컴퓨터 구조

4차 cache

학 과: 컴퓨터정보공학부

담당교수: 이성원교수님

학 번: 2017202087

성 명: 홍 세 정

1. 실험 내용

이번과제는 cache에 대한 성능을 알아보기 위해 비교를 해볼 수 있다. 리눅스 프로그램을 이용하여 L1, L2과 split, unified cache의 성능 차이를 size에 대해 비교할 수 있고 어떤 종류의 cache가 가장 적합한지 알아볼 수 있다. 또한 benchmark 프로그램에 따른 변화도 알아볼 수 있다.

총 3개 cc1, compress, anagram benchmark 프로그램을 확인할 것이다

Cc1(=GCC complier) : Based on the GNU C compiler version 2.5.3. (inst.access = 119208954, data.access = 44468514)
Compress(=compression) : Compresses large text files (about 16MB) using adaptive Lempel-Ziv coding. (inst.access = 80432491, data.access = 29063189)
Anagram(=anagram) : Performs text and numeric manipulations (anagrams/prime number factoring). (inst.access = 7070, data.access = 4007)

Benchmark에 Sim1~Sim5까지 알아보도록 한다.

instruction cache L1 : #of set : block Size : Associative 순서대로이다. SIM1~SIM5까지 수행하면서 값들을 바꿔주어 결과를 도출해 내어 AMAT를 계산할 수 있다.

2. 검증전략, 분석 및 결과

SIM1) Divide L1 by Instruction / Data or Unified

- Cc1

# of Sets	Unified cache Miss rate	Unified cache AMAT	Split cache		Split cache AMAT
			Inst. Miss rate	data Miss rate	
64	0.3542	11.626	0.3249	0.1707	9.490190811
128	0.2878	9.674	0.2691	0.1106	7.821143603
256	0.235	8.1316	0.2323	0.0746	6.765264014
512	0.1693	6.203864	0.1642	0.0491	5.112739891

- Compress

# of Sets	Unified cache Miss rate	Unified cache AMAT	Split cache		Split cache AMAT
			Inst. Miss rate	data Miss rate	
64	0.3070	10.21	0.1991	0.1247	6.3806
128	0.1536	5.648	0.0598	0.1050	3.1939
256	0.0965	3.9766	0.0392	0.0879	2.6453
512	0.0389	2.291864	0.0004	0.0750	1.7309

- Anagram

# of Sets	Unified cache Miss rate	Unified cache AMAT	Split cache		Split cache AMAT
			Inst. Miss rate	data Miss rate	
64	0.2024	7.072	0.1539	0.2216	6.3517
128	0.1873	6.659	0.146	0.2196	6.2187
256	0.1749	6.329	0.1306	0.2184	5.9524
512	0.1646	6.062864	0.1178	0.2181	5.7473

Unified cache AMAT 보다 split cache AMAT가 더 작아 성능이 더 좋은 것을 확인할 수 있다.

Unified cache AMAT = Average miss time x 30(L1) + Hit time

Split cache AMAT = (inst.miss time x (Access number of Inst. Cache / Access number of cache) + data.miss time x (Access number of Inst. Cache / Access number of cache)) x30 + Hit time

다음과 같은 식으로 AMAT를 구할 수 있었다.

SIM2) how the L1 / L2 size

- Cc1

Inst.L1 data.L2 uni.L2	Inst. Miss rate	data Miss rate	uni Miss rate	AMAT
8 / 8 / 1024	0.4578	0.3974	0.2512	32.4998
16 / 16 / 512	0.4255	0.3037	0.3928	40.1403
32 / 32 / 256	0.3827	0.2346	0.6089	49.9926
64 / 64 / 128	0.3249	0.1707	0.8216	53.5716

- Compress

Inst.L1 data.L2 uni.L2	Inst. Miss rate	data Miss rate	uni Miss rate	AMAT
8 / 8 / 1024	0.4355	0.3685	0.0433	13.4416
16 / 16 / 512	0.4035	0.2221	0.0910	14.9988
32 / 32 / 256	0.3226	0.1594	0.2248	19.5142
64 / 64 / 128	0.1991	0.1247	0.4225	20.0465

- Anagram

Inst.L1 data.L2 uni.L2	Inst. Miss rate	data Miss rate	uni Miss rate	AMAT
8 / 8 / 1024	0.1716	0.2573	0.5360	26.9988
16 / 16 / 512	0.1653	0.2458	0.5989	28.4263
32 / 32 / 256	0.1573	0.2321	0.6576	29.2062
64 / 64 / 128	0.1539	0.2214	0.7184	30.4902

128 /128 / 0 은 값이 모두 0으로 나와 AMAT를 구할 수 없었다.

Uni.L2의 사이즈가 큰 것을 사용하는 것이 더 좋은 성능이라는 것을 확인할 수 있다. Inst.L1과 data.L2의 miss rate는 작이지지만 uni.L2의 miss rate가 더 많이 증가하기 때문에 전체적인 AMAT는 증가하는 것을 확인할 수 있다.

SIM3) Experiment by changing the block size

- Cc1

# of Sets	Split cache Miss rate / AMAT							
	1 - way		2 - way		4 - way		8 - way	
64	0.2830	9.4902	0.2167	7.5217	0.1623	5.9119	0.1060	4.2231
128	0.2260	7.8211	0.1639	5.9772	0.1076	4.3114	0.0492	2.5601
256	0.1894	6.7652	0.1192	4.6808	0.0593	2.9049	0.0272	1.9430
512	0.1329	5.1127	0.0699	3.2435	0.0291	2.0437	0.0142	1.5977
1024	0.0956	4.0392	0.0420	2.4527	0.0161	1.7029	0.0068	1.4226
2048	0.0593	2.9956	0.0215	1.8859	0.0075	1.4920	0.0019	1.3252

- Compress

# of Sets	Split cache Miss rate / AMAT							
	1 - way		2 - way		4 - way		8 - way	
64	0.1793	6.3806	0.0686	3.0792	0.0211	1.6739	0.0180	1.5813
128	0.0717	3.1939	0.0215	1.7048	0.0181	1.6253	0.0147	1.5239
256	0.0521	2.6453	0.0183	1.6513	0.0148	1.5718	0.0119	1.4836
512	0.0202	1.7308	0.0151	1.6012	0.0120	1.5310	0.0098	1.4633
1024	0.0171	1.6845	0.0381	1.3077	0.0099	1.5141	0.0079	1.4528
2048	0.0135	1.6225	0.0022	1.3058	0.0079	1.5030	0.0060	1.4449

- anagram

# of Sets	Split cache Miss rate / AMAT							
	1 - way		2 - way		4 - way		8 - way	
64	0.1719	6.1561	0.1648	5.9638	0.1554	5.7036	0.1370	5.1527
128	0.1655	6.0061	0.1648	6.0058	0.1386	5.2427	0.1310	5.0138
256	0.1540	5.7003	0.1409	5.3324	0.1317	5.0769	0.1310	5.0570
512	0.1445	5.4599	0.1341	5.1704	0.1310	5.1021	0.1310	5.1021
1024	0.1370	5.2809	0.1311	5.1281	0.1310	5.1489	0.1310	5.1489
2048	0.1357	5.2902	0.1311	5.1758	0.1310	5.1976	0.1310	5.1976

inst.L1, data.L1의 miss_rate를 이용하여 miss rate / AMAT의 결과를 도출해낼 수 있었다. # of Sets의 크기가 커질수록 AMAT가 작아지는 것을 확인할 수 있고, way가 커질수록 AMAT가 작아지는 것을 확인할 수 있다. 하지만 8-way에서의 결과처럼 set의 크기가 associative가 어느 일정 부분 커지게 되면, miss_rate가 더 이상 작아지지 않게 된다. 그래

서 set와 associative가 계속 늘어나는 것은 의미가 없어진다.

SIM4) Experiment by changing Associativity

- Cc1

Block size	uni Miss rate	AMAT
16	0.1693	6.079
64	0.0396	2.188
128	0.0203	1.609
256	0.0120	1.36
512	0.0075	1.225

- Compress

Block size	uni Miss rate	AMAT
16	0.0389	2.167
64	0.0126	1.378
128	0.0111	1.333
256	0.0042	1.126
512	0.0032	1.096

- Anagram

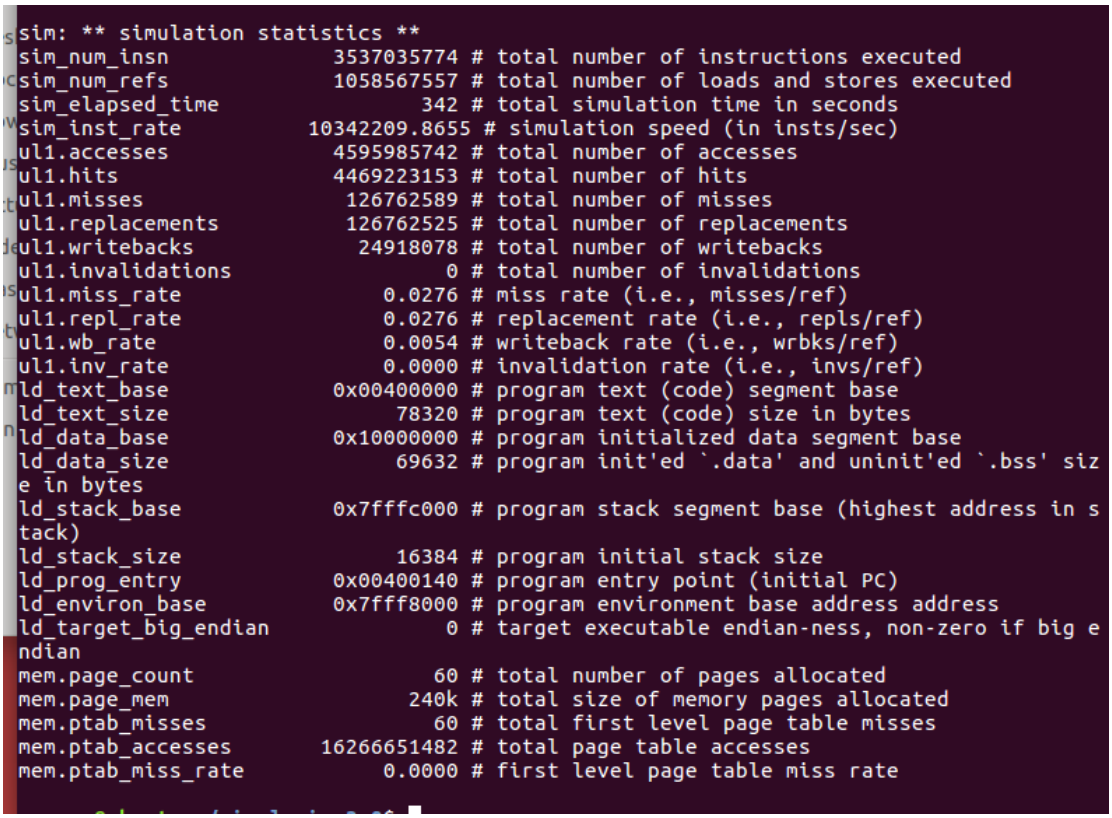
Block size	uni Miss rate	AMAT
16	0.1645	5.935
64	0.0466	2.398
128	0.0241	1.723
256	0.0140	1.42
512	0.0092	1.276

SIM4는 다른 부분은 고정해두고 block size만 변경하게 된다. 블록 사이즈가 커질 때 AMAT가 줄어들게 되어서 성능이 좋다고 할 수 있다. 16AMAT와 64 AMAT는 차이가 많이 나는데 비해 256 AMAT와 512 AMAT는 차이가 많이 나지 않는다. 블록사이즈를 늘릴수록 성능이 좋아지지만 크게 많이 좋아진다고는 할 수 없다.

SIM5)

```
sim: ** simulation statistics **
sim_num_insn             3537035774 # total number of instructions executed
sim_num_refs             1058567557 # total number of loads and stores executed
sim_elapsed_time               183 # total simulation time in seconds
sim_inst_rate          19328064.3388 # simulation speed (in insts/sec)
ul1.accesses             4595985742 # total number of accesses
ul1.hits                 4555950182 # total number of hits
ul1.misses                 40035560 # total number of misses
ul1.replacements           40035048 # total number of replacements
ul1.writebacks             12182189 # total number of writebacks
ul1.invalidations                 0 # total number of invalidations
ul1.miss_rate                0.0087 # miss rate (i.e., misses/ref)
ul1.repl_rate                0.0087 # replacement rate (i.e., repls/ref)
ul1.wb_rate                  0.0027 # writeback rate (i.e., wrbks/ref)
ul1.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
ld_text_base             0x00400000 # program text (code) segment base
ld_text_size                 78320 # program text (code) size in bytes
ld_data_base             0x10000000 # program initialized data segment base
ld_data_size                 69632 # 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                  60 # total number of pages allocated
mem.page_mem                   240k # total size of memory pages allocated
mem.ptab_misses                 60 # total first level page table misses
mem.ptab_accesses       16266651946 # total page table accesses
mem.ptab_miss_rate           0.0000 # first level page table miss rate
```

```
sim: ** simulation statistics **
sim_num_insn             3537035774 # total number of instructions executed
sim_num_refs             1058567557 # total number of loads and stores executed
sim_elapsed_time                147 # total simulation time in seconds
sim_inst_rate          24061467.8503 # simulation speed (in insts/sec)
ul1.accesses             4595985742 # total number of accesses
ul1.hits                 4469223152 # total number of hits
ul1.misses                126762590 # total number of misses
ul1.replacements          126762526 # total number of replacements
ul1.writebacks             24918078 # total number of writebacks
ul1.invalidations                 0 # total number of invalidations
ul1.miss_rate                0.0276 # miss rate (i.e., misses/ref)
ul1.repl_rate                0.0276 # replacement rate (i.e., repls/ref)
ul1.wb_rate                  0.0054 # writeback rate (i.e., wrbks/ref)
ul1.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
ld_text_base             0x00400000 # program text (code) segment base
ld_text_size                 78320 # program text (code) size in bytes
ld_data_base             0x10000000 # program initialized data segment base
ld_data_size                 69632 # program init'ed `.data' and uninit'ed `.bs
s' size in bytes
ld_stack_base            0x7fffc000 # program stack segment base (highest addres
s 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                  60 # total number of pages allocated
mem.page_mem                   240k # total size of memory pages allocated
mem.ptab_misses                 60 # total first level page table misses
mem.ptab_accesses       16266651858 # total page table accesses
mem.ptab_miss_rate           0.0000 # first level page table miss rate
```



```
sim: ** simulation statistics **
sim_num_insn              3537035774 # total number of instructions executed
sim_num_refs              1058567557 # total number of loads and stores executed
sim_elapsed_time                 342 # total simulation time in seconds
sim_inst_rate          10342209.8655 # simulation speed (in insts/sec)
ul1.accesses              4595985742 # total number of accesses
ul1.hits                  4469223153 # total number of hits
ul1.misses                 126762589 # total number of misses
ul1.replacements           126762525 # total number of replacements
ul1.writebacks              24918078 # total number of writebacks
ul1.invalidations                 0 # total number of invalidations
ul1.miss_rate                0.0276 # miss rate (i.e., misses/ref)
ul1.repl_rate                0.0276 # replacement rate (i.e., repls/ref)
ul1.wb_rate                  0.0054 # writeback rate (i.e., wrbks/ref)
ul1.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
ld_text_base              0x00400000 # program text (code) segment base
ld_text_size                  78320 # program text (code) size in bytes
ld_data_base              0x10000000 # program initialized data segment base
ld_data_size                  69632 # program init'ed `.data' and uninit'ed `.bss' siz
e in bytes
ld_stack_base             0x7fffc000 # program stack segment base (highest address in s
tack)
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 e
ndian
mem.page_count                   60 # total number of pages allocated
mem.page_mem                   240k # total size of memory pages allocated
mem.ptab_misses                  60 # total first level page table misses
mem.ptab_accesses       16266651482 # total page table accesses
mem.ptab_miss_rate           0.0000 # first level page table miss rate
```

폴더에 sorting, data.csv을 복사하여 넣고 다음을 진행할 수 있었다.

Uni와 split를 비교하고, block size, set 등을 비교하여 최종적으로 좋은 성능의 cache를 찾을 수 있었다. Bubble sort의 좋은 cache를 찾을 수 있었다.

3. 문제점 및 고찰

저번 프로젝트와는 다르게 리눅스를 이용하여 프로젝트를 진행할 수 있었다. 리눅스를 많이 사용해본 것이 아니라서 리눅스로 어떻게 활용해야 할지도 막막하였다. 리눅스에 복사하는 방법부터 파일을 여는 것부터 문제였다. 그래서 여러 사람한테 물어보고, 찾아봐서 프로젝트를 해결해 나갈 수 있었다. 차근차근 설명에 따라서 하나씩 해볼 수 있었다.

이번 프로젝트에서는 cache의 효율성과 성능을 알아보기 위해 cache의 설정을 변경해주어서 어떤 종류의 cache가 가장 좋은 성능을 가지고 있는지 알아볼 수 있었다. <name>:<nsets>:<bsize>:<assoc>:<repl>을 알맞게 입력해주어서 설정을 변경해줄 수 있다. 결과를 어떻게 확인하는지 몰라 다음을 알맞게 변경하고 확인을 해봐도 값이 똑같이 나오게 되었다. 이 부분에서 가장 많이 시간을 보내게 되었는데, sorting을 하고 나온 값을 보는 것이 아니라 result의 .trace의 값을 확인하는 것이었다. .trace의 값을 확인하고 miss_rate의 값을 확인하여 AMAT의 값을 구하여 cache의 성능을 알아볼 수 있었다.