Simple yet Fast Lock-Free Atomic Shared Pointers

Jörg P. Schäfer[1]

[1]Inst. of Transportation Systems, *German Aerospace Center*, joerg.schaefer@dlr.de

Abstract

CPU's don't increase in speed anymore, as Moore's Law has claimed for so long. Although, "the free lunch is over" (Herb Sutter), parallel algorithms can gain more throughput and reduce latency, which is crucial to complex real-time applications like audio and video processing, robotics, or real-time sensor data processing in embedded hardware. Parallel algorithms, however, come with the price of concurrency and synchronization. For example, the priority-inversion is a problem, where low-priority threads can block high-priority threads due to locking data structures used by both threads. Lock-free data structures, on the other hand, use atomic CPU instructions to avoid these problems. They, however, are hard to implement and even harder to prove correct.

Atomic shared pointers have been proposed as a (part of a) solution to making lock-free algorithms easier to write and verify. Since they are a fundamental tool in the toolbox of parallel algorithms, their run-time performance has a huge trailing impact. So far, there is just a hand full of existing implementations to atomic shared pointers. This work contributes a simple implementation to atomic shared pointers, a proof of its correctness, and an extensive performance evaluation in comparison to other implementations showing that it is at least competetive and even outperforms others in several use-cases.

Keywords: Concurrency, Algorithm, Atomic Shared Pointers

1 Introduction

Real-time applications such as in robotics or video and audio processing increase their workload and their processing capabilities by using multiple threads. In such applications, there are often multiple threads running with different priorities, e.g., at least one low-priority and one high-priority thread. Such threads usually communicate via common variable and data structures. To avoid race-conditions, most of these data structures ensure thread-safety using exclusive locking variables (*mutexes*).

Using a mutex for synchronization comes with the priority inversion problem, where low priority threads slow down high priority threads by locking such a common mutex: When the low priority thread is sent to sleep while holding the lock, the high priority thread needs to wait until the low priority thread is finished. This situation is exactly, what real-time applications need to avoid.

On the other hand, modern CPU's provide atomic operations on single (or double) words, e.g., exchange and compare-and-swap (*CAS*) to name the most important ones. Several simple data structures have already been implemented *lock-free*, i.e., without using mutexes and instead relying on these atomic instructions. However, this area of research is an open field because it is particularly hard to develop even simple data structures lock-free [7, 15, 11]. Several new problems arise, one of which is the ABA problem, where concurring threads might change an *atomic variable* forth and back to the value we last saw, making us believe that nothing happened during our last visit of this atomic variable. This false assumption might lead to algorithms that are not correct in this unlikely situation.

Herb Sutter proposed to use lock-free atomic shared pointers [14, 13], as they solve several problems including the ABA problem. In his proposal, he could show that a singly connected list is implementable in a few lines of (C++) code without any further ado. At that time, however, no implementation for atomic shared pointers existed.

A *shared pointer* is a concept in programming languages, which usually leave memory management to the language's user. Here, shared pointers are smart pointers that help managing the lifetime of dynamically allocated memory and the object that resides in there. When a shared pointer is created to manage the lifetime of a dynamically allocated object, it takes ownership of it, i.e., the last living copy is responsible for its proper destruction. From here on, we focus on C++ (up to C++23) as programming language.

The ubiquitous way to implement shared pointers is by additionally allocating a *control block* for a particular object, which contains a reference counter. Thus, shared pointers referencing the same object also share the same control block including this shared reference counter. Copying a shared pointer increments this reference counter. On the other hand, when a shared pointer dereferences its object, e.g. by being destroyed or being made referencing another object, it needs to decrement its reference counter. Upon reaching zero, the object destruction and memory deallocation also needs to take place.

Now, when a shared pointer dereferences an object, it needs to write to at least two different memory addresses: the pointer to the control block and the reference counter within the control block, which cannot be done atomically in most of the current CPU's.[1] One way to achieve atomic operations on shared pointers is guarding them with an exclusive locking variable. At least GNU-C++ 12.3.0 and Clang 14.0.0 use this approach for the implementation of `std::atomic<std::shared_ptr>`.

1.1 Contribution

Now that most of the important terms have been placed and put into relation, Section 1.2 provides a brief overview of the history and related work regarding atomic shared pointers. Section 2 contributes a new split-reference counting based algorithm for lock-free atomic shared pointers that allows negative counters.[2] Since that is unintuitive, Section 3 provides a proof for its correctness. Section 4 provides an evaluation and comparison of the proposed implementation against existing implementations regarding throughput in various scenarios.

1.2 Related Work

This millennium started with a "fundamental turn toward concurrency in software" [12]. A crucial aspect in writing such multi-threaded applications is the synchronization of multiple threads. The problem that comes with it starts at a very low level: Scott Meyers already mentioned lots of race conditions that can occur, when the programming language is not designed to be used for multi-threaded applications. In particular, he mentioned a series of problems that can occur even in very sim-

[1] While TSX enables transactions on CPU-level, substantial reasons exist, not to use it [9].

[2] Source code: `https://github.com/frygge/atomic_shared_ptr`.

ple C++ (up to version C++-03) code due to compiler optimizations [10].

For C++-11, the ISO C++ committee [1] thoroughly introduced threads in the language, including a memory model [2] to avoid these problems. This standard also introduced the `atomic` class, that guarantees atomic access to object, though mostly with a hidden lock (`mutex`). In some cases (e.g. for up to 64 Bit primitive types such as an `integer`), the `atomic`-implementation uses atomic CPU instructions instead of locks, which yields lock-free algorithms and data structures.

As it turned out, not only is the design of lock-free data structures a very hard problem for simple data structures already [7], even lots of reviewed and published proposals for lock-free data structures and algorithms have been proven incorrect [11]. In 2014, Herb Sutter proposed atomic shared pointers [13, 14] as a basic building block for lock-free data structures, as they automatically would solve several problems, including the well-known ABA-problem [4].

To that, Timur Doumler held a talk about shared pointers, in which he compared three existing shared pointer implementations against each other (and against the lock-based standard implementation) [5, 6]. One of these atomic shared pointer implementations was a proof of concept implementation provided by Anthony Williams [16], who took a leading part in the C++ committee enabling multi-threaded applications in the C++-11 standard [15]. Though, his implementation uses 128 Bit `atomic` variables, probably hoping at the time of writing that C++ would support atomic operations on double words. But, compiler builders decided against that, probably in favor of being backwards compatible to some AMD x86-64 CPU's produced until 2008, which do not support these instructions. Hence, Anthony's implementation is technically not lock-free, but can be easily made to be one. The most competitive implementation regarding throughput comes from Facebook in their open-source library Folly [8].

While all implementations above and the one proposed in this work are based on a *split reference counting* technique, a talk has been announced for the CppCon 2023, in which a new approach based on hazard pointers and deferred reclamation will be shown [3].

2 Algorithm

The proposed implementation of (atomic) shared pointers is very similar to the ones that already exist: A shared pointer instance is a pointer to a control block, which contains (a pointer to) the actual data as well as a reference counter (aka *usage counter*). Copying the shared pointer increments the usage counter while destroying a shared pointer instance (or letting it point to something else) decrements the usage counter. The last instance decrementing the usage counter is responsible for the destruction of the control block and the actual object.

In our proposed implementation, each shared pointer instance also has a *local (temporary) counter*. Furthermore, the control block has a usage counter and a global (temporary) counter. Algorithm 1 provides pseudo code for the data structures.

Having a global temporary reference counter enables always having a correct usage counter, which counts the amounts of atomic and non-atomic instances referencing an object. Incrementing the local counter by one thread before accessing the control block provides a hint to concurrent threads that the control block is still in use, so they will not destroy it before the first thread could increase the usage counter. Henceforth, I assume that the reader is familiar with the basic implementation details of shared pointers.

Algorithm 1 Data Structures

```
struct paired_counter {
    signed tmp
    unsigned ref
}
struct ctrl_block<T> {
    atomic<paired_counter> pc
    T* data
}
struct counted_ptr<T> {
    signed tmp_ref
    ctrl_block<T>* pctrl
}
struct sptr<T> {
    counted_ptr<T> cptr
}
struct atomic_sptr<T> {
    atomic<counted_ptr<T>> cptr
}
```

Our implementation needs to simultaneously read and update the local counter and a pointer to the control block in atomic instances. To that, we pack a pointer (which requires only 48 Bits in current systems) and a 16 Bit counter in a 64 Bit word. This trick avoids the necessity of atomic operations on double words (i.e., 128 Bit). Changing this detail would not affect the proof of correctness in Section 3.

Analogously, our implementation needs to read and update a pair of counters from the control block. Here, we use a similar trick, splitting a word in two 32 Bit counters.

It is worth mentioning, that the counter is placed into the packed pointer such that integer overflows do not change the value of the pointer. In particular, this allows the counter to be signed. For the same reason, the global temporary counter in the paired counter can be signed. This implementation detail is crucial to some of the optimizations over the other existing implementations.

Similar to the family of atomic arithmetic operations (e.g., `fetch_add`, `fetch_xor`, etc), such operations can be performed on `counted_ptr` and `paired_counter`, except, that the operation shall only be performed on the counter of the `counted_ptr` or simultaneously on both counters of the `paired_counter` using a pair of arguments. The details for the implementation of these structures are omitted here, but can be found in the source code. For example, `cptr.fetch_sub( 2 )` would subtract 2 from the counter of the counted pointer `cptr` and then return the old value of the `counted_ptr` object. Similarly, `pc.fetch_add( 3, 5 )` would atomically add 3 to `tmp` and 5 to `ref` of the `paired_counter` object `pc`.

2.1 Notation and assumptions

This subsection declares some notation and defines the terms *atomic instance*, *non-atomic instance*, *temporary instance* that are used throughout the rest of this work.

Objects in memory that are referenced by shared pointers (also called *shared objects*) are denoted via letters $p, q, r, \ldots$.

Definition 1 (Shared Pointer) *A (non-atomic) shared pointer is an instance of a structure to access an object in memory and manage its lifetime. The shared object referenced by multiple shared pointers will be destroyed immediately when the last instance stops referencing it (i.e., by being destroyed itself or being updated to not reference the shared object anymore).*

Non-atomic shared pointers can be read concurrently. In particular, multiple threads can make copies of a shared pointer concurrently without external synchronization.

Note, that a non-atomic shared pointer is not necessarily encapsulated into a `sptr<T>` structure. For example, Line 4 of Algorithm 3 moves a non-atomic instance into an atomic instance (after which, the non-atomic instance does not exist anymore), simultaneously retrieving the old pointer of the atomic instance, making it a non-atomic instance.

Definition 2 (Atomic Shared Pointer) *An atomic shared pointer is an instance of a structure that provides atomic access to a shared pointer via the operations `load`, `store`, `exchange`, and `cas` (compare-and-swap). These instructions are similar in their semantic interpretation as the corresponding CPU instructions in modern CPU's, except that they work on shared pointers including the lifetime management of the referenced objects.*

Definition 3 (Temporary Shared Pointer) *A temporary instance of a shared pointer is a quasi-instance of a non-atomic shared pointer that only lives temporarily during the execution of the operations of atomic shared pointers.*

Atomic and non-atomic shared pointers referencing an object p are denoted by a^p and s^p, respectively.

Being referenced by shared pointers, each shared object p has a global reference counter U^p (or shortly *usage counter*)[3] and a global temporary reference counter T^p (or shortly *global counter*).

Each atomic and non-atomic instance a^p and s^p has a *local temporary reference counter* (or shortly *local counter*) denoted by l_a^p and l_s^p, respectively.

We denote the number of temporary instances referencing a shared object p by J^p and the total number of (atomic, non-atomic, and temporary) instances referencing p by I^p.

[3]As we will see in Lemma 1, U^p always counts the global number of atomic and non-atomic shared pointer instances for this particular object p; hence the short name *usage counter*.

From here on, we assume correct usage of shared pointers. In particular, this requires that non-atomic shared pointers are either read and written to by exactly one thread or accessed read-only by multiple concurrent threads. In the end, concurrent write access to shared pointers is the essence of atomic shared pointers.

2.2 Operation: load

To understand the wording used in the proof in Section 3, this section describes the behaviour of the `load`-operation in more detail.

Algorithm 2 Atomic Shared Pointer: **load**

```
1 load( atomic_sptr<T>& a ): sptr<T>
2     cptr = a.cptr.fetch_add( 1 )
3     if( cptr.pctrl == NIL )
4        return NIL
5     cptr.pctrl->pc.fetch_add( 1, 1 )
6     return { 0, cptr.pctrl }
```

The `load`-operation (Algorithm 2) creates a non-atomic copy s^p of the *atomic instance* a^p stored in the atomic shared pointer `a` and returns it to the caller. To that, it first creates a temporary instance in Line 2 indicated by incrementing the local counter l_a^p. If that instance is empty (aka NIL, i.e., it does not point to any object), it is returned as such. Otherwise, Line 5 converts the just retrieved temporary instance (stored in `cptr`) into a non-atomic instance s^p by balancing the temporary (local and global) counters l_a^p and T^p and simultaneously incrementing the usage counter U^p. Then, Line 6 returns the non-atomic instance s^p, stored in `cptr` to the user with a local counter $l_s^p = 0$.

2.3 Operation: store

The `store`-operation (Algorithm 3) creates a copy of the non-atomic instance stored in its

Algorithm 3 Atomic Shared Pointer: **store**

```
1 store( atomic_sptr<T>& a, const sptr<T>& s )
2     if( s.cptr.pctrl != NIL )
3         s.cptr.pctrl->cptr.fetch_add( 0, 1 ) // 1., acquire globally
4     old_cptr = a.cp.exchange({ 0, s.cptr.pctrl }) // 2., exchange pointers
5     if( old_cptr.pctrl != NIL )
6         release( old_cp.pctrl, { old_cptr.tmp_ref, 1 }) // 3. release old pointer

8 store( atomic_sptr<T>& a, sptr<T>&& s )
9     s.cptr = a.cptr.exchange( s.cptr ) // a direct swap if v is a temporary variable
```

parameter `s` (Line 3) and converts the copy into an atomic instance by putting into the atomic variable `a` (Line 4). Simultaneously, the `exchange` in Line 4 converts the old atomic instance stored in `a` into a non-atomic instance and stores it inside the variable `old_ctpr`. Since `store` is supposed to simply overwrite the old value of the atomic variable, Line 6 releases the old instance, which might be the last one, in which case the shared object and its control block is being destroyed through the function `release`.

The second implementation in Line 7 uses an *R-value reference*, which is specific to C++. For the pseudo-code, however, the only important thing to know is, that it is a reference, for which the value may be destroyed after the execution of this operation. Here, however, the parameter will simply be made to point to the old object, thus the caller is responsible for its destruction.

2.4 Operation: exchange

The `exchange`-operation (Algorithm 4) atomically swaps the atomic instance a^p stored in `a` with the non-atomic instance s^q stored in the parameter `s` in Line 4 and 8, respectively. To that, the implementation in Line 1 first needs to create a copy of s^q (incrementing the usage counter) in Line 3. The second implementation in Line 7 does not need to do this step, as it can move the non-atomic instance s^q from `s` to `a`. However, it needs to prevent the structure behind `s` to release s^p, which is done by simply resetting the content of `s`.

2.5 Operation: Compare-and-Swap

The `cas`-operation (short for compare-and-swap) tests if the shared object p referenced by the atomic instance a^p in `a` is the same as the shared object q referenced by s^q in `expected`. If that is the case, it puts a copy of the instance s^r stored in `desired` into `a` (similar to `store` in Algorithm 3). If the comparison failed, the `cas`-operation provides a non-atomic copy of the atomic instance stored in `a`.

The `cas`-operation can be implemented in a weak and a strong fashion, but the algorithmic implementation provided here basically remains the same.[4]

With the wording provided in the three operations `load`, `store`, and `exchange`, the code of Algorithm 5 speaks for itselve. However, there are some interesting notes to make:

1) The algorithm initially jumps to a label inside the `while`-loop. With minor changes, this jump can be avoided, which would make the algorithm to start (optimistically) with an

[4] *Real* weak implementations have not shown to perform better on the systems available for the evaluation.

Algorithm 4 Atomic Shared Pointer: **exchange**

```
exchange( atomic_sptr<T>& a, const sptr<T>& s ): sptr
    if( s.cptr.pctrl != NIL )
        s.cptr.pctrl->pc.fetch_add( 0, 1 ) // acquire globally
    old_cptr = a.cptr.exchange({ 1, s.cptr.pctrl }) // exchange pointers
    return { old_cptr } // return the old pointer

exchange( atomic_sptr<T>& a, sptr<T>&& s ): sptr<T>
    old_cptr = a.cptr.exchange( s.cptr )
    s.cptr = { 0, NIL } // prevent old pointer from getting released
    return { old_cptr }
```

atomic `cas` of the packed pointer in Line 6. But it turned out, that first retrieving a copy of the atomic instance in form of a temporary instance (Line 14) yields better run-time performance.

2) We have to put a copy of the desired instance s^r into the atomic variable, which is created in Line 28 before executing the `cas` in Line 6. If we would create a copy of the non-atomic instance s^r after we put it into the atomic variable, a (very rare and unlikely) situation could be constructed where the global and usage counter T^r and U^r would be zero although there were still threads holding onto r via temporary instances.[5] Apart from that, a (not so difficult) solution to allow negative *usage counters* would be required as well. I mention that only as an anecdote, since I had it implemented this way and suprisingly just found the bug through trying to prove its correctness (which apparently did not work). I rather expected seeing that bug to occur in the tests or applications I built using this shared pointer implementation.

3) Although unlikely, something close to the ABA-problem can happen: When the `cas` fails in Line 6 due to an unexpected shared object being referenced (i.e., the condition in Line 12 is true), Line 14 copies the atomic instance creating a temporary instance. Now, in the meantime, the atomic variable `a` could have been changed (back) to referencing our expected object q (i.e., Line 15 is false). In this situation, we release the temporary instance referencing q by decrementing the local counter of the non-atomic instance to q stored in the variable `expected`. Here, we can avoid an instruction on an atomic variable due to the ability of the local and global (temporary) reference counters to be negative.

[5]Other situations with a wrong usage counter could be created, too.

2.6 Helper functions

There is only one helper function to explain: `release`. It reduces the global and usage counter of a shared object by a specified amount. When that results in both counters being 0, it destroys the control block and shared object.

2.7 Diverging temporary reference counters

The implementation above has a drawback: The reference counters can diverge, i.e., local and global (temporary) reference counters can increase indefinitely. At some point, this leads to integer overflows breaking the whole algo-

Algorithm 5 Atomic Shared Pointer: **compare_and_exchange (cas)**

```
 1 cas( atomic_sptr<T>& a, sptr<T>& expected, const sptr<T>& desired ): bool
 2     exp_cptr = { 0, expected.cptr.pctrl }
 3     acquired = false
 4     goto start // although not necessary, this turned out to be more efficient
 5     while( true ) {
 6         if( a.cptr.cas( exp_cptr, desired.cptr )) {
 7             if( exp_cptr.pctrl != NIL )
 8                 release( exp_cptr.pctrl, { exp_cptr.tmp_ref, 1 })
 9             return true
10         }
11         // failed due to unexpected pointer or counter...
12         if( expected.cptr.pctrl != exp_cp.pctrl ) {
13 start:
14             old_cptr = a.cptr.fetch_add( 1 ) // enter: increase local ref counter
15             if( expected.cptr.pctrl != old_cptr.pctrl ) { // if still unexpected
16                 if( old_cptr.pctrl != NIL )
17                     old_cptr.pctrl->pc.fetch_add( 1, 1 ) // balance and acq.
18                 expected = { 0, old_cptr.pctrl }
19                 if( acquired and desired.cptr.pctrl )
20                     release( desired.cptr.pctrl, { 0, 1 })
21                 return false
22             }
23             exp_cptr = { old_cptr.tmp_ref+1, old_cptr.pctrl }
24             expected.cptr.tmp_ref -= 1 // compensate ”enter” in line 16

26             if( not acquired ) {
27                 if( desired.cptr.pctrl )
28                     desired.cptr.pctrl->pc.fetch_add( 0, 1 )
29                 acquired = true
30             }
31         }
32     }
```

Algorithm 6 Atomic Shared Pointer: **release**

```
release( ctrl_block<T>* ctrl,
         paired_counter pc )
    old = ctrl->pc.fetch_sub( pc )
    if( old == pc ) { // $pc - old = 0$
        delete ctrl->data
        delete ctrl
    }
```

rithm. This problem can be solved by simultaneously reducing local and global reference counters by the same amount when they are above a certain threshold. For example, in the load function, we can balance the counters after Line 5, which yields Algorithm 7.

The same strategy has been implemented in the `cas`-operation, which also creates temporary instances potentially diverging the temporary counters. Details can be found in the source code.

3 Correctness

Though the core idea of the proposed implementation is similar to the other existing implementations, i.e., using split reference counters, the proposed algorithm differs by allowing *negative local reference counters*. Since it is therefore not trivial to see the correctness of the algorithm, this section proves it.

The goal of this section is the proof of Theorem 1, which claims that an object is destroyed *iff*[6] the last instance releases it.

The following lemmata help formulate the argumentation of Theorem 1. They prove essential invariants of the algorithm.

It is worth mentioning, that we only need to focus on the lines of the pseudo code executing (atomic) operations on variables that are shared by multiple threads, because only there, race conditions can occur due to concurrency.

[6]This word is a common synonym for *if and only if* in mathematics and theoretical computer science.

Lemma 1 *For each shared object p, the global usage counter U^p always equals the amount of atomic and non-atomic instances referencing p.*

Lemma 2 *Let a_i^p and s_j^p be the atomic and non-atomic instances referencing a shared object p. Then,*

$$\sum l^p_{a_i} + \sum l^p_{s_j} - T^p = J^p \tag{1}$$

counts the number of temporary instances referencing p.

Lemma 3 *Let a_i^p and s_j^p be the atomic and non-atomic instances referencing a shared object p. Then,*

$$\underbrace{\sum l^p_{a_i} + \sum l^p_{s_j} - T^p}_{=J^p} + U^p = I^p. \tag{2}$$

counts the total number of instances referencing p.

Proof. Assuming that an invariant holds true before the execution of an atomic instruction, we have to show, that the invariant still holds true right after it as well. Following this approach, we will proof the invariants of Lemma 1, 2, and 3 simultaneously.

load: Line 2 retrieves a reference to the shared object p of the atomic instance a^p, simultaneously increasing the local counter l^p_a. Hence, right after the execution of this instruction, we have a new temporary instance referencing p and Lemma 2 still holds. Equation 2 of Lemma 3 remains intact as well.

Line 5 increments U^p, simultaneously incrementing T^p, i.e., the temporary instance is atomically converted to a non-atomic instance, which is then returned

Algorithm 7 Atomic Shared Pointer: **load and balance**

```
1  load( atomic_sptr<T> &a ): sptr<T>
2     cptr = a.cptr.fetch_add( 1, 0 )
3     if( cptr.pctrl == NIL )
4        return NIL
5     cptr.pctrl->pc.fetch_add( 1, 1 )
6     if( cptr.tmp_ref > THRESHOLD ) {
7        tmp = cptr + { 1, 0 }
8        if( a.cptr.cas( tmp_cptr, { 0, cptr.pctrl })
9           cptr.pctrl->pc.fetch_sub({ tmp_cptr.tmp_ref, 0 })
10    }
11    return { 0, cptr.pctrl }
```

to the caller, i.e., Lemma 1 still holds. Lemma 3 also still holds, since both, U^p and T^p have been incremented, zeroing each other out. Lemma 2 remains intact because the only change was the decrement of an l_a^p on the left hand side of Equation 1, which corresponds to the release of a temporary instance.

store: As this operation can read from the non-atomic instance s^p via the constant parameter `s`, Line 3 increments the usage counter U^p, which is the moment of a new non-atomic instance s_2^p being born, i.e., Lemma 1 and 3 remains intact. Understand, that it makes no difference, whether this new non-atomic instance is stored in a separate variable owned only by this thread or not. More important is the interpretation, that now there is a second non-atomic instance that references the same object p as the read-only variable `s`. Since neither a temporary (local or global) counter has been changed nor was a temporary instance created or released, Lemma 2 remains intact, trivially.

Line 4 converts this new non-atomic instance s_2^p into an atomic instance (by putting it into the atomic variable `a`), simultaneously converting the old instance a^q from the atomic variable into a non-atomic instance s^q. Hence, both invariants of Lemma 2 and 3 still hold for both, p and q. Lemma 1 remains intact, trivially.

Now, the old instance s^q from `a` is not required anymore, so it is released in Line 6, simultaneously reducing the usage counter U^q; thus, all invariants remain intact for q.

In the second implementation, the non-atomic instance s^p from `s` is moved into `a`, converting it into an atomic instance. Simultaneously, the atomic instance a^q from `a` is retrieved and converted into a non-atomic instance s^q. For both objects p and q, the usage counters did not change, as it was an atomic conversion of both instances, simultaneously. Thus, all invariants hold after the execution of the instruction in Line 9.

exchange: Here, the proof of the invariants is similar to the proof of the invariants in the `store`-operation. The only difference is, that we don't release the old instance but rather return it to the caller. Though, in the second implementation, we have to reset the content of the parameter `s`, which is, however, not affected by concurrency.

cas: The interface of the `cas`-operation is more complex, as is its behaviour and implementation. At the beginning of the operation, let a^p be the atomic instance stored in `a`, e^q be the non-atomic instance stored in `expected` and d^r be the non-atomic instance stored in the parameter `desired`.

Although, there is a `goto`-statement in Line 4 jumping to Line 15, we rather proof the invariant for this operation in another order, as it makes the proof more concise.

In this operation, we have to discuss the atomic instructions in Lines 6, 8, 14, 17, 20, and 28.

Following the code flow, Line 6 can only be reached after Line 27 has been executed. There and then in Line 28, a new non-atomic instance s^r is virtually created increasing U^r. Hence, Lemma 3 1 remain intact.

Further on, in Line 6, we have to **distinct two cases**: If 1) the atomic cas-instruction is successful, two important changes have been made. On the one hand, the old atomic instance a^p in `a` is no more. Remember though, that the instruction is only successful, if `a`'s content equals `exp_cptr`, which implies that p and q are the same object. By interpreting `exp_cptr` as a non-atomic instance from here on, we can interpret the cas as a conversion of the atomic instance a^p to a non-atomic instance s^p. Since no temporary counter or temporary instance has been changed, all invariants remain intact.

On the other hand, `a` now holds an atomic instance to r. We interpret this situation, such that the non-atomic instance s^r from Line 28 has been converted into an atomic instance a^r. Again, there are no temporary counter or instance changes. Ergo, all invariants also remain intact for r. Line 8 releases the non-atomic instance s^p that was previously stored in `a` decrementing U^p and reducing T^p by l_s^p. Therefore, Lemma 1 and Equation 2 of Lemma 3 still hold.

If 2) the atomic cas-instruction in Line 6 was not successful, no change has been made so far. The next atomic instruction in Line 14 retrieves the atomic instance $a^{p'}$ (note, that the referenced object might have changed from p to p' since the last atomic operation), simultaneously creating a temporary instance by incrementing the local reference counter $l_a^{p'}$. Lemma 2 still holds, since the number of temporary instances increased by one as well as the left hand side of Equation 1. Equation 2 also still holds, since I^p increased by one due to the new temporary instance and the left hand side increased by one due to the increment of $l_a^{p'}$. Lemma 1 remains intact since there is no change in the amount of atomic or non-atomic instances.

After that, a second case distinction is necessary in Line 15: Either the referenced object is still not the `expected` object q, then Line 17 converts the temporary instance into a non-atomic instance and Line 18 hands that non-atomic reference to the caller. Analogously to previous conversions of temporary instances into non-atomic instances, all invariants remain intact. Before returning to the caller, the previously created non-atomic instance s^r (if it was created) will be released in Line 20. It is easy to see, that all invariants remain intact.

The second case in Line 15 is that the atomic shared pointer changed in the meantime referencing the `expected` object, i.e. p' and q are the same now. In this case, Line 24 decrements the local counter l_e^q, which compensates incrementing the local counter $l_a^{p'} = l_a^q$ earlier, eventually destroying the temporary instance; thus, both sides of Equation 2 of Lemma 3 decremented by one. Also, Lemma 2 remains intact due to releasing the temporary object being reflected in decrementing the right hand side of Equation 1. Lemma 1 also trivially remains intact. In both cases and branches, all invariants for all relevant objects still hold.

Considering the analysis above as the step of a proof by induction, the following is the start of it:

In every application, each object p is first constructed and referenced by a non-atomic shared pointer instance s^p with $U^p = 1$, $T^p = 0$ and $l_s^p = 0$. So, all invariants of Lemma 1, 2, and 3 hold for freshly created and referenced objects. From there on, the case distinction above shows, that each atomic operation does not change either of the invariants, which concludes the proof of Lemma 1, 2, and 3. □

Theorem 1 *A shared object p is destroyed immediately after there is no (atomic, non-atomic, or temporary) instance left, referencing it.*

Proof. Each shared object p is destroyed via the `release`-function (Algorithm 6). There, an object p is destroyed if and only if $T^p = 0$ and $U^p = 0$. Now, we have to show that

$$U^p = 0 \wedge T^p = 0 \Longleftrightarrow I^p = 0$$

holds, i.e., this condition is true if and only if there are no instances referencing p.

"$\Rightarrow$": Since $U^p = 0$, there are neither atomic nor non-atomic instances referencing p (Lemma 1). This implies, that there are no local counters l_a^p or l_s^p left, such that Equation 1 reduces to $J^p = -T^p$. Since $T^p = 0$, there are no temporary instances left, either.

"$\Leftarrow$": Since $I^p = 0$, there are no temporary, atomic, or non-atomic instances referencing p. In particular, there are no local counters l_a^p or l_s^p. Therefore, $U^p = 0$ and $J^p = -T^p = 0$ due to Lemma 1 and 2, respectively.

Now, what remains to be seen is, that the last instance referencing p is always a non-atomic instance. If that holds, the correct time of p's destruction clearly must be in `release`, because non-atomic instances are always released via this function.

There are only two code-locations, where temporary instances are held. The first location is in Algorithm 2. Here, however, their is no branch and the temporary instance (created in Line 2) will always be converted into a non-atomic instance (in Line 5). In the meantime, creating the temporary instance prevents `release` from destroying the shared object as it caused $I^p \geq 1$.

The second location is in Algorithm 5 in Line 14. Here, however, the temporary instance either references p also being referenced by the variable `expected`, or it will be converted into a non-atomic instance in Line 17. In the first case, the temporary instance will be destroyed in Line 24. However, this temporary instance cannot be the last instance referencing p because the variable `expected` is exclusively owned by the executing thread and also references p. In the second cas, the object will not be destroyed in the meantime for the same reason as in Algorithm 2. □

4 Evaluation

This section compares Anthony William's [16], Facebook's (folly's) [8], and the standard's (provided by the C++ STL) atomic shared pointer implementations regarding the throughput of their atomic operations.

All experiments have been conducted on three different CPU's: AMD Ryzen 7 3700X (16 hardware threads); AMD Ryzen 9 6900HS (16 hardware threads); and AMD Ryzen 9 5950X (32 hardware threads). In our evaluation, we study different situations regarding the number of (software) threads and the contention of these threads. Since the CAS operation often occurs in a loop, we measured the throughput of single CAS calls as well as CAS loops which count as finished until an expected value could be replace by a new desired value.

Since the term of contention comes with no mathematical metric to measure it, we can only apply heuristics to increase or decrease the contention of the concurring threads. To do that, we measure the throughput of an operation by running n threads in a loop performing this particular operation round robin on a finite pool of m atomic variables. Here, decreasing the pool size m increases the probability of two threads to conflict, i.e, the contention increases. Letting all threads perform on exactly one atomic variable corresponds to maximum contention. Although, two concurring threads might miss each other (e.g. because one operates on the atomic shared pointer while the other thread just left it), their memory access still causes the cache to bounce. The other extreme, no contention, would correspond to an infinite number of atomic variables. Here, the experiment provides individual atomic variables to each thread, such that they work on completely distinguished cache lines.

In most of the experiments, each thread is associated with an additional individual *local* shared pointer. To avoid side-effects due to the allocation routine, all atomic and non-atomic shared pointers are initialized beforehand with their own pre-allocated memory locations. Each of these shared pointers resides in their own cache-line to avoid unnecessary cache-bouncing. Details about how the pre-allocated memory is used follows in the operation's subsections.

All experiments have the same initialization and measurement strategy: First, all threads synchronize using a barrier. Then, the main thread (which is managing the experiment) sleeps for a short warm-up time ($100ms$) while the worker threads start doing their job (which is calling the operation in question in a loop). After the warm-up time, the main thread collects the progress (in number of finished calls) from worker-individual atomic counters and then sleeps again for a longer measurement time ($2s$). When the main thread wakes up, it reads the progress of each worker thread again and uses the previous values to gain the progress during the measurement phase.

Since folly's implementation and the proposed implementation are superior in all of the following experiments, the analysis mainly addresses the comparison between these two.

4.1 Operation: load

To measure the throughput of the `load`-operation, each thread simply loads all atomic shared pointers round robin, immediately discarding the result.

Figure 1 shows that our implementation and folly's lead the race, being on par for low number of threads. However, when the number of threads grows (and therefore the contention too), the speedup of our implementation over folly's increases drastically. Consult the Appendix A to see similar behaviour on other scenarios for the `load`-operation.

Although high-contention scenarios should be avoided in the first place, sometimes there is no way around, in particular for the load-operation. If that is the case, the least is to hold the throughput on a constant level. As the high-contention diagrams show, our pro-

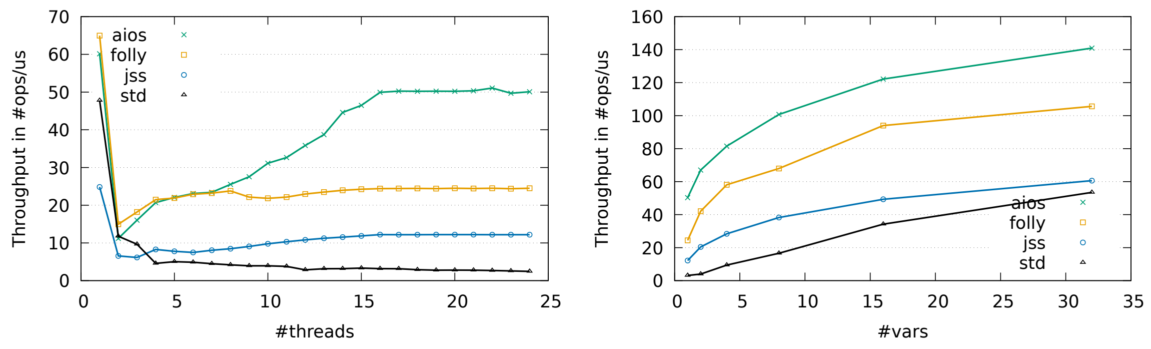


Figure 1: Throughput of the load operation on an AMD Ryzen 7 3700X 16-core machine under high contention (left), and growing contention for maximum processor subscription (right).

posed implementation holds up to this demand. On the other hand, even folly's implementation stagnates on an increasing number of threads, allegedly because of using a CAS loop in their implementation of `load`. Those fail more often under high contention, thus the expected number of turns increases until the CAS loop succeeds. The proposed implementation, however, outperforms this approach because it only contains two determined atomic calls (increasing the local and then the global and usage counter).

4.2 Operation: store

To measure the throughput of the store operation, each thread simply stores their own shared pointer round robin in the pool's atomic shared pointers.

In most scenarios, the proposed implementation is on par with folly's implementation. Though, our implementation is slightly faster in the max-subscription use-case on the AMD 9 6900HS, but it is slightly slower in the same use-case on the AMD 9 5950X (cf. Figure 10 and 11 for approximate numbers). Figure 2, however, shows a general disadvantage of the standard implementation, which uses `mutexes` for synchronization: Its performance drops drastically in over-subscription scenarios. A full overview can be found in Appendix A.

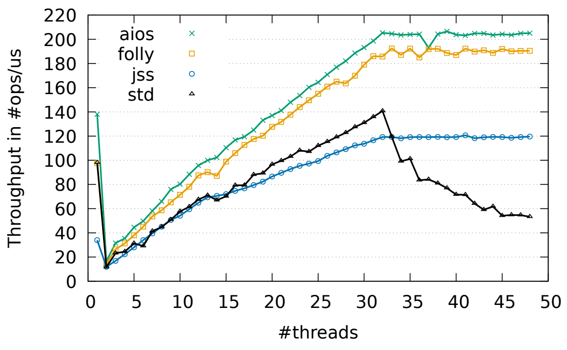


Figure 2: Throughput of the store operation on an AMD Ryzen 9 5950X 32-core machine under low contention.

4.3 Operation: exchange

To measure the throughput of the exchange operation, each thread loops over the pool of atomic shared pointers round robin swapping their current own shared pointer with the one in the pool's atomic shared pointer.

Since our implementation of the `exchange`-operation is quite similar to the `store`'s implementation, the throughput is quite similar as well. In almost all situations, the throughput

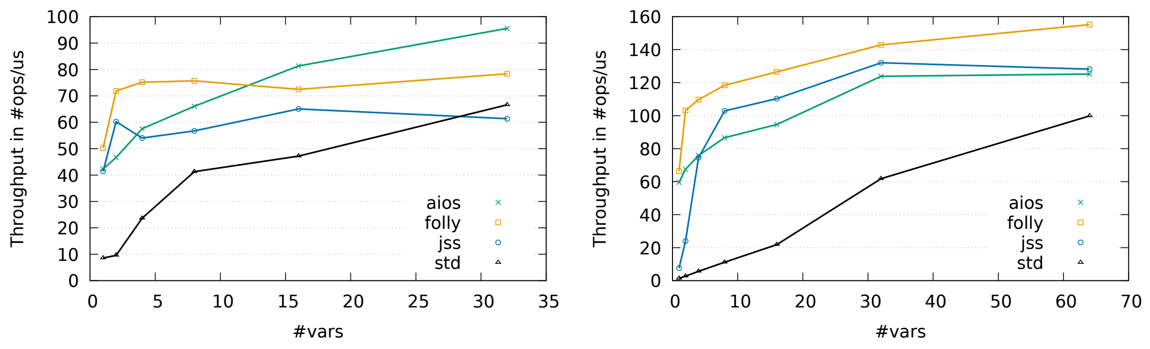


Figure 3: Throughput of the exchange operation on a Ryzen 7 3700X (left) and a Ryzen 9 5950X (right) machine for maximum processor subscription.

of our implementation is on par with folly's. Sill, there are situations where the qualitative behaviour seems to differ (cf. Figure 3) depending on the executing machine.

4.4 Operation: weak and strong CAS

Measuring the throughput of a single CAS call is done similar to measuring the throughput of the store operation: Each thread tries to put its own shared pointer into one of the pool's atomic shared pointers. We set the `expected` argument of the `cas`-operation to be empty (NIL). Since none of the atomic variables is empty, this always leads to a fail of the CAS operation, i.e., we measure the branch that provides us with the current shared pointer in the atomic variable.

Figure 4 shows exemplary that the proposed implementation outperforms the other implementations. In comparison to folly's shared pointers, the reason might be the same as in the `load`-operation: Instead of using a CAS loop to retrieve (load) the current value in the atomic variable, Algorithm 5 (`cas`) only uses a determined and finite set of atomic operations to succeed.

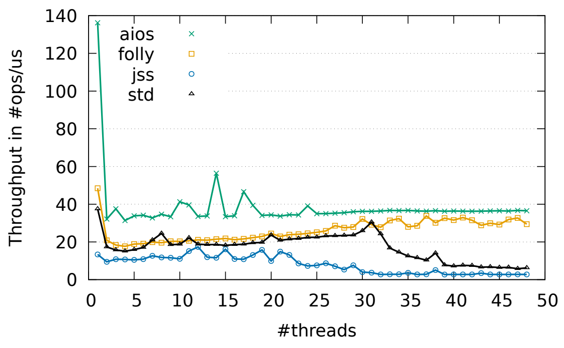
4.5 Operation: weak and strong CAS-loops

Measuring the throughput of a CAS loop is done similar to measuring a single CAS call, except that a CAS loop has to run until the CAS was successful. Clearly, we use the retrieved value from a previous turn in the CAS loop for the `expected` argument to the next CAS call.

It turns out that the throughput of the proposed implementation is close to folly's throughput. Still, the weak CAS loops are approximately 10% slower, though the strong CAS loops are pretty much on par. Figure 5 shows exemplary results. Full results are, again, available in Appendix A.

4.6 Discussion

It's no surprise that the standard implementation suffers under over-subscription, since it uses `mutexes` to synchronize concurrent threads. On the other hand, Anthony William's implementation is not lock-free either because his implementation operates atomically on 128 Bit, which is not supported in a lock-free fashion by most C++ compilers. Therefore, his implementation uses `mutexes` internally as well. Still, it does not suffer under



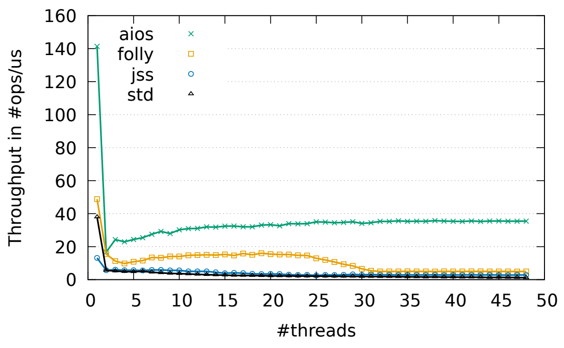


Figure 4: Throughput of the (strong) CAS operation on an AMD Ryzen 9 5950X machine under low contention (left) and high contention (right).

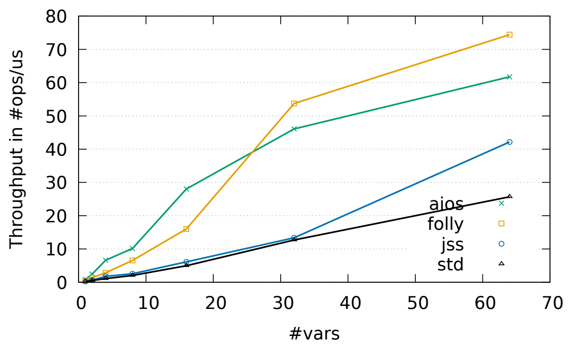


Figure 5: Throughput of the **weak CAS loop** on an AMD Ryzen 9 5950X with growing contention for maximum processor subscription.

over-subscription. I suspect that the reason for still performing well in over-subscription use-cases is the fine-granularity of the `mutex`-based locks. On the other hand, the lock-inversion problem would still occur, but has not been tested here.

Due to the arrangement of the packed pointer in the proposed implementation, adding or subtracting a value from its counter really just is an atomic `fetch_add` or `fetch_sub`. In folly's implementation, however, a CAS loop is required to avoid an integer overflow when manipulating the packed pointer's counter. This CAS loop is prone to fail under heavy contention; and hence, the performance drops for an operation that should yield at least constant throughput under increasing pressure of threads.

An interesting observation also is, that the proposed implementation outperforms the other implementations usually by more than factor 2 when only one thread is running. The simplicity of the code might be suspect for the reason.

Comparing all diagrams in Appendix A, the unstable behaviour of the Ryzen 9 6900HS is noticeable. The machine got pretty hot during the experiments; thus, I suspect heat management due to CPU frequency scaling to be the cause.

5 Conclusion and Outlook

This work provides a proposal for the implementation of atomic shared pointers, a crucial building brick for lock-free data structures. The implementation extents the split-reference counting approach by allowing the local (temporary) reference counters to be negative, which allows to reduce the number of atomic instructions. Section 3 contributes a proof for the correctness of the proposed algorithm, which is especially important for lock-free data structures, since they are extra hard to debug. The evaluation in Section 4 shows that the proposed implementation outperforms other implementations up to an order of magnitude in some high-contention scenarios and up to factor 2 in many other cases (e.g. single `cas` or `load` operations). On the other hand, Facebook's implementation yields a 10% run time advantage in some use cases (in particular weak `cas`-loops).

The technique of the proof of Lemma 3 gives a hint for a debugging framework of atomic data structures: Wrapping an atomic data structure such that each atomic operation synchronizes with a central instance right before and after the atomic operation is executed allows this central instance to force different scenarios, e.g., by decelerating certain threads or prioritizing threads depending on their current execution state. This way, even very seldom patterns could be reproduced and tested deterministically.

A lot of the performance gets lost due to the necessity of cas loops where a `masked_cas` as CPU instruction could avoid the problem: A `shared_ptr` implementation requires a CAS on a data type (e.g. 64 or 128 Bit) that combines a (split) reference counter and an actual value (here a pointer). However, the *compare*-part of the CAS shall only compare the value, i.e., the pointer of the atomic value but ignore the counter in the comparison. Now, if the value matches the expected value, the whole atomic variable shall be changed. Either way, the old value shall be returned, so that the caller of the `masked_cas` also gets to know the part that was masked out from the comparison. Having such a CPU instruction could yield an increased performance in both, the `counted_pointer` as well as the `paired_counter`. Using a bitmask for the comparison could be beneficial in other algorithms as well. Here, an example might be the multi-producer-multi-consumer queue that Tony Van Eerd is working on [7].

References

[1] ISO C++ committee, 2011. `https://isocpp.org/`.

[2] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

[3] Daniel Anderson. Lock-free Atomic Shared Pointers Without a Split Reference Count? CppCon 2023. `https://isocpp.org/blog/2023/09`.

[4] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs, 2010. `http://www.stroustrup.com/isorc2010.pdf`.

[5] Timur Doumler, 04 2022. `https://accu.digital-medium.co.uk/wp-content/uploads/2022/04/talk.pdf`.

[6] Timur Doumler. A lock-free atomic shared_ptr. ACCU, 04 2022. `https://accu.org/video/spring-2022-day-1/`.

[7] Tony Van Eerd. Lock-free by Example. CppCon, 2014. `https://github.com/CppCon/CppCon2014`.

[8] Facebook. Facebook Open-Source Library "Folly". `https://github.com/facebook/folly`, 2023.

[9] Intel. Performance Monitoring Impact of Intel® Transactional Synchronization Extension Memory Ordering Issue, 2023. `https://cdrdv2.intel.com/v1/dl/getContent/604224`.

[10] Scott Meyers and Andrei Alexandrescu. C++ and the Perils of Double-Checked Locking. *Doctor Dobbs Journal*, 29, 07 2004.

[11] Herb Sutter. Effective Concurrency: Lock-Free Code — A False Sense of Security. `https://herbsutter.com/2008/08/05/`.

[12] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[13] Herb Sutter. Atomic smart pointers, rev. 1. Standard C++ Foundation, 2014. `https://isocpp.org/files/papers/N4162.pdf`.

[14] Herb Sutter. Lock-Free Programming (or, Juggling Razor Blades). CppCon, 2014. `https://github.com/CppCon/CppCon2014`.

[15] Anthony Williams. *C++ Concurrency in Action: Practical Multithreading.* Manning Pubs Co Series. Manning, 2012.

[16] Anthony Williams. Lock-free atomic shared pointers. `https://github.com/anthonywilliams/atomic_shared_ptr`, 2021.

A Run-Time Experiments

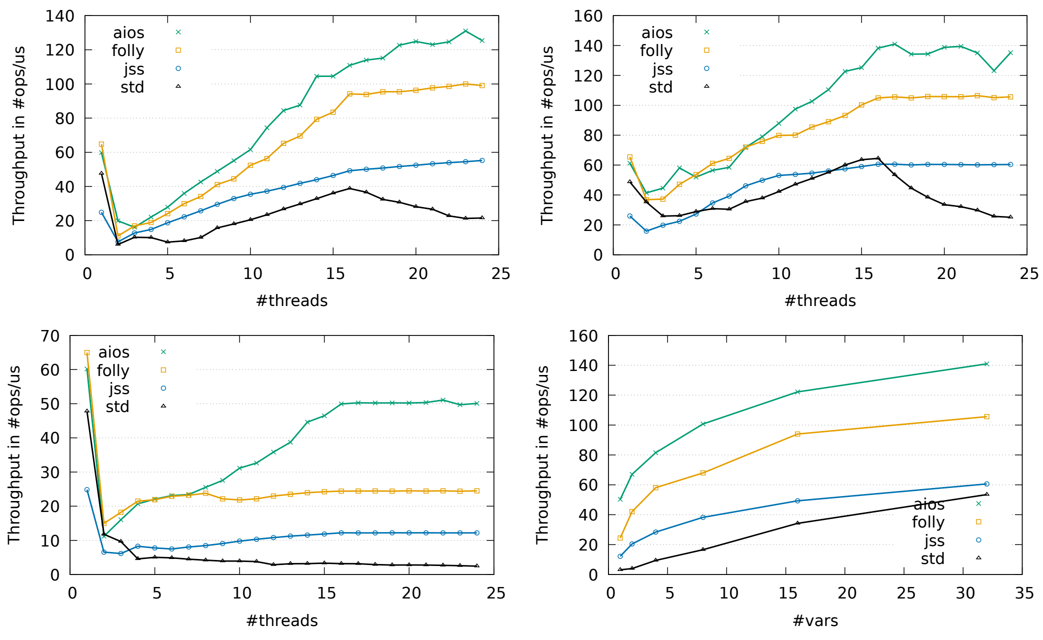


Figure 6: Throughput of the **load** operation on a **16-core** machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

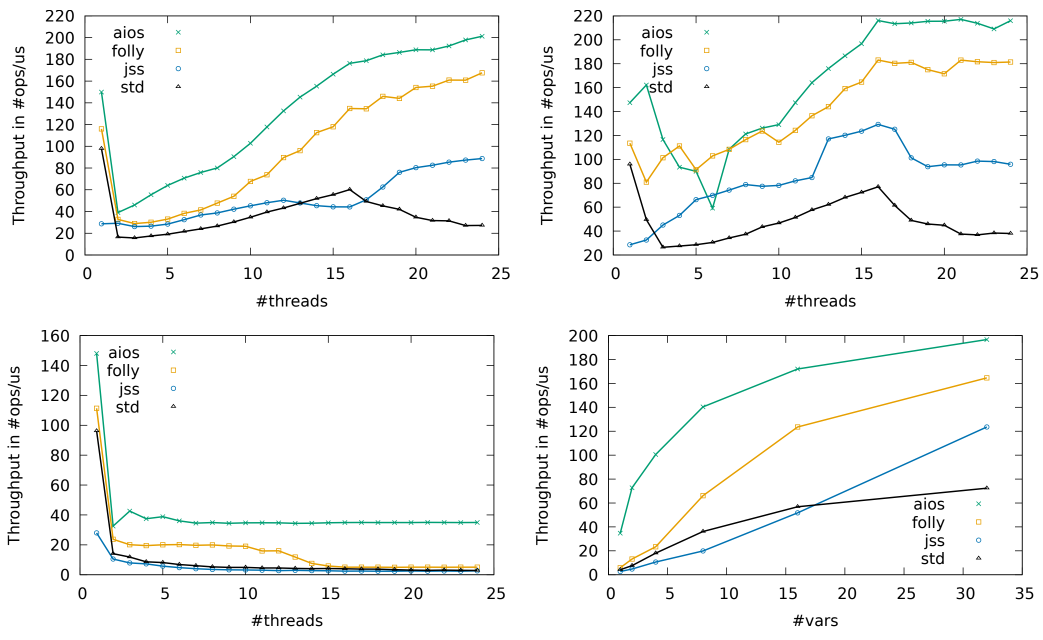


Figure 7: Throughput of the **load** operation on a **16-core** (AMD6000) machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

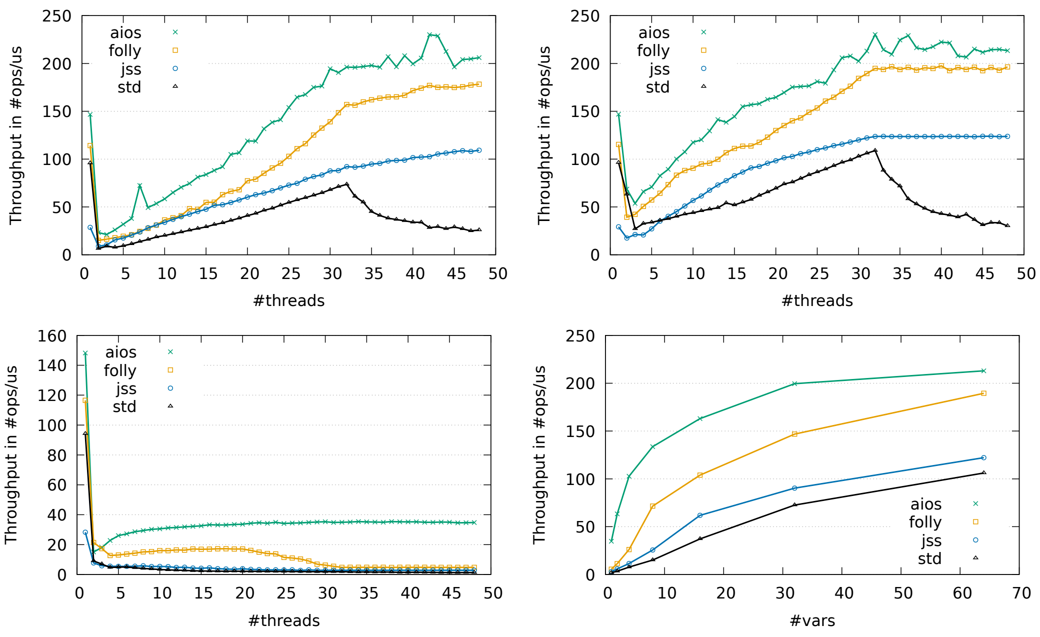


Figure 8: Throughput of the **load** operation on a **32-core** machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

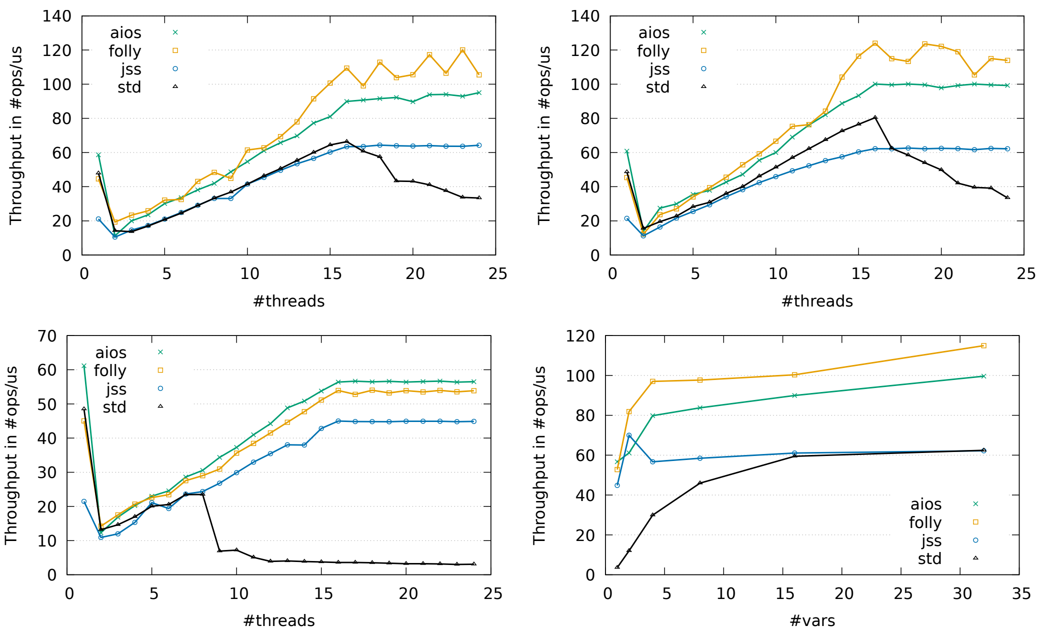


Figure 9: Throughput of the **store** operation on a **16-core** machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

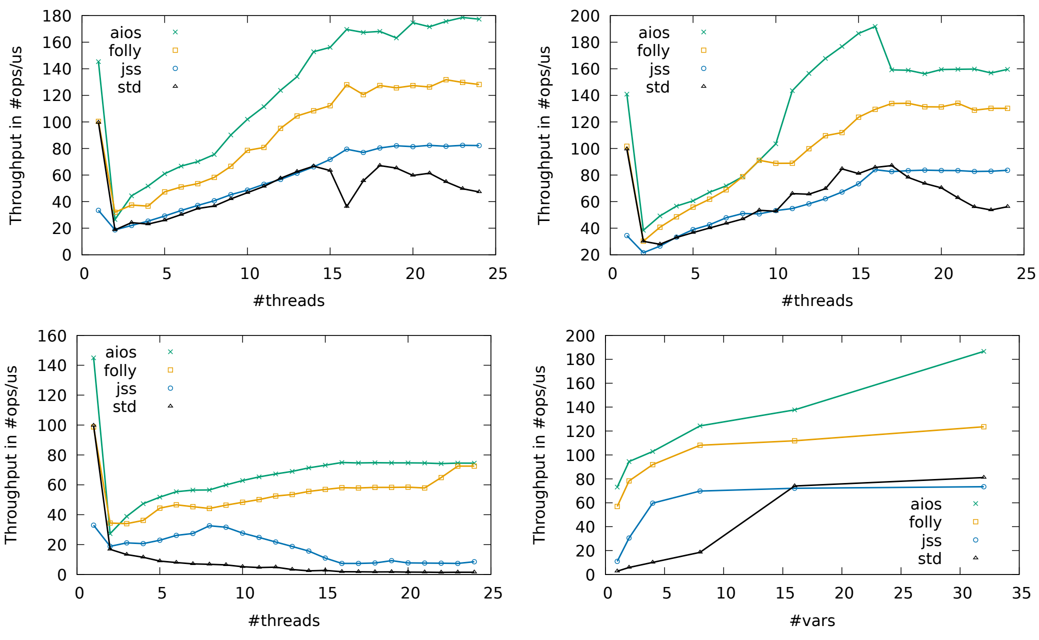


Figure 10: Throughput of the **store** operation on a **16-core** (AMD6000) machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

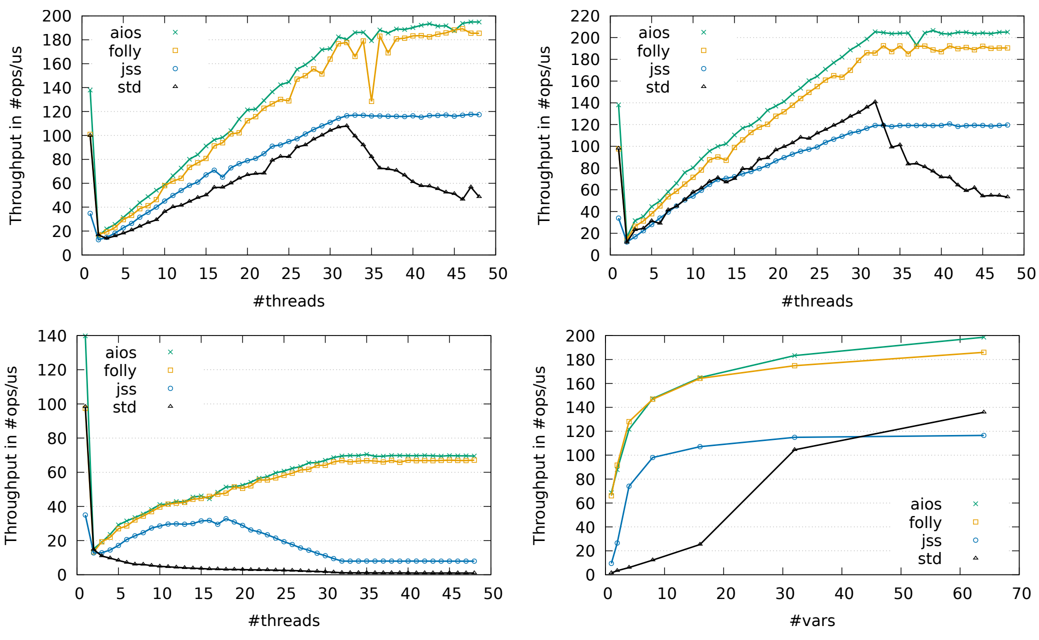


Figure 11: Throughput of the **store** operation on a **32-core** machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

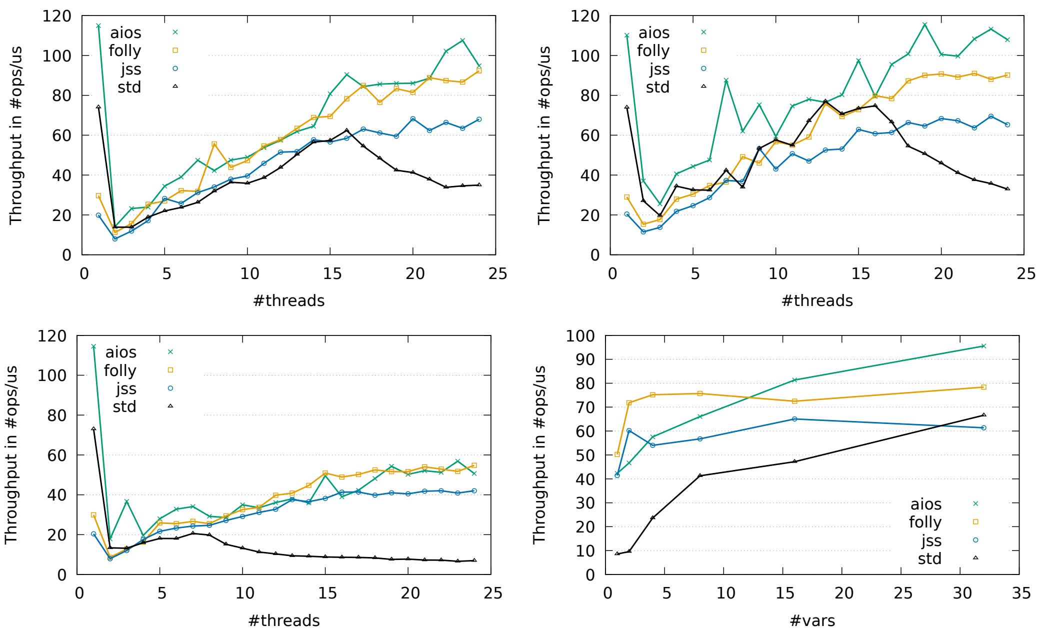


Figure 12: Throughput of the **exchange** operation on a **16-core** machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

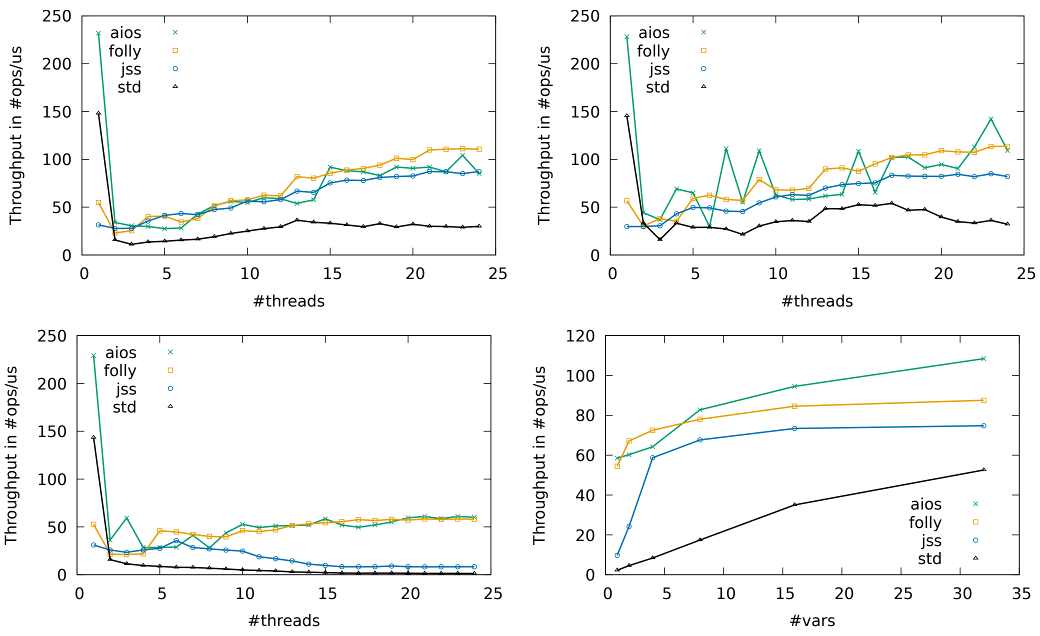


Figure 13: Throughput of the **exchange** operation on a **16-core** (AMD6000) machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

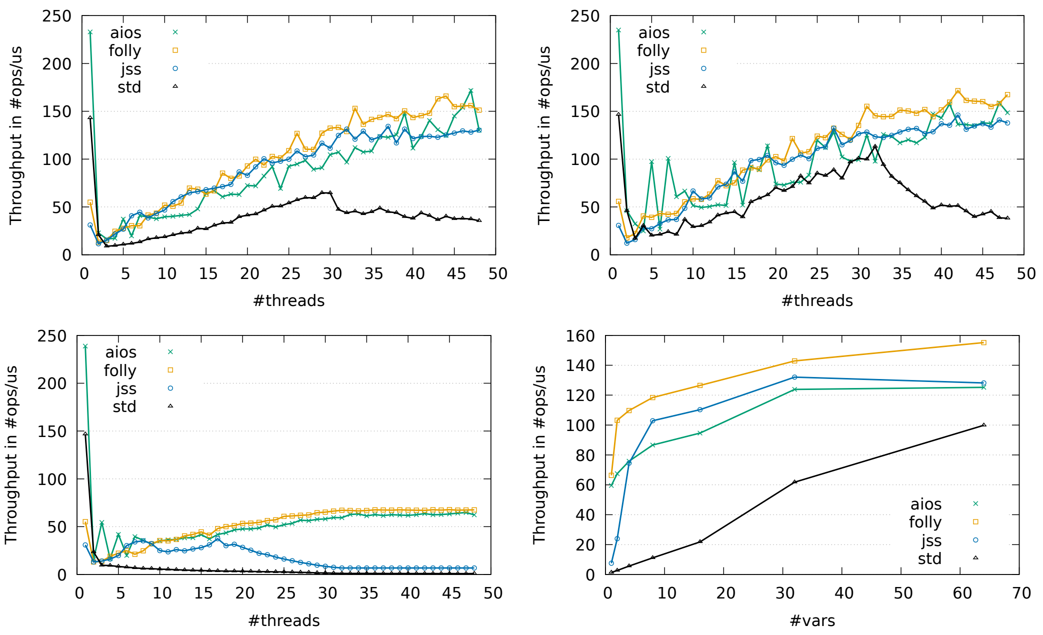


Figure 14: Throughput of the **exchange** operation on a **32-core** machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

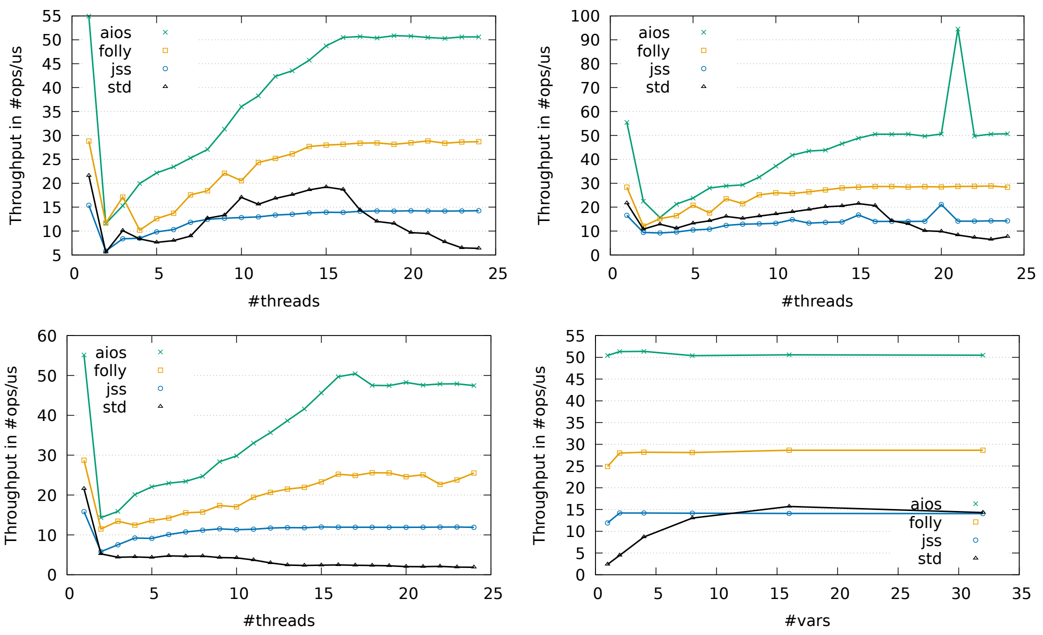


Figure 15: Throughput of the **weak CAS** operation on a **16-core** machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

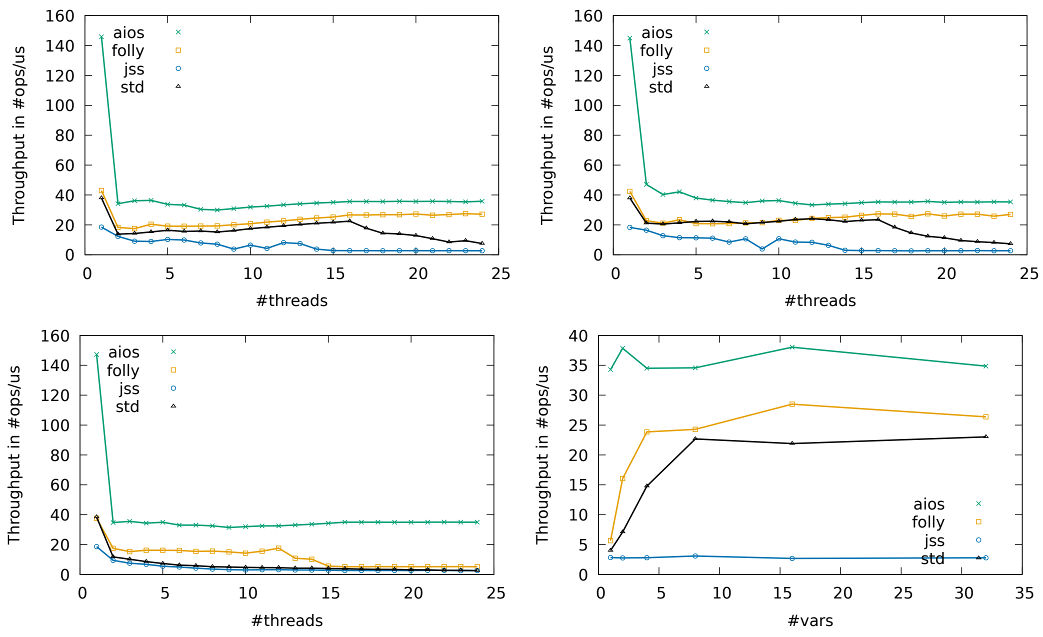


Figure 16: Throughput of the **weak CAS** operation on a **16-core** (AMD6000) machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

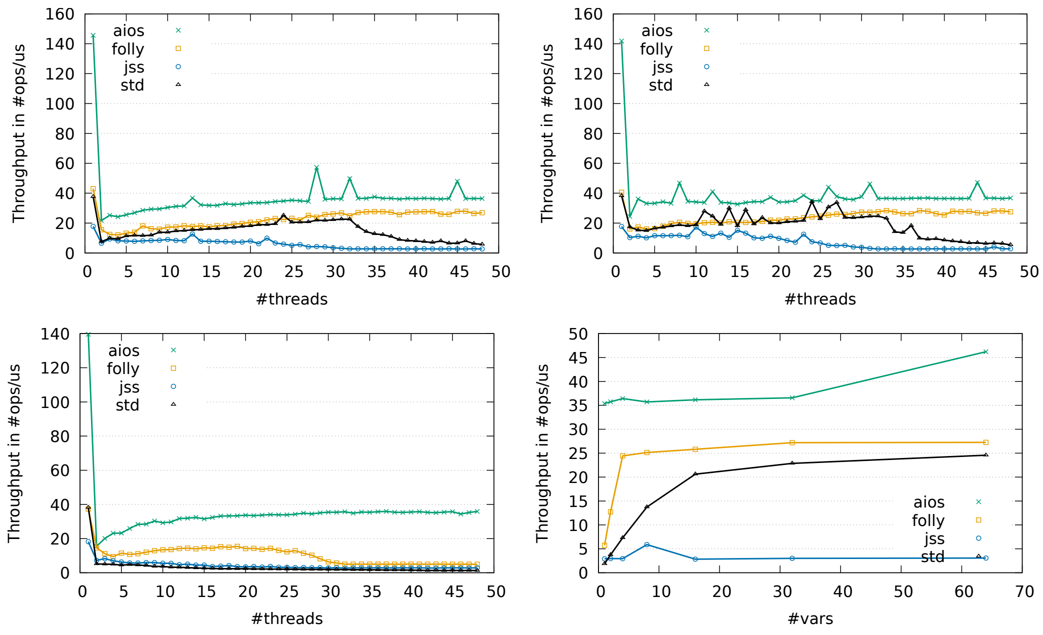


Figure 17: Throughput of the **weak CAS** operation on a **32-core** machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

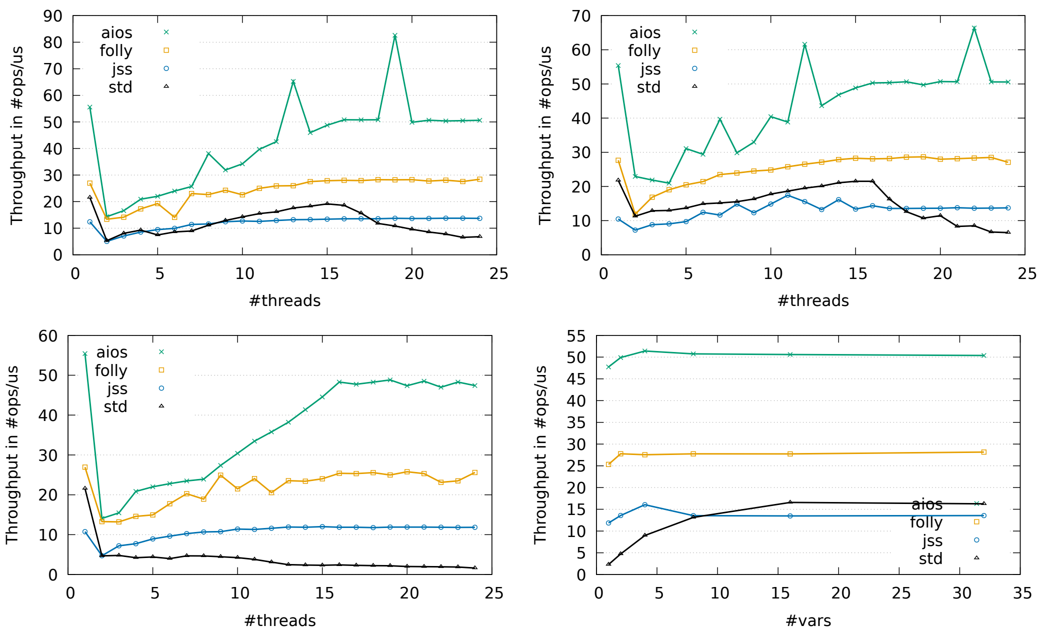


Figure 18: Throughput of the **strong CAS** operation on a **16-core** machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

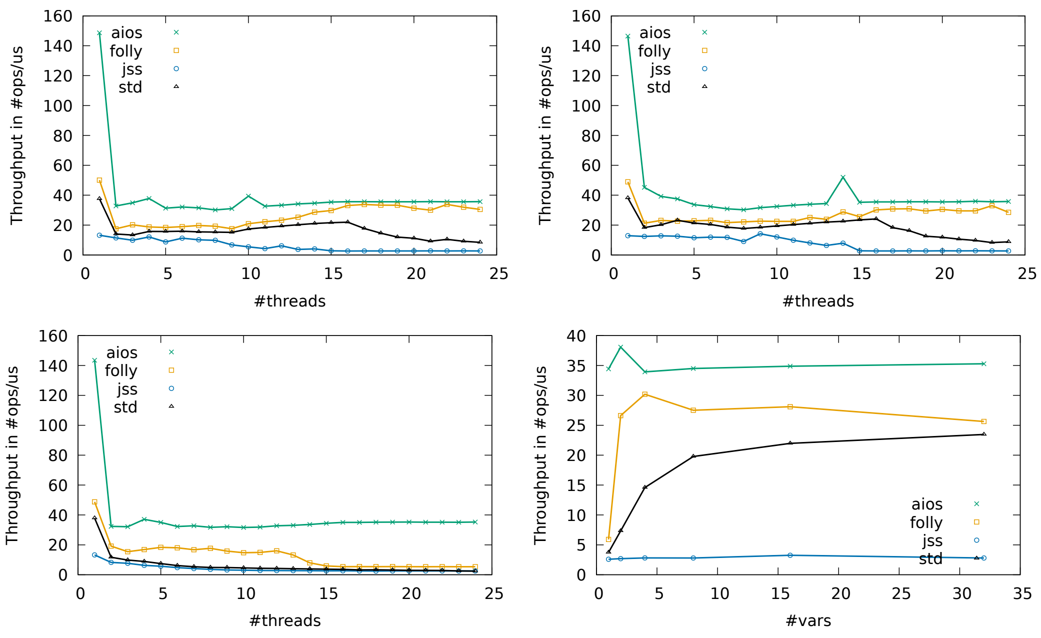


Figure 19: Throughput of the **strong CAS** operation on a **16-core** (AMD6000) machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

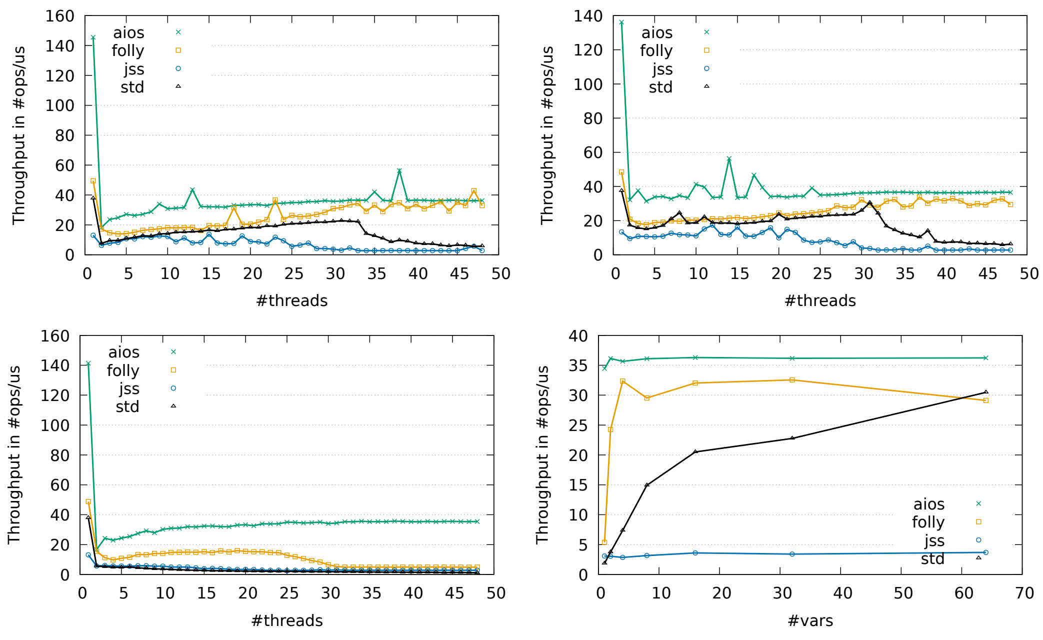


Figure 20: Throughput of the **strong CAS** operation on a **32-core** machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

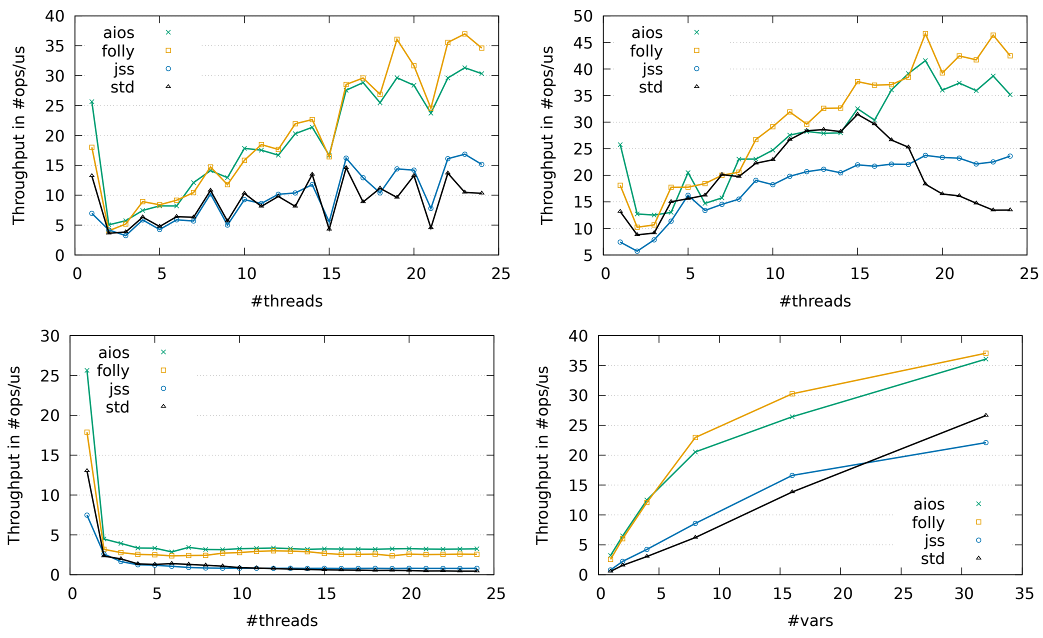


Figure 21: Throughput of the **looped weak CAS** operation on a **16-core** machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

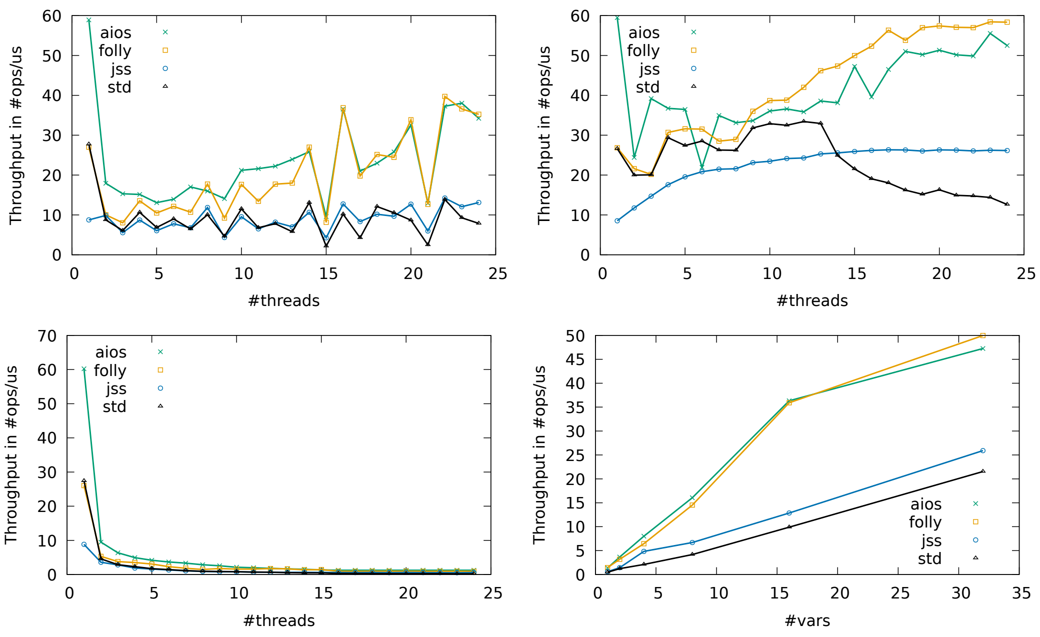


Figure 22: Throughput of the **looped weak CAS** operation on a **16-core** (AMD6000) machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

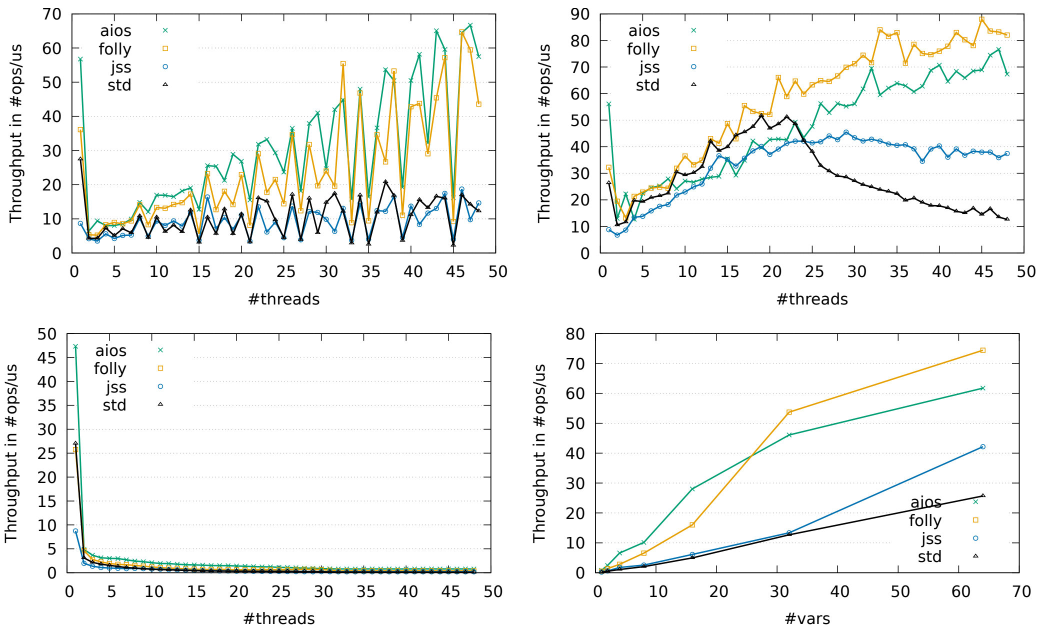


Figure 23: Throughput of the **looped weak CAS** operation on a **32-core** machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

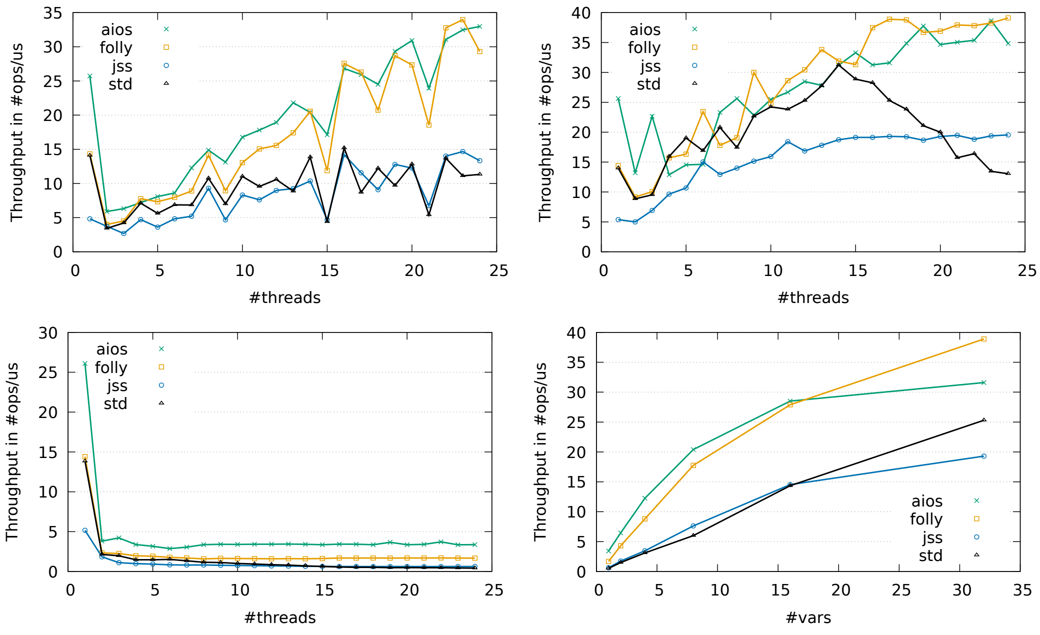


Figure 24: Throughput of the **looped strong CAS** operation on a **16-core** machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

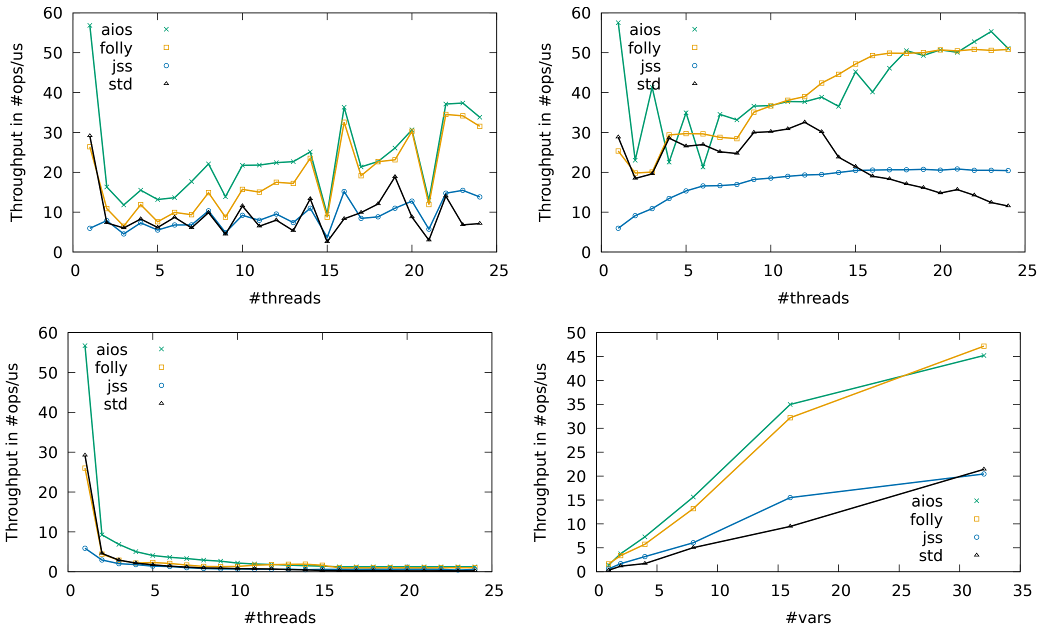


Figure 25: Throughput of the **looped strong CAS** operation on a **16-core** (AMD6000) machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).

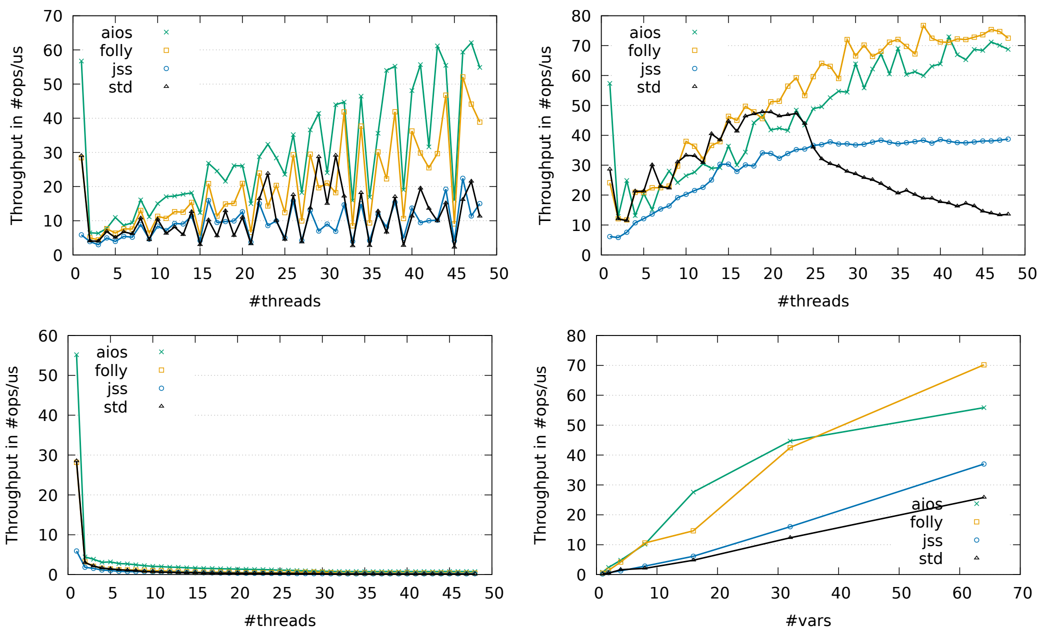


Figure 26: Throughput of the **looped strong CAS** operation on a **32-core** machine under no contention (top left), low contention (top right), high contention (bottom left), and growing contention for maximum processor subscription (bottom right).