DeAliaser:
Alias Speculation Using Atomic Region Support

Wonsun Ahn*, Yuelu Duan, Josep Torrellas
University of Illinois at Urbana Champaign
http://iacoma.cs.illinois.edu
Memory Aliasing Prevents Good Code Generation

- Many popular compiler optimizations require code motion
  - Loop Invariant Code Motion (LICM): Body → Preheader
  - Redundancy elimination: Redundant expr. → First expr.

- Memory aliasing prevents code motion

- Problem: compiler alias analysis is notoriously difficult
Alias Speculation

• Compile time: optimize assuming certain alias relationships

• Run time: check those assumptions
  – Recover if assumptions are incorrect

• Enables further optimizations beyond what’s provable statically
Contribution: Repurpose Transactions for Alias Speculation

- Atomic Regions (a.k.a transactions) are here:
  - Intel TSX, AMD ASF, IBM Bluegene/Q, IBM Power
- HW for Atomic Regions performs:
  - Memory alias detection across threads
  - Buffering of speculative state
- DeAliaser: Repurpose it to detect aliasing within a thread as we move accesses
- How?
  - Cover the code motion span in an Atomic Region
  - Speculate that may-aliases in the span are no-aliases
  - Check speculated aliases using transactional HW
  - Recover from failure by rolling back transaction
Repurposing Transactional Hardware

<table>
<thead>
<tr>
<th>SR</th>
<th>SW</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Repurpose SR (Speculatively Read) bits to mark load locations that need monitoring due to code motion
  - Do not mark SR bits for regular loads inside the atomic region
  - Atomic region cannot be used for conventional TM
Repurposing Transactional Hardware

- Repurpose SR (Speculatively Read) bits to mark load locations that need monitoring due to code motion
  - Do not mark SR bits for regular loads inside the atomic region
  - Atomic region cannot be used for conventional TM
- SW (Speculatively Written) bits are still set by all the stores
  - Record all the transaction’s speculative data for rollback

<table>
<thead>
<tr>
<th>SR</th>
<th>SW</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Repurposing Transactional Hardware

- Repurpose SR (Speculatively Read) bits to mark load locations that need monitoring due to code motion
  - Do not mark SR bits for regular loads inside the atomic region
  - Atomic region cannot be used for conventional TM
- SW (Speculatively Written) bits are still set by all the stores
  - Record all the transaction’s speculative data for rollback
- Add ISA extensions to manipulate and check SR and SW bits
Instructions to Mark Atomic Regions

- `begin_atomic_opt PC / end_atomic_opt`
  - Starts / ends optimization atomic region
  - PC is the address of the Safe-Version of atomic region
    - Atomic region code without speculative optimizations
    - Execution jumps to Safe-Version after rollback

→ Same as regular atomic regions in TM systems except that SR bit marking by regular loads is turned off
Extensions to the ISA
(for Recording Monitored Locations)

- **load.r r1, addr**
  - Loads location *addr* to *r1* just like a regular load
  - Marks SR bit in cache line containing *addr*
  - Used for marking monitored loads

- **clear.r addr**
  - Clears SR bit in cache line containing *addr*
  - Used to mark end of load monitoring

→ Repurposing of SR bits allows selective monitoring of the loaded location between *load.r* and *clear.r*
→ Recall: all stored locations monitored until end of atomic region
Extensions to the ISA
(for Checking Monitored Locations)

• **storechk.(r/w/rw) r1, addr**
  • Stores $r1$ to location $addr$ just like a regular store
  • $r$: If SR bit is set $\rightarrow$ rollback
  • $w$: If SW bit is set $\rightarrow$ rollback
  • $rw$: If either SR or SW set $\rightarrow$ rollback

• **loadchk.(r/w/rw) r1, addr**
  • Loads $r1$ to location $addr$ just like a regular load
  • $r$: If SR bit is set $\rightarrow$ rollback
  • $w$: If SW bit is set $\rightarrow$ rollback
  • $rw$: If either SR or SW set $\rightarrow$ rollback
  • $r$, $rw$: set SR bit after checking
How are these Instructions Used?

• Four code motions are supported
  – Hoisting / sinking loads
  – Hoisting / sinking stores

• Some color coding before going into details
  – **Green**: moved instructions
  – **Red**: instructions “alias-checked” against moved instructions
  – **Orange**: instructions “alias-checked” against moved instructions unnecessarily (checks due to imprecision)
Code Motion 1: Hoisting Loads

begin_atomic_opt
store X
load A
end_atomic_opt

begin_atomic_opt
store X
load A
end_atomic_opt
Code Motion 1: Hoisting Loads

```
begin_atomic_opt
  store X
  load A
end_atomic_opt
```

```
begin_atomic_opt
  load A
  store X
end_atomic_opt
```
Code Motion 1: Hoisting Loads

1. Change \textit{load A} to \textit{load.r A} to set up monitoring of A
Code Motion 1: Hoisting Loads

1. Change `load A` to `load.r A` to set up monitoring of A
Code Motion 1: Hoisting Loads

1. Change \textit{load A} to \textit{load.r A} to set up monitoring of \( A \)
2. Change \textit{store X} to \textit{storechk.r X} to check monitor
Code Motion 1: Hoisting Loads

1. Change `load A` to `load.r A` to set up monitoring of `A`
2. Change `store X` to `storechk.r X` to check monitor
Code Motion 1: Hoisting Loads

1. Change \textit{load A} to \textit{load.r A} to set up monitoring of \textit{A}
2. Change \textit{store X} to \textit{storechk.r X} to check monitor
3. Insert \textit{clear.r A} to turn off monitoring at end of motion span
Code Motion 1: Hoisting Loads

1. Change \textit{load A} to \textit{load.r A} to set up monitoring of \( A \)
2. Change \textit{store X} to \textit{storechk.r X} to check monitor
3. Insert \textit{clear.r A} to turn off monitoring at end of motion span
1. Change \textit{load A} to \textit{load.r A} to set up monitoring of \textit{A}
2. Change \textit{store X} to \textit{storechk.r X} to check monitor
3. Insert \textit{clear.r A} to turn off monitoring at end of motion span
4. If overlapping monitor, \textit{loadchk.r A} is used instead of \textit{load.r A}
1. Change *load A* to *load.r A* to set up monitoring of A
2. Change *store X* to *storechk.r X* to check monitor
3. Insert *clear.r A* to turn off monitoring at end of motion span
4. If overlapping monitor, *loadchk.r A* is used instead of *load.r A*
1. Change \textit{load} \textit{A} to \textit{load.r} \textit{A} to set up monitoring of \textit{A} \\
2. Change \textit{store} \textit{X} to \textit{storechk.r} \textit{X} to check monitor \\
3. Insert \textit{clear.r} \textit{A} to turn off monitoring at end of motion span \\
4. If overlapping monitor, \textit{loadchk.r} \textit{A} is used instead of \textit{load.r} \textit{A} \\
   – Checks whether \textit{load.r} \textit{B} set up monitor in same cache line \\
   – Prevents \textit{clear.r} \textit{A} from clearing monitor set up by \textit{load.r} \textit{B}
Code Motion 1: Hoisting Loads

1. Change `load A` to `load.r A` to set up monitoring of `A`
2. Change `store X` to `storechk.r X` to check monitor
3. Insert `clear.r A` to turn off monitoring at end of motion span
4. If overlapping monitor, `loadchk.r A` is used instead of `load.r A`
   - Checks whether `load.r B` set up monitor in same cache line
   - Prevents `clear.r A` from clearing monitor set up by `load.r B`
Code Motion 2: Sinking Stores

```
begin_atomic_opt
load.r W
store X
store A
load Y
store Z
end_atomic_opt
```

```
begin_atomic_opt
load.r W
store X
store A
load Y
store Z
end_atomic_opt
```
Code Motion 2: Sinking Stores

begin_atomic_opt
load.r  W
store  X
store  A
load  Y
store  Z
end_atomic_opt

begin_atomic_opt
load.r  W
store  X
load  Y
store  Z
store  A
end_atomic_opt
1. Change \textit{store A} to \textit{storechk.rw A} to check preceding reads and writes
Code Motion 2: Sinking Stores

1. Change \textit{store A} to \textit{storechk.rw A} to check preceding reads and writes
Code Motion 2: Sinking Stores

begin_atomic_opt
load.r W
store X
store A
load Y
store Z
end_atomic_opt

begin_atomic_opt
load.r W
store X
load Y
store Z
storechk.rw A
end_atomic_opt

1. Change **store A** to **storechk.rw A** to check preceding reads and writes
2. Change **load Y** to **loadchk.r Y** to setup monitoring of Y
Code Motion 2: Sinking Stores

1. Change `store A` to `storechk.rw A` to check preceding reads and writes

2. Change `load Y` to `loadchk.r Y` to setup monitoring of Y
Code Motion 2: Sinking Stores

1. Change \textit{store} \textit{A} to \textit{storechk.rw A} to check preceding reads and writes
2. Change \textit{load} \textit{Y} to \textit{loadchk.r Y} to setup monitoring of \textit{Y}
3. Note \textit{store Z} is already monitored so no change is needed
1. Change `store A` to `storechk.rw A` to check preceding reads and writes
2. Change `load Y` to `loadchk.r Y` to setup monitoring of Y
3. Note `store Z` is already monitored so no change is needed
4. Note `load.r W` and `store X` are checked unnecessarily even if not in code motion span
Code Motion 2: Sinking Stores

1. Change \textit{store A} to \textit{storechk.rw A} to check preceding reads and writes
2. Change \textit{load Y} to \textit{loadchk.r Y} to setup monitoring of Y
3. Note \textit{store Z} is already monitored so no change is needed
4. Note \textit{load.r W} and \textit{store X} are checked unnecessarily even if not in code motion span

Alias check is \textbf{imprecise}

• Checks against all preceding stores and monitored loads
Code Motion 3: Sinking Clears

```
begin_atomic_opt
loadchk.r A
storechk.r X
**clear.r A**
store Y
storechk.r Z
end_atomic_opt

begin_atomic_opt
loadchk.r A
storechk.r X
**clear.r A**
store Y
storechk.r Z
end_atomic_opt
```
Code Motion 3: Sinking Clears

1. Sink `clear.r A` to the end of the atomic region
Code Motion 3: Sinking Clears

1. Sink `clear.r A` to the end of the atomic region
Code Motion 3: Sinking Clears

1. Sink `clear.r A` to the end of the atomic region
2. Trivially remove `clear.r A` at the end of atomic region
Code Motion 3: Sinking Clears

1. Sink \texttt{clear.r A} to the end of the atomic region
2. Trivially remove \texttt{clear.r A} at the end of atomic region
Code Motion 3: Sinking Clears

1. Sink `clear.r A` to the end of the atomic region
2. Trivially remove `clear.r A` at the end of the atomic region
3. Change `loadchk.r A` to `load.r A`
1. Sink \texttt{clear.r A} to the end of the atomic region.
2. Trivially remove \texttt{clear.r A} at the end of the atomic region.
3. Change \texttt{loadchk.r A} to \texttt{load.r A}.
Code Motion 3: Sinking Clears

1. Sink `clear.r A` to the end of the atomic region
2. Trivially remove `clear.r A` at the end of atomic region
3. Change `loadchk.r A` to `load.r A`
4. Note `storechk.r Z` may now trigger an unnecessary rollback
Code Motion 3: Sinking Clears

- Sinking *clears* can **reduce overhead** at the price of potentially increasing imprecision.
- *Clears* are the **only source** of instrumentation overhead (Besides *begin atomic* and *end atomic*)
  → Can perform alias checking with almost no overhead.
Illustrative Example: LICM and GVN

// a, b, *q may alias with *p
for (i = 0; i < 100; i++) {
    a = b + 10;
    *p = *q + 20;
    ... = *q + 20;
}

begin.atomic_opt PC

for (i = 0; i < 100; i++) {
    load r1, b
    r2 = r1 + 10
    store r2, a
    load r3, *q
    r4 = r3 + 20
    store r4, *p
    load r5, *q
    r6 = r5 + 20
    ...
}

end.atomic_opt

• Put atomic region around loop
• Perform optimizations after inserting appropriate checks
Illustrative Example: LICM and GVN

// a,b,*q may alias with *p
for(i=0; i < 100; i++) {
    a = b + 10;
    *p = *q + 20;
    ...
}

begin_atomic_opt PC
for(i=0; i < 100; i++) {
    load r1, b
    r2 = r1 + 10
    store r2, a
    load r3, *q
    r4 = r3 + 20
    store r4, *p
    load r5, *q
    r6 = r5 + 20
    ...
}
end_atomic_opt

- Put atomic region around loop
- Perform optimizations after inserting appropriate checks
  - Hoist b + 10 (LICM)
Illustrative Example: LICM and GVN

```c
// a, b, *q may alias with *p
for(i=0; i < 100; i++) {
    a = b + 10;
    *p = *q + 20;
    ... = *q + 20;
}
```

```c
begin_atomic_opt PC
load r1, b
r2 = r1 + 10
for(i=0; i < 100; i++) {
    store r2, a
    load r3, *q
    r4 = r3 + 20
    store r4, *p
    load r5, *q
    r6 = r5 + 20
    ...
}
end_atomic_opt
```

- Put atomic region around loop
- Perform optimizations after inserting appropriate checks
  - Hoist b + 10 (LICM)
Illustrative Example: LICM and GVN

```c
// a, b, *q may alias with *p
for(i=0; i < 100; i++) {
    a = b + 10;
    *p = *q + 20;
    ... = *q + 20;
}
```

- Put atomic region around loop
- Perform optimizations after inserting appropriate checks
  - Hoist b + 10 (LICM)

```c
begin_atomic_opt PC
load.r r1, b
r2 = r1 + 10
for(i=0; i < 100; i++) {
    store r2, a
    load r3, *q
    r4 = r3 + 20
    store r4, *p
    load r5, *q
    r6 = r5 + 20
    ...
}
end_atomic_opt
```
Illustrative Example: LICM and GVN

// a, b, *q may alias with *p
for(i=0; i < 100; i++) {
    a = b + 10;
    *p = *q + 20;
    ... = *q + 20;
}

begin_atomic_opt PC
load.r r1, b
r2 = r1 + 10
for(i=0; i < 100; i++) {
    store r2, a
    load r3, *q
    r4 = r3 + 20
    store r4, *p
    load r5, *q
    r6 = r5 + 20
    ...
}
clear.r b
end_atomic_opt

• Put atomic region around loop
• Perform optimizations after inserting appropriate checks
  – Hoist b + 10 (LICM)
Illustrative Example: LICM and GVN

// a,b,*q may alias with *p
for(i=0; i < 100; i++) {
    a = b + 10;
    *p = *q + 20;
    ... = *q + 20;
}

begin_atomic_opt PC
load.r r1, b
r2 = r1 + 10
for(i=0; i < 100; i++) {
    store r2, a
    load r3, *q
    r4 = r3 + 20
    load r5, *q
    r6 = r5 + 20
    ...
}
clear.r b
end_atomic_opt

• Put atomic region around loop
• Perform optimizations after inserting appropriate checks
  – Hoist b + 10 (LICM)
Illustrative Example: LICM and GVN

// a, b, *q may alias with *p
for(i=0; i < 100; i++) {
    a = b + 10;
    *p = *q + 20;
    ... = *q + 20;
}

begin_atomic_opt PC
load.r r1, b
r2 = r1 + 10
for(i=0; i < 100; i++) {
    store r2, a
    load r3, *q
    r4 = r3 + 20
    storechk.r r4, *p
    load r5, *q
    r6 = r5 + 20
    ...
}
clear.r b
end_atomic_opt

• Put atomic region around loop
• Perform optimizations after inserting appropriate checks
  – Hoist b + 10 (LICM)
  – Remove 2\textsuperscript{nd} *q + 20 (GVN)
Illustrative Example: LICM and GVN

// a, b, *q may alias with *p
for(i=0; i < 100; i++) {
    a = b + 10;
    *p = *q + 20;
    ... = *q + 20;
}

begin_atomic_opt PC
load.r r1, b
r2 = r1 + 10
for(i=0; i < 100; i++) {
    store r2, a
    loadchk.r r3, *q
    r4 = r3 + 20
    storechk.r r4, *p
    clear.r *q
    ...
}  
clear.r b
end_atomic_opt

• Put atomic region around loop
• Perform optimizations after inserting appropriate checks
  – Hoist b + 10 (LICM)
  – Remove 2\textsuperscript{nd} *q + 20 (GVN)
Illustrative Example: LICM and GVN

// a,b,*q may alias with *p
for(i=0; i < 100; i++) {
    a = b + 10;
    *p = *q + 20;
    ... = *q + 20;
}

begin_atomic_opt PC
load.r  r1, b
r2 = r1 + 10
for(i=0; i < 100; i++) {
    store  r2, a
    loadchk.r  r3, *q
    r4 = r3 + 20
    storechk.r  r4, *p
    clear.r *q
    ...
}
clear.r b
end_atomic_opt

- Put atomic region around loop
- Perform optimizations after inserting appropriate checks
  - Hoist b + 10 (LICM)
  - Remove 2\textsuperscript{nd} *q + 20 (GVN)
  - Sink / remove all clears
Illustrative Example: LICM and GVN

// a,b,*q may alias with *p
for(i=0; i < 100; i++) {
    a = b + 10;
    *p = *q + 20;
    ... = *q + 20;
}

begin_atomic_opt PC
load.r r1, b
r2 = r1 + 10
for(i=0; i < 100; i++) {
    store r2, a
    loadchk.r r3, *q
    r4 = r3 + 20
    storechk.r r4, *p
    ...
}
end_atomic_opt

- Put atomic region around loop
- Perform optimizations after inserting appropriate checks
  - Hoist b + 10 (LICM)
  - Remove 2nd *q + 20 (GVN)
  - Sink / remove all clears
Illustrative Example: LICM and GVN

// a, b, *q may alias with *p
for(i=0; i < 100; i++) {
    a = b + 10;
    *p = *q + 20;
    ... = *q + 20;
}

begin_atomic_opt PC
load.r  r1, b
r2 = r1 + 10
for(i=0; i < 100; i++) {
    store  r2, a
    loadchk.r  r3, *q
    r4 = r3 + 20
    storechk.r  r4, *p
    ...
}
end_atomic_opt

- Put atomic region around loop
- Perform optimizations after inserting appropriate checks
  - Hoist b + 10 (LICM)
  - Remove 2nd *q + 20 (GVN)
  - Sink / remove all clears
  - Sink store r2, a (LICM)
// a,b,*q may alias with *p
for(i=0; i < 100; i++) {
    a = b + 10;
    *p = *q + 20;
    ... = *q + 20;
}

begin_atomic_opt PC
load.r r1, b
r2 = r1 + 10
for(i=0; i < 100; i++) {
    loadchk.r r3, *q
    r4 = r3 + 20
    ...
}
storechk.w r2, a
end_atomic_opt

- Put atomic region around loop
- Perform optimizations after inserting appropriate checks
  - Hoist b + 10 (LICM)
  - Remove 2\textsuperscript{nd} *q + 20 (GVN)
  - Sink / remove all clears
  - Sink store r2, a (LICM)
Illustrative Example: LICM and GVN

• Loop body reduced from 8 instructions to 3 instructions
• With no alias check overhead
Issues

• Imprecision
  – Issue: Single set of SR & SW bits make checks imprecise
  – Solution: Could add more SR & SW bits to encode different code motion spans in different sets
    • Can be implemented efficiently using HW Bloom filters

• Isolation
  – Issue: Repurposing SR bits compromises isolation
  – Solution: Do not use the same atomic region for both alias speculation and TM
Compiler Toolchain

1. Performs loop blocking that uses memory footprint estimation

2. Wraps loops in atomic regions and create safe versions

3. Performs speculative optimizations using DeAliaser

4. Profiles binary to find out what the beneficial optimizations are according to a cost-benefit model

5. Disables unbeneﬁcial optimizations in the ﬁnal binary
Experimental Setup

• Compare three environments using LICM and GVN/PRE optimizations:
  – **BaselineAA:**
    • Unmodified LLVM-2.8 using *basic alias analysis*
    • Default alias analysis used by –O3 optimization
  – **DSAA:**
    • Unmodified LLVM-2.8 using *data structure alias analysis*
    • Experimental alias analysis with high time/space complexity
  – **DeAliaser:**
    • Modified LLVM-2.8 using DeAliaser to perform alias speculation

• Applications:
  – SPEC INT2006, SPEC FP2006

• Simulation:
  – SESC timing simulator with Atomic Region support
  – 32KB 8-way associative speculative L1 cache w/ 64B lines
Breakdown of Alias Analysis Results

- DeAliaser is able to convert almost all *may-aliases* to *no-aliases*
• DeAliaser speeds up SPEC INT by 2.5% and SPEC FP by 9%
Summary

• Proposed set of ISA extensions to expose Atomic Regions to SW for alias checking

• Performed hoisting / sinking of loads and stores
  – With minimal instrumentation overhead
  – Some imprecision due to HW limitations

• Evaluated using LICM and GVN/PRE
  – May-alias results: 56% → 4% SPEC INT, 43% → 1% SPEC FP
  – Speedup: 2.5% for SPEC INT, 9% for SPEC FP
Questions?
### Atomic Region Characterization

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Coverage(%)</th>
<th>BaselineAA</th>
<th>DSAA</th>
<th>LAS</th>
<th>ALAT</th>
<th>L1 Occ. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Insts</td>
<td>Reduc.(%)</td>
<td>Insts</td>
<td>Reduc.(%)</td>
<td>Insts</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>9</td>
<td>933</td>
<td>0</td>
<td>734</td>
<td>21</td>
<td>930</td>
</tr>
<tr>
<td>403.gcc</td>
<td>23</td>
<td>439</td>
<td>0</td>
<td>337</td>
<td>23</td>
<td>440</td>
</tr>
<tr>
<td>429.mcf</td>
<td>9</td>
<td>1228</td>
<td>0</td>
<td>1053</td>
<td>14</td>
<td>1207</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>12</td>
<td>1229</td>
<td>0</td>
<td>1082</td>
<td>12</td>
<td>1218</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>8</td>
<td>8580</td>
<td>1</td>
<td>6935</td>
<td>19</td>
<td>7787</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>69</td>
<td>1401</td>
<td>20</td>
<td>1124</td>
<td>20</td>
<td>1405</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>8</td>
<td>848</td>
<td>1</td>
<td>549</td>
<td>35</td>
<td>811</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>73</td>
<td>3185</td>
<td>0</td>
<td>3014</td>
<td>5</td>
<td>3028</td>
</tr>
<tr>
<td>473.astar</td>
<td>4</td>
<td>528</td>
<td>0</td>
<td>426</td>
<td>19</td>
<td>623</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>1</td>
<td>1171</td>
<td>0</td>
<td>1073</td>
<td>8</td>
<td>1171</td>
</tr>
<tr>
<td><strong>INT Average</strong></td>
<td>18</td>
<td>1702</td>
<td>2</td>
<td>1434</td>
<td>16</td>
<td>1628</td>
</tr>
<tr>
<td>410.bwaves</td>
<td>72</td>
<td>6261</td>
<td>0</td>
<td>5914</td>
<td>5</td>
<td>6101</td>
</tr>
<tr>
<td>433.milc</td>
<td>54</td>
<td>16027</td>
<td>18</td>
<td>11113</td>
<td>31</td>
<td>14620</td>
</tr>
<tr>
<td>434.zeusmp</td>
<td>5</td>
<td>9005</td>
<td>2</td>
<td>8415</td>
<td>7</td>
<td>8879</td>
</tr>
<tr>
<td>435.gromacs</td>
<td>3</td>
<td>2592</td>
<td>3</td>
<td>2257</td>
<td>13</td>
<td>2566</td>
</tr>
<tr>
<td>436.cactusADM</td>
<td>0</td>
<td>3072</td>
<td>0</td>
<td>1228</td>
<td>60</td>
<td>2282</td>
</tr>
<tr>
<td>437.leslie3d</td>
<td>58</td>
<td>5331</td>
<td>4</td>
<td>1853</td>
<td>65</td>
<td>3739</td>
</tr>
<tr>
<td>444.namd</td>
<td>38</td>
<td>24907</td>
<td>1</td>
<td>24574</td>
<td>1</td>
<td>24915</td>
</tr>
<tr>
<td>447.dealII</td>
<td>36</td>
<td>2713</td>
<td>4</td>
<td>1957</td>
<td>28</td>
<td>2362</td>
</tr>
<tr>
<td>450.soplex</td>
<td>25</td>
<td>1377</td>
<td>19</td>
<td>1104</td>
<td>20</td>
<td>1395</td>
</tr>
<tr>
<td>454.calcui x</td>
<td>0</td>
<td>1002</td>
<td>1</td>
<td>882</td>
<td>12</td>
<td>995</td>
</tr>
<tr>
<td>459.GemsFDTD</td>
<td>64</td>
<td>28270</td>
<td>29</td>
<td>15571</td>
<td>45</td>
<td>26776</td>
</tr>
<tr>
<td><strong>FP Average</strong></td>
<td>29</td>
<td>8379</td>
<td>12</td>
<td>6241</td>
<td>26</td>
<td>7885</td>
</tr>
</tbody>
</table>

- Low L1 cache occupancy due to not buffering speculatively read lines
- Overhead amortized over large atomic region
Speedups (SPECINT)

- Normalized against BaselineAA
- D = DSAA, A = Line-granularity DeAliaser, W = Word-granularity DeAliaser
Speedups (SPECFP)

- Normalized against BaselineAA
- D = DSAA, A = Line-granularity DeAliaser, W = Word-granularity DeAliaser
Commit Latency Sensitivity (SPECINT)

- Normalized against BaselineAA
- DeAliaser with A = 1-cycle commit, B = 10-cycle commit, C = 100-cycle commit
Commit Latency Sensitivity (SPECFP)

- Normalized against BaselineAA
- DeAliaser with A = 1-cycle commit, B = 10-cycle commit, C = 100-cycle commit
Rollback Overhead (SPECINT)

- Normalized against BaselineAA
- A = DeAliaser, G = Aggressive DeAliaser ignoring cost model
Rollback Overhead (SPECFP)

- Normalized against BaselineAA
- A = DeAliaser, G = Aggressive DeAliaser ignoring cost model
Dynamic Instruction Reduction (SPECINT)

- B = BaselineAA, D = DSAA, A = DeAliaser
Dynamic Instruction Reduction (SPECFP)

- B = BaselineAA, D = DSAA, A = DeAliaser
Alias Analysis Results (SPECINT)

- B = BaselineAA, D = DSAA, A = DeAliaser

![Bar chart showing percentage of must alias, no alias, and may alias for different benchmarks and tools.](image-url)
Alias Analysis Results (SPECFP)

- B = BaselineAA, D = DSAA, A = DeAliaser