

Evaluator Overview

Christine Flood
Sun Microsystems



Evaluator Overview

Copyright © 2008 Sun Microsystems, Inc. ("Sun").
All rights are reserved by Sun except as expressly stated as follows.
Permission to make digital or hard copies of all or part of this
work for personal or classroom use is granted, provided that
copies are not made or distributed for profit or commercial
advantage and that copies bear this notice and the full citation
on the first page. To copy otherwise, or republish, to post on
servers, or to redistribute to lists, requires prior specific written
permission of Sun.

Fortress Evaluator

Review Visitor Pattern

Some Examples

forBlock

evalExprList

forTupleExpr

evalExprListParallel

details of work stealing

forAtomic

details of current transactional memory implementation



Visitor Pattern

[The intent of the visitor pattern] “is to represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates”

Design Patterns, Grady Booch

Block

```
public FValue forBlock(Block x) {  
    List<Expr> exprs = x.getExprs();  
    return evalExprList(exprs, x);  
}
```

evalExprList

```
public FValue evalExprList(List<Expr> exprs, AbstractNode tag,  
                             Evaluator eval) {  
    FValue res = FVoid.V;  
    for (Expr exp : exprs) {  
        ...  
        res = exp.accept(eval);  
    }  
}  
return res;  
}
```

forTuple

Tuples are evaluated in parallel in Fortress.

```
public FValue forTupleExpr(TupleExpr x) {  
    List<Expr> exprs = x.getExprs();  
    return FTuple.make(evalExprListParallel(exprs));  
}
```

evalExprListParallel

```
<T extends Expr> List<FValue> evalExprListParallel(List<T> exprs) {
    ArrayList<FValue> resList = new ArrayList<FValue>(sz);
    TupleTask[] tasks = new TupleTask[exprs.size()];
    int count = 0;
    for (Expr e : exprs) {
        tasks[count++] = new TupleTask(e, this);
    }

    TupleTask.forkJoin(tasks);

    for (int i = 0; i < count; i++) {
        if (tasks[i].causedException()) {
            Throwable t = tasks[i].taskException(); // ellided exception handling
        }
        resList.add(tasks[i].getRes());
    }
    return resList;
}
```

Things to Note:

Tasks

FortressTaskRunner

Exception Handling



Tasks

Tasks are units of interpreter work which are put on deques

We have three types:

EvaluatorTask primordial task

TupleTasks tuples and desugaring for loops

SpawnTasks fair threads

Fair threads are for when you really want a separate OS level thread.

Spawn bug.

Why do we need a FortressTaskRunner?

We use two third party packages:

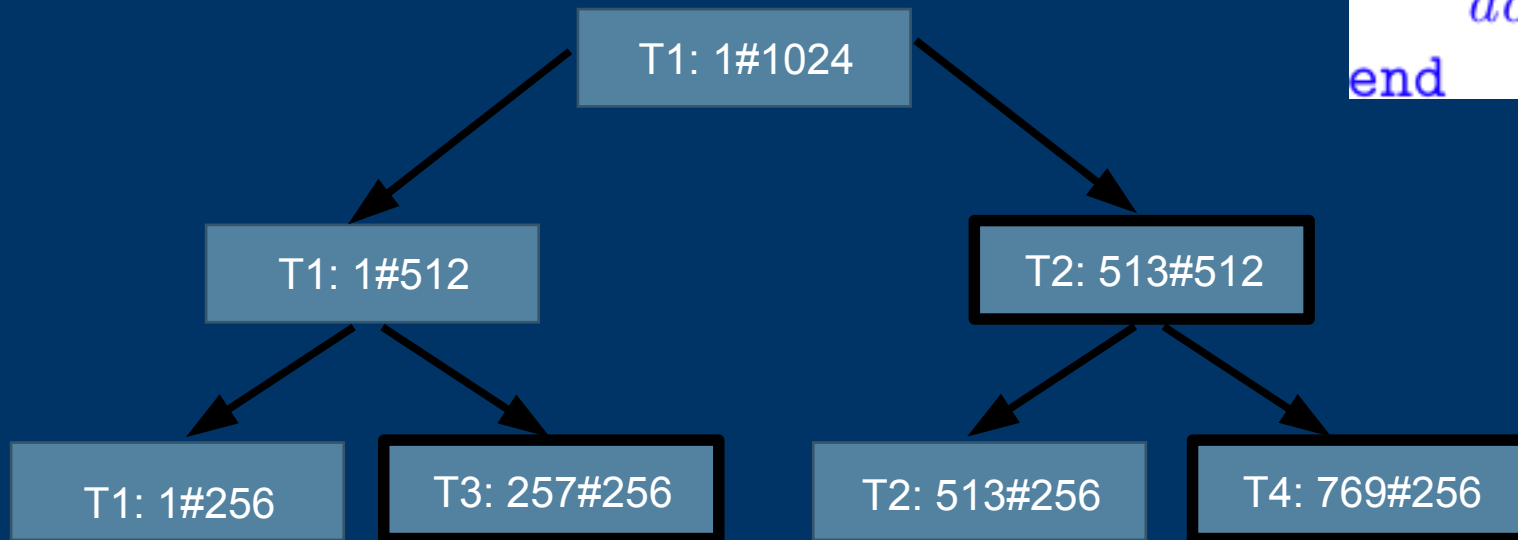
- Doug Lea's JSR166y forkjoin package (work stealing)

- Maurice Herlihy / Scalable Synch's DSTM software transactional memory package

They both want to override `java.lang.Thread` so we extend `ForkJoinWorkerThread` and add in the transactional memory stuff.

What is Work Stealing?

```
for i ← 1 # 1024 do  
    doSomething(i)  
end
```



The work is quickly distributed among threads (T1, T2, T3, and T4)
Mostly without synchronization.

The darker boxes represent work that was stolen by idle threads

If machine is busy work stays local.

How many FortressTaskRunners?

Defaults to number of available processors.

You can set with environment variable
`FORTRESS_THREADS`.



Ways of thinking of computation state.

Java stack which shows which task is currently running.

Work stealing queues which show which tasks are waiting to run.

Task Trace which shows you who your parent/child tasks are.



Exceptions

Exceptions need to follow the task trace.

```
public FValue forTry(Try x) {  
    Evaluator ev = new Evaluator(this, x);  
    Block body = x.getBody();  
    FValue res = FVoid.V;  
    try {  
        res = body.accept(this);  
        return res;  
    } catch (FortressException exc) {
```

...

If body contains a forkjoin construct (eg for loop)
then the exception must be propagated to the
parent task so that the appropriate Java thread may
catch the exception.

Atomic

```
public FValue forAtomicExpr(AtomicExpr x) {
    final Expr expr = x.getExpr();
    final Evaluator current = new Evaluator(this);
    FValue res = FortressTaskRunner.dolt (
        new Callable<FValue>() {
            public FValue call() {
                Evaluator ev = new Evaluator(new BetterEnv(current.e,
                                                            expr));
                return expr.accept(ev);
            }
        }
    );
    return res;
}
```

FortressTaskRunner doIt

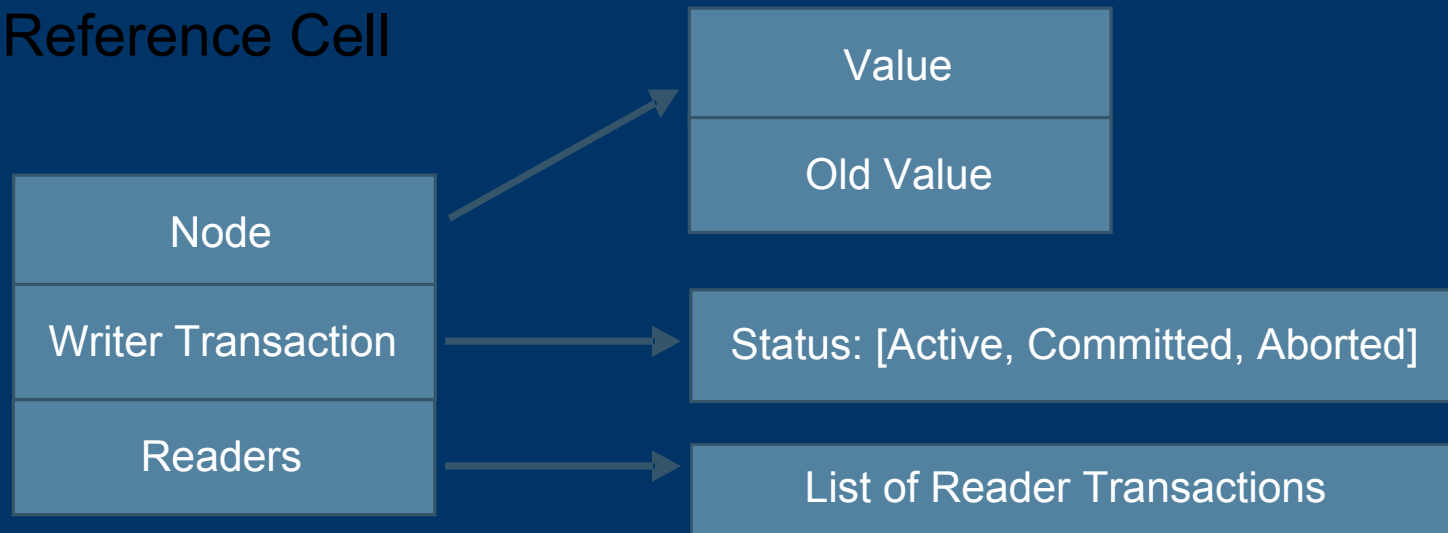
```
public static <T> T doIt(Callable<T> xaction) {  
    while (true) {  
        try {  
            T result = doItOnce(xaction);  
            return result;  
        } catch (AbortedException e) {  
            ...  
        }  
    }  
}
```

FortressTaskRunner doItOnce

```
public static <T> T doItOnce(Callable<T> xaction) {
    ThreadState threadState = BaseTask.getThreadState();
    ContentionManager manager = threadState.manager();
    T result = null;
    threadState.beginTransaction();
    try {
        result = xaction.call();
        if (threadState.commitTransaction()) {
            return result;
        } else {
            throw new AbortedException();
        }
    }
    ...
}
```

Reference Cell

Reference Cell



Status update via compare and set

Reference Cell during Transaction

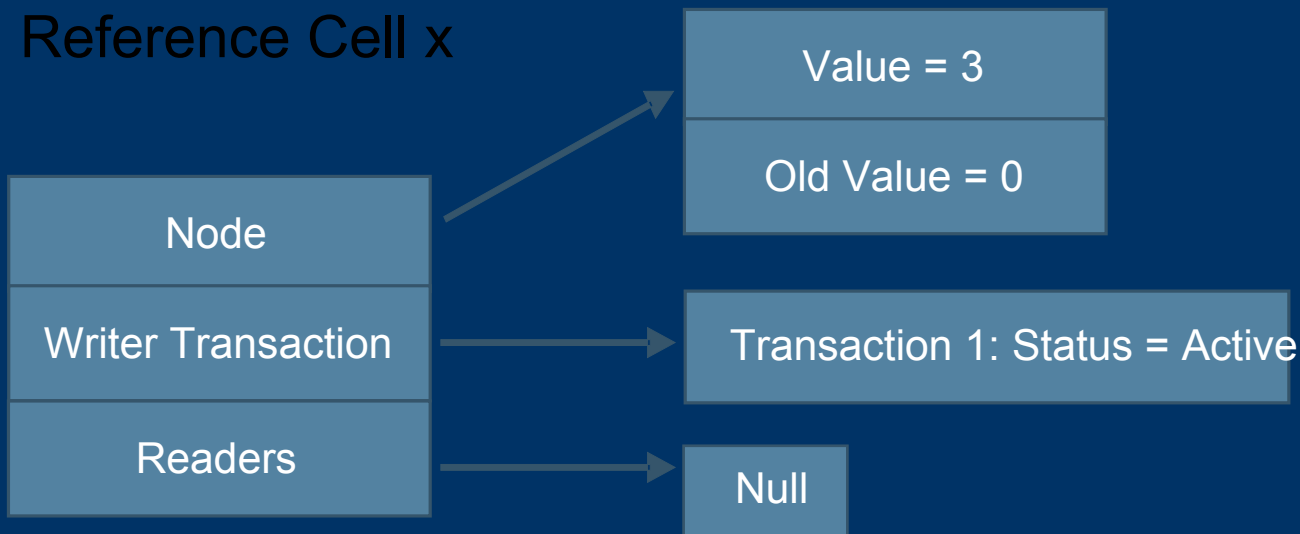
```
atomic do
  x := 3
  y := 7
end
```

← Thread 1

```
atomic do
  x := 30
  y := 70
end
```

Thread 2
discovers
conflict with
Thread 1

Reference Cell x



Contention Managers

Contention Managers
Current strategy

Transaction created by the lowest numbered thread wins.

Losers backoff via spin and retry.

One transaction always makes progress.



Per object readsets?

Why not have per transaction read sets instead of per object read sets?

A transaction would keep track of every value it read and then prior to committing updates it would validate that the read values haven't changed.

Validating the reads before a commit may take a long time; we can't block other threads for that long.

Limits to our current approach

Currently we flatten nested transactions which means that if you write:

```
atomic do
  x:=1
  atomic do
    y:=1
  end
end
end
```

It is impossible for the inner transaction to retry without retrying the outer transactions.

Limits to our current approach

Currently we disable parallelism inside of a transaction.

```
atomic do
  for i <- 1 # 30 do
    doSomething(i)
  end
end
```

The for loop will be executed sequentially.
