임베디드 시스템을 위한 복합 결함 허용 기법

국중진° 홍지만
숭실대학교 컴퓨터학과
tipsiness@gmail.com°, jiman@ssu.ac.kr

The Hybrid Fault Tolerant Technique for Embedded System

Joongjin Kook° Jiman Hong
Soongsil University

요 약

검사점 및 복구 도구(Checkpointing & Recovery Facility)를 이용하여 임베디드 시스템에서 결함 허용(Fault Tolerance) 기법을 적용할 경우 쓰기 작업의 오버헤드로 인해 실용성이 크게 떨어지게 된다. 실시간 운영체제와 함께 어떠한 한계 상황에서 결함 허용 및 복구 도구가 오히려 시스템의 성능을 저하시키는 요인으로 작용하게 되면 이는 결국 쓸모없는 도구가 되어 사용되지 않을 것이다. 따라서 프로세스의 복구를 위해 저장하는 프로세스 이미지의 기록에 소요되는 시간을 크게 낮추어야만 비로소 검사점 도구가 그 진가를 발휘하게 될 수 있다.

본 논문에서는 NVSRAM(Non Volatile SRAM)을 검사점 및 복구 도구의 저장 장치로 활용함으로써 기존의 검사점 도구에서 성능을 저하시키는 주원인이었던 검사점 기록의 오버헤드를 개선하기 위한 연구를 수행하였다. 검사점 기록 시간을 줄이기 위한 방법으로 주 메모리에 저장된 프로세스의 복구와 관련된 데이터를 SRAM 특성을 갖는 비휘발성 저장 장치인 NVSRAM에 저장하여 디스크 접근에 소요되는 시간을 최소화시킴으로써 임베디드 시스템에서 실용적으로 사용 가능한 검사점 도구를 구현하였고, 이러한 연구의 결과를 검증하기 위해 기존 시스템에서 저장 장치로 사용되던 플래시 메모리, 주 메모리, 원격 메모리를 사용하는 경우의 성능과 NVSRAM을 활용할 때의 성능을 비교해 보았다.

본 연구에서 제안하는 결함 허용 도구는 실제 시스템에 적용하여 효과적인 성능을 발휘할 수 있을 것이며, 차세대 메모리를 이용한 결함 허용 도구의 연구에 기여를 할 수 있을 것으로 기대된다.

1. 서 론

상당수의 임베디드 시스템들은 열악한 환경에서 사람을 대신하여 업무를 진행하게 되며, 이러한 업무들은 오류 또는 결함 발생 등의 이유로 정상적인 수행이 이루어지지 않을 경우 생명이나 재산상의 피해를 가져올 수 있다. 따라서 임베디드 시스템에는 오류를 극복하기 위한 하드웨어 또는 소프트웨어적 장치가 필요하다.

범용 컴퓨터, 분산 컴퓨팅 환경, 병렬 컴퓨팅 환경에서는 시스템의 하드웨어적인 결함이나 소프트웨어적인 결함으로 인한 데이터의 유실을 방지하기 위해 하드웨어적인 결함 허용 기법과 소프트웨어적인 결함 허용 기법이 연구되어 왔다. 랩탑(laptop) 컴퓨터의 경우 배터리 잔량이 얼마 남지 않아 시스템이 곧 종료되는 경우 메모리에 남아있는 모든 데이터를 하드디스크에 복사해 두는 하이버네이션(hibernation)[1] 을 이용하여 결함에 대비한다. 저장 장치에 대해서도 결함에 대한 대비책이 많이 적용되는 것을 볼 수 있는데 RAID(Redundant Array of Independent(or Inexpensive) Disks)[2] 가 그 예이다. RAID는 여러 개의 하드 디스크를 하나의 가상 디스크(Virtual Disk)[3] 로 구성하여 대용량 저장장치로 사용할 수 있도록 하며, 여러 개의 하드 디스크에 데이터를 분할 저장하여 전송속도를 향상시키고 시스템 가동 중 생길 수 있는 하드 디스크의 에러를 시스템 정지 없이 교체하여 데이터의 자동복구가 가능하도록 하는 방법이다.

이 외에도 서버(server)의 역할을 수행하는 시스템의 경우, 동일한 작업을 수행하는 하드웨어를 두 개 이상 배치함으로써 오류가 발생해도 전체 결과에 영향을 미치지 않도록 하는 구성 요소 수준 내결함성[4] 이라 불리는 하드웨어 중복(hardware redundancy)의 기법을 사용한다.

이와 같은 하드웨어적인 결함 허용 기법은 부가적인 하드웨어 장치를 필요로 하며, 이는 전체 시스템의 비용이 증가되는 결과를 초래한다.

소프트웨어적인 결함 허용 기법으로는 발생될 수 있는 결함을 예측하여 회피하기 위한 전방 에러 복구 기법(forward recovery)과 오류가 발생했을 때 빠르게 복구하기 위한 후방 에러 복구 기법(backward recovery)이 있으며, 시스템에서 발생될 오류를 모두 예측하는 것은 사실상 불가능하기 때문에 주로 후방 에러 복구 기법에 대한 연구가 활발히 진행되었다.

후방 에러 복구 기법의 대표적인 알고리즘은 검사점 및 복구 도구(Checkpoint and Recovery Facility)[5] 이며, 이는 수행 중인 프로세스의 상태를 주기적으로 안전한 저장 장치에 저장하였다가 결함이 발생됐을 때 저장된 프로세스 정보를 이용하여 프로세스를 복구시키는 방법을 사용한다.

소프트웨어적인 결함 허용 기법은 하드웨어적인 기법과는 달리 부가적인 하드웨어 장치를 필요로 하지 않기 때문에 비용 상승의 문제는 발생시키지 않으나, 구조적 오버헤드(structural overhead)와 수행 직업의 지연 오버헤드(operational time overhead)가 발생하게 된다. 검사점 연산을 수행하기 위해 만들어지는 부가적인 코드 부분에 해당하는 구조적 오버헤드는 소프트웨어적인 방법으로 인한 불가피한 요소이기 때문에 수행 작업의 지연 오버헤드에 대한 연구가 주를 이루고 있으며, 이전까지는 프로세스의 상태 정보를 얻기 위한 과정에서 발생되는 오버헤드(extraction overhead)에 초점을 맞추어 연구가 진행되어 왔다. 이는 프로세스의 상태 정보를 디스크에 저장하기 위한 과정에서 발생되는 오버헤드(saving overhead)는 저장 매체의 발달에 따라 자연적으로 개선될 수 있다는 생각에서 기인된 문제일 것이다. 본 논문에서는 검사점 도구를 별도의 저장 공간과 소프트웨어적인 요소를 결합하여 저장 시간의 오버헤드를 개선하기 위한 연구를 다루고 있다.

현재까지의 하드웨어적인 결함 허용 기법과 소프트웨어적인 결함 허용 기법은 모두 범용 시스템을 기반으로 설계 및 구현된 방법들이기 때문에 범용 시스템과는 다른 특성을 갖는 임베디드 시스템에 적용시키기에는 적합하지 않은데, 그 이유를 살펴보면 다음과 같다.

첫째, 하드웨어적인 결함 허용 기법인 하드웨어 중복의 경우, 비용 상승의 문제와 더불어 시스템의 물리적인 크기 또한 증가되는데, 이는 저가격과 소형화라는 임베디드 시스템의 정책과 상반되는 결과를 초래하게 된다.

둘째, 소프트웨어적인 결함 허용 기법인 검사점 도구를 적용시킬 경우, 임베디드 시스템에서는 극히 일부의 경우를 제외하고는 하드디스크와 같은 대용량 비휘발성 저장 장치가 아닌 플래시 메모리(flash memory)를 사용하기 때문에 장치의 특성을 고려한 설계 및 구현이 필요하다.

마지막으로, 검사점을 만들어 내기 위한 연산에 의해 발생되는 오버헤드와 만들어진 검사점을 저장 장치에 저장하기 위해 소요되는 시간에 의한 오버헤드를 최소화시킬 수 있어야만 임베디드 시스템의 가장 중요한 특성 중에 하나인 실시간성을 해치지 않을 수 있다.

본 논문에서는 앞서 나열한 임베디드 시스템의 특성에 최대한 부합하는 형태의 결함 허용 도구를 제안한다. 제안하는 결함 허용 도구의 특징은 다음과 같다.

검사점을 저장하기 위한 저장 장치로써 NVSRAM(Non-Volatile Static RAM)[11]을 사용하고, 임베디드 시스템에서 검사점을 생성 및 이를 이용한 복구를 수행할 수 있도록 ARM 프로세서 기반의 검사점 및 복구 도구를 구현한다. User-directed[6] 방식의 검사점 형성 시스템 호출을 통해 프로세스가 명시적으로 검사점을 형성하도록 하며, 가상메모리 기반의 검사점 형성[14], 가상메모리 기반의 점진적 검사점 형성[14], 페이지 기반의 점진적 검사점 형성[12] 등의 검사점 형성 방법을 구현하여 성능을 비교하였다. 또한 이러한 검사점 도구를 통해 검사점을 만들고 이를 저장할 때, 저장 장치로써 임베디드 시스템에서 가장 일반적으로 사용되는 플래시 메모리에 검사점을 저장할 때와 원격 시스템의 메모리(SDRAM)에 검사점을 저장할 때, 로컬 시스템의 램디스크에 검사점을 저장할 때, 그리고 NVSRAM에 검사점을 저장할 때와 복구할 때의 성능의 차이를 보임으로써 제안된 방법이 임베디드 시스템을 위한 효과적인 방법임을 증명한다.

2. 본론

지금까지의 결함 허용 기법은 주로 범용시스템, 서버 시스템, 분산시스템, 병렬시스템을 대상으로 연구가 진행되어 왔으며, 후방 에러 복구 기법(backward recovery)으로 알려진 검사점 및 복구 도구에 관한 연구가 가장 활발히 수행되었다.

현재까지 제안된 검사점 및 복구 도구[6,12,13] 는 범용시스템를 기반으로 하기 때문에 검사점 저장을 위한 장치로써 하드디스크가 사용 된다. 저장 장치로써 하드디스크를 사용하는 경우 검사점을 형성하기 위한 오버헤드 외에도 검사점 데이터를 기록하기 위한 오버헤드가 발생하게 된다. 이러한 오버헤드를 줄이기 위한 방법으로 페이지 단위의 점진적 검사점 도구[12] 가 제안되었지만 검사점 기록에 드는 오버헤드는 하드디스크의 특성상 여전히 해결되지 않았다. 병렬시스템을 기반으로 설계된 Diskless checkpointing[13] 에서는 버스를 통해 공유되는 메모리에 검사점을 저장하는 방법이 제안되어 디스크에 검사점을 기록하기 위해 발생되는 오버헤드를 해소하였지만 임베디드 시스템에는 적합하지 않다.

따라서 본 논문에서는 하드웨어적 결함 허용 기법에서 사용하는 하드웨어 중복과 소프트웨어적 결함 허용 기법인 검사점 및 복구 도구를 융합시킨 형태의 검사점 도구를 제안하여 임베디드 시스템에 적합한 결함 허용 기법임 실험을 통하여 증명한다.

본 장에서는 소프트웨어적인 결함 허용 도구로써 본 논문의 설계 및 구현에서 적용시킬 검사점 및 복구 도구에 대한 분석을 통하여, 임베디드 시스템에 적용시키기 위한 방법에 대해 알아보고 검사점 및 복구 도구의 오버헤드를 줄이기 위한 알고리즘들에 대해 분석해 봄으로써 가장 작은 오버헤드를 갖는 검사점 및 복구 알고리즘을 적용하려 한다.

2.1 검사점 및 복구 도구 (Checkpointing & Recovery Facility)

후방 에러 복구(backward error recovery) 기법으로 알려진 검사점(checkpoint) 및 복구(recovery) 기법은 예상하지 못한 결함을 허용할 수 있는 매커니즘으로 소프트웨어 결함 허용 기법에서 아주 중요한 의미를 갖는다. 이 기법은 시스템에 결함 허용을 제공하기 위해 광범위하게 사용되어 왔으며 장시간 동안 수행되는 프로그램에 결함 허용을 제공하기 위해 대단히 유용하게 사용될 수 있다.

검사점 및 복구 도구는 시스템 상의 프로세스들이 결함

이 없는 상태로 수행되는 도중에 각 프로세스의 상태를 주기적으로 안전한 장소(디스크)에 저장하여 시스템에 결함이 발생하였을 때 최근 저장된 상태로부터 프로세스가 다시 시작될 수 있도록 한다. 이로 인해 시스템에 결함이 발생하더라도 프로세스가 잃어버리는 작업을 최소화함으로써 결함 허용을 제공할 수 있다. 저장된 프로세스의 상태를 검사점(checkpoint)이라 하고, 이전의 검사점 상태에서 새롭게 시작하는 절차를 복구(recovery)라고 한다.

일반적으로 프로세스의 검사점을 만들기 위해 대상이 되는 실행 프로그램에 주기적으로 인터럽트를 걸고 프로그램의 상태를 디스크에 저장한다. 저장되어야 할 정보에는 하드웨어 레지스터 상태와 프로세스의 메모리 상태를 포함한다. 복구 시에는 메모리 상태와 레지스터 상태가 재구성되어 검사점의 대상이 되었던 프로그램들은 마지막 검사점 지점부터 계속 수행될 수 있어야 한다.

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

첫째, 커널 내의 모든 주소 공간을 읽고 쓸 수 있는 권한을 가질 수 없기 때문에 접근할 수 있는 프로세스와 관련된 모든 정보를 검사점으로 만들지는 못한다는 것이다. 즉 응용 프로그램은 운영체제를 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 프로세스의 데이터 세그먼트의 주소 공간의 크기를 변경시키는 등의 시스템 콜도 제대로 지원하지 못한다. 이에 따라 커널 수준에서 구현하는 검사점 및 복구 도구와는 달리 복구를 어느 정도까지 해줄 수 있는지가 문제가 된다. 따라서 프로세스의 상태 정보를 정확하게 얻을 수 없거나 복구할 수 없는 부분이 많아지게 되고, 검사점을 만들기 위해서는 응용 프로그램이 시스템의 상태에 의존하지 않아야 한다는 제약점이 있다.

둘째, 응용 프로그램을 검사점을 위해 제공되는 라이브러리와 링크시켜 재컴파일해야 하기 때문에 응용 프로그램의 소스가 반드시 필요하다는 제약점이 있다.
운영체제 커널 수준에서 구현하는 검사점 및 복구 도구는 검사점을 위한 시스템 호출과 복구를 위한 시스템 호출을 운영체제에 추가함으로써 구현할 수 있다. 운영체제 커널 수준에서 검사점 및 복구 도구를 구현함으로써 메모리나 레지스터 영역의 상태를 손쉽고 빠르게 얻을 수 있고, 외부시스템의 상태로 접근하기가 쉬울 뿐만 아니라 검사점의 오버헤드 측면에서도 상당한 성능 향상을 가져올 수 있다.

Kckpt[6]에서 제안된 User-directed checkpoint[6]는 프로세스에서 검사점을 생성하기 위한 시스템 함수를 명시적으로 호출하는 방법을 이용하여 프로세스의 상태를 주기적으로 저장하게 되고, 이 때 프로세스가 사용하는 메모리 영역은 가상메모리를 기반으로 이루어지게 된다. <그림 1> 은 리눅스에서 프로세스가 차지하는 선형 주소 공간에 대해 실제 사용하는 메모리 영역에 대한 가상 메모리 기반의 구조를 나타낸다.

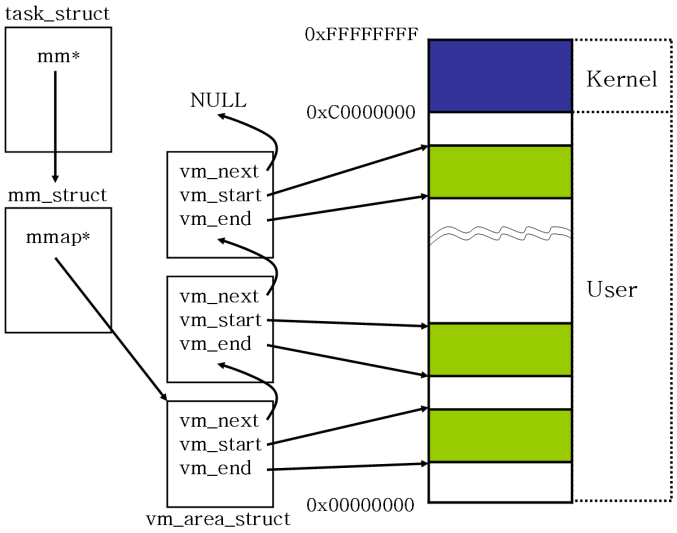

그림 1. 선형 주소 공간에서 프로세스가 차지하는 메모리

<그림 1> 의 가상메모리에서 mm_struct 타입의 mmap 포인터가 가리키는 vm_area_struct 타입의 vma 포인터는 vm_start, vm_end, vm_next 등의 필드들로 이루어진 연결 리스트로써 각각의 리스트는 프로세스의 공유메모리 비롯하여 텍스트, 데이터, 스택 영역을 가리키게 된다. Kckpt 에서의 검사점 형성을 위한 시스템 호출은 프로세스가 사용하는 메모리의 변화 여부에 관계없이 주기적으로 프로세스의 상태를 저장하게 되어 오버헤드가 크다.

검사점 및 복구 도구를 통한 소프트웨어적 결함 허용 기법에서는 오버헤드를 최소화시키기 위한 여러 가지 방법이 제안되었다. Kckpt에서의 무조건적인 프로세스 상태 저장과는 달리 각각의 메모리 청크를 단위로 하여 변화가 생긴 부분만을 디스크에 쓰기 위한 점진적 가상메모리 기반의 검사점 및 복구 도구[14] 가 제안되었고, Kckpt에서 Forked checkpoint를 제안하여 검사점 형성을 위한 프로세스를 파생시켜 원래의 프로세스가 검사점을 형성하느라 작업 수행에 지연이 발생되는 현상을 방지할 수 있는 방법을 제안하였다.

또한 [15] 에서는 오버헤드를 좀 더 효율적으로 개선하기 위하여 가상메모리 기반이 아닌 페이지 단위의 검사점 및 복구 도구를 제안하였다. 여기에서는 프로세스가 사용하는 메모리를 페이지 단위로 검사하여 변화가 발생된 페이지만을 교체하는 방식을 사용함으로써 검사점 형성에 발생되는 오버헤드를 줄이고자 하였다.

하지만 앞서 언급한 모든 형태의 검사점 및 복구 도구는 비휘발성 저장 장치로써 하드디스크를 사용하고 있기 때문에 검사점 형성에 따르는 오버헤드를 근본적으로 감

소시키기에는 다소 무리가 있다. 게다가 임베디드 시스템에서는 저장 장치로써 거의 대부분 하드디스크가 아닌 플래시 메모리 또는 플래시 롬, EEPROM 등을 사용하고 있기 때문에 이러한 장치들을 기반으로 한 검사점 및 복구 도구의 구현이 필요하다[14].

따라서 본 논문에서는 <그림 2>에서와 같은 프로세스 이미지를 주기적으로 담은 검사점 데이터를 NVSRAM에 저장하고, 이를 이용한 복구 방법을 통하여 임베디드 시스템에서의 향상된 검사점 및 복구 도구가 적용될 수 있음을 보인다.

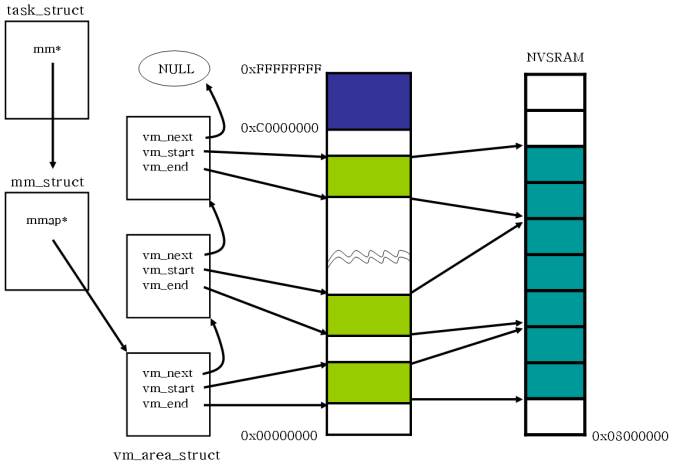

그림 2. NVSRAM 을 통한 검사점 저장

2.2 하드웨어 중복 기법 (Redundancy)

하드웨어적인 결함 허용은 어떤 한 부품에 장애가 생겼을 때 예비 부품이나 절차가 즉시 그 역할을 대체 수행함으로써 서비스의 중단이 없도록 하는 것을 말한다. 결함 허용 서비스는 소프트웨어에 의해서만 제공되는 경우도 있지만, 하드웨어에 내장된 형태 또는 몇 가지가 조합된 형태로 제공될 수 있다. 소프트웨어에서의 구현을 살펴보면, 운영체계는 프로그래머가 하나의 트랜잭션 내에서 미리 정해진 시점에 매우 중요한 데이터를 검사할 수 있도록 인터페이스를 제공한다.

하드웨어 차원에서, 결함 허용은 각 하드웨어 부품들을 이중화함으로써 달성된다. 디스크들은 당연히 미러링되며, 다중 프로세서들이 서로 계단 모양으로 묶여 있어 각자가 내놓은 결과의 정확도를 비교하게 된다. 이상이 발생되면 장애가 생긴 부품이 운영 라인에서 즉시 배제되지만, 컴퓨터 시스템은 평상시처럼 기능을 계속하게 된다.

분산시스템에서 하드웨어 중복을 통한 결함 허용 기법이 많이 사용되지만, 동일한 장치를 여거 개씩 사용하게 되므로 많은 비용 상승을 초래하게 된다.

하드웨어적인 결함 허용 기법 중에서 운영체제가 시스템의 전원이 방전되기 전에 주기억장치(RAM)의 내용을 하드디스크와 같은 비휘발성 저장장치(non-volatile storage)에 기록하는 것을 하이버네이트(hibernate)라고 한다. 하이버네이션(hibernation)과 하이버네이트에 의해 저장된 데이터를 이용한 복구는 일반적으로 콜드부트(cold boot) 보다 빠르고, 사용자와의 상호작용이 없이도 가능하다. 하이버네이션을 사용하기 위해서는 하드디스크의 남은 공간이 시스템에 존재하는 주기억장치의 용량보다 커야 한다.

Windows 계열에서는 'Windows 2000' 이상의 버전에서 하이버네이션을 제공하고 있으며 별도의 드바이스 드라이버가 필요하지 않다.

Linux에서는 '소프트웨어 서스펜드-2' 라 불리는 리눅스 커널 패치를 별도로 제공하여 하이버네이션을 지원한다[1].

하드웨어적인 측면에서의 결함 허용 기법들은 하드웨어의 중복을 통해 이루어지게 되고, 소프트웨어적인 방법에서처럼 수행중인 프로세스를 대상으로 하는 것이 아니라 정적인 데이터를 대상으로 구현된다. 또한 소프트웨어적인 결함에 의해 시스템이 마비되는 것이 아닌 하드웨어 자체의 오류에 대한 해결 방안을 제시하고 있기 때문에 프로세스 단위의 결함 허용을 꾀할 수는 없다.

2.3 복합 결함 허용 (Hybrid Fault Tolerance)

앞 절에서 지적한 바와 같이 지금까지 제기되었던 결함 허용 도구들은 모두 범용 컴퓨터 또는 분산 시스템에서 적합한 형태로 개발되었다. 따라서 한정적인 자원을 갖는 임베디드 시스템에서는 적용하기 어렵거나 효율적인 성능을 나타내기 어렵다.

임베디드 시스템 기반에서의 검사점 및 복구 도구를 구현하기 위해서는 몇 가지 고려가 필요하다. Kckpt에서 언급했던 User-directed 형식의 검사점 및 복구 도구를 임베디드 시스템에서 구현하고자 하는 경우, 임베디드 시스템에는 범용 컴퓨터와는 달리 하드디스크와 같은 대용량 비휘발성 저장 장치가 사용되지 않기 때문에 검사점 파일을 플래시 메모리 상에 기록해야 한다. 따라서 플래시 메모리의 특성과 플래시 메모리를 위한 리눅스 운영체제에서 지원하는 파일시스템에 대한 고려가 필요하다. 또한 대부분 ARM 계열의 CPU를 이용하기 때문에 하드웨어에 의존적인 운영체제의 코드들에 대한 고려가 필요하다.

본 논문에서는 앞에서 언급했던 하드웨어적인 결함 허용 기법과 소프트웨어적인 결함 허용 기법의 장점들을 뽑아 임베디드 시스템에 최적화시킨 결함 허용 기법을 제안한다.

먼저 하드웨어적인 결함 허용 기법에서 제시되고 있는 하드웨어 중복을 통해 결함이 발생되었을 때 유실될 가능성이 있는 데이터를 저장한다. 이를 위한 저장 장치로써 NVSRAM을 사용하였다. NVSRAM은 일반 시스템에서 메모리 장치로 사용되는 SDRAM에 비해 빠른 접근 속도를 가지고 있으며, 전원이 차단되어도 데이터가 유실되지 않는다. 하지만 SRAM의 특성으로 인해 대용량 데이터를 저장하기 어려우며 물리적인 크기가 증가되는 단점이 있고, SDRAM에 비해 가격이 비싸다. 시스템의 전체 메모리를 NVSRAM에 백업한다면 DRAM의 크기와 같은 크기의 NVSRAM을 사용해야 하므로 시스템의 가

격이 상당히 높아질 수 있지만, 하이버네이션처럼 전체 메모리 영역을 백업하는 것이 아니라 특정 프로세스에 대한 검사점 파일만을 백업하는 용도로 사용하기 때문에 검사점 도구를 사용하는 프로세스의 개수와 각 프로세스가 생성하는 검사점 파일의 크기만을 고려하여 NVSRAM의 크기를 줄일 수 있다. 본 논문에서는 NVSRAM을 개발 보드의 확장 메모리로 인식시키기 위한 하드웨어 설계를 포함한다. 현재 판매되고 있는 NVSRAM의 최대 크기가 2MB 이므로 8개의 NVSRAM을 가진 회로를 구성하여 총 16MB 크기의 검사점 저장소를 사용한다. <그림 3> 에 16MB 크기를 갖는 NVSRAM의 회로 설계를 보인다. NVSRAM을 사용하는 가장 큰 이유는 램디스크[16]가 갖는 장점을 최대한 살리면서 램디스크의 단점인 데이터의 휘발성을 극복하기 위함이다.

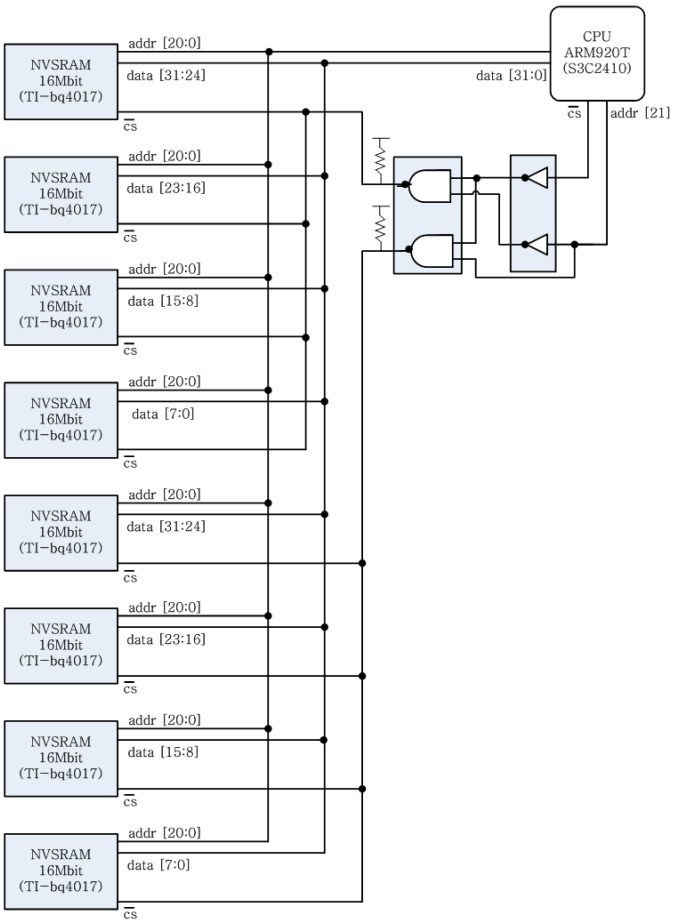

그림 3. 16MB의 NVSRAM 회로

다음으로 소프트웨어적인 기법인 검사점 및 복구 도구를 적용함에 있어 검사점의 크기를 최소화시키고 검사점 생성에 따른 오버헤드를 줄일 수 있는 페이지 기반의 점진적 검사점 도구[12,15]를 사용한다. 페이지 기반의 점진적 검사점 및 복구 도구는 페이지 단위로 이전 검사점과 변경된 페이지가 있는지 검사를 수행하여 변화가 생긴 페이지만을 교체하는 방법을 사용하기 때문에 가상메모리 기반의 검사점 형성에 비해 빠른 성능을 보인다.

3. 실험 및 결론

본 실험에서는 기존에 제안되었던 검사점 및 복구 도구를 임베디드 시스템에 적용시켰을 때의 검사점 형성 및 검사점 기록 그리고 복구에 따른 오버헤드와 본 논문에서 제안하는 복합 결함 허용 도구를 임베디드 시스템에 적용시켜 검사점 형성 및 기록 그리고 복구에 따른 오버헤드의 비교를 통하여 제안하는 기법을 통한 성능 향상을 증명하려 한다.

JFFS2 파일시스템 기반의 플래시 메모리를 사용하는 임베디드 시스템에서의 검사점 및 복구 도구의 성능과 일반적인 메모리를 나타내는 DRAM에서의 성능, 원격의 DRAM을 이용했을 때의 성능, 그리고 본 논문에서 제안하는 NVSRAM을 램디스크 형태로 사용하는 임베디드 시스템에서의 검사점 및 복구 도구의 수행 시간 비교를 보인다.

이 때, 연산 오버헤드는 배제하고 단지 검사점 기록의 오버헤드만을 측정하기 위해 커널 수준에서 검사점 기록이 시작되는 부분과 검사점 기록이 끝나는 부분의 시간 차를 구해보았다. 또한, 검사점을 호출하는 응용 프로그램을 검사점의 크기가 작게 형성되는 것과 검사점의 크기가 다소 큰 복잡한 연산을 수행하는 두 가지를 사용하여, 플래시 메모리 기반의 시스템과NVSRAM 기반의 시스템에서 검사점 크기에 따른 오버헤드도 비교해 보인다.

검사점 알고리즘에 따른 오버헤드의 측정을 위해 연산과 기록을 하나의 프로세스에서 함께 처리하는 형태의 일반적인 검사점 알고리즘과 다중 프로세서 기반에서 병렬처리를 위해 제안된 연산과 기록을 분리하여 각각 다른 프로세스가 처리하도록 만든 포크 체크포인팅 알고리즘을 이용하여 성능의 차이를 비교해 보았다.

또한, 검사점 기록에 소요되는 시간을 저장 매체의 종류에 따라, 그리고 검사점 알고리즘에 따라 측정해 보았다. 플래시 메모리를 저장 매체로 이용하는 경우에만 검사점 알고리즘이 검사점 기록 시간에 영향을 미치는 것을 확인할 수 있으며, 그 외 다른 저장 매체에서는 알고리즘이 검사점 기록에 큰 영향을 미치지 않는 것을 볼 수 있다.

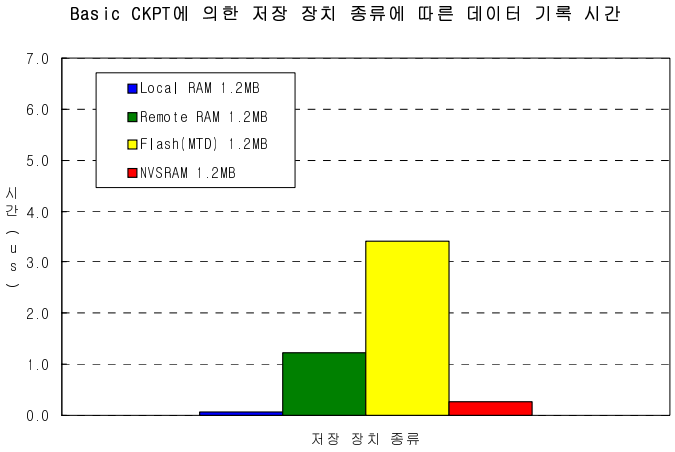

그림 4. Basic CKPT에 의한 저장 장치 종류에 따른 데이터 기록 시간

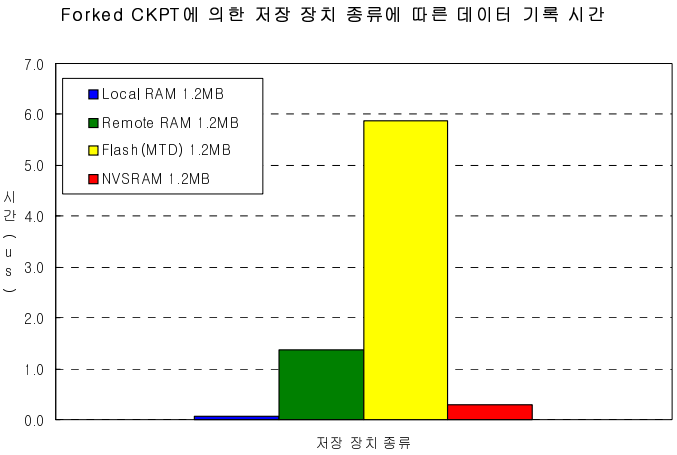

그림 5. Forked CKPT에 의한 저장 장치 종류에 따른 데이터 기록 시간

<그림 4>는 가상메모리를 기반으로 검사점을 만드는 검사점 도구를 이용하여 로컬시스템의 메모리, 원격시스템의 메모리, 플래시 메모리, 그리고 NVSRAM 각각에 검사점을 기록하는데 소요되는 시간을 측정한 결과이고 <그림 5>는 동일한 저장 장치를 대상으로 포크된 검사점 도구를 이용했을 때의 결과를 나타낸다.

실험 결과인 <그림 4, 5> 에서 SDRAM을 이용하는 램디스크를 저장 매체로 이용할 때의 검사점 기록 시간이 약 70ms/1MB 로 가장 짧은 것으로 나타났으며, NVSRAM을 이용한 검사점 기록 시간은 약280ms/1MB 로 SDRAM 보다 약 4배 가량 더 시간이 소요됨을 확인할 수 있었다. NVSRAM 은 플래시 메모리에 비해서는 읽기와 쓰기 연산 모두에서 우월한 성능을 보이지만 SDRAM에 비해서는 읽기와 쓰기 연산 모두 더 많은 시간이 소모됨을 알 수 있다.

본 실험을 통해 NVSRAM을 검사점 저장 장치로 사용할 경우 SDRAM 만큼은 아니지만, 플래시 메모리나 원격 메모리를 검사점 저장 장치로 이용할 때 보다는 검사점 기록의 오버헤드를 훨씬 적게 만들 수 있었다. 또한 임베디드 시스템 환경에서의 전력 소모까지 고려한다면 NVSRAM은 더욱 효율적인 저장 장치로써 손색이 없을 것이라 생각된다. 아직까지는 NVSRAM 의 단가가 상당히 비싼 편이고, 물리적인 크기 또한 큰 편이기 때문에 실제 임베디드 시스템에 적용시키기 어려울 것이라 생각되지만 시스템에서 중대한 역할을 하는 일부 프로세스에 한해서는 NVSRAM과 같은 메모리를 검사점 저장 장치로 활용함으로써 임베디드 시스템에서 검사점 및 복구 도구를 이용하여 시스템의 신뢰성을 향상 시키는 것이 가능할 것이다.

4. 참고문헌

[1] http://en.wikipedia.org/wiki/Hibernate_(OS_feature)
[2] Derek Vadala, Managing RAID on Linux, O'Reilly, December 2002.
[3] http://docsrv.sco.com:507/en/OSAdminG/vdmC.vdtypes.html
[4] http://www.microsoft.com/technet/prodtechnol/exchange/KO/Guides/E2k3HighAvGuide
[5] Laura L. Pullum, Software Fault Tolerance Techniques and Implementation, Artech House, p. 80~88, 2001.
[6] Jiman Hong, Y. H. Yeom and Yookun Cho, Kckpt: An Efficient Checkpoint and Recovery Facility on UnixWare Kernel, Proceedings of the ISCA 15th International Conference on Computers and Their Applications Mar. 2000, pp.303-308, New Oreleans, USA.
[7] Rémy Card, Theodore Ts'o, Stephen Tweedie, Design and Implementation of the Second Extended Filesystem, First Dutch International Symposium on Linux, December, 1994.
[8] Dr. Stephen Tweedie, EXT3, Journaling Filesystem, 20 July, 2000.
[9] http://support.microsoft.com/kb/154997/ko
[10] David Woodhouse, Red Hat, Inc., JFFS : The Journalling Flash File System, 2001.
[11] bq4016YMC-70 Specification, Texas Instrument, 2005.
[12] 이상호, 허준영, 조유근, 홍지만, 효율적인 페이지 단위 점진적 검사점의 설계 및 구현 한국정보과학회 학술발표회, pp.595-597, 2004.
[13] James S. Plank, Kai Li, Micahel A. Pueuning, Diskless Checkpointing, December 18, 1997.
[14] 박상준, 국중진, 홍지만, Implementation of Checkpointing in Embedded Environment, 2006 한국컴퓨터종합학술대회 논문집Vol. 33, No. 1(A), 2006.
[15] James S. Plank, Jian Xu, Robert H. B. Netzer, Compressed Differences: An Alogrithm for Fast Incremental Checkpointing, August 22, 1995.
[16] Paul Gortmaker, http://www.vanemery.com/Linux/Ramdisk/ramdisk-kerneldoc.txt, December, 1995.
[17] iMO Platform Board Specification, MDS Corp., 2004.08.10.
[18] S3C2410 Specification, Samsung Corp., 2003.