An Automated Framework for Decomposing Memory Transactions to Exploit Partial Rollback

Aditya Dhoke
Virginia Tech
adityad@vt.edu

Roberto Palmieri
Virginia Tech
robertop@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

***Abstract*—In this paper, we present a framework that automatically decomposes programmer-written flat transactions into closed-nested transactions. The framework relies on two key mechanisms for the decomposition. The first is a static tool that analyzes application source code and produces a compact representation of transactions' business logic. The second is a runtime monitor that captures the actual contention level of shared objects and, relying on the outcome of the static tool, triggers the optimal closed-nested configuration for the workload at hand. We implemented this framework atop QR-CN, an open source fault-tolerant DTM written in Java. Our experimental studies conducted using the TPC-C, Vacation and Bank benchmarks reveal that the framework yields better performance than flat nesting and manual closed nesting, especially when the workload changes.**

***Keywords*-Transaction Nesting, Distributed Transactional Memory, Adaptivity**

I. Introduction

The ubiquity of multicore architectures is forcing application software developers to expose greater concurrency in software to exploit the increasing hardware parallelism. This is a challenging problem due to the programmability and scalability shortcomings (e.g., deadlocks, livelocks, composability) of traditional abstractions for concurrency control (i.e., coarse-grained or fine-grained locking, lock-free synchronization). Software Transactional Memory (STM) [1], [2], [3], [4], [5] is an emerging concurrency control abstraction that promises to alleviate these difficulties. STM is a software layer that provides transparent support for synchronizing activities of different threads operating on the same shared data. A programmer's only obligation is to demarcate application code blocks that require transactional execution properties such as atomicity, isolation, and consistency. The programmability and scalability shortcomings of lock-based synchronization are exacerbated in distributed systems due to multi-computer concurrency. Thus, similar to STM, distributed transactional memory (DTM) [6] is emerging as a promising distributed synchronization abstraction that offers high programmability.

In DTM, transaction execution time is longer than in centralized STM due to the multiple interactions with other nodes in the network. Aborting a transaction involves re-executing all the transactional operations performed before the transaction is aborted. In a centralized setting, transactions are comprised of in-memory operations, and re-executing a transaction after an abort event has lesser impact than in DTM on overall transaction execution. In fact, in DTM many operations require network access, such as retrieving new copies of objects from other nodes and, as a consequence, repeated re-executions due to aborts can significantly degrade performance.

One way to address this problem is through *transactional scheduling*, which optimizes the execution order of transactional operations to minimize conflicts [7], [8]. Unfortunately, when a conflict occurs, most of the time involved in triggering a transaction's abort is inevitable. An effective technique to alleviate the problem of restarting transactions from start after an abort is *partial rollback* [9], [10]. Here, an aborted transaction, instead of restarting from the start and re-executing in its entirety, saves intermediate states of its execution, and restarts from one such state, re-issuing only those operations that have been invalidated. Even though promising, partial rollback has the overhead of saving intermediate transaction execution states (or checkpointing), which can be significant.

A lightweight mechanism for supporting partial rollback is the *closed nesting model* [9], [10]. Closed nesting treats transactions as containers for inner (or sub-) transactions. While sub-transactions encounter conflicts during their execution, they are aborted independently of their enclosing or "parent" transactions, thereby potentially reducing the scope of rollbacks. When a sub-transaction commits, its state is merged with its parent's state, but it is not made globally visible until the parent commits. If a conflict is detected after the sub-transaction commits, the parent is aborted and re-issued together with all its sub-transactions. In contrast to classical checkpointing [11], in closed nesting, the scope of a rollback can only be chosen from among the enclosing transaction boundaries. A recent work [10] that compares checkpointing and closed nesting shows that the latter has lesser overhead and better performance in DTM.

A. Motivations

The closed nesting model assumes that the programmer manually decomposes transactions into a set of sub-transactions. Manually configuring sub-transactions to effectively exploit the partial rollback mechanism is non-trivial. There are three main factors that affect the effectiveness of closed nesting transactions: *1)* the nesting granularity in terms of the number of operations performed; *2)* the contention of the shared objects accessed; and *3)* the position in the source code of each closed-nested transaction. In the following, we provide an intuition of why these parameters are important and, in Section III, we detail the discussion. The following arguments point out the complexity of designing effective closed-nested transactions and, at the same time, they represent the motivations of our work because our proposal aims at solving these issues.

The first parameter is complex to predict because it is affected by transaction semantics. For example, it is useless to enclose transactional operations that depend on each other in different closed-nested transactions if no other shared objects are accessed (i.e., only local computation). Say T_{p1} a transaction performing the following operations: $\{ Read(O_A)$, $Read(O_B)$, $C = O_A + O_B$, $D = C + \phi \}$ where O_A, O_B are shared objects and C, D are private variables. When T_{p1} is decomposed into closed-nested transactions, the operation $D = C + \phi$ is always wrapped in the same sub-transaction of $C = O_A + O_B$ because any invalidation on O_A or O_B implies the re-execution of $D = C + \phi$. If we enclose $D = C + \phi$ in a separate sub-transaction, then when O_A or O_B become invalid, the entire T_{p1} will be re-executed, losing the gain of closed nesting. In contrast, say $T_{p2} = \{ Read(O_A)$, $Read(O_B)$, $C = O_A + O_B$, $Read(O_D)$, $E = O_D + C \}$, with O_A, O_B, O_D shared objects and C, E private variables. Here the operation $E = O_D + C$ involves another shared object (i.e., O_D). For this reason, it can be enclosed in a separate sub-transaction because an invalidation on O_D will cause the ex-execution of only the two operations: $Read(O_D)$ and $E = O_D + C$, preserving the previous computation (i.e., $Read(O_A)$, $Read(O_B)$, $C = O_A + O_B$), that is still valid.

The second parameter depends on the application's workload and therefore can dynamically change with the evolution of the system and the users' behavior. In the previous example, T_{p1} has two operations involving remote communication: $Read(O_A)$, $Read(O_B)$. The contention probability (P) of O_A and O_B changes over time depending on the workload. On one hand, when $P(O_A)$ is comparable with $P(O_B)$, these two operations could be enclosed in the same sub-transaction. On the other hand, when $P(O_A) >> P(O_B)$, then if both appear in the same closed-nested transaction, the invalidation of O_A will always cause the re-execution of $Read(O_B)$, even though O_B is still valid. In this scenario, the best configuration is splitting the two operations into two different closed-nested transactions. By this example, it is clear how complex it is to statically predict such phenomena.

The positions of the sub-transactions in parent transactions significantly affect closed nesting's effectiveness. In contrast to checkpointing, where, after a conflict, the transaction's execution can restart from any point, the closed nesting model is effective only when a conflict occurs on an object accessed for the first time in the sub-transaction currently executing. For example, sub-transactions that access highly contended objects can be moved closer to the transaction commit phase to save a large part of the transaction execution state, as this organization decreases the transaction's abort probability. In fact, when highly contended objects are accessed in a sub-transaction, say Ts_1, located at the beginning of its parent transaction Tp, then after Ts_1's commit, they become part of Tp's history (i.e., Tp's read-set as described in Section IV). At this stage, if an invalidation of those shared objects previously accessed by Ts_1 after the commit of Ts_1 occurs, it entails the re-execution of the entire Tp.

Moving such sub-transactions closer to the parent transaction's end reduces the abort probability because it reduces the total time in which their objects are stored in the parent's read- and write-sets. In this case, when an abort happens, almost the entire transaction is already executed and it does not need to be rolled back. Although important, this approach is doable only if transactions expose independent parts. In other words, the order of sub-transactions within a parent transaction, upon which all operations depend, cannot be changed.

B. Contributions

All three parameters are significantly influenced by the application workload. Thus, even though programmers may know the entire application business logic and the expected workload and therefore can manually define closed-nested transactions, it will not be effective when the workload changes at run-time.

Motivated by these observations, we propose *ACN*, an automated framework for defining closed-nested transactions at run-time according to the application's workload. ACN analyzes application source code and creates a graph-based representation of transactions' logic as defined by the programmer. Subsequently, it provides an initial definition of sub-transactions for each original transaction. At run-time, based on the workload dynamics, ACN adjusts the granularity of sub-transactions for maximizing the effectiveness of partial rollback according to the contention of objects accessed. ACN also identifies possible independent segments of a transaction's logic and configures the segment containing the most contended objects closer to the commit phase. This way, ACN transparently exploits the closed nesting technique for implementing partial rollback. Using ACN, programmers can take advantage of partial abort of transactions without the overhead of checkpointing and the burden of manual definition of closed-nested transactions.

We implemented ACN and integrated it into QR-CN [10]. QR-CN is a fault-tolerant DTM framework which supports manual closed nesting. The integration produces QR-ACN. Note that the ACN methodology can also be applied in non-DTM systems. In this paper we selected DTM as a baseline transactional system because of the significant effectiveness of partial rollback on improving DTM performance as shown in [10]. Also, the great programmability of STM/DTM motivates ACN because ACN hides to the programmer the exploitation of closed nesting as a partial rollback technique.

In order to conduct a fair and exhaustive evaluation, our study spans from scenarios where closed nesting is effective, to configurations where its usage does not improve the overall performance significantly. ACN is a framework that optimizes the effectiveness of closed nesting, therefore its applicability is not limited to specific cases whereas its performance impact is relevant when the partial rollback mechanism pays off. In fact, there are applications, as well as workloads, in which the partial abort technique does not provide benefit over restarting the transaction from the very beginning. In such cases, ACN can still be applied but the expected performance improvement is not substantial. Nevertheless, as we will show through experimental results, in those cases the impact of ACN's overhead is minimal, thus, it can still be adopted without degrading application performance.

We evaluated QR-ACN using benchmarks commonly used for assessing performance of transactional systems such as TPC-C [12], Bank and Vacation [13]. Results reveal that QR-ACN improves performance over flat nesting transactions by as much as 120% and over manual closed nesting by 83%. In addition, running configurations where the partial rollback is not effective, QR-ACN guarantees performance similar to flat nesting, thus exposing minimal overhead.

To the best of our knowledge, this work is the first to provide a methodology and a practical framework for defining closed-nested transactions automatically. In addition, as detailed in Section III, this paper provides also the first analysis of factors affecting closed-nesting transactions' effectiveness.

II. Background

A. Transaction Nesting Models

Transactions are nested when they appear within another transaction's boundary. Transactional nesting makes code composability easy: multiple operations inside a transaction are executed atomically without breaking encapsulation. Three transactional nesting models have been proposed in the literature: Flat, Closed, and Open. In this paper, we focus on the first two, because open nesting cannot be used for aborting transactions partially.

Flat nesting, which is the simplest form of nesting, simply ignores the existence of transactions in inner code. All operations are executed in the context of the parent transaction. Aborting any inner transaction causes the parent transaction to abort. Thus, no partial rollback is possible with this model. Clearly, flat nesting does not yield any performance improvement over non-nested transactions.

Closed nesting [14], [15], [10] allows inner transactions[1] to abort individually. Aborting an inner transaction does not necessarily abort the parent transaction (i.e., partial rollback is possible). However, inner transactions' commits are not visible outside of the parent transaction. An inner transaction commits its changes only into the private context of its parent transaction, without exposing any intermediate results to other transactions. The shared state is modified only when the parent transaction commits, thus the closed nesting model does not affect the system's correctness. If an object accessed within the context of an inner transaction becomes invalid after the commit of the inner transaction, the entire transaction (i.e., the parent) aborts and restarts from the very beginning.

[1]We use the terms inner transaction, sub-transaction, and closed-nested transaction synonymously.

B. The QR-CN Protocol

Dhoke *et al.* [10] present QR-CN (Quorum-based Replication protocol with Closed Nesting), a fault-tolerant DTM protocol that supports closed-nested transactions. QR-CN is an extension of QR-DTM (Quorum-based Replication protocol) [16], a fault-tolerant DTM concurrency control protocol that uses quorums for managing transactional meta-data. QR-DTM uses full replication: each object is replicated on all nodes. Each node is designated a read quorum and a write quorum, where a quorum is a set of nodes having specific properties. A read quorum services a transaction's read and write requests on objects, while a write quorum is used to commit changes to objects through two-phase commit. A transaction executing on a node uses the read and write quorums designated for that node for executing its operations. In QR-CN, each transactional access implies a remote call to quorum nodes.

The QR-DTM protocol ensures 1-copy serializability [17]. In order to retrieve the latest copy of accessed objects, QR-DTM leverages the quorum intersection property, namely any write quorum and read quorum always intersect [18]. This property also helps to provide incremental validation, i.e., on each read request for an object, the previously read objects are validated. This method is useful in detecting aborts early during transaction execution. The nodes in QR-DTM form a logical ternary tree. QR-DTM uses Agrawal *et al.*'s [18] procedures for creating read and write quorums. A read quorum is the majority of children at a level of the tree, while a write quorum is the majority of children at every level.

III. Effectiveness of Closed Nesting

The main distinguishing aspect between closed nesting and checkpointing – as a means for implementing partial abort – is the transaction's rollback point. With checkpointing at the finest granularity (i.e., saving the transaction state whenever the transaction issues the first read operation on a shared object), when a conflict occurs due to an invalid read, the transaction restores the execution from the checkpoint prior to the invalid read. With closed nesting, even though the implementation overhead is minimal, a parent transaction exploits the advantage of partial rollback only when the invalid object is accessed by the currently executing closed-nested transaction. If the object has been read by a previous closed-nested transaction, then closed nesting is ineffective, and the transaction must be re-executed from its very beginning.

The nature of the closed nesting model presents important design choices for the programmer, for making this lightweight partial rollback strategy effective. These include:
Granularity. The size of the closed-nested transactions is a factor responsible for partial rollback's effectiveness. On the one hand, wrapping each read operation in a sub-transaction (the finest grain) results in poor performance gain, because, the probability of detecting an object invalidation while executing a sub-transaction composed of only one shared object accessed is very low. With this granularity, likely most of the invalid

objects are already accessed for the first time in a previous sub-transaction. Therefore, those aborts are handled by restarting the transaction from its very beginning, instead of somewhere in the middle. On the other hand, when the size of sub-transactions is large, the number of valid operations that are re-executed due to invalidations reduces the benefits of partial rollback. In fact, adopting the maximum granularity for sub-transactions is equivalent to use the flat nesting model (or classical, non-nested, transactions). Finding the proper trade-off between these two end-points is a way for increasing performance using closed nesting.
Contention probability. The contention degree of each shared object significantly impacts closed nesting's effectiveness. Heavily contended objects (also called "hot spots") are the main cause of a transaction abort. When operations on hot spot objects are combined with those on less contended shared objects in the same sub-transaction, valid operations (mostly those on non-contended objects) are re-executed when an abort occurs due to the invalidation of a hot spot object. In this case, partial rollback's effectiveness is reduced, along with its performance gain. Operations on system's hot spots should be separated from operations accessing less contended objects to reduce the number of valid operations to rollback.
Code repositioning. The position of transactional operations is defined by the programmer; however there are parts of a transaction that can sometimes be moved within the transaction's boundary because they are independent from other parts (i.e., there is no data-dependency between them). The basic property of transaction processing is that the whole transaction is executed "all or nothing", therefore repositioning such independent parts neither changes the transaction logic, nor affects transaction correctness. The closed nesting technique can take advantage of such code repositioning – in transactions where it is possible – for increased performance. In closed nesting, when an object accessed by a committed sub-transaction is invalidated, the entire transaction has to re-execute. As discussed before, such re-executions can be mitigated by shifting independent sub-transactions closer to their parent's commit phase (i.e., at the end of an atomic section). The effectiveness of code repositioning is particularly evident when combined with the contention probability of independent sub-transactions. Moving sub-transactions that access system's hot spots closer to their parent's commit phase significantly reduces the transaction's invalidation probability. This benefit is because contended objects, that cause invalidation, are not accessed at the beginning of the transaction but close to its end. As a consequence, when an abort occurs during the execution of the shifted sub-transaction, almost all of the original (i.e., parent) transaction has already been executed, reducing the amount of work to be re-done.

These factors represents guidelines that the programmer should consider while defining closed-nested transactions. Combining all these factors is critical for effectively exploiting closed nesting as a partial rollback mechanism. However, given the number of trade-offs that must be considered when these factors are combined, effectively using closed nesting could be difficult for a programmer, and sometimes impossible, when the application workload changes over time. Our contribution, the ACN framework, monitors the current contention probability of the shared objects and adjusts sub-transactions' definitions at run-time to find the best trade-off among the aforementioned three factors.

IV. System Model

We consider a distributed system, which consists of nodes that communicate by message-passing links. A set of *distributed transactions* $\mathcal{T} := \{T_1, T_2, \ldots\}$ sharing objects $\mathcal{O} := \{o_1, o_2, \ldots\}$ distributed over the network is assumed. A transaction consists of a sequence of requests, each of which is a read or a write operation for an object, followed by a commit operation. An object has the following meta-data that are used for QR-CN's operations:

- *Version number* maintains the object's version number and is used during object validation.
- *Protected* is a boolean field. When *true*, read or write to the object is disabled until commit is complete, after which it is set to *false*.

According to QR-CN, write operations, after fetching the requested objects, are buffered in the *write-set* and the return values of read operations are logged in the *read-set*. Read-set and write-set are private (i.e., non-shared with other transactions) memory spaces. Writes are applied to the shared state at commit time, after acquiring locks on the read-set's elements. A two-phase-commit protocol is used at commit time for finalizing the commit procedure. The complete description of QR-CN and QR-DTM can be found in [10] and [16] respectively. Hereafter, we refer to the requesting transaction as *client*, and the quorum node as *server*.

ACN has been designed for decomposing transactions with only one level of nesting, which means that when a sub-transaction T_S is defined, no further nested transaction are allowed in the context of T_S. In transaction processing, due to the complexity of defining nested transactions, the common model is using only one level of nesting (e.g., [14], [15]).

V. ACN: Automated Closed Nesting

As [10] shows, closed nesting can improve performance of fault-tolerant DTM. However, in certain cases, performance can degrade due to large numbers of partial aborts. Therefore, it is important to appropriately compose closed-nested transactions to effectively utilize the benefits of partial aborts. In order to determine the correct composition of closed-nested transactions, one needs to know the degree of contention of the objects involved in transaction execution. Contention of an object depends on factors such as the write workload, access pattern, and the number of servers (i.e., quorum nodes) and clients in the system. Given the dynamic nature of these factors, contention of objects can only be determined during application run-time. Therefore, it is complicated for a programmer to effectively compose closed-nested transactions while taking into account these run-time behaviors.

To this end, we develop an automated closed nesting framework, *QR-ACN*, which dynamically (re-)determines the composition and order of closed-nested transactions. We follow a hybrid approach where we use static and dynamic analyses to determine the closed-nested transactions' composition. Static analysis involves data dependency analysis of transactional code at compile-time, while dynamic analysis involves measuring the contention of objects at run-time. The information collected from these analyses are used to properly compose closed-nested transactions.

A. Overview

The basic idea behind automated closed nesting is exploiting the contention probability of shared objects accessed for guiding the definition of sub-transactions. In fact, we can summarize ACN's activities in the following two steps: *1)* it defines the size of a sub-transaction depending on the contention of its accessed objects; and *2)* it moves independent sub-transactions that access heavily contended objects closer to the parent's commit phase. By doing so, sub-transactions accessing low contended objects are executed at the beginning of parent's execution while those sub-transactions containing accesses to high contended objects are executed later. These two steps are then repeated periodically such that the sub-transaction's definition is refreshed according to the application workload.

We illustrate this with an example. Consider a bank transaction (of Bank benchmark), where funds are transferred from a bank account *Account1* belonging to branch *Branch1* to another account *Account2* belonging to branch *Branch2*. The pseudo-code for this transaction, written as a flat-nested transaction, is shown in Figure 1. Lines 2-5 correspond to reading remote object copies of the branch objects *Branch1* and *Branch2*, and withdrawing and depositing amounts from them, respectively. Lines 6-9 correspond to reading remote object copies, and withdrawing and depositing on *Account1* and *Account2*, respectively.

```
1   T_FLAT:
2           branch1 = getRemote(branchId1);
3           branch2 = getRemote(branchId2);
4           branch1.withdraw(amt1);
5           branch2.deposit(amt2);
6           account1 = getRemote(accountId1);
7           account2 = getRemote(accountId2);
8           account1.withdraw(amt1);
9           account2.deposit(amt2);
10          if !commit()
11                retry T_FLAT;
```

Fig. 1. Example of flat-nested Bank transaction.

Objects *Branch1* and *Branch2* are globally shared objects for their respective branches, hence, other transactions will also access them. Thus, at run-time, they will be highly contended. On the other hand, objects *Account1* and *Account2* will have low contention as they would have lesser number of accesses. Thus, the contention levels of the branch and account objects will differ significantly. For this transaction, having the branch operations (lines 2-5) and the account operations (lines 6-9) inside the same closed-nested transaction would not yield any performance improvement over flat nesting. That is because the probability of aborting while executing the branch operations is higher, thus, each time one of these objects (i.e., *Branch1* and *Branch2*) is invalidated by another committing transaction, the entire sub-transaction is restarted, repeating operations on *Account1* and *Account2* that are likely still valid. Adopting this configuration, the behavior would be the same as that of flat nesting.

```
1   T_PARENT
2           account1 = getRemote(accountId1);
3           account2 = getRemote(accountId2);
4           account1.withdraw(amt1);
5           account2.withdraw(amt2);
6   T_CLOSED
7                   branch1 = getRemote(branchId1);
8                   branch2 = getRemote(branchId2);
9                   branch1.withdraw(amt1);
10                  branch2.deposit(amt2);
11                  if (!commit())
12                          retry T_CLOSED
13          if !commit()
14                retry T_PARENT;
```

Fig. 2. Modified flat-nested Bank transaction.

Furthermore, if we look closely at Figure 1, there is no data dependency between the branch operation and the withdrawal operation. We can potentially change the order of these operations as shown in Figure 2. With this change, the branch operations are closer to the commit phase. We observe that the transaction is more likely to encounter conflict due to branch objects. Hence, we wrap the branch operation inside a closed-nested transaction, as shown in Figure 2 (lines 7-12). The conflict encountered in the closed-nested transaction can now be resolved using partial abort and restarted from line 7. Operations already performed on *Account1* and *Account2* are still valid and they do not need to be re-executed.

Note that changing the order of the operations will not affect the correctness of the transaction because it is executed atomically. The functional behavior of the transaction still remains the same. In fact, in both the cases, the transaction read-set and write-set are exactly in the same state, and the changes are applied during the commit phase. Moreover, a transactional operation doing only local computation, e.g., line 10 in Figure 2, depends on the shared object globally retrieved before, e.g., line 8. These operations should be enclosed in the same sub-transaction such that, when the accessed object becomes invalid, both can be re-executed without involving the re-execution of the whole transaction.

In order to devise a framework that automatically determines the composition of closed-nested transactions, we need a mechanism that detects data dependencies and contention level of objects.

B. ACN Design

QR-ACN can change the composition of closed-nested transactions dynamically depending upon the contention level of the objects accessed. To achieve this goal, transaction code has to be transformed during initialization. We now introduce a few definitions to illustrate how transactional code execution happens in QR-ACN.

UnitBlock. We divide transactional code into distinct sections called *UnitBlock*s. A UnitBlock is the smallest logical unit of code in QR-ACN, and it comprises of exactly one remote object invocation. The contention level of a UnitBlock is the contention level of the remote object it opens.

We consider the UnitBlock as a logical unit of code because it contains the first access to a shared object. When the transaction aborts due to the invalidation of an object o accessed, the partial abort mechanism restores the transaction's execution from a point before performing the read operation on o such that the invalid object (i.e., o) can be refreshed with the new version in the system. In closed nesting, when an object inside the sub-transaction is invalidated, the sub-transaction is restarted, including the operation for retrieving the object. For this reason we consider a UnitBlock composed of the first transactional read on a shared object.

Transactional operations resulting in only local computation on the shared object accessed are included in the same UnitBlock containing the access to the object. Clearly, there are local operations accessing more than one shared object (e.g., $C = A + B$ where A and B are shared objects). In this case, the read operation on A will be enclosed in a UnitBlock ($UnitBlock_A$); the read operation on B will be enclosed in another UnitBlock ($UnitBlock_B$); and the operation $C = A + B$ will be included in the latest sub-transaction preceding the operation (e.g., in a transaction composed of: $\{Read(A), Read(B), C = A + B\}$, the local operation $C = A + B$ will be wrapped in the same UnitBlock as $Read(B)$'s, while $Read(A)$ will define a new UnitBlock.

Block. Multiple UnitBlocks can be combined to form a Block. ACN leverages on the actual contention level of adjacent UnitBlocks for making the decision about merging the UnitBlocks into a single Block. ACN allows programmers to provide custom models for calculating the contention level of a Block starting from the contention level of all the objects accessed in its UnitBlocks. In this paper we approximate the object's contention level as a means for defining the abort probability of a transaction accessing that object. With this approximation, we rely on a simple and fast-to-compute analytic model for calculating the abort probability of the entire Block, using the same methodology presented in [19]. A Block enclosing multiple UnitBlocks represents a piece of code to be executed.

Executor Engine. This module is responsible for maintaining the sequence of Blocks that comprises a transaction and for executing those Blocks in that order. In order to execute a Block, the module executes all the UnitBlocks enclosed inside it. The sequence of Blocks can be changed over time, depending on the workload. Note that the sequence can be changed only by the Algorithm module (see below).

Figure 3 shows the Bank transaction arranged in UnitBlocks and Blocks. In the original Block sequence (Figure 3(a)), each Block contains one UnitBlock which in turn consists of one remote object invocation. As already stated in Section V-A, `branch` objects are more contended than account objects. For this reason, UnitBlocks corresponding to branch objects are merged inside a single Block (i.e., *B2*). Similarly, `account` objects are merged into Block *B1*. B1 and B2 will define two different closed-nested transactions.

The different contention levels of B1 and B2 also trigger a change in the order of sub-transactions. Moving B2 closer to the parent transaction's commit increases the effectiveness of closed nesting because, after an invalidation of `Branch1` or `Branch2`, the entire B1 is still valid and it does not need to be re-executed. In contrast, leaving B2 before B1 results in re-processing B1 each time an object on B2 is invalidated while B1 is executing.

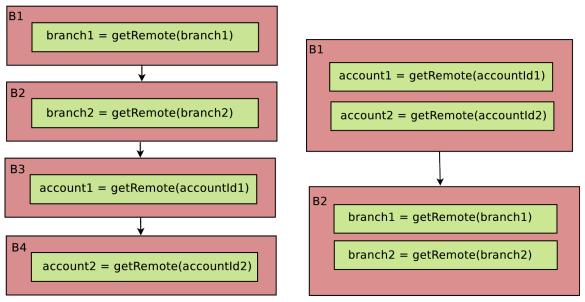

B1
branch1 = getRemote(branch1)
B2
branch2 = getRemote(branch2)
B3
account1 = getRemote(accountId1)
B4
account2 = getRemote(accountId2)
B1
account1 = getRemote(accountId1)
account2 = getRemote(accountId2)
B2
branch1 = getRemote(branch1)
branch2 = getRemote(branch2)
(a) Original Block Sequence (b) Modified Block Sequence

Fig. 3. Code Arrangement using UnitBlocks and Blocks.

We now describe the core modules of QR-ACN.

Static Module. This module maintains static information of transaction code. It is triggered at the beginning of the application and creates a graph model of transaction code, called *UnitGraph*. During run-time, the graph model is queried by the *Algorithm Module* for detecting data dependencies.

Dynamic Module. This module collects run-time parameters such as objects' write and abort ratios and feeds them as input to the *Algorithm Module*.

Algorithm Module. This module is invoked periodically. It accepts input from the *Static Module* and the *Dynamic Module* and processes it to produce a sequence of Blocks. (We describe the algorithm used to compute the Block sequence in Section V-C.) The sequence is then input to the *Executor Engine*, which executes the Blocks in that order.

C. Algorithm

1) Data Dependency Analysis: We use the Soot [20] Java source code analysis framework for dependency analysis of transactional code. Soot converts Java classes into four different intermediate representations, each of which can be used for different kinds of program analyses. The framework creates a control-flow graph, called *UnitGraph*, where a node represents a program statement and an edge represents the control flow between the nodes. In order to determine the data dependency among the nodes, we perform data flow analysis on top of the UnitGraph. Specifically, for every node in the UnitGraph, the in-flow and out-flow data is tracked to create data dependency edges among the nodes.

Next, we identify the invocations to access remote objects from the UnitGraph. This task is straightforward because in QR-CN, as well as in QR-ACN, the first read and write operation on a shared object always involves remote interaction for retrieving the latest version of the object. *UnitBlocks* are created from those UnitGraph nodes that contain remote invocations. Each UnitBlock is associated with the remote object's class and is registered in the ACN Module. Subsequently, all the transactional operations performing only local computations are assigned to the just-defined UnitBlocks. Each one is enclosed to the latest UnitBlock that contains the access on a shared object managed by the local operation. If a local operation $Oloc_i$ does not manipulate any shared object, it is necessarily depending on another local operation $Oloc_j$ (or via a chain of local operations) that manages a shared object (note that operations completely independent from the rest of the code are usually not enclosed in transactions). In this case, $Oloc_i$ will be merged with the UnitBlock containing $Oloc_j$.

As an example, say T a transaction performing: $\{Read(A)$, $Read(B)$, $Read(C)$, $Read(D)$, $var = A + B$, $var = var/2$, $Read(E)$, $var2 = E+B\}$. Using the UnitGraph, the operation $var = A + B$ will be part of the UnitBlock containing $Read(B)$ because it is the last UnitBlock where the read operation on A or B has been performed. Following the same rule, the operation $var2 = E + B$ will be included in the UnitBlock containing $Read(E)$. The operation $var = var/2$ does not manipulate any shared objects but it has a dependency with the operation $var = A + B$. In this case, $var = var/2$ will be enclosed in the same UnitBlock that contains $var = A + B$. Associating local computation to a UnitBlock is again straightforward by simply analyzing the UnitGraph flows.

Dependencies between the UnitBlocks are registered. If there is a dependency between nodes belonging to different UnitBlocks, then there is a dependency between the UnitBlocks as well. Each UnitBlock has an ID, which is its index in the sequence of UnitBlocks. At this point, the transaction source code is automatically transformed: a new function is defined with the Block (see Section V-B for the definition of Block) to be executed as the input. The body of this function consists of all the UnitBlocks. A UnitBlock in the function body is executed only when the Block input to the function contains that UnitBlock's ID. Figure 3(b) provides an example of this transformation.

Block Sequence is the sequence of Blocks that can be executed without altering the transaction's semantics. During initialization, a Block is created from a single UnitBlock and the sequence found in the UnitGraph is followed. At run-time, UnitBlocks can be merged according to their contention level, creating bigger Blocks. Thus, a Block can consist of multiple UnitBlocks. This Block Sequence forms the input to the Executor. Each Block represents a closed-nested transaction.

This configuration is the first used when the system is deployed. It does not represent the optimal configuration but, by using only the static analysis, run-time behaviors like objects' contention cannot be captured. For example, a transaction T=$\{Read(O_A)$, $Read(O_B)$, $var = O_A + O_B\}$ results from the static analysis as two Blocks, each with one UnitBlock: BL_1=$\{Read(O_A)\}$ and BL_2=$\{Read(O_B), var = O_A+O_B\}$. The operation $var = O_A + O_B$ is enclosed in BL_2; however it could also be enclosed in BL_1, moving the resulting BL_2 before BL_1. This way, if the contention level of BL_1 is higher than BL_2, BL_1 can be moved closer to T's commit phase, increasing the effectiveness of closed nesting (and vice-versa). With the configuration resulting from the static analysis, BL_1 has a data dependency with BL_2 because BL_2 uses an object (i.e., O_A) read by BL_1, therefore BL_1 cannot be moved after BL_2. For addressing this problem, the composition of subtransactions will be further refined at run-time, taking into account the actual application workload.

According to the evolution of an application's workload, some UnitBlock can also be extracted from its Block to become a new, smaller Block. The data dependency analysis generates the so-called *dependency model* for each transaction.

2) Contention Level: The contention level of each shared object is the parameter we use for identifying system's hot spots. However QR-ACN is flexible in doing so, as it allows programmers or system administrators to provide a custom model for calculating the contention level. We use the term "contention level" as an indicator for representing when an object is a hot spot. Different programmers can classify hot spots in different ways, therefore QR-ACN offers the opportunity to provide custom characterization for them.

The purpose of QR-ACN is not providing a methodology for profiling system's hot spots, although, for the sake of completeness, we overview a simple analytical model that we used during the evaluation of QR-ACN. We approximate the contention level of a shared object according to the number of write operations occurred on that object since the last observation. This information is maintained by quorum nodes. Whenever a transaction, during its commit phase, sends a write request of an object, the node increases a counter associated to that object. The algorithm for assessing the compliance with the current composition of closed-nested transactions is invoked periodically (as we will illustrate in Section V-C3). Each period represents a time window. Moving from one time window to the next one implies resetting the counters of write operations. This way, at any point in time, the contention level of an object is calculated as the number of write requests happened in the last time window. The greater this value, the greater the contention level for that object.

For obtaining the sequence of Blocks to provide better performance, the clients request the contention level of accessed objects. They create the list of objects that are contained in the Block sequence. This list is sent as part of the request to the quorum nodes. Upon receiving the request, the quorum nodes determine the contention level of an object as the number of write operations in the last time window, and the list of contention levels is sent back to the requesting client. This process introduces just a small overhead because meta-data are coupled with existing network messages, which slightly increases the network transmission delay. However, messages are compressed such that this additional cost is minimized.

A Block could be composed of multiple UnitBlocks merged together. As a result, this Block will contain a number of accesses to shared objects equal to the number of UnitBlocks merged. In this case, QR-ACN should calculate the contention level of the whole Block. However, as we will show in the next sub-section, the algorithm for creating Blocks from UnitBlocks always starts decomposing all merged Blocks into UnitBlocks, as in the original version after analyzing the UnitGraph. For this reason, QR-ACN only takes into account the contention level of single object because each UnitBlock is composed of only one access to a shared object.

We approximate the contention level of a shared object with the abort probability of a transaction accessing only that object. In fact, when the contention level of an object o is high, the probability that a transaction (or a sub-transaction) accessing o is aborted increases. More complex analytical models can be adopted for determining the contention level and the abort probability of UnitBlocks. An example of these models applied to STM can be found in [19]. However, all of them should consider the time needed for solving the model itself. Expensive computations are usually not suited for on-line transaction processing. Our solution is fast and provides a good representation of the actual contention level of an object.

3) Algorithm Module: The Algorithm Module is triggered periodically on the client nodes. The input of this module is the current Block sequence, the contention level of each object within a Block, and the dependency model (see Section V-C1 for the definition of dependency model) between the Blocks. The module output is the new Block sequence. The dependency model is created during the initialization of the system, while the contention level of objects (at that point in time) is requested from the quorum nodes.

The Algorithm module works on the basis that highly contended objects (hot spots) should be as close to the transaction commit phase as possible. In order to achieve this, the module attempts to compose and shift Blocks accessing highly contended objects, while maintaining data dependencies among Blocks. The process has three steps:

Step 1. In the first, the algorithm seeks a new Blocks configuration such that it can exploit the current objects' contention levels. The current Block sequence is discarded and merged Blocks are split again. All the local operations (managing shared objects or depending on other local operations that manage those objects) are separated by their current UnitBlocks. For each local operation *Oloc*, the algorithm seeks the most contended UnitBlock that contains the access to one of the objects managed by *Oloc*. This way, UnitBlocks accessing highly contended objects also contain their dependent local operations. The first step enables the possibility to move Blocks closer to the transaction's commit phase. This step also solves the problem of static analysis, as reported at the end of Section V-C1.

Step 2. The second step seeks to merge adjacent dependent UnitBlocks that expose similar (less than a configured threshold) contention levels. This is because, in case they are dependent and with a similar contention level, they can: *i)* be moved together closer to the transaction's commit phase; *ii)* maximize the effectiveness of partial rollback because an invalidation of one of their objects accessed causes the re-execution of only the new bigger Block instead of the (possibly) entire transaction due to the presence of two different UnitBlocks.

Step 3. The third step sorts the new Blocks depending on their current contention levels. Starting from the lowest contention level, each Block is shifted such that all the Blocks executing before it have lower contention levels, while preserving the data dependency.

As we will show later (Figure 4(d)), the overhead of this algorithm is limited because, usually, transactions' sizes are not as big to make its computation unfeasible. This algorithm is executed asynchronously and periodically by clients running the transactional applications.

VI. Experimental Evaluation

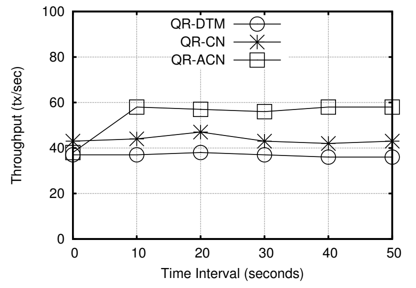
We implemented QR-ACN extending QR-CN [10]. We evaluated QR-ACN using three benchmarks widely adopted in transaction processing such as TPC-C [12], Bank and Vacation [13]. TPC-C [12] is the popular On-line Transaction Processing (OLTP) benchmark emulating an order processing system for a wholesale supplier with multiple districts and warehouses; Bank emulates a bank application; and Vacation is a transactional application from the famous STAMP suite [13] that mimics the process of booking vacations.

Our test-bed consists of up to 30 nodes, 10 serving as servers and up to 20 as clients, interconnected using a 1Gbps switched network. Each node is an AMD Opteron machine with eight CPU cores running at 1.9GHz. We do not show results increasing the number of nodes because an evaluation study of QR-CN is already presented in [10]. We measured the throughput on client nodes as transactions committed per second. We compared the throughput of QR-DTM and QR-CN (i.e., manual closed nesting) with QR-ACN. We ran QR-ACN's algorithm for assessing the effectiveness of the current closed nesting configuration every 10 seconds, and measured the system throughput for every 10 second time interval. Every datapoint reported is the average of four repeated experiments.

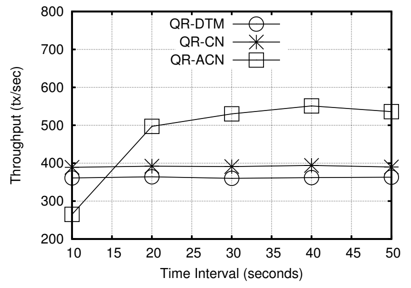
In this evaluation we span scenarios where closed nesting is particularly effective (e.g., Figure 4(b)), as well as scenarios where this partial rollback mechanism does not pay off in terms of performance improvement (e.g., Figure 4(d)). This way we also provide a hint about the applicability of ACN on other benchmarks.

A. TPC-C Benchmark

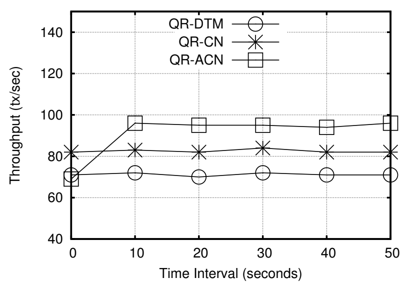
We compared QR-DTM, QR-CN, and QR-ACN for write intensive workloads on the TPC-C benchmark such that the partial abort mechanism can be effective on the overall performance. We configured the clients to generate 100% *NewOrder* transactions of the TPC-C specification. Figure 4(a) shows the throughput of QR-DTM, QR-CN, and QR-ACN for different time intervals. We observe that QR-DTM and QR-ACN have the same throughput up to $t = 10$ sec because QR-ACN is still



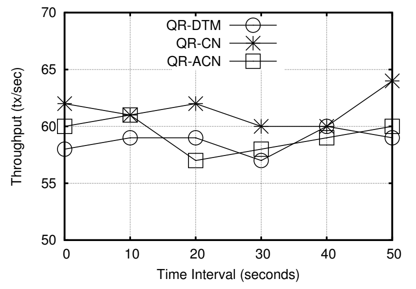
(a) 100% NewOrder.



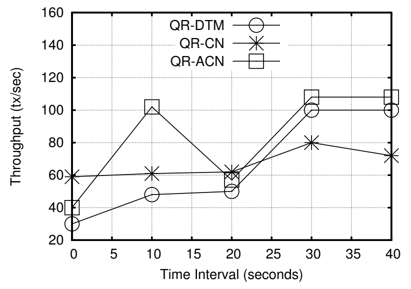
(b) 100% Payment.



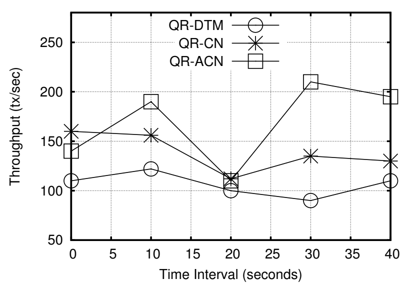
(c) 50% Payment and New Order.



(d) 100% Delivery.



(e) Vacation Benchmark.



(f) Bank Benchmark.

Fig. 4. Performance results with TPC-C, Vacation and Bank benchmarks.

monitoring the workload for figuring out the best closed nesting configuration. In this first observation, QR-CN outperforms the others. In the next time interval, QR-ACN determines the heavily contended objects (*District* in this case). The transactional code as per the TPC-C specification performs the remote operations initially in the execution. QR-ACN moves the remote operation on District as close to the commit phase as possible, while satisfying data dependencies. Additionally, QR-ACN merges the blocks with similar contention levels. This adjustment results in throughput improvement of 53% over QR-DTM and of 38% over QR-CN.

We performed a similar comparison for *Payment*, another write-transaction profile of the TPC-C specification. The throughput variation with time is shown in Figure 4(b). We observe that, initially, QR-ACN's throughput is lower than that of flat nesting and manual closed nesting, because the initial configuration of closed-nested transactions is not effective for partial abort. However, after t = 10 sec, the composition is changed. QR-ACN determines the high contention objects to be *District* and *Warehouse*, and shifts them closer to the commit phase. This shift results in a throughput improvement of 53% over QR-DTM and 45% over QR-CN.

Next, we compared QR-DTM, QR-CN, and QR-ACN for the TPC-C profile of 50% of Payment and NewOrder transactions. The results are shown in Figure 4(c). We observe that after QR-ACN "kicks in" the throughput improvement over QR-DTM is 28% and that over QR-CN is 9%.

TPC-C has another write-transaction profile, *Delivery*, which accesses objects such that the difference between their contention levels is not significant (all the objects have similar low contention levels). In this scenario, the closed nesting technique (QR-CN and QR-ACN) is not effective. This can be seen from the Figure 4(d), as there is no major difference in performance of QR-DTM, QR-CN and QR-ACN. However, this plot assesses the minimal overhead of QR-ACN with respect to the manual QR-CN. Even though QR-CN is mostly better than QR-ACN, this gap is always less than 3%, highlighting that the overhead of QR-ACN does not significantly deteriorate the classical closed nesting performance.

B. Vacation Benchmark

We evaluated QR-ACN for the Vacation benchmark, an application of the STAMP benchmark suite [13]. In our setup, we changed the contention of objects during different time intervals. Figure 4(e) shows the variation of throughput for QR-DTM, QR-CN, and QR-ACN for the time intervals. The highly contended objects change in the second and fourth time intervals. At the beginning of the experiment, QR-CN has the best performance, because QR-ACN is still monitoring the workload. Subsequently, in the second time interval, we observe that QR-ACN adjusts dynamically to the changing workload, and yields a throughput improvement of 120% over QR-DTM and 35% over QR-CN, whose throughput remains the same. In the fourth time interval, QR-DTM's throughput increases as its existing composition favors the changed workload. QR-ACN still outperforms QR-DTM by 8%. QR-DTM and QR-CN do not change their behavior depending on the objects' contention levels. Thus, their effectiveness is unpredictable when the workload changes.

C. Bank Benchmark

We configured the benchmark to generate 90% of write transactions and, as in Vacation, we changed the contention of objects in the second and fourth time intervals. At the beginning, QR-CN performed the best because the configuration suggested by the static analysis of QR-ACN does not take into account contention level. In fact, in the subsequent data-point, QR-ACN moves branch operations closer to the transaction's commit phase and defines two sub-transactions wrapping account operations and branch operations respectively, maximizing the effectiveness of partial rollback. As the results in Figure 4(f) show, we can observe better performance of QR-CN at the beginning but subsequently QR-ACN optimizes sub-transactions such that the gain becomes up to 55%.

VII. Related Work

Transactional nesting has been studied for TM, but largely in the multiprocessor context. Harris *et al.* [21] argued that closed-nested transactions, supporting partial rollback, are important for implementing composable transactions, and presented an `orElse` construct that relied on closed nesting. In [22], Adl-Tabatabai *et. al.* presented an STM that provides both nested atomic regions and orElse, and introduced the notion of mementos to support efficient partial rollback. All of these works assume manual definition of closed-nested transactions and they do not consider the problem of system hot spots while creating sub-transactions.

Herlihy and Koskinen [11] proposed checkpointing and partial aborts (in multiprocessor TM), as an alternate to nesting. They argued that fine-grained checkpointing can be achieved and closed nesting is a more rigid alternative. However, checkpointing needs to save and restore the processor context to resolve conflicts, and therefore may not be suited for all platforms. Rainbow OS [23] proposed to checkpoint the entire DTM system state, which can be used in case of node failures. Note that this is different from the concept of transactional checkpointing discussed in this paper.

Code transformation and analysis have been widely applied in the context of concurrent applications. The domination locking protocol [24] transforms sequential code to fine-grained locking. The technique applies to systems where the shape of memory forms a forest. CONCURRENCER [25] is a code refactoring tool, which helps to transform sequential Java code into that using the `java.util.concurrent` library in an error-free and omission-free manner. CalFuzzer [26] implements a two-phase active testing framework for concurrent programs. Soot [20] is a Java code analysis framework that is widely used. JDBC checker [27] checks the correctness of dynamically-generated query strings, and internally uses Soot. ACN relies on Soot for building transactions' dependency graphs and identifying independent segments.

VIII. Conclusion

Our work reveals that it is indeed possible to take the programmer out of this loop: transactions can be automatically decomposed into closed-nested transactions with effective performance. Additionally, the work reveals that the performance gain with closed nesting is significantly affected by the system's hot spots, which are likely to change over time in some application situations, necessitating the need for automated *dynamic* decomposition of transactions.

Acknowledgments

This work is supported in part by US National Science Foundation under grants CNS-1217385 and CNS-1116190.

References

[1] N. Shavit and D. Touitou, "Software transactional memory," ser. PODC, 1995.

[2] L. Dalessandro, M. F. Spear, and M. L. Scott, "NOrec: streamlining STM by abolishing ownership records," in *PPOPP*, 2010.

[3] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *Distributed Computing*. Springer, 2006, pp. 194–208.

[4] A. Turcu, B. Ravindran, and R. Palmieri, "Hyflow2: a high performance distributed transactional memory framework in scala," in *PPPJ*, 2013.

[5] A. Hassan, R. Palmieri, and B. Ravindran, "Remote invalidation: Optimizing the critical path of memory transactions," in *IPDPS*, 2014.

[6] M. Herlihy and Y. Sun, "Distributed transactional memory for metric-space networks," in *DISC*, 2005.

[7] D. Hendler, A. Naiman, S. Peluso, F. Quaglia, P. Romano, and A. Suissa, "Exploiting locality in lease-based replicated transactional memory via task migration," in *DISC*, 2013.

[8] J. Kim and Ravindran, "Scheduling transactions in replicated distributed software transactional memory," in *CCGRID '13*.

[9] J. Kim and B. Ravindran, "Scheduling closed-nested transactions in distributed transactional memory," in *IPDPS*, 2012.

[10] A. Dhoke, B. Ravindran, and B. Zhang, "On closed nesting and checkpointing in fault-tolerant distributed transactional memory," in *IPDPS*, 2013.

[11] E. Koskinen and M. Herlihy, "Checkpoints and continuations instead of nested transactions," in *SPAA*, 2008.

[12] "TPC-C benchmark: Transaction processing performance council. www.tpc.org."

[13] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC*, 2008.

[14] J. E. B. Moss and A. L. Hosking, "Nested TM: Model and architecture sketches," *Sci Comp Prog*, vol. 63, pp. 186–201, 2006.

[15] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood, "Supporting nested transactional memory in logtm," in *ASPLOS*, 2006.

[16] B. Zhang and B. Ravindran, "A quorum-based replication framework for distributed software transactional memory," ser. OPODIS, 2011.

[17] P. Bernstein and N. Goodman, "Multiversion concurrency controltheory and algorithms," *TODS*, 1983.

[18] D. Agrawal and A. El Abbadi, "The tree quorum protocol: An efficient approach for managing replicated data," in *VLDB '90*.

[19] P. di Sanzo, B. Ciciani, R. Palmieri, F. Quaglia, and P. Romano, "On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking," *Perform. Eval.*, vol. 69, no. 5, 2012.

[20] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*. IBM Corp., 2010.

[21] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *PPoPP*, 2005, pp. 48–60.

[22] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman, "Compiler and runtime support for efficient software transactional memory," in *PLDI*, 2006.

[23] T. Schmitt, N. Kammer, P. Schmidt, A. Weggerle, S. Gerhold, and P. Schulthess, "Rainbow os: A distributed stm for in-memory data clusters," in *MIPRO*, 2011.

[24] G. Golan-Gueta, N. Bronson, A. Aiken, G. Ramalingam, M. Sagiv, and E. Yahav, "Automatic fine-grain locking using shape properties," ser. OOPSLA '11.

[25] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential java code for concurrency via concurrent libraries," ser. ICSE '09.

[26] P. Joshi, M. Naik, C.-S. Park, and K. Sen, "CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs," ser. CAV '09.

[27] C. Gould, Z. Su, and P. Devanbu, "Jdbc checker: A static analysis tool for sql/jdbc applications," ser. ICSE '04.