Variable Granularity Access Tracking Scheme for Improving the Performance of Software Transactional Memory

Sandya S.Mannarswamy
CSA, IISc and Hewlett Packard
Bangalore,India
sandya@hp.com

Ramaswamy Govindarajan
SERC, Indian Institute of Science,
Bangalore, India
govind@serc.iisc.ernet.in

Abstract— *Software transactional memory (STM) has been proposed as a promising programming paradigm for shared memory multi-threaded programs as an alternative to conventional lock based synchronization primitives. Typical STM implementations employ a conflict detection scheme, which works with uniform access granularity, tracking shared data accesses either at word/cache line or at object level. It is well known that a single fixed access tracking granularity cannot meet the conflicting goals of reducing false conflicts without impacting concurrency adversely. A fine grained granularity while improving concurrency can have an adverse impact on performance due to lock aliasing, lock validation overheads, and additional cache pressure. On the other hand, a coarse grained granularity can impact performance due to reduced concurrency. Thus, in general, a fixed or uniform granularity access tracking (UGAT) scheme is application-unaware and rarely matches the access patterns of individual application or parts of an application, leading to sub-optimal performance for different parts of the application(s). In order to mitigate the disadvantages associated with UGAT scheme, we propose a Variable Granularity Access Tracking (VGAT) scheme in this paper. We propose a compiler based approach wherein the compiler uses inter-procedural whole program static analysis to select the access tracking granularity for different shared data structures of the application based on the application's data access pattern. We describe our prototype VGAT scheme, using TL2 as our STM implementation. Our experimental results reveal that VGAT-STM scheme can improve the application performance of STAMP benchmarks from 1.87% to up to 21.2%.*

Keywords-compiler, software transactional memory

1. Introduction

An atomic section is a programmer-specified region of source code that executes atomically (other concurrent code sees either none or all of the updates it makes to program state) and in isolation from other concurrent code. Replacing locks by atomic sections relieves the programmer of the cumbersome task of identifying particular locks to protect particular data structures. The atomic section is an abstraction, likely to be a programming language construct. Atomic sections simplify the task of writing concurrent software since programmers can specify the code region which needs to execute atomically by simply enclosing the code region with the keyword 'atomic'.

One way of supporting atomic sections is through transactional memory [1, 2, 20, and 21]. Transactional Memory (TM) can be implemented either in hardware or in software or a combination of the two. In order to be widely adopted, a TM system must support transactions of unbounded size and duration, and allow transactions to be integrated with a language environment [12]. Since Software Transactional Memory (STM) helps to achieve these objectives, there has been considerable interest in developing high performance implementations of STMs. STM implementations can be broadly classified as: *lock*-based and *obstruction-free*. Lock-based STMs typically employ a variant of the two-phase locking protocol [30]. Obstruction-free STMs [2] do not use any blocking synchronization mechanisms (such as locks), and guarantee progress even when some of the transactions are delayed. Lock based STM implementations [4, 17, 18, 29] have been shown to have lesser validation overhead and hence exhibit better performance than non-blocking ones. Therefore we focus our attention to lock based STMs in this paper.

STMs allow for optimistic execution by permitting multiple atomic sections to run concurrently assuming they will not conflict. However, in case a conflict does occur, STMs employ a mechanism to detect and recover from such conflicts. Most STMs employ the single-writer-multiple-readers strategy. Two concurrent transactions conflict when they access the same location and at least one of the accesses is a write (update). In order to commit, a transaction must eventually *acquire write locks for* every memory location that is written by it. Locks can be acquired eagerly, i.e., at the time of the first update operation by the transaction on the memory location, or *lazily*, i.e., when the transaction is about to commit. Reads to shared data can either be visible or invisible [20] to other transactions accessing the same data. In an STM which supports invisible reads, a transaction reading a shared datum x needs to detect any possible conflicts on x with other transactions that write x concurrently, i.e., validating its read set.

To enable conflict detection, STMs associate metadata/locks for shared data accessed inside an atomic section. Conflict detection consists of examining the metadata/locks corresponding to the shared reads and writes to check if there has been a read-write or write-write conflict. Each lock typically contains a bit indicating whether the shared data associated with the given lock has been locked, and the remaining bits are used to represent a version number for the associated shared data. STMs typically support a global fixed size lock table and shared data is mapped to the lock table elements using a hash function [4, 17 and 18].

The granularity at which shared data is mapped to the metadata/locks is the access tracking granularity for an STM implementation. This is basically the number of consecutive words of shared data that map to a single lock entry in the STM's global lock table and is the basic unit of tracking shared data items by the STM. Shared data items which fall within a single unit of access tracking granularity (hereafter referred to as unit granularity region) is treated as a single entity by the STM for conflict detection and validation since a single lock covers all the data items lying within a unit granularity region.

Access tracking granularity has a significant impact on the performance of STM [7, 13]. These impacts can be classified under 3 categories namely

(a) Impact on false conflicts and hence on concurrency
(b) Impact on cache
(c) Impact on read set validation costs and lock acquire/release costs

A false conflict occurs in an STM when two different addresses are mapped to the same metadata or lock and this results in two transactions which are accessing truly disjoint data, getting falsely diagnosed as conflicting when they are not. Such false conflicts result in increased number of aborts/rollbacks and hence can impact the execution time [3, 7 and 13]. Based on how false conflicts occur, they are classified into two types, namely intra-conflicts and inter-conflicts. Intra-conflicts occur wherein two different data items getting accessed in independent transactions are mapped to the same lock due to both of them falling within the same unit granularity region which gets mapped to a single lock. Intra-conflicts occur when concurrent transactions access data which have high inter-transaction data locality.

Inter-conflicts occur wherein two different data items getting accessed in independent transactions are mapped to the same lock, even when they are not in the same unit granularity region. This happens due to the limited size of the lock table. Since the number of locks is often fewer than the numbers of shared data (grouped at a fixed access tracking granularity), multiple uncorrelated data items can get mapped to the same lock. This is known as lock-aliasing. When such uncorrelated data items which have been mapped to the same lock are accessed in concurrently executing transaction, they cause false conflicts between the transactions, which we term as inter-conflicts. The terms intra and inter refer to the fact that the data accesses are within the same memory region (of size equal to the access tracking granularity) or across two different memory regions.

Access tracking granularity also has a significant impact on the cache performance of STM applications. Use of fine grained access tracking granularity also leads to more number of metadata/lock accesses in a transaction. This can impact the performance adversely for transactions which touch a large volume of data. Every unit granularity memory region of shared data accessed in a transaction results in a corresponding programmer invisible metadata/lock access corresponding to that shared data. Smaller the access tracking granularity, higher is the number of lock accesses for the volume of shared data accessed in a transaction. While on one hand, fine grained locking granularity improves concurrency, but it also increases the pressure on the cache. As we see in Section 2.4, the lock accesses are a significant contributor to the D-cache misses in STM applications and hence impact performance.

The granularity at which accesses are tracked and conflicts are detected also has an impact on the read-validation costs and write-set lock acquire costs. Read-set validation needs to validate each shared data read by tracking their consistency at the level of unit granularity. Similarly a transaction needs to acquire locks for each data item in the write-set at the level of unit granularity. Smaller the access tracking granularity, higher is the number of locks/metadata associated with a given volume of shared data accessed/updated in a given transaction, hence greater is the number of validation operations and lock acquire/release operations. This in turn increases the total lock operation costs and validation costs of the STM.

The three factors discussed above have conflicting requirements with respect to the access tracking granularity. Reducing the access tracking granularity reduces the intra false conflicts, while it can end up increasing the number of lock accesses. This in turn can end up increasing the inter-false conflicts, the cache pressure due to lock accesses, read-set validation cost and the write-set locking cost. On the other hand, increasing/coarsening the access tracking granularity can reduce the number of lock accesses in given transaction, and can reduce the inter-conflicts, cache pressure due to lock accesses, and read-validation and write-locking costs, but it can increase the intra-conflicts leading to a reduction in concurrency, and hence STM performance. Quantification of the impact of these factors on STM performance would depend on the shared data access patterns of the STM application.

Hence the selection of access tracking granularity that results in higher performance needs to factor in these complex and conflicting requirements, while taking into account the data access patterns of a given STM application. However most of the current STM implementations use a fixed size uniform access tracking granularity [4, 17 and

18]. TL2 [4] supports two fixed access tracking granularity schemes namely 'Per Stripe' (PS scheme) and a 'Per Object' (PO Scheme). Either of these two schemes can be chosen by the programmer when compiling the application, by enabling certain flags. Shavit [4] found that PS scheme performed better than PO scheme for a given set of benchmarks and hence TL2 supports PS scheme with word size as the access tracking granularity by default. Dragojević [18] evaluated the impact of the locking granularity on STM performance. They found that a lock granularity of four words performed better than both word-level and cache line-level locking by 4% and 5% respectively across the set of benchmarks they studied, demonstrating that the access tracking granularity which is best for an application, is highly application/application region dependent. To the best of our knowledge, we are not aware of any lock based STM implementation which supports different granularity of access tracking for different data structure types of the same application.

There has been considerable prior work on compile time lock allocation schemes for atomic sections. Compiler analysis to improve allocate minimum number of locks to atomic sections has been studied in [22, 23, 24]. But these approaches require highly sophisticated alias analysis information which makes their applicability restricted.

In order to overcome the disadvantages associated with uniform granularity access tracking (UGAT), we propose a Variable Granularity Access Tracking (VGAT) scheme. Our scheme uses static analysis by the compiler to select the access tracking granularity for different shared data structures of the application based on the application's data access patterns.

We have implemented a prototype of our VGAT scheme in Open64 compiler [11] with TL2 [4] as our underlying STM for our experimental evaluation. Performance results on a set of STAMP benchmark programs [6] indicate that our VGAT scheme performs better than the baseline STM's UGAT scheme. Our VGAT scheme improves application performance in 6 of the STAMP [6] benchmarks from 1.87% to 21.2% over the base STM implementation.

This paper is organized as follows: In Section 2, we provide the necessary motivation for our VGAT-STM scheme. Section 3 describes our VGAT scheme. We report the results of our experimental evaluation in Section 4. We discuss related work in Section 5 and conclude with a short summary in Section 6.

2. Motivation

This section motivates the need for a Variable Granularity Access Tracking (VGAT) scheme. Since applications contain atomic sections with different characteristics, some with fine-grained parallelism and some with coarse-grained parallelism, we demonstrate that a uniform access tracking granularity does not serve best all types of atomic sections. Even in a single application, different code regions can be best served by different access tracking granularities. We illustrate this next with a couple of examples.

2.1. Motivating Example 1

Figure 1 Code Snippet contains two code regions (Example 1a and Example 1b), simplified and modified from the STAMP benchmark program *Ssca2*. The code snippet in Figure 1 Example 1a accesses the shared data array '*GPr->indegree[]*' using the index '*v*' which is obtained from the adjacency vertex list of the inner loop index '*j*'. The accesses to '*GPtr->inDegree []*' are non-contiguous and hence this region benefits from the word level access tracking granularity supported by the TL2 STM.

On the other hand, consider the code snippet in Figure 1 leading to cache pressure and lock acquire/release overheads. This can be seen in the small code snippet in Figure 2 Example 2. In the code snippet shown, each thread operates on a disjoint portion of the array *'centers'*. Consider an STM which uses a access tracking granularity of a word and a lock table size of 1000. Each thread will need to make 1000 lock accesses per transaction. This leads to cache pressure as well as to an increased lock acquire/release overhead costs. And also the 10000 array elements will map to the 1000 locks resulting in lock aliasing and inter-conflicts.

```
for (i = i_start; i < i_stop; i++) {
   for (j = GPtr->outVertexIndex[i];.. ; j++) {
        int v = GPtr->outVertexList[j];

        TM_BEGIN()

        long inDegree = GPtr->inDegree[v];
        GPtr->inDegree[v] = (inDegree + 1);

       TM_END();
  }
}//end for
```

Figure 1 Example 1a

```
// 'thread_section_start' and 'thread_section_stop'
// define independent  array sections for each thread

TM_BEGIN

for  (i = thread_section_start;
      i < thread_section_stop; i++)
{
        GPtr->outDegree[i] = my_outdegree[i];
}

TM_END
```

Figure 1 Example 1b

```
#define ArraySectionSize 1000
int centers[10000];

start()  {
  for (int tid = 0; tid < 10; tid++)  {
    int start  = tid * ArraySectionSize;
    pthread_create(perform, &centers[start], 1000);
  }
}

TMBEGIN
perform(int* mycenters, int size) {
  for (int j = 0; i < size; j++) {
        *(mycenters + j) = ....
  }
}
TMEND
```

Figure 2 Example 2 code snippet

Though an increased lock table size will avoid inter-false conflicts, the cache pressure and lock acquire/release overheads due to fine grained locking (which is not actually required in this case) still impact performance adversely. On the other hand an STM scheme which supports variable granularity access tracking can use a coarser tracking granularity of 1000 for this array, requiring only 1 lock access per atomic section. This leads to reduced read-set validation and write-set lock acquire costs, as well as reduced cache pressure for this atomic section, without having an adverse impact on concurrency.

2.2. Issues with Object-Level Granularity

Next we consider whether using object level access tracking granularity can help mitigate some of the issues we have discussed above. Consider the code snippet in Figure 3 Example 3. In an STM with object level access tracking granularity, each 'myobject' object is mapped to a lock. However assigning such object-level lock prevents the two atomic sections in Figure 3 Example 3 from being executed concurrently even though each of the atomic sections operate on different sets of fields of 'myobject' thereby leading to loss of concurrency.

```
Atomic  {
  myobject->field1 = t1;
  myobject->field2 =...;
}
```

```
Atomic  {
  myobject->field3 = t2;
  myobject->field4 = ...;
}
```

Figure 3 Example 3

BM	%execution cycles incurred on Dcache Misses	% of D-cache Miss Cycles incurred on Lock Accesses	% execution cycles incurred on dcache misses during Lock accesses
Kmeans	84.12%	51.37%	43.21%
Genome	56.98%	50.63%	28.84%
Vacation	56.85%	54.07%	30.73%
Intruder	54.25%	55.12%	29.91%
Ssca2	53.58%	34.05%	18.24%
Yada	51.58%	44.02%	22.70%

Table-1 – Impact of Lock Accesses On Data Cache Misses

On the other hand, a word level granularity will enable the two atomic sections in Figure 3 Example 3 to execute concurrently with a lock word being accessed for each shared data word access, requiring 2 lock accesses for each transaction. A variable access tracking granularity scheme which tracks '*field1'* and '*field2*' together as a single entity and *'field3'* and '*field4'* together as a single entity respectively (assuming that there exist no other atomic section in the applications where the fields of '*myobject*' are accessed in a different pattern) will reduce the number of lock accesses without reducing the concurrency.

2.3. Impact of Fine Grained Locking granularity on Cache Performance of STM

Another major impact of fine grained locking is the impact on data cache (D-cache) performance of STM applications. Column 2 of Table 1 shows the percentage of execution cycles due to D-cache miss stall cycles experienced on TL2 STM implementation [4] for a set of STAMP applications [6] run with 32 threads on a 32 core IA-64 server BL-890c, with 8 Intel Itanium Processors 9350 with 4 cores per socket, and a total of 127.71GB of physical memory. Each core operates at 1.73 GHz and has 6 MB non-shared L3 cache per core. We see from Column 2 that more than 50% of total execution cycles in all the six benchmarks are spent on data cache misses. Column 3 gives the % of D-cache miss cycles which occur due to lock accesses and Column 4 gives the % of execution cycles due to data cache misses from lock accesses. The D-cache miss data was obtained using the performance profile tool caliper [16] from the IA-64 hardware Performance Monitoring Unit counters.

Table-1 shows that a significant percentage of the total execution cycles were on data cache misses that are due to STM lock accesses in each of the benchmarks. This shows that the word level granularity used by TL2, though improving concurrency, increases the cache pressure. This in turn can adversely affect overall performance. Also applications contain many regions which do not require such fine grained locking, wherein using a fixed fine grained granularity uniformly over the entire application results in paying the cache performance penalty over the whole application except where it may be actually justified for achieving fine grained parallelism.

BM	Granularity (in bytes) at which best performance occurs	Performance Variation with Different granularity levels
Bayes	4	-4.75% to 0%
Kmeans	8	-11.32% to 27.81%
Genome	16	-7.86% to 15.32%
Intruder	64	-14.73% to 22.38%
labyrinth	32	-7.32% to 3.21%
Ssca2	8	-8.21% to 5.93%
Vacation	32	-18.13% to 24.21%
Yada	16	-11.34% to 4.76%

Table 2 – Performance Potential for varying the granularity

2.4. Performance Potential for Variable Granularity in STM

From the various code snippets illustrated above, we can see that a fixed uniform access tracking granularity can not serve all applications or even different parts of the same application equally well. Shared data structures in atomic sections of an STM application have different data access patterns. Such data structures are better served by varying the access tracking granularity of the STM.

In order to understand the potential for a variable granularity access tracking scheme for STM, we measured the execution time for different values of access tracking granularity for a set of STAMP benchmarks using TL2 STM implementation with 32 threads on 32 core IA-64 server. TL2 has the default granularity equal to a word, and in our experiments, we compared the performance with granularity being 8, 16, 32, 64, and 128 bytes respectively to the baseline. We report the optimal access tracking granularity for each of the benchmark (at which the reduction in execution time is maximum compared to the baseline) in Column 2 of Table-2 and we report the performance variation compared to the baseline with the varying access tracking granularity in Column 3 of Table-2. We find that the best performance improvements occur at different access tracking granularity for different applications and different access tracking granularities have a varying impact on the performance for each of the benchmarks. This shows that there is considerable potential for using a variable granularity access tracking scheme for STMs.

3. Variable Granularity Access Tracking Scheme

In this section, we describe our compile time variable granularity access tracking scheme (VGAT) for STM, which uses compiler analysis to determine the appropriate access tracking granularity for each data structure in the application.

3.1. Overview of our VGAT scheme

Our Variable Granularity Access Tracking scheme (VGAT) consists of the following two major steps:

1. Selection of candidate data structure types on which VGAT scheme can be applied. The candidate selection step consists of two parts, namely:
   a) Legality checks to determine whether a data structure is eligible for VGAT transformation.
   b) Determination of the appropriate access tracking granularity for the fields of the shared data structure.
2. Code Transformations by the compiler to modify the shared data references of the selected VGAT candidate types to use the appropriate access tracking granularity determined for that data structure by the VGAT analysis. The compiler also needs to communicate the modified access tracking granularity to the STM so that this can be used in place of the default STM's access tracking granularity. Since the access tracking granularity determines the granularity of the locks in an STM, varying the access tracking granularity also changes the mapping between shared data addresses to the STM assigned locks for the shared data.

3.2. Candidate Selection for VGAT scheme

Initially we consider all shared data structure types as candidates for VGAT scheme. We then apply legality checks on the initial candidate sets to identify those which are eligible for VGAT transformation. We explain next why we need these legality checks and the type of checks used.

3.2.1. Legality Checks for VGAT candidates

When the VGAT scheme changes the lock mapping between a shared data reference and its meta-data, in order to ensure a uniform locking discipline throughout the application, the compiler must be able to update the modified locking information for all the shared data references of a selected VGAT candidate. If this cannot be done, then the candidate data structure cannot be chosen for VGAT. For example, if an instance of the VGAT candidate type escapes to an external function call, whose source code is not visible to the compiler, the compiler cannot apply VGAT to that type since it cannot modify the affected references in the external opaque function. A similar issue occurs if a VGAT candidate data type is cast to a different type, since there is ambiguity as to what would be the correct access tracking granularity that should be applied. Full fledged field sensitive points-to analysis information if available, can help ensure which types are safe to transform for VGAT. However such sophisticated 'points-to' analysis information is expensive to compute and may not be available readily. Hence we adopt a more conservative but less expensive approach of applying a set of simple and practical tests to check whether a type is VGAT eligible.

Hence our set of legality checks includes the following:

a. **Dangerous type casts:** This test looks for a cast to a candidate type or a cast from a candidate type. This indicates type unsafe use of a VGAT candidate type and hence such types are marked ineligible for VGAT. Note that in C/C++ programs, if there are

dynamic allocations for a candidate type data, then a cast from a (void *) to the result type will be found, as *malloc* & *calloc* return (void *). We maintain a list of such special cases and tolerate the casts found in such cases.

b. **Type escaping to an external opaque function:** If the VGAT candidate type escapes to an external function not visible during the compiler's whole program analysis and if the function is not a standard library function whose semantics is known, then that type is marked as ineligible for VGAT.
c. **Address Arithmetic on the pointer to VGAT type**: If the compiler detects address arithmetic operations on the pointers to VGAT types, it marks the type as ineligible.

Other than the ones mentioned above, our legality checks also include a few other corner case checks which are similar to those applied for any data layout transformation eligibility [28].

Our legality check implementation is split into 2 phases, with type cast and pointer arithmetic checks being done in the optimizer front end summary phase while the type escape checks are being done in the inter-procedural whole program analysis of the compiler. Also our static VGAT scheme requires whole program analysis. As mentioned earlier, our legality check is conservative and can result in certain candidates not being selected for VGAT scheme when they are actually safe. This limitation has no impact on the correctness of our approach, although it may leave some performance on the table. We plan to study using sophisticated points to analysis for determining VGAT eligible candidates as part of future work.

3.2.2. Determining Access Tracking Granularity

Once a shared data structure type is marked as eligible for VGAT by the legality analysis, our compiler then determines the access tracking granularity for that data structure. Our approach tries to improve the access tracking granularity based on the data access patterns of the shared data structure with the objective of reducing the number of metadata (lock) accesses associated with it, but without adversely impacting the concurrency. For our approach, we consider the basic access tracking granularity to be a word, which is the default for our underlying TL2 STM implementation. Our approach attempts to determine the most profitable access tracking granularity for each of the data structure of the application being compiled. We first explain certain terms used in our profitability analysis.

We use the term 'Access Group' to denote the set of fields of a data structure D. An Access Group is tracked as a single entity by the STM's access tracking scheme. In the baseline TL2 STM implementation, each 'Access Group' represents a word of memory and a single data structure is tracked using multiple Access Groups, each word sized. In an object based STM, each 'Access Group' represents an object; On the other hand, in our VGAT scheme, the number and size of Access Groups for tracking a given data structure is a variable, determined by our compiler analysis. Given a shared data structure SD, the goal of our VGAT scheme is to partition the set of fields of SD into a set of Access Groups AG (SD) so that each Access Group can be tracked as a single entity by the STM algorithm.

Recall our discussion in Section 1 which describes the impact of access tracking granularity on STM performance. These factors are: false conflicts (intra, inter), cache pressure, read-set validation and write-set locking costs. We term the performance impacts arising due to these factors for a given access tracking granularity as:

1. false conflicts cost, which represents the performance lost due to falsely conflicting transactions;
2. cache cost which represents the performance impact arising from the increased cache pressure due to lock accesses; and
3. Book-keeping cost which represents the performance impact arising from the read-set validation costs and write-set lock acquire tests.

Thus, the overall performance impact is given by the sum of False Conflicts Cost, Cache Cost and Book-keeping cost.

An ideal model which attempts to decide whether two fields *'f1'* and *'f2'* should be placed in the same access group should determine the impact of that decision on each of these costs and make the choice to place the two fields in the same access group if it will lead to an overall improvement in the performance. Note that a change in granularity can have a positive or negative impact on each of the above costs. For instance an increase in granularity can decrease the cache costs and book keeping costs due to a reduction in number of lock accesses per transaction whereas it can increase the false conflict costs due to the increased intra-conflicts.

The construction of an ideal model which computes these costs and uses them to drive the partitioning of fields into access groups poses many challenges. Each of these quantities are not directly and accurately computable. While it is possible to use extensive instrumentation to compute these quantities, the instrumentation itself can perturb the execution, impacting the measured quantities. Therefore we approximately model these costs in our approach as explained below.

Recall that a change in access tracking granularity results in a change in the number of lock accesses needed to track a given set of shared data items in an atomic section. Change in the number of lock accesses in turn impacts the inter-conflicts, cache costs and book keeping costs. Granularity change also impacts false conflicts and hence the concurrency. Therefore we approximate our model by considering only the change in the number of lock accesses and the potential intra-conflicts introduced by the change in the granularity.

Our current model does not include the effects of inter-conflicts. Therefore, the partitioning of the fields in to

Access Groups in our simplified model is driven by the following criteria:

(i) The Access Groups determined by VGAT scheme results in a reduction in the number of lock accesses compared to the baseline Access Tracking Granularity for that data structure over the entire application.

(ii) There is no reduction in the application concurrency due to intra-conflicts.

Criterion (i) is modeled in our cost-benefit calculations as the reduction in the number of execution cycles due to reduction in number of lock accesses by the proposed VGAT scheme. Hence this quantity is referred to as *VGAT_Gain*. If a given VGAT scheme results in loss of concurrency, then this will be modeled as an increase in the number of execution cycles due to loss of application concurrency. Hence criterion (ii) is modeled using a *VGAT_Loss* metric which captures the increase in execution cycles due to a given VGAT scheme. Therefore the access tracking granularity selected by a specific VGAT scheme is beneficial if the difference between (*VGAT_Gain* – *VGAT_Loss*), which represents the Net Gain, is positive.

We represent the *VGAT_Gain* and *VGAT_Loss* effects on the fields of a VGAT candidate type by a Field Access Graph (FAG). The nodes of the Field Access Graph represent the fields of the VGAT candidate type. FAG is a weighted undirected graph. Weight of an edge between two fields f1 and f2 indicates the VGAT net gain which is (*VGAT_Gain – VGAT_Loss*) and represents the benefits of placing f1 and f2 together in the same access group. The goal is to determine the partition of the FAG into different access groups so that the net gain is maximized. This can be realized by using graph partitioning algorithms that maximize the intra-cluster edge weights and minimize inter-cluster edge weights.

We use the following approximation to compute the *VGAT_Gain* as:

$$VGAT\_Gain(f1, f2) = \sum_{i \epsilon I} ExecutionFreq(ASi) \times k$$

where Execution Frequency (ASi) is the execution frequency in atomic section i∈ I where I is the set of all atomic sections in which f1 and f2 are accessed together. 'k' denotes the average lock access cost. We compute the *VGAT_Loss* as:

$$VGAT\_Loss(f1, f2) = \sum_{j \epsilon J} ExecutionFreq(ASj) \times k'$$

where ExecutionFrequency(ASj) is the execution frequency in atomic section j∈ J where J is the set of all concurrent atomic sections such that j contains the access of f1 but not f2 or the access of f2 but not f1. If f1 and f2 were not grouped together, then two independent transactions which access f1 and f2 respectively can execute without conflict, whereas grouping f1 and f2 together results in them conflicting leading to one of them getting aborted. k` denotes the average transaction abort cost for a given atomic section.

Once we construct the FAG, we compute the partition of the FAG into Access Groups using a simple graph clustering algorithm. Initially all nodes of the FAG are in their own individual Access Group. The nodes of the FAG are sorted by their execution frequencies as obtained from the execution profile information. The mostly frequently accessed node among the unassigned nodes is picked next and placed in a new cluster. FAG construction proceeds by adding the next node to the current cluster which maximizes the sum weights within a cluster. A cluster is grown until we end up with all unassigned nodes with negative edge weights to the nodes already in the cluster. Once a cluster is grown to the maximum possible, we mark it as complete and mark the fields of the cluster as belonging to the same Access Group. Then, the process is repeated until all the fields are assigned to an access group.

Each Access Group is selected as a basic access tracking entity for the STM operations by the compiler. Therefore the size of the each access group is the access tracking granularity for the fields belonging to that access group. Compiler uses the access group information so determined to establish the lock mapping between the shared data references as we discuss in the next section.

3.3. Communication of Access Tracking Granularity to STM

Our VGAT scheme has been implemented in the Open64 compiler [11] using TL2 [4] as our baseline STM. We built our legality checking and access group determination phase leveraging the Open64 compiler's structure field layout analysis and optimization phase [11, 28] which performs field access identification and computing execution frequencies of field accesses inside basic blocks. Before we discuss how the VGAT access tracking granularity is communicated to the STM, we provide a brief overview of TL2.

3.3.1. TL2 Basics

TL2 is a lock based based STM with a word based granularity. It supports invisible readers and uses a global timestamp based validation scheme. It uses write locks to protect shared memory locations. A table of lock words is allocated by TL2. The default size of the global lock table is 2^{20} entries. Note that default TL2 implementation uses a single lock table with a fixed uniform access tracking granularity equal to word size. TL2 STM uses a hash function of the memory address to map a given shared memory location to its associated lock word.

3.3.2. Code Transformations and STM Modifications

To support the VGAT scheme, our compiler needs to allocate the locks to satisfy the selected access tracking granularity and communicate the mapping between the shared data address and the address of the compiler created

lock to the STM. Instead of a single lock table, we use multiple lock tables, with one lock table being associated with each VGAT candidate. This allows us to specify different access tracking granularity for each of the different data structure types. We use the type id of the VGAT candidate to select the associated lock table and given the address of the field of a shared data structure in a shared data reference, we compute the base address of the structure and map this to an address in the associated lock table for that structure. We use the field id corresponding to the shared data reference to determine the access group it belongs and hence the corresponding lock mapping.

Compiler also transforms the shared data references for the VGAT candidate data types so that the compiler specified lock mapping is used instead of the default STM lock mapping. Compiler transforms all transactional references (TxLoad/TxStore) for each of the shared data whose types are selected for the VGAT scheme by calls to new STM interfaces `VGATLoadWithLock` and `VGATStoreWithLock` to which the compiler specified lock mapping is passed as an extra argument. `VGATLoadWithLock` and `VGATStoreWithLock` are two new interfaces added to the VGAT STM implementation. These are exactly identical to TxLoad and TxStore interfaces in their functionality except that the lock address is passed by the compiler as an extra parameter and STM uses this lock for the transactional access instead obtaining the lock from the default word granularity lock table.

3.4. Overheads of our approach

VGAT has following sources of overheads on the application's compile time and space compared to an unmodified STM implementation. Compile time overheads are due to the VGAT whole-program analysis phase. We report the compile time overheads due to our approach in Section 4.3. Space overheads can arise as we allocate multiple lock tables of smaller sizes compared to a single lock table of a large size. While the default TL2 STM uses a single lock table of 2^{20} entries, we used multiple lock tables of size 2^{14} entries and our selected VGAT candidates in all our experimental benchmarks were less than 32. Therefore we did not observe any major increase in memory requirements for the modified implementation. However this can be an issue with other applications and should be factored into account while building a production quality VGAT STM.

3.5. Establishing the correctness of VGAT scheme

We need to ensure that our VGAT scheme does not violate the original STM semantics of the atomic sections. Hence we need to ensure that:

a) There is a locking discipline which associates each shared data item accessed within a transaction to a lock, and
b) The locking discipline is consistent.

Both these conditions are not violated by VGAT. The only change VGAT incorporates is in the lock mapping. VGAT assigns a lock to each shared datum whose type is selected as a VGAT candidate and other shared data whose types are not VGAT candidates are covered by the default STM lock mapping. Hence the first condition is clearly satisfied. Since VGAT legality checks and transformation ensures that all the shared data references to VGAT candidates are modified appropriately throughout the entire application, the second condition is also satisfied for all VGAT candidates. For shared data references to non-VGAT candidate types, there is no change in original STM behavior.

4. Experimental Evaluation

4.1. Experimental Methodology

To evaluate the effectiveness of our VGAT scheme, we used the STAMP benchmark suite [6] version 0.9.10. We implemented the VGAT scheme prototype in the Open64 C/C++ compiler [11]. For the performance runs, the benchmarks, compiled with the VGAT scheme enabled at optimization level 3 with inter-procedural analysis enabled, were run on the modified VGAT TL2 implementation (as described in Section 3.3.2). For our baseline, we compiled the benchmarks at the same optimization level (level 3) with inter-procedural analysis enabled, but with VGAT phase turned off and run the executable on the unmodified TL2 implementation [4]. The native input sets of the STAMP benchmarks were used for our performance evaluation. We used a 32 core IA-64 BL-890c, with 8 Intel Itanium Processors 9350 with 4 cores per socket, and a total of 127.71GB of physical memory. Each core operates at 1.73 GHz and has 6 MB non-shared L3 cache per core. The latencies of L1, L2, and L3 caches are 1, 5 and 20 cycles respectively. Cell local cache to cache (c2c) latency, cell local memory latency, one-hop remote memory latency, 2 hop remote memory and 1 or 2 hop remote c2c access times are 150, 300, 650, 750, 750+ cycles respectively.

Access groups determination in VGAT scheme described in Section 3.3, requires the execution frequencies of the atomic sections, which are obtained from the application's profile data. It is possible to use either dynamic profile or static profile data [27]. For our experiments, we used the dynamic profile data with the profile collection runs being performed using STAMP simulator input data sets (which are much smaller than the native inputs).

Determining the access tracking granularity also requires the computation of average lock access cost k and the average transaction abort cost k'. The values of average transaction abort cost and average lock access cost for each atomic section can be empirically determined using instrumentation of atomic sections during profiling runs. However for our current experiments, we adopt the simplistic approach of using a fixed value of unit lock access cost and a uniform average transaction abort cost per

Benchmark	VGAT candidates found
Kmeans	2
Vacation	6
Genome	4
Intruder	7
Labyrinth	4
Ssca2	6
Yada	8
Bayes	3

Table 3 – VGAT candidates

application computed using the profile data instead of computing them individually for each atomic section.

4.2. Performance Comparison

The number of candidate data structure types selected by VGAT scheme for each benchmark is reported in Column 2 of Table 3. We find that VGAT scheme had performance impact on only 6 of the 8 benchmarks namely *Kmeans*, *Intruder*, *Genome*, *Yada*, *Vacation* and *Ssca2*. In the remaining 2 benchmarks, we observed *negligible performance differences (< ±1%) on applying VGAT (we discuss the reasons for this later in the section). Therefore we report performance results only for the 6 benchmarks, in which VGAT has a performance impact.* We report the percentage improvement in execution time for 4, 8, 16 and 32 threads in Table-4a, Table-4b, Table-4c and Table-4d respectively for the 6 STAMP benchmarks. We measured the reduction in the D-cache miss latency cycles for these benchmarks as compared with our baseline STM. We report these results also in Table-4a, 4b, 4c and 4d in Column 5. We note that the scheme improves the performance in all of these applications. Further the performance improvement increases with increasing number of threads. VGAT scheme also helps improve the memory performance of the application by reducing the number of lock accesses (due to the variable access tracking granularity). Data cache miss information reported in Column 5 was obtained using the performance profile tool HP-Caliper [16] from the IA-64 hardware performance counters.

Intruder has 3 atomic sections. VGAT scheme selects 7 data structure types as candidates. VGAT improves performance by associating a larger access tracking granularity (equal to '*sizeof(queue_t)*') for '*decodedQueue* ' and '*packetQueue*' in two atomic sections which are quite hot. The remaining 5 data structures such as '*Stream*', '*Decoded_t*', '*packet_t*' etc selected by VGAT scheme are not hot in intruder and hence do not contribute much to improved performance.

Two of the hot atomic sections in *Intruder* access data structure types with different access granularity requirements. The decoder map needs to have a fine grained access tracking granularity in order to avoid false conflicts whereas the decoder queue should have a coarse grained access tracking granularity. Therefore a single access tracking granularity cannot benefit all the data structures in the hot atomic sections and therefore VGAT shows considerable performance benefits.

In *Vacation*, there is no single atomic section that contributes to most of the lock accesses. Instead all the three atomic sections contribute to the lock accesses. VGAT shows performance improvement of 5.23% to 19.08%. In the benchmark *Kmeans*, VGAT improves performance by 9% to 21%. *Kmeans* has 3 atomic sections. The array '*centre_len*' accessed in the atomic section in normal.c is selected as a candidate by VGAT since VGAT analysis detects that the default access tracking granularity of word size is sub-optimal for this array access. This atomic section has two data structures which are accessed requiring different access tracking granularity. One of them has high intra-transaction data locality which benefits from a coarse grained access tracking granularity whereas the other has a high inter-transaction data locality which benefits from a fine grained access tracking granularity. Hence the VGAT scheme benefits this benchmark considerably.

BM	Execution Time In seconds		% improvement Execution Time	% Improvement in dcache miss Latency
	TL2	TL2 VGAT		
Kmeans	11.43	10.38	9.15%	5.28%
Genome	7.01	6.57	6.23%	2.86%
Intruder	70.42	67.01	4.84%	2.31%
Ssca2	75.24	73.38	2.47%	1.21%
Vacation	67.01	63.50	5.23%	3.41%
Yada	7.12	6.98	1.87%	0.93%

Table 4a Performance Improvements with 4 threads

BM	Execution Time		% improvement in Execution Time	% Improvement in dcache miss Latency
	TL2	TL2-VGAT		
Kmeans	8.01	6.96	13.18%	7.01%
Genome	5.28	4.89	7.21%	4.67%
Intruder	62.64	57.73	7.83%	4.09%
Ssca2	61.62	59.25	3.83%	2.07%
Vacation	42.46	38.42	9.51%	4.85%
Yada	5.06	4.92	2.67%	1.52%

Table 4b Performance Improvements with 8 threads

BM	Execution Time		% improvement in Execution Time	% Improvement in Dcache miss Latency
	TL2	TL2-VGAT		
Kmeans	5.61	4.58	18.21%	11.16%
Genome	3.78	3.39	10.06%	6.03%
Intruder	51.03	44.90	12.51%	6.98%
Ssca2	51.23	48.64	5.04%	3.18%
Vacation	22.24	18.84	15.29%	8.34%
Yada	3.92	3.76	3.97%	1.93%

Table 4c Performance Improvements with 16 threads

BM	Execution Time		% improvement in Execution Time	% Improvement in Dcache miss Latency
	TL2	TL2-VGAT		
Kmeans	4.12	3.24	21.19%	15.65%
Genome	2.05	1.75	14.62%	8.56%
Intruder	50.08	40.45	19.23%	10.76%
Ssca2	46.39	42.69	7.98%	4.39%
Vacation	15.20	12.29	19.08%	10.86%
Yada	2.03	1.93	5.14%	2.87%

Table 4d Performance Improvements with 32 threads

In *Yada* and *Ssca2*, VGAT scheme finds 6 and 8 candidates respectively. However many of the VGAT candidate types are accessed in atomic sections which are not quite hot. Hence the performance improvements with VGAT are not high in these benchmarks. Though VGAT scheme finds candidates in *Labyrinth* and *Bayes* as shown in Table-3, these benchmarks have negligible performance improvements over unmodified STM. We found that the candidates selected by VGAT scheme occur in atomic sections which are cold in these benchmarks. For instance, in the benchmark '*labyrinth*', VGAT selects the shared data structure '*pathVectorList*' as a candidate. However the atomic section where this is accessed is cold and hence VGAT has no impact on the application performance for this benchmark.

Access tracking granularity has a considerable impact on false conflicts and hence on aborts. Identifying false conflicts and measuring them explicitly would require extensive instrumentation and would introduce perturbation which can affect the quantity being measured. Instead we report the % reduction in total aborts (which include aborts caused due to both true and false conflicts) for our benchmarks in Table-5 over our baseline STM for 4,8,16 and 32 threads. We find that VGAT results in a significant reduction in aborts in intruder, kmeans, and vacation, which reflect as performance improvements due to VGAT in these benchmarks.

4.3. Overheads

We measured the compile time overheads due to VGAT. We found that the compile time overheads due to VGAT analysis range from 0.94% to 2.31%. Note that VGAT scheme requires profile information. It is possible to use either dynamic profile or static profile data using compiler's non-profile based heuristics for the branch frequencies and loop iteration counts [4]. Since we used the dynamic profile data, we also incurred the additional cost of the dynamic profile collection runs using the simulator size (small size) inputs of the STAMP benchmarks. Including the profile collection run times in our compile time overheads result in compile time overheads of 2.98% to 6.14%.

5. Related Work

Various schemes have been studied widely in the context of improving the performance of STM implementations by selecting different fixed access tracking granularities [7, 18]. Well known word-based STM implementations use either word-level locking (e.g., TL2 [4] and TinySTM [17]) or cache-line level locking (e.g., McRT-STM C/C++ [14]). There has been considerable work on object level granularity STMs especially in managed environments [9, 12]. TL2 [4] supports two fixed access tracking granularity schemes namely 'Per Stripe' (PS scheme) and a 'Per Object' (PO Scheme). The programmer can choose one or the other for his application by enabling certain flags while compiling his application. Shavit [4] showed that PS scheme performed better than PO for a set of benchmarks and hence TL2 uses PS scheme with word size as the access tracking granularity by default.

Dragojević [18] evaluated the impact of the locking granularity on STM performance. They found that a lock granularity of four words performed better than both word-level and cache line-level locking by 4% and 5% respectively across the set of benchmarks they studied, demonstrating that the access tracking granularity which is best for an application, is highly application/application region dependent. Hence a single fixed granularity access tracking scheme which may perform well for one set of

BM	% reduction in aborts			
	4T	8T	16T	32T
Kmeans	18.28%	21.61%	23.12%	25.9%
Genome	7.46%	9.01%	12.32%	14.98%
Intruder	13.31%	14.89%	17.91%	19.32%
Ssca2	4.23%	5.74%	6.35%	6.98%
Vacation	37.3%	44.21%	45.2%	49.7%
Yada	1.43%	1.87%	1.62%	2.19%

Table-5 Reduction in aborts

applications can be sub-optimal for a different set of applications or different parts of the same application, motivating the need for a variable granularity access tracking approach like ours.

McRT STM [14] supports object level access tracking for small objects. However this requires a special purpose allocator supported by the STM. While Riegel et al [15] proposed objectifying transactional accesses in word based STM where possible, their scheme supports only two fixed granularities namely either object level access tracking or word level access tracking. As shown in our motivating Example 3 in section 2.3, object level access tracking can lead to loss of application concurrency since not all parts of the application are best served by the same access tracking granularity. On the other hand, our approach by allowing variable granularity access tracking per data structure, overcomes the disadvantages of pure object level access tracking approach.

Implementation of atomic sections using compile time lock assignment schemes has been studied in [22, 23, 24, 25, 26 and 31]. Emmi [25] formulates the lock allocation problem as an Integer Linear Programming (ILP) problem whose goal is to minimize the conflict cost between atomic sections and the total number of locks assigned to critical sections. Sreedhar [22] propose a compiler inferred lock assignment scheme which they model as a minimum lock allocation problem. Our approach is complementary and can co-exist with any of the above approaches since our work focuses on evaluating the best access tracking granularity for a given shared data structure based on the data access patterns of the different code regions.

Picking different global granularities and measuring performance based on different runs is a trial and error approach and hence is not practical. Hence trying to select a global granularity for a large application by trial and error is not practical. VGAT is completely automatic, is not a trial and error method and hence is practical. Our legality checks mark certain data structure types as ineligible, whereas in a global granularity approach, all the data is tracked with the selected granularity. We note that the non-selection of certain types for variable access tracking granularity scheme can result in leaving some performance on the table in the case of VGAT-STM scheme compared to the performance possible with the best global granularity scheme in certain specific cases.

A hybrid approach to lock assignment in STM using an interference graph to allocate locks to interfering critical sections was studied in [24]. However it requires precise alias analysis information since only shared data references which have only must aliases are considered for compile time lock allocation, unlike our type based VGAT scheme which does not require sophisticated alias analysis. Also their compile time lock allocation incurs dynamic memory allocation overheads due to individual locks for each candidate and requires synchronization for the lock allocation table which makes it non-scalable unlike our approach. Their approach does not include any cost-benefit model for guiding the lock assignment. It simply uses an interference graph to minimize the number locks without impacting concurrency. VGAT builds a cost-benefit model to find the appropriate granularity for different field groups, based on execution frequencies of atomic sections.

6. Conclusion

In this paper, we have proposed a compiler aided Variable Granularity Access Tracking (VGAT) scheme which can help improve STM performance by reducing the number of lock accesses required without adversely impacting application concurrency. While our current work focuses on a compile time VGAT scheme, we are also investigating an adaptive runtime access tracking granularity switching scheme as part of future work.

References

1. Shavit, N. and Touitou, D. 1995. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (Ottawa, Ontario, Canada, August 20 - 23, 1995). PODC '95. ACM, New York, NY, 204-213.
2. Herlihy, M., Luchangco, V., Moir, M., and Scherer, W. N. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing* (Boston, Massachusetts, July 13 - 16, 2003). PODC '03. ACM, New York, NY, 92-101.
3. M.F. Spear, V.J. Marathe, W.N. Scherer III, and M.L. Scott, "Conflict Detection and Validation Strategies for Software Transactional Memory," Proc. of the 20th Int'l Symp. on Distributed Computing, Sept. 2006.
4. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In Proceedings of the 20th International Symposium on Distributed Computing (DISC), Stockholm, Sweden, September 2006.
5. Cascaval, C., Blundell, C., Michael, M., Cain, H. W., Wu, P., Chiras, S., and Chatterjee, S. 2008. Software Transactional Memory: Why Is It Only a Research Toy? Queue 6, 5 (Sep. 2008), 46-58.
6. C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In IISWC '08: Proc. IEEE International Symposium on Workload Characterization,
7. Dice and N. Shavit. 2007. Understanding Tradeoffs in Software Transactional Memory. In Proceedings of the international Symposium on Code Generation and Optimization (March 11 - 14, 2007).
8. Ni, Y., Welc, A., Adl-Tabatabai, A., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., and Tian, X. 2008. Design and implementation of transactional constructs for C/C++. In Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (Nashville, TN, USA, October 19 - 23, 2008). OOPSLA '08. ACM, New York, NY, 195-212.
9. Shpeisman, T., Menon, V., Adl-Tabatabai, A., Balensiefer, S., Grossman, D., Hudson, R. L., Moore, K. F., and Saha, B. 2007. Enforcing isolation and ordering in STM. SIGPLAN Not. 42, 6 (Jun. 2007), 78-88.

10. Spear, M. F., Dalessandro, L., Marathe, V. J., and Scott, M. L. 2009. A comprehensive strategy for contention management in software transactional memory. *SIGPLAN Not.* 44, 4 (Feb. 2009), 141-150.
11. http://www.open64.net/documentation
12. Adl-Tabatabai, A., Lewis, B. T., Menon, V., Murphy, B. R., Saha, B., and Shpeisman, T. 2006. Compiler and runtime support for efficient software transactional memory. In Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (Ottawa, Ontario, Canada, June 11 - 14, 2006). PLDI '06. ACM, New York, NY, 26-37.
13. Yoo, R. M., Ni, Y., Welc, A., Saha, B., Adl-Tabatabai, A., and Lee, H. S. 2008. Kicking the tires of software transactional memory: why the going gets tough. In Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (Munich, Germany, June 14 - 16, 2008). SPAA '08. ACM, New York, NY, 265-274.
14. Saha, B., Adl-Tabatabai, A., Hudson, R. L., Minh, C., and Hertzberg, B. 2006. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (New York, New York, USA, March 29 - 31, 2006). PPoPP '06. ACM, New York, NY, 187-197.
15. Torvald Riegel and Diogo B. D. Brum (feb 2008). Making Object-Based STM Practical in Unmanaged Environments. In: *TRANSACT~'08: 3rd Workshop on Transactional Computing.*
16. Hundt, R. 2000. HP Caliper: A Framework for Performance Analysis Tools. IEEE Concurrency 8, 4 (Oct. 2000), 64-71.
17. Felber, P., Fetzer, C., and Riegel, T. 2008. Dynamic performance tuning of word-based software transactional memory. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Salt Lake City, UT, USA, February 20 - 23, 2008). PPoPP '08. ACM, New York, NY, 237-246.
18. Dragojević, A., Guerraoui, R., and Kapalka, M. 2009. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland, June 15 - 21, 2009). PLDI '09. ACM, New York, NY, 155-165.
19. Wang, C., Chen, W., Wu, Y., Saha, B., and Adl-Tabatabai, A. 2007. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In Proceedings of the international Symposium on Code Generation and Optimization (March 11 - 14, 2007). Code Generation and Optimization. IEEE Computer Society, Washington, DC, 34-48.
20. Larus J., Rajwar R. Transactional Memory. Morgan and Claypool Publishers
21. Herlihy, M. and Moss, J. E. 1993. Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21, 2 (May. 1993), 289-300.
22. Zhang, V. Sreedhar, W. Zhu, V. Sarkar, and G. Gao. Optimized lock assignment and allocation: A method for exploiting concurrency among critical sections. TR-CAPSL-TM-065, University of Delaware, Newark, DE, 2007.
23. M. Hicks, J. Foster, and P. Pratikakis. Lock inference for atomic sections. In TRANSACT'06: Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, 2006.
24. Mannarswamy, S., Chakrabarti, D. R., Rajan, K., and Saraswati, S. 2010. Compiler aided selective lock assignment for improving the performance of software transactional memory. In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Bangalore, India, January 09 - 14, 2010). PPoPP '10. ACM, New York, NY, 37-46.
25. Emmi, M., Fischer, J. S., Jhala, R., and Majumdar, R. 2007. Lock allocation. SIGPLAN Not. 42, 1 (Jan. 2007), 291-296.
26. S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. Technical Report MSR-TR-2007-111, MSR, August 2007.
27. Wu, Y. and Larus, J. R. 1994. Static branch frequency and program profile analysis. In Proceedings of the 27th Annual international Symposium on Micro architecture (San Jose, California, United States, November 30 - December 02, 1994). MICRO 27. ACM, New York, NY, 1-11.
28. G. Chakrabarti and F. Chow, Structure Layout Optimizations in the Open64 Compiler: Design, Implementation and Measurements, Open64 Workshop, CGO 2008
29. R. Ennals. Efficient software transactional memory. Technical report, Intel Research Cambridge, Jan 2005
30. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.
31. Upadhyaya, G., Midkiff, S. P., and Pai, V. S. 2010. Using data structure knowledge for efficient lock generation and strong atomicity. SIGPLAN Not. 45, 5 (May. 2010), 281-292.