Part III: Compiler Backend Technology on IA-64

Jesse Fang
Microprocessor Research Lab
Overview

- Predication
  - If-conversion
    - Uses regular, unconditional and parallel compares
- Software pipelining
  - Modulo scheduling and rotating register allocation
    - Uses rotating registers, stage predicates, loop branches
- Global scheduling
  - Instruction scheduling cross basic block boundaries
    - Uses control and data speculation, predication, post-increments, multiway branches
- Global register allocation
  - Allocate registers for predicate code
    - Uses register stack, ALAT associativity
- Driven by information provided by machine model
(1) Predication

- When branch misprediction rate is high, it is better to predicate
- Predication creates more ILP
- Predication has the potential cost of increasing the critical path length

Techniques

- If-conversion
- Parallel compare to reduce control height
If-conversion for Predication

- Identifying region of basic blocks based on resource requirement and profitability (branch miss rate, miss cost, and parallelism)
- Result: a single predicated basic block

```
a < b

s = s + a  \rightarrow  s = s + b

*p = s

cmp.lt p1,p2=a,b
(p1)  s = s + a
(p2)  s = s + b
     *p  = s
```
Reducing Control Height with parallel compares

- Convert nested if’s into a single predicate
- Result: shorter control path by reducing the number of branches

\[
\begin{align*}
a < b & \quad \text{Y} \\
b < c & \quad \text{N} \\
s1 & \\
\text{p1=0} \\
\text{cmp.lt.or p1,p0=a,b} \\
\text{cmp.lt.or p1,p0=b,c} \\
(p1) & \text{br s1} \\
s2 & \text{N}
\end{align*}
\]
Multiway Branch Example

- Use Multiway branches
  - Speculate compare (i.e. move above branch)
  - Do not reduce number of branches
  - Avoid predicate initialization

If \((a > b \land b > c)\)
then \(Y\)
else \(X\)

```plaintext
cmp.le p1,p0=a,b
cmp.le p2,p0=b,c
(p1) br X
(p2) br X
Y
```

---

Hot Chips tutorial 1999
If-Conversion Algorithm

Procedure assign-predicate-to-bblock (bblock-list G) {
    rearrange G in Breadth First Order;
    create post-dominate list P;
    set T to head block with order # 0;
    for (each bblock BB of G in forward order) {
        empty post-dominate list P;
        add all BB’s predecessors to P;
        for (each bblock BP of P in backward order of G) {
            if (BB post-dominate BP)
                if (BP dominate BB) BB.predicate = BP.predicate
                else add all BP’s predecessors to P instead of BP;
            else { BB.predicate = new (predicate);
                if (BB post-dominate true-successor of BP)
                    add BB.predicate into BP’s true-list
                else add BB.predicate into BP’s false-list;
            }
        }
    }
}
Example of If-Conversion

BB0: br0

BB1: br1

BB2:

BB3:

BB4: br4

BB5:

BB6:

BB7:

T:

P1: P2:

(P1) (P2,P4)

P3: P4:

(P4)

P5:

(P5)

P6:

(P6)
(2) Software Pipelining

- Exploit parallelism across iterations without code bloat
- Architectural features used:
  - Rotating registers
  - Rotating predicates (stage predicates)
  - Predication to collapse control flow
  - Counted loop
  - While loop
- Modulo Scheduling
- Rotating Register Allocation
Overlapping Loop Iterations

- Exploit parallelism across loop iterations
- Result: kernel-only code without code expansion

L1: ld4 r4 = [r5], 4; // 0
    add r7 = r4, r9; // 2
    st4 [r6] = r7, 4; // 3
    br.cloop L1;;

Prolog
Kernel
Epilog

L1:
(p16) ld4 r32 = [r5], 4 // Cycle 0
(p18) add r35 = r34, r9 // Cycle 0
(p19) st4 [r6] = r36, 4 // Cycle 0
    br.ctop L1;
# Rotating Registers

- Register rotation provides an automatic renaming mechanism
- Rotating predicate registers unify prolog, kernel and epilog

<table>
<thead>
<tr>
<th>Iteration 1</th>
<th>r32 r33 r34 r35 r36</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p16 p17 p18 p19 p20</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration 2</th>
<th>r33 r34 r35 r36 r32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p17 p18 p19 p20 p16</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration 3</th>
<th>r34 r35 r36 r32 r33</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p18 p19 p20 p16 p17</td>
</tr>
<tr>
<td></td>
<td>1 0 0 1 1</td>
</tr>
</tbody>
</table>

$L1$:
- `(p16) ld4 r32 = [r5], 4 // Cycle 0`
- `(p18) add r35 = r34, r9 // Cycle 0`
- `(p19) st4 [r6] = r36, 4 // Cycle 0`
- `br.cstop L1;`

**P16 set automatically**
Control Speculation

- Hoist loads and computation above branches
- Result: improved software pipelining for loops with multiple exits and for while-loops

```
for i = 1, M
load;
if (P) exit;
end_for
```

No control speculation

Control speculation
Software Pipelining Example

for (j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++)
;

---------------------------
B6: { .mmi
    add r34=1,r35 //0
    add r32=1,r33 //0
    nop.i 0 ;;
} { .mii
    ld1.s r33=[r34] //1
    (p19) sxt1 r35=r39 ;; //6
    (p19) cmp4.eq p16,p0=r35,r38 ;;/7
} { .mii
    (p16) cmp.ne.unc p19,p0=r0,r0 //3
    (p16) chk.s r33,.B23 //3
    (p16) cmp4.eq.unc p0,p18=r33,r0 ;;/3
}
.B24: { .mii
    (p18) ld1 r38=[r32] //4
    (p16) add r35=1,r36 //4
    (p18) sxt1 r37=r33 //4
} { .mfb
    nop.m 0
    nop.f 0
    (p16) br.wtop.dptk .B6;; //4
------------------------RECOVERY CODE -------------------
.B23:
    ld1 r33=[r34] ;; //0
    (p16) cmp4.eq.unc p0,p18=r33,r0 //2
    br.cond.sptk .B24 ;; //2

• Uses if-conversion to remove control flow within loop
• Side exits removed using predication
• Control speculation used to improve overlap between iterations (reduce II)
• Data speculation used to reduce recurrences caused by unlikely memory dependencies
• Post-increments materialized to reduce resource II
(3) Global Code Scheduling

- **Objective**
  - Increase parallelism
  - Remove unnecessary dependencies
  - Fully use machine width

- **Needs**
  - Accurate machine model

- **Uses architectural features**
  - Large number of registers
  - Control and data speculation, checks and recovery code
  - Multiway branches
Region Formation

- Scheduling Regions are acyclic

{A,C,D} a nested region as NR
Speculative Upward Code Movement

- Speculate both the load and the use
- Result: efficient use of machine resources
Predicated Upward Code Movement

- Predicate with fall-through predicate
  - motion bounded by compare
- Result: predication can avoid speculative side effects

```
cmp.unc.eq p1,p2 = r1,r2
  :
  (p1) br label
    :
    ld   r4 = [r3]
    add  r4 = r4,1

  (p2)  br  label
  :
  (p2)  ld    r4 = [r3]
  (p2)  add  r4 = r4,1
```

```
cmp.unc.eq p1,p2 = r1,r2
  :
  (p2)  ld    r4 = [r3]
  (p2)  add  r4 = r4,1
  :
  (p1)  br  label
```
Example of Instruction Scheduling
- Control Flow Graph

struct dyn-array {
    int *x;
    int count;
}
dyn-array *p;

If( n < p->count ) {
    (*log)++;
    return p->x[n];
} else {
    return 0;
}
Example of Instruction Scheduling
- with predication and possible speculation

- If-conversion to generate predicates
- During dependence graph construction, potentially control and data speculative edges and nodes are identified
- Check nodes are added where possibly needed (note that only data speculation checks are shown here)
Example of Instruction Scheduling
- ready for scheduling

Speculative edges may be violated. Here the graph is re-drawn to show the enhanced parallelism.

Note that the speculation of both writes to the out0 register would require insertion of a copy. The scheduler must consider this in its scheduling.

Nodes with sufficient slack (e.g. writes to out0) will not be speculated.
Downward Code Movement

Predication enables downward code movement from A to C without compensation code in B

Use predication to merge sparse code in compensation block with code in merge block
### Code Motion Tradeoffs

**Downward Code Motion**

- Slots available in hot path
- Predication can pull instructions from lower weight path

**Upward Code Motion**

- Scheduler can move instructions from above and below
Multiple Branches in Single Cycle

- Multiway branches: Speculate compare to get predicates ready
- Result: processing multiple branches in single cycle

```
cmp.lt p1,p0=a,b
cmp.lt p2,p0=b,c
(p1) br s1
(p2) br s2
s3
```
Global Code Scheduling Design

Before:

- Move code above branches using predication and/or speculation
- Move code below branches using predication
- Move loads and uses across stores that are unlikely to interfere and generates checks and recovery
- Combine memory references and address increments to generate post-increments if r6 and r7 only have life-range in the blocks

```
ld r5= [r6]  
r7= r6+4  
st [r7]= br
```

```
ld.s r5 =r6,4     
br
```

```
(r1) chk.s r5  
(r1) st [r6]= br
```

After:
(4) Global Register Allocation

● **Objective**
   - Eliminate memory references. Minimize register spill and copying after load-store elimination (virtual registers)

● **Incorporating architectural features**
   - Large number of registers
   - Predication
   - Data speculation
Register Allocation Example

- Modeled as a graph-coloring problem
  - Nodes in the graph represent live ranges of variables
  - Edges represent a temporal overlap of the live ranges
  - Nodes sharing an edge must be assigned different colors (registers)

\[
\begin{align*}
  x &= \ldots \\
  y &= \ldots \\
  &= \ldots x \\
  z &= \ldots \\
  &= \ldots y \\
  &= \ldots z
\end{align*}
\]
Register Allocation Example

With Control Flow

ReQUIRES TWO COLORS
Register Allocation Example

With Predication

Now Requires Three Colors
Predicate Analysis for the Example

$p_0$

$p_1$ and $p_2$ are disjoint
If $p_1$ is TRUE, $p_2$ is false and vice versa

$x = \ldots$
$y = \ldots$
$z = \ldots$

$p_1$
$p_2$

$x = \ldots y$
$x = \ldots$
$= \ldots z$
$= \ldots x$
Register Allocation Example

With Predicate Analysis

\[
\begin{align*}
x &= \ldots \\
y &= \ldots \\
z &= \ldots \\
(p2) &= \ldots y \\
(p1) &= x = \ldots \\
(p1) &= \ldots z \\
(p2) &= \ldots x
\end{align*}
\]

Now Back to Two Colors
Effect of Predicate-Aware Register Allocation

- Reduces register requirements for individual procedures by 0% to 75%
  - Depends upon how aggressively predication is applied
- Average dynamic reduction in register stack allocation for gcc is 4.7%

\[
\begin{align*}
(p1) & \ v1 = 10 \\
(p2) & \ v2 = 20 ;; \\
(p1) & \ st4 [v10] = v1 \\
(p2) & \ v11 = v2 + 1 ;; \\
(p1) & \ r32 = 10 \\
(p2) & \ r32 = 20 ;; \\
(p1) & \ st4 [r33] = r32 \\
(p2) & \ r34 = r32 + 1 ;; \\
\end{align*}
\]

overlapped live ranges

same register for v1 and v2
Minimize ALAT Conflict

- Assign registers to the live ranges of the advanced loads to eliminate possible conflict in ALAT (Advanced Load Address Table)
(5) Modeling The Machine

Objective: Provide micro-architecture information to the rest of the compiler

- Machine characteristics read from microarchitecture description file, which is used by simulator and other tools.
Modeling The Machine

- **Objective**
  - Map instruction to resources as instructions are scheduled. Manage machine resources.
  - Help in the scheduling of the backend of the pipeline (after decoupling buffer).
  - Handle the bundling details, coupled with dispersal knowledge.
  - Decide on mid bundle stop bits based on:
    - resource requirements
    - code size

- Details of the machine are abstracted away for the rest of the compiler.
Summary

• Compiler is critical for IA-64 performance
  ➢ backend should take full advantages of IA-64 architectural features to generate optimal code

• If-conversion
  ➢ predication with various compares

• Software pipelining
  ➢ rotating registers, stage predicates, loop branches

• Global scheduling cross basic block boundaries
  ➢ control and data speculation, post-increments, multiway branches,

• Global register allocation
  ➢ register stack, ALAT associativity

• Driven by information provided by machine model
Part IV: Dynamic Optimization Technology on IA-64

Jesse Fang
Microprocessor Research lab
IA-64 Architecture Advantages for Java

• Java has more method invocations than C/C++ function calls
  ➢ IA-64 has more registers and register stack engine
• Java has smaller basic blocks in methods
  ➢ IA-64 has predication and speculation
• Garbage collection has write barrier as bottleneck
  ➢ IA-64 has more registers and predication
• Java has exception handling functionality
  ➢ IA-64 recovery code mechanism can handle it very well
• Jini requires large “name space” for Java
  ➢ IA-64 has 64-bit address
IA-64 Java Software Convention

- JIT generated code follows IA-64 software conventions including
  - parameter passing
  - memory stack management
  - general register usage guidelines

- JVM may reserve extra registers within IA-64 register usage guidelines for memory and thread management, which should be allocated
  - from r4 up for preserved registers
  - from r14 up for scratch registers
  - JVM cannot assume that native code abides by the additional Java register usage restrictions
Java JVM/JIT Design on IA-64

- Dynamic profiling information
  - not only branch frequency but also info from performance monitor registers
- Dynamic optimization JIT for hot regions
  - not only hot methods but also hot basic blocks
  - book-keeping in JVM
- Light-weight optimization algorithms
  - speculation in global code motion
  - predication in if-conversion
- Garbage collection on IA-64
  - memory hierarchy management
  - preserve registers for write-barrier (read-barrier)
- Java virtual machine on IA-64
  - 64-bit pointer
Object-Oriented Code on Merced

Challenges

- Small Procedures, many indirect (virtual)
  - Limits size of regions, scope for ILP
- Exception Handling
- Bounds Checking (Java)
  - Inherently serial - must check before executing load or store
- Garbage collection

Solutions

- Inlining
  - for non-virtual functions or provably unique virtual functions
  - Speculative inlining for most common variant
- Dynamic optimization (Java)
  - Make use of dynamic profile
- Speculative execution
  - Guarantees correct event behavior
- Liveness analysis
  - Architectural support for speculation ensures recoverability
  - More register preserved for GC
Dynamic Optimizations for C/C++

- Profiling information is not always ready for static compiler
  - it relies on input data files sometimes
  - not all ISVs would like to use profiling
- Advantages of IA-64 to support dynamic profiling
  - not only collect branch frequency but also info from performance monitor registers
- Dynamic optimizations focus on
  - cache missing
  - branch misprediction
- Various models for dynamic optimizations
  - on-line profiling collection
  - off-line (or on-line) optimizations
- More challenging for C/C++ than for Java
Summary

● Java on IA-64
  - IA-64 advantages for Java
  - IA-64 Java software convention
  - JVM/JIT design on IA-64

● Object-Oriented code on IA-64
  - virtual procedure calls
  - precise exception handle
  - garbage collection

● Take advantages of IA-64 architecture for dynamic optimizations