StAdHyTM: A Statically Adaptive Hybrid Transactional Memory

A Scalability Study on Large Parallel Graphs

Mohammd A. Qayum*, Abdel-Hameed A. Badawy*, and Jeanine Cook†
*Klipsch School of Electrical and Computer Engineering, New Mexico State University, Las Cruces, NM
†Sandia National Labs, Albuquerque, NM
qayum@nmsu.edu, badawy@nmsu.edu, jeacook@sandia.gov

***Abstract*—In this paper, we present a Statically Adaptive Hybrid Transactional Memory (StAdHyTM) that outperforms not only existing Hardware TM (HTM) and Software TM (STMs) but also common synchronization schemes such as locks. StAdHyTM is statically tuned to adapt to the application behavior to improve the performance. We focus in particular on large parallel graph applications. Our StAdHyTM implementation outperforms coarse-grain locks by up to 8.1x and STM by up to 2.6x in total execution time for computation kernel in the SSCA-2 benchmark. It also outperforms HTM by up to 2.1x on a 28-cores and 64GB machine. We tested large graphs of up to 268 million vertices and 2.147 billion edges on a 64-core and 128 GB machine. To the best of our knowledge, this work is the first sc a lability st u dy of synchronizations involving all the TM implementations- HTM, STM, HyTM, and Adaptive HyTM.**

***Index Terms*—Hybrid Transactional Memory; Parallel Graphs; Scalability; Multicores; Synchronization; Locks**

I. Introduction

Computing has reached the limits of instruction level parallelism (ILP), maxed out the power envelope, and hit the memory wall, each at a varying degree of severity. We need to add to this list of walls the synchronization wall. This is inescapable in the era of multicore and manycore processors [1]. Researchers exerted significant e f f ort t o s o l ve the synchronization problem. Developers have tried tirelessly to avoid synchronization or resorted to naive coarse grain locks. To exploit all the cores in a system, developers have to concoct complex synchronization schemes such as fine g rain l ocks or non-blocking locks that are microarchitecture-aware [2].

In most big data graph applications, graphs are sparse in nature [3]. Therefore, transactional memory (TM) will outperform most existing synchronization schemes as it is inherently non-blocking [1] and provides more concurrency in low conflict c ases s uch a s c omputations o f c ritical sections in sparse graphs while most schemes are blocking in nature.

Interestingly, graphs with tens or even hundreds of billions of vertices and edges can fit in main memory or PCIe SSDs of a single node. Since most communications are taking place within a single node, shared memory multicore or manycore systems are significantly faster, more efficient, and less power-consuming (per core and per dollar) than some existing distributed memory systems with multiple nodes [4]. Furthermore, shared-memory data structures and algorithms are much simpler and easier to program than their distributed counterparts. TM used in shared memory architecture works well since it uses lightening fast communication on-chip and inter-chip communication through modified cache-coherence protocol [5].

TM policies can be implemented in software, hardware, or in a combination of them. A Hybrid Transactional Memory (HyTM) scheme uses a combination of Software Transactional Memory (STM) and Hardware Transactional Memory (HTM) depending on the requirements of the application [6].

An Adaptive Hybrid Transactional Memory (AdHyTM) is a TM scheme that adapts to the application's characteristics. The adaptation can be static or dynamic in its nature. In a static adaption, the TM policies are tuned by profiling before program execution. On the other hand, in a dynamic adaptation, policies are tuned at different phases of the program execution based on application behavior. HTM implementations are costly due to chip area constraints [7]. On the other hand, STM is limited by speed and high overheads since all the policies are implemented in high level abstractions (software) [8]. AdHyTM can be the best approach because there is no single best TM implementation for all applications given the resource limitations of HTM and slowness of STM.

AdHyTM combines a fast HTM as the primary execution path and a slow STM as the fallback path. It also enforces cooperation between HTM and STM to speedup the synchronization process in the general case and especially for large graph applications. Since most of the transactions will be executed in best-effort HTM path, it is beneficial to add a small overhead in interface to efficiently fallback to simple but slow STM path upon failures to get overall good performance.

Most HyTM designs use a combination of existing HTMs implemented in commercial architectures and various STMs to accelerate synchronization. In this paper, our StAdHyTM combines the HTM implemented in Intel's Broadwell architecture, code named TSX [9], and the STM implemented in the GCC compiler [10]. We tested our implementations on two SMP machines 28 and 64 cores respectively using large scale graphs (100s of millions of vertices and billions of edges) to demonstrate that TM, particularly, StAdHyTM is a better synchronization scheme.

978-1-5090-4228-9/17/$31.00

The contributions of this paper can be summarized as follows:

1) We have developed a novel yet simple HyTM optimized for large graph applications. We show that our StAdHyTM outperforms coarse-grain locks, the GCC STM and the Intel' HTM (TSX).
2) We have studied the scalability of synchronizing schemes for large graph applications. To the best of our knowledge, this is the first such scalability study of synchronizations involving all the TM implementations-HTM, STM and HyTM.
3) We also showed that large scale graphs (100s of millions of vertices and billions of edges) can be processed efficiently on a single node built from off-the-self hardware.

The rest of the paper is organized as follows: section II discusses existing synchronization schemes for graph applications and TM in detail. Section III describes our StAdHyTM implementation and Section IV discusses our experimental results. Section V discusses related works and Section VI concludes the paper.

II. Background

A. Parallel Processing on SMPs

A decade ago, only a few applications required performance beyond a single processor or required machines with specialized architectures such as massively parallel processing (MPP) or symmetric multiprocessing (SMP). These systems had custom built components and proprietary data-paths, and hence, were very expensive and harder to program.

Nowadays, due to the popularity of inexpensive off-the-self commodity components (processors, memories, hard disks, *etc.*), clusters of compute servers are dominantly used for HPC applications. Currently, the most common and the highest performing computer systems in the Top500 [11] supercomputers are clusters. Cluster computers are cost-effective, fast and easy to program since they are built on the top of popular commodity x86 architectures that use industry standard fast interconnects such as Gigabit Ethernet and InfiniBand.

Recently, Intel has released Xeon E7 Haswell (V3) EX (Extreme) processor [12]. This architecture supports up to eight processors in a single SMP node. Many vendors such as HP, Supermicro, Cisco, SGI *etc.* have released four socket servers that can have 72 cores or 144 logical cores (with hyperthreading) in a single node.

Intel Broadwell-EX [13] and Skylake-EX [14] will support eight sockets and Skylake-EX will support 28 cores. This means 224 cores or 448 logical cores in a single node which will be equivalent to a cluster with several nodes of older legacy processors.

In this paper, we use large shared memory systems with relatively large core counts and main memories that can compute on graphs with millions of nodes and billions of edges on a single node.

B. Synchronization in a large graph

In a large, random and complex graph, getting a connection *i.e.* a relationship between two nodes (*e.g.* shortest path) might seem quite hard. On the contrary, it is quite easy to find such connections in real world graphs [15]. The connection topology among the nodes of a large and complex graph is either random or regular. However, many large graphs that represent practical problems such as biological, technological and social networks have topologies that fall in between regular and random.

The diameter of a graph is defined as the shortest distance between two vertices. Distance is measured in terms of the number of edges that connect the two vertices. In large graphs, most of the vertices are connected by only a few edges. The diameter of a real world large graph follows the power law [16] *i.e.* as the graph size increases [17], the diameter of the graph does not necessarily grow but on the contrary it stays the same or possibly decreases. The power grid of the US, neural connections in our brains, and the collaboration of film actors exhibit the small world phenomenon [18]. This means that most nodes are connected via a small set of "powerful" nodes. This is also known as the "six degrees of separation" rule. Therefore, synchronization in the general case would involve non-directly connected nodes and thus the synchronization requirements for large parallel graph applications are different from regular applications in general. Synchronization that involves two directly connected nodes would infrequently happen on such graphs.

Hong *et al.* [19] have studied Breadth First Search (BFS) processing on a large parallel graph on a multi-socket multi-core system. Due to the small world phenomenon, the synchronization overhead is expected to be low. Due to the sparsity of the data structure [3] in most graph applications, transactional memory (TM) will outperform most existing synchronization policies as it is inherently non-blocking [20] and the chances of aborts due to conflicts will be relatively small.

C. Description of HyTMs

A Hybrid Transactional Memory (HyTM) scheme uses a combination of STM and HTM depending on the application requirements [6]. In most HyTMs, an HTM provides a fast execution path, while an STM serves as a fallback to handle situations where the HTM cannot execute the transaction successfully. Supporting the semantics of both HTM and STM is not trivial. For example, strong isolation can be lost if an HTM supports it whereas the STM does not. In HyTM, the hardware and software transactions not only coexist but also monitor common memory locations related to the status of a transaction. In a HyTM design, best-effort HTM [6] is combined with the best applicable STM through supporting libraries. Best-effort HTM tries to execute as many transactions as possible within its implementation constraints (*e.g.* cache size) but some transactions are bound to abort no matter what. Aborted and any remaining transactions are executed via an STM that suits the application requirements. Depending on how the HTM and STM are combined, there are three types of HyTMs that are common in the literature [8] (more details is provided in Section III):

1) A fast path HTM with a slow STM as a fallback *e.g.* Hybrid NOrec [7], ASF [21], and Reduced Hardware NOrec [8]
2) HTM and STM in phases *e.g.* PhTM [22]
3) HTM and STM in parallel *e.g.* Unbounded TM [23] and the HyTM of Damron *et al.* [6].

Most of the current HyTM designs take existing best-effort HTM from simulators or commercial implementations and add STM to them. Most of the complexity in HyTM is in implementing the software extensions. Therefore, there are many research opportunities for novel HyTM ideas such as TM extensions to higher level caches, a larger speculative store buffer, or using more hardware status registers to assist STM-HTM collaboration.

III. Implementation

A. Statically Adaptive HyTM (StAdHyTM)

StAdHyTM is an interesting approach because there is no size fits all TM *i.e.* no TM performs best for all applications. StAdHyTM combines a fast HTM as the primary execution path and a slow STM as a fall back path. It also forces the HTM to cooperate with the STM to speed up synchronization of applications. Since transactions are mostly executed in HTM, it is beneficial to make a few costly software extensions in the interface in order to efficiently fallback to a simple but slow STM path upon failure(s). For adaptability, StAdHyTM can take several approaches on how HTM and STM can cooperate:

1) Never run on HTM but run on STM if a task size is large due to limited hardware resources
2) Always run on HTM if task size is small
3) Run cooperatively with HTM and STM if task size is not large yet not small. In that case, since HTM cannot execute all the transactions due to capacity limits, STM can execute the remaining transactions.

As we have asserted in the previous sections, the sparsity of graphs in most real applications allows TM to outperform most existing synchronization techniques which are mostly blocking. We believe our StAdHyTM is a better TM approach because it not only combines the best features of HTM and STM, but also adapts to the application requirements. Our StAdHyTM also extends the HTM with several software extensions in the interface to allow cooperation with the STM to speed up the synchronization(s). The purpose of these extensions is to simplify the process of falling back to STM upon failure to execute in HTM. For designing our StAdHyTM, there are three possibilities:

1) To have a non-adaptive HyTM. In such a version, HTM will switch to STM when it fails to complete after a **fixed** number of retries.
2) To use a random retrial range, HTM will switch back to STM upon a **random** number of retries.
3) To tune the number of retries manually (statically) before falling back to STM for better performance with design space exploration **(DSE)** for retrial number ranges.

```
tries= NUM_RETRIES
if HW_BEGIN begins successfully
  if (gbllock is locked)
    abort
  else
    transactional code
    HW_COMMIT
else if (tries >= 0)
    decrement tries
    retry in HW
else //retrials quota ends
  atomic add(gblloc ,1)
  SW_BEGIN
  if (no conflict in SW)
        transactional code
        SW_COMMIT
  else
        SW_ABORT, retries in SW
  atomic sub(gblloc ,1)
return
```

Fig. 1: StAdHyTM pseudo code

Researches have used the first two policies to design HyTMs [24]. We chose to have a simplistic design for lower overheads than complex algorithms to adapt the fallback to STM. We have tried random number retrials (*e.g.* $1-50$) with a fast random number generator such as the "xorshift" [25]. It performs worse due to the time taken for generating the random number which turns out to be quite significant compared to total execution time compared to the best policy. We have chosen a fixed number with design space exploration through several random retrial ranges.

Our StAdHyTM implementation is shown in the code listing in Figure 1, a transaction first tries to execute in HTM (HW_BEGIN). If no STM transaction have captured the critical section (gblloc is free), HTM will commit (HW_COMMIT). If it fails to begin in HTM due to capacity limits, conflicts, or other reasons (*e.g.* context switch), it retries in HTM for a fixed number of retries. The number of retries (NUM_RETRIES) is set and tuned with several offline experiments to tune (statically) performance. When a transaction fails it gets assigned a number (quota) of possible retries. When it completes its quota, it will take a global lock and execute in STM (SW_BEGIN). However, every time a transaction tries to enter the critical section in HTM, it will check for the global lock's availability (if the critical section is already locked, the transaction will abort in HTM) which could be already acquired by another STM. The global lock can be captured by several STMs. When an STM takes the global lock, it increases its value by one. Therefore, when the value of the global lock decreases to zero, the HTM transaction(s) can go forward. Since an STM transaction can conflict with other STM transactions, in that case, one of the STM transactions will succeed (SW_COMMIT) and the global lock's value will be restored (decreased). Even if an STM transaction fails, it

TABLE I: Configurations of the machines our experiments.

Node Name	Shepard	Mickey
Processor version	Xeon E5-2698 V3	Xeon E5-2680 V4
Frequency	2.40 GHz	2.40 GHz
Microarchitecture	Haswell [12]	Broadwell [13]
# sockets per node	2	1
Cores/Logical Cores	32/64	14/28
L1 Cache/Core (d-/i- cache)	16/32KB	14/32KB
L2 Shared Cache	16x256KB	14x256KB
L3 Cache (smart cache)	35MB	35MB
Memory	128 GB DDR3	64GB DDR4

restores the lock's value. Therefore, the lock will be released eventually when an STM transaction commits or aborts.

B. HTMs for Benchmarking

We have experimented with three versions of HTMs. (1) HTM with atomic lock "HTMALock". Instead of an STM fallback, it reverts to a lock based execution. However, when a transaction reverts to using a lock, it waits for the lock to be free by other transactions before it can take the lock exclusively. This part is implemented with a while loop. (2) HTM with a spinlock which is called "HTMSLock". Hardware transactions frequently check the availability of the lock by spinning on the lock, while in the atomic lock version of HTM, hardware transactions atomically check for the availability of the lock. (3) Intel's Hardware Lock Elision (HLE). The transaction first attempts in speculative mode, if it fails, in the next attempt, it executes in non-speculative mode which aborts other concurrent speculative transactions.

IV. Experimental Results

We benchmark the scalability of our StAdHyTM against an STM, a coarse gain lock, and the three HTMs described in Section III-B. We conduct two sets of experiments on large graphs and kernels from the Scalable Synthetic Compact Applications graph analysis 2 (SSCA-2) [26] benchmark. We intend to compare our StAdHyTM with state-of-the-art STMs, HTMs and HyTMs, however, most STMs and HyTMs [7], [8] have high overheads due to complexity of implementations and may perform worse. Therefore, HTM on the native machine will be the fastest scheme that we can compare against.

SSCA-2 benchmark [26] is characterized by integer operations, a large memory footprint, and irregular memory access patterns. It has multiple kernels accessing a single data structure representing a weighted, directed multigraph. In addition to a kernel to construct the graph from the input tuple list, there are three additional computational kernels that operate on the graph. In this paper, we only use two of the four kernels. Namely, the generation kernel and the computation kernel. The generation kernel generates a large graph scale as the input. The generated graph is based on the power law and R-MAT format. The computation kernel extracts edges by weight from the generated graph and forms a list of the selected edges. This benchmark is readily available with an OpenMP version with its critical section having a built-in OpenMP lock. We modified the critical section to support STM, HTM, and StAdHyTM. All experiments are conducted

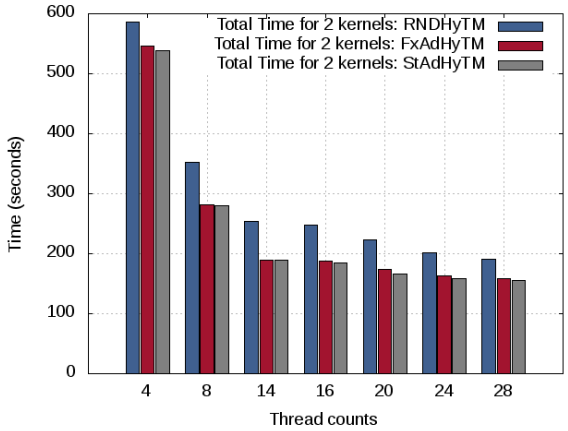


Fig. 2: Execution time of the Generation and Computation kernels with RndHyTM, FxHyTM & StAdHyTM for scale 27 on "Mickey" (Table I)

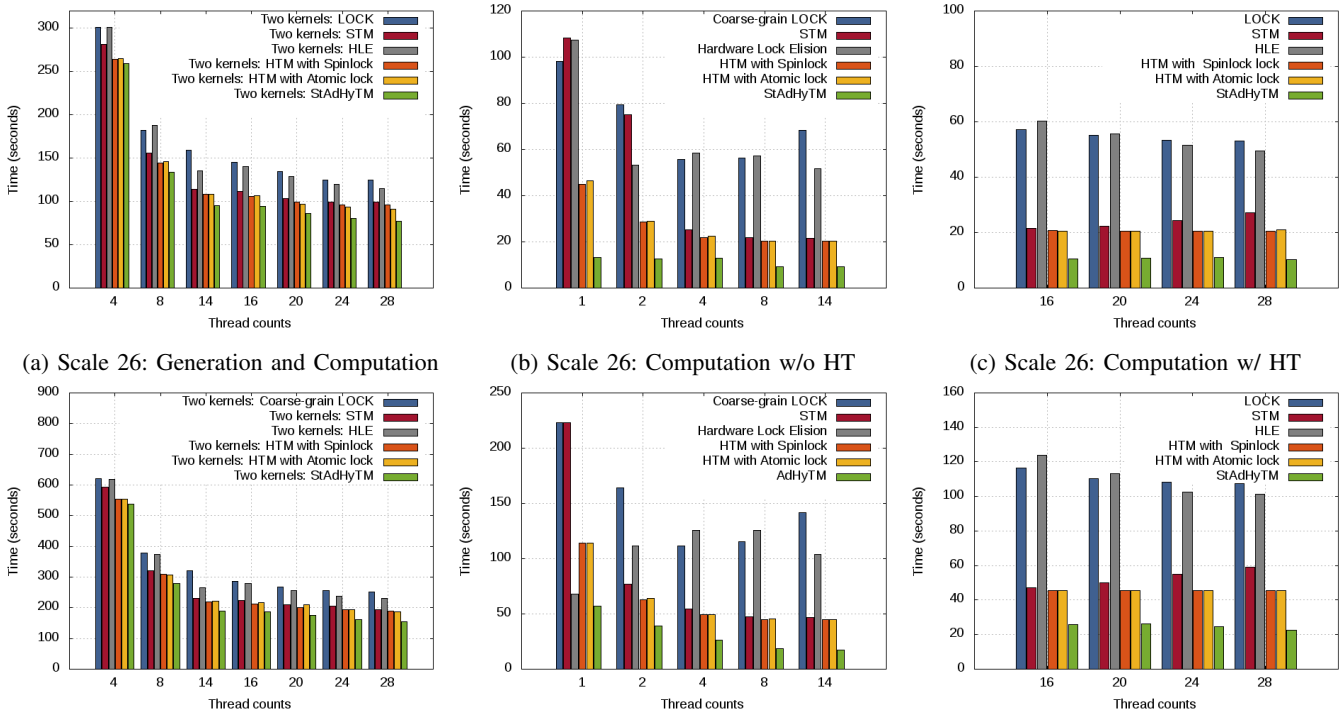
10 times and the mean execution time is reported. We have also considered more number of runs and the median. However, the results vary only by 0.2–3%. For example, for a thread count of 14 and a scale of 27, mean and median of computation kernel's (StAdHyTM) execution time are 16.6 and 16.3 seconds (*i.e.* a 2.33% difference) respectively, and standard deviation is only 0.836. We implement all our algorithms in C++ with GCC 5.4.

We show in Figure 2 a HyTM with random number of retrials (HyTM solution #2) in the (1–50) range. In this case, it performs worse than our StAdHyTM (HyTM solution #3). However, HyTM with a fixed number of retrials (FxHyTM) without design space explorations (HyTM solution #1) can have unpredictable performance. In the next set of experiments, we scale the core count and the scales of the SSCA-2 benchmark (*i.e.* scale the problem size up). The larger scales (*i.e.* 23 – 27) on all the policies were executed on "Mickey"(Table I).

Figures 3(a) and 3(d) show total execution time of both graph generation and computation kernels with all the synchronization policies for scales 26 and 27. Although, we have experimented with lower scales (23 – 25), we report 26 and 27 results only for space limitations.

The results show that a simplistic STM implementation outperforms coarse grain locking for all scales and all thread counts. We note that in the case of a lock, sequential execution is forced in the region that is being locked, while in TM, there is possibly more concurrency. StAdHyTM performs best among all the other policies.

For the computation kernel without hyperthreading, we gain about 7.3x and 8.1x speedup in execution time from using StAdHyTM compared to coarse-grain locking; and more than 2.2x and 2.6x speedup compared to the next best policy for a thread count of 14 as in Figures 3(b) and 3(d) for scales 26 and 27 respectively. While With hyperthreading, we gain about 4.8x and 5.2x speedup from using StAdHyTM compared to coarse-grain lock; and more than 2.05x and 2.02x speedup compared to the next best policy for thread count of 28 as in



(a) Scale 26: Generation and Computation

(b) Scale 26: Computation w/o HT

(c) Scale 26: Computation w/ HT

(d) Scale 27: Generation and Computation

(e) Scale 27: Computation w/o HT

(f) Scale 27: Computation w/ HT

Fig. 3: Performance improvement for: the two kernels (a) and (d), computation kernel without hyperthreading (b) and (e), computation kernel with hyperthreading (c) and (f). StAdHyTM is compared against coarse grain lock, STM, HLE, HTMs for large scale(s) 26 and 27 on a "Mickey" (Table I). The x-axis represents thread count and the y-axis represents execution time

Figures 3(c) and 3(e) for scales 26 and 27 respectively. These results are significant since, the speedup is in total execution time for a large graph where the critical section is only a fraction of the total computation.

In the spinlock version of HTM, hardware transactions frequently check the availability of the lock by spinning on the lock for retrials while in the atomic-lock version of HTM, hardware transactions atomically check the lock for its availability. Therefore, HTM with Spinlock perform slightly better than Atomic lock version of HTM. HLE is the worst performing scheme among HTM policies due to the high rate of aborts among HLE transactions. Once an HLE transaction fails in speculative mode, it begins in non-speculative mode and aborts all other concurrent speculative transactions. Since non-speculative HLE transactions are slower due to serialization, and in the meantime, it aborts a large number of speculative concurrent HLE transactions on the fly.

The largest scale we run on Mickey is 27. At this scale, most of the machine's memory (64 GBs) is utilized for storing the graph (134 millions vertices and 1.03 billion edges). Our StAdHyTM implementation outperforms coarse-grain lock by 1.62x and STM by 1.25x speedup for both kernels (generation and computation) at the largest thread count of 28 as in Figure 3(d). Both HTM policies outperform lock and STM by similar percentages, while StAdHyTM outperforms the next best policy, HTM with Spinlock by 1.21x. In StAdHyTM, since both HTM and STM transactions can execute cooperatively, we get better performance than just HTMs. However, very few HTM transactions fail to execute even after several retries and eventually, fall back to STM transactions although our goal is to maximize the number of transactions that succeed in committing in HTM. This is the case obviously because HTMs are the fastest. However, we can design an HTM without lock fallback that just keeps retrying with HTM, but it performs worse than falling back to a lock or STM. The graph generation kernel is a simple kernel with symmetric concurrency (*i.e.* conflict probability is similar), therefore execution times are similar for all TM policies.

The results show that for a thread count of 14, StAdHyTM gets the best execution time. Note that, beyond 14 threads on this machine we are using Hyperthreading. In Hyperthreading, the hardware threads share the caches, the microarchitecture and the functional units, and therefore threads become more resource constrained and eventually become slower and transactional conflicts arise due to cache capacity limitations. Therefore, all policies perform worse for thread counts beyond 14. In the case of a thread count of 28, overheads and conflicts kill the benefits of concurrency only for the computation kernel. However, overall gain in total execution time for both kernels comes from the generation kernel since it takes 6x the execution time of the computation kernel for a thread count of 28 for scale 27 (*i.e.* the generation kernel dominates execution of the two kernels).

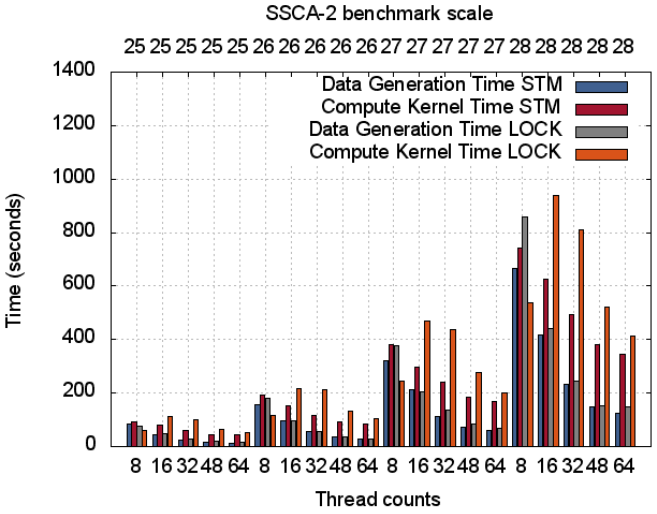


Fig. 4: Performance improvement of generation and computation kernel with STM over coarse grain lock (OMP lock) for scales 25 – 28 on "Shepard"

The largest scale (*i.e.* 28) of our experiments with the benchmark were executed on "Shepard" (Table I). A single SMP node from "Shepard" contains two sockets with 32 physical core (*i.e.* 64 with hyperthreading) and 128 GBs memory. Each socket has a single Xeon Haswell processor with 16 cores (*i.e.* 32 with hyperthereading). The largest scale that we can run on this node is 28. At this scale, most of the node's memory is utilized for storage of the graph (268 millions vertices and 2.147 billion edges). In this experiment, we are only able to test STM and OpenMP lock version of the benchmarks since HTM is not available in this version of the Intel's Processor.

Figure 4 shows the execution time of both the generation and computation kernels of the SSCA-2 benchmark for TM (*i.e.* STM) and coarse grain lock (*i.e.* OpenMP lock).

The results show that the STM outperforms the coarse grain lock for all the scales and most of the thread counts. At the largest scale (*i.e.* 28) and at a thread count of 64, STM outperforms lock in the generation kernel by 17% and computation kernel by 16%.

Results are similar for thread counts from 16 – 64, however, for thread counts of 8 or lower, we get an interesting scenario. For a thread count of 8 at scale 28, total execution time of both kernels with STM and lock are 1410.03 and 1399.34 seconds respectively. However, execution time of the computation kernel under STM is 743.1 seconds which is worse than the time for lock (539.19 seconds). The reason is that at eight threads, conflict probability is higher than in the case of larger thread counts, therefore, transactions are aborting frequently and taking more time than usual. This case further supports the premise that the larger the graph, the larger the need for more cores and more memory which is the conventional wisdom.

V. Related Work

We have experimented with most implementations of TM not only STMs but also HTMs and HyTMs. We have designed an StAdHyTM that outperforms HTMs of the native platform. In this section, we will do an overview of the most relevant and related works in the literature.

Kang and Bader [1] implemented an efficient minimum spanning forest (MSF) and it was one of the first research efforts to propose transactional memory for synchronization in graph applications. They implemented MSF with STM and reported better performance against coarse-grain locks. They predicted HTM or HyTM may perform better for graph applications. Interestingly, our StAdHyTM is one such implementation.

Shun and Blelloch [4] have designed a lightweight graph processing framework called ligra for shared memory multicore architectures to allow graph applications to outperform distributed implementations of graph algorithms. Their implementation was on a 40-core machine was more efficient than clusters with much more core counts. However, though their implementations is a good scalability study for overall graph applications, we studied specifically synchronization on graph applications and our StAdHyTM can be used in the ligra platform to improve the performance of graph applications further.

Shang *et al.* [27] have designed a two phase fine grain locking mechanism for synchronization of graph applications. Their design uses a locking-based synchronization for high degree vertices with high conflicts and a non-locking-based scheme for low degree vertices with low conflicts. It is an interesting implementation since some graph applications such as social networks have high degrees for some nodes while most nodes have low degrees. They implemented their non-blocking algorithm using STM. Interestingly, our StAdHyTM can be used in lieu of their non-blocking algorithm. They tested their implementation on 36 cores with 60 GBs of memory. The number of vertices of their graphs is in the range of 50 – 106 million vertices and the numbers of edges is in the range of 1.9 – 3.7 billion edges. In the same ballpark as our scales.

Most STM implementations have different purposes behind their designs *e.g.* NOrec [28] aims at maximizing performance at low thread counts. SwissTM [29] is optimized for high contention scenarios, and TinySTM [30] is a lightweight word-based STM implementation.

Most HTM implementations are agnostic of the applications' behaviors. The very first papers that discusses HyTMs are Damron *et al.* [6] and Kumar *et al.* [31]. Later on, other works include PhTM [22] that is based on the Sun Ultra Sparc HTM. Riegel *et al.* [32] proposed a HyTM based on AMDs proposed Advanced Synchronization Facility (ASF) which is simulator based. Wang *et al.* [33] proposed another HyTM based on IBM Blue Gene/Qs best-effort HTM with a Single-Global-Lock fallback. State-of-art HyTMs are Hybrid NOrec [7], Reduced Hardware-NOrec [8], and Invyswell [34]. All of them are based on existing HTMs such as Intel's TSX or AMD's ASF with a single global lock.

Our StAdHyTM is based on Intel's TSX and is tuned for large graph applications, designed with simplicity as a goal. It is also designed based on a single global lock for cooperation

between HTM and STM. Most of the state-of-the-art HyTMs are designed for general purpose applications and are tested with SSCA-2 benchmark mostly for small scales (less than 22) and under low thread counts (less than 8) [7], [8], [34]. In most of the related work, TM implementations are tested within high overhead instrumentation; however, we choose to compare our StAdHyTM against both low overhead STMs and HTMs which we believe are the fastest policies in native architectures. Finally, scalability for large graph applications was never in the equation before.

VI. Conclusions & Future Work

TM is easy to program, adapts to data granularity, inherently non-blocking and scalable. TM can provide the best optimized results for concurrent applications such as large parallel graphs. Implementing TMs only in hardware is costly in terms of chip area requirements and in software leads to high overheads. One solution is a hybrid TM scheme that combines the best features of both HTMs and STMs with static tuning to the applications. For a best effort HTM, we use Intel's TSX. For STMs, we are using GCC's STM. Our results show that our HyTM provides better results than conventional synchronization methods, STMs and even HTMs for graphs of large size as we scale the core/thread count, memory size and problem size. Our StAdHyHTM outperforms coarse-grain lock by 1.62x, STM by 1.25x, HTM with Spinlock by 1.21x in speedup for two kernels of SSCA-2 benchmark on a 28-cores and 64GB of memory machine. In the future, we would like to explore a dynamically (at runtime) adaptive HyTM.

References

[1] S. Kang and D. A. Bader, "An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs," in *ACM Sigplan Notices*, vol. 44, no. 4. ACM, 2009, pp. 15–24.

[2] B. Cantrill and J. Bonwick, "Real-world concurrency," *Queue*, vol. 6, no. 5, pp. 16–25, 2008.

[3] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[4] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM SIGPLAN Notices*, vol. 48, no. 8. ACM, 2013, pp. 135–146.

[5] M. Spear, A. Shriraman, V. Marathe, S. Dwarkadas, M. Scott, D. Eisenstat, C. Heriot, and W. Scherer, "Hardware acceleration of software transactional memory," 2005.

[6] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *ACM Sigplan Notices*, vol. 41, no. 11. ACM, 2006, pp. 336–346.

[7] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear, "Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 39–52, 2011.

[8] A. Matveev and N. Shavit, "Reduced hardware norec: An opaque obstruction-free and privatizing hytm," *TRANSACT*, 2014.

[9] R. Rajwar and M. Dixon, "Intel transactional synchronization extensions," in *Intel Developer Forum San Francisco*, vol. 2012, 2012.

[10] M. Schindewolf, A. Cohen, W. Karl, A. Marongiu, and L. Benini, "Towards transactional memory support for gcc," in *1st GCC Research Opportunities Workshop*, 2009.

[11] "Top500 lists," https://www.top500.org/lists/, 2016, [Online; accessed 1-August-2016].

[12] P. Hammarlund, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, and S. Kaushik, "Haswell: The fourth-generation intel core processor," *IEEE Micro*, no. 2, pp. 6–20, 2014.

[13] A. Nalamalpu, N. Kurd, A. Deval, C. Mozak, J. Douglas, A. Khanna, and F. Paillet, "Broadwell: A family of ia 14nm processors," in *Symposium on VLSI Circuits (VLSI Circuits)*. IEEE, 2015, pp. C314–C315.

[14] E. Fayneh, M. Yuffe, E. Knoll, M. Zelikson, and Abozaed, "4.1 14nm 6th-generation core processor soc with low power consumption and improved performance," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2016, pp. 72–73.

[15] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-worldnetworks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[16] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *ACM SIGCOMM computer communication review*, vol. 29, no. 4. ACM, 1999, pp. 251–262.

[17] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999.

[18] J. Kleinberg, "The small-world phenomenon: An algorithmic perspective," in *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. ACM, 2000, pp. 163–170.

[19] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 78–88.

[20] V. Pankratius and A.-R. Adl-Tabatabai, "A study of transactional memory vs. locks in practice," in *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2011, pp. 43–52.

[21] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman, "Asf: Amd64 extension for lock-free data structures and transactional memory," in *MICRO, 2010 43rd Annual IEEE/ACM International Symposium on*. IEEE, 2010, pp. 39–50.

[22] Y. Lev, M. Moir, and D. Nussbaum, "Phtm: Phased transactional memory," in *Workshop on Transactional Computing (Transact)*, 2007.

[23] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. IEEE, 2005, pp. 316–327.

[24] Q. Wang, S. Kulkarni, J. Cavazos, and M. Spear, "A transactional memory with automatic performance tuning," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 54, 2012.

[25] F. Panneton and P. L'ecuyer, "On the xorshift random number generators," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 15, no. 4, pp. 346–361, 2005.

[26] D. A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, and T. Meuse, "Hpcs scalable synthetic compact applications 2 graph analysis," *SSCA*, vol. 2, p. v2, 2006.

[27] Z. Shang, F. Li, J. X. Yu, Z. Zhang, and H. Cheng, "Graph analytics through fine-grained parallelism."

[28] L. Dalessandro, M. F. Spear, and M. L. Scott, "Norec: streamlining stm by abolishing ownership records," in *ACM Sigplan Notices*, vol. 45, no. 5. ACM, 2010, pp. 67–78.

[29] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," in *ACM Sigplan Notices*, vol. 44, no. 6. ACM, 2009, pp. 155–165.

[30] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-based software transactional memory," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 21, no. 12, pp. 1793–1807, 2010.

[31] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2006, pp. 209–220.

[32] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer, "Optimizing hybrid transactional memory: The importance of nonspeculative operations," in *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2011, pp. 53–64.

[33] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of blue gene/q hardware support for transactional memories," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 127–136.

[34] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy, "Invyswell: a hybrid transactional memory for haswell's restricted transactional memory," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 187–200.