

컴퓨터 구조 8조 보고서

Sim-Scalar를 이용한 최적의 벤치마크 분석

제출일	2025. 12. 14	과목	컴퓨터 구조	담당교수	이영서
이름	김정훈	학번		전공	차세대반도체공학
이름	박수혁	학번		전공	전자정보공학부 IT 융합전공
이름	박정원	학번		전공	전자정보공학부 IT 융합전공
이름	박효정	학번		전공	전자정보공학부 IT 융합학과

[목차]

1 프로젝트 개요.....	5
1.1 프로젝트 목적.....	5
1.2 기대효과.....	5
1.3 Sim-Scalar 및 벤치마크 프로그램.....	5
2 주요 Configuration 분석.....	7
2.1 Scalar.....	7
2.2 Functional Unit.....	8
2.3 Branch Predictor.....	10
2.4 Cache configuration.....	11
3 Processor 변화에 따른 성능 분석.....	13
3.1 ALU의 개수 변화.....	13
3.1.1 gcc.....	13
3.1.2 Go.....	14
3.1.3 mcf.....	15
3.1.4 gzip.....	16
3.2 Scalar width 변화.....	17
3.2.1 gcc.....	17
3.2.2 Go.....	18
3.2.3 mcf.....	19
3.2.4 gzip.....	20

3.3	Branch predictor 종류.....	21
3.3.1	gcc.....	21
3.3.2	Go.....	22
3.3.3	mcf.....	23
3.3.4	gzip.....	24
4	Cache 변화에 따른 성능 분석.....	26
4.1	Block size 변화.....	26
4.1.1	gcc.....	26
4.1.2	Go.....	27
4.1.3	mcf.....	28
4.1.4	gzip.....	29
4.2	Associativity 변화.....	30
4.2.1	gcc.....	30
4.2.2	Go.....	31
4.2.3	mcf.....	32
4.2.4	gzip.....	33
4.3	Replacement Policy 변화.....	34
4.3.1	gcc.....	34
4.3.2	Go.....	35
4.3.3	mcf.....	36
4.3.4	gzip.....	37
5	최적 Cache configuration 탐색.....	38

5.1	gcc.....	38
5.2	Go.....	39
5.3	mcf.....	41
5.4	gzip.....	43
6	Application별 최종 System configuration.....	45
6.1	gcc.....	45
6.2	Go.....	46
6.3	mcf.....	50
6.4	gzip.....	51
7	프로젝트 수행 일지.....	54
8	Troubleshooting.....	56

1 프로젝트 개요

1.1 프로젝트 목적

본 프로젝트의 목적은 Sim-Scalar 시뮬레이터를 활용하여 프로세서에서 사용되는 다양한 마이크로아키텍처 요소들이 실제 성능에 어떤 영향을 미치는지를 실험적으로 분석하는 것이다. 이를 위해 ALU의 개수, Scalar width, Branch Predictor 종류, Cache 구조(Block Size, Associativity, Replacement Policy)의 변화가 SPEC2000 벤치마크(gcc, go, mcf, gzip)의 IPC·CPI와 같은 성능 지표에 어떻게 기여하는지를 측정하였다.

1.2 기대효과

본 프로젝트를 통해 다음과 같은 기대효과를 얻을 수 있다.

[프로세서 마이크로아키텍처 이해 향상]

실험을 통해 Issue width, ALU 수, Branch Prediction, Cache 구조가 서로 어떻게 상호작용하며 전체 파이프라인 성능을 결정하는지 명확히 이해하게 된다.

[벤치마크별 프로그램 특성 분석 능력 강화]

각 프로그램의 메모리 접근 패턴, 분기 패턴, 연산 집중도가 서로 다르다는 점을 체감함으로써, 단일 설계가 모든 프로그램에 최적일 수 없다는 사실을 확인할 수 있다.

[구성 요소 간 상호작용 효과 확인]

Cache 구조만 좋아서는 성능이 오르지 않고, ALU만 늘려도 병목이 해결되지 않으며, Branch Predictor·Scalar·Cache가 서로 영향을 주는 것을 확인한다. 즉, 아키텍처는 개별 구성 요소의 단순 합이 아니라 복합적 최적화 문제임을 이해하고 해결 방안에 대해 고민할 수 있다.

1.3 Sim-Scalar 및 벤치마크 프로그램

Sim-Scalar는 Superscalar 기반 마이크로아키텍처의 동작을 cycle-accurate 방식으로 시뮬레이션하는 도구로, Issue·Dispatch·Execute·Commit 단계까지의 내부 pipeline 동작을 상세하게 모델링하여 성능을 분석할 수 있다.

본 프로젝트에서는 SPEC2000 Benchmark 중 다음 네 가지를 활용했다.

프로그램	설명
gcc	C 언어를 다른 형태의 코드로 변환하는 컴파일러로, 주요 단계인 파싱(Parsing), 중간 코드 생성, 최적화, 코드 생성(Code Generation) 진행 정수 연산(Integer ALU 연산) 비중이 매우 높음 분기(branch)가 다량 발생
Go	바둑 게임의 착수(search), 판정 평가(evaluation), 게임 상태 업데이트를 반복 수행하는 프로그램 배열·보드 상태에 대한 반복 접근이 많아, 메모리 Load/Store 비중 높음

	인접 위치 탐색, 주변 패턴 검사 등 공간적 지역성(Spatial locality)이 강하게 나타남
mcf	네트워크 플로우(Network Flow) 최적화 프로그램 그래프(edge, node)에 대한 반복 탐색과 비용 계산을 수행 포인터 기반 연결 구조 탐색이 많아, 메모리 접근 패턴이 불규칙(irregular) 데이터가 넓은 메모리 범위에 분산됨 → Cache miss가 빈번
gzip	Deflate 알고리즘 기반으로 문자열을 탐색하고 패턴을 치환하여 데이터를 압축하는 프로그램 문자열 탐색(String search), 연속된 바이트를 반복적으로 읽고 비교하는 공간적 지역성 매우 높음 루프 기반 구조가 많아 정수 연산 집중 Huffman Tree 구성 등에서 분기 패턴 존재

2 주요 Configuration 분석

2.1 Scalar

Scalar(한 사이클에 동시에 처리 가능한 instruction 개수)는 프로세서의 병렬 실행 능력을 결정하는 핵심 요소이며, SimpleScalar에서는 다음 세 가지 parameter가 이를 직접적으로 좌우한다.

(1) [Fetch : ifqsize - Instruction Fetch Queue Size]

역할 : fetch : ifqsize는 프론트엔드가 가져온 instruction을 임시로 저장하는 queue의 크기를 의미한다. 이 queue는 decode 단계에 충분한 instruction을 공급하기 위한 버퍼 역할을 한다.

○ 성능에 미치는 영향 :

- queue가 너무 작으면 : decode 단계에서 instruction 부족이 발생 superscalar 폭(decode/issue 폭)이 넓어도 활용 불가 → IPC 저하

- queue가 충분히 크면 : 메모리 지연이나 branch로 인해 fetch가 일시적으로 끊겨도 decode 단계는 계속 instruction을 소모할 수 있음. 즉, front-end의 공급 능력을 보완하여 병렬성을 높여줌

- Scalar와의 연결고리 : issue 폭을 2개, 4개로 늘리더라도 fetch queue가 작으면 병렬성을 전혀 활용하지 못하고 사실상 scalar=1처럼 동작. 따라서 ifqsize는 superscalar CPU에서 front-end bandwidth 병목을 제거하는 역할

(2) [Decode : width - Decode 폭]

역할 : 한 사이클에 decode 가능한 instruction의 개수를 의미하며, superscalar의 이론적 최대 IPC를 결정하는 첫 번째 관문이다.

○ 성능 영향 :

- decode 폭 = 1 → scalar 프로세서, decode 폭 = 2, 4 → superscalar 구조

- decode 폭이 넓을수록 병렬 execution이 가능하지만 다음 조건들이 갖춰져야 실제 IPC 증가 :

- fetch 단계가 충분히 많은 instruction을 공급할 것.

- issue 폭이 decode 폭 이상일 것.

- dependency가 적은 workload일 것.

- Scalar와의 연결고리 : decode 폭이 넓어도 issue 폭이 좁으면 issue 단계에서 병목 발생 decode 폭이 issue 폭보다 작으면 decode 단계가 전체 병렬성의 상한선이 됨. 즉, decode는 superscalar 폭의 최소 상한

(3) [Issue : width - Issue 폭]

○ 역할 : Scheduler가 한 사이클 동안 실제 실행 유닛에 발행할 수 있는 instruction의 최대 개수이다. 이는 superscalar CPU의 실제 IPC를 가장 직접적으로 결정하는 파라미터이다.

○ 성능 영향

- issue 폭이 커질수록 ILP를 활용할 가능성이 증가
- 하지만 실제 성능 향상은 다음 조건에 의해 제한됨 :

- decode 폭이 충분히 클 것
- fetch queue가 충분할 것
- dependency chain이 짧을 것
- L1/L2 cache miss가 적을 것

따라서 issue 폭을 단독으로 늘리는 것은 효과가 제한적이며, 전체 front-end 구성과 함께 고려해야 한다.

○ Scalar와의 연결고리

■ $\text{decode:width} \geq \text{issue:width} \geq \text{commit:width}$ 를 만족하지 않으면 superscalar 구조로 동작하지 않음. issue 폭이 넓어도 fetch queue가 부족하거나 decode 폭이 좁거나 FP/ALU 유닛 개수가 부족하면 → 실효 IPC는 scalar 수준에 머무르게 됨.

2.2 Functional Unit

Functional Unit은 프로세서의 파이프라인 단계 중 Execute(실행) 단계에서 실제 데이터의 연산(덧셈, 곱셈, 논리 연산 등)을 수행하는 하드웨어 유닛이다.

Instruction Queue(RUU 등)에서 준비된 명령어들은 사용가능한 Functional Unit(FU)이 있어야만 실행될 수 있다. 만약 실행해야 할 명령어는 많은데 해당 연산을 처리할 FU가 부족하다면, 명령어는 FU가 비워질 때까지 기다려야 한다. 이를 '구조적 해저드'라고 하며 성능 저하의 주원인이 된다.

Simplescalar의 baseline.cfg에서 res:로 시작하는 파라미터는 해당 기능을 수행할 수 있는 하드웨어 유닛의 개수를 의미한다.

(1) res : ialu (Integer Arithmetic Logic Unit)

정수 연산 장치의 개수이다. 정수 덧셈(ADD), 뺄셈(SUB), 논리 연산(AND, OR, XOR), 시프트(SHIFT) 등의 기본적인 정수 처리를 담당한다. 또한 메모리 접근을 위한 주소 계산(Address Calculation)도 보통 이 유닛에서 수행되거나 별도의 주소 생성 유닛(AGU)이 없을

분석 및 중요성 :

대부분의 프로그램에서 가장 빈번하게 발생하는 연산이다. (루프 카운트 제어, 인덱스 계산 등) 이 개수가 부족하면 가장 흔한 명령어들이 병목 현상을 겪게 되어 IPC 급격히 떨어질 수 있다. Issue Width가 넓어도, ialu가 1개라면 한 사이클에 1개의 정수 연산만 처리 가능하므로 슈퍼스칼라의 이점을 살릴 수 없다.

(2) **res : imult (Integer Multiplier/Divider)**

정수 곱셈 및 나눗셈 장치의 개수이다. 정수 곱셈(MUL), 나눗셈(DIV) 등을 담당한다.

분석 및 중요성 :

ALU 연산에 비해 Latency(실행 사이클)가 길다. 빈도는 ALU보다 낮지만, 연산 시간이 길기 때문에 한 번 점유되면 오랫동안 해당 유닛을 사용할 수 없다. 따라서 연속적인 곱셈/나눗셈이 나오는 코드에서는 이 유닛의 개수가 성능에 큰 영향을 미친다.

(3) **res : fpalu (Floating Point Arithmetic Logic Unit)**

부동소수점(실수) 연산 장치의 개수이다. 부동소수점 덧셈(FADD), 뺄셈(FSUB), 비교(FCMP), 변환(CVT) 등을 수행한다.

분석 및 중요성 :

과학 계산이나 그래픽 처리와 같이 실수를 많이 다루는 프로그램에서 중요하다. 이번 과제의 벤치마크인 gcc, go, mcf, gzip 은 대표적인 Integer Benchmark(정수형 벤치마크)들이다. 따라서 이 값의 변화가 성능에 미치는 영향은 ialu에 비해 미미할 것으로 예상할 수 있다.

(4) **res : fpmult (Floating Point Multiplier/Divider)**

부동소수점 곱셈 및 나눗셈 장치의 개수이다. 부동소수점 곱셈(FMUL), 나눗셈(FDIV), 제곱근(SQRT) 등을 수행한다.

분석 및 중요성 :

가장 복잡하고 긴 Latency를 가지는 연산 장치 중 하나이다. fpalu와 마찬가지로 Integer Benchmark 위주의 실험에서는 사용 빈도가 낮아 성능 영향도가 낮을 수 있다.

(5) **Functional Unit과 슈퍼스칼라와의 관계**

Issue : width를 통해 한 번에 많은 명령어를 내보내게 할 수 있지만, 이를 실제로 처리할 res : ialu

등의 FU 개수가 그보다 적으면 resource conflict (자원 충돌)이 발생해서 파이프라인이 STALL 될 수 있다.

(6) 벤치마크 특성 반영

이번 과제에 사용되는 gcc, gzip 등은 정수 연산 위주의 벤치마크이므로, res : fpalu나 res : fpmult보다는 res : ialu의 개수가 성능 향상에 지배적인 영향을 미칠 것으로 예상된다.

2.3 Branch Predictor

Branch Predictor는 분기 명령어가 실제로 실행되기 전에 해당 분기가 taken인지 not-taken인지 미리 예측해 파이프라인이 멈추지 않도록 하는 하드웨어이다.

분기 결과를 기다리면 파이프라인이 중단되어 성능이 크게 떨어지기 때문에, 프로세서는 예측된 경로의 명령어를 먼저 가져와 처리한다. 예측이 맞으면 그대로 이어서 실행되지만, 틀리면 잘못 가져온 명령어를 폐기하는 페널티가 발생한다.

예측 방식은 고정형과 적응형으로 나뉘며, not-taken과 taken 같은 고정형 방식은 단순하지만 코드 패턴에 따라 성능 편차가 크다. 반면 1-bit, 3-bit와 같은 적응형 방식은 과거 분기 결과를 기반으로 학습하여 더 높은 정확도를 제공하며, 반복문과 조건문이 섞인 실제 프로그램에서 가장 효과적이다.

(1) Not-taken Predictor

Not-taken 예측기는 모든 분기를 항상 not-taken으로 간주하는 가장 단순한 방식이다. 구조가 매우 간단하여 예측 오버헤드가 거의 없고, 분기 없이 순차적으로 실행되는 코드에서는 비교적 정확한 예측을 제공한다. 그러나 반복문에서는 대부분의 분기가 taken으로 발생하기 때문에 예측이 지속적으로 틀리게 된다. 이로 인해 루프가 많은 프로그램에서는 성능 저하가 크며, 전체적으로는 특별한 패턴이 없는 단순 코드에서만 제한적으로 좋은 성능을 보인다.

(2) Taken Predictor

Taken 예측기는 모든 분기를 무조건 taken으로 판단한다. 이 방식은 반복문(loop)의 backward branch를 매우 잘 맞추기 때문에 루프가 많은 프로그램에서는 not-taken 방식보다 높은 예측률을 보인다. 하지만 조건문이 많고 분기의 방향이 자주 바뀌는 일반적인 프로그램에서는 오차가 늘어나는 편이다. 따라서 loop 중심의 구조에서는 안정적인 성능을, 분기 패턴이 다양하거나 복잡한 코드에서는 성능 손실을 보여 일관성이 떨어진다.

(3) 1-bit Predictor

1-bit 예측기는 각 분기마다 이전 분기 결과만 저장하여 다음 예측에 반영하는 단순 적응형 방식이다. 동일한 방향의 분기가 반복되는 패턴에 대해서는 빠르게 적응하여 taken 또는 not-taken 방향을 따라가며, 고정형 예측기보다 전반적으로 나은 정확도를 제공한다. 그러나 반복문에서 심각한 단점이 존재한다. 루프의 마지막 반복에서 not-taken이 발생하면 예측 비트가 즉시 뒤집히고, 다음 반복의 첫 taken 분기를 틀리게 예측하게 된다. 이로 인해 반복문 경계에서 반복적으로 예측 실패가 발생하며, 그 결과 loop-heavy 프로그램에서는 성능 향상이 제한적이다.

(4) **3-bit Predictor**

3-bit 예측기는 포화 카운터를 사용하여 taken과 not-taken 사이를 여러 단계로 표현하고, 분기 방향이 잠깐 바뀌더라도 예측 상태가 쉽게 뒤집히지 않도록 설계되어 있다. 이러한 특성 덕분에 반복문에서 마지막 한 번의 not-taken이 발생해도 카운터가 바로 반대 방향으로 이동하지 않으며, 다음 반복에서도 정확한 taken 예측을 유지할 수 있다. 또한 조건문이 반복되는 일반적인 코드에서도 예측이 빠르게 안정화되며, 잡음이나 일시적인 패턴 변화에도 잘 견딘다. 결과적으로 네 가지 방식 중 가장 높은 예측 정확도를 제공하며, 오버헤드 역시 현대 시스템에서 충분히 작은 수준이다.

Not-taken과 Taken 방식은 단순하지만 코드 패턴에 따라 성능이 크게 흔들리는 단점이 있다. 1-bit predictor는 적응형이지만 반복문에서 구조적 취약점이 존재한다. 반면 3-bit 포화 카운터 방식은 반복문과 일반적인 조건문 모두에서 안정적이고 높은 예측률을 보여 전체적으로 가장 우수한 성능을 제공한다.

2.4 Cache configuration

Cache는 프로세서와 메모리 간의 병목을 완화하는 핵심 구성 요소로, 캐시 구조에 따라 프로그램의 miss rate·메모리 접근 시간·전체 IPC/CPI가 크게 달라진다. 본 프로젝트에서는 SimpleScalar에서 제공하는 4종의 캐시(dl1, dl2, il1, il2) 중 Data Cache 2종을 중심으로 구조와 벤치마크 프로그램에 미치는 영향을 분석하였다.

구성요소	설명
dl1	Data L1 Cache – Load/Store 단계에서 사용
dl2	L2 Cache – L1 miss 시 접근하는 통합 캐시

[DL1 Cache (Data Level-1 Cache)]

DL1은 Load/Store 명령의 가장 첫 번째 캐싱 계층으로, 성능에 미치는 영향이 가장 크다.

(1) Block Size의 영향

Block Size가 크면 공간적 지역성(spatial locality)을 더 잘 활용할 수 있어 연속적 데이터 접근을 갖는 프로그램에서 miss rate 감소하였다.

반대로 Block Size가 너무 크면 불필요한 데이터까지 가져오는 Over-fetch 발생 Set 수 감소로 인해 conflict miss 증가한다, 따라서 프로그램별로 최적 Block Size가 다르게 설정될 수 있다.

(2) Associativity의 영향

1-way(Direct-mapped)는 conflict miss가 발생할 확률이 높다, 2-way, 4-way로 증가하면 conflict miss가 감소하며 IPC가 큰 폭으로 증가한다. 하지만 4-way가 2-way에 비해 하드웨어 복잡도는 증가하는 trade-off의 관계에 있다. 결론적으로 DL1은 프로그램 특성에 따라 2-way 또는 4-

way에서 가장 안정적인 성능을 보였다.

(3) Replacement Policy의 영향

DL1에서는 LRU가 전반적으로 안정적인 성능을 보이며, FIFO·Random 대비 miss rate가 낮은 편으로 측정되었다.(시간적 지역성 기반) 특히 데이터 재사용률이 높은 프로그램에서는 LRU가 매우 유리하다. 단, DL1의 block/associativity 설정이 miss rate에 영향을 많이 미치는 만큼 교체 정책의 차이는 크게 두드러지지 않았다.

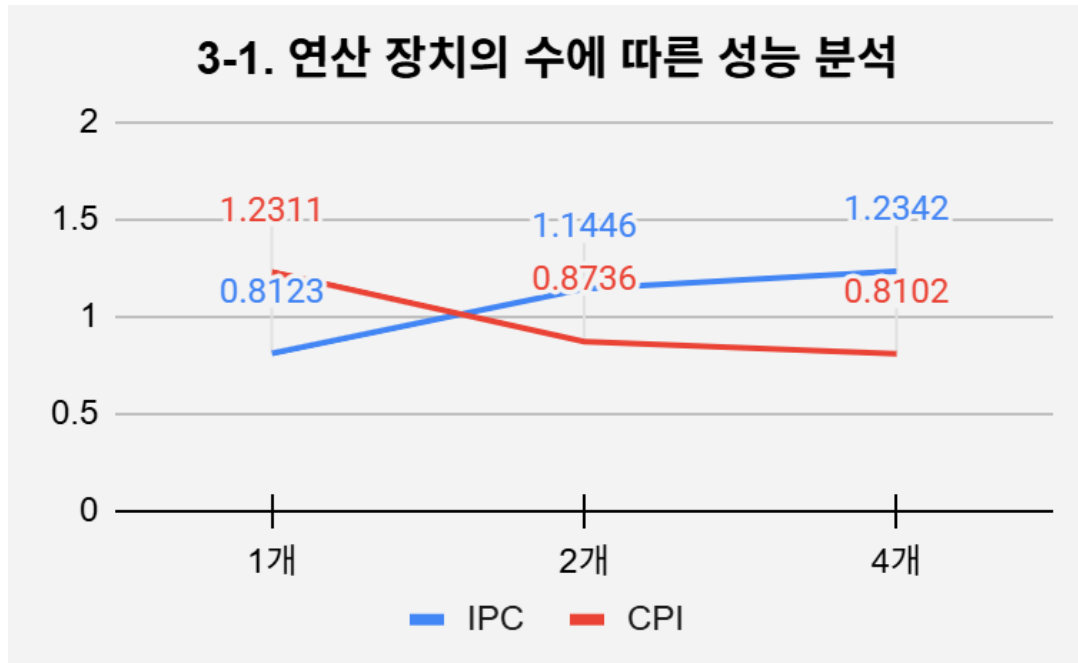
[DL2 (Unified L2 Cache)]

L2는 L1 miss 시 접근하는 계층으로, 메인 메모리 접근을 줄이는 역할을 한다.

3 Processor 변화에 따른 성능 분석

3.1 ALU의 개수 변화

3.1.1 gcc



[데이터 요약]

ALU 1개 : IPC 0.8123 / CPI 1.2311

ALU 2개 : IPC 1.1446 / CPI 0.8736

ALU 4개 : IPC 1.2342 / CPI 0.8102

[분석 및 해석]

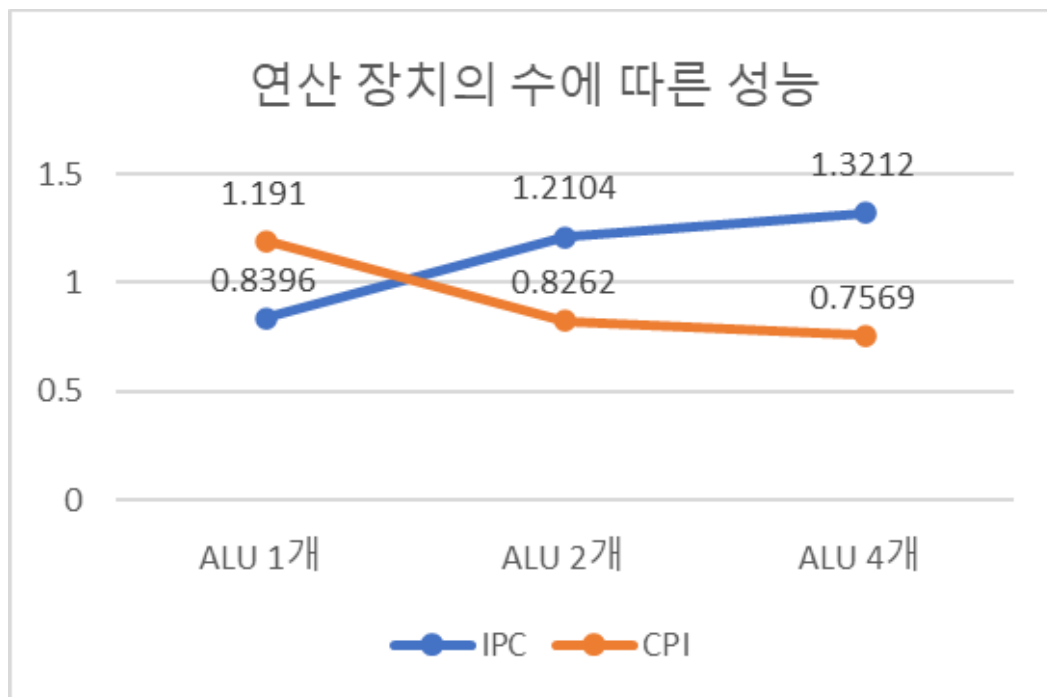
연산 장치(ALU)의 개수가 증가할수록 IPC는 상승하고 CPI는 감소하여, ALU가 4개일 때 가장 높은 성능을 보였다. 1개에서 2개로 확장할 경우 IPC는 약 **41% 증가**, CPI는 **29% 감소**하며 가장 큰 폭의 향상이 나타났다. 이는 동시에 처리할 수 있는 산술·논리 연산의 수가 증가하면서 병렬화가 활발히 이루어졌기 때문이다.

그러나 2개에서 4개로 확장할 때는 IPC 증가율이 7.8%로 크게 줄며 성능 향상이 포화되는 경향을 보인다. 이는 issue 단계가 증가한 유닛을 모두 활용하지 못하거나, 프로그램 내부의 독립 명령어 수가 충분하지 않아 병렬 처리 이득이 제한되기 때문이다. 즉, ALU 수 증가는 분명한 성능 향상을 제공하지만, 일정 수준을 넘어서면 한계 효용이 감소한다.

추가로 ALU가 많아질수록 하드웨어적 비용도 증가한다. 연산 장치 수가 늘어나면 칩

면적이 커지고 배선 구조가 복잡해지므로 클럭 향상에 제약이 생길 수 있다. 또한 모든 ALU가 활성화될 때 전력 소모와 전력 밀도가 증가해 발열 관리가 어려워지며, 이에 따라 냉각 비용과 전력 예산 부담도 커진다. 결국 ALU 수를 늘리는 것은 성능 향상에는 효과적이지만, 전력·면적·발열 측면의 비용과 균형을 고려해야 한다.

3.1.2 Go



[데이터 요약]

ALU 1개 : IPC 0.8396 / CPI 1.191

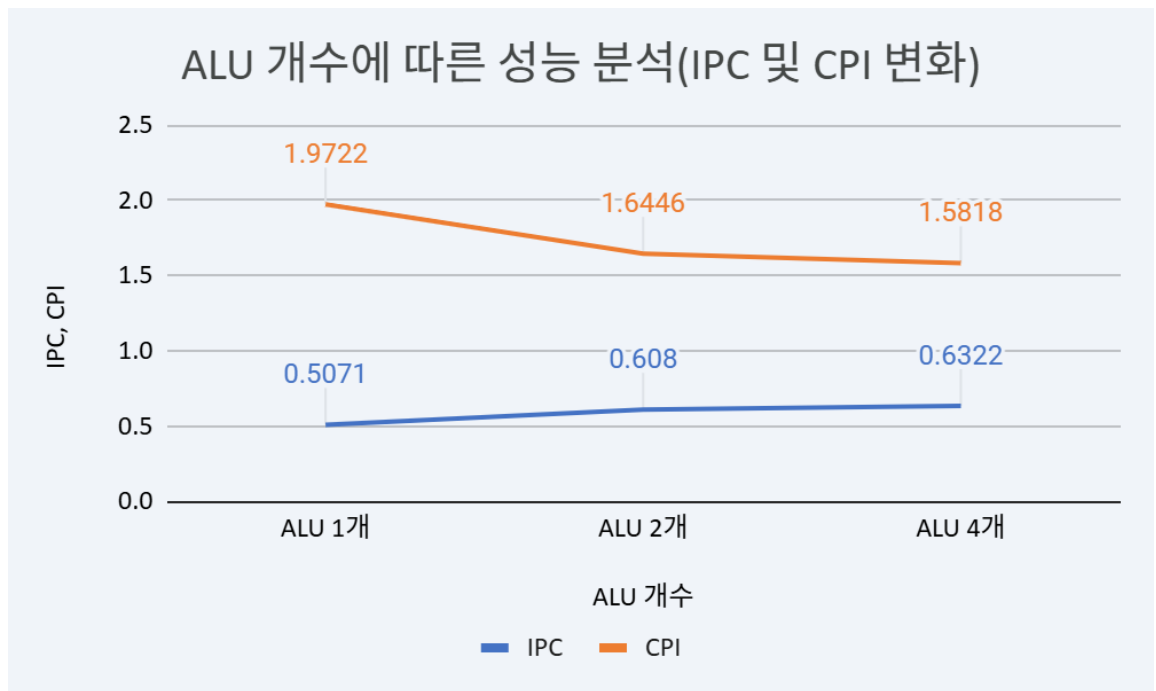
ALU 2개 : IPC 1.2104 / CPI 0.8262

ALU 4개 : IPC 1.3212 / CPI 0.7569

[분석 및 해석]

go는 한 번에 많은 수를 읽어야 하므로 병렬로 처리할 수 있는 명령어의 수가 많다. ALU가 1개일 때는 병목이 심하지만, 2개로 늘렸을 때 IPC가 약 44% 급증하는 것을 볼 때, 다수의 정수 연산기가 성능 확보에 필수적임을 알 수 있다.

그러나 ALU의 개수를 높였을 때에 물리적인 연산 유닛이 늘어나는 만큼 칩의 면적을 많이 차지하게 되며, 이는 제조 비용 상승으로 이어지는 단점이 있다. 또한 사용하지 않는 순간에도 누설 전류가 발생하며, 4개가 동시에 가동될 때의 동적 전력소모가 커지게 된다. 그리고 연산 유닛이 많아지면 레지스터 파일과 데이터를 주고받는 데이터 버스와 포트가 복잡해진다.



[데이터 요약]

ALU 1개: IPC 0.5071 / CPI 1.9722

ALU 2개: IPC 0.6080 / CPI 1.6446

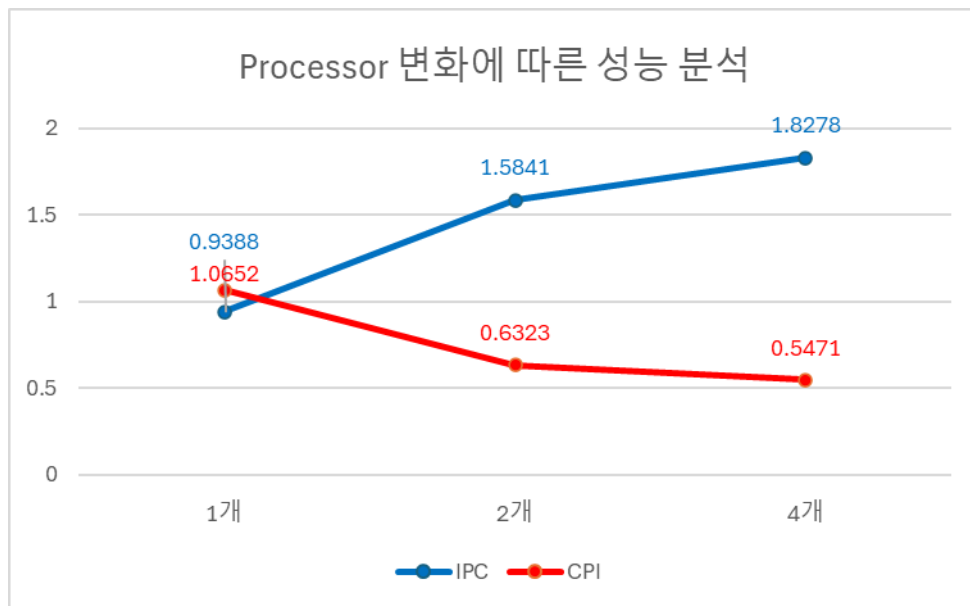
ALU 4개: IPC 0.6322 / CPI 1.5818

[분석 및 해석]

성능 향상 확인: ALU의 개수가 1개에서 4개로 증가함에 따라 IPC는 약 **24.6% 증가**(0.5071 → 0.6322)하였으며, 이에 따라 CPI는 감소하는 경향을 보였다. 이는 동시에 처리할 수 있는 연산 자원이 늘어나 구조적 해저드(Structural Hazard)가 완화되었기 때문이다.

ALU가 1개에서 2개로 늘어날 때의 성능 향상 폭(IPC 약 0.1)이, 2개에서 4개로 늘어날 때 (IPC 약 0.024)보다 훨씬 크다.

원인: ALU 자원이 충분하더라도, 프로그램 내 명령어 간의 데이터 의존성(RAW Hazard)으로 인해 병렬로 실행할 수 있는 명령어의 수에 한계가 존재하기 때문이다. 즉, 4개의 ALU가 있어도 동시에 실행 가능한 독립적인 명령어가 부족하면 유휴 자원이 발생한다.



[데이터 요약]

ALU 1개: IPC 0.9388 / CPI 1.0652

ALU 2개: IPC 1.5841 / CPI 0.6323

ALU 4개: IPC 1.8278 / CPI 0.5471

[분석 및 해석]

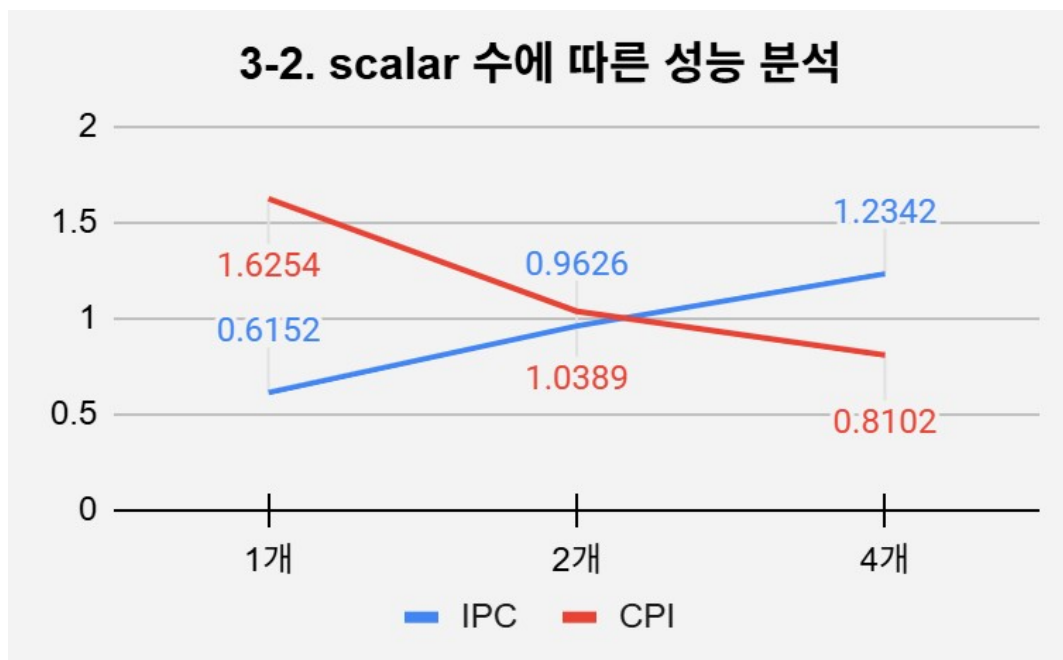
결과 : ALU 개수가 클수록 IPC가 증가했으며 1개와 4개일 때 차이는 **약 2배정도** 차이가 난다.

이유 : Scalar가 4로 고정되어 있는 경우, 이를 처리할 수 있는 연산 장치의 수 차이가 곧 성능에 직접적인 영향을 미친다. Scalar가 2인 경우 ALU 개수가 4개라면 2개는 작업이 없어 ALU가 놀고 있는 경우가 발생한다. 하지만 4개로 고정되어 있을 때 4개에 대한 처리를 바로바로 해줄 수 있는 경우는 ALU가 4개일 때뿐 이므로 4개일 때 가장 좋은 성능 결과를 확인할 수 있었다.

단점 : ALU 개수를 늘려주면 속도가 향상되지만, 추가 ALU 개수만큼 칩 면적이 넓어지고 더 많은 전력을 소모한다. 또한 gzip 특성 상 분기 명령이 다른 명령에 비해 많기 때문에 적절한 예측 없이 연산만 진행하는 것은 오히려 자원이 낭비되는 경우로 이어질 수 있다.

3.2 Scalar width 변화

3.2.1 gcc



[데이터 요약]

Scalar 1개 : IPC 0.6152 / CPI 1.6254

Scalar 2개 : IPC 0.9626 / CPI 1.0389

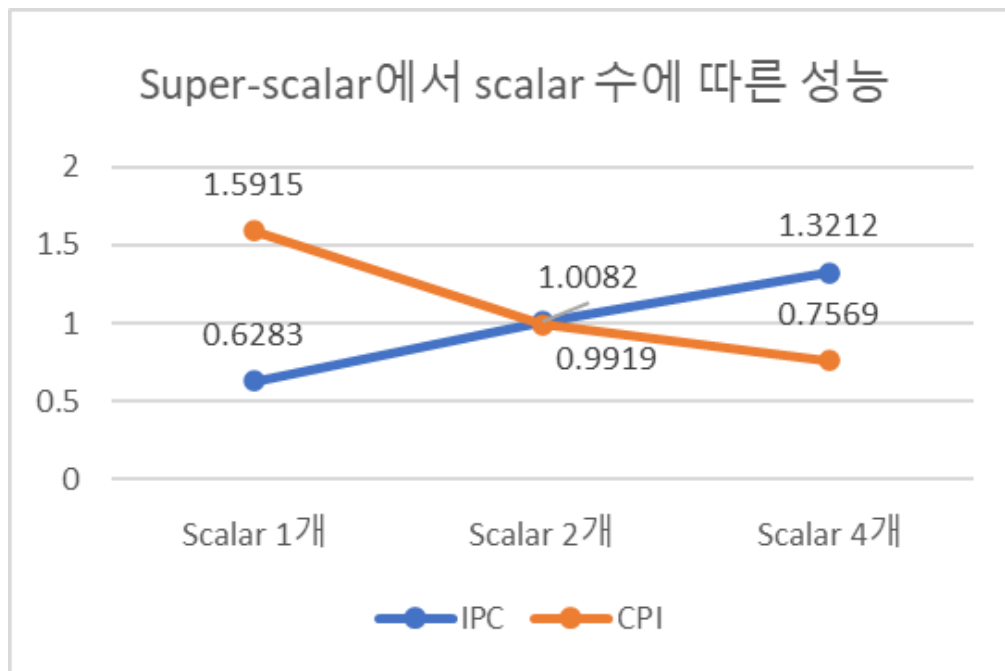
Scalar 4개 : IPC 1.2342 / CPI 0.8102

[분석 및 해석]

Scalar(명령어 발행 폭)를 늘리면 IPC는 증가하고 CPI는 감소하여 4-scalar에서 가장 높은 성능을 보였다. 1개에서 2개로 확장했을 때 **IPC는 56% 증가, CPI는 36% 감소**하며 가장 큰 폭의 향상이 나타났다. 이는 한 사이클에 발행 가능한 명령어가 증가해 병렬 실행이 크게 강화되기 때문이다.

하지만 2개에서 4개로 확장할 때 IPC 증가율은 28%로 줄어들어 효과가 점차 감소했다. 프로그램 내부의 독립 명령어 수가 발행 폭보다 적거나, 분기·메모리 접근 병목으로 인해 issue 단계가 완전히 활용되지 못하면서 병렬화 효과가 포화되는 것이다.

Scalar가 증가할수록 issue 로직, rename 로직, scoreboard 등 제어 회로가 커지면서 회로 복잡도와 전력 소모가 증가한다. 또한 파이프라인의 전반적 길이와 타이밍 여유가 줄어들어 고클록 설계에 부담이 생긴다.



[데이터 요약]

Scalar 1 : IPC 0.6283 / CPI 1.5915

Scalar 2 : IPC 1.0082 / CPI 0.9919

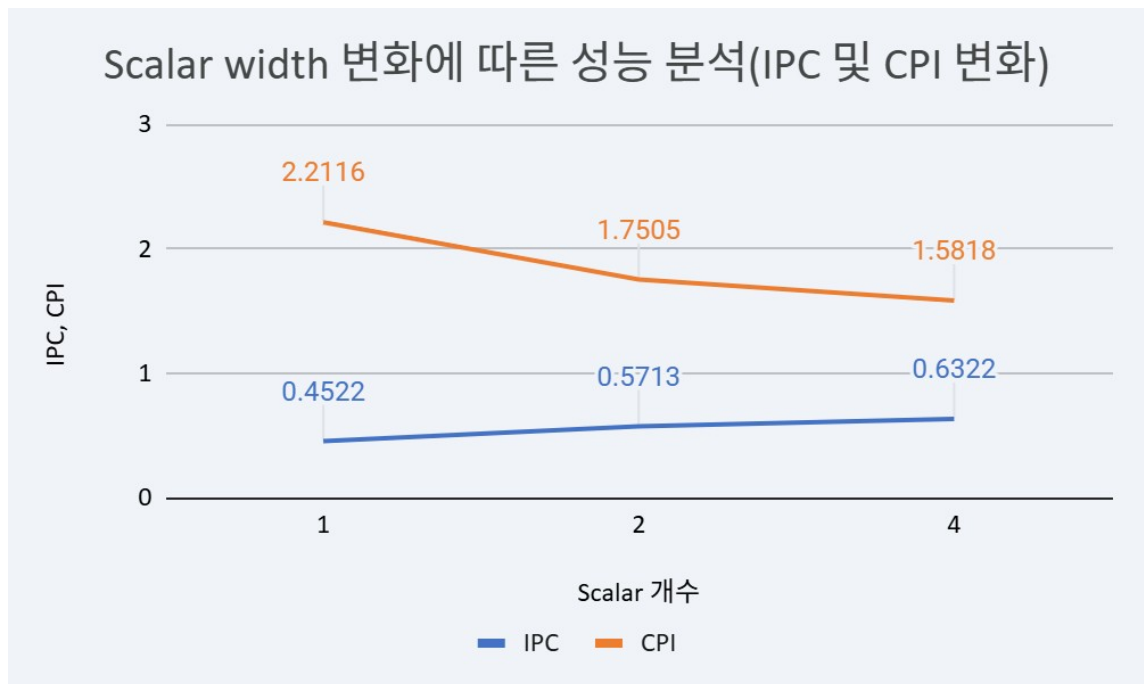
Scalar 4 : IPC 1.3212 / CPI 0.7569

[분석 및 해석]

1-issue는 ALU가 아무리 많아도 명령어를 공급하지 못해 성능이 가장 낮았다. Width를 4개로 늘렸을 때 IPC가 2배 이상 상승한 점은 go가 높은 대역폭의 프론트 엔드를 요구한다는 것을 증명한다.

그러나 scalar를 높였을 경우 명령어 간의 의존성을 검사하는 하드웨어 로직은 Issue Width의 제공에 비례하여 복잡해지는 단점이 있다. 4-issue는 1-issue 대비 하드웨어 복잡도가 훨씬 크다. 또한 의존성 검사 로직이 복잡해지면 신호 처리 시간이 길어져, CPU의 동작 주파수를 높이기 어려워질 수 있다.

즉, IPC는 높지만 클럭 속도가 낮아져 실제 실행 시간은 기대만큼 안 줄어들 수도 있다.



[데이터 요약]

Scalar 1 : IPC 0.4522 / CPI 2.2116

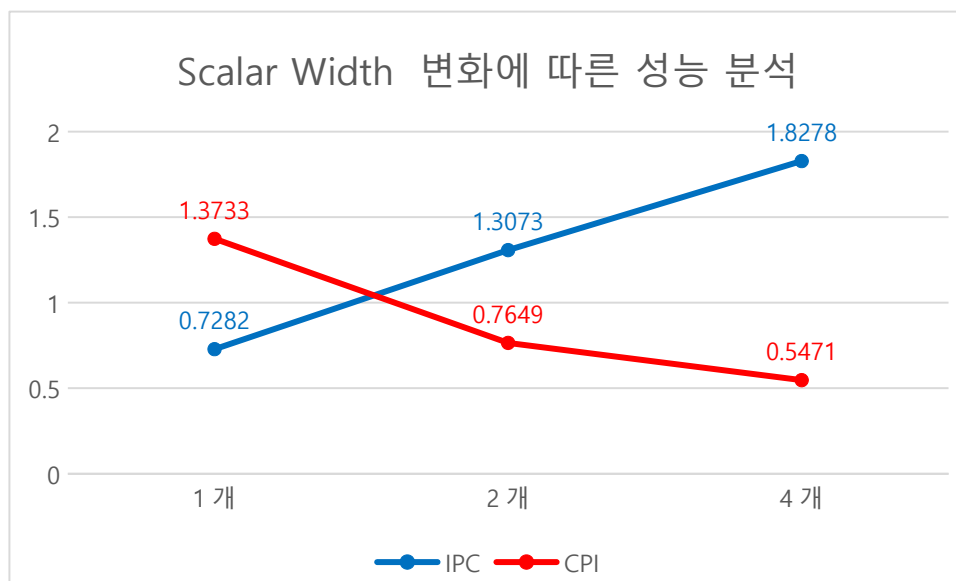
Scalar 2 : IPC 0.5713 / CPI 1.7505

Scalar 4 : IPC 0.6322 / CPI 1.5818

[분석 및 해석]

대역폭 확장에 따른 선형적 개선 : Scalar Width가 증가함에 따라 IPC가 꾸준히 상승했다. 특히 Width 1 대비 Width 4에서 **IPC가 약 39.8% 향상**되었다.

병목 현상의 해소 : Scalar 1 구조에서는 아무리 연산 자원이 많아도 한 번에 하나의 명령어만 처리할 수 있어 병목이 발생한다. Width를 늘림으로써 Superscalar 구조의 이점을 활용하여 프로세서의 성능을 끌어올렸다.



[데이터 요약]

Scalar 1 : IPC 0.7282 / CPI 1.3733

Scalar 2 : IPC 1.3073 / CPI 0.7649

Scalar 4 : IPC 1.8278 / CPI 0.5471

[분석 및 해석]

결과 : Scalar 수가 1, 2, 4로 커질 때마다 IPC가 커지며 clk당 실행한 명령어 개수가 증가하였다. 4는 1대비 무려 **2배 이상** 차이가 발생하였다.

이유 : 한 사이클에서 처리 가능한 명령어 수 증가는 동시에 여러 명령어를 수행하는 것과 동일하다. Scalar 수가 1이라면 한 clock에 한 명령어만 진행하지만, 4라면 한 clock 신호에 4개의 명령어가 진행되는 것과 같다.

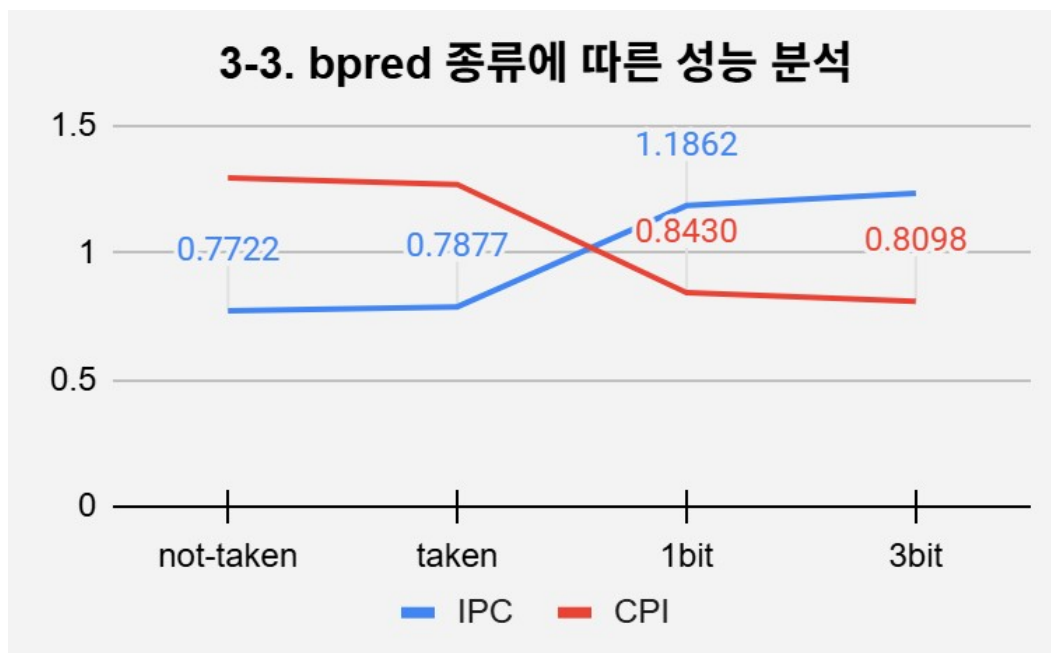
한 clock의 한 명령어만 진행한 경우 속도는 현저히 느려져 1clock당 한 개의 명령어 수행도 어려움을 보이고 있다. 이는 분기, load 등 메모리 접근과 control hazard로 수행 중인 명령어를 flush하고 다시 연산을 진행하기 때문에 발생하는 현상이다.

하지만 4개를 동시에 처리하는 경우, gzip에서 적절한 분기 예측만 된다면 성능이 현저히 향상될 수 있다.

단점 : 더욱 많은 폭의 명령을 한번에 실행하게 되어도 데이터 의존성과 분기, 메모리 지연으로 인해 향상되는 것에 한계가 보다 명확하고 잘못된 분기를 예측했을 때 더욱 많은 잘못된 명령을 실행하고 flush하는 현상이 발생하며 자원을 낭비하는 경우가 발생할 수 있다.

3.3 Branch predictor 종류

3.3.1 gcc



[데이터 요약]

Not-taken : IPC 0.7722 / CPI 1.2950

Taken : IPC 0.7877 / CPI 1.2694

1-bit : IPC 1.1862 / CPI 0.8430

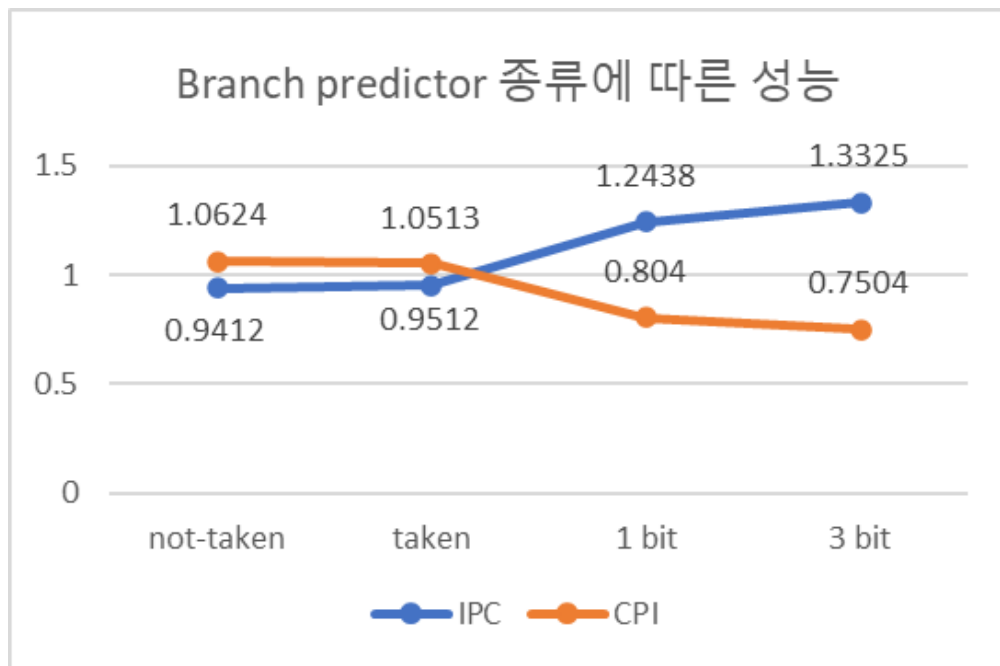
3-bit : IPC 1.2349 / CPI 0.8098

[분석 및 해석]

분기 예측기가 단순 방식에서 학습 기반 방식으로 전환될 때 가장 큰 성능 향상이 나타났다. not-taken에서 taken 변경 시 IPC는 약 2% 증가로 효과가 미미했지만, taken에서 1bit predictor 전환 시 **IPC가 약 50% 증가, CPI는 27% 감소**하며 매우 큰 향상이 있었다. 이는 1bit predictor가 분기 패턴을 학습해 예측률을 크게 높였기 때문이다.

반면 1bit → 3bit predictor 확장 시 IPC 증가율은 4%로 작아, 예측기의 정교함이 일정 수준을 넘으면 성능 기여도가 작아진다는 점을 보여준다.

학습 기반 분기 예측기는 테이블 크기 증가, 히스토리 버퍼 추가 등으로 인해 면적 증가, 전력 소모 증가, 폴트 시 초기 warm-up 필요 등의 부담이 따른다.



[데이터 요약]

Not-taken : IPC 0.9412 / CPI 1.0624

Taken : IPC 0.9512 / CPI 1.0513

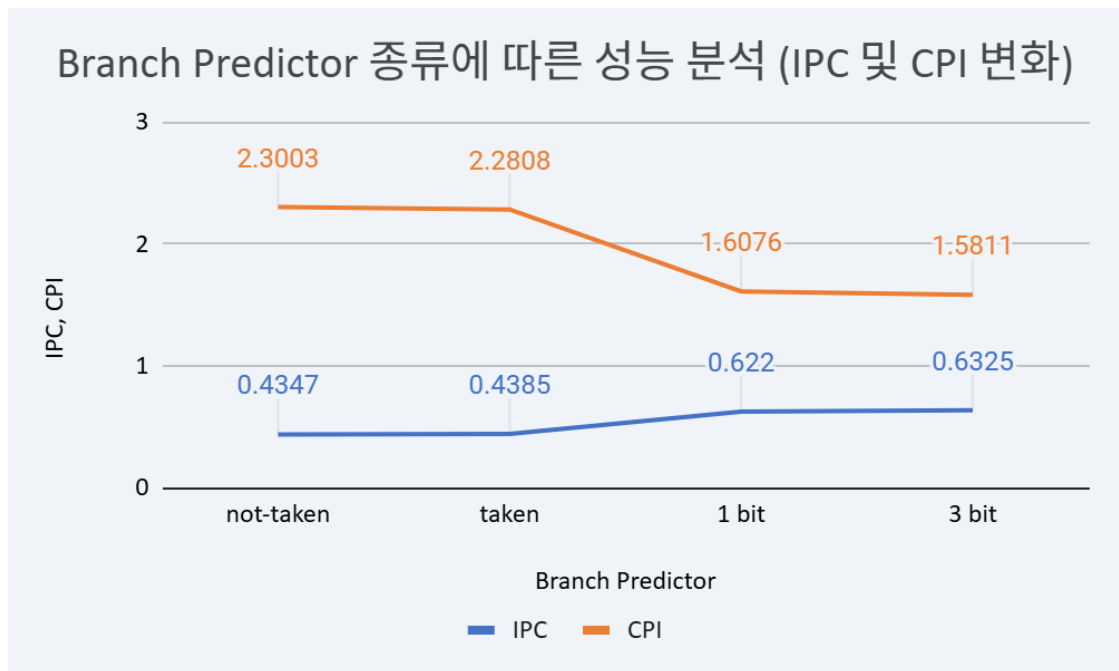
1-bit : IPC 1.2438 / CPI 0.804

3-bit : IPC 1.3325 / CPI 0.7504

[분석 및 해석]

go는 매우 불규칙한 분기 패턴을 가진다. 단순한 정적 예측(Taken/Not-taken)은 성능이 0.9대에 머물렀으나, 과거 이력을 참조하는 동적 예측(1bit)을 도입하자마자 성능이 1.24로 급상승했다. 특히 더 긴 이력을 보는 3bit 예측기에서 최고 성능이 나온 것은 go의 복잡한 로직을 처리하기 위해 정교한 분기 예측이 중요함을 의미한다.

그러나 정교한 예측기를 사용하는 경우 예측 정확도를 높이기 위해 더 큰 패턴 히스토리 테이블(PHT)과 분기 목적지 버퍼(BTB) 메모리가 필요하다는 단점이 있을 수 있다. 또한 예측기가 패턴을 학습하는 데 시간이 걸린다. 프로그램 실행 초반이나 문맥 교환 직후에는 오히려 단순 예측기보다 성능이 떨어질 수도 있다.



[데이터 요약]

Static (Not-taken): IPC 0.4347/ CPI 2.3003

Static (Taken): IPC 0.4385 / CPI 2.2808

Dynamic (1-bit): IPC 0.6220 / CPI 1.6076

Dynamic (3-bit): IPC 0.6325 / CPI 1.5811

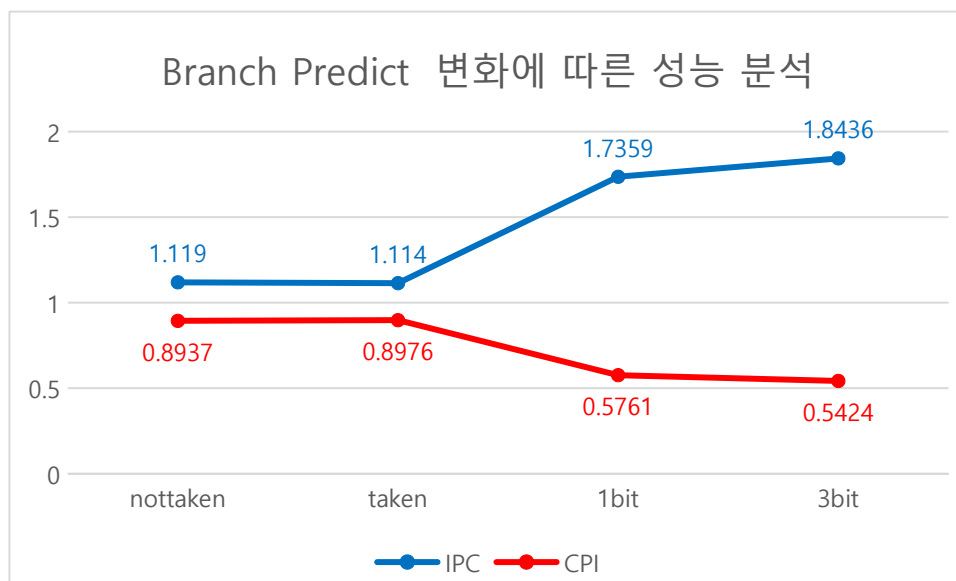
[분석 및 해석]

Static vs Dynamic 예측의 차이 : Not-taken/Taken과 같은 정적 예측 방식은 IPC가 0.43 수준으로 매우 낮다. 이는 분기 예측 실패로 인한 Pipeline Flush가 빈번하게 발생하여 클럭 사이클 낭비가 심함을 의미한다.

반면, 동적 예측(1-bit, 3-bit)을 적용했을 때 IPC가 0.62 이상으로 급격히 상승(약 45% 성능 향상)했다. 이는 런타임에 분기 기록을 활용하여 Control Hazard를 효과적으로 줄였음을 보여준다.

History Bit 수에 따른 차이 (1-bit vs 3-bit) : 1-bit 예측기(0.622)와 3-bit 예측기(0.6325) 간의 성능 차이는 크지 않다.

원인: 해당 벤치마크 프로그램의 분기 패턴이 비교적 단순하여, 1-bit 예측기만으로도 대부분의 분기 패턴을 커버할 수 있거나, 3-bit 카운터가 포화되는 데 걸리는 시간이 오히려 짧은 패턴 변화에 민감하게 반응하지 못했을 가능성이 있다. 하지만 미세하게나마 3-bit가 높은 것은 복잡한 패턴에서의 정확도가 소폭 상승했기 때문이다.



[데이터 요약]

Not-taken : IPC 1.119 / CPI 0.8937

Taken : IPC 1.114 / CPI 0.8976

1-bit : IPC 1.7359 / CPI 0.5761

3-bit : IPC 1.8436 / CPI 0.5424

[분석 및 해석]

결과 : taken과 not-taken에서는 미미한 차이를 보였지만, 1bit, 3bit로 분기 예측을 진행한 경우 IPC 수치가 증가하는 것을 확인할 수 있다.

이유 : 위 원인을 분석하기 위해 예측 성공, 실패 비율을 확인해 보았다. 결과 출력 파일에서 bpred_dir_hits와 misses 수를 비교해 보았다.

bpred_taken.dir_hits	67354415 # total number of direction-predicted hits (includes addr-hits)
bpred_taken.misses	132008984 # total number of misses
bpred_nottaken.dir_hits	67354415 # total number of direction-predicted hits (includes addr-hits)
bpred_nottaken.misses	132008984 # total number of misses
bpred_1bit.dir_hits	175063133 # total number of direction-predicted hits (includes addr-hits)
bpred_1bit.misses	24300266 # total number of misses
bpred_3bit.dir_hits	182884182 # total number of direction-predicted hits (includes addr-hits)
bpred_3bit.misses	16479217 # total number of misses

Taken과 not-taken의 경우 동일한 hit와 miss가 발생했으며 miss 발생 수가 훨씬 많은 것을 확인했고, 1bit의 경우 hit가 대폭 상승하였으며, 3bit의 경우 약 91%까지 상승하였다.

gzip의 경우 문자열 매칭 -> 조건 분기 -> 해시 업데이트 -> 루프 반복/종료 -> 다음 블록

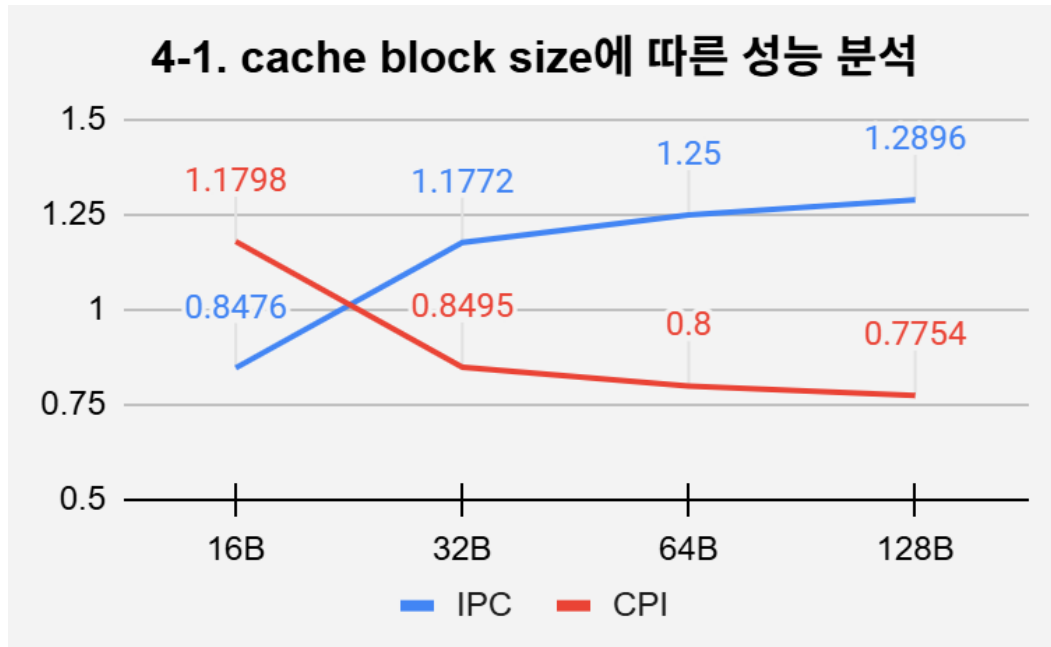
일련의 과정으로 연산이 진행된다. 이때 분기의 비율이 상당히 높고 반복 구조가 많기 때문에 taken, not-taken의 단순 정책보다 최근 결과를 기억하여 taken, not-taken 방향을 학습하기 때문에 90%이상의 예측률이 나올 수 있다. 또한 예측 품질이 좋아지면서 flush 횟수가 감소하고 이로 인해 IPC 성능이 증가함을 알 수 있다.

단점 : 정적 분기 예측보단 동적 분기 예측은 과거 상태를 기록하기 위한 buffer, 공간 등이 추가로 필요하게 되어 복잡도 또는 하드웨어 비용이 증가하게 된다. 1bit의 경우 분기마다 1bit를 필요하지만 3bit의 경우 분기마다 3bit의 FSM을 사용하기 때문에 더 많은 메모리, 설계 복잡도가 증가한다.

4 Cache 변화에 따른 성능 분석

4.1 Block size 변화

4.1.1 gcc



[데이터 요약]

16B : IPC 0.8476 / CPI 1.1798

32B : IPC 1.1772 / CPI 0.8495

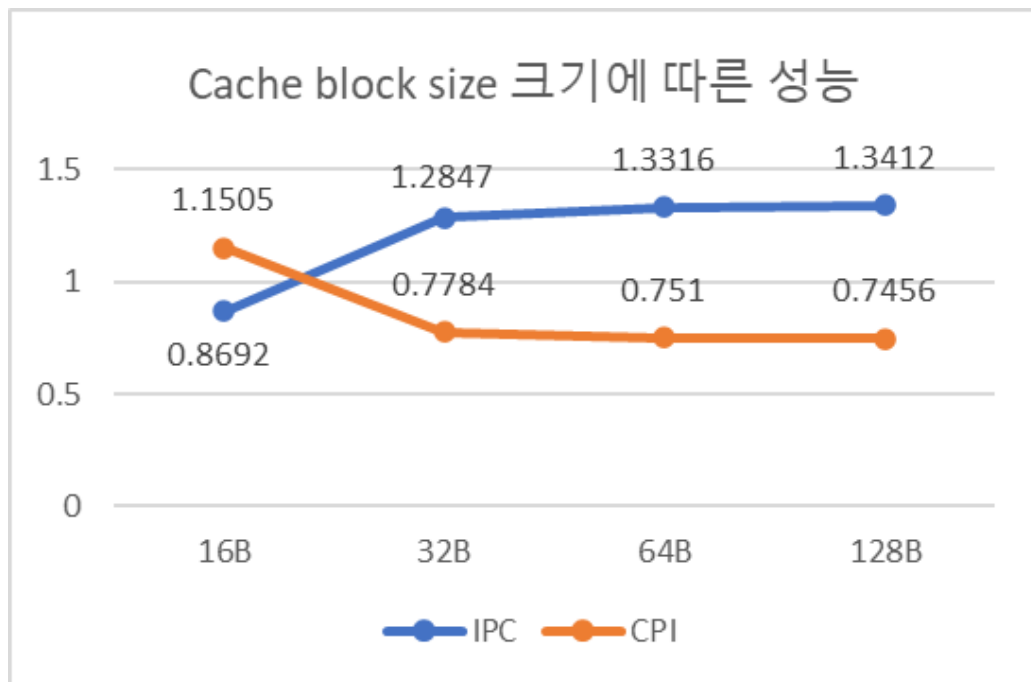
64B : IPC 1.2500 / CPI 0.8000

128B : IPC 1.2896 / CPI 0.7754

[분석 및 해석]

Block size가 16B에서 32B로 증가할 때 IPC는 **38.9% 증가**, **CPI는 27.9% 감소**해 가장 큰 성능 향상을 보였다. 이는 spatial locality가 크게 개선되어 miss가 감소했기 때문이다. 하지만 32B → 64B에서 IPC 증가율은 6%, 64B → 128B에서는 3%로 점점 줄어 성능 향상이 둔화되었다. Block size가 커질수록 불필요한 데이터까지 가져오는 fetch overfetch 현상이 증가해 오히려 비효율을 초래하기 때문이다.

Block size가 커질수록 miss penalty가 커지고, 캐시 line refill 시 메모리 대역폭을 더 많이 소모하며, 캐시 내부 저장 효율도 떨어진다.



[데이터 요약]

16B : IPC 0.8692 / CPI 1.1505

32B : IPC 1.2847 / CPI 0.7784

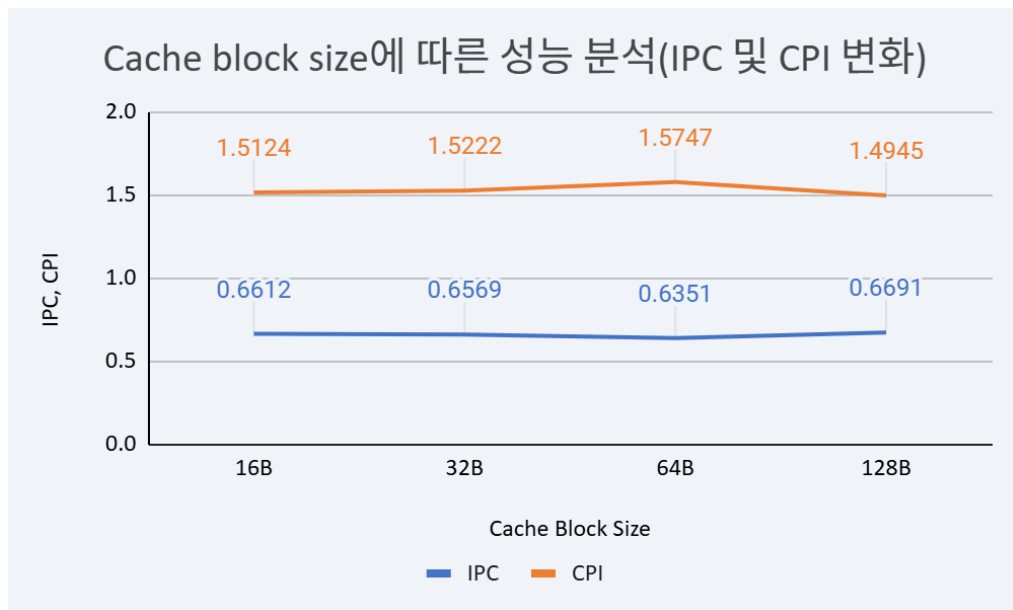
64B : IPC 1.3316 / CPI 0.751

128B : IPC 1.3412 / CPI 0.7456

[분석 및 해석]

Block Size가 커질수록 성능이 지속적으로 향상된다. 특히 16B에서 32B로 갈 때 성능이 폭발적으로 증가했다. 이는 바둑판 데이터가 메모리에 연속적으로 배열되어 있어, 한 번 가져올 때 인접 데이터를 많이 가져오는 큰 블록 크기가 Cache Hit Rate를 높이는 데 결정적임을 보여준다. (높은 공간적 지역성)

그러나 Block Size를 키우는 경우 캐시 미스가 발생했을 때, 메인 메모리에서 가져와야 할 데이터 양이 많아지므로 전송 시간이 길어진다. 만약 가져온 128B 중 딱 4B만 쓰고 나머지를 안 쓴다면(공간적 지역성이 낮은 구간), 대역폭과 시간을 낭비하게 된다. 또한, 당장 필요하지 않은 데이터까지 너무 많이 가져와서, 정작 필요한 유용한 데이터를 캐시에서 밀어낼 위험이 있다.



[데이터 요약]

16B: IPC 0.6612 / CPI 1.5124

32B: IPC 0.6569 / CPI 1.5222

64B: IPC 0.6351 / CPI 1.5747

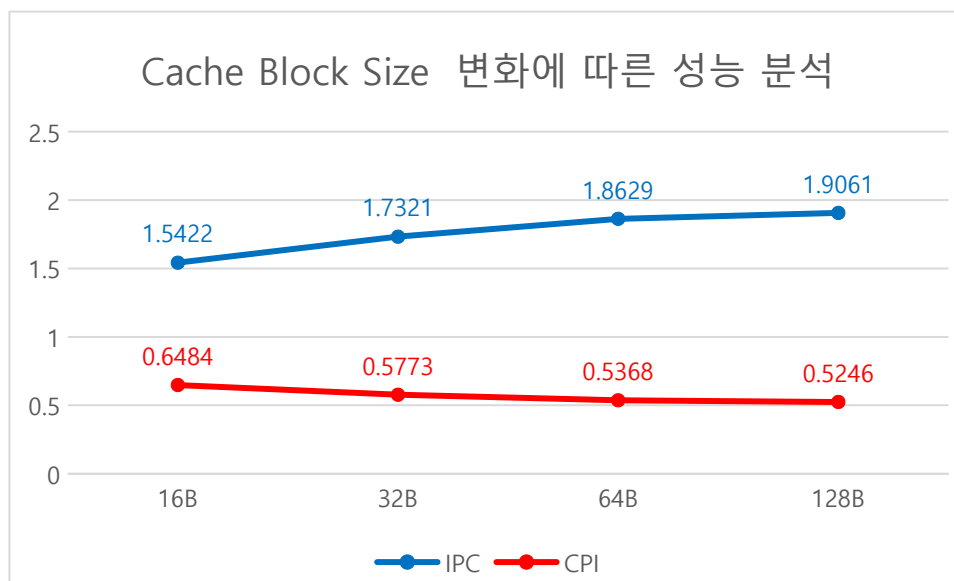
128B: IPC 0.6691 / CPI 1.4945

[분석 및 해석]

공간적 지역성(Spatial Locality)의 효과: 일반적으로 Block Size가 커질수록 한 번에 인접한 데이터를 많이 가져오므로 순차적인 데이터 접근에서 공간적 지역성을 활용하여 성능이 향상된다. **128B에서 가장 높은 IPC(0.6691)를 기록한 것이 이를 증명한다.**

캐시 오염과 페널티 : 흥미로운 점은 16B에서 64B로 갈 때 오히려 성능이 하락(0.6612 -> 0.6351)했다는 점이다. 이는 제한된 캐시 용량 내에서 Block Size만 커지면 캐시 라인의 총개수가 줄어들어 서로 다른 주소의 데이터가 같은 자리를 놓고 다투는 Conflict Miss가 증가했기 때문으로 보인다. 또한 Block이 너무 크면 당장 쓰지 않을 불필요한 데이터까지 가져오는 캐시 오염이 발생하여 유효한 데이터를 밀어낼 수 있다.

결론적으로 Block Size는 무조건 크면 좋은 것이 아니라, 프로그램의 특성과 전체 캐시 크기에 맞춰 적절한 타협점을 찾아야 한다.



[데이터 요약]

16B : IPC 1.5422 / CPI 0.6484

32B : IPC 1.7321 / CPI 0.5773

64B : IPC 1.8629 / CPI 0.5368

128B : IPC 1.9061 / CPI 0.5246

[분석 및 해석]

결과 : block size가 커질수록 IPC가 상승한다.

이유 : gzip의 특징으로 문자열 패턴 탐색과 sliding window 기반 deflate 알고리즘으로 연속적인 메모리 접근 패턴에 우수한 성능을 보인다. 이 말은 공간적 지역성이 매우 우수하다고 말할 수 있다. 실제 cache miss rate를 비교하면 아래와 같다.

```
dl1.miss_rate 0.0533
```

```
dl1.miss_rate 0.0368 # miss rate (i.e., misses/ref)
```

```
dl1.miss_rate 0.0245 # miss rate (i.e., misses/ref)
```

```
dl1.miss_rate 0.0144 # miss rate (i.e., misses/ref)
```

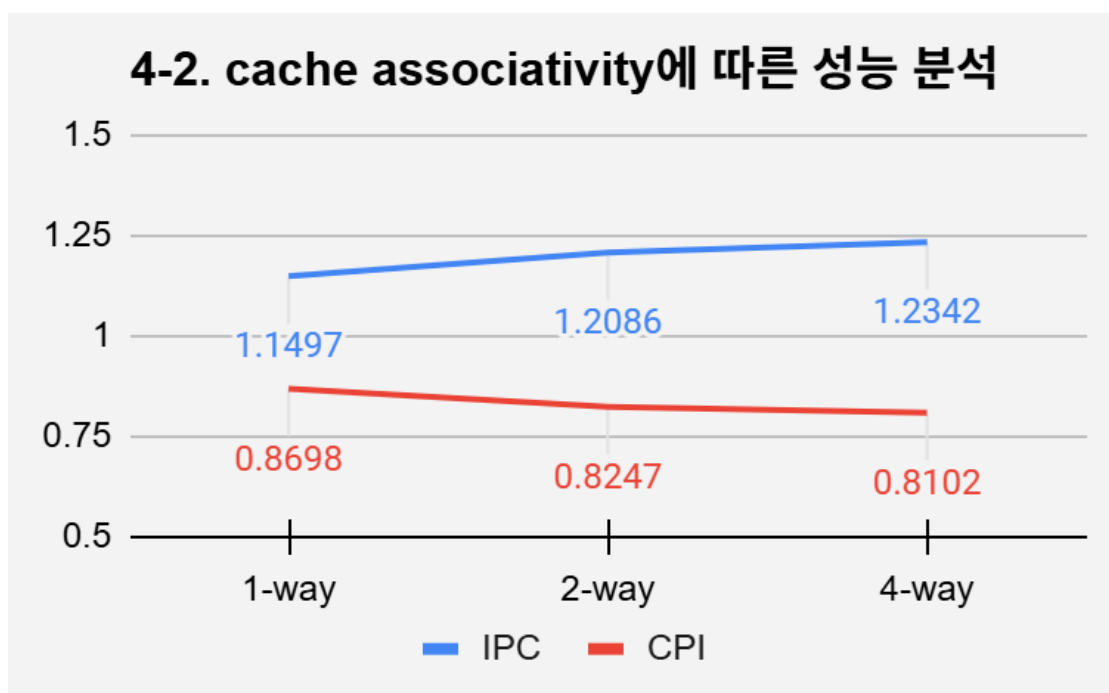
순서대로 16B, 32B, 64B, 128B 일 때 Miss_rate이다. 공간적 지역성이 큰 만큼 block_size를 키워주었을 때 성능이 miss_rate는 감소하고, 이 감소로 인해 동일한 penalty에서 access time이 줄어들기 때문에 성능이 향상된다. Block size가 크면서 miss 한번 발생 시에 더 많은 데이터를 가져오고 공간적 지역성으로 이후 hit 수가 많아지면서 miss_rate가 감소한다. Miss_rate 감소는 stall이 적어지고 이는 결국 IPC를 증가시켜준다.

단점 : gzip에 한정하여 공간적 지역성이 좋기 때문에 효율이 극대화 되었을 뿐 보통의 경우 64B에서 최적의 성능을 내는 경우가 많다. block size가 커짐에 따라 한 번 miss시 교체되는 시간, 즉 miss penalty가 커진다는 단점이 존재하며, 한정된 cache 크기에서 entry 수를 줄여 Capacity miss 와 Conflict miss를 초래할 수 있다.

이외에도 불필요 데이터까지 block에 포함될 수 있으며, pollution 문제로 인해 교체가 발생할 수도 있다.

4.2 Associativity 변화

4.2.1 gcc



[데이터 요약]

1-way : IPC 1.1497 / CPI 0.8698

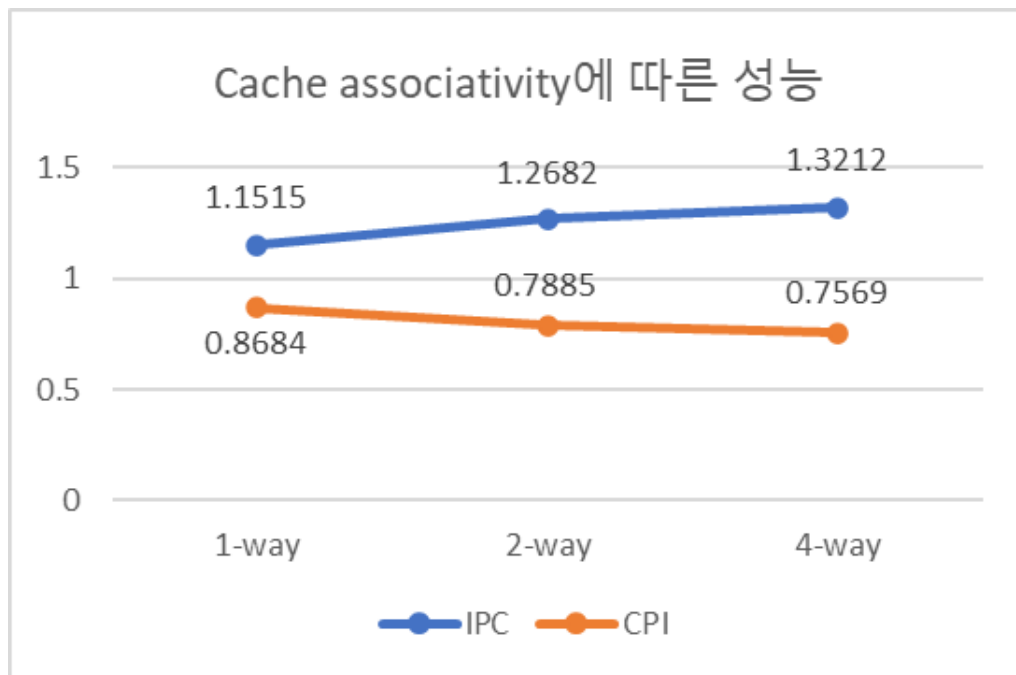
2-way : IPC 1.2086 / CPI 0.8247

4-way : IPC 1.2342 / CPI 0.8102

[분석 및 해석]

1-way에서 2-way로 증가할 때 conflict miss 감소로 **IPC는 약 5% 증가, CPI는 5% 감소**해 일정 수준의 성능 향상이 나타났다. 그러나 2-way에서 4-way로 확장할 때 IPC 증가는 2%에 그치며 대부분의 conflict miss는 이미 2-way에서 해결되는 것으로 나타났다.

Associativity가 높아질수록 비교해야 할 tag가 많아져 access latency 증가, 전력 소모 증가, 회로 복잡도 증가 문제가 발생한다.



[데이터 요약]

1-way : IPC 1.1515 / CPI 0.8684

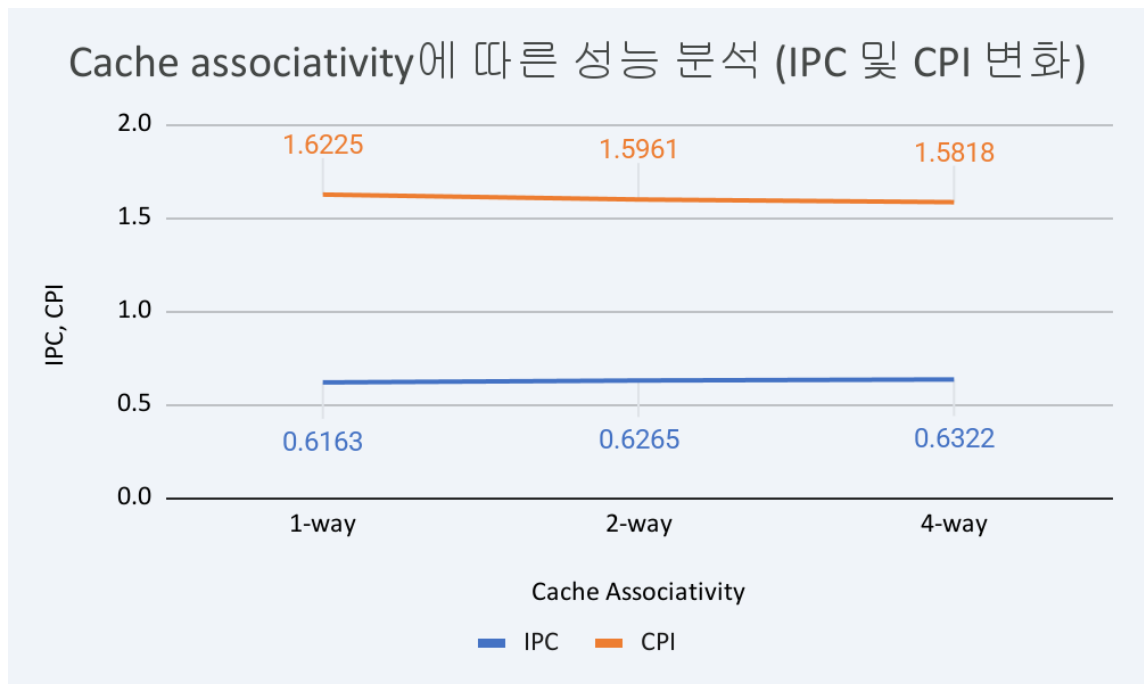
2-way : IPC 1.2682 / CPI 0.7885

4-way : IPC 1.3212 / CPI 0.7569

[분석 및 해석]

1-way는 데이터 충돌로 인해 성능이 낮다. 4-way로 늘려 유연성을 확보했을 때 성능이 개선되었다. 다만 Block Size 변화만큼 큰 차이를 나타내지는 않는다.

그러나 1-way는 바로 데이터를 찾지만, 4-way는 4개의 태그를 동시에 비교하고 그중 맞는 것을 골라내는 MUX과정을 거쳐야 한다. 이로 인해 캐시 접근 속도가 느려질 수 있다. 그리고 Associativity를 4개로 높였을 경우 매 접근마다 4개의 비교기를 동시에 켜야 하므로 에너지를 더 많이 사용한다.



[데이터 요약]

1-way : IPC 0.6163 / CPI 1.6225

2-way : IPC 0.6265 / CPI 1.5961

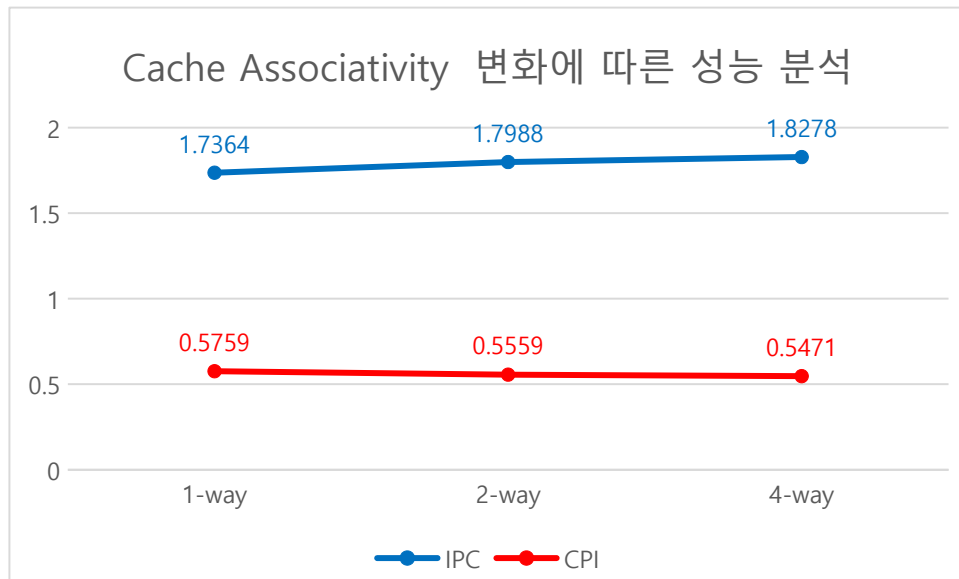
4-way : IPC 0.6322 / CPI 1.5818

[분석 및 해석]

Conflict Miss의 완화 : Associativity가 1-way에서 4-way로 증가함에 따라 IPC가 꾸준히 상승하는 경향을 보인다.

1-way 방식의 한계 : 1-way 방식은 특정 주소의 데이터가 들어갈 수 있는 자리가 딱 한 곳뿐이다. 따라서 서로 다른 두 데이터가 우연히 같은 자리를 배정받으면 번갈아 가며 서로를 밀어내는 Thrashing이 발생하기 쉽다.

유연성 확보 : 4-way 방식은 데이터가 들어갈 수 있는 자리가 4 set 있으므로, 충돌을 피할 확률이 높아져 Miss Rate가 감소하고 성능이 향상된다. 다만, 하드웨어 복잡도와 Access Time은 증가할 수 있다.



[데이터 요약]

1-way : IPC 1.7364 / CPI 0.5759

2-way : IPC 1.7988 / CPI 0.5559

4-way : IPC 1.8278 / CPI 0.5471

[분석 및 해석]

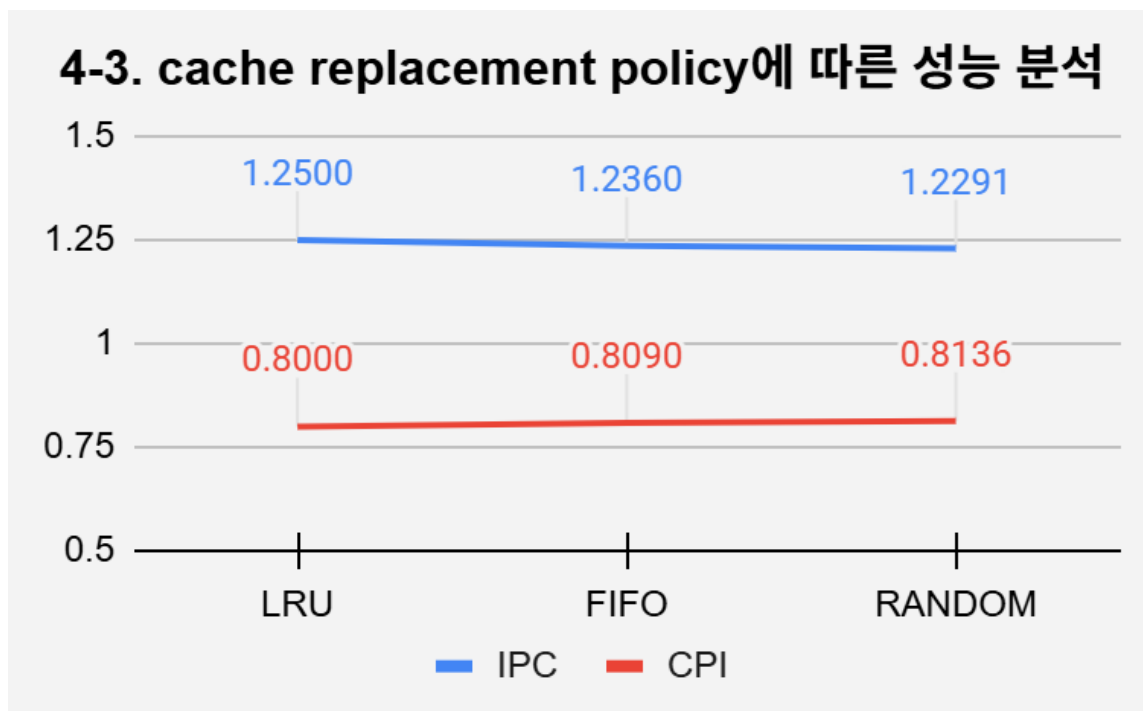
결과 : associativity가 증가함에 따라 IPC 성능이 점차 증가한다.

이유 : associativity이 커지면서 conflict miss가 감소한다. 1-way의 경우 direct mapped 방식과 동일하게 동작하므로 가장 많은 conflict miss가 발생하고 miss에 따른 지연 penalty가 더 자주 발생한다. 이는 IPC를 감소하는 요인이다. 4-way에서 conflict miss가 가장 최소가 되고 이에 따라 IPC 성능이 가장 좋게 나오게 되었다.

단점 : n-way만큼 n개의 비교기가 필요하게 되고, 선택 로직이 늘어나며 설계 크기, 난이도 등이 증가하게 된다. 해당 로직이 추가되며 critical path에 부담이 생길 수 있으며 지역성 해소에 따른 전체 성능에 영향을 줄 수 있다.

4.3 Replacement Policy 변화

4.3.1 gcc



[데이터 요약]

LRU : IPC 1.2500 / CPI 0.8000

FIFO : IPC 1.2360 / CPI 0.8090

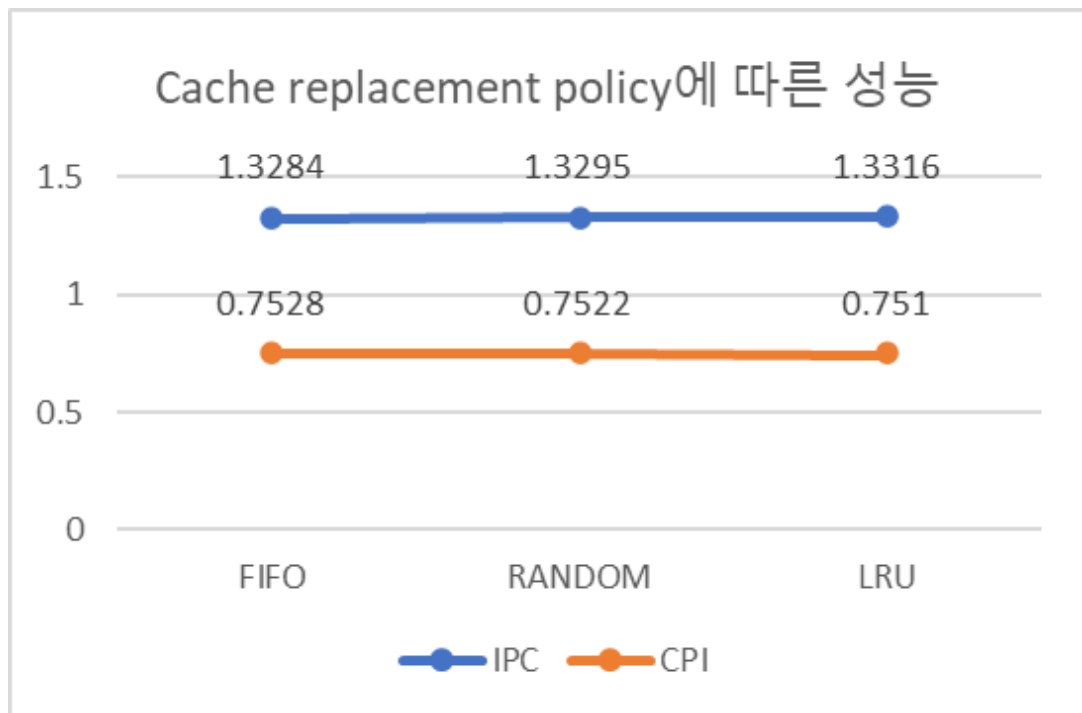
RANDOM : IPC 1.2291 / CPI 0.8136

[분석 및 해석]

Replacement 정책(LRU, FIFO, Random)은 성능에 거의 영향을 주지 않았지만, LRU가 가장 높은 성능을 보였다.

LRU → FIFO 전환에서 IPC는 1.1% 감소, FIFO → RANDOM에서는 0.5% 감소로 성능 변화가 미미했다. 이는 워크로드가 locality가 잘 유지되어 교체 정책 차이가 miss rate에 큰 영향을 주지 않았기 때문이다. LRU는 최근에 사용된 데이터가 가까운 미래에도 다시 사용될 가능성이 높다는 시간적 지역성을 잘 반영하여 불필요한 캐시 교체를 줄이기 때문에 가장 좋은 성능을 보인다.

LRU는 구현 복잡도가 높고 하드웨어 비용이 증가한다. 반면 FIFO나 RANDOM은 단순하지만, 특정 상황에서 최적의 교체 결정을 보장하지 못할 수 있다.



[데이터 요약]

LRU : IPC 1.3316 / CPI 0.751

FIFO : IPC 1.3284 / CPI 0.7528

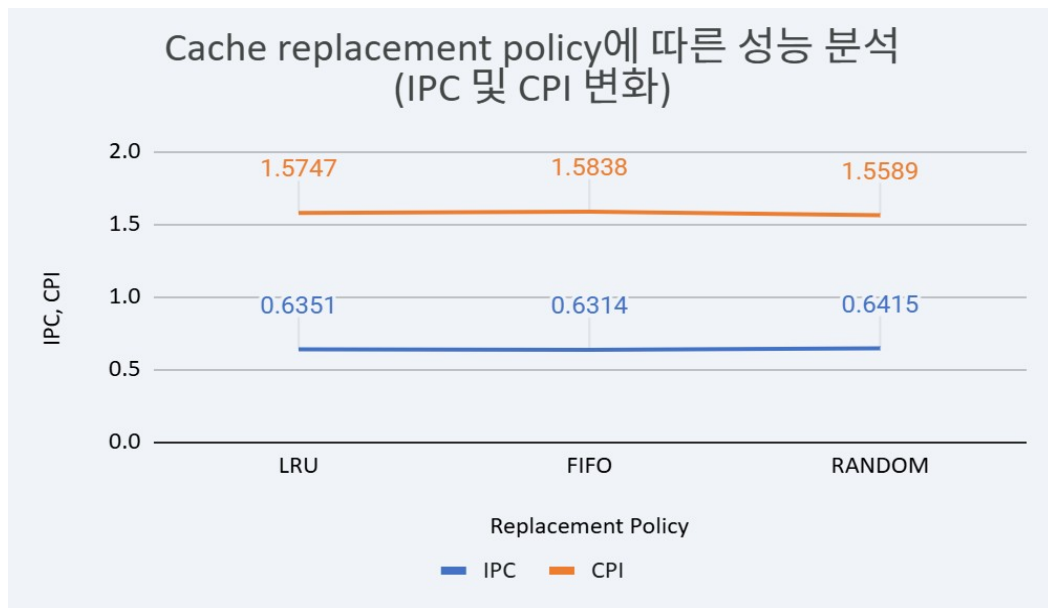
RANDOM: IPC 1.3295 / CPI 0.7522

[분석 및 해석]

세 정책 간 차이는 미미하지만, LRU가 가장 높다. 이는 가장 최근에 본 바둑판 패턴을 다시 볼 확률이 높다는 시간적 지역성을 반영한다.

그러나 LRU를 사용하는 경우, '진짜 LRU'를 하드웨어로 구현하려면, 모든 블록의 접근 순서를 기록하는 비트(Matrix bit 등)를 계속 업데이트해야 한다. Way 수가 늘어날수록 이 로직이 매우 비싸고 복잡해지는 단점이 있다.

또한 데이터가 캐시 크기보다 아주 조금 더 큰 상태로 순차적으로 반복 접근될 경우, LRU는 모든 데이터가 미스가 나는 최악의 경우를 만들 수 있다.(오히려 Random이 나올 때도 있음)



[데이터 요약]

LRU : IPC 0.6351 / CPI 1.5747

FIFO : IPC 0.6314 / CPI 1.5838

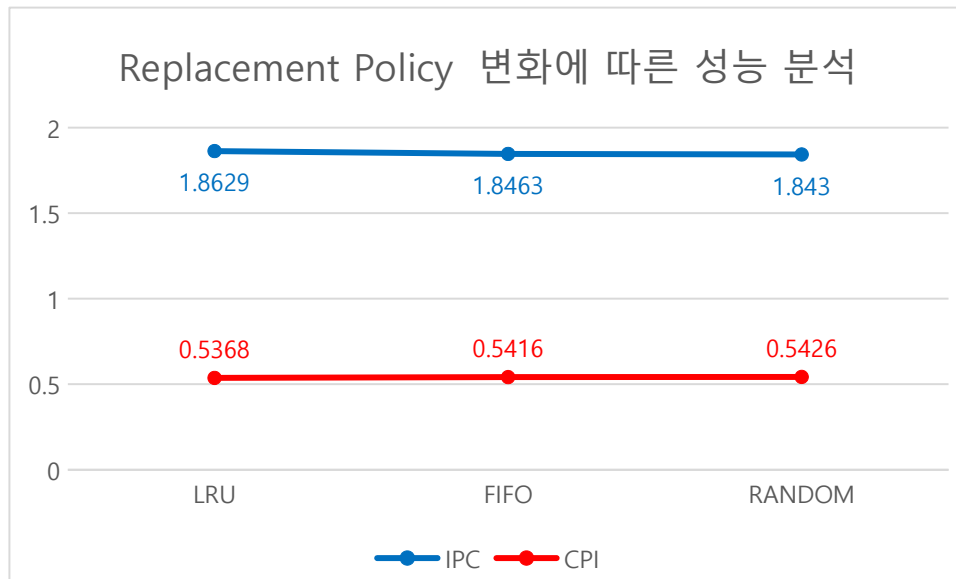
RANDOM: IPC 0.6415 / CPI 1.5589

[분석 및 해석]

시간적 지역성(Temporal Locality)과 LRU : 보통은 가장 오랫동안 참조되지 않은 데이터를 버리는 LRU가 가장 성능이 좋은 것이 일반적이다. 그러나 본 실험에서는 RANDOM 정책이 가장 높은 성능을 보였다.

LRU의 병리적 현상 : LRU가 RANDOM보다 성능이 낮은 경우는, 프로그램이 캐시 크기보다 약간 더 큰 데이터를 반복적으로 loop 할 때 발생한다. 예를 들어, 캐시가 4개를 저장할 수 있는데 데이터 1, 2, 3, 4, 5를 계속 순서대로 부르면, LRU는 방금 쓴 1을 버리고 5를 넣고 다시 1을 부를 때 2를 버리는 식으로 100% miss를 낼 수 있다. 이런 상황에서는 차라리 무작위로 버리는 것이 운 좋게 필요한 데이터를 남길 확률이 있어 성능이 더 높게 나올 수 있다.

FIFO의 성능 : 가장 먼저 들어온 것을 버리는 FIFO는 지역성을 전혀 고려하지 않으므로, 셋 중 가장 낮은 성능을 보였다.



[데이터 요약]

LRU : IPC 1.8629 / CPI 0.5368

FIFO : IPC 1.8463 / CPI 0.5416

RANDOM: IPC 1.843 / CPI 0.5426

[분석 및 해석]

결과 : LRU, FIFO, RANDOM 방식 모두 비슷한 성능이지만 그 중 LRU가 가장 높은 IPC를 보여준다.

이유 : gzip의 경우 슬라이딩 윈도우 + 잦은 재참조로 인해서 사용했던 데이터, 시간적 지역성이 높아 최근 사용한 데이터에 다시 접근할 가능성이 높다. 따라서 다른 두 정책에 비해 cache miss 횟수가 적고 IPC가 가장 높게 나온다.

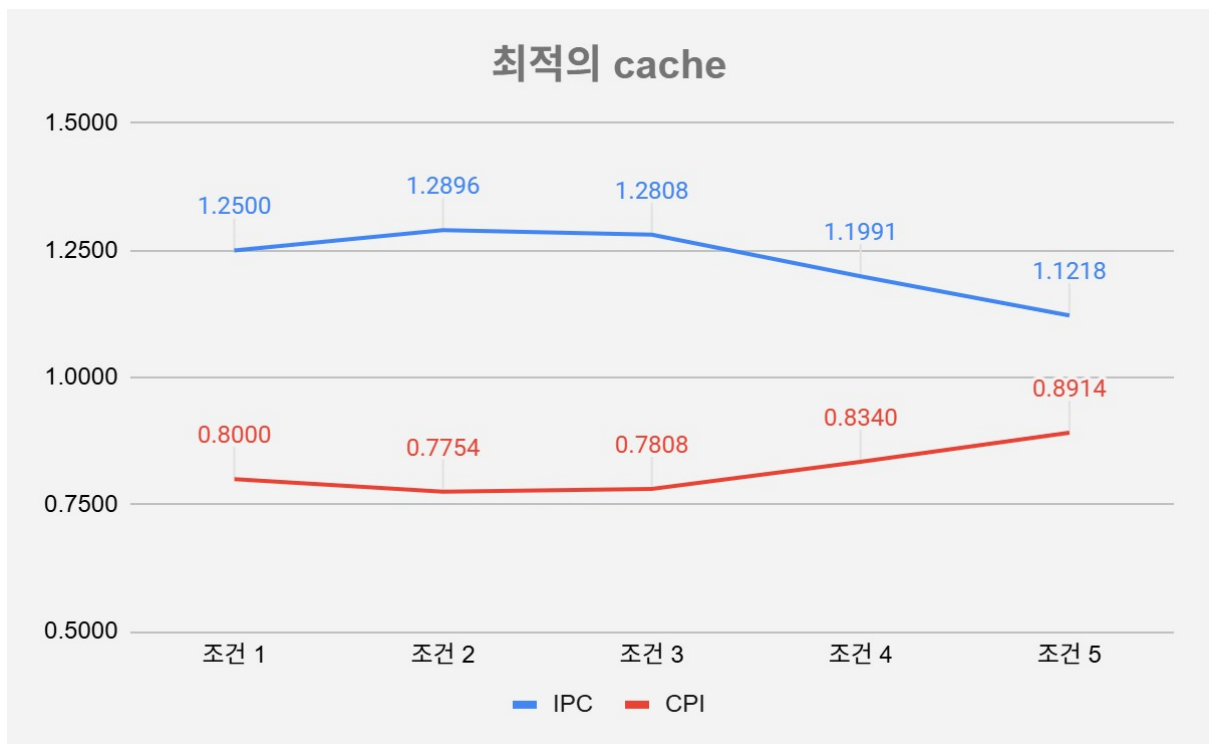
단점 : LRU는 다른 두 로직에 비해서 ‘가장 오래 안 쓰인’ 이란 조건을 확인하기 위해서 이를 LRU bit로 기억하고 있어야 하며, 이를 구성하기 위한 로직이 다른 두 조건에 비해 구현 회로가 복잡해진다.

5 최적 Cache configuration 탐색

5.1 gcc

[조건 정보 및 데이터 요약]

5. 최적의 캐시 조합	번호	block size	associativity	replacement policy	IPC	CPI
	조건 1	64B	4-way	LRU	1.2500	0.8000
	조건 2	128B	4-way	LRU	1.2896	0.7754
	조건 3	128B	4-way	FIFO	1.2808	0.7808
	조건 4	128B	2-way	LRU	1.1991	0.8340
	조건 5	64B	2-way	LRU	1.1218	0.8914



[분석 및 해석]

가장 높은 성능을 보인 조합은 조건 2(128B, 4-way, LRU)로 IPC 1.2896, CPI 0.7754가 측정되었다. 이는 두 번째로 좋은 성능을 보인 조건 3(IPC 1.2808)보다 약 0.7%, 조건 1(IPC 1.2500)보다 약 3.17%, 가장 낮은 성능인 조건 5(IPC 1.1218) 대비 약 14.9% 성능이 향상된 수치다. CPI 또한 조건 2가 가장 낮아, 메모리 접근 지연을 최소화한 안정적인 파이프라인 동작이 이루어졌음을 보여준다.

조건 2가 가장 높은 성능을 보인 이유는 세 가지 요소가 동시에 최적 조합으로 작용했기 때문이다. 첫째, 128B 블록 크기는 공간 지역성(spatial locality)을 극대화한다. 프로그램이 연속된 주소를 자주 접근할 때, 큰 블록 크기는 한 번의 캐시 접근으로 더 많은 데이터를 가져와 이후 miss를 줄이고 hit율을 높인다. 둘째, 4-way associativity는 conflict miss를 효과적으로 감소시킨다. 같은 블록 크기를 사용하더라도 2-way보다 4-way가 충돌 가능성이 훨씬 낮아, 특히 메모리 접근 패턴이 복잡한 프로그램에서 안정적인 hit율 확보가 가능하다.

셋째, LRU 교체 정책은 locality 기반의 접근 패턴을 잘 반영해, 가장 오래 사용되지 않은 블록을 교체하는 방식으로 불필요한 miss를 줄인다. 같은 128B·4-way 구조를 사용하더라도 FIFO 기반의 조건 3보다 조건 2의 IPC가 더 높았던 이유가 바로 이 점이다.

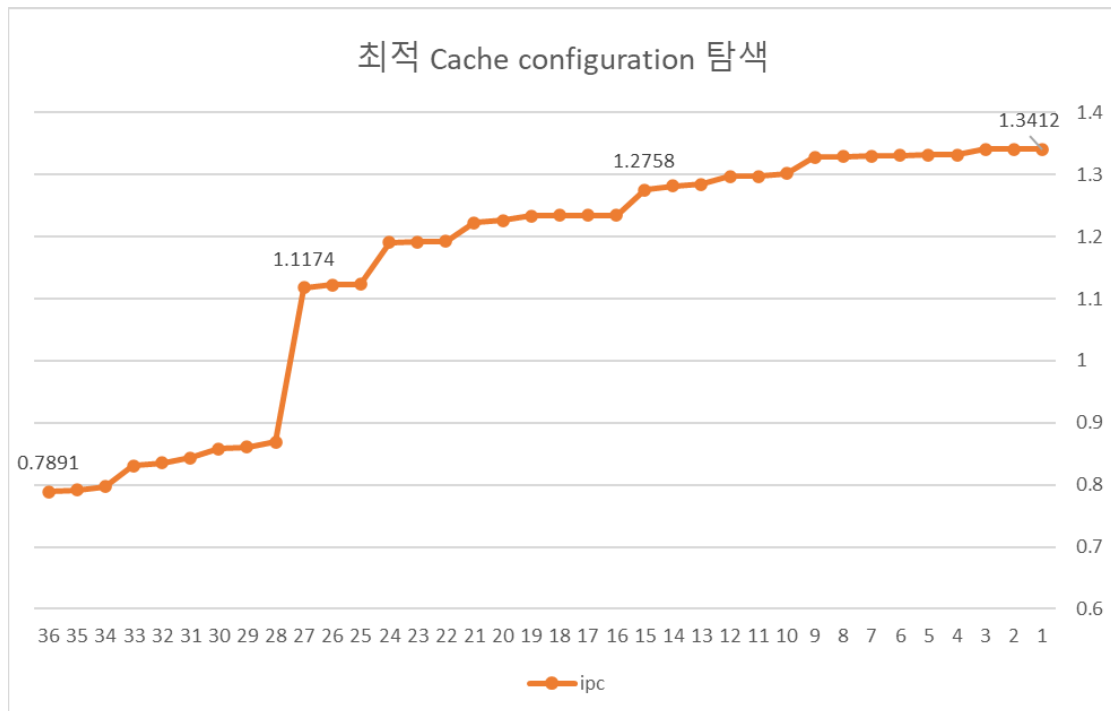
반대로 성능이 가장 낮았던 조건 5는 64B 블록 크기와 2-way associativity 조합이 주요 원인이다. 블록 크기가 작아 spatial locality를 제대로 활용하지 못하고, 2-way 구조는 conflict miss가 증가해 메모리 접근 빈도가 높아진다. 또한 LRU를 사용하더라도 낮은 associativity 구조 자체가 병목이 되어 충분한 성능을 내기 어렵다. 그 결과 IPC는 조건 2 대비 약 14.9%나 낮아졌다.

종합적으로, 성능에 영향을 미치는 요소들의 기여도를 비교하면 block size > associativity > replacement policy 순으로 성능 차이를 좌우하는 경향이 뚜렷하게 나타났다. 캐시 크기가 동일할 때 block size와 associativity는 miss rate을 직접적으로 줄이는 핵심 요인이며, replacement policy는 이 두 요소가 제공하는 구조적 이점을 미세 조정하는 역할을 한다.

다만 블록 크기 증가와 associativity 상승은 성능 향상과 함께 단점도 갖는다. 128B 블록 크기는 locality가 낮은 프로그램에서는 오히려 불필요한 데이터를 가져와 메모리 대역폭을 낭비할 수 있고, 4-way 구조는 태그 비교가 늘어나 회로 면적 증가·접근 지연 증가·전력 소모 증가를 유발한다. LRU 또한 FIFO보다 구현 복잡도가 높아 추가적인 하드웨어 자원을 필요로 한다.

결론적으로 조건 2는 hit율 개선을 최대로 끌어올릴 수 있는 조합으로 가장 높은 IPC를 보여주었고, 이는 spatial locality와 conflict miss 감소, LRU 정책의 효율이 함께 맞물린 결과이다. 그러나 이러한 최적 조합은 면적·전력·회로 복잡도 측면에서의 비용 증가를 수반하므로, 실제 설계에서는 성능과 자원 효율의 균형을 고려해야 한다는 점도 확인할 수 있다.

5.2 Go



[조건 정보 및 데이터 요약]

3번, 4번 실험 결과를 토대로 Block Size(4종) × Associativity(3종) × Policy(3종) 조합을 시뮬레이션을 진행했다.

상위 10개의 조합의 조건 정보는 다음과 같다.

block	assoc	policy	ipc	cpi
1	128	4 l	1.3412	0.7456
2	128	4 f	1.3408	0.7458
3	128	4 r	1.3407	0.7459
4	128	2 l	1.3318	0.7509
5	64	4 l	1.3316	0.751
6	128	2 f	1.3305	0.7516
7	128	2 r	1.3297	0.7521
8	64	4 r	1.3295	0.7522
9	64	4 f	1.3284	0.7528
10	64	2 l	1.3022	0.7679

하위 10개의 조합의 정보는 다음과 같다.

컴퓨터 구조 Sim-Scalar 결과보고서 8조

block	assoc	policy	ipc	cpi
36	16	1 r	0.7891	1.2673
35	16	1 f	0.7914	1.2636
34	16	1 l	0.7974	1.2541
33	16	2 r	0.8309	1.2035
32	16	2 f	0.835	1.1976
31	16	2 l	0.8437	1.1853
30	16	4 r	0.8581	1.1654
29	16	4 f	0.8611	1.1613
28	16	4 l	0.8692	1.1505
27	32	1 f	1.1174	0.895

[분석 및 해석]

IPC 상위권은 모두 Block Size 128B인 케이스들이 차지했다.

최적 조합 1위: Block 128B / Assoc 4-way / Policy LRU (IPC 1.3412)

최적 조합 2위: Block 128B / Assoc 4-way / Policy FIFO (IPC 1.3408)

go 어플리케이션의 메모리 성능은 Block Size가 지배적인 요인이며, Associativity와 Policy는 이를 보조하는 역할을 한다.

Block Size (128B) : go 어플리케이션은 바둑판 전체 데이터를 수시로 참조하고 특정 지역을 집중 분석하는 특성이 있어 공간적 지역성이 매우 높다. 128B의 큰 블록 크기를 사용하는 것이 메모리에 연속 배열된 인접 데이터를 한 번에 가져와 Cache Hit Rate를 높이는 데 결정적인 역할을 하여 성능이 폭발적으로 증가했다.

Associativity (4-way) : 1-way 방식에서 발생하는 데이터 충돌을 완화하기 위해 4-way로 유연성을 확보했다. 이를 통해 필요한 데이터를 캐시 내에 더 안정적으로 유지할 수 있게 되어 Stall이 감소하고 연산 성능이 향상되었다.

Replacement Policy (LRU) : 바둑판의 패턴 매칭 특성상 가장 최근에 본 패턴을 다시 볼 확률이 높은 시간적 지역성을 가진다. LRU 정책이 이를 가장 잘 반영하여 FIFO나 Random 방식보다 미세하지만 더 높은 성능(IPC 1.3412)을 달성했다.

단점 :

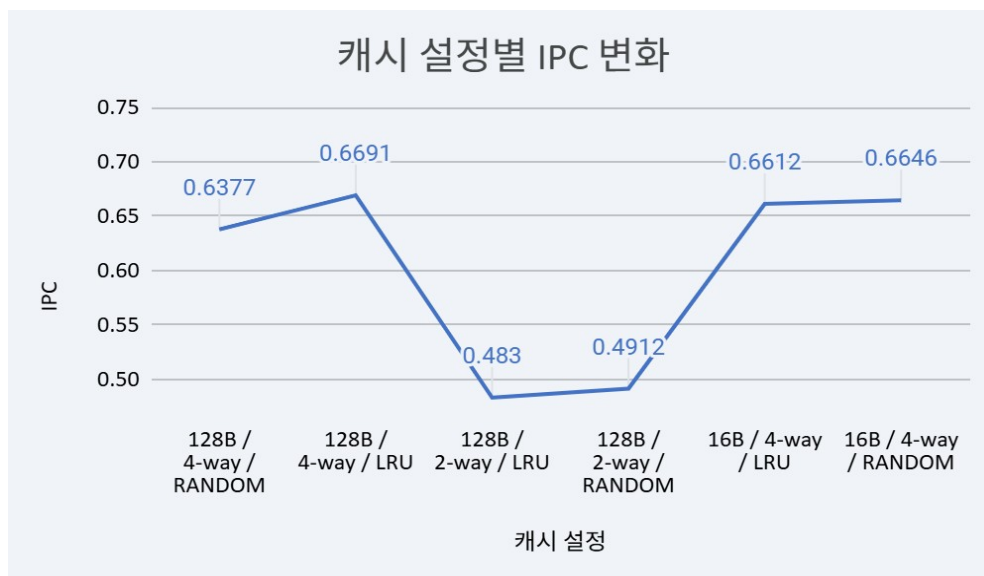
Block Size (128B) : 블록 크기가 커진 만큼 캐시 미스 발생 시 메인 메모리로부터 데이터를 가져오는 전송 시간이 길어져 Miss Penalty가 증가한다. 또한, 당장 필요하지 않은 데이터까지 가져오는 Cache Pollution(캐시 오염)이 발생하여 유용한 데이터를 밀어낼 위험이 있다.

Associativity (4-way) : 4개의 Way 중 맞는 데이터를 찾기 위해 태그를 비교하고 선택하는 MUX 과정이 추가되어, 1-way 방식보다 회로가 복잡해지고 캐시 접근 시간이 지연될 수

있다. 또한, 비교기 동시 동작으로 인해 전력 소모가 증가한다.

Replacement Policy (LRU) : '진짜 LRU'를 하드웨어로 구현하기 위해서는 모든 접근 순서를 기록해야 하므로 구현 비용이 매우 비싸고 로직이 복잡해지는 단점이 있다.

5.3 mcf



[실험 개요]

앞서 수행한 개별 실험(④)에서 성능 향상에 기여도가 높았던 Block Size(128B, 16B), Associativity(4-way, 2-way), Replacement Policy(LRU, Random)를 조합하여 mcf 벤치마크에 최적화된 Cache Configuration을 탐색하였다.

[실험 결과 및 분석]

(1) 최적의 조합 선정 : 128B / 4-way / LRU 실험 결과, Block Size 128B, Associativity 4-way, Replacement Policy LRU 조합에서 IPC 0.6691로 가장 높은 성능을 기록하였다. 이는 Baseline 및 개별 실험의 최고치와 동등하거나 상회하는 결과로, mcf의 메모리 병목을 해소하기 위해서는 큰 블록 크기와 높은 연관도(Conflict 방지)가 동시에 충족되어야 함을 시사한다.

(2) Block Size와 Associativity의 상관관계 분석 (성능 저하의 원인) 데이터에서 가장 주목할 점은 Block Size 128B일 때 Associativity에 따른 성능 격차이다.

- 4-way일 때: IPC 0.6691 (최상위권)
- 2-way일 때: IPC 0.4830 (최하위권) 동일한 128B 블록이라도 Associativity를 2-way로 줄이자 성능이 약 28% 급락하였다.

● 원인 분석 : 전체 캐시 용량이 고정된 상태에서 Block Size를 128B로 키우면, 캐시의 전체 Set 개수(Index)가 줄어든다. 이 상황에서 Associativity마저 2-way로 낮아지면, 서로 다른 주소가 같은 Set에 매핑될 때 이를 수용할 공간(Way)이 부족해져 치명적인 Conflict Miss가 발생한 것으로 분석된다. 즉, 큰 블록 사이즈를 사용할 때는 반드시 높은 Associativity가 보장되어야 성능을 유지할 수 있다.

(3) Replacement Policy의 역전 현상 (Random vs LRU) 앞선 개별 실험(Block Size 64B 기준)에서는 Random 정책이 LRU보다 우세했으나, 최적화 조합에서는 결과가 달라졌다.

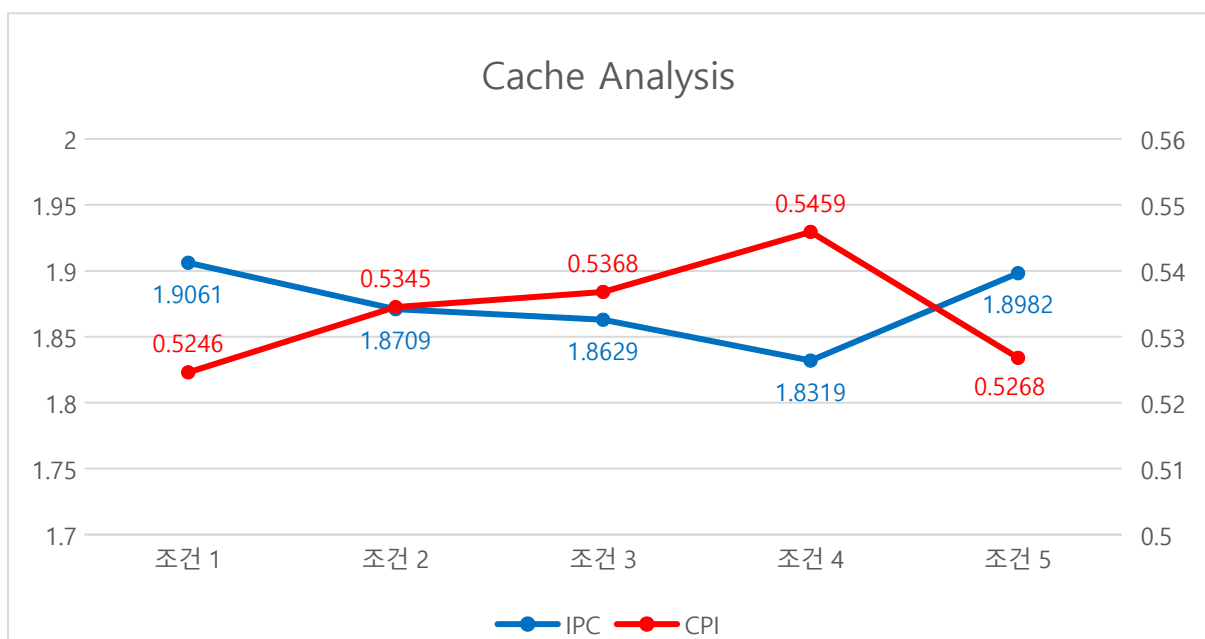
- 16B (작은 블록) 조합: Random(0.6646) > LRU(0.6612) → Random 우세 유지
- 128B (큰 블록) / 4-way 조합: LRU(0.6691) > Random(0.6377) → LRU 역전

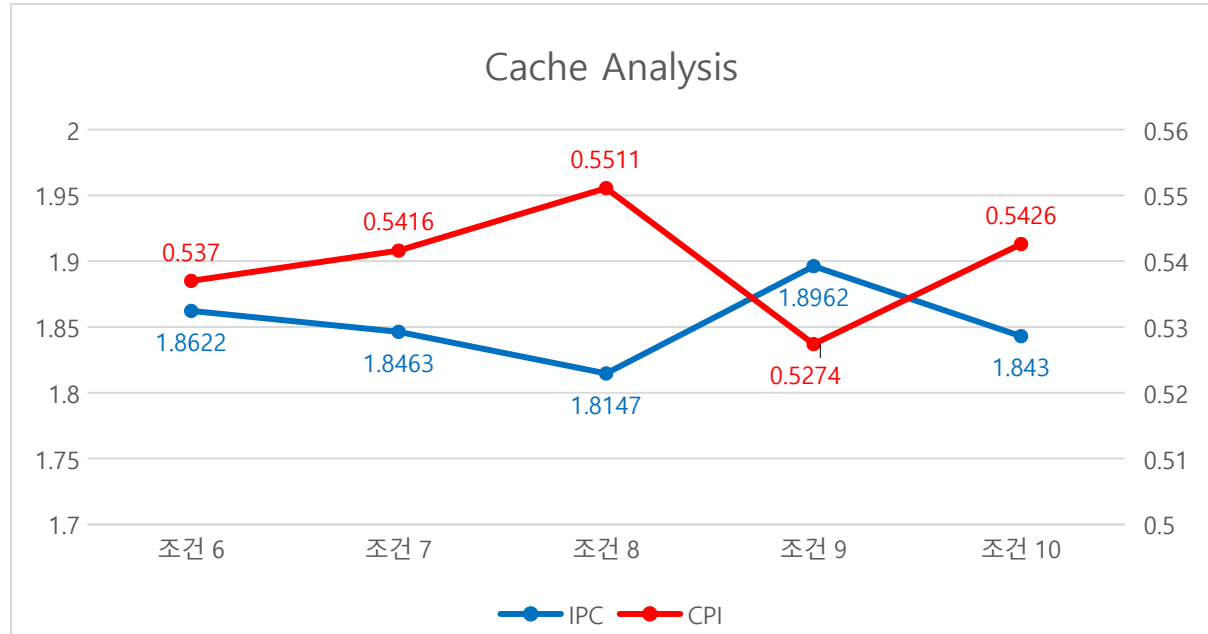
● 원인 분석 : 16B처럼 블록이 작아 Entry 개수가 충분할 때는 mcf의 불규칙한 접근 패턴에 대해 Random 방식이 유효했다. 그러나 128B로 블록이 커지면서 Entry 개수가 줄어들고, 4-way로 연관도가 확보된 상태에서는 무작위 교체보다 참조 지역성을 고려한 LRU 방식이 Cache Pollution을 줄이는 데 더 효과적이었던 것으로 판단된다.

[결론]

단순히 개별 실험의 1위 값들을 조합하는 것이 항상 최적의 결과를 보장하지 않음을 확인하였다. mcf 벤치마크의 경우, Block Size를 128B로 확장하여 공간적 지역성을 극대화하되, 이에 따른 Set 감소 부작용을 4-way Associativity와 LRU 정책으로 보완하는 [128B / 4-way / LRU] 설정이 최종적인 최적의 해임을 확인하였다.

5.4 gzip





[조건 정보 및 데이터 요약]

	조건1			조건2			조건3			조건4			조건5		
	128B	4-way	LRU	128B	2-way	LRU	64B	4-way	LRU	64B	2-way	LRU	128B	4-way	FIFO
IPC		1.9061			1.8709			1.8629			1.8319			1.8982	
CPI		0.5246			0.5345			0.5368			0.5459			0.5268	
	조건6			조건7			조건8			조건9			조건10		
	128B	2-way	FIFO	64B	4-way	FIFO	64B	2-way	FIFO	128B	4-way	RAND	64B	4-way	RAND
IPC		1.8622			1.8463			1.8147			1.8962			1.843	
CPI		0.537			0.5416			0.5511			0.5274			0.5426	

[분석 및 해석]

결과 : 총 10개의 조건을 바탕으로 진행한 결과 128B block_size, 4-way set associativity, LRU를 사용한 조건이 IPC 성능이 가장 좋게 분석되었다.

이유 : gzip이 공간적 지역성이 우수하여 128B 블록 크기를 이용하는 것이 miss rate를 감소시키고 이는 stall의 감소로 이어져 성능이 증가하였다.

4-way associativity를 통해 conflict miss를 완화하여 동일 set 내에 여러 블록을 유지하고 cache miss rate를 감소시킨다. 이로 인해 동일하게 stall 수가 감소하여 연산 성능이 증가한다.

LRU를 통해 접근한 데이터가 재사용되는 시간적 지역성이 우수해 gzip과 합이 좋았고 다른 두 정책에 비해 높은 성능을 내고 있다.

단점 : 블록 크기가 큰 만큼 miss penalty가 증가하고, pollution으로 인해 지역성이 약한, 필요없는 데이터까지 같이 포함되어 캐시가 급방 더러워질 수 있다.

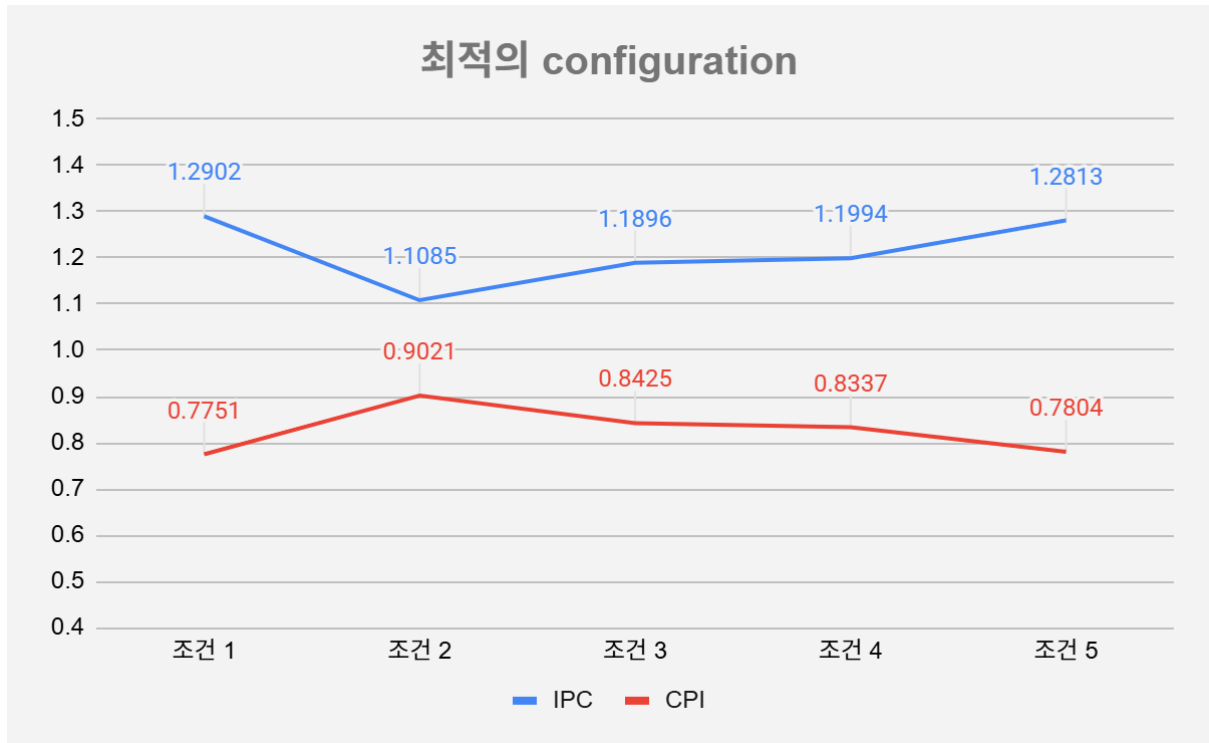
4-way 방식인 만큼 1-way인 direct mapped 방식보다 회로가 더 들어가게 되고 mux 복잡도 역시 증가하여 캐시 접근하는데 지연이 발생하거나 에너지 소모가 늘어날 수 있다.

6 Application별 최종 System configuration

6.1 gcc

[조건 정보 및 데이터 요약]

6. 최적의 config 조합	번호	block size	associativity	replacement policy	연산장치	스칼라	분기예측	IPC	CPI
	조건 1	128B	4-way	LRU	4개	4개	3bit	1.2902	0.7751
	조건 2	64B	2-way	FIFO	4개	4개	3bit	1.1085	0.9021
	조건 3	128B	2-way	FIFO	4개	4개	3bit	1.1896	0.8425
	조건 4	128B	2-way	LRU	4개	4개	3bit	1.1994	0.8337
	조건 5	128B	4-way	FIFO	4개	4개	3bit	1.2813	0.7804



[분석 및 해석]

전체적으로 가장 높은 성능을 보인 조합은 조건 1(IPC 1.2902)이며, 그 다음은 조건 5(IPC 1.2813)로 나타났다. 두 조건의 공통점은 블록 크기가 128B이고 4-way set associative 구조를 사용한다는 점이다. 이 구조는 L1 캐시에서 conflict miss를 줄이는 데 효과적이며, 비교적 큰 블록 크기를 통해 spatial locality를 충분히 활용해 캐시 hit율을 크게 높여 준다. 또한 LRU(조건 1)와 FIFO(조건 5)의 차이가 있음에도 불구하고 성능 차이가 크지 않은 이유는, 교체 정책보다 associativity와 블록 크기가 캐시 성능에 훨씬 지배적인 영향을 주기 때문이다.

반대로 가장 낮은 성능은 조건 2로, IPC가 1.1085에 불과했다. 이 조건은 블록 크기가 유일하게 64B로 가장 작고 associativity도 2-way로 낮다. 블록 크기가 작아지면 한 번의 캐시 접근으로 가져오는 데이터 양이 줄어 spatial locality 활용도가 떨어지고, associativity가 낮아지면 conflict miss가 증가한다. 결국 메모리 접근 지연이 늘어나 파이프라인 공급이 원활하지 못해 IPC가 가장 낮게 나타난 것이다. 조건 3과 4는 모두 128B 블록 크기와 2-way 구조를 기반으로 하지만, replacement policy 차이로 IPC에 차이가 생겼다. LRU를 사용하는

조건 4가 FIFO 기반의 조건 3보다 성능이 더 높았는데, 이는 LRU가 locality 기반 프로그램 특성을 더 잘 반영하여 불필요한 miss를 줄일 수 있기 때문이다.

128B 블록 크기 내에서는 associativity의 영향이 크게 드러난다. 조건 4(128B·2-way·LRU)보다 조건 5(128B·4-way·FIFO)의 성능이 더 높은 이유는, 교체 정책의 효율보다 associativity 증가로 인한 conflict miss 감소 효과가 더 크게 작용했기 때문이다. 즉 FIFO라도 4-way라면 LRU 기반 2-way보다 더 나은 hit율을 확보할 수 있다. 이러한 결과를 종합하면 성능에 기여하는 요소의 중요도는 블록 크기 > associativity > 교체 정책 순으로 나타난다.

성능이 가장 좋은 조건 1과 조건 5는 캐시 hit율을 최대한 높여 파이프라인이 끊김 없이 명령을 공급받을 수 있게 해주는 구성이다. 연산장치 4개와 스칼라 4개가 병렬 실행을 담당하고, 3-bit 분기 예측이 분기 실패를 줄여 stall을 억제한 상태에서, 캐시 시스템이 높은 hit율을 유지하면 전체 IPC가 가장 안정적으로 증가한다. 특히 128B 블록 크기는 연속된 주소를 자주 접근하는 일반적인 프로그램 패턴을 잘 반영하여 메모리 접근 빈도를 크게 줄이는 핵심 요소로 작용했다.

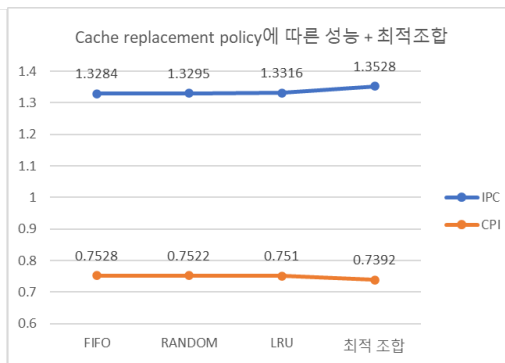
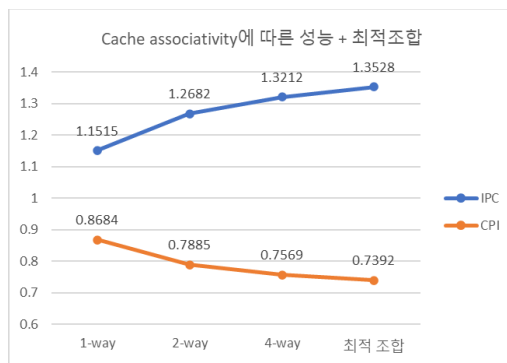
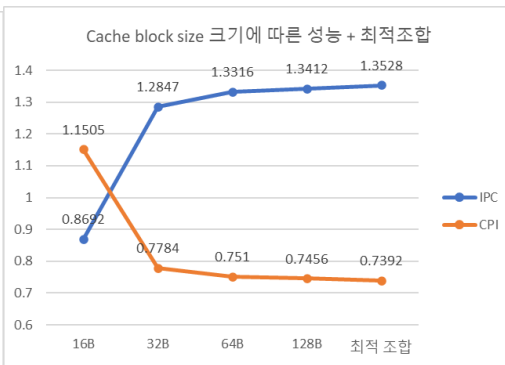
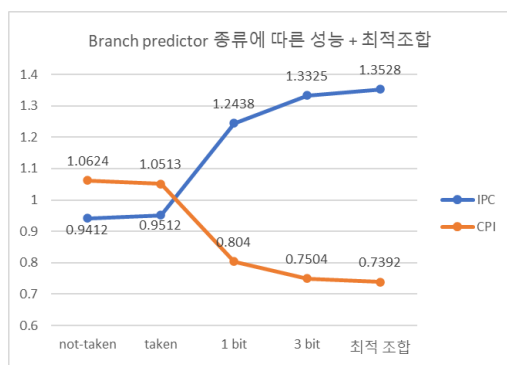
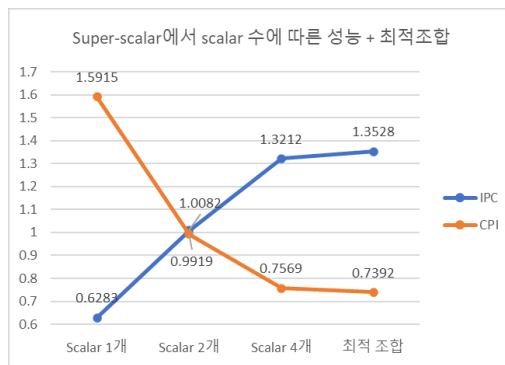
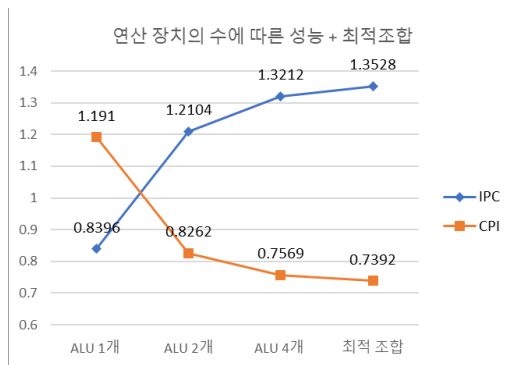
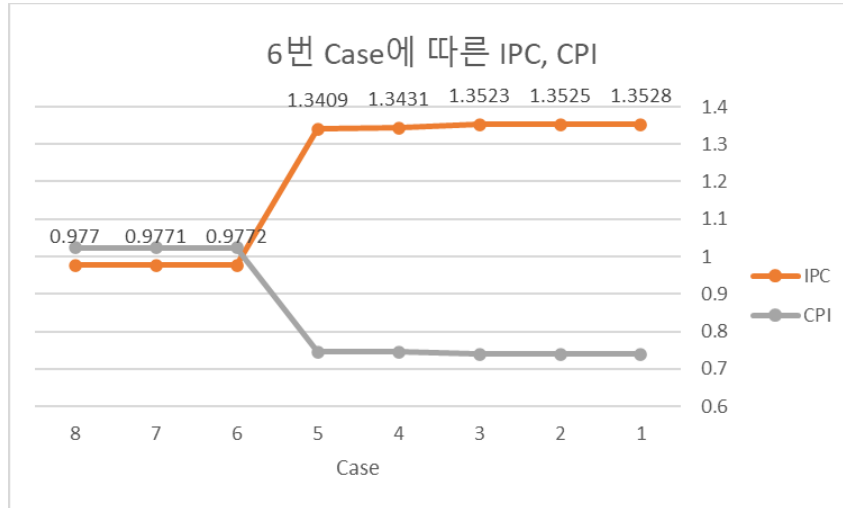
다만 성능이 가장 좋은 조합이 항상 비용 대비 효율이 좋은 것은 아니다. 우선 4-way associativity는 2-way보다 비교해야 할 tag 수가 많아져 회로 복잡도가 증가하고, 캐시 접근 지연도 늘어날 수 있다. 블록 크기가 128B로 커지면 locality가 낮은 프로그램에서는 불필요한 데이터까지 불러오게 되어 메모리 대역폭을 낭비할 위험이 있다. 또한 LRU와 같은 교체 정책은 FIFO보다 구현 비용이 크며 하드웨어 자원이 더 필요하다. 마지막으로 associativity 상승과 대형 블록 처리는 전력 소모와 발열을 증가시키기 때문에, 칩 설계에서 전력 제약이 있는 환경에서는 오히려 비효율적일 수 있다.

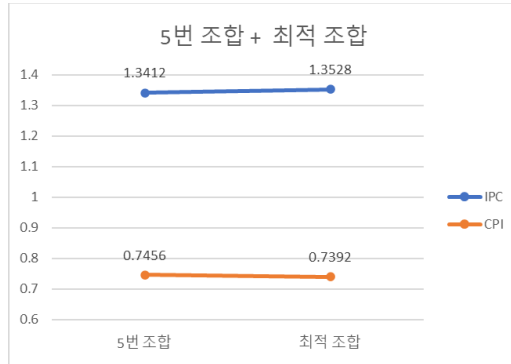
결론적으로, 가장 높은 성능을 제공한 조합은 128B 블록 크기와 4-way 구조를 사용하는 조건 1과 조건 5이며, 이는 locality와 conflict miss 측면에서 가장 이상적인 캐시 환경을 제공하기 때문이다. 그러나 이러한 성능 향상은 회로 복잡도, 면적 증가, 전력·발열 부담 같은 비용이 함께 증가한다는 점에서, 실제 설계에서는 성능과 자원 효율을 균형 있게 고려해야 한다는 점이 확인된다.

6.2 Go

[조건 정보 및 데이터 요약]

Case	Proc_Type	ALU	Scalar	Bpred	Block	Assoc	Policy	IPC	CPI
1 (8 8)	High	4	4	3bit	128	4	l	1.3528	0.7392
2	High	4	4	3bit	128	4	f	1.3525	0.7394
3	High	4	4	3bit	128	4	r	1.3523	0.7395
4	High	4	4	3bit	64	4	l	1.3431	0.7446
5	High	4	4	3bit	64	4	r	1.3409	0.7458
6	Mid	2	2	1bit	128	4	l	0.9772	1.0233
7	Mid	2	2	1bit	128	4	f	0.9771	1.0235
8	Mid	2	2	1bit	128	4	r	0.977	1.0235





[분석 및 해석]

최적의 Configuration 조합 (Case 1) :

Processor: ALU 4개, Scalar 4개 (High performance)

Branch Predictor: 3bit

Cache: Block Size 128B, Associativity 4-way, Policy LRU

성능 지표: IPC 1.3528, CPI 0.7392

이 조합이 최적의 성능을 낸 이유는 Processor의 연산 능력 확장과 Memory 계층의 데이터 공급 능력이 균형을 이루었기 때문이다. 세부적인 선정 근거는 다음과 같다.

Processor: 이전 실험 결과에서 Scalar가 4개일 때 IPC 1.3212로 가장 높았으며, ALU 역시 4개일 때 IPC 1.3212로 가장 높았다. Scalar만 높고 ALU가 적으면 병목이 발생하므로, 두 자원을 4/4로 통일하여 명령어 인출과 실행의 균형을 맞추려고 했다.

Branch Predictor: go 프로그램(바둑 AI)은 조건 분기가 매우 빈번하고 불규칙하므로 3bit 예측기(IPC 1.3325)가 1bit(IPC 1.2438)나 정적 예측(Not-taken/Taken)보다 월등히 높은 성능을 보였다. 이는 3bit 카운터가 go의 복잡한 분기 패턴 히스토리를 더 세밀하게 추적하여 파이프라인 플러시를 최소화했음을 의미하므로 최종 조합에 3bit를 채택하였다.

Cache Memory Block Size: 128B에서 IPC 1.3412로 가장 높은 성능을 보였다. 이는 go가 연속된 메모리 주소를 참조하는 공간적 지역성이 높음을 의미하여, 한 번의 Miss 시 더 큰 데이터를 가져오는 128B를 선정하였다.

Cache Memory Associativity & Policy: 4-way(IPC 1.3212)와 LRU(IPC 1.3316)가 각각의 비교군 내에서 가장 우수했다. 이는 캐시의 충돌을 줄이고 시간적 지역성을 활용하는 것이 중요하기 때문이다.

이전 결과들과 최적조합을 비교하는 그래프들을 활용하여 상호 영향성을 분석해보면,

Processor <-> Memory: 3번 실험(Processor 최적화)의 최고 기록인 1.3212보다 5번 실험

(Cache 최적화)의 최고 기록인 1.3412가 더 높다. 이는 go 벤치마크가 CPU 연산 속도보다 메모리 접근 성능에 더 민감하다는 것을 보여준다.

하지만 Cache만 최적화한 1.3412에서 멈추지 않고, Processor 자원(ALU 4/Scalar 4)을 결합했을 때 1.3528로 성능이 추가 상승하였다. 이는 향상된 캐시 시스템이 데이터를 빠르게 가져오더라도 이를 처리할 연산 유닛이 부족하면 성능 향상에 한계가 있음을 보여주며, Processor와 Memory가 동시에 최적화되어야 최고 성능을 낼 수 있음을 증명한다.

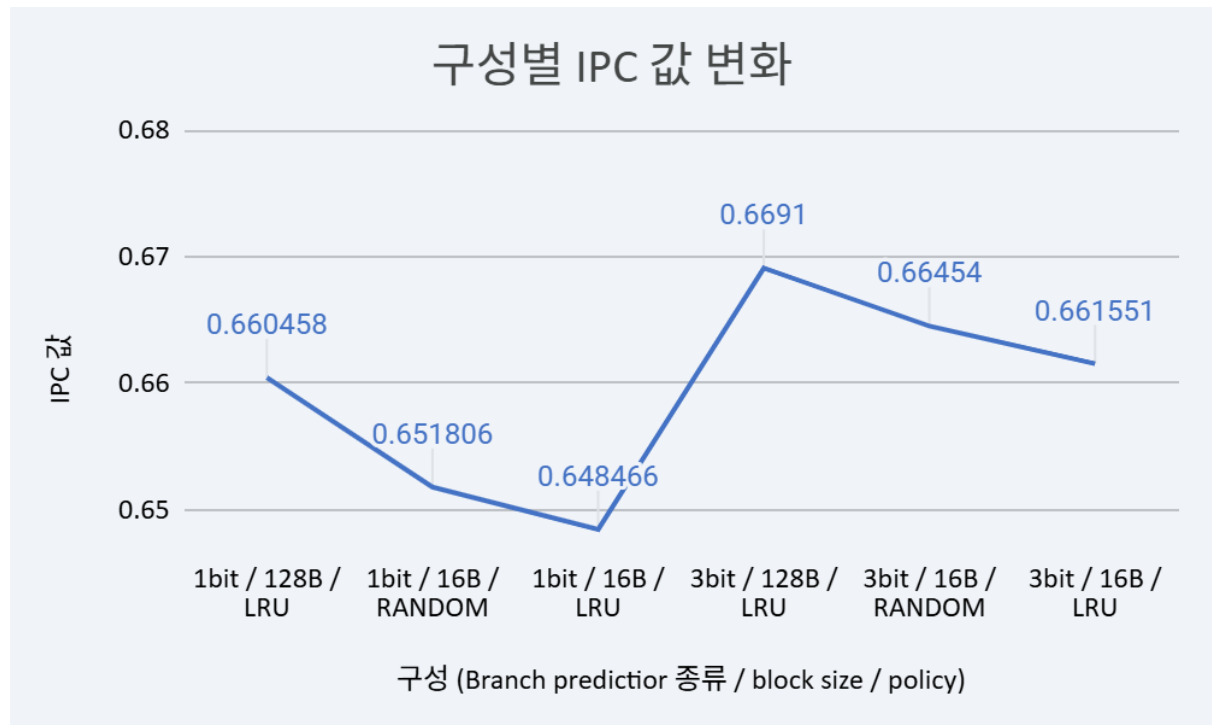
최적 조합인 ALU 4개/ Scalar 4개 / Branch Predictor 3bit / Cache Block Size 128B / Cache Associativity 4-way / Cache Policy LRU 는 IPC 1.3528로 높은 성능을 낼 수 있지만, 설계 및 효율성 측면에서 단점 또한 존재할 수 있다.

하드웨어 비용 및 복잡도 증가: 최적 조합은 Block Size 128B와 4-way Associativity를 채택했는데, 이는 64B나 Direct Mapped 방식에 비해 데이터 버스 대역폭을 많이 차지하고, Tag 비교기 및 MUX 등의 회로 복잡도를 크게 증가시킨다. 또한, ALU와 Scalar를 각각 4개로 유지하는 것은 칩 면적을 많이 차지하게 된다.

전력 소모 대비 효율성: 3, 4번 결과 표를 보면 3 bit 분기 예측기 만으로도 IPC 1.3325를 달성할 수 있었다. 반면, 막대한 하드웨어 자원(ALU 4, 128B Cache, 4-way 등)을 모두 투입한 최종 결과는 1.3528로, 성능 향상 폭(약 1.5%)에 비해 투입되는 하드웨어 자원과 전력 소모가 과도할 수 있다. 즉, 전력 소모 대비 효율성이 떨어지는 구조가 될 수 있다.

Over-fetching 문제: 128B의 큰 블록 사이즈는 공간적 지역성 활용에는 좋지만, go 프로그램의 실행 단계에 따라 불필요한 데이터까지 캐시로 가져오는 Over-fetching 문제를 일으켜 메모리 트래픽을 낭비할 가능성이 있다.

6.3 mcf



연산 장치 수	Scalar의 수	Branch predictor	block size	associativity	policy	cpi	ipc
4	4	1bit	128B	4-way	LRU	1.5141	0.6605
4	4	1bit	16B	4-way	RANDOM	1.5342	0.6518
4	4	1bit	16B	4-way	LRU	1.5421	0.6485
4	4	3bit	128B	4-way	LRU	1.4945	0.6691
4	4	3bit	16B	4-way	RANDOM	1.5048	0.6645
4	4	3bit	16B	4-way	LRU	1.5116	0.6616

[분석]

Block Size와 Replacement Policy의 상호작용 (Interaction)

현상 : Block Size가 작을 때(16B)는 RANDOM 정책이 LRU보다 우수했으나, Block Size를 128B로 확장하자 LRU가 RANDOM을 역전하여 더 높은 성능을 보였다.

원인 : Block Size가 128B로 커지면 전체 캐시 용량이 고정된 상태에서 Set(Entry)의 개수가 줄어든다. Set의 개수가 부족해 경쟁이 치열해진 상황에서는, 무작위로 데이터를 버리는 Random 방식보다 참조의 지역성을 고려해 신중하게 데이터를 교체하는 LRU 방식이 Cache Pollution을 방지하는 데 훨씬 효과적이기 때문이다.

Associativity의 임계점

현상 : 128B Block 사용 시 4-way에서 2-way로 연관도를 낮추자 성능이 약 28% 급락하였다.

원인 : 큰 블록 사이즈(128B)로 인해 Set의 개수가 줄어든 상태에서 Associativity까지

낮아지면, 서로 다른 주소가 같은 Set에 몰릴 때 이를 수용할 공간이 절대적으로 부족해진다. 이로 인해 빈번한 Conflict Miss가 발생하여 성능이 낮아진 것으로 분석된다. 즉, 큰 블록을 사용할 때는 반드시 4-way 이상의 높은 Associativity가 보장되어야 한다.

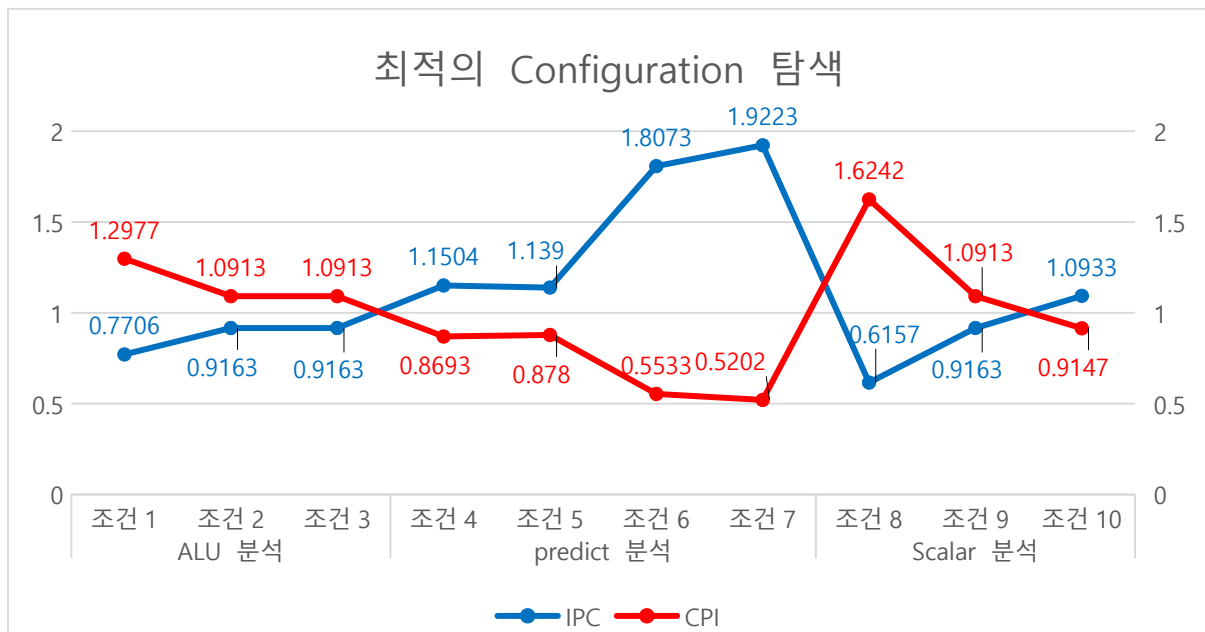
2) Processor 설정과의 결합 (System 최적화)

추가적으로 분기 예측기(Branch Predictor)의 성능을 비교한 결과, 동일한 캐시 설정(128B/4-way/LRU) 하에서 3bit 예측기가 1bit 예측기보다 우수한 성능을 보였다. 이는 mcf의 성능 병목이 주로 메모리(Cache)에 있지만, 정교한 분기 예측을 통해 파이프라인 효율을 높이는 것 또한 성능 향상에 기여함을 보여준다.

3) 최종 결론

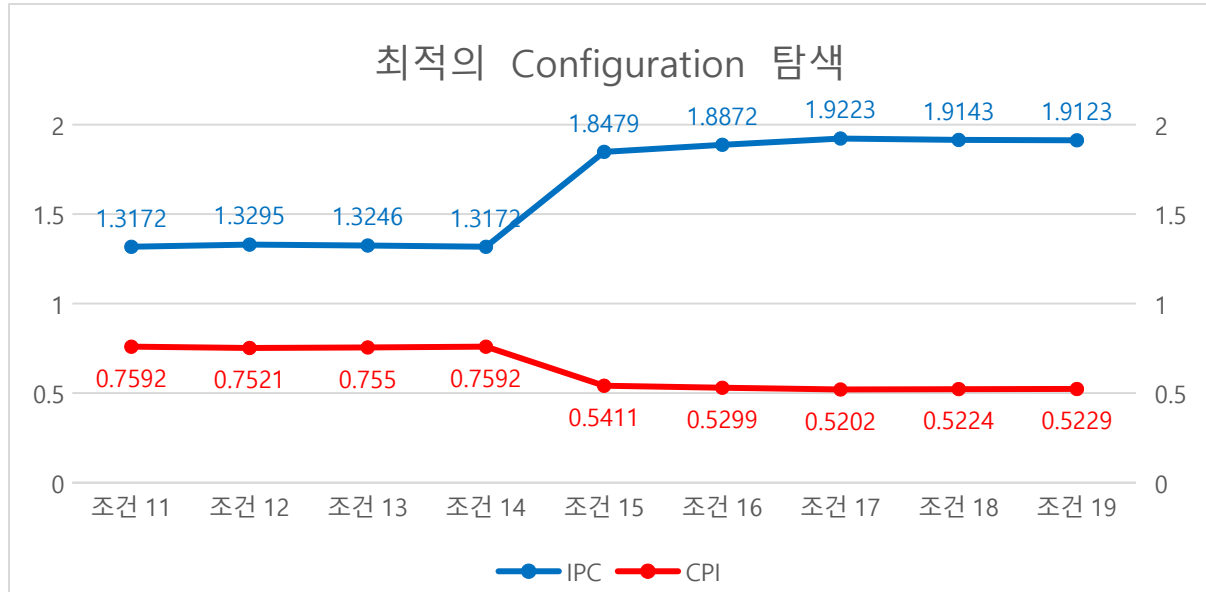
실험 데이터를 종합한 결과, mcf 벤치마크를 위한 최적의 시스템 구성은 [Block Size: 128B, Associativity: 4-way, Policy: LRU, Branch Pred: 3bit]로 선정하였다. 이 조합은 큰 블록 크기를 통해 mcf의 공간적 지역성(Spatial Locality)을 극대화하고, 줄어든 Set 개수로 인한 충돌 부작용을 4-way 연관도와 LRU 정책으로 보완하여 최상의 성능을 달성한 것으로 평가된다.

6.4 gzip



[조건 정보 및 데이터 요약]

	ALU 분석						predict 분석					
	조건1	조건2	조건3	조건4	조건5	조건6	조건7	조건8	조건9	조건10	공통 Cache 조건	
	1	2	4	4	4	4	4	2	2	4	Block_size	
	2 nottaken	2 nottaken	2 nottaken	4 nottaken	4 taken	4 1bit	4 3bit	1 nottaken	2 nottaken	4 nottaken	128B	
IPC	0.7706	0.9163	0.9163	1.1504	1.139	1.8073	1.9223	0.6157	0.9163	1.0933	associativity	
CPI	1.2977	1.0913	1.0913	0.8693	0.878	0.5533	0.5202	1.6242	1.0913	0.9147	4-way	
											replacement_policy	
											LRU	



[조건 정보 및 데이터 요약]

근거	분기 기반으로 3bit를 miss rate를 최소화하기 위해 2/2 조건				block size가 영향을 미치는 경우를 판단하기 위해 동일 조건 block 크기만 변경				associative 성능 분석을 위한 4-way 적용				주로 연산이 ALU에 영향을 받아 4개로 선정			
	2	2 3bit	64B	2-way LRU	2	2 3bit	128B	2-way LRU	2	2 3bit	64B	4-way LRU	4	2 3bit	64B	2-way LRU
IPC																
CPI																
근거	Scalar로 읽어오는 개수와 ALU 개수 일치 미일치 성능 비교				ALU와 Scalar가 4개일 때 성능이 좋으므로 공간적 지역성이 좋을 때 128B 조건 확인				associative 평가를 위해 4way로 앞과 비교				마지막으로 FIFO, RANDOM 비교 분석			
	4	4 3bit	64B	2-way LRU	4	4 3bit	128B	2-way LRU	4	4 3bit	128B	4-way LRU	4	4 3bit	128B	4-way FIFO
IPC																
CPI																
근거	마지막으로 FIFO, RANDOM 비교 분석															
	4	4 3bit	128B	4-way RANDOM												
IPC																
CPI																

[분석 및 해석]

결과 : 조건1부터 19까지 여러 조합의 ALU 개수부터 교체 정책 중 조건 17에서 가장 좋은 결과의 IPC를 얻었다. 이때 Scalar와 ALU 개수 통일하여 4로 gzip에 중요한 branch predict는 3bit를, Block size와 associativity 가장 큰 수인 128B과 4-way를 교체 정책은 LRU를 채택하였다.

이유 : 먼저 Scalar는 한번에 읽어오는 명령어의 개수인 만큼 ALU 수와 통일하여 사용하는 것이 가장 효율적으로 사용 가능할 것으로 판단하여 4/4로 선정하였다. 조건 1~3을 통해 scalar 2의 경우 ALU 수와 연관성을 비교한 결과 동일한 수의 ALU와 Scalar 조합이 가장 좋은 성능을 내는 것을 확인하였다.

Branch Predict는 3-3절에서 분석한 것과 같이 gzip의 특징으로 분기 명령이 매우 많다는 것을 바탕으로 3bit에서 miss 수가 가장 적은 것을 바탕으로 설정 값을 3bit를 선정하였다. 이는 flush를 최소화하고 틀린 분기 연산을 최소화하는 결과를 얻었다.

Block size의 경우 보편적인 값 64B와 공간적 지역성이 우수한 gzip의 경우를 비교하기 위해 128B 값을 비교 분석하였고, 조건 11, 12에서 미미하지만 차이를 확인하여 128B를 선정하게 되었다. 그 후 associativity 분석을 위한 조건 11, 13을 비교 분석하여 최종적으로 miss rate가 가장 적어지는 조건인 128B와 4-way set associativity를 값을 선정하게 되었다.

마지막 교체 정책에 따른 성능 분석을 위한 조건 17~19를 진행하였고, 로직적으로는 RANDOM이 간단하지만 간소한 성능차이로 LRU가 gzip의 시간적 지역성과 연관되어 더 좋은 성능 지표를 갖게 되었다.

단점 : 모든 조건은 gzip에 한정되어 선정된 값이므로 다른 프로그램에 따라서 성능이 천차만별일 것이다.

조건 17이 가장 우수한 IPC를 제공하지만 사용된 하드웨어 자원과 메모리 대역폭이 두 배 수준까지 증가할 수 있다는 점에서 성능/자원 효율 측면에서 단점이 부각된다.

모든 구조가 추가적인 저장 공간과 로직을 요구하게 되면서 설계 난이도, 전력 소모 등이 다른 구조에 비해 많이 요구되고 있다. 결국 설계에 성공하여 운용하더라도 자원을 많이 소모한다는 점에서 단점은 확실하게 부각되고 있다.

7 프로젝트 수행 일지

이름	김정훈(조장)	박수혁	박정원	박효정
모임일자	수행 내용			
11-19	프로젝트 회의 진행 역할 분담 및 프로젝트 기한 설정			
11-20	gzip baseline.cfg 분석 및 실행			
11-21	gzip 3-1절 진행 및 결과 정리	go 3절 Processor 변화에 따른 성능 분석	gcc 3절, 4-1절 실행	mcf 3절 prosector 변화에 따른 성능 분석
11-22	gzip 3, 4절 진행 및 결과 정리	go 4절 Cache 변화에 따른 성능 분석, 결과 정리 및 업로드	gcc 4-2절, 4-3절 실행	mcf 4절 Cache 변화에 따른 성능 분석
11-23			gcc 3절, 4절 실행 결과 정리 및 시각화 자료 준비	mcf 데이터 정리 + 시각화 자료 제작
11-24	진행상황 공유 및 5, 6절 진행 방식 토의			
11-25	gzip 5절 진행 및 결과 정리	go 5절 분석, 결과 정리 및 업로드	gcc 5절 조합 선정 및 실행 1차	mcf 5절 분석, 데이터 정리 + 시각화 자료 제작
11-26	gzip 6절 조건 1~10 진행 및 결과 정리		gcc 5절 조합 선정 및 실행 2차	mcf 6절 최적 조건 분석, 데이터 정리 + 시각화 자료 제작
11-27	gzip 6절 조건 1~10 진행 및 결과 정리	go 최적조건 분석, 결과 정리 및 업로드	gcc 6절 조합 선정 및 실행	
11-28	gzip 6절 조건 11~19 진행 및 결과 정리			mcf 6절 최적 조건에서 오류 발생 확인 후 다른 조합으로 재실험
11-29			gcc 5절, 6절 실행 결과 정리	mcf 6절 최적 조건 분석, 데이터 정리 + 시각화 자료 재제작
11-30		이전 결과에 문제가 있음을 발견, go 최적조건	gcc 5절, 6절 실행 결과 파일 추출 시각화 자료 준비	

		재분석		
12-1	결과 보고서 작성 시작			
12-2	gzip 3, 4절 내용 작성	go (최종) 최적조건 분석, 결과 정리 및 업로드	gcc 보고서 작성 시작	
12-3	gzip 5, 6절 내용 작성	보고서 작성 시작	gcc 보고서 작성 완료	mcf 보고서 작성 완료
12-4	2번 configuration cache 내용 작성	보고서 작성 완료		
12-5	1번 프로젝트 개요 작성			
12-6				
12-7				
12-8				
12-9				
12-10				
12-11				
12-12	PPT 제작 및 발표 대본 작성	PPT 및 발표 대본 작성 완료	PPT 개인 파트 제작 시작	
12-13	녹화 진행		PPT 개인 파트 제작 완료 및 대본 작성 보고서 오타자 및 일부 내용 수정	발표 ppt 개인 파트 제작 시작+ 완료, 보고서 오타자 수정
12-14	자료 업로드			

8 Troubleshooting

발생 오류	해결 상태	발견자	과정	발생시점	메모 및 내용
gzip 실행 시간	해결 완료	김정훈	기타	2025.11.20	압축률 설정에 따른 프로그램 실행 시간이 길어져 13시간 동안 처리. 설정 값 인자를 1로 설정하여 시간 단축
6번 최적화 결과	해결 완료	박효정	기타	2025.11.28	6절에서 최적의 cache 조합을 찾아내는 과정에서 5 절의 조합보다 IPC 값이 작게 나왔다. 이는 실험 값을 조정하는 과정에서 오류가 있었다고 판단하여 다시 조합을 재조합해서 실험을 진행하였다. 다시 진행해본 결과 IPC가 가장 크게 나왔다.
결과 파일이 빈 파일로 저장됨	해결 완료	박효정	기타	2025.11.29	결과를 저장하려고 > result.txt를 썼는데 나중에 파일을 열어보니 아무 내용도 없음. 이는 simplescalar는 시뮬레이션 결과를 표준 출력이 아니라 표준 에러로 출력하기 때문에 >을 쓰면 결과가 파일이 안 담기기 때문이었음. > 대신에 >2를 사용하여 파일로 저장하니 해결됨.
가상 머신에서 Windows로 cfg 결과 파일 추출	해결 완료	박정원	기타	2025.11.30	Virtual Box에서 드래그 앤 드롭 기능(복사& 붙여넣기)으로 해결 시도 -> 실패 (Ubuntu 재부팅까지 했지만 설정 오류인지 드래그 앤 드롭 기능 활성화가 안 됨) Firefox explorer 활용하여 파일 추출 성공
6번 최적화 과제 결과 IPC 값	해결 완료	박수혁	기타	2025.11.30	6번 과제를 수행하는 과정에서 최적화 결과를 도출했는데, 그 값이 5번 캐시 최적화 결과 값보다 작은 값이 나오는 문제가 발생했었다. 이는 최적화를 했다고 보기 어려운 결과이므로 문제로 판단했고, 다시 수행과정들을 복기하며 문제를 발견했다. 최적화를 하는 과정에서 Cache associativity는 d11 만 수정, Cache block size는 d11, d12 만 수정해야하는데, 이를 정확하게 수행하지 못해 문제가 발생했었다. 문제 해결 후 정상적인 결과를 얻을 수 있었음.