목 차

1. 서론: 고성능 컴퓨팅

1.1. 고성능 컴퓨팅

고성능 컴퓨팅이란, 넓게는 대량의 계산을 보다 빠르게 실행하는 컴퓨터 시스템을 구축하기 위한 하드웨어와 소프트웨어 기술을 통칭하는 것이며, 좁게는 슈퍼컴퓨터와 슈퍼컴퓨터 상에서 실행되는 소프트웨어를 개발하는데 집중하는 컴퓨터 사이언스의 한 분야이다.

고성능 컴퓨팅(High Performance Computing)에 대한 정의는 슈퍼컴퓨터의 개념이 변하는 것과 같이해 최근 10여 년 동안 매우 많이 바뀌었다. 1988년 월 스트리트 저널에 “Attack of the Killer Micros”라는 제목으로 기사가 나온 적이 있는데, 이 기사에서는 여러 개의 비교적 저렴한 프로세서로 구성된 시스템이 거대한 슈퍼 컴퓨터를 머지않아 퇴물로 만들어 버릴 것이라는 내용이 실려 있었다. 당시, 대당 약 3,000달러 정도하는 개인용 컴퓨터 한대는 0.25 Mflops 정도의 성능을 가지고 있었고 대당 약 20,000달러 정도하는 워크스테이션이 3 Mflops의 성능을 가지고 있었다. 가격이 약 3,000,000달러 정도였던 그 당시 슈퍼 컴퓨터의 성능은 100 Mflops 정도였다. 그래서, 단순히 개인용 PC 400대를 연결해 같이 사용하면 1,200,000달러의 비용으로 슈퍼컴퓨터와 동일한 성능을 내게 될 것으로 생각할 수 있었다.

이와 같은 클러스터링 개념과 더불어 마이크로프로세서 자체의 성능이 지속적으로 발전해 이제는 슈퍼컴퓨터의 성능에 근접하게 되었다. 마이크로프로세서의 성능이 빠르게 발전하게 된 이유는 PC 영역에서는 성능을 향상시키기 위한 기술적 여유공간이 많이 있었던 반면에 1980년대 후반의 슈퍼 컴퓨터 성능은 답보 상태에 있었기 때문이다. 게다가, 슈퍼컴퓨터 회사에서 하나의 기술적 장벽을 어렵게 돌파하면 마이크로프로세서 회사에서는 슈퍼컴퓨터 디자인의 성공적인 요소를 재빠르게 채택하여 불과 수년 내에 이러한 성능 향상을 얻기도 하였다. 또 한 가지 이유로는 끊임없이 성능 향상을 요구하는 거대한 개인용/사업용 컴퓨터 시장의 출현을 생각할 수 있다. 3차원 그래픽, GUI, 멀티미디어, 그리고 게임과 같은 컴퓨터 사용 영역은 이러한 시장 증가의 중대 발전요소였으며, 이로 인해 값싼 고성능 프로세서 개발에 막대한 연구 개발비가 투자되었다. 더 빠르고 더 작은 컴퓨터 개발을 향한 이러한 흐름의 결과로 워크스테이션 제조회사들이 슈퍼컴퓨터 제조사들을 사들이게 되었다. 실제로 Silicon Graphics는 Cray를 사들였고, HP는 1996년에 Convex를 사들였다.

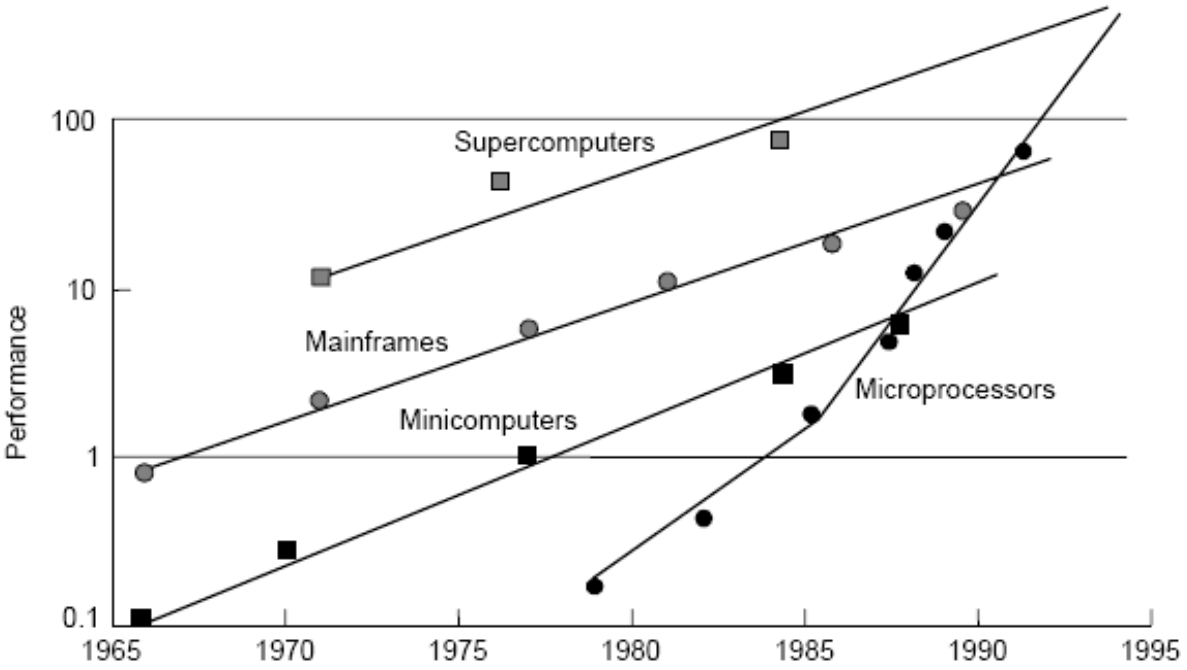


그림 1-1. 컴퓨터 성능의 발전

고성능 컴퓨팅은 우리가 일상적으로 사용하는 데스크톱 컴퓨터에서 대규모의 병렬 처리 시스템까지 넓은 범위의 시스템에 걸쳐 이용된다. 현재 대부분의 고성능 시스템들은 RISC(Reduced Instruction Set Computer) 프로세서를 기반으로 하는 병렬 아키텍처를 가진다. 병렬 아키텍처를 사용하는 주된 이유는 역시 성능 때문이다. 일부에서는 단일 CPU 시스템이 앞으로도 충분히 빨라질 것이기 때문에 프로그램 작성을 위해 부가적인 기술을 요구하는 최신의 병렬 아키텍처를 이해하는 것이 불필요한 일이라고 주장한다. 그러나, 지금까지 컴퓨팅 분야에서 저가의 단일 프로세서 성능이 비록 수천 배 이상 증가해 왔지만 이와 같은 프로세서를 수천 개 묶어 사용함으로써 수백만 배의 성능을 얻을 수 있다는 것은 역시 흥미로운 일일 것이다. 고성능 컴퓨팅을 위한 기반 요소들의 값은 더욱 싸지고 있고 많은 프로세서를 연결해 사용함으로써 얻는 이득은 점점 커지고 있다. 현재 512개의 프로세서를 연결한 병렬 시스템의 성능보다 더 빠른 단일 프로세서 시스템이 미래에는 나오게 될 것이다. 그렇지만 다시 이와 같은 새로운 프로세서를 512개 연결한 병렬시스템으로부터 얼마만큼의 성능 이득을 얻게 될지 상상해 보는 것은 어려운 일이 아니다.

사용하는 컴퓨터 시스템의 성능을 최대한 이끌어 내기 위해서는 시스템의 성능에 직접적인 영향을 주는 컴퓨터 아키텍처에 대해 이해할 필요가 있다. 컴퓨터 기술이 발전해감에 따라 고성능 시스템들의 아키텍처는 시간이 갈수록 복잡해져 가고 있으나 이를 사용하는 사용자들이 아키텍처에 대한 이해와 지식 없이 프로그램을 작성함으로써 실제로 그 시스템이 가지고 있는 성능의 10분의 1에도 미치지 못하는 성능으로 계산을 수행하는 경우가 많다.

컴퓨터를 이용한 실험 및 연구를 좀 더 효율적이고 빠르게 수행하기 위해서는 사용하는 시스템의 구조 및 작동 방식에 대한 기본적인 지식과 이해가 필요하다. 그리고 이러한 이해를 바탕으로 사용하고자 하는 하드웨어상에서 알고리즘이 최대의 성능을 낼 수 있도록 구현하게 되는데 이 과정에서 성능 최적화 기법들이 요구된다.
성능 최적화의 관점에서 프로그래머 에게는 아래와 같은 것들이 필요하게 된다.

- 현대 컴퓨터 아키텍처에 대한 기본적인 이해. 컴퓨터 공학에 대한 고급 수준의 지식이 필요한 것은 아니라 기본적인 용어들을 이해할 수 있는 수준 정도가 필요하다.
- 벤치마킹 또는 성능 측정에 대한 기본적인 이해. 이를 통해 최적화의 성공 또는 실패 여부를 정량화할 수 있으며, 이렇게 얻은 정보를 성능향상을 위해 이용할 수 있다.

개인용 컴퓨터가 가지는 성능 이상의 것이 요구되는 수준에서 고성능 컴퓨팅은 더욱 중요한 역할을 하게 될 것이다. 고성능 컴퓨터들의 상위 계층으로 올라 갈수록 그 시스템에 최적화되도록 코드를 맞추기 위해 언어 확장, 라이브러리 호출, 컴파일러 지시어 등과 같은 새로운 기법들을 배워야 할 필요가 있다. 그렇지만 이러한 기법들을 많이 사용할수록 코드의 이식성은 떨어지게 된다.

본 교재는 컴퓨터 공학자 수준이 아닌 단지 실험과 연구의 수단으로서 컴퓨터를 사용하는 고급 사용자(high-end)들을 대상으로 하며 또한 수행하는 코드의 성능을 최대로 높이고자 하는 사람들을 위한 것이다. 고성능 시스템 아키텍처의 구동방식에 대한 이해를 바탕으로 코드 성능을 최대로 높이기 위해 어떤 기법들을 사용할 수 있는지에 대해 기술하였다.

1.2. HPC 프로세서 아키텍처

주요 HPC 프로세서 아키텍처의 기본적인 특성과 이것이 프로그램 코드의 성능에 어떻게 영향을 미치는지에 대해 알아본다. 사용자는 현재의 고성능 시스템 아키텍처에 대해 이해함으로써 어떤 종류의 성능 데이터를 수집해서 어떻게 분석할 것 인가를 알 수 있게 된다. HPC 프로세서 아키텍처가 발전해온 역사를 이해함으로써 사용자는 과거에 작성된 코드들을 이전, 유지하고 그 성능을 향상시키는 것에 도움을 받을 수 있을 것이다.

1.2.1. 아키텍처의 발전

무어는 1965년에 마이크로 프로세서의 성능이 18개월에 약 두 배씩 향상될 것이라 예측했다. 이 무어의 법칙은 자연의 법칙이 아니라 공학의 발전 정도에 대한 관측을 표현한 것이지만, 지난 50여 년간 놀랍게도 잘 맞아 왔다. 프로세서의 발전만큼 놀라운 발전이 저장장치 분야에서도 진행되었는데 메모리 크기와 그 비용이 지수 함수적으로 떨어져 왔다. 그렇지만, 메모리에 접근하기 위해 기다리는 지연시간(latency)은 그와 같은 비율로 떨어지지 않았다. CPU 사이클이 낭비된다는

관점에서 봤을 때 메모리 지연시간에 대한 상대적인 CPU 클럭 수의 증가는 메모리 접근 비용(cost)의 폭등을 의미하는 것이다. 이러한 CPU와 메모리 사이의 성능 불균형이 컴퓨터 아키텍처를 보다 깊고 복잡한 메모리 계층구조를 가지는 "load-store" 아키텍처로 진화되게 하였다.

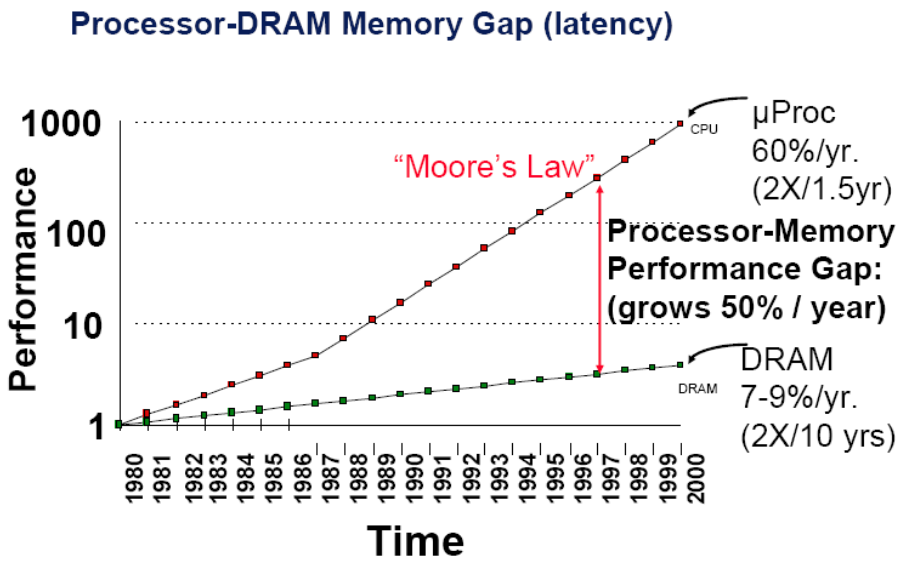


Figure credit: "A Case for Intelligent RAM: IRAM," by David Patterson et. al.

그림 1-2. CPU와 메모리 성능의 차이

위의 그림은 메모리 성능과 CPU 성능의 발전을 비교해 보여준다. 이와 같은 비교가 프로그램의 성능에 중요하게 영향을 미치는 점은 알고리즘 개발과 그것을 코드에서 구현할 경우에 항상 메모리 지연시간을 최소화 하도록 최적화해야 한다는 것이다. 메모리 접근을 가능한 한 효율적으로 하고 메모리 계층구조 디자인을 이용해서 이러한 최적화를 수행할 수 있다.

1.3. HPC 프로세서

프로세서 패밀리는 높은 수준의 디자인 특성을 공유하는 마이크로프로세서 집합을 말한다. 지난 30여 년간의 프로세서 패밀리는 크게 4가지로 구분해 볼 수 있다.

프로세서 패밀리	명령어 집합 구조(ISA: Instruction Set Architecture)	프로세서
벡 터	Cray Convex NEC	Cray C90 Convex C-4 NEC SX-5
CISC	DEC VAX	VAX-11/780

Complex Instruction Set Computer	Intel 80x86(IA-32)	Intel Pertium Pro
RISC Reduced Instruction Set Computer	HP PA-RISC SGI MIPS DEC Alpha Sun SPARC IBM PowerPC	PA-8600 MIPS R1000 Alpha21264 Sun UltraSparc-3 IBM Power4
VLIW Very Long Instruction Word	Multiflow Cydrome Intel IA-64	Multiflow Trace Cydrome Cydra 5 Intel Itanium

표 1-1. 프로세서 패밀리

1.3.1. Complex Instruction Set Computer (CISC)

1970년대 발표돼 지금까지 사용되고 있다. 디자인 목표는 고급 언어로 개발된 코드가 가능한 한 최소의 어셈블리 언어 명령어로 번역되도록 어셈블리 명령어 집합을 정의하는 것이다. 결과로 많은 명령어 타입이 있고 메모리에 접근하는 명령어 타입도 서로 다른 것이 많이 있다. CISC 명령어 집합(ISA)은 Fortran이나 C와 같은 고급언어의 primitive가 수행하는 기능과 유사한 역할을 하는 강력한 primitive들로 구성돼 있다.

실행 unit이 레지스터에 있는 데이터에 대해 연산을 수행하는 과정을 통해 CISC 아키텍처의 디자인 특성을 알아보자.

아래 그림은 일상적인 컴퓨터의 메모리 구조를 나타낸 것이다. 그림에서 주 기억장치(Main Memory)의 위치가 (1,1)에서 (6,4)까지 구분돼 있다. 실행 unit은 모든 계산을 수행하는 부분이지만, 6개의 레지스터에 올라온 데이터에 대해서만 계산 수행이 가능하다. 메모리 영역 (2,3)과 (5,2)에 저장된 두 개의 데이터에 대해 곱셈을 수행해서 그 결과를 다시 메모리 영역 (2,3)저장하는 계산을 수행하는 과정을 살펴보자.

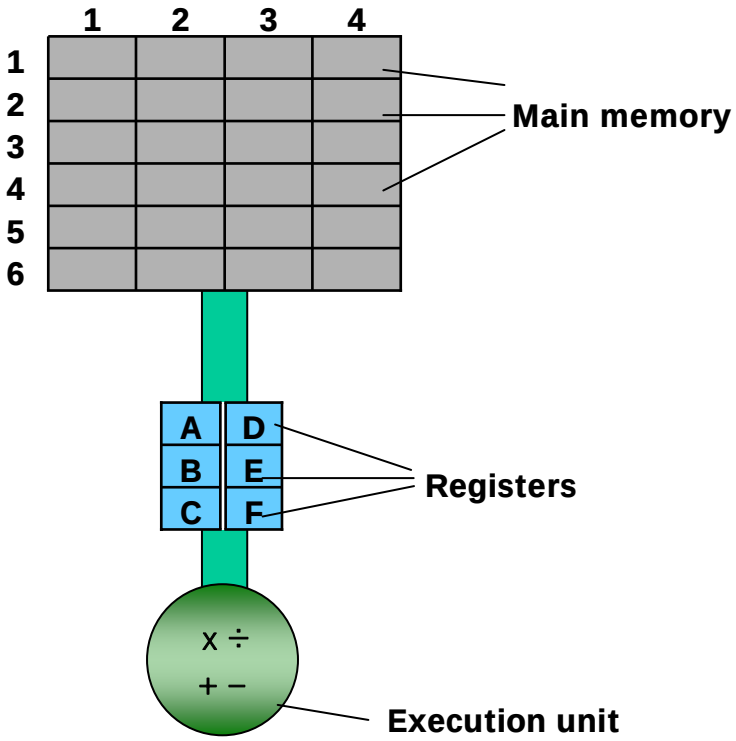


그림 1-3. 컴퓨터의 메모리 구조

CISC는 일련의 연산을 이해하고 수행할 수 있는 하드웨어(CISC 프로세서)를 통해 가능한 작은 수의 어셈블리 라인으로 작업을 수행하려고 한다. 위에서 언급한 계산을 수행하기 위해 CISC에 준비된 명령어 MULT를 가정하면, 명령

MULT 2:3, 5:2

에 의해 모든 작업(MULT 명령어는 레지스터로 두 값을 로드하고 실행 unit에서 연산을 하고 그 결과를 적절한 레지스터에 저장한다.)이 이루어 진다. 이때의 MULT가 complex instruction이 된다. 이 명령어는 컴퓨터 메모리 뱅크에 직접 작용하고 프로그래머가 “load”나 “store” 같은 함수의 호출을 명시적으로 해 줄 필요가 없다. 이와 같은 방식은 고급 언어의 명령어와 아주 유사하다. 이를테면 기억장치 주소 2:3의 값을 a로 나타내고 주소 5:2의 값을 b로 나타내면 위의 명령은 a=a*b와 그 의미가 동일하다.
CISC 시스템의 장점은 고급 언어 문장을 어셈블리로 번역하는 과정에서 컴파일러의 역할이 최소화 된다는 것이다. 또한 코드의 길이가 상대적으로 짧고 명령어 저장을 위한 메모리 공간이 적게 필요해진다. CISC 아키텍처는 하드웨어에 직접 building 되는 Complex instruction을 강조한다.

1.3.2. RISC

RISC 아키텍처는 1980년대에 소개 되었다. RISC 프로세서 디자이너들의 목표는 빠른 clock rate을 가진 고성능 단일 칩 프로세서를 개발하는 것이었다. 또 다른 목표는 CPI(Cycles Per Instruction) 값을 낮춰 가능한 많은 명령어들을 파이프라인 하는 것이었는데 이러한 목표들은 Load/store set 아키텍처, 분기 예측(branch prediction), 슈퍼스칼라 프로세싱 등이 결합된 형태로 구현돼 왔다.

RISC 프로세서는 하나의 clock cycle에 실행될 수 있는 간단한 명령어를 사용한다. 위의 예에서 CISC 명령어 MULT는 LOAD(메모리에서 레지스터로 데이터 이동), PROD(레지스터내의 두 연산수에 대해 곱셈), STORE(레지스터에서 메모리로 데이터 이동)로 구분된다. 전체 계산이 완료되기 위해서는 다음과 같이 4줄의 어셈블리 코드가 필요하게 된다.

```
LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A
```

예에서 보듯이 RISC는 연산 수행에 있어 비 효율적으로 보여진다. 코드가 상대적으로 길어지고 어셈블리 수준의 명령어 저장을 위해 더 많은 메모리 필요하게 된다. 그리고, 컴파일러는 고급 언어를 이와 같은 코드로 번역하기 위해 더 많은 일을 해야 한다. 그러나, 각 명령어는 1 clock cycle에 실행되어 전체 코드는 다중 cycle이 필요한 MULT 명령어 하나에 의한 실행과 비슷한 시간이 소요된다. Reduced instructions은 하드웨어 공간의 트랜지스터가 complex instruction 보다 적게 요구되어, general purpose 레지스터 공간에 여유가 있으며, 모든 명령어가 같은 시간에 실행되어 파이프라이닝이 가능하다.

RISC에서처럼 LOAD와 STORE 명령어를 분리하면 실질적으로 컴퓨터가 수행해야 할 일의 양을 감소 시킨다. CISC 시스템에서 MULT명령이 실행된 후 프로세서는 자동으로 레지스터를 삭제하게 된다. 이때 만약 삭제되는 연산수 중 하나가 다른 계산에 사용된다면 프로세서는 메모리에서 레지스터로 데이터를 다시 load 해야 하지만, RISC 시스템의 경우 명시적으로 다른 값이 load될 때까지 레지스터에 연산수가 남아 있게 된다.

1.3.3. CISC vs. RISC

CISC 아키텍처의 특징인 길이가 다른 명령어와 실행 시간이 다른 명령어 들은 파이프라이닝이 어렵다. 비용을 줄이고 빠른 clock 속도를 얻기 위해 RISC 구조는 단일 칩에 프로세서를 위치시킨다. 이로 인해 최소화된 명령어 집합이 필요하고 따라서 RISC는 모두 동일한 길이의 상대적으로 소수인 명령어 집합을 가지게 된다.

일반적으로 컴퓨터에서 실행되는 프로그램의 성능을 다음과 같이 표현할 수 있다.

Time/program = time/cycle * cycles/instruction * instructions/program

이와 같은 방식으로 결정되는 프로그램 성능을 좋게 하기 위해 CISC는 명령어당 cycles(CPI)값을 희생하고 프로그램당 명령어 개수를 최소화 하려는 시도를 하고 있는 반면, RISC는 그 반대로 프로그램당 명령어 개수를 희생하는 대신 명령어당 cycles 수를 최소화 하는 것을 목표로 하는 것이다.
CISC와 RISC 아키텍처를 아래 표에 비교 정리해 두었다.

CISC	RISC
Emphasis on hardware Includes multi-clock complex instructions Memory-to-memory: "LOAD" and "STORE" incorporated in instructions Small code sizes, high cycles per second Transistors used for storing complex instructions	Emphasis on software Single-clock, reduced instruction only Register to register: "LOAD" and "STORE" are independent instructions Low cycles per second, large code sizes Spends more transistors on memory registers

표 1-2. CISC 프로세서와 RISC 프로세서의 비교

1.3.4. 최신 스칼라 프로세서 기술

1.3.4.1. CISC and RISC Convergence

최신의 프로세서 기술은 RISC칩이 처음 소개(80년대 초)된 이후로 많이 변했다. 다수의 진보된 기술이 RISC와 CISC 프로세서에서 같이 사용되고 있어 두 기술 사이의 경계가 점차 모호해지고 있다. 프로세서 속도가 빨려져서 CISC 칩은 이제 single clock에 하나 이상의 명령어를 실행할 수 있게 됐고 파이프라이닝이 가능해 졌다. 다른 기술의 향상으로 다수의 트랜지스터를 단일 칩에 넣을 수 있게 돼, RISC 프로세서는 보다 복잡한 CISC-like한 명령어를 가질 수 있게 되었다. RISC 프로세서는 슈퍼스칼라 실행을 위한 여분의 function unit을 사용하는 보다 복잡한 하드웨어를 사용한다. 이로 인해 post-RISC 시대에 대한 논의가 있다. 그러나 여전히 RISC는 중요한 특성을 가지며, 엄격하게 uniform, single-cycle 명령어를 사용한다. 또한 register-to-register, load-store 구조를 유지한다. 명령어 집합의 확장에도 불구하고 RISC 칩은 여전히 다수의 general purpose 레지스터를 가지고 있다.

1.3.4.2. Simultaneous Multi-Threading (SMT)

SMT는 파이프라인으로 다른 스레드의 명령어를 넣어 여러 스레드가 동시에 실행되도록 한다. 이

로 인해 다중 스레드가 프로세스를 향상시키며, 특정 스레드가 어떤 순간 프로세서를 독점하지 않게 된다.

1.3.4.3. Value Prediction

value prediction은 특정 load 명령어가 적용될 값을 미리 예측하는 것이다. Load 명령어가 적용돼 load되는 값은 일반적으로 random이 아니며, 평균적으로 한 프로그램에서 load 명령의 반 정도는 이전 실행에서 사용한 값을 불러오게 된다. Load는 느리면서 자주 실행되는 대표적 명령어이다. Load value가 마지막에 사용한 것과 같을 것이라 예측해서 미리 값을 load 시켜 둠으로써 프로세서 성능을 많이 향상시킬 수 있다.

1.3.5. Vector Processors

벡터 프로세서는 1970년대 후반기에 발표 되었고, 1980년대 고성능 컴퓨팅 분야에서는 대부분 벡터 프로세서를 사용하였다. 1976년 Seymour Cray에서 발표한 Cray1이 성공적인 첫 번째 벡터 프로세서 슈퍼컴퓨터였다. 벡터 프로세서는 벡터 데이터에 대한 연산을 수행한다. 벡터 레지스터, 메모리 파이프라인, 벡터 명령어 등을 사용하는데, 빠르고 좋은 효율을 가지고 있지만 값이 비싸다는 단점이 있다. 벡터 명령어 하나로 여러 데이터에 대한 작업을 수행하므로 명령어 인출 횟수가 작고 많은 파이프라인 단계가 가능하다. 또한 한꺼번에 메모리에 접근해 벡터 데이터를 읽고 쓰기가 가능해 메모리 접근 지연이 작으며 여러 메모리 뱅크에서 동시에 연산수를 제공해 줄 수 있다. 벡터 프로세서 시스템은 자연스럽게 파이프라인 될 수 있는 well-defined 패턴의 데이터에 접근할 때 최적의 성능을 보여준다. 값이 비싸며 제한된 분야에서만 최적의 성능을 보여준다는 한계로 인해 현재는 많이 사용되지 않고 있다.

1.3.6. VLIW

하나의 clock cycle에 여러 명령어를 실행할 수 있도록 하기 위해 슈퍼스칼라 아키텍처는 여러 개의 실행 unit을 이용해 명령어를 병렬 처리한다. Very Long Instruction Word(VLIW) 프로세서는 슈퍼스칼라와는 다른 접근법을 이용해 명령어를 병렬 처리하는 아키텍처이다. VLIW 프로세서는 컴파일러나 전처리기를 통해 병렬 수행 가능한 RISC 명령어들을 긴 명령어 워드에 채우고 긴 명령어를 수행하면 이 명령어들을 병렬로 수행하게 된다. 명령어에 의해 명시적으로 지정된 명령어들 사이에서 하나의 큰 명령어 또는 병렬성을 가지는 고정된 명령어 패킷으로 형식화된 고정된 개수의 명령어를 실행하기 때문에 EPIC(Explicitly Parallel Instruction Computers)으로도 불리어진다. EPIC 아키텍처는 컴파일러가 병렬성에 대한 정보를 하드웨어에 전달해 주는 것을 허용한다.

1.4. HPC 프로세서 구조

현재 HPC 분야의 지배 프로세서는 RISC기반이다. 여기서는 RISC 프로세서의 기반이 되는 파이프라이닝 기술을 기반으로 슈퍼파이프라이닝, 슈퍼스칼라 등에 대해 소개한다. 아울러 메모리 접

근 시간을 최소화 하기 위해 대부분의 컴퓨터 아키텍처에 적용돼 사용되고 있는 메모리 계층구조에 대해 알아 본다. 특히 성능 최적화를 위해 메모리 계층 구조에 대한 이해는 필수적이다.

1.4.1. 파이프라이닝

파이프라이닝은 여러 개의 명령어 실행을 중첩시키는 구현 기술이며, 프로세서를 빠르게 만드는 핵심적인 기술이다.

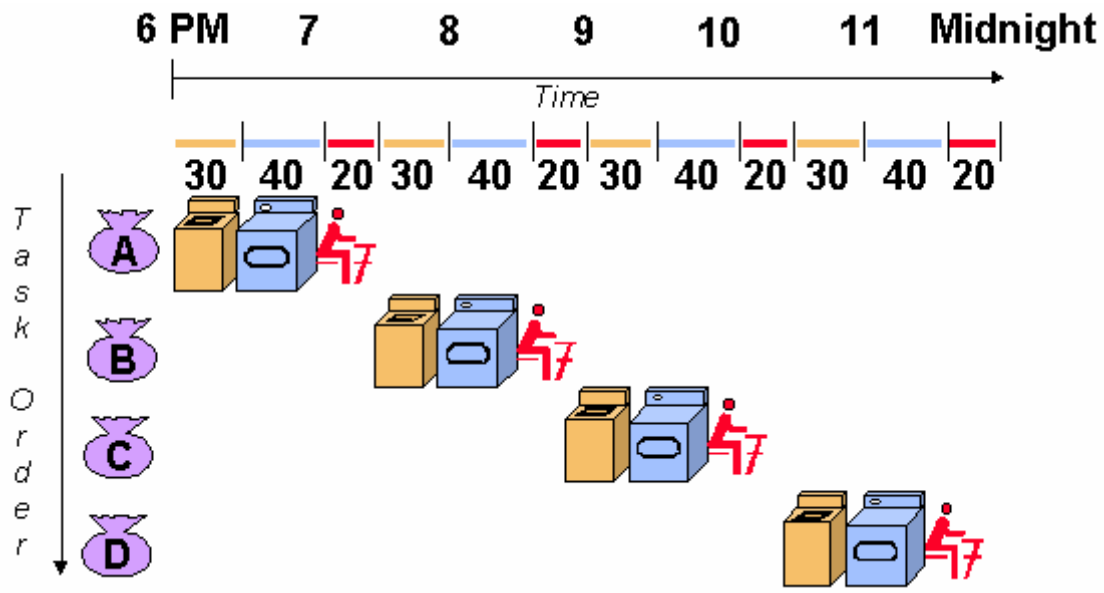


그림 1-4. 파이프라인되지 않은 작업처리

위의 그림은 세탁에 대해 파이프라인되지 않은 방법을 나타낸 것이다. 즉,

1. 세탁기에 세탁물을 넣는다.
2. 세탁이 끝난 세탁물을 건조기에 넣는다.
3. 건조된 세탁물을 꺼내어 테이블 위에 놓고 접는다.

와 같은 과정을 거쳐 A세탁물에 대해 세탁이 끝나면 다시 B세탁물에 대해 동일한 과정을 거쳐 세탁을 하는 것이다. 이 세탁 과정을 파이프라인화 시켜 진행하는 과정을 나타낸 것이 다음 그림이다.

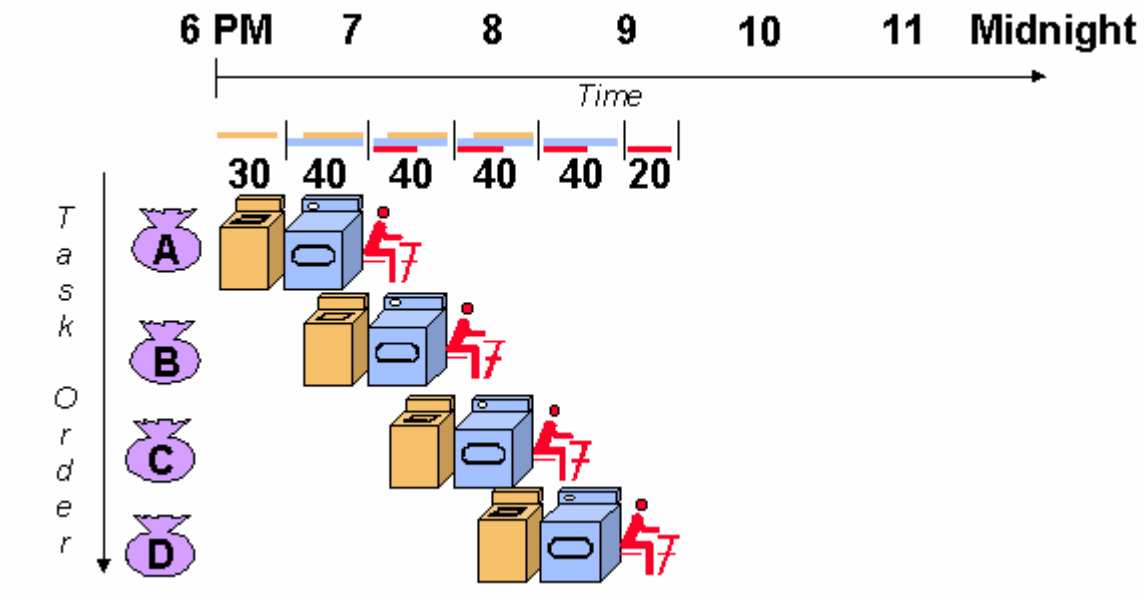
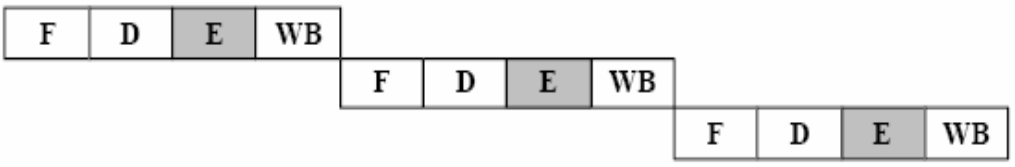


그림 1-5. 파이프라인된 작업처리

이와 같은 원리가 프로세서에 적용되는데 프로세서에서는 명령어 실행을 파이프라인 한다. 모든 프로세서 아키텍처가 그런 것은 아니지만 일반적으로 명령어 처리를 다음과 같이 간단한 파이프라인 단계로 구분해 볼 수 있다.

- instruction fetch cycle(F): 프로세서는 메모리에서 명령어를 꺼내옴
- instruction decode/register fetch cycle(D): 프로세서는 명령어를 해석하고 레지스터를 읽어 들인다.
- Execution/effective address cycle(E): 이전 단계에서 준비된 연산을 수행하거나 load 또는 store 명령의 경우 effective 주소를 계산한다.
- Write-back cycle (WB): 프로세서는 레지스터 파일로 결과를 쓴다.



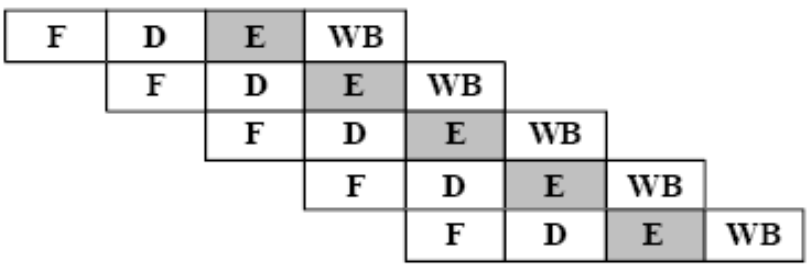


그림 1-6. 컴퓨터 명령어의 파이프라인 처리

하나의 절차가 서로 다른 자원을 사용하는 구분 가능한 단계들로 나눠질 수 있는 경우 파이프라인은 효과적이다. 파이프라인 처리에 의해 명령어 하나의 처리 시간이 짧아지는 것은 아니다. 파이프라인에 의한 명령어 처리는 병렬로 작동하여 단위 시간에 많은 일을 할 수 있다는 것이다. 모든 단계가 거의 같은 시간이 걸리며, 할 일이 충분히 있다면 파이프라이닝에 의한 속도 증가는 파이프라인의 단계 수와 같다.
파이프라인의 길이는 가장 긴 단계에 의해 결정되게 되는데 따라서 같은 길이 명령어를 가지는 RISC 프로세서에서 파이프라인 구현이 유리해진다. 이상적인 RISC프로세서 파이프라인의 각 단계는 1 lock cycle이며 평균 CPI가 1이 되는데, 실제로는 데이터 의존성과 분기 명령어 등에 의해 CPI가 1보다 커지게 된다.

1.4.1.1. 데이터 의존성

다음과 같이 명령어가 앞선 명령어 실행 결과에 의존적인 경우를 말한다.

```
add $r3, $r2, $r1
add $r5, $r4, $r3
```

첫 번째 명령어는 레지스터 r1과 r2의 값을 더한 결과를 레지스터 r3에 저장하도록 프로세서에게 지시하는 것이고, 두 번째 명령어는 r3와 r4의 값을 더해서 그 결과를 r5에 저장하도록 하는 것이다. 두 명령어의 실행을 파이프라인 처리하도록 하면 두 번째 명령어의 두 번째 단계에서 프로세서는 레지스터 r3와 r4 읽기를 한다. 그러나 이때 첫 번째 명령어는 r1과 r2의 값을 더하고 있는 세 번째 단계에 있으므로 아직 r3에 값이 쓰기 완료돼 있지 않은 상태가 된다. 따라서 두 번째 명령어는 레지스터 r3의 값을 읽을 수가 없고 r3의 값이 쓰기완료 될 때까지 기다려야 한다.
이와 같은 데이터 의존성에 의한 파이프라인 지연은 긴 명령어 일수록 더 많은 영향을 받게 된다. 의존성에 의한 파이프라인 지연을 처리하는 한 가지 방법은 위의 두 명령어 사이에 두 명령어와 무관한 다른 명령어를 집어 넣은 것이다(code reordering). 이렇게 함으로서 파이프라인이 의존성 때문에 지연되는 것을 방지할 수 있다. 일반적으로 code reordering은 컴파일러가 담당하게 된다.

1.4.1.2. 분기 명령어

분기 명령어는 다른 명령어의 결과를 통해 다음에 실행할 명령어가 무엇인지 결정해 프로세서에게 알려주는 역할을 한다. 파이프라인에서 아직 완료되지 않은 명령어의 결과에 따라 분기가 결정되는 경우 분기 명령어는 파이프라인 지연의 한 원인이 된다.

```
Loop:   add $r3, $r2, $r1
        sub $r6, $r5, $r4
```

beq $r3, $r6, Loop

위의 명령어는 r1과 r2를 더해 결과를 r3에 저장하고, r5에서 r4를 빼 그 결과를 r6에 저장하도록 한다. 여기서 세 번째 명령어 beq는 “같다면”을 의미하는 분기 명령어이다. 위에서는 r3와 r6가 같다면 Loop로 이름 붙여진 명령어를 실행시키고(분기 발생) 그렇지 않으면 다음에 나오는 명령어를 실행시키게(분기 없음) 되는데 r3와 r6중 어느 하나라도 레지스터에 아직 쓰기가 되지 않았다면 분기 명령어는 다음 단계 실행을 결정할 수 없게 된다.
분기 명령어에 의한 지연을 처리하는 방법으로 분기 예측(branch prediction)이 있다. 프로세서는 어떤 명령어로 분기될지 예측해 명령어를 파이프라인에 넣어 실행시키게 되는데, 만약 예측이 잘 못됐다면 레지스터로 읽어 들인 모든 것들을 삭제하고 파이프라인은 다시 시작돼야 한다. 분기가 발생하지 않는다고 예측하고 이 예측이 맞는 경우 파이프라인은 최고 속도로 진행된다.
루프 실행의 마지막에는 루프의 처음으로 돌아가는(후방 분기) 분기 명령어가 있다. 이 분기는 발생할 가능성이 높지만 전방 분기의 경우는 발생할 가능성이 그다지 높지 않다. 따라서 후방 분기에 대해서는 발생 한다고 예측(정확도 약 90%)하고 전방 분기에 대해서는 분기가 발생하지 않는다고 예측(정확도 약 50%)하는 것이 논리적이다.
프로세서가 이전 분기의 행동에 근거해 현재 또는 앞으로의 분기 명령어 행동을 예측하는 동적 예측도 있다. 동적 예측을 하는 프로세서의 경우 약 90%의 정확도를 가지고 있다.
분기 명령어에 의해 영향 받지 않는 명령어를 분기 명령 다음에 두어, 분기가 이 명령어 실행 이후에 일어 나도록 하는 지연 결정(delayed decision) 방법도 사용된다.

1.4.2. 파이프라인의 확장

좀더 빠른 프로세서를 위해 파이프라인이 확장되고 있는데 슈퍼 파이프라이닝, 슈퍼 스칼라, 동적 파이프라이닝 등이 그것이다.

1.4.2.1. Superpipelining

슈퍼파이프라이닝은 파이프라인 단계를 더욱 세분화 한 것을 말한다. 각 단계가 더 짧아 지므로 더 빠른 파이프라인 구현이 가능하다. 이상적으로는, 4단계 파이프라인은 파이프라인이 없는 경우와 비교해 4배만큼 더 빠른데, 각 단계가 완료되는 속도로 명령어가 실행되고 각 단계는 파이프라인이 없는 경우와 비교해 4분의 1만큼 시간이 걸리기 때문이다. 같은 이유로, 8단계 파이프라인은 4단계 파이프라인 보다 더 빠르다.

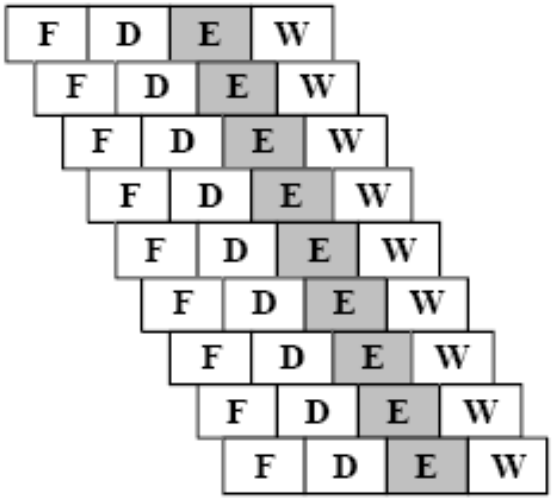


그림 1-7. 슈퍼파이프라이닝

1.4.2.2. Superscalar

슈퍼스칼라 파이프라이닝은 병렬 실행이 가능한 여러 개의 파이프라인을 가진다. 프로세서는 여러 파이프라인의 일부 혹은 모두를 이용해 여러 명령어를 동시에 실행할 수 있다. 최신의 슈퍼스칼라 시스템들은 2 ~ 6개 정도의 명령어를 모든 파이프 라인 단계에서 실행 가능하도록 시도 하고 있다.

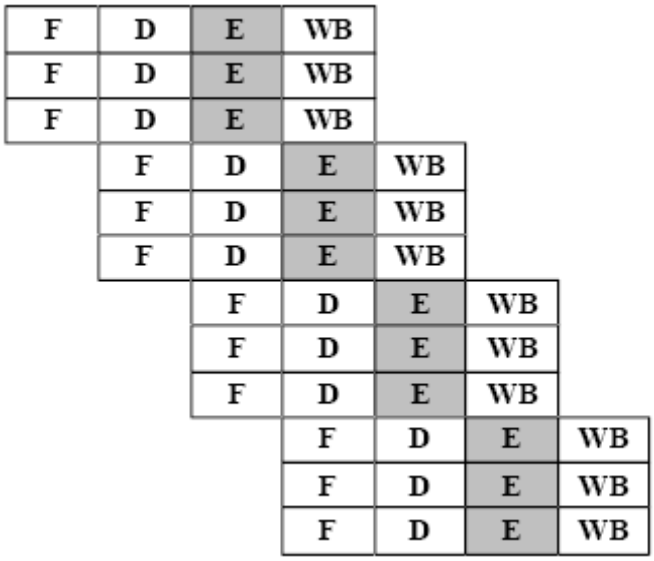


그림 1-8. 슈퍼스칼라 파이프라이닝

1.4.2.3. 동적 파이프라인

동적 파이프라인은 파이프라인 지연을 피하기 위해 하드웨어가 알아서 스케줄링 하는 것이다. 하나의 동적 파이프라인은 다음 세 개의 unit으로 나뉜다.

1. instruction fetch와 decode unit
2. 5 ~ 10개의 execute 혹은 functional units
3. 하나의 commit unit

각 execute unit에는 연산과 연산수 저장을 위한 버퍼 역할을 하는 reservation station이 있다.

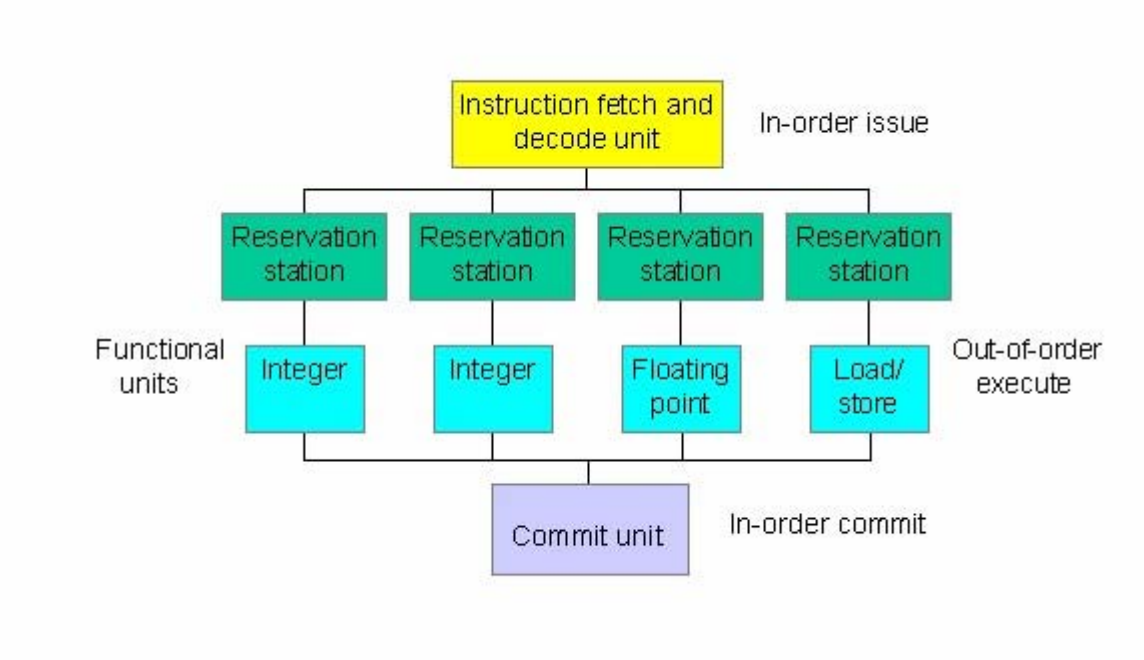


그림 1-9. 동적 파이프라인

functional unit이 무작위 실행이 가능한 반면, instruction fetch/decode unit과 commit unit은 순서대로 동작해야 한다. 명령어가 실행되고 결과가 계산될 때, commit unit은 결과를 언제 저장하는 것이 안전한지 결정하게 된다. 지연이 발생하면 프로세서는 지연이 처리될 때까지 다른 명령어가 실행되도록 스케줄 할 수 잇다.

1.4.3. 메모리 계층구조

메모리 계층구조의 목표는 가장 싼 가격에 가장 빠른 속도를 가지는 메모리 시스템을 제공하는 것이다. 이러한 목표는 데이터가 필요한 순간에 가장 빠른 레벨의 캐시에 있다면 달성 가능하다. 우리는 컴퓨터의 메모리를 메모리 계층구조로 구현함으로써 다음의 지역성의 원칙를 이용할 수 있다.

- 시간적 지역성(temporal locality): 프로그램은 가장 최근 사용한 명령어와 데이터를 사용하려는 경향이 있다. 한 내용이 참조되면 곧바로 다시 참조 되기가 쉽다.
- 공간적 지역성(spatial locality): 프로그램은 최근 접근된 item 주소의 이웃 주소에 위치한 item에 접근하려는 경향이 있다. 한 내용이 참조되면 그 근처에 있는 다른 내용들이 곧바로 참조될 가능성이 높다.

캐시메모리는 보통 두 단계 혹은 세 단계로 구성된다. 계층구조의 단계는 그 다음 더 느린 단계의

부분집합이 되어 한 단계에서 발견할 수 있는 데이터는 한 단계 더 느린 단계에도 존재한다. 모든 데이터는 가장 낮은 단계에 모두 저장돼 있다. 계층구조로 인해 지역성의 원칙을 이용할 수 있으며, 사용자는 가장 느리고 큰 메모리를 사용하지만, 가장 빠른 메모리를 접근하는 것과 같은 효과를 본다.

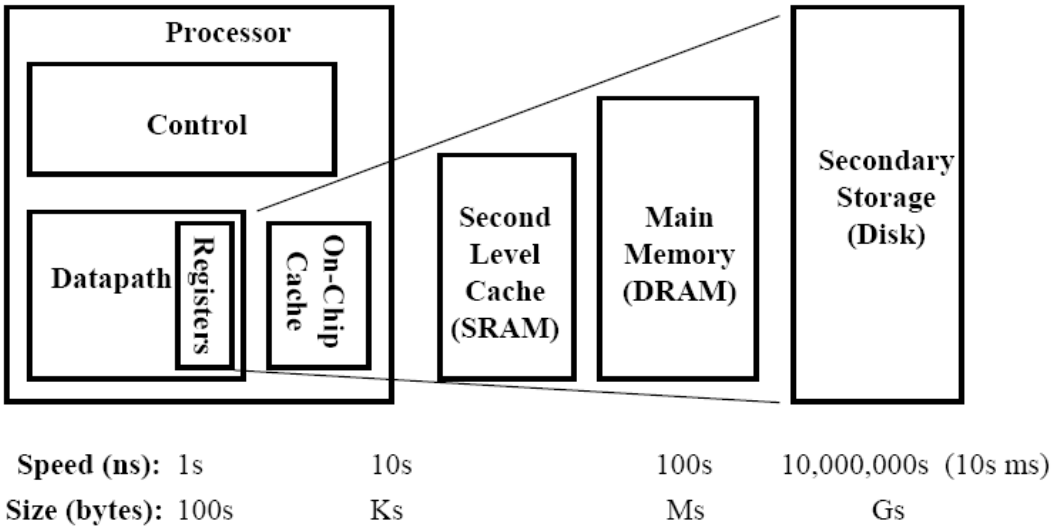


그림 1-10. 메모리 계층구조

1.4.3.1. 메모리 기술

메모리는 가격과 성능이 다르다. 거의 모든 메모리는 반도체로 만들어지며, 메인 메모리는 DRAM(Dynamic Random Access Memory) chip으로 만들어진다. 캐시는 더 빠르고 비싼 SRAM(Static Random Access Memory)으로 만들어 진다. DRAM은 각 bit를 매우 작은 축전기에 저장되는 전하로 표현하는 charge 기반 장치인데, 전하는 누전 되므로 시스템은 지속적(주기적)으로 refreshed 되어야 한다. DRAM에서 bit를 읽으면 그 bit는 방전되므로 다시 refresh 되어야 한다. 반면 SRAM 메모리는 전원이 끊어지지 않는 한 데이터를 그대로 유지하여 데이터 refresh가 불필요하다.

메모리 cycle 시간은 메모리에 대한 반복 요청 사이의 최소 시간이 된다. SRAM의 cycle 시간은 DRAM 보다 8배에서 16배 정도 더 빠르다. 상식적인 수준의 가격을 가지는 시스템이라면 SRAM만 가지고 메모리 전체를 구성할 수 없다. Cost-effective solution은 메모리 계층구조를 1~3단계의 SRAM 캐시, DRAM 메인 메모리와 디스크를 사용하는 가상 메모리로 구성하는 것이다.

메모리 성능은 접근 시간과 사이클 시간 두 가지에 의해 결정될 수 있다.

- 접근 시간(access time): 메모리 참조에 대한 요청 이후 응답까지 걸리는 시간
- 사이클 시간(Cycle time): 연속적 메모리 요청 사이의 최소 시간. 메모리가 작업완료와 동시에 Ready 신호를 내놓고 다음 신호를 받을 준비가 된 상태까지 걸리는 시간을 의미한다. 실제 작업에 소요되는 시간이라고 볼 수 있는데 보통 접근 시간의 1.5~2배의 시간이

다. SRAM은 refresh가 불필요하므로 접근 시간과 사이클 시간이 동일하다.
간단한 DRAM에서 메모리 트랜잭션에는 접근 시간 더하기 사이클 시간이 필요하다.

메모리 기술	일반적인 접근 시간(ns)	가격($)[1]
SRAM	5 ~ 25	100 ~ 250
DRAM	60 ~ 120	5 ~ 10
Magnetic Disk	10 ~ 20 Million	0.10 ~ 0.20

표 1-3. 메모리 계층별 접근시간과 가격

1.4.3.2. 메모리 인터리빙

메모리 대역폭은 인터리빙에 의해 증가될 수 있다. 메모리를 여러 개의 뱅크로 구성하여 연속적인 워드를 다른 뱅크에 위치하도록 한다. 여러 개의 뱅크는 동시에 접근 할 수 있고 따라서 유효 사이클 시간 (effective cycle time)을 줄일 수 있다.

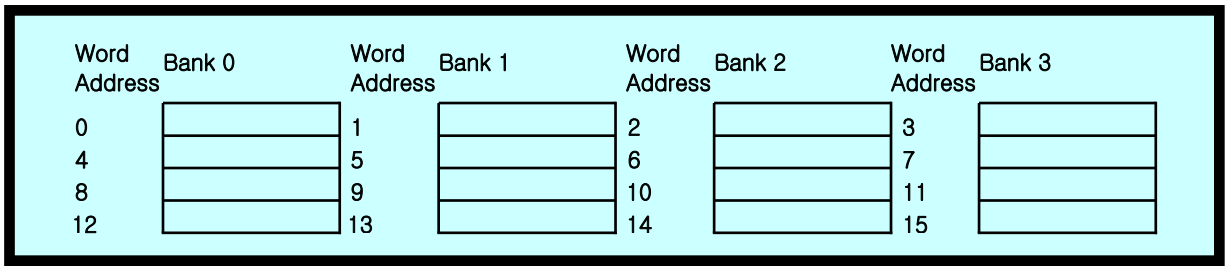


그림 1-11. 워드 어드레싱 4-way interleaved 메모리 다이어그램

위와 같은 인터리빙 방식은 연속적인 메모리 접근을 최적화 한다. 메모리 접근 패턴이 같은 뱅크에 반복적으로 접근하는 형태라면, 뱅크 지연(stall or contention)이 발생할 수 있다.

1.4.4. 캐시 메모리

캐시 메모리 시스템은 아래와 같은 특성들에 의해 결정지어진다.

- 캐시의 수
- 캐시 크기
- 캐시라인 크기
- 캐시 연관 정도(associativity)
- 캐시라인 교체 정책(replacement policy)
- 쓰기 규칙(Write strategy)

어느 메모리 위치가 캐시에 있는지 추적하기 위해 사용되는 공간을 줄이기 위해 캐시메모리는 같은 크기의 슬롯(캐시라인)으로 나뉘어져 있다. 캐시라인은 메인 메모리로 또는 메인 메모리로부터

[1] 1997년 기준

전송 가능한 메모리 최소 단위로 보통 32에서 128 바이트 사이의 크기를 가진다.

1.4.4.1. 캐시 연관 정도

캐시는 메인 메모리보다 크기가 작아서 일부 메모리 위치가 같은 캐시 위치를 공유해야 한다. 메모리 위치가 캐시 위치에 할당되는 방식을 사상(Mapping) 이라고 한다.

N-way 연관 캐시에서 메모리 위치는 n개 엔트리로 구성되는 캐시 집합으로 사상된다. 1-way 연관 캐시를 직접 사상 캐시라고 하며 메모리 위치는 캐시내의 한곳에만 위치할 수 있다. 완전 연관 캐시는 캐시라인 하나가 캐시내의 어느 곳에도 위치할 수 있는 것이다. 직접 사상 캐시는 구현이 가장 쉽고 빠르지만 실패율이 높고 캐시 thrashing 발생 가능성이 높다. 완전 연관 캐시는 같은 크기의 직접 사상 캐시보다 캐시 thrashing 발생 가능성이 작으며, 실패율은 낮지만 적중시간이 느리다는 단점이 있다.

캐시 thrashing: 코드 일부가 같은 캐시라인에 사상되는 다른 메모리 위치 사이에서 반복적으로 접근되는 경우 발생하는 문제로 모든 메모리 접근이 캐시 실패를 야기시켜 마치 캐시가 없는 것 같은 결과를 보여준다. 캐시 thrashing은 직접 사상 캐시에서 발생 가능성이 높지만, 같은 캐시라인 집합에 사상되는 코드 일부가 반복적으로 일정 간격 떨어진 위치에 접근되는 경우 n-way 연관 사상에서도 발생 가능하다.

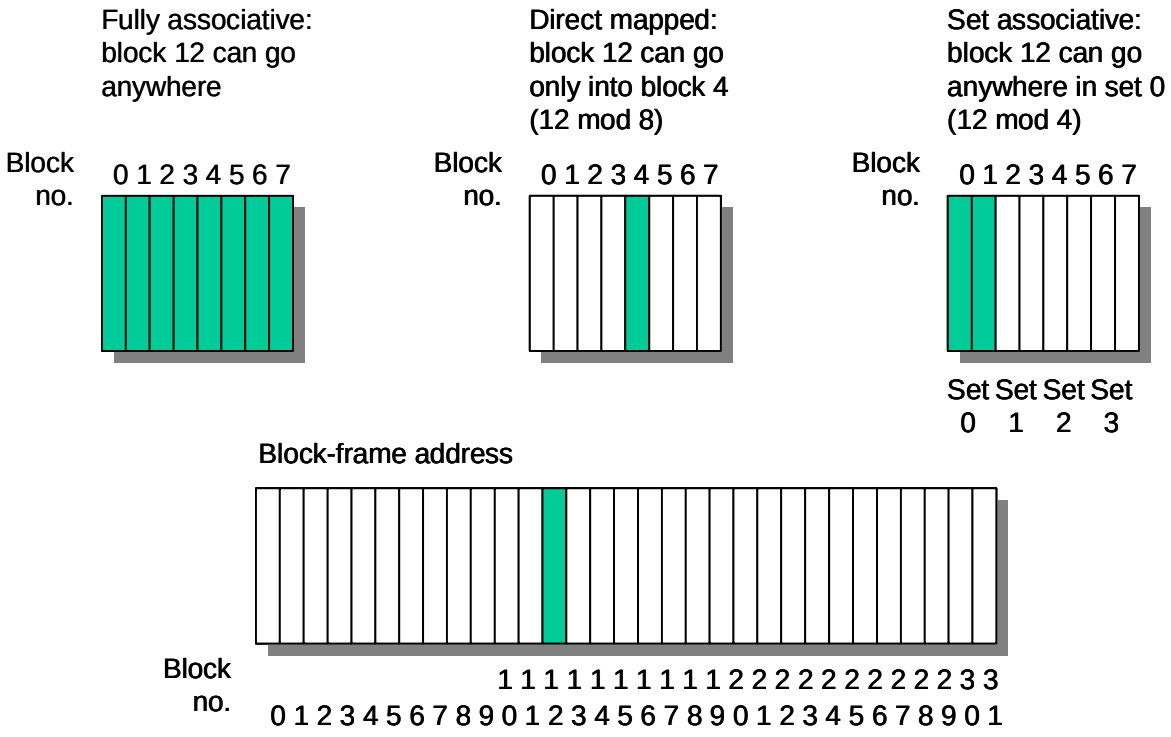


그림 1-12. 캐시 연관성

1.4.4.2. 캐시 적중과 실패

캐시 적중(cache hit)은 CPU가 필요한 데이터를 캐시에서 찾았을 때 발생하고 캐시 실패(cache miss)는 CPU가 필요한 데이터를 캐시에서 찾지 못했을 때 발생한다. 캐시실패가 발생했을 때 하나의 캐시라인이 메인 메모리(또는 아래 단계의 캐시)에서 검색돼 캐시에 올려진다.
캐시 실패율은 캐시 접근에 대한 캐시실패 발생 율이며, 대부분의 프로세서는 캐시실패 회수와 메모리 참조를 헤아리기 위해 하드웨어 카운터를 제공하고 있는데 이를 통해 캐시실패율을 계산할 수 있다. 한번의 캐시실패로 인해 소비되는 시간을 캐시실패 손실(penalty)이라 하고 하위 계층에서 대응되는 블록을 상위 계층으로 복사해 오는 시간에 그 블록을 프로세서에 보내는데 걸리는 시간을 더한 값이 된다. 이는 메모리 대역폭과 지연시간에 의존적이다. 캐시실패는 하드웨어에 의해 처리되고 필요한 데이터가 가용하게 될 때까지 프로세서의 순차실행을 지연시키는 원인이 된다. CPU가 지연되어 메모리 접근을 기다리는 동안의 사이클을 메모리 지연 사이클이라고 한다. 프로그램 실행 시간 측정의 척도가 되는 CPU 시간을 결정하는데 메모리 지연 시간은 중요한 역할을 한다.

CPU 시간 = (CPU 클럭 사이클 + 메모리 지연 클럭 사이클) x 클럭 사이클 시간

캐시실패의 형태는 다음과 같이 세 가지로 분류될 수 가능하다.

1. compulsory: 캐시라인에 대한 첫 번째 접근에서 발생
2. capacity: 캐시가 프로그램 실행에 필요한 모든 캐시라인을 포함할 수 없을 때 발생하며 캐시 크기를 증가시켜 캐시실패를 줄일 수 있다.
3. conflict: 집합 연관 또는 직접 사상 캐시에서 같은 집합(set)에 너무 많은 캐시라인이 사상돼 한 블록이 버려지거나 나중에 검색되어야 할 때 발생하며 캐시 크기를 증가시키고 연관 정도를 높여서 캐시실패를 줄일 수 있다.

1.4.4.3. 다단계 캐시

빠른 프로세서와 상대적으로 긴 접근시간이 필요한 주기억장치와의 성능 차이를 더욱 줄이기 위해 여러 계층의 다중 레벨 캐시를 사용한다. 2차 캐시는 1차 캐시(primary cache)에서 실패가 발생될 때 접근된다. 2차 캐시가 원하는 데이터를 갖고 있으면 실패손실은 2차 캐시의 접근시간이 되며 이 값은 메인 메모리 접근시간보다 훨씬 작다. 시스템에 따라서는 3차 캐시를 둘 수도 있으며, 느리고 큰 캐시를 사용함으로써 메인 메모리로 넘어갈 많은 접근을 사전에 방지할 수 있으므로 실패 손실을 감소 시킨다.
2중 레벨 캐시의 평균 메모리 접근 시간은 다음과 같다.

Average memory access time = Hit time(L1) + Miss rate(L1) x Miss Penalty(L1)
Miss penalty(L1) = Hit time(L2) + Miss rate(L2) x Miss penalty(L2)

그래서

Average memory access time = Hit time(L1) + Miss rate(L1) x (Hit time(L2) + Miss rate(L2) x Miss penalty(L2))

데이터 지역성(locality)에 대한 대부분의 성능이득을 1차 캐시에서 볼 수 있으므로 지역 실패율(local miss rate)은 2차 또는 3차 캐시에서 크게 나타난다. 전체 실패율은 메모리 참조에 대한 모든 메모리 계층에서 실패가 발생된 참조 비율이며 이것이 얼마나 자주 메모리에 접근해야 되는가를 알려 주는 보다 좋은 척도가 된다.

- local miss rate: n차 캐시에서 발생한 실패의 수를 n차 캐시 메모리에 대한 접근 수로 나눈 것
- global miss rate: 캐시에서 발생한 실패의 수를 CPU에서 발생된 전체 메모리 접근 수로 나눈 것(1단계 캐시는 global miss rate과 local miss rate이 동일: 전체 메모리 접근은 1차 캐시를 거친다.)

1.4.4.4. 캐시 교체 정책

새로운 캐시라인이 읽어지면 다른 하나는 버려져야 한다. 직접 사상 방식에서는 캐시 실패가 발생하면 요구된 블록은 정확히 한 곳으로만 갈 수 있고 그 위치에 이미 있는 블록은 교체 되어야 한다. 집합연관 방식의 캐시에서는 요구된 블록을 어디에 위치시킬지 선택해야 하며, 이는 바로 교체시켜야 할 블록을 선택하는 것이다. 완전연관 캐시는 모든 블록이 교체의 후보가 되며, 집합연관 캐시의 경우는 선정된 집합내의 블록 중에서 선택해야 한다. 삭제되어야 할 캐시라인을 선택하는 방법을 캐시 교체 정책(replacement policy)이라고 하며, 대표적인 3가지는 다음과 같다.

1. Least Recently Used(LRU): 가장 오랜 시간 쓰이지 않은 라인을 교체
2. Random(or pseudorandom)
3. Round Robin, or First-in First-out(FIFO): 가장 오래된 block 교체

가장 많이 사용되는 LRU 정책은 캐시라인에 대하여 접근을 항상 추적해야 하므로 구현하기 복잡하고 종종 근사될 뿐이다. 그러나 LRU는 작은 캐시에서는 random이나 round robin보다 더 나은 성능을 보여준다.

1.4.4.5. 캐시 쓰기

수정된 데이터로 캐시에 새로운 값을 저장할 때 대응 되는 메인 메모리 내의 데이터를 갱신하기 위한 방법을 캐시 쓰기 규칙(write strategy)라고 하며 두 가지 방법이 있다.

1. 나중 쓰기(Write-back): 캐시에만 수정된 데이터를 저장하고, 수정된 캐시라인이 교체될 때 하위 수준(낮은 계층 메모리)으로 데이터 저장
2. 즉시 쓰기(Write-through): 데이터가 캐시와 하위 계층 메모리 모두에 즉시 쓰여 지는 것

나중 쓰기의 경우 다수의 쓰기가 행해진 블록을 하위 계층에 한 번만 쓰면 되고 하위 수준에 쓰

여질 때 전체 블록을 한꺼번에 쓰기 때문에 높은 대역폭을 효과적으로 사용할 수 있어, 메모리 traffic이 현저히 감소한다. 가상 메모리 시스템에서 주로 사용한다.
즉시 쓰기의 경우 실패는 블록이 나중에 하위 수준에 쓰여지는 것을 요구하지 않으므로 간단하고 비용이 적다. 나중 쓰기보다 구현이 쉽지만, 고성능 시스템의 경우 쓰기 버퍼를 사용한다.

1.4.4.6. SMP 시스템의 캐시 contention

SMP 시스템의 경우 프로세서들은 메인 메모리를 공유하지만 개별적으로 캐시를 가진다. 이로 인해 여러 캐시가 공유한 데이터의 복사본을 각각 가지게 되므로 데이터의 일관성을 유지하는 것이 중요하다. 공유 데이터를 캐시에 기억시키지 않고 주기억장치에서 바로 읽도록 할 수도 있지만 속도가 너무 떨어진다.

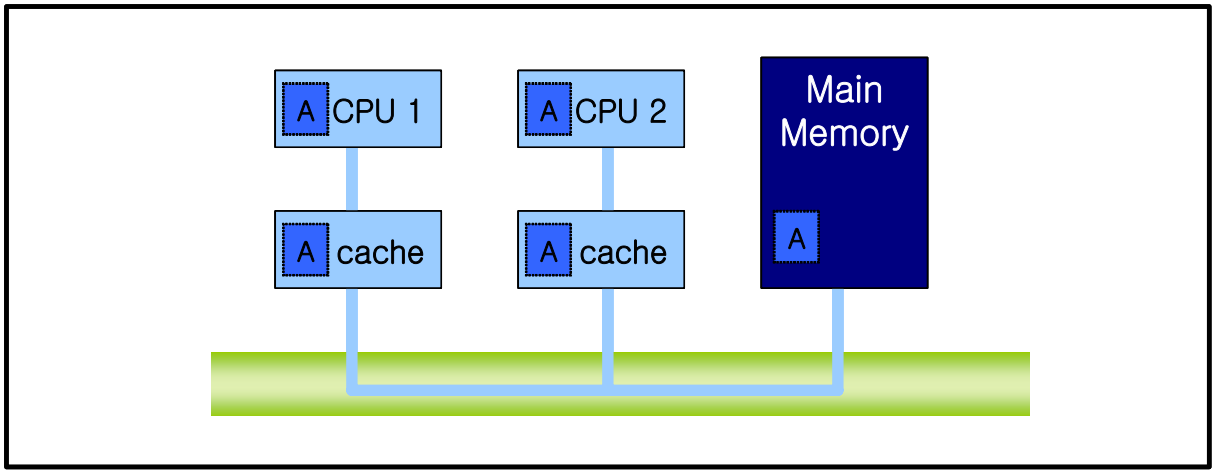


그림 1-13. SMP 시스템의 캐시 일관성 유지

여러 프로세서의 캐시가 갖고 있는 데이터간의 일관성을 유지하는 방법을 캐시 일관성 유지 프로토콜(cache coherency protocol)이라고 한다. 캐시 일관성 유지 프로토콜은 모든 프로세서들이 주기억장치의 값을 일관성 있게 보도록 보장한다. 한 CPU가 캐시라인에 쓰기를 하면, 복사본을 가지고 있는 다른 CPU는 그 복사본을 폐기하거나 새로운 값을 받아 들인다. 일관성을 유지 하기 위해 쓰기를 할 때 어떻게 하느냐에 따라 캐시 일관성 유지 프로토콜은 두 가지로 나눌 수 있다.

1. 쓰기 무효화(write invalidate): 값을 바꿀 프로세서는 먼저 다른 모든 캐시에 있는 복사본을 무효화 한 후 자기 캐시에 쓴다. 여러 프로세서가 동시에 읽는 것은 허용하나 쓰기는 한 프로세서 에게만 허용한다.
2. 쓰기 갱신(write update): 공유된 블록을 모두 무효화하는 대신 버스를 통해 새로운 값을 모두에게 보낸다. 각 캐시는 이 값을 받아 복사본의 값을 갱신한다.

쓰기 무효화는 필요한 버스 대역폭을 작게 한다는 측면에서 나중 쓰기와 같은 장점이 있고 쓰기 갱신은 새로운 값이 다른 캐시에 즉시 반영되므로 지연시간을 줄일 수 있다는 장점이 있다. 상용의 다중 프로세서 시스템에서는 버스 통신량을 줄여 한 버스에 더 많은 프로세서를 연결하기 위

해 나중 쓰기 캐시가 주로 사용되고 마찬가지로 쓰기 무효화 프로토콜을 표준으로 사용한다.
캐시 contention은 두 개 이상의 CPU가 교대로 반복해서 동일한 캐시라인을 갱신할 때 발생하며 코드 성능을 악화시킨다. 이러한 현상은 두 개 이상의 프로세서가 같은 값을 갱신 하는 실제 메모리 contention으로 인해 발생되거나, 서로 다른 프로세서가 같은 캐시라인에 있게 된 다른 값들을 갱신하고자 하는 허위 공유(false sharing)로 인해 발생한다. 메모리 contention을 수정하기 위해서는 알고리즘을 바꿔야 하지만 허위 공유는 데이터 레이아웃을 바꿔서 수정할 수 있다.
허위 공유(false sharing): 서로 다른 공유 변수가 같은 캐시라인에 속할 때 각 스레드가 다른 변수에 접근하더라도 전체 캐시라인이 이동하는 상황이다. 한 스레드에 의한 할당이 이전 소유자의 캐시라인을 무효화 한다.

```
REAL A(M,N), S(M)
!$OMP PARALLEL DO
DO I = 1,M
  S(I) = 0.0
  DO J = 1, N
    S(I) = S(I) +  A(I,J)
  ENDDO
ENDDO
```

위 코드에서 M=4, 4개의 스레드를 가정하면 각 스레드가 같은 캐시라인에 속한 서로 다른 변수 S(1), S(2), S(3), S(4)에 접근하게 되고, 한 번에 하나의 스레드만이 그 캐시라인을 소유할 수 있다. 한 스레드에 의한 모든 할당은 이전 소유자의 캐시라인을 무효화 하므로 사실상 순차실행이 되어 병렬화의 효과를 볼 수 없다.
위의 코드에서 허위 공유가 발생하지 않도록 데이터 레이아웃을 변경하면 다음과 같다.

```
REAL A(M,N), S(32,M)
!$OMP PARALLEL DO
DO I = 1,M
  S(1,I) = 0.0
  DO J = 1, N
    S(1,I) = S(1,I) +  A(I,J)
  ENDDO
ENDDO
```

1.4.5. 가상 메모리

주기억장치가 보조 기억장치(디스크)를 위한 캐시로 이용되는 것을 가상 메모리 기술이라고 한다.

가상 메모리를 사용하는 이유는 다수의 프로그램을 동시에 수행할 때 메모리를 효과적으로 공유할 수 있게 하고, 작고 제한된 크기의 주기억장치에서 프로그래밍해야 하는 제약을 제거하기 위해서 이다. 가상 메모리 시스템을 이해하기 위해 중요한 몇 가지 용어 및 개념을 소개한다.

- 가상 주소(virtual address): 프로그램 또는 프로세스에 주어진 주소로 프로세스가 참조하는 가상 주소 범위를 주소공간(address space)이라고 한다.
- 실제 주소(physical address): 메인 메모리의 주소로 메모리 공간(memory space)이라고도 한다.
- 메모리 사상: 가상 주소와 실제 주소 사이의 사상으로 주소 변환(address translation)이라고도 한다.

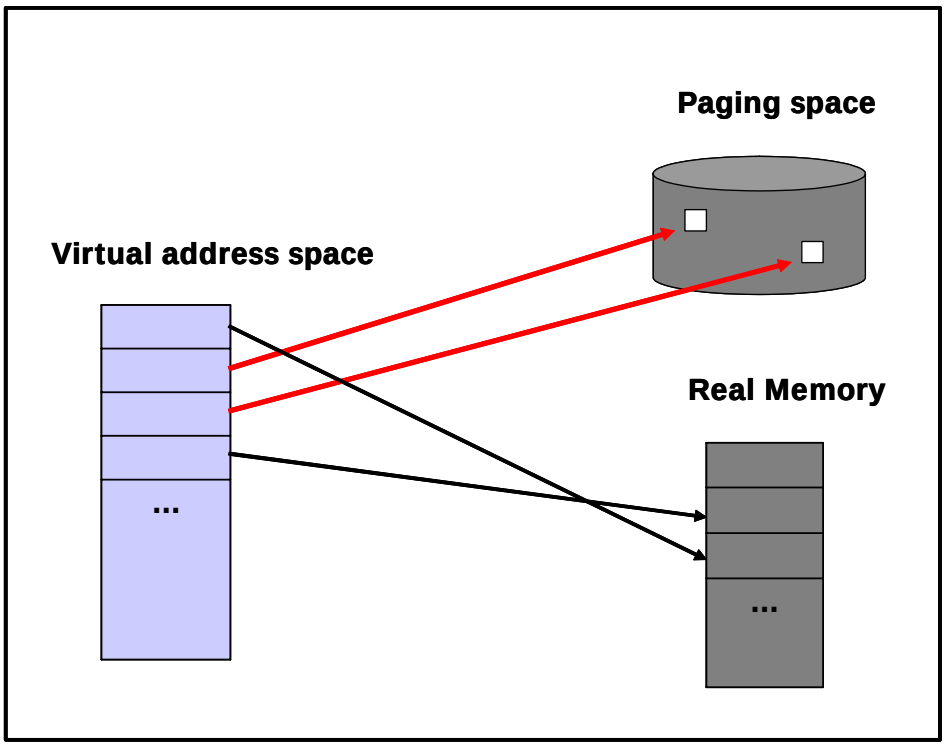


그림 1-14. 가상 메모리

주기억장치는 많은 프로그램의 활성화된 부분만을 담고 있다. 이를 통해 주기억장치와 프로세서까지도 효율적인 공유가 가능해 진다. 물론 많은 프로그램이 같은 메모리를 사용할 경우 서로 방해 받지 않아야 하며 하나의 프로그램은 자기에게 할당된 주기억장치의 부분만을 읽고 쓸 수 있어야 한다. 가상 메모리를 통해 프로그램의 주소 공간을 실제 주소로 변환해 주는 번역 과정이 한 프로그램의 주소 공간을 다른 프로그램으로부터 보호해 준다.

가상 메모리 기술을 통해 주기억장치보다 큰 메모리를 사용하는 프로그램 작성이 가능하다. 가상 메모리는 주기억장치(실제 메모리: physical memory)와 디스크로 구성되는 2단계의 메모리 계층을 관리하는데, CPU는 가상 주소를 만들어 내고 가상 주소는 주기억장치를 접근하는데 사용하는

실제 주소로 번역된다.

페이지는 OS가 프로그램에게 할당하는 (연속)메모리 블록의 최소단위이다. 운영체제는 각 프로세스를 위해 가상 주소 공간을 만들어 불필요한 페이지를 디스크에 보관해 두고, 필요할 때 메인 메모리에 올린다. 모든 메모리 접근에서 운영체제는 가상 주소를 실제 주소로 전환하는데, 이때 페이지 테이블을 이용한다. 각 프로세스는 자신의 페이지 테이블을 가지며 페이지 테이블은 그 프로세스의 가상 주소 공간을 주기억장치로 사상한다. 만약 찾고자 하는 가상 주소가 주기억장치에 없다면 이 페이지는 디스크에 존재하게 된다. 실제 페이지는 두 개의 가상 주소가 같은 실제 주소를 가리키도록 해서 공유 가능하다. 이렇게 해서 두 개의 다른 프로그램이 데이터와 코드를 공유할 수 있다.

페이지 테이블은 메모리를 색인하는 테이블로서 주기억장치 내부에 존재하며 가상 주소의 페이지 번호로 색인돼 있고 대응되는 실제 페이지 번호를 갖고 있다. 페이지 테이블이 주기억장치 내에 있어서 프로세스에 의한 모든 메모리 접근은 최소한 두 번 필요한데 실제 주소를 얻기 위한 메모리 접근과 데이터를 얻기 위한 접근이 그것이다.

페이지 테이블에 대한 참조의 지역성을 이용해 접근 성능을 증가시키기 위해 TLB(Translation Look-aside Buffer)를 이용한다. 가장 최근에 사용된 주소 전환을 TLB에 저장하고 메모리 접근이 있을 경우 가장 먼저 TLB에서 실제 주소에 대한 사상을 찾게 된다. 프로세스가 TLB에 있지 않은 가상 주소를 사용하려 할 때 TLB 실패가 발생하고, 페이지 테이블을 통해 실제 주소를 찾게 된다. 만약 참조된 페이지가 실제 메모리에 없다면 페이지 부재(page fault)가 발생하고 디스크에 저장된 페이지를 찾게 되는 가장 나쁜 경우가 발생하게 된다.

2. 성능 측정

최적화를 시도하기 전에 먼저 코드의 성능에 대해 알고 있어야 할 것이다. 코드의 성능을 결정하는 방법도 여러 가지가 있다. 성능 결정의 기준은 코드의 실행시간이 될 수도 있고 단위시간당 부동소수 연산 회수인 Flops가 될 수도 있다. 코드 실행시간도 CPU time과 Wall Clock time이 다르다. 여기에서는 성능 측정의 기준이 되는 몇 가지 대표적인 척도를 소개하고 그 측정 방법에 대해 알아본다.

2.1. CPU 성능 결정

컴퓨터의 성능을 결정짓는 핵심요소는 역시 CPU이다. CPU에 의해서만 컴퓨터의 성능이 결정되는 것은 아니지만 다른 무엇보다 CPU는 컴퓨터의 성능을 결정하는데 중요한 역할을 한다.
CPU의 최대 성능을 알면 실행 프로그램이 실제 CPU 성능의 몇 퍼센트 정도로 실행되고 있는지 판단할 수 있을 것이다. CPU의 이론 최대 성능은 Flops 단위로 다음과 같이 결정된다.

CPU 이론 최대 성능 = [Machine Cycle] x [1 or 2 (FMA)] x [Pipelining Functional Unit의 개수] Flops

IBM의 POWER4 프로세서는 2개의 pipelining functional unit을 가지는 슈퍼스칼라 프로세서로 부동소수 곱셈과 덧셈 연산을 한꺼번에 처리하는 FMA 기능을 가지고 있어 1.7 GHz CPU의 이론 최대 성능은 1.7 x 2 x 2 = 6.8 GFlops가 된다. 인텔의 펜티엄4(3.0 GHz) 프로세서는 한꺼번에 두 개의 배정도 부동소수 연산을 한꺼번에 처리하는 SIMD 명령어를 가지고 있어서 이론 최대 성능이 3.0 x 1(FMA 기능 없음) x 2 = 6.0 GHz가 된다.
코드의 실행시간 측정에 의해 결정되지 않는 성능 문제들은 대부분 코드가 실행되는 시스템 아키텍처에 의존적이다. 메모리 접근 시간이 하나의 예가 될 수 있는데, 많은 코드에서 메모리 접근 시간은 clock 속도보다 시스템 성능에 더 큰 영향을 주게 되며 clock 속도가 높을수록 더더욱 그렇다. 대부분의 프로세서는 하드웨어와 관련돼 작동되는 것들의 진동수를 알 수 있도록 하드웨어 카운터를 가지고 있다. 하드웨어 카운터는 clock cycle, 메모리 load와 store, 캐시 실패, TLB 실패 등과 같은 성능관련 이벤트를 카운트 한다.
하드웨어 카운터는 매우 유용한 정보를 제공해 주지만 표준화돼 있지는 않다. 하드웨어 카운터에 의해 수집된 정보는 HPM Toolkit과 같은 프로그램에 의해 처리돼 사용자에게 제공된다.

2.2. Time

프로그램을 개발하고 사용하는데 있어 중요한 판단기준 중의 한 가지는 시간이며, 여기에는 실행시간뿐 아니라 개발시간도 포함된다. 성능 최적화는 실행시간에 그 초점을 맞춘다.

- user time: 프로그램 작업을 수행하는데 걸린 시간
- system time: O/S가 프로그램을 지원하는데 걸린 시간(시스템 호출을 통해 운영체제가 지원하는 기능을 사용하는데 걸린 시간)
- CPU time: user time + system time
- Wall clock time: 프로그램 실행 시작부터 완료까지 걸린 시간

프로그램의 CPU 시간은 다음과 같이 결정할 수 있다.

CPU Time = [CPU cycles for the program]/[Clock rate]
= [Instruction count] x [CPI] / [Clock rate]

여기서 명령어 개수는 하드웨어 카운터 등을 이용하여 측정할 수 있으나 명령어당 cycle 수를 나타내는 CPI는 명령어 배합이나 그 구현 방식 등에 따라 달라질 수 있으므로 측정이 쉽지 않다. 어떤 프로그램 전체에 대한 CPI는 각 명령어 실행에 필요한 클럭 수와 각 명령어의 실행 빈도에 따라 달라진다.

순차실행

하나의 작업이 프로세서를 독점적으로 사용하는 시스템이 있는 반면 어떤 시스템들은 여러 작업들이 서로 교체되면서 프로세서를 공유해 사용한다. 작업이 교체돼서 나가면 그 작업에 대한 wall clock 시간은 계속해서 누적되지만, CPU 시간은 누적되지 않는다. 이 경우 wall clock 시간은 시스템 부하의 함수가 되어 최적화를 위한 척도가 되기에 부적절하다. 프로세서를 독점적으로 사용하는 경우라도 sleep 함수를 호출하는 경우와 같이 코드 일부가 더 이상의 진행을 못하고 어떤 사건의 발생을 기다리는 경우가 있는데, 이런 경우 sleep 시간 동안 wall clock 시간은 누적되지만 CPU 시간은 계산되지 않는다. 이런 관점에서 순차코드 최적화에서 중요한 것은 CPU 시간을 감소시키는 것이다.

병렬실행

병렬 프로그램 최적화에서도 순차코드에서와 같이 CPU 시간 감소는 여전히 중요하다. 그러나, 병렬 프로그램에서는 모든 프로세스가 동기화 되어야 하는 경우처럼, 일부 병렬 프로세스가 더 느린 프로세스의 작업이 완료되기를 기다려야 하는 경우가 있다. 이때 해당 병렬 프로그램의 wall clock 시간은 누적되지만 CPU 시간은 누적되지 않는다. 병렬 코드 최적화의 관점에서 대기 시간을 최소화 하는 것은 중요한 작업이 된다. 따라서 병렬코드 최적화 과정에서는 wall clock 시간을 감소시키는 것이 중요하다.

2.2.1. How to Time

프로그램 또는 프로그램 일부분의 실행시간 측정은 해당 시스템에서 사용할 수 있는 타이머 중

가장 정확하고 타이머 자체의 부하가 가장 작은 것을 사용하는 것이 좋다. 또한, 프로그램이 실행되는 시스템은 전용(dedicated) 실행이 보장되는 시스템이거나 가급적 부하가 작은 시스템, 또는 전용 자원을 확보해 주는 큐잉(queuing) 시스템을 이용하는 것이 좋다.
전용 실행이 보장되지 않는 시스템에서는 wall clock time보다 CPU time이 더 정확한 척도 일 수 있다.

Timing Methods

- UNIX Time
- Source Timer Routines
- Profiling

Timer	Usage	Wallclock / CPU Time	Resolution	Languages
time	shell / script	both	1/100th second	any
timex	shell / script	both	1/100th second	any
gettimeofday	subroutine	Wall-clock	microsecond	C/C++
read_real_time	subroutine	Wall-clock	nanosecond	C/C++
rtc	subroutine	Wall-clock	microsecond	Fortran
irtc	subroutine	Wall-clock	nanosecond	Fortran
dtime_	subroutine	CPU	1/100th second	Fortran
etime_	subroutine	CPU	1/100th second	Fortran
mclock	subroutine	CPU	1/100th second	Fortran
timef	subroutine	Wall-clock	millisecond	Fortran
MPI_Wtime	subroutine	Wall-clock	microsecond	C/C++, Fortran

표 2-1. 여러 가지 timer 함수

2.2.2. UNIX Time

2.2.2.1. time

time 명령어는 유닉스 shell 명령어로 프로그램의 총 실행시간을 반환한다. 출력 형식은 korn shell과 C shell에서 조금씩 다르며 기본적으로 real time, user time, system time의 세가지 정보를 출력한다.

- real time: 프로그램이 시작되어 마칠 때까지 걸리는 실제 총 실행시간(wall-clock time)
- user time: 프로그램 실행에 사용된 총 CPU 시간
- system time: 프로그램 실행에서 운영체제 호출에 사용된 총 CPU 시간

Korn shell에서 time 명령어의 실행 결과

```
$ time mpitest -procs 4
real    0m2.87s
```

```
user    0m3.29s
sys     0m1.57s
```

C shell에서 time command의 실행 결과

```
$ time mpitest -procs 4
1.150u 0.020s 0:01.76 66.4% 15+3981k 24+10io 0pf+0w
  ①    ②      ③    ④      ⑤       ⑥    ⑦  ⑧
```

C shell에서의 실행결과는 각각 다음의 의미를 가진다.

- user CPU time : 1.15초
- system CPU time : 0.02초
- real time(wall-clock time) : 0분 1.76초
- real time에서 CPU time(user+system)이 차지하는 정도 : 66.4%
- 메모리 사용량 : 공유(15Kbytes) + 공유되지 않음(3981Kbytes)
- 입력(24블록) + 출력(10블록)
- 페이지 폴트 없음
- 스왑 없음

2.2.2.2. timex

timex 명령어는 IBM 유닉스인 AIX에서 사용 가능한 명령어로 별도의 옵션 없이 사용하면 어떤 shell 환경에서든지 time 명령어를 Korn shell에서 사용한 것과 똑 같은 형식의 결과를 출력한다. timex 명령어는 여러 가지 옵션과 같이 사용하여 프로세스 통계 등과 같이 다양한 결과를 출력할 수 있다.

-o : 읽혀지거나 기록된 전체 블록 수와 실행 프로그램과 그 하위 프로세스가 전송한 전체 문자 수를 출력한다.
-p : 실행 프로그램과 그 하위 프로세스에 대한 프로세스 사용 통계 레코드를 출력한다.
-s : 프로그램 실행 중 전체 시스템 활동(sar 명령에 나열된 모든 데이타 항목)을 출력한다.

timex 명령어의 실행결과(-p 옵션)

```
$ timex -p naive
real 64.66
user 64.63
sys   0.02

START AFT: Wed Jan 29 16:49:49 KORST 2003
END BEFOR: Wed Jan 29 16:50:54 KORST 2003
COMMAND                    START   END      REAL     CPU      CHARS     BLOCKS
NAME       USER   TTYNAME  TIME    TIME     (SECS)   (SECS)   TRNSFD    READ
naive      my_id   pts/52  16:49:49 16:50:53  64.64    64.64    4000256      0
```

2.2.3. Source Timer Routines

코드 전체가 아닌 일부분의 실행시간을 알고자 할 때 타이밍 함수(또는 서브루틴)를 소스코드에 넣어 사용한다.

2.2.3.1. gettimeofday

gettimeofday() 루틴은 대부분의 Unix 시스템 표준 C 라이브러리에 포함되어 있으며, 1970년 1월 1일 0시부터의 시간을 초와 백만 분의 일초 단위로 출력한다. C 프로그램의 어디에서든 호출하여 코드 부분의 시작시간과 종료시간을 알 수 있으며 종료시간과 시작시간의 차이를 구해 코드 부분의 실행시간을 알아볼 수 있다. 측정 시간의 정밀도는 시스템마다 다를 수 있다.

gettimeofday() 루틴의 사용 예제(C)와 실행결과

```
/* gettime.c */
#include <stdio.h>
#include <sys/time.h>
#define ARRAY_SIZE          1000

float a[ARRAY_SIZE][ARRAY_SIZE];
float b[ARRAY_SIZE][ARRAY_SIZE];
float c[ARRAY_SIZE][ARRAY_SIZE];
struct timeval start_time, end_time;

main()
{
    int i, j;
    int total_usecs;

    /* First, call gettimeofday() to get start time */
    gettimeofday(&start_time, (struct timeval*)0);

     for(i=0;i<1000;i++)
         for(j=0;j<1000;j++)
             c[i][j] = c[i][j] + a[i][j] * b[i][j];

    /* Now call gettimeofday() to get end time */
    gettimeofday(&end_time, (struct timeval*)0);    /* after time */

    /* Print the execution time */
    total_usecs = (end_time.tv_sec-start_time.tv_sec) * 1000000 +
                        (end_time.tv_usec-start_time.tv_usec);

    printf("Total time was %d uSec.₩n", total_usecs);
}
```

```
$ gettime
 Total time was 48616 uSec.
```

2.2.3.2. rtc()

함수 rtc()는 사용이 간편하고 백만분이 일초 단위의 정밀도를 가지는 Real(8) 값을 출력한다.

rtc()함수의 사용 예제(Fortran)와 실행결과

```
PROGRAM RTC_TIME
INTEGER SIZE
PARAMETER(SIZE=5000)
REAL(8) N(SIZE,SIZE), M(SIZE,SIZE), L(SIZE,SIZE)
REAL(8) A, B, rtc
A = rtc()
DO i = 1, SIZE
    DO j = 1, SIZE
        N(i,j) = N(i,j) +  M(i,j)*L(i,j)
    ENDDO
END DO
B = rtc()
PRINT *, 'Seconds elapsed: ',B - A
END
```

```
$ rtc_time
 Seconds elapsed : 20.0878545045852661
```

2.2.3.3. irtc()

irtc() 함수는 rtc()와 똑 같은 방법으로 사용하지만 10억분의 일초 단위의 정밀도를 가지는 INTEGER(8) 값을 출력한다.

irtc()함수의 사용 예제(Fortran)와 실행결과

```
PROGRAM IRTC_TIME
INTEGER SIZE
PARAMETER(SIZE=5000)
REAL(8) N(SIZE,SIZE), M(SIZE,SIZE), L(SIZE,SIZE)
INTEGER(8) A, B, irtc
A = irtc()
DO i = 1, SIZE
    DO j = 1, SIZE
        N(i,j) = N(i,j) +  M(i,j)*L(i,j)
    ENDDO
END DO
B = irtc()
PRINT *, 'Nanoseconds elapsed: ',B - A
END
```

```
$ irtc_time
 Nanoseconds elapsed : 19747222262
```

2.2.3.4. MPI_Wtime

MPI는 고유의 timing 루틴을 가지고 있다. MPI 프로그램 내에서 특정 영역의 실행시간을 측정하기 위해 사용되는 MPI_Wtime()과 MPI_Wtick()을 소개한다.

① MPI_Wtime(): 루틴을 호출하는 현재 time의 값을 초 단위의 double precision 실수로 출력한다. 출력 값은 과거 어느 시점으로부터의 경과 시간을 나타내며 두 지점에서 출력된 time 값의 차이가 호출된 두 지점 사이의 실제 실행시간을 나타내게 된다.

② MPI_Wtick(): MPI_Wtime()의 호출에 의해 출력되는 time의 정밀도를 초단위로 출력한다. 출력되는 값은 연속되는 시간 간격을 초단위로 나타낸 것이다.

MPI timing 루틴의 사용에 사용된 예제(Fortran)와 실행결과

```
    program wtime_test
    include 'mpif.h'
    integer n,m,ierr
    double precision start,end,resolution
    call MPI_INIT(ierr)
    start = MPI_WTIME()
    do m=1,2000000
        n = n +  m
    end do
    end = MPI_WTIME()
    resolution = MPI_WTICK()
    print *,'Wallclock times(secs): start= ',start,'end=',end
    print *,'elapsed=',end-start,'resolution=',resolution
    call MPI_FINALIZE (ierr)
    end
```

```
$ wtime_test
 Wallclock times(secs): start=   1043828035.01158202 end= 1043828035.04210198
 elapsed= 0.30519962310791015E-01 resolution= 0.16199999999999995E-06
```

2.2.3.5. dtime()

dtime() 함수는 이전에 dtime()이 호출된 시점으로부터 경과된 유저 CPU time과 시스템 CPU time을 백분의 일초 단위 정밀도로 출력한다.

dtime()함수의 사용 예제(Fortran)와 실행결과

```
PROGRAM DTIME_TIME
REAL(4) DELTA, dtime
TYPE CT_TYPE
SEQUENCE
   REAL(4) USRTIME
   REAL(4) SYSTIME
END TYPE
TYPE (CT_TYPE) DTIME_STRUCT

INTEGER ARRAY_SIZE
PARAMETER(ARRAY_SIZE=2000)

REAL*8 a(ARRAY_SIZE,ARRAY_SIZE)
REAL*8 b(ARRAY_SIZE,ARRAY_SIZE)
REAL*8 c(ARRAY_SIZE,ARRAY_SIZE)

DELTA = dtime(DTIME_STRUCT)
DO i=1,2000
   DO j=1,2000
      c(i,j) = c(i,j) +  a(i,j) * b(i,j)
   ENDDO
ENDDO

DELTA = dtime(DTIME_STRUCT)
PRINT *, 'User time: ',DTIME_STRUCT%USRTIME, 'seconds'
PRINT *, 'System time: ',DTIME_STRUCT%SYSTIME, 'seconds'
PRINT *, 'Elapsed time: ',DELTA, 'seconds'
END
```

```
$ dtime_time
  User time: 1.570000052 seconds
  System time: 0.1599999964 seconds
  Elapsed time: 1.730000019 seconds
```

2.2.3.6. etime_()

etime() 함수는 프로세스가 시작된 시점으로부터 경과된 유저 CPU time과 시스템 CPU time을 백분의 일초 단위 정밀도로 출력한다

etime()함수의 사용 예제(Fortran)와 실행결과

```
PROGRAM ETIME_TIME
REAL(4) ELAPSED, etime
TYPE CT_TYPE
SEQUENCE
   REAL(4) USRTIME
   REAL(4) SYSTIME
END TYPE
TYPE (CT_TYPE) ETIME_STRUCT
```

```
INTEGER ARRAY_SIZE
PARAMETER(ARRAY_SIZE=2000)

REAL*8 a(ARRAY_SIZE,ARRAY_SIZE)
REAL*8 b(ARRAY_SIZE,ARRAY_SIZE)
REAL*8 c(ARRAY_SIZE,ARRAY_SIZE)

DO i=1,2000
    DO j=1,2000
        c(i,j) = c(i,j) +  a(i,j) * b(i,j)
    ENDDO
ENDDO

ELAPSED = etime(ETIME_STRUCT)
PRINT *, 'User time: ',ETIME_STRUCT%USRTIME, 'seconds'
PRINT *, 'System time: ',ETIME_STRUCT%SYSTIME, 'seconds'
PRINT *, 'Elapsed time: ', ELAPSED, 'seconds'
END
```

```
$ etime_time
  User time: 1.789999962 seconds
  System time: 0.1800000072 seconds
  Elapsed time: 1.970000029 seconds
```

2.2.3.7. timef()

timef() 함수는 처음 timef가 호출된 순간을 0로 하여 그 시간부터 경과된 시간을 1000분의 일초 단위로 출력한다.

timef()함수의 사용 예제(Fortran)와 실행결과

```
PROGRAM TIMEF_TIME
INTEGER SIZE
PARAMETER(SIZE=5000)
REAL(8) N(SIZE,SIZE), M(SIZE,SIZE), L(SIZE,SIZE)
REAL(8) elapsed, timef
elapsed = timef()
PRINT *, ' 'Start time : ', elapsed
DO i = 1, SIZE
    DO j = 1, SIZE
        N(i,j) = N(i,j) +  M(i,j)*L(i,j)
    ENDDO
END DO
elapsed = timef()
PRINT *, ' Elapsed time : ', elpased, 'milliseconds.'
END
```

```
$ timef_time
  Start time : 0.00000000000000000E+00
  Elapsed CPU time: 21120.7036972045898
```

2.2.4. Profiling

대부분의 코드는 많은 수의 서브프로그램으로 구성되며, 일부 서브프로그램에서 실행시간의 대부분을 소비하는 경우가 많다. 최적화가 필요한 부분에 대해 노력을 집중하기 위해 대부분의 시간이 어디에서 소비되고 있는지를 아는 것이 중요하다. 프로파일러는 각 서브프로그램이 소비하는 CPU 시간을 알려주는 툴이다. 컴파일러와 운영체제마다 다양한 프로파일러가 있지만 가장 많이 사용되는 것으로 prof과 gprof이 있다.

2.2.4.1. prof

prof 명령은 지정된 프로그램의 각 외부 루틴에 대한 CPU 사용의 프로파일을 표시하는데 자세히 설명하면 다음과 같다.

(1) 임의 루틴의 주소와 그 다음 루틴의 주소 사이에서 사용된 실행 시간의 퍼센트

(2) 함수가 호출된 횟수

(3) 단위 호출당 걸린 평균시간(ms)

prof 명령은 실행파일에 대한 monitor() 서브루틴에 의해 수집된 프로파일 데이터를 해석하여 이를 생성된 프로파일 파일(mon.out)과 관련시키며, 그 결과는 터미널로 출력되거나, 파일로 redirection할 수 있다.

prof 명령을 사용하려면 -p 옵션을 사용하여 C, FORTRAN, PASCAL, COBOL로 작성된 소스 프로그램을 컴파일한다. 이렇게 하면 monitor() 서브루틴을 호출하는 오브젝트 파일에 프로파일링 시작함수를 삽입한다. 프로그램이 실행되면 monitor() 서브루틴이 실행 시간을 트랙하기 위한 mon.out파일을 작성한다. 또한 -p 플래그는 컴파일러가 각각의 함수에 대해 생성된 목적코드에 mcount() 서브루틴을 호출하도록 한다. 프로그램이 수행되는 동안 상위함수가 하위함수를 호출할 때 하위 함수는 상위-하위 함수의 쌍에 대해 카운터를 증가시키는 mcount() 서브루틴을 호출한다. 기본적으로 표시되는 정보는 CPU time에 대한 퍼센트에 따라 정렬되며, 이는 -t 플래그를 지정할 때와 같다.

prof 명령과 같이 사용되는 여러 가지 플래그에 대한 설명은 다음과 같다.

-a : 루틴 주소가 증가하는 방향으로 정렬

-c : 호출 회수의 감소하는 방향으로 정렬

-n : 루틴 이름에 따라 정렬

-t : 전체 계산시간에 대한 백분율이 감소하는 방향으로 정렬(기본 플래그)

-o : 루틴 이름에 따라 각 루틴의 주소를 8진법으로 표시

-x : 루틴 이름에 따라 각 루틴의 주소를 16진법으로 표시

-g : 비전역 루틴을 포함

-h : 일반적으로 표시되는 헤드를 생략

-m *MonitorData* : mon.out 파일 대신 *MonitorData* 파일로 프로파일을 할 때 사용

-z : 0번 호출되고 0초 걸린 루틴에 대해서도 프로파일하고자 할 때 사용

다음 예는 Whetstone 벤치마크 프로그램을 -p 옵션을 주어 컴파일한 후 prof 명령을 수행한 결과의 앞부분을 보여준다.

```
$ xlc -O3 -qstrict -o cwhet -p -lm cwhet.c
$ cwhet > cwhet.out
$ prof
Name                  %Time      Seconds      Cumsecs   #Calls     msec/call
.main                 33.6         5.92         5.92         1     5920.
.__mcount             14.9         2.62         8.54
.P3                    7.4         1.30         9.84  89900000        0.0000
.log                   7.2         1.26        11.10   9300000        0.0001
.P0                    6.8         1.19        12.29  61600000        0.0000
.exp                   6.6         1.16        13.45   9300000        0.0001
._sqrt                 6.1         1.08        14.53
.cos                   6.1         1.07        15.60  19200000        0.0001
.PA                    5.3         0.93        16.53   1400000        0.0007
.atan                  3.6         0.64        17.17   6400000        0.0001
.sin                   2.5         0.44        17.61   6400000        0.0001
.__nl_langinfo_std     0.0         0.00        17.61         1        0.
.free                  0.0         0.00        17.61         2        0.
.isatty                0.0         0.00        17.61         1        0.
.__ioctl               0.0         0.00        17.61         1        0.
.ioctl                 0.0         0.00        17.61         1        0.
._findbuf              0.0         0.00        17.61         1        0.
._wrtchk               0.0         0.00        17.61         1        0.
.free_y                0.0         0.00        17.61         2        0.
.exit                  0.0         0.00        17.61         1        0.
.monitor               0.0         0.00        17.61         1        0.
.moncontrol            0.0         0.00        17.61         1        0.
.printf                0.0         0.00        17.61         3        0.
```

먼저 소스 코드를 -p 옵션을 주어 컴파일을 한 후 생성된 실행파일을 실행하면 mon.out파일이 생성된다. 그런 후 prof 명령을 수행하면 실행파일과 이 mon.out 파일을 읽어 들여 프로파일을 하게 되는데 정확하게 실행하려면 $ prof cwhet -m mon.out과 같은 식으로 하면 된다.

이 예제에서 P3(), log(), P0(), exp(), cos() 루틴에 대한 호출이 여러 번 나왔다. 그래서 먼저 소스코드를 확인하여 왜 그렇게 많이 사용되었는지 확인하고, 또한 왜 이러한 루틴들의 호출을 줄일 수 있는 방안을 고려하는 것이 곧 성능향상의 길이 될 것이다.

2.2.4.2. gprof

gprof 명령은 C, PASCAL, FORTRAN 또는 COBOL 프로그램을 프로파일 할 때 사용된다. 이러한 gprof 명령은 프로그램이 CPU 자원을 어떻게 사용하고 있는가를 살펴볼 때 유용하게 사용되

며, prof 명령보다 한 계 높은 추가적이고 가시적인 정보를 제공해 준다.

gprof 명령을 사용하려면 일단 소스 코드를 컴파일 할 때 -pg옵션을 사용하여야 한다. 이렇게 하면 컴파일러가 오브젝트 코드에 mcount() 함수의 호출을 삽입한다. mcount()함수는 상위함수가 하위함수를 호출할 때마다 카운트하고, 또한 monitor()는 각 루틴에서 걸린 시간을 측정한다.
gprof명령은 다음과 같이 2가지의 유용한 정보를 보여준다.

(1) 이 프로파일은 루틴을 CPU 시간 역순으로 그 하위 정보와 함께 보여준다. 어떤 루틴이 특정 루틴을 가장 많이 호출했고 특정 루틴에 의해 어떤 하위 루틴이 가장 자주 호출되었는지를 알 수 있다.

(2) prof 명령이 보여주는 정보와 유사하게 CPU 사용의 플랫 프로파일을 보여주며 이는 또한 루틴의 사용 및 호출회수에 따른 정보를 보여 준다.

gprof 명령과 같이 유용하게 사용되는 플래그에 대한 설명은 다음과 같다.
-b : 프로파일의 각 열에 있는 설명을 출력하지 않음
-e *Name* : *Name* 루틴과 모든 그 하위 루틴에 대한 프로파일을 출력하지 않음
-f *Name* : *Name* 루틴과 모든 그 하위 루틴에 대한 프로파일을 출력

소스코드를 -pg 옵션을 주어 컴파일한 후 실행을 하면 gmon.out 파일이 생성이 되는데, 이 파일에는 다음과 같은 정보가 binary형태로 저장되어 있다.

- 실행파일 및 공유 라이브러리 오브젝트 이름
- 각 프로그램 세그먼트에 할당된 가상 메모리 주소
- 각 상위-하위 루틴에 대한 mcount() 데이터
- 각 프로그램 세그먼트에 대해 누적된 시간(ms)

gprof 명령이 실행되면 실행파일과 gmon.out 파일을 함께 읽어 들여 call-graph 프로파일과 플랫 프로파일을 생성한다. 일반적으로 gprof 명령의 출력 내용이 아주 길기 때문에 임의의 파일로 redirection하여 저장한 후 열어보는 것이 좋다.

다음 예는 Whetstone 벤치마크 프로그램을 gprof 명령을 통해 프로파일하는 과정을 보여 준다.

```
$ xlc -O3 -qstrict -o cwhet -pg -lm cwhet.c
$ cwhet > cwhet.out
$ gprof cwhet > cwhet.gprof
```

앞서 설명한 것처럼 소스코드를 -pg 옵션을 주어 컴파일한 후 실행하면 gmon.out 파일이 생성됨을 확인 할 수 있다. 그런 후 위 예의 마지막 부분과 같이 실행하면 cwhet.gprof파일에 gprof명령의 결과들이 redirection되어 저장된다. cwhet.gprof 파일의 내용은 call-graph 프로파일, 플랫 프로파일 그리고 함수 이름 인덱스로 나눌 수 있는데 각각에 대해서 자세히 살펴보면 다음과 같

다.

Call-Graph 프로파일

call-graph 프로파일은 cwhet.gprof 파일의 처음 부분이며 다음과 같은 형식이다.

```
The sum of self and descendents is the major sort
            for this listing.

            function entries:

index       the index of the function in the call graph
            listing, as an aid to locating it (see below).

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

처음 부분은 각 열에 대한 설명을 나타내는 것으로 이는 앞에서 설명한 바와 같이 gprof 명령을 수행할 때 -b 플래그를 주어 실행하면 표시되지 않는다.

```
ngranularity: Each sample hit covers 4 bytes. Time: 21.09 seconds

                                     called/total       parents
index   %time     self descendents  called+ self     name              index
                                     called/total       children

                6.30        8.12         1/1             .__start [2]
[1]      68.4   6.30        8.12         1           .main [1]
                2.30        0.00 89900000/89900000       .P3 [4]
                1.33        0.00 9300000/9300000       .exp [5]
                1.03        0.00 9300000/9300000       .log [7]
                0.93        0.00 19200000/19200000       .cos [8]
                0.91        0.00 1400000/1400000       .PA [9]
                0.74        0.00 6400000/6400000       .atan [10]
                0.54        0.00 61600000/61600000       .P0 [11]
                0.34        0.00 6400000/6400000       .sin [12]
                0.00        0.00         3/3           .printf [22]
                0.00        0.00         2/2           .time [27]

-----------------------------------------------

6.6s                                                 <spontaneous>
[2]      68.4   0.00       14.42                 .__start [2]
                6.30        8.12         1/1             .main [1]
                0.00        0.00         1/1             .exit [33]

-----------------------------------------------

6.6s                                                 <spontaneous>
[3]      23.7   4.99        0.00                 .__mcount [3]
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

다음에는 각 함수에 대한 프로파일 정보가 보여지는데 이를 읽는 방법은 다음과 같다. 먼저 첫 번째 인덱스가 [1]이라고 표시 되어 있는데 이렇게 표시되어 있는 부분이 현재 함수를 나타낸다. 즉 .main함수가 현재 함수이고 이는 .__start[2]라는 상위함수가 호출을 하여 시작되었으며, 그 밑에 있는 .P3[4], .exp[5], .log[7] 등의 하위함수들을 호출하고 있다. 하위함수들은 위와 같이 현재함수의 아래에 위치해 있다. 현재함수에 대한 하위함수의 self열과 descendents 열에 있는 시간들이 모두 더해져서 현재함수의 descendents에 표시된다. 즉 위의 예에서는 하위함수인 .P3[4] ~ .time[27]함수들의 self 열과 descendents 열에 있는 시간들을 모두 더하면 .main[1]의 descendents에 표시되어 있는 8.12초가 된다.
그 아래에는 첫 번째 인덱스가 [2]라고 표시되어 있으며 이는 곧 .__start[2] 함수가 현재함수이면서 .main[1], .exit[33]이라는 하위함수를 호출하고 있다.
이와 같이 gprof 명령에 의해 수행한 프로그램의 함수들에 대한 상위-하위관계 즉 트리 구조를 일반 텍스트로 알 수 있으며 이러한 구조 내에서 각 함수들의 수행시간을 알 수 있다.

플랫 프로파일

플랫 프로파일은 prof 명령에 의한 결과와 상당히 유사하며, prof 명령에 의해 보여지는 정보에 몇 가지가 추가되어 있다.

```
flat profile:

 %            the percentage of the total running time of the
time          program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds     for by this function and those listed above it.

 self         the number of seconds accounted for by this
seconds       function alone.   This is the major sort for this
              listing.

calls         the number of times this function was invoked, if
              this function is profiled, else blank.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

ngranularity: Each sample hit covers 4 bytes. Time: 21.09 seconds

  %    cumulative    self                 self       total
 time    seconds    seconds      calls   ms/call   ms/call   name
 29.9        6.30      6.30          1   6300.00 14420.00   .main [1]
 23.7       11.29      4.99                                 .__mcount [3]
 10.9       13.59      2.30 89900000      0.00      0.00   .P3 [4]
  6.3       14.92      1.33   9300000      0.00      0.00   .exp [5]
  5.2       16.02      1.10                                 ._sqrt [6]
```

```
 4.9      17.05     1.03   9300000      0.00      0.00   .log [7]
 4.4      17.98     0.93  19200000      0.00      0.00   .cos [8]
 4.3      18.89     0.91   1400000      0.00      0.00   .PA [9]
 3.5      19.63     0.74   6400000      0.00      0.00   .atan [10]
 2.6      20.17     0.54  61600000      0.00      0.00   .P0 [11]
 1.6      20.51     0.34   6400000      0.00      0.00   .sin [12]
 1.2      20.77     0.26                                .__stack_pointer [13]
 0.9      20.96     0.19                                .qincrement1 [14]
 0.6      21.09     0.13                                .qincrement [15]
 0.0      21.09     0.00        11      0.00      0.00   .fwrite [16]
 0.0      21.09     0.00        11      0.00      0.00   .fwrite_unlocked [17]
 0.0      21.09     0.00        11      0.00      0.00   .memchr [18]
 0.0      21.09     0.00         3      0.00      0.00   ._doprnt [19]
 0.0      21.09     0.00         3      0.00      0.00   ._xflsbuf [20]
 0.0      21.09     0.00         3      0.00      0.00   ._xwrite [21]

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

이는 Call-Graph 프로파일 다음 부분이며, 마찬가지로 각 열에 대한 설명이 필요 없을 경우에는 gprof 명령을 수행할 때 -b 플래그를 주어 실행하면 된다. 여기서 보여지는 프로파일 결과 중에서 self seconds 열은 하위함수에서 걸린 시간은 제외한 단지 그 함수에서만 사용된 CPU 시간(초)을 의미하며, calls 열은 각 함수가 상위함수에 의해 호출된 회수를 의미한다.
일반적으로 목록의 위쪽에 있는 함수들이 최적화를 할 때 먼저 고려해야 할 대상이지만, 경우에 따라서는 이러한 함수들이 얼마나 많이 호출이 되었는지를 파악해야 한다. 즉 자주 호출되는 함수를 조금 향상시키는 것이 최적화에 있어서 아주 많을 영향을 끼칠 수 있다.

함수 이름 인덱스

gprof 명령의 마지막 부분에는 그 위에서 나온 함수들의 이름과 인덱스를 정리해 두었다. 다음과 같은 형식으로 정리되어 있으며, 함수 이름 순으로 정렬되어 있다.

```
Index by function name

  [11] .P0                    [8] .cos                   [38] .monitor
   [4] .P3                   [33] .exit                  [39] .myfcvt
   [9] .PA                    [5] .exp                   [40] .nl_langinfo
  [29] .__ioctl              [25] .free                  [41] .pre_ioctl
   [3] .__mcount              [26] .free_y               [22] .printf
  [30] .__nl_langinfo_std     [16] .fwrite               [15] .qincrement
  [13] .__stack_pointer       [17] .fwrite_unlocked      [14] .qincrement1
  [19] ._doprnt               [34] .ioctl                [12] .sin
  [31] ._findbuf              [35] .isatty               [23] .splay
   [6] ._sqrt                  [7] .log                  [27] .time
  [32] ._wrtchk                [1] .main                 [28] .time_base_to_time
  [20] ._xflsbuf              [18] .memchr               [24] .write
  [21] ._xwrite               [36] .mf2x1
  [10] .atan                  [37] .moncontrol
```

2.2.5. hpmcount

hpmcount는 사용자가 작성한 프로그램의 실제 실행 시간과 하드웨어 카운터에 관련된 정보, 사용 자원 등의 전반적인 성능을 제공하는 커맨드라인 유틸리티이다.

Sequential programs on AIX:

$hpmcount [-o <filename>] [-n] [-g <group>] <program>

Parallel programs (MPI) on AIX:

$poe hpmcount [-o <filename>] [-n] [-g <group>] <program>

-o <filename>: 출력파일 <filename>.<pid> 생성옵션. 병렬 프로그램에서는 프로세스마다 하나의 파일이 생성되며, 디폴트는 표준출력.

-n: 표준출력을 하지 않고 파일로만 출력. -o옵션과 같이 사용됨.

-g <group>: (POWER4 only) 0에서 60까지 그룹 지정 가능하며, 디폴트는 60.

hpmcount 사용 예제와 실행결과(-g 60은 없어도 됨)

```
$ hpmcount -o hpmtest -n -g 60 a.out
 Seconds elapsed:  20.1621870994567871
$ vi hpmtest_0000.384218

 hpmcount (V 2.4.3) summary

 Total execution time (wall clock time): 3.890695 seconds

 ########  Resource Usage Statistics  ########

 Total amount of time in user mode            : 3.800000 seconds
 Total amount of time in system mode          : 0.060000 seconds
 Maximum resident set size                    : 23564 Kbytes
 Average shared memory use in text segment    : 3088 Kbytes*sec
 Average unshared memory use in data segment  : 9007560 Kbytes*sec
 Number of page faults without I/O activity   : 5893
 Number of page faults with I/O activity      : 0
 Number of times process was swapped out      : 0
 Number of times file system performed INPUT  : 0
 Number of times file system performed OUTPUT : 0
 Number of IPC messages sent                  : 0
 Number of IPC messages received              : 0
 Number of signals delivered                  : 0
 Number of voluntary context switches         : 4
 Number of involuntary context switches       : 3

 #######  End of Resource Statistics  ########

  PM_FPU_FDIV (FPU executed FDIV instruction)                  :           0
  PM_FPU_FMA (FPU executed multiply-add instruction)           :  1000002329
  PM_FPU0_FIN (FPU0 produced a result)                         :   627844244
```

```
PM_FPU1_FIN (FPU1 produced a result)                    :      1377820044
PM_CYC (Processor cycles)                               :      3758316603
PM_FPU_STF (FPU executed store instruction)             :      1003408033
PM_INST_CMPL (Instructions completed)                   :      4143170873
PM_LSU_LDF (LSU executed Floating Point load instruction) :    2002211584

Utilization rate                                        :         74.306 %
                                                          ①
Load and store operations                               :       3005.620 M
                                                          ②
Instructions per load/store                             :          1.3786
                                                          ③
MIPS                                                    :       1064.892
                                                          ④
Instructions per cycle                                  :          1.102
                                                          ⑤
HW Float points instructions per Cycle                  :          0.534
                                                          ⑥
Floating point instructions + FMAs                      :       2002.259 M
                                                          ⑦
Float point instructions + FMA rate                     :     514.627 Mflip/s
                                                          ⑧
FMA percentage                                          :         99.887 %
                                                          ⑨
Computation intensity                                   :          0.666
                                                          ⑩
```

위의 실행결과에서 수집된 정보에 대해 살펴보면 다음과 같다.

① Utilization rate = User time / Wall clock time

② Load and store operations = Loads + Stores(Total LS)
2002211584 + 1003408033 = 3005619617 = 3005.620 M

③ Instructions per load/store = Instructions completed / Total LS
4143170873 / 3005619617 = 1.3786

④ MIPS = 0.000001 * Instructions completed / Wall clock time
0.000001 * 4143170873 / 3.890695 = 1064.892
※ MIPS(Millions of Instructions per second)

⑤ Instructions per cycle = Instructions completed / Cycles
4143170873 / 3758316603 = 1.102

⑥ HW Float points instructions per Cycle = (FPU 0 + FPU 1) / Cycles
(627844244 + 1377820044) / 3758316603 = 0.534
※ FPU (Floating-point Processing Unit : 부동 소수점 연산장치) POWER4에는 프로세서 하나에 2개씩 있음)

⑦ Floating point instructions + FMAs (flip) = FPU 0 instructions + FPU 1 instructions + FMAs executed – FPU Stores (POWER4)
627844244 + 1377820044 + 1000002329 – 1003408033 = 2002258584 = 2002.259 M
※ FMA (Floating-point Multiply/Add), FDIV(Floating-point Divide)
※ POWER3 : flip = FPU 0 instructions + FPU 1 instructions + FMAs executed

⑧ Float point instructions + FMA rate = 0.000001 * flip / Wall clock time (Mflip/s)
0.000001 * 2002258584 / 3.890695 = 514.627 Mflip/s

⑨ FMA percentage = 100 * FMAs executed * 2 / flip
100 * 1000002329 * 2 / 2002258584 = 99.887 %
⑩ Computation intensity = flip / Total LS
2002258584 / 3005619617 = 0.666

2.3. 병렬 성능: 성능향상도와 확장성

병렬 프로그램 성능은 프로그램 실행시간외에 성능향상도와 확장성에 의해서도 판단될 수 있다. 성능향상도는 순차 프로그램의 실행시간(T_s)과 N개 프로세서를 사용해 실행된 병렬 프로그램의 실행시간($T_p(N)$) 사이의 비로 정의 된다.

성능향상도(Speedup): $S(N) = T_s/T_p(N)$

N개의 프로세서를 사용한 병렬 프로그램의 성능향상도가 N에 가까울수록 그 병렬 프로그램은 확장성이 좋다. 또, 프로세서 개수와 문제크기를 각각 N배 증가 시켜도 실행시간이 변화가 없다면 그 병렬 프로그램은 확장성이 좋다고 할 수 있다.
보다 일반적으로 프로그램의 성능향상도는 원래의 코드와 비교해 성능 최적화(병렬화는 최적화의 한 방법) 시킨 코드가 얼마나 빨라 졌는가를 측정하게 된다.

성능향상도(Speedup) = T_{orig}/T_{new}

2.3.1. 확장성에 제한을 주는 요인들

병렬 프로그램의 확장성에 제한을 주는 요인들로 Amdahl의 법칙, 통신 부하, 부하 불균형 등이 있다.

2.3.1.1. Amdahl's Law

Amdahl의 법칙은 최적화에 의해 얻을 수 있는 성능향상 정도는 최적화를 적용할 수 있는 코드 부분(fraction)에 의해 제한 받는다는 것을 기술한다. 성능향상에 영향을 주는 두 가지 요인으로 다음 두 가지를 들 수 있다.

- 성능이득을 얻기 위해 최적화될 수 있는 원(original) 코드의 계산시간 부분(Fraction)
- 최적화에 의해 얻어진 성능이득(최적화된 코드 부분이 얼마나 더 빨라 졌는가)

다음과 같이 성능향상도를 결정할 때

$$Speedup_{overall} = T_{orig} / T_{new}$$

최적화된 프로그램의 실행시간 T_{new}는 다음과 같이 주어진다.

$$T_{new} = T_{orig} \times (1 - Fraction_{enhanced}) + \left(\frac{T_{orig}}{Seedup_{enhanced}} \right) \times Fraction_{enhanced}$$

병렬화에 의한 성능향상의 경우 순차영역의 실행 시간을 T_s, 병렬화 가능한 부분의 실행시간을 T_p, 그리고 프로세서 개수 N에 대해, $Speedup_{enhanced} = N$ 이라고 하면, N개의 프로세서를 사용한 병렬 프로그램의 성능향상도는 다음과 같다.

$$Seedup(N) = \frac{\left(T_s + T_p\right)}{\left(T_s + \frac{T_p}{N}\right)} \le \frac{T(1)}{T_s}$$

2.3.1.2. Communication Overhead

병렬화된 코드의 성능향상도는 통신부하에 의해 제한 받을 수 있다. 통신 비용을 프로세서개수 N에 대한 로그로 가정하면 성능향상도는 다음과 같다.

$$\begin{aligned} Speedup(N) &= T(1)/(T_s + T_p/N + c\log N) \\ &= O(1/\log N) \end{aligned}$$

여기서, 프로세서 개수(N)를 무한대로 증가시키면 성능향상도는 0으로 수렴하게 됨을 알 수 있다.

2.3.1.3. Load Imbalance

- 부하 불균형(load imbalance)은 충분하지 못한 병렬성이나 동일하지 않은 작업 크기로 인해 프로그램 실행 도중 일부 프로세서가 대기(idle) 상태에 있게 되는 것을 말한다. 동일하지 않은 작업 크기는 특정 영역에 종속되는 계산이나 트리 구조의 계산과 같이 구조적으로 균등 분배가 어려운 계산 등으로 인해 발생할 수 있다.

2.4. 성능 최적화 과정

병렬 코드의 성능에 가장 큰 영향을 미치는 것은 순차코드의 성능이다. 따라서, 순차 코드에 대한 성능 최적화는 병렬 코드 성능 최적화를 위해서도 필수적이다.
병렬화를 포함하여 코드의 최대 성능을 이끌어내기 위한 최적화 과정은 다음과 같이 정리해 볼 수 있다.

- 클러터 제거
- 컴파일러를 이용한 최적화
- 고성능 라이브러리 사용

- Hotspot을 찾기 위한 프로파일
- 성능 데이터를 활용해 많은 실행시간을 소비하는 코드 블록에 대한 최적화
- 병렬화
- 병렬코드 최적화

3. 컴파일러 성능 최적화

최적화 컴파일러의 목표는 고급 언어로 작성된 코드를 정확하게 그리고 가능한 가장 빠른 기계어로 효율적인 번역을 하는 것이다. 좋은 컴파일러는 이러한 번역을 통해 소스코드에서 얻고자 하는 정확한 답을 더 빠르게 얻을 수 있도록 해준다.
올바른 결과를 내지 못한다면 프로그램이 얼마나 빠르게 실행되느냐 하는 것은 실질적으로 중요하지 않다. 그러나, 최적화 컴파일러는 올바른 결과를 내도록 표현하는 범위 내에서 프로그램 코드를 능률화하는 방법을 찾는다. 이러한 능률화 과정으로 코드 단순화, 관계없는 명령어 제거, 같은 결과를 필요로 하는 여러 문장에서 중간 계산 결과에 대한 공유 등을 생각해 볼 수 있다. 좀 더 진보적인 최적화 과정에서 컴파일러는 프로그램 코드를 재구성할 수 있으며 그 결과로 실행 명령어 개수가 감소하더라도 전체 코드의 크기가 증가할 수도 있다.
번역된 기계어 코드를 완성할 때 컴파일러는 명령어를 꺼내는 규칙과 레지스터에 대해 알고 있어야 하고, 프로그램 성능을 위해 명령어들의 처리 시간과 파이프라인과 같은 시스템 자원의 지연시간에 대해서도 알고 있어야 한다. 이러한 사실들은 단위 시간당 하나 이상의 명령어를 처리할 수 있는 프로세서들에게는 더욱 중요하다. 컴파일러는 프로그램이 실행되는 동안 시스템 상태를 항상 '바쁨(busy)'으로 유지시키기 위해 부동소수, 고정소수, 메모리, 그리고 분기(branch) 연산 들이 적절히 배치되는 균형 잡힌 명령어 혼합을 이루도록 기계어 코드를 완성해야 한다.
과거의 컴파일러는 단지 어셈블리 언어보다 읽기 쉬운 코드를 작성할 수 있게 해주는 도구 정도였다. 고급언어로 작성된 코드를 단일 프로세서 아키텍처에서 다중 프로세서 아키텍처에 이르기까지 다양한 플랫폼에 맞는 최적화된 기계어 코드로 번역하는 오늘날의 컴파일러는 인공 지능의 경계선에 위치해 있다. 고성능 컴퓨팅 분야에서 컴파일러는 때때로 프로세서나 메모리 아키텍처보다 프로그램의 성능에 더 큰 영향을 주기도 한다. 과거의 고성능 컴퓨팅 분야에서는 고급언어로 작성된 코드의 성능이 만족스럽지 못할 경우 프로그램의 일부 또는 전체를 어셈블리 언어로 다시 작성하기도 했으나 오늘날에는 컴파일러 성능의 발달로 그런 단계가 불필요하게 되었다.
컴파일러 옵션을 이용한 성능 최적화는 프로그램 성능을 향상시키기 위해 시도하는 첫 번째 방법으로, 프로그래머는 최소의 노력으로 상당한 성능향상 효과를 얻을 수 있다. 때때로 과도한 컴파일러 최적화가 엉뚱한 결과를 내 놓거나 성능을 오히려 나쁘게 할 수 있으므로 새로운 컴파일러 옵션을 사용한다면 프로그램 성능과 결과의 정확성 여부를 항상 검토해야 한다.
여기서는 최적화 컴파일러가 코드 최적화를 위해 수행하는 기본적인 기능에 대해 알아 볼 것이다.

3.1. 컴파일러의 역할과 컴파일 과정

3.1.1. 컴파일러의 역할

컴파일러의 역할은 프로그램의 higher-level, abstract representation을 특정 명령어 집합 아키텍처(ISA)로 변환하는 것으로, 변환과정에서 다음과 같은 몇 가지 목표를 추구한다.

- 정확하게 컴파일된 프로그램

- 효과적으로 컴파일된 프로그램
- 빠른 컴파일
- 디버깅과 성능 분석 지원

이러한 목표들은 서로 상충될 수 있다.

3.1.2. 컴파일 과정

컴파일러는 대개 2~4개의 pass 또는 phase로 구성되며, 최적화 수준이 높을수록 더 많은 pass를 가지며, 앞선 pass에서 진행된 변환 과정으로 다시 되돌아가지 않는다. 대부분의 컴파일러는 전체적으로 다음 그림과 같은 컴파일 과정을 거치게 된다.

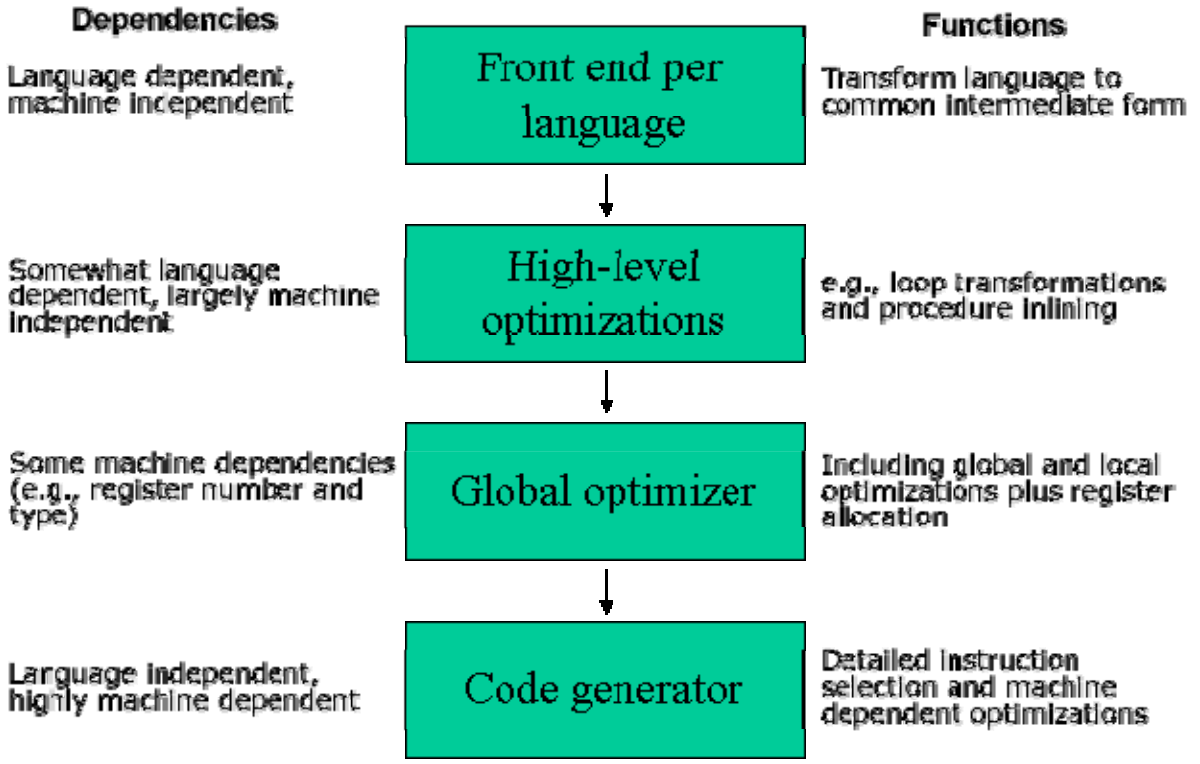


그림 3-1. 컴파일 과정

컴파일러의 Front end는 어휘(lexical) 분석과 프로그램의 구문 분석(parsing)을 하고 소스코드를 최적화를 위해 준비하는 Intermediate Language(IL)로 변환한다.

1. Precompiler or preprocessor phase: 소스 코드에 대한 simple textual manipulation
2. Lexical analysis phase: source 문장들을 변수, 상수, 주석, language 요소들과 같은 token들로 분해
3. Parsing phase: 입력이 문법적으로 검사되며 컴파일러는 프로그램을 최적화 준비단계인 IL로 전환

중간 언어는 같은 계산을 원본 프로그램처럼 표현하지만 컴파일러가 좀더 쉽게 처리할 수 있는 형태이다. 고수준 최적화를 거친 코드는 global optimizer에 의해 다시 최적화 된다. 코드 발생기(code generator)는 프로세서의 아키텍처 특성을 고려해서 intermediate 언어를 목적 코드로 변환한다.

1. IL 코드에 대해 한 번 또는 다수의 최적화 수행
2. 목적코드 발생기가 IL 코드를 어셈블리 코드로 전환하며 이 과정에서 특정 프로세서 아키텍처에 대한 고려를 하게 된다.

3.1.2.1. Intermediate Language (IL) Representation

Parse 단계의 출력물인 IL 코드는 원본 프로그램과 같게 표현되며 컴파일러가 보다 쉽게 처리할 수 있는 형태로 구성된다. 배열 참조에 대한 주소 표현과 같은 소스에 없는 명령어가 나타나며 이후 코드 나머지 부분에서도 이 명령어들을 계속 볼 수 있다. 이것들은 최적화 과정에서 사용된다. IL은 기계어와 매우 유사하여 바로 목적코드로 변환이 가능하다. 실제 최저 수준의 최적화는 IL을 기계어로 단순히 변환만 한 것으로 코드가 크고 성능은 좋지 않다.

Fortran 코드

```
DO WHILE (j < n)
        k = k +  j*2
        m = j*2
        j = j +  1
ENDDO
```

IL 코드

```
A ::    t1 := j
        t2 := n
        t3 := t1 < t2
        jmp (B) t3
        jmp (C) TRUE
B ::    t4 := k
        t5 := j
        t6 := t5 * 2
        t7 := t4 +  t6
        k := t7
        t8 := j
        t9 := t8 * 2
```

```
        m := t9
        t10 := j
        t11 := t10 + 1
        j := t11
        jmp (A) TRUE
C ::
```

3.1.2.2. Basic Blocks

IL 코드를 생성한 후, 이것을 basic block으로 분해한다. Basic block은 one entrance(at the top)와 one exit(at the bottom)를 가지는 코드 블록이다. 이러한 basic block은 코드 분석을 보다 쉽게 한다.

```
A ::
    t1 := j
    t2 := n
    t3 := t1 < t2
    jmp (B) t3
```

```
    jmp (C) TRUE
```

```
B ::
    t4 := k
    t5 := j
    t6 := t5 * 2
    t7 := t4 + t6
    k := t7
    t8 := j
    t9 := t8 * 2
    m := t9
    t10 := j
    t11 := t10 + 1
    j := t11
    jmp (A) TRUE
```

3.2. 컴파일러 최적화

언어 의존성의 정도와 그 효과의 범위에 의해 구분되는 몇 가지 컴파일러 최적화 class가 있다. 여러 분기점에 걸친(across branches) 최적화나 서브루틴 경계를 넘어선(across subroutine boundaries) 최적화와 같은 더 넓은 범위의 최적화는 더 까다롭고 컴파일 시간이 더 걸린다. 그렇지만 잠재적으로 보다 큰 성능향상 효과를 기대할 수 있다.

- high-level optimizations
- Local optimizations
- Global optimizations
- Register allocations
- Processor-dependent optimizations
- Loop optimizations
- Inter-procedural optimizations

3.2.1. High-Level Optimizations

Source level에서 행해지는 변환이고 프로세서에 독립적이다. 예로, 프로시저 호출을 프로시저 본체로 대체하는 Procedure 인라이닝을 들 수 있다.

3.2.2. Local optimizations

코드내의 basic block에서 수행되는 최적화이다.

- common sub-expression 제거: 동일의 중복된 instances를 임시 저장 결과를 가지는 하나의 계산으로 대체

```
D = C*(A+ B)
E = (A+ B) /2.
```

```
temp = A+ B
D = C*temp
E = temp/2.
```

- Constant propagations: 상수만으로 표현된 식을 컴파일 단계에서 계산하여 상수로 대체

```
INTEGER I, K
PARAMETER (I=100)
K = 200
J = I+ K
```

```
INTEGER I, K
PARAMETER (I=100)
K = 200
J = 300
```

- Dead code 제거: 프로그래머에 의해 또는 컴파일러에 의해 생성된 dead code 제거. dead code는 실행되지 않는 명령어와 사용되지 않는 결과를 생성하는 명령어로 구분 가능

```
PROGRAM MAIN
I = 2
WRITE (*,*) I
STOP
I = 4       ! Unreachable
WRITE (*,*) I
END
```

- Stack height reduction: expression evaluation에 필요한 리소스를 최소화 하기 위해 expression tree를 재 정렬

3.2.3. Global Optimizations

Local optimization을 여러 분기에 걸친(across branches) 수준으로 확장하는 것이며. 이 단계에서는 루프 최적화가 목표가 된다.

- global common sub-expression(across branches) 제거
- Copy Propagations: 컴파일러가 A=X와 같이 X가 할당된 변수 A의 모든 instances를 X로 대체

```
x = y
z = 1.0 + x
```

```
x = y
z = 1.0 + x
```

- Code motion: 반복 계산에서 같은 값을 계산하는 코드를 루프 밖으로 이동

```
DO I=1,N
    A(I) = B(I) +  C*D
    E = G(K)
ENDDO
```

```
temp = C*D
DO I=1,N
    A(I) = B(I) +  temp
ENDDO
E = G(K)
```

- Induction variable elimination: 루프내의 배열 주소 계산을 단순화 또는 제거

```
DO I=1,N
    K  =  I*4  +  M    ! K is induction
variable
    ...
ENDDO
```

```
K = M
DO I=1,N
    K = K +  4
    ...
ENDDO
```

3.2.4. Register Allocation

자주 사용되는 변수들을 빠르게 접근하기 위해 프로세서 레지스터에 저장되도록 하는 것이다. 레지스터 할당은 레지스터와 관련 연산을 묶어 다른 최적화를 유용하게 하고 코드 속도를 증가하는데 중심적인 역할을 한다. 최적의 레지스터 할당은 NP-complete 문제이다. 경험적으로 레지스터 할당은 최소 16개의 general-purpose 레지스터와 부동소수 연산을 위한 추가적인 레지스터가 있을 때 가장 좋은 결과를 얻을 수 있으며, global 변수보다는 stack-allocated objects에서 보다 효과적이다. aliased 변수(여러 이름을 가지는 변수)들은 레지스터에 올 수 없음을 기억하라.

3.2.5. Processor-dependent Optimizations

코드 전체의 성능향상을 위해 특정 아키텍처의 특성을 사용하는 것이다.

- strength reduction: expensive 연산을 덜 expensive한 연산으로 대체, 예: 상수 곱셈을

add와 shift로 대체

- pipeline scheduling: 파이프라인 성능 향상을 위해 명령어 순서를 바꿈

3.2.6. Loop 최적화

소스 코드의 루프에 대해 high-level 변환을 수행해 캐시 utilization과 명령어 scheduling의 개선을 시도한다. single 루프 최적화와 nested loop 최적화가 있다.

3.2.7. Inter-procedural Optimization (IPO)

전체 프로그램을 대상으로 한번에 하나 이상의 루틴을 분석하고, 루틴 영역을 넘어 최적화 수행을 시도한다. 의존성 분석 정보를 서브루틴 경계를 넘어(across routine boundaries) 전달할 수 있고 이 정보를 루프 nest 최적화, 소프트웨어 파이프라이닝 등과 같은 이어지는 최적화 과정에서 사용하도록 한다.

3.3. 컴파일러 성능 최적화 기법

3.3.1. Microprocessor Chip Related Options

대부분의 컴파일러는 전체 family of chips에 대한 코드 생성이 가능하도록 개발한다. 특정 시스템에서 최적의 성능을 얻기 위해서는 코드가 실행될 특정 chip에 대한 정보를 컴파일러에게 알려줘야 한다. 이렇게 함으로서 Chip family에 대한 Generic 어셈블리어 코드를 생성하는 대신 특정 마이크로 프로세서 칩의 하드웨어적인 특성을 효과적으로 모두 사용하는 것이 가능해 진다. 이러한 마이크로 프로세서 옵션들은 Chip name, 버전, 아키텍처, 캐시크기, 타깃 시스템 타입 등을 필요로 할 수 있다. 이러한 정보들은 시스템 자료를 참고하거나 적절한 유닉스 명령어를 통해 알아볼 수 있다.

UNIX 명령어 uname (-m 또는 -p 옵션 사용)은 시스템/프로세서 정보를 준다.

컴파일러의 기본 타깃은 현재 컴파일러가 실행되는 프로세서가 된다.

3.3.1.1. Pipelined Functional Units

마이크로 프로세서의 Functional Units은 기본 연산을 수행한다. 두 개의 functional unit을 가지는 프로세서를 보면, 실수 연산을 하는 것 하나와 decision-making 연산을 하는 것 하나로 이루어 진다. 파이프라인된 functional unit은 계산을 위한 어셈블리 라인처럼 작동한다.

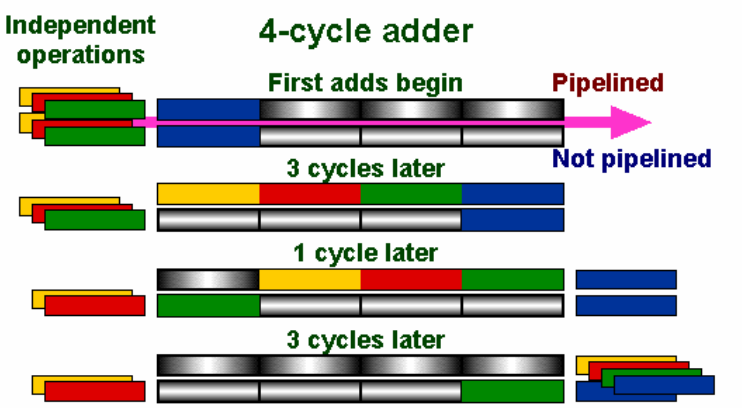


그림 3-2. Functional unit이 덧셈을 수행하는 그림. 각 덧셈은 4 사이클(clock 주기) 걸린다.

속도 때문에 현재의 high-end 마이크로 프로세서에서 파이프라인 로직은 하드웨어에서 수행되고, 소프트웨어를 통해 제어되지 않는다. 따라서 컴파일러에는 파이프라이닝 관련 옵션이 없다. 그러나, 프로그래머는 어떤 연산이 파이프라인 되는지 안 되는지를 알아서 가능한 한 파이프라인된 연산을 이용해 코드를 작성하는 것이 좋다.

3.3.1.2. Chip Registers

마이크로 프로세서 칩의 또 다른 항목이다. Functional unit이 곱셈과 같은 연산을 수행할 수 있기 전에 해당 연산수가 레지스터에 있어야 하며, 데이터는 캐시로부터 레지스터로 옮겨지거나 메모리에서 올려지게 된다.

루프 계산에서는 루프 카운터가 필요하며 각 반복마다 카운터 변수(i,j,k…)는 증가되고 테스트 된다. 루프 카운터 변수는 임시 변수이고, 이로 인해 메모리나 캐시 공간을 가급적 낭비하지 않는 것이 좋다. 카운터 변수가 사용될 동안 레지스터에 있게 되면, 레지스터 기억(register storage)이 모든 반복에서 카운터 변수에 대한 두 연산의 속도를 증가 시키게 된다. 일부 프로세서는 특별히 루프 카운터 변수를 위해 설계된 fetch and increment 레지스터를 가진다. 그러한 레지스터에서 카운터 변수가 검색될 때 검색과 동시에 1이 증가된다.

만약 컴파일러 옵션을 통해 레지스터 기억을 지정하는 것이 가능하다면 그렇게 하는 것이 좋다. 다른 방법으로는 프로그램 자체에서 레지스터 기억을 사용하도록 하는 것이다. C나 C++에서는 다음과 같이 레지스터 기억 클래스 키워드(register storage class keyword)를 사용할 수 있다.

```
register int i,j;
```

이와 같은 선언을 통해 컴파일러에게 레지스터 기억(register storage)을 사용하도록 요청하지만, 실제로 컴파일러가 레지스터 기억을 사용할 것이 보장되지는 않는다.

3.3.1.3. Prefetching Data

데이터 선인출(prefetching)은 데이터가 필요한 시점 이전에 데이터에 접근하는 것이다. 선인출로 인해 칩이 명령어를 분석하고 해당 명령을 실행하는 중에 이미 데이터를 가지고 있게 되며, 동시에 다음에 필요한 데이터가 올라온다. 계산과 데이터 이동이 동시에 수행되어 latency hiding을 구현할 수 있다.

현재의 마이크로 프로세서 칩에서 선인출 명령어는 어셈블리 언어 명령어 집합의 일부이다. 컴파일러는 코드를 분석해서 적절한 지점에 선인출 명령어를 자동으로 삽입한다. 컴파일러 옵션을 통해 첫 단계의 선인출을 설정해보고 점차 단계를 높여 볼 수 있다. 대부분의 컴파일러는 선인출에 있어 매우 aggressive 하다.

3.3.2. All-in-One 최적화 옵션

컴파일러에는 다수의 영역에서 최적화를 하는 generic 옵션이 있다. 그렇지만 코드의 특정 부분을 완전히 최적화하기 위해서는 다른 옵션을 추가해야 한다. 이러한 최적화 옵션은 대부분의 컴파일러에서 -On의 형태를 가지며, n은 일반적으로 0에서4 사이의 값을 가진다. 이 값이 클수록 더 많은 최적화를 수행하게 된다. 각 단계(n의 값)에서 어떤 작업이 수행되는지 특정 상황에 적합한지 알아야 한다(컴파일러 매뉴얼 참조). 예로, 최고 수준의 -O 옵션 단계에서는 루프 병렬화를 시도하지만, 만약 코드가 단일 프로세서에서 실행된다면 이러한 작업은 무의미하다.

컴파일러의 generic 최적화에 대해서 default 수준을 알고 있어야 한다. 컴파일러마다 최적화 수준을 결정하는 기본 값이 다를 수 있는데, 일부 컴파일러는 매우 conservative해서 기본 단계를 낮추는 것이 필요하다. 그렇지만, 좀더 aggressive한 접근을 한다면 그 값을 크게 설정할 필요가 있다.

대부분 컴파일러에서 -O3 옵션이 실질적으로 무난하며, 이러한 최적화 generic 옵션은 각 수준별로 각각의 구성 옵션으로 분리할 수 있다.

대부분 컴파일러에서 -O0 옵션은 모든 최적화 기능을 사용하지 않도록 하는 것을 의미한다. 최적화가 전혀 없는 코드의 실행시간을 측정함으로써 컴파일러와 칩 수준의 하드웨어 최적화 조합을 통해 얻어지는 코드의 성능 향상 정도를 정량화할 수 있다.

3.3.3. Inter-procedural options (IPO)

IPO는 프로그램에서 사용되는 프로시저에 대한 최적화를 목표로 하는 옵션들의 집합이다. 특별히 IPO는 프로시저들과 다른 프로시저들과의 상호작용(메인 프로그램에서 다른 함수의 참조 등)을 제어한다.

3.3.3.1. Automatic 인라이닝

함수 호출 자체를 없애 함수 호출의 부하를 제거하는 것이다. 어셈블리 언어 코드에서는 함수 호출부분에 함수 자체를 삽입시킨다. 인라이닝 기법을 통해 많은 성능향상을 볼 수 있다.

프로그램 작성을 모듈화하여 접근하는 것이 편리하지만 인라이닝은 이러한 모듈화 접근과는 반대

의 입장으로, 이러한 인라이닝을 컴파일러가 자동으로 수행 하도록 하는 것이 프로그래머에게는 편리하다. 인라이닝 옵션에 의해 컴파일러는 인라인되어야 하는 함수를 찾고 자동으로 그 함수에 대한 각 호출마다 인라인한다.
컴파일러가 인라인되는 함수를 결정하는 방식은 어떤가? 함수가 그 연산을 실행하는데 걸리는 시간, 함수 호출의 회수, 추정되는 부하에 기반한 인라인 factor를 계산한다. 이 인라인 factor가 충분히 크면 그 함수는 자동적으로 인라인된다.

3.3.3.2. Disabling Recursion

재귀적 호출을 하는 함수는 매우 느리게 실행된다. 각 함수 호출이 리턴 되지 않은 상태(open)에 있기 때문인데, 이로 인해 마지막 호출에서 스택 아래의 이전 호출을 닫기 시작할 때까지 반복적으로 호출되는 함수 각각을 모두 스택에 쌓아두게 된다. 전체 과정에서 각 호출마다 모든 메모리 참조가 필요하며 많은 시간을 소비하게 된다.
모든 재귀적 알고리즘은 단순 루프 접근 방식으로 대체될 수 있다. Factorial 함수의 루프 버전과 recursive 버전을 비교하면 recursive 버전이 대략 1000배 정도 더 느리다. Recursive 알고리즘이 보기 좋고 간략하지만 성능을 고려한다면 사용을 가급적 자제하는 것이 좋다. IPO에는 recursion을 없앨 수 있는 기능이 있다.

3.4. Single Loop Transformations

대부분의 컴파일러는 루프내의 명령어 수준 병렬성을 개선하기 위해 다음과 같은 여러 가지 형식의 단일 루프 변환을 이용한다.

- Induction variable Optimization
- Prefetching
- Test promotion in loops
- Loop peeling
- Loop fusion
- Loop fission
- Copying
- Block and copy
- Loop unrolling
- Software pipelining
- Loop invariant code motion
- Array padding
- Optimizing reductions

3.4.1. Induction variable optimization

복잡한 루프 변수 표현을 단순한 표현으로 대체하는 것이다.

```
DO I = 1, n
    J = 2*I-1
  ...
ENDDO
```

위의 코드를 다음과 같이 변환한다.

```
J=-1
Do I = 1, n
    J = J + 2
  ...
ENDDO
```

주소 레지스터가 더 빠르게 증가 되므로 루프 실행에 있어 성능증가 효과를 확실히 볼 수 있다.

3.4.2. 선인출 (Prefetching)

데이터를 사용하기 전에 미리 메모리에서 캐시로 옮겨 메모리 지연을 숨긴다. 데이터가 캐시크기보다 너무 큰 compute-intensive 코드에 유용하며, 데이터 액세스가 일정한 패턴을 가지는 경우 효과적이다. 하드웨어에서 지원될 수도 있지만, 컴파일러는 선인출 명령어를 프로그램에 넣거나 load 명령어를 schedule에서 보다 앞쪽으로 옮겨서 선인출을 구현한다.

3.4.3. Test Promotion

각 판단(test) 구문의 분기마다 원래의 루프를 복제하는 방법으로 판단(test) 구문을 루프 밖으로 옮긴다. 복제된 루프는 원래의 루프보다 판단구문이 더 적거나 아예 없게 돼 더 빠르게 실행된다.

```
DO I = 1, 100
    IF(VAL) THEN
        A(I) = B(I)
    ELSE
        A(I) = C(I)
    ENDIF
ENDDO
```

위의 코드는 다음과 같이 변환돼 더 빠르게 실행될 수 있다.

```
IF(VAL) THEN
   DO I = 1, 100
      A(I) = B(I)
   ENDDO
ELSE
   DO I = 1, 100
      A(I) = C(I)
   ENDDO
ENDIF
```

3.4.4. Loop peeling

루프에서 특정 반복을 제거하는 것이다. 이때 제거되는 것은 대부분 처음과 마지막 반복이 된다. Loop peeling은 Loop fusion을 방해하는 데이터 의존성을 제거하기 위해 사용한다.

```
DO I = 1, N
   A(I) = B(I+ 1) +  B(I-1)
ENDDO
DO I = 1, N
   B(I) = A(I+ 1) +  A(I-1)
ENDDO
```

위의 코드에서 두 루프는 다음과 같이 peeling을 거쳐 하나로 합쳐질 수 있다(fusion).

```
A(1) = B(2) +  B(0)
DO I =2, N
  A(I) = B(I+ 1) +  B(I-1)
  B(I-1) = A(I) +  A(I-2)
ENDDO
B(N) = A(N+ 1) +  A(N-1)
```

3.4.5. Loop unrolling

루프의 각 반복에는 루프 카운터 변수를 변화시키고 그 값이 정지 값에 도달했는가 알아보기 위한 추가 시간이 소비된다. 이 시간은 loop overhead가 된다. Loop unrolling은 루프 반복을 줄여서 루프 오버헤드를 줄인다.

```
DO I = 1, 160
    A(I) = B(I) +  C(I)
ENDDO

Unrolling by a factor of 2
DO I = 1, 80, 2
    A(I) = B(I) +  C(I)
    A(I+ 1) = B(I+ 1) +  C(I+ 1)
ENDDO

Unrolling by a factor of 4
DO I = 1, 40, 4
    A(I) = B(I) +  C(I)
    A(I+ 1) = B(I+ 1) +  C(I+ 1)
    A(I+ 2) = B(I+ 2) +  C(I+ 2)
    A(I+ 3) = B(I+ 3) +  C(I+ 3)
ENDDO
```

컴파일 옵션을 이용해 컴파일러가 unrolling을 수행하도록 하며 unrolling factor를 명시적으로 줄 수 있다. Unrolling factor를 계속 키우면 결국 루프는 없어지고 160개의 문장으로 펼쳐질 것이다. 이때 루프가 'unwound 되었다'라고 한다.

unrolling은 한 번의 루프 반복에 대해 더 많은 명령어가 실행되므로 코드에서의 분기를 감소시키고 명령어를 좀더 그룹화해서 효과적인 명령어 pipelining을 할 수 있도록 한다. 이와 같은 변환을 적용해볼 수 있는 최적의 후보는 제한된 제어 흐름을 가지는 innermost 루프이다. 최상의 최적화 결과를 얻기 위해 종종 data prefetching과 같이 사용된다.

3.4.6. Software Pipelining

명령어들이 루프내의 다른 반복에 걸쳐 실행되도록 하는 명령어 scheduling 기법이다. 루프 반복이 겹쳐지게 돼 이전 반복이 끝나기 전에 다른 반복 실행이 시작되도록 한다.
Long-latency 연산의 영향을 감소시켜 루프 실행을 빠르게 한다. 캐시 미스 영향을 감소시키기 위해 data prefetching을 할 수 있으며, 종종 루프 unrolling과 같이 사용된다.
소프트웨어 파이프라이닝은 슈퍼스칼라 시스템이나 VLIW 시스템에서 명령어 수준 병렬성을 이용하도록 한다.

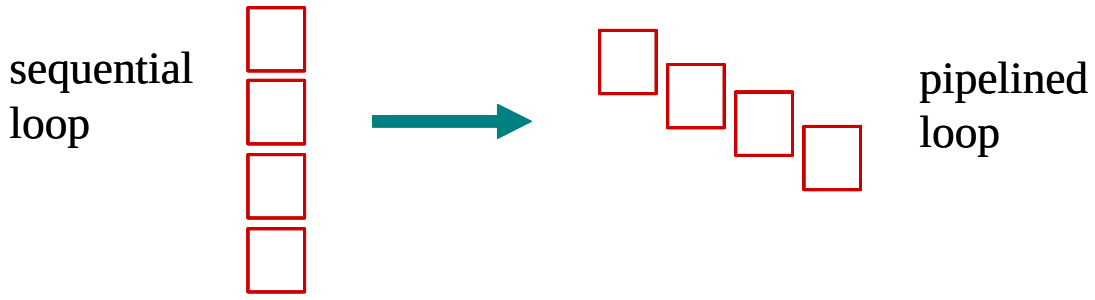


그림 3-3. 소프트웨어 파이프라이닝

3.4.7. Loop Fusion and Splitting

동일한 제어 조건을 가지는 두 개의 루프는 루프 fusion을 통해 합쳐질 수 있다. 이때 루프 카운터의 조건이 모든 면에서 일치해야 한다. 두 개의 루프를 하나로 합치게 되면 루프 overhead를 반으로 줄일 수 있다.

루프 내에 100개의 실행 문장이 있는데 그들 중 하나가 전체 루프가 최적화 되는 것을 방해하고 있는 경우를 가정해 보자. 아래 코드와 같이 데이터 의존성을 가지는 문장이 루프 내에 포함되는 경우를 생각해볼 수 있다.

```
DO I = 1, 1000
    ...
    (100 lines of optimizable code)
    ...
    A(i) = b(i) * A(i-1)   ! data dependant line
ENDDO
```

이와 같은 경우 루프를 최적화 가능한 부분과 최적화할 수 없는 부분으로 분리(splitting)해 최적화 가능한 부분에 대해서 만큼은 컴파일러가 최적화를 수행할 수 있도록 한다.

3.4.8. Loop nest linearization

Loop nest는 다음과 같이 다차원 배열의 개별 원소에 대한 계산 수행을 위해 주로 사용된다.

```
DO I =1, IMAX
    DO J = 1, JMAX
        DO K = 1, KMAX
            ...
            USE/CHANGE A(I,J,K)
            ...
```

```
        ENDDO
    ENDDO
ENDDO
```

다차원 배열을 사용하는 이유는 자료구조가 프로그램이 하는 작업과 일치해 해결하려고 하는 문제의 시각화에 적합하기 때문이다. 영상처리를 하는 프로그램은 기본적으로 영상 데이터를 2차원 배열로 저장한다. 편미분 방정식을 수치적으로 풀거나 물체의 특성을 모의 실험할 때 데이터를 3차원 배열에 저장하는 것이 일반적이다. 그러나, 성능 측면에서 3차원 데이터는 결점을 가진다. 3단계의 nested 루프는 단일 루프보다 3배의 루프 오버헤드를 가진다. 최적화는 세 개의 루프 중 하나에 대해서만 시도되는 경우가 많으며, 루프 순서가 메모리 접근 stride를 1이 아니게 해서 불연속 메모리 접근이 발생하기도 한다. 연속 메모리 접근이 불연속 접근(메모리 jump)보다 일반적으로 더 빠르며 데이터 캐시 사용 측면에서 캐시 실패도 적게 발생한다. 메모리에서의 데이터 load는 데이터 캐시에서의 로드보다 약 10배정도 더 느리다.
다중 배열을 1차원 배열로 저장하면 루프 오버헤드도 줄고 메모리 접근이 연속적으로 되게 할 수 있다. 다차원 배열의 선형화(linearization)로 코드 성능을 좋게 하는 것이 가능하다.
프로그램 시작부터 3차원 데이터를 1차원 배열로 선언해 사용한다면, 프로그램 작성이 어려워지고 읽기 어려운 코드가 될 수 있다. 메인 프로그램에서는 데이터를 3차원 배열로 선언해 사용하고 이 데이터를 선형화해 계산을 수행하는 서브 프로그램을 호출해 사용하는 것이 효과적인 방법이 될 수 있다. 이것은 3차원 배열을 실제 인수로 넘겨주고 그것을 받는 dummy를 1차원으로 선언해 사용하는 방법으로, 모든 작업이 수행되는 서브 프로그램에서 사용되는 데이터를 선형화해서 속도 향상 효과를 얻을 수 있다.
이와 같은 다중 차원 배열의 선형화를 컴파일러가 수행할 수 있다. 최근의 컴파일러들은 아래와 같이 루프 자체가 상대적으로 간단한 경우 nest된 루프를 자동으로 선형화 하는 기능이 있다.

```
DO K = 1, KMAX
    DO J = 1, JMAX
        DO I = 1, IMAX
            VELOCITY(I,J,K) = 0.0
        ENDDO
    ENDDO
ENDDO
```

Fortran90의 array syntax는 같은 내용을 루프를 써서 표현해야 하는 Fortran77 보다 편리하다. 또한 일부 컴파일러에서는 수행되는 배열 계산을 자동으로 선형화해서 처리하기 때문에 성능 측면에서도 우수한 결과를 보인다.

3.4.9. Optimizing Specific Loops

프로그래머는 사용하는 컴파일러의 지시어를 이용해 컴파일러에게 특별히 최적화를 집중해야 하는 루프를 알려 줄 수 있다. 또한 제한된 수준에서 루프에 대한 자동 병렬화를 수행할 수도 있다.

3.5. Numerical Optimizations

컴파일러 옵션을 통해 코드에 대한 numerical 최적화를 수행할 수 있다.

- Precision
- MADD Instruction
- Data Memory Alignment
- Restricted Pointers

3.5.1. Precision

수치적 최적화의 한 방법으로 프로그램에서 데이터가 요구하는 정밀도를 설정하는 것이다. 마이크로 프로세서 칩은 32비트 또는 64비트 크기의 워드를 가진다. 32비트 칩에서 부동소수 데이터의 정밀도는 십진수 7자리 정도, 64비트 칩에서는 13자리 정도이다. 칩 레지스터 크기, 데이터 캐시 라인 길이, 내부 버스 등 모든 하드웨어는 한 워드 크기의 변수들에 대한 계산을 수행하도록 설계된다. 이것은 기본적인 칩 연산들이 기본 워드 크기의 데이터에 대해 가장 빠르게 실행됨을 의미한다.

일부 컴파일러에서는 30자리 정밀도를 가지는 128비트 크기의 변수도 지원한다. 128비트 변수를 사용할 때의 문제점은 기본 연산이 자동적으로 하드웨어에서 처리되는 것이 아니고 소프트웨어적으로 처리된다는 것이다. 이로 인해 속도 저하(약 50%)가 발생하므로, 특별한 이유 없이 고 정밀도의 데이터를 사용하지 않은 것이 좋다.

3.5.2. The MADD Instruction

MADD 명령어는 덧셈과 곱셈을 한꺼번에 처리하는 명령어이다. 대부분의 컴파일러는 이와 같은 기능을 하는 어셈블리 명령어를 가지고 있다. 하나의 명령어가 두 연산을 따로 처리하는 것 보다 더 빠르게 연산을 수행한다. 1차원 배열의 내적, 행렬 곱셈, simultaneous 선형 방정식 집합 계산 등과 같은 선형 대수 계산에 매우 유용하게 사용된다.

컴파일러는 기본적으로 IEEE 연산 rule을 엄밀하게 따르게 되는데 MADD 명령어는 IEEE 정식 rule은 아니다. 따라서 MADD 명령어 사용을 위해서는 IEEE rule에 대한 relaxation을 허용하는 컴파일 옵션 사용이 필요할 경우도 있다.

3.5.3. Data Memory Alignment

가장 효과적인 최적화는 필요한 데이터가 캐시에 있도록 하고 캐시라인이 완전히 사용될 수 있도록 하는 것이다. 대부분 시스템에서 데이터 캐시는 4~8 워드 정도의 길이를 가지는 여러 개의 캐

시라인으로 구성된다. 8워드 길이의 데이터 캐시라인 크기를 가지는 프로세서에서 첫 번째 배열 데이터 하나에 접근하게 되면 메모리로부터 8개의 배열 원소가 데이터 캐시에 올라오게 된다. 그렇지만, 이것은 캐시에 올라가는 4바이트(또는 8바이트) 데이터 하나가 1워드에 저장된다고 가정한 경우이다. 실제로 이것이 항상 보장되는 것은 아니다. 대부분의 컴파일러에는 데이터 정렬이 워드 단위로 이루어지도록 강제하는 옵션이 있다.

3.5.4. Restricted Pointers

C/C++ 코드에서 일반적으로 포인터에 의해 참조되는 메모리는 컴파일 단계에서 알려지지 않으므로 컴파일러의 최적화 수행에 제한을 준다. 이때 restricted pointer 설정으로 컴파일러가 포인터 참조를 하는 루프에 대해서도 최적화할 수 있도록 할 수 있다. 이것은 옵션을 통해 컴파일러에게 두 개 이상의 포인터가 동일한 메모리 주소를 가리키지 않도록 하는 것이다. 보통은 –noalias 또는 –restricted 와 같은 컴파일러 옵션이 사용된다. 코드 내에서 restricted 속성으로 포인터를 선언하는 것도 가능하다.

4. 순차코드 최적화

최적의 성능을 얻기 위해 코드내의 루프 또는 시간이 많이 걸리는 부분에 대한 직접적인 핸드튜닝이 가능하다. 그렇지만, 이와 같은 핸드튜닝 이전에 컴파일러 최적화 옵션을 사용하거나 직접 작성한 루틴을 최적화된 라이브러리 호출로 대체하는 것 등의 방법을 우선적으로 사용하는 것이 효과적이다. 핸드튜닝에 의한 직접적인 성능향상도 물론 중요 하지만 컴파일러가 진행하는 최적화가 보다 효과적으로 진행될 수 있도록 코드 수정을 하는 것도 핸드튜닝의 중요한 목표가 된다.

4.1. 최적화 가이드라인

최적의 성능을 얻기 위해 코드 작성 단계에서 프로그래머가 기본적으로 주의해야 할 점들을 이곳에서 미리 정리해 둔다. 이곳에서 정리된 내용들에 대해 뒤에서 모두 다루게 될 것이다.

1) 컴파일러는 한 프로그램 단위에서 제한적으로 사용되는 지역변수의 사용에 대해 정밀 분석이 가능하다. 여러 프로그램 단위에 걸쳐 사용되는 전역변수는 컴파일러에 의한 최적화 수행을 어렵게 하므로 자동변수와 같은 지역변수를 많이 사용하는 것이 좋다.
2) 컴파일 과정에서 포인터를 쫓는 것은 어렵거나 불가능 하기 때문에 포인터 변수는 대부분의 메모리 최적화를 방해한다. 가급적 포인터를 사용하지 않는 것이 최적화에 도움이 된다.
3) 컴파일러는 동일 수식은 잘 인지하지만, 교환 법칙을 이해하지는 못한다. 동일한 수식을 표현할 때의 변수들은 동일 순서로 배열하는 것이 컴파일러에 의한 최적화 수행에 도움이 된다.
4) 데이터 타입을 섞어 쓰지 않는다. 혼합된 표현으로 인해 코드 실행 중 (예로 정수와 실수 사이의) 타입 변환이 강제로 일어나 성능이 나빠진다. 또한 루프 변수로 실수 표현을 쓰지 않는 것이 좋다.
5) 가능한 최소의 부동소수 정밀도를 사용한다. 단정도 실수 연산은 배정도 실수 연산보다 더 빠르다. IBM POWER에서는 모든 부동소수 계산을 IEEE 배정도 모드로 수행한다. 단정도 실수 연산도 배정도 실수로 변환해 연산을 수행하고 그 결과를 단정도로 round off 한다. 단정도 실수를 배정도 실수로 변환하는 과정에서 성능 감소는 없다. 배정도 실수를 단정도 실수로 round off하는 과정에는 성능 감소가 있으나, 그 정도가 작아서 IBM POWER 시스템에서는 단정도 실수 연산과 배정도 실수 연산의 성능 차이는 매우 작다.
6) 작은 함수를 많이 사용하는 것을 피한다.
7) 최적화된 루프 실행을 위해 주의할 점들
   - 루프의 크기를 관리하기 쉽게 유지한다.
   - 데이터 접근을 순차적으로 한다.
   - 루프 안의 IF 구문을 최소화 한다.
   - 루프 안에서 서브루틴/함수 호출을 피한다.
   - 배열과 루프 인덱스 표현을 가급적 단순화 시킨다.

- 루프 인덱스 변수는 정수형을 사용한다.
- 루프 안에서 GOTO, STOP, PAUSE, RETURN 등과 같은 흐름 관련 구문의 사용을 피한다.
- 루프 안에서 최적화를 방해하는 EQUIVALENCE 구문의 사용을 피한다.
- 루프 안에서 LOGICAL*1, BYTE, INTEGER*1, INTEGER*2, REAL*16, COMPLEX*32 CHARACTER, INTEGER*8 등과 같은 최적화 불가능한 데이터 형의 사용을 피한다.
- I/O 구문의 사용을 자제한다.

8) 루프 최적화의 예

- IF 구문의 제거

최적화 전	최적화 후

DO I=1,N
IF(D(J).LE.0.0)X(I)=0.0
A(I)=B(I)+ C(I)*D(I)
E(I)=X(I)+ F*G(I)
ENDDO
	IF(D(J).LE.0.0)THEN
	DO I=1,N
	A(I)=B(I)+ C(I)*D(I)
	X(I)=0.0
	E(I)=F*G(I)
	ENDDO
	ELSE
	DO I=1,N
	A(I)=B(I)+ C(I)*D(I)
	E(I)=X(I)+ F*G(I)
	ENDDO
	ENDIF

- 경계조건 IF 테스트

최적화 전	최적화 후

DO I=1,N
IF(I.EQ.1)THEN
X(I)=0.0
ELSEIF(I.EQ.N)THEN
X(I)=1.0
ENDIF
A(I)=B(I)+ C(I)*D(I)
E(I)=X(I)+ F*G(I)
ENDDO
	A(1)=B(1)+ C(1)*D(1)
	X(1)=0.0
	E(1)=F*G(1)
	DO I=2,N-1
	A(I)=B(I)+ C(I)*D(I)
	E(I)=X(I)+ F*G(I)
	ENDDO
	X(N)=1.0
	A(N)=B(N)+ C(N)*D(N)
	E(N)=1.0+ F*G(N)

- 반복적인 고유함수의 계산

최적화 전	최적화 후

DO I=1,N
  DO J=1,N
  A(J,I)=B(J,I)*SIN(X(J))
  ENDDO
ENDDO
	DIMENSION SINX(N)
	…
	DO J=1,N
	  SINX(J)=SIN(X(J))
	ENDDO
	DO I=1,N
	  DO J=1,N
	  A(J,I)=B(J,I)*SINX(J)
	  ENDDO
	ENDDO

- 나눗셈 계산을 곱셈으로 치환

최적화 전	최적화 후

DO I=1,N
  A(I)=B(I)/C(I)
  P(I)=Q(I)/C(I)
ENDDO
	DO I=1,N
	  OC=1.0/C(I)
	  A(I)=B(I)*OC
	  P(I)=Q(I)*OC
	ENDDO

DO I=1,N
  A(I)=B(I)/C(I)
  P(I)=Q(I)/D(I)
ENDDO
	DO I=1,N
	  OCD=1.0/(C(I)*D(I))
	  A(I)=B(I)*D(I)*OCD
	  P(I)=Q(I)*C(I)*OCD
	ENDDO

- 2의 거듭제곱의 배수를 갖는 배열 문제

최적화 전	최적화 후

```
INTEGER NX,NZ
PARAMETER (NX=2048,NZ=2048)
REAL P(2,NX,NZ)
...
DO IX=2,NX-1
   DO IZ=2,NZ-1
      P(IT1,IX,IZ)=   -P(IT1,IX,   IZ)
&
                  + S*P(IT2,IX-1  ,IZ)
&
                  + S*P(IT2,IX+ 1,IZ)
&
                  + S*P(IT2,IX,  IZ-1)
&
                  + S*P(IT2,IX, IZ+ 1)
   END DO
END DO
```

```
INTEGER NX,NZ
PARAMETER (NX=2048,NZ=2048)
REAL P(2,2080,NZ)
...
DO IX=2,NX-1
   DO IZ=2,NZ-1
     P(IT1,IX,IZ)=       -P(IT1,IX,IZ)
&
                  + S*P(IT2,IX-1,IZ)
&
                  + S*P(IT2,IX+ 1,IZ)
&
                  + S*P(IT2,IX,IZ-1)
&
                  + S*P(IT2,IX,IZ+ 1)
   END DO
END DO
```

9) Fortran 코드의 최적화

- FORTRAN에서 EQUIVQLENCE 문의 사용을 자제한다. EQUIVALENCE문은 Unaligned data를 강제하거나 data가 natural boundaries span 되도록 하여 최적화를 방해한다. 제어 변수가 EQUIVALENCE 문에 포함되면 Global data analysis, 또는 implied-DO 루프 collapsing 등을 방해.
- Fortran 90에서 common block보다 모듈의 사용을 권장한다. 불필요한 포인터의 사용을 피하라. 컴파일러는 최적화 중에 포인터를 추적하기 어려움. Dead store 제거와 store motion과 같은 메모리 최적화를 방해한다.
- 루틴의 직접 작성보다는 Fortran 90에서 지원하는 배열 속성 프로시저 사용을 권장한다.
- 명시적인 루프 사용보다는 Fortran 90의 array syntax 사용을 권장한다.
- Intent 속성의 사용을 권장하고, allocatable, pointer 등과 같은 동적배열 사용을 자제한다.

10) C 코드의 최적화

- String 연산이 있는 경우 string.h를 include해 사용한다.
- Math 라이브러리를 사용하는 경우 math.h를 include해 사용한다.
- Struct가 double을 포함하는 경우 첫 부분에 선언한다.
- Globals, parameter, 함수에 대해 const 선언을 사용한다.

4.2. 클러터 제거

클러터(clutter)는 결과에 기여하는 바 없이 실행시간을 소비하는 부분이다. 지나친 모듈화나 과잉 코드 또는 최적화를 위해 수정한 부분이 오히려 코드의 클러터를 증가시키고 최적화를 수행하는 컴파일러에게 혼동을 줄 수 있다. 최적화된 코드 작성을 위해 클러터를 가급적 줄여야 하는데 이를 위해 여러 가지 클러터에 대해 알아본다.
클러터는 크게 부하관련 클러터와 컴파일러 유연성을 방해하는 클러터의 두 가지로 나눌 수 있다.

부하관련 클러터: 프로시저 호출, 간접메모리 참조, 루프에 포함된 판단 구문, 형 변환, 불필요한 변수 예약 등
컴파일러 유연성을 방해하는 클러터: 프로시저 호출, 루프에 포함된 판단 구문, 불분명한 포인터

프로시저 호출이나, 루프에 포함된 판단 구문 등은 두 가지 클러터 모두에 포함되는데, 이러한 클러터 들은 이전의 명령어와 이후에 나타나는 명령어가 안전하게 혼합될 수 없도록 하는 울타리를 프로그램 내에 만들어 코드 성능에 나쁜 영향을 주게 된다.

4.2.1. 프로시저 호출

프로시저 호출에서 모듈에 대한 enter와 exit가 상대적으로 짧은 시간에 한번만 이뤄진다면 saving registers와 preparing argument lists의 부하가 그다지 크지 않다. 소수의 작은 서브루틴을 계속해서 호출하는 경우라면 부하가 매우 커질 수 있다. 이때는 루틴을 호출하지 않고 그 자리에서 바로 일을 수행하는 것이 더 나을 수 있다.
서브루틴 호출은 컴파일러 유연성을 방해한다. 컴파일러는 서로 독립적인 명령어들을 자유롭게 intermix함으로서 파이프라인 처리 등을 통해 코드 성능을 좋게 할 수 있다. 그러나 컴파일러가 서브루틴과 함수를 뚫어 볼 수 없으므로 이런 기회는 사라지게 된다. 똑 같은 계산을 수행하는 다음 두 코드를 비교하자.

코드 1.

```
DO I = 1, N
    A(I) = A(I) +  B(I)*C
ENDDO
```

코드 2.

```
DO I = 1, N
    CALL MADD(A(I),B(I),C)
ENDDO
SUBROUTINE MADD (A, B, C)
```

```
A=A+ B*C
END
```

좀 과장된 경우이긴 하지만 코드 2에서는 작은 계산을 하는 서브루틴이 계속적으로 호출되고 있다. 특히 위의 계산은 부동소수 연산이어서, 병렬성 손실과 프로시저 호출 부하로 인한 코드 성능 저하가 상대적으로 더 커지게 된다. 파이프라인 처리되는 연산은 1 CPI에 도달하기 전에 wind-up 시간이 존재하게 되는데, 서브루틴 호출 사이의 계산이 적을수록 이러한 wind-up, wind-down에 소비되는 시간이 성능에 많은 영향을 주게 된다.
서브루틴과 함수 호출은 컴파일러가 COMMON, external 변수를 효과적으로 처리하는 것을 복잡하게 만든다. 컴파일러는 많은 변수들의 계산 값을 저장하기 위해 레지스터를 사용한다. 프로시저가 호출될 때 컴파일러는 그 프로시저가 COMMON 또는 external 변수를 바꿀 것인지 아닌지를 알 수 없다. 그래서 수정된 COMMON, external 변수들이 모두 메모리에 다시 저장되도록 강제한다. 호출된 루틴이 수정된 값을 찾을 수 있도록 하기 위함이다. 호출에서 리턴될 때도 같은 변수들이 레지스터로 다시 load 되어야 한다. 컴파일러가 더 이상 이전의 레지스터에 남아있는 값을 신뢰할 수 없기 때문이다. 이러한 변수 restoring과 saving 시간을 무시할 수 없다. 다음처럼 로컬 변수가 external 이나 COMMON으로 선언된 경우도 문제가 된다.

```
COMMON /USELESS/ K
DO K = 1, 1000
  IF (K == 1) CALL AUX
ENDDO
```

K는 로컬 변수의 역할만 하지만 COMMON으로 선언되어 AUX에 대한 호출 때 마다 컴파일러는 K를 저장하고 reload하게 된다. 프로시저 호출의 비효율성을 이야기 했으나 모듈화된 프로그램의 작성은 여전히 유효하다. 실질적으로 maintainability와 modularity는 small performance 개선보다 더 중요하다. 큰 작업에서의 모듈화는 필요하나 작은 작업의 반복적인 호출은 지양하는 것이 좋다.

4.2.1.1. 매크로

매크로는 컴파일과정에서 인라인되는 작은 프로시저이다. FORTRAN에서는 statement function이지만 90에서는 없어졌다.

```
#define average(x, y) ((x+ y)/2)
main()
{
    float q=100, p=50;
```

```
        float a;
        a = average(p,q);
        printf("%f₩n",a);
    }
```

C 프로그램의 첫 컴파일 단계는 preprocessor, cpp를 거치는 것이다. cpp는 #define 문을 인라인 시킨다. 매크로 정의와 일치하는 패턴을 대체하는데 위의 코드에서는 a = average(p,q)가 a = ((p+ q)/2)로 대체된다.

cpp는 단지 패턴을 글자 그대로 대체하므로 매크로를 정의할 때는 주의가 필요하다. #define multiply(a,b) (a*b)와 같이 정의하고 c = multiply(x+ t, y+ v)와 같이 호출하였다면, c = x+ t*y+ v의 결과가 될 수 있으므로 주의해야 한다.

많은 C 헤더파일(.h)이 매크로 정의를 포함하고 있다. 실제 일부 표준 C라이브러리 함수들은 헤더파일 내에서 매크로로 정의된다. 예로 getchar 함수는 헤더파일 <stdio.h>에서 매크로로 정의돼 있다.

FORTRAN 프로그램을 위한 cpp 매크로 작성이 가능하다.

```
#define AVERAG(X,Y) ((X+ Y)/2)
        PROGRAM MAIN
        REAL R, P, Q
        DATA P, Q /50., 100./
        A = AVERAG(P,Q)
        WRITE (*,*) A
        END
```

위의 프로그램은 cpp를 통해 전처리 되어야 한다. 두 단계의 컴파일 과정을 거쳐야 하는데cpp 지시어를 포함하는 파일을 .F로 저장하고 전처리한 결과를 .f파일로 저장할 수 있다.

```
    %/lib/cpp/ -P < average.F > average.f
```

생성된 average.f 파일은 다음과 같다.

```
        PROGRAM MAIN
        REAL R, P, Q
        DATA P, Q /50., 100./
        A = ((P+ Q)/2)
        WRITE (*,*) A
        END
```

만약 컴파일러가 .F 확장자를 인식한다면 두 단계를 거칠 필요가 없다. 컴파일러는 .F 파일에서 자동으로 cpp를 호출하고 처리한 결과를 .f(중간단계) 파일로 보낸다.

4.2.1.2. 프로시저 인라이닝

프로시저 호출 부하를 제거하고 병렬성을 증가시키기 위해 길이가 비교적 짧은 코드 부분은 프로시저 호출보다 인라이닝이 성능증가에 효과적이다. 컴파일러가 프로시저를 호출하는 모듈 속으로 해당 프로시저를 인라이닝할 수 있다면 프로그래머는 프로시저에 의한 모듈화된 프로그램을 작성하더라도 프로시저 호출에 의한 성능감소는 경험하지 않을 수 있을 것이다.
컴파일러에 의해 다음과 같은 인라이닝 관련 기능이 제공될 수 있다.

- 컴파일러 옵션을 통한 인라인 수행 프로시저 지정
- 소스 코드에 인라인 관련 지시어 삽입
- 자동 인라이닝 실행

지시어나 인라이닝 컴파일러 옵션 등은 표준이 아니어서 지원여부나 사용법 등에 대해 해당 컴파일러 설명서를 참조해야 한다. 자동 인라이닝 기능은 소수의 컴파일러에 의해 지원된다. 모든 프로시저 호출에 대한 인라이닝은 실행코드의 크기를 증가시키고 이로 인해 많은 명령어 캐시 실패를 가져와 오히려 코드 성능을 나쁘게 할 수 있다. 프로파일링을 통해 인라인될 프로시저를 선택할 수 있으며, 최적의 후보로는 크기가 작으면서 자주 호출되는 프로시저라고 할 수 있다.

4.2.2. 루프에 포함된 분기

프로그램에서 if 구문에 의해 수행되는 판단 과정은 나노(nano)초 수준의 시간이 소요되지만 루프에서와 같이 자주 실행되는 코드 내에서 처리돼야 하는 분기 명령은 코드 성능을 나쁘게 할 수 있다.
코드에서 분기에 의한 영향을 감소시키기 위한 두 가지 처리 방법이 있다.

- 분기를 streamline화
- 루프 밖으로 이동

분기에 의한 성능감소를 막기 위해 루프내의 어떤 분기 들이 재구성 가능한지 불가능한지 알아야 할 것이다. 루프내의 분기들은 다음과 같이 몇 가지 영역으로 구분할 수 있다.

- Loop invariant conditionals
- Loop index dependent conditionals
- Independent loop conditionals
- Dependent loop conditionals
- Reductions
- Conditionals that transfer control

4.2.2.1. Loop Invariant Conditionals

루프내의 계산과 무관한 conditional이다. invariant는 결과가 항상 같다는 것을 의미한다. 다음 코드에서 N의 값은 A, B, C, I의 값과 무관하다.

클러터 존재

```
DO I = 1, K
   IF(N .EQ. 0) THEN
      A(I) = A(I) +  B(I)*C
   ELSE
      A(I) = 0.
   ENDIF
ENDDO
```

클러터 제거

```
IF(N .EQ. 0) THEN
   DO I = 1, K
      A(I) = A(I) +  B(I)*C
   ENDDO
ELSE
   DO I = 1, K
      A(I) = 0.
   ENDDO
ENDIF
```

클러터가 제거된 코드는 K-1회의 테스트가 감소했으며, if구문에 의한 분기 가능성이 사라진 루프 계산은 컴파일러에 의해 파이프라인 처리가 가능해져 많은 성능향상 효과를 볼 수 있다.

4.2.2.2. Loop Index Dependent Conditionals

항상 참 또는 항상 거짓이 아니라 루프 인덱스 값에 따라 판단 결과가 달라지는 경우에 해당된다.

클러터 존재

```
DO I = 1, N
   DO J = 1, N
      IF(J .LT. I) THEN
         A(J,I) = A(J,I) +  B(J,I)*C
      ELSE
```

```
            A(J,I) = 0.
         ENDIF
      ENDDO
   ENDDO
```

클러터 제거

```
   DO I = 1, N
      DO J = 1, I-1
         A(J,I) = A(J,I) +  B(J,I)*C
      ENDDO
      DO J = I, N
         A(J,I) = 0.
      ENDDO
   ENDDO
```

클러터가 제거된 코드는 N이 충분히 크다면 이전의 코드보다 향상된 성능을 보여 준다. N 값이 작다면, 오히려 클러터가 증가된다고 볼 수도 있지만, 이런 경우엔 코드 실행시간 자체가 매우 작아지므로 클러터에 의한 성능 효과는 매우 미미해 진다.

4.2.2.3. Independent Loop Conditionals

조건이 루프 인덱스에 직접적으로 의존하지 않는 경우이다.

```
   DO I = 1, N
      DO J = 1, N
         IF(B(J,I) .LT. 1.0)   A(J,I) = A(J,I) +  B(J,I)*C
      ENDDO
   ENDDO
```

계산 분리가 불가능하지만, 반복 계산이 서로 독립적이므로 unrolling이나 병렬화에 의한 계산 수행이 가능하다.

4.2.2.4. Dependent Loop Conditionals

조건이 루프에서 계산되는 값에 의존하는 경우이다. 계산이 완료돼야 어떤 반복 계산으로 코드가 진행될지 결정할 수 있다.

```
   DO I = 1, N
```

```
    IF(X .LT. B(I))   X = X +  A(I)*C
ENDDO
```

이와 같은 경우는 의존성으로 인해 최적화 수행이 일반적으로 어렵다.

4.2.2.5. Reductions

배열로부터 하나의 스칼라 값을 얻게 되는 종류의 계산을 reduction이라 부른다. if 구문에 의해 배열의 원소 중 최대값이나 최소값을 결정하는 것을 생각해 볼 수 있다. reduction 연산이 수행되는 루프를 다음과 같이 한 번의 반복에서 여러 개를 테스트하도록 재구성해서 병렬성을 증가 시키고 코드 성능을 증가시킬 수 있다.

클러터 존재

```
for (i=0; i<n; i++)
    z = a[i] > z ? a[i] : z;
```

클러터 제거

```
z0 = 0.;
z1 = 0.;
for (i=0; i<n-1; i+=2){
    z0 = z0 < a[i] ? a[i]      : z0;
    z1 = z1 < a[i] ? a[i+1] : z1;
}
z = z0 < z1 ? z1 : z0;
```

일반적으로 이와 같은 루프 재구성은 컴파일러의 유연성에 제한을 줄 수 있어 핸드 튜닝의 방법으로 부적절하다. 병렬 시스템 환경에서는 컴파일러에 의해 수행되는 reduction 연산의 수행을 권장한다.

4.2.2.6. Conditionals that Transfer Control

조건문은 테스트 결과에 의해 새로은 할당을 수행하는 경우가 많지만 모든 조건이 할당으로 끝나는 것은 아니다. 테스트 결과로 서브루틴 호출이나 goto 문에 의한 제어의 흐름을 생각할 수 있다.

```
DO I = 1, N
    DO J = 1, N
        IF(B(J,I) .EQ. 0) THEN
            PRINT*, I, J
```

```
            STOP
          ENDIF
          A(J,I) = A(J,I) / B(J,I)
        ENDDO
      ENDDO
```

위의 코드는 반복 계산이 순차적으로 진행되어야 하므로 성능에 나쁜 영향을 미치게 된다.

4.2.3. 기타 클러터

4.2.3.1. 형 변환

실행 중 형 변환을 해야 하는 문장은 실행될 때마다 성능 손실이 있게 된다.

```
INTEGER N, I
PARAMETER (N=10000)
REAL*8 A(N)
REAL*4 B(N)
DO I = 1, N
   A(I) = A(I) +  B(I)
ENDDO
```

위의 코드에서 B(I)는 계산 수행 전에 배정도 타입으로 형 변환이 일어나 성능 손실을 야기한다.
C 코드의 경우 모든 고유함수의 부동소수연산은 배정도로 처리된다.
대부분의 시스템에서 문자 연산은 프로시저 호출에 의해 처리되므로 정수 연산보다 더 느리다. 컴파일러는 문자형 변수에 대해 최적화 시도를 하지 않는다. 다음은 문자 연산이 IF 구문에 포함된 클러터를 가지는 코드이다.

클러터 존재

```
DO I = 1, 10000
   IF(CHVAR(I) .EQ. 'Y') THEN
      A(I) = A(I) +  B(I)*C
   ENDIF
ENDDO
```

문자 연산을 정수형 변수에 의한 연산으로 수정해 성능 손실을 막을 수 있다.

클러터 제거 1.

```
DO I = 1, 10000
   IF(IFLAG(I) .EQ. 1) THEN
      A(I) = A(I) + B(I)*C
   ENDIF
ENDDO
```

클러터 제거 2.

```
DO I = 1, 10000
   A(I) = A(I) + B(I)*C*IFLAG(I)
ENDDO
```

위 코드에서 IFLAG(I)의 값은 1 또는 0 이다.

4.2.3.2. Common Sub-expression 제거

컴파일러는 반복되는 코드 패턴을 찾아내 임시 변수로 대체해 처리한다.

```
c = a + b+ d
e = q + a + b
```

위에서 반복되는 a+b 부분은 아래와 같이 임시변수 temp로 대체해 처리함으로써 반복되는 계산에서 연산 수행을 줄일 수 있다.

```
temp = a + b
c = temp + d
e = q + temp
```

컴파일러는 a+b와 b+a를 같은 것으로 판단하지 못한다. 성능에 중요한 영향을 미치는 부분에서 컴파일러가 처리하지 못하도록 복잡하게 표현된 common subexpression 부분들은 프로그래머가 직접 제거할 필요가 있다.
다음은 common subexpression에 프로시저 호출이 포함된 경우이다.

클러터 존재.

```
x = r*sin(a)*cos(b)
y = r*sin(a)*sin(b)
z = r*cos(a)
```

클러터 제거.

```
temp = r*sin(a)
x = temp*cos(b)
y = temp*sin(b)
z = r*cos(a)
```

컴파일러는 프로시저 호출이 포함된 common subexpression을 임시변수로 대체하지 않는다. 컴파일러가 볼 수 없는 변수나 인수들의 상태가 변하지 않는다는 것을 확인할 수 없기 때문이다. 그러나 고유함수가 common subexpression에 포함된 경우는 이를 인식해 임시변수로 대체한다.

4.2.3.3. Code Motion

루프 반복에 불필요한 부분이나 반복 동안 변하지 않는 부분을 루프 밖으로 이동하는 것이다. 루프 위로의 이동을 hoisting, 아래로의 이동을 sinking 이라고 한다.

클러터 존재.

```
DO I = 1, N
   A(I) = A(I) / SQRT(x*x +  y*y)
ENDDO
```

클러터 제거.

```
temp = 1 / SQRT(x*x +  y*y)
DO I = 1, N
   A(I) = A(I) * temp
ENDDO
```

클러터 존재.

```
while (*p != ' ')
   c = *p++;
```

클러터 제거.

```
while (*p++ != ' ');
   c = *(p-1);
```

4.2.3.4. 루프에 포함된 배열 원소의 처리

루프에서 배열 원소의 사용이 반복되는 경우 해당 원소를 메모리에서 한 번만 불러오도록 해서 성능 손실을 줄일 수 있다. 다음 루프에서 A(I)는 한 번의 반복 동안 두 번 사용된다.

클러터 존재.

```
DO I = 1, N
   AOLD(I) = A(I)
   A(I) = A(I) + AINC(I)
ENDDO
```

A(I)를 두 번 사용하는 것은 주소 계산과 memory load 연산의 반복 수행을 의미한다.

클러터 제거.

```
DO I = 1, N
   TEMP = A(I)
   AOLD(I) = TEMP
   A(I) = TEMP + AINC(I)
ENDDO
```

Fortran 컴파일러가 이와 같은 최적화를 수행하게 되지만, 경우에 따라 임시 변수의 사용을 프로그래머가 직접 지정해야 하는 경우가 있다. 루프 내에서 프로시저를 호출하거나, 일부 변수들이 external 또는 COMMON으로 선언되었을 때가 그런 경우이다. 이때 임시변수와 다른 변수 사이의 형을 동일하게 해주는 것도 성능 손실을 막기 위해 중요하다.

```
doinc(int xold[], int x[], int xinc[], int n)
{
  for(i=0, i<n, i++) {
    xold[i] = x[i];
    x[i] = x[i] +  xinc[i];
  }
}
```

위의 C코드에서 컴파일러가 x, xinc, xold의 선언을 알 수 없다면, 컴파일러는 각 변수들을 동일 저장공간을 가리키는 포인터로 가정하고 load와 store를 반복하게 된다. 이때 x, xold, xinc 값을 임시 변수로 두고 최적화를 수행하는 것은 컴파일러에 의해 수행되지 못하는 최적화 이다.
루프내에 스칼라 임시변수를 두는 것은 RISC 슈퍼스칼라 시스템에서 유용하다. 병렬 컴파일러는 스칼라를 제거하거나 임시 벡터변수로 대체하려고 시도한다. 따라서 위와 같은 임시 스칼라 변수의 도입은 병렬 시스템에서는 오히려 성능손실을 가져올 수도 있다.

4.3. 루프 최적화

대부분의 과학 계산 코드에서 실행시간을 소비하는 부분은 루프이다. 앞에서 루프 내에서 성능을 저하시키는 클러터에 대해 알아 보았고, 여기서는 클러터가 없는 루프의 성능 향상 기법을 알아본다.

여기서 소개하는 내용들은 컴파일러가 자동으로 처리할 수도 있고 직접 코드를 손봐야 하는 경우도 있다. 성능을 위한 코드 수정은 항상 신중하게 해야 한다. 어떤 컴파일러에게는 성능 향상을 가져오는 코드 수정이 다른 컴파일러에서는 성능 저하를 가져올 수도 있으며, 다른 아키텍처에서도 성능을 방해하는 요인이 될 수 있다. 새로운 아키텍처에서 성능 테스트를 위해서는 코드 원본을 보관해 두는 것이 언제나 좋다. 만약, 코드 수정에 의한 성능 이득이 작다면 보다 간단한 원본을 쓰는 것이 좋을 것이다.

다양한 루프 최적화 기법들이 있다.

- Loop unrolling
- Nested loop optimization
- Loop interchange
- Memory reference optimization
- Blocking
- Out-of-core solutions

앞으로 이러한 루프 최적화가 컴파일러 수준에서 모두 자동화 될 수 있을 것이다. 대개 loop unrolling은 통상의 컴파일러 최적화 단계에서 수행되며, 다른 최적화 역시 명시적인 컴파일 옵션에 의해 수행될 수 있다. 프로그래머는 직접 수행하고자 하는 최적화 과정이 컴파일러에 의해 수행될 수 있는 최적화인지 알아보고 어느 쪽이 더 나은 성능을 보이는지 테스트해 볼 필요가 있다.

4.3.1. Operation Counting

루프를 새로 작성하거나 루프 순서를 재 구성하기 전에, 루프 몸체가 각 반복에서 무엇을 하는지 알아야 한다. Operation counting은 operation mix를 이해하기 위해 루프를 조사하는 과정이다.

루프최적화를 수행하고자 하는 프로그래머는 루프의 반복 마다 로드, 저장, 부동소수, 정수와 라이브러리 호출 회수를 알아야 한다. 이 회수로부터 그 루프의 operation mix가 얼마나 프로세서 능력과 일치하는지 알 수 있게 된다. 컴파일러가 operation counting을 통해서 루프의 효과적인 representation을 생성하도록 하는 것은 아니다. 그러나, operation counting은 프로그래머가 직접적인 최적화 수행을 하는데 있어 루프에 대한 좋은 정보를 제공한다.

한 시스템에서 명령어 mix가 균형적일지라도 다른 시스템에서는 불균형일 수 있음을 명심해야 한다. 요즘의 프로세서들은 클럭 사이클당 하나에서 많게는 네 개까지의 연산 조합을 실행할 수 있다. 주소 연산은 종종 메모리를 참조하는 명령어에 포함되는데 컴파일러가 복잡한 루프 주소 계산을 단순한 표현으로 대체할 수 있어서 operations counting에서 주소 연산은 무시할 수 있다

```
DO I = 1, N
    A(I,J,K) = A(I,J,K) + B(J,I,K)
ENDDO
```

위의 루프는 부동소수 연산 1회와 3회(2 loads, 1 store)의 메모리 참조를 포함한다. 복잡한 배열 인덱스 표현이 있지만 컴파일러에 의해 단순화 되어서 메모리, 부동소수 연산과 더불어 동일 사이클에서 실행된다. 각 루프의 반복에서 인덱스 변수를 증가시키고 루프가 완료 되었는가를 검사한다.
메모리 참조와 부동소수 연산의 비가 3대 1이라는 것은, 메모리 접근 경로가 하나 이상이 아니라면 루프에서는 최대 부동소수 연산 성능의 1/3 이상을 낼 수 없다는 것을 의미한다. 좋지 않은 소식 이지만 좋은 정보를 준다. 위의 ratio는 메모리 참조 최적화를 먼저 고려해야 한다는 점을 프로그래머에게 알려 준다.

```
DO I = 1, N
    A(I) = A(I) + B(J)
ENDDO
```

위의 루프는 1회의 부동소수 연산과 2회(load 1회, store 1회)의 메모리 참조를 포함한다. B(J)가 loop invariant 이므로 한 번만 load된다. 여기서도 부동소수 연산의 throughput은 제한된다. 메모리 참조 대 부동소수 연산의 비는 2 대 1이 된다.
다음 루프는 복소수 벡터의 곱을 수행하는 것이다. 결과는 앞의 벡터에 할당된다. 여기서는 6회의 메모리 연산(4 loads, 2 stores)과 6회의 부동소수 연산이 수행(2 addition, 4 multiplication)된다.

```
for (i=0; i<n; i++) {
    xr[i] = xr[i]*yr[i] - xi[i]*yi[i];
    xi[i] = xr[i]*yi[i] + xi[i]*yr[i];
}
```

위의 루프는 사이클당 부동소수 연산과 메모리 연산이 같은 회수만큼 수행될 수 있어 프로세서에 대해 균형 잡혀 보인다. 많은 프로세서들은 부동소수 곱셈과 덧셈을 단일 명령어(MADD or FMA)로 처리한다. 해당 연산을 컴파일러가 multiply-add가 적합하다고 인식하기에 충분하다고 했을 때, 이 역시도 메모리 참조에 의해 제한을 받게 된다. 위 코드의 각 반복에서는 2회의 곱셈과 2회의 multiply-add가 수행되도록 컴파일 된다.
Operation counting은 루프의 requirements가 시스템 능력에 얼마나 잘 부합되는지를 추정하는 간단한 방법이다. 많은 경우에 루프의 성능은 메모리 참조에 의해 좌우됨을 알 수 있다. 따라서, 메모리 참조에 대한 튜닝은 매우 중요하다.

4.3.2. Basic Loop Unrolling

루프 최적화의 가장 기초는 unrolling이다. 루프 최적화의 기본으로서 최근의 컴파일러는 이득이 있다고 보여지면 자동으로 unrolling을 수행한다. 따라서, 프로그래머가 코드에 대한 직접적인 loop unrolling을 수행하는 것을 권장하지 않는다.

unrolling으로 인한 성능 이득은 한 번의 반복에 더 많은 연산을 수행하도록 하기 때문이다. 각 반복을 마치고 인덱스 값은 증가, 테스트되고 제어는 다음 루프의 시작부분으로 분기돼 돌아 가는데, unrolling에 의해 루프 실행 당 loop-ends가 줄어든다. unrolling은 분기 수를 현저히 감소 시키며 분기와 분기 사이에 프로세서 명령어를 더 많이 둔다(이로 인해 기본 block의 크기가 증가).

```
DO I = 1, N
    A(I) = A(I) +  B(I)*C
ENDDO
```

위의 코드를 다음과 같이 unroll할 수 있다.

```
DO I = 1, N, 4
    A(I)     = A(I)     + B(I)   *C
    A(I+ 1)  = A(I+ 1) +  B(I+ 1)*C
    A(I+ 2)  = A(I+ 2) +  B(I+ 2)*C
    A(I+ 3)  = A(I+ 3) +  B(I+ 3)*C
ENDDO
```

적은 반복 회수(적은 루프 overhead)로 동일 연산이 수행되는 장점이 있고, 서로 독립적인 반복 연산이 함께 계산됨으로써 파이프라인 처리가 가능해 지고, 슈퍼스칼라 프로세서에서 네 개의 문장이 병렬로 실행되는 것이 가능하다.

위의 코드에서 루프는 4회 unroll 되었다. N이 4의 배수가 아니라면 1회, 2회, 또는 3회의 실행되지 않는 여분의 반복이 있을 수 있다. 이를 처리하기 위해 다음과 같이 여분의 루프(preconditioning loop)를 둔다.

```
II= IMOD(N,4)
DO I = 1, II
    A(I) = A(I) +  B(I)*C
ENDDO

DO I = 1+ II, N, 4
```

```
A(I)          = A(I)     + B(I)   *C
    A(I+ 1)   = A(I+ 1) +  B(I+ 1)*C
    A(I+ 2)   = A(I+ 2) +  B(I+ 2)*C
    A(I+ 3)   = A(I+ 3) +  B(I+ 3)*C
ENDDO
```

4.3.3. Unrolling이 부적합한 루프

모든 루프를 unrolling할 수 있는 것은 아니다. Unrolling 수행이 부적합한 몇 가지 형태의 루프에 대해 소개한다.

4.3.3.1. Loops with Low Trip Counts

unrolling이 적합한 루프는 반복 계산의 횟수가 많은 루프이다. 반복계산 횟수가 작다면 preconditioning 루프가 상대적으로 많은 일을 하게 되고, 이것은 결국 코드의 클러터가 된다.
반복 계산 횟수가 작아도 unrolling을 수행하는 경우는 반복 횟수가 상수이고 그 값이 컴파일이 수행될 때 알려져 있는 경우이다.

```
PARAMETER (N = 3)
DO I = 1, N
   A(I) = B(I) * C
ENDDO
```

3회의 반복계산을 가지는 위 코드는 preconditioning 루프에 관한 부담 없이 3의 깊이를 가지도록 unrolling 될 수 있다.

```
PARAMETER (N = 3)
A(1) = B(1) * C
A(2) = B(2) * C
A(3) = B(3) * C
```

4.3.3.2. Fat 루프

unrolling은 루프 반복당 더 많은 계산을 수행하도록 해서 성능향상 효과를 얻는다. 만약 어떤 루프가 이미 반복당 많은 계산을 하고 있다면(fat 루프), unrolling의 효과가 그다지 크지 않을 것이다. Fat 루프에 대한 unrolling은 텍스트 세그먼트의 크기를 증가시켜 메모리 시스템에 부하를 가중 시키므로 오히려 성능손실을 가져올 수도 있다.

4.3.3.3. Loops Containing Procedure Calls

반복마다 프로시저를 호출하는 루프는 이미 fat 루프일 가능성이 높다. 그리고, 컴파일러는 프로시저와 그것을 호출하는 프로그램 단위를 독립적으로 컴파일 하므로 명령어 intermix를 수행하기 어렵다. 프로시저 호출에서는 레지스터가 저장되어야 하고 인수 리스트가 준비 되어야 하는 등 그 자체의 부하가 커서 프로시저를 호출하고 리턴하는데 소비되는 시간이 루프 부하 자체보다 더 클 수 있다.

루프 내에서 호출되는 프로시저는 가급적 루프 밖으로 이동시킨 후, 해당 루프에 대한 unrolling 수행의 가능성을 점검하는 것이 좋다.

4.3.3.4. Loops with Branches in Them

루프 내에 분기를 가지면 unrolling이 부적절하지만, 반복과 무관한 분기를 가진다면 unrolling을 수행해 성능향상 효과를 볼 수도 있다.

```
DO I = 1, N
   DO J = 1, N
      IF (B(J,I) .GT. 1.) A(J,I) = A(J,I) +  B(J,I) * C
   ENDDO
ENDDO
```

위의 코드에서 각 반복은 다른 모든 반복에 대해 독립적이어서 unrolling 수행으로 성능향상을 기대할 수 있다.

```
II = IMOD(N,4)
DO I = 1, N
   DO J = 1, II
      IF (B(J,I) .GT. 1.) A(J,I) = A(J,I) +  B(J,I) * C
   ENDDO
   DO J = II+ 1, N, 4
      IF (B(J,I) .GT. 1.) A(J,I) = A(J,I) +  B(J,I) * C
      IF (B(J+ 1,I) .GT. 1.) A(J+ 1,I) = A(J+ 1,I) +  B(J+ 1,I) * C
      IF (B(J+ 2,I) .GT. 1.) A(J+ 2,I) = A(J+ 2,I) +  B(J+ 2,I) * C
      IF (B(J+ 3,I) .GT. 1.) A(J+ 3,I) = A(J+ 3,I) +  B(J+ 3,I) * C
ENDDO
```

4.3.4. Nested Loops

루프 안에 다른 루프가 들어가는 형태의 루프를 loop nest라고 한다. 루프 nest에서 바깥쪽 루프

를 outer 루프, 안쪽 루프를 inner 루프로 구분한다. 다차원 배열에 대한 계산을 수행할 때 흔히 loop nest를 사용하게 되는데, loop nest는 루프 ordering에 대해 유연성이 있어 적절치 못한 ordering으로 인해 성능손실이 발생할 수 있다.
loop nest에 대한 여러 가지 최적화 방법이 있는데 대부분 메모리 접근 패턴에 대한 개선을 수행하는 것이다.

4.3.4.1. Outer Loop Unrolling

대부분 loop nest에서 inner 루프에 대한 unrolling을 수행하지만, outer loop에 대해 unrolling을 수행해야 할 필요가 있는 경우도 있다.

```
DO I = 1, N
   DO J = 1, N
      DO K = 1, N
         A(K,J,I) = A(K,J,I) +  B(K,J,I)*C
      ENDDO
   ENDDO
ENDDO
```

위 코드에서 K 루프의 바깥 루프인 J 루프에 대해 unrolling을 수행하면 다음과 같다.

```
DO I = 1, N
   DO J = 1, N, 2
      DO K = 1, N
         A(K,J,I) = A(K,J,I) +  B(K,J,I)*C
         A(K,J+ 1,I) = A(K,J+ 1,I) +  B(K,J+ 1,I)*C
      ENDDO
   ENDDO
ENDDO
```

다음과 같이 K 루프에 대한 unrolling을 동시에 수행할 수도 있다.

```
DO I = 1, N
   DO J = 1, N, 2
      DO K = 1, N, 2
         A(K,J,I) = A(K,J,I) +  B(K,J,I)*C
         A(K,J+ 1,I) = A(K,J+ 1,I) +  B(K,J+ 1,I)*C
```

```
            A(K+ 1,J,I) = A(K+ 1,J,I) +  B(K+ 1,J,I)*C
            A(K+ 1,J+ 1,I) = A(K+ 1,J+ 1,I) +  B(K+ 1,J+ !,I)*C
         ENDDO
      ENDDO
   ENDDO
```

4.3.4.2. Outer Loop Unrolling to Expose Computations

loop nest에서 inner 루프의 반복 횟수가 작다면 inner 루프에 대한 unrolling 효과를 보기 어렵다. 이때 outer 루프에 대한 unrolling을 통해 성능향상을 기대할 수 있다.

```
   DO I = 1, N
      DO J = 1, M          ! M is small
         A(J,I) = B(J,I) +  C(J,I)*D
      ENDDO
   ENDDO
```

위 코드에 대해 I 루프에 대한 unrolling을 수행하면 다음과 같다.

```
   II = IMOD(N,4)
   DO I = 1, II
      DO J = 1, M
         A(J,I) = B(J,I) +  C(J,I)*D
      ENDDO
   ENDDO
   DO I = 1+ II,N,4
      DO J = 1, M
         A(J,I) = B(J,I) +  C(J,I)*D
         A(J,I+ 1) = B(J,I+ 1) +  C(J,I+ 1)*D
         A(J,I+ 2) = B(J,I+ 2) +  C(J,I+ 2)*D
         A(J,I+ 3) = B(J,I+ 3) +  C(J,I+ 3)*D
      ENDDO
   ENDDO
```

이과 같은 outer 루프 unrolling에서 메모리 접근이 연속적이지 못해 캐시 실패가 많이 발생한다는 문제가 있으나, inner 루프의 반복 횟수가 작아 실패율이 그다지 높지는 않을 것으로 기대할 수 있다.

inner 루프에 recursion이 있어 unrolling을 수행하기 어려울 때, outer 루프에 대한 unrolling으로 성능향상을 기대할 수 있다.

```
DO I = 1, N
   DO J = 2, M
      A(J,I) = A(J,I) +  A(J-1,I)*D
   ENDDO
ENDDO
```

다음과 같이 outer 루프에 대한 unrolling을 수행한다.

```
II = IMOD(N,4)
DO I = 1, II
   DO J = 2, M
      A(J,I) = A(J,I) +  A(J-1,I)*D
   ENDDO
ENDDO
DO I = 1+ II,N,4
   DO J = 2, M
      A(J,I) = A(J,I) +  A(J,I)*D
      A(J,I+ 1) = A(J,I+ 1) +  A(J-1,I+ 1)*D
      A(J,I+ 2) = A(J,I+ 2) +  A(J-1,I+ 2)*D
      A(J,I+ 3) = A(J,I+ 3) +  A(J-1,I+ 3)*D
   ENDDO
ENDDO
```

4.3.5. Loop Interchange

loop nest에서 inner 루프와 outer 루프를 재배치시켜 이후 unrolling과 같은 다른 최적화를 수행하기 편리하도록 한다. 반복 횟수가 많은 루프를 안쪽으로 배치시키면 inner 루프에 대한 unrolling으로 성능향상 효과를 기대할 수 있다.

```
PARAMETER(IDIM=1000, JDIM=1000, KDIM=3)
...
DO I = 1, IDIM
   DO J = 1, JDIM
      DO K=1, KDIM
```

```
            D(K,J,I) = D(K,J,I) +  V(K,J,I)*DT
        ENDDO
    ENDDO
ENDDO
```

위의 코드는 다음과 같이 루프를 재배치시킨 후 inner 루프에 대한 unrolling을 수행할 수 있다.

```
DO K = 1, KDIM
    DO J = 1, JDIM
        DO K=1, IDIM
            D(K,J,I) = D(K,J,I) +  V(K,J,I)*DT
        ENDDO
    ENDDO
ENDDO
```

병렬화 적합한 루프는 outer 루프로 배치시키고, inner 루프의 메모리 접근을 최적화 하는 방향으로 배치하는 것이 좋다. 이러한 루프 재배치를 수행할 경우 재배치에 의해 결과가 달라지지 않도록, 수행되는 계산의 의존성에 대해 항상 주의 하여야 한다. 아래 코드는 이러한 의존성으로 인해 루프 재배치가 어려운 경우이다.

```
DO I = 1, N-1
    DO J=2, N
        A(I,J) = A(I+ 1, J-1) * B(I,J)
        C(I,J) = B(J,I)
    ENDDO
ENDDO
```

4.3.6. 메모리 접근 패턴

메모리 접근은 캐시적중이 최대가 되도록 하는 것이 중요하다. 캐시실패를 줄임으로써 메모리 접근 속도를 향상시킬 수 있는데 이를 위해서 연속적인 데이터 접근이 되도록 해야 한다. 연속적인 메모리 접근은 순차적으로 최소단위만큼 증가하도록 접근하는 것이다. 이를 위해 1차원 배열 접근은 한 번에 한 개의 원소씩 순차적으로 접근해 가도록 한다. 다차원 배열에서의 연속 메모리 접근을 위해서는 우선적으로 Fortran과 C 언어의 배열 저장방식에 대한 이해가 필요하다.
어떤 경우든 다차원 배열은 1차원 배열로 메모리에 저장되지만, Fortran의 경우는 열우선 순으로 C의 경우는 행우선 순으로 메모리에 저장된다. 따라서 연속 메모리 접근이 되도록 하기 위해 Fortran에서는 leftmost subscript로 C에서는 rightmost subscript로 접근해야 한다.

```
DO J = 1, N
    DO I=1, N
        A(I,J) = B(I,J) +  C(I,J)*DT
    ENDDO
ENDDO
```

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        a[i][j] = b[i][j] +  c[i][j]*dt
```

메모리 접근을 연속적으로 하는 것은 캐시와 메모리 성능 향상을 위한 가장 기본적인 최적화 기법이다. 컴파일러가 reorder할 수도 있지만 항상 메모리 접근 stride가 1이 되도록 코드를 구성하는 것이 좋다. Stride 1이 불가능한 코드의 경우 data cache blocking 기법을 사용해 캐시실패를 줄이도록 한다.

4.3.6.1. Loop Interchange to Ease Memory Access Patterns

각 반복이 서로에 대해 독립적이면, loop nest를 재배치해 연속 메모리 접근이 되도록 할 수 있다.

```
DO J = 1, N
    DO I=1, N
        A(J,I) = B(J,I) +  C(J,I)*DT
    ENDDO
ENDDO
```

위의 코드는 다음과 같이 루프를 재배치시켜 메모리 접근이 연속이 되도록 할 수 있다.

```
DO I = 1, N
    DO J=1, N
        A(J,I) = B(J,I) +  C(J,I)*DT
    ENDDO
ENDDO
```

4.3.6.2. 행렬의 곱셈

다음은 흔히 볼 수 있는 두 행렬의 곱셈을 수행하는 코드이다.

```
DO I = 1, N
   DO J=1, N
      SUM=0.0
      DO K=1,N
         SUM = SUM +  A(I,K)*B(K,J)
      ENDDO
      C(I,J) = SUM
   ENDDO
ENDDO
```

위의 코드에서 A(I,K)에 대한 접근은 연속적이지 않다. inner 루프의 각 반복은 두 번의 load(한 번은 불연속 접근), 한 번의 곱셈과 한 번의 덧셈으로 구성된다.
다음과 같이 코드를 구성하여 모든 메모리 접근이 연속적으로 수행되도록 할 수 있다.

```
DO J = 1, N
   DO I=1, N
      C(I,J) = 0.0
   ENDDO
ENDDO
DO K=1,N
   DO J=1,N
      SCALE = B(K,J)
      DO I=1,N
         C(I,J) = C(I,J) +  A(I,K)*SCALE
      ENDDO
   ENDDO
ENDDO
```

4.3.6.3. 루프 재배치가 불가능한 경우

루프 재배치에 의해 항상 메모리 접근 패턴이 연속이 되도록 할 수 있는 것은 아니다.

```
DO I = 1, N
   DO J=1, M
      A(J,I) = B(I,J)
   ENDDO
ENDDO
```

```
DO J = 1, M
   DO J=1, N
      A(J,I) = B(I,J)
   ENDDO
ENDDO
```

위와 같은 몇 가지 예에서 보듯 루프 배치를 어떻게 하더라도 불연속 메모리 접근이 발행하게 되는 경우가 있다. 이때 프로그래머는 메모리 load를 연속적으로 접근할 것인지, 메모리 store를 연속적으로 하게 할 것인지 판단을 해야 하지만 둘 모두에 대해 메모리 접근 패턴을 개선할 수 있는 방법이 있다. 다음에서 이에 대해 알아본다.

4.3.7. Loop Blocking

캐시 적중률을 높이기 위해 메모리 참조를 최적화 하는 방법으로 앞서의 예와 같이 단순히 루프 재배치에 의해 메모리 참조를 연속적으로 하기 어려운 경우 또는 캐시보다 훨씬 큰 크기를 가지는 배열 데이터를 참조해야 할 때 유용하게 사용할 수 있다. 루프 블로킹은 기본적으로 strip mining과 루프 interchange를 통해 데이터 지역성을 최대로 하는 것이 목표이다.

4.3.7.1. Strip Mining

strip mining은 루프 블로킹, Unrolling, Fusion 또는 병렬화를 위해 사용된다. strip mining은 단일 루프를 splitting해서 루프 nest로 만든다. 그 결과 inner 루프는 원래 루프의 한 구역(strip)에 걸쳐 반복되고, outer 루프의 반복에 의해 모든 구역에 걸친 루프 계산이 완료되게 된다. 이때 inner 루프의 반복 횟수를 ‘strip length’라고 한다.

```
DO I = 1, 10000
   A(I) = A(I) * B(I)
ENDDO
```

위의 루프 계산을 strip length = 1000으로 하여 strip mining을 하면 다음과 같다.

```
DO IOUTER = 1, 10000, 1000
   DO ISTRIP = IOUTER, IOUTER+ 999
      A(ISTRIP) = A(ISTRIP) * B(ISTRIP)
   ENDDO
ENDDO
```

위 코드에서는 루프 반복이 동일하게 분할 되었지만 경우에 따라서는 전체 루프 반복 횟수가 strip length의 배수가 되지 않는 경우도 있을 것이다.

4.3.7.2. Simple Loop Blocking

캐시보다 큰 배열을 처리하기 위해 루프 블로킹을 수행하는 과정을 살펴보자. 블로킹 작업은 컴파일러가 의해 수행할 수 있다.
다음 코드에서 배열은 약 16메가바이트 정도의 메모리 공간을 차지한다. 이런 경우 루프 블로킹을 통해 메모리 접근 속도를 향상 시킬 수 있다.

```
REAL*8 A(1000,1000), B(1000,1000)
REAL*8 C(1000), D(1000)
COMMON /BLK/ A, B, C
...
DO J = 1, 1000
   DO I = 1, 1000
      A(I,J) = B(J,I) +  C(I) +  D(J)
   ENDDO
ENDDO
```

우선 I 루프에 대해 strip mining을 하면 다음과 같다.

```
DO J = 1, 1000
   DO IOUT = 1, 1000, IBLOCK
      DO I = IOUT, IOUT+ IBLOCK-1
         A(I,J) = B(J,I) +  C(I) +  D(J)
      ENDDO
   ENDDO
ENDDO
```

여기서 IBLOCK이 strip length(또는 block factor라고도 함)가 되며 그 크기는 배열과 캐시 크기에 의해 결정된다. 다음 과정은 바깥쪽 strip 루프(IOUT)를 가능한 바깥쪽으로 이동시키는 것이다(interchange).

```
DO IOUT = 1, 1000, IBLOCK
   DO J = 1, 1000
      DO I = IOUT, IOUT+ IBLOCK-1
```

```
            A(I,J) = B(J,I) +  C(I) +  D(J)
          ENDDO
        ENDDO
      ENDDO
```

새로운 코드에서는 모든 J에 대해 배열 A의 IBLOCK개 행과 배열 B의 IBLCOK개 열이 접근되고 있다. IOUT의 모든 반복에 대해 배열 A에서는 세로 폭이 IBLOCK인 열이 1000회 접근되고, 배열 B에서는 가로 폭이 IBLOCK인 행이 1000회 접근된다.

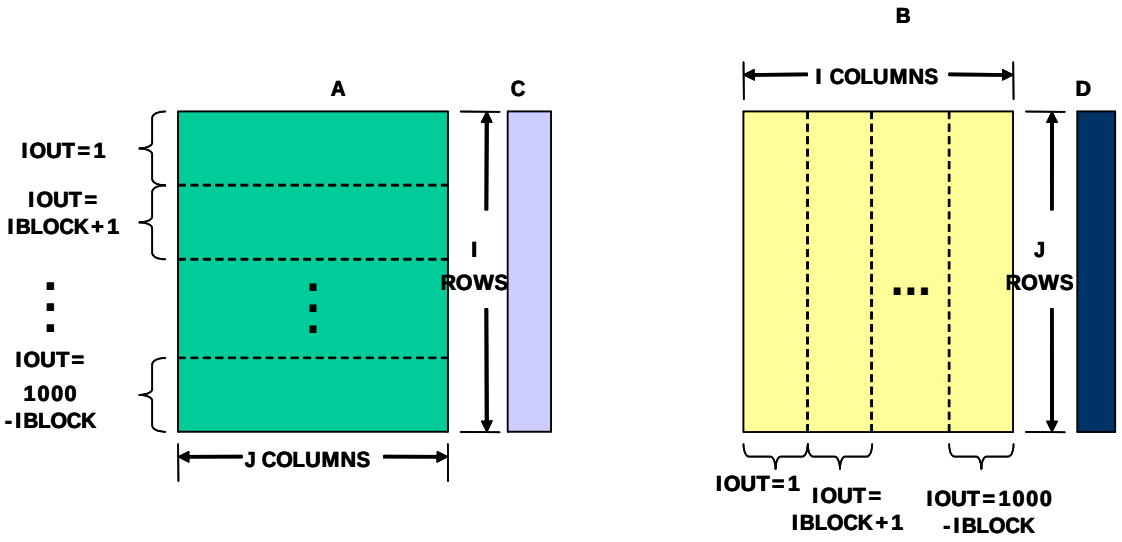


그림 4-1. Loop Blocking에 의한 데이터 접근

메모리에서 배열 A의 원소를 하나 꺼내오면 더불어 연속된 세 개의 원소(캐시라인 크기에 의존)가 같이 캐시에 저장된다. 세 개의 원소는 이어지는 반복 계산에서 사용되므로 A의 공간적 지역성을 이용된다. I 루프는 배열 B의 행 원소에 접근하므로 B의 데이터 하나를 꺼내오면서 같이 캐시에 올라오는 데이터들에 대해 J값이 증가하기 전까지는 공간적인 지역성 이용이 불가능하다. IBLOCK은 배열 A와 B의 공간적 지역성을 잘 활용할 수 있도록 그 크기가 선택 되어야 한다. 이것은 컴파일러가 결정하지만 핸드 튜닝의 경우라면 프로그래머가 직접 그 크기를 정해야 할 것이다.

코드 실행과정에서 각 배열의 캐시라인들이 어떻게 접근되는지 아래 그림에서 확인할 수 있다. 그림에서 색칠된 부분은 처음 캐시라인에 올라오는 데이터들을 표시한다.

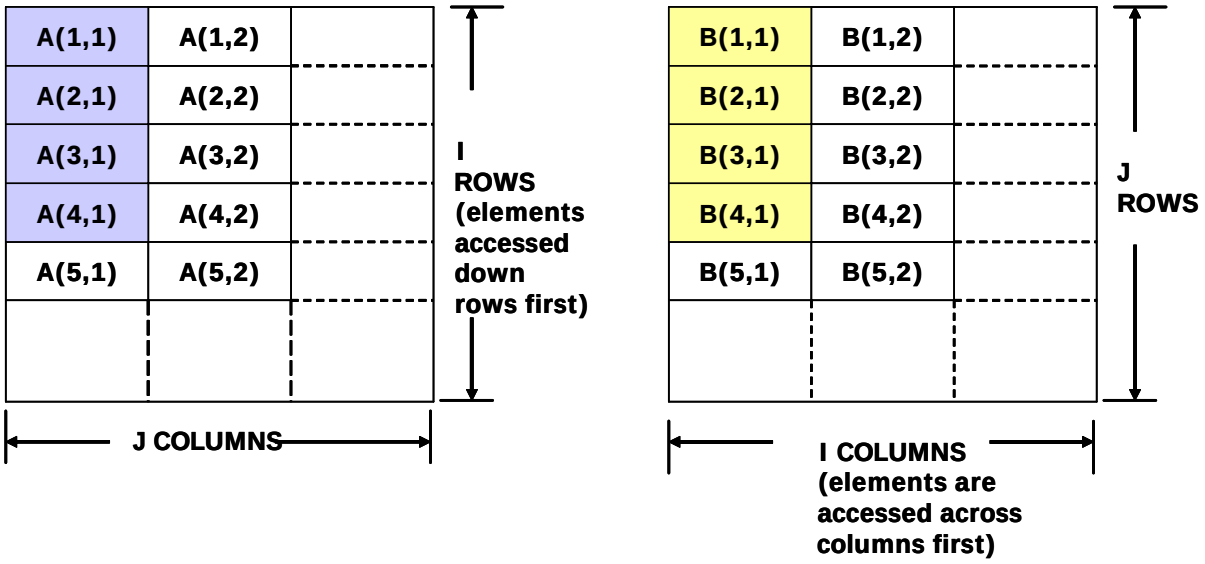


그림 4-2. 각 배열의 캐시라인 접근

- A(1,1)이 접근될 때 A(1:4,1)이 꺼내진다. A(2:4,1)은 I의 연속되는 2,3,4 반복에서 사용된다.
- B(1:4,1)은 I=1일 때, 꺼내지지만, B(2:4,1)은 J값이 2,3,4가 될 때까지 사용되지 않는다. B(1:4,2)는 I=2일 때 꺼내진다.

C의 IBLOCK개 원소는 덮어쓰기 되기 전에 J의 몇 차례 반복 동안 캐시에 남아 있어서 그 반복 횟수 동안 시간적 지역성이 이용된다. 배열 D는 모든 I의 반복 동안 레지스터에 남아 있게 된다.

4.3.7.3. Loop Blocking: 행렬 곱셈

```
INTEGER, PARAMETER : : N = (SOME # ) , M = (SOME #)
INTEGER, PARAMETER : : K = (SOME # )
REAL(8), DIMENSION (M, N) : : C
REAL(8), DIMENSION (M, L) : : A
REAL(8), DIMENSION (L, N) : : B
COMMON/BLOK/C, A, B
. . .
DO J = 1, N
    DO I = 1, M
        DO K = 1, L
            C(I, J) = C(I, J) +  A(I, K) * B(K, J)
        END DO
    END DO
```

END DO

위와 같은 행렬의 곱셈에서 배열 C의 한 원소의 계산은 배열 A의 한 행과 배열 B의 한 열을 필요로 한다.

C(1, 1) = A(1, 1) * B(1, 1) + A(1, 2) * B(2, 1) + . . . + A(1, L) * B(L, 1)
C(2, 1) = A(2, 1) * B(1, 1) + A(2, 2) * B(2, 1) + . . . + A(2, L) * B(L, 1)
…
C(M, N) = A(M, 1) * B(1, N) + A(M, 2) * B(2, N) + . . . + A(M, L) * B(L, N)

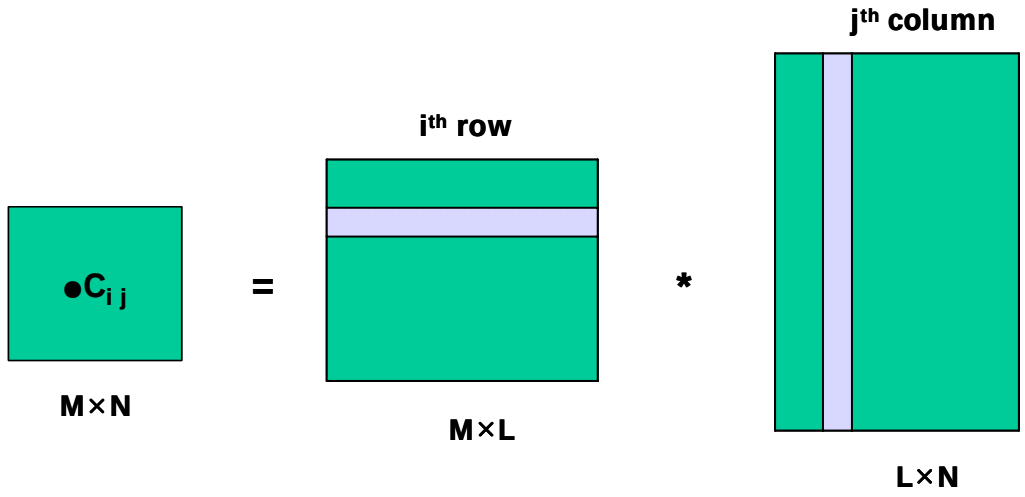


그림 4-3. 행렬의 곱셈

따라서 배열 C의 M*N개 원소 계산을 위해 배열 A의 M개 행이 N번 사용되고 배열 B의 N개 행이 M번 사용 된다. 이와 같은 패턴으로 메모리의 데이터를 꺼내오는 것이 큰 부하를 발생시킨다. 첫 계산에서 A(1,1)과 같이 A(2,1), A(3,1), A(4,1)이 올라온다. 그렇지만 다음 계산에서 A(2,1)이 아니라 A(1,2)가 필요하다. A(2,1)은 L+1번째 계산에서 필요하게 된다. 그래서 L번의 루프 계산 동안 A(2,1)이 캐시에 남아 있게 된다면 배열 A의 공간적 지역성 사용이 가능해 진다. A(3,1)과 A(4,1)은 각각 2L+1, 3L+1 번째 계산에서 필요하고 이때까지 캐시에 남아 있을 수 있다면 좋을 것이다. B(1,1)은 첫 루프 계산에서 캐시로 올라오고 배열 C의 M개 원소 계산 동안 계속 사용 돼야 한다. B(1,1)이 M*L 루프 반복 동안 캐시에 남아 있으면 시간적 지역성 이용이 가능해 질것이다.
만약 A, B, C가 캐시 크기에 비해 크다면, 캐시 라인들에 겹쳐 쓰기가 발생해 이러한 공간적, 시간적 지역성은 감소되거나 아예 사라져 버릴 수도 있다. L가 충분히 크다면 A(2,1)은 L+1번째 루프 계산 이전에 사라져 버리고 그 만큼 많은 캐시 미스가 발생하게 되는 것이다.
이러한 상황에서 캐시 블로킹을 통해 성능 저하를 방지할 수 있다. 블로킹은 많은 항을 포함하는

계산을 소수의 항을 가지는 일련의 독립된 계산으로 나누는 것이다. 독립적인 계산 각각에 포함되는 항의 개수는 그 계산에 필요한 모든 데이터가 캐시에 들어갈 수 있도록 설정한다.
C(I, 1) = A(I, 1) * B(1, 1) + A(I, 2) * B(2, 1) + . . . + A(I, L) * B(L, 1)의 계산을 L/KBLOCK 계산으로 나눈다.

Strip Mining:

```
DO KOUT = 1, L, KBLOCK
   DO K = KOUT, KOUT + (KBLOCK - 1)
      C(I,1) = C(I,1) + A(I,K) * B(K,1)
   END DO
END DO
```

배열 C의 전체 첫 열은 인덱스 I의 반복에 의해 계산 가능하다.

```
DO I = 1, M
   DO KOUT = 1, L, KBLOCK
      DO K = KOUT, KOUT + (KBLOCK - 1)
         C(I,1) = C(I,1) + A(I,K) * B(K,1)
      END DO
   END DO
END DO
```

내포된 루프의 첫 K 반복 계산 동안 B(1,1)은 한 번만 사용되고 A(2,1)은 사용되지 않고 있다. 여기서 다음과 같이 KOUT 루프와 I 루프의 위치를 바꿔주면 배열 A의 공간적 지역성과 B(1,1)의 시간적 지역성을 개선 시킬 수 있다.

Interchange:

```
DO KOUT = 1, L, KBLOCK
   DO I = 1, M
      DO K = KOUT, KOUT + (KBLOCK - 1)
         C(I,1) = C(I,1) + A(I,K) * B(K,1)
      END DO
   END DO
END DO
```

A(2,1)은 KBLOCK 반복 이후에 사용되고 B(1,1)은 모든 KBLOCK 반복 계산에서 사용된다. 만약

M이 충분히 크다면 I 루프에 대한 unrolling 이나 블로킹을 통해 더 나은 공간적, 시간적 지역성 이용 효과를 볼 수 있다.

Unrolling:

```
DO KOUT = 1, L, KBLOCK
   DO I = 1, M, IBLOCK
      DO K = KOUT, KOUT + (KBLOCK - 1)
         C(I,1) = C(I,1) + A(I,K) * B(K,1)
         C(I + 1,1) = C(I + 1,1) + A(I + 1,K) * B(K,1)
         C(I + 2,1) = C(I + 2,1) + A(I + 2,K) * B(K,1)
         ...
         C(I+ IBLOCK-1, 1) = C(I+ IBLOCK-1, 1) + A(I+ IBLOCK-1,K) * B(K,1)
      END DO
   END DO
END DO
```

Blocking:

```
DO IOUT = 1, M, IBLOCK
   DO KOUT = 1, L, KBLOCK
      DO J = 1, N
         DO I = IOUT, IOUT+ IBLOCK-1
            DO K = KOUT, KOUT+ KBLOCK-1
               C(I, J) = C(I, J) + A(I, K) * B(K, J)
            END DO
         END DO END DO
      END DO
   ENDDO
ENDDO
```

이와 같이 캐시 블로킹을 통하여 다음과 같은 효과를 얻을 수 있다.

- K루프에 관하여 B의 공간적 지역성 사용
- I루프에 관하여 B의 시간적 지역성 사용
- I루프에 관하여 A의 공간적 지역성 사용
- J루프에 관하여 A의 시간적 지역성 사용
- I루프에 관하여 C의 공간적 지역성 사용
- K루프에 관하여 C의 시간적 지역성 사용

4.3.8. Loop Fusion

루프 fusion은 루프 인덱스의 범위와 반복횟수가 동일한 여러 루프를 합쳐서 하나의 루프로 만드는 것을 의미한다. 이로 인해 루프 부하, 메모리 접근을 감소시키고 레지스터 이용 정도를 높일 수 있다. 또한 fusion을 통해 합쳐진 루프가 병렬화 가능한 경우, 루프 하나당 작업량이 증가돼 상대적으로 병렬화 효율을 높일 수 있게 된다.
fusion이 불가능한 모양의 루프도 interchange 등의 루프 변환을 거쳐 fusion이 가능해 질 수 있다. Fortran의 배열표현은 컴파일러에 의해 루프로 처리되는데 이때 fusion이 가능한 연산들은 모두 fused되어 처리된다.

```
DO I = 1, N
   A(I) = B(I) +  C(I)
ENDDO
DO J = 1, N
   IF(A(J) .LT. 0)   A(J) = B(J)*B(J)
ENDDO
```

위의 두 루프는 다음과 같이 fusion될 수 있다.

```
DO I = 1, N
   A(I) = B(I) +  C(I)
   IF(A(I) .LT. 0)   A(I) = B(I)*B(I)
ENDDO
```

다음은 Fortran 코드에서의 배열 표현을 컴파일러가 처리하는 과정에서 루프 fusion을 사용하는 예이다.

```
REAL A(100,100), B(100,100), C(100,100)
...
C = 2.0 * B
A = A +  B
```

컴파일러는 위의 표현을 다음과 같은 형태의 루프로 변환한다.

```
DO TEMP1 = 1, 100
   DO TEMP2 = 1, 100
```

```
        C(TEMP2, TEMP1) = 2.0 * B(TEMP2, TEMP1)
      ENDDO
    ENDDO
    DO TEMP3 = 1, 100
      DO TEMP4 = 1, 100
        A(TEMP4,TEMP3)=A(TEMP4,TEMP3)+ B(TEMP4,TEMP3)
      ENDDO
    ENDDO
```

이때 위의 두 루프는 다음과 같이 fusion돼 처리될 수 있다.

```
    DO TEMP1 = 1, 100
      DO TEMP2 = 1, 100
        C(TEMP2,TEMP1) = 2.0 * B(TEMP2, TEMP1)
        A(TEMP2,TEMP1)=A(TEMP2,TEMP1)+ B(TEMP2,TEMP1)
      ENDDO
    ENDDO
```

이렇게 fusion된 루프에 대해 컴파일러는 다른 최적화를 시도할 수 있다.

4.3.8.1. Loop Peeling

이웃 루프와 반복횟수가 하나(+1 또는 -1) 차이가 나면 컴파일러는 두 루프 중 하나에서 반복을 peeling 한 후 fusion을 시도한다.

```
    DO I = 1, N-1
      A(I) = I
    ENDDO
    DO J = 1, N
      A(J) = A(J) +  1
    ENDDO
```

두 루프의 인덱스가 1만큼 차이가 나므로 J루프에서 N번째 계산을 밖으로 빼버리면 두 루프의 인덱스 반복이 동일하게 될 것이다.

```
    DO I = 1, N-1
      A(I) = I
```

```
ENDDO
DO J = 1, N-1
   A(J) = A(J) +  1
ENDDO
A(N) = A(N) +  1
```

peeling을 거쳐 두 개의 루프는 다음과 같이 fusion될 수 있다.

```
DO I = 1, N-1
   A(I) = I
   A(I) = A(I) +  1
ENDDO
A(N) = A(N) +  1
```

4.3.9. Loop Distribution

루프 distribution은 loop nest의 계산을 innermost 루프에서 실행되도록 루프를 변환하는 것이다. distribution은 루프 반복당 메모리 참조를 감소시키고, 캐시 thrashing 발생을 감소시키며, 또한 interchange를 통한 루프 최적화 기회를 증가시킨다.

```
DO I = 1, N
   C(I) = 0
   DO J = 1, M
      A(I,J) = A(I,J)+ B(I,J)*C(I)
   ENDDO
ENDDO
```

위의 코드에서 배열 C에 대한 할당을 따로 분리하면 loop nest의 모든 할당이 innermost 루프에서 이루어지도록 할 수 있다.

```
DO I = 1, N
   C(I) = 0
ENDDO
DO I = 1, N
   DO J = 1, M
      A(I,J) = A(I,J)+ B(I,J)*C(I)
   ENDDO
```

```
ENDDO
```

이와 같은 변환을 거친 코드는 다음과 같이 interchange를 통해 최적화 될 수 있다.

```
DO I = 1, N
   C(I) = 0
ENDDO
DO J = 1, M
   DO I = 1, N
      A(I,J) = A(I,J)+ B(I,J)*C(I)
   ENDDO
ENDDO
```

4.4. 기타 메모리 최적화

4.4.1. Cache Thrashing

캐시 thrashing은 프로그램에서 자주 사용되는 두 개 혹은 그 이상의 데이터가 동일한 캐시 주소에 사상되는 상황에서 발생하게 된다. 캐시 thrashing이 발행하면 필요한 데이터가 캐시에 올라올 때마다 이미 올라와 있는 또 다른 필요 데이터를 덮어쓰게 돼 캐시실패를 유발하고 데이터 재사용에 나쁜 영향을 준다.
1M의 크기를 가지는 직접 사상 캐시 시스템에서 아래 코드를 실행하는 경우에 대해 살펴보자.

```
INTEGER, PARAMETER :: N = 128*512
REAL(8), DIMENSION(N) :: A, B, C
COMMON/BLOK/A,C,B
. . .
DO I = 1, N
  C(I) = A(I) +  B(I)
ENDDO
```

세 배열 A, B, C 각각 8*(128*512) = 524288 bytes = 0.5 M의 크기를 가진다. 그리고, 각 배열은 common block의 사용으로 A, C, B 순서로 메모리에 연속적으로 위치한다. 직접 사상 1M 캐시를 가지는 시스템에서 코드를 실행한다면 아래 그림에서와 같이 배열 A와 B의 각 원소는 캐시 내에서 동일한 주소를 가지게 된다.

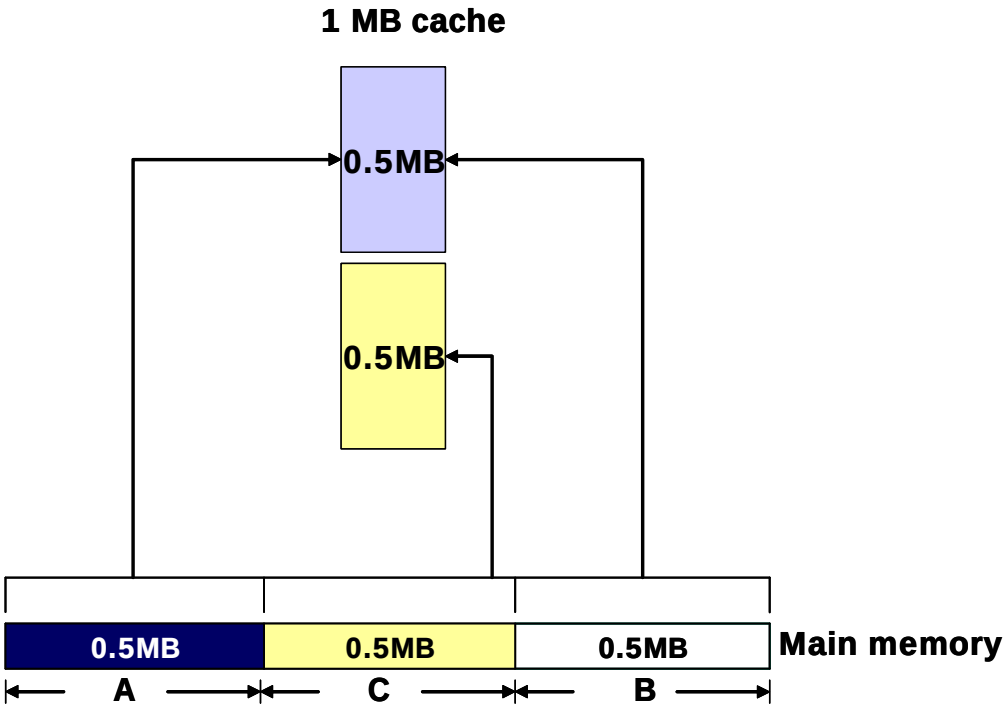


그림 4-4. Cache Thrashing

배열 A, B는 루프의 반복마다 서로를 thrash 한다. 이처럼 한 캐시라인이 다른 캐시라인에 의해 overwrite 되는 것을 캐시 thrashing이라고 한다.
캐시라인 크기가 32바이트(64바이트)인 경우 세 개(일곱 개)의 8바이트 데이터가 추가적으로 메모리에서 꺼내어져 캐시에 올라온다. 추가적으로 올라온 데이터가 사용될 때까지 캐시에 있도록 해서 캐시실패를 줄여야 하지만 thrashing이 발생하면 추가적으로 올라온 데이터는 참조되기 전에 삭제돼 버려, 결국 메모리 성능저하의 원인이 된다.

```
INTEGER, PARAMETER :: N = 128*512
REAL(8), DIMENSION(N) :: A, B, C
COMMON/BLOK/C, A, B
. . .
```

위의 코드에서도 여전히 캐시 thrashing은 발생한다. load뿐 아니라 store 과정에서도 캐시라인에 대한 겹쳐 쓰기 문제가 남아있기 때문이며, C(j)가 B(j)를 겹쳐 쓰게 되는 문제가 있다.

4.4.1.1. Cache Padding

캐시 thrashing은 두 가지 방식의 캐시 padding에 의해 최소화 될 수 있다.

- 데이터 정렬 padding

- 배열 내부 padding

① 데이터 정렬 padding

두 배열이 캐시 내에서 같은 주소를 공유할 때 한 배열을 다른 캐시라인을 사용하도록 이동시켜 캐시 thrashing을 제거할 수 있다. 배열을 이동시키기 위해서는 아래 코드와 같이 thrashing이 발생하는 배열과 배열 사이에 padding 배열(배열 fake)을 삽입하는 방법이 있다.
캐시라인의 크기가 32바이트 라면 다음과 같은 padding을 생각할 수 있다.

```
INTEGER, PARAMETER :: N = 128*512
INTEGER, PARAMETER :: M = 4
REAL(8), DIMENSION(N) :: A, B, C
REAL(8), DIMENSION(M) :: FAKE
COMMON/BLOK/A,C,FAKE,B
. . .
DO I = 1, N
    C(I) = A(I) + B(I)
ENDDO
```

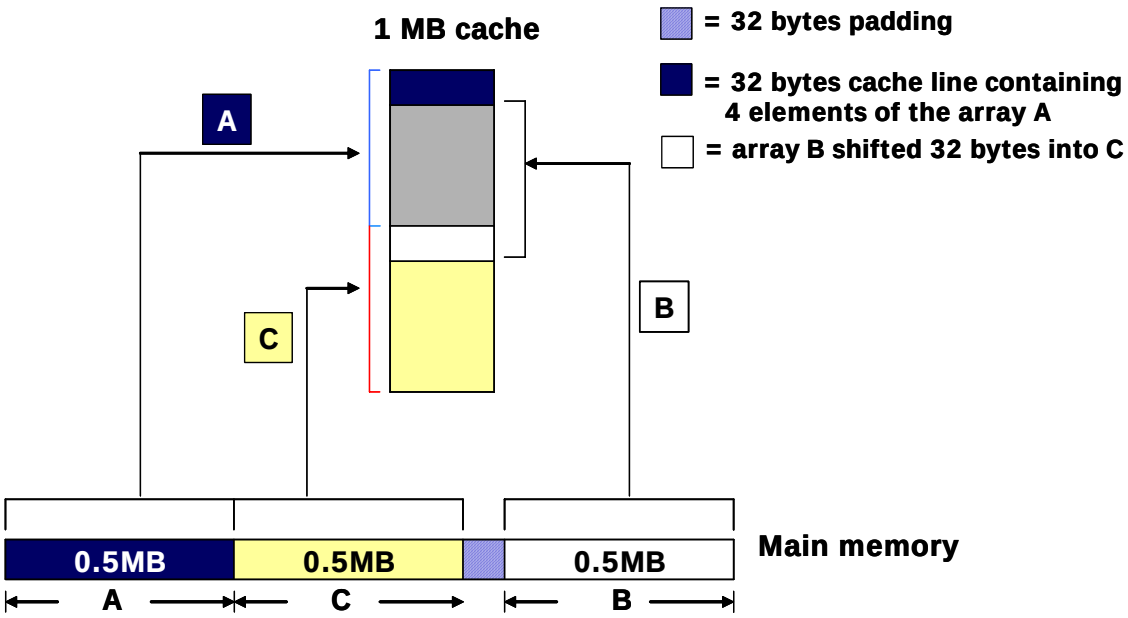


그림 4-5. 데이터 정렬 padding

common block에 배열 fake를 삽입함으로써 배열 B의 캐시주소를 배열 C의 주소로부터 32바이트 아래로 이동 시키는 효과를 볼 수 있다. 이 32바이트 이동이 같은 인덱스를 가지는 두 개의

원소가 동일한 캐시주소를 공유하는 것을 방지해서 전체 캐시라인이 참조되기 전까지 캐시 겹쳐쓰기가 발행하지 않게 된다. 그래서 배열 A, B, C에서 각각 4개의 8바이트 원소가 캐시에 올려지면 모두 참조되고 난 후 삭제된다.
padding은 최신의 컴파일러에서 자동적으로 처리된다. 컴파일러는 배열 원소의 크기, 캐시라인의 크기, 메모리 크기 등을 고려해 pad 크기 결정하게 된다. padding을 수행한 컴파일러는 명시적인 리포트를 통해 pad된 캐시 블록과 더불어 그 크기를 보여줄 것이다.

② 배열 내부 padding
padding의 또 다른 방법은 한 배열의 캐시주소를 이동시켜 캐시 thrashing 발생을 제거하는 것으로 배열 padding이라고 한다.

```
INTEGER, PARAMETER :: N = 128*512
REAL(8), DIMENSION(N) :: B, C
REAL(8), DIMENSION(N+ 4) :: A
COMMON/BLOK/A,C,B
. . .
DO I = 1, N
  C(I) = A(I) +  B(I)
ENDDO
```

위의 코드에서와 같이 배열 A의 길이를 4개(32바이트) 더 늘여 놓음으로써 배열 B가 배열 A와 캐시주소를 공유하지 못하도록 하는 padding의 효과를 얻을 수 있다. 배열 A 대신 배열 C의 크기를 32바이트 증가시키는 것도 같은 효과를 줄 것이다.

4.4.2. 루프와 인덱스 순서
수치계산 프로그램에서 다루는 배열들은 대부분 그 크기가 시스템의 캐시크기보다 더 큰 경우가 많으며, 수행되는 연산은 배열들의 모든 원소에 대해 적용되게 된다. 이로 인해 루프에 의한 계산을 하나 수행할 때마다 캐시 전체가 여러 번 교체되어야 하는 상황이 발생한다. 따라서 데이터의 메모리 레이아웃과 루프의 실행 순서 등이 가급적 캐시실패를 줄일 수 있도록 주의 깊게 다루어져야 한다.

```
!Code to calculate V = (VU) U
REAL V(1024,3), S(1024), U(3)
DO I=1,1024
     S(I) = U(1)*V(I,1)
END DO
```

```
DO I=1,1024
    S(I) = S(I) +  (U(2)*V(I,2))
END DO
DO I=1,1024
    S(I) = S(I) +  (U(3)*V(I,3))
END DO
DO J=1,3
    DO I=1,1024
        V(I,J) = S(I) * U(J)
    END DO
END DO
```

위의 코드에서 배열 S는 쓰기가 세 번, 읽기가 다섯 번해서 모두 여덟 차례 접근되어야 하는데 다음과 같이 코드를 수정하여 S에 대한 접근 횟수를 줄일 수 있다.

```
!Code to calculate V = (VU) U
REAL V(3,1024), U(3)
REAL S
DO I=1,1024
    S = (U(1)*V(1,I)) +  (U(2)*V(2,I)) +  U(3)*V(3,I))
    DO J=1,3
        V(J,I) = S * U(J)
    END DO
END DO
```

여기서 사용되는 스칼라 변수 S는 레지스터에서 접근 가능하므로 메모리 접근 양이 앞선 코드와 비교해 대폭 줄어들게 된다. 또한 배열 V의 인덱스 순서를 바꿔서 메모리 접근이 연속적으로 이뤄 지도록 하고 있다.

4.4.3. 스칼라 임시변수

캐시 접근이 메모리 접근보다는 빠르지만 레지스터 접근 보다는 느리다. 가능하다면 필요한 데이터를 컴파일러가 레지스터에 유지시킬 수 있도록 하는 것이 속도 향상에 도움을 준다. 만약, 특정 값이 두 가지 다른 방법으로 참조된다면, 컴파일러는 해당 데이터를 레지스터에 보관해 두지 않게 된다.

```
DO I=2, N
```

```
        A(I) = A(I-1) +  B(I)
    ENDDO
```

위의 코드는 다음과 같이 명시적으로 값을 스칼라 변수에 저장해 전체 계산수행 동안 스칼라 변수에 접근하도록 수정할 수 있다.

```
    T = A(1)
    DO I = 2, N
       T = T +  B(1)
       A(I) = T
    ENDDO
```

스칼라 변수는 컴파일러가 어떤 변수들을 레지스터에 저장하고 어떤 변수들을 캐시에서 다시 읽어 들일지 판단하기 위한 힌트가 될 수 있다.
대부분 컴파일러가 부가적인 재계산을 피하기 위해 공통된 표현들을 찾아내 스칼라 임시변수로 대체하고 있지만, 프로그래머가 명시적으로 소스코드를 수정하는 것도 성능향상에 도움이 된다.

4.4.4. Recalculating Values

메모리 접근시간이 계산시간보다 상대적으로 크기 때문에, 경우에 따라서는 이미 계산된 값을 메모리에서 읽어 들이지 않고 필요할 때마다 다시 계산하는 방법을 통해 프로그램 성능향상을 얻을 수 있다. 이때 결과를 다시 계산하는 것이 추가적인 메모리 접근 비용과 비교해 얼마나 빠를 것인가에 대한 판단이 필요하다.
다음 코드는 inner 루프 반복마다 4개의 값을 읽어 들이고 3회의 곱셈 계산을 수행하고 있다.

```
    DO I = 1, N
       A2(I) = k*A(I)
    ENDDO
    DO J = 1, 5
       DO I = 1, N
          B(1,I) = B(1,I) * A(I)
          B(2,I) = B(2,I) * A2(I)
          B(1,I) = B(1,I) * B(2,I)
       ENDDO
    ENDDO
```

다음은 위의 코드에 곱셈 계산을 추가시키고 메모리 접근 횟수를 줄인 것이다.

```
DO J = 1, 5
   DO I = 1, N
      B(1,I) = B(1,I) * A(I)
      B(2,I) = B(2,I) * k*A(I)
      B(1,I) = B(1,I) * B(2,I)
   ENDDO
ENDDO
```

이와 같은 최적화는 하드웨어 환경에 매우 의존적이므로 특별히 주의 깊게 사용돼야 한다.

4.4.5. Data Grouping

캐시 사용을 극대화하기 위해 자료 구조의 구성을 변경할 필요가 있는 경우가 있다.

```
D=0.0
DO I=1,N
   J=INDEX(I)
   D = D + SQRT(X(J)*X(J) + Y(J)*Y(J) + Z(J)*Z(J))
ENDDO
```

위의 코드는 배열에 대한 참조가 간접적이다. 각 루프 반복 계산에 대해 x, y, z에 대한 세 개의 새로운 캐시라인이 필요하게 되는데, 다음과 같이 x, y, z를 2차원 배열 r로 grouping해서 코드 수정을 하면 하나의 캐시라인만 필요하게 된다.

```
D=0.0
DO I=1,N
   J=INDEX(I)
   D = D + SQRT(R(1,J)*R(1,J) + R(2,J)*R(2,J) + R(3,J)*R(3,J))
ENDDO
```

r(1,j), r(2,j), r(3,j)는 메모리에서 연속이므로 하나의 캐시라인 필요하다.

4.4.6. Data Alignment

최적의 성능을 위해 데이터는 naturally aligned돼야 한다. natural boundary는 데이터 아이템 크기의 배수가 되는 메모리 주소이다. 예를 들자면, 8바이트 실수 데이터는 시작 주소가 8의 배수가 되는 natural boundary 상에 놓여질 때 naturally aligned 되는 것이다. 시작 주소가 natural

boundary가 되는 모든 데이터는 naturally aligned 된다. natural boundary에 aligned 되지 않은 데이터를 unaligned 데이터라고 한다.
데이터에 대한 접근이 효율적으로 이루어지도록 대부분 컴파일러는 가능한 모든 데이터를 naturally aligned한다.
Fortran에서 EQUIVALENCE문, common block의 사용, derived type이나 record structure 등의 사용으로 unaligned 데이터가 생길 수 있다. unaligned 데이터 발생을 막기 위해 다음과 같은 규칙을 따르는 것이 좋다.

- 항상 크기가 제일 큰 수치 데이터를 우선적으로 선언한다.
- 문자 변수와 수치 변수가 같이 사용되면 수치 변수를 먼저 선언한다.
- unaligned 데이터를 align시키기 위해 padding을 한다.

변수를 선언할 때는 INTEGER(4), REAL(4)와 같이 명시적으로 해당 변수의 크기를 지정하는 것이 좋다. 기본 크기를 갖도록 INTEGER, REAL과 같이 선언된 변수는 컴파일러 옵션에 의해 그 크기가 달라질 수 있고 이로 인해 데이터 alignment가 변경될 수 있다.
아래 코드를 컴파일하는 과정에서 컴파일러는 A를 align하기 위해 BOOL 뒤에 6바이트의 padding을 해야 하고 D를 align하기 위해 C 뒤에 4바이트의 padding을 해야 한다.

```
LOGICAL*2 BOOL
INTEGER*8 A, B
REAL*4 C
REAL*8 D
```

위 코드는 다음과 같이 데이터 선언 순서를 조절함으로써 모든 데이터에 대해 natural boundary alignment가 수행될 수 있다.

```
INTEGER*8 A, B
REAL*8 D
REAL*4 C
LOGICAL*2 BOOL
```

4.5. 프로세서, I/O, 라이브러리 관련 최적화

특정 마이크로프로세서 아키텍처와 관련된 최적화를 수행하기 위해서는 해당 아키텍처의 세부 특성을 알아야 할 필요가 있다. 일반적인 최적화 방안이 아닌 아키텍처의 지원여부에 의존하는 최적화 방법들로는 ILP(Instruction-Level Parallelism)를 최대한 이용하기 위한 파이프라이닝, 부동소수 연산 명령어, Instruction Set Extensions 등이 있다. 이들에 대해 알아보고 I/O 관련 최적화 방안, 고성능 연산 라이브러리 사용 등을 소개한다.

4.5.1. 파이프라이닝

최적의 성능을 얻기 위해 부동소수 파이프라인이 항상 채워져 연산이 진행되는 것이 좋다. 그러기 위해 독립적인 부동소수 연산이 컴파일된 루프에 충분히 있어야 한다.

파이프라이닝 처리를 하는 시스템이라면 파이프라인을 가장 효율적으로 사용하기 위해 컴파일러가 명령어 스케줄을 시도하게 된다. 다음 코드는 각각의 부동소수 연산을 수행하기 위해 이전 연산 결과를 필요로 하기 때문에 파이프라인 지연이 생겨 파이프라인 처리가 제대로 되지 않게 된다.

```
A = V(1)
A = A*V(2)
A = A*V(3)
A = A*V(4)
A = A*V(5)
A = A*V(6)
A = A*V(7)
A = A*V(8)
```

코드를 다음과 같이 수정하면 독립적인 연산이 이어지게 되어 파이프라인 지연을 줄일 수 있고 이로 인해 성능향상 효과를 볼 수 있다.

```
A = V(1)
B = V(2)
A = A*V(3)
B = B*V(4)
A = A*V(5)
B = B*V(6)
A = A*V(7)
B = B*V(8)
A = A*B
```

4.5.2. 부동소수 명령어

과학계산 프로그램은 대부분 부동소수 연산으로 구성되므로 프로세서의 부동소수 연산 처리 능력은 프로그램의 성능을 결정하는데 아주 중요한 역할을 한다. 부동소수 처리 성능을 높이기 위해 최신 프로세서들의 FPU(Floating-Point Unit) 디자인에 많은 특성이 추가되고 있다.

IEEE 754는 대부분의 최신 아키텍처들이 따르고 있는 부동소수 처리에 대한 표준이다. 보다 최적

화된 코드를 만들기 위해 IEEE 754에 대한 이해가 도움이 될 것이다.

4.5.2.1. Combined Multiply/Add Instructions: MADD, FMA

모든 프로세서 아키텍처는 프로세서 코어에 적어도 하나 이상의 FPU를 가진다. 이때 FPU는 하나로 곱셈과 덧셈을 모두 처리할 수 있는 것도 있고, 각각 덧셈과 곱셈을 처리하는 분리된 형태의 FPU로 있을 수도 있다.

IBM POWER4와 Itanium Family와 같은 일부 최신 아키텍처에서는 덧셈과 곱셈을 하나의 명령어로 한꺼번에 실행할 수 있는 FPU를 가진다. 한꺼번에 처리되는 덧셈/곱셈 패턴은 많은 과학계산 알고리즘의 기본이 되는 선형대수 계산에서 매우 흔하기 때문에 이를 지원하는 아키텍처는 그렇지 않은 아키텍처에 비해 현저한 성능 증가를 보인다.

MADD 연산 처리가 가능하고, 8개의 부동소수 레지스터가 있는 FPU 하나를 가지는 가상의 프로세서에서 다음과 같은 코드를 실행하는 경우를 살펴보자.

```
INTEGER I,N
PARAMETER (N=1000)
REAL X(N), A(N), B(N), C(N)
...
DO I=1,N
   X(I)=A(I)*B(I)+ C(I)
END DO
```

MADD 명령어를 사용하지 않는다면 컴파일러는 위 코드에 대해 아래와 유사한 명령어 흐름을 발생시킬 것이다.

```
top:    LD &a+ i,rf1          # load a(i) into FP register 1
        LD &b+ i,rf2          # load b(i) into FP register 2
        FMU rf1,rf2,rf3       # multiply the contents of FP registers 1 & 2
                              # and store the result in FP register 3
        LD &c+ i,rf4          # load c(i) into FP register 4
        FAD rf3,rf4,rf5       # add the contents of FP registers 3 & 4
                              # and store the result in FP register 5
        ST rf5,&x+ I          # store FP register 5 into x(i)
        INC i,1               # increment i by 1
        BLT i,n,top           # branch to top if i<n
```

한 번의 반복 계산 동안 10개의 명령어가 실행되고, 5개의 레지스터를 사용하고 있다. 그러나,

MADD 명령어를 사용한다면 다음과 같이 명령어 흐름이 간단해 진다. 루프 반복당 실행되는 명령어 개수가 9개, 사용되는 레지스터 개수도 4개가 돼 하나씩 줄어 들었다.

```
top:    LD &a+i,rf1          # load a(i) into FP register 1
        LD &b+i,rf2          # load b(i) into FP register 2
        LD &c+i,rf3          # load c(i) into FP register 3
        FMA rf1,rf2,rf3,rf4  # multiply the contents of FP registers 1
                             # and 2, add the contents of FP register 3,
                             # and store the result in FP register 4
        ST rf4,&x+i          # store FP register 5 into x(i)
        INC i,1              # increment i by 1
        BLT i,n,top          # branch to top if i<n
```

MADD 명령어를 지원하는 프로세서 아키텍처에서는 기본적으로 이 명령어를 사용한다. 만약 필요에 의해 MADD 명령어 사용을 막아야 한다면, 해당 아키텍처를 지원하는 컴파일러에서 제공하는 컴파일러 옵션을 이용해 MADD 명령어 사용을 금하도록 할 수 있다.

4.5.2.2. Division

나눗셈(division)은 덧셈이나 곱셈보다 실행시간이 훨씬 더 걸린다. 덧셈 또는 곱셈은 파이프라인으로 처리 가능하고 연산이 완료되기까지 보통 5~10 클럭 사이클이 소비되지만, 나눗셈은 파이프라인으로 처리 되지 않으며, 연산이 완료되기까지 20~60 사이클 정도가 소비 된다. 이러한 이유로 다음과 같이 루프 nest에서 inner 루프에 나눗셈이 들어가는 경우는 가급적 피하는 것이 좋다.

```
INTEGER I,N
PARAMETER (N=1000)
REAL X(N), A(N), B
DO I=1,N
   X(I)=A(I)/B
END DO
```

성능이 좋지 않은 나눗셈을 루프 nest에서 제거하기 위해 다음과 같이 역수를 이용한 변환이 가능하다.

```
INTEGER I,N
PARAMETER (N=1000)
REAL X(N), A(N), B, RB
```

```
RB = 1./B
DO I=1,N
   X(I)=A(I)*RB
END DO
```

이와 같은 변환이 컴파일러 수준에서 자동 처리되면 편리하겠으나, 부동소수 처리에 대한 IEEE 754 표준에서는 정밀도 손실 문제로 인해 컴파일러에 의한 자동 변환을 금지하고 있다. 그러나, 대부분의 컴파일러에서는 이와 같은 변환을 컴파일 단계에서 수행하도록 하는 옵션을 따로 제공하고 있다.

4.5.3. Instruction Set Extensions

한 프로세서 패밀리는 대부분 아키텍처를 공유한다. 프로세서 패밀리(인텔의 펜티엄, IBM의 POWER 등)가 개발되고 이후 발전해 가면서 처음 아키텍처의 명령어 집합에서 지원하지 않던 많은 특성과 성능이 새롭게 추가되게 된다. 이로 인해 명령어 집합을 더 확장하게 되는데, 이렇게 확장된 명령어를 사용하는 코드 생성을 위해서 컴파일러와 어셈블러의 기능 확장도 필요해진다.
확장된 명령어 집합의 예로 선인출(Prefetching)과 SIMD 명령어에 대해 소개한다.

4.5.3.1. 선인출

선인출은 Non-blocking load 명령어이다. 메모리 접근 지연을 감추기 위해 사용하는 선인출 명령어는 그 결과가 필요한 시점 이전에 미리 실행되어 결과가 필요하기 전에 완료 된다. 대부분의 아키텍처에서 선인출은 컴파일러 옵션 수준(대부분 -O2 또는 -O3)을 통해 처리되고 heuristics를 통해 얼마나 미리 선인출 명령어를 실행할 것인가를 결정한다. 너무 conservative한 선인출 heuristics는 선인출이 완료되기를 기다리는 의존적인 명령어 사용이 될 수 있으며 너무 aggressive한 경우 데이터가 너무 일찍 선인출 되어 사용도 되기 전에 캐시에서 삭제될 수 있다. 아키텍처가 가지는 선인출 기능을 효과적으로 활용하기 위해서 데이터의 선인출 흐름을 최적화 하도록 코드를 구성하는 것이 좋다. 이를 위해 다음과 같은 사항에 주의한다.

- 성능에 많은 영향을 주는 루프에서 흐름의 수가 너무 많거나 너무 적지 않게 한다.
- 성능에 많은 영향을 주는 루프에서 흐름의 길이를 너무 짧지 않게 한다.
- 가급적 실행순서가 예측 가능한 구조를 사용한다.
- 전역변수의 사용, aliasing, 강제 형 변환, 복잡한 제어 흐름 등을 최소화 한다.

IBM의 POWER4 프로세서에서는 선인출이 하드웨어를 통해 이루어 진다. POWER4 시스템에서 선인출은 루프 내에서 4 ~ 8개 정도의 흐름에 최적화 돼 있다. 선인출 흐름 수를 조절하기 위해 fusion이나 midpoint bisection 등의 기법을 사용할 수 있다.

```
DO I=1,N
```

```
    S = S + B(I) * A(I)
ENDDO
DO I=1,N
    R(I) = C(I) + D(I)
ENDDO
```

위의 코드에서와 같이 루프 내 흐름 수가 너무 적다면 fusion을 통해 흐름 수를 늘일 수 있다.

```
DO I=1,N
    S = S + B(I) * A(I)
    R(I) = C(I) + D(I)
ENDDO
```

루프의 중간지점을 양분(midpoint bisection)하는 방법을 통해서도 선인출 흐름의 수를 조절할 수 있다.

```
DO I=1,N
    S = S + B(I) * A(I)
ENDDO
```

내적 계산을 수행하는 위의 코드는 두 배열 A, B에 대해 두 개의 흐름이 존재한다. 다음과 같이 루프의 중간지점을 양분해서 흐름의 수를 두 배로 할 수 있다.

```
NHALF = N/2
S0=0.D0
S1=0.D0
DO I=1,NHALF
    S0 = S0 + A(I)*B(I)
    S1 = S1 + A(I+ NHALF)*B(I+ NHALF)
ENDDO
IF(2*NHALF.NE.N) S0 = S0 + A(N)*B(N)
S = S0+ S1
```

또한 여기서는 벡터의 길이가 반으로 줄어서 캐시 메모리 사용을 보다 최적화 하는 효과도 얻을 수 있다.

4.5.3.2. SIMD 명령어

Single Instruction Multiple Data 또는 short vector 명령어라고 한다. 벡터를 이용하여 상대적으로 길이가 짧은(2 또는 4) 연산수에 대해 동일 연산들을 동시에 수행하도록 한다. 전통적인 벡터형 슈퍼 컴퓨터의 경우 한꺼번에 연산을 수행할 수 있는 벡터의 길이가 길다(32 ~128 연산수에 대해 연산 가능). 대부분 SIMD 명령어 집합은 정수와 부동소수 연산을 모두 지원한다. 초기에는 이미지나 비디오 프로세싱과 같은 멀티미디어 응용 프로그램 지원을 위해 만들어져, 대부분이 64비트 배정도 실수 연산을 포함하고 있지 않다.

SIMD Instruction Set Extensions on Commodity Microprocessors						
SIMD Instruction Set	Processor Family	Integer?	32-bit FP?	64-bit FP?	Number of Vector Registers	Register Length (32-bit words)
MMX	Pentium and Pentium II	Y	N	N	8	2
SSE	Intel Pentium III*	Y	Y	N	8	4
SSE2	Intel Pentium 4	Y	Y	Y	8	2(64-bit)
3DNow!	AMD Athlon	Y	Y	N	8	2
AltiVec	Motorola 7410/Apple G4 and IBM PowerPC 970/Apple G5	Y	Y	N	32	4
* The Pentium 4 and Athlon XP also support SSE instructions.						

표 4-1. SIMD 명령어 집합

과학계산 프로그램에서 SIMD 명령어가 사용되는 어떻게 적용되는지 살펴보자.

```
INTEGER I,N
PARAMETER (N=1000)
REAL X(N), A(N), B(N), C(N)
...
DO I=1,N
   X(I)=A(I)*B(I)+ C(I)
END DO
```

Combined Multiply/Add Instructions에서 위의 코드가 전통적인 부동소수 연산을 이용해 어떻게 실행되는지 알아 보았었다. 8개의 벡터 레지스터를 가지는 SIMD Unit을 이용할 때, 위의 코드에 대한 어셈블러 코드는 다음과 같은 모양으로 처리된다. .

```
top:    VLD &a+i,rv1          #load a(i:i+vlen) into vector register 1
```

```
VLD &b+ i,rv2          #load b(i:i+ vlen) into vector register 2
VFM rv1,rv2,rv3        #multiply the contents of vector registers 1 & 2
                       #and store the result in vector register 3
VLD &c+ i,rv4          #load c(i:i+ vlen) into vector register 4
VFA rv3,rv4,rv5        #add the contents of vector registers 3 & 4
                       #and store the result in vector register 5
VST rv5,&x+ i          # store vector register 5 into x
```

여기서 vlen이 SIMD unit의 벡터 길이를 나타낸다. 위의 연산은 SIMD unit에서 완전히 처리되므로 스칼라 정수와 스칼라 부동소수 처리 unit은 동시에 다른 코드를 실행하는 것이 가능해 진다. SIMD 명령어의 사용 여부는 컴파일 단계에서 옵션을 이용해 지정할 수 있다.

4.5.4. Efficient I/O

프로그램에서 입출력 부분은 많은 시간을 소비하는 곳이다. 그래서 입출력이 많은 코드는 전반적으로 성능이 좋지 못하다. 전체 코드의 실행 시간을 줄이기 위해 가능하다면 코드 내의 입출력을 줄여야 하고 꼭 입출력을 해야 한다면 가급적 성능이 좋은 입출력 방법을 택하도록 해야 한다. 여기서는 효율적인 입출력 처리를 위한 몇 가지 방법들을 소개한다.

① 가능한 unformatted 파일을 사용할 것

수치 데이터는 unformatted I/O가 formatted 보다 더 효율적이고 정밀하다. 데이터를 formatted 파일에 쓸 때 formatted 데이터는 출력을 위해 character로 전환돼야 한다. unformatted 파일은 binary로 저장되고 formatted 파일은 ASCII로 저장되는데, formatted 데이터를 다시 binary로 읽을 경우 정밀도 손실이 생길 수 있다. formatted 데이터가 보다 쉽게 다른 시스템으로 이식될 수 있지만, 대부분 컴파일러들은 서로 다른 naive formats 사이의 전환을 지원한다.

```
OPEN(UNIT=1, FILE='data.out', STATUS='NEW', FORM='UNFORMATTED')
```

Fortran의 기본 파일접근 방식은 sequential access 이고 이때 formatted 연결이 기본이므로 위와 같이 FORM='UNFORMATTED'로 명시해 unformatted 파일을 사용하도록 한다.

② Direct Access 파일 사용

Fortran에서 외부 파일 접근 방식은 Direct 접근과 Sequential 접근 두 가지가 있다. Sequential 접근은 저장한 순서대로 자료 값에 접근하는 파일 단위의 접근 방식을 이용하고, Direct 접근은 해당 레코드로 직접 연결하여 필요한 자료 값을 저장하고 읽는 레코드 단위 접근을 이용한다. 기본 Sequential 파일 접근을 사용하지만, 접근 속도 면에서는 Direct 접근이 더 빠르다.

③ 각 원소 단위로 쓰지 말고 배열, string 전체로 쓸 것

불필요한 overhead를 줄이기 위해, 각 원소를 여러 번에 걸쳐 쓰지 말고 한번에 배열 전체나 string을 쓰는 것이 좋다. 이것은 I/O 리스트의 각 아이템이 자신의 호출 sequence를 발생시키기 때문이다. 같은 이유로 Fortran 90에서 전체 배열에 접근할 때 implied DO 루프 대신 배열 이름을 가지고 접근하는 것이 좋다.

배열 A(N,N)을 파일에 쓰는 경우를 살펴보자.

Case 1. Best. N*N개 값의 1개 레코드

```
WRITE(1) A
```

Case 2. N개 값의 N개 레코드

```
DO I = 1, N
   WRITE(1) (A(J,I), J = 1, N)
ENDDO
```

Case 3. Worst. 1개 값의 N*N개 레코드

```
DO I = 1, N
   DO J = 1, N
      WRITE(1) A(J,I)
   ENDDO
ENDDO
```

④ natural storage order 순으로 배열 데이터를 쓸 것

natural order(Fortran의 경우 열우선 순)를 사용할 수 없다면, I/O 수행하기 전에 메모리 내에서 reorder를 하는 것이 보다 효율적이다.

⑤ 가능하면 buffered I/O 사용

Fortran run-time 시스템에서는 기본적으로 unbuffered disk 쓰기를 한다. 이것은 나중에 디스크에 쓰도록 buffer에 저장하지 않고 각 record의 쓰기 때마다 즉시 디스크에 쓰기를 하는 것이다. buffered 쓰기를 사용하면 큰 데이터 블록을 디스크에 쓰기 하는 회수가 줄기 때문에 보다 효율적인 디스크 I/O가 된다. 문제는 buffered 쓰기의 경우, 아직 디스크에 쓰기 전에 시스템 오류가 발생하면 데이터를 모두 잃어 버릴 가능성도 있다는 점이다.

⑥ MPI 코드에서는 가능한 MPI I/O 사용

4.5.5. Large Page Sizes

OS가 지원한다면 메모리 접근이 많은 코드의 TLB미스를 줄여 성능을 높이기 위해 가급적 크기가 큰 페이지를 사용하는 것이 좋다. SGI의 IRIX 시스템에서 MIPSpro 컴파일러에는 환경변수 PAGESIZE_DATA, PAGESIZE_STACK, PAGE_SIZE_TEXT 설정과 함께 사용되는 -bigp_on 옵션이 있다. 페이지 크기를 16, 256, 1024, 4096, 16384KB로 설정 가능하다. IBM의 AIX에서도 기본 4KB인 페이지 크기를 16MB까지 지원하고 있다.

4.6. Mathematical Library

새로운 마이크로 프로세서 칩을 장착한 슈퍼 컴퓨터가 나올 경우, 이를 지원하는 OS, chip-specific 컴파일러, device drivers 등과 같은 새로운 소프트웨어가 같이 나오게 된다. 또한 새로운 software에는 일반적인 수학 라이브러리 루틴에 대한 chip-optimized 라이브러리가 포함된다. 프로그래머는 해당 시스템에 최적화된 이러한 라이브러리를 최대한 사용하는 것이 좋다.
시스템 최적화된 라이브러리 이외에 특정 시스템에 의존하지 않는 많은 '전통적인' 수학 라이브러리가 있다. 설치와 이용하기가 쉽고 다양한 영역의 수학 문제를 처리하는 많은 수의 루틴을 가지며 일부 라이브러리는 free로 제공되기도 한다.

4.6.1. Vendor-Supplied Mathematical libraries

대부분의 시스템 업체에서는 해당 프로세서 칩에 최적화된 수학 라이브러리를 제공하고 있으나, 모든 분야에 대한 것은 아니며, 상대적으로 많이 사용되는 수학 분야의 문제에 국한된다.

4.6.1.1. Calculator Functions

trigonometric, log와 exponential, hyperbolic, exponential, factorial 같은 기본적인 수학 함수들이다. 사용 가능한 calculator 함수는 보통 vendor-optimized된 normal 라이브러리와 특별히 높은 수준으로 최적화된 special 라이브러리의 두 가지 버전이 있다. 대개의 경우, 높은 수준으로 최적화된 special 함수들은 그 수가 작으며, 주로 trigonometric, log와 exponent 함수 정도이다. 또, 이러한 special 함수들은 수치 연산에 대한 IEEE의 규칙을 따르지 않을 수 있으므로, 이러한 루틴을 사용하기 전에 noieee와 같은 컴파일러 옵션을 확인 후 사용하도록 해야 한다. special 함수를 사용한 프로그램에서 정밀도 손실 문제는 프로그래머의 경험과 테스트에 의해 판단되어야 한다.

4.6.1.2. Linear Algebra

자주 사용되는 선형대수 라이브러리들이 업체에서 제공하는 수학 라이브러리에 종종 포함된다. 선형대수 라이브러리로 대표적인 BLAS 라이브러리는 벡터-벡터(BLAS1), 행렬-벡터(BLAS2), 행렬-행렬(BLAS3) 계산을 수행하는 루틴들로 구성돼 있다. 업체에서는 시스템에 최적화시킨 선형대수 계산 라이브러리의 루틴 이름을 BLAS 루틴의 이름과 동일하게 제공하는 방식으로 프로그래

머에게 최적화된 수학 라이브러리 환경을 제공하고 있다. 예를 들자면, BLAS2에서 제공하는 행렬-벡터 곱셈 루틴 이름이 SGEMM(Single precision General Matrix Multiply)인데 업체에서 제공하는 최적화된 수학 라이브러리에서도 같은 계산을 하는 루틴의 이름을 SGEMM으로 사용하는 것이다.

LAPACK은 BLAS 루틴을 기반으로 개발된 선형대수 계산 라이브러리로 선형 방정식, 고유치(eigenvalue)와 고유벡터(eigenvector) 문제, 선형 최소자승 맞춤(linear least square fitting) 등의 문제해결에 관련된 루틴을 제공하고 있다. 업체에서 제공하는 라이브러리는 LAPACK 라이브러리를 포함하고 있는 것이 대부분이다.

4.6.1.3. Fast Fourier Transforms

FFT 루틴들은 대부분의 계산 수학 라이브러리에 포함되며, 1차원, 2차원 FFT, complex variable FFT 등 다양한 루틴들이 존재한다. 업체에서 제공하는 라이브러리에 대부분 FFT 계산 루틴이 포함돼 있다.

4.6.1.4. Parallel Numerical Libraries

지난 수년 동안, 많은 수학 라이브러리들의 병렬화 작업이 시도돼 왔으며, 그 중 BLAS와 LAPACK의 병렬화 작업이 가장 먼저 이뤄졌다. BLAS의 병렬 버전은 PBLAS, LAPACK의 병렬 버전은 ScaLAPACK으로 안정화 됐으며 이에 따라, 최근의 업체 제공 라이브러리에는 대부분 PBLAS와 ScaLAPACK이 포함돼 있다.

PBLAS와 ScaLAPACK의 기능은 BLAS와 LAPACK과 대부분 유사하며 프로그래머가 글로벌 배열 데이터를 로컬 부분 배열로 나눠 개별 프로세서의 로컬 메모리로 분배하도록 하는 기능과 해당 계산을 병렬로 실행할 수 있는 기능을 추가적으로 가지고 있다. 이와 같은 병렬 라이브러리에서는 내부적으로 데이터 통신이 필수적이고 이를 위해 또 다른 라이브러리인 BLACS를 사용하게 된다. BLACS는 해당 병렬 시스템에서 사용하는 메시지 패싱 라이브러리(MPI, PVM 등)를 기반으로 하고, 특히 배열의 전송에 그 기능을 맞춰 개발된 데이터 통신 라이브러리이다.

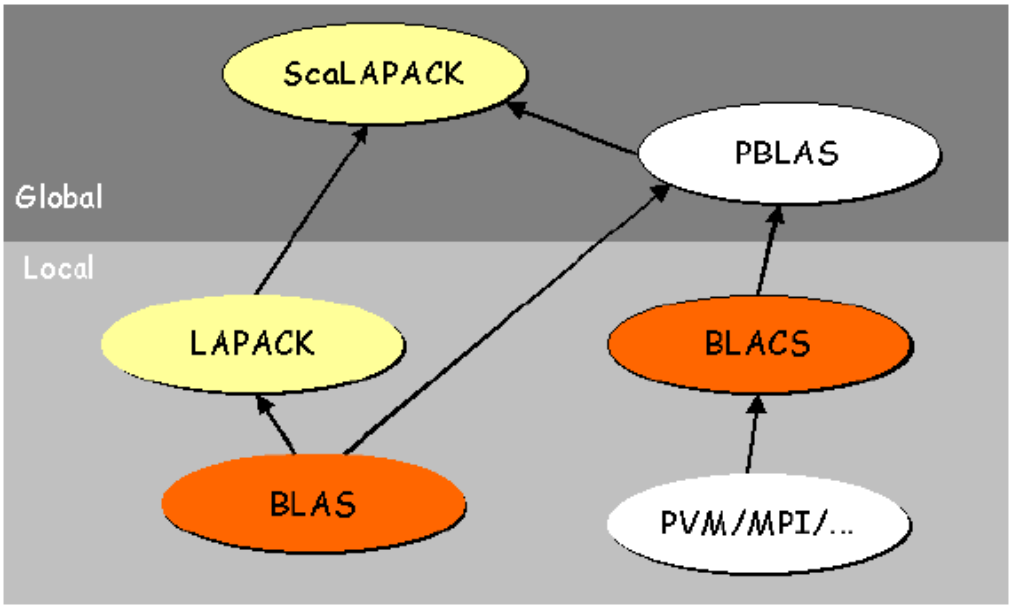


그림 4-6. 선형대수 계산 라이브러리

4.6.1.5. Code Optimization with Library Calls

코드 성능 최적화를 언제나 강조되는 것은 프로그램을 실행하고자 하는 시스템 업체에서 제공하는 라이브러리를 사용하라는 것이다. 행렬-행렬 곱셈과 같은 계산을 직접 프로그래밍하는 것 보다는 관련 라이브러리를 호출해 사용하는 것이 코드도 간단하게 하면서 코드 성능도 좋게 하는 지름길이 된다. 물론 관련 라이브러리가 해당 업체에서 제공하는 최적화된 라이브러리라면 더욱 좋을 것이다.

일부 매우 잘 개발된 컴파일 시스템에서는 '라이브러리 루틴 대체' 옵션을 제공하는 경우도 있다. 이는 컴파일러가 프로그래머가 직접 작성한 수학 계산 코드를 분석해 동일한 계산을 하는 업체 제공 라이브러리 루틴으로 컴파일 단계에서 대체해 버리는 기능이다.

4.6.2. Traditional Math Libraries

NAG와 IMSL 라이브러리는 전통적인 수학 라이브러리를 대표한다. 이 두 개의 라이브러리는 수학 및 통계 분야의 광범위한 문제를 다루고 있다.

- 프로그램 안에서 어떤 수학 계산을 하고자 한다면, 대부분 NAG나 IMSL 라이브러리에서 그에 해당하는 루틴을 찾을 수 있다. 다음은 NAG에서 다루고 있는 수학 계산들을 정리해 놓은 것이다.

Sample of NAG Mathematical Library Routines

- Complex Arithmetic
- Zeros of Polynomials m Roots of Transcendental Equations
- Extremes (Min/Max) of a function
- Matrix Factorizations m Eigenvalues & Eigenvectors

- Fourier Transforms (1D, 2D, and 3D)
- Quadrature Numerical Integration
- Ordinary Differential Equations m Partial Differential Equations
- Mesh Generation
- Interpolation m Curve and Surface Fitting
- Random Number Generation
- Time Series Analysis
- Mathematical Constants
- Utility Functions
- Determinants
- Simultaneous Linear Equations
- Linear Algebra m Sparse Linear Algebra
- Statistical Analysis
- Correlation & Regression m Multivariate Methods m Univariate Estimation
- Contingency/Survival Analysis
- Sorting
- Special Functions

- NAG와 IMSL 라이브러리는 사용 알고리즘과 루틴의 각 인수들, 발생하는 오류 코드, 해당 루틴을 사용하는 간단한 예제 프로그램과 그 프로그램의 결과 등에 대해 상세히 설명된 documents를 제공하고 있다.
- 루틴들에 대한 병렬화 작업이 계속 이루어 지고 있다.
- 새로운 수학 문제에 대한 루틴이 계속 추가되고 있다.
- 다양한 시스템, 운영체제, 프로세서 등에서 사용할 수 있다. 단, 그 모든 시스템에 최적화된 성능을 보장하지는 못한다.
- NAG와 IMSSL 라이브러리의 단점은 고가의 라이센스 및 유지보수 비용.

5. 병렬 프로그램 성능 최적화

병렬 프로그램의 성능을 위해 가장 중요한 것은 순차 프로그램의 성능이다. 이와 더불어 병렬로 수행되는 알고리즘에 대한 최적화 작업이 필요하게 되는데 통신, 동기화, 로드 밸런싱 등에 대해 특별히 신경 쓰도록 해야 한다.

5.1. 병렬 프로그램 성능과 확장성

Amdahl의 법칙은 병렬 프로그램의 성능 한계를 이론적으로 보여 준다. 대부분의 병렬 프로그램은 최적의 프로세서 개수가 있으며, 그 개수를 넘겨 사용하게 되면 오히려 성능이 더 나빠지게 되는데 이러한 확장성 문제는 병렬 시스템이 가지는 하드웨어적인 요소와 프로그램의 알고리즘이 가지는 소프트웨어적 요소로 나눠볼 수 있다.

병렬 프로그램의 부하는 프로그램 실행 소스가 여러 개이기 때문에 발생한다. 가장 나쁜 병렬 부하는 일부 또는 모든 프로세서들이 한 프로세서가 작업을 모두 완료할 때까지 대기하는 순차화(serialization)이다. 순차화는 동기화에 의해 발생하거나, 단일 프로세서가 입력파일을 읽어 broadcast하는 상황 등에서 발생할 수 있다. 또한 인접한 이웃과 경계 데이터를 교환하는 통신의 경우 순차화(교착이 발생할 수도 있음) 될 수 있으므로 비동기 통신을 이용해 주의 깊게 처리돼야 한다.

분산 메모리 시스템에서 특히 중요한 병렬 부하로 통신 지연이 있다. 네트워크가 얼마나 빠른가에 상관없이 한 단위의 데이터를 전송하는데 일정시간이 소비돼야 한다. 이 시간을 통신 지연이라고 하며, 이것은 해를 얻기 위해 소비되는 계산 시간은 아니다.

병렬 프로그램의 성능 최적화를 위해 우선적으로 중요한 것은 순차코드의 최적화이다. 그리고 순차 프로그램에서 병렬화 가능한 부분을 가급적 증가시키는 최적화가 필요하며, 병렬화 과정에서의 로드 밸런싱과 통신 및 동기화 과정에서의 소비시간을 최소화 시키는 노력이 필요하다.

5.1.1. 분산 메모리 프로그래밍 모델

분산 메모리 시스템상에서 실행할 코드를 개발하고 최적화 할 때는 분산 메모리 시스템상의 프로그래밍 특성을 고려해야 한다.

분산 메모리 시스템에서 우선적으로 중요한 것은 특정 프로세스가 다른 프로세스의 메모리 공간으로 접근이 불가능하다는 것이다. 때문에 분산 메모리 시스템 상에서 원격 메모리에 대한 접근은 시스템의 표준 메모리 참조 시스템에 의해서가 아닌, 다른 메커니즘을 이용해, 프로그래머에 의해 명시적으로 시작되어야 한다. 이러한 메커니즘으로 메시지 패싱과 분산 공유 메모리(Distributed Shared Memory)를 가장 많이 사용한다. 이 메커니즘들은 하드웨어와 소프트웨어의 조합에 의해 수행되는데, 그 상세한 구현 방식에 따라 병렬코드의 성능에 큰 영향을 줄 수 있다.

5.1.1.1. Message Passing

원격 메모리 접근을 위해 가장 많이 사용된다. 현재의 연구된 네트워크 기술과 하드웨어를 바탕으로 구현되고, 의도되지 않은 data sharing 및 false sharing이 발생하지 않는다. 그리고 다양한 하드웨어 영역에서 좋은 성능을 보여준다.

메시지 패싱을 이용한 데이터 전송은 다양한 형태를 갖는다.

- two-sided point-to-point 전송: send, receive는 blocking 또는 non-blocking 가능
- one-sided point-to-point 전송: 하나의 프로세스가 다른 프로세스의 메모리 타깃을 명시적으로 포함시켜 해당 메모리 위치를 put 또는 get 하는 것이 가능.
- collective operations: 다수의 프로세스가 동시에 서로 데이터를 전송

MPI, PVM, CORBA 등이 있으며 MPI를 가장 많이 사용하고 있다.

5.1.1.2. Distributed Shared Memory(DSM)

분산 메모리 시스템 노드의 로컬 메모리에 메모리 위치를 대응시킨 가상 메모리 주소 공간을 제공하는 것이다. 병렬 프로그램을 처음 접하는 프로그래머 입장에서 프로그램 작성이 매우 간단해 보일 수 있지만 user-space-accessible DMA 엔진이 없는 네트워크 하드웨어 환경인 경우 성능 좋은 프로그램을 구현하는 것이 매우 어렵다. 대부분의 NUMA 아키텍처들이 하드웨어와 운영체제 기반으로 DSM을 지원한다.

클러스터의 경우 DSM 인터페이스는 주로 소프트웨어 기반이 된다. Cray의 shmem, Pacific Northwest Laboratory의 Global Array Toolkit 등과 같은 DSM API를 많이 사용하며, one-sided 메시지 패싱과 같이 동작하는 명시적인 put, get 연산을 제공한다.

5.1.2. 병렬 성능과 확장성의 정량화

실제 성능향상도 S(n)은 Amdahl의 법칙을 따르지 않는다. 프로세서 개수가 작을 때는 S(n)=n의 이상적 성능향상도에 가깝다가, 이후 급격히 감소된다. 이것은 프로세스에서 계산을 위해 소비된 시간보다 더 많은 시간이 통신에 소비되기 때문이다.

경우에 따라서 측정된 성능향상도가 S(n)=n을 초과하는 경우도 있는데 이를 superlinear speedup 이라고 한다. 상황에 따라 그 원인을 확실하게 밝히는 것이 어렵지만, 대부분은 캐시 효과 때문이다. 프로세스에서 사용되는 메모리 양이 프로세서의 outermost 캐시 크기의 정수배 정도로 줄면 캐시 사용률이 거의 100%가 되고 이때 메인 메모리의 속도는 성능에 영향을 주지 않게 된다.

성능향상도를 프로세서 개수로 나눈 병렬효율도 자주 사용된다. 병렬효율은 순차 프로세스와 비교해 각 병렬 프로세스가 평균적으로 어느 정도의 성능을 발휘했는가를 나타내는 데, 이상적인 수치는 1이다. 대부분 1보다 작은 값을 가지며 만약, superlinear의 경우라면, 1보다 큰 값을 보일 것이다.

5.2. 병렬 프로그램의 병목 지점

성능을 저하시키는 병목 지점을 찾는 것이 복잡하고 어려운 일이지만, 병렬 프로그램 최적화 과정에서 순차화와 불필요한 통신 등에 의해 야기된 문제들을 찾아내는 것은 반드시 필요하다.

5.2.1. 순차화

코드 확장성에 가장 나쁜 영향을 주는 것은 하나의 프로세스가 작업을 완료할 때까지 많은 프로세스들이 기다리게 되는 순차화이다.

- MPI_Barrier 등과 같은 동기화 루틴을 포함한 집합 통신의 과도한 사용.
- 하나의 프로세스가 입력 파일을 읽어 들이고, 다른 프로세스들은 그것을 기다리는 상황과 같은 단일 프로세스 I/O의 사용 → 병렬 I/O 사용을 권장
- blocking 점대점 통신의 사용 → non-blocking 점대점 통신의 사용 권잗

5.2.2. 불필요한 통신

통신은 계산보다 훨씬 많은 시간을 소비한다. 1GHz 클럭 속도를 가지는 프로세서의 경우 4사이클이 소비되는 부동소수 연산은 4ns에 완료되지만, Quadrics(3.2Gbit/s link speed, 250~350 MB/s 대역폭, 2~5μs latency를 가지는 네트워크 장비)상에서 크기가 0인 데이터를 MPI_Send()를 이용해 전송할 경우 최소 소프트웨어 latency는 약 4μs 정도가 소비된다.

- 작은 데이터를 여러 번 통신하는 경우 가능하다면 한 번의 큰 데이터 통신으로 대체하는 것이 좋다.
- 집합 통신의 과도한 사용을 자제한다. 예로, Reduce로 충분한 곳을 Allreduce를 사용하지 말 것.

5.3. 통신성능

메시지 패싱 프로그램에서 통신성능은 전체 프로그램의 성능에 영향을 준다. 통신성능을 분석하는 방법과 통신성능 향상을 위한 MPI 특성 이용 방법 등에 대해 알아본다.

5.3.1. 통신성능에 영향을 주는 요인들

통신성능에 영향을 주는 요인들은 다음과 같이 매우 다양하다.

- 플랫폼/아키텍처 관련 요인: 네트워크 어댑터의 type, latency와 대역폭. Multithreading의 사용과 interrupt handling과 같은 OS 특성
- 네트워크 관련 요인: 네트워킹 하드웨어(Ethernet, FDDI, switch, router 등), 프로토콜, 네트워크에서 routing과 configuration, network contention과 saturation 등
- 어플리케이션 관련 요인: 알고리즘 효율성과 확장성, 통신 to 통신 ratios, 로드 밸런스, MPI 루틴의 type과 메시지 크기 등.

- MPI 구현 관련 요인: 메시지 버퍼링 방법, 메시지 패싱을 구현하는데 사용되는 프로토콜, sender-receiver 동기화 type 등 → 여러 MPI 루틴을 구현하는데 사용하는 알고리즘의 효율성은 MPI 구현마다 다를 수 있으며 프로그램 성능에도 영향을 끼친다.
- SMP 클러스터 specific 요인들: SMP 클러스터 환경에서는 하나의 SMP 노드에서 실행되는 MPI 작업의 수가 통신 성능에 영향을 준다. 일반적으로 한 노드에서 실행되는 MPI 작업이 하나보다 많아지게 되면 작업당 통신 대역폭이 감소한다. aggregate 대역폭은 saturation에 도달할 때까지 MPI 작업의 수에 따라 증가한다. saturation된 후 steady 상태이거나 혹은 감소하게 된다. intra-node 통신에 대해 shared memory를 사용하면 한 노드에서 실행되는 작업의 수가 증가함에 따라 작업당 통신 대역폭이 감소하는 것을 최소화 할 수 있다.

5.3.2. 메시지 버퍼링

메시지 버퍼링은 송신의 시작과 그에 대응되는 수신의 완료 사이에서 데이터를 저장해 두는 것을 말한다. 버퍼링은 시스템에 의해서 또는 유저에 의해 수행될 수 있다. 시스템 버퍼링은 유저에게 노출되지 않으며 그 구동 방식은 implementation 의존적이다. MPI 표준은 시스템 버퍼링이 어떤 방식으로 구현되는가에 대해 의도적으로 정의하지 않고 있다. 유저가 제공하는 버퍼 공간은 명시적으로 선언되고 사용자 프로그램에 의해 운영된다. 송신에 대한 버퍼링 구현은 다음과 같은 방식이 있으며, 성능에 다르게 영향을 준다.

- 송신측 버퍼링
- 수신측 버퍼링
- 버퍼링 없음
- 일정 조건이 만족될 때 버퍼링 (eager, rendezvous 프로토콜)

시스템 버퍼링의 장점은 비동기 통신을 통하여 성능을 개선시킬 수 있다는 것이다. 시스템 buffered 송신의 경우 대응되는 수신 연산이 포스트(post) 되지 않은 경우에도 기다리지 않고 송신 작업을 수행할 수 있다.

버퍼링의 단점은 안정성(robustness)에 있다. 버퍼가 고갈(depletion)되면 프로그램이 다운(fail)되거나 지연될 수 있다. 버퍼링이 되는지 된다면 어떻게 구현되는지 MPI 구현(implementation)마다 다르기 때문에 이쪽 시스템에서 문제없는 코드라도 다른 시스템에서는 그렇지 않을 수 있다. 이와 같은 의존성을 가지는 코드는 대부분의 시간을 제대로 실행해서 바른 결과를 내놓더라도 불안정하다.

5.3.3. MPI 메시지 패싱 프로토콜

MPI 메시지 패싱 프로토콜은 MPI implementation이 메시지 전달을 위해 사용하는 정책과 내부 방식을 기술한다. 프로토콜은 MPI 표준에서 정의되지 않는다. MPI implementation은 한 MPI 루틴에 대해서 몇 가지 프로토콜의 조합을 사용할 수 있다. 예로 표준 송신은 작은 메시지의 경우 eager 프로토콜을 큰 메시지에 대해서는 rendezvous 프로토콜을 사용할 수 있다. MPI 메시지 패

싱 프로토콜은 종종 메시지 버퍼링과 함께 동작한다.

5.3.3.1. Eager Protocol

비동기 프로토콜. eager 프로토콜은 송신 프로세스가 메시지를 보내면 수신 프로세스가 메시지를 저장할 수 있다는 가정하에 작동한다. 수신 프로세스가 포스트 되지 않은 상태라면 메시지는 버퍼에 저장된다. 이 가정에 의해 MPI 구현은 수신 프로세스를 위한 일정량의 가용 버퍼 공간을 확보해 두어야 한다. eager 프로토콜은 일반적으로 메시지 크기가 작을 때 사용(이때 메시지 크기는 MPI 작업 수에 의해 제한됨)한다.

eager 프로토콜의 장점은 동기화 지체의 감소에 있다. 즉, 송신 프로세스는 메시지 송신을 위해 수신 프로세스로부터의 OK 신호를 기다릴 필요가 없다. eager 프로토콜은 프로그래머가 MPI_Send만 이용해 비동기 통신의 효과를 누릴 수(사실은 구현에 의존적이지만) 있으므로 프로그래밍이 단순하다.

eager 프로토콜의 단점은 scalable 하지 않다는 것이다. 임의의 수의 송신프로세스로부터 메시지를 받기 위해 많은 버퍼링 공간이 필요하며, 버퍼공간이 초과되면 메모리 exhaustion과 프로그램 termination이 발생할 수 있다. 그리고, 소수의 메시지가 보내진다면 버퍼 공간이 낭비될 수도 있다. 메시지를 네트워크에서 끌어와서 버퍼 공간에 데이터를 복사하므로 수신 프로세스 측면에서 보면 eager 프로토콜이 CPU에 부하를 줄 수도 있다.

5.3.3.2. Rendezvous Protocol

동기 프로토콜. 수신 프로세스의 버퍼 공간이 만들어 질 수 없을 때, eager 프로토콜의 버퍼 한계가 초과된 경우 사용한다. 송신, 수신 프로세스 사이에 동기화(handshaking)이 필요하며 아래와 같은 과정을 거쳐 통신이 이루어지게 된다.

- 송신 프로세스는 수신 프로세스로 메시지 봉투 정보를 보낸다.
- 수신 프로세스에 봉투 정보가 수신되고 저장된다.
- 버퍼 공간 사용 가능하면, 수신 프로세스는 송신 프로세스에게 응답한다.
- 송신 프로세스는 수신 프로세스로부터 응답을 받고 데이터를 보낸다.
- 수신 프로세스는 데이터를 받는다.

Rendezvous 프로토콜은 eager 프로토콜보다 scalable하고 안정적이다. 수신 측에서는 작은 메시지 봉투 정보를 저장할 버퍼만 필요하게 된다. 송신 프로세스 user space에서 수신 프로세스 user space로 직접 복사를 통해 데이터 전송이 이뤄지므로 데이터를 복사하는 부하가 eager 보다 줄어든다.

송신과 수신 사이에 동기화(handshaking)가 필요하므로 동기화 지체가 발생하는 것이 단점이다. 수신 프로세스로부터의 신호를 기다리는 동안 블로킹 되는 것을 피하기 위해 논-블로킹을 사용하면 프로그래밍 복잡성이 증가하게 된다.

5.3.4. Sender-Receiver 동기화

rendezvous 프로토콜을 사용하는 경우와 같이, 동기 MPI 통신은 송신 수신 작업 사이에 동기화가 필요하다. MPI 표준은 이와 같은 동기화가 어떻게 구현돼야 하는가를 정의하지 않는다. MPI implementor에게는 다음과 같은 의문이 남겨진다.

- 수신 프로세스는 다른 프로세스에서 송신을 하려 한다는 것을 어떻게 알 수 있을까?
- 수신 프로세스는 얼마나 자주 수신 메시지를 체크해야 할까?
- 논-블로킹 송신이 시작된 후, 프로세스는 송신 작업을 위해 얼마나 많은 CPU 사이클 시간을 할당해야 할까?

MPI 구현은 대부분 송-수신 동기화를 위해 polling과 interrupt, 두 가지 다른 모드를 사용한다. polling 모드에서 사용자의 MPI 작업은 일정한 간격(implementation defined)을 두고 통신 이벤트 서비스와 check를 위해 시스템에 의해 주기적으로 interrupted 된다. user task가 다른 작업을 하느라 바쁜 경우 통신 이벤트가 발생하면, 기다려야 한다. interrupt 모드에서 사용자의 MPI 작업은 통신 이벤트가 발생했을 때 시스템에 의해 interrupted 된다. 다음과 같은 특성을 가지는 프로그램은 interrupt 모드를 사용할 경우 성능 개선 효과가 있다.

- non-blocking 송-수신을 하는 프로그램
- 동기화 되지 않은 송-수신 pair를 가지는 프로그램. 예, 대응되는 수신에 대하여 다른 시간에 송신 루틴 호출되는 경우.
- non-blocking 송수신에서 송신 수신 이후 바로 wait를 호출하지 않고 이에 앞서 다른 계산을 하는 경우.

5.3.5. 메시지 크기

메시지 크기는 MPI 프로그램 성능에 큰 영향을 주는 요인 중 하나이다. 대부분의 경우 메시지 크기를 증가시키면 성능이 나아진다. 통신이 많은 프로그램에서는 “규모의 경제”식의 메시지 크기 장점을 살리도록 알고리즘을 개선하는 것이 필요하다.

5.3.6. 점대점 통신

블로킹, 논-블로킹, persistent 통신 등으로 분류할 수 있다. 점대점 통신의 성능은 어떤 루틴이 사용됐는지 어떻게 구현 되었는가에 따라 달라진다. 일반적으로 논-블로킹 통신이 성능이 우수하다. persistent 통신은 같은 인수를 가지는 메시지 패싱 루틴이 반복적으로 호출되는 프로그램에서 통신 부하를 감소시키기 위해 유용하게 사용. persistent 루틴은 논-블로킹 이다.

5.3.7. 집합통신

커뮤니케이터내의 모든 프로세스가 참여해야 하는 통신이다. 작업 사이의 동기화와 로드 밸런싱은 코드 성능에 중요한 영향을 준다. 동일한 통신을 점대점을 이용해 구현하는 것이 언제나 가능하다. MPI-1에서는 블로킹 집합통신 루틴만 정의돼 있으며, MPI-2에서 논-블로킹 집합통신을 정의 하고 있다.

MPI 표준에서는 어떻게 집합통신이 구현되어야 하는가에 대한 정의가 없다. 구현 알고리즘은 사용자에게 감춰지는데, 중요한 프로그램이라면, hand-coded 방법과 업체가 제공하는 집합통신 루틴을 사용한 경우를 비교해 보는 것도 좋다. 또한 동일 플랫폼서 다른 MPI 구현을 비교하거나, 다른 플랫폼에서 다른 MPI 구현을 비교하는 것도 좋다.

5.3.8. 네트워크 contention

MPI 작업들 사이에서 통신되는 데이터의 양이 네트워크 대역폭에 가까워질 때 발생하며, 모든 작업에 대해 통신 성능을 저하시킨다.

6. 부록: 성능 분석 도구

6.1. HPM Toolkit

HPM(Hardware Performance Monitor) Toolkit은 물리적인 프로세서내의 하드웨어 이벤트 카운터에서 사용자가 필요한 정보를 얻을 수 있도록 하는 라이브러리와 유틸리티의 모음으로 IBM에서 제공하고 있다. HPM Toolkit은 hpmcount, libhpm, hpmviz 등으로 구성되어 있으며 다음과 같은 다양한 하드웨어 이벤트들을 측정할 수 있다.

① clock cycles
② instructions completed
③ L1 load/store misses
④ L2 load/store misses
⑤ TLB misses
⑥ FPU/FXU activity
⑦ number of branches
⑧ branch mispredictions
⑨ loads/stores completed
⑩ FMAs executed

POWER4 프로세서에서는 POWER3 프로세서에서와 마찬가지로 하드웨어 이벤트 카운터에서 동시에 8가지의 정보를 측정할 수 있다. IBM에서는 사용자가 사용하기에 편리하도록 하드웨어 이벤트 카운터 정보를 8개씩 묶어 0부터 60까지 총 61개의 그룹을 제공하고 있으며(POWER3의 경우는 4개의 그룹 제공) 디폴트로 사용되는 그룹은 60번이다. 각 그룹에서 다루는 정보들에 대해서는 IBM AIX 시스템의 /usr/pmapi/lib/POWER4.gps에 그 내용이 있다.
사용자가 프로그램의 성능분석에 자주 이용하는 정보를 담은 몇 개의 그룹들을 소개하면 다음과 같다.

① 그룹 60 : cycles, instructions, FP 연산(나누기, FMA, 로드, 저장 포함) 회수 등
② 그룹 59 : cycles, instructions, TLB 미스, 로드, 저장, L1 미스 회수 등
③ 그룹 5 : L2, L3, 그리고 메모리로부터의 로드 회수.
④ 그룹 58 : cycles, instructions, L3로부터의 로드, 메모리부터의 로드 회수 등
⑤ 그룹 53 : cycles, instructions, 고정소수점 연산, FP 연산(나누기, SQRT, FMA, and FMOV or FEST 포함) 회수 등

6.1.1. hpmcount

hpmcount는 사용자가 작성한 프로그램의 실제 실행 시간과 하드웨어 카운터에 관련된 정보, 사용 자원 등의 전반적인 성능을 제공하는 커맨드라인 유틸리티이다.

Sequential programs on AIX:

$hpmcount [-o <filename>] [-n] [-g <group>] <program>

Parallel programs (MPI) on AIX:

$poe hpmcount [-o <filename>] [-n] [-g <group>] <program>

-o <filename> : 출력화일 <filename>.<pid> 생성옵션. 병렬 프로그램에서는 프로세스마다 하나의 파일이 생성되며, 디폴트는 표준출력.
-n : 표준출력을 하지 않고 파일로만 출력. -o옵션과 같이 사용됨.
-g <group> : (POWER4 only) 0에서 60까지 그룹 지정 가능하며, 디폴트는 60.

hpmcount 사용 예제와 실행결과(-g 60은 없어도 됨)

```
$ hpmcount -o hpmtest -n -g 60 a.out
 Seconds elapsed:  20.1621870994567871
$ vi hpmtest_0000.384218

 hpmcount (V 2.4.3) summary

 Total execution time (wall clock time): 3.890695 seconds

 ########  Resource Usage Statistics  ########

 Total amount of time in user mode            : 3.800000 seconds
 Total amount of time in system mode          : 0.060000 seconds
 Maximum resident set size                    : 23564 Kbytes
 Average shared memory use in text segment    : 3088 Kbytes*sec
 Average unshared memory use in data segment  : 9007560 Kbytes*sec
 Number of page faults without I/O activity   : 5893
 Number of page faults with I/O activity      : 0
 Number of times process was swapped out      : 0
 Number of times file system performed INPUT  : 0
 Number of times file system performed OUTPUT : 0
 Number of IPC messages sent                  : 0
 Number of IPC messages received              : 0
 Number of signals delivered                  : 0
 Number of voluntary context switches         : 4
 Number of involuntary context switches       : 3

 #######  End of Resource Statistics  ########

  PM_FPU_FDIV (FPU executed FDIV instruction)                  :           0
  PM_FPU_FMA (FPU executed multiply-add instruction)           :  1000002329
  PM_FPU0_FIN (FPU0 produced a result)                         :   627844244
  PM_FPU1_FIN (FPU1 produced a result)                         :  1377820044
  PM_CYC (Processor cycles)                                    :  3758316603
```

```
PM_FPU_STF (FPU executed store instruction)                 :      1003408033
PM_INST_CMPL (Instructions completed)                       :      4143170873
PM_LSU_LDF (LSU executed Floating Point load instruction) :      2002211584

Utilization rate                                    :            74.306 %
                                                    ①
Load and store operations                           :         3005.620 M
                                                    ②
Instructions per load/store                         :            1.3786
                                                    ③
MIPS                                                :         1064.892
                                                    ④
Instructions per cycle                              :            1.102
                                                    ⑤
HW Float points instructions per Cycle              :            0.534
                                                    ⑥
Floating point instructions + FMAs       :   2002.259 M                 ⑦
Float point instructions + FMA rate      :       514.627 Mflip/s        ⑧
FMA percentage                                      :           99.887 %
                                                    ⑨
Computation intensity                               :            0.666
                                                    ⑩
```

위의 실행결과에서 수집된 정보에 대해 살펴보면 다음과 같다.

① Utilization rate = User time / Wall clock time

② Load and store operations = Loads + Stores(Total LS)
2002211584 + 1003408033 = 3005619617 = 3005.620 M

③ Instructions per load/store = Instructions completed / Total LS
4143170873 / 3005619617 = 1.3786

④ MIPS = 0.000001 * Instructions completed / Wall clock time
0.000001 * 4143170873 / 3.890695 = 1064.892
※ MIPS(Millions of Instructions per second)

⑤ Instructions per cycle = Instructions completed / Cycles
4143170873 / 3758316603 = 1.102

⑥ HW Float points instructions per Cycle = (FPU 0 + FPU 1) / Cycles
(627844244 + 1377820044) / 3758316603 = 0.534
※ FPU (Floating-point Processing Unit : 부동 소수점 연산장치) POWER4에는 프로세서 하나에 2개씩 있음)

⑦ Floating point instructions + FMAs (flip) = FPU 0 instructions + FPU 1 instructions + FMAs executed – FPU Stores (POWER4)
627844244 + 1377820044 + 1000002329 – 1003408033 = 2002258584 = 2002.259 M
※ FMA (Floating-point Multiply/Add), FDIV(Floating-point Divide)
※ POWER3 : flip = FPU 0 instructions + FPU 1 instructions + FMAs executed

⑧ Float point instructions + FMA rate = 0.000001 * flip / Wall clock time (Mflip/s)
0.000001 * 2002258584 / 3.890695 = 514.627 Mflip/s

⑨ FMA percentage = 100 * FMAs executed * 2 / flip
100 * 1000002329 * 2 / 2002258584 = 99.887 %

⑩ Computation intensity = flip / Total LS
2002258584 / 3005619617 = 0.666

hpmcount를 이용해 얻은 이와 같은 많은 정보들 중에서 사용자가 관심을 둘 항목은 코드에 의해 수행된 부동소수점 연산의 효율을 나타내는 Mflip/s(Millions of FLoat Instructions Per Second)이다. 이 값은 초당 수행된 부동 소수점 연산의 회수를 나타내며 Mflops와 동일한 정보를 준다. 위의 결과 514.627 (Mflip/s)가 IBM 1.0GHz POWER4 프로세서에서 실행시킨 것이라면 프로세서 하나의 이론 최고 성능(Peak Performance)이 4000Mflops 이므로 이론 최고 성능의 약 13%에 해당되는 결과가 되는 셈이다. 따라서 예제로 사용된 코드는 많은 부분을 최적화 시킬 필요가 있음을 알 수 있다.

6.1.2. libhpm

libhpm은 사용자 코드에서 호출하여 사용하는 Fortran/C/C++ 루틴들로 구성되는 라이브러리 이다. hpmcount가 코드 전체의 성능을 알아보는데 사용되는 것에 반해 libhpm는 코드내에서 호출하여 특정 부분에 대한 성능을 알아보는 목적으로 사용된다. libhpm 라이브러리는 OpenMP 또는 스레드 프로그램을 지원한다. libhpm 루틴을 호출하여 사용하는 프로그램을 컴파일하는 경우 사용자는 적절한 라이브러리를 링크 시켜줘야 하는데, OpenMP 또는 스레드 프로그램의 경우는 libhpm_r을 링크 시켜야 한다. libhpm은 프로그램의 실행 중에 필요한 정보를 수집하게 되므로 루프 내부에서 호출하여 사용하는 경우 프로그램 실행에 많은 부하를 줄 수도 있다.

6.1.2.1. 주요 함수들

C/C++ : hpmInit(taskID, progName)

Fortran : f_hpminit(taskID, progName)

- taskID : 노드 ID를 나타내는 정수
- progName : 프로그램 이름을 나타내는 스트링

C/C++ : hpmStart(instID, label)

Fortran : f_hpmstart(instID, label)

- instID : 성능측정을 원하는 구간을 나타내는 정수, 100까지의 자연수 사용.
- label : hpmviz를 사용할 때 나타나는 스트링

C/C++ : hpmStop(instID)

Fortran : f_hpmstop(instID)

- 각 instID에 대해 hpmStart에 대응하는 hpmStop이 반드시 필요

C/C++ : hpmTstart(instID, label)

Fortran : f_hpmtstart(instID, label)

- 스레드를 사용하는 프로그램에서 사용

C/C++ : hpmTstop(instID)

Fortran : f_hpmtstop(instID)

- 스레드를 사용하는 프로그램에서 사용

C/C++ : hpmGetTimeAndCounters(numCounters, time, values)

Fortran : f_GetTimeAndCounters (numCounters, time, values)

- 함수가 호출될 때마다 hpmInit이 호출된 이후에 축적된 회수와 초 단위의 시간을 출력한다.
- numCounters : 접근 회수를 나타내는 정수
- time : 배정도 실수
- values : numCounters의 크기를 가지는 long long 배열

C/C++ : hpmGetCounters(values)

Fortran : f_GetCounters (values)

- hpmGetTimeAndCounters와 유사하며 단지 축적된 회수만 출력하여 부하를 최소화 한다.

C/C++ : hpmTerminate(taskID)

Fortran : f_hpmterminate(taskID)

- 출력파일을 생성한다. 만약 호출하지 않으면 아무런 정보도 얻을 수 없다.

6.1.2.2. 출력

각 프로세스 마다 다음과 같은 두개의 파일을 생성한다.

- perfhpm[taskID].[pid] : 성능 관련 데이터를 포함하는 텍스트 파일
- hpm[taskID]_[progName]_[pid].viz : hpmviz 프로그램을 위한 데이터를 포함하는 파일. 필요 없으면 환경변수 HPM_VIZ_OUTPUT = FALSE로 설정.

이때 생성되는 파일에 수집되는 정보는 hpmcount에서 사용된 하드웨어 이벤트 카운터 집합에 포함되는 정보와 동일하다. 즉, POWER4 시스템의 경우 하드웨어 이벤트 정보를 분류한 61개(0 ~ 60)의 그룹 중의 하나를 지정해 원하는 정보를 출력하도록 할 수 있다. 그룹의 선택은 환경변수 HPM_EVENT_SET을 이용해 지정하며 디폴트는 60이다.

6.1.2.3. 컴파일과 링크

libhpm 라이브러리를 사용하기 위해서 프로그램 작성시 C/C++는 “libhpm.h”를 Fortran에서는 “f_hpm.h”를 각각 Include 시켜야 한다. 작성된 프로그램을 링크하는 과정에서는 libpmapi.a, libhpm.a(또는 libhpm_r.a), 그리고 liblm.a를 링크시켜야 한다. 확장자가 f인 Fortran 프로그램의 경우는 컴파일할 때 -qsuffix=cpp=f 옵션을 주어야 한다. 아래 예제의 컴파일 과정에서는 시스템에 HPM 사용을 위한 경로(path) 지정이 되어있지 않아 -I(include 경로 지정) -L(라이브러리 경

로 지정) 옵션을 주고 현재 KISTI IBM 시스템에 설치된 HPM의 경로를 모두 지정해 주었다.

6.1.2.4. 사용 예

libhpm 사용 예제(Fortran) 프로그램 : hpmtest.f

```
PROGRAM HPMTEST
#include "f_hpm.h"
INTEGER SIZE
PARAMETER(SIZE=2000)
REAL(8) N(SIZE,SIZE), M(SIZE,SIZE), L(SIZE,SIZE)
INTEGER taskID
taskID = 0
CALL f_hpminit( taskID, "hpmtest" )
CALL f_hpmstart( 1, "Do Loop"  )
DO i = 1, SIZE
   DO j = 1, SIZE
      N(i,j) = N(i,j) +  M(i,j)*L(i,j)
   ENDDO
END DO
CALL f_hpmstop( 1 )
CALL f_hpmterminate( taskID )
END
```

컴파일과 링크

```
$xlf -o hpmtest hpmtest.f -I/applic/hpm_2_4_3/include -L/applic/hpm_2_4_3/lib
-lhpm -lpmapi -lm -qsuffix=cpp=f
```

생성된 성능정보 저장 텍스트 파일 : perfhpm0000.2744552

```
$ vi perfhpm0000.2744552

libhpm (Version 2.4.3) summary - running on POWER4

 Total execution time of instrumented code (wall time): 2.027903 seconds

 ########  Resource Usage Statistics  ########

 Total amount of time in user mode            : 1.850000 seconds
 Total amount of time in system mode          : 0.220000 seconds
 Maximum resident set size                    : 94708 Kbytes
 Average shared memory use in text segment    : 25172 Kbytes*sec
 Average unshared memory use in data segment  : 13301940 Kbytes*sec
 Number of page faults without I/O activity   : 23677
 Number of page faults with I/O activity      : 0
 Number of times process was swapped out      : 0
 Number of times file system performed INPUT  : 0
 Number of times file system performed OUTPUT : 0
 Number of IPC messages sent                  : 0
 Number of IPC messages received              : 0
 Number of signals delivered                  : 0
```

```
Number of voluntary context switches          : 22
Number of involuntary context switches        : 10

#######   End of Resource Statistics   ########

Instrumented section: 1 - Label: Do Loop - process: 0
file: hpmtest.f, lines: 9 <--> 15
 Count: 1
 Wall Clock Time: 2.027757 seconds
 Total time in user mode: 1.37717590153846 seconds

 PM_FPU_FDIV (FPU executed FDIV instruction)                    :              0
 PM_FPU_FMA (FPU executed multiply-add instruction)             :        4003433
 PM_FPU0_FIN (FPU0 produced a result)                           :        7989366
 PM_FPU1_FIN (FPU1 produced a result)                           :          24891
 PM_CYC (Processor cycles)                                      :     1790328672
 PM_FPU_STF (FPU executed store instruction)                    :        4010925
 PM_INST_CMPL (Instructions completed)                          :      112055809
 PM_LSU_LDF (LSU executed Floating Point load instruction)      :       12156326

 Utilization rate                          :          67.916 %
 Load and store operations                 :          16.167 M
 Instructions per load/store               :          6.931
 MIPS                                      :          55.261
 Instructions per cycle                    :          0.063
 HW Float points instructions per Cycle    :          0.004
 Floating point instructions + FMAs        :          8.007 M
 Float point instructions + FMA rate       :          3.949 Mflip/s
 FMA percentage                            :          100.001 %
 Computation intensity                     :          0.495
```

6.1.3. hpmviz

hpmviz는 libhpm라이브러리를 호출하여 생성한 성능분석결과를 GUI 기반으로 보여주는 유틸리티이다. 위의 예제(hpmtest.f)를 실행한 결과로 생성된 hpm0000_hpmtest_2744552.viz를 보기 위하여 적절한 X-윈도우 설정을 한 후 다음과 같이 실행 시킨다.

```
$ hpmviz hpm0000_hpmtest_2744552.viz
```

실행결과

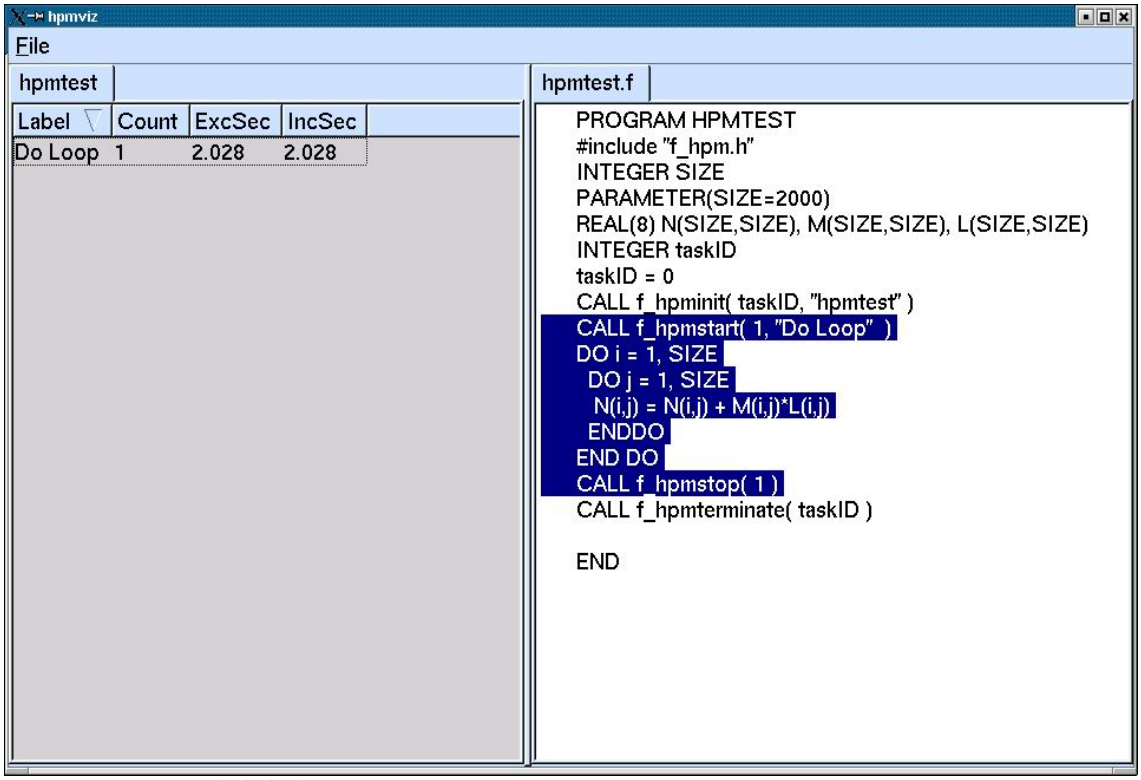


그림 6-1. hpmviz 실행화면

열려진 창의 좌측은 코드내에 성능 측정을 위해 hpmlib 루틴을 삽입한 부분과 그 부분의 실행시간 정보를 표시하고 우측 창에는 실제 코드를 표시한다. 좌측 창에서 hpmlib 루틴을 삽입한 부분을 선택하여 마우스 오른쪽 버튼을 누르면 그 부분에 대한 보다 자세한 HPM 정보를 보여주는 새로운 창이 하나 뜬다.

6.2.PCT와 Jumpshot을 이용한 MPI 프로그램 성능 분석

6.2.1. PE Benchmarker

PE Benchmarker는 IBM AIX Parallel Environment 환경에서 구동 되는 프로그램들의 성능을 분석할 수 있는 응용프로그램과 도구들로 구성된 툴셋이며, 크게 다음 세가지로 이루어져 있다.

- PCT(Performance Collection Tool) : 하나 혹은 그 이상의 응용프로그램 프로세스로부터 MPI 이벤트 트레이스 데이터 혹은 하드웨어/운영체제 성능 데이터를 수집한다. 커맨드라인 인터페이스 모드와 그래픽 사용자 인터페이스(GUI) 모드로 실행 가능하다.
- UTE(Unified Trace Environment) 도구 모음 : PCT를 이용해 수집된 MPI 이벤트 트레이스 정보를 저장해둔 AIX 트레이스 파일들을 UTE interval 파일로 변환시키고, 변환된 파일로부터 성능분석 테이블을 생성한다. 다음과 같은 UTE 도구들이 있으며 커맨드라인 인터페이스 모드로 실행된다.
  - uteconvert : AIX 이벤트 트레이스 파일을 UTE interval 파일로 변환한다.
  - utemerge : 여러 개의 UTE interval 파일을 하나의 interval 파일로 합친다.
  - utestats : UTE interval 파일로부터 얻은 정보에 대한 통계 테이블을 생성한다.
  - slogmerge : Argonne 국립 연구소의 MPI 프로그램 성능분석 도구인 Jumpshot을 이용할 수 있도록 UTE interval 파일들을 SLOG(Scalable logfile) 파일 포맷으로 변환시키고 합친다.
- PVT(Performance Visualization Tool) : PCT를 이용해 하드웨어/운영체제 성능 데이터를 수집하면, 각 프로세스별로 수집된 프로파일 정보는 netCDF(network Common Data Form) 파일로 저장된다. PVT는 netCDF 파일로부터 프로파일 정보를 읽고 요약해 사용자에게 보여준다. 커맨드라인 모드와 GUI 모드로 실행 가능하다.

PE Benchmarker 툴셋을 이용한 프로그램의 성능분석은 PCT와 PVT를 이용한 하드웨어/운영체제 성능분석과 PCT와 UTE 혹은 PCT와 UTE 그리고, Jumpshot을 이용한 MPI 프로그램 성능분석으로 나누어 볼 수 있다(그림 6-2). 이중 PCT와 PVT를 이용한 하드웨어/운영체제 성능분석은 HPM 툴킷과 그 기능이 유사하다. 이곳에서는 PCT와 UTE를 이용하여 MPI 프로그램의 성능 정보를 수집하고 Argonne 연구소의 Jumpshot을 이용하여 수집된 정보를 GUI 환경에서 분석하는 방법을 소개한다.

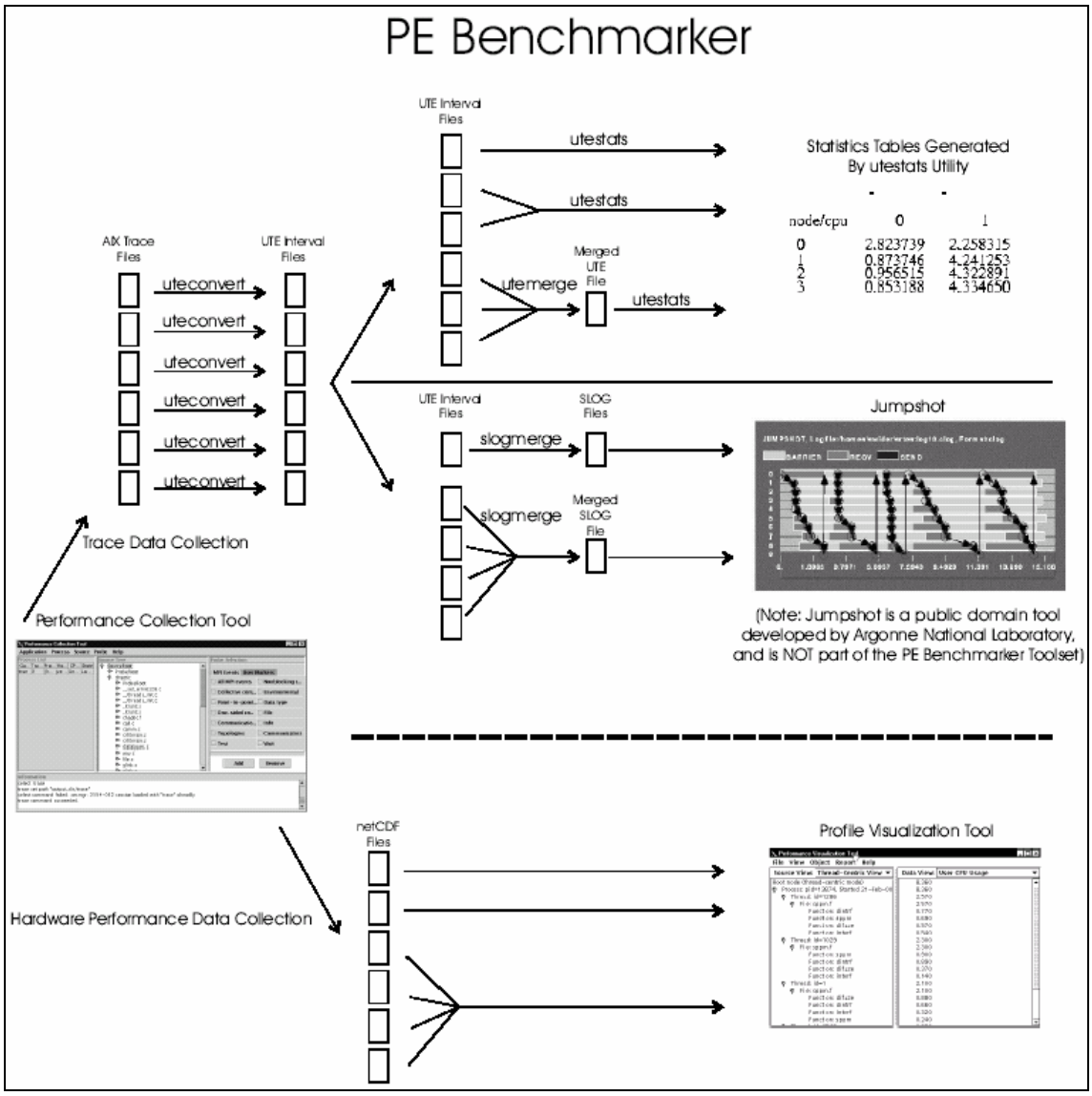


그림 6-2. PE Benchmarker를 이용한 프로그램 성능분석

6.2.2. MPI 프로그램 프로파일을 위한 PCT 사용법

앞서 설명했듯이 PCT는 커맨드라인 인터페이스와 GUI를 모두 지원한다. 여기서는 GUI를 기준으로 하여 PCT의 사용법을 설명할 것이다. 커맨드라인 인터페이스에서 PCT를 사용하는 것에 대해서는 IBM의 PE for AIX V3R2.0 Operation and Use, Vol. 2의 105 페이지에 소개되어 있다.

Jumpshot을 이용한 MPI 프로그램 성능분석을 위하여, 분석에 사용될 MPI 트레이스 파일을 PCT를 이용하여 생성해야 한다. 이를 위해 성능분석이 필요한 프로그램을 컴파일하기 전에 먼저 필요한 라이브러리를 링크시켜야 하는데, 이는 환경변수 MP_UTE를 yes로 설정해두면 된다. 만약 사용자의 UNIX 쉘 환경이 ksh이면 export MP_UTE=yes 명령어를 커맨드라인 상에서 입력하면 된다. 그리고, 이렇게 설정된 MP_UTE 환경변수를 통해 UTE 라이브러리를 추가할 수 있도록 하기위해서는 컴파일 스크립트를 반드시 "_r" 버전을 사용해야 한다.

1) 환경변수 설정

```
$ export MP_UTE=yes
```

2) 프로그램 컴파일

성능분석을 위한 MPI 프로그램을 스레드 사용 가능한(thread-enabled) 컴파일 스크립트(_r)를 이용하여 컴파일 한다. 다음에 사용된 pipelined.f 코드는 2차원 Laplace 방정식을 푸는 프로그램으로 IBM 레드북 PE for AIX V3R2.0, Hitchhiker's Guide에 소개되어 있다.

```
$ mpxlf_r -o pipelined pipelined.f
```

3) PCT 실행

사용자의 시스템에서 PCT를 GUI로 사용하기 위해서는 적절한 X윈도우 환경이 우선적으로 설정되어 있어야 한다.

```
$ pct
```

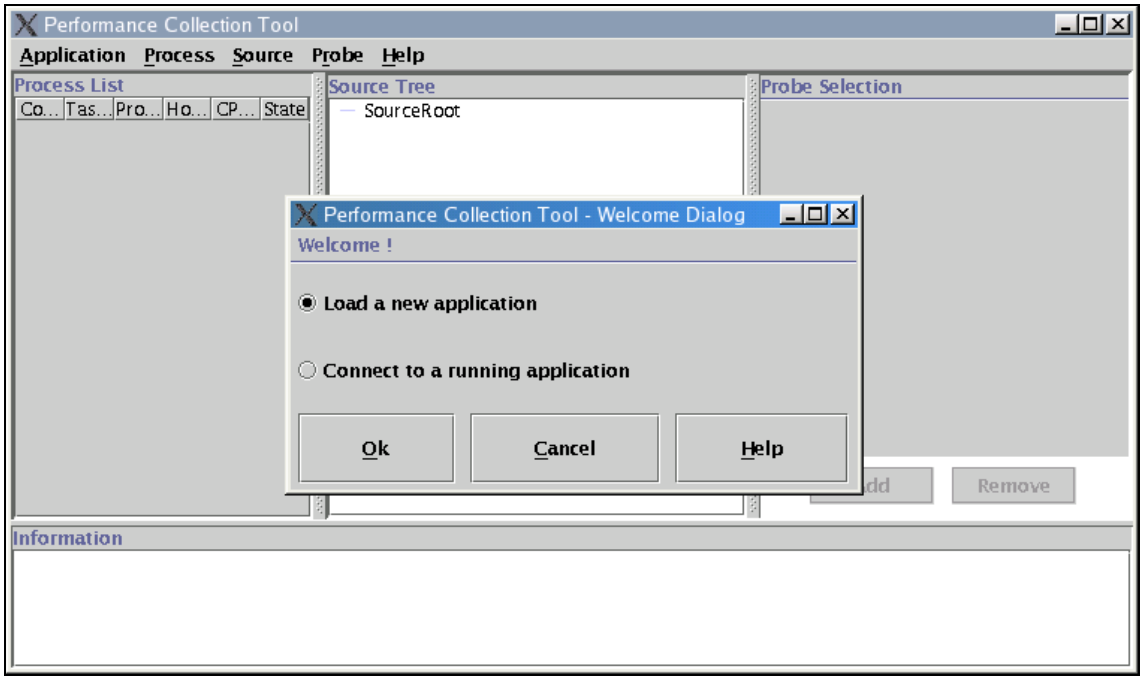


그림 6-3. PCT 실행 화면

① Load a new application을 선택하고 OK를 클릭한다. (Load Application 윈도우가 열린다.)

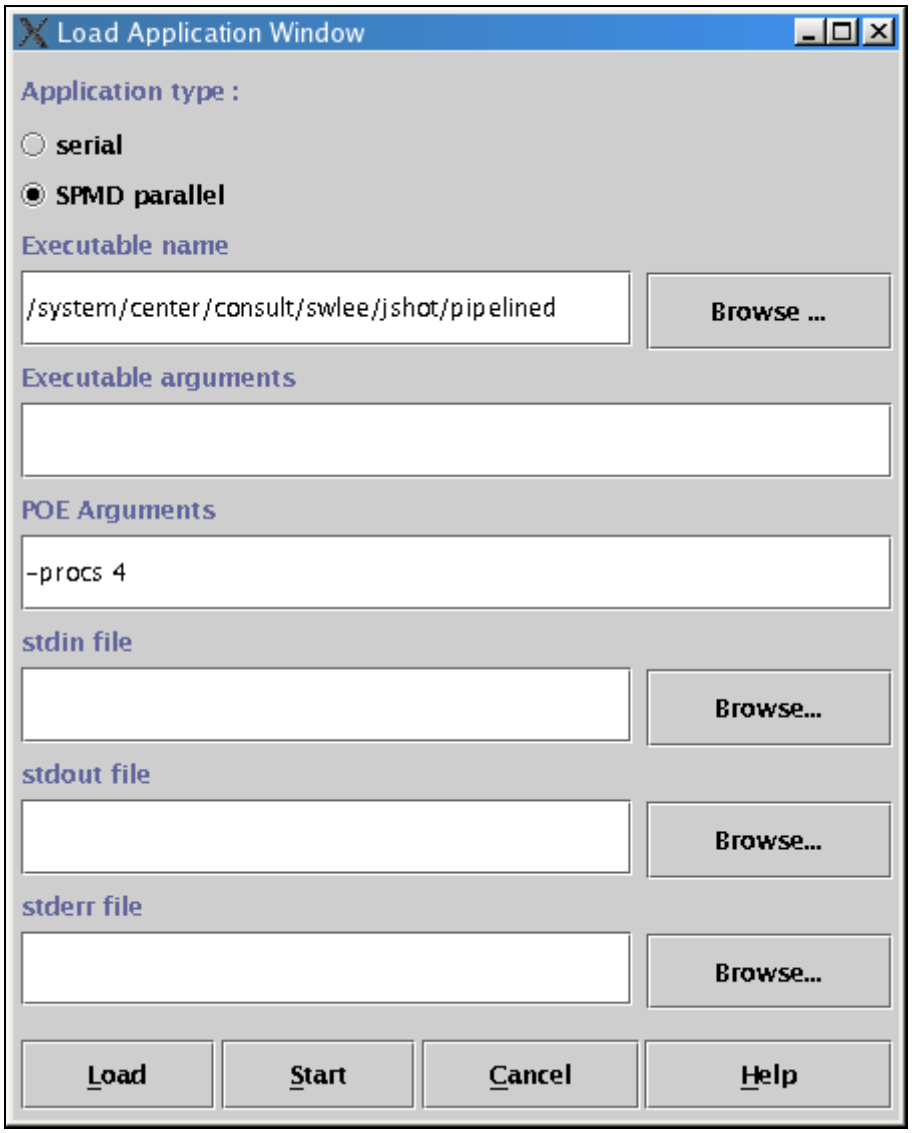


그림 6-4. Load Application 윈도우

② Load Application 윈도우에서 SPMD parallel을 선택하고 Executable name 필드에서 Browse 버튼을 이용하여 성능분석이 필요한 프로그램을 선택한다.

③ POE Arguments 필드에 적절한 POE 실행 옵션을 넣는다. 예를 들면, 병렬실행에 참여하는 프로세스 개수(-procs n) 등을 넣어준다.

④ Load 버튼을 클릭하여 프로그램을 로드한다. (Probe Data Selection 윈도우가 열린다.)

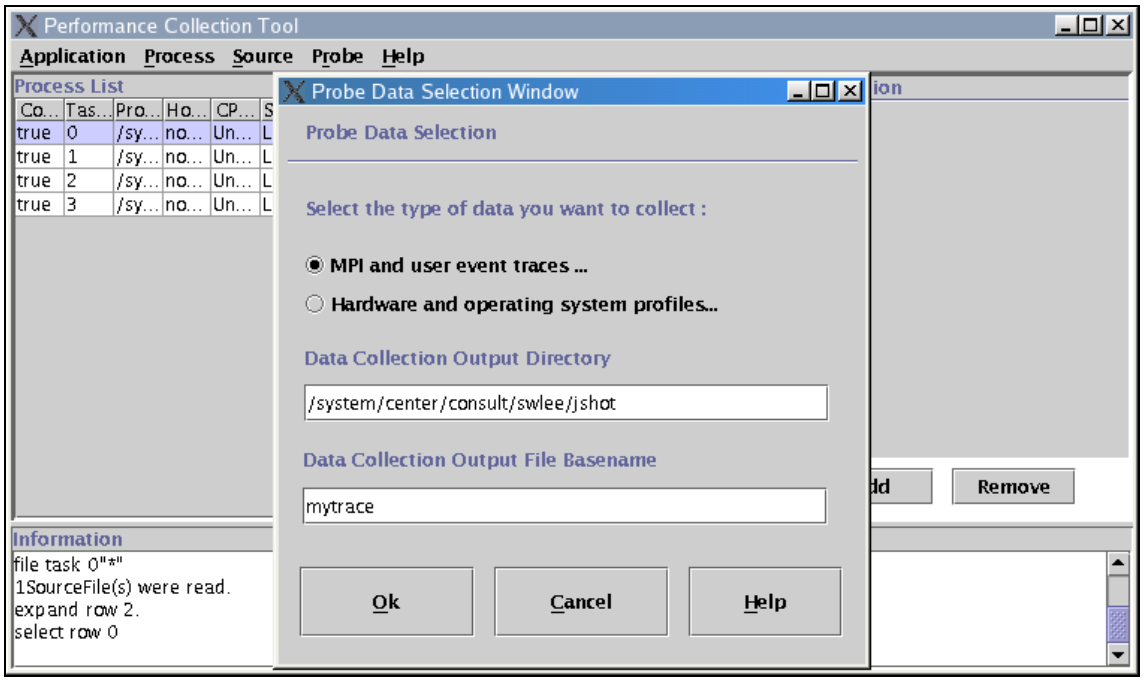


그림 6-5. Probe Data Selection 윈도우

⑤ 수집을 원하는 데이터의 타입을 선택하는데 지금과 같이 MPI 프로그램 성능분석을 위해서는 MPI and user event traces를 선택해야 하지만 만약 하드웨어/운영체제 성능 데이터를 원한다면 Hardware and operating system profiles를 선택해야 한다.

⑥ 출력파일이 생성될 디렉터리와 기본파일명을 적어주고 OK를 클릭한다. (메인 윈도우가 열린다.)

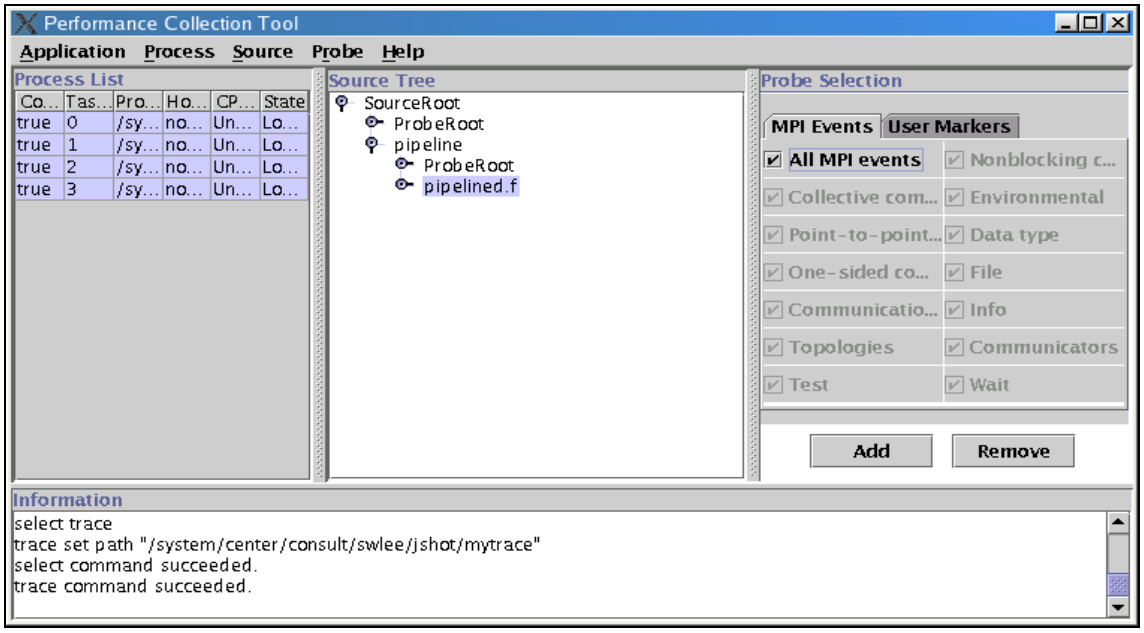


그림 6-6. PCT 메인 윈도우

⑦ 모든 프로세스의 진행 상황을 보기위해 메인 윈도우에서 Process → Select All Connected를 선택한다.

⑧ Source Tree에서 pipelined.f를 선택한다.

⑨ 트레이스 정보를 수집하기 위해 Probe Selection 영역에서 All MPI Events를 선택한다.

⑩ Add 버튼을 클릭해 필요한 정보 수집을 위한 탐침(probe)을 설치한다.

⑪ 메뉴 바에서 Application → Start를 선택해 프로그램을 실행시킨다.

⑫ 프로그램의 실행이 완료되면 Target Application Exited 윈도우가 나타나고 OK버튼을 클릭하면 PCT가 종료된다.

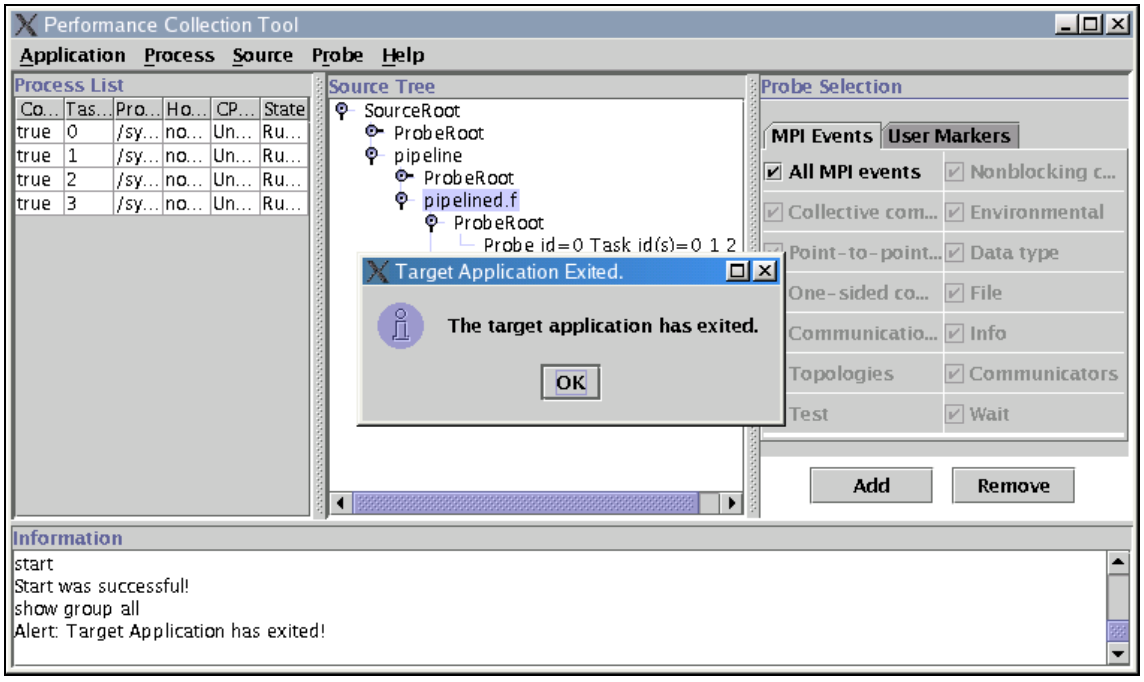


그림 6-7. PCT를 이용한 정보수집 완료

위와 같은 과정을 거쳐 성공적으로 메시지 패싱에 대한 데이터를 수집했다면 AIX 트레이스 파일이 생성된다. 이 트레이스 파일은 각 노드마다 하나씩 생성되는데, 위의 예에서 사용한 네 개의 프로세스 모두를 하나의 노드에서 실행되도록 했다면 트레이스 파일은 mytrace.#의 형식으로 하나가 생성된다. 여기서 #은 프로세스가 네 개이므로 0부터 3까지의 값 중 임의의 값을 하나 가지게 된다.

4) uteconvert 실행

생성된 AIX 트레이스 파일을 uteconvert를 이용하여 UTE interval 파일로 변환한다. 기본적으로 트레이스 파일의 이름을 그대로 가져와 mytrace.ute.# 파일이 생성되지만 -o 옵션을 이용하여 원하는 이름을 지정해 줄 수 있다.

```
$ uteconvert mytrace.1
```

```
$ uteconvert -o tracefile.ute mytrace.1
```

5) slogmerge 실행

수집한 정보를 Jumpshot에서 볼 수 있도록 slogmerge를 이용해 UTE interval 파일을 SLOG 파일로 변환한다. 기본적으로 (UTE 파일 이름).slog 파일이 생성되지만 역시 -o 옵션을 이용하면 원하는 이름을 지정해 줄 수 있다.

```
$ slogmerge mytrace.ute.1
```

```
$ slogmerge -o tracefile.slog tracefile.ute
```

6.2.3. Jumpshot 사용법

Jumpshot은 PE Benchmarker 툴셋에 포함된 프로그램이 아니며 이는 미국 Argonne 국립 연구소에서 개발한 공개 MPI 구현 패키지인 MPICH/MPE에 포함된 자바기반의 성능분석 가시화 툴이다. 이는 현재 KISTI IBM 1차시스템의 /applic/bin 디렉토리에 jumbshot이라는 이름으로 설치되어 있으며 인터넷에서 받아 개별적으로 설치해 사용할 수도 있다. GUI 기반의 Jumpshot도 PCT와 마찬가지로 실행하기 전에 적절한 X윈도우 설정이 필요하다.

성능분석을 원하는 MPI 프로그램의 트레이스 파일 생성부터 이것을 다시 SLOG 파일 포맷으로 변환시키는 과정까지 무사히 마쳤다면 이제 Jumpshot을 실행해 트레이스 파일 정보를 가시화시켜 보자.

1) Jumpshot 실행

```
$ /applic/bin/jumpshot
```

① 메뉴 바에서 File → Select Logfile을 이용해 만들어둔 SLOG 파일을 로드한다.(View and Frame Selector 윈도우가 열린다.)

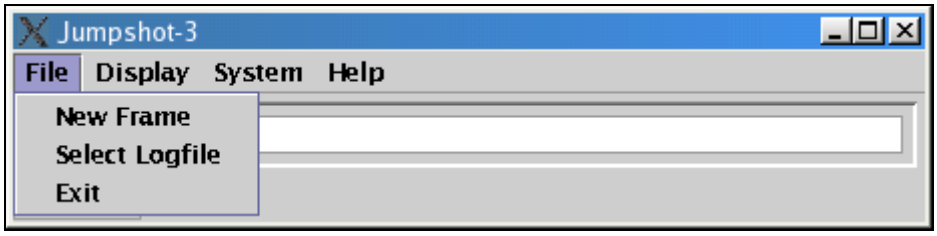


그림 6-8. Jumpshot 실행화면

② View and Frame Selector 윈도우에는 Event Count vs. Time의 그래프가 나타난다. 프로그램의 진행시간에 따른 적절한 프레임을 선택하고 View Options에서 MPI-Process를 선택한 후 Display 버튼을 클릭하면 Time line 윈도우를 볼 수 있다.

그림 6-9. View and Frame Selector 윈도우

③ Time Line 윈도우의 X축은 프로그램의 진행시간을 Y축은 프로세스 랭크를 나타낸다. MPI

서브루틴은 하나의 box로 표현되고 함수 내부 혹은 함수 사이의 통신은 화살표로 표시된다. 상단의 Zoom Operations에서 In/Out 버튼을 이용하여 선택한 프레임의 MPI 이벤트 발생을 좀더 자세히 살펴볼 수 있다.

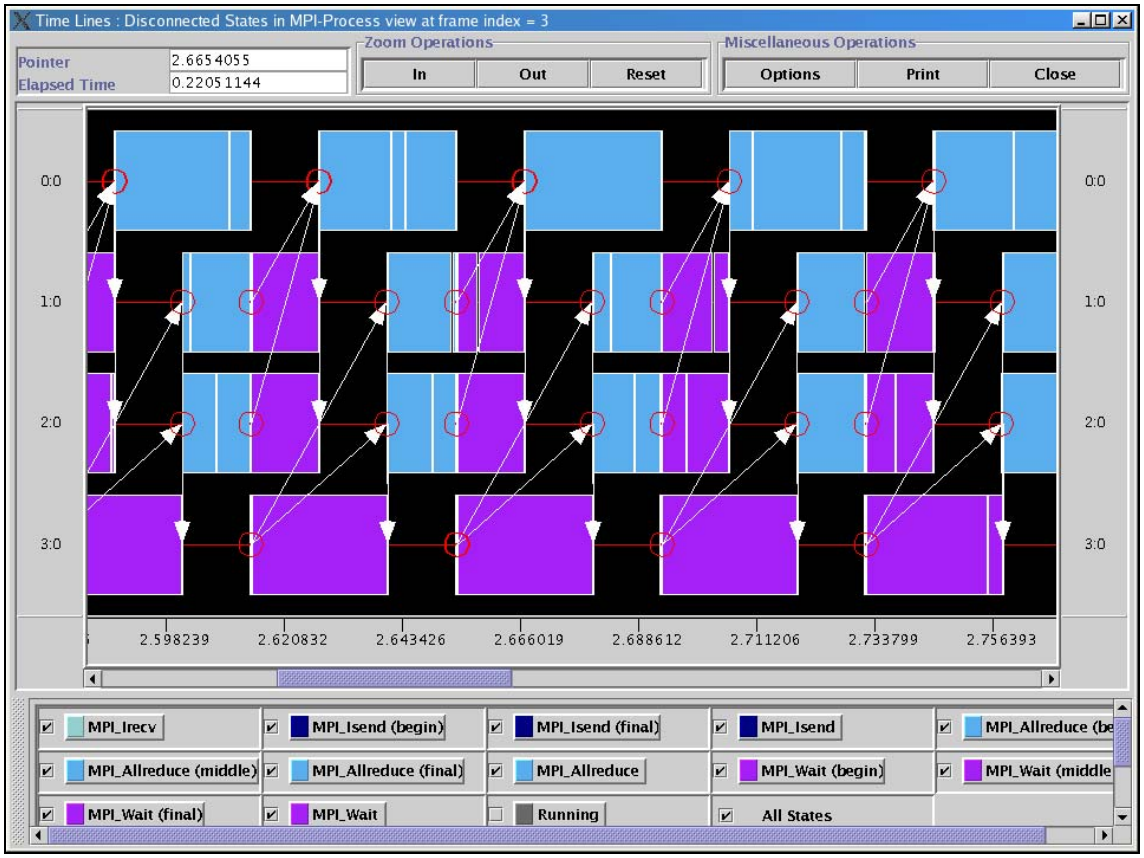


그림 6-10. Time Line 윈도우

사용자는 Time Line 윈도우의 MPI 프로그램 진행상황과 통신특성 등을 살펴보고 자신의 프로그램에 대한 성능분석을 한 후 적절한 최적화 방안을 마련할 수 있다. 가령 위의 그림을 예로 보면 프로세스들이 작업을 진행하기 전에 다른 프로세스들의 작업완료를 기다리며 많은 시간을 낭비하고 있음을 볼 수 있다. 즉 프로세스 랭크 1, 2, 3의 실행에서 밝은 보라색으로 표시된 부분이 MPI_Wait의 실행을 하면서 대기하는 것을 나타내고 있는데, 특히 3번 프로세스의 경우 대기시간이 대부분을 차지하고 있음을 확인할 수 있다.

참고자료

1. Kevin Dowd and Charles Severance. High Performance Computing, second edition. O'Reilly. 1998
2. The POWER4 Processor Introduction and Tuning Guide (http://www.ibm.com/redbooks)
3. User's Guide , XL Fortran for AIX, Version 8 Release 1
4. Language Reference, XL Fortran for AIX, Version 8 Release 1
5. Andrews. Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley. 2000.
6. SP Parallel Programming Workshop http://www.mhpcc.edu/training/workshop/
7. Performance Tuning for Clusters http://foxtrot.ncsa.uiuc.edu:8900
8. Introduction to Performance Engineering http://foxtrot.ncsa.uiuc.edu:8900
9. Lawrence Libermore National Laboratory http://www.llnl.gov/computing/tutorials/workshops/workshop/
10. David A. Patterson and John L. Hennessy. Computer Organization and Design, second edition. Morgan Kaufmann. 1998
11. Gropp, Lusk, and Skjellum. Using MPI, second edition. MIT Press. 1999
12. Snir, Otto, Huss-Lederman, Walker, and Dongarra. MPI-The Complete Reference Volume 1. Second Edition. MIT Press. 1998.
13. IBM Software Publications: IBM Parallel Environment for AIX, Hitchhiker's Guide
14. . IBM Software Publications: IBM Parallel Environment for AIX, Operation and Use vol 1, vol 2
15. Performance Optimizations: Edinburgh Parallel Supercomputing Center
16. Parallel Programming Guide for HP-UX Systems: http://docs.hp.com/en/B3909-90015/index.html
17. HPC 기술서 (http://www.supercomputing.re.kr)
    - IBM 기술서 1권 (2002년 9월)
    - IBM 기술서 3권 (2003년 3월)
    - IBM 기술서 4권 (2003년 5월)

찾아보기