 위해 우리는 UnixWare에 *ckpt()*라는 검사점 시스템 호출을 추가하였다. *ckpt()*가 호출되면, 주어진 화일 이름에 검사점 화일임을 나타내는 확장자 '*_ckpt*'를 추가된 검사점 파일을 만든다.

검사점 파일은 한 프로세스가 사용하는 세그먼트의 수와 오픈한 파일의 개수를 기록하는 검사점 헤더, 프로세스의 상태 정보(status, time, signal 등)를 저장하는 *proc* 구조체와 *lwp* 구조체, 사용자 모드의 범용 레지스터 정보를 저장하는 레지스터 정보, 사용 중인 가상주소 공간 정보 및 메모리 이미지를 저장하는 세그먼트와 데이터 세그먼트 그리고 프로세스가 사용 중인 화일의 정보 (flag, offset, file id 등)를 저장하는 오픈한 파일 정보로 구성되어 있다. 검사점 화일이 만들어지면, 그림 1과 같은 과정으로 프로세스의 검사점을 만든다.

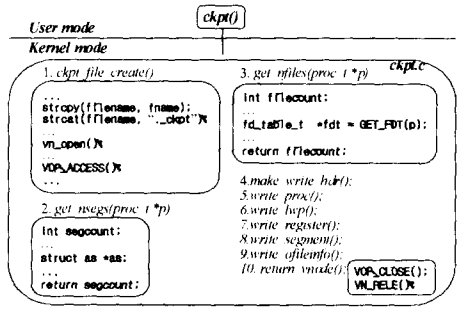

그림 1. 검사점 시스템 호출 : *ckpt()*

결함이 발생한 프로세스의 복구를 위해 UnixWare에 *recover()* 시스템 호출을 추가하였다. 이 시스템 호출의 인자로는 검사점 화일의 이름이 들어가며 이 화일 이름을 이용해 검사점 화일을 찾아서 연다. 다음으로 *spawn_proc()* 함수를 호출하여 새로운 프로세스를 생성한다. 새로운 프로세스를 생성하는 과정은 *fork()* 시스템 호출과 유사하다. 다만 그림 2에서 보여주듯이 자식 프로세스의 커널 내 시작 루틴은 *restore_ckpt()*이라는 것이 *fork()* 시스템 호출과 다르다.

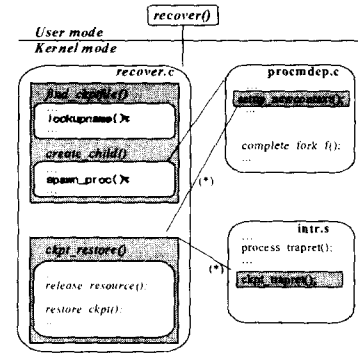

그림 2. 복구 시스템 호출 : *recover()*

복구 루틴을 수행할 자식 프로세스가 생성된 후 자식 프로세스는 *ckpt_restore()* 함수에서 수행이 시작되고(그림 2에서 * 부분) 실제적인 복구 작업이 이루어진다. *ckpt_restore()*는 먼저 *lwp*와 프로세스간 통신을 위한 공유 메모리를 해제하고 검사점 화일의 내용을 읽어 들인다. 거기에 따라 *proc* 구조체, *lwp* 구조체, register, 세그먼트, 실제 데이터, 열린 화일의 순으로 복구가 이루어진다.

3.1 forked 검사점

단일 처리기 응용의 검사점을 얻는 방법은 크게 연속 검사점(sequential checkpoint)과 forked 검사점으로 나눌 수 있다[4]. 연속 검사점은 프로세스가 검사점을 만들기를 원할 때 자신의 실행을 잠시 멈추고, 자신의 상태를 안전한 저장소에 저장한다. 연속 검사점의 오버헤드는 검사점 파일을 디스크에 쓰는 시간과 같다고 할 수 있다[4]. 이러한 검사점의 오버헤드를 줄이기 위해서 Kckpt는 forked 검사점 기능을 제공한다.

forked 검사점을 위해서 *proc* 구조체에 forked 검사점을 위한 플래그를 새로 추가하였다. 검사점을 만들어야 하는 프로세스가 스케쥴링 될 때 *proc* 구조체의 플래그를 확인하고 *fckpt()* 커널 내부 함수를 부르게 된다. 이 함수는 자식 프로세스를 생성하고 자식 프로세스가 실행할 함수를 *fckpt_child()*로 설정한 뒤 바로 리턴하여 부모 프로세스는 계속 자신의 작업을 할 수 있게 해준다. *fckpt_child()*는 자식 프로세스의 문맥에서 실행되며, 검사점 작업을 수행한다.

3.2 사용자에게 완전히 투명한 검사점

지금까지 설명한 검사점을 만들기 위해서는 검사점을 만들기를 원하는 프로세스가 스스로 시스템 호출을 호출해야 한다. 즉 사용자가 소스 코드의 원하는 지점에 *ckpt()* 시스템 호출을 호출해야 한다.

사용자에게 투명한 검사점이란 프로그램 자신이 스스로 검

사점을 만드는 것을 의미한다. 일반적으로 투명한 검사점은 특별한 검사점 라이브러리와 응용 프로그램을 링크 시켜 컴파일함으로써 구현될 수 있다. 그러나 Kckpt는 커널 수준에서 구현한 검사점 도구이기 때문에 새롭게 응용 프로그램을 검사점을 위한 라이브러리와 링크시켜 재컴파일 하지 않고 사용자에게 투명성을 제공할 수 있다.

사용자에게 완전히 투명한 검사점을 만들기 위해서 우리가 구현한 검사점 도구는 프로세스 번호를 이용하여 검사점을 만들 대상 프로세스를 검색한 후, 대상 프로세스의 프로세스 테이블 엔트리에 검사점 작성 대상이라는 정보와 실행 화일의 이름을 기록하는 역할을 할 수 있는 *tckpt()*라는 새로운 시스템 호출을 UnixWare에 추가하였다. *tckpt()* 시스템 호출의 인자는 검사점을 만들어야 하는 프로세스의 프로세스 번호와 그 프로세스의 실행 화일의 이름이 된다.

어떤 한 프로세스가 *tckpt()* 시스템 호출을 호출하면 인자로 받은 프로세스 번호에 해당하는 프로세스를 찾고, 그 프로세스가 실행될 때 검사점을 만들 프로세스의 *proc* 구조체에 검사점을 만들어야 한다는 것을 나타내는 플래그를 설정하고 실행 화일의 이름을 기록한 뒤 *tckpt()* 시스템 호출은 종료된다. *proc* 구조체 내에 검사점을 만들어야 한다는 플래그 설정을 위해 *proc* 구조체 내에 새로운 플래그 필드를 추가하였다. 만약 검사점 만들기 플래그가 설정된 프로세스가 스케쥴링 되어 실행되기 전에 프로세스 자신의 검사점을 만들게 된다. 즉, 한 번 *tckpt()* 시스템 호출을 호출할 때마다 검사점을 만들기를 원하는 프로세스의 검사점이 만들어지게 된다.

4. 실험 및 결과

검사점의 오버헤드를 비교하기 위해 행렬 곱셈 응용과 셀룰라 오토마타 응용을 각각 10번의 검사점이 포함된 수행 시간과 검사점 없이 수행한 시간을 측정하였다. 또한 검사점 도구의 성능을 비교하기 위해 사용자 수준의 검사점 라이브러리인 Lickpt를 UnixWare에 이식을 하여 Kckpt와 같은 환경에서 같은 응용을 실험하였다. 실험은 펜티엄 166Mhz PC 상에 수행 중인 UnixWare 2.1에서 하였다.

행렬 곱셈 응용 실험에서는 500x500에서 1500x1500까지 행렬의 크기를 변화시켜가며 실험을 하였으며, 셀룰라 오토마타 응용의 실험에서는 셀 크기를 1024에서 2048까지 그리고 세대 크기를 15에서 30까지 변화시켜 가며 실험을 하였다. 실험한 결과는 표 1에 나타나 있다.

Kckpt가 UnixWare 커널 상에서 구현되었으며 모든 커널 주소 공간을 쉽게 접근할 수 있기 때문에 모든 응용에서 사용자 라이브러리 수준에서 구현한 검사점 도구인 Libckpt보다 훨씬 좋은 성능을 보였다. 또한 Kckpt의 forked 검사점의 오버헤드는 Libckpt의 forked 검사점의 오버헤드와는 달리 거의 무시할 수준임을 실험을 통해 알 수 있었다.

5. 결론

검사점 및 복구 도구는 장시간 수행되는 프로그램에 결함 허용을 제공하는 아주 중요한 도구이다. 본 논문에서는 UnixWare 커널 수준에서 구현한 검사점 및 복구 도구의 구현 내용과 성능에 대해 설명하였다.

검사점과 복구를 위해 UnixWare 커널에 새로운 시스템 호출 *ckpt()*, *recover()* 그리고 *tckpt()*를 추가하였다. 또한 사용자에게 완전히 투명한 검사점을 위해 *proc* 구조체에 새로운 플래그를 추가하였다.

사용자에게 완전히 투명한 검사점은 소스 코드의 변경이 전혀 필요 없을 뿐만 아니라 바이너리 실행 파일만을 가진 사용자에게도 응용 프로그램의 검사점을 만들 수 있게 한다. 또한 다양한 응용 프로그램의 실행을 통해 커널 수준에서 구현한 검사점 및 복구 도구는 사용자 라이브러리 수준에서 구현한 검사점 및 복구 도구 보다 더 좋은 성능을 보임을 알 수 있었다.

facility	app	size	method	running time (sec)	overhead (sec)	overhead (%)
No ckpt	mat	500×500		57	0	0.00
		615×615		111	0	0.00
		1000×1000		544	0	0.00
		1500×1500		1868	0	0.00
No ckpt	cellular	1024(15)		13	0	0.00
		1024(30)		26	0	0.00
		2048(15)		56	0	0.00
		2048(30)		115	0	0.00
Libckpt	mat	500×500	sequential	93	36	63.16
		500×500	forked	93	36	63.16
		615×615	sequential	195	84	75.68
		615×615	forked	194	83	74.77
		1000×1000	sequential	825	281	51.65
		1000×1000	forked	823	279	51.29
		1500×1500	sequential	2923	1057	56.58
		1500×1500	forked	2922	1054	56.42
	cellular	1024(15)	sequential	22	9	69.23
		1024(15)	forked	21	8	61.54
		1024(30)	sequential	46	18	64.29
		1024(30)	forked	43	15	53.57
		2048(15)	sequential	87	31	55.36
		2048(15)	forked	82	26	46.43
		2048(30)	sequential	172	57	49.57
		2048(30)	forked	165	50	43.48
Kckpt	mat	500×500	sequential	62	5	38.46
		500×500	forked	58	1	7.69
		615×615	sequential	118	7	25.00
		615×615	forked	112	1	3.57
		1000×1000	sequential	556	14	25.00
		1000×1000	forked	545	3	1.79
		1500×1500	sequential	1905	37	37.17
		1500×1500	forked	1870	2	1.14
	cellular	1024(15)	sequential	17	4	30.77
		1024(15)	forked	14	1	7.69
		1024(30)	sequential	39	11	39.29
		1024(30)	forked	29	1	3.57
		2048(15)	sequential	70	14	25.00
		2048(15)	forked	58	2	3.57
		2048(30)	sequential	144	29	25.22
		2048(30)	forked	120	5	4.35

표 1. 실험 결과

6. 참고 문헌

[1] P. E. Chung, Y. Huang, S. Yajnik, G. Fowler, K. P. Vo, and Y. M. Wang, "Checkpointing in CosMiC: a User-level Process Migration Environment," Int. Symp. on Fault-Tolerant Systems, 1997.

[2] James S. Plank, Micah Beck, Gerry Kingsley and Kai Li, "Libckpt: Transparent Checkpointing under UNIX," Usenix Winter 1995 Technical Conference, 1995.

[3] Johny Srouji, Paul Schuster, Maury Bach and Yulik Kuzmin, "A Transparent Checkpoint Facility On NT," 2nd USENIX Windows NT Symposium, 1998.

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

[5] Y. M. Wang, Y. Huang, K. P. Vo, P. E. Chung and C. Kintala, "Checkpointing and its applications," IEEE Fault-Tolerant Computing Symposium, 1995.