Evaluation of the Impact of Cache Coherence Protocol and Data Locality on the Efficiency of Atomic Operations on Multicore Processors*

Evgeniy Goncharenko[0000−0002−6203−8034] and Alexey Paznikov[0000−0002−3735−6882]

Saint Petersburg Electrotechnical University "LETI", 5 Professora Popova Str., Saint Petersburg 197022, Russia kesvesna@rambler.ru, apaznikov@gmail.com

Abstract. In this work, we analyze the efficiency of atomic operations compare-and-swap (CAS), fetch-and-add (FAA), swap (SWP), load and store on modern multicore processors. These operations implemented in hardware as processor instructions are highly demanded in multithreaded programming (design of thread locks and non-blocking data structures). In this article we study the influence of cache coherence protocol, size and locality of the data on the latency of the operations. We developed a benchmark for analyzing the dependencies of throughput and latency on these parameters. We present the results of the evaluation of the efficiency of atomic operations on modern x86-64 processors and give recommendations for the optimizations. Particularly we found atomic operations, which have minimum (load), maximum ("successful CAS", store) and comparable ("unsuccessful CAS", FAA, SWP) latency. We showed that the choice of a processor core to perform the operation and the state of cache-line impact on the latency at average 1.5 and 1.3 times respectively. The suboptimal choice of the parameters may increase the throughput of atomic operations from 1.1 to 7.2 times. Our evidences may be used in the design of new and optimization of existing concurrent data structures and synchronization primitives.

Keywords: Atomic operations · Atomics · Multithreading · Multicore · Cache coherence.

1 Introduction

1.1 A Subsection Sample

Multicore shared-memory computer systems (CS) include desktop and server systems as well as computer nodes within distributed CS (cluster systems, mas-

* The reported study was funded by RFBR according to the research projects No 19-07-00784, 18-57-34001 and was supported by Russian Federation President Council on Grants for governmental support for young Russian scientists (project SP-4971.2018.5).

sively parallel systems, supercomputers). Such systems may include tens and hundreds of processor of processor cores. For example, a computer node of Summit supercomputer (first place in TOP500 supercomputer ranking, more than 2 million processor cores, 4608 nodes) include two 24-core universal processor IBM Power9 and six graphical accelerators NVIDIA Tesla V100 (640 cores). Sunway TaihuLight (more than 10 million processor cores, third place in TOP500) is equipped with 40960 Sunway SW26010 processors with 260 cores. Processor cores in such systems are connected via processor interconnects (Intel QPI, AMD HyperTransport) according to the NUMA architecture. Notice, that cache memory in such systems is organized in very complicated ways, including different policies (inclusive, exclusive cache) and coherence protocols (MESI, MOESI, MESIF, MOWESI, MERSI, Dragon).

One of the most relevant tasks in parallel programming is the design of efficient tools for parallel thread synchronization. The main synchronization methods are locks, non-blocking (lock-free, wait-free, obstruction-free) concurrent (thread-safe) data structures and transactional memory [1–8]. Atomic operations (atomic) are used for implementation of all synchronization methods. The operation is atomic, if it's performed in one undividable step relative to other threads. In other words, no thread can observe this operation as "partially completed". If two or more threads perform operations with a shared variable and at least one of them writes (stores) to it, then the threads must use atomic operations to avoid data races.

The most common atomic operations are compare-and-swap (CAS), fetch-and-add (FAA), swap (SWP), load (read), store (write). Load(m, r) reads the variable's value from the memory address m to the processor register r. Store(m, r) writes the value of the variable from the processor register r to the memory address m. FAA(m, r_1, r_2) increments (or decrements) by register's value r_1 the value of variable in memory address m and returns the previous value in the register r_2. SWP(m, r) exchanges the values of the memory address m and register r. CAS(m, r_1, r_2) compares the memory value m with the register r_1; if the values are equal, modify memory m to a new value r_2. Designing parallel programs, we should consider the impact to the atomic operation efficiency such aspects as cache coherence protocol, buffer size, thread number, data locality.

Despite the widespread use, the efficiency of atomic operations has not been adequately analyzed at the current moment. For example, some works still claim that CAS is slower than FAA [9] and its semantics cause the "wasted work" [10, 11], since unsuccessful comparisons of the data in memory and in the register lead the additional load to the processor core. The works [12] states that the performance of atomic operations is similar on multi-socket systems because of the overheads from the hops between the sockets. The paper [11] analyzes the performance of atomic operations on graphical processors, but currently the urgent problem is the evaluation of the costs of atomic operations on universal processors. The work [13] considers the efficiency of atomic operations CAS, FAA, SWP for analyzing the impact of dynamical parameters of a program to the atomic operation execution time. The results show undocumented properties

of the investigated systems and the evaluation of the atomic operation execution time. However, the experiments have been carried out with fixed processor core frequency, with huge pages activated and disabled prefetching. We suppose that these conditions distort simulation results for real programs. In addition, the operations load and store have not been examined.

In this work we consider operations CAS, FAA, SWP, load, store, wherein the conditions of the experiments are as close as possible to the real conditions of parallel program execution. In particular, the core frequency is not fixed, huge pages are disabled, and cache prefetching is enabled. We evaluate the efficiency of atomic operations for modern x86-64 processor architectures depending on the cache line state (in cache coherence protocol), buffer size and its locality (relative to the core, executed the operations). Besides, we consider different variants of CAS operation (successful and unsuccessful).

2 Design of benchmarks

As a metrics we used latency l of atomic operation execution and throughput b. Throughput $b = n/t$, where n is the number of executed operations of sequential access to the buffer's cells in time t.

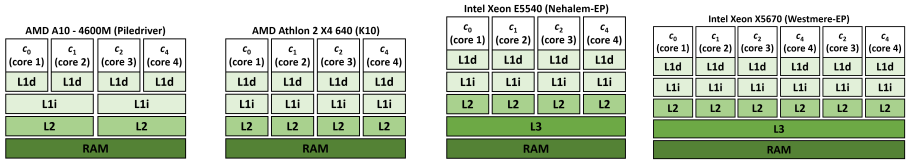
To measure the execution time, we used the instruction set RDTSCP. To avoid reordering while executing operations we used full memory barriers.

We investigated the influence of cache line state (within the cache coherence protocol: M, E, S, O, I, F) to the atomic operation efficiency. Define the basic states of cache lines. M (Modified) – cache-line is modified and contains actual data. O (Owned) – cache line contains actual data and is the only owner of this data. E (Exclusive) – cache-line contains actual data, which is equal to the memory state. S (Shared) – cache-line contains actual data and other processor cores have actual copies of this data at the same time. F (Forwarded) – cache-line contains the most actual correct data and other cores may have copies of this data in shared state. I (Invalid) – cache-line contains invalid data.

We designed a benchmark. In this benchmark we allocate integer buffer q of size $s = 128$ MiB. The data is placed in the cache-memory and cache-lines are translated to a given state of the cache coherence protocol. Notice, that we don't use virtual memory. All the data was unaligned. To set state M for cache-lines we write arbitrary values to the buffer's cells. To set state E we write arbitrary values to the buffer's cells and then perform clflush instruction to set the state of cache-lines to I followed by reading the buffer's cells. To set the state S a processor core reads buffer's cells from the cache-lines with state E of another core. To set the state O a processor reads from the cache-lines with state M of another core's cache-memory (cache-lines are switched from M to O).

For the CAS operation we performed two experiments: for successful and unsuccessful operation. Unsuccessful CAS is such CAS, when $m \neq r_1$ (the memory is not changed). The deliberately unsuccessful execution of CAS is achieved by comparing the address of the pointer with the data on that pointer. For successful CAS $m = r_1$ (the memory is changed).

We conducted experiments with the next processors: AMD Athlon II X4 640 (microarchitecture K10), AMD A10-4600M (microarchitecture Piledriver) (cache coherence protocol MOESI [14]), Intel Xeon X5670 (microarchitecture Westmere-EP) and Intel Xeon E5540 (microarchitecture Nehalem-EP) (cache coherence protocol MESIF [15]). Cache line size if 64 bytes. As a compiler we used GCC 4.8.5, operating system is SUSE Linux Enterprise Server 12 SP3 (kernel 4.4.180).



(a) Piledriver (b) K10 (c) Nehalem-EP (d) Westmere-EP

Fig. 1: Target microarchitectures.

Next, we describe the main steps of the algorithm for measuring the execution time of atomic operations for state E (fig. 2a). Auxiliary function DoOper executes defined atomic operation for all the elements of the buffer. The main algorithm is performed until the buffer size reach the maximum value (line 1). Used range of test array $testSizeBuffer$ in the experiments is varied from 6 KiB (minimal size L1 cache) to 128 MiB (larger than L3 cache). Each experiment is executed $nruns$ times (line 2). Then we store arbitrary values to the buffer's cells to set cache-lines of the core c_0 to the state M (for the other cores the state is changed to I) (line 3). After that, we invalidate cache-lines (line 8) followed by the reading to set cache-lines of the core c_0 to the state E (line 5). On the next step for each variable we perform atomic operation (line 7). In the end we compute the operation execution time and total execution time (line 9), compute the latency (line 11), throughput (line 12) and increase the current buffer size by $step = L_x/8$, where L_x is the cache memory size for levels $x = 1, 2, 3$ (line 13).

Here are the main steps of the algorithm for time measurement of atomic operation execution by the core c0 for cache-lines in state S (fig. 2b). This algorithm is performed in two threads, affined to the cores c_0 and c_2. Synchronization is implemented by means of atomic flags. The first thread on the core c_0 writes arbitrary data to the buffer, set the state of cache-lines to M (at the same time for the other cores the state of these cache-lines is changed to I) (line 4). Then we clean cache-lines (set to I) (line 5) and read data for changing the state of cache-lines of c_0 to the E (line 6). The second thread (core c_2) reads the data, changes the cache-line state of c_0 to c_2 cores to S (line 8). Then the first thread on core c_0 implements atomic operation with the elements of a buffer (line 11). After that we get the time for operation execution and the metrics (lines 15-17).

```
Function DoExpExlusive(oper)
1  while d ≤ testSizeBuffer do
2     for i = 0 to nruns do
3        DoOper(store(1), buf)
4        CLFlush(buffer, d)
5        DoOper(load, buf)
6        start= GetTime()
7        DoOper(oper, buf))
8        end = GetTime()
9        sumTime = sumTime + (end–start)
10    end for
11    latency = sumTime/nruns/d
12    bandwidth = (d/sumTime/nruns) × 10^9/2^20
13    d = d + step
14 end while
```

(a) Exclusive state

```
Function DoExpShared(oper)
1  while d ≤ testSizeBuffer do
2     for i = 0 to nruns do
3        ▷ First thread on c0
4        DoOper(store(1), buf)
5        CLFlush(buffer, d)
6        DoOper(load, buf)

7        ▷ Second thread on c2:
8        DoOper(load, buf)

9        ▷ First thread on c0:
10       start =GetTime()
11       DoOper(oper, buf)
12       end = GetTime()
13       sumTime = sumTime + (end–start)
14    end for
15    latency = sumTime/nruns/d
16    bandwidth = (d/sumTime/nruns) × 10^9/2^20
17    d = d + step
18 end while
```

(b) Shared state

Fig. 2: Algorithm for measuring the latency and throughput of atomic operations SWP/FAA/CAS/Load/Store and throughput

The following are the main steps of the algorithm for measurement of execution time of atomic operations for the cores c_1 and c_2 in state S (fig. 3b). The algorithm is performed in three threads, affined to the cores c_0, c_1, $c2$. The first thread (core c_0) writes to the buffer to set the state of cache-line of c_0 to M (for the rest of the cores the state is changed to I) (line 4). Then cache-line is invalidated to the state I (line 5) followed by the reading data to set the state of c_0 to E (line 6). The second thread (core c_1) reads the data (the state of cache-lines in c_0 and c_1 is changed to S) (line 8). On the next step the third thread (core c_2) perform atomic operation with each element of the buffer (line 11). After that metrics are computed (lines 15-17).

For atomic operation latency measurement on the c_1 we use the similar algorithm, in which the steps for c_1 and c_2 cores are changed places with each other.

The following are the main steps for the algorithm for measurement of execution time of atomic operations for cache-lines in state O on the core c_0 (fig. 3a). The algorithm is performed in two threads, binded to the cores c_0 and c_2. The first thread (core c0) arbitrary data is written into the buffer to set the state of cache-lines of the core c_0 to M (for the rest cores the state is changed to I) (line 4). The second thread (core c_2) reads the data to change the state of cache-lines of the local core c_0 to O and cache-lines of the core c_2 to S (line 8). On the next step the first thread (core c_0) perform atomic operation for each variable in the buffer (line 9). After that we compute atomic operation execution time and the metrics (lines 13-15).

```
Function DoExpLocalityShared(oper)
1  while d ≤ testSizeBuffer do
2     for i = 0 to nruns do
3        ▷ First thread on c0:
4        DoOper(store(1), buf)
5        CLFLush(buf, d)
6        DoOper(load, buf)

7        ▷ Second thread on c1
8        DoOper(Load)

9        ▷ Third thread on c2:
10       start =GetTime()
11       DoOper(oper, buf))
12       end =GetTime()
13       sumTime = sumTime + (end–start)
14    end for
15    latency = sumTime/nruns/d
16    bandwidth = (d/sumTime/nruns) × 10^9/2^20
17    d = d + step
18 end while
```

(a) Exclusive state

```
Function DoExpLocalityOwned(oper)
1  while d ≤ testSizeBuffer do
2     for j = 0 to nruns do
3        ▷ First thread on c0:
4        DoOper(store(1), buf)

5        ▷ Second thread on c2:
6        DoOper(load, buf)

7        ▷ First thread on c0:
8        start =GetTime()
9        DoOper(oper, buf)
10       end =GetTime()
11       sumTime = sumTime + (end–start)
12    end for
13    latency = sumTime/nruns/d
14    bandwidth = (d/sumTime/nruns) × 10^9/2^20
15    d = d + step
16 end while
```

(b) Shared state

Fig. 3: Algorithm for measuring the latency and throughput of atomic operations SWP/FAA/CAS/Load/Store and throughput

3 Results and discussion

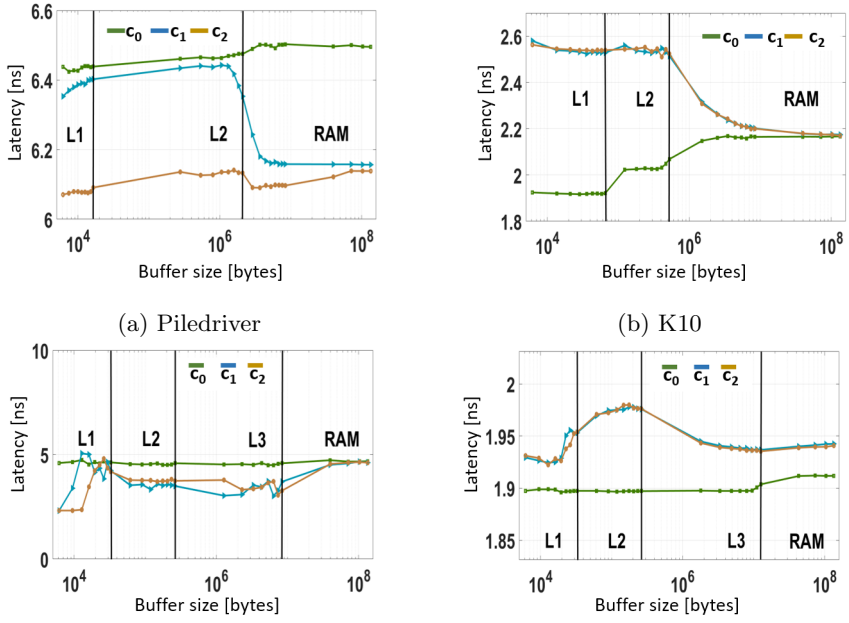
Fig. 4, 5 represent the experimental results for SWP operation. Latency on all processor except Nehalem-EP highly depends on the locality. Local core c_0 provides minimal latency especially for buffer size less than L2 cache. In shared state (Fig. 5), the latency grows when buffer size exceeds L2 cache and in most cases the impact of locality is insignificant on a large buffer size.

Figure 6 represents the latency of SWP operation on K10 processor for different cache-line states. It shows that state Modified gives minimal latency and Invalid state gives the maximum latency. Exclusive and Shared states are comparable with each other. Meanwhile, execution on local processor (c_1) provide substantially lower latency, compared with remote ones (c_2, c_3). With increasing buffer size, the effect of locality on latency decreases, because all the data is not cached and locates in main memory.

Fig. 7 depicts latency evaluations for different atomic operations in Shared state on K10 processor. As expected, Load has the minimal latency. Successful Compare-and-swap gives substantially larger latency compared with the other operations. Meanwhile, unsuccessful Compare-and-swap has the latency similar to FAA and SWP. Store operation also has larger latency. The similar results have been obtained for other processors.

Fig. 8 shows the results for the operation SWP, state O (Owned). The results are similar to the other states.

The tables 1 and 2 show suboptimal parameters of atomic operation execution on Westmere-EP and Nehalem-EP processors. Similar results have been obtained for the other processors. For example, on microarhictecures K10, Nehalem-EP, Westmere-EP operation load has the minimal latency for state S on the core



(a) Piledriver (b) K10

(c) Nehalem-EP (d) Westmere-EP

Fig. 4: Latency of atomic operation SWP for cache lines in the Exclusive state

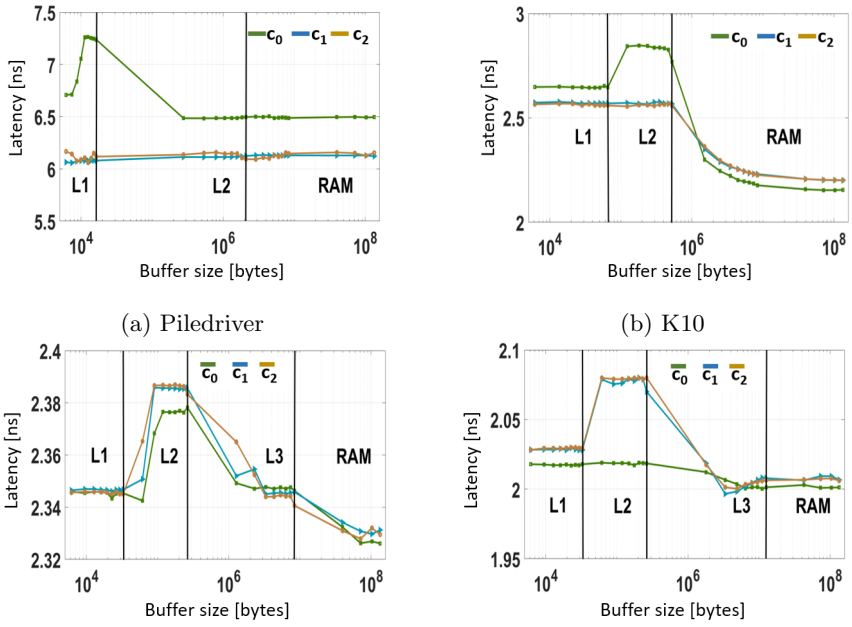
c_0 (from 1.14 to 1.81 ns), while for a Piledriver processor this operation has the lowest latency on c_0 and c_2 cores (from 1.8 to 2.4 ns).

Table 3 shows the ratio of maximum and minimum throughput.

Table 1: Suboptimal parameters for the architecture Westmere-EP.

Operation	**Cache-line state**	**Buffer size**	**Core**	**Shared memory level**	l, **ns**	b, **MiB/s**
Load	Exclusive	L2	c_0, c_1, c_2	L3	1.1 – 1.15	823 – 866
Store	Modified	RAM	c_0, c_1, c_2	L3	4.1 – 4.21	226 – 230
FAA	Invalid	L1	c_0, c_1, c_2	L3	1.89 – 1.91	490 – 499
SWP	Invalid, Modified	RAM	c_0, c_1, c_2	L3	1,93	493
unCAS	Exclusive	L1, L2	c_0, c_1, c_2	L3	2.55 – 2.59	367 – 372
CAS	Invalid	L2, L 3	c_0, c_1, c_2	L3	4.9 – 19.3	49 – 194

State of cache-lines is M. Metrics for atomic operations SWP, FAA, unCAS on the core c_0 for Piledriver architecture slightly varies with buffer resizing, meanwhile for the cores c_1 and c_2 the increasing of the buffer more than L2 leads the latency reduction down to minimum value for the both cores. For K10 at buffer size exceeding L2 latency of operations SWP, FAA, unCAS differs



(a) Piledriver (b) K10

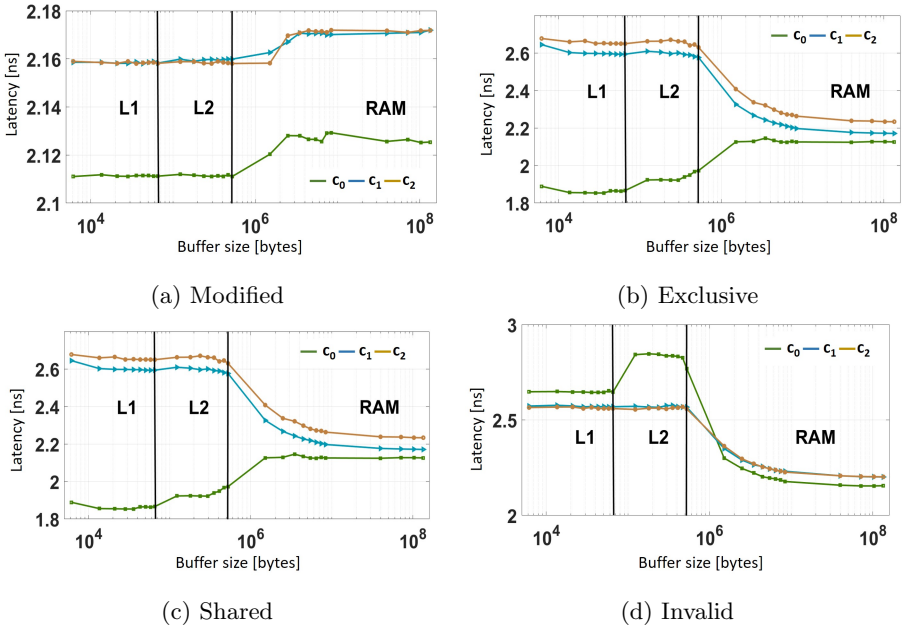
(c) Nehalem-EP (d) Westmere-EP

Fig. 5: Latency of atomic operation SWP for cache lines in the Shared state

slightly. Latency of load, SWP, CAS for Nehalem-EP latency is comparable for all cores for buffer size more than L3. There are similar observations for load, SWP for Westmere-EP. For other operations (store, CAS, FAA, unCAS) latency is the minimum for the core c_0 and doesn't depends on the buffer size.

State of cache-lines is E. Operations SWP, FAA, unCAS on Piledriver architectures and the buffer size more than L2 for c_1 and c_2 latency is comparable, for c_0 core we obtained maximum values at the same time. For K10 architecture latency of operations SWP, FAA, unCAS is similar for all the cores for buffer size exceeding L2. For Nehalem-EP latency of SWP, load is similar on all the cores for buffer size more than L3. For Westmere-EP minimum latency for load, SWP, FAA, CAS operations was obtained on the core c_0. For c_1 and c_2 and buffer size more than L3 latency varies slightly. State of cache-lines is S. Operations SWP, FAA on Piledriver architecture for buffer size L1 latency is maximum on the core c_0. On K10 for buffer size L2 maximum latency was obtained on the core c0 for SWP, FAA, store operations. For Nehalem-EP and Westmere-EP for buffer size L2 latency of SWP, FAA is maximum on c_1 and c_2 cores.

State of cache-lines is I. Operations SWP, FAA, unCAS on Piledriver architecture buffer size the buffer size does not significantly affect the runtime for all cores; the lowest latency was obtained on c_1 and c_2 cores. Load and SWP operations on K10 at buffer size more than L2 latency is minimal for c_0 core. For Nehalem-EP latency of SWP, load is similar for all the cores at buffer size more



(a) Modified

(b) Exclusive

(c) Shared

(d) Invalid

Fig. 6: Latency of atomic operation SWP for K10 processor core for different cache-line states

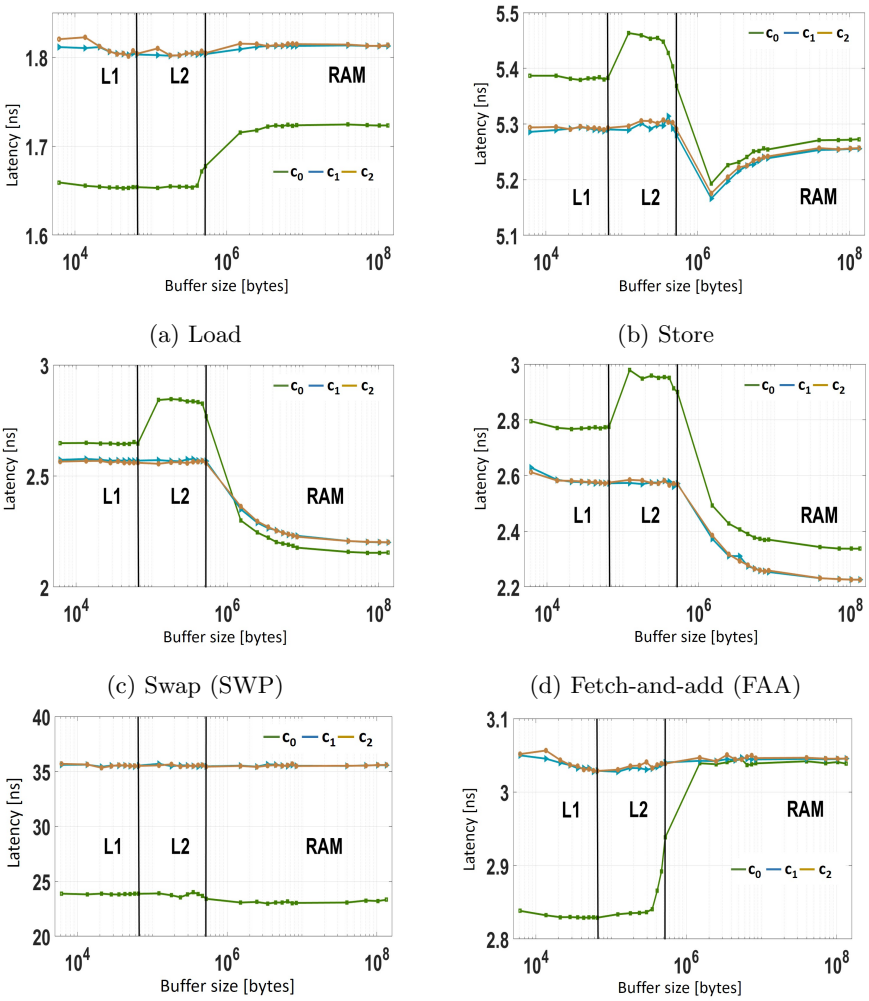
than L3. For operations load, SWP, FAA, CAS on Westmere-EP the minimal latency was obtained on the core c_0.

State of cache-lines is O. Operations SWP, FAA on Piledriver at buffer size L1 the maximum latency was obtained on the c0 core. For the core c_1 buffer size does not affect to the operation latency. The maximal latency of operations SWP, FAA, store on K10 is obtained for buffer size L2 on c_0 core.

If we compare all the operations among each other, "successful CAS" has the highest latency and load has the lowest latency. For example, for the K10, Westmere-EP cores minimal latency was obtained for the state S on the core c_0 and equals from 12 to 24 ns; for Piledriver processor CAS has the minimal

Table 2: Suboptimal parameters for the architecture Nehalem-EP.

Operation	**Cache-line state**	**Buffer size**	**Core**	**Shared memory level**	l, **ns**	b, **MiB/s**
Load	Shared	L2	c_0, c_1, c_2	L3	1,30 – 1,34	706 - 730
Store	Shared	L1, RAM	c_0, c_1, c_2	L3	2,32	410
FAA	Shared	RAM	c_0, c_1, c_2	L3	2,32 – 2,33	409 - 410
SWP	Shared	L2	c_0, c_1, c_2	L3	2,77 – 2,79	341 - 343
unCAS	Exclusive	L2, L3, RAM	c_1, c_2	L3	4,97 – 5,12	186 - 191
CAS	Modified	L3, RAM	c_0, c_1	L3	9,86 – 12,8	75 - 97



(a) Load

(b) Store

(c) Swap (SWP)

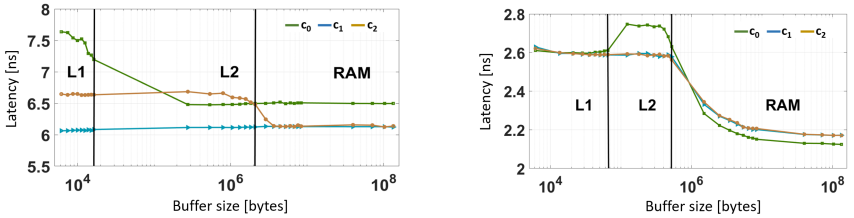
(d) Fetch-and-add (FAA)

(e) Successful CAS

(f) Unsuccessful CAS (unCAS)

Fig. 7: Latency of atomic operations for K10 processor for Shared cache-line state

value on cores c_0 and c_2 (from 42 to 44 ns); on Nehalem-EP in the state S CAS performs with minimal latency on the core c_0 (from 22 to 46 ns), wherein the maximal latency (46 ns) was obtained for buffer size L2. For Piledriver architecture latency of load operation (1.76 ns) exceed the minimal latency of CAS (12.39 ns) by 7 times. For K10 architecture minimal latency of load (1.72 ns) exceeds minimal latency of CAS (22.38 ns) by 12 times. For Nehalem-EP the ration of minimum load latency (1.3 ns) to minimum CAS latency (9.86) is 7.5. For Westmere-EP architecture, the ratio of minimum load latency (1.1 ns) to minimum CAS latency (4.9 ns) is 4.5. Comparing the microarchitecture, we note



(a) Piledriver (b) K10

Fig. 8: Latency of atomic operation SWP for cache lines in the Owned state

Table 3: The ratio of maximum and minimum throughput (b_{max}/b_{min}) when performing atomic operations.

Operation	Piledriver	K10	Nehalem-EP	Westmere-EP
Load	1.4	1.1	2.1	2
FAA	1.1	1.2	2.2	1.1
SWP	1.1	1.2	2.1	1.1
unCAS	1.1	1.2	2.4	2.0
Store	3.9	1.1	2.1	1.1
CAS	1.1	1.6	6.1	7.2

that at average the lowest latency was obtained on the Westmere-EP processor (MESIF protocol), and the largest on the Piledriver (MOESI protocol).

4 Conclusions

In this work, we developed the algorithms and software tools and conducted experiments for efficiency analysis of atomic operations on modern multicore shared-memory systems depending on buffer size, cache line state and data locality. We experimentally show that operations "unsuccessful CAS", FAA and SWP has the minimum latency. Load operation has the minimum latency and operations "successful CAS" and store – maximum latency.

We analyzed the experimental results and gave the recommendations for increasing the throughput and minimizing the latency of atomic operation performance of modern processors. So, the application of our recommendations will increase the throughput of atomic operations on the Piledriver processors from 1.1 to 3.9 times, on the K10 processor - from 1.1 to 1.6 times, on the Nehalem-EP processor from 2.1 to 6, 1 time, on the Westmere-EP processor - from 1.1 to 7.2 times. Thus, the results show that the execution time of atomic operations can vary widely, depending on the conditions of their execution (cache line state, localization, and buffer size). These evidences should be considered for designing new concurrent data structures and synchronization primitives.

References

1. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2011).
2. Paznikov, A., Shichkina, Y.: Algorithms for optimization of processor and memory affinity for Remote Core Locking synchronization in multithreaded applications. Information **9**(1), pp. 21. (2018) https://doi.org/10.3390/info9010021
3. Anenkov, A. D., Paznikov, A. A., Kurnosov, M. G.: Algorithms for access localization to objects of scalable concurrent pools based on diffracting trees in multicore computer systems. In: 2018 XIV International Scientific-Technical Conference on Actual Problems of Electronics Instrument Engineering (APEIE) pp. 374-380. IEEE (2018) https://doi.org/10.1109/APEIE.2018.8545197
4. Tabakov, A. V., Paznikov, A. A.: Algorithms for optimization of relaxed concurrent priority queues in multicore systems. In: 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus) pp. 360-365. IEEE (2019) https://doi.org/10.1109/EIConRus.2019.8657105
5. Tabakov, A. V., Paznikov, A. A.: Using relaxed concurrent data structures for contention minimization in multithreaded MPI programs. Journal of Physics: Conference Series **1399**(3), pp. 033037. IOP Publishing (2019) https://doi.org/10.1088/1742-6596/1399/3/033037
6. Paznikov, A. A., Smirnov, V. A., Omelnichenko, A. R. Towards Efficient Implementation of Concurrent Hash Tables and Search Trees Based on Software Transactional Memory. In: 2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon) pp. 1-5. IEEE (2019). https://doi.org/10.1109/FarEastCon.2019.8934131
7. Kulagin, I., Kurnosov, M. Instrumentation and Optimization of Transactional Sections Execution in Multithreaded Programs. In: Proc. of the Institute for System Programming of the RAS, **27**(6), pp. 135-150 (2015) https://doi.org/10.15514/ISPRAS-2015-27(6)-9
8. Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing **10**(2), pp. 99-116. (1997) https://doi.org/10.1007/s004460050028
9. Morrison, A., Afek, Y.: Fast concurrent queues for x86 processors. ACM SIGPLAN Notices **48**(8), pp. 103-12. (2013) https://doi.org/doi.org/10.1145/2442516.2442527
10. Harris, T. L.: A pragmatic implementation of non-blocking linked lists. In: International Symposium on Distributed Computing pp. 300-314. Springer (2001)
11. Elteir, M., Lin, H., Feng, W. C.: Performance characterization and optimization of atomic operations on amd gpus. In: 2011 IEEE International Conference on Cluster Computing, pp. 234-243. IEEE (2011) https://doi.org/10.1109/CLUSTER.2011.34
12. David, T., Guerraoui, R., Trigonakis, V.: Everything you always wanted to know about synchronization but were afraid to ask. In: Proc. of the Twenty-Fourth ACM Symp. on Operating Systems Principles pp. 33-48. ACM (2013)
13. Schweizer, H., Besta, M., Hoefler, T.: Evaluating the cost of atomic operations on modern architectures. In: 2015 Int. Conf. on Parallel Architecture and Compilation (PACT), pp. 445-456. IEEE (2015) https://doi.org/10.1109/PACT.2015.24
14. Dey, S., Nair, M. S.: Design and implementation of a simple cache simulator in Java to investigate MESI and MOESI coherency protocols. International Journal of Computer Applications **87**(11), pp. 6-13. (2014)
15. Molka, D., Hackenberg, D., Schone, R., Muller, M. S.: Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In: 2009 18th Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 261-270. IEEE (2009) https://doi.org/10.1109/PACT.2009.22