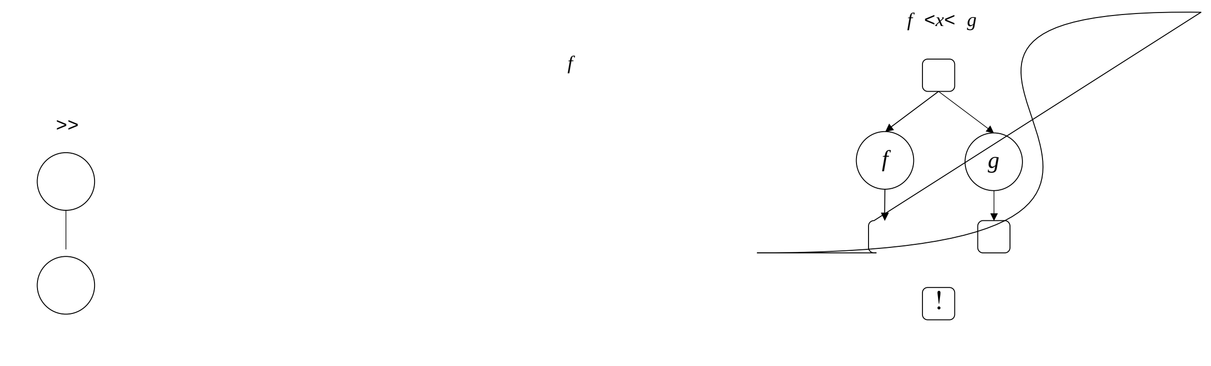
Transactional Orc

Katherine E. Coons

May 7, 2008

Abstract

how con icting non-transactional accesses are handled. Highly contended data may



(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* include calling a site, creating duplicate tokens to initiate concurrent operations, or moving the token to a di erent node.

respect to version management, con ict 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 `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

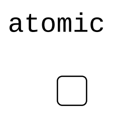
DAG is constructed recursively, preserving a single root node and a single sink node for the DAG at each step. We modi ed the compiler to insert the construct shown in Figure 2(a) when an atomic expression is encountered.

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* *j* *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



(a) Atomic DAG

with a non-null transaction object, then it is entering a nested transaction. The 9ew transaction's *parent* eld is set to the transaction object at the time it began processing the `enterTx` node. If multiple tokens from the same transaction enter 9ested transactions, as in Figure 2(c), then both 9ested transactions have the same parent transaction, forming a tree of transaction objects.

4.3 Graph Traversal

As a token leaves the `enterTx`

transaction searching for un nished tokens. If no un nished 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 speci c 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

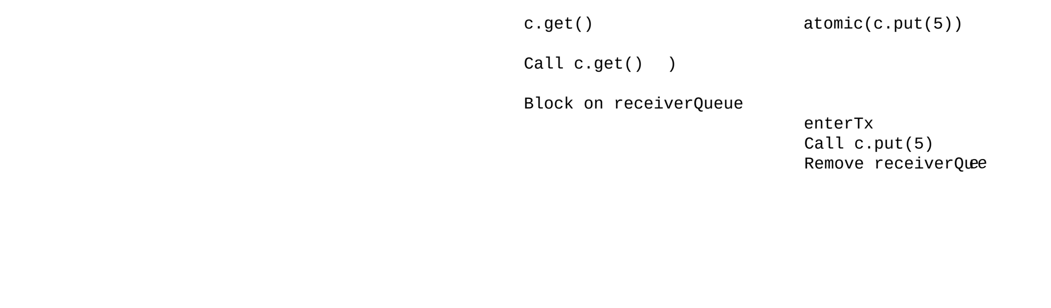
the *senderQueue* empty, the receiver returns with a special Orc value called `NIL`, indicating that no value was present.

The second channel implementation, the *receiver-blocking channel*, treats senders

ferentiate between read and write operations. Thus, only a single transaction can have ownership of the entire channel at any time. A slightly more optimistic con ict detection manager di erentiates between di erent data items within the channel, rather than treating the entire channel as a single piece of data(tia35(Th)27(u)1(s)-1(,)]TJ/F35 10.909
y-28(et)-4,20(only)419(ta)-420(single)-420(transaction)419((can)-420op)-28(eformn)-420ae

A channel is an example of a data structure that could bene t from various combinations of version management and con ict detection policies depending on the scenarios in which the channel will be use8. 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 con ict detection. In-place updates ensure that commits will be fast - the committed values are already in visible state - which is

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*.



(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 e ects, they may be useful in cases where a highly contended value is updated in a way that can be undone even af-

Figure 4 shows the operations required for two transactions to communicate with 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

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]

[17] Rajwar, R., Herlihy, M., and Lai, K. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*