Reconciling Transactional Conflicts with Compiler's Help

Sandya S. Mannarswamy
Hewlett Packard India and IISc
Bangalore, India
sandyasm@gmail.com

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

ABSTRACT

Software transactional memory(STM) is a promising programming paradigm for shared memory multithreaded programs. While STM offers the promise of being less error-prone and more programmer friendly compared to traditional lock-based synchronization, it also needs to be competitive in performance in order for it to be adopted in mainstream software. A major source of performance overheads in STM is transactional aborts. Conflict resolution and aborting a transaction typically happens at the transaction level which has the advantage that it is automatic and application agnostic. However it has a substantial disadvantage in that STM declares the entire transaction as conflicting and hence aborts it and re-executes it fully, instead of partially re-executing only those part(s) of the transaction, which have been affected due to the conflict. This "Re-execute Everything" approach has a significant adverse impact on STM performance.

In order to mitigate the abort overheads, we propose a compiler aided *Selective Reconciliation STM (SR-STM)* scheme, wherein certain transactional conflicts can be reconciled by performing partial re-execution of the transaction. Ours is a selective hybrid approach which uses compiler analysis to identify those data accesses which are legal and profitable candidates for reconciliation and applies partial re-execution only to these candidates selectively while other conflicting data accesses are handled by the default STM approach of abort and full re-execution. We describe the compiler analysis and code transformations required for supporting selective reconciliation. We find that SR-STM is effective in reducing the transactional abort overheads by improving the performance for a set of five STAMP benchmarks by 12.58% on an average and up to 22.34%.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: [Concurrent Programming]

General Terms

Performance

CGO '12, March 31-April 4, San Jose, US

Keywords

compiler, software transactional memory, conflicts

1. INTRODUCTION

Software transactional memory (STM) [17] is a promising programming paradigm for shared memory multithreaded programs as an alternative to traditional lock-based synchronization. While STM offers the promise of being less error-prone and more programmer friendly, it also needs to be competitive in performance in order for it to be adopted in mainstream software. Hence improving the performance of applications running on STM becomes increasingly crucial in enabling the mainstream adoption of STM.

TM systems typically use optimistic concurrency control mechanisms [17]. This allows the TM to execute the transactions concurrently without obtaining exclusive ownership of data at the beginning of an atomic section. If any data conflicts actually are encountered, they are dynamically detected and resolved by the underlying TM runtime. This is achieved either by appropriately reordering the transactions if they are serializable or by allowing only one out of the group of conflicting transactions to proceed while aborting all the other conflicting transactions. When a transaction aborts, the STM runtime will roll back the transaction's local state and restart execution from the beginning of the atomic block. Two concurrent transactions conflict when they access the same location and at least one of the accesses is a write (update). The set of shared memory locations read by a transaction is known as its *read set* and the set of shared memory locations written by the transaction is its *write set*. In order to commit, a transaction must ensure that all variables in its read set remain unaltered (i.e., have not been written by other committed transactions since the current transaction was started, a process known as read-set validation) and it is able to acquire the locks for each variable in its write set. The transaction aborts if either of these two conditions is not satisfied.

If there are conflicting memory accesses between two concurrent transactions, then the STM runtime aborts one of the transactions completely and re-executes it in full. Since transactions execute optimistically and typically roll back if a conflict is detected, there may be lot of wasted work done by transactions which eventually abort. Aborted transactions reduce performance, scalability and waste computing resources. Therefore reducing transactional aborts is an important consideration in improving the performance of the software transactional memory systems. Prior work by other researchers [7, 22] and our performance analysis of STM applications show that conflicts leading to transactional aborts are a major source of performance bottleneck for STM applications.

Conflict detection happens at multiple points during transactional execution, i.e., during the speculative transactional execution and during the commit time read set validation. A conflict that is detected during commit time read set validation phase results in the transaction throwing away the entire transactional work done (at the commit read set validation step, the entire speculative transactional execution would have completed fully). Aborting and re-executing the entire transaction on a conflicting memory access has the advantage that it can be done completely automatically by the STM without any knowledge of the application's data structures and program semantics. However it has a substantial disadvantage in that it declares the entire transaction as conflicting, instead of identifying which part(s) of the computation, have been affected due to the conflicting data access and re-execute only that part(s).

Typically a transaction consists of a number of computational steps. A single data access conflict is likely to impact only a specific part of the computation being performed in a transaction and not all the computational steps of the transaction. Since the STM runtime has no way of analyzing what part of the computations are dependent on the conflicting data access, it has no way of re-executing only the required part of the transactional computation. Instead it chooses the simplistic approach of re-executing the entire transaction. While this "Re-execute Everything" approach works well generally for short transactions where the amount of wasted work is relatively small, it can prove to be costly for large transactions. Further full re-execution can end up in a pathological situation where short write updates on a shared data structure such as a linked list can prevent concurrent long running data structure query transactions getting aborted frequently [3].

The point to note is that if the impact of the conflicting data access is local and is confined to one or a few of the intermediate computation steps of the transaction, it would be profitable to simply re-execute the intermediate computation on the changed data and propagate the change instead of performing a full blown re-execution of the entire transaction. We term such transactions as reconcilable transactions, since they can be reconciled without having to abort. However the challenge in reconciling conflicting transaction is identifying the part of the computation involving the stale value, safely undoing them and ensuring that all inputs are available to perform selective re-execution of dependent computations correctly.

In this paper, we show that it is possible to use compiler analysis to identify reconcilable transactions and the compiler can generate code to reconcile such transactions instead of aborting and re-executing the transaction in its entirety. We propose a selective hybrid approach which we term as *Selective Reconciliation STM (SR-STM)*. SR-STM uses compiler analysis to determine those candidate data accesses which are legal and profitable for reconciliation and performs automatic code transformations to apply partial re-execution to those conflicting accesses selectively while all other conflicting data accesses are handled by the default STM approach. Changes in the STM runtime needed to support automatic selective reconciliation have been implemented.

Note that our selective reconciliation technique can reconcile only certain classes of transactions which are legal and profitable for reconciliation by partial re-execution. It complements but does not completely replace the traditional STM runtime's approach of full abort of a transaction on a conflict. Since our approach is based on automatic compiler analysis and transformation technique, it needs to be conservative unlike manual techniques which involve programmer involvement. In case of large transactions which are amenable to reconciliation, the proposed approach mitigates the abort overheads considerably, as can be seen from our experimental evaluation of this technique. Its key advantage is that it does not require any programmer intervention and is fully automatic. As compilers mature on integrating compiler analyses and transformations to mitigate the STM performance overheads, our proposed SR-STM technique will be a useful weapon in their arsenal to address the transactional abort overheads associated with the application unaware *"Re-execute Everything"* approach employed by STM runtime.

There has been considerable prior work on relaxing the low level consistency checks performed by the STM for conflict detection in order to reduce the aborts incurred by the transactions [14, 13, 11] and have been shown to yield considerable performance gains. However these techniques typically require *programmer intervention* in the form of annotating an early release access or marking a transaction as elastic or providing inverses for operations as in the case of boosted transactions. *Requiring programmer intervention in relaxing the consistency checks defeats the essence of transactional memory paradigm to the extent that it brings the programming burden back to the programmer. Unlike these approaches, ours is a fully automatic compiler aided approach for reducing the transactional abort overheads*.

We have implemented a prototype of our SR-STM scheme in Open64 compiler with TL2 as our underlying STM for our experimental evaluation. Performance results on a set of STAMP benchmark programs indicate that our SR-STM scheme improves application performance in 5 of the STAMP benchmarks from 3.24% to 22.34% over the base STM. The rest of the paper is organized as follows: Section 2 provides the motivation for this work. Section 3 describes the SR-STM scheme. We discuss the experimental results in Section 4. We discuss related work in Section 5 and provide concluding remarks in Section 6.

2. MOTIVATION

This section motivates the need for a selective re-conciliation scheme

2.1 Motivating Examples

The point to note is that, when an aborted transaction is re-executed, most of its re-computation is performed on the unchanged part of the shared data inputs. Further the computation performed on the changed part of the shared data inputs to the transaction typically may not impact all the externally visible results computed by the transactions. There exist transactions in which a conflicting memory access impacts only a single or a very few of the total computation steps and hence re-execution can be confined only to those impacted computation steps.

Consider a simplified and modified code snippet from the STAMP benchmark *Intruder* shown in Figure 1 wherein which multiple concurrent transactions perform membership query operations on a common data structure `D` such as a linked list.

There are no conflicts among the concurrent threads when they are executing the query operation in statement S1. However they can conflict on the update to the shared counter in S2 which maintains the number of query operations performed. On encountering such a conflict, the entire transaction has to be re-executed including the expensive query operation on the data structure D. In fact, the only

```
foo(int item)
{
        bool result;
        atomic {
          S1:  result = IsMember(D, item);
          S2:  no_of_queryoperations_performed++;
        }
        return result;
}
```

Figure 1: An Example of a Reconcilable Transaction

dependent computation on the conflicting data access is the update to the shared counter $no\_of\_queryoperations\_performed$ and it suffices to re-execute just the dependent computation with the modified shared counter value to reconcile the conflict. While the code snippet shown in Figure 1 is a trivial example, it helps to illustrate the point that a conflicting data access does not necessarily require a full blown re-execution of the transaction.

Another example of transactions which is amenable to selective re-execution is the data structure queries and invariant checks. Consider the abstract data type `Set` which is implemented using a sorted linked list. The `Set` abstract data type supports `Insert`, `Delete` and `IsMember`. Two concurrent transactions `T1` and `T2` with `T2` performing an operation `Insert(10)` and `T1` performing `IsMember(200)` can actually commit together without violating program semantics. However these transactions when executed together will conflict. Specifically if `T2` is committed before `T1` performed the commit time read set validation. `T1` detects a low level memory access conflict during its commit-time read set validation (since `T2` has inserted 10 and has committed, the underlying structure of the list has changed and hence this leads to a structural conflict).

Typically all STMs will abort and re-do the transaction `T1` in its entirety. However the intermediate steps performed by `T1` are mutually independent in that each intermediate step simply checks a single node of the linked list to see if it contains 200 or not. Therefore it suffices to re-execute the check for the membership of 200 on the node newly added by the committed transaction `T2` which causes the conflict.

In both the examples above, the impact of the conflicting data access is local and is confined to one or a few of the intermediate computation steps of the transaction. In such cases, it would be profitable to simply re-execute the intermediate computation on the changed data only and propagate the change instead of performing a full blown re-execution of the entire transaction. We term such transactions as reconcilable transactions, since they can be reconciled to conflict without having to abort. Next we discuss the performance potential for reconciliation of transactional conflicts by partial re-execution.

2.2 Performance Potential for Reconciliation

In spite of considerable prior work [22, 8, 17] on techniques to reduce conflicts, recent work [25, 9, 26, 15] point to transactional aborts as a significant contributor to STM performance overheads. In order to understand whether transactional aborts are still significant on STM, we measured the percentage of the aborts experienced by STAMP benchmarks [4] on a state of the art STM, namely TL2 [6]. We used the native (non-simulator) input sets of the STAMP benchmark suite in our experiments. We used a 32 core IA-64 BL-890c system, with 8 Intel IA-64 Processors [21] with 4 cores per socket, with each core operating at 1.5 GHz.

Table 1: Abort Behavior of STAMP benchmarks

Benchmark	Number of Atomic Sections	Data structures accessed in atomic sections	Number of Transactions started	% of aborts for total transactions started	% of reconcilable aborts
Bayes	15	Linked list, struct learner	3279	17.12	13.81
Kmeans	3	2 int arrays, 2 globals	51698806	76.38	0
Genome	5	Linked list	2677857	16.06	11.27
Intruder	3	Queue, linked list	131015915	82.13	34.31
Labyrinth	3	3d array	1726	23.17	2.19
Ssca2	10	Array	22436386	3.36	0.27
Vacation	3	4 RB trees	6750230	72.65	41.72
Yada	6	element	70892	52.11	18.25

Table 1 describes the abort characteristics of the STAMP benchmarks when run on an TL2 with 32 threads. We report the number of atomic sections in each of these benchmarks in Column 2, the major data structures used by the application in Column 3, the total number of started transactions in Column 4, the percentage of aborts to total started transactions in Column 5, and the percentage of potentially reconcilable aborts to total started transactions in Column 6 of Table 1. We determine such reconcilable aborts as those transactional executions which did not have any changes in their write set when they re-executed the same transaction after the abort.

We instrumented TL2 to obtain the number of aborts and number of reconcilable aborts for each atomic section in the application. We find that in 4 out of 8 benchmarks, more than 40% of transactions started were aborted, showing that aborts are still a major performance bottleneck. We find that 2.19% to 41.72% of the aborts in stamp benchmarks are potentially reconcilable in that re-execution after abort did not cause any changes in the externally visible memory locations written by those transactions from the original execution. Therefore considerable performance improvement potential for STM exists if the abort overheads can be reduced.

3. OUR APPROACH

Next we describe our approach for selective reconciliation in detail.

3.1 Overview of SR-STM

The set of computational steps performed by the transaction can be represented as a computation dependence graph G as shown in Figure 2, where the leaf nodes of the graph are the data inputs coming into the transaction and the internal nodes represent the computation steps performed by the transaction. Certain nodes are labeled as result nodes since they produce external visible results of the transaction. Let D = d_1, d_2, d_3,... d_i, ... d_n be the set of shared data items of the atomic section. Let $D_{conflict}$ be the set of shared data accesses which incur commit time read-set validation conflicts in D for a transaction T executing the atomic section.

A data dependence exists from step S_p to step S_q if step S_p computes a value which is used by S_q. Dependent computations of d_i include both the direct and indirect dependents. We denote the set of dependent computations for d_i as DG_i. Note that we treat

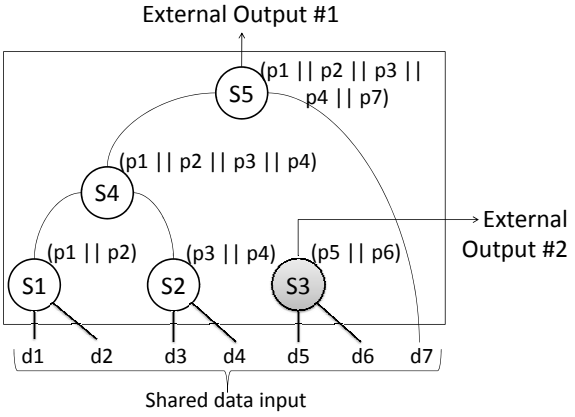

Figure 2: Illustrating the Predicated Computation Graph

control dependences by converting them to data dependences using predication and hence we do not differentiate between control and data dependences in our discussion here. Given a shared data input $d_i \in D_{conflict}$, if the compiler can determine the set of computations dependent on each of these d_i and re-execute only the dependent transactional computations with the correct inputs, such a transaction can be reconciled due to the changes in d_i, instead of having to re-execute even those computational steps whose inputs have not changed. For example, Consider a conflict detected for d_5 during commit time read set validation. The computation step(s) which are dependent on d_5 is only the single computation step $S3$ which needs to be re-executed with the changed input. On the other hand if the entire transaction is aborted and re-executed, all five computation steps of the transaction are re-executed leading to considerable wasted work. This is the key idea behind our approach to reconciling transactions by partial re-execution.

For each of the conflicting data accesses $d_i \in D_{conflict}$, compiler determines the sub-graph which constitutes its dependent subgraph. The union of the dependent sub-graphs of DG_i of the conflicting data accesses $d_i \in D_{conflict}$ is denoted by $DG_{Dconflict}$, which is the sub-graph induced by the vertex set $D_{conflict}$ on G. Under the assumption that the effects of the original transactional execution with the stale values of each conflicting data access d_i in $D_{conflict}$ has been rolled back completely and that all the inputs needed to perform the re-computation represented by $DG_{D_{conflict}}$ are available, reconciliation consists of re-execution of the computational steps in $DG_{Dconflict}$ emitted in a topologically sorted order.

Reconciling the conflicts by partial re-execution is achieved in our approach as a predicated execution of the transaction's full computation graph with the predicates being set by the changes to the shared data inputs. Let p_i be the predicate denoting that a conflict has occurred in shared data input d_i. Then we can annotate the nodes of the computation graph for re-execution as shown in Figure 2. In the simplest terms, partial re-execution can be visualized as the original computation graph of the transaction with the computation nodes predicated by conditions on the predicates p_i. The set of computational nodes selected for re-execution represent the reconcilable region of the transaction. A reconcilable region can be as small as a single computation node or as large as the entire computation graph. If the reconcilable region is large, then reconciliation may not yield any benefits. Therefore certain thresholds need to be applied on deciding when it would be beneficial to try and reconcile a transaction. We discuss this in Section 3.3.1.

Next let us consider the case where *node S*3 represents an iterative computation such as searching for an item in the linked list. Let d_5 represents the head pointer of the list and d_6 be the item being searched for. The read-set of the transaction would consist of all the linked list elements visited. Now if any of the read-set elements were modified by a concurrent transaction which is detected as a conflict during commit time read-set validation, it would cause the transaction performing the search to abort and re-execute completely. In this case, compiler analysis of the iterative computation step can identify that each of the intermediate computation steps is independent, checking for the membership of the item d_6 at that node and hence the computation is reconcilable by partial re-execution and propagating the change to the externally visible result.

In order to enable reconcilable execution, the input and output values of the iterative computation step *S*3 need to be checkpointed for each iteration, which adds to the book keeping overheads of selective reconciliation. On a read-set validation failure of the iterative computation, using the checkpointed values, SR-STM re-executes the computation from the iteration for which the input has changed and continue the reconcilable execution with the modified value until the output matches the checkpointed output or a threshold number of iterations are executed. In the latter case, the reconciliation has failed and the transaction is aborted.In the earlier case, the reconciliation has succeeded. However a full read-set validation is required to ensure that no other read-set element conflict has occurred for other memory locations/variables. Hence a subsequent full read-set validation takes place. If the read-set validation succeeds, the transaction is successfully committed. Else the transaction is aborted.

Our discussion above implicitly assumes that it is possible to completely roll back the effect of the dependent computations executed with the stale data input and that the all the inputs needed for the re-executing the dependent computations of the changed input are available for reconciliation to occur. However these two assumptions are non-trivial. For instance, consider the example where *node S*3 requires an additional input I_{local} which was later overwritten in the transaction. The original value of I_{local} needs to be preserved somewhere before it is over-written in the transaction so that *node S*3 can be re-executed. Such cases require expensive book keeping operations which can nullify the savings in wasted work we would obtain from reconciliation instead of aborting. Therefore reconciliation using partial re-execution has to be applied carefully and selectively for transactions. Our scheme is a selective approach which applies a number of legality and profitability checks to decide whether a candidate region is eligible for reconciliation.

3.2 Transactional Execution Under SR-STM

Our scheme is similar to the baseline TL2 [6] transactional execution except for certain changes which we describe here for those transactions which are marked as reconcilable by our compiler analysis. SR-STM maintains additional book keeping information during the normal transactional execution to support incremental re-execution during the reconciliation phase. This includes checkpointing certain values that are overwritten during the transactional execution, but may be required for the reconciliation handler to execute. Also in order to identify whether the execution of a part of the reconciliation handler needs to be continued or not, checkpoint-

ing of the input and outputs of the intermediate computation steps are required.

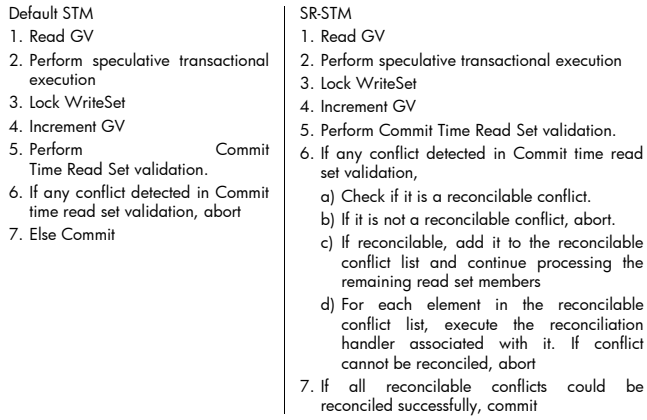

Figure 3: Transactional Execution on SR-STM

During the commit time read-set validation if a conflict is detected for a read-set member of the transaction, SR-STM checks to see if the conflicting data accesses have been marked as reconcilable. If any of the conflicts are non-reconcilable, then the transaction aborts. Otherwise SR-STM looks up the reconciliation handlers associated with the conflicting data accesses and executes them. At the end of execution of all the reconciliation handlers, the entire read-set is validated. If it succeeds then the writes of the transaction are committed and the transaction is completed successfully.

Note that the read-set validation at the end of the reconciliation handler execution is required for the entire read-set and not just for the data items that involved the reconciliation handlers. Also the write-locks are held through during the reconciliation and are released only after the writes are committed or when the transaction is aborted. The changes in SR-STM transactional execution are shown in Figure 3.

3.3 Steps of SR-STM Scheme

Our scheme consists of the following 3 major steps:

Identification of reconcilable regions: Compiler analyzes the atomic sections to identify those computational code regions which can be reconciled by partial re-execution.

Generation of reconciliation header: When there are conflicts incurred by the shared data accesses in reconcilable code regions, they need to be reconciled. Compiler generates the reconciliation handler for a reconcilable code region identified in step 1. Reconciliation handler is the outlined version of the code which constitutes the reconciliation region and consists of the set of instructions which depend on the values produced by the reconcilable data accesses present in that region.

Modification of STM interfaces: Our SR-STM scheme requires modified STM interfaces to register the reconciliation handlers associated with a given transaction, with the STM so that they can be invoked by the STM runtime during commit time read-set validation phase when reconciling conflicts. There are also changes required in the STM library routines to perform the book-keeping needed to support reconciliation.

Next we discuss each of these steps in detail.

3.3.1 Candidate Selection

Rolling back the effects of the computations performed using the stale value requires comprehensive check pointing in order to roll back to a state from which the dependent computations of the conflicting data access can be re-executed with the correct value. While this can be achieved for any arbitrary code region with infrastructure support for checkpoints and program continuations [16], the cost involved is non-trivial. We target our selective reconciliation scheme on such code regions which do not require expensive roll back support. Therefore we impose the following conditions for a candidate code region to be considered as a reconcilable region:

(i) The inputs to the reconcilable region consist only of (a) members of the read-set of the transaction and (b) transaction local variables whose values do not change in subsequent computational steps of the transaction.

(ii) Operands of the reconciliation region computations are not overwritten by any of the subsequent computations of the transaction following the reconciliation region.

(iii) There are no dynamic memory allocations/IO/system calls or other irrevocable actions performed inside the reconcilable region.

(iv) None of the instructions executed inside the reconcilable region can be a potential exception causing instruction

(v) There are no entries to the reconcilable region except through its head (No jumps into the middle of the region).

(vi) The reconciliation region does not include any external function calls

Conditions (i) and (ii) are imposed so that the operands needed for re-execution are available without having to support expensive book keeping needed to ensure availability of operands. Conditions (iii) to (vi) are imposed to avoid costly full fledged check pointing and continuation requirements not provided by our current implementation. Support for full fledged check pointing and continuations can eliminate these requirements and allow more candidate code regions to be reconcilable compared to our current scheme. Our compiler marks those code regions which satisfy the legality checks as candidate reconcilable code regions.

A trade off exists between the overheads incurred in supporting transactional reconciliation versus the benefits obtained by avoiding abort induced full re-execution. In a default STM without any selective reconciliation, the cost of full re-execution is incurred for all aborts. In our SR-STM scheme, we incur the cost of full re-execution for non reconcilable aborts only. Whereas for reconcilable aborts, we incur only the cost of partial re-execution. However SR-STM incurs an additional book keeping cost for supporting reconciliation.

Given an atomic section *AS* which contains a reconcilable region as identified by the compiler, let N be the total number of executions of *AS*. Those aborts which are caused by conflicts detected during

commit time read-set validation occurring in the reconcilable region are the ones which can be reconciled by SR-STM scheme and we denote this as $N_{recon-aborts}$. Other aborts of the atomic section *AS* occur due to non-reconcilable conflicts and we denote this as $N_{nonrecon-aborts}$. Let C_{bk} be the book keeping cost associated with each transactional execution of the atomic section *AS* to support reconciliation. Let $C_{full-reexecution}$ be the cost of full re-execution of the atomic section *AS*. Let $C_{partial-reexecution}$ be the cost of partial re-execution of the reconcilable region in *AS*. In order for selective reconciliation to be profitable, we should have

$$C_{full-reexecution} * (N_{recon-aborts}) > (N * C_{bk}) + (N_{recon-aborts} * C_{partial-reexecution}) \quad (1)$$

We select a candidate for reconciliation only if it is profitable as per Equation 1. We use the static profile heuristics available in our compiler [23] to estimate the execution frequencies of the different code regions and estimate the various cost quantities associated with full re-execution, partial re-execution and book keeping. The estimates for $N_{recon-aborts}$ and $N_{nonrecon-aborts}$ are obtained from the conflict profile of the transaction. It is possible to use dynamic profile for execution frequencies to compute the costs more accurately. We obtain conflict profile information from dynamic conflict profiles obtained by runs using STAMP small inputs, while performance results reported in Section 4.2 are for native inputs.

Our compiler marks those code regions which satisfy the legality/profitability checks as reconcilable code regions and the shared data accesses contained in them as reconcilable accesses. This information is written to a non-loadable section of the application executable as an annotation and is used by the STM to determine whether a conflict detected during commit time read-set validation is reconcilable or not.

3.3.2 Generation of Reconciliation Handler

When a potentially reconcilable conflict is detected by the STM during the commit-time read-set validation, it implies that one or more of the shared data inputs to an intermediate computation step of the transaction has changed. In order to reconcile these conflicting data accesses and their dependent computations, our compiler synthesizes a reconciliation handler which contains the code region involving the dependent computations of the conflicting data accesses. Thus executing the handler involves re-executing the reconcilable code region, with the new values of the conflicting data accesses. The results of the computations involving the shared data accesses are propagated forward over the dependent computations of the reconcilable region by the handler. Change propagation stops when an intermediate computation step whose inputs have not changed due to re-execution is reached.

Consider our example computation graph shown in Figure 2. Assume that the computation sub-graph induced by the shared data accesses d_5, d_6 and their dependent computations node *S*3 (*S*3 is a non-iterative computation in this example) was marked as a reconcilable region by the compiler. The outline of the reconciliation handler generated by the compiler for this reconcilable region is as shown in Figure 4.

The generated reconciliation handlers are recorded in a reconciliation handler table, similar to the exception handler tables. The

```
Recon_Handler ( livein d5 , livein d6 )
{
   predicate p1 = d5 incurs a conflict ;
   predicate p2 = d5 incurs a conflict ;
   predicate p3 = p1 || p2 ;
   (p1) read new value for d5 ;
   (p2) read new value for d6 ;
   (p3) compute node(3) ;
}
```

Figure 4: Reconciliation Handler

reconciliation handlers are mapped to the set of shared data accesses using a look up table known as *ReconciliationHandlerMap*, indexed by the tuple ⟨program counter, data symbol of the conflicting data access⟩. The handler table is written out to the executable as an annotation by the compiler.

3.3.3 Modification of STM interfaces for SR-STM

Next we discuss the changes needed in STM interfaces to support selective reconciliation.

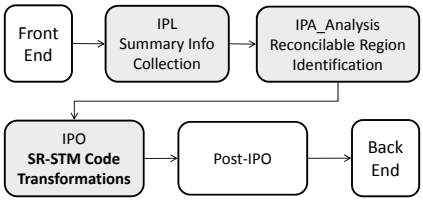

Figure 5: Open64 Compiler Phases

We implemented the SR-STM scheme in Open64 compiler [5]. The analysis and code transformation required to support reconciliation is performed during the inter-procedural analysis and optimization phase of Open64 compiler. Figure 5 shows the Open64 compiler phases which were modified to support SR-STM. The identification of reconcilable regions is performed during the Inter-Procedural Analysis (IPA) phase of our compiler using the summary information collected during the IPL summary collection phase. The code transformations required to support reconciliation is performed by the compiler at the end of the Inter-Procedural Optimization (IPO) phase.

Our compiler during its analysis phase, annotates the atomic sections which contain reconciliation region candidates as potentially reconcilable. For each such atomic section, compiler registers the associated reconciliation handlers information by passing the information as argument to the new STM interface *ReconcilableTxStart* which is used to start a reconcilable transaction. *ReconcilableTxStart* is similar in operation to the default *TxStart* routine of the STM, except that extra book-keeping is performed to record the inputs and outputs of the computational steps of the reconcilable regions contained in it.

Two new STM interfaces *TxReconcilableLoad* and *TxReconcilableStore* are also added to the SR-STM. Our compiler transforms the transactional load/store calls for the reconcilable reads/writes to call *TxReconcilableLoad*/*TxReconcilableStore*. These new interfaces perform additional book-keeping to ensure that the data inputs required for re-execution to support reconciliation are check-

pointed and preserved. A new STM interface *ReconcilableTxCommit* is also added to the SR-STM which performs reconciliation during commit time read set validation on a reconcilable data conflict instead of aborting and re-executing the transaction completely.

3.4 Overheads of SR-STM

Supporting reconciliation requires collecting and maintaining extra book-keeping information during the normal transactional execution for those transactional code regions which are marked by the compiler as reconcilable. This causes runtime overheads as well as additional memory overheads for maintaining the additional book-keeping information.We report the runtime overheads for supporting reconciliation in Section 4.3. The conditions we impose for selecting candidates for reconciliation is intended to filter out those candidates for which considerable checkpointing would be required for reconcilable re-execution. Specifically, our legality checks ensure the book-keeping overheads for reconcilable region is minimum. Also our profitability check filters out those reconcilable regions whose book-keeping overheads would overshadow the performance benefits obtained through reconciliation. Last, even though our method collects additional book-keeping information for memorizing results, they are used to ensure that we can terminate execution early during re-execution.

3.5 Preserving the Transactional Semantics

Next we discuss how reconciliation through selective re-execution preserves the transactional semantics. Let us consider T_s as the current transaction which encounters a commit time read-set validation and let d_i be the shared data access for which a conflict has occurred in T_s. Let T_r be another transaction which had caused a modification to d_i after T_s was started and T_r has since then committed before the read-set validation of T_s was completed. For simplicity, we initially assume that there is only one conflicting data access detected during commit time read set validation of T_s. As discussed earlier, we denote the set of dependent computations for d_i as DG_i.

In the above situation in the default STM, T_s aborts and re-execution of T_s is initiated. If T_s commits successfully on re-execution, it would mean that the commit order of the transactions is T_r, followed by T_s. We argue informally why selective re-execution of DG_i would yield the same results as would have been done in the case of an abort and a full re-execution of the transaction T_s. That is, it would yield the same results as in a serial execution when T_s follows T_r. Since we already know that STM semantics are preserved on full re-execution in the default STM, this would suffice to show that reconciliation through partial re-execution in SR-STM also preserves the STM semantics.

As there is no conflicting data access other than d_i between T_r and T_s, when T_s follows T_r in a serial execution, the computation steps of T_s would execute using the unchanged values of the shared data items for all $D-d_i$ and the new value for d_i which resulted from the execution of T_r. T_s can be considered as the concatenation of the two sequences of computations, namely those which are not dependent on d_i namely $(G-DG_i)$ and those which are dependent on d_i namely DG_i.

Since the reconciliation handler is constructed using the topological sorted order of the vertices of the dependent graph of d_i, the original dependencies of the computation graph are preserved during selective re-execution when executed with the new value of the conflicting data access d_i. Further, since there were no other data conflicts other than d_i in the original transactional execution, $(G-DG_i)$ also obeyed the serial execution constraints of preserving all data dependences. Moreover, all the required inputs for the computation of DG_i are also made available, as ensured by the reconcilable region selection conditions as discussed in Section 3.3.1.

While the normal transactional execution preserves the data dependencies for $D-d_i$ as in the default STM, the selective reconciliation preserves the data dependences by re-executing DG_i with the modified value for d_i produced by T_r. Hence all data dependencies which would have been preserved in a full re-execution scenario of default STM are honored during the selective re-execution. Hence concatenating the normal transactional execution of the non-dependent computations with the selective re-execution of the dependent computations of the conflicting data is equivalent to a serial execution in which T_s follows T_r. While our above discussion assumed a single conflicting data access, the same argument can be extended to multiple conflicting data accesses by considering the dependent graph consisting of the union of the dependent computations for the multiple conflicting data accesses.

4. EXPERIMENTAL EVALUATION

In this section, we describe our experimental methodology and discuss the performance results obtained with SR-STM.

4.1 Experimental Methodology

To evaluate the effectiveness of our SR-STM scheme, we used the STAMP benchmarks [4] with the native input sets. We implemented our SR-STM prototype in the Open64 C/C++ compiler [5]. For the performance runs, the benchmarks, compiled with the SR-STM scheme enabled at optimization level 3 with inter-procedural analysis enabled, were run on the modified SR-STM TL2 implementation. For our baseline, we compiled the benchmarks at the same optimization level (level 3) with inter-procedural analysis enabled, but with SR-STM phase turned off and ran the executable on the unmodified TL2 implementation [6]. We used a 32 core IA-64 BL-890c system, with 8 IA-64 Processors [21] with 4 cores per socket, each core at 1.5 GHz with 6 MB non-shared L3 cache and a total of 127.71 GB of RAM.

4.2 Performance Comparison

In this section, we report the performance results of SR-STM on STAMP benchmarks [4]. The number of candidate reconcilable regions selected for selective reconciliation scheme for each benchmark is reported in Column 2 of Table 2. SR-STM requires inter-procedural analysis to identify reconciliation candidates and also needs to perform code transformations to support reconciliation. We find that the compilation overheads are application dependent and range from 2.98% to 7.92% for our set of benchmarks.

We find that SR-STM had performance impact only on 5 of the 8 STAMP benchmarks. SR-STM shows negligible performance impact ($< 1\%$) on the remaining three benchmarks and we discuss the reasons for this later. We report the percentage improvement in execution time and the percentage reduction in aborts for 4, 8, 16 and 32 threads in Table 3, Table 4, Table 5 and Table 6 respectively for the five STAMP benchmarks on which SR-STM has a performance impact. We find that SR-STM reduces the execution time in the 5 benchmarks by 3.24% to 22.34%. The average improvement for 32 threads is 12.58%. These reduction in the execution times are accompanied by a corresponding reduction in the number of aborts. Further the performance improvements increase with

Table 2: SR-STM Candidates

Benchmark	SR-STM candidates found
Bayes	3
Genome	4
Kmeans	0
Intruder	7
Labyrinth	2
Ssca2	0
Vacation	3
Yada	6

increasing number of threads. This is because with increased number of threads, contention increases which leads to an increase in conflicts and hence the number of aborts in default STM. However many of these conflicts can be reconciled in SR-STM without having to abort the transaction itself.

We note that the performance improvements we obtain with 32 threads is smaller than the performance potential of reconcilable aborts as observed in Table 1. Since our current approach supports reconciliation only for those code regions which do not require extensive check-pointing and book-keeping, our legality and profitability checks filter some of the candidate reconcilable aborts, which results in a lower performance for our scheme compared to performance potential of reconciliation approach. Increasing the scope of our SR-STM scheme to target more reconcilable aborts is part of our future work.

Table 3: Performance Improvements with 4 Threads

Benchmark	Execution Time (in seconds)		% improvement in Execution Time	% reduction in Aborts
	TL2	TL2 SR-STM		
Bayes	6.32	5.73	6.23	4.74
Genome	23.21	21.82	6.01	5.42
Intruder	122.36	116.33	4.93	6.82
Vacation	108.10	96.06	11.14	11.32
Yada	11.79	11.41	3.24	2.01

Most of the conflicts encountered by the *Genome* benchmark occur in the atomic section which constructs the set of unique segments. The hash table buckets are implemented as linked lists and the linked lists suffer from considerable aborts due to read-write conflicts. Also the auxiliary data computation updating the occupancy of the hash table leads to considerable aborts. We find that SR-STM is able to reconcile the conflicts which cause aborts in linked list operations and in the hash table occupancy updates. However the number of conflicts encountered in the *Genome* benchmark overall is relatively low. Hence the impact of SR-STM on performance by reducing the execution time is moderate (10.29%), while the reduction in aborts due to SR-STM is 9.34%. The majority of the conflicts in *Intruder* occur in the two data structures namely, `fragmentedMapPtr` and `decodedQueuePtr`. The `fragmentedMapPtr` object is a map data structure which is used to reassemble the fragmented packets. The map data structure is implemented using the resizable chained hash table. We find that SR-STM can reduce the number of aborts by reconciling the conflicts encountered by the chained hash table. The other data structure in the benchmark *Intruder* which incurs high conflicts is

Table 4: Performance Improvements with 8 Threads

Benchmark	Execution Time (in seconds)		% improvement in Execution Time	% reduction in Aborts
	TL2	TL2 SR-STM		
Bayes	3.93	3.62	7.89	5.82
Genome	17.08	15.88	7.03	6.42
Intruder	109.34	101.76	6.93	9.82
Vacation	92.39	79.33	14.14	14.43
Yada	9.32	8.85	5.04	4.01

`decodedQueuePtr`. It is a queue data structure which exhibits high write-write conflicts due to the lack of disjoint access parallelism in the queue operations which require all transactions updating the queue to conflict on the `head` and `elements` fields of the queue data structure. Selective reconciliation scheme was not applicable for the conflicts incurred by the `decodedQueuePtr` object.

Table 5: Performance Improvements with 16 Threads

Benchmark	Execution Time (in seconds)		% improvement in Execution Time	% reduction in Aborts
	TL2	TL2 SR-STM		
Bayes	2.03	1.85	8.87	8.11
Genome	10.86	9.88	9.02	8.12
Intruder	96.81	86.71	10.43	13.82
Vacation	80.16	66.26	17.34	17.46
Yada	6.41	5.96	7.02	4.94

In *Vacation* benchmark, we find that SR-STM helps to reconcile the conflicts introduced due to table insertions and allows readers and writers to proceed concurrently reducing the aborts for the readers. Since the transactions are reasonably large in *Vacation*, avoiding full re-execution helps to reduce the large amount of wasted work that would have been incurred by aborted transactions in the default TL2 STM. In *Bayes* benchmark, SR-STM is able to reconcile conflicts occurring on operations on the list data structure in this benchmark.

In the benchmark *Yada*, the code regions selected by SR-STM for reconciliation, are the traversal operations on the linked list which is used to hold the neighbors of an element triangle and RB trees used to implement the set of mesh boundary segments. Transactions involving these operations are large in this benchmark and hence conflicts cause considerable abort overheads in the baseline TL2 STM. By reconciling the read-write conflicts on the linked list and RB tree data structure, SR-STM improves performance by 9.47%.

Table 6: Performance Improvements with 32 Threads

Benchmark	Execution Time (in seconds)		% improvement in Execution Time	% reduction in Aborts
	TL2	TL2 SR-STM		
Bayes	1.26	1.12	11.11	9.03
Genome	6.51	5.84	10.29	9.34
Intruder	87.27	75.55	13.43	16.82
Vacation	66.04	51.29	22.34	22.12
Yada	3.92	3.55	9.47	7.12

SR-STM did not have any performance impact on the remaining 3 STAMP benchmarks. SR-STM does not find any reconcilable candidates in *Kmeans*. The amount of conflicts encountered in *Ssca*2

benchmark is very less. Though SR-STM detects eligible candidates after the legality checks, the profitability check of SR-STM filters out these candidates as non-profitable. In the benchmark *Labyrinth*, the candidates are part of cold atomic sections and hence SR-STM has no performance impact on it.

4.3 Overheads of SR-STM approach

Table 7: Runtime Overheads

Benchmark	Runtime Overheads
Bayes	5.76%
Genome	3.24%
Intruder	7.31%
Vacation	8.19%
Yada	4.91%

SR-STM incurs runtime overheads in supporting selective reconciliation which includes maintaining additional book-keeping information during regular transactional executions for transactions which contain reconcilable regions. We measure the runtime overheads by allowing the collection of additional book-keeping information and the invocation of the new transactional library routines added for supporting reconciliation during regular transactional execution. After invoking the routine *ReconcilableTxCommit* and identifying the reconcilable accesses, SR-STM does not reconcile the transaction, instead it aborts the transaction and the overheads incurred in this process are measured as the runtime overheads of SR-STM. We report the runtime overheads in Table 4.3 for 32 threads. We find that the runtime overheads are moderate and range from 3.24% to 8.19%.

5. RELATED WORK

There has been considerable work on relaxing the low level consistency checks performed by the STM for conflict detection in order to reduce the aborts incurred by the transactions [14, 13, 11, 19, 12, 17]. Early release [14] is a technique to allow certain memory locations from the read set to be excluded from further validation after a certain point in the transactional execution, which is much before the transaction commit point. Once a memory address has been released from the transactional read set using early release mechanism, other transactions can write to this address without leading to a conflict with the transaction which performed the early release. Typically early release is specified by the programmer. Hence the programmer should ensure that specifying early release for a memory location does not violate the transactional semantics. Thus early release while useful in reducing the conflicts encountered by an application, puts the burden of extra complexity on the programmer since he/she needs to decide when a memory location is safe for early release. Harris et al. [12] proposed the concept of Abstract Nested Transactions (ANT) in order to reduce the amount of roll-back costs due to low level conflicts. If there are conflicts encountered during execution of transactions which are specified as *ANT*, the *ANT* can be re-executed without the need for re-running the larger transaction that contains the *ANT*. Harris et al. [12] discuss a variety of benign conflicts encountered in transactional execution and show how they can be avoided using *ANT*.

Open nesting is a technique which also aims at reducing low level conflicts leading to spare aborts, by supporting abstract serializability of transactions [18]. Open Nested Transactions (ONT) distinguish between physical serializability and abstract serializability [18]. A set of concurrently executing transactions are set to be physically serializable if the resulting state of memory is consistent with some serial execution of these transactions. On the other hand, a concurrent set of transactions is said to be abstract serializable if the resulting abstract view of data is consistent with some serial execution of the transactions. *ONT* model supports abstract serializability using open nesting. Open nesting allows subtransactions to commit and become visible before outermost transactions containing them commit [18]. Yang Ni et al. [20] developed language support for *ONT* by specifying new language constructs for Java to support open nesting transactions. They point out that while open nesting transactions increase the scalability of concurrent data structures for long running transactions, it requires considerable programming complexity as the programmer needs to define abstract compensating action for each open nested transaction in order to support roll backs.

Herlihy et al. [13] proposed transactional boosting methodology for synchronization and recovery through data structure semantics instead of using read/write sets. Transactional boosting is intended to transform a large class of highly concurrent linearizable objects into highly concurrent transactional objects, as long as the linearizable implementation satisfies certain regularity properties [13]. Although transactional boosting enhances concurrency by relaxing constraints imposed by read/write semantics at low-level, it requires the programmer to identify the commutative operations and to define inverse operations for non-commutative ones.

Felber et al. [11] proposed *elastic transactions*, wherein the strict serializability requirement for the entire transaction is relaxed under this approach. Instead an elastic transaction consists of a sequence of mini-transactions with each mini transaction operating on a consistent set of data, but different mini-transactions of the same transaction can be operating on data which is not mutually consistent across mini-transactions. It is up to the programmer to decide when to use elastic transactions. Afek et al. [1] proposed *view transactions* as a new model for relaxed consistency checks. A *view transaction* always operates on a consistent state (snapshot) of memory, but may commit in a different snapshot than the one, in which it worked on, due to the relaxed consistency checks. With *view transactions*, it suffices for the programmer to ensure that the commit time snap shot must be such that had the transaction operated on it, the externally visible actions would be the same. However *view transactions* require that the programmer must identify the transaction's critical view.

Each of the above techniques relaxes STM consistency checks in different ways and have been shown to yield considerable performance gains. However they typically also require *programmer intervention* in terms of annotating a early release access [14] or marking a transaction as elastic [11] or providing inverses for operations in boosted transactions [13]. Requiring programmer intervention in relaxing the consistency checks defeats the essence of transactional memory paradigm to the extent that it brings back the burden of programmability back on the programmer, unlike our approach which is automatic and performed by the compiler.

Transactional scheduling has also been proposed as a means of reducing transactional aborts [24, 8, 10]. All these approaches typically require runtime profiling and contention monitoring information to make decisions. Hardware mechanisms have been proposed to selectively repair true conflicts arising due to auxiliary data updates which is typically independent of the rest of the transactional

computation, thereby preventing full re-execution due to aborts [2]. Ours is a purely software-based technique which does not require any hardware support.

6. CONCLUSION

In this paper, we proposed and evaluated a software based, automatic compiler aided hybrid approach known as SR-STM to handle certain kind of transactional conflicts which are amenable to repair by partial re-execution. We implemented our approach in Open64 compiler with TL2 as our underlying STM. We showed that our SR-STM scheme is effective in improving the performance for a set of STAMP benchmarks ranging from 3.24% to 22.34%.

7. REFERENCES

[1] Y. Afek, A. Morrison, and M. Tzafrir. Brief announcement: view transactions: transactional model with relaxed consistency checks. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 65–66, New York, NY, USA, 2010. ACM.

[2] C. Blundell, A. Raghavan, and M. M. Martin. Retcon: transactional repair without replay. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 258–269, New York, NY, USA, 2010.

[3] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. *SIGARCH Comput. Archit. News*, 35:81–91, June 2007.

[4] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[5] S. C. Chan, G. R. Gao, B. Chapman, T. Linthicum, and A. Dasgupta. Open64 tutorial. In *IPDPS*, page 1, 2008.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06: Proc. 20th International Symposium on Distributed Computing*, pages 194–208, September 2006.

[7] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 21–33, March 2007.

[8] S. Dolev, D. Hendler, and A. Suissa. Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of PODC*, pages 125–134. August 2008.

[9] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. *SIGPLAN Not.*, 44:155–165, June 2009.

[10] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *PODC '09: Proc. 28th ACM Symposium on Principles of Distributed Computing*, August 2009.

[11] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Proceedings of the 23rd international conference on Distributed computing*, DISC'09, pages 93–107, Berlin, 2009.

[12] T. Harris and S. Stipic. Abstract nested transactions. In *TRANSACT '07*, August 2007.

[13] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP '08*, pages 207–216, New York, NY, USA, 2008. ACM.

[14] M. Herlihy, V. Luchangco, M. Moir, and William Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03*, pages 92–101, July 2003.

[15] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. Rms-tm: a comprehensive benchmark suite for transactional memory systems. *SIGSOFT Software Engineering Notes*, 36:42–43, Sept. 2011.

[16] E. Koskinen and M. Herlihy. Checkpoints and continuations instead of nested transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 160–168, 2008.

[17] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2006.

[18] J. E. B. Moss. Open nested transactions: Semantics and support (poster). In *Workshop on Memory Performance Issues*, February 2006.

[19] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63:186–201, December 2006.

[20] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP '07*, pages 68–78, New York, NY, USA, 2007. ACM.

[21] L. Schaelicke and E. DeLano. Intel itanium quad-core architecture for the enterprise. In *EPIC '08: Proceedings of the Eighth Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology*, pages 21–33, April 2010.

[22] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[23] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, pages 1–11, New York, NY, USA, 1994. ACM.

[24] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA '08*, pages 169–178, June 2008.

[25] 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 *SPAA '08*, pages 265–274, June 2008.

[26] 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 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 285–294, New York, NY, USA, 2010. ACM.