CTS: 예방 정비를 위한 클러스터 시스템 검사 도구

(CTS: A Cluster System Test Suite for Preventive Maintenance)

차 광 호 †

(Kwangho Cha)

요 약 현재 클러스터 시스템은 여러 분야의 문제들을 위하여 폭 넓게 이용되어지고 있으며 유용한 고성능 컴퓨팅 자원으로 인식되고 있다. 클러스터 시스템의 사용자가 늘어남에 따라 클러스터 시스템의 성능 개선 못지 않게 안정적인 운영을 유지하는 것도 중요한 상황이다. 하드웨어 예방 정비가 정상 운영을 위해서 중요한 것임에도 불구하고, 예방 점검 시간에 일반적인 클러스터 시스템을 위하여 사용될 수 있는 검사 도구는 주요 관심사가 되지 못했다. 본 논문에서는 하드웨어 예방 정비를 고려하여 클러스터 시스템을 위한 하드웨어 검사 도구를 제안한다. CTS(Cluster system Test Suite)로 명명된 클러스터 시스템 검사 도구는 메모리와 NIC를 점검하기 위한 두개의 검사 루틴을 가지고 있다. CTS를 설계시, CTS가 일반적인 클러스터 시스템이 가지는 공통된 특징을 지원하도록 노력하였으며 검사 조건 설정에서 결과 조회까지 모든 작업은 통합 GUI환경에서 진행될 수 있도록 하였다. 두 종류의 클러스터 시스템을 점검할 때, CTS를 사용하였고 클러스터 시스템을 관리하는데 유용한 정보가 제공됨을 확인하였다.

키워드 : 클러스터 시스템, 예방 정비, 하드웨어 검사

Abstract Cluster systems have been widely used for solving problems in various application domains, and regarded as useful high performance computing resources. As the number of cluster system user is increasing, it is no less important to maintain stable operation than to improve cluster system performance. Although hardware preventive maintenance is important for keeping normal operation, the testing tool which can be used for general cluster systems during maintenance has received little attention. In this paper, considering hardware preventive maintenance, we suggest a testing tool for hardware of cluster system. The cluster system testing tool which is named CTS(Cluster system Test Suite) has two check routines; one for memory, and the other for NIC respectively. The CTS is designed to support the common features of general cluster systems and all the jobs such as setting test conditions to querying the results can be done entirely within an integrated GUI environment. CTS is used as the testing tool for two kinds of cluster systems during maintenance, and the experimental results show that CTS reports useful information for cluster systems management.

Key words : Cluster System, Preventive Maintenance, Hardware Test

1. 서 론

클러스터 시스템의 안정된 운영을 위해서는 시스템 구성 요소에 대한 예방 정비 작업이 중요하다. 그러나 현재까지 클러스터 시스템의 예방 정비를 위한 소프트웨어는 특정 시스템에 국한되어 있어서 보급이 미비한 상황이다. 본 연구에서는 클러스터 시스템을 대상으로 한 예방 정비 도구인 CTS(Cluster system Test Suite)를 구현하고 이를 실제 클러스터 시스템에 적용한 결과를 설명하고자 한다.

특정 과학계산 분야에서 실험적인 성격의 시스템으로 간주되던 클러스터 시스템이 점차 확산되면서 최근 그 응용 분야를 넓혀가고 있다. 시스템을 구성하는 단위 노드의 수도 증가하여 규모나 성능 면에서 급격한 발전을 보이고 있다. 전 세계 슈퍼컴퓨터의 순위 정보를 제공하는 Top 500 리스트에서 클러스터 시스템이 차지하는 비중이 증가하고 있는 점도 이와 같은 사실을 뒷받침하고 있다[1].

클러스터 시스템의 성능이나 규모에 못지않게 중요한

† 정 회 원 : 한국과학기술정보연구원 연구원
khocha@kisti.re.kr
논문접수 : 2004년 4월 16일
심사완료 : 2004년 8월 2일

것이 시스템의 안정된 운영이라 할 수 있다. 임의의 시스템에 대한 지속적이고 안정된 사용을 위해서 예방 정비가 중요시되고 있는데, 소프트웨어 공학 분야에서 필요성이 제기되어 개념이 정립된 후, 현재는 하드웨어 시스템에서도 하드웨어 예방 정비라는 개념이 적용되고 있다[2-4]. 하드웨어 예방 정비는 일정 주기마다 각 하드웨어 구성 요소의 이상 유무를 점검하여 이상이 발견되거나 발견될 소지가 있는 요소의 수리나 교체를 통하여 시스템 장애에 미리 대처하고자 하는 것으로[3] 각 요소의 이상 유무를 감지하는 기능이 중요하다. 특히, 클러스터 시스템의 단위 구성 요소인 개인용 컴퓨터에 대한 최근의 장애 분석 연구에 따르면 전체 장애 중 약 10%가 하드웨어에서 약 20%가 시스템 소프트웨어에서 기인한다고 보고 되었다[5,6]. 이는 시스템의 안정성에 하드웨어가 직간접적으로 영향을 미치고 있다는 점과 함께 하드웨어에 대한 주기적인 예방 정비의 필요성을 보여주고 있다.

클러스터 시스템의 확산이 Rock[7], OSCAR[8], SCore[9] 같은 관리 소프트웨어들의 발전을 가져왔으나 이들 대부분 소프트웨어들은 클러스터 시스템의 초기 설치 및 설정 기능에 중점을 두고 있을 뿐 예방 정비를 위해 필요한 시스템 구성 요소에 대한 이상 유무를 점검하는 기능은 포함하고 있지 않다. 물론 특정 하드웨어 제조사의 제품에 특화된 점검 프로그램들이 존재하지만, 특수 하드웨어 기능을 활용해야만 하는 제약으로 인하여 일반적인 하드웨어로 구성된 클러스터 시스템에는 공통적으로 적용하기 힘들다는 단점이 있다[10,11].

이와 같이 클러스터 시스템의 규모나 중요도는 증가하는 반면, 상용 슈퍼컴퓨터처럼 적당한 예방 점검용 도구나 기법이 존재하지 않으므로, 본 논문에서는 클러스터 시스템의 일반적인 특성을 활용하여 일부 하드웨어 구성 요소의 이상 유무를 판별하는 방안을 제시한다. 더 나아가서는 이를 바탕으로 한 클러스터 시스템용 하드웨어 검사 도구의 구현 및 사용 결과를 설명하고자 한다.

본 논문에서 소개하고자하는 CTS는 클러스터 시스템의 각 계산 노드의 하드웨어 구성 요소를 검사하기 위하여 개발되었으며 관리 서버에서 각 계산 노드의 메모리와 네트워크 접속 상태를 검사한 뒤, 전반적인 검사 결과를 관리자에게 제공하여 시스템의 이상 유무를 파악하는데 도움을 줄 수 있도록 하였다.

본 논문에서는 표 1과 같이 클러스터 시스템의 대표적인 운영체제라고 할 수 있는 리눅스를 탑재한 클러스터 시스템을 대상으로 하고 있다. 설계 및 구현을 진행하면서 주로 고려한 상황은 일반적인 리눅스 클러스터 시스템의 공통 요소를 반영하고, 가능한 검증 작업이 어느 정도 진행된 공개 소프트웨어를 분석·활용함으로서 검사의 신뢰성을 높이도록 하였다. 즉, 관리 서버에서 각 계산 노드에 대한 검사를 수행한 뒤, 결과를 취합하는 단계에서는 클러스터 시스템들이 주로 사용하는 네트워크 파일 시스템(NFS)를 이용하였으며, 메모리 점검과 네트워크의 접속 상태 점검을 위한 프로그램은 각각 공개 소프트웨어인 Memtester[12]와 MII-tools [13]를 참조하여 설계되었다. 각 노드의 세부사양이 상이한 64노드 클러스터 시스템과 128노드 클러스터 시스템에서 CTS를 사용하였고 관리자에게 유용한 정보가 제공될 수 있음을 확인하였다.

본 논문의 구성은 다음과 같다. 2장에서는 CTS를 설계하기에 앞서 참조한 기존 연구들에 대하여 설명하고, 3장에서는 CTS의 구조, 4장에서는 실제 운영 테스트 및 운영 결과, 끝으로 5장에서는 결론과 향후 계획에 대하여 설명한다.

2. 관련 연구

본 장에서는 클러스터 컴퓨터 검사 도구와 관련된 기존 검사 프로그램에 대해서 설명한다. 리눅스 기반 공개 소프트웨어로 배포되고 있는 단일 시스템용 검사 프로그램들을 비교 분석하였으며, 이를 바탕으로 적합한 검사 프로그램을 복수의 계산 노드에 적용할 수 있도록 수정 보완하여 개발하였다.

2.1 메모리 검사

클러스터 시스템이 특정 운영체제를 요구하는 것은 아니지만, 표 1과 같이 일반적으로 많은 시스템들이 유닉스나 리눅스를 운영체제로 사용하고 있으며, 본 연구에서는 리눅스용 메모리 검사 프로그램을 대상으로 비교 분석하였다. 우선 분석 대상으로 고려한 메모리 검사 도구중 하나가 Memtest86이다[15]. X86기반 아키텍쳐를 대상으로 하는 검사 프로그램으로 가장 큰 특징은 커널을 배제하고 검사를 수행한다는 점이다. 리눅스에서 검사 프로그램을 생성한 후, 재 부팅하면 커널 대신 검사 프로그램이 로딩되어 메모리 검사를 수행하는 구조이다. 커널의 간섭없이 직접 검사가 실행되어 엄밀한 검사를 할 수 있다는 장점이 있으나, 커널이 수행하는 하드웨어 제어 루틴을 내부에 포함하고 있어야하므로 하드웨어에 의존적인 도구이다. 현재는 X86 아키텍쳐용

표 1 Clusters@Top500[14]에 등재된 클러스터 시스템의 운영 체제별 분포

운영체제	Linux	Solaris	Tru64 UNIX	Windows	BSD	기타
비율	90.31%	3.1%	1.55%	1.16%	1.16%	2.71%

프로그램만 제공되고 있다.

Memtest86이 수행하는 메모리 검사 루틴과 유사한 기능을 가진 검사 도구로 Memtester가 있다. Memtest86이 커널을 배제하고 독립적으로 검사를 수행하는 반면, Memtester는 커널 상에서 동작하는 특징을 갖는다[12]. 커널 상에서 동작하는 사용자 응용 프로그램의 일종이므로 전체 메모리 검사에는 제약이 따르지만, X86기반 아키텍쳐에서만 동작한다는 Memtest86의 단점을 보완할 수 있다. 또한 커널이 제공하는 서비스를 그대로 받을 수 있다는 장점도 있다. 표 2는 Memtest86와 Memtester에서 각각 사용된 메모리 검사 루틴들의 종류를 보여주고 있다.

이상의 검사 프로그램들 이외에, 스크립트 수준의 간단한 프로그램들도 찾아 볼 수 있으나[16], 검사의 범위와 결과의 취합 방법등을 고려하여 Memtester를 변경하여 계산 노드의 메모리를 검사하도록 하였다. 즉, Memtest86이 보다 엄밀한 검사를 진행함이 사실이나, 검사 결과를 취합하는 과정에서 용이하지 못함과, Memtester를 클러스터 시스템의 진단 도구 중 일부로 사용하는 관련 연구를 고려하여 본 구현에서도 Memtester를 계산 노드에 대한 검사 모듈로 사용하였다[17-19]. Memtester를 사용함에 있어 기존 연구와의 차이점은 기존 연구에서는 Memtester에 대한 수정없이 그대로 사용하였으나, 본 CTS에서는 메모리에 대한 검사 영역을 늘리는 방법과 결과를 취합하는 과정에 적합하도록 Memtester를 수정하여 사용하였다.

2.2 네트워크 검사

클러스터 시스템의 성능 및 안정성에 크게 영향을 미치는 요소 중 하나가 네트워크이다. 물론 클러스터 시스템에 특화된 SAN(System Area Network)의 사용이 성능 면에서는 가장 좋은 방법이지만 비용 문제로 인하여 계산 네트워크와 관리 및 파일 서비스 네트워크를 분리하여 복수개의 이더넷을 이용하여 시스템을 구성하는 방법도 많이 적용되고 있다. 본 네트워크 검사에서는 이와 같이 복수개의 이더넷을 이용하여 시스템을 구성하는 경우에 각 계산 노드가 가지는 네트워크 인터페이스 카드의 접속 상태를 관리 서버에서 확인하는 것을 목적으로 하고 있다.

네트워크 인터페이스 카드의 상태를 파악할 수 있는 효과적인 방법 중 하나가 네트워크 인터페이스 카드에 존재하는 MII(Media Independent Interface) 레지스터의 상태 정보를 확인하는 것이다. 이를 위한 유틸리티로는 MII-diag나 MII-tool등이 있는데 단점은 호환성이 떨어지는 네트워크 인터페이스 카드의 경우 MII 레지스터의 기능을 지원하지 않는 점과 네트워크 인터페이스 카드의 종류에 따라 연결 상태 이외의 정보가 정확하지 않을 수 있다는 점이다[13].

이러한 상황을 고려하여 본 논문에서는 호환성이 높은 네트워크 인터페이스 카드를 주요 대상으로 하였으며, 가장 중요하면서도 비교적 정확한 정보인 '연결 상태'만을 MII 레지스터로부터 추출하여 사용하였다. 즉, 서버는 각 계산 노드에 설치된 네트워크 인터페이스 카드들의 '연결 상태' 정보를 취합하여 네트워크 구성 상태를 파악하게 된다.

3. 설계 및 구현

일반적인 클러스터 시스템 형태인 베어울프형 클러스터 시스템이 갖는 특징 중 하나는 각각의 계산 노드들이 각자 자신의 운영체제를 가지고 있다는 점이다. 이는 계산 노드의 검사를 위하여 앞서 설명된 단일 시스템용

표 2 메모리 검사 루틴의 종류

Memtest86	Memtester
° Address test , walking ones(nc) ° Moving Inversions , ones&zeros(c) ° Address test , own address(nc) ° Moving Inversions , 8 bit pattern(c) ° Moving Inversions , 32 bit pattern(c) ° Block move , 64 moves(c) ° Modulo 20 , ones&zeros(c) ° Moving Inversions , ones&zeros(nc) ° Block move , 512 moves(c) ° Moving Inversions , 8 bit pattern(nc) ° Modulo 20 , 8 bit(c) ° Moving Inversions , 32 bit pattern(nc) ※ c = cached, nc = no cache	° Stuck Address ° Random value ° XOR comparison ° SUB comparison ° MUL comparison ° DIV comparison ° OR comparison ° AND comparison ° Sequential Increment ° Solid Bits ° Block Sequential ° Checkerboard ° Bit Spread ° Bit Flip ° Walking Ones ° Walking Zeroes

검사 도구들을 적절히 사용할 수도 있다는 의미가 된다. 그러나 계산 노드 수가 증가할 경우, 검사의 실행과 결과 취합에 제약이 따르게 되므로 클러스터 시스템과 같은 일종의 분산 환경을 고려하여 수정하여야 한다.

검사 도구를 설계하면서 고려한 주요 기능은, 관리 서버와 같은 특정 서버 시스템에서 다수의 계산 노드에 대한 자동화된 검사 수행 및 결과 취합이었다. 또한 특정 시스템에 국한되는 구성을 배제하고 호환성을 유지하기 위하여, 기존의 공개 소프트웨어들을 수정하여 검사 도구를 구성하는 방향으로 설계를 진행하였으며, 클러스터 시스템에서 공통적으로 사용되는 네트워크 파일 시스템(NFS)을 결과 파일의 취합을 위한 중간 매체로 사용하였다.

3.1 기본 기능

검사 도구의 개략적인 구조는 그림 1과 같다. 관리자에 의하여 검사에 사용될 인자가 설정되어지면 검사 실행용 스크립트를 이용하여 해당 검사를 원격지인 각 계산 노드에서 수행하게 된다. 그 후 검사 결과는 데이타베이스로 집계되어지며 사용자 인터페이스를 통하여 사용자는 결과를 확인하게 된다. 사용자 인터페이스는 파이선 스크립트와 파이선용 그래픽 라이브러리인 wxPython을 이용하여 구현하였고 그림 2와 3은 사용자 인터페이스의 예를 보여주고 있다.

특히 그림 3은 검사를 위한 각 프로그램의 위치 정보와 검사에 필요한 인자를 설정하고 검사를 수행하는데 필요한 인터페이스를 보여주고 있다. CTS가 실행되면

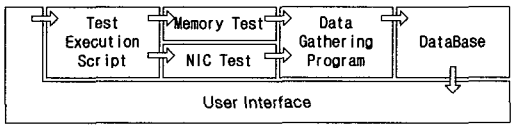

그림 1 CTS의 구조

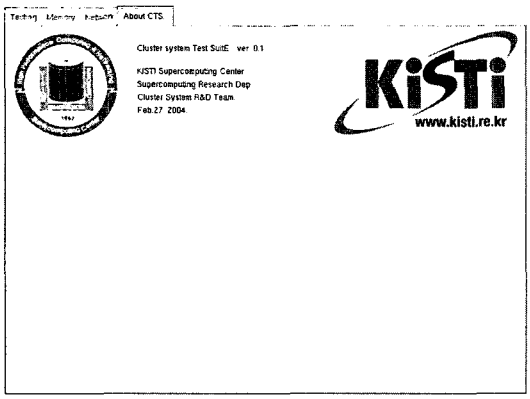

그림 2 CTS의 초기 화면

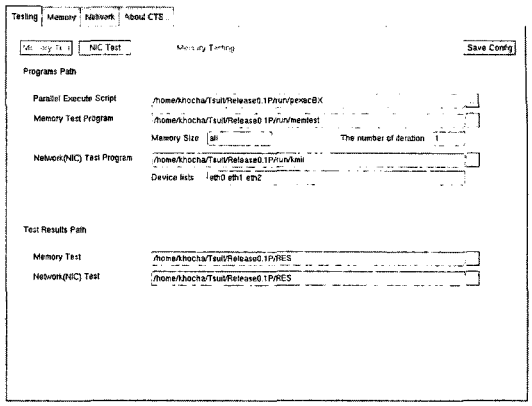

그림 3 검사 프로그램을 위한 인자 설정 화면

아래 그림 4와 같은 내용의 setup.inf 파일을 읽어 검사를 위한 환경 정보와 검사 프로그램을 위한 기본 인자 설정을 진행하게 되며 사용자는 그림 3의 인터페이스를 이용하여 필요한 값으로 설정을 변경한 후 검사를 수행할 수 있다.

```
# This is a configuration file for CTS.

# The location of Programs
Parallel Exe  = /home/khocha/Tsuit/Release0.1P/run/pexecBX
Memory Test   = /home/khocha/Tsuit/Release0.1P/run/memtest
Memory Size   = all
No. of Loop   = 1

Nic Test      = /home/khocha/Tsuit/Release0.1P/run/kmii
Nic Lists     = eth0 eth1 eth2

# The locations of result files.
Memory Result = /home/khocha/Tsuit/Release0.1P/RES
Nic Result    = /home/khocha/Tsuit/Release0.1P/RES
```

그림 4 setup.inf 파일

환경 정보 중에는 검사를 각 계산 노드에서 수행시키기 위해 어떤 종류의 병렬 수행 스크립트를 사용할지에 대한 내용이 포함되어야 하는데, 공개 스크립트인 PTOOLS[20]의 pexec를 수정하여 사용하거나 IBM의 클러스터 시스템 관리 도구인 xCAT[21]중의 psh을 활용할 수 있다. 이상의 과정을 거쳐 검사 준비가 완료되면 그림 5와 같은 각 단계를 수행하게 되며 각 단계별 수행 내용은 다음과 같다.

① 네트워크 파일 시스템에 검사 프로그램을 위치시키고 검사에 필요한 인자를 설정한다.

② 병렬 수행 스크립트를 이용하여 각 노드에서 검사 프로그램을 수행한다.

③ 각 노드에서 수행된 검사의 결과는 임시 파일의 형태로 공유 디렉토리에 저장된다. 단일 파일에 모든 계산 노드가 결과를 기록할 경우에는 부분적인 손실이 발생할 수 있으므로, 각 계산 노드별로 결과 파일을 생성하도록 하였다.

④ 결과 취합 프로그램에 의해서 결과 파일들이 데이타베이스로 저장되며, 검사 결과는 데이타베이스를 조회하여 확인한다.

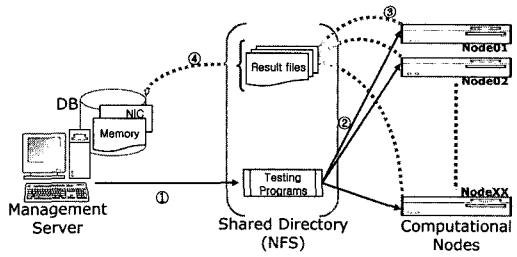

그림 5 검사 수행 과정

이상의 과정과 같이 검사의 최종 결과는 데이타베이스로 취합되며, 이에 필요한 데이타베이스는 MySQL을 이용하여 구성하였다[22]. 다음의 표 3과 4는 각 검사 결과를 저장하는 테이블의 구조를 보여주고 있다.

3.2 테스트 프로그램의 구현

CTS의 구성 요소 중, 중요 요소인 테스트 프로그램들은 기존에 발표된 오픈 소스 기반의 테스트 프로그램을 수정하여 개발하였다. 본 절에서는 각 테스트 프로그램에 수정된 부분과 그 특징을 설명한다.

3.2.1 메모리 테스트 프로그램

메모리 테스트에 있어서는 앞서 설명된 것처럼 Memtester를 수정하여 사용하였다. 기존의 Memtester는 다량의 테스트 로그가 각 단계별로 사용자 화면과 파일로 출력되는 방식을 취하고 있다. 그러나 다수의 계산 노드에서 동시에 테스트를 수행하고 결과를 취합하는데 이 방법은 한계가 있으므로 최종 테스트 결과를 요약해서 임시 파일로 출력하게 하였다.

결과 출력 방식의 수정과 더불어 Memtester에 추가된 또 하나의 특징은 테스트 메모리 영역의 확장이다. 기존의 Memtester는 리눅스 커널이 갖는 특징으로 인하여 테스트를 수행하는 메모리의 크기에 제약이 있다. 그림 6은 Memtester가 테스트를 수행하는 과정에서 메모리를 확보하는 과정을 슈도 코드로 표현한 것이다. 슈도 코드에서 보는 바와 같이 Memtester는 메모리 확보와 확보된 메모리의 페이징을 금지하기 위하여 malloc()과 mlock()함수를 각각 사용한다. 즉, mlock()을 시도하여 성공적으로 페이징을 금지할 수 있는 영역에 한해서

```
...
mem_avail = get available memory size;
mem_allocate = malloc(mem_avail);
while(mlock(mem_allocate)==ERROR){
    free(mem_allocate);
    mem_avail--;
    mem_allocate = malloc(mem_avail);
}
...
memory_test(mem_allocate);
...
```

그림 6 Memtester의 메모리 확보 과정

표 3 메모리 검사 결과용 테이블

필 드 명	설 명	유 형	NULL	Key	Default
host_id	계산 노드의 호스트명	char(128)		PRI	
n_of_test	메모리 검사 반복 횟수	int(11)	YES		NULL
n_of_error	에러 발견 횟수	int(11)	YES		NULL
exe_time	실행 소요 시간	float	YES		NULL
mem_size	검사된 메모리 크기	int(11)	YES		NULL
Tdate	검사 실행일	char(10)	YES		NULL
Ttime	검사 실행시각	char(8)	YES		NULL

표 4 네트워크 인터페이스 검사 결과용 테이블

필 드 명	설 명	유 형	NULL	Key	Default
host_id	계산 노드의 호스트명	char(128)		PRI	
device	네트워크 디바이스	char(8)		PRI	
Tdate	검사 실행일	char(10)	YES		NULL
Ttime	검사 실행시각	char(8)	YES		NULL
log	검사 결과 로그	char(100)	YES		NULL

만 메모리 테스트를 수행하게 되며 이는 가상 메모리가 아닌 실제 메모리를 테스트하기 위한 조치인 것이다. 그러나 버전 2.6 이전의 리눅스 커널들은 mlock()을 통해서는 실제 메모리의 1/2 이상 크기의 메모리에 대한 잠금을 수행할 수 없게 되어 있다. 전체 시스템을 이상 상태로 빠지게 하는 것을 방지하기 위한 임시적인 방편이라고 되어 있으나 사용자들의 요구로 인하여 버전 2.6에서는 이 제약이 풀린 상황이다.

이와 같은 이유로 최신 리눅스 버전인 2.6 커널을 사용하지 않는 시스템에서 가용 메모리의 크기가 실제 메모리의 1/2보다 큰 경우에는 가용 메모리의 일부분만을 테스트하게 된다. 이를 해결하기 위해서는 커널의 'mlock.c' 파일을 수정한 후 패치하거나 2.6 버전의 커널로 업데이트하는 방안을 고려할 수 있다. 그러나 CTS를 사용하기 위하여 커널 패치나 운영체제의 업데이트를 요구하는 것은 시스템 관리자에게 부담으로 작용될 수 있고 또한 현재 2.6 커널에 여러가지 문제가 제기되는 상황이어서 이러한 시스템 소프트웨어에 대한 변경은 배제하였다. 이러한 이유로 모든 버전의 리눅스 커널에서 가용 메모리를 제약 없이 테스트하도록 기존의 Memtester를 수정하였다.

임의의 시점에서 가용 메모리의 최대 크기는 실제 메모리에서 커널이 사용한 메모리를 제외한 크기가 된다. 이는 항상 가용 메모리의 절반 크기는 실제 메모리의 1/2보다 작다는 의미가 되므로, 테스트하고자 하는 메모리를 두개의 영역으로 나누어 각각 mlock()을 요청하고 테스트를 수행하면 앞서 언급된 문제를 해결할 수 있다. 이때 대부분의 메모리 영역에 대한 페이징이 금지되므로 갑자기 다량의 새로운 프로세스들이 생성되는 경우 문제가 될 수도 있으나, 앞서 언급한바와 같이 CTS의 용도는 모든 사용자 서비스가 중지된 상태에서 관리자에 의한 시스템 점검 시간에 사용하는 것을 전제로 하기 때문에 시스템에 큰 문제를 발생시키지 않는다. 그림 7은 수정된 Memtester의 메모리 할당 과정을 보여 준다. 그림 7과 같이 자식 프로세스를 생성하여 부모와 자식 프로세스가 각각 가용 메모리의 1/2 크기를 테스트 후, 테스트 결과를 취합하도록 하였다.

표 5는 기존의 Memtester와 수정된 Memtester가 테스트를 수행한 메모리의 최대 크기를 보여주고 있으며, CTS를 위하여 수정된 Memtester가 더 큰 메모리 영역

```
                …
mem_avail = get available memory size;
fork a child process;
mem_avail = mem_avail / 2;
mem_allocate = malloc(mem_avail);
while(mlock(mem_allocate)==ERROR){
      free(mem_allocate);
      mem_avail--;
      mem_allocate = malloc(mem_avail);
}
                …
memory_test(mem_allocate);
                …
gather the result of two process;
```

그림 7 수정된 Memtester의 메모리 확보 과정

을 테스트함을 확인할 수 있다.

3.2.2 네트워크 테스트 프로그램

네크워크 인터페이스의 MII 장치의 상태를 체크하고 설정하는데, mii-tool가 사용되고 있다. 본 논문에서는 연결 상태 확인에 중점을 두고 있으므로 mii-tool에서 '연결 상태' 정보를 가져오는 루틴을 제외한 MII register를 조작하는 다른 루틴은 제거하여 최소 규모의 축약된 프로그램으로 수정하였다. 축약된 네트워크 테스트 프로그램은 '연결 상태' 정보를 가져오기 위하여 socket()와 ioctl() 함수를 사용하고 있다. 수정된 네트워크 테스트 프로그램의 수행 절차는 다음과 같다.

```
1: PHY와 MII레지스터 정보 획득용 소켓 생성
2: 대상 디바이스의 PHY 점검
3: MII 레지스터 정보 획득(연결 상태 정보 추출)
4: MII 레지스터 정보 분석 및 출력
```

그림 8 네트워크 테스트 프로그램의 수행 절차

3.3 동질적인 클러스터 시스템을 고려한 기능

앞서 언급한 것처럼 CTS는 특별한 하드웨어에 종속적인 프로그램이 아니라 사용자 응용 수준의 프로그램들이므로 경우에 따라 정확한 검사에 제약이 따를 수 있다. 이러한 약점을 보완하기 위하여 클러스터 시스템에 동질성이 존재할 경우, 이를 이용하는 방법을 고려하였다. 클러스터 시스템을 구성함에 있어서 절대적인 사항은 아니지만, 많은 시스템이 동질적인 성격의 계산 노드로 구성되어지고 있다. 이는 동일한 환경에서 검사를

표 5 메모리 테스트에 사용된 메모리의 최대 크기 비교

실제메모리 (bytes)	커널이 사용한메모리 (bytes)	테스트된 메모리 크기 (bytes)	
		기존의 Memtester	수정된 Memtester
526135296	195039232	268017664	317431808
3173392384	146812928	1609072640	2969591808

수행하게 되는 것을 의미하므로 클러스터 시스템이 동일한 구성의 계산 노드로 구성되어진 경우, 각 계산 노드에서의 검사 실행시간이 유사해야 한다는 전제하에 계산 노드들의 검사 실행시간과 평균 실행시간을 비교하여 의심되는 계산 노드 정보를 제공하게 하였다. 이는 메모리 검사에 있어서 중점적으로 고려되었으며 검사 결과는 다음과 같이 3가지 경우로 구분된다.

- 검사 도중 이상이 발견되지 않은 경우
- 검사 도중 이상이 발견된 경우
- 검사 도중 이상이 발견되지는 않았으나 의심 징후가 보이는 경우

검사 도중 이상이 발견되지 않은 경우는 일단 메모리 검사 프로그램이 특이한 이상을 발견하지 못한 상태로 다른 계산 노드의 검사 실행시간과 비교하여 큰 차이가 없으면 본 논문에서는 정상인 노드로 간주한다. 검사 도중 이상이 발견된 경우는 검사 중 이상이 발견되어 메모리에 대한 보다 정밀한 점검과 해당 메모리의 교체를 필요로 한다. 검사 중 이상이 발견되지는 않았으나 의심 징후가 보이는 경우는 검사 프로그램 자체는 이상을 발견하지 못하였으나 검사 소요 시간이 다른 계산 노드에 비해 상당한 차이를 보이는 것으로 이상이 있을 수 있다는 가정하에 해당 노드에 대한 상태 점검을 수행하여야 한다.

4. 적용 실험

CTS의 적용 가능성을 확인하기 위하여, 한국과학기술정보연구원에서 보유하고 있는 128노드 클러스터 시스템과 64노드 클러스터 시스템의 예방 점검 시, CTS를 이용한 시스템 검사를 수행하였다. 표 6는 각 클러스터 시스템의 구성을 보여 주고 있다. CTS의 개발 목적이 예방 점검 기간에 시스템 검사를 위하여 사용하는 것이므로 일반 사용자의 접근이 배제된 정기 점검 시간에 작업이 이루어졌다.

4.1 메모리 검사 결과

그림 9는 메모리 검사 결과에서 감지된 오류의 예를 보여주고 있다. 이 경우는 메모리 검사를 수행하는 과정에서 오류가 감지된 경우로서 CTS에서 오류가 보고된 계산 노드를 대상으로 실행한 추가 검사에서도 그림 10의 검사 결과처럼 오류가 감지되었으며 오류 횟수는 오

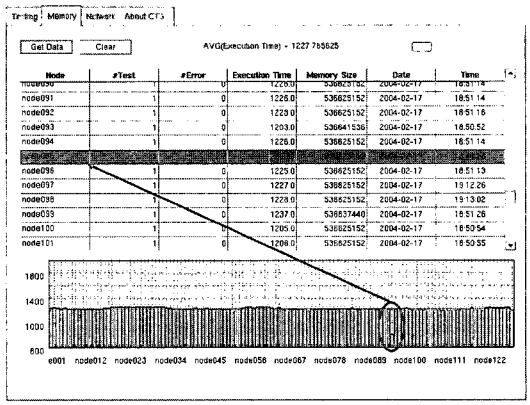

그림 9 128노드 클러스터 시스템에서의 메모리 검사 결과

```
Testing 536518656 bytes at 0x40301000 (4088 bytes lost to page alignment).

Run    1:
  Test  1:         Stuck Address:  Setting...Passed.
  Test  2:          Random value:  Setting...Testing...
FAILURE: 0x2e7e76a7 != 0x2e7c76a7 at offset 0x0007f56a.
Skipping to next test...
  Test  3:        XOR comparison:  Setting...Testing...
FAILURE: 0x5fc1e513 != 0x5fc3e513 at offset 0x0007f56a.
Skipping to next test...
  Test  4:        SUB comparison:  Setting...Testing...
FAILURE: 0x17ca7030 != 0x17cc7030 at offset 0x0007f56a.
Skipping to next test...
  Test  5:        MUL comparison:  Setting...Testing...
FAILURE: 0xe5972bc0 != 0xd4bf2bc0 at offset 0x0007f56a.
Skipping to next test...
  Test  6:        DIV comparison:  Setting...Testing...
FAILURE: 0x00000001 != 0x00000000 at offset 0x0012d56a.
Skipping to next test...
  Test  7:         OR comparison:  Setting...Testing...
FAILURE: 0x4fb4a327 != 0x4fb4a326 at offset 0x0012d56a.
Skipping to next test...
  Test  8:        AND comparison:  Setting...Testing...Passed.
  Test  9:  Sequential Increment:  Setting...Testing...Passed.
  Test 10:            Solid Bits:  Setting...Passed.
  Test 11:      Block Sequential:  Setting...
FAILURE: 0x02020202 != 0x02000202 at offset 0x0007d56a.
Skipping to next test...
  Test 12:          Checkerboard:  Setting...Passed.
  Test 13:            Bit Spread:  Setting...Passed.
  Test 14:              Bit Flip:  Setting...Passed.
  Test 15:          Walking Ones:  Setting...Passed.
  Test 16:         Walking Zeroes:  Setting...
FAILURE: 0xfffffffe != 0xfffdfffe at offset 0x0007d56a.
Skipping to next test...
Run    1 completed in 818 seconds (8 tests showed errors).
munlock'ed memory.
1 runs completed.  8 errors detected.  Total runtime:  818 seconds.

Exiting...
```

그림 10 이상 노드의 Memtester 로그 파일

표 6 실험에 사용된 클러스터 시스템의 구성

시스템 명	Pluto (128노드)	Venus (64노드)
CPU	Intel Pentium IV 1.7GHz	Intel Pentium IV 1.7GHz
CPU/Node	1	1
Node 수	128	64
Memory	1 GB	512 MB
계산 네트워크	Myrinet 2000	Gigabit Ethernet
파일 서비스 네트워크	Fast Ethernet	Gigabit Ethernet

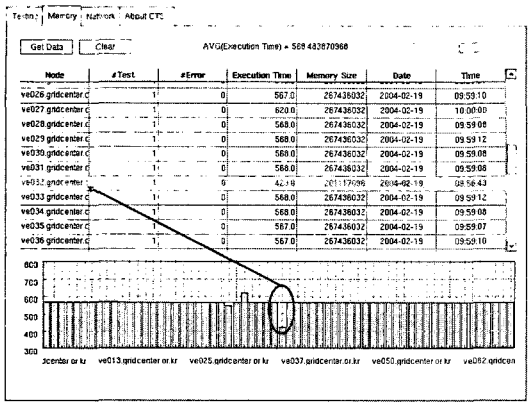

그림 11 64노드 클러스터 시스템에서의 메모리 검사 결과

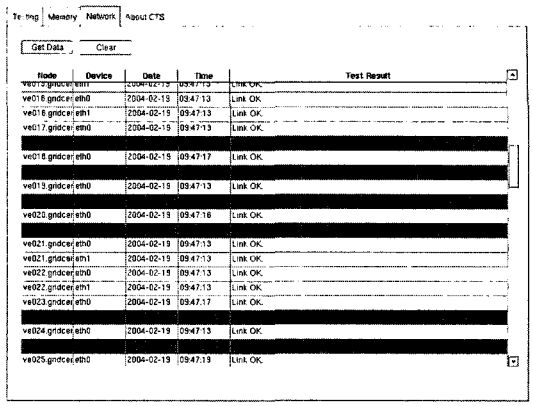

그림 12 64노드 클러스터 시스템에서의 네트워크 인터페이스 검사 결과

히려 증가되어 나타났다.

그림 11은 64노드 클러스터 시스템에서 CTS를 이용한 검사를 실행한 결과로서 이상 징후를 보이는 정보를 바탕으로 메모리 오류를 감지한 경우이다. 앞서 언급한 바와 같이 메모리 테스트를 수행한 결과, 메모리 오류를 발견하지 못하였더라도 각 계산노드에서의 실행 시간을 비교하여 실행 시간의 차이가 발견되는 노드들을 보고하도록 하였다. 그림 11에서처럼 결과 조회 화면의 우측 상단에 표시되는 평균 실행 시간을 기준으로 관리자가 편차 범위를 조정하여 그 편차 범위를 벗어나는 실행 시간을 갖는 계산 노드들을 표시하도록 하였다. 그림의 예는 평균 실행 시간 보다 100초 이상의 차이를 가지고 있는 노드를 표시하고 있다. 이는 메모리 검사를 수행하는 과정에서 현재 계산 노드가 가용할 수 있는 모든 메모리를 검사하도록 하였는데, 그 결과 다른 계산 노드와는 달리 상대적으로 적은 메모리만을 사용하여 실행 시간에서 차이를 보이는 것으로 나타났다. 해당 계산 노드를 조사한 결과, 다른 계산 노드와 같이 동일한 크기의 메모리를 장착하고 있음에도 불구하고 메모리에 발생한 문제로 인하여 올바른 크기의 메모리를 사용하지 못하고 있는 경우로 판명되었다.

4.2 네트워크 인터페이스 검사 결과

앞서 이미 설명한 바와 같이 본 검사는 SAN(System Area Network)과 같은 클러스터 전용 네트워크가 아니라 일반적인 이더넷을 이용하여 클러스터 시스템을 구성한 경우를 주요 대상으로 하고 있으며, 표 6의 64노드 클러스터 시스템처럼 이중의 네트워크를 보유한 시스템이 가장 적합한 대상이다.

물론 한 가지 단점은 네트워크 파일 시스템을 이용하여 검사를 수행하기 때문에 파일 서비스 네트워크의 문제는 실제 검사가 실행되기 전에 이미 파악되므로 무의미 할 수 있으나, 계산 네트워크와 같이 평상시에는 사용이 적어서 상태 확인을 원하는 시점에 각 노드별로 검사를 수행해야하는 경우에 유용하게 사용될 수 있다.

그림 12는 64노드 클러스터 시스템에서 네트워크 인터페이스에 대한 검사를 수행한 결과로서 계산 네트워크에 대한 오류가 감지된 예를 보여주고 있다. 이는 클러스터 시스템 운영시 이용이 잦아 이상 유무를 확인하기 쉬운 파일 서비스 네트워크보다 계산 작업 실행시에만 사용되는 계산 네트워크의 이상 유무를 파악하는데 유용한 정보를 줄 수 있다는 점을 보여주고 있다.

5. 결론 및 향후 계획

클러스터 시스템이 다양한 분야에 사용되어지면서 사용자도 증가하게 되어 초기의 실험적인 시스템이 아닌 안정적인 고성능 계산 자원으로서의 역할이 기대되어지고 있다. 안정된 시스템의 운영을 위해서는 주기적인 예방 정비가 필요하며 실제로 유명 고성능 컴퓨터 제조사들은 자사 제품에 대한 예방 정비 절차 및 지원 도구를 제공하여 왔다. 그러나 클러스터 시스템의 경우에는 현재까지 성능 향상 위주로 연구가 진행되어 안정된 시스템 운영을 위한 방법이나 도구에 대한 연구 및 개발은 소수의 하드웨어 업체 위주로 진행되었다. 이러한 이유로 본 논문에서는 예방 정비 과정에 사용할 수 있는 클러스터 시스템을 위한 검사 도구를 제안하였다.

개발의 효율성을 감안하여 기존의 공개 소프트웨어인 단일 시스템 기반의 메모리 및 네트워크 인터페이스 검사를 참조하여 분산된 각 노드들에서 검사를 실행할 수 있도록 구현하였다. 검사 결과를 데이타베이스에 저장하여 관리할 수 있도록 하였으며, 검사의 실행에서 결과의 조회까지 일관된 작업 환경을 제공하도록 사용자 인터페이스 기능을 추가하였다.

다수의 클러스터 시스템이 동질적인 계산 노드를 이

용하여 구성한다는 점을 고려하여 타 계산 노드의 실행 시간과 비교하여 이상 징후가 보이는 계산 노드의 정보를 보이는 기능을 추가하였는데, 이는 실제 운영 과정에서 검사 프로그램의 오류 감지 기능을 강화 할 수 있다는 사실을 보여 주었다.

향후 계획으로는 웹 인터페이스를 통한 검사 결과 조회 및 메모리와 네트워크 인터페이스 이외의 하드웨어 구성 요소에 대한 검사 기능의 추가를 계획하고 있다.

참 고 문 헌

[1] TOP 500 Supercomputer sites, http://www.top500.org

[2] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990(1991 Corrected Edition), The Institute of Electrical and Electronics Engineers, Inc., 1994.

[3] Mira Kajko-Mattsson, "Can we learn anything from hardware preventive maintenance?," Proc. of Seventh IEEE International Conference on Engineering of Complex Computer Systems, pp. 106~111, 2001.

[4] V. Biscaglia, C. Malaguti, and M. Paoletti Gualandi, "Maintenance planning on MV distribution network," IEE Conference Publication No. 438(14th International Conference and Exhibition on Electricity Distribution. Part 1. Contributions.), Vol. 3(19), pp. 1~4, 1997.

[5] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer, "Failure data analysis of a LAN of Windows NT based computers," Proc. of the 18th IEEE Symposium on Reliable Distributed Systems, pp. 178~187, 1999.

[6] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer, "Networked Windows NT system field failure data analysis," Proc. of Pacific Rim International Symposium on Dependable Computing, pp. 178~185, 1999.

[7] Rocks Cluster Distribution : An Open Source High Performance Linux Cluster Solution, http://www.rocksclusters.org/Rocks

[8] OSCAR : Open Source Cluster Application Ressources, http://oscar.openclustergroup.org/tiki-index.php

[9] SCore Cluster System Software 5.6 Documents, http://pdswww.rwcp.or.jp/score/dist/score/html/en/index.html

[10] HP Integrated Lights-Out Advanced, http://h18013.www1.hp.com/products/servers/management/iloadv/index.html

[11] IBM Redbook, "Implementing Systems Management Solutions using IBM Director," 2003, http://publib-b.boulder.ibm.com/Redbooks.nsf/9445fa5b416f6e32852569ae006bb65f/59299a2cb12fea3f85256c75004e2dd3?OpenDocument

[12] Charles Cazabon, "Memtester," http://www.qcc.ca/~charlesc/software/memtester

[13] Linux Ethercard Status, Diagnostic and Setup Utilities, http://web.archive.org/web/20030608223511/www.scyld.com/diag

[14] Clusters@Top500, http://clusters.top500.org/

[15] Memtest86 - A Stand alone Memory Diagnostic, http://www.memtest86.com

[16] Michael D. Crawford, "Using Test Suites to Validate the Linux Kernel," http://linuxquality.sunsite.dk/articles/testsuites

[17] James H. Laros III, Lee Ward, Nathan W. Dauchy, James Vasak, Ruth Klundt, Glen Laguna, Marcus Epperson, and Jon R. Stearley, "The Cluster Integration Toolkit - An Extensible, Portable, Scalable Cluster Management Software Implementation," Proc. of 1st Cluster World Conference and Expo, pp 23~26, 2003.

[18] James H. Laros III, Lee Ward, Nathan W. Dauchy, Ron Brightwell, Trammell Hudson, and Ruth Klundt, "An Extensible, Portable, Scalable, Cluster Management Software Architecture," Proc. of IEEE International Conference on Cluster Computing, pp 287~295, 2002.

[19] The Computational Plant project, http://www.cs.sandia.gov/cplant.

[20] The Parallel Tools Consortium, http://www.ptools.org

[21] IBM Redbook, "Building a Linux HPC Cluster with xCAT," 2002, http://publib-b.boulder.ibm.com/Redbooks.nsf/0/7b1ce6b3913cafb386256bdb007595e8?OpenDocument&Highlight=0,SG24-6623-00

[22] MySQL Website, http://www.mysql.com

차 광 호

1999년 8월 숭실대학교 컴퓨터학부(공학사). 2002년 2월 한국정보통신대학교(ICU) 정보공학부(공학석사). 2002년 4월~현재, 한국과학기술정보연구원 슈퍼컴퓨팅센터 연구원. 관심분야는 병렬 컴퓨팅, 클러스터 컴퓨팅, 병렬 파일 시스템