

15-740/18-740
Computer Architecture
Lecture 26: Predication and DAE

Prof. Onur Mutlu
Carnegie Mellon University

Announcements

- Project Poster Session
 - December 10
 - NSH Atrium
 - 2:30-6:30pm
- Project Report Due
 - December 12
 - The report should be like a good conference paper
- Focus on Projects
 - All group members should contribute
 - Use the milestone feedback from the TAs

Final Project Report and Logistics

- **Follow the guidelines in project handout**
 - **We will provide the Latex format**
- Good papers should be similar to the best conference papers you have been reading throughout the semester
- **Submit all code, documentation, supporting documents and data**
 - Provide instructions as to how to compile and use your code
 - This will determine part of your grade
- This is the single most important part of the project

Today

- Finish up Control Flow
 - Wish Branches
 - Dynamic Predicated Execution
 - Diverge Merge Processor
 - Multipath Execution
 - Dual-path Execution
 - Branch Confidence Estimation
 - Open Research Issues
- Alternative approaches to concurrency
 - SIMD/MIMD
 - Decoupled Access/Execute
 - VLIW
 - Vector Processors and Array Processors
 - Data Flow

Readings

- Recommended:
 - Kim et al., “Wish Branches: Enabling Adaptive and Aggressive Predicated Execution,” IEEE Micro Top Picks, Jan/Feb 2006.
 - Kim et al., “Diverge-Merge Processor: Generalized and Energy-Efficient Dynamic Predication,” IEEE Micro Top Picks, Jan/Feb 2007.

Approaches to Conditional Branch Handling

- Branch prediction
 - Static
 - Dynamic

- Eliminating branches
 - I. Predicated execution
 - Static
 - Dynamic
 - HW/SW Cooperative
 - II. Predicate combining (and condition registers)

- Multi-path execution
- Delayed branching (branch delay slot)
- Fine-grained multithreading

Approaches to Conditional Branch Handling

- Branch prediction
 - Static
 - Dynamic

- Eliminating branches
 - I. Predicated execution
 - Static
 - Dynamic
 - HW/SW Cooperative
 - II. Predicate combining (and condition registers)

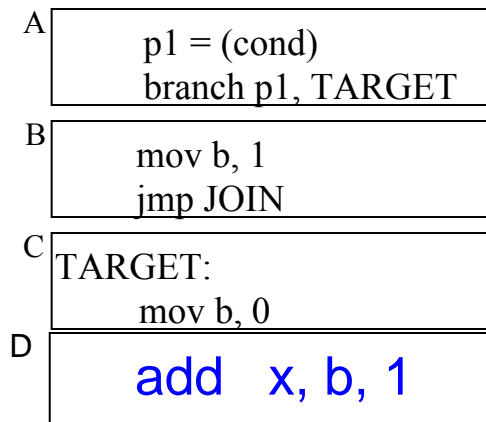
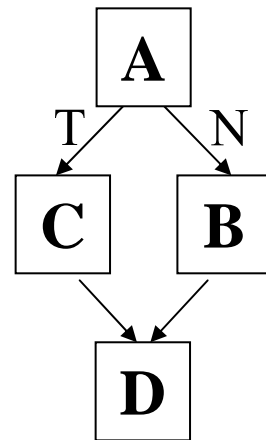
- Multi-path execution
- Delayed branching (branch delay slot)
- Fine-grained multithreading

Predication (Predicated Execution)

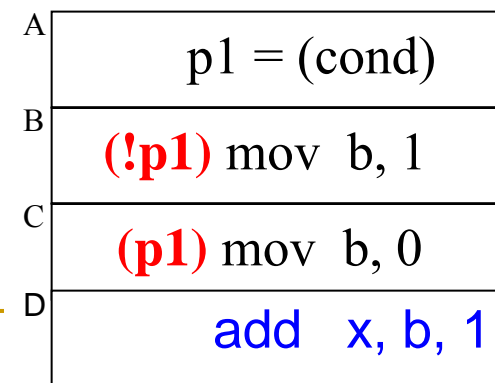
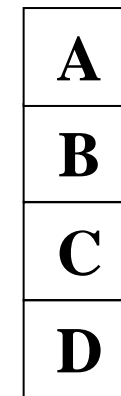
- Idea: Compiler converts control dependency into a data dependency → branch is eliminated
 - Each instruction has a predicate bit set based on the predicate computation
 - Only instructions with TRUE predicates are committed (others turned into NOPs)

(normal branch code)

```
if (cond) {  
    b = 0;  
}  
else {  
    b = 1;  
}
```



(predicated code)

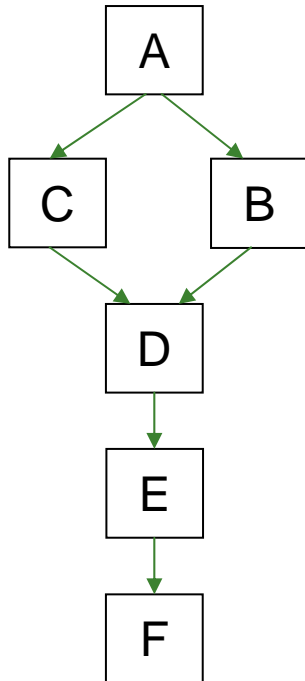


Conditional Move Operations

- Very limited form of predicated execution
- CMOV R1 ← R2
 - $R1 = (\text{ConditionCode} == \text{true}) ? R2 : R1$
 - Employed in most modern ISAs (x86, Alpha)

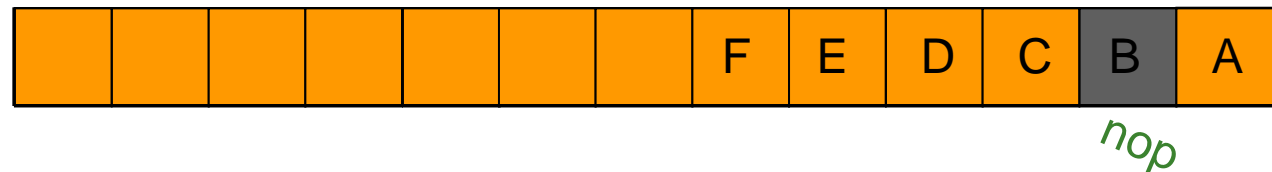
Predicated Execution (II)

- Predicated execution can be high performance and energy-efficient



Predicated Execution

Fetch Decode Rename Schedule RegisterRead Execute



Branch Prediction

Fetch Decode Rename Schedule RegisterRead Execute



Pipeline flush!!

Predicated Execution (III)

■ Advantages:

- + Eliminates mispredictions for hard-to-predict branches
 - + No need for branch prediction for some branches
 - + **Good if misprediction cost > useless work due to predication**
- + Enables code optimizations hindered by the control dependency
 - + Can move instructions more freely within predicated code
 - + Vectorization with control flow
- + Reduces fetch breaks (straight-line code)

■ Disadvantages:

- Causes useless work for branches that are easy to predict
 - **Reduces performance if misprediction cost < useless work**
 - **Adaptivity**: Static predication is not adaptive to run-time branch behavior. Branch behavior changes based on input set, phase, control-flow path.
- Additional hardware and ISA support (complicates renaming and OOO)
- Cannot eliminate all hard to predict branches
 - Complex control flow graphs, function calls, and loop branches
- Additional data dependencies delay execution (problem esp. for easy branches)

Idealism

- Wouldn't it be nice
 - If the branch is eliminated (predicated) when it will actually be mispredicted
 - If the branch were predicted when it will actually be correctly predicted

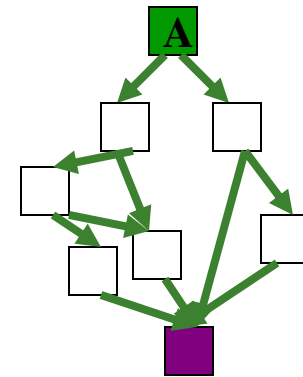
- Wouldn't it be nice
 - If predication did not require ISA support

Improving Predicated Execution

- Three major limitations of predication
 1. **Adaptivity**: non-adaptive to branch behavior
 2. **Complex CFG**: inapplicable to loops/complex control flow graphs
 3. **ISA**: Requires large ISA changes

- Wish Branches
 - Solve 1 and partially 2 (for loops)

- Dynamic Predicated Execution
 - Dynamic simple hammock predication
 - Solves 1 and 3
 - Diverge-Merge Processor
 - Solves 1, 2, 3

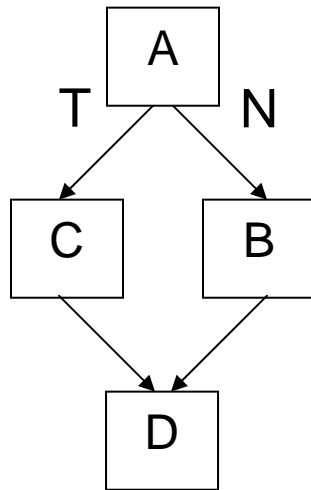


Wish Branches

- The **compiler** generates code (with wish branches) that can be executed **either** as predicated code **or** non-predicated code (normal branch code)
- The **hardware decides** to execute predicated code or normal branch code at run-time based on the confidence of branch prediction
- **Easy to predict: normal branch code**
- **Hard to predict: predicated code**
- Kim et al., "**Wish Branches: Enabling Adaptive and Aggressive Predicated Execution**," IEEE Micro Top Picks, Jan/Feb 2006.

Wish Jump/Join

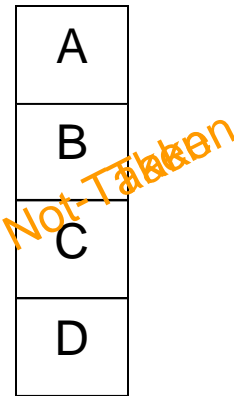
High Confidence



```

A  p1 = (cond)
   branch p1, TARGET
B  mov b, 1
   jmp JOIN
C  TARGET:
   mov b,0
  
```

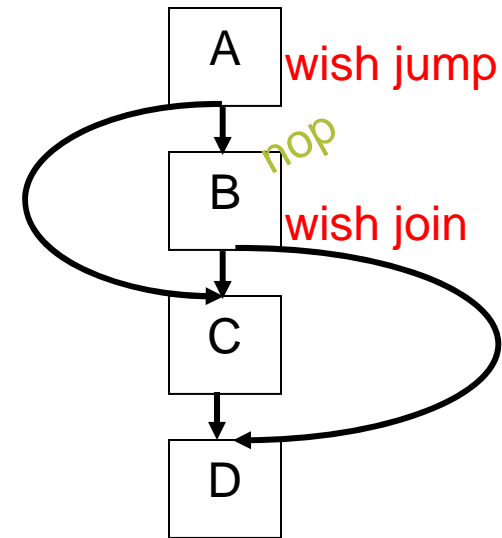
normal branch code



```

A  p1 = (cond)
B  (!p1) mov b,1
C  (p1) mov b,0
  
```

predicated code

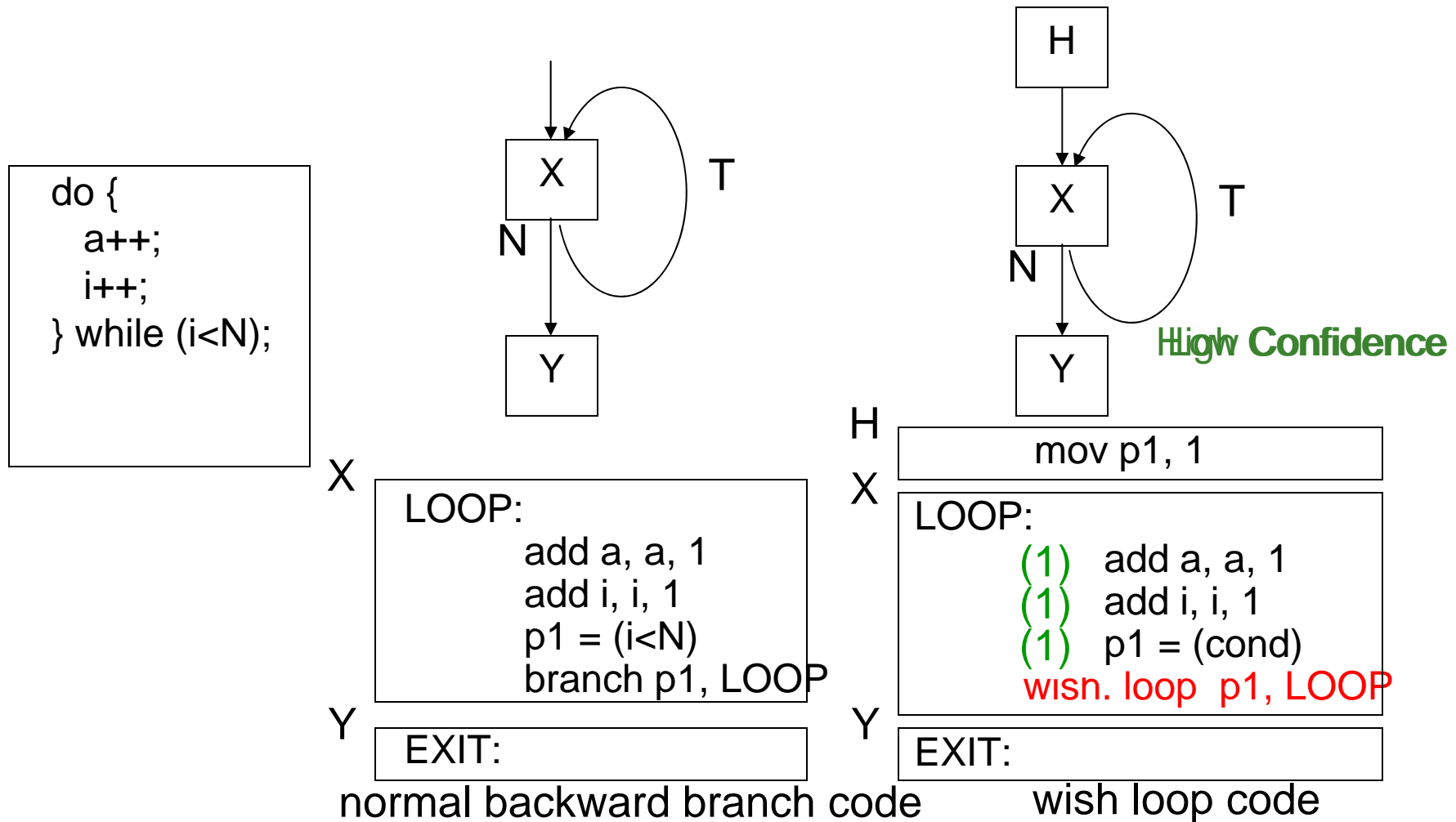


```

A  p1=(cond)
   wish.jump p1 TARGET
B  (!p1) mov b,1
   wish.join p1 JOIN
   nop
C  TARGET:
   (p1) mov b,0
D  JOIN:
  
```

wish jump/join code

Wish Loop



Wish Branches vs. Predicated Execution

- Advantages compared to predicated execution
 - **Reduces the overhead** of predication
 - Increases the benefits of predicated code by allowing the compiler to generate more **aggressively-predicated code**
 - Provides a mechanism to exploit predication to reduce the branch misprediction penalty for **backward branches (Wish loops)**
 - Makes predicated code less dependent on machine configuration (e.g. branch predictor)

- Disadvantages compared to predicated execution
 - Extra branch instructions use machine resources
 - Extra branch instructions increase the contention for branch predictor table entries
 - **Constrains the compiler's scope for code optimizations**

Wish Branches vs. Branch Prediction

- Advantages
 - Can eliminate hard-to-predict branches (determined dynamically)

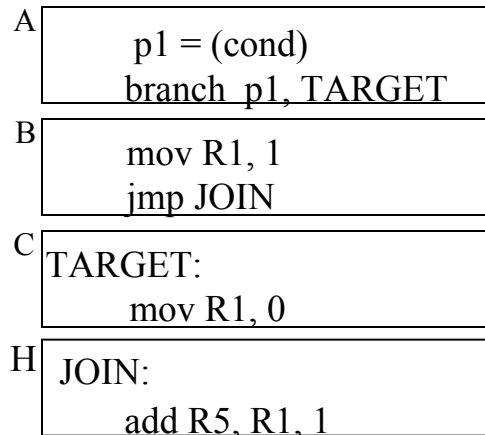
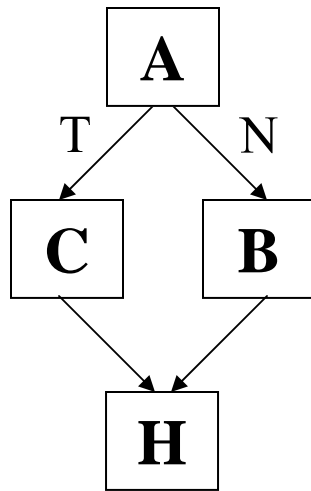
- Disadvantages
 - What if the confidence estimation is wrong?
 - Requires predication support in the ISA
 - Requires extra instructions in the ISA
 - Inapplicable to complex control flow graphs

- Remember the three major limitations of predication
 1. **Adaptivity**: non-adaptive to branch behavior
 2. **Complex CFG**: inapplicable to loops/complex control flow graphs
 3. **ISA**: Requires large ISA changes

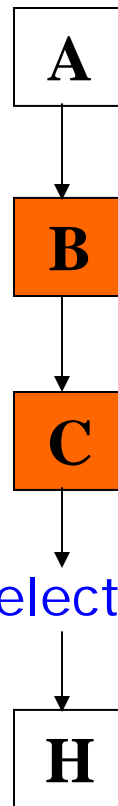
Dynamic Predicated Execution (I)

- The **compiler** identifies
 - Diverge branches
 - Control-flow merge (CFM) points
- The **microarchitecture** **decides when** and **what** to predicate dynamically.
- Klauser et al., "**Dynamic hammock predication**," PACT 1998.
- Kim et al., "**Diverge-Merge Processor: Generalized and Energy-Efficient Dynamic Predication**," IEEE Micro Top Picks, Jan/Feb 2007.

Dynamic Hammock Predication (II)



Low-confidence



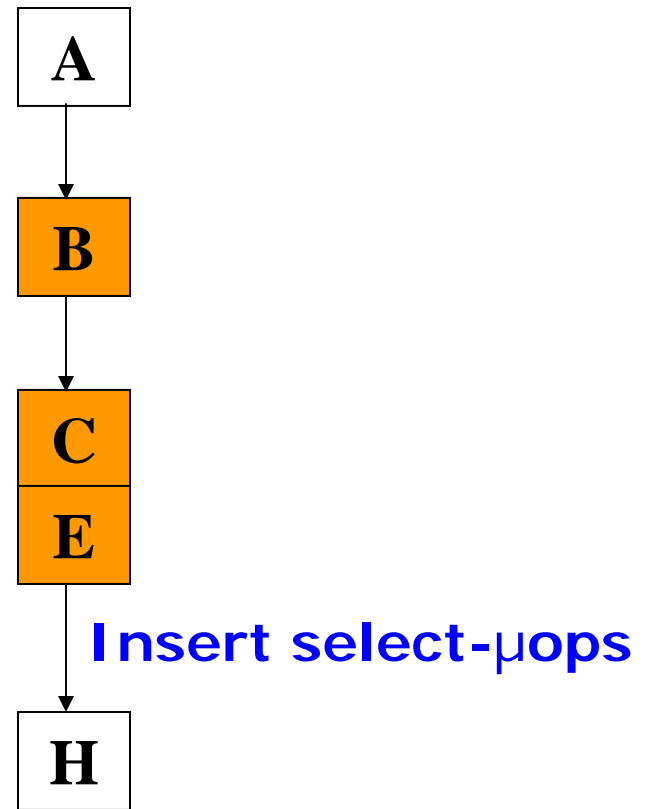
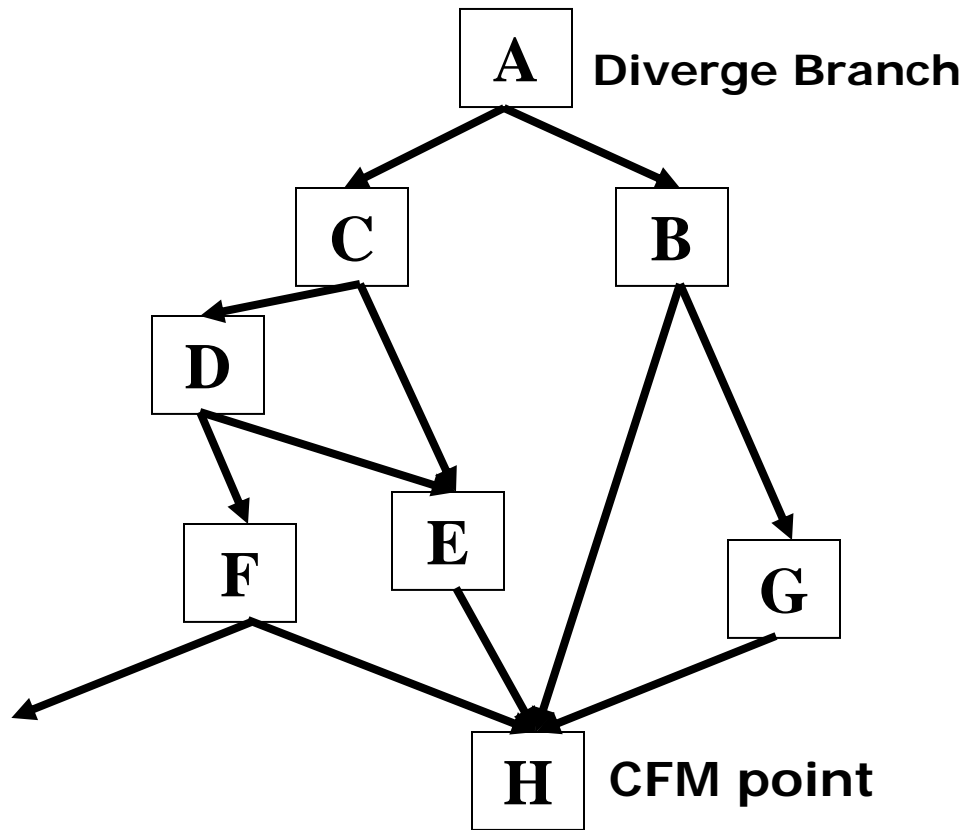
(mov R1, 1)
PR10 = 1



(mov R1, 0)
PR11 = 0

select- μ ops (ϕ -nodes in SSA)

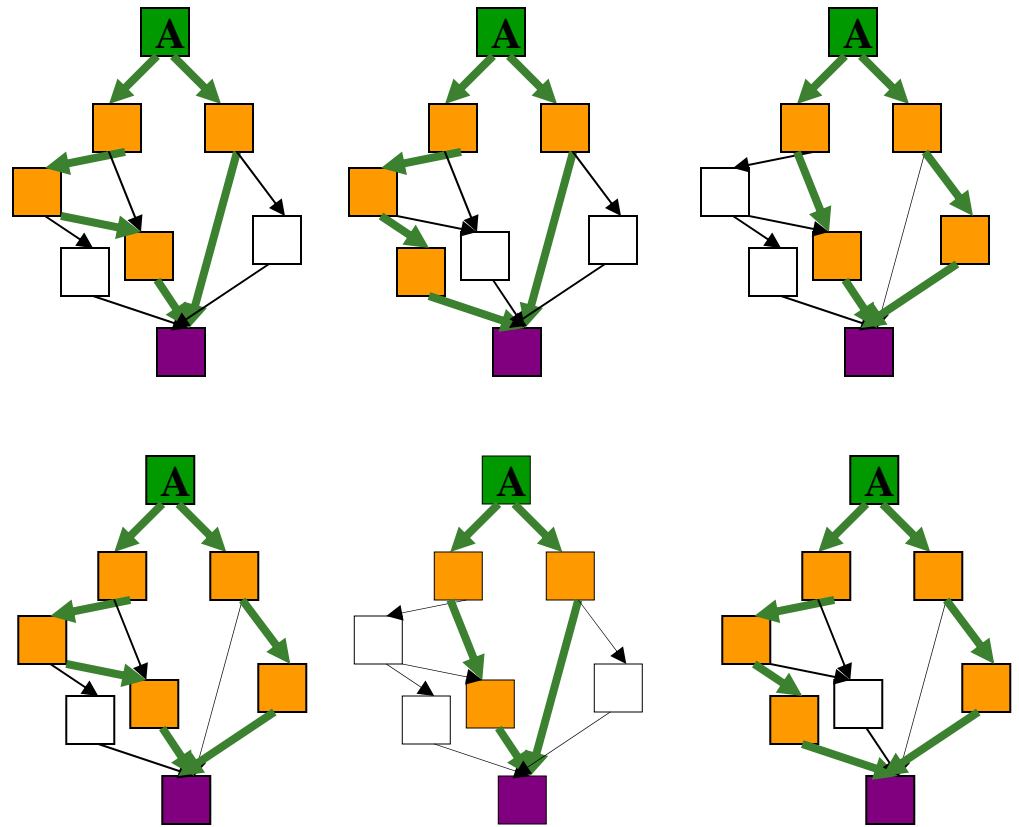
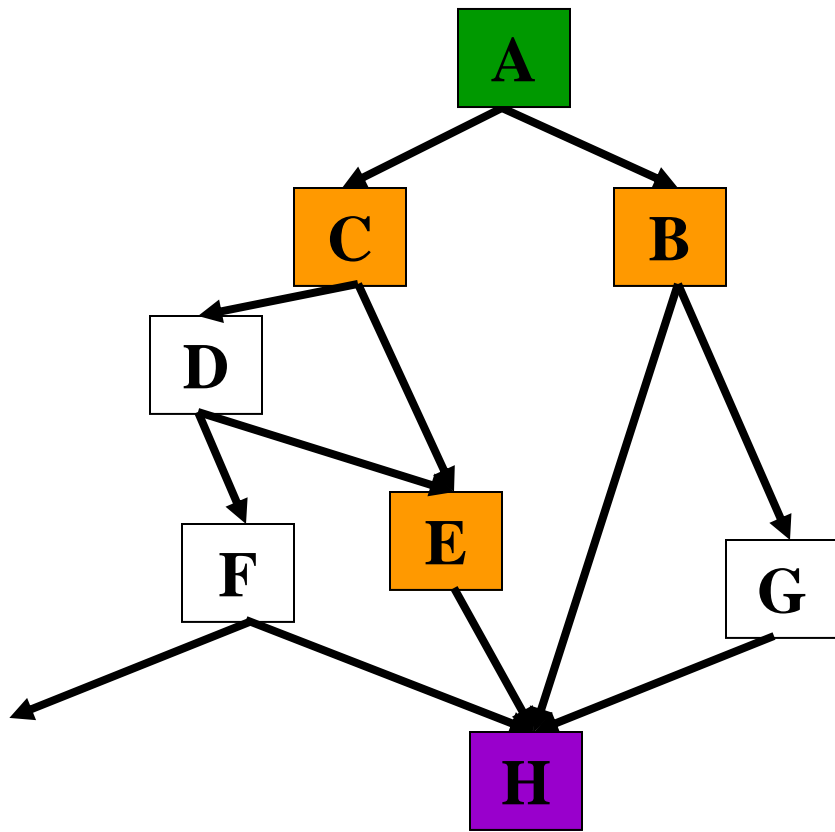
PR12 = (cond) ? PR11 : PR10

Diverge-Merge Processor (III)



-  Frequently executed path
-  Not frequently executed path

Diverge-Merge Processor (IV)



→ Frequently executed path ■ diverge-branch ■ executed block ■ CFM point
→ Not frequently executed path

Dynamic Predicated Execution (V)

■ Advantages:

- + Adapts to branch behavior based on accurate runtime information
 - + Easy to predict: Predict
 - + Hard to predict: Predicate
 - ++ Hardware can more accurately determine easy vs. hard
- + Enables predication of complex control flow graphs, loops, ...
- + No need for predicated instructions & pred. registers in the ISA

■ Disadvantages:

- Hardware complexity increases (see Kim et al., MICRO 2006)
- Still requires some ISA support
 - Determining CFM points is costly in hardware
- No code optimization benefits of conventional predication

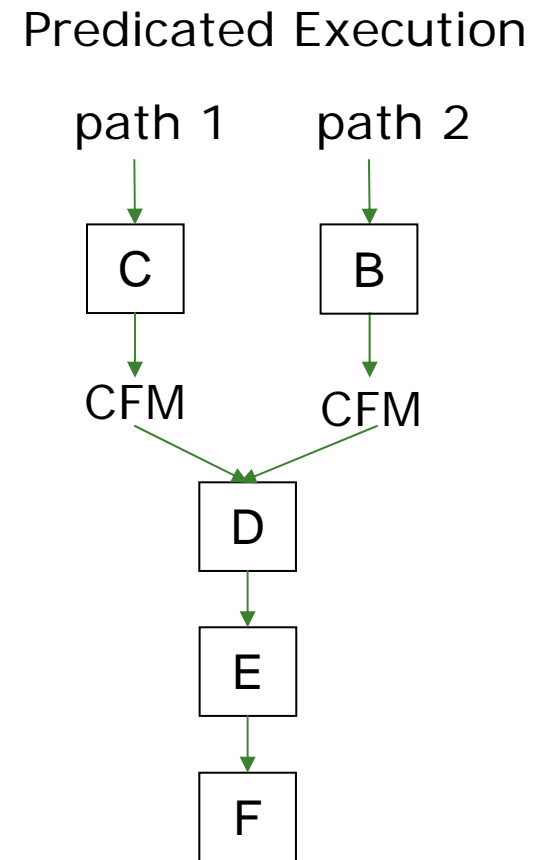
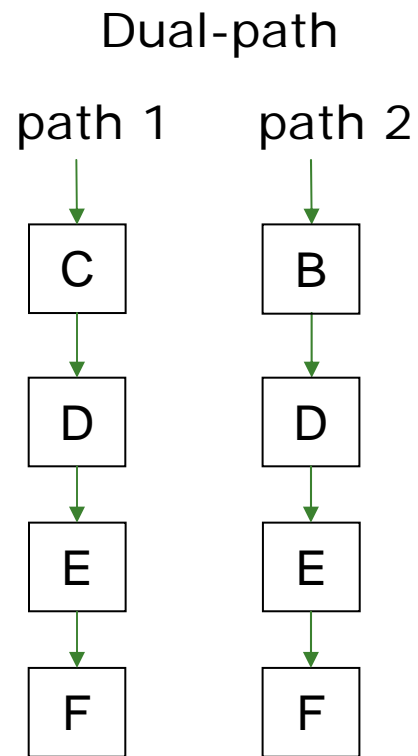
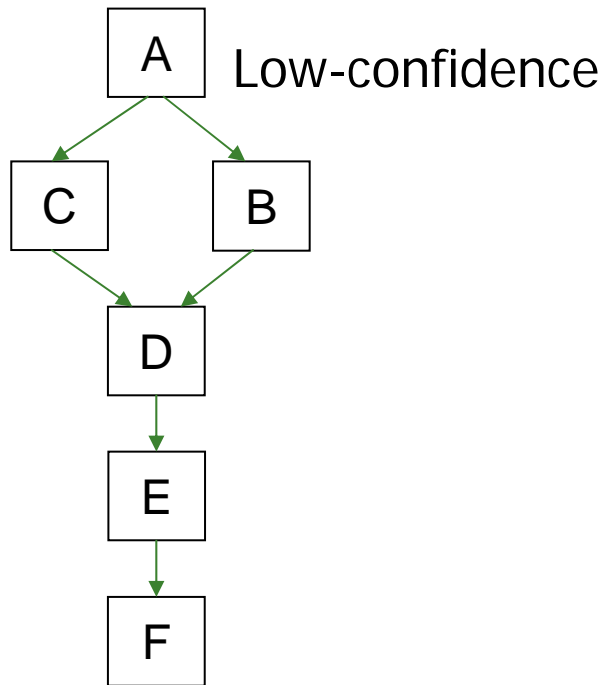
Multi-Path Execution

- Idea: Execute both paths after a conditional branch
 - For all branches: Riseman and Foster, "The inhibition of potential parallelism by conditional jumps," IEEE Transactions on Computers, 1972.
 - For a hard-to-predict branch: Use dynamic confidence estimation

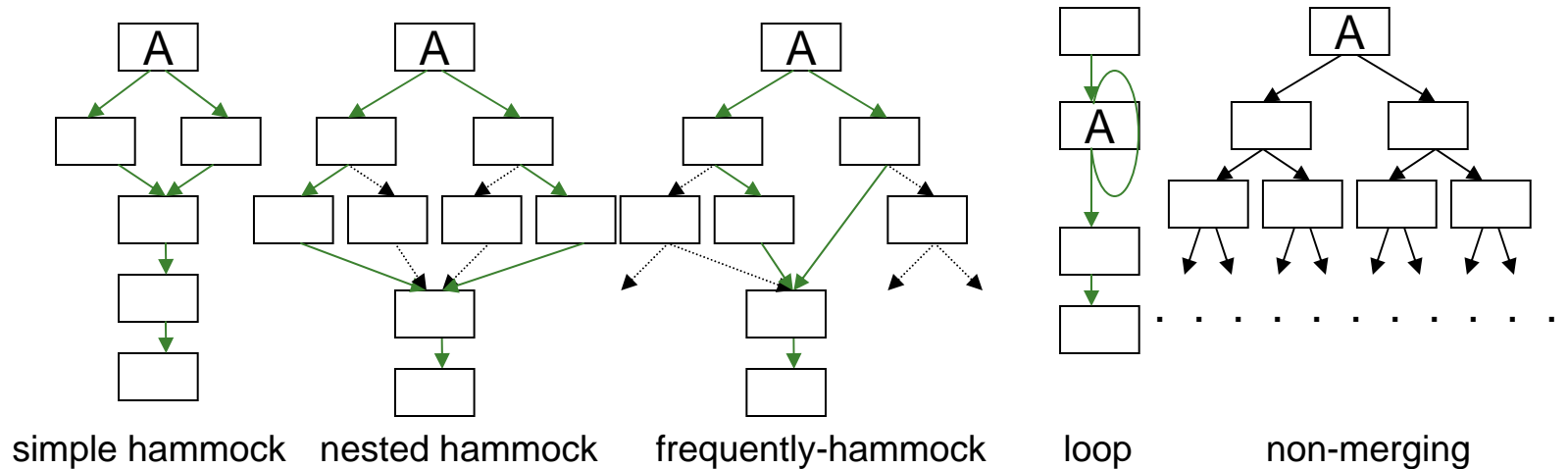
- Advantages:
 - + Improves performance if misprediction cost > useless work
 - + No ISA change needed

- Disadvantages:
 - What happens when the machine encounters another hard-to-predict branch? Execute both paths again?
 - Paths followed quickly become exponential
 - Each followed path requires its own register alias table, PC, GHR
 - Wasted work (and reduced performance) if paths merge

Dual-Path Execution versus Dynamic Predication

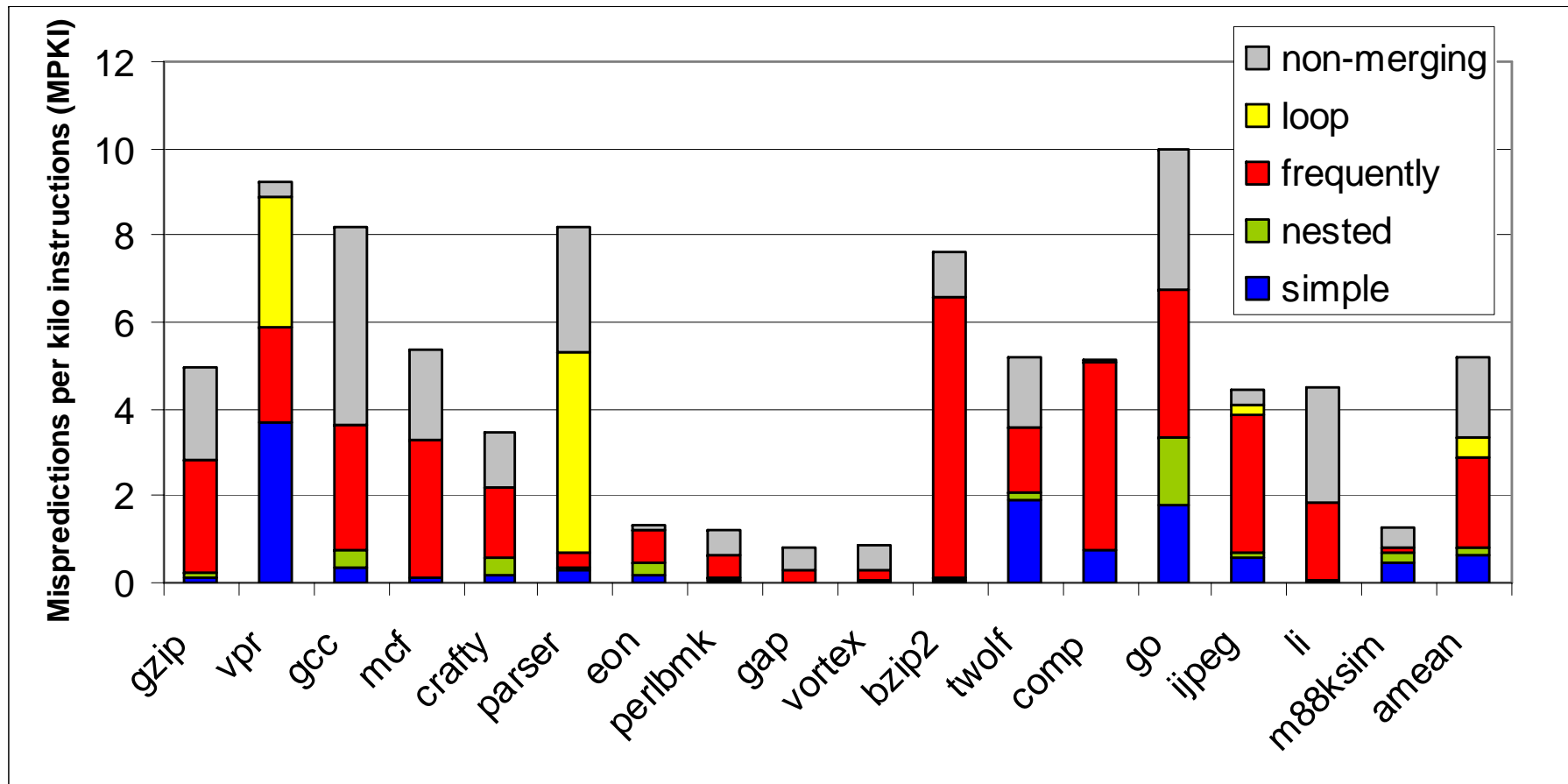


Summary of Alternative Branch Handling Techniques



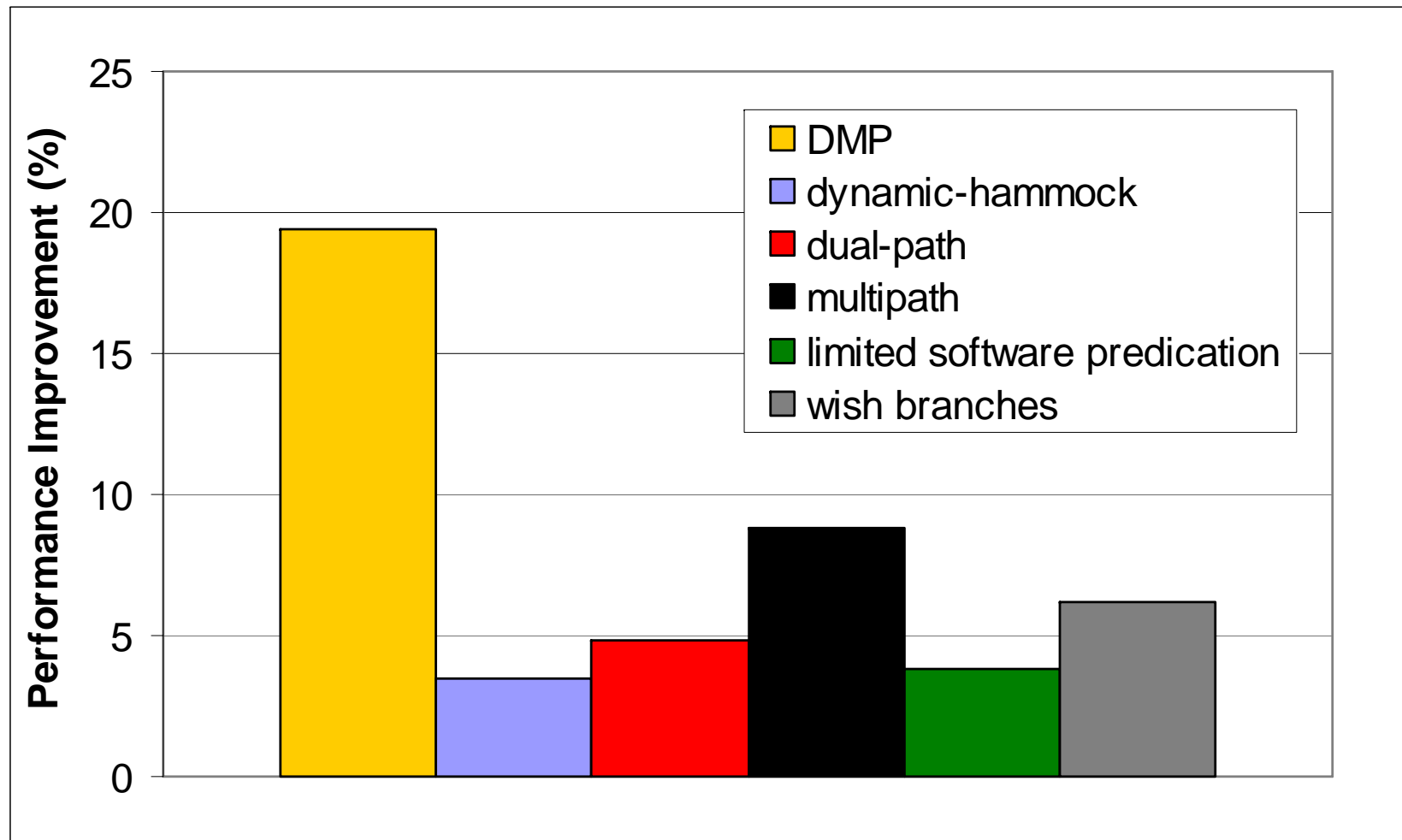
	simple hammock	nested hammock	frequently-hammock	loop	non-merging
Diverge-Merge	●	●	●	●	×
Dynamic-hammock	●	×	×	×	×
Software predication	●	▲	sometimes	×	×
Wish br.	●	▲	sometimes	●	×
Dual-path	▲	▲	▲	▲	●

Distribution of Mispredicted Branches

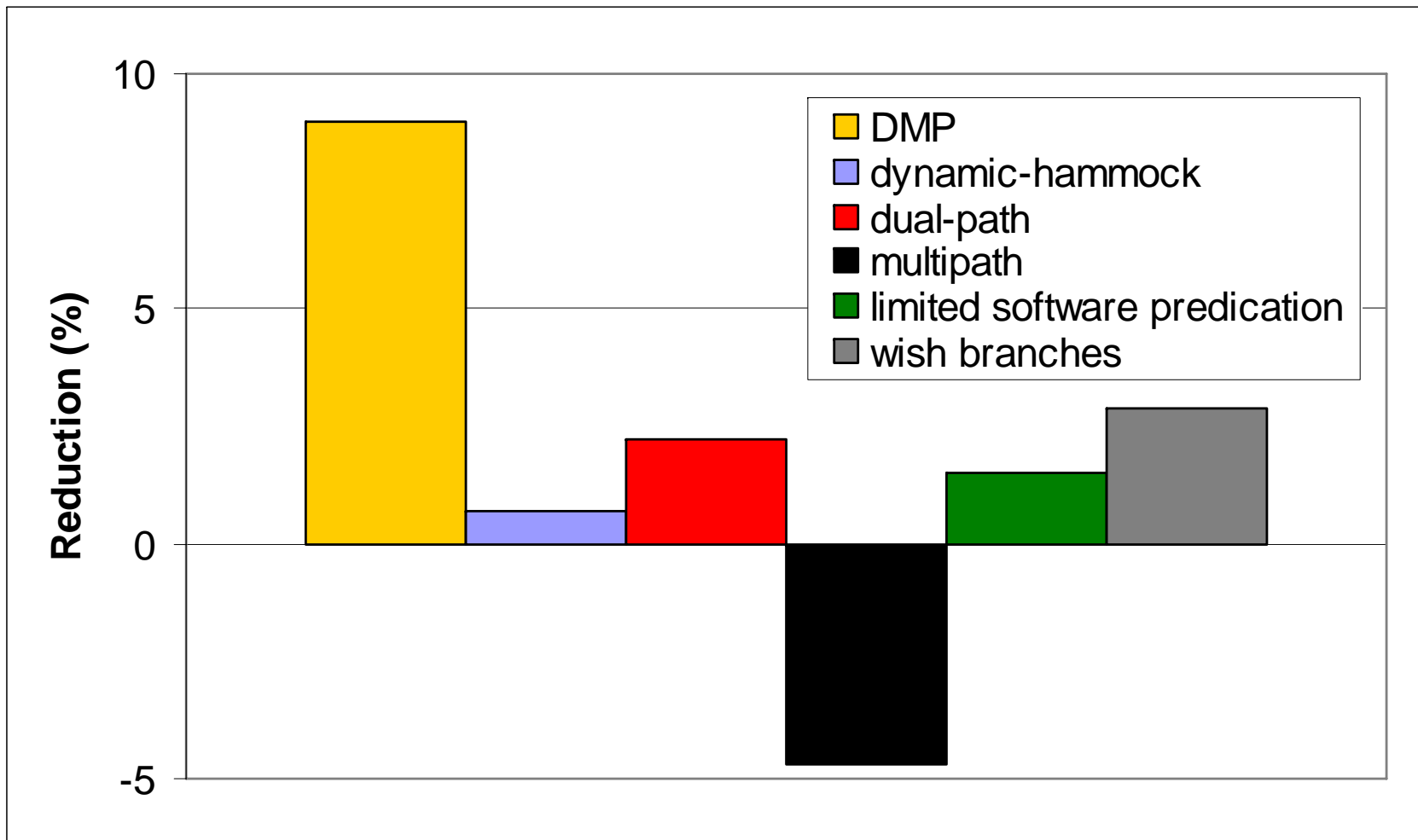


- Kim et al., "Diverge-Merge Processor (DMP): Dynamic Predicated Execution of Complex Control-Flow Graphs Based on Frequently Executed Paths," MICRO 2006.
 - Slides 24-27

Performance of Alternative Techniques



Energy Savings of Alternative Techniques



Branch Confidence Estimation

- How do we dynamically decide whether or not a branch is hard to predict?
 - Idea: Use a table of counters to keep track of the mispredictions for a branch (organized like a branch predictor)
 - If (misprediction saturating counter $>$ threshold)
 - Estimate branch is difficult to predict
 - Jacobsen et al., "Assigning Confidence to Conditional Branch Predictions," MICRO 1996.
- Many things can be done for a difficult to predict branch
 - Stall fetch (save energy)
 - Fetch from a thread with easier-to-predict branches
 - Wish branches, dynamic predicated execution, selective dual-path
 - Reverse branch prediction?

Research Issues in Control Flow Handling

- **More hardware/software cooperation**
 - Software has scope and powerful analysis techniques
 - Hardware has dynamic information
 - Can we combine the strengths of both?

- **Reducing waste**
 - Exploiting control flow independence
 - Identifying difficult-to-predict branches
 - Gating fetch, context switching
 - Recycling useful work done on wrong path
 - Is wrong-path execution always useless?

- **Indirect jump handling**
 - Common in object oriented languages/programs and virtual machines

Alternative Approaches to Concurrency

Outline

- We have seen out-of-order, superscalar execution (restricted data flow) to exploit instruction level parallelism
 - Burton Smith calls this the HPS cannon
 - B. J. Smith, "**Reinventing Computing**," talk at various venues.

- There are many other approaches to concurrency
 - SIMD/MIMD classification
 - DAE: Decoupled Access/Execute
 - VLIW: Very Long Instruction Word
 - SIMD: Vector Processors and Array Processors
 - Data Flow → Mainly in ECE 742 (Spring 2011)
 - Multithreading → Mainly in ECE 742 (Spring 2011)
 - Multiprocessing → Mainly in ECE 742 (Spring 2011)
 - Systolic Arrays → ECE 742 (Spring 2011)

Readings

- Required:
 - Fisher, “**Very Long Instruction Word architectures and the ELI-512,**” ISCA 1983.
 - Huck et al., “**Introducing the IA-64 Architecture,**” IEEE Micro 2000.

- Recommended:
 - Russell, “**The CRAY-1 computer system,**” CACM 1978.
 - Rau and Fisher, “**Instruction-level parallel processing: history, overview, and perspective,**” Journal of Supercomputing, 1993.
 - Faraboschi et al., “**Instruction Scheduling for Instruction Level Parallel Processors,**” Proc. IEEE, Nov. 2001.

SIMD/MIMD Classification of Computers

- Mike Flynn, “[Very High Speed Computing Systems](#),” Proc. of the IEEE, 1966
- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
 - Array processor
 - Vector processor
- **MISD?** Multiple instructions operate on single data element
 - Closest form: systolic array processor?
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
 - Multiprocessor
 - Multithreaded processor

SPMD

- Single procedure/program, multiple data
 - This is a programming model rather than computer organization
- Each processing element executes the same procedure, except on different data elements
 - Procedures can synchronize at certain points in program, e.g. barriers
- Essentially, multiple instruction streams execute the same program
 - Each program/procedure can 1) execute a different control-flow path, 2) work on different data, at run-time
 - Many scientific applications programmed this way and run on MIMD computers (multiprocessors)
 - Modern GPUs programmed in a similar way on a SIMD computer

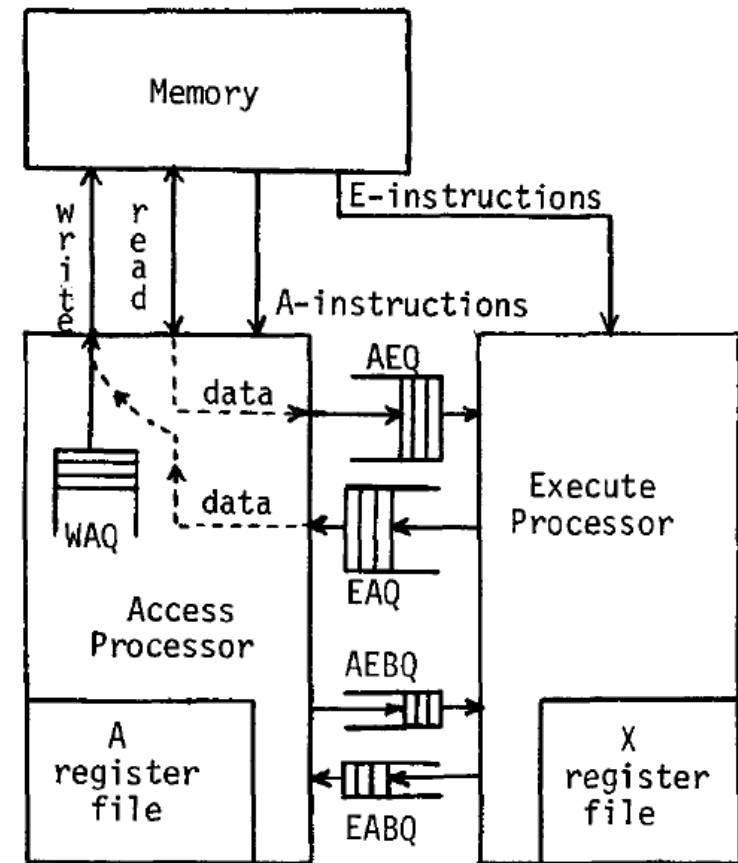
SISD Parallelism Extraction Techniques

- We have already seen
 - Superscalar execution
 - Out-of-order execution

- Are there simpler ways of extracting SISD parallelism?
 - Decoupled Access/Execute
 - VLIW (Very Long Instruction Word)

Decoupled Access/Execute

- Motivation: Tomasulo's algorithm too complex to implement
 - 1980s before HPS, Pentium Pro
- Idea: Decouple operand access and execution via **two separate instruction streams that communicate via ISA-visible queues**.
- Smith, "**Decoupled Access/Execute Computer Architectures**," ISCA 1982, ACM TOCS 1984.



Decoupled Access/Execute (II)

- Compiler generates two instruction streams (A and E)
 - Synchronizes the two upon control flow instructions (using branch queues)

```

q = 0.0
Do 1 k = 1, 400
1 x(k) = q + y(k) * (r * z(k+10) + t * z(k+11))
    
```

Fig. 2a. Lawrence Livermore Loop 1 (HYDRO EXCERPT)

A7 ← -400	. negative loop count
A2 ← 0	. initialize index
A3 ← 1	. index increment
X2 ← r	. load loop invariants
X5 ← t	. into registers
loop: X3 ← z + 10, A2	. load z(k+10)
X7 ← z + 11, A2	. load z(k+11)
X4 ← X2 *f X3	. r*z(k+10)-flt. mult.
X3 ← X5 *f X7	. t * z(k+11)
X7 ← y, A2	. load y(k)
X6 ← X3 +f X4	. r*z(x+10)+t*z(k+11))
X4 ← X7 *f X6	. y(k) * (above)
A7 ← A7 + 1	. increment loop counter
x, A2 ← X4	. store into x(k)
A2 ← A2 + A3	. increment index
JAM loop	. Branch if A7 < 0

Fig. 2b. Compilation onto CRAY-1-like architecture

<u>Access</u>	<u>Execute</u>
.	.
.	.
.	.
AEQ ← z + 10, A2	X4 ← X2 *f AEQ
AEQ ← z + 11, A2	X3 ← X5 *f AEQ
AEQ ← y, A2	X6 ← X3 +f X4
A7 ← A7 + 1	EAQ ← AEQ *f X6
x, A2 ← EAQ	.
A2 ← A2 + A3	.
.	.
.	.
.	.

Fig. 2c. Access and execute programs for straight-line section of loop

Decoupled Access/Execute (III)

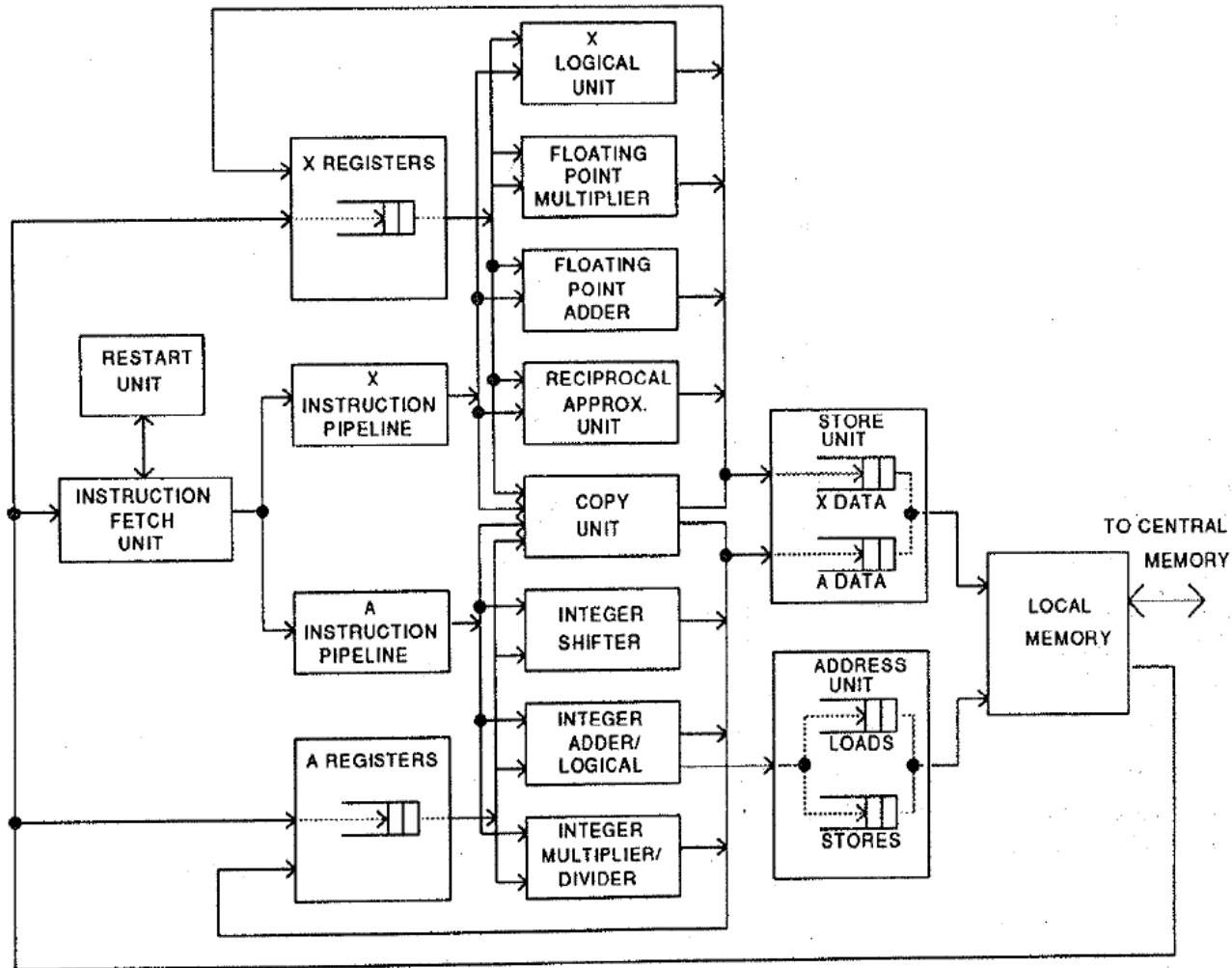
■ Advantages:

- + Execute stream can run ahead of the access stream and vice versa
 - + If A takes a cache miss, E can perform useful work
 - + If A hits in cache, it supplies data to lagging E
 - + Queues reduce the number of required registers
- + Limited out-of-order execution without wakeup/select complexity

■ Disadvantages:

- Compiler support to partition the program and manage queues
 - Determines the amount of decoupling
- Branch instructions require synchronization between A and E
- Multiple instruction streams (can be done with a single one, though)

Astronautics ZS-1



- Single stream steered into A and X pipelines
- Each pipeline in-order
- Smith et al., "The ZS-1 central processor," ASPLOS 1987.
- Smith, "Dynamic Instruction Scheduling and the Astronautics ZS-1," IEEE Computer 1989.

Astronautics ZS-1 Instruction Scheduling

- Dynamic scheduling
 - A and X streams are issued/executed independently
 - Loads can bypass stores in the memory unit (if no conflict)
 - Branches executed early in the pipeline
 - To reduce synchronization penalty of A/X streams
 - Works only if the register a branch sources is available
- Static scheduling
 - Move compare instructions as early as possible before a branch
 - So that branch source register is available when branch is decoded
 - Reorder code to expose parallelism in each stream
 - Loop unrolling:
 - Reduces branch count + exposes code reordering opportunities

Loop Unrolling

```
i = 1;
while ( i < 100 ) {
    a[i] = b[i+1] + (i+1)/m
    b[i] = a[i-1] - i/m
    i = i + 1
}
```

```
i = 1;
while ( i < 100 ) {
    a[i] = b[i+1] + (i+1)/m
    b[i] = a[i-1] - i/m
    a[i+1] = b[i+2] + (i+2)/m
    b[i+1] = a[i] - (i+1)/m
    i = i + 2
}
```

- Idea: Replicate loop body multiple times within an iteration
- + Reduces loop maintenance overhead
 - Induction variable increment or loop condition test
- + Enlarges basic block (and analysis scope)
 - Enables code optimization and scheduling opportunities
- What if iteration count not a multiple of unroll factor? (need extra code to detect this)
- Increases code size