IEICE TRANS. INF. & SYST., VOL.E105–D, NO.5 MAY 2022

PAPER Special Section on Data Engineering and Information Management

Implementation of a Multi-Word Compare-and-Swap Operation without Garbage Collection

Kento SUGIURA[†a)], ***Nonmember and*** **Yoshiharu ISHIKAWA**[†], ***Member***

SUMMARY With the rapid increase in the number of CPU cores, software that can utilize these many cores is required. A lock-free algorithm based on compare-and-swap (CAS) operations is one of the concurrency control methods to implement such multi-threading software. A *multi-word CAS (MwCAS)* operation is an extension of a CAS operation to swap multiple words atomically. However, we noticed that the performance of the existing MwCAS implementation is limited because of *garbage collection* even if in a low-contention environment. To achieve high performance in low-contention workloads, we propose a new MwCAS algorithm without garbage collection. Experimental results show that our approach is three to five times faster than implementation with garbage collection in low-contention workloads. Moreover, the performance of the proposed method is also superior in a high-contention environment.

key words: *lock-free algorithms, CAS operations, multi-threading software*

1. Introduction

With the rapid increase in the number of CPU cores, software that can utilize these many cores is required. In the case of databases, several transaction engines such as Silo [1] and HyPer [2] and indexes such as Masstree [3], ART [4], and P-Tree [5] have been proposed to improve performance by utilizing multi-threads. These implementations achieve excellent scalability with respect to the number of cores and provide a baseline for the future software.

A lock-free algorithm [6], [7] is one of the concurrency controls to implement such multi-threading software. Lock-free algorithms often utilize a *compare-and-swap (CAS)* operation, which compares the current value of a target memory address with a given expected value and then stores a new value if and only if the two values are the same, to reduce a conflict-inducing period. Because a CAS operation is usually prepared as a CPU instruction, carefully implemented lock-free algorithms achieve high scalability by multi-threads. In the case of indexes, Bw-tree [8], [9] and BzTree [10] have been proposed as completely lock-free index structures.

A *multi-word CAS (MwCAS)* operation is an extension of a CAS operation that swaps multiple words*atomically [11]–[14]. Note that the target words are stored on disjoint memory addresses. Because it is difficult to implement a complex data structure with a lock-free feature, it was expected that using MwCAS operations would simplify that task. However, the performance of a MwCAS operation decreases in a *high-contention* environment because its conflict-inducing period is longer than that of a CAS operation. Since one of the main advantages of using CAS operations is reduction of a conflict-inducing period, MwCAS operations are not mainstream and are not currently implemented as CPU instructions.

Wang et al. [15] and Arulraj et al. [10] re-examined and used an MwCAS operation to implement lock-free indexes in persistent memory. Compared with existing lock-free data structures such as a lock-free queue [6] and deque [7], lock-free indexes make a relatively *low-contention* environment. For example, in the case of a lock-free queue, enqueue and dequeue operations always require access to the same memory addresses (i.e., tail and head pointers), and these addresses are directly accessed from a queue instance. In contrast, the CAS target words of lock-free indexes are distributed to each leaf node, and it is necessary to traverse a tree structure from its root to access these leaf nodes. This makes a relatively low-contention environment for MwCAS operations, and Wang et al. [15] reported that their Bw-tree implemented by an MwCAS operation performs comparably to the original one [8].

However, we noticed that the performance of Wang et al.'s MwCAS implementation is limited because of *garbage collection* even if used in a low-contention environment. Their implementation is based on Harris et al.'s CASN algorithm [11], which uses a *MwCAS descriptor* to control each MwCAS operation. The memory space of MwCAS descriptors are allocated dynamically and released in garbage collection. Wang et al. prepared a pool of MwCAS descriptors in advance to improve performance, but its garbage collection also degrades both throughput and latency. Consequently, its throughput of single-word CAS is ten times smaller than that of a standard CAS operation.

To achieve high performance in a low-contention environment, we propose a new MwCAS algorithm without garbage collection. We summarize our contributions as follows.

- We propose a new MwCAS algorithm and remove garbage collection from its implementation. We implement the proposed method as a C++ library**.
- The experimental results show that our approach is

Manuscript received June 26, 2021.
Manuscript revised November 16, 2021.
Manuscript publicized February 3, 2022.

[†]The authors are with Graduate School of Informatics, Nagoya University, Nagoya-shi, Aichi, 464–8601 Japan.
a) E-mail: sugiura@i.nagoya-u.ac.jp

DOI: 10.1587/transinf.2021DAP0011

*A word refers to data of a certain byte length, usually 8 bytes in the current environment.

**https://github.com/dbgroup-nagoya-u/mwcas

three to five times faster than the implementation with garbage collection in low-contention workloads. Moreover, the performance of the proposed method is also superior in a high-contention environment.
- The experimental results suggest that the effect of multi-threading assistance for complex lock-free data structures is small. This indicates that we must reconsider the manner of structure modification in lock-free indexes.

The rest of this paper is organized as follows. We present related work in Sect. 2, and in Sect. 3 we introduce the linearization strategy and Harris et al.'s CASN algorithm as preliminaries. We explain the ideas of the proposed method and a detailed algorithm in Sect. 4. We evaluate the proposed method experimentally in Sect. 5, and we conclude the paper in Sect. 6.

2. Related Work

There have been several previous studies on implementing MwCAS operations [11]–[15].

Harris et al. [11] proposed the first practical algorithm for implementing an MwCAS operation without special hardware support, and it was used as a baseline in the subsequent work. Because our algorithm is also based on the Harris–Fraser–Pratt MwCAS algorithm, we explain the details later. Wang et al. [15] implemented a persistent MwCAS (PMwCAS) library based on the Harris–Fraser–Pratt algorithm to support MwCAS operations in persistent memory. Although their main contribution was persistency support of an MwCAS operation, their PMwCAS library can perform the original algorithm in volatile memory by setting a compile option. Therefore, we use the PMwCAS library as a comparison implementation in this paper.

Sundell [12] and Feldman et al. [13] proposed MwCAS algorithms with wait-free features. However, because their proposed methods are intended to be used in high-contention workloads such as a lock-free deque, the direction of improvement differs from that of the proposed method; also, the experimental results of Sundell and Feldman et al. showed that their algorithms do not outperform the Harris–Fraser–Pratt algorithm in a low-contention environment.

Guerraoui et al. [14] proposed the AOPT algorithm to reduce the number of CAS operations for each MwCAS operation. The AOPT algorithm embeds an MwCAS descriptor in target memory addresses, but does not reclaim it. This approach achieves a k-word CAS operation with $k + 1$ CAS operations in the ideal case (i.e., no conflicts occur), but it increases the cost of reading actual values. That is, if a certain thread reads some embedded descriptor, it must refer to the actual value by using the information in the descriptor. Moreover, embedded descriptors must be reclaimed in garbage collection, and reclamation of each descriptor requires k CAS operations. This means that the AOPT algorithm requires $2k + 1$ CAS operations in total, while the proposed method can perform a k-word CAS operation with $2k$ CAS operations in the ideal case.

Although the performance of MwCAS operations decreases in high-contention workloads, Wang et al. [15] and Arulraj et al. [10] proposed the new direction of using an MwCAS operation with lock-free indexes. For example, we implemented a lock-free queue by using our MwCAS library as a test, but it is three to five times slower than the implementation based on a standard CAS operation [6]; moreover, its throughput decreases as the number of threads increases. On the other hand, Wang et al. reported that their Bw-tree based on the PMwCAS library is comparable to the original one [8], and Arulraj et al. implemented BzTree by using the PMwCAS library and found that it outperformed Bw-tree. These results suggest that using MwCAS operations in relatively low-contention workloads such as lock-free indexes is useful for implementing practical multi-threading software.

3. Preliminaries

In this section, we introduce a basic linearization strategy for an MwCAS operation and Harris et al.'s CASN algorithm [11]. We use an algorithm in Fig. 1 as a CAS operation to explain MwCAS algorithms. Note that herein and in our implementation, we assume that all words are represented as 8-byte data, but our approach can be adopted with different word lengths.

3.1 Linearization Strategy

To linearize MwCAS operations, we make the following assumption.

Assumption 1. A set of all CAS candidate words W has a total order relation $\leq$.

In other words, we assume that all target words can be sorted in a unique order. This assumption can be easily achieved by using logical memory addresses because logical addresses (i.e., natural number) have a total order relation and each word is assigned to one of them. However, we do not recommend using logical addresses. Since the logical address of each word is determined at program execution, dynamic sorting of words arises for each MwCAS operation. As such dynamic sorting would degrade the entire performance, we recommend constructing another logical order for each data

```
Input: addr              // a target memory address
Input: expected          // an expected (old) value
Input: desired           // a desired (new) value
Output: word
1 word ← a read word from addr
2 if word = expected then
3     store desired into addr
4 return word
```

Fig. 1 An algorithm assumed herein to be a compare-and-swap (CAS) operation.

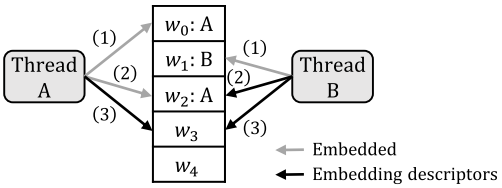

Fig. 2 Linearization based on embedding descriptors.

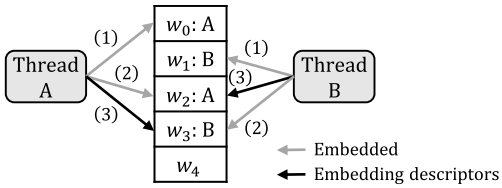

Fig. 3 Deadlock due to unsatisfied Assumption 1.

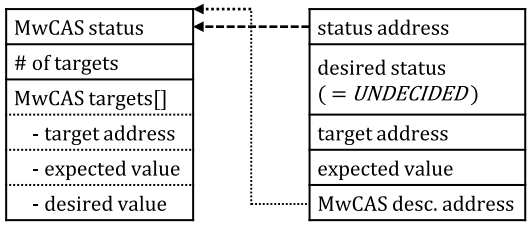

Fig. 4 Descriptors introduced in [11].

structure.

For example, we consider the case of BzTree [10]. BzTree is a lock-free extension of B$^+$-tree and so has a tree structure. In this case, we can easily order all the target words by using the following rules.

1. If the target words are in different nodes and the nodes are in different levels, we select words in the higher node as previous ones.
2. If the target words are in different nodes and the nodes are in the same level, we select words in the left node as previous ones.
3. If the target words are in the same node, we order words based on BzTree's node layout.

Because this logical order is determined before program execution, BzTree can maximize the performance of MwCAS operations.

Once the target words are ordered, the idea of linearization is less complicated. To swap multi-words atomically, Harris et al.'s CASN operation [11] embeds an MwCAS descriptor in each word address and swaps them after all the embedding has been completed (we describe the details later). That is, if two MwCAS operations have the same target word, they try to embed their own descriptors in the same memory address. Only one MwCAS operation can embed its descriptor successfully because embedding is performed by a CAS operation. This means that a winning MwCAS operation will be executed before the others and all the MwCAS operations are linearized in the same way.

Figure 2 shows an example of the linearization in MwCAS operations. Threads A and B have the three-words $\{w_0, w_2, w_3\}$ and $\{w_1, w_2, w_3\}$ as the MwCAS targets, respectively. In Fig. 2, we assume that the target words are arranged from top to bottom, and thus each MwCAS operation embeds its descriptors in this order. Because threads A and B have the same targets w_2 and w_3, they try to embed their descriptors in w_2 after embedding their first word (i.e., w_0 and w_1, respectively). However, because they use a CAS operation to embed their descriptors, only one thread (thread A in the figure) can embed it successfully. The losing threads (thread B in the figure) defer their MwCAS operations and assist the embedded one (thread A's MwCAS operation in the figure). When any thread completes the embedded MwCAS operation, the losing threads resume deferred embedding to complete their MwCAS operations.

On the other hand, if Assumption 1 is unsatisfied, then concurrent MwCAS operations may cause deadlock. In the case of Fig. 3, the target words are not linearized and threads A and B perform embedding in the orders of $w_0 \leq w_2 \leq w_3$ and $w_1 \leq w_3 \leq w_2$, respectively. Threads A and B successfully embed $\{w_0, w_2\}$ and $\{w_1, w_3\}$, respectively, but they find that the descriptors are already embedded in their final target (w_3 and w_2, respectively). Consequently, threads A and B try to assist each other's MwCAS operations and cause deadlock.

3.2 CASN Algorithm

We introduce the original MwCAS algorithm briefly to explain the benefits of the proposed method. Harris et al.'s CASN operation [11] is based on the above linearization strategy with two types of descriptor: MwCAS and restricted double-compare single-swap (RDCSS). MwCAS descriptors are used to linearize MwCAS operations as described above, and RDCSS descriptors assist the embedding MwCAS descriptors in lock-free concurrency control. Figure 4 shows the layouts of MwCAS and RDCSS descriptors.

An MwCAS descriptor has three components: 1) MwCAS operation status (SUCCEEDED, FAILED, or UNDECIDED), 2) the number of MwCAS targets, and 3) information about each MwCAS target (memory address and expected/desired values). MwCAS status begins with UNDECIDED and becomes either SUCCEEDED if embedding succeeds for every target or FAILED if any of the targets have changed. An MwCAS target region is implemented as a static array[†], and thus the number of targets is stored explicitly. Each MwCAS target is expressed as its memory address, expected (i.e., currently stored) value, and desired one. This information is used to swap each target in the CASN operations.

An RDCSS descriptor has three components: 1) the memory address of the corresponding status of an MwCAS

[†]We use a static array for the optimal performance. The maximum number of MwCAS targets is set by a compile option in our implementation.

Table 1 Control bits and corresponding data.

Control bits	Stored data (remaining 62 bits)
00	an actual value
01	an RDCSS descriptor
10	an MwCAS descriptor

descriptor, 2) the expected MwCAS status (*always UNDECIDED*), and 3) information for MwCAS descriptor embedding[†]. As the name implies, an RDCSS operation performs double-compare to swap a single word. In the case of the CASN algorithm, an RDCSS operation compares both the MwCAS status and the target address where an MwCAS descriptor is to be embedded. First, an RDCSS operation checks whether its target address has an expected value, and if so its RDCSS descriptor is embedded there. Second, the status of a corresponding MwCAS operation is checked, and an RDCSS operation either embeds an MwCAS descriptor in the target address if the status is UNDECIDED or reverts if the MwCAS operation is already SUCCEEDED or FAILED.

Note that the CASN algorithm uses the last two bits in each word to distinguish descriptors from actual values. Table 1 lists the control bits and corresponding stored data. Two zero bits mean that the target memory address contains an actual (i.e., expected or desired) value, and thus any thread can read its value and embed a descriptor for an RDCSS or MwCAS operation. If either bit is set to one, then the target address already contains a descriptor and a concurrent MwCAS operation is being processed. In such a case, the thread reads the remaining 62 bits and casts them in the pointer of an MwCAS/RDCSS descriptor. Because each descriptor has sufficient information to execute the same MwCAS/RDCSS operation, a thread performs the embedded operation to assist it.

4. MwCAS Algorithm without Garbage Collection

In this section, we begin by explaining the problems associated with Harris et al.'s CASN algorithm and ideas for constructing a practical implementation. We then describe the proposed MwCAS algorithm and its read algorithm in detail.

4.1 Problems with the CASN Algorithm

We can implement an MwCAS operation by using the CASN algorithm [11], but there are two problems regarding achieving high performance: 1) garbage collection and 2) redundant CAS operations.

First, the CASN algorithm requires garbage collection for MwCAS descriptors (and RDCSS descriptors with naïve implementations). Every descriptor is embedded in a corresponding target address, and it may be read by other threads. That is, even if a certain MwCAS operation has finished, any thread may access its descriptor. To prevent threads from reading freed memory addresses, garbage collection must be implemented. For instance, the PMwCAS library [15] uses epoch-based garbage collection in its implementation[††]. However, such garbage collection causes additional processes, such as the creation of epochs and the freeing of garbage descriptors, and this reduces the overall performance of MwCAS operations in terms of both throughput and latency.

Second, the CASN algorithm requires at least three CAS operations to swap a single word. Even if there is no concurrent MwCAS operation, the CASN algorithm must embed both RDCSS and MwCAS descriptors. Because CAS operations must be used for embedding, the maximum throughput is limited even in a low-contention environment. Moreover, if some MwCAS operations fail because of concurrency issues, they must restore old values by using CAS operations and restart from the embedding RDCSS descriptors. This increases the total number of executed CAS operations and degrades the performance.

4.2 Findings for Improving MwCAS Implementation

These problems motivated us to implement an MwCAS algorithm without garbage collection. To explain the idea of excluding garbage collection, we first describe the multi-threading assistance in lock-free algorithms in detail.

In lock-free algorithms, the assistance of a target operation by multi-threads is adopted in many data structures, such as Michael's lock-free queue [6] and deque [7]. To reduce conflict-inducing periods, lock-free algorithms usually allow some threads to read an intermediate state. If a certain thread reads an intermediate state, it first try to complete the found state before processing its own operation. In this paper, we refer to such completing intermediate states by other threads as *multi-threading assistance*. For example, in the case of a lock-free queue, some threads may read an *old* (i.e., intermediate) tail node because two CAS operations are required to complete an enqueue. An enqueue operation swaps the (null) pointer in a current tail node for a new tail address and then updates a tail pointer. If a certain thread reads an old tail node, it tries to update a tail pointer to the latest one, which is reachable from the old tail node, and continues its own processing. Such multi-threading assistance actively maintains queue consistency and improves performance.

However, in the case of MwCAS operations, the performance improvement by using multi-threading assistance is questionable. The assistance in a lock-free queue is to perform a CAS operation only once, whereas the CASN algorithm requires multiple CAS operations to complete its assistance. For example, we consider the case of Fig. 5. Threads A and B perform each three-word CAS operation, and thread A has already embedded its descriptor and com-

[†]The addresses of an MwCAS descriptor and its status are the same in Fig. 4, but this depends on the implementation.

[††]https://github.com/microsoft/pmwcas

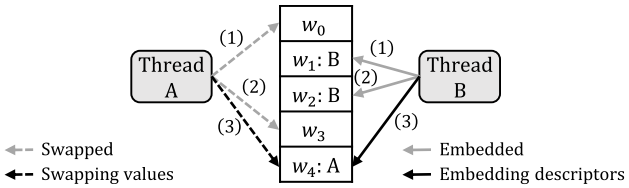

Fig. 5 Wasteful assistance of an MwCAS operation.

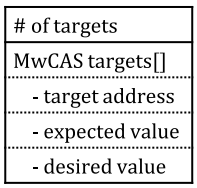

Fig. 6 An MwCAS descriptor for the proposed method.

pleted swapping two words. Thread B is embedding its descriptor, and it finds the descriptor of thread A in the embedding of the last word. Thus, thread B performs CAS operations from the *first word* (i.e., w_0) to assist thread A's MwCAS operation, but it must fail because these words have been swapped already. Thread A probably swaps its last word while thread B is wasting its time with such wasteful swapping. Moreover, the latency of thread B's MwCAS operation increases because it is required to try to *swap all the words* to finish its assistance.

Furthermore, the necessity to use garbage collection is caused by multi-threading assistance. When a certain thread assists any MwCAS operation, it must read a corresponding descriptor to swap target data. In other words, if we do not adopt multi-threading assistance, then there is no need to read the descriptors themselves because we can use the control bits to determine whether the descriptors are embedded. This enables us to maintain MwCAS descriptors in the rule of the RAII (resource acquisition is initialization) technique [16] and remove garbage collection from our implementation.

4.3 Proposed Algorithms

Based on these findings, we propose an MwCAS algorithm without multi-threading assistance and garbage collection. Figure 6 shows the layout of an MwCAS descriptor for the proposed method. Our descriptor does not have the status of a corresponding MwCAS operation because our algorithm does not use an RDCSS operation and its descriptor. The rest of the components are the same as those of the original descriptor in Fig. 4. Note that our algorithm uses only one bit as the control bit because we do not use RDCSS operations.

Figure 7 shows the proposed MwCAS algorithm, which has two phases: embedding a descriptor (lines 3–9) and completing an MwCAS operation (lines 10–14). First, our algorithm embeds an MwCAS descriptor to reserve the memory addresses of the target words. Unlike the CASN

```
   Input: desc                          // an MwCAS descriptor
   Output: boolean                      // true if MwCAS succeeds
1  d_addr ← a memory address of desc with a control bit
2  success ← true
   // Phase 1: embedding a descriptor
3  foreach t ∈ desc do                  // t is an MwCAS target
4      do
5          cur_w ← CAS(t.addr, t.expected, d_addr)
6      while cur_w is a descriptor
7      if cur_w ≠ t.expected then
8          success ← false
9          break
   // Phase 2: completing MwCAS
10 foreach t ∈ desc do
11     if success = true then
12         CAS(t.addr, d_addr, t.desired)
13     else
14         CAS(t.addr, d_addr, t.expected)
15 return success
```

Fig. 7 MwCAS algorithm.

```
   Input: addr                          // a target memory address
   Output: word
1  do
2      word ← an atomically read word from addr
3  while word is a descriptor
4  return word
```

Fig. 8 Algorithm to read a word from an MwCAS target address.

algorithm, the proposed method uses a *spinlock* to embed a descriptor (lines 4–6). The algorithm reads the current value of a target word (*cur_w*) as the result of a CAS operation (line 5). If there is another MwCAS descriptor in *cur_w*, the algorithm retries embedding to wait for another MwCAS operation to finish its processing. If *cur_w* is not any descriptor, we check whether the value of *cur_w* is the expected one (lines 7–9). If *cur_w* is not an expected value, the MwCAS operation fails because other threads have already updated a target word (line 8). Then, our algorithm completes its MwCAS operation according to its status (lines 10–14). If all the embedding has succeeded, we update the target words to the desired values (lines 11–12). If any target words have been modified by other threads, we reset those target words to the expected (i.e., originally contained) values (lines 13–14). Finally, the algorithm returns the status of whether the MwCAS operation has succeeded (line 15).

Note that we also propose an algorithm in Fig. 8 to read an MwCAS target field. When a certain memory address is a target of MwCAS operations, a specific read operation must be used because the memory address may contain an MwCAS descriptor. Our read algorithm also uses a spinlock to avoid reading any descriptor (lines 1–3). If a read word is not a descriptor, the algorithm returns it as a result (line 4).

5. Experiments

In this section, we evaluate the proposed method by using synthetic datasets. Table 2 summarizes the experimental environment.

Table 2 Experimental environment.

Item	Value
CPU	Intel(R) Xeon(R) Gold 6258R (two sockets)
RAM	DIMM DDR4 (Registered) 2933 MHz (16GB × 12)
OS	Ubuntu 20.04.2 LTS
Compiler	GNU C++ ver. 9.3.0

We implemented the proposed method and a program for benchmarking[†] in C++, and we developed the benchmarking program for MwCAS operations with the following features.

- It prepares one million words W as candidates for MwCAS operations. All the words are 8-byte integers and initialized by zeros. Note that all the words are distributed to all NUMA nodes for NUMA architecture.
- Each MwCAS operation selects the specified number of words randomly. Note that we use a parameter α to control the skew of the k-th word to be selected according to Zipf's law [17]:

$$f(k;\alpha,|W|) = \frac{1/k^{\alpha}}{\sum_{n=1}^{|W|} 1/n^{\alpha}}. \quad (1)$$

- Every MwCAS operation reads the current value of each target word and adds one atomically. If an MwCAS operation fails, it *continues* its own processing until it succeeds.
- It executes one billion MwCAS operations in total. When it uses multiple threads for benchmarking, the MwCAS operations are distributed to each thread so that the numbers of executions are approximately equal.

We ran the benchmarking program five times and measured the average throughput or latency. The following graphs contain no error bars because the measurement results are sufficiently stable. Note that we used all the threads (112 threads) to measure each latency and performed sampling to compute percentile latency, and thus we use one and ninety-nine percentile latency instead of the minimum and maximum latency, respectively.

5.1 Performance Degradation Due to Garbage Collection

First, we demonstrate the improvement in performance by removing garbage collection. To compare the proposed method with implementations with garbage collection, we use the PMwCAS library [15] and our implementation[††] as the reproduction of Harris et al.'s CASN algorithm [11] and Guerraoui et al.'s AOPT algorithm [14], respectively. In the following, we use low-contention ($\alpha = 0$) and high-contention ($\alpha = 1$) workloads. Note that we omit cases in which the number of target words is not two because the results show a similar trend.

[†]https://github.com/dbgroup-nagoya-u/mwcas-benchmark
[††]https://github.com/dbgroup-nagoya-u/mwcas-aopt

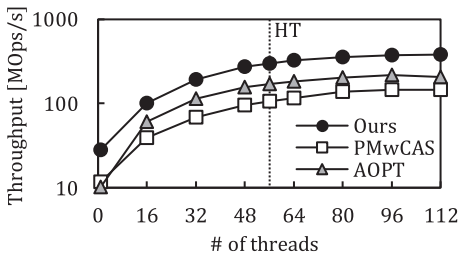

Fig. 9 Comparison with the existing methods on throughput of double-word CAS in a low-contention workload.

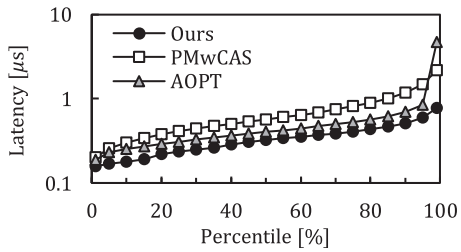

Fig. 10 Comparison with the existing methods on percentile latency of double-word CAS in a low-contention workload.

Figures 9 and 10 show the throughput and latency, respectively, of double-word CAS in a low-contention workload. These results show that the proposed method is two to three times faster than the existing MwCAS algorithms in terms of both throughput and latency. In particular, all the latency of the proposed method remains less than 1 μs, while those of the PMwCAS library and the AOPT algorithm exceed 2 and 4 μs, respectively, despite the low-contention workload. The PMwCAS library prepares a pool of MwCAS descriptors to improve its performance, but garbage collection is executed when all of them have been consumed. Our implementation of the AOPT algorithm uses jemalloc[†††] as an efficient memory allocator and releases expired descriptors in the background, but the reclamation of embedded descriptors is performed in the foreground for garbage collection. Such processing related to garbage collection increases the tail percentile latency of the existing methods and also degrades their throughput.

Figures 11 and 12 show the throughput and latency, respectively, of double-word CAS in a high-contention workload. These results show that the performance of the proposed method remains superior to those of the existing methods. In other words, there is no effect of multi-threading assistance at all for MwCAS operations. In a high-contention workload, because each thread selects the same target word frequently, they also help each MwCAS operation to complete quickly. However, the experimental results indicate that such multi-threading assistance does not provide a clear performance improvement but rather degrades the overall performance because of the cost of garbage collection.

[†††]https://github.com/jemalloc/jemalloc

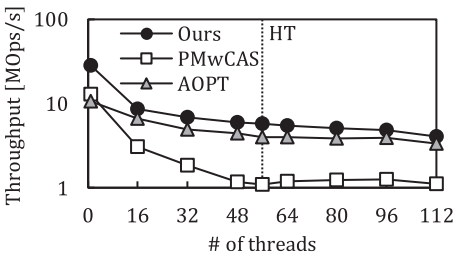

Fig. 11 Comparison with the existing methods on throughput of double-word CAS in a high-contention workload.

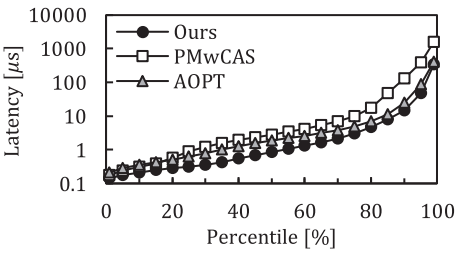

Fig. 12 Comparison with the existing methods on percentile latency of double-word CAS in a high-contention workload.

5.2 Comparison with CAS Operation

We compare the proposed method with a single CAS operation (i.e., the `std::atomic::compare_exchange_weak` function in C++) to evaluate the fundamental performance of the proposed method. We continue to use the same low/high-contention workloads in this experiment. Note that we also show the results of the PMwCAS library but omit their explanation because these trends are similar to the previous ones.

Figures 13 and 14 show the throughput and latency, respectively, of single-word CAS in a low-contention workload. Figure 13 shows that the proposed method is twice as slow than a CAS operation. However, we consider these results to be reasonable because the proposed method requires at least two CAS operations because of descriptor embedding. Because Fig. 14 also shows that the percentile latency of the proposed method is comparable to that of a CAS operation, we conclude from these results that the proposed method achieves approximately ideal performance in a low-contention environment.

Figures 15 and 16 show the throughput and latency, respectively, of single-word CAS in a high-contention workload. These results indicate that the proposed method becomes approximately ten times slower than a CAS operation with many threads in a high-contention environment. This is the fundamental limitation of an MwCAS operation based on descriptor embedding. The proposed method (and the Harris–Fraser–Pratt CASN algorithm) embeds an MwCAS descriptor to swap each word, and it increases the number of CAS conflicts. Consequently, the performance of the proposed method becomes sensitive to skew.

Note that the AOPT algorithm outperforms the proposed method in the high-contention workload with single-word CAS, but this result suggests that multi-threading assistance for lock-free algorithms is only effective for quite simple data structures. In the case of single-word CAS, the multi-threading assistance of the AOPT algorithm is only performed to update a current status of a certain MwCAS descriptor. That is, if some threads find an UNDECIDED descriptor, they try to update its status to SUCCEEDED to assist the found MwCAS operation. This simple assistance is similar to the multi-threading assistance in lock-free queue and deque implementations. However, when we

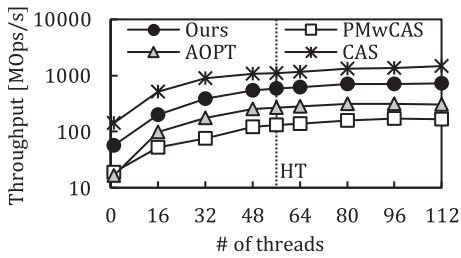

Fig. 13 Comparison with CAS operation on throughput of single-word CAS in a low-contention workload.

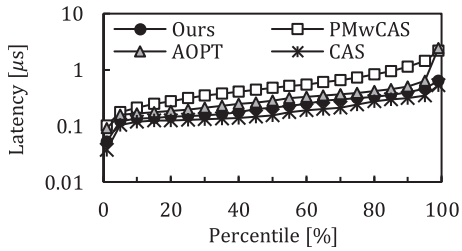

Fig. 14 Comparison with CAS operation on percentile latency of single-word CAS in a low-contention workload.

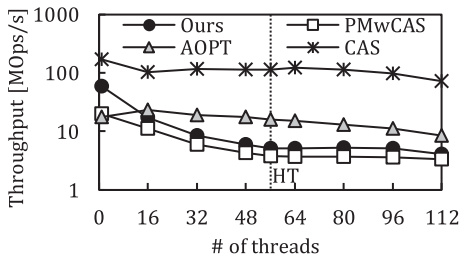

Fig. 15 Comparison with CAS operation on throughput of single-word CAS in a high-contention workload.

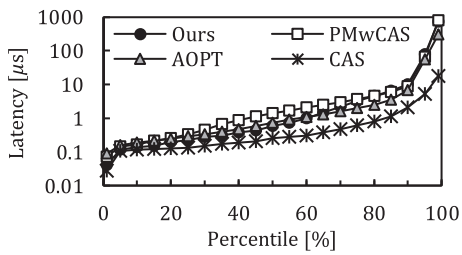

Fig. 16 Comparison with CAS operation on percentile latency of single-word CAS in a high-contention workload.

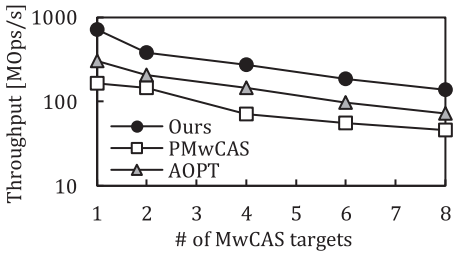

Fig. 17 Throughput of multi-word CAS in a low-contention workload.

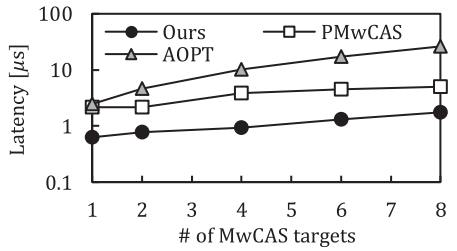

Fig. 18 Ninety-nine percentile latency of multi-word CAS in a low-contention workload.

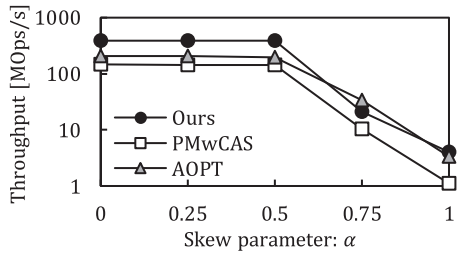

Fig. 19 Throughput of double-word CAS with various values of the skew parameter.

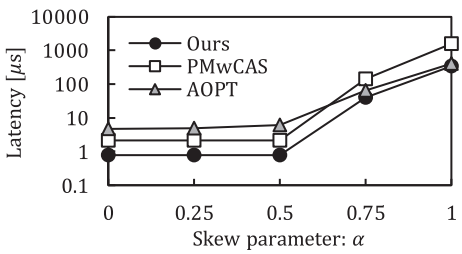

Fig. 20 Ninety-nine percentile latency of double-word CAS with various values of the skew parameter.

swap two words by using MwCAS operations, Figs. 11 and 12 show that the AOPT algorithm is inferior to the proposed method despite the high-contention workload. This suggests that multi-threading assistance has no effect if it requires complex procedures such as multiple CAS operations.

5.3 Effects of Parameters

We investigate the effects of two parameters: the number of MwCAS targets and a skew parameter α. In this experiment, we use all the threads to measure both throughput and latency.

Figures 17 and 18 show the throughput and ninety-nine percentile latency, respectively, of MwCAS with various numbers of words in a low-contention workload. These results show that the performance of the proposed method deteriorates linearly as the number of words is increased. These results are expected because the ideal (i.e., when no conflicts occur) computation time of the proposed method is linear with the number of target words.

Figures 19 and 20 show the throughput and ninety-nine percentile latency, respectively, of double-word CAS with various values of the skew parameter. These results show that the performance of the proposed method remains with low-contention workloads but decreases drastically when there are many conflicts. The main cause of the performance deterioration is the long execution period of MwCAS operations. Because an MwCAS operation requires all current (i.e., expected) values of the target words as its input, a program needs a preparation step before MwCAS execution. Because every step (preparation, embedding a descriptor, and completing an MwCAS operation) is a conflict-inducing period, the performance is easily reduced by skew.

6. Conclusion

In this paper, we proposed an algorithm for an MwCAS operation for low-contention workloads. In the proposed algorithm, a spinlock is used to linearize concurrent MwCAS operations, and garbage collection has been removed from its implementation. The experimental results showed that our approach improves both throughput and latency in a low-contention environment and retains comparable performance in a high-contention environment. Furthermore, these results imply that multi-threading assistance has no effect on complicated lock-free algorithms in practice.

In future work, we intend to extend the proposed method to persistency support such as PMwCAS [15] and implement lock-free indexes by using our MwCAS library. MwCAS operations have performance problems in a high-contention environment, but they are useful for implementing complex lock-free data structures. Because it is even more difficult to implement such data structures in persistent memory, persistency support will be a practical extension. We also intend to implement lock-free indexes such as Bw-tree [8] and BzTree [10] to investigate their performance in detail. Although the present experimental results showed that multi-threading assistance has no effect on MwCAS operations, the lock-free indexes also adopt multi-threading assistance to perform their structure modification. Because this may degrade performance, particularly with respect to latency, we will investigate them by implementing such indexes.

Acknowledgments

This paper is based on results obtained from a project, JPNP16007, commissioned by the New Energy and Industrial Technology Development Organization (NEDO). In addition, this work was supported partly by KAKENHI (16H01722, 20K19804, and 21H03555).

References

[1] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," Proc. SOSP, pp.18–32, Nov. 2013.
[2] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots," Proc. ICDE, pp.195–206, 2011.
[3] Y. Mao, E. Kohler, and R.T. Morris, "Cache craftiness for fast multicore key-value storage," Proc. EuroSys, pp.183–196, April 2012.
[4] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," Proc. ICDE, pp.38–49, 2013.
[5] Y. Sun, G.E. Blelloch, W.S. Lim, and A. Pavlo, "On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes," PVLDB, vol.13, no.2, pp.211–225, Oct. 2019.
[6] M.M. Michael and M.L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," Proc. PODC, pp.267–275, May 1996.
[7] M.M. Michael, "CAS-based lock-free algorithm for shared deques," Proc. Euro-Par, pp.651–660, 2003.
[8] J.J. Levandoski, D.B. Lomet, and S. Sengupta, "The Bw-tree: A B-tree for new hardware platforms," Proc. ICDE, pp.302–313, 2013.
[9] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D.G. Andersen, "Building a Bw-tree takes more than just buzz words," Proc. SIGMOD, pp.473–488, May 2018.
[10] J. Arulraj, J. Levandoski, U.F. Minhas, and P.A. Larson, "BzTree: A high-performance latch-free range index for non-volatile memory," PVLDB, vol.11, no.5, pp.553–565, Jan. 2018.
[11] T.L. Harris, K. Fraser, and I.A. Pratt, "A practical multi-word compare-and-swap operation," Proc. DISC, pp.265–279, 2002.
[12] H. Sundell, "Wait-free multi-word compare-and-swap using greedy helping and grabbing," Int. J. Parallel Prog, vol.39, no.6, pp.694–716, 2011.
[13] S. Feldman, P. LaBorde, and D. Dechev, "A wait-free multi-word compare-and-swap operation," Int. J. Parallel Prog, vol.43, no.4, pp.572–596, 2015.
[14] R. Guerraoui, A. Kogan, V.J. Marathe, and I. Zablotchi, "Efficient multi-word compare and swap," Proc. DISC, pp.4:1–4:19, 2020.
[15] T. Wang, J. Levandoski, and P.A. Larson, "Easy lock-free indexing in non-volatile memory," Proc. ICDE, pp.461–472, 2018.
[16] B. Stroustrup, The C++ Programming Language (4th Edition), Addison-Wesley Professional, 2013.
[17] G.K. Zipf, Selected studies of the principle of relative frequency in language, Harvard University Press, 1932.

Kento Sugiura is an Assistant Professor in the Graduate School of Informatics, Nagoya University. He received B.S., M.S., and Ph.D. degrees from Nagoya University in 2013, 2015, and 2018, respectively. His research interests include data stream processing, uncertain data management, and spatiotemporal data processing. He is a member of DBSJ, IPSJ, and ACM.

Yoshiharu Ishikawa is a Professor in the Graduate School of Informatics, Nagoya University. His research interests include spatio-temporal databases, mobile databases, scientific databases, data mining, and Web information systems. He is a member of the Database Society of Japan, IPSJ, IEICE, JSAI, ACM, and the IEEE Computer Society.