Opportunities and pitfalls of multi-core scaling using Hardware Transaction Memory*

Zhaoguo Wang†, Hao Qian‡, Haibo Chen‡, Jinyang Li§
† *School of Computer Science, Fudan University*
‡ *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*
§ *Department of Computer Science, New York University*

Abstract

Hardware transactional memory, which holds the promise to simplify and scale up multicore synchronization, has recently become available in main stream processors in the form of Intel's restricted transactional memory (RTM). Will RTM be a panacea for multicore scaling? This paper tries to shed some light on this question by studying the performance scalability of a concurrent skip list using competing synchronization techniques, including fine-grained locking, lock-free and RTM (using both Intel's RTM emulator and a real RTM machine). Our experience suggests that RTM indeed simplifies the implementation, however, a lot of care must be taken to get good performance. Specifically, to avoid excessive aborts due to RTM capacity miss or conflicts, programmers should move memory allocation/deallocation out of RTM region, tuning fallback functions, and use compiler optimization.

1 Introduction

The increasing number of cores in a commodity machine has made multicore scaling a grand challenge to many software applications. To improve performance and scaling, programmers have to resort to a fined-grained lock scheme, which is often difficult to design and implement. Furthermore, prior experiences show that even a small contention on a cache line due to locking may collapse whole application performance [4].

Hardware transactional memory (HTM) has been proposed by Herlihy and Moss two decades ago to simplify multicore synchronization with good performance [14]. Recently, Intel's Haswell processor shipped with HTM support, a landmark event signalling the mass availability of HTM. Intel's HTM mechanism, called Restricted Transaction Memory (RTM), is a best-effort transaction memory in two aspects. First, a transaction's working set must fit in the per-core private cache. Otherwise, the transaction will be aborted. Second, a transaction is also aborted when encountering exceptions and faults, such as a syscall instruction or page fault.

Can Intel's RTM simplify systems applications and enable them to scale to many cores? This paper tries to provide some insights on this question by studying the performance scalability of a concurrent skiplist. We choose skiplist [16] as it is a key data structure in many important systems applications such as in-memory sorted key/value stores and databases. The scalability of the skiplist often limits the scalability of the entire database or key-value systems. Specifically, we derive our skiplist implementation from the one in LevelDB [3], a popular key-value store from Google. The current implementation of LevelDB updates the skiplist serially, which becomes its main scalability bottleneck.

We implemented both fine-grained locking and lock-free skiplists and our experiments on a 40-core machine show that both can achieve scalable performance. However, the correctness reasonings for both implementations are non-trivial. By comparison, our RTM-based skiplist is simple to implement and prove correct. However, our experiments using Intel's official RTM emulator and a 4-core Haswell machine show that a lot of care must be taken to avoid excessive transaction aborts.

Based on our experience with scaling an RTM-based skiplist, we make several observations and suggestions. (1) Move memory allocation outside a transaction re-

*This work was supported by China National Natural Science Foundation under grant numbered 61003002.

gion. Doing so avoids page faults and system calls (e.g., mmap) that will abort the transaction; (2) Use well-tuned fallback functions according to workload behavior; (3) Employ compiler optimizations. Compiler optimizations can significantly reduce memory accesses in a transaction and result in fewer conflicts. (4) the real Haswell processor can accomodate a larger working set for reads than for writes, a bahavior different from Intel's RTM emulator.

2 Background and Related Work

Transactional Memory Transactional memory (TM) [14] allows a group of instructions to execute in an atomic way and be completely isolated among cores in a multi-processor system. Here, we only consider hardware transactional memory (HTM) as it has low overhead and has recently been widely available in commodity processors. HTM implementations on real hardware have always been best-effort with no guaranteed forward progress. Examples include Sun's Rock processor [5], AMD advanced synchronization family [7], Intel transitional synchronization extensions (TSX) [1] and IBM Blue Gene/Q [17]. Among these proposals and implementations, the recent release of Haswell processor with Intel TSX marks the wide availability of HTM on the mass market.

Many have reported their experience with using HTM. Dice et al. [11, 10] use HTM of Sun's Rock prototype processor to implement concurrent data structures such as hash table and red-black tree. Others have used HTM to build synchronization primitives such as read-writer lock [9] and software transactional memory [8], as well as scientific applications [6, 17].

Our work differs from prior performance studies of HTM in two ways. First, ours is the first published study targeting at the Intel RTM. Existing studies used different hardware [17, 5] and their conclusions may not apply to Intel RTM due to the hardware differences. For example, while Dice et al. [11] observed that deferred instructions due to excessive cache misses were the major reason of transaction aborts, our experiences with Intel RTM show that the major reason are from capacity and conflict misses. Second, existing studies focus on compute-intensive scientific applications such as the STAMP benchmark suite [15] or different data structures. Instead, we focus on skip list from a real world key/value store. The asymmetric working set for read and write in Intel RTM leads to different performance characteristics for skip list.

Intel's Restricted Transactional Memory. Intel's RTM introduces three new instructions called *xbegin*, *xend* and *xabort*. The first two marks the start and end of a transaction and the third one explicitly aborts an transaction. Intel's RTM uses the private CPU caches (e.g., L1 and/or L2 caches) to track read and write set of transactions. The conflict detection is handled through existing coherence protocol and a transaction whose memory accesses exceed the capacity of the cache will be aborted. A number of hardware and software events such as context/mode switches, interrupt and I/O instructions will cause a transaction to abort. Intel's RTM does not guarantee forward progress. To ensure good performance, programmers should optimize transactions such that they succeed most of the time in practice. This consideration motivates our study on opportunities and pitfalls of multicore scaling using RTM.

3 Scaling Up Skiplist

A skiplist is a probabilistic data structure commonly used in databases and key/value stores. It is essentially a multi-level sorted linked list, of which the bottom level list includes all nodes and the upper levels contain express links that skip some nodes to ensure O(n) search. An insertion first finds the location to insert and then adds a node of random height with links to forward and backward nodes.

Our performance study uses the skiplist implementation in Google's popular key-value store LevelDB. LevelDB uses skiplists to store its recently written key/values pairs in memory before writing them back to persistent storage. The skiplist in LevelDB treats deletion as a special insertion operation that inserts a key with an empty value.

3.1 Performance of Traditional Techniques

In the current LevelDB implementation, updates to the skiplist are completely serialized, resulting in a significant scalability bottleneck. We implemented a design with fine-grained locking [13]. In this design, we first finds an insertion location and then lock the predecessors of the location and then check if the successors have been changed. If any successor has been changed since the insertion, we release all locks and then retry from the beginning. Otherwise, we insert the key/value pair and release all locks. Both the original skiplist and our fine-grained version are shown in Figure 1.

The fine-grained locking design requires holding all predecessors' locks before insertion. Thus, nodes cannot be inserted into the same position at different levels in the skiplist, even though doing so can be correctly done in parallel. Our lock-free implementation [12] unleashes such parallelism. As shown in Figure 1, the lock-free version leverages the fact that each layer of skiplist is

Search

```
FindPosition(key):
  x ← head
  level ← MaxHeight -1
  repeat
    next ← x.next[level]
    if key > next.key then
      x ← next
    else
      preds[level] ← x
      succs[level] ← next
      if level = 0 then
        return <next, preds, succs>
      else level - -
  until false
```

Origin Insert

```
Insert(key):
  lock()
  < x, preds , succs> ← FindPosition(key)
  height ← RandomHeight()
  n ← NewNode(key, height)
  for i = 0 to height -1 do
    n.next[i] ← preds[i].next[i]
    preds[i].next[i] ← n
  unlock()
```

Lock free

```
LF-Insert(key):
  < x, preds , succs> ← FindPosition(key)
  height ← RandomHeight()
  n ← NewNode(key)
  for i = 0 to height -1 do
    repeat
      succs[i] ← preds[i].next[i]
      while key > succs[i].key do
        preds[i] ← succs[i]
        succs[i] ← preds[i].next[i]
      n.next[i] ← preds[i].next[i]
      succ ←CAS(preds[i].next[i],succs[i],n)
      if succ = succs[i] then
        break
    until false
```

RTM-Largest

```
RLEST-Insert(key):
  beginTX()
  height ← RandomHeight()
  n ← NewNode(key, height )
  <x, preds, succs> ← FindPosition(key)
  for i = 0 to height -1 do
    succs[i] ← preds[i].next[i]
    while key > succs[i].key do
      preds[i] ← succs[i]
      succs[i] ← preds[i].next[i]
    n.next[i] ← preds[i].next[i]
    preds[i].next[i] ← n
  endTX()
```

Fine-grained Lock

```
LOCK-Insert(key):
  height ← RandomHeight()
  repeat
    <x, preds, succs> ←FindPosition(key)
    lockedpreds ← ∅
    for i = 0 to height -1 do
      pred ← preds[i]
      succ ← succs[i]
      if pred ∉ lockedpredd then
        lock(pred)
        lockedpreds← lockedpreds∪{pred}
      valid ← (succ = pred.next[i])
      if !valid then break
    if !valid then
      for all p ∈ lockedpreds do
        unlock(p)
      continue
  n ← NewNode(key)
  for i = 0 to height -1 do
    n.next[i] ← preds[i].next[i]
    preds[i].next[i] ← n
  for all p ∈ lockedpreds do
    unlock(p)
```

RTM-Large

```
RL-Insert(key):
  height ← RandomHeight()
  n ← NewNode(key, height )
  beginTX()
  <x, preds, succs> ← FindPosition(key)
  for i = 0 to height -1 do
    succs[i] ← preds[i].next[i]
    while key > succs[i].key do
      preds[i] ← succs[i]
      succs[i] ← preds[i].next[i]
    n.next[i] ← preds[i].next[i]
    preds[i].next[i] ← n
  endTX()
```

RTM-Read

```
RT-Read(key):
  beginTX()
  <x, preds, succs>←FindPosition(key)
    if x.key = key then return x.value
    else return false
  endTX()
```

RTM-Small

```
RS-Insert(key):
  height ← RandomHeight()
  n ← NewNode(key, height )
  <x, preds, succs> ← FindPosition(key)
  beginTX()
  for i = 0 to height -1 do
    succs[i] ← preds[i].next[i]
    while key > succs[i].key do
      preds[i] ← succs[i]
      succs[i] ← preds[i].next[i]
    n.next[i] ← preds[i].next[i]
    preds[i].next[i] ← n
  endTX()
```

RTM-Tiny

```
RT-Insert(key):
  height ← RandomHeight()
  n ← NewNode(key, height )
  <x, preds, succs> ← FindPosition(key)
  for i = 0 to height -1 do
    beginTX()
    succs[i] ← preds[i].next[i]
    while key > succs[i].key do
      preds[i] ← succs[i]
      succs[i] ← preds[i].next[i]
    n.next[i] ← preds[i].next[i]
    preds[i].next[i] ← n
    endTX()
```

FindPosition: Find the position of a key in the skiplist (Original version).
Origin Insert: Original skiplist insertion by using a global lock protection.
Fine-grained Lock: Concurrent skiplist insertion by using fine grained lock.
Lock-free: Non-blocking skiplist insertion by using CAS.
RTM-Largest: Include the entire write process in the TX.
RTM-Large: Move the node creation out of the TX.
RTM-Read: Read function paired with the RTM-Large.
RTM-Small: RTM version paired with the Fine-grained lock.
RTM-Large: RTM version paired with the Lock-free.

Figure 1: Skiplist implementation: original, fine-grained locking, lock-free and the corresponding TM versions.

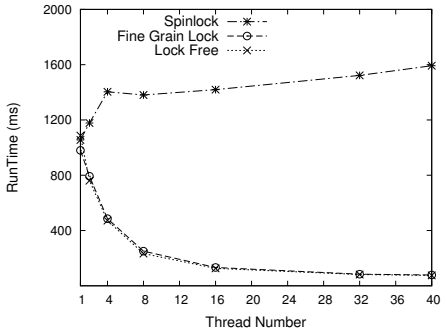

Figure 2: The Skiplist Performance Evaluation

an independent linked list and uses multiple layers of lock-free linked list to construct the skiplist. However, the lock-free version gets only a small amount of benefit compared with the fine-grained locking version. The reason is that the average height of every node is about 1.3, and thus per layer parallelism does not get much more benefit.

Figure 2 shows the performance of different skiplist implementations on a 40-core machine. Our workload issues a total of 1 million key/value pairs to the skiplist. As expected, the performance of the original skiplist drops with more cores, while the fine-grained locking and lock-free versions scale reasonably well on the 40-core machine.

Although we are able to scale the skiplist on multicore using traditional techniques, our experiences show that it is usually not easy to construct correct fine-grained locking and lock-free implementations as they usually requires non-trivial ordering and atomicity semantic guarantees. Further, our current skiplist implementation is a simplified one without real deletion which is much harder to guarantee and reason about correctness.

3.2 Performance of RTM with Emulator

We first study RTM-based skiplists using the official Intel SDE [2]. Compared to the real hardware, the SDE can emulate many more cores for collecting statistics regarding transactional aborts. We use GCC-4.8.0 and the SDE version is 5.38.0. The SDE is configured using L1 cache to track the read/write set for transactions, like that in the real hardware [1]. The L1 cache is 32 KB and 8-way set associativity as in a typical Intel processor. In the default implementation, we simply retry if a transaction aborts. We will discuss the situation if we acquire a global lock upon transaction aborts in the fallback routine in next section.

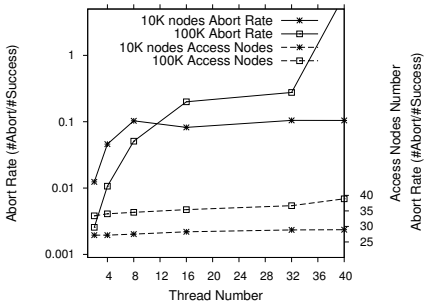

Figure 3: *The Working Set Evaluation of Large TX*

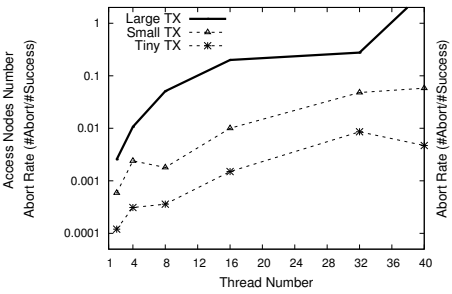

Figure 4: *100K Nodes Workload*

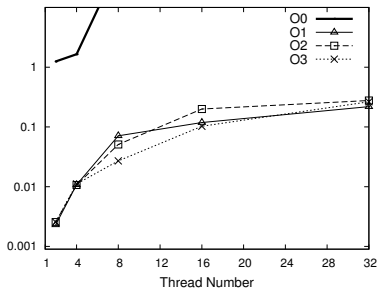

Figure 5: *Different Compile Optimization Options*

The most straightforward approach is to enclose the entire insertion process of the skiplist between *xbegin* and *xend*, such that the RTM can ensure the atomicity and isolation of multiple writers, as shown in *RTM-largest* in Figure 1. For this method, there are a large number of transaction aborts due to page faults. This is because, creating a new node may experience a page fault since the constructor will write the newly allocated memory, or ultimately make a system call (e.g., mmap) in the memory allocator to extend the heap. For a page fault, a TX will abort and a simple retry will again cause an abort due to faulting on the same address such that no forward progress can be made. Similarly, retrying a TX due to system calls will experience persistent aborts[1]. To reduce transaction aborts due to page faults and system calls, we should move the creation (and deletion) of nodes out of the transactions, forming the skiplist implementation in RTM-large in Figure 1. After this, the transaction aborts due to page faults and system calls are mostly eliminated.

For RTM-large, we vary the working set by increasing the amount of nodes from 10K, 100K to 1M. We also calculate the number of nodes read in a transaction on average. As the complexity for the search operation is O(log(n)), the working set does not increase notably when the skiplist size increases dramatically. We measure the abort rate by dividing the number of abort operations by the number of success ones. As shown in Figure 3, the abort rate increases with both the number of cores and the workload size. Further, our evaluation shows that the skiplist incurs too many transaction aborts so that it cannot make forward progress when inserting 1M nodes. This is due to conflict aborts where a TX conflicts with others due to conflict memory accesses, as well as capacity misses such that a cache line in the read/write set will be evicted out of the L1 cache. Both cache capacity miss and cache set conflict miss will

[1]As Intel's RTM emulator is a user-level one, we currently can only observe transaction aborts due to mmaps.

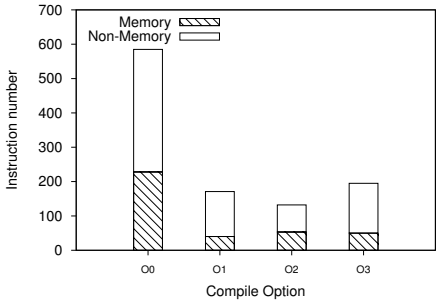

Figure 6: Instruction Generated In the TX Body

cause the cache line eviction. In this case, as the working set of the TM region still fit into the L1 cache, the major reason is due to the cache set conflict miss.

Though the code size for the TM region is not huge, the working set of the TM region may be dynamic according to the workload. When there are a large number of threads, even a small increase in the working set can cause large abort events. For RTM-large, when inserting 1M nodes, although the working set is smaller than L1 cache size, the application incurs extremely frequent aborts. The abort is caused by the cache set conflict. As the working set increases, the probability of cache set conflict increases as well.

The main reason for the cache set conflict misses is that many nodes in the skiplist are allocated with the same page offset by multiple writers. Hence, when a writer searches the skiplist, it will fetch multiple nodes with the same page offset, which unfortunately maps to the same cache set, where only 8 cache lines could be held at a time. Hence, some victims are inevitably being evicted out, causing transaction aborts.

We also conduct a simple evaluation to study how compiler optimization affects efficiency in RTM. We compile the application using different compile optimization options. As shown in the Figure 5, our

skiplist implementation experiences an excessive amount of transaction aborts and cannot execute beyond 8 cores with "-O0". With other compile flags, due to the significantly reduced memory accesses, transaction aborts reduces as well and can execute up to 32 cores. Further, according to Figure 6, the "-O1" is with the smallest amount of memory instructions and thus can outperform others options after 16 cores. This means that using compiler optimization towards less memory accesses may benefit transactional execution.

After applying the above observations, we find that even using a conservative way of defining transactions, we can still get reasonably good performance with only a few amount of transaction aborts. This indicates that transaction memory may indeed help simply multicore scaling with reasonable performance.

In the following, we further apply RTM to the above fine-grained locking and lock-free versions and collect the performance statistics. RTM-small and RTM-tiny in Figure 1 show the corresponding code. Figure 4 shows the performance for different algorithms when inserting 100K nodes. We can see that, the tiny TX algorithm has lower abort rate than large TX and the small TX gets the best performance. This again may require some efforts as the the case of using fine-grained and lock-free efforts, though the code is less complex and may be easier to reason about.

3.3 Performance of RTM on Real Hardware

This section describes our experiments on an Intel Haswell 4770 CPU chip (4 core) with 32GB memory.

Comparing the real hardware with the emulator. Haswell uses L1-cache to track read set and write set [1]. When running the application on the emulator, any eviction for read or write set cache line from L1 cache will abort the transaction. For the current hardware implementation, an eviction of write cache line will abort the transaction. But an eviction of a read set cache line does not always abort the transaction, because Haswell may use an implementation-specific second-level structure to track these lines. As a result, on our Haswell processor, the read set is much larger than that for the write set (approximately 270KB vs. 26KB).

We also find that we can run larger workload sizes on the hardware than those on an emulator. We use 10M nodes as default workload size for the following evaluations. To evaluate the working set effect, we vary the workload sizes from 10K to 100M. Figure 7 shows the abort rate when running a single thread: the abort rate increases with the workload not only because larger working set incurs more cache line eviction that aborts the transaction, but also because longer execution time means more system events (etc. timer interrupt) that abort the transaction. Figure 7 also shows the abort rate when run two and four threads, they both have a turning point at which there is a lowest abort rate. The reason is because when the workload grows, although the abort caused by cache line eviction and system event increases, the abort event caused by shared data conflict decreases as the size of skiplist grows.

We also evaluat the RTM-Large algorithm with different compiler options on the real hardware, although workload sizes are different, we still get similar results with those on the emulator."O1" has the lowest abort rate compared with other options, but O2 and O3 have slightly better performance.

For the evaluation to compare with RTM-Small and RTM-Tiny, the evaluation result is also similar with result on the emulator. RTM-Large has highest abort rate for it has the largest RTM region, RTM-Large has abort rate from 5% to 17% when run different threads, but the RTM-Tiny's abort rate range from 0.4% to 0.9%. We also evaluate the execution time, it seems that although RTM-Large has higher abort rate, it incurs only a little performance overhead on a small number of cores (0.35% to 4.7% compared with RTM-Tiny). To illustrate this, we also evaluate how many CPU cycles wasted due to transaction abort, the abort cycles range from 2.42% to 6.86% of the total execution time. This means although RTM-Large has higher abort rate, it does not hurt the performance significantly in a small scale processor.

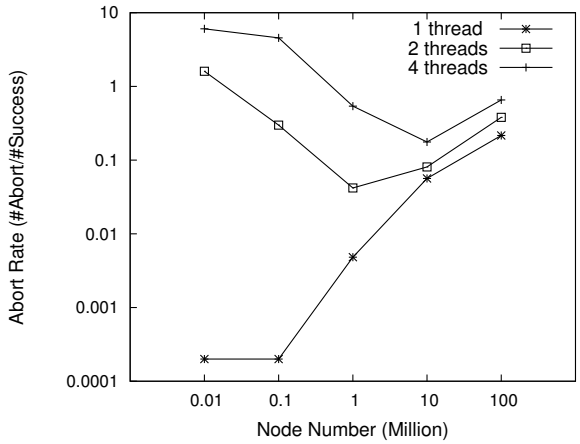

Figure 7: Abort rate of different workload on real hardware

Fallback Functions When a transaction aborts, the CPU will resume the execution from a fallback handler specified by the xbegin instruction. The default implementation just simply retries from the *xbegin* instruction. However, since the transaction may never make forward

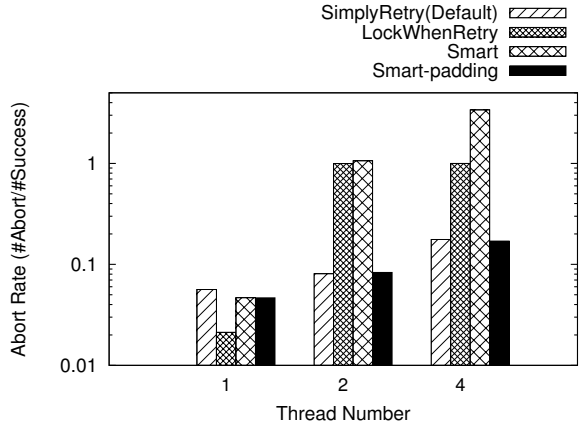

Figure 8: Abort rate of the different fallback handler implementations

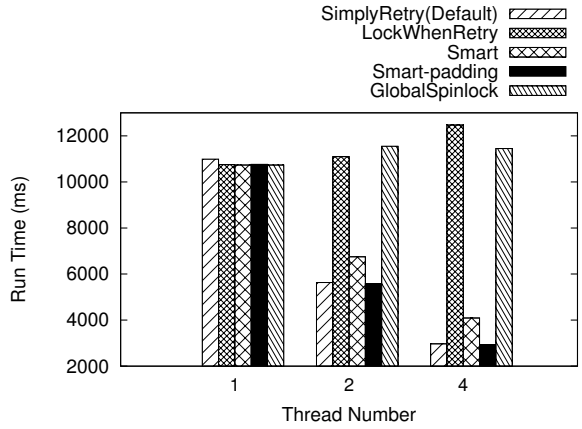

Figure 9: Run time of different fallback handler implementations

progress, programmers may need to combine the usage of RTM with a global spinlock (i.e., transactional lock elision) to ensure forward progress. The simplest scheme is when a transaction starts, the lock is read to check if it has been held to put it into read set. If a transaction aborts because the lock is acquired or other abort events happen, it will try to acquire the lock in the fallback function immediately.

Figure 8 and Figure 9 shows the runtime and abort rate of different fallback algorithms with the default implementation (simply retry). The figures also show the implementation using a global spinlock to protect the entire insertion routine. When simply acquiring a lock in the fallback handler, it performs better than directly using a global spinlock to protect the insertion code for one thread. The reason is that, holding a lock instead of simply retry, avoids the abort events caused by cache eviction or system events happened in the transaction region. However, when running 2 or 4 threads, it has higher abort rate and thus lower performance. The reason is that a thread trying to acquire lock will abort all concurrent transactions. In our workload, most of the time is spent in the transaction body, as a result it incur notable waste of CPU cycles and heavy contention.

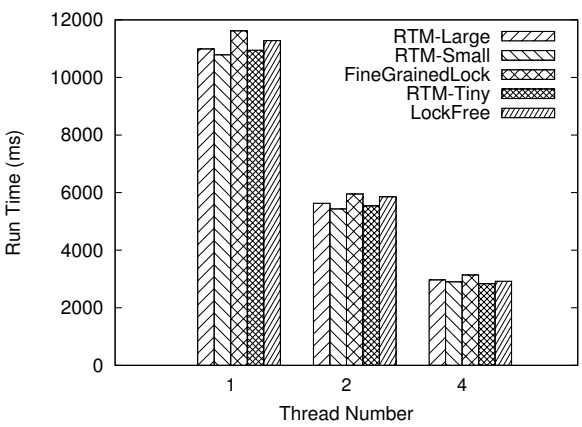

Figure 10: Run time of different skiplist algorithms

For a single thread, the main reasons for abort are system events and cache line eviction. For 2 or 4 threads, the main reasons are conflicts caused by shared data accesses. To this end, we design a smart algorithm in the fallback function to reduce abort rate. First, we classify the abort events into conflict aborts caused by shared data access, capacity abort caused by the internal buffer overflow and other aborts caused by some system events like timer interrupt. We set independent threshold for each abort reason and acquire the lock when a single threshold is reached. The thresholds for capacity abort, conflict abort and all remaining reasons are 4, 8 and 32 accordingly, according to our tuning.

With these heuristics in the fall back function, our evaluation confirms that the abort rate has been decreased. However, it still has higher abort rate than the simply retry version. This is because the global spinlock shares the same cache line with other global shared data. To avoid false conflict, we add the padding to let the spinlock occupy a cache line exclusively. After padding, the performance and abort rate are both similar with the simply retry version. As the transaction execution time is about 97% - 98% of the total time, acquiring a lock will not only avoid the future local abort, but also aborts in concurrently executing transactions.

Comparing with traditional techniques. We compare the performance of RTM-Large with that of global fine-grained lock and lock free implementations. Figure 10 shows that RTM-Large has comparable performance scalability in a 4-core machine. For this case,

RTM can get comparable performance with the lock-free algorithm, but it makes programming much more easier and easy to prove the correctness.

3.4 Lessons Learned

Lessons 1: Avoiding memory allocation/deallocation in a TM region. In practice, we should try best to avoid creating new objects in a TM region. This may avoid system calls as well as page faults inside a TM. Further, if a TM region may touch a memory region that may experience page faults, it would be better to move the page fault instruction out of the transaction. If the instruction cannot be moved out, it might be a good idea to pre-touch the memory outside transactions. Otherwise, it is necessary to touch the faulting address in the programmer-specified fallback routine.

Lessons 2: Compiler optimization is important for transaction execution. Code quality may significantly affect the transaction execution efficiency. As all memory accesses inside a TM region will be tracked by the processor, using compiler optimization that reduces memory accesses inside a TM region may reduce transactional abort due to cache set conflict misses. For some cases, we speculate that using a biased compile ration scheme that trades code quality in a non-TM region for that in the TM region may be beneficial. For example, allocating more registers for a TM region code might be beneficial.

Lessons 3: RTM prefer read than write. Although L1 cache is used to track the read/write set. It seems a program can read more than L1 cache in the RTM region. A code region that has more read than write would be an ideal case for RTM (e.g., skiplist insertion). However, the read set is depending on the micro-architecture implementation and access pattern.

Lessons 4: When using RTM for lock elision, fallback handler should be tuned according to workload. Acquire a global lock in the fallback handler will abort any concurrent transactions and may incur heavy contention. Instead of simply acquiring a lock, the fallback handler should be tuned with the workload. The abort reasons, the workload execution behavior and the performance goal (fairness, real time, scalability) all will affect the policy used in the fallback handler.

4 Conclusion and Future Work

This paper described our preliminary study in leveraging the recent Intel's restricted transactional memory (RTM) to scale up a skiplist implementation. Our study led to several lessons regarding RTM. In our future work, we will validate these lessons on other concurrent data structures and apply such findings to scale up real, large applications. Further, we will study how other configurations such as hyperthreading affect our observations.

References

[1] Intel 64 and ia-32 architectures optimization reference manual. https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[2] Intel software development emulator. http://software.intel.com/en-us/articles/intel-software-development-emulator.

[3] Leveldb: A fast and lightweight key/value database library by google. http://code.google.com/p/leveldb/, 2013.

[4] BOYD-WICKIZER, S., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. Non-scalable locks are dangerous. In *Linux Symposium* (2012).

[5] CHAUDHRY, S., CYPHER, R., EKMAN, M., KARLSSON, M., LANDIN, A., YIP, S., ZEFFER, H., AND TREMBLAY, M. Rock: A high-performance sparc cmt processor. *Micro, IEEE 29*, 2 (2009), 6–16.

[6] CHRISTIE, D., CHUNG, J.-W., DIESTELHORST, S., HOHMUTH, M., POHLACK, M., FETZER, C., NOWACK, M., RIEGEL, T., FELBER, P., MARLIER, P., ET AL. Evaluation of amd's advanced synchronization facility within a complete transactional memory stack. In *Proc. EuroSys* (2010), ACM, pp. 27–40.

[7] CHUNG, J., YEN, L., DIESTELHORST, S., POHLACK, M., HOHMUTH, M., CHRISTIE, D., AND GROSSMAN, D. Asf: Amd64 extension for lock-free data structures and transactional memory. In *Proc. MICRO* (2010), IEEE, pp. 39–50.

[8] DALESSANDRO, L., CAROUGE, F., WHITE, S., LEV, Y., MOIR, M., SCOTT, M. L., AND SPEAR, M. F. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. *ACM SIGARCH Computer Architecture News 39*, 1 (2011), 39–52.

[9] DICE, D., LEV, Y., LIU, Y., LUCHANGCO, V., AND MOIR, M. Using hardware transactional memory to correct and simplify and readers-writer lock algorithm. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming* (2013), pp. 261–270.

[10] DICE, D., LEV, Y., MARATHE, V. J., MOIR, M., NUSSBAUM, D., AND OLSZEWSKI, M. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proc. SPAA* (2010), ACM, pp. 325–334.

[11] DICE, D., LEV, Y., MOIR, M., NUSSBAUM, D., AND OLSZEWSKI, M. Early experience with a commercial hardware transactional memory implementation. In *Proc. ASPLOS* (2009).

[12] FOMITCHEV, M., AND RUPPERT, E. Lock-free linked lists and skip lists. In *Proc. PODC* (2004).

[13] HERLIHY, M., LEV, Y., LUCHANGCO, V., AND SHAVIT, N. A provably correct scalable concurrent skip list. In *Proc. Conference On Principles of Distributed Systems (OPODIS)* (2006).

[14] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proc. ISCA* (1993).

[15] MINH, C., CHUNG, J., KOZYRAKIS, C., AND OLUKOTUN, K. STAMP: Stanford transactional applications for multi-processing. In *Proc. IISWC* (2008).

[16] PUGH, W. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM 33*, 6 (1990), 668–676.

[17] WANG, A., GAUDET, M., WU, P., AMARAL, J. N., OHMACHT, M., BARTON, C., SILVERA, R., AND MICHAEL, M. Evaluation of blue gene/q hardware support for transactional memories. In *Proc. PACT* (2012).