Using threads for Fault Tolerance

pablo montesinos
Agenda

- Introduction
- Design Space
- Uniprocessors:
  - Transient fault detection
  - Transient fault recovery
- Multiprocessors:
  - Transient fault detection
  - Transient fault recovery
Concept

- Traditional fault tolerance schemes have used hardware replication to provide redundancy.
- Simultaneous Multithreading improves the performance of a processor by allowing multiple independent threads to execute simultaneously (same cycle) in different functional units.
- Use the replication provided by the different threads to run two copies of the same program so we are able to detect errors.
Why is so attractive?

- Require less hardware than conventional hardware replication
  - Time and Data redundancy can be used in places where Space redundancy is not critical
- Increased performance due to dynamic resource partition
  - HP might not agree on this, though
What are the problems?

- Cycle-by-cycle output comparison and input replication are inappropriate for this technique
- Equivalent instructions from different threads might execute in different cycles
- Equivalent instructions from different threads might execute in different order with respect to other instructions in the same thread
- Precise scheduling of the threads is crucial if we want to achieve optimal performance
Sphere of Replication

- Logical boundary of redundant execution within a system
  - Components within the sphere enjoy fault coverage due to the redundant execution
  - Components outside must be protected via other means
- Its size matters:
  - Error detection latency
  - Stored-state size
Another Sphere of Replication
## Design Space (II)

<table>
<thead>
<tr>
<th></th>
<th>Uniprocessor</th>
<th>Multiprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>base technology</td>
<td>SMT</td>
<td>CMP</td>
</tr>
<tr>
<td>+ detection</td>
<td>SRT, RMT</td>
<td>CRT</td>
</tr>
<tr>
<td>+ detection + recovery</td>
<td>SRTR</td>
<td>CRTR</td>
</tr>
</tbody>
</table>
Processor Model
Input Replication

- Instructions: no change needed
- Cached load data:
  - Active Load Address Buffer
  - Load Value Queue
- Uncached load data: synchronized when comparing addresses that leave the SoR
- External Interrupts:
  - Stall the leading thread
  - Deliver to the leading and record
Active Load Address Buffer

- Delays a cache block’s replacement or invalidation after the retirement of the trailing load

  | tag | counter | validate |

- When a cache block is replaced, the ALAB is searched first for an entry matching the block’s address

- It can deadlock
Load Value Queue

- Originally: simple, ECC-protected FIFO
  - When loads commit store address and value in the LVQ
  - Trailing thread execute loads in program order
- Optimization:
  - Allow out-of-order load issue from the trailer thread
Output comparison

- Stores: must verify the address and data of every committed store before leaving the SoR
- Bottleneck if store queue is shared
- Separate per-thread store queues boost performance
- Cached load addresses: no need to compare them because they don’t modify the architectural state of the machine
- Uncached load addresses: they stall in the execute stage until the corresponding instruction in the other thread arrives, at which point the addresses can be compared
Want more performance?

- Slack fetch: maintains a constant slack of instructions between the threads
  - Prevents the trailing thread from seeing mispredictions and cache misses
- Branch Misprediction Queue: sends the outcomes of the committed branch outcomes to the trailing thread
  - Can be tricky to implement on a real processor
- Preferential Space Redundancy: if possible, try to execute instructions in different FUs
Some Results

SMT efficiency of an individual thread as the IPC of the thread when it would run in single-thread divided by the IPC of the thread when it would run in single thread mode in the same machine.

Figure 6. SMT-Efficiencies for one logical thread for four single-processor configurations. Base2 = base processor running one logical thread with two redundant copies, but no input replication or output comparison. SRT + ptSQ = SRT with per-thread store queue (with each store queue having 64 entries). SRT + noSC = SRT with no Store Comparison. “1.0” on the vertical axis represents the SMT-Efficiency of the base processor for one logical thread (running the same program) with no redundant copies.
What’s wrong with STR?

- A leading non-store instruction may commit before the check for the fault occurs
- Relies on the trailing thread to trigger the detection
- However, a STR processor works well in a fail-fast architecture
SRTR’s basic principles

- SRTR must not allow any leading instruction to commit before checking occurs
- SRTR exploits the time between the completion and commit time of leading instruction and checks the results as soon as the trailing completes
  - In SPEC95, complete to commit takes about 29 cycles
  - This short slack has some implications:
    - Leading thread provides branch predictions
    - The StB, LVQ and BOQ need to handle mispredictions
SRTR’s additions to SMT
**The ALs and the LVQ**

- Leading and trailing instructions occupy the same positions in their ALs
  - May enter their AL at different times and become ready to commit at different times
- The LVQ has to be modified to allow speculative loads
  - The Shadow Active List hold pointers to LVQ entries
  - It is possible that a trailing load is issued before the leading load
  - Branches place the LVQ tail pointer in the SAL
    - The LVQ’s tail pointer points to the LVQ has to be rolled back in a misprediction
Checking Instructions

- SRTR checks when the trailing instruction completes
- The Register Value Queue is used to store register values for checking, avoiding pressure on the register file
  - RVQ entries are allocated when instructions enter the AL
  - Pointers to the RVQ entries are placed in the SAL to facilitate their search
- If check succeeds, the entries in the CV vector are set to checked-ok
- If check fails, the entries in the CV vectors are set to failed
- Rollback done when entries in head of AL
DBCE

1 add r1 = r2 + r3
2 sub r20 = r21 - r22
3 shift r4 = r1, 2
4 sub r24 = r25 - r26
5 add r5 = r4 + 20

<table>
<thead>
<tr>
<th>reg</th>
<th>AL</th>
<th>chain tag</th>
<th>flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>120</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>104</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>124</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>105</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

DBQ entries showing only leading instructions

<table>
<thead>
<tr>
<th>not check</th>
<th>last-instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Check Table
Some Results

SRT vs SRTR with different slacks
Multiprocessors: Fault Detection
A High Level View of CRT
CRT’s Basic Principles

- Replicates register values but not memory values
- The leading thread commits stores only after checking
  - Memory is guaranteed to be correct
  - Other instructions commit without checking
- The leading thread sends committed values for branch outcomes, load values, store addresses and store values
  - Remember: the slack is big
- Uncached accesses are performed non-speculatively
Some Data

**Figure 4: CRT vs. SRTR.**

**Table 2: Benchmarks.**

<table>
<thead>
<tr>
<th>Benchmark set</th>
<th>set #</th>
<th>IPC pairing</th>
<th>2-CPU IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp+galgel</td>
<td>1</td>
<td>FP/FP low/high</td>
<td>2.20</td>
</tr>
<tr>
<td>fma3d+equake</td>
<td>2</td>
<td>FP/FP low/low</td>
<td>1.62</td>
</tr>
<tr>
<td>gap+sixtrack</td>
<td>3</td>
<td>Int/FP high/high</td>
<td>2.27</td>
</tr>
<tr>
<td>gcc+vortex</td>
<td>4</td>
<td>Int/Int low/low</td>
<td>1.46</td>
</tr>
<tr>
<td>gzip+parser</td>
<td>5</td>
<td>Int/Int high/low</td>
<td>1.91</td>
</tr>
<tr>
<td>perlbenchmark+swim</td>
<td>6</td>
<td>Int/FP low/low</td>
<td>1.89</td>
</tr>
<tr>
<td>twolf+mgrid</td>
<td>7</td>
<td>Int/FP high/high</td>
<td>2.45</td>
</tr>
<tr>
<td>vpr+crafty</td>
<td>8</td>
<td>Int/Int high/low</td>
<td>2.26</td>
</tr>
<tr>
<td>wupwise+mesa</td>
<td>9</td>
<td>FP/FP high/high</td>
<td>2.34</td>
</tr>
<tr>
<td>mcf+eon</td>
<td>10</td>
<td>Int/Int low/low</td>
<td>2.51</td>
</tr>
</tbody>
</table>
CRTR’s Basic Principles

- CRTR does not allow any trailing instruction to commit before it’s checked for faults, so that the register state of the trailing thread is used for recovery
- CRTR also communicates committed register values
Sending Leading Values

- Sending values at commit (in program order) eliminates the need for the values to be cleared up in mispredictions
  - We could avoid having a RVQ
  - But committed instructions don’t have values at commit, so need to access the register file in both threads
  - Which increases the pressure on the register files
- CRTR places values at writeback in the RVQs
  - The leading thread RVQ entries are sent to the trailing thread
  - Values are read from it at commit
Recovery

- The trailing thread preserves the faulting instruction PC so that execution can restart from that instruction
- The exception handler saves the trailing register state and PC to the CMP shared memory
- The leading CPU launches a restoring thread that copies it to the processor.
- The restoring copies the new state of the leading thread to the memory and compare both
Some Results

Table 2: Benchmarks.

<table>
<thead>
<tr>
<th>Benchmark set</th>
<th>set #</th>
<th>IPC pairing</th>
<th>2-CPU IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp+galgel</td>
<td>1</td>
<td>FP/FP low/high</td>
<td>2.20</td>
</tr>
<tr>
<td>fma3d+equake</td>
<td>2</td>
<td>FP/FP low/low</td>
<td>1.62</td>
</tr>
<tr>
<td>gap+sixtrack</td>
<td>3</td>
<td>Int/FP high/high</td>
<td>2.27</td>
</tr>
<tr>
<td>gcc+vortex</td>
<td>4</td>
<td>Int/Int low/low</td>
<td>1.46</td>
</tr>
<tr>
<td>gzip+parser</td>
<td>5</td>
<td>Int/Int high/low</td>
<td>1.91</td>
</tr>
<tr>
<td>perlbench+swim</td>
<td>6</td>
<td>Int/FP low/low</td>
<td>1.89</td>
</tr>
<tr>
<td>twolf+mgem</td>
<td>7</td>
<td>Int/FP high/high</td>
<td>2.45</td>
</tr>
<tr>
<td>vpr+crafty</td>
<td>8</td>
<td>Int/Int high/high</td>
<td>2.26</td>
</tr>
<tr>
<td>wupwise+mesa</td>
<td>9</td>
<td>FP/FP high/high</td>
<td>2.34</td>
</tr>
<tr>
<td>mcf+eon</td>
<td>10</td>
<td>Int/Int low/high</td>
<td>2.51</td>
</tr>
</tbody>
</table>