Integrating and Optimizing Transactional Memory In a Data Mining Middleware

Vignesh T. Ravi   Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University Columbus OH 43210
{raviv,agrawal}@cse.ohio-state.edu

Abstract

As the size of available datasets in various domains is growing rapidly, there is an increasing need for scaling data mining implementations. Coupled with the current trends in computer architecture, where scaling only seems possible with effective utilization of the increasing number of cores, this is leading to a programmability *and* performance *challenge for data mining applications on emerging multi-core architectures. Recently,* Software Transactional memory *(STM) has been gaining popularity as a viable tool for easing programmability on shared memory machines.*

This paper focuses on utilizing, optimizing, and evaluating STM for data mining applications on multi-core architectures. The specific contributions of this paper are three-fold: 1) An existing STM algorithm (Transactional Locking II) has been integrated with a parallel data mining middleware, FREERIDE. This enables transparent use of the STM technique by any application developed using this middleware. 2) We have developed a new Hybrid Replication- Transactional Memory scheme, which substantially reduces the memory overhead of a replication scheme, while also reducing the number of conflicts and aborts in the STM technique, and 3) We have performed a comprehensive performance evaluation of STM techniques, where they have been compared with a replication-based scheme (which may not be scalable with increasing number of cores), and a highly optimized locking scheme. Our results show that, both STM and HyRepSTM techniques are competitive with other schemes in most cases. Also, the Hybrid Replication-Transactional memory scheme substantially reduces the number of aborts and conflicts when the number of concurrent threads are high.

1 Introduction

The availability of large datasets and increasing importance of data analysis in commercial and scientific domains is creating a new class of high-end applications. Recently, the term *Data-Intensive SuperComputing* (DISC) has been gaining popularity [2], and includes applications that perform large-scale computations over massive datasets. The deluge of available data for analysis demands the need to scale the performance of data mining implementations. Starting within the last 3-4 years, it is no longer possible to improve processor performance by simply increasing clock frequencies. As a result, multi-core architectures have become cost-effective means for scaling performance. The RMS (Recognition, Mining, Synthesis) analysis from Intel [11, 25] points out that data mining and gaming applications are critical for multi-core architectures. There is clearly a trend towards increasing number of cores on chips. For example, the Larrabee architecture proposed by Intel [31] has indicated that many-core CPU/ GPU can provide a promising platform for optimized data-parallel processing.

Thus, one of the major challenges today is *programmability* and *performance* for data mining applications on multi-core and many-core machines, especially with the trend towards growing number of cores. In recent years, there has been a significant amount of work on transactional memory as a shared memory programming approach [9, 8, 12, 14, 21, 29, 32]. The goal of transactional memory is to help ease parallel programming, by allowing programmers to declare interactions with shared objects as transactions, and letting the runtime system maintain correctness when there are concurrent accesses.

This paper focuses on utilizing, optimizing, and evaluating transactional memory for data-intensive applications on multi-core systems. Particularly, we have focused on three issues. First, we show how transactional memory can be transparently used through a parallel data mining middleware, i.e., users can program an application with the same API, and use either of replication, locking, or transactional memory. Particularly, we have integrated the Rochester Software Transactional Memory (RSTM) library with FREERIDE (FRamework for Rapid Implementation of Datamining Engines) [17, 18, 19]. FREERIDE is based upon the observation that parallel versions of several well-known data mining techniques share a relatively similar structure. The computation on each node involves reading the data instances in an arbitrary order, processing each data instance, and performing a *local reduction*. The reduction involves only commutative and associative operations, which means the result is independent of the order in which the data instances are processed. After the local reduction on each node, a *global reduction* is performed. This structure is exploited by FREERIDE to enable parallelization on a variety of parallel architectures starting from a high-level API. Through our integration of RSTM, we allow the use of transactional memory as another shared memory parallelization technique, in addition to replication and locking based mechanisms previously developed in FREERIDE.

Second, we have developed a new technique to help overcome the main impediment to scaling transactional memory to large number of cores, which is the increasing number of conflicts with larger number of threads. These conflicts lead to more aborts. We combine transactional memory with replication. This hybrid scheme has significantly fewer aborts than transactional memory, and at the same time, does not have the memory overheads of just using full replication.

Third, we present a detailed experimental study, focusing on the use of transactional memory for data-intensive applications, and on comparing the performance of transactional memory with replication and locking based schemes. Overall, we show that in most cases we experimented so far, transactional memory can be competitive with full replication, a technique that is not expected to scale with increasing number of cores, and cache-sensitive locking, a highly optimized locking scheme that needs to be tuned for every architecture. Further, we show that the hybrid replication-transactional memory scheme we have developed, significantly reduces the number of aborts as the number of concurrent threads increase.

2 FREERIDE Middleware

This section gives an overview of the FREERIDE middleware, focusing on the API it offers and shared memory parallelization techniques that were earlier implemented in it. The next two sections focus on the integration of transactional memory and a new hybrid parallelization technique we have developed.

The FREERIDE system was motivated by the difficulties in implementing and performance tuning parallel versions of data-intensive algorithms. FREERIDE is based upon the observation that parallel versions of several well-known data mining, OLAP, and scientific data processing applications share a relatively similar structure, which is that of a *generalized reductions*. This observation has some similarities with the *map-reduce* paradigm that Google has developed [6]. There are also some differences in the generalized reductions that FREERIDE supports and the map-reduce style of computations. Particularly, the FREERIDE API alleviates the need for expensive sorting of reduction elements, and thus can help achieve better performance on data mining applications.

2.1 Middleware Interface

The interface exploits the similarity among parallel versions of several data-intensive algorithms, as we have observed. The following functions need to be written by the application developer using our middleware. Most of these functions can be easily extracted from a sequential version that processes main memory resident datasets.

Local Reductions: The data instances owned by a processor and belonging to the subset specified are read. A local reduction function specifies how, after processing one data instance, a *reduction object* (declared by the programmer), is updated. The result of this processing must be independent of the order in which data instances are processed on each processor. The order in which data instances are read from the disks is determined by the runtime system. The reduction object is maintained in the main memory.

Global Reductions: The reduction objects on all processors are combined using a global reduction function.

Iterator: A parallel data-intensive application comprises of one or more distinct pairs of local and global reduction functions, which may be invoked in an iterative fashion. An iterator function specifies a loop which is initiated after the initial processing and invokes local and global reduction functions.

2.2 Exploiting Multi-Cores for Scalable Data Processing

```
{* Outer Sequential Loop *}
While() {
    {* Reduction Loop *}
    Foreach(element e) {
        (i, val)   =   Compute(e) ;
        RObj(i)    =   Reduc(RObj(i),val) ;
    }
}
```

Figure 1. Structure of Common Data Mining Algorithms

Initially, we describe the challenges in enabling multi-processing for our target class of applications.

We can extract a common structure that fits a variety of data-intensive applications. This structure is shown in Figure 1. The function *Reduc* is an associative and commutative function. Thus, the iterations of the for-each loop can be performed in any order. The data-structure *RObj* is referred to as the reduction object.

The main correctness challenge in parallelizing a loop like this on a shared memory machine arises because of possible race conditions (or *access conflicts*) when multiple threads update the same element of the reduction object. A number of factors make these loops challenging to execute efficiently and correctly. First, it is not possible to statically partition the reduction object so that different processors update disjoint portions of the collection. Thus, race conditions must be avoided at runtime. Second, the execution time of the function $Compute$ can be a significant part of the execution time of an iteration of the loop. Thus, runtime preprocessing or scheduling techniques cannot be applied to avoid race conditions. Furthermore, in many of algorithms, the size of the reduction object can be quite large. This means that the reduction object cannot be replicated or privatized without significant memory overheads. Moreover, because of the number of elements in the reduction object, the memory overhead of locks (or latches) can also be significant. Finally, the updates to the reduction object are *fine-grained*. The reduction object comprises a large number of elements that take only a few bytes, and the foreach loop comprises a large number of iterations, each of which may take only a small number of cycles. Thus, if a locking scheme is used, the overhead of locking and synchronization can be significant.

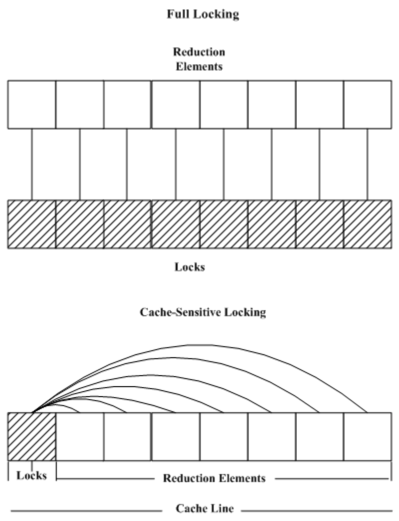

Figure 2. Memory Layout for Locking Schemes

Two obvious approaches for addressing these problems are, *Full-replication* and *Full-locking*. In the Full-replication approach, each processor can update its own copy of the reduction object and these copies are then merged together later. In the full-locking approach, shown in Figure 2, one lock or latch is associated with each aggregated value. However, in our experience with data-intensive algorithms, the memory hierarchy impact of using locking when the reduction object is large was significant. We observed that supporting a large numbers of locks results in overheads of two types. The first is the high memory requirement associated with a large number of locks. The second overhead comes from cache misses. Consider an update operation. If the total number of elements is large and there is no locality in accessing these elements, then the update operation is likely to result in two cache misses, one for the element and second for the lock. This cost can slow down the update operation significantly on modern machines with deep memory hierarchies.

To overcome these overheads, we have designed several more efficient locking techniques [18]. Among these, *cache-sensitive* locking resulted in best performance in most cases, also shown in Figure 2. Thus, in comparing other schemes with locking, we will only consider cache-sensitive locking. The basic idea in this scheme is as follows. Consider a 32 byte cache block and a 4 byte reduction element. We use a single lock for all reduction elements in the same cache block. Moreover, this lock is allocated in the same cache block as the elements. So, each cache block will have 1 lock and 7 reduction elements. Cache-sensitive locking reduces both the types of overheads associated with full-locking, i.e., it reduces the memory requirements and the number of cache misses.

3 Software Transactional Memory (STM)

3.1 Background

Over the last few years, transactional memory programming scheme has appeared as a promising tool for alleviating the difficulties in the parallel programming. Transactional Memory was initially designed as a novel architectural support for synchronization of non-blocking data structures [15]. This idea was adapted to a software implementation, referred to as Software Transactional Memory (STM), by Shavit and Touitou [32]. With the current trend of increasing number of cores, traditional fine-grained locking mechanisms are not considered scalable, besides being difficult to manage for programmers. While coarse-grained locks might be easier to program with, the resulting performance is often not adequate.

A Software Transactional Memory system can help automatically convert a sequential execution into a correct concurrent execution [21]. In order to write parallel programs using STM, the programmer simply has to tag the code sequence that uses the shared memory as an atomic and isolated transaction. When multiple threads are executed in parallel, concurrency control is the responsibility of the STM runtime system. A number of STM systems have been designed and developed [14, 20, 12, 9, 29, 8]. The research in STM has taken two directions: Non-blocking STMs [14, 20, 32] and blocking or lock-based STMs [9, 8, 12, 29]. Several efforts have also targeted

Hardware-only Transactional Memory systems, which implement version management and conflict detection by improving the cache-coherence [1, 15, 13, 24, 26], and Hybrid Transactional Memory systems [4, 3, 23, 33], which use a mix of both hardware and software implementation to reduce the overheads incurred by STM. Ennals [12] has argued that STM systems should not be non-blocking, and more specifically, *obstruction-free*. However, previous research and experimental results [9, 8, 10] indicate that lock-based STM is better than non-blocking STMs, at least by a factor of 2 for most cases.

Since performance is very important for the data-intensive applications we are targeting, we focus on lock-based STMs. Particularly, we are using Transactional Locking II (TL2) algorithm developed by Dice *et al.* [8]. Since we are integrating transactional memory with FREERIDE, which has been developed using C++ and pthreads, we use the Rochester Software Transactional Memory [22], which is a C++ library with an implementation of TL2 and other algorithms.

3.2 Transactional Locking II

We give a brief overview of this algorithm. A detailed description can be found from the original paper by Dice *et al.* [8]. The Transactional Locking II (TL2) algorithm is a global version-clock based two phase locking scheme that acquire locks at the commit time. Since this is a *word-based* locking scheme, a lock is associated with every word in the memory. Only one bit in the lock is used to identify if the word is locked and the remaining bits are used to store the write-version (time-stamp) of the word updated by the transaction that modified the data last.

The following are the four basic operations involved in the TL2 algorithm.

STMBeginTransaction - A transaction is started using this function by checkpointing and recording the global clock in the read version variable RV.

STMWrite - This function is used to update any word in the memory. First, it checks for any conflict with the committing or committed transactions using the lock. A conflict is detected when the word is locked or when the time-stamp in the lock is greater than RV. In this case, if there is a conflict, the transaction aborts immediately. If there is no conflict, the address to be updated and value are added to the $write - log$.

STMRead - This function is used when a word is read by a transaction. First, it searches the address to be read in the $write - log$. If it is found, then the value is read and returned to the user. Otherwise, it checks for any conflict with the committing or committed transaction. If there is no conflict, address is added to the $read - log$ and the data is read from the memory and returned.

STMCommit - This function is used when the work done by a transaction needs to be committed. It first tries to acquire the locks for all addresses in the $write - log$. If it fails on any lock, conflict is detected and the transaction aborts. Assuming no conflict while acquiring locks, it then atomically increases the global version clock. Then, it performs the read validation for all using the $read - log$. If there is no conflict at this stage, then the transaction is guaranteed to commit successfully. Then the corresponding memory addresses are updated from the $write - log$. The last step is to update the new time stamp for the updated words and release the locks.

If the lock acquire fails while committing, it spins over the lock for a limited time and then eventually aborts. Conflicts in all other cases will be aborted. The aborted transactions will be retried after a random back-off.

3.3 Motivation for Our Work

Our effort on integrating and further optimizing transactional memory with FREERIDE was driven by several goals.

The first is the limitation of the existing parallelization techniques for this class of applications. Full replication has high memory overheads, and cannot continue to scale on many-core architectures. Cache-sensitive locking is a technique that needs to be tuned for each cache configuration, and needs to be implemented using assembly language. This implies a significant challenge, and risk of introducing bugs, in porting the middleware to a new architecture. While the same is also true for transactional memory implementations, we can leverage on the large number of existing efforts in this area.

Integrating a TM library with FREERIDE offers other advantages as well. As we will show, we allow a *transparent* use of TM library, i.e., an application programmer using FREERIDE can use transactional memory without needing to know any details of the RSTM library. Thus, from a simple API that FREERIDE supports, replication, locking, and transactional memory can all be used transparently. Besides giving more options to users, it also enables detailed comparison of the performance achieved by different techniques.

4 Using RSTM (TL2) library for FREERIDE

While STM systems manage the difficulty of concurrency control, the underlying APIs are not yet popular in the broader parallel computing community. Moreover, using STM for concurrent programming requires the understanding of transactions and caveats involved with the transactional memory programming. In this paper, we have integrated the RSTM implementation with our FREERIDE middleware. The use of RSTM's implementation of TL2 is offered as another shared memory technique, along with replication and the various locking based approaches. This integration enables the application programmers to choose

STM technique, while not having to program or know the details of the RSTM library. Thus, our integration allows a *transparent* use of TL2 implementation for data mining application development.

To understand how we can transparently support RSTM from within FREERIDE, we initially consider how this transactional memory library can be used by programmers. A more detailed description of the API is available elsewhere [5]. $init()$ and $shutdown()$ functions need to be called to initialize and clean up the system on a per-thread basis. The $shared$ class encapsulating the shared elements should be derived from $stm :: Object$. The elements could be array or simple built-in types. This makes the newly created class, which is the shared transactional memory. All elements in the $shared$ class should be created using $GENERATE\_DATATYPE$ macros. This internally creates $m\_element$, with a $getter$ and a $setter$ for $m\_element$. The container class that use the transactional memory should create a shared pointer (sh_ptr) for the $shared$ class. The code sequence that reads or writes the shared memory should be declared with $BEGIN\_TRANS$ and END_TRANS macros. This makes it an atomic and isolated transaction. Words in the transactional memory can be opened in Read-only and Read-Write mode only using read pointer (rd_ptr) and write pointer (wr_ptr) respectively. It should be noted that conflict detection, conflict resolution, commit, abort and retry mechanisms are handled internally by the STM system and hence it is separated from the application and the reduction logic.

Since, in our research we are focusing only on parallelizing data-intensive applications, RSTM API need to be mapped to the generalized reduction loop shown in Figure 1. In FREERIDE, the reduction object is the shared memory that is to be read and updated by the applications. In our middleware, the reduction object takes the structure of a 2-D array that could be dynamically allocated and resized through FREERIDE API. All the reads and writes to this reduction object are normally performed using the FREERIDE $read()$ and $write()$ API. The high-level ideas in our implementation are as follows. All the threads that share the reduction object have to initialize the STM system at first. As these threads are created and managed within the FREERIDE middleware, such initialization is also transparent to the application developers. For using STM, this reduction object has to be converted to be a software transactional memory. This conversion is done by deriving our $reductionobject$ class from RSTM $stm :: Object$ class. Again, since memory allocation for reduction object is from within the middleware, these details can be hidden from the application developer. Once, the object is converted to a transactional memory, all the reads and writes performed on this object have to be declared as atomic transactions. Also, all the reads and writes can be performed only through the

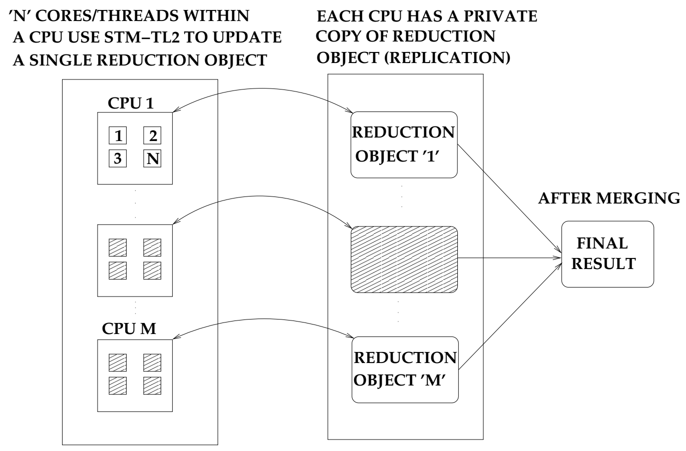
'N' CORES/THREADS WITHIN A CPU USE STM–TL2 TO UPDATE A SINGLE REDUCTION OBJECT
EACH CPU HAS A PRIVATE COPY OF REDUCTION OBJECT (REPLICATION)
CPU 1
1
2
3
N
CPU M
REDUCTION OBJECT '1'
REDUCTION OBJECT 'M'
AFTER MERGING
FINAL RESULT

Figure 3. Hybrid Scheme

special access pointers provided by RSTM API. We invoke these functions from the code supporting the FREERIDE API.

One important issue is the granularity of the transactions. In FREERIDE, the reduction function can be called for each data element by the application. In this function, the data element is processed and the resulting (i, val) pair is updated into the reduction object using a special pointer. Consistent with this structure, we make the processing associated with every data element a separate transaction. In the future, we will consider other alternatives, such as allowing the process associated with a set of data elements to be a single transaction. Our current choice of the granularity of transaction results in a low number of conflicts, and a relatively small overhead of aborts. However, the number of transactions can be very large, and start-up costs can be high.

5 Hybrid Replication and Software Transactional Memory

As the number of cores in multi-core and many-core architectures is expected to continue to grow, scalability of shared memory parallelization techniques is extremely important. This section describes an approach that combines both replication and the use of transactional memory. The goal of this approach is to alleviate the main scalability bottlenecks associated with each of replication and transactional memory.

Clearly, the main bottleneck with full replication is the increase in memory requirements with increasing number of cores. For certain applications, it can severely impact the scalability, as we have seen even in our earlier experiments with 8 cores [28]. STM has a different set of overheads, which have been widely studied [21, 14, 20, 12, 9, 29, 8, 3, 10]. Most importantly, whenever there is a conflict, the conflicting transactions needs to be aborted. When aborted, all the operations already completed by the transaction need to be rolled back. Thus, aborts are clearly a detrimental factor in the performance of STM. The number of conflicts, which leads to aborts, can increase rapidly with increasing number of threads (or cores).

To address these problems, we have designed and implemented a new scheme, which we refer to as the Hybrid Replication-transactional Memory scheme. Let us suppose we have $T = M \times N$ threads. We can create M copies of the reduction object. We also create M groups of N threads. All N threads in a group update elements from the same copy of the reduction object. Within each such group, correctness is maintained using transactional memory. After all input elements have been processed, the M copies of the reduction object are merged.

This scheme can be particularly attractive for a multi-core or a many-core machine that has M CPUs with each CPU containing N cores. When different CPUs have independent $L2$ caches, we can more effectively exploit the aggregate cache using separate copy of the reduction object on each CPU. This case is shown in Figure 3. We can further generalize the idea for cases where we want to run T threads, such that T is greater than $M \times N$. We can still create M copies of the reduction object, and have T/M threads on each CPU to update a single reduction object.

The advantages of this scheme are three-fold. First, the number of replicated copies of reduction object are reduced by a factor of N, as compared to the full replication scheme. This reduces the memory requirements of the Full-replication scheme, and hence will also likely reduce the cache misses due to the reduction object. Second, it also reduces the merging cost by a factor of N. Third, as compared to the normal use of STM, where T threads will compete for one memory location, now only T/M threads are competing for one memory location. Hence, we are reducing the conflicts and aborts by a significant factor.

6 Experimental Results

6.1 Parallel Efficiency of Datamining Applications

This section reports on a number of experiments we conducted to evaluate the use of transactional memory and our hybrid replication-transactional memory scheme. The multi-core machine we used is an Intel xeon CPU E5345, comprising two quad-core CPUs, with 6GB main memory and 8 MB L2 cache. Each core has a clock frequency of 2.33GHz. Our experiments compared four approaches, which were, *Full-replication*, *Cache-sensitive locking*, *TL2*, which is one of the transactional memory algorithms implemented through RSTM, and the hybrid replication-transactional memory. Throughout this section, these four techniques are referred to as `f-r` (for Full-replication), `cs-l` (for Cache-sensitive locking), `stm-tl2` (for TL2 transactional memory algorithm) and `rep-stm` (for the hybrid replication-transactional memory scheme).

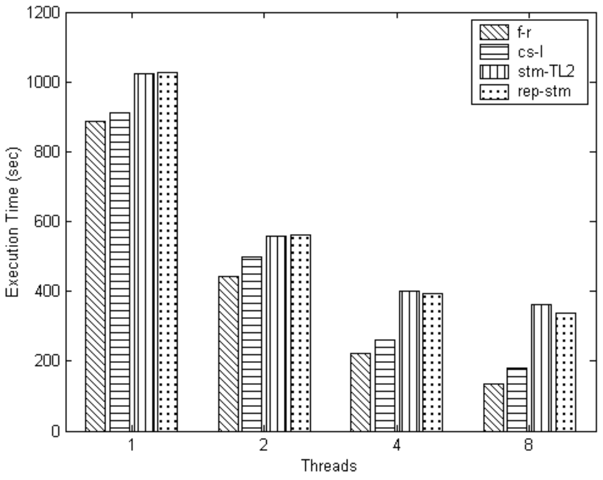

Figure 4. Results from K-Means

Though our experiments are conducted on machine with 8 cores, the main issue we want to focus on is the scalability of the techniques. With expected growth in the number of cores and memory limitations in most systems, we cannot expect `f-r` scheme to continue to scale. However, this limitation of `f-r` is not seen on 8 cores for most applications. It should also be noted that the locking scheme we compare against, which is cache-sensitive locking, needs to be optimized on each architecture, by programming in assembly. Thus, it is a challenge to port this scheme correctly to each architecture.

Our experiments were conducted using three popular data mining applications. K-means [16] is one of the most popular algorithms for clustering, a key data mining problem. Expectation Maximization (EM) is another popular clustering algorithm [7]. The third application we use, Principal Components Analysis (PCA) is a popular dimensionality reduction method. This method was developed by Pearson in 1901.

Results from K-Means: We used a dataset of size 6GB. The dataset contains 3-dimensional points. The number of clusters, k was set to be 250. The results are presented in Figure 4. The size of reduction object in K-Means is quite small, which allows replication-based scheme to scale easily. Thus, `f-r` outperforms `cs-l`, `stm-tl2` and `rep-stm` techniques. The speedups with 8 threads and the use of `f-r`, `cs-l`, `stm-tl2`, and `rep-stm` techniques were 6.57, 4.9, 2.82, and 3.14, respectively. Some observations are as follows. The overhead for `cs-l` is as low as 2.81% ,when compared to a single thread version with `f-r`, while the overhead of `stm-tl2` and `rep-stm` are 15.3%, which is quite high. This is because in this application, the amount of computation between updates in not quite high.

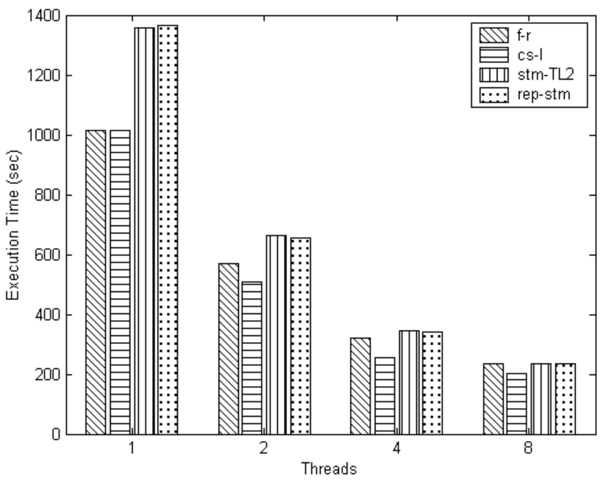

Figure 5. Results from E-M

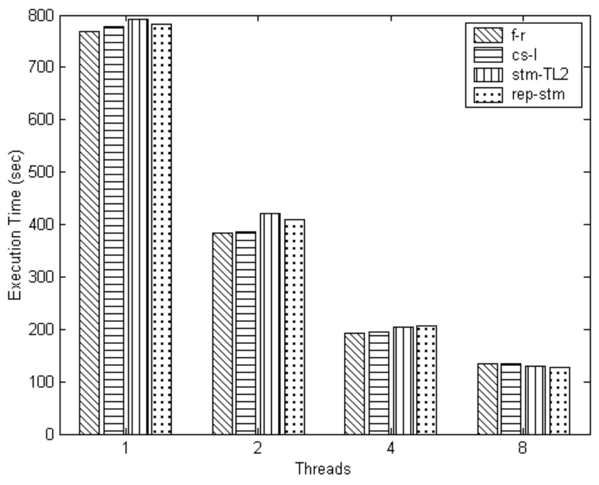

Figure 6. Results from PCA

So, most of the time is being spent by the STM techniques in maintaining and revalidating the read and write words. For every read, STM techniques need to execute a read barrier and revalidate its time-stamp during the commit phase. For every write, they are required to acquire lock, copy data and release the lock. Hence, the over all scalability for both `stm-tl2` and `rep-stm` are poor for K-Means.

Results from E-M: Since E-M is also a clustering algorithm, we used the same dataset as for K-Means. The number of clusters, k, was set to 60. We used 3-dimensional points. The tolerance for loglikelihood, ϵ, was set to 2. The results are presented in Figure 5. The results observed with the E-M algorithm are quite interesting when compared to the results from K-Means. Here, lock-based technique, `cs-l`, resulted in better performance. The speedups on 8 threads with `f-r`, `cs-l`, `stm-tl2`, and `rep-stm` techniques, when compared to the fastest single thread version, were 4.4, 4.95, 4.4, and 4.4, respectively. On one thread, the overhead of `cs-l` is extremely low, i.e., between 0-1%. But, compared to a single thread version, the overhead of `stm-tl2` and `rep-stm` are as high as 33.7%. The reasons for the overheads of STM techniques with K-Means holds good for the E-M algorithm also. But, the interesting result here is that, both `stm-tl2` and `rep-stm` have better relative speedup to 8 threads, over the other two techniques. This is because, E-M is compute-intensive, i.e., the amount of computation between updates for reduction object is very large. Thus, the overhead of locking is quite small, whereas the merge overhead of `f-r` is relatively high. Since, the computation is very high between updates, STM techniques also achieve good scalability. There is no observed difference between `stm-tl2` and `rep-stm`, as the number of conflicts and aborts stays very low even with 8 threads. Later in this section, we will further evaluate how the number of conflicts and aborts can grow with increasing number of threads.

Results from PCA: The third application that we used is PCA. The size of the dataset used for this application is 8.5 GB. The results are shown in Figure 6. The speedups of `f-r`, `cs-l`, `stm-tl2` and `rep-stm` on 8 threads are 5.73, 5.78, 5.96 and 6.1, respectively, when compared to the fastest single thread version. All four techniques have good scalability and are competitive to each other, though `stm-tl2` and `rep-stm` are slightly better than `f-r` and `cs-l`. This result is quite different from the previous two applications due to the nature of the application. PCA does large computations to calculate `mean` and `standard deviation` matrices, before updating the reduction object. So, the overheads caused by the lock handling in `cs-l` technique, and also the maintenance and revalidation of read and write sets in transactional memory, are amortized by the huge amount of computation. Specifically, the overheads of single thread version of `cs-l`, `stm-tl2` and `rep-stm` are as low as 1.4%, 2.3% and 2.2%, respectively. Again, in this application, `rep-stm` is not providing much improvement in performance when compared to the `stm-tl2` technique. This is because of very few conflicts and aborts with 8 threads.

6.2 Results from a Canonical Loop

Overall, the results we have presented in the previous subsection show that transactional memory is competitive with other techniques for two of the three applications. This result is interesting, since as we noted above, full replication cannot be expected to continue to scale with growing number of cores, and cache-sensitive locking implementation needs to be optimized for each cache configuration.

To further understand the relative performance of these

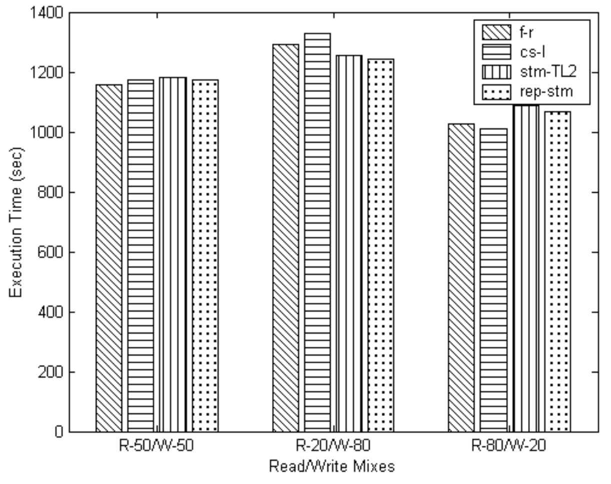

Figure 7. Results with 8 threads: Canonical Loop

techniques and to see the potential for scalability that `rep-stm` has, we conducted an additional set of experiments and analysis with a synthetic *canonical* loop.

Experiment setup: The canonical loop that we used is based on the generalized reduction structure shown in Figure 1. The loop can be parameterized with the amount of computation performed before updating the reduction object element, the size of the reduction object, and the number of reduction elements updated while processing each input element. We have experimented with a large number of combinations of parameters to understand the trade-offs between different techniques. For clarity reasons, we only present results from a set of values that correspond to an application that has low to intermediate computation, reasonably large reduction object and intermediate contention. We further consider different mixes of read and write operations over reduction elements.

Parallel Efficiency with Read-Write mixes: The first set of experiments we conducted evaluated the four techniques on the canonical loop, with different read-write mixes. The three different Read-Write mixes we use are Read dominated (read-80% and write-20%), Write dominated (read-20% and write-80%) and Balanced Read-Write (read-50% and write-50%) The second set of experiments were performed using up to 32 threads. These threads were executed on a machine with only 8 cores, but our focus is mainly on the number of conflicts and aborts with increasing number of threads, and not the additional speedups.

The performance comparison results for all SMP techniques with 8 threads is shown in Figure 7. When the mix is read dominated, `cs-l` seems to have the best performance. Since, most of the transactions are reads, there is not much contention and the overhead with lock handling in `cs-l` is very low. On the other hand, with `f-r`, since the size of reduction object is reasonably large, size of the reduction object after replication exceeds the cache size and the application suffers additional cache misses. Also, the merge overhead can be quite high. On the other hand, both STM techniques have performed worse than `cs-l` and `f-r`, with `rep-stm` being slightly better than `stm-tl2`. This is because of the high load of reads, and the maintenance or validation of read-sets in an STM. Comparatively, other techniques do not have overheads when a read is involved.

When the mix is write-dominated, the hybrid scheme, `rep-stm`, results in the best performance. The performance of `stm-tl2` is only slightly lower. But, on the contrary to what we saw for the read-dominated case, both `f-r` and `cs-l` have lower performance. With more writes, `cs-l` experiences higher overhead of locking handling. Again, `f-r` experiences more cache misses with the access to the reduction object. But, here, the advantage with the `rep-stm` is that it reduces the conflicts encountered by `stm-tl2`. At the same time, there are only two copies of the reduction object, compared to 8 with full replication. Thus, the reduction object fits into cache and there is only a small merge overhead.

Finally, when the read-write mix is balanced, `f-r` results in the best performance. In this case, `cs-l` has an increase in the overheads due to the lock handling and conflicts, as compared to its performance in the read-domination case. At the same time, an increasing number of reads reduces the advantage that transactional memory techniques have. But, `stm-tl2` seems to give the lowest performance, while, `rep-stm` is an improvement, being quite similar to `cs-l` and only slightly slower than `f-r`.

Evaluation of conflicts and Aborts: In the second experiment, we are comparing the rate of aborts only between `stm-tl2` and `rep-stm` for each Read-Write mix. As we have mentioned previously, growing number of conflicts and aborts can limit the scalability of transactional memory techniques, when the number of cores and threads are high. To demonstrate the benefit of using the hybrid replication-transactional memory scheme, these experiments report the number of aborts with increasing number of threads. The relative number of aborts experienced by `stm-tl2` and `rep-stm` are shown in Figure 8. The number of aborts are normalized with respect to the aborts from `stm-tl2` on 8 threads. These experiments were run with 8, 16, 24, and 32 threads. In our case, for `rep-stm`, there is a separate copy of transactional memory for every 4 threads.

From the results, we can clearly see that in each read-write mix, there is a significant reduction in the number of aborts with `rep-stm` when compared to `stm-tl2`. In case of Read-dominated mix, there is a drastic reduction of about 55% in the number of aborts with 32 threads. Similarly, with Write-dominated mix and Balanced mix there is

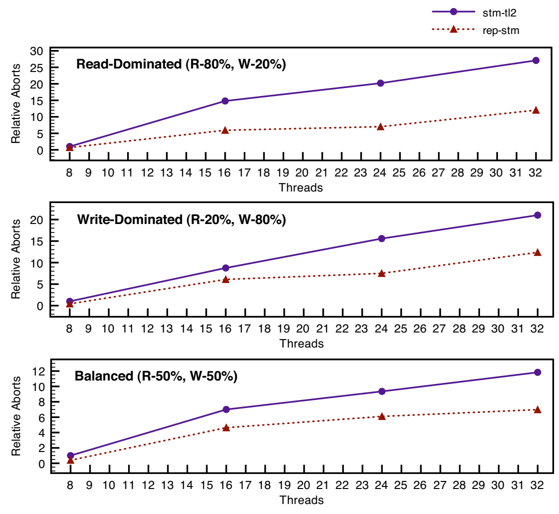

Figure 8. Canonical loop - Relative Aborts on different RW mixes

a substantial reduction of aborts up to 41% in both cases. From the Figure 8, in the Balanced mix, which is a average case, the growth of aborts in `rep-stm` is slower when compared to the growth of aborts in `stm-tl2` with the increase of threads. Thus, these results show that the growing number of aborts will reduce the scalability of existing transactional memory. At the same time, the hybrid replication-transactional memory scheme can improve on the scalability, with only a reasonable small increase in the memory requirements and merge costs.

6.3 Discussion

To summarize the results from the above three applications and the canonical loop, we have shown that both transactional memory techniques can be very competitive, with the exception of k-means clustering. This is an application with a very small reduction object, and high frequency of updates to the reduction object. The characteristics of an application is important for determining the relative performance of different techniques. For the canonical loop, the performance results with different read-write mixes show that different techniques performs well with different read-write mixes. But, in all the read-write mixes, `rep-stm` is better than `stm-tl2` by at least a small factor. This is attributed to taking advantage of both lower conflicts and relatively lower memory overhead. In the last set of experiments, between `stm-tl2` and `rep-stm`, replicated STM version results is significant improvement in terms of number of aborts. Thus, from the above results, we can expect `rep-stm` to be a promising technique for emerging multi-core and many-core architectures with a large number of cores.

7 Related Work

As we have stated, there is a large body of work on transactional memory techniques and their software, hardware, and hybrid implementations. Our main focus in this paper has been on utilizing and evaluating transactional memory techniques for a particular class of applications, which are the data-intensive or data mining applications. Most of the existing work has used kernels like threaded implementations of linked list, or Red-black tree and its variants [9, 8]. Delaunay triangulation application has been parallelized by the RSTM group [30]. While the work at stanford has used the k-means algorithm as one of the benchmarks [3], we are not aware of any work that has used EM or PCA, or has specifically considered a number of data mining applications. A multi-core runtime system, McRT [29], has integrated STM. But, again, target applications have been Linked list, AVL, and Binary search trees.

There is also ongoing work targetting parallelization of data-intensive applications. The *map-reduce* paradigm that Google has developed [6] has received a lot of attention. However, Map-Reduce does not include techniques that focus on shared memory parallelization. There are also some differences in the generalized reductions that FREERIDE supports and the map-reduce style of computations. Particularly, map-reduce requires sorting of reduction elements, which could be quite expensive for many data mining algorithms. Phoenix [27] is an implementation of MapReduce for shared-memory systems that includes a programming API and runtime system. FREERIDE and Phoenix differ significantly in their parallelization techniques, and there is no integration of transactional memory with Phoenix. The SALSA (Service Aggregated Linked Sequential Activities) effort [25] also targets data-intensive applications on multicores. The two frameworks differ significantly in the shared memory techniques. In SALSA, Concurrency and Computation Runtime (CCR) supports MPI style synchronization, while FREERIDE supports replication, lock-based approaches, and STM techniques.

8 Conclusions

This paper has focused on utilizing, optimizing, and evaluating transactional memory for data-intensive applications on multi-core systems. First, we have shown how transactional memory can be transparently used through a parallel data mining middleware, by integrating the Rochester Software Transactional Memory (RSTM) library with FREERIDE. Second, we have developed a new technique to help overcome the main impediment to scaling transactional memory to large number of cores, which is the increasing number of conflicts with larger number of threads. We have developed a hybrid scheme that combines transactional memory with replication.

We have evaluated Full-Replication, Cache-sensitive locking, integrated Rochester library (STM-TL2) and the hybrid scheme for three applications. Both STM techniques are comparable with other two techniques in two of the three applications. Moreover, with increasing number of threads, the hybrid scheme reduces the number of aborts by up to 55%.

Acknowledgements

This work was supported by NSF grants 0541058, 0619041, and 0833101. The equipment used for the experiments reported here was purchased under the grant 0403342. The authors thank Michael Scott and his research group at Rochester for making the RSTM libraries available for integration. Particularly, we thank Micheal Spear for helping us port TL2 library to a new architecture, and for his clarifications of API usage.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 316–327. Feb 2005.

[2] R. E. Bryant. Data-Intensive Supercomputing: The Case for DISC. Technical Report CMU-CS-07-128, School of Computer Science, Carnegie Mellon University, 2007.

[3] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007.

[4] M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the 20th Intl. Symposium on PPoPP*, 2006.

[5] L. Dalessandro, V. J. Marathe, M. F. Spear, and M. L. Scott. Capabilities and limitations of library-based software transactional memory in c++. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*. Portland, OR, Aug 2007.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[7] A. Dempster, N. Laird, and D. Rubin. Maximum Likelihood Estimation from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[9] D. Dice and N. Shavit. What really makes transactions faster? In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. Jun 2006.

[10] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proceedings of the Intl. Symposium on Code Generation and Optimization (CGO)*, 2007.

[11] P. Dubey. Teraflops for the masses:killer apps for tomorrow. In *Workshop on Edge Computing Using New Commodity Architectures, UNC 23 May 2006*, 2006.

[12] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[13] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004.

[14] T. Harris and K. Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402. Oct 2003.

[15] M. Herlihy, J. Eliot, and B. Moss. Transactional memory: architectural support for lock-free data structures. In *in Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[16] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[17] R. Jin and G. Agrawal. A middleware for developing parallel data mining implementations. In *Proceedings of the first SIAM conference on Data Mining*, Apr. 2001.

[18] R. Jin and G. Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. In *Proceedings of the second SIAM conference on Data Mining*, Apr. 2002.

[19] R. Jin and G. Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2005.

[20] V. J. Marathe, W. N. S. III, and M. L. Scott. Adaptive software transactional memory. In *Proc. of the 19th Intl. Symp. on Distributed Computing*, pages 354–368, 2005.

[21] V. J. Marathe and M. L. Scott. A qualitative survey of modern software transactional memory systems. Technical Report TR 839, University of Rochester Computer Science Dept., Jun 2004.

[22] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. Technical Report TR 893, Computer Science Department, University of Rochester, Mar 2006.

[23] M. Moir. Hybridtm: Integrating hardware and software transactional memory. Technical Report Archivist 2004-0661, Sun Microsystems Research, August 2004.

[24] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. Feb 2006.

[25] X. Qiu, G. Fox, H. Yuan, S.-H. Bae, G. Chrysanthakopoulos, and H. Nielsen. Parallel datamining on multicore clusters. In *Seventh International Conference on Grid and Cooperative Computing, 2008. GCC '08*, pages 41–49, 2008.

[26] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505. IEEE Computer Society, Jun 2005.

[27] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of International Symposium on High Performance Computer Architecture, 2007*, pages 13–24, 2007.

[28] V. T. Ravi and G. Agrawal. Performance Issues in Parallelizing Data-Intensive Applications on a Multi-core Cluster. In *In proceedings of Conference on Clustering Computing and Grids (CCGRID)*, 2009.

[29] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proc. 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '06)*, pages 187–197. Mar 2006.

[30] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay triangulation with transactions and barriers. In *PROC of the 2007 IEEE INTL Symposium on Workload Characterization*. Boston, MA, Sep 2007.

[31] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, August 2008.

[32] N. Shavit and D. Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[33] A. Shriraman, M. F. Spear, H. Hossain, V. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34rd Annual International Symposium on Computer Architecture*. Jun 2007.