Just-In-Time
Software Pipelining

Hongbo Rong
Youfeng Wu

Hyunchul Park
Cheng Wang

Programming Systems Lab
Intel Labs, Santa Clara
What is software pipelining?

A loop optimization exposing instruction-level parallelism (ILP)

for (i = 0; i < N; i++) {
  a: x = y + 1
  b: y = A[i] + x
  c: B[i+2] = B[i]*x
}
What is software pipelining?

A loop optimization exposing instruction-level parallelism (ILP)

```c
for (i = 0; i < N; i++) {
    a: x = y + 1
    b: y = A[i] + x
    c: B[i+2] = B[i]*x
}
```

Local dependence
Loop-carried dependence
Initiation interval ($II$) = 2
Initiation interval ($II$) = 2
Initiation interval ($II$) = 2
Initiation interval ($II$) = 2
Initiation interval ($II$) = 2

- Different iterations work on different stages in parallel
Initiation interval ($II$) = 2

- Different iterations work on different stages in parallel.
Initiation interval ($II$) = 2

- Different iterations work on different stages in parallel
Initiation interval \((II) = 2\)

• Different iterations work on different stages in parallel

• \(II\) is the performance indicator
Software pipelining has been static

- Extensively studied in 3 decades, and efficient for wide-issue architectures
  - VLIW [Lam 1988]
  - superscalar [Ruttenberg et al. 1996]
Software pipelining has been static

- Extensively studied in 3 decades, and efficient for wide-issue architectures
  - VLIW [Lam 1988]
  - superscalar [Ruttenberg et al. 1996]
- It is seen only in static compilers
Software pipelining has been static

- Extensively studied in 3 decades, and efficient for wide-issue architectures
  - VLIW [Lam 1988]
  - superscalar [Ruttenberg et al. 1996]
- It is seen only in static compilers
- Most works aim to minimize II but not compile overhead
It is time now to extend software pipelining to dynamic compilers!
It is time now to extend software pipelining to dynamic compilers!

• Dynamic languages are increasingly popular
  – JavaScript and PHP 88.9% and 81.5% in client and server websites (W3Techs)
It is time now to extend software pipelining to dynamic compilers!

- Dynamic languages are increasingly popular
  - JavaScript and Php 88.9% and 81.5% in client and server websites (W3Techs)
- Huge amount of legacy code
  - Small optimization scope: a loop iteration
  - Software pipelining enlarges the scope to many iterations
It is time now to extend software pipelining to dynamic compilers!

• Dynamic languages are increasingly popular
  – JavaScript and PhP 88.9% and 81.5% in client and server websites (W3Techs)

• Huge amount of legacy code
  – Small optimization scope: a loop iteration
  – Software pipelining enlarges the scope to many iterations

• Minimizing compile overhead must be the 1st objective
  – Only simple/fast algorithms can be used
It is time now to extend software pipelining to dynamic compilers!

• Dynamic languages are increasingly popular
  – JavaScript and PHP 88.9% and 81.5% in client and server websites (W3Techs)

• Huge amount of legacy code
  – Small optimization scope: a loop iteration
  – Software pipelining enlarges the scope to many iterations

• Minimizing compile overhead must be the 1\textsuperscript{st} objective
  – Only simple/fast algorithms can be used
  – linear-time algorithms are preferred
Challenges

• Memory aliases kill parallelism
  – Hardware: Atomic region + rotating alias registers [MICRO-46]
Challenges

• Memory aliases kill parallelism
  – Hardware: Atomic region + rotating alias registers [MICRO-46]

```c
for (i = 0; i<N; i++){
    a
    b
    c
    d
}
```
Challenges

• Memory aliases kill parallelism
  – Hardware: Atomic region + rotating alias registers [MICRO-46]

```c
for (i = 0; i<N; i++){
    a
    b
    c
    d
}
```
Challenges

- Memory aliases kill parallelism
  - Hardware: Atomic region + rotating alias registers [MICRO-46]

Original optimization scope

```c
for (i = 0; i < N; i++) {
  a
  b
  c
  d
}
```

```c
for (j = 0; j < N; j += M) {
  for (i = j; i < j + M; i++) {
    a
    b
    c
    d
  }
}
```
Challenges

- Memory aliases kill parallelism
  - Hardware: Atomic region + rotating alias registers [MICRO-46]
Challenges

- Memory aliases kill parallelism
  - Hardware: Atomic region + rotating alias registers [MICRO-46]
- Costly rollback
  - Software: Light-weight checkpointing

Original optimization scope

```plaintext
for (i = 0; i < N; i++){
    a
    b
    c
    d
}
```

Atomic region

```plaintext
for (j = 0; j < N; j += M) {
    for (i = j; i < j + M; i++) {
        a
        b
        c
        d
    }
}
```
Challenges

• Memory aliases kill parallelism
  – Hardware: Atomic region + rotating alias registers [MICRO-46]
• Costly rollback
  – Software: Light-weight checkpointing
• Scheduling is expensive

```java
for (i = 0; i < N; i++) {
  a
  b
  c
  d
}

Original optimization scope

for (j = 0; j < N; j += M) {
  for (i = j; i < j + M; i++) {
    a
    b
    c
    d
  }
}

Atomic region
```
Framework
(on Transmeta CMS)
x86 binary

Framework
(on Transmeta CMS)
x86 binary

Interpreter & profiler

Framework
(on Transmeta CMS)
x86 binary

Interpreter & profiler

Hot region optimizations

Framework
(on Transmeta CMS)
Hot region optimizations

Interpreter & profiler

x86 binary

Framework (on Transmeta CMS)

Acyclic scheduler
x86 binary

Interpreter & profiler

Hot region optimizations

Acyclic scheduler

Assembler

Framework
(on Transmeta CMS)
x86 binary

Interpreter & profiler → Hot region optimizations

Acyclic scheduler

Code cache ← Assembler

Framework (on Transmeta CMS)
x86 binary

Interpreter & profiler → Hot region optimizations

<table>
<thead>
<tr>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>(on Transmeta CMS)</td>
</tr>
</tbody>
</table>

Acyclic scheduler

Code cache → Assembler
Framework
(on Transmeta CMS)

- Hot region optimizations
  - Acyclic scheduler
    - Assembler
Framework
(on Transmeta CMS)

→ Hot region optimizations

→ Loop selection

→ Acyclic scheduler

→ Assembler
Framework (on Transmeta CMS)

- Hot region optimizations
- Loop selection
- Acyclic scheduler
- Assembler

Initialization
Framework (on Transmeta CMS)

- Hot region optimizations
- Initialization
- Loop selection
- Scheduling
- Acyclic scheduler
- Assembler
Framework (on Transmeta CMS)

→ Hot region optimizations

↓

Loop selection

↓

Acyclic scheduler

↓

Assembler

Initialization

↓

Scheduling

Rotating alias reg alloc
Framework (on Transmeta CMS)

- Hot region optimizations
- Loop selection
- Acyclic scheduler
- Assembler

Initialization

Scheduling

- Rotating alias reg alloc
- Code generation
Framework
(on Transmeta CMS)

- Hot region optimizations
- Loop selection
- Acyclic scheduler
- Assembler

- Initialization
- Scheduling
- Rotating alias reg alloc
- Code generation
Framework (on Transmeta CMS)

- Hot region optimizations
  - Loop selection
    - Acyclic scheduler
    - Assembler

- Initialization
  - Scheduling
    - Rotating alias reg alloc
  - Code generation

- Atomic region
- Rotating alias register file
Scheduling is expensive

• NP-complete problem to find an optimal schedule [Colland et al. 1996]
Scheduling is expensive

- NP-complete problem to find an optimal schedule [Colland et al. 1996]
- $O(V^3)$ at least, exponential at worst [Rau et al. 1992]
  - $V$: number of operations
Scheduling is expensive

- NP-complete problem to find an optimal schedule [Colland et al. 1996]
- $O(V^3)$ at least, exponential at worst [Rau et al. 1992]
  - $V$: number of operations
- *Can we linearize software pipelining?*
Keep scheduling time under control
Keep scheduling time under control

• Schedule in linear time
  – Use either simple or fast sub-algorithms
    • Avoid cubic or exponential complexity
Keep scheduling time under control

• Schedule in linear time
  – Use either simple or fast sub-algorithms
    • Avoid cubic or exponential complexity
  – For iterative sub-algorithms, have a threshold: the maximum #iterations allowed
Keep scheduling time under control

• Schedule in linear time
  – Use either simple or fast sub-algorithms
    • Avoid cubic or exponential complexity
  – For iterative sub-algorithms, have a threshold: the maximum #iterations allowed
    • The smaller the threshold, the less the compile overhead
    • Once exceeded, abort software pipelining
Keep scheduling time under control

• Schedule in linear time
  – Use either simple or fast sub-algorithms
    • Avoid cubic or exponential complexity
  – For iterative sub-algorithms, have a threshold: the maximum #iterations allowed
    • The smaller the threshold, the less the compile overhead
    • Once exceeded, abort software pipelining
    • Key question: *How small can the threshold be?*
Keep scheduling time under control

• Schedule in linear time
  – Use either simple or fast sub-algorithms
    • Avoid cubic or exponential complexity
  – For iterative sub-algorithms, have a threshold: the maximum #iterations allowed
    • The smaller the threshold, the less the compile overhead
    • Once exceeded, abort software pipelining
    • Key question: *How small can the threshold be?*

• Find a good enough schedule
  – No backtracking
  – Priority function: approximate and never update
  – Separate dependence and resource constraints
  – Separate local and loop-carried dependences
Keep scheduling time under control

• Schedule in linear time
  – Use either simple or fast sub-algorithms
    • Avoid cubic or exponential complexity
  – For iterative sub-algorithms, have a threshold: the maximum #iterations allowed
    • The smaller the threshold, the less the compile overhead
    • Once exceeded, abort software pipelining
    • Key question: How small can the threshold be?

• Find a good enough schedule
  – No backtracking
  – Priority function: approximate and never update
  – Separate dependence and resource constraints
  – Separate local and loop-carried dependences

• Iteratively improve a schedule
Just-In-Time Software Pipelining
Just-In-Time Software Pipelining

- Quickly creates an initial schedule to start with
Just-In-Time Software Pipelining

- Quickly creates an initial schedule to start with
- Handles local dependences and resources
Just-In-Time Software Pipelining

- Prepartition
  - Quickly creates an initial schedule to start with
- Local scheduling
  - Handles local dependences and resources
- Kernel expansion
  - Adjusts the schedule for loop-carried dependences
Just-In-Time Software Pipelining

- Quickly creates an initial schedule to start with
- Handles local dependences and resources
- Adjusts the schedule for loop-carried dependences

1. Prepartition
   \[ H, B \]

2. Local scheduling

3. Kernel expansion

Exit
Just-In-Time Software Pipelining

- Quickly creates an initial schedule to start with
- Handles local dependences and resources
- Adjusts the schedule for loop-carried dependences
- Iteratively improves the schedule
Just-In-Time Software Pipelining

- Prepartition
  - Handles local dependences and resources
  - Adjusts the schedule for loop-carried dependences
  - Iteratively improves the schedule
  - Time complexity: $O(V+E)$
    - $V$: #operations
    - $E$: #dependences

- Local scheduling
- Kernel expansion
- Rotation
- Exit
Illustration

- Prepartition
- Local scheduling
- Kernel expansion
- Exit
- Rotation

- Local dependences
- Resources
- Loop-carried dependences

Iteration
0 1 2 3
Prepartition

Local scheduling

Kernel expansion

Exit

Rotation

Local dependences

Resources

Loop-carried dependences

Illustration

Iteration

RecMII

abc
d

0 1 2 3
Illustration

Local dependences
Resources
Loop-carried dependences

Prepartition
Local scheduling
Kernel expansion
Exit
Rotation

Iteration
0 1 2 3

RecMII
Kernel

abc
d
abc
d
abc
d
abc
d
abc
d
abc
d
abc
d

Local scheduling
Kernel expansion
Prepartition
Exit
Rotation
Calculating RecMII

• Recurrence Minimum II
  – II determined by the biggest dependence cycles
  – Needed by almost every software pipelining method
Calculating RecMII

• Recurrence Minimum II
  – II determined by the biggest dependence cycles
  – Needed by almost every software pipelining method

Conventional

$O(V^3)$
Calculating RecMII

- Recurrence Minimum II
  - II determined by the biggest dependence cycles
  - Needed by almost every software pipelining method

Conventional

$O(V^3)$

- Turn the problem into a Markov decision process
  - 1st time Howard algorithm is applied to pipelining
Calculating RecMII

• Recurrence Minimum II
  – II determined by the biggest dependence cycles
  – Needed by almost every software pipelining method

Conventional Howard policy iteration algo.

- $O(V^3)$ $\quad O(\text{exponential}^*E)$

• Turn the problem into a Markov decision process
  – 1\textsuperscript{st} time Howard algorithm is applied to pipelining
Calculating RecMII

- Recurrence Minimum II
  - II determined by the biggest dependence cycles
  - Needed by almost every software pipelining method

Conventional Howard policy iteration algo. Linearized Howard

- Turn the problem into a Markov decision process
  - 1st time Howard algorithm is applied to pipelining
- Linearize Howard with a small constant $H$
Calculating RecMII

- Recurrence Minimum II
  - II determined by the biggest dependence cycles
  - Needed by almost every software pipelining method

Conventional Howard policy iteration algo. \( O(V^3) \)

Linearized Howard \( O(\text{exponential} \times E) \)

- Turn the problem into a Markov decision process
  - 1\(^{st}\) time Howard algorithm is applied to pipelining

- Linearize Howard with a small constant \( H \)
Calculating RecMII

- Recurrence Minimum II
  - II determined by the biggest dependence cycles
  - Needed by almost every software pipelining method

Conventional Howard policy iteration algo. Linearized Howard

\[ O(V^3) \quad O(\text{exponential} \times E) \quad O(E) \]

- Turn the problem into a Markov decision process
  - 1\textsuperscript{st} time Howard algorithm is applied to pipelining
- Linearize Howard with a small constant \( H \)
- A by-product: critical operations
Divide operations into stages

Bellman-Ford

$O(V^*E)$
Divide operations into stages

Bellman-Ford

O(V*E)

• Also an iterative sub-algorithm
Divide operations into stages

Bellman-Ford \( \mathcal{O}(V \cdot E) \) \xrightarrow{\text{linearize}} \text{Linearized Bellman-Ford} \( \mathcal{O}(B \cdot E) \)

- Also an iterative sub-algorithm
- Linearize it with a constant \( B \)
Divide operations into stages

- Bellman-Ford
  \[ O(V \times E) \]
- Linearized Bellman-Ford
  \[ O(B \times E) \]

- Also an iterative sub-algorithm
- Linearize it with a constant \( B \)
- Specific order in visiting edges
  - Scan nodes in sequential order, and visit their incoming edges
    - Values are propagated along local edges in the 1\(^{st}\) itr.
    - Values are propagated along loop-carried edges in the 2\(^{nd}\) itr.
Divide operations into stages

Bellman-Ford \[ O(V^*E) \] \[ \rightarrow \]
Linearized Bellman-Ford \[ O(E) \]

- Also an iterative sub-algorithm
- Linearize it with a constant \( B \)
- Specific order in visiting edges
  - Scan nodes in sequential order, and visit their incoming edges
    - Values are propagated along local edges in the 1\textsuperscript{st} itr.
    - Values are propagated along loop-carried edges in the 2\textsuperscript{nd} itr.
Illustration

Prepartition

Local scheduling

Kernel expansion

Exit

Rotation

0 1 2 3 Iteration

abc

d

RecMII

abc

d

Kernel

Local dependences

Resources

Loop-carried dependences
Prepartition

Local scheduling

Kernel expansion

Exit

Rotation

Iteration

0 1 2 3

RecMII

Local dependences

Resources

Loop-carried dependences

Illustration
Illustration (Cont.)

- Prepartition
  - Local scheduling
    - Kernel expansion
      - Exit
      - Rotation

Kernel expansion
- Local dependences
- Resources
- Loop-carried dependences

Iteration
- 0
- 1
- 2
- 3

abc
abc
abc
abc

Kernel

Illustration (Cont.)
Local scheduling

• Any local scheduling algorithm can be used
  – E.g. list scheduling
Local scheduling

• Any local scheduling algorithm can be used
  – E.g. list scheduling

• Weakness:
  – Loop-carried dependences may be violated
Local scheduling

- Any local scheduling algorithm can be used
  - E.g. list scheduling
- Weakness:
  - Loop-carried dependences may be violated
- To reduce the chance of violation:
  - Before scheduling, priority function considers loop-carried dependences in advance
  - Prioritize critical operations
Illustration (Cont.)

- Local dependences
- Resources
- Loop-carried dependences
Illustration (Cont.)

- Local dependences
- Resources
- Loop-carried dependences
Illustration (Cont.)

- **Local dependences**: X
- **Resources**: X
- **Loop-carried dependences**: 

Diagrams showing iterations: 0, 1, 2, 3
Illustration (Cont.)

- Prepartition
- Local scheduling
- Kernel expansion
- Exit
- Rotation

Local dependences
Resources
Loop-carried dependences

Iteration

0 1 2 3

Kernel

Rotation

Exit

Local scheduling

Kernel expansion

Prepartition

Illustration (Cont.)
Illustration (Cont.)
Illustration (Cont.)

- Prepartition
- Local scheduling
- Kernel expansion
- Exit

- Rotation

- Local dependences
- Resources
- Loop-carried dependences
Experiments

- Transmeta CMS on SPEC2k traces
Experiments

• Transmeta CMS on SPEC2k traces
• Functional simulator to comprehensively
  – Explore thresholds $H$ and $B$
  – Evaluate compile overhead and schedules’ quality
Experiments

• Transmeta CMS on SPEC2k traces
• Functional simulator to comprehensively
  – Explore thresholds $H$ and $B$
  – Evaluate compile overhead and schedules’ quality
• Cycle-accurate simulator
  – Simulates cache misses, latencies, ...
  – Initial performance study
Linearized Howard vs. exponential backoff + binary search [Rau et al. 1992]
Linearized Howard vs. exponential backoff + binary search [Rau et al. 1992]
Linearized Howard vs. exponential backoff + binary search [Rau et al. 1992]
Linearized Howard vs. exponential backoff + binary search [Rau et al. 1992]

Average 9X faster
Peak 812X faster
$H \leq 3$ for 96% of 11,992 loops
Linearized Howard vs. exponential backoff + binary search [Rau et al. 1992]

Average 9X faster
Peak 812X faster
$H \leq 3$ for 96% of 11,992 loops
$H \leq 14$ for all loops
Linearized Bellman-Ford
Linearized Bellman-Ford

% Total loops

0% 50% 100%

2 3 4 5 B
Linearized Bellman-Ford

- $B \leq 3$ for 98.8% of 11,992 loops
Linearized Bellman-Ford

- $B \leq 3$ for 98.8% of 11,992 loops
Linearized Bellman-Ford

- \( B \leq 3 \) for 98.8% of 11,992 loops
- \( B \leq 5 \) for all the loops
Linearized Bellman-Ford

- \( B \leq 3 \) for 98.8% of 11,992 loops
- \( B \leq 5 \) for all the loops
- From now on, we set \( H=10, B=5 \) \( \Rightarrow \) 11,910 loops scheduled
Scheduling overhead & schedules’ quality

Overhead

% loops with optimal schedules

RS2  DESP  JITSP

RS2  DESP  JITSP

Overhead % loops with optimal schedules

95%
Scheduling overhead & schedules’ quality

- JITSP achieves optimal schedules for 95% loops
Scheduling overhead & schedules’ quality

- JITSP achieves optimal schedules for 95% loops
Scheduling overhead & schedules’ quality

- JITSP achieves optimal schedules for 95% loops

Overhead: RS2 > 12X, DESP > 1X, JITSP = 1X

% loops with optimal schedules: RS2 = 13%, DESP = 95%, JITSP = 95%
Scheduling overhead & schedules’ quality

- JITSP achieves optimal schedules for 95% loops
Scheduling overhead & schedules’ quality

- JITSP achieves optimal schedules for 95% loops
Compile overhead distribution

- Acyclic scheduler, 27%
- Software pipelining, 33%
- Assembler, 6%
- Hot region optimizations, 34%

Note: acyclic scheduler handles acyclic code, or loops NOT selected for software pipelining
Preliminary performance

• 40 hot loops
Preliminary performance

- 40 hot loops

- Successfully generated code: 25%
- Filtered: 19%
- Too many registers: 55%
- Too many unrolls: 1%
40 hot loops

- 25% Successfully generated code
- 19% filtered
- 55% too many registers
- 1% too many unrolls

Preliminary performance
Preliminary performance

- 40 hot loops
- The architecture has bottleneck in registers
  - 2/7/24/32 predicate/static alias/integer/floating point available for pipelining
Preliminary performance

- 40 hot loops
- The architecture has bottleneck in registers
  - 2/7/24/32 predicate/static alias/integer/floating point available for pipelining
• 40 hot loops

• The architecture has bottleneck in registers
  – 2/7/24/32 predicate/static alias/integer/floating point available for pipelining
Preliminary performance (Cont.)

II/MII (The lower, the better)

Optimal
Preliminary performance (Cont.)

• JITSP achieves optimal schedules for all but 1 loops
Preliminary performance (Cont.)

- JITSP 10~36% speedup. Better than the others
Preliminary performance (Cont.)

- JITSP 10~36% speedup. Better than the others
- Exception: loop 2. Optimal schedule but slowdown due to memory aliases ➔ retranslation needed
Preliminary performance (Cont.)

- JITSP 10~36% speedup. Better than the others
- Exception: loop 2. Optimal schedule but slowdown due to memory aliases ➔ retranslation needed
- Speedup swim(5.3%), ammp(4.4%), mcf(3.6%)
Conclusion
Conclusion

• 1st linear software pipelining algorithm implemented for dynamic compilers
Conclusion

• 1st linear software pipelining algorithm implemented for dynamic compilers
• Turns a traditionally-expensive optimization into linear time $O(V+E)$
Conclusion

• 1st linear software pipelining algorithm implemented for dynamic compilers
• Turns a traditionally-expensive optimization into linear time $O(V+E)$
  – Taking advantages of results from various domains:
    • hardware circuit design (Retiming)
    • stochastic control (Howard algorithm)
    • Graph (Bellman-Ford)
    • software pipelining (Rotation scheduling and DESP)
Conclusion

• 1st linear software pipelining algorithm implemented for dynamic compilers
• Turns a traditionally-expensive optimization into linear time $O(V+E)$
  – Taking advantages of results from various domains:
    • hardware circuit design (Retiming)
    • stochastic control (Howard algorithm)
    • Graph (Bellman-Ford)
    • software pipelining (Rotation scheduling and DESP)
• Generates optimal or near-optimal schedules with reasonable compile overhead
Future work

• Register availability
  – Add more architecture registers
  – Algorithm: Register pressure-aware

• Implementation
  – Loop selection
  – Re-translation

• Evaluation
  – Benchmarks
Backup slides
Howard algorithm

• Each operation is rewarded to reach a cycle via 1 policy edge
  – The bigger the cycle, the more the reward
Howard algorithm

• Each operation is rewarded to reach a cycle via 1 policy edge
  – The bigger the cycle, the more the reward
Howard algorithm

• Each operation is rewarded to reach a cycle via 1 policy edge
  – The bigger the cycle, the more the reward
Howard algorithm

- Each operation is rewarded to reach a cycle via 1 policy edge
  - The bigger the cycle, the more the reward
Distribution statistics of JITSP

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>median</th>
<th>mean</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td># operations</td>
<td>4</td>
<td>10</td>
<td>13.82</td>
<td>96</td>
</tr>
<tr>
<td># dependences</td>
<td>12</td>
<td>38</td>
<td>50.48</td>
<td>353</td>
</tr>
<tr>
<td># local dependences</td>
<td>1</td>
<td>13</td>
<td>22.36</td>
<td>283</td>
</tr>
<tr>
<td># loop-carried deps</td>
<td>5</td>
<td>22</td>
<td>28.12</td>
<td>156</td>
</tr>
<tr>
<td>MII</td>
<td>1</td>
<td>3</td>
<td>4.78</td>
<td>55</td>
</tr>
<tr>
<td>II - MII</td>
<td>0</td>
<td>0</td>
<td>0.05</td>
<td>5</td>
</tr>
<tr>
<td>II / MII</td>
<td>1</td>
<td>1</td>
<td>1.01</td>
<td>1.5</td>
</tr>
<tr>
<td># local scheduling</td>
<td>1</td>
<td>1</td>
<td>1.26</td>
<td>4</td>
</tr>
</tbody>
</table>