RetCon: Transactional Repair without Replay

Colin Blundell, Arun Raghavan, and Milo M. K. Martin
University of Pennsylvania
This work licensed under the Creative Commons Attribution-Share Alike 3.0 United States License

• You are free:
  • to Share — to copy, distribute, display, and perform the work
  • to Remix — to make derivative works

• Under the following conditions:
  • Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
  • Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

• For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to:

  [http://creativecommons.org/licenses/by-sa/3.0/us/](http://creativecommons.org/licenses/by-sa/3.0/us/)

• Any of the above conditions can be waived if you get permission from the copyright holder.

• Apart from the remix rights granted under this license, nothing in this license impairs or restricts the author's moral rights.
A Troubling Example for Transactional Memory

In reality, it doesn’t quite work out that way:
The "modcount" field is bumped for every update … 'put' operations always have a true data conflict.

Cliff Click (Azul Systems), on Azul’s experiences with HTM
A Troubling Example for Transactional Memory

```
insert(k,v){
    size++;
    if (size > max_size)
        resize();
    b = buckets[hash(k)];
    b->insert(Entry(k,v));
}
```

```
T1
atomic{
    ...
    h->insert(k1,v1);
    ...
    h->insert(k2,v2);
    ...
}
```

```
T2
atomic{
    ...
    ...
    h->insert(k3,v3);
    ...
}
```

If keys hash to distinct buckets... still conflict on h->size

...T1 and T2 should execute in parallel
A Troubling Example for Transactional Memory

<table>
<thead>
<tr>
<th>T1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic{</td>
<td>atomic{</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>h-&gt;insert(k1,v1);</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>h-&gt;insert(k3,v3);</td>
</tr>
<tr>
<td>h-&gt;insert(k2,v2);</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>}</td>
</tr>
</tbody>
</table>

One implication: put effort into devising smarter hashtables
However, hashtable example is part of **broader problem**:

“general pattern of updates to peripheral shared values
... is very common and it kills the HTM.”

Cliff Click (Azul Systems), on Azul’s experiences with HTM
The Peripheral Data Problem

- Auxiliary data that **serializes** parallel operations
  - Hashtable size fields
  - Reference counts
  - Transaction ID’s allocated from a global counter
  - ...

- Can significantly **degrade performance**
  - genome (STAMP): `-DHASHTABLE_RESIZABLE` ➞ 50% slower
  - python: reference counts **serialize execution**
  - specjbb: ID’s cause **60% performance loss** [Chung+’06]

**Our goal:** mitigate impact in hardware
Our Approach: RetCon

- Peripheral data conflicts have **limited impact**
  - Often do not change control flow/dataflow
- **Ignore** these conflicts…**repair state** at commit
  - Inspired by selective replay [Srinivasan+’04,Sarangi+’05,…]
- **RetCon**: repair **without replay**
  - Maintain **symbolic values of outputs**
  - Track **constraints on inputs**
  - At commit: reacquire inputs, check, plug into outputs

retcon, verb:
“Deliberate changing of previously-established facts”
Wikipedia
Roadmap

- Repair via Symbolic Tracking
- The RetCon Architecture
- Evaluation
- Future Work & Conclusions
Repair via Symbolic Tracking: Motivation

What happens if T1 simply **ignores** T2’s update?

\[
h \rightarrow \text{insert}(k1, v1);
\]
\[
\vdots
\]
\[
h \rightarrow \text{insert}(k3, v3);
\]
\[
\vdots
\]

**Ignore** peripheral data conflicts during execution
**Repair** peripheral data values at commit
**Detect** more complex effects and abort

...T1 and T2 still conflict on \( h \rightarrow \text{size} \)

**Value of** \( h \rightarrow \text{size} **incorrect**

**Infrequently** impacts control flow
**Doesn’t** impact dataflow
Repair via Symbolic Tracking

• Track symbolic values of outputs
  
  ```
  size++;  
  ...
  size++;  
  ```

  \[ \text{size}_{out} = \text{size}_{in} + 2 \]

• Control flow: generate constraint on input
  
  ```
  // refct = 7
  refct--;  
  if (refct <= 0){
  ```

  \[ \text{refct}_{in} > 1 \]

• Complex dataflow: disallow input change
  
  ```
  // ptr = 0xbfff
  t = ptr->task;
  process_task(t);
  ```

  \[ \text{ptr}_{in} == 0xbfff \]

• At commit: reacquire inputs and use to repair
  • Constraints satisfied? Generate correct outputs
RetCon: Overview

• Foundation: baseline hardware TM
  • Uses read/written bits on L1 cache lines
• Selectively employs symbolic tracking
  • Via predictor that trains on history of conflicts
• New structures to maintain symbolic info
  • Shadow register file entries
  • Cache-like structures to track through memory
  • More specific in a bit
• During repair: enforces atomicity via R/W bits
RetCon: Example

Predicted “problem block”
During execution, conflict occurs
Via tracking, RetCon repairs outputs

Code sequence
```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)  // not taken
  store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```
Initiating Symbolic Tracking

Code sequence:

```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8) // not taken
    store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```
Initiating Symbolic Tracking

Code sequence:

\begin{verbatim}
xaction_begin;
... 
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
  // not taken
store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
\end{verbatim}

Would normally set read bit
Initiating Symbolic Tracking

Code sequence

```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
  // not taken
store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```

Instead, buffer value of `a`...  
...and track output

Would normally set read bit
Initiating Symbolic Tracking

Code sequence:

```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
  // not taken
store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```

Instead, buffer value of `a`...

...and track output
Initiating Symbolic Tracking

Instead, buffer value of $a$...
...and track output

Code sequence

```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8) // not taken
    store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```
Executing Past Conflicts

Code sequence

```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)  // not taken
    store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```

Regfile

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r1:</td>
<td>2</td>
</tr>
<tr>
<td>r2:</td>
<td>0</td>
</tr>
</tbody>
</table>

L1 cache

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a:</td>
<td>2</td>
</tr>
</tbody>
</table>

Input buf

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a: 2</td>
</tr>
</tbody>
</table>

Give up block without rollback

inv a
Computation on Symbolic Values

Code sequence:

```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
    // not taken
store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```

Diagram:
- **Regfile**:
  - r1: 2, a
  - r2: 0, NIL

- **L1 cache**:
  - Val: NIL
  - State: NIL

- **Input buf**:
  - a: 2
Computation on Symbolic Values

Code sequence

\[
\text{xaction\_begin;}
\ldots
\text{load } [a], r1;
\text{r1 = r1 + 1;}
\text{if (r1 >= 8) // not taken}
\text{store r1, [b];}
\ldots
\text{load [b], r2;}
\text{r2 = r2 + 1;}
\ldots
\text{xaction\_end;}
\]
Computation on Symbolic Values

Code sequence

\[
xaction\_begin;
\]
\[
... 
\]
\[
load \ [a], \ r1; 
\]
\[
r1 = r1 + 1; 
\]
\[
if \ (r1 >= 8) 
\]
\[
// not taken 
\]
\[
store \ r1, \ [b]; 
\]
\[
... 
\]
\[
load \ [b], \ r2; 
\]
\[
r2 = r2 + 1; 
\]
\[
... 
\]
\[
xaction\_end; 
\]
Constraint: ?

Control Flow

Code sequence

\begin{verbatim}
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
  // not taken
store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
\end{verbatim}
Code sequence

```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
   // not taken
  store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```

Control Flow

- **Regfile**
  - r1: 3, a + 1
  - r2: 0, NIL

- **L1 cache**
- **Input buf**
  - a: 2

Constraint: ?
Control Flow

Code sequence

\begin{verbatim}
xaction_begin;
  ...
  load [a], r1;
  r1 = r1 + 1;
  if (r1 >= 8)
    // not taken
  store r1, [b];
  ...
  load [b], r2;
  r2 = r2 + 1;
  ...
  xaction_end;
\end{verbatim}

Constraint: \((a + 1) < 8\)
Symbolic Stores

Store concrete & symbolic val

Code sequence

```c
xaction_begin;
... 
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
    // not taken 
store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
  // not taken
  store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;

Store concrete & symbolic val
Forwarding From Symbolic Stores

Code sequence

```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
  // not taken
store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```

```
Val  Sym
r1:  3  a + 1
r2:  0  NIL

Sym store buf
Val  Sym
b:   3  a + 1

Input buf
Val  Cond
a:   2  a < 7

Forward concrete & symbolic val
```

```
Val  State
L1 cache

Regfile
```

RetCon - Blundell - ISCA 2010
Forwarding From Symbolic Stores

Code sequence

```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
  // not taken
store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```

Forward concrete & symbolic val
And On We Go…

Code sequence:

```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
    // not taken
store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```
xaction_begin;
... load [a], r1;
  r1 = r1 + 1;
if (r1 >= 8)
  // not taken
  store r1, [b];
... load [b], r2;
  r2 = r2 + 1;
... xaction_end;
Code sequence

```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)  
    // not taken
store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```

(Re)acquire all blocks
xaction_begin;
... 
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8) // not taken
  store r1, [b];
... 
load [b], r2;
r2 = r2 + 1;
... 
xaction_end;

(Re)acquire all blocks

R/W bits ensure atomicity

req a, S
req b, M
Initiating Repair

Code sequence

\begin{verbatim}
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
  // not taken
store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
\end{verbatim}

Use new input value to repair
Checking Constraints

Code sequence

```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
  // not taken
store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```

- **Regfile**
  - `r1`: 3, `a + 1`
  - `r2`: 4, `a + 2`

- **L1 cache**
  - `a`: 4, `S R`
  - `b`: 7, `M W`

- **Sym store buf**
  - `b`: 3, `a + 1`

- **Input buf**
  - `a`: `a < 7`
xaction_begin;
... 
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8) 
  // not taken
store r1, [b];
... 
load [b], r2;
r2 = r2 + 1;
... 
xaction_end;
```
xaction_begin;
... 
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8) // not taken
  store r1, [b]; 
... 
load [b], r2;
r2 = r2 + 1;
... 
xaction_end;
```

**Step 1: update values**

- **Regfile**
  - `r1`: `3`, `a + 1`
  - `r2`: `4`, `a + 2`

- **L1 cache**
  - `a`: `4`, `S`, `R`
  - `b`: `7`, `M`, `W`

- **Sym store buf**
  - `b`: `3`, `a + 1`

- **Input buf**
  - Empty
Code sequence

```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8) // not taken
  store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```

Step 1: update values
Code sequence

```
xaction_begin;
... 
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
  // not taken
  store r1, [b];
... 
load [b], r2;
r2 = r2 + 1;
... 
xaction_end;
```
Repair

Code sequence

```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if(r1 >= 8) // not taken
  store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```

Step 2: perform stores
Code sequence

```c
xaction_begin;
...
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
  // not taken
store r1, [b];
...
load [b], r2;
r2 = r2 + 1;
...
xaction_end;
```

Step 2: perform stores ✓
Code sequence

```c
xaction_begin;
... 
load [a], r1;
r1 = r1 + 1;
if (r1 >= 8)
    // not taken
store r1, [b];
... 
load [b], r2;
r2 = r2 + 1;
... 
xaction_end;
```
RetCon Key Points

• How do we decide which blocks to track?
  • Predictor that trains up on conflicts…
  • …trains down on violated constraints

• What computation do we track?
  • Currently: expressions of form “[addr] + value”
  • Compact representation; sufficient for our use cases

• How do we handle computation we can’t track?
  • Constrain the input to original value
  • Input doesn’t change ➔ dataflow doesn’t change
RetCon Key Points

• How do we decide which blocks to track?
  • Predictor that trains up on conflicts…
  • …trains down on violated constraints

• See paper for handling of real-world issues
  • Condition codes
  • Coarser-than-word-granularity cache blocks
  …

• How do we handle computation we can’t track?
  • Constrain the input to original value
  • Input doesn’t change \(\Rightarrow\) dataflow doesn’t change
Bonus: RetCon Has Other Benefits

- Value-based conflict detection [Olszewski+’07, Tabba+’09]
  - Compare values to detect conflicts
  - Eliminates aborts due to false/silent sharing conflicts
  - RetCon: detects conflicts via (value-based) constraints

- Lazy conflict detection [Hammond+’04, Ceze+’06]
  - Delay writes until transaction commit
  - Mitigates convoying of readers behind writers
  - RetCon: selectively delays writes

- Will examine impact in evaluation
Evaluation Methodology

• Simulator: in-house version of FeS2 [Neelakantam+’08]
  • Full-system, execution-driven simulation

• Simulated machine: 32-core x86-based MP
  • RetCon: 8-entry input buf, 32-entry symbolic store buf

• Workloads: STAMP [Minh+’08], python, raytrace
  • STAMP: compiled with -DHASHTABLE_RESIZEABLE
  • python-opt: handful of uses of __thread keyword
  • intruder-opt: split lists, replaced r-b tree w/ hashtable
RetCon’s Performance Impact

Takeaway #1:
RetCon mitigates the peripheral data problem
Analyzing RetCon’s Other Benefits

- Lazy-qb: RetCon variant capturing laziness/false sharing only
- Neato! 100% speedup from laziness
Analyzing RetCon’s Other Benefits

Takeaway 2: RetCon has benefits beyond optimizing peripheral data

100% speedup from laziness

Woohoo! More papers!

neato!
Related Work

• ReSlice \cite{Sarangi+05}
  • Maintains insns in dependent slice of conflicting operation in TLS
  • To repair, re-executes these instructions

• Dependence-aware transactional memory \cite{Ramadan+08}
  • Forwards speculative values to optimize ordered communication
  • Unlike RetCon, can’t handle conflicts with cyclic communication…
  • …but OTOH, can handle arbitrarily complex computation

• Advanced TM interfaces & data structure implementations
  • Open nesting \cite{Moss+05,Moravan+06,Ni+07}
  • Transactional boosting \cite{Herlihy+08}
  • Abstract nesting \cite{Harris+07}
  • Lock-free hashtables \cite{Click07,…}
  • Scalable non-zero indicators \cite{Ellen+07}
Conclusions

• Focus of this work: the peripheral data problem
  • Auxiliary data that serializes parallel operations
  • Can significantly degrade overall performance
• Our solution: repair via symbolic tracking
  • Exploits simplicity of peripheral data computation…
  • …as well as limited impact of peripheral data conflicts
• RetCon: repair without replay
  • Adds selective symbolic tracking to baseline HTM
  • Side benefit: unifies prior-proposed optimizations
  • Mitigates impact of peripheral data conflicts