13. 성능 측정 및 최적화

13.1 IPC (Instruction per Cycle) 측정

> 프로세서가 "한 사이클에 얼마나 많은 일을 했는가?"의 척도

🧠 IPC란?

IPC (Instructions Per Cycle)는
CPU가 **한 클럭 사이클에 평균적으로 실행한 명령어 수**를 나타내는 지표이다.

> 성능을 결정짓는 대표적인 수치 중 하나이며,
> IPC가 높을수록 같은 클럭에서도 더 많은 일을 한다는 의미이다.

📏 기본 공식

```
IPC = Instruction Count / CPU Clock Cycles
```

항목	설명
Instruction Count	실제 실행된 명령어 개수
CPU Clock Cycles	명령어 수행에 소요된 사이클 수
CPI	Cycles Per Instruction = 1 / IPC

🔍 IPC의 해석 기준

IPC 값	의미	예시
~0.5 이하	파이프라인 활용률 낮음	메모리 병목, 분기 실패
~1.0	기본 성능 수준	비순차 구조 없음
1~2 이상	파이프라인 및 캐시 효율적 사용	Superscalar 구조 활용
3~5 이상	Out-of-Order + Parallelism 매우 우수	고성능 ARM, x86 OoO Core

🧩 IPC 측정 방식

✅ A. 하드웨어 PMU (Performance Monitoring Unit) 기반

리눅스 `perf` 도구 (x86, ARM 공통):

```
sudo perf stat ./my_program
```

💡 출력 예:

```
Performance counter stats for './my_program':

        4,300,000 instructions
        2,150,000 cycles

       → IPC = 2.0
```

측정 항목:

이벤트	의미
`instructions`	실행된 명령어 수
`cycles`	클럭 사이클 수
`branches`, `cache-misses`	병목 분석도 가능

✅ B. ARM Cortex A 계열 PMU 직접 접근

```
echo 0x08 > /sys/bus/event_source/devices/armv8_pmuv3_*/events/instructions
echo 0x11 > /sys/bus/event_source/devices/armv8_pmuv3_*/events/cycles
```

또는 `perf` 또는 `trace-cmd`, `LTTng` 등으로 연동 가능

✅ C. 소프트웨어 추정 기반 (간접 방식)

방식	설명
시뮬레이터	QEMU + perf 통합
VM 기반 측정	Hypervisor에서 guest IPC 분석
Android	simpleperf (NDK tool)

```
adb shell simpleperf stat --app com.example.app
```

🧪 테스트 예시 (x86 기준)

```
$ gcc -O2 -o test test.c
$ perf stat ./test
```

출력 예시:

```
1,234,567 instructions
  678,910 cycles
→ IPC = 1.82
```

🧠 IPC에 영향을 주는 요소

항목	설명
Branch Misprediction	잘못된 분기 예측으로 인한 플러시
Cache Miss	메모리 지연으로 파이프라인 stall
Dependency Stall	이전 명령어가 완료되어야 다음 명령 수행 가능
Instruction Mix	ALU vs Load/Store 비율
파이프라인 병렬성	Superscalar, OoO 구조 활용도

🎯 IPC 향상 전략

전략	설명
알고리즘 최적화	분기 최소화, 순차 연산 증가
메모리 지역성 개선	Cache hit 비율 향상
루프 언롤링	분기 감소, 병렬 실행 유도
컴파일러 최적화 옵션 사용	`-O2`, `-march=native`, `-funroll-loops` 등
멀티코어 병렬화	단일 코어 IPC는 유지되지만 전체 처리량 증대 가능

📌 요약 정리

항목	설명
IPC란?	한 클럭 사이클 당 수행된 명령어 수
측정 도구	`perf`, PMU, ARM DTrace, Android `simpleperf`
IPC 해석	1.0 이상이면 성능 양호, 2.0+은 고성능
영향 요소	캐시, 분기, 파이프라인 구조
최적화 방향	메모리 접근 최소화, 병렬성 극대화

13.2 CPI 분석

> 프로세서가 "한 개의 명령어를 수행하는 데 걸리는 평균 사이클 수"

🧠 CPI란?

CPI (Cycles Per Instruction)는
CPU가 **하나의 명령어를 실행하는 데 평균 몇 클럭 사이클이 소요되는지** 나타내는 지표이다.

```
CPI = Total CPU Cycles / Total Instructions Retired
```

항목	설명
낮을수록	고성능 (빠르게 명령 처리)
높을수록	병목 존재 (지연 원인 있음)

🔁 IPC와의 관계

```
IPC = Instructions / Cycle
CPI = Cycle / Instructions
→ 서로 역수 관계: CPI = 1 / IPC
```

IPC 높음 ↔ CPI 낮음

📏 기준 CPI 수치 예

CPU 종류	이론적 최적 CPI	실제 범위
단일 파이프라인 RISC	1.0	1.1 ~ 2.0
Superscalar	< 1.0	0.3 ~ 1.0
OoO + 멀티발행	< 0.5	0.2 ~ 0.8

🧪 perf를 통한 실전 측정 (Linux 기준)

```
perf stat ./your_program
```

출력 예시:

```
Performance counter stats for './your_program':

        10,000,000 cycles
         5,000,000 instructions
         2.00  → CPI (10M / 5M)
```

→ **CPI가 2.0**이면 명령어 하나당 2 사이클 걸렸다는 의미

🧩 CPI 상승의 원인별 분해

CPI는 다음과 같은 이벤트에 의해 **합산**되는 구조로 모델링할 수 있다:

```
CPI = Base CPI
    + Memory Stall CPI
    + Branch Stall CPI
    + Structural Hazard CPI
    + Resource Contention CPI
```

항목	설명
Base CPI	완벽한 캐시 히트, 분기 예측 성공 가정
Memory Stall	L1/L2/L3 Miss에 의한 대기
Branch Stall	분기 실패 후 파이프라인 flush
Resource Stall	포트 충돌, OoO 자원 부족 등

🛠️ 상세 이벤트 측정 (x86 / ARM 공통)

```
perf stat -e cycles,instructions,cache-misses,branch-misses ./prog
```

결과 분석 예시:

항목	값	의미
cycles	20,000,000	총 사이클 수
instructions	10,000,000	총 실행된 명령어
cache-misses	500,000	L1/L2 Miss
branch-misses	100,000	분기 예측 실패 수
→ CPI	2.0 (20M / 10M)	

📉 CPI 문제 진단 전략

증상	가능한 원인	개선 전략
CPI > 2.0	메모리 병목	배열 접근 최적화, Cache blocking
Branch-misses ↑	예측 실패	조건문 단순화, Loop unrolling
Instructions ↓	컴파일러 최적화 부족	`-O2`, `-march=native`
cycles ↑	파이프라인 flush, 레이턴시	구조 병렬화, out-of-order 활용

📊 고급 분석: CPI Breakdown

이벤트	perf 이벤트명	설명
L1 miss	`L1-dcache-load-misses`	Load 지연
LLC miss	`LLC-load-misses`	RAM 접근
Branch mispredict	`branch-misses`	flush 비용
Frontend Stalls	`idq_uops_not_delivered.core`	디코딩 병목

이벤트	perf 이벤트명	설명
Backend Stalls	`backend_stall_cycles`	ALU 등 자원 부족

🧠 CPI 개선 실무 전략

전략	설명
메모리 지역성 향상	연속 메모리 접근으로 Cache Miss 감소
Branch Reduction	정적 예측 가능한 흐름 구성
컴파일 최적화	`-O2`, `-funroll-loops`, `-fomit-frame-pointer`
SIMD 활용	명령어 수 증가 없이 처리량 향상
병렬 분할	독립 루틴을 다른 코어에 분산

📌 요약 정리

항목	설명
CPI 정의	명령어 1개당 평균 걸린 사이클 수
측정 공식	`CPI = cycles / instructions`
이상적 범위	Superscalar CPU: 0.5 ~ 1.0
측정 도구	`perf`, PMU, OProfile, LTTng
분석 키워드	Cache Miss, Branch Miss, Resource Stall
개선 전략	메모리/분기 최적화, 컴파일러 튜닝, 병렬화

13.3 캐시 미스율 분석

🧠 캐시 미스란?

캐시 미스(Cache Miss)는
CPU가 원하는 데이터를 **캐시에서 찾지 못해 더 느린 계층으로 접근하는 상황**을 말한다.

→ 캐시 미스는 성능에 **직접적인 지연(CPU stall)**을 발생시킴.

🔍 캐시 계층 구조

```
CPU Core
 └─ L1 Cache (Instruction/Data)
     └─ L2 Cache (Unified)
         └─ L3 Cache (Shared across cores)
             └─ Main Memory (DRAM)
```

📏 캐시 미스율(Miss Rate) 공식

```
Miss Rate = Cache Misses / Total Cache Accesses
```

계층	이벤트	의미
L1 Miss	`L1-dcache-load-misses` / `L1-dcache-loads`	매우 빠른 계층에서 실패
LLC Miss	`LLC-load-misses` / `LLC-loads`	마지막 캐시 방어선 실패 → RAM 접근

📦 Miss의 유형

유형	정의	영향
Compulsory Miss	처음 보는 데이터	불가피
Capacity Miss	캐시 공간 부족으로 교체됨	구조적 한계
Conflict Miss	동일 인덱스에 다른 블록이 경쟁	캐시 설계 의존
Coherence Miss	멀티코어 간 캐시 일관성 문제	SMP 구조 영향

🧪 실전 측정 예 (Linux `perf` 기반)

```
perf stat -e L1-dcache-loads,L1-dcache-load-misses \
          -e LLC-loads,LLC-load-misses \
          ./your_program
```

💡 결과 예시:

```
  10,000,000 L1-dcache-loads
   1,500,000 L1-dcache-load-misses → 15% miss rate

   1,200,000 LLC-loads
     300,000 LLC-load-misses       → 25% miss rate
```

🧩 캐시 미스율 해석 기준

Miss Rate	해석
< 5%	매우 우수한 지역성
5~20%	개선 여지 있음
> 20%	캐시 효율 낮음 → 병목 가능성 높음
LLC Miss > 30%	메모리 병목 의심

📊 캐시 미스가 CPI에 미치는 영향

```
Stall_Cycles = Misses × Memory_Latency
CPI ↑ ← Stall_Cycles ↑ ← Miss Rate ↑
```

예시:

- L1 Miss: 4~5 cycles penalty
- LLC Miss: 50~200+ cycles penalty (DRAM 접근)

→ 캐시 미스율이 높으면 IPC ↓, CPI ↑

🛠️ 캐시 미스 최적화 전략

전략	설명
메모리 지역성 개선	배열 순서대로 접근 (행 우선 vs 열 우선 등)
데이터 구조 재설계	연속성 있는 구조체 배열 사용 (AoS → SoA)
루프 변형	Loop Tiling, Loop Fusion
Pre-fetch 사용	컴파일러 `__builtin_prefetch()`, HW prefetcher 활성화
캐시 친화적 정렬	`aligned_alloc`, `__attribute__((aligned))` 등

🔧 고급 분석 이벤트 (x86 기준)

이벤트	설명
`L1-dcache-load-misses`	L1D miss 수
`L1-icache-load-misses`	명령어 캐시 miss
`LLC-loads`, `LLC-load-misses`	Last Level Cache 미스 통계
`dTLB-load-misses`	TLB miss도 성능 영향 큼
`mem-stores`, `mem-loads`	전체 접근 수 (분모 역할)

📈 시각화 예: 캐시 미스율 시간 그래프

```
Time(ms) | L1 Miss (%) | LLC Miss (%) | Memory Bandwidth
---------|-------------|--------------|-----------------
0        |     5       |     10       |     1.2 GB/s
10       |    25       |     35       |     3.8 GB/s
20       |    12       |     15       |     2.1 GB/s
```

→ 특정 시간 구간에 **캐시 효율 저하 + 메모리 대역폭 상승** 발생 → 병목 구간 탐색 가능

📌 요약 정리

항목	설명
Miss Rate	Misses / Accesses
측정 도구	`perf`, PMU, Intel VTune, ARM Streamline
미스 영향	CPI 증가, IPC 감소, 메모리 병목
분석 지표	L1, LLC Miss Rate 별도 추적
최적화 방법	지역성 개선, 데이터 정렬, 루프 재작성
성과 기준	L1 < 10%, LLC < 25% 이상이면 개선 필요

13.4 프로파일링 도구 사용법 (perf, VTune 등)

항목	`perf` (Linux)	`Intel VTune`	`ARM Streamline`
지원 플랫폼	Linux (x86, ARM)	Windows/Linux (x86)	Embedded Linux, Android (ARM)
방식	CLI + 샘플링	GUI 중심 정밀 분석	PMU + 실시간 타임라인
주요 기능	IPC, CPI, cache/branch miss, flamegraph	CPI breakdown, memory hierarchy, thread 분석	CPU/GPU load, memory BW, 온도/전력
분석 대상	user space binary, PID/TID	multithreaded 앱, 라이브러리, 커널 스레드	전체 SoC 수준 성능 이벤트
출력	텍스트/그래프	상세 보고서 + 시각화	타임라인 + 통계 시트

✅ perf 실전 사용법 (Linux)

기본 성능 요약 (IPC, CPI 확인)

```
perf stat ./program
```

함수별 병목 분석

```
perf record ./program
perf report
```

캐시, 분기 실패 분석

```
perf stat -e cycles,instructions,cache-misses,branch-misses ./program
```

FlameGraph 시각화

```
perf record -F 99 -g ./program
perf script > out.perf
stackcollapse-perf.pl out.perf | flamegraph.pl > flame.svg
```

✅ VTune Profiler (Intel CPU용)

설치 후 실행

```
vtune -collect hotspots -result-dir r1 ./program
vtune-gui r1
```

주요 모드

모드	설명
Hotspots	CPU 소비 높은 함수 추적
Microarchitecture	CPI, 스톨 분석
Memory Access	L1~L3 미스, DRAM 대기 분석
Threading	스레드 간 로드 밸런싱 확인

✅ ARM Streamline (ARM SoC용)

사용 흐름

1. 타겟 보드에 `gator` 설치 및 실행
2. 호스트 PC에서 `Streamline` GUI 실행
3. 프로파일링 후 Timeline에서 hotspot 분석 + 코드 연계

측정 예

- CPU Load, Frequency
- Cache Miss Rate
- Memory Bandwidth
- Thermal, Power
- GPU/NPU Load

🧠 프로파일링 결과 해석 기준

메트릭	해석 기준
IPC < 1.0	파이프라인 효율 저하 가능
CPI > 1.5	메모리 지연/스톨 가능성
cache-miss > 10%	캐시 구조 비효율 → 지역성 개선 필요
branch-miss > 3%	분기 예측 실패 → 분기 단순화 필요
특정 함수 시간 비중 > 50%	해당 루틴이 병목 → 최적화 대상

📌 정리

목적	사용 도구
텍스트 기반 기본 성능 측정	`perf stat`, `perf record`
고급 병목 분석 (x86)	`Intel VTune`
ARM/SoC 전력 + 성능 통합	`ARM Streamline`
병목 루틴 시각화	`FlameGraph`, `VTune call graph`, `Streamline timeline`

13.5 병목 분석 및 파이프라인 최적화

🧠 병목(Bottleneck)이란?

병목은 CPU 자원 중 하나 이상이 **과도하게 지연되거나 낭비**되어
전체 파이프라인 성능이 **제한되는 상태**를 말한다.

> 예: 분기 실패로 인해 파이프라인 flush, 메모리 접근 지연 등

🔍 파이프라인 구조에서의 병목 위치

단계	설명	주요 병목
Frontend	명령어 fetch/decoding	Branch Misprediction, I-cache Miss
Backend	명령어 실행/메모리 접근	Memory Stalls, Execution Port Block
Retire	결과 커밋	Commit Rate 포화, Flush
Memory Subsystem	L1/L2/L3 Miss, TLB Miss	Load latency ↑, DRAM 대기

🧪 병목 분석 with `perf`

```
perf stat -e cycles,instructions,cache-misses,branch-misses \
          -e frontend_stall_cycles,backend_stall_cycles ./program
```

💡 예시 결과:

```
cycles                : 1,000,000
  instructions          :   500,000  → IPC = 0.5
  branch-misses         :   60,000
  cache-misses          :  150,000
  frontend_stall_cycles :  300,000
  backend_stall_cycles  :  400,000
```

→ **Frontend Stalls** + **Branch Miss** ↑ → 분기 예측 병목
→ **Backend Stalls** + **Cache Miss** ↑ → 메모리 병목

🧪 병목 분석 with Intel VTune

유용한 뷰

분석 항목	설명
Hotspot	함수별 시간 소비 비율
Microarchitecture Analysis	Frontend/Backend Stalls, IPC
Memory Access	Miss Latency, Load Hit/Miss %
Top-down Breakdown	Retiring / Bad Speculation / Backend Bound / Frontend Bound %

> **Backend Bound > 50%** → 캐시/DRAM 병목
> **Bad Speculation** ↑ → 분기 예측 실패
> **Frontend Bound** ↑ → 명령어 fetch 실패 (i-cache, TLB 등)

🔩 병목 원인별 최적화 전략

✅ 1. Frontend 병목 (Branch, Instruction Fetch)

문제	해결 방안
분기 실패	정적 예측 가능한 분기, if → 조건 배열화
루프 조건 복잡	Loop unrolling, flattening
instruction cache miss	코드 크기 감소, hot path 정렬

✅ 2. Backend 병목 (Memory, Execution Stall)

문제	해결 방안
L1/L2 캐시 미스	메모리 지역성 개선, 데이터 구조 재설계
DRAM 대기 ↑	blocking → tiling, blocking matrix 연산
ALU Port 충돌	독립 명령어 순서 조정, 레지스터 분산

✅ 3. Retire 병목

문제	해결 방안
Flush 빈도 ↑	예외 최소화, 데이터 유효성 보장
스레드 간 간섭	스레드 분리, false sharing 제거

📈 루틴 단위 병목 확인 (Flamegraph / perf report)

```
perf record -F 99 -g ./program
perf report
```

→ 특정 함수에서 샘플 비율 50% 이상 → 해당 루틴이 병목
→ loop 분석, 데이터 접근 패턴 확인

📦 파이프라인 최적화 예시

예: 행렬 곱 (Naive vs Cache Optimized)

```
// Naive
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    for(k=0;k<N;k++)
      C[i][j] += A[i][k] * B[k][j];
```

➡️ 이 방식은 `B[k][j]` 접근이 cache locality에 불리

개선 (Loop Tiling)

```
#define BLOCK 32
for(ii=0;ii<N;ii+=BLOCK)
  for(jj=0;jj<N;jj+=BLOCK)
    for(kk=0;kk<N;kk+=BLOCK)
      for(i=ii;i<ii+BLOCK;i++)
        for(j=jj;j<jj+BLOCK;j++)
          for(k=kk;k<kk+BLOCK;k++)
            C[i][j] += A[i][k] * B[k][j];
```

→ Cache Miss 급감, CPI 감소, IPC 증가

📌 요약 정리

항목	설명
병목 분석 목적	CPI/IPC 악화 요인 정량 추적
Frontend 병목	분기 예측 실패, I-cache Miss
Backend 병목	캐시 미스, DRAM 대기
분석 도구	`perf`, `VTune`, `FlameGraph`
최적화 전략	데이터 지역성, 분기 단순화, 병렬성 활용
핫스팟 루틴	시간 소비 함수에서 반복 구조 분석