CMFstone: 유닉스 기반의 합성 사용자 프로그래밍 벤치마크

이 시 진[†] · 박 성 욱[††] · 권 혁 인[†††]

요 약

벤치마크는 컴퓨터 시스템의 성능을 측정하기 위한 프로그램이며, 컴퓨터 시스템의 성능은 사용자가 실행하는 응용 프로그램의 실행되는 속도에 의해서 결정된다. 따라서, 컴퓨터 시스템의 성능을 정확히 측정하기 위한 벤치마크 프로그램은 사용자가 실행시키는 응용 프로그램과 동일한 동작 특성을 가져야 한다. 본 연구에서는, Chaustone, Mchaustone, Fhsstone으로 이루어진 벤치마크 프로그램인 CMFstone을 구현한다. CMFstone을 구현·실행한 결과, 다른 여러 벤치마크를 실행한 결과의 평균과 유사한 측정 결과를 보였다. 따라서, 컴퓨터 시스템의 성능 측정을 위해서 여러 벤치마크를 실행할 필요 없이, CMFstone을 실행시키는 것만으로도 다양한 성능 측정이 가능하다.

CMFstone: Synthetic User Programming Benchmark based on UNIX

Sijin Lee[†] · Sunguk Park[††] · Hyeogin Kwon[†††]

ABSTRACT

The purpose of benchmark program is to measure the performance of a computer system. The performance of a computer system is determined by the amount of execution time of user application programs. Thus, it is assumed that a benchmark program must have the same features with user application programs to test. In this paper, we have designed and implemented CMFstone which is consisted of Chaustone, Mchaustone and Fhsstone. After applied the CMFstone, designed and implemented in this paper, to the real situations, the results of comparison show that CMFstone is similar to geometric mean of other benchmarks results. Thus, we have concluded that CMFstone is good enough to measure the performance of a computer system.

1. 서 론

컴퓨터 시스템의 성능은 대형 컴퓨터나 미니 컴퓨터를 독립 시스템으로 사용하던 시대에서부터 다양한 종류의 다중 처리기 시스템, 워크스테이션 또는 근거리 통신망(LAN)을 통해 상호 연동 될 수 있는 클라이언트-서버 모델을 기반으로 하는 분산 시스템에 이르는 현재까지 주요 관심사가 되고 있다. 많은 컴퓨터 시스템 개발자 또는 관리자들이 시스템의 성능을 측정하고 분석하는 주요 이유는 시스템의 운용에 관련된 주요 변수들을 계속적으로 관찰하고 시스템 각 자원의 사용 현황을 측정함으로써 시스템이 효율적으로 운용되도록 지원할 뿐만 아니라 시스템 성능을 전체적으로 향상시키며, 측정된 자료를 추후 시스

† 정 회 원: 중앙대학교 대학원 컴퓨터 공학과 박사과정
†† 비 회 원: LG 정보 통신
††† 정 회 원: 중앙대학교 경영학과 조교수

논문접수: 1996년 2월 9일, 심사완료: 1996년 5월 15일

템 확장 계획에 반영하기 위함이다[1][2][3].

컴퓨터 시스템의 성능 측정을 위한 기법은 모니터링(monitoring) 도구를 이용한 동적 성능 측정 기법과 벤치마크(benchmark)를 이용한 정적 성능 측정 기법으로 분류될 수 있다. 모니터링 도구를 이용하여 시스템의 성능을 동적으로 측정하는 기법은 추적 구동형(trace-driven) 기법, 시간 구동형(time-driven) 기법, 그리고 이벤트 구동형(event-driven) 기법 등으로 분류할 수 있다. 벤치마크를 이용하는 정적 성능 측정 기법은 CPU의 속도, 평균 디스크 접근시간과 같이 항상 일정한 결과를 가지며, 평균치를 기반으로 하는 성능 정보를 수집하기 위하여 사용된다. 본 논문에서는 벤치마크를 이용한 정적 성능 측정 기법에 대하여 논하며, 본 논문에서 컴퓨터 시스템의 성능이라 함은 정적 성능을 말한다. 벤치마크는 컴퓨터 시스템의 성능을 비교, 평가하기 위한 기준을 가리킨다. 이 기준치를 얻기 위한 작업을 벤치마킹(benchmarking), 수단으로 이용하는 프로그램을 벤치마크 프로그램, 얻어지는 결과 즉 기준치를 벤치마크 값이라 부른다.

본 논문의 구성은 다음과 같다. 제 2장에서는 연구의 배경으로 벤치마크 프로그램의 종류와, 대표적인 벤치마크인 Dhrystone, Whestone, Linpack 등에 관해서 살펴본다. 제 3장에서는 본 논문에서 구현한 벤치마크 프로그램인 CMFstone에 관한 구현 내용을 기술한다. 제 4장에서는 CMFstone과 다른 벤치마크 프로그램들 그리고 실제 유닉스 응용 프로그램들의 실행 속도를 비교하여 CMFstone의 정확성을 평가한다. 마지막으로, 제 5장에서는 결론과 향후 연구방향에 대해서 기술한다.

2. 연구 배경

이 장에서는 컴퓨터 시스템의 성능 측정을 위한 벤치마크 프로그램을 구현하기 위한 연구 배경을 기술한다.

2.1 벤치마크 프로그램의 필요성

컴퓨터 시스템의 성능을 나타내는 척도로는 최소 연산 실행시간, MIPS (Millions Instruction Per Second), FLOPS(FLoating point Operations Per Second) 등이 사용되어 왔다. 그러나, 명령 축소형 컴퓨터 시스템(Reduced Instruction Set Computer, RISC)의 출현으로 인하여 이러한 척도는 의미를 잃게 되었다. 왜냐하면, 동일한 작업을 수행하는 프로그램이 명령 복합형 컴퓨터 시스템(Complex Instruction Set Computer, CISC)과 명령 축소형 컴퓨터 시스템에서 동일한 작업 시간을 소모하였다고 할 때, 명령 축소형 컴퓨터 시스템에서는 명령 복합형 컴퓨터 시스템에서보다 실행 프로그램의 크기가 크므로 큰 MIPS 수치를 유도한다. 그러나, 이 때, 명령 축소형 컴퓨터와 명령 복합형 컴퓨터 시스템에서 실행된 프로그램은 동일한 작업을 동일한 시간에 수행하였으므로, 명령 축소형 컴퓨터 시스템이 명령 복합형 컴퓨터 시스템보다 큰 MIPS 수치를 가지지만 성능이 더 뛰어나다고 말 할 수 없기 때문이다. 따라서, 컴퓨터 시스템의 성능을 정확히 파악하기 위해서는 응용 프로그램을 직접 실행시켜 실행시간을 측정해야 한다. 그러나, 컴퓨터 시스템의 성능을 측정하기 위하여 측정하고자 하는 컴퓨터 시스템에 응용 프로그램을 사용자가 직접 수행하기는 실제로 거의 불가능하다. 그것은 성능을 측정하고자 하는 컴퓨터 시스템의 종류, 실행하고자 하는 응용 프로그램의 종류를 정확히 선택할 수 없기 때문이다[4]. 따라서, 좀더 현실적인 방법으로 실행하려는 응용 프로그램의 동작 특성을 감안한 프로그램을 만들어서 실행시킬 수 있는데, 이렇게 만들어진 프로그램을 벤치마크 프로그램이라 한다. 그러나, 벤치마크 프로그램을 직접 만들기는 어려우므로 이미 이용되고 있는 벤치마크 프로그램을 사용하는 것이 대부분이다.

2.2 벤치마크 프로그램의 분류

벤치마크 프로그램은 개인용 컴퓨터에서부터 슈퍼컴퓨터에 이르기까지 다양하게 활용된다. 그러나, 벤치마크 프로그램을 모든 컴퓨터 시스템에 적용하여 사용할 수 있는 것은 아니다. 또한, 대부분의 벤치마크 프로그램들의 평가 대상은 전체 컴퓨터 시스템 중의 특정한 서브시스템의 성능을 측정한다.

또한, 벤치마크 프로그램은 구현된 방법과 내부적인 알고리즘에 따라 커널 벤치마크(kernel benchmark), 지역 벤치마크(local benchmark), 부분 벤치마크(partial benchmark), 재귀 벤치마크(recursive benchmark), 합성 벤치마크(synthetic benchmark), 유닉스 유틸리

티 및 응용 벤치마크(UNIX utility and application benchmark) 등으로 구분될 수 있다[5]. 먼저, 커널 벤치마크는 실제 프로그램에서 대부분의 실행시간을 차지한다고 생각되는 부분의 코드를 추출해서 만들어진 것이다. 이들 대부분은 합성 벤치마크와 유사한 장점 즉, 코드의 크기가 작고, 실행시간이 길다는 특징을 가진다. 대표적인 커널 벤치마크 프로그램으로는 Livermore, Linpack 및 SPEC의 Matrix 300 등이 있다. 지역 벤치마크는 특정한 컴퓨터 시스템에서만 사용될 수 있는 벤치마크이며, 부분 벤치마크는 응용 프로그램의 일부분을 선택해서 만들어진 벤치마크이다. 재귀 벤치마크는 재귀 알고리즘을 구현한 것이다. 예로써, Tower of Hanoi, Nine Queens 및 SPEC의 li 등이 대표적이다. 합성 벤치마크는 실제 프로그램의 평균적인 특성을 축소하여 작은 프로그램으로 대표하게 한다. 대표적인 합성 벤치마크 프로그램으로는 Dhrystone과 Whetstone등이 있다. 마지막으로, 유닉스 유틸리티와 응용 벤치마크는 사용자들이 많이 사용하는 프로그램들을 벤치마크로 사용하는 것으로서 grep과 nroff 등을 예로 들 수 있다.

2.3 벤치마크 프로그램의 연산

벤치마크 프로그램은 CPU, 메모리 및 디스크 서브시스템 등에 대하여 연산을 수행하고 벤치마크 값을 계산한다. 먼저, CPU에 대해서는 정수 연산, 부동 소수점 연산, 정수 비교, 부동 소수점 비교, 분기(branch), 함수 호출, 대정수(large integer) 연산 등을 수행해야 한다. 메모리에 대해서는 CPU에서 메모리로의 데이타 이동, 메모리에서 CPU로의 데이타 이동, 메모리에서 메모리로의 데이타 이동, 대배열 어드레싱(addressing large array) 등의 연산을 수행해야 한다. 그리고, 디스크에 대해서는 읽기, 쓰기, 복사, 플러쉬(flush), 마운트(mount), 언마운트(unmount) 및 버퍼 고찰(consider buffer) 등의 연산을 수행해야 하며, 그 밖에 시스템 호출, 문맥 교환, NFS(Network File System)등의 연산을 수행하는 벤치마크 프로그램들이 있다.

2.4 관련연구

2.4.1 Dhrystone

Dhrystone[6][7]은 Reinhold P. Weiker에 의해 개발된 벤치마크 프로그램으로써 개인용 컴퓨터나 유닉스 계열의 워크스테이션에서 정수 연산에 대한 성능 평가의 기준으로써 자주 사용된다. Dhrystone 벤치마크는 여러 시스템에서 사용되는 프로그래밍 언어들의 문장, 연산자, 피연산자 등의 분포를 통계적으로 측정한 자료에 따라 설계되었다. Dhrystone 벤치마크에서는 내부적으로 사용하는 명령의 내용 자체에는 의미가 없으며, 정수 연산이나 주기억장치 접근과 같은 일련의 CPU 처리 명령을 반복 실행한다. 그 결과치는 1초 동안 CPU 명령을 일정 비율로 섞은 명령 mix를 몇번 실행했는가에 의해 결정된다. 또한, Dhrystone 결과값을 VAX-11/780의 Dhrystone값인 1757로 나누어 VAX-MIPS 수치를 구할 수 있다.

그러나, Dhrystone 벤치마크는 그 기초가 되는 문장, 연산자, 피연산자 등의 자료가 UNIX에서 사용되는 C 언어로 작성된 응용 프로그램에서 추출된 것이 아니므로, 현재 유닉스 시스템에서 사용되는 프로그래밍 유형을 대표한다고 볼 수 없다. 또한, Dhrystone은 프로그램 자체의 크기가 작아서 전체 프로그램이 캐쉬(cache)에 들어갈 수 있기 때문에 컴퓨터 시스템의 최고 성능을 측정 할 수 있지만, 실제 응용 프로그램을 실행했을 때의 성능은 정확히 평가 할 수 없다는 단점이 있다.

2.4.2 Whetstone

Dhrystone이 정수 연산 성능을 측정하는데 비하여 Whetstone[7]은 부동 소수점 연산 성능을 측정하는 벤치마크 프로그램이다. 성능의 단위는 KWIPS(Kilo Whetstone Instructions Per Second)로 나타낸다. Whetstone은 벤치마크를 위해서 만들어진 최초의 프로그램이라는 데에 그 의미를 찾을 수 있다. Whetstone은 부동 소수점 연산 작업 비율과 수학 라이브러리 함수를 실행시키는 시간의 비율이 높으며, 지역 변수를 거의 사용하지 않고, 전역 변수를 많이 사용하며, 코드 지역성이 높다는 특징이 있다. Whetstone은 처리 내용이 현재의 하드웨어 발달을 따라가지 못하기 때문에 요즘은 거의 사용되지 않고 있다.

2.4.3 Linpack

Linpack[7]은 Whetstone과 마찬가지로 부동 소수점 연산 성능의 평가를 위한 벤치마크 프로그램이다.

Whetstone이 스칼라의 부동 소수점 연산만을 평가하는데 비하여, Linpack은 벡터의 부동 소수점 연산 평가가 중심이며, MFLOPS(Million FLoating-point Operations Per Second)를 단위로 사용한다. Linpack의 결과값은 본래는 1초 동안 부동 소수점 연산을 몇 회 실행하는가를 나타내지만, Dhrystone과 마찬가지로 VAX-11/780과의 상대비로 나타내기도 한다. Linpack은 원래 벤치마크를 위해서 만들어진 프로그램은 아니었으나, 후에 Linpack의 결과들을 모아 벤치마크의 특성을 갖게 했다. Linpack은 행렬 방정식을 푸는 프로그램으로 이루어져 있으며, 전체 실행시간 중 하나의 작은 함수를 실행시키는 데에 대부분의 시간이 소요된다. 그 함수를 saxpy라고 부르며 다음과 같은 코드를 실행한다.

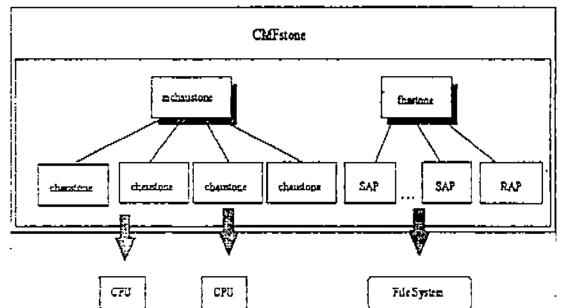
y[i] = y[i] + a * x [i]

따라서, Linpack은 메모리 접근시간을 측정하는 데에 유용하게 사용된다. Linpack은 Dhrystone과 달리 코드가 커서 캐쉬에 모두 들어갈 염려가 없어 메모리 시스템의 평균 성능을 측정하는 데에 사용될 수 있다는 장점을 가지지만, 코드 지역성은 높고, 데이타 지역성은 낮으므로 행렬의 크기가 크면 캐쉬 실패율이 증가하고, 낮은 Mflops의 수치를 출력하게 된다.

3. CMFstone 벤치마크 구현

본 논문에서는 UNIX 운영체제를 사용하는 워크스테이션과 미니 컴퓨터 시스템의 CPU와 화일 시스템의 성능 측정을 위한 벤치마크 프로그램을 구현한다. 사용자가 실제로 느끼는 성능에 대해서 최대로 근접한 결과를 제공할 수 있는 벤치마크를 구현하기 위해서, 사용자 환경을 조사·분석하고, 이를 토대로 합성 벤치마크 프로그램인 CMFstone을 구현한다.

Chaustone은 워크스테이션과 미니 컴퓨터 시스템의 CPU 성능을 측정하기 위하여 새롭게 구현하는 벤치마크 프로그램으로, 정수형 연산과 단정밀도(single precision) 부동 소수점, 배정밀도(double precision) 부동 소수점 연산 모두를 수행한다. Chaustone은 SPEC과 같이 정수형과 부동 소수점 연산 각각에 대해서 성능 측정 결과를 출력하는 것이 아니라, 하나의 수로 합성된 결과를 출력한다. Chaustone은 사용자가 응용 프로그램을 실행시킬 때와 가장 근접한 작업부하(workload)를 만들기 위하여 여러 사용자 응용 프로그램들에 대해서 구성 요소의 출현 횟수를 통계적으로 측정하고, 각 응용 프로그램들의 실행 빈도에 의한 가중치를 추가함으로써 구현된다. 따라서, Chaustone은 실제 사용자가 응용 프로그램을 실행시키는 것과 동일한 형태의 워크로드를 성능 측정하고자 하는 컴퓨터 시스템에 부여한다. 그러므로, Chaustone의 결과값은 실제 사용자가 응용 프로그램을 실행시켰을 때와 동일한 CPU 성능을 출력할 수 있게 된다. 그러나, Chaustone은 하나의 프로세스로 이루어져 있기 때문에, 다중 처리기 시스템의 성능을 측정하지 못한다는 단점이 있다. 이 문제점을 해결하기 위하여, 대칭형(symmetric) 다중처리기 시스템의 성능 측정을 위한 벤치마크 Mchaustone을 구현한다. 또한, 화일 시스템의 성능 측정을 위한 벤치마크 Fhsstone을 구현한다. Fhsstone은 Howard[8]의 제안에 기초해서 만들어지며, 특히, Fhsstone은 순차접근(sequential access)과 임의접근(random access)의 횟수와 분포를 고려했기 때문에, 디스크 버퍼에 대한 효율을 함께 측정할 수 있다는 장점을 가지고 있다.



(그림 1) CMFstone의 내부 구조
(Fig. 1) Internal structure of CMFstone

CMFstone의 벤치마크들과 측정 하고자 하는 시스템과의 관계는 (그림 1)과 같다. CPU의 성능을 측정하는 벤치마크인 Chaustone은 CPU 갯수의 두배만큼 실행되며, Mchaustone은 Chaustone을 실행하기 위한 모듈이다. Fhsstone는 아홉개의 순차 접근 프로세스(Sequential Access Process;SAP)와 하나의 임의 접근

프로세스(Random Access Process ; RAP)로 구성된다. 이들은 각각 데이타 화일을 통해서 화일 시스템에 접근하고, 접근한 화일 시스템의 성능을 측정한다.

3.1 Chaustone

Chaustone은 사용자가 응용 프로그램을 실행시킬 때와 가장 근접한 작업부하를 만들기 위하여 UNIX 운영체제에서 사용자들에게 일반적으로 자주 사용되는 응용 프로그램인 vi, gcc, gzip, xview등을 가지고, 소스 코드 수준에서 각각의 문장 형식과 연산자, 피연산자, 지역성 등을 추출하여, 이와 동일한 분포를 갖도록 Chaustone을 구현한다. 따라서, Chaustone은 UNIX 운영체제를 사용하는 컴퓨터 시스템에서의 사용자 응용 프로그램을 정확히 대변할 수 있으므로, 사용자가 느끼는 CPU 성능에 최대로 근접한 결과를 출력한다.

3.1.1 각 응용 프로그램들의 구성요소의 분포

응용 프로그램의 구성 요소에는 문장의 유형, 연산자 종류, 피연산자 유형, 지역성의 정도 등이 있다. 각 구성요소의 출현 빈도에 따라 컴퓨터 시스템에 부여되는 작업부하에 차이가 날 수 있으므로, 이들 구성 요소의 출현 빈도를 정확히 추출해야 한다.

3.1.1.1 문장 형식 별 출현 빈도

먼저, 각 응용 프로그램들의 구성 요소의 출현 빈도를 구하기 위하여, 반복문과 선택문에 대해서 다음 사항을 가정한다. 첫째, 반복 횟수가 반복문에 명시된 경우, 각 반복 블록은 명시된 횟수만큼 실행되는 것으로 가정한다. 예를 들어, 다음과 같은 문장에서 반복 블록은 100회 수행되는 것으로 가정한다.

```
for(i=0;i<100;i++){
    a=100*i;
    b=a*c;
```

둘째, if나 switch같은 선택문은 각 선택 블록이 (1/선택 블록의 갯수)회 실행되는 것으로 가정한다. 예를 들어, 다음과 같은 문장에서 각 선택 블록은 0.25회씩 수행되는 것으로 가정한다.

```
if(...){
        a=100*i;
        b=a*c;
    }
else if(...){
        a=10*i;
        b=a+c;
    }
else if(...)
        a=100;
else
        b=100;
```

셋째, 다음의 문장과 같이, 반복 횟수가 반복문에 명시되지 않은 경우, 각 반복 문장은 38회 수행되는 것으로 한다. 이것은 gcc, gzip, vi, xview에서 100개씩의 반복 횟수가 명시되지 않은 문장들을 추출하고, 직접 실행 횟수를 알기 위한 문장을 삽입하여 측정한 평균치이다.

```
int i;
while(i<var){
        a=100*i;
        b=a*c;
}
```

위와 같은 사항을 가정한 뒤 추출한 각 응용 프로그램의 문장 형식별 출현 빈도는 〈표 1〉과 같다.[1]

3.1.1.2 연산자 출현 빈도

대부분의 연산자는 할당문에 나타나지만 단순한 선택 수식에 나타나는 경우도 있다. 하지만 기계어 수준에서 동일한 연산을 수행하므로 연산자별 출현 빈도를 추출할 때 이들을 별도로 구분할 필요는 없

1. 본 논문의 모든 통계치는 소수점 이하 첫번째 자리에서 반올림한 수치이며, 반올림하여 0% 되는 항목은 삭제하였다. 따라서, 총합이 100%를 초과 또는 미만이 될 수 있다.

〈표 1〉 각 응용 프로그램별 문장 형식 출현 빈도
〈Table 1〉 Distribution of statements of each application programs

프로그램 구성 요소	gcc	gzip	vi	xview
assignment	45%	52%	53%	42%
function call	29%	23%	25%	22%
if	5%	5%	4%	6%
if.. else	7%	3%	4%	4%
switch	1%	4%	3%	3%
for	3%	5%	2%	6%
while	3%	4%	1%	7%
do..while	1%		1%	3%
break	2%	2%	4%	4%
continue	1%	1%	3%	3%
return	3%	4%	4%	4%

다. 또한, 각 연산자는 피연산자에 따라 기계어 수준에서 다른 연산을 수행할 수도 있다. 즉, '+'는 피연산자가 정수형일 때와 실수형일 때 다른 연산을 사용한다. 그러나, 고수준 언어에서는 동일한 연산자로 표기하며, 다음에 구해질 피연산자별 분포가 Chaustone의 구현시에 고려되므로 역시 구분할 필요는 없다. 이와 같은 사항을 고려하여 추출한 각 응용 프로그램의 연산자 출현 빈도는 〈표 2〉와 같다.

3.1.1.3 피연산자 유형 출현 빈도

피연산자의 유형별 출현 빈도를 측정하기 위해서는 구조체, 배열, 포인터로 사용된 피연산자에 유의해야 한다. 이들 유형을 분석하는 방법[6]은 다음 3가지 방법이 있다.

- 마지막 피연산자의 유형을 참조한다.
- 최초 피연산자의 유형을 참조한다.
- 모든 피연산자의 유형을 참조한다.

예를 들어, 다음과 같은 프로그램의 일부를 가정하자.

```
struct{
    int i;    char c;
}var[100];
var[10]. i=3;
```

〈표 2〉 각 응용 프로그램별 연산자 출현 빈도
〈Table 2〉 Distibution of operators of each application programs

프로그램 구성요소	gcc	gzip	vi	xview
+	12%	11%	11%	16%
−	9%	10%	8%	13%
*	9%	9%	6%	5%
/	5%	6%	3%	4%
%	3%	2%	1%	3%
−>	5%	3%	2%	1%
++	9%	10%	12%	2%
−−	7%	6%	8%	6%
sizeof	1%	1%	1%	3%
?:		2%	3%	2%
<	5%	3%	2%	3%
>	4%	4%	3%	3%
<=	3%	2%	2%	4%
>=	3%	1%	1%	3%
+=	2%	4%	1%	1%
−=	1%	3%		2%
*=	1%	4%	1%	1%
&	1%	4%	3%	4%
==	5%	3%	8%	5%
!=	3%	3%	9%	4%
&&	1%	3%	2%	1%
||	1%	3%		1%
.	2%	1%	6%	3%
!	2%		2%	
>>		2%	1%	2%
<<		1%	1%	2%

위 할당문의 좌변을 분석하는 경우에, 첫 번째 방법을 이용해서 분석하면, 'i'의 유형만을 참조하는 것이고, 두 번째 방법을 이용하는 경우에는 'var'의 유형만을 참조하는 것이며, 세 번째 방법을 이용하는 경우에는 'var'과 'i'의 유형을 모두 참조하여 구조체의 요소, 배열의 요소, 정수형 모두로 인식하는 것이다. 응용 프로그램에 사용된 피연산자의 유형별 출현 빈

도를 정확하게 추출하기 위해서는 모든 피연산자의 유형을 참조해야 하므로 본 연구에서는 위에서 기술한 방법 중 세 번째 방법을 이용하여 모든 피연산자의 유형을 인식하는 방법을 택한다. 이와 같은 방법을 이용하여 추출한 각 응용 프로그램들의 피연산자 유형별 출현 빈도는 <표 3>과 같다.

<표 3> 각 응용 프로그램별 피연산자 유형 출현 빈도
<Table> Distribution of operands of each application programs

프로그램 / 구성요소	gcc	gzip	vi	xview
constant	18%	17%	21%	19%
int	27%	32%	24%	22%
char	8%	7%	14%	7%
short	2%	1%	1%	
long	1%	2%		4%
float	1%	1%		2%
double	7%	3%	1%	2%
pointer	21%	18%	14%	22%
struct member	8%	7%	9%	12%
array member	9%	12%	14%	17%

3.1.1.4 지역성

지역성(locality)은 캐쉬 적중률(hit ratio)과 밀접한 관계를 가지므로 응용 프로그램 문장의 전체적인 분포가 동일하더라도 지역성에 차이가 있다면, 실행시간이 다를 수밖에 없다. 지역 변수와 전역 변수의 분포는 데이타 캐쉬에, 중문(compound statement)의 평균 길이는 명령어 캐쉬와 데이타 캐쉬에 영향을 미친다. 따라서, 문장 분포 추출시 지역성도 고려해야 한다. 각 응용 프로그램들의 지역성은 <표 4>와 같다.

<표 4> 각 응용 프로그램별 지역성
<Table> Locality of each application programs

프로그램 / 구성요소	gcc	gzip	vi	xview
global variable	18%	14%	13%	20%
local variable	82%	86%	87%	80%
compound statement length	13	10	8	9

<표 4>에서 보듯이 지역 변수가 많이 등장할수록 지역성은 높아지고, 반면에 전역 변수가 많이 등장하면, 지역성이 낮아지는 것을 의미하기 때문에 이를 반영해 줘야 한다. 또한 중문으로 구성된 문장은 주로 그 안에서 loop 등과 같은 반복 실행이 자주 일어나기 때문에 중문에서의 문장수도 지역성안에서 고려돼야만 한다.

3.1.2 Chaustone의 분포

앞 절의 <표 1>과 <표 3>에서의 각 응용 프로그램들에 대한 구성 요소들의 출현 분포를 하나의 단일한 수치로 병합해야 한다. 병합된 단일한 수치는 Chaustone의 구현에 적용된다. Chaustone은 문장 형식, 연산자, 피연산자, 지역성 등의 분포와 병합된 수치가 동일하도록 구현된다. 병합하는 방법에는 실제 응용 프로그램들의 실행 횟수를 고려하는 방법과 고려하지 않는 방법, 그리고 각각에 대해서 백분율 분포에 따른 평균과 실제 출현 횟수에 따른 평균으로 하는 방법등, 다음과 같이 총 네 가지로 나눌 수 있다.

3.1.2.1 방법 1 및 방법 2

방법 1과 방법 2는 각 응용 프로그램들의 실행 빈도를 생각하지 않는다. 즉, 각 응용 프로그램에서 추출된 각각의 요소들의 평균을 구해서 Chaustone의 분포로 삼는다. 이 방법은 평균을 구하는 기준에 따라 다시 두 가지로 나뉘어 질 수 있다. 그 중 방법 1은 각 응용 프로그램들에서 추출된 각 요소들의 출현 빈도를 기준으로 평균을 구하는 것이다. 예를 들어, if 문이, gcc에서 5%, xview에서 10%, gzip에서 15%, vi에서 20% 였다면, 결론적으로 $(5+10+15+20)/4=12.5\%$로 Chaustone에서 if문의 출현 비율이 결정된다. 방법 2는 각 응용 프로그램 요소들의 실제 출현 횟수를 기준으로 하는 방법이다. 예를 들어, if문이, gcc에서 50회, xview에서 100회, gzip에서 150회, vi에서 200회 출현하였다면, 결론적으로 $(50+100+150+200)/4=125$회 출현한 것으로 한다.

방법 1과 방법 2는 단순하며, 후에 다른 응용 프로그램들에 대한 통계를 다시 추출해서 추가하기에 용이하다는 장점이 있다. 즉, 새로이 추가하고자 하는 응용 프로그램에 대해서만 다시 출현 빈도의 통계를 구하면, 즉시 Chaustone의 확장이 가능하다. 그러나,

실제 사용자의 응용 프로그램을 정확히 대변하지 못한다는 단점을 가지고 있다.

3.1.2.2 방법 3 및 방법 4

실제 각 응용 프로그램들의 수행 빈도가 다르기 때문에, 방법 1 및 방법 2와 같은 방법은 실제 사용자 응용 프로그램의 실행을 대변한다는 Chaustone의 취지에 맞지 않다. 방법 3과 방법 4는 실제 각 응용 프로그램의 실행 빈도를 고려하기 위해서, 일정 기간 각 응용 프로그램들의 실행 횟수를 구하고, 그 비율에 따라 가중치를 두어서, Chaustone의 작업부하를 실제 사용자들이 응용 프로그램을 실행시킬 때의 작업부하와 동일하도록 한다. 방법 3은 각 응용 프로그램들에서 추출한 요소들의 출현 비율에 가중치를 부여하는 방법이다. 예를 들어, gcc:xview:gzip:vi의 실행 비율이 1:2:3:4이고, 출현 빈도 비율이 방법 1에서의 예와 같다면, if 문의 출현 비율을 (5%*1+10%*2+15%*3+20%*4)/4=37.5%로 하는 것이다. 그런데, 이 예에서 보듯, 각각 5%, 10%, 15%, 20%의 실행 비율에 의한 결론이 37.5%라는 결과가 나오므로 방법 3은 비현실적이며 수용할 수 없다. 방법 4는 각 요소들의 출현 횟수에 가중치를 부여하는 방법이다. 예를 들어, gcc:xview:gzip:vi의 실행 비율이 1:2:3:4이고, 출현 횟수가 방법 2에서의 예와 같다면, if 문의 출현 횟수를 50*1+100*2+150*3+200*4=1500회 출현한 것으로 한다.

방법 4는 방법 1 및 방법 2에 비해서, 실제 사용자의 응용 프로그램 실행시의 작업부하와 동일한 작업부하를 만들어 낼 수 있다는 장점이 있으나, 후에 다른 응용 프로그램의 통계치를 추가하려 할 때, 각 응용 프로그램들의 실행 빈도를 모두 다시 구해야 한다는 단점이 있다. 즉, 빈번한 버전업(version-up)이 어렵다는 단점이 발생한다. 또한, 실행 빈도 측정 지역에 따라 실행 빈도가 다를 수 있으므로, 여러 장소에서 오랫동안 실행 빈도를 측정해서 한다.

t_a를 구성요소의 모음 중에서 a번째 구성 요소라 하자. 즉, 연산자의 출현 분포를 구할 때 t_2는 〈표 4〉의 두 번째 구성 요소이므로 -(뺄셈 연산자)이다. 또한, N(t, p)를 구성 요소 t의 응용 프로그램 p에서의 출현 횟수, D(t, p)를 구성 요소 t의 응용 프로그램 p에서의 출현 비율, E1, E2,...,EA를 각 응용 프로그램 p1, p2,...,pA의 실행 비율이라 하자. 단, A는 분포 측정에 사용된 응용 프로그램의 갯수이다. 그러면, 위의 각 방안들에서 구성 요소 t_a의 최종 분포 비율은 각각 다음과 같다.

- 방법 1: $$\frac{\sum_{i=1}^{A} D(t_a, pi)}{A} \quad (1)$$

- 방법 2: $$\frac{\sum_{k=1}^{A} N(t_a, pk)}{\sum_{j}\sum_{i} N(t_i, pj)} \quad (2)$$

- 방법 3: $$\frac{\sum_{i=1}^{A} (D(t_a, pi) * Ei)}{A} \quad (3)$$

- 방법 4: $$\frac{\sum_{k=1}^{A} (N(t_a, pk) * Ek)}{\sum_{i=1}^{A}\sum_{j} (N(t_j^a, pi) * Ei)} \quad (4)$$

3.1.2.3 Chaustone에서의 방법

본 논문에서는 응용 프로그램 구성 요소들의 통계 결과를 병합하여 Chaustone을 구현하기 위한 방안으로서 방법 4를 선택한다. 그 이유는, 실제 사용자들의 작업부하와 동일한 작업부하를 만들어 내는 것이 중요하기 때문이다.

각 응용 프로그램들의 실행 빈도(Ei) 측정 결과는 gcc:gzip:vi:xview=36:4:100:3이었다. 이 측정은 1995년 5월 1일 부터 1995년 9월 15일 까지, 중앙대학교 전산센터의 주전산기 II(ticom.cc1.cau.ac.kr)와 중앙대학교 공과대학 컴퓨터 공학과의 Solbourne 5/600

〈표 5〉 Chaustone의 문장 형식 분포
〈Table 5〉 Distribution of statements of Chaustone

구성 요소	Chaustone	구성 요소	Chaustone
assignment	49%	while	2%
function Call	25%	do-while	1%
if	5%	break	3%
if-else	5%	continue	3%
switch	2%	return	4%
for	2%		

〈표 6〉 Chaustone의 연산자 분포
〈Table 6〉 Distribution of operators of Chaustone

구성 요소	Chaustone	구성 요소	Chaustone
+	11%	>=	2%
-	8%	+=	1%
*	7%	-=	1%
/	4%	*=	1%
%	2%	&	2%
->	4%	==	7%
++	11%	!=	7%
--	8%	&&	2%
sizeof	1%	||	
?:	2%	.	5%
<	4%	!	2%
>	3%	>>	1%
<=	3%	<<	1%

〈표 7〉 Chaustone의 피연산자 타입 분포
〈Table 7〉 Distribution of operands of Chaustone

구성 요소	Chaustone	구성 요소	Chaustone
constant	20%	float	
int	25%	double	2%
char	13%	pointer	16%
short	1%	struct member	9%
long	1%	array member	13%

〈표 8〉 Chaustone의 지역성
〈Table 8〉 Locality of Chaustone

구성 요소	Chaustone	구성 요소	Chaustone
local variable	14%	global variable	86%
compound statement length		9	

(solmono.cse.cau.ac.kr), 및 인터넷 유즈넷(news) 그룹 comp. benchmark의 20여명의 회원들에 의해 측정된 결과이다. 이렇게 측정된 실행 빈도(Ei)를 구성 요소들의 통계 결과에 (식 4)를 적용시켜 병합한 결과는 각각 〈표 5〉에서 〈표 8〉까지와 같다. 이 결과 분포가 Chaustone의 분포가 된다.

Chaustone은 4개의 함수에 걸쳐 총 100개의 문장이 분포하며, 이 4개의 함수가 1000회 수행된 동안의 시간을 측정한다. 4개의 함수 중 1부터 999까지의 합을 구하는 function_1과 이진 탐색이 포함되어 있는 function_2가 의미 있는 연산을 하고 있으나, 실제로 그 결과를 사용하지는 않는다. 이러한 의미 없는 연산의 반복은 합성 벤치마크(synthetic benchmark)의 특징중 하나이다[5].

최종 결과 수치는 Dhrystone[6] 방법에 의거해서 다음과 같이 계산한다.

$$CHAUSTONE = \frac{60 * 1000}{(\text{총 수행시간})} \text{ Chaustone/sec} \quad (5)$$

따라서, CHAUSTONE 결과값이 클수록 빠른 CPU임을 나타낸다.

3.2 Mchaustone

Chaustone은 하나의 프로세스로 이루어져 있으므로, 그 자체로는 다중 처리기의 성능을 정확히 측정할 수 없다. 멀티쓰레드(multi-thread)를 지원하지 않는 유닉스 계열의 운영체제에서는, 하나의 프로세스는 하나의 CPU로만 로드되어, 시스템 자원을 충분히 사용하지 못하기 때문에, 모든 CPU의 성능을 종합적으로 측정할 수 없다. 따라서, UNIX 운영체제를 사용하는 다중 처리기 시스템의 성능 측정을 위해서는 기본적으로 여러개의 프로세스로 구성된 벤치마크가 필요하다. 본 연구에서는 Chaustone을 여러개 fork 함으로써 다중 처리기 시스템에 대한 성능 측정을 수행하도록 한다. Mchaustone이 수행하는 작업은 다음과 같다. 우선 하나의 Chaustone을 실행시킨다. 이 때, Chaustone은 하나의 CPU에서만 수행된다. 이 때의 실행 결과를 R(1, c)라 하자. 단, R(n, c)는 c개의 CPU를 가지는 컴퓨터 시스템에서 n개의 Chaustone을 실행시켰을 때의 결과이다. 두번째 작업으로 CPU 갯수만큼의 Chaustone을 동시에 실행시킨다. 위의 두 가지 작업의 결과로 다중처리기의 효율 E를 다음과 같이 정한다.

$$E = \frac{R(1, c)}{R(c, c)}$$

그러나, 다중처리기 시스템은 CPU 갯수 이상으로 프로세스가 수행되면, 스케쥴러(scheduler)등의 영향

을 받게 된다. 본 연구에서는 이들의 성능 측정을 위해 실행되는 Chaustone의 갯수를 증가시킨다. 이 때, 다중처리기 시스템의 효율은 다음과 같다.

$$E(n)=\frac{R(1,c)}{R(c,c)}$$, 단 n은 Chaustone 프로세스 갯수이다.

Daniel Tabak의 연구[11]에 의하면, Symmetry Whetstone과 Symmetry Dhrystone에 의한 성능 평가 결과가 CPU 갯수의 증가에 따라서 linear한 결과를 나타냈으므로, 이에 근거해서 Mchaustone 에서도, CPU 갯수의 2배만큼의 Chaustone을 실행시키는 것으로 다중처리기 시스템의 효율을 결정할 수 있다. 따라서, 다중처리기 시스템의 벤치마크 값은 아래와 같다.

$$MCHAUSTONE = c * R(1, c) * E(2c) \quad (6)$$

단, Mchaustone은 모든 CPU에서 동일한 Chaustone 프로세스를 실행하므로, CPU 유형이 동일한 대칭형 다중 처리기 시스템의 성능을 측정할 때에만 유효한 결과를 출력할 수 있다.

3.3 Fhsstone

Fhsstone은 화일 시스템의 성능 측정을 위해서 구현된 벤치마크 프로그램이다. Fhsstone은 Howard[8]의 제안에 따라서 구현된다. 화일 시스템의 성능을 측정할 수 있는 벤치마크를 만들기 위한 Howard의 제안은 먼저, 데이타 화일들을 복사할 디렉토리를 만들고, 둘째, 만들어진 디렉토리로 데이타 화일들을 복사하고, 셋째, 복사가 잘 되었는지 디렉토리를 검사하고, 넷째, 모든 데이타 화일을 한번 순차 접근하고, 다섯째, 메이크(make)를 한다. Howard의 제안에 따르면, 화일 시스템과 관련된 대부분의 연산들을 수행하게 된다. 그러나, 위의 제안은 화일 시스템의 속도에 결정적 영향을 미치는 임의 접근에 대한 연산이 고려되지 않는다. 이를 해결하기 위하여 단계 5에서는 임의 접근을 반복 실행한다. 임의 접근의 횟수는 Ousterhout[6]의 연구에 기초하여, 순차 접근 횟수의 1/9 회이다.

3.3.1 사용자 화일의 크기 분포

화일의 크기에 따라서 접근시간에 차이가 날 수 있으므로, 화일 시스템의 성능 측정을 위한 벤치마크에는 데이타 화일의 크기 및 그에 따른 분포가 중요한 구성 요소가 된다. Fhsstone의 데이타 화일의 크기 및 분포는 사용자가 실제 접근하는 분포와 동일하도록 구현된다. Fhsstone의 데이타 화일에 대한 접근을 실제 사용자의 화일 시스템으로의 접근과 동일하도록 하기 위하여 실제 화일의 크기 및 갯수의 분포와 동일한 화일의 분포로 데이타 화일을 만들어야 한다. 이를 위해서, 현재 사용 중인 컴퓨터 시스템의 각 사용자 디렉토리 이하의 화일들에 대하여 각각의 크기와 디렉토리의 갯수를 측정하였다. 여기에서 측정에 사용된 컴퓨터 시스템은 중앙대학교 전산센터의 주전산기 II (ticom.cc1.cau.ac.kr)와 중앙대학교 공과대학 컴퓨터 공학과의 Solbourne 5/600 (solmono.cse.cau.ac.kr), 그리고 인터넷 유즈넷(news) 그룹인 comp.benchmark에서 임의로 추출된 20여명의 회원들 각자가 사용하고 있는 시스템을 대상으로 했다. 측정 결과, 사용자 디렉토리 이하의 부디렉토리에서 크기가 10 Kbytes 이하인 화일들의 크기별 분포는 〈표 9〉와 같다.

〈표 9〉 10 Kbyte 이하의 크기를 가지는 화일의 크기별 분포
〈Table 9〉 Distribution of files under 10 Kbytes size

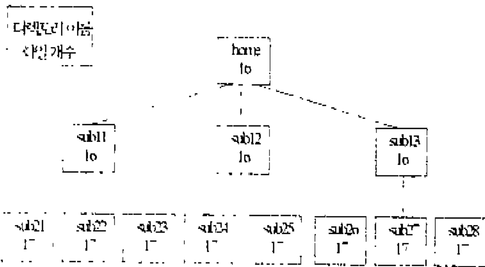
크기(Kbytes 미만)	1	2	3	4	5	6	7	8	9	10
비율(%)	31	17	8	6	3	3	2	1	1	1

〈표 8〉에서 10 Kbyte 이하의 화일이 전체 화일의 80% 정도를 차지하며, 특히 1 Kbyte 이하의 화일이 아주 많음을 알 수 있으나, 측정 결과 1 Kbyte이하의 화일들은 거의 고른 분포를 나타내었으므로 다시 분류하지 않는다. 평균적으로 한 디렉토리는 2. 8개의 부디렉토리를 포함하고 있으며, 부디렉토리를 포함하는 디렉토리는 16개의 화일을, 부디렉토리가 없는 디렉토리는 17개의 화일을 포함한다. 부디렉토리를 가지는 디렉토리와 부디렉토리를 갖지 않는 디렉토리 사이에서 화일 크기에 따른 분포는 거의 차이를 나타내지 않았다.

3.3.2 Fhsstone의 데이타 화일 분포

Fhsstone은 총 200 개의 데이타 화일을 가지며, 각 화일들의 크기에 대한 비율은 〈표 8〉과 동일하다. 이

는 본 벤치마크의 근본적인 목표인 사용자의 실제 사용 환경과 동일한 환경 제공의 측면을 만족하기 위해서이다. 디렉토리의 구조와 각 디렉토리가 포함하는 데이타 화일의 갯수는 (그림 2)와 같다. 단, 각 디렉토리 화일의 크기에 따른 분포는 임의적이다. 즉, 모든 데이타 화일들은 크기에 관계없이 임의로 분산되어 있다.



(그림 2) 데이타 화일의 갯수 분포와 디렉토리 구조
(Fig. 2) Disribution of data files and directory structure

Record id (4 Bytes)	Null Data Field

(그림 3) Fhsstone에 사용되는 데이타 화일의 형식
(Fig. 3) Data file Format of Fhsstone

Fhsstone의 데이타 화일들은 (그림 3)과 같은 형태의 레코드들의 연속으로 이루어져 있다.

화일 형식에서, 레코드 식별자는 임의 접근시에 레코드 지정을 위해 사용되며, 널 데이타 항의 크기는 Fhsstone 설치시 사용자로부터 1에서 10까지의 수를 입력받을 수 있으며, 이때 널 데이타 항의 크기는 입력받은 값에 따라서 4, 12, 28, 60,⋯,4092 Bytes가 되어야 한다. 따라서 널 데이타 항의 크기를 수식으로 나타내면 (식 7)과 같다. 여기서 n은 사용자로부터 입력받은 값이다.

$$4*(2^n-1) \text{ Bytes (단, n} > 0) \quad (7)$$

따라서, 이 데이타 항이 길어질수록 Fhsstone의 실제 실행시간이 작아지며, 이는 Fhsstone의 실행시간 조절을 위해 사용될 수 있다. 총 데이타 화일의 크기는 2632 Kbytes이며, 한 레코드의 크기는 다음과 같다.

$$4+4*(2^n-1)=2^{n+2} \quad (8)$$

그러므로, 총 레코드의 갯수는 $\frac{2632Kbyte}{2^{n+2}}$ 가 되며, 이를 계산하면 $\frac{673792}{2^n}$ 개이다. 따라서 임의 접근 횟수는 총 레코드 갯수의 1/9인 $\frac{74866}{2^n}$ 회이다. 단, 임의 접근을 최소 1회 이상 수행하여야 하므로 다음과 같다.

$$\frac{74866}{2^n} > 1$$

$$n < \log\left(\frac{74866}{2}\right) = 10.530... \quad (9)$$

따라서, $0 < n \leq 10$의 범위를 가져야한다.

3.3.3 Fhsstone의 데이타 접근 방법

임의 접근을 모든 순차 접근 전이나 후에 일괄적으로 수행하는 것과, 순차 접근 중에 임의 접근을 동시에 수행하는 실행시간에 차이가 날 수 있다. 이 차이는 디스크 버퍼의 성능에 의해 좌우된다. Fhsstone은 이 두가지 모두에 대해서 작업하고, 각각의 결과를 출력한다. 이때, 순차 접근 후 임의 접근의 결과는 최대 성능을 나타낼 수 있고, 순차 접근중 임의 접근의 결과는 평균 성능을 나타낼 수 있다.

Fhsstone은 크게 두가지 단계로 작업한다. 그 중, 첫 번째는 모든 데이타 화일을 순차 접근 한 후에, 전체 레코드 중의 1/9개의 레코드를 임의 접근하는 것이다. 두 번째 작업은, 전체 데이타 화일에 대한 순차 접근과, 전체 레코드 중의 1/9개의 레코드에 대한 임의 접근을 동시에 수행하는 것이다. 이때, 순차 접근은 순차 접근 프로세스(SAP)에 의해서 이루어진다. SAP은 Fhsstone의 설치시에 지정된 화일과 레코드에 의해서 지정된 레코드를 순차적으로 접근한다. 임의 접근은 임의접근 프로세스(RAP)에 의해서 이루어진다. RAP은 난수 발생기로부터 난수를 얻어서, 그 난수가 지정하는 데이타 화일의 레코드를 접근한다. 순차 접근과 임의 접근을 조화시키기 위하여, 즉, 임의 접근을 순차 접근의 전 구간에 고르게 분포시키기 위하여 SAP은 모두 9개의 프로세스로 나뉘어 지며, 각각은 동시에 실행되지 않으며 각 SAP은 내부적으로 정해진 데이타 화일을 순차 접근한다. 하나의 RAP은 각 SAP들과 동시에 실행되며, 난수 발생기에 의한 난

수에 의해 지정된 레코드를 접근한다. Fhsstone은 위와 같은 방법으로 화일 시스템에 대한 연산을 수행함으로써, 특히, 순차 접근과 임의 접근의 횟수를 SAP과 RAP에 의해 고려하므로 디스크 버퍼의 효율이 고려된 결과를 출력할 수 있다.

4. 실행 결과 및 고찰

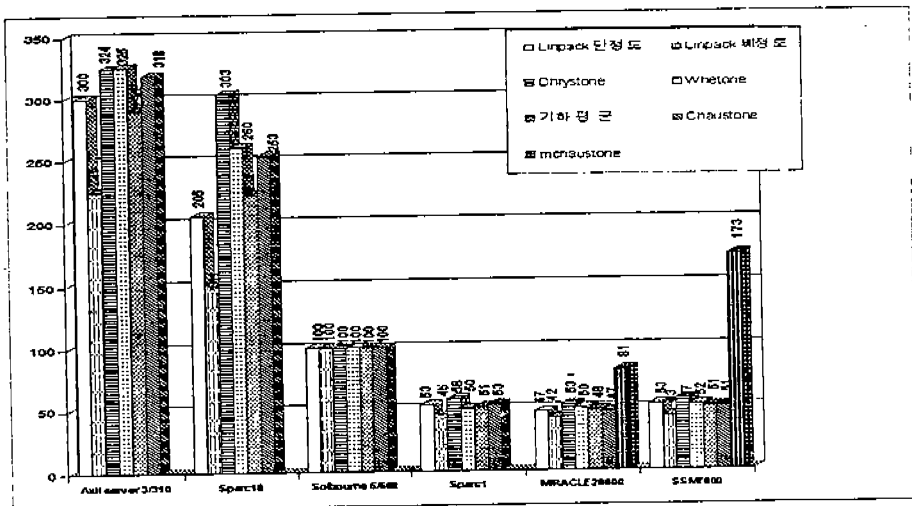
벤치마크는 컴퓨터 시스템의 성능을 측정하는 도구이다. 따라서, 가장 좋은 벤치마크는 대상 컴퓨터 시스템의 성능을 정확하게 측정하는 것이다. 그러나, 컴퓨터 시스템의 정확한 성능을 알 수 없으므로, 다른 벤치마크들의 실행 결과와의 유사성으로 벤치마크를 평가한다. 벤치마크의 결과로 컴퓨터 시스템의 성능을 결정하는 방법은 Fleming[10]의 연구에 잘 나타나 있다. 그의 연구는 컴퓨터 시스템의 벤치마크 결과의 조화평균으로 그 컴퓨터 시스템의 성능을 결정하는 것이다. Axil, Sparc 10, Solbourne 5/600, Sparc 1 및 Ticom 등 5대의 컴퓨터 시스템에서 Chaustone 및 Mchastone과, 잘 알려진 벤치마크인 Dhrystone, Whetstone, 200×200 배열을 가지는 단정도형 Linpack, 200×200 배열을 가지는 배정도형 Linpack을 실행한 결과와 그들의 조화 평균을 비교하

〈표 10〉 Chaustone과 다른 벤치마크들 실행결과
〈Table 10〉 The execution result of Chaustone and other benchmarks

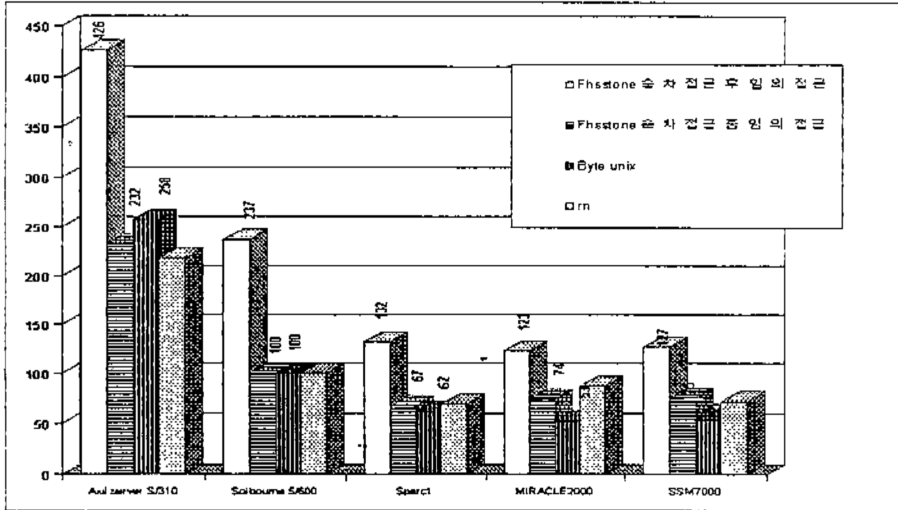
시스템 / 벤치마크	Axil	Sparc 10	Solbourne 5/600	Sparc 1	Ticom (2 CPUs)	평균과의 표준 편차
Linpack-s	300	205	100	53	47	19.8
Linpack-d	225	151	100	45	42	96.6
Dhrystone	324	303	100	58	53	88.3
Whetstone	325	260	100	50	50	51.7
Chaustone (Mchaustone)	318	253	100	53	47 (81)	41.8
평균	290	222	100	51	48	–

〈표 11〉 Fhsstone과 Byte 실행 결과
〈Table 11〉 The execution result of Fhsstone and Byte

시스템 / 벤치마크		Axil	Solbourne 5/600	Sparc 1	Ticom (2 CPUs)
Fhsstone	순차 접근 후 임의 접근	426	237	132	123
	순차 접근 중 임의 접근	232	100	67	74
Byte		258	100	62	53



(그림 4) Chaustone과 다른 벤치마크들 실행 결과
(Fig. 4) The execution result of Chaustone and other benchmarks



(그림 5) Fhsstone과 다른 벤치마크들 실행 결과
(Fig. 5) The execution result of Fhsstone and other benchmarks

면 〈표 10〉 및 (그림 4)와 같다. 이 표를 보면 각 벤치마크들에 대한 평균값과의 표준편차가 Chaustone이 가장 적다는 것을 알 수 있다. (Linpack-s는 2.4절에서 이미 언급한 대로 CPU 성능 측정만을 위해서는 사용될 수 없다.) 즉, CPU의 성능을 측정하기 위하여 여러 벤치마크를 실행할 필요없이, Chaustone만을 실행하면 실제 성능에 유사한 결과를 얻을 수 있다는 것을 뜻한다. Byte 벤치마크와 Fhsstone을 Axil과 Sclbourne, Sparc1에서 실행한 결과를 비교하면 〈표 11〉 및 (그림 5)와 같다.

5. 결 론

기존의 여러 벤치마크 프로그램들은 현재 워크스테이션과 미니 컴퓨터에서 많이 운영되는 UNIX를 탑재한 컴퓨터 시스템에서 사용하기에는 여러 문제점들을 가지고 있었다. 본 논문에서는 이러한 문제점들을 해결하기 위하여 Chaustone, Mchaustone, Fhsstone으로 이루어진 벤치마크 프로그램인 CMFstone을 구현하였다.

Chaustone은 한 CPU의 성능을 측정하기 위한 벤치마크 도구로써, 여러 응용 프로그램들의 문장 형식, 연산자, 피연산자, 지역성 등의 분포를 실행 빈도에 의한 가중치를 부여함으로써 구현되었다. Mchaustone은 대칭형 다중 처리기 시스템의 성능을 측정하기 위한 벤치마크 도구로써, Chaustone을 여러개 실행하여 다중처리기의 효율을 구하고, 이를 CPU의 갯수와, 한 CPU의 Chaustone 값에 곱함으로써 다중 처리기의 성능을 측정한다. Fhsstone는 화일 시스템의 성능을 측정하기 위한 벤치마크 도구로써, Howard의 제안을 바탕으로 하여, 화일 시스템과 관련된 여러 연산을 수행하며, 특히, 순차 접근과 임의 접근의 횟수를 고려하므로 디스크 버퍼의 효율도 측정할 수 있다. 여러 컴퓨터 시스템에서 CMFstone을 실행한 결과, 컴퓨터 시스템의 성능을 측정하기 위해서, 기존의 여러 벤치마크를 실행하고 다시 그들의 조화 평균을 구하여 측정한 컴퓨터 시스템의 성능을 결정할 필요없이, CMFstone을 실행하는 것만으로 그와 유사한 결과를 구할 수 있었다.

향후 연구방향으로 비대칭형(asymmetric) 다중 처리기 컴퓨터 시스템의 성능을 측정할 수 있는 벤치마크 프로그램과 CPU간의 통신 방법인 공유 메모리(shared memory), 메시지 전달(message passing) 방식 각각에 대한 메모리 시스템의 성능 측정을 위한 벤치

마크 프로그램과 터미널, 프린터 등의 입·출력 장치와 네트워크의 성능 측정을 위한 벤치마크 프로그램이 필요하다. 마지막으로, 각각의 벤치마크 결과값을 분석하여 전체 컴퓨터 시스템의 성능을 측정할 수 있는 방법이 연구 되야 한다.

참 고 문 헌

[1] A. Sheeman, "Performance Analysis: An Introduction," Technical Report 89. 3, Part No. 19150, Mar. 1989.

[2] T. Bemmerl, "Distributed Monitoring Systems-A Basis for General Purpose Distributed Multiprocessors," Pannel Section, Proc. of the Int'l Conf. on Distributed Computing Systems, pp. 380, May 1991.

[3] P. C. Bates, "Effective Instrumentation is the Key to Effective Monitoring," Pannel Section, Proc. of the Int'l Conf. on Distributed Computing Systems, pp. 379, May 1991.

[4] "SPEC 벤치마크," KUUG Newsletter, Vol. 5, No. 6, pp. 1~4, 1990.

[5] Thomas M. Conte and Wen-mei W. Hwu, "Benchmark Characterization," IEEE Computer, pp. 48~56, Jan. 1991.

[6] Reinhold P. Weiker, "DHRYSTONE: A SYNTHETIC SYSTEMS PROGRAMMING BENCHMARK," Communications of the ACM, Vol. 27, No. 10, pp. 1013~1030, Oct. 1984.

[7] Reinhold P. Weiker, "An Overview of Common Benchmarks," IEEE Computer, pp. 65~75, Dec. 1990.

[8] Howard, J. H. et. al., "Scale and performance in a distributed file system," ACM transaction on computer systems, Vol. 6, No. 1, pp. 51~81, 1988.

[9] Ousterhout, J. K. et. al., "A trace-driven analysis of the UNIX 4. 2 BSD file system," Proceedings of Tenth ACM symposium on Operating Systems Principles, Orcas Island, Wash., pp. 15~24, 1985.

[10] Fleming, Wallace, "How not to lie with statistics: The correct way to summarize benchmark results," Communications of ACM, Vol. 29, No. 3, pp. 218~221, Mar. 1986.

[11] Daniel Tabak, Multiprocessors, Prentice Hall, NJ, pp. 140~150, 1990.

이 시 진

1990년 중앙대학교 컴퓨터 공학과 졸업(학사)
1992년 중앙대학교 대학원 컴퓨터 공학과 (공학석사)
1992년~현재 중앙대학교 대학원 컴퓨터 공학과 박사과정
관심 분야: 분산 시스템, 멀티미디어 시스템, 객체지향 시스템, 성능평가

박 성 욱

1994년 중앙대학교 컴퓨터 공학과 졸업 (학사)
1996년 중앙대학교 대학원 컴퓨터 공학과 (공학석사)
1996년~현재 LG 정보 통신 근무중
관심 분야: 운영체제, 성능 평가, 네트워크, 멀티미디어

권 혁 인

1983년 중앙대학교 전자계산학과 (학사)
1992년 중앙대학교 대학원 전자계산학과 (석사)
1994년 프랑스 Paris 6 대학 전산학과 (전산학 박사)
1995년~현재 중앙대학교 경영학과 조교수
관심 분야: 컴퓨터 네트워크, 분산 처리 시스템, 네트워크 관리