Transactional Orc

Katherine E. Coons

May 7, 2008

Abstract

Transactions provide a language construct that programmers can use to specify that operations execute atomically, in isolation from all computation outside the transaction. Transactions permit optimistic concurrency for accesses to shared data, which may improve performance over locking mechanisms. Transactions also allow programmers to specify atomic regions in a composable way, which locks do not permit. Although transactions are theoretically capable of these performance and programmability improvements, researchers have found it difficult to exploit these benefits in practice via transactional memory. Most transactional memory implementations provide a single transaction management policy. Which policy is desirable, however, often differs with the memory location. Some objects are highly contended and may benefit from deferred updates, while others seldom suffer from contention and may benefit from in-place update, for instance. By implementing transactions in Orc, a language that separates computation from concurrency, we allow different transaction management policies to be used in a single program, and even within a single transaction. Although composability is one of the key programmability goals of transactions, most nesting solutions also suffer from high overheads, unnecessary restrictions, unintuitive nesting policies, or excessive aborts at runtime that may limit gains in programmability. Orc provides nested parallel transactions, and allows computations and data structures to decide how, and whether, they will support nested transactions.

1 Introduction

Transactions provide a simple mechanism for programmers to express that operations should be atomic and isolated from other computation. One way to ensure atomicity and isolation of operations within a transaction is to use transactional memory. In a transactional memory system, transactional accesses to shared memory are systematically modified to record any state modifications such that they can be rolled back if the transaction fails to execute in isolation.

Speculatively executing transactions concurrently can provide performance improvements over locks, yet the policies that a transactional memory system chooses affect the ability of the transactional memory system to exploit this concurrency effectively. A transactional memory system usually dictates a single policy that all memory accesses use when modifying memory. This policy dictates where modified state is stored, when conflicts are detected, how transactions are nested, and how conflicting non-transactional accesses are handled. Highly contended data may prefer a deferred update system that allows fast aborts at the expense of slower commits, whereas infrequently contended data may prefer an in-place update system that makes commits fast at the expense of slower aborts.

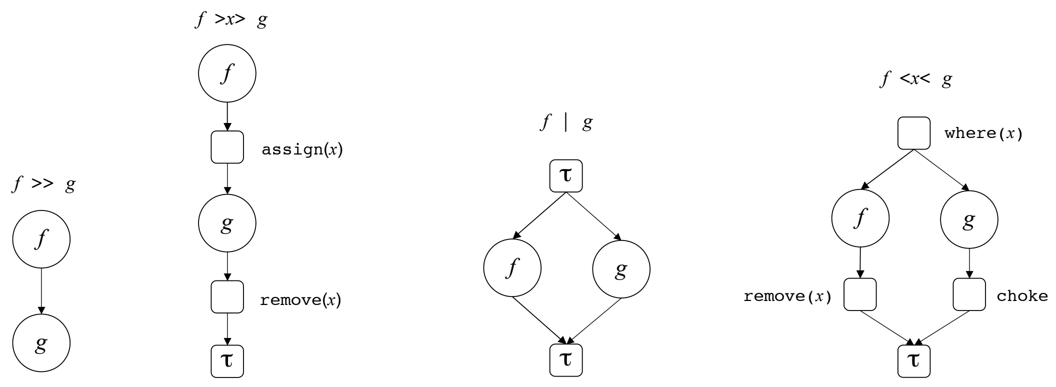
Rather than supplying a single transaction manager that handles all transactional data, we propose a system in which the transaction manager's characteristics may vary with the data being manipulated. We use Orc, a language that separates computation and concurrency, to separate the management of transactional data from the orchestration of the transaction. This separation allows each computation, object, or data structure to choose its own transaction management policy. As a result, different data structures, or different instances of the same data structure, can select the transaction management policy that best suits their expected usage scenarios. By allowing each data structure to specify its own transaction management policy for its data, transactions in Orc also provide the opportunity for data structures, if desired, to abstract away from transactional memory in favor of actions and compensating actions.

In addition to allowing concurrent execution of atomic regions, transactions provide the programmer with the ability to compose atomic regions. This composition is difficult with locks, particularly when libraries make the details of nested code invisible. Nested transactions in transactional memory implementations often suffer from various problems, however, such as high overhead, unintuitive semantics, unnecessary restritions, or excessive aborted transactions. Nested transactions in Orc differ from nested transactions in traditional transactional memory systems because a nested child transaction may conflict with other, concurrent nested child transactions, and it may also conflict with its parent transaction.

Orc provides nested parallel transactions, similar to those provided in [2], and parallel threads within a single transaction. Unlike the Cilk implementation of parallel nested transactions in [2], however, Orc provides a setting in which parallelism within a transaction, and among nested transactions, is relatively simple to implement. Also, because Orc is a language that only orchestrates other computations and does not perform any computations itself, an Orc program can leverage legacy code. An Orc program simply provides a mechanism to specify concurrency in that code at any desired granularity. By adding transactions to Orc, an Orc program can specify not only the concurrency among those operations, but also any atomicity and isolation requirements among those operations.

2 Background

This section provides background information necessary to understand how Orc compiles and executes transactions, as well as how transactions in Orc relate to traditional transactional memory systems.



(a) Sequential (b) Sequential assignment (c) Parallel symmetric (d) Parallel asymmetric

Figure 1: DAG for each Orc operator.

2.1 Orc

Orc provides a structured way to express concurrent and distributed programming [12]. Orc cleanly separates concurrency from computation by handling concurrency within Orc, but relying on primitive *sites* to perform all sequential computation and data manipulation. A site may receive parameters as input, and it will provide at most one response. A site may be a function, a method call for an object, a monitor procedure, or a webservice, and it can be implemented in any language.

Orc provides three operators to orchestrate computation between sites. In the following descriptions f and g represent *expressions*, which can be any combination of the three operators and site calls. The simplest expression is a single site call. The three operators that Orc provides are:

- Symmetric parallel composition: $f \quad | \quad g$
  Initiate f and g in parallel, and publish all values published by f or g.
- Sequential: $f \quad >x> \quad g$
  Execute f. For all values x published by f, do g.
- Asymmetric parallel composition: $g \quad <x< \quad f$
  Initiate f and g in parallel. When f publishes its first value, bind that value to x for use by g, and terminate evaluation of f.

For further discussion of the Orc operators, see [12]. The most important concept for this discussion is that Orc operators explicitly expression concurrency. In addition, all computation and data manipulation is performed not via Orc instructions, but via site calls, and the code implemented at a site can be implemented in any language.

The Orc compiler builds a DAG to represent each Orc expression. Figure 1 shows the DAG representing each Orc operator, where f and g are Orc expressions. To evaluate the result of an expression, the Orc runtime places a *token* at the expression's root node. Each node in the graph executes instructions that may include calling a site, creating duplicate tokens to initiate concurrent operations, or moving the token to a different node.

The Orc language is extremely simple, and provides a powerful mechanism to express concurrency in a structured way. It does not, however, provide any mechanism for atomicity. For example, consider a work queue that contains an ordered list of work items that must be distributed to workers in priority order. Many workers monitor the work queue, but only some subset of the workers are qualified to handle each work item. Workers monitor the queue waiting for a task to appear that they are qualified to handle, and when such a task appears at the head of teh queue, a qualified worker removes that task and begins to work on it.

Without atomicity, it is impossible for a worker to ensure that the task observed at the head of the work queue is the same task removed from the head of the work queue, as some other worker may have removed it. If a different task is removed from the queue, the worker may not be qualified to handle it, and the work queue can no longer guarantee that the work items will be distributed to workers in their priority order.

If the work queue handles transactions, however, then workers can perform atomic operations that observe the item at the head of the queue and remove that item only if it is one on which they are qualified to work. Becacuse the read of the head of the work queue and the removal of the head of the work queue are performed atomically, the worker can guarantee that no other workers will interfere before the transaction is complete.

2.2 Transactional Memory

Transactional memory allows programmers to specify regions of code that should execute atomically, in isolation from all other code. Typically, a transactional memory system tracks the memory addresses accessed within transactions and detects conflicting accesses. Two accesses conflict if they access the same memory address and at least one of the two accesses is a write.

Transactional memory support can be implemented in hardware with minimal software support, e.g. [13, 9, 17], in software without hardware support, e.g. [1, 21, 19], or using a combination of hardware and software techniques, e.g. [11, 22, 20]. A transactional memory system can be characterized by which choices it makes with respect to several policies including version management, conflict detection, isolation, and nesting. Version management describes the system's mechanism for storing multiple copies of the memory modified by a transaction. Conflict detection determines when the system detects that a conflict between transactions has occurred. The isolation policy dictates to what extent transactions are isolated from non-transactional accesses. Finally, the nesting policy determines when and whether the state modified by a nested transaction is made visible to the parent transaction, and when the nested transaction's modified state is committed to globally visible state.

Although existing transactional memory systems make a variety of choices with respect to version management, conflict detection, isolation, and nesting, almost all transactional memory systems use a single policy for the entire memory system. Some systems allow the user to vary the policy from one execution to the next, but for a given execution the entire memory system uses a single policy. In contrast, transactions in Orc allow each site to choose the policies it will use. Because tasks like contention management and conflict detection are distributed and implemented separately by each site, a single Orc program may call sites with a variety of different policies, and multiple policies may even be invoked within a single transaction.

The next section discusses the semantics for transactions in Orc and describes the interface that the Orc engine uses to communicate with sites. Section 4 describes how Orc was modified to handle transactions, and Section 5 describes a transactional implementation for one particular site, a channel. As each transactional policy is discussed in Section 5, additional background information regarding that policy will be provided.

3 Transactional Semantics for Orc

A transaction in Orc could be described by many possible semantics. We chose a semantics that focuses on the intuitive meaning of *atomic*, built upon the assumption that an expression that is within an `atomic` region should execute exactly as it would if it were not atomic, except that all computation performed must be isolated. In particular, if f is an Orc expression,

`atomic`(f) publishes whatever f would have published if it were executed in isolation

- If f is silent, `atomic`(f) is also silent
- If `atomic`(f) publishes, all values published were computed in isolation from anything outside of f
- If any computation in f conflicts with an access outside of f, `atomic`(f) aborts and retries

Because Orc separates computation from the orchestration of concurrency, transactions in Orc can similarly be divided into computation and orchestration. Sites are responsible for managing the data modified by the transaction, while the Orc engine is responsible for orchestrating the transaction. This separation of concerns means that the Orc engine is not responsible for any policies with respect to contention management, version management, nesting, or concurrent non-transactional accesses.

Each site can select its own policies for managing any data that it stores. The Orc engine and transactional sites communicate with one another via an interface that each site must implement. The Orc engine uses the methods in this interface to orchestrate transactions, and the sites implement the methods in this interface to handle transactions for any data that they manage. Every site in Orc implements the following methods:

- `void` `callSite`(*args*, *token*)
- `void` `callSiteTx`(*args*, *token*)
- `void` `validateTx`(*tx*)
- `void` `commitTx`(*tx*)
- `void` `rollbackTx`(*tx*)
- `boolean` `handlesTx`( )
- `boolean` `handlesNestedTx`( )

The `callSite` method must be implemented by all sites to implement the site's behavior, even if the site does not support transactions. The default behavior for the `callSiteTx`, `validateTx`, `commitTx`, and `rollbackTx` methods is to throw an error indicating that the site does not handle transactions. The `handlesTx` and the `handlesNestedTx` methods both return false by default. The Orc engine will throw an error if a `callSiteTx` call occurs within a transactional context for a site that does not handle transactions. The engine will also throw an error if a `callSiteTx` call occurs in a nested transactional context for a site that does not handle nested transactions.

A transactional site overrides these default behaviors with behaviors that support transactions for the site. The specific behavior of each method will depend on the site's version management, conflict detection, isolation, and nesting policies. If a site does not maintain any state, then it may not need to do anything when these methods are called. For instance, the sites that perform arithmetic operations, comparisons, and conditional operations do not need to commit, rollback, or validate anything. Their `callSiteTx` method simply calls the `callSite` method.

A site that maintains state, however, must keep track of state changes made by transactions, detect conflicts, and mediate conflicts between transactional and non-transactional site calls. The next section describes the Orc engine's role in implementing these transactional semantics in greater detail, and the following section describes the site's role, using transactional channels as an example.

4 Transactions in Orc

This section describes how the compiler constructs the DAG for an Orc program that contains transactions. In addition, it describes ways in which normal graph traversal is modified for tokens involved in a transaction, and the process the Orc engine uses to validate, commit, and abort all transactions that complete.

4.1 DAG Construction

The Orc compiler builds a directed acyclic graph (DAG) with a single root node and a single sink node for every expression in an Orc program. Each expression's

DAG is constructed recursively, preserving a single root node and a single sink node for the DAG at each step. We modified the compiler to insert the construct shown in Figure 2(a) when an atomic expression is encountered.

The nodes in Figure 2 marked `enterTx` and `leaveTx` provide an opportunity for the Orc engine to manage transactions. Each token that passes through the `enterTx` node will cause the Orc engine to generate a new transaction. Thus, the following Orc expression will produce two transactions:

(f | g) >> `atomic` (h)

The Orc engine will create two tokens to execute f and g concurrently, and each token will pass through the `enterTx` node for the atomic expression h, generating a new transaction. Thus, two instances of the atomic expression h will execute at runtime.

The circular node f in Figure 2(a) represents an atomic expression, which can consist of any Orc expression including both parallel expressions and other atomic expressions. For example, Figure 2(b) shows a DAG in which f is a parallel expression containing concurrent expressions g and h. In this program, operations performed in expressions g and h can execute concurrently, yet they must be atomic and isolated with respect to any computations outside of the atomic expression f.

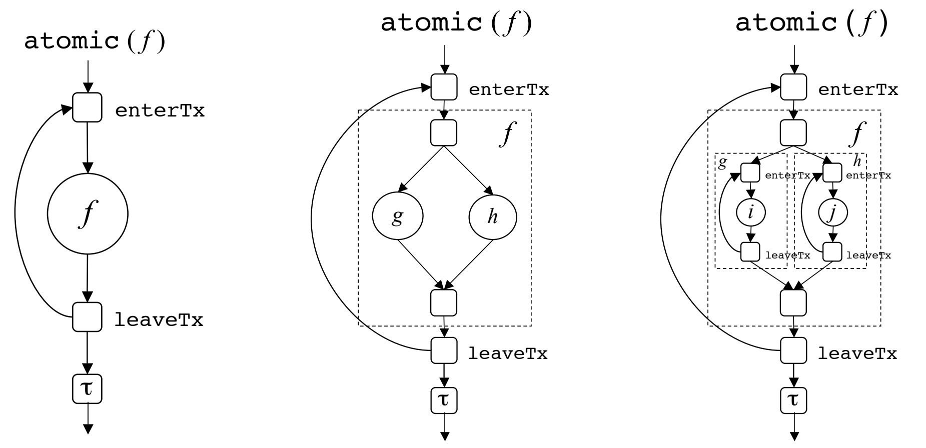
Figure 2(c) shows that the expressions within f may be nested atomic operations. In Figure 2(c) the atomic expression f consists of two parallel expressions, g and h, just as in Figure 2(b). In this case, g and h are atomic expressions i, and j, respectively. The operations in i must be atomic with respect to the operations in j, and all computation within f must be atomic with respect to everything outside f.

Figure 2(c) shows that nested transactions in Orc have a tree-like structure in which a parent transaction can have any number of child transactions. These child transactions must all be atomic with respect to one another, yet they can execute concurrently provided that isolation is maintained. In contrast, traditional nested transactional memory regions that share a common parent cannot execute concurrently, except those in transactional Cilk [2].

Each Orc expression may be arbitrarily large and consist of any combinations of parallel expressions, sequential expressions, and where expressions. Every expression is guaranteed to have a single root node and a single sink node, making it easy to place the `enterTx` node and the `leaveTx` node that initiate and complete the transaction. The Orc compiler places the `enterTx` node immediately before the root node of the atomic expression, and it places the `leaveTx` node immediately after the expression's sink node.

4.2 The `enterTx` node

When processing the `enterTx` node, the Orc engine immediately creates a copy of the token exactly as it was at entry. This copy of the entry token allows the engine to initiate an identical retry transaction if the current attempt fails. Each token



(a) Atomic DAG (b) Atomic Parallel DAG (c) Nested Atomic DAG

Figure 2: Atomic expression DAGs created by the Orc compiler.

contains a small amount of state that may change as the transaction executes, so this copy is necessary to ensure the correct state for retry.

In addition to creating a copy of the entry token, the `enterTx` node generates a new transaction object. The transaction object contains the following information about the transaction:

- A pointer to the entry node copy, described above
- A *token set* containing all tokens involved in the transaction
- A *site set* containing all sites called by tokens in the token set
- A unique identifier for the transaction
- A copy of the `leaveTx` node, so a transaction may be aborted at any time
- The state of the transaction, which is currently an enumeration consisting of `active`, `doomed`, `validating`, `rollingback`, `aborted`, `committing`, and `committed`
- The enclosing parent transaction, or null for an outermost transaction

When a token is processing the `enterTx` node, it is the only token in the transaction's token set. The site set is empty, as no sites have been called yet. The transaction's state is initialized to `active`.

To handle nested transactions, the `enterTx` node sets the new transaction object's *parent* field to the token's transaction object at the time it enters the transaction. The transaction object for a non-transactional token is null, thus the outermost transaction will always have a null parent. If a token arrives at the `enterTx` node with a non-null transaction object, then it is entering a nested transaction. The new transaction's *parent* field is set to the transaction object at the time it began processing the `enterTx` node. If multiple tokens from the same transaction enter nested transactions, as in Figure 2(c), then both nested transactions have the same parent transaction, forming a tree of transaction objects.

4.3 Graph Traversal

As a token leaves the `enterTx` node and traverses the graph for the atomic expression, it may create copies of itself to enable concurrent execution. Anytime a token is copied, its pointer to the transaction object is propogated to the copy, and the new copy token is added to the transaction object's token set. Thus, the token list in the transaction object will always be up-to-date, and every token involved in the transaction is guaranteed to contain a pointer to the transaction object.

When a token involved in a transaction makes a site call, it calls the site's `callSiteTx` method, rather than the normal `callSite` method. Prior to making this call, the engine checks the `handlesTx` method for the site to ensure that the site is prepared to handle transactions. If this call returns false, the Orc engine throws an error and does not call the site. If `handlesTx` returns true, the engine makes the site call and adds the site to the transaction's site set.

Section 5 provides details about the transaction's execution at the site. From the Orc engine's point of view, the site is entirely responsible for managing the atomicity and isolation of any data accessed within the site call. The site has the option to *doom* a transaction at any time if its atomicity or isolation is violated. To doom a transaction, the site calls the transaction object's `doom` method, which iterates through all tokens in the token set, killing each one. Then, the `doom` method sets the transaction's state to `doomed`.

Tokens die for various reasons while traversing the graph. For instance, a token that enters a `silent` node dies. A token involved in an unsatisfied `if` node will die, as well. If a transaction is doomed by the transaction object's `doom` method, all tokens involved in the transaction die. For the `leaveTx` node to work correctly, the engine must guarantee that a token from the transaction's token set will pass through the `leaveTx` node at a point when all tokens are finished with the transaction. To guarantee this behavior, each token contains a boolean called $txFinished$ that is initialized to false, and that is set to true when either of the following two events occurs:

- The token enters the `leaveTx` node
- The token dies

If the last unfinished token in a transaction dies without passing through the `leaveTx` node, the Orc engine may hang because live tokens waiting to escape the `leaveTx` node upon commit will never be released. Thus, prior to killing a transactional token, the `die` method iterates through all tokens involved in the transaction searching for unfinished tokens. If no unfinished tokens exist, the `die` routine does not kill the token, but instead activates the token at the `leaveTx` node so that it can initiate validation. In addition, the `die` routine marks that token by setting a `doomed` boolean, embedded within the token, to true. This `doomed` value indicates that the `leaveTx` node should not activate the token upon commit, but kill it. The specific actions performed at the `leaveTx` node will be discussed further in Section 4.4.

Nested transactions do not require any special treatment as tokens traverse the graph. The tokens involved in nested transactions are not included in the parent transaction's token set until the nested transaction is committed. The only token associated with an uncommitted nested transaction that is part of the parent transaction's token set is the entry token copy. Because this token is alive and its *txFinished* value is false, the parent transaction will not be allowed to commit until the nested child transaction has committed.

4.4 The `leaveTx` node

When a token processes the `leaveTx` node, the Orc engine immediately sets the token's *txFinished* value to true. After marking the token finished with the transaction, the `leaveTx` node iterates through the transaction's token set searching for tokens whose *txFinished* values are false. If any such token is found, nothing further occurs and the token does not proceed beyond the `leaveTx` node, remaining inactive.

If the *txFinished* value is true for every token in the transaction's token set, then the transaction is finished, and the `leaveTx` node begins validating the transaction, settings its state to `validating`. To validate the transaction, the `leaveTx` node first checks whether the transaction is in the `doomed` state. If the transaction is `doomed`, it fails without further validation. Sections 5.2 and 5.4 describe the circumstances under which a transaction will enter the `doomed` state. If the transaction is still in the `active` state, the node iterates through the transaction's site set, calling each site's `validateTx` method. If any site returns false, validation fails.

If validation fails, the transaction enters the `rollingback` state, and the `leaveTx` node calls every site's `rollbackTx` method. After every site has been rolled back, the `leaveTx` node kills all tokens in the token set, then activates the entry token copy at the `enterTx` node, retrying the transaction.

If all sites return true, the transaction enters the `committing` state and the `leaveTx` node calls every site's `commitTx` method. After all sites have committed, the `leaveTx` node iterates through all tokens in the token set. If a token is `doomed`, it is killed. If a token is alive and is not `doomed`, then it is activated at the `leaveTx` node's successor. Thus, the atomic section guarantees that all values published by the transaction were computed in an atomic, isolated fashion at the transaction's commit point, which was the point at which the transaction entered the `validating` state. No guarantees exist about the order in which the values published by the transaction are published, however.

To handle nested transactions, the `leaveTx` node checks the state of the parent transaction prior to retrying an aborted nested transaction. If the parent transaction is in the `active` state, the nested transaction retries as usual. If the parent transaction is in the `doomed` state, however, the nested transaction does not retry. Instead, the `leaveTx` node kills all tokens involved in the nested transaction, then kills the nested transaction's entry token copy. By killing the entry token copy, the nested transaction notifies the parent transaction that it is finished attempting to execute the transaction. The entry token copy is part of the parent transaction, and if it is the last token in the parent transaction to die, it will become doomed and intitiate validation for the parent transaction, as described in Section 4.3.

While the Orc engine is responsible for orchestrating transactions by initiating validation, commit, and rollback, and keeping track of the tokens and sites involved, the sites in Orc are responsible for enforcing atomicity and isolation for any data managed by the site. The next section discusses the implementation of transactions for one particular site, a channel.

5 Transactional Channels

If an Orc site supports transactions, it must guarantee that any site calls performed by a given transaction are atomic and isolated with respect to any calls to that site made by other transactions. The site must have a mechanism to undo any state changes performed by a transaction that has not yet committed. The site can choose whether to handle concurrent non-transactional accesses to shared data, and it must either implement a nesting policy or allow the `handlesNestedTx` method to return false. This section describes an implementation of transactions for one particular site, a transactional channel.

Because the site is responsible for managing transactional data, different sites may choose to manage their data in different ways. If a site does not maintain any state, it may not need to implement any additional code to support transactions at all. The site that performs integer addition, for instance, does not need to do anything special for a transactional operation.

Sites that do require special treatment of transactions can use any mechanism desired to achieve this goal - transactional memory is not a requirement, and may be unnecessary if the actions peformed by the site are limited or do not lend themselves to such a model. The transactional channel described here could be supported by mechanisms from transactional memory, with a modified JVM. Instead, we chose to implement a more abstract form of transactions that uses compensating actions to demonstrate the feasibility of this approach.

5.1 Channel Types

We implemented three types of channels that interact differently in a transactional system. The *non-blocking channel* maintains a *senderQueue* of sent values and resumes both senders and receivers immediately. If a receiver arrives and finds the *senderQueue* empty, the receiver returns with a special Orc value called `NIL`, indicating that no value was present.

The second channel implementation, the *receiver-blocking channel*, treats senders in exactly the same way as the non-blocking channel does. If a receiver arrives at the queue and finds that the queue is empty, however, the receiver will add itself to a *receiverQueue* and block. The receiver will only be released after a sender arrives and finds it at the head of the receiverQueue, at which point the sender and receiver both resume.

The final channel implementation, the *blocking channel*, blocks both senders and receivers until a matching operation arrives. This channel maintains both a *senderQueue* and a *receiverQueue*, and maintains the invariant that at least one of the two queues will be empty at all times. If a sender arrives and the *receiverQueue* is empty, it will add itself to the *senderQueue* and block. If it finds receivers waiting on the *receiverQueue*, it will pop the head of the *receiverQueue* and both the sender and the receiver will resume. A receiver will behave in a similar manner, either blocking until a sender arrives, or popping the head of the *senderQueue* and resuming both sender and receiver. This channel is particularly interesting because neither thread in a sender/receiver pair can resume without either observing another thread's state, or its state being observed by another thread.

In addition to the `get` and `put` operations provided by Orc's built-in channel class, we added a `peek` method for each type of channel that peeks at the value that a `get` operation would get, but does not actually remove the value from the head of the *senderQueue*, or resume the sender for the blocking channel. The peek operation was useful because it provided a read operation that observes but does not actually modify shared state, unlike the `put` and `get` operations, which both modify shared state. The remainder of this section describes the policies we chose for the transactional channel, as well as interesting observations about blocking channels when used in transactions.

5.2 Conflict Detection

A conflict detection policy is *eager* if conflicts are detected immediately, and *lazy* if conflicts are not detected until validation [13]. The Orc engine can support both policies. If a single transaction includes multiple site calls, those calls can use different conflict detection policies. The Orc engine supports eager conflict detection by providing sites with a `doom` method in the transaction object that allows the site to immediately terminate the transaction if a conflict is detected eagerly. The Orc engine also supports lazy conflict detection because it does not commit a transaction until it calls the `validateTx` method for every site called by the transaction. Thus, sites that detect conflicts lazily will detect the conflict when their `validateTx` method is called.

We used an eager conflict detection scheme for the transactional channel to prevent a doomed transaction from performing useless work. We implemented three conflict detection managers each using a different degree of optimism. The least optimistic channel monitors ownership at the channel granularity and does not differentiate between read and write operations. Thus, only a single transaction can have ownership of the entire channel at any time. A slightly more optimistic conflict detection manager differentiates between different data items within the channel, rather than treating the entire channel as a single piece of data. Thus, `put` and `get` operations from different transactions can execute concurrently, provided that the `get` operation does not observe the value provided by the `put` operation. Finally, the most optimistic conflict detection policy differentiates between read and write operations, and allows multiple transactions to peek at the channel's head simultaneously, yet only a single transaction can perform a `put` or `get` operation at a time.

When the conflict detection manager determines that a conflict is about to occur, it eagerly aborts one of the transactions involved. Which transaction is aborted and when a conflict is detected varies with the management policy discussed above. To eagerly abort the transaction the channel first calls the transaction object's `doom` method, which kills all tokens involved in the transaction, sending the last one to the `leaveTx` node. Then, the channel changes the transaction's state to `doomed`.

5.3 Version Management

A version management policy describes how a transactional system handles both new and old versions of data that has been modified [13]. The new version of the data is the version that should be visible if the transaction commits, and the old version is the version that should be visible if the transaction rolls back. Only one of these two versions of the data can be stored in visible state, the other must be maintained "on the side". If a system stores speculative transactional updates in visible state and reverts them to their original values on rollback, then it uses *in-place update*. If a system stores speculative values on the side and only updates visible state upon commit, then it uses *deferred update*.

Because a channel has a very simple set of operations that occur at very limited locations - at either end of a queue - we chose to implement a more abstract mechanism for version management than a traditional transactional memory system would provide. A transactional `put` or `get` operation will record the action it performed in an undo log that is used during rollback to revert the channel to its state prior to the beginning of the transaction. Rather than recording the address of the memory it modified, the `put` or `get` operation records an abstract notion of what it did - removing or adding a value to either the sender or the receiver queue - and the channel provides a compensating action to execute during rollback.

On rollback, the channel performs a compensating action for each operation in the undo log. One potential disadvantage to this approach is that, if the channel operates on only a single memory location, by performing alternating `get` and `put` operations, for instance, then rather than reverting a single memory location, the system must undo each operation individually. To account for this scenario, however, it is relativley simple to modify the undo log to eliminate operations that undo each other. This abstract transactional memory implementation is simple to implement, and does not require any changes to the compiler or JVM.

A channel is an example of a data structure that could benefit from various combinations of version management and conflict detection policies depending on the scenarios in which the channel will be used. For instance, a channel that is not very highly contended, but whose quick response is imperative, may choose an in-place update scheme with eager conflict detection. In-place updates ensure that commits will be fast - the committed values are already in visible state - which is desirable if speed matters and conflicts are rare.

In contrast, a channel that is highly contended may perform better with a deferred update system because abort is much faster with deferred update - the modified values can simply be thrown away. If a channel could be used in either context, it might be desireable to create channels of different types, and even use them within the same transaction if desired. Orc provides a mechanism with which each instantiation of a channel can choose its own version management and conflict detection policies.

5.4 Isolation Policy

An isolation policy dictates how the transaction system responds when a concurrent non-transactional access to transactional data occurs. Blundell et al. deconstruct the subtleties of the interaction between transactional and non-transactional code [5], and their notions of "strong atomicity" and "weak atomicity" will be referred to here as *strong isolation* and *weak isolation* as others have done [11], as they refer more closely to a transaction's isolation than to its atomicity. Because most software transactional memory systems do not instrument non-transactional accesses, it is impossible for them to detect conflicts between transactional and non-transactional accesses.

A system that provides *strong isolation* guarantees that a transaction will always be atomic with respect to non-transactional accesses. Unfortunately, strong atomicity can be very costly in terms of performance for non-transactional code, requiring read barriers and a write barrier that contains an atomic operation [21]. Weak isolation, however, may lead to unintuitive results when transactional and non-transactional accesses conflict, as demonstrated in [21]. As a result, it may be desirable to choose a different isolation policy for different sites depending on their expected usage.

For the transactional channel we implemented strong isolation because we expect transactional and non-transactional accesses to conflict frequently. Section 5.6 may provide justification for this expectation. A non-transactional access always aborts a conflicting uncommitted transactional access because Orc does not currently support any mechanism for the non-transactional access to wait until the transaction completes. A non-transactional access requests permission to perform an operation prior to performing the operation, and if permission is denied then it aborts the transaction preventing it from performing the action.

To abort the transaction, the channel first calls the transaction object's `doom` method, which kills all tokens involved in the transaction, sending the last one to the `leaveTx` node to initiate validation and rollback. The transaction also pre-

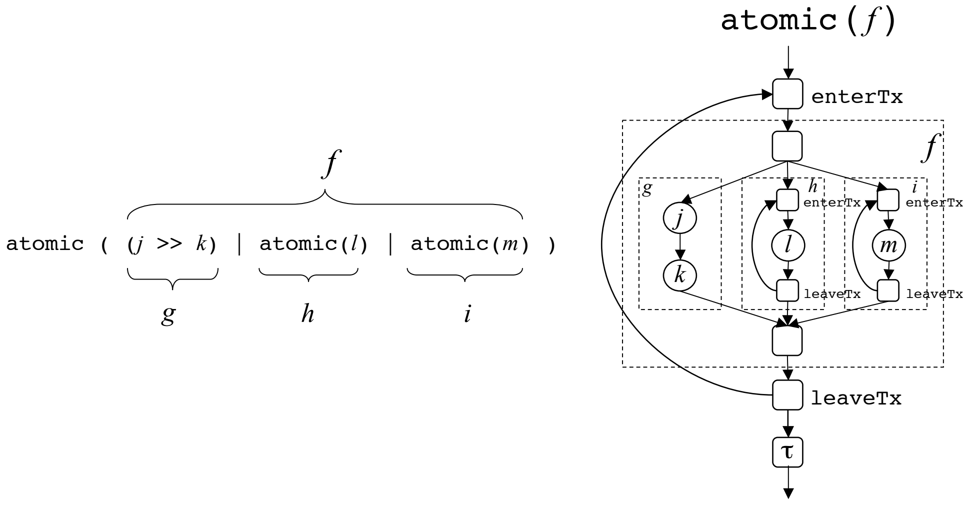


Figure 3: A small Orc program that uses nested parallelism. Atomic expression l must be atomic with respect to sibling atomic expression m, and also with respect to expressions j and k, which belong to l's parent transaction, atomic expression f.

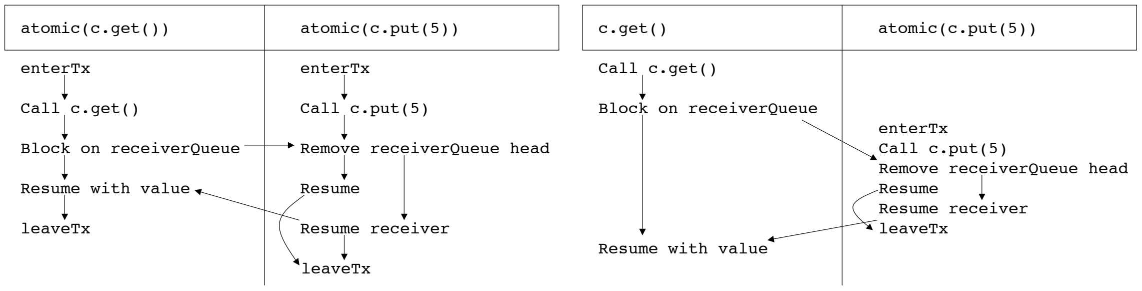
emptively rolls back the transaction at the channel. The non-transactional access cannot wait for the Orc engine to rollback the transaction because it *must* complete successfully. Because the transaction has already been doomed and is thus guaranteed not to validate, this pre-emptive rollback is acceptable. The compensating actions are destroyed after rollback such that, when the Orc engine asks the channel to rollback the transaction again after validation, no further compensating actions will be performed.

5.5 Nesting Policy

Various semantics for nesting transactions have been proposed in the transactional memory literature. Flat-nested transactions essentially inline the child transaction directly into the parent transaction [3, 23, 8, 18, 7, 17]. Flat nesting does not work correctly with transactions in Orc because nested transactions may be concurrent, and must thus be isolated from each other. As a result, merging the transactions into the read and write set of the parent transaction will eliminate isolation between multiple concurrent nested transactions.

Closed nested transactions allow a nested transaction's read and write set to be tracked separately from its ancestors [3, 10, 15, 20, 24, 1, 19, 6]. The nested transaction can access state generated by its ancestors, but it can be independently rolled back and retried. When a closed-nested transaction commits, its modified state merges with that of its parent transaction, but no speculative state becomes visible outside the parent transaction. Most closed nested transactional memory implementations use a stack of logs. To support closed-nested transactions in Orc, a tree, rather than a stack, is required to handle concurrent child transactions.

In an open nested transaction, the speculative state inside the nested transac-



(a) Dependences between transactions (b) Transaction/non-transaction dependences

Figure 4: Comparison of dependences between two transactions, and between a transaction and a non-transaction performing matching `get` and `put` operations on a blocking channel.

tion commits directly to shared state, and the parent transaction may execute a *compensating action* upon abort to undo changes made by the child transaction [10, 23, 16, 25, 11, 14, 21, 15]. Although open nested transactions are often viewed as dangerous because they can have unintuitive effects, they may be useful in cases where a highly contended value is updated in a way that can be undone even after having been modified by other threads. For example, if a nested transaction increments a highly contended counter that is used only after all transactions finish their computation, an open-nested implementation will be correct as long as a compensating decrement occurs when the transaction aborts [15].

Although implementing open nesting in a manner similar to [16] should be possible in Orc, we suspect that the feasibility of nested parallelism and concurrently executing nested transactions may alleviate the most common problem with closed nesting: a long-running parent transaction that cannot commit due to a child transaction that accesses highly-contested data. By allowing concurrency within a transaction, we hope that long-running transactions may become less frequent.

To support closed nested transactions, the channel site allows a transaction to obtain permission to perform a `get` or a `put` operation if the current owner of `get`/`put` permission is its ancestor. The ancestor's log of state changes is set aside and a new log is created for the child transaction so that it can be rolled back and retried independently of its parent.

Unlike a traditional transactional memory system, the parent may be executing actions concurrently with the child transaction, so the child transaction must ensure isolation from *all* transactions including both siblings and ancestors. Figure 3 shows an example of a small Orc program, with its corresponding DAG, in which parallel nested transactions h and i execute concurrently both with each other, and with expressions j and k that belong to the parent transaction, f.

5.6 Blocking Channels in Transactions

The blocking channel was particularly interesting when used in transactions because it fundamentally requires an exchange of intermediate state. We quickly observed that two transactions *cannot* communicate with one another via a blocking channel.

Figure 4 shows the operations required for two transactions to communicate with one another via a blocking channel (Figure 4(a)), and for a transaction and a non-transactional access to communicate with one another (Figure 4(b)).

In Figure 4(a), a cyclic dependence exists between the `put` and the `get` operation that makes it impossible for the two transactions to be serialized with respect to one another without violating a dependence. Figure 4(b) shows that a non-transactional access can exchange information with a transactional access because the transaction can be serialized with respect to the non-transactional accesses without violating any dependences.

To make the scenario in Figure 4(b) possible, a non-transactional access that communicates via a blocking channel with a transactional access cannot be resumed until the transaction has been committed. Thus, the `commitTx` method for the blocking and receiver-blocking channels releases all non-transactional accesses that must be released by the transaction. This same strategy does not work when two transactions communicate with one another because the transaction waiting to be released *cannot commit* until the other transaction has released it, thus the circular dependence remains.

One interesting side effect of delaying the release of non-transactional accesses until commit occurs is that sequential accesses cannot ever exchange values with accesses that are in the same transaction. For instance, if a non-transactional access performs two sequential `get` operations, and a transaction performs two parallel, atomic `put` operations, it is impossible for both `get` operations and both `put` operations to match up:

$tx:$ $\texttt{atomic}(c.\texttt{put}(1) \mid c.\texttt{put}(2))$ $\qquad$ $non-tx:$ $\text{c}.\texttt{get}() >> c.\texttt{get}()$

The first `get` operation cannot return until the transaction commits, thus the second `get` operation will not be initiated until the transaction is finished, and cannot get the other value put by the transaction.

Another interesting result of the dependences on the blocking channel is that, if both a `get` and a `put` operation occur within the same transaction, they *must* communicate with one another, and they will do so only on an empty channel. Thus, the following code would essentially privatize the shared channel for the explicit use of the transaction:

$tx:$ $\texttt{atomic}(c.\texttt{put}(1) \mid c.\texttt{get}())$

The transaction does not have any active power to privatize the channel, however; it simply waits for the channel to become empty. If the channel never becomes empty, the transaction will never commit or abort. The reason that the transactional `get` and `put` must communicate with one another is that the operations within a transaction, in order to communicate with non-transactional accesses, must operate on non-transactional accesses that could have already been initiated prior to the start of the transaction, as shown in Figure 4(b). With a blocking channel, however, it is impossible for both a `get` and a `put` operation to be initiated prior to the start of a transaction and yet neither to have completed, because this would break the invariant that either the *senderQueue* or the *receiverQueue* must be empty at all times. At least one of the two operations must have exchanged information with another access, or the two operations may have exchanged information with each other, but they cannot both be waiting when the transaction initiates.

Because atomic regions in Orc can contain concurrent operations within them, it is possible for communication to occur across a blocking channel within an atomic region. This same scenario, in which the atomic operations gain private access to the channel and communicate with one another, would be impossible in a traditional transactional memory system.

6 Related work

Transactions in Orc bear the most similarity to transactions in Cilk [4], as described in [2]. Much like Orc, Cilk provides simple language constructs to express parallelism. Cilk targets fork/join parallelism, and includes a runtime system that uses a work-stealing scheduler to run multi-threaded programs. Adding transactions to Cilk leads to nested parallelism within transactions as well as nested parallel transactions, much like in Orc.

Unlike Orc, however, Cilk does not separate computation from concurrency, and the XCilk transactional system provides a single transaction management policy that uses eager conflict detection, in-place updates, strong isolation, and closed nesting for all data. Agrawal et al. provide one solution to the problems that arise when conflict detection is generalized to concurrent nested transactions. The XCilk runtime creates an online *computation tree* that models nested parallel transactions. We show that other solutions to this problem are possible, and that separating computation from concurrency orchestration allows different sites to use different policies if desired.

Most other transactional memory systems operate under the assumption that the code within a transaction will be serial, and only a single nested transaction may execute at a time. Section 5 provides the appropriate related work at the start of each policy section.

7 Conclusions

By separating computation and concurrency, Orc provides an interesting context to explore the possibility of specifying transactional policies that vary with the data being modified. Orc separates the orchestration of transactions from the management of transactional data, and allows primitive sites to ensure atomicity and isolation for their data as they see fit. Orc does not include a single monolithic transaction manager that handles conflict detection and version management for all memory accesses. Instead, each site determines whether it wishes to support transactions at all, whether it wishes to allow nested transactions, to what extent it prioritizes non-transactional code, and which conflict detection and version management policies are most appropriate for the computations it performs. Because transactions in Orc are not directly tied to transactional memory, it is also possible for sites to specify their rollback actions in more abstract ways, such as providing compensating actions.

Although Orc is a programming language that is unfamiliar to most, it does allow the programmer to make use of legacy code, as there are no restrictions on how sites are implemented. Orc is particularly useful for distributed computation, where both sites and communication may fail. To fully support transactions in this context, future work will include defining a semantics with which timers can be incorporated into transactions to allow timeout, and other strategies to handle unreliable resources.

References

[1] Adl-Tabatabai, A.-R., Lewis, B. T., Menon, V., Murphy, B. R., Saha, B., and Shpeisman, T. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (New York, 2006), Association of Computing Machinery, pp. 26–37.

[2] Agrawal, K., Fineman, J. T., and Sukha, J. Nested parallelism in transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (New York, 2008), Association of Computing Machinery, pp. 163–174.

[3] Agrawal, K., Leiserson, C. E., and Sukha, J. Memory models for open-nested transactions. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness* (New York, 2006), Association of Computing Machinery, pp. 70–81.

[4] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, 1995), Association of Computing Machinery, pp. 207–216.

[5] Blundell, C., Lewis, E. C., and Martin, M. M. K. Deconstructing transactional semantics: The subtleties of atomicity. In *Proceedings of the Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)* (2005).

[6] Brevnov, E., Dolgov, Y., Kuznetsov, B., Yershov, D., Shakin, V., Chen, D.-Y., Menon, V., and Srinivas, S. Practical experiences with java software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (New York, 2008), Association of Computing Machinery, pp. 287–288.

[7] Chuang, W., Narayanasamy, S., Venkatesh, G., Sampson, J., Biesbrouck, M. V., Pokam, G., Calder, B., and Colavin, O. Unbounded

page-based transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, 2006), Association of Computing Machinery, pp. 347–358.

[8] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, 2006), Association of Computing Machinery, pp. 336–346.

[9] Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture* (Washington, DC, 2004), IEEE Computer Society, p. 102.

[10] McDonald, A., Chung, J., Carlstrom, B. D., Minh, C. C., Chafi, H., Kozyrakis, C., and Olukotun, K. Architectural semantics for practical transactional memory. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture* (Washington, DC, 2006), IEEE Computer Society, pp. 53–65.

[11] Minh, C. C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., and Olukotun, K. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture* (New York, 2007), Association of Computing Machinery, pp. 69–80.

[12] Misra, J., and Cook, W. R. Computation orchestration: A basis for wide area computing. *Journal of Software and Systems Modeling* (2006).

[13] Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D., and Wood, D. A. LogTM: Log-based transactional memory. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA '06)* (2006).

[14] Moravan, M. J., Bobba, J., Moore, K. E., Yen, L., Hill, M. D., Liblit, B., Swift, M. M., and Wood, D. A. Supporting nested transactional memory in logTM. *SIGOPS Operating Systems Review 40*, 5 (2006), 359–370.

[15] Moss, J. E. B., and Hosking, A. L. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming 63*, 2 (2006), 186–201.

[16] Ni, Y., Menon, V. S., Adl-Tabatabai, A.-R., Hosking, A. L., Hudson, R. L., Moss, J. E. B., Saha, B., and Shpeisman, T. Open nesting in software transactional memory. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, 2007), Association of Computing Machinery, pp. 68–78.

[17] Rajwar, R., Herlihy, M., and Lai, K. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture* (Washington, DC, 2005), IEEE Computer Society, pp. 494–505.

[18] Ramadan, H. E., Rossbach, C. J., Porter, D. E., Hofmann, O. S., Bhandari, A., and Witchel, E. MetaTM/TxLinux: transactional memory for an operating system. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture* (New York, 2007), Association of Computing Machinery, pp. 92–103.

[19] Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., and Hertzberg, B. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, 2006), Association of Computing Machinery, pp. 187–197.

[20] Saha, B., Adl-Tabatabai, A.-R., and Jacobson, Q. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, 2006), IEEE Computer Society, pp. 185–196.

[21] Shpeisman, T., Menon, V., Adl-Tabatabai, A.-R., Balensiefer, S., Grossman, D., Hudson, R. L., Moore, K. F., and Saha, B. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (New York, 2007), Association of Computing Machinery, pp. 78–88.

[22] Shriraman, A., Spear, M. F., Hossain, H., Marathe, V. J., Dwarkadas, S., and Scott, M. L. An integrated hardware-software approach to flexible transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture* (New York, 2007), Association of Computing Machinery, pp. 104–115.

[23] Smaragdakis, Y., Kay, A., Behrends, R., and Young, M. Transactions with isolation and cooperation. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications* (New York, 2007), Association of Computing Machinery, pp. 191–210.

[24] Wang, C., Chen, W.-Y., Wu, Y., Saha, B., and Adl-Tabatabai, A.-R. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC, 2007), IEEE Computer Society, pp. 34–48.

[25] Yen, L., Bobba, J., Marty, M. R., Moore, K. E., Volos, H., Hill, M. D., Swift, M. M., and Wood, D. A. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture* (Washington, DC, 2007), IEEE Computer Society, pp. 261–272.