Transactional Pointers: Experiences with HTM-Based Reference Counting in C++

Maria Carpen-Amarie[1], Dave Dice[2], Gaël Thomas[3], and Pascal Felber[1]

[1] Université de Neuchâtel, Switzerland
firstname.lastname@unine.ch
[2] Oracle Labs, USA
dave.dice@oracle.com
[3] Telecom SudParis, France
gael.thomas@telecom-sudparis.eu

Abstract. The most popular programming languages, such as C++ or Java, have libraries and data structures designed to automatically address concurrency hazards in order to run on multiple threads. In particular, this trend has also been adopted in the memory management domain. However, automatic concurrent memory management also comes at a price, leading sometimes to noticeable overhead. In this paper, we experiment with C++ smart pointers and their automatic memory-management technique based on reference counting. More precisely, we study how we can use hardware transactional memory (HTM) to avoid costly and sometimes unnecessary atomic operations. Our results suggest that replacing the systematic counting strategy with HTM could improve application performance in certain scenarios, such as concurrent linked-list traversal.

1 Introduction

With the increasing degree of concurrency in nowadays hardware, lock-free implementation of applications or data structures gained extensive attention in the last few years. In this context, using classical synchronization mechanisms based on locks (such as mutexes, barriers, etc.) tends to become more and more complex and error-prone. *Transactional memory* (TM) [8] offers an elegant solution for implementing lock-free synchronization. Until recently, TM algorithms were mostly reserved to the research environment, since the considerable overhead generated by software transactional memory (STM) implementations made them unsuitable for real-life applications. However, the emergence of hardware transactional memory (HTM) in mainstream processors overcame the performance pitfall, while conserving the benefits in scalability and correctness.

Automatic memory management mechanisms often suffer from performance drops due to their synchronization strategies. A notable example is represented by the smart pointer implementation in the C++ standard library. This uses *reference counting* to protect a raw pointer from being illegally deallocated and to avoid any other memory hazards. Smart pointers are thread-safe and the

operations on the shared reference counter are atomic. This provides adequate and safe memory management for multi-threaded programs. Nonetheless, the reference counting strategy is costly and sometimes unnecessary, e.g., when manipulating copies of a smart pointer with a reference count that never drops below 1 and hence never needs to release memory.

In this paper, we explore possible scenarios where HTM could improve the performance of applications that use C++ smart pointers. Specifically, we replace the original reference counting logic based on atomic operations with hardware transactions. The hardware transaction protects the raw pointer against invalid accesses. In this manner, we avoid executing the unnecessary atomic operations required by the reference counting strategy. On the one hand, we expect HTM to improve the performance of smart pointers over the original implementation. On the other hand, by adding this low abort-rate HTM fast-path, we are also addressing some concurrency problems related to smart pointer handling. Gottschlich et al. [7] show that template-based generic structures, such as C++ smart pointers, are deadlock-prone, among other synchronization issues, and they also propose the use of TM in their implementation.

Our contribution consists of an extensive study on the benefits of HTM for C++ smart pointers. We added transactional support for smart pointers, and tested their performance on: (1) *micro-benchmarks*, with mono- and multi-threaded settings, with and without batching multiple pointers in a single transaction, on two different architectures (Intel Haswell and IBM POWER8); and a (2) *concurrent data structure*, with and without batching. The results are generally encouraging and we observe performance improvements in most, but not all scenarios: in some cases there are no or negligible gains (e.g., multi-threaded micro-benchmark with batching enabled), whereas in others the execution time is improved by 50% (e.g., concurrent lookup operation over a linked-list, with batching enabled).

2 Background and Related Work

Automatic memory management is split into two representative approaches: *reference counting* and *tracing*. Tracing algorithms are most often used in high performance settings, while the reference counting strategy is usually avoided due to its major drawbacks. A critical downside is represented by its considerable overhead over tracing, estimated at 30% in average [11]. The concept of reference counting is simple: it keeps track of the number of references for each object, updating a counter when references are removed or new ones are added. The object is destroyed only when the count reaches zero. However, this means that each pointer mutation has to be tracked and intercepted, making a naive implementation of reference counting very expensive. Recently, reference counting techniques were reconsidered and optimized, becoming comparable to tracing in terms of performance [11, 1, 12]. A noteworthy memory management mechanism that depends on reference counting is illustrated by C++ smart pointers.

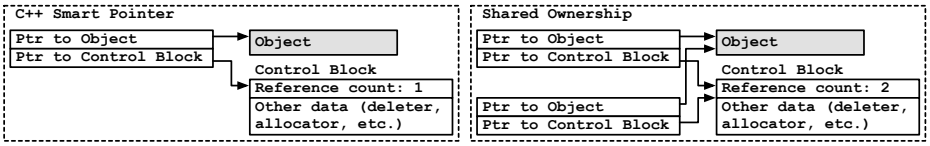


Fig. 1. C++ smart pointer components and reference counting mechanism.

2.1 C++ Smart Pointers

A smart pointer is an abstract data type that encapsulates a pointer while providing additional features, such as automatic memory management or bounds checking. These features were added to classical pointers in order to reduce programming bugs (created by manually managing the memory), while keeping the same efficiency. Smart pointers can prevent most memory leaks and dangling pointers.

In C++, smart pointers are implemented on top of traditional (raw) pointers, but provide additional memory management algorithms. We focus on the `std::shared_ptr` implementation located in the `<memory>` header. A `shared_ptr` represents a container for a raw pointer, for which it maintains reference counted ownership (Fig. 1). The object referenced by this pointer will be destroyed when there are no more copies of the `shared_ptr`.

The smart pointers are implemented as a C++ template that uses a reference counting strategy for memory management. The increments and decrements of the counts are synchronized and thread-safe. The default synchronization mechanism for C++ smart pointers employs atomic operations (increment/decrement).

2.2 Hardware Transactional Memory

Transactional memory (TM) [8] is a synchronization mechanism that can provide lock-freedom by encapsulating blocks of instructions in transactions and executing them atomically. In order to keep track of the changes, TM typically records the write set and applies it to memory atomically if the transaction succeeds; otherwise updates are discarded. There is no moment in time when an intermediate state can be observed. The most common cause for aborting a transaction is upon memory conflict, that is when two threads try to access the same memory areas. An abort can also be deliberately triggered by the application.

Transactional memory was first implemented in software (STM). Even though the benefits of TM over classical synchronization methods are significant, notably in terms of ease of use, STM was the subject of long debates whether it is only a research toy [3, 6]. The most important issues are the significant overhead due to the instrumentation of the application, and the limitation to "weak atomicity" (i.e., it identifies conflicts only between two transactional accesses). Starting in 2013, Intel made available for public use its "Haswell" processor with fully integrated hardware transactional memory (HTM) support. HTM overcomes the aforementioned problems of STM. However, it has its own disadvantages: first,

the size of the transactions is limited. Careful planning for the contents of the transaction is needed in order to both avoid overflows and amortize the cost of starting and committing the transaction. Moreover, transactions can be aborted at any time by interrupts, faults and other specific instructions, such as debug or I/O. HTM requires a non-transactional fallback path in case of abort, to ensure progress for the transactions that cannot commit. HTM is therefore a very powerful tool in a multi-core environment, although not suitable for all types of applications because of the aforementioned limitations. Nonetheless, it appears to be a suitable solution for tackling specialized concurrency problems, such as concurrent memory management.

2.3 Related Work

Considering the ever increasing interest in transactional memory in the last few years, a reasonable amount of effort has been focused on integrating TM with mainstream programming languages, such as C++. Crowl et al. [4] present a general design that would permit the insertion of transactional constructs into C++. They identify the main issues that need to be addressed and propose a new syntax that could be incrementally adopted in the existing code base. Ni et al. [10] go even further and implement a fully working STM system that adds language constructs for transactional programming in C++. The system includes new C++ language extensions, a compiler and an STM runtime library. They conduct an extensive evaluation on 20 parallel benchmarks ported to use their C++ language extensions. The results show that the STM system performs well on all workloads, especially in terms of scalability. A more focused work is presented by Gottschlich and Boehm [7] regarding the need for transactional memory in generic programming. They give as example C++ shared pointers and similar constructs, and indicate that implementing them with transactions would avoid deadlocks and other synchronization issues. In this case, the authors do not explore the performance of a potential transactional implementation, but the correctness of such a strategy.

In what concerns the synchronization of concurrent data structures with HTM, opinion is divided on the performance benefits of transactions. For example, David et al. [5] report an increase in throughput of at most 5% when using HTM for concurrent search data structures, considering the improvement as negligible. On the other hand, Bonnichsen et al. [2] present a concurrent ordered map implementation with HTM that performs up to 3.9 times faster than the state of the art.

In this paper, we apply the guidelines that recommend enhancing C++ smart pointers with transactional support, thus avoiding specific concurrency issues, and evaluate the potential performance improvement when using HTM on concurrent data structures.

3 Transactional Pointers

We call *transactional pointer* a C++ smart pointer that protects an object with a hardware transaction and does not modify its reference count. The goal is to avoid the execution of undesired atomic operations on the shared reference counter of an object. The transaction intercepts hazardous accesses to the object (e.g., a non-transactional access trying to release a pointer still in use) and a safe path is chosen.

3.1 Algorithm

The original algorithm for C++ smart pointers is straightforward: when the pointer is created, it contains a raw pointer and a control block for this reference. The reference count is initialized with 1. As seen in Fig. 1, when a new copy of the same pointer is created, they will have in common the reference and the reference count field, which is updated by atomic increment. Every time when a copy of the pointer is destructed, the shared reference count is atomically decreased by one, and the rest of the object destroyed. If there is only one reference left, then the memory is automatically freed. This allows the application to function without any risk of incorrect accesses, dangling pointers or memory leaks.

Our goal was to eliminate the atomic operations on the shared reference count, while keeping the reference protected from memory hazards. In order to do that, we defined a constructor that initializes the reference count with 0 and tries to start a hardware transaction. Inside the transaction, we read a shared field of the transactional pointer, called `state`. In this way, `state` is added to the read-set of the transaction and automatically monitored in hardware. Any other thread that tries to modify this field will cause an abort. If there is no abort, the application will continue its execution, using the transactional pointer protected by the transaction. If a conflict or another event causes the currently running transaction to abort, the transactional pointer will follow a fallback path corresponding to the original implementation of the smart pointers, i.e., the reference count is initialized with the number of references of the smart pointer that we are copying and atomically incremented, or with 1 if it is a new smart pointer. When the transactional pointer is destroyed by the application, we check if there is a transaction running: if yes, the transaction commits; otherwise, the object is destroyed in the same way as a normal smart pointer.

Further on, we modified the algorithm to support batching. More specifically, multiple transactional pointers can be added to an already started transaction, without having their reference count modified and without starting a transaction on their own. In order to achieve this, the constructor exploits an additional parameter indicating whether the transactional pointer in question is the first one in the batch (or a single pointer that needs to be protected) or it needs to be added to an already existing transaction. In the former case, the algorithm follows the steps described above. In the latter, we take advantage of the fact that all initializations happen inside the transaction. Thus, we do not need to specifically read the `state` field anymore. Finally, we add a supplementary

Algorithm 1 Transactional pointer implementation

```
 1: function TX_PTR::INIT(boolean add_to_tx, smart_ptr ptr)
 2:     this.ptr ← ptr
 3:     this.refcount ← 0
 4:     this.state ← ptr.state
 5:     this.add_to_tx ← add_to_tx
 6:     if this.add_to_tx ∧ is_fallback then
 7:         FALLBACK(ptr)
 8:     end if
 9:     if ¬this.add_to_tx then
10:         if TX_START() then
11:             READ(this.state)
12:             is_fallback ← false
13:         else
14:             is_fallback ← true
15:             FALLBACK(ptr)
16:         end if
17:     end if
18: end function

19: function TX_PTR::DESTROY()
20:     WRITE(this.state)
21:     if ¬this.add_to_tx ∧ TX_TEST() then
22:         TX_END()
23:     end if
24: end function
```

check in the destructor of the transactional pointer: we only try to commit the transaction when the pointer that started it is destroyed. This design assumes a scenario in which:

- either all pointers die at once (e.g., at the end of a function in which they have been created), in which case they are destroyed in the reverse order of creation, thus making the first pointer to be destroyed last and keeping the transaction that protects all pointers running until it is safe to commit; or,
- the pointers added to the transaction are explicitly destroyed before the pointer that started the transaction.

This is particularly convenient for applications using well-delimited groups of pointers, such as some operations on classical data structures.

The above steps are summarized in Algorithm 1. We call `tx_ptr` the data type that enhances C++ smart pointers with hardware transactions. The constructor creates a transactional pointer from an already existing smart pointer, which is passed as a parameter. We use `this` to designate the current transactional pointer being created. We cover in this pseudo-code the extended algorithm suitable both for batching pointers as well as for single transactional pointers. Therefore, the constructor features a boolean parameter `add_to_tx` that indicates whether the current pointer has to be added to a running transaction or start a new one by itself. If it is the first pointer (Line 9), it tries to start a transaction. All subsequent transactional pointers will be monitored by the same transaction and will not attempt to start a new one. If the transaction starts, we read the `state` field, as mentioned; otherwise, the algorithm takes the fallback path. The call to a generic function `READ()` (Line 11) stresses the idea of reading the field inside the transaction, without entering into implementation details. The

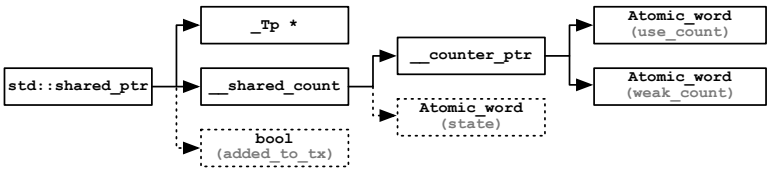


Fig. 2. Class structure of `std::shared_ptr`, with the additional fields for `tx_ptr` in dashed lines.

variable `is_fallback` is a thread-local variable, set when the first pointer takes the fallback path. When a transaction aborts, all changes are rolled back and the execution is restarted. This means that all the added pointers will run their constructor from the beginning, following the path in Line 6. In other words, all transactional pointers will take a non-transactional path, similar to a classical C++ smart pointer. While the transaction is running correctly, `is_fallback` remains false. We claim that the presented algorithm correctly implements a smart pointer structure, while aiming to reduce the overhead of atomic operations.

3.2 Implementation

We built our transactional pointers on top of the `std::shared_ptr` structure in C++. In particular, we extended the `std::shared_ptr` class with a new constructor and modified several internal methods in order to accommodate the transactional logic. As such, a `tx_ptr` can simulate the normal behaviour of a classical smart pointer and `tx_ptr`s can be created from `std::shared_ptr`s.

The `std::shared_ptr` class is implemented as a C++ template, with the raw pointer (of generic type _Tp) and a variable of type `__shared_count` (Fig. 2) as the main fields. The latter is the class that implements the shared reference count object. The reference count object contains a pointer to two counters: `use_count` and `weak_count`. The role of the latter is out of the scope of this work. The former contains the number of references a pointer has throughout the execution. We added in the diagram with dashed lines the necessary fields for implementing `tx_ptr`:

- A boolean variable in the main class, with the aim of indicating which pointers are to be added to the existing transactions or start a new one. This information is critical in the destructor when using batching, since the transaction must be committed only when all pointers in the group have been destroyed.
- The shared `state` field in the reference count class. This field is initialized in the constructor and read inside the transaction, in order to be monitored in hardware. It is further modified in the destructor. Thus, if any other copy of the same pointer tries to destroy the pointer and deallocate the memory, writing the `state` field forces the transaction to abort and the `tx_ptr` to restart as a normal smart pointer.

We implement the transactional memory operations on two different architectures: Intel Haswell and IBM POWER8. While there are several subtle differences in the APIs and the underlying HTM implementations, most of our code is common to both architectures.

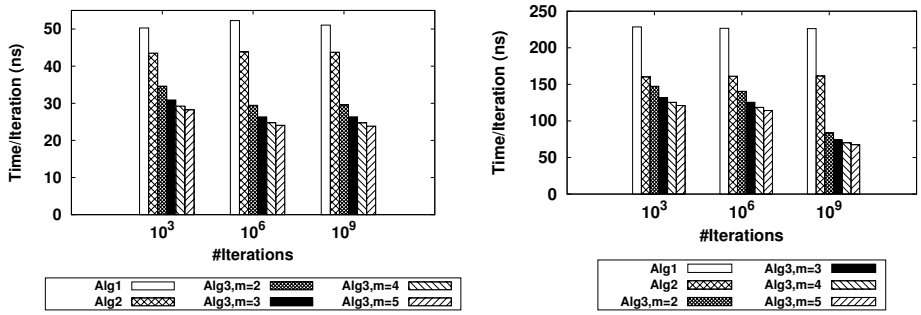
4 Evaluation with Micro-benchmarks

In order to have a preliminary idea of the benefits of our transactional pointer implementation over the original C++ smart pointers, we devised two micro-benchmarks. This enabled us to test both implementations in mono-threaded and multi-threaded scenarios, with or without batching, on two different architectures: Intel Haswell and IBM POWER8.

4.1 Mono-threaded Scenario

We want to evaluate the possible gains of replacing atomic operations with hardware transactions. We developed a mono-threaded micro-benchmark for studying how many transactional pointers have to be packed in a transaction in order to improve the performance over a pair of atomic operations, when the application runs on a single thread. The micro-benchmark consists of the scenarios presented in Algorithm 2. By *tx shared pointer* we refer to a `tx_ptr` implementation. In the first scenario, starting at Line 1, we measure the time it takes to repeatedly create and destroy a normal C++ shared pointer, for a fixed number of iterations. As previously mentioned, when the pointer is created, an atomic increment is performed on the shared reference count; likewise, an atomic decrement is performed when the pointer is destroyed. This strategy reflects the performance when using a pair of increment/decrement atomic operations for `num_iter` iterations. The second scenario, starting at Line 7 replaces the pair of atomic operations in each iteration with a hardware transaction. The third scenario (Line 15) groups multiple create/destroy operations (i.e., multiple iterations) in a transaction. It behaves identically to the second scenario when $m = 1$.

We implemented and tested the micro-benchmark on two different platforms: a 4-core (2 threads per core) Intel Haswell@3.40GHz machine with 12GB RAM and a 10-core (8 threads per core) IBM POWER8@3.42GHz with 30GB RAM, both with fully integrated HTM support. For the third scenario at Line 15, we varied m from 2 to 5, since for values greater than 5 the performance was visibly better than the original implementation of shared pointers. We observed that the measured time was varying during the first executions of the benchmark. In order to have accurate results, we first ran the benchmark 50 times until the results were stable (standard error deviation less than 1%). Subsequently, we considered the average value per iteration over another ten runs for each version of the benchmark. We tested for 10^3, 10^6 and 10^9 iterations. The execution time of one iteration is measured with the system's high resolution clock. Figure 3 shows the performance in all scenarios for the mentioned values of m and number of iterations, both on the Haswell machine and on POWER8. On the Y axis we have the time in nanoseconds per iteration. We made the following observations:



(a) Intel Haswell (b) IBM POWER8

Fig. 3. Mono-threaded performance (time per iteration) for repeated create/destroy of: original shared pointer (Algorithm 2, Line 1), transactional pointer with one transaction per iteration (Algorithm 2, Line 7) and transactional pointer with one transaction per m iterations (Algorithm 2, Line 15) with $m = 2, 3, 4, 5$.

Algorithm 2 Scenarios for the mono-threaded micro-benchmark

```
 1: function SCENARIO_1
 2:     for i ← 1, num_iter do
 3:         p ← new shared pointer
 4:         delete p
 5:     end for
 6: end function

 7: function SCENARIO_2
 8:     for i ← 1, num_iter do
 9:         BEGIN-TX
10:         p ← new tx shared pointer
11:         delete p
12:         COMMIT-TX
13:     end for
14: end function

15: function SCENARIO_3
16:     for i ← 1, num_iter/m do
17:         BEGIN-TX
18:         for i ← 1, m do
19:             p ← new tx shared pointer
20:             delete p
21:         end for
22:         COMMIT-TX
23:     end for
24: end function
```

1. When running on a single thread, using a single hardware transaction per iteration results in better performance than a pair of atomic operations. In other words, the second scenario (Line 7) performed better than the first (Line 1) for any number of iterations on both platforms.
2. The performance improves when m increases (up to a certain threshold when the group of instructions becomes too large and the transaction overflows).

In conclusion, according to the presented mono-threaded benchmark, a hardware transaction should be able to replace a single pair of atomic operations without affecting the performance of the application. The application would gain if multiple pairs of atomic operations were replaced by a single hardware transaction.

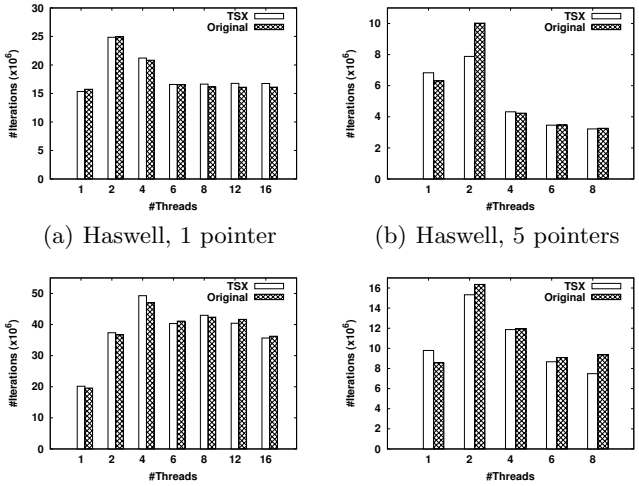
4.2 Short-Lived Pointers

Consider now the following common scenario where a smart pointer is copied to a local variable inside a function, i.e., the copy of the smart pointer has the lifespan of that function. Generally, when creating such a copy, the reference counter is atomically incremented, while at the end of the function, there is an atomic decrement. If the pointer is not accessed concurrently in the meantime, then the increment/decrement operations are unnecessary. We aim to replace this pair of atomic operations with one transaction spanning the entire function. In order to obtain this behaviour, we use the `tx_ptr` pointer defined in Sect. 3.

We created a micro-benchmark that starts multiple threads which share an array of smart pointers. Each thread picks a random element from the shared array and calls a function. In the function, the thread creates a `tx_ptr` copy of the element. Then, it executes several constant-time operations. These operations are meant to simulate a computational workload that accesses the pointer value. If transactional pointers are used, these operations will be executed inside a transaction. Finally, the thread exits the function (which calls the destructor of the transactional pointer, thus committing the transaction). We measure how many iterations of this function are done by each thread in a certain amount of time (customizable by the user). We compare the total number of iterations (i.e., the sum of iterations over all threads) of our `tx_ptr` implementation with the original implementation of smart pointers. We configured our experiments as follows: shared array of 1,000 smart pointers, run time of 5 seconds, 100 constant-time operations. The experiments consist of running the micro-benchmark 10 times for an increasing number of threads on both platforms and taking the average over the total number of iterations in each case.

Figure 4(a) shows our first results with this implementation on the Haswell machine. On the X axis we show the number of threads, while on the Y axis we have the number of iterations performed divided by 10^6 (higher values are better). We tested on up to 16 threads. We observe that starting with 4 threads, our implementation performs better than the original. However, the improvement is less than 4% (on 16 threads). Moreover, on the POWER8 server (Fig. 4(c)) there is almost no performance gain on more than 4 threads, indicating that the transactional implementation on this architecture suffers more from contention than the atomic operations. This result led us to the conclusion that, in a multi-threaded environment where many operations are involved, the creation of a single `tx_ptr` does not bring an improvement over a pair of atomic operations. As an optimization, we enabled batching in the micro-benchmark, i.e., the creation of multiple pointers in a single transaction. The idea was that, if a pair of atomic operations has almost the same overhead as a transaction, then replacing multiple pairs of atomic increment/decrement with a single transaction would improve the performance.

We modified the benchmark as follows: instead of a single random element, each thread now picks several random elements from the shared array (number defined at runtime). It creates a new array with these elements and calls the specific function having this array as a parameter. The function makes `tx_ptr`



(a) Haswell, 1 pointer (b) Haswell, 5 pointers

(c) POWER8, 1 pointer (d) POWER8, 5 pointers

Fig. 4. Number of iterations for one or multiple short-lived `tx_ptr` pointer copies (TSX) and smart pointer copies (Orginal) in a function.

copies of all pointers, using the additional boolean parameter in the constructor in order to indicate which pointers will be added to the running transaction.

Figures 4(b) and 4(d) show the results of this strategy with a group of 5 pointers per transaction. In this scenario, however, contrary to our expectations, the performance actually suffers. We tested on up to 8 threads, pinned to the cores. We conclude that, by trying to optimize the previous results with batching, we also increased the overhead of the transaction with extra operations. This explains why in this setting we could not observe any improvement over the version with one pointer. Given the negligible performance gain of the latter, we deduce that in this scenario using transactional pointers does not have a significant advantage over the original C++ smart pointers.

5 Evaluation with Shared Data Structures

We implemented a simple data structure to showcase the possible performance improvement of `tx_ptr`s over the original implementation of C++ smart pointers. We chose to build a simply-linked list because of the natural occurrence of pointers with a reference count ≥ 1 (they will always be referenced by at least one other pointer until they are removed or the list is destroyed). That allows us to exploit the design of `tx_ptr` and the benefits of transactions for repeated concurrent traversals of the list.

5.1 Implementation

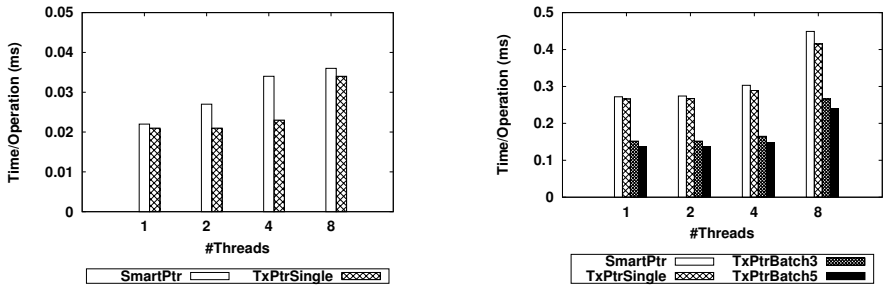
The shared list was implemented in two steps. First, we designed a concurrent liked-list structure only based on shared pointers and *compare and swap* (CAS) operations. For simplicity and reproducibility of our tests, we only inserted elements at the end of the list and we always removed the first element. Basically, in this experiment the implementation behaved like a concurrent queue, with an additional lookup function.

We implemented the data structure using a classical lock-free queue algorithm [9]. The use of C++ smart pointers for the nodes guarantees the correctness when accessing and manipulating elements. We use CAS operations specifically defined for shared pointers in the C++ standard library `libstdc++-v3`, included in the GCC5 release.[4] The result of a CAS operation is repeatedly checked in a loop, until it confirms that the desired operation took place. The insert and delete operations are easily implemented with shared pointers and CAS, by changing atomically the first element with the new node, respectively the last element with the next node in the queue. The lookup function iterates over the list sequentially until it finds the requested value or reaches the end of the list. We considered that the list traversal could benefit the most from our implementation of `tx_ptr`s. The next step was to change the above implementation to use transactional pointers. The only modification needed in the code is replacing the constructor of the pointer that will iterate over the list with the customized constructor defined in Sect. 3.

Our goal was to encapsulate each iteration of the loop in the lookup function in a hardware transaction. In the original implementation, when the pointer iterating over the list passes from a node to the next, it creates and destroys a copy of a smart pointer. As previously mentioned, this is equivalent to a pair of atomic operations. Thus, we replace a pair of atomic increment/decrement with a hardware transaction. In order for the transactional pointers to work transparently in this case, we also extended the overloaded '=' (assignment) operator of C++ smart pointers. More precisely, the first transaction is started when the iterator is initialized in the `tx_ptr` constructor. Then, each time the iterator moves to the next node, the transaction commits and a new one is started (which will last until the move to the next node and so on). If there is a conflict, the transaction aborts and takes the fallback path described in Sect. 3.

Finally, we implemented support for batching multiple pointers in a single transaction. In the case of list traversal, this means that a hardware transaction will span the traversal of multiple nodes. The size of the group of nodes included in a single transaction is customizable. In order to maintain the transparency of the implementation, we could not reuse in this case the batching strategy described in Sect. 4.2. Rather, we implemented an internal counter for `tx_ptr` and modified the '=' operator to commit and start a new transaction when the counter indicates the end of a batch. Whenever the transaction aborts due to a

[4] In C++11 the operation `atomic_compare_exchange_weak(p, expected, desired)` checks if `p` has the same value as `expected`: if so, the value `desired` is atomically assigned to `p`; otherwise, `expected` becomes equal to `p`.



(a) No batching (60% lookup) (b) Batching (100% lookup)

Fig. 5. Execution time per operation for a concurrent queue implemented with smart pointers and transactional pointers, with (a) a transaction per iteration during lookup and (b) multiple iterations grouped in a single transaction.

conflict, all the changes made to the group of pointers are rolled back and all pointers are recreated as common C++ smart pointers with reference counting.

5.2 Evaluation

We developed a benchmark for comparing the performance of smart and transactional pointer implementations of the concurrent queue. The benchmark works as follows: we initialize the queue and populate it with elements. We start a number of threads that share the queue. Each thread applies insert, delete and lookup operations on the shared queue by a given ratio. We measure the time it takes each thread to finish the associated operations. In order for all threads to have comparable workloads, we generate the workloads before starting the threads. Specifically, we generate a random succession of operations according to the proportions given for each type of operation. Then, we generate a list of elements that will be inserted in the queue, and a list of elements that will be looked up, based on the elements that are inserted. Given the dynamic character of the benchmark (a large number of concurrent insert and delete operations), not all the elements in the lookup list will be found in the queue at the moment when the operation is performed.

First, we experimented with the implementation based on the original C++ smart pointers and our simple transactional version (without batching). We tested on a 4-core Intel Haswell server, on up to 8 threads pinned to the cores. We set the list to be initially populated with 1,000 elements. Each thread had to execute 10^6 operations on the shared list, out of which 20% insert, 20% delete and 60% lookup operations. We measured the time with the `high_resolution_clock` C++ function. We ran each test 10 times, after first observing that the results were stable, with negligible variations from a run to another. For each run we took the maximum between the times reported by each thread, then computed the average over the 10 runs. The results for this test are shown in Fig. 5(a). We

observe that our implementation does not perform notably better than the original. However, this result indicates that even if we replace a single pair of atomic operations with a hardware transaction, we already start gaining in performance.

We then tested the transactional version with batching enabled. Since the only difference between the two implementations of the concurrent queue (i.e., with `shared_ptr` and with `tx_ptr`) is in the way in which the lookup function works, we focused on stressing and comparing strictly this operation. Thus, we modified the previous configuration to run a workload of 100% lookup operations, for 10^6 operations per thread, on a 10^4-element shared array. At least half of the elements that will be looked up by the benchmark are found in the initial shared array. Figure 5(b) shows the results in this scenario for the implementation with the original C++ smart pointers, as well as transactional pointers with one transaction per pointer, one transaction for a group of 3 pointers, and one transaction for a group of 5 pointers. We make the following observations: first, when grouping 5 pointers in a transaction, i.e., replacing 10 atomic operations with a hardware transaction, we see an improvement of up to 50% in the execution time. Second, we observe that the performance increase is more spectacular when passing from no batching to a group of 3 pointers than from a batch of 3 to one of 5 pointers. While the batch size increases, the performance improvement will reach a plateau and start degrading when the batch becomes too large for being handled properly by a hardware transaction. Finally, we remark that the improvement is less noticeable on 8 threads, because of contention.

6 Conclusion and Future Work

Concurrency and automatic memory management are two key components of today's complex multi-core systems. While the number of cores per CPU keeps increasing, the attention of developers seems to turn more and more towards lock-free algorithms and implementations. Transactional memory, and especially its hardware implementation (HTM), represents a suitable non-blocking solution for concurrency hazards. At the same time, reference counting is a useful form of memory management with interesting properties and synchronization features, where each object is protected from invalid accesses by keeping a shared reference counter. Sometimes the atomic increment/decrement operations on the shared counter prove to be unnecessary and expensive. We considered this to be a promising opportunity for improvement with HTM.

We designed a transactional pointer structure on top of the C++ `shared_ptr`, which uses reference counting for correctly managing the memory. Our goal was to replace the atomic operations needed for the creation/destruction of the smart pointer with a hardware transaction. We experimented with micro-benchmarks, in mono- and multi-threaded settings, on two different architectures and with the possibility of batching multiple pointers in a transaction. We also compared the performance of the original and transactional implementations on a concurrent queue of smart pointers. We believe that the results provide valuable insights into which scenarios would benefit most from using a transactional pointer.

Given the promising results for the concurrent queue (up to 50% improvement on the execution time for lookup operations), we plan to further pursue this idea and implement more complex data structures with transactional pointers. A future objective could be the implementation of a transactional pointer specialized for concurrent data structure traversal.

References

1. Blackburn, S.M., McKinley, K.S.: Ulterior reference counting: Fast garbage collection without a long wait. In: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications. pp. 344–358. ACM, USA (2003)
2. Bonnichsen, L.F., Probst, C.W., Karlsson, S.: Hardware transactional memory optimization guidelines, applied to ordered maps. In: Trustcom/BigDataSE/ISPA, 2015 IEEE. vol. 3, pp. 124–131. IEEE (2015)
3. Cascaval, C., Blundell, C., Michael, M., Cain, H.W., Wu, P., Chiras, S., Chatterjee, S.: Software transactional memory: Why is it only a research toy? Queue 6(5), 40:46–40:58 (2008)
4. Crowl, L., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Integrating transactional memory into C++. In: Workshop on Transactional Computing (2007)
5. David, T., Guerraoui, R., Trigonakis, V.: Asynchronized concurrency: The secret to scaling concurrent search data structures. In: Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 631–644. ACM (2015)
6. Dragojević, A., Felber, P., Gramoli, V., Guerraoui, R.: Why STM can be more than a research toy. Commun. ACM 54(4), 70–77 (2011)
7. Gottschlich, J.E., Boehm, H.J.: Generic programming needs transactional memory. In: The 8th ACM SIGPLAN Workshop on Transactional Computing (2013)
8. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture. pp. 289–300. ACM, USA (1993)
9. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., USA (2008)
10. Ni, Y., Welc, A., Adl-Tabatabai, A.R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., Tian, X.: Design and implementation of transactional constructs for C/C++. In: Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications. pp. 195–212. ACM, USA (2008)
11. Shahriyar, R., Blackburn, S.M., Frampton, D.: Down for the count? Getting reference counting back in the ring. In: Proceedings of the 2012 International Symposium on Memory Management. pp. 73–84. ACM, USA (2012)
12. Shahriyar, R., Blackburn, S.M., Yang, X., McKinley, K.S.: Taking off the gloves with reference counting Immix. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. pp. 93–110. ACM, USA (2013)