Scalable Software Transactional Memory for Chapel High-Productivity Language

Srinivas Sridharan and Peter Kogge, U. Notre Dame
Brad Chamberlain, Cray Inc
Jeffrey Vetter, Future Technologies Group, ORNL
Talk Overview

• Motivation:
  – Most STM designs target shared memory systems
  – Need for concurrency control on large-scale systems
    • Emerging applications do not fit the MPI model
    • Distributed memory is globally addressable (e.g. PGAS model)

• GTM: Global Transactional Memory
  – Targets large-scale distributed memory systems
    • *STM metadata overhead $\ll$ Network Latency*
  – Asynchronous STM abstractions: Parallelism inside TXs
  – Multi-core & Multi-node Environment
  – On-going work on Chapel Language STM exploration
Chapel Language

• Chapel is a parallel language developed by Cray Inc, part of the DARPA’s HPCS program
  – Primary goal is to enhance programmer productivity
    • Improve programmability, without sacrificing performance
• TM concepts satisfy Chapel’s productivity goals
  – Atomic keyword included in language specification
    • Identify transactional code segments
  – Semantics distinct from implementation mechanism
    • Based on target platform: HTM, STM, or HyTM
  – Chapel’s Multiresolution language philosophy
    • High-level counterpart to low-level \texttt{Sync} variables
Atomic keyword in Chapel

- Number of open questions under investigation:
  - Strong vs Weak Isolation?
  - Memory Consistency Model?
  - I/O in atomic blocks?
  - `Sync` variables in atomic blocks?
  - Support STM semantics across multiple Locales?
    - `Locale` is an architectural unit of locality
    - Threads within a locale have uniform access to local memory
    - Memory within other locales accessible at a price
    - E.g.: A multicore processor node in a cluster system
Distributed STM: Rationale

• Need for concurrency control across nodes
• STM can provide productivity benefits
  – Programmability advantages over locks
  – Lock-based approaches don’t scale (serialization issues)
    • No global hardware cache coherence
  – STM metadata overhead $<<$ Network Latency
    • In multicores: locks preferred over STM for performance reasons
    • Comparable performance between locks and STM if communication requirements are the same
• Key: Tolerate remote communication latency
Example: Bank Transaction

Chapel Source Code

```chapel
var balance: [1..num] int;
atomic {
  balance[i] -= amount;
  balance[j] += amount;
}
```

Multicore STM Library

```c
TX_BEGIN;
t1 = TX_LOAD(&balance[i]);
t1 = t1 - amount;
TX_STORE(&balance[i], t1);
t2 = TX_LOAD(&balance[j]);
t2 = t2 + amount;
TX_STORE(&balance[j], t2);
TX_COMMIT;
```

Atomic statement block is mapped to a sequence of STM library calls.

- **TX_BEGIN**: Start Transaction
- **TX_LOAD**: Transactional Read
- **TX_STORE**: Transactional Write
- **TX_COMMIT**: Commit Transaction

Shared Memory

(balance[1..num])
Example: Bank Transaction

Chapel Source Code

```chapel
var balance: [1..num] int;
atomic {
    balance[i] -= amount;
    balance[j] += amount;
}
```

Distributed STM Library

```chapel
TX_BEGIN;
t1 = TX_LOAD(node1, &balance[i]);
t1 = t1 - amount;
TX_STORE(node1, &balance[i], t1);
t2 = TX_LOAD(node2, &balance[j]);
t2 = t2 + amount;
TX_STORE(node2, &balance[j], t2);
TX_COMMIT;
```

PGAS models allow direct access to remote memory. Global Memory Access = `<Node-id, Local address>`

- **TX_LOAD**: Remote Transactional Read
- **TX_STORE**: Remote Transactional Write

Distributed Memory (PGAS)
GTM: Global Transactional Memory

Motivation

Asynchronous STM Abstractions
GTM Design
Scalability Results
Related Work
Future Directions
Asynchronous STM Abstraction

- STMs enforce a blocking STM abstraction
  - Return from STM call only after request fully satisfied
  - Performance ramifications:
    - Multicores: STM metadata management overheads
    - Distributed memory: Remote communication overheads

- Asynchronous abstraction helps resolve issue
  - Differentiate between when request is issued from when request is expected to complete
    - Simultaneous STM requests in-flight
  - Overlap remote latency with local computation and/or other independent communication
  - Reduce single-node STM overheads (future work)
Example: Bank Transaction

Synchronous (Blocking) STM Abstraction

```plaintext
TX_BEGIN;
t1 = TX_LOAD(node1, &balance[i]);
t2 = TX_LOAD(node2, &balance[j]);
t1 = t1 - amt;
t2 = t2 + amt;
TX_STORE(node1, &balance[i], t1);
TX_STORE(node2, &balance[j], t2);
TX_COMMIT;
```

Asynchronous (Non-blocking) STM Abstraction

```plaintext
TX_BEGIN;
TX_L_NB(t1, node1, &balance[i]);
TX_L_NB(t2, node2, &balance[j]);
TX_WAIT(t1);
t1 = t1 - amt;
TX_S_NB(node1, &balance[i], t1);
TX_WAIT(t2);
t2 = t2 + amt; ...
```

TX_LOAD and TX_STORE:
Issue Transactional Read/Write request and wait for results to arrive. Remote communication latency affects performance.

TX_L_NB and TX_S_NB:
Issue Transactional Read/Write request and return immediately.
TX_WAIT:
Wait for request to complete.
GTM: Global Transactional Memory

Motivation
Asynchronous STM Abstractions

GTM Interface
Scalability Results
Related Work
Future Directions
GTM Framework

Standalone Framework

- C Language
- GTM
- GASNet Layer
- Network Conduit (e.g. Portals, MPI)

Rest of this talk

Chapel-GTM Framework

- Chapel Compiler
- GTM
- Chapel Runtime
- GASNet, MPI
- Network Conduit

On-going Work
GTM Execution Environment

- Fixed number of SPMD tasks created at startup
- SPMD tasks/nodes can be multithreaded
  - Exploit hardware thread-level parallelism
- Partitioned Global Address Space (PGAS) model
  - Transactional access of entire global address space
- Compatible with Chapel’s runtime environment
GTM Interface Functionality

• Initialize and Clean-up STM runtime
• Start and Commit transactions
• Blocking and Non-blocking Variations
  – Transactional load/store:
    • Transfer data between global memory and private storage
  – Transactional malloc/free:
    • Dynamically manage local/remote transactional storage
  – Transactional Remote Procedure Call (RPC):
    • Execute user-level procedures on the target node
      • For exploiting locality (stay tuned...)
• Manage pending non-blocking requests
GTM Descriptors

• Transaction Descriptor or TDesc (tx):
  – Handle for identifying a transaction
  – Tracks private metadata describing the transaction

• Handle Descriptor or HDesc (op):
  – Handle for identifying a non-blocking request
  – NULL for synchronous/blocking requests

• Node Descriptor or NDesc:
  – Target node on whose context request must execute
  – Each operation has target source and node
    • If source and target are same, then operation is local else
      operation will generate remote communication
## Managing Transactional State

<table>
<thead>
<tr>
<th>GTM Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tx = gtm_tx_create()</code></td>
<td>Returns a new TDesc <code>tx</code></td>
</tr>
<tr>
<td><code>gtm_tx_destroy(tx)</code></td>
<td>Destroys the transaction <code>tx</code></td>
</tr>
<tr>
<td><code>gtm_tx_begin(tx)</code></td>
<td>Begin executing transaction <code>tx</code></td>
</tr>
<tr>
<td><code>gtm_tx_commit(tx)</code></td>
<td>Attempt to commit transaction <code>tx</code></td>
</tr>
<tr>
<td><code>gtm_tx_abort(tx)</code></td>
<td>Abort transaction <code>tx</code></td>
</tr>
<tr>
<td><code>op = gtm_op_create()</code></td>
<td>Returns a new handle descriptor <code>op</code></td>
</tr>
<tr>
<td><code>gtm_op_destroy(op)</code></td>
<td>Destory the handle <code>op</code></td>
</tr>
</tbody>
</table>

- Transactions must be started and committed by same node
- All calls are local and blocking
  - Commit/Abort may implicitly generate messages

Scalable STM for the Chapel High-Productivity Language
**GTM Call Semantics**

<table>
<thead>
<tr>
<th>Call Semantics</th>
<th>HDesc (op)</th>
<th>NDesc (tgt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Blocking</td>
<td>NULL</td>
<td>Source</td>
</tr>
<tr>
<td>Local Non-Blocking</td>
<td>Valid HDesc</td>
<td>Source</td>
</tr>
<tr>
<td>Remote Blocking</td>
<td>NULL</td>
<td>Remote</td>
</tr>
<tr>
<td>Remote Non-Blocking</td>
<td>Valid HDesc</td>
<td>Remote</td>
</tr>
</tbody>
</table>

- **HDesc and NDesc determine call semantics**
  - **HDesc**: Blocking or Non-Blocking
    - Valid HDesc: No active request, Non-NULL
  - **NDesc**: Local or Remote operation
**Transaction Load Interface**

```c
#define gtm_tx_load(tx, op, tgt, destAddr, srcAddr, size)

On node0:
  gtm_tx_begin(tx);
  gtm_tx_load(tx, NULL, node0, addr1, addr2, 4);
  ...
  gtm_tx_commit(tx);
```

- **Blocking Call**
- **Local Operation**

![Diagram](image)
**Transactional Load Interface**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gtm_tx_load</code></td>
<td>Transactional load operation</td>
</tr>
</tbody>
</table>

On node0:

```c
gtm_tx_begin(tx);
gtm_tx_load(tx, NULL, node1, addr1, addr2, 4);
...
gtm_tx_commit(tx);
```

**Blocking Call**

**Remote Operation**

[Diagram showing nodes node0 and node1 with addr1 and addr2]
### Transactional Load Interface

```c
#define gtm_tx_load

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gtm_tx_load</code></td>
<td><code>(tx, op, tgt, destAddr, srcAddr, size)</code></td>
</tr>
</tbody>
</table>

On node0:

```c
HDesc *op1 = gtm_op_create();
gtm_tx_begin();
gtm_tx_load(tx, op1, node1, addr1, addr2, 4);
```

**Non-Blocking Call**

**Remote Operation**

![Diagram showing transactional load interface](image)
## Managing Non-Blocking Requests

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gtm_op_wait(tx, op)</code></td>
<td>Wait for request on op to complete. If op fails then abort tx.</td>
</tr>
<tr>
<td><code>gtm_op_test(tx, op)</code></td>
<td>Return status of request on op.</td>
</tr>
</tbody>
</table>

### On node 0:

```c
gtm_tx_load(tx, op1, node1, addr1, addr2, 4);
<computation or independent communication>
gtm_op_wait(tx, op1);
```

```c
gtm_tx_load(tx, op1, node1, addr1, addr2, 4);
...
gtm_op_test(tx, op1);
```
# Transactional Data Management

- **Same call semantics as `gtm_tx_load`**
  - Must be called inside transactional boundaries
  - Use the same calls for managing non-blocking requests

```plaintext
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gtm_tx_store(tx, op, tgt, srcAddr, size, destAddr)</code></td>
<td>Transactional store of size bytes from <code>destAddr</code> on <code>tgt</code> to <code>srcAddr</code> on callee.</td>
</tr>
<tr>
<td><code>gtm_tx_malloc(tx, op, tgt, size, addr)</code></td>
<td>Transactional allocation of <code>size</code> bytes starting at <code>addr</code> on <code>tgt</code>.</td>
</tr>
<tr>
<td><code>gtm_tx_free(tx, op, tgt, size, addr)</code></td>
<td>Transactional free of <code>size</code> bytes starting at <code>addr</code> on <code>tgt</code>.</td>
</tr>
</tbody>
</table>
```
Transactioal RPC Mechanism

```
node1

node0

node2

tx_begin(tx1);
tx_load(tx1, node1)
tx_load(tx1, node2)
...
tx_store(tx1,node2)
tx_commit(tx1)

node1

node0

node2

fn (tx1) {
    tx_load(tx1,node2)
    ...
    tx_rpc(tx1,fn,node2)
    ...
    tx_store(tx1,node2)
}
```
Transactional RPC Interface

```plaintext
gtm_tx_fn(tx, op, tgt, fName, iBuf, isize, oBuf, oSize)
```

Execute `fName` on `tgt` node.
Local or Remote variations.
Blocking or Non-Blocking variations.
Input arguments: `iBuf` (size `iSize`)
Output results: `oBuf` (size `oSize`)

- `gtm_op_test` and `gtm_op_wait` for managing pending requests.
- Can be called from outside transactional boundary
  - Execute independent transactions on remote nodes
## STM Algorithmic Choices

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Algorithmic Choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nesting semantics</td>
<td>nesting transactional blocks</td>
<td>Flat</td>
</tr>
<tr>
<td>Granularity</td>
<td>size of transactional data</td>
<td>Word</td>
</tr>
<tr>
<td>Conflict Detection</td>
<td>when conflicts are detected</td>
<td>Early</td>
</tr>
<tr>
<td>Write synchronization</td>
<td>how writes are handled</td>
<td>Deferred</td>
</tr>
<tr>
<td>Read synchronization</td>
<td>how reads are handled</td>
<td>Read-Versioning</td>
</tr>
<tr>
<td>Conflict tolerance</td>
<td>semantics for read</td>
<td>Validation</td>
</tr>
<tr>
<td>Forward Progress</td>
<td>completion guarantees</td>
<td>None</td>
</tr>
</tbody>
</table>
GTM: Global Transactional Memory

Motivation
Asynchronous STM Abstractions
GTM Interface

Scalability Results
Related Work
Future Directions
Experimental Methodology

- **ORNL NCCS Jaguar Cray XT4**
  - 2.1 GHz Quad-core Opteron, 8GB memory
- **Red-Black Tree Benchmark**
  - Each node maintains balanced RB Tree
  - One thread per SPMD task
    - Insert, Delete, and Update operations across all nodes
- **Additional results not presented**
  - Priority Queue and Bank Transaction Benchmark
  - Effects of multithreading, Serialization issues in GASNet
Execution Time: Red-Black Tree

- **Strong Scaling**
  - Symbols: <GTM_B, S>, <GTM_B, M>, <GTM_B, L>, <GTM_NB, S>, <GTM_NB, M>, <GTM_NB, L>
  - Total TX Commits:
    - S: $2^{24}$
    - M: $2^{28}$
    - L: $2^{30}$

- **Weak Scaling**

Number of SPMD Nodes vs. Total Execution Time in seconds (logscale)
Speedup: Red-Black Tree

![Graph showing speedup vs number of SPMD nodes for different Total TX Commits (S: $2^{24}$, M: $2^{28}$, L: $2^{30}$).]
GTM: Global Transactional Memory

Motivation
Asynchronous STM Abstractions
GTM Interface
Scalability Results
Related Work
Future Directions
Related Work: Cluster-STM

- Cluster-STM: Chapel’s past collaboration with UIUC
  - PPoPP ‘08: Bocchino, Adve, and Chamberlain
- First to provide RPC with STM semantics

<table>
<thead>
<tr>
<th>Feature</th>
<th>GTM</th>
<th>Cluster-STM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallelism Environment</td>
<td>SPMD-Threads</td>
<td>Strict SPMD</td>
</tr>
<tr>
<td>Asynchronous Abstraction</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Transactional Memory Region</td>
<td>Global Address Space</td>
<td>Limited to fixed segment</td>
</tr>
<tr>
<td>Transaction Identifier</td>
<td>TDesc</td>
<td>SPMD Id</td>
</tr>
<tr>
<td>STM Algorithms</td>
<td>One</td>
<td>Four</td>
</tr>
</tbody>
</table>
GTM: Global Transactional Memory

Motivation
Asynchronous STM Abstractions
GTM Interface
Scalability Results
Related Work

Future Directions
Future Directions

- Chapel Runtime
  - Under progress: Chapel-GTM runtime exploration
  - Use asynchronous abstraction to reduce scalar STM metadata management overheads

- Chapel Compiler
  - Implement `Atomic` keyword
    - Compiler optimizations

- Develop benchmarks to benefit from Chapel-GTM
  - Under progress: Bader MST, SAT solver, NAS UA
  - Suggestions and possible collaborations...
More information:
chapel.cs.washington.edu

Carpe TM!
Thank You.