Allocating Rotating Registers by Scheduling

Hongbo Rong              Hyunchul Park
Cheng Wang               Youfeng Wu

Intel Labs

MICRO-46, Dec. 2013
Memory disambiguation in dynamic compilers

• Dynamic languages are increasing popular
  – JavaScript 88.9% in client-side websites (W3Techs)
  – PhP 81.5% in sever side (W3Techs)

• Huge amount of legacy code
  – Dynamic binary translator
  – Optimization scope tends to be small: For a loop, usually a loop iteration
    • Software pipelining: enlarge the scope
  – Memory aliases are a bottleneck to performance
Memory disambiguation in dynamic compilers (Cont.)

• A fundamental component in compilers
  — Determines if two memory operations are aliased

• Approach 1: Alias analysis at compile time
  — Too expensive or conservative for dynamic compilers
    — Only simple alias analysis can be done

• Approach 2: Alias detection at runtime with hardware support
Alias Registers

• Enable data speculation
  – Compiler optimistically assumes the memory operations do not alias with each other, and schedules them out of order
  – Compiler guards every memory operation with an alias register to catch any alias when the schedule runs
Alias Registers

• Enable data speculation
  – Compiler optimistically assumes the memory operations do not alias with each other, and schedules them out of order
  – Compiler guards every memory operation with an alias register to catch any alias when the schedule runs

• Example:
Alias Registers

• Enable data speculation
  – Compiler optimistically assumes the memory operations do not alias with each other, and schedules them out of order
  – Compiler guards every memory operation with an alias register to catch any alias when the schedule runs

• Example:

Original order

LD [w]
ST [y]
ST [z]
Alias Registers

• Enable data speculation
  – Compiler optimistically assumes the memory operations do not alias with each other, and schedules them out of order
  – Compiler guards every memory operation with an alias register to catch any alias when the schedule runs

• Example:

Original order
LD [w]
ST [y]
ST [z]

After speculation
ST [y]
ST [z]
LD [w]
Reordered!
Static Alias Registers
Static Alias Registers

- SR0
- SR1
- SR2
- SR3
Static Alias Registers

SR0

SR1

SR2

SR3

set

check
Static Alias Registers

<table>
<thead>
<tr>
<th>Mem addr</th>
<th>Set</th>
<th>Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>$x$</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SR1</td>
</tr>
</tbody>
</table>

- set
- check
Static Alias Registers

SR0

SR1

SR2

SR3

ST x ... SR1

Mem addr

Set

Check

set

check
Static Alias Registers

SR0

SR1

SR2

SR3

Mem addr | Set | Check

ST x ... SR1

ST y ... SR2
Static Alias Registers

SR0  

SR1  

SR2  

SR3  

Mem addr  Set  Check

ST  x  ...  SR1

ST  y  ...  SR2

set

check
Static Alias Registers

SR0

SR1

SR2

SR3

Mem addr  Set  Check

ST  x  ...  SR1

ST  y  ...  SR2

LD  z  ...  SR1,2
Static Alias Registers

SR0

Mem addr

ST  x  ...

Set  SR1

SR1  x

Check

SR2  y

ST  y  ...

Set  SR2

SR3

LD  z  ...

Set  SR1,2
Static alias registers are not enough

- 24 hot loops from SPEC2000, with static alias registers only, loop throughput is far below optimal
- Need more scalable hardware

Experiments on Transmeta Efficeon with 14 static alias regs

Throughput is 11~106% below optimal
Rotating Alias Registers

- RR0
- RR1
- RR2
- RR3
Rotating Alias Registers

<table>
<thead>
<tr>
<th>Mem addr</th>
<th>Set</th>
<th>Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>x</td>
<td>...</td>
</tr>
</tbody>
</table>
Rotating Alias Registers

RR0 ➔ RR1 ➔ RR2 ➔ RR3

Mem addr  Set  Check
ST  x  ...  RR1
Rotating Alias Registers

RR0 → RR1 → RR2 → RR3

Mem addr
Set
Check

ST x ...
RR1

ST y ...
RR2
Rotating Alias Registers

RR0 -> RR1 -> RR2 -> RR3

Mem addr: ST x..., ST y...
Set: RR1, RR2
Check:

set
check
Rotating Alias Registers

RR0 → RR1 → RR2 → RR3

Mem addr | Set | Check
ST x ... RR1
ST y ... RR2
LD z ... RR1
Rotating Alias Registers

- **RR0**: x
- **RR1**: y
- **RR2**:
- **RR3**:

<table>
<thead>
<tr>
<th>Mem addr</th>
<th>Set</th>
<th>Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST x</td>
<td>...</td>
<td>RR1</td>
</tr>
<tr>
<td>ST y</td>
<td>...</td>
<td>RR2</td>
</tr>
<tr>
<td>LD z</td>
<td>...</td>
<td>RR1</td>
</tr>
</tbody>
</table>
Rotating Alias Registers

```
set  check

RR0  
RR1  x
RR2  y
RR3  

Unidirectional check

Mem addr  Set  Check

ST x ... RR1
ST y ... RR2
LD z ... RR1
```
Rotating Alias Registers

- Encode 1 register, check many: scalable

Unidirectional check
False positives

```
RR0

RR1 x

RR2 y

RR3

Mem addr  Set  Check
ST x ... RR1
ST y ... RR2
LD z ... RR1
```
False positives

An earlier unrelated OP:

ST  z  ...  RR3

ST  x  ...  RR1

ST  y  ...  RR2

LD  z  ...  RR1

Mem addr  Set  Check

set  check
False positives

An earlier unrelated OP:

- ST z ... RR3
- Mem addr
- Set
- Check
- ST x ... RR1
- ST y ... RR2
- LD z ... RR1
False positives

- Avoid by a good allocation
- Resort to static alias registers

An earlier unrelated OP:

- ST | z | ... | RR3
- ST | x | ... | RR1
- ST | y | ... | RR2
- LD | z | ... | RR1

Set

Check
False positives

- Avoid by a good allocation
- Resort to static alias registers

An earlier unrelated OP:

```
Mem addr  Set  Check
ST  x  ...  RR1
ST  y  ...  RR2
LD  z  ...  RR1
```
False positives

- Avoid by a good allocation
- Resort to static alias registers

An earlier unrelated OP:

ST z ... SR0
Mem addr Set Check
ST x ... RR1
ST y ... RR2
LD z ... RR1
Rotating alias registers (Cont.)

• Comparison:
  – ALAT in Itanium does not detect aliases between stores
  – DeAliaser [Ahn, Duan, Torrellas 2013] can only check all speculative stores

• Effective for sequential code [Wang et al. 2012]

• A new problem: How to apply them to loops?
Software Pipelining

• A loop with 3 operations in order

a b c
Software Pipelining

• A loop with 3 operations in order

Sequential execution
Software Pipelining

- A loop with 3 operations in order

Sequential execution

0 3 6 9

Time

a b c
Software Pipelining

• A loop with 3 operations in order

Sequential execution

0 3 6 9

Time

a b c

a b c

a b c
Software Pipelining

• A loop with 3 operations in order

Sequential execution

0  3  6  9  Time
a b c  a b c  a b c
Software Pipelining

• A loop with 3 operations in order
  \[a \ b \ c\]
  Sequential execution

  \[\begin{array}{cccc}
  0 & 3 & 6 & 9 \\
  a & b & c & a \ b \ c \ a \ b \ c \\
\end{array}\]

• To speedup
  – Schedule operations out of order
    Overlap the execution of iterations
Software Pipelining (Cont.)

• Software-pipelined execution

0  3  6  9

Time

1st Itr

a  b  c
Software Pipelining (Cont.)

- Software-pipelined execution

\[
\begin{align*}
0 & \quad 3 & \quad 6 & \quad 9 & \quad \text{Time} \\
\text{c} & \quad \text{b} & \quad \text{a} & \quad \text{1}\text{st Itr}
\end{align*}
\]
Software Pipelining (Cont.)

• Software-pipelined execution

0 3 6 9

Time

1st Itr

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>b</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

2nd Itr

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>b</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>
Software Pipelining (Cont.)

• Software-pipelined execution

0  3  6  9

Time

1st Itr

2nd Itr

3rd Itr
Software Pipelining (Cont.)

- Software-pipelined execution

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Itr</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>II=3</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2nd Itr</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>3rd Itr</td>
<td>c</td>
<td>b</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>
Software Pipelining (Cont.)

• Software-pipelined execution

0  3  6  9

\[ c \rightarrow b \rightarrow a \]

\[ \text{Time} \]

1\text{st} Itr

II=3

\[ c \rightarrow b \rightarrow a \]

2\text{nd} Itr

\[ c \rightarrow b \rightarrow a \]

3\text{rd} Itr

• Modulo scheduling
  • Modulo property
Software Pipelining (Cont.)

- Software-pipelined execution

```
   c   b   a

0  3  6  9
```

1\text{st Itr}

II=3

```
   c   b   a

   c   b   a
```

2\text{nd Itr}

```
   c   b   a

   c   b   a
```

3\text{rd Itr}

- Modulo scheduling
  - Modulo property
  - Dependence constraints
Software Pipelining (Cont.)

• Software-pipelined execution

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1\textsuperscript{st} Itr

\[ II=3 \]

\[ \text{c} \quad \text{b} \quad \text{a} \]

2\textsuperscript{nd} Itr

3\textsuperscript{rd} Itr

• Modulo scheduling
  • Modulo property
  • Dependence constraints
  • Resource constraints
Objective

• Given a software-pipelined schedule of a loop, how to allocate rotating alias registers for the memory operations?
  – Detect ALL aliases
  – NO false positive
  – Minimal usage of rotating & static alias registers
Rotating Register Allocation ≡ Scheduling!

• It is a software pipelining problem
  – A register allocation is a modulo schedule of lifetimes
  – Any existing modulo scheduling algorithm can solve it

• Contributions
  – Framework
  – A simple algorithm LCP
  – LCP usually achieves the best allocations in the least time
  – Generalization: allocation of general-purpose rotating registers is also a software pipelining problem
    • It derives bin-packing of Rau et al. 1992
Software Pipelining Reviewed

- Software-pipelined execution

<table>
<thead>
<tr>
<th>Time</th>
<th>1st Itr</th>
<th>2nd Itr</th>
<th>3rd Itr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

0 3 6 9 → Time
Software Pipelining Reviewed

• Software-pipelined execution

0 3 6 9

Time

1st Itr

c ← b a

2nd Itr

c b a

3rd Itr

c b a

Check
Software Pipelining Reviewed

• Software-pipelined execution

0 3 6 9

Time

1st Itr

0 3 6 9

Time

2nd Itr

1st Itr

3rd Itr

Check
Software Pipelining Reviewed

- Software-pipelined execution

0  3  6  9  \(\rightarrow\) Time

\(c\) \(\leftarrow\) \(b\) \(\leftarrow\) \(a\)  \(1^{st}\) Itr

\(c\) \(\leftarrow\) \(b\) \(\leftarrow\) \(a\)  \(2^{nd}\) Itr

\(c\) \(\leftarrow\) \(b\) \(\leftarrow\) \(a\)  \(3^{rd}\) Itr

\(\leftarrow\) \(\leftarrow\) Check
A Register Allocation

<table>
<thead>
<tr>
<th>Time</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

- **1st Itr**: Registers 0 and 1 are allocated.
- **2nd Itr**: Registers 2 and 3 are allocated.
- **3rd Itr**: Register 5 is allocated.
A Register Allocation

1st Itr

2nd Itr

3rd Itr

Register

Time

0 3 6 9

0 1 2 3 4 5

Check

b a

c

b a

c

b a

c

b a

c
A Register Allocation

Time

1st Itr

2nd Itr

3rd Itr

Register

RegII=2

Check

b a

c

b a

c

b a

c
A Register Allocation

Time

1st Itr

RegII=2

2nd Itr

Check

3rd Itr

“Time”
A Register Allocation

"Time"

"Resource"

1st Itr

2nd Itr

3rd Itr

RegII=2
A Register Allocation

“Time”

Modulo schedule of lifetimes!
Problem formulation

View Lifetimes as “Operations”
Registers as “Time”
Time as “Resources”

Then the allocation is a modulo schedule:

- Modulo property: A constant initiation interval $R$
  $r(a, i + 1) = r(a, i) + R, \ \forall i$
- Dependence constraints
  - The ordering requirement between lifetimes
    $r(a, i) \leq r(b, i + d), \ \forall i$
- Resource constraints
  - Two lifetimes in the same register cannot overlap in time
Unifying Dependence and Resource Constraints

\[
DIST(a, b) = DIST_{dep}(a, b) \bigcap DIST_{res}(a, b)
\]

\[
DIST_{dep}(a, b) = \bigcap_{\forall \text{dependence } (a \rightarrow b, \delta, d)} [\delta - d \times R, +\infty) \bigcap \bigcap_{\forall \text{dependence } (b \rightarrow a, \delta, d)} (-\infty, -\delta + d \times R]
\]

\[
DIST_{res}(a, b) = \begin{cases} 
DIST_{res\_1}(a, b) & \text{if } a \text{ or } b \text{ is a pure checker} \\
DIST_{res\_2}(a, b) \cup DIST_{res\_3}(a, b) & \text{otherwise}
\end{cases}
\]

\[
DIST_{res\_1}(a, b) = (-\infty, +\infty)
\]

\[
DIST_{res\_2}(a, b) = (-\infty, +\infty) \setminus R \times (-\infty, +\infty)
\]

\[
DIST_{res\_3}(a, b) = R \times (-\infty, +\infty) \bigcap \left\{ [-\left\lfloor \frac{\text{start}(a) - \text{end}(b)}{II} \right\rfloor \times R, +\infty) \bigcup (-\infty, -\left\lfloor \frac{\text{end}(a) - \text{start}(b)}{II} \right\rfloor \times R] \right\}
\]
LCP (Local Compaction followed by Pacing)
LCP (Local Compaction followed by Pacing)

- Two heuristics to reduce false positives
  - Max R and Earliest start
Overall flow
Overall flow

Loop

Modulo scheduling (operations)

JITSP(CGO’14)
Overall flow

Loop

Modulo scheduling (operations)

Rotating alias register allocation

JITSP (CGO’14)

JITSP, LCP, RS2, DESP, Ideal
Overall flow

Loop

- Modulo scheduling (operations) [JITSP (CGO’14)]
- Rotating alias register allocation [JITSP, LCP, RS2, DESP, Ideal]
- Post-processing false positives
Overall flow

Loop

- Modulo scheduling (operations)
- Rotating alias register allocation
- Post-processing false positives
- Code generation

JITSP(CGO’14)

JITSP, LCP, RS2, DESP, Ideal
Overall flow

Loop

Modulo scheduling (operations)

Rotating alias register allocation

Post-processing false positives

Code generation

- Implemented in Transmeta Code Morphing Software (CMS)
- simulated with a variable-size rotating alias register file
• 11,825 software-pipelined loops from SPEC2000 dynamic traces
#false positives and allocation time

- LCP has 0.16 false positives per loop iteration
- LCP takes 2.46% translation time, roughly about 0.07% total time
- #false positives and allocation time relative to LCP

<table>
<thead>
<tr>
<th></th>
<th>#false positive</th>
<th>Allocation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS2</td>
<td>19X</td>
<td>4X</td>
</tr>
<tr>
<td>DESP</td>
<td>12X</td>
<td>1.6X</td>
</tr>
<tr>
<td>JITSP</td>
<td>14X</td>
<td>1.7X</td>
</tr>
</tbody>
</table>
Summary

• Rotating alias registers detect memory aliases at runtime
  – Relatively new hardware
• Converting the allocation problem to scheduling problem
  – Software pipelining scheduling has been studied for 3 decades
  – Benefit from reusing well-known/stable algorithms
  – Not limited to software pipelined loops: non-pipelined loops can be treated as pipelined ones with only one stage
• Proposed LCP algorithm
  – Simple and fast
• Extended to Itanium general rotating registers
  – RegII \equiv 1
  – Resource constraints
BACKUP
Rotation

Base pointer

RR0

RR1

RR2

RR3

Demonstration: Rotation size 2
Rotation

Demonstration: Rotation size 2
Rotation

Base pointer

RR2

RR3

RR0

RR1

Demonstration: Rotation size 2
Rotate RegII registers every II time steps