ВЕСТНИК ТОМСКОГО ГОСУДАРСТВЕННОГО УНИВЕРСИТЕТА
2020 Управление, вычислительная техника и информатика № 51

УДК 004.254
DOI: 10.17223/19988605/51/12

Е.А. Гончаренко, А.А. Пазников

АНАЛИЗ ЭФФЕКТИВНОСТИ ВЫПОЛНЕНИЯ АТОМАРНЫХ ОПЕРАЦИЙ В МНОГОЯДЕРНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ С ОБЩЕЙ ПАМЯТЬЮ

Публикация выполнена при поддержке РФФИ в рамках проекта № 19-07-00784 и при поддержке Совета по грантам Президента РФ (проект СП-4971.2018.5).

Проводится анализ эффективности выполнения атомарных операций «сравнение с обменом» (compare-and-swap, CAS), «выборка и сложение» (fetch-and-add, FAA), «обмен» (swap, SWP), «чтение» (load) и «запись» (store) на современных многоядерных вычислительных системах с общей памятью. Данные операции реализованы в виде процессорных инструкций и применяются при разработке параллельных программ (средства блокировки потоков и неблокируемые структуры данных). Исследуется зависимость влияния механизма когерентности кэш-памяти (cache coherence), размера и локальности данных на время выполнения атомарных операций. Разработана тестовая программа, позволяющая анализировать зависимость пропускной способности и латентности выполнения операций. Приводятся результаты анализа эффективности атомарных операций для процессоров архитектуры x86-64 и рекомендации по оптимизации их выполнения. В частности, определены атомарные операции, характеризующиеся наименьшей (load), наибольшей («удачный CAS», store) и сопоставимой («неудачный CAS», FAA, SWP) латентностью. Показано, что при различном выборе процессорного ядра для выполнения операции и состояния кэш-линии время выполнения операций может различаться в среднем в 1,5 и 1,3 раза соответственно. Выбор субоптимальных параметров позволяет увеличить пропускную способность выполнения атомарных операций от 1,1 до 7,2 раза.

Ключевые слова: атомарные операции; протокол когерентности кэша; многопоточные программы.

К многоядерным вычислительным системам (ВС) с общей памятью относятся как автономные десктопные и серверные системы, так и вычислительные узлы в составе распределенных ВС (кластерные ВС, системы с массовым параллелизмом). Такие системы могут включать десятки и сотни процессорных ядер. Например, вычислительный узел суперкомпьютера Summit, занимающего первое место в рейтинге TOP-500 (более 2 млн процессорных ядер, 4 608 узлов), включает два 24-ядерных универсальных процессора IBM Power9 и шесть графических ускорителей NVIDIA Tesla V100 (640 ядер). Sunway TaihuLight (более 10 млн процессорных ядер, третье место в рейтинге TOP-500) укомплектован 40 960 процессорами Sunway SW26010, включающими 260 ядер. При реализации кэш-памяти в таких ВС применяются различные политики включения (инклюзивный и эксклюзивный кэш), протоколы когерентности (MESI, MOESI, MESIF, MOWESI, MERSI, Dragon) и топологии межпроцессорных шин в NUMA-системах (Intel QPI, AMD HyperTransport).

Одной из наиболее значимых задач при разработке параллельных программ является создание эффективных средств синхронизации параллельных потоков. Основными методами синхронизации являются средства блокировки потоков (locks, mutexes), неблокируемые (non-blocking, lock-free) потокобезопасные (concurrent, thread-safe) структуры данных и транзакционная память [1–5]. Атомарные операции (atomic operations, atomics) применяются при реализации всех методов синхронизации. Операция называется атомарной, если она завершается в один неделимый шаг относительно других потоков. Ни один из потоков не может наблюдать операцию «частично завершенной». Если два или

более потоков выполняют операции над разделяемой переменной и хотя бы один выполняет операцию записи, то потоки должны использовать атомарные операции для избежания гонки данных (data race).

К наиболее распространенным относятся атомарные операции «сравнение с обменом» (compare-and-swap, CAS), «выборка и сложение» (fetch-and-add, FAA), «обмен» (swap, SWP), «чтение» (load) и «запись» (store). Операция load(m, r) реализует чтение значения переменной из ячейки памяти m в регистр r; store(m, r) – запись значения переменной в ячейку памяти m из регистра r. FAA(m, r_1, r_2) увеличивает (уменьшает) на размещенное в регистре r_1 значение в ячейке памяти m и возвращает предыдущее значение в регистр r_2. SWP(m, r) – обмен значений между ячейкой памяти m и регистром r. CAS(m, r_1, r_2) реализует сравнение значения ячейки памяти m с регистром r_1; если значения равны, то в m устанавливается r_2. При разработке параллельных программ необходимо учитывать влияние на эффективность атомарных операций таких аспектов, как выполнение механизма когерентности кэш-памяти, размер буфера, числа потоков, удаленности данных от ядра.

Несмотря на широкое применение, эффективность атомарных операций не проанализирована в достаточной степени. Например, считается, что CAS медленнее FAA [6] и ее семантика позволяет ввести понятие «бесполезная работа» (wasted work) [7, 8], поскольку неудачные попытки сравнения данных в памяти и в регистре приводят к дополнительной нагрузке на ядро. В работе [8] анализируется производительность атомарных операций на графических процессорах, однако сегодня остро стоит задача оценки эффективности атомарных операций на универсальных процессорах. В [9] рассматривается эффективность выполнения операций CAS, FAA, SWP с целью анализа влияния динамических параметров программы на время выполнения атомарных операций. Результаты исследований демонстрируют недокументированные свойства тестируемых систем и оценки времени выполнения атомарных операций. Однако тесты проводились при фиксированной частоте процессорных ядер, использовании больших страниц памяти и с выключенной предварительной выборкой данных, что искажает результаты моделирования для реальных программ. Кроме того, не исследовались атомарные операции load и store.

В работе рассматриваются операции CAS, FAA, SWP, load, store, при этом условия проведения экспериментов максимально приближены к реальным условиям выполнения параллельных программ, в частности не проводятся фиксация частоты ядер, изменение размера страниц, отключение предварительной выборки данных. Проводится анализ эффективности атомарных операций для процессоров архитектуры x86-64. Кроме того, рассматриваются разные варианты выполнения CAS (удачный и неудачный), а также выполнение атомарных операций при использовании данных локальной и удаленной кэш-памяти (по отношению к ядру, выполняющему операции).

1. Методика проведения экспериментов

В качестве показателей эффективности используются латентность l выполнения атомарной операции и пропускная способность b. Пропускная способность $b = n/t$, где n – число выполненных операций за время t при реализации последовательного доступа к ячейкам буфера.

Для точного измерения времени применялся набор инструкций RDTSCP. Для предотвращения переупорядочивания инструкций использовались полные барьеры памяти.

Исследуется влияние состояния кэш-линии в рамках протоколов когерентности (M, E, S, O, I, F) на эффективность выполнения атомарных операций. Определим основные состояния кэш-линий. M (Modified) – кэш-линия модифицирована и содержит актуальные данные. O (Owned) – кэш-линия содержит актуальные данные и является их единственным владельцем. E (Exclusive) – кэш-линия содержит актуальные данные, которые соответствуют состоянию памяти. S (Shared) – кэш-линия содержит актуальные данные, при этом другие процессорные ядра имеют копии этих данных в разделяемом состоянии. F (Forwarded) – кэш-линия содержит наиболее свежие, корректные данные, при этом другие ядра в системе могут иметь копии данных в разделяемом состоянии. I (Invalid) – кэш-линия содержит некорректные данные.

Разработана тестовая программа. Перед началом выполнения организуется буфер q в виде целочисленного массива размера s = 128 Мб. Данные размещаются в кэш-памяти, кэш-линии переводятся в заданное состояние протокола когерентности. Подкачка страниц не применялась. Данные не выравнивались. Перевод кэш-линий в состояние M происходит при записи произвольных значений в ячейки буфера. Для перевода в состояние E выполняется запись в ячейки буфера, затем – инструкция clflush для перевода кэш-линий в состояние I, после этого – чтение ячеек буфера. Переход в состояние S происходит после чтения ячеек буфера из кэш-памяти в состоянии E одного ядра в кэш-память другого ядра. Перевод кэш-линий в состояние O достигается путем чтения из кэш-памяти в состоянии M одного ядра в кэш-память другого ядра, при этом кэш-линии ядра из M переключаются в состояние O.

Для операции CAS выполняется два эксперимента: для успешной и неудачной операции. Неудачным считается CAS, при выполнении которого $m \neq r_1$. Заведомо неудачное выполнение CAS достигается путем сравнения адреса указателя с данными по этому указателю. В успешном CAS $m = r_1$. Эксперименты проводились на процессорах AMD Athlon II X4 640 (микроархитектура K10), AMD A10-4600M (микроархитектура Piledriver) (протокол когерентности MOESI [10]), Intel Xeon X5670 (микроархитектура Westmere-EP) и Intel Xeon E5540 (микроархитектура Nehalem-EP) (MESIF [11]) (рис. 1). Размер кэш-линии 64 байта. В качестве компилятора использовался компилятор GCC 4.8.5, операционная система SUSE Linux Enterprise Server 12 SP3 (версия ядра 4.4.180).

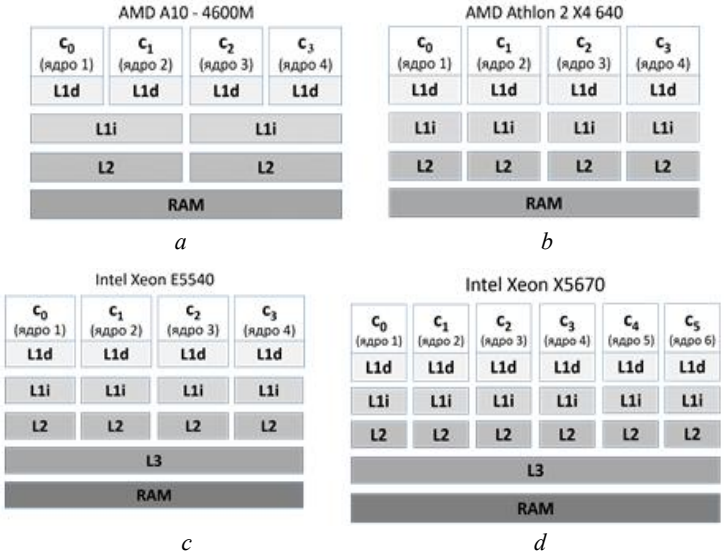

Рис. 1. Исследуемые микроархитектуры: *a* – Piledriver, *b* – K10, *c* – Nehalem-EP, *d* – Westmere-EP
Fig. 1. Target microarchitectures: *a* – Piledriver, *b* – K10, *c* – Nehalem-EP, *d* – Westmere-EP

Опишем основные шаги алгоритма измерения времени выполнения атомарных операций для состояния E (рис. 2). Вспомогательная функция DoOper реализует выполнение заданной атомарной операции для всех элементов буфера (строки *1–4)*. Основной алгоритм выполняется до тех пор, пока размер буфера не достигнет максимального (строка *5*). Задействованный диапазон тестового массива *testSizeBuffer* в экспериментах варьирует от 6 Кб (минимальный размер кэша первого уровня) до 128 Мб. Каждый эксперимент запускается *nruns* раз (строка *6*). Выполняется запись произвольных данных в ячейки буфера для перевода кэш-линий ядра c_0 в состояние M (для остальных ядер состояние кэш-линий переводится в I) (строка *7*). Далее выполняется инвалидация кэш-линий (строка *12*) с последующим чтением, что переводит кэш-линий ядра c_0 в состояние E (строка *9*). На следующем шаге над каждой переменной выполняется атомарная операция (строка *11*). Вычисляется время выполнения операции и суммарное время выполнения (строка *13*). Рассчитываются латентность (строка *15*), пропускная способность (строка *16*) и увеличивается текущий размер буфера на $step = L_x / 8$ (где L_x – размер кэш-памяти уровня x = 1, 2, 3)) (строка *17*).

Алгоритмы выполнения атомарных операций для состояний M и I аналогичны описанному, при этом для M строки *8*, *9* не выполняются, для I не выполняются строки *7*, *9*.

```
1  Function DoOper(oper)
2  for i = 0 to d do
3     buffer.oper()
4  end for
5  while d ≤ testSizeBuffer do
6     for j = 0 to nruns do
7        DoOper(store(1))
8        clflush(buffer, d);
9        DoOper(load)
10       start = get_time();
11       DoOper(AtomicOp(buffer[i]))
12       end = get_time();
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

```
1  Первый поток на c0:
2  while d ≤ testSizeBuffer do
3     for j = 0 to nruns do
4        DoOper(store(1))
5        clflush(buffer,d);
6        DoOper(load)
7  Второй поток на c2:
8        DoOper(load)
9  Первый поток на c0:
10       start = get_time();
11       DoOper(AtomicOp(buffer[i]))
12       end = get_time();
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

Рис. 2. Алгоритм измерения латентности и пропускной способности выполнения атомарных операций SWP/FAA/CAS/Load/Store: *a* – Exclusive c_0, *b* – Shared c_0

Fig. 2. Algorithm for measuring the latency and throughput of atomic operations SWP/FAA/CAS/Load/Store and throughput: *a* – Exclusive c_0, *b* – Shared c_0

Основные шаги алгоритма измерения времени выполнения атомарных операций для c_0 в состоянии кэш-линий S (см. рис. 2, *b*). Алгоритм выполняется в двух потоках, привязанных к ядрам c_0 и c_2. Синхронизация реализуется посредством атомарных флагов. В первом потоке на ядре c_0 выполняется запись произвольных данных в буфер, перевод кэш-линий в состояние M, при этом для остальных ядер состояние изменяется на I (строка *4*). Далее выполняются очистка кэш-линий (сброс в I) (строка *5*) и чтение данных для изменения состояния кэш-линий c_0 в состояние E (строка *6*). Во втором потоке (ядро c_2) выполняется чтение данных, что изменяет состояние кэш-линий ядер c_0 и c_2 на S (строка *8*). Далее первый поток на ядре c_0 реализует атомарную операцию над элементами тестового буфера (строка *11*). Вычисляются время выполнения операции и показатели эффективности (строки *15–17*).

На рис. 3, 4 приведены результаты экспериментов для операции SWP.

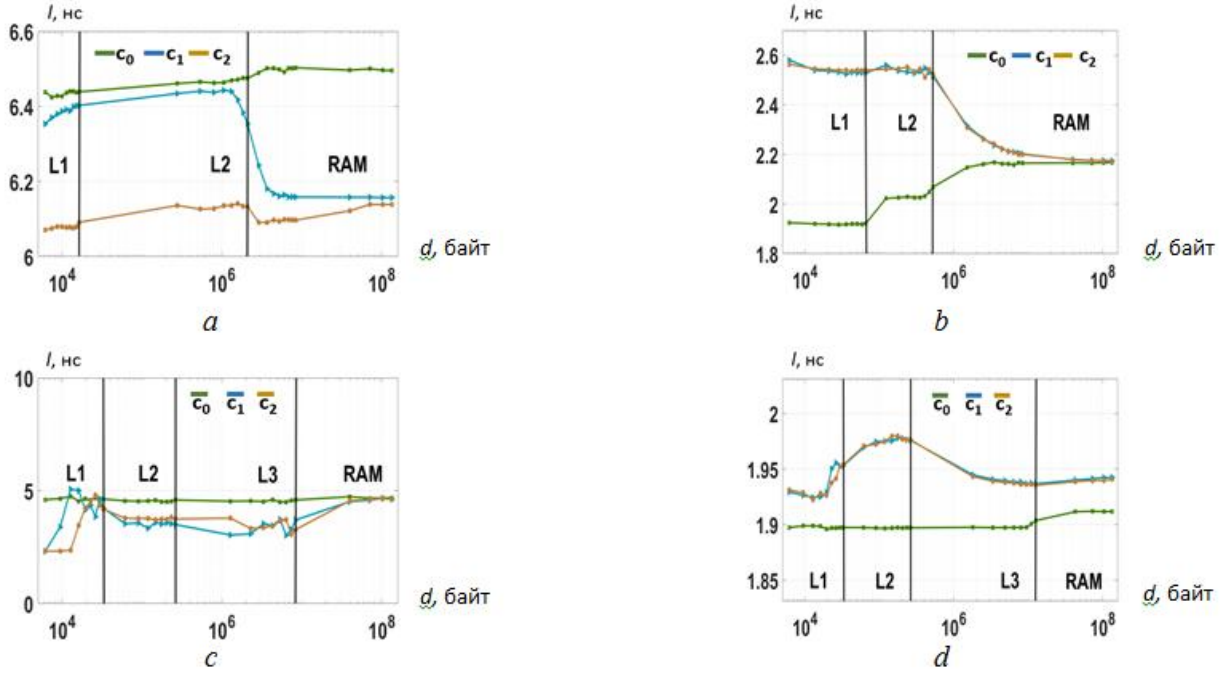

Рис. 3. Латентность выполнения атомарной операции SWP для кэш-линий в состоянии Exclusive: *a* – Piledriver, *b* – K10, *c* – Nehalem-EP, *d* – Westmere-EP

Fig. 3. Latency an atomic operation SWP for cache lines in the Exclusive state: *a* – Piledriver, *b* – K10, *c* – Nehalem-EP, *d* – Westmere-EP

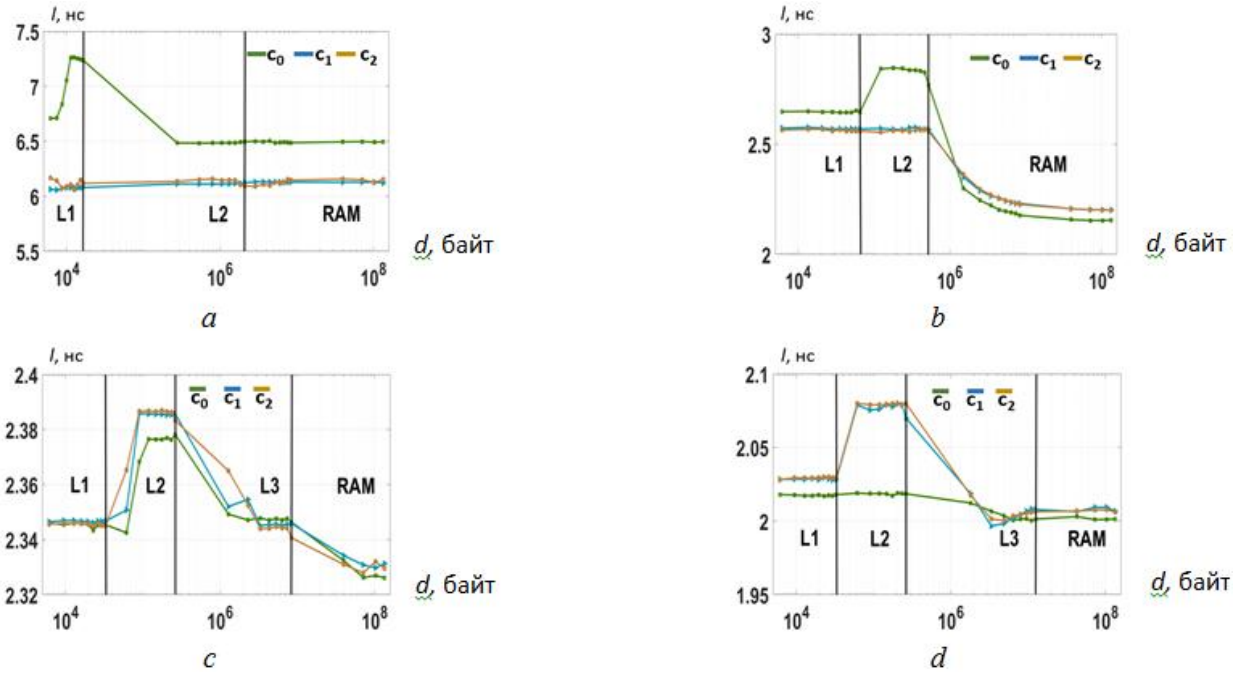

Рис. 4. Латентность выполнения атомарной операции SWP для кэш-линий в состоянии Shared: *a* – Piledriver, *b* – K10, *c* – Nehalem-EP, *d* – Westmere-EP

Fig. 4. Latency an atomic operation SWP for cache lines in the Shared state: *a* – Piledriver, *b* – K10, *c* – Nehalem-EP, *d* – Westmere-EP

Опишем основные шаги алгоритма измерения времени выполнения атомарных операций для ядер c_1 и c_2 в состоянии S кэш-линий ядра c_0 (рис. 5). Алгоритм выполняется в трех потоках, привязанных к ядрам c_0, c_1, c_2. Первый поток (ядро c_0) реализует запись в буфер, что переводит состояние кэш-линий c_0 в M (для остальных ядер состояние изменяется на I) (строка *4*). Далее выполняются сброс состояния в I (строка *5*) и чтение данных, что переводит кэш-линии c_0 в состояние E (строка *6*). Во втором потоке (ядро c_1) выполняется чтение данных (состояние кэш-линии c_0 и c_1 изменяется на S) (строка *8*). На следующем шаге в третьем потоке (ядро c_2) над каждым элементом буфера выполняется атомарная операция (строка *11*). Вычисляются показатели эффективности (строки *15–17*). Для измерения латентности выполнения атомарных операций на ядре c_1 используется аналогичный алгоритм, в котором шаги для ядер c_1 и c_2 меняются местами.

```
1  Первый поток на c0:
2  while d ≤ testSizeBuffer do
3     for j = 0 to nruns do
4        DoOper(store(1))
5        clflush(buffer,d);
6        DoOper(load)
7  Второй поток на c1:
8        DoOper(load)
9  Третий поток на c2:
10       start = get_time();
11       DoOper(AtomicOp(buffer[i]))
12       end = get_time();
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
```

а

```
1  Первый поток на c0:
2  while d ≤ testSizeBuffer do
3     for j = 0 to nruns do
4        DoOper(store(1))
5  Второй поток на c2:
6        DoOper(load)
7  Первый поток на c0:
8        start = get_time();
9        DoOper(AtomicOp(buffer[i]))
10       end = get_time();
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
```

б

Рис. 5. Алгоритм измерения латентности и пропускной способности выполнения атомарных операций: *a* – кэш-линии ядра c_0 в состоянии Shared, измерения для ядер c_1 и c_2, *b* – состояние Owned

Fig. 5. Algorithm for measuring the latency and throughput of atomic operations: *a* – Shared c_0, measurements for c_1 and c_2, *b* – Owned state

Опишем основные шаги алгоритма измерения времени выполнения атомарных операций в состоянии кэш-линий O на ядре c_0 (см. рис. 5, *b*). Алгоритм выполняется в двух потоках, привязанных к ядрам c_0 и c_2. В первом потоке (ядро c_0) произвольные данные записываются в буфер для изменения состояния кэш-линий ядра c_0 в M (для остальных ядер состояние изменяется I) (строка *4*). Во втором потоке (ядро c_2) выполняется чтение данных, что изменяет состояние кэш-линий локального ядра c_0 на O, а ядра c_2 – на S (строка *8*). На следующем шаге в первом потоке (ядро c_0) над каждой переменной буфера выполняется атомарная операция (строка *9*). Вычисляются время выполнения операции и показатели эффективности (строки *13–15*).

На рис. 6 приведены результаты для операции SWP, состояние O.

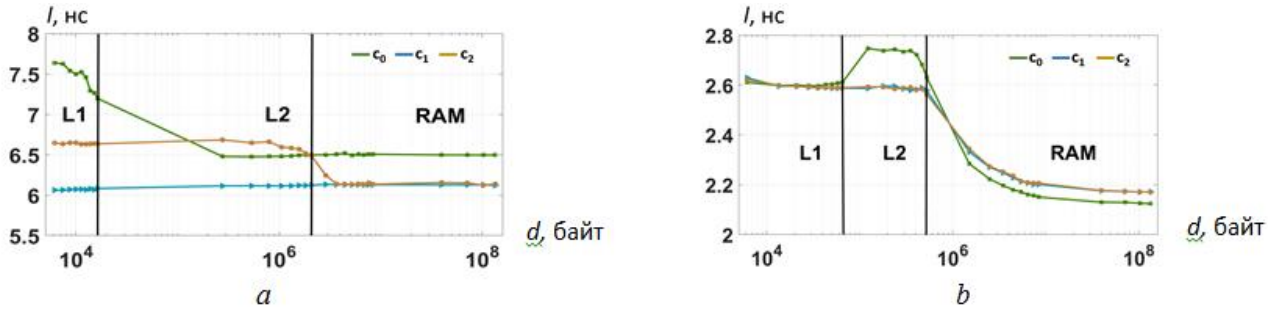

Рис. 6. Латентность выполнения атомарной операции SWP кэш-линий в состоянии Owned: *a* – архитектура Piledriver, *b* – архитектура K10

Fig. 6. Latency of atomic operation SWP for cache lines in the Owned state: *a* – Piledriver, *b* – K10

Таблица 1

Субоптимальные параметры для архитектуры Westmere-EP

Операция	Субоптимальные параметры			Общий уровень памяти	*l*, нс	*b*, Мб/с
	Состояние кэш-линий	Размер буфера	Ядро			
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, L3	c_0, c_1, c_2	L3	4,9–19,3	49–194

Таблица 2

Отношения максимальной и минимальной пропускной способности при выполнении атомарных операций

Операция	Piledriver b_{max}/b_{min}	K10 b_{max}/b_{min}	Nehalem-EP b_{max}/b_{min}	Westmere-EP b_{max}/b_{min}
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

В табл. 1 представлены субоптимальные параметры выполнения атомарных операций на процессоре с микроархитектурой Westmere-EP. Аналогичные результаты получены и для других процессоров. Так, например, для процессоров микроархитектур K10, Nehalem-EP, Westmere-EP наименьшая латентность выполнения операции load получена в состоянии S на ядре c_0 (от 1,14 до 1,81 нс), в то время как для процессора с микроархитектурой Piledriver данная операция имеет наименьшую латентность на ядрах c_0 и c_2 (от 1,8 до 2,4 нс). В табл. 2 даны отношения максимальной и минимальной пропускной способности.

2. Анализ результатов экспериментов

Состояние кэш-линий M. Значения показателей эффективности для атомарных операций SWP, FAA, unCAS на ядре c_0 архитектуры Piledriver незначительно варьируют при изменении размера буфера, в то время как для ядер c_1 и c_2 увеличение размера буфера более L2 приводит к сокращению латентности до минимального значения для обоих ядер. Для K10 при размере буфера, превышающем L2, латентность операций SWP, FAA, unCAS различается в пределах погрешности для всех ядер. При выполнении load, SWP, CAS для Nehalem-EP латентность сопоставима для всех ядер при размере буфера более L3. При выполнении операций load, SWP для Westmere-EP наблюдается аналогичная ситуация, для остальных операций (store, CAS, FAA, unCAS) латентность является наименьшей для ядра c_0 и не зависит от размера буфера.

Состояние кэш-линий E. При выполнении операций SWP, FAA, unCAS на архитектуре Piledriver и размере буфера более L2 для ядер c_1 и c_2 латентность сопоставима, при этом на ядре c_0 получены максимальные значения. Для архитектуры K10 латентность операций SWP, FAA, unCAS аналогична для всех ядер при размере буфера, превышающем L2. Для Nehalem-EP латентность SWP, load сопоставима на всех ядрах при размере буфера более L3. Для Westmere-EP минимальная латентность для load, SWP, FAA, CAS получена на ядре c_0, для ядер c_1 и c_2 при размере буфера более L3 латентность различается незначительно.

Состояние кэш-линий S. При выполнении SWP, FAA на архитектуре Piledriver при размере буфера L1 латентность максимальна на ядре c_0. Для K10 при размере буфера L2 наибольшая латентность получена на ядре c_0 для SWP, FAA, store. Для Nehalem-EP и Westmere-EP при размере буфера L2 латентность выполнения SWP, FAA максимальна на ядрах c_1, c_2.

Состояние кэш-линий I. При выполнении SWP, FAA, unCAS на архитектуре Piledriver размер буфера незначительно влияет на время выполнения для всех ядер, наименьшая латентность получена на ядрах c_1 и c_2. При выполнении load, SWP на K10 при размере буфера не более L2 латентность выполнения минимальна для ядра c_0. Для Nehalem-EP латентность SWP, load сопоставима на всех ядрах при размере буфера более L3. При выполнении load, SWP, FAA, CAS на Westmere-EP наименьшая латентность получена на ядре c_0.

Состояние кэш-линий O. При выполнении атомарных операций SWP, FAA на архитектуре Piledriver при размере буфера L1 наибольшая латентность получена на ядре c_0. Для ядра c_1 размер буфера не влияет на время выполнения операций. Наибольшая латентность операций SWP, FAA, store на архитектуре K10 получена при размере буфера L2 на ядре c_0.

Наибольшей латентностью, по сравнению с другими операциями, характеризуется операция CAS (успешный). Операция load выполняется с наименьшей латентностью. Например, для процессоров K10, Westmere-EP минимальная латентность CAS получена для состояния S на ядре c_0 и составляет от 18 до 24 нс, для процессора Piledriver CAS имеет наименьшую латентность на ядрах c_0 и c_1 (от 42 до 44 нс), на процессоре Nehalem-EP в состоянии S CAS выполняется с наименьшей латентностью на ядре c_0 (от 22 до 46 нс), при этом наибольшая латентность (46 нс) получена при размере буфера L2. Для архитектуры Piledriver минимальная латентность операции load (1,76 нс) превышает минимальную латентность операции CAS (12,39 нс) в 7 раз. Для архитектуры K10 минимальная латентность load (1,72 нс) превышает минимальную латентность CAS (22,38 нс) в 13 раз. Для Nehalem-EP отношение минимальной латентности load (1,3 нс) к минимальной латентности CAS (9,86 нс) – 7,5. Для архитектуры Westmere-EP отношение минимальной латентности load (1,1 нс) к минимальной латентности CAS (4,9 нс) – 4,5. Сравнивая микроархитектуры, отметим, что в среднем наименьшая латентность получена на процессоре Westmere-EP (MESIF), наибольшая – на Piledriver (MOESI).

Заключение

Разработан инструментарий и проведен анализ эффективности выполнения атомарных операций в многоядерных вычислительных системах с общей памятью в зависимости от размера буфера,

состояния кэш-линий и локальности данных. Экспериментально показано, что операции «неудачный CAS», FAA и SWP характеризуются сопоставимой задержкой. Для операций load получена наименьшая латентность, для «удачный CAS» и store – наибольшая.

В результате анализа результатов экспериментов построены рекомендации по увеличению пропускной способности и минимизации латентности выполнения атомарных операций на протестированных процессорах. Так, применение данных рекомендаций позволит увеличить пропускную способность на процессоре с микроархитектурой Piledriver от 1,1 до 3,9 раз, для процессора K10 – от 1,1 до 1,6 раз, для процессора Nehalem-EP от 2,1 до 6,1 раз, для процессора Westmere-EP – от 1,1 до 7,2 раз. Таким образом, полученные результаты показывают, что время выполнения атомарных операций может варьировать в широких пределах в зависимости от условий их выполнения (состояние кэш-линии, локализация и размер буфера). Это необходимо учитывать при разработке разделяемых структур данных и примитивов синхронизации.

ЛИТЕРАТУРА

1. Herlihy M., Shavit N. The art of multiprocessor programming. Morgan Kaufmann, 2012. 537 p.
2. Пазников А.А. Оптимизация делегирования выполнения критических секций на выделенных процессорных ядрах // Вестник Томского государственного университета. Управление, вычислительная техника и информатика. 2017. № 38. С. 52–58.
3. Аненков А.Д., Пазников А.А. Алгоритмы оптимизации масштабируемого потокобезопасного пула на основе распределяющих деревьев для многоядерных вычислительных систем // Вестник Томского государственного университета. Управление, вычислительная техника и информатика. 2017. № 39. С. 73–84.
4. Кулагин И.И., Курносов М.Г. Оптимизация обнаружения конфликтов в параллельных программах с транзакционной памятью // Вестник Южно-Уральского государственного университета. Сер. Вычислительная математика и информатика. 2016. Т. 5, № 4. С. 46–60.
5. Shavit N., Touitou D. Software transactional memory // Distributed Computing. 1997. V. 10, No. 2. P. 99–116.
6. Morrison A., Afek Y. Fast Concurrent Queues for x86 Processors // PPoPP'13. 2013. P. 103–112.
7. Harris T.L. A Pragmatic Implementation of Non-blocking Linked-Lists // DISC'01. 2001. P. 300–314.
8. Elteir M., Lin H., Feng W. Performance characterization and optimization of atomic operations on amd gpus // IEEE Int. Conf. on Cluster Computing. 2011. P. 234–243.
9. Schweizer H., Besta M., Hoefler T. Evaluating the cost of atomic operations on modern architectures // 2015 Int. Conf. on Parallel Architecture and Compilation (PACT). 2015. P. 445–456.
10. Dey S., Nair M.S. Design and Implementation of a Simple Cache Simulator in Java to Investigate MESI and MOESI Coherency Protocols // Int. J. of Computer Applications. 2014. V. 87, No. 11. P. 6–13.
11. Molka D., Hackenberg D., Schone R., Muller M.S. Memory performance and cache coherency effects on an intel nehalem multiprocessor system // 18th Int. Conf. on Parallel Architectures and Compilation Techniques. 2009. P. 261–270.

Поступила в редакцию 8 июля 2019 г.

Goncharenko E.A., Paznikov A.A. (2020) ANALYSIS OF THE EFFICIENCY OF ATOMIC OPERATIONS IN MULTI-CORE SHARED-MEMORY COMPUTER SYSTEMS. *Vestnik Tomskogo gosudarstvennogo universiteta. Upravlenie vychislitelnaja tehnika i informatika* [Tomsk State University Journal of Control and Computer Science]. 51. pp. 102–110

DOI: 10.17223/19988605/51/12

One of the most relevant problems of parallel programming, both for shared-memory and distributed-memory systems, is the development of effective thread synchronization tools. The main synchronization methods are locks (spinlocks, mutexes), non-blocking (lock-free, wait-free, obstruction-free) concurrent data structures and software transactional memory. Atomic operations (atomics), implemented in hardware as processor instructions, are used in all these synchronization techniques. An operation is called atomic if it completes in one indivisible step relative to other threads. None of the threads can see the modification of a variable partially completed. In other words, if two or more threads perform operations on a shared variable and at least one writes into it, then both threads should use atomics.

For efficient parallel programming with atomics, we should consider the caching algorithms. In particular, we have to study the influence of the cache coherence protocol on the efficiency of atomic operations, as well as the effect of buffer size, the number of threads, the distance of data from the processor core (shared cache level). 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 multi-core processors. These operations are the most widely used and implemented as atomic instructions at the hardware level on most microarchitectures. We study the impact of the cache coherence protocol, buffer size and data locality on the efficiency of atomic operations. In particular, we study the impact of the cache line state within the cache coherence protocol (M, E, S, O, I, F) on the efficiency of atomic operations. M (Modi-

fied) – a cache line is modified and contains actual data. O (Owned) – a cache line contains actual data and is their sole owner. E (Exclusive) – the cache line contains actual data that correspond to the state of the main memory. S (Shared) – cache line contains actual data and other processor cores have actual copies of this data in a shared state. F (Forwarded) – the cache line contains correct data, while other cores in the system may have copies of the data in a shared state. I (Invalid) – the cache line contains incorrect data.

At present, the efficiency of atomic operations has not been sufficiently analyzed. This paper discusses the CAS, FAA, SWP, load, store operations and experimental conditions are as close as possible to the actual parallel programs: the core frequency is not fixed, pages are not resized, and data prefetching is enabled. In addition, we consider different variants of CAS operation (successful and unsuccessful), as well as atomic operations using data from the local and remote caches (with respect to the core performing the operations).

We designed the algorithms and software toolkit for evaluation efficiency of atomic operations. We experimentally shown that the operations "unsuccessful CAS", FAA and SWP have comparable latency. For operations load and CAS, the smallest and largest latencies are obtained, respectively. A "successful CAS" with any buffer size and cache line conditions runs faster on the local core. The unsuccessful CAS operation is performed faster on the local core in any states, if the buffer size does not exceed the sizes of L1 and L2 caches. SWP operation has the lowest execution latency on the local core for any size of a buffer in the Modified and Exclusive cache line states.

We analyze the experimental results and give the recommendations to increase the throughput and minimize latency of atomic operations on the given processor architectures. For example, based on these recommendations, we can increase the throughput of atomics on processors with Piledriver microarchitecture from 1.1 to 3.9 times, for K10 processors from 1.1 to 1.6 times, for Nehalem-EP processors from 2.1 to 6.1 time, for Westmere-EP processors from 1.1 to 7.2 times. Thus, the obtained results show that the latency of atomic operations can vary in wide range, depending on the conditions of their execution (cache line state, locality and buffer size). These evidences should be considered in designing concurrent data structures and synchronization primitives.

Keywords: atomic operations; cache coherence protocol; multithreaded programs.

GONCHARENKO Evgeniy Andreevich (Student, Department of Computer Science and Engineering, Saint Petersburg Electrotechnical University "LETI", Saint-Petersburg, Russian Federation).
E-mail: kesvesna@rambler.ru

PAZNIKOV Alexey Alexandrovich (Candidate of Technical Sciences, Associate Professor, Senior Researcher, Department of Computer Science and Engineering, Saint Petersburg Electrotechnical University "LETI", Saint-Petersburg, Russian Federation).
E-mail: apaznikov@gmail.com

REFERENCES

1. Herlihy, M. & Shavit, N. (2012) *The art of multiprocessor programming*. Morgan Kaufmann.
2. Paznikov, A.A. (2017) Optimization method of remote core locking. *Vestnik Tomskogo gosudarstvennogo universiteta. Upravlenie, vychislitel'naya tekhnika i informatik*a – *Tomsk State University Journal of Control and Computer Science*. 38. pp. 52–58. DOI: 10.17223/19988605/38/8
3. Anenkov, A.D. & Paznikov, A.A. (2017) Algorithms of optimization of scalable thread-safe pool based on diffracting trees for multicore computing systems. *Vestnik Tomskogo gosudarstvennogo universiteta. Upravlenie, vychislitel'naya tekhnika i informatik*a – *Tomsk State University Journal of Control and Computer Science*. 39. pp. 73–84. DOI: 10.17223/19988605/39/10
4. Kulagin, I.I. & Kurnosov, M.G. (2016) Optimization of conflict detection in parallel programs with transactional memory (2016). Vestnik Yuzhno-Ural'skogo gosudarstvennogo universiteta. Ser. Vychislitel'naya matematika i informatika – *Bulletin of the South Ural State University. Series: Computational Mathematics and Software Engineering*. 4. pp. 46–60. DOI: 10.14529/cmse160404
5. Shavit, N. & Touitou, D. (1997) Software transactional memory. *Distributed Computing*. 10(2). pp. 99–116. DOI: 10.1007/s004460050028
6. Morrison, A. & Afek, Y. (2013) Fast Concurrent Queues for x86 Processors. *PPoPP'13*. pp. 103–112. DOI: 10.1145/2442516.2442527
7. Harris, T.L. (2001) A Pragmatic Implementation of Non-blocking Linked-Lists. *DISC'01*. pp. 300–314. DOI: 10.1007/3-540-45414-4_21
8. Elteir, M., Lin, H., & Feng, W.C. (2011) Performance characterization and optimization of atomic operations on amd gpus. *IEEE Int. Conf. on Cluster Computing*. pp. 234–243. DOI: 10.1109/CLUSTER.2011.34
9. Schweizer, H., Besta, M. & Hoefler, T. (2015) Evaluating the cost of atomic operations on modern architectures. *2015 Int. Conf. on Parallel Architecture and Compilation (PACT)*. pp. 445–456. DOI: 10.1109/PACT.2015.24
10. Dey, S. & Nair, M. S. (2014) 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. DOI: 10.5120/15250-3757
11. Molka, D., Hackenberg, D., Schone, R. & Muller, M.S. (2009) Memory performance and cache coherency effects on an intel nehalem multiprocessor system. *18th Int. Conf. on Parallel Architectures and Compilation Techniques*. pp. 261–270. DOI: 10.1109/PACT.2009.22