Hardware and Software for VLIW and EPIC

By

Divya Navaneetha Krishna
Sharanya Chinnusamy

Outline

- Detecting and Enhancing Loop-Level Parallelism
- Finding and Eliminating dependences
- Software Pipelining
- Global Code Scheduling
  - Trace Scheduling
  - Superblocks
- Hardware Support for Exploiting Parallelism - Predicate instructions
- Hardware Support for Compiler Speculation
  - Preserving exception behavior
  - Memory reference speculation
- Outline of Intel Architecture
- Demo
- Comparisons

VLIW

- Very Long Instruction Word
- One large instruction consisting of independent MIPS instructions (or)
- Packet of instructions which can be executed in parallel
- Compiler is responsible to minimize hazards and form packets
- Loop unrolling and code scheduling (Local and Global)

Detecting and Enhancing Loop-Level Parallelism

- Analyzed at the source level
- Determine the dependences that exist
  - Data dependences
  - Name dependences
  - Loop carried dependences

```
for (i=1; i<=100; i=i+1) {
      A[i+1] = A[i] + C[i];   /* S1 */
      B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

- Loop carried dependency
- True dependency

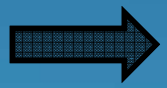
```
for (i=1; i<=100; i=i+1) {
      A[i] = A[i] + B[i];   /* S1 */
      B[i+1] = C[i] + D[i]; /* S2 */
}
```

- Loop carried dependency that does not prevent parallelism

Detecting and Enhancing Loop-Level Parallelism

Overlapping iterations

```
for (i=1; i<=100; i=i+1) {
      A[i] = A[i] + B[i];   /* S1 */
      B[i+1] = C[i] + D[i]; /* S2 */
}
```



A[1] = A[1] + B[1]
B[2] = C[1] + D[1]

A[2] = A[2] + B[2]
B[3] = C[2] + D[2]

•
•

A[100] = A[100] + B[100]
B[101] = C[100] + D[100]

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1) {
      B[i+1] = C[i] + D[i];
      A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

Detecting and Enhancing Loop-Level Parallelism

- Loop-carried dependences are in the form of a recurrence
- Reasons:
  - Provide support for recurrences (Vector computers)
  - Helps in parallelism

```
for (i=6;i<=100;i=i+1) {
        Y[i] = Y[i-5] + Y[i];
}
```

Dependency distance is **5**

Finding and Eliminating Dependences

- Data dependency using register renaming
- Dependency analysis using affine indices (a x i + b)
  - 2 iteration indices j and k within a for loop such that (m <= j <=n , m <= k <= n)
  - indexed as (a x j + b) and (c x k + d), then a x j + b = c x k + d
  - GCD test [GCD (c , a) must divide (d-b)]

```
for ( i=1;  i<=100;  i=i+1) {
        X[ 2*i + 4 ]  =  X[ 2* i ] + 5.0
 }
```

a = 2
b = 4
c = 2
d = 0
GCD(c , a) = 2
d-b = -4

Finding and Eliminating Dependences

Limitations of array-oriented dependences :

- Using pointers to reference arrays
- Sparse array (X[Y[i]]) -- non affine
- False dependency

  At runtime the inputs never take the value which may have resulted in dependency
- Interprocedural analysis

Finding and Eliminating Dependences

- Back Substitution

- Copy Propagation

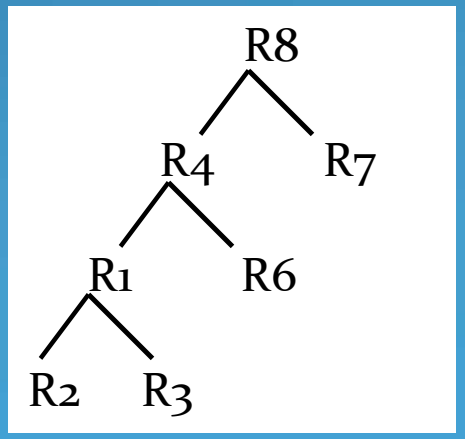
```
DADDUI    R1,R2,#4
DADDUI    R1,R1,#4
```

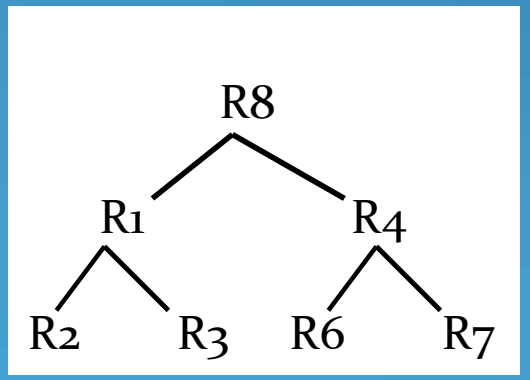
```
DADDUI    R1,R2,#8
```

- Tree height reduction

```
ADD       R1,R2,R3
ADD       R4,R1,R6
ADD       R8,R4,R7
```

```
ADD       R1,R2,R3
ADD       R4,R6,R7
ADD       R8,R1,R4
```





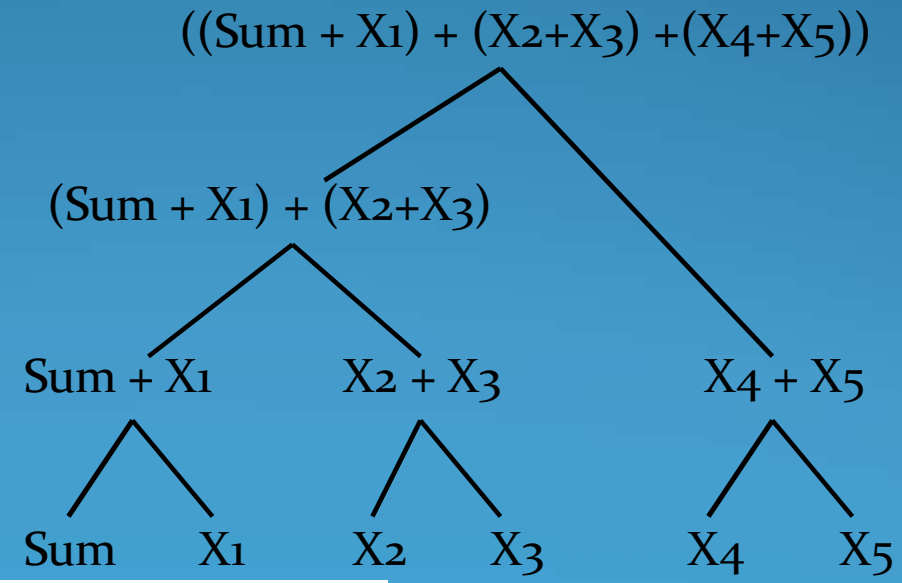
Finding and Eliminating Dependences

- Optimizing unrolled recurrence relation

Sum = Sum + X

Unroll the loop

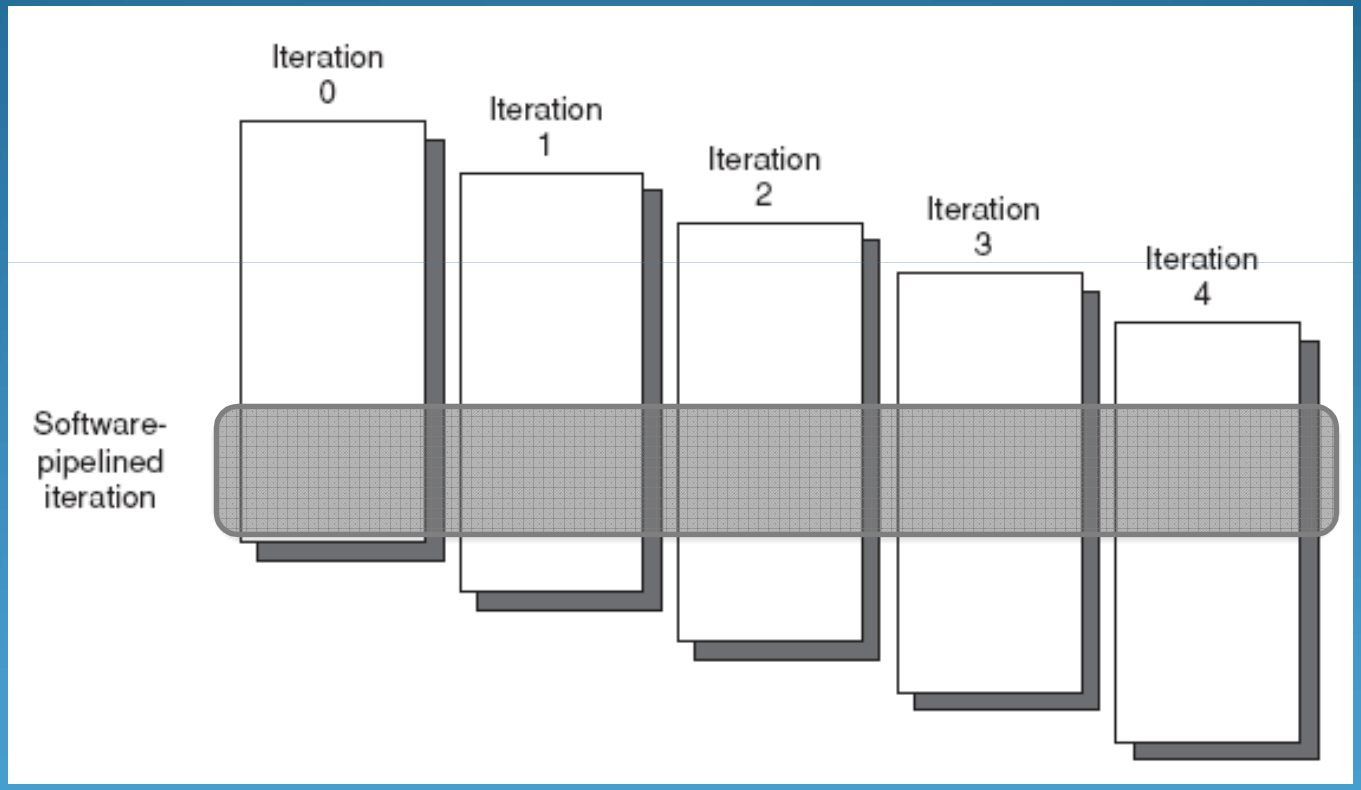
Sum = Sum + X1 + X2 + X3 + X4 + X5



Sum = ((Sum + X1) + (X2 + X3) + (X4 + X5))

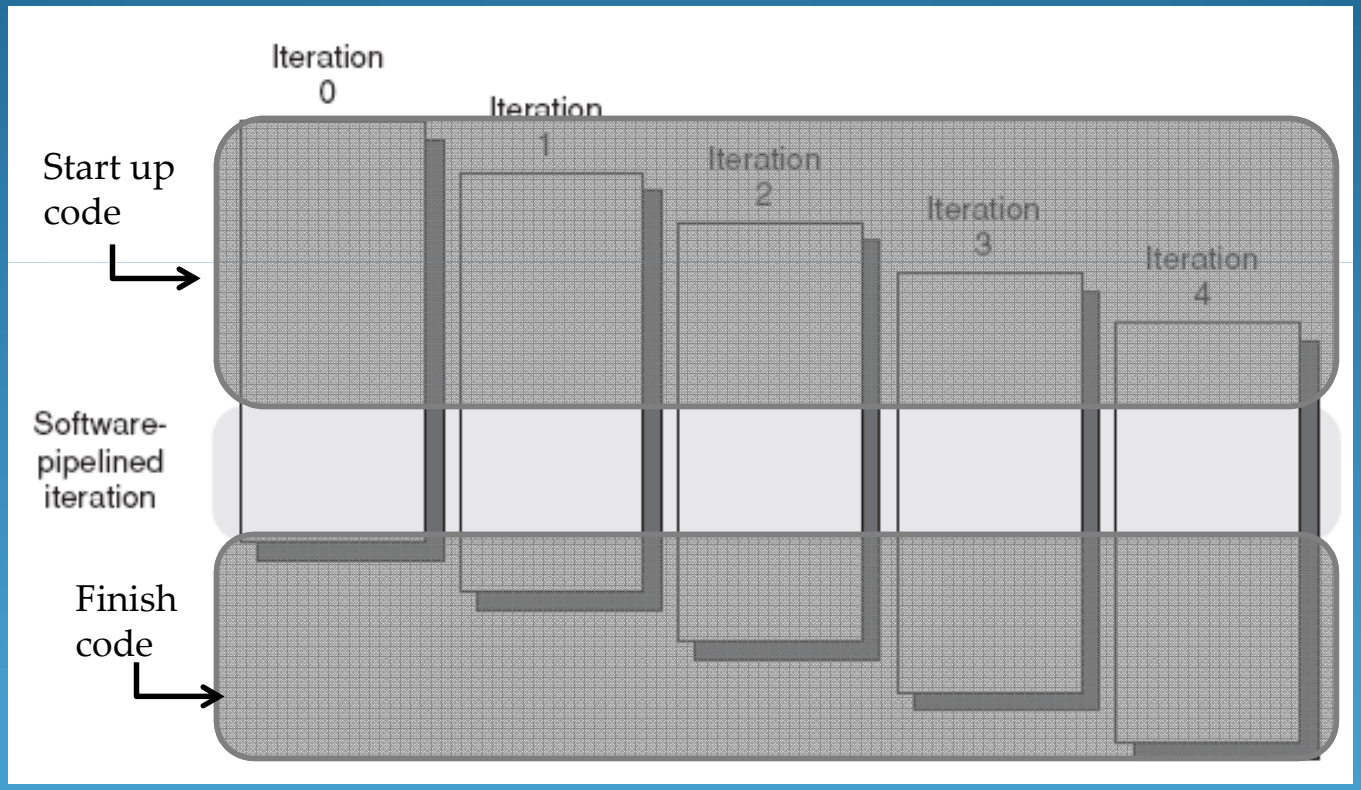
Software Pipelining

- Interleaves instruction from different iterations without unrolling the loop



Software Pipelining

- Interleaves instruction from different iterations without unrolling the loop

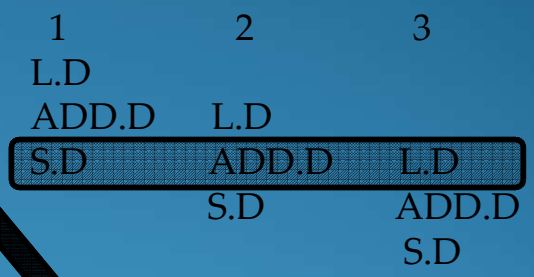


Software Pipelining

```
LOOP:   L.D      F0, 0(R1)
        ADD.D    F4, F0, F2
        S.D      F4,0(R1)
        DADDUI   R1,R1,#-8
        BNE      R1,R2, LOOP
```

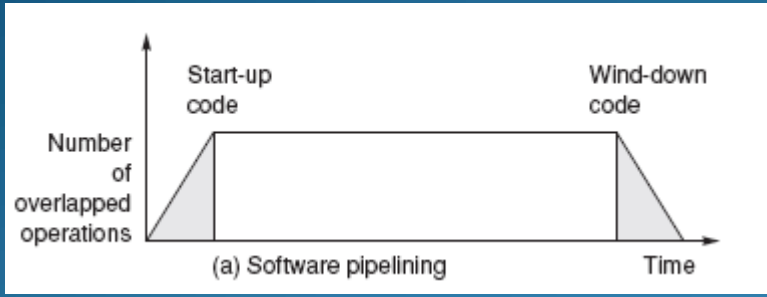
Loop Unrolling

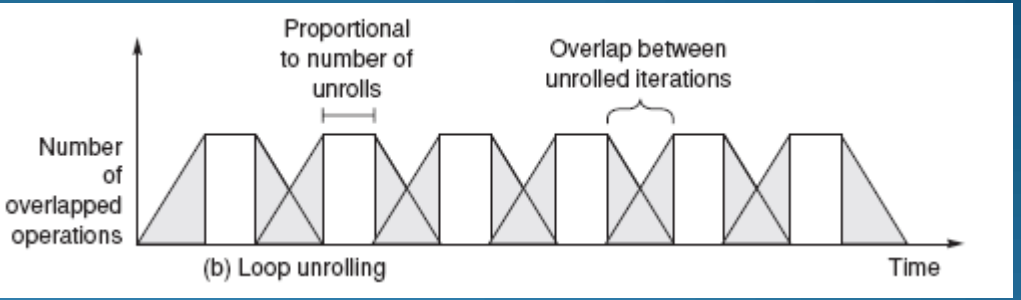
```
LOOP:   L.D      F0,0(R1)
        ADD.D    F4,F0,F2
        S.D      F4,0(R1)

        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)

        DADDUI   R1,R1,#-24
        BNE      R1,R2,LOOP
```




```
LOOP:   S.D      F4, 16(R1)
        ADD.D    F4, F0, F2
        L.D      F0,0(R1)
        DADDUI   R1,R1,#-8
        BNE      R1,R2, LOOP
```

Software Pipelining





Advantages:

- Consume less code space
- Reduces time when the loop is not running at peak speed to once per loop at the beginning and end

Advantage:

- Reduces over head of the loop

Disadvantage :

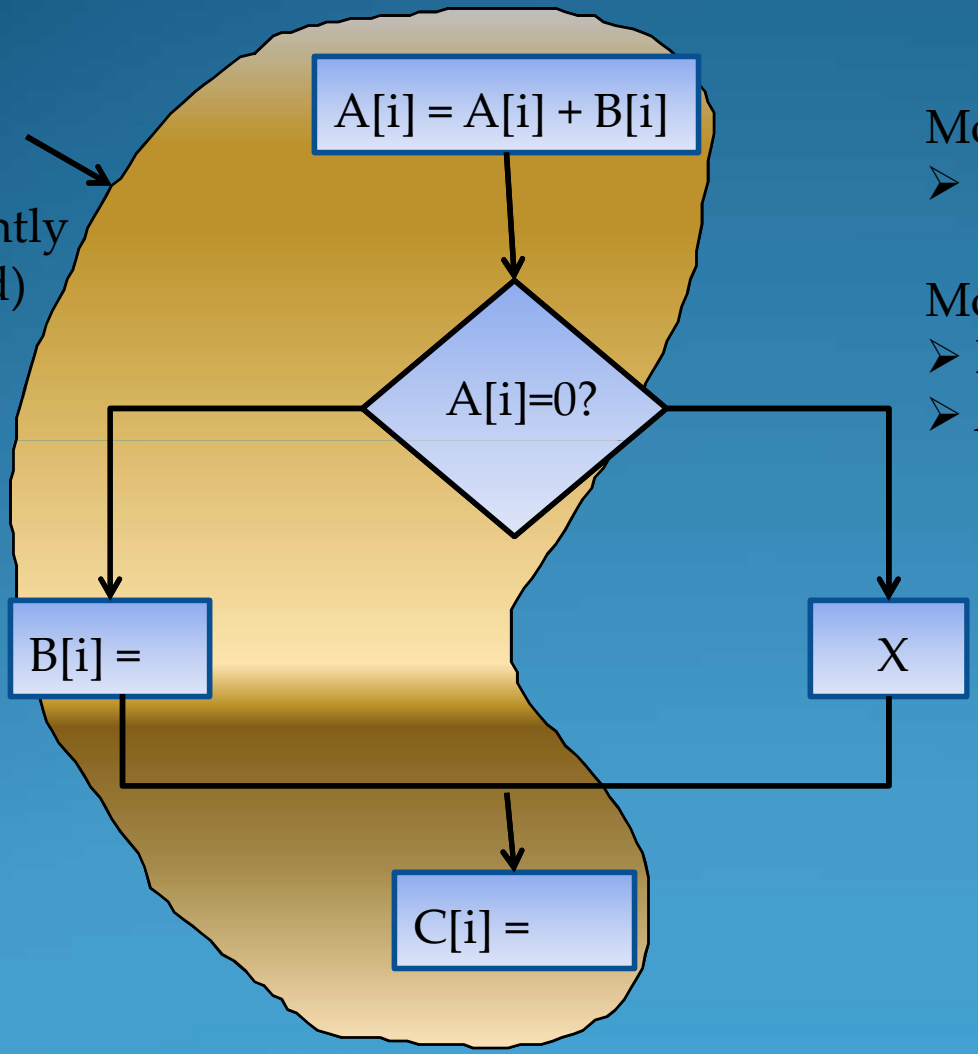
- Fills and drains the pipeline each time the loop is to be executed

- Advantages of using both
- IA-64 added hardware support

Global Code Scheduling

- Effective scheduling require moving instructions across branches
- Preserves both data and control dependences
- Data dependence removed by unrolling and dependence analysis
- Control dependence is removed by unrolling and moving code across branches
- Estimates the frequency of different paths for code movement

Global Code Scheduling



Movement of B

- Dependency in X – Shadow copy

Movement of C

- Into then part – copy in X
- Across the branch

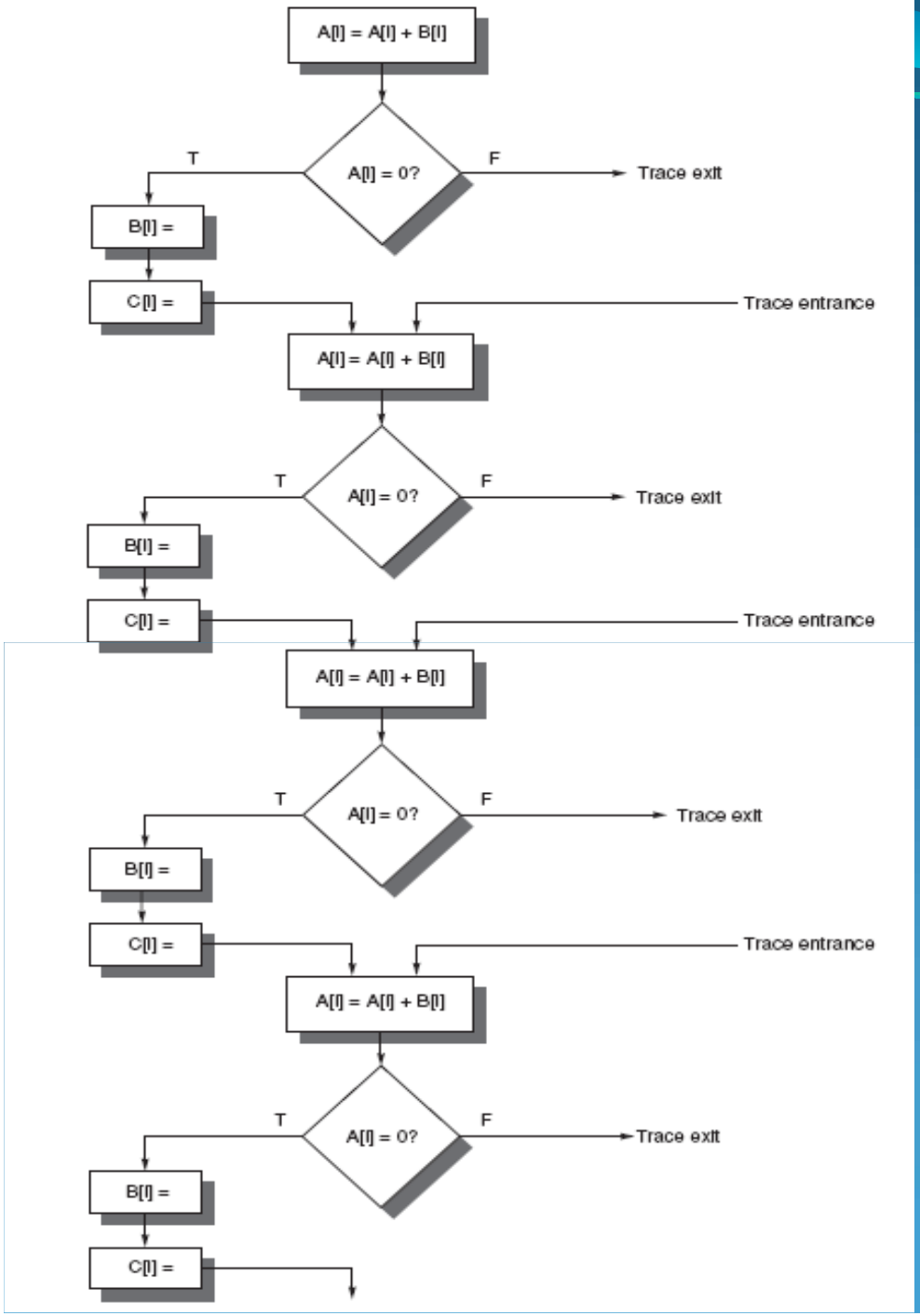
Global Code Scheduling

Factors to be considered

- Relative execution frequency of then and else clause
- Cost of executing B above branch
- Change in execution time – movement of B
- B or C – best code fragment to move
- Cost of compensation code

Trace Scheduling

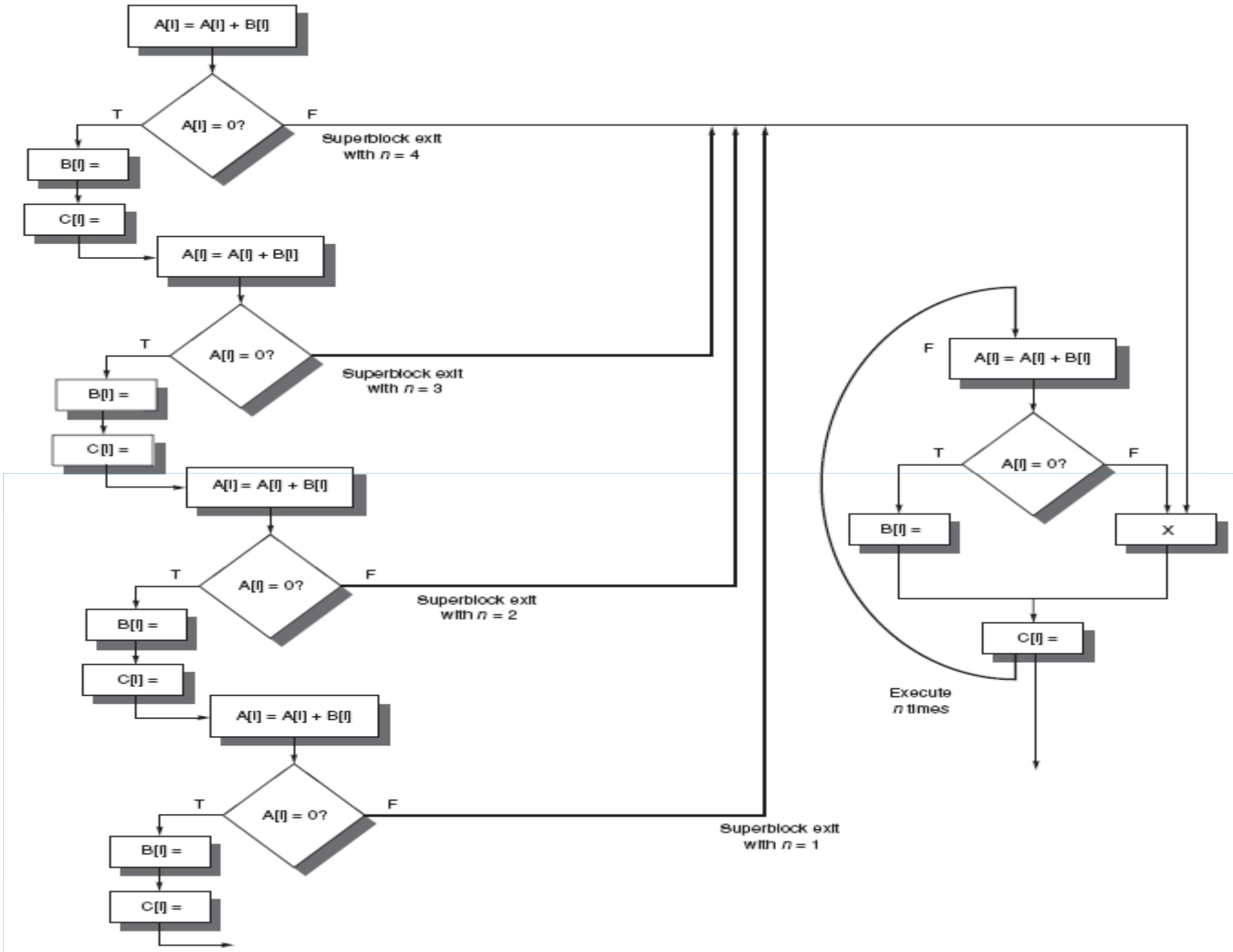
- Used when
  - Processors with large number of issues per clock
  - Predicated or conditional branch unsupported
  - Unrolling is not sufficient
  - Significant difference in frequency between different paths
- Steps
  - Trace selection
  - Trace compaction



- Loop unrolling
- Static branch prediction
- Branches are jumps into or out of the trace
- Common set of instructions are executed sequentially
- Bookkeeping for trace entrance and exits
- Good for loop intensive code

Super blocks

- Reduces complication caused by various entries and exits into the middle of the trace
- Similar to trace but only one entrance and many exits
- Tail duplication that corresponds to the portion of the trace after entry
- The residual loop handles iterations when exited (unpredicted path selected)
- Frequency of residual loop – high – create super block for it


A[i] = A[i] + B[i]
A[i] = 0?
T
F
B[i] =
C[i] =
Superblock exit with n = 4
A[i] = A[i] + B[i]
A[i] = 0?
T
B[i] =
C[i] =
Superblock exit with n = 3
A[i] = A[i] + B[i]
A[i] = 0?
T
F
B[i] =
C[i] =
Superblock exit with n = 2
A[i] = A[i] + B[i]
A[i] = 0?
T
F
B[i] =
C[i] =
Superblock exit with n = 1
F
A[i] = A[i] + B[i]
A[i] = 0?
T
F
B[i] =
X
C[i] =
Execute n times

Summary

- Loop unrolling helps reduce the loop overhead
- Software pipelining reduces stalls due to single loop body
- Advantages of using both Software pipelining and loop unrolling
- Trace Scheduling / Superblocks for global code scheduling across branches
- Hybrid usage – compilers
- All fail if branch prediction is unreliable

Hardware Support for Exposing Parallelism
Predicated Instructions

Motivation

- Loop unrolling, software pipelining, and trace scheduling work well – But only when branches are predicted at compile time
- In other situations branch instructions can severely ***limit parallelism!!!!***

Solution

- Let the architect extend the instruction set to include ***conditional or predicated instructions.***

What do these instructions do?

- Instructions refer to a condition that is evaluated at the time of execution
  - If the condition is true, the instruction is executed normally
  - Else, it behaves like a no-op.

Example

Consider the following statement:

if (A == O) { S = T };

Assume that:

R1 ⟵ A R2 ⟵ S R3 ⟵ T

The code for this would look like:

```
        BNEZ    R1, L          ; if R1 is not equal to zero, jump to L
        ADDU    R2, R3, R0     ; else Move R3 to R2
L:
```

A better way to have this is with the use of a *Conditional Move*

```
        CMOVZ   R2, R3, R1     ;When R1 is zero, move R3 to R2
```

What does this do?

- Converting a control dependence to a data dependence
- For a pipelined processor this essential moves the dependence

  From: front of the pipeline To: end of the pipeline

  (where branches are resolved) (where register writes occur)
- Eliminates simple branches and improves the pipeline's performance

When is this inefficient and why?

When branches guard large blocks of code. Because this will introduce many conditional moves.

Remedy to this?

- Have support for full predication
- Execution of all the instructions is controlled by a predicate
- When the predicate is true, the instruction is executed else it becomes a no-op
- It is valuable for Global Code Scheduling because it eliminates non loop branches

Conditional Instructions in a Superscalar processor

Two – issue superscalar that can issue – one memory reference + either An ALU operation / branch

First Instruction Slot	Second Instruction Slot
LW R1, 40(R2)	ADD R3, R4, R5
	ADD R6, R3,R7
BEQZ R10, L	
LW R8, O(R10)	
LW R9, O(R8)	

Problems?

- Waste of a memory operation slot in cycle -2
- Incurs a data dependence (RAW) if the branch is not taken

How can this code be improved with a predicated LW instr?

First Instruction Slot		Second Instruction Slot	
LW	R1, 40(R2)	ADD	R3, R4, R5
LW C	R8, O(R10), R10	ADD	R6, R3,R7
BEQZ	R10, L		
LW	R9, O(R8)		

- The predicate is – if R10 is not zero
- If the predicate is true: R8 ⟵ R10
- Else – The operation turns to a no-op

This improves performance! How?

- Eliminates an issue cycle for one instruction
- Saves the last load from stalling due to a stall
- Overall we remove control dependences by making instruction predicted

Problems with Predicated instructions?

What happens when a predicated instruction generates an exception ?
(NOTE: The predicate was false)

Its hard to implement. Why?

When do you annul an instruction?

Two ways:

- Annulled during the execution issue
  - Requires that the value of the controlling condition be available early in the pipeline - Might cause a potential data hazard
- Or later before they commit any results
  - All existing processors follow this
  - Disadvantage is that these annulled instructions have already consumed functional resource
  - Might affect performance

So when are predicated instructions useful?

- Implementing short alternative control flows
- Eliminating some unpredictable branches
- Reducing the overhead of global code scheduling
- When the predicate can be evaluated early – will help potential data hazards

Factors that limit its usefulness:

- Predicated instructions that are annulled also consume processor resources
- Slows the program down if the predicated instructions were not going to be executed during the normal program flow
- When the control flow involves more than a simple alternative sequence
- Consume more cycles than an unconditional instruction. Must be used judiciously when they are expensive

Hardware Support for Compiler Speculation

Speculation:

Compiler speculation is desired for improving the scheduling or increase the issue rate

Three capabilities are required to speculate ambitiously:

- Ability of the compiler to speculatively move instructions using register renaming without affecting program data flow - Compiler Capability

- Ability to be able to ignore exceptions in speculated instructions
- The ability to speculatively interchange loads and stores, or stores and stores, which may have address conflicts - Hardware Support

Hardware support to preserve Exception Behavior

How can exception behavior be preserved?

- The results of a mis-predicted speculated sequence should not used in the final computation
- Such an instruction should not cause an exception

Four methods have been investigated

- Hardware and OS cooperatively ignore exceptions for speculated instructions.
- Speculated instructions should never raise exceptions. Introduce checks to determine when an exception should occur
- Poison bits are attached to the result registers written by such instructions that cause exceptions. The poison bits cause a fault when a normal a instruction attempts to use the register
- A mechanism to indicate that an instruction is speculative. So that the hardware can buffer the instruction result until it is certain that the instruction is no longer speculative

Two kinds of exceptions:

- Exceptions that cause the program to terminate
  - Eg: Memory protection violation, illegal operation
  - Should not be handled for speculated instructions unless it is certain that the instruction is no longer speculative
- Exceptions that can be handled and program can be resumed
  - Eg: Page fault, I/o
  - Such exceptions can be handled for speculated instructions as for normal instructions
  - Drawback is that it might cause performance penalty if the instruction was not executed during normal program execution

Speculation by hardware and OS co-operation

- Resumable exceptions are handled normally (even for speculated instr 's)
- Returns an undefined value for exceptions that cause termination

Okay....Not okay????

Example: if (A == O) A = B; else A = A+4;

A ⟵ O(R3)

B ⟵ O(R2)

Instructions			Comments
	LD	R1, O(R3)	Load A
	BNEZ	R1, L1	Test A
	LD	R1, O(R2)	Then clause
	J	L2	Skip else
L1:	DADDI	R1, R1, #4	Else clause
L2:	SD	R1, O(R3)	Store A

How can this code be compiled speculatively?

Assume that the then case will almost always be taken

Instructions			Comments
	LD	R1, O(R3)	Load A
	LD	R14, O(R2)	Speculatively Load B
	BEQZ	R1, L3	Other branch of the if
	DADDI	R14, R1, #4	Else clause
L3:	SD	R14, O(R3)	Non-Speculative store

Second Approach: Poison Bits

- Exceptions are tracked as they occur
- Terminating exceptions are postponed until when their value is actually used

How is this accomplished?

- Two bits are added to each register:
  - A poison bit
  - Another bit to indicate if the instruction was speculative
- The poison bit is set for the destination register whenever a speculative instruction results in a terminating exception
- Normal exceptions are handled immediately
- If a normal instruction attempts to use a source register with its poison bit turned on then an exception is raised
- May require special support for instructions that set and reset the poison bit

Third Approach: Reorder Buffer (ROB)

- Compiler marks instructions as speculative, also indicating the compilers assumption of taken/not taken
- This information is used by the hardware to locate where the speculated instruction originally was.
- Each original location is marked by a sentinel, that tells the hardware that the earlier speculative instruction is no longer speculative
- All instructions are placed in the ROB, and commit is forced in the program order
- The ROB postpones write-back of speculated instructions until:
  - All the braches that were speculated for the instruction are ready to commit
  - Or the sentinel for the instruction is reached
- If the speculated instruction should have been executed and it generated a terminating exception, the program is terminated

Hardware support for Memory Reference Speculation

- The critical path length can be reduced by the compiler by moving loads across stores
- Moving loads across stores - requires checks to see there are no address conflicts
- This special instruction is left at the original location of the load, and the load is then moved across one or more stores
- Hardware stores the address of the memory location after a speculated load
- If subsequent stores change the location before the check, speculation has failed, else it was successful

Two ways to handle failed Speculation

- If only the load was speculated – Redo the load at the point of the check
- If additional instructions dependent on the load were speculated – Redo all the speculated instructions after the load

EPIC – Explicitly Parallel Instruction Computer

- RISC architectures were reaching a limit at one instruction per cycle
- VLIW allowed for multiple operations to be encoded in every instruction, which could then be processed by multiple execution units
- EPIC aimed to move the complexity of instruction scheduling from the CPU hardware to the software compiler
- Also, to further exploit *ILP*, by using the compiler to find and exploit additional opportunities for parallel execution
- VLIW had fixed instruction formats and the load instructions from the memory hierarchy did not have deterministic delays

Is this a drawback???

- This made scheduling of load instructions by the compiler very difficult!

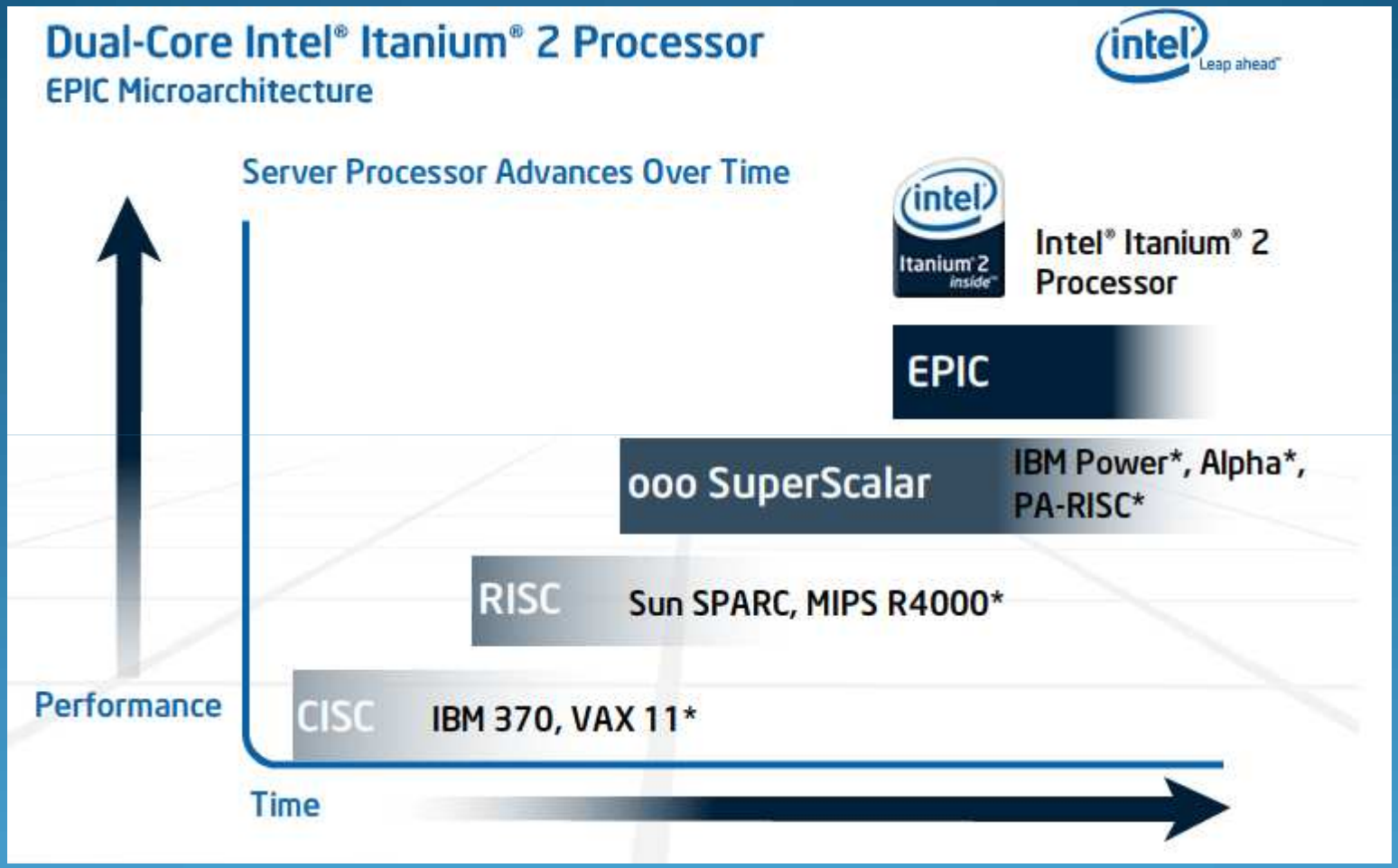
Features of EPIC architecture:

- Had greater flexibility in expressing parallelism among instructions and formats
- Implements speculative loads – as a form of data prefetch
- Supports a check load instruction to check for dependencies over the previous stores
- Supports Predicated execution to decrease the occurrences of branches
- Supports Delayed exceptions to increase speculation
- Supports large architectural register files avoid the need for register renaming

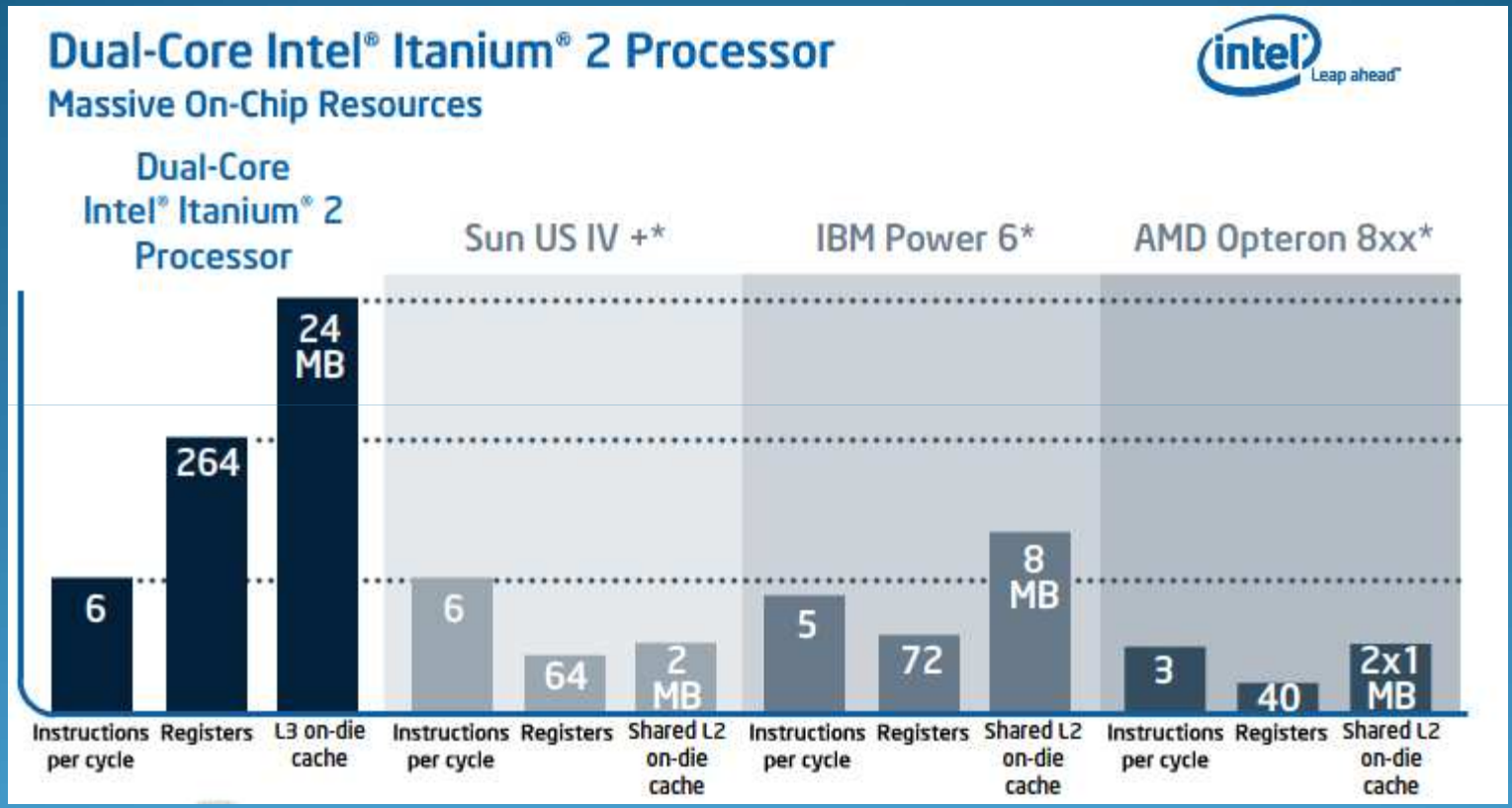
Demo

http://www.intel.com/products/processor/itanium2/demo/index.htm?iid=itanium+body_demo?iid=itanium+body_demo

Demo


Dual-Core Intel® Itanium® 2 Processor
EPIC Microarchitecture
intel
Leap ahead
Server Processor Advances Over Time
intel
Itanium 2
inside
Intel® Itanium® 2 Processor
EPIC
ooo SuperScalar
IBM Power*, Alpha*, PA-RISC*
RISC
Sun SPARC, MIPS R4000*
CISC
IBM 370, VAX 11*
Performance
Time

Demo


Dual-Core Intel® Itanium® 2 Processor
Massive On-Chip Resources
intel Leap ahead
Dual-Core Intel® Itanium® 2 Processor
Sun US IV +*
IBM Power 6*
AMD Opteron 8xx*
6
264
24 MB
6
64
2 MB
5
72
8 MB
3
40
2x1 MB
Instructions per cycle
Registers
L3 on-die cache
Instructions per cycle
Registers
Shared L2 on-die cache
Instructions per cycle
Registers
Shared L2 on-die cache
Instructions per cycle
Registers
Shared L2 on-die cache

References

- Computer Architecture – A Quantitative Approach
  Author: John L Hennessy and David A. Patterson
- http://www.intel.com/intelpress/chapter-scientific.pdf
- http://www.cs.clemson.edu/~mark/epic.html
- http://www.siliconintelligence.com/people/binu/coursework/686_vliw/vliw.pdf
- http://en.wikipedia.org/wiki/Explicitly_Parallel_Instruction_Computing

Thank You!!

QUESTIONS????