poster session, PoPP 2007

Featherweight Transactions: Decoupling Threads and Atomic Blocks

Virendra J. Marathe (University of Rochester)

Tim Harris and James R. Larus (Microsoft Research)

Transactional Memory (TM)

- A powerful concurrent programming abstraction
- Promises to simplify concurrent programming

Language Constructs

```
atomic {
    ... // arbitrarily complex code
    if (cond)
        retry; // conditional waiting
    ... // more complex code
}
```

Observations

Exposing TM via atomic blocks

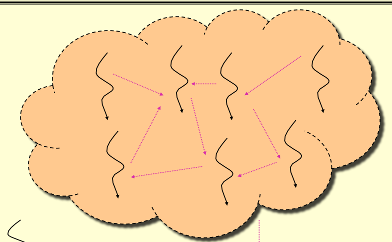
- Implicitly binds transactions to threads
- Cannot scale to support thousands/millions of transactions

Using TM for fine-grain parallelism

e.g. parallel SAT Solver

- Large numbers of short interacting pieces of work
- TM-based abstractions do not help much
- Challenge: Complex coordination among concurrent computations

Large-Scale Fine-Grain Parallelism



Parallel Computation   Coordination

`retry` enables coordination among transactions, but does not help much in controlling an aggregate group

Our Solution

Featherweight Transactions

- Transactions as schedulable *atomic work items* that run to a "quiescent" state (commit, abort or retry)
  - Insight: Transactions in quiescent state do not need a thread stack
- Daemon Workers: Re-executable work items – re-executed by the runtime system whenever any of their inputs changes
- Iterative computations on a data item

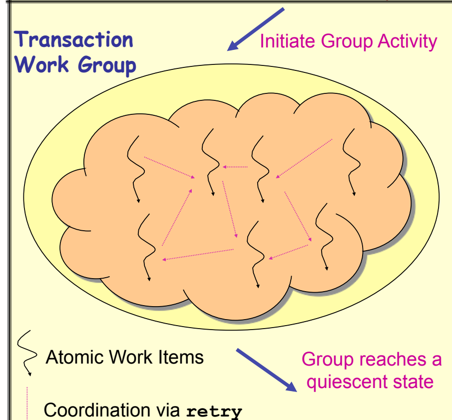
Transaction Work Groups

- Group of work items work on data parallel aggregates
- Rich semantics of work groups:
  - *wait for all work items to reach a quiescent state*
  - *suspend/resume groups of work items*
  - *group level joins, splits, parallel reductions*
  - *ordering within and among groups*
- Further investigation of work group semantics for future

Large-Scale Fine-Grain Parallelism via Transaction Work Groups

Transaction Work Group

Initiate Group Activity



Atomic Work Items

Group reaches a quiescent state

Coordination via `retry`

Parallelizing ZChaff, an efficient SAT Solver

- ZChaff employs several state-of-the-art heuristics for literal assignments
- Boolean constraint propagation (BCP): (i) recursively propagate implied literal assignments; (ii) 80% of running time
- Target BCP (fine-grain parallelism) in our parallelization (coarse-grain parallelism: no reliable performance gains)
- Conventional TM abstractions do not aid programmer in coordinating clauses for implied literal assignments

Using our new TM abstractions

- BCP parallelization is simple
- Assign distinct work item to each clause
- Main thread makes explicit literal assignments, and waits for BCP to finish

Atomic Work Item

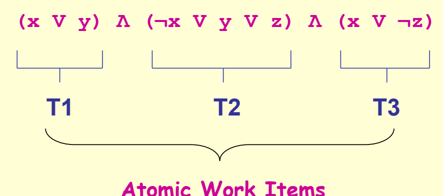
```
read all literals
 if an implied literal assignment
   make it explicit and re-execute
 else
   retry
```

Main Thread

```
while new literal assignment possible
   make an explicit literal assignment
   wait for work group to finish BCP
```

An Example

$(x \vee y) \wedge (\neg x \vee y \vee z) \wedge (x \vee \neg z)$



Atomic Work Items

Implementation of runtime in progress in MSR's Bartok backend research compiler