Today’s Objectives

- Provide background on key IA-64 architecture concepts
  - Today’s focus on architecture

- Demonstrate IA-64 architecture innovation benefits
  - A few key features

- Describe what the IA-64 architecture means for developers
  - New levels of performance
  - Better utilization of wider machines
IA-64 Architecture Innovations

Outline

• Background
  – IA-64 architecture objectives
  – Explicit Parallelism
  – Predication
  – Control Speculation

• IA-64 at work: Code Examples

• Summary
IA-64 Architecture Objectives

- Enable industry leading system performance
  - Overcome traditional architectures’ performance limiters
- Provide end user investment protection
  - Enable full binary compatibility with IA-32 software
- Allow scalability over a wide range of implementations
- Full 64-bit computing
### Explicitly Parallel Instruction Computing

<table>
<thead>
<tr>
<th>Explicit parallelism</th>
<th>Features that enhance Instruction Level Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP is explicit in machine code</td>
<td>Predication</td>
</tr>
<tr>
<td>Compiler schedules across a wide scope</td>
<td>Speculation</td>
</tr>
<tr>
<td>Binary compatibility across all family members</td>
<td>Others...</td>
</tr>
</tbody>
</table>

### Resources for parallel execution

- Many registers
- Many functional units
- Inherently scalable
A Few Key IA-64 Architecture Innovations

- Explicit Parallelism
  - Enables compiler to expose Instruction Level Parallelism

- Predication
  - Removes branches & mispredicts

- Speculation
  - Minimizes the effect of memory latency
Extracting Parallelism

- Sequential execution model

- Compiler has limited, indirect view of hardware

Traditional architectures enable limited parallelism.
Better Strategy: Explicit Parallelism

Compiler exposes, enhances, and exploits parallelism in the source program and makes it *explicit* in the machine code.

Explicit parallelism increases ILP
Branches Limit Performance

Control flow introduces branches

if
Load a[i].ptr
p1, p2 = cmp a[i].ptr != 0
branch if p2
then
Load a[i].l
store b[i]
branch
else
Load a[i].r
store b[i]
i = i + 1

else

If a[i].ptr != 0
b[i] = a[i].l;
else
b[i] = a[i].r;
i = i + 1

Traditional Architectures: 4 basic blocks
Predication

if

Load a[i].ptr
p1, p2 = cmp a[i].ptr != 0
branch if p2

then

Load a[i].l
store b[i]
branch

else

Load a[i].r
store b[i]

i = i + 1

Predication removes branches and eliminates mispredicts

If a[i].ptr != 0
b[i] = a[i].l;
else
b[i] = a[i].r;
i = i + 1
Predication Enhances Parallelism

**Traditional Architectures:** 4 basic blocks

- **if**
  - Load a[i].ptr
  - p1, p2 = cmp a[i] != 0
  - jump if p2

- **then**
  - Load a[i].l
  - store b[i]
  - jump

- **else**
  - Load a[i].r
  - store b[i]

  `i = i + 1`

**IA-64™ Architecture:** 1 basic block

- **if**
  - Load a[i].ptr
  - p1, p2 = cmp a[i] != 0

- **then**
  - `<p1>` Load a[i].l
  - `<p2>` Load a[i].r
  - `<p1>` store b[i]
  - `<p2>` store b[i]

  `i = i + 1`

Predication enables more effective use of parallel hardware
Memory Latency Causes Delays

- Loads significantly affect performance
  - Often first instruction in dependency chain of instructions
  - Can incur high latencies

Traditional Architectures

Add \( t1 + 1 \)
comp \( t1 > t2 \)
branch

Load \( a[t1-t2] \)
Load \( b[j] \)
add \( b[j] + 1 \)

If \( t1 > t2 \)
\( j = a[t1 - t2] \)
\( b[j] ++ \)

Loads can cause exceptions
Speculation with IA-64 Architecture

- **Separate load behavior from exception behavior**
  - Speculative load instruction (**load.s**) initiates a load operation and detects exceptions
  - Propagate an exception “token” (stored with destination register) from load.s to check.s
  - Speculative check instruction (**check.s**) delivers any exceptions detected by load.s
Speculation Minimizes the Effect of Memory Latency

- Give scheduling freedom to the compiler
  - Allows `load.s` to be scheduled above branches
  - `check.s` remains in home block, branches to fix up code if an exception is propagated
## Predication & Speculation

### With Predication

<table>
<thead>
<tr>
<th><strong>Instruction</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Load a[i].ptr</td>
<td>Load a[i].ptr</td>
</tr>
<tr>
<td>p1, p2 = cmp a[i].ptr != 0</td>
<td>p1, p2 = cmp a[i].ptr != 0</td>
</tr>
<tr>
<td>&lt;p1&gt; Load a[i].l</td>
<td>&lt;p1&gt; Load a[i].r</td>
</tr>
<tr>
<td>&lt;p1&gt; store b[i]</td>
<td>&lt;p1&gt; store b[i]</td>
</tr>
<tr>
<td>i = i + 1</td>
<td>i = i + 1</td>
</tr>
</tbody>
</table>

### With Predication & Speculation

<table>
<thead>
<tr>
<th><strong>Instruction</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Load a[i]</td>
<td>Load a[i]</td>
</tr>
<tr>
<td>load.s a[i].l</td>
<td>load.s a[i].r</td>
</tr>
<tr>
<td>p1, p2 = cmp a[i] != 0</td>
<td>p1, p2 = cmp a[i] != 0</td>
</tr>
<tr>
<td>&lt;p1&gt; check.s</td>
<td>&lt;p1&gt; check.s</td>
</tr>
<tr>
<td>&lt;p1&gt; store b[i]</td>
<td>&lt;p1&gt; store b[i]</td>
</tr>
<tr>
<td>i = i + 1</td>
<td>i = i + 1</td>
</tr>
</tbody>
</table>

### Predication and Speculation = higher ILP

If a[i].ptr != 0

```
b[i] = a[i].l;
```

else

```
b[i] = a[i].r;
i = i + 1
```
IA-64: Next Generation Architecture

- A unique combination of innovative features:
  - Explicit Parallelism
  - Predication
  - Speculation

IA-64 Architecture: Performance, Scalability, and Compatibility
IA-64 Architecture Innovations

Outline

- Background
- IA-64 at work: Code Examples
  - xlxgetvalue from LI
    - control speculation to chase pointers
  - puzzle code fragment
    - loop with nested if statements
  - treeins code fragment
    - classic if-then-else statement
- Summary
Demonstrate the IA-64 architecture benefits on representative code fragments

- **Pointer chasing**
  - control speculation to load data before pointer safety check

- **Loop with conditional statement**
  - control speculation to unroll loop
  - predication to remove hard to predict branches

- **If-then-else statements**
  - predication to execute all paths in parallel
IA-64 Architecture Innovations

Outline

- Background
- IA-64 at work: Code Examples
  - `xixgetvalue from LI`
    - control speculation to chase pointers
  - puzzle code fragment
    - loop with nested if statements
  - `treeins code fragment`
    - classic if-then-else statement
- Summary
Xlxgetvalue in a Nutshell

- **Code fragment from SpecInt95 benchmark LI**
  - representative of pointer chasing code
- **Technique used:** Control Speculation
- **Benefits:**
  - hide memory latency
  - expose Instruction Level Parallelism allowing parallel execution
Xlxgetvalue Step by Step

- Compile one iteration
  - use control speculation to issue loads as early as possible

- Unroll the loop
  - use control speculation to start next iteration before it is safe to do so
  - take advantage of the machine width to execute several iterations in parallel

Expose ILP with Control Speculation in pointer chasing code
for (fp = xlenv; fp;
   fp = cdr(fp))
   for (ep = car(fp); ep;
      ep = cdr(ep))
      if (sym ==
          car(car(ep)))
       return (cdr(car(ep)));

fp = cdr(fp)
if (sym == car(car(ep)))
ep != nil
fp != nil
exit
return
Compiling Serial Code . . .

Ld fp
cmp fp == nil
br to exit if true
Load ep
Cond1 = (cmp ep == nil)
br nxt_fp if Cond1
load car(ep)
load x= car(car(ep))
Cond2 = (comp sym== x)
br to return if Cond2
br nxt_ep
Example Machine Model for xilxgetval

- 6 functional units
- 1 cycle load latency
- 2 memory ports
- 6 Execution units, 2 memory ports, 1 cycle load latency

Diagram:
- Register File
- L0 DCache
- L0 Icache
- Instruction Pointer
- Instruction Decode and Dispatch
- Resteer
- 8 cycle branch mispredict
### Compiling...

```lisp
for (ep = car(fp); ep; ep = cdr(ep))
  if (sym == car(car(ep)))
```

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load ep1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle</td>
<td>Unit 1</td>
<td>Unit 2</td>
<td>Unit 3</td>
<td>Unit 4</td>
<td>Unit 5</td>
<td>Unit 6</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>1</td>
<td>Load ep1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Cond1 = ep1 == nil</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compiling . . .

for (ep = car(fp); ep; ep = cdr(ep))
if (sym == car(car(ep)))
for (ep = car(fp); ep; ep = cdr(ep))
    if (sym == car(car(ep)))

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load ep1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Cond1 =</td>
<td>Load.s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ep1 == nil</td>
<td>car(ep1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>check.s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
for (ep = car(fp); ep; ep = cdr(ep))
if (sym == car(car(ep)))

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load ep1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Cond1 = ep1 == nil</td>
<td>Load x=car</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>car(ep1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>check.s</td>
<td>Load x=car</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(car(ep1))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
for (ep = car(fp); ep; ep = cdr(ep))
if (sym == car(car(car(ep))))

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load ep1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Cond1 =</td>
<td>Load.s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ep1 == nil</td>
<td>car(ep1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>check.s</td>
<td>Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x=car(car(ep1))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Cond2 =</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sym== x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### First Iteration

Speculation allows the loads to be started early.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load ep1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Cond1 = ep1 == nil</td>
<td>Load.s car(ep1)</td>
<td>Br nxt_fp if cond1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>check.s</td>
<td>Load x=car(car(car(ep1)))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Cond2 = sym== x</td>
<td>Br return if cond2</td>
<td>Br nxt_ep</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For the second iteration, the unrolling process is described as follows:

```
for (ep = car(fp); ep; ep = cdr(ep))
if (sym == car(car(ep)))
```

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>U. 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld ep1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Ld.s car(ep1)</td>
<td>Cond1 = Cmp ep == nil</td>
<td>Ld.s ep2 = cdr(ep1)</td>
<td>Br nxt_fp if cond1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Check.s</td>
<td>Ld car car(ep1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Cond2 = Cmp == symm</td>
<td></td>
<td></td>
<td>Br return if cond2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Second Iteration: Unrolling . . .

for \((ep = \text{car}(fp); ep; ep = \text{cdr}(ep))\)
if \((\text{sym} == \text{car}(	ext{car}(ep)))\)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>U. 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld ep1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Ld.s car(ep1)</td>
<td>Cond1 = Cmp ep == nil</td>
<td>Ld.s ep2 = cdr(ep1)</td>
<td>Br nxt_fp if cond1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Check.s</td>
<td>Ld car car(ep1)</td>
<td>Ld.s car(ep2)</td>
<td>Cond3= Cmp ep2==nil</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Cond2 = Cmp == symm</td>
<td>Check.s</td>
<td>Br return if cond2</td>
<td>Br nxt_fp if cond3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Only 1 check for 2 dependent loads
Second Iteration: Unrolling...

```lisp
for (ep = car(fp); ep; ep = cdr(ep))
  if (sym == car(car(ep)))
```

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>U. 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld ep1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Ld.s car(ep1)</td>
<td>Cond1 = Cmp ep == nil</td>
<td>Ld.s ep2 = cdr(ep1)</td>
<td>Br nxt_fp if cond1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Check.s car(ep1)</td>
<td>Ld car car(ep1)</td>
<td>Ld.s car(ep2)</td>
<td>Cond3 = Cmp ep2==nil</td>
<td>Br return if cond2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Cond2 = Cmp == symm</td>
<td>Check.s</td>
<td>Ld car car(ep2)</td>
<td>Br return if cond2</td>
<td>Br nxt_fp if cond3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Second Iteration: Unrolling...

```
for (ep = car(fp); ep; ep = cdr(ep))
if (sym == car(car(ep)))
```

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>U. 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld ep1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Ld.s car(ep1)</td>
<td>Cond1 = Cmp ep == nil</td>
<td>Ld.s ep2 = cdr(ep1)</td>
<td>Br nxt_fp if cond1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Check.s Ld car(car(ep1))</td>
<td>Ld s car(ep2)</td>
<td>Cond3 = Cmp ep2 == nil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Cond2 = Cmp == sym</td>
<td>Check.s Ld car(car(ep2))</td>
<td>Br return if cond2</td>
<td>Br nxt_fp if cond3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Cond4 = Cmp == sym</td>
<td>Br return if cond4</td>
<td>Br nxt_ep</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Speculation enables efficient machine utilization
## Optimized Code

```plaintext
for (ep = car(fp); ep; ep = cdr(ep))
  if (sym == car(car(ep)))
```

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>U. 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ld ep1</td>
<td>(Done outside of the loop)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Ld.s car(ep1)</td>
<td>Cond1 = Cmp ep == nil</td>
<td>Ld.s ep2</td>
<td>Br nxt_fp</td>
<td>if cond1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ld.s car(ep2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Check.s</td>
<td>Ld car ep1</td>
<td>Cond3 = Cmp</td>
<td>Br return</td>
<td>if cond3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ep2==nil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Cond2 = Cmp</td>
<td>Cond4 = Cmp == sym</td>
<td>Br return</td>
<td>Br nxt_fp</td>
<td>if cond3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>symp == nil</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Ld nxt</td>
<td>Br return</td>
<td>Br nxit_ep</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ep1 = cdr(ep2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**First load can be done at the bottom of the loop**
### Scheduled without Control Speculation

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>U. 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ld ep1</td>
<td>Done outside of the loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Ld.s car(ep1)</td>
<td>Cond1 = Cmp ep == nil</td>
<td>Ld.s ep2 = cdr(ep1)</td>
<td>Br nxt_fp if cond1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Check.s</td>
<td>Ld car car(ep1)</td>
<td>Ld.s car(ep2)</td>
<td>Cond3 = Cmp ep2==nil</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Cond2 = Cmp == symm</td>
<td>Check.s</td>
<td>Ld car car(ep2)</td>
<td>Br return if cond2</td>
<td>Br nxt_fp if cond3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Ld nxt ep1 = cdr(ep2)</td>
<td>Cond4 = Cmp == sym</td>
<td>Br return if cond4</td>
<td>Br nxt_ep</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- Loads are delayed by one clock
- Scheduled without Control
- Scheduled without Speculation
## Recompiled without Speculation

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Cond1 = Cmp ep == nil</td>
<td></td>
<td>Br nxt_fp if cond1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Ld car(ep1)</td>
<td>Ld ep2 = cdr(ep1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Ld car(car(ep1))</td>
<td></td>
<td>Cond3 = Cmp ep2==nil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Cond2 = Cmp == symm</td>
<td>Ld car(ep2)</td>
<td></td>
<td>Br return if cond2</td>
<td>Br nxt_fp if cond3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Ld car(car(ep2))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Ld nxt ep1 = cdr(ep2)</td>
<td>Cond4 = Cmp == sym</td>
<td></td>
<td>Br return if cond4</td>
<td>Br nxt_ep</td>
<td></td>
</tr>
</tbody>
</table>

**Inefficient use of machine width**
Control speculation Benefits:

- hides memory latency through
  - loading data before knowing if the address is a valid pointer
  - loading data before knowing if the next loop iteration is valid
- enables the compiler to expose parallelism in pointer chasing code

On average over 50% of loads can be executed speculatively
Speculation provides a significant performance advantage
IA-64 Architecture Innovations

Outline

- Background
- IA-64 at work: Code Examples
  - `ggetvalue` from LI
    - control speculation to chase pointers
  - puzzle code fragment
    - loop with nested if statements
  - `treeins` code fragment
    - classic if-then-else statement
- Summary
Puzzle in a Nutshell

- **Classic Code fragment**
  - representative of loop with nested if statements

- **Technique used:** Predication and Control Speculation

- **Benefits:**
  - remove hard to predict branches & mispredicts
  - expose Instruction Level Parallelism allowing parallel execution
Puzzle Step by Step

- Compile one iteration
  - use predication to remove a hard to predict branch

- Unroll the loop
  - use control speculation to start next iteration before it is safe to do so
  - take advantage of the width of the machine to execute several iterations in parallel

Exposé ILP with predication in loop with nested if statements
int fit(i,j)
int i,j;
{
    int k;

    for (k = 0;
        k <= kmax;
        k++)
    
        if (p[i][k])
        if (puzzle[j+k])
            return (false);

    return(true);
}
Hard to Predict Branch

Highly Mispredicted Branch

Remove the branch by predicing this block

New Compiler Scheduling Scope without the branch

Remaining Branches are easy to predict

if (p[i][k])

if (puzzle[j+k])

Return (false)

return (true)

next k
int fit(i,j)
int i,j;
{
    int k;
    for (k = 0;
        k <= kmax;
        k++)
        if (p[i][k])
            if (puzzle[j+k])
                return (false);
        return(true);
}

@p +=1
Load x= p[I][k]
cond1 = cmp x== 1
br to k+1 if !cond1
@puzzle += 1
Load y= puzzle[j+k]
cond2 = cmp y == 1
br to return cond2
k ++
cmp k<= kmax
br next_k

Branch prediction incurs performance penalties.

Highly mispredicted branch
Example Machine Model for Puzzle

6 functional units

Register File

L0 DCache

Instruction Decode and Dispatch

Instruction Pointer

Resteer

2 cycle load latency

L0 DCache

2 memory ports

6 Execution units, 2 memory ports, 2 cycle load latency

8 cycle branch mispredict
for (k = 0; k <= kmax; k++)
    if (p[i][k])
        if (puzzle[j+k])
            return (false);

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load x = p[l][k]</td>
<td>@p += 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>P1 = Cmp x == 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
for (k = 0;  k <= kmax;  k++)
    if (p[i][k])
        if (puzzle[j+k])
            return (false);

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load x= p[l][k]</td>
<td>Load y= puzzle[j+k]</td>
<td>@p +=1</td>
<td>@puzzle +=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>P1 = Cmp x==0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>&lt;p1&gt; p2=Cmp y==0</td>
<td></td>
<td></td>
<td>&lt;p2&gt;Br to return</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**First Iteration**

```plaintext
for (k = 0; k <= kmax; k++)
    if (p[i][k])
        @p += 1
        @puzzle += 1
        return (false);
```

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load x = p[l][k]</td>
<td>Load y = puzzle[j+k]</td>
<td>@p += 1</td>
<td>@puzzle += 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>P1 = Cmp x == 0</td>
<td>K++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>&lt;p1&gt; br to k+1</td>
<td>&lt;p2&gt; Br to return</td>
<td>&lt;p3&gt; Br to next_k</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Using predication to remove a hard to predict branch
Second Iteration
Unrolling . . .

for (k = 0; k <= kmax; k++)
if (p[i][k])
if (puzzle[j+k])
return (false);

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load x= p[l][k]</td>
<td>Load y= puzzle[j+k]</td>
<td>@p +=1</td>
<td>@puzzle +=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Load.s x2=p[l][k]</td>
<td>Load.s y2= puzzle[k+k]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>P1 = Cmp x1==0</td>
<td>K++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>p1=Cmp x1==0</td>
<td>P3=Cmp k &lt;=kmax</td>
<td>&lt;p2&gt;Br to return</td>
<td>&lt;p3&gt; br to exit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Check.s</td>
<td>Check.s</td>
<td>P6 = cmp k&lt;=kmax</td>
<td>&lt;p5&gt; br to return</td>
<td>&lt;p6&gt; br next_k</td>
<td></td>
</tr>
</tbody>
</table>

Using speculation to unroll the loop
for (k = 0;  k <= kmax  k++)
  if (p[i][k])
    if (puzzle[j+k])
      return (false);

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load x= p[l][k]</td>
<td>Load y= puzzle[j+k]</td>
<td>@p +=1</td>
<td>@puzzle +=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Load.s x2=p[l][k]</td>
<td>Load.s y2=puzzle[j+k]</td>
<td>@p +=1</td>
<td>@puzzle +=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>P1 = Cmp x==0</td>
<td>K++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>&lt;p1&gt; p2=Cmp y==0</td>
<td>P3=Cmp k&lt;=kmax x2==0</td>
<td>P4 = cmp y2==0</td>
<td>K +=1</td>
<td>&lt;p2&gt; Br to return</td>
<td>&lt;p3&gt; br to exit</td>
</tr>
<tr>
<td>5</td>
<td>Check.s</td>
<td>Check.s</td>
<td>&lt;p4&gt; p5 = cmp k&lt;=kmax</td>
<td>&lt;p5&gt; br to return</td>
<td>&lt;p6&gt; br next_k</td>
<td></td>
</tr>
</tbody>
</table>
Scheduled without Predication or Speculation

for (k = 0; k <= kmax; k++)
  if (p[i][k])
    if (puzzle[j+k])
      return (false);

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>P3=Cmp k&lt;=kmax</td>
<td>@p +=1</td>
<td>@puzzle +=1</td>
<td>Br to exit if p3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>K ++ Load x2=p[i][k]</td>
<td>Load y2=puzzle</td>
<td>P1=Cmp x==0</td>
<td>Br to 3 if p1</td>
<td>Loop to 1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>p2=Cmp y==0</td>
<td></td>
<td></td>
<td>Br to return if p2</td>
<td>Loop to 1</td>
<td></td>
</tr>
</tbody>
</table>

A hard to predict branch cannot be avoided, mispredict penalties add execution time
Puzzle Code Fragment

Conclusions

- **Predication Benefits**
  - remove difficult to predict branches
  - remove high penalty for branch mispredicts

- **Control Speculation Benefits**
  - unroll the loop and execute loads of next iteration
  - unroll as much as needed to fill wide machine

- **Benefit over Out of Order Execution:**
  - on average, predication can eliminate over 50% of branches

Combining Predication & Speculation results in a significant performance advantage
Comparing Bills

With speculation and predication:
- 5 cycles

Without speculation nor predication:
- 5 cycles
- 2 branches
- 2 clocks
- 4 cycles
- X
- 9 cycles

Schedule for 2 iteration:
- Branch Mispredict Penalty
- 25% mispredict rate
With speculation and predication

- xlxgetvalue: 2 cycles per iteration
- puzzle: 2.5 cycles per iteration

Without speculation or predication

- xlxgetvalue: 3 cycles per iteration
- puzzle: 4.5 cycles per iteration
IA-64 Architecture Innovations

Outline

- Background
- IA-64 at work: Code Examples
  - xlxgetvalue from LI
    - control speculation to chase pointers
  - puzzle code fragment
    - loop with nested if statements
  - treeins code fragment
    - classic if-then-else statement
- Summary
Treeins in a Nutshell

- Classic Code fragment
  - representative of if-then-else control structures
- Technique used: Predication
- Benefits:
  - remove hard to predict branches
  - execute all paths in parallel
  - expose Instruction Level Parallelism allowing parallel execution
L10: /* compare */
if (s.nodes[i].a < x)
    if (s.nodes[i].l != 0) {
        i = s.nodes[i].l;
        goto L10;}
else {
    s.nodes[i].l = j;
    goto L20;}
else{ 
    if (s.nodes[i].r != 0) {
        i = s.nodes[i].r;
        goto L10;}
    else{ 
        s.nodes[i].r = j;
        goto L20;}
}
L20: /* insert */
    s.nodes[j].a = x;
    s.nodes[j].l = 0;
    s.nodes[j].r = 0;
    s.nxt_avail = j+1;
Treeins Step by Step

- Assign a predicate to each block
- Schedule the left most "then path"
- Schedule the second "then path"
- Schedule the "else paths" in parallel with the "then paths"
```c
L10: /* compare */
    if (s.nodes[i].a < x)
        if (s.nodes[i].l != 0 ) {
            i = s.nodes[i].l;
            goto L10;
        } else {
            s.nodes[i].l = j;
            goto L20;
        }
    else{
        if (s.nodes[i].r != 0) {
            i = s.nodes[i].r;
            goto L10;
        } else{
            s.nodes[i].r = j;
            goto L20;
        }
    }
L20: /* insert */
    s.nodes[j].a = x;
    s.nodes[j].l = 0;
    s.nodes[j].r = 0;
    s.nxt_avail = j+1;
```
Example Machine model for Treeins

6 functional units

Register File

L0 DCache

Instruction Pointer

Instruction Decode and Dispatch

Instruction

Resteer

1 cycle load latency

6 Execution units, 3 memory ports, 1 cycle load latency

3 memory ports

8 cycle branch mispredict
L10: /* compare */
if (s.nodes[i].a < x)
  if (s.nodes[i].l != 0 ) {
    i = s.nodes[i].l;
    goto L10;}
else {
  s.nodes[i].l = j;
  goto L20;}
else{
  if (s.nodes[i].r != 0) {
    i = s.nodes[i].r;
    goto L10;}
  else{
    s.nodes[i].r = j;
    goto L20;}
}
L20: /* insert */
s.nodes[j].a = x;
s.nodes[j].l = 0;
s.nodes[j].r = 0;
s.nxt_avail = j+1;
L10: /* compare */
if (s.nodes[i].a < x)
  if (s.nodes[i].l != 0) {
    i = s.nodes[i].l;
    goto L10;
  }
else {
  s.nodes[i].l = j;
  goto L20;
}
else {
  if (s.nodes[i].r != 0) {
    i = s.nodes[i].r;
    goto L10;
  }
else {
    s.nodes[i].r = j;
    goto L20;
  }
}
L20: /* insert */
s.nodes[j].a = x;
s.nodes[j].l = 0;
s.nodes[j].r = 0;
s.nxt_avail = j+1;
L10: /* compare */
    if (s.nodes[i].a < x)
        if (s.nodes[i].l != 0) {
            i = s.nodes[i].l;
            goto L10;
        } else {
            s.nodes[i].l = j;
            goto L20;
        }
    else {
        if (s.nodes[i].r != 0) {
            i = s.nodes[i].r;
            goto L10;
        } else {
            s.nodes[i].r = j;
            goto L20;
        }
    }
L20: /* insert */
    s.nodes[j].a = x;
    s.nodes[j].l = 0;
    s.nodes[j].r = 0;
    s.nxt_avail = j+1;

<table>
<thead>
<tr>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shladd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Shladd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Ld a</td>
<td>Ld l</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Cmp p9, p8= a&lt;x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt;p9&gt; cmp p7,p5= l!=0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
L10: /* compare */
if (s.nodes[i].a < x)
    if (s.nodes[i].l != 0) {
        i = s.nodes[i].l;
        goto L10;
    } else {
        s.nodes[i].l = j;
        goto L20;
    }
else{
    if (s.nodes[i].r != 0) {
        i = s.nodes[i].r;
        goto L10;
    } else{
        s.nodes[i].r = j;
        goto L20;
    }
}
L20: /* insert */
    s.nodes[j].a = x;
    s.nodes[j].l = 0;
    s.nodes[j].r = 0;
    s.nxt_avail = j+1;

Unit 1  Unit 2  Unit 3  Unit 4
1    shladd
2    Shladd
3    add    add
4    Ld a    Ld l
5    Cmp p9,
p8= a<x
6    <p9> cmp
    p7,p5= l!=0
7    <p7> mov
L10: /* compare */
if (s.nodes[i].a < x)
    if (s.nodes[i].l != 0 ) {
        i = s.nodes[i].l;
        goto L10;
    }
else {
    s.nodes[i].l = j;
    goto L20;
}
else{
    if (s.nodes[i].r != 0) {
        i = s.nodes[i].r;
        goto L10;
    }
else{
        s.nodes[i].r = j;
        goto L20;
    }
}
L20: /* insert */
s.nodes[j].a = x;
s.nodes[j].l = 0;
s.nodes[j].r = 0;
s.nxt_avail = j+1;

<table>
<thead>
<tr>
<th></th>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shladd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Shladd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add</td>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Ld a</td>
<td>Ld l</td>
<td>Ld r</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Cmp p9, p8= a&lt;x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt;p9&gt; cmp p7, p5= l!=0</td>
<td>&lt;p8&gt; cmp p6, p4 r!=0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>&lt;p7&gt; mov</td>
<td>&lt;p6&gt; mov</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Compiling the “then” paths

```c
if (s.nodes[i].a < x) {
    if (s.nodes[i].l != 0) {
        i = s.nodes[i].l;
        goto L10;
    }
}
else {
    if (s.nodes[i].r != 0) {
        i = s.nodes[i].r;
        goto L10;
    }
}
```

<table>
<thead>
<tr>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shladd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Shladd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add</td>
<td>add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Ld a</td>
<td>Ld l</td>
<td>Ld r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Cmp p9, p8= a&lt;x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt;p9&gt; cmp p7,p5= l!=0</td>
<td>&lt;p8&gt; cmp p6, p4 r!=0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>&lt;p7&gt; mov i=l</td>
<td>&lt;p6&gt; mov i=r</td>
<td>branch next_loop</td>
<td>branch next_loop</td>
<td></td>
</tr>
</tbody>
</table>
Compiling the “then” paths

```c
if (s.nodes[i].a < x)
    if (s.nodes[i].l != 0 ) {
        i = s.nodes[i].l;
        goto L10;
    }
else{
    if (s.nodes[i].r != 0) {
        i = s.nodes[i].r;
        goto L10;
    }
}
```

<table>
<thead>
<tr>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shladd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Shladd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add</td>
<td>add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Ld a</td>
<td>Ld l</td>
<td>Ld r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Cmp p9, p8= a&lt;x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt;p9&gt; cmp p7, p5= l!=0</td>
<td>&lt;p8&gt; cmp p6, p4 r!=0</td>
<td>&lt;p9&gt; mov i=l</td>
<td>&lt;p8&gt; mov i=r</td>
<td>&lt;p6&gt; branch next_loop</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Using predicate promotion to shorten the loop path
L10: /* compare */
    if (s.nodes[i].a < x)
        if (s.nodes[i].l != 0 ) {
            i = s.nodes[i].l;
            goto L10;}
    else {
        s.nodes[i].l = j;
        goto L20;}
else{
    if (s.nodes[i].r != 0) {
        i = s.nodes[i].r;
        goto L10;}
    else{
        s.nodes[i].r = j;
        goto L20;}
}  
L20: /* insert */
    s.nodes[j].a = x;
    s.nodes[j].l = 0;
    s.nodes[j].r = 0;
    s.nxt_avail = j+1;

Then paths are scheduled
Else paths are not
Compiling the “else” paths

```plaintext
else {
  s.nodes[i].l = j;
  goto L20;
}

else{
  s.nodes[i].r = j;
  goto L20;
}
```

<table>
<thead>
<tr>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shladd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Shladd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add</td>
<td>add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Ld a</td>
<td>Ld l</td>
<td>Ld r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Cmp p9, p8= a&lt;x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt;p9&gt; cmp</td>
<td>&lt;p8&gt; cmp</td>
<td>&lt;p9&gt; mov</td>
<td>&lt;p8&gt; mov</td>
<td>&lt;p7&gt; branch</td>
</tr>
<tr>
<td></td>
<td>p7,p5= l!=0</td>
<td>p6, p4=r!=0</td>
<td>i=l</td>
<td>i=r</td>
<td>next_loop</td>
</tr>
<tr>
<td>7</td>
<td>&lt;p5&gt; store l</td>
<td>&lt;p4&gt; store r</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
## Compiling...

<table>
<thead>
<tr>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>shladd</td>
<td>shladd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shladd</td>
<td></td>
<td>shladd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>Ld a</td>
<td>Ld l</td>
<td>Ld r</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cmp p9, p8= a&lt;x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;p9&gt; cmp p7, p5= l!-0</td>
<td>&lt;p8&gt; cmp p6, p4 r!=0</td>
<td>&lt;p9&gt; mov i=l</td>
<td>&lt;p8&gt; mov i=r</td>
<td>&lt;p7&gt; branch next_loop</td>
<td>&lt;p6&gt; branch next_loop</td>
</tr>
<tr>
<td>&lt;p5&gt; store [l].l</td>
<td>&lt;p4&gt; store [l].r</td>
<td>Store [j].a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store [j].l</td>
<td>Store [j].r</td>
<td>Store nxt_avail</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

L20: /* insert */

```c
s.nodes[j].a = x;
s.nodes[j].l = 0;
s.nodes[j].r = 0;
s.nxt_avail = j+1;
```

Scheduling the else path does not slow the then paths
### Treeins Schedule

<table>
<thead>
<tr>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shladd</td>
<td>shladd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Shladd</td>
<td>shladd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>4</td>
<td>Ld a</td>
<td>Ld l</td>
<td>Ld r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Cmp p9, p8= a&lt;x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt;p9&gt; cmp p7, p5= l!-0</td>
<td>&lt;p8&gt; cmp p6, p4 r!=0</td>
<td>&lt;p9&gt; mov i=l</td>
<td>&lt;p8&gt; mov i=r</td>
<td>&lt;p7&gt; branch next_loop</td>
</tr>
<tr>
<td>7</td>
<td>&lt;p5&gt; store [I].l</td>
<td>&lt;p4&gt; store [I].r</td>
<td>Store [j].a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Store [j].l</td>
<td>Store [j].r</td>
<td>Store nxt_avail</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**All paths scheduled in parallel**
# Scheduled Without Predication

<table>
<thead>
<tr>
<th>Unit 1</th>
<th>Unit 2</th>
<th>Unit 3</th>
<th>Unit 4</th>
<th>Unit 5</th>
<th>Unit 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shladd</td>
<td>shladd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Shladd</td>
<td>shladd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>4</td>
<td>ld</td>
<td>ld</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Cmp p9, p8=</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt;p9&gt; cmp p7,p5=</td>
<td>&lt;p8&gt; cmp p6,p4</td>
<td>&lt;p9&gt; mov</td>
<td>&lt;p8&gt; mov</td>
<td>&lt;p7&gt; branch next_loop</td>
</tr>
<tr>
<td>7</td>
<td>&lt;p5&gt; store</td>
<td>&lt;p4&gt; store</td>
<td>store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>store</td>
<td>store</td>
<td>store</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2 difficult to predict branches

Separate paths
Branches are predicted.

Instructions are speculated along one path.

All instructions are thrown away if branches are mispredicted.

Mispredicts have significant performance penalties.
IA-64 processor executes instructions along 4 control flow paths in parallel

- advantage over Out of Order Execution which can only follow one path
- take advantage of a wide machine

IA-64 architecture enables significant performance advantages
Comparing Bills

With speculation and predication:
- Schedule: 6 or 8 cycles
- Branch: 2 branches
- Mispredict Penalty: 25% mispredict rate
- 7 cycles

Without speculation nor predication:
- Schedule: 7 cycles
- 2 branches
- 2 clocks
- 4 cycles
- 11 cycles
With speculation and predication:
- **xlxgetvalue**: 2 cycles per iteration
- **puzzle**: 2.5 cycles per iteration
- **treeins**: 7 cycles per iteration

Without speculation nor predication:
- **xlxgetvalue**: 3 cycles per iteration
- **puzzle**: 4.5 cycles per iteration
- **treeins**: 11 cycles per iteration
IA-64 at Work Summary

Speculation and predication benefits

- Pointer chasing: Execute loads before pointer safety check
- Loop with conditional: Unroll loop, remove hard to predict branch
- If-then-else: Execute all paths in parallel, Eliminate branch mispredict penalties
IA-64 Architecture Innovations

Outline

- Background
- IA-64 at work: Code Examples
- Summary
  - What IA-64 architecture means for developers
    - New levels of performance
    - Wider machines: better utilization
      and inherent scalability
IA-64 Architecture Benefits

Summary

- Explicit parallelism leads to greater ILP
- Efficient use of large resources
- Benefits over out-of-order execution
  - Exposes greater parallelism through compiler’s larger scheduling scope
  - Executes multiple paths in parallel by removing branches & mispredicts
  - Enables parallel execution by minimizing memory latency impact

IA-64: Enabling Industry Leading Performance for Server and Workstation Workloads