IS2202 - Computer Systems Architecture

Lab 1

Memory hierarchy, Coherence, Multithreading and Memory Models

-Submitted by

Tamilselvan Shanmugam tamsha@kth.se

Chandrasekaran Lakshminarayanan lnch@kth.se

3. Cache simulation with Simics

3.2 Collecting statistics from a simple cache

Tested on 23-Apr-2014; 03:20am; atlantis.it.kth.se

	Benchmark		
	vortex	equake	parser
Total number of Transactions	584375	483264	575749
Data read transactions	189750	127749	225332
Data read misses	47113	16560	29815
Data read hit ratio	75.17%	87.04%	86.77%
Instruction fetch transactions	257475	282845	225513
Instruction fetch missed	109388	58733	15219
Instruction fetch hit ratio	57.52%	79.23%	93.25%
Data write transactions	137150	72670	124904
Data write misses	99815	12947	53437
Data write hit ratio	21.22%	82.18%	57.22%

Simple cache size of 4kB with 128 lines of each 32 byte.

Best performance: Quake benchmark had shown fairly good performance than other two. Even though Quake's Instruction fetch hit ratio is lower than Parser benchmarks, Quake exceeds in Data read hit ratio and Data write hit ratio.

Worst performance: vortex is the ugly guy. Its benchmarks are far lower than other two.

3.3 Determining benchmark working set size

Working set is the size of cache when hit rate is close to hundred percentages.

Tested on 24-Apr-2014; 04:45pm; atlantis.it.kth.se

	vortex			
	8kB	16kB	32kB	64kB
Data read hit ratio	83.74%	90.69%	91.61%	97.39%
Instruction fetch hit ratio	75.29%	84.99%	90.42%	95.67%
Data write hit ratio	43.21%	80.23%	50.50%	98.75%

	quake			
	8kB	16kB	32kB	64kB
Data read hit ratio	85.65%	91.57%	95.72%	95.17%
Instruction fetch hit ratio	83.68%	85.53%	96.75%	97.26%
Data write hit ratio	82.58%	98.30%	99.28%	94.86%

	parser			
	8kB	16kB	32kB	64kB
Data read hit ratio	88.01%	93.47%	96.60%	97.81%
Instruction fetch hit ratio	97.14%	96.63%	98.97%	99.13%
Data write hit ratio	38.23%	80.15%	84.52%	94.11%

Since we don't have all the column values very close to hundred, let's say the size of working set is 95% rather than 100%.

Benchmark	Approx working set
vortex	64kB
quake	32kB
parser	64kB

Largest working set: By looking at the table, it's obvious that Data write hit ratio in vortex and quake is not increasing gradually as the cache size increases. This makes the calculation derail. With the available information, it seems that the **"vortex"** benchmark is lagging in numbers compares to parser benchmark. This implies that vortex should have the largest working set.

3.4 Minimal instruction and data caches

Tested on 25-Apr-2014; 10:30am; atlantis.it.kth.se

	vortex		quake		parser	
	32B	128B	32B	128B	32B	128B
Data read hit ratio	23.35%	26.09%	4.62%	24.35%	28.71%	34.23%
IF hit ratio	35.84%	61.60%	42.34%	57.00%	22.41%	70.92%
Data write hit ratio	1.19%	2.16%	33.14%	15.23%	24.80%	27.97%

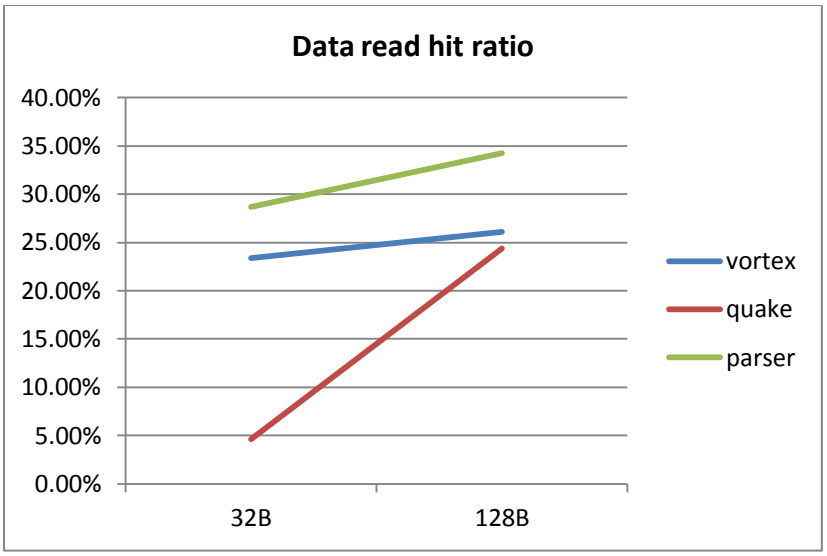
Relative locality of data vs Instruction access:

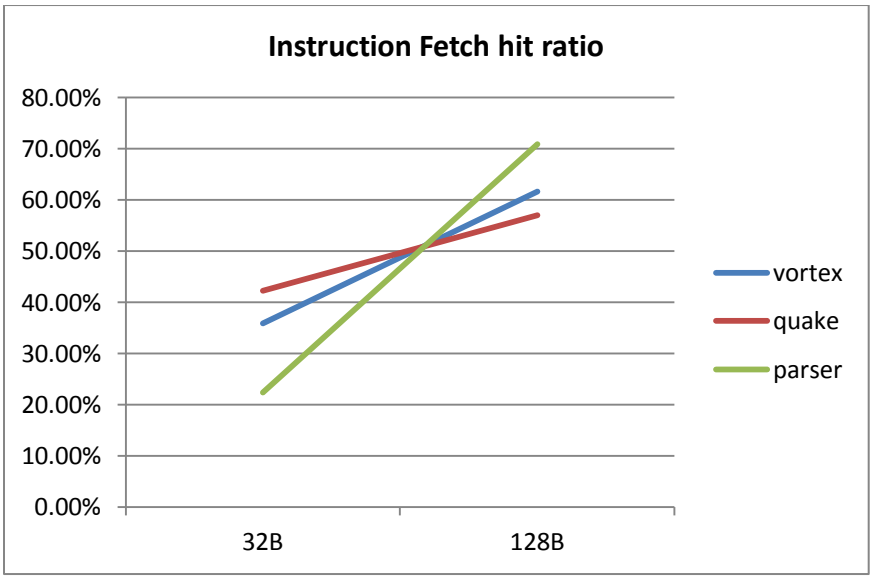
Spatial locality of data may or may not be utilized. It depends on the program. The result shows that the usage of spatial locality in all benchmarks. It's poorly used in quake benchmarks which resulted lower data access hit rate.

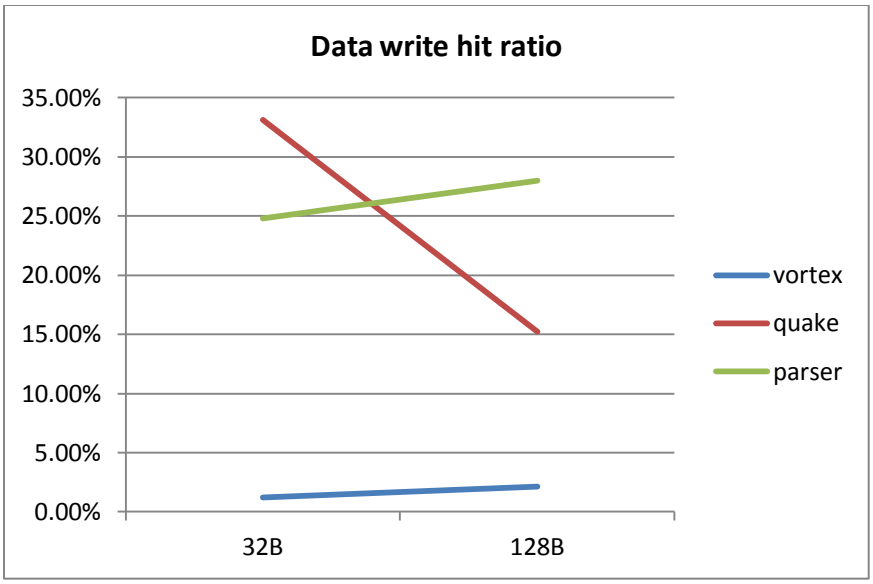
Instructions are stored adjacent in memory which resulted in better use of spatial locality. Unless there is a branch, always spatial locality is maintained.

Increasing the line size:

Increasing cache line size increases spatial locality as expected in all the benchmarks, all the parameters except data write hit ratio of quake benchmark. It reads data from the cache line and tries to write in a location which is not in the cache. This destroys the performance.


Data read hit ratio
40.00%
35.00%
30.00%
25.00%
20.00%
15.00%
10.00%
5.00%
0.00%
32B
128B
vortex
quake
parser


Instruction Fetch hit ratio
80.00%
70.00%
60.00%
50.00%
40.00%
30.00%
20.00%
10.00%
0.00%
32B
128B
vortex
quake
parser



3.5 Collecting statistics from a cache hierarchy

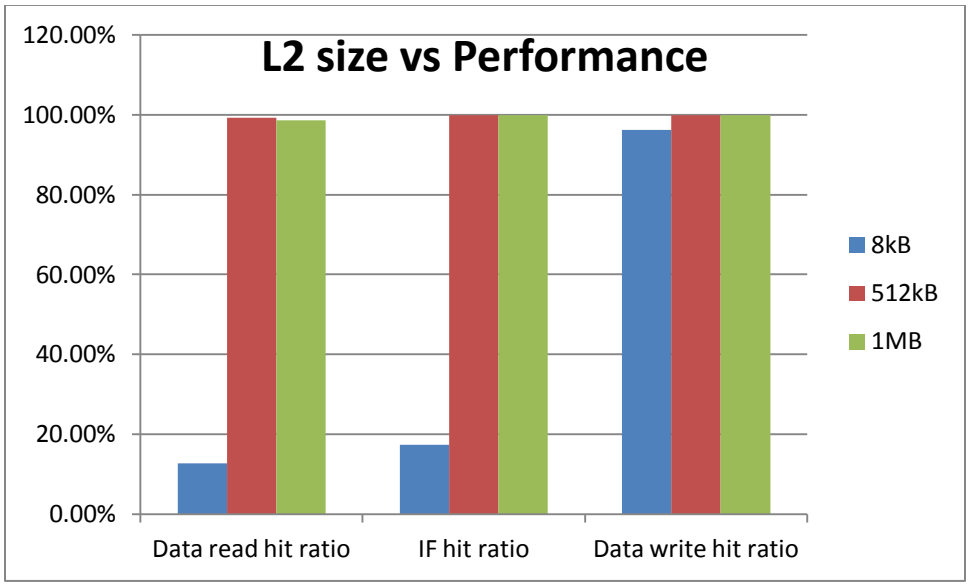
Tested on 26-Apr-2014; 01:10am; atlantis.it.kth.se

	512kB L2 cache					
	vortex		quake		parser	
	L1	L2	L1	L2	L1	L2
Data read hit ratio	88.97%	99.29%	93.26%	91.78%	95.16%	63.70%
IF hit ratio	80.78%	99.94%	96.31%	99.38%	98.90%	99.30%
Data write hit ratio	63.72%	99.94%	96.17%	100.00%	81.08%	99.99%

512kB L2 cache configuration is keeping well backup of L1 cache misses hence produces close to 100% data availability.

Varying L2 size:

	vortex					
	8kB		512kB		1MB	
	L1	L2	L1	L2	L1	L2
Data read hit ratio	90.27%	12.79%	88.97%	99.29%	88.51%	98.52%
IF hit ratio	88.26%	17.37%	80.78%	99.94%	80.82%	99.80%
Data write hit ratio	69.44%	96.11%	63.72%	99.94%	65.48%	99.94%



Smaller L2 is not extensively collecting data. Any miss in L2 cache goes to memory access which is expensive operation. It takes nearly 200 cycles to bring the data in. So choosing 8kB L2 is the worst idea.

Both 1MB, 512kB of L2 are pretty same. L1 cache in 1MB configuration out performs L1 cache in 512kB L2 configuration in terms of data write hit ratio. It's wise to go with 512kB configuration for better performance and optimal cost.

Access time:

Access time = L1 access time + L2 access time + Memory access time

L1 access cycles = L1 hit_rate x 3

L2 access cycles = (1 – L1 hit_rate) x L2 hit_rate x 10

L3 access cycles = (1 - (1 – L1 hit_rate) x L2 hit_rate) x 200

	Data read	Instruction Fetch	Data write
vortex 8kB	19.8	22.25	7.39
vortex 512kB	3.92	4.36	5.58
vortex 1MB	4.12	4.41	5.45
quake 512kB	4.52	3.3	3.26
parser 512kB	6.67	3.09	4.32

Access time table proves that the 512kB design is better in terms of cost without compromising performance.

3.6 Open-ended exercise: Application tuning for a given cache hierarchy

Assumptions:

Clock speed is 2GHz i.e 2 cycles every nano second.

L2 access time = 12 cycles

Memory access time = 200 cycles

Access pattern of gemm.c

```
A[0]     B[0]
A[128]     B[1]
A[256]     B[2]
A[384]     B[3]
C[0]

A[1]     B[0]
A[129]     B[1]
A[257]     B[2]
A[385]     B[3]
C[1]

A[2]     B[0]
A[130]     B[1]
A[258]     B[2]
A[386]     B[3]
C[2]
```

This pattern continues for next iterations. These information are sufficient to decide the cache size.

L2 cache:

We have seen from the above simulations that 512kB of L2 cache gives excellent performance in all the benchmarks. So we will reduce L2 size from 512kB and check for the performance.

L1 cache:

From the access pattern we can see that it is accessing 4 consecutive elements for 4 iterations. Accommodate them in a cache line to get the maximum performance.

Double size: 8 bytes;

No.of Elements in a line: 4

Line size: 4 x 8 Byte = 32B

No.of lines per way: 4

No.of ways: 3 (matrix A,B,C) + 1 (other variables)

	L1	L2	Configuration	
DR hit ratio	96.34%	99.83%	Instru cache	128B=4x32B 1way
IF hit ratio	12.75%	99.98%	L1 cache	512B=16x32B 4way
DR hit ratio	90.24%	99.88%	L2 cache	256kB=2048x128B 8way

Instruction cache size has to be increased to improve the hit ratio. L2 cache is performing well. So we can still reduce the L2 size and check it.

	L1	L2	Configuration	
DR hit ratio	96.10%	95.41%	Instru cache	512B=16x32B 1way
IF hit ratio	45.21%	99.93%	L1 cache	512B=16x32B 4way
DW hit ratio	90.68%	99.77%	L2 cache	128kB=1024x128B 8way

	L1	L2	Configuration	
DR hit ratio	96.15%	92.41%	Instru cache	1kB=32x32B 1way
IF hit ratio	79.98%	99.62%	L1 cache	512B=16x32B 4way
DW hit ratio	90.74%	99.79%	L2 cache	64kB=512x128B 8way

	L1	L2	Configuration	
DR hit ratio	79.87%	77.94%	Instru cache	1kB=32x32B 1way
IF hit ratio	72.58%	86.01%	L1 cache	512B=16x32B 4way
DW hit ratio	50.02%	97.64%	L2 cache	32kB=256x128B 8way

32kB L2 configuration deteriorates the performance. We will go to one level up. 64kB L2, 512B L1 would give the best hit rate and access time.

Instruction Cache	L1	L2
1kB=32x32B 1way	512B=16x32B 4way	64kB=512x128B 8way

64kB cache

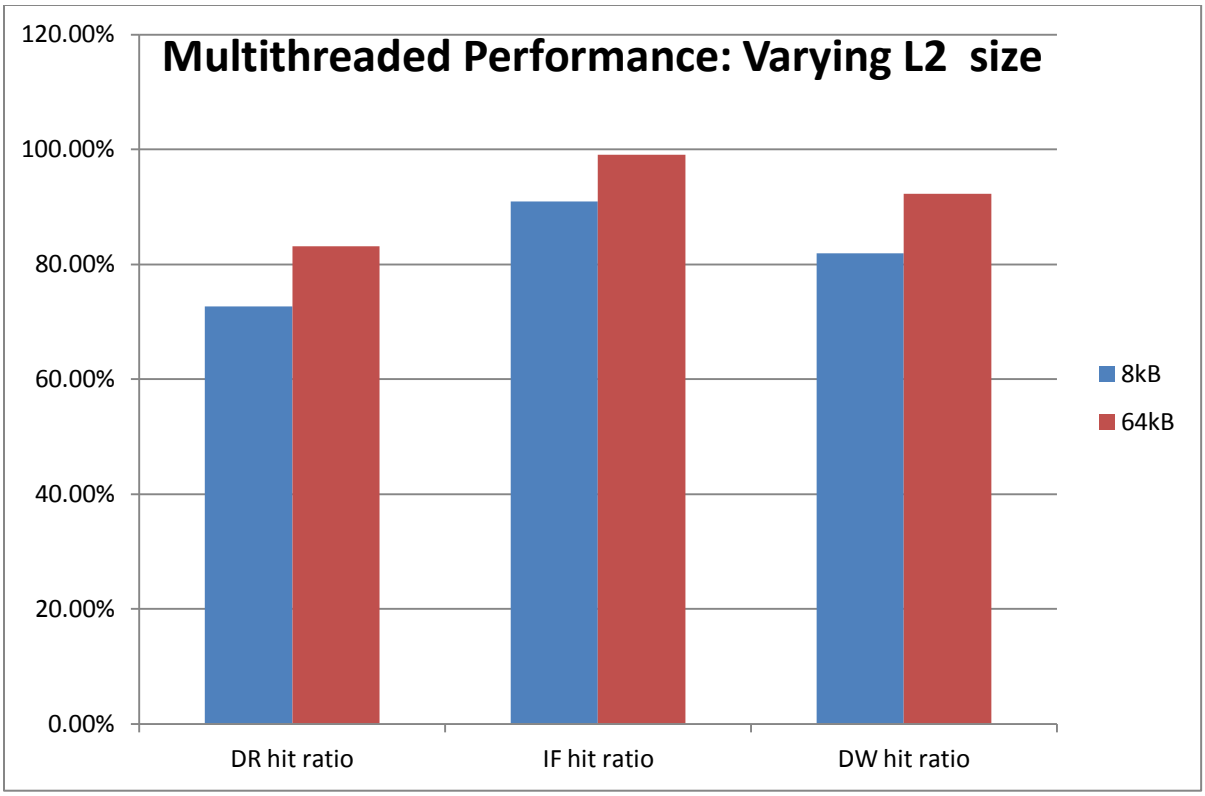
Access time (ns)	5.83
Total area (mm^2)	34.9
Total read dynamic energy per read port(nJ)	7.71

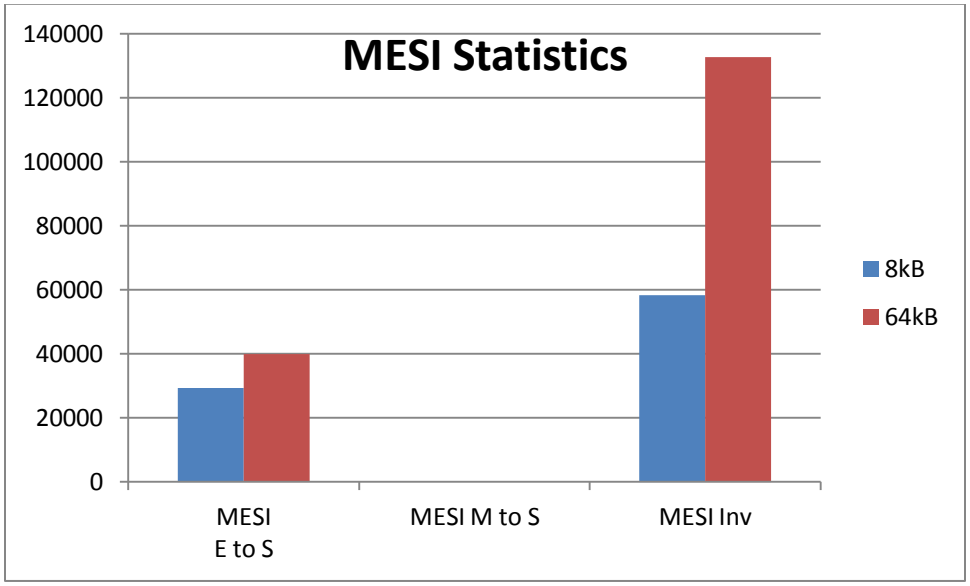
	Avg Access cycles
Data Read	2.2
Instruction Fetch	3.3
Data Write	2.1

4 Multithreading simulation with Simics

Level	Size	DR hit ratio	IF hit ratio	DW hit ratio	MESI E to S	MESI M to S	MESI Inv
L1_0	8kB	65.57%	88.30%	80.44%	29673	0	59313
	64kB	79.87%	99.25%	91.79%	40655	0	134681
L1_1	8kB	64.99%	89.04%	83.09%	30074	0	58670
	64kB	80.09%	99.28%	92.43%	44433	0	123834
L1_2	8kB	66.20%	89.23%	83.23%	29907	0	58547
	64kB	79.71%	99.05%	84.16%	46491	0	128554
L1_3	8kB	66.46%	89.11%	83.53%	29642	0	58770
	64kB	78.93%	99.04%	93.41%	38494	0	143077
L1_4	8kB	66.39%	89.12%	81.99%	29398	0	59059
	64kB	79.04%	99.25%	92.27%	37295	0	133493
L1_5	8kB	66.17%	89.03%	82.48%	29055	0	58212
	64kB	78.63%	99.09%	91.45%	33400	0	139969
L1_6	8kB	66.04%	88.22%	70.50%	27370	0	56612
	64kB	79.20%	99.12%	93.08%	39676	0	125735
L1_7	8kB	66.05%	88.97%	81.43%	29007	0	57495
	64kB	79.36%	99.34%	93.50%	37974	0	131688
Avg	8kB	65.98%	88.88%	80.84%	29265.8	0	58334.8
	64kB	79.35%	99.18%	91.51%	39802.3	0	132629
L2 512kB	8kB	99.23%	99.85%	94.18%	NA	NA	NA
	64kB	99.44%	98.54%	92.05%	NA	NA	NA

In all the cases, increasing L1 cache size increases hit rate as the cache can hold more amount of data. As we studies earlier, 512kB L2 cache gives outstanding performance regardless of the benchmark. This face is again proved here.





Boosting cache capacity increases the data shared across other caches which eventually resulted in more sharing and invalidations. Higher invalidation implies occupying higher bus bandwidth. Under bandwidth constrains, we can implement SCI (Scalable Coherence Interface) for better performance.

4.5 Open-ended exercise: Design space exploration of the shared memory hierarchy

Given:

Size (kB)	1	2	4	8	16	32	64	128	256	512	1024
Access time (cycles)	1	2	3	4	5	6	7	8	9	10	10

From the benchmark simulations, we have seen that 512kB L2 cache is delivering good hit ratio. So there is no need for increasing L2 cache size beyond 512kB. Let's concentrate L1 cache.

		DR hit rate	IF hit rate	DW ht rate
8kB	L1_0	65.57%	88.30%	80.44%
	L1_1	64.99%	89.04%	83.09%
	L1_2	66.20%	89.23%	83.23%
	L1_3	66.46%	89.11%	83.53%
	L1_4	66.39%	89.12%	81.99%
	L1_5	66.17%	89.03%	82.48%
	L1_6	66.04%	88.22%	70.50%
	L1_7	66.05%	88.97%	81.43%
	Avg	65.98%	88.88%	80.84%
512kB	L2	99.23%	99.85%	94.18%
Avg Access cycles		7.21	4.9	7.6

		DR hit rate	IF hit rate	DW ht rate
64kB	L1_0	79.87%	99.25%	91.79%
	L1_1	80.09%	99.28%	92.43%
	L1_2	79.71%	99.05%	84.16%
	L1_3	78.93%	99.04%	93.41%
	L1_4	79.04%	99.25%	92.27%
	L1_5	78.63%	99.09%	91.45%
	L1_6	79.20%	99.12%	93.08%
	L1_7	79.36%	99.34%	93.50%
	Avg	79.35%	99.18%	91.51%
512kB	L2	99.44%	98.54%	92.05%
Avg Access cycles		8.05	7.06	8.69

		DR hit rate	IF hit rate	DW ht rate
4kB	L1_0	57.96%	82.28%	75.88%
	L1_1	56.96%	81.77%	74.88%
	L1_2	58.39%	82.23%	75.73%
	L1_3	58.69%	82.56%	75.75%
	L1_4	56.98%	80.10%	62.25%
	L1_5	58.11%	82.20%	75.08%
	L1_6	58.45%	82.20%	74.26%
	L1_7	58.74%	82.62%	75.33%
	Avg	58.04%	82.00%	73.65%
512kB	L2	99.26%	99.90%	94.82%
Avg Access cycles		7.3	4.65	7.9

		DR hit rate	IF hit rate	DW ht rate
64kB	L1_0	81.31%	99.10%	92.80%
	L1_1	80.74%	99.20%	92.66%
	L1_2	80.33%	98.95%	86.40%
	L1_3	79.24%	99.06%	92.23%
	L1_4	79.71%	99.14%	91.10%
	L1_5	79.20%	99.05%	92.13%
	L1_6	79.69%	99.09%	92.70%
	L1_7	80.34%	99.15%	91.66%
	Avg	80.07%	99.09%	91.46%
256kB	L2	78.26%	75.54%	89.71%
Avg Access cycles		15.67	7.44	8.84

Above analysis shows that cache size and hit rate are directly proportional. However, time to access cache also increases as size grows. 8kB L1, 512kB L2 combination and 4kB L1, 512 kB L2 combination performs approximately equal.

4kB L1 - 1way	
Access time (ns)	0.58
Total read dynamic energy per read port(nJ)	0.14
Total area (mm^2)	1.06

512kB L2 - 8way	
Access time (ns)	3.48
Total read dynamic energy per read port(nJ)	2.06
Total area (mm^2)	17.95

Size	
L1_0 to L1_7	4kB
L2	512kB
Total	544kB

Considering space occupied, energy guidelines, faster access time, implementation complexity factors 4kB L1 and 512kB L2 recipe is the best to put into action.

As far as we have simulated, 64kB L1 and 256kB L2 combination gives bad performance than other configurations.

5. Directed portion:

1. Running critical section with pthread

./lab1.5 -t critical –c pthread

Execution:

Using the test section in "test_critical.c" which has increment and decrement code, with argument "critical" we are executing two threads where one thread is assigned with increment and another thread is assigned with decrement functions.

Since no iteration is mentioned, the code snippet will execute for default iteration count.

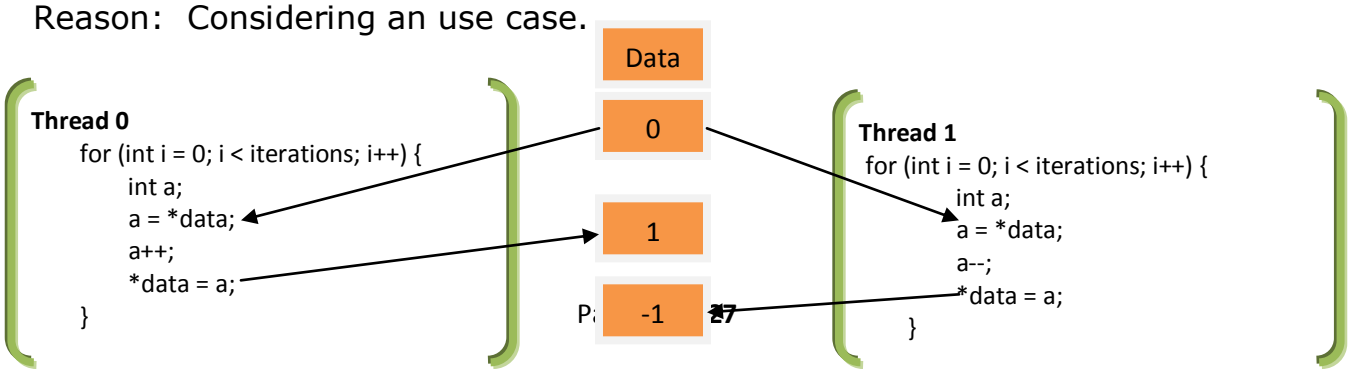
Ptherad is the critical section implementation algorithm we are using. (Since the critical implementation code is not called in "test" functions, this is of no use)

Output:

Since the each thread's critical section is not protected, the shared variable is modified by each thread inconsistently. Hence final output will not be 0.

```
[lnch@subway lab1_5]$ ./lab1.5 -t critical -c pthreads
Configuration:
        Test implementation: critical
        Critical sections implementation: pthreads
        Iterations: 1000000
Statistics:
        Thread 0: 0.0105 s (9.5157e+07 iterations/s)
        Thread 1: 0.0106 s (9.4572e+07 iterations/s)
        Average execution time: 0.0105 s
        Avergage iterations/second: 9.4863e+07
INCONSISTENCY after 1000000 iterations - data = -101206, but started at 0
```

Reason: Considering an use case.



2. Implementing "enter_critical" and "exit_critical" into test_critical.c

Execution:

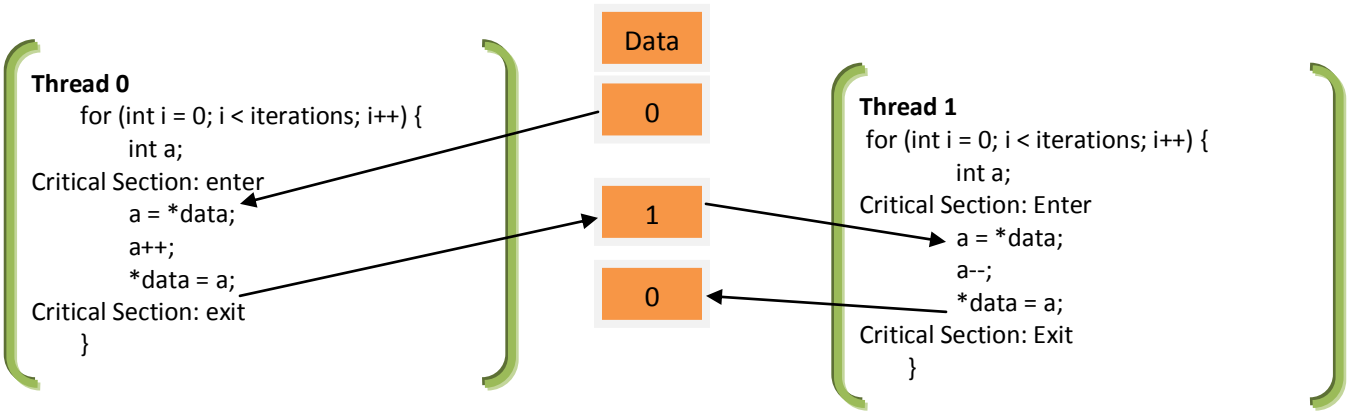
$ ./lab1.5 -t critical -c pthreads

Test code critical section is protected with "enter_critical" and "exit_critical".

Output:

Since critical is protected, each thread will have the synchronisation, results in "0"

```
[lnch@subway lab1_5]$ ./lab1.5 -t critical -c pthreads
Configuration:
        Test implementation: critical
        Critical sections implementation: pthreads
        Iterations: 1000000
Statistics:
        Thread 0: 0.2330 s (4.2917e+06 iterations/s)
        Thread 1: 0.2197 s (4.5522e+06 iterations/s)
        Average execution time: 0.2263 s
        Avergage iterations/second: 4.4181e+06
NO INCONSISTENCY after 1000000 iterations
```



3. Dekker's algorithm for synchronization:

Execution:

./lab1.5 -t critical -c dekker

Output:

```
[lnch@subway lab1_5]$ ./lab1.5 -t critical -c dekker
Configuration:
        Test implementation: critical
        Critical sections implementation: dekker
        Iterations: 1000000

Statistics:
        Thread 0: 0.1059 s (9.4433e+06 iterations/s)
        Thread 1: 0.1052 s (9.5081e+06 iterations/s)
        Average execution time: 0.1055 s
        Avergage iterations/second: 9.4756e+06

INCONSISTENCY after 1000000 iterations - data = 14289, but started at 0
```

Reason:

a) Why flag and turn variable have to be volatile?

Since both the threads checking and updating flag and turn. If this is not volatile, the flag data is read from cache, hence to read other thread's update at right time we need to read from main memory. For that only volatile is used.

b) Why straight forward implementation doesn't work?

In straight forward implementation, the writing of flag value order may change, due to that; other thread also would enter in to critical section simultaneously.

Hence in this scenario both exit from while loop, and execute the critical section.

```
flag[0] = 1;
while (flag[1]) {
        if (turn != 0) {
                flag[0] = 0;
                while (turn != 0);
                flag[0] = 1;
        }
}
```

Flag[1] = 0

```
flag[1] = 1;
while (flag[0]) {
        if (turn != 1) {
                flag[1] = 0;
                while (turn != 1);
                flag[1] = 1;
        }
}
```

Flag[0] = 0

4. Dekker's Algorithm with memory fence:

After adding memory fence,

```
flag[0] = 1;
MFENCE();
 while (flag[1]) {
        if (turn != 0) {
                flag[0] = 0;
                while (turn != 0);
                flag[0] = 1;
                 MFENCE();
           }
}
```

```
flag[0] = 1;
 MFENCE();
 while (flag[1]) {
        if (turn != 0) {
                flag[0] = 0;
                while (turn != 0);
                flag[0] = 1;
                 MFENCE();
           }
}
```

Output:

```
[lnch@subway lab1_5]$ ./lab1.5 -t critical -c dekker
Configuration:
        Test implementation: critical
        Critical sections implementation: dekker
        Iterations: 1000000
Statistics:
        Thread 0: 0.2555 s (3.9132e+06 iterations/s)
        Thread 1: 0.2556 s (3.9123e+06 iterations/s)
        Average execution time: 0.2556 s
        Avergage iterations/second: 3.9127e+06
NO INCONSISTENCY after 1000000 iterations
```

5. Atomic and non Atomic increment:

Non Atomic increment:

Output:

```
[lnch@subway lab1_5]$ ./lab1.5 -t incdec_no_atomic
Configuration:
        Test implementation: incdec_no_atomic
        Critical sections implementation: pthreads
        Iterations: 1000000
Statistics:
        Thread 0: 0.0098 s (1.0201e+08 iterations/s)
        Thread 1: 0.0099 s (1.0130e+08 iterations/s)
        Average execution time: 0.0098 s
        Avergage iterations/second: 1.0165e+08
INCONSISTENCY after 1000000 iterations - data = -115176, but started at 0
```

Reason:

In asm_dec_int32 operation, the increment happened non-atomically. Hence while incrementing, other thread can change the shared variable value and update it. Thus forms the inconsistency

Atomic increment:

Output

```
[lnch@subway lab1_5]$ ./lab1.5 -t incdec_atomic
Configuration:
        Test implementation: incdec_atomic
        Critical sections implementation: pthreads
        Iterations: 1000000
Statistics:
        Thread 0: 0.0338 s (2.9602e+07 iterations/s)
        Thread 1: 0.0353 s (2.8347e+07 iterations/s)
        Average execution time: 0.0345 s
        Avergage iterations/second: 2.8961e+07
NO INCONSISTENCY after 1000000 iterations
```

Reason:

In asm_atomic_inc_int32, the increment operation happens atomically. So other thread cannot interrupt and modify the shared variable data. In that way the increment and decrement happen synchronously.

6. Compare and Exchange:

a) Non atomic compare and Exchange:

Output:

```
Non atomic instruction:
[lnch@subway lab1_5]$ ./lab1.5 -t cmpxchg_no_atomic
Configuration:
        Test implementation: cmpxchg_no_atomic
        Critical sections implementation: pthreads
        Iterations: 1000000
Statistics:
        Thread 0: 0.0119 s (8.4225e+07 iterations/s)
        Thread 1: 0.0121 s (8.2775e+07 iterations/s)
        Average execution time: 0.0120 s
        Avergage iterations/second: 8.3493e+07
INCONSISTENCY after 1000000 iterations - data = -56699, but started at 0
```

Reason:

```
a = *data;
 asm_cmpxchg_int32(data, a, (a+1));
```

```
asm_cmpxchg(*mem, old, new):
        load mem -> tmp
        if tmp == old:
                store new -> mem
return tmp
```

At any instance of the above code, other thread can modify the global data, hence the synchronisation is not maintained at all, and though we check the return value the consistency is not guaranteed.

b) Atomic compare and Exchange:

Output:

```
[lnch@subway lab1_5]$ ./lab1.5 -t cmpxchg_atomic
Configuration:
        Test implementation: cmpxchg_atomic
        Critical sections implementation: pthreads
        Iterations: 1000000
Statistics:
        Thread 0: 0.0532 s (1.8790e+07 iterations/s)
        Thread 1: 0.0562 s (1.7808e+07 iterations/s)
        Average execution time: 0.0547 s
        Avergage iterations/second: 1.8286e+07
INCONSISTENCY after 1000000 iterations - data = 1, but started at 0
```

Reason:

```
a = *data;
 asm_atomic_cmpxchg_int32(data, a,
(a+1));
```

```
asm_cmpxchg(*mem, old, new):
        load mem -> tmp
        if tmp == old:
                store new -> mem
return tmp
```

In atomic compare and exchange, since the compare and exchange is done atomically the window where other thread can enter during execution of current thread is the time between getting the shared variable and calling

atomic compare and exchange operation. Since the window is very short for other thread to modify the data of shared variable, the INCONSISTENCY count is very less compare the non atomic.

And Consistency can be achieved by checking the return value and redo the iteration again.

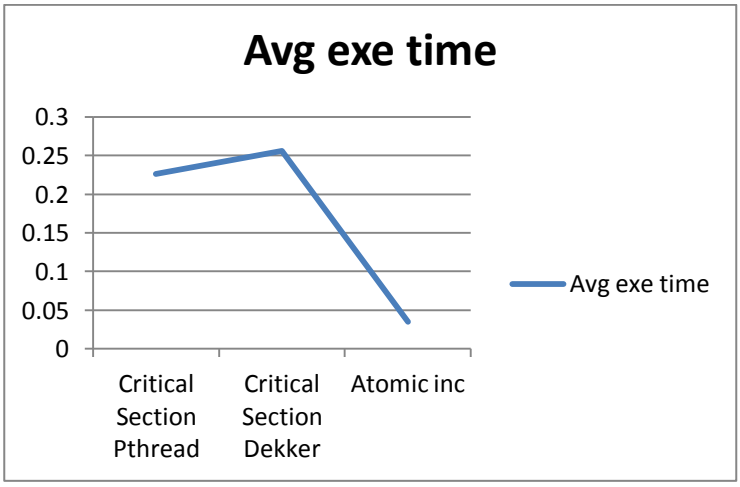
```
for (int i = 0; i < iterations; i++) {
    int32_t a,ret;
    a = *data;
    ret =
asm_atomic_cmpxchg_int32((int32_t
*)data, a, (int32_t)(a+1));
    if (ret != a) i--;
}
```

```
[lnch@subway lab1_5]$ ./lab1.5 -t cmpxchg_atomic
Configuration:
        Test implementation: cmpxchg_atomic
        Critical sections implementation: pthreads
        Iterations: 1000000
Statistics:
        Thread 0: 0.0533 s (1.8757e+07 iterations/s)
        Thread 1: 0.0612 s (1.6329e+07 iterations/s)
        Average execution time: 0.0573 s
        Avergage iterations/second: 1.7459e+07
NO INCONSISTENCY after 1000000 iterations
```

7. Performance comparison of critical section and atomic increment and decrement:

Compare to critical section implementation, Atomic increment and decrement is 7 times lesser.

Type	Avg exe time
Critical Section Pthread	0.2263
Critical Section Dekker	0.2556
Atomic inc/dec	0.0345



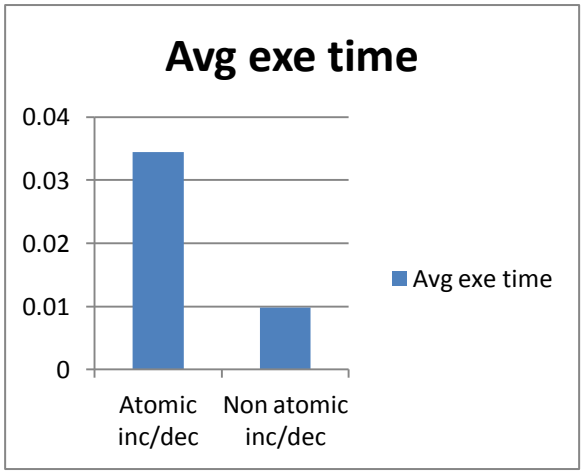
Reason:

1. In Critical section implementation, the number of instructions is high when compare to atomic increment. Because, the instructions to get, execute and release critical section is more. Whereas atomic inc/dec is just one assemble instruction call, and which in turn end up with very less opcode for execution, where the critical section causes more execution compare to Critical section implementation.
2. Most of the architecture supports atomic inc/dec in hardware level. That would further increase the performance.

8. Comparison between atomic and non atomic increment and decrement:

Non atomic increment and decrement is more than three times faster than atomic increment and decrement.

Type	Avg exe time
Atomic inc/dec	0.0345
Non atomic inc/dec	0.0098



Reason:

Implementing atomic property adds overhead on memory access and thread synchronisation. Due to this overhead, the performance of atomic increment

and decrement is comparatively lesser than Non atomic increment and decrement.

5.6. Open ended portion:

Critical section implantation using queue algorithm:

Acquire:

Step 1: "I" and "P" are the data element of a thread which points to a memory location.

Step 1:
I -> Location l1,
P-> Location l1

Step2: Make the location l1 active and swap P with lock L

Step 2
l1 -> set,
Swap (L, P)

Step3: If P is free, then acquire the critical section, or wait until it gets free.

Step 3:
If P-> Free, then get CS
Else, wait till P gets Free

Release:

Reset I, and make P points to I

Thus with this implementation, the lock acquire and release happen in queue (FIFO) format.

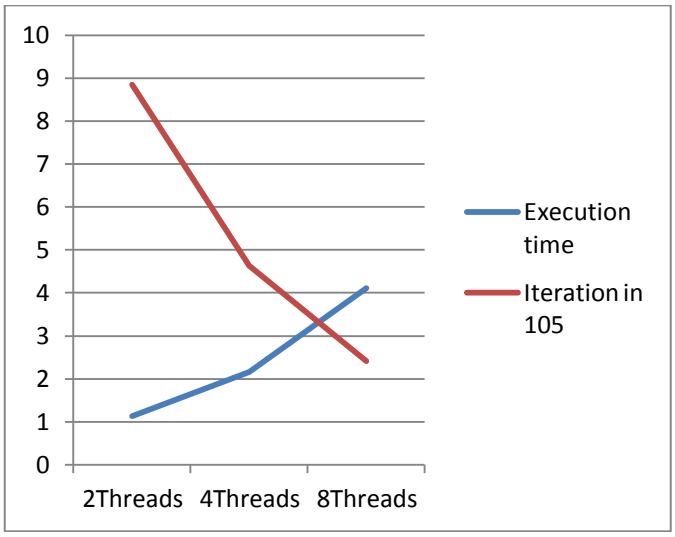
Swap is implemented by asm_atomic_xchg_voidp call.

Output:

```
[lnch@subway lab1_5]$ ./lab1.5 -t critical -c queue
Configuration:
        Test implementation: critical
        Critical sections implementation: queue
        Iterations: 1000000
Statistics:
        Thread 0: 1.1288 s (8.8591e+05 iterations/s)
        Thread 1: 1.1288 s (8.8589e+05 iterations/s)
        Average execution time: 1.1288 s
        Avergage iterations/second: 8.8590e+05
NO INCONSISTENCY after 1000000 iterations
```

Compare to other critical section implementation, this CS_queue algorithm takes more time, since every iteration, each thread does the atomic swap operation irrespective of lock's availability.

Threads	Execution time	Iteration in 10^5
2Threads	1.1288	8.85
4Threads	2.1564	4.63
8Threads	4.1187	2.42



On increase of threads, the execution time increases twice, since the order of the lock release is preserved, and also, there is an additional overhead in atomic swap instruction.