Thread-Safe Dynamic Binary Translation using Transactional Memory

JaeWoong Chung, Michael Dalton, Hari Kannan, Christos Kozyrakis

Computer Systems Laboratory
Stanford University
http://csl.stanford.edu
Dynamic Binary Translation (DBT)

- DBT
  - Short code sequence is translated in run-time
  - PIN, Valgrind, DynamoRIO, StarDBT, etc

- DBT use cases
  - Translation on new target architecture
  - JIT optimizations in virtual machines
  - Binary instrumentation
    - Profiling, security, debugging, …
Example: Dynamic Information Flow Tracking (DIFT)

```
t = XX ; // untrusted data from network
taint(t) = 1;
........
swap t, u1;
swap taint(t), taint(u1);
u2 = u1;
taint(u2) = taint(u1);
```

- Untrusted data are tracked throughout execution
  - A taint bit per memory byte is used to track untrusted data.
  - Security policy uses the taint bit.
    - E.g. untrusted data should not be used as syscall argument.

- Dynamic instrumentation to propagate and check taint bit.
DBT & Multithreading

- Multithreaded executables as input

- Challenges
  - Atomicity of target instructions
    - e.g. compare-and-exchange
  - Atomicity of additional instrumentation
    - Races in accesses to application data & DBT metadata

- Easy but unsatisfactory solutions
  - Do not allow multithreaded programs (StarDBT)
  - Serialize multithreaded execution (Valgrind)
DIFT Example: MetaData Race ⇒ Security Breach

- User code uses atomic instructions.
  - After instrumentation, there are races on taint bits.

Thread 1

\[ \text{swap } t, u1; \]

\[ \text{swap taint}(t), \text{taint}(u1); \]

Thread 2

\[ u2 = u1; \]

\[ \text{taint}(u2) = \text{taint}(u1); \]

Variables

<table>
<thead>
<tr>
<th>t</th>
<th>u1</th>
<th>u2</th>
</tr>
</thead>
<tbody>
<tr>
<td>XX</td>
<td>XX</td>
<td></td>
</tr>
</tbody>
</table>

Taint bits

<table>
<thead>
<tr>
<th>t</th>
<th>u1</th>
<th>u2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Can We Fix It with Locks?

- **Idea**
  - Enclose access to data and associated metadata, within a locked region.

- **Problems**
  - Coarse-grained locks
    - performance degradation
  - Fine-grained locks
    - locking overhead, convoying, limited scope of DBT optimizations
  - Lock nesting between app & DBT locks
    - potential deadlock
  - Tool developers should be a feature + multithreading experts.

DBT+TM, HPCA'08
Transactional Memory

- Atomic and isolated execution of a group of instructions
  - All or no instructions are executed.
  - Intermediate results are not seen by other transactions.

- Programmer
  - A transaction encloses a group of instructions.
  - The transaction is executed sequentially with the other transactions and non-transactional instructions.

- TM system
  - Parallel transaction execution.
  - Register checkpoint, data versioning, conflict detection, rollback.
  - Hardware, software, or hybrid TM implementation.
Transaction for DBT

- **Idea**
  - DBT instruments a transaction to enclose accesses to (data, metadata) within the transaction boundary.

  ```
  Thread 1
  TX_Begin
  swap t, u1;
  swap taint(t), taint(u1);
  TX_End
  Thread2
  TX_Begin
  u2 = u1;
  taint(u2) = taint(u1);
  TX_End
  ```

- **Advantages**
  - Atomic execution
  - High performance through optimistic concurrency
  - Support for nested transactions
Yes, it fixes the problem. But …

- DBT transaction per instruction is heavy.
- User locks are nested with DBT transactions.
- User transactions overlap partially with DBT transactions.
- There will be I/O operations within DBT transactions.
- User-coded conditional synchronization may be tricky.
- Transactions are not free.
Granularity of Transaction Instrumentation

- **Per instruction**
  - High overhead of executing TX_Begin and TX_End
  - Limited scope for DBT optimizations

- **Per basic block**
  - Amortizing the TX_Begin and TX_End overhead
  - Easy to match TX_Begin and TX_End

- **Per trace**
  - Further amortization of the overhead
  - Potentially high transaction conflict

- **Profile-based sizing**
  - Optimize transaction size based on transaction abort ratio
Interaction with Application Code (1)

- User lock’s semantics should be preserved regardless of DBT transactions.
  - If transaction $\supset$ locked region, fine.
  - To TM, lock variables are just shared variables.
  - If transaction $\subset$ locked region, fine.
    - Transactions are executed in critical sections protected with locks.
  - If partially overlapped, split the DBT transaction.

- User transactions may partially overlap with DBT transactions.
  - If fully nested, fine.
    - Either true transaction nesting or subsumption.
  - If partially overlapping, split the DBT transaction.
Interaction with Application Code (2)

- I/O operations are not rolled back.
  - Terminate the DBT transaction.
    - Typically, they work as barriers in DBT’s optimization.

- Conditional synchronization may cause live-lock.
  - Re-optimize the code to have a transaction per basic block.

Initially, done1 = done2 = false

Thread 1

- TX_Begin
- while(!done2);
- done1 = true;
- TX_End

Thread 2

- TX_Begin
- done2 = true;
- while(!done1);
- TX_End
Evaluation Environment

- DBT framework
  - PIN v2.0 with multithreading support
  - DIFT as a PIN tool example

- Execution environment
  - x86 server with 4 dual-core processors
  - Software TM system

- Multithreaded applications
  - 6 from SPLASH
  - 3 from SPECOmp
Baseline Performance Results

- 41% overhead on the average
  - Transaction at the DBT trace granularity
Transaction Overheads

- Transaction begin/end
  - Register checkpoint
  - Initializing and cleaning TM metadata

- Per memory access
  - Tracking the read-set & write-set
  - Detecting conflicts
  - Data versioning

- Transaction abort
  - Applying logs and restarting the transaction
  - In our tests, 0.03% of transactions abort
Transaction Begin/End Overhead

Transaction Sizing
- Longer TX amortizes the TX_Begin/End overhead.
Per Memory Access Overhead

- **Instrumentation of software TM barrier**
  
  ```
  TX_Begin
  read_barrier(t);
  write_barrier(u);
  u = t;
  read_barrier(taint(t));
  write_barrier(taint(u));
  taint(u) = taint(t);
  TX_end
  ```

- **What happens in the barrier?**
  
  - Conflict detection by recording addresses
    - Observation 1: needed only for shared variables
  
  - Data versioning by logging old values
    - Observation 2: not needed for stack variables
Software Transaction Optimization (1)

- Categorization of memory access types

Diagram:
- In stack?
  - Y: Data Versioning
  - N: Conflict Detection
- Private?
  - Y, W/ lock?: BENIGN RACE
  - N, W/ lock?: SHARED
- After TX_Begin?
  - Y: IDEMPOTENT STACK
  - N: STACK

DBT+TM, HPCA'08
Software Transaction Optimization (2)

- Average runtime overhead
  - 36% with STACK
  - 34% with BENIGN RACE
Hardware Acceleration (1)

- With software optimization, the overhead is about 35%.

- Emulating 3 types of hardware acceleration
  - Cycles spent in transaction violation is under 0.03%.

<table>
<thead>
<tr>
<th></th>
<th>Register Checkpointing</th>
<th>Conflict Detection</th>
<th>Data Versioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>STM+</td>
<td>HW (single-cycle)</td>
<td>SW read-set / write-set</td>
<td>SW Undo-log</td>
</tr>
<tr>
<td>HybridTM</td>
<td>HW (single-cycle)</td>
<td>HW Signatures</td>
<td>SW Undo-log</td>
</tr>
<tr>
<td>HTM</td>
<td>HW (single-cycle)</td>
<td>HW read-set / write-set</td>
<td>HW Undo-log</td>
</tr>
</tbody>
</table>
Hardware Acceleration (2)

- Overhead reduction
  - 28% with STM+, 12% with HybridTM, and 6% with HTM
  - Notice the diminishing return
Conclusion

- Multi-threaded executables are a challenge for DBT in the era of multi-core.
  - Races on metadata access

- Use transactions for multi-threaded translated code.
  - 41% overhead on average

- With software optimization, the overhead is around 35%.
  - Further software optimization may be possible.

- Hardware acceleration reduces the overhead down to 6%.
  - Remember the diminishing return.