Code Generation and Optimization for Transactional Memory Construct in an Unmanaged Language


Programming Systems Lab
Microprocessor Technology Labs
Intel Corporation

*Computer Science Division
University of California, Berkeley
Motivation

• Existing Transactional Memory (TM) constructs focus on managed Language

• Efficient software transactional memory (STM) takes advantages of managed language features
  – Optimistic Versioning (direct update memory with backup)
  – Optimistic Read (invisible read)

• Challenges in Unmanaged Language (e.g. C)
  – Consistency
    • No type safety, first-class exception handling
  – Function call
    • No just-in-time compilation
  – Stack rollback
    • Stack alias
  – Conflict detection
    • Not object oriented
Contributions

• First to introduce comprehensive transactional memory construct to C programming language
  – Transaction, function called within transaction, transaction rollback, ...

• First to support transactions in a production-quality optimizing C compiler
  – Code generation, optimization, indirect function calls, ...

• Novel STM algorithm and API that supports optimizing compiler in an unmanaged environment
  – quiescent transaction, stack rollback, ...
Outline

• TM Language Construct
• STM Runtime
• Code Generation and Optimization
• Experimental Results
• Related Work
• Conclusion
TM Language Constructs

- `#pragma tm_atomic`
  ```
  {
    stmt1;
    stmt2;
  }
  ```

- `#pragma tm_atomic`
  ```
  {
    stmt 1;
    #pragma tm_atomic
    {
      stmt2;
      ...
      tm_abort();
    }
  }
  ```

- `#pragma tm_function`
  ```
  int foo(int);
  ```

- `#pragma tm_function`
  ```
  int bar(int);
  ```

- `#pragma tm_atomic`
  ```
  {
    foo(3); // OK
    bar(10); // ERROR
  }
  ```

- `#pragma tm_atomic`
  ```
  {
    foo(2) // OK
    bar(1) // OK
  }
  ```
Consistency Problem

Thread 1

```c
#pragma tm_atomic
{
  if(tq->free) {
    for(temp1 = tq->free;
       temp1->next &&...,
       temp1 = temp1->next);
    task_struct[p_id].loc_free = tq->free;
    tq->free = temp1->next;
    temp1->next = NULL;
  }
}
```

Thread 2

```c
#pragma tm_atomic
{
  if(tq->free) {
    for(temp2 = tq->free;
       temp2->next &&...,
       temp2 = temp2->next);
    task_struct[p_id].loc_free = tq->free;
    tq->free = temp2->next;
    temp2->next = NULL;
  }
}
```

- **Solution**: timestamp based aggressive consistent checking
Inconsistency Caused by Privatization

Thread 1

```c
#pragma tm_atomic
{
    if(tq->free) {
        for(temp1 = tq->free;
            temp1->next && ...,
            temp1 = temp1->next);
        task_struct[p_id1].loc_free = tq->free;
        tq->free = temp1->next;
        temp1->next = NULL;
    }
    ...
}
```

Thread 2

```c
#pragma tm_atomic
{
    if(tq->free) {
        for(temp2 = tq->free;
            temp2->next && ...,
            temp2 = temp2->next);
        task_struct[p_id2].loc_free = tq->free;
        tq->free = temp2->next;
        temp2->next = NULL;
    }
    ...
}
```

- **Solution:** Quiescent Transaction
Quiescent Transaction

Thread 1
#pragma tm_atomic
{
    if(tq->free) {
        for(temp1 = tq->free;
            temp1->next &&...,
            temp1 = temp1->next);
        task_struct[p_id1].loc_free = tq->free;
        tq->free = temp1->next;
        temp1->next = NULL;
    }
    ...
TM Runtime Issues (Stack Rollback)

```c
#include <tm.h>

void bar(int *p) {
    *p = 42;
}

void foo() {
    int a;
    bar(&a);
    ... // abort
}
```

- **Solution**: Selective Stack Rollback
#pragma tm_atomic
{
    a = b + 1;
    ...; // may alias a or b
    a = b + 1;
}

desc = stmGetTxnDesc();
rec1 = IRComputeTxnRec(&b);
ver1 = IRRestore(desc, rec1);
t = b;
IRCheckRead(desc, rec1, ver1);

desc = stmGetTxnDesc();
rec2 = IRComputeTxnRec(&a);
IWRite(desc, rec2);
IRUndoLog(desc, &a);
a = t + 1;

desc = stmGetTxnDesc();
rec1 = IRComputeTxnRec(&b);
ver1 = IRRestore(desc, rec1);
t = b;
IRCheckRead(desc, rec1, ver1);

desc = stmGetTxnDesc();
rec2 = IRComputeTxnRec(&a);
IWRite(desc, rec2);
IRUndoLog(desc, &a);
a = t + 1;
Experiment Setup

• Target System
  – 16-way IBM eServer xSeries 445, 2.2GHz Xeon
  – Linux 2.4.20, icc v9.0 (with STM), -O3

• Benchmarks
  – 3 synthetic concurrent data structure benchmarks
    • Hashtable, btree, avltree
  – 8 SPLASH-2 benchmarks
    • 4 SPLASH-2 benchmarks spend little time in critical sections
  – Fine-grained lock v. coarse-grained lock v. STM
    • Coarse-grain lock: replace all locks with a single global lock
    • STM:
      – Replace all lock sections with transactions
      – Put non-transactional conflicting accesses in transactions
Hashtable

- STM scales similarly as fine grain lock
- Manual and compiler STM comparable performance
• STM is much better than coarse-grain lock
Splash 2

- STM can be more scalable than locks
Optimization Benefits

- The overhead is within 15%, with average only 6.4%
Related Work

• Transactional Memory
  – [Herlihy, ISCA93]
  – [Ananian, HPCA05], [Rajwar, ISCA05], [Moore, HPCA06],
    [Hammond, ASPLOS04], [McDonald, ISCA06], [Saha, MICRO 06]

• Software Transactional Memory
  – [Shavit, PODC95], [Herlihy, PODC03], [Harris, ASPLOS04]

• Prior work on TM constructs in managed languages
  – [Adl-Tabatabai, PLDI06], [Harris, PLDI06], [Carlstrom, PLDI06],
    [Ringlegerg, ICFP05]

• Efficient STM
  – [Saha, PPoPP06]

• Time-stamp based approach
  – [Dice, DISC06], [Riegel, DISC06]
Conclusion

• We solve the key STM compiler problems for unmanaged languages
  – Aggressive consistency checking
  – Static function cloning
  – Selective stack rollback
  – Cache-line based conflict detection

• We developed a highly optimized STM compiler
  – Efficient register rollback
  – Barrier elimination
  – Barrier inlining

• We evaluated our STM compiler with well-known parallel benchmarks
  – The optimized STM compiler can achieve most of the hand-coded benefits
  – There are opportunities for future performance tuning and enhancement
Questions?
STM Runtime API

TxnDesc*  stmGetTxnDesc();
uint32  stmStart(TxnDesc*, TxnMemento*);
uint32  stmStartNested(TxnDesc*, TxnMemento*);
void  stmCommit(TxnDesc*);
void  stmCommitNested(TxnDesc*);
void  stmUserAbort(TxnDesc*);
void  stmAbort(TxnDesc*);
uint32  stmValidate(TxnDesc*);
uint32*  stmComputeTxnRec(uint32* addr);
uint32  stmRead(TxnDesc*, uint32* txnRec);
void  stmCheckRead(TxnDesc*, uint32* txnRec, uint32 version);
void  stmWrite(TxnDesc*, uint32* txnRec);
Void  stmUndoLog(TxnDesc*, uint32* addr, uint32 size);
Example 1

• #pragma tm_atomic
• {
  • t = head;
  • Head = t->next;
• }

• ... = *t;

• #pragma tm_atomic
• {
  • s = head;
  • *s = ...;
• }
Example 2

• #pragma tm_atomic
• {
•   t = head;
•   head = t->next;
• }

• ... = *t;

• #pragma tm_atomic
• {
•   s = head;
•   *s = ...;
•   head = s->next;
• }

Example 3

- #pragma tm_atomic
- {
-   t = head;
-   head = t->next;
- }
- *t = ...;

- #pragma tm_atomic
- {
-   s = head;
-   ... = *s;
-   head = s->next;
- }

-
Optimization Issues (Register Checkpointing)

• Source Code

```c
#pragma tm_atomic
{
    t1 = 0;
    t2 = t1 + t2;
    ...
}
t1 = t3;
t3 = 1;
```

• Checkpointing Code

```c
t2_bkup = t2;
while(setjmp(...)) {
    t2 = t2_bkup;
}
stmStart(...)
    t1 = 0;
    t2 = t1 + t2;
    ...
stmCommit(...);
t1 = t3;
t3 = 1;
```

• Optimized Code

```c
t2_backup = t2;
t1 = 0;
while(setjmp(...)) {
    t2 = t2_backup;
}
stmStart(...);
t2 = t1 + t2;
t1 = t3;
t3 = 1;
stmCommit(...);
```

• Checkpointing all the live-in local data does not work with compiler optimizations across transaction boundary

Abort

can not recover
**TimeStamp based Consistency Checking**

**Thread 1**

```c
#pragma tm_atomic
{
    if(tq->free) {
        for(temp1 = tq->free;
            temp1->next &&...,
            temp1 = temp1->next);
        task_struct[p_id].loc_free = tq->free;
        tq->free = temp1->next;
        temp1->next = NULL;
    }
}
```

**Thread 2**

```c
#pragma tm_atomic
{
    if(tq->free) {
        for(temp2 = tq->free;
            temp2->next &&...,
            temp2 = temp2->next);
        task_struct[p_id].loc_free = tq->free;
        tq->free = temp2->next;
        temp2->next = NULL;
    }
}
```

**Global Timestamp 0**

**Local Timestamp 0**
Checkpointing Approach

normal entry

\[ \begin{align*}
    t_2\_bkup &= t_2 \\
    t_3\_bkup &= t_3
\end{align*} \]

retry entry

\[ \begin{align*}
    t_2 &= t_2\_bkup \\
    t_3 &= t_3\_bkup
\end{align*} \]

#pragma tm_atomic
\[
\begin{align*}
    \{ \\
    \quad t_1 &= 0 \\
    \quad t_2 &= t_1 + t_2; \\
    \quad \ldots \\
    \} \\
    t_1 &= t_3; \\
    t_3 &= 1;
\end{align*}
\]

Optimization

#pragma tm_atomic
\[
\begin{align*}
    \{ \\
    \quad t_2 &= t_1 + t_2; \\
    \quad t_1 &= t_3; \\
    \quad t_3 &= 1; \\
    \quad \ldots \\
    \} 
\end{align*}
\]

retry entry

\[ \begin{align*}
    t_2\_bkup &= t_2 \\
    t_3\_bkup &= t_3 \\
    t_1 &= 0
\end{align*} \]
Function Clone

• Source Code

```c
#pragma tm_function
void foo(...) {
    ...
}
```

• STM Code

```c
<stdio-4>: 
&foo_tm
<stdio>: // normal version
no-op maker
... // normal code

<stdio_tm>: // transactional version
... // code for transaction

#pragma tm_atomic
{
    foo();
    (*fp)();
```

```c
foo_tm();
    if(*fp == "no-op marker")
        (**(fp-4))(); // call foo_tm
    else
        handle non-TM binary
```
• STM is much better than coarse-grain lock (fine lock ???)