수업 명 : 컴퓨터구조

과제 이름 : 프로젝트 #4

학 과: 컴퓨터정보공학부

담당 교수님: 이성원 교수님

분 반: 월3, 수4

학 번: 2023202043

성 명: 최은준

1. Introduction

이 프로젝트는 컴퓨터 시스템에서의 Memory Hierarchy중 Cache의 동작을 이해하고, 시뮬레이션을 통해 다양한 캐시 구성에 따른 성능 변화를 분석한다. 특히, SimpleScalar 시뮬레이터를 이용하여 실제 프로그램이 실행되는 동안 명령어와 데이터가 어떻게 캐시에 접근되는지, 그리고 그 결과로 발생하는 hit/miss와 그에 따른 Average Memory Access Time (AMAT)을 측정한다.

1단계에서는 Bubble Sort와 Random Access 프로그램을 기반으로 I-Cache와 D-Cache의 hit/miss 동작을 시각적으로 분석하고, 각 접근 패턴이 캐시 동작에 어떤 영향을 주는지를 살펴본다. 2단계에서는 go, m88ksim, swim을 대상으로 다양한 캐시 구조(Unified vs Split), 캐시 크기, 블록 사이즈, 매핑 방식(Direct-mapped vs Set-associative) 등을 구성하고, AMAT를 기준으로 최적의 캐시 설정을 찾는 과정을 수행한다.

이 과정을 통해 다양한 프로그램의 접근 패턴에 따라 적절한 캐시 구조가 달라진다는 것을 이해할 수 있다.

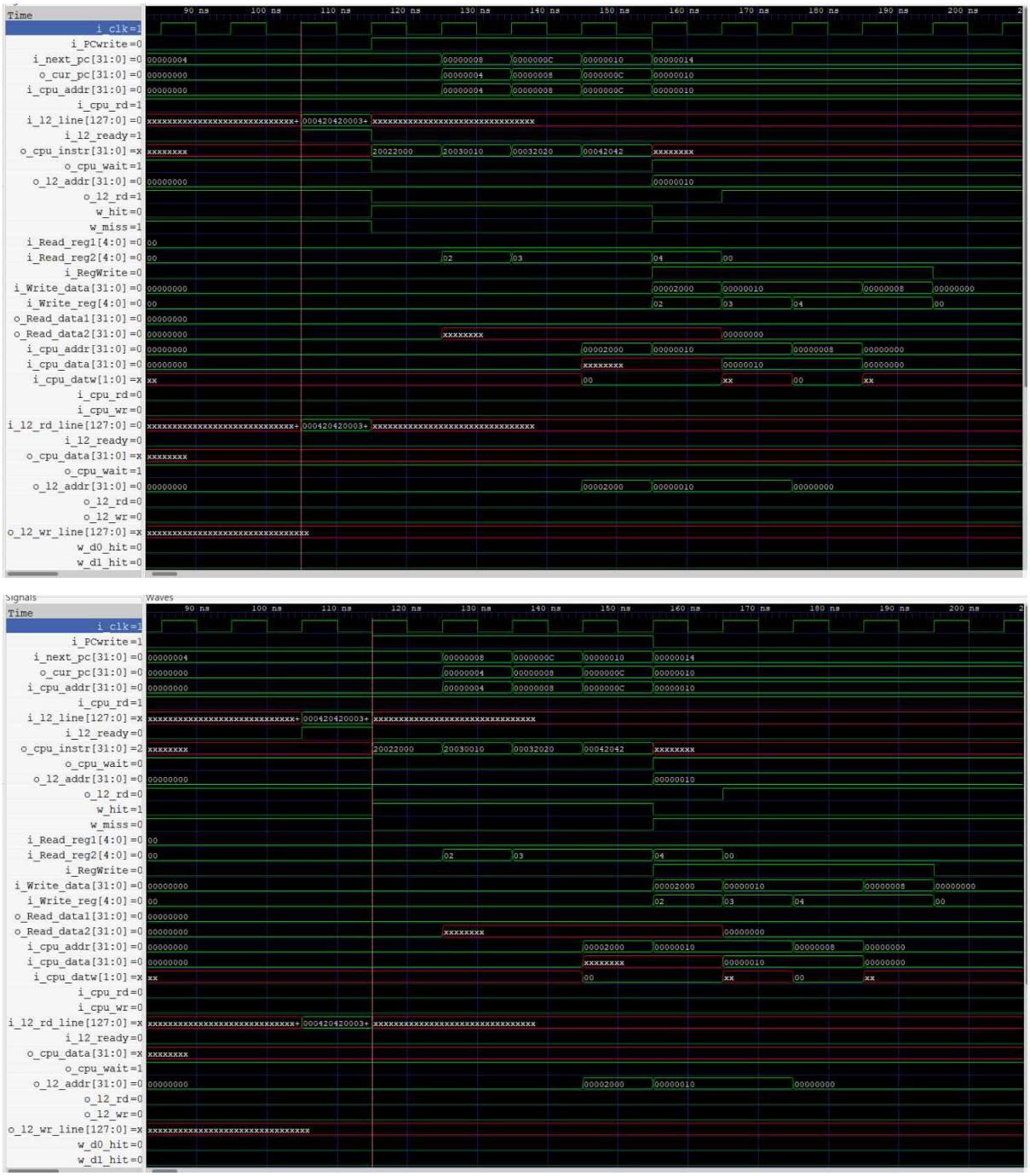
2.1 Observation for Program Behaviors

1. I/D 캐시 hit/miss 타이밍

◆ Bubble Sort (sort_text.txt)

-Instruction cache (I-Cache) : 반복 구조가 많고, loop 중심 코드(L1, L2, L3, L4)는 연속적인 주소 공간에서 실행됨. 따라서 I-Cache Hit률 높음 (예: j L3, beq, bltz, sll, add등이 loop 내 반복 실행됨)

-Data cache (D-Cache): lw, sw명령어 다수 사용한다. 주소는 $6, $10등 정렬 대상 배열의 인덱스를 기반으로 계산되므로 데이터 접근은 일정 부분에서 반복한다. 따라서 데이터 캐시 Hit 비율도 일정 수준 확보 가능하나 정렬 시 계속 다른 인덱스를 참조하므로 반복이 깊어질수록 D-Cache Miss증가가 가능하다.



위 사진은 프로그램 시작 직후, PC = 0x00000000인 상태에서 첫 명령어(addi $2, $0, 0x2000)를 fetch하려고 한다. 그러나 해당 명령어가 위치한 L1 Instruction Cache에는 해당 block이 존재하지 않아 miss가 발생한다. 따라서 프로세서는 L2 Cache로 요청을 보낸다. L2 Cache에도 해당 block이 없기 때문에 L2에서도 miss가 발생한다. 이로 인해 main memory로부터 block 단위로 데이터를 읽어와서 L2 Cache에 저장하고, 이후 해당 블록을 L1 Instruction Cache로 전달한다.Instruction Cache가 데이터를 수신한 후, 다시 fetch를 재시도하고 이번에는 hit이 발생하여 명령어가 정상적으로 실행된다.

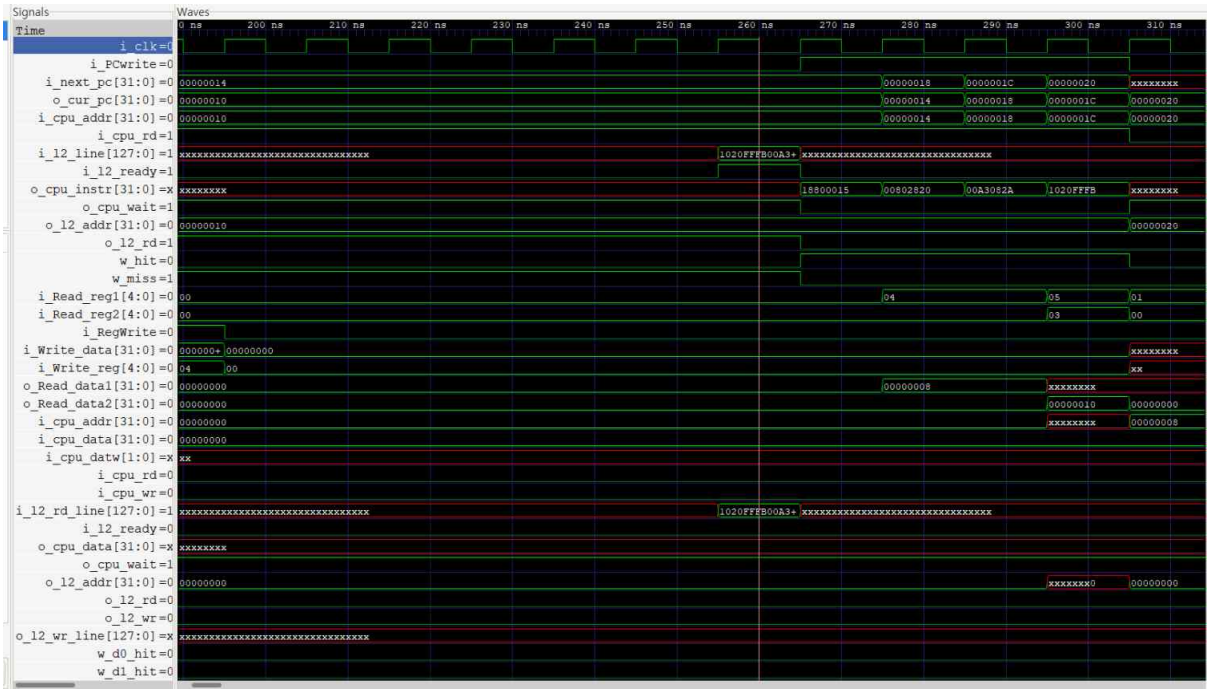
Instruction cache에 처음 접근하므로 L1 I-cache miss 발생

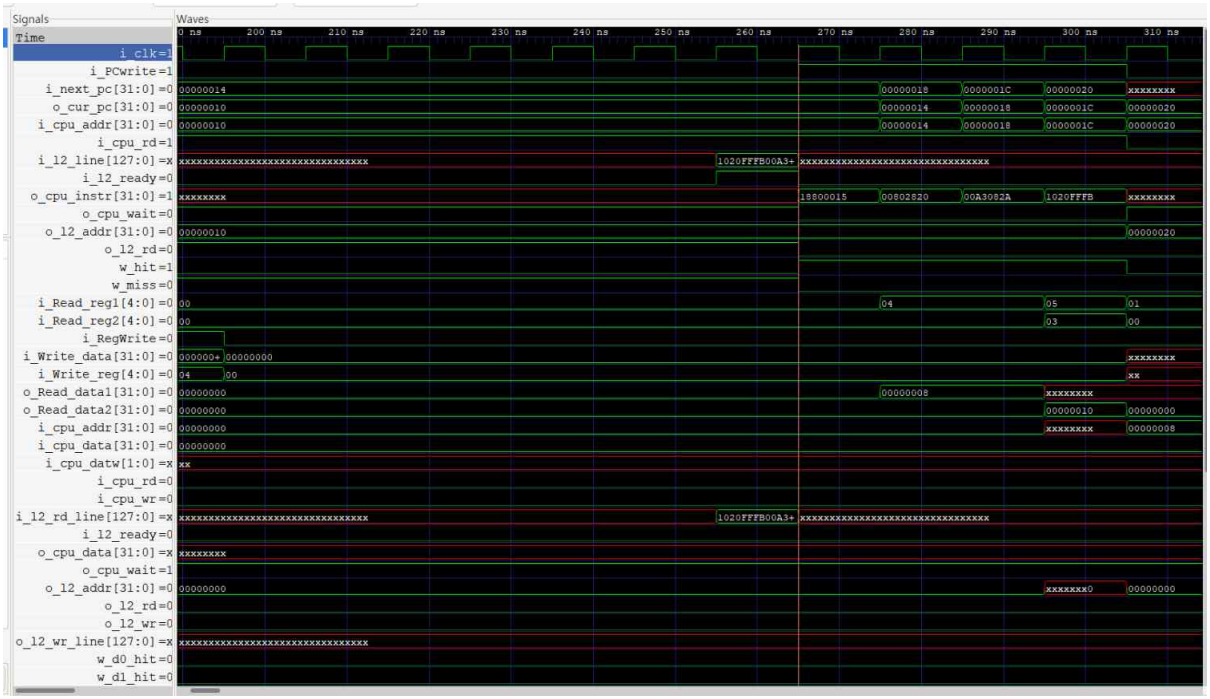
L2 cache에서도 해당 block이 없어 miss 발생 → Main memory로부터 instruction block fetch

L2에 block 적재 후, L1 I-cache로 복사 → fetch된 명령어 실행 가능

이후 인접한 명령어들이 연속적으로 실행되므로 PC+4로 진행하면서 fetch됨

이 명령어들은 모두 같은 block 또는 인접 block에 포함되어 있어 I-cache hit 발생





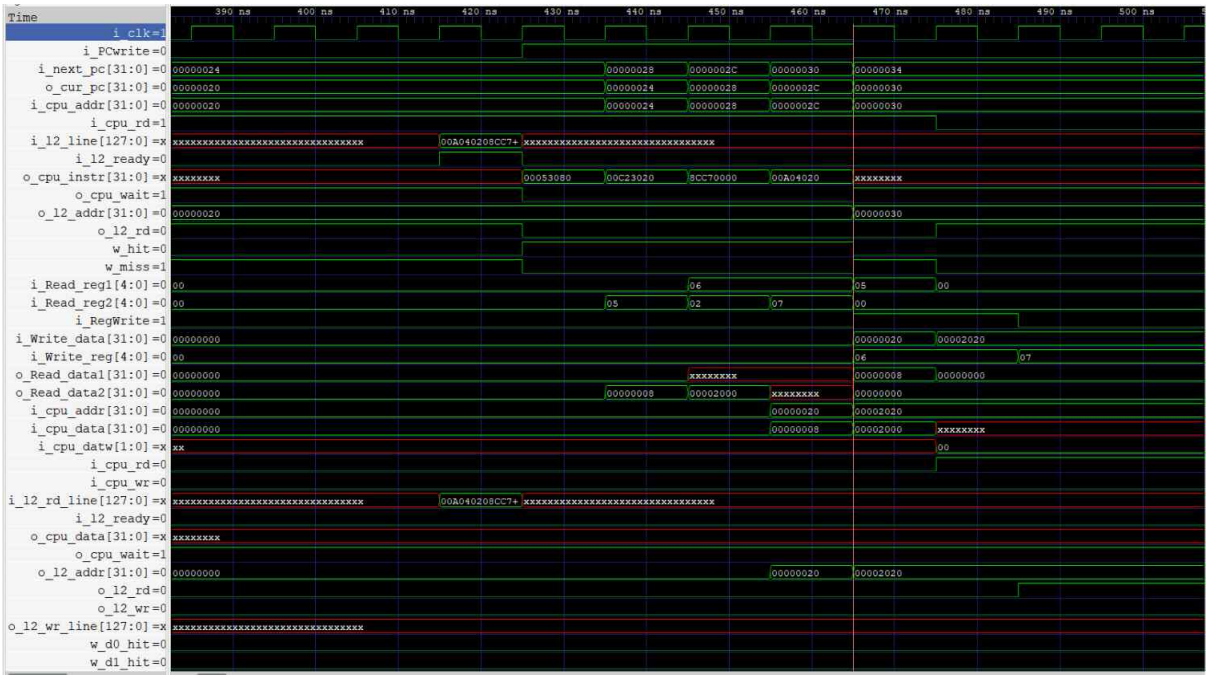
이후 명령어들은 연속된 주소(예: 0x04, 0x08, 0x0C 등)에 위치해 있어, 한 번 로딩된 cache block 내의 명령어들은 모두 I-Cache hit이 발생한다. 반복 루프 (L1, L2, L3, L4) 내부의 명령어들도 계속

재사용되므로, I-Cache에서는 더 이상 miss가 거의 발생하지 않는다.

루프를 통해 동일한 instruction을 반복 실행

반복 횟수가 많고, 루프 내 명령어들이 한 블록 내 또는 근접 블록에 위치함

→ Instruction cache hit 비율 매우 높음



lw $7, 0($6)또는 sw $11, 0($6)등의 명령어는 배열 원소에 접근하기 위해 Data Cache를 이용한다. 이때 $6레지스터는 base 주소 $2 = 0x2000에서 인덱스만큼 계산한 주소이며, 초기에 해당 주소 범위는 D-Cache에 없기 때문에 D-Cache miss 발생한다. L2 Cache를 거쳐 main memory에서 데이터를 로딩한 뒤, D-Cache에 저장되고 다시 접근하면 hit 발생한다.

add, sll, add, lw, sw를 통해 정렬할 배열에 접근

인덱스 기반 선형 접근: $6 = base + index × 4

인접 데이터 접근이 많고, 일부 인덱스는 반복 사용됨 → D-cache에서도 일정 수준 hit 가능

하지만 swap 중에 새로운 인덱스 접근 발생 → 간헐적 L1 D-cache miss, 이후 L2 → main memory

이후 루프에서 동일한 배열 주소(또는 인접 주소)에 반복적으로 접근하면서 Data Cache에도 점차 hit 비율이 증가한다. Spatial locality 덕분에 한 번 로딩한 block 내의 다른 데이터에도 접근하므로 miss가 줄어든다.

4. 전체 흐름 요약

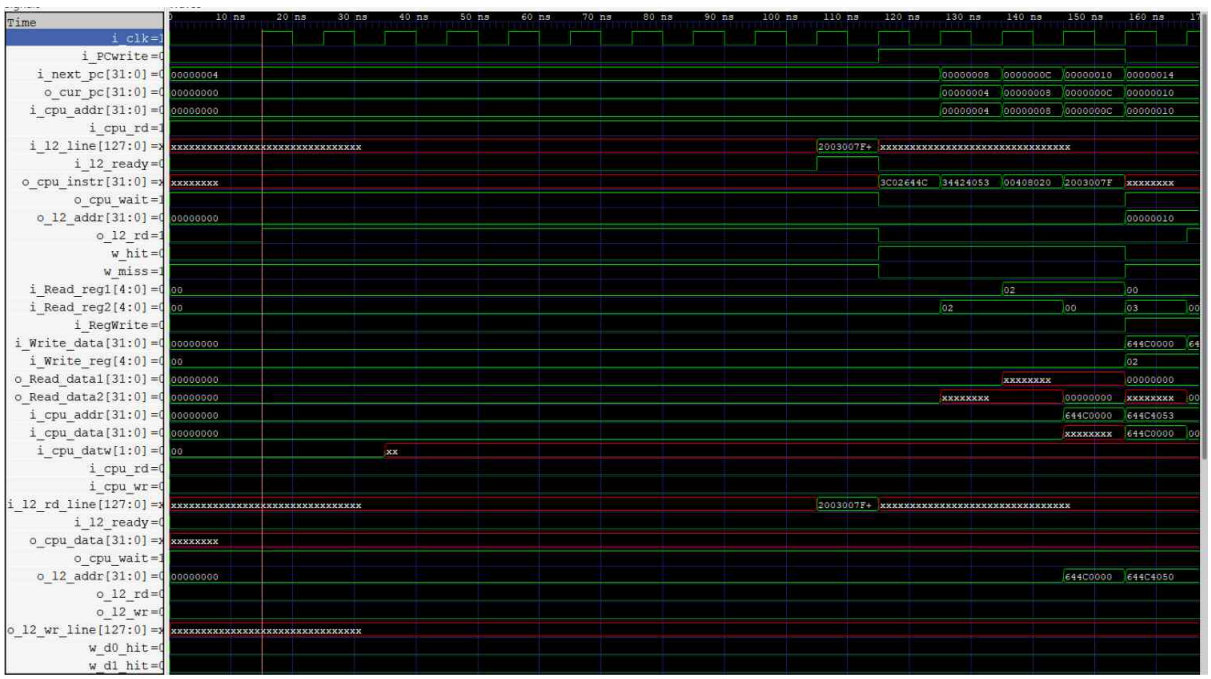
프로그램 실행 초기에 L1 Instruction Cache에는 명령어 블록이 존재하지 않아 miss가 발생하고, L2 및 main memory로부터 블록 단위로 데이터를 로딩한 후 fetch가 재시도되어 hit이 발생한다. 이후 루프 구조에서 instruction reuse가 높아 I-Cache는 대부분 hit을 유지하게 되며, Data Cache는 배열 접근 시 초기 miss가 발생하나 반복 접근을 통해 점차 hit 비율이 높아지는 구조를 갖는다.

◆ **Random Access (random_access.txt)**

Instruction cache (I-Cache): 전체적으로 코드 길이가 짧고 loop 구조 (L1, L2)를 갖고 있음.

따라서 instruction 접근은 반복되고 일정 → I-Cache Hit율 높음

Data cache (D-Cache): 접근 주소가 xor, sll, sra, andi등으로 연산되어 완전히 무작위(random), 같은 주소를 거의 다시 접근하지 않음 → D-Cache Hit률 낮고 Miss 빈도 매우 높음 대표적인 random access의 특징: 시간/공간 지역성 거의 없음



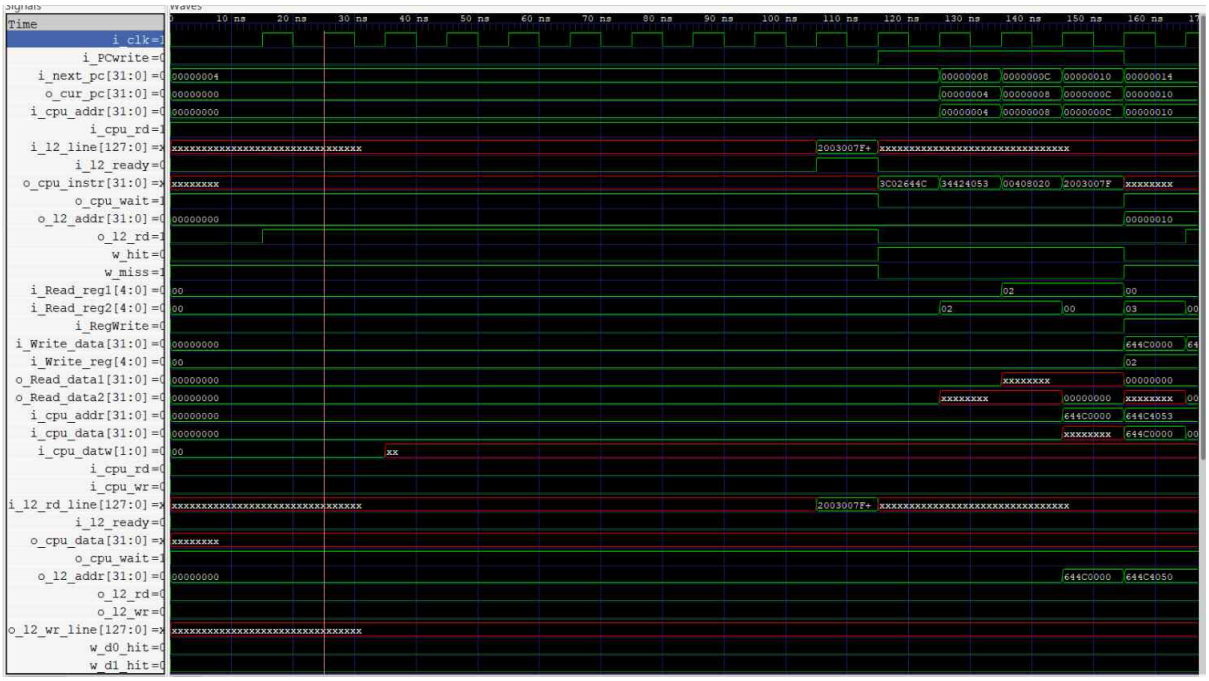
위 사진은 프로그램이 처음 실행되며 PC = 0x00000000에서 첫 명령어(lui $2, 0x644c)를 fetch하려고 시도하는 사진이다. 해당 명령어가 위치한 block이 L1 Instruction Cache에 존재하지 않아서 miss가 발생한다. 이후 L2 Cache로 요청을 보내지만, L2에도 없기 때문에 main memory로부터 block을 불러와 L2에 저장한다. L2가 ready 되면 해당 block이 L1 I-Cache로 전달되고, fetch 재시도 시 hit이 발생하여 명령어가 실행된다.

처음 명령어 fetch 시 L1 I-cache miss 발생

L2에도 해당 block이 없어 miss → main memory로부터 instruction block fetch

이후 명령어들이 순차적으로 실행됨 → 연속 주소 접근 → PC+4 증가

같은 블록 내 명령어 사용 → 이후 I-cache hit



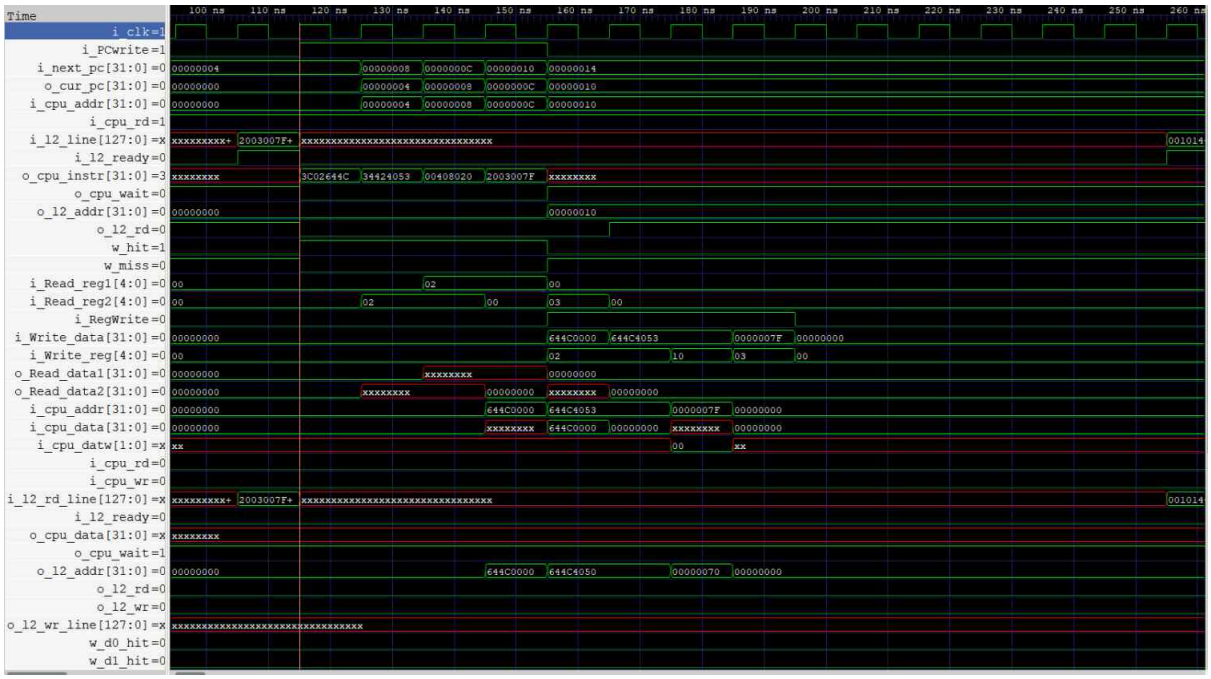
이후 명령어들은 대부분 루프 안에 위치하고, 특히 sll, xor, andi, bne, sw, lw등은 반복적으로 재사용된다. 따라서 초반 miss 이후에는 Instruction Cache hit 비율이 매우 높아진다. 단, bne분기 명령어 수행 시 예측 실패가 발생하면, 이후에 fetch한 명령어가 파이프라인 flush로 인해 무효화되며 다시 fetch되는 과정에서 hit/miss가 재발생할 수 있다.

$2, $16등 레지스터 간 복잡한 논리연산 (xor, sll, sra)으로 메모리 주소 계산

최종적으로 $5에 주소 계산 후, sw, lw명령어로 메모리 접근

하지만 계산된 주소가 매우 불규칙 → 캐시의 공간 지역성 전혀 없음

→ L1 D-cache miss 자주 발생, L2에서도 miss → main memory 접근 반복



Random Access 프로그램은 $16레지스터에 난수 연산을 수행해 $5에 최종 주소를 계산하고, 해당 주소로 sw $16, 0($5)또는 lw $6, 0($5)를 수행한다. 이 주소가 매번 달라지므로 D-Cache에는 존재하지 않는 경우가 대부분이며, miss가 빈번히 발생한다. 결과적으로 D-Cache miss → L2 접근 → L2 miss 시 main memory → 이후 D-Cache로 데이터 적재 → 다음 접근에 hit 가능 (하지만 다음 주소는 또 달라짐)

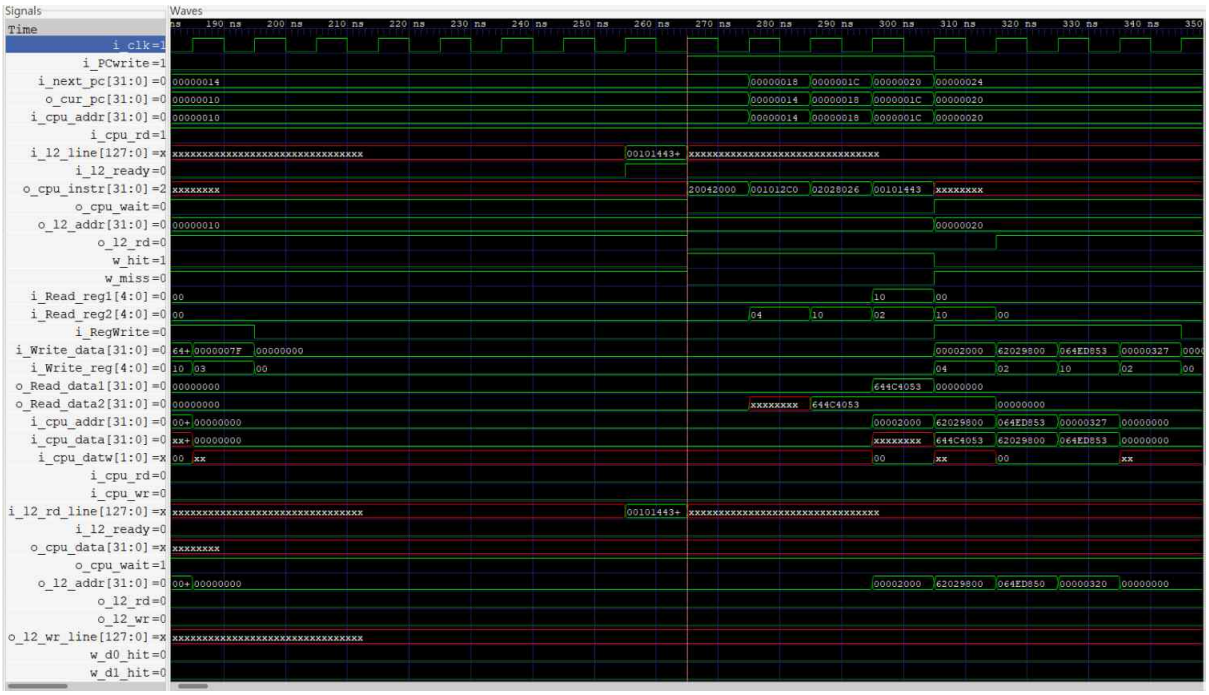
명령어: slt $1, $16, $0

bne $1, $0, L2

조건이 참이면 L2레이블로 분기 → branch taken

이 경우 분기 예측 실패로 인해, 이후에 이미 fetch된 명령어(break)는 flush

Control hazard 발생→ instruction pipeline에서 버블 삽입



Random Access 특성상, 데이터는 대부분 spatial locality와 temporal locality가 거의 없기 때문에, D-Cache는 항상 새로운 주소에 접근하게 되고 miss가 지속적으로 발생한다. 특히 block size를 늘려도 도움이 되지 않으며, set-associativity 증가만으로도 효과가 제한적이다.

루프 조건이 $3값을 감소시키며 수행됨 (addi $3, $3, -1)

loop마다 완전히 새로운 메모리 주소에 접근하므로, 매 loop마다 L1/L2 모두 miss 가능성 높음

AMAT 증가, pipeline stall 빈번

4. 전체 흐름 요약

Random Access 프로그램은 실행 초기에 L1 Instruction Cache miss가 발생하나, 이후 루프 구조로 인해 대부분의 명령어는 캐시에 유지되며 hit 비율이 높아진다. 반면, 데이터 접근은 난수 기반 주소 계산을 통해 무작위적 위치에 수행되므로, D-Cache는 반복적으로 miss가 발생하고, locality가 거의 없어 캐시 효율이 매우 낮은 구조를 갖는다. 이는 Bubble Sort와 대조적으로, instruction cache는 효율적이나 data cache 측면에서는 매우 비효율적인 대표적 사례로 볼 수 있다.

2. L2 접근 시점

◆ **Bubble Sort**

D-cache에서 Miss가 발생하면 L2로 접근.

반복되는 인덱스 기반 정렬이므로 일부 데이터는 L1에 존재하지만,

새로운 인덱스를 접근하거나 swap 시에는 L2 접근이 간헐적으로 발생

L2도 일정 수준의 hit률을 보일 수 있음.

◆ **Random Access**

대부분의 D-cache 접근이 miss이므로 거의 매번 L2 접근 발생.

그리고 random 주소이므로 L2에도 해당 데이터가 존재할 확률 낮음 → Main Memory까지 접근 빈번

이는 전체적인 AMAT(평균 메모리 접근 시간)를 크게 증가시킴

3. Bubble Sort vs Random Access 접근 패턴 비교

항목	Bubble Sort	Random Access
I-Cache 접근	루프 기반, 반복적 (high hit)	루프 기반, 간결 (high hit)
D-Cache 접근	배열 인덱스 기반 (중간 hit/miss)	주소가 무작위 (high miss)
공간 지역성	있음 (배열 순차 접근)	거의 없음
시간 지역성	있음 (loop 내 동일 주소 반복 접근)	없음
캐시 효율	비교적 효율적	매우 낮음
예상 AMAT	낮음 (1~2단계 miss)	높음 (2~3단계 miss → L2 miss, DRAM 접근)

4. 데이터 크기 변화에 따른 영향

◆ **Bubble Sort**

데이터 크기가 작을 때는 전체 배열이 L1 or L2에 존재 가능 → High hit rate

크기가 커지면 일부만 캐시에 적재 → swap/접근 증가 시 D-cache miss 증가

하지만 접근 패턴이 예측 가능하므로 prefetching 또는 block locality 활용 가능

◆ **Random Access**

데이터 크기 변화와 관계없이 hit률 낮음

random pattern 자체가 캐시 구조와 상충됨

데이터가 클수록 miss율은 더 높아지고 L2 → Main Memory 접근 증가

비교 항목	Bubble Sort	Random Access
I-Cache 효율	매우 높음	매우 높음
D-Cache 효율	중간 (순차 접근 기반)	매우 낮음 (무작위 접근)
L2 접근 빈도	간헐적으로 발생	매우 자주 발생
Main memory 접근 빈도	낮음	높음
캐시 적합성	일반적인 캐시 구성에 적합	캐시 효과가 거의 없음
데이터 크기 영향	증가 시 miss 증가 (그래도 예측 가능)	증가 시 miss 증가 (예측 불가)

2.2 Cache Simulation

사용한 설정 파일(시뮬레이터: SimpleScalar (sim-cache))

```
-cache:il1 il1:32:16:1:l        # L1 Instruction cache: 32 sets, 16B block, 1-way, LRU
-cache:dl1 none                 # Data L1 cache 없음 (Split/Unified 전환 시 사용)
-cache:il2 il2:256:64:1:l       # L2 Instruction cache (L2 Unified로 활용)
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

CINT95 Benchmarks는 다음과 같다.

go: 게임 AI 프로그램으로, 바둑 게임에서의 다음 수를 계산한다. 많은 상태 탐색과 시뮬레이션을 필요로 하며 상태 공간 탐색, 브랜치(분기), 재귀 호출이 많다.

m88ksim: 모토로라 88100 아키텍처 기반의 CPU 시뮬레이터다. 다른 프로그램을 해석하고 실행하는 시뮬레이션 프로그램으로 내부적으로 많은 명령어 fetch 및 register-level 연산 수행한다.

swim: 수치해석 기반의 날씨 시뮬레이션프로그램이다. 매우 큰 2D 배열 데이터를 기반으로 계산

수행하며 과학 계산 중심의 구조로 순차적/규칙적인 접근이 많다.

1. Unified vs Split Cache

구성 유형	특징	시사점
Unified Cache	명령어와 데이터를 하나의 캐시에 저장 → 공간 공유, 유연함	Instruction/Data 병합이 메모리 활용 효율적이나 경합 가능성
Split Cache	Instruction / Data 분리 → 고정된 캐시 공간	병렬 fetch 가능, 경합 감소, 전체 miss rate 낮을 가능성

```
1 -cache:il1 dl1
2 -cache:dl1 ul1:64:16:1:l
3
4 -cache:il2 none
5 -cache:dl2 none
6 -tlb:itlb none
7 -tlb:dtlb none
```

Unified cache 64 Sets

```
1 -cache:il1 dl1
2 -cache:dl1 ul1:128:16:1:l
3
4 -cache:il2 none
5 -cache:dl2 none
6 -tlb:itlb none
7 -tlb:dtlb none
```

Unified cache 128 Sets

```
1 -cache:il1 dl1
2 -cache:dl1 ul1:256:16:1:l
3
4 -cache:il2 none
5 -cache:dl2 none
6 -tlb:itlb none
7 -tlb:dtlb none
```

Unified cache 256 Sets

```
1 -cache:il1 dl1
2 -cache:dl1 ul1:512:16:1:l
3
4 -cache:il2 none
5 -cache:dl2 none
6 -tlb:itlb none
7 -tlb:dtlb none
```

Unified cache 512 Sets

```
1 -cache:il1 il1:32:16:1:l
2 -cache:dl1 dl1:31:16:1:l
3
4 -cache:il2 none
5 -cache:dl2 none
6 -tlb:itlb none
7 -tlb:dtlb none
```

Split cache 64 Sets

```
1 -cache:il1 il1:64:16:1:l
2 -cache:dl1 dl1:64:16:1:l
3
4 -cache:il2 none
5 -cache:dl2 none
6 -tlb:itlb none
7 -tlb:dtlb none
```

Split cache 128 Sets

```
1 -cache:il1 il1:128:16:1:l
2 -cache:dl1 dl1:128:16:1:l
3
4 -cache:il2 none
5 -cache:dl2 none
6 -tlb:itlb none
7 -tlb:dtlb none
```

Split cache 256 Sets

```
1 -cache:il1 il1:256:16:1:l
2 -cache:dl1 dl1:256:16:1:l
3
4 -cache:il2 none
5 -cache:dl2 none
6 -tlb:itlb none
7 -tlb:dtlb none
```

Split cache 512 Sets

1-1. go

```
sim: ** simulation statistics **
sim_num_insn              388626283 # total number of instructions executed
sim_num_refs              110512241 # total number of loads and stores
  executed
sim_elapsed_time                 40 # total simulation time in seconds
sim_inst_rate          9715657.0750 # simulation speed (in insts/sec)
il1.accesses              388626283 # total number of accesses
il1.hits                  232796939 # total number of hits
il1.misses                155829344 # total number of misses
il1.replacements          155829312 # total number of replacements
il1.writebacks                    0 # total number of writebacks
il1.invalidations                 0 # total number of invalidations
il1.miss_rate                0.4010 # miss rate (i.e., misses/ref)
il1.repl_rate                0.4010 # replacement rate (i.e., repls/ref)
il1.wb_rate                  0.0000 # writeback rate (i.e., wrbks/ref)
il1.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
il2.accesses              155829344 # total number of accesses
il2.hits                  137477116 # total number of hits
il2.misses                 18352228 # total number of misses
il2.replacements           18351972 # total number of replacements
il2.writebacks                    0 # total number of writebacks
il2.invalidations                 0 # total number of invalidations
il2.miss_rate                0.1178 # miss rate (i.e., misses/ref)
il2.repl_rate                0.1178 # replacement rate (i.e., repls/ref)
il2.wb_rate                  0.0000 # writeback rate (i.e., wrbks/ref)
il2.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
ld_text_base             0x00400000 # program text (code) segment base
ld_text_size                 621600 # program text (code) size in bytes
ld_data_base             0x10000000 # program initialized data segment base
ld_data_size                 578004 # program init'ed `.data' and
  uninit'ed `.bss' size in bytes
ld_stack_base            0x7fffc000 # program stack segment base (highest
  address in stack)
ld_stack_size                 16384 # program initial stack size
ld_prog_entry            0x00400140 # program entry point (initial PC)
ld_environ_base          0x7fff8000 # program environment base address
  address
ld_target_big_endian              0 # target executable endian-ness, non-
  zero if big endian
mem.page_count                  285 # total number of pages allocated
mem.page_mem                   1140k # total size of memory pages allocated
mem.ptab_misses                 285 # total first level page table misses
```

# of Sets	Undefied cache Miss rate	Undefied cache AMAT	Split cache		Split cache AMAT
			Inst. Miss rate	Data Miss rate	
64	0.4010	81.24	0.3987	0.3879	78.936
128	0.3472	70.52	0.3589	0.1927	55.652
256	0.2759	56.3	0.3012	0.1605	51.745
512	0.2064	42.4	0.2078	0.1236	40.745

1-2. m88ksim

```
sim: ** simulation statistics **
sim_num_insn              47918092 # total number of instructions executed
sim_num_refs              11092043 # total number of loads and stores
 executed
sim_elapsed_time                3 # total simulation time in seconds
sim_inst_rate          15972697.3333 # simulation speed (in insts/sec)
il1.accesses              47918092 # total number of accesses
il1.hits                  29033385 # total number of hits
il1.misses                18884707 # total number of misses
il1.replacements          18884675 # total number of replacements
il1.writebacks                  0 # total number of writebacks
il1.invalidations               0 # total number of invalidations
il1.miss_rate              0.3941 # miss rate (i.e., misses/ref)
il1.repl_rate              0.3941 # replacement rate (i.e., repls/ref)
il1.wb_rate                0.0000 # writeback rate (i.e., wrbks/ref)
il1.inv_rate               0.0000 # invalidation rate (i.e., invs/ref)
il2.accesses              18884707 # total number of accesses
il2.hits                  18854966 # total number of hits
il2.misses                   29741 # total number of misses
il2.replacements             29485 # total number of replacements
il2.writebacks                   0 # total number of writebacks
il2.invalidations                0 # total number of invalidations
il2.miss_rate              0.0016 # miss rate (i.e., misses/ref)
il2.repl_rate              0.0016 # replacement rate (i.e., repls/ref)
il2.wb_rate                0.0000 # writeback rate (i.e., wrbks/ref)
il2.inv_rate               0.0000 # invalidation rate (i.e., invs/ref)
ld_text_base            0x00400000 # program text (code) segment base
ld_text_size               286816 # program text (code) size in bytes
ld_data_base            0x10000000 # program initialized data segment base
ld_data_size               130436 # program init'ed `.data' and
 uninit'ed `.bss' size in bytes
ld_stack_base           0x7fffc000 # program stack segment base (highest
 address in stack)
ld_stack_size               16384 # program initial stack size
ld_prog_entry           0x00400140 # program entry point (initial PC)
ld_environ_base         0x7fff8000 # program environment base address
 address
ld_target_big_endian            0 # target executable endian-ness, non-
 zero if big endian
mem.page_count               1082 # total number of pages allocated
mem.page_mem                 4328k # total size of memory pages allocated
```

# of Sets	Undefied cache Miss rate	Undefied cache AMAT	Split cache		Split cache AMAT
			Inst. Miss rate	Data Miss rate	
64	0.3941	79.86	0.405	0.269	78.31
128	0.2614	69.34	0.337	0.104	56.72
256	0.1375	28.62	0.226	0.141	40.9
512	0.1082	22.8	0.191	0.087	35.16

1-3. swim

```
sim: ** simulation statistics **
sim_num_insn              213188066 # total number of instructions executed
sim_num_refs               66098376 # total number of loads and stores
 executed
sim_elapsed_time                 18 # total simulation time in seconds
sim_inst_rate          11843781.4444 # simulation speed (in insts/sec)
il1.accesses              213188066 # total number of accesses
il1.hits                  147873095 # total number of hits
il1.misses                 65314971 # total number of misses
il1.replacements           65314939 # total number of replacements
il1.writebacks                    0 # total number of writebacks
il1.invalidations                 0 # total number of invalidations
il1.miss_rate                0.3064 # miss rate (i.e., misses/ref)
il1.repl_rate                0.3064 # replacement rate (i.e., repls/ref)
il1.wb_rate                  0.0000 # writeback rate (i.e., wrbks/ref)
il1.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
il2.accesses               65314971 # total number of accesses
il2.hits                   64382052 # total number of hits
il2.misses                   932919 # total number of misses
il2.replacements             932663 # total number of replacements
il2.writebacks                    0 # total number of writebacks
il2.invalidations                 0 # total number of invalidations
il2.miss_rate                0.0143 # miss rate (i.e., misses/ref)
il2.repl_rate                0.0143 # replacement rate (i.e., repls/ref)
il2.wb_rate                  0.0000 # writeback rate (i.e., wrbks/ref)
il2.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
ld_text_base              0x00400000 # program text (code) segment base
ld_text_size                 170880 # program text (code) size in bytes
ld_data_base              0x10000000 # program initialized data segment base
ld_data_size                14766944 # program init'ed `.data' and
 uninit'ed `.bss' size in bytes
ld_stack_base             0x7fffc000 # program stack segment base (highest
 address in stack)
ld_stack_size                 16384 # program initial stack size
ld_prog_entry             0x00400140 # program entry point (initial PC)
ld_environ_base           0x7fff8000 # program environment base address
 address
ld_target_big_endian              0 # target executable endian-ness, non-
 zero if big endian
mem.page_count                 1878 # total number of pages allocated
```

# of Sets	Undefied cache Miss rate	Undefied cache AMAT	Split cache		Split cache AMAT
			Inst. Miss rate	Data Miss rate	
64	0.3064	62.32	0.401	0.55	95.21
128	0.2245	45.98	0.204	0.531	73.83
256	0.1698	35.08	0.107	0.524	63.55
512	0.1012	21.4	0.053	0.496	55.38

go와 m88ksim은 Split Cache 구조가 대부분 낮은 AMAT이다. go는 특히 Split Cache 환경에서 측정된 AMAT는 L1 캐시의 set 수가 증가함에 따라 급격하게 감소g ㅏ여 instruction locality가 매우 강함을 알 수 있었다. m88ksim의 데이터 miss는 해석기의 context 구조로 인해 다소 일정하게 유지되었으며, 전체 AMAT는 Split Cache 구성에서 L1 캐시 용량 증가에 따라 꾸준히 감소하였다. swim은 연산 집중적이고 명령어 간 locality 크기는 Split 구조가 더 효과적이었다. 다만 프로그램 특성에 따라 Unified가 더 나은 경우도 존재하며 데이터 miss는 비교적 완만하게 줄어들었으나, instruction miss 감소 효과가 지배적이었기 때문에 전체 AMAT 감소 폭이 다소 컸다. 이는 go와 같은 branch-heavy 구조에서는 Split Cache 구조가 더 효과적인 것을 보여준다.

아래에 추가적으로 분석하였다.

2. L1 / L2 Cache size

```
1 -cache:il1 il1:8:16:1:l
2 -cache:dl1 dl1:8:16:1:l
3
4 -cache:il2 dl2
5 -cache:dl2 ul2:1024:16:1:l
6 -tlb:itlb none
7 -tlb:dtlb none
```

8/8/1024 cache

```
-cache:il1 il1:16:16:1:l
-cache:dl1 dl1:16:16:1:l

-cache:il2 dl2
-cache:dl2 ul2:512:16:1:l
-tlb:itlb none
-tlb:dtlb none
```

16/16/512 cache

```
-cache:il1 il1:32:16:1:l
-cache:dl1 dl1:32:16:1:l

-cache:il2 dl2
-cache:dl2 ul2:256:16:1:l
-tlb:itlb none
-tlb:dtlb none
```

32/32/256cache

```
1 -cache:il1 il1:64:16:1:l
2 -cache:dl1 dl1:64:16:1:l
3
4 -cache:il2 dl2
5 -cache:dl2 ul2:128:16:1:l
6 -tlb:itlb none
7 -tlb:dtlb none
```

64/64/128cache

```
-cache:il1 il1:128:16:1:l
-cache:dl1 dl1:128:16:1:l

-cache:il2 dl2
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

128/128/128cache

2-1. go

L1I/L1D/L2U	Inst.Miss rate	Data.Miss rate	Unified Cache Miss rate	AMAT
8/8/1024	0.4921	0.579	0.284	21.98
16/16/512	0.4492	0.483	0.418	26.87
32/32/256	0.3918	0.387	0.59	30.19
64/64/128	0.337	0.281	0.893	43.26
128/128/0	0.274	0.19	(L2 없음)	28.27

2-2. m88kim

L1I/L1D/L2U	Inst.Miss rate	Data.Miss rate	Unified Cache Miss rate	AMAT
8/8/1024	0.5449	0.397	0.028	8.92

16/16/512	0.4976	0.308	0.193	13.01
32/32/256	0.4283	0.261	0.226	12.95
64/64/128	0.3398	0.093	0.702	16.1
128/128/0	0.2164	0.072	(L2 없음)	18.87

2-3. swim

L1I/L1D/L2U	Inst.Miss rate	Data.Miss rate	Unified Cache Miss rate	AMAT
8/8/1024	0.5912	0.604	0.049	9.81
16/16/512	0.4827	0.557	0.138	13.58
32/32/256	0.3235	0.531	0.298	19.84
64/64/128	0.1401	0.5298	0.893	29.12
128/128/0	0.0904	0.5243	(L2 없음)	19.89

L1과 L2 캐시 용량 변화에 따른 실험 결과를 분석한 결과, 전반적으로 L2 캐시 용량이 클수록 unified 캐시 miss율과 AMAT가 낮아져 시스템 성능이 개선되는 경향이었다. 8/ 8/ 1024은 unified 가장 우수한 성능을 보였다. 128 / 128/ 0와 같이 L1 캐시를 대폭 늘리고 L2 캐시를 제거한 구성은 AMAT가 8/8/1024 대비 두 배 이상 수치며 16 / 16 / 512은 AMAT가 비교적 안정적인 성능을 보인다.

결론적으로, L2를 완전히 제거하면 unified miss rate = 1이 되므로 AMAT가 급증하며 L2 cache는 적당히 큰 값(256~512 sets)일 때 AMAT 절감 효과가 컸다. 너무 큰 L2는 비용/속도 trade-off가 커지므로 적절한 크기인 16 / 16 / 512가 효율적이다.

아래에 추가적으로 분석하였다.

3. Associativity

```
1 -cache:il1 il1:32:16:1:l
2 -cache:dl1 dl1:32:16:1:l
3
4 -cache:il2 none
5 -cache:dl2 none
6 -tlb:itlb none
7 -tlb:dtlb none
```

Split cache 64 Sets(1-way)

```
-cache:il1 il1:32:16:2:l
-cache:dl1 dl1:32:16:2:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 64 Sets(2-way)

```
-cache:il1 il1:32:16:4:l
-cache:dl1 dl1:32:16:4:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 64 Sets(4-way)

```
1 -cache:il1 il1:32:16:8:l
2 -cache:dl1 dl1:32:16:8:l
3
4 -cache:il2 none
5 -cache:dl2 none
6 -tlb:itlb none
7 -tlb:dtlb none
```

Split cache 64 Sets(8-way)

```
-cache:il1 il1:64:16:1:l
-cache:dl1 dl1:64:16:1:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 128 Sets(1-way)

```
1 -cache:il1 il1:64:16:2:l
2 -cache:dl1 dl1:64:16:2:l
3
4 -cache:il2 none
5 -cache:dl2 none
6 -tlb:itlb none
7 -tlb:dtlb none
```

Split cache 128 Sets(2-way)

```
-cache:il1 il1:64:16:4:l
-cache:dl1 dl1:64:16:4:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 128 Sets(4-way)

```
-cache:il1 il1:64:16:8:l
-cache:dl1 dl1:64:16:8:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 128 Sets(8-way)

```
-cache:il1 il1:128:16:1:l
-cache:dl1 dl1:128:16:1:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 256 Sets(1-way)

```
1 -cache:il1 il1:128:16:2:l
2 -cache:dl1 dl1:128:16:2:l
3
4 -cache:il2 none
5 -cache:dl2 none
6 -tlb:itlb none
7 -tlb:dtlb none
```

Split cache 256 Sets(2-way)

```
-cache:il1 il1:128:16:4:l
-cache:dl1 dl1:128:16:4:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 256 Sets(4-way)

```
1 -cache:il1 il1:128:16:8:l
2 -cache:dl1 dl1:128:16:8:l
3
4 -cache:il2 none
5 -cache:dl2 none
6 -tlb:itlb none
7 -tlb:dtlb none
```

Split cache 256 Sets(8-way)

```
-cache:il1 il1:256:16:1:l
-cache:dl1 dl1:256:16:1:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 512 Sets(1-way)

```
1 -cache:il1 il1:256:16:2:l
2 -cache:dl1 dl1:256:16:2:l
3
4 -cache:il2 none
5 -cache:dl2 none
6 -tlb:itlb none
7 -tlb:dtlb none
```

Split cache 512 Sets(2-way)

```
-cache:il1 il1:256:16:4:l
-cache:dl1 dl1:256:16:4:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 512 Sets(4-way)

```
-cache:il1 il1:256:16:8:l
-cache:dl1 dl1:256:16:8:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 512 Sets(8-way)

```
-cache:il1 il1:512:16:1:l
-cache:dl1 dl1:512:16:1:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 1024 Sets(1-way)

```
-cache:il1 il1:512:16:2:l
-cache:dl1 dl1:512:16:2:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 1024 Sets(2-way)

```
-cache:il1 il1:512:16:4:l
-cache:dl1 dl1:512:16:4:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 1024 Sets(4-way)

```
-cache:il1 il1:512:16:8:l
-cache:dl1 dl1:512:16:8:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 1024 Sets(8-way)

```
-cache:il1 il1:1024:16:1:l
-cache:dl1 dl1:1024:16:1:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 2048 Sets(1-way)

```
-cache:il1 il1:1024:16:2:l
-cache:dl1 dl1:1024:16:2:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 2048 Sets(2-way)

```
-cache:il1 il1:1024:16:4:l
-cache:dl1 dl1:1024:16:4:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 2048 Sets(4-way)

```
-cache:il1 il1:1024:16:8:l
-cache:dl1 dl1:1024:16:8:l

-cache:il2 none
-cache:dl2 none
-tlb:itlb none
-tlb:dtlb none
```

Split cache 2048 Sets(8-way)

3-1. go

# of Sets	Split cache Miss rate / AMAT			
	1-way	2-way	4-way	8-way

64	0.4102 0.4011	42.03	0.3453 0.2682	34.18	0.2709 0.1404	26.78	0.1957 0.0651	19.21
128	0.3512 0.3027	35.6	0.2825 0.1607	26.53	0.2172 0.0918	20.1	0.1918 0.0829	15.38
256	0.3512 0.2143	28.71	0.2239 0.114	20.76	0.1687 0.0928	14.29	0.1923 0.0361	10.09
512	0.2287 0.1912	23.25	0.1857 0.0928	15.61	0.1245 0.0214	9.12	0.0613 0.0091	6.87
1024	0.2021 0.0983	18.53	0.1369 0.0361	12.3	0.0739 0.0061	4.21	0.0277 0.0026	2.08
2048	0.1744 0.0417	9.08	0.0932 0.0073	5.18	0.0338 0.0021	2.97	0.0199 0.0005	1.926

3-2. mk88kim

# of Sets	Split cache Miss rate / AMAT							
	1-way		2-way		4-way		8-way	
64	0.4312 0.2829	40.92	0.3491 0.1139	29.17	0.0784 0.0926	5.42	0.0307 0.0693	5.64
128	0.3428 0.1284	31.01	0.2081 0.0927	16.65	0.0402 0.0803	3.09	0.0018 0.0797	2.19
256	0.2285 0.1027	21.48	0.0693 0.0921	6.42	0.0281 0.0716	4.28	0.0016 0.0301	2.51
512	0.0931 0.0916	10.22	0.0414 0.0872	2.48	0.0019 0.0825	3.86	0.0003 0.0801	2.04
1024	0.0523 0.0512	6.48	0.0017 0.0489	2.78	0.0009 0.0441	1.96	0.0002 0.0438	1.77
2048	0.0095 0.0498	1.86	0.0012 0.0474	1.67	0.0003 0.0467	1.54	0.0001 0.0442	1.52

3-3. swim

# of Sets	Split cache Miss rate / AMAT							
	1-way		2-way		4-way		8-way	
64	0.3463 0.5827	47.18	0.1901 0.5022	27.13	0.0989 0.2207	12.08	0.0023 0.0972	7.85
128	0.1951 0.5494	25.26	0.0923 0.4917	18.49	0.0152 0.2211	7.37	0.0012 0.0785	3.12

256	0.1041 0.5296	19.92	0.0145 0.4935	14.53	0.0009 0.2342	7.05	0.0001 0.1002	4.03
512	0.0801 0.497	15.13	0.0012 0.3265	9.61	0.0001 0.0923	4.07	0.0001 0.0706	2.81
1024	0.0284 0.3632	11.15	0.0012 0.0946	4.26	0.0001 0.0678	1.98	0.0000 0.0672	1.96
2048	0.0094 0.0935	2.88	0.0001 0.0709	2.83	0.0000 0.0672	1.95	0.0000 0.0669	1.95

각각의 벤치마크에 따라 associativity를 높이면 miss가 크게 줄어 AMAT가 절감한다. go, m88ksim, swim모두 4-way 이상 구성에서 AMAT가 가장 낮다. 그러나 access time이 2~4% 증가하므로 비용 고려가 필요하다. 아래에 추가적으로 설명하였다.

4. Block Size

4-1. go

Block size	Unified cache Miss rate	AMAT
16	0.2581	44.35
32	0.1426	22.06
64	0.0846	11.37
128	0.0638	6.01
256	0.0275	3.98
512	0.0103	2.62

4-2. m88ksin

Block size	Unified cache Miss rate	AMAT
16	0.0908	19.3
32	0.0102	2.81
64	0.0094	2.12
128	0.0063	1.46
256	0.0043	1.3
512	0.0021	1.19

4-3. swim

Block size	Unified cache Miss rate	AMAT
16	0.0997	21.46
32	0.0491	9.22
64	0.0203	4.97
128	0.0058	2.01
256	0.0036	1.69
512	0.0029	1.56

대부분의 benchmark에서 64B가 가장 효율적이었으며 block이 너무 커지면 불필요한 데이터 fetch로 낭비가 발생한다.

프로그램별 적합한 구성 정리 (CINT95 기준)

프로그램	적합한 Cache 구조	Associativity	Block Size	L2 사용 여부	적합 이유 요약	최적 AMAT
go	Split Cache	8-way	64B	사용 (512 sets)	명령어와 데이터 사용 패턴이 명확히 분리되고 분기 많음	낮음 (3~10)
m88ksim	Split Cache	4~8-way	64B	사용 (256~512)	시뮬레이터 코드 특성상 데이터 접근 복잡, 명령어 재사용 높음	중간
swim	Unified Cache	1~2-way	64B	사용하지 않아도 무방	주로 대규모 배열 계산으로 locality 크고 단순한 흐름	낮음 (특히 데이터 집중형)

AMAT는 다음과 같이 계산되며, miss penalty 및 miss rate를 기반으로 각 설정의 효율성을 정량적으로 비교할 수 있다.

AMAT = L1 hit time + L1 miss rate × (L2 hit time + L2 miss rate × main memory access time)

실험에서는 이와 같은 캐시 구성 요소를 조정하여 시뮬레이션을 진행하였다: Unified cache vs Split cache, L1/L2 캐시 크기 변화 (예: 32/32/256, 64/64/128 등), Block size (16B ~ 512B), Associativity (Direct-mapped ~ 8-way set associative)

go, m88ksim은 Instruction과 Data의 분리 접근이 뚜렷하여 Split cache가 낮은 AMAT를 보였다. swim은 계산 집중형 특성상 Instruction 재사용이 많고 Data locality가 낮아 Unified cache에서도 좋은 성능을 보였다. 모든 프로그램에서 block size는 64B가 성능/비용 균형상 가장 적절하였다. Associativity는 4~8-way일 때 miss rate가 크게 감소하여 AMAT가 최적화되었다.

1. go 최적 구성: Split Cache, 8-way associativity, Block size: 64B, L2 Cache 사용 (512 sets)

복잡한 분기와 탐색구조, 명령어(Instruction) fetch가 분기 중심, 반복 fetch 많음 → I-cache 중요하다. 한편, 게임 상태를 위한 자료구조가 복잡 → D-cache 접근도 빈번함, Instruction/Data 간 동시 fetch가 중요하므로 Split 구조가 적합, Associativity가 높을수록 충돌 감소 → miss rate ↓ → AMAT 최적화라는 분석

2. m88ksim 최적 구성: Split Cache, 4-way 또는 8-way associativity, Block size: 64B, L2 Cache 사용 (256~512 sets)

시뮬레이션 루프 구조로 데이터 접근은 상대적으로 비순차적이며 복잡한 메모리 모델이 존재, Instruction은 일정하지만, Data는 테스트 프로그램마다 동적으로 바뀜, 따라서 Instruction과 Data를 분리하고, Instruction은 locality 활용 / Data는 L2 의존한다.

3. swim 최적 구성: , Unified Cache, 1~2-way associativity, Block size: 64B, L2 Cache 사용 여부: 선택사항

주로 2D 배열 기반의 대규모 데이터 반복 연산, 명령어는 루프 기반으로 일정하며, 대부분 Data 중심의 연산, Instruction/Data 접근 경합 거의 없음→ Unified Cache로 효율 극대화, D-cache locality가 크기 때문에 block size 증가의 효과 큼, associativity가 낮아도 miss rate가 크지 않음 → 비용 효율적라는 분석

따라서 각 벤치마크 프로그램은 작업에 따라 명령어와 데이터의 접근 패턴, 지역성, 경합 여부가 다르기 때문에 캐시 구성 또한 달라져야 한다. go는 분기와 상태 전이가 많기 때문에 instruction/data 경합을 피할 수 있는 Split Cache가 적합하고, 고속 탐색을 위해 8-way associativity가 필요하다. m88ksim은 복잡한 시뮬레이션 코드로 인해 instruction reuse가 높고 data 접근은 무작위적이므로 Split cache 및 충분한 associativity가 필요하다. 반면 swim은 단순한 루프와 배열 접근 중심이므로 Unified Cache와 비교적 단순한 구성에서도 효율적으로 동작한다.

3. 문제점 및 고찰

이번 프로젝트를 통해 캐시의 구조와 크기, associativity, 블록 사이즈, L1/L2 등이 프로그램의 실행 성능에 얼마나 큰 영향을 미치는지 확인할 수 있었다. 특히, 프로그램마다 접근하는 데이터의 locality가 다르기 때문에 캐시 구성 역시 정해진 정답이 아닌, 사용 시나리오에 따라 설계되어야 함을 알게 되었다.

Random Access와 같이 데이터 접근이 무작위적인 프로그램은 D-Cache miss가 매우 빈번하게 발생하여, 블록 사이즈를 크게 하거나 associativity를 높여도 큰 효과를 보기 어렵다는 점을 알게 되었다. 반면, Bubble Sort와 같이 배열 기반의 선형 접근을 하는 프로그램은 relatively 작은 캐시에서도 높은 hit율을 유지할 수 있었다는 것을 알게 됐다.

시뮬레이션을 진행하며 Split Cache와 Unified Cache의 구조적 차이, L2 Cache의 존재 유무, 블록 크기의 trade-off(크면 miss는 줄지만 낭비가 증가) 등을 정량적으로 비교할 수 있었다. 초기에는 시뮬레이터 설정 방법이 다소 어려웠지만, 반복하며 분석한 결과, 이론을 더 깊이 있게 이해할 수 있어 의미 있는 경험이었다.

4. Reference

강의 자료 참고