Dynamic Optimization with Hardware Alias Detection

- Application Code
- Dynamic Optimization System
  Speculation on memory alias
- Optimizer
- Optimized Code in Atomic Regions
- Alias Detection
- Alias HW
- CPU
Dynamic Optimization with Hardware Alias Detection

Application Code

Dynamic Optimization System

Runtime

Trigger Re-Optimization

Optimizer

Optimized Code in Atomic Regions

Alias Exception

Roll Back

Alias Detection

CPU

Alias HW

Programming Systems Lab
Background 1: Hardware Alias Detection

\[ \text{unlikely alias} \begin{cases} M_0: \text{st} \ [r0] = ... \\ M_1: ... = \text{ld} \ [r1] \end{cases} \]
Background 1: Hardware Alias Detection

\[
\begin{align*}
\text{M}_1: & \quad \ldots = \text{ld} \ [r1] \quad \text{, set alias register AR0} \\
\text{M}_0: & \quad \text{st} \ [r0] = \ldots \quad \text{, check alias register AR0}
\end{align*}
\]
Background 1: Hardware Alias Detection

\[
\begin{align*}
\text{SW} & \quad M_1: \ldots = \text{ld} [r1], \text{set alias register AR0} \\
\text{HW} & \quad M_0: \text{st} [r0] = \ldots, \text{check alias register AR0} \\
\text{AR0} & \quad [r1, r1+3]
\end{align*}
\]
Background 1: Hardware Alias Detection

\[ M_1: \ldots = \text{ld} [r1], \text{set alias register AR0} \]

\[ M_0: \text{st} [r0] = \ldots, \text{check alias register AR0} \]

\[ \text{AR0} \]

\[ [r1, r1+3] \]

\[ \text{check overlap} \]

\[ [r0, r0+3] \]
Multiple Alias Check Issue

\[ M_0: \text{st} [r0+4] = \ldots \]
\[ M_1: \ldots = \text{id} [r1] \]
\[ M_2: \text{st} [r0] = \ldots \]
\[ M_3: \ldots = \text{id} [r2] \]
Multiple Alias Check Issue

\[
\begin{align*}
M_3 & : \ldots = \text{ld} [r2] & \text{, set AR0} \\
M_1 & : \ldots = \text{ld} [r1] & \text{, set AR1} \\
M_2 & : \text{st} [r0] = \ldots & \text{, check AR0} \\
M_0 & : \text{st} [r0+4] = \ldots & \text{, check AR0, AR1}
\end{align*}
\]
Multiple Alias Check Issue

unlikely alias

\[
\begin{align*}
M_3 &: \quad \ld [r2] \quad , \text{set AR0} \\
M_1 &: \quad \ld [r1] \quad , \text{set AR1} \\
M_2 &: \quad \st [r0] = \ld \ld [r0] \quad , \text{check AR0} \\
M_0 &: \quad \st [r0+4] = \ld \ld [r0+4] \quad , \text{check AR0, AR1}
\end{align*}
\]

- A memory operation (i.e. \(M_0\)) may need to check multiple alias registers (i.e. AR0, AR1)
Multiple Alias Check Issue

\[
\begin{align*}
M_3: \ldots &= \text{ld} \ [r2] \quad , \text{set AR0} \\
M_1: \ldots &= \text{ld} \ [r1] \quad , \text{set AR1} \\
M_2: \text{st} \ [r0] &= \ldots \quad , \text{check AR0} \\
M_0: \text{st} \ [r0+4] &= \ldots \quad , \text{check AR0, AR1}
\end{align*}
\]

- A memory operation (i.e. $M_0$) may need to check multiple alias registers (i.e. AR0, AR1)
- Transmeta Efficeon uses a bitmask in the instruction to specify the individual alias registers to be checked
  - cannot scale up to a large number of alias registers
Multiple Alias Check Issue

\[
\begin{align*}
M_3 & : \ldots = \text{ld [r2]} \quad , \text{set AR0} \\
M_1 & : \ldots = \text{ld [r1]} \quad , \text{set AR1} \\
M_2 & : \text{st [r0]} = \ldots \quad , \text{check AR0} \\
M_0 & : \text{st [r0+4]} = \ldots \quad , \text{check AR0, AR1}
\end{align*}
\]

- A memory operation (i.e. \(M_0\)) may need to check multiple alias registers (i.e. AR0, AR1)

- Transmeta Efficeon uses a \textit{bitmask} in the instruction to specify the individual alias registers to be checked
  - cannot scale up to a large number of alias registers

- Itanium Advanced Load Address Table (ALAT) checks \textit{all alias registers} set by previous advanced loads
  - may lead to false positive (i.e. \(M_2\) checks \(M_1\))
  - can not check alias between stores
Background 2: Order-Based Alias Detection

\[ M_0: \text{st } [r0+4] = \ldots \]
\[ M_1: \ldots = \text{ld } [r1] \]
\[ M_2: \text{st } [r0] = \ldots \]
\[ M_3: \ldots = \text{ld } [r2] \]
Background 2: Order-Based Alias Detection

\[ M_3: \ldots = \text{ld [r2]} \quad , \text{order 3} \]
\[ M_1: \ldots = \text{ld [r1]} \quad , \text{order 1} \]
\[ M_2: \text{st [r0]} = \ldots \quad , \text{order 2} \]
\[ M_0: \text{st [r0+4]} = \ldots \quad , \text{order 0} \]
Background 2: Order-Based Alias Detection

M₃: ... = ld [r2] , order 3   // set AR3
M₁: ... = ld [r1] , order 1   // set AR1
M₂: st [r0] = ... , order 2   // set AR2, check AR3
M₀: st [r0+4] = ... , order 0

<table>
<thead>
<tr>
<th>AR0</th>
<th>AR1</th>
<th>AR2</th>
<th>AR3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>[r1, r1+3]</td>
<td>[r0, r0+3]</td>
<td>[r2, r2+3]</td>
<td></td>
</tr>
</tbody>
</table>

check overlap
Background 2: Order-Based Alias Detection

\[ M_3: \ldots = \text{ld} [r2], \text{order } 3 \quad \text{// set AR3} \]
\[ M_1: \ldots = \text{ld} [r1], \text{order } 1 \quad \text{// set AR1} \]
\[ M_2: \text{st} [r0] = \ldots, \text{order } 2 \quad \text{// set AR2, check AR3} \]
\[ M_0: \text{st} [r0+4] = \ldots, \text{order } 0 \quad \text{// set AR0, check AR1, AR2, AR3} \]

<table>
<thead>
<tr>
<th>queue</th>
<th>AR0</th>
<th>AR1</th>
<th>AR2</th>
<th>AR3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[r0+4, r0+7]</td>
<td>[r1, r1+3]</td>
<td>[r0, r0+3]</td>
<td>[r2, r2+3]</td>
<td></td>
</tr>
</tbody>
</table>

check overlap
Background 2: Order-Based Alias Detection

\[ M_3: \ldots = \text{ld} [r2], \text{order 3} \]  // set AR3
\[ M_1: \ldots = \text{ld} [r1], \text{order 1} \]  // set AR1
\[ M_2: \text{st} [r0] = \ldots, \text{order 2} \]  // set AR2, check AR3
\[ M_0: \text{st} [r0+4] = \ldots, \text{order 0} \]  // set AR0, check AR1, AR2, AR3

<table>
<thead>
<tr>
<th>AR0 queue</th>
<th>AR1 queue</th>
<th>AR2 queue</th>
<th>AR3 queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>[r0+4, r0+7]</td>
<td>[r1, r1+3]</td>
<td>[r0, r0+3]</td>
<td>[r2, r2+3]</td>
</tr>
</tbody>
</table>

- Adopted in out-of-order CPU (ld/st queue) for HW speculation
- Detect all the aliases between reordered memory operations without any false positive
  - HW automatically identifies loads to prevent alias check between them (E.g. \( M_1 \) does not check AR3)
Order-Based Alias Detection Issues

\[ M_3: \ldots = \text{ld} [r2], \text{order 3} \] // set AR3
\[ M_1: \ldots = \text{ld} [r1], \text{order 1} \] // set AR1
\[ M_2: \text{st} [r0] = \ldots, \text{order 2} \] // set AR2, check AR3
\[ M_0: \text{st} [r0+4] = \ldots, \text{order 0} \] // set AR0, check AR1, AR2, AR3

- Do not leverage compiler analysis for efficient alias register usage
  - 4 alias registers are used
  - \( M_0 \) does not need to check \( M_2 \)
Order-Based Alias Detection Issues

M₃: ... = ld [r2] , order 3 // set AR3
M₁: ... = ld [r1] , order 1 // set AR1
M₂: st [r0] = ... , order 2 // set AR2, check AR3
M₀: st [r0+4] = ... , order 0 // set AR0, check AR1, AR2, AR3

• Do not leverage compiler analysis for efficient alias register usage
  - 4 alias registers are used
  - M₀ does not need to check M₂

• Only consider memory reordering, but not general optimizations such as load/store elimination
  - May need alias check between non-reordered memory operations
Check Between Non-Reordered Memory Operations

\[ \begin{align*}
M_0 &: r3 = ld [r0] \\
M_1 &: \ldots = ld [r1] \\
M_2 &: st [r2] = \ldots \\
M_3 &: r4 = ld [r0] \\
\end{align*} \]

Load elimination

\[ \begin{align*}
M_0 &: r3 = ld [r0] \\
M_1 &: \ldots = ld [r1] \\
M_2 &: st [r2] = \ldots \\
M_3 &: r4 = r3 \\
\end{align*} \]

Store elimination

\[ \begin{align*}
M_0 &: \text{st} [r0] = r3 \\
M_1 &: \ldots = ld [r1] \\
M_2 &: \text{st} [r2] = \ldots \\
M_3 &: \text{st} [r0] = r4 \\
\end{align*} \]
Motivation

SW-Managed Alias Register File
- SW Speculation on In-Order CPU
- Itanium - False Positive Issue
- Transmeta - Scalability Issue

HW-Managed Alias Register Queue
- HW Speculation on Out-of-Order CPU
- Efficiency Issue
- Generalization Issue

SMARQ: SW-Managed Alias Register Queue
- Novel architecture features and compiler algorithms
- Solve all the issues in previous works
SMARQ Example

$M_0$: st [r0+4] = ...  
$M_1$: ... = ld [r1]  
$M_2$: st [r0] = ...  
$M_3$: ... = ld [r2]$
SMARQ Example

- Compiler analysis to derive necessary alias checking for the optimization
  - Check-constraint $A \rightarrow B$: $A$ needs to check $B$ for alias
SMARQ Example

M₃: ... = ld [r2], P
M₁: ... = ld [r1], P
M₂: st [r0] = ..., C
M₀: st [r0+4] = ..., C

- Compiler analysis to derive necessary alias checking for the optimization
  - Check-constraint A → B: A needs to check B for alias

- Leverage architecture features for efficient management of the alias register queue
  - P bit for setting alias register and a C bit for checking alias registers (may have both)
SMARQ Example

\[
\begin{align*}
M_3: & \ldots = \text{ld} [r2], P, \text{order 1} \\
M_1: & \ldots = \text{ld} [r1], P, \text{order 0} \\
M_2: & \text{st} [r0] = \ldots, C, \text{order 1} \\
M_0: & \text{st} [r0+4] = \ldots, C, \text{order 0}
\end{align*}
\]

- Compiler analysis to derive necessary alias checking for the optimization
  - Check-constraint \( A \rightarrow B \): \( A \) needs to check \( B \) for alias

- Leverage architecture features for efficient management of the alias register queue
  - \( P \) bit for setting alias register and a \( C \) bit for checking alias registers (may have both)

- Alias register order allocation respecting check-constraints
  - \( A \rightarrow B \Rightarrow \text{order}(A) \leq \text{order}(B) \)
  - The alias register order for different instructions with \( P \) bit must be different
SMARQ Example

M₃: ... = ld [r2], P, order 1  // set AR1
M₁: ... = ld [r1], P, order 0  // set AR0
M₂: st [r0] = ... , C, order 1  // check AR1
M₀: st [r0+4] = ... , C, order 0

HW

queue
[ r1, r1+3 ] [ r2, r2+3 ] [ r0, r0+3 ]

- Compiler analysis to derive necessary alias checking for the optimization
  - Check-constraint A \(\rightarrow\) B: A needs to check B for alias

- Leverage architecture features for efficient management of the alias register queue
  - P bit for setting alias register and a C bit for checking alias registers (may have both)

- Alias register order allocation respecting check-constraints
  - A \(\rightarrow\) B \(\Rightarrow\) order(A) \(\leq\) order(B)
  - The alias register order for different instructions with P bit must be different

Programming Systems Lab
SMARQ Example

\[ M_3: \ldots = \text{ld} [r2] , P , \text{order 1} \]  // set AR1
\[ M_1: \ldots = \text{ld} [r1] , P , \text{order 0} \]  // set AR0
\[ M_2: \text{st} [r0] = \ldots , C , \text{order 1} \]  // check AR1
\[ M_0: \text{st} [r0+4] = \ldots , C , \text{order 0} \]  // check AR0, AR1

- Compiler analysis to derive necessary alias checking for the optimization
  - Check-constraint A → B: A needs to check B for alias

- Leverage architecture features for efficient management of the alias register queue
  - P bit for setting alias register and a C bit for checking alias registers (may have both)

- Alias register order allocation respecting check-constraints
  - A → B ⇒ order(A) ≤ order(B)
  - The alias register order for different instructions with P bit must be different
Prevent False Positive

\[
\begin{align*}
M_3 &: \ldots = \text{ld} [r2] \quad P, \quad \text{order 1} \quad // \text{set AR1} \\
M_1 &: \ldots = \text{ld} [r1] \quad P, \quad \text{order 0} \quad // \text{set AR0} \\
M_2 &: \text{st} [r0] = \ldots \quad C, \quad \text{order 0} \quad // \text{check AR0, AR1} \\
M_0 &: \text{st} [r0+4] = \ldots \quad C, \quad \text{order 0} \quad // \text{check AR0, AR1}
\end{align*}
\]

- Alias register order respecting check-constraint may cause false positive in alias checking
  - No restriction on the alias register order between \( M_1 \) and \( M_2 \) and Incorrect order between them will cause \( M_2 \) to check \( M_1 \).
Prevent False Positive

\[
\begin{align*}
M_3: \ldots &= \text{ld [r2]} & P & \text{, order 1} & // \text{set AR1} \\
M_1: \ldots &= \text{ld [r1]} & P & \text{, order 0} & // \text{set AR0} \\
M_2: \text{st [r0]} &= \ldots & C & \text{, order 1} & // \text{check AR1} \\
M_0: \text{st [r0+4]} &= \ldots & C & \text{, order 0} & // \text{check AR0, AR1}
\end{align*}
\]

- Alias register order respecting check-constraint may cause false positive in alias checking
  - No restriction on the alias register order between $M_1$ and $M_2$ and Incorrect order between them will cause $M_2$ to check $M_1$

- Anti-constraint $A \rightarrow B$: $A$ should not be checked by $B$
  - $A \rightarrow B \Rightarrow \text{order}(A) < \text{order}(B)$
Advanced SMARQ Features

• Alias register rotation*
  – HW organize the alias registers as a circular queue rotated through a base alias register pointer
  – SW rotate the alias register queue to release dead alias register, just like the release of ld/st queue entry in out-of-order CPU

• Handle order cycles*
  – The alias register order respecting all the check-constraints and anti-constraints may contain cycles
    – No alias register order can detect all the aliases without any false positive
  – New architecture alias-register-move feature to break all the cycles

• Prevent alias register overflow*
  – It is hard to support alias register spill
  – Allocate alias register during list scheduling so that if alias register may overflow, schedule instruction to respect all aliases without using new alias registers

* See details in the paper
Experiment Setup

- dynamic optimization framework translates and optimizes x86 binary codes into code running on an internal VLIW CPU modeled by a cycle-accurate

<table>
<thead>
<tr>
<th>Architecture Features</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-wide VLIW</td>
<td>2 INT units, 2 FP units, 2 MEM unit, 1 BRANCH unit, 1 ALIAS unit</td>
</tr>
<tr>
<td>L1 I-Cache</td>
<td>4-way 256KB</td>
</tr>
<tr>
<td>L1 D-Cache</td>
<td>4-way 64KB, HW prefetch</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>8-way 2MB, 8 cycle latency, HW perfetch</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>8-way 8MB, 25 cycle latency</td>
</tr>
<tr>
<td>Memory</td>
<td>1GB, 104 cycle latency</td>
</tr>
<tr>
<td>Alias Register Queue</td>
<td>64 entries</td>
</tr>
</tbody>
</table>
Average Region Size

Memory Operations

ammp  applu  apsi  equake  facerec  fma3d  galgel  lucas  mesa  swim  wupwise
Speedup with Different Alias Detection

- Baseline: no HW alias detection
- SMARQ: 39% speedup on average
- SMARQ16: restrict alias register queue to 16 entries
  - 29% speedup
- Itanium-like approach: non ordered alias detection (with false positives)
  - 26% speedup
Speedup with Different Alias Detection

- Baseline: no HW alias detection
- SMARQ: 39% speedup on average
- SMARQ16: restrict alias register queue to 16 entries
  - 29% speedup
- Itanium-like approach: non ordered alias detection (with false positives)
  - 26% speedup
Alias Register Working Set

Average Working Set Size

- Baseline: allocate each memory operation a unique alias register in their program order
- SMARQ: reduce the average alias register working set by 74%.
- Lower bound: maximum number of overlapped alias register live-ranges
  - SMARQ achieves alias register working set size close to the lower bound.
Conclusions

• We identify in details the issues in all the previous works on alias registers

• We proposed SMARQ, a software managed alias register queue to solve all the issues in previous works
  – improve the overall performance by 39% as compared to the optimization without alias register
  – reduce the alias register working set by 74%