Computer Architecture

Project 4

(Cache Design)

학　　과: 컴퓨터정보공학부

담당교수: 이성원 교수님

실습분반: 금34

학　　번: 2020202037

성　　명: 염정호

1. 프로젝트 개요

본 프로잭트에서는 cache의 setting을 변경해 가며 해당 값의 변화가 캐시의 성능에 어떤 영향을 미치는지 알아본다. 캐시의 구성 요소로는 set의 개수, 하위 캐시의 존재 여부, L1캐시의 데이터와 명령어 캐시 분리, 통합캐시, ways의 변화 등 여러 요소가 캐시의 성능에 영향을 미친다. 캐시의 성능을 알 수 있는 기준으로 Average memory access time을 사용 하는데 이는 캐시 히트율과 캐시 미스율, 그리고 각 계층의 접근 시간을 기반으로 계산됩니다. AMAT는 다음과 같은 공식을 통해 계산된다.

AMAT=Hit time+(Miss rate×Miss penalty) 식을 이용해 계산 가능하며
Hit time: 데이터가 캐시에 있을 때 데이터를 접근하는 데 걸리는 시간.
Miss rate: 캐시에 데이터가 없는 경우의 비율 (미스율).
Miss penalty: 데이터가 캐시에 없어서 더 낮은 계층 (보통 메인 메모리)에서 데이터를 가져오는 데 걸리는 추가 시간을 각각 의미한다.

주어진 세 벤치마크 li, tomcat, vertex를 이용해 각각의 밴치마크가 어떤 사향의 캐시를 이용했을 때 최고의 성능을 발현하는지 확인하고 해당 데이터가 발생하는 이유와 조건 변경에 따른 캐시 성능의 변화를 설명한다. 원래의 캐시 성능이라면 AMAT외에도 비용과 같은 부가적인 측면을 고려해야 하지만 이번 실험에서는 AMAT만을 이용해 평가를 진행하였다. 다음으로 ramdom access 와 radix sort의 성능 차이를 비교한다.
MIPS-based 5-stage pipelined architecture에서 비교를 수행하며 32bit의 캐시 블록, 2 ways set. miss 발생시 LRU방식으로 캐시 블록을 교체하며 블록 하나의 크니근 4word로 지정한다.

2. 실험 결과

2.1Cache Simulation

Simulation 1. Unified vs splits

본 시뮬레이션은 명령어 메모리와 데이터 메모리가 분리된 split cache와 데이터와 명령어 메모리가 하나로 구성된 unfied cache에 대한 시뮬레이션을 진행했다. 대부분의 process의 경우
레벨 1 cache는 instruction 과 data가 분리된 캐시를 사용하는데 이는 분리해서 사용할 경우 상대적으로 많이 사용되는 명령어 캐시의 접근을 줄이는 것이 가능해 unified cache대비 missrate를 줄이는 것이 용이하기 때문이다. 하지만 명령어와 데이터를 분리해 2개의 cache를 다루기 때문에 복잡성이 증가하게 된다.
통합 캐시와 분할 캐시는 set의 개수에 따른 성능 차이도 확인 할 수 있는데 set의 개수가 증가하

면 캐시에 담을 수 있는 양이 증가한다. 하지만 우리가 사용하는 명령어의 개수는 한정적이다 때문에 분할 캐시에서 set의 개수가 커진다면 명령어를 담고 남은 메모리를 데이터 캐시로 활용하는 것이 불가능 하기 때문에 이는 성능의 저하로 이어진다.

	# of Sets	access	miss	hit time
li	64	261,190,238	81,188,978	180,001,260
	128	261,190,238	70,179,369	191,010,869
	256	261,190,238	34,612,401	226,577,837
	512	261,190,238	17,729,118	243,461,120

# of Sets	d1 access	d1 miss	d1 hit time	l1 access	l1 miss	l1 hit time
64	77,847,524	13,003,135	64,844,389	183,342,714	66,175,597	117,167,117
128	77,847,524	9,381,847	68,465,677	183,342,714	54,818,074	128,524,640
256	77,847,524	6,709,637	71,137,887	183,342,714	50,284,326	133,058,388
512	77,847,524	5,006,078	72,841,446	183,342,714	18,693,275	164,649,439

	# of Sets	access	miss	hit time
tomcat	64	332,199,527	120,680,188	211,519,344
	128	332,199,527	92,151,220	240,048,307
	256	332,199,527	73,877,547	258,321,980
	512	332,199,527	59,217,120	272,982,407

# of Sets	d1 access	d1 miss	d1 hit time	l1 access	l1 miss	l1 hit time
64	85,306,939	19,753,324	65,553,615	246,892,588	95,029,236	151,863,352
128	85,306,939	14,608,371	70,698,568	246,892,588	85,861,672	161,030,916
256	85,306,939	9,204,260	76,102,679	246,892,588	69,740,522	177,152,066
512	85,306,939	6,678,091	78,628,848	246,892,588	56,551,019	190,341,569

	# of Sets	access	miss	hit time
vortex	64	190,554	39,188	151,366
	128	190,554	34,232	156,322
	256	190,554	27,172	163,382
	512	190,554	19,398	171,156

# of Sets	d1 access	d1 miss	dl1 hit time	l1 access	l1 miss	l1 hit time
64	47,053	5,918	41,135	143,501	30,231	113,270
128	47,053	4,315	42,738	143,501	27,109	116,392
256	47,053	2,144	44,909	143,501	25,044	118,457
512	47,053	1,850	45,203	143,501	18,897	124,604

Block size = 16, Associativity = 1

name	# of Sets	Unified Cache Miss rate	Unified Cache AMAT	Split Cache		SplitCache AMAT
				Inst. Miss rate	Data Miss rate	
li	64	0.3108423	63.1684628	0.1670334	0.3609393	61.6291664
	128	0.2686906	54.7381255	0.1205157	0.2989924	50.1595103
	256	0.1325180	27.5035947	0.0861895	0.2742641	44.6417252
	512	0.0678782	14.5756360	0.0643062	0.1019581	19.1471966

li banch mark의 수행 결과를 보면 Set의 개수가 512이고 해당 cache를 unified로 사용했을 때 Average Memory Access Time이 가장 낮은 것을 볼 수 있는데 이러한 결과가 발생한 이유로는 split cache에서는 공간을 5:5 로 분할해서 사용했기 때문에 데이터 블록이 저장 될 수 있는 공간이 제한적이다. 때문에 data공간에 많은 access가 일어날수록 d1의 missrate가 증가하게 되고 이에 따라 cache를 split했음에도 unified한 캐시보다 더 안좋은 성능을 보인 것으로 추측된다.

name	# of Sets	Unified Cache Miss rate	Unified Cache AMAT	Split Cache		SplitCache AMAT
				Inst. Miss rate	Data Miss rate	
tomcat	64	0.3632762	73.6552437	0.2315559	0.3849011	70.1045897
	128	0.2773972	56.4794408	0.1712448	0.3477693	61.4877701
	256	0.2223891	45.4778159	0.1078958	0.2824731	48.5285337
	512	0.1782577	36.6515378	0.0782831	0.2290511	39.0669476

tomcat bench마크의 경우도 마찬가지로 unified한 cache에서 더 낮은 AMAT성능을 보이는 것으로 나타나는데 이는 data access와 intr의 access의 명령어의 비율에 의한 차이로 해석되며 위 표를 보면 set의 개수가 64개로 비교적 적을 때에는 split Cache의 성능이 더 좋음을 나타내는데 이는 명령어의 개수가 한정적이기 때문에 set의 개수가 적을 때에는 효율적인 cache운영이 가능하지만 cache의 set이 커져 명령어 데이터를 담고도 남을 정도가 되더라도 남은 공간을 data cache로 활용하지 못하기 때문에 이러한 차이가 발생한다.

name	# of Sets	Unified Cache Miss rate	Unified Cache AMAT	Split Cache		SplitCache AMAT
				Inst. Miss rate	Data Miss rate	
vortex	64	0.2056530	42.1305982	0.1257731	0.2106675	38.9409511
	128	0.1796446	36.9289230	0.0917051	0.1889116	33.9817270
	256	0.1425948	29.5189500	0.0455656	0.1745214	29.5357431
	512	0.1017979	21.3595831	0.0393174	0.1316855	22.7754547

votrex벤치마크를 분석해보면 이전 벤치마크 와는 달리 set의 개수가 256개 일 때 까지는 비슷한 성능을 보인다. 이는 vortex가 비교적 적은 횟수의 데이터 접근을 하는 것으로 생각한다. 이유는 split cache의 성능은 명령어 캐시를 사용할 때 상승하고 데이터 캐시를 사용할 때 비교적 떨어진다. 하지만 set의 개수를 256개 까지 키웠음에도 데이터의 miss rate가 증가하지 않았다는 것은 많은 공간의 data access를 필요로 하지 않는 것으로 생각했다.

위 세가지 bench mark모두 set의 개수가 512개인 통합 캐시에서 가장 좋은 성능을 보였다. 따라서 세 벤치마크 모두 set의 개수가 512개인 통합 캐시를 사용하는 것이 성능적 측면을 고려한다면 가장 좋다.

Simulation 2. L1/L2 size

본 시뮬레이션은 unified Cache에서 L2,L3 cache를 추가한다. L2, L3 cache란 L1 cache에서 miss가 발생할 경우 사용할 수 있는 다음 단계 캐시로 L1 cache보단 접근 하는데 더 큰 사이클이 소요되지만 memory에서 직접 데이터를 가져오는 것 보다는 클럭 사이클이 덜 소모 된다는 장점이 있다.

	RESULT				
	L1L2L3	Inst. Miss rate	data. Miss rate	Unified Cache Miss rate	AMAT
li	8 / 8 / 1024	0.515362519	0.314803872	0.071197112	15.52713
	16 / 16 / 512	0.436000893	0.235024739	0.160509099	19.07242
	32 / 32 / 256	0.360939334	0.167033379	0.36680262	26.88601
	64 / 64 / 128	0.298992378	0.120515676	0.870023775	44.37695
	128 / 128 / 0	0.274264108	0.086189472	-	50.15951

					access		
	L1/L1D/L2U	L1 miss	L1Dmiss	L2Umiss	L1	L1D	L2U
li	8/8/1024	94487963	24506702	9361406	183342714	77847524	131485755
	16 / 16 / 512	79937587	18296094	17291876	183342714	77847524	107731438
	32 / 32 / 256	66175597	13003135	31653483	183342714	77847524	86295684
	64 / 64 / 128	54818074	9381847	60515599	183342714	77847524	69556259
	128 / 128 / 0	50284326	6709637	-	183342714	77847524	-

위 지표는 L1 L2 L3 캐시가 커짐에 따라 각각 miss rate의 변화를 보여준다. 원래 생각했던 결과의 경우 속도가 빠른 L1,L2 cache의 크기가 증가하면 당연히 전체 clock수가 줄어들 것이라 생각해 128/128/0으로 캐시를 분할 했을 때가 가장 좋은 성능을 보일 것이라 생각했으나 실험 결과 8/8/1024로 분할한 캐시가 가장 좋은 성능을 보이는 것으로 나타났다. 해당 실험 결과에 대한 원인을 분석하자면 캐시의 전체적인 크기가 가장 크기 때문에 그만큼 main memory acccess를 줄일 수 있던 것이 아닐까 생각한다. 이에 대한 반증으로 I1의 분리된 캐시에서 missrate는 8/8/1024로 가장 높

지만 통합 캐시의 관점에서 봤을 때 가장 적기 때문에 결국 가장 적은 메인 메모리 access가 발생했을 것이다.

	L1L2L3	Inst. Miss rate	data. Miss rate	Unified Cache Miss rate	AMAT
tomcat	8 / 8 / 1024	0.531351144	0.324686002	0.244935087	31.29107
	16 / 16 / 512	0.488712338	0.278381786	0.615356025	57.67603
	32 / 32 / 256	0.38490113	0.231555888	0.590802271	44.51351
	64 / 64 / 128	0.347769338	0.171244815	0.800568453	50.5706
	128 / 128 / 0	0.28247313	0.107895795	-	48.52853

	L1/L1D/L2U	miss	miss	miss	L1	L1D	L2U
tomcat	8/8/1024	131186659	27697969	41563190	246892588	85306939	169690633
	16 / 16 / 512	120659454	23747898	94614756	246892588	85306939	153756122
	32 / 32 / 256	95029236	19753324	72054128	246892588	85306939	121959802
	64 / 64 / 128	85861672	14608371	84339371	246892588	85306939	105349356
	128 / 128 / 0	69740522	9204260	-	246892588	85306939	-

tomcat banch mark에서도 마찬가지로 8/8/1024로 캐시를 분할 했을 때 가장 성능이 좋은 것을 확인 할 수 있는데 이는 이전과 마찬가지로 memory access를 줄이는 것이 가장 큰 역할을 했다고 볼 수 있다 추가로 64/64/128과 128/128/0으로 분할 했을 때의 성능을 비교해 보면 L3를 사용하지 않고 같은 크기의 cache를 L1 L2로 분리한 것의 성능이 더 좋은 것을 확인 할 수 있는데 이는 메인 메모리와 가까운 곳의 캐시를 이용할수록 hit가 발생할 확률이 적은데 L3메모리 크기 조차 작다보니 해당 캐시 내에서 hit가 거의 발생하지 않아 이러한 결과가 발생했다.

	L1L2L3	Inst. Miss rate	data. Miss rate	Unified Cache Miss rate	AMAT
vortex	8 / 8 / 1024	0.278283775	0.212696321	0.201824737	15.55384
	16 / 16 / 512	0.225057665	0.052257476	0.426395003	21.08556
	32 / 32 / 256	0.210667521	0.041240131	0.639604668	26.56621
	64 / 64 / 128	0.188911576	0.030069477	0.87303978	30.19216
	128 / 128 / 0	0.174521432	0.014940662	-	29.53574

	L1/L1D/L2U	miss	miss	miss	L1	L1D	L2U
vortex	8/8/1024	39934	10008	10972	143501	47053	54364
	16 / 16 / 512	32296	7499	18500	143501	47053	43387
	32 / 32 / 256	30231	5918	25045	143501	47053	39157
	64 / 64 / 128	27109	4315	29562	143501	47053	33861
	128 / 128 / 0	25044	2144	-	143501	47053	-

해당 벤치마크 또한 마찬가지로 8/8/1024 크기로 나눠진 캐시에서 가장 좋은 성을 보이는 것을 확 확인 할 수 있는데 메인 메모리에 대한 접근을 L3캐시를 이용해 최대한 줄인 결과라고 볼 수 있다.

즉 L1 L2 L3캐시 분할 크기에 따른 성능의 경우 메인 메모리에 대한 access를 줄이는 것이 관건인데 이를 위해선 충분히 큰 크기의 L3 cache를 활용해 hit rate를 높이는 것이 중요하다.

Simulation 3. Associativity

Block size = 16

# of Sets	d1 access	l1 access
	77,847,524	183,342,714

# of Sets	d1 access	l1 access
	85,306,939	246,892,588

# of Sets	d1 access	l1 access
	47,053	143,501

Split Cache Miss rate / AMAT

# of Sets	name	1-way			2-way			3-way			4-way		
		l1 miss rate	d1 miss rate	AMAT	l1 miss rate	d1 miss rate	AMAT	l1 miss rate	d1 miss rate	AMAT	l1 miss rate	d1 miss rate	AMAT
64.	li	0.167033	0.360939	61.63	0.2214	0.0836	25.93	0.1452	0.0561	17.53	0.0537	0.0453	10.56
	tomcat	0.231556	0.384901	70.10	0.3429	0.1287	37.74	0.3077	0.0558	25.10	0.212	0.0233	15.35
	vertax	0.125773	0.210668	38.94	0.1904	0.0662	20.37	0.1714	0.0336	14.53	0.1505	0.026	12.35
128.	li	0.120516	0.298992	50.16	0.1558	0.0624	19.05	0.0643	0.0459	11.28	0.0055	0.0369	6.51
	tomcat	0.171245	0.347769	61.49	0.2858	0.0739	26.66	0.2157	0.036	17.43	0.1912	0.0204	13.85
	vertax	0.091705	0.188912	33.98	0.1733	0.0393	15.48	0.1401	0.0261	11.85	0.0494	0.0256	7.30
256.	li	0.086189	0.274264	44.64	0.1113	0.0487	14.47	0.0192	0.0374	7.40	0.0001	0.0316	5.44
	tomcat	0.107896	0.282473	48.53	0.2153	0.0552	20.26	0.1878	0.0306	15.19	0.0906	0.0196	8.57
	vertax	0.045566	0.174521	29.54	0.1354	0.0271	11.77	0.0571	0.0257	7.69	0.0154	0.025	5.53
512.	li	0.064306	0.101958	70.10	0.0261	0.0396	37.74	0.0003	0.0321	25.10	0.00002	0.028	15.35
	tomcat	0.078283	0.229051	39.07	0.186	0.0489	17.82	0.0951	0.0297	10.30	0.0008	0.0196	3.95
	vertax	0.039317	0.131685	22.78	0.0719	0.0258	8.44	0.0166	0.025	5.59	0.0142	0.0247	5.42
1024.	li	0.0379	0.05	10.28	0.0015	0.0333	5.76	0.000026	0.028	4.93	0.000018	0.0206	3.89
	tomcat	0.1812	0.066	20.12	0.1139	0.0319	11.59	0.0134	0.0169	4.20	0.000019	0.0143	3.13
	vertax	0.0864	0.0317	10.04	0.0278	0.0251	6.15	0.0148	0.0248	5.47	0.0141	0.0247	5.42
2048.	li	0.0087	0.0401	7.15	0.0001	0.0284	4.99	0.000019	0.0198	3.78	0.000018	0.0007	1.10
	tomcat	0.1236	0.0307	11.91	0.0349	0.0163	5.22	0.00002	0.0143	3.13	0.000019	0.0141	3.10
	vertax	0.0431	0.0275	7.27	0.0171	0.0248	5.50	0.0141	0.0247	5.31	0.0141	0.0247	5.31

본 시뮬레이션은 set의 개수와 ways의 개수에 따른 캐시 메모리의 변화를 확인해 볼 예정이다. 캐시에서 set의 증가는 캐시 메모리의 hit율 증가와 연관된다. 세트의 개수가 많아지면 인덱스를 갖는 메모리 주소가 캐시의 다른 세트에 분포되기 때문에 이는 캐시의 용량이 증가하는 것과 같은 역할을 수행 할 수 있다. ways의 증가는 세트 내에서 저장할 수 있는 블록을 늘리는 것이다. 이는 각 세트 내에서 더 많은 블록을 저장 할 수 있도록 도와준다.

name	# of Sets	1-way	2-way	3-way	4-way
li	64	61.63	25.93	17.53	10.56
	128	50.16	19.05	11.28	6.51
	256	44.64	14.47	7.40	5.44
	512	70.10	37.74	25.10	15.35
	1024	10.28	5.76	4.93	3.89
	2048	7.15	4.99	3.78	1.10

위 표를 통해 같은 ways를 사용할 경우 Set의 개수가 큰 것이 가장 좋은 AMAT를 보이며 같은 set의 크기에서 ways의 크기가 가장 클 때 가장 좋은 성능을 보이는 것으로 확인된다. 이유는 위에서 말했듯 세트수의 증가로 인한 데이터 탐색 시간의 감소와 set의 증가로 인해 캐시 안에 저장할 수 있는 데이터가 많아지면서 그에 따라 hit율이 증가한 것으로 보인다. 따라서 set의 개수가 2048이고 4ways를 이용할 때의 성능이 가장 우수한 것을 확인 할 수 있는데 이는 비용적인 측면을 제외하고 오롯이 성능만을 분석한 결과이며 비용적인 측면을 고려한다면 다른 방법을 사용할 수 있을 것이다.

name	# of Sets	1-way	2-way	3-way	4-way
tomcat	64	70.10	37.74	25.10	15.35
	128	61.49	26.66	17.43	13.85
	256	48.53	20.26	15.19	8.57
	512	39.07	17.82	10.30	3.95
	1024	20.12	11.59	4.20	3.13
	2048	11.91	5.22	3.13	3.10

위의 결과와 마찬가지로. ways와 set의 개수가 증가함에 따라 성능이 좋아지는 모습이 보인다. 하지만 이번 결과에선 set의 개수를 1024개를 사용하는 것과 2048개의 set을 사용하는 것에 대해 큰 성능 차이를 발견 할 수 없다. 따라서 이번 tomacat밴치마크를 사용한 결과에서는 1024개의 set과 4-ways방식을 이용한 캐시의 활용이 합리적으로 보인다.

name	# of Sets	1-way	2-way	3-way	4-way
vertex	64	38.94	20.37	14.53	12.35
	128	33.98	15.48	11.85	7.30
	256	29.54	11.77	7.69	5.53
	512	22.78	8.44	5.59	5.42
	1024	10.04	6.15	5.47	5.42
	2048	7.27	5.50	5.31	5.31

위 결과도 확인 해보면 set의 개수가 증가함에 따라 성능이 개선되고 있는 것 은 자명하나 256개의 set이후로는 큰 효과를 보지 못하는 것으로 확인된다 이는 해당 밴치마크의 특성일 수 있다. 해당 벤치마크 자체의 메모리 사용량이 적어 어느 순간부터 메모리의 크기나 set의 개수를 늘려도 크게 효과를 보지 못하는 것으로 판단된다. vertex의 경우 3ways 1024set을 사용하는 것이 합리적인 선택으로 보인다.

Simulation 4. Block size

Number of Sets = 512, Associativity = 1

시뮬레이션 4의 경우 set의 개수와 Associativity를 1로 고정시킨 뒤 캐시 블록으 크기를 조절하여 각 성능에 대한 분석을 실시한다 이는 해당 밴치마크의 공간 지역성을 확인 할 수 있는 실험으로 공간 지역성이라 특정 메모리에서 데이터를 가져올 때 블록 형태로 데이터를 가져와 해당 데이터 근처의 데이터가 쓰일 것이라는 가정 하에 같이 가져와 해당 주소에 대한 메모리 엑세스를 줄이는 것이 목적이다.

	BLOCKSIZE	access	miss	hit time	Unified Cache Miss rate	AMAT
li	16	261190238	17729118	243461120	0.06787818	13.71
	64	261190238	3808785	257381453	0.014582417	2.946
	128	261190238	1619200	259571038	0.006199313	1.252
	256	261190238	803579	260386659	0.003076604	0.621
	512	261190238	70519	261119719	0.000269991	0.055

위 실행 결과의 경우 블록 사이즈가 512일 때 가장 낮은 AMAT를 갖는 것으로 확인된다. 이는 어느정도 공간 지역성을 띄고 있다는 것을 나타낸다. 블록 사이즈가 16에서 64로 변화 할 때 가장 많은 AMAT감소 효과를 볼 수 있는데 이는 64bit를 이용한다면 해당 벤치마크의 공간지역성을 어느정도 해결 기에 충분한 사이즈라고 생각된다.

	BLOCKSIZE	access	miss	hit time	Unified Cache Miss rate	AMAT
tomcat	16	332199527	59217120	272982407	0.178257689	36.01
	64	332199527	13224822	318974705	0.039809876	8.042
	128	332199527	4847282	327352245	0.014591478	2.947
	256	332199527	1897351	330302176	0.00571148	1.154
	512	332199527	896670	331302857	0.002699191	0.545

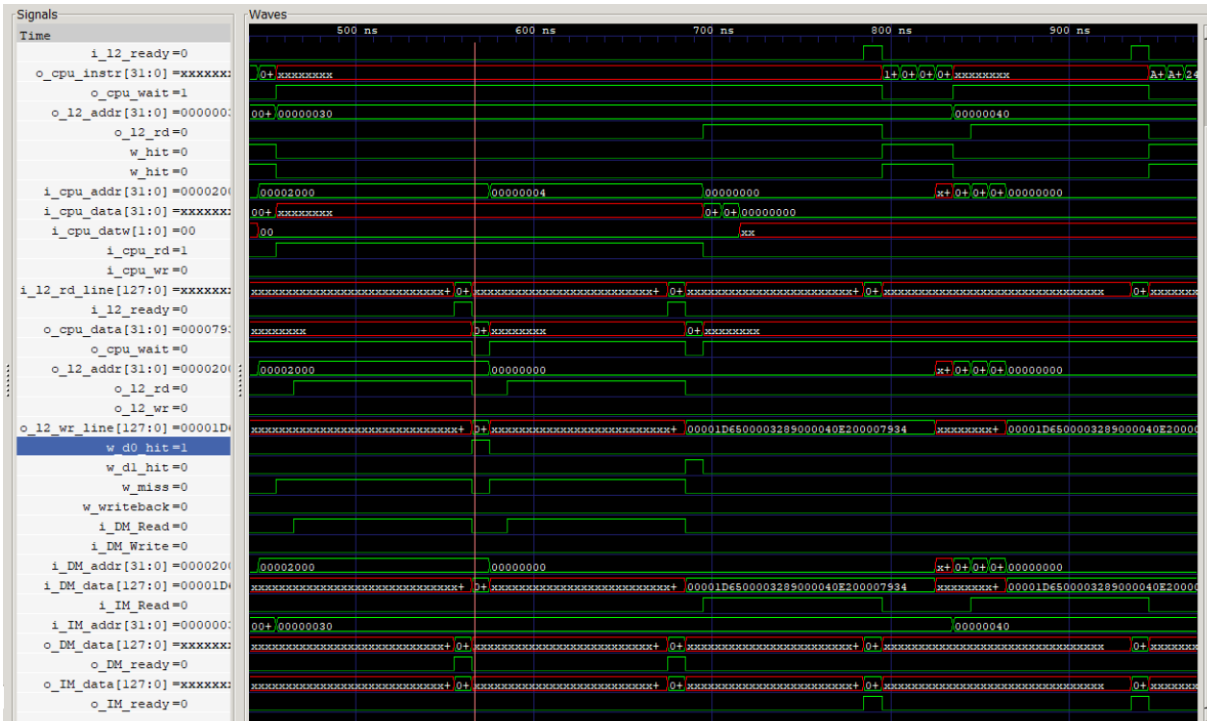
블록 사이즈가 가장 클 때인 512일 때 AMAT가 가장 작아지는 것을 확인할 수 있다 이는 해당 밴치마크가 공간 지역성을 띄는 것을 확인 할 수 있다. 하지만 너무 큰 블록 사이즈의 경우 캐쉬 내에서 탐색 시간을 증가시키고 블록사이즈가 증가한 만큼 캐시에 저장 할 수 있는 블록이 몇 개 안되기 때문에 가진 캐시의 크기에 따라 어느정도 절충이 필요할 것이다.

	BLOCKSIZE	access	miss	hit time	Unified Cache Miss rate	AMAT
vortex	16	190554	19398	171156	0.101797916	20.56
	64	190554	4653	185901	0.024418275	4.932
	128	190554	1820	188734	0.009551098	1.929
	256	190554	1261	189293	0.006617547	1.337
	512	190554	488	190066	0.002560954	0.517

이전 결과들과 마찬가지로 블록 사이즈가 가장 클 때 AMAT가 최소가 되는 것을 확인 할 수 있다. 이는 해당 벤치마크가 공간지역성을 갖는다는 의미가 된다.

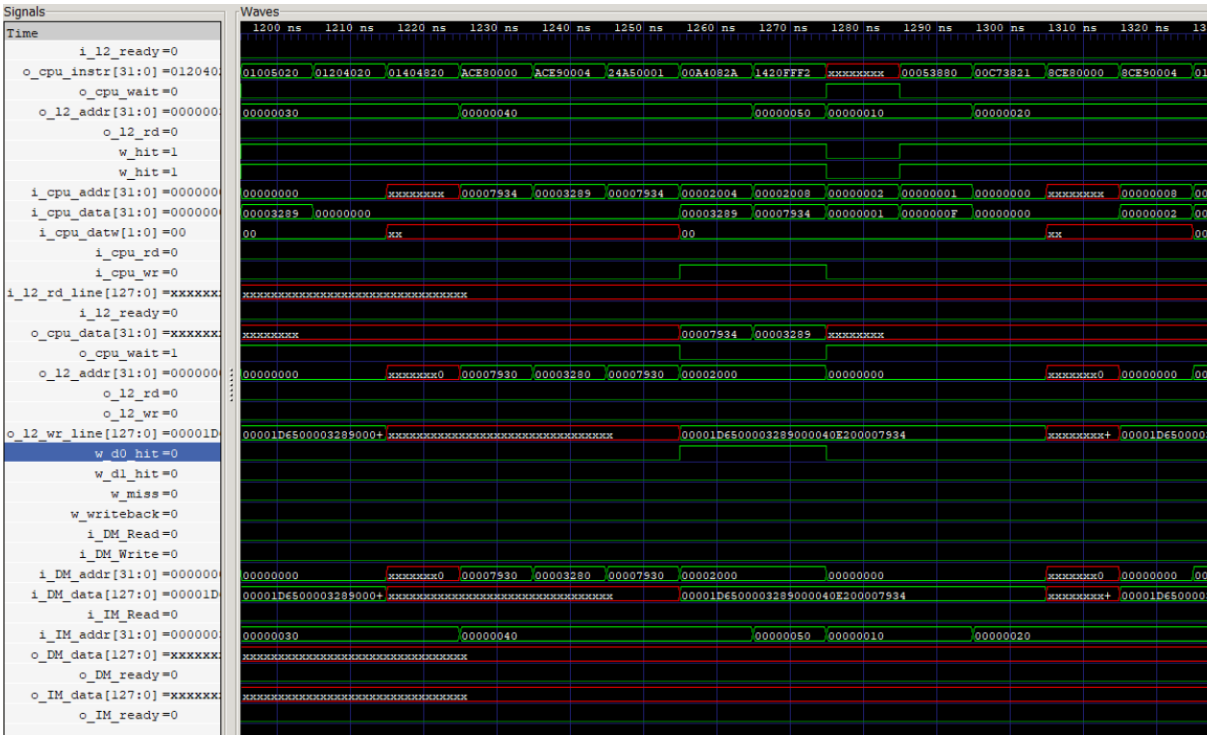
SORT 테스트 벤치 분석

intr hit

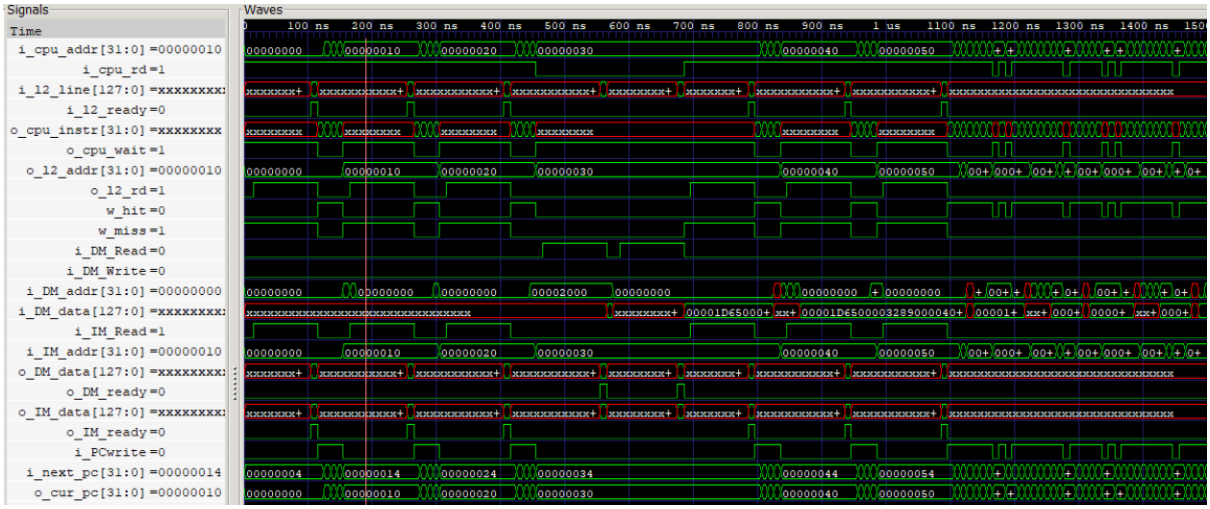


명령어 캐시에서 hit된 경우의 사진이다 fetch를 통해 명령어를 읽어온 후 L1메모리 캐시를 탐색하고 해당 캐시에 명령어가 존재하는 경우 이를 1cylce만에 처리 할 수 있다.

data hit



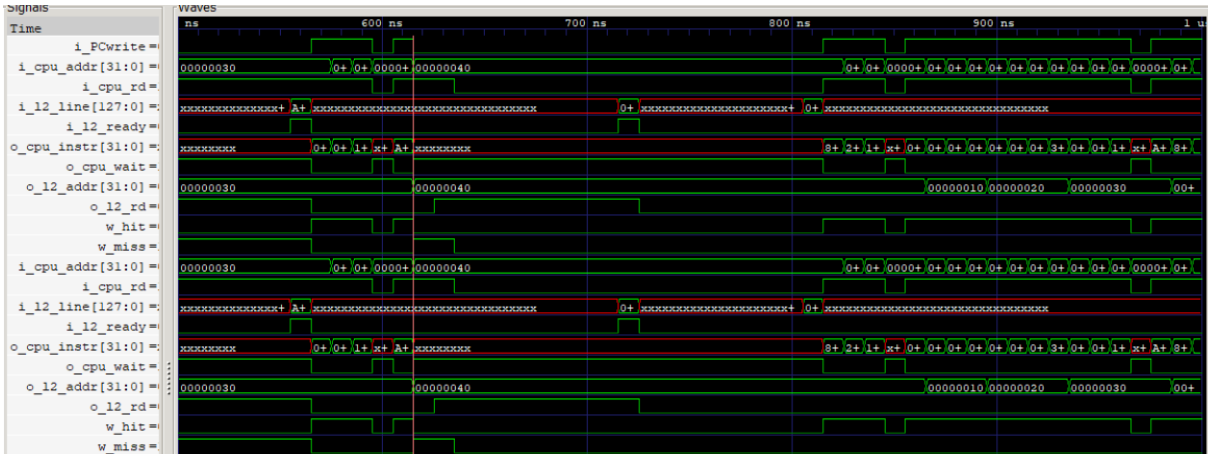
위는 데이터 에서 hit난 경우로 명령어 캐시와 마찬가지로 L1데이터 캐시에서 바로 1cycle만에 데이터를 가져 오는 것을 확인 할 수 있다.



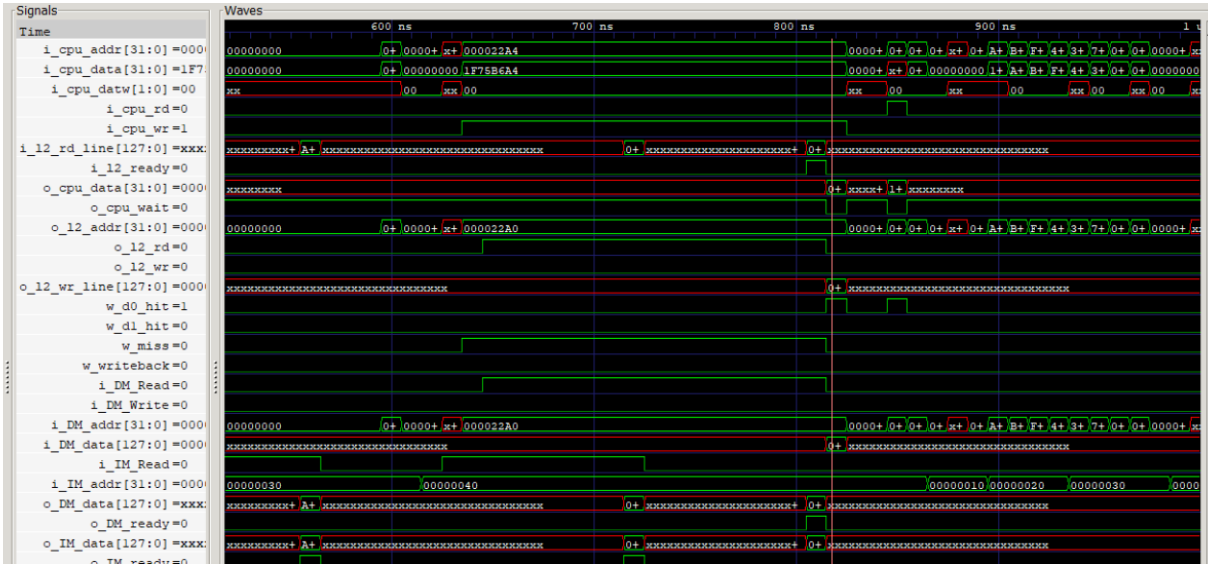
데이터 또는 명령어가 캐시에 존재하지 않을 경우 메이네 메모리에 access를 시도하며 해당 데이터 블록을 읽어와 ready상태가 on이라면 캐시에 해당 메모리 블록을 넣고 해당 캐시로부터 값을 읽어오는 것을 확인 할 수 있다.

Random Access 테스트 벤치 분석

intr메모리와 캐시 접근

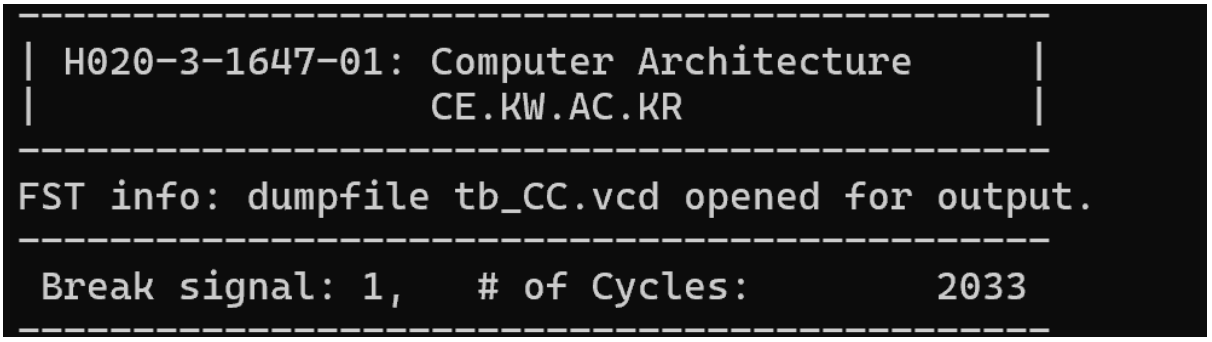


bubble sort와 마찬가지로 hit발생시 1cycle만에 해당 명령어를 가져와 사용하는 것이 가능하고 miss가 발생하면 intr memory에 접근해 해당하는 데이터 블록을 가져온 뒤 read를 통해 해당 블록을 캐시에 삽입한다.



데이터 캐시와 메모리도 마찬가지로 해당 데이터가 캐시에 존재하면 캐시에서 데이터를 가져오는데 1cycle이 소요되며 miss발생시 데이터 블록을 가져오기 위해 메모리에 접근해야 하며 해당 데이터 블록을 가져오면 read를 통해 데이터 블록을 캐시에 삽입한다.

bubble sort



random access

동일한 캐시를 이용한 hit발생과 miss발생시 처리가 같은 두 프로그램의 cycle이 1.5배나 차이가 나는 이유를 설명하기 위해서는 공간 지역성에 대해 알아야 한다. 캐시에서의 공간 지역성이란 최근에 접근 한 데이터 근처의 다른 데이터를 접근 한다는 것이다. 이를 이용하면 cach block을 사용하는 이유를 알 수 있다. 데이터를 하나만 가져오는 것이 아닌 block 단위로 가져오는 것은 근처 데이터가 사용 될 수 있기 때문이다. 때 bubbles sort의 경우 양옆의 데이터를 서로 비교해 가며 정렬을 시도하기 때문에 공간 지역성이 상다히 높은 편이다 반면에 random access의 경우 매번 현재 데이터와 연관 없는 어딘가의 데이터를 가져오기 때문에 제한된 block size내에서 공간 지역성을 띄기 어렵기 때문에 bubble sort에 비해 많은 cycle소모를 요구한다.

3. 고찰

이번 실험을 통해 캐시의 스펙 변화에 따른 성능 변화와 동일 캐시를 같은 기능을 수행하는 다른 알고리즘에서 적용시켰을 때 실제 성능 변화를 어떻게 관찰 할 수 있는지를 확인 할 수 있었다. 캐시의 스펙 변화에 따른 분석의 경우 처음 각 요소들의 역할을 이해하는 데에 오래 걸렸던것 같다. 해당 요소가 캐시의 어느 부분에 사용되는지 모른다면

실험 결과를 이해 할 수 없을 것이라고 생각했기 때문이다. Random Access와 bubble Sort의 분석 결과 random access는 데이터 접근 패턴이 불규칙하여 캐시 히트율이 낮았고, 이에 따라 사용되는 clk가 크게 증가하였다. 반면 radix sort는 정렬 알고리즘 특성상 데이터 접근 패턴이 비교적 규칙적이어서 캐시 히트율이 높았고, 사용되는 clk이 낮았다. 본 프로젝트를 통해 다양한 캐시 설정이 성능에 미치는 영향을 분석함으로써 특정 벤치마크에 최적화된 캐시 구성을 찾는 데 중요한 시사점을 제공하였다. AMAT를 통해 캐시 성능을 판단 할 수 있는 능력을 키울 수 있었다.