Louis DURAND
EIT Digital Master School – Embedded Systems – Year 1

lcdurand@kth.se

COMPUTER SYSTEMS ARCHITECTURE

LAB1 REPORT

3.2 COLLECTING STATISTICS FROM A SIMPLE CACHE

By running the following command, we can collect these ratios for every benchmark.

```
./simics -stall -c 'benchmark_at_magic_breakpoint.conf' -x lab1_3-2.simics
```

The "conf" file is a checkpoint where the chosen benchmark is paused at its magic breakpoint. The simics script configures the cache and run the requested commands as presented in the lab manual.

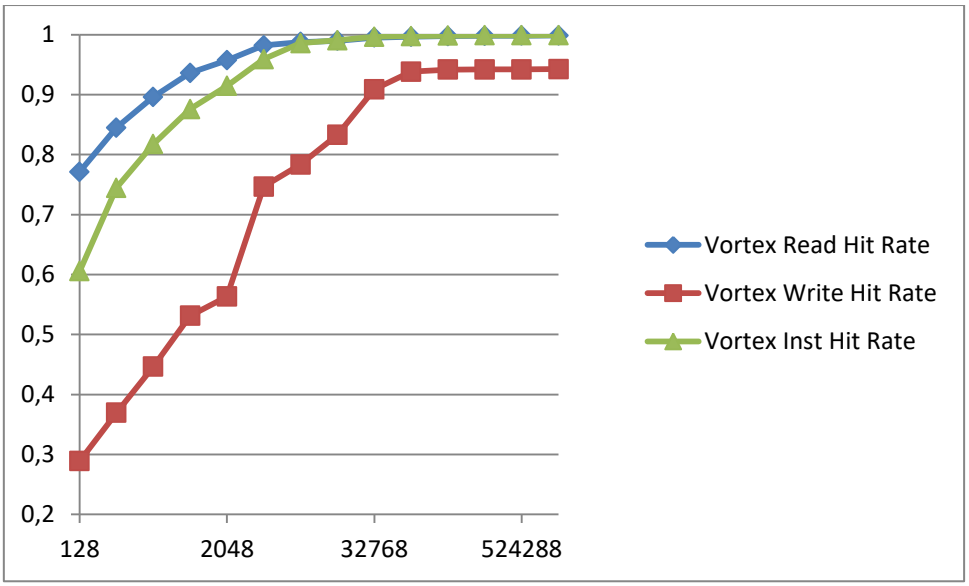
Access Type	Cache Hit Rate
Vortex	
Data Read	77.11 %
Instruction Fetch	60.55 %
Data Write	28.90 %
Quake	
Data Read	86.74 %
Instruction Fetch	79.17 %
Data Write	82.54 %
Parser	
Data Read	87.20 %
Instruction Fetch	92.71 %
Data Write	60.63 %

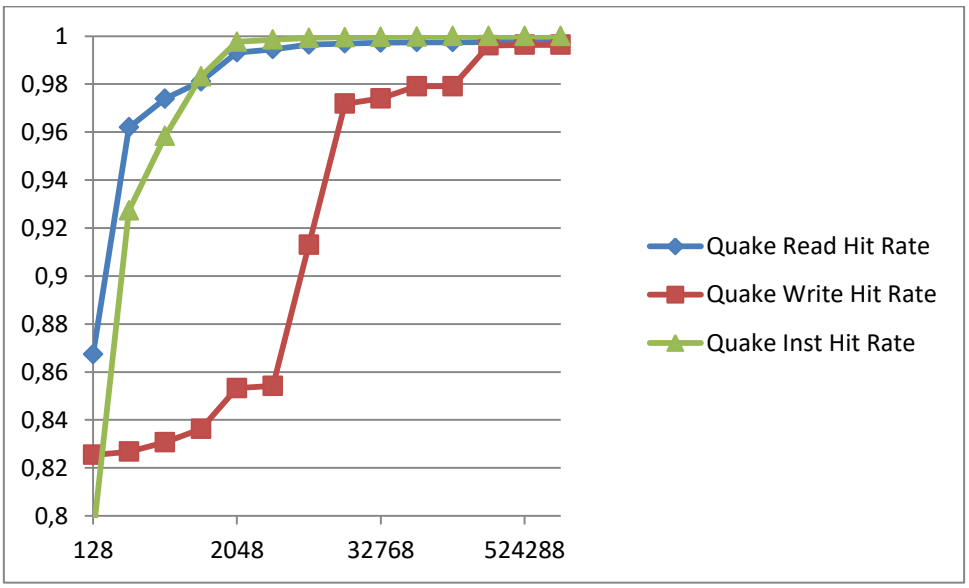
Table 1: Hit ratios for the three benchmarks, using the simple unified cache.

The Parser benchmark has the highest hit rate for data read and instruction fetch while the Quake benchmark has significantly better results for data write. The Vortex benchmark has the worst cache performance on all three access types.

3.3 DETERMINING BENCHMARK WORKING SET SIZE

In order to answer this question, I used a script that doubled the cache size ranging from 128 lines up to 1 048 576 = 2^{20} lines, and executed the benchmark procedure for each value to measure the hit rates. I built a csv file with the different hit rates that I could plot in order to approximately infer the working set size of each benchmark. The three following plots represent the hit rates (Y axis) as a function of the number of lines in the cache (X axis), represented on a binary logarithmic scale.





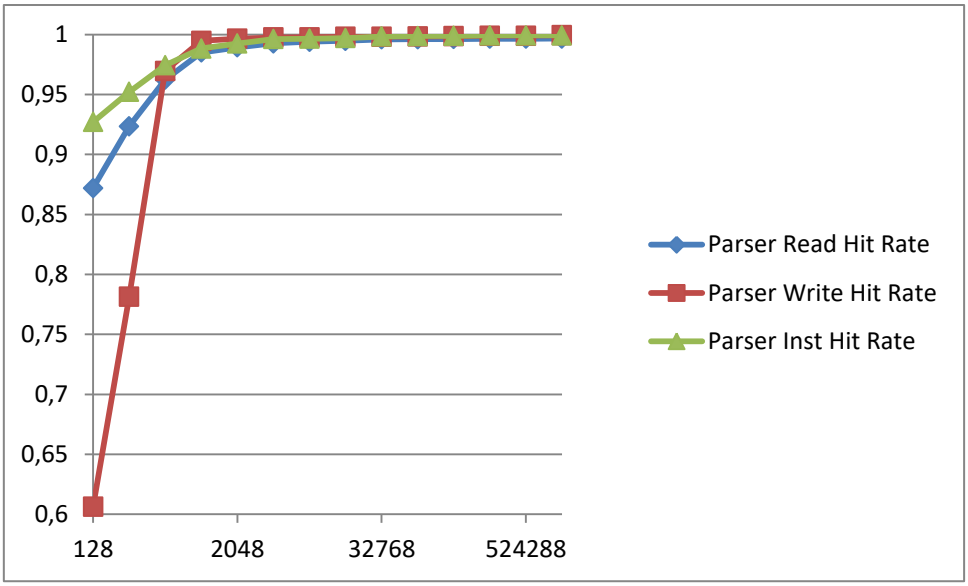


Figure 1: Hit ratios as a function of the cache's number of lines for the three benchmarks.

As we increase the cache size, the number of conflict and capacity misses decreases. At some point, the curve becomes steady. **This point gives us a good approximation of the program's working set size, including data *and* instructions.** Sometimes the curve stabilizes at a lower value than 100%. Since the cache has been warmed up, there should be very few cold misses, so the remaining misses might be conflict misses due to a poor choice of addresses for the data. It would have been more accurate to use a fully associative cache with LRU policy to determine the working set size.

From the graph, by noting the maximum value of the "steady point" of both read and write hit rate curves, we get these working set sizes for the **1 000 000 instructions** execution timespan:

Vortex: 65536 lines, so about **2 MiB**

Quake: 262144 lines, so about **8 MiB**

Parser: 1024 lines, so about **32 kiB**

3.4 Minimal instruction and data caches

By running the following command, we can collect these ratios for every benchmark.

```
./simics -stall -c 'benchmark_at_magic_breakpoint.conf' -x lab1_3-4.simics
```

Access Type	Cache Hit Rate (32 bytes line)	Cache Hit Rate (128 bytes line)
Vortex		
Data Read	21.23 %	24.77 %
Instruction Fetch	36.54 %	62.79 %
Data Write	4.00 %	5.30 %

Quake		
Data Read	27.19 %	28.78 %
Instruction Fetch	29.31 %	56.18 %
Data Write	57.17 %	58.89 %
Parser		
Data Read	59.67 %	70.00 %
Instruction Fetch	33.05 %	59.10 %
Data Write	1.21 %	1.41 %

Table 2: Hit ratios for the three benchmarks, using the split cache with 32 or 128 bytes line size.

For all three benchmarks, we can see that the instruction fetches have quite similar hit rates of around 33% for 32 bytes lines, and 60 % for 128 bytes lines, proving that spatial locality is a particularly relevant concept for instructions.

Concerning data accesses, we get more diverse results depending on the program. They probably iterate through their data in different ways that are more or less in agreement with the spatial locality assumption (Or the steps are too large for the line's size).

Increasing the line size doesn't change much the data hit rates (only increasing it of less than 3%) except for the Parser where the data read hit rate has been increased of 10%.

However, increasing the line size almost doubles the hit rate for instruction fetches for all three benchmarks with a hit rate of around 60 %.

3.5 COLLECTING STATISTICS FROM A CACHE HIERARCHY

By running the following command, we can collect these ratios for every benchmark.

```
./simics -stall -c 'benchmark_at_magic_breakpoint.conf' -x lab1_3-5.simics
```

Access Type	**L1 Cache Hit Rate**	**L2 Cache Hit Rate**	**Global Hit Rate**
Vortex			
Data Read	90.55 %	99.25 %	99,93 %
Instruction Fetch	82.23 %	99.82 %	99,97 %
Data Write	53.69 %	99.92 %	99,96 %
Quake			
Data Read	98.73 %	90.79 %	99,88 %
Instruction Fetch	89.01 %	100.00 %	100,00 %
Data Write	90.62 %	99.93 %	99,99 %
Parser			
Data Read	95.79 %	99.57 %	99,98 %
Instruction Fetch	98.64 %	100.00 %	100,00 %

Data Write	84.96 %	100.00 %	100,00 %

Table 3: Hit ratios for the three benchmarks, within the L1 and L2 caches, and for the global cache.

We can see that once the caches are warm (measurements are taken after 100 000 000 instructions for warm up), they can contain almost the complete working set. The L2 cache hit rates are very high for most of the cases (99-100%) except for the Quake benchmark where the Data Read case has a L2 hit rate of only 90.79 %. However, the L1 hit rate for the Data Read accesses in the Quake benchmark is the highest with 98.73 % which means that in absolute, there are not so much misses at the L2 level causing a memory access: 147 over a total of 125752 data read transactions, which results as a comfortable 99.88 % global hit rate. The last column of the table represents the global hit rates of the complex cache. They are all very close to 100 %.

Now we are going to change the size of the L2 cache (number of lines: 64, 4096, 8192) and record the hit rates for the Vortex benchmark.

Access Type	**L2 Cache Hit Rate (8 kiB)**	**L2 Cache Hit Rate (original 512 kiB)**	**L2 Cache Hit Rate (1 MiB)**
Vortex			
Data Read	26.32 %	99.25 %	99.87 %
Instruction Fetch	40.60 %	99.82 %	100.00 %
Data Write	96.06 %	99.92 %	99.94 %

Table 4: Hit ratios of the L2 cache with different sizes, for the Vortex benchmark.

As we have seen in 3.3, the augmentation of a cache's size increases the hit rate but of course not proportionally. There is a point when increasing the size doesn't improve much the hit rate anymore. We can see here that a L2 cache of 8 kiB has decent results (especially for Data Write transactions) but quite poor compared to the performances of the L1 caches that have a similar size. Therefore it seems that the L2 cache (especially since it is a unified cache) has to be much larger than the L1 caches in order to get similar results. The original size of 512 kiB provides really good results of 99-100% hit rates, and doubling the size of it only improves slightly the results, so it is probably not worth the price.

Assuming that the time penalties cumulate along the memory hierarchy, the Average Memory Access Times below have been calculated using this formula:

$$t_{mem} = \sum_{l=1}^{3} \left(t_l * \prod_{k=1}^{l-1} (1 - h_k) \right) = t_1 + t_2 * (1 - h_1) + t_3 * (1 - h_1) * (1 - h_2)$$

Where:

- $(t_i)_{i \in [\![1,3]\!]}$ being the time penalties of each level, so $t_1 = 3$, $t_2 = 10$ and $t_3 = 200$.
- $(h_i)_{i \in [\![1,2]\!]}$ being the hit rates of each level.

I decided to apply the formula for each request type.

Access Type	8 kiB L2 Cache	512 kiB L2 Cache	1 MiB L2 Cache
Vortex			
Data Read	18.55	4.09	3.97
Instruction Fetch	18.28	4.84	4.78
Data Write	8.95	7.71	7.51
Any	11.98	5.21	5.12

Table 5: Average memory access times (in cycles) for the three architectures with the Vortex benchmark.

4.4 MULTITHREADED PERFORMANCE STUDY

Here are the data collected from the cache hierarchy with the following command for the PDES benchmark. The script automatically computes the average data from the eight L1 caches.

```
./simics -stall -c 'pdes_benchmark_at_magic_breakpoint.conf' -x lab1_4-4.simics
```

Stat value	8 kiB L1 caches	64 kiB L1 caches
L1 private caches (averages)		
Data Read Hit Rate	65.28 %	78.75 %
Instr. Fetch Hit Rate	88.93 %	99.24 %
Data Write Hit Rate	78.43 %	88.80 %
MESI Excl. to Shared	29279	40369
MESI Modif. to Shared	0	0
MESI Invalidates	58603	139334
L2 shared cache		
Data Read Hit Rate	99.58 %	99.59 %
Instr. Fetch Hit Rate	99.85 %	98.05 %
Data Write Hit Rate	96.56 %	95.34 %
Total Transactions	39 887 714	22 365 547

Table 6: Hit ratios for the two levels of caches and MESI statistics for the eight L1 caches.

When the L1 caches are increased in size, their hit rates increase of about 10 to 13 units. The L2 shared cache keeps similar hit rates although slightly lesser for instruction fetch and data write. The reason might be that new hits in the L1 cache were probably hits for the L2 cache in the 8kiB case.

Indeed, for instruction fetches, there were 15 004 misses in the L2 cache for 9 957 722 instruction fetch transactions in the case of 8 kiB L1 caches, and 14 327 misses for 735 805 transactions in the case of 64 kiB L1 caches. So the absolute number of misses is quite similar,

and even lower, even though the ratio is slightly worse. This must be a common repercussion on upper level caches when lower level caches are improved.

The average MESI statistics of the L1 caches have drastically changed. Exclusive to Shared transitions have increased of about 38 %, while the number of Invalid blocks have increased of 138 %. Indeed, the augmentation of size increases chances of duplicated data among the private caches, hence augmenting the number of Shared blocks. As a result, a larger amount of Shared blocks implies a higher amount of Invalidates in case of write operations.

Now we can wonder: why the increase of Invalidates is of another order of magnitude than the increase in Shared blocks? My guess is that not only the number of Exclusive blocks becoming Shared increases, but also the number of duplicates of them. So maybe some block that used to be shared in only two caches of the first case, might get shared between more caches in the 64 kiB case, therefore increasing the number of Invalidates when a write occurs.

However, there are still no transitions from Modified to Shared. This means that probably, in the program, a block that has been written to is never requested afterwards by another processor before that the modified cache block is replaced – and thus written back to memory.

5.5 MULTIPROCESSORS AND MEMORY ORDERING: DIRECTED PORTION

1. The result of the counter in the end of the 1 000 000 iterations is a seemingly random number, different of 0. This reveals that the read-and-update concurrent operations on the shared counter aren't done in an atomic manner.
2. Indeed in the source code of the `increment` and `decrement` functions, there is no call to any lock mechanism. I added a call to `enter_critical` before the read of the counter and a call to `exit_critical` after the store of its modified value. Now, using pthreads algorithm, the consistency is preserved! However, we can observe an execution time about 10 times superior than when no mutual exclusion is enforced since the mutual exclusion create stalls in the execution.
3. a) The volatile keyword indicates to the compiler that no optimization should be performed on the variable (simplification or coalescing that would be allowed by the program flow). It is necessary when external threads may access and modify this variable – which cannot be seen in the program flow by the compiler – which is exactly the fate of the `flag` and `turn` variables.
   b) With a straightforward implementation of Dekker's algorithm, the counter is closer to 0 in average than without any lock mechanism, but still inconsistent. Sometimes, the program even gets stuck into a deadlock. This is probably related to the of out-of-order execution on x86 machines: the algorithm doesn't work properly without memory barriers to synchronize memory reads before critical sections.
4. I had to add three MFENCE instructions: one before and after the main while loop of the enter_critical function, and one in the beginning of the exit_critical function in order to guarantee the consistency. Note that the execution time of my Dekker algorithm is about 2 times longer than the *pthreads*.

5. The use of non-atomic instructions results in an inconsistent final number. Although we are using the single assembly instructions `incl` and `decl`, this doesn't make them atomic: because of pipeline execution, it is possible that another processor is fetching the counter's value between the memory fetch stage of the instruction and the memory write stage. Using *atomic* assembly instructions – `incl` or `decl` preceded with `lock` – preserves the consistency. I tried to understand how such a mechanism could be implemented. According to *THE INTEL® 64 AND IA-32 ARCHITECTURES SOFTWARE DEVELOPER'S MANUAL, VOLUME 3A: SYSTEM PROGRAMMING GUIDE, PART 1* [1], the lock mechanism asserts on the memory bus that the specific memory address in use should be locked during the execution of the instruction, preventing concurrency problems. Some more recent processors also make use of the cache coherency mechanism to avoid locking the memory bus. Chapter 8.1.4:
   For the Intel486 and Pentium processors, the LOCK# signal is always asserted on the bus during a LOCK operation, even if the area of memory being locked is cached in the processor.
   For the P6 and more recent processor families, if the area of memory being locked during a LOCK operation is cached in the processor that is performing the LOCK operation as write-back memory and is completely contained in a cache line, the processor may not assert the LOCK# signal on the bus. Instead, it will modify the memory location internally and allow it's cache coherency mechanism to ensure that the operation is carried out atomically. This operation is called "cache locking." The cache coherency mechanism automatically prevents two or more processors that have cached the same area of memory from simultaneously modifying data in that area.
   Once again, the atomic instructions are slower to execute: about 4 times slower than non-atomic instructions. But we have better timing than when using critical sections!
6. The use of non-atomic instructions here also results in inconsistencies for the same reasons. The use of the atomic version – `cmpxchg` preceded with `lock` – preserves the consistency, using the same mechanism as mentioned in the previous question. The performance of this version is almost half of the atomic increment, but still faster than critical sections. The asset of this instruction is that it might enable us to make more complex assignments in an atomic manner.
7. As briefly mentioned in 5, the atomic increments/decrements are faster than using critical sections. If we run the `incdec_atomic` test, we get an average performance of about 2.5e+7 iterations/second, while the `critical` test with the `pthreads` algorithm gives an average performance of about 2.5e+6 iterations/second. Using atomic instructions to solve racing problems is therefore about ten times faster than using critical sections because it performs a lock at the processor level instead of software level. The lock operates only during one specific instruction instead of several instructions, reducing the stalls for waiting processes. Using atomic instructions should always be preferred when possible. However, when more complex manipulations than a simple increment must be done in an atomic manner, then we might have no other choice than using higher level locking mechanisms like critical sections.
8. Atomic increments/decrements perform at about 2.5e+7 iterations/second while non-atomic instructions give around 1.0e+8 iterations/second. As mentioned in 5, non-atomic instructions are indeed about four times faster than atomic instructions. The lock mechanism of x86 architectures performs a lock on the memory bus, as explained in 5, creating stalls for other processes that request the same memory address. This results in a necessary loss in performance, in the benefit of correctness.

5.6 MULTIPROCESSORS AND MEMORY ORDERING: OPEN ENDED PORTION

I researched about the CLH queue locking algorithm and inspired from a lecture [2] to implement the `lh_acquire` and `lh_realease` functions.

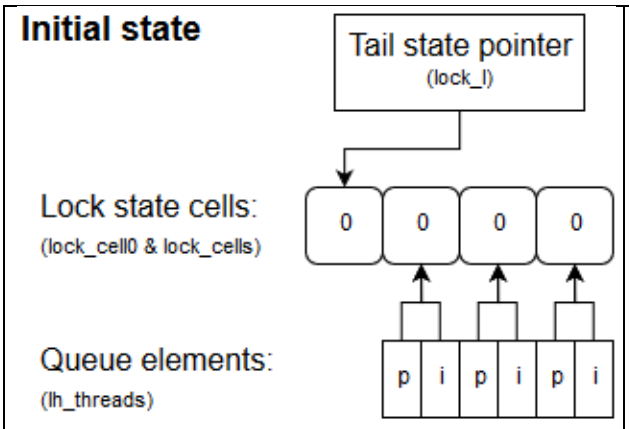
Here is my implementation:

```
/**
* Acquire a lock using the CLH locking algorithm.
* \param l Double pointer to the "state" of the tail element of the queue
* \param i Double pointer to the "state" of the current element
* \param p Double pointer to the "state" of the previous element
*/
static void
lh_acquire(int ** volatile l, int ** volatile i, volatile int ** volatile p) {
        assert (**i == 0);
        assert (*i == *p);

        // Current element's state <- true ⇔ Successor must wait
        **i = 1;

        // *p <- *l and *l <- *i: the tail points to the current element's state
        // and the previous element's state is defined as the state of the former
        // tail element. This must be done atomically!
        *p = (int *) asm_atomic_xchg_voidp((void **) l, (void *) *i);

        // Waiting for the previous element's state to pass to false.
        while(**p) {}
}

/**
* Release a lock using the CLH locking algorithm.
* \param i Double pointer to the "state" of the current element
* \param p Double pointer to the "state" of the previous element
*/
static void
lh_release(int **i, int **p) {
        assert (**i != 0);
        assert (*i != *p);

        // Current element's state <- false ⇔ Successor can exit the while loop in
        // lh_acquire
        **i = 0;

        // *i and *p must point to a same available "state cell".
        *i = *p;
}
```

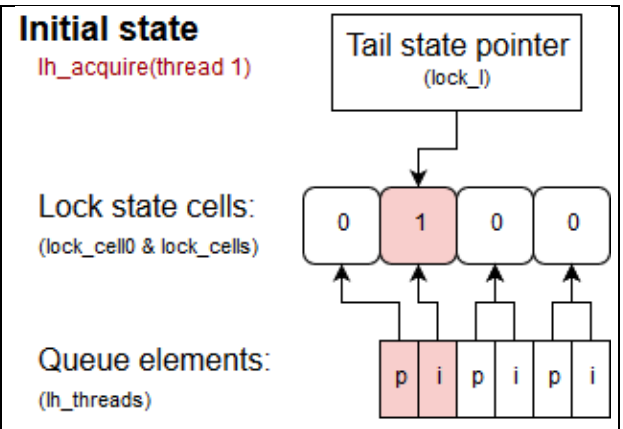
The skeleton of the algorithm was a bit (unnecessarily) difficult to comprehend since it uses direct pointers to the data instead of the structures; and the variables were not very documented and badly named.

The code above is documented and commented in order to explain every step.

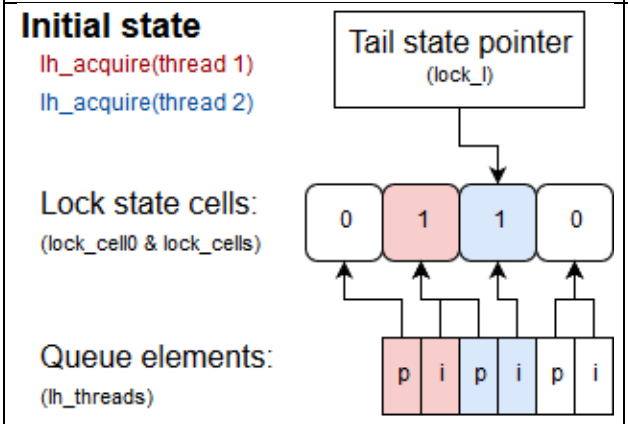
I made the following diagrams to represent how the algorithm works through an example:



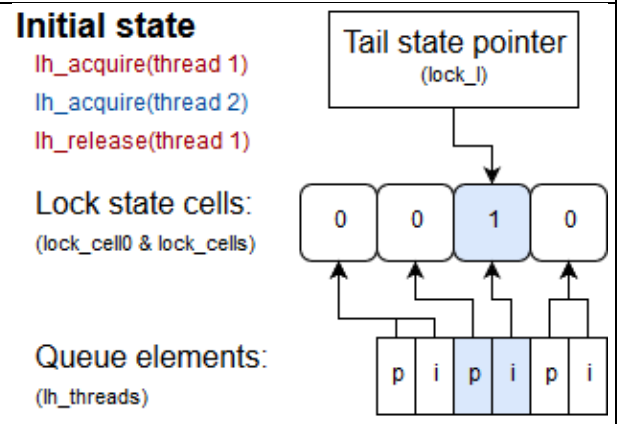
Step 0: Initial configuration.



Step 1: Thread 1 acquires the lock and enters its Critical Section.



*Step 2: Thread 2 requests the lock and must wait since *p==1.*



*Step 3: Thread 1 releases the lock; Thread 2 can enter its Critical Section since *p==0.*

Performance analysis:

The table below lists the performance in increments/second of the CLH Queue implementation, compared with the performance of the pThreads algorithm, for 2, 4 and 8 racing threads.

Number of threads	CLH Queue	pThreads
2 Threads	1.1 e+6	3.5 e+6
4 Threads	5.3 e+5	1.8 e+6
8 Threads	2.7 e+5	1.1 e+6

Table 7: Performance of the CLH Queue and pThreads algorithms in increments/second.

The pThreads algorithm performs 3 to 4 times better than the CLH Queue. Also the performance of the CLH Queue seems to be more affected by the number of threads: the ratio between 2-threads and 8-threads performance is 4.07 for the CLH Queue, and only 3.18 for pThreads.

Also, it is interesting to see that the execution time of each thread is almost exactly the same when using the CLH Queue, whereas it can differ from about 14 % between the different threads when using the pThreads algorithm. This is probably due to the FIFO behavior of the CLH queue that has the asset of avoiding starvation: the requests are served in the order of their arrival.

REFERENCES:

[1] The Intel® 64 And IA-32 Architectures Software Developer's Manual, Volume 3a: System Programming Guide, Part 1. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

[2] Mutual Exclusion: Classical Algorithms for Locks, Bill Scherer, Rice University, page 45. Available: https://www.cs.rice.edu/~vs3/comp422/lecture-notes/comp422-lec19-s08-v1.pdf