Open Research Compiler (ORC) for Itanium™ Processor Family

Presenters:

Roy Ju (MRL, Intel Labs)
Sun Chan (MRL, Intel Labs)
Chengyong Wu (ICT, CAS)
Ruiqi Lian (ICT, CAS)
Tony Tuo (MRL, Intel Labs)

Micro-34 Tutorial

December 1, 2001

Agenda

- Overview of ORC
- New Infrastructure Features
  - Region-based compilation
  - Rich support for profiling
- New IPF* Optimizations
  - Predication and analysis
  - Control and data speculation
  - Global instruction scheduling
  - Parameterized machine model
- Research Case Study
  - Resource management during scheduling
- Demo of ORC
- Release and Future Plans

* IPF for Itanium Processor Family in this presentation

Overview of ORC

Objectives of ORC

- To provide a leading open source IPF (IA-64) compiler infrastructure to the compiler and architecture research community
  - To encourage compiler and architecture research
  - To minimize the resource investments for university groups
  - Performance better than existing IPF open source compilers
  - Fair comparison on a common infrastructure

Requirements for ORC

- Robustness
  - solid research compiler infrastructure
- Timing of availability
  - to enable research in early IPF systems
- Flexibility
  - modularity and clean interface to facilitate prototyping novel ideas
- Performance
  - leading performance among IPF open source compilers
  - sufficiently high to make research results from this compiler trustworthy

What's in ORC?

- C/C++ and Fortran compilers targeting IPF
- After evaluation, chose to base on the Pro64 open source compiler from SGI
  - Retargeted from the MIPSpro product compiler
  - Mostly meet our requirements
  - `open64.sourceforge.net`
- Major components:
  - Front-ends: C/C++ FE and F90 FE
  - Interprocedural analysis and optimizations
  - Loop-nest optimizations
  - Scalar global optimizations
  - Code generation
- On the Linux platform

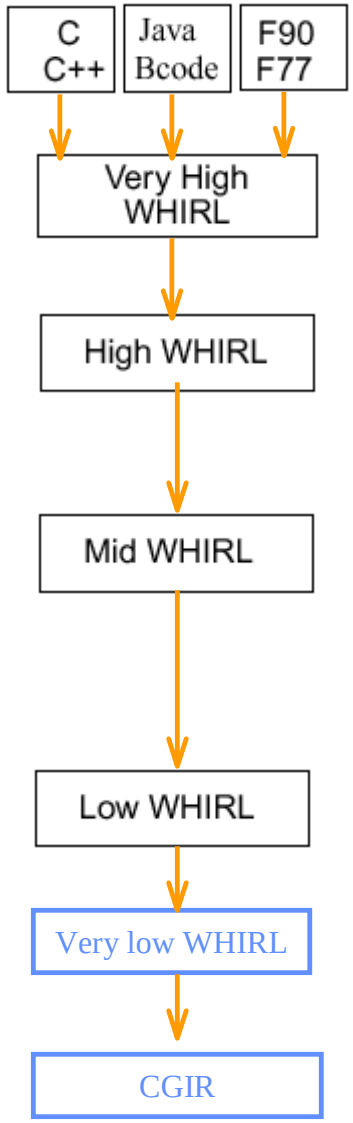
BE Components Inherited from Pro64

- Inter-procedural analysis and optimizations (IPA)
  - mod/ref summary, aliasing analysis, array section analysis, call tree, inlining, dead function elimination, …
- Loop-nest optimizations (LNO)
  - Locality opt., parallelization, loop distribution, unimodular transformations, array privatization, *OpenMP*, …
- Scalar global optimizations (WOPT)
  - SSA-based partial redundancy elimination, induction variable recognition, strength reduction, ld/st-PRE, copy propagation, …
- A Pro64 tutorial by Gao, Amaral, Dehnert at PACT 2000
  - `http://www.cs.ualberta.ca/~amaral/Pro64/index.html`
- A list of publications from MIPSpro posted by SGI.

Intermediate Representations (IR)

- WHIRL:
  - AST-based IR
  - To communicate among IPA, LNO, WOPT, and CG
  - Well documented and released by SGI
- Symbol table:
  - Document also released by SGI
- CGIR:
  - Register-based IR used in CG

Flow of IR



VHO
standalone inliner

IPA
PREOPT
LNO

WOPT
RVI1

RVI2

CG

CG

Front-ends

Lower aggregates
Un-nest calls
Lower COMMAs, RCOMMAs

Lower ARRAYs
Lower Complex Numbers
Lower high level control flow
Lower IO
Lower bit-fields
Spawn nested procedures for parallel regions

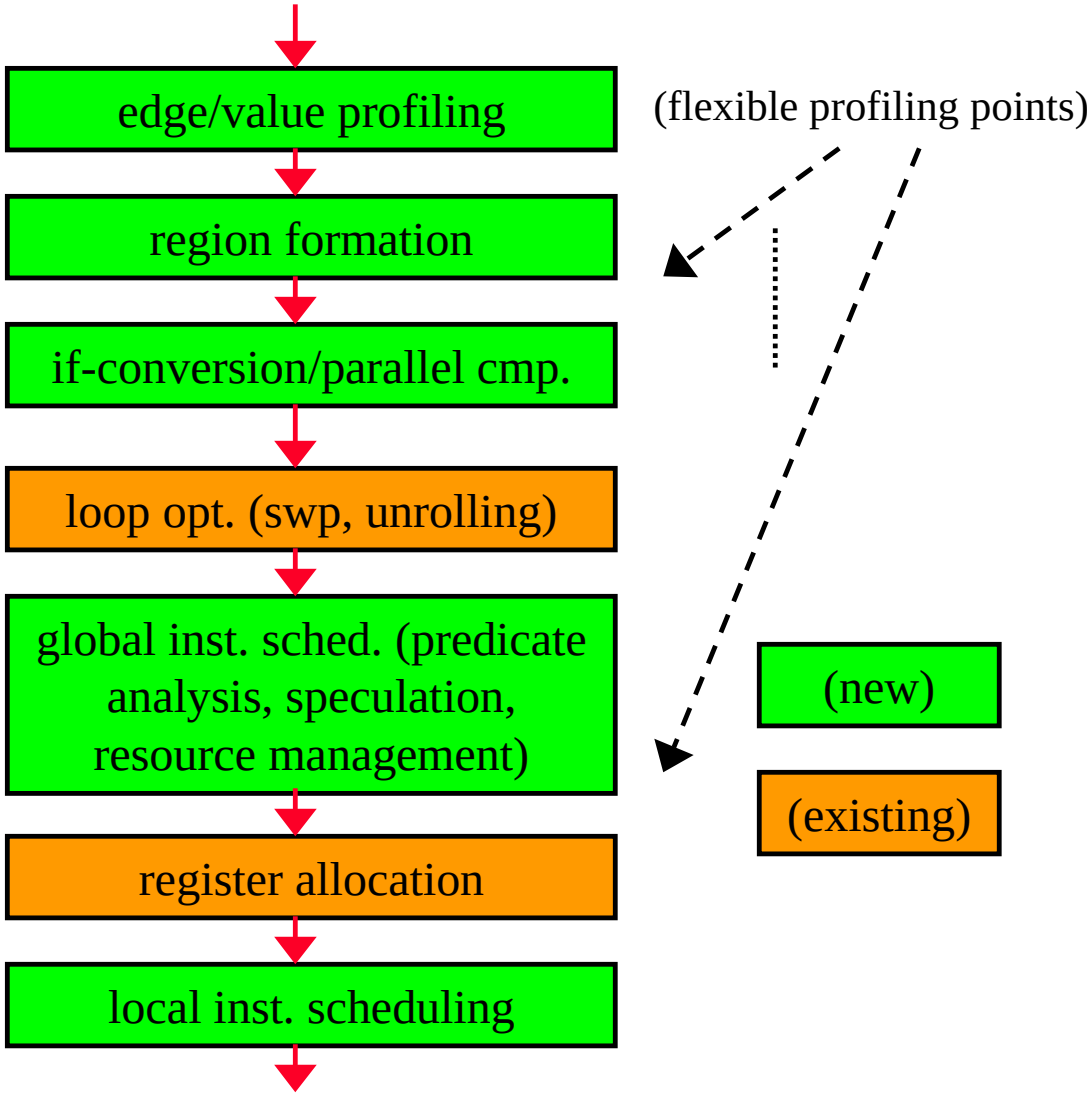
Lower intrinsics to calls
Generate simulation code for quads
All data mapped to segments
Lower loads/stores to final form
Expose code sequences for constants and addresses
Expose $gp for -shared
Expose static link for nested procedures

Code Generation

What's new in ORC?

- A largely redesigned CG
- Research infrastructure features:
  - Region-based compilation
  - Rich profiling support
  - Parameterized machine descriptions
- IPF optimizations:
  - If-conversion and predicate analysis
  - Control and data speculation with recovery code generation
  - Global instruction scheduling with resource management
- More beyond the first release

Major Phase Ordering in CG



Perspective Research Usage of ORC

- Performance-driven optimizations in all components
- Co-design of compilers and architecture for new hardware features
- Thread-level parallelism
- Retarget to emerging languages (e.g. CLI, Java, …)
- Power management
- Type safety under optimizations
- Optimizations for memory hierarchy
- Program analysis
- Co-design of static and dynamic compilation
- …

The ORC Project

- Initiated by Intel Microprocessor Research Labs (MRL)
- Joint efforts among
  - Programming Systems Lab, MRL
  - Institute of Computing Technology, Chinese Academy of Sciences
  - Intel China Research Center, MRL
- Development efforts started in Q4 2000
- Development team: 10 – 15 people
- First release scheduled for Jan 2002

- Overview of ORC
- **New Infrastructure Features**
- New IPF Optimizations
- Research Case Study
- Demo of ORC
- Release and Future Plans

Region-based Compilation

Region-based Compilation

- Motivations:
  - To form a scope for optimizations
  - To control compilation time and space
- What's a region?
  - Nodes: basic blocks
  - Edges: control flow transfer
  - Acyclic
    - Loops impose region boundary
    - Exception: irreducible regions
  - (Currently) single-entry-multiple-exit
  - Regions under hierarchical relations
    - Regions could be nested within regions

Features of Region-based Compilation

- Region structure can be constructed and deleted at different optimization phases
  - Incremental update also supported
- Optimization-guiding attributes at each region, e.g.
  - No further optimizations, e.g. swp'ed regions
  - No optimization across region boundary
  - These attributes need to be preserved if region structure is rebuilt
- Region formation algorithm decoupled from the region structure
  - Can construct and support different types of regions
- Basic blocks, superblock/hyperblock, treegion, etc. are special cases of SEME regions

Region-based Compilation

- Utility and support:
  - Iterators to traverse regions
  - Each region marked with its attributes
  - Regional CFG
  - Incremental update due to CFG changes
  - Validation of regions
- Region formation takes into account:
  - Size
  - Shape and topology
  - Exit probability
  - Tail duplication and duplication ratio

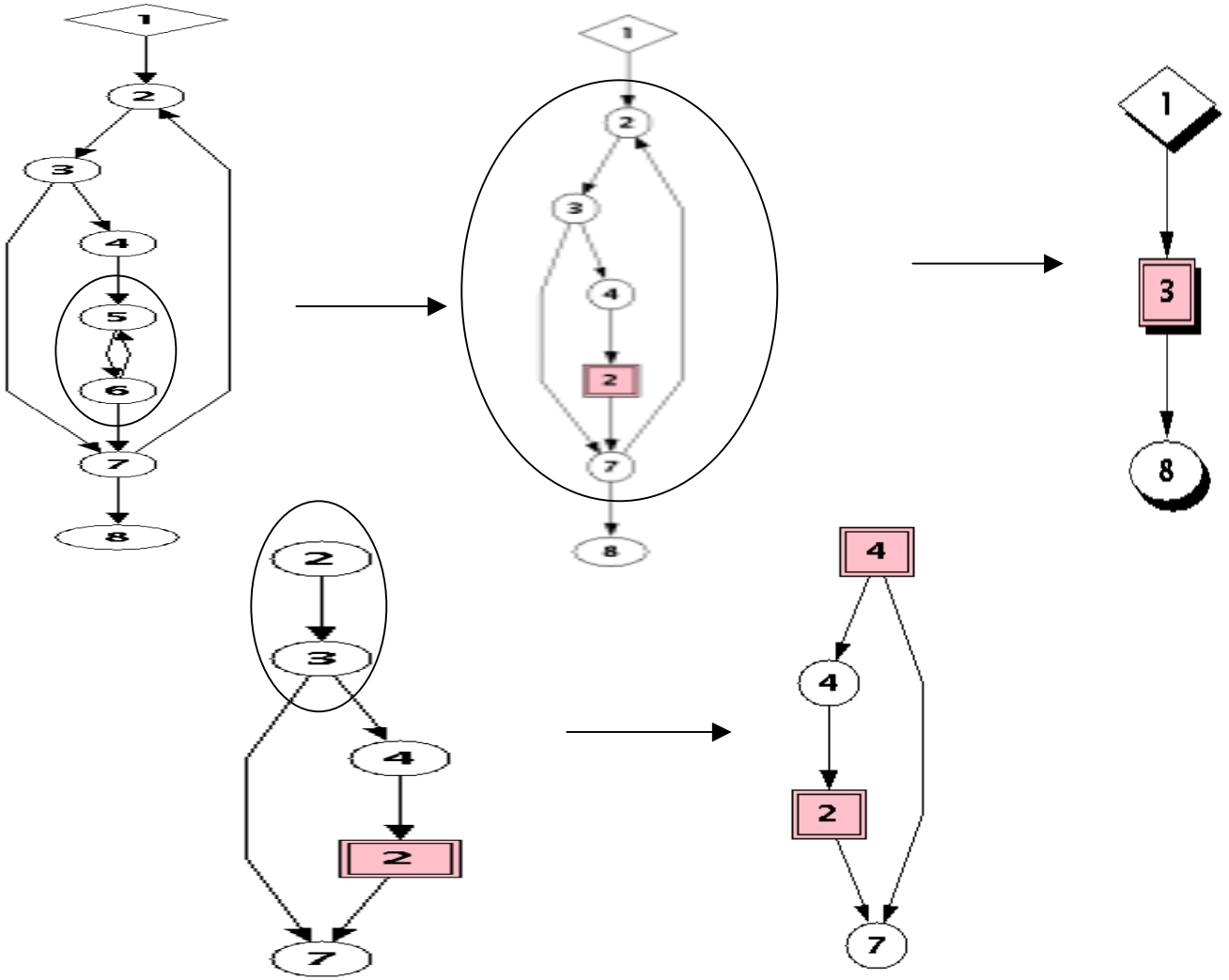
Region-based Compilation

- Current ORC implementation
  - Region structure constructed right before if-conversion
  - Preserved till after global instruction scheduling
  - Phases working under regions
    - The majority of CG phases
    - If-conversion, predicate analysis, loop optimizations, instruction scheduling, speculation w/ recovery code gen, EBO, CFLOW, etc.
    - Noticeable exception: register allocation (GRA)
  - Different from the incomplete region work in Pro64

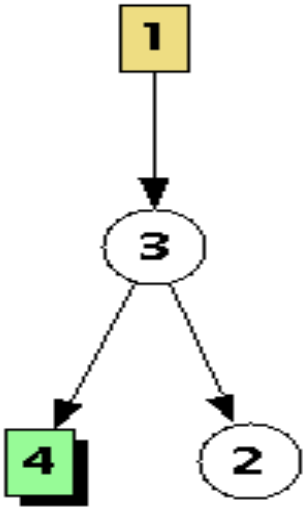
Outline of Region Formation

- Form interval regions
- Form MEME regions
  - Find a seed with the highest frequency
  - Extend to form a hot path and then an MEME region
- Form SEME regions from each MEME region
- SEME Region Formation
  1. For node x whose exit probability > *threshold*, add x to candidate exits
  2. For every candidate exit, traverse backward to form a candidate set *m'*
  3. Try the candidate exit y with the largest size of *m'*
  4. If *m'* is already SEME, done
  5. If *m'* has side entries, select nodes to cut and compute the duplication ratio
  6. If the ratio is within the budget, tail duplicate to trim *m'* and done
  7. If the ratio is beyond the budget, remove y from the candidate exits and go back to step 3

Example of Region Hierarchy

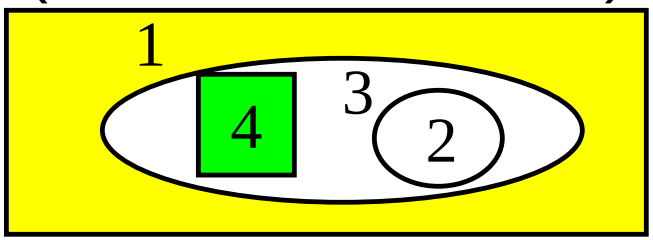


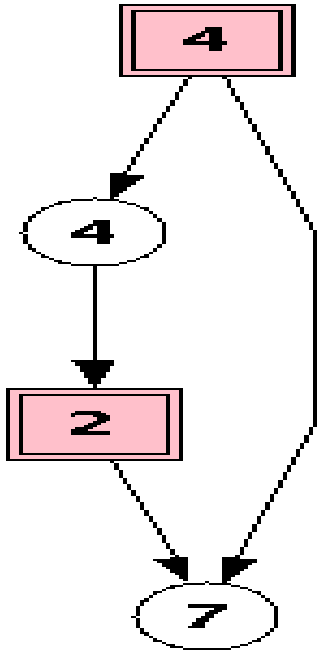
Example of Region Hierarchy (cont.)



Region Tree

(hierarchical relation)





Regional CFG of Region 3

Usage of Region-based Compilation

- Current usage
  - Forming profitable optimization scopes
    - For global instruction scheduling and if-conversion
  - Controlling compilation time and space
- Perspective research usage
  - Regions as optimization boundary
  - Optimization-guiding attributes to propagate info from an opt. phase to a later one
  - E.g. to support the multi-threading regions partitioned by compilers
  - Comparing optimizations under different shapes of regions

Rich Support for Profiling

Profiling Support in Pro64

- Pro64 user model:
  - Edge profiling for execution frequency
  - Instrumentation and feedback annotation at same point of compilation phase
  - Consistent optimization levels to ensure the same inputs at both instrumentation and annotation
  - Later phases maintain feedback correctness through propagation and verification
  - Instrumentation right after FE

Usage:

-fb_create directory-path

-fb_opt directory-path

Profiling Support in ORC

- Edge profiling, value profiling, and beyond
- Various instrumentation points in CG
- Same user model as Pro64
- Co-exist with the Pro64's profiling model
- Optimizations after feedback point update and verify feedback correctness
- Or start a new instrumentation/annotation point to avoid update

ORC Profiling in CG

Usage, backward compatible with Pro64:

- -fb_create dir-path { -fb_phase n } {-fb_type m}
- -fb_opt dir-path { -fb_phase n } {-fb_type m}

  - where n is instrumentation point.
    - Pro64 model
    - beginning of cg
    - after if-conversion in cg

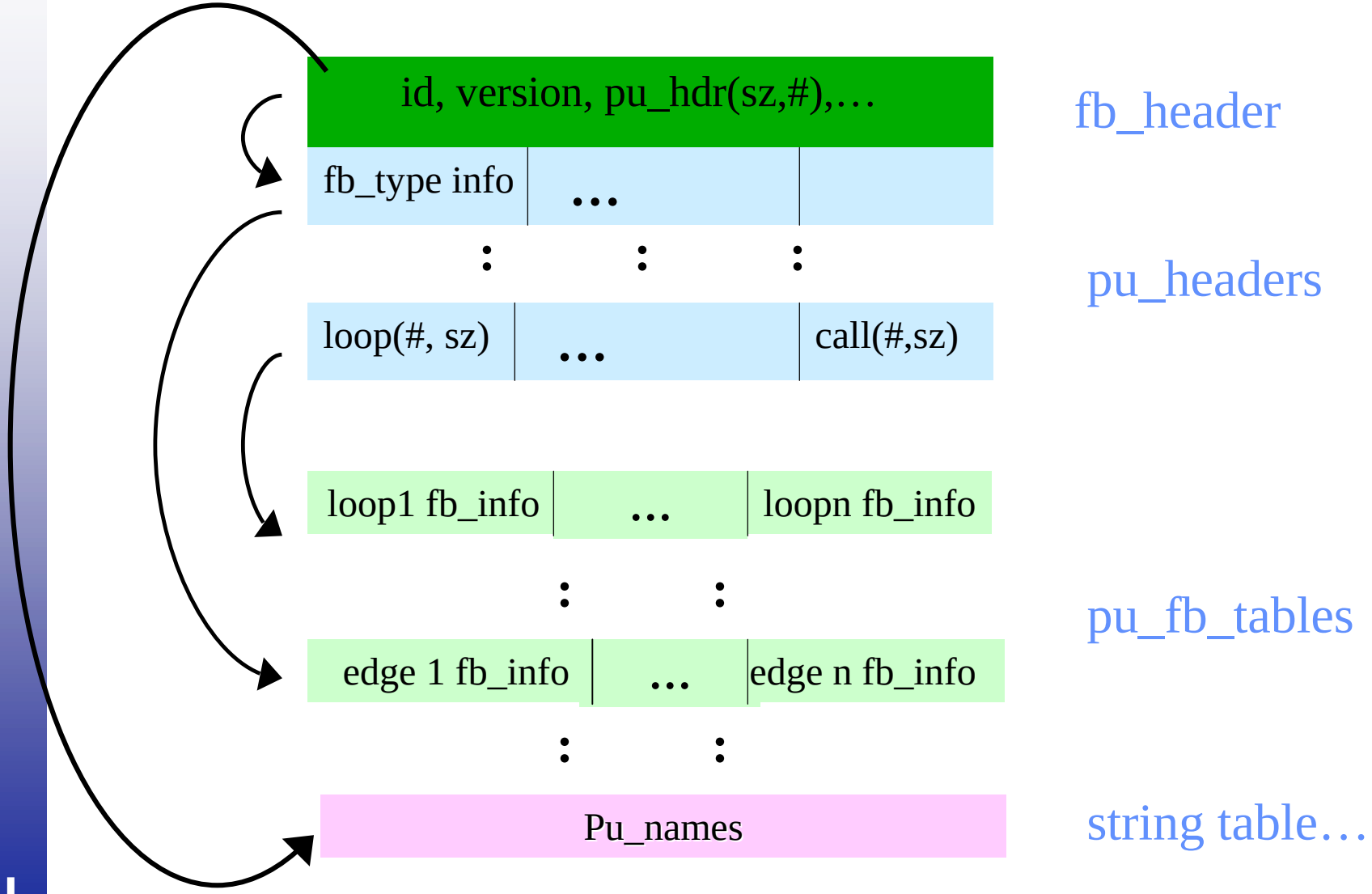
Value Profiling

- Profiling the values of instruction operands
- Important tool for limit study or to collect program statistics
- Based on Calder, Feller, Eustace, “Value Profiling”, *Micro-30*
- Top value tables in feedback file
- Current usage
  - Profiling target values of loads at the beginning of CG
  - Profiling values for selective loads at a later phase

Extend to Other Profiling

- Feedback format
  - Flexible to extend
  - Simple to maintain backward compatibility
  - Same format for every phase
- Feedback at different phases go to different feedback files – simple scheme to deal with various profiles
- Perspective research usage
  - Extend to memory profiling, data profiling, return value profiling, …
  - Collect program statistics

Feedback format



- Overview of ORC
- New Infrastructure Features
- **New IPF Optimizations**
- Research Case Study
- Demo of ORC
- Release and Future Plans

If-conversion and Predicate Analysis

Architecture Support of Predication

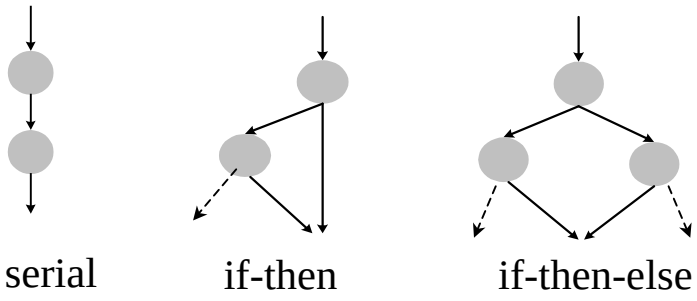
- Predicate registers
  - 64 predicate registers(pr16-pr63 rotating registers)
  - Predicate register transfers:
    - mov pr = … / mov … = pr / mov pr.rot = /… …
- Conditional execution
  - Qualifying predicate: to decide if the guarded instructions modify architectural state
- Compare instructions
  - Normal compare
  - Unconditional compare
  - Parallel compare: and, or , and/orcm, or/andcm

Compilation Support of Predication

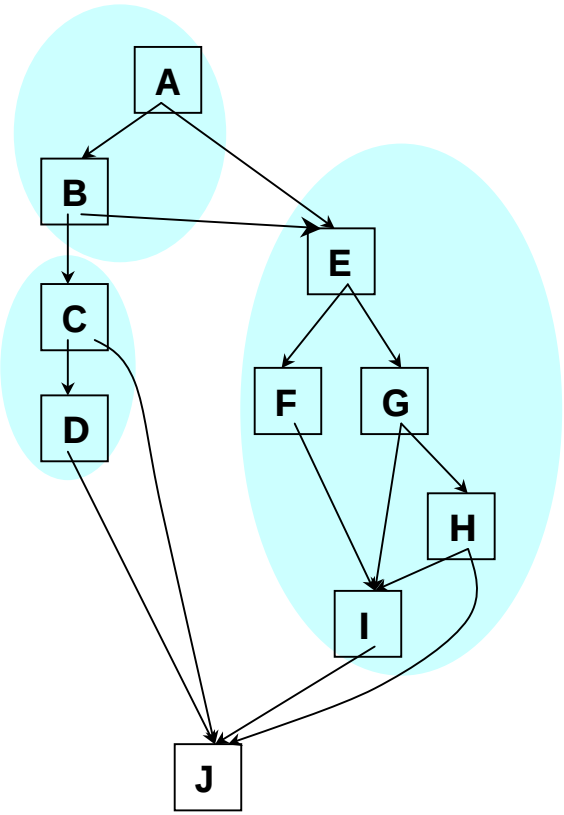
- If-conversion
  - Converts control flow (branches eliminated) to predicated instructions
  - Generates parallel compare instructions
  - Invoked after *region formation* and before *loop optimization*
  - Replaces the hyperblock formation in Pro64
- Predicate analysis
  - Analyze relations among predicates and control flow
  - Relations stored in Predicate Relation Database (PRDB)
  - Interface provided to query PRDB
  - PRDB can be deleted and recomputed as wish without affecting correctness

If-conversion

- Simple and effective framework
  - Step1: select candidates
    - Checking adjacent nodes to match one of three types



    - Iterative detection within a region
  - Step2: convert selected candidates to predicated code

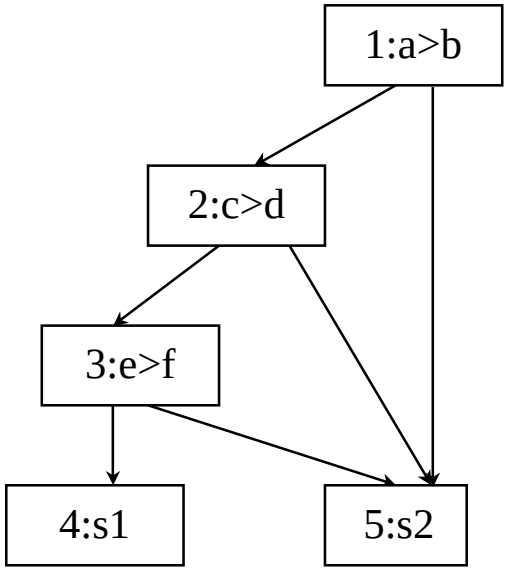


Generation of Parallel Compare

- Part of the if-conversion phase
  - Step1: profitability checking
    - Current heuristics to detect simple patterns
  - Step2: inserting parallel compare instructions
- Example

```
if ( a>b && c>d && e>f )
        s1;
else
        s2;
```

```
p, q = 1
cmp. gt. and. orcm p,q = a,b
cmp.gt.and.orcm p,q = c,d
cmp.gt.and.orcm p,q = e,f
(p) s1
(q) s2
```



Features of If-conversion

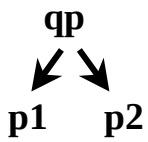
- Effective and extensible cost model
  - Taking into account
    - critical path length
    - resource usage
    - branch mis-prediction rate (approx.) and penalty
    - number of instructions
  - Separate legality and profitability checking
    - Easy to tune and extend the cost model
- Utilize parallel compare instructions
- Clean interface
  - Feasible to change the phase ordering or replace with a new implementation

Features of Predicate Analysis

- Analyze predicate relations among both control flow and explicit predicates
- Query interface to PRDB: disjoint, subset/superset, complementary, sum, difference, probability, …
  - Currently used during the construction of dependence DAG
- PRDB can be incrementally updated or deleted/re-computed at any phase
- Relations tracked using the well-known predicate partition graph but the analysis not assuming SSA form
- No coupling between the if-conversion and predicate analysis
  - Can replace just one of them if wish

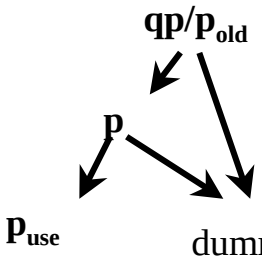
Predicate Partition Graph

- Partition generated by normal compare type

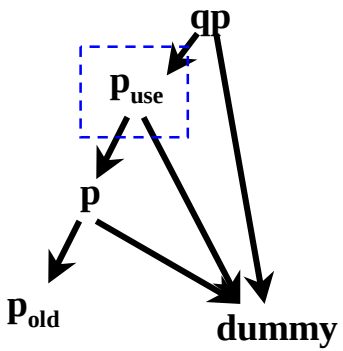


(qp) p1, p2 = cmp.unc <condition>

- Partitions generated by parallel compares



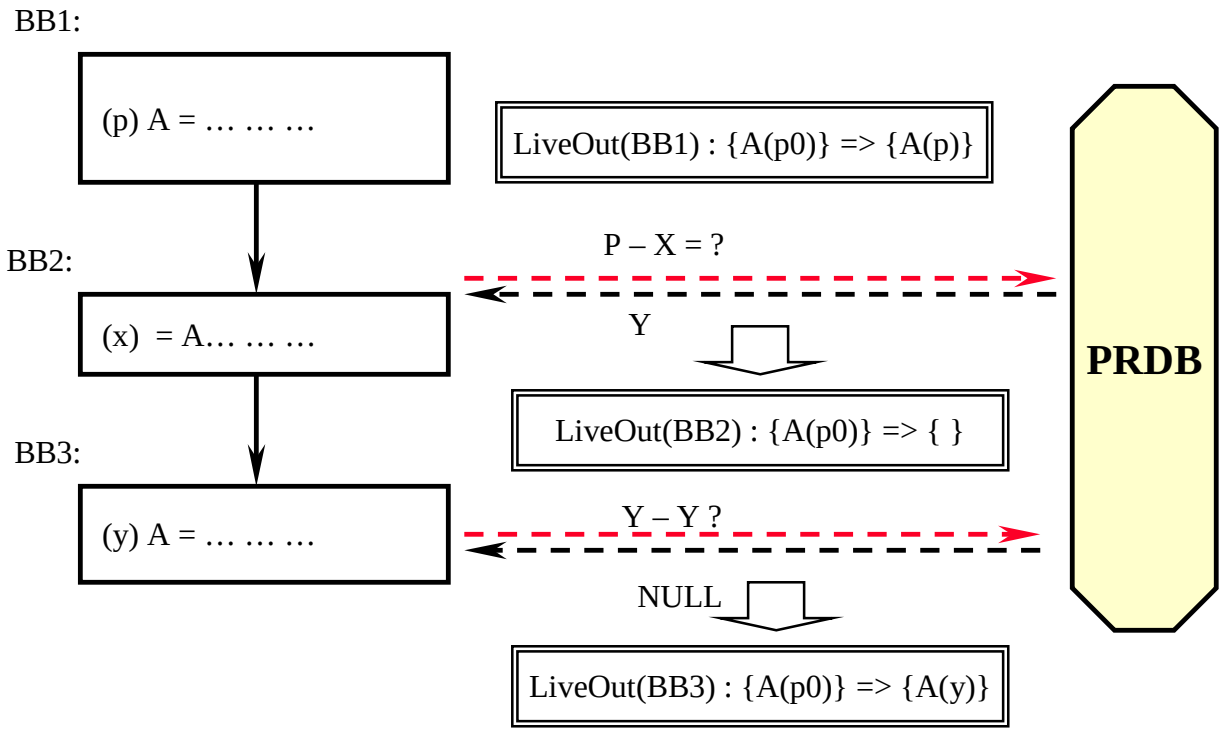
(qp) p, .. = cmp.and <condition>



(qp) p, .. = cmp.or <condition>

Additional Uses of PRDB

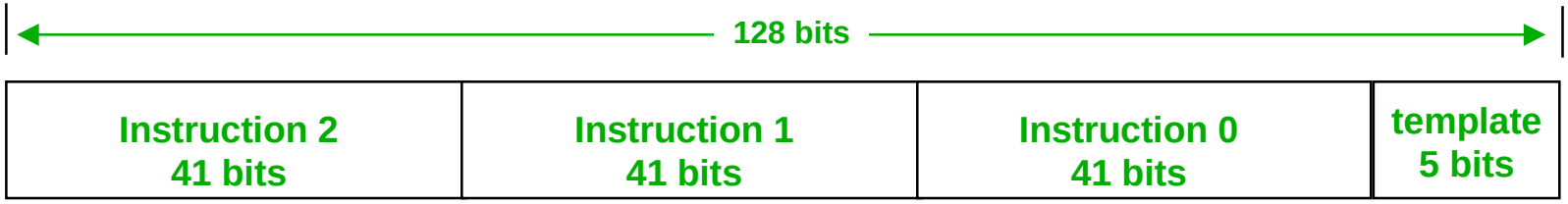
- Predicate-aware data flow analysis (e.g. in register allocation)
- Example in calculating live sets
  - Query PRDB to get *Sum* or *Diff* of predicates to refine data-flow solutions



Global Instruction Scheduling

Hardware Features for Instruction Scheduling

- Wide execution resources:
  - Itanium: 2 I-units, 2 M-units, 2 F-units and 3 B-units
- Instruction mixes specified by templates



  - Itanium: 2 bundles/6 instructions per cycle
  - Each generation of IPF has its own dispersal rules and micro-architectural features
- Large register files:
  - 128 GRs, 128 FRs, 64 PRs, and 8 BRs

Key Features of Instruction Scheduling

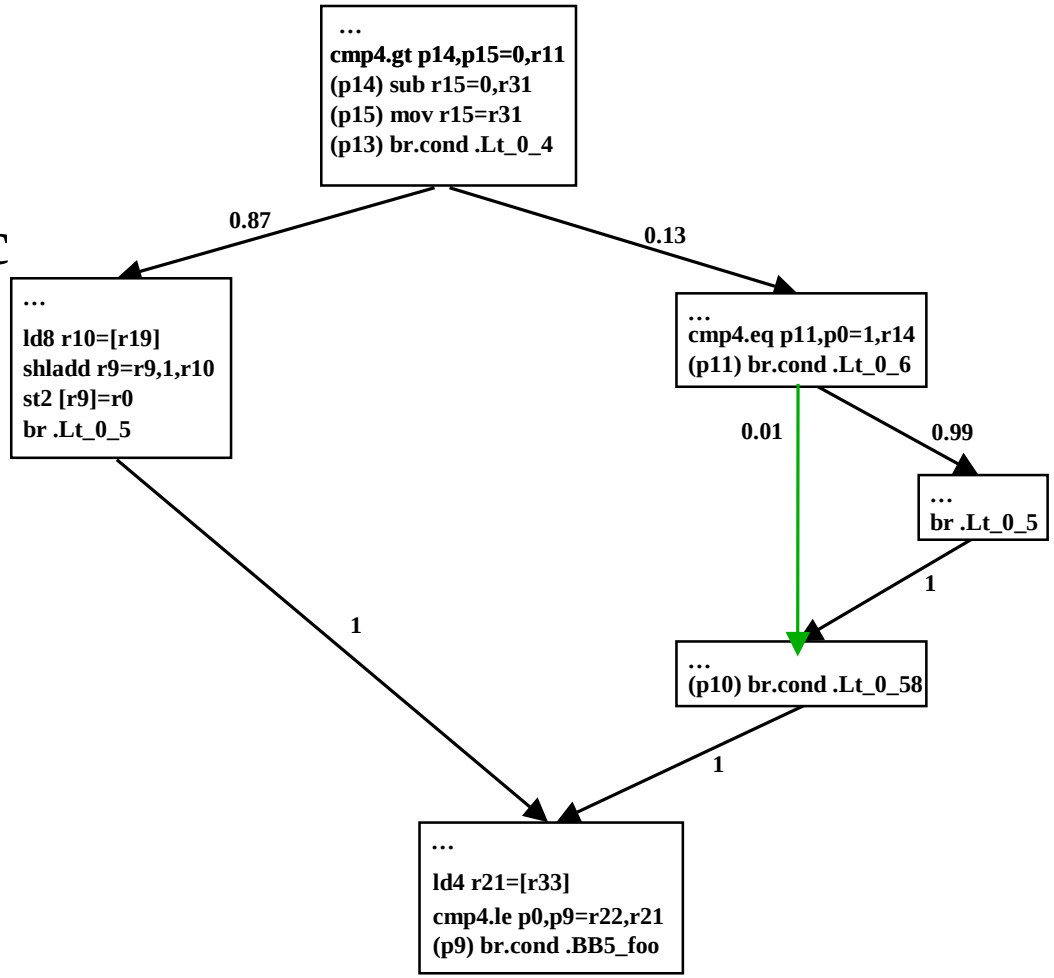
- Based on: D. Berstein, M. Rodeh, “Global Instruction Scheduling for Superscalar Machines,” *PLDI 91*
- Features:
  - performs on the scope of SEME regions
  - cost function based on frequency-weighted path lengths computed from a region-based dependence DAG
  - DAG construction makes use of PRDB
  - drives a number of IPF-specific optimizations, e.g. speculation
  - integrated with full resource management
- Global and local scheduling share the same implementation with difference in their scopes

Framework of Global Scheduling

- Build a global DAG for an entire SEME region
  - Nested regions folded with summary info
- Select target BBs to schedule based on their topological ordering and execution frequencies
- For each target BB, identify all source BBs and then candidate instructions
- For each cycle, select ready instructions in their priority to schedule
  - A forward, cycle scheduling
- Check the availability of machine resources to each selected instruction

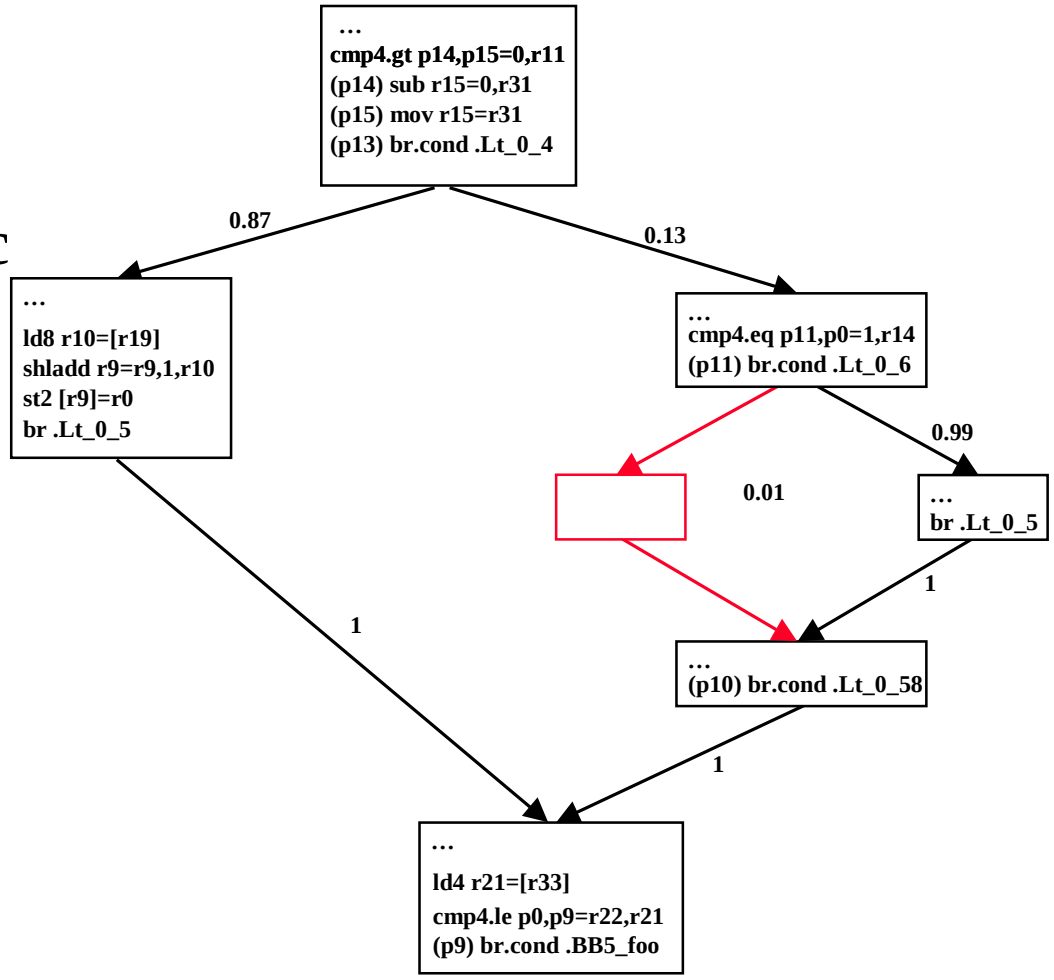
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



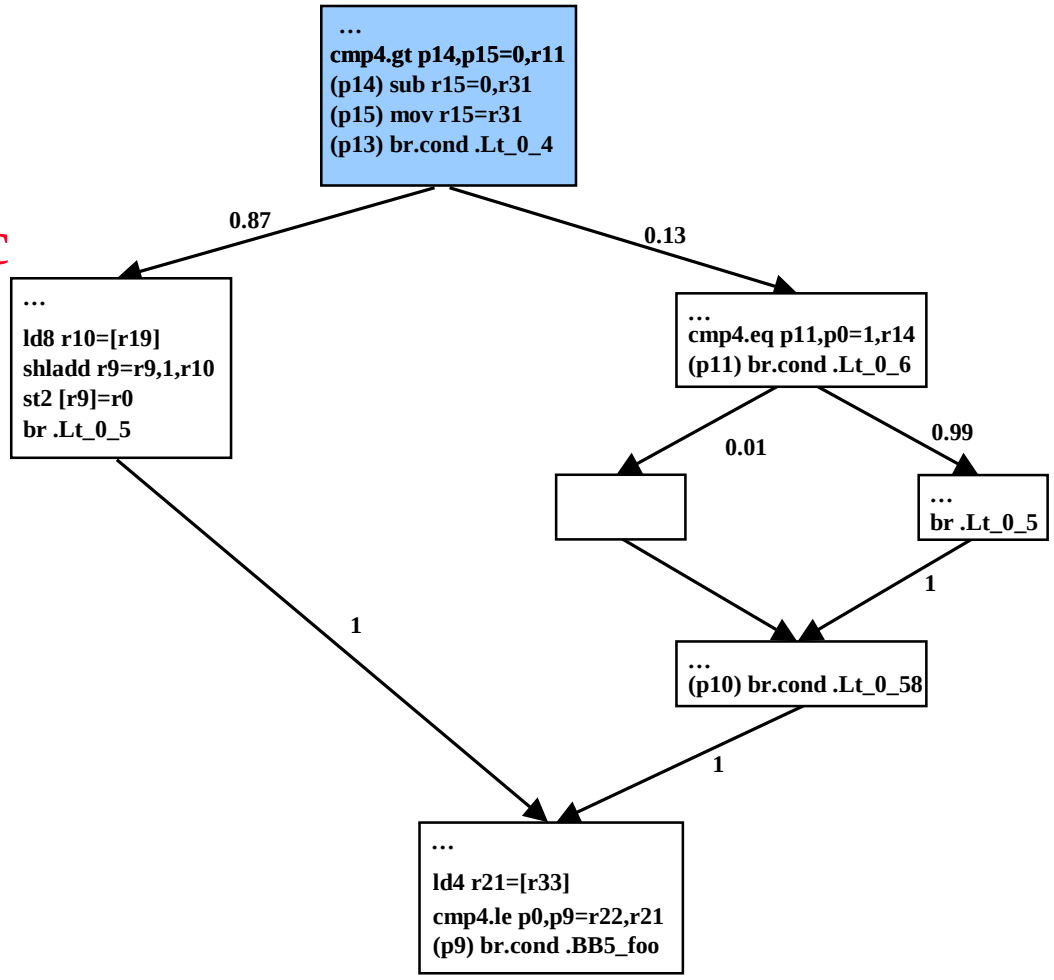
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



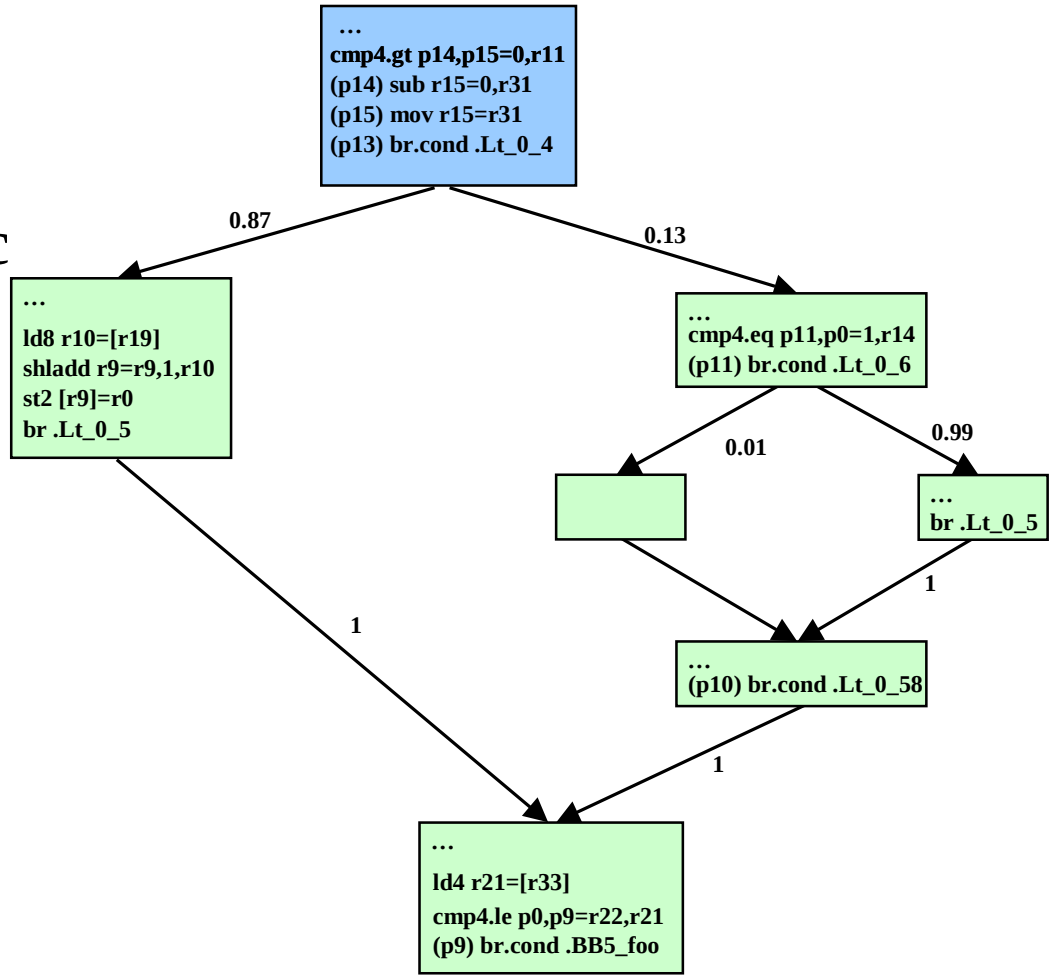
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



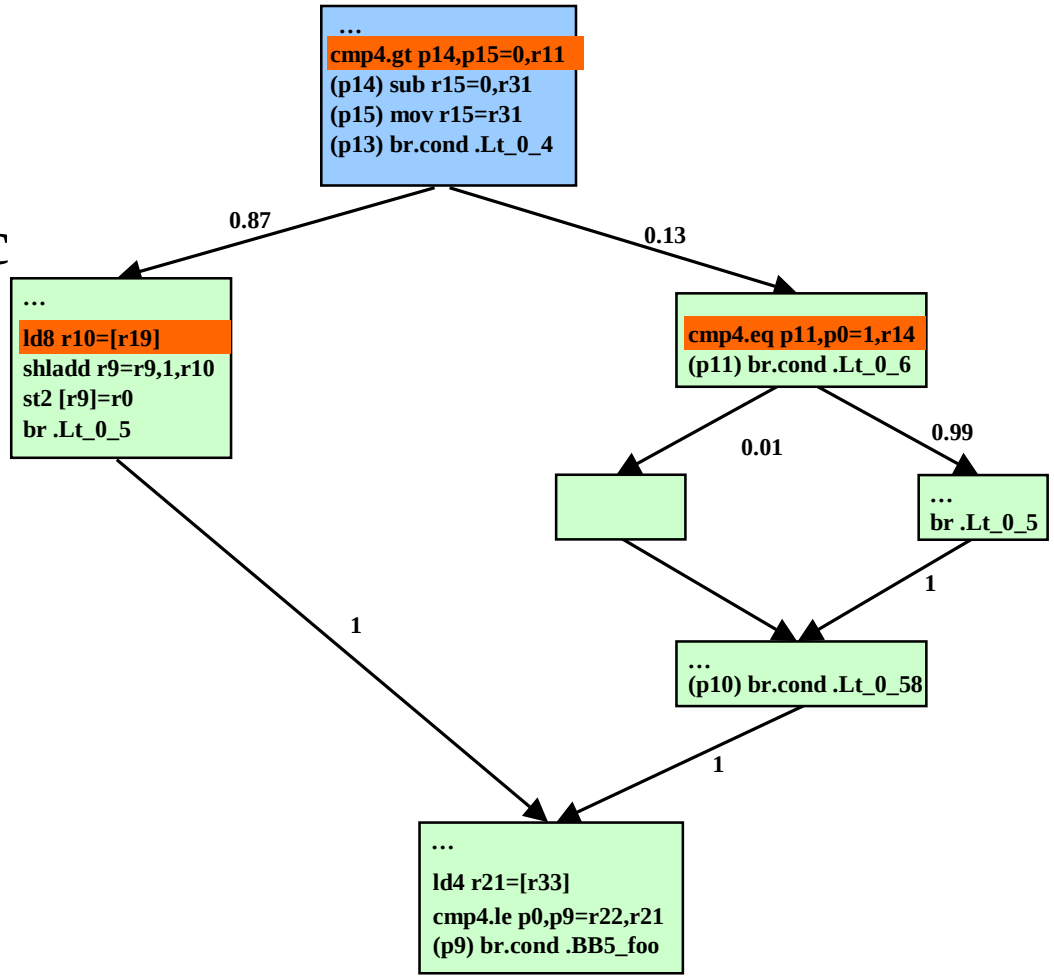
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



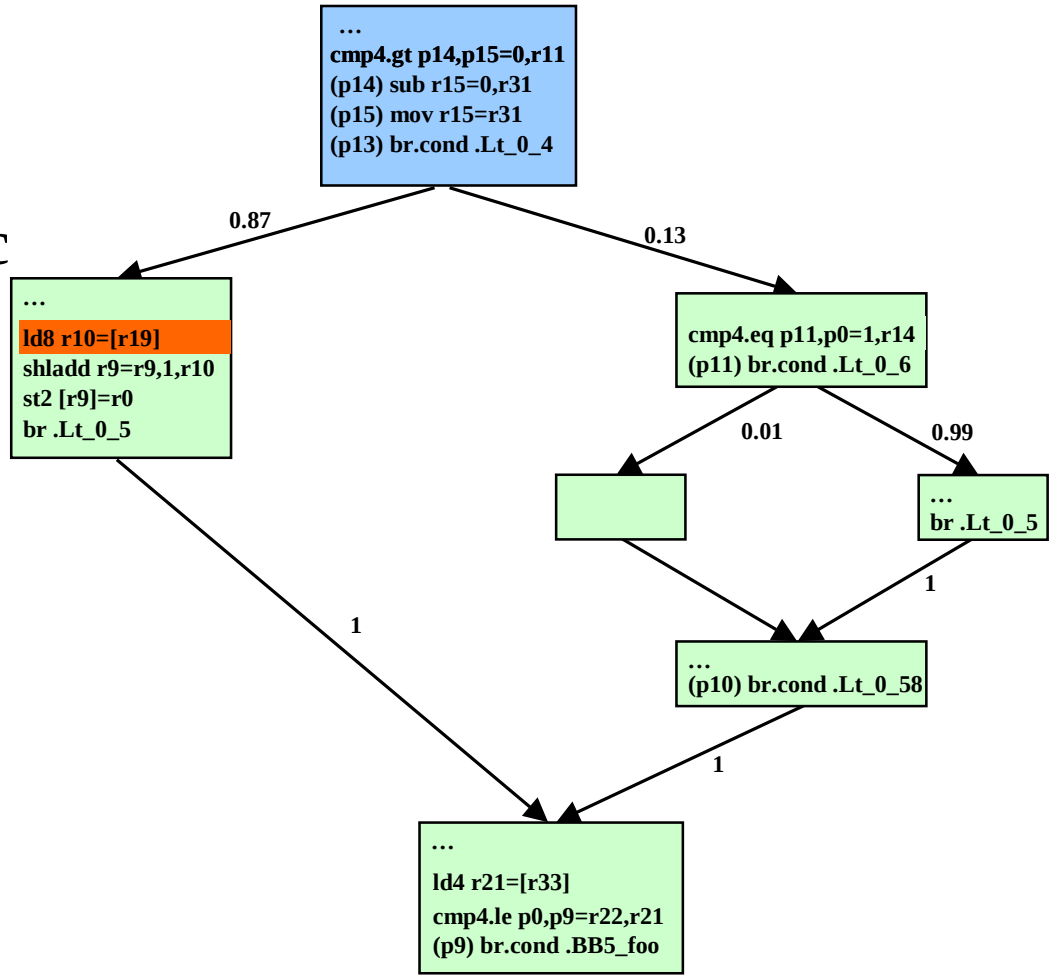
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



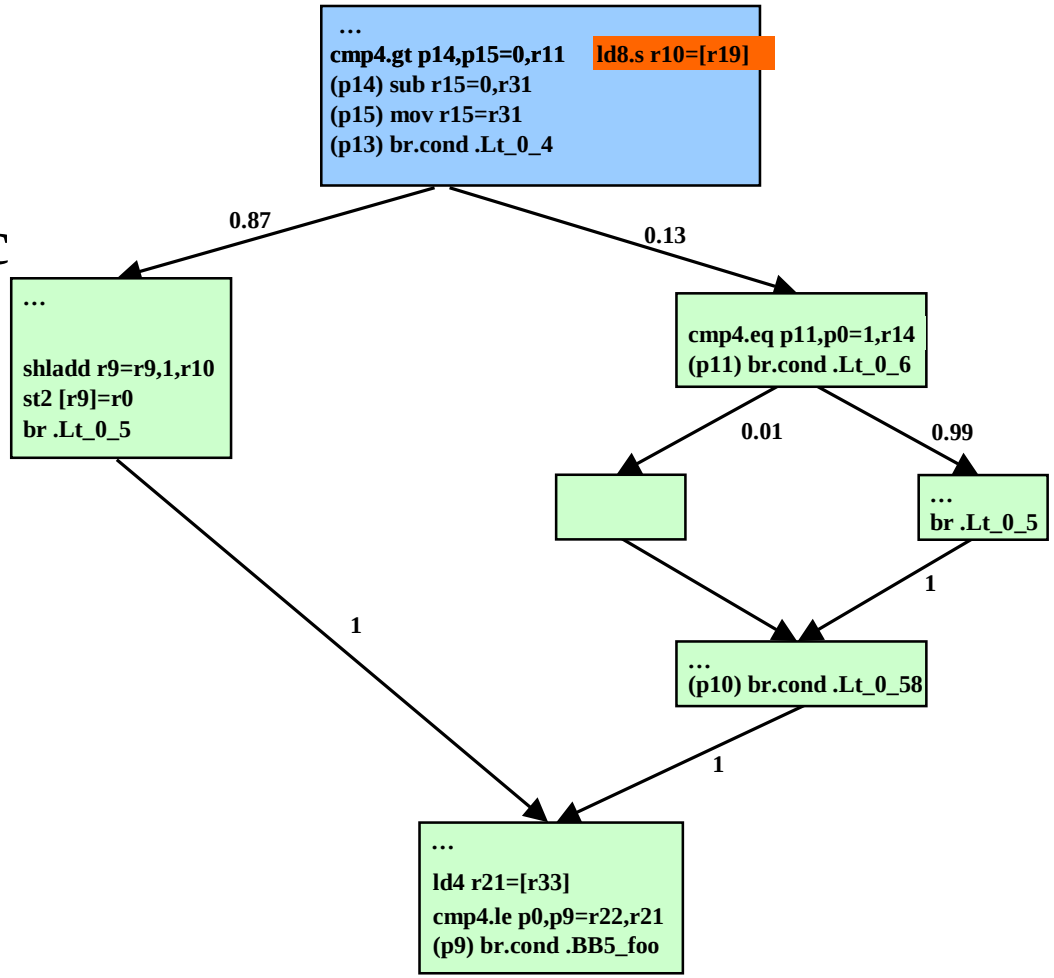
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



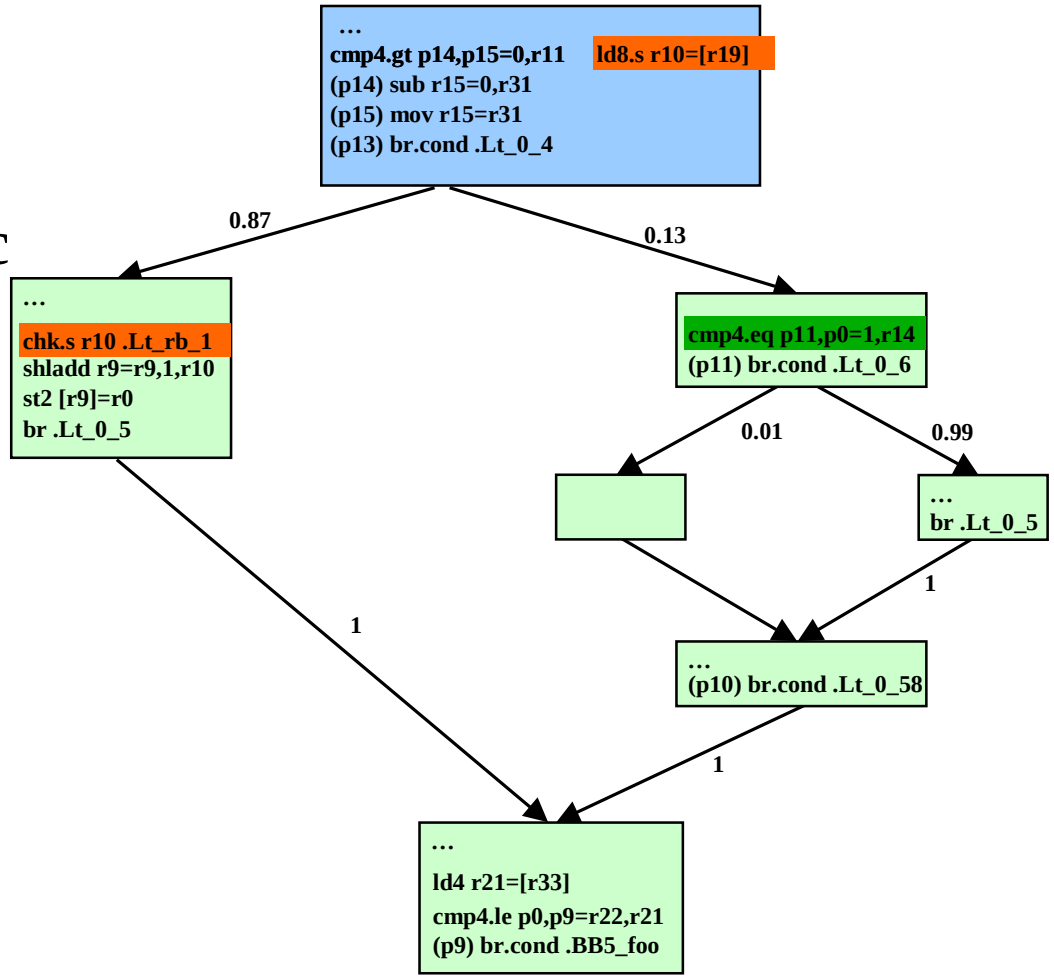
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



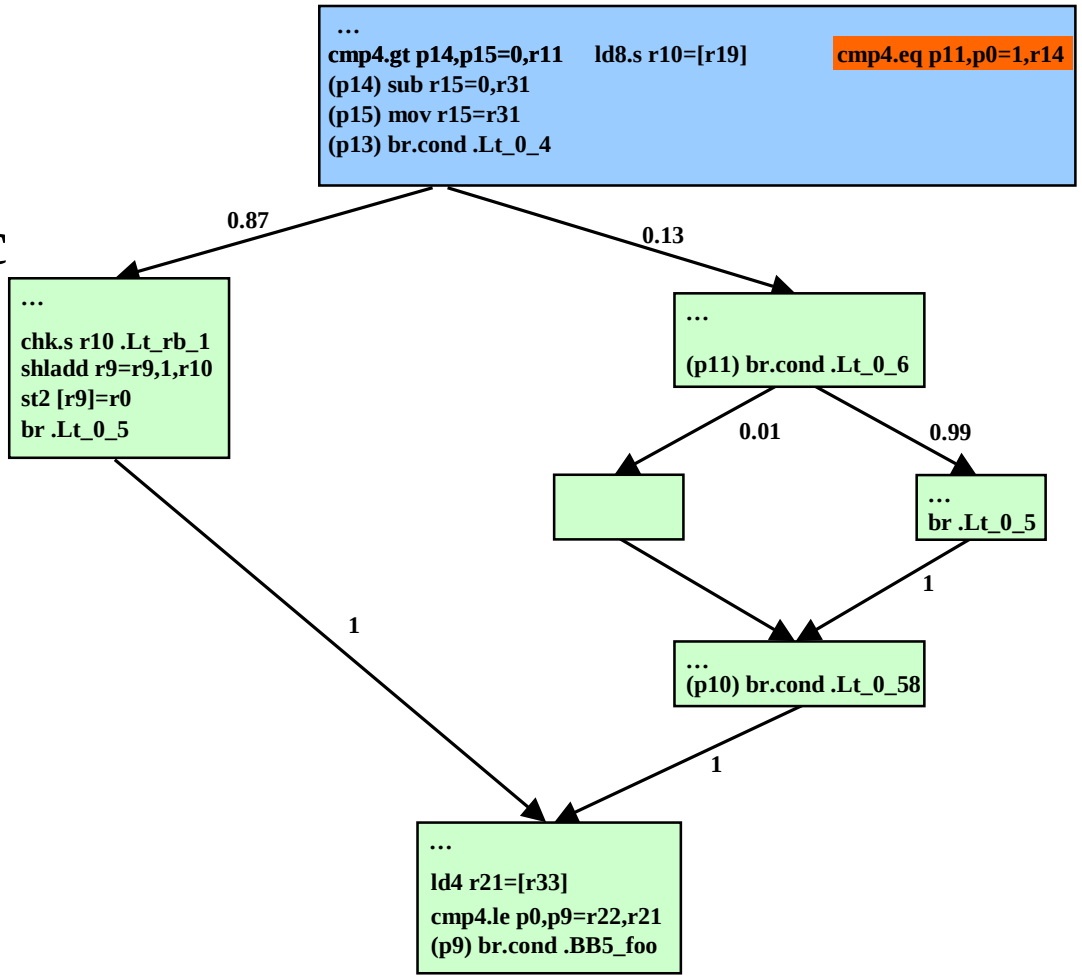
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



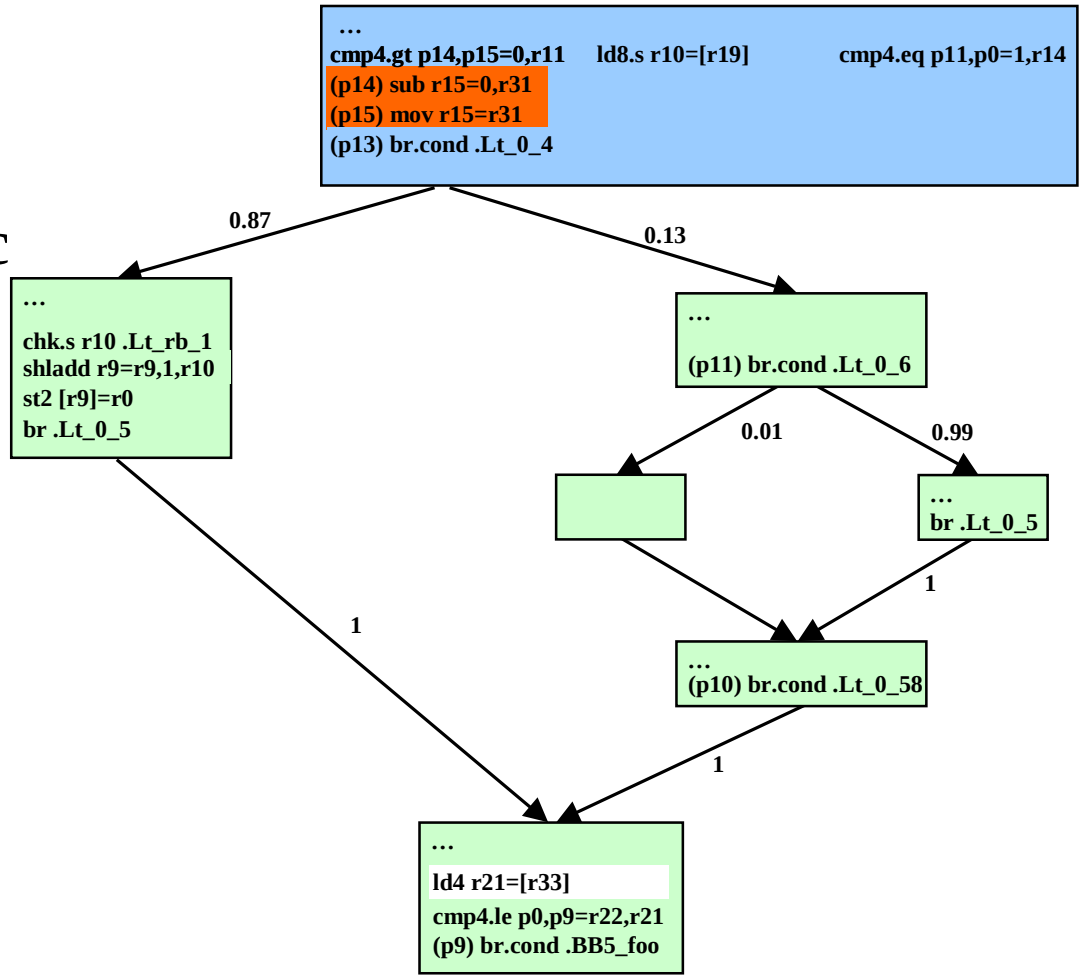
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



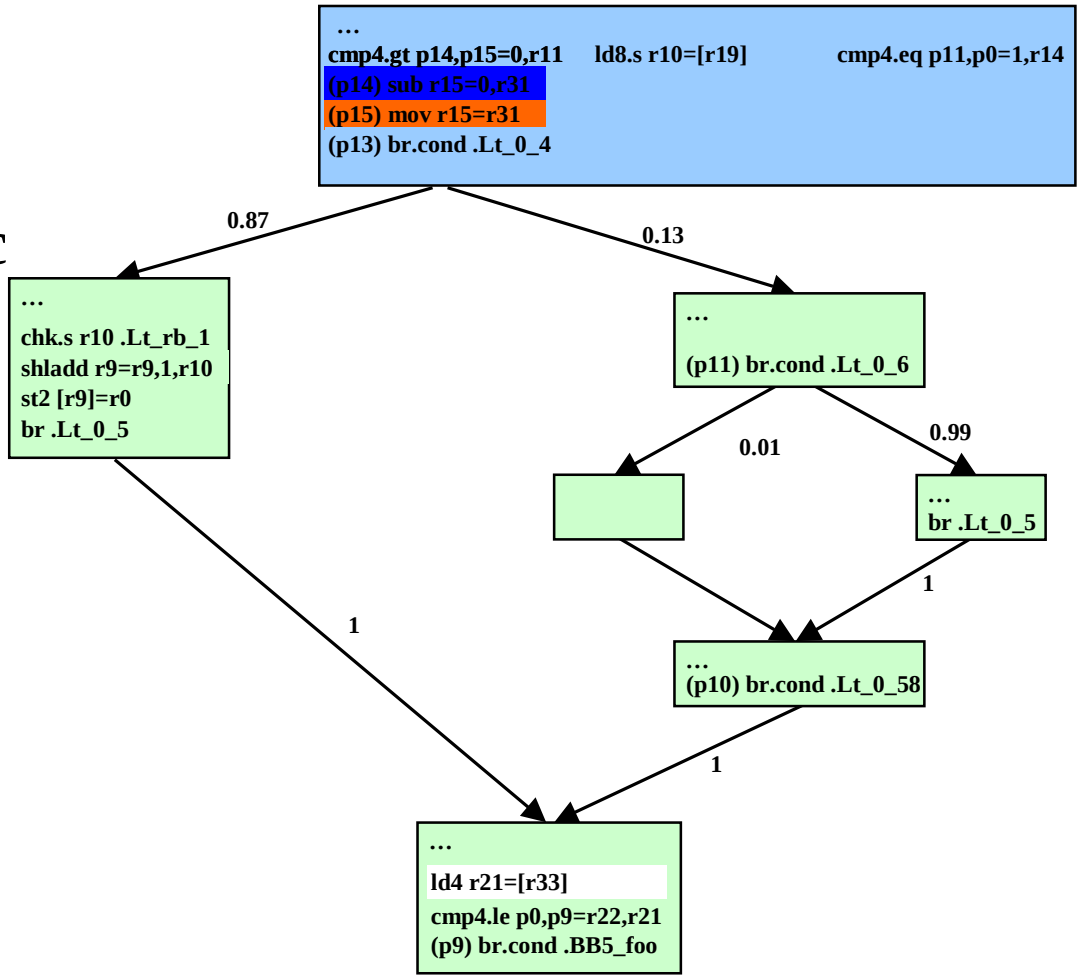
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



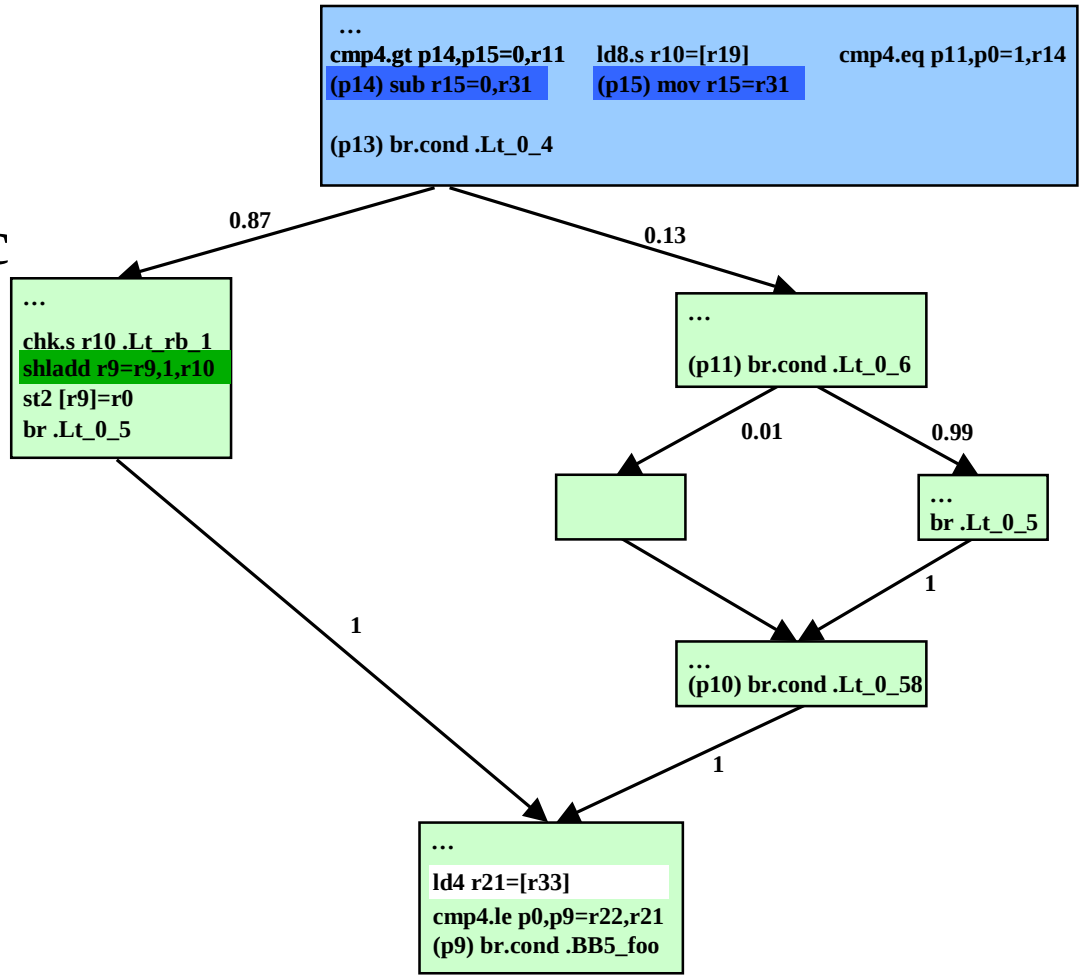
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



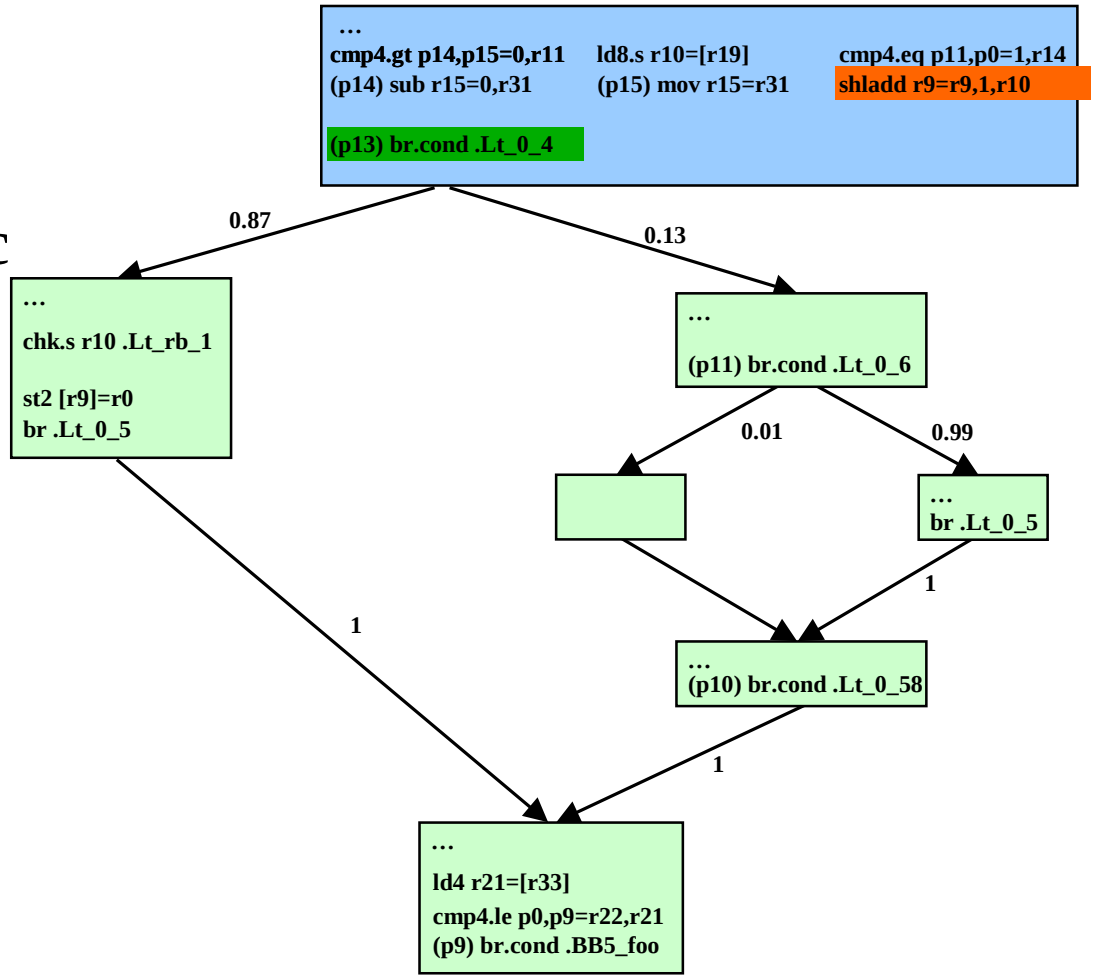
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



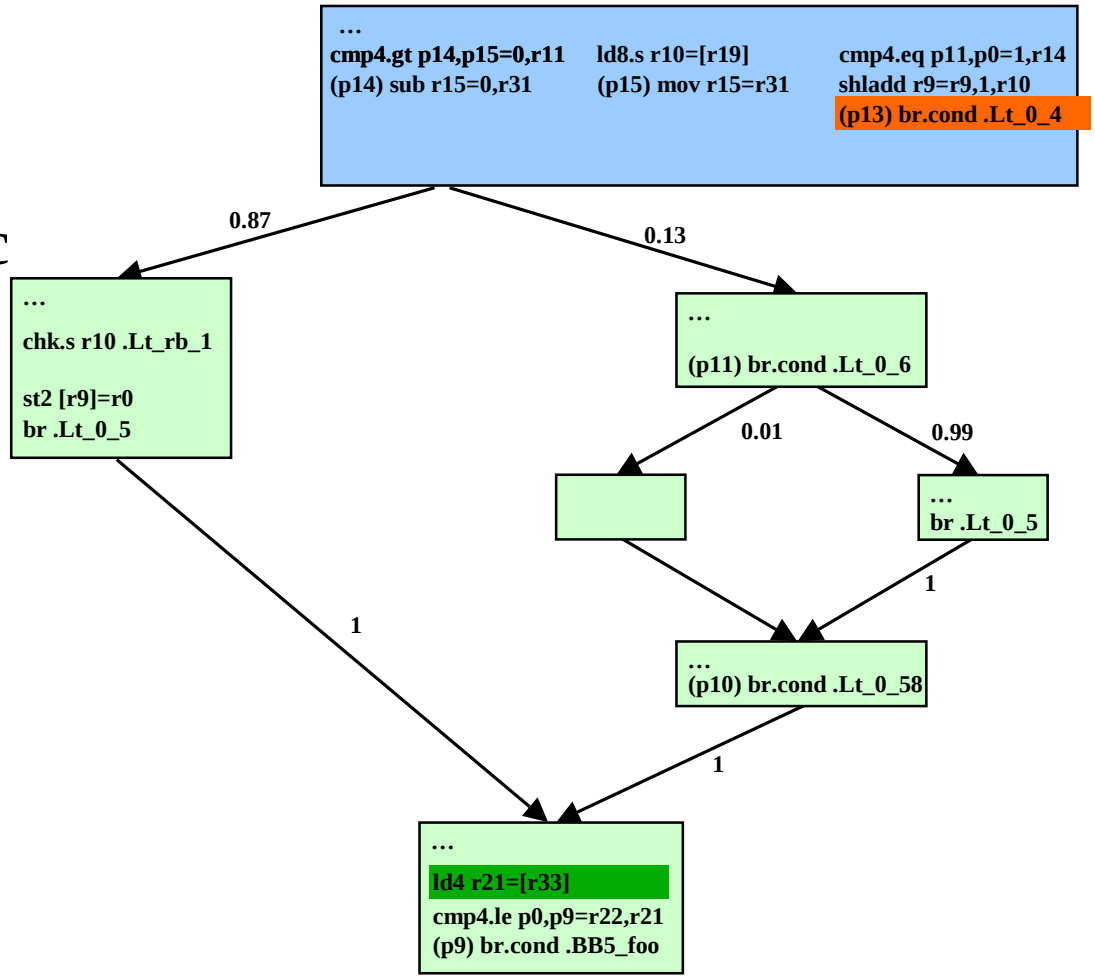
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



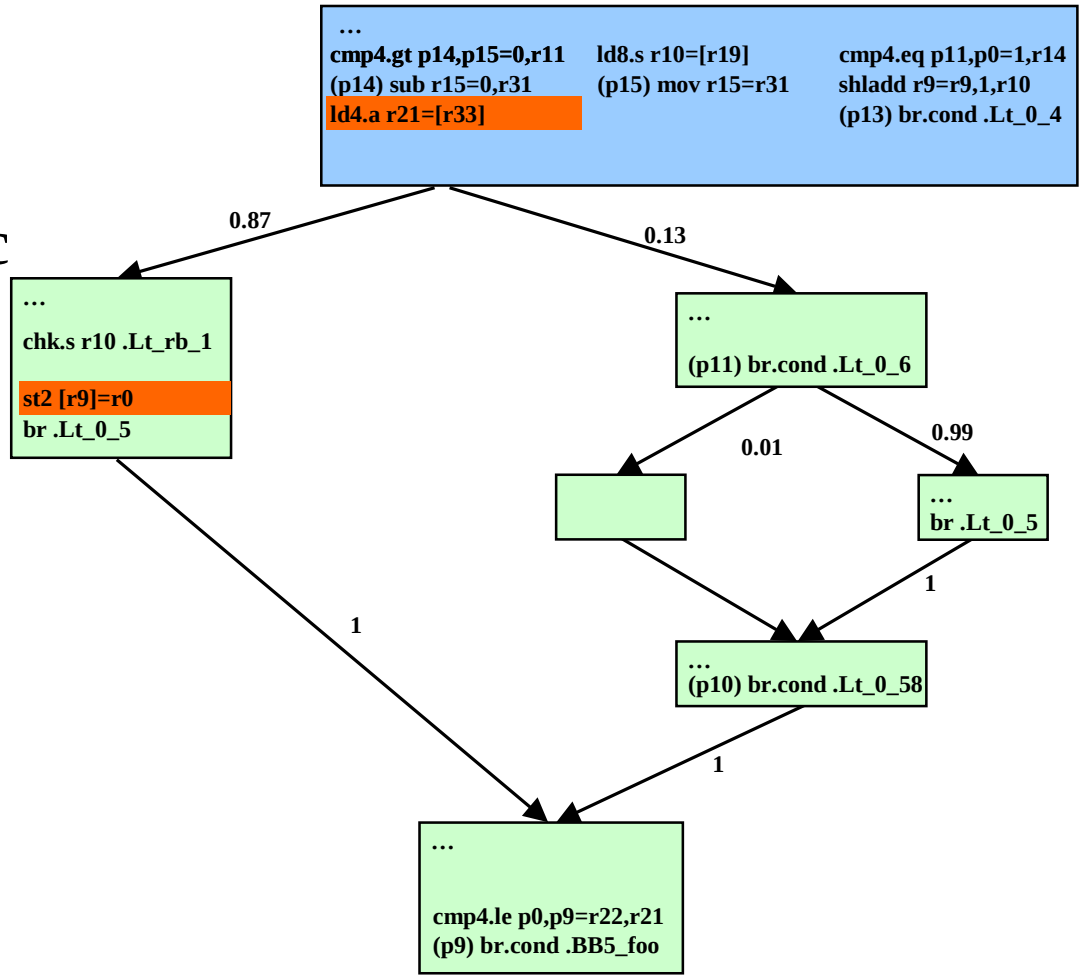
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



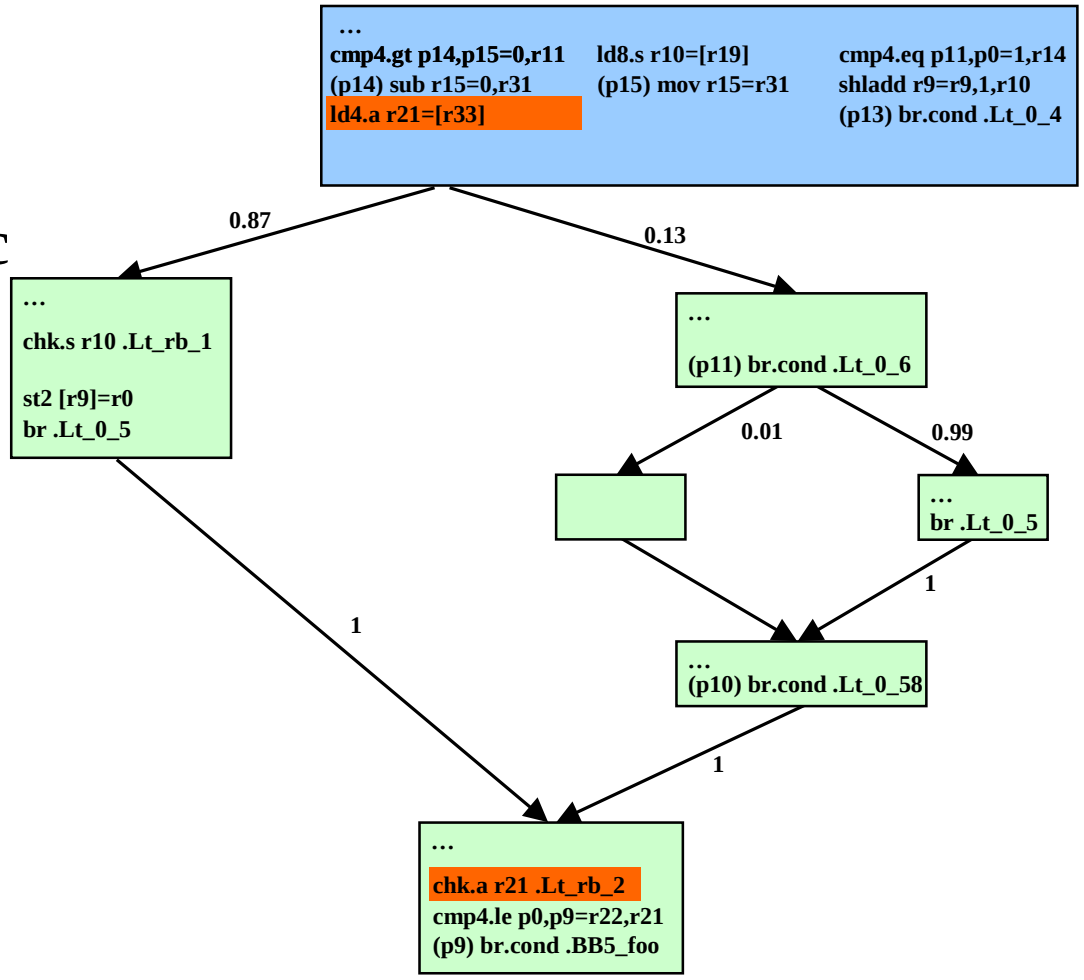
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



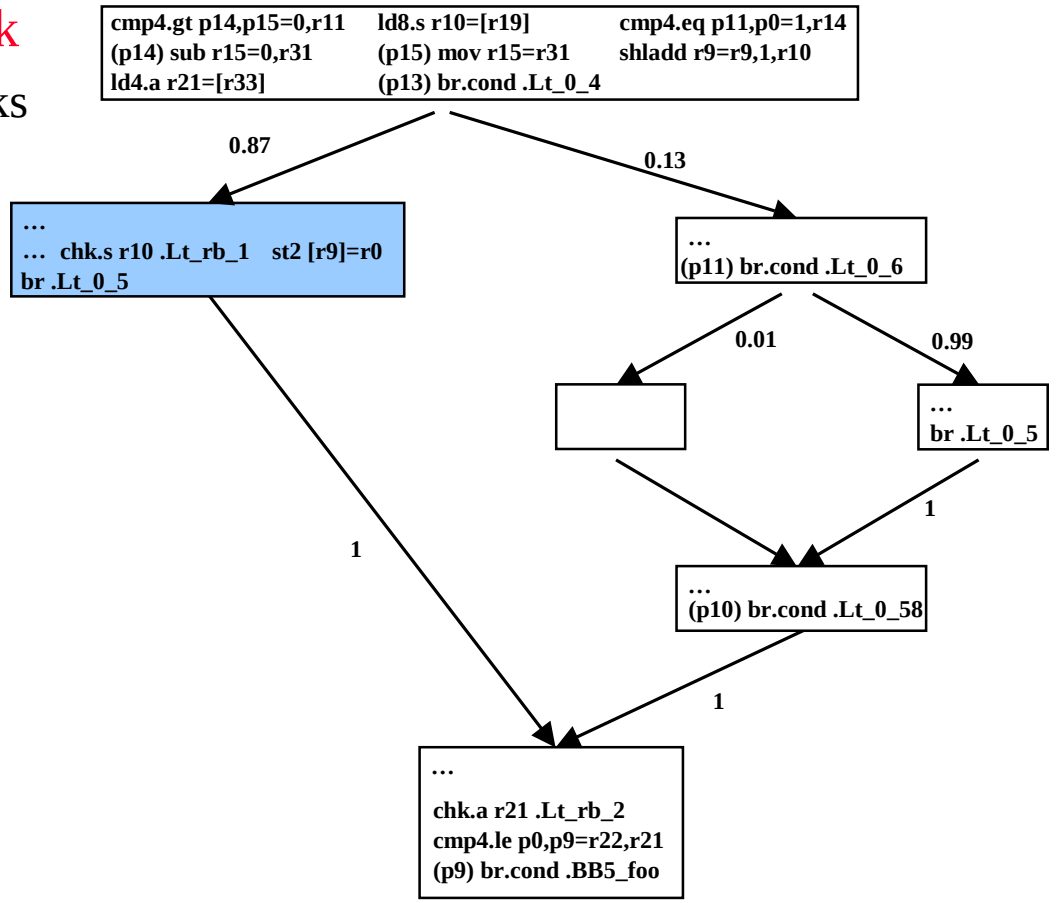
Process of Instruction Scheduling

- Critical edge splitting
- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Control speculation
  - Code motion
  - Motion of code with disjoint predicates
  - Code motion
  - Data speculation



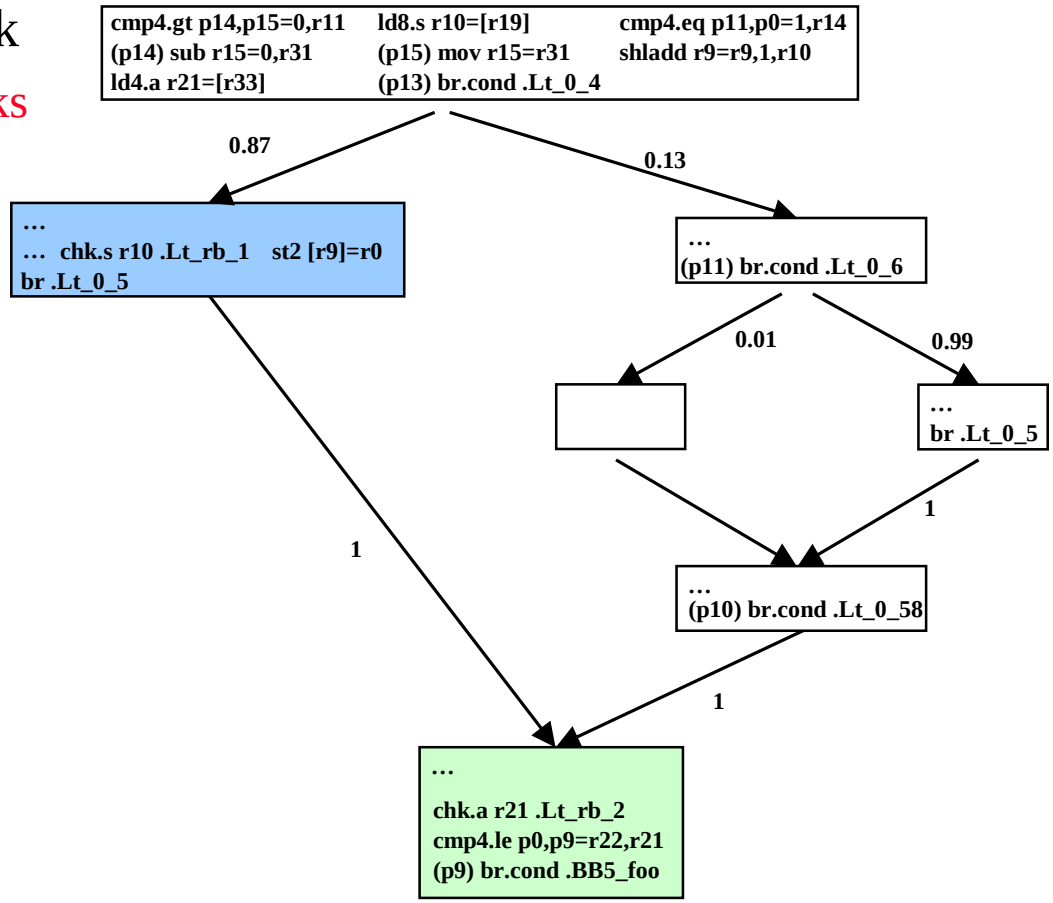
Process of Instruction Scheduling (Cont.)

- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Code motion
  - Compensation code generation



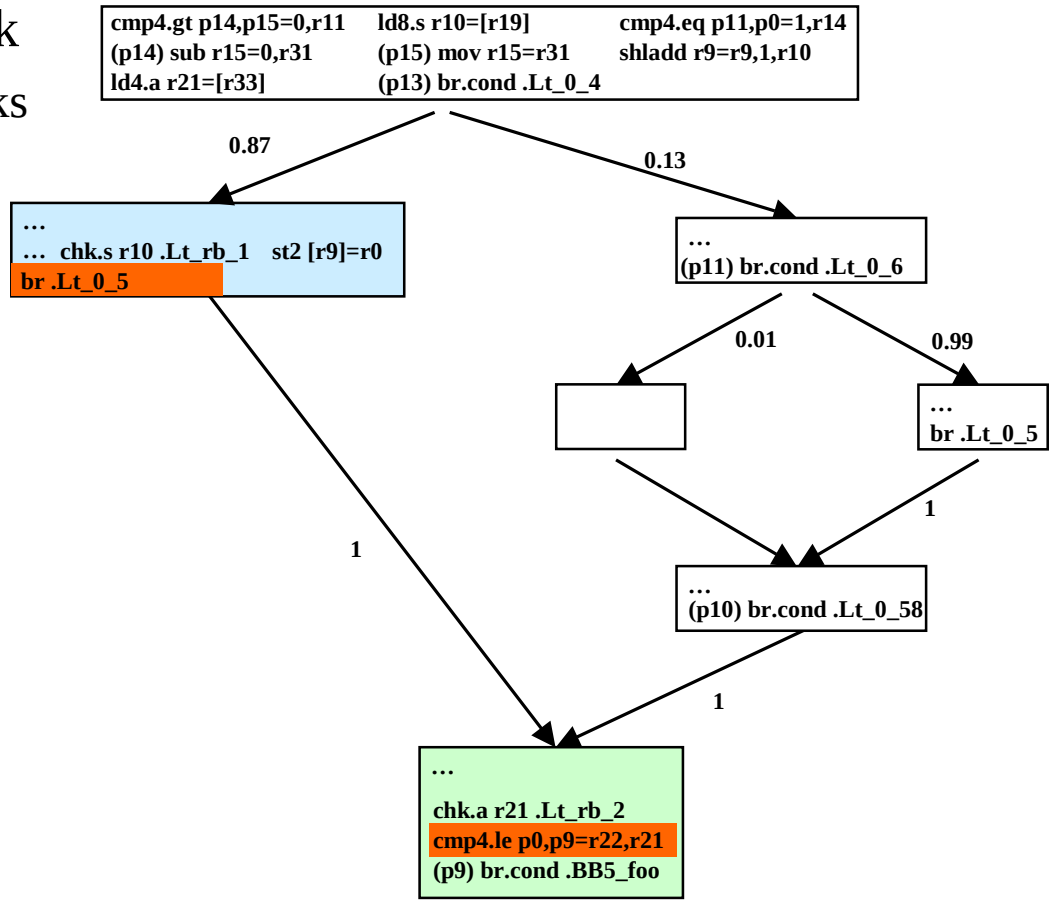
Process of Instruction Scheduling (Cont.)

- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Code motion
  - Compensation code generation



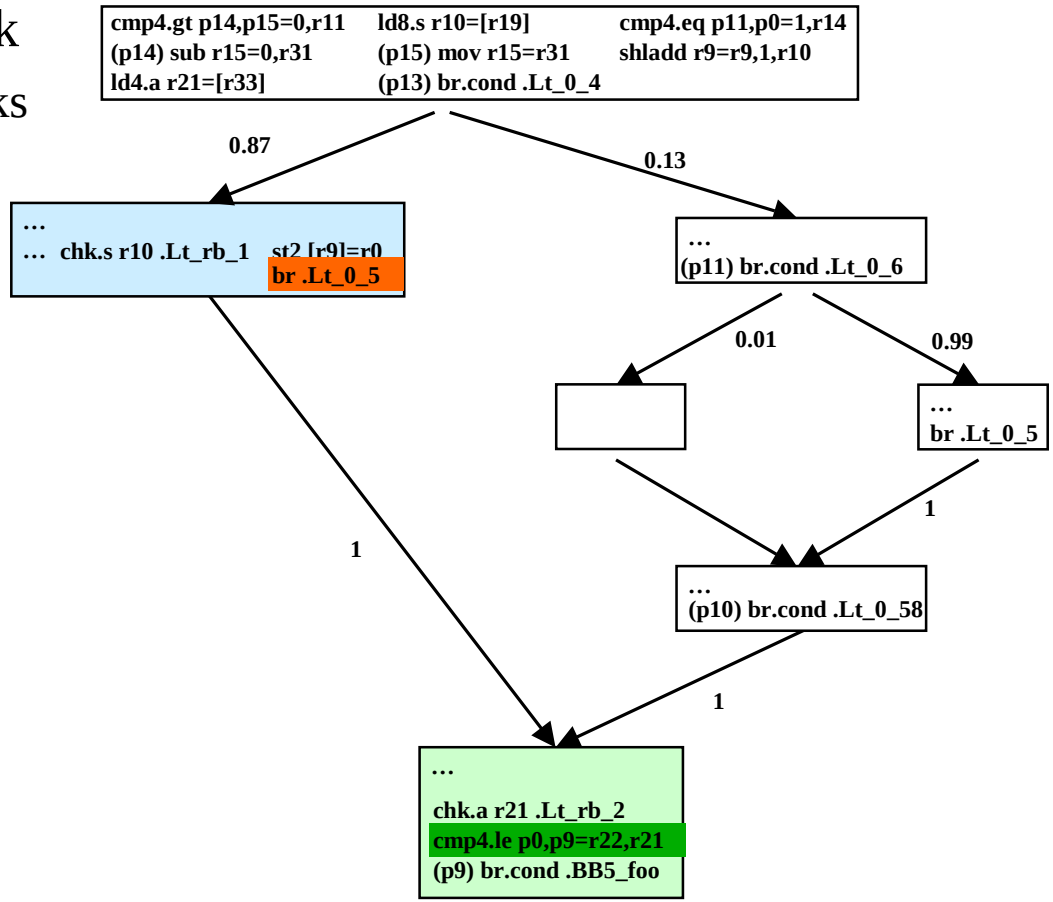
Process of Instruction Scheduling (Cont.)

- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Code motion
  - Compensation code generation



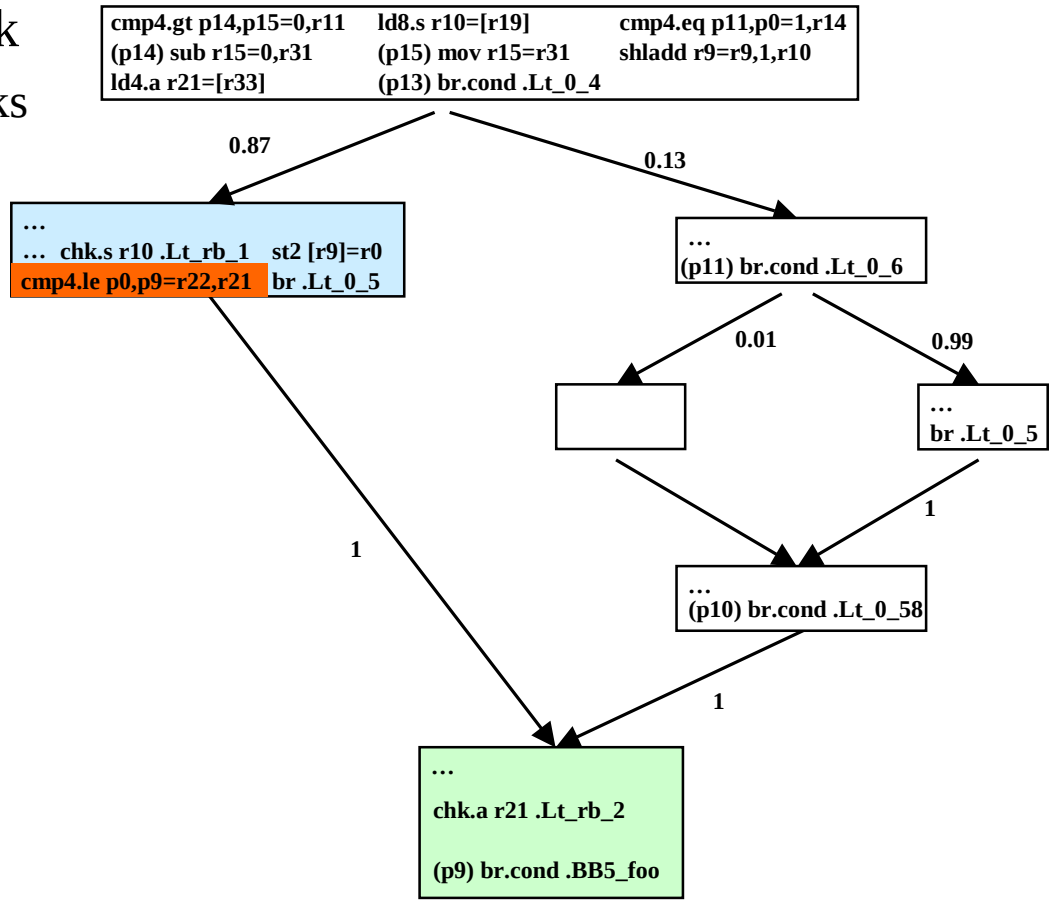
Process of Instruction Scheduling (Cont.)

- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Code motion
  - Compensation code generation



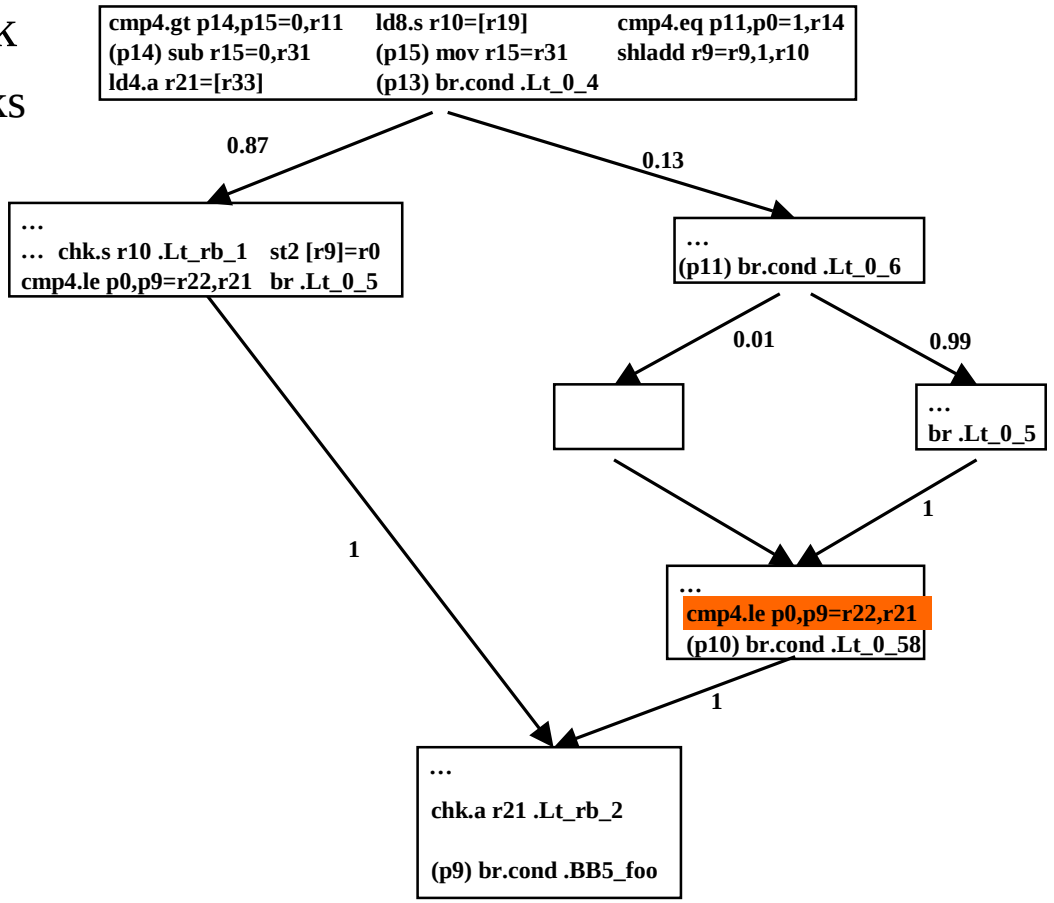
Process of Instruction Scheduling (Cont.)

- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Code motion
  - Compensation code generation



Process of Instruction Scheduling (Cont.)

- Choose target basic block
  - Find source basic blocks
  - Find candidates
  - Select best one
  - Code motion
  - Compensation code generation



Perspective Research Usage

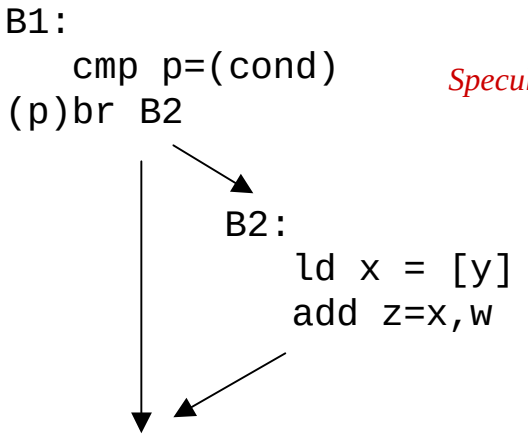
- Experiment with different scheduling heuristics
- Drive additional IPF optimizations
  - E.g. post-increment, multiway branch synthesis, …
- Be conscious about register pressure
- Replace it with your own scheduler
- Make it a standalone instruction scheduler
  - Connect it with other compilation systems, e.g. a binary translation system.

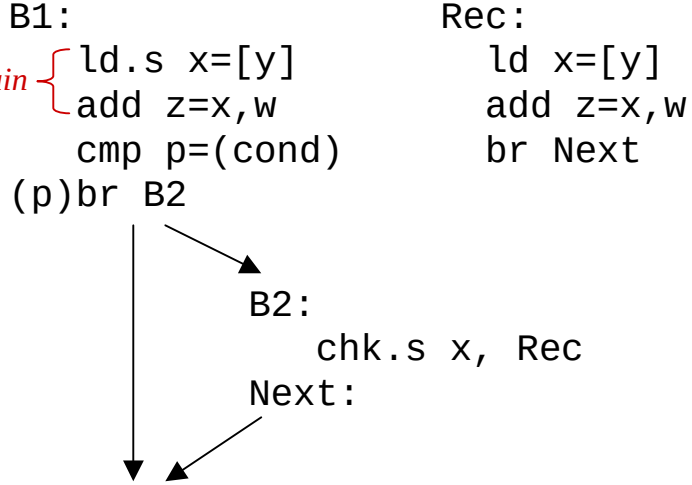
Control and Data Speculation

An introduction: Ju, Nomura, Mahadevan, and Wu, “A Unified Compiler Framework for Control and Data Speculation,” PACT 2000.

Architectural Support for Control Speculation

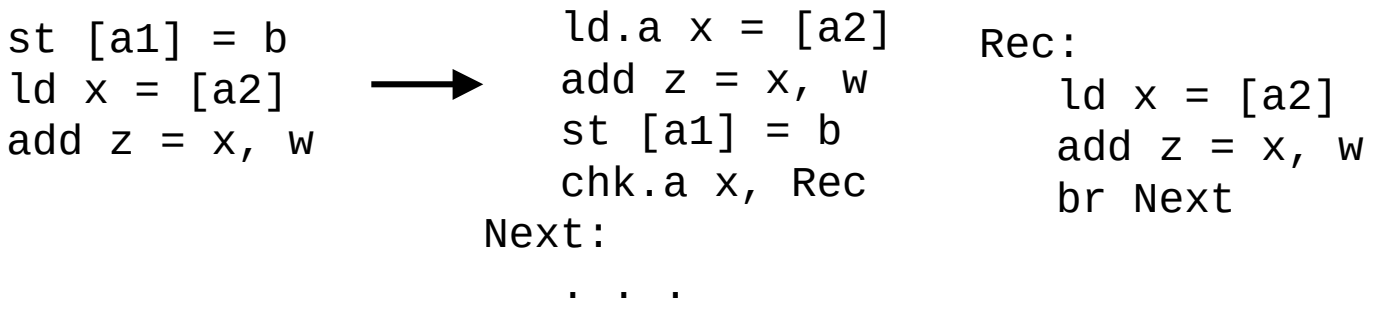
- NaT bits on registers
- Speculative and non-speculative versions of trapping instructions
- Speculative instructions to defer exceptions
- Check instructions, *chk.s*





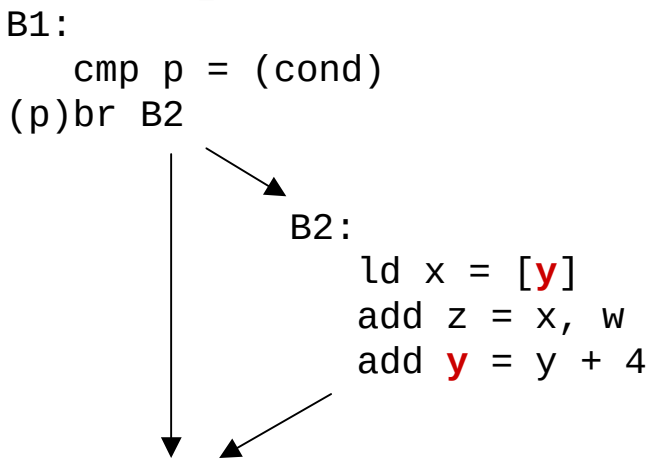
Architectural Support for Data Speculation

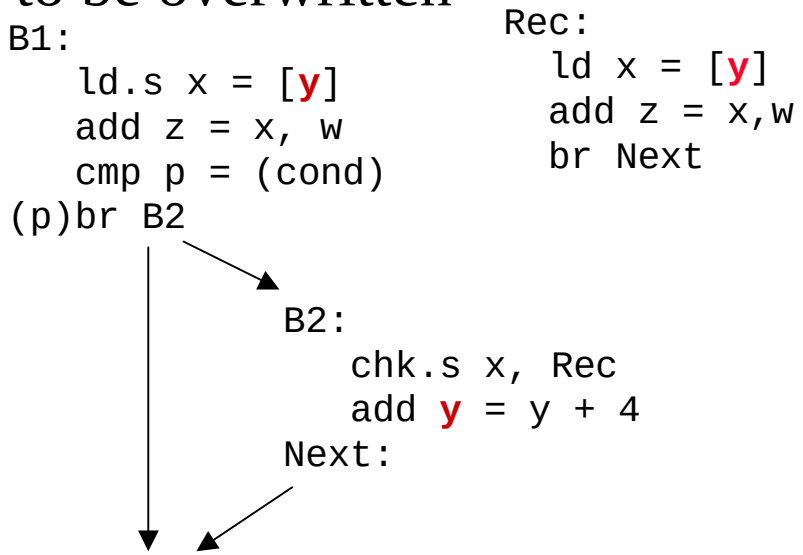
- Advanced (data speculative) load *ld.a*
- Advanced Load Address Table (ALAT)
- Store invalidates aliasing entries in ALAT
- Check instruction, *chk.a*



Compilation Issues for Control Speculation – Interferences and Live-in Values

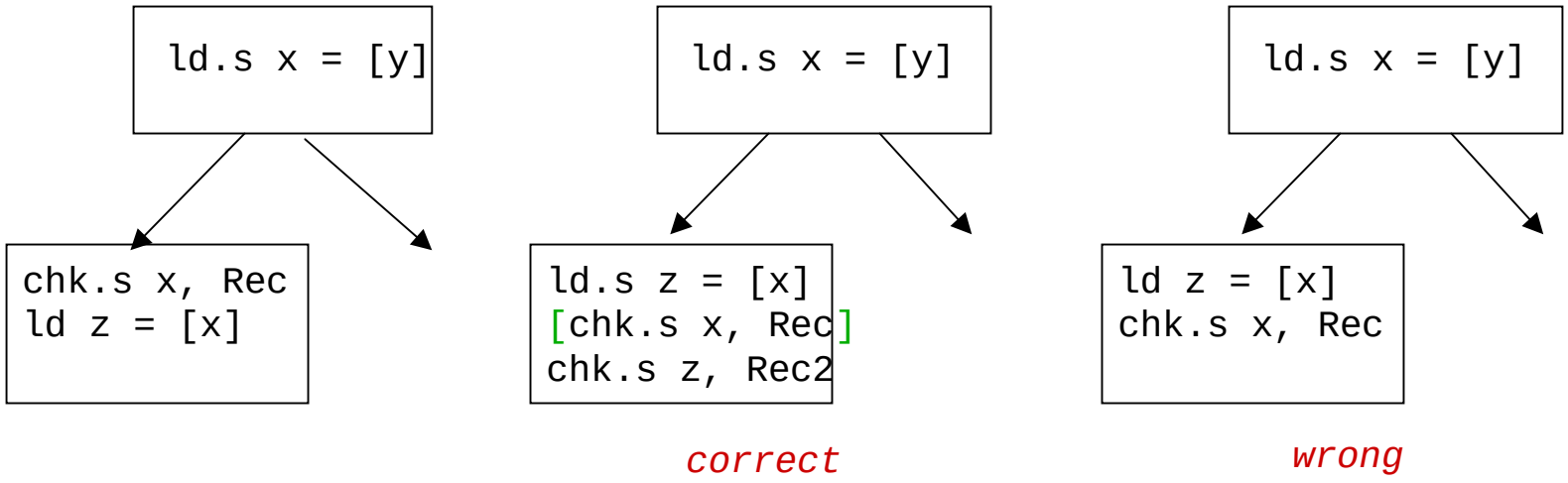
- Avoiding interferences for the destination registers of speculated instructions
- Recovering from a deferred exception
  - Upward-exposed values used in the speculative chain not to be overwritten





Compilation Issues for Control Speculation – ld across check

- Dependent load (to ld.s) scheduled across a check
  - The load must be put under speculative mode
  - Deferred exception propagated and not to be signaled before the first check



Compilation Issues for Data Speculation - Predication

- Excluding predicated code from a data speculative chain when the qualifying predicate is defined on the chain

```
   ld.a x = [a1]
   cmp p = x < y
   st [a2] = w
   chk.a x, Rec
(p)add z = b, c
```

```
   ld.a x = [a1]
   cmp p = x < y
(p)add z = b, c
   st [a2] = w
   chk.a x, Rec
```

Wrong – z may not be recoverable

```
Rec:
   ld x = [a1]
   cmp p = x < y
(p)add z = b, c
```

Cascaded Speculation

- A value defined by a speculative load directly or indirectly feeds into another speculative load scheduled before the first check
- Combinations in cascaded speculation
  - Control-speculation-led
  - Data-speculation-led

Cascaded Speculation

Control-Speculation-Led

- Different strategies to generate recovery code
- Code size vs. recovery overhead vs. ease of implementation

```
   ld.s x = [a]
   . . .
   ld.s y = [x]
[(p)chk.s x, Rec1]
c1:
(q)chk.s y, Rec2
c2:
```

```
Rec1:
   ld x = [a]
   . . .
   br c1

Rec2:
   ld y = [x]
   br c2
```

Strategy 1

```
Rec1:
   ld x = [a]
   ld.s y = [x]
   . . .
   br c2

Rec2:
   ld y = [x]
   br c2
```

Strategy 2

```
Rec2: // only
   ld x = [a]
   ld y = [x]
   . . .
   br c2
(w/o 1st chk.s;
NaT prop. to y)
```

Strategy 3

Cascaded Speculation

Data-Speculation-Led

- If the first load is mis-speculated, no NaT to propagate the fault
- The first recovery block to invalidate the second chk to ensure the second recovery block executed

```
   ld.a x = [a]
   . . .
   ld.sa  y = [x]
(p)chk.a x, Rec1
c1:
(q)chk.a y, Rec2
c2:
```

```
Rec1:
   ld x = [a]
   . . .
   invala y
   br c1

Rec2:
   ld y = [x]
   br c2
```

correct

```
Rec1:
   ld x = [a]
   . . .


Rec2:
   ld y = [x]
```

wrong

Scheduling Speculative Instructions

- Speculation is part of DAG-based list scheduling phase
- Marking speculative dependence edges for identified candidates during DAG construction
  - Control and data speculative edges
- Instruction is ready when all of its non-speculative predecessors scheduled
- Scheduler decides the loads to be speculated
  - Insert *chk* instruction
  - Add DAG edges from chk to the successors of speculated load to ensure recoverability

Recovery Code Generation

- Recovery code generation decoupled from scheduling phase
  - Reduce the complexity of the scheduler
- To generate recovery code
  - Starting from the speculative load, follow flow and output dependences to re-identify speculated instructions
  - Duplicate the speculated instructions to a recovery block under the non-speculative mode
- Once a recovery block is generated, avoid changes on the speculative chain
- Allow GRA to properly color registers in recovery blocks

Parameterized Machine Model

Machine Model

- Motivations:
  - To centralize the architectural and micro-architectural details in a well-interfaced module
  - To facilitate the study of hardware/compiler co-design by changing machine parameters
  - To ease the porting of ORC to future generations of IPF
- Two aspects:
  - Parameterized machine descriptions
  - Micro-scheduler to model resource constraints

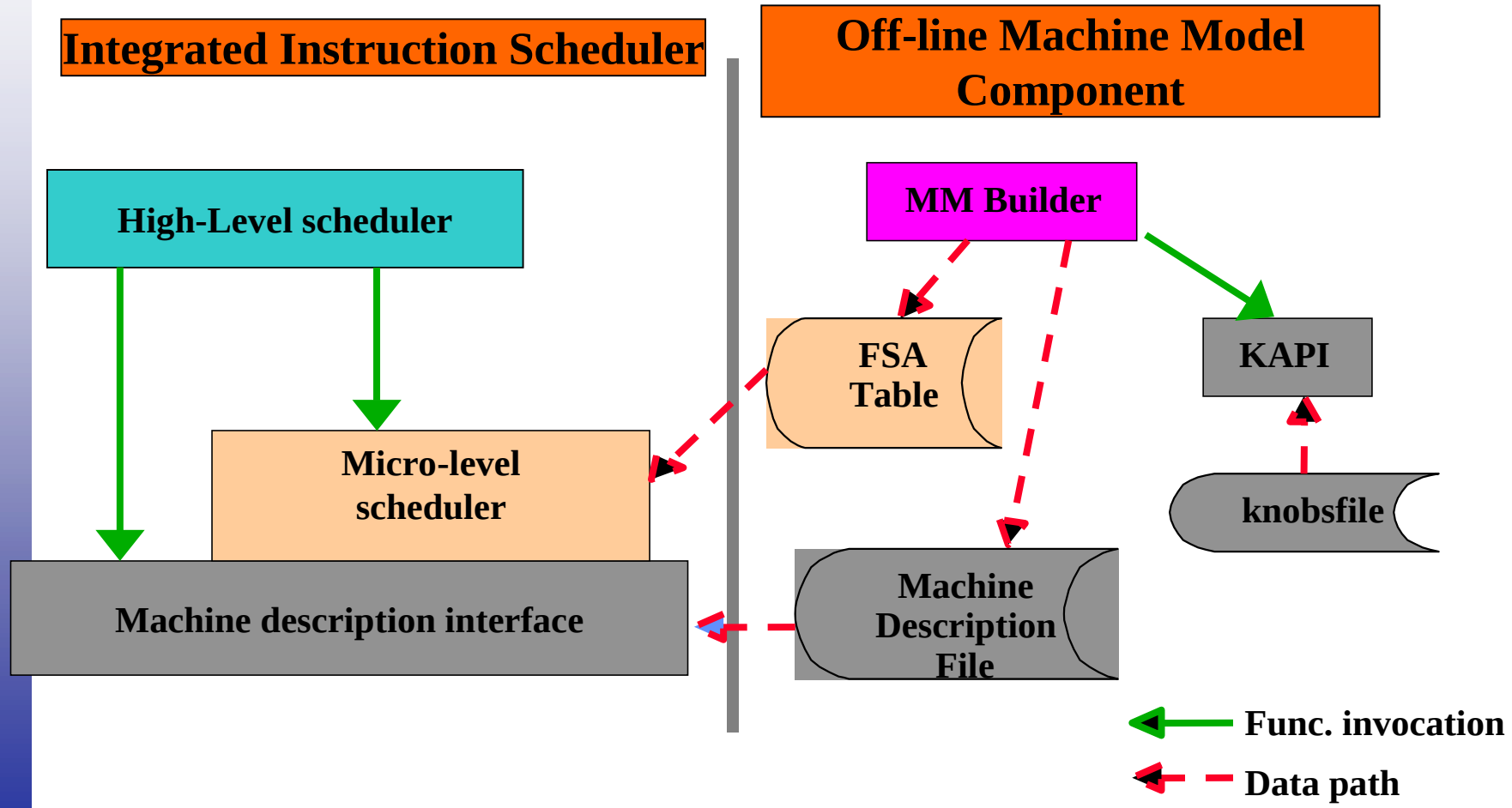
Machine Descriptions

- Read in the (micro-)architecture parameters from KAPI (Knobsfile API) published by Intel
  - E.g. machine width, FU class, latencies, templates, bypass …
  - In v26-itanium-41-external.knb
- Keep additional hardware specifications in a separate file
  - E.g. Pro64 opcode, registers, # of issue slots, …
  - In v11-itanium-extra.knb
- Automatically generate the machine description tables in Pro64
  - E.g. targ_isa_….[c|h|exported], targ_proc…, topcode…
- Avoid multiple changes of the same info duplicated into different tables
- The machine descriptions can be consumed by various optimization phases

Micro-Scheduler

- Manage resource constraints
  - E.g. templates, dispersal rules, FU's, machine width, …
- Model instruction dispersal rules
- Interact with the high-level instruction scheduler
  - Yet to be integrated with SWP
- Reorder instructions within a cycle
- Use a finite state automata (FSA) to model the resource constraints
  - Each state represents occupied FU's
  - State transition triggered by incoming scheduling candidate

Modules in Machine Model



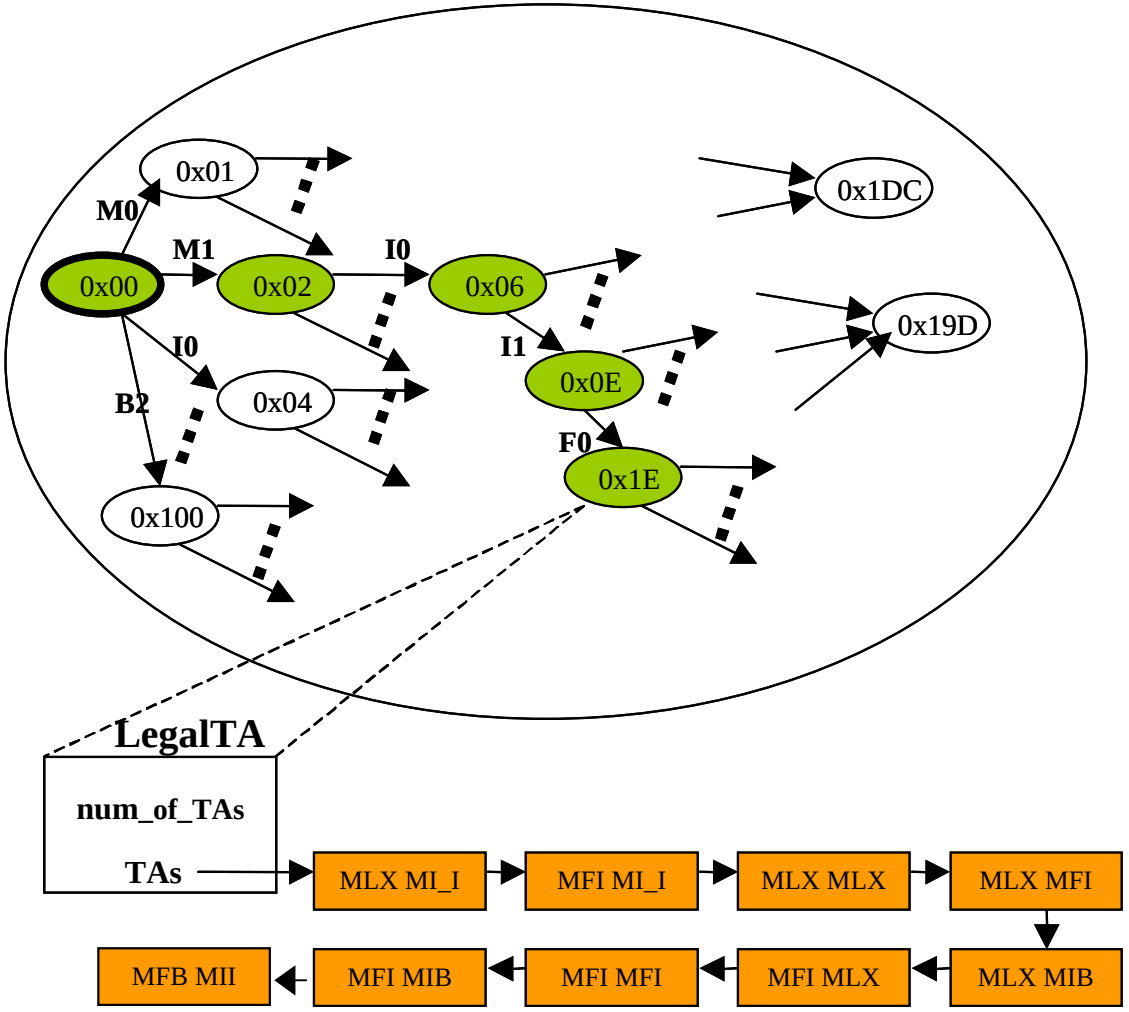
Functional-unit Based FSA

- Generate FSA prior to compilation
- Model resource constraints
- FSA states based on occupied FU's for space efficiency
- Each state contains a list of legal template assignments
  - Sorted in a priority order, e.g. for code size
- State transition triggered
  - Incoming scheduling candidate
  - Reordering to obtain the needed FU's

Functional-unit Based FSA

- Template assignment not selected except for
  - Intra-cycle (0-cycle) dependence
  - Finalizing template assignment with 1-cycle delay
- Able to utilize compressed templates
- For Itanium, 2 bundles per cycle
  - ORC FSA has 235 states
  - Each state has at most 38 valid template assignments.
  - 75% if the states have < 10 assignments
- Changing the machine parameters, e.g. machine width, will generate a new FSA automatically

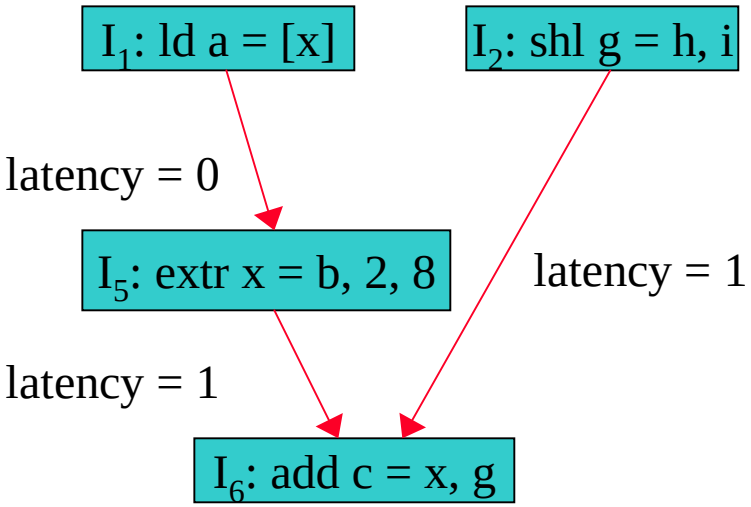
Functional-unit Based FSA (Example)



Integrated Instruction Scheduling

- Instruction scheduling integrated with full resource management through micro-scheduler
- Repeatedly pick the best candidate based on scheduling cost function
- Micro-scheduler to make state transition in FSA to check the availability of resources
- Micro-scheduler may permute FU assignments to meet the dispersal rules
- If the resource constraints met, the scheduler can choose to commit the candidate
- If resources fully utilized or no ready candidate available, the scheduler advance to schedule the next cycle
- Template assignment finalized with 1-cycle delayed

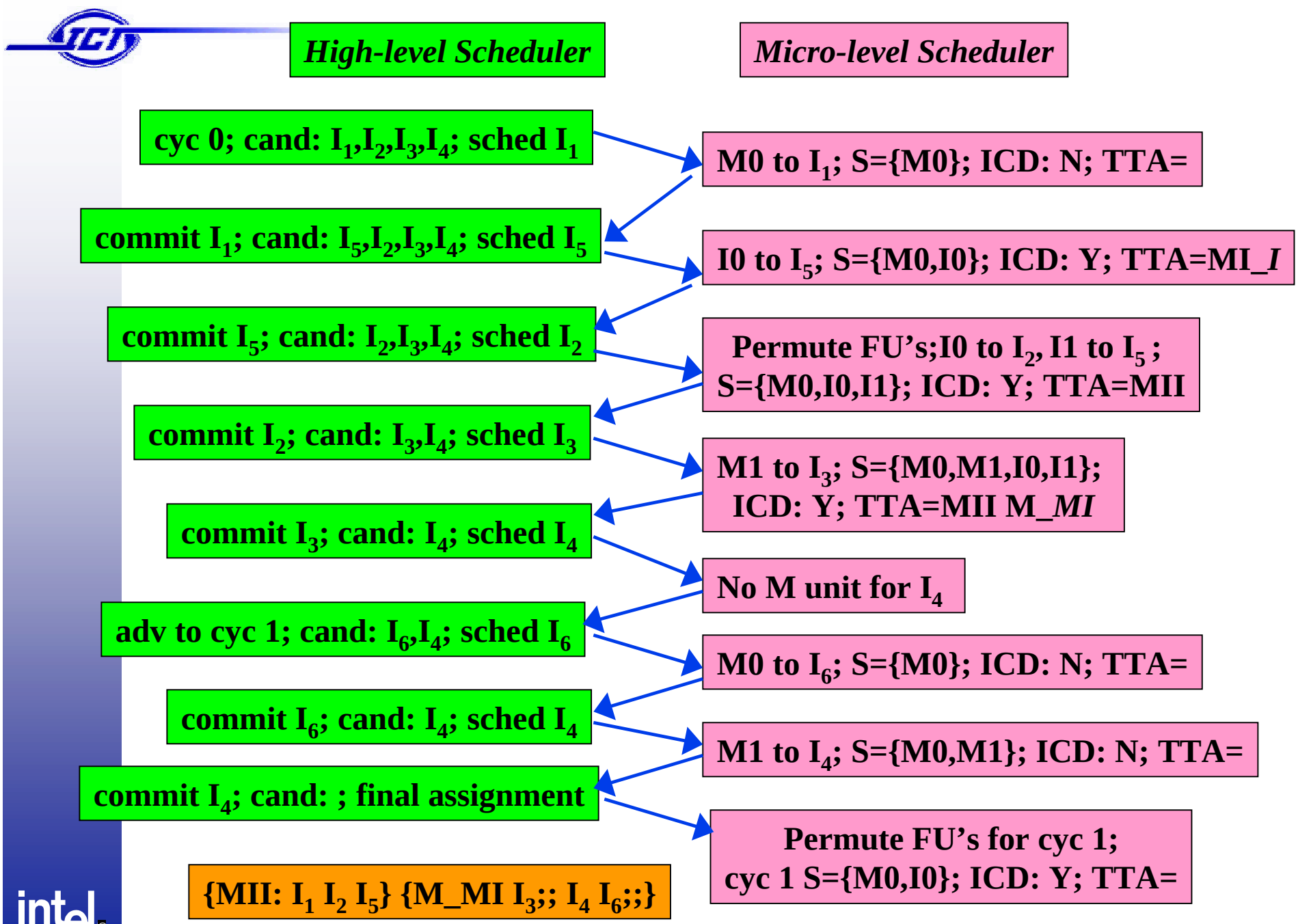
Integrated Scheduling - Example



Valid FUs for instructions

I_1,I_3,I_4: M0, M1
I_2: I0
I_5: I0,I1
I_6: M0,M1,I0,I1

FSA state: {}
Intra-cycle dependence (ICD): N or Y
Tentative template assignment (TTA)


High-level Scheduler
Micro-level Scheduler
cyc 0; cand: I1,I2,I3,I4; sched I1
M0 to I1; S={M0}; ICD: N; TTA=
commit I1; cand: I5,I2,I3,I4; sched I5
I0 to I5; S={M0,I0}; ICD: Y; TTA=MI_I
commit I5; cand: I2,I3,I4; sched I2
Permute FU's;I0 to I2, I1 to I5 ;
S={M0,I0,I1}; ICD: Y; TTA=MII
commit I2; cand: I3,I4; sched I3
M1 to I3; S={M0,M1,I0,I1};
ICD: Y; TTA=MII M_MI
commit I3; cand: I4; sched I4
No M unit for I4
adv to cyc 1; cand: I6,I4; sched I6
M0 to I6; S={M0}; ICD: N; TTA=
commit I6; cand: I4; sched I4
M1 to I4; S={M0,M1}; ICD: N; TTA=
commit I4; cand: ; final assignment
Permute FU's for cyc 1;
cyc 1 S={M0,I0}; ICD: Y; TTA=
{MII: I1 I2 I5} {M_MI I3;; I4 I6;;}

- Overview of ORC
- New Infrastructure Features
- New IPF Optimizations
- **Research Case Study**
- Demo of ORC
- Release and Future Plans

Instruction Scheduling and Resource Management

Integrated vs. Decoupled

Research Case Study

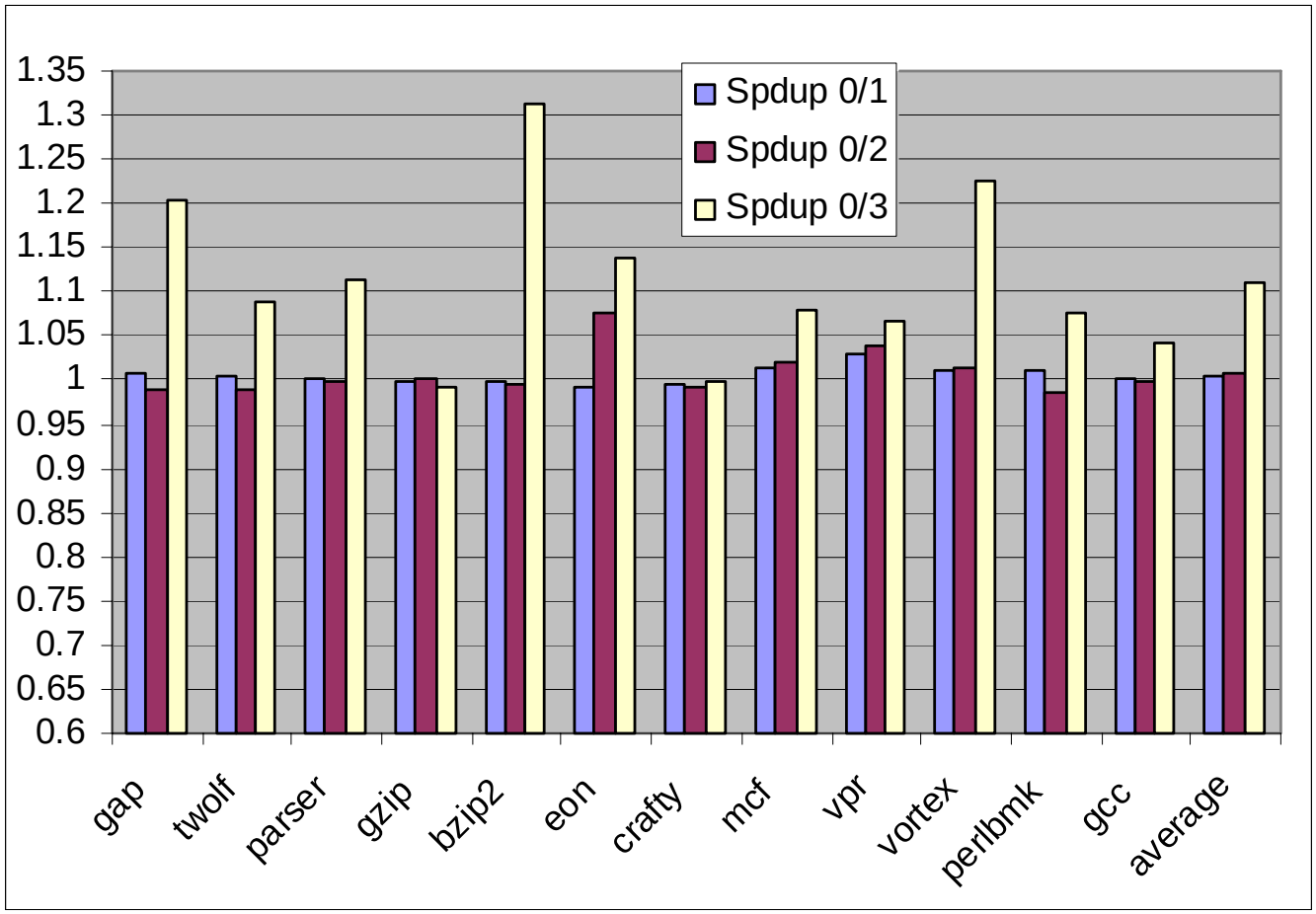
- Want to demonstrate the advantages of scheduling integrated with resource management
- To contrast with decoupled approaches
  - Traditional scheduler followed by a separate bundling phase
- Minimize the effort to implement the decoupled approaches
- Build an independent bundling phase using micro-scheduler
  - Select template assignments for scheduled intructions
  - Honor the cycle breaks placed by the traditional scheduler
  - Trivial effort
  - More powerful than the *handle_hazard()* in Pro64

Scheduling/Bundling Approaches

- Level 0: scheduler w/o any resource management + separate bundling
- Level 1: scheduler w/ machine width constraint + separate bundling
- Level 2: scheduler w/ constraints on width and FU's + separate bundling
- Level 3: integrated scheduling and resource management
  - Template assignment, dispersal rules, …
- Perform experiments to collect various data on all SPEC CPU2000* integer programs

* Other names and brands may be claimed as the property of others

Speedup of Execution Time



* Source: CAS

Some Observations

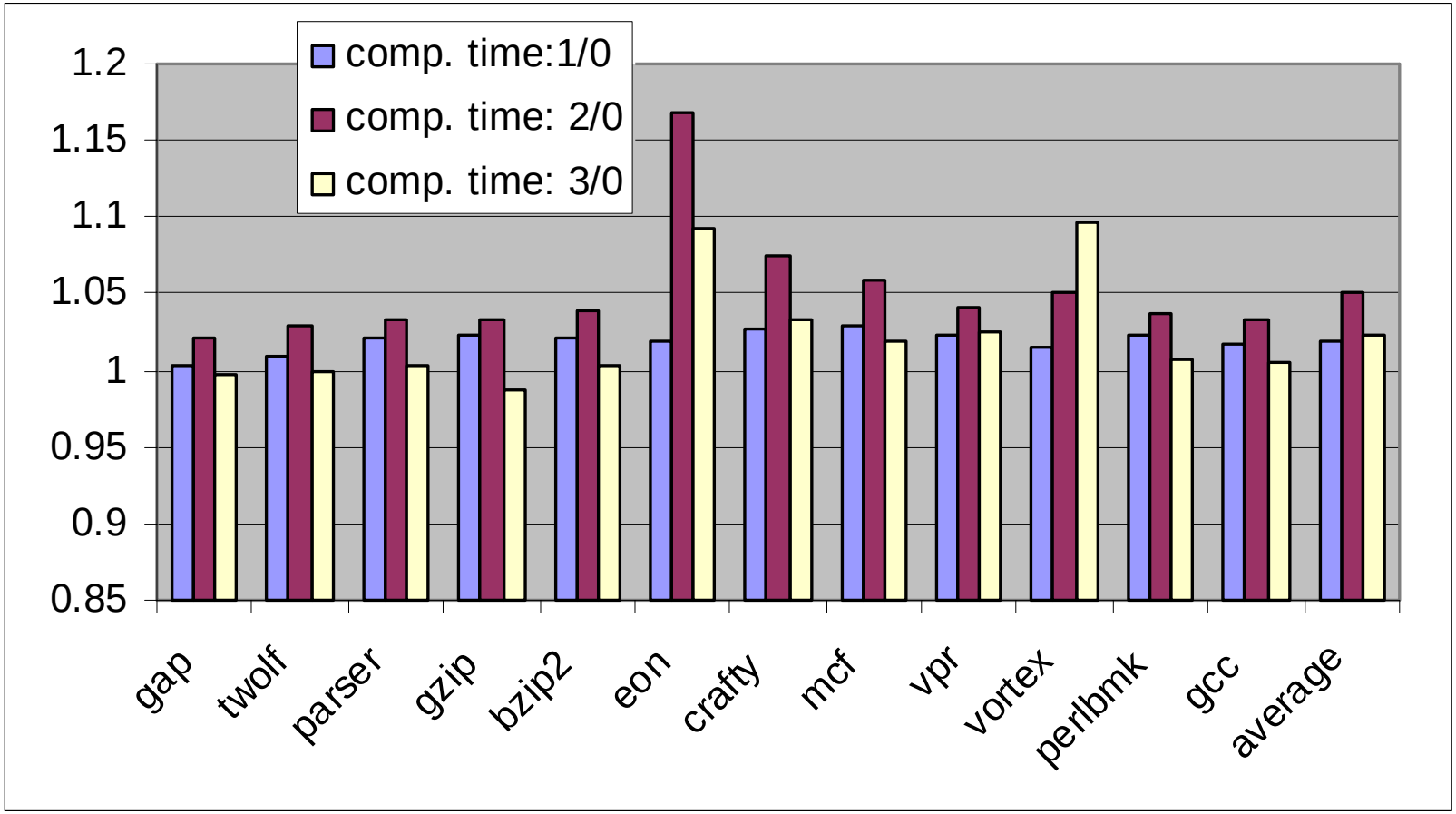
- Little performance change at levels 0-2
- IPC well below machine width
  - The width constraint at level 1 not critical
- FU utilization also low
  - The FU constraints at level 2 not critical either
- Level 3 shows an impressive average 11% improvement
  - IPC improved by 13%
  - NOP ratio, bundles per cycle, instruction count all increase
  - Gain parallelism at the cost of code size
  - Still an overall performance win!

* Performance measurement disclaimer

Performance Disclaimer

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/procs/perf/limits.htm or call (U.S.) 1-800-628-8686 or 1-916-356-3104.

Compilation Time



* Source: CAS

More Observations

- Compilation time includes the time for global and local scheduling and bundling
  - Bundling time is typically well below 10%
- Within 5% differences for all levels
- Important to manage all resource constraints during scheduling
- Our integrated scheduling approach provides
  - Good performance improvements over decoupled approaches
  - Time and space efficiency
- Possibly apply to other architectures:
  - VLIW, DSP, superscalar w/ complex dispersal rules, …

Lessons Learned

- Trivial effort to implement and plug-in alternatives for research study
  - Modularized design, clean interface, …
- Robust compiler infrastructure to run sizeable benchmarks, such as CPU2K and other applications
- Focus effort on studying the key research problem and collecting experimental results for in-depth analysis
  - Minimize the effort on the rest infrastructure issues
- This study “On-the-fly Resource Management during Instruction Scheduling for the EPIC Architecture” is submitted to PLDI 2002.

- Overview of ORC
- New Infrastructure Features
- New IPF Optimizations
- Research Case Study
- **Demo of ORC**
- Release and Future Plans

- Overview of ORC
- New Infrastructure Features
- New IPF Optimizations
- Research Case Study
- Demo of ORC
- **Release and Future Plans**

Release and Future Plan

Agenda

- First release report
- Second release and beyond
- Licensing, distribution and support
- User groups
- Contributing organizations and individuals

First Release Report

State of ORC

- -O0 and –g go through Pro64 path
- Supported optimization levels
  - -O2 (global scalar opt, if-conversion, global scheduling, simple array dep. analysis, GRA, SWP, unrolling, …)
  - -O3 (all –O2 optimizations, loop nest opt., aggressive array dep. analysis, more global scalar opt)
  - Various profiling at code generation time:
    - Edge profiling
    - Value profiling
    - Memory operation profiling and distribution

Performance

- Pro64 standing:
  - About 5% - 10% better than GCC (2.96) at O2 and O3
  - About 10% - 15% slower than Intel IPF Compiler (5.0 and 6.0 Beta) at O2 and O3
  - Seen extreme cases both ways
- ORC standing:
  - Focus is on correctness and infrastructure for this time
  - Performance is better than Pro64 at O2 and O3

* Performance measurement disclaimer

Testing Status

- Well tested at ***–O2*** and ***–O3*** level for general purpose applications.
  - Tests/suits passed:
    - Stanford, Olden, Jpeg, Mesa, ADPCM, CPU2000int…
- Adequately tested at ***–ipa*** (with ***–O2*** and ***–O3***)
  - Pro64 has 4 failures with CPU2000int
  - ORC in par with Pro64 correctness-wise
  - Will fix in second release

Testing Status (cont'd)

- Scientific programs:
  - Adequately tested, but not enough
    - CPU2000fp has 4 known problems at O2 and O3 as we speak, plan to fix ASAP
    - Linpack, Livermore loops passed
    - Perfect club not tried
  - Not major focus for our limited resource

Usage Model

- Invoking ORC:
  - orcc hello.c –o hello { -O2 | -O3 }
  - orc++ hello.cpp –o hello { -O2 | -O3 }
  - orf90 hello.f90 –o hello { -O2 | -O3 }
  - orf90 hello.f -o hello { -O2 | -O3 }
- Skipping ORC:
  - Add option:
    - -ORC:=off
    - Reverts the compiler to be the same as Pro64

Second Release and Beyond

Planned Features

Key concentration for second release

- Performance – *general purpose apps* only
  - O2 / O3 comparable to best Itanium compiler
    - To ensure solid infrastructure framework
  - Sufficient peak performance to make research results trustworthy
- Infrastructure for research remains the key
  - No benchmark tricks
  - No micro-architectural tuning that cannot be translated cleanly into other *u*Arch

Optimizations

- Memory optimizations
  - E.g. data prefetching, various profiling extensions, …
- Better utilization of IPF features
  - E.g. post-increment, predicate aware in various phases, …
- Inter-procedural analysis
  - E.g. aggressive alias analysis, …
- Scheduling/speculation
  - E.g. tune down aggressiveness, partial ready code motion, …

Infrastructure Features

- Multithread support
  - Multithread centric region formation
  - Code motion barriers without disabling optimization
- Annotations of binaries
  - co-design of architecture and compiler
- Interface to simulators
  - Architectural studies
  - Plan to work with Liberty and/or other simulators

Fix Existing Issues

- Existing Pro64 issues
  - IPA not fully functional
  - Register pressure too high for IPF
  - Compiler binary not native built
  - Other tuning issues
    - SWP, inlining,…
  - Fix extreme cases compared with other compilers
- Existing ORC issues
  - Monitor compile time in scheduler
  - Instrumented binary overhead not minimized

Release Timeline

- **Jan 2002**
  - First source/binary release
- **Middle of 2002**
  - Compiler stable with IPA
  - Some benchmark performance comparable to best Itanium compilers
- **End of 2002**
  - Performance goal achieved
  - Infrastructure features

All dates specified are preliminary, for planning purposes only, and are subject to change without notice

Issues Not Planned to Address

- Integration with 3.0 gcc
- FP performance
- Compiler bootstrap itself
- Linux build using ORC
- F90 frontend or library problems

Inviting Research based on ORC

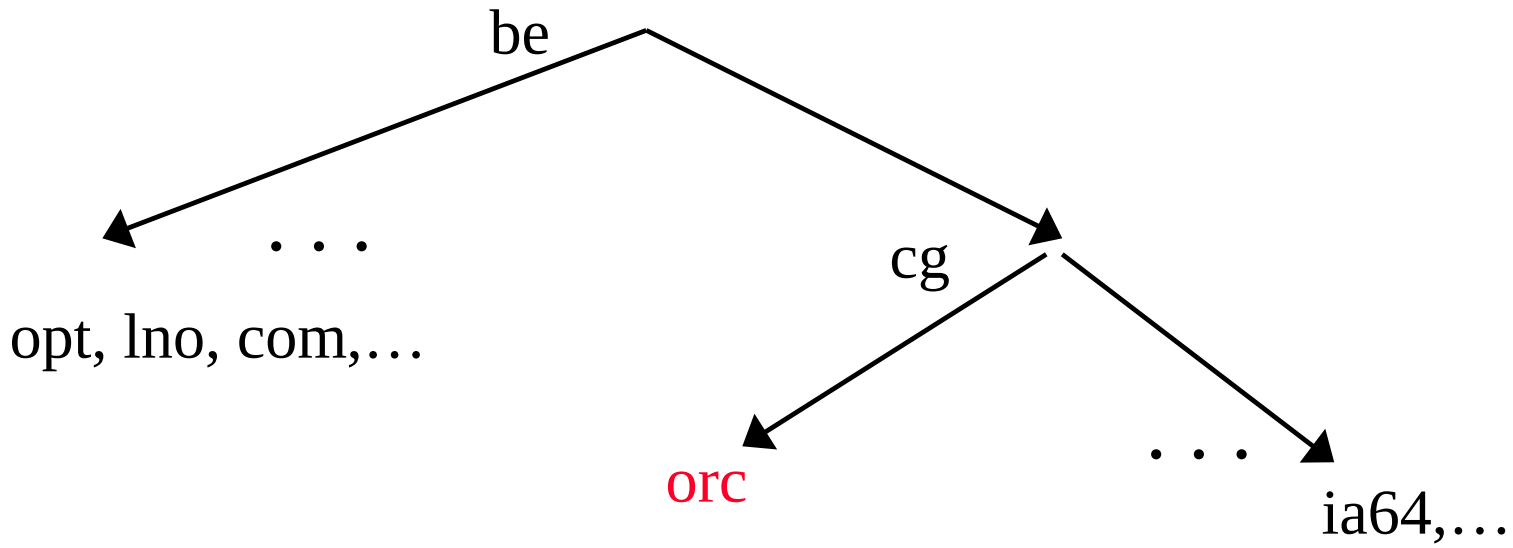
- Performance-driven optimizations
- Thread level parallelism
- Co-design of architecture and compiler
- Retarget to type-safe language (CLI, Java, …)
- Type-safety through aggressive optimizations
- Component level optimizations (.Net, .so, …)
- Higher typed optimizations (user defined types, operators, …)

Inviting Research based on ORC

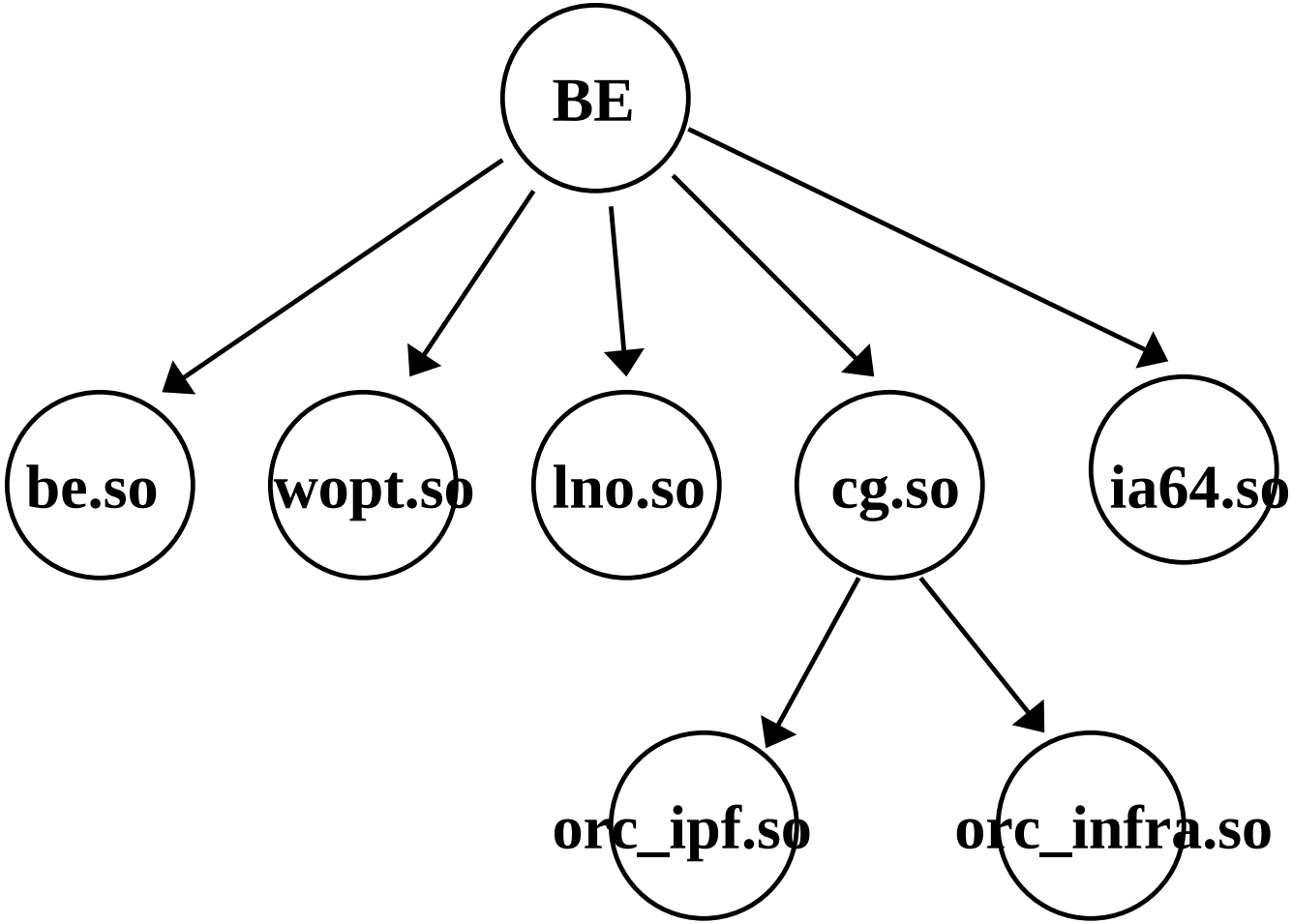
- Optimization for memory hierarchy
- Co-design of static and dynamic compilation
- Program analysis
  - Context sensitive and flow sensitive alias analysis
  - Type hierarchy
  - …
- Inlining/outlining/partial inlining
- Power management
- . . .

Source and Binary Structure

- Source tree structure
  - Same as Pro64 source organization



Backend Binary Components



ORC Testing Infrastructure

- Testing model
  - Developer written tests for specific optimization
    - White box tests for his specific component
  - Developer written tests for integration
    - White box tests for his component working well with other optimizations and components
  - Various open source test suites for black box testing
  - Open source include simple Perl script to run tests at various levels defined by the development team.

ORC Testing Infrastructure

- Simple Perl script to run checkin and/or nightly extensive testing
- Can specify default options and required options
  - Required option to ensure specific opt turned on
  - Default option can be overridden to enable testing by permuting options
- Can specify running of entire test suites at specified options

Development Aids

- Debugging
  - gdb, xxgdb
  - Traces:
    - Before and after optimization IR dump
    - Detail traces of optimization/analysis info and decisions
  - Log:
    - Log of what optimizations performed at what phase
  - Various IR and symbol table dumping tools
  - Elaborate interface to daVinci graph by ORC:
    - Display inside gdb of cfg, regions, BB, …
    - Similar display in files through command-line options

Development Aids

- Debugging (cont'd)
  - Triage tools
    - How to debug file from Pro64 0.13 release:

      "howto-debug-compiler"
    - Major components built as ".so"s
      - Easy to pinpoint component for regressions
    - All optimizations can be turned on/off
      - Easy to pinpoint which opt. that triggers the problem
        - Turn off optimization in reverse phase order

Development Aids

- Debugging (cont'd)
  - Triage tools
    - Automatic tool by ORC
      - Can pinpoint file, function, BB, expression, region or instruction level where bug is manifested.
      - Can pinpoint which component, optimization where bug is manifested

Usage:

triage.pl –iset test –phase speculation –f gzip.mk

iset: input test set

phase: optimization phase to narrow down

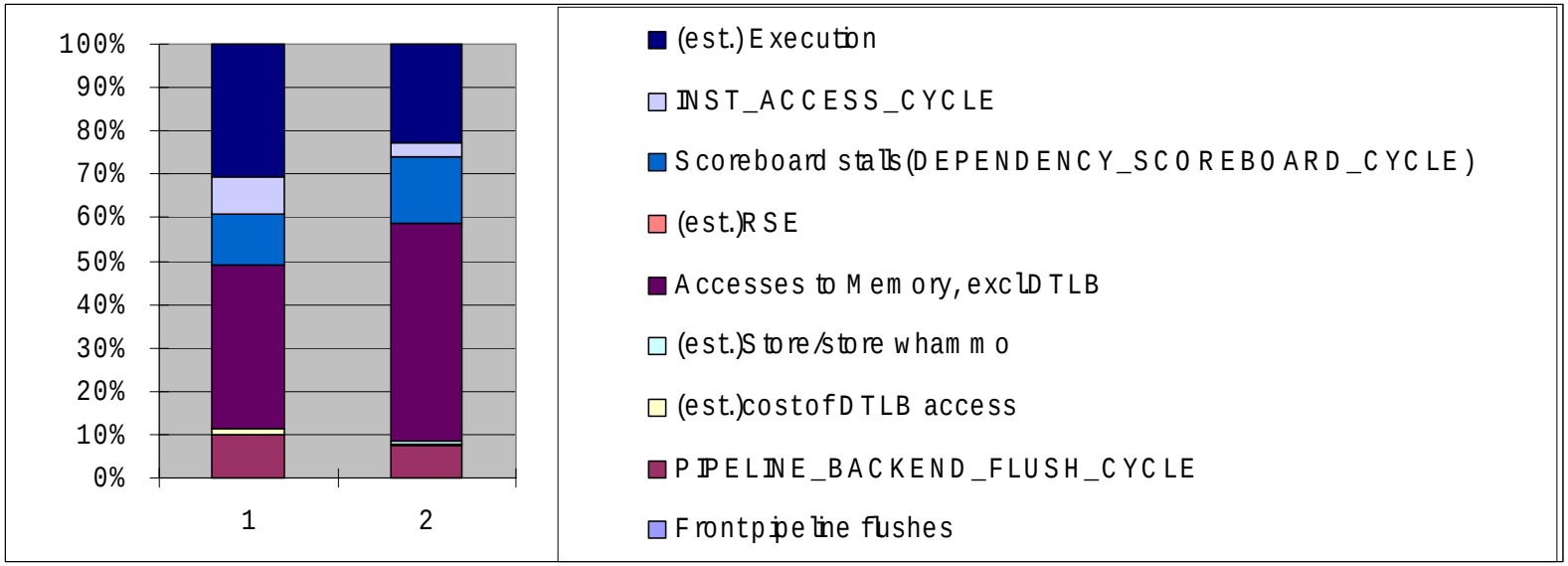
Development Aids

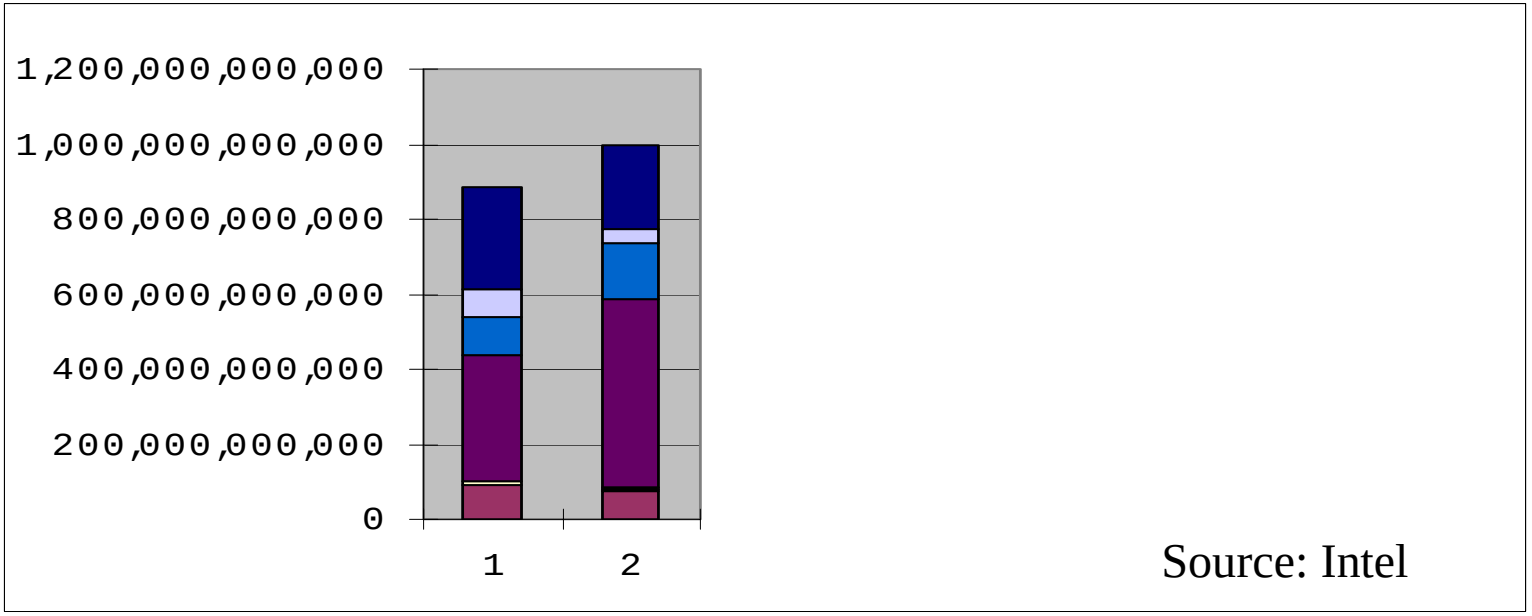
- Expose bugs at compile time philosophy
  - Heavy use of **assertions**
  - Optional **devwarns** for potential problems and temporary workarounds that might have forgotten
  - Heavy use of verification tools and **verifiers** of all sorts
- Styles and coding convention document in first release:
  - Stay close to Pro64 coding style and explanation
  - Include “how to” for memory management, asserts, compile time accounting,…

Development Aids

- Performance analysis and regression tracking
  - PFMON available from
    - `ftp://ftp.hpl.hp.com/pub/linux-ia64/pfmon-0.06.tar.gz`
    - Hardware counters
    - PC sampling
  - Cycle counting tools by ORC
    - Static estimation of cpu execution time
  - Instrumentation/profile tools
    - Dynamic runtime cycle count estimation
    - Memory distribution analyzer

PFMON Data





Source: Intel

Development Aids

- Simulators
  - NUE
    - IPF functional simulator from website:
      http://www.software.hp.com/ia64linux
  - Liberty
    David August, Princeton
- Open for interface to other simulators
  - Cache simulator?
  - Simple Scalar?
  - Others (any takers?)

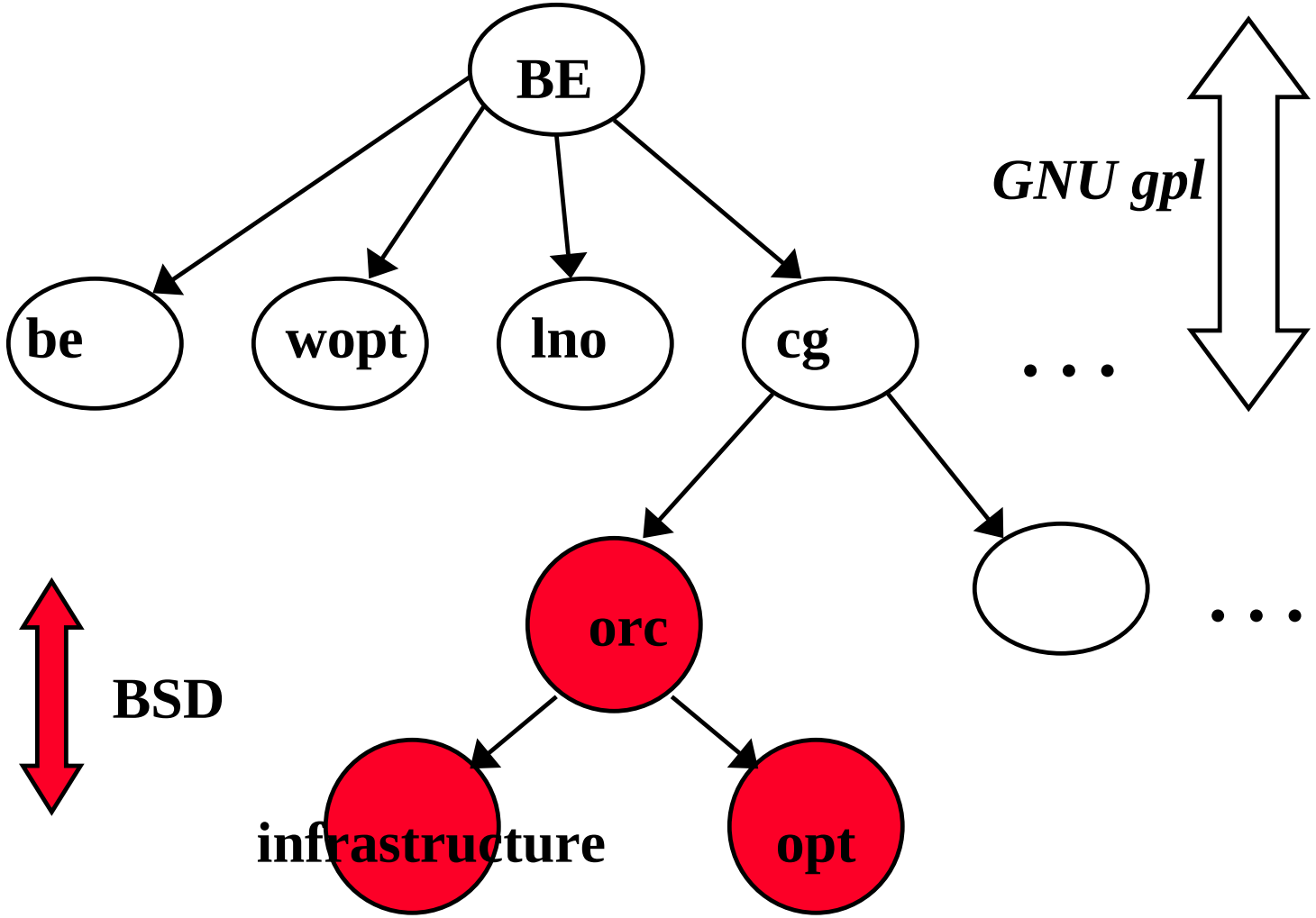
Licensing, distribution and support

Licensing, Distribution and Suppport

- The compiler will be distributed in the web-site
  http://sourceforge.net/projects/ipf-orc

- Latest information and update are placed there also
- Distribution includes:
  - Binary
  - Source Code
  - Test and triage infrastructure including scripts
  - Documents and various tools

Licensing



Licensing

- BSD license url

http://www.opensource.org/licenses/bsd-license.html

Distribution

- Download:

http://prdownloads.sourceforge.net/ipf-orc/orc-1.0.0.tgz

  - Compiler binary are IA32 images (cross built)
  - Will run slow on an Itanium machine

Distribution and Installation

- To install on an Itanium machine (Redhat 7.1):

  %su

  ./install.sh
- To install under nue:

  %su

  #nue

  #./install.sh

Distribution and Installation

- To do on an IA32 machine
  - cross build, easiest is to use ***nue***
  - Be careful about library compatibility issues
    - ***nue*** is not 7.1 and up compatible
  - Don't pick up IA32 includes, libraries and .so's
- Cross compile on an IA32 machine under ***nue***
  - IA32 side:
    - orcc –c file.c –o file.o
      - Produce object file
    - ftp IA64
      - Transfer object files to itanium machine
  - IA64 side:
    - orcc file.o –o exec
      - Produce binary with right libraries
    - ./exec

Support

- For issues with installation, compiler usage, use mail alias:

  ipf-orc-support@lists.sourceforge.net

- Please sign on to

  ipf-orc-support@lists.sourceforge

Support

- Bugs can be reported via ***Sourceforge*** in the website
  - Select ***support requests*** under ***tracker***
  - Choose ***submit new***
- Will fix ***ORC*** specific problems
- Cannot promise to fix ***Pro64™*** problems

Reporting Problems

- You can help resolving problems quickly
  - Give precise characterization of problem/symptom
  - Give detailed description of
    - Compile and optimization options
    - Command to execute application if runtime error
  - Include
    - fully preprocessed (**cpp**) source to reproduce problem
    - Other input files such as data files needed to run
  - Reduce your test case to as small as possible
  - Use triage tool and/or follow how-to-find-problems to narrow down possible culprit

Accepting Contributions

- Contribute to the source code
  - No clear policy on how to accept changes yet
  - Welcome suggestions
  - Will look at how Open64 user group operates
  - Will post policy on websites when decided

User Groups

User Groups

ORC user group

ipf-orc-support@lists.sourceforge

Pro64™ user group

open64-devel-support@lists.sourceforge

Open64 User Forum

Steering Committee:

Guang R. Gao

Jose Nelson Amaral

Date/Time: **tonight, 8:00p.m. – 10:00 p.m.**

Place: Marriott Hotel

Contributing Organizations and Individuals

This project is a joint development effort between

Microprocessor Research Labs,

Intel Research Labs

&&

Institute of Computing Technology,

Chinese Academy of Sciences

Contributing individuals
ICT

- Xiqian Dong
- Ge Gan
- Ruiqi Lian
- Lixia Liu
- Yang Liu
- Fei Long
- Fang Lu
- Yunzhao Lu
- Shuxin Yang
- Chen Fu *
- Ren Li *
- Zhanglin Liu *
- Chengyong Wu
- Lizhe Wang *
- Shukang Zhou *
- Kexin Zhang
- Zhaoqing Zhang

* *No longer at ICT*

Contributing individuals
Intel

Sun Chan
Kaiyu Chen
BuiQi Cheng
Jesse Fang
Roy Ju
Tony Tuo
QingYu Zhao
DongYuen Chen
William Chen *
ZhaoHui Du
Tao Huang
Tin-Fook Ngai
YouFeng Wu
Dagen Wong *

* *No longer at Intel*

Thanks to Pro64 developers

Murthy Chandrasekhar
Fred Chow
Robert Cox
Peter Dahl
Alban Douillet
Ken Lesniak
Jin Lin
Mike Murphy
Ross Towle
Lilian Leung
Raymond Lo
ShinMing Liu
Wilson Ho
Zhao Peng
HongBo Rong
David Stephenson
Peng Tu