Programming Support for Speculative Execution with Software Transactional Memory

Min Feng  
Rajiv Gupta  
Iulian Neamtiu

NEC Laboratories America  
University of California, Riverside

University of California, Riverside
Loop Parallelization

Thread 1  Thread 2  Thread 3  Thread 4

Speculative parallelization
with
Software Transaction Memory
Practical Issues

- Larger transactions => excessive instrumentation
  - Hundreds of potential shared accesses
  - Many functions called inside transactions
- Library functions
  - Source code not available
  - Sometimes irreversible
Transaction Representation

• Conceptual form

// insert a node into a linked list
atomic {
    node = head;
    while (node->next != NULL &&
            node->val != newnode->val)
    {
        node = node->next;
        insert(newnode, node);
    }
}

void insert(Node *newnode, Node *node) {
    newnode->next = node->next;
    node->next = newnode;
}
STM Libraries

// insert a node into a linked list
{
    _begin_tx(txDesc);   // <-- transaction boundary
    node = __tx_read(txDesc, &head);  // <-- shared read
    while (__tx_read(txDesc, &node->next) != NULL &&
        __tx_read(txDesc, &node->val) != __tx_read(txDesc, &newnode->val))
        __tx_write(txDesc, &node, __tx_read(txDesc, &node->next));  // <-- shared write
    insert(newnode, node, txDesc);
    _end_tx(txDesc);       // <-- transaction boundary
}

void insert(Node *newnode, Node *node, TxDescriptor *txDesc) {
    // <-- pass descriptor
    __tx_write(txDesc, &newnode->next, __tx_read(txDesc, &node->next));
    __tx_write(txDesc, &node->next, newnode);
}

For each STAMP Benchmark, on average 177 STM constructs are added and
2750 lines of code need to be examined.
// insert a node into a linked list
__tm_atomic {
    node = head;
    while (node->next != NULL &&
        node->val != newnode->val)
        node = node->next;
    insert(newnode, node);
}

__attribute__((tm_callable))  // <-- annotate function header
void insert(Node *newnode, Node *node) {
    newnode->next = node->next;
    node->next = newnode;
}

For each STAMP benchmark, there are on average 58 functions called inside transactions.
Our Constructs

```c
#pragma tm transaction
{
    node = head;
    while (node->next != NULL &&
           node->val != newnode->val)
        node = node->next;
    insert(newnode, node);
}

void insert(Node *newnode, Node *node) {
    newnode->next = node->next;
    node->next = newnode;
}
```

**Our constructs do not require:**
1. Inserting shared read/write barriers.
2. Annotating functions called inside transactions.
Code Generation for Copying/Commit

- Code with STM pragmas
- Function Translation
- Insert low-level STM constructs
- Optimization
- Source code with low-level STM constructs
- GCC
- Executable Binary
- Our Compiler
- Call Graph Generator
- Data Race Analyzer
- STM library
Function Translation

• Classification
  ▫ Atomic functions
  ▫ Non-atomic functions
  ▫ Double-duty functions
  ▫ Dynamically-called functions

```c
int original_func()
{
    if ( inside_transaction == true )
        return atomic_func();
    // original statements start here
}
```
Code Instrumentation

1. Find shared variables
   - Use static data race analysis

2. Normalize operators
   - \( b=++a \)  \( \Rightarrow \)  \( b=(a=a+1) \)
   - \( b=a++ \)  \( \Rightarrow \)  \( b=(t=a, a=a+1, t) \)

3. Insert Barriers
   - \( a=a+1 \)  \( \Rightarrow \)  \( \_tx\_write(txDesc, \&a, \_tx\_read(txDesc, \&a) +1) \)
Optimizations

- Eliminate redundant read/write barriers

<table>
<thead>
<tr>
<th>Original code</th>
<th>Intermediate code</th>
<th>Generated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>b=a+1; b=a*b;</td>
<td>_tx_write(txDesc, &amp;b, _tx_read(txDesc, &amp;a)+1); _tx_write(txDesc, &amp;b, _tx_read(txDesc, &amp;a)* _tx_read(txDesc, &amp;b));</td>
<td>t_a=_tx_read(txDesc, &amp;a); t_b=t_a+1; t_b=t_a*t_b; _tx_write(txDesc, &amp;b, t_b);</td>
</tr>
</tbody>
</table>
Safety

- Aliasing of shared variable
  - **Problem**: Multiple temporary variables may be created for a variable pointed to by multiple pointers, thus causing data inconsistency
  - **Solution**: Kill temporary variables when a new pointer is dereferenced

- Exception handling
  - **Problem**: An exception may happen in speculatively parallel regions due to misspeculation
  - **Solution**: Atomicity checks and commits need to be performed before throwing the exception
Library Functions

- Two types of library functions cannot be transact-ified
  - Precompiled library functions
    - Source code not available
  - System calls and I/O operations
    - Cannot be rolled back
Precompiled Functions

• Code example
  char src[LEN], dst[LEN];
  #pragma tm transaction
  {
    #pragma tm precompiled \ read(dst, (*src, size), src, size) \ write((*dst, size))
    memcpy(dst, src, size);
  }

• Translated code
  // create local copies
  int l_size = stmReadInt(tx, &size);
  void *l_src = (void*)malloc(size);
  void *l_dst = (void*)malloc(size);
  stmReadBytes(tx, (void*)l_src,
                (void*)src, l_size);
  // execute the function call
  memcpy(l_dst, l_src, l_size);
  // update shared variables
  stmWriteBytes(tx, (void*)dst,
                (void*)l_dst, l_size);
  free(l_src); free(l_dst);
Irreversible Functions

- Code example
  ```c
  char ch;
  #pragma tm transaction
  {
    #pragma tm suspend read(ch)
    putchar(ch);
  }
  ```

- Translated code
  ```c
  stmBegin(tx);
  // store arguments and functions
  args.sharedpush(tx, &c, sizeof(char));
  funcs.push( F.PutCHAR );
  stmEnd(tx);
  ```

  ```c
  // call function after the transaction
  // succeeds
  void resume_suspended_funcs() {
    while ( !funcs.empty() )
      switch ( funcs.pop() ) {
        case F.PutCHAR:
          putchar( *((char*)args.pop()) );
          break;
        ...
      }
  }
  ```
Evaluation

- Our source-to-source translator was built on top of ROSE.
- We use the TL2 STM library.
- Platform: Dell Poweredge T605 (Intel Xeon 8-core 2.0Ghz) running CentOS v5.5
- Benchmarks: STAMP benchmarks + two real applications – Velvet and Incremental Tree Inducer.
## Programming Effort

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>STM Lib</th>
<th>Intel STM</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bayes</td>
<td>176</td>
<td>75</td>
<td>15</td>
</tr>
<tr>
<td>Labyrinth</td>
<td>98</td>
<td>31</td>
<td>3</td>
</tr>
<tr>
<td>Genome</td>
<td>122</td>
<td>31</td>
<td>5</td>
</tr>
<tr>
<td>Intruder</td>
<td>265</td>
<td>95</td>
<td>3</td>
</tr>
<tr>
<td>Kmeans</td>
<td>17</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Ssca2</td>
<td>55</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Vacation</td>
<td>359</td>
<td>151</td>
<td>3</td>
</tr>
<tr>
<td>Yada</td>
<td>324</td>
<td>115</td>
<td>6</td>
</tr>
</tbody>
</table>

97% fewer programming constructs inserted compared to low-level STM API
91% fewer programming constructs inserted compared to the Intel STM compiler
## Impact of Data Race Analysis

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline</th>
<th>DRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bayes</td>
<td>172</td>
<td>87</td>
</tr>
<tr>
<td>Labyrinth</td>
<td>107</td>
<td>67</td>
</tr>
<tr>
<td>Genome</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Intruder</td>
<td>183</td>
<td>152</td>
</tr>
<tr>
<td>Kmeans</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Ssca2</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>Vacation</td>
<td>196</td>
<td>180</td>
</tr>
<tr>
<td>Yada</td>
<td>424</td>
<td>304</td>
</tr>
</tbody>
</table>

DRA reduces the number of barriers by **25.3%** on average.
We achieve 1.6x speedup on average
Performance on Real Applications

(a) Velvet

(b) Incremental Tree Inducer
Conclusion

• Speculative parallelization with existing STM libraries and compilers is not an easy task.
  ▫ Excessive instrumentation
  ▫ Library functions

• Our compiler addresses these issues
  ▫ Only require marking transaction boundaries
  ▫ Support precompiled and irreversible library functions
  ▫ Optimize the code with data race analysis and redundant read/write barriers elimination