IA-64 Architecture and Compiler Technology

Allan Knies at Intel/IPD
Jesse Fang at Intel/MRL
Wei Li at Intel/MSL

Compiler Technology for IA-64 (Part II)

Jesse Fang
Microprocessor Research Lab

Overview

- Optimizations
  - Profiling-based optimizations
  - Interprocedural optimizations
- Predication
  - If-conversion
    - ✔ Uses regular, unconditional and parallel compares
- Global scheduling
  - Instruction scheduling cross basic block boundaries
    - ✔ Uses control and data speculation, predication, post-increments, multiway branches
- Global register allocation
  - Allocate registers for predicate code
    - ✔ Uses register stack, ALAT associativity
- Driven by information provided by machine model

1. Profile-based Optimizations

- New IA-64 features enable aggressive optimizations and scheduling.

- The highest performance can be achieved with the most accurate profiling information.
  - Characterize the execution behavior of the program
  - Use the profile information to guide optimizations

Profile Generation

- Static profiling
  - Program-based heuristics (e.g. loop branch, pointer, call, opcode, loop exit, return, store, loop header, guard, error) for branch probabilities.
  - Estimation of execution frequencies on basic blocks and edges from probabilities and control-flow graph.

- Dynamic profiling
  - Program instrumentation (compile once with counters inserted).
  - Run the instrumented program with a sample input set.
  - Performance monitor registers in IA-64

Using Profile

- Profile can be used in many parts of the IA-64 compiler
- Branch probability guide (examples)
  - Predication region
  - Scheduling region
  - Speculation
  - Code placement
- Execution frequency guide (examples)
  - Procedure inlining/partial inlining
  - Loop optimizations
  - Software pipelining
  - Register allocation
- D-Cache missing guide (examples)
  - Data speculation
  - Data prefetch

Interprocedural Analysis and Optimization

- Use of Alias Analysis
  - Indirect call conversion
  - Better disambiguation
- Use of Mod/ref Analysis
  - Fewer *kills* due to unknown modification (PRE) or reference (PDSE)
- Constant propagation
  - Makes constant parameters and globals explicit.
  - Enables cloning.
  - Remove unnecessary conditional code.
  - Improves data dependence analysis.
- Inlining
  - Increase scope of optimization and instruction scheduling
  - Partial inlining

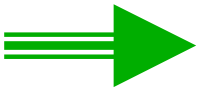
Procedure Inlining

- Benefits:
  - Larger code region to schedule
  - Larger code region for optimizations (such as loop transformations)
  - Interprocedural information for a specific call site.
- Cost:
  - Code size increases
- Selective integration (partial inlining and cloning) to reduce code size growth.

Inlining

- Inlining a function body into a call site.

```
void func1()
{
    int i;
    for (i=0;…)
            func2(i);
}
void func2(int x)
{
    a[x] = 1.0;
}
```

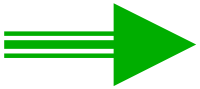


```
void func1()
{
    int i;
    for (i=0;…)
            a[i] = 1.0
}
```

Partial Inlining

- Copy the *hot* portion of a function into a call site.
- Remainder becomes a *splinter* function.

```
void foo(x)
{
  if (P(x))
    a hot region here
  else
    a large cold region
    here.
}

void main()
{
  call foo(y)
}
```

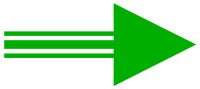


```
void foo_cold(x)
{
  a large cold region
    here.
}

void main()
{
  if (P(y))
    a hot region here
  else
    call foo_cold(y)
}
```

Cloning

- Specializing a function to a specific class of call sites.

```
void func1()
{
    func2(0, n);
    func2(0, m);
    func2(i, m);
}
void func2(int i, int j)
{
    if (i == 0)
      // do something
    else
      // do something else
}
```



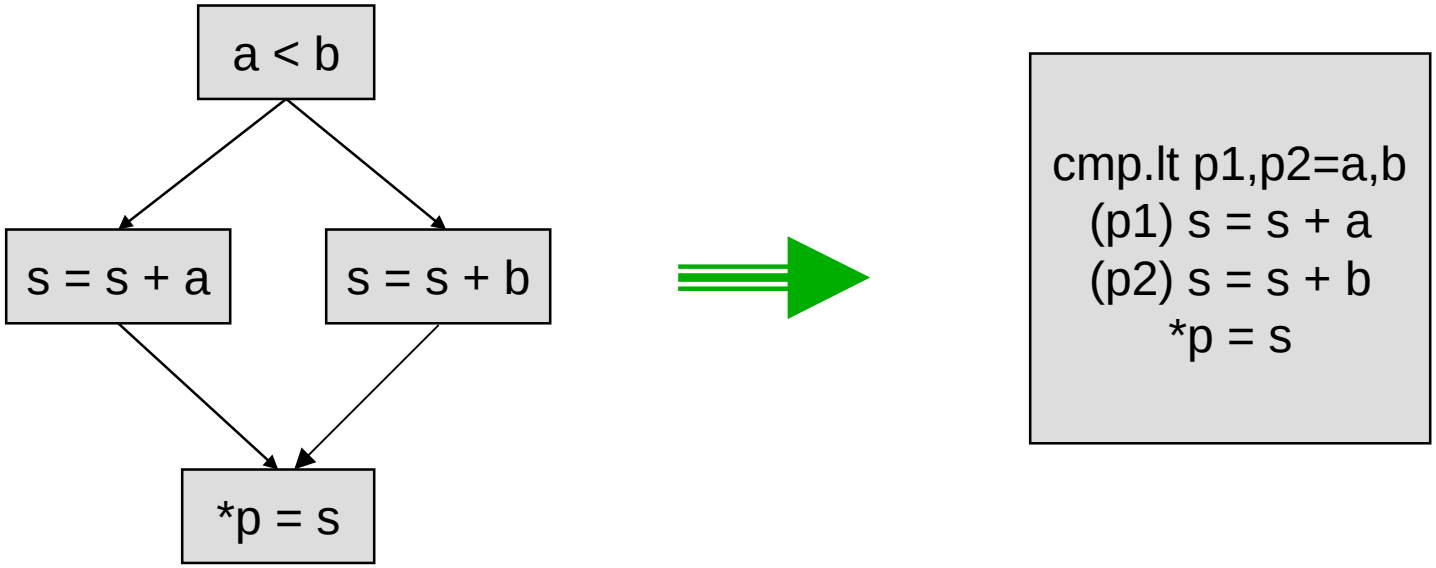
```
void func1()
{
    func2_0(n);
    func2_0(m);
    func2(i, m);
}

void func2_0(int j)
{
    // do something
}
```

(2) Predication

- When branch misprediction rate is high, it is better to predicate
- Predication creates more ILP
- Predication has the potential cost of increasing the critical path length
- Techniques
  - If-conversion
  - Parallel compare to reduce control height

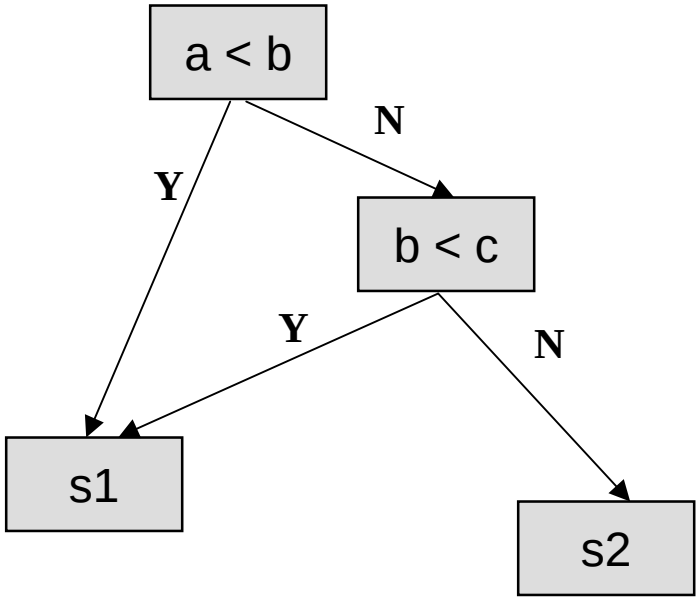
If-conversion for Predication

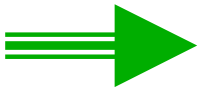
- Identifying region of basic blocks based on resource requirement and profitability (branch miss rate, miss cost, and parallelism)
- Result: a single predicated basic block



Reducing Control Height
with parallel compares

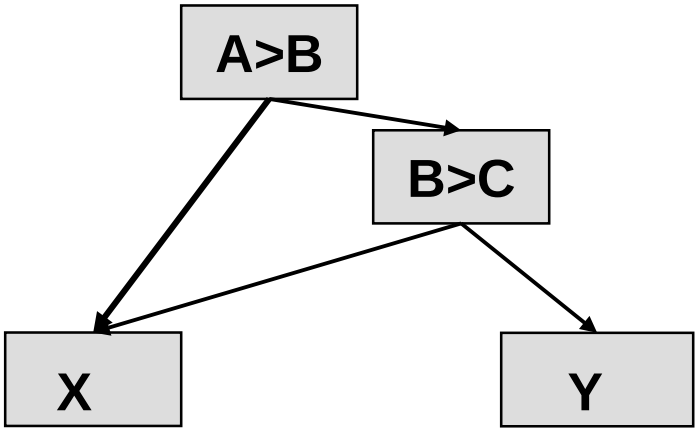
- Convert nested if's into a single predicate
- Result: shorter control path by reducing the number of branches





```
p1=0
cmp.lt.or p1,p0=a,b
cmp.lt.or p1,p0=b,c
(p1) br s1
s2
```

Multiway Branch Example



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

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

If (a>b && b>c)

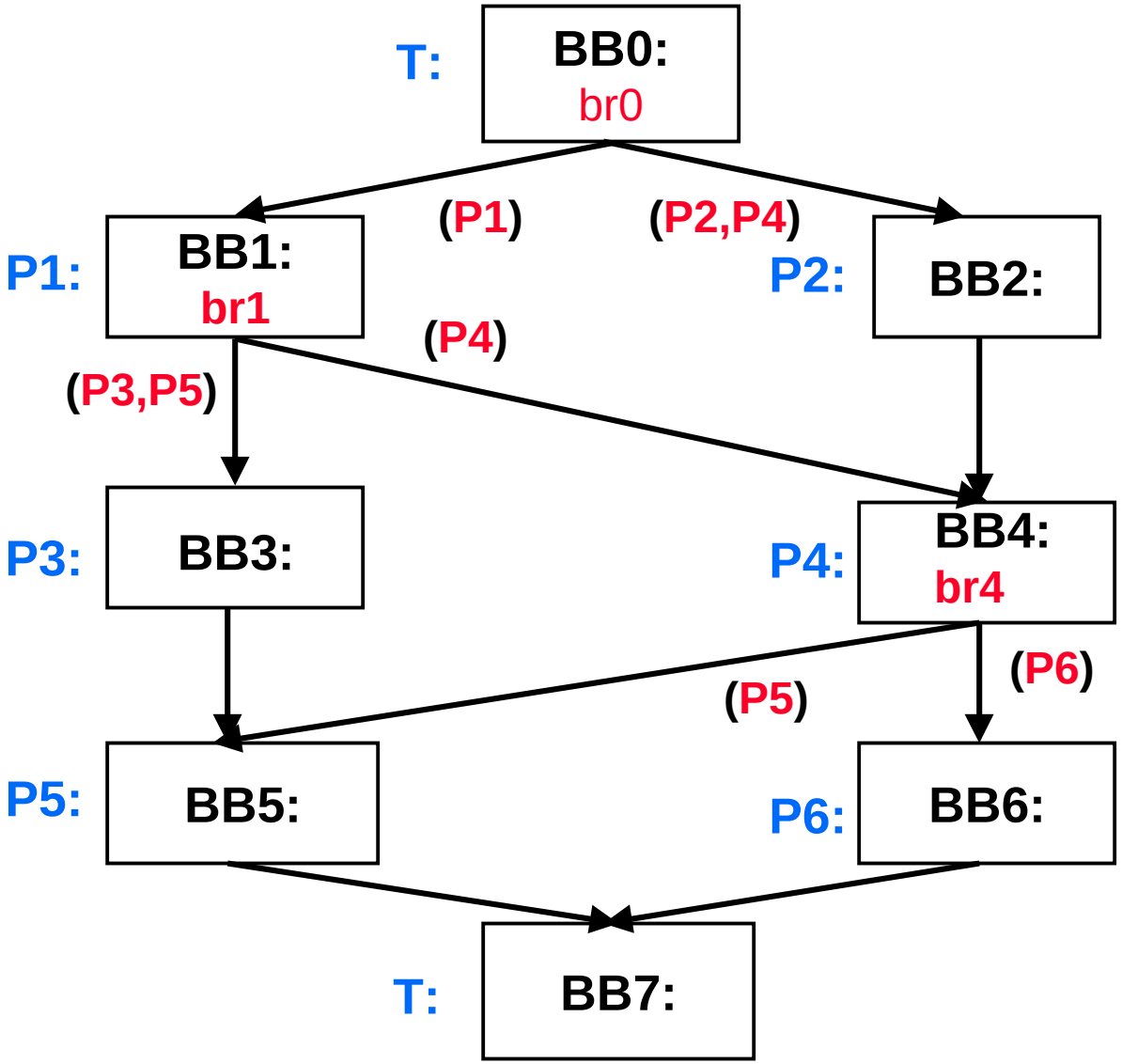
then Y

else X

If-Conversion Algorithm

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

Example of If-Conversion

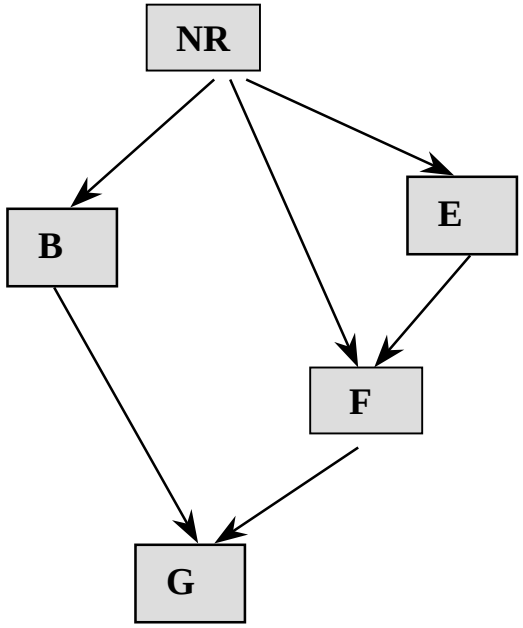


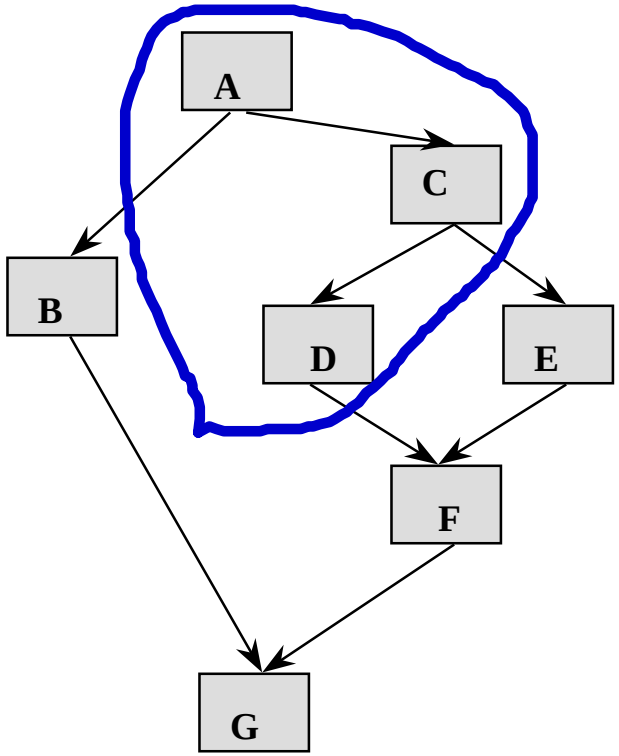
(3) Global Code Scheduling

- Objective
  - Increase parallelism
  - Remove unnecessary dependencies
  - Fully use machine width
- Needs
  - Accurate machine model
- Uses architectural features
  - Large number of registers
  - Control and data speculation, checks and recovery code
  - Multiway branches

Region Formation

- Scheduling Regions are acyclic

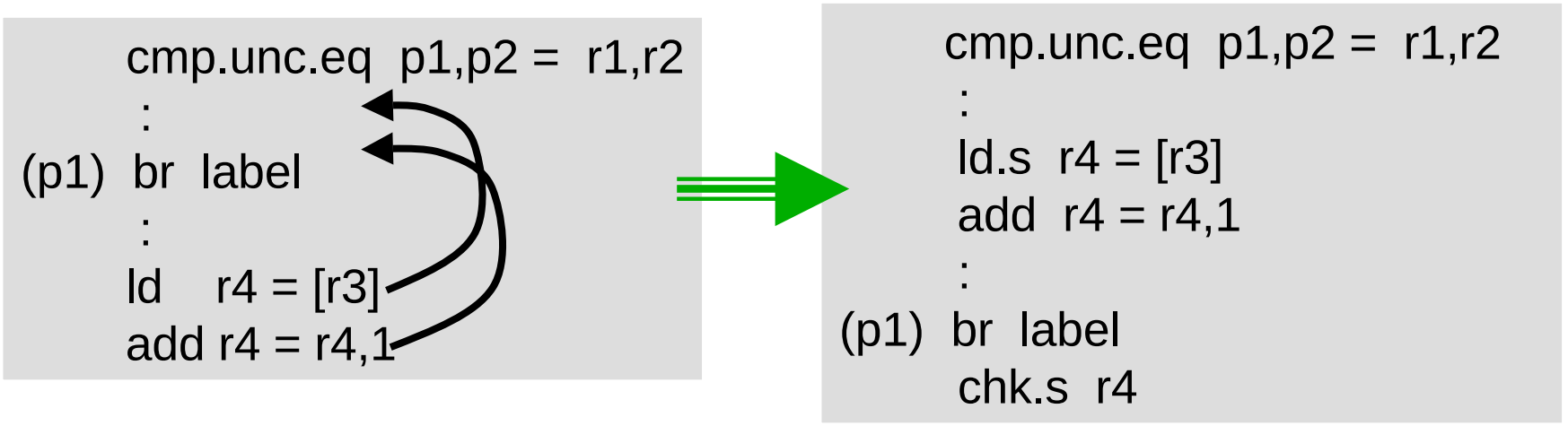




{A,C,D} a nested region as NR

Speculative Upward Code Movement

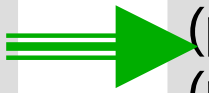
- Speculate both the load and the use
- Result: efficient use of machine resources



Predicated Upward Code Movement

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

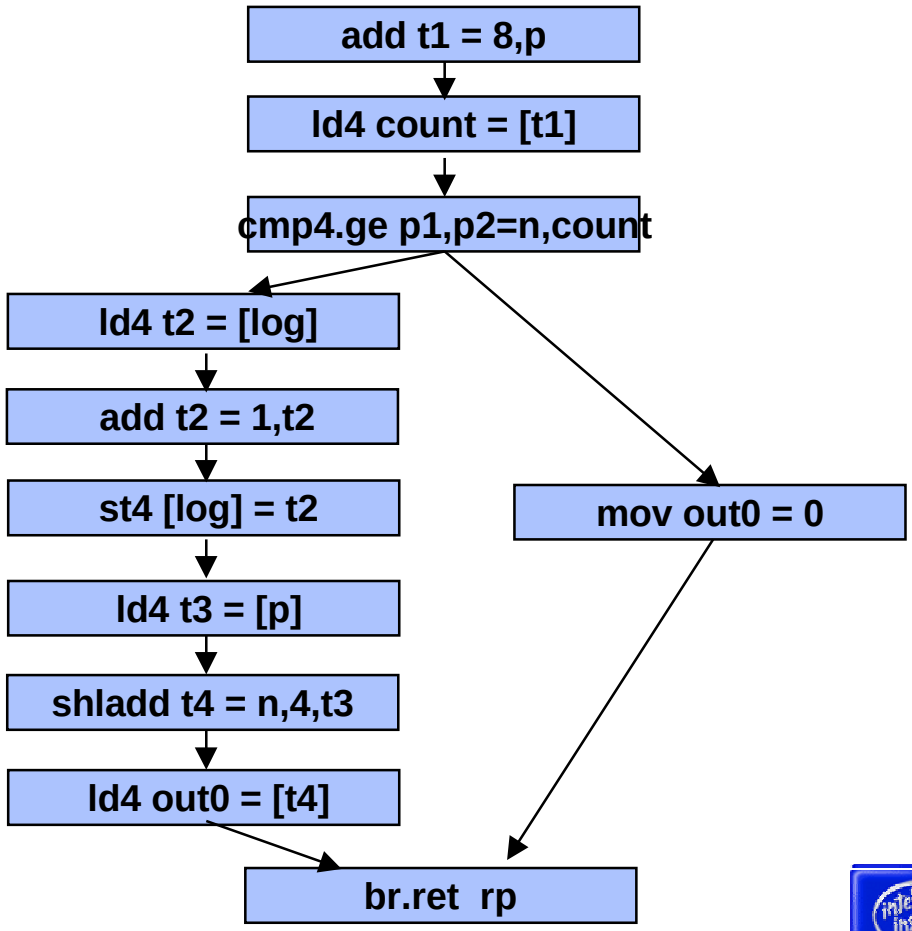
```
      cmp.unc.eq  p1,p2 =  r1,r2
       :
(p1)  br  label
       :
      ld    r4 = [r3]
      add r4 = r4,1
```



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

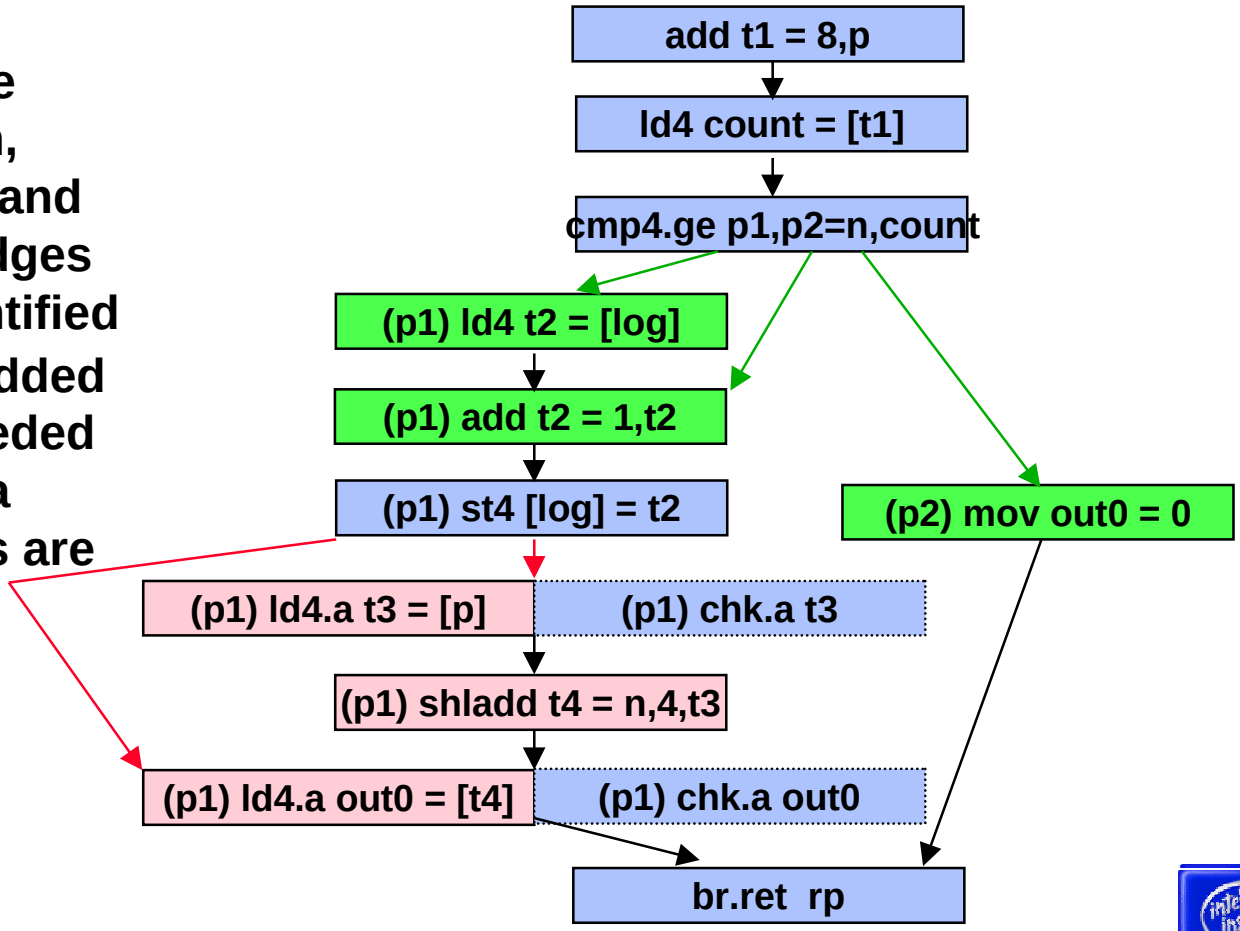
Example of Instruction Scheduling
- Control Flow Graph

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

If( n < p->count ) {
      (*log)++;
      return p->x[n];
} else {
      return 0;
}
```

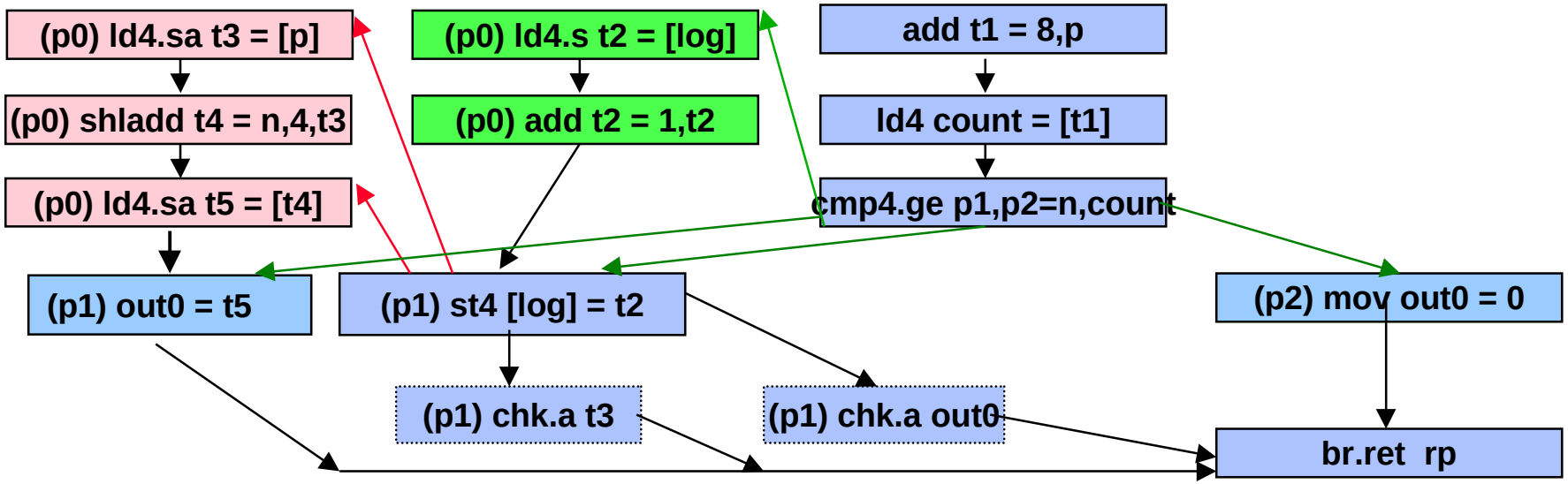


Example of Instruction Scheduling
- with predication and possible speculation

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

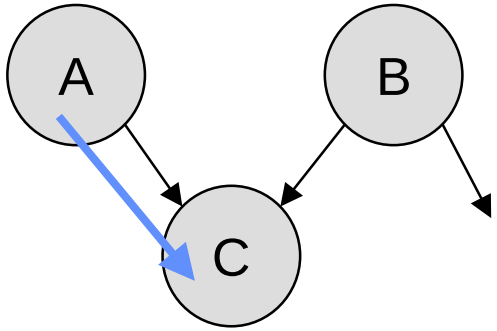


Example of Instruction Scheduling
- ready for scheduling

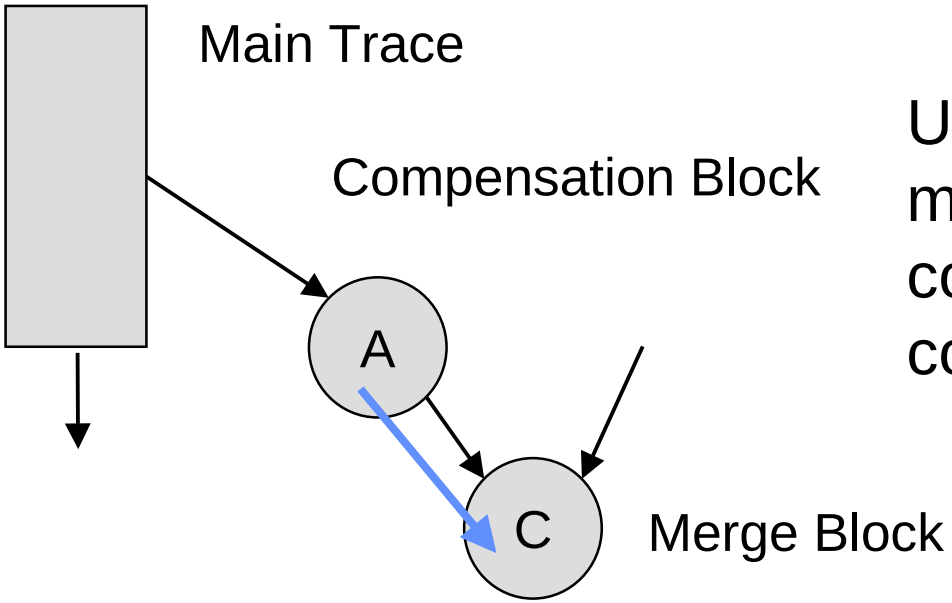


- Speculative edges may be violated. Here the graph is re-drawn to show the enhanced parallelism
- Note that the speculation of both writes to the out0 register would require insertion of a copy. The scheduler must consider this in its scheduling
- Nodes with sufficient slack (e.g. writes to out0) will not be speculated

Downward Code Movement



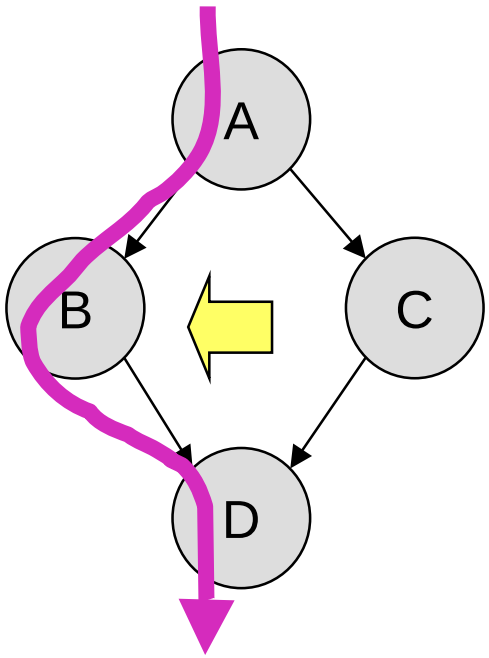
Predication enables downward code movement from A to C without compensation code in B



Use predication to merge sparse code in compensation block with code in merge block

Code Motion Tradeoffs

Downward Code Motion



Upward Code Motion

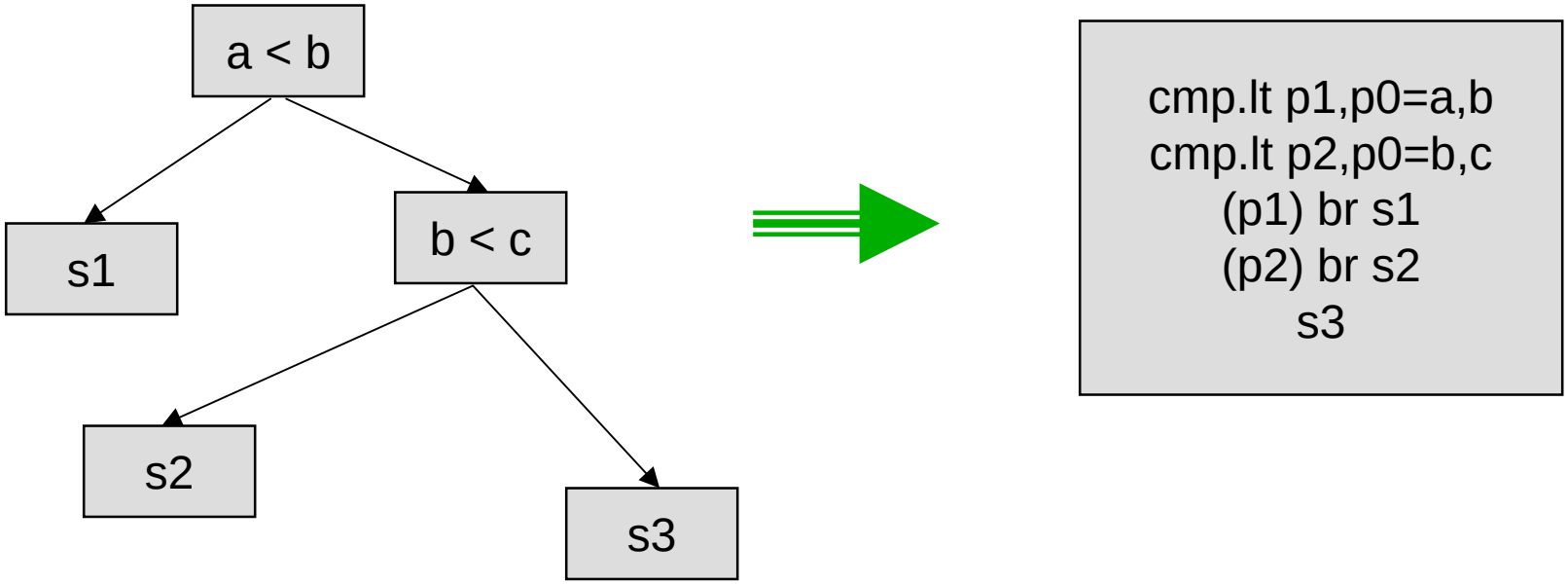
Slots available in hot path

Predication can pull instructions from lower weight path

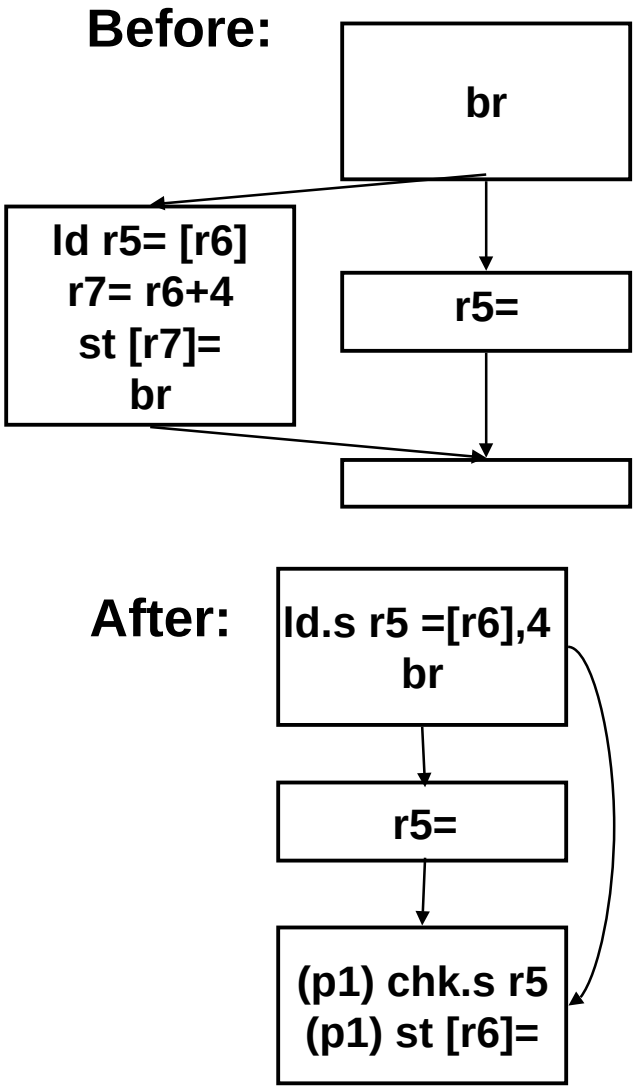
Scheduler can move instructions from above and below

Multiple Branches in Single Cycle

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



Global Code Scheduling Design



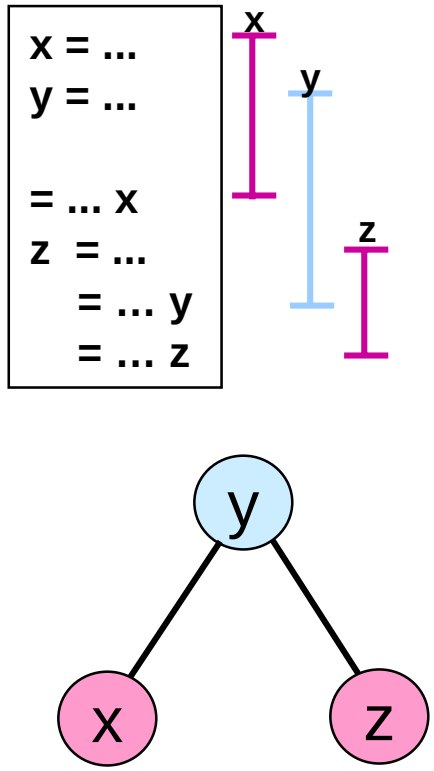
- Move code above branches using predication and/or speculation
- Move code below branches using predication
- Move loads and uses across stores that are unlikely to interfere and generates checks and recovery
- Combine memory references and address increments to generate post-increments if r6 and r7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

(4) Global Register Allocation

- Objective
  - Eliminate memory references. Minimize register spill and copying after load-store elimination (virtual registers)
- Incorporating architectural features
  - Large number of registers
  - Predication
  - Data speculation

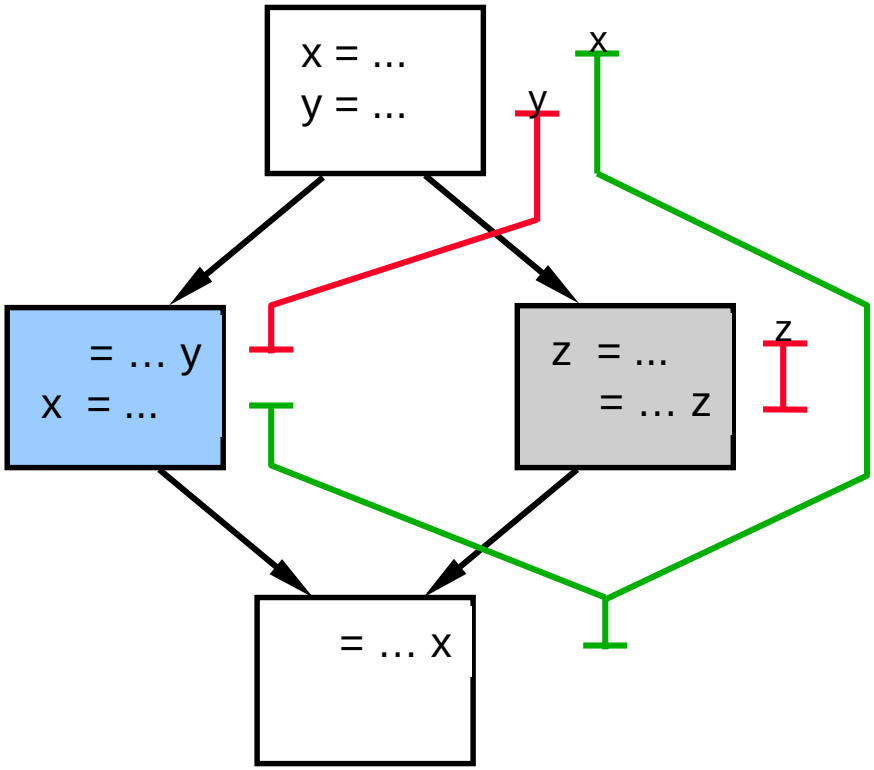
Register Allocation Example

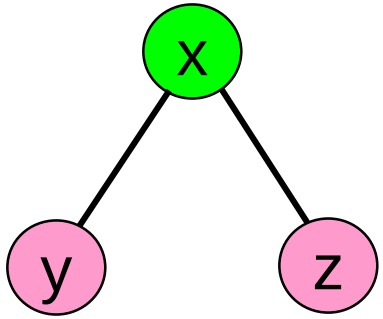
- Modeled as a graph-coloring problem
  - Nodes in the graph represent live ranges of variables
  - Edges represent a temporal overlap of the live ranges
  - Nodes sharing an edge must be assigned different colors (registers)



Register Allocation Example

With Control Flow

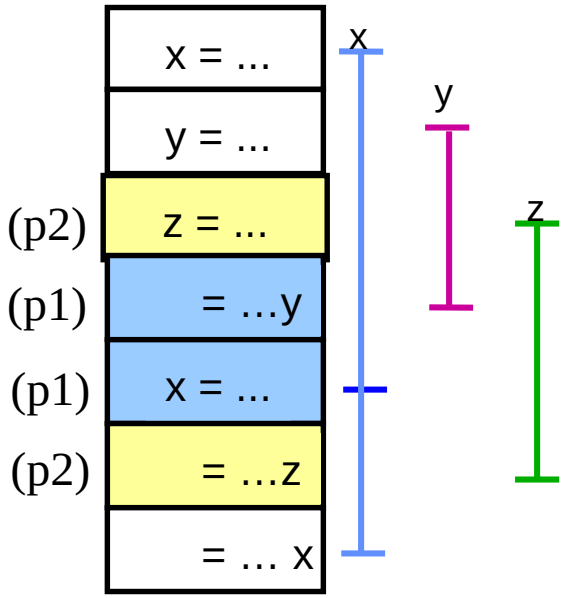


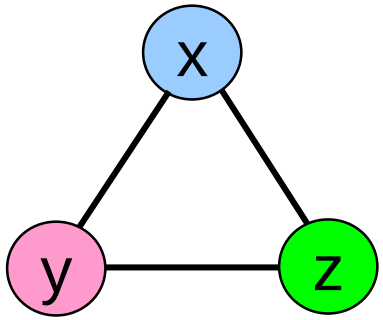


Requires Two Colors

Register Allocation Example

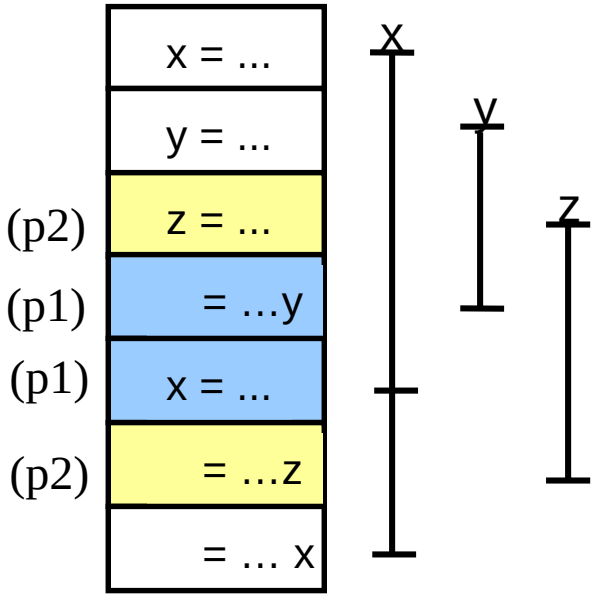
With Predication

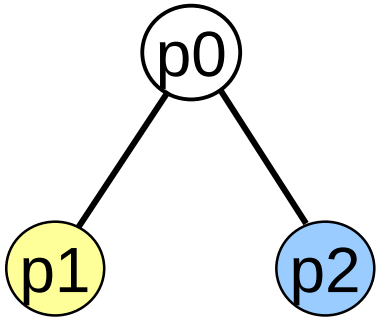




Now Requires Three Colors

Predicate Analysis for the Example

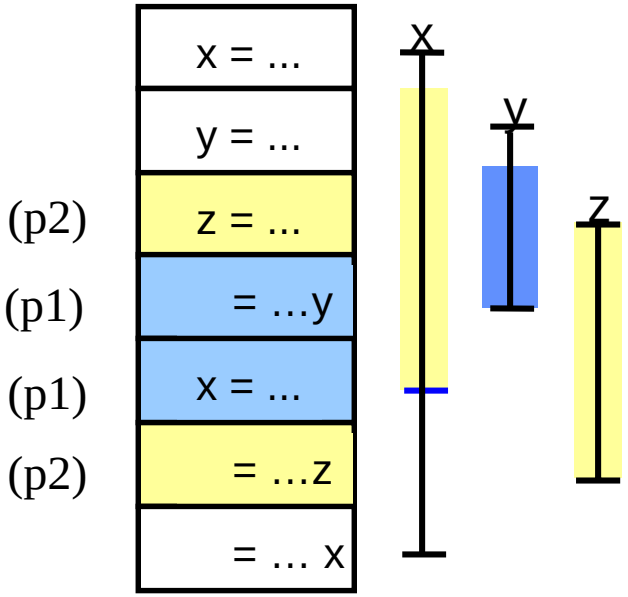


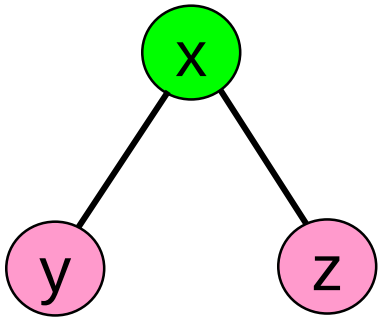


p1 and p2 are disjoint
If p1 is TRUE, p2 is false
and vice versa

Register Allocation Example

With Predicate Analysis



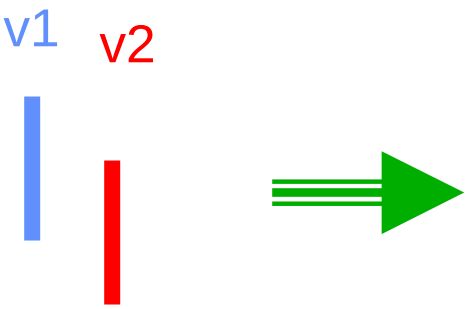


Now Back to Two Colors

Effect of Predicate-Aware Register Allocation

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

```
(p1) v1 = 10
(p2) v2 = 20 ;;
(p1) st4 [v10]= v1
(p2) v11 = v2 + 1 ;;
```



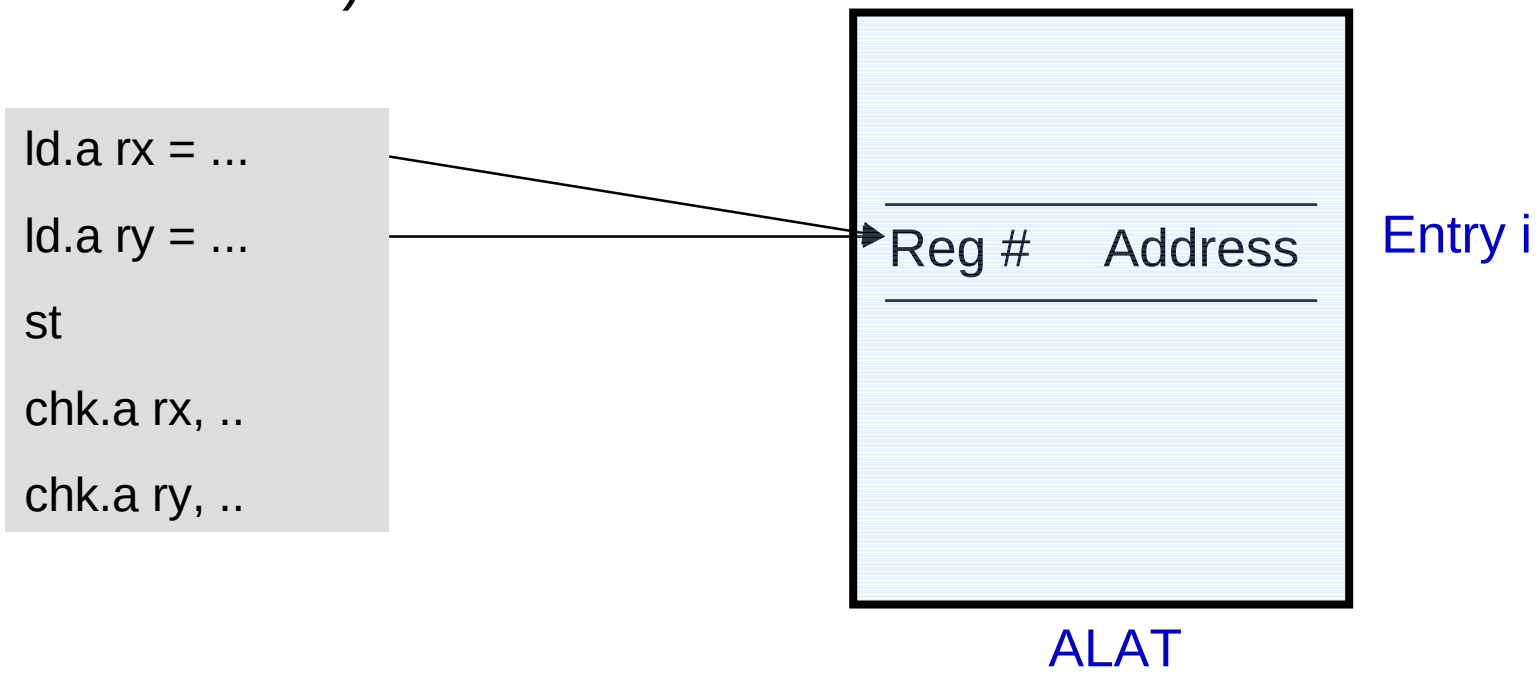

```
(p1) r32 = 10
(p2) r32 = 20 ;;
(p1) st4 [r33]= r32
(p2) r34 = r32 + 1 ;;
```

overlapped live ranges

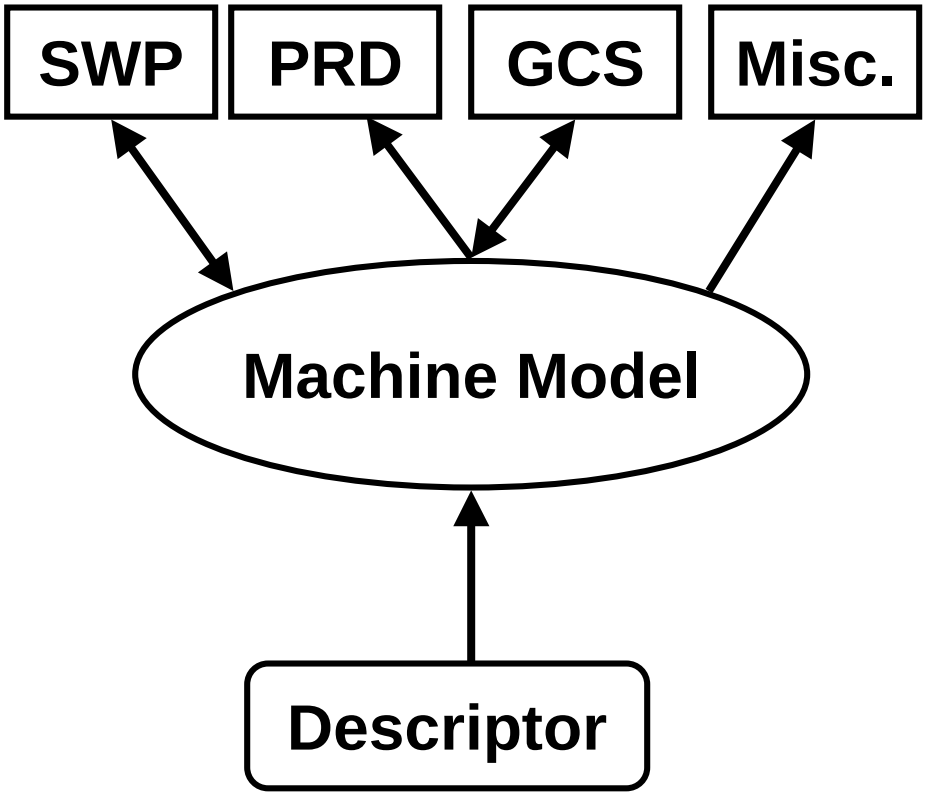
same register for v1 and v2

Minimize ALAT Conflict

• Assign registers to the live ranges of the advanced loads to eliminate possible conflict in ALAT (Advanced Load Address Table)



(5) Modeling The Machine



- Objective: Provide micro-architecture information to the rest of the compiler
  - Machine characteristics read from microarchitecture description file, which is used by simulator and other tools.

Modeling The Machine

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

Summary

- Compiler is critical for IA-64 performance
  - backend should take full advantages of IA-64 architectural features to generate optimal code
- Optimizations
  - Profiling
  - Interprocedural optimizations
- If-conversion
  - predication with various compares
- Global scheduling cross basic block boundaries
  - control and data speculation, post-increments, multiway branches,
- Global register allocation
  - register stack, ALAT associativity
- Driven by information provided by machine model

Dynamic Optimization Technology on IA-64

Jesse Fang
Microprocessor Research lab

IA-64 Architecture Advantages for Java

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

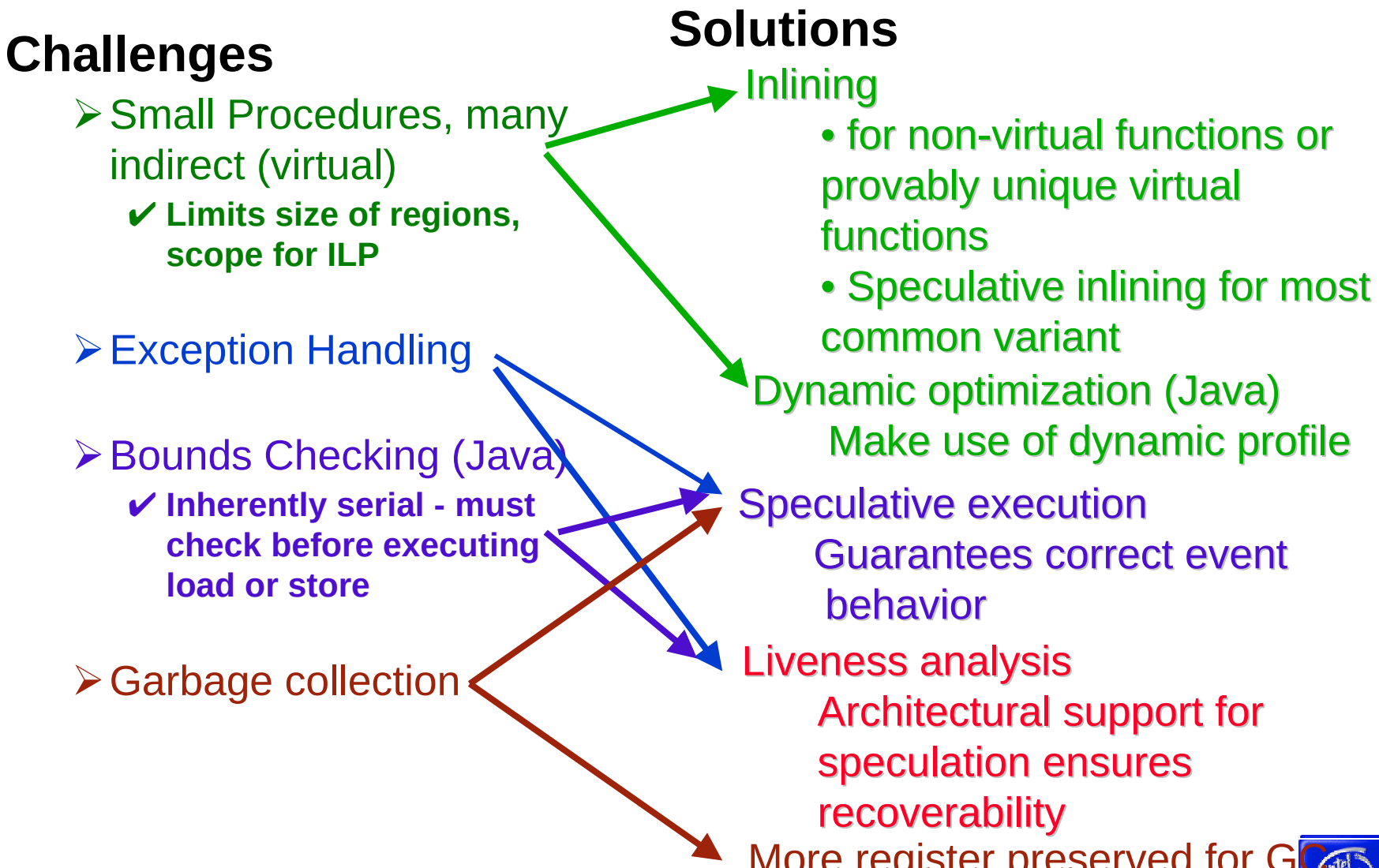
IA-64 Java Software Convention

- JIT generated code follows IA-64 software conventions including
  - parameter passing
  - memory stack management
  - general register usage guidelines
- JVM may reserve extra registers within IA-64 register usage guidelines for memory and thread management, which should be allocated
  - from r4 up for preserved registers
  - from r14 up for scratch registers
  - JVM cannot assume that native code abides by the additional Java register usage restrictions

Java JVM/JIT Design on IA-64

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

Object-Oriented Code on IA-64



Dynamic Optimizations for C/C++

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

Summary

- Java on IA-64
  - IA-64 advantages for Java
  - IA-64 Java software convention
  - JVM/JIT design on IA-64
- Object-Oriented code on IA-64
  - virtual procedure calls
  - precise exception handle
  - garbage collection
- Take advantages of IA-64 architecture for dynamic optimizations