컴퓨터구조 Report

Project #4 – Cache Design

담당교수: 이성원 교수님

실습분반: 수 7교시

학번: 2019202050

이름: 이강현

[Introduction]

이번 프로젝트는 insertion sorting, random access과 같은 알고리즘 그리고 cc1,ijpeg,perl과 같은 벤치마크에 따른 cache access에 대해 분석하고 비교해보는 것이 목표이다. 먼저 알고리즘에 따라 instruction cache 와 data cache로 이루어져 있는 L1 cache의 access timing에 대해 분석해본다. insertion sorting은 삽입 정렬로 정렬 알고리즘이기 때문에 branch와 loop를 많이 사용한다. random access는 삽입정렬보다는 branch와 loop를 덜 사용한다. 두 프로그램이 수행되면서 L1 cache에서 발생하는 cache miss, cache hit에 대해 분석하고 이와 같은 경우에 어떻게 동작하는지를 확인한다.

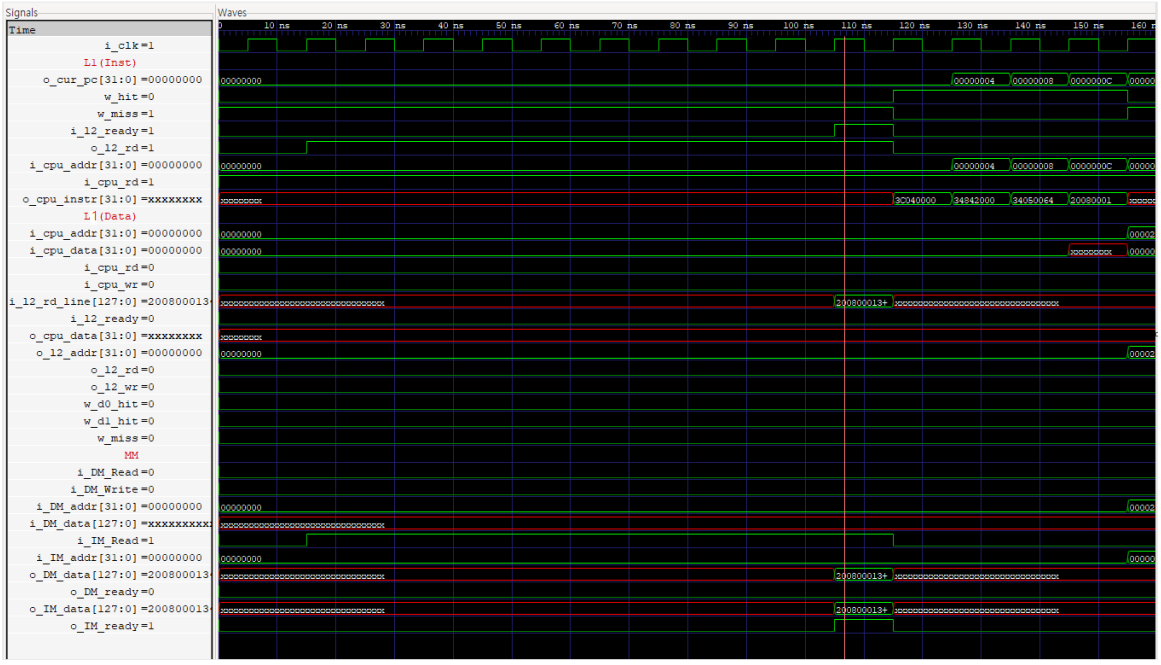
두번째로 3개의 벤치마크를 SimpleScalar라고 불리는 microprocessor로 cache access의 결과를 확인 및 비교하고 가장 적합한 cache가 무엇인지 확인할 수 있다. 벤치마크란 컴퓨터의 성능을 테스트할 수 있는 하나의 척도이고 특정 작업을 지시하고 이를 실행해보면서 성능을 측정할 수 있는 것을 말한다. 이번 과제에서는 SPEC CINT95 program 3가지(cc1,ijpeg,perl)를 사용하여 캐시 성능을 측정해 본다.

[Assignment]

- **Observation for Program Behaviors**

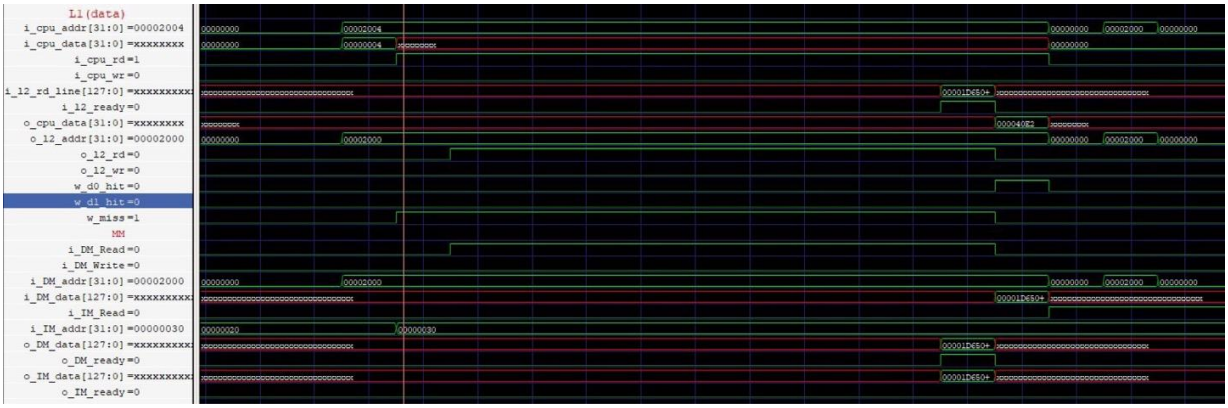
  먼저 insertion sorting, random access 프로그램을 실행시켜보며 캐시의 동작 방법과 타이밍에 대해 알아본다.

  <insertion sorting> - L1 instruction cache hit/miss -

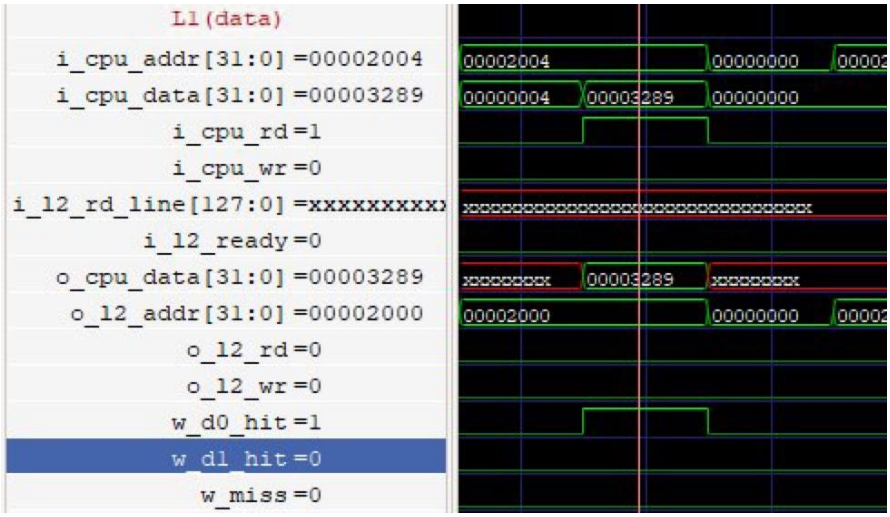


먼저 프로그램 시작 후 바로 나오는 instruction L1 cache의 miss 상황이다. 명령어가 캐시내에 없기 때문에 miss가 나타나고 필요한 데이터를 o_l2_rd을 1로 만듦으로써 L2에서 읽겠다는 것을 나타내고 이러한 신호를 받은 main memory는 i_IM_Read가 1이 되어 instruction memory가 현재 데이터를 필요로 한다라는 것을 알게 되어 데이터를 준비한다. 데이터 access가 완료되면 o_IM_ready를 1로 바꾸어 데이터가 준비되었음을 알리고 i_IM_Read는 다시 0으로 바뀌고 instruction L1 cache는 데이터가 도착했음을 i_l2_ready신호로 알게 되고 다음 클록에서 데이터를 정상적으로 얻었기에 hit을 나타낸다. 이때 데이터는 블록 단위로 가져오기 때문에 4씩 증가하며 명령어를 실행하는 명령어의 특성상 instruction cache는 0x0000,0x0004,0x0008,0x000C까지 hit을 나타낼 수 있다. 이는 spatial locality를 보여주는 부분이다. 따라서 위처럼 miss가 나타나면 메인메모리에게 요청하여 데이터를 블록 단위로 받아오게 되고 hit이 나타나면 바로 캐시에서 데이터를 쓰는 것을 알 수 있다.

- L1 data cache hit/miss -

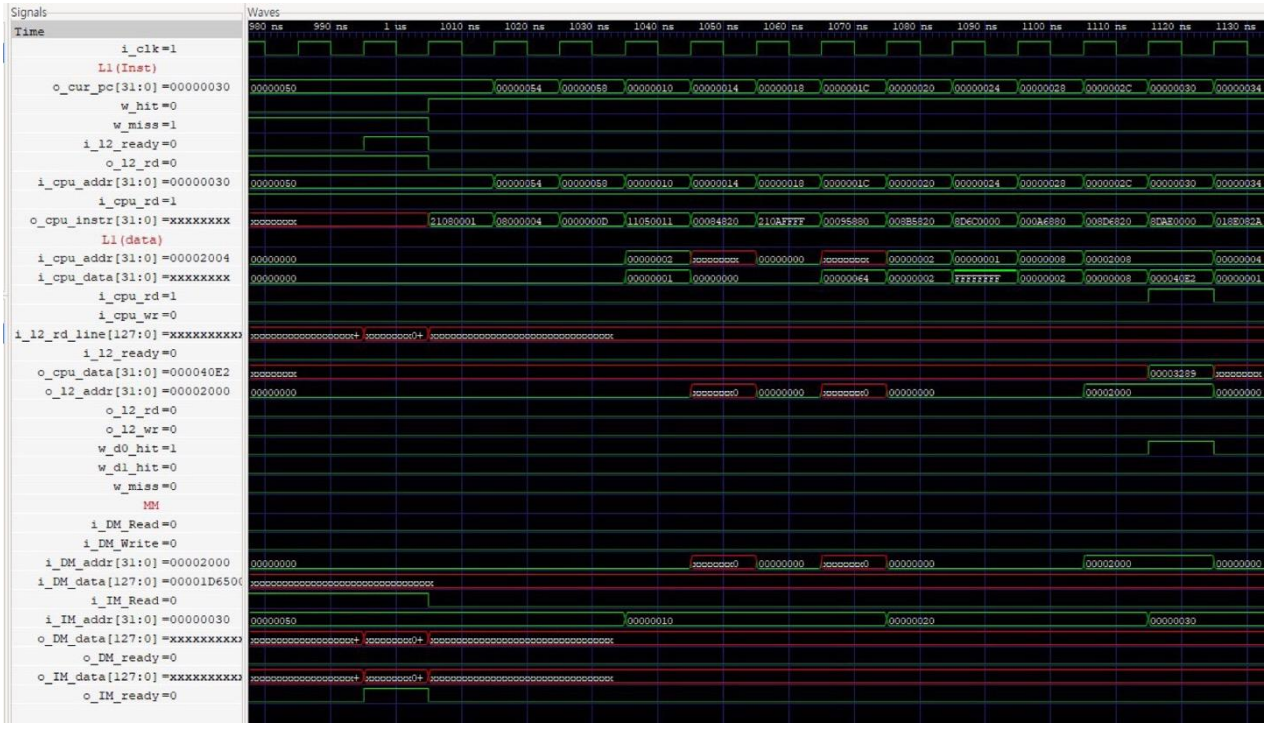


이번엔 L1 data cache의 miss를 보여주는 부분이다. i_cpu_rd가 1로 cpu가 읽으려고 했으나 데이터가 없어 w_miss가 1이되었고 따라서 L1 data cache는 메인 메모리에게 o_l2_rd를 1로 만듦으로써 데이터를 요청하게 된다. 이를 메인 메모리에서 i_DM_Read가 1이 되어서 알게되고 데이터를 준비한 후 o_DM_ready를 1로 만들어 준비된 데이터를 L1 data cache에게 보내준다. L1 data cache는 2-way associative 방식이므로 d0혹은 d1에 저장되고 위의 경우에는 d0에 저장되어 w_d0_hit이 1이된 것을 확인할 수 있다.



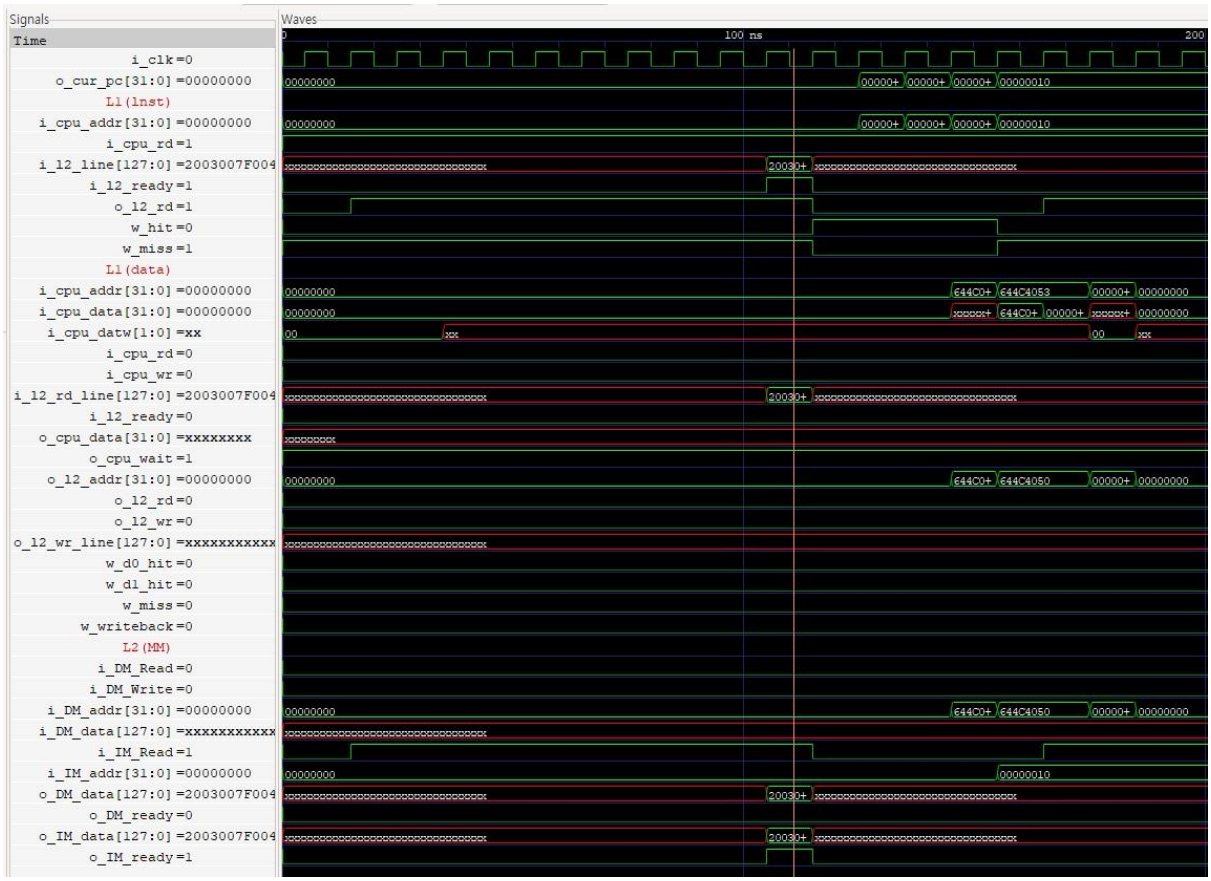
hit이 나는 경우는 위와 같이 아래 계층에 데이터를 요청할 필요없이 바로 hit이 나타나 데이터를 바로 캐시에서 사용할 수 있다.

추가적으로 insertion sorting방식은 random access와 다르게 branch와 loop문을 자주 돌기 때문에 아래와 같이 한번 접근했던 데이터를 다시 사용하게 된다. 따라서 hit의 연속적인 발생이 일어난다.



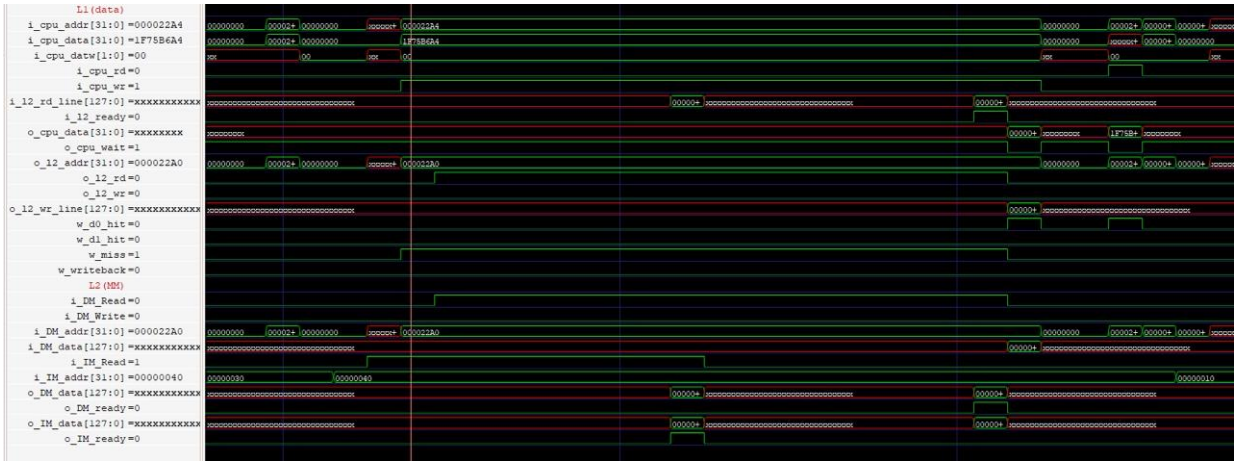
루프를 돌아 0x10에 해당하는 명령어를 다시 반복 실행하게 될 때 이미 캐시에 가져왔던 명령어기 때문에 w_hit이 바로 1이되는 것을 알 수 있다. 따라서 temporal locality를 잘 보여주는 예이다.

<random access> - L1 instruction cache hit/miss -



random access 또한 insertion sorting 방식과 같이 첫 명령어를 실행할 때 miss 가 나기 때문에 위와 같이 메인 메모리에게 데이터를 요청하여 받게 된다. 동일하게 데이터는 블록단위로 가져오기 때문에 한번 hit이 난 후 4씩 증가하며 실행하는 명령어 특성상 hit이 나타나는 것을 확인할 수 있다.

- L1 data cache hit/miss -



data cache의 miss의 경우는 i_cpu_wr가 1이 되었고 w_miss가 1이 되어 cache miss가 났음을 알 수 있다. 따라서 o_l2_rd가 1이되어 메인메모리를 읽게되고 메인 메모리는 L1 data cache에서 데이터를 필요로 한다는 것을 i_DM_Read가 1이 되는 것으로 알게 되어 데이터를 준비한 후 o_DM_ready를 1로 만들어 데이터가 준비되었음을 알린다. 그 후 데이터를 받은 L1 data cache는 d0와 d1중 d0에 저장하게 되고 d0에서 hit이 나게 된다.

데이터를 받고 hit이 난 후 다다음 클록에서 동일한 데이터에 대한 접근이 진행될때는 이미 캐시에 데이터를 가져다 놓은 후이므로 바로 hit이 나는 것을 확인할 수 있다.

그럼 cache의 data size를 증가 시킨다고 가정하면 어떤 양상을 띌까?

그렇다면 캐시가 miss가 났을 때 더 많은 데이터를 저장해둘 수 있기에 miss rate가 줄어들 가능성이 있다. 하지만 캐시의 크기가 너무 커진다면 오히려 access 시간이 늘 수 있고 이는 오히려 역효과를 낼 수 있다.

- **Cache Simulation**

먼저 각 벤치마크에 대한 추가 설명을 진행한다.

cc1은 컴파일러의 성능을 측정하는데 중점을 두고 특히 C코드 컴파일에 중점을 둔다. 따라서 컴파일러의 속도, 효율성등을 평가하기 위해 사용한다. 따라서 명령어를 얼마나 빠르게 가져오느냐와 같은 것에 영향을 받기에 instruction cache의 성능을 테스트하는데 좋은 벤치마크가 될 것으로 예상한다.

ijpeg는 이미지 압축 알고리즘으로 jpeg 파일의 인코딩 디코딩의 성능을 평가하는데 사용된다. 따라서 jpeg로 이미지를 압축해보고 원래 상태로 압축을 푸는데 걸리는 시간을 측정한다. 따라서 픽셀 값, 매트릭스와 관련된 데이터를 읽는 작업을 많이 하게되고 이는 data cache를 얼마나 많이 빠르게 읽느냐가 성능에 영향을 줄 가능성이 있다.

perl은 perl 프로그래밍 언어 해석기의 성능 측정에 중점을 둔다. perl 인터프리터의 실행속도와 효율성을 평가하는데 사용된다. perl 스크립트에는 데이터 구조를 조작하는 명령 실행이 포함되어 있기에 메모리에서 명령어를 가져와 메모리에

저장된 데이터를 조작해야 한다. 따라서 instruction cache와 data cache 모두에 영향을 받을 것이다.

AMAT 계산은 제안서에서 제시한 각 캐시 access time과 block size에 따라 다른 cycle time증가 비율, associativity에 따라 다른 cycle time 증가비율을 고려하여 계산하였다.

unified cache AMAT은 (hit time) + (miss rate) X (miss penalty) 식으로 구하였고 이때 miss penalty는 L1 cache만 존재한다고 가정하므로 main memory access time = 200cycle로 하였다. split cache AMAT은 %instr X (instr hit time + instr miss rate X instr miss penalty) + %data X (data hit time + data miss rate X data miss penalty) 식을 이용하여 각 캐시에 접근횟수를 비율로 나누어 계산하였다.

<cc1>

먼저 cc1 벤치마크의 경우이다.

- **Unified vs splits(block size = 16, Associativity = 1로 고정)**

# of Sets	Unified cache Miss rate	Unified cache AMAT	Split cache		Split cache AMAT
			Inst. Miss rate	Data Miss rate	
64	0.3411	69.5015	0.3697	0.2248	66.7822
128	0.2745	56.2328	0.3116	0.1654	55.1577
256	0.2203	45.4461	0.2558	0.1154	44.3371
512	0.1667	34.7815S	0.2131	0.0793	36.2120

split cache와 unified cache로 각각 나누었을 때 AMAT을 이용해 성능을 분석해보면 miss rate는 캐시 사이즈가 커지면 커질수록 낮아지는 것을 확인할 수 있다. 가장 좋은 성능을 낸 경우는 set을 512로 한 Unified cache이다.

- **L1/L2 size(Block size = 16, Associativity = 1로 고정)**

L1I/L1D/L2U	Inst.Miss rate	Data.Miss rate	Unified Cache Miss rate	AMAT
8/8/1024	0.4827	0.3761	0.2450	32.2358
16/16/512	0.4294	0.2954	0.3910	39.0616
32/32/256	0.3697	0.2248	0.5967	46.4157
64/64/128	0.3116	0.1654	0.8267	50.6750
128/128/0	0.2558	0.1154		44.4254

none으로 하고
miss rate 로 고정함.

L1을 instruction cache와 data cache로 나누고 L2를 unified cache로 구성했을 때 cache size를 다르게 하여 테스트해보면 L1의 miss rate의 경우 점점 줄어들지만 Unified cache의 miss rate의 증가 폭이 더 커 결과적으론 AMAT이 높아지는 결과를 낳는다. 따라서 가장 좋은 결과를 낸 것은 8/8/1024이다.

- **Associativity(Block size = 16으로 고정)**

# of Sets	Split Cache Miss rate / AMAT									
	inst/data 1-way		AMAT	inst	data 2-way	AMAT	4-way		AMAT	8-way	AMAT
기준 64	0.3697	0.2248	66.7822	0.2965	0.1320	31.0587	0.2403	0.0178	39.9517	0.1926	0.0456	31.3051
128	0.3118	0.1654	55.1577	0.2456	0.0881	41.3186	0.1949	0.0483	31.8485	0.1378	0.0298	22.5833
256	0.2558	0.1154	44.2271	0.1986	0.0564	32.8174	0.1408	0.0313	23.1205	0.0858	0.0192	14.5582
512	0.2131	0.0793	36.2120	0.1499	0.0362	24.7261	0.0901	0.0200	15.2437	0.0383	0.0122	9.3843
1024	0.1617	0.0524	29.5563	0.0990	0.0228	16.7032	0.0440	0.0126	8.2443	0.0145	0.0062	3.6742
2048	0.1172	0.0338	19.9613	0.0563	0.0140	10.1195	0.0182	0.0064	4.2415	0.0082	0.0021	2.5277

이번엔 cache size와 Associativity를 다르게 구성하며 테스트해보았다. cache size가 증가할수록 Associativity가 증가할수록 더 좋은 성능을 냈다. 가장 좋은 결과를 보여준 건 Associativity = 8, set = 2048이다.

- **Block size(Number of Sets = 512, Associativity = 1으로 고정)**

hit time + miss rate X miss penalty (200)

Block size	Unified cache Miss rate	AMAT
기준 16	64804288 / 388175301 = 0.1667	34.34
64	15952184 / 388175301 = 0.0410	9.2816
128	8018884 / 388175301 = 0.0206	5.2449
256	4578941 / 388175301 = 0.0118	3.5299
512	2503457 / 388175301 = 0.0064	2.4967

이번엔 블록사이즈를 다르게 해보며 테스트해보았다. 블록 사이즈가 클수록 miss rate는 줄었고 성능도 향상되었다. 가장 좋은 경우는 block size=512이다.

결과적으로 cc1 벤치마크에는 가장 적합한 캐시 구성이 block size를 바꿔가며 진행한 시뮬레이션에서 나왔고 캐시 구성은 Block size를 512, Number of Sets = 512,

Associativity = 1인 Unified Cache가 가장 좋은 성능을 보여주었다.

<ijpeg>

ijpeg 벤치마크의 경우이다.

- **Unified vs splits(block size = 16, Associativity = 1로 고정)**

# of Sets	Unified cache Miss rate	Unified cache AMAT	Split cache		Split cache AMAT
			Inst. Miss rate	Data Miss rate	
64	7029845 / 37389281 = 0.1880	38.8179	7560571 / 29242613 = 0.2585	2094612 / 8146668 = 0.2571	52.6390
128	3759613 / 37389281 = 0.1006	21.3866	3184598 / 29242613 = 0.1089	1787696 / 8146668 = 0.2194	27.6353
256	1709398 / 37389281 = 0.0457	10.4503	618645 / 29242613 = 0.0212	1208584 / 8146668 = 0.1484	10.8647
512	1203872 / 37389281 = 0.0322	7.8100	137231 / 29242613 = 0.0047	644333 / 8146668 = 0.0791	5.3070

miss rate는 cache size가 커질수록 감소했고 가장 좋은 성능을 낸 경우는 set을 512로 한 Split cache이다.

- **L1/L2 size(Block size = 16, Associativity = 1로 고정)**

L1I/L1D/L2U	Inst.Miss rate	Data.Miss rate	Unified Cache Miss rate	AMAT
8/8/1024	14362746 / 29242613 = 0.4912	3418580 / 8146668 = 0.4193	680329 / 18700211 = 0.0810	13.4590
16/16/512	10916907 / 29242613 = 0.3733	2880633 / 8146668 = 0.3511	1046953 / 14851867 = 0.0729	13.4980
32/32/256	7560571 / 29242613 = 0.2585	2094612 / 8146668 = 0.2571	1205461 / 10117098 = 0.1261	12.3676
64/64/128	3184598 / 2924 2613 = 0.1089	1787696 / 8146668 = 0.2194	2165063 / 5308509 = 0.4078	14.3347
128/128/0	618645 / 29242613 = 0.0212	1208584 / 8146668 = 0.1484		10.9529

L1을 instruction cache와 data cache로 나누고 L2를 unified cache로 구성했을 때 cache size를 다르게 하여 테스트해보면 L1의 miss rate의 경우 점점 줄어든다. cc1 벤치마킹때와는 다르게 L1 cache에 해당하는 부분의 miss rate가 줄어드는 폭이 압도적이라 L2를 아예 사용하지 않는 경우가 계층적으로 나누는 것보다 더 좋은 성능을 냈다. 이는 ijpeg 벤치마크가 반복 작업을 많이하여 L1에서의 hit rate가 매우 높기 때문에 이런 결과가 나온 것으로 예상한다.

- **Associativity(Block size = 16으로 고정)**

# of Sets	Split Cache Miss rate / AMAT											
	1-way		AMAT	2-way		AMAT	4-way		AMAT	8-way		AMAT
기준 64	0.2585	0.2511	53.6308	0.1102	0.1812	29.6218	0.0173	0.0404	5.9814	0.0050	0.0251	3.0542
128	0.1089	0.2194	28.2790	0.0170	0.1587	11.2142	0.0035	0.0857	3.0740	0.0012	0.0154	2.0452
256	0.0212	0.1484	11.6057	0.0054	0.0711	5.4290	0.0014	0.0160	2.1266	0.0004	0.0113	1.7663
512	0.0047	0.0791	5.7405	0.0015	0.0517	3.2114	0.0004	0.0114	1.7937	0.0003	0.0090	1.6835
1024	0.0035	0.0518	4.2560	0.0010	0.0126	1.9663	0.0003	0.0092	1.7168	0.0003	0.0084	1.7016
2048	0.0025	0.0157	2.3688	0.0003	0.0099	1.7654	0.0003	0.0085	1.7309	0.0003	0.0081	1.7365

cache size와 Associativity를 다르게 구성하며 테스트해보았다. cache size가 증가할수록 Associativity가 증가할수록 더 좋은 성능을 냈다. 하지만 어느 수준 이상으로 증가하였을 때 오히려 역효과가 나는 경우가 생겼다. 이는 associativity나 캐시 사이즈를 증가했을 때 생기는 miss rate의 감소 효과보다 증가하는 memory access cycle time이 더 크다는 것을 알 수 있다. 따라서 가장 좋은 결과를 보여준 건 Associativity = 8, set = 512이다.

- **Block size(Number of Sets = 512, Associativity = 1으로 고정)**

Block size	Unified cache Miss rate	AMAT
기준 16	120382 / 3738928 = 0.0322	7.4400
64	34870 / 3738928 = 0.0093	2.9416
128	5219 / 3738928 = 0.0014	1.4049
256	3007 / 3738928 = 0.0008	1.3299
512	2079 / 3738928 = 0.0006	1.3317

블록사이즈를 다르게 해보며 테스트해보았다. 블록 사이즈가 클수록 miss rate는 줄었고 성능도 향상되었다. 블록 사이즈도 256을 넘어가게 되면 miss rate의 감소보다 access time의 증가폭이 더 커 역효과를 낸다. 따라서 가장 좋은 경우는 block size=256의 경우이다.

결과적으로 ijpeg 벤치마크에 제일 적합한 캐시 구성은 block size를 바꿔가며 진행한 시뮬레이션에서 나왔고 캐시 구성은 Block size를 256, number of Sets = 512, Associativity = 1인 Unified Cache가 가장 좋은 성능을 보여주었다.

<perl>

먼저 perl 벤치마크의 경우이다.

- **Unified vs splits(block size = 16, Associativity = 1로 고정)**

# of Sets	Unified cache Miss rate	Unified cache AMAT	Split cache		Split cache AMAT
			Inst. Miss rate	Data Miss rate	
64	167548 / 451517 = 0.3711	75.4951	132588 / 327284 = 0.4051	29011 / 124233 = 0.2392	72.8907
128	131070 / 451517 = 0.2903	59.3862	115217 / 327284 = 0.3520	22455 / 124233 = 0.1807	62.0135
256	103964 / 451517 = 0.2303	47.4392	91390 / 327284 = 0.2792	17246 / 124233 = 0.1388	49.1955
512	81960 / 451517 = 0.1815	37.7344	69811 / 327284 = 0.2133	13435 / 124233 = 0.1081	39.9958

기준

miss rate는 cache size가 커질수록 감소했고 가장 좋은 성능을 낸 경우는 set을 512로 한 Unified cache이다.

- **L1/L2 size(Block size = 16, Associativity = 1로 고정)**

L1I/L1D/L2U	Inst.Miss rate	Data.Miss rate	Unified Cache Miss rate	AMAT
8/8/1024	149859 / 327284 = 0.4579	46565 / 124233 = 0.3748	67385 / 219372 = 0.3072	36.4293
16/16/512	142095 / 327284 = 0.4342	36813 / 124233 = 0.2963	82012 / 199676 = 0.4149	41.5418
32/32/256	132588 / 327284 = 0.4051	29011 / 124233 = 0.2342	102215 / 199493 = 0.5159	49.1301
64/64/128	115217 / 327284 = 0.3520	22455 / 124233 = 0.1807	121090 / 149829 = 0.8108	55.9827
128/128/0	91390 / 327284 = 0.2792	17246 / 124233 = 0.1388		49.2638

기준

L1을 instruction cache와 data cache로 나누고 L2를 unified cache로 구성했을 때 cache size를 다르게 하여 테스트해보면 L1의 miss rate의 경우 점점 줄어든다. L1 cache에서 miss rate의 감소폭보다 L2 cache에서의 miss rate 증가폭이 더 커서 성능이 오히려 감소하는 것을 볼 수 있다. 따라서 가장 좋은 경우는 8/8/1024의 경우이다.

- **Associativity(Block size = 16으로 고정)**

# of Sets	Split Cache Miss rate / AMAT										
		1-way	AMAT		2-way	AMAT		4-way	AMAT	8-way	AMAT
기준 64	0.4051	0.2391	94.1514	0.3520	0.1507	61.8781	0.2176	0.1039	46.8008	0.1978	0.0693	34.5201
128	0.3520	0.1807	63.3202	0.2806	0.1108	49.1473	0.1998	0.0728	35.0221	0.1230	0.0461	22.3617
256	0.2792	0.1388	50.3665	0.2028	0.0821	35.9419	0.1297	0.0514	23.3558	0.0670	0.0360	12.0084
512	0.2133	0.1081	38.7982	0.1404	0.0594	25.3209	0.0728	0.0397	14.0618	0.0342	0.0201	7.8394
1024	0.1684	0.0917	30.4412	0.0871	0.0400	16.1807	0.0372	0.0204	8.3548	0.0214	0.0128	5.0122
2048	0.1418	0.0590	25.1518	0.0615	0.0353	12.2043	0.0240	0.0290	6.5028	0.0184	0.0288	5.4641

이번엔 cache size와 Associativity를 다르게 구성하며 테스트해보았다. cache size가 증가할수록 Associativity가 증가할수록 더 좋은 성능을 냈다. 가장 좋은 결과를 보여준 건 Associativity = 8, set = 2048이다.

- **Block size(Number of Sets = 512, Associativity = 1으로 고정)**

Block size	Unified cache Miss rate	AMAT
기준 16	$\frac{81960}{451517} = 0.1815$	39.3000
64	$\frac{22243}{451517} = 0.0493$	10.9416
128	$\frac{11489}{451517} = 0.0254$	6.2049
256	$\frac{6914}{451517} = 0.0153$	4.2299
512	$\frac{3289}{451517} = 0.0072$	2.6567

이번엔 블록사이즈를 다르게 해보며 테스트해보았다. 블록 사이즈가 클수록 miss rate는 줄었고 성능도 향상되었다. 가장 좋은 경우는 block size=512이다.

결과적으로 perl 벤치마크에는 가장 적합한 캐시 구성이 block size를 바꿔가며 진행한 시뮬레이션에서 나왔고 캐시 구성은 Block size를 512, Number of Sets = 512, Associativity = 1인 Unified Cache가 가장 좋은 성능을 보여주었다.

동일한 시뮬레이션으로 각각의 프로그램의 성능을 평가한 결과 적합한 캐시가 다른 이유는 프로그램마다 취하는 메모리 access의 방식과 경향이 다르기 때문이다. cc1은

컴파일러이기 때문에 명령어를 가져오는 access의 수가 많을 것이고 ijpeg는 사진을 압축하고 압축해제하는 데이터를 주로 access하는 경우가 많을 것이고 또 비슷한 데이터부분을 반복접근하는 경우가 많을 것이다. 마지막으로 perl은 명령어를 통해 데이터를 수정하는 경우가 많기에 둘 다 적절히 접근할 것이고 또한 이 세가지 모두 access의 수가 다르다. 따라서 적합한 캐시 구성의 차이가 발생한다.

이렇게 다양한 구성의 캐시를 벤치마킹을 통해 검증해본 결과 정렬 프로그램에 적합한 cache를 찾으라면 정렬 프로그램은 반복문을 통한 데이터 access를 진행하기 때문에 이와 비슷한 memory access를 진행하고 반복문을 많이 사용하는 ijpeg 벤치마킹의 결과를 보는게 적합하다고 생각한다. 따라서 ijpeg 벤치마킹에서 가장 좋은 성능을 보여준 캐시 구성인 Block size를 256, number of Sets = 512, Associativity = 1인 Unified Cache를 사용하는 것이 좋을 것으로 예상된다.

[Conclusion]

이번 프로젝트는 cache access가 다른 경우의 분석과 벤치마크에 따른 cache access의 성능을 비교하는 것이었다. insertion sorting과 random access은 어떻게 hit/miss의 상황에서 작동하는지를 waveform에서 각 신호를 통해 확인할 수 있었다. 전체 사이클 수를 같은 값으로 고정하였을 때 반복문을 사용하여 동일한 데이터를 재참조하는 경향이 있는 정렬 알고리즘쪽이 random access보다 더 hit의 비율이 높을 것이라 예상한다. 캐시 성능 비교시에는 AMAT을 사용하였는데 이는 평균 메모리 접근 시간으로 캐시 성능을 비교하는 척도이다. 캐시구성을 바꿔가며 성능을 테스트하면서 단순히 특정 구성을 과도하게 높인다고 높은 성능이 나는 것은 아닌 것을 알았다. 캐시 구성 요소들 사이의 적절한 밸런스를 유지해야 좋은 성능을 뽑아낼 수 있었다. 또한 결과적으로는 모든 벤치마크에 가장 적합한 캐시 구성을 block size를 바꿔가며 진행한 시뮬레이션에서 확인할 수 있었는데 이러한 이유를 간단하게 생각해보면 다른 시뮬레이션에서는 set을 증가시켜가며 성능을 비교하였다. 대체적으로 set이 증가할수록 성능은 좋아졌고 512가 적당히 최적화된 set의 크기임을 알 수 있다. 심지어 ijpeg 벤치마크를 associativity위주의 시뮬레이션으로 분석할 때 512를 넘어서면 오히려 성능이 감소되는 것 또한 확인할 수 있다. 따라서 적당히 최적화된 set의 크기에 블록 사이즈를 올려줬기에 가장 좋은 성능을 뽑아낼 수 있었지 않았을까라고 예상한다.