The Runtime Abort Graph and its Application to Software Transactional Memory Optimization

Dhruva R. Chakrabarti, Prithviraj Banerjee, Hans-J. Boehm, Pramod G. Joisha, Robert S. Schreiber

HP Laboratories
HPL-2010-179

Abstract:
Programming with atomic sections is a promising alternative to locks since it raises the abstraction and removes deadlocks at the programmer level. However, implementations of atomic sections using software transactional memory (STM) support have significant bookkeeping overheads. STM programmers therefore need tools that provide insights and means to improve performance.

This paper attempts to identify the source of an abort at the granularity of a transactional memory reference. The resulting abort patterns are captured in the form of a runtime abort graph (RAG). We show how to build this graph efficiently using compiler instrumentation and discuss its qualitative semantics. We then describe a technique that works on the RAG and automatically recommends STM policy changes to improve performance. Significant performance improvements, upto 90% individually, have been obtained. We present detailed experimental results showing the tradeoffs in building the RAG and its use in reducing aborts and improving performance.

External Posting Date: February 16, 2011 [Fulltext] Approved for External Publication
Internal Posting Date: November 8, 2010 [Fulltext]

Parts of this work will appear in the International Symposium on Code Generation and Optimization (CGO), 2011. Copyright for those parts of this work is held by IEEE.

The Runtime Abort Graph and its Application to Software Transactional Memory Optimization

Dhruva R. Chakrabarti, Prithviraj Banerjee, Hans-J. Boehm, Pramod G. Joisha, Robert S. Schreiber
Hewlett-Packard Laboratories
{dhruva.chakrabarti, prith.banerjee, hans.boehm, pramod.joisha, rob.schreiber}@hp.com

Abstract

Programming with atomic sections is a promising alternative to locks since it raises the abstraction and removes deadlocks at the programmer level. However, implementations of atomic sections using software transactional memory (STM) support have significant bookkeeping overheads. STM programmers therefore need tools that provide insights and means to improve performance.

This paper attempts to identify the source of an abort at the granularity of a transactional memory reference. The resulting abort patterns are captured in the form of a runtime abort graph (RAG). We show how to build this graph efficiently using compiler instrumentation and discuss its qualitative semantics. We then describe a technique that works on the RAG and automatically recommends STM policy changes to improve performance. Significant performance improvements, upto 90% individually, have been obtained. We present detailed experimental results showing the tradeoffs in building the RAG and its use in reducing aborts and improving performance.

1. Introduction

Transactional memory (TM) has emerged as a promising paradigm for writing shared memory multithreaded programs. In the last few years, a number of STMs have come into prominence. Intel released a prototype compiler-based STM system [9]. A number of API-based STM libraries, including TL2 [6] and RSTM [11], have been released. While a lot of work has gone into building efficient STM support, not much is known about effectively analyzing performance characteristics or using them to produce automated techniques for improving performance of an STM application.

Recently, a few research papers have appeared on this topic. It is generally agreed that overall abort rate is not helpful in characterizing the complex behavior of realistic TM workloads and that *averaging* transactional statistics can be quite deceiving [18]. As a solution, statistics of STM behavior on an atomic block basis have been generated [9, 14, 18]. Debugging and profiling techniques have been reported [20, 21] that identify conflicts from an execution and correlate them to source constructs. However, none of these capture aborters and victims at the level of a memory access.[1]

[1] Aborters and victims are discussed more in Section 2.

Parts of this work will appear in the International Symposium on Code Generation and Optimization (CGO), 2011.

They capture victims at the memory access granularity but aborters are only captured at the atomic section granularity. In this paper, we build upon our prior work [5] and define the semantics of a runtime abort graph (RAG) for an STM application, show how to build it efficiently using compiler instrumentation, and then describe automated techniques that utilize the actionable data obtained from the RAG and generate optimized, higher-performing STM applications.

The technique we use essentially follows the trajectory of a classical offline feedback-directed compilation scheme. The application is first compiled with instrumentation turned on. When the instrumented executable is run, an annotated RAG is created. An offline tool analyzes the abort patterns in the RAG and generates a modified execution recipe aimed at better runtime performance. For this paper, the offline tool is designed to emit policy recommendations for conflict detection at the atomic section level. In the modified execution, some atomic sections are executed using an eager policy while the rest use a lazy policy. This paper makes two broad contributions:

- It shows that runtime abort relationships in an STM application can be captured efficiently at load/store granularity.
- It shows that such abort relationships offer actionable data that can be utilized by automatic techniques to improve runtime performance.

Our STM support is based on TL2 [6], an open source blocking STM implementation. Shared reads and writes are protected by read barriers (denoted by TxLoad) and write barriers (denoted by TxStore) respectively. TL2 allocates a table of locks and assigns a lock to every shared location. The read barrier ensures that mutually inconsistent memory locations are never accessed. The write barrier either buffers writes or performs in-place updates after acquiring locks for the shared locations. If a transaction cannot proceed because of a conflict, it is rolled back and automatically restarted. Otherwise, a commit is attempted at the end of the transaction.

After outlining some basic assumptions and definitions in Section 2, we introduce the RAG and its semantics in Section 3. In Section 4, we describe the machinery developed by us for building the RAG. No changes are required at the programmer level (i.e. the API), the classical atomic{} construct continues to work. New STM interfaces and changes to some existing ones called by the compiler are required (i.e. the ABI needs extending). These changes and other details including the *events* that need instrumentation are described in this section. We also discuss counter-based sampling that helps lower the runtime overheads of instrumented code. Section 5 introduces the components of our framework, illustrates their interactions, and shows how the RAG is represented on persistent store for arbitrary clients to consume. In Section 6, we introduce HyAcq or hybrid acquire. The RAG drives the transformation in HyAcq that is designed to generate policy recommendations at the atomic section level with the aim to reduce aborts and wasted work.

Section 7 discusses the tradeoffs between overheads in building the RAG and its accuracy and shows that the RAG can be built without prohibitive overheads. Results on HyAcq show that using hybrid policies on an atomic section level improves performance. We discuss related work in Section 8 and conclude in Section 9.

2. Basic Assumptions and Definitions

Atomic Section vs Transaction: This paper assumes that the input program is multithreaded and uses atomic sections. An atomic section is a static notion since it refers to a block of code that must appear to execute in an indivisible manner. For the purpose of this paper, we designate a transaction to be a dynamic execution instance of an atomic section. Hence, there may be multiple transactions corresponding to the same atomic section, executing concurrently in distinct threads.

Association between Shared Datum and Ownership Record (*orec*): Our technqiues are directly applicable to blocking STMs. We assume an association between shared datum and an ownership record, which can be thought of as a lock protecting the shared datum. It does not matter whether an *orec* is assigned per object or per memory stripe [6] as long as there is a mapping function that accepts an address and returns the *orec* that protects it. Let us call this mapping the *Datum to Ownership Record (DOR)* function. As is consistent with blocking STMs, a single ownership record is assumed to be associated with any given address throughout the lifetime of a given execution. TL2 maintains a table of *orecs* (or the *orec*-table) on which the DOR function is applied.

Conflicts, Aborters, and Victims: A conflict occurs when two concurrently executing transactions reference or appear to reference[2] the same shared memory location and at least one of these references is a write. In such a scenario, one of these transactions may be aborted because of the conflicting reference it makes. When a transaction aborts, it is because execution cannot proceed with a task initiated by a memory reference. We designate this memory reference as the victim of that particular abort. For a given abort, the aborter is the memory reference whose execution prevented the victim from proceeding. Here is an example: if a memory reference m_1 in transaction t_1 acquires an *orec* o and another memory reference m_2 in transaction t_2 tries to acquire the same prec subsequently, t_2 may abort since it cannot immediately proceed. In this case, m_2 is the victim and m_1 is the aborter. Note that this terminology has also been used at the granularity of an atomic section [14]. In such a case, the atomic section corresponding to t_1 is the aborter and that corresponding to t_2 is the victim.[3]

False Conflicts and Orec Aliasing: Conflicts can be categorized into true and false. If the two memory references involved indeed access the same shared location, it is a true conflict. Otherwise it is a false conflict. In this context, we introduce *orec aliasing*. Two memory references (regardless of whether their addresses are disjoint or not) are said to be *orec-aliased* if the shared locations accessed by them map to the same *orec*. Thus *orec aliasing* covers both true and false conflicts.

Transactional Functions and References: A function is transactional if it can be transitively called from an atomic section. A reference is transactional if it executes under the control of a transactional memory system. A dynamic transactional memory reference (DR) refers to an execution instance of a transactional load or a store. On the other hand, a static transactional memory reference SR is a transactional load or a store in the intermediate representation (IR) of a program. Hence, a single SR may correspond to more than one DR. For the rest of this paper, a read or write memory reference will be assumed to be transactional unless specified otherwise.

[2] The latter scenario is the result of a false conflict defined later in **False Conflicts and *Orec* Aliasing**.

[3] This definition needs to be adapted for implementations such as logTM which pause instead of aborting.

3. Runtime Abort Graph (RAG)

A RAG is a directed graph. A node corresponds to an SR. An edge captures an abort relationship and is directed from the aborter to the victim SR. A node can have the following annotations.

- α_o: id of dynamically outermost enclosing atomic section containing the SR[4]
- SR_{id}: an identifier corresponding to the SR
- L: source code information for the SR
- S_{rd}: average readset size of the outermost enclosing transaction at the point the corresponding SR is the victim
- S_{wr}: average writeset size of the outermost enclosing transaction at the point the corresponding SR is the victim
- CN_a: Total number of aborts suffered by the SR

Every node is keyed with the duple $N_k =< \alpha_o, \text{SR}_{id} >$ and the key determines whether a corresponding node exists or a new one needs to be created. An edge is annotated with CE_a, the total number of times the source node aborts the target node. For a given node, CN_a is computed as the sum of CE_a over all incoming edges. Note that S_{rd} and S_{wr} are ignored if the node is never a victim.

3.1 Qualitative Semantics of a RAG

A RAG captures the aborts that are encountered during an execution of a program. The aborter and victim of an abort are identified and represented.[5] The graph is sound but not necessarily complete, meaning that an edge is never added between two nodes unless and until the corresponding memory references are involved in an abort at runtime. All runtime aborts may not be captured leading to its incompleteness. These properties are discussed more in Section 4.7.

There is a subtle point regarding the information captured by a RAG. Consider the following example where r1 and r2 are temporaries and x, y, and z are shared data.

```
Thread 1                          Thread 2
atomic {  // AS1                  atomic {  // AS2
1: r1 = z                         3: x = 10
// Entire AS2 executes here       4: y = 10
2: r2 = x                         5: z = 10
}                                 }
```

Assume that at runtime, the transactions, corresponding to the two atomic sections shown above, interleave the way shown. Let's also assume that x and y are *orec*-aliased (as defined in Section 2). So when the load from x on line 2 is a victim, the store to y on line 4 will be found to be the aborter (as opposed to the store to x on line 3 that also conflicts with the load from x). This is because the last SR corresponding to a given *orec* is tracked. But in another execution using a different DOR function that eliminates the false conflict, the store to y on line 4 is not the aborter. This example illustrates that for a given victim, the last aborter is captured exactly as it happens during that execution but that can lead to masking other possible aborters.

[4] (1) We support closed nesting and track only the outermost atomic section for profiling purposes. Since any nested atomic section is primarily for partial rollback, lack of inner atomic section information in the profile data does not impose any serious drawback. (2) If an SR is contained within more than one distinct outermost atomic sections, a separate node is added to the graph for each such instance.

[5] Note that the aborter is always a store, the victim may be a store or a load.

Note the following points:

- Difference between a RAG and a complete runtime graph: For the purpose of this paper, a RAG is summary of a complete runtime graph. All aborts encountered during execution may not be captured in a RAG.
- Difference between a RAG and a concurrency graph [19]: A RAG captures aborts encountered during execution. On the other hand, a static conflict graph represents all possible conflicts between static memory references. Because of this critical difference, optimizations that rely on conflict patterns to generate correct code (such as lock inference) cannot be applied on a RAG. However, optimizations that improve performance based on application characteristics work quite well on a RAG.

3.2 Thread Independence of a RAG

A RAG does not encode any information about threads. When an abort relationship is captured, the aborter and victim belong to different threads. But since a RAG is the result of aggregation of all abort patterns seen in an execution, a given node in a RAG may have executed in different threads. No attempt is made to include any of these thread-related information in a RAG since it is meant to capture memory references that have contention among themselves at runtime.

4. Building a RAG

4.1 Unique Identifiers

We require that every atomic section and SR be assigned a program-wide unique identifier. We now describe how these are assigned and maintained in the compiler and the STM.

4.1.1 Atomic Section Identifier

A unique id, α, is assigned to every atomic section at runtime. We introduce a new STM ABI function, RegisterAS, that a compiler inserts just after the call to start a transaction.[6] Figure 1 shows the intermediate code generated by a compiler and an implementation of RegisterAS. A global lock protects the global counter that is used for id assignment. The common case invocation of the registration function will return quickly without encountering any synchronization operation. The number of times the critical section within RegisterAS is executed is bounded by the total number of threads in the program. The compiler uses a file static (id1 in Figure 1(b)) to store the assigned id.[7]

Note that distinct file statics must be used for distinct atomic sections. The value of α is cached in the transaction metadata making it available to all subsequent STM interfaces.

4.1.2 SR Identifier and Source Location Information

We use an identifier, β, to distinguish between transactional functions. Its computation is supported by a new STM ABI function, RegisterFn, that a compiler inserts at the start of every transactional function and any function that has an atomic section within its body. The implementation of RegisterFn is similar to that of RegisterAS, the only difference being that a different global lock

```
foo() {
  TxStart();
  ...
  TxCommit();
}
(a) Original IR

static volatile uint32_t
  id1 = UINT32_MAX;
foo() {
  Desc * d = TxStart();
  RegisterAS(&id1);
  d->id = id1;
  // Use d->id
  ...
  TxCommit(d);
}
(b) IR after id
    assignment
```

```
// global lock
lock_type as_id_lock;
RegisterAS(volatile uint32_t *
  as_id) {
  if (*as_id == UINT32_MAX) {
    lock(&as_id_lock);
    if (*as_id == UINT32_MAX) {
      increment as_global_id;
      *as_id = as_global_id;
    }
    unlock(&as_id_lock);
  }
}
(c) STM Interface implementation
```

Figure 1. Compiler and STM support code for unique identification of atomic sections

is used for assigning function ids. Like in RegisterAS, the number of times the critical section within RegisterFn is executed is bounded by the total number of threads in the program.

γ is assigned statically by the compiler in a way that every SR within the lexical scope of the enclosing function gets a unique value. Figure 2(a) and (b) show a simple example. Note that transactional references within different functions may have the same value of γ. The duple $\mathtt{SR}_{id} =< \beta, \gamma >$ uniquely identifies a transactional reference within an entire application.

We also track some source information, referred to as $L =< \lambda, \rho, \tau >$, where λ is the mangled name of the caller function and ρ and τ are the line number and column number respectively.

4.1.3 Resiliency of Identifiers through Downstream Compiler Transformations

The compiler lowers an atomic section into STM calls pretty early in the compilation pipeline. A natural question is whether the identifiers used in our technique are affected by subsequent compiler transformations. Towards that end, we investigated the effects of inlining since it could have a big impact in the presence of file statics.[8] It turns out that no modification to the classical inline transformation is required and that post-inline identifiers capture information the way they are intended to.

Consider Figure 2. In pseudo-code format, we show the original program (in two different source files s1.c and s2.c), the pre-inline, and the post-inline code in Figure 2(a), (b), and (c) respectively. Note that standard inter-file inline transformation techniques without any modification are used. These consist of promoting file statics to uniquely named globals and expanding a function call into an inlined instance. Two aspects are worth pointing out: (1) In post-inline code, multiple calls to RegisterFn may occur within the same function, as in f1 in s1.c in Figure 2(c). If these calls are made with the same arguments, as in lines 11 and 15, those other than the first are guaranteed to return immediately without executing any critical section. The number of times critical sections are executed within RegisterFn does not change after inlining. (2) The original transactional function and its inlined instances use the same id. An example is id_fn1_s2 (Figure 2(c), s1.c, lines 11 and 15, Figure 2(c), s2.c, line 6. This is consistent with the original behavior since the corresponding transactional references originated

[6] If there is an atomic section within the lexical scope of a transactional function, the compiler does not generate a call to RegisterAS. The transaction descriptor passed down to this transactional function is used for instrumentation purposes and as a parameter in calls made from this transactional function. This method thus automatically discards nested descriptors and results in tracking the outermost one.

[7] According to the upcoming C++0x semantics [3], it should be an atomic variable with memory_order_relaxed. Currently it is implemented using the C keyword volatile.

[8] We have not investigated a large class of compiler optimizations for this purpose — that remains part of future work. However, any transformation that does not clone identifiers is expected to work properly and indeed, most transformations fall in this category. Transformations that don't fall in the above category appear to have no effect either as exemplified by inlining.

```
s1.c
----
 1: f1() {
 2:   atomic {
 3:     x = ..;
 4:     f2();
 5:     f3();
 6:     f2();
 7:     w = ..;
 8:   }
 9: }
```

```
s2.c
----
 1: f2() {
 2:   y = ..;
 3: }
 4:
 5: f3() {
 6:   z = ..;
 7: }
```

(a) User program pseudo-code

```
s1.c
----
 1: static uint32_t id_fn1 =
 2:   uint32_max;
 3: static uint32_t id_as1 =
 4:   uint32_max;
 5: f1() {
 6:   RegisterFn(&id_fn1);
 7:   TxStart();
 8:   RegisterAS(&id_as1);
 9:   TxStore(x, .., <id_fn1,0>);
10:   f2();
11:   f3();
12:   f2();
13:   TxStore(w, .., <id_fn1,1>);
14:   TxCommit();
15: }
```

```
s2.c
----
 1: static uint32_t id_fn1 =
 2:   uint32_max;
 3: static uint32_t id_fn2 =
 4:   uint32_max;
 5: f2() {
 6:   RegisterFn(&id_fn1);
 7:   TxStore(y, ..,
 8:           <id_fn1,0>);
 9: }
10:
11: f3() {
12:   RegisterFn(&id_fn2);
13:   TxStore(z, ..,
14:           <id_fn2,0>);
15: }
```

(b) Intermediate compiler generated pseudo-code (pre-inline)

```
s1.c
----
 1: static uint32_t id_fn1 =
 2:   uint32_max;
 3: static uint32_t id_as1 =
 4:   uint32_max;
 5: f1() {
 6:   RegisterFn(&id_fn1);
 7:   TxStart();
 8:   RegisterAS(&id_as1);
 9:   TxStore(x, .., <id_fn1,0>);
10:   RegisterFn(&id_fn1_s2);
11:   TxStore(y, .., <id_fn1_s2,0>);
12:   RegisterFn(&id_fn2_s2);
13:   TxStore(z, .., <id_fn2_s2,0>);
14:   RegisterFn(&id_fn1_s2);
15:   TxStore(y, .., <id_fn1_s2,0>);
16:   TxStore(w, .., <id_fn1,1>);
17:   TxCommit();
18: }
```

```
s2.c
----
 1: uint32_t id_fn1_s2 =
 2:   uint32_max;
 3: uint32_t id_fn2_s2 =
 4:   uint32_max;
 5: f2() {
 6:   RegisterFn(&id_fn1_s2);
 7:   TxStore(y, ..,
 8:           <id_fn1_s2,0>);
 9: }
10:
11: f3() {
12:   RegisterFn(&id_fn2_s2);
13:   TxStore(z, ..,
14:           <id_fn2_s2,0>);
15: }
```

(c) Intermediate compiler generated pseudo-code (post-inline)

Figure 2. Automatic Handling of Ids during Inlining

from the same SR and hence it is only natural that they have the same $\texttt{SR}_{id}$.

4.2 Global Tables for Tracking Aborters

When an abort occurs, the victim SR is trivially available, being the currently processed one. However, the aborter SR is not immediately available as that must have been processed earlier by a different thread. In order to make this connection, we maintain shared tables (or global tables) that exist throughout the lifetime of the application. For the purpose of our technique, a global table has two characteristics: (1) It is shared among threads allowing communication of information across them.[9] (2) Once created, it remains persistent for the entire execution including across transactions. Updated information overwrites any previous information for a given entry.

The first of these, the *orec*-table, implements the DOR function mentioned in Section 2 for the core STM. For building the RAG, we introduce two new global tables: committed table (Ct) and acquired table (At). Ct maps a runtime address to the atomic section and the SR that modified it last. At maps a runtime address to the atomic section and the SR that has acquired the *orec* for that address. In our implementation, every word of Ct or At encodes α using 10 bits, β using 12 bits, and γ using 10 bits. Encoding all the relevant information within a 32-bit word maintains mutual consistency (assuming reads and writes of a 32-bit word are atomic) between α, β, and γ. Using the available bits, we are able to encode distinct 1K atomic sections, 4K functions, and 1K references within the lexical scope of a given function.

The *orec*-table, Ct, and At are tagless hash tables. A tagless 32-bit word design is simple because reading and writing of individual values can proceed without locking but some false aliasing issues may arise. Values in the hash tables are table safe, i.e. references to them cannot fault. All of these tables are indexed using addresses of shared memory locations.

4.3 Captured events

In order to modularize building the RAG we identify events within a transactional execution setting that lead to an aborter-victim relationship. The motivation is to have these events abstract and general enough to be usable in virtually any STM regardless of policies and implementation detail. Note that this is not an exhaustive list of all possible events but only the most popular ones. Figures 3 and 4 show pseudo-code implementation of the STM ABI functions. The events and interfaces for the RAG-build are in boldface.

- Locked location (E_{ll}): During execution of a transaction $T1$, if a reference r to a location m is attempted and if the corresponding *orec* lm is already acquired by another transaction $T2$, the contention manager may decide to abort $T1$. In such a case, the SR corresponding to r would be the victim and the memory reference that caused the *orec*-acquire in $T2$ would be the aborter. As shown in Figures 3 and 4, E_{ll} can occur during a transactional load (TxLoad, line 10, through the interface HandleNonOwnedOrec), a transactional store (TxStore, line 12), or during commit (TxCommit, line 5).
- Validation failure during Speculative execution(E_{vs}): When a location is opened for the first time within a transaction, the following steps are performed. If the timestamp of the location being opened is more recent than the timestamp maintained by the transaction, a round of validation is initiated. Elements of the readset are validated to make sure they are consistent with the location being opened. If validation fails, the transaction is aborted and restarted. In such a case, this event is recorded and both the location being opened the first time and the location that failed validation are marked victims. In a typical execution, the very first validation failure causes the transaction abort. If the abort is deferred till all elements of the readset are validated, more victims may be found. Currently, we capture only the first victim. E_{vs} can happen during a transactional load (TxLoad, line 12).
- Validation failure during Commit (E_{vc}): A round of validation is performed during the commit phase and any element that fails validation is a victim (as shown in TxCommit, line 8). Note that E_{vc} is different from E_{vs} in that E_{vc} is unconditionally encountered in many STMs.
- Read Speculation tracking (E_{rs}): This event is responsible for adding $\texttt{SR}_{id}$ of the read transactional reference to the readset. This information would be required for RAG-build if the corresponding readset entry were to fail validation later in the transaction.

[9] Currently these are implemented using the C volatile keyword.

```
1: intptr_t TxLoad(Desc * desc, intptr_t * addr, Info * info) {
2:     /* lazy mode only, lines 3-7 */
3:     if (addr ∈ wrset) {
4:         RdWrSetEntry * we = GetEntry(wrset, addr);
5:         /* No new event, already captured by previous write */
6:         return (*we).value;
7:     }
8:     intptr_t value = *addr;
9:     Orec orec = GetOrec(addr);
10:    HandleNonOwnedOrec(orec, desc, addr, info);
11:    if (orec.ts > desc.ts /* compare timestamps */
12:    && !TxValidate(desc)) /* E_vs */
13:        UpdateRAG&Abort(desc, addr, info, Ct);
14:    RdWrSetEntry * re = RdLog(desc, addr);
15:    SetRefId(re, info); /* E_rs */
16:    return value;
17: }
```

```
1: TxStore(Desc * desc, intptr_t * addr, intptr_t value, Info * info) {
2:     /* eager mode only, lines 3-14 */
3:     Orec orec = GetOrec(addr);
4:     HandleNonOwnedOrec(orec, desc, addr, info);
5:     if (orec.locked and orec.owner == desc) {
6:         RdWrSetEntry * we = GetEntry(wrset, addr);
7:         SetRefId(we, info); /* E_ws */
8:         *addr = value;
9:         return;
10:    }
11:    status = TryLockingOrec(addr);
12:    if (status == fail) /* E_ll */
13:        UpdateRAG&Abort(desc, addr, info, At);
14:    else insert mapping (addr, orec) in At /* E_aq */
15:    /* lazy mode only, lines 16-21 */
16:    if (addr in wrset) {
17:        RdWrSetEntry * we = GetEntry(wrset, addr);
18:        /* No new event, already captured by previous write */
19:        (*we).value = value;
20:        return;
21:    }
22:    RdWrSetEntry * we = WrLog(desc, addr, value);
23:    SetRefId(we, info); /* E_ws */
24:    *addr = value; /* eager mode only */
25: }
```

```
1: TxValidate(Desc * desc) {
2:     for (every entry in rdset)
3:         if (!valid(entry)) /* E_vs or E_vc */
4:             UpdateRAG&Abort(desc, entry.addr, entry.info, Ct);
5: }
```

Figure 3. Implementation of STM ABI functions

- Write Speculation tracking (E_{ws}): Any change made to the transactional state of a thread is not exposed to other threads until the transaction is committed. This event is the write-counterpart of E_{rs} and is responsible for adding $\texttt{SR}_{id}$ of the write transactional reference to the writeset.
- Acquire event (E_{aq}): At the point of an orec-acquire, the corresponding memory address and N_k are added to At.
- Commit event (E_{ce}): The hitherto speculative write reference information (i.e. the atomic section identifier and that added by E_{ws}) is committed during this phase.

4.4 Building Nodes and Edges in a RAG

The RAG is built dynamically as the program executes. Nodes and edges are built through UpdateRAG&Abort in Figure 4. If a transactional reference cannot proceed and has to abort, a corresponding target node is either created or reused if one already exists. The

```
1: TxCommit(Desc * desc) {
2:     /* lazy mode only, lines 3-7*/
3:     for (every entry in wrset) {
4:         status = TryLockingOrec(entry.addr);
5:         if (status == fail) /* E_ll */
6:             UpdateRAG&Abort(desc, entry.addr, entry.info, At);
7:         else insert mapping (entry.addr, orec) in At /* E_aq */
8:     TxValidate(desc); /* E_vc */
9:     Writeback(desc); /* lazy mode only */
10:    /* Traverse the undo/redo log and for every entry,
11:    commit the corresponding SR_id in Ct.
12:    Clear the corresponding entry in At. */
13:    CommitInfo(); /* E_ce */
14:    ReleaseOrecs();
15: }
```

```
1: HandleNonOwnedOrec(Orec orec, Desc * desc,
2: intptr_t * addr, Info * info) {
3:     if (orec.locked and orec.owner ! = desc) /* E_ll */
4:         UpdateRAG&Abort(desc, addr, info, At);
5: }
```

```
1: UpdateRAG&Abort(Desc * desc, intptr_t * addr,
2: Info * info, Mode mode) {
3:     /* Create target RAG-node from info */
4:     if (mode == Ct)
5:         Query Ct to create source RAG-node
6:     else query At to create source RAG-node
7:     /* Add or update connecting edge */
8:     TxAbort(desc);
9: }
```

Figure 4. Implementation of STM ABI functions (Contd.)

```
void RegisterAS(uint32_t *);
void RegisterFn(uint32_t *);
intptr_t TxLoad(Desc *, intptr_t *, Info *);
void TxStore(Desc *, intptr_t *, intptr_t, Info *);
```

Figure 5. STM ABI Changes

source of the abort is obtained by querying either Ct or At. The returned key is used to either create a new RAG-node for the source or an existing node is used if it already exists. Note that a node is thus created only if the corresponding SR is involved in an abort (either as a victim or as an aborter).

An edge is added from the source to the target. If both the source and target nodes already existed and the edge was also present, it is reused and annotations updated.

4.5 STM ABI Extensions

The changes to the STM ABI are shown in Figure 5. Existing interfaces, TxLoad and TxStore are extended to accept a pointer to a structure (referred to as Info in Figure 5) containing $\texttt{SR}_{id}$ and L for the corresponding SR. The rest are new interfaces. Desc refers to the transaction descriptor.

Our current technique requires the above ABI changes. However, it might be valuable to be able to build the RAG without ABI changes since that would permit transparent replacement of STMs without any recompilation. The central requirement of our scheme is that every SR be assigned a program-wide unique identifier. In the presence of whole program compilation, this can be achieved at compile-time. Even in the presence of separate compilation mode (i.e. our current assumption), there may be an efficient implementation that is able to assign unique identifiers with either compiler generated mangled names alone or STM support alone. If no inter-

action between the compiler and STM is required, clearly no ABI changes are required. For example, an STM could hash the program counter for an SR to generate its unique id. But since queries would have to be supported, a hashtable would be required with support for collision resolution. Since this hashtable would be mutable across threads, it would have to be synchronized. Synchronizing the hashtable may be expensive — it appears that the number of synchronization operations executed in such an alternative scheme would be substantially higher than that in the current one. But this is something we have not explored further and has been left for future work.

4.6 Mutual Consistency of the Global Tables

When a transaction is committed, the committing thread releases every ownership record it holds by writing a new version into it (Figure 4, TxCommit, line 14). Just before this step, the corresponding entries in Ct are updated with the $\texttt{SR}_{id}$ of the references to be committed (Figure 4, TxCommit, line 13). The information installed in Ct during TxCommit is queried to get to the aborter in Figure 3, TxLoad, line 13 and Figure 3, TxValidate, line 4. But the query of Ct happens only after a new version is seen in the ownership record which must have been written after the corresponding entry was inserted in Ct. So the Ct-query always sees the updated value and no explicit synchronization is required to maintain mutual consistency of the *orec*-table and Ct.

Now consider mutual consistency of the *orec*-table and At. Subsequent to an *orec*-acquire, At is updated with the $\texttt{SR}_{id}$ of the corresponding reference as shown in Figure 3, TxStore, line 14 and Figure 4, TxCommit, line 7. The information installed in At is queried to get to the aborter when event E_{ll} is encountered as shown in Figures 3 and Figure 4. But note that E_{ll} may be interleaved in between the *orec*-acquire and the update of At. If that happens, the aborter information will be wrong leading to a *false positive*. The solution is to clear the At-entries corresponding to the committed addresses (as shown in Figure 4, TxCommit, line 12) by writing a magic number into them. When an At-query returns a magic number, this case is just ignored and no update to the RAG is made. This implies that our technique may fail to capture some abort patterns leading to *false negatives*.

4.7 Soundness of RAG Construction

We say that the RAG is sound when an edge is added to the graph only when the corresponding source and target SR have been involved in an abort relationship at runtime. The RAG can, however, be incomplete in the sense that some abort relationships are not captured. In other words, the RAG can have false negatives but not false positives. This characteristic can be verified by examining the pseudo-code in Figures 3 and 4. The entry point to adding an edge is UpdateRAG&Abort in Figure 4. Two modes are possible: one when Ct is queried, the other when At is queried. We established in Section 4.6 that neither false positives nor false negatives can happen in the former case and that only false negatives can happen in the latter case.

In order to establish soundness, another factor needs to be examined. The *orec* and the corresponding $\texttt{SR}_{id}$ are stored separately, namely in the *orec*-table and Ct or At respectively. Coupled with this issue is the fact that the global tables are tagless leading to false conflicts. In order to make sure that the *orec* returned by the DOR function for an address corresponds to the *correct* $\texttt{SR}_{id}$ from Ct or At, the same hash function must be used for all the 3 global tables. A returned $\texttt{SR}_{id}$ is correct for an *orec*, o_1, if it is indeed the one that was added to Ct or At when o_1 was added to the *orec*-table. In other words, the same false conflicts, if any, must exist for all the tables.[10]

4.8 Counter-based Instrumentation Sampling

Instrumentation adds runtime overhead. We use compiler-inserted counter-based sampling [2] to reduce this overhead. Essentially, this requires counting a particular event and executing the instrumented code when the counter reaches a threshold as shown below.

```
--counter;
if (counter <= 0) {
   TakeSample(); // i.e. execute instrumented code
   counter = N; // reset to initial value
}
```

Setting N to 1 is equivalent to executing instrumented code every time an event is encountered. In our framework, we maintain distinct thread-specific counters for each event itemized in Section 4.3 (with the exception of E_{ws}, E_{aq}, and E_{ce}). Our scheme avoids the need for any cross-thread synchronization and samples events proportionate to their execution frequencies thus maintaining the desired statistical significance. E_{ws}, E_{aq}, and E_{ce} are always executed ensuring that Ct and At are always updated. This guarantee is necessary for sound construction of the RAG.

4.8.1 Sampling frequency vs RAG accuracy

Introducing instrumentation sampling begs the question about loss of accuracy in the collection of events. Clearly, this is a tradeoff between the runtime overhead of the instrumented code and the fidelity of the abort relationships captured in the RAG. In Section 7.3, we discuss the results we obtained by varying the values of N and arrive at what appears to be a fairly reasonable default that balances overhead and accuracy. In practice, we expect N to be a tuning parameter that may have to be adjusted based on the application.

5. Applications of the RAG

The RAG exposes abort relationships between memory references and hence can be used by any client that seeks to take advantage of such information. Since the memory references are correlated with location information, the RAG can be communicated back to the programmer as advisory information. For an automatic optimization, we believe we need a feedback-driven machinery whereby the RAG is written out into persistent storage in the first phase and the second phase utilizes the persistent RAG to generate an optimized executable. In this paper, we present an automated conflict detection policy inference technique (details in Section 6) that aims to improve runtime performance of the application.

Figure 6 shows how the various components interact. The TM program is fed to the compiler that produces an instrumented executable with the option *-stm=profile*. The instrumented executable, which also links in appropriate routines from the STM, is run to generate a profile database, shown as *prof_db*. The offline analyzer works on *prof_db* to produce tuning and optimization decisions, *opt_info*. Both *prof_db* and *opt_info* contain information meaningful to a programmer. In the second phase, the compiler utilizes *opt_info* to generate an optimized executable.

The profile database is designed to have a number of fairly general parts followed by optimization specific parts. Its current format is shown below.

/* Part 1: RAG as a list of abort relationships */

src:<α, β, γ> tgt:<α, β, γ> count:<N> tgt_t:<ld/str>

[10] If we can devise hash functions that ensure the absence of false conflicts in all of the global tables, then it does not matter — we can then indeed use different hash functions for different global tables.

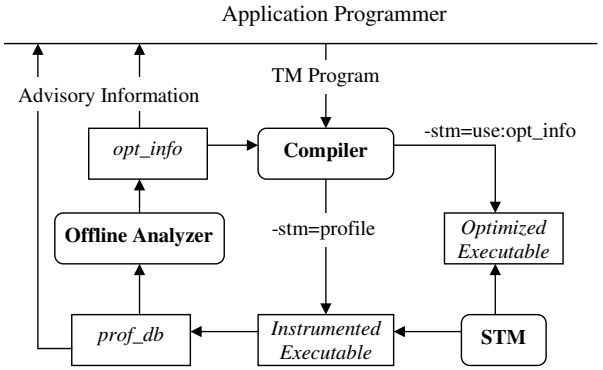

Figure 6. Feedback-driven optimization of a program with atomic sections whereby the instrumented program is run to generate a profile database which is subsequently used by a compiler to generate an optimized executable

/* Part 2: A list of S_{rd} and S_{wr} for references */
$<\alpha, \beta, \gamma>: <S_{rd}> <S_{wr}>$
/* Part 3: A list of source locations for SRs */
$<\beta, \gamma>: <\lambda, \rho, \tau>$
/* Part 4: A list of source locations for atomic sections */
$<\alpha: <\lambda, \rho, \tau>$
/* Part 5: A list of application specific information */
$<\alpha, \beta, \gamma>: <S^s_{rd}> <S^s_{wr}>$

Part 1 captures the core information for the RAG, Part 2 correlates an SR with the corresponding S_{rd} and S_{wr}, Part 3 captures the location information for every SR, and Part 4 captures the location information for every atomic section. Subsequent parts are application specific. For example, the conflict detection policy inference technique described in Section 6 requires the data in Part 5 — S^s_{rd} and S^s_{wr} are speculative readset and writeset sizes and are described later in Section 6.3.3.

6. Conflict Detection Policy Optimization

This section describes how the RAG can be used to drive conflict detection policy optimization.

6.1 Eager and Lazy Acquire

During the execution of a transaction, a conflict may be detected early or late [10]. For every memory address a transaction updates, it must acquire the corresponding *orec*. If the acquisition happens at the first write reference, the policy is called eager. If it occurs during commit, the policy is called lazy. Using optimistic concurrency, a read reference never acquires an *orec*.[11] In the eager mode, a read barrier succeeds if the corresponding location has not been modified since transaction-start and no other thread has acquired the corresponding *orec*, otherwise it aborts. In the lazy mode, the read barrier can proceed by validating its readset even if the location has been modified after transaction-start.

For a given atomic section, it is hard to tell in advance which policy would perform better. With eager policy, wasted work is avoided if the transaction is doomed to fail but on the downside, the *orec* is held longer potentially reducing concurrency. On the other hand, a lazy policy delays *orec* acquisition thereby producing a small contention window only at commit time but can result in a lot of wasted work if the transaction was doomed to fail [15].

[11] We assume that reads are invisible [10].

6.2 Mixed Invalidation

Most STMs detect both read-write and write-write conflicts the eager way (e.g. Intel STM [9]) or the lazy way (e.g. TL2 [6]). Prior work has tried to take the best of both worlds and has experimented with mixed invalidation [7, 15] whereby write-write conflicts are detected eagerly and read-write conflicts are detected lazily.

6.3 Hybrid Acquire (HyAcq)

In prior work, regardless of whether the STM employs eager, lazy, or mixed invalidation, the same policy is used throughout the application (i.e. for all transactional references). Another possible dimension to the solution could be using different policies for different atomic sections in order to maximize performance. This is because not all atomic sections may have the same performance profile and hence allowing them different policies may reduce the number of aborts and total wasted work.

To understand the combinations of policies that may work well, we start from the base STM, which is TL2 in our case. In TL2, vulnerabilities associated with reading inconsistent memory states are avoided by having the read barrier ensure that the *orec* is not held by another thread and that validation succeeds if the *orec* has changed since transaction start. This is the behavior of TL2's read barrier for both eager and lazy policies[12] and we will denote such a read by Rd. For a write, the *orec* is acquired at encounter time in eager policy and at commit time in lazy policy. Let us denote these two behaviors by $\texttt{Wr}_e$ and $\texttt{Wr}_l$ respectively.

6.3.1 Correctness Argument

We define HyAcq as a hybrid conflict detection technique whereby a given atomic section can either use the eager or the lazy policy. This choice can be made in complete isolation from that of any other atomic section. To understand why this is correct, we start from the established correctness guarantee of the base STM [10]. With HyAcq, we need to ensure that interactions between two different atomic sections are correct since one of them can be eager and the other lazy. Any given atomic section is still consistent within itself since it is wholly either eager or lazy. All transactions, regardless of policy, map a given address to the same *orec* table entry. Note also that the behavior of a read has not changed from the base STM to HyAcq. So it essentially boils down to whether interactions between write references are any different. In HyAcq mode, consider two transactions Tx1 and Tx2. Without loss of generality, as far as Tx1 is considered, *orecs* may be acquired in Tx2. It does not matter where in Tx2 *orecs* are acquired. So it does not matter whether Tx2 is using eager or lazy mode. Thus by treating transactions as black boxes, the only visible effect outside a transaction is its timing of *orec* acquires which does not pose any correctness issues.

6.3.2 Locally Preferred Solutions

We now discuss how we proceed with identifying conflict detection policies that improve performance. We first find pair-wise solutions at the reference level and attempt to propagate those decisions to the atomic section level in a way that is globally optimized across the application.

The performance penalty incurred by a transactional reference is determined by the aborts it suffers and the work that is wasted due to an abort. We model this penalty by defining the cost of an SR (or the corresponding RAG-node) as $\mathcal{C}_{sr}{=}A_N{\times}(S_{rd}{+}S_{wr})$, where A_N, S_{rd}, and S_{wr} respectively represent the abort count, the readset size, and the writeset size of the RAG-node. The cost of a RAG-edge is computed as $\mathcal{C}_e{=}A_E{\times}(S_{rd}{+}S_{wr})$, where A_E, S_{rd} and S_{wr} respectively represent the abort count of the edge, the

[12] The original TL2 used lazy policy alone. The adapted TL2 distributed through STAMP [12] implemented both eager and lazy policies.

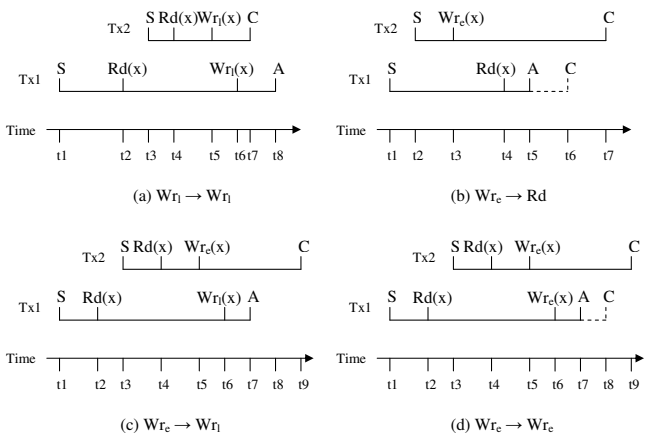

Figure 7. Abort Scenarios where the Rd has a fixed policy but the write could follow eager ($\mathtt{Wr}_e$) or lazy ($\mathtt{Wr}_l$) policy. (a) $\mathtt{Wr}_l$ -> $\mathtt{Wr}_l$ indicates that a lazy write is aborted by another lazy write, (b) $\mathtt{Wr}_e$ -> Rd indicates that a read is aborted by an eager write, (c) $\mathtt{Wr}_e$ -> $\mathtt{Wr}_l$ indicates that a lazy write is aborted by an eager write, and (d) $\mathtt{Wr}_e$ -> $\mathtt{Wr}_e$ indicates that an eager write is aborted by another eager write.

readset size, and writeset size of the target RAG-node. The total cost of the RAG, $\mathcal{C}_{tot}$ is simply the summation of $\mathcal{C}_{sr}$ over all nodes.

For a given SR, we use the *victim-aborter* relationship from Section 2 to develop intuition about the policy that might minimize $\mathcal{C}_{sr}$. Figure 7 shows four scenarios that start with a possible policy combination of the victim and the aborter and recommends a new combination that should reduce the cost of the victim. Note that these are *local point solutions* in the sense that they consider only the cost of the victim (not the aborter) and that too in complete independence of the rest of the victims in the application (i.e. purely local). Later on, Section 6.3.3 explains how to derive a globally optimized solution.

In each of the scenarios in Figure 7, we show the progression of two transactions (Tx1 and Tx2) on a time scale. Tx1 is the victim and Tx2 is the aborter in every scenario. In Figure 7(a), both transactions use the lazy policy, the transactional write to x in Tx1 happens at t6, the transactional write to the same location in Tx2 happens at t5, and Tx2 commits before Tx1 at t7. Since Tx1 fails readset validation in the commit phase at t8, it aborts. As shown in the figure, Tx1 could be a long transaction with a large amount of wasted work. If its cost $\mathcal{C}$ turns out to be large, it may be beneficial for Tx1 to have its transactional write execute in an eager mode, so that it acquires ownership of x at t6 allowing it to successfully commit at t8. Note that this potentially causes Tx2 to fail (because it continues using the lazy policy) but at this point we are interested in optimizing the cost of the victim alone.

Consider Figure 7(b) where Tx1 executes a read to location x at t4 and Tx2 executes an eager write to the same location at t3. Tx1 aborts at t5 since at that point Tx2 has ownership of x. However, if Tx1 is a relatively short transaction as suggested in the figure, it may be beneficial to run Tx2 in lazy mode potentially allowing both transactions to commit successfully.

The scenario shown in Figure 7(c) involves a lazy write in Tx1 and an eager write in Tx2. Since Tx2 acquires ownership of x at t5, Tx1 detects at t7 (during the commit phase) that the *orec* is held and aborts. Just by considering the cost of Tx1 in isolation, it may be beneficial to run Tx1 in eager mode and Tx2 in lazy mode, likely allowing Tx1 to commit.

Initial Aborter -> Victim	Optimized Aborter -> Victim
1. $\mathtt{Wr}_l$ -> Rd	No change
2. $\mathtt{Wr}_l$ -> $\mathtt{Wr}_l$	$\mathtt{Wr}_l$ -> $\mathtt{Wr}_e$
3. $\mathtt{Wr}_l$ -> $\mathtt{Wr}_e$	No change
4. $\mathtt{Wr}_e$ -> Rd	$\mathtt{Wr}_l$ -> Rd
5. $\mathtt{Wr}_e$ -> $\mathtt{Wr}_l$	$\mathtt{Wr}_l$ -> $\mathtt{Wr}_e$
6. $\mathtt{Wr}_e$ -> $\mathtt{Wr}_e$	$\mathtt{Wr}_l$ -> $\mathtt{Wr}_e$

Table 1. All possible locally preferred solutions

The last scenario in Figure 7(d) considers eager writes in both the transactions. Assuming Tx1 is relatively short, has a large number of aborts, and considering its cost in isolation, it may be beneficial to run Tx2 in lazy mode, allowing Tx1 to commit.

To identify the possible scenarios, consider the aborter to victim relationship, captured by $\mathtt{P}_a\mathtt{T}_a$ -> $\mathtt{P}_v\mathtt{T}_v$, where $\mathtt{P}_a$ and $\mathtt{P}_v$ denote the policies of the aborter and the victim respectively and $\mathtt{T}_a$ and $\mathtt{T}_v$ denote the memory reference type (read/write) of the aborter and the victim respectively. Since we are considering only eager and lazy policies and a reference could be a read or a write, there can be 16 scenarios. Since the aborter cannot be a read, we are left with 8. Since we do not model any difference between an eager and a lazy read, we are left with just 6 scenarios listed in the first column of Table 1. The second column lists the optimized policies of the aborter and the victim considering the latter in isolation and only when the cost of the latter justifies doing so.

6.3.3 Globally Optimized Policies at the Atomic Section Level

Given a RAG and a table of locally preferred solutions, the problem is finding the policy for every atomic section that minimizes the total cost of the RAG. We use a greedy algorithm to solve this problem. Before we describe our technique, we need to introduce some useful concepts.

Node Criticality: A RAG node is considered *critical* if its cost exceeds a certain threshold $\mathcal{T}$. Our technique works by primarily optimizing the critical nodes through identifying policies that minimize their cost.

RAG-edge adjustment factor, $\mathcal{F}$: $\mathcal{F}$ denotes the number that is used to multiply the abort count of a RAG-edge when the policy of either the aborter or the victim is changed. The abort counts obtained from *prof_db* introduced in Section 5 are for scenarios where all references follow the same policy. In order to estimate whether a different policy for a certain node reduces the cost of the RAG, we need to have estimates for a RAG where some nodes could be eager and some nodes could be lazy. Since such experimental data is not available from a given profile run using the base STM, we heuristically come up with adjustment factors guided by the locally preferred solutions. The values of $\mathcal{F}$ used in our experiments are shown in Table 2 in Section 7.

Speculative rdset and wrset sizes: The rdset and wrset sizes are used in estimating wasted work when an abort happens. Similar to abort counts, our technique needs new rdset and wrset sizes for a RAG-node when policy changes are effected. However, in contrast with abort counts, it is possible to obtain sufficiently accurate estimates of rdset and wrset sizes for a different policy from a single profile run. Let us extend the notations for the readset and writeset sizes by using S^e_{rd} and S^e_{wr} for eager policy and S^l_{rd} and S^l_{wr} for lazy policy. Consider the following cases:

- A read reference is the target of a RAG-edge that has its abort count adjusted: In this case, $S^e_{rd} = S^l_{rd}$ and $S^e_{wr} = S^l_{wr}$ since the behavior of a read does not change across policies.
- A write reference is transitioned from eager to lazy acquire: The required S^l_{rd} and S^l_{wr} are obtained by augmenting the profiling

```
1: Optimize() {
2:    T = ComputeCriticality();
3:    Q = BuildPriorityQueue(T);
4:    while (!Q.empty()) {
5:       Vertex tgt = Q.top();
6:       Q.pop();
7:       if (!isValid(tgt)) continue;
8:       vector edge_vec = GetSortedCandidates(tgt);
9:       Policies new_policies = Analyze(edge_vec); /* category 1 */
10:      Vertices adj_nodes = AnalyzeAdjNodes(new_policies); /* category 2 */
11:      total_cost_change = ComputeCostChange(new_policies, adj_nodes);
12:      if (total_cost_change < 0) {
13:         foreach node v in new_policies and adj_nodes {
14:            update policy in RAG if v.kind == category 1;
15:            if (abort count has not changed) continue;
16:            update abort count and criticality in RAG;
17:            if (IsProcessed(v)) continue; /* don't add to Q */
18:            if(v != tgt && ComputeNewCost(v) > T) Q.push(v);
19:         }
20:      }
21:      IsProcessed[tgt] = true;
22:   }
23: }
```

(a) Main Driver Routine

```
1: Analyze(vector edge_vec) {
2:    Policies policies; /* initialize */
3:    for every edge in edge_vec {
4:       PointSoln ps = hasPointSoln(edge);
5:       if (ps && !policies.find(edge.source) && IsASMutable(edge.source) &&
6:          !policies.find(edge.target) && IsASMutable(edge.target)) {
7:          policies.insert(ps, edge.source); MutateAS(edge.source, policies);
8:          policies.insert(ps, edge.target); MutateAS(edge.target, policies);
9:       }
10:   }
11:   return policies;
12: }
```

(b) Speculative Build of New Policies

```
1: ComputeCostChange(Policies new_policies, adj_nodes) {
2:    old_cost = new_cost = 0;
3:    for every node v in new_policies and adj_nodes {
4:       if (v.kind == category 1) {
5:          np = GetNewPolicy(v, new_policies);
6:          op = GetPolicyFromRAG(v);
7:       }
8:       else np = op = GetPolicyFromRAG(v);
9:       for every in_edge(v) {
10:         Lookup F from table;
11:         new_abort_count = old_abort_count × F;
12:         old_cost += old_abort_count × (S_rd^<op> + S_wr^<op>);
13:         new_cost += new_abort_count × (S_rd^<np> + S_wr^<np>);
14:      }
15:   }
16:   return new_cost - old_cost;
17: }
```

(c) Cost Change Computation

Figure 8. Global Optimization of Policies

support to collect the readset and writeset sizes at commit time (in addition to collecting them at encounter time aborts).

- A write reference is transitioned from lazy to eager acquire: The required S^e_{rd} and S^e_{wr} are obtained by augmenting the profiling support to collect the readset and writeset sizes at write encounter times (even though the lazy transactions do not abort at encounter times).

Modification of the profiling phase as described above results in Part 5 mentioned in Section 5 and ensures that every SR will have corresponding values of S^e_{rd}, S^e_{wr}, S^l_{rd}, and S^l_{wr}.

Key Data Structures: In addition to the RAG, a key data structure used during global optimization is a priority queue, called $\mathcal{Q}$. The critical references are maintained in $\mathcal{Q}$, allowing them to be examined in the order of decreasing cost.

The Algorithm: Figure 8 shows the main driver routine and a couple of helper functions used for choosing policies of atomic sections. The central idea is to examine the critical transactional references for policy changes in decreasing order of their cost function. If a locally preferred solution for a transactional reference is found, the potential change in cost for applying that policy to all transactional references within that atomic section is computed. If found beneficial, the resulting policy changes are committed to the RAG.

The driver routine starts off by computing the critical threshold and building $\mathcal{Q}$ based on this threshold. The *while* loop in line 4 examines the nodes in $\mathcal{Q}$ giving priority to a higher $\mathcal{C}_{sr}$. Every time a vertex is popped off $\mathcal{Q}$, a *round* of processing is initiated. Given a candidate node, all edges incident on it are obtained in a sorted order with descending $\mathcal{C}_e$ (Figure 8(a), line 8). They are then examined in this sorted order in the routine Analyze when searching for beneficial policies. If a *locally preferred* solution is found, two conditions need to be satisfied (for both the source and target nodes of the examined edge) before proceeding (Figure 8(b), lines 5-6). The first check (let us call it C1) returns true if a new policy has not already been assigned to the node during this round — an earlier policy assignment during this round is given higher priority since it is associated with a higher $\mathcal{C}_{sr}$. The second check (let us call it condition C2) bridges the gap between policy assignment at the reference and the atomic section level. Note that eventual policy change happens at the atomic section level, not at the SR level. The call to IsASMutable (i.e. C2) for a given node returns true if either of the following holds for every node in the corresponding atomic section: (1) C1 is true (2) There is no conflict between the policies determined by a previous round and this round. MutateAS caches the new policies of all the nodes in the corresponding atomic section so that they can be committed to the RAG later on (if so determined).

The nodes that are potentially affected in a given round can be of two types, referred to as *categories 1 and 2* in Figure 8(a). The first category contains nodes that have a change in policy in a given round and the second contains nodes that do not have a change in policy in that round but are adjacent to at least one node in category 1. The cost can change for each of the nodes in either category and the change in $\mathcal{C}_{tot}$ is computed as in Figure 8(c).

If the change in cost shows that the new policy assignment is beneficial, the following steps are performed: (i) The new policies of category 1 nodes and the updated abort counts and other attributes of nodes in both categories are reflected in the RAG. (ii) If a node in either category has not already been processed and has an updated cost higher than the critical threshold, it is pushed into the priority queue. This step has an important effect on how the entries in $\mathcal{Q}$ are processed. Note that as a round completes, some entries in $\mathcal{Q}$ might have an updated abort count and other attributes. However, there is no efficient way to update these entries in $\mathcal{Q}$. Instead the corresponding changes are made in the RAG. Every time an entry is obtained from $\mathcal{Q}$, it is validated against the authoritative entry in the RAG (Figure 8(a), line 7).

The following characteristics of the algorithm are worth noting:

- Multiple entries corresponding to the same node may be present in $\mathcal{Q}$ but Figure 8(a), line 7 ensures that the only valid one is ever processed, others are ignored.
- A processed node is never added to $\mathcal{Q}$ ensuring that there can be a maximum of n rounds (where n is the number of nodes in the RAG) thus guaranteeing termination.
- There are O(n) nodes in $\mathcal{Q}$ at the start of the algorithm. At the end of a round, O(n) nodes can be added to $\mathcal{Q}$. Since there can be a maximum of n rounds, the maximum number of nodes in $\mathcal{Q}$ at any point of time is $O(n^2)$.
- A critical node that has its cost updated to a value lower than $\mathcal{T}$ does not get removed from $\mathcal{Q}$ but will be ignored because of the validity check on Figure 8(a), line 7.
- $\mathcal{C}_{tot}$ is non-increasing by construction of the algorithm.

6.4 Feeding Optimization Decisions back to the Compiler

Section 4 describes the identifiers and the technique we use for assigning them. Since the values assigned depend on the order in which atomic sections and functions get executed, these identifiers may be different from one run to another causing a matching problem. Hence, if we want to use profile data from across runs (such as in merging profile data obtained from different runs), a matching process is required. Given two profile databases, the matching process is necessary to identify all the information corresponding to the same `SR` though such information may be tagged with different identifiers in two different executions. The current solution is to use the source position to perform the matching. The assumption is that no two atomic sections have the same source position — hence any two atomic sections, that are obtained from different runs, with the same source position are one and the same.

Another issue related to the above matching problem occurs when feeding back the results of `HyAcq` to the second compilation phase (see Section 5). Notice that the only information required is the policy that is to be followed for a given atomic section. Currently, this handshake is implemented by having the offline analyzer create a file containing the policy of an atomic section, the latter identified with its source position. Thus, the format is simply

```
<source position of atomic section> <policy>
...
```

For example, the file contents for genome may look like

```
line 290 lazy
line 369 lazy
line 395 eager
line 408 lazy
line 476 lazy
```

When `-stm=use:opt_info` is used (as shown in Figure 6), the compiler reads the above file and, while lowering an atomic section into an intermediate representation, follows the policy indicated if its source position matches a provided entry (otherwise the default is used).

7. Experimental Results

We implemented the `STM` extensions on top of TL2 [6] and evaluated our techniques on the STAMP [12] benchmark suite (v 0.9.10). The applications span a variety of fields including machine learning, security, and data mining. This suite contains 8 benchmarks but one of them (bayes) fails on 64-bit machines.[13] Hence we report our results on the other 7 benchmarks. Unless otherwise mentioned, we used the larger (non-simulator) input and the high contention parameters (whenever available). We performed all our experiments on a shared memory machine (running Red Hat Enterprise Linux) with 4 quad-core sockets and 32 GB of memory. Each socket has an Intel(R) quad-core Xeon(R) E7330 CPU running at 2.4GHz and with 6MB of shared L2. Any result reported is an average of 4 runs with the same input parameters.

7.1 Experimental Methodology

As described in Section 5, profile data obtained by running an instrumented executable is used by an offline analyzer and the results of the analysis are fed to the second compile. We first show the overheads involved in building the `RAG` and discuss cost vs accuracy overheads when using instrumentation sampling.[14] Subsequently, we describe the policy configurations recommended by the offline analysis tool and examine the performance changes obtained by re-running the applications using the new policies.

For the results discussed in this paper, we always start from either application-wide eager or lazy policy. However, there is nothing inherent in our technique that precludes us from starting with a hybrid policy configuration.

Our experiments are evaluated on 1, 2, 4, 8, and 16 processors. We discuss the effects of *training*[15] the application on various processor and input configurations. The offline analysis time is at most a couple of seconds given any benchmark we tested.

Section 6.3.3 mentioned that we use heuristically determined values for $\mathcal{F}$. Table 2 shows the mapping from state transitions (for an edge in the `RAG`) to values used in our experiments. A state transition happens when given an edge in the `RAG`, either the source or the target or both have a change in policies.

Current State	New State	$\mathcal{F}$
Wr_l -> Wr_l	Wr_l -> Wr_e	0.50
Wr_e -> Wr_l	Wr_l -> Wr_e	
Wr_e -> Wr_e	Wr_l -> Wr_e	
Wr_l -> Wr_l	Wr_e -> Wr_e	0.75
Wr_e -> Wr_l	Wr_e -> Wr_e	
Wr_e -> Wr_l	Wr_l -> Wr_l	
Wr_l -> Rd	Wr_e -> Rd	1.00
Wr_e -> Rd	Wr_l -> Rd	

Table 2. Heuristically determined values of $\mathcal{F}$. The transitions not shown here use the value 2.0.

7.2 Establishing Baselines

Since instrumentation has its overheads, we would like to compare our scheme with some baselines that have negligible or very low overhead. It is sufficient to measure the possible perturbation of the original behavior of the application and in this case, the total execution time and the number of aborts are good indicators. Let us establish the *level 1* baseline (or `L1`) that measures just these two attributes and is known to have negligible overhead. However, `L1` does not provide any fine grained information that would be required to evaluate the accuracy of schemes that collect a lot more application characteristics (such as the `RAG`). So we establish the *level 2* baseline (or `L2`) where a limited amount of fine grained information is collected but with minimum overheads. `L2` collects total runtimes, total number of aborts, and abort counts for every reference that aborts at least once. Table 3 tries to show that `L2` indeed has minimum overhead. For this purpose, we respectively define $|T_d|$ and $|A_d|$ as the absolute deviation of total runtimes and total abort counts of `L2` with respect to `L1` in percentage. As Table 3 shows for 1, 2, 4, 8, and 16 processor counts, both these metrics are only a few percentages across the applications and hence `L2` can be assumed to be a reliable baseline with which more elaborate techniques like building the `RAG` can be compared.

7.3 Evaluating RAG Building Overheads vs Accuracy

The overheads of building the `RAG`, evaluated by comparing $|T_d|$ and $|A_d|$ with respect to `L1`, should not be prohibitive (let us call it requirement 1). But the `RAG`'s accuracy needs to be evaluated as well. In other words, we need to ensure that the abort characteristics it captures belong to the uninstrumented application. Towards that goal, we constrain the `RAG`-build to collect a superset of the data that `L2` collects. Next, we would like to verify whether there is a high degree of correlation in the data common to `L2` and the `RAG`-build (let us call it requirement 2). If both requirements 1 and

[13] This is a known bug and apparently has been fixed in the latest release but it still continues to fail on our systems.

[14] This set of results has been obtained using the lazy policy throughout a given application.

[15] The input used to generate the `RAG` is called the training input.

Bmarks	1p	2p		4p		8p		16p		Avg	
	$\|T_d\|$	$\|T_d\|$	$\|A_d\|$	$\|T_d\|$	$\|A_d\|$	$\|T_d\|$	$\|A_d\|$	$\|T_d\|$	$\|A_d\|$	$\|T_d\|$	$\|A_d\|$
genome	4.8	9.6	14.8	6.5	5.7	1.8	0.4	4.2	5.6	5.4	6.6
kmeans	5.2	10.3	7.5	8.2	1.9	7.8	2.3	6.5	2.6	7.6	3.6
intruder	3.4	3.3	6.6	3.5	5.2	3.5	2.9	1.2	3.2	3.0	4.5
labarinth	0	0.3	2.5	0.1	6.4	0.5	8.9	12.6	10.5	2.7	7.1
vacation	4.4	4.9	15.4	2.4	9.8	4.7	10.9	5.6	10.8	4.4	11.7
ssca2	1.6	3.5	3.0	2.9	2.9	0.1	3.6	0.5	6.5	1.7	4.0
yada	1.9	1.7	3.7	4.6	5.2	1.4	4.2	2.4	6.0	2.4	4.8
Avg	3.0	4.8	7.6	4.0	5.3	2.8	4.7	4.7	6.5	3.9	6.0

Table 3. Comparison of Baselines: T_d and A_d respectively measure L2's absolute additional cost in total runtimes and total aborts in percentage compared to L1.

	L2		rag-1		rag-5		rag-10		rag-20		rag-50	
Bmarks	$\|T_d\|$	$\|A_d\|$	$\|T_d\|$	$\|A_d\|$	$\|T_d\|$	$\|A_d\|$	$\|T_d\|$	$\|A_d\|$	$\|T_d\|$	$\|A_d\|$	$\|T_d\|$	$\|A_d\|$
genome	5.4	6.6	22.7	16.7	20.8	15.0	28.8	14.3	13.7	13.7	11.4	12.2
kmeans	7.6	3.6	14.4	29.5	28.7	5.0	35.1	5.4	25.3	4.5	11.7	5.4
intruder	3.0	4.5	17.5	9.6	10.9	12.2	12.1	8.2	13.4	7.9	10.3	7.3
labyrinth	2.7	7.1	7.1	3.3	3.8	5.2	3.9	7.6	8.4	8.6	10.3	19.8
vacation	4.4	11.7	37.5	39.3	6.4	31.3	6.2	27.1	7.2	25.0	6.0	21.5
ssca2	1.7	4.0	16.5	5.6	8.4	4.9	8.7	5.6	8.5	2.9	8.6	6.4
yada	2.4	4.8	25.2	43.6	17.9	33.6	17.2	25.9	9.7	10.5	22.1	26.2
Avg	3.9	6.0	20.1	21.1	13.8	15.3	16.0	13.4	12.3	10.4	11.5	14.1

Table 4. Comparing the perturbation of various configurations: T_d and A_d respectively measure the absolute additional cost in total runtimes and total aborts in percentage, compared to L1. The configurations denoted by rag-N differ with respect to the sampling parameter N introduced in Section 4.8.

2 hold, we can say with reasonable certainty that the RAG-build closely matches the runtime characteristics of the uninstrumented application.

Table 4 shows $|T_d|$ and $|A_d|$ (with respect to L1), averaged over 1, 2, 4, 8, and 16 processor counts, for various RAG-build configurations and also for L2 for ready reference. rag-N denotes building the RAG with counter-based instrumentation sampling turned on with the sampling parameter N as discussed in Section 4.8. The larger the value of N, the fewer the number of events captured. The Avg row shows the average of the runtime perturbation for a given RAG-build configuration over all applications. When all events are captured (i.e. rag-1), the average runtime and abort overheads are 20.1% and 21.1% respectively. The overheads are reduced somewhat when fewer events are captured as evidenced by the average perturbation of rag-5, rag-10, rag-20, and rag-50. We feel that an overhead of less than 20% is not unreasonable and that requirement 1 appears to be satisfied. The Avg row indicates that the overheads drop significantly going from rag-1 to rag-5 but beyond N=5, any such drop tapers off. Hence, it appears that from an overhead standpoint, any value of N equal to or above 5 is appropriate to use. However, using N=5 still leads to capturing a lot of events and that may experience large overheads in general — so in the following results, we ignore rag-5. We ignore rag-50 as well since there is negligible overhead reduction beyond N=20 and there is the risk of losing important events.

Table 5 tries to address requirement 2 by examining the accuracy of abort information for 4-processor runs.[16] We always tabulate results for rag-1 for accuracy measurement purposes since this configuration captures *all* events. Additionally, results for rag-10 and rag-20 are presented, others are ignored for reasons mentioned above. Let us define an *important* reference as one that has at least

	rag-1		rag-10		rag-20	
Bmarks	N_c	$\|A_c\|$	N_c	$\|A_c\|$	N_c	$\|A_c\|$
genome	0	11	0	7	0	5
kmeans	0	10	0	11	0	13
intruder	0	10	0	9	0	9
labyrinth	0	4	0	6	0	2
vacation	1	29	1	16	0	16
ssca2	0	16	1	12	1	10
yada	1	26	0	6	0	1
Avg	0	15	0	10	0	7

Table 5. Comparing Accuracy of Abort Information of various RAG-build configurations with respect to baseline L2 (on 4 processors)

c% of all aborts.[17] We define N_i as the number of important references that are different between L2 and a given rag-N run. We define A_i as the absolute deviation in aborts, between L2 and a rag-N run, computed in percentage over the top 10 important references. As shown in Table 5, there is very little difference when it comes to the important references — practically all of them are captured by all the three configurations. Additionally, the deviation in the abort count is small for any given configuration. Thus, requirement 2 appears to be satisfied.

Notice that A_i reduces as fewer events are captured indicating that the relatively higher instrumentation overhead of rag-1 does affect the abort patterns. As mentioned in Section 4.8, the sampling parameter N should be tunable but with a default value. From our results, it appears that a value between 10 and 20 would be an appropriate default.

Ultimately, a RAG's usefulness should be measured by the amount of optimization it can drive. In this regard, we show some

[16] Results for other processor configurations are similar and not shown here.

[17] We use c=5 for the results in Table 5.

Stats		Benchmarks				
		genome	kmeans	intruder	vacation	yada
rag-1	V	14	6	37	52	306
	E	13	6	88	186	173
	H	3	5	14	17	13
rag-10	V	8	6	25	34	204
	E	8	6	45	81	87
	H	5	5	16	13	14
rag-20	V	4	6	23	30	171
	E	4	6	37	68	66
	H	3	5	14	14	13

Table 6. RAG statistics for different build configurations on 4 processors. V, E, and H denote respectively the number of vertices in the RAG, the number of edges in the RAG, and the initial size of the priority queue maintained by the offline analysis tool.

Start Policy	Profile Parameter	Benchmarks			
		genome	intruder	vacation	yada
Lazy	N_1P_4	G1	I1	V1	Y1
	$N_{10}P_4$	G1	Eager	V1	Y1
	$N_{20}P_4$	Lazy	I1	Lazy	Y1
	N_1P_8	G1	I1	V1	Y1
	$N_{10}P_8$	G1	I1	V1	Y1
	$N_{20}P_8$	Lazy	I1	Lazy	Y1
Eager	N_1P_4	G2	I2	Eager	Y2
	$N_{10}P_4$	G2	I3	Eager	Y3
	$N_{20}P_4$	G2	I2	Eager	Y3
	N_1P_8	G3	I2	Eager	Y2
	$N_{10}P_8$	G3	I3	Eager	Y3
	$N_{20}P_8$	G2	I2	Eager	Y3

Table 7. Recommended HyAcq Configurations. With an initial state either lazy or eager throughout and with different profiling parameters, the configurations obtained are shown. The profile parameter is N_nP_p where n=sampling parameter and p=processor count used to generate the RAG.

statistics of the RAG and the priority queue maintained by the offline analysis tool for various build configurations. The goal is to understand the kind of loss of information as fewer events are captured. In Table 6, V, E, and H denote respectively the number of vertices in the RAG, the number of edges in the RAG, and the initial size of the priority queue maintained by the offline analysis tool. The results presented are for 4-processor executions. labyrinth and ssca2 have very few aborts, of the order of a few hundreds, and our technique does practically nothing on them. Hence, these two benchmarks are excluded from further discussion of results. Even though V and E change from one configuration to another, H pretty much stays the same. Note that H refers to the number of critical references in the priority queue and serves as the driver of the HyAcq optimization. Thus, Table 6 indicates that the important characteristics are maintained even as fewer events are captured with instrumentation sampling.

7.4 Impact of HyAcq

Before we examine the performance impact of HyAcq, we need to identify the policy recommendations it generates. As noted before, we are considering builds of the RAG with sampling parameter values of 1, 10, and 20. This sampling parameter determines the rate at which events are captured. However, there is another parameter that impacts the RAG-build, the number of processors on which the instrumented executable is run to obtain the RAG. Consider that an application may experience different kinds of contention depending on the number of processors and this difference in runtime behavior may lead to a different RAG. Hence, we define a profile parameter as the combination of two factors: the sampling parameter and the processor count. This is denoted by N_nP_p in Table 7 where n is the sampling parameter and p is the processor count used to generate the RAG. For example, N_1P_4 indicates a RAG-build configuration where all events are captured and the RAG is built using data from a 4-processor run. We also consider another dimension of the problem: the initial policy of the application used to build the RAG and the same policy that is optimized by using the RAG. As shown in Table 7, we always start from either application-wide eager or lazy policy. However, there is nothing inherent in our technique that precludes us from starting with a hybrid policy configuration. Note that kmeans has been excluded from Table 7 since the recommendation for all configurations is to use the eager policy throughout.

It turns out that for a given application, a small set of optimized configurations were recommended (the specifics of every new configuration appear after this paragraph). For example, for genome, the recommended configurations are G1, lazy throughout, G2, and G3. The results are similar for other benchmarks. A small set of recommended configurations indicates a degree of convergence. However, note that the offline analysis tool does not aim to achieve convergence, its only driver is improved performance. For example, if we start with lazy policy and a certain atomic section does not impact performance, our technique would leave it untouched. On the other hand, if we start with eager policy, the same atomic section might be left untouched again. Thus we may not have convergence between results starting with eager and lazy policies. Here are the details of the new configurations where an atomic section is denoted by its corresponding identifier, i.e. α. Note that genome has 5 atomic sections, intruder 3, vacation 3, and yada 6.

```
genome:
0: sequencer.c, line 290
1: sequencer.c, line 369
2: sequencer.c, line 395
3: sequencer.c, line 408
4: sequencer.c, line 476
G1 = [Lazy: 0-1, 3-4 Eager: 2]
G2 = [Lazy: 0, 2 Eager: 1, 3-4]
G3 = [Lazy: 0 Eager: 1-4]
-------------------------
intruder:
0: intruder.c, line 199
1: intruder.c, line 210
2: intruder.c, line 226
I1 = [Lazy: 2 Eager: 0-1]
I2 = [Lazy: 0 Eager: 1-2]
I3 = [Lazy: 1 Eager: 0, 2]
-------------------------
vacation:
0: client.c, line 196
1: client.c, line 247
2: client.c, line 267
V1 = [Lazy: 1-2 Eager: 0]
-------------------------
yada:
0: yada.c, line 207
0: yada.c, line 215
0: yada.c, line 228
0: yada.c, line 233
0: yada.c, line 246
0: yada.c, line 254
Y1 = [Lazy: 1, 3-5 Eager: 0, 2]
Y2 = [Lazy: 2 Eager: 0-1, 3-5]
Y3 = [Lazy: 2, 4 Eager: 0-1, 3, 5]
---------------------------------
```

Tables 8, 9, and 10 show the performance characteristics of the initial configurations, eager and lazy. These will serve as baselines when we present numbers for the recommended configurations. The presented runtimes are in seconds, the aborts are in thousands,

Bmark	Eager					Lazy				
	1p	2p	4p	8p	16p	1p	2p	4p	8p	16p
genome	25.4	15.5	12.9	22.0	31.9	27.0	16.5	9.3	5.3	3.6
intruder	66.7	43.8	28.1	22.0	19.1	74.4	48.4	31.2	24.2	39.0
vacation	75.4	45.6	24.1	14.4	10.8	108.8	59.2	33.1	17.8	10.6
yada	24.6	22.1	16.3	15.3	13.8	39.1	29.3	19.3	14.0	11.5

Table 8. Absolute Runtimes (in seconds) on 1, 2, 4, 8, and 16 processors

Bmark	Eager				Lazy			
	2p	4p	8p	16p	2p	4p	8p	16p
genome	4.7	90.9	330.9	701.9	5.6	11.9	27.1	56.8
intruder	948.0	3884.1	12889.3	14568.3	1257.0	4568.1	13183.1	15073.5
vacation	5.9	15.1	33.9	82.1	5.4	16.3	36.8	73.5
yada	3356.8	7214.3	12293.4	15207.5	1497.3	2373.8	3009.9	2800.0

Table 9. Absolute Aborts (in thousands) on 2, 4, 8, and 16 processors

Bmark	Eager				Lazy			
	2p	4p	8p	16p	2p	4p	8p	16p
genome	10	29	100	638	10	26	62	126
intruder	81	280	867	1154	140	529	1464	1687
vacation	0.7	2	5	12	0.9	3	7	14
yada	50	92	133	163	60	113	153	187

Table 10. Absolute Total Measured Cost in millions on 2, 4, 8, and 16 processors

and the total cost $\mathcal{C}_{tot}$ is in millions. The cost is computed from the aborts and the readset/writeset sizes by running the applications in eager and lazy modes. Table 8 shows that excepting genome in eager mode and intruder in lazy mode, there is a reduction in execution times as the number of processors is increased to 16. As shown in Table 9, given either eager or lazy configuration, the number of aborts goes up with the number of processors indicating an increase in contention. The number of aborts may not always correlate with the execution time since the former may not be indicative of the performance bottlenecks. In addition to the aborts, wasted work needs to be accounted for as well. For example, though the execution times for yada in eager mode are better than those in lazy mode for 2 and 4 processor configurations and worse for 8 and 16 processor configurations, the number of aborts presents a different picture. However, the total cost $\mathcal{C}_{tot}$ shown in Table 10 is more consistent with the execution time. Specifically, for yada, the values of $\mathcal{C}_{tot}$ track the execution times better than aborts (though it fails to capture the cross-over at 8-processor count). This trend justifies our decision to use the cost as the driver of `HyAcq`.

Table 11 shows performance numbers for the optimized configurations in terms of ratios. In addition to the execution time, we also show data on how the number of aborts and the cost of the atomic sections change with the optimized configurations. Let us denote an optimized configuration by $\mathcal{O}$ and the corresponding total execution time by $\mathrm{T}_{\mathcal{O}}$, the total number of aborts by $\mathrm{A}_{\mathcal{O}}$, and the total cost of the atomic sections (as defined in Section 6.3.2) by $\mathrm{C}_{\mathcal{O}}$. Let $\mathrm{T}_{\mathcal{I}}$, $\mathrm{A}_{\mathcal{I}}$, and $\mathrm{C}_{\mathcal{I}}$ be the total execution time, the total number of aborts, and the total cost of the atomic sections respectively for the corresponding initial state. The initial states are available from Table 7. For example, the initial state for configuration G1 is lazy and hence for G1's results in Table 11, $\mathrm{T}_{\mathcal{I}}$, $\mathrm{A}_{\mathcal{I}}$, and $\mathrm{C}_{\mathcal{I}}$ respectively refer to the total execution time, the total number of aborts, and the total cost of the atomic sections when genome is run in lazy mode. We can now define the notations used in Table 11. For a given configuration $\mathcal{O}$, $\mathrm{T}_r = \frac{T_{\mathcal{O}}}{T_{\mathcal{I}}}$, $\mathrm{A}_r = \frac{A_{\mathcal{O}}}{A_{\mathcal{I}}}$, and $\mathrm{C}_r = \frac{C_{\mathcal{O}}}{C_{\mathcal{I}}}$. In Table 11, a number lower than 1 indicates improvement.

Consider T_r in Table 11. For genome, most configurations show improvements, the largest one being 90% for G2 and G3 on 16 processors. For intruder, some configurations show a 10% benefit while some show a 10% slowdown, though there is an overall improvement. In vacation, there can be a 20% improvement and there is no slowdown. `yada` is very interesting in that some of the 2 and 4 processor configurations have slowdowns but the same configurations achieve a speedup on 8 and 16 processors.

Now consider A_r in Table 11. Note that there may not always be a correlation between T_r and A_r. For example, though `vacation` runtimes always improve, some configurations show an increased number of aborts. This is because `HyAcq` optimizes the total cost which takes into account both the number of aborts and the total wasted work. In some cases, the aborts may increase but the total costs decrease (such as in vacation). For genome, the maximum reduction in aborts is 90%, for intruder 30%, and for yada 80%. For vacation, the abort count increased in some cases, by upto 40%. For yada, some `HyAcq` configurations generated higher aborts and slower runtimes, but very interestingly the same executable achieves lower aborts and better runtimes at 8 and 16 processor counts.

Consider C_r in Table 11. Note that `HyAcq` attempts to reduce $\mathcal{C}$ using an estimation technique and the data in Table 11 illustrates how successful the estimation technique is. In almost all cases, T_r is consistent with C_r meaning that an increase in one is not accompanied with a decrease in another.[18] This shows that using $\mathcal{C}_{tot}$ to model the total execution time is reasonable. A very related question is whether `HyAcq` is able to accurately reduce $\mathcal{C}_{tot}$. As Table 11 shows, $\mathcal{C}_{tot}$ does not increase in general, the major outliers being I3, Y2, and Y3. It appears that there are at least a couple of reasons for a possible increase in $\mathcal{C}_{tot}$. The first relates to the use of heuristically determined values shown in Table 2 — more experimentation and insight are necessary to come up with more accurate values. The second is the way we use the `RAG` — `HyAcq` implicitly assumes that no new abort edges are introduced by change of policies. This is not always true. We found that in yada, new abort relationships get introduced by the recommended policy changes but `HyAcq` has no way to account for the costs on those new `RAG`-edges.

[18] The only exception is yada on 8 and 16 processors.

Bmark	1p	2p			4p			8p			16p			Avg		
Confs.	T_r	T_r	A_r	C_r	T_r	A_r	C_r	T_r	A_r	C_r	T_r	A_r	C_r	T_r	A_r	C_r
G1	0.9	0.9	1.1	0.9	0.9	0.9	0.9	0.9	0.9	0.8	0.9	0.8	0.8	0.9	0.9	0.9
G2	1.0	1.1	1.0	1.1	0.6	0.1	0.9	0.2	0.1	0.6	0.1	0.1	0.2	0.6	0.3	0.7
G3	1.0	0.9	1.0	1.0	0.6	0.1	0.8	0.2	0.1	0.5	0.1	0.1	0.2	0.6	0.3	0.6
I1	0.9	0.9	0.7	0.6	0.9	0.8	0.5	0.9	0.9	0.6	0.5	0.9	0.6	0.8	0.8	0.6
I2	1.0	1.0	0.9	1.0	0.9	0.9	1.0	0.9	0.9	1.0	1.0	1.0	1.1	0.9	0.9	1.0
I3	1.1	1.1	1.4	1.7	1.1	1.2	1.9	1.1	1.2	1.7	1.0	0.9	1.0	1.1	1.2	1.6
V1	0.7	0.8	1.4	0.9	0.8	1.2	0.7	0.8	1.1	0.9	1.0	1.1	0.9	0.8	1.2	0.8
Y1	0.6	1.0	2.2	0.8	1.0	2.9	0.8	1.0	4.0	0.9	1.2	5.2	0.9	0.9	3.6	0.9
Y2	1.6	1.3	0.4	1.2	1.2	0.3	1.3	0.9	0.3	1.2	0.9	0.2	1.2	1.2	0.3	1.2
Y3	1.6	1.3	0.4	1.2	1.2	0.3	1.3	0.9	0.3	1.2	0.8	0.2	1.1	1.2	0.3	1.2

Table 11. Evaluation of total execution times, total number of aborts, and total application cost for the new configurations obtained with HyAcq: $T_r = \frac{T_{\mathcal{O}}}{T_{\mathcal{I}}}$, $A_r = \frac{A_{\mathcal{O}}}{A_{\mathcal{I}}}$, and $C_r = \frac{C_{\mathcal{O}}}{C_{\mathcal{I}}}$, where $T_{\mathcal{O}}$, $A_{\mathcal{O}}$, and $C_{\mathcal{O}}$ are the execution time, the number of aborts, and the application cost respectively for the corresponding configuration. $T_{\mathcal{I}}$, $A_{\mathcal{I}}$, and $C_{\mathcal{I}}$ are the execution time, the number of aborts, and the application cost respectively for the corresponding initial state. The initial states are available from Table 7. Note that the absolute reference numbers for the eager and the lazy configuration are reported in Table 8. Notice too that the cost is computed by instrumenting a given configuration, extracting the corresponding `RAG`, and computing its cost as described in Section 6.3.2.

7.5 Discussion of Results

Overall, `HyAcq` produces significant performance improvements. From Table 11, individual performance improvements of upto 90% were seen. Using the execution times for all recommended policy configurations on 1, 2, 4, 8, and 16 processor counts, the average improvement is 9% overall.

An interesting question is whether `HyAcq` is able to beat both eager and lazy schemes. Note that this aspect has not been factored into our analysis technique since our optimization technique builds on the profile results obtained from only one run (currently either eager or lazy throughout). However, by examining the execution times obtained from our experiments, there are a few such cases. G1, G2, and G3 improve over lazy (and lazy is better than eager for genome) on 4, 8, and 16 processor runs by around 8% on average. I1 and I2 beat eager (and eager is better than lazy for intruder) on 4, 8, and 16 core runs by a small margin (3% on average). Needless to say, the benefit from `HyAcq` is an artifact of the transactional application. In the case of STAMP benchmarks, either eager or lazy appears to lead to the best results (or very close). Finding applications and patterns where a hybrid configuration consistently produces better results remains a part of future work.

The *Average*-column in Table 11 indicates that given an application and any recommended configuration, the chance of achieving higher performance over the initial configuration is quite high.[19] Though `HyAcq` appears to produce performance improvements on average, some individual slowdowns are seen for some configurations. Additionally, given that a `RAG` is built at runtime and may not capture all possible abort relationships (especially with inputs having completely different concurrency patterns), we think that profile-driven optimization for `STM`, as described by us, is probably not yet mature enough to universally guarantee performance improvements. Indeed, its results should be considered a set of templates that the programmer can try out and derive insights from.

7.6 Effect of Training Input on HyAcq

With any profile-driven scheme, a natural question arises about the relationship between *training* and *reference* inputs. For our purpose, the training input is the one using which the `RAG` is built. The reference input is the one on which final runtime performance is measured. For results presented in Section 7.4, the training and the reference set are the same. To understand how a difference between the training and the reference inputs might affect our technique, we considered two of the STAMP benchmarks from Section 7.4 (`vacation` and `yada`), ones that have alternative inputs specified as part of the distribution. We performed two classes of experiments:

1. For a given benchmark, we use the first input set in the training run and the second in the reference run and vice-versa.
2. For a given benchmark, we aggregate the effects of all the inputs in the RAG during the training run and use each input one by one during the reference run.

For vacation, a low contention input and a high contention input are available. For the first class of experiment, the resulting policy configurations are exactly the same. In Section 7.4, we report results with the high contention input. It turns out that the performance trends for the low contention input are pretty much identical to the high contention input.[20] We looked at the `RAG` and `HyAcq` statistics and there were expectedly less aborts seen in the low contention case but it appears that all the important ones were found in both. For the second class of experiments, we considered the 3 different `ttimeu` inputs in `yada` and again the policy recommendations and the performance trends were similar. It is worth mentioning in this context that our technique is able to handle multiple training inputs. A simple aggregation of the `RAG` is done in such a case where common nodes, edges, and annotations are merged and new ones added to the new `RAG`. We applied this aggregation for both `vacation` and `yada` and the results did not change. While it is hard to generalize based on a few benchmarks, it is quite likely that our experience with training inputs holds for a large number of programs. This is because the contention points in a program tend to be the same — the contention may be low or high depending on the input but the relative contention very often stays the same.

8. Related Work

The multi-faceted nature of transactional memory research has probably been best summarized in the `Transactional Memory`

[19] 7 out of 10 configurations produce runtime improvements averaged over all processor runs. Notice that this number is skewed by the results of yada. Notice further that the recommended configurations for yada show better scalability and the average may be different for yada if tested on more number of processors.

[20] In fact, `V1` (generated with lazy as the initial policy) performs better than the eager configuration in the low contention case by around 10-15%.

book [10]. Another rich source of information is the Transactional Memory Bibliography [17]. While initial work focussed on HTM and STM implementations [10], API and ABI proposals have recently been published [9]. Though a lot of progress has been made in STM research, there is concern that the overheads may overshadow its promise [4]. But it has also been pointed out that certain optimizations such as redundant barrier removal will play a big role in getting the overheads to a reasonable level. However, at this point, there aren't any well-defined tools to point out these optimization opportunities. As STM implementations mature, we need good debugging and performance analysis tools that complete the entire ecosystem. The goal is two-fold: understanding the characteristics of the application itself to make the best use of the underlying TM model and having enough information about the STM so as to choose the best design points.

8.1 STM Profiling

It has been noted that overall abort rate is not helpful in characterizing the complex behavior of realistic TM workloads and that *averaging* transactional statistics could be quite *deceiving* [18]. As a solution, abort rates per atomic block were implemented and used to pinpoint the specific atomic block that adversely affected scalability of the application. Similar support has been implemented in the Intel STM prototype [9]. Statistics of STM behavior on an atomic block basis were used to examine the conflict relationships between atomic blocks [14]. Another related work is tm_db [8], a library that provides programmers with generic transactional debugging features.

The profiling technique that is most closely related to ours is conflict point discovery [20]. The conflict point discovery is a debugger feature that provides information about the victims of aborts. Notably missing is information about the aborters. This technique thus provides data similar to what L2 would provide in our framework. Similar to our observations, the authors [20] note that there is a *probe* effect of collecting this kind of information but that they are low and do not introduce any new high level contention. The basic conflict point discovery has been extended [21] to support more extensive contextual information about conflicts and account for all conflicting memory accesses within aborted transactions (instead of just the first). A major difference between the results of extended conflict point discovery and the RAG is that the the former captures the aborter at the atomic section granularity while the latter is able to capture the aborter at the reference granularity. Notice that collecting information at finer granularity involves careful orchestration of cross-thread interactions to minimize overheads but it leads to more useful information. Note also that our framework has the capability to expose some source location information in the form of the outermost atomic section and the transactional reference, both for the victim and the aborter. However, exporting the RAG to the user in the most meaningful way has not been the focus of our work. The extended conflict point discovery technique attempts to collect as many potential conflicts as possible at runtime by not aborting the transaction at the first conflict but continuing till the end of the transaction. While this idea does lead to more conflict identification in a single run, it does have consistency issues after the first abort point and may not be feasible in an unmanaged environment. It is also likely that for our purpose of building the RAG, such information could easily pessimize the RAG to such an extent that automatic analysis might not be effective any more. However, this is something we have not explored and may provide complementary opportunities. Our work builds on the basic relationship between aborters and victims but develops it fully and shows substantial performance improvements with an automated optimization technique. To the best of our knowledge, this is the only work that attempts to develop a framework to automatically assign the most beneficial conflict detection policy at the transaction level.

8.2 Performance Analysis of General Multithreaded Applications

Thread Profiler [1] helps analysis of general lock-based programs by providing performance metrics and associating them with synchronization objects. The work on analysis of lock contention [16] is even more powerful by assigning *blame* for idleness to lock holders in a contextual manner. At a high level, the central idea of identifying lock contention and blaming lock holders for spinning threads is related to our work of associating aborters with victims of aborts, though in a very different STM setting.

8.3 Supporting Multiple Policies at the Transaction Level

The *unified STM algorithm* [13] supports four execution modes (optimistic, pessimistic, obstinate, and serial) and allows the runtime to choose dynamically between the pessimistic and the optimistic mode on a per-transaction basis. However, we are not aware of an automatic performance-driven analysis to drive this choice. While the pessimistic and optimistic mode combination is different from that of eager and lazy acquire hybridization, the RAG and the general nature of our analysis can be complementary to the unified STM framework.

9. Conclusions

We described an instrumentation technique to build the runtime abort graph (RAG) for an STM, capturing both victims and aborters at the memory reference granularity. We showed that the the runtime cost of building the RAG is low and that the information captured closely tracks the characteristics of the uninstrumented program. The RAG has been incorporated in a profile driven optimization framework that tries to choose the most beneficial conflict detection policy on an atomic section basis. Experimental results for policy hybridization showed some significant performance improvements over existing techniques.

However, given that the RAG denotes the abort pattern in a given execution, more accurate modeling is required to represent unseen patterns that may occur in another configuration obtained through HyAcq or otherwise. Additionally, for future work, we would like to explore other applications of the runtime abort graph. For instance, utilizing the RAG for online optimizations is a promising approach but one where additional issues such as phase identification and balancing overheads vs optimization gains have to be addressed.

References

[1] Intel VTune Performance Analyzer with Intel Thread Profiler. http://software.intel.com/en-us/intel-vtune/.

[2] M. Arnold and B. G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[3] C++ Standards Committee, Pete Becker, ed. Programming Languages - C++ (final committee draft). C++ standards committee paper WG21/N3092=J16/10-0082, http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2010/n3092.pdf, March 2010.

[4] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: Why is it only a research toy? *Communications of the ACM*, 51(11):40–46, Nov. 2008.

[5] D. R. Chakrabarti. New Abstractions for Effective Performance Analysis of STM Programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 333–334, 2010.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, pages 194–208, Sept. 2006.

[7] A. Dragojević, R. Guerraoui, and M. Kapałka. Stretching transactional memory. In *PLDI '09: Proc. 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 155–165, jun 2009.

[8] M. Herlihy and Y. Lev. tm_db: A Generic Debugging Library for Transactional Programs. In *Proceedings of the Parallel Architectures and Compilation Techniques*, Sept. 2009.

[9] *Intel C++ STM Compiler, Prototype Edition 3.0*. Intel Corp., Dec 2008. At `http://whatif.intel.com`.

[10] J. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2007. ISBN 1–59829–124–6.

[11] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. S. III, and M. L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory . In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers and Hardware Support for Transactional Computing*, pages 1–11, June 2006.

[12] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Oct. 2008.

[13] Y. Ni, A. Welc, Ali-Reza-Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2008.

[14] N. Sonmez, A. Cristal, O. S. Unsal, T. Harris, and M. Valero. Profiling Transactional Memory applications on an atomic block basis: A Haskell case study . In *Second Workshop on Programmability Issues for Multi-Core Computers*, Jan. 2009.

[15] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for softwaretransactional memory. In *Proceedings of the Twentieth International Symposium on Distributed Computing*, Sep 2006.

[16] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing Lock Contention in Multithreaded Applications. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 269–279, 2010.

[17] *Transactional Memory Bibliography*. University of Wisconsin at Madison, 2009. At `http://www.cs.wisc.edu/trans-memory/biblio`.

[18] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 265–274, June 2008.

[19] Y. Zhang, V. C. Sreedhar, W. Zhu, V. Sarkar, and G. R. Gao. Optimized Lock Assignment and Allocation: A Method for Exploiting Concurrency among Critical Sections. CAPSL Technical Memo Revised 65, University of Delaware, Mar. 2007.

[20] F. Zyulkyarov, T. Harris, O. S. Unsal, A. Cristal, and M. Valero. Debugging Programs that use Atomic Blocks and Transactional Memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 57–66, 2010.

[21] F. Zyulkyarov, S. Stipic, T. Harris, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. Discovering and Understanding Performance Bottlenecks in Transactional Applications. In *Proceedings of the Parallel Architectures and Compilation Techniques*, 2010.