

CS252
Graduate Computer Architecture
Lecture 7

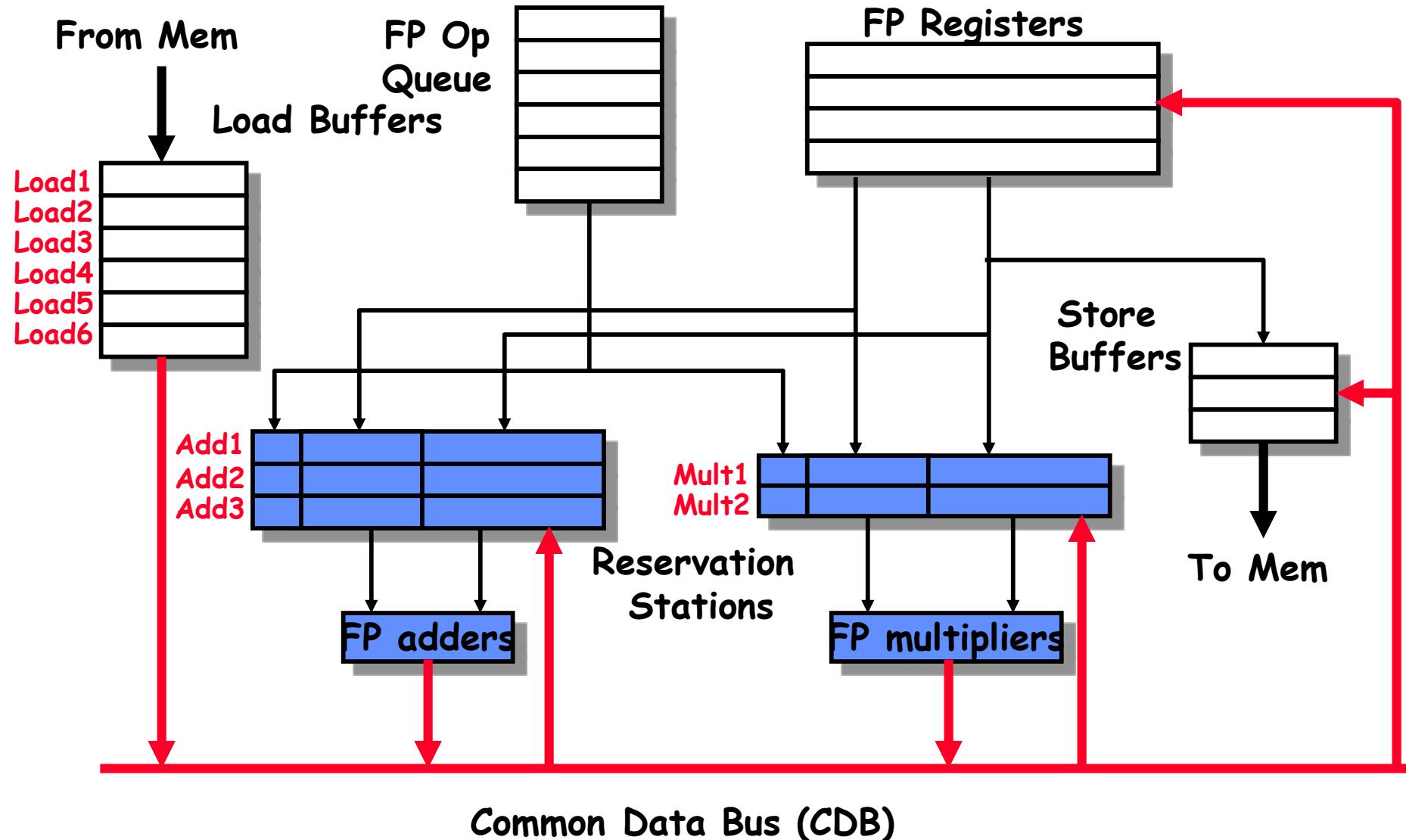
Reorder Buffers
and
Explicit Register Renaming

September 22, 2000
Prof. John Kubiakowicz

Review: Dynamic hardware techniques for out-of-order execution

- HW exploitation of ILP
 - Works when can't know dependence at compile time.
 - Code for one machine runs well on another
- Scoreboard (ala CDC 6600 in 1963)
 - Centralized control structure
 - No register renaming, no forwarding
 - Pipeline stalls for WAR and WAW hazards.
 - Are these fundamental limitations??? (No)
- Reservation stations (ala IBM 360/91 in 1966)
 - Distributed control structures
 - Implicit renaming of registers (dispatched pointers)
 - WAR and WAW hazards eliminated by register renaming
 - Results broadcast to all reservation stations for RAW

Review: Tomasulo Organization



Review: Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

If reservation station free (no structural hazard), control issues instr & sends operands (renames registers).

2. Execution—operate on operands (EX)

When both operands ready then execute;
if not ready, watch Common Data Bus for result

3. Write result—finish execution (WB)

Write on Common Data Bus to all awaiting units;
mark reservation station available

- Common data bus: data + source ("come from" bus)
 - 64 bits of data + 4 bits of Functional Unit source address
 - Write if matches expected Functional Unit (produces result)
 - Does the broadcast

Review: Why can Tomasulo overlap iterations of loops?

- **Implicit Register Renaming**
 - Multiple iterations use different physical destinations for registers (dynamic loop unrolling)
 - No WAR or WAW hazards to worry about
 - » Note that a compiler couldn't get rid of these hazards without unrolling the loop
 - On-the-fly setup of data flow graph
- **Reservation stations...**
 - Serve as destinations for information \Rightarrow surrogate registers
 - Serve as temporary holding places for register values
 - Serve as distributed scheduling points for information
- **Tomasulo building “DataFlow” graph on the fly.**

Review: Loop Example Cycle 9

Instruction status:

ITER	Instruction	j	k	Exec Write		Busy	Addr	Fu
				Issue	CompResult			
1	LD	F0	0	R1	1 9	Load1	Yes	80
1	MULTD	F4	F0	F2	2	Load2	Yes	72
1	SD	F4	0	R1	3	Load3	No	
2	LD	F0	0	R1	6	Store1	Yes	80
2	MULTD	F4	F0	F2	7	Store2	Yes	72
2	SD	F4	0	R1	8	Store3	No	

Reservation Stations:

Time	Name	Busy	Op	Vj	V _k	S1	S2	R _S	Code
						Q _j	Q _k		
	Add1	No							LD F0 0 R1
	Add2	No							MULTD F4 F0 F2
	Add3	No							SD F4 0 R1
1	Mult1	Yes	Multd			R(F2)	Load1		SUBI R1 R1 #8
1	Mult2	Yes	Multd			R(F2)	Load2		BNEZ R1 Loop

Register result status

Clock	R1	F0	F2	F4	F6	F8	F10	F12	...	F30
9	72	Fu	Load2		Mult2					

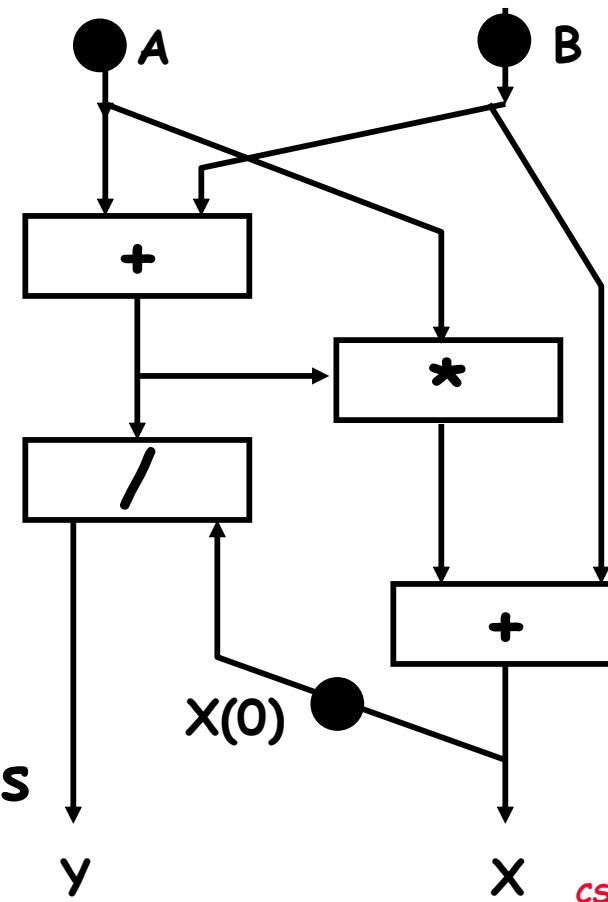
- Dataflow graph constructed completely in hardware

Review: Data-Flow Architectures

- Basic Idea: Hardware represents direct encoding of compiler dataflow graphs:

Input: a, b
 $y := (a+b) / x$
 $x := (a * (a+b)) + b$
Output: y, x

- Data flows along arcs in “Tokens”.
- When two tokens arrive at compute box, box “fires” and produces new token.
- Split operations produce copies of tokens

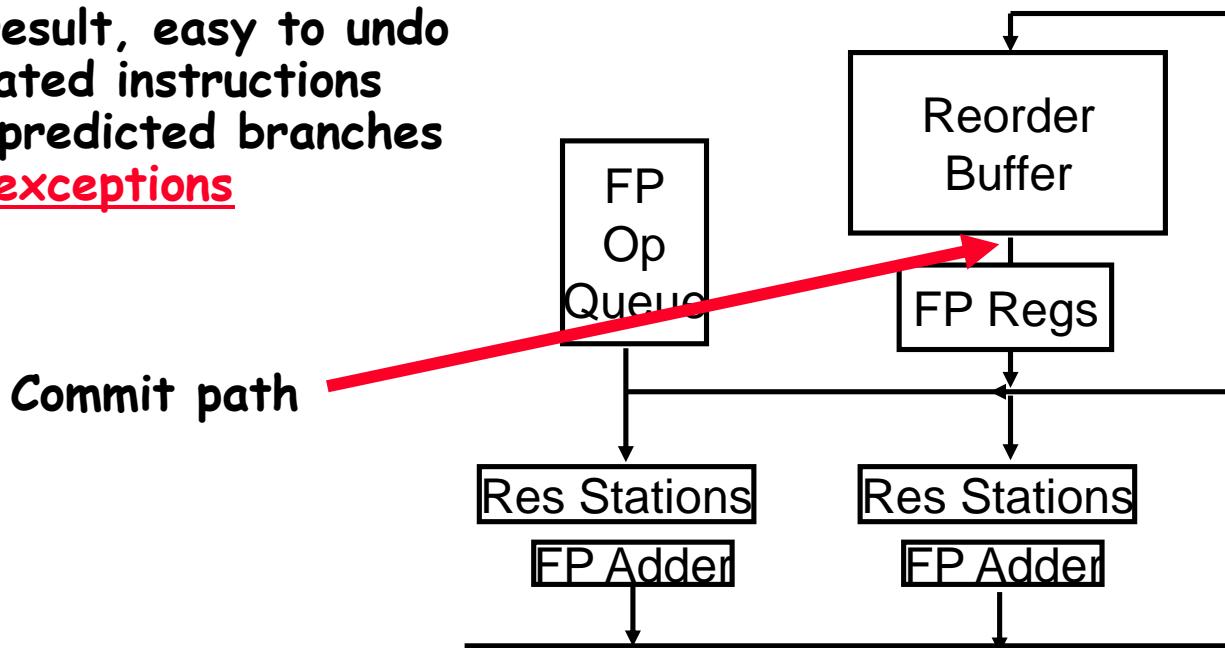


What about Precise Exceptions/Interrupts?

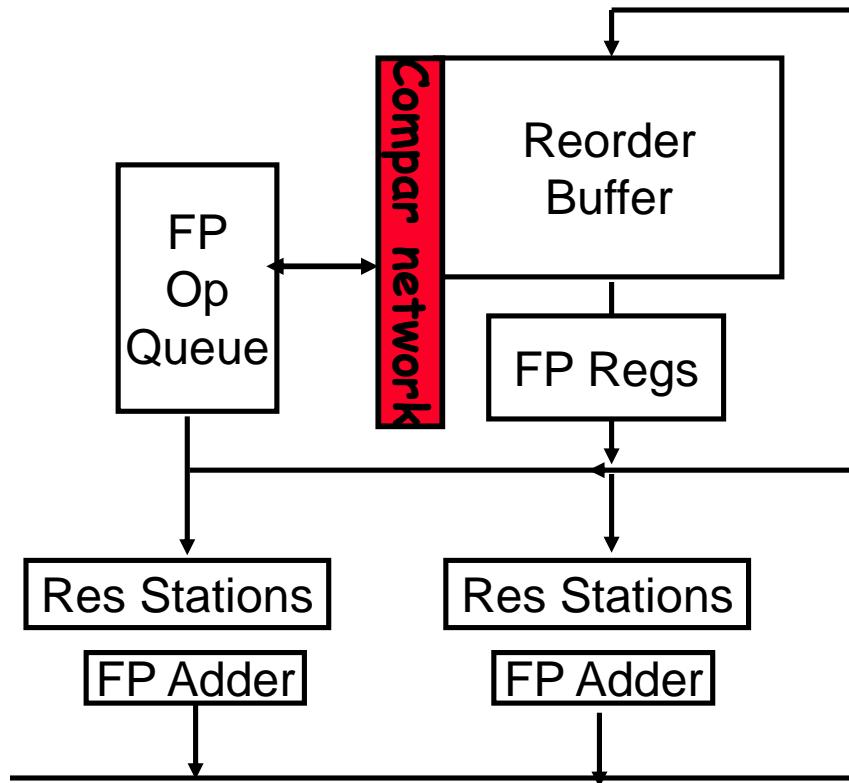
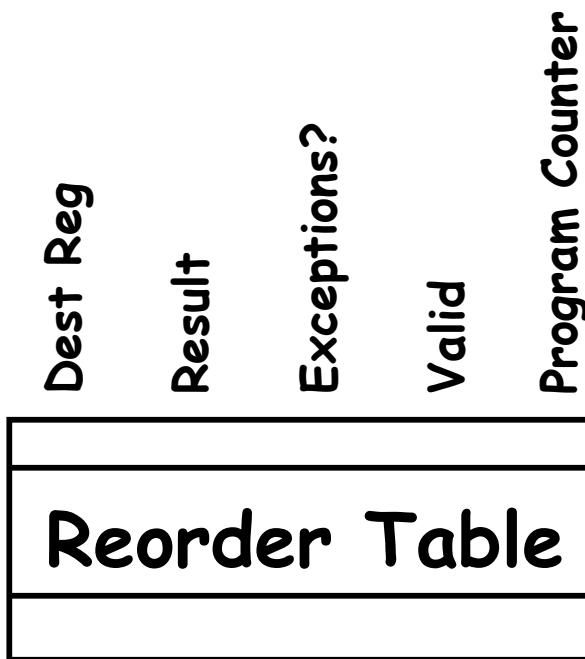
- Both Scoreboard and Tomasulo have:
 - *In-order issue, out-of-order execution, out-of-order completion*
- **Recall:** An interrupt or exception is *precise* if there is a single instruction for which:
 - All instructions before that have committed their state
 - No following instructions (including the interrupting instruction) have modified any state.
- Need way to resynchronize execution with instruction stream (I.e. with issue-order)
 - Easiest way is with *in-order completion* (i.e. reorder buffer)
 - Other Techniques (Smith paper): Future File, History Buffer

HW support for precise interrupts

- Concept of Reorder Buffer (ROB):
 - Holds instructions in FIFO order, exactly as they were issued
 - » Each ROB entry contains PC, dest reg, result, exception status
 - When instructions complete, results placed into ROB
 - » Supplies operands to other instruction between execution complete & commit \Rightarrow more registers like RS
 - » Tag results with ROB buffer number instead of reservation station
 - Instructions **commit** \Rightarrow values at head of ROB placed in registers
 - As a result, easy to undo speculated instructions on mispredicted branches or on exceptions

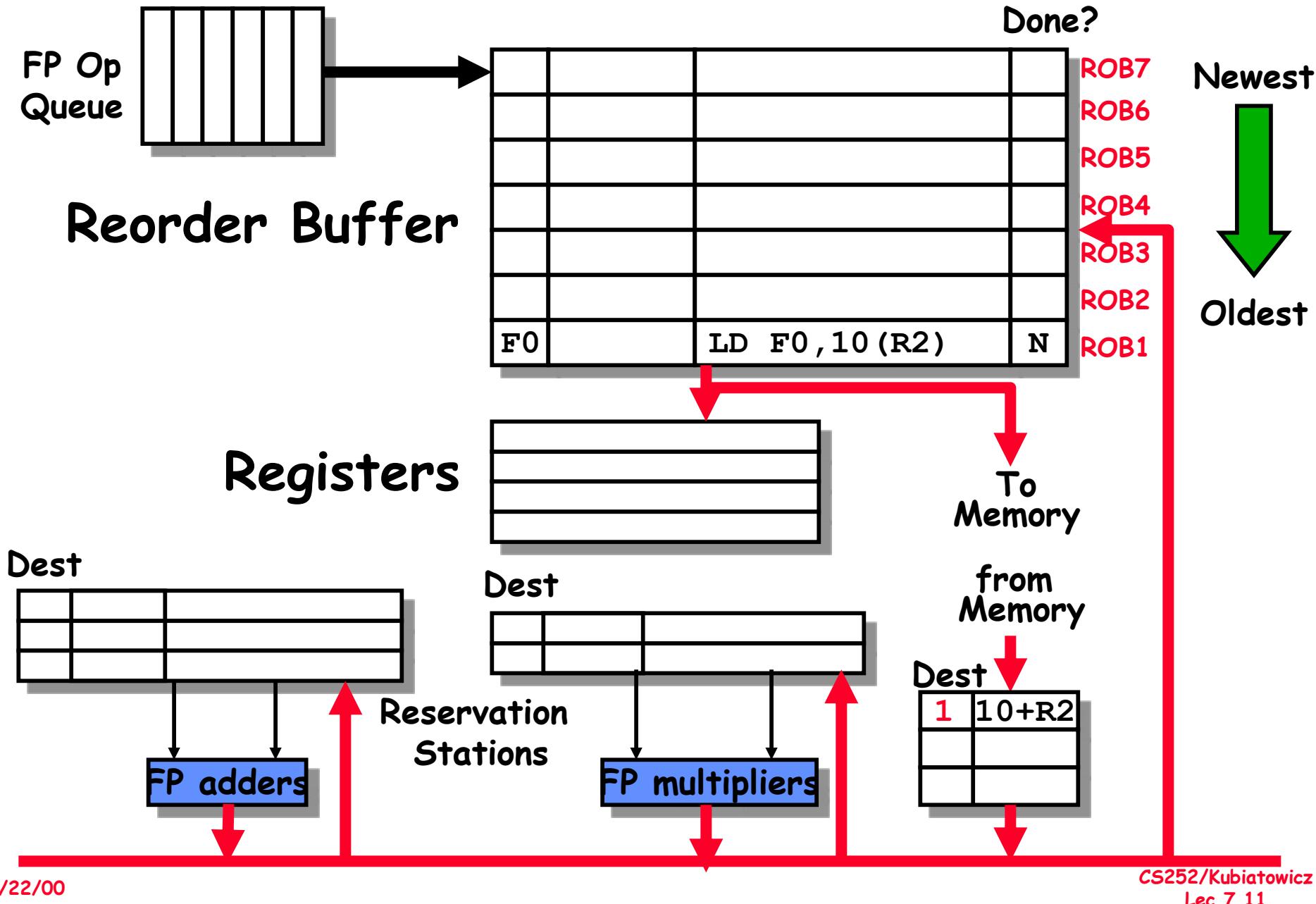


What are the hardware complexities with reorder buffer (ROB)?

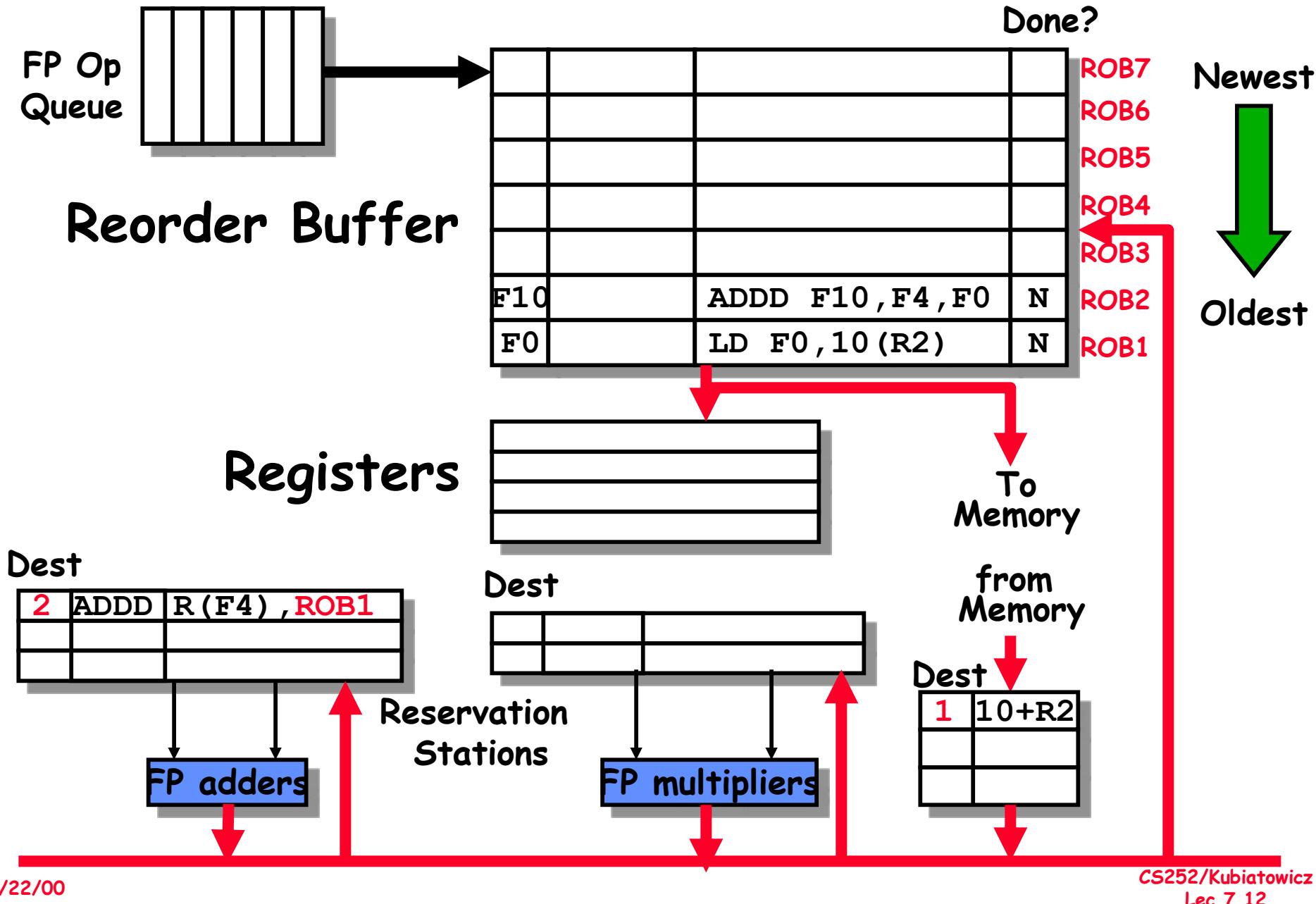


- How do you find the latest version of a register?
 - As specified by Smith paper, need associative comparison network
 - Could use future file or just use the register result status buffer to track which specific reorder buffer has received the value
- Need as many ports on ROB as register file

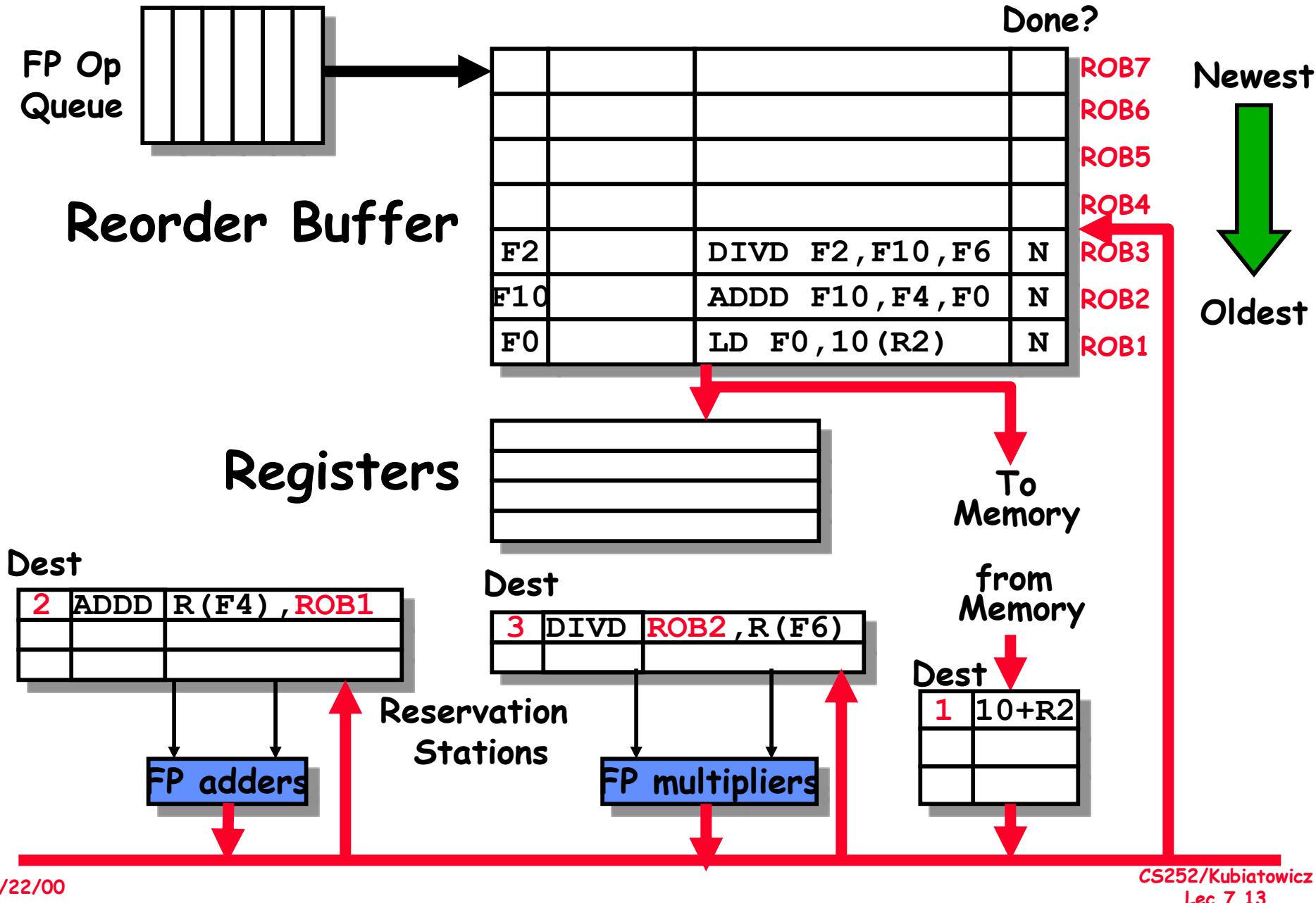
Tomasulo With Reorder buffer:



Tomasulo With Reorder buffer:

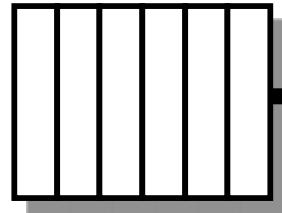


Tomasulo With Reorder buffer:



Tomasulo With Reorder buffer:

FP Op Queue



Reorder Buffer

Done?			
F0	ADDD F0, F4, F6	N	ROB7
F4	LD F4, 0 (R3)	N	ROB6
--	BNE F2, <....>	N	ROB5
F2	DIVD F2, F10, F6	N	ROB4
F10	ADDD F10, F4, F0	N	ROB3
F0	LD F0, 10 (R2)	N	ROB2
			ROB1

Newest
Oldest

Registers

Dest

2	ADDD	R (F4) , ROB1
6	ADDD	ROB5 , R (F6)

FP adders

Reservation Stations

Dest

3	DIVD	ROB2 , R (F6)

FP multipliers

To Memory

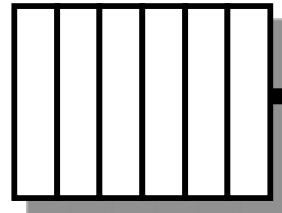
from Memory

Dest

1	10+R2
5	0+R3

Tomasulo With Reorder buffer:

FP Op Queue



Reorder Buffer

	ROB5	ST 0 (R3) , F4	Done?
F0		ADDD F0 , F4 , F6	N
F4		LD F4 , 0 (R3)	N
--		BNE F2 , <....>	N
F2		DIVD F2 , F10 , F6	N
F10		ADDD F10 , F4 , F0	N
F0		LD F0 , 10 (R2)	N

Newest
Oldest

Registers

Dest

2	ADDD	R (F4) , ROB1
6	ADDD	ROB5 , R (F6)

FP adders

Reservation Stations

Dest

3	DIVD	ROB2 , R (F6)

FP multipliers

Dest

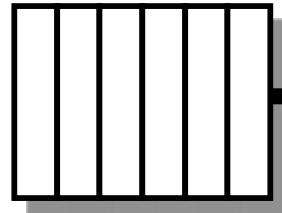
1	10+R2
6	0+R3

from
Memory

To
Memory

Tomasulo With Reorder buffer:

FP Op Queue



Reorder Buffer

			Done?
	M[10]	ST 0 (R3) , F4	Y
F0		ADDD F0 , F4 , F6	N
F4	M[10]	LD F4 , 0 (R3)	Y
--		BNE F2 , <....>	N
F2		DIVD F2 , F10 , F6	N
F10		ADDD F10 , F4 , F0	N
F0		LD F0 , 10 (R2)	N

Newest
Oldest

Registers

Dest

2	ADDD	R (F4) , ROB1
6	ADDD	M[10] , R(F6)

FP adders

Reservation Stations

Dest

3	DIVD	ROB2 , R (F6)

FP multipliers

Dest

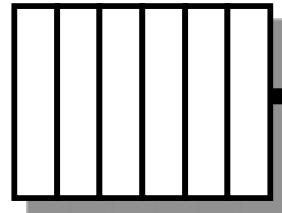
1	10+R2

from
Memory

To
Memory

Tomasulo With Reorder buffer:

FP Op Queue



Reorder Buffer

			Done?
--	M[10]	ST 0 (R3) , F4	Y
F0	<val2>	ADDD F0 , F4 , F6	Ex
F4	M[10]	LD F4 , 0 (R3)	Y
--		BNE F2 , <....>	N
F2		DIVD F2 , F10 , F6	N
F10		ADDD F10 , F4 , F0	N
F0		LD F0 , 10 (R2)	N

Newest
Oldest

Registers

Dest

2	ADDD	R (F4) , ROB1

FP adders

Reservation Stations

Dest

3	DIVD	ROB2 , R (F6)

FP multipliers

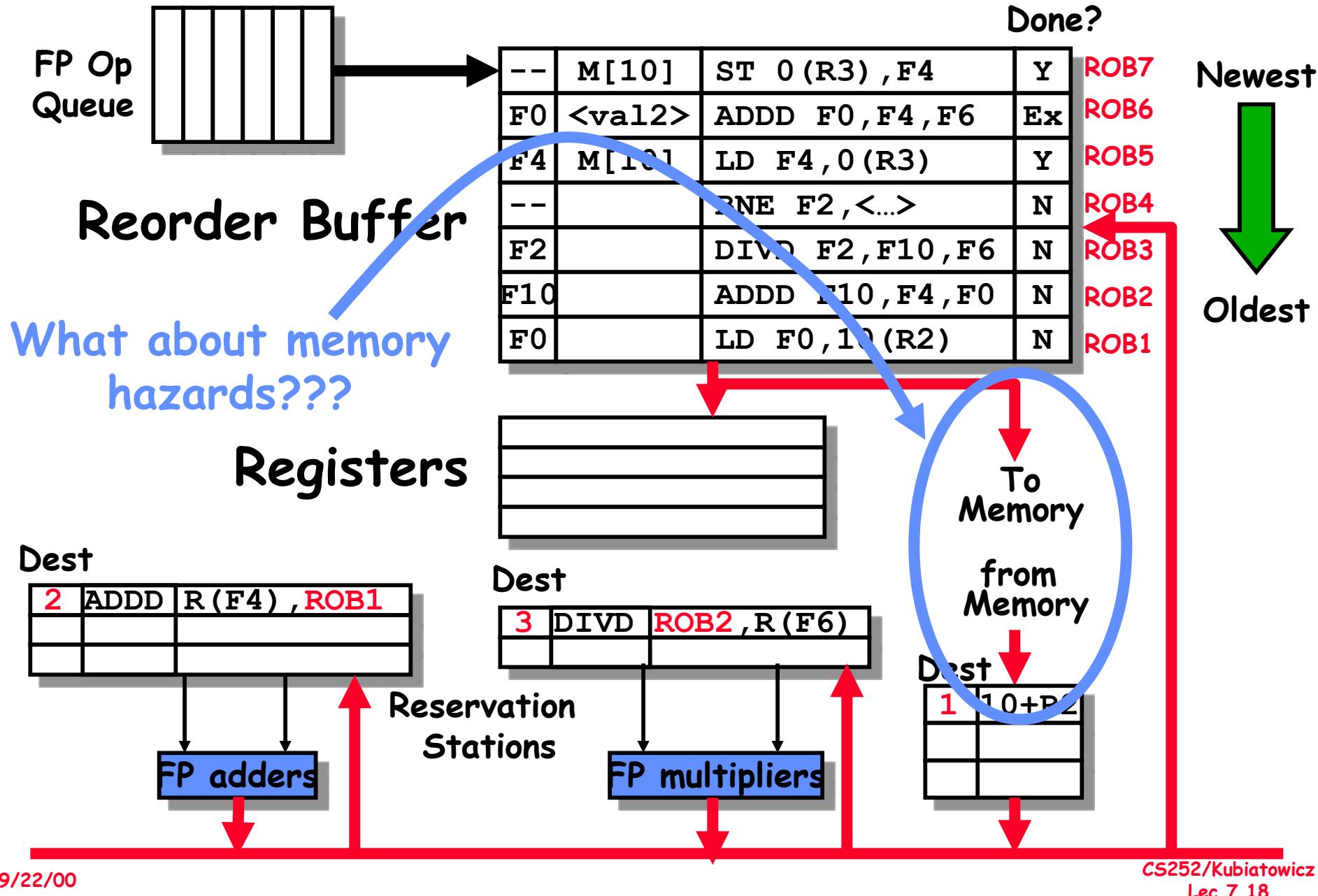
To Memory

from Memory

Dest

1	10+R2

Tomasulo With Reorder buffer:



Memory Disambiguation: Sorting out RAW Hazards in memory

- Question: Given a load that follows a store in program order, are the two related?
 - (Alternatively: is there a RAW hazard between the store and the load)?

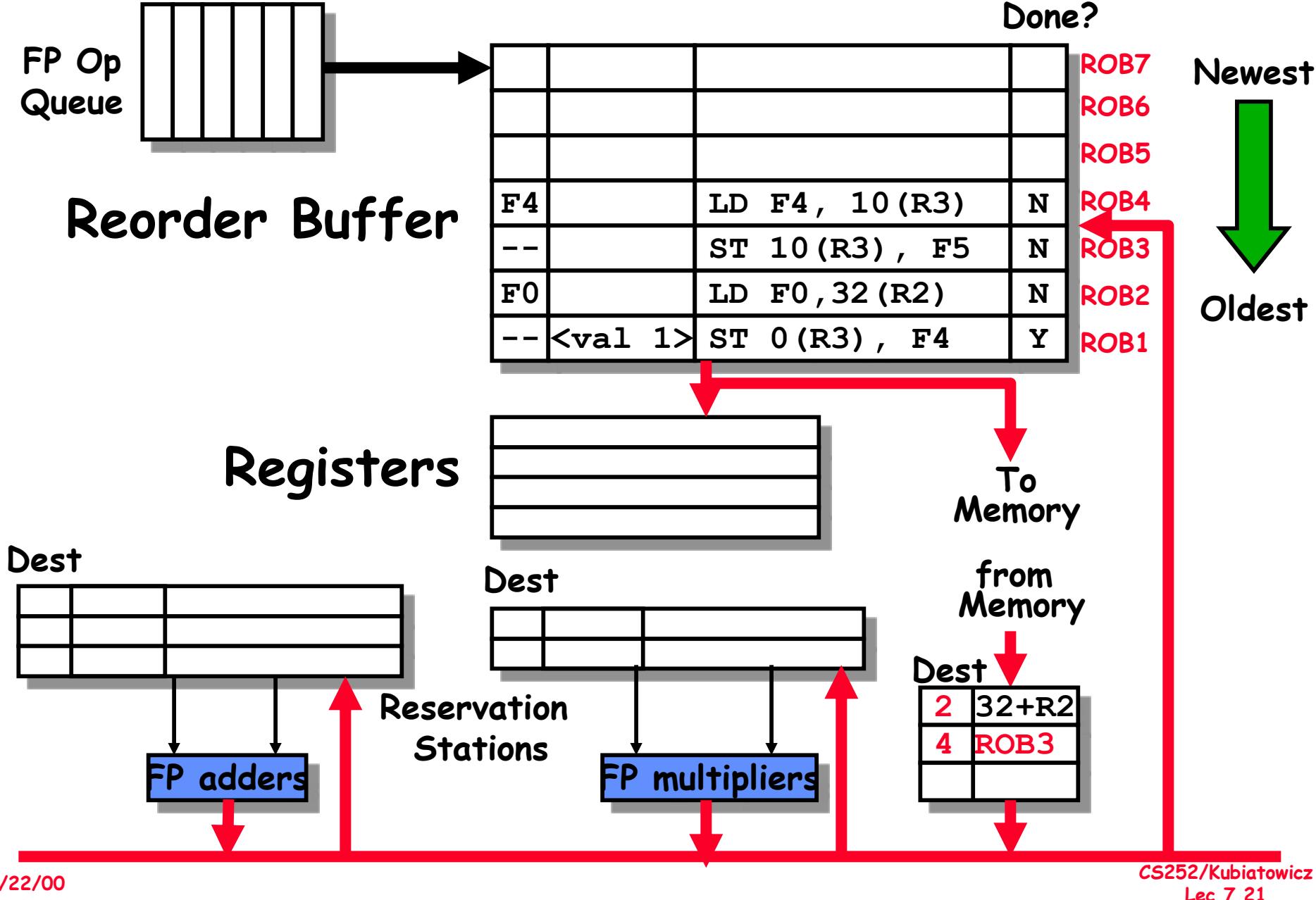
Eg: st 0 (R2) , R5
 ld R6 , 0 (R3)

- Can we go ahead and start the load early?
 - Store address could be delayed for a long time by some calculation that leads to R2 (divide?).
 - We might want to issue/begin execution of both operations in same cycle.
 - Today: Answer is that we are not allowed to start load until we know that address $0(R2) \neq 0(R3)$
 - Next Week: We might guess at whether or not they are dependent (called “**dependence speculation**”) and use reorder buffer to fixup if we are wrong.

Hardware Support for Memory Disambiguation

- Need buffer to keep track of all outstanding stores to memory, in program order.
 - Keep track of address (when becomes available) and value (when becomes available)
 - FIFO ordering: will retire stores from this buffer in program order
- When issuing a load, record current head of store queue (know which stores are ahead of you).
- When have address for load, check store queue:
 - If **any** store prior to load is waiting for its address, stall load.
 - If load address matches earlier store address (associative lookup), then we have a **memory-induced RAW hazard**:
 - » store value available \Rightarrow return value
 - » store value not available \Rightarrow return ROB number of source
 - Otherwise, send out request to memory
- Actual stores commit in order, so no worry about WAR/WAW hazards through memory.

Memory Disambiguation:



Relationship between precise interrupts and speculation:

- Speculation is a form of guessing
 - Branch prediction, data prediction
 - If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
 - This is exactly same as precise exceptions!
- Branch prediction is a very important
 - Need to “take our best shot” at predicting branch direction.
 - If we issue multiple instructions per cycle, lose lots of potential instructions otherwise:
 - » Consider 4 instructions per cycle
 - » If take single cycle to decide on branch, waste from 4 - 7 instruction slots!
- Technique for both precise interrupts/exceptions and speculation: *in-order completion or commit*
 - This is why reorder buffers in all new processors

Administrative

- **Solutions for Prereq Quiz up on web site**
 - Everyone who was going to take it has.
- **Assignment #1 up later today/tomorrow**
 - Usually have 1.5 to 2 weeks to do assignment
- **Computers in the news:**
 - Microprocessor report has some preliminary information on the latest x86 processor (Pentium 4). Will hand out for tonight (as soon as we get the scanner fixed)
 - Intel "crashed" the stock market: down 12 in after-hours trading yesterday.

Explicit Register Renaming

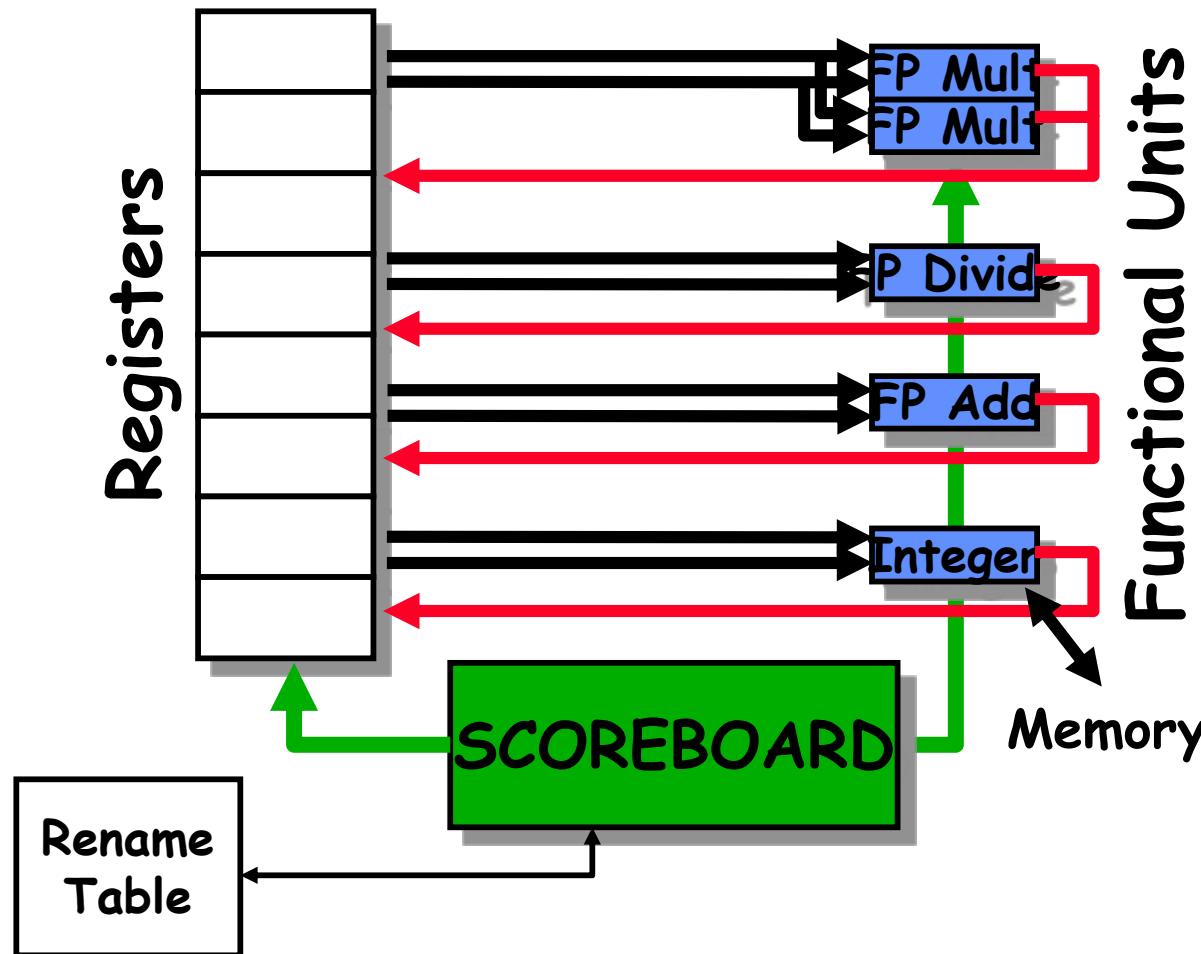
- Make use of a *physical register file* that is larger than number of registers specified by ISA
- Key insight: Allocate a new physical destination register for every instruction that writes
 - Very similar to a compiler transformation called Static Single Assignment (SSA) form — but in hardware!
 - Removes all chance of WAR or WAW hazards
 - Like Tomasulo, good for allowing full out-of-order completion
 - Like hardware-based dynamic compilation?
- Mechanism? Keep a translation table:
 - ISA register \Rightarrow physical register mapping
 - When register written, replace entry with new register from freelist.
 - Physical register becomes free when not used by any active instructions

Advantages of Explicit Renaming

- Decouples *renaming* from *scheduling*:
 - Pipeline can be exactly like “standard” DLX pipeline (perhaps with multiple operations issued per cycle)
 - Or, pipeline could be tomasulo-like or a scoreboard, etc.
 - Standard forwarding or bypassing could be used
- Allows data to be fetched from single register file
 - No need to bypass values from reorder buffer
 - This can be important for balancing pipeline
- Many processors use a variant of this technique:
 - R10000, Alpha 21264, HP PA8000
- Another way to get precise interrupt points:
 - All that needs to be “undone” for precise break point is to undo the table mappings
 - This provides an interesting mix between reorder buffer and future file
 - » Results are written immediately back to register file
 - » Registers names are “freed” in program order (by ROB)

Question:

Can we use explicit register renaming with scoreboard?



Four Stages of Scoreboard Control With Explicit Renaming

- **Issue**—decode instructions & check for structural hazards & allocate new physical register for result
 - Instructions issued in program order (for hazard checking)
 - Don't issue if no free physical registers
 - Don't issue if structural hazard
- **Read operands**—wait until no hazards, read operands
 - All real dependencies (RAW hazards) resolved in this stage, since we wait for instructions to write back data.
- **Execution**—operate on operands
 - The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard
- **Write result**—finish execution
- Note: No checks for WAR or WAW hazards!

Scoreboard With Explicit Renaming

Instruction status:

Instruction	j	k	Read Exec Write			
			Issue	Op	Comp	Result
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	dest		S1	S2	FU	FU	Fj?	Fk?
		Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj
	Int1	No							
	Int2	No							
	Mult1	No							
	Add	No							
	Divide	No							

Register Rename and Result

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
	FU	P0	P2	P4	P6	P8	P10	P12	P30

- **Initialized Rename Table**

Renamed Scoreboard 1

Instruction status:

Instruction	<i>j</i>	<i>k</i>	Read Exec Write			
			<i>Issue</i>	<i>Op</i>	<i>Comp</i>	<i>Result</i>
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	<i>dest</i>		<i>S1</i>	<i>S2</i>	<i>FU</i>	<i>FU</i>	<i>Fj?</i>	<i>Fk?</i>
		<i>Busy</i>	<i>Op</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rj</i>
	Int1	Yes	Load	P32		R2			Yes
	Int2	No							
	Mult1	No							
	Add	No							
	Divide	No							

Register Rename and Result

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
1	<i>FU</i>	P0	P2	P4	P32	P8	P10	P12	P30

- Each instruction allocates free register
- Similar to single-assignment compiler transformation

Renamed Scoreboard 2

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	Yes	Load	P32		R2				Yes
	Int2	Yes	Load	P34		R3				Yes
	Mult1	No								
	Add	No								
	Divide	No								

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	FU	P0	P34	P4	P32	P8	P10	P12		P30

Renamed Scoreboard 3

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3
LD	F2	45+	R3	2	3	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	Yes	Load	P32		R2				Yes
	Int2	Yes	Load	P34		R3				Yes
	Mult1	Yes	Multd	P36	P34	P4	Int2		No	Yes
	Add	No								
	Divide	No								

Register Rename and Result

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU	P36	P34	P4	P32	P8	P10	P12	P30

Renamed Scoreboard 4

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	2	3	4
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	dest	S1	S2	FU	FU	Fj?	Fk?		
		Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	No								
	Int2	Yes	Load	P34		R3			Yes	
	Mult1	Yes	Multd	P36	P34	P4	Int2		No	Yes
	Add	Yes	Sub	P38	P32	P34		Int2	Yes	No
	Divide	No								

Register Rename and Result

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	P36	P34	P4	P32	P38	P10	P12	P30

Renamed Scoreboard 5

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	2	3	4 5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	No								
	Int2	No								
	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
	Add	Yes	Sub	P38	P32	P34			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU	P36	P34	P4	P32	P38	P40	P12	P30

Renamed Scoreboard 6

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	2	3	4 5
MULTD	F0	F2	F4	3	6	
SUBD	F8	F6	F2	4	6	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	No								
	Int2	No								
10	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
2	Add	Yes	Sub	P38	P32	P34			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
6	FU	P36	P34	P4	P32	P38	P40	P12	P30

Renamed Scoreboard 7

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	2	3	4 5
MULTD	F0	F2	F4	3	6	
SUBD	F8	F6	F2	4	6	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	No								
	Int2	No								
9	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
1	Add	Yes	Sub	P38	P32	P34			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	P36	P34	P4	P32	P38	P40	P12		P30

Renamed Scoreboard 8

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	2	3	4 5
MULTD	F0	F2	F4	3	6	
SUBD	F8	F6	F2	4	6	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	No								
	Int2	No								
8	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
0	Add	Yes	Sub	P38	P32	P34			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
8	FU	P36	P34	P4	P32	P38	P40	P12	P30

Renamed Scoreboard 9

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	2	3	4 5
MULTD	F0	F2	F4	3	6	
SUBD	F8	F6	F2	4	6	8 9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	No								
	Int2	No								
7	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
	Add	No								
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
9	FU	P36	P34	P4	P32	P38	P40	P12	P30

Renamed Scoreboard 10

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	2	3	4 5
MULTD	F0	F2	F4	3	6	
SUBD	F8	F6	F2	4	6	8 9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	10		

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	No								
	Int2	No								
6	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
	Add	Yes	Addd	P42	P38	P4			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

WAR Hazard gone!

Register Rename and Result

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
10	FU	P36	P34	P4	P42	P38	P40	P12	P30

- Notice that P32 not listed in Rename Table
 - Still live. Must not be reallocated by accident

Renamed Scoreboard 11

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	2	3	4 5
MULTD	F0	F2	F4	3	6	
SUBD	F8	F6	F2	4	6	8 9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	10	11	

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	No								
	Int2	No								
5	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
2	Add	Yes	Addd	P42	P38	P34			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
11	FU	P36	P34	P4	P42	P38	P40	P12	P30

Renamed Scoreboard 12

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	2	3	4 5
MULTD	F0	F2	F4	3	6	
SUBD	F8	F6	F2	4	6	8 9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	10	11	

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	No								
	Int2	No								
4	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
1	Add	Yes	Addd	P42	P38	P34			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
12	FU	P36	P34	P4	P42	P38	P40	P12	P30

Renamed Scoreboard 13

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	2	3	4 5
MULTD	F0	F2	F4	3	6	
SUBD	F8	F6	F2	4	6	8 9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	10	11	13

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	No								
	Int2	No								
3	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
0	Add	Yes	Addd	P42	P38	P34			Yes	Yes
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
13	FU	P36	P34	P4	P42	P38	P40	P12	P30

Renamed Scoreboard 14

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	2	3	4 5
MULTD	F0	F2	F4	3	6	
SUBD	F8	F6	F2	4	6	8 9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	10	11	13 14

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	No								
	Int2	No								
2	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
	Add	No								
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
14	FU	P36	P34	P4	P42	P38	P40	P12	P30

Renamed Scoreboard 15

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	2	3	4 5
MULTD	F0	F2	F4	3	6	
SUBD	F8	F6	F2	4	6	8 9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	10	11	13 14

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	No								
	Int2	No								
1	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
	Add	No								
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
15	FU	P36	P34	P4	P42	P38	P40	P12		P30

Renamed Scoreboard 16

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	2	3	4 5
MULTD	F0	F2	F4	3	6	16
SUBD	F8	F6	F2	4	6	8 9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	10	11	13 14

Functional unit status:

Time	Name	Busy	Op	dest	S1	S2	FU	FU	Fj?	Fk?
				Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	No								
	Int2	No								
0	Mult1	Yes	Multd	P36	P34	P4			Yes	Yes
	Add	No								
	Divide	Yes	Divd	P40	P36	P32	Mult1		No	Yes

Register Rename and Result

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
16	FU	P36	P34	P4	P42	P38	P40	P12	P30

Renamed Scoreboard 17

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	2	3	4 5
MULTD	F0	F2	F4	3	6	16 17
SUBD	F8	F6	F2	4	6	8 9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	10	11	13 14

Functional unit status:

Time	Name	dest	S1	S2	FU	FU	Fj?	Fk?		
		Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Int1	No								
	Int2	No								
	Mult1	No								
	Add	No								
	Divide	Yes	Divd	P40	P36	P32	Mult1		Yes	Yes

Register Rename and Result

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
17	FU	P36	P34	P4	P42	P38	P40	P12	P30

Renamed Scoreboard 18

Instruction status:

Instruction	j	k	Issue	Read	Exec	Write
				Op	Comp	Result
LD	F6	34+	R2	1	2	3 4
LD	F2	45+	R3	2	3	4 5
MULTD	F0	F2	F4	3	6	16 17
SUBD	F8	F6	F2	4	6	8 9
DIVD	F10	F0	F6	5	18	
ADDD	F6	F8	F2	10	11	13 14

Functional unit status:

Time	Name	Busy	dest	S1	S2	FU	FU	Fj?	Fk?
			Op	Fi	Fj	Fk	Qj	Qk	Rj
	Int1	No							
	Int2	No							
	Mult1	No							
	Add	No							
40	Divide	Yes	Divd	P40	P36	P32	Mult1	Yes	Yes

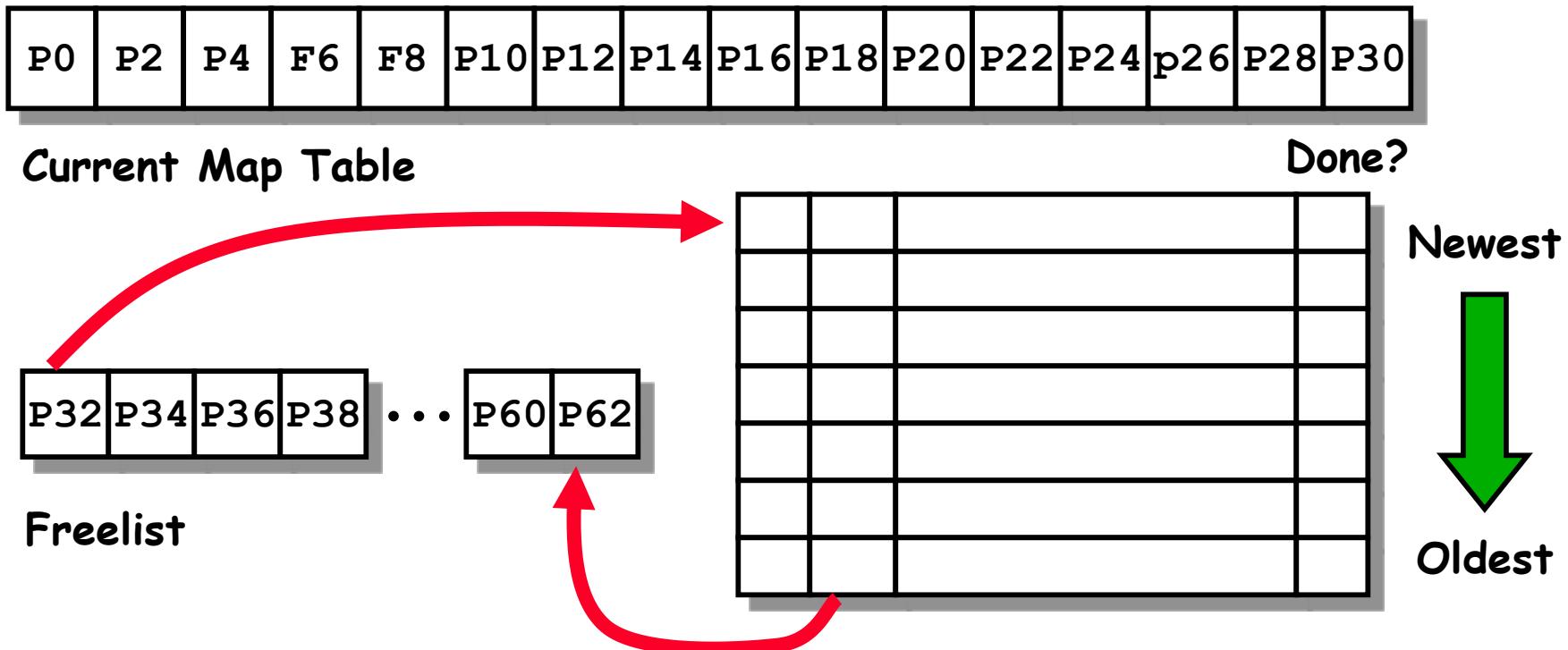
Register Rename and Result

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
18	FU	P36	P34	P4	P42	P38	P40	P12	P30

Explicit Renaming Support Includes:

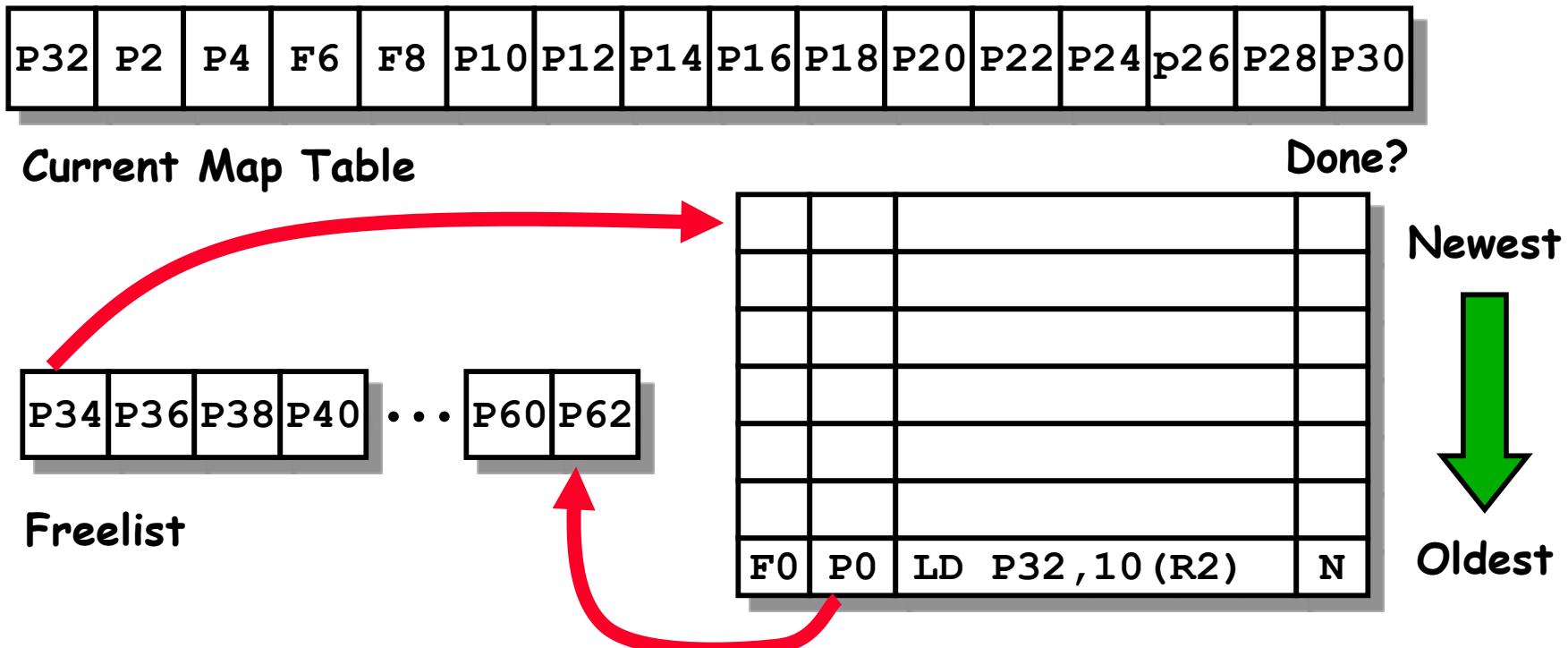
- Rapid access to a table of translations
- A physical register file that has more registers than specified by the ISA
- Ability to figure out which physical registers are free.
 - No free registers \Rightarrow stall on issue
- Thus, register renaming doesn't require reservation stations. However:
 - Many modern architectures use explicit register renaming + Tomasulo-like reservation stations to control execution.
- Two Questions:
 - How do we manage the “free list”?
 - How does Explicit Register Renaming mix with Precise Interrupts?

Explicit register renaming: (R1000 Style)



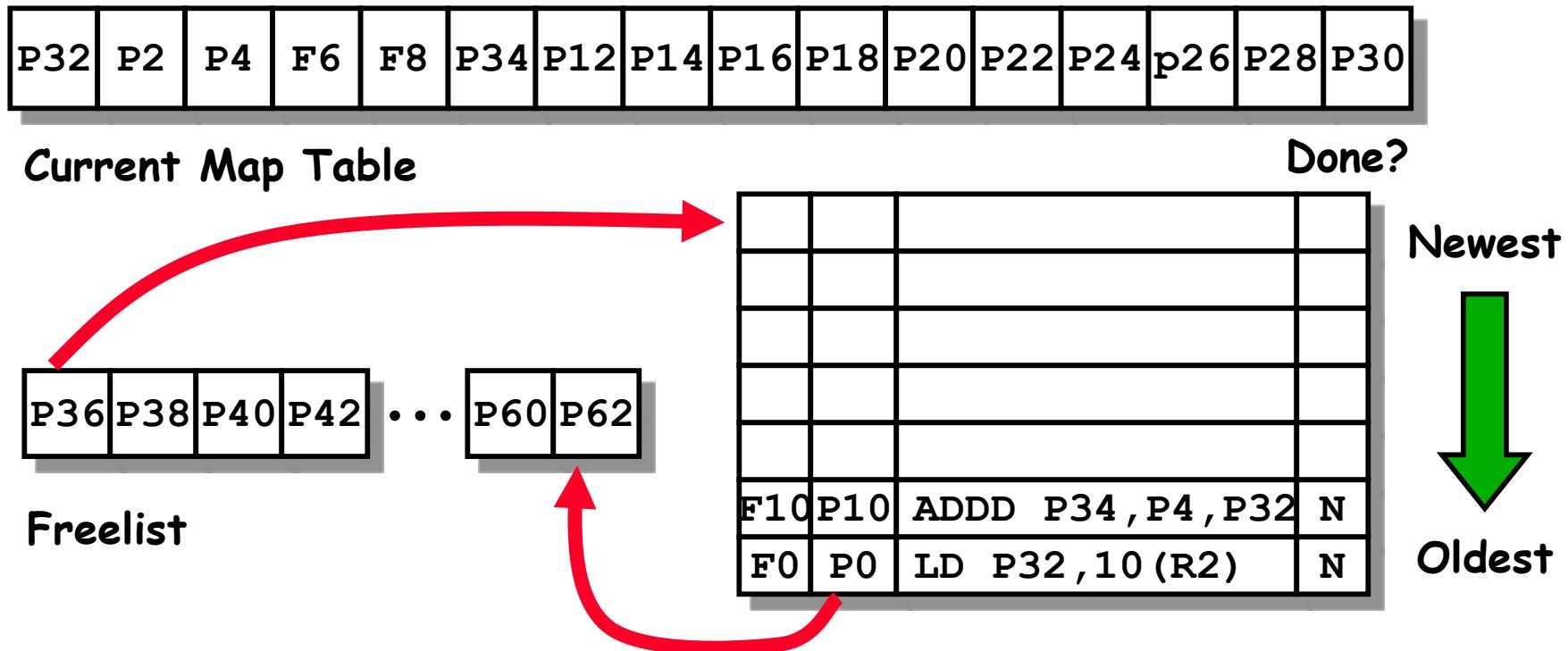
- Physical register file larger than ISA register file
- On issue, each instruction that modifies a register is allocated new physical register from freelist

Explicit register renaming: (R1000 Style)

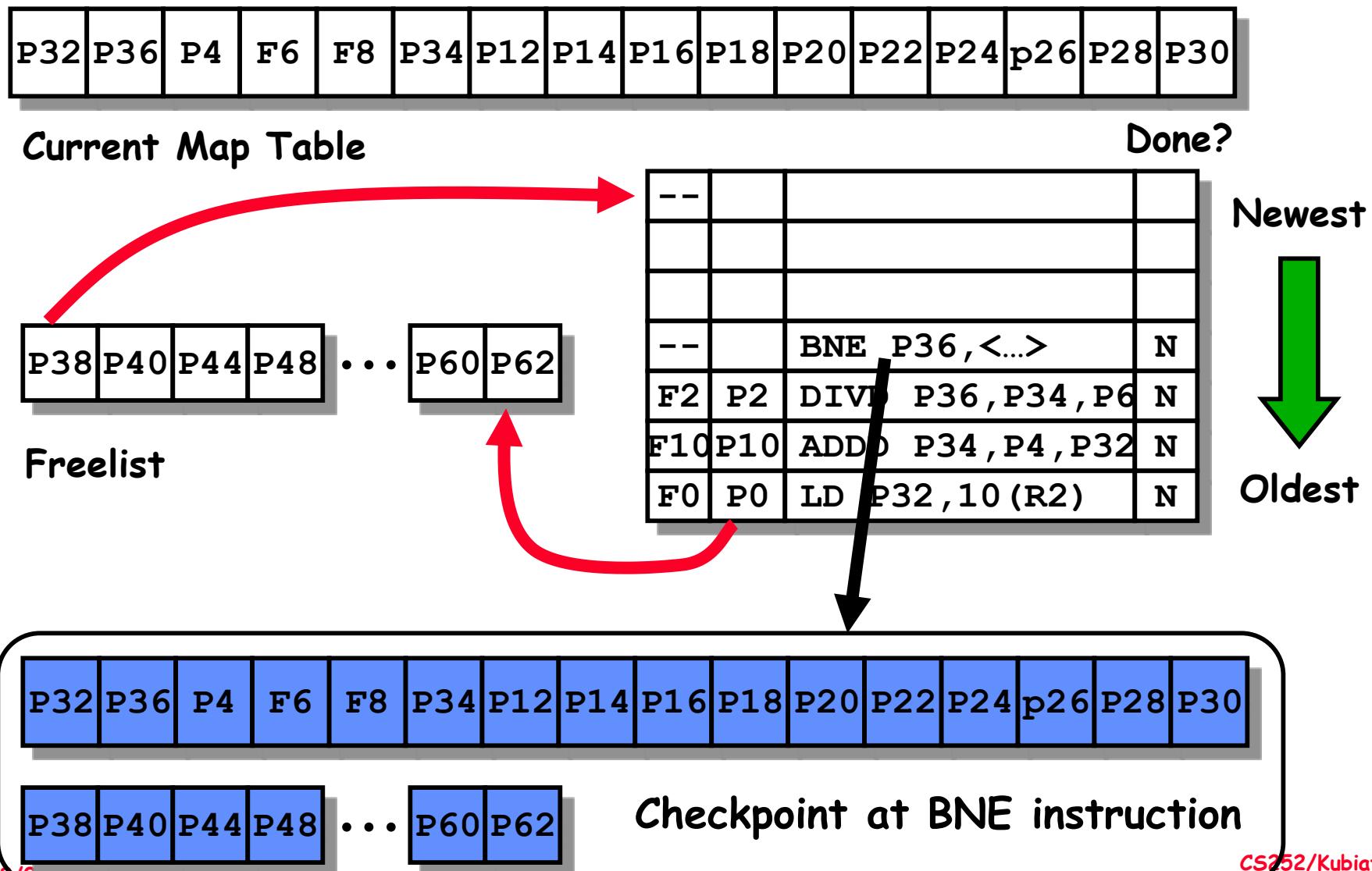


- Note that physical register P0 is “dead” (or not “live”) past the point of this load.
 - When we go to commit the load, we free up

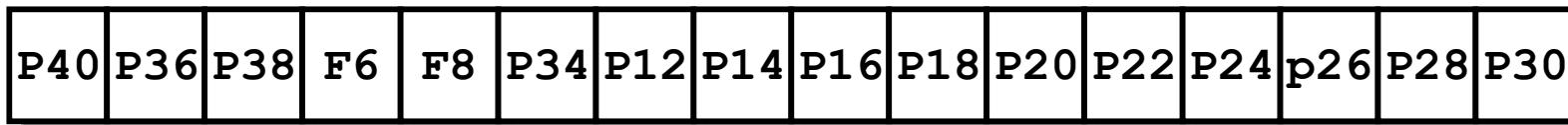
Explicit register renaming: (R1000 Style)



Explicit register renaming: (R1000 Style)



Explicit register renaming: (R1000 Style)



Current Map Table

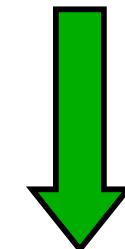


Freelist

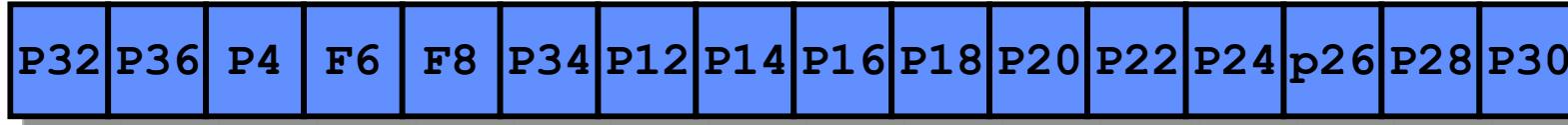
--		ST 0 (R3) , P40	Y
F0	P32	ADDD P40 , P38 , P6	Y
F4	P4	LD P38 , 0 (R3)	Y
--		BNE P36 , <...>	N
F2	P2	DIVD P36 , P34 , P6	N
F10	P10	ADDD P34 , P4 , P32	y
F0	P0	LD P32 , 10 (R2)	y

Done?

Newest

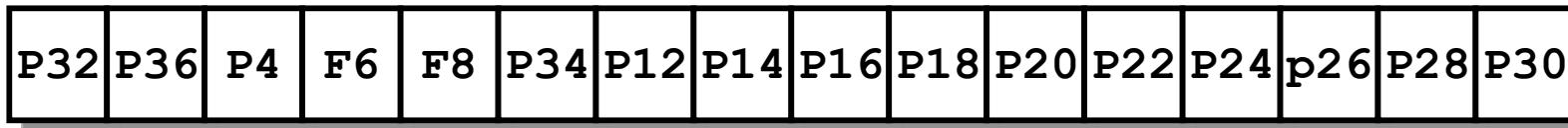


Oldest

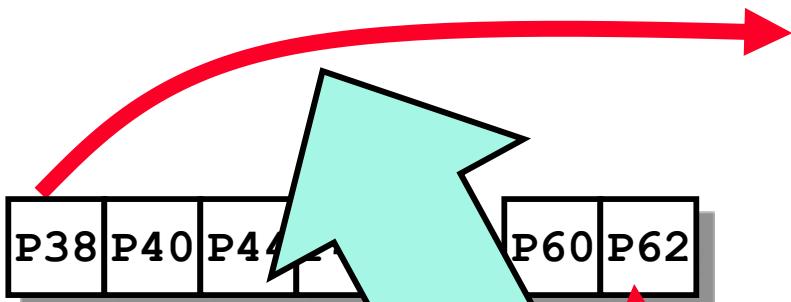


Checkpoint at BNE instruction

Explicit register renaming: (R1000 Style)



Current Map Table



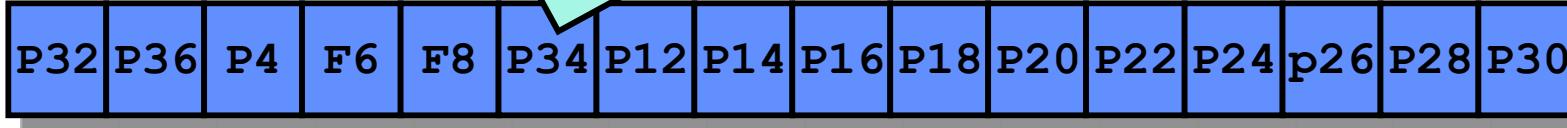
Freelist

A table labeled 'Done?' representing the current map table. The table has 16 columns and 10 rows. The last row is highlighted in green and contains the following data: F2, P2, DIVD, P36, P34, P6, N. The row above it is also highlighted in green and contains: F10, P10, ADDD, P34, P4, P32, y. The row below it is also highlighted in green and contains: F0, P0, LD, P32, 10 (R2), y. A green arrow points from the 'Freelist' to the 'Done?' table, indicating the flow of information.

F2	P2	DIVD	P36	P34	P6	N										
F10	P10	ADDD	P34	P4	P32	y										
F0	P0	LD	P32	10	(R2)	y										

Newest
↓
Oldest

Speculation error fixed by restoring map table and freelist



Checkpoint at BNE instruction

Summary #1

- Dynamic hardware schemes can unroll loops dynamically in hardware
 - Form of limited dataflow
- Reorder Buffer
 - In-order issue, Out-of-order execution, In-order commit
 - Holds results until they can be committed in order
 - » Serves as source of info until instructions committed
 - Provides support for precise exceptions/Speculation: simply throw out instructions later than expected instruction.
- Memory Disambiguation:
 - Tracking of RAW hazards through memory
 - Keep program-order queue of stores
When have address for load, check store queue:
 - » If **any** store prior to load is waiting for its address, stall load.
 - » If load address matches earlier store address (associative lookup), then we have a **memory-induced RAW hazard**:
 - » Otherwise, send out request to memory

Summary #2

- **Explicit Renaming:** more physical registers than needed by ISA.
 - Separates renaming from scheduling
 - » Opens up lots of options for resolving RAW hazards
 - Rename table: tracks current association between architectural registers and physical registers
 - Potentially complicated rename table management

The Microarchitecture of Superscalar Processors

JAMES E. SMITH, MEMBER, IEEE, AND GURINDAR S. SOHI, SENIOR MEMBER, IEEE

Invited Paper

Superscalar processing is the latest in a long series of innovations aimed at producing ever-faster microprocessors. By exploiting instruction-level parallelism, superscalar processors are capable of executing more than one instruction in a clock cycle. This paper discusses the microarchitecture of superscalar processors. We begin with a discussion of the general problem solved by superscalar processors: converting an ostensibly sequential program into a more parallel one. The principles underlying this process, and the constraints that must be met, are discussed. The paper then provides a description of the specific implementation techniques used in the important phases of superscalar processing. The major phases include: 1) instruction fetching and conditional branch processing, 2) the determination of data dependences involving register values, 3) the initiation, or issuing, of instructions for parallel execution, 4) the communication of data values through memory via loads and stores, and 5) committing the process state in correct order so that precise interrupts can be supported. Examples of recent superscalar microprocessors, the MIPS R10000, the DEC 21164, and the AMD K5 are used to illustrate a variety of superscalar methods.

I. INTRODUCTION

Superscalar processing, the ability to initiate multiple instructions during the same clock cycle, is the latest in a long series of architectural innovations aimed at producing ever faster microprocessors. Introduced at the beginning of this decade, superscalar microprocessors are now being designed and produced by all the microprocessor vendors for high-end products. Although viewed by many as an extension of the reduced instruction set computer (RISC) movement of the 1980's, superscalar implementations are in fact heading toward increasing complexity. And superscalar methods have been applied to a spectrum of instruction sets, ranging from the DEC Alpha, the "newest" RISC instruction set, to the decidedly non-RISC Intel x86 instruction set.

Manuscript received February 9, 1995; revised August 23, 1995. This work was supported in part by NSF Grants CCR-9303030 and MIP-9505853, in part by ONR Grant N00014-93-1-0465, in part by the University of Wisconsin Graduate School, and in part by Intel Corporation.

J. E. Smith is with the Department of Electrical and Computer Engineering, The University of Wisconsin, Madison, WI 53706 USA.

G. S. Sohi is with the Computer Sciences Department, The University of Wisconsin, Madison, WI 53706 USA.

IEEE Log Number 9415183.

A typical superscalar processor fetches and decodes the incoming instruction stream several instructions at a time. As part of the instruction fetching process, the outcomes of conditional branch instructions are usually predicted in advance to ensure an uninterrupted stream of instructions. The incoming instruction stream is then analyzed for data dependences, and instructions are distributed to functional units, often according to instruction type. Next, instructions are initiated for execution in parallel, based primarily on the availability of operand data, rather than their original program sequence. This important feature, present in many superscalar implementations, is referred to as *dynamic instruction scheduling*. Upon completion, instruction results are resequenced so that they can be used to update the process state in the correct (original) program order in the event that an interrupt condition occurs. Because individual instructions are the entities being executed in parallel, superscalar processors exploit what is referred to as *instruction level parallelism* (ILP).

A. Historical Perspective

Instruction level parallelism in the form of pipelining has been around for decades. A pipeline acts like an assembly line with instructions being processed in phases as they pass down the pipeline. With simple pipelining, only one instruction at a time is initiated into the pipeline, but multiple instructions may be in some phase of execution concurrently.

Pipelining was initially developed in the late 1950's [8] and became a mainstay of large scale computers during the 1960's. The CDC 6600 [61] used a degree of pipelining, but achieved most of its ILP through parallel functional units. Although it was capable of sustained execution of only a single instruction per cycle, the 6600's instruction set, parallel processing units, and dynamic instruction scheduling are similar to the superscalar microprocessors of today. Another remarkable processor of the 1960's was the IBM 360/91 [3]. The 360/91 was heavily pipelined, and provided a dynamic instruction issuing mechanism, known as *Tomasulo's algorithm* [63] after its inventor. As with the CDC 6600, the IBM 360/91 could sustain only one

instruction per cycle and was not superscalar, but the strong influence of Tomasulo's algorithm is evident in many of today's superscalar processors.

The pipeline initiation rate remained at one instruction per cycle for many years and was often perceived to be a serious practical bottleneck. Meanwhile other avenues for improving performance via parallelism were developed, such as vector processing [28], [49] and multiprocessing [5], [6]. Although some processors capable of multiple instruction initiation were considered during the 1960's and 1970's [50], [62], none were delivered to the market. Then, in the mid-to-late 1980's, superscalar processors began to appear [21], [43], [54]. By initiating more than one instruction at a time into multiple pipelines, superscalar processors break the single-instruction-per-cycle bottleneck. In the years since its introduction, the superscalar approach has become the standard method for implementing high performance microprocessors.

B. The Instruction Processing Model

Because hardware and software evolve, it is rare for a processor architect to start with a clean slate; most processor designs inherit a legacy from their predecessors. Modern superscalar processors are no different. A major component of this legacy is *binary compatibility*, the ability to execute a machine program written for an earlier generation processor.

When the very first computers were developed, each had its own instruction set that reflected specific hardware constraints and design decisions at the time of the instruction set's development. Then, software was developed for each instruction set. It did not take long, however, until it became apparent that there were significant advantages to designing instruction sets that were compatible with previous generations and with different models of the same generation [2]. For a number of very practical reasons, the instruction set architecture, or binary machine language level, was chosen as the level for maintaining software compatibility.

The sequencing model inherent in instruction sets and program binaries, the *sequential execution model*, closely resembles the way processors were implemented many years ago. In the sequential execution model, a program counter is used to fetch a single instruction from memory. The instruction is then executed—in the process, it may load or store data to main memory and operate on registers held in the processor. Upon completing the execution of the instruction, the processor uses an incremented program counter to fetch the next instruction, with sequential instruction processing occasionally being redirected by a conditional branch or jump.

Should the execution of the program need to be interrupted and restarted later, for example in case of a page fault or other exception condition, the state of the machine needs to be captured. The sequential execution model has led naturally to the concept of a *precise state*. At the time of an interrupt, a precise state of the machine (architecturally visible registers and memory) is the state

that would be present if the sequential execution model was strictly followed and processing was stopped precisely at the interrupted instruction. Restart could then be implemented by simply resuming instruction processing with the interrupted instruction.

Today, a computer designer is usually faced with maintaining binary compatibility, i.e., maintaining instruction set compatibility *and* a sequential execution model (which typically implies precise interrupts).¹ For high performance, however, superscalar processor implementations deviate radically from sequential execution—much has to be done in parallel. As a result, the program binary nowadays should be viewed as a specification of *what* has to be done, not *how* it is done in reality. A modern superscalar microprocessor takes the sequential specification as embodied in the program binary and removes much of the nonessential sequentiality to turn the program into a parallel, higher-performance version, yet the processor retains the outward appearance of sequential execution.

C. Elements of High Performance Processing

Simply stated, achieving higher performance means processing a given program in a smaller amount of time. Each individual instruction takes some time to fetch and execute; this time is the instruction's *latency*. To reduce the time to execute a sequence of instructions (e.g., a program), one can: 1) reduce individual instruction latencies, or 2) execute more instructions in parallel. Because superscalar processor implementations are distinguished by the latter (while adequate attention is also paid to the former), we will concentrate on the latter method in this paper. Nevertheless, a significant challenge in superscalar design is to not *increase* instruction latencies due to increased hardware complexity brought about by the drive for enhanced parallelism.

Parallel instruction processing requires: the determination of the dependence relationships between instructions, adequate hardware resources to execute multiple operations in parallel, strategies to determine when an operation is ready for execution, and techniques to pass values from one operation to another. When the effects of instructions are committed, and the visible state of the machine updated, the appearance of sequential execution must be maintained. More precisely, in hardware terms, this means a superscalar processor implements:

- 1) Instruction fetch strategies that simultaneously fetch multiple instructions, often by predicting the outcomes of, and fetching beyond, conditional branch instructions.
- 2) Methods for determining true dependences involving register values, and mechanisms for communicating these values to where they are needed during execution.
- 3) Methods for initiating, or *issuing*, multiple instructions in parallel.

¹Some recently developed instruction sets relax the strictly sequential execution model by allowing a few exception conditions to result in an “imprecise” saved state where the program counter is inconsistent with respect to the saved registers and memory values.

- 4) Resources for parallel execution of many instructions, including multiple pipelined functional units and memory hierarchies capable of simultaneously servicing multiple memory references.
- 5) Methods for communicating data values through memory via load and store instructions, and memory interfaces that allow for the dynamic and often unpredictable performance behavior of memory hierarchies. These interfaces must be well matched with the instruction execution strategies.
- 6) Methods for committing the process state in correct order; these mechanisms maintain an outward appearance of sequential execution.

Although we will discuss the above items separately, in reality they cannot be completely separated—nor should they be. In good superscalar designs they are often integrated in a cohesive, almost seamless, manner.

D. Paper Overview

In Section II, we discuss the general problem solved by superscalar processors: converting an ostensibly sequential program into a parallel one. This is followed in Section III with a description of specific techniques used in typical superscalar microprocessors. Section IV focuses on three recent superscalar processors that illustrate the spectrum of current techniques. Section V presents conclusions and discusses future directions for instruction level parallelism.

II. PROGRAM REPRESENTATION, DEPENDENCES AND PARALLEL EXECUTION

An application begins as a high level language program; it is then compiled into the *static machine level program*, or the *program binary*. The static program in essence describes a set of executions, each corresponding to a particular set of data that is given to the program. Implicit in the static program is the sequencing model, the order in which the instructions are to be executed. Fig. 1 shows the *assembly code* for a high level language program fragment (The assembly code is the human-readable version of the machine code). We will use this code fragment as a working example.

As a static program executes with a specific set of input data, the sequence of executed instructions forms a *dynamic* instruction stream. As long as instructions to be executed are consecutive, static instructions can be entered into the dynamic sequence simply by *incrementing* the program counter, which points to the next instruction to be executed. When there is a conditional branch or jump, however, the program counter may be *updated* to a nonconsecutive address. An instruction is said to be *control dependent* on its preceding dynamic instruction(s), because the flow of program control must pass through preceding instructions first. The two methods of modifying the program counter—incrementing and updating—result in two types of control dependences (though typically when people talk about control dependences, they tend to ignore the former).

```

for (i=0; i<last; i++) {
    if (a[i] > a[i+1]) {
        temp = a[i];
        a[i] = a[i+1];
        a[i+1] = temp;
        change++;
    }
}

```

(a)

```

L2:
move  r3,r7      #r3->a[i]
lw    r8,(r3)    #load a[i]
add  r3,r3,4    #r3->a[i+1]
lw    r9,(r3)    #load a[i+1]
ble  r8,r9,L3  #branch a[i]>a[i+1]

move  r3,r7      #r3->a[i]
sw    r9,(r3)    #store a[i]
add  r3,r3,4    #r3->a[i+1]
sw    r8,(r3)    #store a[i+1]
add  r5,r5,1    #change++
L3:
add  r6,r6,1    #i++
add  r7,r7,4    #r4->a[i]
blt  r6,r4,L2  #branch i<last

```

(b)

Fig. 1. (a) A high level language program written in C; (b) its compilation into a static assembly language program (unoptimized). This code is a part of a sort routine; adjacent values in array $a[]$ are compared and switched if $a[i] > a[i+1]$. The variable $change$ keeps track of the number of switches (if $change = 0$ at the end of a pass through the array, then the array is sorted.)

The first step in increasing instruction level parallelism is to overcome control dependences. Control dependences due to an incrementing program counter are the simplest, and we deal with them first. One can view the static program as a collection of *basic blocks*, where a basic block is a contiguous block of instructions, with a single entry point and a single exit point [1]. In the assembly code of Fig. 1, there are three basic blocks. The first basic block consists of the five instructions between the label L2 and the *ble* instruction, inclusive, the second basic block consists the five instructions between the *ble* instruction, exclusive, and the label L3, and the third basic block consists of the three instructions between the label L3 and the *blt* instruction, inclusive. Once a basic block has been entered by the instruction fetcher, it is known that all the instructions in the basic block will be executed eventually. Therefore, any sequence of instructions in a basic block can be initiated into a conceptual *window of execution*, en masse. We consider the window of execution to be the full set of instructions that may be simultaneously considered for parallel execution. Once instructions have been initiated into this window of execution, they are free to execute in parallel, subject only to data dependence constraints (which we will discuss shortly).

Within individual basic blocks there is some parallelism, but to get more parallelism, control dependences due to updates of the program counter, especially due to condi-

tional branches, have to be overcome. A method for doing this is to *predict* the outcome of a conditional branch and *speculatively* fetch and execute instructions from the predicted path. Instructions from the predicted path are entered into the window of execution. If the prediction is later found to be correct, then the speculative status of the instructions is removed, and their effect on the state is the same as any other instruction. If the prediction is later found to be incorrect, the speculative execution was incorrect, and recovery actions must be initiated so that the architectural process state is not incorrectly modified. We will discuss branch prediction and speculative execution in more detail in Section III. In our example of Fig. 1, the branch instruction (*ble*) creates a control dependence. To overcome this dependence, the branch could be predicted as not taken, for example, with instructions between the branch and the label L3 being executed speculatively.

Instructions that have been placed in the window of execution may begin execution subject to *data dependence* constraints. Data dependences occur among instructions because the instructions may access (read or write) the same storage (a register or memory) location. When instructions reference the same storage location, a *hazard* is said to exist—i.e., there is the possibility of incorrect operation unless some steps are taken to make sure the storage location is accessed in correct order. Ideally, instructions can be executed subject only to *true dependence* constraints. These true dependences appear as *read-after-write* (RAW) hazards, because the consuming instruction can only *read* the value created by the producing instruction has *written* it.

It is also possible to have artificial dependences, and in the process of executing a program, these artificial dependences have to be overcome to increase the available level of parallelism. These artificial dependences result from *write-after-read* (WAR), and *write-after-write* (WAW) hazards. A WAR hazard occurs when an instruction needs to write a new value into storage location, but must wait until all preceding instructions needing to read the old value have done so. A WAW hazard occurs when multiple instructions update the same storage location; it must appear that these updates occur in proper sequence. Artificial dependences can be caused in a number of ways, for example: by unoptimized code, by limited register storage, by the desire to economize on main memory storage, and by loops where an instruction can cause a hazard with itself.

Fig. 2 shows some of the data hazards that are present in a segment of our working example. The *move* instruction produces a value in register r3 that is used both by the first *lw* and *add* instructions. This is a RAW hazard because a true dependence is present. The *add* instruction also creates a value which is bound to register r3. Accordingly, there is a WAW hazard involving the *move* and the *add* instructions. A dynamic execution must ensure that accesses to r3 made by instructions that occur after the *add* in the program access the value bound to r3 by the *add* instruction and not the *move* instruction. Likewise, there is a WAR hazard involving the first *lw* and the *add* instructions. Execution must ensure that the value of r3 used in the *lw* instruction

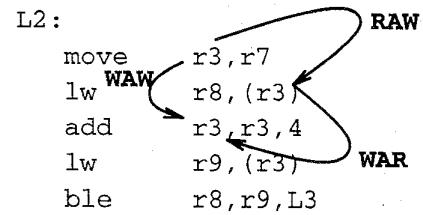


Fig. 2. Example of data hazards involving registers.

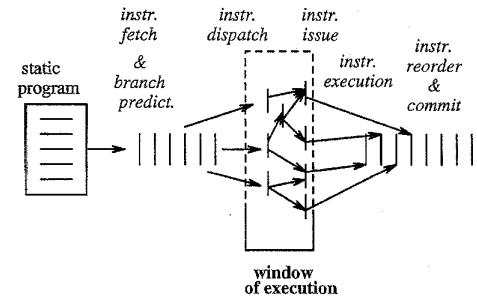


Fig. 3. A conceptual figure of superscalar execution. Processing phases are listed across the top of the figure.

is the value created by the *move* instruction, and not the value created by the *add* instruction.

After resolving control dependences and artificial dependences, instructions are issued and begin execution in parallel. In essence, the hardware forms a *parallel execution schedule*. This schedule takes into account necessary constraints, such as the true dependences and hardware resource constraints of the functional units and data paths.

A parallel execution schedule often means that instructions complete execution in an order different from that dictated by the sequential execution model. Moreover, speculative execution means that some instructions may complete execution when they would not have executed at all had the sequential model been followed (i.e., when speculated instructions follow an incorrectly predicted branch.) Consequently, the architectural storage, (the storage that is outwardly visible) cannot be updated immediately when instructions complete execution. Rather, the results of an instruction must be held in a temporary status until the architectural state can be updated. Meanwhile, to maintain high performance, these results must be usable by dependent instructions. Eventually, when it is determined that the sequential model would have executed an instruction, its temporary results are made permanent by updating the architectural state. This process is typically called *committing* or *retiring* the instruction.

To summarize and to set the stage for the next section, Fig. 3 illustrates the parallel execution method used in most superscalar processors. Instructions begin in a static program. The instruction fetch process, including branch prediction, is used to form a stream of dynamic instructions. These instructions are inspected for dependences with many artificial dependences being removed. The instructions are then dispatched into the window of execution. In Fig. 3,

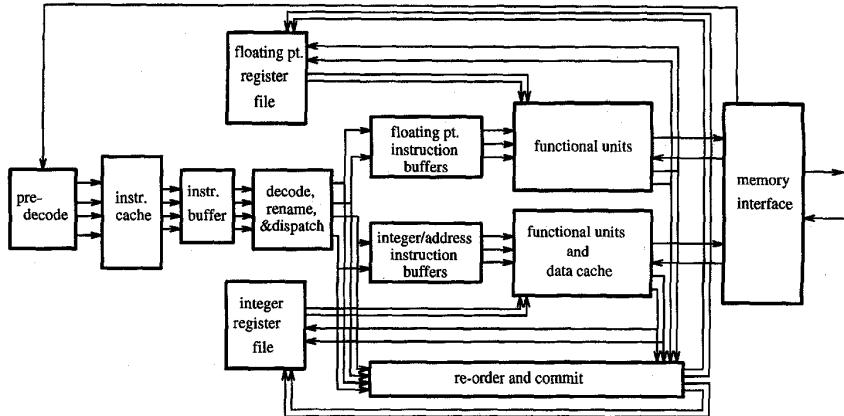


Fig. 4. Organization of a superscalar processor. Multiple paths connecting units are used to illustrate a typical level of parallelism. For example, four instructions can be fetched in parallel, two integer/address instruction can issue in parallel, two floating point instructions can complete in parallel, etc.

the instructions within the window of execution are no longer represented in sequential order, but are partially ordered by their true data dependences. Instructions issue from the window in an order determined by the true data dependences and hardware resource availability. Finally, after execution, instructions are conceptually put back into sequential program order as they are retired and their results update the architected process state.

III. THE MICROARCHITECTURE OF A TYPICAL SUPERSCALAR PROCESSOR

Fig. 4 illustrates the microarchitecture, or hardware organization, of a typical superscalar processor. The major parts of the microarchitecture are: instruction fetch and branch prediction, decode and register dependence analysis, issue and execution, memory operation analysis and execution, and instruction reorder and commit. These phases are listed more-or-less in the same order as instructions flow through them; they will each be discussed in following subsections.

Keep in mind that underlying this organization there is a pipelined implementation where specific pipeline stages may or may not be aligned with the major phases of superscalar execution. The underlying pipeline stages are to some extent a lower-level logic design issue, depending on how much work is done in each pipeline stage. This will affect the clock period and causes some very important design tradeoffs regarding the degree of pipelining and the width of parallel instruction issue. However, we will not discuss these issues in detail here; discussions of these types of tradeoffs can be found in [13], [34], [38].

A. Instruction Fetching and Branch Prediction

The instruction fetch phase of superscalar processing supplies instructions to the rest of the processing pipeline. An *instruction cache*, which is a small memory containing recently used instructions [52], is used in almost all current processors, superscalar or not, to reduce the latency and

increase the bandwidth of the instruction fetch process. An instruction cache is organized into *blocks* or *lines* containing several consecutive instructions; the program counter is used to search the cache contents associatively to determine if the instruction being addressed is present in one of the cache lines. If so, there is a cache *hit*; if not, there is a *miss* and the line containing the instruction is brought in from main memory to be placed in the cache.

For a superscalar implementation to sustain the execution of multiple instructions per cycle, the fetch phase must be able to fetch multiple instructions per cycle from the cache memory. To support this high instruction fetch bandwidth, it has become almost mandatory to separate the instruction cache from the data cache (although the PowerPC 601 [41] provides an interesting counter example).

The number of instructions fetched per cycle should at least match the peak instruction decode and execution rate and is usually somewhat higher. The extra margin of instruction fetch bandwidth allows for instruction cache misses and for situations where fewer than the maximum number of instructions can be fetched. For example, if a branch instruction transfers control to an instruction in the middle of a cache line, then only the remaining portion of the cache line contains useful instructions. Some of the superscalar processors have taken special steps to allow wider fetches in this case, for example by fetching from two adjacent cache lines simultaneously [20], [65]. A discussion of a number of alternatives for high bandwidth instruction fetching appears in [11].

To help smooth out the instruction fetch irregularities caused by cache misses and branches, there is often an instruction buffer (shown in Fig. 4) to hold a number of fetched instructions. Using this buffer, the fetch mechanism can build up a “stockpile” to carry the processor through periods when instruction fetching is stalled or restricted.

The default instruction fetching method is to increment the program counter by the number of instructions fetched, and use the incremented program counter to fetch the next

block of instructions. In case of branch instructions which redirect the flow of control, however, the fetch mechanism must be redirected to fetch instructions from the branch target. Because of delays that can occur during the process of this redirection, the handling of branch instructions is critical to good performance in superscalar processors. Processing of conditional branch instructions can be broken down into the following parts:

- 1) recognizing that an instruction is a conditional branch,
- 2) determining the branch outcome (taken or not taken),
- 3) computing the branch target, and
- 4) transferring control by redirecting instruction fetch (in the case of a taken branch).

Specific techniques are useful for handling each of the above.

1) *Recognizing Conditional Branches*: This seems too obvious to mention, but it is the first step toward fast branch processing. Identifying all instruction types, not just branches, can be sped up if some instruction decode information is held in the instruction cache along with the instructions. These extra bits allow very simple logic to identify the basic instruction types.² Consequently, there is often predecode logic prior to the instruction cache (see Fig. 4) which generates predecode bits and stores them alongside the instructions as they are placed in the instruction cache.

2) *Determining the Branch Outcome*: Often, when a branch instruction is fetched, the data upon which the branch decision must be made is not yet available. That is, there is a dependence between the branch instruction and a preceding, uncompleted instruction. Rather than wait, the outcome of a conditional branch can be predicted using one of several types of branch prediction methods [35], [40], [44], [53], [66]. Some predictors use static information, i.e., information that can be determined from the static binary (either explicitly present or put there by the compiler.) For example, certain opcode types might more often result in taken branches than others, or a backward branch direction (e.g., when forming loops) might be more often taken, or the compiler might be able to set a flag in an instruction to indicate the most likely branch direction based on knowledge of the high level language program. Profiling information—program execution statistics collected during a previous run of the program—can also be used by the compiler as an aid for static branch prediction [18].

Other predictors use dynamic information, i.e., information that becomes available as the program executes. This is usually information regarding the past history of branch outcomes—either the specific branch being predicted, other branches leading up to it, or both. This branch history is saved in a *branch history table* or *branch prediction table* [40], [53], or may be appended to the instruction cache line that contains the branch. The branch prediction table is typi-

²Although this could be done through careful opcode selection, the need to maintain compatibility, dense assignments to reduce opcode bits, and cluttered opcode assignments caused by generations of architecture extensions often make this difficult in practice.

cally organized in a cache-like manner and is accessed with the address of the branch instruction to be predicted. Within the table previous branch history is often recorded by using multiple bits, typically implemented as small counters, so that the results of several past branch executions can be summarized [53]. The counters are incremented on a taken branch (stopping at a maximum value) and are decremented on a not-taken branch (stopping at a minimum value). Consequently, a counter's value summarizes the dominant outcome of recent branch executions.

At some time after a prediction has been made, the actual branch outcome is evaluated. Dynamic branch history information can then be updated. (It could also be updated speculatively, at the time the prediction was made [26], [60].) If the prediction was incorrect, instruction fetching must be redirected to the correct path. Furthermore, if instructions were processed speculatively based on the prediction, they must be purged and their results must be nullified. The process of speculatively executing instruction is described in more detail later.

3) *Computing Branch Targets*: To compute a branch target usually requires an integer addition. In most architectures, branch targets (at least for the commonly used branches) are relative to the program counter and use an offset value held in the instruction. This eliminates the need to read the register(s). However, the addition of the offset and the program counter is still required; furthermore, there may still be some branches that do need register contents to compute branch targets. Finding the target address can be sped up by having a branch target buffer which holds the target address that was used the last time the branch was executed. An example is the Branch Target Address Cache used in the PowerPC 604 [22].

4) *Transferring Control*: When there is a taken (or predicted taken) branch there is often at least a clock cycle delay in recognizing the branch, modifying the program counter, and fetching instructions from the target address. This delay can result in pipeline bubbles unless some steps are taken. Perhaps the most common solution is to use the instruction buffer with its stockpiled instructions to mask the delay; more complex buffers may contain instructions from both the taken and the not taken paths of the branch. Some of the earlier RISC instruction sets used *delayed branches* [27], [47]. That is, a branch did not take effect until the instruction after the branch. With this method, the fetch of target instructions could be overlapped with the execution of the instruction following the branch. However, in more recent processor implementations, delayed branches are considered to be a complication because they make certain assumptions about the pipeline structure that may no longer hold in advanced superscalar implementations.

B. Instruction Decoding, Renaming, and Dispatch

During this phase, instructions are removed from the instruction fetch buffers, examined, and control and data dependence linkages are set up for the remaining pipeline phases. This phase includes detection of true data dependences (due to RAW hazards) and resolution of other

register hazards, e.g., WAW and WAR hazards caused by register reuse. Typically, this phase also distributes, or *dispatches*, instructions to buffers associated with hardware functional units for later issuing and execution. This phase works closely with the following issue stage, as will become apparent in the next section.

The job of the decode phase is to set up one or more execution *tuples* for each instruction. An execution tuple is an ordered list containing: 1) an operation to be executed, 2) the identities of storage elements where the input operands reside (or will eventually reside), and 3) locations where the instruction's result must be placed. In the static program, the storage elements are the architectural storage elements, namely the architected, or *logical*, registers and main memory locations. To overcome WAR and WAW hazards and increase parallelism during dynamic execution, however, there are often *physical* storage elements that may differ from the logical storage elements. Recall that the logical storage elements can be viewed simply as a device to help describe what must be done. During parallel execution, there may be multiple data values stored in multiple physical storage elements—with all of the values associated with the same logical storage element. However, each of the values correspond to different points in time during a purely sequential execution process.

When an instruction creates a new value for a logical register, the physical location of the value is given a “name” known by the hardware. Any subsequent instructions which use the value as an input are provided with the name of its physical storage location. This is accomplished in the decode/rename/dispatch phase (see Fig. 4) by replacing the logical register name in an instruction's execution tuple (i.e., the register designator) with the new name of the physical storage location; the register is therefore said to be *renamed* [36].

There are two register renaming methods in common usage. In the first, there is a physical register file larger than the logical register file. A mapping table is used to associate a physical register with the current value of a logical register [21], [30], [45], [46]. Instructions are decoded and register renaming is performed in sequential program order. When an instruction is decoded, its logical result register (destination) is assigned a physical register from a *free list*, i.e., the physical registers that do not currently have a logical value assigned to them, and the mapping table is adjusted to reflect the new logical to physical mapping for that register. In the process, the physical register is removed from the free list. Also, as part of the rename operation, the instruction's source register designators are used to look up their current physical register names in the mapping table. These are the locations from which the source operand values will be read. Instruction dispatch is temporarily halted if the free list is empty.

Fig. 5 is a simple example of physical register renaming. The instruction being considered is the first *add r3,r3,4* instruction in Fig. 1. The logical, architected registers are denoted with lower case letters, and the physical registers use upper case. In the *before* side of the figure, the mapping

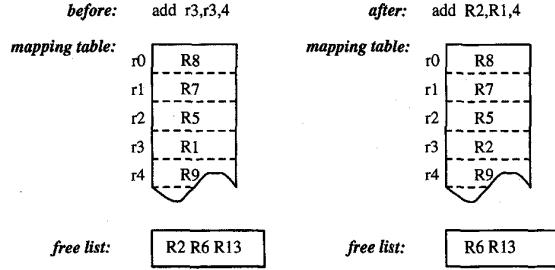


Fig. 5. Example of register renaming; logical registers are shown in lowercase, and physical registers are in uppercase.

table shows that register r3 maps to physical register R1. The first available physical register in the free list is R2. During renaming, the *add* instruction's source register r3 is replaced with R1, the physical register where a value will be placed by a preceding instruction. The destination register r3 is renamed to the first free physical register, R2, and this register is placed in the mapping table. Any subsequent instruction that reads the value produced by the *add* will have its source register mapped to R2 so that it gets the correct value. The example shown in Fig. 8, to be discussed later, shows the renaming of all the values assigned to register r3.

The only remaining matter is the reclaiming of a physical register for the free list. After a physical register has been read for the last time, it is no longer needed and can be placed back into the free list for use by other instructions. Depending on the specific implementation, this can require some more-or-less complicated hardware bookkeeping. One method is to associate a counter with each physical register. The counter is incremented each time a source register is renamed to the physical register and is decremented each time an instruction issues and actually reads a value from the register. The physical register is free whenever the count is zero, *provided* the corresponding logical register has been subsequently renamed to another physical register.

In practice, a method simpler than the counter method is to wait until the corresponding logical register has not only been renamed by a later instruction, but the later instruction has received its value and has been committed (see Section III-E). At this point, the earlier version of the register is guaranteed to no longer be needed, including for precise state restoration in case of an interrupt.

The second method of renaming uses a physical register file that is the same size as the logical register file, and the customary one-to-one relationship is maintained. In addition, there is a buffer with one entry per active instruction (i.e., an instruction that been dispatched for execution but which has not yet committed.) This buffer is typically referred to as a *reorder buffer* [32], [55], [57] because it is also used to maintain proper instruction ordering for precise interrupts.

Fig. 6 illustrates a reorder buffer. It is easiest to think of it as FIFO storage, implemented in hardware as a circular buffer with head and tail pointers. As instructions are dispatched according to sequential program order, they

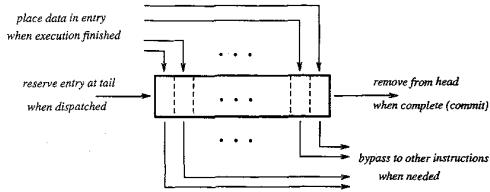


Fig. 6. A reorder buffer; entries are reserved and released in FIFO order. However, instruction results may be placed in the entry at any time as they complete, and results may be read out at any time for use by dependent instructions.

are assigned entries at the tail of the reorder buffer. As instructions complete execution, their result values are inserted into their previously assigned entry, wherever it may happen to be in the reorder buffer. At the time an instruction reaches the head of the reorder buffer, if it has completed execution, its entry is removed from the buffer and its result value is placed in the register file. An incomplete instruction blocks at the head of the reorder buffer until its value arrives. Of course, a superscalar processor must be capable of putting new entries into the buffer and taking them out more than one at a time; nevertheless, adding and removing new entries still follows the FIFO discipline.

A logical register value may reside either in the physical register file or may be in the reorder buffer. When an instruction is decoded, its result value is first assigned a physical entry in the reorder buffer and a mapping table entry is set up accordingly. That is, the table indicates that a result value can be found in the specific reorder buffer entry corresponding to the instruction that produces it. An instruction's source register designators are used to access the mapping table. The table either indicates that the corresponding register has the required value, or that it may be found in the reorder buffer. Finally, when the reorder buffer is full, instruction dispatch is temporarily halted.

Fig. 7 shows the renaming process applied to the same *add r3,r3,4* instruction as in Fig. 5. At the time the instruction is ready for dispatch (the *before* half of the figure), the values for r1 through r2 reside in the register file. However, the value for r3 resides (or will reside) in reorder buffer entry 6 until that reorder buffer entry is committed and the value can be written into the register file. Consequently, as part of the renaming process, the source register r3 is replaced with the reorder buffer entry 6 (rob6). The *add* instruction is allocated the reorder buffer entry at the tail, entry number 8 (rob8). This reorder buffer number is then recorded in the mapping table for instructions that use the result of the *add*.

In summary, regardless of the method used, register renaming eliminates artificial dependences due to WAW and WAR hazards, leaving only the true RAW register dependences.

C. Instruction Issuing and Parallel Execution

As we have just seen, it is in the decode/rename/dispatch phase that an execution tuple (opcode plus physical register

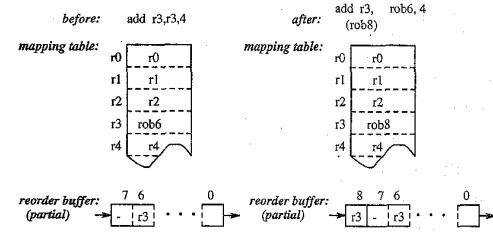


Fig. 7. Example of renaming with reorder buffer.

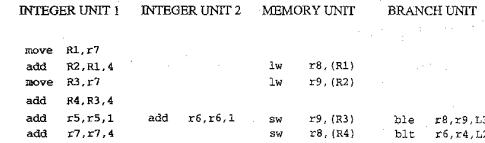


Fig. 8. Example of parallel execution schedule. In the example, we show only the renaming of logical register r3. Its physical register names are in upper case (R1, R2, R3, and R4).

storage locations) is formed. Once execution tuples are created and buffered, the next step is to determine which tuples can be *issued* for execution. Instruction issue is defined as the run-time checking for availability of data and resources. This is an area of the processing pipeline which is at the heart of many superscalar implementations—it is the area that contains the window of execution.

Ideally an instruction is ready to execute as soon as its input operands are available. However, other constraints may be placed on the issue of instructions, most notably the availability of physical resources such as execution units, interconnect, and register file (or reorder buffer) ports. Other constraints are related to the organization of buffers holding the execution tuples.

Fig. 8 is an example of one possible parallel execution schedule for an iteration of our working example (Fig. 1). This schedule assumes hardware resources consisting of two integer units, one path to the memory, and one branch unit. The vertical direction corresponds to time steps, and the horizontal direction corresponds to the operations executed in the time step. In this schedule, we predicted that the branch *ble* was not going to be taken and are speculatively executing instructions from the predicted path. For illustrative purposes, we show only the renamed values for r3; in an actual implementation, the other registers would be renamed as well. Each of the different values assigned to r3 is bound to a different physical register (R1, R2, R3, R4).

Following paragraphs briefly describe a number of ways of organizing instruction issue buffers, in order of increasing complexity. Some of the basic organizations are illustrated in Fig. 9.

1) *Single Queue Method*: If there is a single queue, and no out-of-order issuing, then register renaming is not required, and operand availability can be managed via simple reservation bits assigned to each register. A register is *reserved* when an instruction modifying the register issues, and the reservation is *cleared* when the instruction completes. An instruction may issue (subject to physical

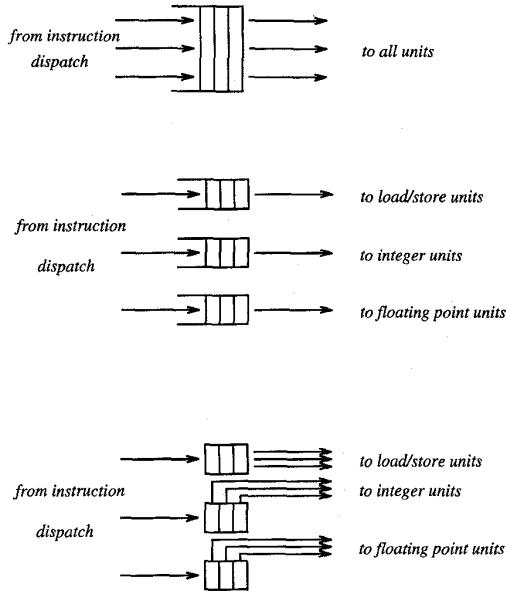


Fig. 9. Methods of organizing instruction issue queues.

resource availability) if there are no reservations on its operands.

2) *Multiple Queue Method*: With multiple queues, instructions issue from each queue in order, but the queues may issue out of order with respect to one another. The individual queues are organized according to instruction type. For example, there may be a floating point instruction queue, an integer instruction queue, and a load/store instruction queue. Here, renaming may be used in a restricted form. For example, only the registers loaded from memory may be renamed. This allows the load/store queue to “slip” ahead of the other instruction queues, fetching memory data in advance of when it is needed. Some earlier superscalar implementations used this method [21], [43], [54].

3) *Reservation Stations*: With reservation stations (see Fig. 10), which were first proposed as a part of Tomasulo’s algorithm [63], instructions may issue out of order; there is no strict FIFO ordering. Consequently, all of the reservation stations simultaneously monitor their source operands for data availability. The traditional way of doing this is to hold operand data in the reservation station. When an instruction is dispatched to the reservation station, any already-available operand values are read from the register file and placed in the reservation station. Then reservation station logic compares the operand designators of unavailable data with the result designators of completing instructions. When there is a match, the result value is pulled into the matching reservation station. When all the operands are ready in the reservation station, the instruction may issue (subject to hardware resource availability). Reservations may be partitioned according to instruction type (to reduce data paths) [45], [63] or may be pooled into a single large block [57]. Finally, some recent reservation station implementations do not hold the actual source data, but

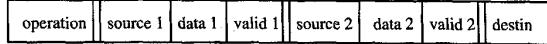


Fig. 10. A typical reservation station. There are entries for the *operation* to be performed and places for each *source* designator, its *data* value, and a *valid* bit indicating that the data value is present in the reservation station. As an instruction completes and produces result data, a comparison of the instruction’s destination designator is made with the source designators in the reservation station to see if instruction in the station is waiting for the data; if so, the data is written into the value field, and the corresponding valid bit is set. When all the source operands are valid, the instruction in the reservation station is ready to issue and begin execution. At the time it begins execution, the instruction takes its destination designator along, to be used later for its reservation station comparisons.

instead hold pointers to where the data can be found, e.g., in the register file or a reorder buffer entry.

D. Handling Memory Operations

Memory operations need special treatment in superscalar processors. In most modern RISC instruction sets, only explicit load and store instructions access memory, and we shall use these instructions in our discussion of memory operations (although the concepts are easily applied to register-storage and storage-storage instruction sets).

To reduce the latency of memory operations, memory hierarchies are used. The expectation is that most data requests will be serviced by data cache memories residing at the lower levels of the hierarchy. Virtually all processors today contain a data cache, and it is rapidly becoming commonplace to have multiple levels of data caching, e.g. a small fast cache at the *primary* level and a somewhat slower, but larger, *secondary* cache. Most microprocessors integrate the primary cache on the same chip as the processor; notable exceptions are some of processors developed by HP [4] and the high-end IBM POWER series [65].

Unlike ALU instructions, for which it is possible to identify during the decode phase the register operands that will be accessed, it is not possible to identify the memory locations that will be accessed by load and store instructions until after the issue phase. The determination of the memory location that will be accessed requires an *address calculation*, usually an integer addition. Accordingly, load and store instructions are issued to an execute phase where address calculation is performed. After address calculation, an *address translation* may be required to generate a physical address. A cache of translation descriptors of recently accessed pages, the *translation lookaside buffer* (TLB), is used to speed up this address translation process [52]. Once a valid memory address has been obtained, the load or store operation can be submitted to the memory. It should be noted that although this suggests address translation and memory access are done in series, in many superscalar implementations these two operations are overlapped—the initial cache access is done in parallel with address translation and the translated address is then used to compare with cache tag(s) to determine if there is a hit.

As with operations carried out on registers, it is desirable to have a collection of memory operations execute in as

little time as possible. This means reducing the latency of memory operations, executing multiple memory operations at the same time (i.e., overlapping the execution of memory operations), overlapping the execution of memory operations with nonmemory operations, and possibly allowing memory operations to execute out-of-order. However, because there are many more memory locations than registers, and because memory is indirectly addressed through registers, it is not practical to use the solutions described in the previous sections to allow memory operations to be overlapped and proceed out of order. Rather than have rename tables with information for each memory location, as we did in the case of registers, the general approach is to keep information only for a *currently active* subset of the memory locations, i.e., memory locations with currently pending memory operations, and search this subset associatively when memory needs to be accessed [3], [7], [19], [46].

Some superscalar processors only allow single memory operations per cycle, but this is rapidly becoming a performance bottleneck. To allow multiple memory requests to be serviced simultaneously, the memory hierarchy has to be multiported. It is usually sufficient to multiport only the lowest level of the memory hierarchy, namely the primary caches since many requests do not proceed to the upper levels in the hierarchy. Furthermore, transfers from the higher levels to the primary cache are in the form of lines, containing multiple consecutive words of data. Multiporting can be achieved by having multiported storage cells, by having multiple banks of memory [9], [29], [58], or by making multiple serial requests during the same cycle [65]. To allow memory operations to be overlapped with other operations (both memory and nonmemory), the memory hierarchy must be *nonblocking* [37], [58]. That is, if a memory request misses in the data cache, other processing operations, including further memory requests, should be allowed to proceed.

The key to allowing memory operations to be overlapped, or to proceed out of order, is to ensure that hazards are properly resolved, and that sequential execution semantics are preserved.³ *Store address buffers*, or simply store buffers, are used to make sure that operations submitted to the memory hierarchy do not violate hazard conditions. A store buffer contains addresses of all pending store operations. Before an operation (either a load or store) is issued to the memory, the store buffer is checked to see if there is a pending store to the same address. Fig. 11 illustrates a simple method for implementing hazard detection logic.

Once the operation has been submitted to the memory hierarchy, the operation may hit or miss in the data cache. In the case of a miss, the line containing the accessed location has to be fetched into the cache. Further accesses to the missing line must be made to wait, but other accesses may proceed. Miss handling status registers (MHSR's) are used to track the status of outstanding cache misses, and allow

³Some complex memory ordering issues can arise in shared memory multiprocessor implementations [15], but these are beyond the scope of this paper.

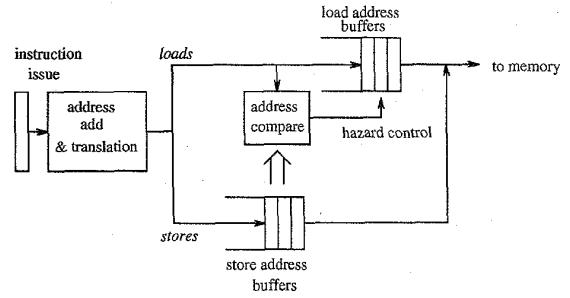


Fig. 11. Memory hazard detection logic. Store addresses are buffered in a queue (FIFO). Store addresses must remain buffered until 1) their data is available and 2) the store instruction is ready to be committed (to assure precise interrupts). New load addresses are checked with waiting store addresses. If there is a match, the load must wait for the store it matches. Store data may be bypassed to the matching load in some implementations.

multiple requests to the memory hierarchy to be overlapped [16], [37], [58].

E. Committing State

The final phase of the lifetime of an instruction is the *commit* or the *retire* phase, where the effects of the instruction are allowed to modify the logical process state. The purpose of this phase is to implement the appearance of a sequential execution model even though the actual execution is very likely nonsequential, due to speculative execution and out-of-order completion of instructions. The actions necessary in this phase depend upon the technique used to recover a precise state. There are two main techniques used to recover a precise state, both of which require the maintenance of two types of state: the state that is being updated as the operations execute and other state that is required for recovery.

In the first technique, the state of the machine at certain points is saved, or *checkpointed*, in either a *history buffer* or a checkpoint [31], [55]. Instructions update the state of the machine as they execute and when a precise state is needed, it is recovered from the history buffer. In this case, all that has to be done in the commit phase is to get rid of history state that is no longer required.

The second technique is to separate the state of the machine into two: the implemented physical state and a logical (architectural) state. The physical state is updated immediately as the operations complete. The architectural state of the machine is updated in sequential program order, as the speculative status of operations is cleared. With this technique, the speculative state is maintained in a reorder buffer; to commit an instruction, its result has to be moved from the reorder buffer into the architectural register file (and to memory in the case of a store), and space is freed up in the reorder buffer. Also, during the commit phase of a store instruction, a signal is sent to the store buffer and the store data is written to memory.

Only part of each reorder buffer entry is shown in Fig. 6. Because we were illustrating the renaming process, only the designator of the result register is shown. Also in a reorder

buffer entry is a place for the data value after it has been computed, but before it is committed. And, there is typically a place to hold the program counter value of the instruction and any interrupt conditions that might occur as a result of the instruction's execution. These are used to inform the commit logic when an interrupt should be initiated and the program counter value that should be entered as part of the precise interrupt state.

Thus far, we have discussed the reorder buffer method as it is used in implementations which use the reorder buffer method of register renaming (as illustrated in Fig. 6). However, the reorder buffer is also useful in the other style of renaming (where physical registers contain all the renamed values as in Fig. 5). With the physical register method, the reorder buffer does not have to hold values prior to being committed, rather they can be written directly into the physical register file. However, the reorder buffer does hold control and interrupt information. When an instruction is committed, the control information is used to move physical registers to the free list. If there is an interrupt, the control information used to adjust the logical-to-physical mapping table so that the mapping reflects the correct precise state.

Although the checkpoint/history buffer method has been used in some superscalar processors [12], the reorder buffer technique is by far the more popular technique because, in addition to providing a precise state, the reorder buffer method also helps implement the register renaming function [57].

F. The Role of Software

Though a major selling point of superscalar processors is to speed up the execution of existing program binaries, their performance can be enhanced if new, optimized binaries can be created.⁴ Software can assist by creating a binary so that the instruction fetching process, and the instruction issuing and execution process, can be made more efficient. This can be done in the following two ways: 1) increasing the likelihood that a group of instructions that are being considered for issue can be issued simultaneously, and 2) decreasing the likelihood that an instruction has to wait for a result of a previous instruction when it is being considered for execution. Both these techniques can be accomplished by *scheduling* instructions statically. By arranging the instructions so that a group of instructions in the static program matches the parallel execution constraints of the underlying superscalar processor, the first objective can be achieved. The parallel execution constraints that need to be considered include the dependence relationships between the instructions, and the resources available in the microarchitecture. (For example, to achieve the parallel issue of four consecutive instructions, the microarchitecture might require all four of them to be independent.) To achieve the second objective, an instruction which produces

⁴The new binary could use the same instruction set and the same sequential execution model as the old binary; the only difference might be the order in which the instructions are arranged.

a value for another instruction can be placed in the static code so that it is fetched and executed far in advance of the consumer instruction.

Software support for high performance processors is a vast subject—one that cannot be covered in any detail in this paper.

IV. THREE SUPERSCALAR MICROPROCESSORS

In this section we discuss three current superscalar microprocessors in light of the above framework. The three were chosen to cover the spectrum of superscalar implementations as much as possible. They are the MIPS R10000 [23], which fits the “typical” framework described above, the DEC Alpha 21164 [24] which strives for greater simplicity, and the AMD K5 [51] which implements a more complex and older instruction set than the other two—the Intel x86.

A. MIPS R10000

The MIPS R10000 does extensive dynamic scheduling and is very similar to our “typical” superscalar processor described above. In fact, Fig. 4 is modeled after the R10000 design. The R10000 fetches four instructions at a time from the instruction cache. These instructions have been predecoded when they were put into the cache. The predecode generates four additional bits per instruction which help determine the instruction type immediately after the instructions are fetched from the cache. After being fetched, branch instructions are predicted. The prediction table is contained within the instruction cache mechanism; the instruction cache holds 512 lines and there are 512 entries in the prediction table. Each entry in the prediction table holds a two bit-counter value that encodes history information used to make the prediction.

When a branch is predicted to be taken, it takes a cycle to redirect instruction fetching. During that cycle, sequential instructions (i.e., those on the not-taken path) are fetched and placed in a *resume cache* as a hedge against an incorrect prediction. The resume cache has space for four blocks of instructions, so four branch predictions can be handled at any given time.

Following instruction fetch, instructions are decoded, their register operands are renamed, and they are dispatched to the appropriate instruction queue. The predecoded bits, mentioned earlier, simplify this process. All register designators are renamed, using physical register files that are twice the size of the logical files (64 physical versus 32 logical). The destination register is assigned an unused physical register from the free list, and the map is updated to reflect the new logical-to-physical mapping. Operand registers are given the correct designators by reading the map.

Then, up to four instructions at a time are dispatched into one of three instruction queues: memory, integer, and floating point, i.e., the R10000 uses a form of queued instruction issue as in Fig. 9(c). Other than the total of at most four, there are no restrictions on the number of instructions that may be placed in any of the queues. The

queues are each 16 entries deep, and only a full queue can block dispatch. At the time of dispatch, a reservation bit for each physical result register is set busy. Instructions in the integer and floating point instruction queues do not issue in a true FIFO discipline (hence, the term "queue" is a little misleading). The queue entries act more like reservation stations; however, they do not have "value" fields or "valid" bits as in Fig. 10. Instead of value fields, the reservation stations hold the physical register designators which serve as pointers to the data. Each instruction in the queue monitors the global register reservation bits for its source operands. When the reservation bits all become "not busy," the instruction's source operands are ready, and subject to availability of its functional unit, the instruction is free to issue.

There are five functional units: an address adder, two integer ALU's, a floating point multiplier/divider/square-rooter and a floating point adder. The two integer ALU's are not identical—both are capable of the basic add/subtract/logical operations, but only one can do shifts and the other can do integer multiplications and additions.

As a general policy, an effort is made to process the older instruction first when more than one instruction has ready operands and is destined for the same functional unit. However, this policy does not result in a strict priority system in order to simplify the arbitration logic. The memory queue, however, does prioritize instructions according to age, which makes address hazard detection simpler.

The R10000 supports an on-chip primary data cache (32 Kb, two-way set associative, 32-byte lines), and an off-chip secondary cache. The primary cache blocks only on the second miss.

The R10000 uses a reorder buffer mechanism for maintaining a precise state at the time of an exception condition. Instructions are committed in the original program sequence, up to four at a time. When an instruction is committed, it frees up the old physical copy of the logical register it modifies. The new physical copy (which may have been written many cycles before) becomes the new architecturally precise version. Exception conditions for noncommitted instructions are held in the reorder buffer. When an instruction with exceptions is ready to be committed, an interrupt will occur. Information in the reorder buffer for instructions following the interrupting one are used to readjust the logical-to-physical register mapping so that the logical state is consistent with the interrupting instruction.

When a branch is predicted, the processor takes a snapshot of the register mapping table. Then, if the branch is subsequently found to be mispredicted, the register mapping can be quickly restored. There is space to allow snapshots of up to four predicted branches at any given time. Furthermore, by using the resume cache, instruction processing can often begin immediately.

B. Alpha 21164

The Alpha 21164 (Fig. 12) is an example of a simple superscalar processor that forgoes the advantages of dynamic instruction scheduling in favor of a high clock

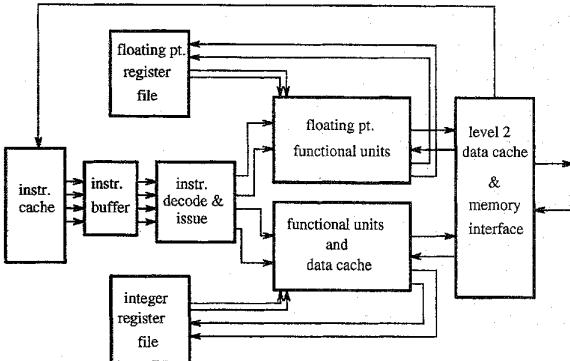


Fig. 12. DEC Alpha 21164 superscalar organization.

rate. Instructions are fetched from an 8 Kbytes instruction cache four at a time. These instructions are placed in one of two instruction buffers, each capable of holding four instructions. Instructions are issued from the buffers in program order, and a buffer must be completely emptied before the next buffer can be used. This restricts the instruction issue rate somewhat, but it greatly simplifies the necessary control logic.

Branches are predicted using a table that is associated with the instruction cache. There is a branch history entry for each instruction in the cache; each entry is a two bit counter. There can be only one predicted, and yet unresolved, branch in the machine at a time; instruction issue is stalled if another branch is encountered before a previously predicted branch has been resolved.

Following instruction fetch and decode, instructions are inspected and arranged according to type, i.e., the functional unit they will use. Then, provided operand data is ready (in registers or available for bypassing) instructions are issued to the units and begin execution. During this entire process, instructions are not allowed to pass one another. The 21164 is therefore an example of the single queue method shown in Fig. 9(a).

There are four functional units: two integer ALU's, a floating point adder, and a floating point multiplier. The integer ALU's are not identical—only one can perform shifts and integer multiplies, the other is the only one that can evaluate branches.

The 21164 has two levels of cache memory on the chip. There are a pair of small, fast primary caches of 8 K bytes each—one for instructions and one for data. The secondary cache, shared by instructions and data, is 96 K bytes and is three-way set associative. The small direct mapped primary cache is to allow single-cycle cache accesses at a very high clock rate. The primary cache can sustain a number of outstanding misses. There is a six entry *miss address file* (MAF) that contains the address and target register for a load that misses. If the address is to the same line as is another address in the MAF, the two are merged into the same entry. Hence the MAF can hold many more than six outstanding cache misses (in theory, up to 21).

To provide a sequential state at the time of a interrupt, the 21164 does not issue out of order, and keeps instructions in

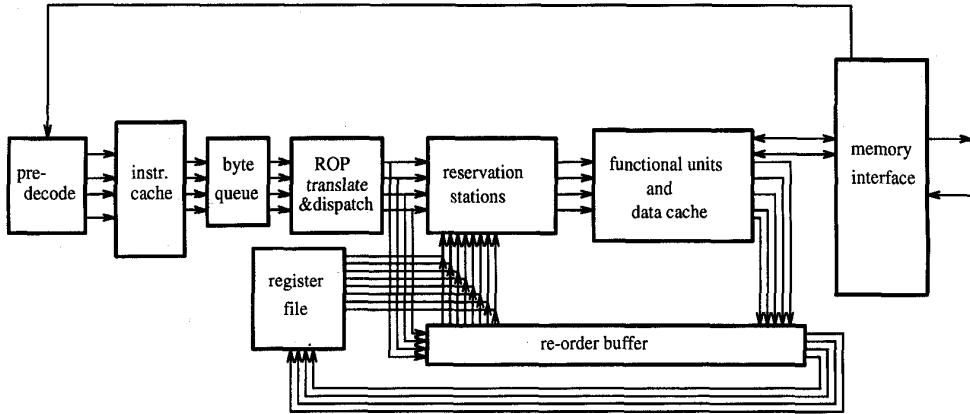


Fig. 13. AMD K5 superscalar implementation of the Intel x86 instruction set.

sequence as they flow down the pipeline. The final pipeline stage updates the register file in original program sequence. There are bypasses along the pipeline so that data can be used before it is committed to the register file. This is a simple form of a reorder buffer with bypasses. The pipeline itself forms the reorder buffer (and order is maintained so “reorder” overstates what is actually done). Floating point instructions are allowed to update their files out of order, and, as a result, not all floating point exceptions result in a precise architectural state.

C. AMD K5

In contrast to the other two microprocessors discussed in this section, the AMD K5 (Fig. 13) implements an instruction set that was not originally designed for fast, pipelined implementation. It is an example of a complex instruction set—the Intel x86.

The Intel x86 instruction set uses variable length instructions. This means, among other things, one instruction must be decoded (and its length established) before the beginning of the next can be found. Consequently, instructions are sequentially predecoded as they are placed into the instruction cache. There are five predecode bits per instruction byte. These bits indicate information such as whether the byte is the beginning or end of an instruction and identifies bytes holding opcodes and operands. Instructions are then fetched from the instruction cache at a rate of up to 16 bytes per cycle. These instruction bytes are placed into a 16-element byte queue where they wait for dispatch.

As in the other microprocessors described in this section, the K5 integrates its branch prediction logic with the instruction cache. There is one prediction entry per cache line. A single bit reflects the direction taken by the previous execution of the branch. The prediction entry also contains a pointer to the target instruction—this information indicates where in the instruction cache the target can be found. This reduces the delay for fetching a predicted target instruction.

Due to the complexity of the instruction set, two cycles are consumed decoding. In the first stage, instruction bytes are read from the byte queue and are converted to simple, RISC-like Operations (called ROP's by AMD). Up to four

ROP's are formed at a time. Often, the conversion requires a small number of ROP's per x86 instruction. In this case, the hardware does a single-cycle conversion. More complex x86 instructions are converted into a sequence of ROP's via sequential lookup from a ROM. In this case, the ROP's are essentially a form of microcode; up to 4 ROP's per cycle are generated from the ROM. After conversion to ROP's, most of the processor is focussed on their execution as individual operations, largely ignoring the original x86 instructions from which they were generated.

Following the first decode cycle, instructions read operand data (if available) and are dispatched to functional unit reservation stations, up to 4 ROP's per cycle (corresponding to up to 4 x 86 instructions). Data can be in the register file, or can be held in the reorder buffer. When available this data is read and sent with the instruction to functional unit reservation stations. If operand data has not yet been computed, the instruction will wait for it in the reservation station.

There are six functional units: two integer ALU's, one floating point unit, two load/store units, and a branch unit. One of the integer ALU's capable of shifts, and the other can perform integer divides. Although they are shown as a single block in Fig. 13, the reservation stations are partitioned among the functional units. Except the FPU, each of these units has two reservation stations; the FPU has only one. Based on availability of data, ROP's are issued from the reservation stations to their associated functional units. There are enough register ports and data paths to support up to four such ROP issues per cycle.

There is an 8 K byte data cache which has four banks. Dual load/stores are allowed, provided they are to different banks. (One exception is if both references are to the same line, in which case they can both be serviced together.)

A 16-entry reorder buffer is used to maintain a precise process state when there is an exception. The K5 reorder buffer holds result data until it is ready to be placed in the register file. There are bypasses from the buffer and an associative lookup is required to find the data if it is complete, but pending the register write. This is a classic reorder buffer method of renaming we described earlier.

The reorder buffer is also used to recover from incorrect branch predictions.

V. CONCLUSION

A. What Comes Next?

Both performance and compatibility have driven the development of superscalar processors. They implement a sequential execution model although the actual execution of a program is far from sequential. After being fetched, the sequential instruction stream is torn apart with only true dependences holding the instructions together. Instructions are executed in parallel with a minimum of constraints. Meanwhile enough information concerning the original sequential order is retained so that the instruction stream can conceptually be squeezed back together again should there be need for a precise interrupt.

A number of studies have been done to determine the performance of superscalar methods [56], [64]. Because the hardware and software assumptions and benchmarks vary widely, so do the potential speedups—ranging from close to 1 for some program/hardware combinations to many 100's for others. In any event, it is clear that there is something to be gained, and every microprocessor manufacturer has embraced superscalar methods for high-end products.

While superscalar implementations have become a staple just like pipelining, there is some consensus that returns are likely to diminish as more parallel hardware resources are introduced. There are a number of reasons for thinking this, we give two of the more important. First, there may be limits to the instruction level parallelism that can be exposed and exploited by currently known superscalar methods, even if very aggressive methods are used [64]. Perhaps the most important of these limits results from conditional branches. Studies that compare performance using real branch prediction with theoretically perfect branch prediction note a significant performance decline when real prediction is used. Second, superscalar implementations grow in complexity as the number of simultaneously issued instructions increases. Data path complexity (numbers of functional units, register and cache ports, for example) grows for obvious reasons. However, control complexity also grows because of the “cross checks” that must be done between all the instructions being simultaneously dispatched and/or issued. This represents quadratic growth in control complexity which, sooner or later, will affect the clock period that can be maintained.

In addition, there are some important system level issues that might ultimately limit performance. One of the more important is the widening gap between processor and main memory performance. While memories have gotten larger with each generation, they have not gotten much faster. Processor speeds, on the other hand have improved markedly. A number of solutions are being considered, ranging from more sophisticated data cache designs, better cache prefetch methods, more developed memory hierarchies, and memory chip designs that are more effective at

providing data. This issue affects all the proposed methods for increasing instruction level parallelism, not just the superscalar implementations.

In the absence of radically new superscalar methods, many believe that 8-way superscalar (more or less) is a practical limit—this is a point that will be reached within the next couple of generations of processors. Where do we go from that point? One school of thought believes that the very large instruction word (VLIW) model [10], [17], [48] will become the paradigm of choice. The VLIW model is derived from the sequential execution model: control sequences from instruction to instruction, just like in the sequential model discussed in this paper. However, each instruction in a VLIW is a collection of several independent operations, which can all be executed in parallel. The compiler is responsible for generating a parallel execution schedule, and representing the execution schedule as a collection of VLIW instructions. If the underlying hardware resources necessitate a different execution schedule, a different program binary must be generated for optimal performance.

VLIW proponents claim several advantages for their approach, and we will summarize two of the more important. First, with software being responsible for analyzing dependences and creating the execution schedule, the size of the instruction window that can be examined for parallelism is much larger than what a superscalar processor can do in hardware. Accordingly, a VLIW processor can expose more parallelism. Second, since the control logic in a VLIW processor does not have to do any dependence checking (the entire parallel execution schedule is orchestrated by the software), VLIW hardware can be much simpler to implement (and therefore may allow a faster clock) than superscalar hardware.

VLIW proponents have other arguments that favor their approach, and VLIW opponents have counter arguments in each case. The arguments typically center around the actual effect that superscalar control logic complexity will have on performance and the ability of a VLIW's compile time analysis to match dynamic scheduling using run-time information. With announced plans from Intel and Hewlett-Packard and with other companies seriously looking at VLIW architectures, the battle lines have been drawn and the outcome may be known in a very few years.

Other schools of thought strive to get away from the single flow of control mindset prevalent in the sequential execution model (and the superscalar implementations of the model). Rather than sequence through the program instruction-by-instruction, in an attempt to create the dynamic instruction sequence from the static representation, more parallel forms of sequencing may be used.

In [39] it is shown that allowing multiple control flows provides the potential for substantial performance gains. That is, if multiple program counters can be used for fetching instructions, then the effective window of execution from which independent instructions can be chosen for parallel execution can be much wider.

One such method is a variation of the time-tested *multiprocessing* method. Some supercomputer and mini-supercomputer vendors have already implemented approaches for automatic parallelization of high level language programs [5], [33]. With chip resources being sufficient to allow multiple processors to be integrated on a chip, multiprocessing, once the domain of supercomputers and mainframes, is being considered for high-end microprocessor design [25], [42]. If multiprocessing is to exploit instruction level parallelism at the very small grain size that is implied by “instruction level,” suitable support has to be provided to keep the inter-processor synchronization and communication overhead extremely low. As with traditional multiprocessing, single-chip multiprocessors will require enormous amounts compiler support to create a parallel version of a program statically.

Looking even farther ahead, more radical approaches that support multiple flows of control are being studied [14], [59]. In these approaches, individual processing units are much more highly integrated than in a traditional multiprocessor, and the flow control mechanism is built into the hardware, as well. By providing mechanisms to support speculative execution, such methods allow the parallel execution of programs that can’t be analyzed and parallelized statically.

Microprocessor designs have successfully passed from simple pipelines, to simple superscalar implementations, to fully developed superscalar processors—all in little more than a decade. With the benefit of hindsight, each step seems to be a logical progression from the previous one. Each step tapped new sources of instruction level parallelism. Today, researchers and architects are facing new challenges. The next logical step is not clear, and the search for new, innovative approaches to instruction level parallelism promises to make the next decade as exciting as the previous one.

ACKNOWLEDGMENT

The authors would like to thank Mark Hill for comments and Scott Breach and Dionisios Pnevmatikatos for their help with some of the figures.

REFERENCES

- [1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.
- [2] G. Amdahl *et al.*, “Architecture of the IBM System/360,” *IBM J. Res. Develop.*, vol. 8, pp. 87–101, Apr. 1964.
- [3] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, “The IBM System/360 model 91: Machine philosophy and instruction-handling,” *IBM J. Res. Develop.*, vol. 11, pp. 8–24, Jan. 1967.
- [4] T. Asprey *et al.*, “Performance features of the PA7100 microprocessor,” *IEEE Micro*, vol. 13, pp. 22–35, June 1993.
- [5] M. C. Aug, G. M. Brost, C. C. Hsiung, and A. J. Schiffleger, “Cray X-MP: The birth of a supercomputer,” *IEEE Comput.*, vol. 22, pp. 45–54, Jan. 1989.
- [6] C. G. Bell, “Multis: A new class of multiprocessor computers,” *Sci.*, vol. 228, pp. 462–467, Apr. 1985.
- [7] L. J. Boland *et al.*, “The IBM System/360 model 91: Storage system,” *IBM J.*, vol. 11, pp. 54–68, Jan. 1967.
- [8] W. Bucholz, Ed., *Planning a Computer System*. New York: McGraw-Hill, 1962.
- [9] B. Case, “Intel reveals pentium implementation details,” *Microprocess. Rep.*, pp. 9–13, Mar. 1993.
- [10] R. P. Colwell *et al.*, “A VLIW architecture for a trace scheduling compiler,” *IEEE Trans. Comput.*, vol. 37, pp. 967–979, Aug. 1988.
- [11] T. M. Conte, P. M. Mills, K. N. Menezes, and B. A. Patel, “Optimization of instruction fetch mechanisms for high issue rates,” in *Proc. 22nd Annu. Int. Symp. on Comput. Architecture*, pp. 333–344, June 1995.
- [12] K. Diefendorff and M. Allen, “Organization of the Motorola 88110 superscalar RISC microprocessor,” *IEEE Micro*, vol. 12, Apr. 1992.
- [13] P. K. Dubey and M. J. Flynn, “Optimal Pipelining,” *J. Parallel and Distrib. Computing*, vol. 8, pp. 10–19, 1990.
- [14] P. K. Dubey, K. O’Brien, and C. Barton, “Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading,” in *Proc. Int. Conf. on Parallel Architectures and Compilation Techn. (PACT ’95)*, June 1995, pp. 109–121.
- [15] M. Dubois, C. Scheurich, and F. A. Briggs, “Synchronization, coherence and event ordering in multiprocessors,” *IEEE Comput.*, vol. 21, pp. 9–21, Feb. 1988.
- [16] K. I. Farkas and N. P. Jouppi, “Complexity/performance trade-offs with nonblocking loads,” in *Proc. 21st Annu. Int. Symp. on Comput. Architecture*, pp. 211–222, Apr. 1994.
- [17] J. A. Fisher and B. R. Rau, “Instruction-level parallel processing,” *Sci.*, pp. 1233–1241, Sept. 1991.
- [18] J. A. Fisher and S. M. Freudenberg, “Predicting conditional branch directions from previous runs of a program,” in *Proc. Architectural Support for Programming Languages and Operating Syst. (ASPLOS-V)*, 1992.
- [19] M. Franklin and G. S. Sohi, “ARB: A hardware mechanism for dynamic memory disambiguation,” to be published in *IEEE Trans. Comput.*.
- [20] G. F. Grohoski, J. A. Kahle, L. E. Thatcher, and C. R. Moore, “Branch and fixed point instruction execution units,” *IBM RISC Syst./6000 Technol.*, Austin, TX.
- [21] G. F. Grohoski, “Machine organization of the IBM RISC System/6000 processor,” *IBM J. Res. Develop.*, vol. 34, pp. 37–58, Jan. 1990.
- [22] L. Gwennap, “PPC 604 powers past Pentium,” *Microprocessor Rep.*, pp. 5–8, Apr. 18, 1994.
- [23] —, “MIPS R10000 uses decoupled architecture,” *Microprocessor Rep.*, pp. 18–22, Oct. 24, 1994.
- [24] —, “Digital leads the pack with the 21164,” *Microprocessor Rep.*, pp. 1, 6–10, Sept. 12, 1994.
- [25] —, “Architects debate VLIW, single-chip MP,” *Microprocessor Rep.*, pp. 20–21, Dec. 5, 1994.
- [26] E. Hao, P.-Y. Chang, and Y. N. Patt, “The effect of speculatively updating branch history on branch prediction accuracy, revisited,” in *Proc. 27th Int. Symp. on Microarchitecture*, pp. 228–232, Dec. 1994.
- [27] J. Hennessy *et al.*, “Hardware/software tradeoffs for increased performance,” in *Proc. Int. Symp. on Arch. Support for Prog. Lang. and Operating Syst.*, Mar. 1982, pp. 2–11.
- [28] R. G. Hintz and B. P. Tate, “Control data STAR-100 processor design,” *COMPCON*, p. 396, Sept. 1972.
- [29] P. Y. T. Hsu, “Design of the R8000 microprocessor,” *IEEE Micro*, pp. 23–33, Apr. 1994.
- [30] W. W. Hwu and Y. N. Patt, “HPSm, a high performance restricted data flow architecture having minimal functionality,” in *Proc. 13th Annu. Int. Symp. on Comput. Architecture*, pp. 297–307, June 1986.
- [31] —, “Checkpoint repair for high-performance out-of-order execution machines,” *IEEE Trans. Comput.*, vol. C-36, pp. 1496–1514, Dec. 1987.
- [32] M. Johnson, *Superscalar Design*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
- [33] T. Jones, “Engineering design of the convex C2,” *IEEE Comput.*, vol. 22, pp. 36–44, Jan. 1989.
- [34] N. P. Jouppi and D. W. Wall, “Available instruction-level parallelism for superscalar and superpipelined machines,” in *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, Boston, MA, Apr. 1989.
- [35] D. R. Kaeli and P. G. Emma, “Branch history table prediction of moving target branches due to subroutine returns,” in *Proc. 18th Annu. Int. Symp. on Comput. Architecture*, May 1991, pp. 34–42.

- [36] R. M. Keller, "Look-ahead processors," *ACM Comput. Surveys*, vol. 7, pp. 66-72, Dec. 1975.
- [37] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proc. 8th Annu. Symp. on Comput. Architecture*, May 1981, pp. 81-87.
- [38] S. R. Kunkel and J. E. Smith, "Optimal pipelining in supercomputers," in *Proc. 13th Annu. Int. Symp. on Comput. Architecture*, June 1986, pp. 404-413.
- [39] M. S. Lam and R. Wilson, "Limits of control flow on parallelism," in *Proc. 19th Annu. Int. Symp. on Comput. Architecture*, May 1992, pp. 46-57.
- [40] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Comput.*, vol. 17, pp. 6-22, Jan. 1984.
- [41] C. R. Moore, "The powerPC 601 microprocessor," in *Proc. Compcon 1993*, Feb. 1993, pp. 109-116.
- [42] B. A. Nayfeh and K. Olukotun, "Exploring the design space for a shared-cache multiprocessor," in *Proc. 21st Annu. Int. Symp. on Comput. Architecture*, Apr. 1994, pp. 166-175.
- [43] R. R. Oehler and R. D. Groves, "IBM RISC System/6000 processor architecture," *IBM J. Res. Develop.*, vol. 34, pp. 23-36, Jan. 1990.
- [44] S.-T. Pan, K. So, and J. T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," in *Proc. Architectural Support for Programming Languages and Operating Syst. (ASPLOS-V)*, Oct. 1992, pp. 76-84.
- [45] Y. N. Patt, W. W. Hwu, and M. Shebanow, "HPS, a new microarchitecture: Rationale and introduction," in *Proc. 18th Annu. Workshop on Microprogramming*, Dec. 1985, pp. 103-108.
- [46] Y. N. Patt, S. W. Melvin, W. W. Hwu, and M. Shebanow, "Critical issues regarding HPS, a high performance microarchitecture," in *Proc. 18th Annu. Workshop on Microprogramming*, Dec. 1985, pp. 109-116.
- [47] D. A. Patterson and C. H. Sequin, "RISC 1: A reduced instruction set VLSI computer," in *Proc. 8th Annu. Symp. on Comput. Architecture*, pp. 443-459, May 1981.
- [48] B. R. Rau, D. W. L. Yen, W. Yen, and R. Towle, "The Cydra 5 departmental supercomputer: Design philosophies, decisions, and tradeoffs," *IEEE Comput.*, vol. 22, pp. 12-35, Jan. 1989.
- [49] R. M. Russel, "The CRAY-1 computer system," *Commun. ACM*, vol. 21, pp. 63-72, Jan. 1978.
- [50] H. Schorr, "Design principles for a high performance system," in *Proc. Symp. on Computers and Automata*, New York, NY, Apr. 1971, pp. 165-192.
- [51] M. Slater, "AMD's K5 designed to outrun Pentium," *Microprocessor Rep.*, pp. 1, 6-11, Oct. 24, 1994.
- [52] A. J. Smith, "Cache memories," *ACM Comput. Surveys*, vol. 14, pp. 473-530, Sept. 1982.
- [53] J. E. Smith, "A study of branch prediction strategies," in *Proc. 8th Annu. Symp. on Comput. Architecture*, pp. 135-148, May 1981.
- [54] J. E. Smith *et al.*, "The ZS-1 central processor," in *Proc. Architectural Support for Programming Languages and Operating Syst. (ASPLOS-II)*, Oct. 1987, pp. 199-204.
- [55] J. E. Smith and A. R. Pleszkun, "Implementing precise interrupts in pipelined processors," *IEEE Trans. Comput.*, vol. 37, pp. 562-573, May 1988.
- [56] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," in *Proc. Architectural Support for Programming Languages and Operating Syst. (ASPLOS-III)*, 1989, pp. 290-302.
- [57] G. S. Sohi, "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers," *IEEE Trans. Comput.*, vol. 39, pp. 349-359, Mar. 1990.
- [58] G. S. Sohi and M. Franklin, "High-bandwidth data memory systems for superscalar processors," in *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Apr. 1991, pp. 53-62.
- [59] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proc. 22nd Annu. Int. Symp. on Computer Architecture*, June 1995, pp. 414-425.
- [60] A. R. Talcott *et al.*, "The impact of unresolved branches on branch prediction performance," in *Proc. 21st Annu. Int. Symp. on Comput. Architecture*, Chicago, IL, Apr. 1994, pp. 12-21.
- [61] J. E. Thornton, "Parallel operation in the control data 6600," *Fall Joint Comput. Conf.*, vol. 26, pp. 33-40, 1961.
- [62] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of independent instructions," *IEEE Trans. Computers*, vol. C-19, pp. 889-895, Oct. 1970.
- [63] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Develop.*, pp. 25-33, Jan. 1967.
- [64] D. W. Wall, "Limits of instruction-level parallelism," in *Proc. Architectural Support for Programming Languages and Operating Syst. (ASPLOS-IV)*, Apr. 1991, pp. 176-188.
- [65] S. Weiss and J. E. Smith, *Power and PowerPC: Principles, Architecture, Implementation*. San Francisco, CA: Morgan Kaufmann, 1994.
- [66] T. Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive training branch prediction," in *Proc. 19th Annu. Int. Symp. on Comput. Architecture*, May 1992, pp. 124-134.



James E. Smith (Member, IEEE) received the B.S., M.S., and Ph.D. degrees from the University of Illinois in 1972, 1974, and 1976, respectively.

In 1976 he joined the faculty of the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison, where he has taught and conducted research in fault-tolerant computing and computer architecture. From 1979 to 1981, he took a leave of absence to work for the Control Data Corporation in Arden

Hills, MN, participating in the design of the CYBER 180/990. While at Control Data and after returning to the University of Wisconsin, he studied several aspects of pipelined implementations including branch prediction, instruction issuing methods, and precise interrupt techniques. From 1984 to 1989, he took a second leave of absence to participate in a startup company, Astronautics Corporation of America. At Astronautics, he was the principal architect for the ZS-1, a scientific computer employing a dynamically scheduled, superscalar processor architecture. In 1989, he joined Cray Research, Inc., Chippewa Falls, WI. While at Cray, he headed a small research team that participated in the development and analysis of future supercomputer architectures. In 1994, he rejoined the Department of ECE at the University of Wisconsin, where he is now Professor. His current research interests focus on new paradigms for exploiting instruction level parallelism.



Gurindar S. Sohi (Senior Member, IEEE) received the Ph.D. degree in electrical and computer engineering from the University of Illinois at Urbana-Champaign in 1985.

He is currently an Associate Professor in the Computer Sciences Department at the University of Wisconsin, Madison, WI. His research interests center on computer architecture, with an emphasis in fine-grain parallel architectures, supercomputers, and memory systems.



CS252

Graduate Computer Architecture

Lecture 10

ILP Limits Multithreading

John Kubitowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

<http://www.eecs.berkeley.edu/~kubitron/cs252>
<http://www-inst.eecs.berkeley.edu/~cs252>



Limits to ILP

- **Conflicting studies of amount**
 - Benchmarks (vectorized Fortran FP vs. integer C programs)
 - Hardware sophistication
 - Compiler sophistication
- **How much ILP is available using existing mechanisms with increasing HW budgets?**
- **Do we need to invent new HW/SW mechanisms to keep on processor performance curve?**
 - Intel MMX, SSE (Streaming SIMD Extensions): 64 bit ints
 - Intel SSE2: 128 bit, including 2 64-bit Fl. Pt. per clock
 - Motorola AltaVec: 128 bit ints and FPs
 - Supersparc Multimedia ops, etc.



Overcoming Limits

- Advances in compiler technology + significantly new and different hardware techniques *may* be able to overcome limitations assumed in studies
- However, unlikely such advances when coupled *with realistic hardware* will overcome these limits in near future



Limits to ILP

Initial HW Model here; MIPS compilers.

Assumptions for ideal/perfect machine to start:

1. ***Register renaming*** – infinite virtual registers
=> all register WAW & WAR hazards are avoided
2. ***Branch prediction*** – perfect; no mispredictions
3. ***Jump prediction*** – all jumps perfectly predicted
(returns, case statements)
2 & 3 \Rightarrow no control dependencies; perfect speculation
& an unbounded buffer of instructions available
4. ***Memory-address alias analysis*** – addresses known
& a load can be moved before a store provided
addresses not equal; 1&4 eliminates all but RAW

Also: perfect caches; 1 cycle latency for all instructions
(FP *, /); unlimited instructions issued/clock cycle;



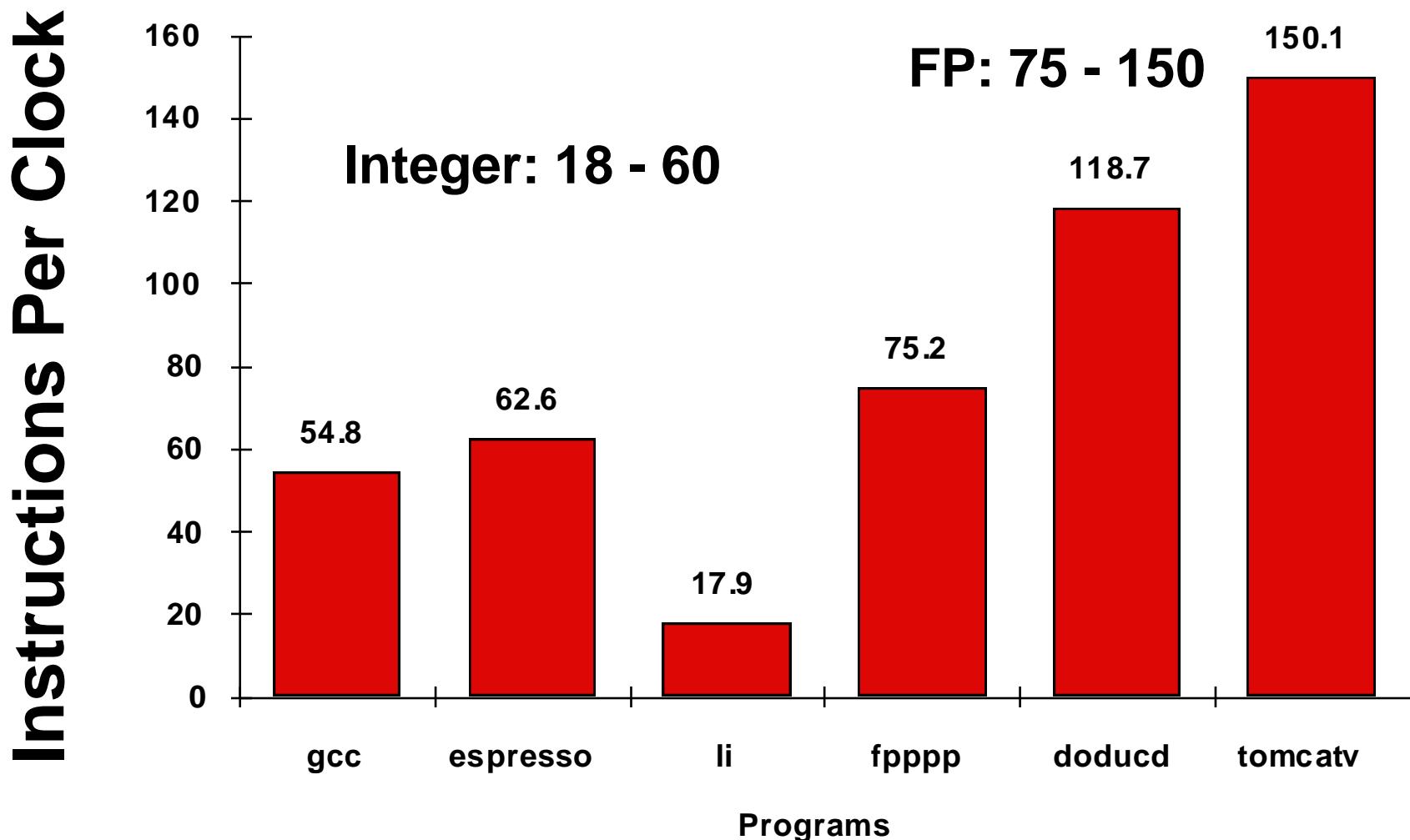
Limits to ILP HW Model comparison

	Model	Power 5
Instructions Issued per clock	Infinite	4
Instruction Window Size	Infinite	200
Renaming Registers	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	Perfect	2% to 6% misprediction (Tournament Branch Predictor)
Cache	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias Analysis	Perfect	??



Upper Limit to ILP: Ideal Machine

(Figure 3.1)





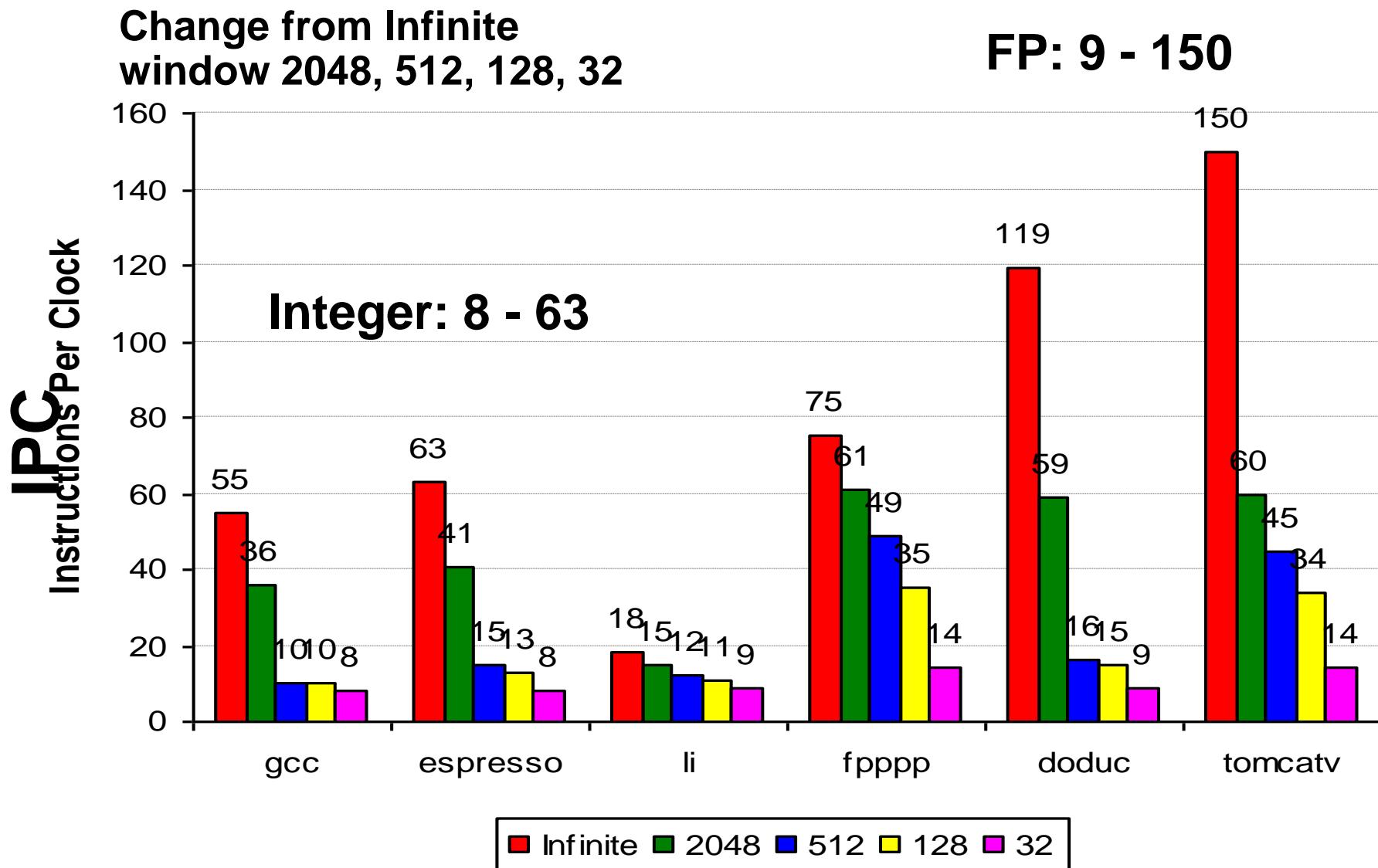
Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	Infinite	Infinite	4
Instruction Window Size	Infinite, 2K, 512, 128, 32	Infinite	200
Renaming Registers	Infinite	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	Perfect	Perfect	2% to 6% misprediction (Tournament Branch Predictor)
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect	Perfect	??



More Realistic HW: Window Impact

Figure 3.2





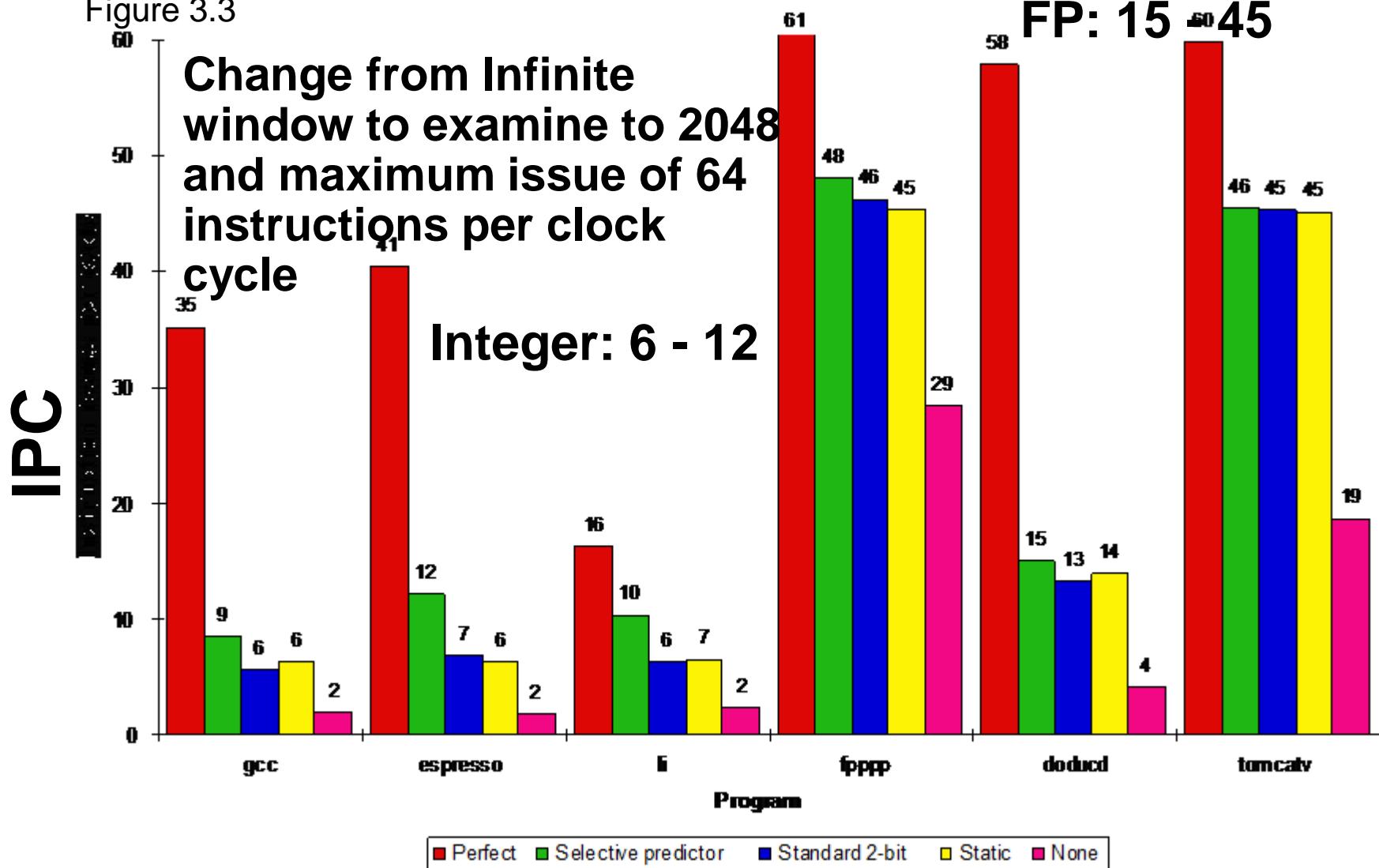
Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	64	Infinite	4
Instruction Window Size	2048	Infinite	200
Renaming Registers	Infinite	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	Perfect vs. 8K Tournament vs. 512 2-bit vs. profile vs. none	Perfect	2% to 6% misprediction (Tournament Branch Predictor)
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect	Perfect	??



More Realistic HW: Branch Impact

Figure 3.3



Perfect

Tournament

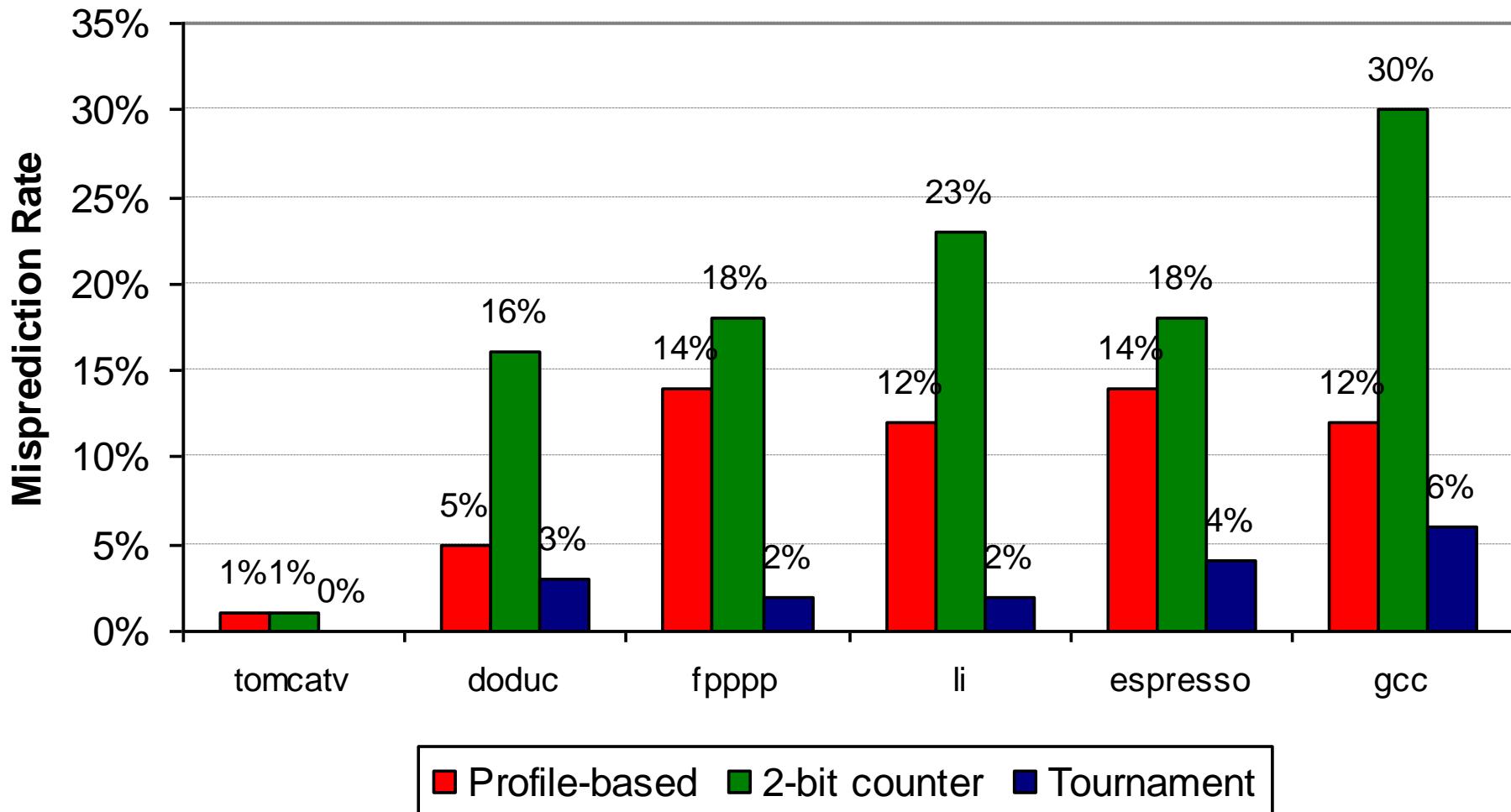
BHT (512)

Profile

No prediction



Misprediction Rates





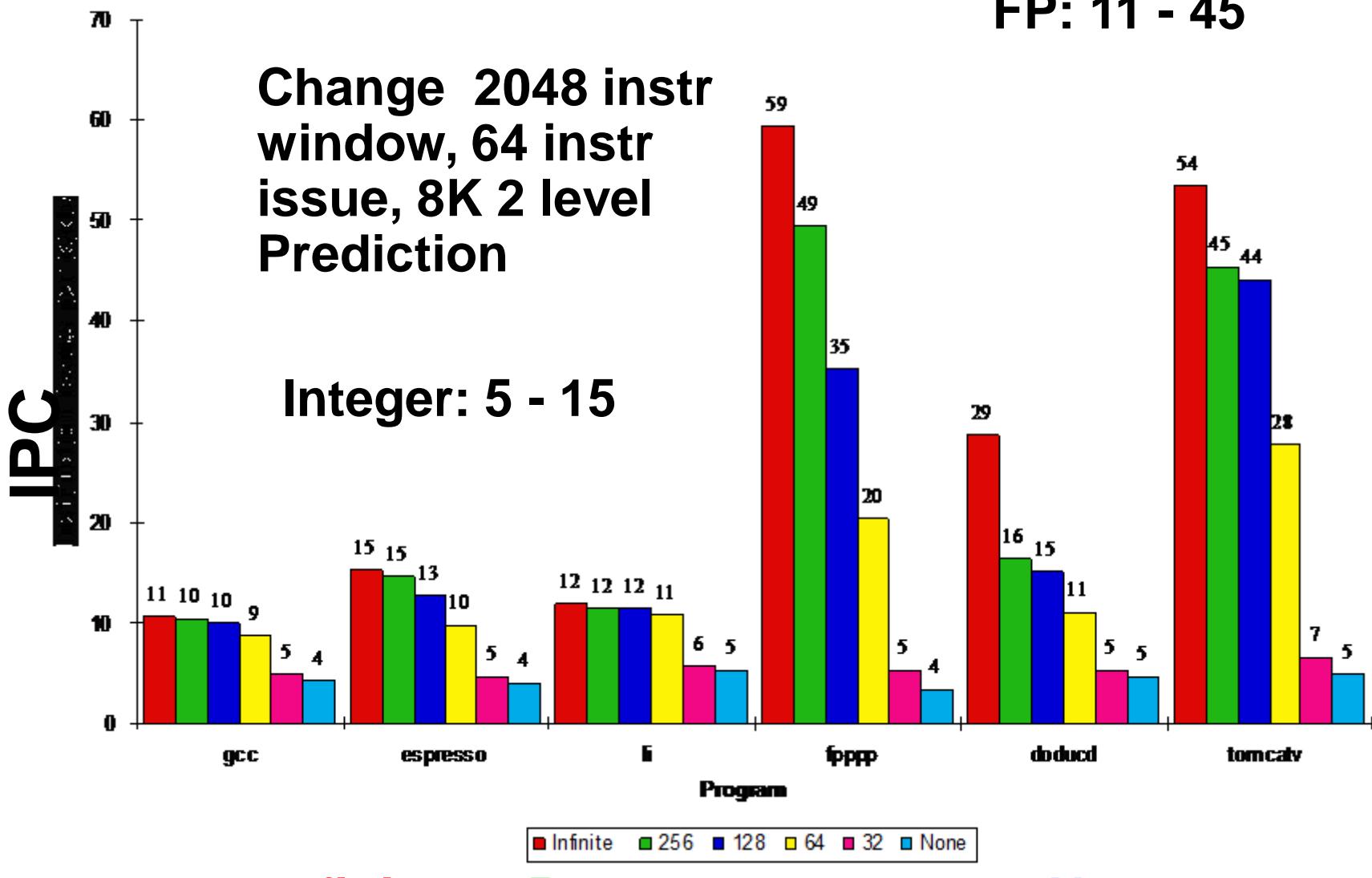
Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	64	Infinite	4
Instruction Window Size	2048	Infinite	200
Renaming Registers	Infinite v. 256, 128, 64, 32, none	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	8K 2-bit	Perfect	Tournament Branch Predictor
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect	Perfect	Perfect



More Realistic HW: Renaming Register Impact (N int + N fp)

Figure 3.5





Limits to ILP HW Model comparison

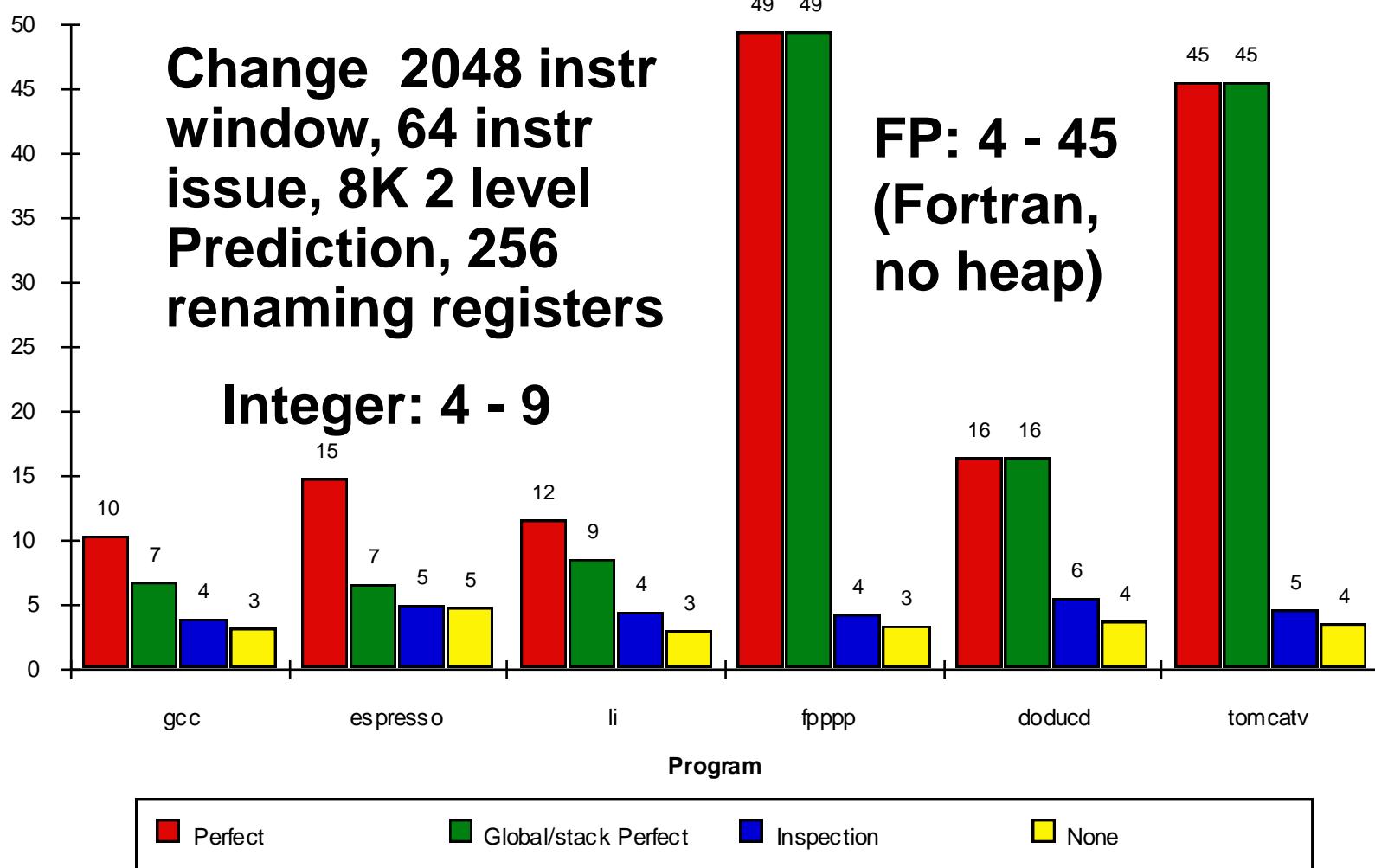
	New Model	Model	Power 5
Instructions Issued per clock	64	Infinite	4
Instruction Window Size	2048	Infinite	200
Renaming Registers	256 Int + 256 FP	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	8K 2-bit	Perfect	Tournament
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect v. Stack v. Inspect v. none	Perfect	Perfect



More Realistic HW: Memory Address Alias Impact

Figure 3.6

IPC



Perfect Global/Stack perf; Inspec.
 heap conflicts None
 3252407, Lecture 10 Assem.



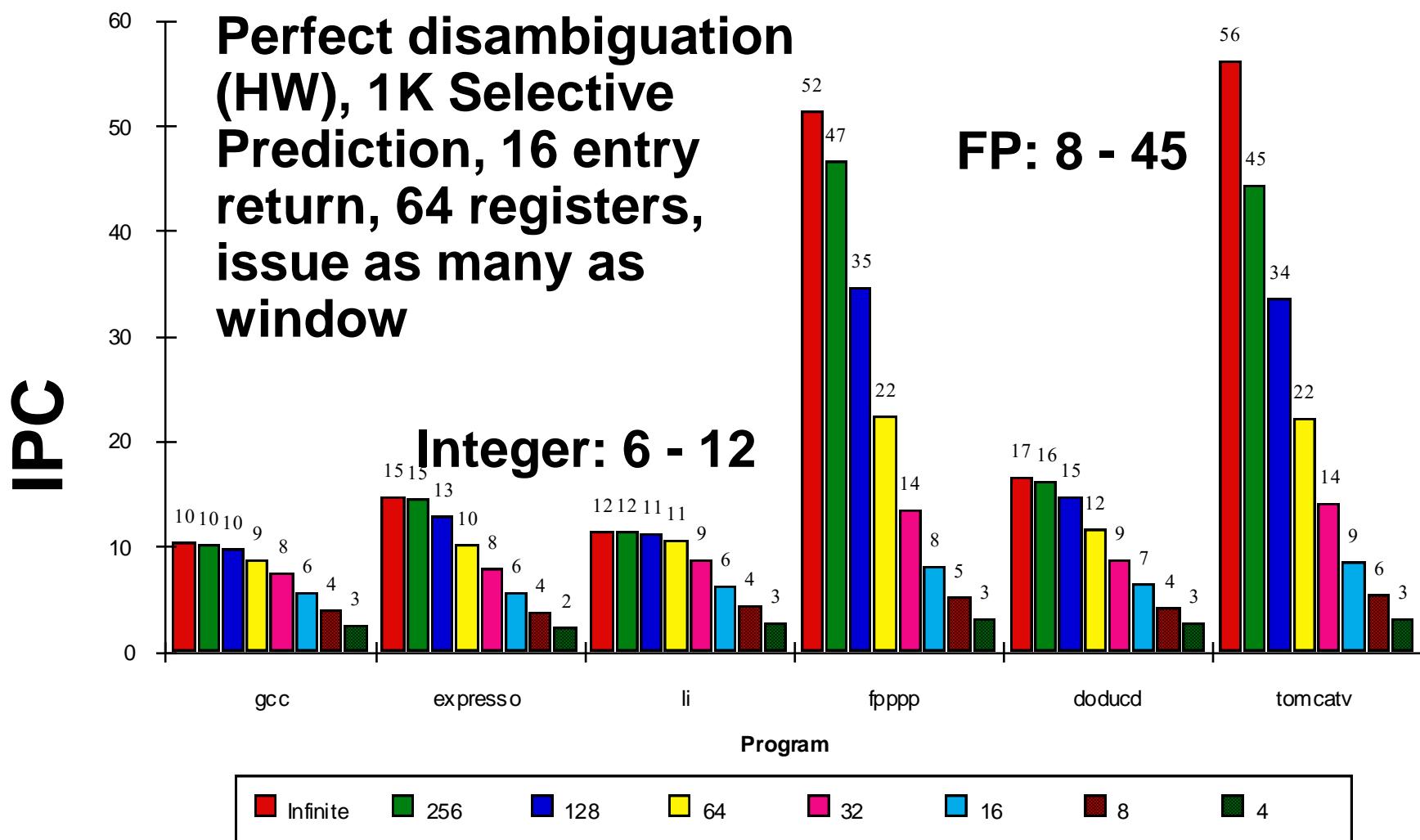
Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	64 (no restrictions)	Infinite	4
Instruction Window Size	Infinite vs. 256, 128, 64, 32	Infinite	200
Renaming Registers	64 Int + 64 FP	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	1K 2-bit	Perfect	Tournament
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	HW disambiguation	Perfect	Perfect



Realistic HW: Window Impact

(Figure 3.7)



Infinite 256 128 64 32 16 8 4



How to Exceed ILP Limits of this study?

- These are not laws of physics; just practical limits for today, and perhaps overcome via research
- Compiler and ISA advances could change results
- WAR and WAW hazards through memory: eliminated WAW and WAR hazards through register renaming, but not in memory usage
 - Can get conflicts via allocation of stack frames as a called procedure reuses the memory addresses of a previous frame on the stack



HW v. SW to increase ILP

- **Memory disambiguation:** HW best
- **Speculation:**
 - HW best when dynamic branch prediction better than compile time prediction
 - Exceptions easier for HW
 - HW doesn't need bookkeeping code or compensation code
 - Very complicated to get right
- **Scheduling:** SW can look ahead to schedule better
- **Compiler independence:** does not require new compiler, recompilation to run well



Performance beyond single thread ILP

- There can be much higher natural parallelism in some applications (e.g., Database or Scientific codes)
- **Explicit Thread Level Parallelism or Data Level Parallelism**
- **Thread:** process with own instructions and data
 - thread may be a process part of a parallel program of multiple processes, or it may be an independent program
 - Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute
- **Data Level Parallelism:** Perform identical operations on data, and lots of data



Administrivia

- **Exam: Wednesday 3/14**
Location: TBA
TIME: 5:30 - 8:30
- **This info is on the Lecture page (has been)**
- **Meet at LaVal's afterwards for Pizza and Beverages**
- **CS252 Project proposal due by Monday 3/5**
 - Need two people/project (although can justify three for right project)
 - Complete Research project in 8 weeks
 - » Typically investigate hypothesis by building an artifact and measuring it against a “base case”
 - » Generate conference-length paper/give oral presentation
 - » Often, can lead to an actual publication.



Project opportunity this semester (RAMP)

- FPGAs as New Research Platform
- As ~ 25 CPUs can fit in Field Programmable Gate Array (FPGA), 1000-CPU system from ~ 40 FPGAs?
 - 64-bit simple “soft core” RISC at 100MHz in 2004 (Virtex-II)
 - FPGA generations every 1.5 yrs; 2X CPUs, 2X clock rate
- HW research community does logic design (“gate shareware”) to create out-of-the-box, Massively Parallel Processor runs standard binaries of OS, apps
 - Gateware: Processors, Caches, Coherency, Ethernet Interfaces, Switches, Routers, ... (IBM, Sun have donated processors)
 - E.g., 1000 processor, IBM Power binary-compatible, cache-coherent supercomputer @ 200 MHz; fast enough for research
- Research Accelerator for Multiple Processors (RAMP)
 - To learn more, read “RAMP: Research Accelerator for Multiple Processors - A Community Vision for a Shared Experimental Parallel HW/SW Platform,” Technical Report UCB//CSD-05-1412, Sept 2005
 - Web page ramp.eecs.berkeley.edu



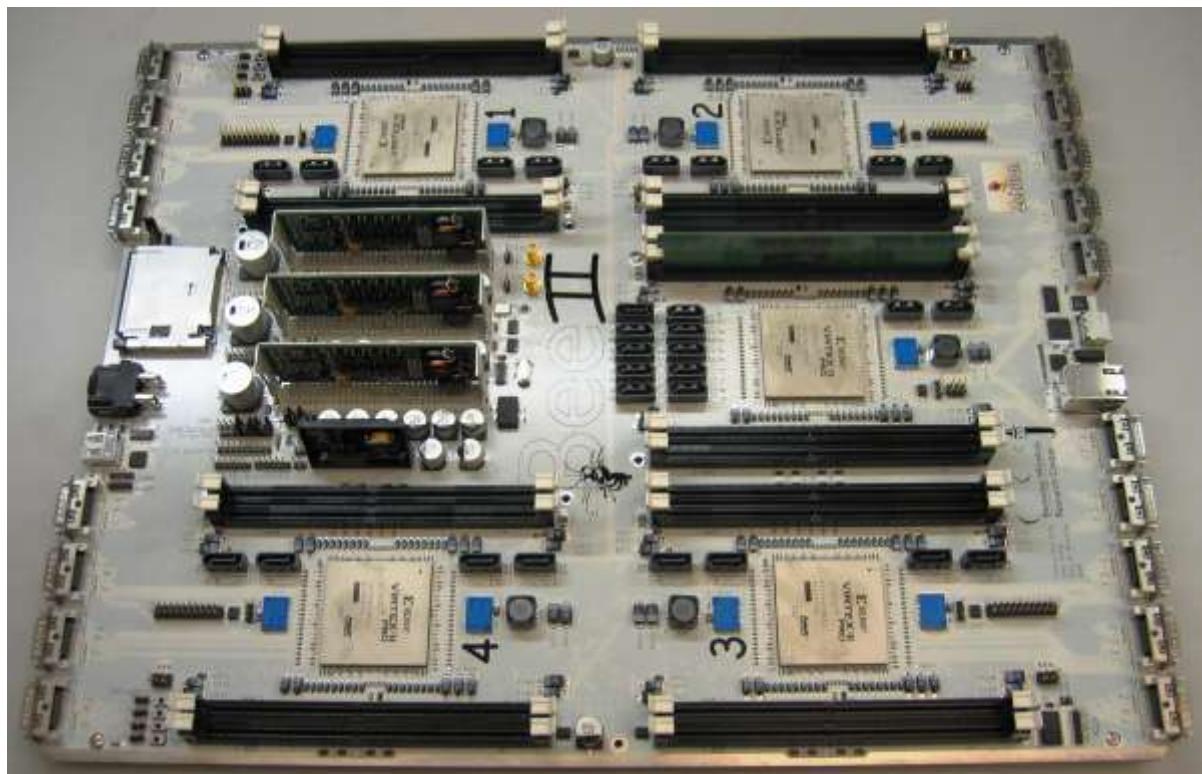
Why RAMP Good for Research?

	SMP	Cluster	Simulate	RAMP
Cost (1000 CPUs)	F (\$40M)	C (\$2M)	A+ (\$0M)	A (\$0.1M)
Cost of ownership	A	D	A	A
Scalability	C	A	A	A
Power/Space (kilowatts, racks)	D (120 kw, 12 racks)	D (120 kw, 12 racks)	A+ (.1 kw, 0.1 racks)	A (1.5 kw, 0.3 racks)
Community	D	A	A	A
Observability	D	C	A+	A+
Reproducibility	B	D	A+	A+
Flexibility	D	C	A+	A+
Credibility	A+	A+	F	A
Perform. (clock)	A (2 GHz)	A (3 GHz)	F (0 GHz)	C (0.2 GHz)
GPA	C	B-	B	A-



RAMP 1 Hardware

- Completed Dec. 2004 (14x17 inch 22-layer PCB)
- Module:
 - FPGAs, memory, 10GigE conn.
 - Compact Flash
 - Administration/maintenance ports:
 - » 10/100 Enet
 - » HDMI/DVI
 - » USB
 - ~4K/module w/o FPGAs or DRAM
- Called “BEE2” for Berkeley Emulation Engine 2

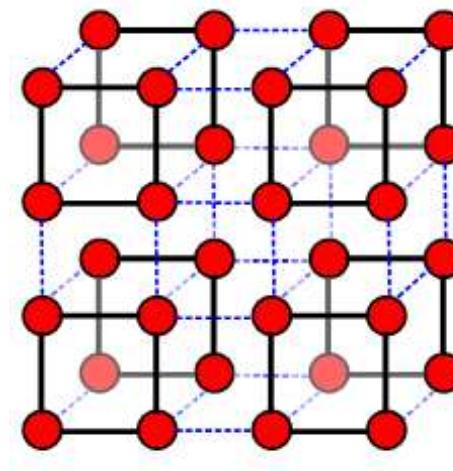




Multiple Module RAMP 1 Systems



- **8 compute modules (plus power supplies) in 8U rack mount chassis**
 - 500-1000 emulated processors
- **Many topologies possible**
- **2U single module tray for developers**
- **Disk storage: disk emulator + Network Attached Storage**



- **FPGA**
- **MGT link**
- **LVCMOS link**



Vision: Multiprocessing Watering Hole



Parallel file system Dataflow language/computer Data center in a box
Thread scheduling Security enhancements **Internet in a box**
Multiprocessor switch design Router design Compile to FPGA
Fault insertion to check dependability Parallel languages

- **RAMP attracts many communities to shared artifact**
⇒ **Cross-disciplinary interactions**
⇒ **Accelerate innovation in multiprocessing**
- **RAMP as next Standard Research Platform?**
(e.g., VAX/BSD Unix in 1980s, x86/Linux in 1990s)



RAMP Summary

- **RAMP as system-level time machine: preview computers of future to accelerate HW/SW generations**
 - Trace anything, Reproduce everything, Tape out every day
 - FTP new supercomputer overnight and boot in morning
 - Clone to check results (as fast in Berkeley as in Boston?)
 - Emulate Massive Multiprocessor, Data Center, or Distributed Computer
- **Carpe Diem**
 - Systems researchers (HW & SW) need the capability
 - FPGA technology is ready today, and getting better every year
 - Stand on shoulders vs. toes: standardize on multi-year Berkeley effort on FPGA platform Berkeley Emulation Engine 2 (BEE2)
 - See ramp.eecs.berkeley.edu
- **Vision “Multiprocessor Research Watering Hole” accelerate research in multiprocessing via standard research platform
⇒ hasten sea change from sequential to parallel computing**



RAMP projects for CS 252

- **Design a guest timing accounting strategy**
 - Want to be able to specify performance parameters (clock rate, memory latency, network latency, ...)
 - Host must accurately account for guest clock cycles
 - Don't want to slow down host execution time very much
- **Build a disk emulator for use in RAMP**
 - Imitates disk, accesses network attached storage for data
 - Modeled after guest VM/driver VM from Xen VM?
- **Build a cluster using components from opencores.org on BEE2**
 - Open source hardware consortium
- **Build an emulator of an “Internet in a Box”**
 - (Emulab/Planetlab in a box is closer to reality)
- **(e.g., sparse matrix, structured grid), some are more open (e.g., FSM).**



More RAMP projects

RAMP Blue is a family of emulated message-passing machines, which can be used to run parallel applications written for the Message-Passing Interface (MPI) standard, or for partitioned global address space languages such as Unified Parallel C (UPC).

- **Investigation of Leon Sparc Core:**
 - The Leon core, was developed to target a variety of implementation platforms (ASIC, custom, etc.) and is not highly optimized for FPGA implementations (it is currently 4X the number of LUTs as the Xilinx Microblaze).
 - A project would be to optimize the Leon FPGA implementation, and put it into the RDL (RAMP Design Language) framework, and integrate it into RAMP Blue.
- **BEEKeeper remote management for RAMP Blue:**
 - Managing a cluster of many FPGA boards is hard. Provide hardware and software support for remote serial and JTAG functionality (programming and debugging) using one such board. The board will be provided.
- **Remote DMA engine/Network Interface for RAMP Blue:**
 - We have a high-performance shared-memory language (UPC) and a high-performance switched network implemented and fully functional. Bridge the gap between the two by providing hardware and software support for remote DMA.

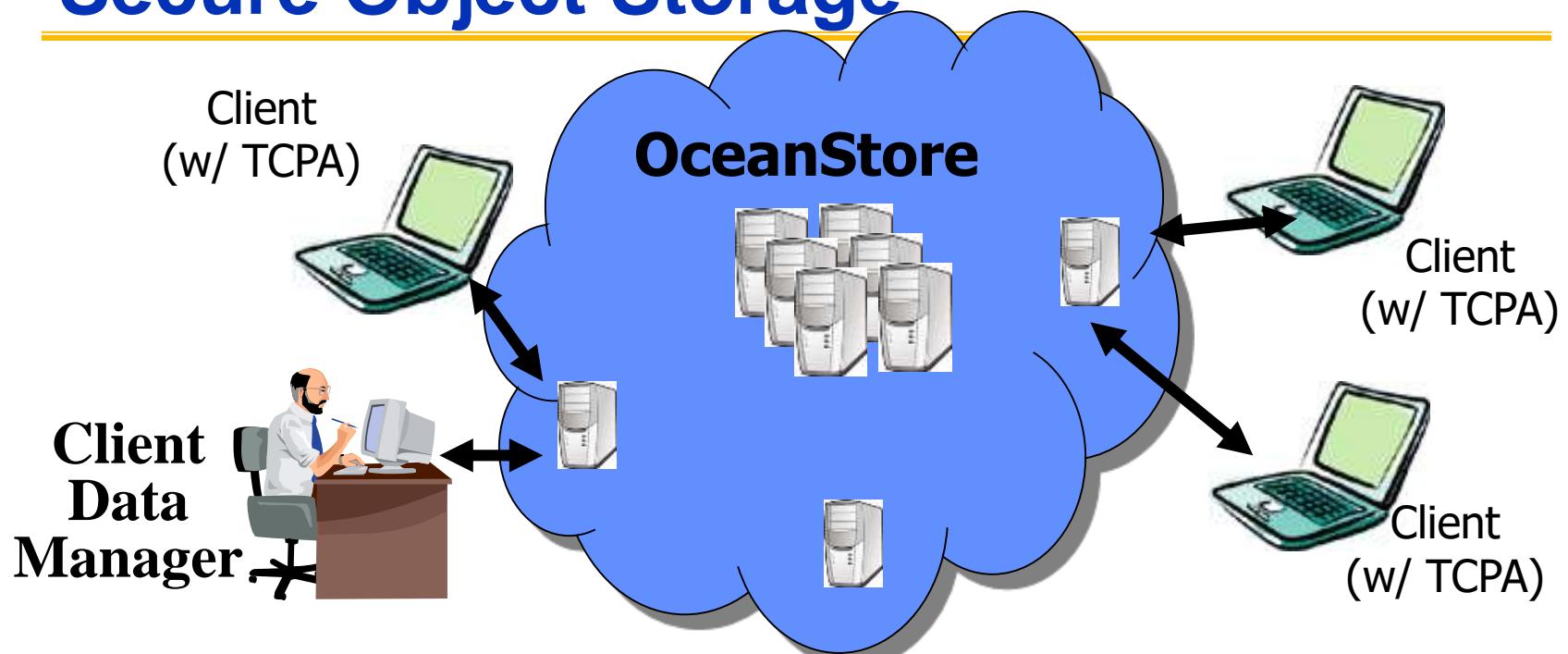


Other projects

- **Recreate results from important research paper to see**
 - If they are reproducible
 - If they still hold
- **13 dwarfs as benchmarks:** Patterson et al. specified a set of 13 kernels they believe are important to future use of parallel machines
 - Since they don't want to specify the code in detail, leaving that up to the designers, one approach would be to create data sets (or a data set generator) for each dwarf, so that you could have a problem to solve of the appropriate size.
 - You'd probably like to be able to pick floating point format or fixed point format. Some are obvious(e.g., dense linear algebra), some are pretty well understood
 - See view.eecs.berkeley.edu
- **Develop and evaluate new parallel communication model**
 - Target for Multicore systems
- **Quantum CAD tools**
 - Develop mechanisms to aid in the automatic generation, placement, and *verification* of quantum computing architectures



Secure Object Storage



- **Security: Access and Content controlled by client**
 - Privacy through data encryption
 - Optional use of cryptographic hardware for revocation
 - Authenticity through hashing and active integrity checking
- **PROJECT: Investigate how secure hardware (such as included in IBM laptops) can be utilized for:**
 - High-performance access to encrypted data
 - Easy revocation of access.



Thread Level Parallelism (TLP)

- **ILP exploits implicit parallel operations within a loop or straight-line code segment**
- **TLP explicitly represented by the use of multiple threads of execution that are inherently parallel**
- **Goal: Use multiple instruction streams to improve**
 1. **Throughput of computers that run many programs**
 2. **Execution time of multi-threaded programs**
- **TLP could be more cost-effective to exploit than ILP**



Another Approach: Multithreaded Execution

- **Multithreading: multiple threads to share the functional units of 1 processor via overlapping**
 - processor must duplicate independent state of each thread e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
 - memory shared through the virtual memory mechanisms, which already support multiple processes
 - HW for fast thread switch; much faster than full process switch \approx 100s to 1000s of clocks
- **When switch?**
 - Alternate instruction per thread (fine grain)
 - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)



Fine-Grained Multithreading

- **Switches between threads on each instruction, causing the execution of multiple threads to be interleaved**
- **Usually done in a round-robin fashion, skipping any stalled threads**
- **CPU must be able to switch threads every clock**
- **Advantage is it can hide both short and long stalls, since instructions from other threads are executed when one thread stalls**
- **Disadvantage is it slows down execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads**
- **Used on Sun's Niagara (will see later)**

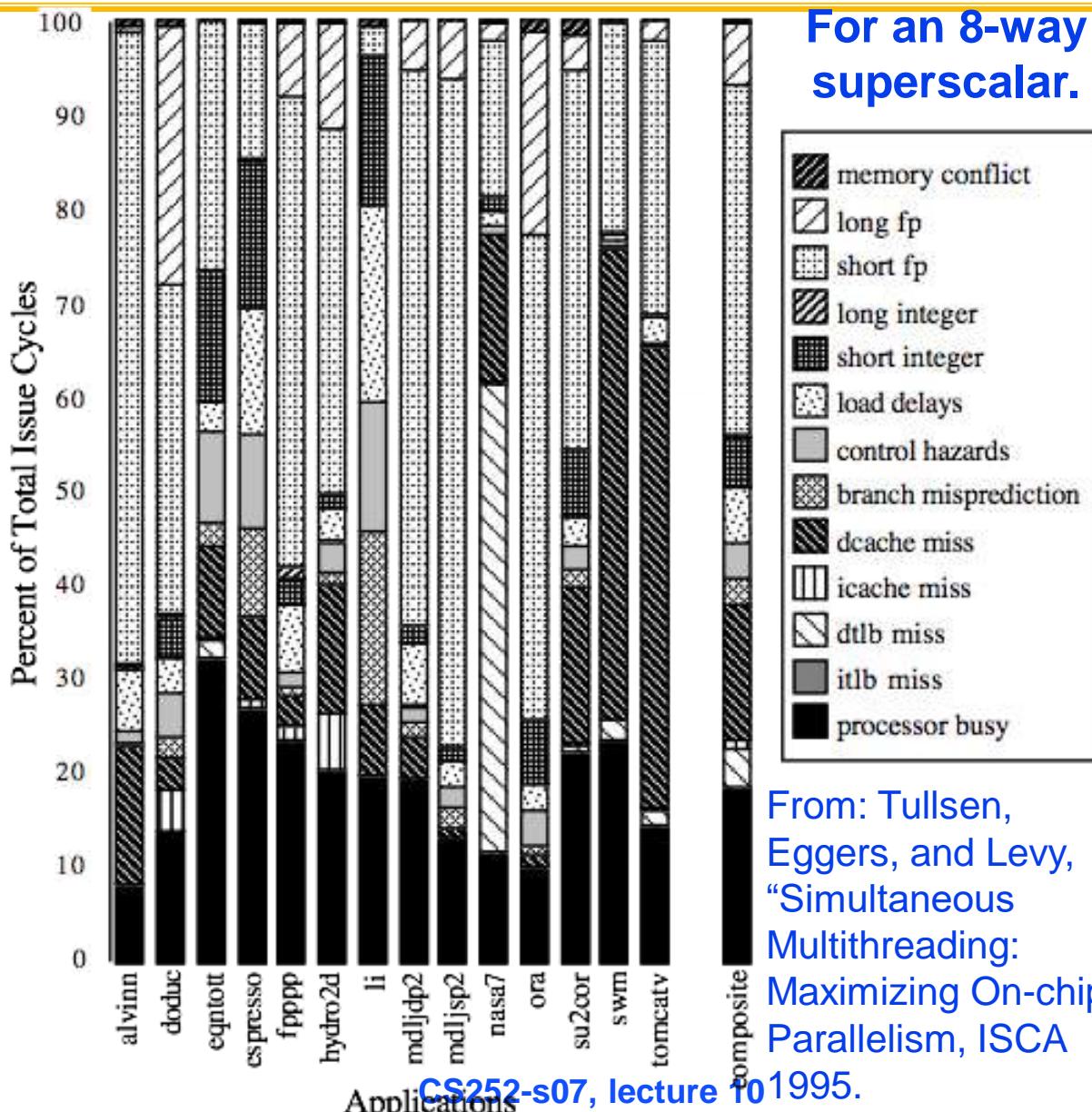


Course-Grained Multithreading

- **Switches threads only on costly stalls, such as L2 cache misses**
- **Advantages**
 - Relieves need to have very fast thread-switching
 - Doesn't slow down thread, since instructions from other threads issued only when the thread encounters a costly stall
- **Disadvantage is hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs**
 - Since CPU issues instructions from 1 thread, when a stall occurs, the pipeline must be emptied or frozen
 - New thread must fill pipeline before instructions can complete
- **Because of this start-up overhead, coarse-grained multithreading is better for reducing penalty of high cost stalls, where pipeline refill << stall time**
- **Used in IBM AS/400**



For most apps: most execution units lie idle





Do both ILP and TLP?

- **TLP and ILP exploit two different kinds of parallel structure in a program**
- **Could a processor oriented at ILP to exploit TLP?**
 - functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
- **Could the TLP be used as a source of independent instructions that might keep the processor busy during stalls?**
- **Could TLP be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?**



Simultaneous Multi-threading ...

One thread, 8 units

Cycle M M FX FX FP FP BR CC

1	Y								Y
2	Y					Y			
3			Y	Y					
4									
5									
6									
7	Y			Y	Y				
8		Y			Y				
9			Y						

M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

Two threads, 8 units

Cycle M M FX FX FP FP BR CC

1	Y								Y
2	Y			Y					B
3		B							Y
4		B							B
5			B						B
6									
7	Y					B	Y	Y	
8			Y				B	Y	B
9	B						Y		B



Simultaneous Multithreading (SMT)

- **Simultaneous multithreading (SMT):** insight that dynamically scheduled processor already has many HW mechanisms to support multithreading
 - Large set of virtual registers that can be used to hold the register sets of independent threads
 - Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in datapath without confusing sources and destinations across threads
 - Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW
- **Just adding a per thread renaming table and keeping separate PCs**
 - Independent commitment can be supported by logically keeping a separate reorder buffer for each thread

Source: Microprocessor Report, December 6, 1999
"Compaq Chooses SMT for Alpha"



Multithreaded Categories





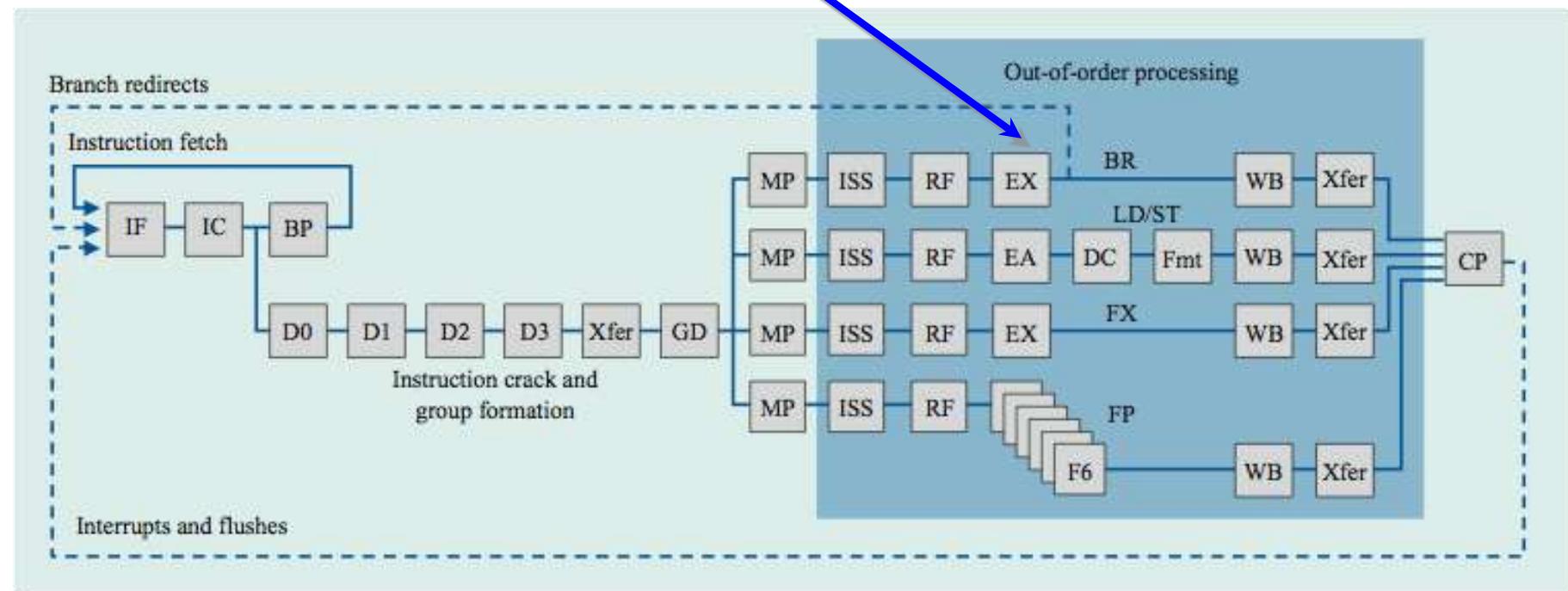
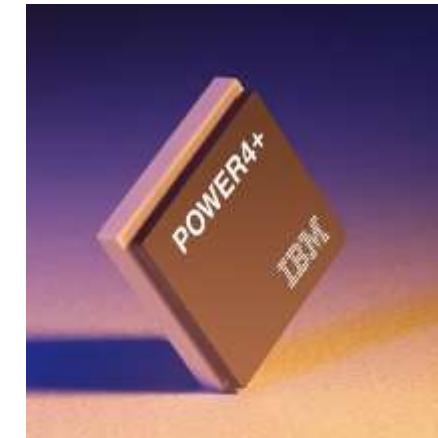
Design Challenges in SMT

- **Since SMT makes sense only with fine-grained implementation, impact of fine-grained scheduling on single thread performance?**
 - A preferred thread approach sacrifices neither throughput nor single-thread performance?
 - Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput, when preferred thread stalls
- **Larger register file needed to hold multiple contexts**
- **Clock cycle time, especially in:**
 - Instruction issue - more candidate instructions need to be considered
 - Instruction completion - choosing which instructions to commit may be challenging
- **Ensuring that cache and TLB conflicts generated by SMT do not degrade performance**



Power 4

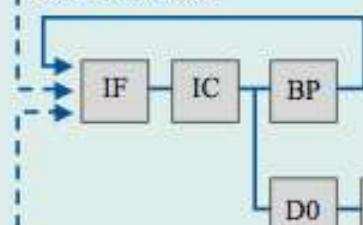
Single-threaded predecessor to Power 5. 8 execution units in out-of-order engine, each may issue an instruction each cycle.



Power 4

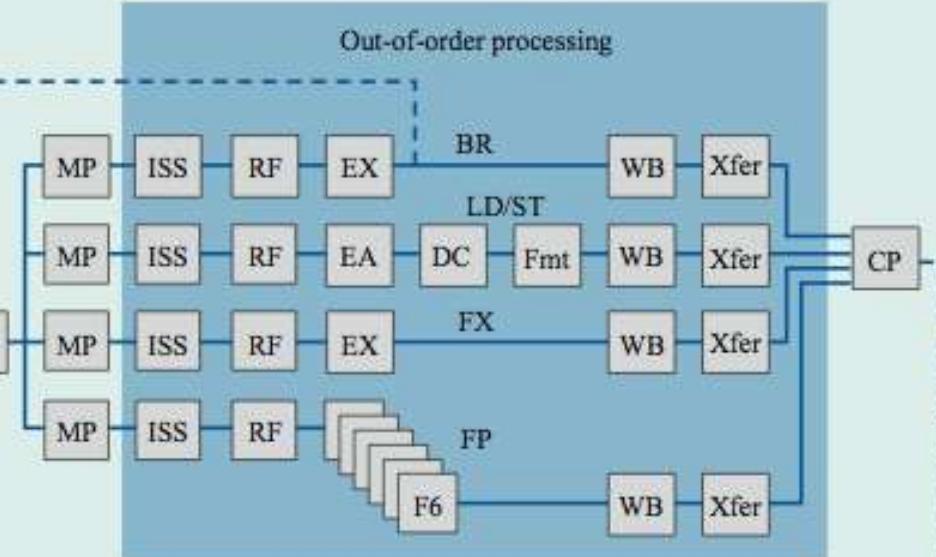
Branch redirects

Instruction fetch



Instruction crack and group formation

Interrupts and flushes

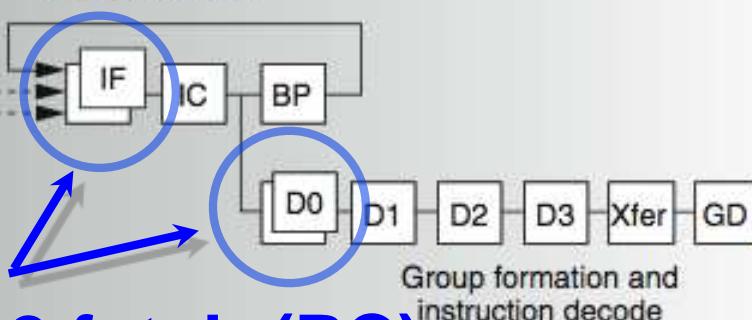


2 commits
(architected
register sets)

Branch redirects

Power 5

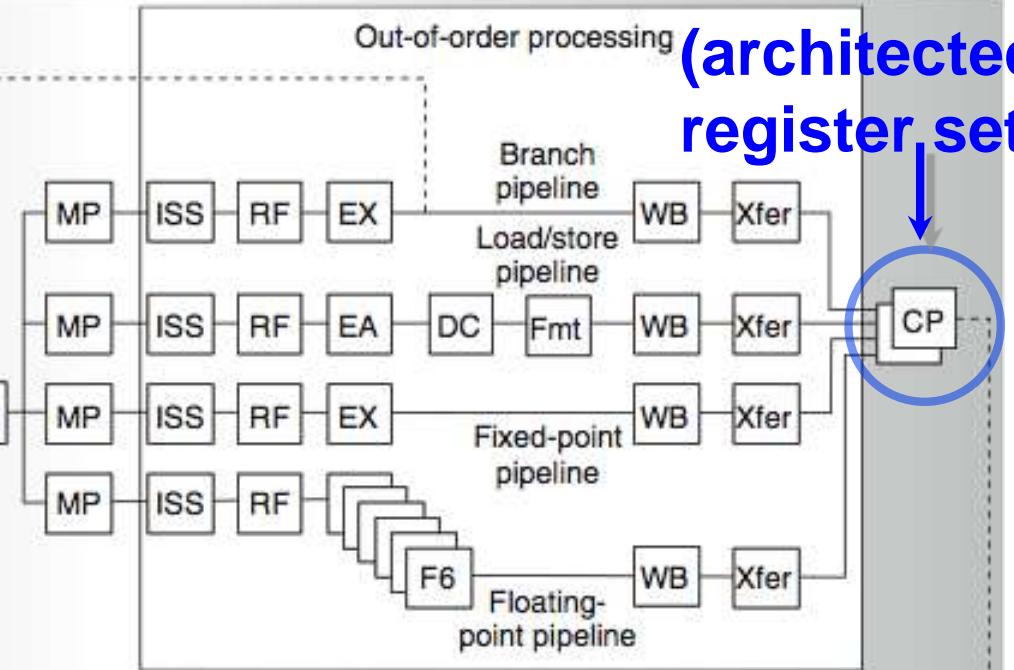
Instruction fetch



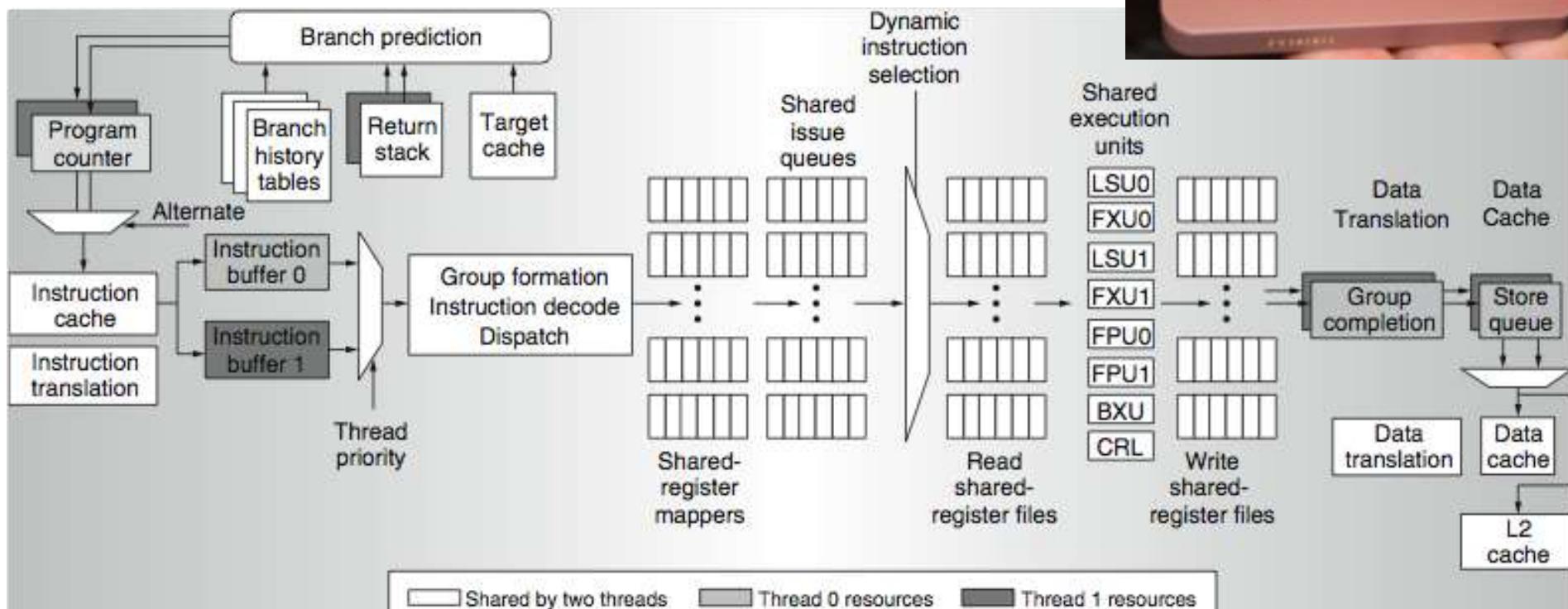
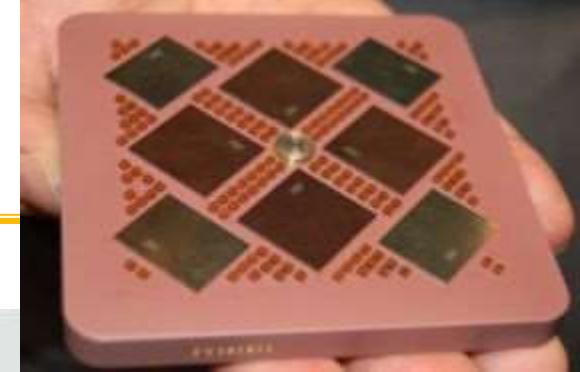
Group formation and instruction decode

2 fetch (PC),
2 initial decodes

Interrupts and flushes



Power 5 data flow ...



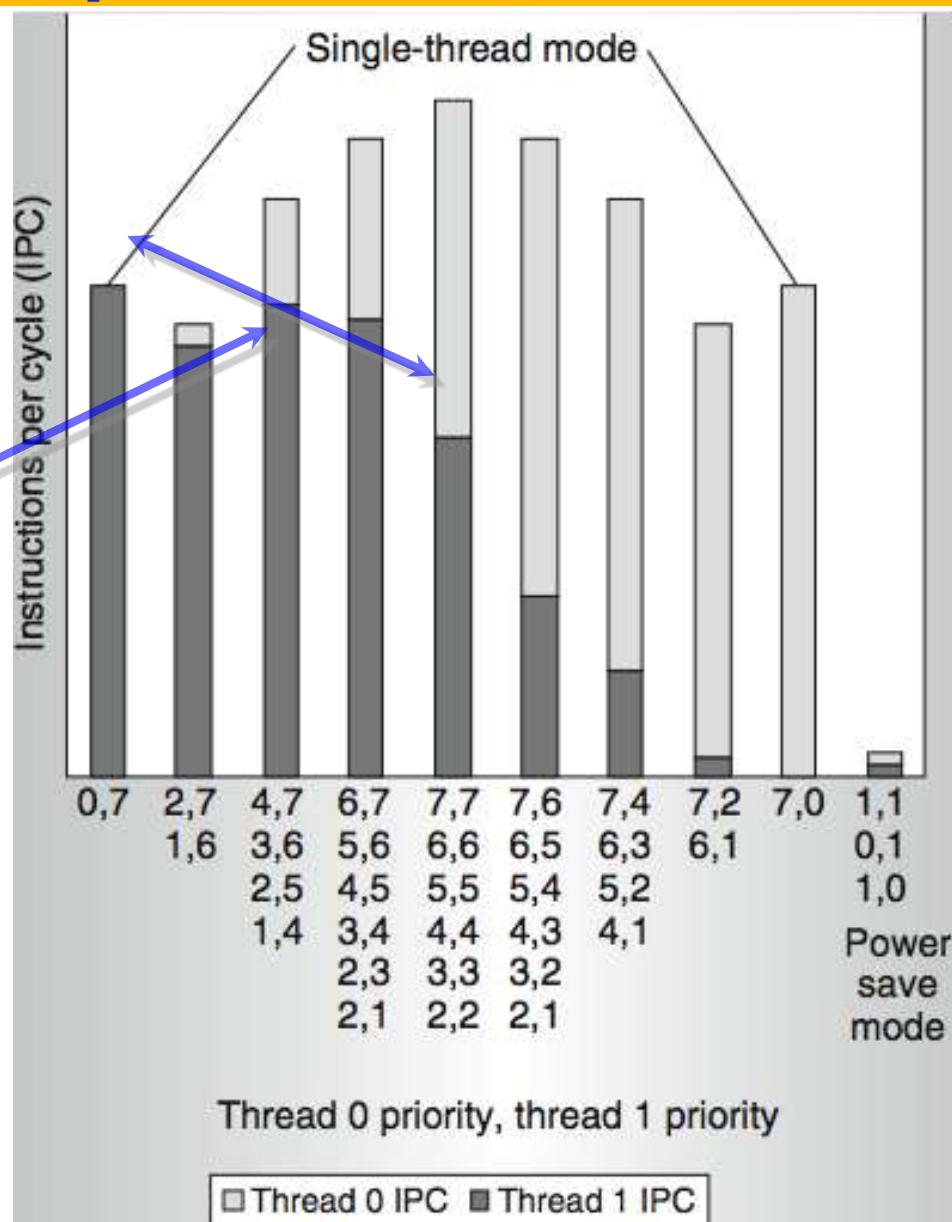
Why only 2 threads? With 4, one of the shared resources (physical registers, cache, memory bandwidth) would be prone to bottleneck



Power 5 thread performance ...

Relative priority of each thread controllable in hardware.

For balanced operation, both threads run slower than if they “owned” the machine.





Changes in Power 5 to support SMT

- Increased associativity of L1 instruction cache and the instruction address translation buffers
- Added per thread load and store queues
- Increased size of the L2 (1.92 vs. 1.44 MB) and L3 caches
- Added separate instruction prefetch and buffering per thread
- Increased the number of virtual registers from 152 to 240
- Increased the size of several issue queues
- The Power5 core is about 24% larger than the Power4 core because of the addition of SMT support



Initial Performance of SMT

- **Pentium 4 Extreme SMT yields 1.01 speedup for SPECint_rate benchmark and 1.07 for SPECfp_rate**
 - Pentium 4 is dual threaded SMT
 - SPECRate requires that each SPEC benchmark be run against a vendor-selected number of copies of the same benchmark
- **Running on Pentium 4 each of 26 SPEC benchmarks paired with every other (26^2 runs) speed-ups from 0.90 to 1.58; average was 1.20**
- **Power 5, 8 processor server 1.23 faster for SPECint_rate with SMT, 1.16 faster for SPECfp_rate**
- **Power 5 running 2 copies of each app speedup between 0.89 and 1.41**
 - Most gained some
 - Fl.Pt. apps had most cache conflicts and least gains



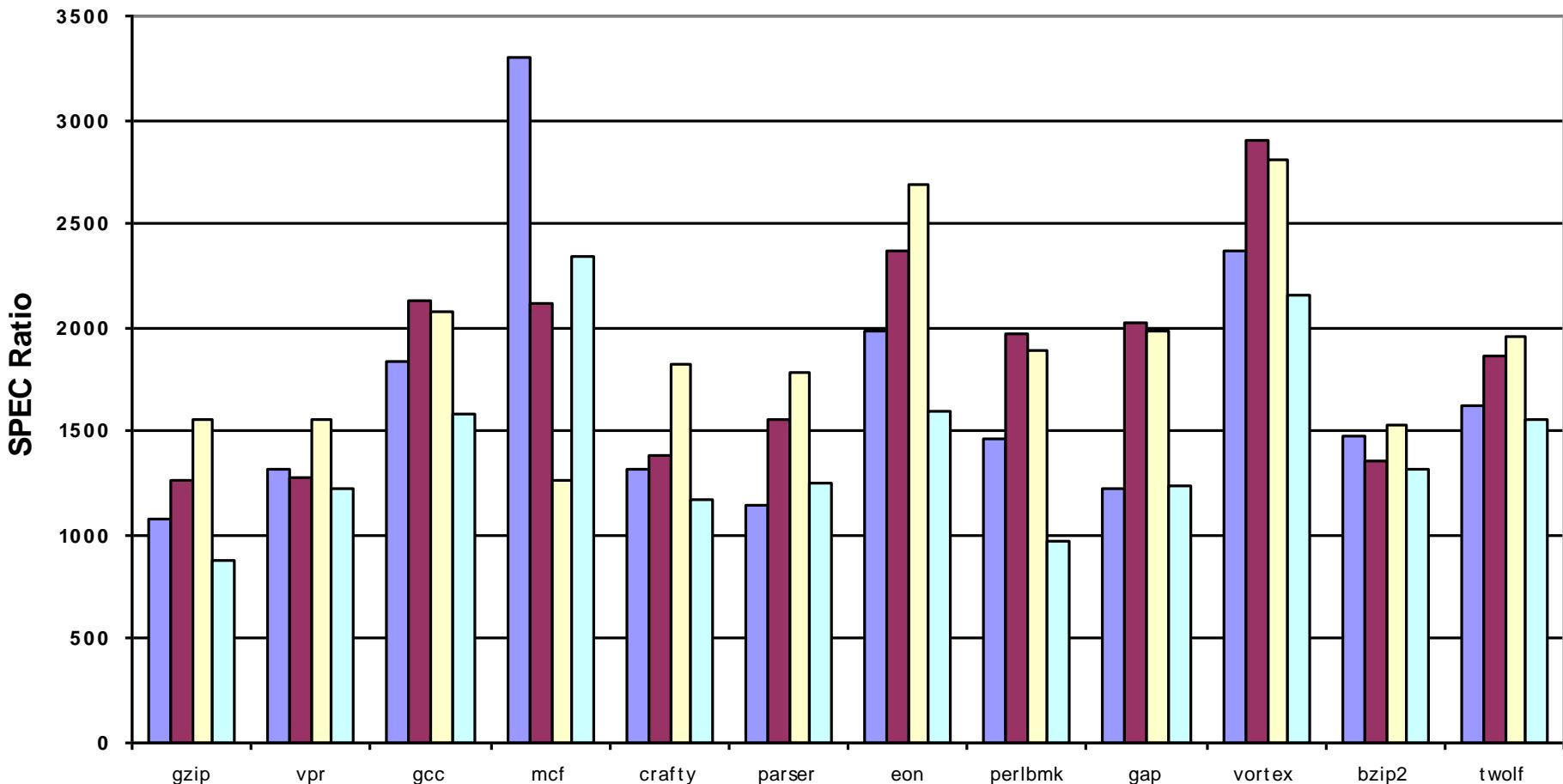
Head to Head ILP competition

Processor	Micro architecture	Fetch / Issue / Execute	FU	Clock Rate (GHz)	Transis-tors Die size	Power
Intel Pentium 4 Extreme	Speculative dynamically scheduled; deeply pipelined; SMT	3/3/4	7 int. 1 FP	3.8	125 M 122 mm ²	115 W
AMD Athlon 64 FX-57	Speculative dynamically scheduled	3/3/4	6 int. 3 FP	2.8	114 M 115 mm ²	104 W
IBM Power5 (1 CPU only)	Speculative dynamically scheduled; SMT; 2 CPU cores/chip	8/4/8	6 int. 2 FP	1.9	200 M 300 mm ² (est.)	80W (est.)
Intel Itanium 2	Statically scheduled VLIW-style	6/5/11	9 int. 2 FP	1.6	592 M 423 mm ²	130 W



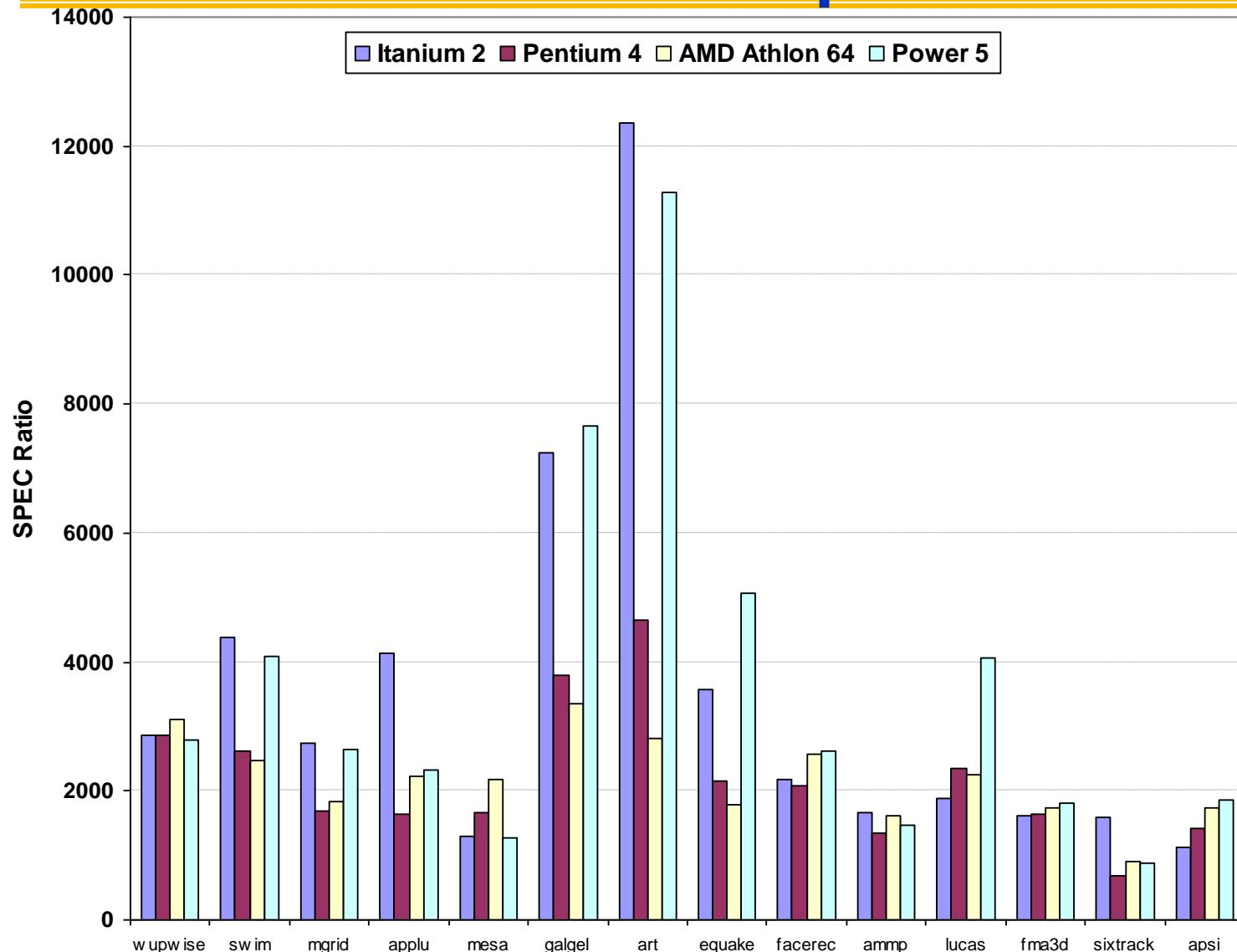
Performance on SPECint2000

■ Itanium 2 ■ Pentium 4 □ AMD Athlon 64 □ Power 5



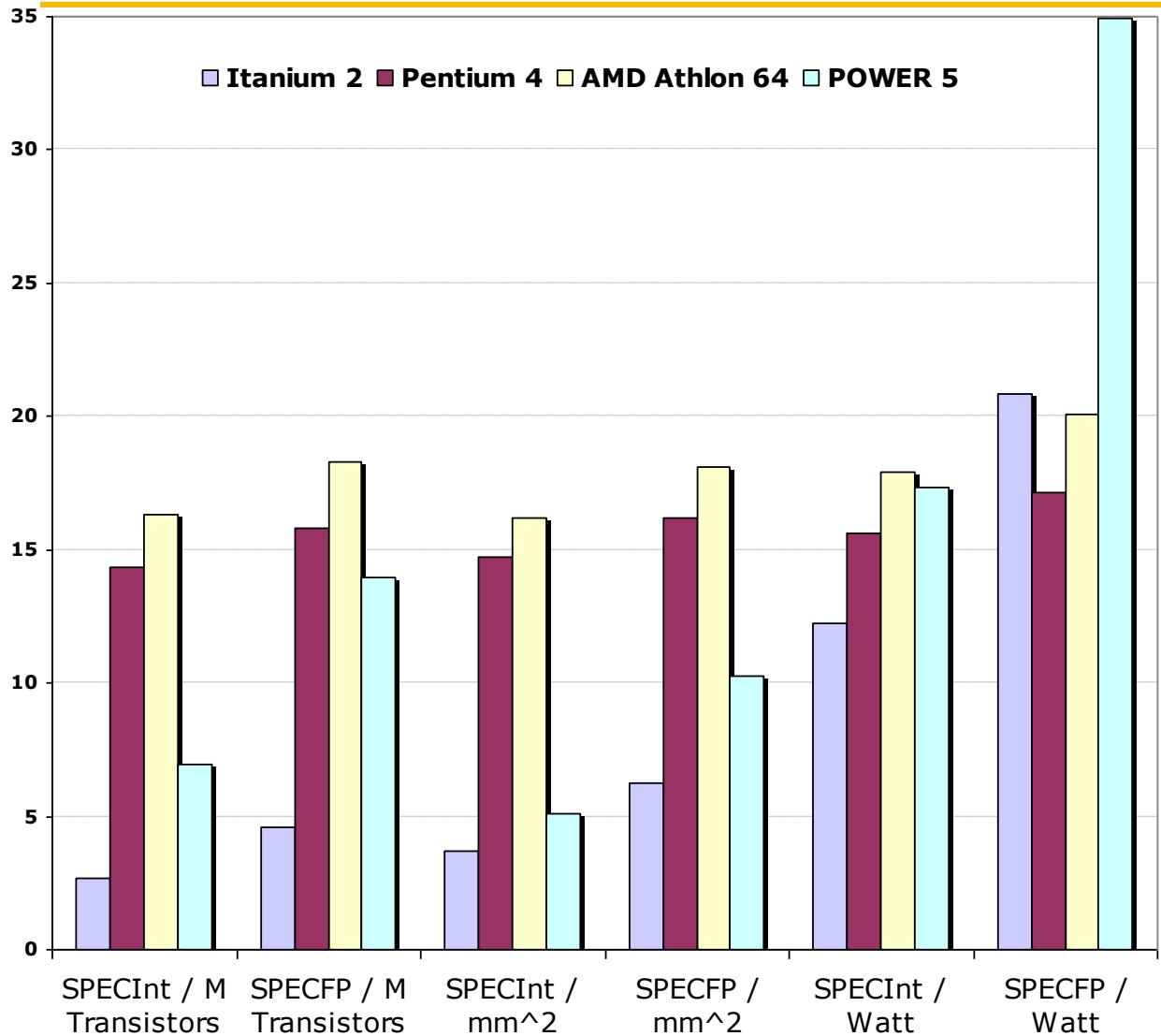


Performance on SPECfp2000





Normalized Performance: Efficiency



Rank	Itanium 2	Pentium 4	AMD Athlon 64	POWER 5
Int/Trans	4	2	1	3
FP/Trans	4	2	1	3
Int/area	4	2	1	3
FP/area	4	2	1	3
Int/Watt	4	3	1	2
FP/Watt	2	4	3	1



No Silver Bullet for ILP

- No obvious over all leader in performance
- The AMD Athlon leads on SPECInt performance followed by the Pentium 4, Itanium 2, and Power5
- Itanium 2 and Power5, which perform similarly on SPECFP, clearly dominate the Athlon and Pentium 4 on SPECFP
- Itanium 2 is the most **inefficient** processor both for Fl. Pt. and integer code for all but one efficiency measure (SPECFP/Watt)
- Athlon and Pentium 4 both make good use of transistors and area in terms of efficiency,
- IBM Power5 is the most effective user of energy on SPECFP and essentially tied on SPECINT



Limits to ILP

- **Doubling issue rates above today's 3-6 instructions per clock, say to 6 to 12 instructions, probably requires a processor to**
 - issue 3 or 4 data memory accesses per cycle,
 - resolve 2 or 3 branches per cycle,
 - rename and access more than 20 registers per cycle, and
 - fetch 12 to 24 instructions per cycle.
- **The complexities of implementing these capabilities is likely to mean sacrifices in the maximum clock rate**
 - E.g., widest issue processor is the Itanium 2, but it also has the slowest clock rate, despite the fact that it consumes the most power!



Limits to ILP

- Most techniques for increasing performance increase power consumption
- The key question is whether a technique is **energy efficient**: does it increase power consumption faster than it increases performance?
- Multiple issue processors techniques all are energy inefficient:
 1. Issuing multiple instructions incurs some overhead in logic that grows faster than the issue rate grows
 2. Growing gap between peak issue rates and sustained performance
- Number of transistors switching = $f(\text{peak issue rate})$, and performance = $f(\text{sustained rate})$, growing gap between peak and sustained performance
⇒ increasing energy per unit of performance



Commentary

- Itanium architecture does **not** represent a significant breakthrough in scaling ILP or in avoiding the problems of complexity and power consumption
- Instead of pursuing more ILP, architects are increasingly focusing on TLP implemented with single-chip multiprocessors
- In 2000, IBM announced the 1st commercial single-chip, general-purpose multiprocessor, the Power4, which contains 2 Power3 processors and an integrated L2 cache
 - Since then, Sun Microsystems, AMD, and Intel have switched to a focus on single-chip multiprocessors rather than more aggressive uniprocessors.
- Right balance of ILP and TLP is unclear today
 - Perhaps right choice for server market, which can exploit more TLP, may differ from desktop, where single-thread performance may continue to be a primary requirement



And in conclusion ...

- **Limits to ILP (power efficiency, compilers, dependencies ...)** seem to limit to 3 to 6 issue for practical options
- **Explicitly parallel (Data level parallelism or Thread level parallelism)** is next step to performance
- **Coarse grain vs. Fine grained multithreading**
 - Only on big stall vs. every clock cycle
- **Simultaneous Multithreading if fine grained multithreading based on OOO superscalar microarchitecture**
 - Instead of replicating registers, reuse rename registers
- **Itanium/EPIC/VLIW is not a breakthrough in ILP**
- **Balance of ILP and TLP decided in marketplace**



CS 152 Computer Architecture and Engineering

Lecture 16 - VLIW Machines and Statically Scheduled ILP

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

<http://www.eecs.berkeley.edu/~krste>
<http://inst.eecs.berkeley.edu/~cs152>

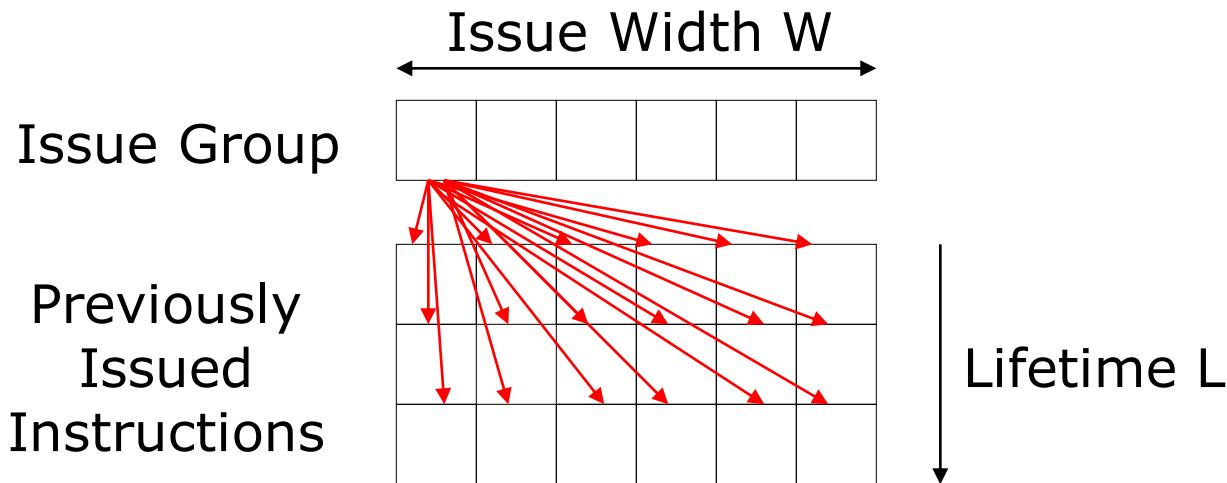


Last time in Lecture 15

- Unified physical register file machines remove data values from ROB
 - All values only read and written during execution
 - Only register tags held in ROB
 - Allocate resources (ROB slot, destination physical register, memory reorder queue location) during decode
 - Issue window can be separated from ROB and made smaller than ROB (allocate in decode, free after instruction completes)
 - Free resources on commit
- Speculative store buffer holds store values before commit to allow load-store forwarding
- Can execute later loads past earlier stores when addresses known, or predicted no dependence



Superscalar Control Logic Scaling



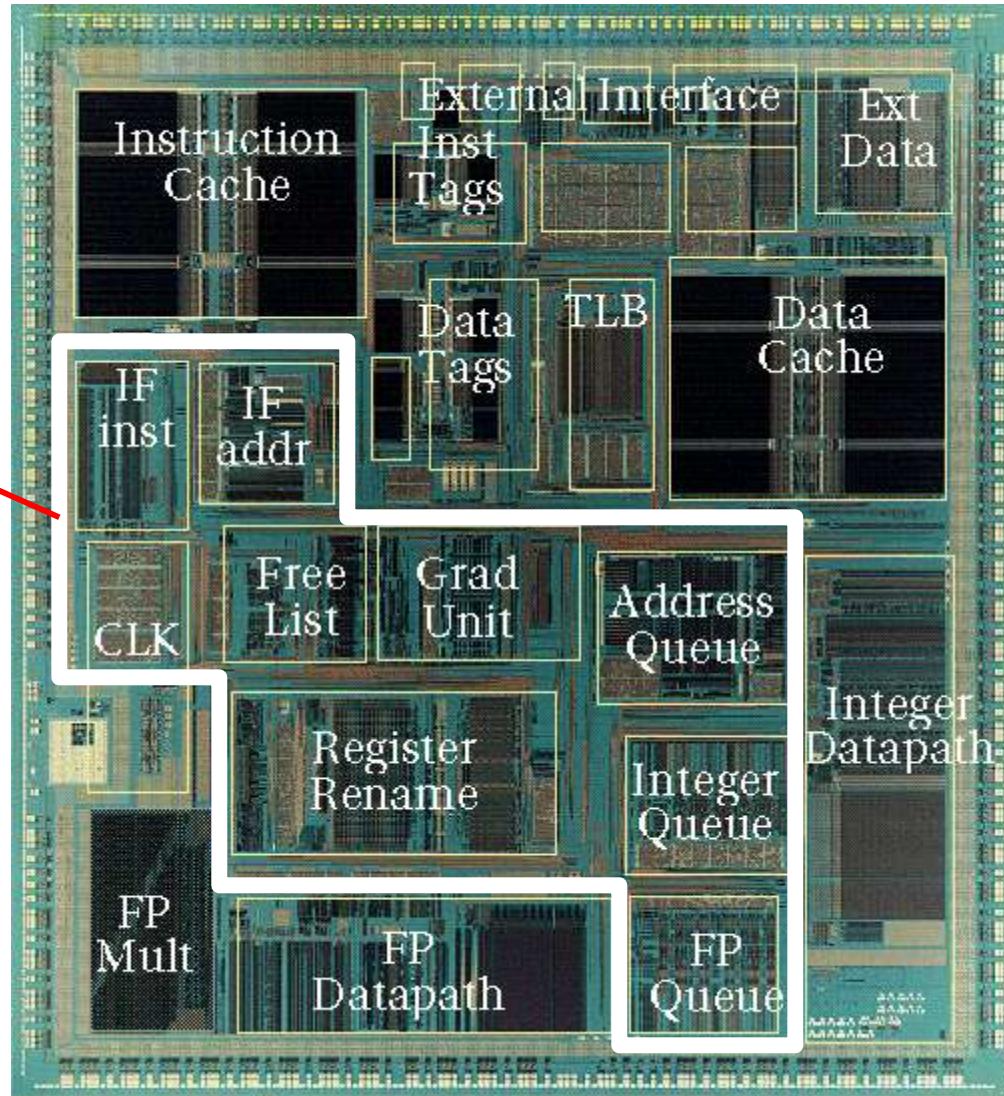
- Each issued instruction must somehow check against $W*L$ instructions, i.e., growth in hardware $\propto W*(W*L)$
- For in-order machines, L is related to pipeline latencies and check is done during issue (interlocks or scoreboard)
- For out-of-order machines, L also includes time spent in instruction buffers (instruction window or ROB), and check is done by broadcasting tags to waiting instructions at write back (completion)
- As W increases, larger instruction window is needed to find enough parallelism to keep machine busy => greater L

=> Out-of-order control logic grows faster than W^2 ($\sim W^3$)



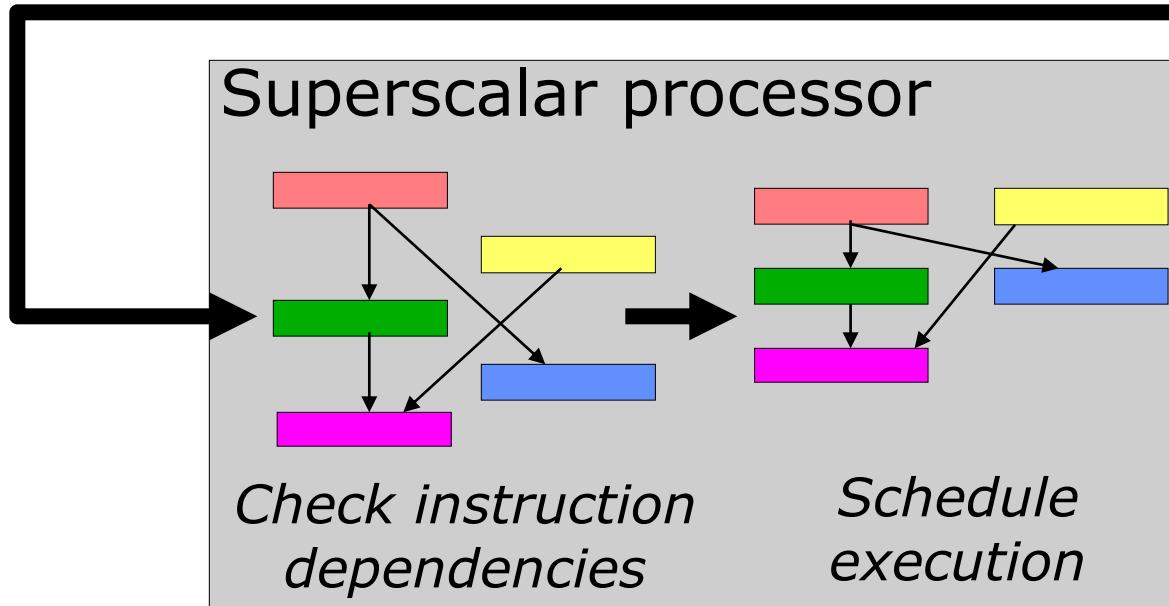
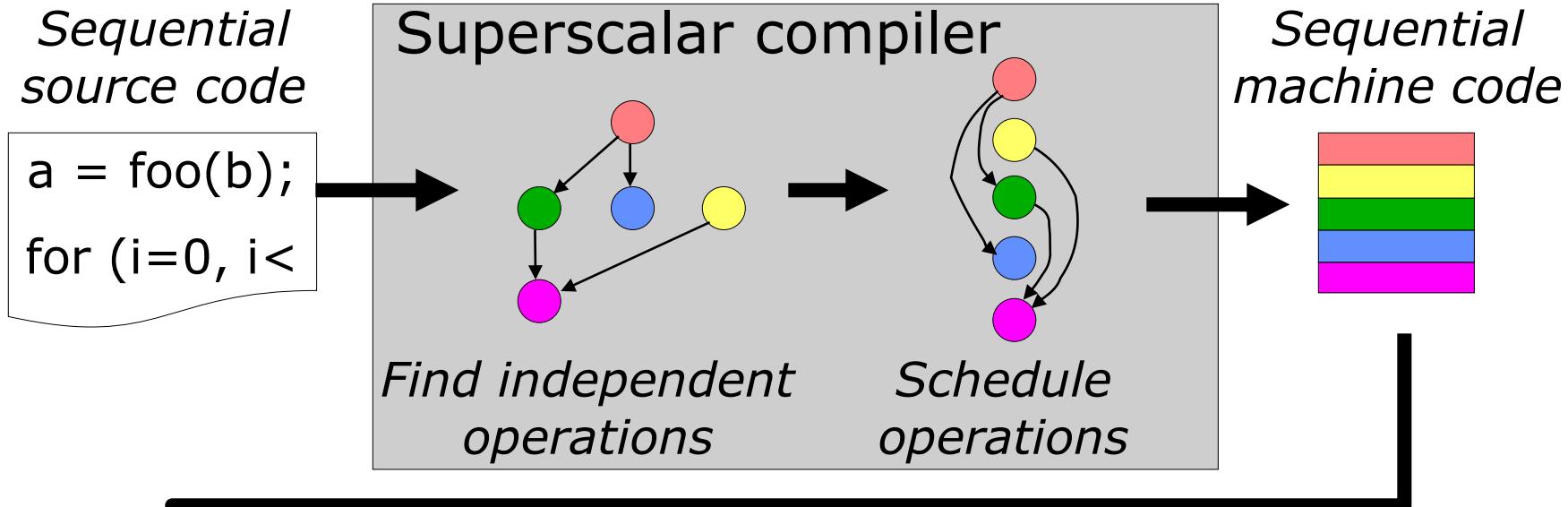
Out-of-Order Control Complexity: MIPS R10000

*Control
Logic*



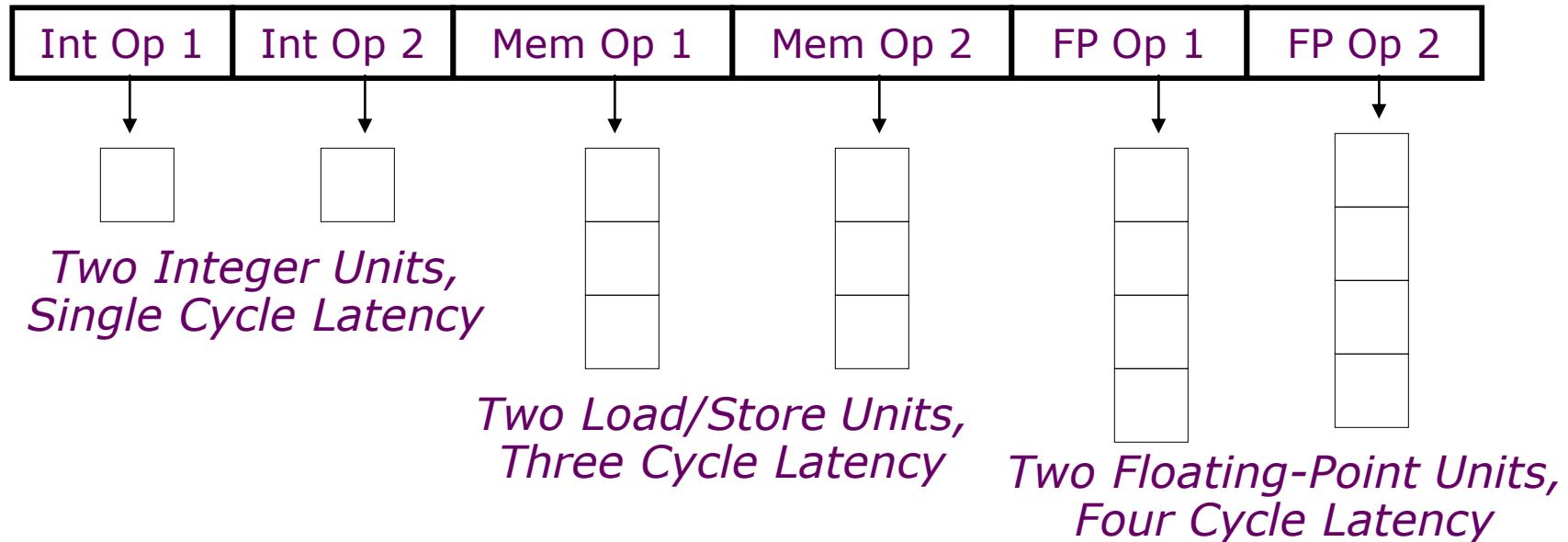
[SGI/MIPS
Technologies
Inc., 1995]

Sequential ISA Bottleneck





VLIW: Very Long Instruction Word



- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
 - Parallelism within an instruction => no cross-operation RAW check
 - No data use before data ready => no data interlocks



VLIW Compiler Responsibilities

- Schedules to maximize parallel execution
- Guarantees intra-instruction parallelism
- Schedules to avoid data hazards (no interlocks)
 - Typically separates operations with explicit NOPs



Early VLIW Machines

- **FPS AP120B (1976)**
 - scientific attached array processor
 - first commercial wide instruction machine
 - hand-coded vector math libraries using software pipelining and loop unrolling
- **Multiflow Trace (1987)**
 - commercialization of ideas from Fisher's Yale group including "trace scheduling"
 - available in configurations with 7, 14, or 28 operations/instruction
 - 28 operations packed into a 1024-bit instruction word
- **Cydrome Cydra-5 (1987)**
 - 7 operations encoded in 256-bit instruction word
 - rotating register file



Loop Execution

```
for (i=0; i<N; i++)  
    B[i] = A[i] + C;
```

Compile

loop: $\text{Id } f_1, 0(r_1)$

add r1, 8

fadd f2, f0, f1

sd f2, 0(r2)

add r2, 8

bne r1, r3, loop

loop:

Schedule

How many FP ops/cycle?

$$1 \text{ fadd} / 8 \text{ cycles} = 0.125$$



Loop Unrolling

```
for (i=0; i<N; i++)  
    B[i] = A[i] + C;
```

Unroll inner loop to perform 4 iterations at once

```
for (i=0; i<N; i+=4)  
{  
    B[i]      = A[i] + C;  
    B[i+1] = A[i+1] + C;  
    B[i+2] = A[i+2] + C;  
    B[i+3] = A[i+3] + C;  
}
```

Need to handle values of N that are not multiples of unrolling factor with final cleanup loop



Scheduling Loop Unrolled Code

Unroll 4 ways

```
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
```

loop:

Schedule →

	Int1	Int 2	M1	M2	FP+	FPx
			ld f1			
			ld f2			
			ld f3			
add r1			ld f4		fadd f5	
					fadd f6	
					fadd f7	
					fadd f8	
			sd f5			
			sd f6			
			sd f7			
	add r2	bne	sd f8			

How many FLOPS/cycle?

$$4 \text{ fadds} / 11 \text{ cycles} = 0.36$$



Software Pipelining

Unroll 4 ways first

```
loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
```

Int1	Int 2	M1	M2	FP+	FPx
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4			
		ld f1		fadd f5	
		ld f2		fadd f6	
		ld f3		fadd f7	
add r1		ld f4		fadd f8	
		ld f1	sd f5	fadd f5	
		ld f2	sd f6	fadd f6	
	add r2	ld f3	sd f7	fadd f7	
add r1 bne		ld f4	sd f8	fadd f8	
			sd f5	fadd f5	
			sd f6	fadd f6	
	add r2		sd f7	fadd f7	
	bne		sd f8	fadd f8	
			sd f5		

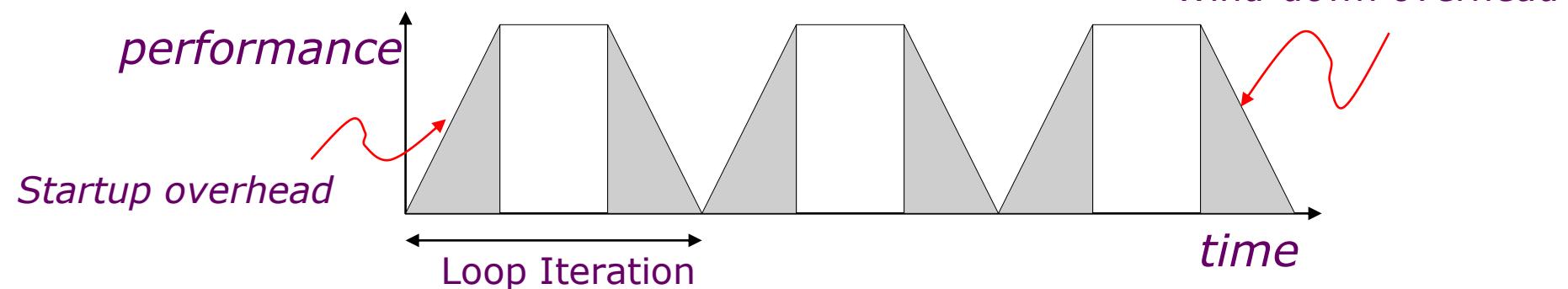
How many FLOPS/cycle?

4 fadds / 4 cycles = 1

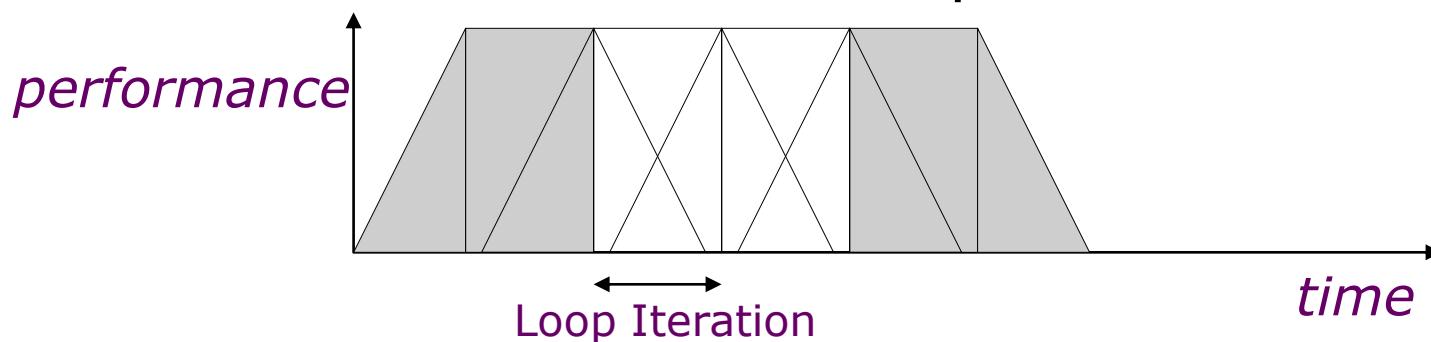


Software Pipelining vs. Loop Unrolling

Loop Unrolled



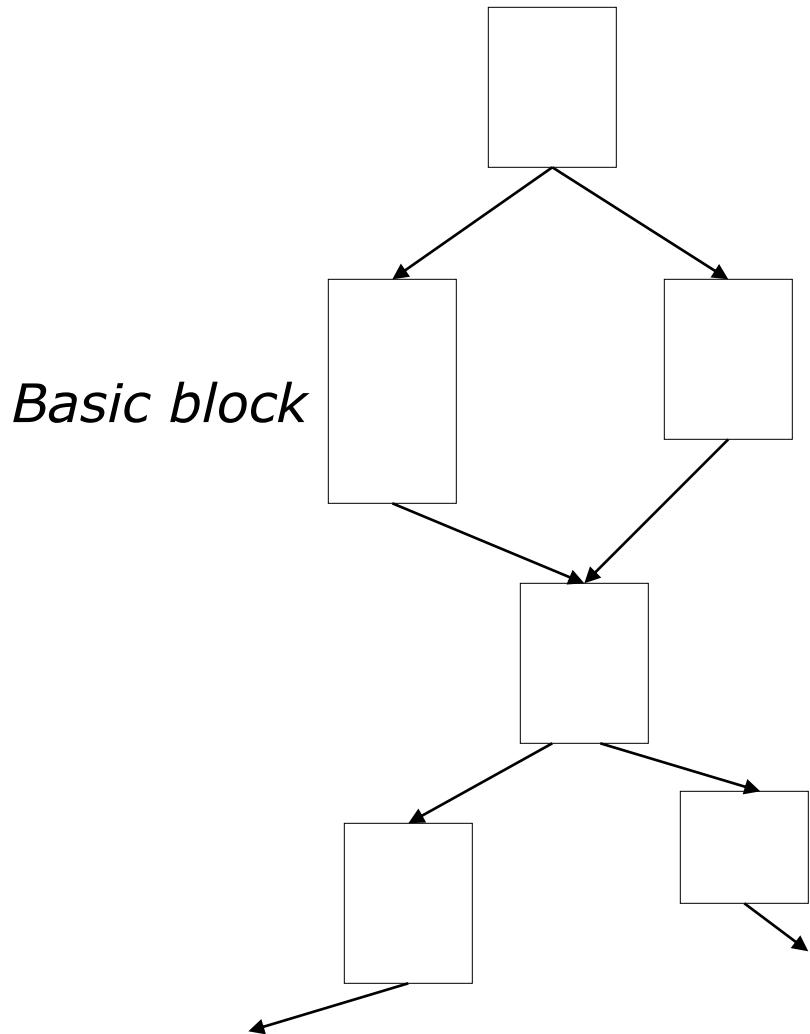
Software Pipelined



Software pipelining pays startup/wind-down costs only once per loop, not once per iteration



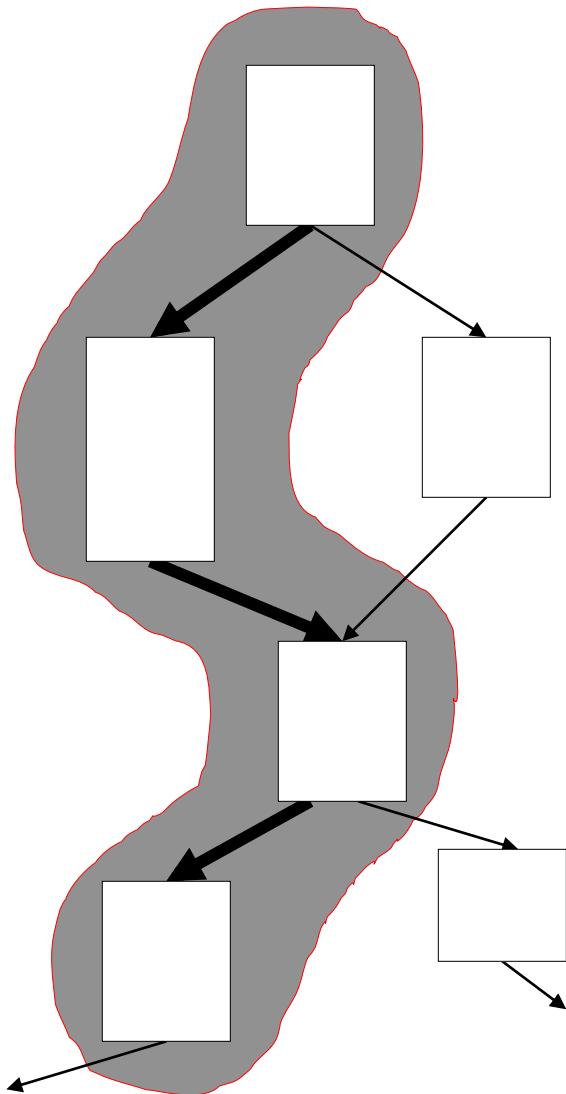
What if there are no loops?



- Branches limit basic block size in control-flow intensive irregular code
- Difficult to find ILP in individual basic blocks



Trace Scheduling [*Fisher,Ellis*]



- Pick string of basic blocks, a *trace*, that represents most frequent branch path
- Use profiling feedback or compiler heuristics to find common branch paths
- Schedule whole “trace” at once
- Add fixup code to cope with branches jumping out of trace

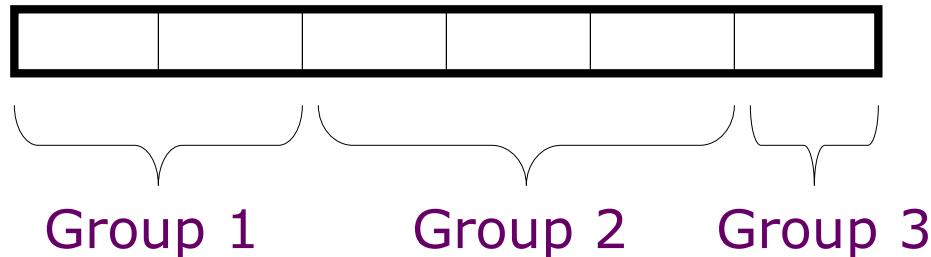


Problems with “Classic” VLIW

- Object-code compatibility
 - have to recompile all code for every machine, even for two machines in same generation
- Object code size
 - instruction padding wastes instruction memory/cache
 - loop unrolling/software pipelining replicates code
- Scheduling variable latency memory operations
 - caches and/or memory bank conflicts impose statically unpredictable variability
- Knowing branch probabilities
 - Profiling requires an significant extra step in build process
- Scheduling for statically unpredictable branches
 - optimal schedule varies with branch path



VLIW Instruction Encoding

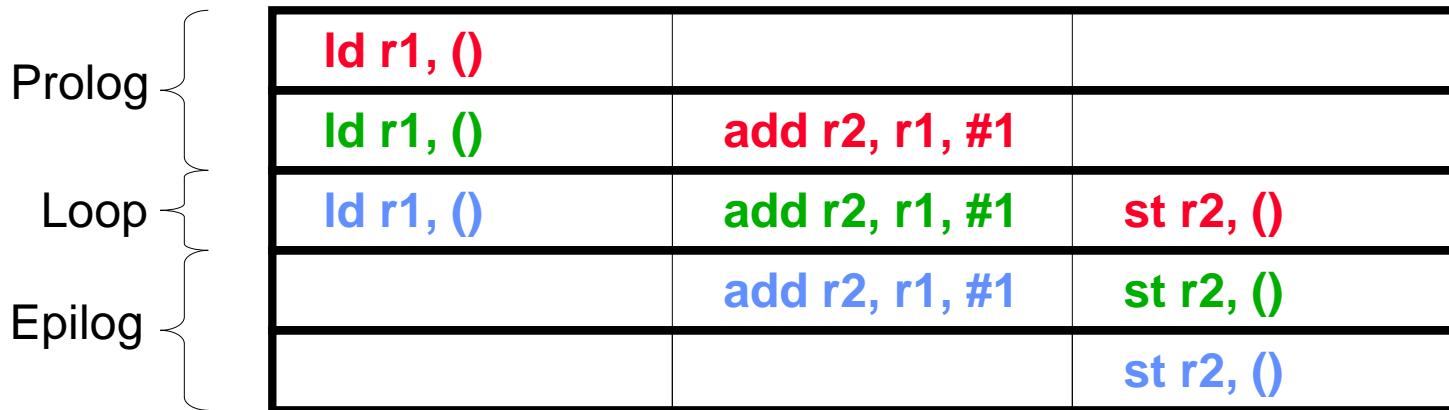
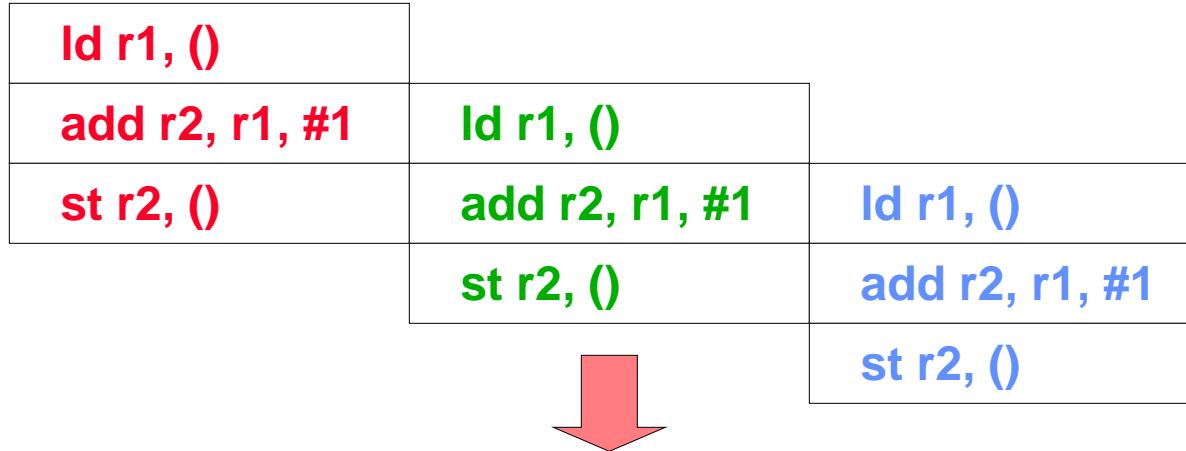


- Schemes to reduce effect of unused fields
 - Compressed format in memory, expand on I-cache refill
 - » used in Multiflow Trace
 - » introduces instruction addressing challenge
 - Mark parallel groups
 - » used in TMS320C6x DSPs, Intel IA-64
 - Provide a single-op VLIW instruction
 - » Cydra-5 UniOp instructions



Rotating Register Files

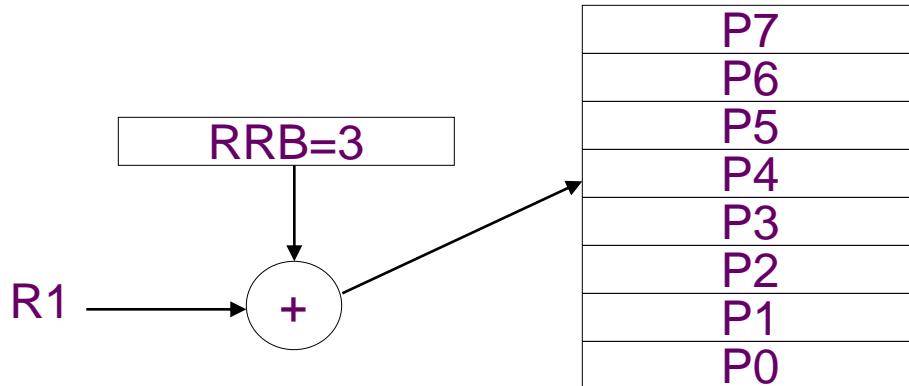
Problems: Scheduled loops require lots of registers,
Lots of duplicated code in prolog, epilog



Solution: Allocate new set of registers for each loop iteration



Rotating Register File



Rotating Register Base (RRB) register points to base of current register set. Value added on to logical register specifier to give physical register number. Usually, split into rotating and non-rotating registers.

Prolog	ld r1, ()		dec RRB
	ld r1, ()	add r3, r2, #1	dec RRB
Loop	ld r1, ()	add r3, r2, #1	st r4, () bloop
		add r2, r1, #1	st r4, () dec RRB
Epilog			st r4, () dec RRB

Loop closing branch decrements RRB



Rotating Register File

(Previous Loop Example)

Three cycle load latency
encoded as difference of 3
in register specifier
number ($f_4 - f_1 = 3$)

Four cycle fadd latency
encoded as difference of 4
in register specifier
number ($f_9 - f_5 = 4$)



ld P9, ()	fadd P13, P12,	sd P17, ()	bloop	RRB=8
ld P8, ()	fadd P12, P11,	sd P16, ()	bloop	RRB=7
ld P7, ()	fadd P11, P10,	sd P15, ()	bloop	RRB=6
ld P6, ()	fadd P10, P9,	sd P14, ()	bloop	RRB=5
ld P5, ()	fadd P9, P8,	sd P13, ()	bloop	RRB=4
ld P4, ()	fadd P8, P7,	sd P12, ()	bloop	RRB=3
ld P3, ()	fadd P7, P6,	sd P11, ()	bloop	RRB=2
ld P2, ()	fadd P6, P5,	sd P10, ()	bloop	RRB=1



Cydra-5: Memory Latency Register (MLR)

Problem: Loads have variable latency

Solution: Let software choose desired memory latency

- Compiler schedules code for maximum load-use distance
- Software sets MLR to latency that matches code schedule
- Hardware ensures that loads take exactly MLR cycles to return values into processor pipeline
 - Hardware buffers loads that return early
 - Hardware stalls processor if loads return late



CS152 Administrivia

- Quiz 4, Tuesday April 7



Intel EPIC IA-64

- EPIC is the style of architecture (cf. CISC, RISC)
 - Explicitly Parallel Instruction Computing
- IA-64 is Intel's chosen ISA (cf. x86, MIPS)
 - IA-64 = Intel Architecture 64-bit
 - An object-code compatible VLIW
- Itanium (aka Merced) is first implementation (cf. 8086)
 - First customer shipment expected 1997 (actually 2001)
 - McKinley, second implementation shipped in 2002
 - Recent version, Tukwila 2008, quad-cores, 65nm



Quad Core Itanium “Tukwila” [Intel 2008]

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

- 4 cores
- 6MB \$/core, 24MB \$ total
- ~2.0 GHz
- 698mm² in 65nm CMOS!!!!
- 170W
- Over 2 billion transistors

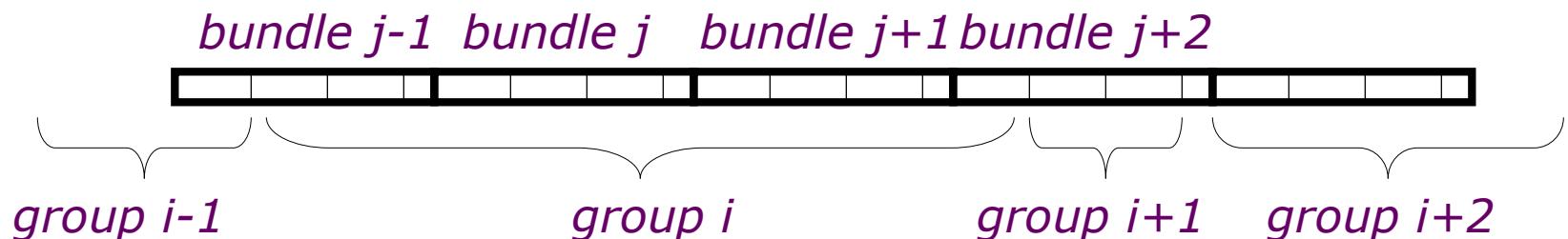


IA-64 Instruction Format



128-bit instruction bundle

- Template bits describe grouping of these instructions with others in adjacent bundles
- Each group contains instructions that can execute in parallel





IA-64 Registers

- 128 General Purpose 64-bit Integer Registers
 - 128 General Purpose 64/80-bit Floating Point Registers
 - 64 1-bit Predicate Registers
-
- GPRs rotate to reduce code size for software pipelined loops

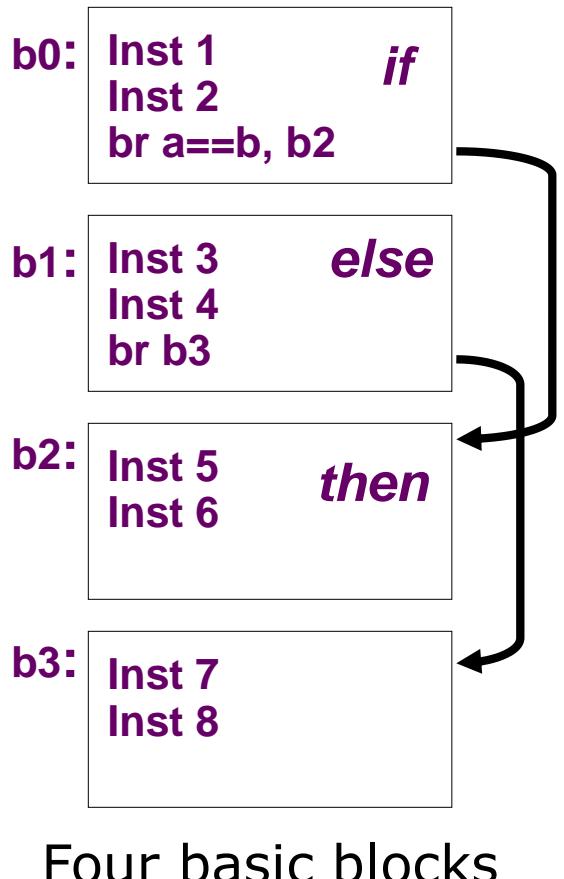


IA-64 Predicated Execution

Problem: Mispredicted branches limit ILP

Solution: Eliminate hard to predict branches with predicated execution

- Almost all IA-64 instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false



Predication

Inst 1
Inst 2
p1,p2 <- cmp(a==b)
(p1) Inst 3 || (p2) Inst 5
(p1) Inst 4 || (p2) Inst 6
Inst 7
Inst 8

One basic block

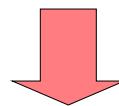
*Mahlke et al, ISCA95: On average
>50% branches removed*



Predicate Software Pipeline Stages

Single VLIW Instruction

(p1) ld r1	(p2) add r3	(p3) st r4	(p1) bloop
------------	-------------	------------	------------



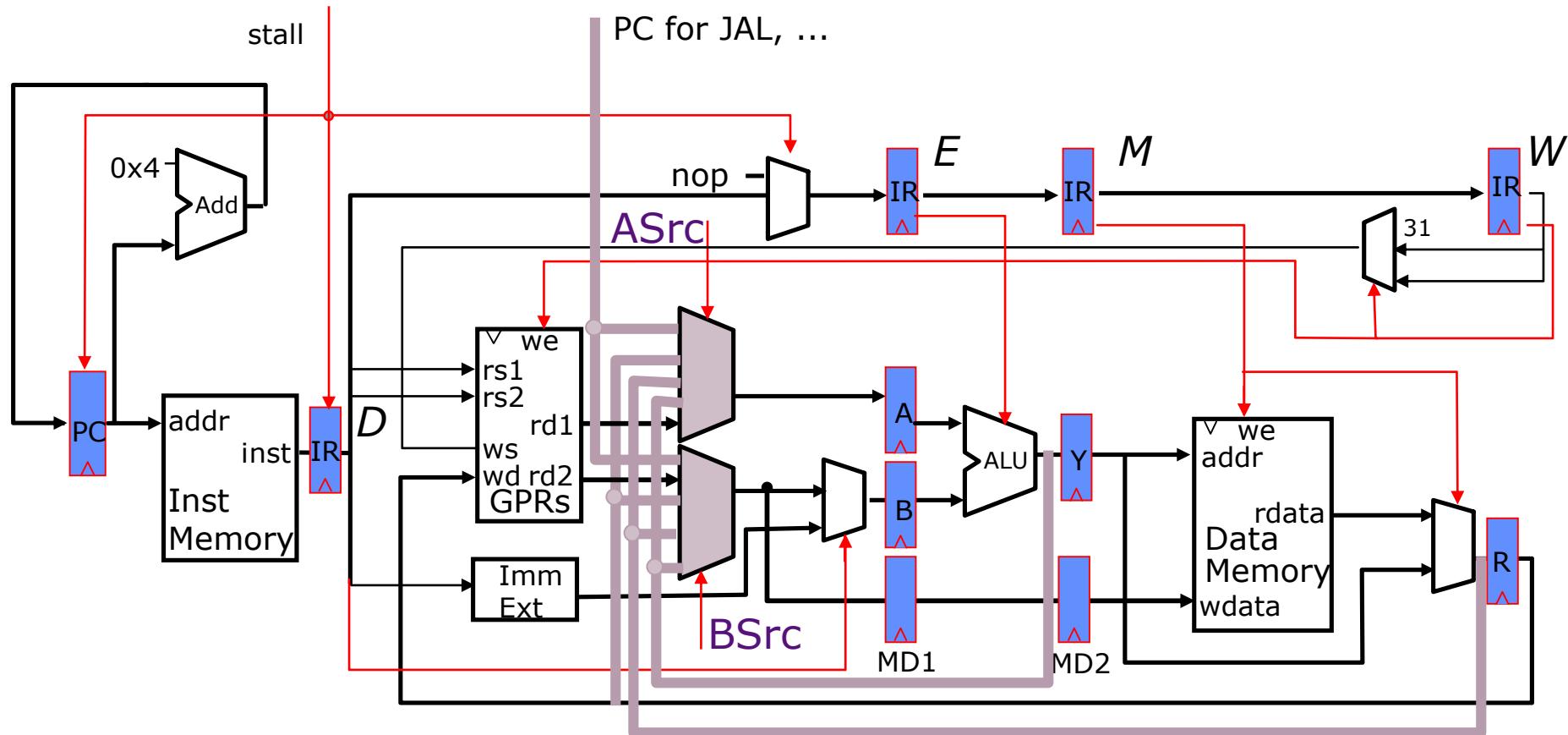
Dynamic Execution

(p1) ld r1			(p1) bloop
(p1) ld r1	(p2) add r3		(p1) bloop
(p1) ld r1	(p2) add r3	(p3) st r4	(p1) bloop
(p1) ld r1	(p2) add r3	(p3) st r4	(p1) bloop
(p1) ld r1	(p2) add r3	(p3) st r4	(p1) bloop
	(p2) add r3	(p3) st r4	(p1) bloop
		(p3) st r4	(p1) bloop

Software pipeline stages turned on by rotating predicate registers → Much denser encoding of loops



Fully Bypassed Datapath



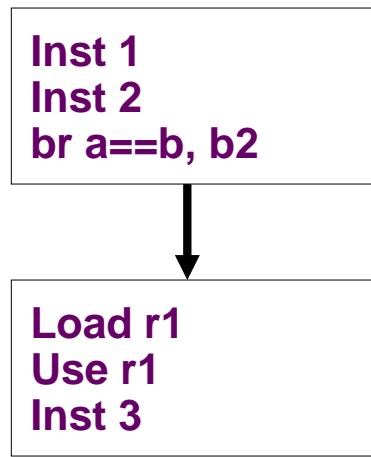
Where does predication fit in?



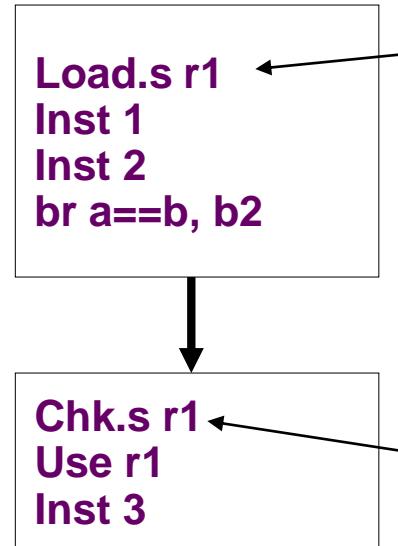
IA-64 Speculative Execution

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don't cause exceptions



***Can't move load above branch
because might cause spurious
exception***



***Speculative load
never causes
exception, but sets
“poison” bit on
destination register***

***Check for exception in
original home block
jumps to fixup code if
exception detected***

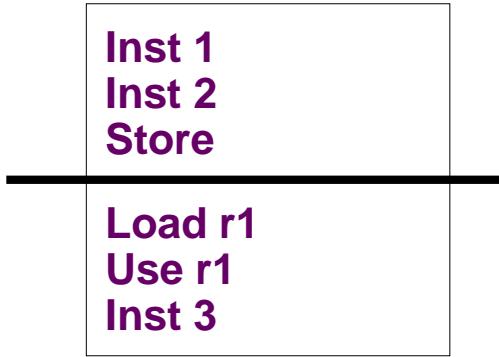
Particularly useful for scheduling long latency loads early



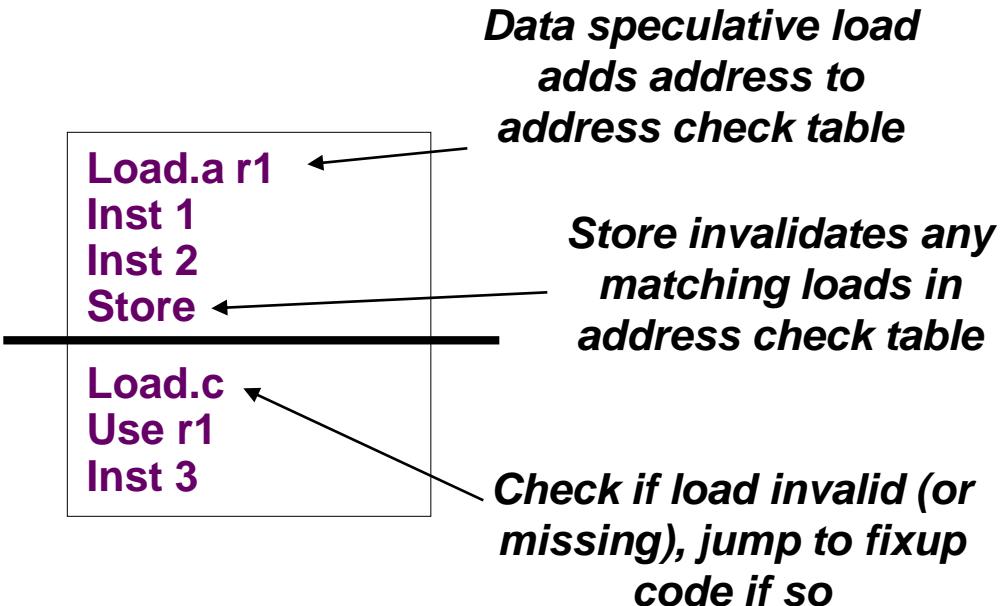
IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling

Solution: Hardware to check pointer hazards



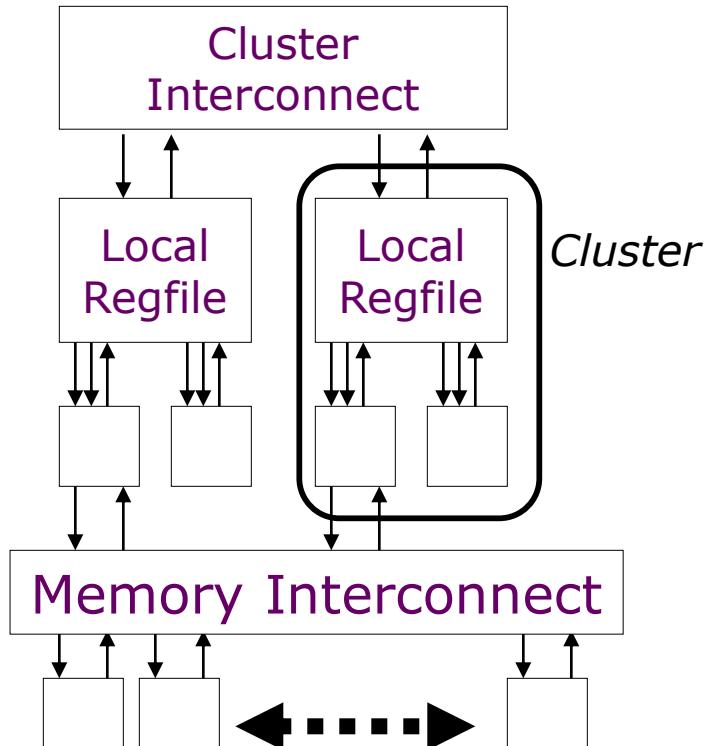
***Can't move load above store
because store might be to same
address***



Requires associative hardware in address check table



Clustered VLIW



Cache/Memory Banks

- Divide machine into clusters of local register files and local functional units
- Lower bandwidth/higher latency interconnect between clusters
- Software responsible for mapping computations to minimize communication overhead
- Common in commercial embedded VLIW processors, e.g., TI C6x DSPs, HP Lx processor
- (Same idea used in some superscalar processors, e.g., Alpha 21264)



Limits of Static Scheduling

- Unpredictable branches
- Variable memory latency (unpredictable cache misses)
- Code size explosion
- Compiler complexity



Acknowledgements

- These slides contain material developed and copyright by:
 - Arvind (MIT)
 - Krste Asanovic (MIT/UCB)
 - Joel Emer (Intel/MIT)
 - James Hoe (CMU)
 - John Kubiatowicz (UCB)
 - David Patterson (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252

COMP4211 – Advanced Computer Architectures & Algorithms

University of NSW

Seminar Presentation

Semester 1 2004

Software Approaches to Exploiting Instruction Level
Parallelism

Lecture notes by: ~~David A. Patterson~~

Boris Savkovic

Outline

- | | | |
|--|---|--------|
| 1. Introduction | → | TALK ☺ |
| 2. Basic Pipeline Scheduling | → | TALK ☺ |
| 3. Instruction Level Parallelism and Dependencies | → | TALK ☺ |
| 4. Local Optimizations and Loops | → | TALK ☺ |
| 5. Global Scheduling Approaches | → | TALK ☺ |
| 6. HW Support for Aggressive Optimization Strategies | → | TALK ☺ |

What is scheduling?

- **Scheduling** is the ordering of program execution so as to improve performance without affecting program correctness.
- Our focus to date has been on hardware-based scheduling, which involved execution scheduling or rearrangement of issued instructions to reduce execution time.
- Today we'll look at compiler-based scheduling, which is also known as *static scheduling* if the hardware does not subsequently reorder the instruction sequence produced by the compiler.

How does software-based scheduling differ from hardware-based scheduling?

Unlike with hardware-based approaches, the overhead due to intensive analysis of the instruction sequence is generally not an issue:

- We can afford to perform more detailed analysis of the instruction sequence.
- We can generate more information about the instruction sequence and thus involve more factors in optimizing the instruction sequence.

BUT:

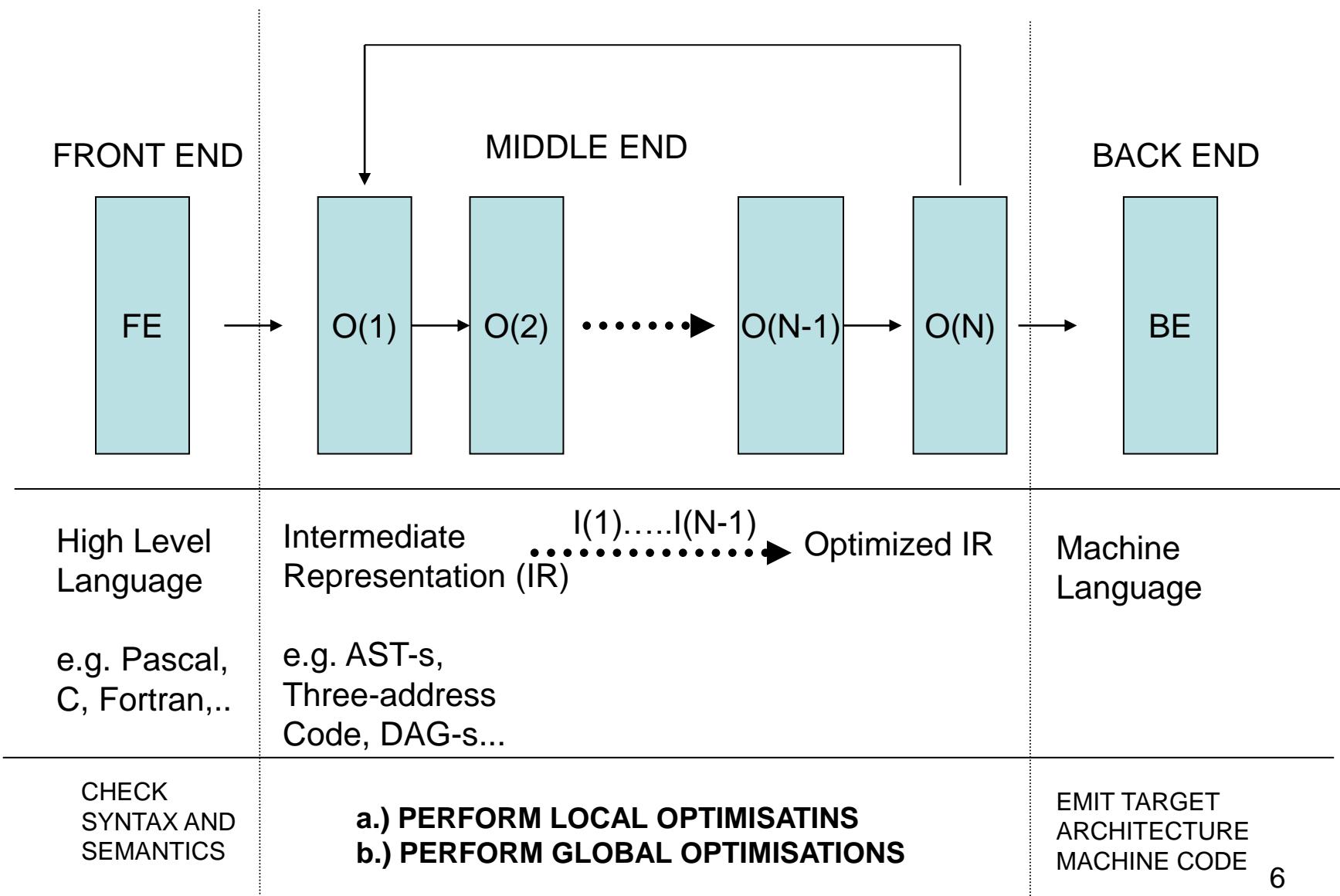
- There will be a significant number of cases where not enough information can be extracted from the instruction sequence statically to perform an optimization:
 - e.g. : → do two pointers point to the same memory location?
 - what is the upper bound on the induction variable of a loop?

How does software-based scheduling differ from hardware-based scheduling?

STILL:

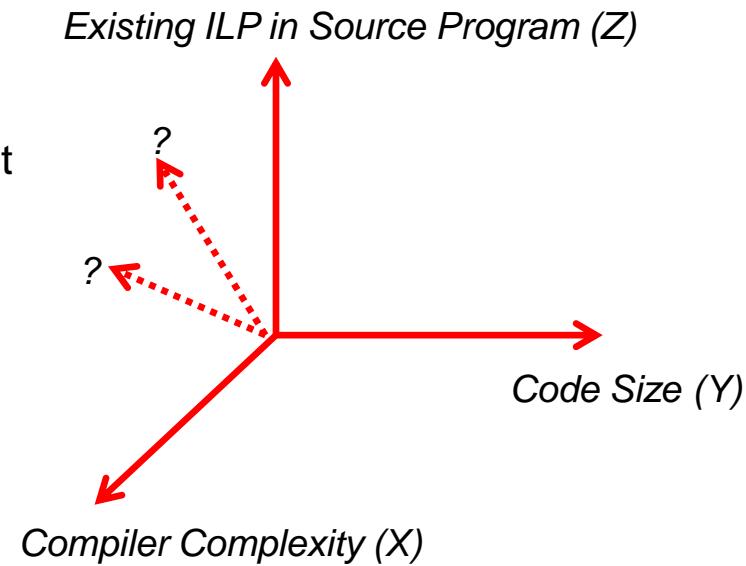
- We can assist the hardware during compile time by exposing more ILP in the instruction sequence and/or performing some classic optimizations.
- We can exploit characteristics of the underlying architecture to increase performance (e.g. schedule a branch delay slot).
- The above tasks are usually performed by an optimizing compiler via a series of analysis and transformation steps (see next slide).

Architecture of a typical optimizing compiler

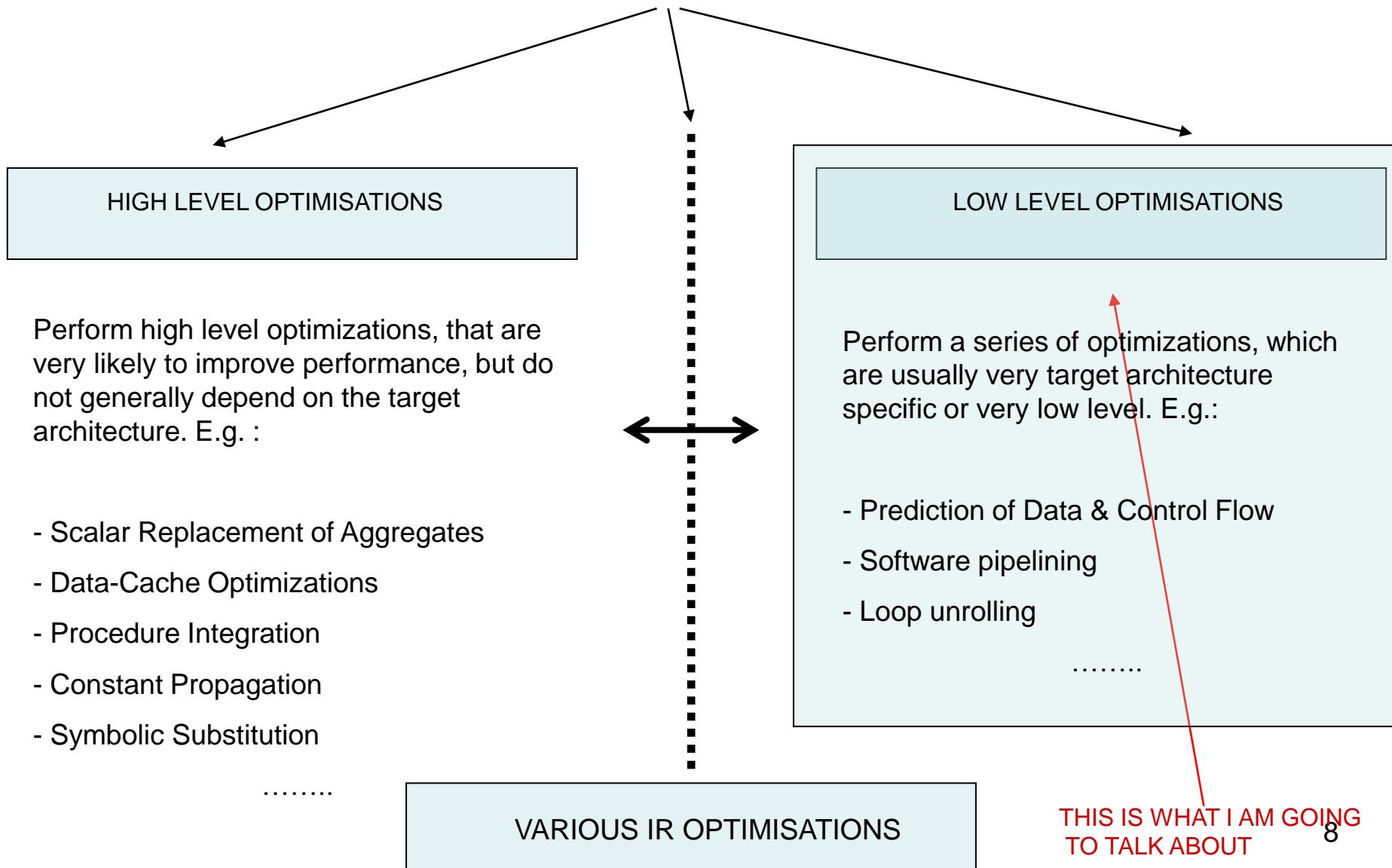


Compile-Time Optimizations are subject to many predictable and unpredictable factors:

- Like with hardware approaches, it might be very difficult to judge the benefit gained from a transformation applied to a given code segment.
- This is because changes at compile-time can have many side-effects, which are not easy to quantify and/or measure for different program behaviours and/or inputs.
- Different compilers emit code for different architectures, so identical transformations might produce better or worse performance, depending on how the hardware schedules instructions.



What are some typical optimizations?

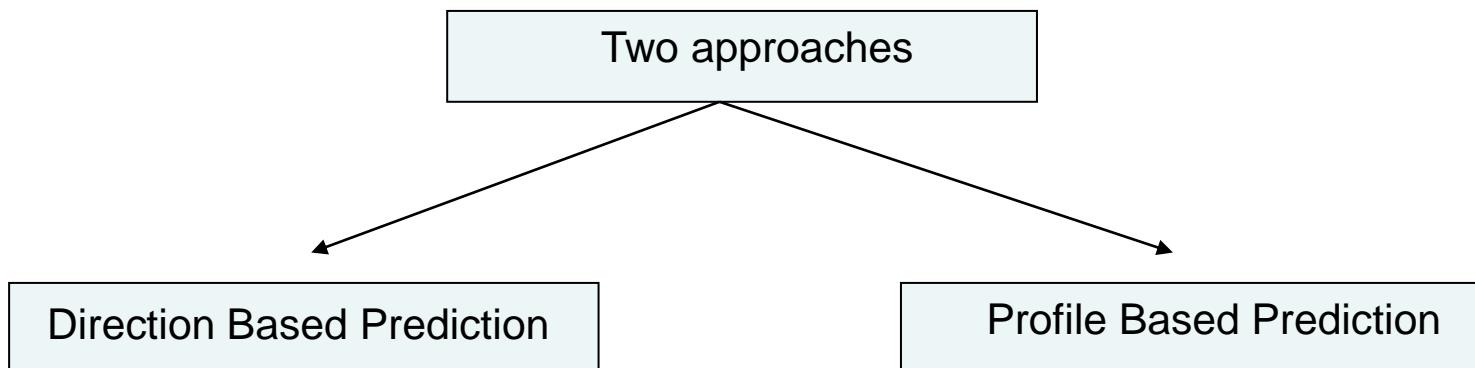


Outline

- | | | |
|--|---|--------|
| 1. Introduction | → | DONE ☺ |
| 2. Basic Pipeline Scheduling | → | TALK ☹ |
| 3. Instruction Level Parallelism and Dependencies | → | TALK ☹ |
| 4. Local Optimizations and Loops | → | TALK ☹ |
| 5. Global Scheduling Approaches | → | TALK ☹ |
| 6. HW Support for Aggressive Optimization Strategies | → | TALK ☹ |

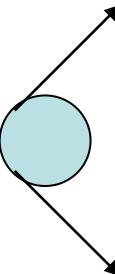
STATIC BRANCH PREDICTION

- Basic pipeline scheduling techniques involve static prediction of branches, (usually) without extensive analysis at compile time.
- Static prediction methods are based on expected/observed behaviour at branch points.
- Usually based on heuristic assumptions, that are easily violated, which we will address in the subsequent slides
- KEY IDEA: Hope that our assumption is correct. If yes, then we've gained a performance improvement. Otherwise, program is still correct, all we've done is “waste” a clock cycle. Overall, we hope to gain.



1.) Direction based Predictions (predict taken/not taken)

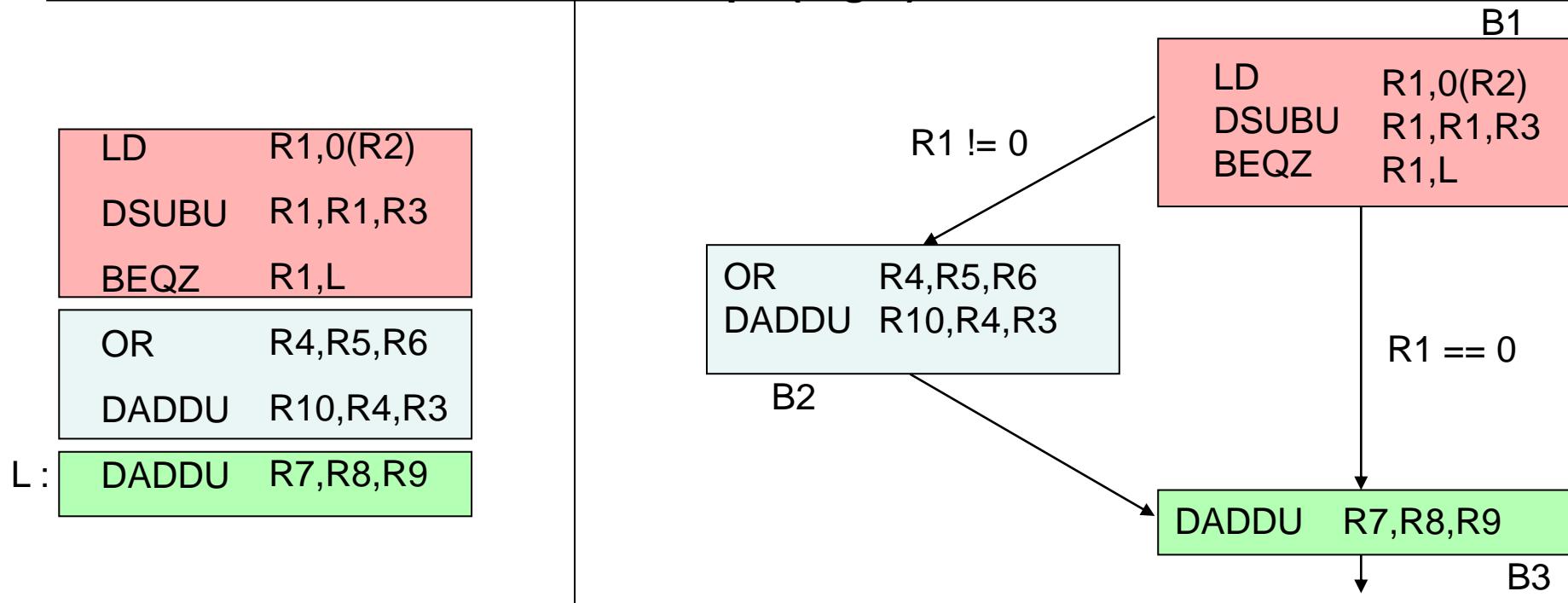
- Assume branch behavior is highly predictable at compile time,
- Perform scheduling by predicting branch statically as either taken or not taken,
- Alternatively, choose forward going branches as “not taken” and backward going branches as “taken”, i.e. exploit loop behaviour,



This is unlikely to produce a misprediction rate of less than 30% to 40% on average, with a variation from 10% to 59%
(CA:AQA)

Branch behaviour is variable. It can be dynamic or static, depending on code. Can't capture such behaviour at compile time with simple direction based prediction!

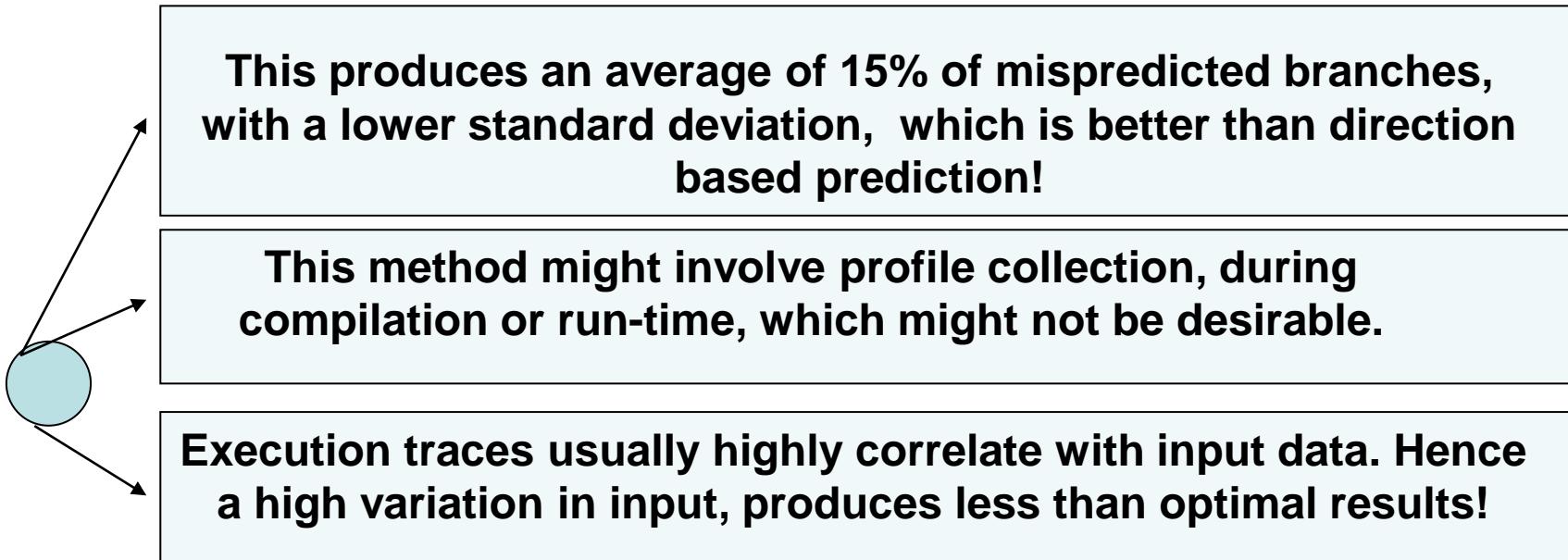
Example: Filling a branch delay slot, a Code Sequence (Left) and its Flow-Graph (Right)



- 1.) DSUBU and BEQZ are output dependent on LD,
- 2.) If we knew that the branch was taken with a high probability, then DADDU could be moved into block B1, since it doesn't have any dependencies with block B2,
- 3.) Conversely, knowing the branch was not taken, then OR could be moved into block B1, since it doesn't affect anything in B3,

2.) Profile Based Predictions

- Collect profile information at run-time
- Since branches tend to be “bimodal”, i.e., highly biased, a more accurate prediction can be made based on collected information.



Outline

- | | | |
|--|---|--------|
| 1. Introduction | → | DONE ☺ |
| 2. Basic Pipeline Scheduling | → | DONE ☺ |
| 3. Instruction Level Parallelism and Dependencies | → | TALK ☹ |
| 4. Local Optimizations and Loops | → | TALK ☹ |
| 5. Global Scheduling Approaches | → | TALK ☹ |
| 6. HW Support for Aggressive Optimization Strategies | → | TALK ☹ |

What is instruction Level Parallelism (ILP)?

- Inherent property of a sequence of instructions, as a result of which some instructions can be allowed to execute in parallel. **(This shall be our definition)**
- Note that this definition implies parallelism across a sequence of instructions (block). This could be a loop, a conditional, or some other valid sequence of statements.
- There is an upper bound, as to how much parallelism can be achieved, since by definition parallelism is an inherent property of the sequence of instructions.
- We can approach this upper bound via a series of transformations that either expose or allow more ILP to be exposed to later transformations.

What is instruction Level Parallelism (ILP)?

- Dependencies within a sequence of instructions determine how much ILP is present.
Think of this as:

To what degree can we rearrange the instructions without compromising correctness?

Hence →

OUR AIM: Improve performance by exploiting ILP !

How do we exploit ILP?

- Have a collection of transformations, that operate on or across program blocks, either producing “faster code” or exposing more ILP. Recall from before :

An optimizing compiler does this by iteratively applying a series of transformations!

- Our transformations should rearrange code, from data available statically at compile time and from our knowledge of the underlying hardware.

How do we exploit ILP?

- **KEY IDEA:** These transformations do one (or both) of the following, while preserving correctness :
 - 1.) Expose more ILP, such that later transformations in the compiler can exploit this exposure of more ILP.
 - 2.) Perform a rearrangement of instructions, which results in increased performance (measured by execution time, or some other metric of interest)

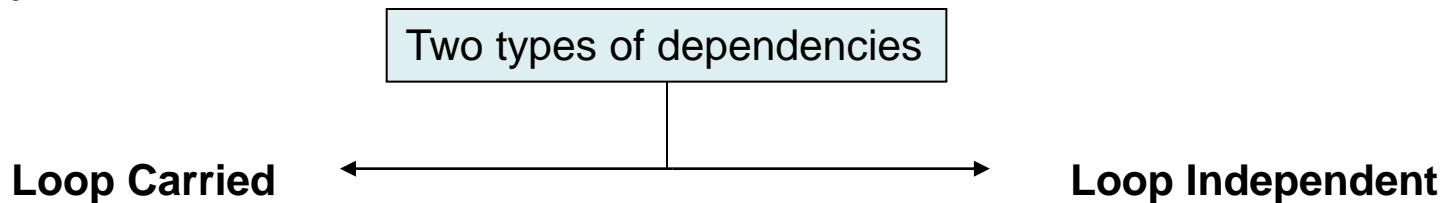
Loop Level Parallelism and Dependence

- We will look at two techniques (software pipelining and static loop unrolling) that can detect and expose more loop level parallelism.

Q: What is Loop Level Parallelism?

A: ILP that exists as a result of iterating a loop.

- Two types of dependencies limit the degree to which Loop Level Parallelism can be exploited.



A dependence, which only applies if a loop is iterated.

A dependence within the body of the loop itself (i.e. within one iteration).

An Example of Loop Level Dependences

- Consider the following loop:

```
for (i = 0; i <= 100; i++) {
```

$$A[i + 1] = A[i] + C[i]; \quad // S1$$
$$B[i + 1] = B[i] + A[i + 1]; \quad // S2$$

```
}
```

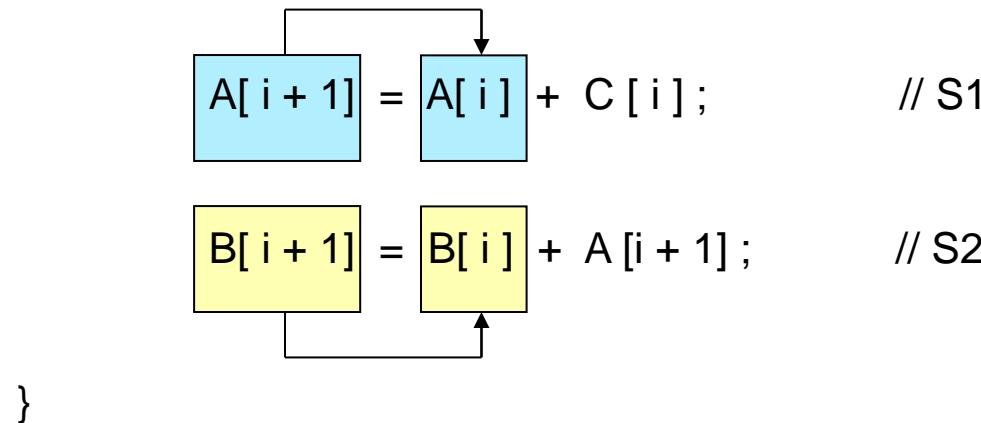
A Loop Independent Dependence

N.B. how do we know $A[i+1]$ and $A[i+1]$ refer to the same location? In general by performing pointer/index variable analysis from conditions known at compile time.

An Example of Loop Level Dependences

- Consider the following loop:

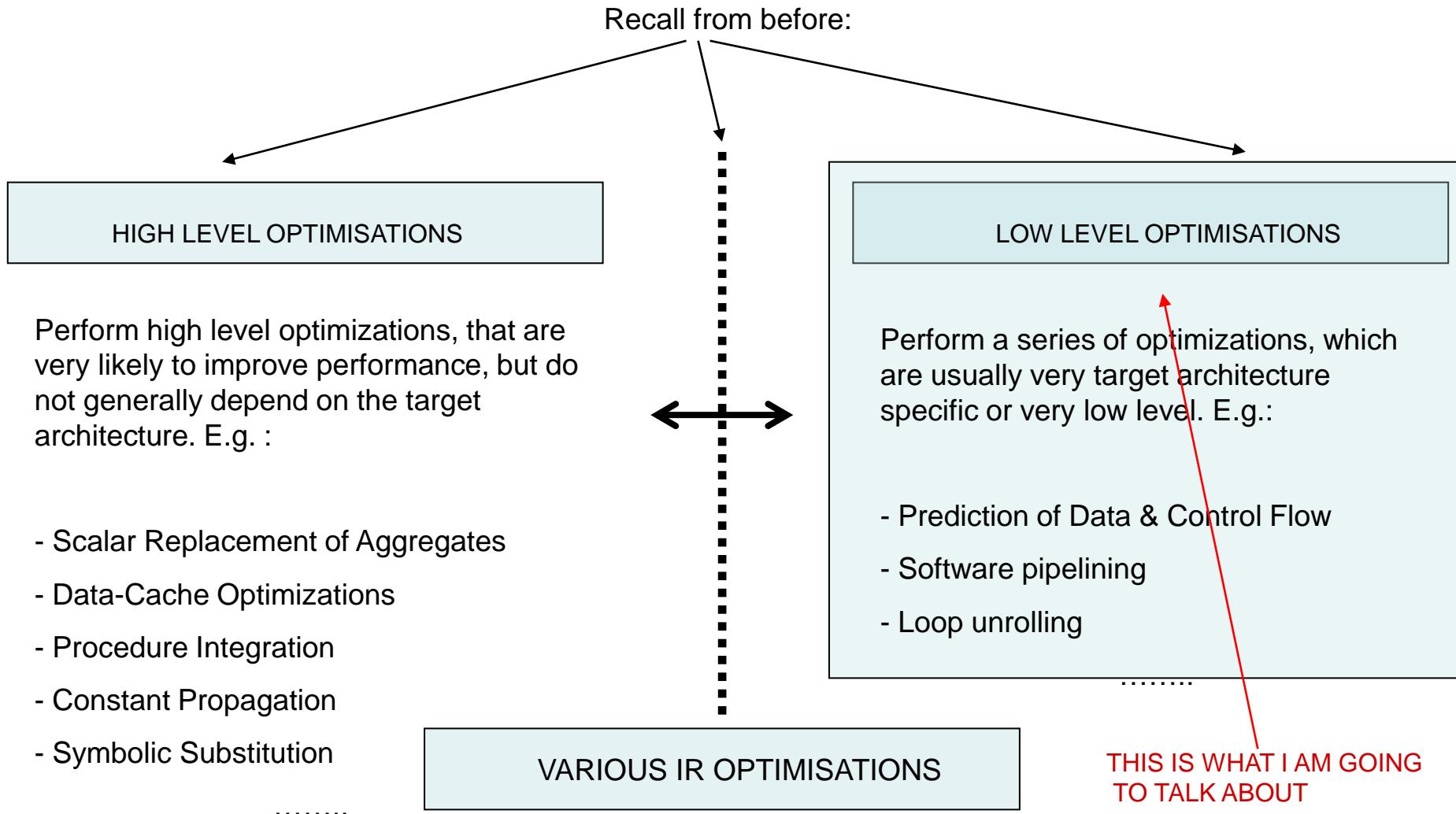
```
for (i = 0; i <= 100; i++) {
```



Two Loop Carried Dependences

We'll make use of these concepts when we talk about software pipelining and loop unrolling !

What are typical transformations?



Let's have a look at some of these in detail !

Outline

- | | | |
|--|---|--------|
| 1. Introduction | → | DONE ☺ |
| 2. Basic Pipeline Scheduling | → | DONE ☺ |
| 3. Instruction Level Parallelism and Dependencies | → | DONE ☺ |
| 4. Local Optimizations and Loops | → | TALK ☹ |
| 5. Global Scheduling Approaches | → | TALK ☹ |
| 6. HW Support for Aggressive Optimization Strategies | → | TALK ☹ |

What are local transformations?

- Transformations which operate on basic blocks or extended basic blocks.
- Our transformations should rearrange code, from data available statically at compile time and from knowledge of the underlying hardware.
- **KEY IDEA:** These transformations do one of the following (or both), while preserving correctness :
 - 1.) Expose more ILP, such that later transformations in the compiler can exploit this exposure.
 - 2.) Perform a rearrangement of instructions, which results in increased performance (measured by execution time, or some other metric of interest)

We will look at two local optimizations, applicable to loops:

STATIC LOOP UNROLLING

Loop Unrolling replaces the body of a loop with several copies of the loop body, thus exposing more ILP.

KEY IDEA:

Reduce loop control overhead and thus increase performance

SOFTWARE PIPELINING

Reschedule instructions from a sequence of loop iterations to enhance ability to exploit more ILP.

KEY IDEA:

Reduce stalls due to data dependencies.

These two are usually complementary in the sense that scheduling of software pipelined instructions usually applies loop unrolling during some earlier transformation to expose more ILP, exposing more potential candidates “to be moved across different iterations of the loop”.

STATIC LOOP UNROLLING

- OBSERVATION: A high proportion of loop instructions executed are loop management instructions (next example should give a clearer picture) on the induction variable.
- KEY IDEA: Eliminating this overhead could potentially significantly increase the performance of the loop:
- We'll use the following loop as our example:

```
for (i = 1000 ; i > 0 ; i -- ) {  
    x[ i ] = x[ i ] + constant;  
}
```

STATIC LOOP UNROLLING (continued) – a trivial translation to MIPS

```
for (i = 1000 ; i > 0 ; i -- ) {
    x[ i ] = x[ i ] + constant;
}
```

Our example translates into the MIPS assembly code below (**without any scheduling**).

Note the loop independent dependence in the loop ,i.e. $x[i]$ on $x[i]$

Loop : L.D F0,0(R1) ; *F0 = array elem.*
 ADD.D F4,F0,F2 ; *add scalar in F2*
 S.D F4,0(R1) ; *store result*
 DADDUI R1,R1,#-8 ; *decrement ptr*
 BNE R1,R2,Loop ; *branch if R1 != R2*

STATIC LOOP UNROLLING (continued)

- Let us assume the following latencies for our pipeline:

INSTRUCTION PRODUCING RESULT	INSTRUCTION USING RESULT	LATENCY (in CC)*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

- Also assume that functional units are fully pipelined or replicated, such that one instruction can issue every clock cycle (assuming it's not waiting on a result!)
- Assume no structural hazards exist, as a result of the previous assumption

STATIC LOOP UNROLLING (continued) – issuing our instructions

- Let us issue the MIPS sequence of instructions obtained:

CLOCK CYCLE ISSUED

→ Loop :	L.D	F0,0(R1)	1
		stall	2
	ADD.D	F4,F0,F2	3
		stall	4
		stall	5
	S.D	F4,0(R1)	6
	DADDUI	R1,R1,#-8	7
		stall	8
	BNE	R1,R2,Loop	9
		stall	10

STATIC LOOP UNROLLING (continued) – issuing our instructions

- Let us issue the MIPS sequence of instructions obtained:

CLOCK CYCLE ISSUED		
→ Loop :	L.D	F0,0(R1) 1
	stall	2
	ADD.D	F4,F0,F2 3
	stall	4
	stall	5
	S.D	F4,0(R1) 6
	DADDUI	R1,R1,#-8 7
	stall	8
	BNE	R1,R2,Loop 9
	stall	10

→ Each iteration of the loop takes 10 cycles!

→ We can improve performance by rearranging the instructions, in the next slide.

We can push S.D. after BNE, if we alter the offset!

We can push ADDUI between L.D. and ADD.D, since R1 is not used anywhere within the loop body (i.e. it's the induction variable)

30

STATIC LOOP UNROLLING (continued) – issuing our instructions

→ Here is the rescheduled loop:

CLOCK CYCLE ISSUED		
Loop :	L.D	F0,0(R1)
	DADDUI	R1,R1,#-8
	ADD.D	F4,F0,F2
	stall	
	BNE	R1,R2,Loop
	S.D	F4,8(R1)

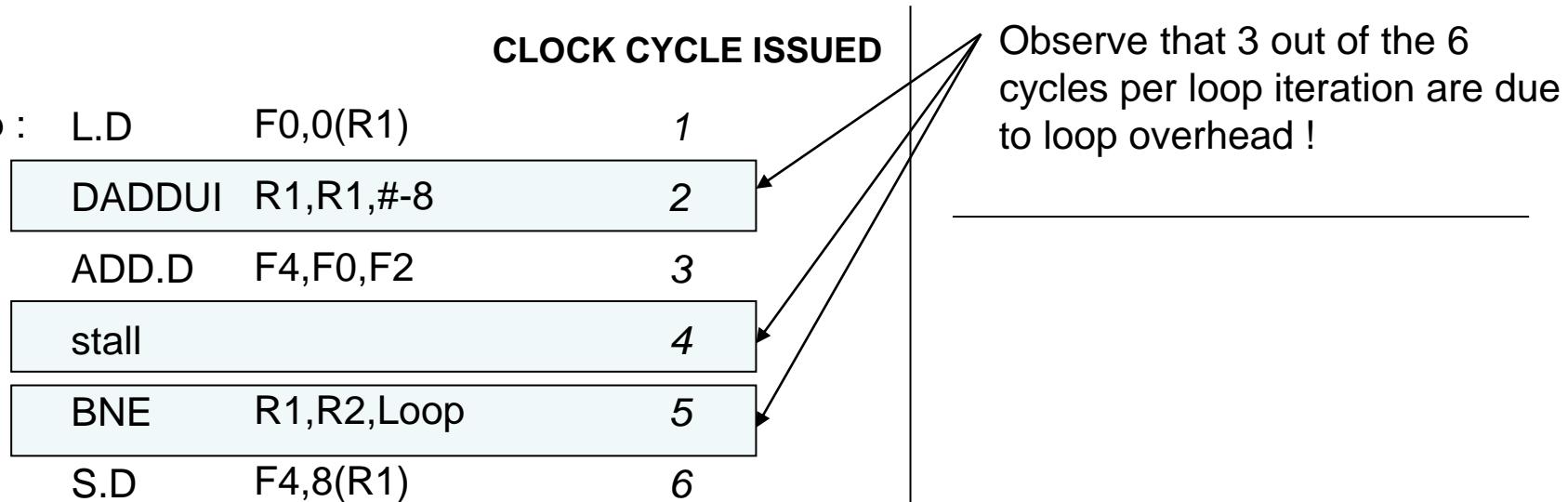
Here we've decremented R1 before we've stored F4. Hence need an offset of 8!

→ Each iteration now takes 6 cycles
→ This is the best we can achieve because of the inherent dependencies and pipeline latencies!

STATIC LOOP UNROLLING (continued) – issuing our instructions

→ Here is the rescheduled loop:

→ Loop : L.D F0,0(R1)



STATIC LOOP UNROLLING (continued)

- Hence, if we could decrease the loop management overhead, we could increase the performance.
- **SOLUTION : Static Loop Unrolling**
 - Make n copies of the loop body, adjusting the loop terminating conditions and perhaps renaming registers (we'll very soon see why!),
 - This results in less loop management overhead, since we effectively merge n iterations into one !
 - This exposes more ILP, since it allows instructions from different iterations to be scheduled together!

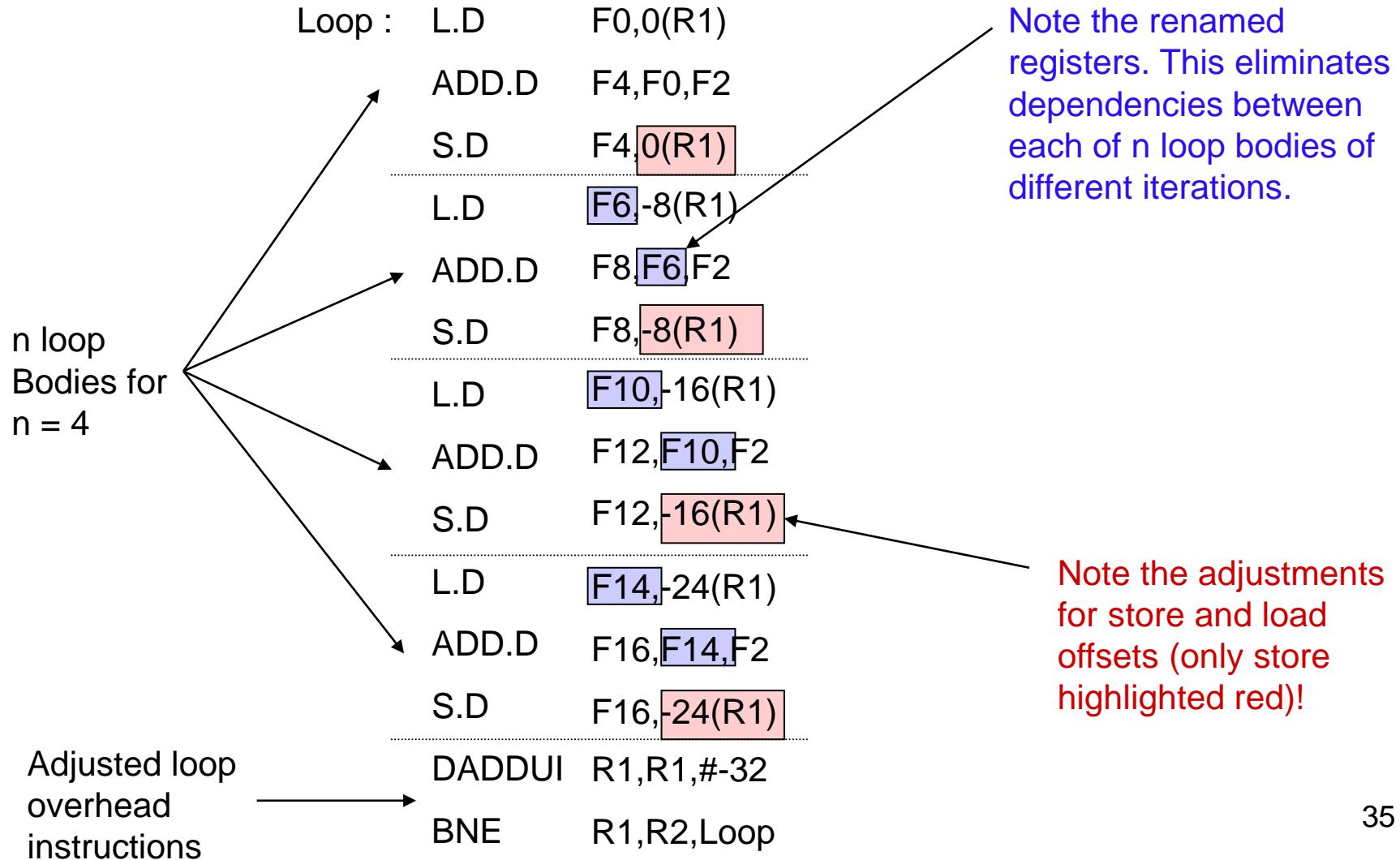
STATIC LOOP UNROLLING (continued) – issuing our instructions

- The unrolled loop from the running example with an unroll factor of $n = 4$ would then be:

Loop :	L.D	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)
	
	L.D	F6,-8(R1)
	ADD.D	F8,F6,F2
	S.D	F8,-8(R1)
	
	L.D	F10,-16(R1)
	ADD.D	F12,F10,F2
	S.D	F12,-16(R1)
	
	L.D	F14,-24(R1)
	ADD.D	F16,F14,F2
	S.D	F16,-24(R1)
	
	DADDUI	R1,R1,#-32
	BNE	R1,R2,Loop

STATIC LOOP UNROLLING (continued) – issuing our instructions

- The unrolled loop from the running example with an unroll factor of $n = 4$ would then be:



STATIC LOOP UNROLLING (continued) – issuing our instructions

→ Let's schedule the unrolled loop on our pipeline:

		CLOCK CYCLE ISSUED
Loop :	L.D F0,0(R1)	1
	L.D F6,-8(R1)	2
	L.D F10,-16(R1)	3
	L.D F14,-24(R1)	4
	ADD.D F4,F0,F2	5
	ADD.D F8,F6,F2	6
	ADD.D F12,F10,F2	7
	ADD.D F16,F14,F2	8
	S.D F4,0(R1)	9
	S.D F8,-8(R1)	10
	DADDUI R1,R1,#-32	11
	S.D F12,16(R1)	12
	BNE R1,R2,Loop	13
	S.D F16,8(R1);	14

STATIC LOOP UNROLLING (continued) – issuing our instructions

→ Let's schedule the unrolled loop on our pipeline:

This takes 14 cycles for 1 iteration of the unrolled loop.

Therefore w.r.t. original loop we now have $14/4 = 3.5$ cycles per iteration.

Previously 6 was the best we could do!

→ We gain an increase in performance, at the expense of extra code and higher register usage/pressure

→ The performance gain on superscalar architectures would be even higher!

CLOCK CYCLE ISSUED

Loop :	L.D	F0,0(R1)	1
	L.D	F6,-8(R1)	2
	L.D	F10,-16(R1)	3
	L.D	F14,-24(R1)	4
	ADD.D	F4,F0,F2	5
	ADD.D	F8,F6,F2	6
	ADD.D	F12,F10,F2	7
	ADD.D	F16,F14,F2	8
	S.D	F4,0(R1)	9
	S.D	F8,-8(R1)	10
	DADDUI	R1,R1,#-32	11
	S.D	F12,16(R1)	12
	BNE	R1,R2,Loop	13
	S.D	F16,8(R1);	374

STATIC LOOP UNROLLING (continued)

However loop unrolling has some significant complications and disadvantages:

- Unrolling with an unroll factor of n , increases the code size by (approximately) n . This might present a problem,
- Imagine unrolling a loop with a factor $n= 4$, that is executed a number of times that is not a multiple of four:
 - one would need to provide a copy of the original loop and the unrolled loop,
 - this would increase code size and management overhead significantly,
 - this is a problem, since we usually don't know the upper bound (UB) on the induction variable (which we took for granted in our example),
 - more formally, the original copy should be included if $(UB \bmod n \neq 0)$, i.e. number of iterations is not a multiple of the unroll factor

STATIC LOOP UNROLLING (continued)

However loop unrolling has some significant complications and disadvantages:

- We usually *ALSO* need to perform register renaming to reduce dependencies within the unrolled loop. This increases the register pressure!
- The criteria for performing loop unrolling are therefore usually very restrictive!

SOFTWARE PIPELINING

- Software Pipelining is an optimization that can improve the loop-execution-performance of any system that allows ILP, including VLIW and superscalar architectures,
 - It derives its performance gain by filling delays within each iteration of a loop body with instructions from different iterations of that same loop,
 - This method requires fewer registers per loop iteration than loop unrolling,
 - This method requires some extra code to fill (preheader) and drain (postheader) the software pipelined loop, as we'll see in the next example.
- KEY IDEA: Increase performance by scheduling instructions from different iterations into a single iteration of the loop.

SOFTWARE PIPELINING

- Consider the instruction sequence from before:

```
Loop :    L.D      F0,0(R1)    ; F0 = array elem.
          ADD.D    F4,F0,F2    ; add scalar in F2
          S.D      F4,0(R1)    ; store result
          DADDUI  R1,R1,#-8    ; decrement ptr
          BNE     R1,R2,Loop  ; branch if R1 != R2
```

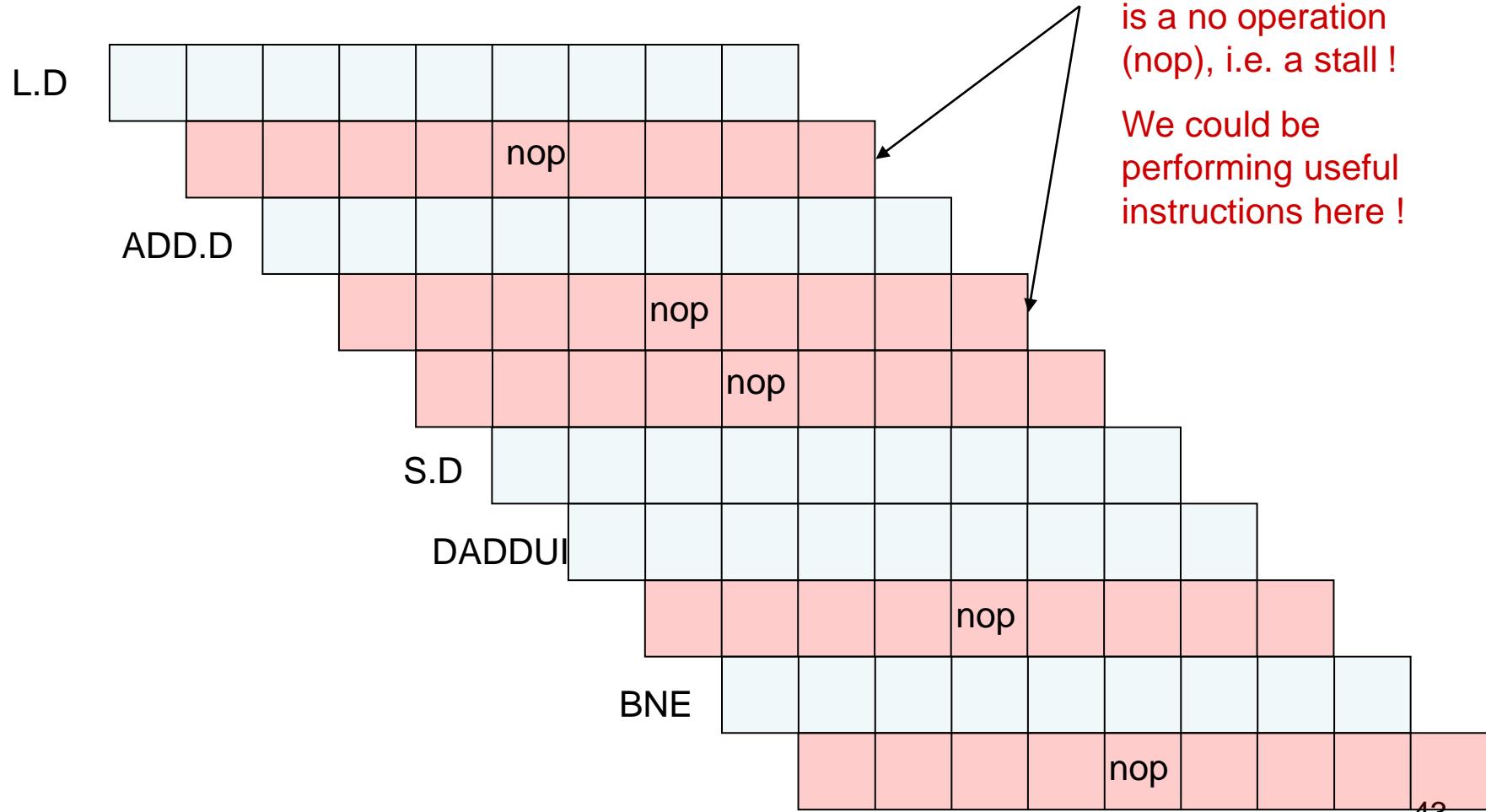
SOFTWARE PIPELINING

- Which was executed in the following sequence on our pipeline:

Loop :	L.D	F0,0(R1)	1
		stall	2
	ADD.D	F4,F0,F2	3
		stall	4
		stall	5
	S.D	F4,0(R1)	6
	DADDUI	R1,R1,#-8	7
		stall	8
	BNE	R1,R2,Loop	9
		stall	10

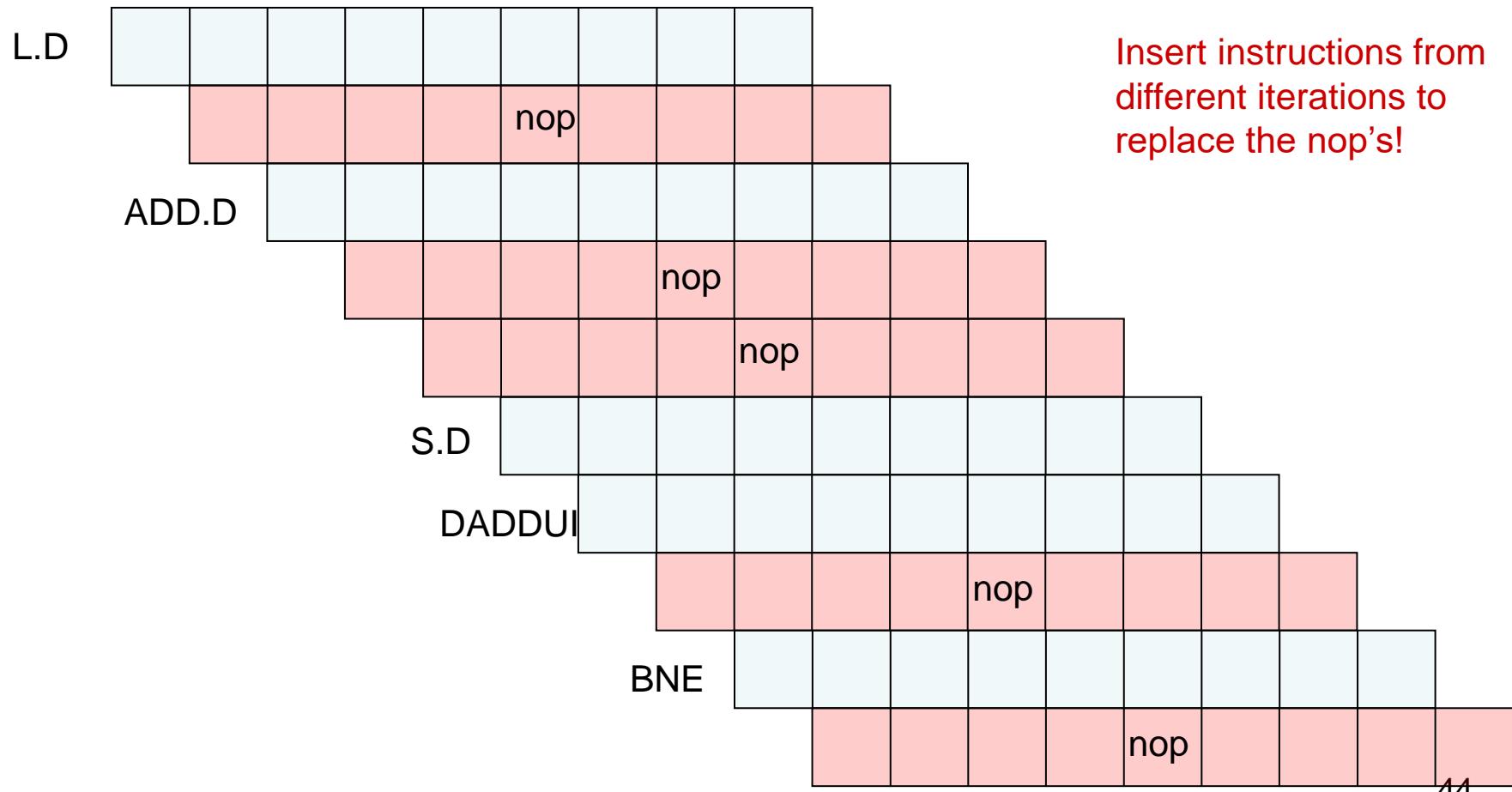
SOFTWARE PIPELINING

- A pipeline diagram for the execution sequence is given by:



SOFTWARE PIPELINING

- Software pipelining eliminates nop's by inserting instructions from different iterations of the same loop body:



SOFTWARE PIPELINING

→ How is this done?

- 1 → unroll loop body with an unroll factor of n. we'll take n = 3 for our example
- 2 → select order of instructions from different iterations to pipeline
- 3 → “paste” instructions from different iterations into the new pipelined loop body

Let's schedule our running example (repeated below) with software pipelining:

Loop :	L.D	F0,0(R1)	<i>; F0 = array elem.</i>
	ADD.D	F4,F0,F2	<i>; add scalar in F2</i>
	S.D	F4,0(R1)	<i>; store result</i>
	DADDUI	R1,R1,#-8	<i>; decrement ptr</i>
	BNE	R1,R2,Loop	<i>; branch if R1 != R2</i>

SOFTWARE PIPELINING

→ **Step 1** → unroll loop body with an unroll factor of n. we'll take n = 3 for our example

Iteration i:	L.D	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)
Iteration i + 1:	L.D	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)
Iteration i + 2:	L.D	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)

Notes:

1.) We are unrolling the loop body
Hence no loop overhead
Instructions are shown!

2.) There three iterations will be
“collapsed” into a single loop body
containing instructions from
different iterations of the original
loop body.

SOFTWARE PIPELINING

→ **Step 2** → select order of instructions from different iterations to pipeline

Iteration i: L.D F0,0(R1)
 ADD.D F4,F0,F2

Iteration i + 1: L.D F0,0(R1)

 ADD.D F4,F0,F2

 S.D F4,0(R1)

Iteration i + 2: L.D F0,0(R1)

 ADD.D F4,F0,F2

 S.D F4,0(R1)

Notes:

1.)

1.) We'll select the following order in our pipelined loop:

2.)

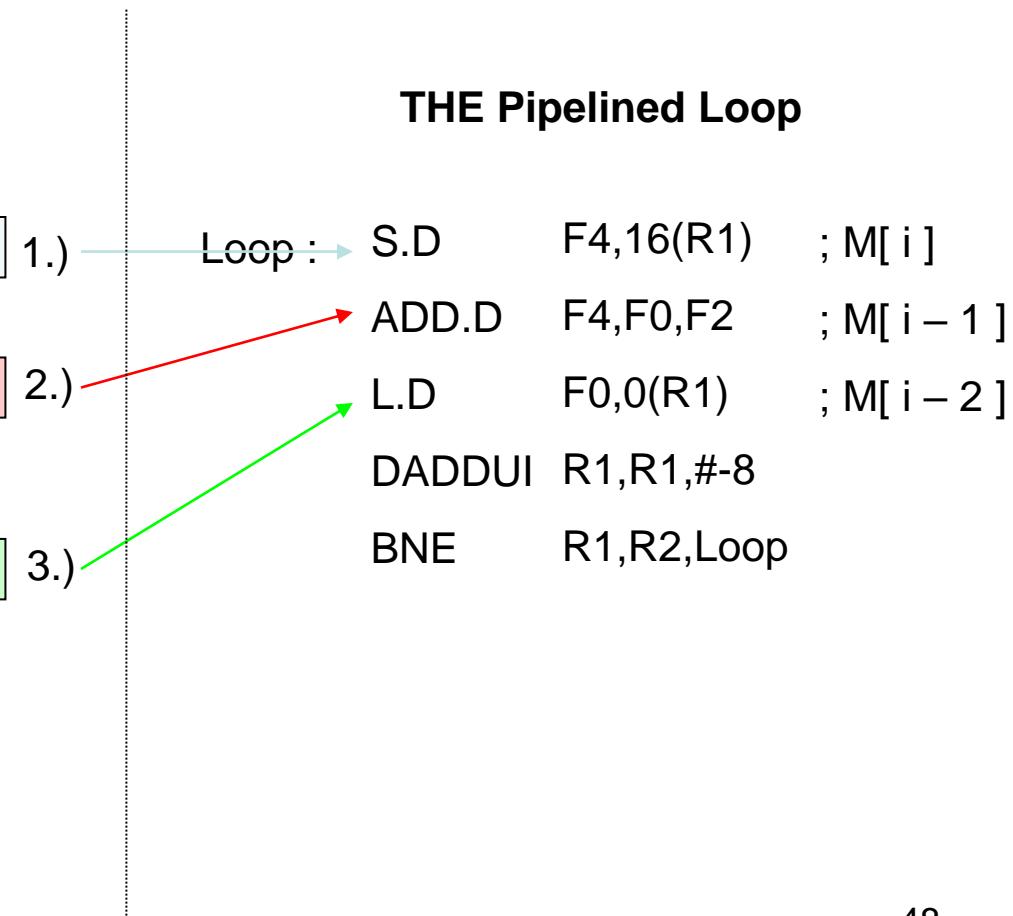
2.) Each instruction (L.D ADD.D S.D) must be selected at least once to make sure that we don't leave out any instructions when we collapse the loop on the left into a single pipelined loop.

3.)

SOFTWARE PIPELINING

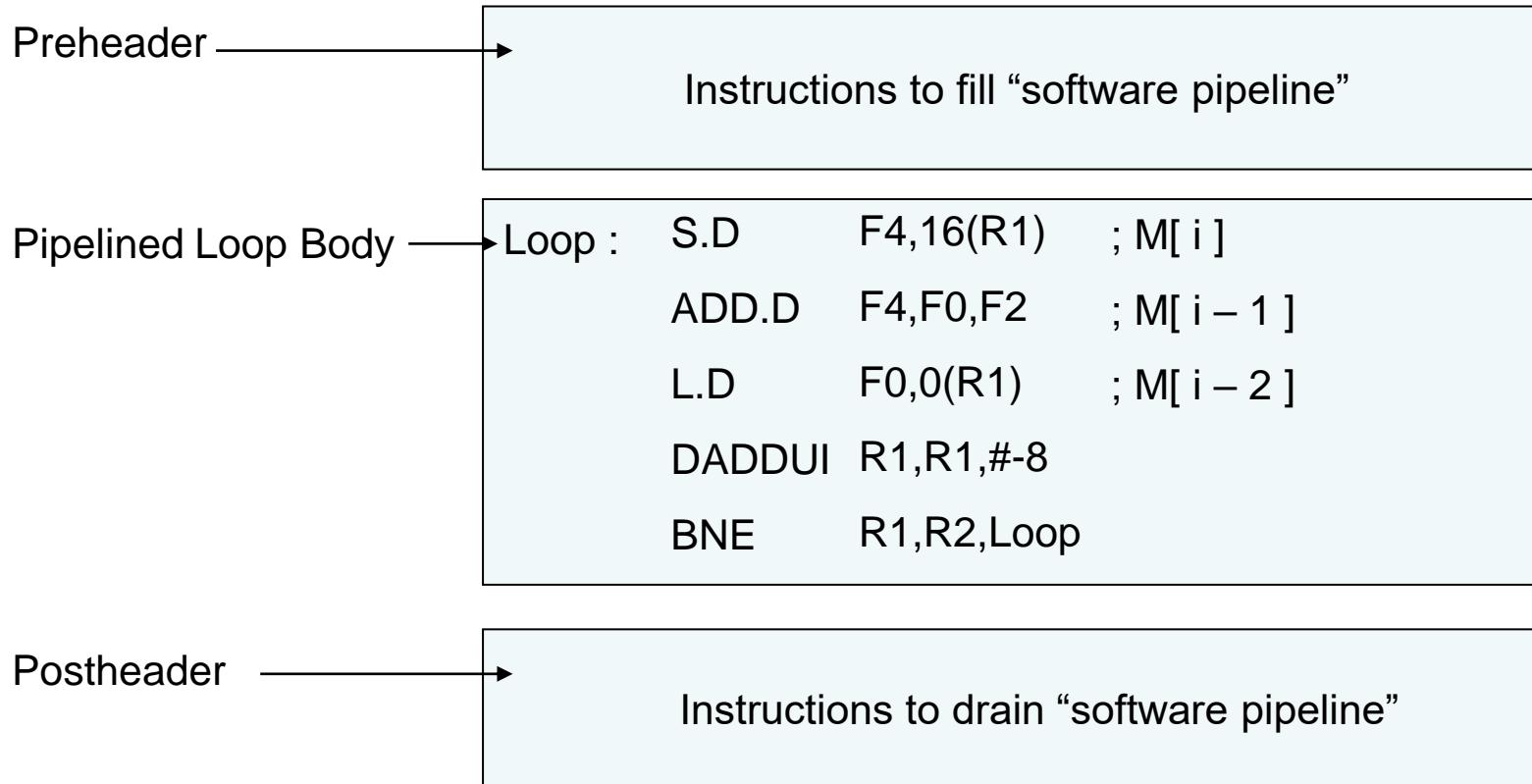
→ Step 3 → “paste” instructions from different iterations into the new pipelined loop body

Iteration i:	L.D	F0,0(R1)	
	ADD.D	F4,F0,F2	
	S.D	F4,0(R1)	1.)
Iteration i + 1:	L.D	F0,0(R1)	
	ADD.D	F4,F0,F2	2.)
	S.D	F4,0(R1)	
Iteration i + 2:	L.D	F0,0(R1)	3.)
	ADD.D	F4,F0,F2	
	S.D	F4,0(R1)	



SOFTWARE PIPELINING

- Now we just insert a loop preheader & postheader and the pipelined loop is finished:



SOFTWARE PIPELINING

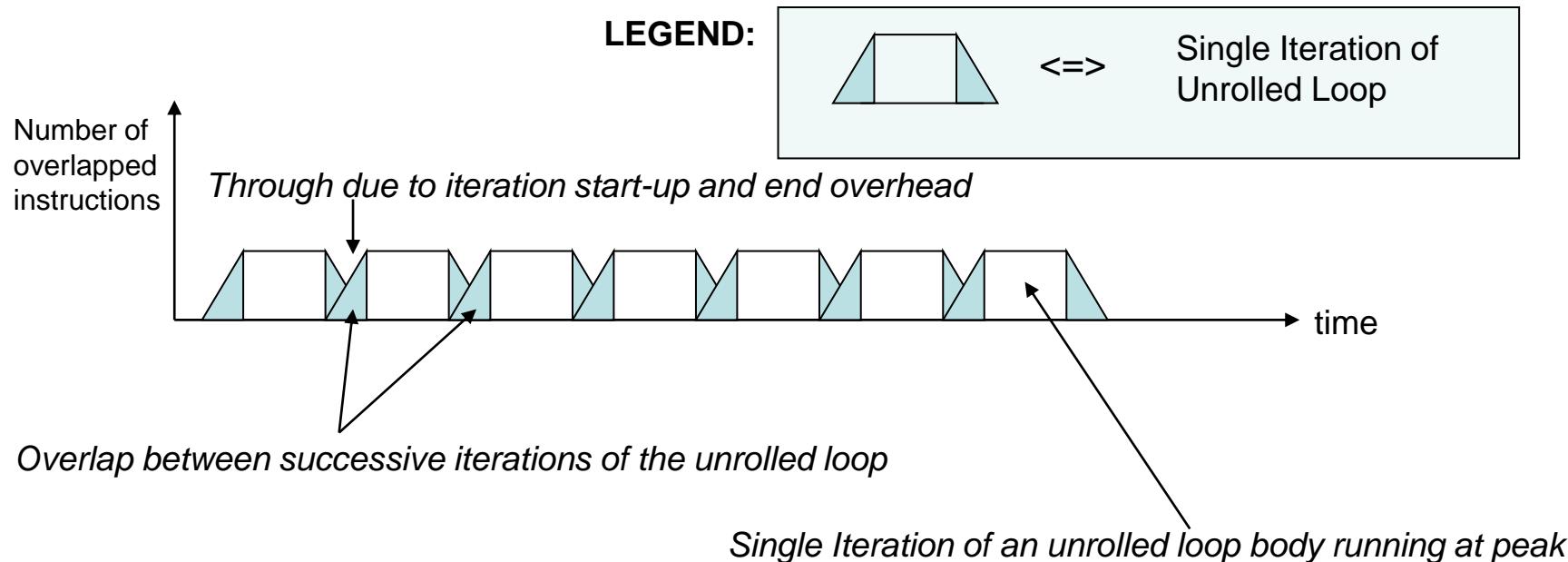
```
Loop : S.D      F4,16(R1)    ; M[ i ]
          ADD.D    F4,F0,F2    ; M[ i - 1 ]
          L.D      F0,0(R1)    ; M[ i - 2 ]
          DADDUI  R1,R1,#-8
          BNE     R1,R2,Loop
```

- Assuming we reschedule the last 2 (iteration) steps, our pipelined loop can run in 5 cycles per iteration (steady state), which is better than the initial running time of 6 cycles per iteration, but less than the 3.5 cycles achieved with loop unrolling
- Software pipelining can be thought of as symbolic loop unrolling, which is analogous to executing Tomasulo's algorithm in software

SOFTWARE PIPELINING & LOOP UNROLLING: A Comparison

LOOP UNROLLING

- Consider the parallelism (in terms of overlapped instructions) vs. time curve for a loop that is scheduled using loop unrolling:

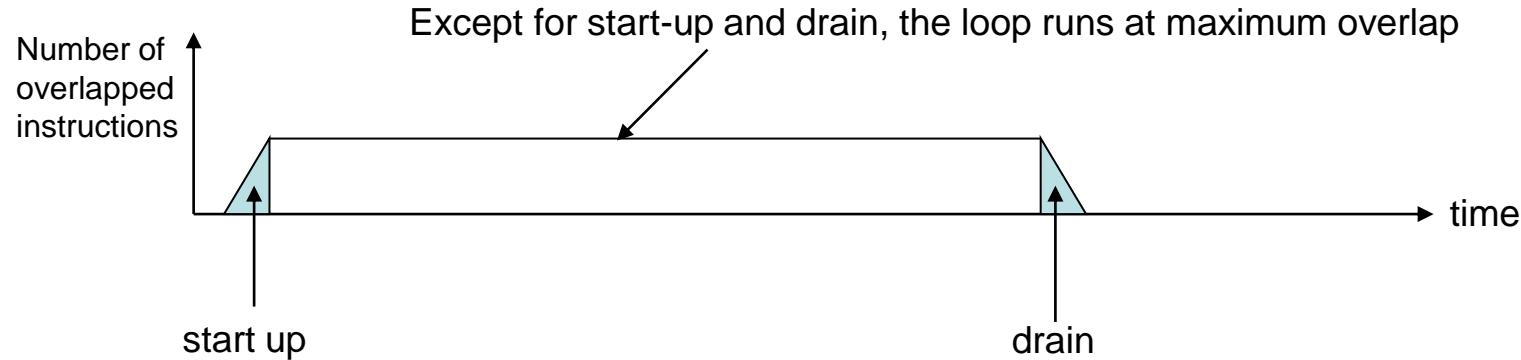


- The unrolled loop does not run at maximum overlap, due to entry and exit overhead associated with each iteration of the unrolled loop.
- A Loop with an unroll factor of n , and m iterations when run, will incur m/n non-maximal throughs

SOFTWARE PIPELINING & LOOP UNROLLING: A Comparison

SOFTWARE PIPELINING

- In contrast, software pipelining only incurs a penalty during start up (pre-header) and drain (post-header):



- The pipelined loop only incurs non-maximum overlap during start up and drain, since we're pipelining instructions from different iterations and thus minimize the stalls arising from dependencies between different iterations of the pipelined loop.

Outline

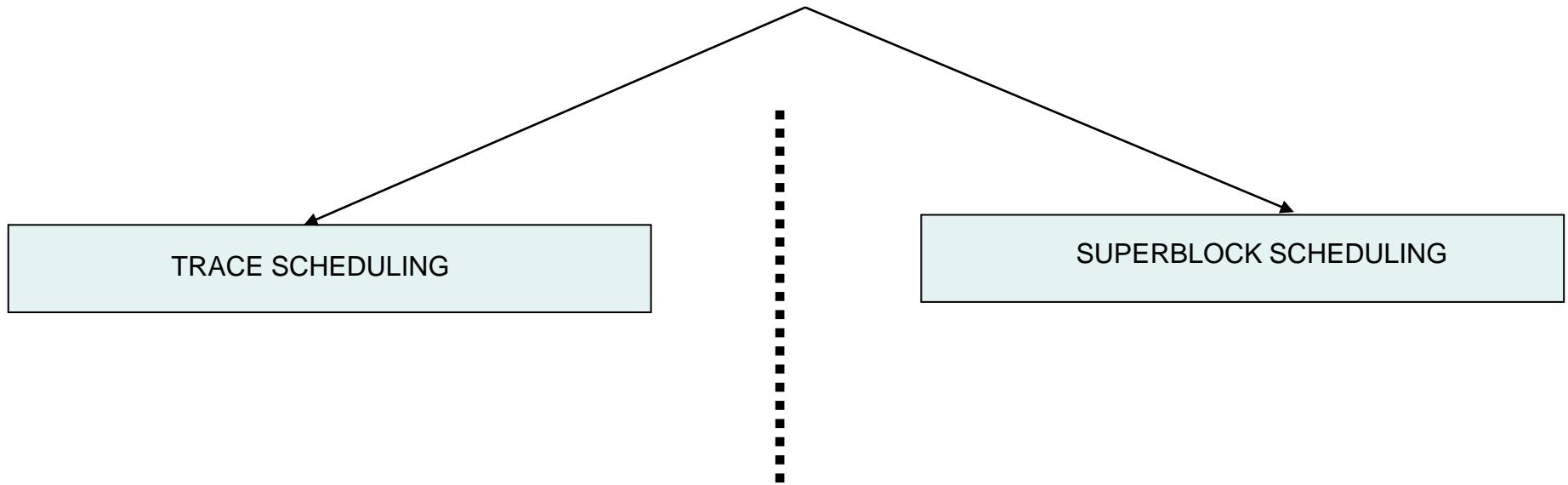
- | | | |
|--|---|--------|
| 1. Introduction | → | DONE ☺ |
| 2. Basic Pipeline Scheduling | → | DONE ☺ |
| 3. Instruction Level Parallelism and Dependencies | → | DONE ☺ |
| 4. Local Optimizations and Loops | → | DONE ☺ |
| 5. Global Scheduling Approaches | → | TALK ☹ |
| 6. HW Support for Aggressive Optimization Strategies | → | TALK ☹ |

Global Scheduling Approaches

- The approaches seen so far work well with linear code segments,
- For programs with more complex control flow (i.e. more branching), our approaches so far are not very effective, since we cannot move code across (non-LOOP) branches,
- Hence we would ideally like to be able to move instructions across branches,
- Global scheduling approaches perform code movement across branches, based on the relative frequency of execution across different control flow paths,
- This approach must deal with both control dependencies (on branches) and data dependencies that exist within and across basic blocks,
- Since static global scheduling is subject to numerous constraints, hardware approaches exist for either eliminating (speculative execution) or supporting compile-time scheduling, as we'll see in the next section.

Global Scheduling Approaches:

We will briefly look at two common global scheduling approaches



- Both approaches are suited to scientific code with intensive loops and accurate profile data,
- Both approaches incur heavy penalties for control flow that does not follow the predicted flow of control,
- The latter is a consequence of moving any overhead associated with global instruction movement to less frequented blocks of code.

Trace Scheduling

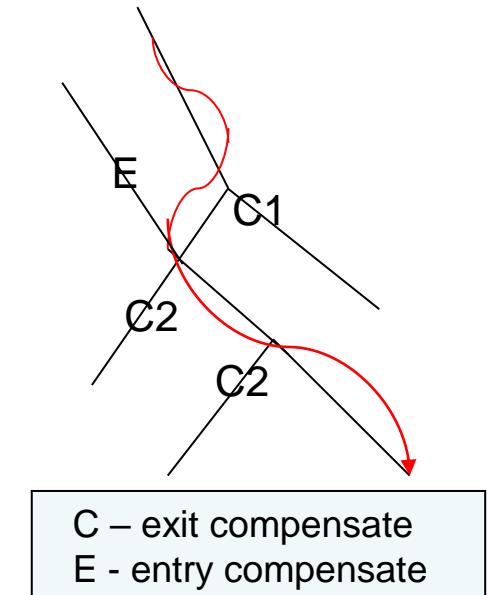
Two Steps:

1.) Trace Selection

- Find likely sequence of basic blocks (**trace**) of (statically predicted or profile predicted) long sequence of straight line code

2.) Trace Compaction

- Try to schedule instructions along the trace as early as possible within the trace. On VLIW processors, this also implies packing the instructions into as few instructions as possible

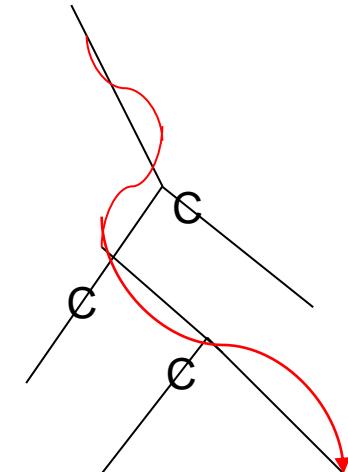


- Since we move instructions, along the trace, between basic blocks, compensating code is inserted along control flow edges that are not included in the trace to guarantee program correctness,
- This means that for control flow deviation from the trace, we are very likely to incur heavy penalties,
- Trace scheduling essentially treats each branch as a jump, hence we gain a performance enhancement if we select a trace indicative of program flow behaviour. If we are wrong in our guess, the compensating code is likely to adversely affect behaviour.

Superblock Scheduling (for loops)

Problems with trace scheduling:

- In trace scheduling entries into the middle of a trace cause significant problems, since we need to place compensating code at each entry,
 - **Superblock** scheduling groups the basic blocks along a trace into extended basic blocks that contain one entry point and multiple exits
 - When the trace is left, we only provide one piece of code C for the remaining iterations of the loop
-



- The underlying assumption is that the compensating code C will not be executed frequently. If it is, then creating a superblock out of C is a possible option
- This approach significantly reduces the bookkeeping associated with trace scheduling
- It can, however, lead to larger code increases than for trace scheduling
- Allows a better estimate of the cost of compensating code C, since we are now dealing with one piece of compensating code

Outline

- | | | |
|--|---|--------|
| 1. Introduction | → | DONE ☺ |
| 2. Basic Pipeline Scheduling | → | DONE ☺ |
| 3. Instruction Level Parallelism and Dependencies | → | DONE ☺ |
| 4. Local Optimizations and Loops | → | DONE ☺ |
| 5. Global Scheduling Approaches | → | DONE ☺ |
| 6. HW Support for Aggressive Optimization Strategies | → | TALK ☹ |

HW Support for exposing more ILP at compile-time

- The techniques seen so far produce potential improvements in execution time but are subject to numerous criteria that must be satisfied before they can be safely applied.
- If our “applicability criteria” fail, then a conservative guess is the best that we can do (so far).
- It is desirable to provide supporting hardware mechanisms that preserve correctness at run time while improving our ability to speculate effectively:

We will briefly have a look at predicated instructions, which allow us to speculate more effectively in the presence of control dependencies

Predicated Instructions

- Consider the following code:

If (A == 0) {S = T;}

- Which we can translate to MIPS as follows (assuming R1,R2,R3 hold A,S,T respectively) :

BNEZ R1,L
ADDU R2,R3,R0

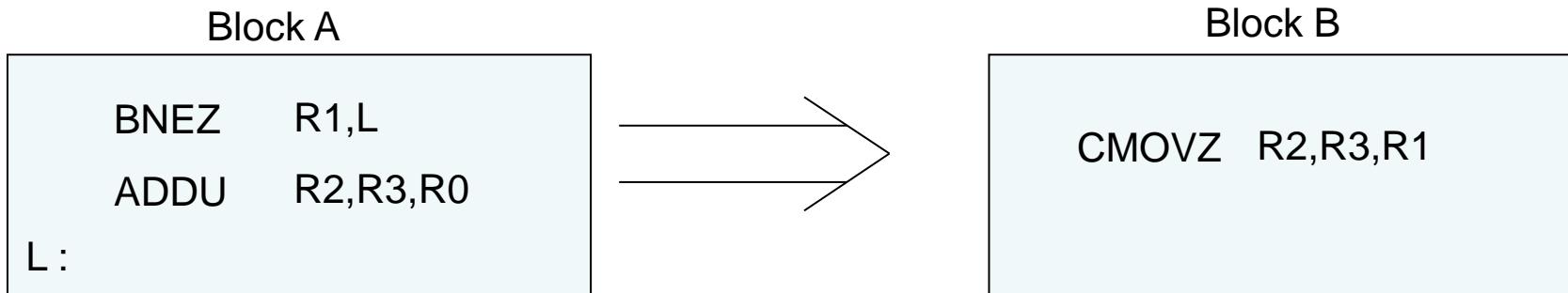
L :

- With support for predicated instructions, the above C code would translate to :

CMOVZ R2,R3,R1 ; if (R1 == 0) move R3 to R2

Predicated Instructions

- We hence performed the following transformation in the last example (a.k.a. if-conversion) :



- What are the implications?

- 1.) we have converted a control dependence in Block A to a data dependence (subject to evaluation of the condition on R1),
- 2.) we have effectively moved the resolution location from the front end of the pipeline (for control dependencies) to the end (for data dependencies),
- 3.) this reduces the number of branches, creating a linear code segment, thus exposing more ILP.

Predicated Instructions

→ What are the implications? (continued)

4.) we have effectively reduced branch pressure, which otherwise might have prevented issue of the second instruction (depending on architecture)

HOWEVER:

5.) usually a whole block is control dependent on a branch. Thus all instructions within that block would need to be predicated, which can be very inefficient

→ this might be solved with full predication, where every instruction is predicated !

6.) annulations of an instruction (whose condition evaluates to be false) is usually done late in the pipeline to allow sufficient time for condition evaluation.

→ this however means, that annulled instructions effectively reduce our CPI. If there are too many (e.g. when predication large blocks), we might be faced with significant performance losses

Predicated Instructions

→ What are the implications? (continued)

- 7.) since predicated instructions introduce data dependencies on the condition evaluation, we might be subject to additional stalls while waiting for the data hazard on the condition to be cleared!
- 8.) Since predicated instructions perform more work than normal instructions (i.e. might require to be pipeline-resident for more clock cycles due to higher workload) in the instruction set, these might lead to an overall increase of the CPI of the architecture.

Hence most current architectures include just a few simple predicated instructions

Outline

- | | | |
|--|---|--------|
| 1. Introduction | → | DONE ☺ |
| 2. Basic Pipeline Scheduling | → | DONE ☺ |
| 3. Instruction Level Parallelism and Dependencies | → | DONE ☺ |
| 4. Local Optimizations and Loops | → | DONE ☺ |
| 5. Global Scheduling Approaches | → | DONE ☺ |
| 6. HW Support for Aggressive Optimization Strategies | → | DONE ☺ |

Just a brief summary to go!

SUMMARY

- 1.) Compile-time optimizations provide a number of analysis-intensive optimizations that otherwise could not be performed at run time due to the high overhead associated with the analysis.
- 2.) Compiler based approaches are usually limited by the inaccuracy or unavailability of run-time data and control flow behaviour.
- 3.) Compilers can reorganize code such that more ILP is exposed for further optimization or exploitation at run time.

CONCLUSION:

- 1.) The most efficient approach is a hardware-software co-scheduling approach, where the hardware and compiler co-operatively exploit as much information as possible within the respective restrictions of each approach.
- 2.) Such an approach is most likely to produce high performance!

REFERENCES

1. "Computer Architecture: A Quantitative Approach".
J.L. Hennessy & D.A. Patterson.
Morgan Kaufmann Publishers, 3rd Edition.
2. "Optimizing Compilers for Modern Architectures".
S. Muchnik.
Morgan Kaufmann Publishers, 2nd Edition.
3. "Advanced Compiler Design & Implementation".
S. Muchnik.
Morgan Kaufmann Publishers, 2nd Edition.
4. "Compilers: Principles, Techniques and Tools":
A.V. Aho, R. Sethi, J.D. Ullman.
Addision Wesly Longman Publishers, 2nd Edition.

The Roofline Model: A pedagogical tool for program analysis and optimization

ParLab Summer Retreat

Samuel Williams, David Patterson

samw@cs.berkeley.edu

- ❖ Performance and scalability of multicore architectures can be extremely non-intuitive to novice programmers
- ❖ Success of the multicore paradigm should be premised on augmenting the abilities of the world's programmers

- ❖ Focused on:
rates and efficiencies (Gflop/s, % of peak),

- ❖ Goals for Roofline:
 - Provide everyone with a graphical aid that provides:
realistic expectations of performance and productivity
 - Show inherent hardware limitations for a given kernel
 - Show potential benefit and priority of optimizations

- ❖ Who's not the audience for the Roofline:
 - Not for those interested in fine tuning (+5%)
 - Not for those challenged by parallel kernel correctness

Principal Components of Performance

- ❖ There are three principal components to performance:
 - **Computation**
 - **Communication**
 - **Locality**
- ❖ Each architecture has a different balance between these
- ❖ Each kernel has a different balance between these
- ❖ Performance is a question of how well an kernel's characteristics map to an architecture's characteristics

- ❖ For us, floating point performance (**Gflop/s**) is the metric of interest (typically double precision)
- ❖ Peak in-core performance can only be attained if:
 - fully exploit ILP, DLP, FMA, etc...
 - non-FP instructions don't sap instruction bandwidth
 - threads don't diverge (GPUs)
 - transcendental/non pipelined instructions are used sparingly
 - branch mispredictions are rare
- ❖ To exploit a form of in-core parallelism, it must be:
 - Inherent in the algorithm
 - Expressed in the high level implementation
 - Explicit in the generated code

- ❖ For us, DRAM bandwidth (**GB/s**) is the metric of interest
- ❖ Peak bandwidth can only be attained if certain optimizations are employed:
 - Few unit stride streams
 - NUMA allocation and usage
 - SW Prefetching
 - Memory Coalescing (GPU)

- ❖ Computation is free, Communication is expensive.
- ❖ Maximize locality to minimize communication
- ❖ **There is a lower limit to communication: compulsory traffic**

- ❖ Hardware changes can help minimize communication
 - Larger cache capacities minimize capacity misses
 - Higher cache associativities minimize conflict misses
 - Non-allocating caches minimize compulsory traffic

- ❖ Software optimization can also help minimize communication
 - Padding avoids conflict misses
 - Blocking avoids capacity misses
 - Non-allocating stores minimize compulsory traffic

Roofline Model

- ❖ Goal: integrate in-core performance, memory bandwidth, and locality into a single readily understandable performance figure
- ❖ Also, must graphically show the penalty associated with not including certain software optimizations

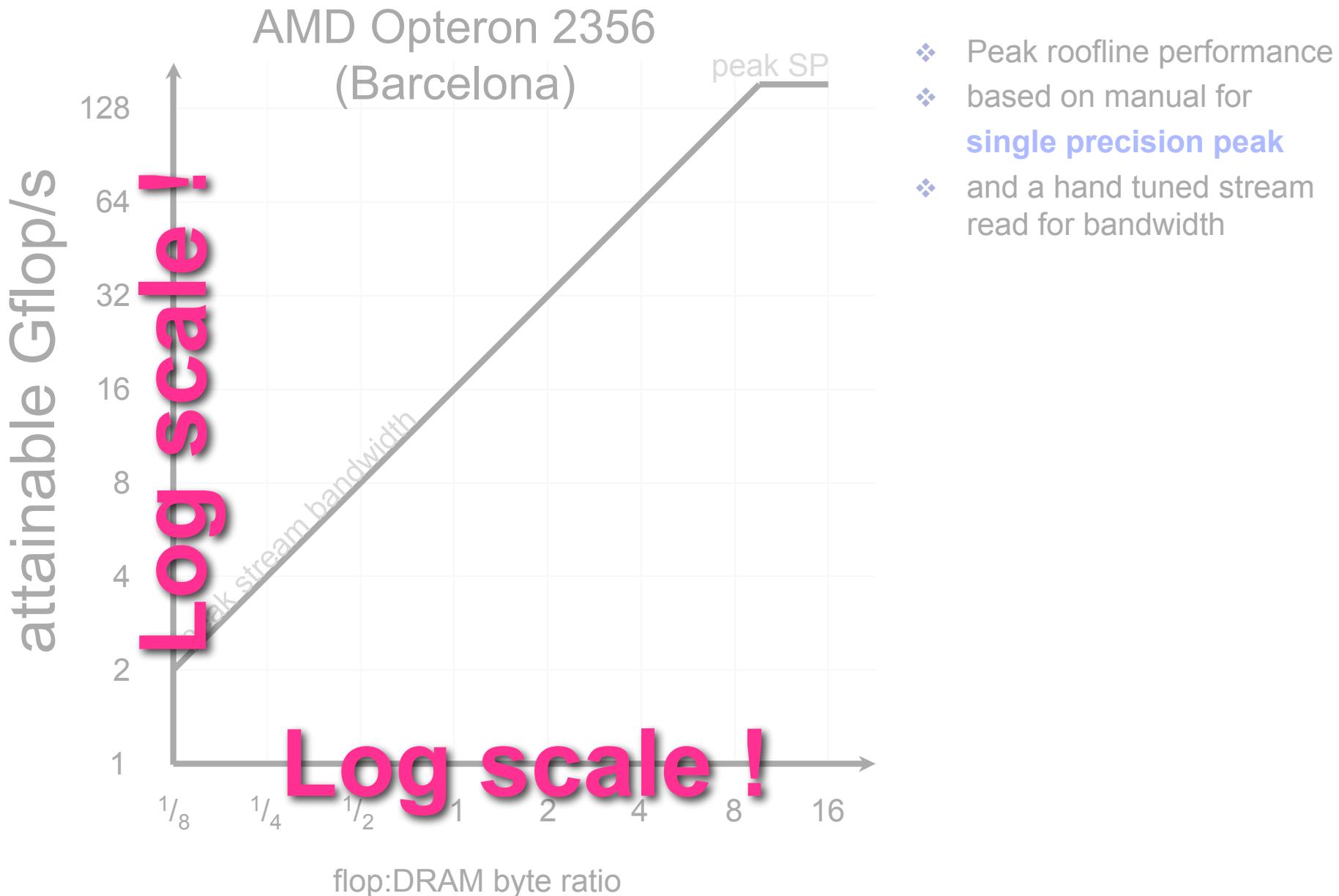
- ❖ Roofline model will be unique to each architecture
- ❖ Coordinates of a kernel are ~unique to each architecture

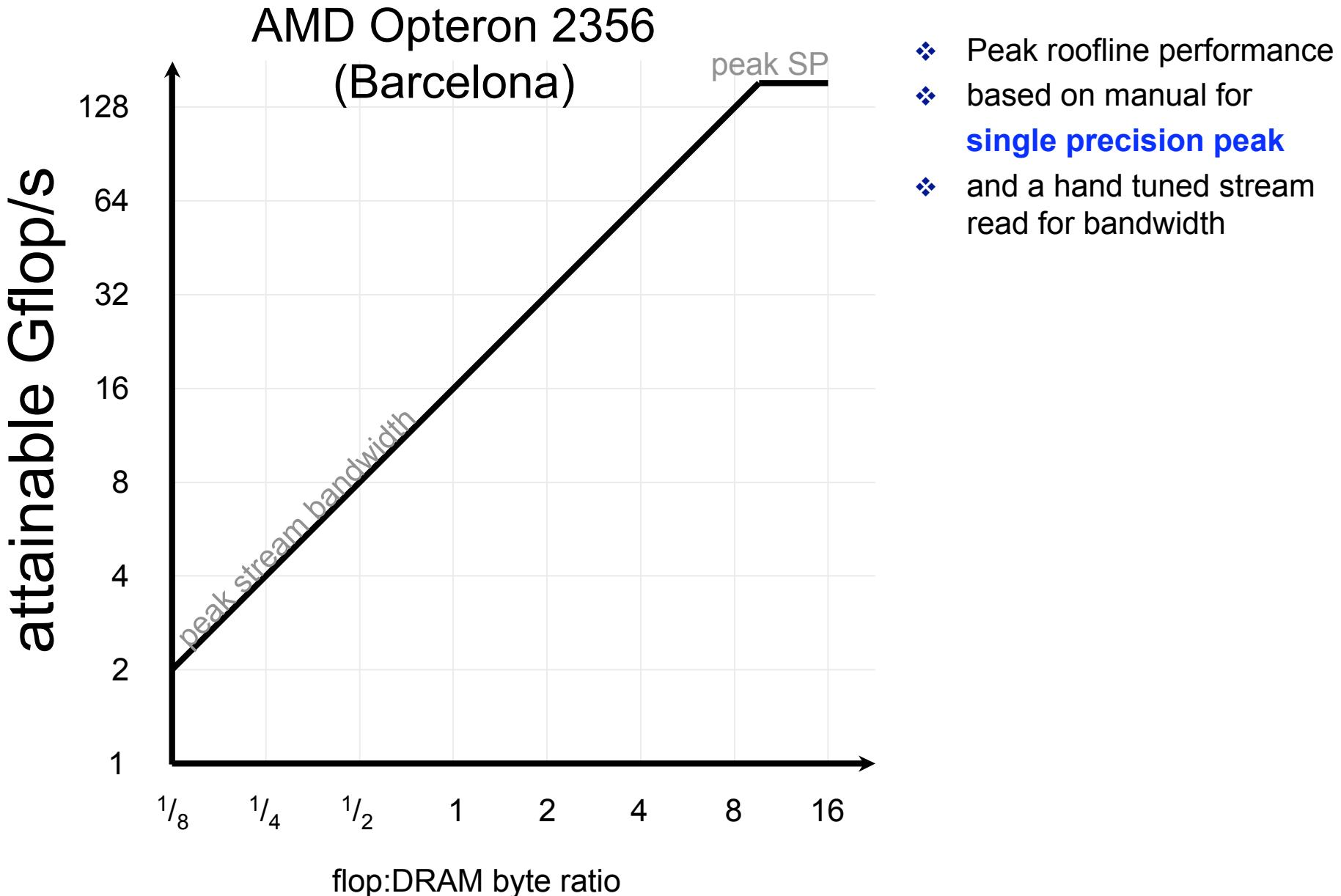
- ❖ Through dimensional analysis, its clear that **Flops:Bytes** is the parameter that allows us to convert bandwidth (GB/s) to performance (GFlop/s)
- ❖ This is a well known quantity: **Arithmetic Intensity** (discussed later)
- ❖ When we measure total bytes, we incorporate all cache behavior (the 3C's) and Locality

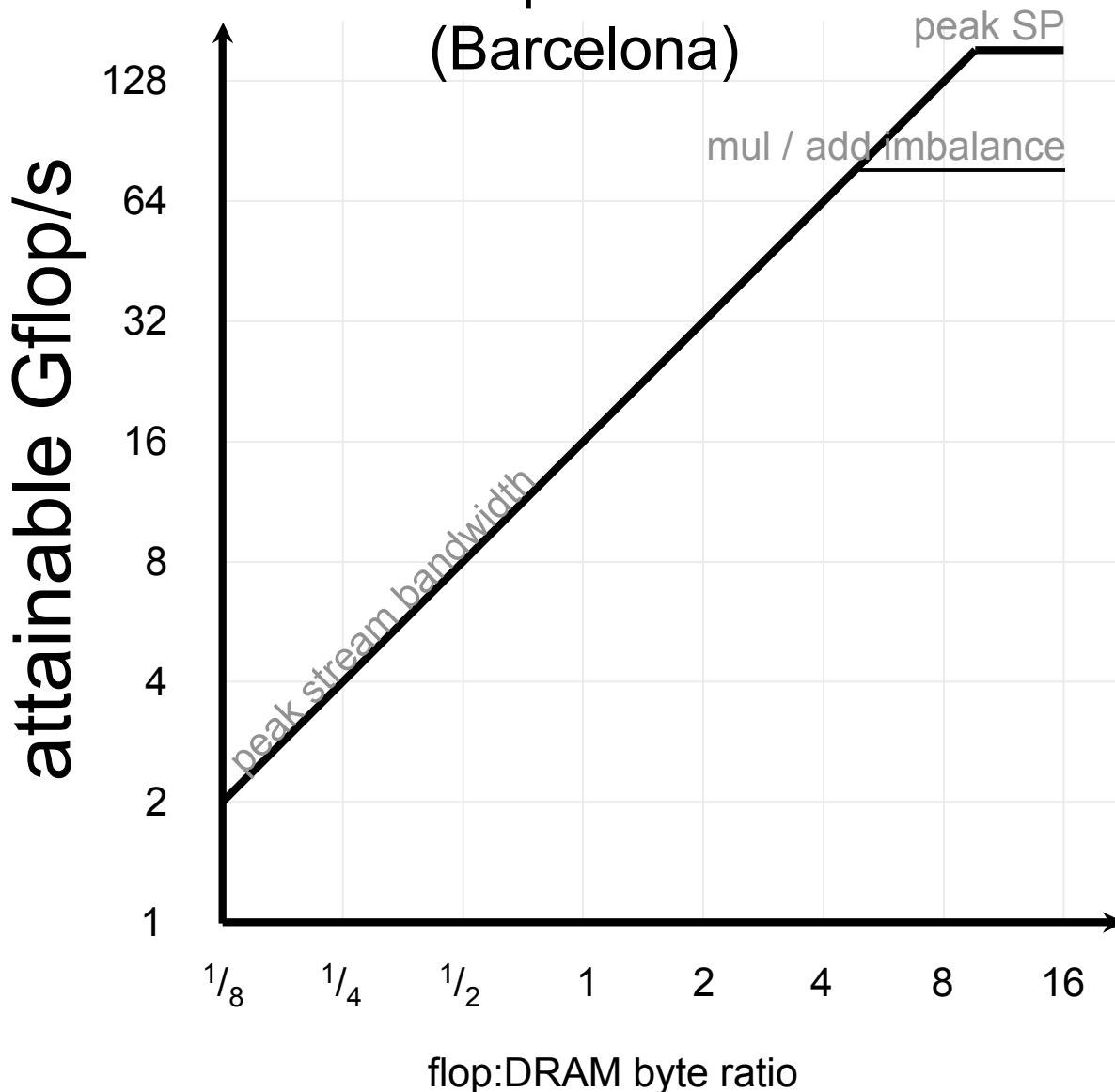
- ❖ Performance is upper bounded by both the peak flop rate, and the product of streaming bandwidth and the flop:byte ratio

$$\text{Gflop/s} = \min \left\{ \begin{array}{l} \text{Peak Gflop/s} \\ \text{Stream BW} * \text{actual flop:byte ratio} \end{array} \right\}$$

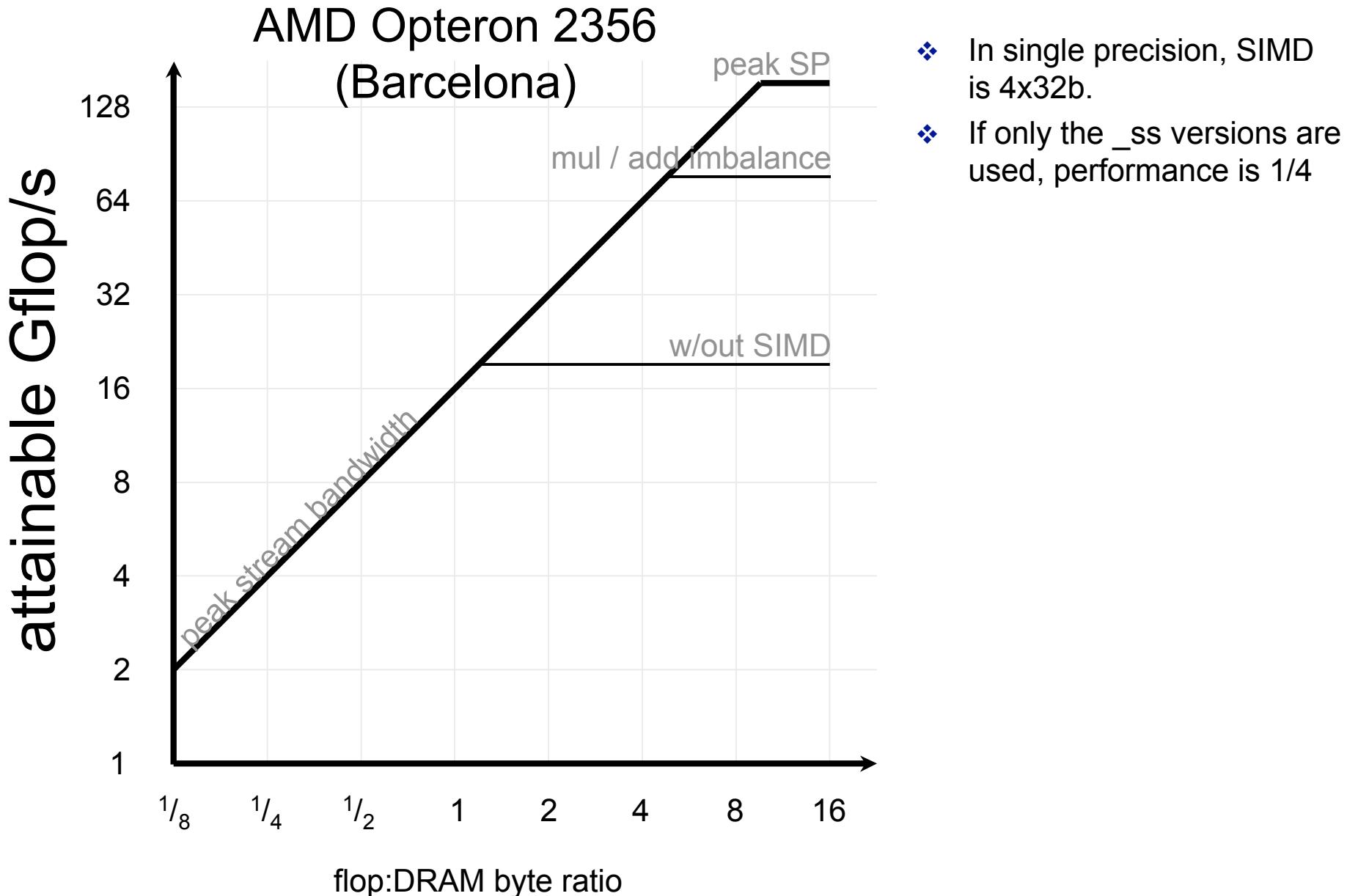
- ❖ *Bandwidth #'s collected via micro benchmarks*
- ❖ *Computation #'s derived from optimization manuals (pencil and paper)*
- ❖ *Assume complete overlap of either communication or computation*

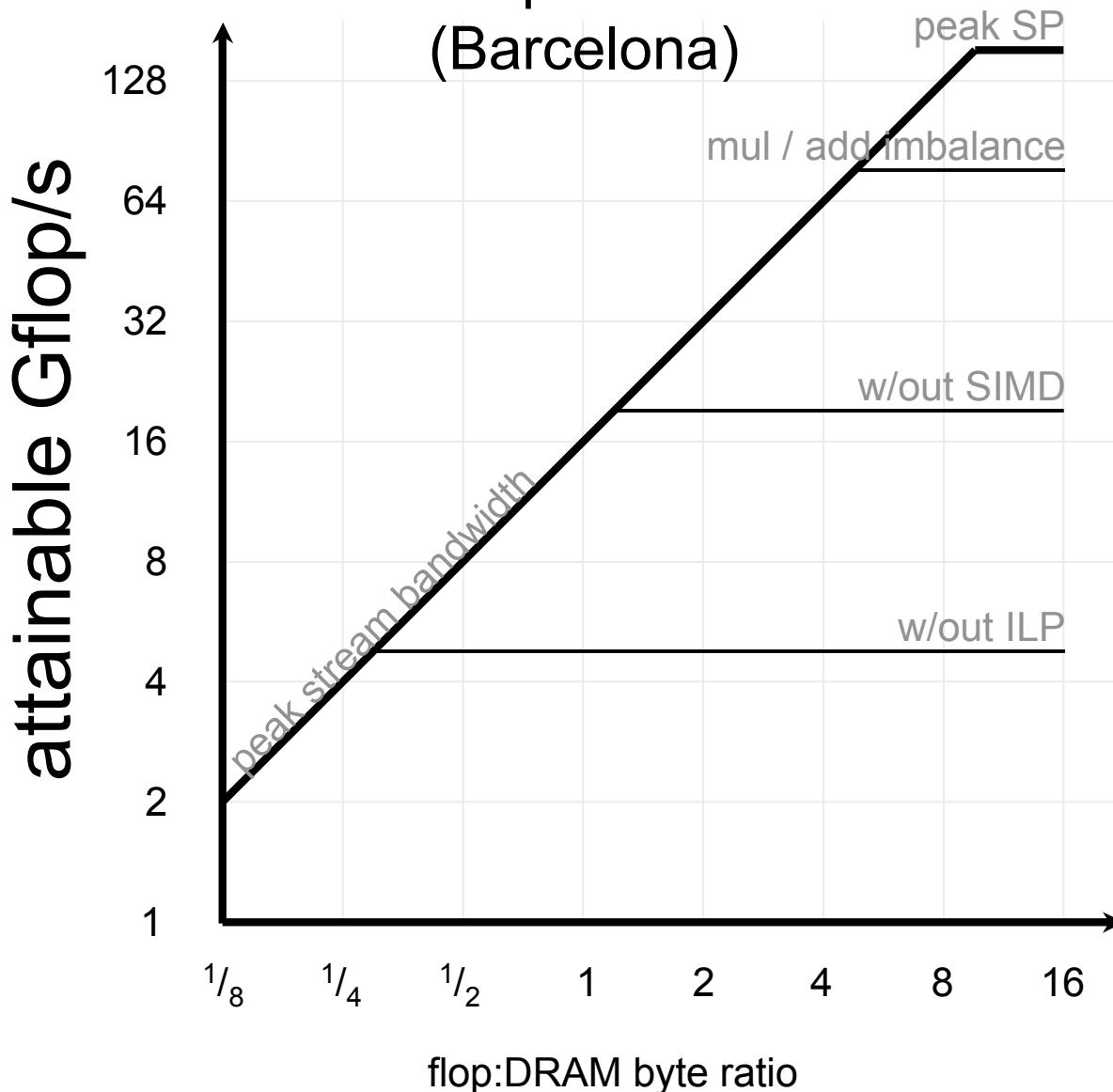




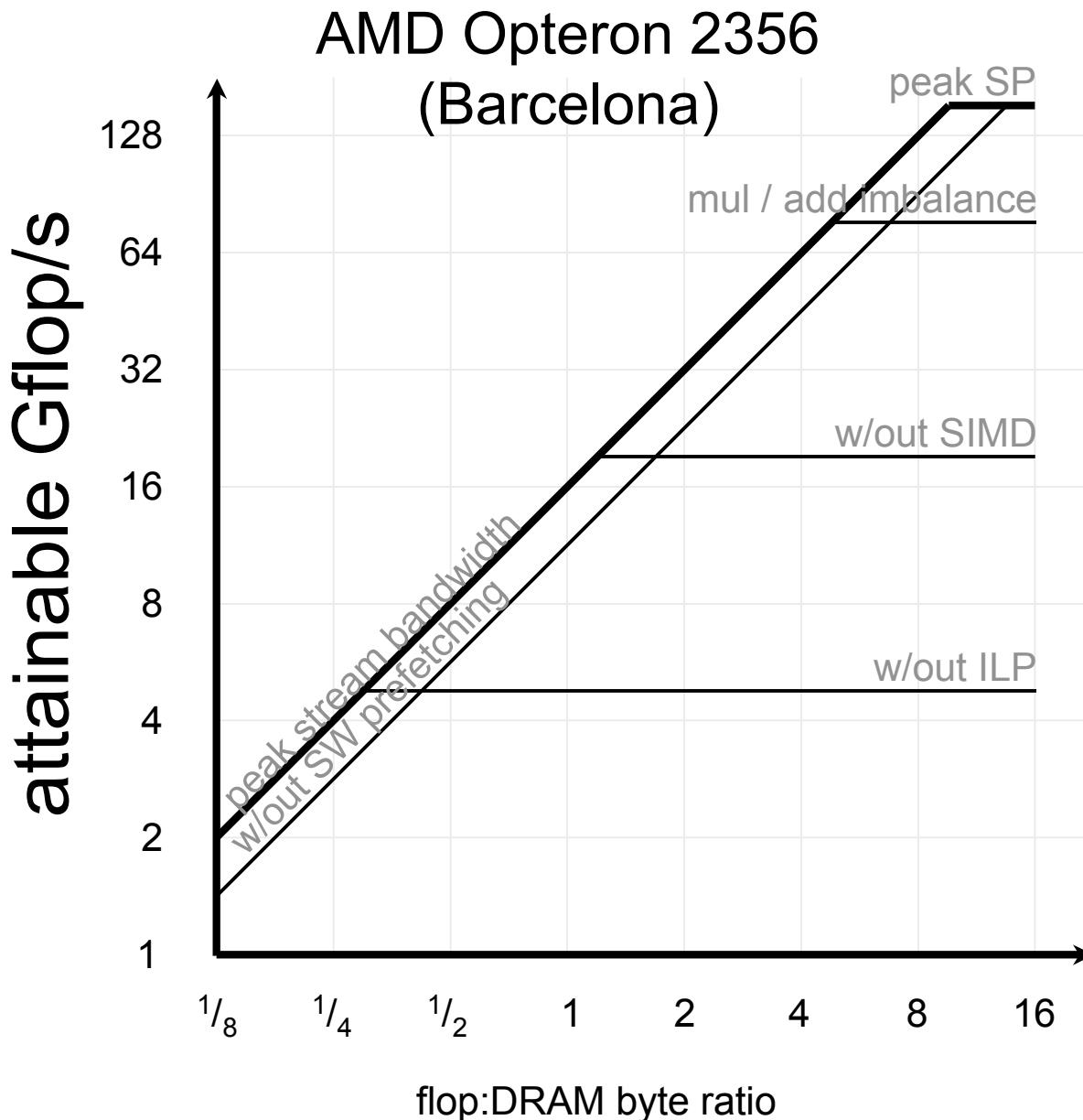
AMD Opteron 2356
(Barcelona)

- ❖ Opterons have separate multipliers and adders
- ❖ 'functional unit parallelism'
- ❖ This is a ceiling beneath the roofline

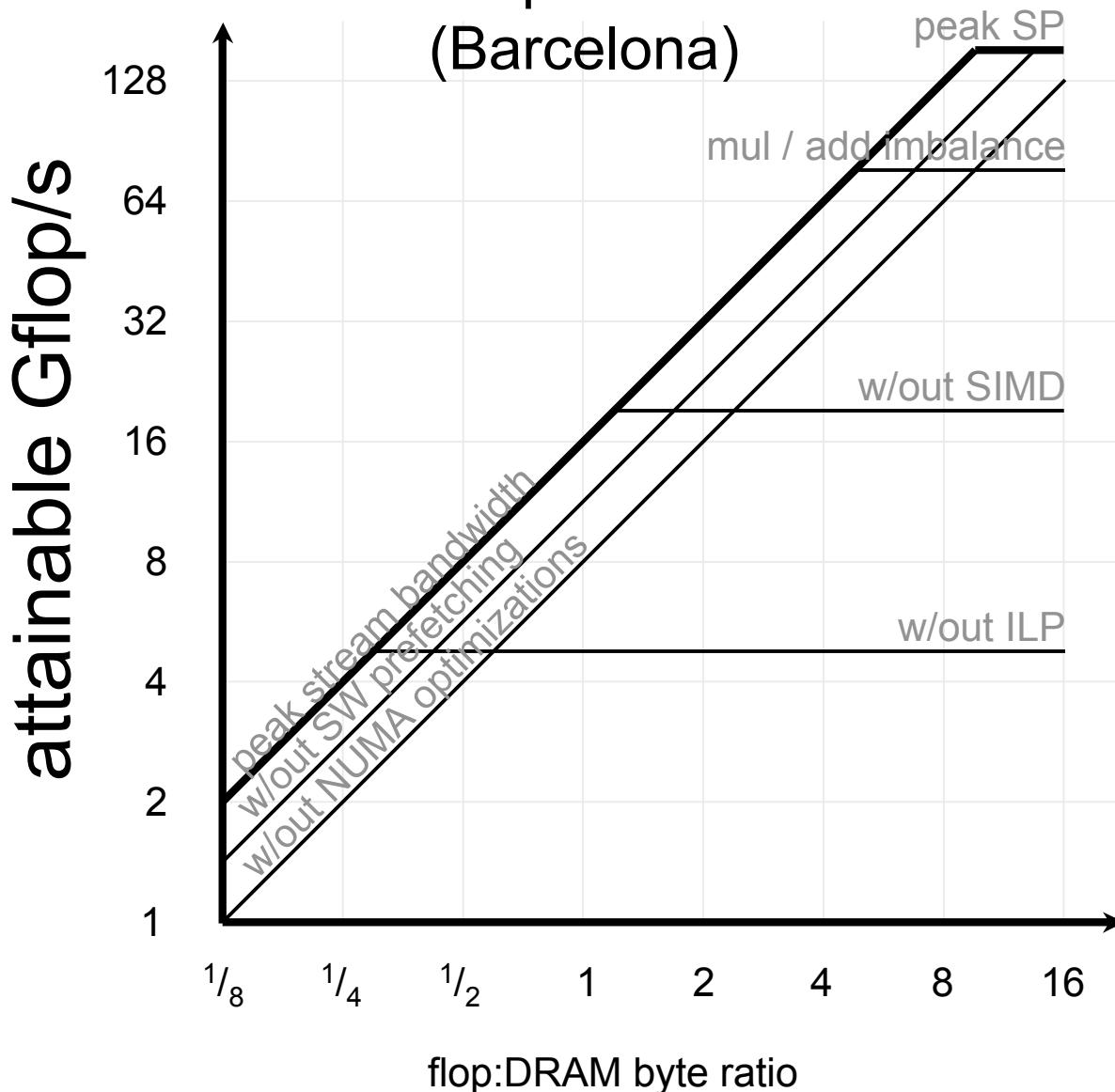


AMD Opteron 2356
(Barcelona)

- ❖ If 4 independent instructions are kept in the pipeline, performance will fall



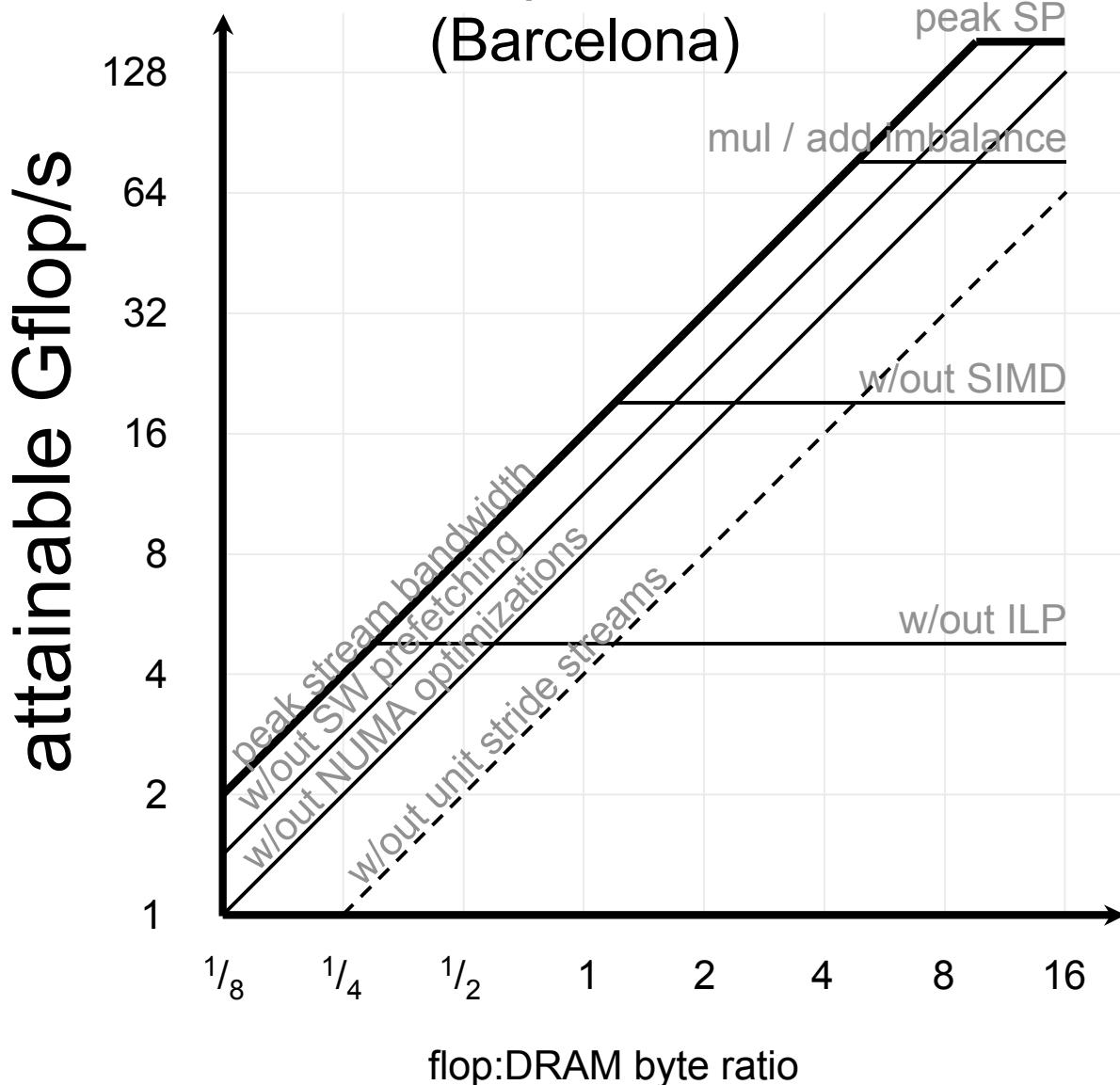
- ❖ If SW prefetching is not used, performance will degrade
- ❖ These act as ceilings below the bandwidth roofline

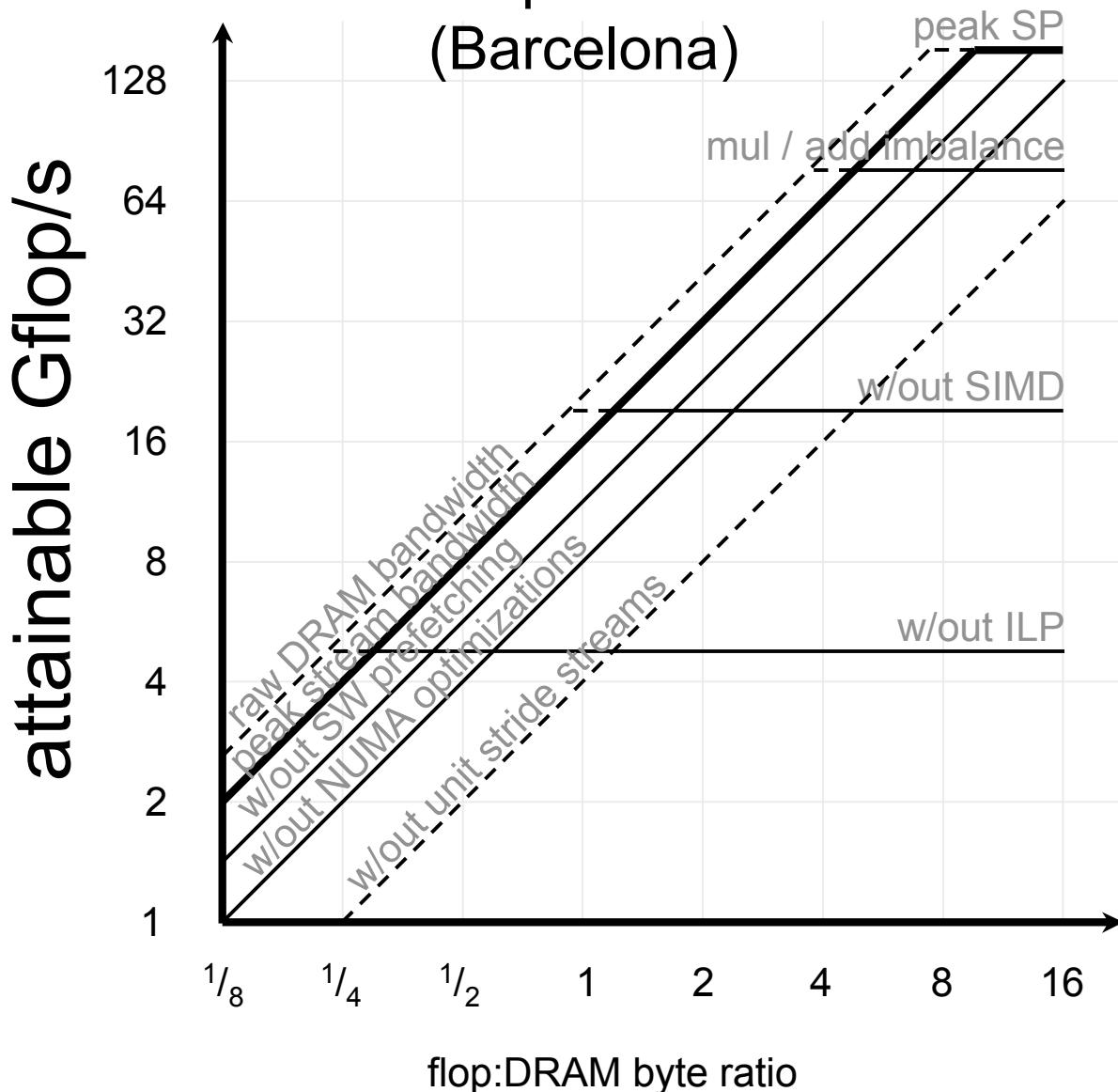
AMD Opteron 2356
(Barcelona)

- ❖ Without NUMA optimizations, the memory controllers on the second socket can't be used.

AMD Opteron 2356
(Barcelona)

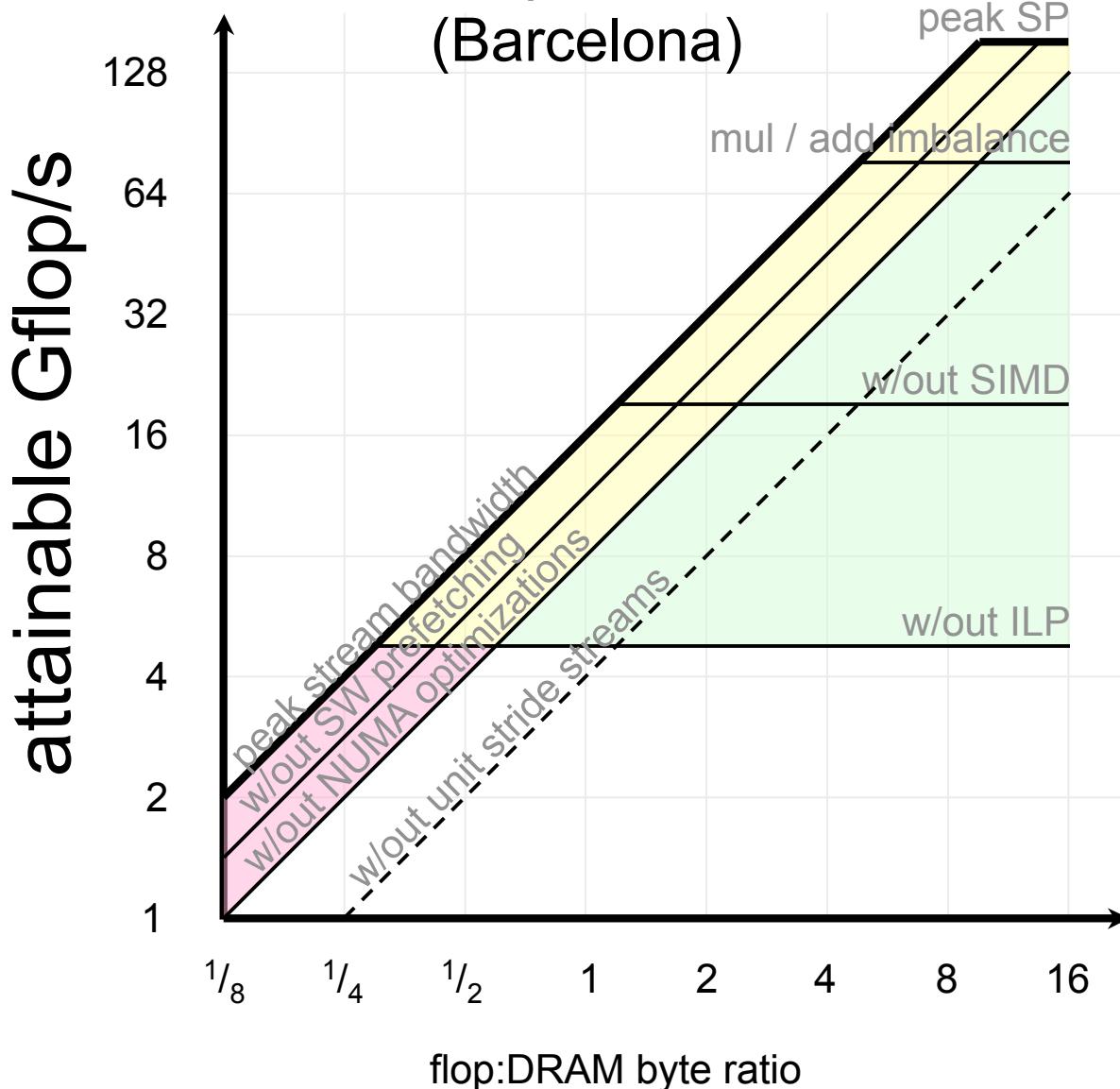
- Bandwidth is much lower without unit stride streams



AMD Opteron 2356
(Barcelona)

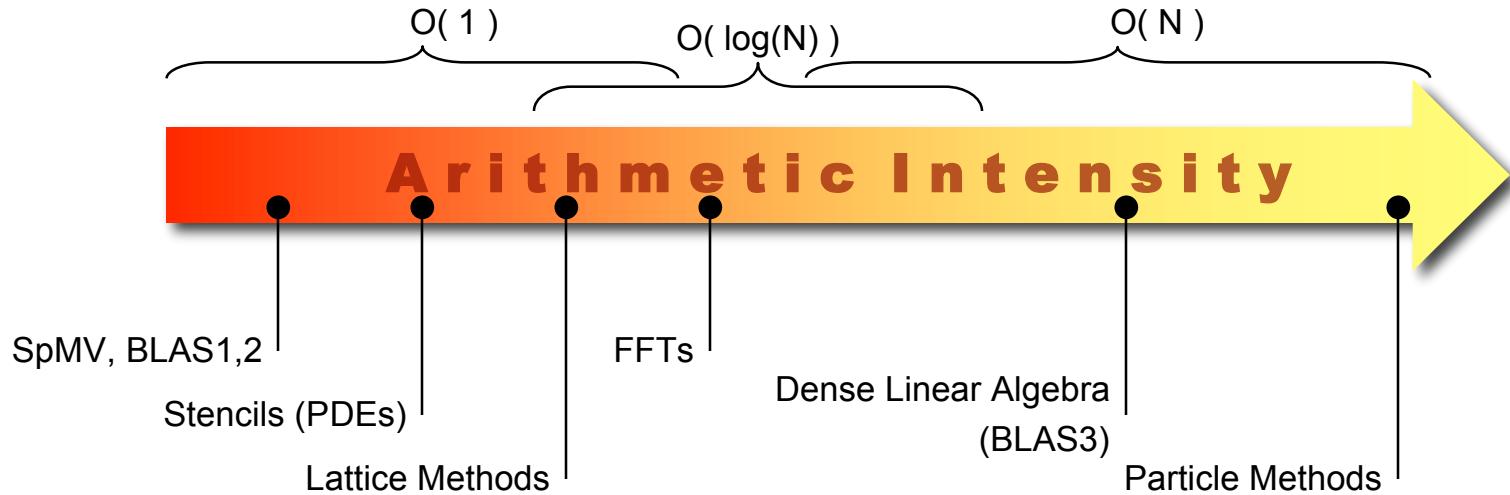
- ❖ Its difficult for any architecture to reach the raw DRAM bandwidth

AMD Opteron 2356 (Barcelona)



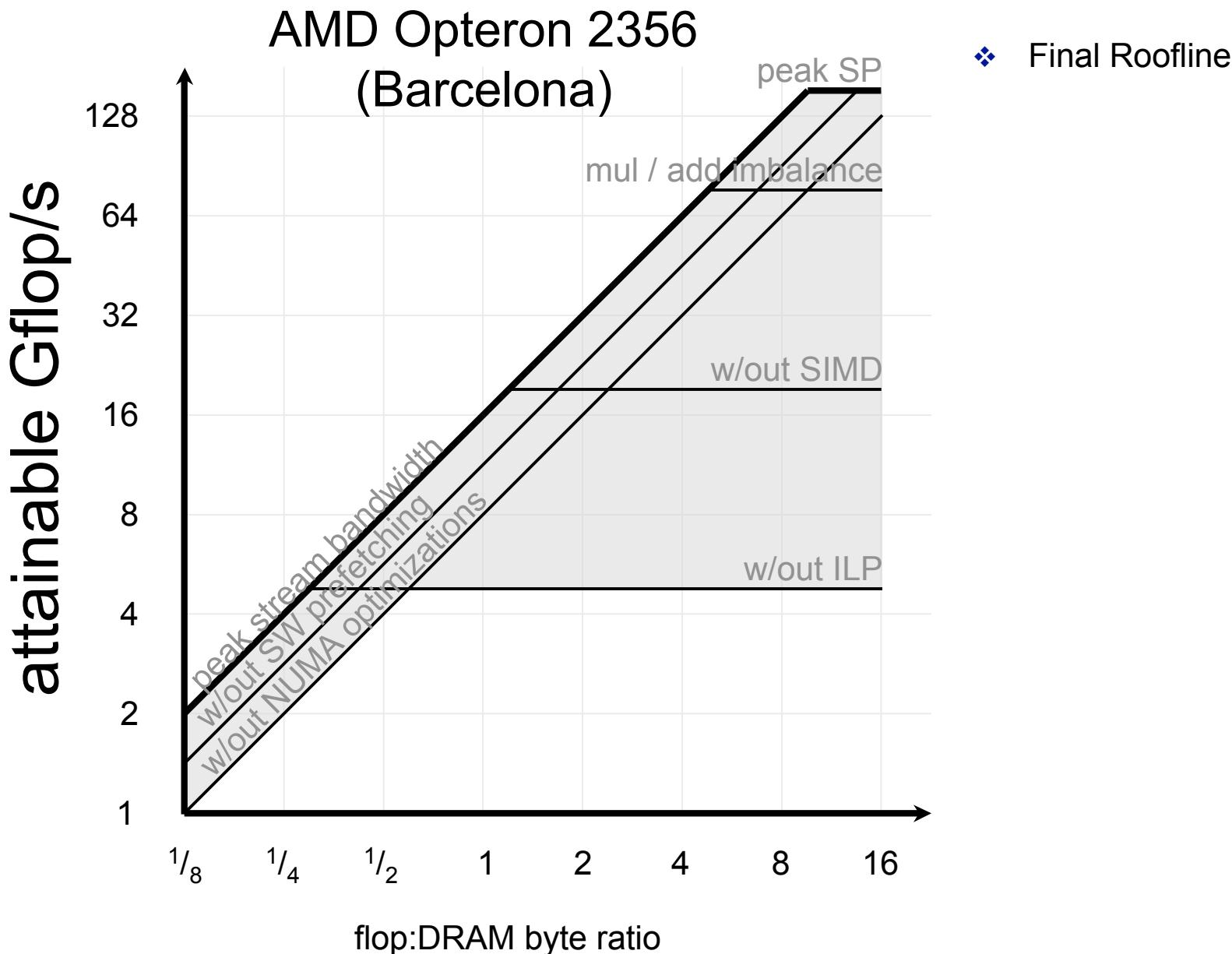
- ❖ Partitions the regions of expected performance into three optimization regions:
 - Compute only
 - Memory only
 - Compute+Memory

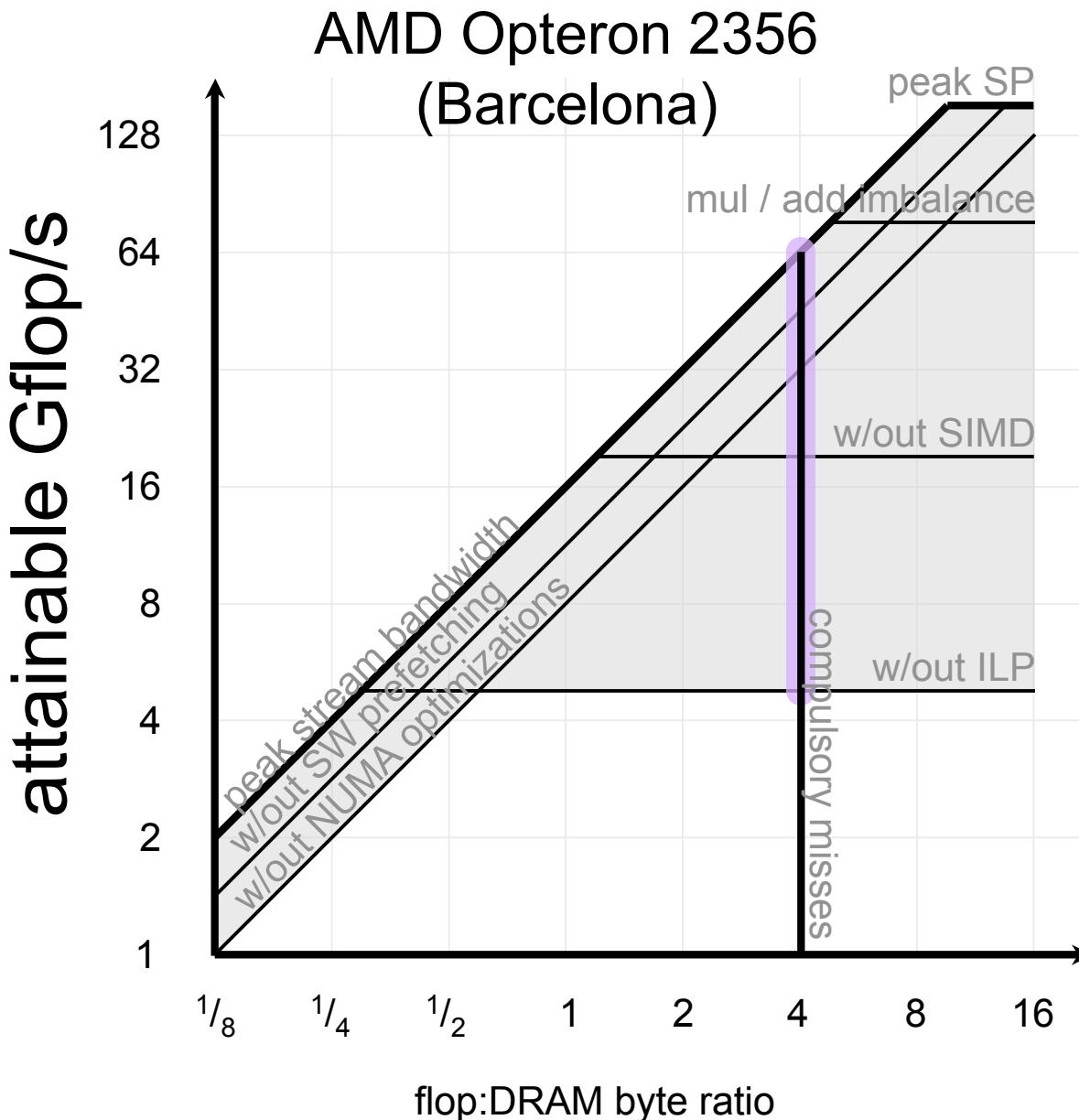
- ❖ There is no single ordering or roofline model
- ❖ The order of ceilings is generally (bottom up):
 - What is inherent in algorithm
 - What a compiler is likely to provide
 - What a programmer could provide
 - What can never be exploited for this kernel
- ❖ For example,
 - FMA or mul/add balance is inherent in many linear algebra routines and should be placed at the bottom.
 - However, many stencils are dominated by adds, and thus the multipliers and FMA go underutilized.



- ❖ **Arithmetic Intensity (AI) \sim Total Flops / Total DRAM Bytes**
- ❖ Some HPC kernels have an arithmetic intensity that's constant, but on others it scales with with problem size (increasing temporal locality)
- ❖ Actual arithmetic intensity is capped by cache/local store capacity

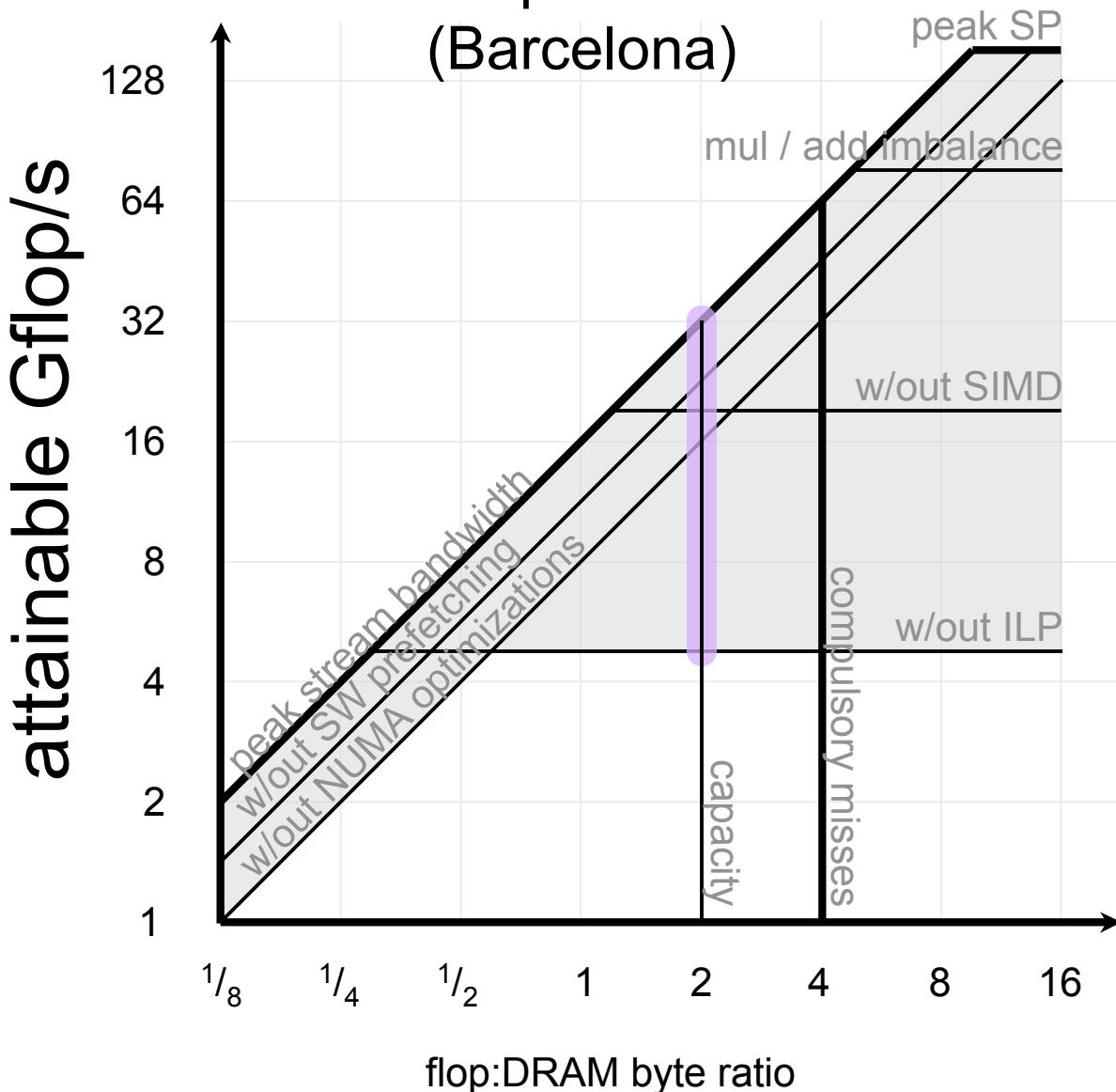
- ❖ Remember the 3C's of caches
- ❖ Calculating the Flop:DRAM byte ratio is:
 - **Compulsory misses**: straightforward
 - **Capacity misses**: pencil and paper (maybe performance counters)
 - **Conflict misses**: must use performance counters
- ❖ Flop:actual DRAM Byte ratio < Flop:compulsory DRAM Byte ratio
- ❖ One might place a range on the arithmetic intensity ratio
- ❖ Thus performance is limited to an area between the ceilings and between the upper (compulsory) and lower bounds on arithmetic intensity



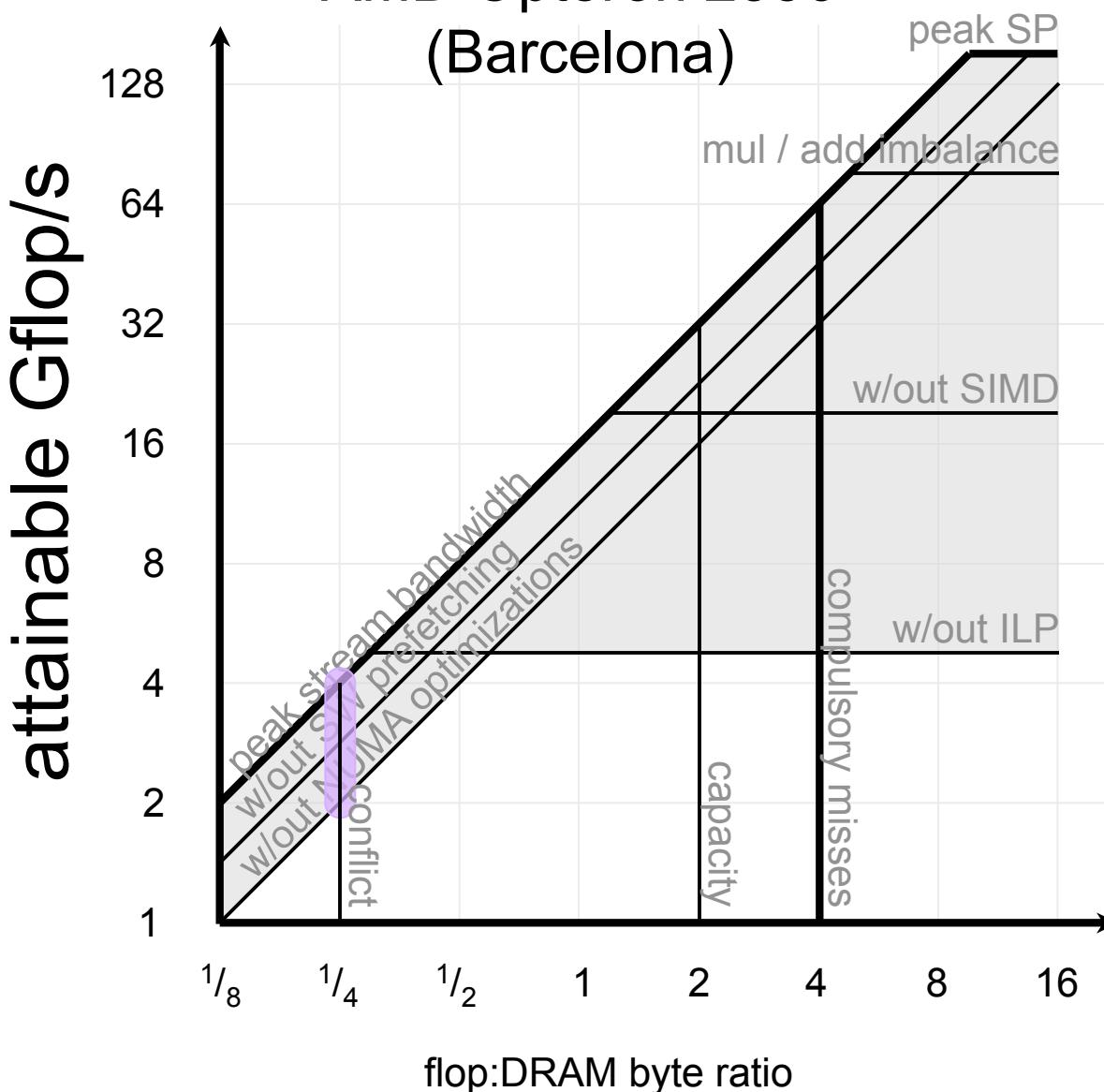


- ❖ Some arbitrary kernel has a flop:compulsory byte ratio of 4
- ❖ Overlaid on the roofline
- ❖ Defines upper bound on range of expected performance
- ❖ Also shows which optimizations are likely

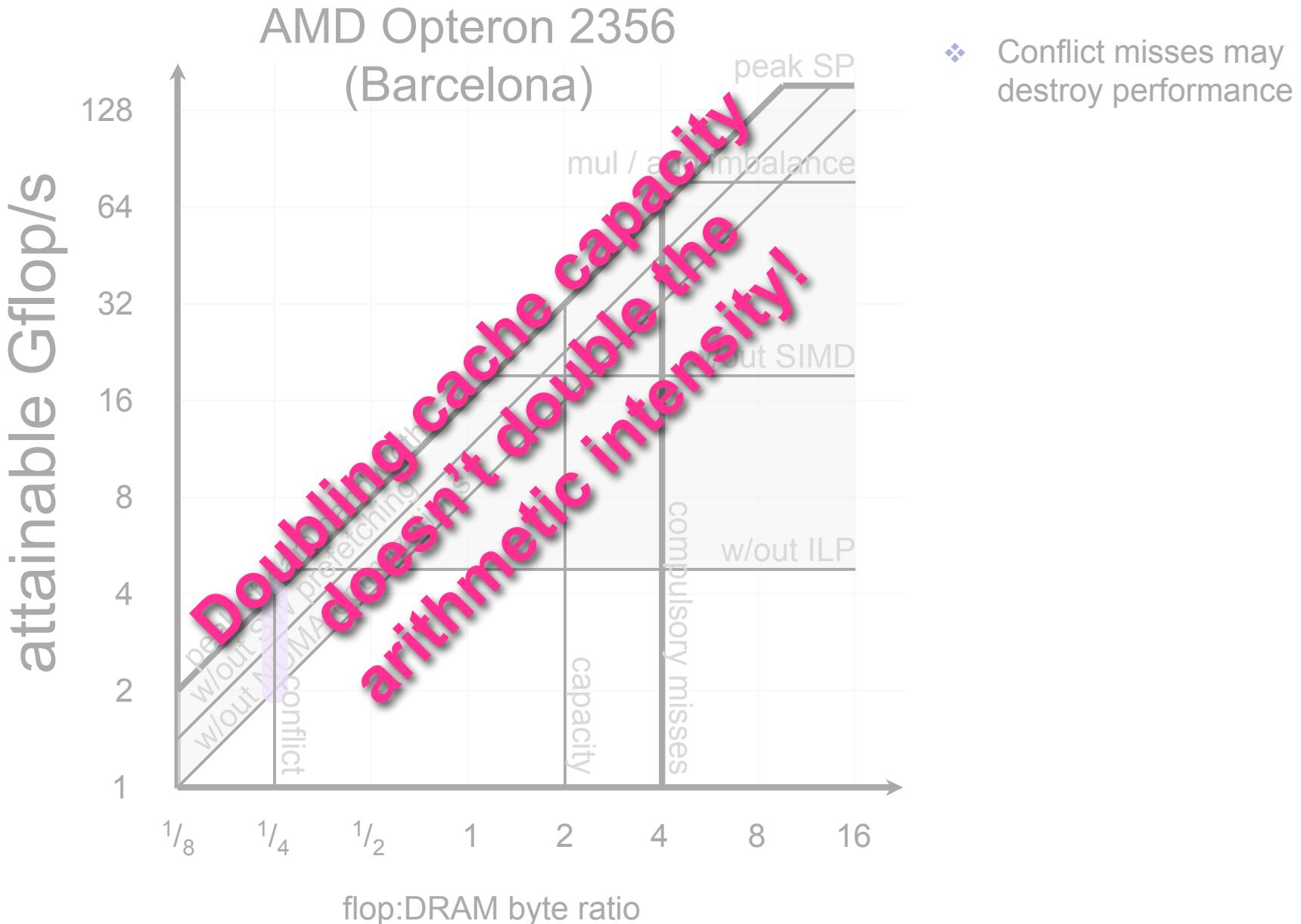
AMD Opteron 2356 (Barcelona)



- ❖ Capacity misses reduce the actual flop:byte ratio
- ❖ Also reduces attainable performance
- ❖ **AI is unique to each combination of kernel and architecture**

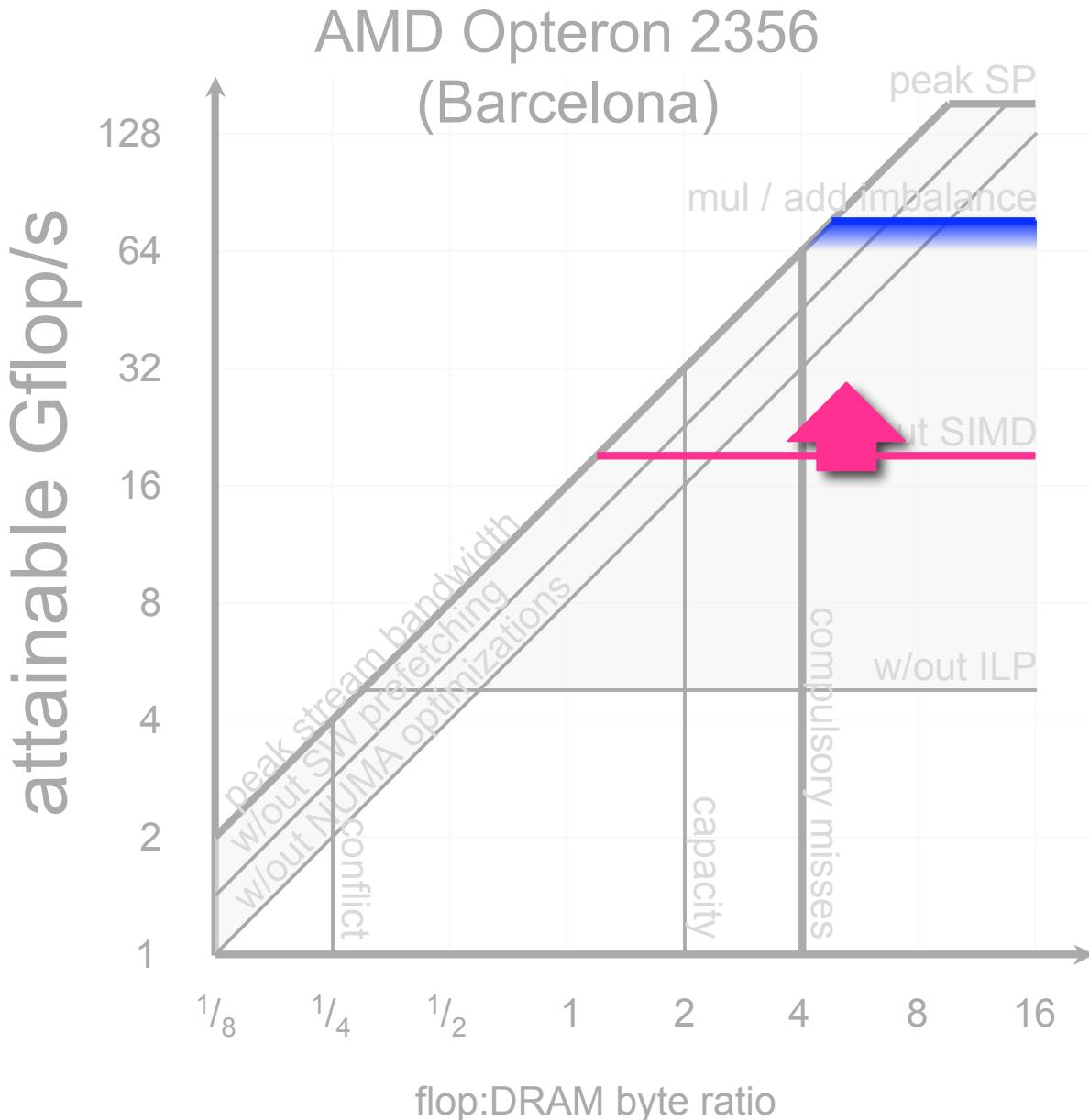
AMD Opteron 2356
(Barcelona)

- ❖ Conflict misses may destroy performance
- ❖ **AI is unique to each combination of kernel and architecture**



Three Categories of Software Optimization

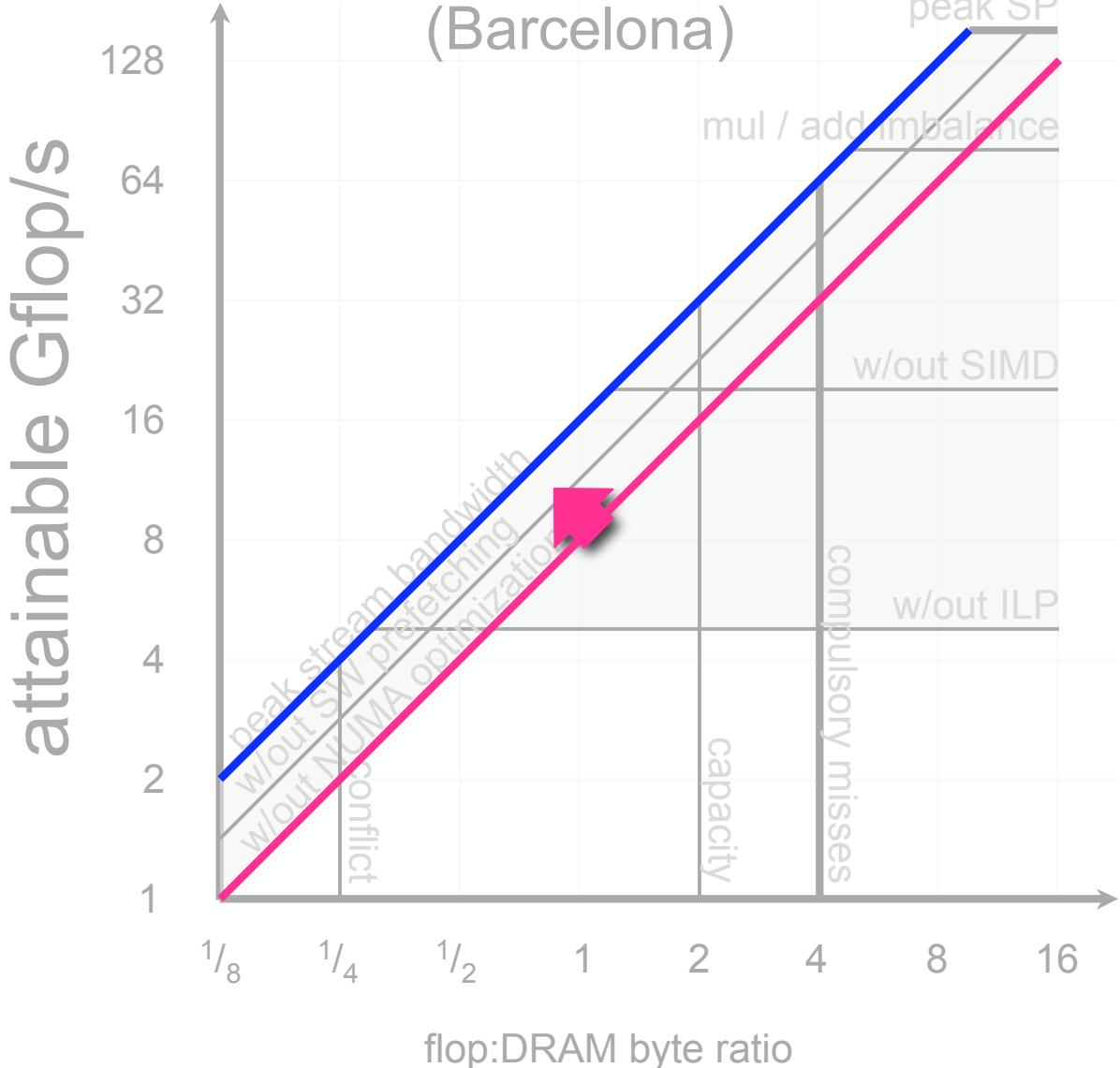
Maximizing Attained in-core Performance



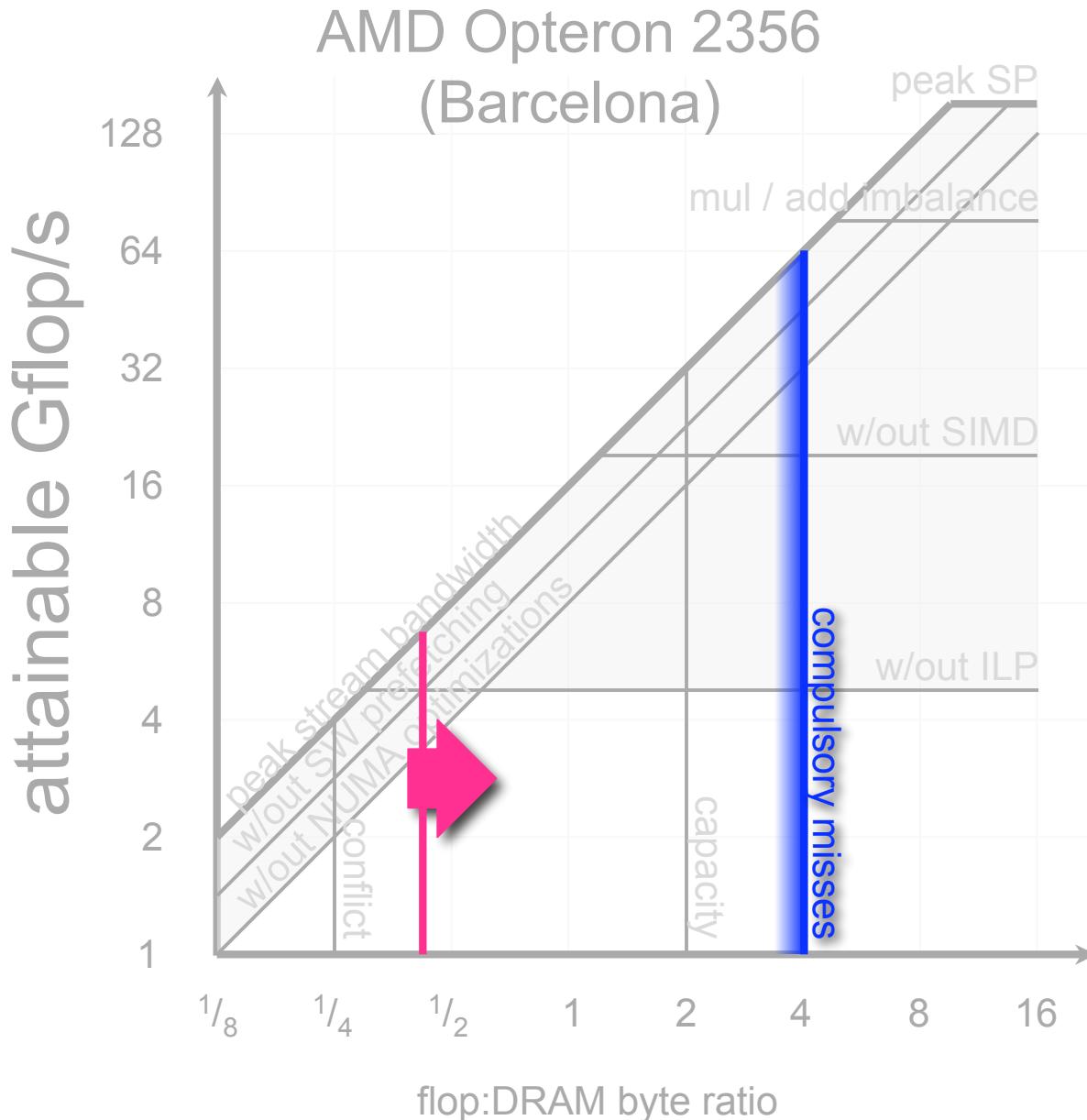
- ❖ Software optimizations such as explicit SIMDization can punch through the horizontal ceilings (what can be expected from a compiler)
- ❖ Other examples include loop unrolling, reordering, and long running loops

Maximizing Attained Memory Bandwidth

AMD Opteron 2356
(Barcelona)

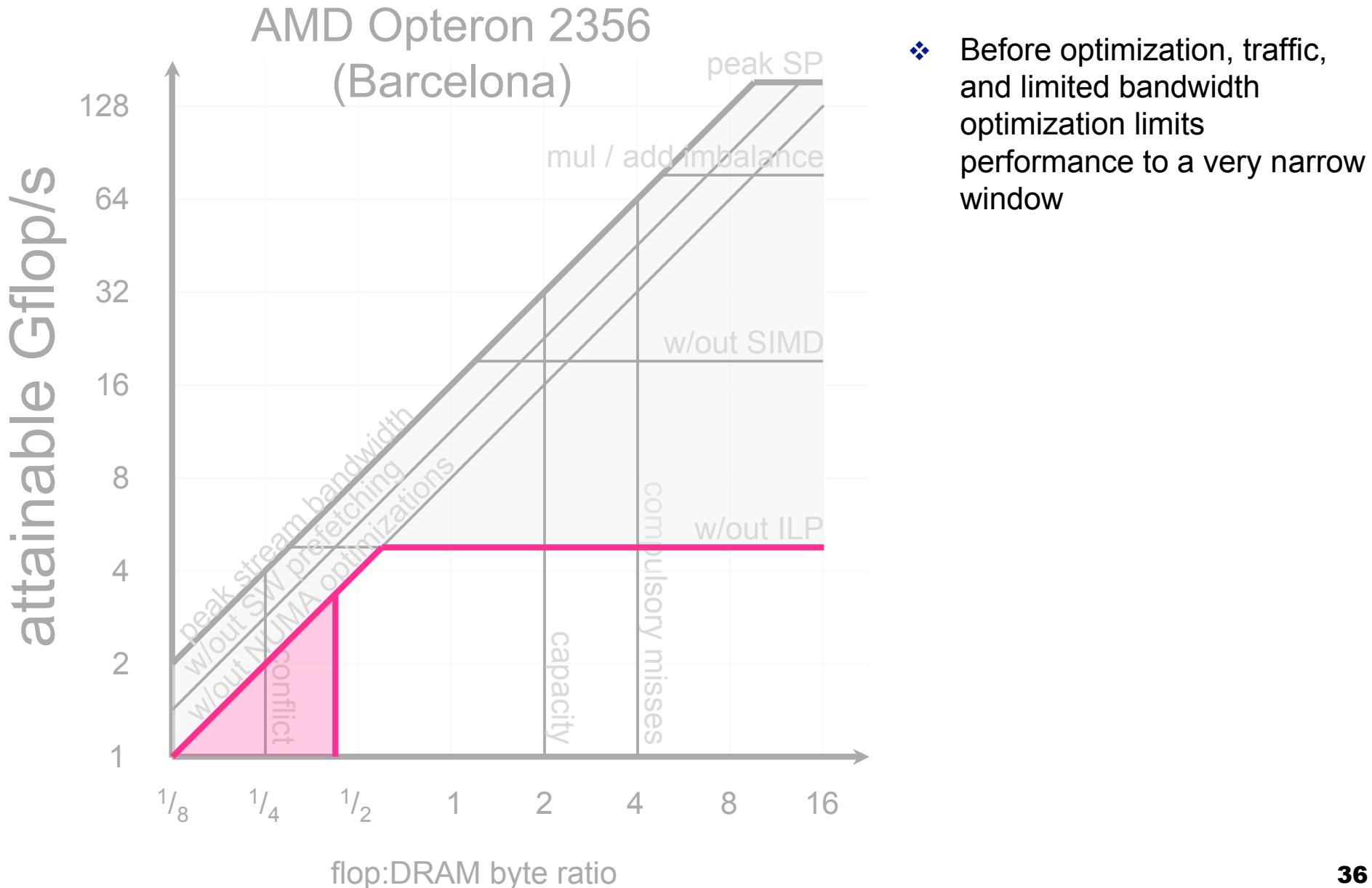


- ❖ Compilers won't give great out-of-the box bandwidth
- ❖ Punch through bandwidth ceilings:
 - Maximize MLP
 - long unit stride accesses
 - NUMA aware allocation and parallelization
 - SW prefetching

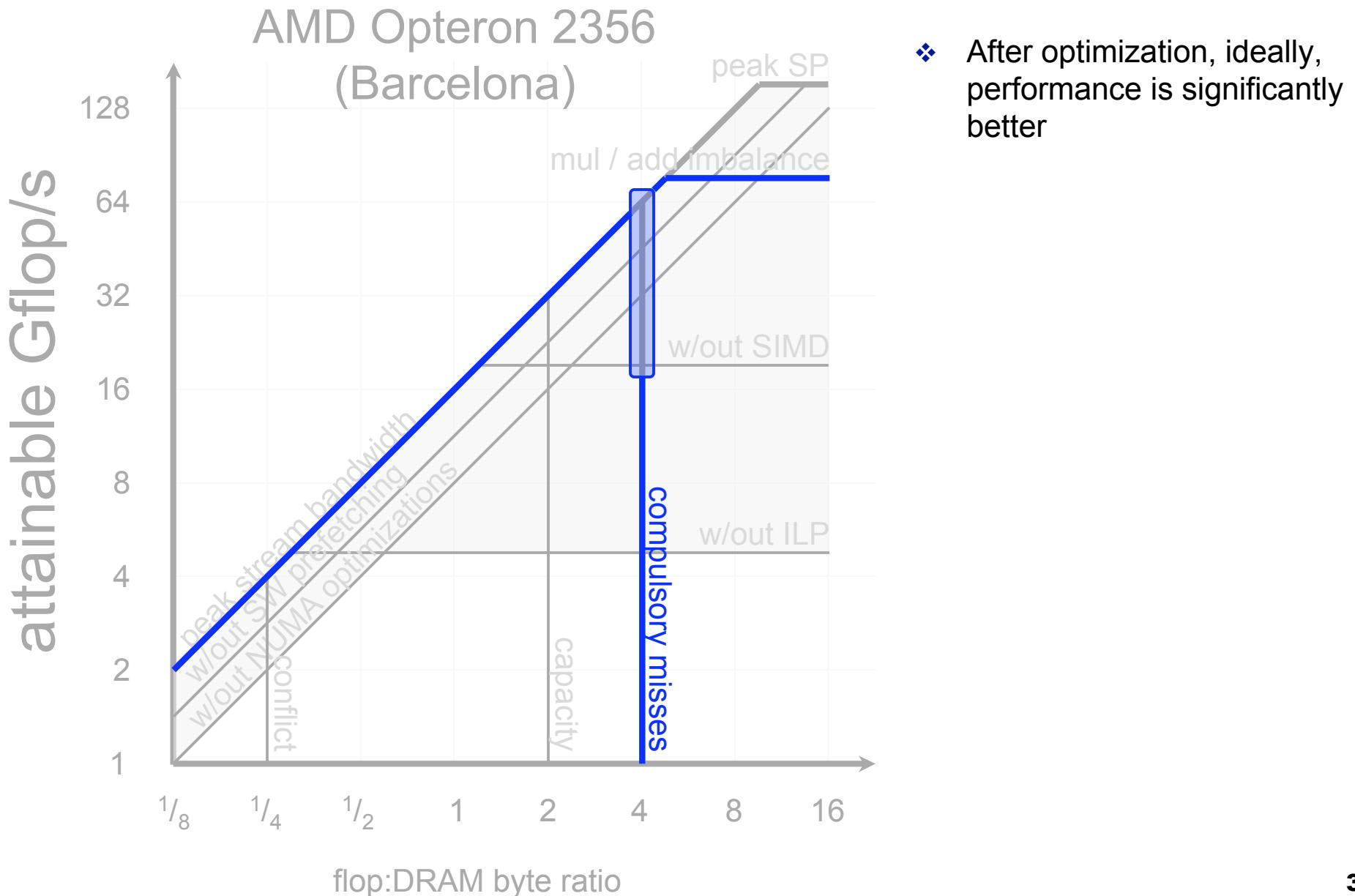


- ❖ Use performance counters to measure flop:byte ratio (AI)
- ❖ Out-of-the-box code may have an AI ratio much less than the compulsory ratio
 - Be cognizant of cache capacities, associativities, and threads sharing it
 - Pad structures to avoid conflict misses
 - Use cache blocking to avoid capacity misses
- ❖ **These optimizations can be imperative**

Effective Roofline (before)



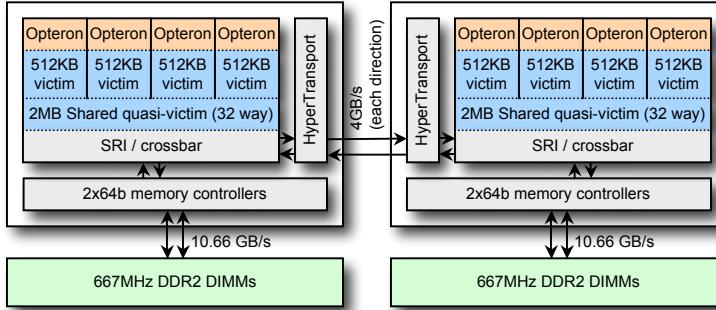
Effective Roofline (after)



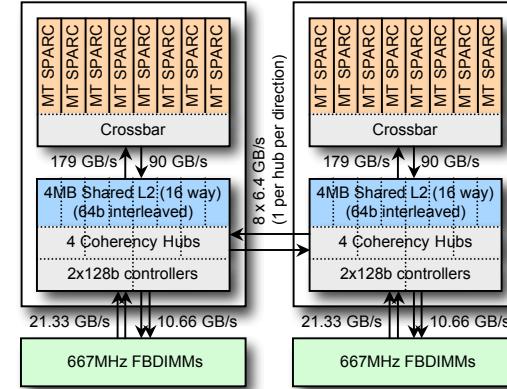
Applicable to Other Architectural Paradigms ?

Four Architectures

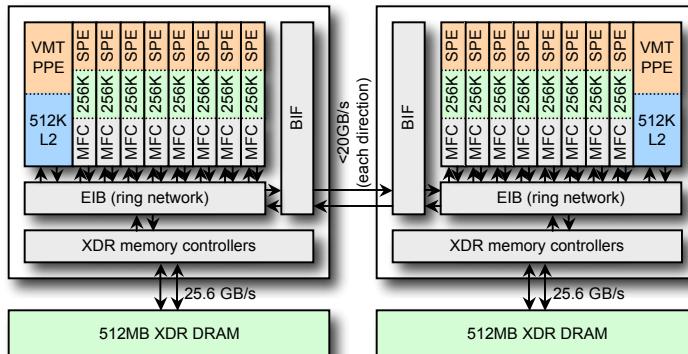
AMD Barcelona



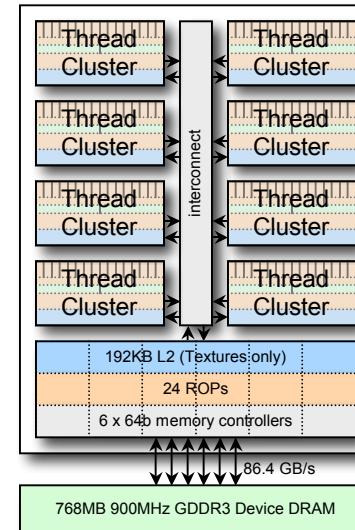
Sun Victoria Falls



IBM Cell Blade

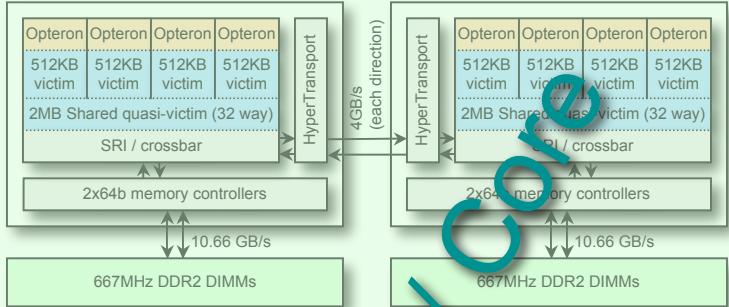


NVIDIA G80

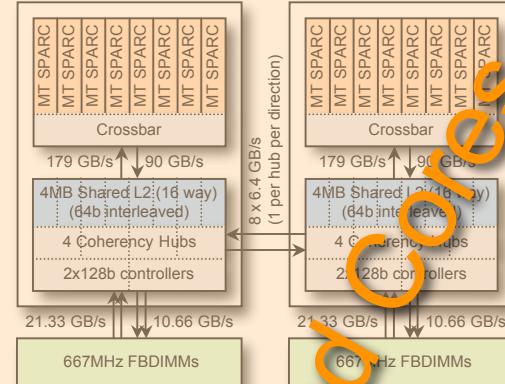


Four Architectures

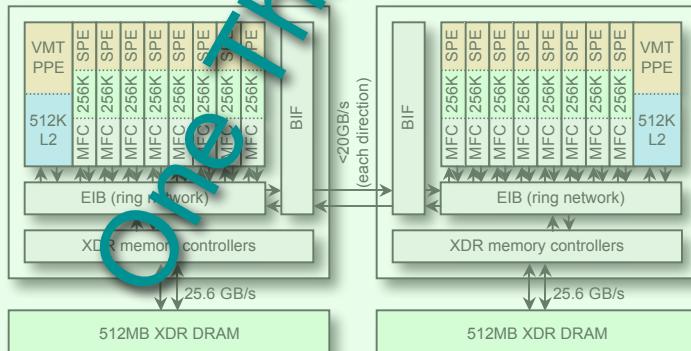
AMD Barcelona



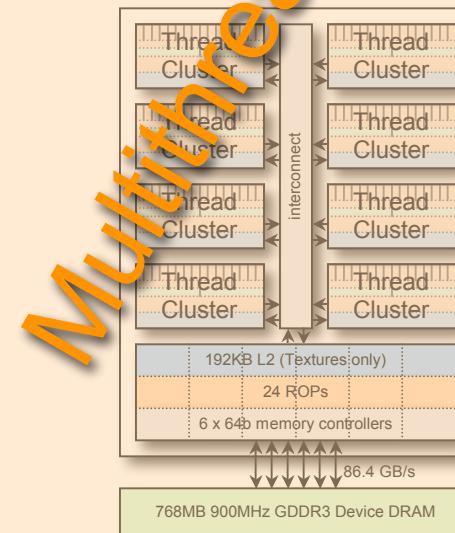
Sun Victoria Falls



IBM Cell Blade



NVIDIA G80

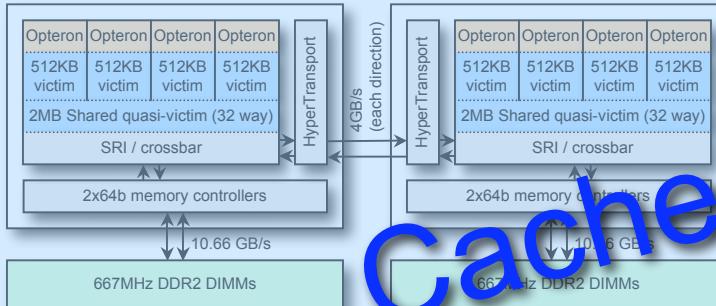


One thread per core

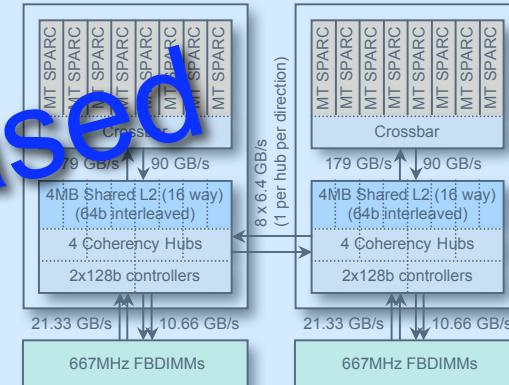
Multiple threads per core

Four Architectures

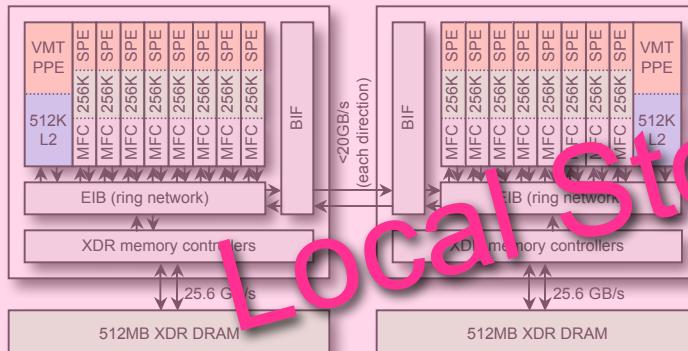
AMD Barcelona



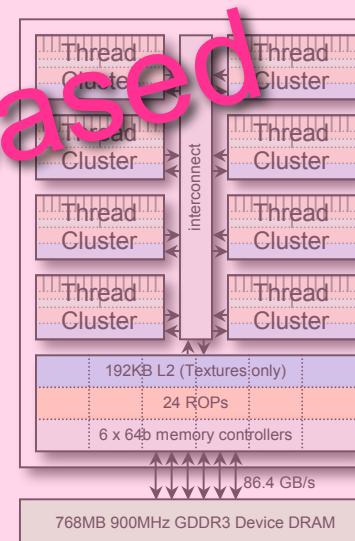
Sun Victoria Falls

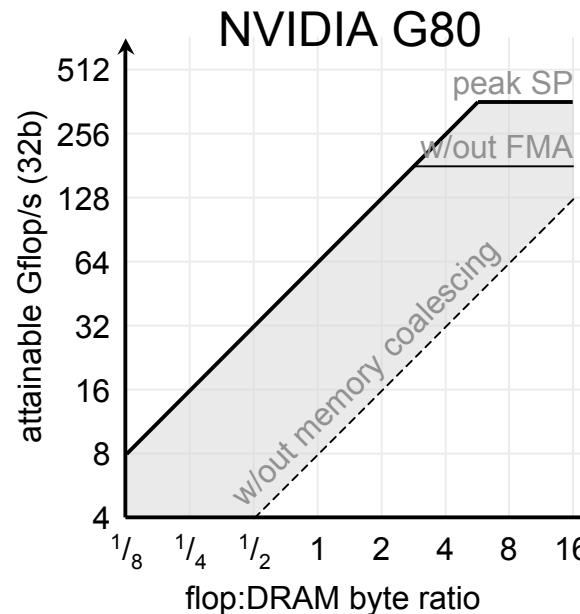
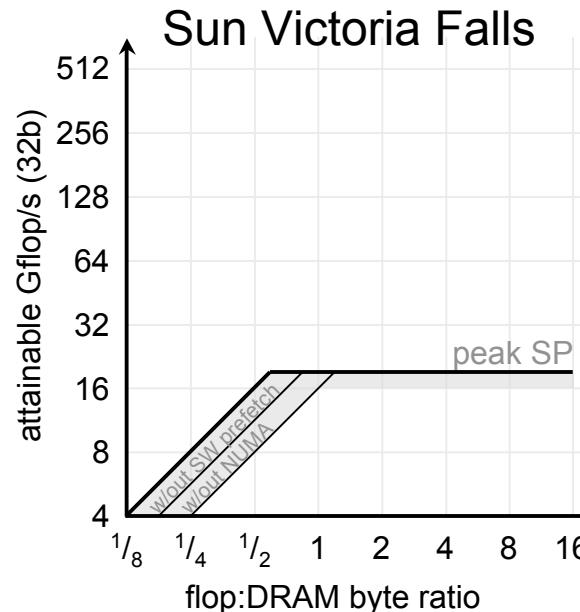
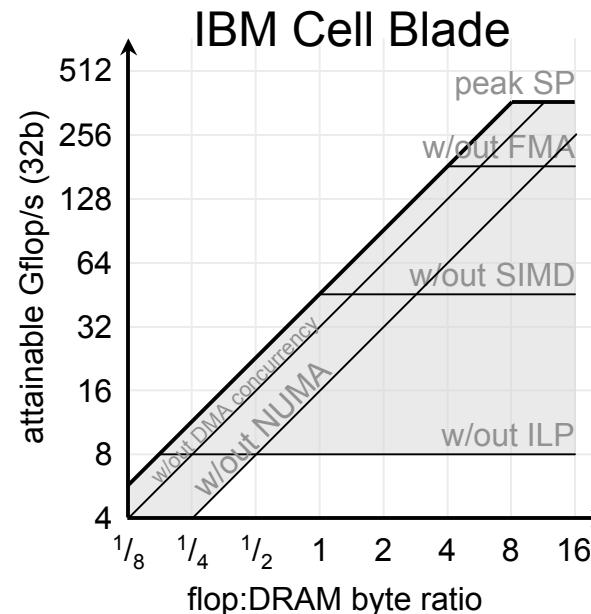
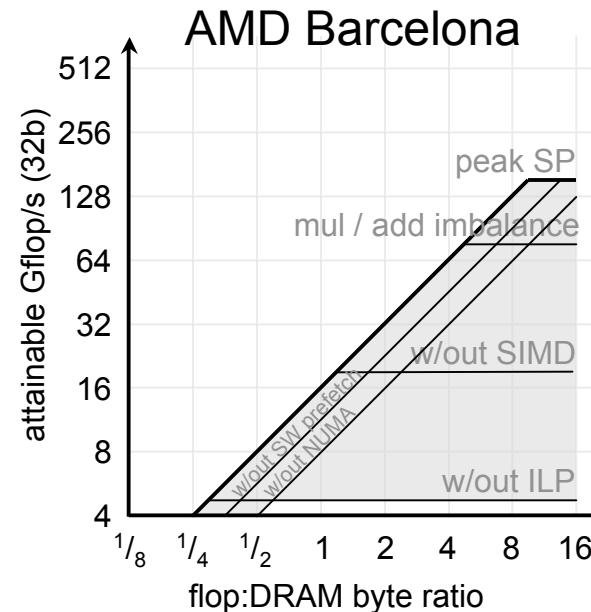


IBM Cell Blade

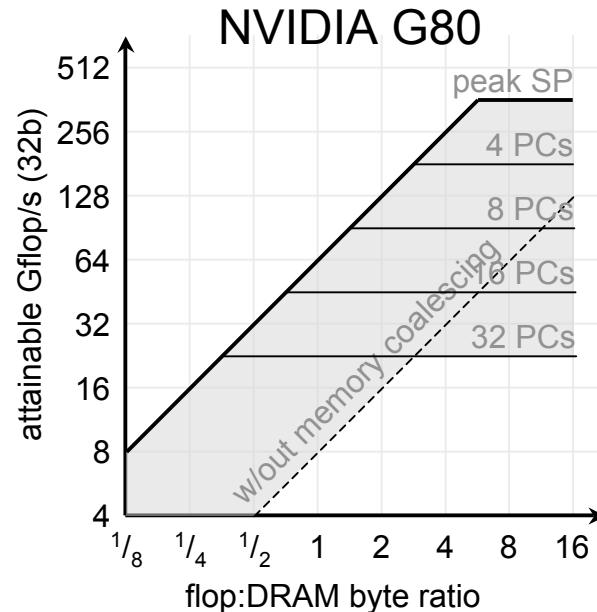
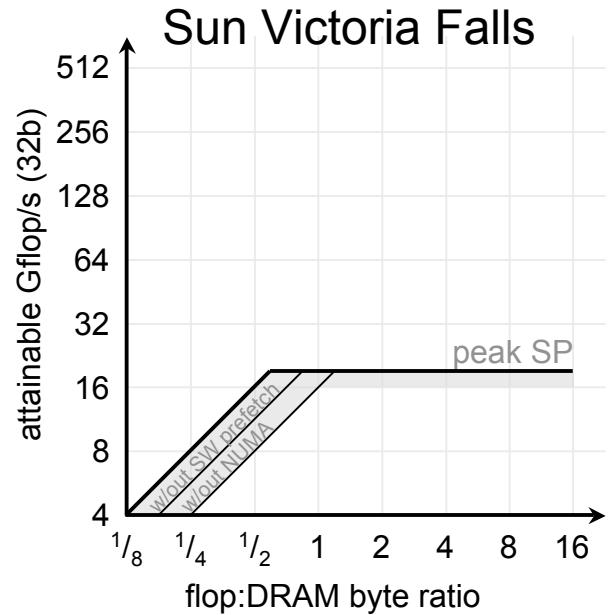
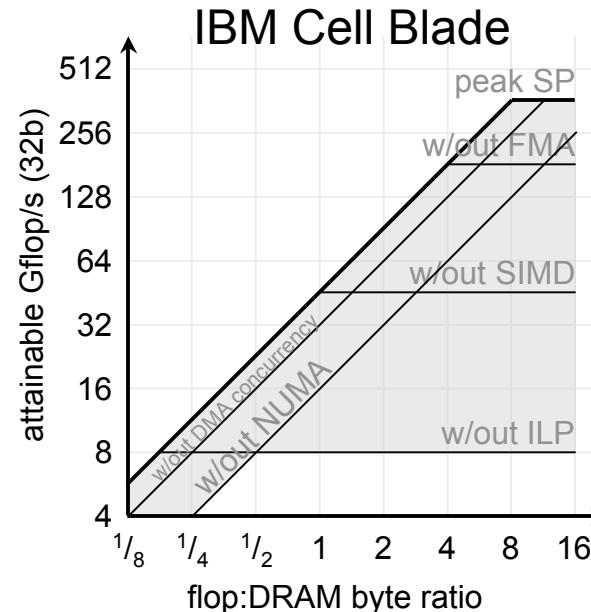
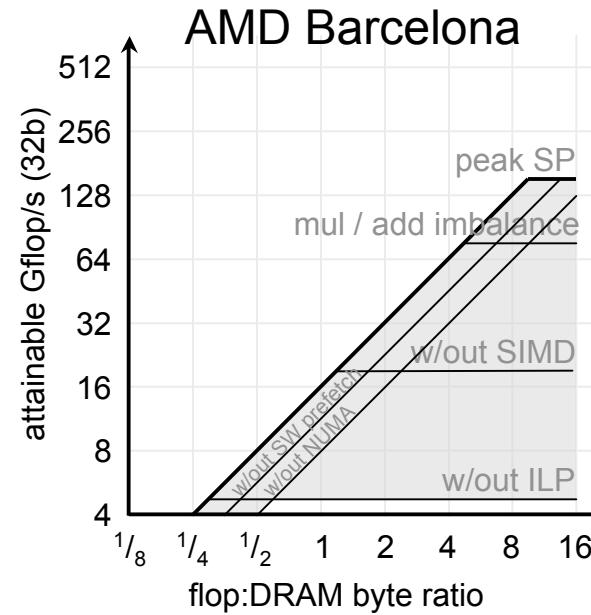


NVIDIA G80



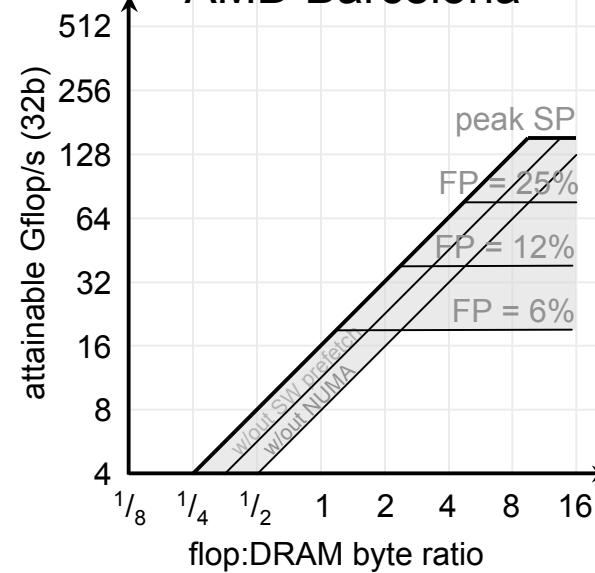


- ❖ Single Precision Roofline models for the SMPs used in this work.
- ❖ Based on micro-benchmarks, experience, and manuals
- ❖ Ceilings = in-core parallelism
- ❖ **Can the compiler find all this parallelism ?**
- ❖ NOTE:
 - log-log scale
 - Assumes perfect SPMD

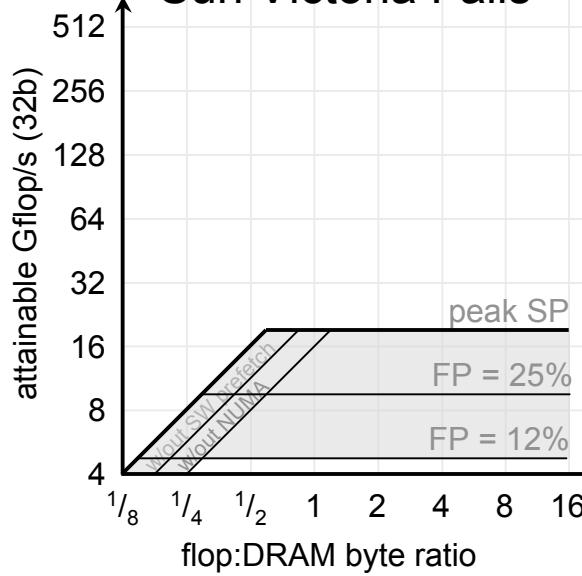


- ❖ G80 dynamically finds DLP (shared instruction fetch)
- ❖ **SIMT**
- ❖ If threads of a warp diverge from SIMD execution, performance is limited by instruction issue bandwidth
- ❖ Ceilings on G80 = number of unique PCs when threads diverge

AMD Barcelona

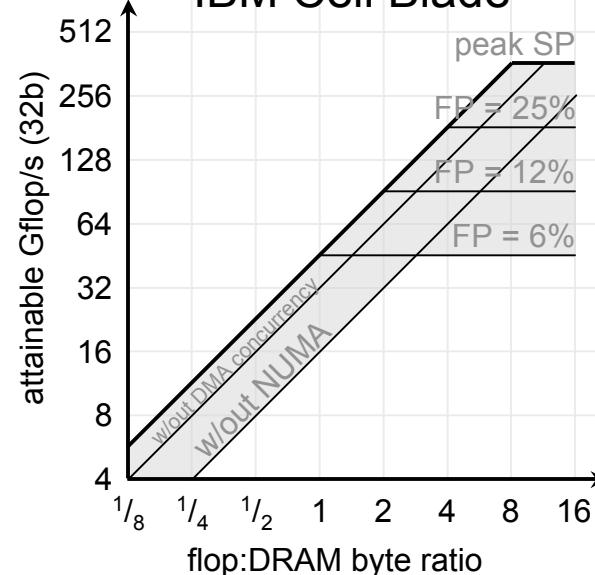


Sun Victoria Falls

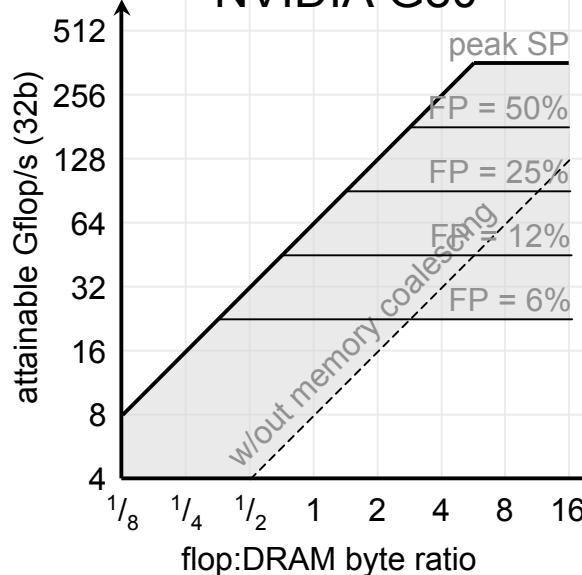


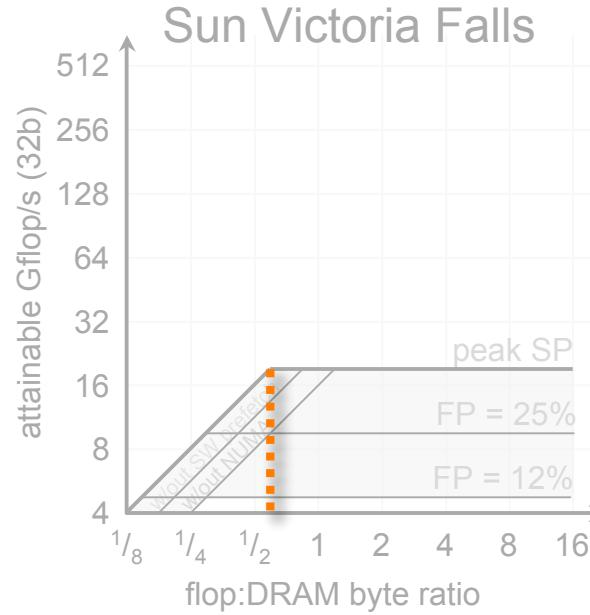
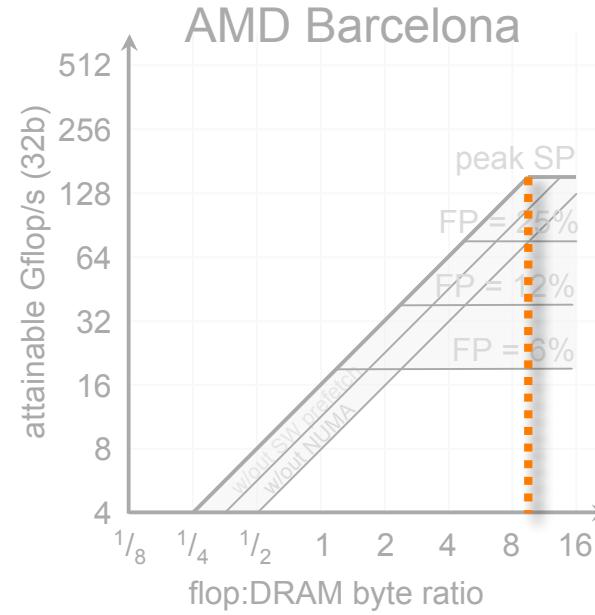
- ❖ Some kernels have large numbers of non FP instructions
- ❖ Saps instruction issue bandwidth
- ❖ Ceilings = FP fraction of dynamic instruction mix
- ❖ NOTE:
 - Assumes perfect in-core parallelism

IBM Cell Blade

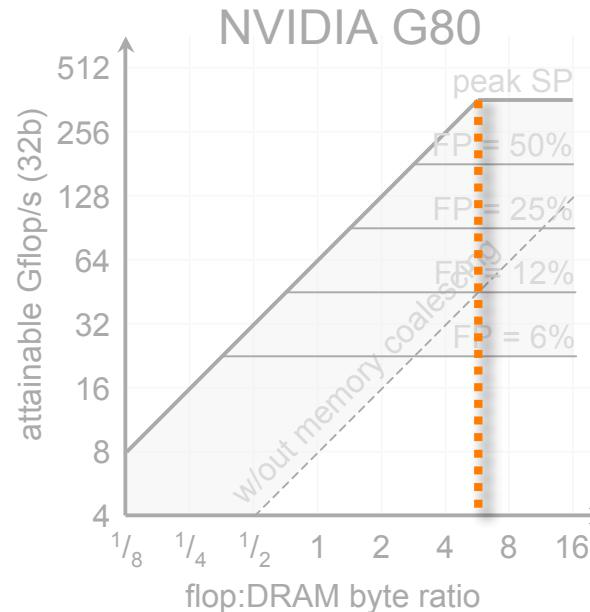
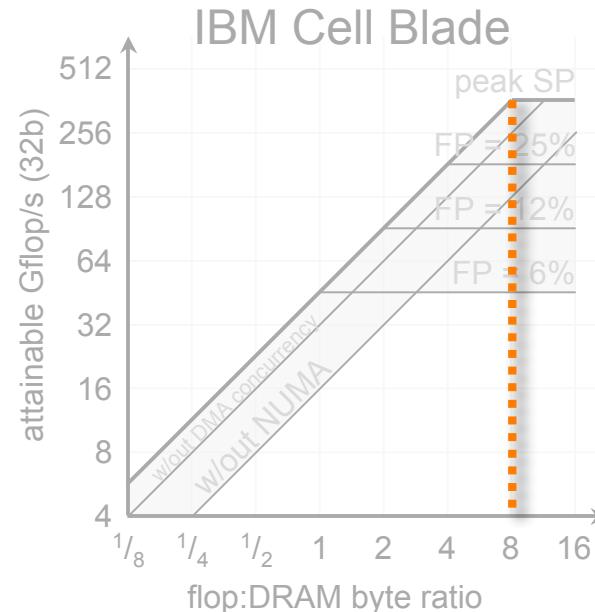


NVIDIA G80





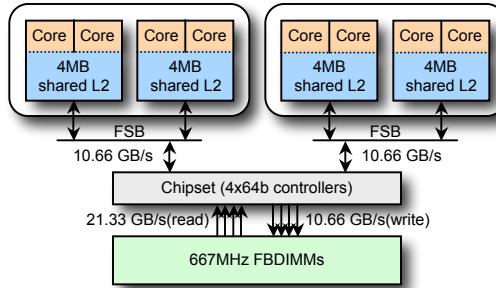
- ❖ Some architectures have drastically different ridge points
- ❖ VF may be compute bound on many kernels
- ❖ Clovertown has $\frac{1}{3}$ the BW of Barcelona = ridge point to the right



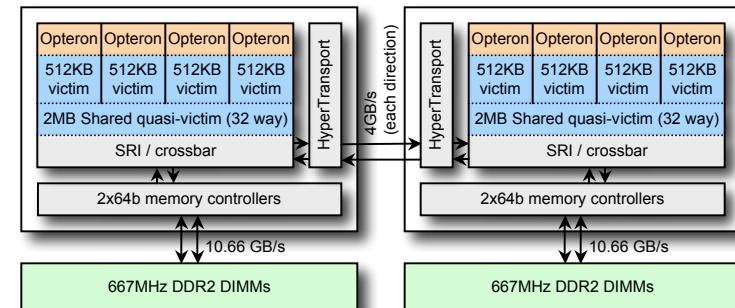
Using Roofline when Auto-tuning HPC Kernels

Multicore SMPs Used

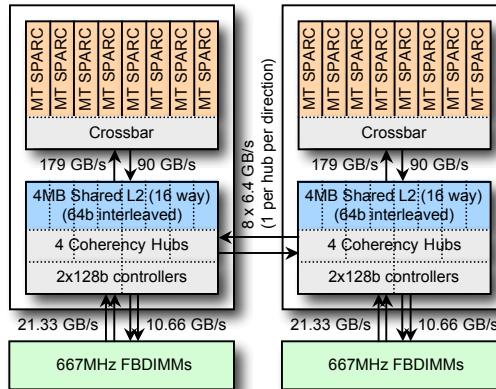
Intel Xeon E5345 (Clovertown)



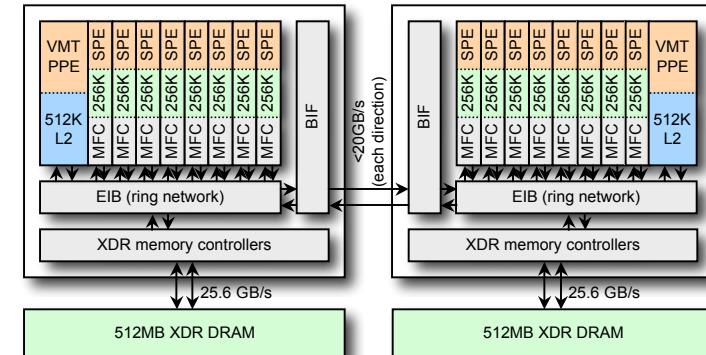
AMD Opteron 2356 (Barcelona)



Sun T2+ T5140 (Victoria Falls)



IBM QS20 Cell Blade



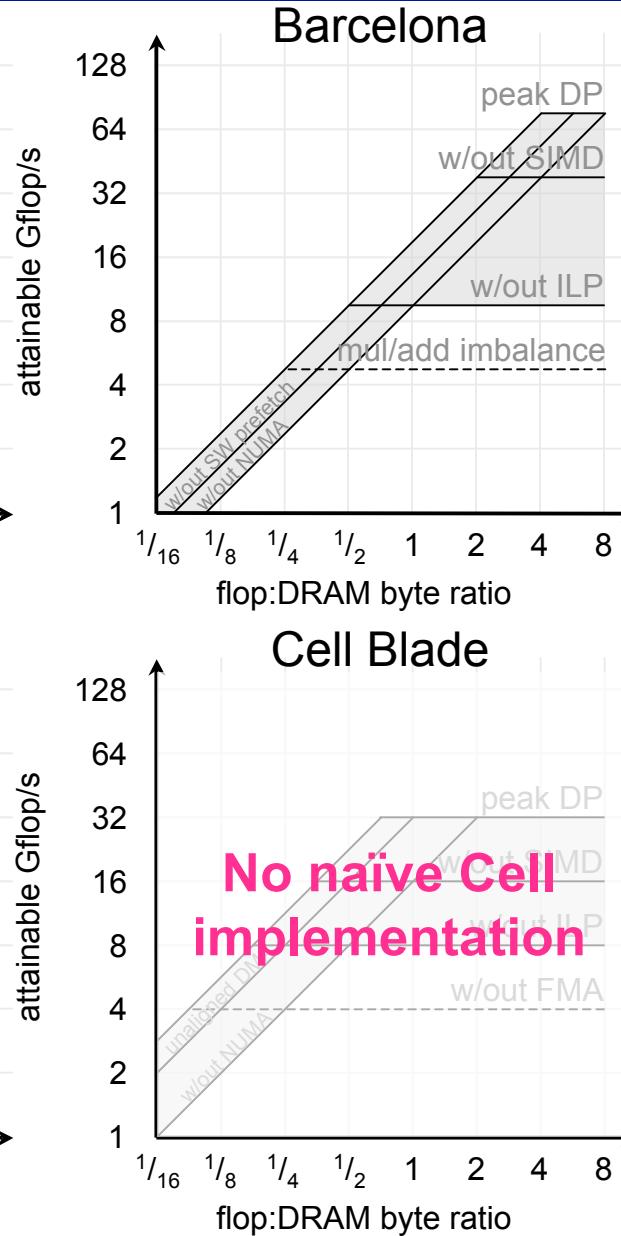
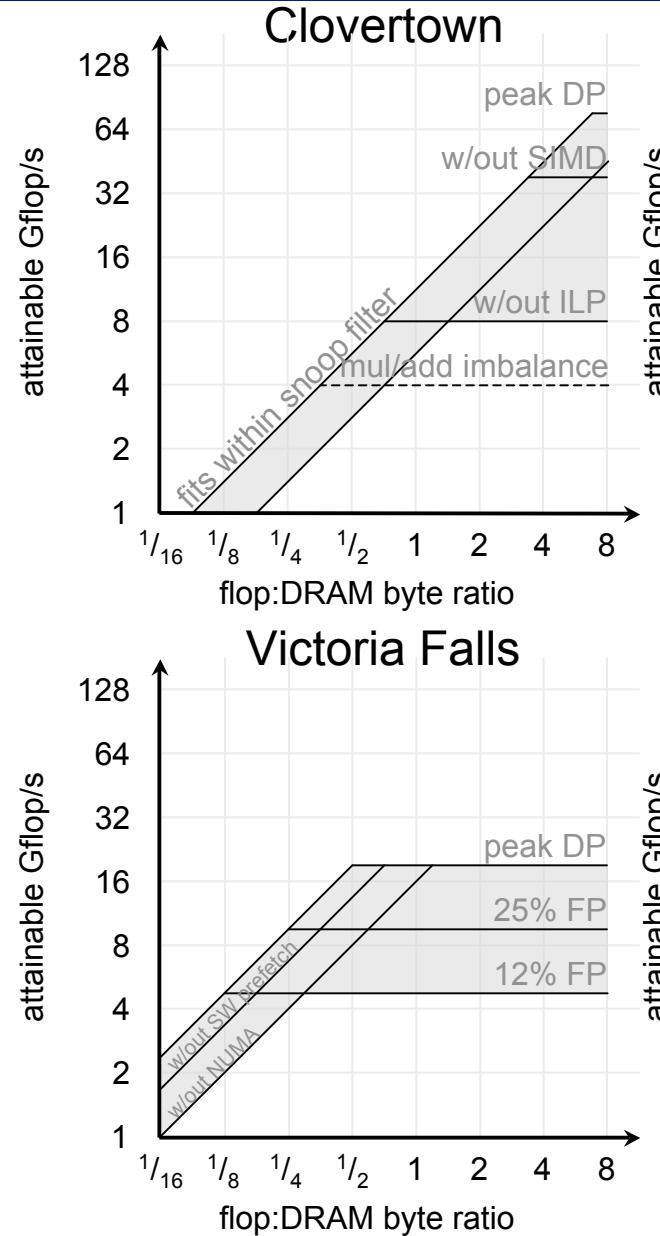
Sparse Matrix-Vector Multiplication (SpMV)

Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, James Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms", Supercomputing (SC), 2007.

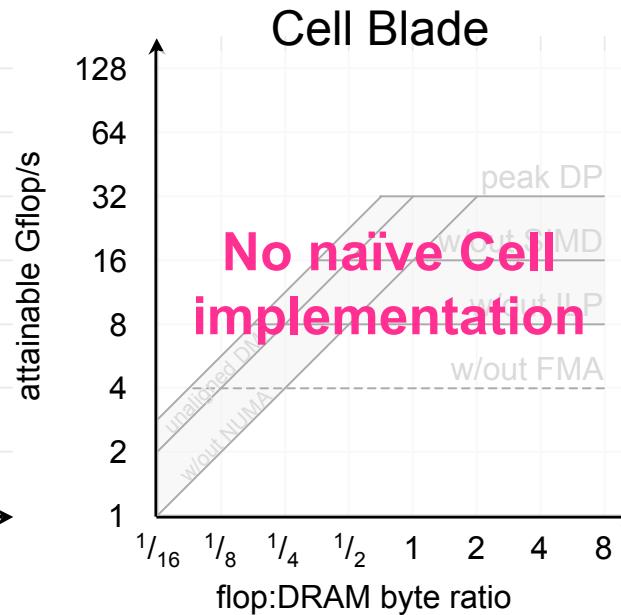
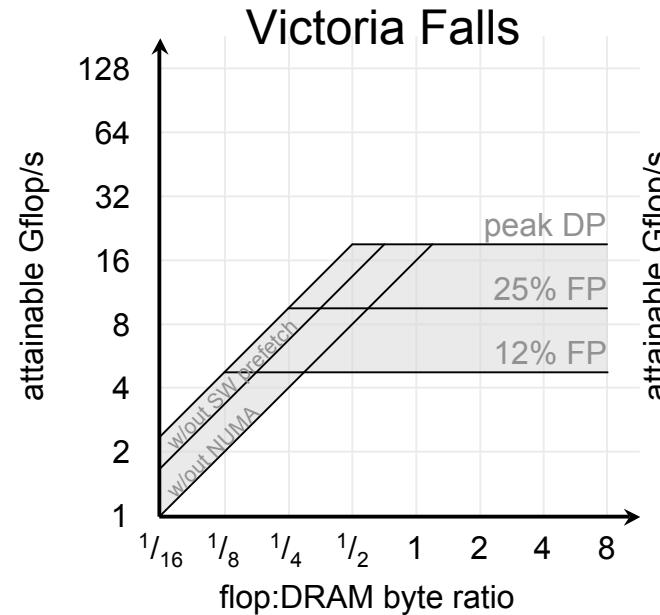
Sparse Matrix
Vector Multiplication

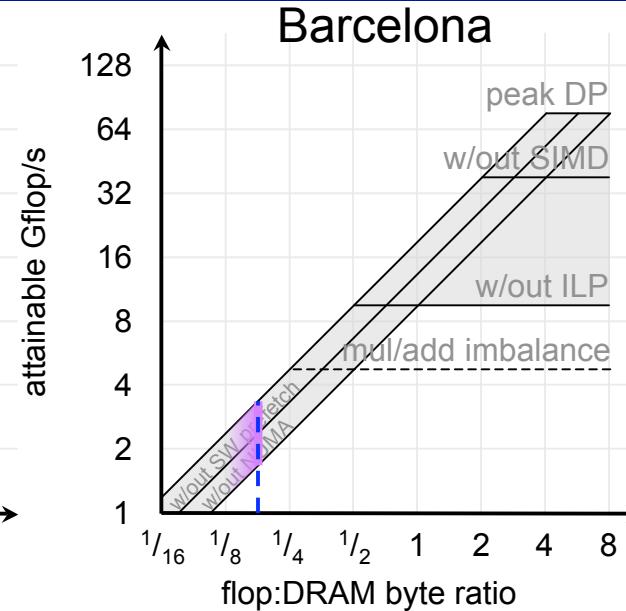
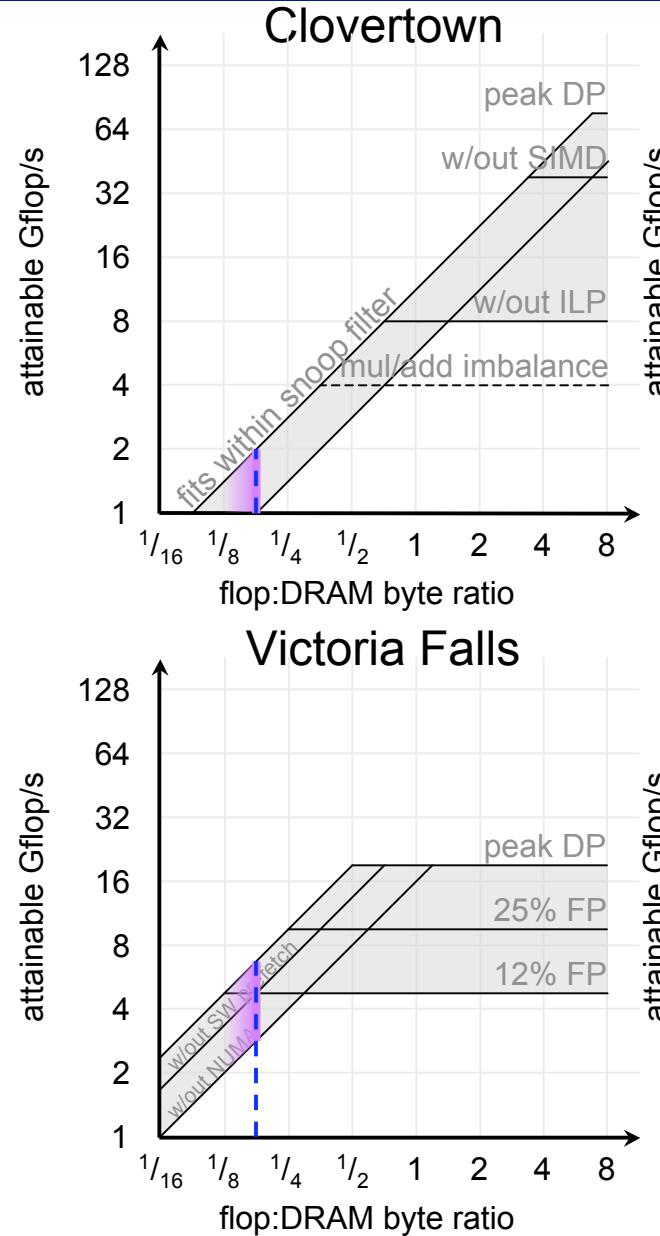
- ❖ Sparse Matrix
 - Most entries are 0.0
 - Performance advantage in only storing/operating on the nonzeros
 - Requires significant meta data
- ❖ Evaluate $y = Ax$
 - A is a sparse matrix
 - x & y are dense vectors
- ❖ Challenges
 - Difficult to exploit ILP(bad for superscalar),
 - Difficult to exploit DLP(bad for SIMD)
 - Irregular memory access to source vector
 - Difficult to load balance
 - **Very low arithmetic intensity (often <0.166 flops/byte)**
= likely memory bound

$$\begin{bmatrix} \text{Sparse Matrix} \\ A \end{bmatrix} \times \begin{bmatrix} \text{Dense Vector} \\ x \end{bmatrix} = \begin{bmatrix} \text{Dense Vector} \\ y \end{bmatrix}$$

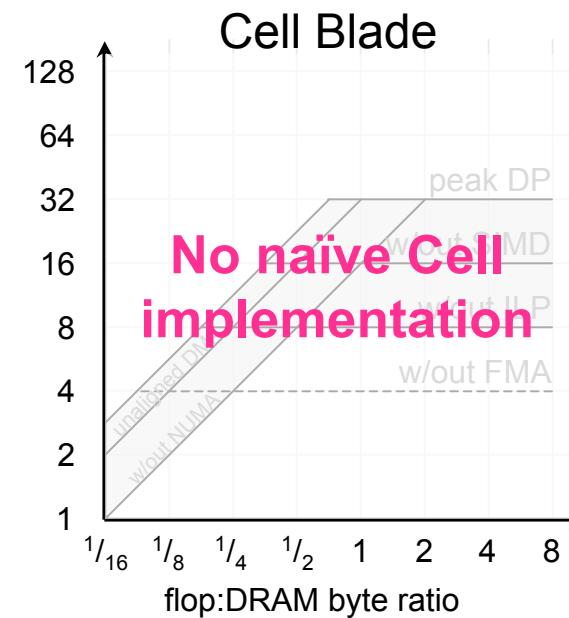
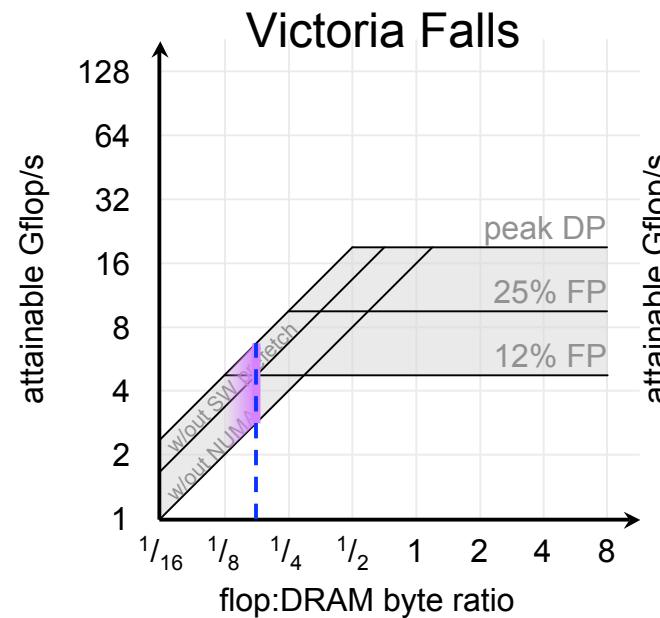


- ❖ Double precision roofline models
- ❖ FMA is inherent in SpMV (place at bottom)

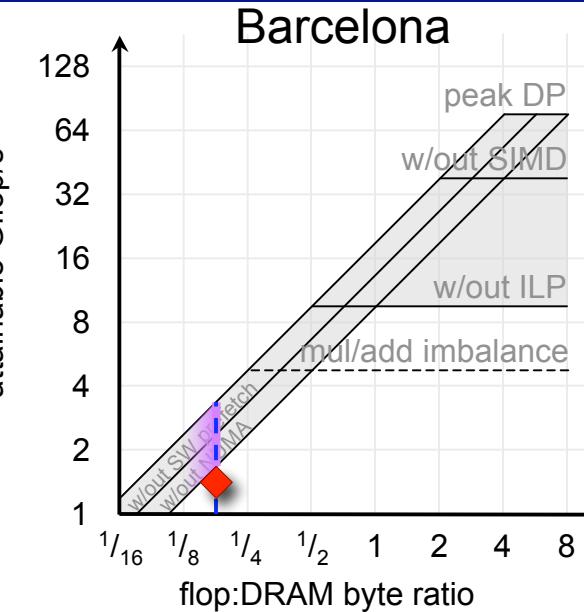
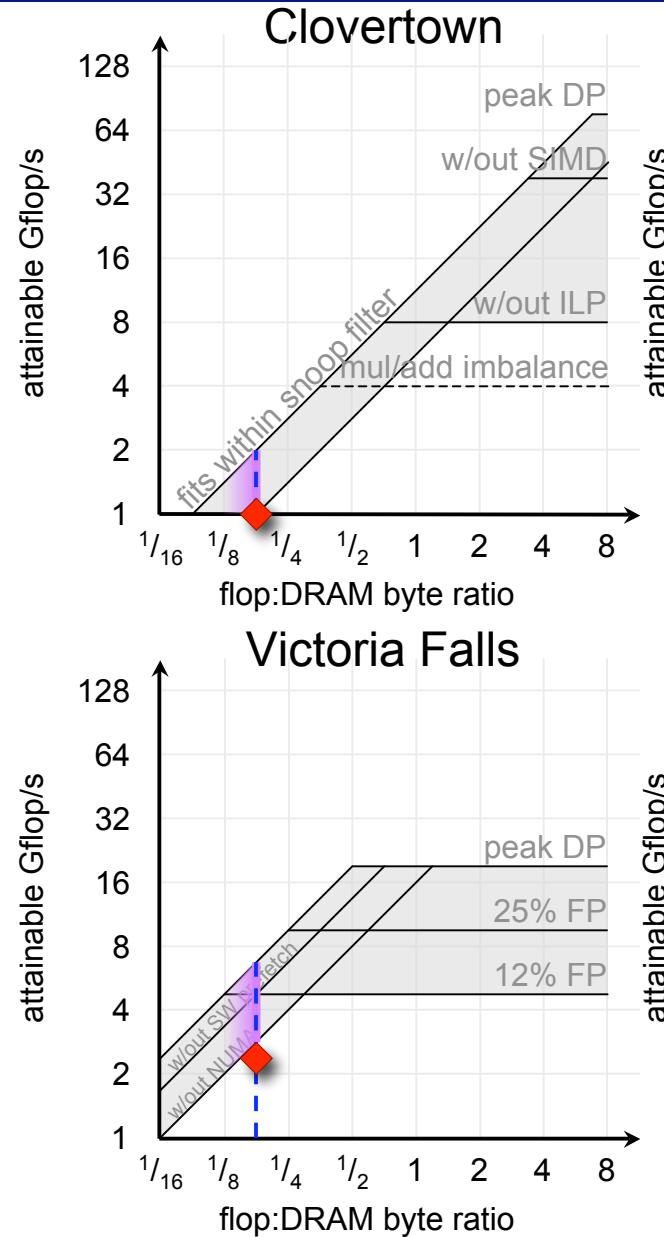




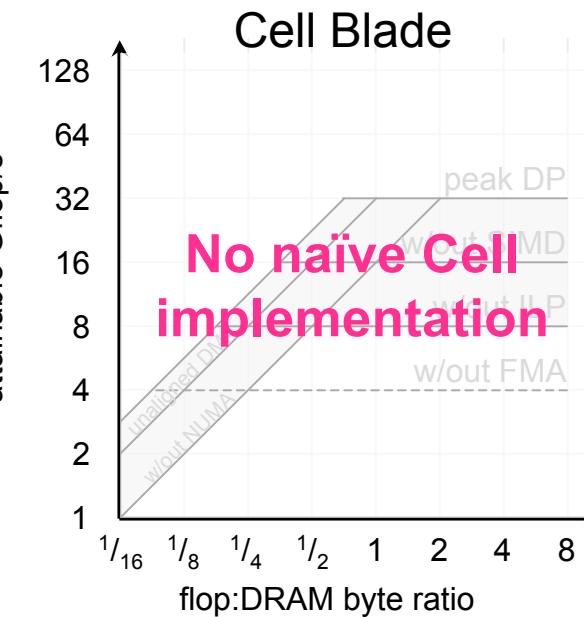
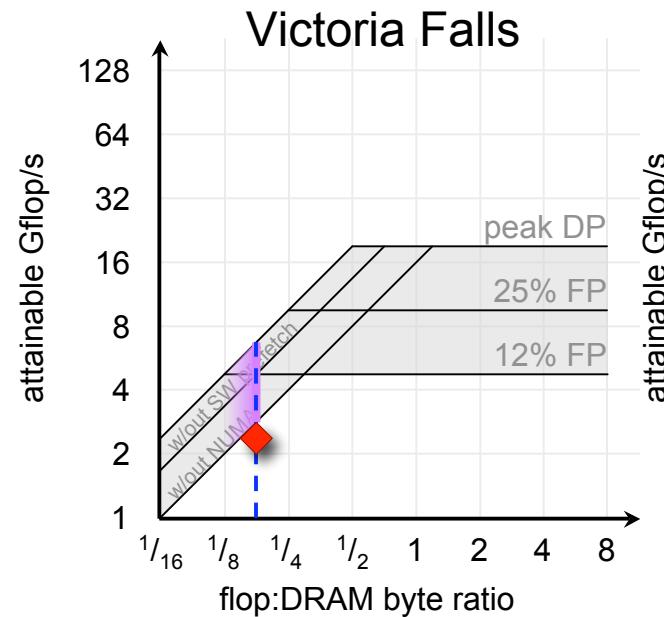
- ❖ Two unit stride streams
- ❖ Inherent FMA
- ❖ No ILP
- ❖ No DLP
- ❖ FP is 12-25%
- ❖ Naïve compulsory flop:byte < 0.166

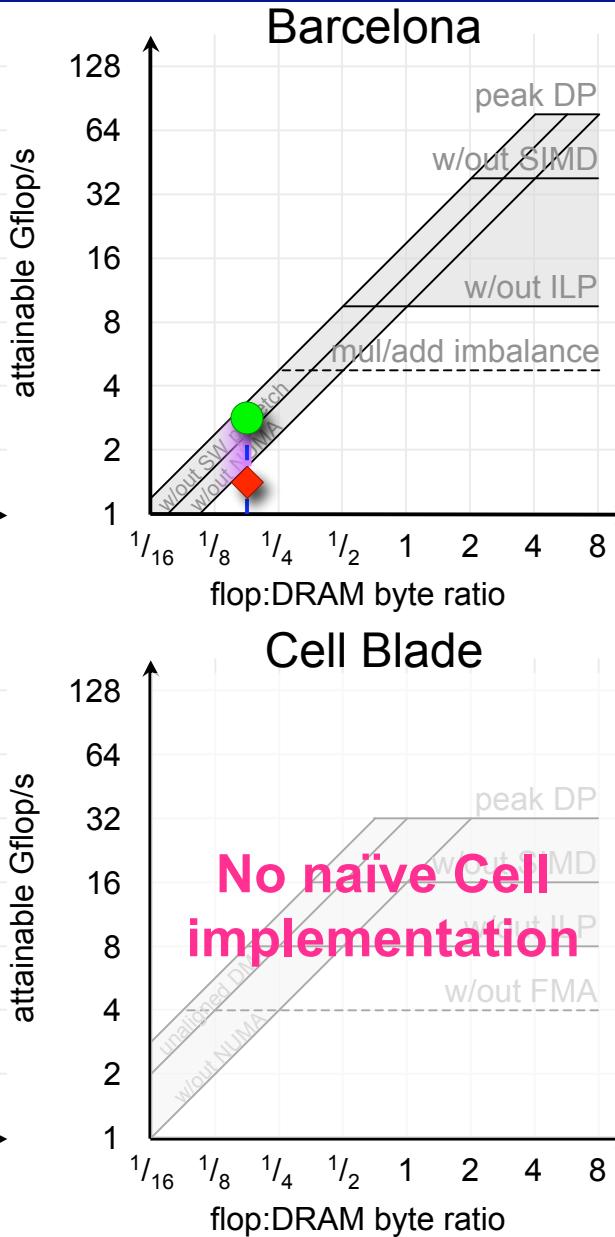
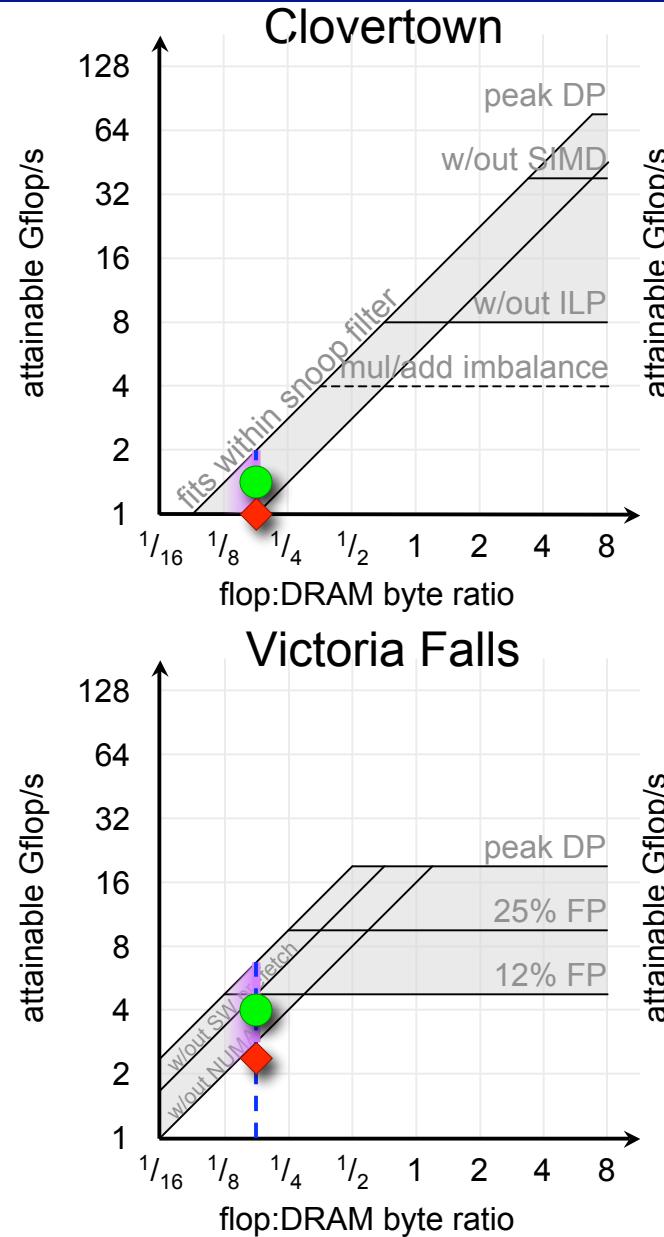


No naïve Cell implementation

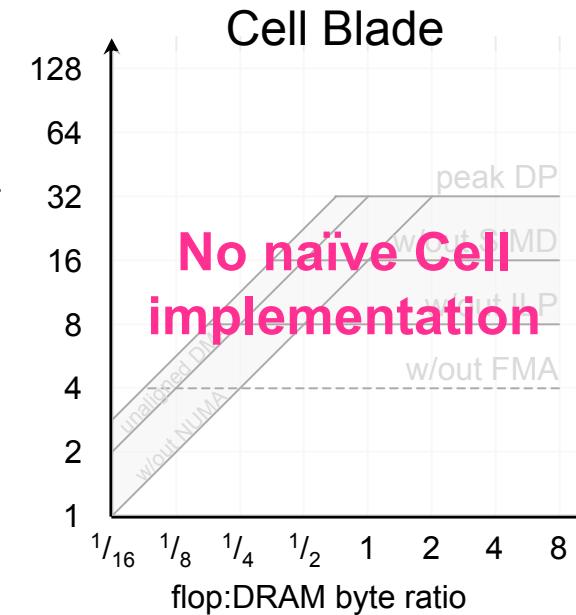
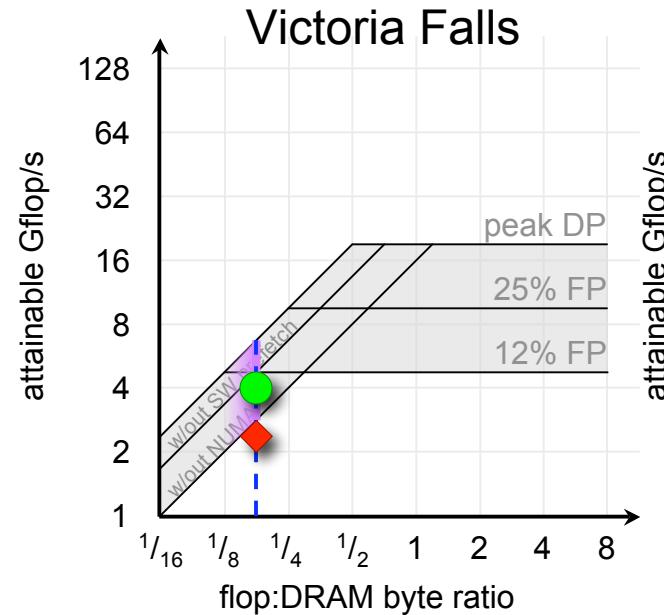


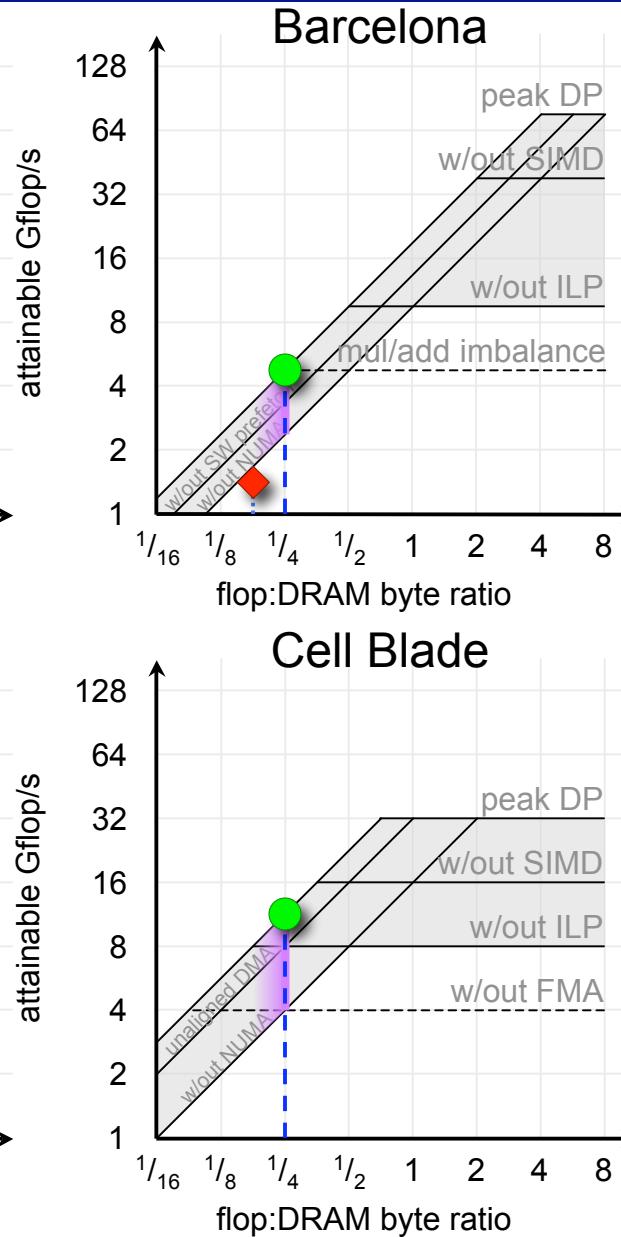
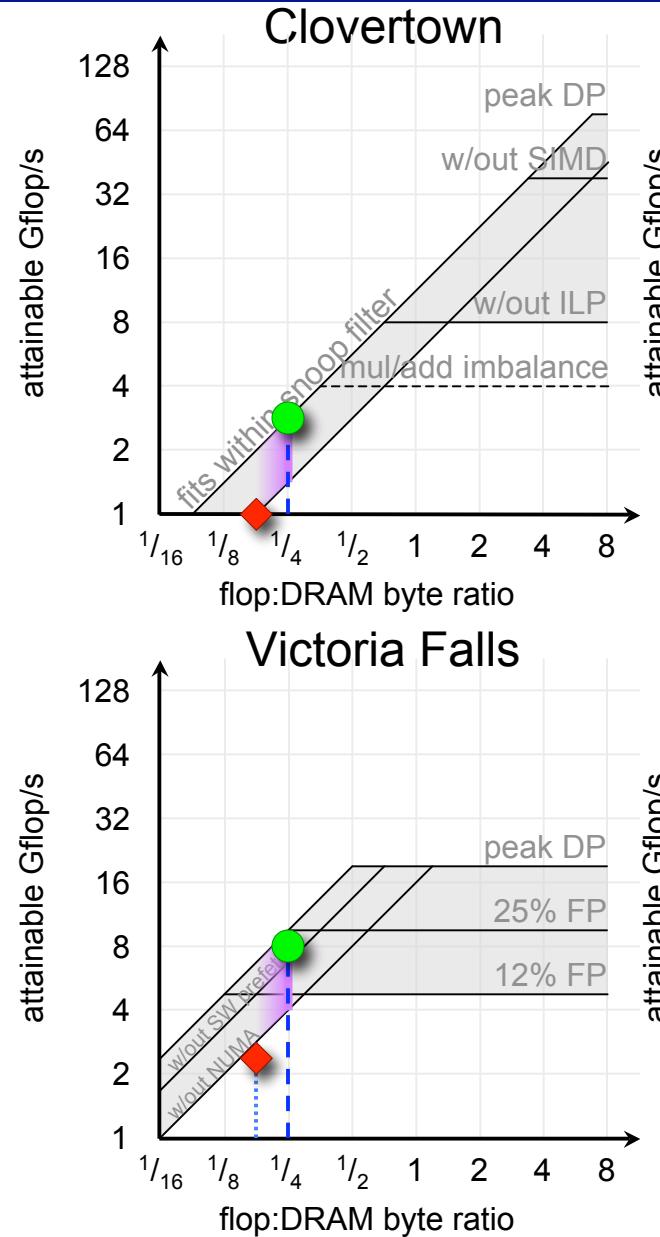
- ❖ Two unit stride streams
- ❖ Inherent FMA
- ❖ No ILP
- ❖ No DLP
- ❖ FP is 12-25%
- ❖ Naïve compulsory flop:byte < 0.166
- ❖ For simplicity: dense matrix in sparse format



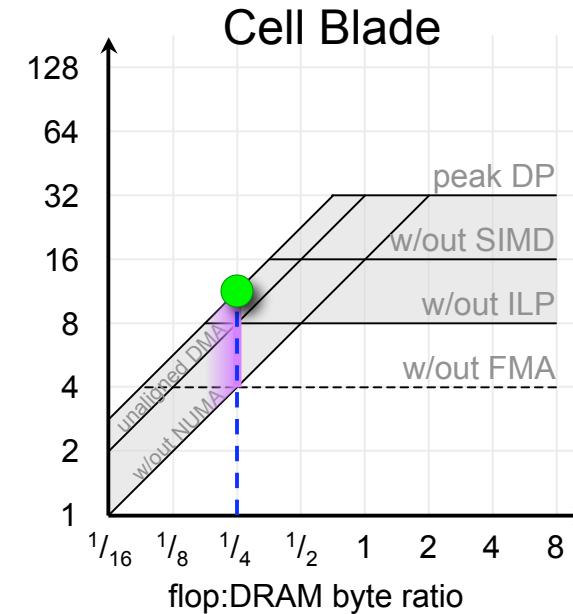
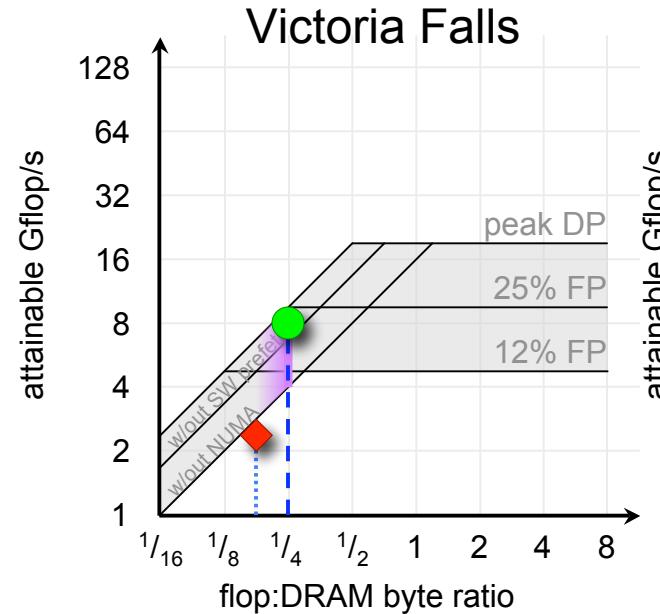


- ❖ compulsory flop:byte ~ 0.166
- ❖ utilize all memory channels





- ❖ Inherent FMA
- ❖ Register blocking improves ILP, DLP, flop:byte ratio, and FP% of instructions

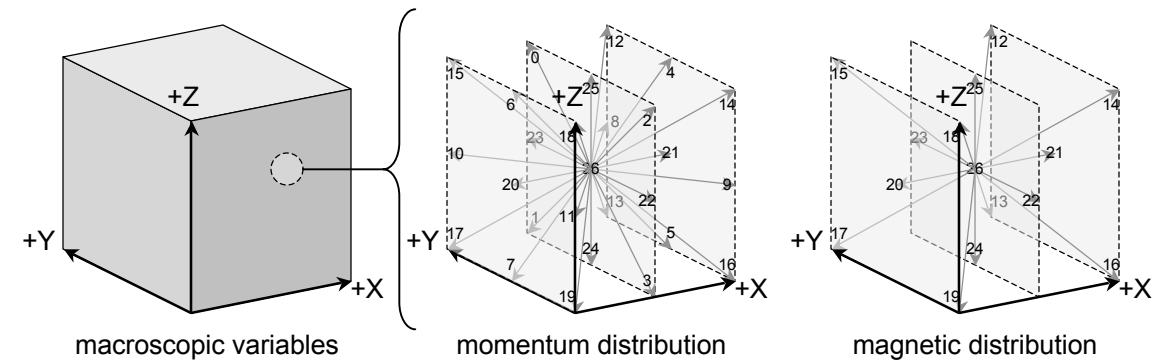


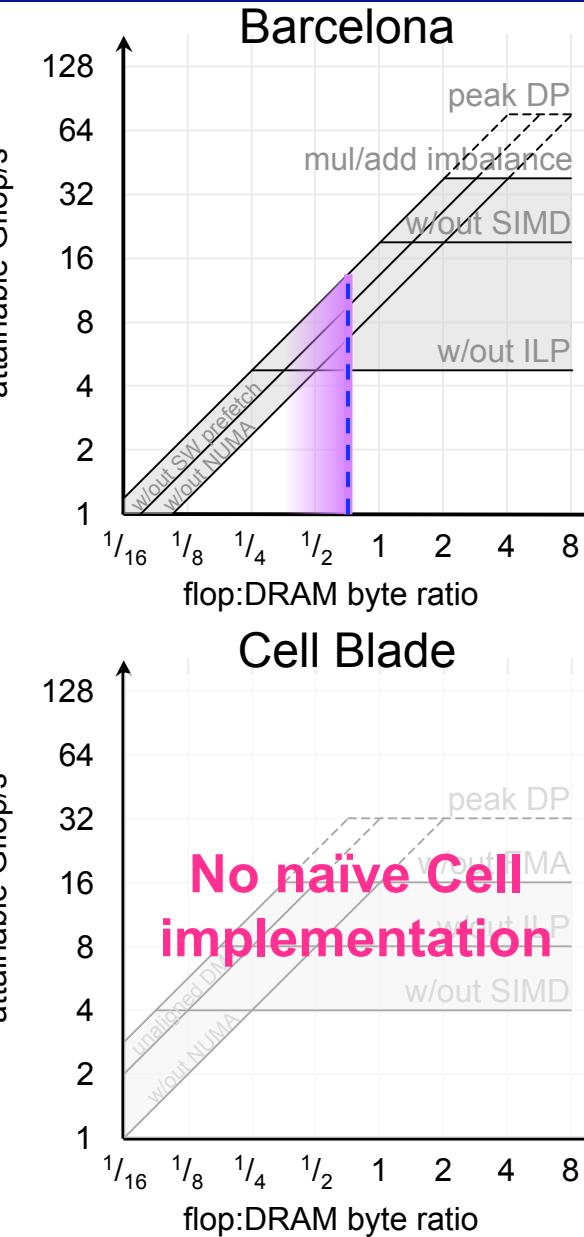
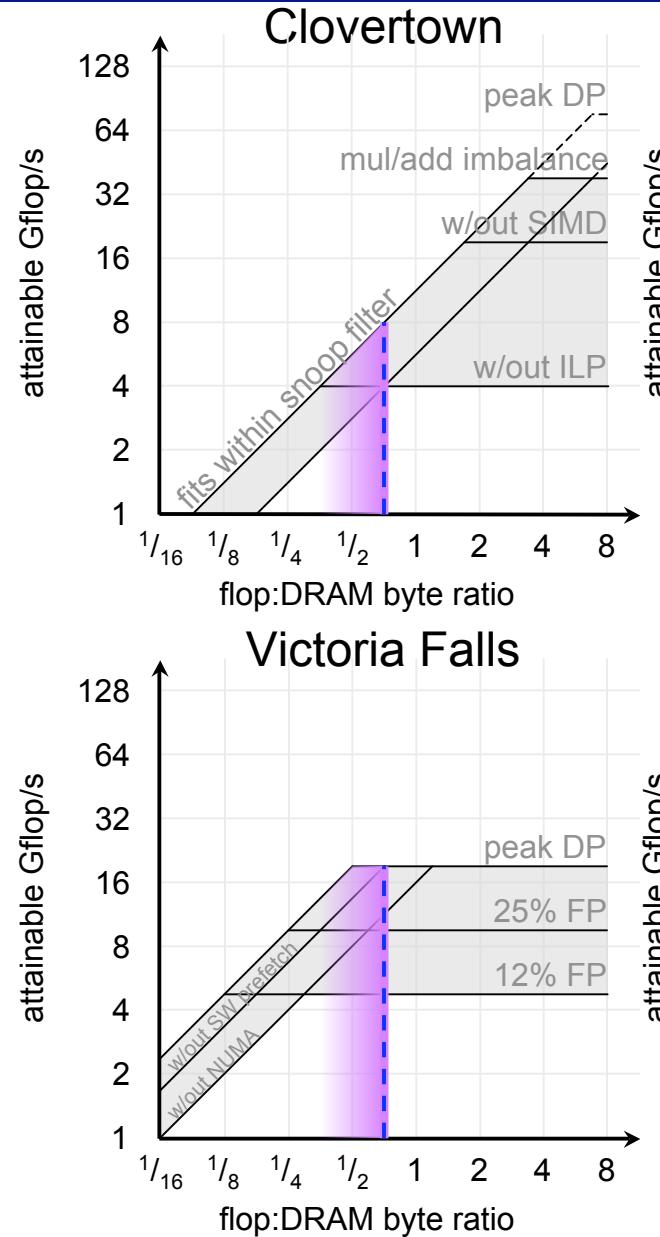
Lattice-Boltzmann Magneto-Hydrodynamics (LBMDH)

Samuel Williams, Jonathan Carter, Leonid Oliker, John Shalf, Katherine Yelick,
"Lattice Boltzmann Simulation Optimization on Leading Multicore Platforms",
International Parallel & Distributed Processing Symposium (IPDPS), 2008.

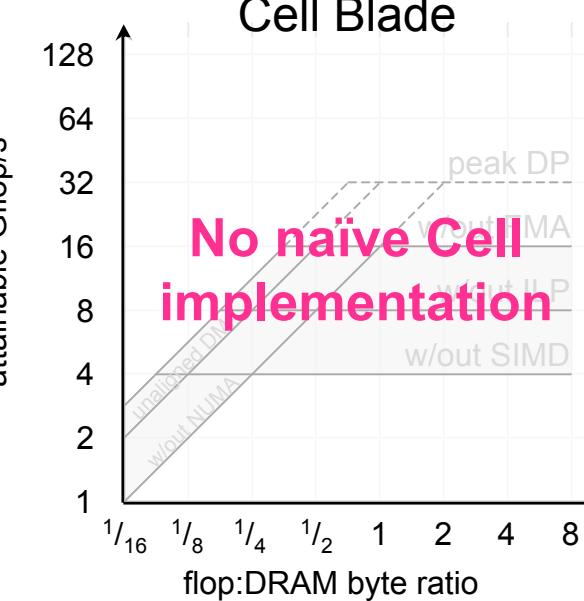
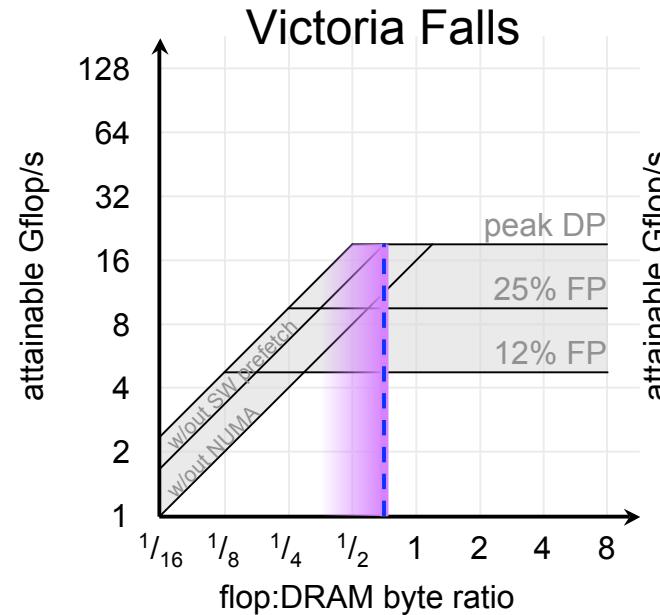
Best Paper, Application Track

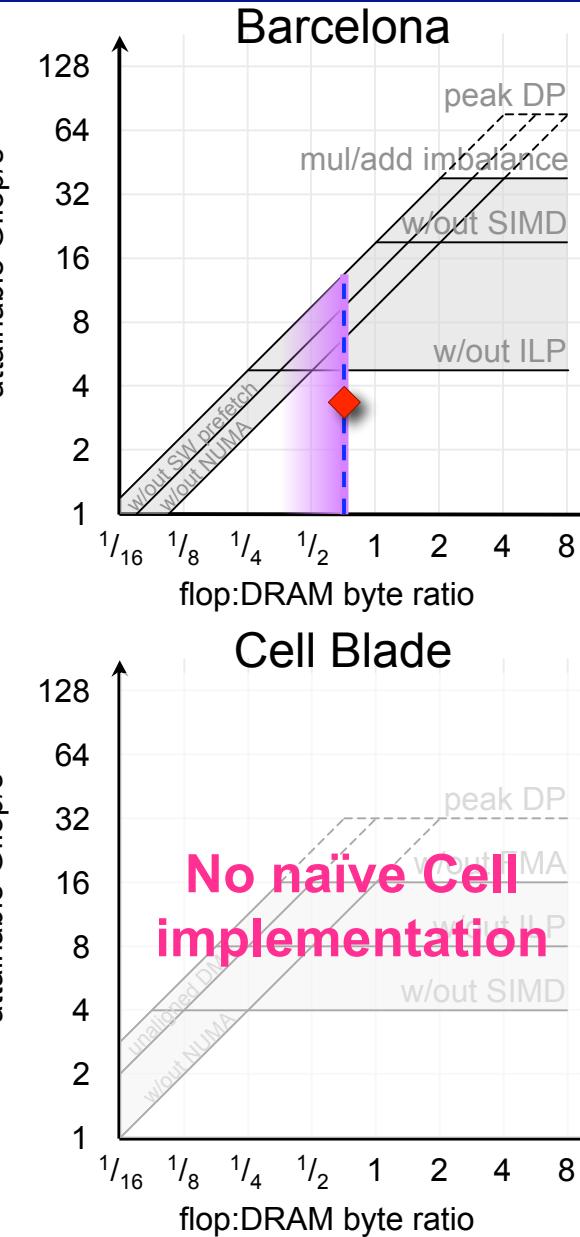
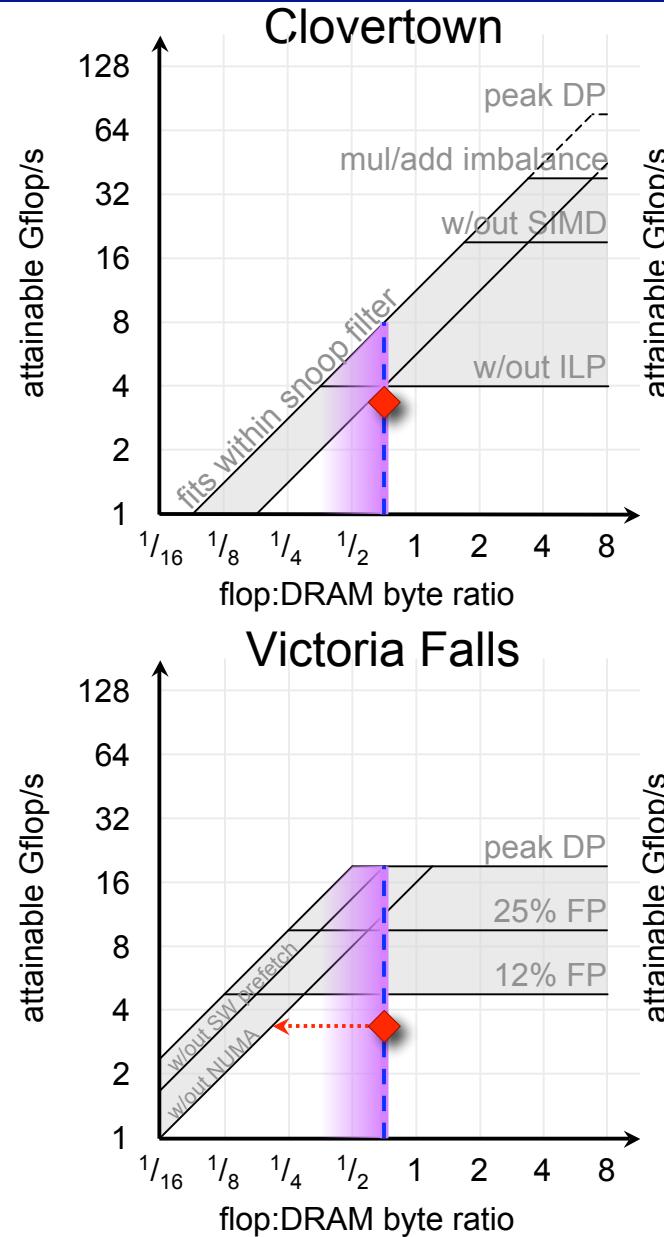
- ❖ Plasma turbulence simulation via Lattice Boltzmann Method
- ❖ Two distributions:
 - momentum distribution (27 scalar components)
 - magnetic distribution (15 vector components)
- ❖ Three macroscopic quantities:
 - Density
 - Momentum (vector)
 - Magnetic Field (vector)
- ❖ Must read 73 doubles, and update 79 doubles per point in space
- ❖ Requires about 1300 floating point operations per point in space
- ❖ Just over 1.0 flops/byte (ideal)



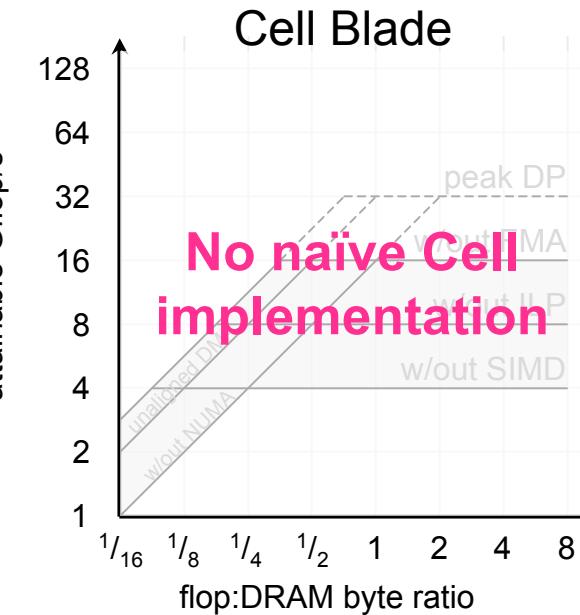
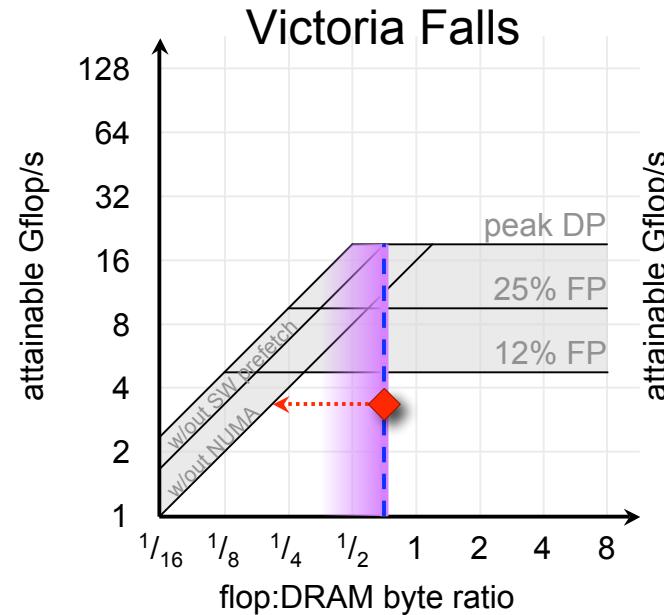


- ❖ Huge datasets
- ❖ NUMA allocation/access
- ❖ Little ILP
- ❖ No DLP
- ❖ Far more adds than multiplies (imbalance)
- ❖ **Essentially random access to memory**
- ❖ Flop:byte ratio ~ 0.7
- ❖ High conflict misses

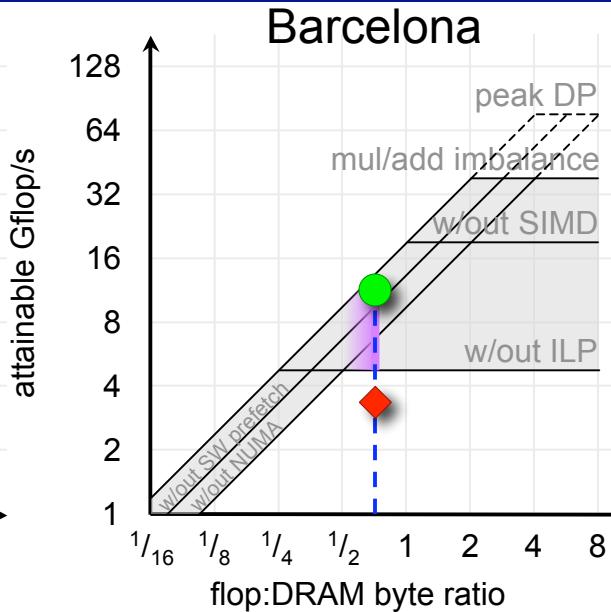
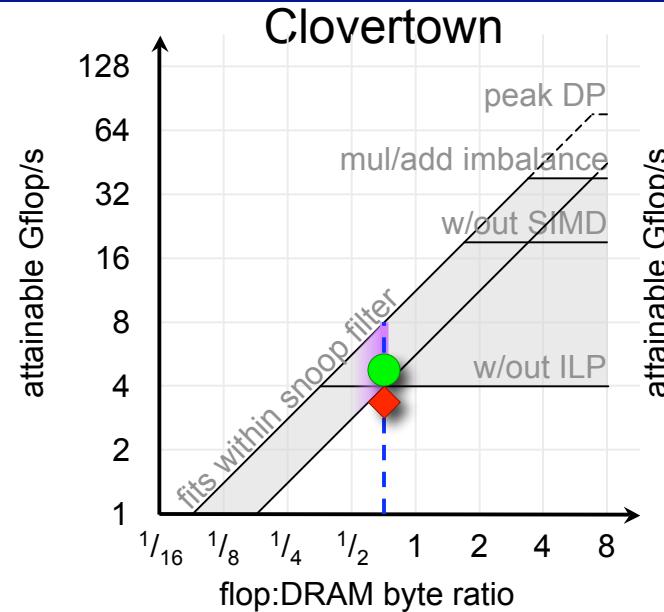




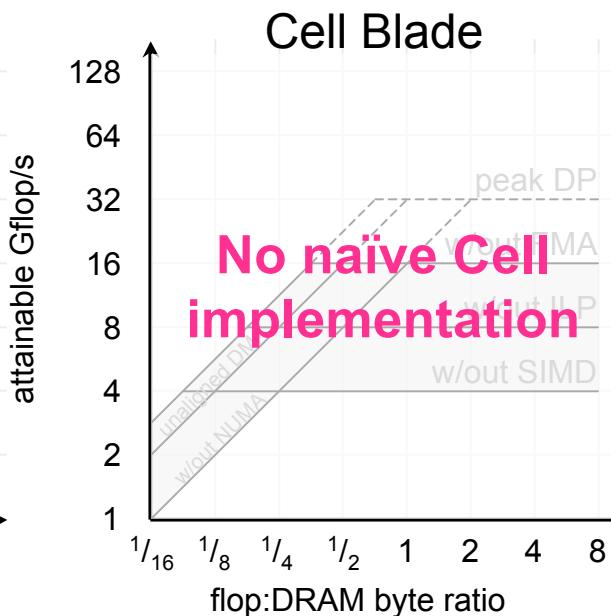
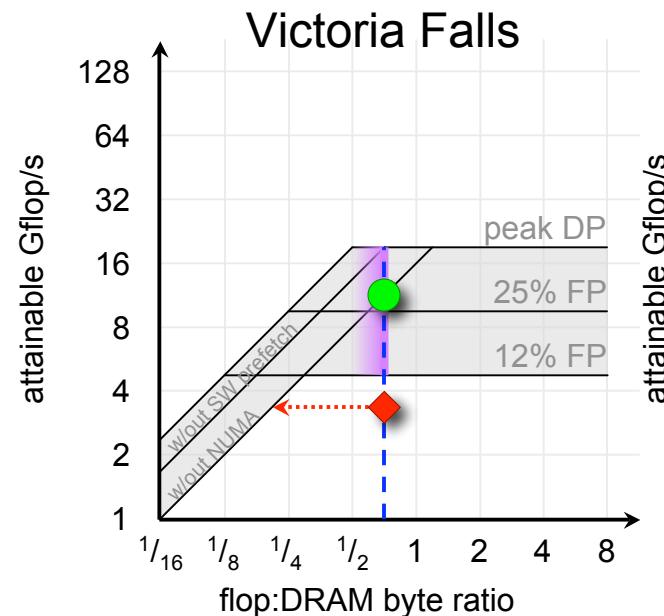
- ❖ Huge datasets
- ❖ NUMA allocation/access
- ❖ Little ILP
- ❖ No DLP
- ❖ Far more adds than multiplies (imbalance)
- ❖ **Essentially random access to memory**
- ❖ Flop:byte ratio ~ 0.7
- ❖ High conflict misses
- ❖ Peak VF performance with 64 threads (out of 128) - high conflict misses

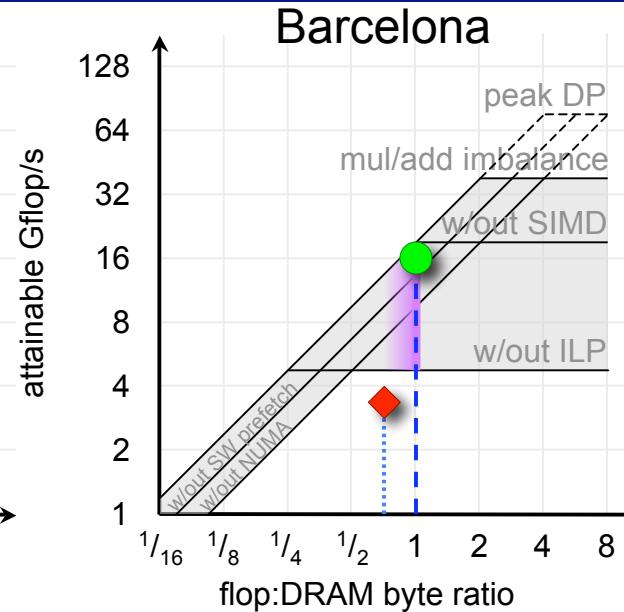
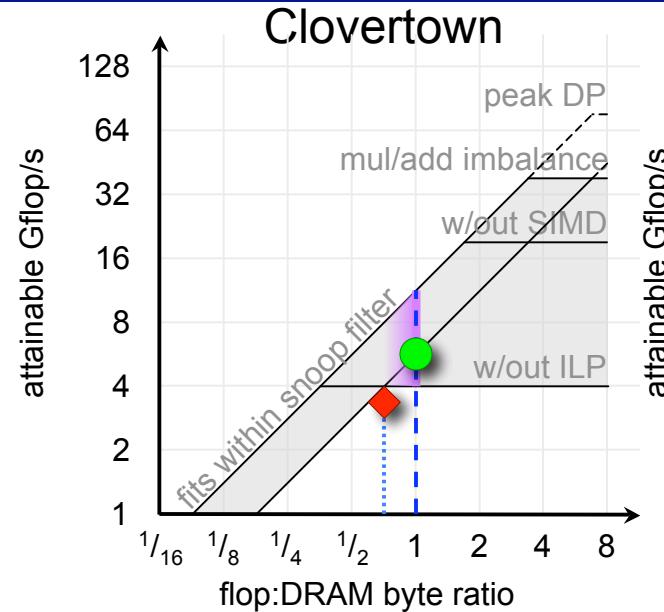


No naive Cell implementation

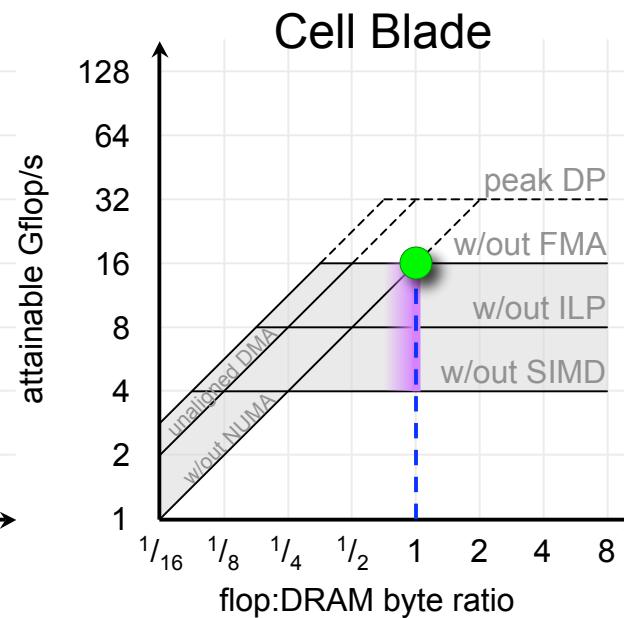
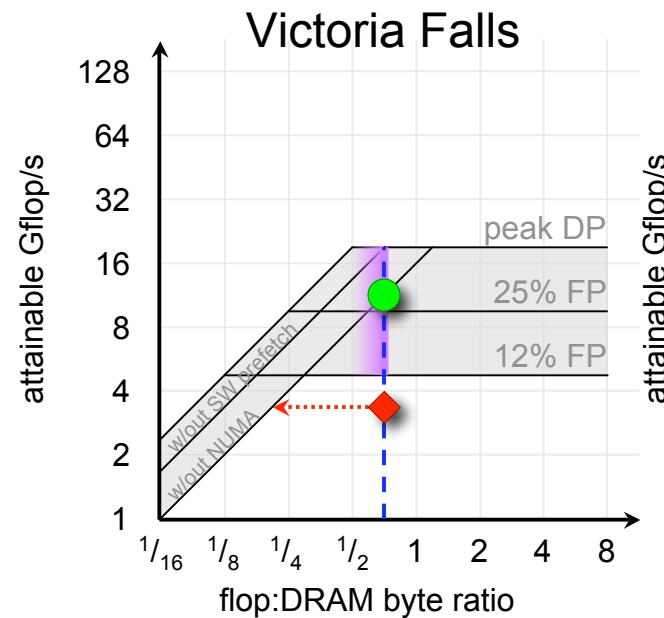


- ❖ Vectorize the code to eliminate TLB capacity misses
- ❖ Ensures unit stride access (bottom bandwidth ceiling)
- ❖ Tune for optimal VL
- ❖ Clovertown pinned to lower BW ceiling





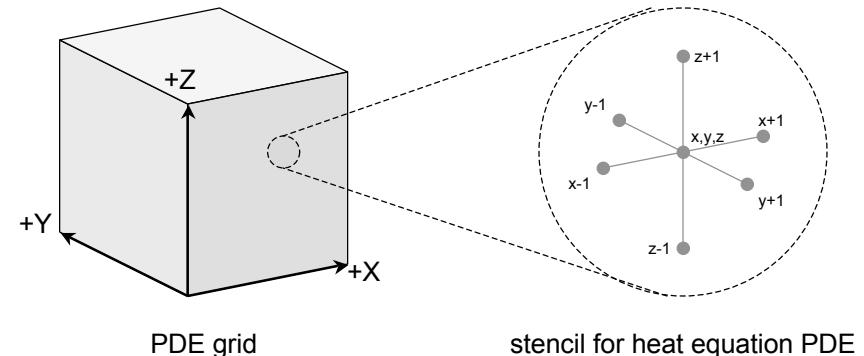
- ❖ Make SIMDization explicit
- ❖ Technically, this swaps ILP and SIMD ceilings
- ❖ Use cache bypass instruction: ***movntpd***
- ❖ Increases flop:byte ratio to ~1.0 on x86/Cell

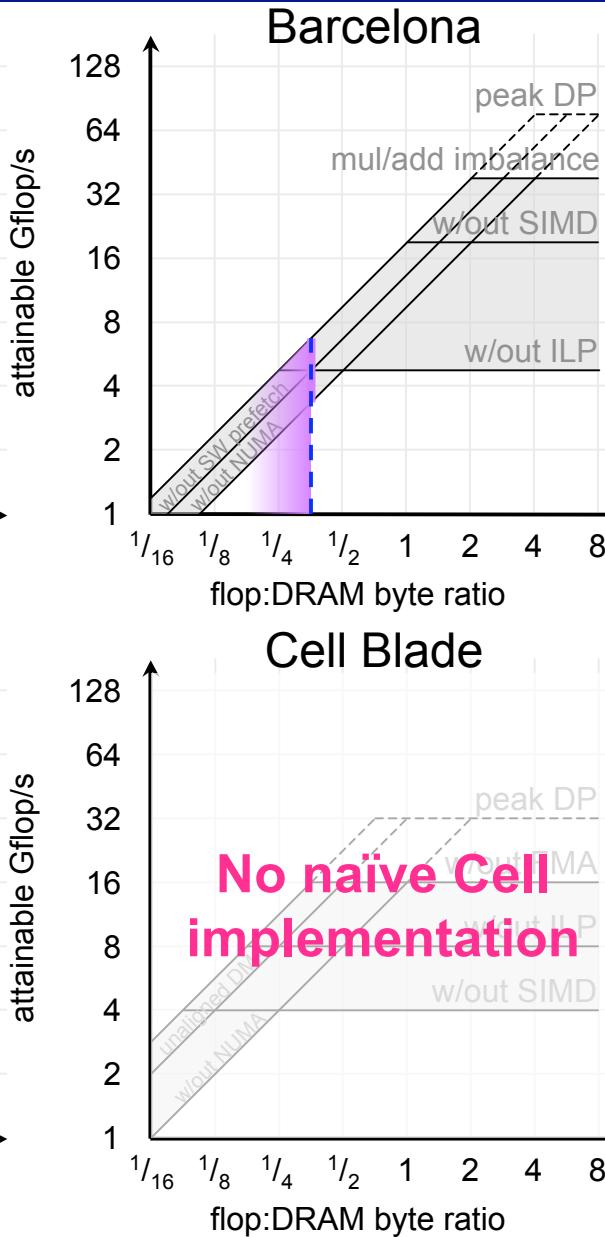
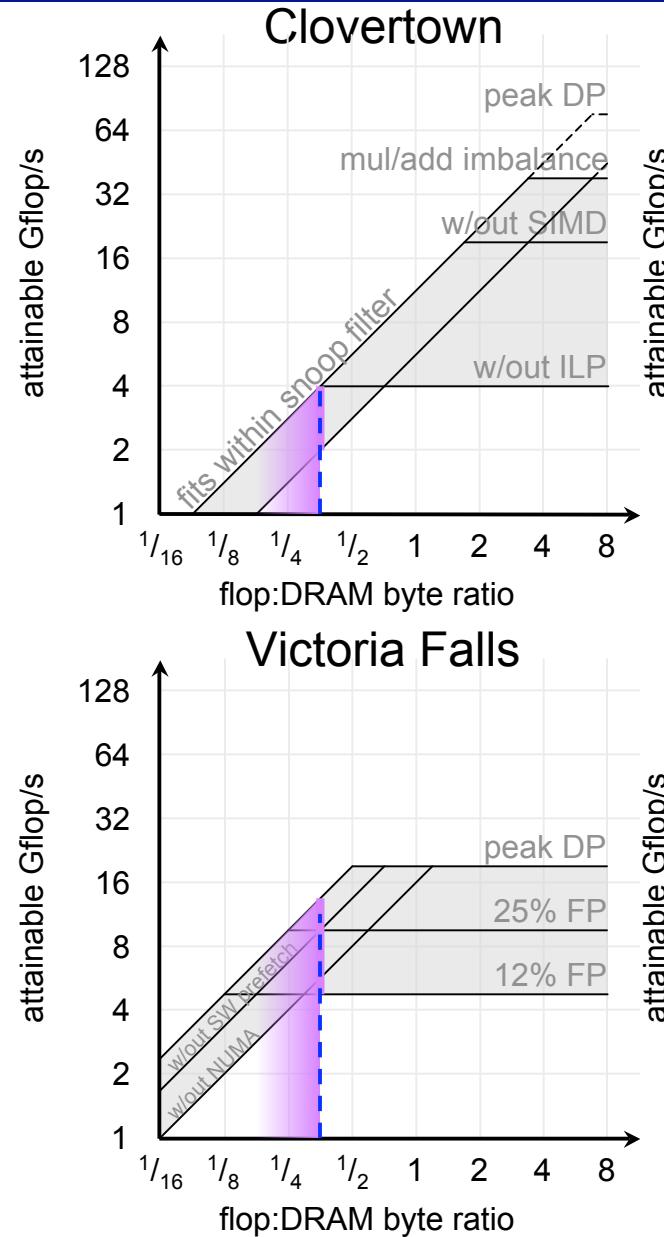


The Heat Equation Stencil

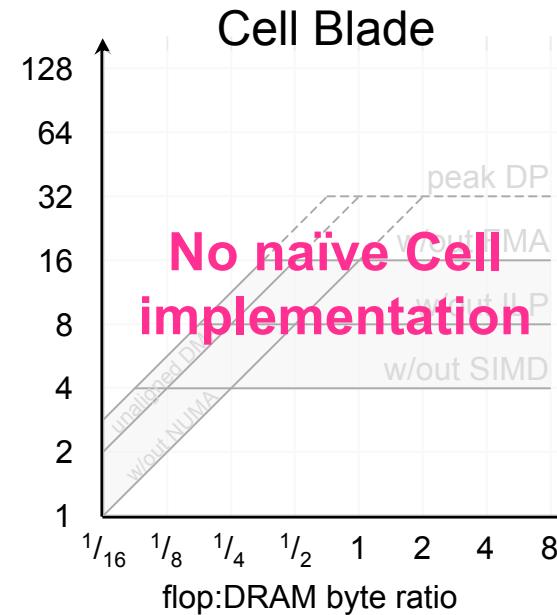
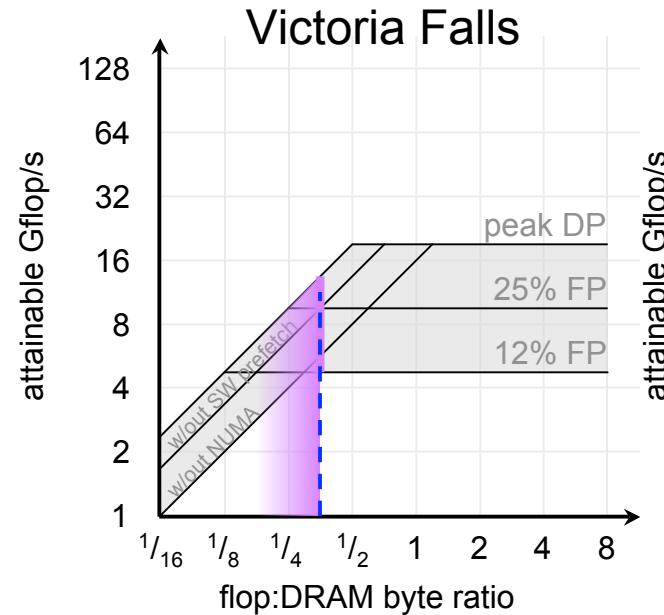
Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, Katherine Yelick, “Stencil Computation Optimization and Autotuning on State-of-the-Art Multicore Architecture”, submitted to Supercomputing (SC), 2008.

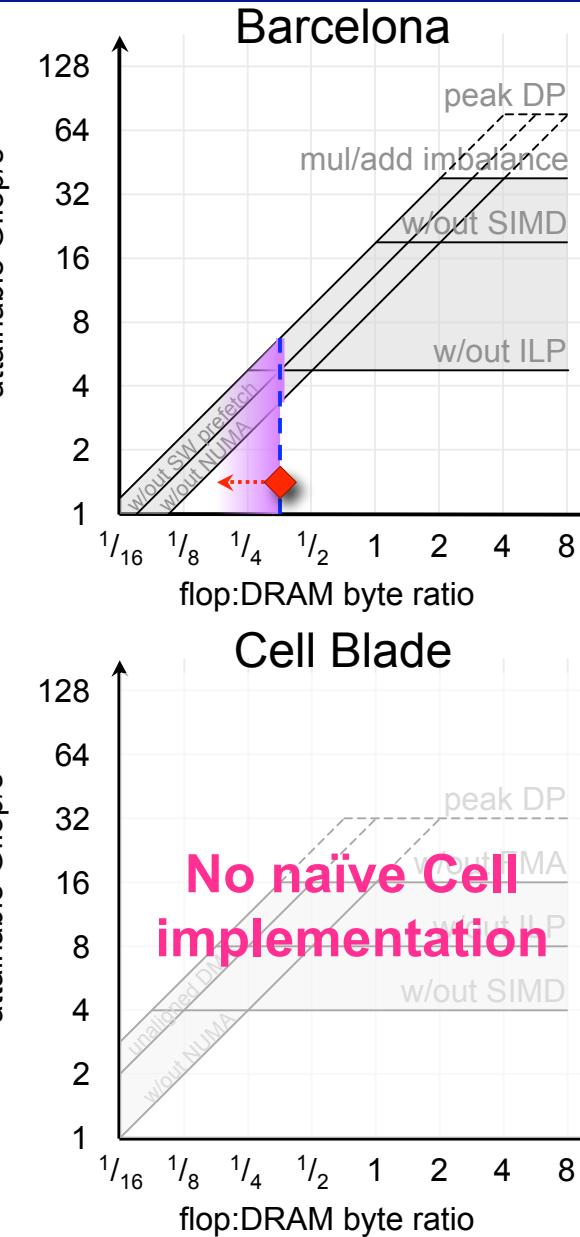
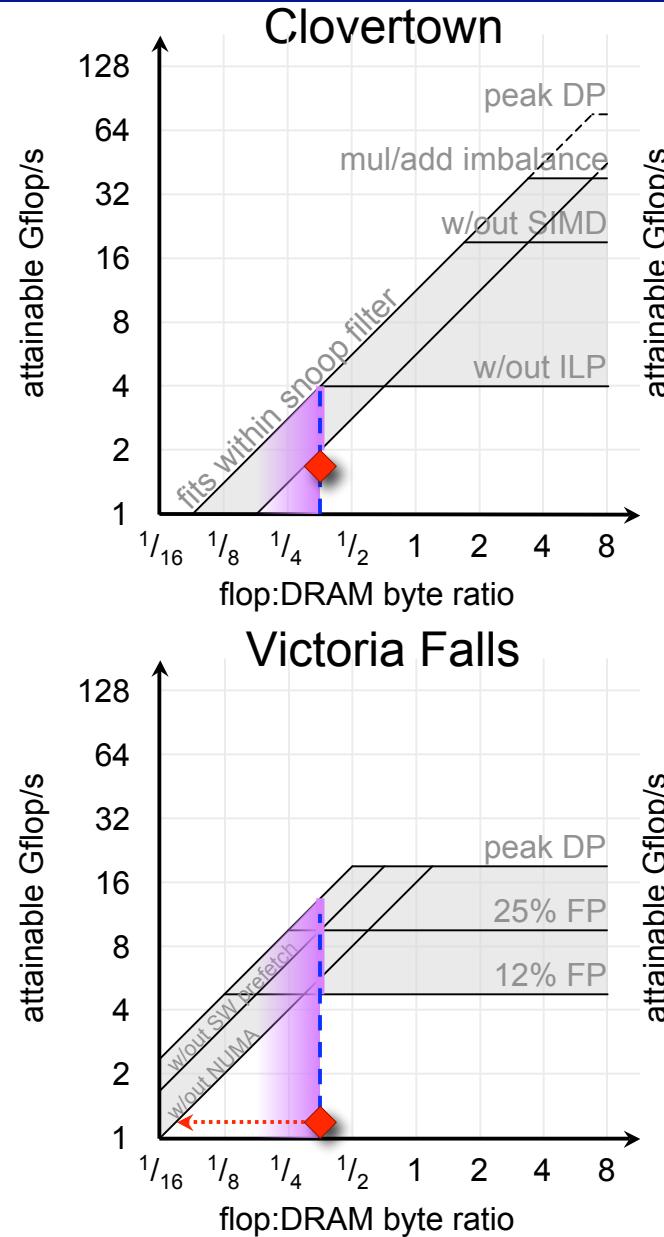
- ❖ Explicit Heat equation on a regular grid
- ❖ Jacobi
- ❖ One double per point in space
- ❖ 7-point nearest neighbor stencil
- ❖ Must:
 - read every point from DRAM
 - perform 8 flops (linear combination)
 - write every point back to DRAM
- ❖ Just over 0.5 flops/byte (ideal)
- ❖ Cache locality is important



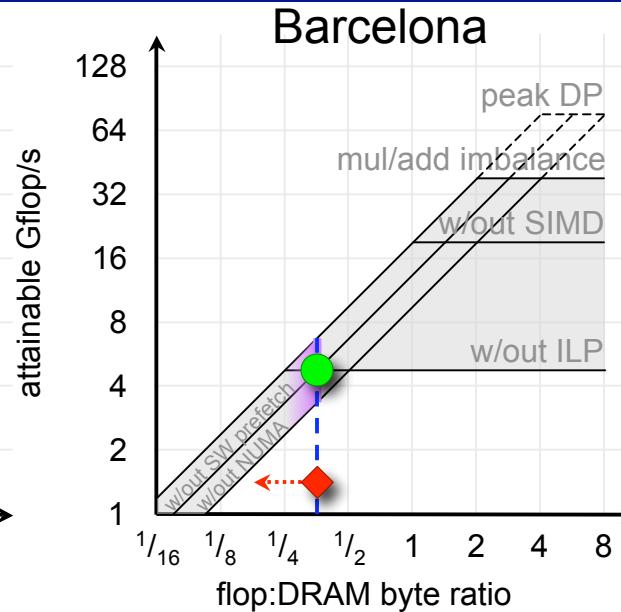
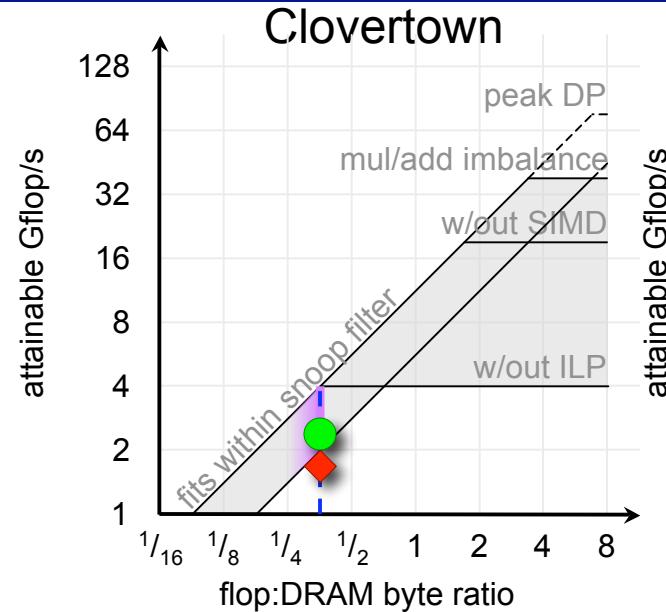


- ❖ Large datasets
- ❖ 2 unit stride streams
- ❖ No NUMA
- ❖ Little ILP
- ❖ No DLP
- ❖ Far more adds than multiplies (imbalance)
- ❖ Ideal flop:byte ratio $1/3$
- ❖ High locality requirements
- ❖ Capacity and conflict misses will severely impair flop:byte ratio

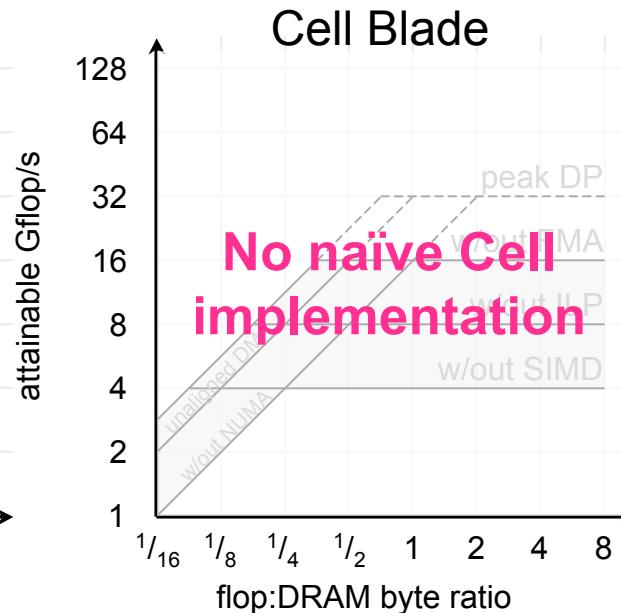
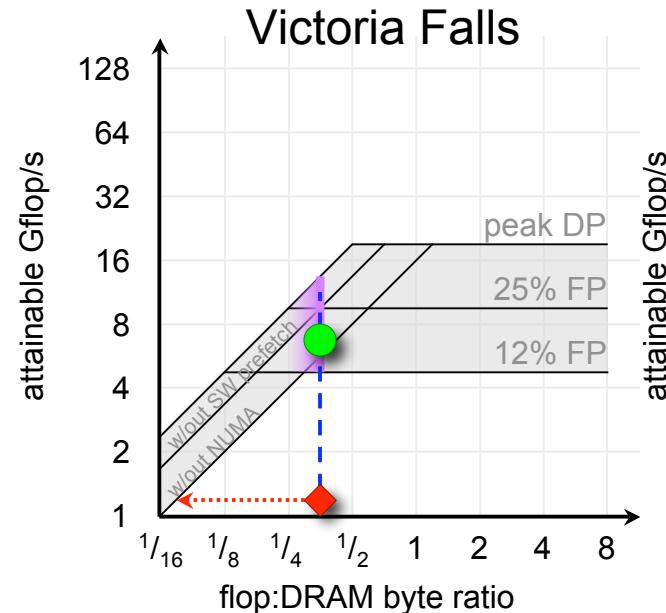


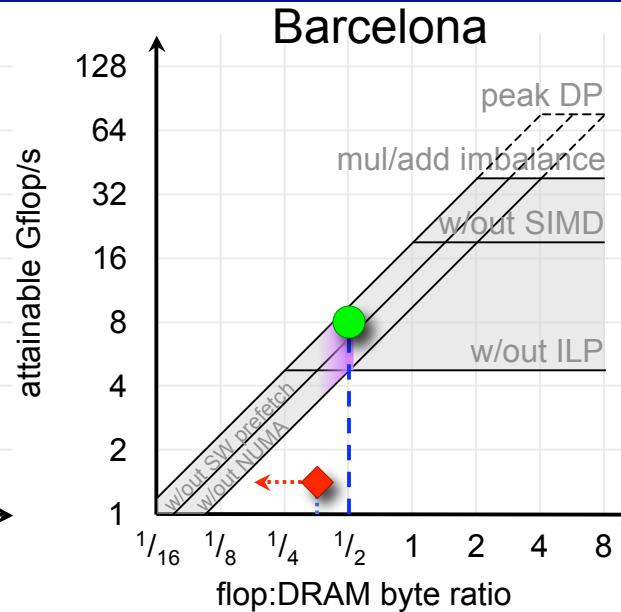
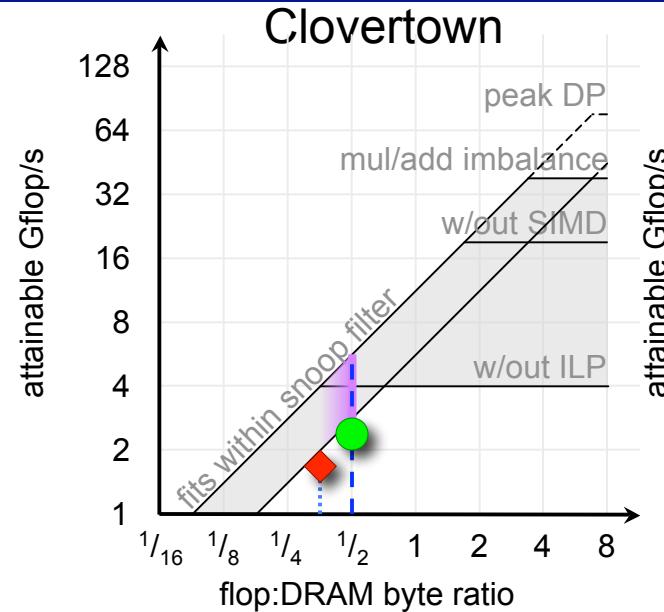


- ❖ Large datasets
- ❖ 2 unit stride streams
- ❖ No NUMA
- ❖ Little ILP
- ❖ No DLP
- ❖ Far more adds than multiplies (imbalance)
- ❖ Ideal flop:byte ratio $1/3$
- ❖ High locality requirements
- ❖ Capacity and conflict misses will severely impair flop:byte ratio

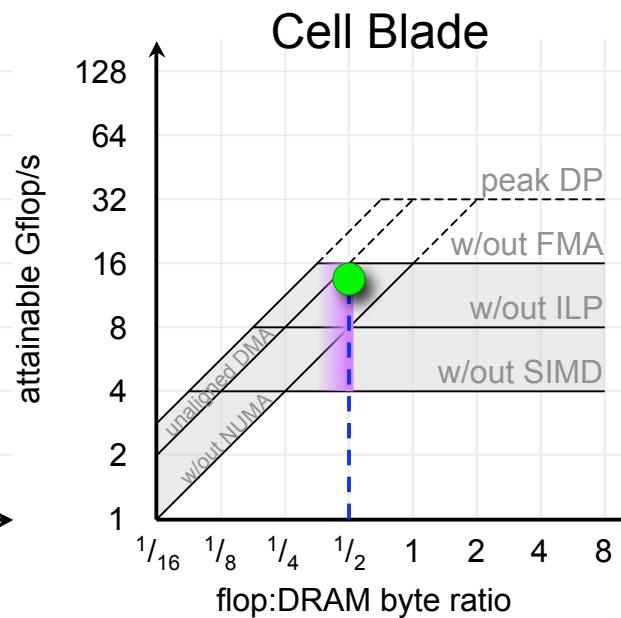
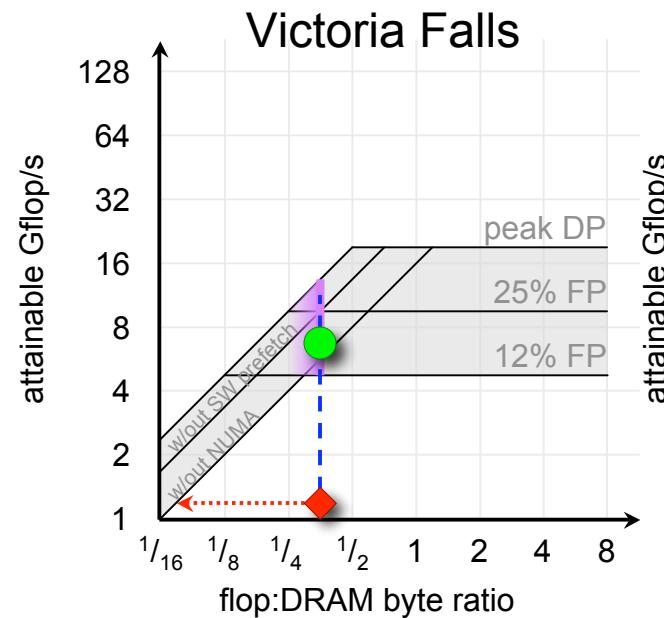


- ❖ Cache blocking helps ensure flop:byte ratio is as close as possible to $\frac{1}{3}$
- ❖ Clovertown has huge caches but is pinned to lower BW ceiling
- ❖ Cache management is essential when capacity/thread is low





- ❖ Make SIMDization explicit
- ❖ Technically, this swaps ILP and SIMD ceilings
- ❖ Use cache bypass instruction: ***movntpd***
- ❖ Increases flop:byte ratio to ~0.5 on x86/Cell

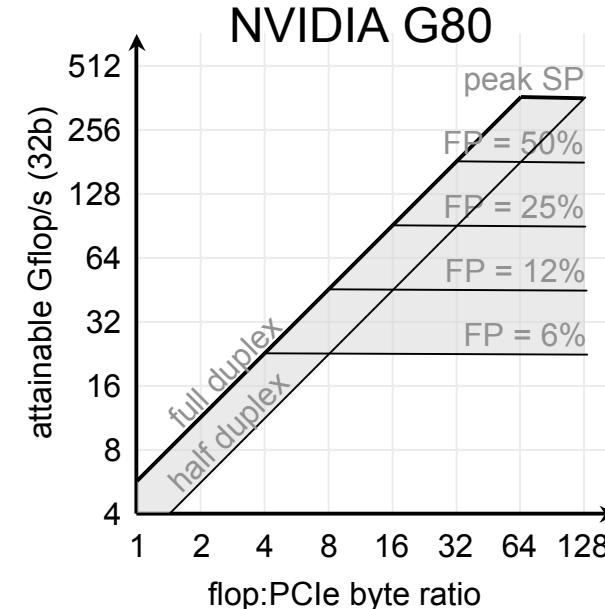


Refining the Roofline

- ❖ There is no reason either floating point (Gflop/s) must be the performance metric
- ❖ Could also use:
 - Graphics (Pixels, Vertices, Textures)
 - Crypto
 - Integer
 - Bitwise
 - etc...

Other bandwidths

- ❖ For our kernels, DRAM bandwidth is the key communication component.
- ❖ For other kernels, other bandwidths might be more appropriate
 - L2 bandwidth (e.g. DGEMM)
 - PCIe bandwidth (offload to GPU)
 - Network bandwidth
- ❖ The example below shows zero overhead double buffered transfers to/from a GPU over PCIe x16
 - How bad is a SP stencil ?
 - What about SGEMM ?
- ❖ No overlap / high overhead tends to smooth performance
 - Performance is half at ridge point



- ❖ In general, you can mix and match as the kernel/architecture requires:
- ❖ e.g. all possibilities is the cross product of performance metrics with bandwidths

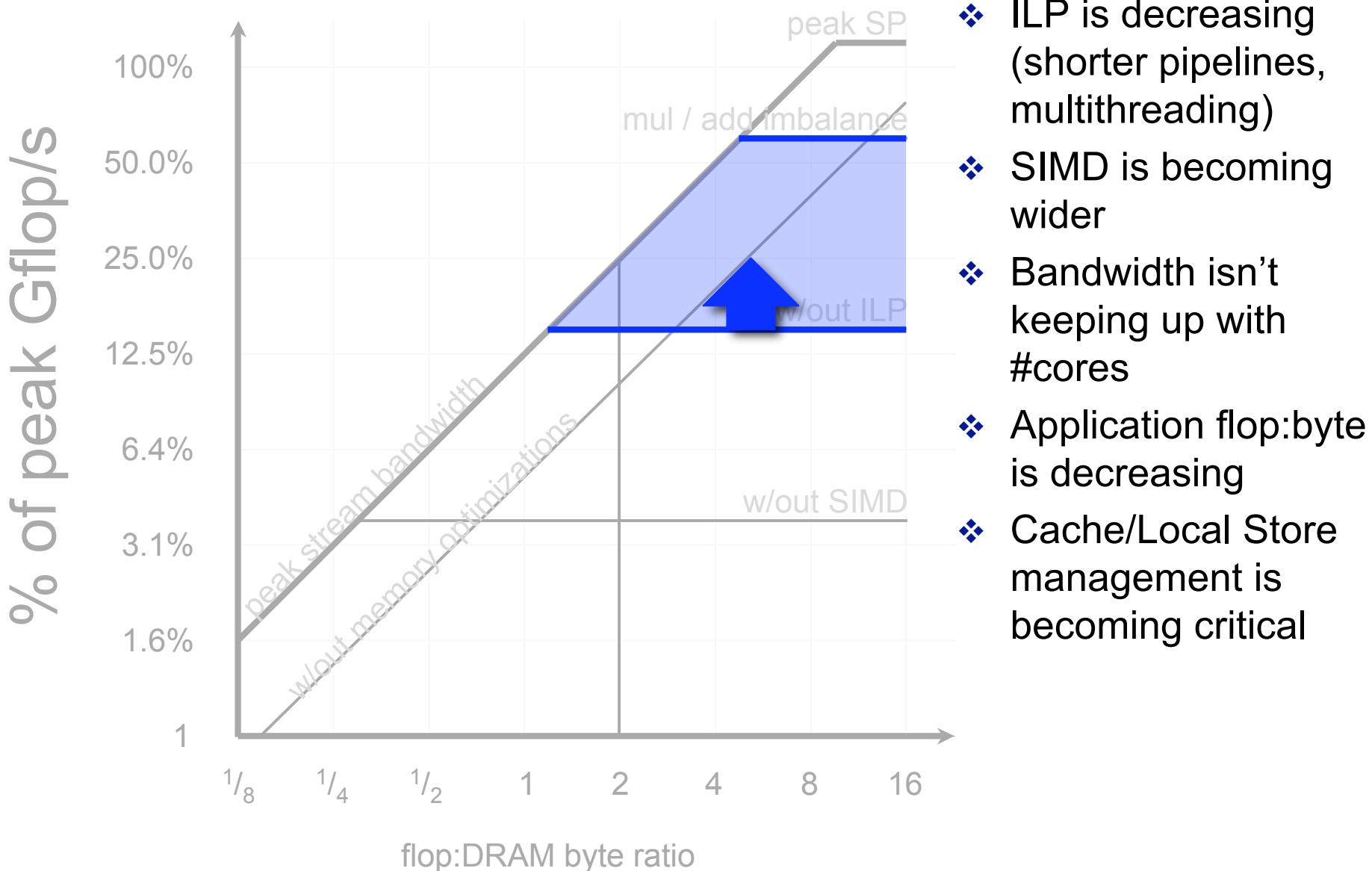
{Gflop/s, GIPS, crypto, ...} \times {L2, DRAM, PCIe, Network}

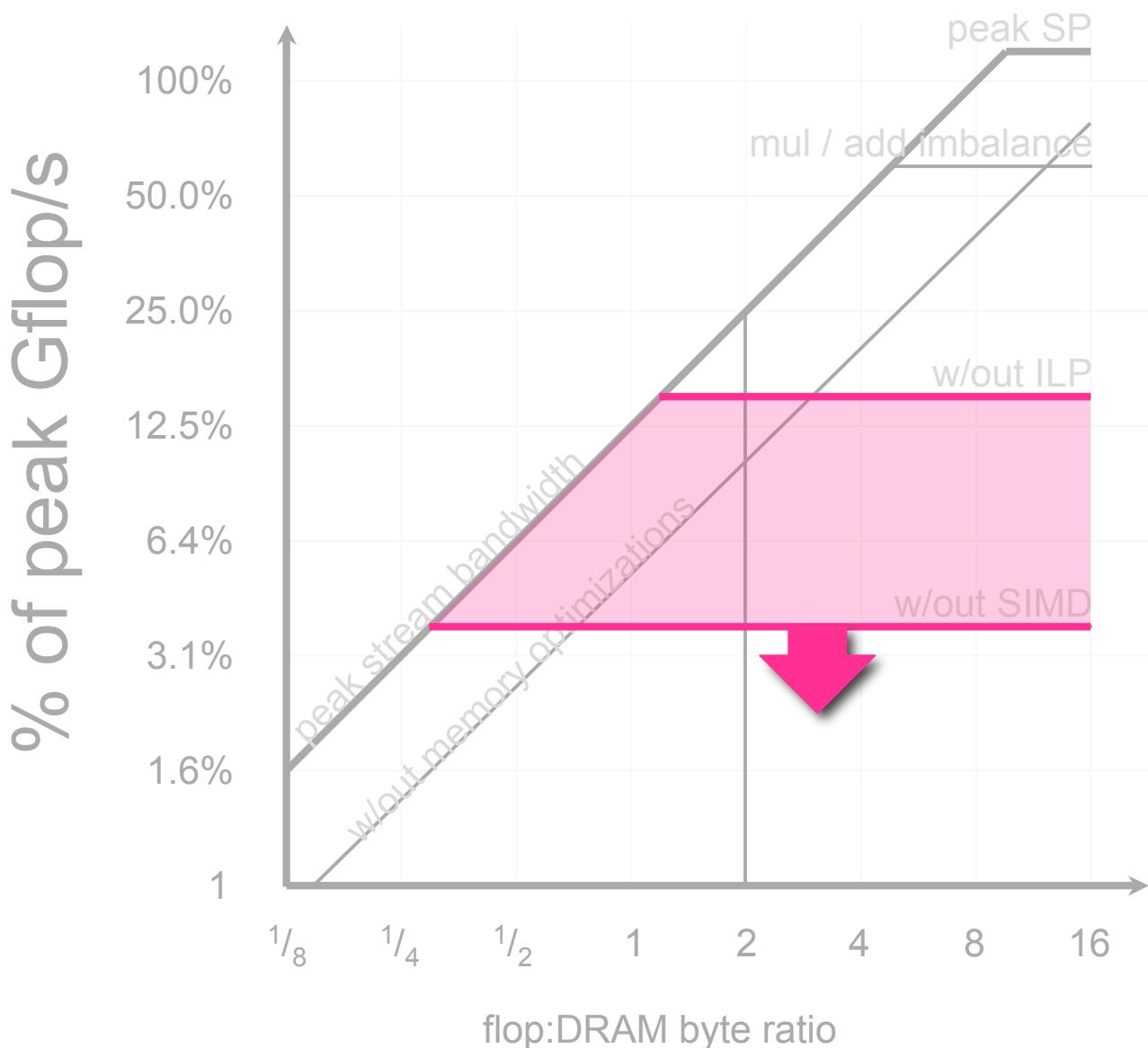
Conclusions

- ❖ The Roofline Model provides an intuitive graph for kernel analysis and optimization
- ❖ Easily extendable to other architectural paradigms
- ❖ Easily extendable to other communication or computation metrics

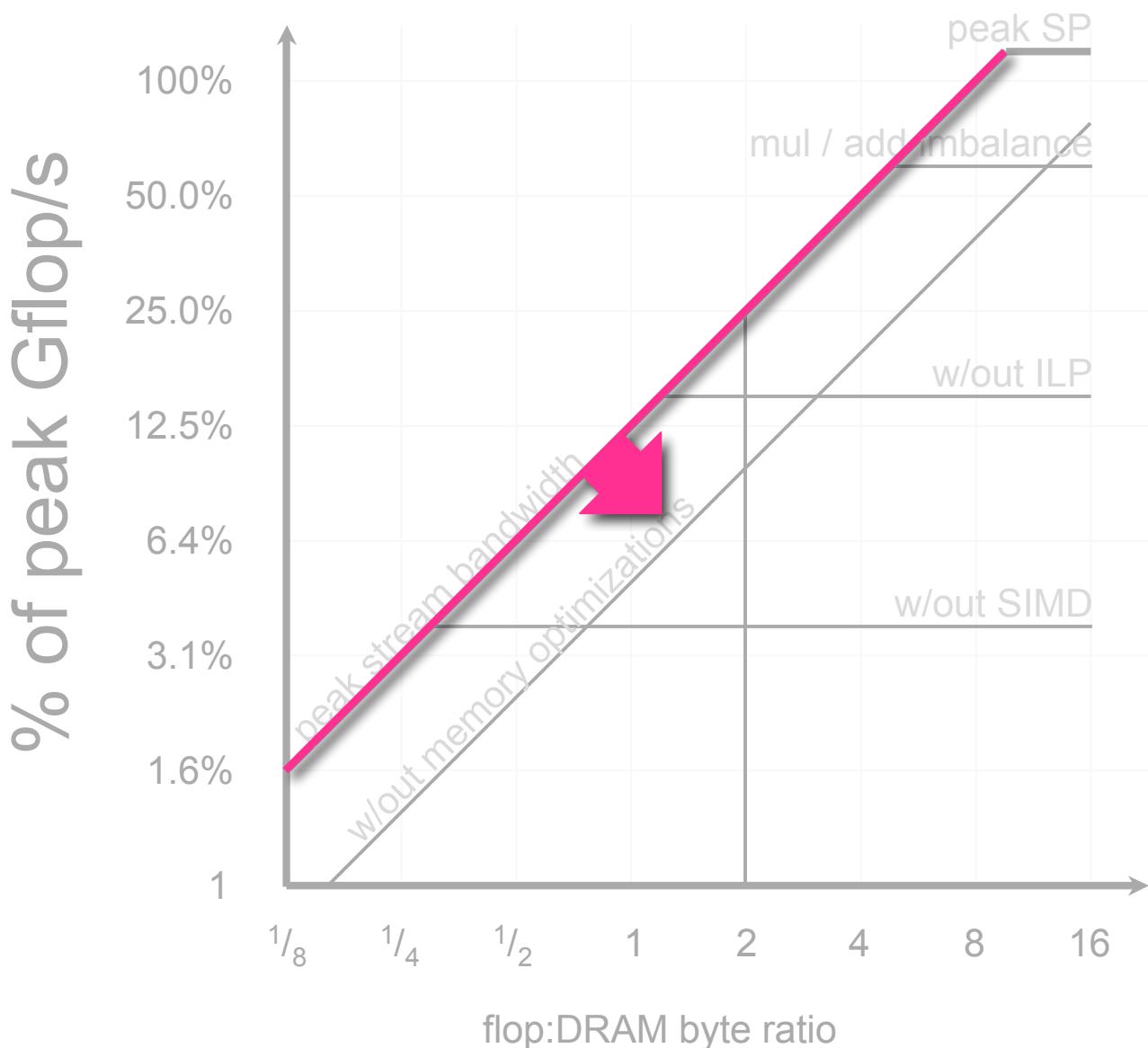
Questions ?

BACKUP SLIDES

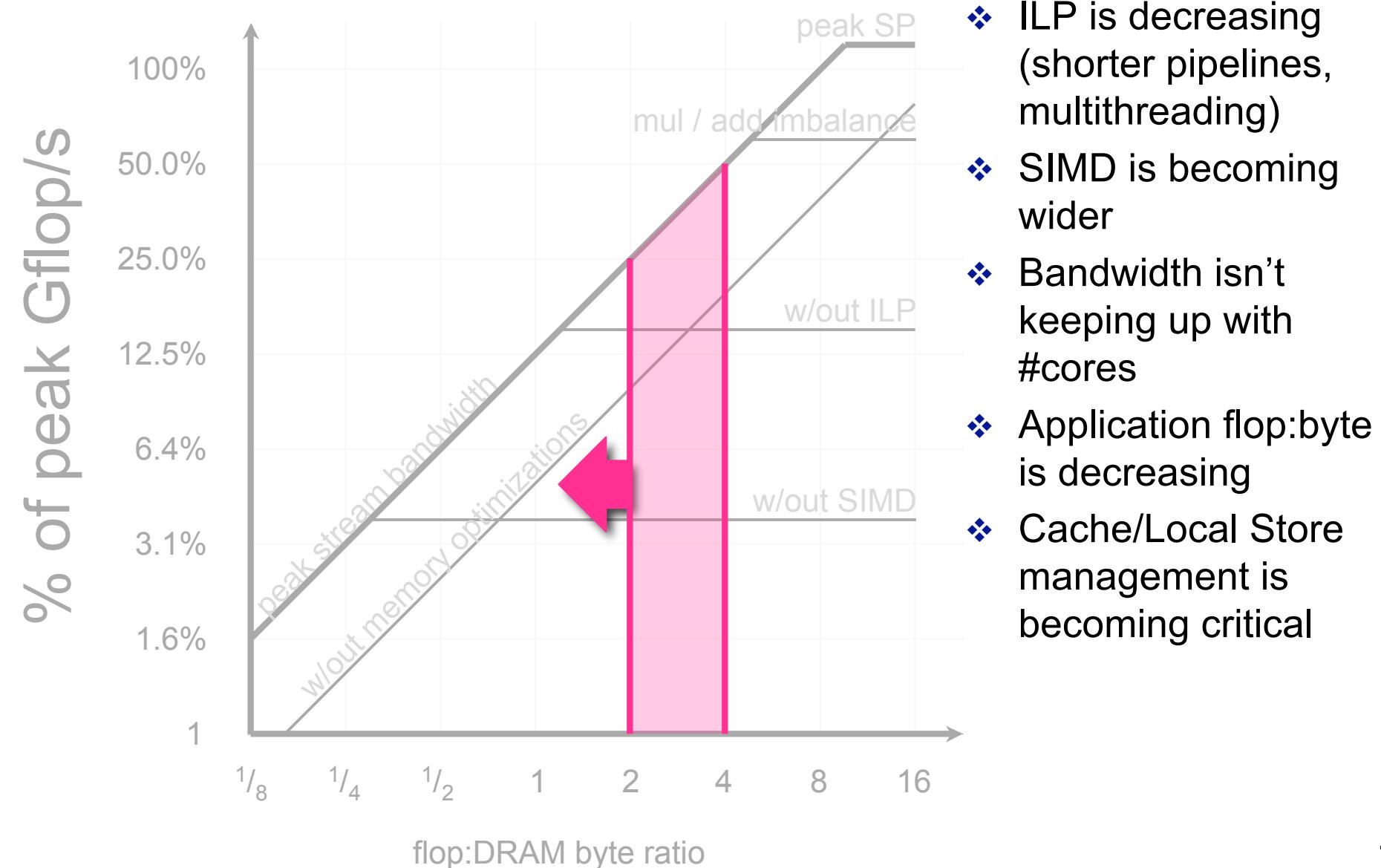


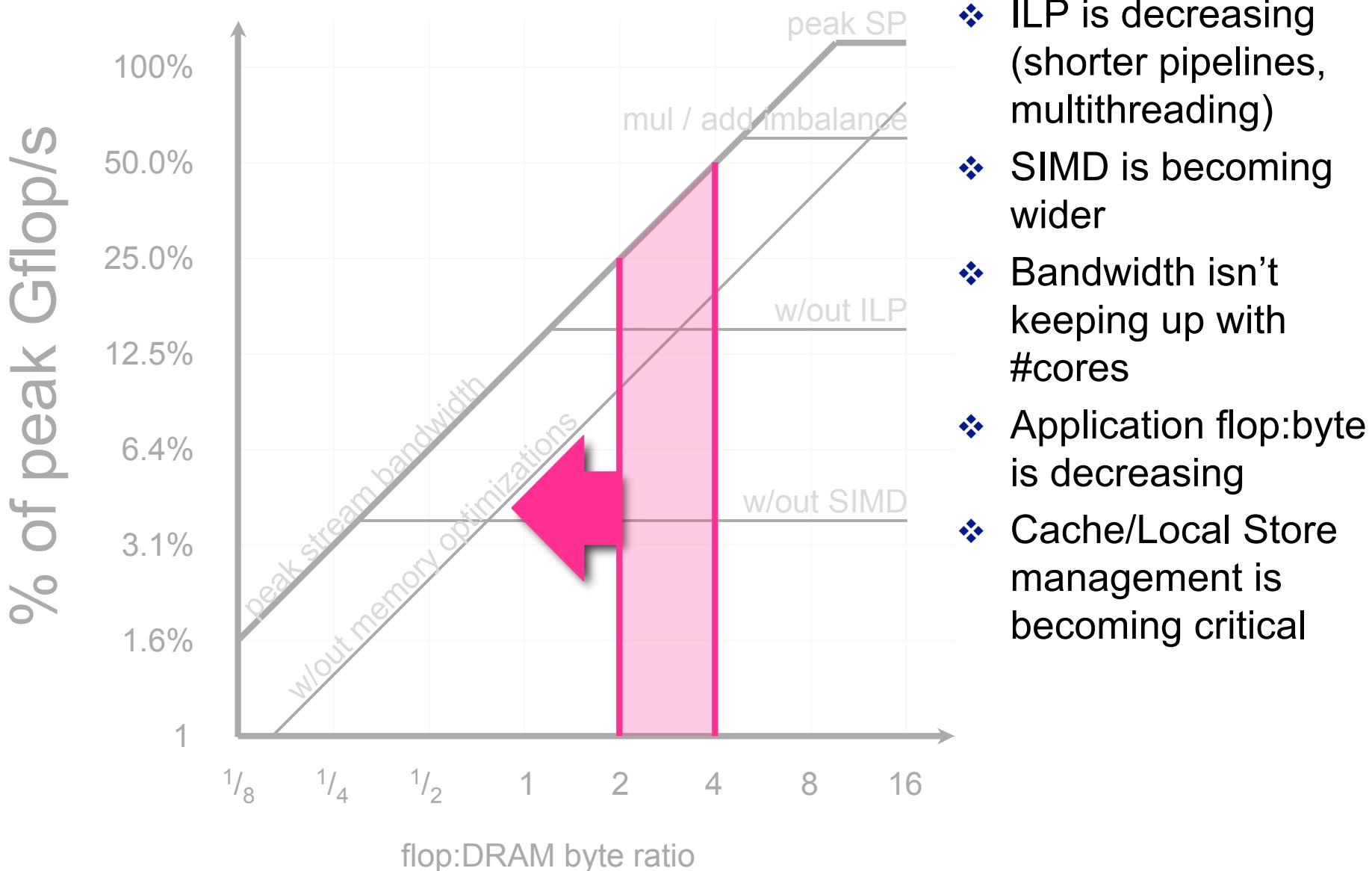


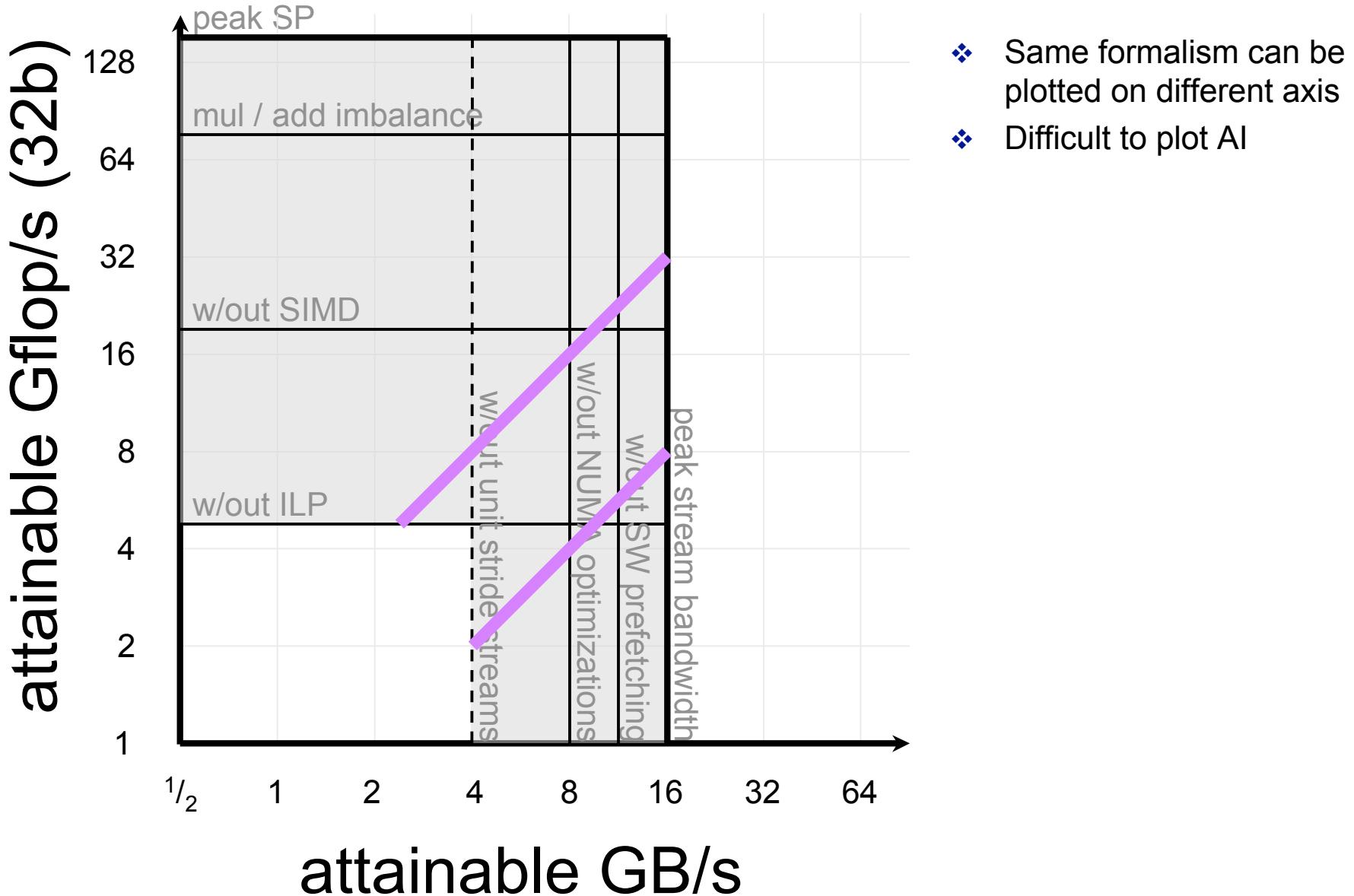
- ❖ ILP is decreasing (shorter pipelines, multithreading)
- ❖ SIMD is becoming wider
- ❖ Bandwidth isn't keeping up with #cores
- ❖ Application flop:byte is decreasing
- ❖ Cache/Local Store management is becoming critical



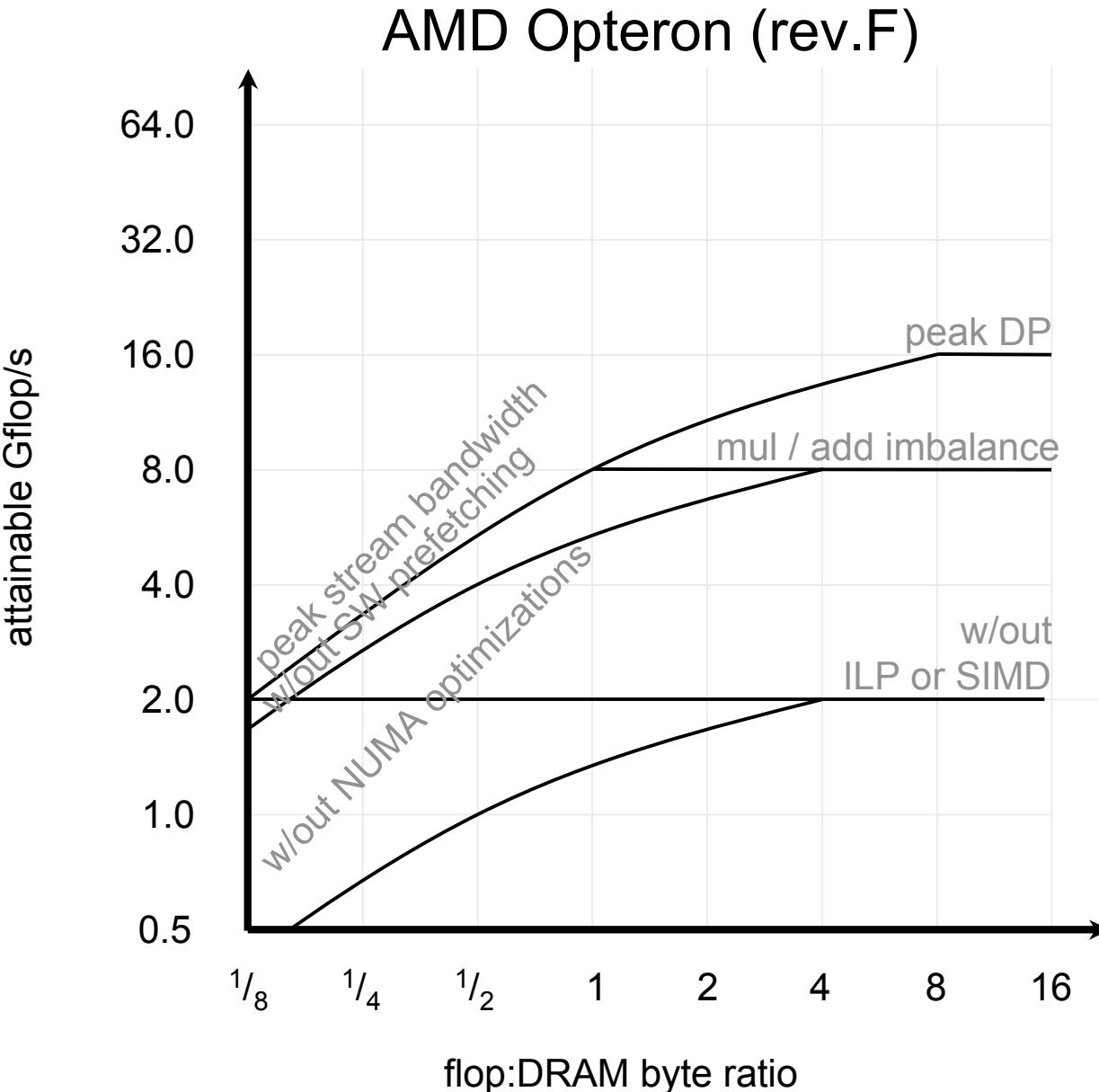
- ❖ ILP is decreasing (shorter pipelines, multithreading)
- ❖ SIMD is becoming wider
- ❖ Bandwidth isn't keeping up with #cores
- ❖ Application flop:byte is decreasing
- ❖ Cache/Local Store management is becoming critical





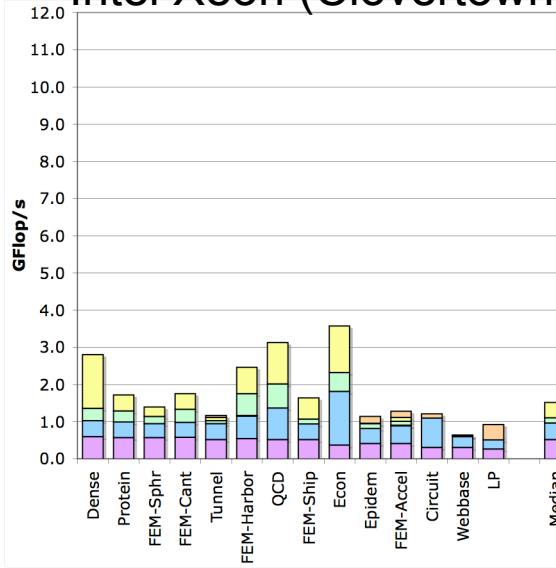


- ❖ What if computation or communication isn't totally overlapped
- ❖ At ridgepoint 50% of the time is spent in each, so performance is cut in half
- ❖ In effect, the curves are smoothed
- ❖ Common for bulk synchronous MPI communication, atypical for DRAM access on modern architectures

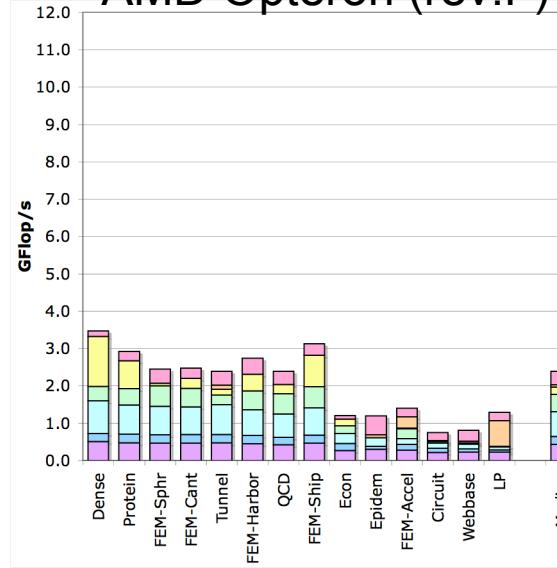


- ❖ Not typical of multi-thread/core architectures
- ❖ Not typical of architectures with ooo or HW prefetchers
- ❖ More common in network accesses

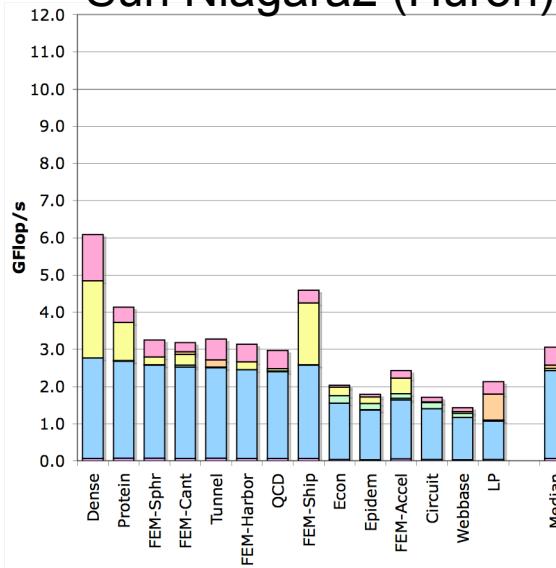
Intel Xeon (Clovertown)



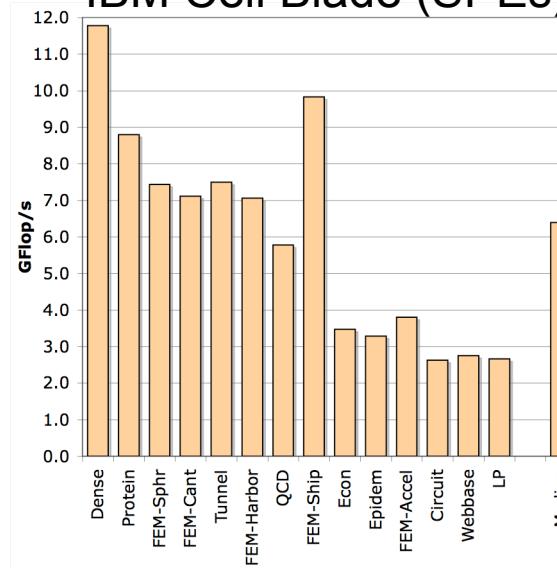
AMD Opteron (rev.F)



Sun Niagara2 (Huron)



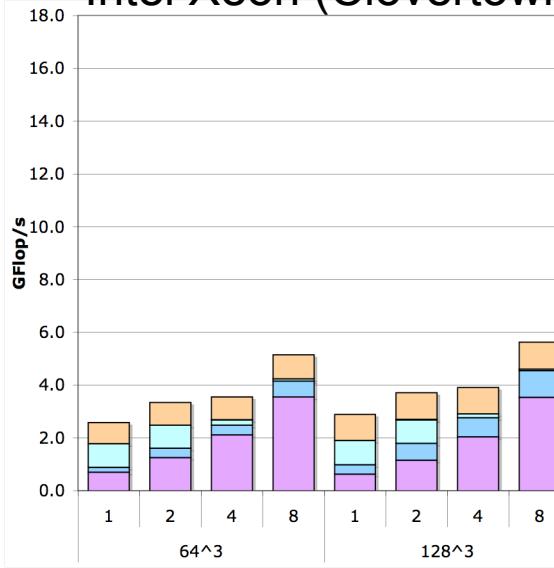
IBM Cell Blade (SPEs)



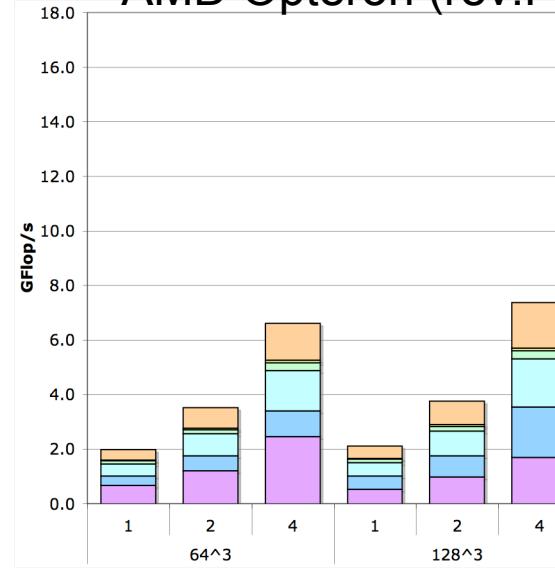
- ❖ Wrote a double precision Cell/SPE version
- ❖ DMA, local store blocked, NUMA aware, etc...
- ❖ Only 2x1 and larger BCOO
- ❖ Only the SpMV-proper routine changed
- ❖ About 12x faster (median) than using the PPEs alone.

- +More DIMMs (Opteron), FW fix, array padding (N2), etc...
- +Cache/TLB Blocking
- +Compression
- +SW Prefetching
- +NUMA/Affinity
- Naïve Pthreads
- Naïve

Intel Xeon (Clovertown)

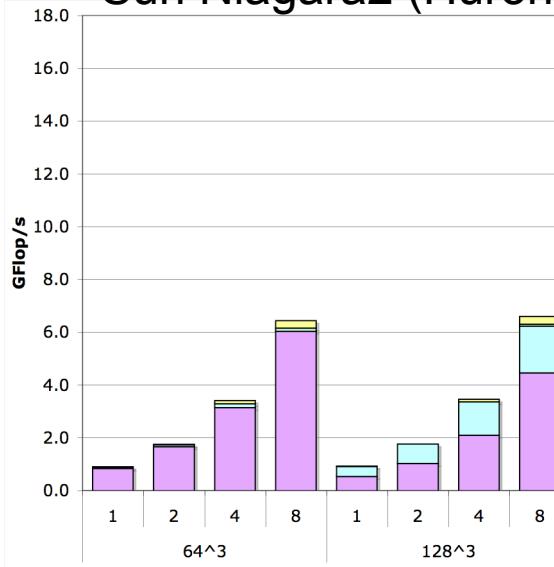


AMD Opteron (rev.F)

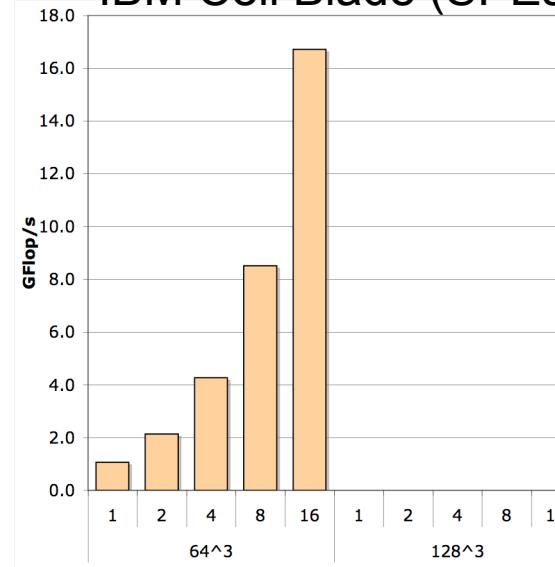


- ❖ First attempt at cell implementation.
- ❖ VL, unrolling, reordering fixed
- ❖ No NUMA
- ❖ Exploits DMA and double buffering to load vectors
- ❖ Straight to SIMD intrinsics.
- ❖ Despite the relative performance, Cell's DP implementation severely impairs performance

Sun Niagara2 (Huron)



IBM Cell Blade (SPEs)*



- +SIMDization
- +SW Prefetching
- +Unrolling
- +Vectorization
- +Padding
- Naïve+NUMA

*collision() only

Where do GPUs fit in?

Expressed at compile time	Discovered at run time
Instruction Level Parallelism	VLIW superscalar, SMT, etc...
Data Level Parallelism	SIMD, Vector G80

- ❖ GPUs discover data level parallelism from thread level parallelism at runtime