特輯

무정지형 상용컴퓨터 SSM6000 개발

李鐵勳, 劉昇和
三星電子 情報 컴퓨터 本部 시스템事業

Ⅰ. 서 론

디지탈 컴퓨터 시스템이 처음 개발되던 시절부터 시스템 신뢰성 (reliability) 은 주요 관심사가 되어 왔다. 초기의 진공관이나 릴레이 등과 같은 신뢰성이 없는 소자들로 만들어진 컴퓨터는 한번에 수 분 이상 연속적으로 수행하는 것 조차 어려웠다. 반도체 소자가 개발되고 그 집적도가 높아갈수록, 기본 단위 기능 당 신뢰도가 획기적으로 향상되고 있는 것이 사실이다. 그러나, 오늘날, 향상된 기능에 대한 요구에 부응하기 위한 컴퓨터 복잡도 (complexity) 의 증가가 거의 이러한 기본 단위 기능 당 신뢰도 증가 속도에 달하므로, 전체 시스템 차원에서 신뢰성 문제는 여전히 이슈가 될 수 밖에 없다. [1] 또한, 예전에는 컴퓨터 시스템의 고장이 경제나 사람의 생명에 치명적인 타격을 줄 수 있는 군사, 산업, 우주 항공 등과 같은 특수한 분야에서만 고 신뢰성 컴퓨터가 사용되었으나, 요즘은 상용 분야에까지 고 신뢰도가 요구되고 있는 실정이다.

이러한 고 신뢰도에 대한 요구를 만족시키기 위한 것이 바로 무정지형 시스템인 결함 허용 (fault-tolerant) 컴퓨터이다. 무정지형 결함 허용 컴퓨터 시스템이라 함은 그것을 구성하는 요소 중의 하나가 고장이 나더라도 계속 수행을 하도록 설계된 시스템을 의미한다. 결함 허용은 상대적인 말로서, 결함 허용성을 어느 정도 가진 시스템이라도 결함 허용 시스템으로 분류되지 않을 수도 있다. 예를 들어, 대부분의 시스템들은 디스크나 테입에 있는 데이타를 읽을 때 에러가 날 경우, 보통 여러번 재 시도를 하도록 설계되어 있다. 이것도 하나의 결함 허용 기법을 사용하고 있지만, 이러한 시스템들을 모두 결함 허용 시스템이라 부르지는 않는다. 오늘날 결함 허용으로 분류되는 시스템은 특정한 표준 수준의 결함 허용성을 가지는 시스템을 말한다. 일반적으로 인정하는 표준은 "결함 허용 시스템은 모든 단일 결함을 허용하는 시스템"이라는 것이다. 모든 상용 결함 허용 컴퓨터들은 이러한 표준을 만족하고 있으며, 멀티프로세서 (multiprocessor) 구조를 취하고 있다. [2, 3]

멀티 프로세서 구조는 크게 소 결합(loosely-coupled)과 밀 결합(tightly-coupled)의 두 가지로 나뉜다. 소 결합 멀티 프로세서 구조에서는 각 프로세서는 자기 자신의 메모리와 I/O를 가지고 있으며 메세지 교환을 통해 다른 프로세서와 통신을 한다. 이에 반해, 밀 결합 멀티 프로세서 구조는 메모리와 I/O를 프로세서들이 공유를 하며, 이러한 공유 메모리(shared memory)를 통해 프로세서간의 통신이 이루어 진다. 소 결합 결함 허용 컴퓨터 로는 Tandom [4], Stratus [5] 등이 있으며, 밀 결합 구조의 결함 허용 컴퓨터의 대표적인 예로 Sequoia [6] 를 들 수가 있다. 소 결합 멀티 프로세서 구조의 가장 큰 장점은 "결함 분리(fault isolation)" 이다. 즉, 어떤 한 프로세서에 발생한 결함은 다른 프로세서에 영향을 미치지 않는다. 그러나, 밀 결합 구조에서는 공유 메모리에 결함이 발생하면 모든 프로세서에게 영향을 미치게 된다. 반면에, 밀 결합 구조에서는 부하(load)가 자동으로 모든 프로세서에게 균등 분배 된다는 장점이 있다. [1] 다시 말하면, 프로세스들(processes)에 대한 준비 큐(ready queue)가 하나 뿐이고, 이것이 모든 프로세서들에게 공유 되기 때문에, 큐에 프로세스가 있는 한, 모든 프로세서는 어떠한 프로세스 들을 수행하게 된다. 또한, 모든 소프트

웨어는 하나의 카피(copy)만 주 메모리 상에 존재하면 되고, 이것은 모든 프로세서들에 의해 공유된다.

이에 대하여, 소 결합 구조에서는 각 프로세스들이 특정한 프로세서들에 할당이 되어야 하므로 부하 분배가 불 균등 해 진다. 이러한 소 결합 구조에서 부하의 균등 분배를 이루기 위해서는 별도의 일의 할당(task assignment) 알고리즘 [7] 등을 사용하여야 하나, 이것은 부하가 동적으로 (dynamically) 변하는 온라인 트랜잭션 처리 (on-line transaction processing) 환경에 부적합할 뿐만아니라, 프로세서가 두 개 이상이 되면 NP-hard 문제가 되므로 부하의 균등 분배를 피하기가 쉽지가 않다. [8] 또다른 방법으로 동적 부하 균등(dynamic load balancing) 방법 [9] 등을 사용할 수가 있으나, 이것 역시 프로세서 간의 통신을 통해 프로세스들을 이동 시켜야 하므로 통신하는데 많은 비용이 든다는 단점이 있다. 또한, 모든 프로세서는 운영 체제나 데이타베이스, 통신등과 같은 공통의 시스템 소프트웨어에 대해서도 각자가 자신의 메모리를 할당해야 하므로 메모리 비용도 많이 든다.

Samsung에서 최근에 개발한 무정지형 상용 컴퓨터 시스템인 SSM6000은 밀 결합 멀티 프로세서 구조로서 위에서 설명한 밀 결합 구조의 단점을 해결하기 위해 결함 허용 기능을 하드웨어 접근 방법을 사용하여 추구하였다. 즉, 모든 하드웨어 요소들을 철저히 이중화하여, 결함이 발생하면 그 클락 사이클 이내에 탐지(detection)를 하여 시스템의 다른 요소들이 그 결함에 의해 영향을 받지 않게 한다. 이렇게 함으로써, 단일 하드웨어 결함에 대해선 완전한 복구 기능을 갖게 된다. 본 논문에서는, 먼저 SSM6000 결함 허용 컴퓨터의 구조를 설명하고, 그것이 어떻게 하드웨어를 사용하여 결함을 탐지하며, 또한 운영 체제가 어떻게 결함으로부터 시스템을 복구 시키는 지에 대해 알아 본다.

Ⅱ. 시스템 개요

1. 하드웨어 구조

밀 결합 멀티프로세서 구조를 취하는 SSM6000 결함 허용 컴퓨터는 크게 프로세서 엘리먼트 (Processor Element), 메모리 엘리먼트 (Momery Element), I/O 엘리먼트 (I/O Element)로 구성되며, 이들은 모두 시스템 버스에 연결 되어 있다. (그림 1 참조) 이들 각각에 대해서 알아 보면 다음과 같다.

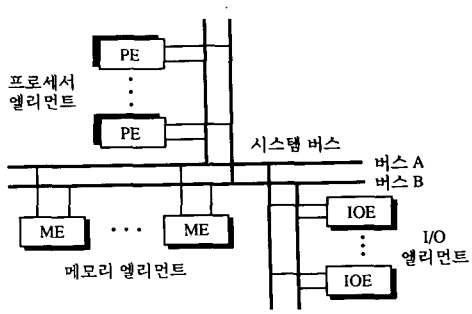

그림 1. SSM6000 하드웨어 구조

1) 시스템 버스

시스템 버스는 독립적으로 동작하는 두 개의 버스로 구성되며 (즉, 버스 A와 버스 B), 이들 각각은 40 비트, 10 MHz 클럭으로 초당 총 80 메가 바이트를 전송할 수 있다. 시스템 버스에는 모두 12 개의 슬럿이 있으며, 이 중 4개는 프로세서 엘리먼트 전용 슬럿이며, 메모리 엘리먼트 전용으로 4 개의 슬럿이 할당되어 있다. 그리고 나머지 4 개의 슬럿 중 2 개는 I/O 엘리먼트 전용이고, 나머지 2 개는 메모리 또는 I/O 엘리먼트가 공유할 수 있도록 설계되어 있다. 따라서, 시스템의 최대 구성은 4 개의 프로세서 엘리먼트, 4 개의 메모리 엘리먼트, 그리고 4 개의 I/O 엘리먼트가 연결된 4:4:4 구성, 또는 4:6:2 구성이 가능하다. 결함 허용 기능을 갖기 위해선 최소로 각 엘리먼트들이 2 개씩 연결된 2:2:2 구성 이어야 한다 또한 각 버스에는 버스 인터페이스 (Bus Interface) 보드가 장착되어 있어서, 그 버스의 중재 (arbitration) 역할을 하며 결함에 대한 각종 정보를 저장한다. 이들 두 버스는 평상시에는 모두 사용되며, 이 중 하나에 결함이 발생하면 다른 쪽 버스가 모든 데이타 전송을 전담하게 된다. 시스템 버스는 프로세서가 메모리를 억세스할 때, 또는 메모리와 메모리 사이의 데이타 전송, 그리고 메모리와 버스 어뎁터 사이의 데이타 전송 시에 주로 이용된다. 이와 같이, 시스템 버스는 거의 대부분 주 메모리를 억세스할 때 사용되는데, 이 경우 버스 A, B 중 어떤 버스를 사용할 것인지는 메모리 영역에 따라 시스템 초기화 시에 미리 결정이 된다.

버스 A, B 중 하나를 'executive' 버스로 정하

고, I/O 인터럽트나 결함 인터럽트 등과 같은 긴급 정보는 모두 executive 버스를 통해 전달되며, 특히 시스템 결함 복구 도중에는 executive 버스만을 사용한다. 이러한 executive 버스로서의 역할은 버스 A, B가 주기적으로 담당을 하게 되어 있으며, 각 프로세서가 executive 버스 변환을 요구할 수도 있다. 버스를 통한 데이타 전송에는 다음과 같은 세 가지 타입이 있다.

(1) Normal Read/Write : 이것은 프로세서가 메모리를 억세스 할 때 사용된다. 데이타 전송 단위는 캐쉬 블럭 사이즈인 128 바이트이며, 한번의 오퍼레이션으로 최대 16 블럭을 전송할 수 있다. 128 바이트를 전송하는 데에 걸리는 시간은 최대 4 us이다

(2) DMA Read/Write : 메모리와 메모리 사이, 또는 메모리와 버스 어뎁터 사이의 데이타 전송 시에 사용된다. 데이타 전송을 위해 필요한 source 및 destination 어드레스는 프로세서 엘리먼트에 의해 지정된다. 일단 어드레스가 지정이 되면, source 모듈은 곧 바로 버스를 통해 destination 모듈에게 데이타를 제공한다. 또한, 두 개 이상의 destination 모듈이 지정될 수도 있는 데, 예를 들면, 하나의 디스크에서부터 프라이머리와 쉐도우 메모리에 모두 데이타를 전송할 경우에 사용된다.

(3) Special Read/Write : 이것은 프로세서가 시스템 내의 다른 모듈에 있는 명령 레지스터 (command register)에 write하거나, 또는 상태 레지스터 (status register)를 read 할 때 사용된다.

2) 프로세서

프로세서 엘리먼트는 모토롤러의 32-비트 마이크로프로세서인 MC68040 (25/33 MHz) 을 가진 동일한 보드 두 장으로 구성되어 있으며, 이들은 서로 lock-step 형태로 동작을 한다. 또한, 비교기 (comparator)가 있어 매 클락 사이클마다 이들 둘의 결과를 비교하여 서로 상이할 경우 결함이 발생한 것으로 간주한다. 이것을 '자기 체크 (self-checking)' 라 부른다. 각 프로세서 엘리먼트는 자신의 로컬(local) 클락으로 구동이 되므로 클락 회로 자체에 결함이 발생해도 다른 프로세서 엘리먼트에 영향을 미치지 않는다. 또한 모든 프로세서 엘리먼트에는 억세스 시간(access time)이 40ns인 1 메가 바이트의 4단 세트 어소시어티브 캐쉬 메모리(4-way set associative cache memory)를 가진다. 캐쉬의 반은 read-only 인스트락션 캐쉬이고, 나머지 반은 writable 데이타를 저장하는 데에 사용된다. 이들 캐쉬 메모리는 이중화 되어 있지는 않고, 두 보드의 캐쉬가 각각 데이타의 반씩을 저장한다.

캐쉬는 non-write-through 방식을 사용한다. 다시 말하면, 캐쉬에 어떤 데이타가 쓰여 지면 이것이 곧 바로 주 메모리에 쓰여지지 않고, 운영 체제가 캐쉬의 더티 블럭(dirty block)을 주 메모리에 플러쉬(flush) 시키도록 요청하거나, 또는 캐쉬 오브 플로우(cache overflow)가 발생했을 때 캐쉬 메모리 내의 변경된 데이타가 실제로 주 메모리에 쓰여진다. 또한, 모든 더티 블럭을 하나의 명령어 (instruction)로 플러쉬할 수 있도록 하드웨어가 설계되어 있다. 1024 개의 캐쉬 블럭을 플러쉬하는 데에 최대 약 4 ms 정도의 시간이 소요된다.

밀 결합 구조의 가장 큰 문제점으로 시스템 버스 트래픽(bus traffic)을 들 수가 있는데, 대부분의 버스 트래픽은 프로세서가 공유 메모리를 억세스함으로써 발생한다. 그러므로 버스 트래픽을 감소 시키기 위해선 캐쉬 메모리의 용량을 증가시켜야 한다. 이러한 큰 용량의 어소시어티브 캐쉬를 사용하여 캐쉬 히트율(hit ratio)을 99% 이상으로 높일 수 있다 . [10] 또한, non-write-through 방식을 써서 공유 메모리 억세스 회수를 줄일 수 있다. 그리고, 캐쉬와 공유 메모리 사이에 전송되는 데이타의 단위를 128 바이트로 높임으로써, 단위 바이트 전송에 대한 프로토콜 오버헤드(protocol overhead)를 줄인다.

3) 주 메모리

각 메모리 엘리먼트는 하나의 보드로 구현되며, 두 개의 동일한 모듈로 구성되어 있고 각 모듈은 메모리 데이타 워드의 반을 (즉, 각 16 비트씩) 저장한다. 어드레스 정보는 두 모듈이 서로 공유한다. 공유 클럭이나 공유 refresh 요청 시그널들을 제외한 모든 콘트롤 신호는 두 모듈에서 동시에 발생하게 되어 있다. 프로세서 엘리먼트와 마찬가지로, 매 클럭 사이클마다 내부 에러 탐지를 위한 자기 체크를 한다.

각 메모리 엘리먼트는 64 메가 바이트의 4단 인터리브드 (4-way interleaved) RAM으로 구성되며, 억세스 시간은 100ns이다. 그리고, 매 16 메가 바이트 마다 1024 개의 test-and-set 락(lock)을 두어 운영 체제로 하여금 공유 메모리에 대한 상호 배제적인 (mutually exclusive) 억세스를 할 수 있게 한다. test-and-set(X,Y)은 atomic하게 수행되며 다음과 같다: (1) 만약 X = 0 이면 (즉, 락이 걸리지

않은 상태), Y(〉 0) 값을 X에 대입함으로써 락을 건다. 그리고 (2) test-and-set이 수행되기 이전의 X 값을 리턴한다.

메모리 에러를 탐지하기 위하여, 각 메모리 엘리먼트에는 패리티 발생 회로 및 패리티 체크 로직이 있다. 또한, 메모리 영역들에 대한 억세스 및 변경 상태를 저장하기 위한 별도의 페이지 상태 메모리(Page Status memory)를 가지고 있어서 변경되지 않은 데이타들을 디스크로 다시 write하는 오버헤드를 없앨 수 있다. 그리고 각 메모리 엘리먼트의 실제 어드레스 영역은 운영 체제가 지정할 수 있도록 되어 있다.

주 메모리는 크게 쉐도우 모드 (shadow mode)와 언쉐도우 모드 (unshadow mode)로 구성할 수 있다. 쉐도우 모드는 프라이머리 (primary) 와 쉐도우 메모리 보드 쌍으로 구성되어 있으며, 프라이머리에 있는 모든 writable 데이타에 대한 백업 카피를 쉐도우 메모리가 가진다. 이들 두 보드의 logical 어드레스는 동일하며, 이들은 어드레스의 최상위 비트로 서로 구분된다. (Logical 어드레스에서는 최상위 비트를 사용하지 않는다.) 주 메모리 상에 데이타를 write할 경우에는 프라이머리와 쉐도우 메모리에 모두 write하며, read 시에는 프라이머리를 사용한다. 만약 프라이머리 메모리에 결함이 발생하면, 쉐도우 메모리로 부터 read한다. 프라이머리에 결함이 발생하더라도 read-only 데이타나 코드들은 디스크 상에 존재하므로 복구가 가능하기 때문에 이들은 쉐도우시키지 않는다. 이렇게 함으로써, 쉐도우에 따른 오버헤드를 줄일 수 있다.

언쉐도우 모드로 메모리를 구성할 수도 있는데, 이 경우 모든 메모리들은 프라이머리가 된다. 그러나 메모리 결함 발생 시에 복구가 불가능하므로, 결함 허용을 위해선 쉐도우 모드로 사용하여야 한다. 따라서 메모리 보드는 항상 짝 수개로 구성이 된다. 그러나, 쉐도우 모드로 사용 중에 메모리 결함이 발생하면, 해당하는 메모리 보드는 시스템에서 제거가 되므로 그 메모리 쌍의 남은 한 쪽 메모리는 언쉐도우 모드로 사용이 될 것이며, 새로운 메모리 보드를 시스템에 추가하면 자동으로 쉐도우 모드로 돌아간다.

4) I/O 서브시스템

그림 2에서 보는 바와 같이 I/O 엘리먼트는 시스템 버스에 연결된 버스 어뎁터(BA)와 I/O 버스로 사용되는 산업 표준 VME 버스에 연결된 VME 어뎁터(VA)로 구성되며, BA와 VA는 전용 고속 버스인 E-버스 (External bus)로 연결되어 있다.

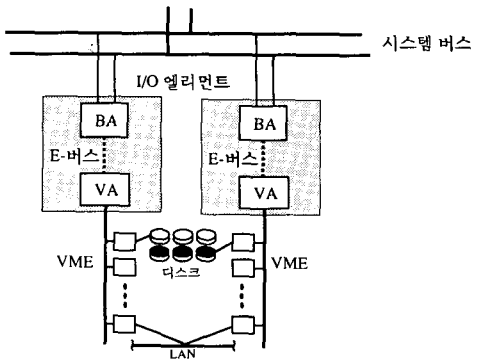

그림 2. I/O 서브시스템

버스 어뎁터는 네 개의 32K 바이트 버퍼와, 동기화 및 직접 메모리 억세스 (direct memory access) 회로를 가지고 있으며, 주 메모리와 VME 어뎁터 사이의 데이타 전송을 담당한다. VME 어뎁터에는 두 개의 MC68030 (32 MHz) 마이크로 프로세서가 있으며, 매 클락 사이클 마다 자기 체크(self-checking)를 한다. 또한, 자체의 운영 체제인 VAOS를 저장 하기 위한 1 메가 바이트의 no-wait-state RAM과, I/O 버퍼용으로 4 메가 바이트의 RAM이 있다. 버스 어뎁터와 VME 어뎁터는 초당 10 메가 바이트의 전송 능력이 있는 전용 버스로 연결되어 있다. 각 VME 버스는 8 개 까지의 테이프, 디스크, 및 통신용 콘트롤러를 지원하며, 초당 40 메가 비트의 전송 능력이 있다.

SSM6000의 I/O 서브시스템 (subsystem) 은 결함 허용을 위해 모두 이중화되어 있다. 그림 2에서 보는 바와 같이, 두 개의 I/O 엘리먼트를 사용하여, 하나는 프라이머리로 또 하나는 백업 용으로 사용한다. 또한 모든 I/O 콘트롤러들은 서로 다른 I/O 엘리먼트에 연결된 것들끼리 쌍으로 묶어서, 프라이머리 및 백업으로 사용된다. 디스크 상의 데이타 무결성 (integrity)를 위해 모든 디스크들은 미러 (mirror) 되어 있다. 이렇게 하여, I/O 서브시스템 내의 모든 단일 하드웨어 결함으로부터 완벽하게 복구를 할 수 있다.

2. 결함 탐지 장치

결함 탐지 장치로는 에러 탐지 코드(error detecting

code), 자기 체크(self-checking), 그리고 프로토콜 감시 (protocol monitoring) 등 세 가지 방법을 사용한다. 이러한 결함 탐지 장치로 시스템 내의 모든 단일 하드웨어 결함을 탐지할 수 있다.

1) 에러 탐지 코드

모든 데이타는 에러 탐지 코드를 이용하여 버스를 통한 전송 중에서나 또는 모든 메모리 영역 (즉, 주 메모리, 프로세서 앨리먼트의 캐쉬, 버스 어뎁터의 버퍼, 그리고 VME 어뎁터의 로컬 메모리 등) 에서 에러가 탐지되게 된다. 에러 탐지 코드로는 모두 바이트 패리티 (parity) 를 사용한다. 또한, 인코드 (encoder) 및 디코드(decoder) 자체의 에러도 확장된 해밍 코드(extended Hamming code)를 이용하여 탐지된다. 항상 4 바이트 어드레스나 데이타 워드 (word) 중 반은 이븐 패리티(even parity), 나머지 반은 오드 패리티 (odd parity)를 사용하여 전형적인 에러 패턴인 모든 비트가 0이거나 1인 경우도 에러로 탐지가 되게 되어 있다.

2) 자기 체크

에러 탐지 코드가 일반적으로 에러를 탐지하는 방법 중 가장 비용이 적게 드는 방법이나, 매우 복잡한 로직(logic)의 하드웨어에 대해서는 그것을 위한 인코드 및 디코드의 구현이 하드웨어 그 자체 보다 더 비용이 드는 경우도 있다. 예를 들면, 마이크로 프로세서나 어드레스 발생 로직, 그리고 캐쉬를 관리하는 로직 등이 그것이다. 이러한 것들에 대한 에러 탐지를 위해서는 그 하드웨어 자체를 똑같은 한 쌍으로 이중화 하고 비교기(comparator)를 통해 그들의 수행 결과를 비교하는 방법이 훨씬 경제적이다. 이러한 비교기 자체도 이중화 하여 비트 단위로 테스트 한다.

자기 체크를 위해 각 엘리먼트의 회로는 두 개의 동일한 모듈로 구성이 되어 있다. 이들 각 모듈은 그 자체로는 완전한 기능을 갖춘 회로가 아니지만, 두 모듈을 합하여 완전한 기능을 발휘한다. 그림 3에서 보듯이, 각 모듈은 데이타 콘트롤 및 처리 유니트 (processing unit)를 각자 가지고 있으며, 완전한 기능을 갖기 위해 필요한 메모리에 대해선 각각 반씩을 가지고 있다. 두 모듈의 처리 유니트들은 동일한 데이타를 동시에 처리하며, 동일한 어드레스 정보가 각 메모리에 제공되게 되어 있다.

에러 탐지를 위하여, 각 콘트롤 유니트에서 해당 메모리로 보내어지는 데이타 및 어드레스 정보들을 서로 비교를 하게 된다. 또한, 외부 버스를 통해 회로로 전달되는 정보나 혹은 콘트롤 유니트가 만들어 내는 정보들은 에러 탐지 코드를 통해 인코드 (encode) 된다.

3) 프로토콜 감시

위의 두 가지 방법으로는 모든 하드웨어 결함을 탐지할 수 없다. 예를 들면, 프로세서가 메모리 엘리먼트를 억세스 하는 데 있어서, 만약 메모리 엘리먼트 내부의 결함으로 인해 거기에 대한 응답이 없다면, 프로세서와 메모리 엘리먼트, 그리고 그들 사이의 버스는 더 이상 사용할 수 없게 된다. 프로토콜 감시기 (protocol monitor)는 엘리먼트들 사이의 통신에 관한 정해진 순서(sequence)나 타이밍(timing)에 대한 위반을 탐지함으로써 그러한 문제들을 해결한다.

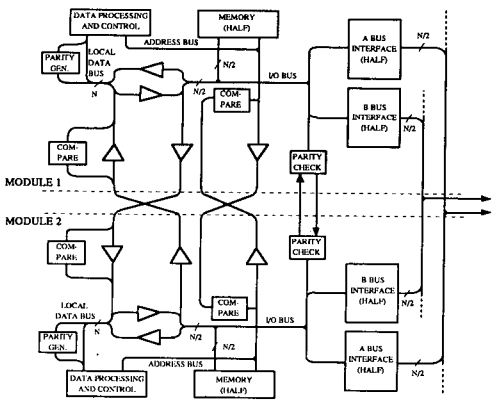

그림 3. 자기 체크 회로(여기에서 N은 전체 데이타 비트와 패리티 비트의 합이다)

이와 같은 결함 탐지 장치들을 통해 하드웨어 결함으로부터 발생되는 모든 단일 에러(single error)는 그것이 발생되는 순간 탐지가 된다. 그러나, 결함 탐지 회로 자체의 에러는 탐지할 수 없으나, 이것은 운영 체제로 하여금 주기적으로 이러한 탐지기들을 테스트하게 함으로써 그들의 에러를 탐지하게 한다. 일단 결함이 탐지가 되면, 해당되는 엘리먼트는 외부로의 출력을 즉시 중단함으로써 그 결함이 다른 엘리먼트로 영향을 미치지 않게 한다. 운영 체제는 결함이 발생한 엘리먼트를 억세스 할 때 그 결함을 알게 되고, 그 즉시 (나중에 설명 하겠지만) 결함 복구 기능을 수행하게 된다.

또한, 소프트웨어 로직 에러를 탐지하기 위하여 운영 체제 내에 'executive assertion'들을 설정해 놓고 있다. Executive assertion이란 운영 체제 내의 각 부분에서 설정해 놓은 기대치들을 말하며, 운영 체제가 수행한 실제 결과치들이 이러한 assertion들과 비교하여 서로 상이할 경우 소프트웨어 결함으로 처리된다.

3 운영 체제 구조

트랜젝션 처리(transaction processing)를 위한 고 성능(high performance) 기능을 가지면서, 또한 표준 운영 제제의 장점을 살리기 위해, Samsung은 Sequoia와 함께 유닉스 시스템 V(Unix System V)의 모든 기능을 가지는 독자적인 커널(kernel)을 구현하였다. [11]

시스템의 구성이 결함 허용을 위한 최소 구성인 2:2:2 (즉, 프로세서, 메모리, I/O 엘리먼트들이 각각 두 개씩인 구성) 이상이면, SSM6000의 커널은 사용자에게 단일 하드웨어 에러로 부터 항상 프로세스(process)와 화일(file)을 보호하는 기능을 제공한다. 예를 들어, 어떠한 프로세스를 수행하던 프로세서 엘리먼트에 결함이 발생하면, 이 프로세스는 다른 프로세서 엘리먼트에 할당이 되어 해당하는 응용 프로그램에 영향을 주지 않고 계속 수행을 하게 한다. 그리고, 모든 메모리 엘리먼트에 저장된 데이타나 코드는 그것의 카피가 시스템의 다른 곳 (즉, 다른 메모리 엘리먼트나 디스크)에 항상 존재하므로 하나의 메모리 엘리먼트에 결함이 발생하더라도 복구를 할 수 있다. 또한, 서로 다른 I/O 엘리먼트에 연결 된 디스크끼리 미러링(mirroring) 시키는 방법을 사용하여 디스크나 콘트롤러, 혹은 I/O 엘리먼트에 결함이 발생하더라도 원하는 화일을 억세스할 수 있다.

SSM6000의 밀 결합 구조에서는 앞에서 설명한 대로 커널은 주 메모리 상에 하나의 카피만 존재하고 모든 프로세서 엘리먼트가 이를 공유한다. 즉, 모든 프로세서는 자신에게 할당 된 프로세스들이 수행하는 시스템 콜(system call)을 통해 동시에 제각기 해당하는 커널 코드를 수행한다. 이와 같이, 여러 개의 프로세서가 동시에 커널을 수행하기 때문에 공유 커널 데이타를 억세스 할 때 프로세서들 사이의 동기화 문제가 발생하는데, SSM6000은 test-and-set 락과 캐쉬 플러쉬를 콘트롤 함으로써 이 문제를 해결한다.

Ⅲ. 프로세서 동기화

단일 프로세서(single processor) 구조에서는 커널 데이타 구조(kernel data structures)의 무결성(integrity)을 보호하기 위해 보통 다음의 두 가지 방법을 사용한다: 유닉스 운영 체제에서는 커널은 커널 모드 (kernel mode)에서 수행되는 프로세스를 프리엠트(preempt) 시켜 다른 프로세스로 컨텍 스위치 (context switch) 하지 않는다; 그리고, 임계 구역 (critical section)을 억세스 할 때에 그 프로세스의 우선권 등급 (priority level)을 높여 주어 그 등급 보다 낮은 인터럽트 (interrupt)를 막아 주고 억세스가 다 끝나면 다시 우선권 등급을 내린다.

그러나, SSM6000과 같은 멀티 프로세서 구조에서는 여러 개의 프로세서들이 동시에 수행하기 때문에 이러한 방법만으로는 커널의 데이타 구조들을 보호할 수가 없다. 여기에서는 프로세서 동기화를 위한 SSM6000의 방법들을 설명한다.

1. 상호 배제 프로토콜

여러 프로세서들이 동시에 커널의 프로세스 디스크립터(process discriptors), 페이지 테이블(page tables), 또는 화일 디스크립터(file discriptors) 등과 같은 공유 데이타 구조들(data structures)을 억세스 하려고 할 때 레이스 컨디션(race condition)이 발생한다. 이러한 레이스 컨디션을 방지하기 위해, 그러한 공유 데이타들을 "임계 구역(critical section)" 으로 지정하고 그들에 대한 억세스를 상호 배제 프로토콜(mutual exclusion protocol)을 사용하여 관리 하여야 한다. 여기에서는 SSM6000의 상호 배제 프로토콜에 대해 설명한다.

SSM6000의 상호 배제 프로토콜에서는 임계 구역으로 지정된 모든 데이타 구조마다 test-and-set 락을 하나씩 부여하고, 프로세서들이 임계 구역 내의 데이타를 억세스 하기 위해선 우선 그 임계 구역에 할당된 락이 다른 프로세서에 의해 걸려 있는지를 확인하고 하여, 만약에 그렇지 않다면 락을 걸고 데이타를 억세스한 다음 락을 풀어주게 한다.

위와 같은 test-and-set 락 만으로는 모든 문제를 해결을 할 수 없다. 왜냐하면, 각 프로세서 엘리먼트의 캐쉬 메모리가 non-write-through 방식을 사용하기 때문에, 앞에서 설명한 대로 캐쉬에 있는 데이

타의 값이 바뀌더라도 이것이 곧 바로 공유 메모리에 쓰여지지 않기 때문이다. 그러므로, 락을 풀어 주기 전에 캐쉬 상의 더티 블럭(dirty block)을 플러쉬 하지 않으면 다른 프로세서가 변경되기 전의 잘못된 데이타를 억세스하는 문제가 생긴다.

또한, 임계 구역을 억세스 하기 위해 락을 건 다음, 캐쉬 내의 넌더티 블럭(non-dirty block)을 무효화(invalidate)시켜야 한다. 그렇지 않으면, 이전에 억세스를 한 임계 구역 내의 데이타를 그 프로세서가 다시 억세스할 때 공유 메모리에 있는 데이타를 가져오지 않고, 캐쉬에 남아 있는 데이타를 바로 억세스 하게 된다. 만약에, 그러한 연속된 두 억세스 사이에 공유 메모리에 있는 그 데이타의 값이 다른 프로세서에 의해 변경되었다면 문제가 된다. 그러므로, 락을 건 다음 모든 넌더티 블럭을 무효화 시킴으로써 임계 구역 내의 데이타를 억세스할 때 공유 메모리로 부터 직접 데이타를 가져오게 한다. 이러한 모든 넌더티 블럭을 무효화 시키는 것도 하나의 명령어로 수행이 되도록 하드웨어가 설계되어 있다.

위의 모든 문제점들을 해결한 SSM6000의 상호 배제 프로토콜은 다음과 같다: (1)test-and-set 락을 건다; (2)넌더티 블럭을 무효화 시킨다; (3)임계 구역 코드를 수행한다; (4)더티 블럭을 플러쉬한다; 그리고 (5)락을 풀어 준다. 이와 같은 락킹 (locking) 및 캐쉬 플러쉬 메카니즘을 사용하여 캐쉬 통일성(cache coherence) 문제를 완전히 해결한다.

교착 상태(deadlock)을 방지하기 위한 방법으로는 전통적인 전체 오더링(totally ordering) 방법을 사용한다. 즉, 락에 대한 등급을 정하고, 모든 프로세서가 어떠한 락 k을 요구할 때에는 먼저 자신이 소유하고 있는 락 중에 k보다 등급이 낮은 모든 락을 풀어 주게 함으로써 교착 상태를 방지한다.

2. 락 컨텐션

프로세서가 임계 구역을 억세스하려고 할 때, 만약 해당하는 락이 다른 프로세서에 의해 사용 되고 있다면 그 락이 풀릴 때까지 기다려야 한다. 그러므로, 락 컨텐션(lock contention)이 자주 발생하면 시스템의 성능을 저하하게 되고, 프로세서의 수를 늘리더라도 성능 향상을 기대할 수 없게 된다. 이러한 락 컨텐션을 줄이기 위해서는 프로세서가 락을 가지는 시간, 즉 락 시간(locked time)과 프로세서가 락을 요구하는 빈도 수(locking rate)를 줄여야 한다. 락 시간을 줄이기 위해 모든 프로세서는 커널 코드를 수행할 때만 락을 가지게하고, 또한 컨택 스위치 시에는 락을 풀어 주게 한다. 예를 들어, 동기 I/O (synchronous I/O)를 요청한 직후 커널은 모든 락을 바로 풀어 주어야 한다.

락 컨텐션을 줄이는 또 하나의 방법으로 모든 중요한 데이타 구조를 분할하는 방법을 사용한다. 만약에, 전체 커널 테이블(kernel table)에 대한 락을 하나만 사용한다면, 단 하나의 프로세서만 그 테이블 내의 데이타를 억세스할 수 있고, 모든 다른 프로세서 들은 그 테이블 내의 다른 데이타를 억세스 할려고 해도 락이 풀릴 때까지 기다려야 한다. 그러므로, 큰 커널 테이블을 보다 작은 서브 테이블(subtable)로 분할하고 각각의 서브 테이블에 락을 하나씩 할당한다. 이렇게 함으로써, 여러 프로세서가 동시에 테이블을 억세스할 수 있게 된다. 이것을 '락 분할(lock partitionng)'이라 부른다.

3. 버스 컨텐션

시스템의 성능에 영향을 미치는 또 하나의 요인은 각 프로세서가 캐쉬를 플러쉬하는 빈도수이다. 만약, 락을 풀어 줄 때 마다 캐쉬를 플러쉬 한다면, 플러쉬가 진행될 때에는 프로세서는 어떠한 일도 수행할 수 없기 때문에, 프로세서의 이용율(processor utilization)을 떨어뜨릴 뿐 아니라 시스템 버스를 사용하므로 버스 컨텐션(bus contention) 문제를 야기한다.

이러한 문제를 해결하기 위하여 캐쉬 플러쉬는 락을 풀어 줄 때 곧 바로 캐쉬를 플러쉬 하지 않고, 캐쉬 오브 플로우(cache overflow)가 발생했거나, 시스템 콜(system call)의 완료 또는 컨텍 스위치로 인한 사용자 상태(user state)로의 전환이 될 때 실제로 캐쉬를 플러쉬 한다.

Ⅳ. 결함 복구 장치

시스템은 항상 프로세서, 메모리, I/O, 버스, 콘트롤러, 혹은 주변 기기 등과 같은 하드웨어에 결함이 발생할 수 있다. 어떠한 하드웨어 결함이 언제, 어떻게 발생하더라도 수행되던 모든 프로세스는 복구가 가능해야 하고, 또한 어떠한 I/O 오퍼레이션(operation)도 분실이 되거나, 중복되어서는 않된다.

여기에서는 모든 하드웨어 결함들에 대한 결함 복구 (fault recovery) 장치에 대해 설명한다.

1. 프로세서 결함

캐쉬 플러쉬가 진행 중인 때를 제외하고는 운영 체제는 항상 주 메모리의 일관성(consistency)을 유지한다. 다시 말하면, 프로세서가 캐쉬로 부터 주 메모리로 무엇이든 쓸 때, 그 시점 부터 다시 수행이 시작될 수 있도록 프로세서의 레지스터(registers) 값들을 포함한 모든 정보들을 메모리로 덤프(dump)한다. 이것을 "체크 포인트(checkpoint)" 라고 부른다. 이러한 체크 포인트를 통해 모든 프로세스들의 일관된 상태(consistent state)가 메모리에 항상 유지가 된다. 만약, 캐쉬 플러쉬를 하지 않을 때에 프로세서에 결함이 발생하면, 그 프로세서에서 수행되던 모든 프로세스들은 직전의 체크 포인트 시의 상태들이 주 메모리에 유지되어 있으므로 다른 프로세서에 의해 그 시점 부터 다시 수행이 가능하다.

그러나, 캐쉬 플러쉬가 진행 중일 때 결함이 발생하면, 메모리 상의 프로세스들은 비일관적인 상태(inconsistent state)가 된다. 이 문제를 해결하기 위해 주 메모리의 모든 writable 페이지(pages)들은 쉐도우(shadow) 시킨다. 즉, writable 페이지들은 다른 메모리 엘리먼트에 백업 카피(back-up copy)를 둔다. 프로그램 페이지나 read-only 페이지들은 캐쉬 플러쉬와 무관하므로 쉐도우 시키지 않는다.

체크 포인트는 항상 백업 카피부터 플러쉬하고 난 다음, 프라이머리(primary)를 플러쉬 한다. 이렇게 해서, 프라이머리와 백업 카피 중 적어도 하나는 항상 일관된 상태를 유지한다. 다시 말하면, 만약에 k번째 체크 포인트 시에 백업 카피 플러쉬 중 프로세서 결함이 발생하면, 프라이머리는 일관된 (k-1)번째 체크 포인트의 데이타를 가지게 되고, 프라이머리 플러쉬 중 프로세서 결함이 발생하면, 백업 카피는 일관된 k번 째 체크 포인트의 데이타를 가지게 된다.

이와 같이, 모든 프로세스는 메모리 상에 일관된 카피가 유지되므로 어떠한 프로세서 결함으로 부터도 복구가 가능하다. 즉, 어떤 프로세서에 결함이 발생해도 그 프로세서가 수행하던 모든 프로세스들은 (엄밀히 말하면, 메모리 상에 존재하는 그들의 일관된 카피들) 다른 프로세서가 수행하는 커널에 의해 준비 큐(ready queue)에 들어 가게 되고, 일정한 시간이 되면 다른 프로세서들에 의해 수행이 된다

2. 메모리 결함

모든 writable 페이지는 두 개의 메모리 엘리먼트 보드에 쉐도우 되어 있고, 모든 프로그램이나 read-only 데이타 들은 디스크 상에 백업 카피가 있다. (운영 체제도, 나중에 설명하겠지만, 결함 복구를 위해 쉐도우 되어 있다.) 그러므로, 어떠한 메모리 엘리먼트에 결함이 발생하더라도 그 메모리 엘리먼트에 있는 모든 페이지들은 시스템의 다른 부분에 존재한다. 따라서, 메모리 엘리먼트 결함으로 부터 복구가 항상 가능하다.

메모리 엘리먼트가 결함을 탐지하게 되면, 어떠한 억세스에 대해서도 응답을 하지 않는다. 그런 다음, 그 메모리를 억세스할려는 프로세서는 버스 전송 timeout 신호를 받게 되고, 이 때 결함 복구 루틴으로 들어 간다. 결함 복구가 진행되는 동안, 그것이 메모리 내부 결함일 경우, 모든 주 메모리는 언쉐도우 모드가 된다. 복구가 끝나면, 운영 체제는 다시 사용가능한 자원들을 재구성하여 메모리를 쉐도우시킨다. (물론, 이 때 메모리 엘리먼트가 쉐도우될 만큼 충분하지 않을 경우에는, 언쉐도우 모드로 남게 된다.)

3. I/O 결함

모든 I/O 엘리먼트는 수행해야 할 I/O 오퍼레이션에 대한 큐를 주 메모리에 가지고 있다. 모든 프로세서는 I/O 오퍼레이션에 대한 명세서(discription)를 자신의 캐쉬에 만들고 이것을 주 메모리의 해당하는 큐에 플러쉬한다. 일단, I/O 명세서가 큐에 들어가게 되면 그 I/O 오퍼레이션의 수행은 보장 받는다. I/O 엘리먼트로 부터 I/O 수행을 완료하였다는 통보(acknowledge)를 받으면, 해당하는 I/O 명세서는 큐에서 지워진다.

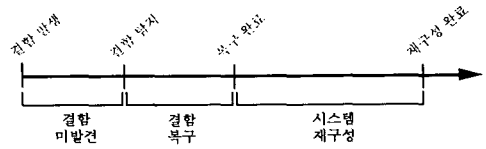

그림 4. 결함 복구 단계

만약, 캐쉬 플러쉬 중에 프로세서 결함이 발생한다면, 위에서 설명한 대로 다음의 두 가지 상태가 된다. 첫째로, 백업 카피 플러쉬 중에 발생한 경우인

데, 이 때에는 실제로 I/O 명세서가 큐에 들어가지 않게 되고 프로세스 또한 I/O를 요청하기 직전의 상태로 복귀가 된다. 둘째로, 프라이머리 플러쉬 중 결함이 발생한 경우인데, 이때는 I/O 명세서가 큐에 들어가 있으며, 또한 프로세스도 I/O를 요청한 직후의 상태로 복귀된다. 그러므로, 모든 I/O 오퍼레이션은 분실이나 중복이 되지 않는다.

모든 디스크는 다른 I/O 엘리먼트에 있는 디스크와 미러링(mirroring) 되어 있다. 커널은 디스크로 데이타를 write할 때에는 두 개의 미러드 디스크(mirrored disks)에 모두 write하며, read 시에는 그 두 개의 미러드 디스크로 부터 각각 반 씩 read함으로써 부하를 균등 분배하고 read bandwidth를 높인다. 또한, 모든 디스크는 듀얼 포트(dual-port)로 구현되어 있다. 즉, 각 디스크에는 항상 두 개의 억세스 패스(path)가 존재한다. 만약, 콘트롤러나 I/O 엘리먼트에 결함이 발생하여 디스크 억세스가 실패하면, 다른 패스를 통해 억세스를 시도하고, 이것도 실패하면, 미러링된 다른 디스크를 사용한다.

모든 통신선(communication line)도 서로 다른 I/O 엘리먼트에 연결된 두 개의 콘트롤러를 통해 억세스가 가능하게 구현되어 있어서, 콘트롤러나 I/O 엘리먼트의 결함 시에는 다른 쪽 패스를 통한 통신이 가능하게 되어 있다.

V. 결함 복구

결함 복구 시스템의 기본적인 목적은 시스템 상에 충분한 자원(resource)이 있다는 가정하에서, 운영체제로 하여금 모든 단일 결함(single fault)로 부터 복구가 가능 하게 하는 것이다. 예를 들어, 여러 개의 메모리 엘리먼트 중 하나가 잘못되더라도 운영 체제는 응용 프로그램에 영향을 주지않고 복구를 할 수 있어야 한다. 그러나, 하나의 메모리 엘리먼트만 있는 시스템에서 그것이 잘못된다면 복구할 수가 없다.

또한, 결함 복구 시스템은 결함의 종류에 따라 다수의 결함(multiple faults)로 부터도 복구가 가능하여야 한다. 예를 들면, 전원 공급상의 문제는 시스템 입장에서는 다수의 결함으로 처리되며, 이것 역시 복구가 가능해야 한다. 그러나, 다수의 결함으로 인해 복구가 불가능한 상황도 있다. 이러한 상황이 되면, 시스템은 항상 이것을 인식할 수 있어야 하고 재시동(rebooting)을 해야 한다.

결함 복구은 그림 4에 나타난 것같이 세 가지 단계로 이루어 진다. 먼저, 결함이 발생한다. 결함의 종류에 따라서는 결함이 탐지될 때까지 시간이 걸릴 수가 있다. (특히, 잘 사용되지 않는 요소에서 결함이 발생할 경우) 결함이 탐지되면 (주로 하드웨어에 의해) 시스템은 결함 복구 상태(fault recovery state)가 되며, 모든 노력을 결함 복구에 집중한다. 즉, 모든 시스템 요소들을 진단하며, 결함이 발생한 요소는 시스템에서 분리시킨다. 이 단계에서는, 어떠한 사용자나 시스템 프로세스도 수행되지 않는다. 결함 복구 단계가 끝나면, 시스템은 시스템 내의 모든 요소들을 재구성(reorganization)한다. 예를 들어, 결함 복구 단계에서 메모리 결함으로 인해 언쉐도우(unshadow)된 writable 페이지들을 쉐도우 되게 시도한다. (만약, 이때 충분한 메모리 엘리먼트가 없다면 언쉐도우 상태로 사용한다.) 여기에서는 이러한 결함 복구를 위한 각 단계에 대해 설명한다.

1. 결함 탐지 단계

하드웨어 결함은 종류에 따라 탐지되는 방법이 다르다. 먼저, 메모리 엘리먼트나 I/O 엘리먼트에 결함이 발생하면, 이들은 에러 상태(error state)로 들어가며 결함이 해소될 때까지 더이상 외부로 부터의 어떠한 요청 (request) 에도 응답 (response) 하지 않는다. 프로세서가 이들을 억세스 하려고 하면 워치닥 타이머 (watchdog timer) 에 의해 에러가 발생되고, 이때 이들의 결함이 탐지된다.

각 프로세서는 주 메모리 상의 정해진 곳에 자신이 동작하는 상태를 일정한 시간마다 기록하게 되어 있으며, 하나의 프로세서를 지정하여 다른 프로세서들의 동작 상태를 주기적으로 폴링(polling)하게 한다. 다른 프로세서들은 지정된 프로세서의 상태를 주기적으로 폴링한다. 만약, 어떤 프로세서든지 비 정상적인 상태의 프로세서를 발견하면 그 프로세서에 결함이 발생한 것으로 간주하고 시스템을 결함 복구 상태로 만든다.

하드웨어 결함을 발견한 프로세서는 즉시 다른 모든 프로세서들에게 우선권이 높은 인터럽트 (high-priority interrupt)로써 알린다. 모든 프로세서는 버스 인터페이스의 공유 레지스터를 이용하여 다음 단계인 결함 복구 단계로 들어갈 것을 동의한다.

2. 결함 복구 단계

일단 결함 복구 단계로 들어 오면 시스템이 불안정한 상태이므로, 시스템은 단일 프로세서 모드 (single-processor mode)가 된다. 즉, 한 번에 하나의 프로세서 씩 결함 복구를 위한 코드(recovery code)를 수행하며, executive 버스만을 사용한다. 각 프로세서가 수행하는 일의 순서는 다음과 같다: (1)커널 메모리를 찾는다; (2)결함 요인을 분석한다; (3)진행중인 캐쉬 플러쉬가 있으면 완료한다. 이들을 각 스텝 별로 자세히 설명하면 아래와 같다

(1) 어떠한 단일 결함이 발생해도 각 프로세서는 복구를 위한 커널 코드를 수행할 수 있어야 한다. 커널 코드는 메모리 상에 쉐도우되어 있고 모든 프로세서는 자신의 PROM에 이것에 대한 정보를 가지기 때문에, 메모리 엘리먼트의 결함 시에도 커널 코드를 억세스 할 수 있다.

(2) 각 프로세서는 진단 프로그램(diagnostic program)을 수행하여 결함 요인을 분석하고, 만약에 결함 요인으로 판단이 되는 것이 있으면 이것을 고발(indictment)한다. 나중에 모든 프로세서들로부터 고발된 요인들이 모여지면 판결(conviction)이 이루어진다. 각 프로세서는 자기자신의 에러와 다른 모듈(module)의 에러를 구분할 수 없으므로, 혼자 판결을 할 수가 없다. 예를 들면, 메모리로부터 응답이 없을 경우, 이것이 프로세서 자신의 버스 출력 부분에 문제가 있는지 혹은 메모리에 문제가 있는 지를 판단할 수 없다. 그러므로, 모든 프로세서들의 고발된 내용을 보고 난후에 결함 요인을 판단해야 한다.

(3) 각 프로세서는 언제든지 결함 발생에 대한 인트랍트를 받을 수가 있다. 만약, 캐쉬 플러쉬 중에 인터랍트가 발생했었다면 이 때 플러쉬를 완료한다.

3. 재구성 단계

모든 프로세서가 위의 세 가지 스텝을 모두 거치면, 시스템은 재구성 단계로 들어간다. 이 단계는 "executive" 라고 불리는 프로세서에 의해 수행되며, 또한 모든 프로세서는 executive 프로세서가 될 수 있다.

먼저, executive 프로세서는 시스템 콜이 진행 중이었던 프로세스들을 모두 찾아서, 그들의 시스템 콜 수행을 완료한다. 엄밀히 말하면, 결함 발생 당시 프로세스들이 가지고 있던 모든 락을 풀어 준다. 이를 위해, 그러한 프로세스들을 순차적으로(sequentially) 락을 풀 때까지 수행하면 된다. 이와 같이, 프로세스들을 순차적으로 수행할 때 락에 대한 처리에 특히 주의해야 한다. 단일 프로세서 모드이므로 수행 중인 프로세스가 어떠한 락을 요청할 때, 그 락이 이미 사용 중이면 락이 풀릴 때까지 기다리지 않고(더 이상 락을 풀어 줄 프로세서가 없기 때문임), 그 프로세스를 블럭킹(blocking) 시키고 다른 프로세스를 수행 해야 한다.

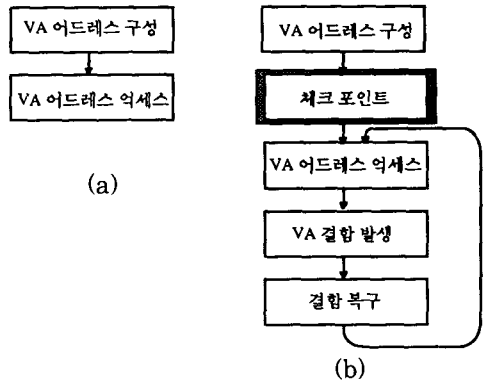

그림 5. 결함 임계 구역

모든 락을 풀고 나서, executive 프로세서는 다음과 같이 결함 요인에 대한 판결을 내린다. 만약, 모든 프로세서들의 고발이 한 모듈로 교차(intersection)가 되면, 그 모듈을 결함 요인으로 판결한다. 만약 그 결함이 영구 결함 (hard fault) 일 경우, 그 모듈을 시스템에서 제거시키고, 일시적 결함 (transient fault) 으로 판명될 경우, executive 프로세서는 그 사실을 기록해 두고 더 이상 결함 모듈에 대한 판결은 내리지 않는다. 일시적 결함이라 하더라도, 그 결함이 짧은 시간동안 많이 발생했다면, 그것을 영구 결함으로 간주한다.

만약, 다수의 모듈이 고발되었을 경우, 고발에 대한 정보들을 기록하고 판결은 내리지 않는다. 그리고, 고발된 모든 모듈들을 하나씩 시스템에서 오프라인 (off line) 시키고, 시스템을 체크한다. 이때 비슷한 결함이 다시 발생되면, 그 모듈은 정상이라고 판단하여 다시 온 라인시키고, 다른 모듈에 대해 시도를 한다. 만약 더 이상 결함이 발생되지 않으면, 오프 라인된 모듈을 결함 요인으로 판결하고 시스템에서 제거한다. 이렇게 함으로써, 모든 단일 결함에 대한 판결을 내릴 수 있다.

결함 진단이 모두 끝나면, executive 프로세서는 메모리 상의 하드웨어 구성 테이블(hardware configuration table)을 조정하고, 모든 프로세서들에게 정상적인 동작을 하도록 신호(signal)를 보낸다. 이러한 모든 결함 복구 작업에 소요되는 시간은 보통 1, 2 초 정도이다.

4. 결함 임계 구역

결함 복구가 완료되면, 결함 발생 시에 수행되던 모든 프로세스들는 자신들의 가장 최근의 체크 포인트로부터 다시 수행된다. 그러나, 결함이 발생하기 이전과의 하드웨어 구성이 바뀔 수 있으므로, 코드(code) 내의 하드웨어와 관련된 특정한 구역(region)들은 하나의 단위로 묶어서, 시스템 재개 시에 그 구역 중간부터 수행이 되는 일이 안 생기도록 해야 할 필요가 있다. 이러한 구역을 "결함 임계 구역 (fault critical region)"이라고 부른다.

예를 들어, 그림 5(a)에 나타난 코드를 보면, 먼저 VME 어뎁터(VA)의 어드레스를 구성하고, 구성된 VA 어드레스를 억세스 한다. 이들 두 오퍼레이션 사이에서 체크 포인트가 일어나고, 그 VA에 결함이 발생했다고 가정하자. 결함 복구 작업을 통해 그 VA는 시스템에서 오프 라인되고, 결함 복구가 끝나면 시스템은 그림 5(b)와 같이 최근의 체크 포인트부터 재개된다. 즉, 이미 존재하지 않는 VA를 억세스 하려고 한다. VA로부터의 응답이 없으므로 다시 결함 모드로 들어 가게 되고, 따라서 시스템은 무한 루프(infinite loop)에 빠지게 된다.

이러한 문제를 방지하기 위해서는, 임계 구역 내에서는 캐쉬 플러쉬가 일어나지 않게 해야 한다. 이를 위해, 임계 구역 내에서는 컨텍 스위치를 허용치 않고, 또한 캐쉬 오버 플로우(cache overflow)로 인한 플러쉬를 막기 위해 캐쉬 미스(cache miss)를 특별한 방법으로 처리한다. 이러한 결함 임계 구역들은 보통 몇 개의 명령어 정도로 짧으며, 이를 위한 오버헤드(overhead)는 카운터(counter)를 증가 시키는 정도이다.

Ⅵ. 결 론

본 논문에서는, 최근 Samsung에서 개발한 밀 결합 멀티 프로세서 구조의 무정지형 결함 허용 컴퓨터 시스템인 SSM6000을 소개하고, 그 시스템이 어떻게 하드웨어로 결함을 탐지하고 결함을 고립시키며, 운영 체제가 어떻게 결함으로 부터 복구하는 지에 대하여 설명하였다. 결함 탐지를 위한 장치로서 에러 탐지 코드, 자기 체크, 및 프로토콜 감시 등의 방법을 사용하였고, 결함 복구를 위해 시스템의 모든 엘리먼트들을 철저히 이중화하였다. Test-and-set 락과 non-write-through 캐쉬의 플러쉬를 콘트롤함으로써, 모든 단일 결함으로 부터 복구가 가능함을 보였다.

여기에서 소개한 구조에서 결함 복구 기능을 위한 비용은 (특히, 소 결합 구조에 비해) 크지 않다고 생각한다. 대부분의 이중화된 하드웨어들은 결함이 발생하지 않은 정상 상태에서는 시스템의 성능 향상에 기여를 한다. 예를 들면, 이중 버스는 서로 독립적으로 동작을 하여 데이타 전송 능력을 배가시키며, 미러드 디스크를 통해 read bandwidth를 높인다. 또한, Stratus 시스템이나 Tandem의 3중화 구조를 사용한 시스템 등과 같이 하드웨어에 의해 자동으로 복구가 되는 시스템들이 하드웨어를 3 내지 4중화 하는 것에 반해, SSM6000은 결함 복구를 운영 체제가 하게 함으로써 하드웨어 이중화만으로도 완벽한 결함 허용 기능을 가진다.

그러나, SSM6000과 같은 밀 결합 구조에서는 모든 프로세서가 커널 데이타를 공유하기 때문에 소프트웨어 버그(bug)로 인해 커널의 데이타들이 잘못 사용되면, 모든 프로세서에 결함을 유발한다. 그러므로, 이러한 밀 결합 구조에서는 시스템의 신뢰성에 대해 운영 체제가 결정적인 영향을 미치므로, 운영 체제 디자인 시에 각별한 주의를 해야한다.

지금까지는 컴퓨터 시스템의 가치가 주로 성능이나, 가격, 그리고 전력 소모 등으로 판단되었으나, 앞으로는 신뢰성이나 결함 복구 기능 등이 비교 기준이 될 것이다. 그러므로, 결함 허용 컴퓨터의 응용 범위는 계속 증가 할 것이며, 따라서 이에 대한 많은 연구가 필요하다고 하겠다. 예를 들어, 밀 결합과 소 결합 구조의 장점 들을 살린 새로운 복합 구조등이 좋은 연구 과제라고 본다. 또한, 현재의 SSM6000은 프로세서 및 메모리 보드들이 4 개 정도인 소규모 시스템이므로 시스템 버스가 성능에 별 영향을 미치지 않고 있으나, 이들의 수가 수 십개 이상으로 늘어나고, 또한 훨씬 고성능의 마이크로 프로세서를 사용하

게 된다면 현재의 시스템 버스는 bottleneck이 될 것이다. 따라서, 이러한 대규모 시스템에 사용될 고성능 버스에 대한 연구와 더불어 이들 다중 버스들을 메모리 영역에 따라 지정된 버스만을 사용하는 static한 방법이 아니라, dynamic하게 억세스할 수 있는 방법에 대한 연구 등이 현재 Samsung에서 진행되고 있다.

參 考 文 獻

[1] D. P. Siewiorek, "Architecture of fault-tolerant computers : an historical perspective," *Proc. IEEE*, vol. 79, no. 12, pp. 1710-1734, Dec. 1991.

[2] D. P. Siewiorek, "Fault tolerance in commercial computers," *IEEE Computer*, pp. 26-37, July 1990.

[3] S. Hariri, A. Choudhary, and B. Sarikaya, "Architectural support for designing fault-tolerant open distributed systems," *IEEE Computer*, pp. 50-62, June 1992.

[4] R. A. Maxion, D. P. Siewiorek, and S. A. Elkind, "Techniques and architectures for fault-tolerant computing," *Annual Review of Computer Science*, vol.2, pp. 469-520, Annual Reviews Inc., 1987.

[5] S. Webber, "The Stratus architecture," in *Reliable Computer Systems: Design and Evaluation*, Bedford, MA: Digital Press, 1992.

[6] P. A. Bernstein, "Sequoia: a fault-tolerant tightly coupled multiprocessor for transaction processing," *IEEE Computer*, pp. 37-45, Feb. 1988.

[7] C.-H. Lee, D. Lee, and M. Kim, "Optimal task assignment in linear array networks," *IEEE Trans. on Computers*, vol. C-41, no. 7, p. 877-880, July 1992.

[8] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, 1978.

[9] Y. C. Chow and W. H. Kohler, "Models of dynamic load balancing in a heterogeneous multiple processor system," *IEEE Trans. on Computers*, pp. 354-361, May 1984.

[10] A. J. Smith, "Cache memories," *ACM Computing Surveys*, pp. 473-530, Sep. 1882.

[11] C.-H. Lee, "Topix: Theory of Operation," *SEC Tech. Report*, S40-OSTPX-PC, 1992저자 소개 : (저자 소개 참조)

筆者紹介

李 鐵 勳

1960年 4月 8日生
1983年 2月 서울대학교 공과대학 전자공학과 졸업(학사)
1988年 2月 한국과학기술원 전기 및 전자공학과 졸업(석사)
1992年 2月 한국과학기술원 전기 및 전자공학과 졸업(박사)

1983年 1月 ~ 1986年 1月 삼성전자 종합연구소 연구원
1992年 3月 ~ 현재 삼성전자 시스템개발실 선임연구원

주관심 분야 : 병렬 및 분산처리, 알고리즘, 그라프이론, 운영 체제, Fault-Tolerant Computing

劉 昇 和

1949年 4月 23日生
1972年 2月 서울대학교 공과대학 응용수학과 졸업(학사)
1980年 6月 University of Kansas 전산공학 졸업 (석사)
1983年 6月 University of Kansas 전산공학 졸업 (박사)

1974年 6月 ~ 1976年 9月 한국과학기술 연구소 연구원
1976年 10月 ~ 1978年 7月 금성통신 ESS 소프트웨어 연구실장
1983年 5月 ~ 1988年 7月 AT&T Bell 연구소 연구원
1989年 9月 ~ 현재 삼성전자 상무

주관심 분야 : 컴퓨터 구조, 컴퓨터 네트워크, 분산 시스템, Performance Evaluation