EPIC Architectures and Compiler Technology

Wen-mei Hwu

Department of Electrical and Computer Engineering
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

IMPACT Compiler Group
http://www.crhc.uiuc.edu/IMPACT/
Microprocessor Vision 1999

• UI/HP/Intel research coalition formed in 1992
• Goals
  – ILP architecture features to greatly increase productive IPC
    • eliminate branches and control dependences
    • support software to approach global scheduling
    • control code size explosion
  – Adaptive management of cache and bus hierarchy
    • reduce memory related stall cycles
  – Support software migration
    • support fine-grain mixture of old and new code
  – No compromise on cycle time
Microprocessor Microarchitecture

**Main Memory**
- 100 cycle latency

**System Bus**

**L2 Cache**
- non-blocking
- 1M-byte, 4-way
- 64B block

**Backside Bus Interface**

**I-Fetch Unit**
- I-Cache
  - 32K-byte
  - direct-mapped
  - 64B block (split)

**Instruction Decoder**
- 32K-byte
- direct-mapped
- 64B block (split)

**Register Alias Table**
- (64 predicate and 128 regular)

**BTB**
- 1K
- direct-mapped
- 2-level

**L1 Cache**
- non-blocking
- 16K-byte, 2-way
- 32B block

**2 Memory Units**

**2 Integer Units**

**1 Branch Unit**

**1 Floating Point Unit**

**2 M_e_m_o_r_y U_n_i_t_s**

**1 Floating Point Unit**

**1 Floating Point Unit**

**1 Floating Point Unit**
EPIC Compiler Technology Overview

Source

- If-Conversion
- Classical Optimization
- Predicate Optimization
- ILP Optimization
- Scheduling/Partial Reverse If-Conversion

Debugging of Optimized Code

- Register Allocation
- Code Generation
- Memory Disambiguation
- Predicated Dataflow
- Predicate Analysis
- Machine Description

Compiler Technology for EPIC Architectures
Wen-mei Hwu
Predicated Execution

• Conditional execution of instructions based on a Boolean source operand
• Execution model
  – Load r1, r2, r3 <p1>
  – If p1 is TRUE, instruction executes normally
  – If p1 is FALSE, instruction treated as NOP (with some exceptions)
• Provides compiler with an alternative to guarding instruction execution with branches
Instruction Set Support for Predicated Execution

- **Full Predication Support**
  - Predicate defining instructions
  - Full set of predicated instructions
  - Separate register file
  - Best performance

- **Partial Predication Support**
  - Limited set of predicated instructions added to existing ISA (CMOV, SELECT)
  - Brings some performance increase to existing ISA’s

- **Dynamic Predication Support**
  - No ISA change needed
  - Smallest performance gain
Predicate Defining Instructions
(HPL PlayDoh Spec)

\[
pred_{< \ cmp \ >} \ P1 < \ type \ >, P2 < \ type \ >, \ src1, \ src2 \ (\text{Pin})
\]

- \(< \ cmp \ >\) - comparison type: =, >, <, etc.
- \(< \ type \ >\)
  - Unconditional (U, U)
  - OR-type (OR, OR)
  - AND-type (AN, AN)
  - Conditional (C, C)
Unconditional Predicate Defines

- Handle blocks executed on one condition

```
if (a < 10)
c = c + 1
else
  if (b > 20)
d = d + 1
  else
e = e + 1
```

```
pred_ge p1(U), p2(U), a, 10
add c, c, 1 (p2)
pred_le p3(U), p4(U), b, 20 (p1)
add d, d, 1 (p4)
add e, e, 1 (p3)
```

```
bge, a, 10, L1
  F
  T
  add c, c, 1
  jump L3
```

```
ble b, 20, L2
  F
  T
  add d, d, 1
  jump L3
```

```
L3:
```

```
p2
L1:
p1
L2:
p4
L3:
p3
```

```
<table>
<thead>
<tr>
<th>Pin</th>
<th>Comparison</th>
<th>Pout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Compiler Technology for EPIC Architectures
Wen-mei Hwu
**OR-type Predicate Defines**

- Handle blocks executed on multiple conditions

```c
if (a && b)
    c = c + 1;
else
    d = d + 1;
```

```c
pred_clr p1
pred_eq p1(OR), p2(\bar{U}), a, 0
pred_eq p1(OR), p3(\bar{U}), b, 0 (p2)
add c, c, 1 (p3)
add d, d, 1 (p1)
```

Pin Comparison OR OR

<table>
<thead>
<tr>
<th>Pin</th>
<th>Comparison</th>
<th>Pout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>- 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 -</td>
</tr>
</tbody>
</table>
**AND-type Predicate Defines**

- More efficiently handle blocks executed on multiple conditions

```c
if (a && b)
    c = c + 1;
else
    d = d + 1;
```

- `pred_clr p1`
- `pred_set p3`
- `pred_eq p1(OR), p3(AND), a, 0`
- `pred_eq p1(OR), p3(AND), b, 0`
- `add c, c, 1 (p3)`
- `add d, d, 1 (p1)`

<table>
<thead>
<tr>
<th>Pin</th>
<th>Comparison</th>
<th>Pout</th>
<th>AND</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Predication Benefits

• Eliminates branches in exchange for increased resource subscription and/or dependence height
  – Reduction in branch resource consumption
  – Reduction in total branch misprediction penalty

• Aggressive control flow transformations
  – Height reduction and aggressive branch motion
  – Logic minimization of programmatic decision sequence

• Aggressive optimizations in the presence of control flow
  – Simplifies static scheduling and optimization along multiple paths
  – Controls code size explosion, makes some optimizations feasible
Predicate-Domain Control Flow Transformation

- Predication allow general restructuring of control flow
- Predication allows significant decision height reduction
Path Height Reduction: Concept

- Path classes
  - dependence limited
  - resource limited
- Optimizations can be performed to exchange dependence height for resource usage
- Goal: balance resource height and dependence height to reduce effective height of path

Sequential Code

- Height goes from 6 to 2
- Operation count went from 10 to 14
- Extra operations absorbed by processor width

Saturated Code
Fully Resolved Predicates: Motivation

• Typical loops have many infrequently taken exit branches

• Infrequent exit branches
  – Impede code motion
  – Increase length of path to frequently taken branches
  – Consume valuable branch resources

• Goal: Use predication to enhance performance in the presence of easily predicted branches
Fully Resolved Predicates: Concept

- Partially Resolved Predicates (PRP)
  - Instruction execution is guarded by predicates or branches.
  - Some control dependencies remain in predicated code.
- Fully Resolved Predicates (FRP)
  - Instructions are guarded by predicates even if guarded by branches.
  - All control dependencies within the region are eliminated.
  - Any instruction can be hoisted above a branch without speculation.
FRP: Computation

Partially Resolved Predicates

Fully Resolved Predicates

Compiler Technology for EPIC Architectures
Wen-mei Hwu
FRP: Case Study

- grep function “execute” inner loop
- Segment accounts for about 40% of total execution time.
- Source:

```c
for (; ;) {
    if (p2 >= ebp)
        /* Excluded from Hyperblock */
        if ((c = *p2++) == '\n')
            break;
    if (c)
        if (p1 < &linebuf[1024-1])
            *p1++ = c;
}
```
Height Reduction by Predication and Speculation

Code example (grep execute)

- FRP Predication reduced height via control flow restructuring
- Data speculation reduced height via reordering of possibly conflicting loads and stores
Code Size Control using Predication

Code example (*MediaBench Experimental Image Compression* reflect1):
Original (Overhead=8/17 instrs (47%))  Optimized (6/19 (30%))  Predicated (3/13 (23%))

Code example (*099.go copyshape)*:

- Predication reduced code size by instruction merging (in example 35%)
Analysis of Predicated Codes

- Dataflow without regard to predicates can lead to conservative results.

- Live Variable Analysis Example:
  - Without Predicate Aware Dataflow (Only instructions on TRUE predicate can kill.)
    - R7 is defined and killed by instruction 5; R7 is used by instruction 6.
    - R7’s live range is (5,6).
    - R3 is not defined and killed by instruction 3 in all cases because it is predicated on P1. R3 is used by instruction 4.
    - R3’s live range is (1,2,3,4) and live out the top of the CB.
  - With Predicate Aware Dataflow
    - R7’s live range is also (5,6).
    - R3’s live range is (3,4) because instruction 3 defines R3 for all uses by instruction 4. This is known by studying the relation of P1 to P2.

1. $(p_{1_{un}}) = (r1 < 0)$
2. $(p_{2_{un}}) = (r2 < 0)$
3. $r3 = r4 + r5$ (p1)
4. $r8 = r3 + 1$ (p2)
5. $r7 = r4 + r6$
6. $r4 = r7 - 1$ (p1)
7. $r9 = r9 / 2$ (p2)

Dataflow without regard to predicates can lead to conservative results.
The Predicate Flow Graph

1. \((p_{1\text{un}}) = (r1 < 0)\)
2. \((p_{2\text{un}}) = (r2 < 0)\) (p1)
3. \(r3 = r4 + r5\) (p1)
4. \(r8 = r3 + 1\) (p2)
5. \(r7 = r4 + r6\)
6. \(r4 = r7 - 1\) (p1)
7. \(r9 = r9 / 2\) (p2)
Dataflow Analysis of Predicated Code

- Traditional dataflow requires reverse if-conversion (RIC)
- RIC of some codes is exponential ($w_c: 5, 20, 80, 240, ...$)
- Factoring reduces order of complexity ($w_c: 8, 15, 22, 28, ...$)
Compile-Time Memory Disambiguation

- Maximize the efficiency of the memory system
  - Eliminate unnecessary loads
  - Reorder loads past independent stores to hide the load latency
  - Instruct the hardware about the possible dependence between loads and stores to prevent run-time mis-speculation

- Indirect memory accesses through pointers
  - Dependence between *p and *q is not obvious

- Function side-effects
  - Analysis between *p and *q difficult when foo(&p, &q) is present

- Efficient and effective interprocedural alias analysis
  - Trade-off between accuracy and complexity
  - Comparable resolution for stack and heap objects
Example

f1() {
f3(s1, &i, &j);
*s1->p = 10;
i = *s1->q + i;
(*s1->fp)(s1);
}
f2() {
f3(s2, &j, &i);
*s2->p = 10;
i = *s2->q + i;
}
f3(s, v1, v2) {
s->p = v1;
s->q = v2;
s->fp = f5;
f4(s);
}
Interprocedural Points-to Analysis

- Flow-Insensitive function-level points-to templates
- Context-Sensitive exchange of function-level points-to templates
Object Elevation

• Report interprocedurally accessed callee objects to the caller

• Not all accessible objects are visible
  – Heap objects allocated in the callee
  – Indirectly accessed non-local variables

• Objects accessed in the callee and accessible in the caller are mapped to the caller with encoded object name

• Object names are encoded by the access path
  – *s => s*
  – s->p => s*.offset_of_p
  – s->p->q => s*.offset_of_p*.offset_of_q
Working Example - 132.ijpeg in SPEC95

- Contains 477 functions and 25,889 lines of code
- Spends 200 seconds and 18MB of memory in analysis
- 229 of 266 indirect call-sites are converted into direct ones
Compile-Time Memory Disambiguation

- Potential performance enhancements
  - Eliminates redundant loads (*s->fp)
  - Reorders loads past independent stores (*s->q and *s->p)
  - Prevents run-time mis-speculation (i and *s->p)

- Challenges of interprocedural pointer analysis
  - Maintaining both efficiency and accuracy
    - Flow-insensitive and context-sensitive
  - Providing comparable results for stack- and heap-pointers
    - Object elevation

- Working example - ijpeg in SPEC95
  - 477 functions and 25,889 lines of code
  - Analysis consumes 200 seconds and 18MB of memory
  - 229 of 266 indirect call-sites converted into direct ones
  - 30% performance improvement observed
Debugging optimized code

• Motivation
  – optimization becomes default when compiling EPIC code
  – software validation issue: what is debugged is what gets shipped

• Provide meaningful information without misleading users
  – truthful behavior
    • make the user aware of optimization effects and surprising outcomes
  – expected behavior
    • hide the effects of optimization
    • current focus of most research and development efforts
Basic idea of recovering expected behavior

- Unexpected behavior caused by
  - program states updated prematurely or too late
  - program states not available
- Basic idea
  - suspend the execution early
  - control the execution of all the instructions necessary for the recovery (*forward recovery*)
  - compile required program states

\[
\begin{align*}
S_1: & \quad a = b + c \\
\text{Suspend execution} & \\
S_2: & \quad x = 2 \\
S_3: & \quad y = z \times 3
\end{align*}
\]

- \(i_1: \text{ld } r1, b\)  
- \(i_2: \text{ld } r2, c\)  
- \(i_3: \text{ld } r5, z\)  
- \(i_4: \text{mul } r6, r5, 3\)  
- \(i_5: \text{mov } r4, 2\)  
- \(i_6: \text{add } r3, r1, r2\)
Issues need to be addressed

- When to take over execution and when to stop forward recovery?
  - original execution order of instructions has to be tracked
  - instructions might be moved up to different paths leading to the breakpoint
  - or down to different paths starting from the breakpoint
Issues need to be addressed (continued)

• How does the debugger confirm a source breakpoint?
  – some object locations which are control equivalent to the breakpoint need to be identified
  – boolean conditions have to be incorporated sometimes

• How does forward recovery work?
  – executing everything or selectively
  – breakpoints and exceptions need to be reported in the expected order

• Where are the locations of variables at run-time?
  – run-time location of a variable may vary or not exist at all at different points of the program
Summary of a new debugging paradigm

• The compiler needs to preserve and maintain (besides the traditional debugging information)
  – original execution order of instructions
  – source statement instance information
  – breakpoint confirmation information
  – variable run-time location information

• The debugger needs to determine (using the above information)
  – when to suspend the normal execution
  – what instructions should be executed
  – where to find the variable values
  – how to ensure the program behavior consistent with what the user expects
Debugging of Optimized Code

- Increased importance due to EPIC
  - optimization essential in EPIC code
  - need to debug software while under test
- Solution must not mislead users
  - expected behavior or truthful behavior
- Keys to providing expected behavior
  - mappings between source breakpoints and object code locations
  - tracking run-time locations of variables
  - recovery of the expected variable values

\[ I_1'(S_4) \]
\[ I_1(S_1) \]
\[ I_2(S_2) \]
\[ I_3(S_3) \]
\[ I_4(S_3) \]
\[ I_1''(S_1,S_4) \]
\[ I_2(S_2) \]
\[ I_3'(S_3) \]
\[ I_4(S_3) \]
Outlook

• Compilers critical to the performance of EPIC uP’s
  – Use of predication and speculation is a serious challenge
  – Any misuse will lead to performance loss.
  – Brand new algorithms will be deployed in the EPIC compilers.
  – Existing software development models must be supported.

• Expect performance robustness issues
  – Awesome performance leap seen for some applications.
  – Less for others due to limitations of analyses and optimizations.
  – It can take years for the performance gain to be universal.

• Evolution of EPIC architectures
  – Revisions of architectures are likely as compilers mature.
  – Code size and power consumption are critical for embedded EPICs.