Int J Parallel Prog (2015) 43:1192–1217
DOI 10.1007/s10766-014-0322-9

The Scalability of Disjoint Data Structures on a New Hardware Transactional Memory System

Gong Su · Stephen Heisig

Received: 23 December 2013 / Accepted: 22 August 2014 / Published online: 4 September 2014

Abstract In this paper we present our experiences constructing and testing in-memory data structures designed to be disjoint enough for transactional memory to be profitable as a serialization mechanism with no fallback to traditional locking. Our goal was to restrict memory conflicts to actual contention situations so that transactional memory techniques could be used as efficiently as possible. We describe the hardware transactional execution facility in the IBM zEnterprise EC12 server. We present an order preserving hashed structure that permits insertion, deletion, and traversal operations typically supported by a sorted linked list. We also present a concurrent open addressing hash table structure. We measure the performance and scalability for these data structures on the IBM zEnterprise EC12 server. Our results show near linear scalability of the insertion and deletion operations for up to 96 CPUs. We also discuss transaction abort frequency and hardware/software interactions.

Keywords Hardware transactional memory · Disjoint data structure · Linked list · Order preserving hashing · Spanning · Open addressing hash table · Probing sequence · Rehash

1 Introduction

The concept of transactional memory has existed since 1986 [1] and Herlihy and Moss coined the actual term Transactional Memory [2] in 1993. Transactional memory can be implemented either purely in software (STM) [3–5], or purely in hardware (HTM) [2,6–9], or a combination of software andhardware (Hybrid TM) [10–15]. Due to the

G. Su (✉) · S. Heisig
IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA
e-mail: gongsu@us.ibm.com

S. Heisig
e-mail: heisig@us.ibm.com

performance overhead [16], STM remains mainly of academic interest. HTM promises better performance but new architectures with HTM support have been slow to come to market. Hybrid TM uses HTM when it is available and switches to STM otherwise. While useful for testing transactional programs on architectures without HTM support, Hybrid TM cannot really be used in practice without HTM since it falls back to STM. It is therefore clear that HTM remains the key implementation for transactional memory.

Although it has taken some time, new architectures with HTM support are finally reaching the market. Sun (now Oracle) ROCK [17] processor is the first commercial general purpose architecture with HTM support although it was canceled in 2010. Azul Vega [18] (a Java accelerator) and IBM Blue Gene/Q [19] (a supercomputer) both have HTM support although they are considered special purpose architectures. Intel Haswell [20] introduced general purpose HTM support in 2013. Released in August 2012, the IBM zEnterprise EC12 server [21] becomes the first general purpose architecture including a Transactional Execution Facility that provides HTM support to reach the market.

Due to the lack of a commercial general purpose HTM architecture, limited work [18,22,23] has been done to evaluate whether HTM will live up to the promise of high concurrency and performance, particularly when scaled to a large number of CPUs. The purpose of this paper is to help fill this gap. While other internal pre-release efforts were involved in attempting to exploit more traditional transactional lock elision techniques, our approach was to try to design new disjoint data structures that would be more naturally suited to transactional execution. Simple lock elision without changing the existing data structure may not exploit the full potential of HTM since the data structures may not be free of false memory conflicts. The initial intent of our work was to find places to exploit HTM in the OS. We faced resistance from software engineers with ‘set size anxiety’. There are many linked list type structures that are normally quite short. However there is no guarantee in the code that under pathological circumstances these lists couldn’t grow without bound. Our first task was to figure out how to process an arbitrarily long linked list using HTM. The solution was to break the structure into pieces small enough to be guaranteed to fit inside a transaction. We ultimately tried to create new data structures as disjoint as possible in order to fully exploit HTM. We demonstrate that, with disjoint data structures, HTM can achieve impressive concurrency and performance, even when scaled to a large number of CPUs. However, there are some downsides: (1) it requires considerable effort to design a disjoint data structure; and (2) the best-effort hardware primitives create considerable complexity in programming HTM. Addressing (1) requires considering HTM constraints as an initial requirement. To address (2), either the hardware needs to provide guaranteed functions, or higher level deterministic software services, such as list insertion, deletion, and traversal, etc., must be provided to hide the lower level probabilistic best-effort primitives.

The key contributions of this paper include the following:

- We demonstrate on currently available hardware that with a disjoint data structure, HTM lives up to some of the promise of high concurrency, performance, and scalability to a large number of CPUs.

- We analyze abort types and frequencies to provide insight into the behavior of HTM, which may help devise better retry strategies and provide feedback to improve data structures and minimize aborts.
- We test and analyze the impact of different transaction sizes on performance.
- We share our experiences in developing code specifically for HTM on a general purpose architecture, useful techniques, and potential pitfalls.

The rest of the paper is structured as follows: Sect. 2 gives a brief description of the transactional memory hardware in the IBM zEnterprise EC12 server and its programming semantics; Sect. 3 describes a disjoint data structure for performing sorted linked list operations; Sect. 4 presents experimental results for this structure; Sect. 5 discusses the pluses and minuses of HTM based on our experience; Sect. 6 surveys related work; and Sect. 7 concludes with future work.

2 Description of HTM on System z

The Transactional Execution Facility in the IBM zEnterprise EC12 provides hardware primitives to specify critical sections of code that will execute *atomically*. Instructions in a transaction execute with both *linearizability* [24] and *opacity* [25]. Linearizability means that concurrent transactions appear to execute sequentially without overlapping in time. Opacity means that transactional instructions appear to have either all executed instantaneously or none has executed. Writes to memory locations by a transaction are isolated from other transactional and non-transactional instructions. This is known as *strong atomicity* and means that any changes to memory locations made in a transaction will not be visible to any other instructions unless the transaction successfully commits. This isolation makes things much easier for programmers since code will never operate on intermediate and thus potentially inconsistent results.

Conceptually there is a read set for memory locations that have been read from and a write set of memory locations that have been changed by a processor in a transaction. The read set is not a dedicated area but a set of tags in the L1 cache array. It may also overflow to include L2 cache lines. The write set is held in a new hardware structure called the Store Cache. It contains half lines and bits to indicate changed bytes with half lines. Figure 1 shows a cartoon of the read set and write set hardware. An in-depth discussion of HTM in the zEnterprise EC12 can be found in [28].

Two types of transactions are defined: constrained transactions (beginning with TBEGINC) and unconstrained transactions (beginning with TBEGIN). Both types are terminated with the TEND instruction. They both also have restrictions on which instructions can be executed. For example, privileged, control type, and cryptographic instructions are all prohibited. Making sure no restricted instructions appear within a transaction is a new dependency for humans and compilers.

In constrained transactions, there are further restrictions on the number (maximum of 32) and type of instructions (e.g., no backwards branches so loops are prohibited). There are also restrictions on how much memory they can reference.

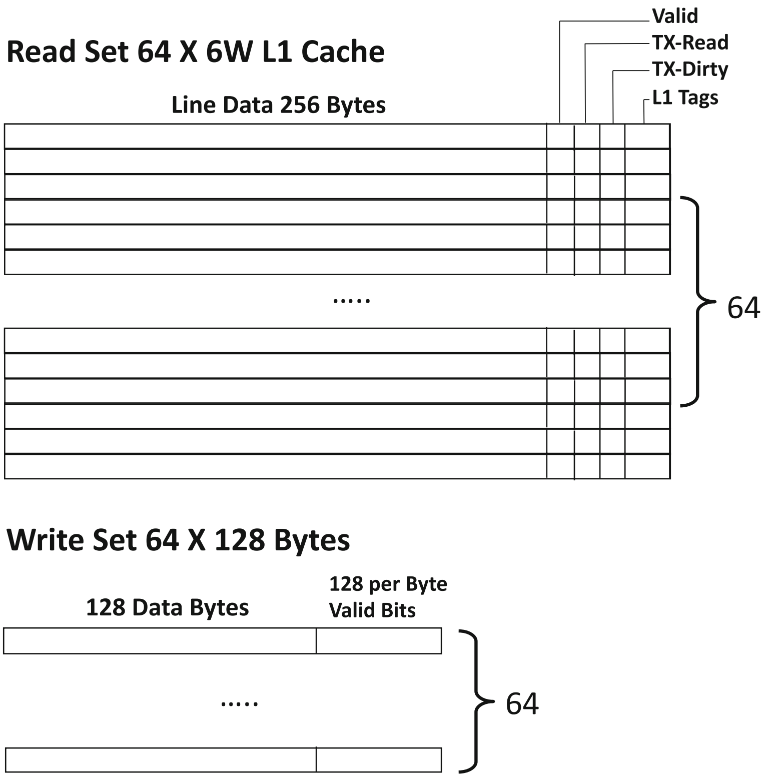

Fig. 1 Read and write set hardware

But if these restrictions are met, the hardware assures the transaction will eventually complete. Retries are handled by the hardware transparent to the software. Therefore, there is no need for a user supplied abort analysis routine. The hardware will take a series of increasingly drastic actions to improve the success of a transaction that has failed multiple times. These actions include inserting random wait times between retries, reducing speculative execution, and delaying conflicting work on other processors. Example 1 shows a sample constrained transaction.

```
XferAmt: TBEGINC
           Donor = Donor - Amt;
           Receiver = Receiver + Amt;
         TEND
```

Example 1: Constrained transaction example

Unconstrained transactions are executed on a best effort basis and so have a more visibly probabilistic nature. They can be thought of as a single large instruction that may fail for various reasons and need to be retried. The TBEGIN instruction has operands for the address of a Transaction Diagnostic Block (TDB), and a directive to

Table 1 Abort reason codes

Abort code	Abort reason	Condition code
2	External interruption	2
4	Program interruption (unfiltered)	2 or 3
5	Machine-check interruption	2
6	I/O interruption	2
7	Fetch overflow	2 or 3
8	Store overflow	2 or 3
9	Fetch conflict	2
10	Store conflict	2
11	Restricted instruction	3
12	Program-interruption condition	3
13	Nesting depth exceeded	3
14	Cache fetch-related	2 or 3
15	Cache store-related	2 or 3
16	Cache other	2 or 3
255	Miscellaneous condition	2 or 3
256	TABORT instruction	2 or 3

save registers. If registers are not saved, the TBEGIN only takes a few cycles. The TDB contains potentially useful information in the case of aborts. It has an abort reason code, and depending on the abort, a conflict address, and an instruction address. See Table 1 for a complete list of abort codes and reasons.

An unconstrained transaction code example is shown in Example 2. An abort in a transaction transfers control to the instruction immediately following the TBEGIN. This instruction tests the condition code (CC) to determine whether this is an abort flow or normal transaction begin. The CC is set to 0 for normal begin, 1 for an indeterminate condition for which the TDB could not be updated, 2 for a transient condition and 3 for a persistent condition. In the case of an abort, we check the retry limit, perform abort analysis by examining the CC and TDB (if it was stored), potentially take some action to increase the success of the next transaction, then go back to retry.

In the case where the registers were saved by TBEGIN they are restored on abort and of course all transaction storage changes will have vanished. This does make it difficult to tell exactly what went wrong if there is a data dependent bug in the transaction. A non-transactional store instruction (NTSTG) can store data from within a transaction without updating the write set. This can be very useful in tracing data addresses in situations where the address causing an abort is unknown or doesn't seem to make sense. There is also a TABORT instruction that can be issued from inside a transaction to immediately abort. In the example, after an abort control passes to the Aborts label where the total number of consecutive aborts is incremented. If the retry limit has not been equaled an abort analysis routine is called and control passes back to the TBEGIN.

```
           Retry = 0;
           Limit = 100;
  XferAmt: TBEGIN Addr(MyTDB) Regs(F)
           IF CC>0 THEN GOTO Aborts;
             Donor = Donor - Amt;
             Receiver = Receiver + Amt;
           TEND
           Compare-And-Swap(NumXfers,...);
           GOTO Success;
   Aborts: Retry = Retry + 1;
           IF Retry=Limit THEN GOTO GiveUp;
           CALL AnalyzeAbort;
           GOTO XferAmt;
  Success: /* Both fields updated    */
   GiveUp: /* Neither field updated */
```

Example 2: Unconstrained transaction example

Missing from the code example above is any footprinting or recording of what resources are held. Unlike code that gets a lock, if the transaction aborts or the program blows up there is no need to worry about cleaning up serialization resources in recovery processing. This is a significant savings in complexity in both main path and recovery code.

Since we were interested in understanding the boundary behavior of large data structures we discuss unconstrained transactions for the rest of the paper.

3 Disjoint Data Structures

Speculative Lock Elision [26] is a technique to allow concurrent access to shared data structures by optimistically speculating that multiple threads will not conflict, hence there is no need to acquire a lock. Conflicts are detected and recovered from by aborting and rolling back the involved thread(s). In fact, one of the often suggested usages of HTM is lock elision, i.e., replacing lock/unlock with TBEGIN/TEND to allow higher degree of concurrency.

The advantage of using HTM for lock elision is that it is easy to implement. One simply needs to replace lock/unlock with TBEGIN/TEND without changing the existing data structure. The disadvantage, however, is that the existing data structure may not be disjoint enough for the multiple threads to access the shared data structure concurrently. For example deleting the largest/smallest element from even a large heap structure requires accessing a single memory location and so guarantees conflict. In order to take maximum advantage of HTM, existing data structures need to be redesigned and/or new data structures need to be invented to minimize memory conflicts. In this section, we describe how to make a traditional sorted linked list as disjoint as possible by *spanning* and *hashing*. We also demonstrate how an open addressing hash table can be made concurrent with HTM very easily.

3.1 Linked List

A sorted linked list is a commonly used data structure where nodes with values (without loss of generality we assume the values are integers) are maintained in a sorted list and accessed via a list head. The list typically supports 3 operations:

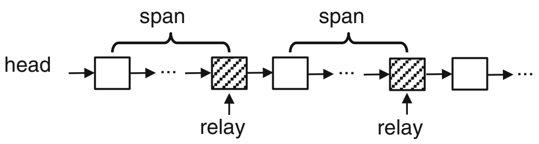

Fig. 2 Spanning with relay nodes

- *insert*: insert a node with a value into the list
- *delete*: delete a node with a value from the list
- *traverse*: enumerate all the node values in the list

Note that both *insert* and *delete* need to traverse at least part of the list in order to find the insertion point (node in front of which the new node is to be inserted) and the deletion point (the node to be deleted), respectively.

We immediately faced two problems when trying to use HTM for concurrent access to a sorted linked list. First, the list can be arbitrarily long. HTM has a limit on the size of a transaction due to the finite capacity in the read and write sets. Therefore, if a list is too long to fit in a single transaction, we cannot completely traverse it in a single transaction. Second, even if the list is short enough to be traversed in a single transaction, all nodes between the list head and the insertion/deletion point will be in the read set of a transaction. Another thread with an insertion/deletion point closer to the list head will change a node that is in the read set of the first thread with insertion/deletion point further from the list head. That is, a transaction which changes a node can abort all other transactions which are deeper in the list. Essentially, a sorted linked list and its operations are *not* disjoint.

3.1.1 Spanning

A natural solution to solve these two problems is to break up the traversal of the list into multiple transactions, each traversing a small number of nodes. For example, we can traverse node 1–32 in a transaction, then traverse node 33–64 in a second transaction, etc., as shown Fig. 2. We call this technique *spanning* and we call the number of nodes processed in one transaction a *span*. A similar technique called *telescoping* was mentioned in [27] but few details were given.

Spanning is a fairly simple and straightforward idea but certain details deserve more attention. Care must be taken with the *relay* nodes (shaded nodes) as indicated in Fig. 2. A relay node is the last node of a span at which we end a transaction during our traversal. The tricky problem is that once a thread ends a span and before it starts the next span, the relay node may be deleted by another thread, making it impossible for the first thread to continue its traversal.

To prevent a relay node from being deleted, thread *A* increments a *stop count* in the relay node just before it ends the transaction. When thread *B* tries to delete a node, it first checks if the stop count of the node is 0. If yes, the node can be immediately deleted as usual. Otherwise, the node can be removed from the list but cannot be freed since one or more threads have stopped at this relay node and are in

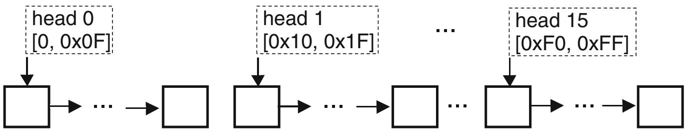

Fig. 3 Order preserving hashing with linear keys

between transactions. Instead, a *dead bit* in the relay node will be set by thread B. Now when thread A starts a new span to continue its traversal, it first decrements the stop count (which is guaranteed to be nonzero) of the relay node. If the count becomes 0, thread A needs to check if the dead bit is set. If yes, the relay node has been deleted by another thread and there is no other thread stopped at this relay node (since the stop count is 0). If so, thread A must now free the relay node and then continue its traversal.

3.1.2 Hashing

While spanning allows traversal of an arbitrarily long list and makes the insertion, deletion, and traversal operations more disjoint, it still suffers from another problem: all threads must start their traversal from the single list head, potentially causing more conflicts and aborts at the beginning of the traversal.

We solve this problem with *order preserving hashing*. A hash function h is order preserving if for any two keys $k_1 \leq k_2$, it follows that $h(k_1) \leq h(k_2)$. As an example, assuming our key space is integers uniformly distributed from 0 to 255 ($0x$FF) and the hash function is $h(k) = (k\& \, 0x\text{F0}) \gg 4$ (taking the highest 4 bits of k). It is not hard to see that h is an order preserving hash function where hash value 0 holds keys 0 to 0x0F, hash value 1 holds keys 0x10 to 0x1F, etc. As another example, hash function $h(k) = \lfloor log_2 k \rfloor$ is also an order preserving hash function.

With order preserving hashing, we can start traversing a sorted linked list from multiple heads computed by the hash function rather than a single list head. Using the same example as above, assuming the node values are from 0 to 255 and with the hash function $h(k) = (k\& \, 0x\text{F0}) \gg 4$, we can start traversing the list from 1 of 16 heads given a node value, as shown in Fig. 3.

Essentially, we break up the list into 16 sub-lists, each having its own head and value range. In the diagram above, node values in the range [0, $0x$0F] hash to 0 and therefore we start traversing from head 0; node values in the range [$0x$10, $0x$1F] hash to 1 and therefore we start traversing from head 1, etc. This allows us to start traversing from the middle of the list rather than always from the beginning. Note that due to the order preserving property, we can still perform a full list traversal to enlist all the nodes in the list by traversing each sub-list in order.

Although our example above assumed a uniform distribution of integer keys, other key distributions can be handled with a suitable order preserving hash function. For example, if the keys are exponentially distributed, a logarithmic hash function such as $h(k) = \lfloor log_2 k \rfloor$ (which gives uniformly distributed hash values) can be used. Figure 4 shows an example of applying this logarithmic hash function to 32-bit integer keys, breaking a list into 32 sub-lists.

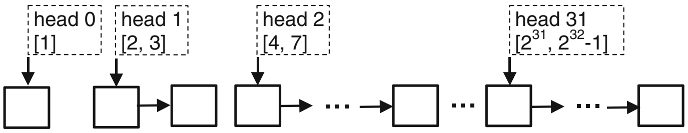

Fig. 4 Order preserving hashing with logarithm keys

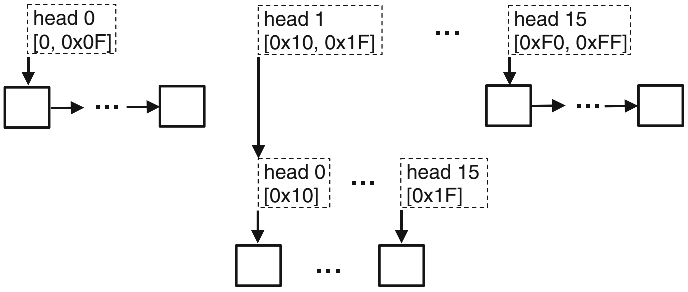

Fig. 5 Adaptive recursive order preserving hashing with linear keys

3.1.3 Adaptive Recursive Hashing

Combining spanning and order preserved hashing, we make a *long* sorted linked list very disjoint. We demonstrate this by presenting results in Sect. 4 to show near linear scalability for insertion and deletion with up to 96 CPUs. With a *short* list, however, our tests still showed bad scalability. This is because with a short list, each sub-list may have fewer nodes than a span's worth, essentially rendering spanning useless.

To resolve this problem, we apply the order preserving hashing recursively to a sub-list to further break it down such that each node gets its own head, i.e., the sub-lists of a sub-list have just one node. We illustrate the idea in the diagram below using the same example as that in Sect. 3.2.

As illustrated in Fig. 5, the sub-list pointed to by head 1, which holds values in the range $[0x10, 0x1F]$, has been further broken into 16 sub-lists, each has just one node pointed to by its own head. Head $1[0x10,0x1F]$ now points to an array of (2nd level) heads 0–15 instead of a list of nodes. In this particular example, the hash function to compute the 2nd level heads is to take the next highest 4 bits after the highest 4 bits taken by the 1st level heads, which is $h(k) = k \ \& \ 0x0F$.

Since creating more levels of list heads incurs higher memory overhead, by default we only create the 1st level list heads. We monitor conflicts in the sub-lists and dynamically expand and contract additional levels of list heads for "hot" and "cold" sub-lists, respectively. With this adaptive recursive hashing technique, we were able to achieve near linear scalability for short list insertion and deletion with up to 96 CPUs as well. We present the results in Sect. 4.

3.2 Open Addressing Hash Table

Hash tables [36] are widely used data structures in computer science. Two common variants of hash tables are separate chaining and open addressing. They differ in how

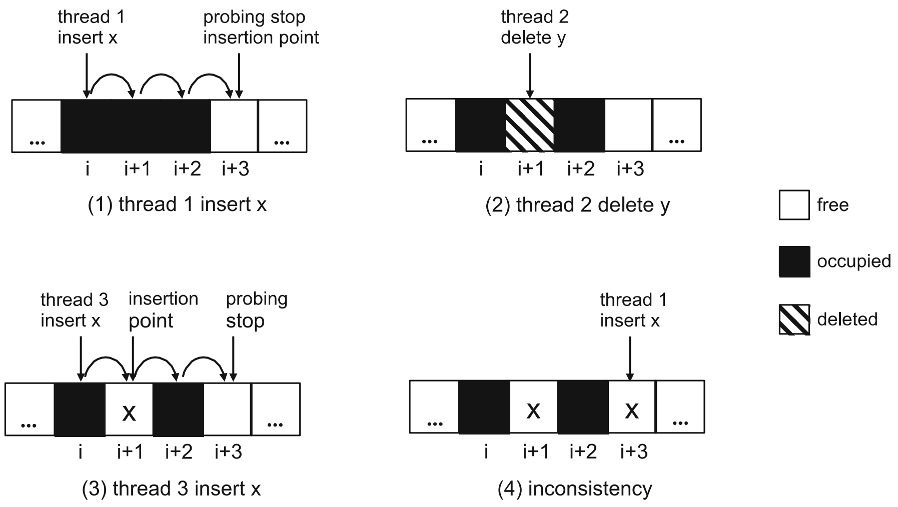

Fig. 6 Race condition in concurrent open addressing hash table

collisions are resolved. To store two different keys that hash to the same slot, separate chaining uses a secondary data structure outside the hash table itself, such as a linked list, while open addressing probes to find alternative slots in the hash table. Each variant has its pros and cons and both are widely used.

Many existing concurrent hash table implementations, such as those supplied with Java JVM, are separate chaining based. The only concurrent hash table implementations based on open addressing we could find [30,31] use a non-blocking algorithm. They are complex and tricky to implement, and neither implementation demonstrated scalability with more than 16 threads.

We show that HTM is a perfect match to make open addressing concurrent. An open addressing hash table is already a fairly disjoint data structure. A good hash function such as xxhash [32] or MurmurHash [33] is expected to distribute the hash keys uniformly across the entire hash table. With HTM, making an open addressing hash table concurrent is as simple as surrounding the single thread algorithm with TBEGIN and TEND.

We present an example to illustrate a race condition for concurrent open addressing hash table. In this case the lock-based approach is very difficult to make it concurrent but with HTM it is almost trivial to do so. Note that there are different probing strategies for open addressing such as linear probing, quadratic probing, and double hashing, etc. For the purpose of our discussion, they do not matter so we will use linear probing with a step size 1 to illustrate the issue. We also assume that readers are familiar with the basic open addressing hash table algorithmcer [34–36]. We also use the simpler hash set (key only) rather than hash map (key-value pair) for our example although the discussion will be exactly the same for both.

As illustrated in Fig. 6, events occur in the following order:

(1) Thread 1 inserts key x, which hashes to slot i. Since slot i is occupied, it probes slot $i + 1$, $i + 2$, which are also both occupied, until slot $i + 3$ which is free. Now

thread 1 concludes that key x is not in the hash table and is ready to insert key x at slot $i + 3$. However, before it gets a chance to do so, events (2) and (3) occur.

(2) Thread 2 deletes key y, which hashed to slot $i + 1$. Now slot $i + 1$ is marked as deleted.

(3) Thread 3 also inserts key x, and will go through the same probing sequence as thread 1 did (the deleted slot $i+1$ does not stop probing). When thread 3 encounters the free slot at $i + 3$, it also concludes that key x is not in the hash table and inserts key x into the hash table. The algorithm is such that it would insert key x at the first deleted slot during probing, which is slot $i + 1$. Therefore, thread 3 inserts key x at slot $i + 1$.

(4) Finally, thread 1 finishes inserting key x at slot $i + 3$. This causes an inconsistency since key x is now inserted twice, which breaks the fundamental hash table association property of a key being unique.

From this example, we can see that it is not enough to serialize only the update of the insertion or deletion slot at the end of the probing sequence. The entire set of probing sequence slots must be serialized. Without protecting the entire probing sequence, two threads attempting to insert the same key can observe different slot states when going through the same probing sequence. As illustrated by the example, thread 1 did not see a deleted slot during its probing while thread 3 did, due to the deletion of a key in the probing sequence by thread 2 “behind thread 1’s back”.

To protect the entire probing sequence with a lock based approach, one would have to associate a lock with each and every slot, and lock the entire set of probing sequence slots. While conceptually possible, it is very cumbersome and incurs additional memory overhead. It would also limit concurrency since if two threads’ probing sequences touched any common slot, one thread would have to wait. On the other hand, this situation is exactly what HTM is designed for: to serialize an arbitrary collection of memory locations. Note that probing consists of a series of reads followed by a write. So only the last memory location of the probing sequence will be in the write set of HTM, while all the others will be in the read set. Therefore, as long as the slot written to by one thread does not intersect the probing sequence of another thread, both threads will succeed without abort. This provides higher concurrency than a lock based solution since only writes (as opposed to reads) to objects in the read or write set cause a retry.

With HTM, it was pretty easy to make an open addressing hash table concurrent. By simply surrounding the single thread algorithm with TBEGIN and TEND, along with abort and retry handling all contention cases were covered. We have implemented such a concurrent open addressing hash table with HTM and we present its scalability results in the next section.

Our focus was on using HTM to speed up hash table insert and delete operations and so we did not test the case where the hash table needs to grow in size by rehashing. However, we did think about how rehashing can be done in a concurrent environment without having to resort to a global lock. Since this technique uses only inserts and deletes which can be serialized with HTM it doesn’t require additional invention. We briefly describe our idea and propose it as future work. The basic idea is that during rehash operations will act on both the old and the new hash tables. The basic algorithm looks like this:

- A flag is set indicating rehashing is in progress. A rehashing thread is now going through the old hash table and migrating the keys to the new hash table. The rehashing thread will remove a key from the old hash table only after it has inserted it into the new table. Therefore, there is a moment a key appears in both the old and the new hash tables. But there is never a moment a key appears in neither.
- For the insert operation, first check if the key is in the old table. If yes, it is a duplicate. If no, insert it into the new hash table (can be a duplicate in the new hash table). Basically, once the flag is set, new incoming keys will only be inserted into the new table.
- For the delete operation, first attempt to delete in the old hash table. Regardless of whether the key is found and deleted from the old hash table, attempt again to delete from the new hash table. We must check both the old and the new hash tables because the key in the old hash table may just have been migrated to the new hash table, and it might not have been removed from the old hash table yet.
- For the lookup operation, first search the old hash table, if not found, search the new hash table.
- Once the rehashing thread finishes migrating all the keys, it resets the flag and all operations now go directly to the new hash table.

4 Experimental Results

We tested our disjoint sorted linked list and open addressing hash table on an IBM zEnterprise EC12 server with a total of 99 CPUs. The results show various aspects of the interaction of the data structure with the hardware. We discuss abort analysis, varying span size, scalability of long lists, short lists, and open addressing hash tables.

4.1 Abort Analysis

Aborts fundamentally impact the performance of transactional memory. It is imperative to thoroughly understand the abort mechanism and behavior of HTM on a particular architecture to minimize aborts and maximize performance. Even without any explicit shared data contention, transactions can abort for a variety of reasons. For example, a cache line holding transactional data can be evicted by the cache LRU algorithm; or a thread can be interrupted by I/O, scheduling event, etc. Therefore, we begin by examining the behavior of single threaded transactional code with no contention.

4.1.1 Cache Size Limitation

Figure 7 shows the results of unspanned traversals of a linked list traversal program that attempts to traverse a list 100 times to increasing depth inside an unconstrained transaction. This program writes into each list element which is 24 bytes long. Since

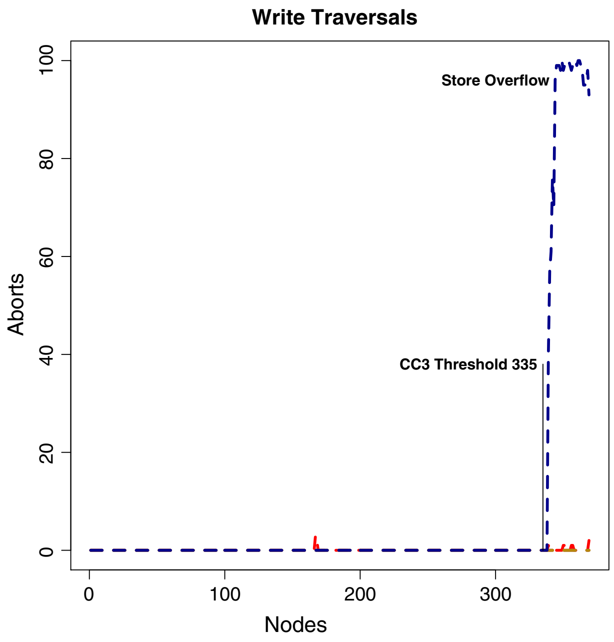

Fig. 7 Aborts for unspanned write traversals

we make 100 traversal attempts at each length we can take the number of aborts as an estimator for the probability that a transaction will fail for each length. It is clear that the abort rate is very low until the list gets close to 340 nodes, at which point it becomes 100 % likely that the transaction will abort with a store overflow abort. We saw previously the write set hardware holds 64 128-byte half lines, so it will hold 341 nodes in those 8,192 bytes. In this isolated scenario we can see a very abrupt boundary between the size of a structure that can be updated in a transaction and a slightly larger one that is 100 % certain to abort. In this case the point where the hardware returns CC3 (persistent condition) on abort is actually very close to the point where all transactions fail.

In Fig. 8 the traversal program only reads the content of each list element. We saw the L1 cache has 64*6 256-byte lines or 96K total capacity so we would expect it to hold roughly 4096 24-byte nodes. The figure shows that the abort probability for store conflicts starts to rise at this point. As TX-dirty lines are LRUed out of L1 they can overflow into the L2 cache and get tracked by the LRU extension mechanism. The hardware starts returning CC3 when lines start overflowing which seems to be pretty conservative since we can traverse almost four times as many nodes at an abort rate of only about 20 %. The private L2 cache is 1 megabyte in size so would hold over forty two thousand list elements if used completely. We can see a sharp jump in fetch overflow aborts at around twenty seven thousand elements. This is due to increasing L2 associativity conflicts where the transaction code is LRUing out its own lines in this situation. As with the write example there is an abrupt jump between an acceptable abort probability and an unacceptably high one dictated by

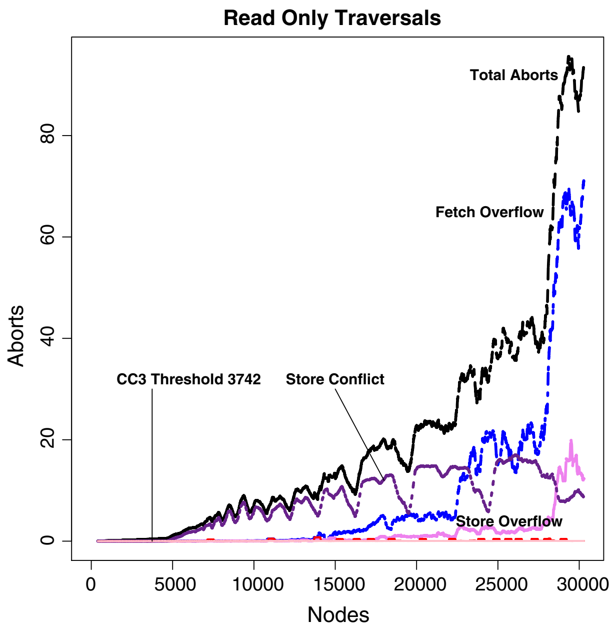

Fig. 8 Aborts for unspanned read only traversals

micro-architectural parameters. One of the objectives of spanned transactions is to eliminate programmer anxiety about how close to this boundary the size of a structure operation might get.

4.1.2 Cache Hierarchy Implication

It should be clear that transaction success can be heavily dependent on the behavior of other threads touching memory that is not in the transaction. In Fig. 9 the same program is run while 3 other copies with their own lists run on other processors. The effect should be to put more pressure on the cache hierarchy without causing any explicit conflicts. Here the persistent condition threshold returned by the hardware was reduced by about one sixth and once lines start overflowing to L2, fetch overflow aborts rise rapidly. The overall 100 % abort threshold occurred abruptly at about 4,000 elements rather than 30,000 with no other work. Since the cache hierarchy in current System Z machines is inclusive, one processor can evict a line out of a higher level shared cache congruence class and cause it to get evicted from all the lower level (including private) caches. When this happens to a line currently in the read or write set of a transaction it will abort. The zEnterprise EC12 has 6 cores per chip, 6 chips per multi-chip module, and 4 multi-chip modules (MCMs). This means there is the potential to see interference from many other processors.

In Fig. 10 the flow of one thread touching unrelated lines causing LRU cross-invalidates (LRU-XIs) is illustrated. There could also be multiple operating system (OS) images running virtualized on an MCM. It is possible a transaction could be

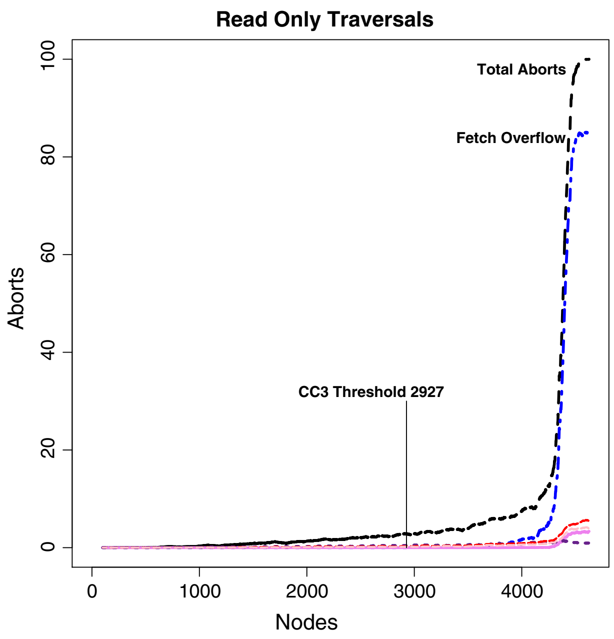

Fig. 9 Aborts for read only traversals with interference

aborted due to interactions with code running in another OS image. It is currently difficult to disambiguate true and false storage conflicts which can result in the same abort reason codes. It is therefore problematic to programmatically discern legitimate conflict in a data structure from the actions of other processors not explicitly touching the same memory locations.

4.1.3 Various Abort Types

The reasons for aborts may need to be taken into account before retrying. For example, in the case of an abort due to an External Interrupt (time slice) it would be a waste of time to invoke back off logic since the code is now at the beginning of a time slice and hasn't seen a conflict. We wrote abort analysis routines to both capture the frequency of aborts and compare the effects of different responses. Figure 11 shows the frequency of various aborts for one of two threads concurrently doing spanned traversals of the same long queue. In the previous figures the code was disabled for interrupts so we could focus on memory effects. In this test the code was enabled so we can see the effects of interrupts. Interrupts of all kinds will abort a transaction. These events were previously (before transactional memory) transparent to software which was unaware it was virtualized but now become visible and a possible constraint to the length of a transaction.

Time slice interrupts effectively place an upper bound on the number of instructions a transaction can execute. For example, a minor time slice in z/OS is 50 ms and a processor can execute roughly 5,000 instructions per microsecond. Therefore the

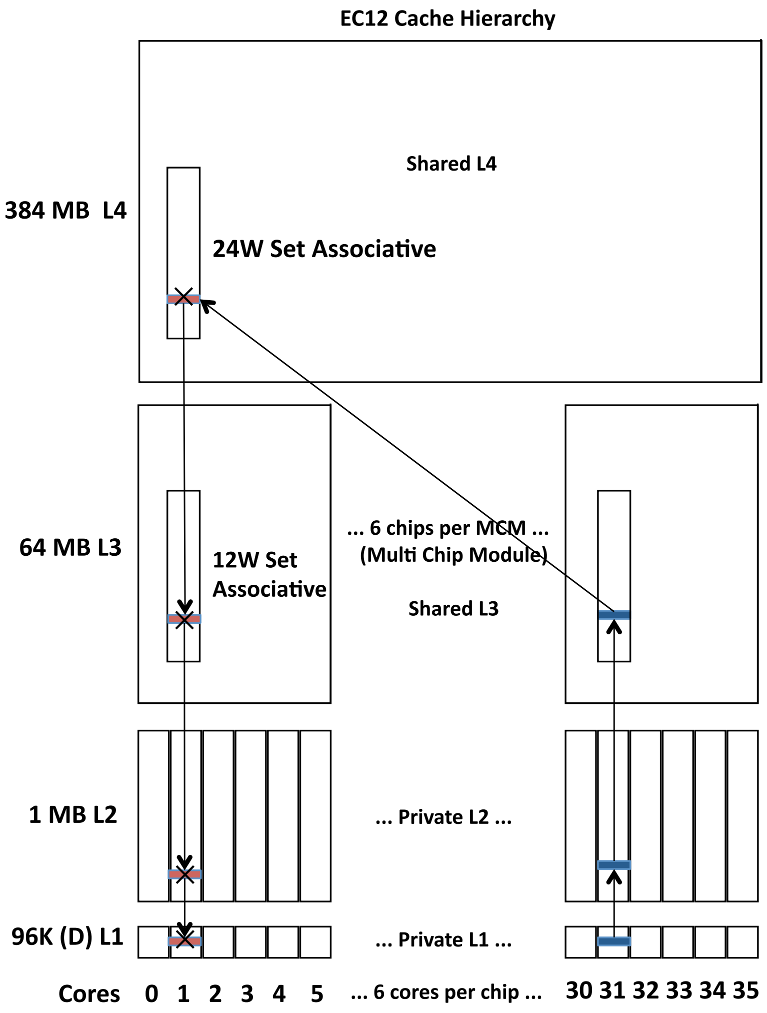

Fig. 10 Cache hierarchy and LRU XI flow

upper bound on the number of instructions in a transaction before a time slice abort is 250K. If a transaction starts at a random point in a time slice you would expect on average to have half a time slice remaining so you would only expect to be able to run 125K instructions before a time slice abort. Programmers need to be aware that transactions have both data size limits and instruction size limits.

There are also hardware millicode interrupts (External Alarm in Fig. 11) which occur when hardware functions have housekeeping to do. These can be more frequent than real conflicts in a system with disjoint structures. External Alarm and External Interrupt (time slice) are the two most frequent abort types and increase linearly with the amount of time spent in a transaction. Conflicts of other types both real and false were much less frequent in this test.

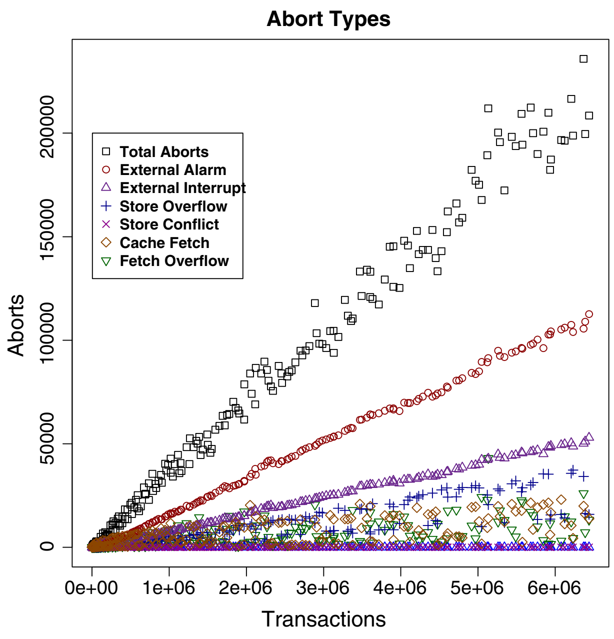

Fig. 11 Abort types

While the architecture strictly defines the syntax of the actual instructions there are clearly ill-defined, machine specific aspects to the facility that need to be understood by the programmer. The size and congruence class arrangement of the read and write sets, number of cores per chip and chips per multi chip module as well as the activity of instructions running on other cores all affect the probability of success of a transaction. Cache hierarchy parameters are machine dependent so code will behave differently from generation to generation.

Other seemingly mysterious effects are the result of hardware prefetch, branch prediction, and the length of the out of order processor pipeline. The hardware prefetch engine may recognize a stride based on instruction activity and bring in lines that have not been referenced by a specific instruction yet. This speculative prefetching can cause aborts in cases where a line is prefetched in exclusive mode and results in a demote cross invalidate to the previous owner. If the line is in the write set of the old owner its transaction will abort. Branch prediction logic can also cause operand fetch for an instruction that never actually gets executed (due to a branch guessed wrong). In complicated data structures this can be common.

Finally, due to the length of the pipeline, an instruction just beyond a TEND such as the Compare-And-Swap in example 2 can go through operand fetch for a location that will now be included in the transaction before the TEND has completed, possibly resulting in an abort. In this case the idea was to move the update of a global performance counter for the number of transfers just outside the transaction and serialize it using compare and swap. Global performance counters are a common code pattern that can increase the abort rate for a critical section. If speculative instructions in the pipeline are suspected, one way to test this is to add an instruction which collapses the pipeline (such as Purge ALB on Z) immediately after the TEND. Although this situation can be detected this way it imposes a new dependency that humans and compilers are not used to handling. Even though programmers may think they know exactly which instructions they are executing and which data locations are being referenced there is previously transparent speculative processing that now has visible consequences.

We found it invaluable during the development process to instrument our code. The abort routine maintained a frequency table of abort addresses, conflict addresses, and abort reason codes. It was easy to see hot spots in data structures and iteratively fix them with this data. During this hotspot analysis it should be noted that it was often necessary to imagine how two or more transactions were interacting. This complexity contradicts the idea that isolated, atomic code sections are easier to reason about than sections linearized by a lock. We also made use of non-transactional stores to create a trace table during debugging to quantify these interactions. This infrastructure can be shared by system services that wrap HTM exploiting code to provide higher level deterministic primitives to OS functions needing serialization. One last concern about abort analysis is that logging data and making retry decisions can introduce new effects which didn't exist previously.

4.2 Varying Span Size

An important parameter in our data structure is the number of nodes in a span. Intuitively, a smaller span size should incur more transaction begin and end costs and a larger span size incur higher abort and retry costs. Therefore one expects a certain middle span size to provide the best tradeoff. We measured the throughput, total aborts, and abort ratio of our data structure while varying the span size from 1 to 128. The list had a total of 1 million nodes broken into 1 thousand sub-lists. Each sub-list had 1 thousand nodes. The results shown in Figs. 12 and 13 are for one, two, and four CPUs. Results for more CPUs are similar and therefore are omitted. The figures match our intuition and provide the basis of our argument against an adaptive span size suggested by [27].

Figure 12 shows throughput per thread and total aborts of all threads versus span size; and Fig. 13 shows the throughput per thread and abort ratio of all threads versus span size. The figures show that throughput reaches its peak somewhere between span size 32 and 64, this region having the fewest number of total aborts. The abort ratio, on the other hand, increases as the span size increases since the total number of transactions decreases with larger span size. We use some simplified analysis to explain the U-shaped curve for total aborts in Fig. 12 which provides insight for the choice of span size.

The first observation is that the number of total aborts is proportional to the total length of transactional execution time. We start with an estimation of the total length of transactional time. We make the following assumptions:

- TBEGIN and TEND add a fixed amount of transactional time t_b and t_e, respectively
- Each node takes a fixed amount of transactional time t_p to process

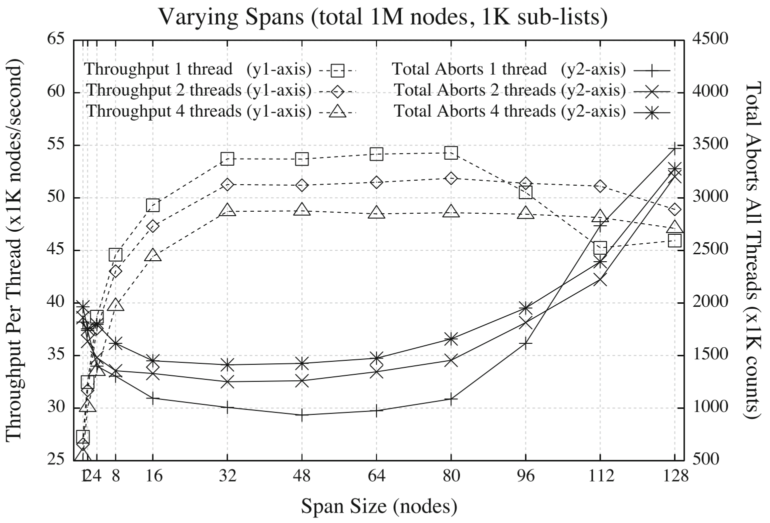

Fig. 12 Throughput and total aborts with varying spans

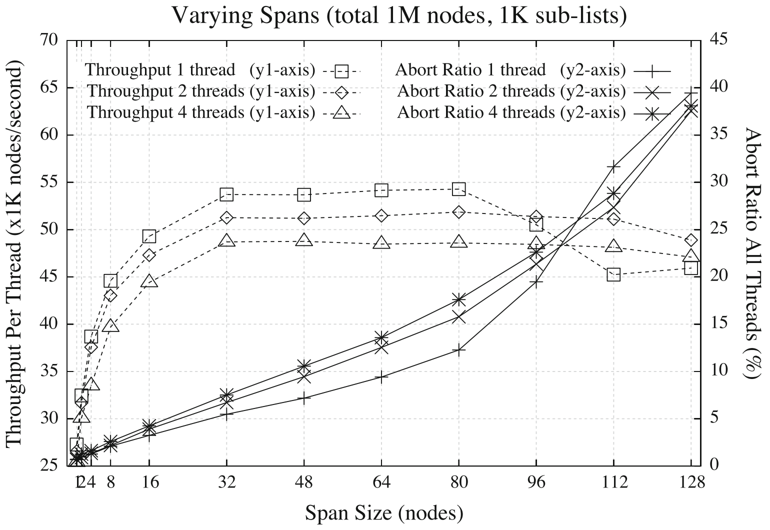

Fig. 13 Throughput and abort ratio with varying spans

- Total number of nodes is N and span size is n

The transactional time for processing one span is $t_\text{b} + t_\text{e} + nt_\text{p}$ and there are N/n spans. Since all spans are eventually processed successfully, the total amount of successful transactional time spent on processing the spans is $(t_\text{b} + t_\text{e} + nt_\text{p}) \times N/n$. However, a certain percentage of the spans aborted before succeeding. We assume this percentage is $\alpha(n)$. For each abort, we assume that on average we waste half of the span processing, i.e., we waste an amount of $t_\text{b} + nt_\text{p}/2$ transactional time for each abort. And we assume that the average number of aborts for a span is some function of its size, call

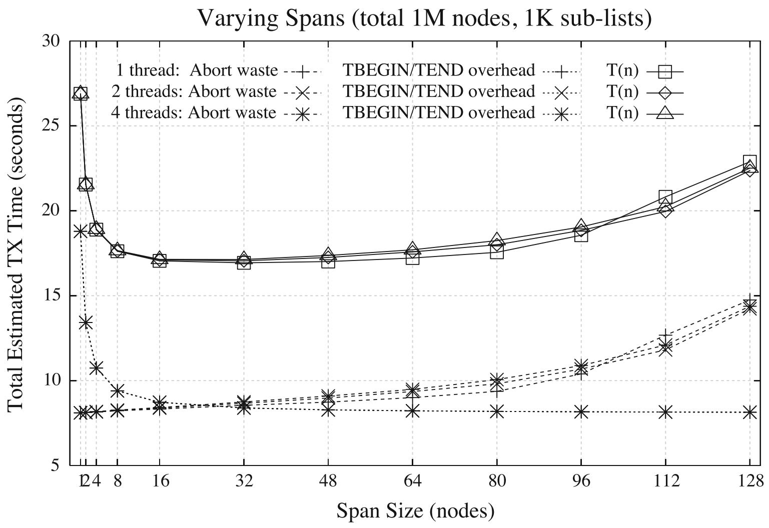

Fig. 14 Total estimated transactional execution time

this $\beta(n)$. Therefore, the total amount of wasted transactional time due to aborts is $\beta(n) \times (t_\mathrm{b} + nt_\mathrm{p}/2) \times \alpha(n)\, N/n$. Adding the two terms, we arrive at our estimation of the total transactional time T as a function of the span size n as:

$$\begin{aligned} T\,(n) &= [\left(t_\mathrm{b} + t_\mathrm{e} + nt_\mathrm{p}\right) \times N/n] + [\beta(n) \times \left(t_\mathrm{b} + nt_\mathrm{p}/2\right) \\ &\times \alpha(n)\, N/n] = N\alpha\,(n)\,\beta\,(n)\left(t_\mathrm{b} + nt_\mathrm{p}/2\right) \\ &/n + N\,(t_\mathrm{b} + t_\mathrm{e})\,/n + Nt_\mathrm{p} \end{aligned}$$

We can further simplify this function by observing that $\beta(n) \times \alpha(n)N/n = A(n)$, where $A(n)$ is the total number of aborts. Therefore, $N\alpha(n)\beta(n) = nA(n)$ and we have:

$$T(n) = \underbrace{A(n)(t_\mathrm{b} + nt_\mathrm{p}/2)}_{\text{Abort waste}} \quad \underbrace{+N(t_\mathrm{b} + t_\mathrm{e})/n + Nt_\mathrm{p}}_{\text{TBEGIN/TEND overhead}}$$

Intuitively, the first term is the wasted time due to aborts. As span size n increases, we expect this term to increase. While we do not know the analytical form of $A(n)$, we can use our test results for $A(n)$ to plot the first term and verify. The second term is the fixed overhead due to TBEGIN/TEND. Clearly, this is a decreasing function of span size n since the overhead as a percentage of the transaction declines with larger spans. The sum of the two terms therefore should give a U-shaped curve of $T(n)$ similar to that of $A(n)$ we observed in Fig. 11. We plot the first term, the second term, and $T(n)$ itself in Fig. 14. For the plot, we used 50 cycles for t_b and t_e, and 75 cycles for t_p.

From our test results and analysis, we believe that an architecture will likely have a fairly narrow range of span sizes that could be practical. An adaptive span size mechanism is unlikely to be very helpful since an increase in aborts can happen when

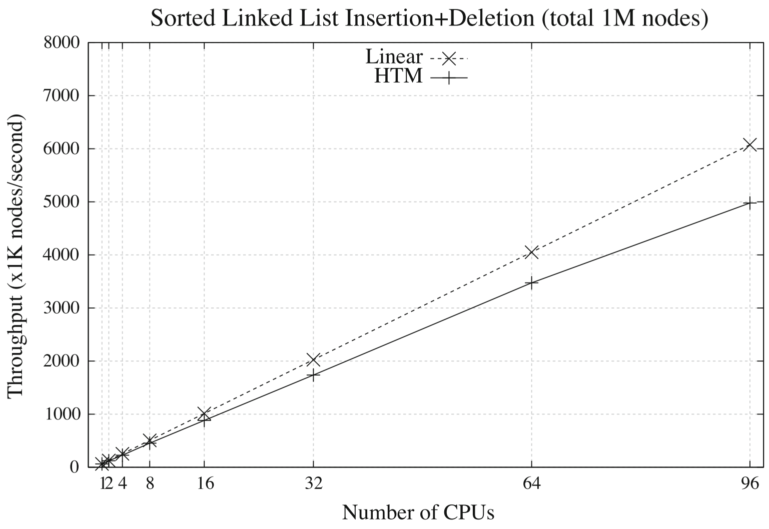

Fig. 15 Long sorted linked list scalability

the span size gets close to either boundary. This further suggests that, unless contention is very low, large or unlimited size transactions are of questionable value. The cost of aborts and retries would be so high even a low abort rate would result in a large amount of wasted rework.

4.3 Long and Short Lists

Below are scalability results for insertion and deletion operations for the disjoint sorted linked list. For the long list, there are a total of 1 million nodes broken into 1 thousand sub-lists. Each sub-list has 1 thousand nodes. For the short list, there are a total of 192 nodes broken into 192 sub-lists. So each sub-list has 1 node. For the long list, we use a span size of 32. For different number of CPUs, each thread is bound to its own CPU and gets N/m nodes (where N is the total number of nodes and m is the total number of threads). Each thread repeatedly inserts and deletes its N/m nodes into and out of the list. The results are shown in Figs. 15 and 16.

We can see that with spanning and order preserving hashing, the throughput of inserting and deleting nodes into and out of a long sorted linked list scales near linearly for up to 96 CPUs. With adaptive recursive hashing, the throughput of inserting and deleting nodes into and out of a short sorted linked list also scales near linearly for up to 96 CPUs. These results show that disjoint data structures can indeed achieve high concurrency, performance, and scalability to a large number of CPUs.

4.4 Open Addressing Hash Table

Figure 17 shows the scalability results for insertion and deletion operations against the open addressing hash table serialized with HTM. A total of 1 million uniformly dis-

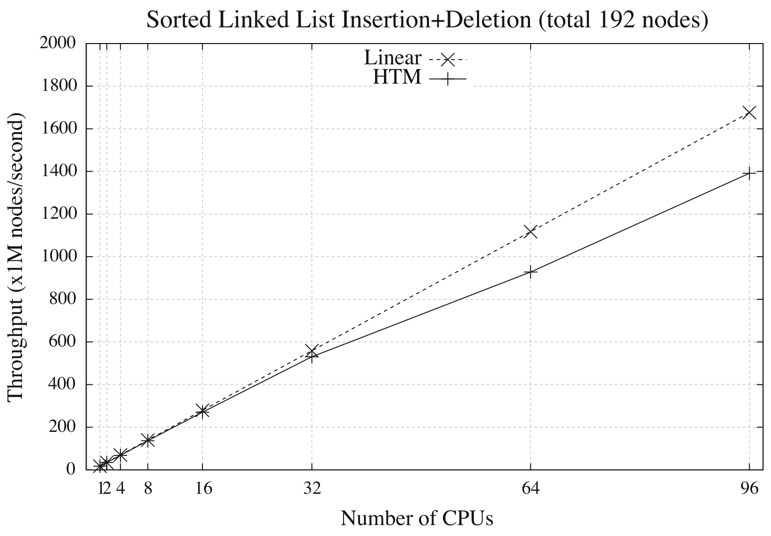

Fig. 16 Short sorted linked list scalability

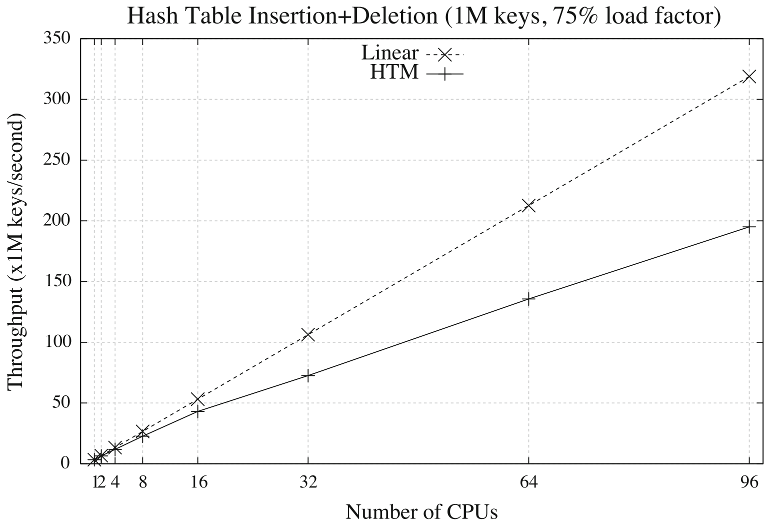

Fig. 17 Open addressing hash table scalability

tributed random keys of 64-bit long integer were inserted and deleted. We maintained a load factor of 75 %, which means that the hash table needed at least 1.3 million slots rounding up to the next power of 2. So the hash table had 2^{21}=2,097,152 slots. While the scalability was not as good as the disjoint sorted linked list, the improvement we saw is still a respectable result given the almost trivial amount of coding needed to make the hash table concurrent.

5 Discussion

Our experience with HTM on the zEnterprise EC12 is congruent with much of the previous work in that we found HTM solves some problems but presents new ones. The pros and cons we experienced included:

+ HTM is conceptually simpler to use than lock free techniques. Linearizability, opacity and the strong atomic guarantee make it easier to argue the correctness of the program.
+ HTM can perform and scale very well with disjoint data structures that minimize memory conflicts.
+ HTM uses no serialization resources so when it aborts there is no need to keep track of or clean up during failure recovery.

– To fully exploit HTM, one needs to understand the microarchitecture and HTM mechanism intimately. This makes HTM programs hard to write, debug, and optimize in practice even though they are conceptually simpler. This also makes HTM programs hard to port from architecture to architecture, and perhaps even from generation to generation on the same architecture.
– Most programmers are used to and prefer deterministic services. Best-effort probabilistic HTM services increase the complexity of programs in practice.
– Debugging HTM programs is challenging because the machine state during a transaction disappears after an abort when memory changes vanish and registers are restored.

We do feel that previous work has not placed enough emphasis on the impact of aborts on HTM. Our experience taught us that it is critically important to thoroughly understand the abort mechanism and behavior on an architecture in order to minimize aborts and maximize performance.

6 Related Work

Although a large body of prior HTM work exists, commercial HTM implementations remain scarce and so there is still a lack of evaluation and experience on commercial HTM. To date, only four other commercial machines besides the IBM zEnterprise EC12 exist: Sun ROCK [17], Azul Vega [18], IBM Blue Gene/Q [19], and Intel Haswell [20]. Papers [22], [18], and [23] provide valuable information on the these architectures, as well as evaluation and experience on them. Our work adds an entry to this collective effort. Wang et al. [23] has a summary table comparing some key characteristics of the three prior architectures. We extend the table to include Haswell and EC12, as shown in Table 2. Note that EC12 has two flavors of transactional execution: constrained (CTX) and unconstrained (UTX).

We used spanning in our disjoint data structure to overcome the physical transaction size limits. A similar idea called telescoping was proposed in [27]. A number of approaches [7–9] describe hardware architectures that overcome resource limitation to provide unlimited transaction size. They generally work by storing a log of overflowing transactional states in memory. Lev and Maessen [15] provides unlimited transactional

Table 2 Comparison of existing commercial HTM features

HTM	BG/Q	Rock	Azul	Haswell	EC12	
					CTX	UTX
Best effort	Yes	Yes	Yes	Yes	No	Yes
Buffer capacity	20MB	32 lines	16KB	64KB(w) 256KB(r)	128B	8KB(w) 96KB(r)
Speculative buffer	L2	L2	L1	L1	L1/L2	L1/L2
Reg. save/restore	No	Yes	No	Yes	Optional	Optional
Unsupported ISA	None	div/call/sync	None	Yes**	Yes*	Yes*
Conflict detection	8-64B	n/a	32B	64B	1B	1B
User-level abort	No	n/a	Yes	Yes	n/a	Yes

* The full list of restricted instructions can be found in [29]
** The full list of restricted instructions can be found in [20]

size by breaking a transaction into small segments. Each segment is executed by HTM but software maintains the states for the entire transaction. These are general solutions not tied to any particular application data structure and/or algorithm. Our experience suggests that in practice very large transactions may be of limited value. Since the rework penalty for aborts in large transactions is so severe the performance advantage of a very large transaction vanishes unless there is very little or no contention.

7 Conclusions and Future Work

We have reported our experience with the first commercial general purpose HTM on the IBM zEnterprise EC12 server. We demonstrated that, with disjoint data structures, HTM achieves scalability up to 96 CPUs. This result came after we spent considerable time understanding abort frequencies and the behavior of the hardware then aggressively refactored our data structure and code to minimize aborts. Our experience suggests that, while transactional memory provides a conceptually simpler programming model than traditional locking, the best effort hardware services and thus the need to handle aborts make programming HTM much more complex in practice. We feel that the necessary infrastructure can be provided by software service wrappers that encapsulate probabilistic HTM behavior and present a deterministic interface. In the context of the disjoint sorted linked list, we have also tested and analyzed the impact of different transaction size on performance. Our experience suggests that for a particular architecture, there is likely a sweet spot for the transaction size that provides the best performance with fewest aborts, neither too small nor too large. This further suggests that unbounded transactions are of questionable value. Our experience with open addressing hashing suggests that, given the right access scenario, HTM can be a powerful tool to improve scalable concurrency with very little extra coding effort.

For future work, we would like to look at structures other than sorted linked lists and hash tables. And we would like to implement our concurrent rehashing for hash table growth idea. Our tests with HTM have also uncovered certain microarchitectural behaviors which contribute to higher than necessary abort rates. We would like to feed our experience back to the hardware engineers to help improve the next generation of hardware.

Acknowledgments The authors would like to thank their colleagues for valuable help in this work. Christian Jacobi provided many insights into the Transactional Execution Facility hardware in zEnterprise EC12. Maged Michael discussed issues related to disjoint data structures. Robin Tanenbaum was critical in wrangling the unreleased hardware so the authors could actually conduct the experiments.

References

1. Knight, T.: An architecture for mostly functional languages. In: Proceedings of the 1986 ACM LISP and Functional Programming Conference
2. Herlihy, M., Moss, J.E.: Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, May (1993)
3. Shavit, N. Touitou, D.: Software transactional memory. In: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, August (1995)
4. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing, Boston, MA, July (2003)
5. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, NY, March (2006)
6. Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistency. In: Proceedings of the 31st Annual International Symposium on Computer Architecture, Munich, Germany, June (2004)
7. Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded transactional memory. In: Proceedings of the 11th IEEE Symposium on High-Performance Computer Architecture, February (2005)
8. Rajwar, R., Herlihy, M., Lai, K.: Virtualizing transactional memory. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture, June (2005)
9. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: LogTM: Log-based transactional memory. In: Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture, Austin, TX, February (2006)
10. Damron, P., Fedorova, A., Lev, Y.: Hybrid transactional memory. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October (2006)
11. Kumar, S., Chu, M., Hughes, C.J., Kundu, P., Nguyen, A.: Hybrid transactional memory. In: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, NY, March (2006)
12. Minh, C.C., Trautmann, M., Chung, J.W., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., Olukotun, K.: An effective hybrid transactional memory system with strong isolation guarantees. In: Proceedings of the 34th Annual International Symposium on Computer Architecture, San Diego, CA, June (2007)
13. Shriraman, A., Spear, M.F., Hossain, H., Marathe, V.J., Dwarkadas, S., Scott, M.L.: An integrated hardware-software approach to flexible transactional memory. In: Proceedings of the 34th Annual International Symposium on Computer Architecture, San Diego, CA, June (2007)
14. Lev, Y., Moir, M., Nussbaum, D.: PhTM: phased transactional memory. In: Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing, Portland, OR, August (2007)
15. Lev, Y., Maessen, J.-W.: Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Salt Lake City, UT, February (2008)

16. Yoo, R.M., Ni, Y., Welc, A., Saha, B., Adl-Tabatabai, A.-R., Lee, H.-H.S.: Kicking the tires of software transactional memory: Why the going gets tough. In: Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June (2008)
17. Tremblay, M.: Transactional memory for a modern microprocessor. In: Keynote speech at 26th Annual ACM Symposium on Principles of Distributed Computing, Portland, OR, August (2007)
18. Click, C.: Azul's experiences with hardware transactional memory. Bay Area Workshop on Transactional Memory, January (2009)
19. Haring, R., Ohmacht, M., Fox, T., Gschwind, M., Satterfield, D., Sugavanam, K., Coteus, P., Heidelberger, P., Blumrich, M., Wisniewski, R., Gara, A., Chiu, G.-T., Boyle, P., Chist, N., Kim, C.: The IBM blue gene/Q compute chip. IEEE Micro **32**(2), 48–60 (2012)
20. Intel Corporation: Intel Architecture Instruction Set Extensions Programming Reference. 319433-014, August (2012)
21. IBM: IBM zEnterprise EC12 Technical Guide. SG24-8049-00, September (2012)
22. Dice, D., Lev, Y., Moir, M., Nussbaum, D.: Early experience with a commercial hardware transactional memory implementation. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, Washington, DC, March (2009)
23. Wang, A., Gaudet, M., Wu, P., Amaral, J.N., Ohmacht, M., Barton, C., Silvera, R., Michael, M.: Evaluation of blue gene/Q hardware support for transactional memories. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, Minneapolis, MN, September (2012)
24. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (June 1990)
25. Guerraoui, R., Kapalka, M.: Opacity: a correctness condition for transactional memory. Technical Report LPD-REPORT-2007-004, EPFL, May (2007)
26. Rajwar, R., Goodman, J.R.: Speculative lock elision: enabling highly concurrent multithreaded execution. In: Proceedings of the 34th International Symposium on Microarchitecture, Austin, TX, December (2001)
27. Dragojevic, A., Herlihy, M., Lev, Y., Moir, M.: On the power of hardware transactional memory to simplify memory management. In: Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, San Jose, CA, June (2011)
28. Jacobi, C., Slegel, T., Greiner, D.: Transactional memory architecture and implementation for IBM system z. In: Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture, Vancouver, Canada, December (2012)
29. IBM: z/Architecture Principles of Operation. SA22-7832-09, 10th edn (2012)
30. Purcell, C., Harris, T.: Non-blocking hashtables with open addressing. Technical Report, University of Cambridge Computer Laboratory. UCAM-CL-TR-639, September (2005)
31. Martin, D.R., Davis, R.C.: A scalable non-blocking concurrent hash table implementation with incremental rehashing. Unpublished manuscript, December 1997
32. xxhash. https://code.google.com/p/xxhash/
33. MurmurHash. http://en.wikipedia.org/wiki/MurmurHash
34. Hash table. http://en.wikipedia.org/wiki/Hash_table
35. Open addressing. http://en.wikipedia.org/wiki/Open_addressing
36. Knuth, D.E.: The Art of Computer Programming, vol. 3, Addison-Wesley, Boston (1998)