Architectures

Compiler Optimization for Modern VLIW/EPIC
Introduction

Instruction scheduling, explicit execution, explicit cache management, advanced optimizations include software pipelining, speculative execution, explicit cache management, advanced instructionscheduling. Nigel. Sharc DSP processor. Also several processors for embedded systems. Intel IA-64 (Itanium), HP Lab's HPL-PD. We will concentrate on VLIW/EPIC architectures. New architectures have hardware features for supporting a range of compiler optimizations.
Lecture 14: Instruction-Level Parallelism

A. Pnueli

VLIW/EPIC Architectures

Very Long Instruction Word (VLIW)

VLIW + New features

Explicitly Parallel Instruction Computing (EPIC)

Predication, rotatign registers, speculations, etc.

Specified completely by compiler (unlike superset scalar machines)

Processor can initiate multiple operations per cycle

This presentation will use the instruction syntax of the HP Labs' HPL-PD. The features of the Intel IA-64 are similar.

Honors Compilers, NYU, Spring 2007

338
Control Speculation Support

What about exceptions?

- Generally occurs due to code motion across conditional branches
- May not have been executed in an unoptimized code.

Control Speculation

Control Speculation Support

These instructions are speculative.

Safe if the effect of the speculative instruction can be ignored or undone if the other branch is taken.

Control Speculation

Control Speculation Support
Speculative Operations

Speculative operations are written identically to their non-speculative counterparts, but with an "E" appended to the operation name.

E.g. DIVE ADDDE PBRF
Speculative Operations (Example)

Here is an optimization that uses speculative instructions:

By the ADD instruction:

- If a divide-by-zero occurs, an exception will be raised.
- The effect of the DIV latency is reduced.

```
v1 = DIV v1, v2
v3 = ADD v1, 5

::: :::
::: 

v1 = DIV v1, v2
v3 = ADD v1, 5
```

Honors Compilers, NYU, Spring, 2007

341
In HPL-PD, most operations can be predicated.

A compare-to-predicate operation:

\[ p_1 = \text{CMPP} \rightarrow r_4, r_5 \]

The values of predicate registers are typically set by

- If the predicate register contains 0, the operation is not performed.
- If the predicate register contains non-zero value, the operand is an extra operand that is a one-bit predicate register.

They can have an extra operand that is a one-bit predicate register.

In HPL-PD, many operations are predicated.
Uses of Predication

- Height reduction of control dependencies
- \emph{e.g.} hyper-blocks
- With more complex compare-to-predicate operations
- A use of predication is to aid code motion by instruction scheduling
- \emph{If}-conversion, in its simplest form, is used with

A. Pnueli
If-conversion replaces conditional branches with predicates operations. For example, the code generated for

\[
\text{if } (d < e) \text{ then } f = e; \text{ else } f = e;
\]
\[
\text{if } (a > b) \text{ then } c = a; \text{ else } c = b;
\]

might be the two EPIC instructions:

| p1 = CMPP. < a,b | p2 = CMPP. = a,b | p3 = CMPP. > a,b | p4 = CMPP. = d,e |

\[c = 4 \text{ if } p1\]
\[c = b \text{ if } p2\]
\[f = d \text{ if } p3\]
\[f = e \text{ if } p4\]
Compare-to-Predicate Instructions

In previous slide, there were two pairs of almost identical instructions (conditional, or, and)

- HPL-PD provides two output CMPP instructions
  - $p_1, p_2 = \text{CMPPW} \cdot \langle \text{UNUC}, r_1, r_2 \rangle$

- Just computing complement of each other

- In previous slide, there were two pairs of almost identical instructions

- There are other possibilities (conditional, or, and)

- $U$ means unconditional, $N$ means normal, $C$ means complement.

A. Pnueli

Honors Compilers, NYU, Spring, 2007
If-Conversion, Revisited

Thus, using two-output CMP instructions, the code might instead be:

\[
\begin{array}{c|c|c}
\text{if } (d > e) & \text{else } & \text{else } \\
\text{if } & f & f \\
\text{if } & d & p_3 \\
\text{if } & c = q & p_2 \\
\text{if } & c & p_1 \\
\text{if } & a & p_4 \\
p_1, p_2 = \text{CMP}p=W.A.U, \text{ UC } d, e \\
p_3, p_4 = \text{CMP}p.W.\text{UC } q, b
\end{array}
\]

For example, the code generated for
\[
\text{if } (a > b) \text{ else } c = b; \text{ else } c = a;
\]

The advantage over the previous scheme is that now we have only 2 CMP instructions in the first EPIC command.

Thus, using two-output CMP instructions, the code generated for
\[
\text{if } a < b \text{ else } c = a; \text{ else } c = b;
\]

\textit{If-Conversion, Revisited}
Hyper-Block Formation

In hyper-block formation, if-conversion is used to form larger blocks of operations than the usual basic blocks.

Tail duplication is used to remove some incoming edges in middle of block.

Larger blocks provide a greater opportunity for code motion to increase ILP.

If-conversion is applied after tail duplication.

Tail duplication is used to remove some incoming edges in middle of block.

In hyper-block formation, if-conversion is used to form larger blocks of operations than the usual basic blocks.
HPL-PD’s memory hierarchy is unusual in that it is visible to the compiler.

The HPL-PD memory hierarchy is unusual in that the assumed latencies will be correct with reasonable expectations that the assumed latencies will be correct.

This supports static scheduling of load/store operations in which cache the data should be left.

In load instructions, the compiler can specify in which cache the data is expected to be found and in which cache the data should be placed.

In store instructions, the compiler can specify in which cache the data should be placed.

The HPL-PD memory hierarchy is unusual in that it is visible to the compiler.
Memory Hierarchy

- Replacement mechanism:
  - Doesn’t require sophisticated cache.
  - Doesn’t require data independent of the first-level cache.
  - Used to store large amounts of cache-polluting data.

- Data Prefetch Cache

- First-level cache

- Second-level cache

- Main Memory

- CPU/Regs
Load/Store Instructions

- Load Instruction: \( r_1 = L.W.C_2, V_1 \) where:
  - \( C_2 \) specifies the source cache
  - \( V_1 \) specifies the target cache
  - \( r_2 \) operand register (contains memory address)

- Store Instruction: \( S.W.C_1, r_2, r_3 \) where:
  - \( C_1 \) specifies the target cache
  - \( r_2 \) contains memory address
  - \( r_3 \) value to be stored

- Load/Store Instructions

What if source cache specifier is wrong?

- **Compilers**, NYU, Spring, 2007
Here’s a desirable optimization (due to long load latencies):

\[
\begin{align*}
    r_1 &= L \ r_2 \\
    \ldots \\
    S \ r_3,4 \\
    S \ r_1 &= ADD \ r_1,7 \\
    r_1 &= L \ r_2 \\
    S \ r_3,4 \\
    S \ r_1 &= ADD \ r_1,7
\end{align*}
\]

However, this optimization is not valid if the load and store reference the same location, i.e., if r2 and r3 contain the same address.

HPL-PD solves this by providing run-time memory disambiguation.
Run-Time Memory Disambiguation (cont)

HPL-PD provides two special instructions that can replace a single load instruction:

speculative load:
\[ r_1 = \text{LDS} \ r_2 \]

speculative load:
\[ r_1 = \text{LDS} \ r_2 \]

Otherwise, it's a no-op.

If so, the new load is issued and the pipeline stalls.

HPL-PD provides two special instructions that can replace a single load instruction:

speculative load:
\[ r_1 = \text{LDS} \ r_2 \]

speculative load:
\[ r_1 = \text{LDS} \ r_2 \]

HPL-PD provides two special instructions that can replace a single load instruction:

speculative load:
\[ r_1 = \text{LDS} \ r_2 \]

speculative load:
\[ r_1 = \text{LDS} \ r_2 \]

speculative load:
\[ r_1 = \text{LDS} \ r_2 \]

speculative load:
\[ r_1 = \text{LDS} \ r_2 \]

speculative load:
\[ r_1 = \text{LDS} \ r_2 \]
Since the LDS to the same memory location branching to compensation code if a store has occurred there is also a BRDV (branch-on-data-verify) for

The previous optimization becomes

Run-Time Memory Disambiguation (cont)
Dependence Analysis

Three types of dependence:

- Output
  \[ \cdot \cdot \cdot = X \]
  \[ \cdot \cdot \cdot = X \]

- Anti
  \[ \cdot \cdot \cdot = X \]
  \[ \cdot \cdot \cdot = X \]
  \[ \cdot \cdot \cdot = X \]

- True/Flow
  \[ \cdot \cdot \cdot = X \]
  \[ \cdot \cdot \cdot = X \]
  \[ \cdot \cdot \cdot = X \]

Determines if the relative order of two operations in the original (sequential) program must be preserved in the optimized version. Including software pipelining, loop optimizations, and foundation of instruction reordering optimizations.
Dependence Analysis (cont)

Dependences can be loop independent

\[
\begin{align*}
\{ & d[i+1] = a[i] \\
& a[i] + [i]q = [i]a \\
\text{for } (i=0; i<n; i++) \}
\end{align*}
\]

Loop Carried

\[
\begin{align*}
\{ & d[i+1] = a[i] \\
& a[i] + [i]q = [i]a \\
\text{for } (i=0; i<n; i++) \}
\end{align*}
\]

Loop Independent

Dependence is either not within a loop or within the same iteration of the loop.

Dependense spans multiple iterations of a loop.

Honors Compilers, NYU, Spring, 2007
Software pipelining is the technique of scheduling instructions across several iterations of a loop. Intuitively, iterations are overlaid so that an iteration starts before the previous iteration has completed.

- Reduces pipeline stalls on sequential pipelines
- Exploits instruction level parallelism on super-scalar machines
- Exploits instruction level parallelism on VLIW machines
- Reduces pipeline stalls on sequential pipelines
- Instruction pipelining is the technique of scheduling

Software pipelining involves scheduling instructions in multiple iterations of a loop across the pipeline stages.

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sequential

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Loop

Software pipelining
```
Software Pipelining Example

A. Pnueli

Lecture 14: Instruction-Level Parallelism

Honors Compilers, NYU, Spring, 2007

Source code

for (i=0; i<n; ++i) sum += a[i]

Loop body in assembly

r10 = L r0

--; stall

r10 = L r9

r6 = add r6, r16

r2 = add r2, r17

--; stall

r1 = L r6

r17 = add r3, r16

r2 = add r2, r17

--; stall

r4 = L r3

r10 = add r0, r16

r2 = add r2, r17

--; stall

r1 = L r0

Unroll loop & allocate registers

\[ r0 = \text{add } r0, 4 \]

\[ r2 = \text{add } r2, r1 \]

\[ r7 = \text{add } r3, r16 \]

\[ r2 = \text{add } r2, r17 \]

\[ r10 = \text{add } r0, r16 \]

\[ r2 = \text{add } r2, r17 \]

\[ r9 = \text{add } r3, r16 \]

\[ r2 = \text{add } r2, r17 \]

\[ r6 = \text{add } r0, r16 \]

\[ r2 = \text{add } r2, r17 \]
### Schedule unrolled instructions, exploiting VLIW (or not)

**Software Pipelining Example (cont)**

<table>
<thead>
<tr>
<th>Instruction-Level Parallelism</th>
<th>A. Pnueli</th>
</tr>
</thead>
</table>

**Identify repeating pattern**

- Kernel:
  - Pattern: `r1 = Lr0, r4 = Lr3, r2 = Addr2, r7 = Lr6` etc.

**Software Pipelining**

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Unrolled</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r1 = Lr0, r4 = Lr3, r2 = Addr2</code></td>
<td><code>r1 = Lr0, r4 = Lr3, r2 = Addr2</code></td>
</tr>
<tr>
<td><code>r7 = Lr7, r4 = Lr3, r1 = Lr1</code></td>
<td><code>r7 = Lr7, r4 = Lr3, r1 = Lr1</code></td>
</tr>
<tr>
<td><code>r1 = Lr0, r4 = Lr3, r2 = Addr2</code></td>
<td><code>r1 = Lr0, r4 = Lr3, r2 = Addr2</code></td>
</tr>
</tbody>
</table>

**Honors Compilers, NYU, Spring, 2007**
Software Pipelining Example (cont)

Loop becomes
Register Usage in Software Pipelining

In the previous example, the kernel contained many instructions. This can have an adverse impact on instruction register allocation. Due to replication of the original loop body, replication of the kernel contained many instructions. The HPL-PD and IA-64 support rotating registers to reduce the code size of the kernel.

Honors Compilers, NYU, Spring 2007

A. Pnueli
Instruction-Level Parallelism

Each register may have a static and a rotating portion.

In HPL-PD, the $i$th static register in file $F$ is named $F_i$.

The $i$th rotating register in file $F$ is named $F_i^{(R)}$.

Indexed off the RRB the rotating register base.

Each register file may have a static and a rotating portion.

Rotating Registers

\[ F_i = \left[ \frac{R_i + (i \mod \text{size}(F))}{\text{size}(F)} \right] \]
In HPL-PD, there are branch instructions e.g. BRF that decrement the RRB.

Note how the kernel can be transformed:

After the BRF instruction, the register that was referred to as 

is now referred to as 

Rotating Registers (cont)
There are also rotating predicate registers. Thirty-two predicate registers can be used as a 32-bit aggregate register. R1 = mov r1

PR is a 32-bit register consisting of 32 1-bit predicate registers.

PR = mov r1

PR causes them also to rotate.

After BRF, p[1] has the value that p[0] had.

Referred to as p[0], p[1], etc.

Rotating Predicate Registers
Lecture 14: Instruction-Level Parallelism

The instruction-level parallelism in a software pipeline is limited by constraints on software pipelining.

- Resource Constraints
- Dependence Constraints
- VLIW instruction width, functional units, bus conflicts, etc.
- Particularly loop-carried dependences between iterations
- The same memory location is used across several iterations
- The same register is used across several iterations
- Arise when the same register is used across several iterations

We refer to this case as memory aliasing.
Aliasing-Based Loop Dependencies

Dependence spans three iterations:

\[ a[i] = a[i-3] + c; \]

for (i=2; i<n; i++)

Source code:

Assembly

Initiation Interval (II)

\( \text{distance}=3 \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)

Load
Add
Store
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
\( \text{incr}^a \)
Alias-based Loop Dependencies

Source code:

```c
for (i = 2; i < n; i++)
a[i] = a[i-1] + c;
```

Distance = 1

a[i] = a[i-1] + c;

for (i = 2; i < n; i++)

Assembly

Pipeline

Initialisation Interval (II)

3 cycles

Kernel

Honors Compilers, NYU, Spring, 2007

Lecture 14: Instruction-level Parallelism
Dynamic Memory Aliasing

What if the code were:

for (i = k; i < n; i++)
    a[i] = a[i-k] + c;

where \( k \) is unknown at compile time?

The worst case is \( k = 1 \) (as on previous slide).

- The compiler has to assume the worst, and generate the most pessimistic pipelined schedule.

- The possibilities are:

  - \( k > 0 \) loop-carried anti-dependence with distance \( k \)
  - \( k < 0 \) loop-carried time dependence with distance \( k \)
  - \( k = 0 \) no loop-carried dependence

- Dynamic "aliasing"

  - The dependence distance is the value of \( k \)

Dynamic Memory Aliasing

A. Pnueli
dynamic aliasing

A new technique for Software Pipelining in the presence of

Another alternative Software Bubbling

Possible code explosion, cost of branch

Branch to the appropriate version at run-time

distances

Generate different versions of the software pipeline for different

What can the compiler do?

\[
\text{Distance} = (b - a) \\
\begin{cases}
\text{a}[i] = b[i] & \text{for } i = 0, \ldots, n-1; i++
\end{cases}
\]

void copy(char *a, char *b, int size)

This situation arises quite frequently:
The software bubbling compiler generates the most optimistic pipeline.

- Software bubbling

The compiler generates the most optimistic pipeline.

- Rotating predicate registers are especially useful, but not absolutely essential.
- All operations in the pipeline kernel are predicted.
- The predication pattern determines if the operations in a given iteration "slot" are executed.
- The predication pattern is assigned dynamically, based on the dependence distance at runtime.

For \( i = k \), \( i > n \), \( i++ \), \( a[i] = a[i-k] + c \):

Continue to use simple example:

\[
\text{for}(i=k; i>n; i++ \ a[i] = a[i-k] + c);
\]

Honors Compilers, NYU, Spring, 2007
Software Bubbling

Lecture 14: Instruction-Level Parallelism

Honors Compilers, NYU, Spring 2007

370
A. Pnueli

The Predication Pattern

- Each iteration slot is predicated upon a different predicate register.
- All operations within the slot are predicated on the same predicate register.

The Predication Pattern
The predication pattern in the kernel rotates.

In this case, the initial pattern is 11010. No operation is predicated on the leftmost bit in this case.

Rotating predication registers are perfect for this.

Bubbling Predication Pattern
slots are enabled. In this case, $T$ out of $3$ slots.

The predication pattern should ensure that only $p$ out of $T$ iteration slots are enabled.

- $T = 3$, where $d = 1$ is the dependence distance
- $II * p / T = DI$

The predication pattern should ensure that only $d$ out of $L$ iteration slots are enabled. In this case, $1$ out of $3$ slots.

$DI = L = d = 1$ is the dependence distance.

$II = 3$, the factor by which the $II$ would have to be increased, assuming the dependence spanned one iteration.

$T = \lceil \text{latency} \text{(store)} - \text{offset} \text{(store)} \text{(load)} \rceil / II = 1$
To enable \( T \) out of iteration slots, we simply create a bit pattern

\[
\text{Computation the Predication Pattern (cont)}
\]

\[
\text{Computing the Predication Pattern (cont)}
\]
Generalized Software Bubbling

So far, we have seen Simple Bubbling

- $d$ is constant throughout the loop

If $d$ changes as the loop progresses, then software bubbling can still be performed.

The predication pattern changes as well

This is called Generalized Bubbling

- The test occurs within the loop
- The iteration slot is only enabled if less than $d$ iteration slots out of the previous $L$ slots have been enabled

Examples of code requiring generalized bubbling appear quite often

- Alvin Spec Benchmark
- Lawrence Livermore Loops Code