Parallel Computer Architecture
Fall 2018

Compile-time ILP extraction
Superblocks, Predication, Speculation, Hyperblocks
Case study: Intel's Itanium

Nikos Bellas

Computer and Communications Engineering Department
University of Thessaly

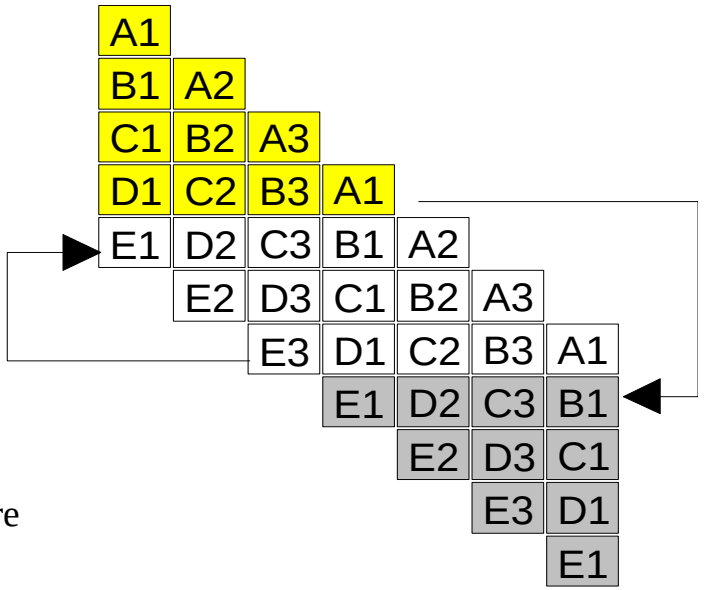
Readings for this lecture

- **Hardware and Software for VLIW and EPIC**
  - H&P, Appendix G in v.4
  - H&P, Sections 4.5-4.7 in v.3

- **Scheduling techniques**
  - W. W. Hwu et al, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", The Journal of Supercomputing, 1993
  - S. Mahlke et al, “Effective Compiler Support for Predicated Execution Using the Hyperblock,” MICRO 24, December 1992.
  - D. August et al, “Integrated Predicated and Speculative Execution in the IMPACT EPIC architecture”, ISCA, June 1998

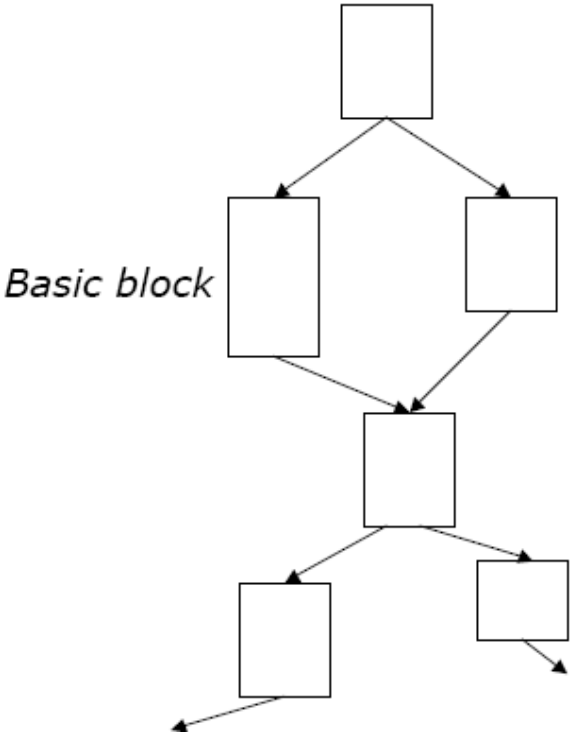
Compiler based ILP extraction

- We examined methods to extract parallelism using dynamic and compiler-based scheduling
- Dynamic scheduling
  - Tomasulo algorithm
  - Precise Exceptions using Reorder Buffers
- Compiler-based scheduling for loops
  - Basic block scheduling
  - Loop unrolling
  - Software pipelining
  - Modulo scheduling



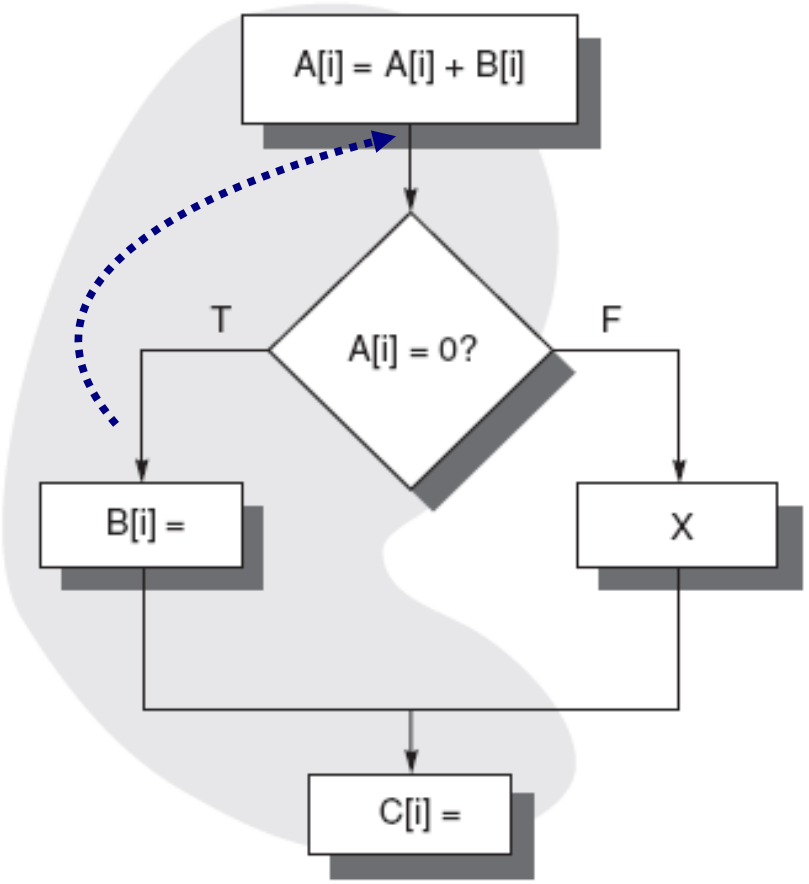
Compiler based ILP extraction

- Modulo scheduling can be applied when the loop is only a basic block
  - No *if-then-else* statements inside the loop
- What about less regular code?



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

Global scheduling



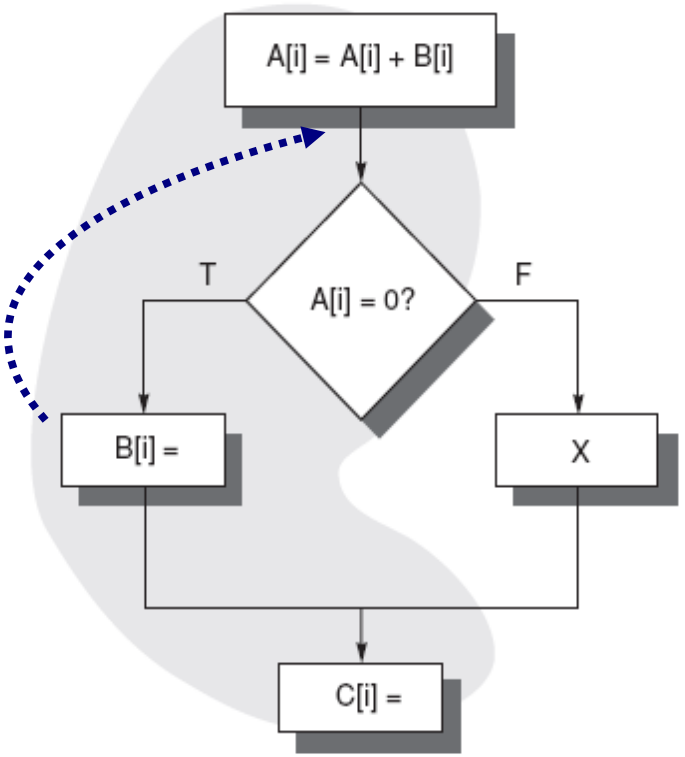
What every scheduler wants is “large, frequently executed basic blocks”

Global scheduling techniques attempt to schedule beyond branches

Move the B[i] and/or C[i] assignments before the conditional to improve the schedule

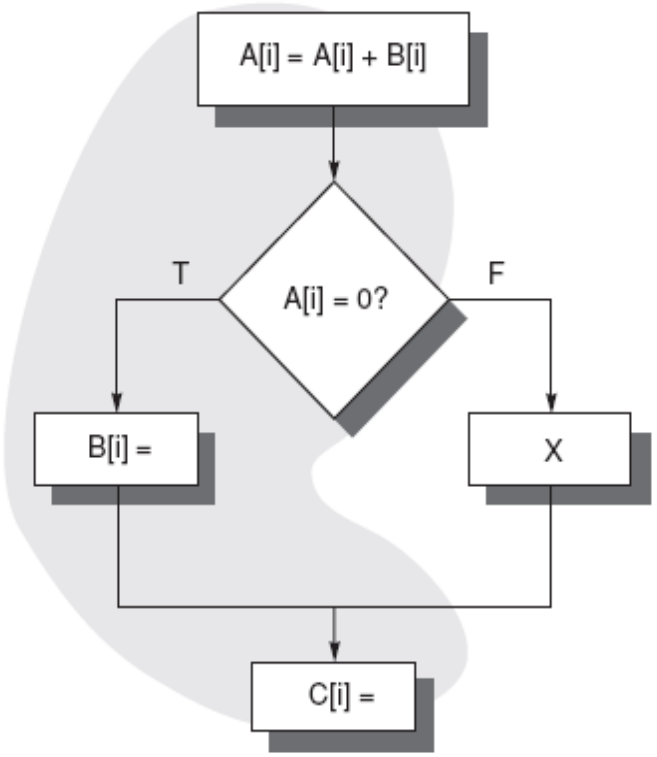
The B[i]=... path is the most frequently executed (after profiling)

Global scheduling



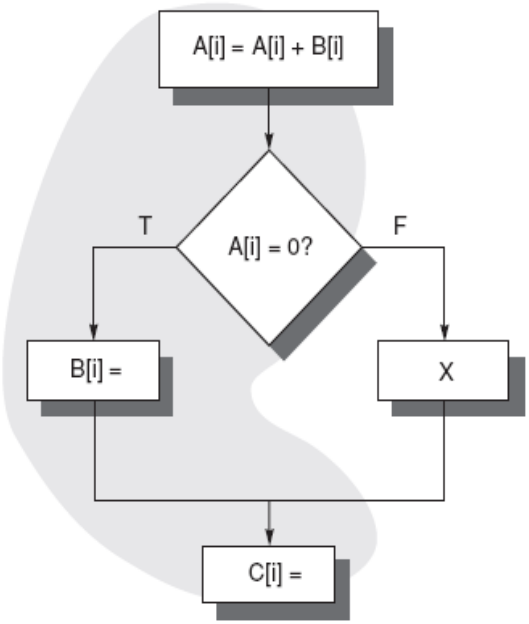
- Such global motion needs to satisfy some constraints to be legal
- **Data flow** and **exception behavior** should remain the same
- For example, we can only move the B[i] assignment before the conditional and commit the result to the memory if:
  - (The conditional proves TRUE and the B[i] is not live before its old assignment OR the conditional proves FALSE and B[i] is not live in path X), AND
  - in case of an exception from B[i] in its new position, we should delay raising the exception after the conditional proves true

Superblocks

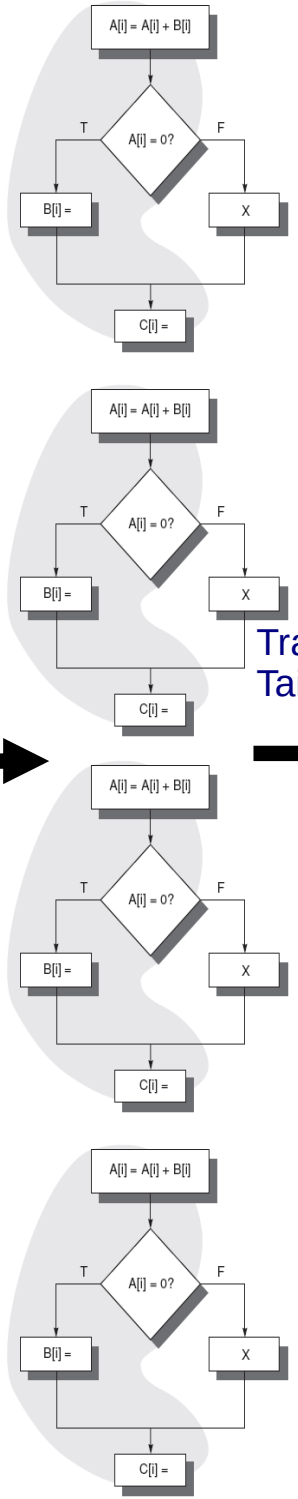


- A compiler technique to increase the effective length of a basic block
- A *basic block (BB)* is a sequence of instructions with a single entry and a single exit
- A *superblock* is a sequence of instructions with a single entry and one or more exits
  - It is a form of extended basic block
- First, loop unrolling if within a loop
- Form a trace of the most frequently executed path of basic blocks (based on profiling)
  - Start from the most freq. executed instruction *s*
  - Grow the path above and below *s*
  - Finally, form a list *L* of instructions (trace) across multiple BBs

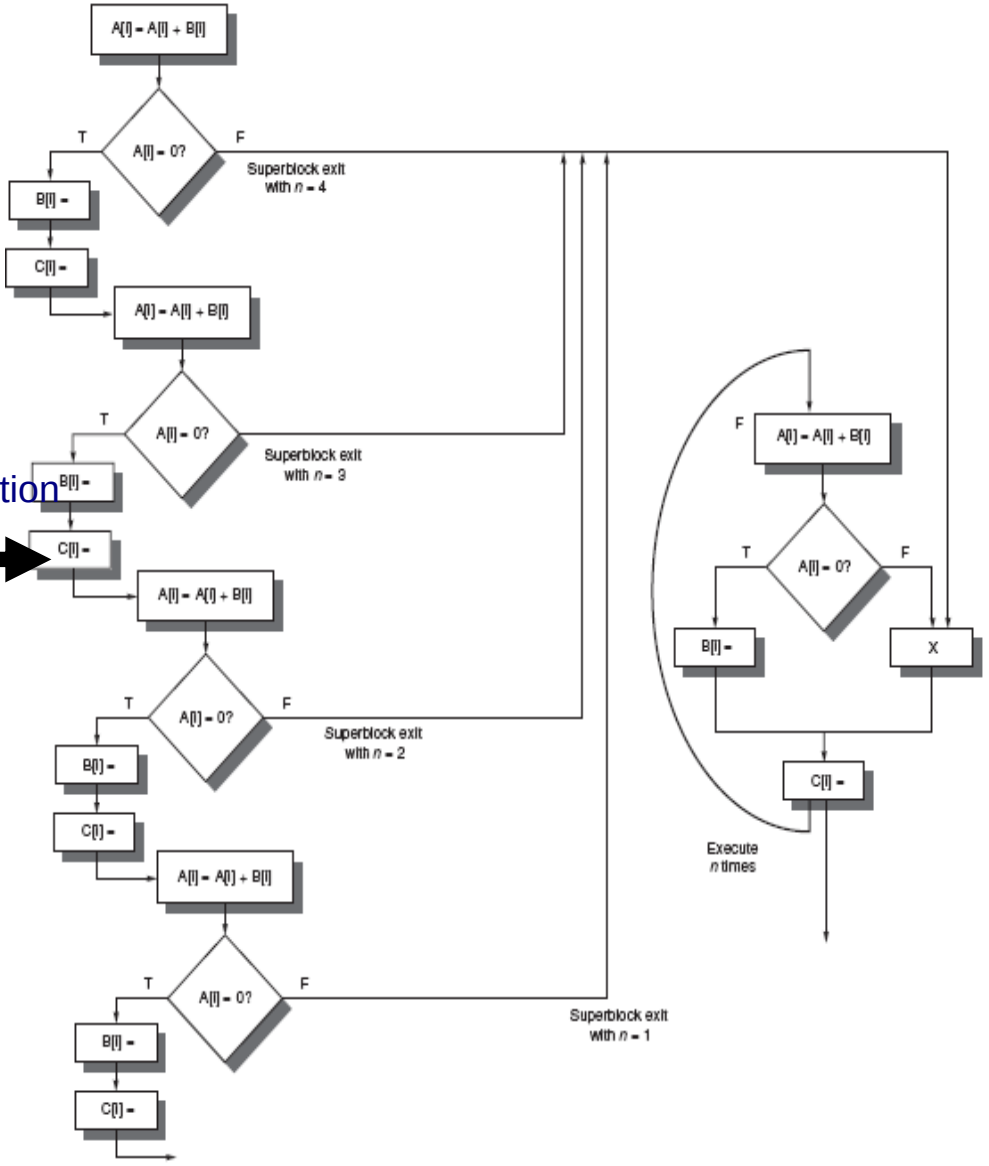
Superblocks



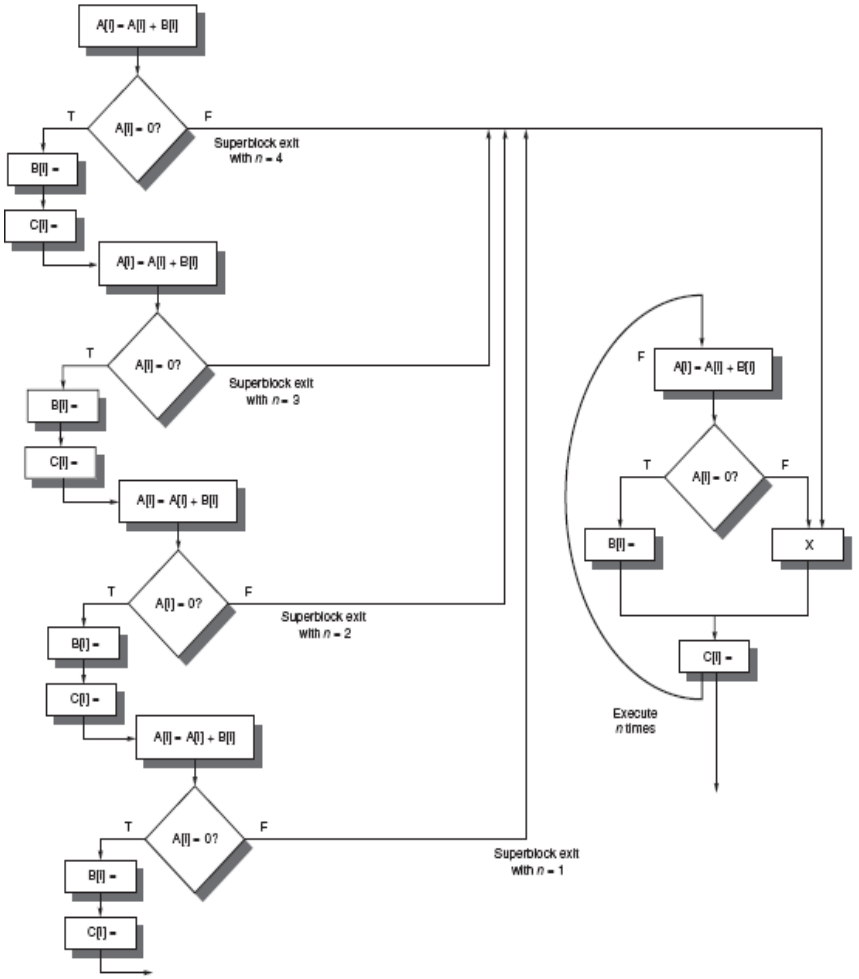
Unroll 4x



Trace & Tail duplication



Superblocks



- Tail duplication to accommodate early exits
- Superblock works best when these exits are not frequently taken
- Superblocks work best if combined with speculation and predication

Predication

- Predication replaces control dependencies with data dependencies in a program
- Allows conditional execution of instructions based on a Boolean source operand
- Provides compiler with an alternative to guarding instruction execution with branches
- Example : if (A==0) {S=T}
  - BNEZ R1, L
    MOV R2, R3

  p1 = (R1 == 0)

  MOV R2, R3, <p1>

L: ....

Predication – Execution Model

- Additional Boolean operand
  - LD R1, 0(R2) <p1>
  - If p1 is TRUE, instruction executes normally
  - If p1 is FALSE, instruction treated as NOP

Full predication support

- For full predication support, each instruction is guarded by a predicate
- Full set of predicated instructions for logic manipulation of predicates
- Separate predicate register file
- Best performance
- ARM , Cydra-5, IA-64, TI-C60, StarCore (Motorola+Alcatel DSP processor)

Predication example in an EPIC machine

Explicitly Parallel Instruction Computing

Wide issue processor
Latencies:
Memory Loads : 2 cycles
Integer Ops : 1 cycle

```
if ((*ptr1 == 0) && ((*ptr2 = *ptr3) == 1) && (*ptr4 > 2))
        val5++;
 else
        val6++;
```

C source code

0	(1) r11 = MEM[r1]	
1		
2	(2) c1 = (r11 != 0)	
3		(3) jump c1, ELSE
4	(4) r13 = MEM[r3]	
5		
6	(5) MEM[r2] = r13	(6) c2 = (r13 != 1)
7		(7) jump c2, ELSE
8	(8) r14 = MEM[r4]	
9		
10	(9) c3 = (r14 <= 2)	
11		(10) jump c3, ELSE
12	(11) r5 = r5 + 1	(12) jump CONTINUE

ELSE:

0	(13) r6 = r6 + 1	

CONTINUE:

Initial schedule

Predication example in an EPIC machine

0	(1) r11 = MEM[r1]	
1		
2	(2) c1 = (r11 != 0)	
3		(3) jump c1, ELSE
4	(4) r13 = MEM[r3]	
5		
6	(5) MEM[r2] = r13	(6) c2 = (r13 != 1)
7		(7) jump c2, ELSE
8	(8) r14 = MEM[r4]	
9		
10	(9) c3 = (r14 <= 2)	
11		(10) jump c3, ELSE
12	(11) r5 = r5 + 1	(12) jump CONTINUE

ELSE:

0	(13) r6 = r6 + 1	

CONTINUE:

Initial schedule

0	(1) r11 = MEM[r1]	(14) p4 = 0
1	(15) p2 = 1	(16) p3 = 1
2	(2) p4of, p1ut = (r11 == 0)	(2') p2at, p3at = (r11 == 0)
3	(4) r13 = MEM[r3] <p1>	
4		
5	(5) MEM[r2] = r13 <p1>	(6) p4of, p2at = (r13 == 1)
6	(8) r14 = MEM[r4] <p2>	(6') p3at = (r13 == 1)
7		
8	(9) p4of, p3at = (r14 > 2)	
9	(11) r5 = r5 + 1 <p3>	(13) r6 = r6 + 1 <p4>

Predicated schedule

- This process is called *if-conversion*
- One predicate for each possible execution path

Predication example in an EPIC machine

```
if ((*ptr1 == 0) && ((*ptr2 = *ptr3) == 1) && (*ptr4 > 2))
        val5++;
 else
        val6++;
```

C source code

0	(1) r11 = MEM[r1]	(14) p4 = 0
1	(15) p2 = 1	(16) p3 = 1
2	(2) p4of, p1ut = (r11 == 0)	(2') p2at, p3at = (r11 == 0)
3	(4) r13 = MEM[r3] <p1>	
4		
5	(5) MEM[r2] = r13 <p1>	(6) p4of, p2at = (r13 == 1)
6	(8) r14 = MEM[r4] <p2>	(6') p3at = (r13 == 1)
7		
8	(9) p4of, p3at = (r14 > 2)	
9	(11) r5 = r5 + 1 <p3>	(13) r6 = r6 + 1 <p4>

Predicated schedule

Unconditional True

p1ut = *cond* means:

```
if (cond)
      p1 = TRUE;
else
      p1 = FALSE
```

Or-False

p1of = (*cond*) means:

```
if (cond)
    p1 = unchanged;
else
    p1 = TRUE;
```

It requires initialization to FALSE

And-True

p1at = (*cond*) means:

```
if (cond)
    p1 = unchanged;
else
    p1 = FALSE;
```

It requires initialization to TRUE

Predication defines

- OR-type
  - For blocks reached on multiple disjunctive conditions (C1 | C2..)
- AND-type
  - For blocks reached on conjunctive conditions : (C1 & C2 …)

Predication example in an EPIC machine

- Predication advantages
  - All control dependencies become data dependencies on predicates – Straight line code
  - No pipeline flushes due to branch instructions
  - Instructions from different paths are scheduled simultaneously (see cycle 9) – ILP increases
  - Advantageous for short sequences
- Predication disadvantages
  - Instructions from different paths are scheduled simultaneously – More instructions executed, resources and fetch bandwidth wasted
  - More complex hardware needed to annul an instruction when the predicate is FALSE
  - The additional data hazard may actually be worse than the conditional hazard if the branch is predicted correctly and the predicate is predicted late.

Speculation

- Speculate on the result of a run-time event at compile time
- Speculation refers to the ability of the compiler to:
  - Find instructions that can be speculatively moved to a different place w/o affecting the program data flow (Control Speculation)
  - Execute an instruction before knowing input is up to date. For example, interchange load/store or store/store instructions (Data Speculation)
  - Ignore exceptions in speculated instructions, until we know that these exceptions should really occur

Speculation example in an EPIC machine

```
if ((*ptr1 == 0) && ((*ptr2 = *ptr3) == 1) && (*ptr4 > 2))
        val5++;
 else
        val6++;
```

C source code

0	(1) r11 = MEM[r1]	
1		
2	(2) c1 = (r11 != 0)	
3		(3) jump c1, ELSE
4	(4) r13 = MEM[r3]	
5		
6	(5) MEM[r2] = r13	(6) c2 = (r13 != 1)
7		(7) jump c2, ELSE
8	(8) r14 = MEM[r4]	
9		
10	(9) c3 = (r14 <= 2)	
11		(10) jump c3, ELSE
12	(11) r5 = r5 + 1	(12) jump CONTINUE

ELSE:

0	(13) r6 = r6 + 1	

CONTINUE:

Initial schedule

Speculation example in an EPIC machine

```
if ((*ptr1 == 0) && ((*ptr2 = *ptr3) == 1) && (*ptr4 > 2))
        val5++;
 else
        val6++;
```

0	(1) r11 = MEM[r1]	
1		
2	(2) c1 = (r11 != 0)	
3		(3) jump c1, ELSE
4	(4) r13 = MEM[r3]	
5		
6	(5) MEM[r2] = r13	(6) c2 = (r13 != 1)
7		(7) jump c2, ELSE
8	(8) r14 = MEM[r4]	
9		
10	(9) c3 = (r14 <= 2)	
11		(10) jump c3, ELSE
12	(11) r5 = r5 + 1	(12) jump CONTINUE

ELSE:

0	(13) r6 = r6 + 1	

CONTINUE:

Initial schedule

0	(1) r11 = MEM[r1]	(4) r13 = MEM[r3]<CS>	(8) r14 = MEM[r4]<CS,DS>
1			
2	(2) c1 = (r11 != 0)	(6) c2 = (r13 != 1)<CS>	(9) c3 = (r14 <= 2)<CS>
3			(3) jump c1, ELSE
4	(4') Check r13	(5) MEM[r2] = r13	(7) jump c2, ELSE
5	(8') Check r14		(10) jump c3, ELSE
6	(11) r5 = r5 + 1		(12) jump CONTINUE

ELSE:

0	(13) r6 = r6 + 1		

CONTINUE:

Schedule with Control & Data Speculation

Speculation

0	(1) r11 = MEM[r1]	(4) r13 = MEM[r3]<CS>	(8) r14 = MEM[r4]<CS,DS>
1			
2	(2) c1 = (r11 != 0)	(6) c2 = (r13 != 1)<CS>	(9) c3 = (r14 <= 2)<CS>
3			(3) jump c1, ELSE
4	(4') Check r13	(5) MEM[r2] = r13	(7) jump c2, ELSE
5	(8') Check r14		(10) jump c3, ELSE
6	(11) r5 = r5 + 1		(12) jump CONTINUE

ELSE:

0	(13) r6 = r6 + 1		

CONTINUE:

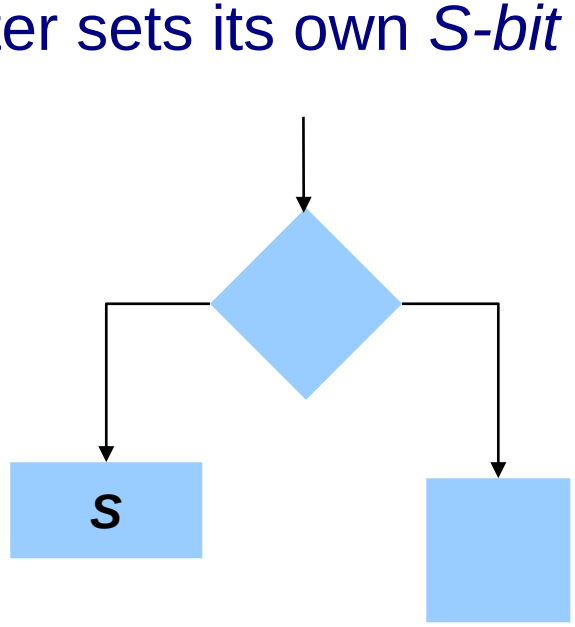
- Load instructions (4) and (8) are hoisted at the beginning of the schedule. Instr. (4) is Control dependent on (3), and (8) on (7)
- Instruction 8 is data dependent on (5) because the store location MEM[r2] may alias with MEM[r4]
- Two check instructions are used to confirm or not if (4) and (8) should commit their results and their exceptions if they have raised any.

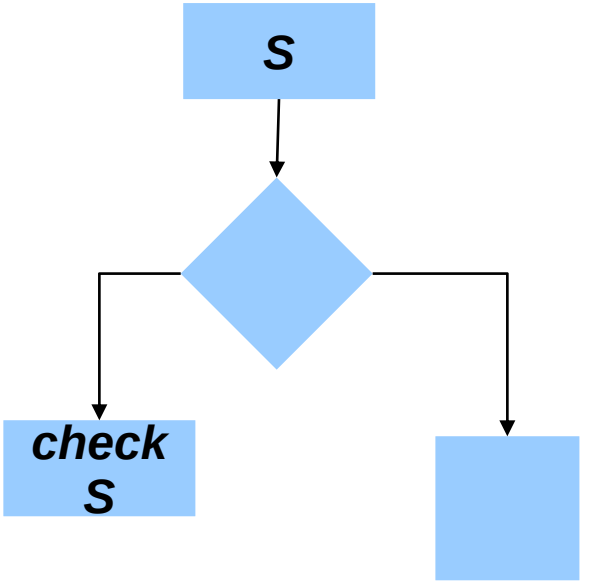
Speculation

- Speculation advantages
  - Increase available ILP by reducing the height of long dependence chains and executing speculatively instructions
  - Enable aggressive data prefetching by moving load instructions upwards the instruction stream (Data Speculation)
- Speculation disadvantages
  - More complex hardware and/or compiler techniques needed to annul an instruction when the speculation is FALSE
  - Speculation may result into worse performance, because more instructions are executed

Control speculation management

- Control speculation management: *check* instructions and *S-bits* (one per register)
- Each register of the processor has en extra bit (S-bit)
- The S-bit is set for the register to which a control speculated instruction *S* writes.
  - For r13 and r14 : r13 = MEM[r3] <CS> and r14 = MEM[r4] <CS, DS>
- Each subsequent speculated instruction that uses the tagged register sets its own *S-bit*



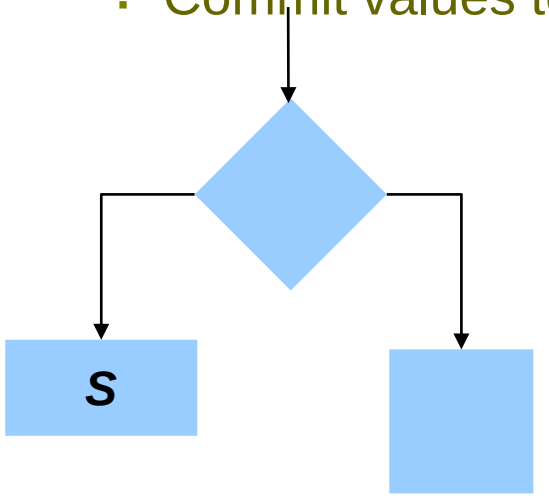


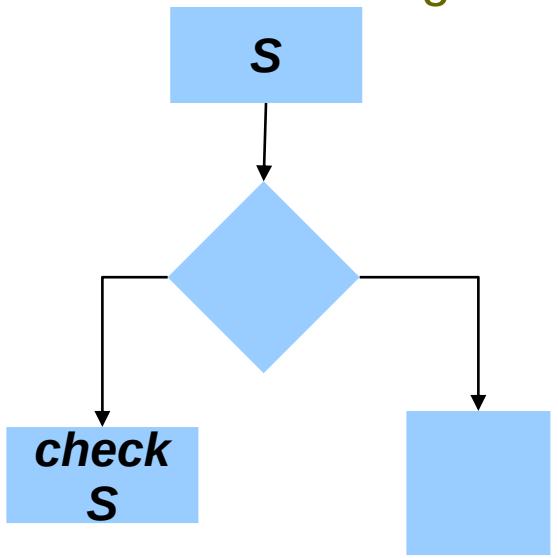
Control speculation management

- An extra bit (E-bits) is placed in each register *r* to denote that the speculated instruction that wrote to *r* caused an exception
- Each subsequent speculated instruction that uses the tagged register, sets its E-bit
  - Tag propagation
- The value of the *E-bit* is finally read by the *check* instr. and an exception is raised if the tag is set

Control speculation management

- Control speculation management: *check* instructions *S-bits* and *E-bits*
- The explicit *check* instructions (e.g. *check r13)* are placed by the compiler as follows:
  - there is a *check* instruction at the original spot of each control speculative instruction *S*
    - If the *check* instruction executes, it means that *S* would have been executed, and the control speculation is correct.
    - It also means that a potential exception from *S* can now be raised
    - Commit values to register and erase S and E bits from all registers





Control speculation management

- Checks are not required when the compiler can prove that the speculative instruction:
  - will never raise an exception (e.g. an integer add) AND
  - will never produce undesirable side-effects

Data Speculation

Consider the load/store dependencies

MEM[r2] =r13

r14 = MEM[r4]

Can we move the *load* before the *store* (Data Speculation)?

- Always access the same location
  - Compiler eliminates redundant load with an assignment r14=r13
- Never access the same location
  - load/store reordering is allowed
- May access the same location
  - They sometimes access the same location
  - No conclusion can be reached
  - How often do they access the same location?

Data speculation management Memory Conflict Buffer (MCB)

- The MCB checks speculated loads against stores
- The result of a speculated load is stored in the MCB, not in the register itself
  - It should not modify the processor state yet
  - E.g. one entry for r14
- Destination addresses of subsequent stores are checked against the addresses in the buffer
- Initiate a recovery routine in case there is a match

Both Predication & Speculation

0	(1) r11 = MEM[r1]	(14) p4 = 0
1	(15) p2 = 1	(16) p3 = 1
2	(2) p4of, p1ut = (r11 == 0)	(2') p2at, p3at = (r11 == 0)
3	(4) r13 = MEM[r3] <p1>	
4		
5	(5) MEM[r2] = r13 <p1>	(6) p4of, p2at = (r13 == 1)
6	(8) r14 = MEM[r4] <p2>	(6') p3at = (r13 == 1)
7		
8	(9) p4of, p3at = (r14 > 2)	
9	(11) r5 = r5 + 1 <p3>	(13) r6 = r6 + 1 <p4>

Schedule with Predication only

(1) r11 = MEM[r1]	(4) r13 = MEM[r3]<CS>	(8) r14 = MEM[r4]<CS,DS>
(2) c1 = (r11 != 0)	(6) c2 = (r13 != 1)<CS>	(9) c3 = (r14 <= 2)<CS>
		(3) jump c1, ELSE
(4') Check r13	(5) MEM[r2] = r13	(7) jump c2, ELSE
(8') Check r14		(10) jump c3, ELSE
(11) r5 = r5 + 1		(12) jump CONTINUE

ELSE:

(13) r6 = r6 + 1		

CONTINUE:

Schedule with Control & Data Speculation Only

0	(1) r11 = MEM[r1]	(4) r13 = MEM[r3]<CS>	(8) r14 = MEM[r4]<CS,DS>		
1	(14) p4 = 0	(15) p2 = 1	(16) p3 = 1		
2	(2) p4of, p1ut = (r11 == 0)	(2') p2at, p3at = (r11 == 0)	(6) p4of, p2at = (r13 == 1)<CS>	(6') p3at = (r13 == 1)<CS>	(9) p4of, p3at = (r14 > 2)<CS>
3	(4') Check r13 <p1>	(5) MEM[r2] = r13 <p1>	(8') Check r14 <p2>	(11) r5 = r5 + 1 <p3>	(13) r6 = r6 + 1 <p4>

Predication and Speculation combined

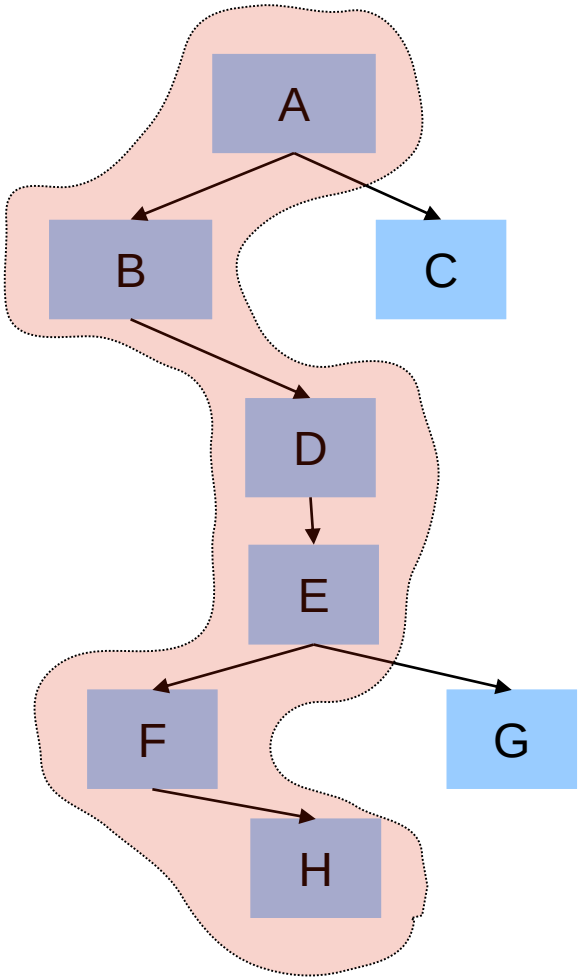
Both Predication & Speculation

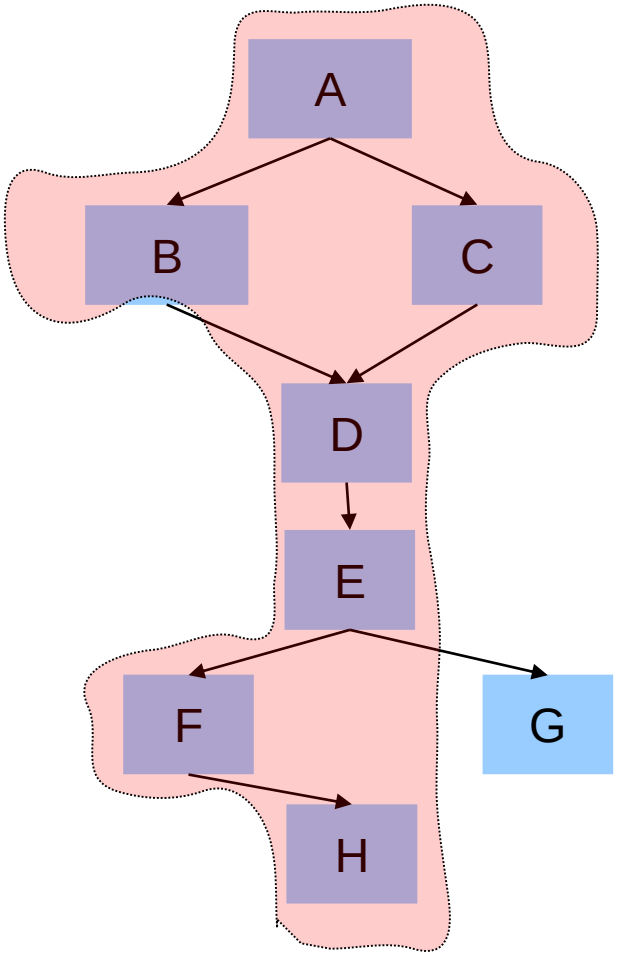
- Remarks:
  - Initial schedule was 5 cycles (min), 10.25 cycles (average), 13 cycles (max)
  - Predicated/Speculated schedule is always 4 cycles
  - Number of instructions executed was 4 (min), 9 (average), 12 (max)
  - Predicated/speculated code always executes 23 instructions
- These techniques are complementary
  - Predication converts control flow to data flow dependencies (dependencies on predicate variables)
  - Speculation is used to execute dependent instructions before we know that they should be executed
  - Hardware enhancements are vital for functionally correct execution
  - More instructions executed than actually needed -> resource pressure

Hyperblocks

- Like superblocks, but the code can be from multiple paths of control
  - Better for **NOT** heavily biased branches
- Set of connected basic blocks
  - One entry block dominates all others
  - Control flow may enter hyperblock only at entry block
  - Control flow may exit hyperblock at any number of locations
  - No nested inner loop exists in the selected blocks (since only one entrance is allowed)
  - Each instruction executes at most once inside the hyperblock

Superblocks VS Hyperblocks





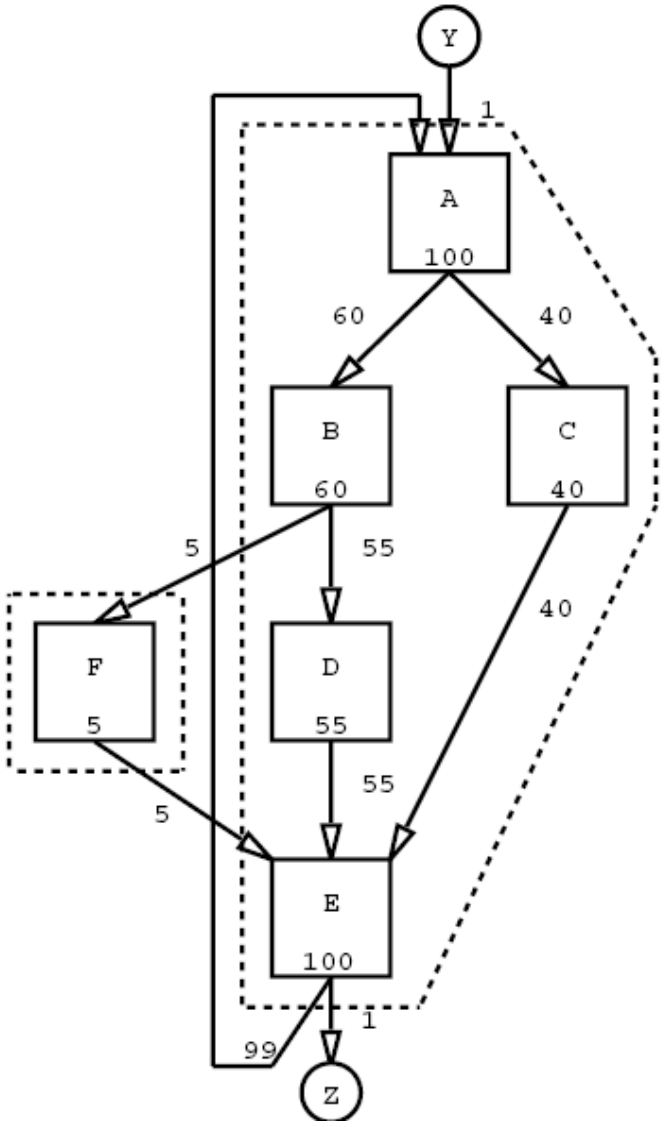
Superblocks only select one of many potential paths

Hyperblocks may select multiple paths

Compilation based on Hyperblocks

- Hyperblock region formation
  - Basic block selection
  - Tail duplication
  - Loop peeling
- Predication (if-conversion)
- Optimizing predicated code
- Speculation in predicated code
- Scheduling speculated-predicated code
  - Modulo scheduling
- Code generation

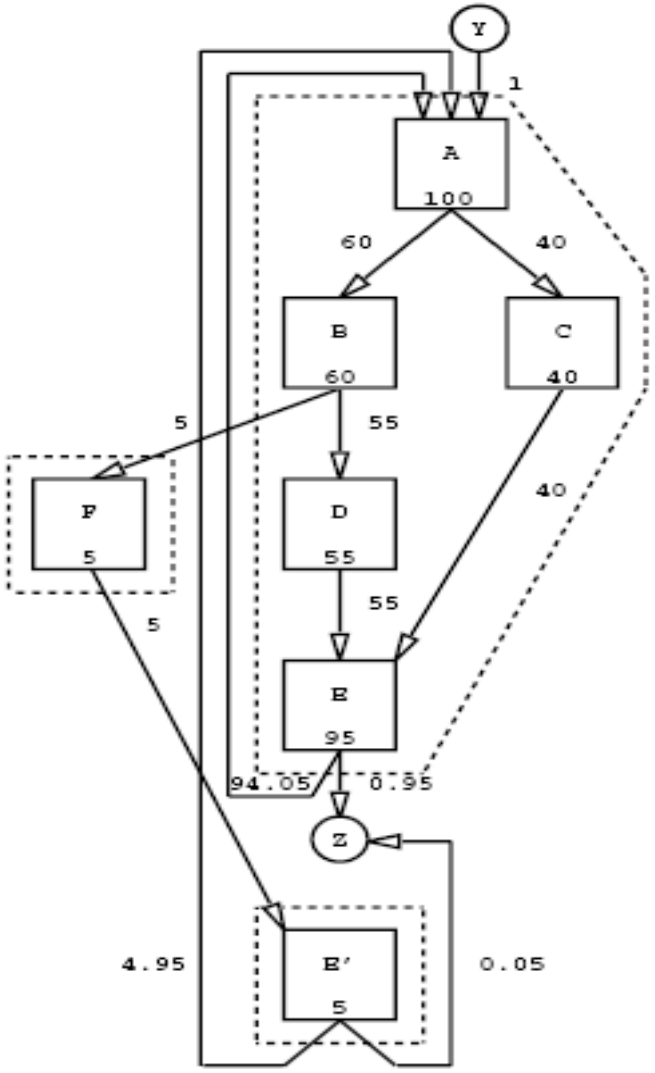
Hyperblock region formation



- Basic block selection

A continuous sequence of BBs
A single dominating BB
A variety of heuristics are used based on execution frequency of BB, size of BB, instruction types, etc.

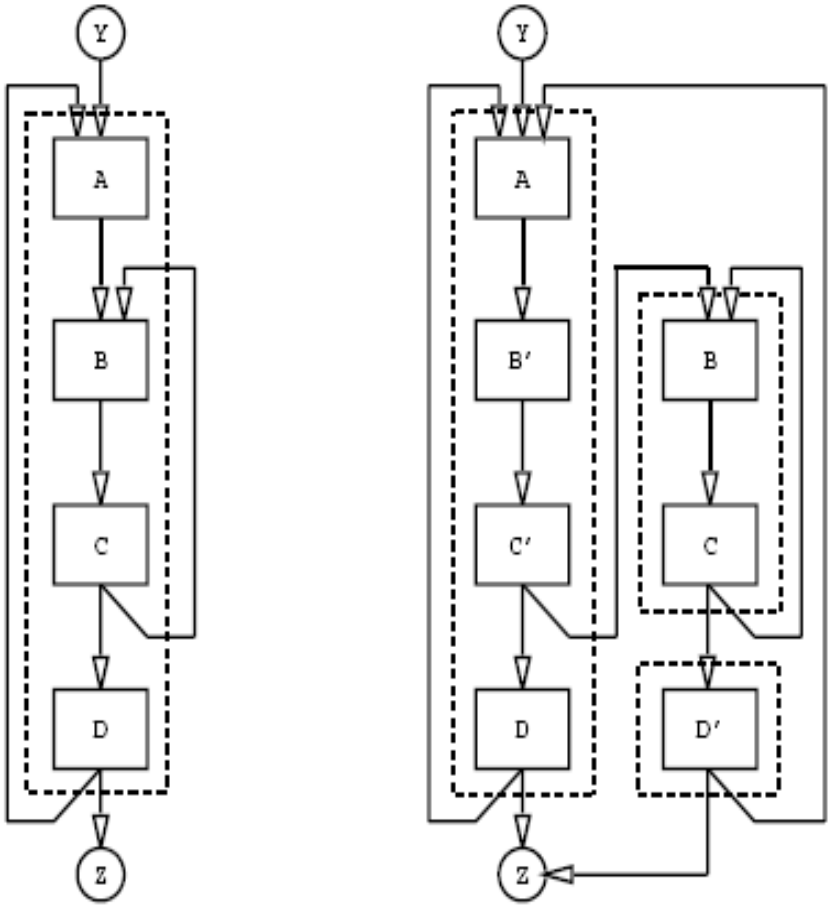
Hyperblock region formation



- Tail duplication

Used to remove entry points into the selected blocks by blocks not selected by inclusion (e.g. F->E)
Need to achieve the single entry property of the hyperblock

Hyperblock region formation



- Loop peeling

Used to remove inner loops from the hyperblock

Unroll the loop (“peeling”) and use tail duplication

Typically unroll as many times as the average number of loop iterations

If the loop executes more times, use the duplicated code for the remainder

Hyperblock region formation

- Predication (if conversion)
  - As shown before
  - Combines instructions from all paths
  - Straight line code for the whole hyperblock
- Predication optimization
- Control and Data Speculation to increase ILP
- Modulo Scheduling
- Code generation

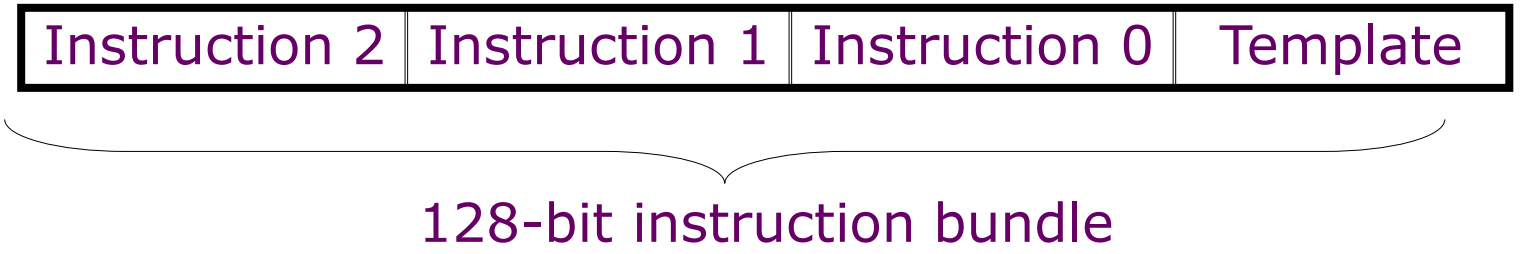
Case study –EPIC architecture

- It is NOT a single processor but rather a family of processors
  - *EPIC* stands for *Explicitly Parallel Instruction Computing*
  - *EPIC* is the style of architecture (like RISC, CISC...)
  - IA-64 (or Itanium) is the ISA (like MIPS, x86, ARM, etc.)
  - McKinley, Merced, Montecito, Montvale, Tukwila, Poulson are specific Itanium (or Itanium2) processors
  - Tukwila released on February 2010, 65nm, 1.73 GHz, 130-185 Watts
  - Poulson released November 2012, 32nm, 3.1 billion transistors, up to 2.53 GHz
  - First implementation McKinley expected for 1997, delivered in 2001!
- Joint project between Intel and HP started in 1994
- Two processor families released: Itanium and Itanium2
- It used to be Intel's 64-bit architecture before Intel64

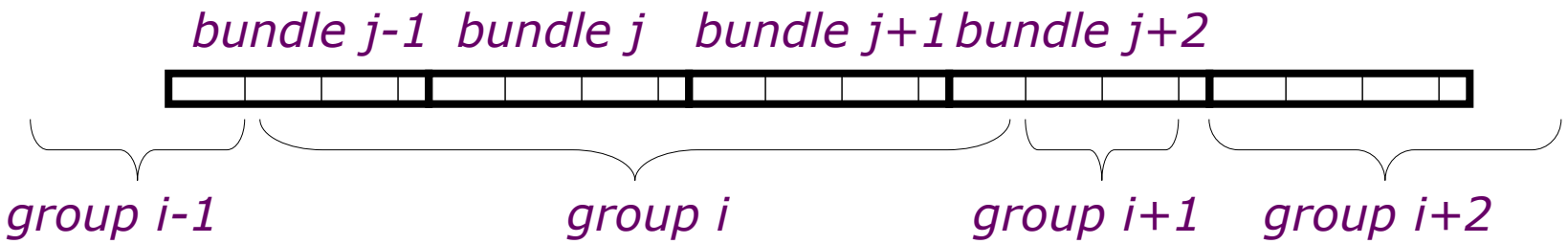
Case study –EPIC architecture

- Itanium2 ISA uses many of the techniques we just discussed
  - Compiler-based instruction predication
  - Speculation both of control and memory references
  - Compiler-assisted exploitation of ILP (e.g. hardware support for modulo scheduling)
- Itanium-based processors are not pure VLIW processors
  - Intel and HP used the term EPIC to distinguish from VLIW and RISC processors
- Itanium uses indicators of possible instruction parallelism rather than a rigid VLIW fixed format

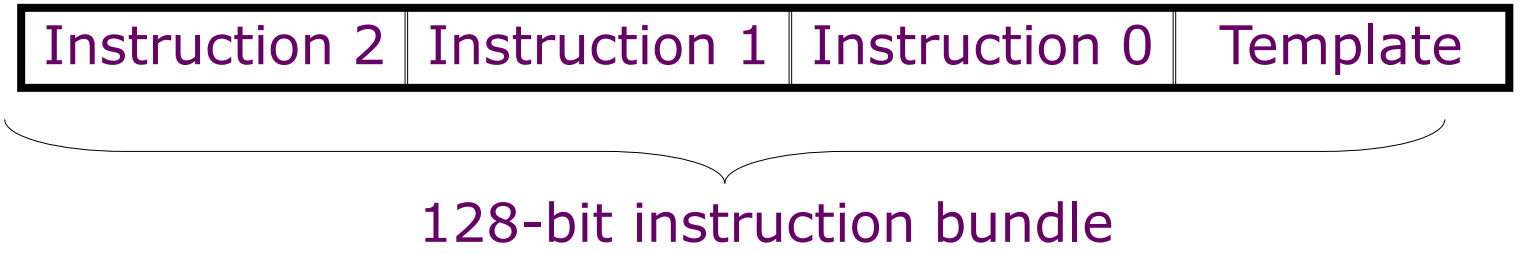
IA-64 Instruction Format



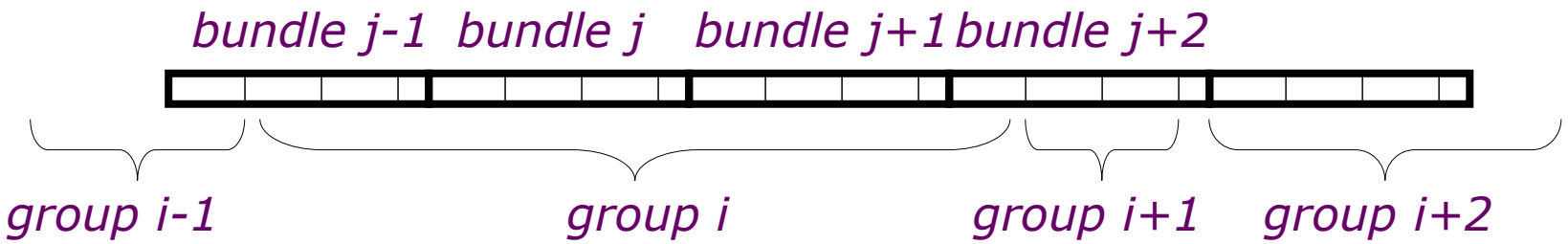
- 1 Bundle = Three 41-bit instruction + One 5-bit template
- Template bits describe what types of execution units needed for each instruction and where are the group boundaries in the bundle.



IA-64 Instruction Format



- Each bundle issued per clock cycle
- Each group contains instructions that can execute in parallel, provided enough hardware resources exist.
- Compiler forms both bundles and instruction groups.



IA-64 Instruction Format

Execution unit slot	Instruction type	Instruction description	Example instructions
I-unit	A	Integer ALU	add, subtract, and, or, compare
	I	Non-ALU integer	integer and multimedia shifts, bit tests, moves
M-unit	A	Integer ALU	add, subtract, and, or, compare
	M	Memory access	Loads and stores for integer/FP registers
F-unit	F	Floating point	Floating-point instructions
B-unit	B	Branches	Conditional branches, calls, loop branches
L + X	L + X	Extended	Extended immediates, stops and no-ops

The 5 execution units slots of IA-64, and the type of instructions they hold

Template	Slot 0	Slot 1	Slot 2
0	M	I	I
1	M	I	I
2	M	I	I
3	M	I	I
4	M	L	X
5	M	L	X
8	M	M	I
9	M	M	I
10	M	M	I
11	M	M	I
12	M	F	I
13	M	F	I
14	M	M	F
15	M	M	F
16	M	I	B
17	M	I	B
18	M	B	B
19	M	B	B
22	B	B	B
23	B	B	B
24	M	M	B
25	M	M	B
28	M	F	B
29	M	F	B

The 24 possible template values and the instruction slots and stops for each format.
Stops are the heavy lines. They may appear within or at the end of the bundle.

Array Increment Example (revisited)

Mem 1	Mem 2	FP 1	FP 2	Int+Br
fld f0,0($1)	fld f6,-8($1)			
fld f10,-16($1)	fld f14,-24($1)			
fld f18,-32($1)	fld f22,-40($1)	fadd f4,f0,f2	fadd f8,f6,f2	
fld f26,-48($1)		fadd f12,f10,f2	fadd f16,f14,f2	
		fadd f20,f18,f2	fadd f24,f22,f2	
fst f4,0($1)	fst f8,-8($1)	fadd f28,f26,f2		
fst f12,-16($1)	fst f16,-24($1)			addi $1,$1,-56
fst f20,-32($1)	fst f24,-40($1)			
fst f28,-48($1)				beq $1,$2,L

Code (x[i] = x[i] + s) statically scheduled

Latencies:

- Load double → FP ALU op = 2 cycles
- FP ALU op → Store double = 3 cycles

Array increment in IA-64

Execute Cycle	Bundle Template	Slot 0	Slot 1	Slot 2
1	9:M M I	fld f0, 0(R1)	fld f6, -8(R1)	
3	14: M M F	fld f10, -16(R1)	fld f14, -24(R1)	fadd f4, f0, f2
4	15: M M F	fld f18, -32(R1)	fld f22, -40(R1)	fadd f8, f6, f2
6	15: M M F	fld f26, -48(R1)	fst f4, 0(R1)	fadd f12, f10, f2
9	15: M M F	fst f8, -8(R1)	fst f12, -16(R1)	fadd f16, f14, f2
12	15: M M F	fst f16, -24(R1)		fadd f20, f18, f2
15	15: M M F	fst f20, -32(R1)		fadd f24, f22, f2
18	15: M M F	fst f24, -40(R1)		fadd f28, f26, f2
21	17: M I B	fst f28, -48(R1)	DADDUI R1,R1, #-56	BNE R1,R2, Loop

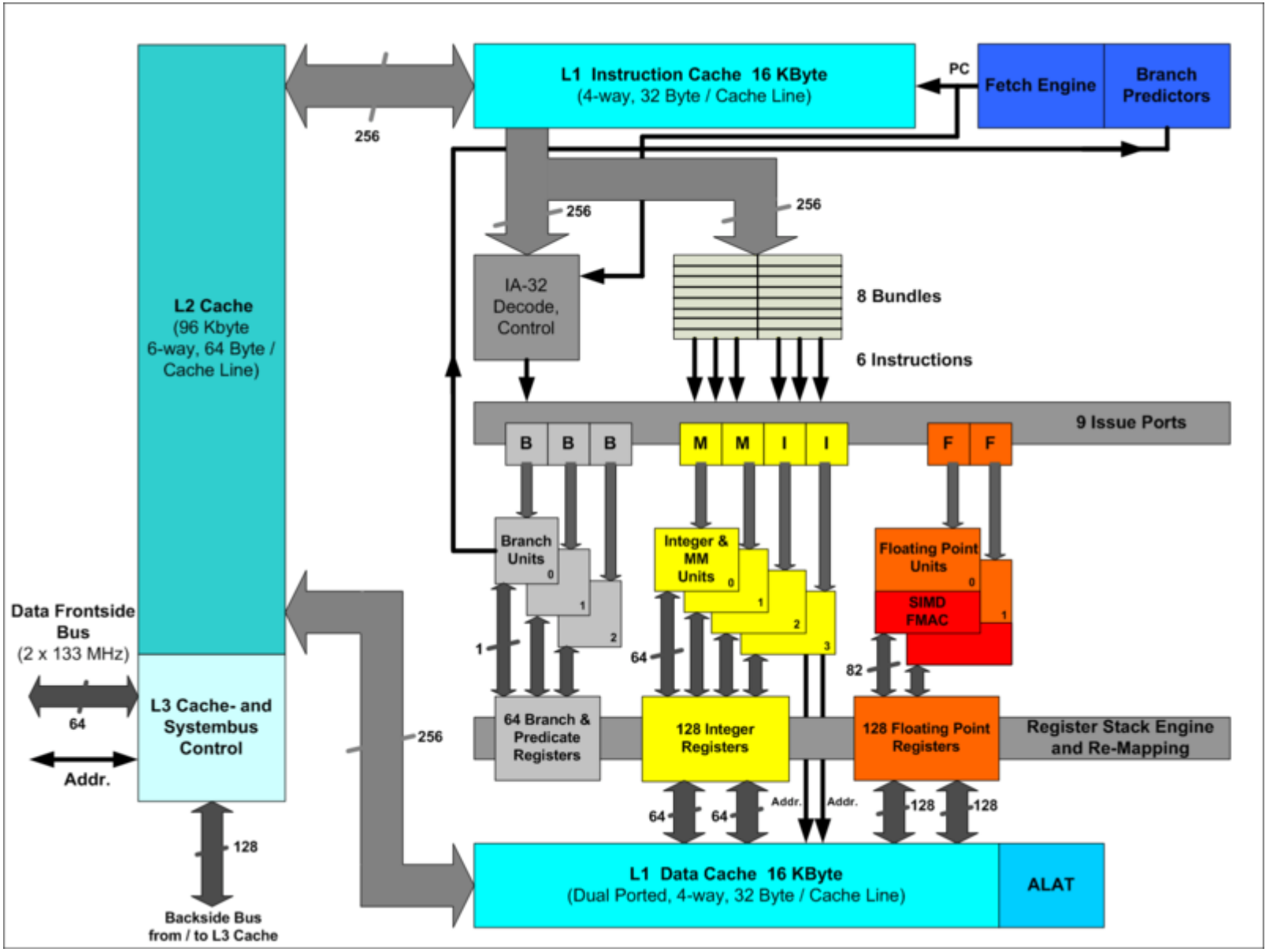
Code scheduled to minimize number of bundles (denser code):
Try to fill up all slots as much as possible

Array increment in IA-64

Execute Cycle	Bundle Template	Slot 0	Slot 1	Slot 2
1	8:M M I	fld f0, 0(R1)	fld f6, -8(R1)	
2	9: M M I	fld f10, -16(R1)	fld f14, -24(R1)	
3	14: M M F	fld f18, -32(R1)	fld f22, -40(R1)	fadd f4, f9, f2
4	14: M M F	fld f26, -48(R1)		fadd f8, f6, f2
5	15: M M F			fadd f12, f10, f2
6	14: M M F		fst f4, 0(R1)	fadd f16, f14, f2
7	14: M M F		fst f8, -8(R1)	fadd f20, f18, f2
8	15: M M F		fst f12, -16(R1)	fadd f24, f22, f2
9	14: M M F		fst f16, -24(R1)	fadd f28, f26, f2
11	9: M M I	fst f20, -32(R1)	fst f24, -40(R1)	
12	17: M I B	fst f28, -48(R1)	DADDUI R1,R1, #-56	BNE R1,R2, Loop

Code scheduled to minimize number of cycles: Schedule based on ASAP

Itanium Architecture



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

IA-64 performance comparison

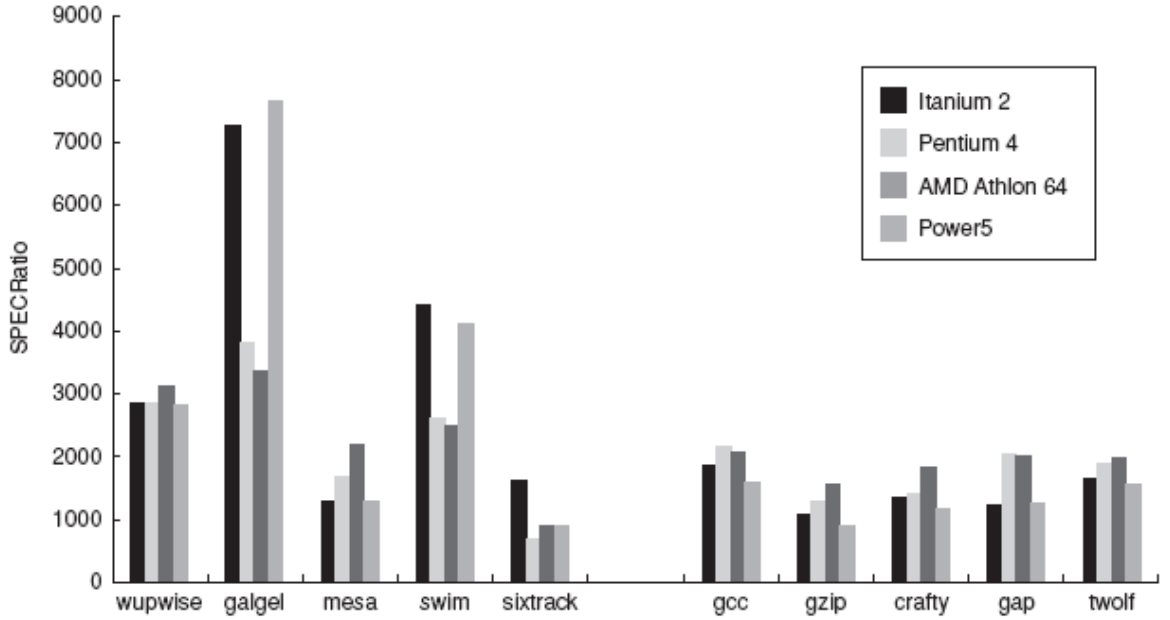


Figure G.11 The performance of four multiple-issue processors for five SPECfp and SPECint benchmarks. The clock rates of the five processors are Itanium 2 at 1.5 GHz, Pentium 4 Extreme Edition at 3.8 GHz, AMD Athlon 64 at 2.8 GHz, and the IBM Power5 at 1.9 GHz.

- Relatively better for FP!
- Then why do we need all the predication, speculation complexity??

IA-64 discussion

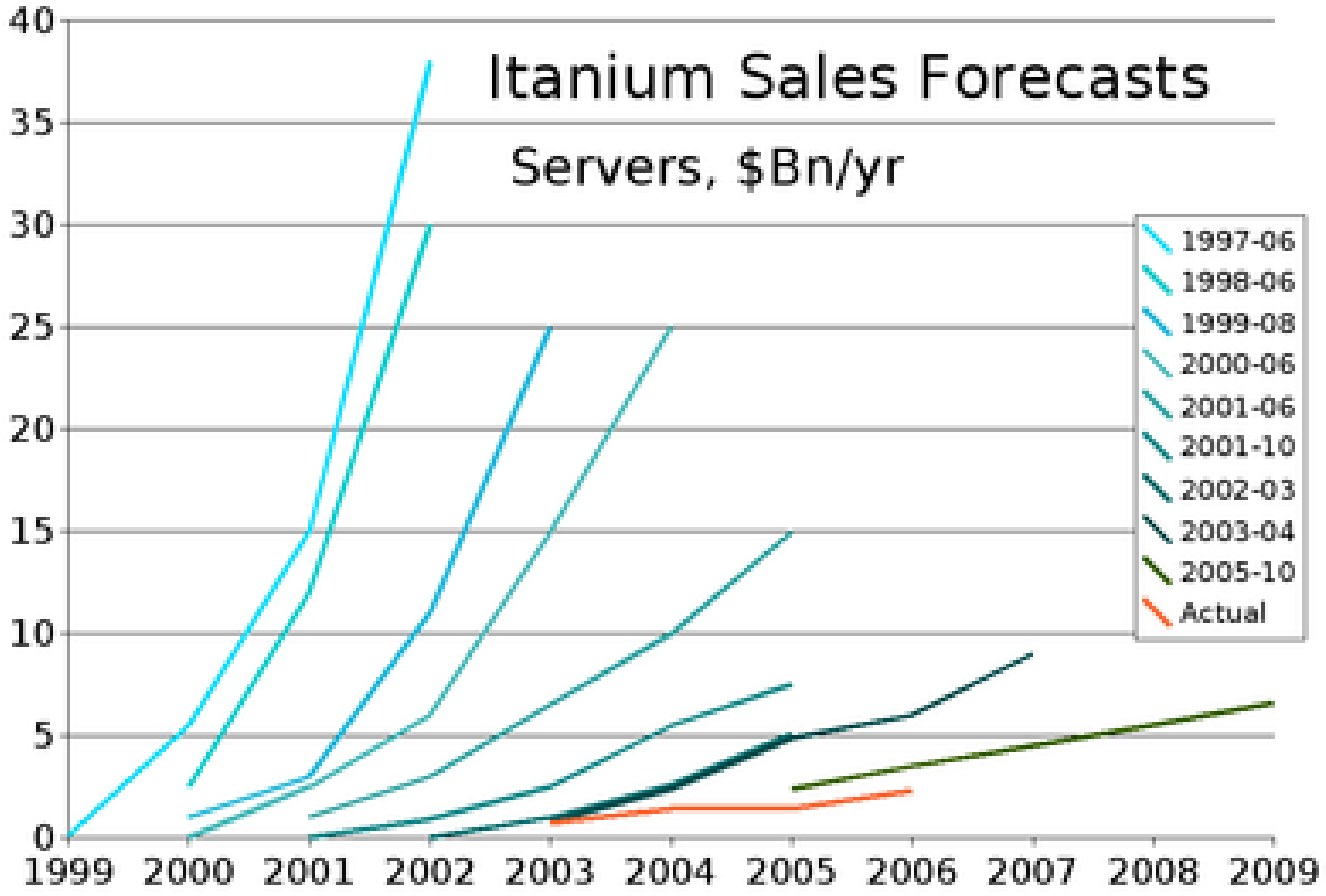
- “One of the surprises about IA-64 is that we hear no claims of high frequency, despite claims that an EPIC processor is less complex than a superscalar processor. It is hard to know why this so, but one can speculate that the overall complexity involved in focusing on CPI, as IA-64 does, makes it hard to get high MHz“

  M. Hopkins [IBM]

IA-64 discussion

- Basic idea of the EPIC technology was to transfer complexity from hardware to compiler
- Idea was never proven correct
- EPIC hardware has proven to be as complex as Pentium's
- Itanium/Itanium2 could never reach even half the clock speed of Pentium-4 (1.7 vs 3.8 GHz)
- High compiler AND hardware complexity

IA-64 discussion



Sales forecasts never materialized