High-Associativity 캐시 환경에서의 교체 정책 성능 측정과 효율적인 설계 제안

컴퓨터공학과

22311947

김서현

[요 약]

본 프로젝트는 캐시 메모리의 Associativity 변화가 다양한 교체 정책의 성능에 미치는 영향을 정량적으로 분석하고, 이를 통해 하드웨어 복잡도 대비 성능 효율성이 가장 뛰어난 캐시 설계를 제안하는 것을 목표로 한다. 이를 위해 SimpleScalar의 소스코드를 수정하여 시간적 지역성을 역행하는 MRU(Most Recently Used) 정책을 추가로 구현하였으며 mcf와 gcc 벤치마크를 활용해 LRU, Random, MRU 정책 간의 성능 추이를 비교하였다.

당초 본 연구는 High-Associativity 환경에서 LRU와 Random 정책의 미스율 격차를 분석하여 구현 비용이 낮은 Random 정책의 타당성을 입증하고자 하였다. 실험 결과, 16KB의 제한된 용량 환경에서는 용량 미스의 지배적 영향으로 인해 연관도 증가에 따른 성능 수렴 효과가 뚜렷하게 관찰되지 않는 한계가 있었다. 하지만 이러한 악조건 속에서도 Random 정책은 고비용의 LRU 대비 불과 0.2% 내외의 근소한 성능 격차만을 유지하며 안정적인 효율성을 보였다. 이에 본 연구는 절대적인 성능 수치뿐만 아니라 성능과 회로 복잡도 간의 상충 관계(Trade-off)를 종합적으로 고려하여 시스템 목적에 따른 현실적인 최적 설계 기준을 시사한다.

▸키워드: computer architecture, cache memory, associativity, set-associative, replacement policy, LRU, Random, MRU, cache miss, miss rate, locality, hardware complexity, cost-effectiveness, trade-off, SimpleScalar, simulation, benchmark, convergence

I. 서론

컴퓨터 구조의 핵심 목표는 하드웨어 자원의 효율적 활용을 통해 시스템 성능을 극대화하는 것이다. 그중에서도 캐시 메모리(Cache Memory)는 CPU와 메인 메모리 간의 속도 차이를 줄여주는 결정적인 역할을 수행한다. 컴퓨터 시스템 설계에서 '절대적으로 완벽한 설계'란 존재하지 않으며 언제나 성능과 비용 사이의 균형, 즉 Trade-off를 고려해야 한다.

현대 프로세서 설계에서 캐시의 Hit Rate을 높이기 위해 연관도를 증가시키는 기법이 널리 사용된다. 하지만 연관도를 높이는 것은 필연적으로 Comparator와 멀티플렉서(MUX) 등 하드웨어 자원의 증가를 가져오며 이는 회로의 복잡도와 전력 소모를 높이는 원인이 된다. 이러한 관점에서 "높은 연관도를 통해 충돌 미스 자체를 구조적으로 줄일 수 있다면 굳이 복잡한 교체 알고리즘을 사용하여 추가적인 하드웨어 비용을 지불해야 할까?"라는 의문이 제기될 수 있다. 본 연구는 이러한 의문으로부터 출발하여 해당 가설의 타당성을 밝히고자 하였다.

구체적으로는 캐시의 하드웨어 구조와 교체 정책 간의 상관관계를 분석하여 최적의 설계 효율성을 찾고자 한다. 이를 위해 SimpleScalar 시뮬레이터를 활용하며 실험의 다양성을 확보하기 위해 표준적인 LRU, 하드웨어 비용이 가장 낮은 Random, 그리고 시간적 지역성을 역행하는 MRU 정책을 비교군으로 설정하였다. 이때 SimpleScalar에 구현되어 있지 않은 MRU 정책의 경우 소스코드를 직접 수정하여 구현하려 한다. 이 과정이 끝나면 다양한 벤치마크 환경에서 연관도 변화가 각 교체 정책의 성능(Miss Rate)에 미치는 영향을 정량적으로 측정할 수 있을 것이다.

결론적으로 본 연구의 목표는 단순히 시뮬레이션 결과를 나열하는 것을 넘어 High-Associativity 환경에서 교체 정책의 영향력이 어떻게 변화하는지를 규명하는 데 있다. 이를 통해 성능 유지와 하드웨어 비용 절감이라는 두 가지 이점을 잡을 수 있는 공학적으로 타당한 최적의 캐시 설계 기준을 제시하고자 한다.

II. 배경지식 및 관련 기술

본 프로젝트를 진행하기 위해서는 컴퓨터 구조의 핵심 요소인 캐시 메모리의 작동 원리와 시뮬레이션 도구에 대한 사전 지식이 필수적이다. 캐시 메모리는 단순한 저장 공간을 넘어, CPU의 처리 속도와 메모리의 접근 속도 간의 거대한 격차를 메워주는 시스템 성능의 핵심 열쇠이다.

1. 캐시 메모리 아키텍처

(1) 캐시 메모리의 역할과 메모리 계층 구조

현대 컴퓨터 시스템에서 CPU의 연산 속도는 무어의 법칙에 따라 비약적으로 발전해 왔으나 메인 메모리의 접근 속도는 이를 따라가지 못하고 있다. 이로 인해 발생하는 프로세서-메모리 격차는 전체 시스템 성능의 병목 현상을 야기한다.

캐시 메모리는 이러한 문제를 해결하기 위해 CPU와 메인 메모리 사이에 위치하는 작고 빠른 SRAM 기반의 메모리다. 잘 설계된 캐시 메모리 시스템은 CPU가 자주 사용하는 데이터를 미리 보관함으로써 사용자에게 CPU의 속도로 동작하는 메인 메모리 크기의 저장장치를 사용하는 것과 같은 체감을 제공한다. 즉, 비싼 비용의 고속 메모리와 저렴한 비용의 대용량 메모리를 결합하여 최적의 가성비를 창출하는 것이 캐시의 존재 목적이다.

(2) 지역성의 원리

캐시가 효율적으로 동작할 수 있는 이론적 근거는 프로그램의 지역성(Locality)에 있다. 지역성은 크게 두 가지로 나뉜다.

시간적 지역성(Temporal Locality)은 한 번 참조된 데이터는 가까운 미래에 다시 참조될 가능성이 높다는 것이다. 반복문의 변수, 서브루틴 등을 그 예로 들 수 있다.

공간적 지역성(Spatial Locality)은 특정 데이터가 참조되면 그 인접한 주소의 데이터가 참조될 가능성이 높다는 것이다. 배열 접근, 순차적 코드 실행 등을 그 예로 들 수 있다. 캐시 정책과 구조는 이러한 지역성을 최대한 활용하여 Hit Rate를 높이는 방향으로 설계된다.

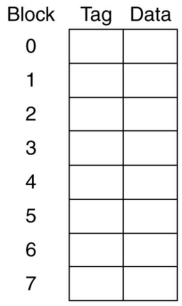
(3) 캐시 매핑 방식

메인 메모리의 데이터를 캐시의 어디에 위치시킬 것인가를 결정하는 매핑 방식은 성능에 결정적인 영향을 미친다.

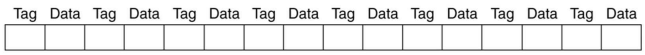
Direct Mapped은 메모리 주소를 특정 캐시 인덱스에 일대일로 대응시킨다. 하드웨어가 가장 간단하고 접근 속도가 빠르지만 서로 다른 주소가 같은 인덱스를 공유할 경우, 빈번한 충돌 미스가 발

생한다. 특히 하나의 자리를 두고 두 개의 블록이 반복적으로 교체되는 핑퐁(Ping-Pong) 현상을 일으킬 수 있다.

Fully Associative는 데이터가 캐시의 어느 위치에나 들어갈 수 있다. 충돌 미스를 최소화할 수 있지만 데이터를 찾기 위해 모든 블록을 동시에 검색해야 하므로 Comparator 등의 하드웨어 비용이 매우 높고 속도가 느려질 수 있다.

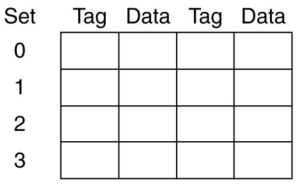


One-Way set associative(direct mapped)

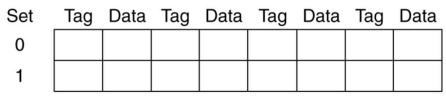


Fully associative

Set-associative는 위 두 방식의 절충안이다. 캐시를 여러 개의 집합(Set)으로 나누고, 각 집합 내에서는 N개의 Way 중 빈 곳에 데이터를 저장할 수 있다. 이는 Direct Mapped의 충돌 문제를 완화하면서도 Fully Associative보다 구현 비용이 저렴하다. 현대 프로세서에서 가장 널리 사용되는 방식이다.



Two-Way set associative



Four-Way set associative

(4) 교체 정책

교체 정책은 캐시가 가득 찼을 때 새로운 데이터를 넣기 위해 누구를 내보낼지 결정하는 알고리즘이다.

LRU(Least Recently Used)는 가장 오랫동안 사용되지 않은 블록을 교체한다. 시간적 지역성에 기반한 가장 표준적인 알고리즘이나 구현 복잡도가 높다는 단점이 존재한다. 메인 메모리의 페이지 교체 정책의 경우 완벽한 LRU를 구현할 수 없어 근사화를 사용한다. 그러나 캐시의 경우 관리할 블록의 수가 적어 LRU를 사용할 수 있다.

FIFO(First-In, First-Out)은 캐시에 들어온 순서대로 교체한다. 현실의 줄 서기와 같은 개념이라고 볼 수 있다. 구현이 간단하다는 장점을 가지고 있지만 자주 쓰이는 데이터가 오래되었다는 이유로 쫓겨날 수 있다는 단점이 존재한다.

Random은 무작위로 교체 대상을 선정하는 것이다. 하드웨어 구현이 매우 간단하며 의외로 High-Associativity 환경에서는 LRU와 대등한 성능을 낼 수 있다고 알려져 있다.

MRU(Most Recently Used)는 가장 최근에 사용된 블록을 교체한다. 일반적인 지역성 원리에는 위배되지만 데이터셋이 캐시 크기보다 큰 순환 루프(Cyclic Reference) 환경에서는 오히려 LRU보다 유리할 수 있다.

2. SimpleScalar 시뮬레이터

본 프로젝트에서 캐시 성능 분석을 위해 SimpleScalar를 사용한다. SimpleScalar는 컴퓨터 아키텍처 연구 및 교육용으로 널리 사용되는 프로그램 기반의 시뮬레이터 툴셋이다. 실제 하드웨어를 제작하지 않고도 프로세서의 구조 변경이나 캐시 파라미터 튜닝에 따른 성능 변화를 소프트웨어적으로 정밀하게 측정할 수 있다.

SimpleScalar 툴셋은 목적에 따라 다양한 시뮬레이터를 제공한다. Sim-cheetah는 다양한 캐시 구성을 한 번에 시뮬레이션하여 최적의 구성을 빠르게 탐색할 때 사용된다. Sim-cache는 본 실험에서 사용하게 될 도구로 캐시 메모리 시스템을 계층적으로 상세히 모델링한다. 사용자는 캐시의 크기, 블록 크기, 연관도, 교체 정책 등을 세밀하게 설정하여 시뮬레이션할 수 있다. Sim-outorder는 비순차 실행 파이프라인까지 포함한 마이크로 아키텍처 레벨의 정밀 시뮬레이터이다.

이러한 도구를 통해 우리는 물리적인 제약 없이 다양한 가설을 설정하고 실행 사이클 및 Miss Rate 데이터를 추출하여 정량적인 분석을 수행할 수 있다.

III. SimpleScalar 분석 및 환경 구축

1. SimpleScalar 설치 및 실행 환경

(1) 구축 환경

본 프로젝트는 Windows 호스트 OS 위의 가상 환경에서 구동되는 Ubuntu를 기반으로 시뮬레이터 실험을 진행하였다. SimpleScalar는 유닉스 계열 시스템에 최적화되어 있어 호환성과 안정성을 위해 리눅스 환경을 채택하였다. 구체적인 환경 명세는 다음과 같다.

- Host OS: Windows 11 (64bit)
- Virtualization Tool: Oracle VM VirtualBox 7.0
- Guest OS: Ubuntu 22.04.2
- Compiler: GCC 3.4
- Simulator: SimpleScalar 3.0 Toolset

(2) 설치 과정

우선 홈 디렉토리에 SimpleScalar를 설치할 'SimpleScalar' 디렉토리를 생성하고 제공된 실습 자료를 해당 위치로 다운로드하였다.

```
user@ubuntu-virtual:~$ cd simplescalar
user@ubuntu-virtual:~/simplescalar$ ls
simplescalar-1.zip  simplesim-3v0e.tgz  simpleutils-2.0d.tar.gz
```

SimpleScalar는 다양한 환경 변수를 필요로 하므로 /home/user/.bashrc 파일에 필요한 환경 변수를 추가하였다. 이후 source .bashrc 명령어를 통해 설정된 환경 변수를 즉시 적용하였다.

```
HOST=i386-pc-linux
IDIR=/home/user/simplescalar
TARGET=sslittle-na-sstrix
```

설치 파일이 위치한 디렉터리($IDIR)로 이동한 후, 다음 명령어를 이용해 SimpleScalar 소스 아카이브를 해제하였다. 압축 해제 후 simplesim-3.0 디렉터리가 생성되는 것을 확인하였다.

```
user@ubuntu-virtual:~/simplescalar$ cd $IDIR
user@ubuntu-virtual:~/simplescalar$ tar xf simplesim-3v0e.tgz
user@ubuntu-virtual:~/simplescalar$ tar xf simpleutils-2.0d.tar.gz
user@ubuntu-virtual:~/simplescalar$ ls
binutils-2.5.2      simplesim-3.0       simpleutils-2.0d.tar.gz
simplescalar-1.zip  simplesim-3v0e.tgz
```

다음으로 binutils-2.5.2 디렉터리로 이동하여 환경 설정 스크립트를 실행하였다.

```
user@ubuntu-virtual:~/simplescalar$ cd binutils-2.5.2
user@ubuntu-virtual:~/simplescalar/binutils-2.5.2$ ./configure --host=$HOST --ta
rget=$TARGET --with-gnu-as --with-gnu-ld --prefix=$IDIR
Created "Makefile" in /home/user/simplescalar/binutils-2.5.2 using "config/mh-li
nux"
user@ubuntu-virtual:~/simplescalar/binutils-2.5.2$ ls
bfd             configure      gas          makeall.bat     README.simplescalar
binutils        configure.bat  gprof        Makefile        texinfo
config          configure.in   include      Makefile.in
config.guess    COPYING        install.sh   move-if-change
config.status   COPYING.LIB    ld           opcodes
config.sub      etc            libiberty    README
```

설정 스크립트 실행이 완료된 후 make 명령어를 통해 컴파일을 시도하였으나 초기 빌드 과정에서 다음과 같은 컴파일 에러가 발생하였다.

```
user@ubuntu-virtual:~/simplescalar/binutils-2.5.2$ make all
make[1]: Entering directory '/home/user/simplescalar/binutils-2.5.2/libiberty'
echo "# !Automatically generated from ./functions.def"\
  "- DO NOT EDIT!" >needed2.awk
grep '^DEFVAR(' < ./functions.def \
 | sed -e '/DEFVAR/s|DEFVAR.\([^,]*\).*|/\1/ { printf "#ifndef NEED_\1\\n#define
 NEED_\1\\n#endif\\n" }|' \
 >>needed2.awk
grep '^DEFFUNC(' < ./functions.def \
 | sed -e '/DEFFUNC/s|DEFFUNC.\([^,]*\).*|/\1/ { printf "#ifndef NEED_\1\\n#defi
ne NEED_\1\\n#endif\\n" }|' \
 >>needed2.awk
gcc -c -g -I. -I./../include  ./dummy.c 2>/dev/null
(gcc -o dummy -g   dummy.o  ) >errors 2>&1 || true
echo "/* !Automatically generated from ./functions.def"\
  "- DO NOT EDIT! */" >lconfig.h
awk -f needed2.awk <errors >>lconfig.h
cp lconfig.h config.h
gcc -c -g -I. -I./../include  argv.c
gcc -c -g -I. -I./../include  basename.c
gcc -c -g -I. -I./../include  concat.c
gcc -c -g -I. -I./../include  cplus-dem.c
```

원인을 분석해 본 결과 SimpleScalar 3.0 버전은 구버전 C 코드로 작성되어 있어 최신 gcc 컴파일러와 호환성 문제가 일으킨다는 것을 확인했다. 이를 해결하기 위해 아래와 같은 코드를 이용해 구버전의 gcc를 설치하였다.

```
sudo apt-get update
sudo apt-get install gcc-3.4 g++-3.4
```

이후 컴파일 과정에서 gcc 버전을 지정해 주니 문제없이 컴파일이 진행되었다.

```
user@ubuntu-virtual:~/simplescalar/binutils-2.5.2$ make CC=gcc-3.4
make[1]: Entering directory '/home/user/simplescalar/binutils-2.5.2/libiberty'
make[2]: Entering directory '/home/user/simplescalar/binutils-2.5.2/libiberty'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/home/user/simplescalar/binutils-2.5.2/libiberty'
make[1]: Leaving directory '/home/user/simplescalar/binutils-2.5.2/libiberty'
```

SimpleScalar는 다양한 타겟 ISA를 지원하지만 실습 자료와 시뮬레이터 예제가 모두 Alpha 아키텍처 기준으로 구성되어 있으므로 make config-alpha 명령을 사용해 Alpha 타겟 환경을 설정한 뒤 빌드를 완료하였다.

```
user@ubuntu-virtual:~/simplescalar/simplesim-3.0$ cd $IDIR/simplesim-3.0
user@ubuntu-virtual:~/simplescalar/simplesim-3.0$ make config-alpha
rm -f config.h machine.h machine.c machine.def loader.c symbol.c syscall.c
ln -s target-alpha/config.h config.h
ln -s target-alpha/alpha.h machine.h
ln -s target-alpha/alpha.c machine.c
ln -s target-alpha/alpha.def machine.def
ln -s target-alpha/loader.c loader.c
ln -s target-alpha/symbol.c symbol.c
ln -s target-alpha/syscall.c syscall.c
rm -f tests
ln -s tests-alpha tests
user@ubuntu-virtual:~/simplescalar/simplesim-3.0$ make
```

설치가 완료된 후 sim-safe 시뮬레이터로 test-math 프로그램을 실행하여 동작 여부를 점검하였다.

```
sim: ** simulation statistics **
sim_num_insn                49382 # total number of instructions executed
sim_num_refs                13624 # total number of loads and stores executed
sim_elapsed_time                1 # total simulation time in seconds
sim_inst_rate           49382.0000 # simulation speed (in insts/sec)
ld_text_base           0x0120000000 # program text (code) segment base
ld_text_size                188416 # program text (code) size in bytes
ld_data_base           0x0140000000 # program initialized data segment base
ld_data_size                41984 # program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base          0x011ff9b000 # program stack segment base (highest address in stack)
ld_stack_size               16384 # program initial stack size
ld_prog_entry          0x012000f750 # program entry point (initial PC)
ld_environ_base        0x011ff97000 # program environment base address address
ld_target_big_endian           0 # target executable endian-ness, non-zero if big endian
mem.page_count                28 # total number of pages allocated
mem.page_mem                 224k # total size of memory pages allocated
mem.ptab_misses               74 # total first level page table misses
mem.ptab_accesses          534266 # total page table accesses
mem.ptab_miss_rate         0.0001 # first level page table miss rate
```

초기화 메시지 및 연산 결과가 정상적으로 출력되었으며 시뮬레이션 통계 또한 오류 없이 생성되었다. 이를 통해 Alpha 기반 SimpleScalar Toolchain이 정상적으로 빌드되었음을 확인하였다.

(3) 실행 방법 및 주요 옵션 설명

본 실험의 핵심 도구인 sim-cache는 CLI 기반 시뮬레이터로 다양한 옵션을 통해 캐시의 구조를 유연하게 변경할 수 있다. 기본적인 실행 명령어의 구조는 다음과 같다.

```
./sim-cache [실행 결과 output 파일] [캐시 구성 옵션] [벤치마크 프로그램 경로] [input 파일 경로]
```

여기서 가장 중요한 부분은 캐시의 구성을 정의하는 -cache 옵션이며 그 형식은 아래와 같다.

옵션 포맷: <name>:<nsets>:<bsize>:<assoc>:<repl>

- name: 캐시의 이름
- nsets: 캐시 Set의 개수
- bsize: Block 하나의 크기, Byte 단위
- assoc: Associativity
- repl: 교체 정책 (l: LRU, f: FIFO, r: Random)

테스트 실행을 위해 gzip 벤치마크에 대해 16KB 용량(L1 D-cache 기준), 32B 블록, Direct Mapped(1-way), LRU 정책을 적용하여 시뮬레이션을 수행하였다. 시뮬레이션 결과는 현재 디렉토리의 cache.txt 파일에 기록되며 총 실행 명령어 수는 1,000,000,000으로 제한하였다.

```
./sim-cache -redir:sim cache.txt -max:inst 1000000000 -cache:dl1
dl1:512:32:1:l ../benchmark/gzip/gzip00.peak.ev6
../benchmark/gzip/input.combined
```

완성된 명령어를 통해 나온 시뮬레이션의 결과는 아래의 통계 데이터와 같다.

```
sim: ** starting functional simulation w/ caches **

sim: ** simulation statistics **
sim_num_insn              1000000000 # total number of instructions executed
sim_num_refs               338203821 # total number of loads and stores executed
sim_elapsed_time                  45 # total simulation time in seconds
sim_inst_rate          22222222.2222 # simulation speed (in insts/sec)
il1.accesses              1000000000 # total number of accesses
il1.hits                   986235581 # total number of hits
il1.misses                  13764419 # total number of misses
il1.replacements            13764165 # total number of replacements
il1.writebacks                     0 # total number of writebacks
il1.invalidations                  0 # total number of invalidations
il1.miss_rate                 0.0138 # miss rate (i.e., misses/ref)
il1.repl_rate                 0.0138 # 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)
dl1.accesses               341480890 # total number of accesses
dl1.hits                   323144834 # total number of hits
dl1.misses                  18336056 # total number of misses
dl1.replacements            18335544 # total number of replacements
dl1.writebacks               8262728 # total number of writebacks
dl1.invalidations                  0 # total number of invalidations
dl1.miss_rate                 0.0537 # miss rate (i.e., misses/ref)
dl1.repl_rate                 0.0537 # replacement rate (i.e., repls/ref)
dl1.wb_rate                   0.0242 # writeback rate (i.e., wrbks/ref)
dl1.inv_rate                  0.0000 # invalidation rate (i.e., invs/ref)
```

본 프로젝트에서는 많은 시뮬레이션 통계 중 접근 횟수, 미스 횟수, 미스율, 교체율을 주요 분석 지표로 활용할 것이다. 이번 테스트 결과에서는 L1 D-cache를 기준으로

- 전체 접근 수: 341,480,890
- 미스 수: 13,716,619
- 미스율: 0.0138
- 교체율: 0.0138

라는 결과가 확인되었다.

2. 소스코드 분석 (Individual Seminar)

본 프로젝트는 개인 과제로 수행됨에 따라 팀 단위의 세미나 과정을 SimpleScalar 핵심 파일에 대한 개인별 심층 분석 및 연구로 대체하였다. 시뮬레이터의 내부 캐시 동작을 이해하고 새로운 교체 정책을 구현하기 위해 cache.h와 cache.c를 중점적으로 분석하였으며 특히 캐시 블록이 메모리 상에서 어떻게 관리되는지, 그리고 미스 발생 시 희생 블록을 어떻게 선정하는지에 집중하여 구현 로직을 파악하였다.

(1) 캐시 블록 및 세트 구조

아래 모든 코드는 cache.h에서 작성된 내용들이다. SimpleScalar는 캐시를 구성하는 각 블록을

구조체로 정의하며, 이들은 이중 연결 리스트 형태로 관리된다. 핵심 구조체는 아래와 같다.

```
// cache.h에 정의된 캐시 블록 구조체
struct cache_blk_t {
    md_addr_t tag;                  // 데이터 블록의 태그 값 (식별자)
    unsigned int status;            // 블록의 상태 (Valid, Dirty 등)
    struct cache_blk_t *way_next;   // 리스트의 다음 블록을 가리키는 포인터
    struct cache_blk_t *way_prev;   // 리스트의 이전 블록을 가리키는 포인터
    // (이하 코드 생략)
};
```

tag는 해당 블록이 어떤 메모리 주소의 데이터를 담고 있는지 식별하는 역할을 한다. way_next와 way_prev는 같은 Set 내의 블록들을 연결하는 포인터로 이 리스트의 순서가 곧 교체 정책(LRU, FIFO 등)의 기준이 된다.

이 블록들을 관리하는 Set 구조체는 다음과 같다.

```
// cache.h에 정의된 캐시 Set 구조체
struct cache_set_t {
    struct cache_blk_t *way_head;   // LRU 리스트의 머리 (가장 최근 사용)
    struct cache_blk_t *way_tail;   // LRU 리스트의 꼬리 (가장 오래전 사용)
    struct cache_blk_t *blks;       // 실제 블록 데이터들의 배열
    struct cache_blk_t **hash;      //High-Associativity에서 검색 비용을 줄이기
                                      위한 해시 테이블
};
```

way_head는 리스트의 가장 앞부분으로, 가장 최근에 참조된 블록을 가리킨다. 반대로 way_tail은 리스트의 가장 뒷부분으로, 가장 오랫동안 참조되지 않은 블록을 가리킨다.

(2) 캐시 접근 및 Hit, Miss 처리 로직

아래 모든 코드들은 cache.c 파일 내부에 존재한다. 캐시에 접근 요청이 들어오면 cache_access 함수가 실행된다. 이 함수는 캐시의 Associativity 수준에 따라 최적화된 탐색 방식을 사용하여 태그 매칭을 수행한다.

```
if (cp->hsize) {
    // High-Associativity 캐시 : 해시 테이블을 통해 고속 탐색
```

```
        int hindex = CACHE_HASH(cp, tag);
        for (blk = cp->sets[set].hash[hindex]; blk; blk = blk->hash_next) {
            if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
                goto cache_hit;          // Hit 발생 시 cache_hit 라벨로 점프
        }
    } else {
        // 일반적인 Associativity 캐시 : 리스트 순차 탐색
        for (blk = cp->sets[set].way_head; blk; blk = blk->way_next) {
            if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
                goto cache_hit;          // Hit 발생 시 cache_hit 라벨로 점프
        }
    }
```

위 코드에서 볼 수 있듯이, 태그가 일치하는 블록을 발견하면 goto cache_hit;을 통해 Hit 처리 구간으로 이동한다. cache_hit 라벨 이후에는 아래과 같이 LRU 리스트를 갱신하는 함수가 호출된다.

```
    // Hit 처리 구간
    cache_hit:
        // (코드 생략)
        // Hit된 블록을 리스트의 맨 앞으로 이동시켜 LRU 상태를 갱신
        update_way_list(&cp->sets[set], blk, HeadDirection);
        return blk;
```

(3) 교체 정책 구현 및 MRU 알고리즘 추가

SimpleScalar의 cache.c 파일 내 cache_access 함수는 미스가 발생했을 때 희생자를 선정하는 로직을 포함하고 있다. 실제 소스코드 분석 결과 switch 문을 통해 정책별로 repl 포인터(교체될 블록)를 결정하는 구조임을 확인하였다.

또한 프로젝트를 위해 기존 LRU, FIFO, Random 정책 아래에 새로운 정책인 MRU를 추가하였다. 각 정책별 상세 동작 및 구현 코드는 다음과 같다.

```
    // cache_access 함수 내부
    switch (cp->policy) {
    // LRU 및 FIFO 정책
    case LRU:
```

```
case FIFO:
    // 리스트의 꼬리(Tail)에 있는 블록을 희생양으로 선택 (가장 오래된 블록)
    repl = cp->sets[set].way_tail;

    // 선택된 블록을 리스트의 머리로 이동시킴
    update_way_list(&cp->sets[set], repl, Head);
    break;

// Random 정책
case Random:
    {
        // 전체 Way 개수 범위 내에서 난수 인덱스 생성
        int bindex = myrand() & (cp->assoc - 1);

        // 2. 해당 인덱스에 위치한 블록을 희생양으로 선택
        repl = CACHE_BINDEX(cp, cp->sets[set].blks, bindex);
    }
    break;

// [추가 코드] 프로젝트를 위한 MRU 구현
  case MRU:
    // 리스트의 머리에 있는 블록을 희생양으로 선택
    repl = cp->sets[set].way_head;
    break;
default:

    panic("bogus replacement policy");
}
```

기존 LRU는 way_tail을 선택한 후 update_way_list()를 호출한다. 이는 꼬리에 있던 블록을 재사용하기 위해 머리로 끌어 올리는 과정을 위한 것이다.

MRU의 구현은 시간적 지역성을 역행하기 위해 가장 최근 블록인 way_head를 바로 교체 대상으로 지목한다. 이때 해당 블록은 이미 리스트의 최상단에 위치해 있으므로 LRU처럼 update_way_list()를 호출하여 위치를 이동시킬 필요가 없다. 이를 통해 불필요한 연산을 줄이는 최적의 MRU를 구현하게 된다.

또한 cache_create 함수 내부에서 호출되는 정책 교환 함수 cache_char2polic()를 수정하여 시뮬레이터가 커맨드 라인 인자로 입력받은 'm' 문자를 유효한 정책으로 인식하고 초기화할 수 있도록 하였다.

```
enum cache_policy cache_char2policy(char c)
{  switch (c) {
   case 'l': return LRU;
   case 'r': return Random;
   case 'f': return FIFO;
   // [추가 코드] 프로젝트를 위한 MRU 구현
   case 'm': return MRU;
   default: fatal("bogus replacement policy, `%c'", c);
   }
}
```

cache_char2policy()의 반환값을 맞춰주기 위해 cache.h 내부의 cache_policy 이넘 클래스에도 MRU의 값을 추가해 준다.

```
enum cache_policy {
   LRU,
   Random,
   FIFO,
   MRU   // [추가 코드] 프로젝트를 위한 MRU 구현
};
```

모든 코드 수정이 끝났다면 수정된 코드를 반영하기 위해 재컴파일을 시도한다. 이후 sim-cache의 교체 정책을 m으로 두고 프로그램을 실행했을 때 문제없이 동작한다는 것을 확인할 수 있다.

```
user@ubuntu-virtual:~/simplescalar/simplesim-3.0$ ./sim-cache -redir:sim cache.t
xt -max:inst 1000000000 -cache:dl1 dl1:512:32:1:m ../benchmark/gzip/gzip00.peak.
ev6 ../benchmark/gzip/input.combined
spec_init
Loading Input Data
Duplicating 3121844 bytes
Duplicating 6243688 bytes
Duplicating 12487376 bytes
Duplicating 24974752 bytes
Duplicating 17159360 bytes
Input data 67108864 bytes in length
Compressing Input Data, level 1
```

MRU은 지역성이 높은 일반적인 프로그램에서는 방금 불러온 데이터를 바로 삭제하게 되어 성능

저하를 유발하지만, 이를 통해 역설적으로 '캐시 성능에서 지역성 유지가 얼마나 중요한지'를 증명하는 대조군 실험 데이터를 얻을 수 있게 한다.

IV. 성능 평가

앞선 장에서 SimpleScalar의 구조를 분석하고 MRU 정책을 추가 구현함으로써 실험을 위한 준비를 마쳤다. 본 장에서는 실제 벤치마크 프로그램을 구동하여 Associativity의 변화가 각 교체 정책(LRU, Random, MRU)의 성능에 미치는 영향을 정량적으로 평가한다. 실험은 하드웨어 복잡도 대비 성능 효율성을 입증하기 위해 통제된 환경에서 수행되었다.

1. 실험 환경

(1) 벤치마크 프로그램 선정

캐시 교체 정책의 성능은 실행되는 프로그램의 메모리 접근 패턴(Memory Access Pattern)에 따라 크게 달라진다. 따라서 본 연구에서는 극단적인 특성을 가진 두 가지 벤치마크 mcf와 gcc를 선정하여 교체 정책의 유효성을 다각도로 검증하고자 한다.

mcf는 대량의 포인터 추적 연산을 수행하는 프로그램으로, 메모리 참조의 지역성이 현저히 낮은 대표적인 Cache Killer 워크로드이다. 빈번한 캐시 미스가 발생하는 환경에서 교체 정책 간의 성능 변별력을 확인하기 위해 필수적인 테스트 프로그램으로 선정하였다.

gcc는 복잡한 분기문과 방대한 코드 크기를 가진 정수 연산 프로그램이다. 이는 일반적인 컴퓨팅 환경을 대변하며, mcf와 같은 특수한 상황 외 범용적인 상황에서도 제안하는 설계가 유효한지를 검증하기 위해 선정되었다.

(2) 시뮬레이션 파라미터 및 변수 통제

정확한 비교 분석을 위해 캐시의 기본 규격은 고정하고 연관도와 교체 정책만을 독립 변수로 설정하여 실험을 설계하였다. 통제 변인의 경우 가능한 현대의 표준 캐시 옵션을 고려하여 설정하였다. 또한 I-Cache 대신 D-Cache를 고른 이유는 순차적 실행 위주인 I-Cache보다 D-Cache의 접근 패턴이 불규칙하여 교체 정책에 따른 성능 변별력을 확인하기에 적합하기 때문이다.

- 통제 변인
    - Target Cache : L1 Data Cache
    - Cache Size : 16KB
    - Block Size : 32 Bytes
- 독립 변인
    - Associativity : 2-way, 4-way, 8-way, 16-way, 32-way
    - Replacement Policy : LRU, Random, MRU

(3) 주요 분석 지표

단순한 미스율 비교를 넘어 시스템 부하와 캐시의 상태를 입체적으로 분석하기 위해 다음 네 가지 지표를 수집하여 분석한다.

- Total Accesses : CPU가 메모리 데이터를 요청한 총횟수
- Misses : 캐시 내에 데이터가 존재하지 않아 상위 메모리 계층으로 접근한 횟수
- Miss Rate : Total Accesses 중 Misses가 발생한 비율
- Replacement Rate : Total Accesses 중 Replacements가 발생한 비율

이때 미스율은 캐시 성능을 나타내는 가장 핵심적인 지표이다. 또한 교체율은 캐시 세트가 가득 차서(full) 기존 블록을 몰아내는 빈도수를 나타낸다. 미스율과 교체율이 유사하다면 Cold Miss를 제외한 대부분의 시간이 캐시가 꽉 찬 상태를 의미하므로 교체 정책의 중요성이 더욱 부각된다.

(4) 수행 계획

Sim-cache 시뮬레이터 구동 시, 연관도 변화에 따라 세트(Set)의 개수를 동적으로 조정하여 전체 캐시 용량을 16KB로 일정하게 유지해야 한다. Set의 개수는 다음 수식에 따라 계산되었다.

$$Sets = \frac{\text{Total Cache Size}(16\text{KB})}{Block\ Size(32B) \times \text{Associativity}}$$

이를 바탕으로 수립한 실험 명령어 구성 테이블은 아래와 같다. 각 설정마다 LRU, Random, MRU 세 가지 정책을 각각 적용하여 총 30회(5 × 3 × 2 benchmarks)의 시뮬레이션을 수행한다.

Associativity	Sets	Configuration Option
2-way	256	dl1:256:32:2:<policy>
4-way	128	dl1:128:32:4:<policy>
8-way	64	dl1:64:32:8:<policy>
16-way	32	dl1:32:32:16:<policy>
32-way	16	dl1:16:32:32:<policy>

또한 원활한 실험 진행을 위해 시뮬레이션 결과는 현재 디렉토리에 텍스트 파일로 기록하였으며 총 실행 명령어 수는 최대 1,000,000,000으로 제한하였다.

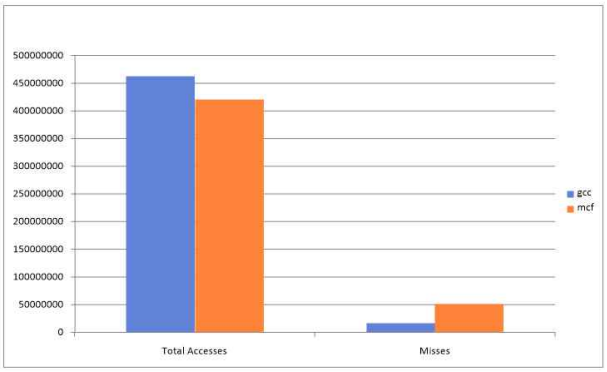
2. 실험 결과 및 분석

본 연구에서는 앞서 설계한 실험 환경을 바탕으로 Associativity의 변화가 각 교체 정책의 효율성에 미치는 영향을 정량적으로 분석하였다. 총 30회의 시뮬레이션을 통해 수집된 로그 데이터를 기반으로 벤치마크별 부하 특성, 교체 정책 간 성능 수렴 현상, 그리고 비용 대비 효율성을 단계별로 기술한다.

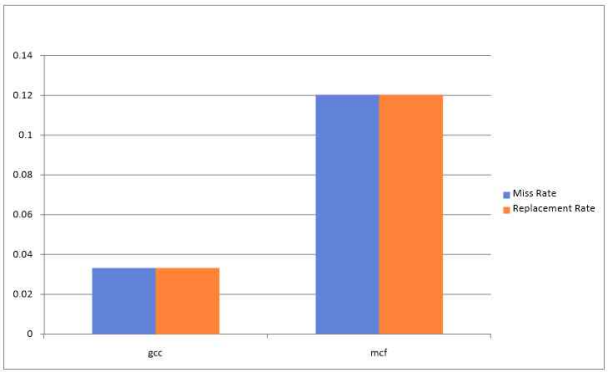
(1) 벤치마크 특성 분석

교체 정책의 성능을 논하기에 앞서, 베이스라인으로 설정된 벤치마크 프로그램들이 캐시 시스템에 어떤 형태의 부하를 주는지 확인하기 위해 기초 통계 데이터를 추출하였다. 아래 데이터는 4-way, 16KB, LRU 환경에서 측정된 결과이다.

벤치마크	Total Accesses	Misses	Miss Rate	Replacement Rate
gcc	462167247	15407324	0.0333	0.0333
mcf	420056233	50457608	0.1201	0.1201



gcc와 mcf에서의 미스 비교



미스율과 교체율 비교

위 표와 그래프에서 확인할 수 있듯이 두 벤치마크의 총접근 횟수는 유사하였으나, mcf가 gcc 대비 약 4배 높은 미스율을 기록하였다. 이는 mcf가 포인터 추적 위주의 연산을 수행하여 공간 지역성의 이점을 거의 받지 못하는 캐시 킬러 워크로드라는 것을 나타낸다.

또한 두 벤치마크 모두 Miss Rate와 Replacement Rate가 소수점 단위까지 거의 일치하는 경향을 보였다. 미스율과 교체율이 거의 수렴하는 형태를 보인 것이다. 이는 초기 Cold Miss 구간을 제외하면 캐시가 실험 내내 가득 찬 상태로 운영되었음을 의미한다.

즉 캐시에서 발생하는 미스의 대부분이 충돌 혹은 용량 부족으로 인해 발생한 것이며 이는 곧 누구를 희생시킬 것인가를 결정하는 교체 정책의 알고리즘 성능이 전체 시스템 효율에 결정적인 영향을 미친다는 것을 증명한다.

(2) Associativity의 변화에 따른 교체 정책 성능 추이

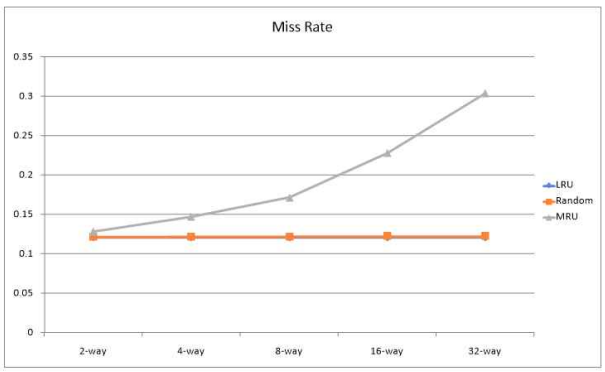
본 프로젝트의 핵심인 Associativity 증가에 따른 LRU, Random, MRU 정책 간의 미스율 변화 추이는 다음과 같다.

① mcf 측정 및 분석

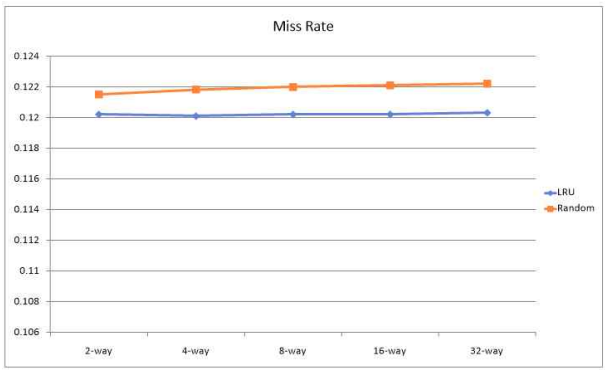
해당 표는 mcf를 통해 측정한 각 교체 정책별 미스율이다.

Assoc	LRU Miss Rate	Random Miss Rate	MRU Miss Rate
2-way	0.1202	0.1215	0.1275
4-way	0.1201	0.1218	0.1463
8-way	0.1202	0.1220	0.1710
16-way	0.1202	0.1221	0.2272
32-way	0.1203	0.1222	0.3032

mcf 벤치마크에서는 기존 예상(Associativity 증가 시 미스율 감소)과 달리, Associativity가 증가해도 LRU와 Random의 미스율이 거의 변화하지 않는 평탄한 양상을 보였다.



정책별 미스율 비교



LRU와 Random의 미스율 비교

이는 16KB라는 L1 캐시 용량이 벤치마크의 워킹 셋을 수용하기에 현저히 부족했기 때문으로 분석된다. 캐시 미스는 크게 Cold 미스, 충돌 미스, 용량 미스로 분류되는데, 이 중 Associativity를 높이는 기법은 주로 충돌 미스를 완화하는데 효과적이다.

그러나 현재 실험 환경은 캐시의 용량이 턱없이 부족하여 대부분의 미스가 용량 미스에 의해 일어나는 상황으로 추측된다. 때문에 Way를 아무리 늘리더라도 캐시가 커버할 수 있는 데이터의 양이 절대적으로 부족하여 성능 향상이 제한된 것이다.

또한 계획에서는 많은 양의 캐시 미스를 일으켜 Associativity와 교체 정책 간의 연관관계를 효과적으로 검증하려 하였으나, mcf의 지역성이 예상보다 더 낮았던 탓에 캐시의 존재가 그리 유의미한 결과를 내지 못했다.

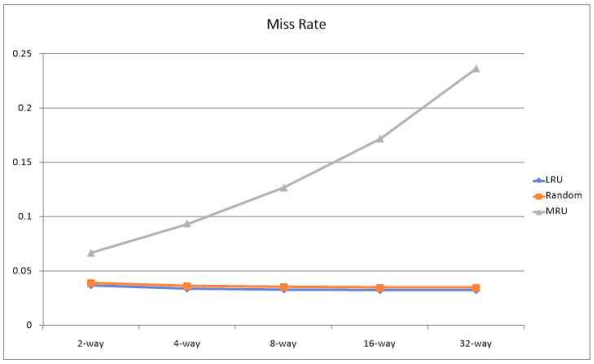
비록 실험 환경의 용량 제약으로 인해 교체 정책 간의 드라마틱한 격차는 관찰되지 않았으나, 현재의 결과를 통해 몇 가지의 아이디어를 도출할 수 있다. 초기의 가설처럼 두 정책의 격차가 0으로 수렴하진 않았으나 32-way 환경에서도 LRU와 Random의 차이는 약 0.19%에 불과했다. 이는 극한의 용량 부족 상황에서도 Random 정책이 유의미한 방어력을 갖는다는 사실을 보여준다.

② gcc 측정 및 분석

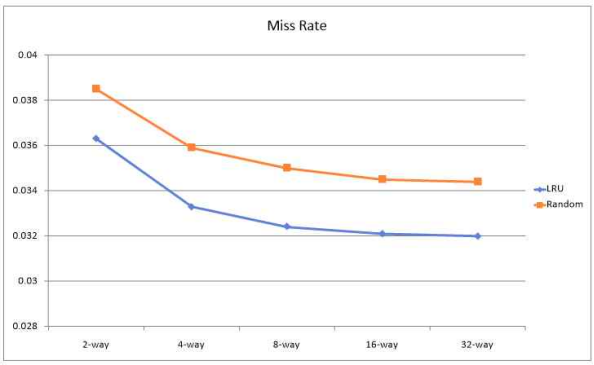
해당 표는 gcc를 통해 측정한 각 교체 정책별 미스율이다.

Assoc	LRU Miss Rate	Random Miss Rate	MRU Miss Rate
2-way	0.0363	0.0385	0.0668
4-way	0.0333	0.0359	0.0932
8-way	0.0324	0.0350	0.1269
16-way	0.0321	0.0345	0.1716
32-way	0.0320	0.0344	0.2366

gcc는 Associativity가 증가함에 따라 LRU와 Random 모두 소폭의 성능의 향상을 보였다. 아래 그래프를 통해 그 점을 더 확실히 확인할 수 있다. 하강 없이 평탄한 모습을 보였던 mcf 그래프와 달리 gcc 그래프에서는 두 정책의 미스율이 꾸준히 하강하고 있는 것을 확인할 수 있다.



정책별 미스율 비교

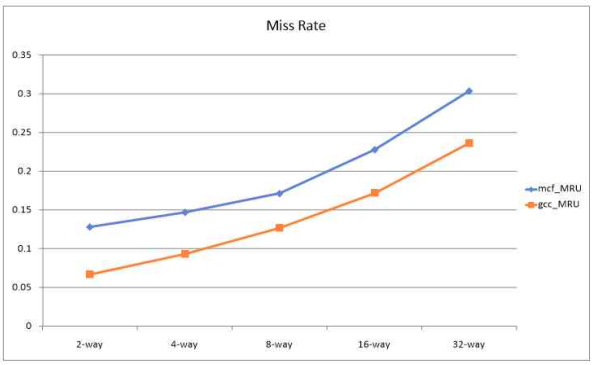


LRU와 Random의 미스율 비교

그러나 여전히 두 정책 간의 성능 격차는 좁혀지지 않고 평행선을 그렸는데, 이 역시 16KB의 용량 한계가 원인으로 판단된다. 다만 두 정책 간의 미스율 격차는 최대 약 0.24% 수준으로 매우 미미하였는데, 이는 고연관도 환경에서 복잡한 LRU 대신 Random을 채택하여도 성능 손실이 크지 않음을 시사한다.

③ MRU 정책의 성능 붕괴

또한 위의 두 실험에서 주목할 만한 결과는 MRU 정책의 성능 추이이다. MRU는 mcf와 gcc 모두에서 Associativity가 높아질수록 미스율이 치솟는 역설적인 성능 붕괴 현상을 일으켰다.



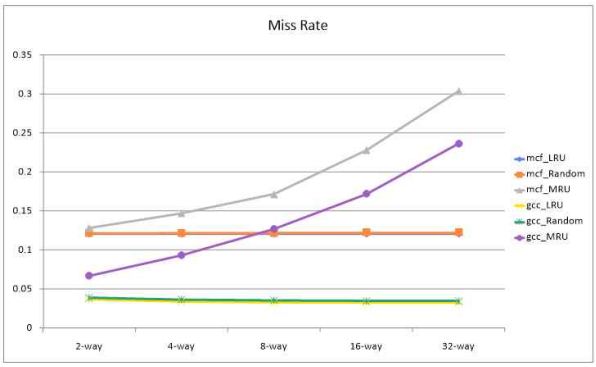
mcf, gcc에서의 MRU 미스율 비교

이는 프로그램 내에 '방금 사용한 데이터는 곧 다시 사용된다'는 시간적 지역성이 강력하게 존재한다는 것을 방증한다. Way가 늘어날수록 MRU 정책 하에서는 접근 빈도가 낮은 오래된 데이터가 캐시 공간을 점유하는 비중이 늘어난다. 32-way에서는 31개의 오래된 데이터가 잔존하는 식이다. 결과적으로 유효 공간이 줄어들어 미스율이 폭등한 것이다.

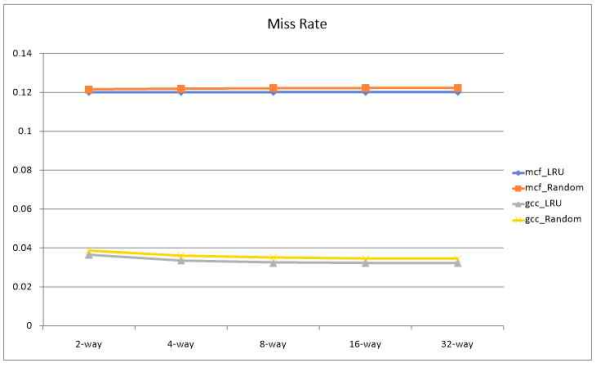
결론적으로 이 실험은 워크로드의 특성을 고려하지 않은 무조건적인 Associativity 증가가 적절치 않은 교체 정책과 결합될 경우, 오히려 독이 될 수 있음을 정량적으로 입증하였다.

V. 결론

본 프로젝트는 High-Associativity 환경에서 캐시 교체 정책이 성능에 미치는 영향을 분석하고, 이를 통해 하드웨어 복잡도 대비 성능 효율성이 가장 뛰어난 캐시 설계를 제안하기 위해 수행되었다. 이를 위해 SimpleScalar 시뮬레이터의 소스코드를 수정하여 MRU 정책을 추가 구현하였으며 mcf와 gcc 벤치마크를 활용하여 LRU, Random, MRU 정책 간의 성능 추이를 비교, 분석하였다.



정책별 미스율 비교



LRU와 Random의 미스율 비교

그러나 실제 실험 결과 당초 가설이었던 '연관도 증가에 따른 LRU와 Random의 성능 수렴' 현상은 뚜렷하게 관찰되지 않았다. 특히 mcf 벤치마크에서는 연관도가 증가함에도 불구하고 미스율이 평행선을 그리거나, MRU의 경우 오히려 성능이 악화되는 경향을 보이기도 했다.

이는 16KB라는 L1 캐시의 제한된 용량으로 인해 캐시의 내에서 충돌 미스보다 용량 미스가 지배적으로 작용했기 때문으로 판단된다. 즉, 교체 정책의 효율성을 검증하기 이전에 절대적인 저장 공간의 부족이 성능의 병목으로 작용한 것이다.

그러나 이러한 극한의 용량 제약 상황에서도 유의미한 공학적 시사점을 도출할 수 있었다. gcc 벤치마크에서는 연관도 증가에 따라 LRU와 Random 모두 꾸준한 성능 향상을 보였으며, 특히 두 정책 간의 미스율 격차는 최대 0.2% 내외에 불과했다. 이는 용량이 부족하여 빈번한 교체가 일어나는 악조건 속에서도 Random 정책이 고비용의 LRU 대비 크게 뒤처지지 않는 방어력을 보여준다는 사실을 입증한다.

본 연구에서는 캐시 크기를 16KB로 고정하였으나, 추후 연구에서는 워크로드의 워킹 셋 크기를 고려하여 캐시 용량을 확장한다면 충돌 미스에 대한 교체 정책의 영향력을 더욱 정밀하게 검증할 수 있을 것이다. 이를 통해 가설로 세웠던 수렴 현상을 보다 명확히 입증할 수 있을 것으로 기대된다.

결론적으로 절대적인 성능 수치에서는 LRU가 미세하게 우세하였으나 그 격차가 0.2% 수준으로 매우 근소하다는 점은 시사하는 바가 크다. 미스율에 상대적으로 관대한 L2, L3 캐시나 전력 효율이 핵심인 임베디드 시스템에서는 성능 저하를 감수하고서라도 $O(N^2)$수준의 회로 복잡도를 가진 LRU 대신 $O(1)$의 Random 정책을 채택하는 것이 합리적인 선택이 될 수 있다.

따라서 본 연구가 제안한 비용 대비 성능을 고려한 Random 정책 도입은 여전히 유효한 공학적 설계 기준임을 확인하였다.