An Overview of IA-64 Architectural Features and Compiler Optimization

Yong-fong Lee
Intel Corporation

intel®

Outline

- IA-64 Architecture Overview
- Architectural Features and Compiler Optimization
  - Control speculation
  - Data speculation
  - Predication
  - Parallel compares
  - Multi-way branches
  - Memory hierarchy control
- IA-64 Architectural Support for Software Pipelining
- Summary

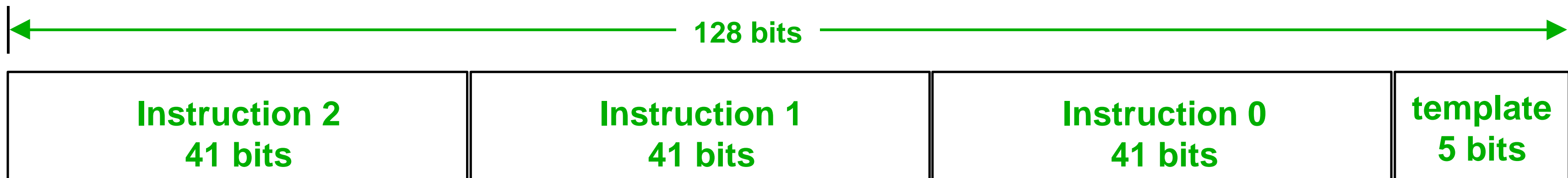
IA-64 Strategies

- Move the complexity of resource allocation and instruction scheduling to the compiler
  - no complex runtime dependence checking and resource management in hardware
- Provide architectural features that enable aggressive compile-time optimization to utilize ILP (instruction-level parallelism)
  - predication, speculation, register rotation, parallel compares, multi-way branches, architecturally visible memory hierarchy
- Provide wide-issue processor implementations that the compiler can take advantage of
  - large register files, multiple execution units

IA-64 Application State

- Directly accessible CPU state
  - 128 x 65-bit General registers (GR)
  - 128 x 82-bit Floating-point registers (FR)
  - 64 x 1-bit Predicate registers (PR)
  - 8 x 64-bit Branch registers (BR)
- Indirectly accessible CPU state
  - Current Frame Marker (CFM)
  - Instruction Pointer (IP)
- Control and Status registers
  - 19 Application registers (AR)
    - LC and EC
  - User Mask (UM)
  - CPU Identifiers (CPUID)
  - Performance Monitors (PMC,PMD)
- Memory

Instruction Formats: Bundles



- Instruction Types
  - M: Memory
  - I: Shifts, MM
  - A: ALU (M or I)
  - B: Branch
  - F: Floating point
  - L+X: Long immediate
- Template types
  - Regular: MII, MLX, MMI, MFI, MMF
  - Stop: MI_I, M_MI
    - One bundle expanded into two for execution
  - Branch: MIB, MMB, MFB, MBB, BBB
  - All come in two versions:
    - with *stop* at end
    - without *stop* at end

intel®

Instruction Groups

- A sequence of instructions with no register dependencies
  - Exception: WAR register dependencies allowed
  - Memory operations still require sequential semantics
- The compiler uses templates with stops to delimit instruction groups
- The hardware issues one or more bundles of instructions within an instruction group
  - Does not have to check for register dependencies
- Dependencies disabled by predication dynamically

Case 1 - Dependent

```
add r1 = r2, r3 ;;
sub r4 = r1, r2 ;;
shl  r2 = r4, r8
```

Case 2 - Independent

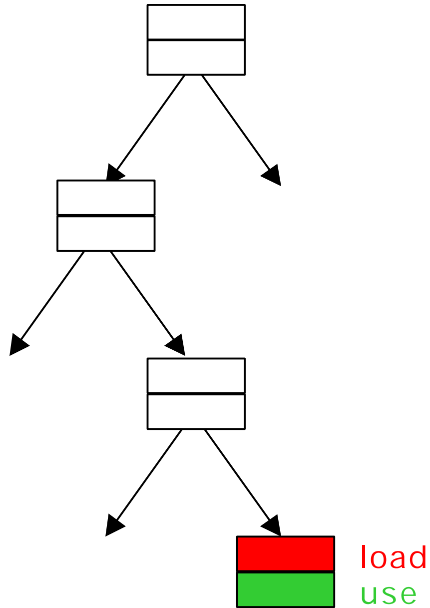
```
add r1 = r2, r3
sub r4 = r11, r21
shl r12 = r14, r8 ;;
```

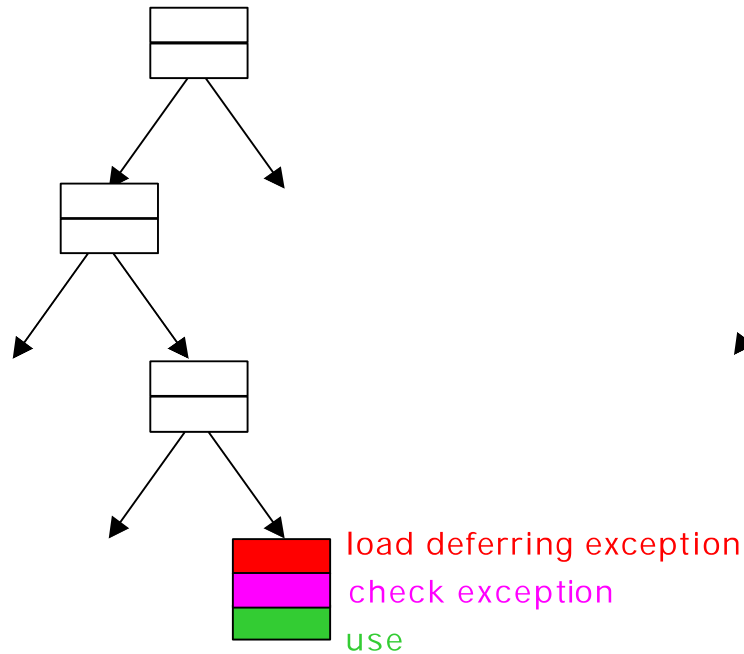
Case 3 - Predication

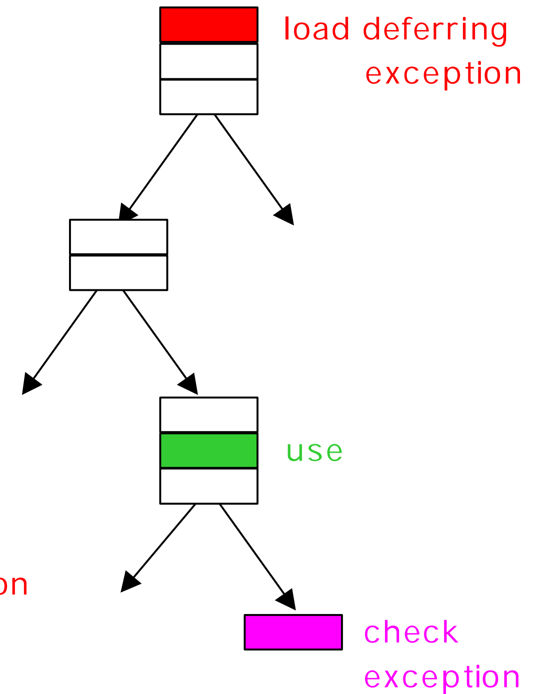
```
(p1) add r1 = r2, r3
(p2) sub r1 = r2, r3 ;;
shl r12 = r1, r8
```

intel®

Control Speculation



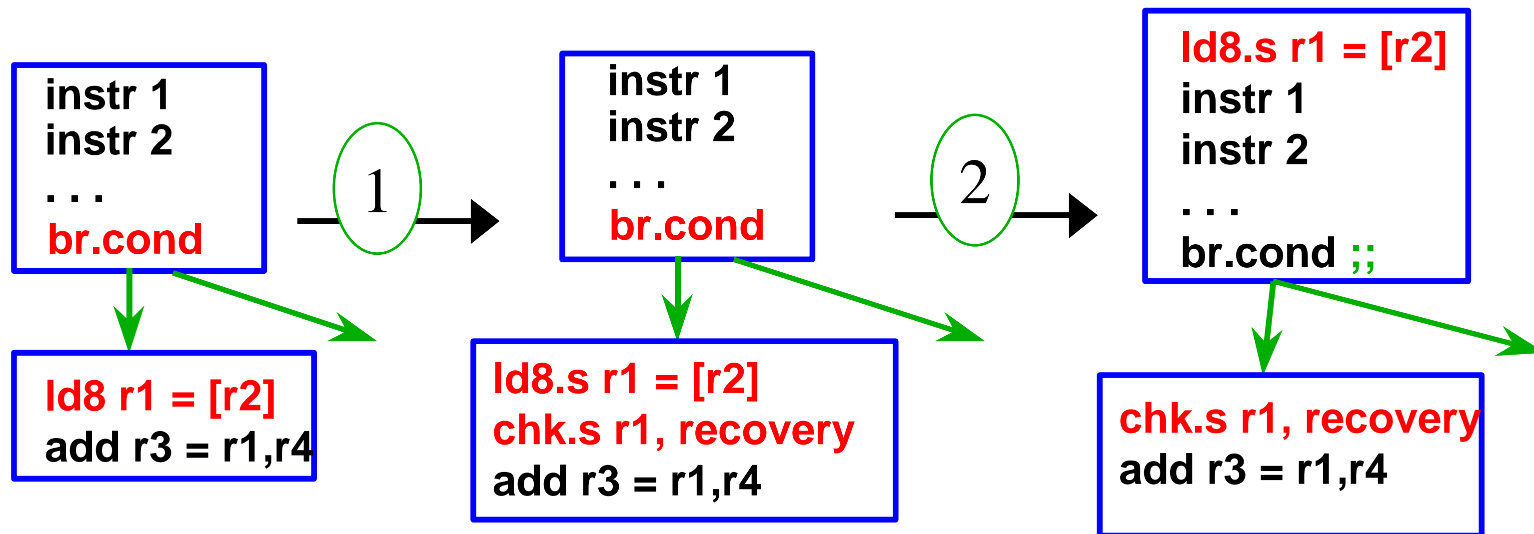




Architectural Support for Control Speculation

- Mechanism for deferred exceptions
  - 65th bit (NaT bit) in each GR indicates if an exception has occurred.
  - Special value NaTVal (a special NaN) in each FR indicates if an exception has occurred.
- Setting and propagation of deferred exceptions
  - Speculative loads (ld.s) set the NaT bit or NaTVal if a deferrable exception occurs.
  - Speculative checks (chk.s) check the NaT bit or NaTVal and branches to recovery, if detected.
  - Computational instructions propagate NaT and NaTVal like IEEE NaN's.
  - Compare instructions propagate "false" when writing predicates or leave them unchanged, depending on the compare type.

Compiler Directed Control Speculation



1
- Separate load behavior from exception behavior
  - ld.s defers exceptions
  - chk.s checks for deferred exceptions

2
- Reschedule ld8.s
  - ld8.s will defer a fault and set the NaT bit on r1
  - chk.s checks r1's NaT bit and branches/faults if necessary

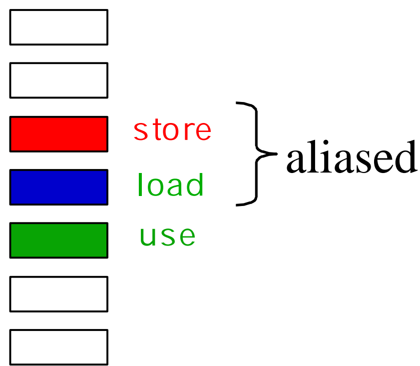
intel®

Cost-Benefit Consideration for Control Speculation

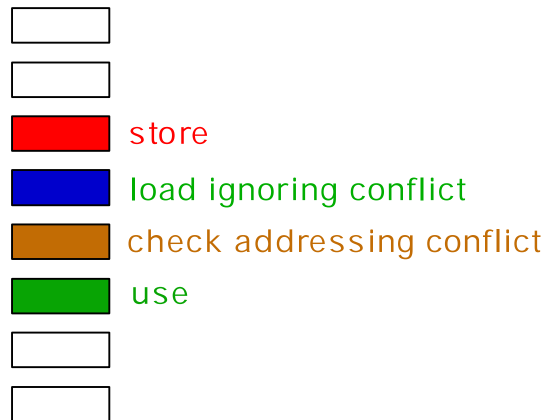
- ❒ Increased resource pressure and code size
  - – chk.s takes an instruction slot
- ❒ Extended register live ranges
- ❒ More memory traffic
  - – ld.s may be executed unnecessarily
- ❒ Profile sensitivity
  - – Should not speculate a load above a branch from an infrequent successor block

Data Speculation

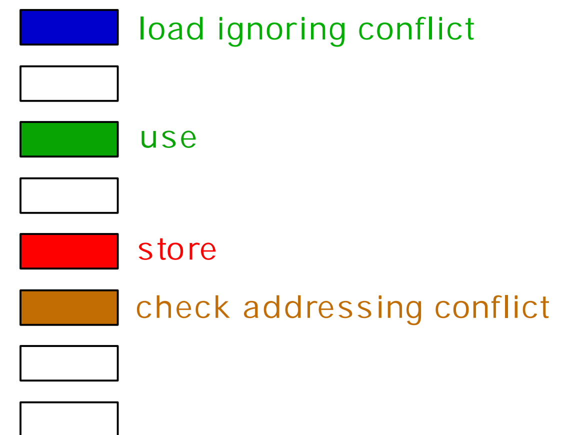
Original



Transform



Reschedule



Architectual Support for Data Speculation

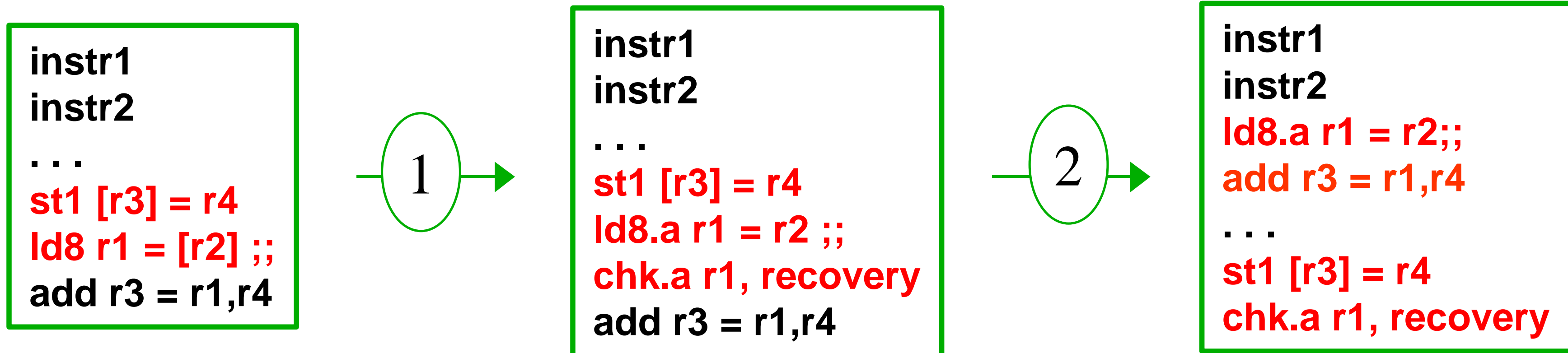
❒ ALAT - Advanced load address table

- HW structure containing information about outstanding advanced load addresses
- "Snoop" on stores and other memory writes to delete overlapping advanced load addresses

❒ Instructions

- ld.a - advanced load
- ld.c - check load
- chk.a - advance load checks
- ld.sa - combined control and data speculation.

Compiler Directed Data Speculation



1

- Separate load behavior from overlap detection
  - ld8.a can be scheduled passing aliased stores
  - chk.a detects conflict

2

- Reschedule ld8.a
  - ld8.a allocates an entry in the ALAT when executed
  - If the st1 overlaps with the ld8.a, the ALAT entry will be removed
  - chk.a checks for matching entry in ALAT -- if found, speculation was ok; if not found, need to perform recovery

intel®

Issues with Data Speculation

- Efficient recovery mechanism
  - Straightforward implementation may incur pipeline flush, branch miss, and Icache miss

- Cost-benefit consideration
  - Increased resource pressure and code size
    - » ld -> ld.a/chk.a
  - Extended register live ranges
  - ALAT size and associativity
  - Conflict rates of memory operations
    - » Average recovery overhead may cost more than the load latency

Predication

- Allow instructions to be dynamically turned on or off by using a predicate register value

- Example:

```
            cmp.eq p1, p2 = r1, r2 ;;
(p1)        add r7 = r2, r4
(p2)        ld8 r7 = [ r8 ]
```

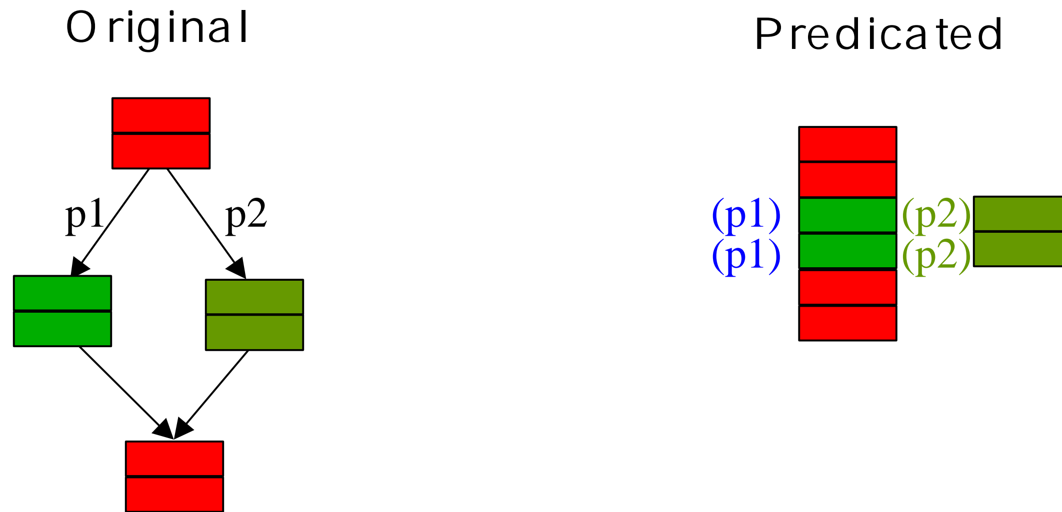
  – If `p1` is true, `add` is performed, else `add` acts as a nop
  – If `p2` is true, `ld8` is performed, else `ld8` acts as a nop

intel®

Architectural Support for Predication

- 64 1-bit predicate registers (true/false)
  - p0 - p63
- Compare and test instructions write two predicates with results of comparison/test
  - most compare/test write result and complement
  - Ex: cmp.eq p1,p2 = r1,0
- Almost all instructions can have a qualifying predicate (qp)
  - Ex: (p1) add r1 = r2, r3
  - if qp is true, instruction executed normally
  - if qp is false, instruction nullified

Compiler IF-Conversion

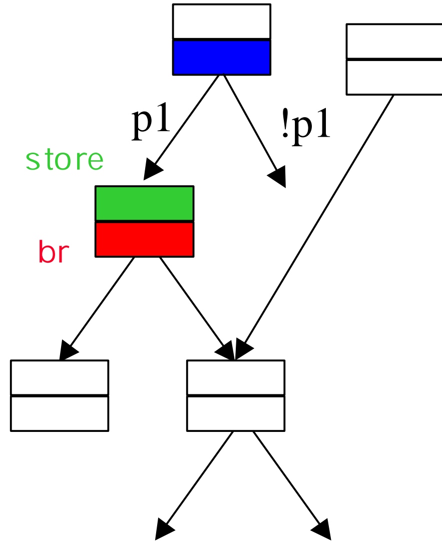


- ❒ Convert control dependence into data dependence
- ❒ Remove branches to
  - Reduce/eliminate branch mispredictions and branch bubbles
  - Improve instruction fetch efficiency
  - Better utilize wide-issue machine resources

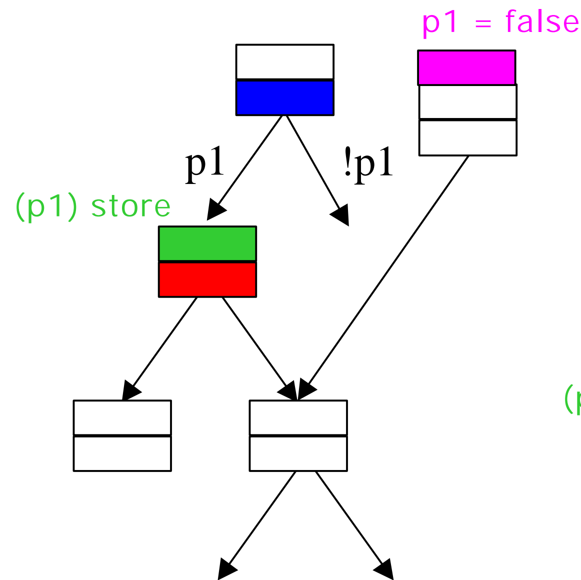
intel®

Downward Code Motion by Predication

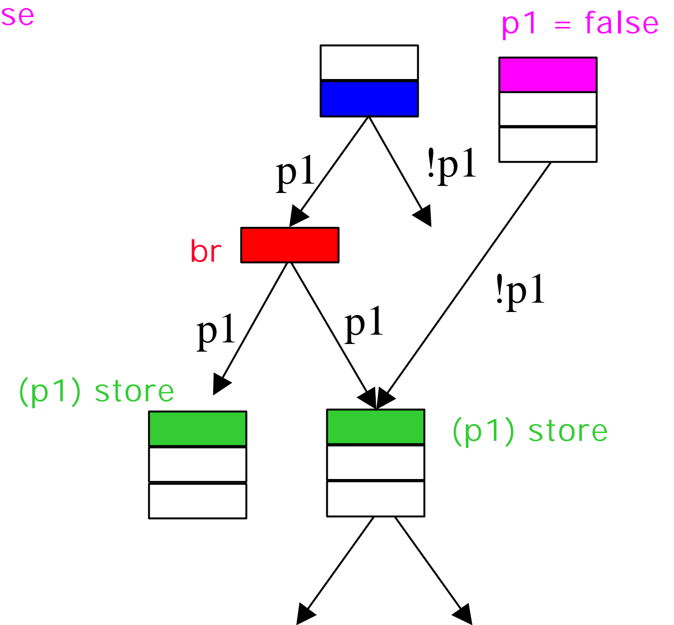
Original



Transform



Reschedule



- To move non-speculative instructions downward
  - such as stores or chk's

Issues with Predication

- Icache pollution from nullified code

- Region formation for IF-conversion
  - Identify hard-to-predict branches
    - Branch frequency?
    - Branch direction change?
    - Branch misprediction profile?
  - Balance cold and hot paths in a region
  - Performance potential for control-intensive scalar code?

- Data flow analysis of predicated code
  - More precise live range information for register allocation
  - Optimization of predicated code

Parallel Compares

- Parallel compares allow compound conditionals to be executed in a single instruction group.
  - Permit WAW in an instruction group
- Example:

```
if ( A && B && C ) { S }
```

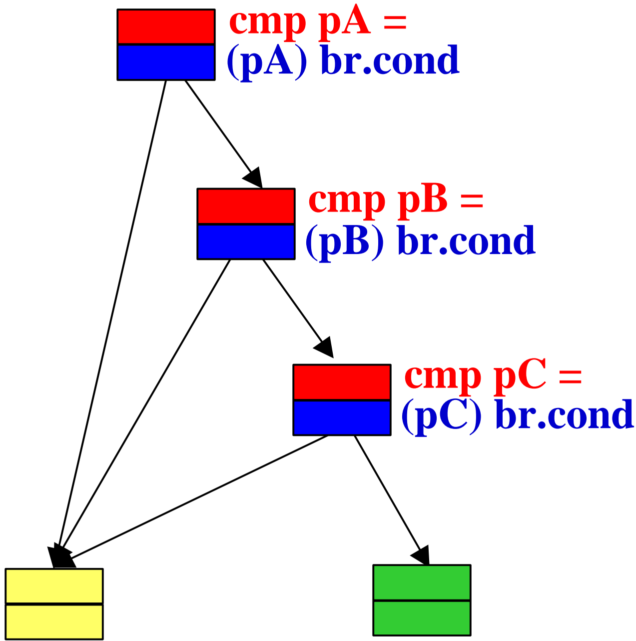
- Assembly:

```
cmp.eq p1 = r0,r0 ;; // initialize p1=1
cmp.ne.and p1 = rA,0
cmp.ne.and p1 = rB,0
cmp.ne.and p1 = rC,0
(p1) S
```

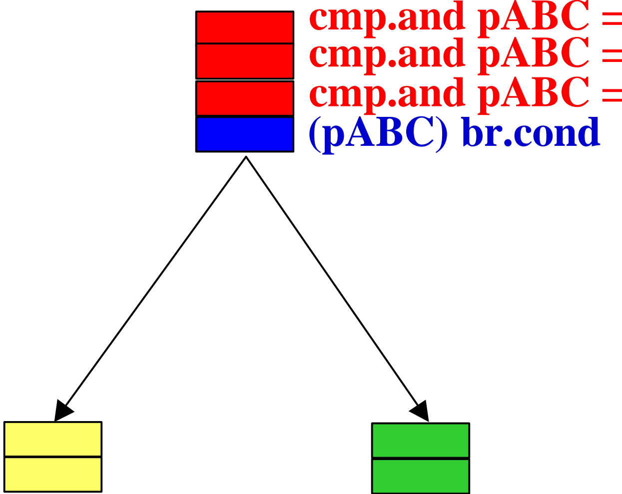
intel®

Control Height Reduction by Parallel Compares

Original



Transform/ Reschedule



Multi-way Branches

- Allow multiple branch targets to be specified in one instruction group

- Example:

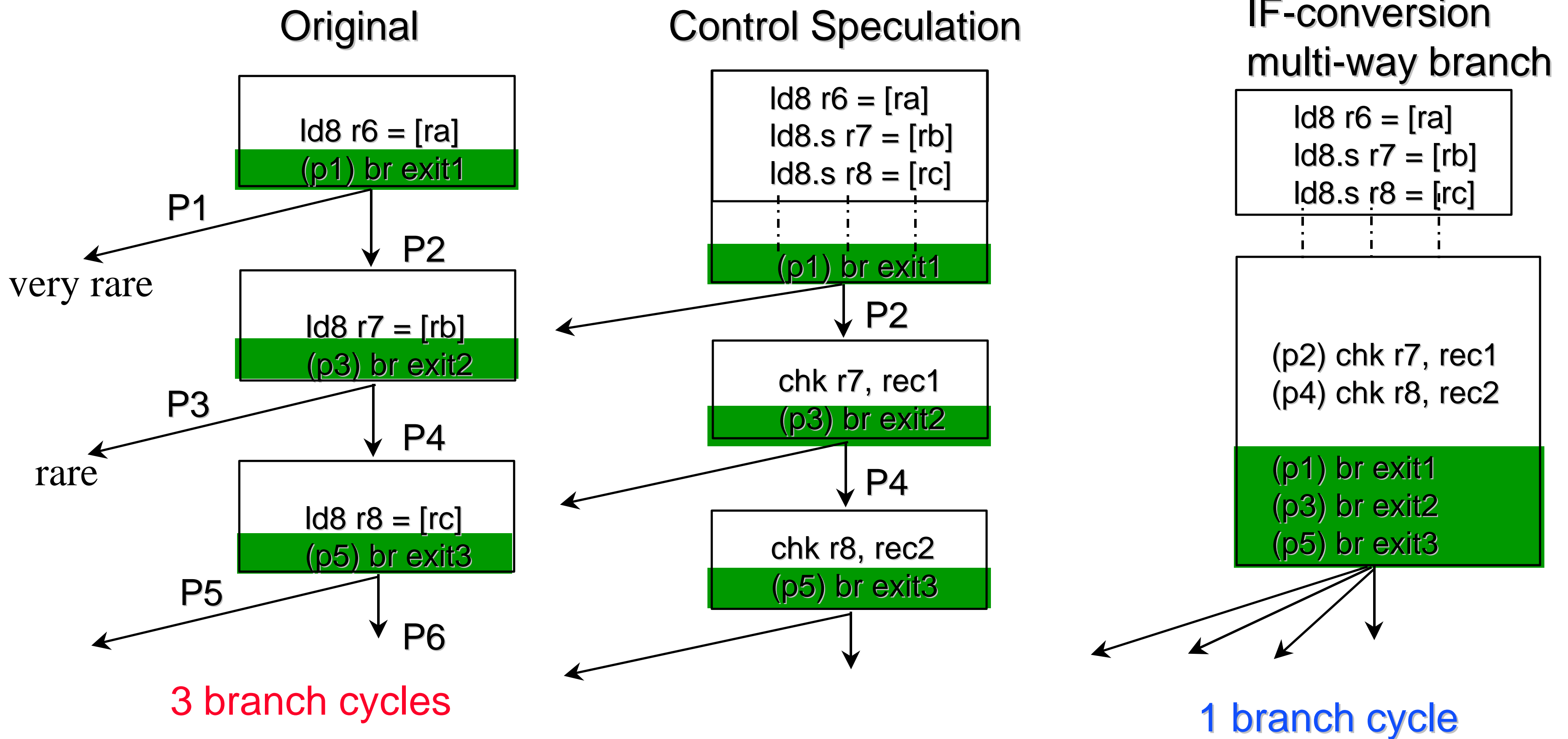
```
{ .bbb
(p1)        br.cond target_1
(p2)        br.cond target_2
(p3)        br.call b1
}
```

- Control transfers to the first true target in the sequence:
  - target_1
  - target_2
  - content of branch register b1
  - fall-through

intel®

Control Height Reduction by Multi-way Branch

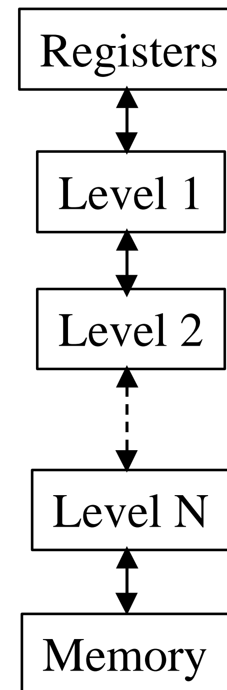
- Speculation, IF-conversion with side exits, multi-way branch



intel®

Memory Hierarchy Control

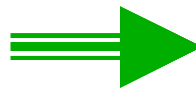
- Orchestrate data movement across the cache hierarchy
  - Locality hints
    - specified in load, store, lfetch instructions
    - Default - temporal locality at level 1
    - NT1/NT2 - no temporal locality at level 1/2
    - NTA - no temporal locality at all levels
  - Explicit prefetch
    - lfetch instruction
  - Implicit prefetch
    - post-increment with load, store, lfetch instructions
    - bring the line containing the post-incremented address



Selective Prefetching

- Reuse analysis for the best level in the hierarchy
- Avoid redundant prefetches (by using predication)

```
for i = 1, M
  for j = 1, N
    A[j, i] = B[0, j]  +  B[0, j+1]
  end_for
end_for
```



```
for i = 1, M
  for j = 1, N
    A[j, i] = B[0, j]  +  B[0, j+1]
    if (mod(j,8) == 0)
      lfetch.nt1(A[j+d, i])
    if (i == 1)
      lfetch.nt1(B[0, j+d])
  end_for
end_for
```

Post-Increment

```
for (i = 1; i < 10; i += 1)
    a[i+N] = a[i-1] + a[i+1]
```

```
        r = a
LOOP:
        r1 = r-8
        tmp1 = load *(r1)
        r2 = r+8
        tmp2 = load *(r2)
        tmp3 = tmp1 + tmp2
        r3 = r+N*8
        store tmp3 (r3)
        r = r+8
        if r < a+80 goto LOOP
```

```
        r = a - 8
        r1 = a + 8
        r2 = a + N*8 +8
LOOP:
        tmp1 = load *(r), r+=8
        tmp2 = load *(r1), r1+=8
        tmp3 = tmp1+ tmp2
        store tmp3 (r2), r2+=8
        if r < a+88 goto LOOP
```

intel®

Outline

- ❒ IA-64 Architecture Overview
- ❒ Architectural Features and Compiler Optimization
  - – Control speculation
  - – Data speculation
  - – Predication
  - – Parallel compares
  - – Multi-way branches
  - – Memory hierarchy control
- ❒ IA-64 Architectural Support for Software Pipelining
- ❒ Summary

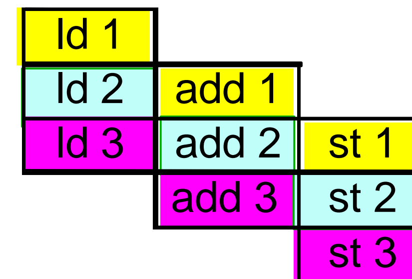
intel®

Software Pipelining

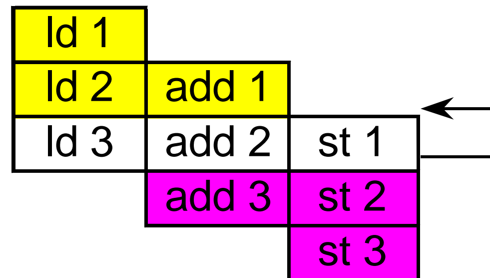
- ❒ Hardware pipelining is a hardware implementation technique for overlapping multiple instructions in execution.
- ❒ Software pipelining is a software/compiler technique for overlapping instruction instances from multiple loop iterations in execution.
  - ld1, add1, st1 from iteration #1
  - ld2, add2, st2 from iteration #2
  - ld3, add3, st3 from iteration #3

```
L1: ld4      r4 = [r5], 4 ;;   // 0
    add      r7 = r4, r9  ;;   // 1
    st4      [r6] = r7, 4      // 2
    br.cloop  L1 ;;            // 2
```

Simplistic Pipeline

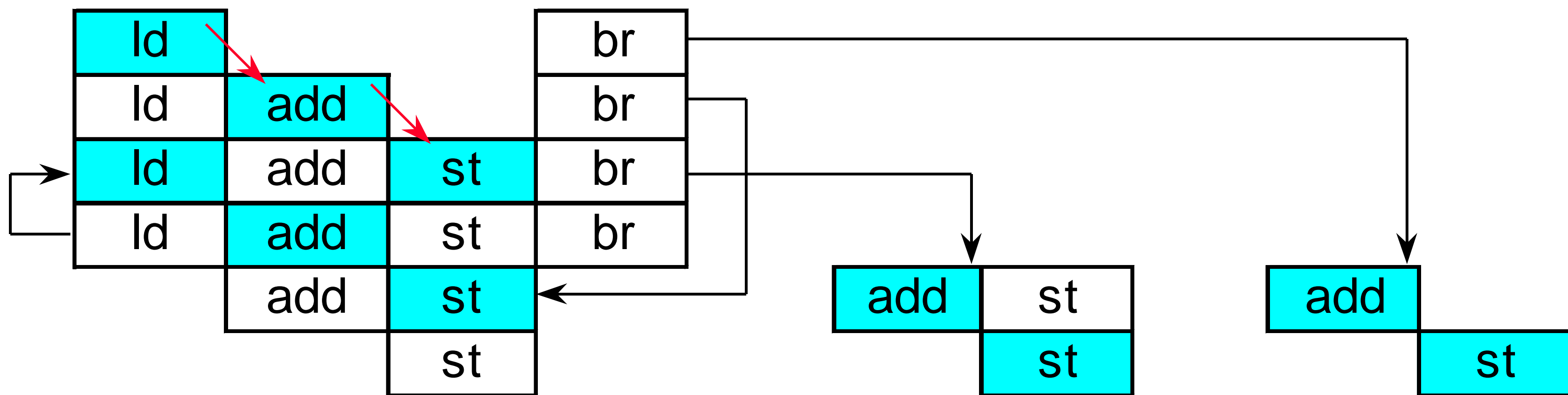


Modulo Scheduling



- Modulo scheduling is a simple, effective algorithm for software pipelining loops
- Initiation Interval (II) - number of cycles between start of successive iterations
  - minimal II = max(Resource II, Recurrence II)
- Prolog - fill pipeline
- Kernel - steady state, 1 iteration completes every II cycles
- Epilog - drain pipeline
- # of stages = ceil(length of loop schedule / II)

Modulo-Scheduled Loop Without Hardware Support



- Kernel must be unrolled because of no rotating registers
- 1 cycle per iteration, 3x performance improvement
- About 5x code-size expansion

intel®

IA-64 Support for SWP

- Full predication by IF-conversion
  - Remove control flow in loops to allow for modulo scheduling
- Rotating registers
- Loop Count (LC) and Epilog Count (EC) application registers
- Special loop-type branches

Objective of IA-64 SWP Support

- Reduce code size expansion
  - Limited unrolling of kernel <= rotating GRs, FRs, and predicates
  - No prolog, epilog code <= rotating predicates, EC
  - Reduced loop-maintenance overhead <= loop-type branches, LC
- Avoid loop-type branch mispredictions
  - Static hints avoid first-iteration misprediction
  - Loop-type branches together with LC and EC allow the hardware to predict loop exit early

```
L1: (p16)  ld4   r32 = [r5], 4
    (p17)  add   r34 = r33, r9
    (p18)  st4   [r6] = r35, 4
     br.ctop.dptk   L1
```

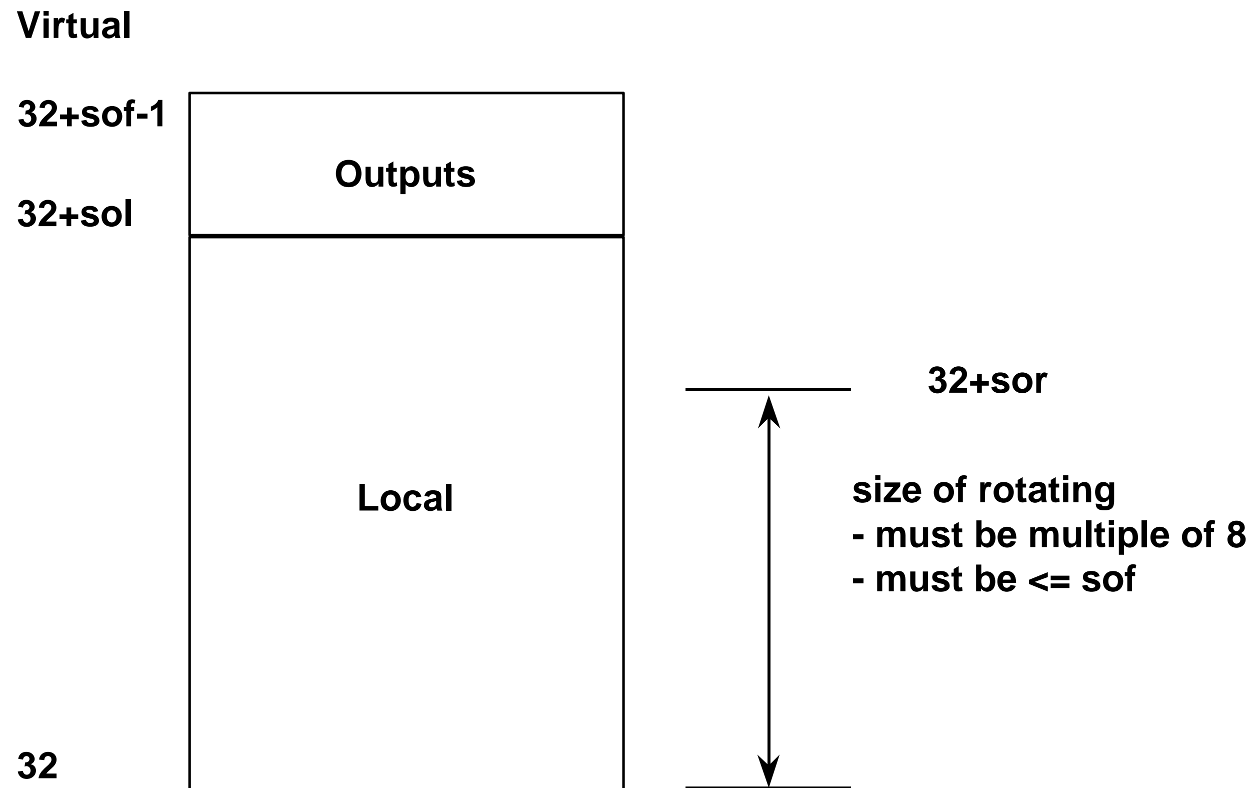
intel

Rotating Registers

- Rotation registers
  - GR rotation: Programmable sized region of the general register file rotates
- FP registers 32 through 127 rotate
- Predicates 16 to 63 rotate
  - FR rotation: 3/4 of FR can be used for rotation
  - PR rotation: 3/4 of PR can be used for rotation
- State
  - CFM.rrb.gr - register rename base for general registers
  - CFM.rrb.fr - rename base for floating-point registers
  - CFM.rrb.pr - rename base for predicate registers
  - CFM.sor - size of rotating region of general registers
- CFM.rrb's decremented modulo the size of the rotating region

Stacking and Rotation

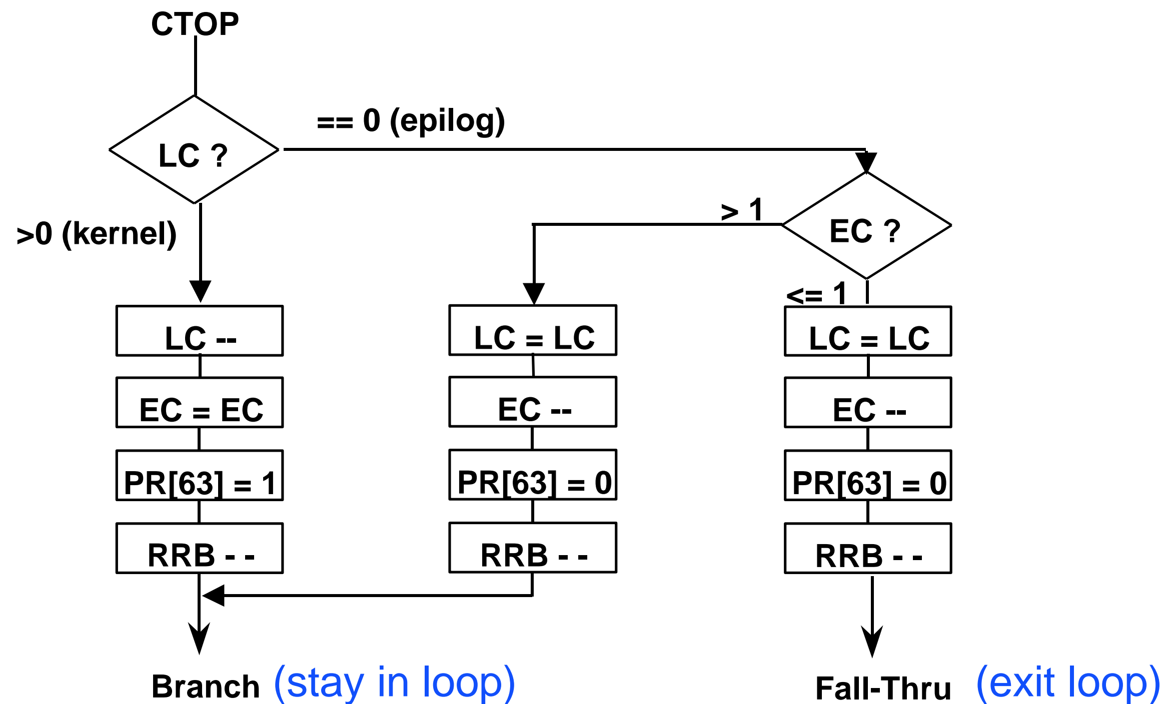
❒ Every frame consists of locals, rotating, and outputs



intel®

Loop-Type Branch

- Loop control registers
  - LC: loop count register
  - EC: epilog count register
- br.ctop uses LC and EC for pipelining counted loops

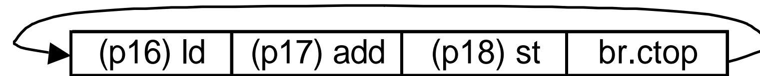


intel®

Rotating Stage Predicates in Modulo-Scheduled Loop

Generate this loop:

```
ar.lc = 4
ar.ec = 3
pr.rot = 0x10000
```



Get this trace: (5 iterations)

					Values before br.ctop					
Cycle					p16	p17	p18	LC	EC	br
1	ld			br.ctop	1	0	0	4	3	T
2	ld	add		br.ctop	1	1	0	3	3	T
3	ld	add	st	br.ctop	1	1	1	2	3	T
4	ld	add	st	br.ctop	1	1	1	1	3	T
5	ld	add	st	br.ctop	1	1	1	0	3	T
6		add	st	br.ctop	0	1	1	0	2	T
7			st	br.ctop	0	0	1	0	1	F
					0	0	0	0	0	

- ❒ 1 cycle per iteration, 3x performance improvement
- ❒ Minimal code size expansion

Issues with Software Pipelining

- ❐ Loop with multiple exits
  - – Cost of converting it into a single exit or generating explicit epilogs
- ❐ Control and data speculation in software pipelined loop
  - – Decision is not as easy as in list scheduling
- ❐ Allocation of rotating register and non-rotating registers
  - – separated or in a single pass?
- ❐ Loops in scalar code
  - – Loops with control flow
    - » Cost of full IF-conversion
  - – Loops with short trip counts
    - » Peeling, unrolling, or software pipelining?
  - – Data dependence testing for pointer references most challenging
    - » Loop carried dependence with distance conservatively assumed to be one

intel®

Summary

- The compiler should take full advantage of IA-64 architectural features to generate optimized code

Architectural Features	Compiler Optimization
Control and data speculation, Predication, Multi-way branches	Global scheduling with control and data speculation, Predicate promotion
Multi-way branches, Parallel compares	Control height reduction
Predication, Parallel compares	IF-conversion, Opt of predicated code
Rotating registers, Predication, Loop-type branches	Modulo scheduling
Locality hints, Prefetch, Post-increment	Memory hierarchy control

Compiler Challenges

- ❒ Compiler architecture issues
  - – Seamless integration of optimization components
    - » IF-conversion generates predicated instructions
    - » Scheduling and register allocation query relationships among predicates
  - – A scheduler-centric compilation framework?
    - » A global scheduler drives all other optimization and transformation components
- ❒ Understand interactions among optimization and transformation components/techniques
  - – Tail duplication increases ILP
  - – Tail duplication increases code size
  - – ILP will help more? Or code size expansion will hurt more?

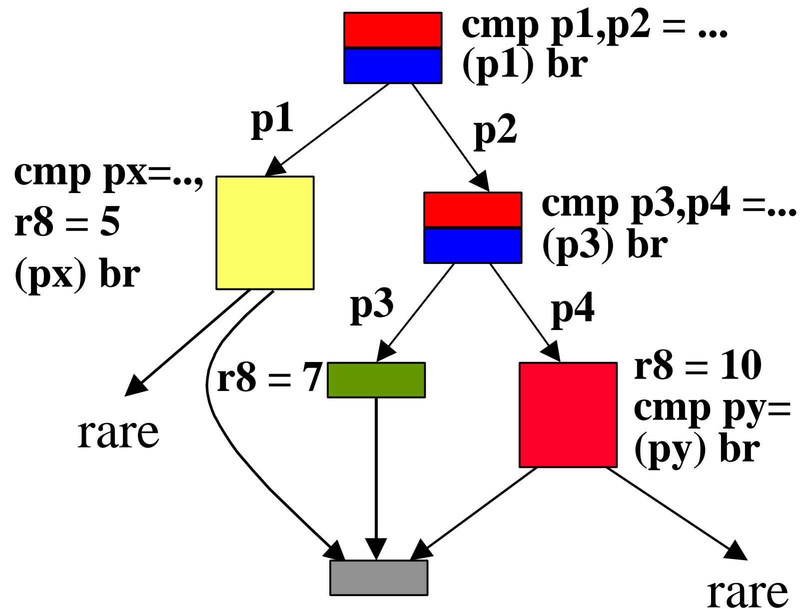
For More Information

- IA-64 Application Developer's Architecture Guide
  - Hard copy (free of charge)
    - » Call 1-800-548-4725
    - » Order number 245188-001
  - Download from IA-64 home page: http://developer.intel.com/design/ia64/index.htm
- Intel Itanium Processor Microarchitecture Overview
  - Download from http://developer.intel.com/design/ia64/microarch_ovw/index.htm

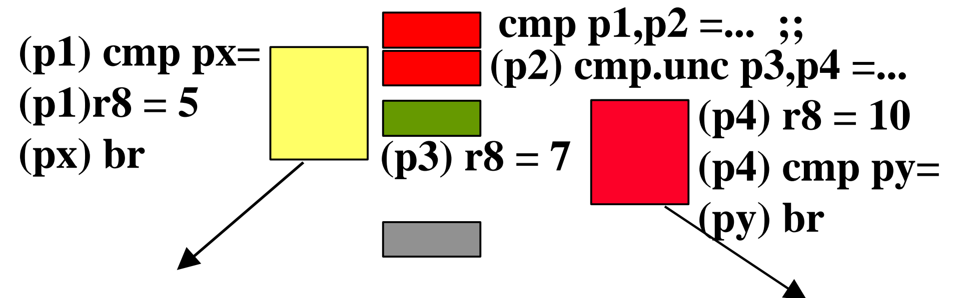
More IF-Conversion

- Side-exits in IF-conversion region

Original



Transform/
Reschedule



intel®

A Simple Counted Loop

```
L1: ld4      r4 = [r5], 4 ;;    // 0
    add      r7 = r4, r9  ;;    // 1
    st4      [r6] = r7, 4       // 2
    br.cloop   L1 ;;            // 2
```

```
L1: ld4      r4 = [r5], 4 ;;     // 0
    ld4      r14 = [r5], 4 ;;    // 1
    add      r7 = r4, r9  ;;     // 2
    add      r17 = r14, r9       // 3
    st4      [r6] = r7, 4  ;;    // 3
    st4      [r6] = r17, 4       // 4
    br.cloop   L1 ;;             // 4
```

- Assume an even number of iterations
- 5 cycles for 2 original iterations vs. 4 cycles for 1 iteration
- ~2x code size
- Induction variables r5 and r6 impose dep on ld->ld and st->st

intel®

Modulo-Scheduled Loop Branch Types and Stage Predicates

- State
  - PR16 is defined to be the first stage predicate for counted loops
    - » simple fill and drain of pipeline
  - Any predicate can be the first stage predicate for while loops
    - » early (speculative) stages of the pipeline may not have a predicate
  - LC: loop count application register
    - » initialize to (trip count - 1)
  - EC: epilog count application register
    - » initialize to (# epilog stages + 1)
- br.ctop, br.cexit always effectively write LC, EC, CFM.rrb’s
- br.wtop, br.wexit always effectively write EC, CFM.rrb’s
- br.ctop, br.cexit, br.wtop, br.wexit always write PR[63]
- PR[63] becomes PR[16] after rotation