Handling Conflicts with Compiler's help in Software Transactional Memory Systems

Sandya Mannarswamy
Indian Institute of Science & HP India
Bangalore, India
sandya@hp.com

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

Abstract *Atomic sections are supported in software through the use of optimistic concurrency by using Software Transactional Memory (STM). However STM implementations incur high overheads which reduce the wide-spread use of this approach by programmers. Conflicts are a major source of overheads in STMs. The basic performance premise of a transactional memory system is the optimistic concurrency principle wherein data updates executed by the transactions are to disjoint objects/memory locations, referred to as Disjoint Access Parallel (DAP). Otherwise, the updates conflict, and all but one of the transactions are aborted. Such aborts result in wasted work and performance degradation.*

While contention management systems in STM implementations try to reduce conflicts by various runtime feedback control mechanisms, they are not aware of the application's structure and data access patterns and hence typically act after the conflicts have occurred. In this paper we propose a scheme based on compiler analysis, which can identify static atomic sections whose instances, when executed concurrently by more than one thread always conflict. Such an atomic section is referred to as Always Conflicting Atomic Section (ACAS). We propose and evaluate two techniques Selective Pessimistic Concurrency Control (SPCC) and compiler inserted Early Conflict Checks (ECC) which can help reduce the STM overheads caused by ACAS. We show that these techniques help reduce the aborts in 4 of the STAMP benchmarks by up to 27.52% while improving performance by 1.24% to 19.31%.

1. Introduction

Atomic sections have been proposed recently as a programming idiom for expressing synchronization at a higher level of abstraction than locks. Programmers can specify what code has to execute atomically by simply enclosing the desired block of code with the keyword 'atomic'. Atomic sections are an interesting alternative to locks as they allow local reasoning and are composable. Atomic sections are supported in software through the use of optimistic concurrency by using transactional memory [1].

Software transactional memory (STM) is a promising programming paradigm for shared memory multithreaded programs as an alternative to traditional lock based synchronization [2]. Atomic sections marked by the programmers are executed as transactions and safe access to shared data is ensured implicitly. The TM system compares each transaction's data accesses against all other transactions for conflicts. If conflicting data accesses are detected between any two transactions, typically one of them is aborted and usually restarted immediately. Selecting which transaction to abort out of two conflicting transactions is based upon the contention management policy of the TM system. If a transaction completes execution without conflicts, then it commits which makes the changes to shared data visible to the whole program.

However STM implementations incur a high overhead, making it difficult for them to be widely used efficient programming paradigm [5]. Conflicts are a major source of overheads in STM implementations [7]. Since transactions execute optimistically and roll back if a conflict is detected, there may be lot of wasted work done by transactions which eventually abort. Aborted transactions reduce performance; reduce scalability and waste computing resources. Moreover certain TM implementations (e.g., update in place) [28] require extra program resources to roll back the program to a consistent state in case of an abort.

The major premise behind performance expectations of TM systems is the principle of optimistic concurrency control (OCC). OCC assumes that data accessed by concurrently executing transactions is typically disjoint and conflicting data accesses are infrequent. Or in other words, applications exhibit considerable amount of Disjoint Access Parallelism (DAP). This allows the TM implementations to execute the transactions without obtaining exclusive ownership of data at the beginning of an atomic section unlike a pessimistic controlled atomic section. In case any data conflicts actually are encountered, TM resolves them by aborting all but one of the conflicting transactions.

Conflicts occur when concurrently executing transactions access non-disjoint data in atomic sections. Executing such atomic sections using OCC has a negative impact on performance because of increased aborts due to conflicting transactions. Regions of low or zero disjoint access parallelism are better served by pessimistic concurrency control since it avoids wasteful work due to aborting transactions.

In applications which were written without keeping optimistic concurrency control in mind, there may exist certain data structures or constructs which do not exhibit disjoint access parallelism [24] and are accessed inside atomic sections whose instances when executed concurrently by more than one thread *always* conflict. Such an atomic section is referred to as Always Conflicting Atomic Section (ACAS). When such applications are ported to STM implementations, the ACAS regions become application bottlenecks, being a major contributor to conflicts and hence to aborts in execution. The presence of a single conflicting operation in an atomic section which otherwise is disjoint access parallel, will act as the funnel for the overall DAP of that atomic section. Hence identifying such ACAS regions and applying special techniques to handle them is important in improving the performance of general purpose applications on STM.

While contention management systems in STM implementations try to reduce overheads due to conflicts by various runtime feedback mechanisms, they are not aware of the application's structure and data access patterns and hence

typically act after the conflicts have occurred [8, 9, 10, 11, 17, 18]. There has been recent work on using runtime analysis to switch the execution mode of transactions from optimistic to pessimistic concurrency (and vice versa) based on the contention experienced at runtime [11]. In [17], Sonmez et al., propose using runtime profiling to switch to pessimistic acquisition mode for certain hot and highly contentious data variables based on the contention experienced at runtime. All these methods generally kick in after the ratio of aborts to commits for an atomic section exceeds certain threshold at runtime and hence act after the problem has set in. Hence a static analysis by the compiler which can determine at compile time which atomic sections are better executed with PCC would be beneficial in improving the performance of STM.

While a static analysis may not be able to determine the amount of DAP available in an atomic section accurately (due to various factors such as conservative alias analysis by the compiler, non-availability of certain information such as the number of concurrently executing threads), it is possible to identify atomic sections with no disjoint access parallelism at all (we denote such atomic sections as Always Conflicting Atomic Sections (ACAS)) and treat them with special techniques to reduce the number of aborts incurred by them. Such a static analysis scheme can act complementary to any runtime contention controlling scheme and aid it as well with the information obtained through static analysis.

In this paper we propose a scheme, based on compiler analysis, which can help identify ACAS regions. We show that a major TM performance pathology known as Restart Convoy described in [22] which arises typically due to such ACAS regions, can be detected using our static analysis scheme. We propose two techniques to handle ACAS regions namely (a) Selective Pessimistic Concurrency Control (SPCC) and (b) compiler inserted Early Conflict Checks (ECC). We show that these techniques can help reduce the aborts and wasted work caused by ACAS regions. We also discuss various uses of the ACAS information.

We used Open64 compiler [27] and TL2 [4] STM implementation to prototype our scheme. We used STAMP benchmark suite [6] for our experimental evaluation. Our results indicate that identifying ACAS regions at compile time and handling them with special techniques improves performance by reducing the number of aborts and the amount of wasted work due to aborting transactions. We show that our SPCC technique helps improve application performance by 1.24% to 19.31% in 4 of the STAMP benchmarks while reducing the number of aborts by 5.45% to 27.22%. Our ECC technique helps improve application performance by 1.23% to 9.49%.

The rest of this paper is organized as follows: In Section 2, we provide a detailed motivation for our work. Section 3 describes our scheme to detect ACAS regions and our techniques to handle them. 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. Background

This section motivates the need for our work by showing how ACAS regions affect the performance of applications on STM which employs optimistic concurrency control for executing atomic sections.

2.1 Disjoint Access Parallelism (DAP)

Software transactional memory systems typically implement atomic sections using optimistic concurrency control. This is unlike lock-based programming where a critical section protected by a lock is entered only after obtaining exclusive ownership of the lock, which means that no two threads could be inside the same critical section at the same time. A pessimistic concurrency control implemented using lock-based programming is based on the premise that shared data accessed/updated within the critical section is expected to be non-disjoint across threads. Hence it is necessary to obtain exclusive ownership of the data before executing the critical section since conflicts are definitely expected during the data accesses.

Pessimistic Concurrency Control (PCC) and Optimistic Concurrency Control (OCC) are similar to 'asking permission' (exclusive ownership before entering the atomic section) and 'apologizing if there is a conflict' (execute assuming data is disjoint, if conflict, apologize and retry). The choice of whether to ask permission first before executing vs.execute optimistically hoping for no conflicts, and deal with conflicts if they arise later, is not an easy decision from a performance viewpoint (for humans as well). Optimistic concurrency is driven by the assumption of disjoint access parallelism among concurrently executing transactions. It supposes that shared data accessed/updated by concurrent transactions is expected to be disjoint and hence conflicts among threads in accessing shared data is typically infrequent.

Disjoint access parallelism is the disjointness of data accesses between the concurrently executing transactions. An atomic section which exhibits disjoint access parallelism is executed by concurrently executing transactions as long as there are no conflicts among the transactions. On the other hand, if the concurrently executing transactions often access/update data which is not access parallel, then the data accesses conflict with each other, leading to aborts for all conflicting transactions except one which commits, leading to wasted work due to aborting transactions.

```
TM_BEGIN
char* TMdecoder_getComplete (Queue*
                        decodedQueuePtr){
    char* data;
    decoded_t* decodedPtr =
                TMQUEUE_POP(decodedQueuePtr);
    data = decodedPtr->data;
    return data;
}
TM_END
TMQUEUE_POP(Queue* decodedQueuePtr) {
   //remove the head element from
   // elements array
   …….
   decodedQueuePtr->size--;
}
```

Figure 1 Code Segment Involving Dequeue

Optimistic concurrency control does not benefit such Always Conflicting Atomic Sections (ACAS), since transactions will definitely conflict and contribute to aborts.

2.2 Motivating Example 1

Consider the code snippet shown in Figure 1 which is simplified from the STAMP benchmark program `Intruder`. In this code snippet, a number of threads execute this atomic section removing elements from the queue. Each pop operation operating on the same queue also changes the size of the queue (inside the 'TMQUEUE_POP' function/method) and the elements of the queue. Hence if there are concurrent transactions executing this atomic section, they will definitely conflict on the update to the fields 'size' and 'elements' of the queue. Therefore two or more transactions concurrently executing this atomic section will definitely suffer a write-write conflict and only one transaction will be allowed to commit and all other concurrently executing transactions will abort and then restart. Thus this atomic section does not exhibit disjoint access parallelism and hence is an ACAS region. Such an ACAS region does not benefit from executing it under optimistic concurrency control., Inter-procedural analysis by the compiler can determine that all threads executing this atomic section operate on the same queue and hence will always conflict, aborting each other. Hence it is possible for the compiler to detect such ACAS regions and apply special techniques to handle them instead of allowing them to be executed under the usual STM's optimistic concurrency control.

The behavior exhibited by the atomic section in Fig 1 is an example of a well known TM performance pathology known as 'Restart Convoy' identified and characterized in [22]. Convoys arise in TM systems with lazy conflict detection when one committing transaction conflicts with (and aborts) multiple instances of the same static transaction. The aborted transactions restart almost simultaneously, compete for system resources, and due to their similarity, finish together. The crowd of transactions competes to commit, and the winner aborts the others. A transaction convoy degrades performance due to repeated conflicts as the restarted transactions contend for system resources, leading to wastage of work done by the loser transactions in the convoy each time.

By compile time analysis of the application, it is possible to identify ACAS regions which can form potential 'Restart Convoys'. This information can be used by the compiler to turn such atomic sections into pessimistic concurrency controlled atomic sections. This information can also be fed to the TM's contention management scheme to execute such transactions under adaptive transaction scheduling [10] or communicated to the application programmer so that he can redesign the application to avoid such pathologies.

2.3 Motivating Example 2

While the atomic section we saw above is dominated by an operation on queue without any disjoint access parallelism, there are also cases where the presence of single conflicting access in an atomic section which is otherwise disjoint access parallel, can have a detrimental impact on performance. Consider the following code snippet which is simplified from the STAMP benchmark Genome.

```
TM_BEGIN(1);
{
  long ii;
  long ii_stop = MIN(i_stop,(i+CHUNK_STEP1));

  for (ii = i; ii < ii_stop; ii++) {
    void* segment = vector_at(
       segmentsContentsPtr, ii);
    TMHASHTABLE_INSERT(
        uniqueSegmentsPtr, segment,
        segment, 1)
  } /* ii */
}
```

Figure 2 Hash Table Manipulation

There are a number of threads executing this atomic section concurrently, with each thread operating on disjoint values of 'ii'. In the 'for' loop, each thread gets a unique segment from the segment vector and inserts into the hash table. As long as the hash values are different, we expect that this operation should exhibit disjoint access parallelism as the different sequences would map to different buckets of the hash table. Hence at first glance, we expect that this atomic section should perform well under optimistic concurrency control on an STM implementation. However we found that when executing `Genome` on TL2 STM implementation, this atomic section was a major contributor to aborts with 52% of the total aborts coming from this atomic section. By using compiler instrumentation to break down the aborts incurred in this atomic section to various data accesses, we found that most of the aborts occur in the hash table insert operation.Figure 3 shows the simplified code snippet for the hash table 'insert' operation. Each bucket is implemented as a linked list and values which hash to different buckets will be disjoint access parallel.

```
TM_BEGIN()
TMhashtable_insert (
   hashtable_t* hashtablePtr,
   void* keyPtr, void* dataPtr)
{
  long numBucket =
      hashtablePtr->numBucket;
   long i =
       hashtablePtr->hash(keyPtr) %
       numBucket;

   TMLIST_INSERT(
       hashtablePtr->buckets[i],
       dataPtr);

   long newSize =
       TM_SHARED_READ(
           hashtablePtr->size) + 1;

   assert(newSize > 0);

   TM_SHARED_WRITE(
       hashtablePtr->size,
       newSize);
```

Figure 3 Hash Table Insert Function

However as we can see in Figure 3, while the insertion of the data item itself is disjoint access parallel, the atomic section

contains an update to hash table size which is non disjoint access parallel (NDAP). This NDAP access results in the funneling the overall DAP for this atomic section.

The conflicting update to the hash table size happens much later in the execution of each transaction. Hence considerable amount of work is done by all the concurrently executing transactions before they reach this conflicting point, leading to much wasted work since all except one transaction will need to abort and restart. Compiler can detect this conflicting update occurring in this atomic section using inter-procedural analysis. The compiler can use the ACAS information to either execute this atomic section selectively under PCC. Or it can try reducing the amount of wasted work by aborting transactions by inserting a conflict check much early in the transaction for the data location which it knows to be likely cause of conflict.

It is easy to see that both the code snippets we have seen in examples 1 and 2 can be part of a large application executing on a STM. While an STM application typically contains atomic sections which are amenable to execution under optimistic concurrency control, there can be certain atomic sections which exhibit no disjoint access parallelism and will always conflict. In this paper, we propose a compiler approach to detect such ACAS regions and apply special techniques to handle them, as we discuss next.

3. Our Approach

3.1 Preliminaries

A concurrent application consists of a set of programmer specified static atomic sections. Each static atomic section instance is dynamically executed as a set of concurrent transactions by a group of threads. In the following discussion, we use the term 'atomic section' to refer to the static instance of the atomic section as present in the application source code and the term 'dynamic instance of atomic section' to refer to its dynamic instance which is executed as a transaction. The basic performance premise of a transactional memory system is the optimistic concurrency principle wherein data updates executed by the transactions are to disjoint objects/memory locations, referred to as Disjoint Access Parallelism (DAP). Otherwise, the updates conflict, and all but one of the transactions are aborted. As discussed in the Introduction, such aborts result in wasted work and performance degradation. This TM performance pathology is referred to as 'Restart Convoy' [22]. Hence, in this work, we focus on determining atomic sections whose instances always result in access conflicts and propose schemes for executing them in an efficient manner.

Conflicts can be either read/write or write/write conflicts. Since read/write conflicts need not lead to aborts in all cases [19], we focus our attention only on write/write conflicts and identifying them at compile time in a conservative manner. In other words, we are interested in identifying atomic section X whose instances when executed concurrently by more than one thread **always** conflict. Such an atomic section is referred to as Always Conflicting Atomic Section (ACAS).

3.2 Always Conflicting Atomic Sections

Consider an update to a shared datum in a static atomic section such as p->a = … where p is pointer to a structure and 'a' is a field of that structure. If 'p' points to a different object in each dynamic instance of this atomic section, then the dynamic instances of the atomic section can execute concurrently (as for as this object is concerned) without any conflicts being incurred during this update. However if 'p' points to the same object in all the dynamic instances of the atomic section, then all such concurrently executing dynamic instances of this atomic section will necessarily conflict on this update. We can use compiler's static analysis to build a data access model for identifying ACAS regions. Our current approach is restricted to 'must conflict' shared data updates (W-W conflicts), which are more amenable to identification by the compiler analysis as we discuss below. It is possible that an ACAS atomic section can be executed non-concurrently by each of the threads so that none of the 'must conflict' updates actually lead to actual conflicts at runtime. In such a case, taking a pessimistic approach is unnecessary. However it is typical of applications that a static atomic section is executed by more than one thread concurrently, especially in the SIMD style programs. Hence our approach assumes that ACAS atomic sections typically are executed by more than one thread concurrently.

Given a shared datum d_i which is accessed inside an atomic section A, if different instances of A update the same shared datum d_i, then the atomic section A is an Always Conflicting Atomic Section. We refer to such an update as a must-conflict (MC) update. An atomic section A which updates multiple shared data items $d_1, d_2, \ldots, d_k$, is ACAS, if any of the update is an MC-update. Conversely, an atomic section is non-ACAS, or has Disjoint Access Parallelism (DAP) only if none of the updates is an MC update Thus a simple compile-time analysis helps to identify ACAS and DAP atomic sections. While it is beneficial to execute a DAP atomic section in an optimistic manner (using transactions), ACAS can incur significant wasted work and performance degradation under optimistic execution. We propose two different schemes for address the execution of ACAS in this paper.

3.2.1 Identifying Must Conflict Updates

Each atomic section contains a list of shared data accesses/updates. Since we focus on W-W conflicts, we restrict our discussion to shared data updates. We analyze the shared data update inside an atomic section by considering the set of objects it can point to. Consider an update operation of the form 'p->a = ….' inside an atomic section. If the reference 'p' points to a single object as determined by static analysis, then the update operation 'p->a' is a must-conflict update since all concurrently executing transactions of that static atomic section will update the same object.

In order to determine whether a shared data reference say x is a must-conflict, we associate the attribute 'Points to One Object' POO(x). POO (x) denotes that the reference x can point to one and only one object as determined by static analysis. If POO(x) is true, then clearly the update is a Must-Conflict update. It should be noted that if x points to more than one object, we cannot still guarantee that the updates do not conflict (due to the conservative points to/alias analysis). However, optimistically, we say that the update can be optimistically tried as exhibiting Disjoint Access Parallelism (DAP). Compiler uses the points to analysis information associated with the reference 'p' to decide whether the reference p points to a single object or not.

```
Class A{                                Class B{
        B b;                                    int d;
        B c;                                    float e;
};                                      };

foo1()
{
  A a;

  T1= Pthread_create(   bar, &a.b);
  T2 = Pthread_create(bar, &a.c);,

  Thread_barrier_wait();
}

bar(B* myB)
{
    Atomic
    {
        myB->d = 10;
    }
}
```

Figure 4 Example Atomic Section with DAP

For each static instance of atomic section, the compiler collects the set of shared data updates 'A' performed inside it. For each shared data update x ∈ A, compiler uses the inter-procedural points to information to determine the POO(x) attribute. If POO(x) is true, then the update is a Must-Conflict update. An atomic section containing a Must-Conflict update is marked as Always Conflicting Atomic Section (ACAS) by the compiler. For each atomic section, the compiler needs to analyse each of the shared data updates to see if it is a must conflicting update. Given an atomic section ASi which contains a list of shared data updates SDU(ASi), ACAS analysis needs to examine each of the |SDU(ASi)| references with |SDU(ASj)| for $i \neq j$. This must be done for all pairs of atomic sections for $i\ j \in (1, N)$ where N is the number of the static atomic sections in the application. We explain these concepts with the help of examples:

In the code snippet in Figure 4, we have a static atomic section AS1 which contains a shared data update 'myB->d = 10'. In order to determine the disjointness of this update, the compiler tries to determine the POO attribute for the reference 'myB'. When AS1 is executed by T1, myB points to a.b; whereas when AS1 is executed by T2, myB points to a.c. Hence compiler sets the POO(myB) as false. Hence the data update operation myB->d is marked as DAP and the AS should be executed as a transaction to exploit optimistic concurrency. However, in this code, if the calls to Pthread_create had the same pointer '&a.b' passed as argument to both T1 and T2, then the updates becomes Must-Conflict. This is because POO(myB) is set to true as myB points to only one object namely 'a.b'. Hence the AS becomes ACAS.

Further, in the above example in Figure 4, the points to set for myB consisted of directly addressed memory locations namely a.b or a.c. In case of expressions involving multi-level dereferences, the members of the points to set can themselves be pointer variables, requiring the compiler to recursively check the points-to members to see if they point to a single object as we illustrate next.

Consider the code sequence in Figure 5. The points-to set of 'r' consists of 'q' which is pointer variable. Hence the compiler

```
struct B {
        int m ;
        int n ;
    } ;

B b ;

int* q = &b;

int** r;

*r = &q;

atomic {

  r->m = ...
}
```

Figure 5 Example Involving Multi-Level Dereferencing

recursively checks the points-to set of 'q'. The points-to set of 'q' consists of a single object 'b' and hence the update r->m has its POO(r) set to true. Therefore the compiler marks the update 'r->m = …' as Must-Conflict in the above code snippet.

Compiler determines whether a shared data update is a must-conflict update using the POO information. Hence the POO information needs to be accurate. Note that it is possible that a shared data reference 'x' is marked with POO(x) as false when it is not so, due to the limitations of the underlying points-to analysis by the compiler, This can lead to an update getting marked as disjoint when it is actually a Must-Conflict. The atomic section containing the update will not be marked as ACAS. Hence it will be executed under OCC, and can suffer considerable conflicts leading to performance degradation. Hence the accuracy of our ACAS analysis scheme is dependent on the pointer analysis scheme supported by the compiler. Since the open64 compiler currently supports only a context insensitive pointer analysis scheme, we implemented a prototype of the context sensitive points to analysis scheme based on the Data Structure Analysis (DSA) framework [25], to support our ACAS analysis[1] and to avoid any performance issues due to ACAS regions not getting identified accurately.

3.2.2 Conflict Costs

The location of the must-conflict shared data updates in an atomic section is important from the perspective of the amount of wasted work, if the atomic section is executed in optimistic concurrency as a transaction. A must conflict on a shared data update which occurs early in the execution of an atomic section will cause an early abort in TMs which support eager conflict detection, resulting in less wasted work as compared to a must-conflict which occurs late during the execution of atomic section[2]. Hence we associate a conflict cost with each of the

[1] Our implementation is largely identical to the Data Structure Analysis framework implemented in the LLVM compiler [30] except for certain implementation differences.

[2] In a TM with lazy conflict detection, update conflicts are detected only during the commit phase when locks are acquired. Hence conflicts are always detected late in such systems.

must-conflict shared data updates identified by the compiler during ACAS analysis to estimate the amount of wasted work due to an abort which occurs because of that conflict.

We compute the conflict cost as the ratio of the number of cycles spent executing the instructions occurring before the conflicting update in the atomic section to the number of cycles spent executing the total number of instructions in the atomic section. In case of branches/backward edges being present in the atomic section, we use the static profile heuristics of the compiler to assign conditional branch/backward edge counts for computing the instruction counts. To compute the number of cycles taken for execution, we associate a average instruction latency (the average number of cycles it takes to execute the instruction) with each instruction, obtained from the machine model used by the compiler. Greater the conflict cost of an atomic section, larger is the amount of wasted work. Section 4.2 reports the conflict cost range for ACAS regions in our benchmark applications. ACAS atomic sections with high conflict costs need to be treated with special techniques to reduce the conflict costs. After the compiler finishes processing all atomic sections in ACAS analysis, the compiler marks each of atomic sections marked either as ACAS or as DAP. Next we discuss how this information can be used to improve STM performance.

3.3 Using the Must-Conflict and ACAS Information

If an atomic section does not have disjoint access parallelism, then it does not benefit from optimistic concurrency model imposed by the software transactional memory implementation. Such an atomic section can have a detrimental effect on performance of an STM application if it is a frequently executed part of the user application. We discuss two techniques namely selective pessimistic concurrency control and Compiler Inserted Early Conflict Checks to handle ACAS.

3.3.1 Selective Pessimistic Concurrency Control (SPCC)

One approach to handle ACAS regions is to turn them into pessimistic concurrency controlled atomic sections. Compiler can transform the atomic section into a pessimistic one by synthesizing a compiler allocated lock which protects the entire atomic section. We call this approach, Selective Pessimistic Concurrency Control. Compiler synthesizes a lock for the selected atomic section and inserts locking instrumentation at the entry and exit of the atomic section. Correct execution requires that a consistent locking discipline is followed throughout the execution of the atomic section. Assuming that any of the atomic sections can be executing concurrently with each other, this implies that each access to a shared address should be consistently mapped to the same lock. Since the compiler has associated a lock with this atomic section, all the shared data items accessed inside this atomic section are protected by this compiler synthesized lock. To ensure the uniformity of the locking discipline, references to these shared data items anywhere in the application should be protected by the same compiler synthesized lock uniformly.

Let AS_{cand} be the candidate atomic section which is being considered for SPCC. Let SD(AScand) be the set of shared data accessed inside AS_{cand}. Let AS_{other} be any other static atomic section which can potentially execute concurrently with AS_{cand}. Let SD(AS_{other}) be the set of shared data accessed inside AS_{other}. If any of the shared data items in SD(AScand) are accessed in AS_{other}, compiler transforms those shared data references in AS_{other} to use the same lock in order to ensure a uniform locking discipline. Only the lock association for the shared datum references in AS_{other} is changed to use this compiler assigned lock, while the shared data references in AS_{other} continue to execute under optimistic concurrency control, without any explicit locking being inserted for AS_{other}. This lock association is automatically performed by the compiler and does not require any programmer effort.

Currently we assign a single lock to protect the ACAS which can impact the concurrency. While it is possible for our compiler to assign an individual lock for each shared datum in the ACAS atomic section, such a naïve assignment can lead to excessive locking overhead. The ideal solution is to assign the minimum number of locks for protecting the shared data items in the ACAS without sacrificing parallelism. We are working on enhancing SPCC to use fine grained locking automatically by the compiler without requiring any programmer effort, as part of our future work, using the approach outlined in [26].

Further, Compiler needs to perform certain checks before it can apply SPCC to an atomic section. It needs to ensure that none of the shared data items accessed inside an AS_{cand} escape to opaque external library functions whose code is not visible to the compiler for transformation. If there are such opaque library calls in an atomic section marked as ACAS, compiler does not transform the ACAS using SPCC. Currently we do not perform any cost-benefit analysis other than checking for eligibility in deciding whether to apply SPCC for an ACAS. The compiler can use either the runtime application profile information if it is available or the static conflict costs as estimated in Section 3.3, to decide whether to apply SPCC or not. We plan to study this as part of future work.

There are also certain semantic issues that need to be considered in applying selective pessimistic concurrency control in an application originally intended to be run under an STM. Pessimistic concurrency control of an atomic section implemented using compiler allocated locks can result in behavior which is not always equivalent to that of a purely transactional mode of execution in certain cases. For instance, with STM using lazy updates (redo logs), memory updates occurring inside the atomic block are visible only after the transaction is committed whereas with lock protected atomic sections, memory updates are immediately visible. This can lead to difference in behavior when atomic sections are executed using transactional memory vs. locks. As expected, all such unexpected behaviors involve transactional and non-transactional code accessing the same shared data with at least one write access. The issue of semantic differences between locks and transactional mode of execution has been discussed in significant detail in previous literature [13, 14, 15, and 23]. Hence we do not discuss this issue further due to space constraints.

3.3.2 Compiler Inserted Early Conflict Checking

If an ACAS is determined to be ineligible for SPCC transformation by the compiler, due to shared objects escaping to opaque library calls, we need an alternative technique to avoid the excessive conflicts due to those atomic sections. Recall that an atomic section gets marked as ACAS by the

compiler because one or more updates to shared data are marked as Must-Conflict. Depending on the conflict detection scheme employed by the STM (whether eager or lazy), the detection of conflict for the shared data updates will happen either at the point of update or at the time of commit. The transaction could have performed considerable amount of work before the conflict is detected leading to waste of useful work. Hence if an ACAS is not eligible for SPCC, we propose an alternative technique known as "compiler inserted Early Conflict Checks (ECC)" for handling such ACAS. This technique requires an extension to the STM implementation, namely a new interface known as 'CheckAndAcquireLock' which we describe later. Note that ECC is complimentary to and can coexist with SPCC in the same application.

Since compiler has determined by means of static analysis, the set of the shared data updates that are Must-Conflict for a given atomic section, it can insert instrumentation at the earliest possible point in the program where the effective address of the shared datum is available, to explicitly check for conflict.

Let AS be an atomic section marked as ACAS and not selected for SPCC. Let SD (AS) be the set of shared data updates determined to be Must-Conflict by the compiler during the analysis phase. For each data access a ∈ SD(AS), compiler applies standard data flow analysis techniques to determine at what point within the atomic section, the effective address of the shared data getting updated can be computed at the earliest. It then inserts a check to see if the data item is already locked by another transaction. The check consists of a call to a new STM interface 'CheckAndAcquireLock' which we had implemented in TL2. CheckAnd AcquireLock takes the address of the shared update as a parameter. If it is locked, the transaction aborts itself immediately to avoid wastage of useful work since it will ultimately abort due to this must-conflict. Else it acquires the lock for that shared datum so as that any other concurrently executing transaction can detect the conflict at the earliest possible point when they execute the call to 'CheckAndLock' and abort early.

While ECC can be used an alternative scheme for reducing wasted work due to conflicts in the event of an atomic section not being eligible for SPCC, all the ACAS identified in our experimental benchmarks were eligible for SPCC. So we evaluate these two techniques independently in our experiments and report results in Section 4.4.

Note that ECC does not result in a direct reduction in the number of conflicts since the atomic section is still executed under OCC. However since a conflict check is made at the earliest point when the effective address of the datum is available, the amount of wasted work by an aborting transaction is reduced. The insertion of early conflict checks by the compiler contributes to additional runtime overheads and we report the runtime overheads for our experimental benchmarks in Section 4.5.

3.3.3 Other uses of ACAS information

The ACAS information on atomic sections and conflicting data accesses can be fed to contention manager so that contention manager can delay/serialize such conflicting transactions. Our underlying STM implementation TL2 does not have any in-built contention manager module. TL2 uses the simple and passive contention management scheme wherein 'requester always aborts'. TL2 also uses a simple back off scheme wherein an aborting transaction has a short time delay before it is restarted. Hence we do not evaluate the idea of feeding the ACAS information to a runtime contention manager in this paper and leave it for future work.

Since ACAS information is available at compile time, compiler can also advise the programmer of such always conflicting atomic sections in the application so that programmer can redesign the application if needed.

4. Experimental Evaluation

4.1 Experimental Methodology

To evaluate the effectiveness of our approach, we used the STAMP benchmark suite [6] version 0.9.10. We implemented a prototype of our approach in Open64 compiler [27] and with TL2 [4] as our underlying STM implementation. The baseline for our experiments is to compile the benchmarks with inter-procedural analysis enabled, but with ACAS detection phase turned off in the compiler and run them on the unmodified TL2 implementation. We used the native (non-simulator) input sets of the STAMP benchmark suite in our experiments. We used a 16 core IA-64 RX7640, with 2 cores per socket, with each core of 1.6 GHz clock frequency and 9 MB non-shared L3 cache per core.

4.2 Identification of ACAS regions

Benchmark	No. of Static Atomic Sections	ACAS found	Statically estimated conflict costs in ACAS
Kmeans	3	2	0.5, 0.6
Vacation	3	0	0
Genome	5	1	0.8
Intruder	3	3	0.6, 0.6, 0.8
Labyrinth	3	2	0.6, 0.7
Ssca2	10	6	0.3 to 0.6
Yada	6	3	0.4, 0.5, 0.7
Bayes	15	5	0.3 to 0.5

Table – 1 ACAS regions

Table-1 gives the total number of static instances of atomic sections in each benchmark in column 2 and the number of atomic sections determined to be non-disjoint access parallel by our ACAS analysis for each benchmark in column 3 of Table-1. We also report the statically estimated conflict costs in column 4 of Table-1 for the ACAS regions. We also measured the compile time overhead due to our ACAS analysis by comparing the compile time of each benchmark with ACAS analysis and without ACAS analysis enabled. We found that compile time overheads due to ACAS analysis were within the range of 0.7% to 3.2%.

4.3 Selective Pessimistic Concurrency Control

We find that applying SPCC technique had performance impact in 4 of the 8 STAMP benchmarks. We report performance results only for these 4 benchmarks, in which SPCC has a performance impact. We measured negligible performance differences (< ±1%) on applying SPCC to the other 4 benchmarks (we discuss the reasons for this later in the section). We report both the % improvement in execution time

and the % reduction in aborts, for 2,4,8 and 16 threads in Table-2 for the 4 STAMP benchmarks on which SPCC had a performance impact (of > 1%).

BM	SPCC % Improvement in	2 threads	4 threads	8 threads	16 threads
Intruder	Exec. Time	3.74%	5.76%	12.24%	19.31%
	Aborts	11.26%	17.23%	21.43%	27.52%
Genome	Exec. Time	2.14%	3.78%	5.35%	8.14%
	Aborts	8.22%	11.23%	13.94%	16.28%
Labyrinth	Exec. Time	1.71%	3.19%	4.92%	6.17%
	Aborts	6.54%	11.23%	14.11%	16.28%
Yada	Exec. Time	1.24%	3.06%	4.63%	5.82%
	Aborts	5.45%	7.22%	8.95%	11.02%

Table – 2 SPCC Performance Improvements

In `Intruder`, SPCC is applied to atomic sections where operations are performed on a single queue. Since all threads operate on the same data structure concurrently, there is no disjoint access parallelism and hence conflicts occur leading to aborts. Hence applying SPCC reduces the number of aborts significantly. In the benchmark `Genome`, SPCC is applied to one atomic section where entries are inserted into a hash table. The insert operation updates a common 'size' field of the hash table structure. Hence all threads executing this atomic section concurrently suffer conflicts on this shared data update. Applying SPCC avoids conflicting on this shared update. In the benchmark 'yada', SPCC is applied to atomic sections which operate on a heap, thereby reducing the conflict on the conflicting update to the shared fields of the heap data structure. In the benchmark `Labyrinth`, SPCC is applied to atomic sections involving updates to a queue and a list respectively. List insertion operation updates the shared data field 'size' which causes conflicts when multiple threads update the same list.

While our analysis identified ACAS regions for applying SPCC in remaining three STAMP benchmarks namely kmeans, ssca2 and bayes, we found that SPCC had negligible performance impact on these benchmarks. In these benchmarks, the ACAS regions involved a tight RMW (Read/Modify/Write) operation on a single global variable. Hence the conflicts experienced in these atomic sections were quite less due to the very short transactions. Also these atomic sections were cold in these benchmarks. So there was no performance impact due to applying SPCC on these atomic sections.

4.4 Compiler Inserted Early Conflict Checks (ECC)

We report the % improvement in execution time for 2,4,8 and 16 threads in Table-3 for the 4 STAMP benchmarks on which ECC had a performance impact (of > 1%). We find that performance improvements due to ECC technique were quite modest compared to those obtained with SPCC over the baseline TM. This is due to the fact that ECC also incurs additional runtime overheads due to explicit early checks inserted by the compiler. We report the runtime overheads in Section 4.5. We did not find any significant change in number of aborts after applying ECC (changes were less than 1%), hence we do not report the details on the aborts in Table 4. This is expected since ECC does not reduce the number of aborts per se, but only reduces the amount of wasted work due to eventually aborting transactions.

BM	2 threads	4 threads	8 threads	16 threads
Intruder	2.32%	5.12%	7.72%	9.49%
Genome	1.27%	2.83%	4.81%	6.38%
Labyrinth	1.42%	3.06%	4.12%	5.36%
Yada	1.23%	2.69%	3.96%	4.81%

Table – 3 ECC Performance Improvements

4.5 Runtime Overheads

We measured the overheads due to early conflict checks inserted by the compiler. In this experiment for measuring the overhead, ACAS analysis scheme inserts the early conflict check code, but the STM does not do an early abort if the ECC succeeds. Instead it uses the TM's normal execution path of going ahead and executing the transaction as if the check did not succeed. The overheads measured as the performance degradations over our base line (run with 16 threads), are shown in Table-4. We find that overhead ranges from 0.56% to 2.17% for five of the STAMP benchmarks. There was no significant performance impact on the other three STAMP benchmarks. Overhead due to ECC can reduce the performance benefits of our approach. Hence we are also investigating compiler heuristics based on profile data which help do ECC insertion only in selected cases.

Benchmark	**Overhead**
Intruder	2.17 %
Genome	1.73 %
Labyrinth	1.34%
Yada	1.22%
Bayes	0.56 %

Table – 4 Runtime Overheads due to ECC

5. Related Work

Contention management schemes have been studied widely in the context of improving the performance of STM implementations in order to reduce the number of aborts encountered in high contention scenarios [3, 5, 7, 9, 20]. The evidence of different contention management schemes proving effective under different scenarios suggests that, in a larger application built using a number of data structures, not all atomic sections are amenable to the default optimistic concurrency control implemented by STM.

Ansari et al. [16] and Yoo et al. [10] examined the approach of adjusting the level of concurrency as a way of avoiding contentionWalliullah et al. [29] proposed a scheme to insert a checkpoint before the first conflicting access to reduce the amount of wasted work due to an aborting transaction in a HTM. Dragojevic et al. [18] describes a transaction scheduler which predicts future contentions based on the access patterns of the past history of the transactions.

Harris et al [17] proposed dynamic selection of pessimistic concurrency control for selected set of 'hot' variables which contribute to large number of aborts. Dolev et al. [8] proposed a proactive collision reduction scheme which requires applications to provide information about transactions' collision probability. Usui et al [11] proposed an adaptive locking technique that dynamically observes whether a critical section would be executed best transactionally or using mutex locks based on runtime profile data. Sonmez at all discuss runtime profile feedback directed selection of pessimistic concurrency control to certain 'hot' variables which cause frequent conflicts.

All the above approaches typically require runtime profiling and contention monitoring information to make decisions on which transactions to schedule/abort/delay/prioritize. They do not use compile time analysis of the application and hence are unaware of the data structures involved in the transactions and their disjoint access parallelism characteristics. Our approach is complementary and can co-exist with any of the above approaches which require runtime profiling/monitoring information with certain ACAS regions handled by SPCC and ECC at compile time, whereas others which are not amenable to static analysis can be handled by runtime schemes.

While it is not the intent of our current work to compare the performance of SPCC and ECC techniques driven by the static analysis information with a full fledged runtime contention management scheme, we show that it is possible to use static analysis by the compiler to identify ACAS and such ACAS regions can be handled by special techniques which can help reduce conflict overhead costs compared to a baseline STM which does not employ any contention management scheme. t .

Mannarswamy et al [12] examined the approach of improving STM's runtime lock assignment using compiler analysis. However their approach only changes the lock assignment associated with selected data while executing all atomic sections under optimistic concurrency control. Our approach is complimentary to such hybrid lock assignment schemes and can co-exist with them.

6. Conclusions

We have discussed how a compiler based static analysis approach can help identify always conflicting atomic sections and how the ACAS information can be used to direct certain atomic sections to be executed under PCC selectively to reduce the number of aborted transactions and amount of wasted work. We showed that our scheme can reduce aborts and improve application performance from 1.24% to 19.31% for certain benchmarks. We are studying a hybrid approach wherein the compiler analysis information can be used along with runtime contention management approach as part of our future work.

References

1. N. Shavit and D. Touitou. Software transactional memory. In Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, Aug 1995.
2. M. Herlihy, V. Luchangco, M. Moir, and W. N.Scherer, III. Software transactional memory for dynamic-sized data structures. In PODC '03: Proc. 22nd ACM Symposium on Principles of Distributed Computing, July 2003.
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, Stockholm, Sweden, 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, Sweeden, September 2006.
5. Cascaval, C.Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. 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, pages 35–46, Sep 2008.
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). Washington, DC, 21-33.
8. S. Dolev, D. Hendler, and A. Suissa. CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In PODC '08. pages 125–134, August 2008.
9. W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing, pages240–248, July 2005.
10. R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in Algorithms and Architectures, 169–178, June 2008.
11. Usui, T., Behrends, R., Evans, J., and Smaragdakis, Y. 2009. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *Proceedings of the 2009 18th international Conference on Parallel Architectures and Compilation Techniques* (September 12 - 16, 2009). PACT. IEEE Computer Society, Washington, DC, 3-14.
12. Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Ollivier, S. Preis, B. Saha, A. Tal, and X. Tian Design and implementation of transactional constructs for C/C++. In International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2008
13. Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*. October 2007.
14. T. Shpeisman, V. Menon, A. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI '07:* NY, USA, 2007. ACM
15. Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Single global lock semantics in a weakly atomic STM. In *3rd workshop of Transactional Computing (TRANSACT)*, 2008.

16. Ansari, M., Kotselidis, C., Jarvis, K., Lujan, M., Kirkham, C., and Watson, I. 2008. Experiences using adaptive concurrency in transactional memory with Lee's routing algorithm. In PPoPP '08. ACM, New York, NY, 261-262.

17. Sonmez, N., Harris, T., Cristal, A., Unsal, O. S., and Valero, M. 2009. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. IPDPS 2009. IEEE Computer Society, Washington, DC, 1-10.

18. Dragojević, A., Guerraoui, R., Singh, A. V., and Singh, V. 2009. Preventing versus curing: avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing* (Calgary, AB, Canada, August 10 - 12, 2009). PODC '09. ACM, New York, NY, 7-16.

19. Dragojevi , R. Guerraoui, and M. Kapalka. Dividing transactional memories by zero. In TRANSACT, 2008.

20. Guerraoui, R., Herlihy, M., and Pochon, B. 2005. Toward a theory of transactional contention managers. PODC '05. ACM, New York, NY, 258-264.

21. 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.

22. Bobba, J., Moore, K. E., Volos, H., Yen, L., Hill, M. D., Swift, M. M., and Wood, D. A. 2007. Performance pathologies in hardware transactional memory. *SIGARCH Comput. Archit. News* 35, 2 (Jun. 2007), 81-91.

23. Martín Abadi , Andrew Birrell , Tim Harris , Michael Isard, Semantics of transactional memory and automatic mutual exclusion, Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, January 07-12, 2008, San Francisco, California, USA

24. Israeli, A. and Rappoport, L. 1994. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing* (Los Angeles, California, United States, August 14 - 17, 1994). PODC '94. ACM, New York, NY, 151-160.

25. Lattner, C., Lenharth, A., and Adve, V. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. *SIGPLAN Not.* 42, 6 (Jun. 2007), 278-289.

26. 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).

27. http://www.open64.net/documentation

28. A.-R. Adl-Tabatabai, B. T. Lewis, V. S. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In PLDI 2006.

29. M. M. Waliullah and P. Stenstrom. Intermediate checkpointing with conflicting access prediction in transactional memory systems. In IPDPS, pages 1--11. IEEE Computer Society, 2008.