

Hardware and Software for VLIW and EPIC

By: Anindya Lahiri

Gagan Arora

{alahiri, arorag}@cs.indiana.edu

What is VLIW and EPIC ?

- Very Long Instruction Word
 - Multiple issue processor
 - Issues a fixed number of instructions formatted as either as one large instruction or as fixed instruction packets
 - Inherently statically scheduled by the compiler
- EPIC (Explicitly parallel instruction computer)

Exploiting ILP Parallelism Statically

- Increasing parallelism using compiler technology
- Dependences preventing parallelism
- Techniques for eliminating such dependences
- Hardware support for such techniques

ILP using Compiler

- Advantages:
 - Do not burden runtime executions
 - Consider wider range of the program
 - Determining parallelism of entire loop
- Disadvantage:
 - Use only compile time information
 - Conservative and assumes worst case

Detecting and Enhancing Loop-Level Parallelism

- Analyzed at source level
- Loop-level dependences analysis among the operands in a loop across iterations
 - Data dependences: RAW hazard
 - Loop-carried dependence: values required from previous iterations
 - Name dependences ?
 - Can be removed by register renaming

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

```
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 */  
}
```

What are the dependences here ?

- S1 and S2 use values from previous iteration
 - Loop-carried
 - Requires execution in series, RAW
- S2 depends on S1, requires A[i+1] from S1
 - Do not impact parallel execution

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

- Dependences between S1 and S2:
 - Loop-carried dependence but not circular i.e. S1 depends on previous value of S2, S2 does not depend on S1
- Parallelizing:

First iteration of S1 depends
on B[1]



```
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];
```

- Complex analysis for data dependence is required

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

- Second reference to A can be to same or different register
- Optimization requires knowledge about the reference made to memory location

- Loop carried dependences in form of recurrences

```
for (i=2;i<=100;i=i+1) {  
    Y[i] = Y[i-1] + Y[i];  
}
```

- Important to detect:
 - Special support for executing recurrences, e.g. Vector processors
 - Parallelism by loop unrolling

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

Dependence distance is 5: five statements will have no dependence

- Finding dependences is important:
 - Leads to good scheduling of code
 - Determining parallelism in loops
 - Eliminating name dependences
- Complexity of dependences due to presence of arrays and pointers, or pass by reference parameters -> aliasing
- Detecting dependences based on indices of arrays being affine, $a \times i + b$
 - Determining dependence between two references to a array in a loop is equivalent to determining whether two affine functions can have same value for different indices between the bounds of the loop

- Store into array indexed by: $a \times j + b$
- Fetched at the same location when indexed by: $c \times k + d$, thus $a \times j + b = c \times k + d$
- GCD dependency test:
 - $GCD(c, a)$ must divide $(d - b)$

```
for (i=1; i<=100; i=i+1) {  
    X[2*i+3] = X[2*i] * 5.0;  
}
```

$a = 2, b = 3, c = 2$ and $d = 0$


$GCD(a, c) = 2$ and $d - b = -3$,

Since 2 does not divide -3, no dependence

```
for (i=1; i<=100; i=i+1) {  
    Y[i] = X[i] / c; /* S1 */  
    X[i] = X[i] + c; /* S2 */  
    Z[i] = Y[i] + c; /* S3 */  
    Y[i] = c - Y[i]; /* S4 */  
}
```

- Dependences:

- True dependences from S1 to S3 and S1 to S4
- Anti-dependence from S1 to S2
- Output dependences from S1 to S4



```
for (i=1; i<=100; i=i+1 {  
    /* Y renamed to T to remove output dependence */  
    T[i] = X[i] / c;  
    /* X renamed to X1 to remove antidependence */  
    X1[i] = X[i] + c;  
    /* Y renamed to T to remove antidependence */  
    Z[i] = T[i] + c;  
    Y[i] = c - T[i];  
}
```

- Drawbacks of dependence analysis:
 - Only references within single loop
 - Only for affine index functions
 - Sparse arrays $x[y[i]]$ are non affine
 - Cannot analyze references made through pointers
 - Different pointers referencing same nodes
 - Doing arithmetic operation on them
 - Limitation in performing inter-procedural analysis
 - Worst case leads to producing expensive but useless information

Eliminating Dependences

- Back substitution
- Copy propagation

DADDUI	R1, R2, #4	→	DADDUI	R1, R2, #8
DADDUI	R1, R1, #4			

- Tree height reduction

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

- Eliminating dependences from recurrences such as

$$sum = sum + x;$$

- Unrolling loop with recurrence:

$$sum = sum + x1 + x2 + x3 + x4 + x5;$$

Five dependent operations

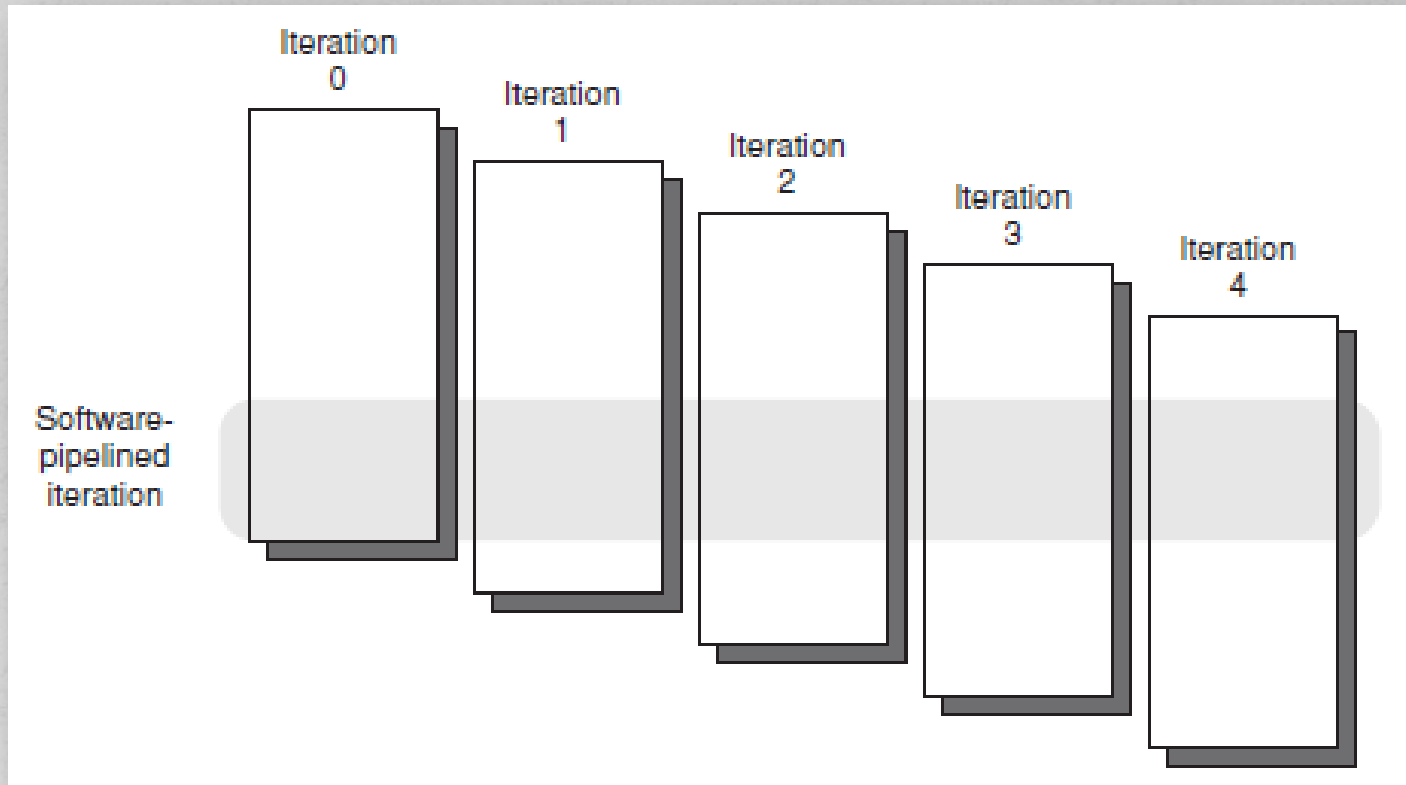
- Optimizing:

$$sum = ((sum + x1) + (x2 + x3)) + (x4 + x5);$$

Three dependent operations

Scheduling and Structuring code for Parallelism

- Loop unrolling to software pipelining and trace scheduling
- Software pipelining:
 - Technique for reorganizing loops
 - Each iteration chosen from different iterations of original loop
 - Interleaves instructions from different loop iterations without unrolling the loop -> dependent computations are separated by an entire loop body



- Some start up code and finish up code at the beginning and end of iteration respectively

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

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

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

Loop
Unrolling

- Software pipelining:
 - Symbolically unrolls loop
 - Selects instruction from each iteration
 - Puts together in a loop

```

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



```

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)
  
```

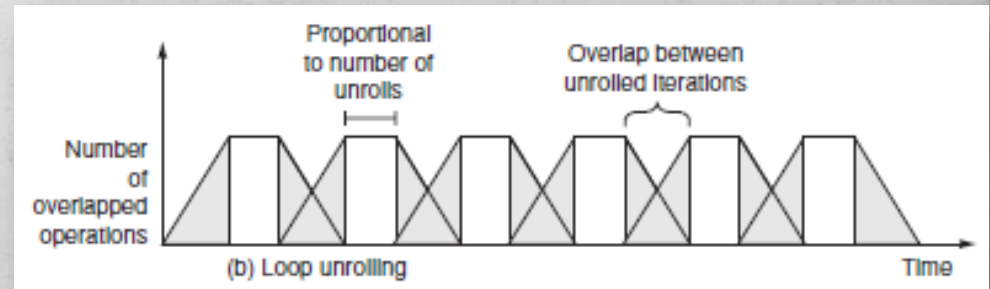
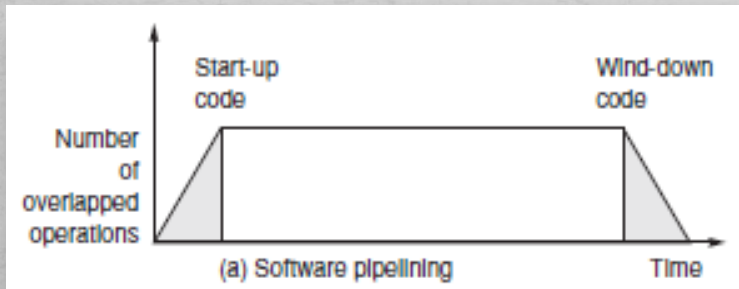


```

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

Software Pipelining Vs Loop Unrolling

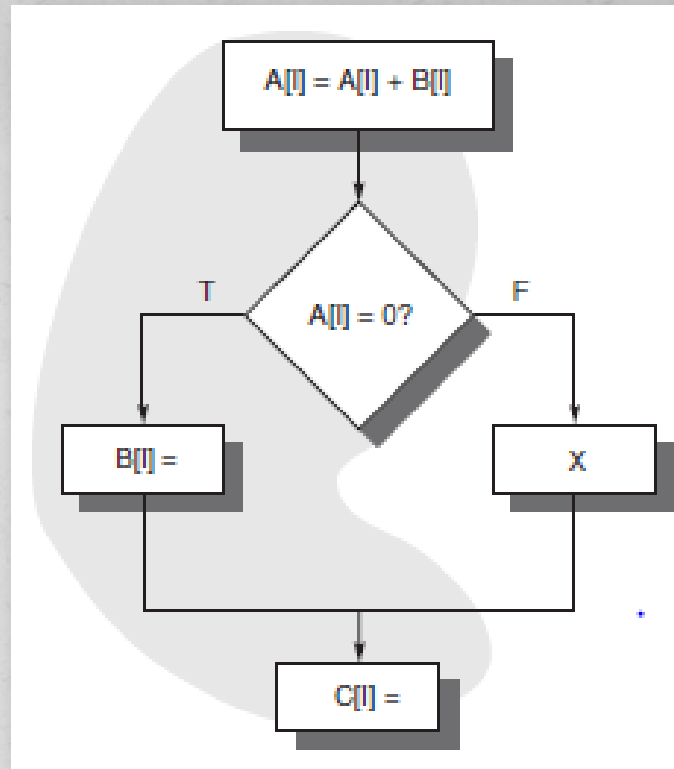
- Software pipelining
 - Reduces code space
 - Reduces time when loop not running at peak speed
 - IA-64 added support but does not eliminate complex compiler support
- Loop unrolling
 - Reduces overhead of loop (branch and counter update)



Global Code Scheduling

- Scheduling becomes difficult when unrolled loops contain internal flow
 - Requires moving instructions across branches -> global code scheduling
- Global code scheduling aims to compact a code fragment with internal control structure into the shortest possible sequence that preserves the data and control dependences
 - Data dependences force partial order
 - Control dependences hinder movement of code

- Removing dependences:
 - Data dependences:
 - Loop unrolling or dependence analysis in case of memory operations
 - Control dependences:
 - Loop unrolling and global code motion, i.e. moving code across branches
 - Global code motion:
 - Requires estimation of the relative frequency of different paths due to the inner loops
 - Important when many inner loops contain conditional statements



- Effective scheduling :
 - Move assignments to B and C before the test of A
 - Movement associated with B is speculative
 - Computation will speed up only if path through B is taken

- Moving B and C without affecting data flow

```

LD    R4,0(R1)    ;load A
LD    R5,0(R2)    ;load B
DADDU R4,R4,R5    ;Add to A
SD    R4,0(R1)    ;Store A
...
BNEZ  R4,elsepart ;Test A
...    ;then part
SD    ...,0(R2)   ;Stores to B
...
J     join        ;jump over else
elsepart: ...    ;else part
X     ;code for X
...
join:  ...        ;after if
SD    ...,0(R3)   ;store C[i]

```

- Using shadow copy of B
- Moving C into then part and a copy into the else part (X)

- Factors taken in to consideration by compiler for computation movement:
 - Frequencies of execution (then case or else case)
 - Cost
 - Change in execution time
 - Comparing different code fragments for movement
 - Cost of compensation for else case

Global code scheduling is extremely complex !

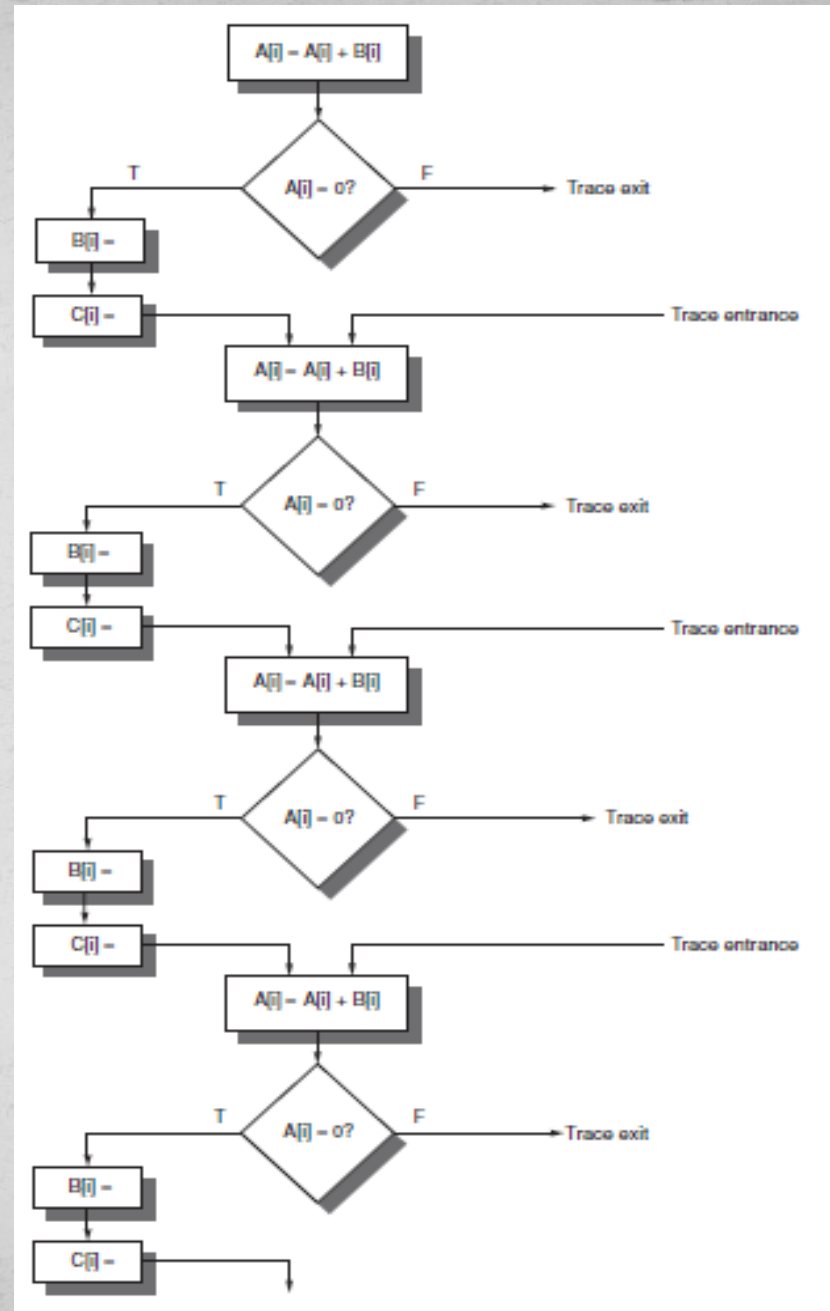
Simpler techniques: Trace scheduling and Superblocks

Trace Scheduling

- Useful for processor with large number of issues per clock
- For processes with unsupported conditional or predicated execution
- Way to organize the global code motion
 - Used only for significant differences in frequency between different paths thus simplifies the decision
- Two steps to trace scheduling
 - Trace selection
 - Trace compaction

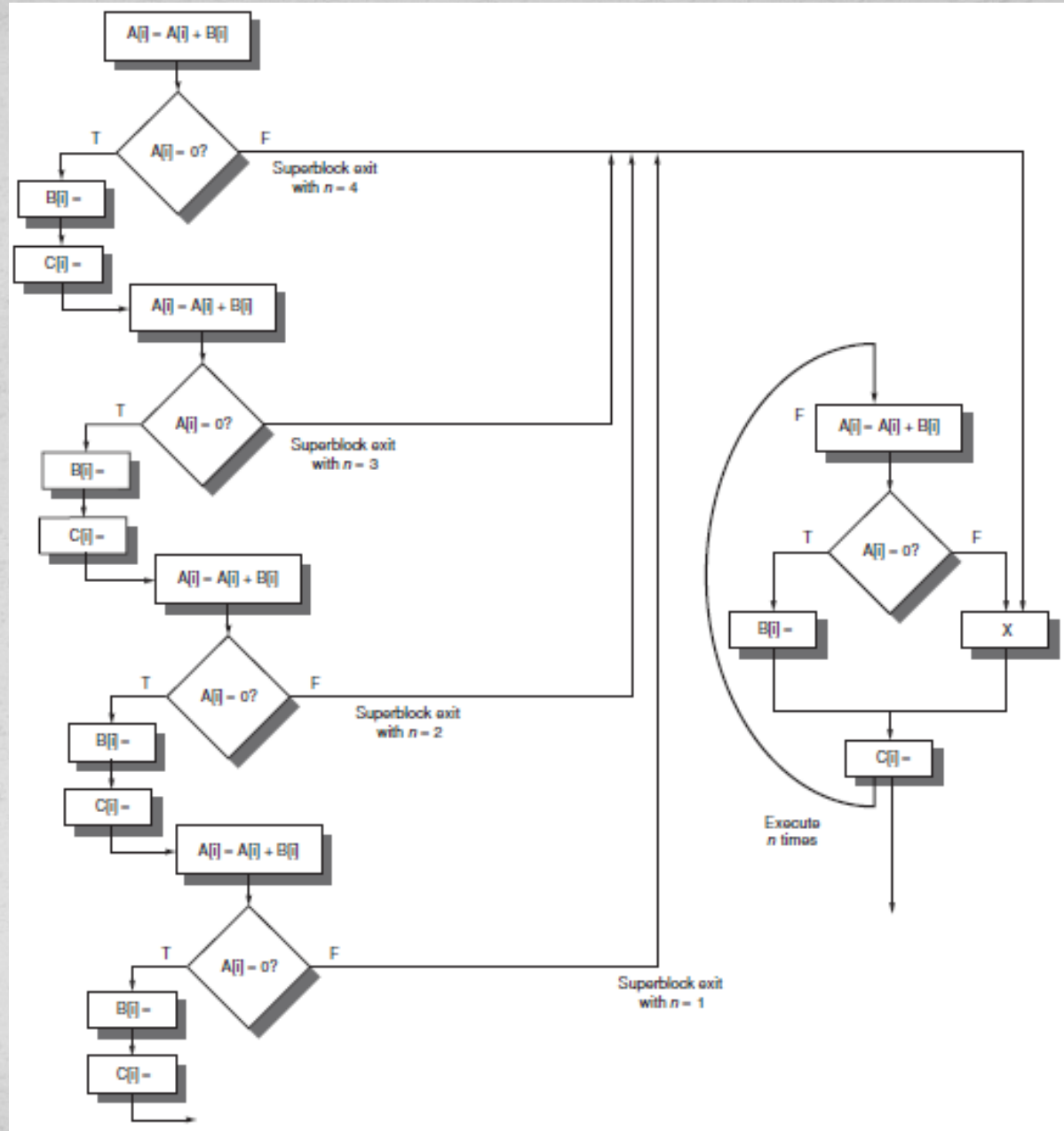
Drawback:

- Entries and exits cause complications
- Requires the compiler to generate and track the compensation code
- Difficult to assess cost of such code



Superblocks

- Similar to traces but extended basic blocks
- Restricted to single entry point but allow multiple exits
 - Tail duplication to create separate block
 - Compacting superblocks is easier
- Reduces complexity of bookkeeping and scheduling than trace method
- May enlarge code size



Hardware support for Exposing Parallelism : *Predicated* Instructions

- Loop unrolling, S/W pipelining, and trace scheduling work well till the branches are fairly *predictable* at *compile time*
- BUT when the branches are not fairly predictable , *parallelism* is *hampered*
- Solution -> *Predicated Instructions*
- Predicated Instructions help eliminate *BRANCHES*
- They convert a *Control* Dependence into a *Data* Dependence and potentially improve *performance*

Concept behind Branch Predication

An instruction refers to a **CONDITION**

- If the condition is true, the instruction is executed normally
- Else, it behaves like a no-op

Example: Consider the given code

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

Assuming: A, S and T rest in R₁, R₂, R₃

R₁ ← A, R₂ ← S, R₃ ← T

BNEZ R₁, L ; /*if R₁ is not equal to zero, jump to L*/

L: ADDU R₂, R₃, R₀ ; /*else Move R₃ to R₂*/

Using *Conditional* move we can implement this statement
in one instruction

CMOVZ R2, R3, R1

Here the conditional instruction converts the *Control*
dependence into *Data* dependence

This transformation is called *if-Conversion* in Vector
Computers

This moves the place to resolve dependence from the front of
the pipeline to the end of the pipeline

- Conditional moves are good useful for *short* sequences, they are *inefficient* to eliminate branches which guard the execution of a *large block of code*

REMEDY: “FULL PREDICATION”

It allows us to convert *large* blocks of Code that are *branch dependent*

Predication is useful with *global code scheduling*, since it can eliminate *non-loop* branches which significantly complicate instruction scheduling

The code sequence below wastes a memory operation slot in the second cycle and will incur a data dependence stall if not taken , since the second LW after the branch depends on prior load

First instruction slot	Second instruction slot
LW R1,40(R2)	ADD R3,R4,R5
	ADD R6,R3,R7
BEQZ R10,L	
LW R8,0(R10)	
LW R9,0(R8)	

Here we call the predicated version load word LWC and assume the load occurs unless third operand is zero. LW -> LWC and is moved up to the second issue slot:

First instruction slot	Second instruction slot
LW R1,40(R2)	ADD R3,R4,R5
LWC R8,0(R10),R10	ADD R6,R3,R7
BEQZ R10,L	
LW R9,0(R8)	

Execution time is reduced by a few cycles since one instruction issue slot is reduced, if the compiler mispredicted the branch, predicated instruction will have no effect on the performance, hence the transformation is *speculative*.

Challenges related to Predicated instructions...

- What happens when a predicated instruction generates an exception...? (i.e. The predication was false.)
- It becomes hard to **IMPLEMENT**, WHY?
- When do you annul the instruction?
- Annulled during execution issue :
 - Requires that the value of 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 efficient?
 - 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 in avoiding potential data hazards
- Factors that hinder 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

Useful only when control dependencies can be completely eliminated by *if*-conversion.

To speculate ambitiously requires three capabilities:

- Ability to find instructions, which with possible register renaming, can be speculatively moved and not affect program data flow
- The ability to ignore exceptions in speculated instructions, until we know that such exceptions really occur
- Ability to exchange loads and stores, stores and stores which may have address conflicts
- The *first* one is a compiler capability, the other two require hardware support, which is next

Hardware Support Preserving Exception Behavior

How can exception behavior be preserved?

- The results of a mispredicted speculated sequence should not be used in final computation.
- Such an instruction should not cause exception.

Four Methods have been investigated for supporting more ambitious speculation.

1. OS and Hardware cooperatively ignore exceptions.
2. Speculated instructions should never raise exceptions. Introduce checks to determine when an exception should occur.
3. POISON bits are attached to the result registers of the instruction which cause exceptions, these bits cause a fault when a normal instruction tries to access the register.
4. A mechanism is provided 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 which force the program to ***terminate***
e.g. Memory protection violation, Illegal operation
- Exceptions that can be handled and resumed
e.g. Page fault
 - Such exceptions can be handled for speculated instructions as for normal instructions.
 - The 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 instructions are executed normally, even for speculated instructions
- Returns an undefined value for exceptions that cause termination
- Lets see an *Example*

if (A == 0) A = B; else A = A+4;

A \longrightarrow O(R₃), B \longrightarrow O(R₂)

```
LD      R1,0(R3)    ;load A
BNEZ    R1,L1       ;test A
LD      R1,0(R2)    ;then clause
J       L2          ;skip else
L1:     DADDI   R1,R1,#4 ;else clause
L2:     SD      R1,0(R3) ;store A
```

How can this code be compiled speculatively...???

Assuming the then clause is almost always executed, and assuming that R14 is unused and available

```
LD      R1,0(R3)    ;load A
LD      R14,0(R2)   ;speculative load B
BEQZ   R1,L3       ;other branch of the if
DADDI  R14,R1,#4   ;the else clause
L3:    SD      R14,0(R3) ;nonspeculative store
```

An alternate approach to do this by using POISON Bits.

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

The Approach taken to accomplish this is as follows.....

- Two bits are added to each register
 - A poison bit
 - Another bit to indicate whether 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 or reset the poison bit

Yet another approach.....

Reorder Buffer (ROB)

- Compiler marks instructions as speculative, also indicating its assumption of taken or not taken
- This information is used by the hardware to find the original location of the speculated instruction
- 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 instruction until:
 - All the branches 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
- This 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 Instructions Computer

- With the advent of IA-64 architecture, EPIC architecture was an evolution of VLIW which has absorbed many of the best ideas from superscalar processors
- EPIC aimed to move the complexity of instruction scheduling from the CPU hardware to the software compiler
- It also used the compiler to find and exploit additional opportunities to enhance ILP
- EPIC indicates parallelism between neighboring instructions
- EPIC has greater support for software speculation as compared to earlier VLIW schemes, which showed minimal support

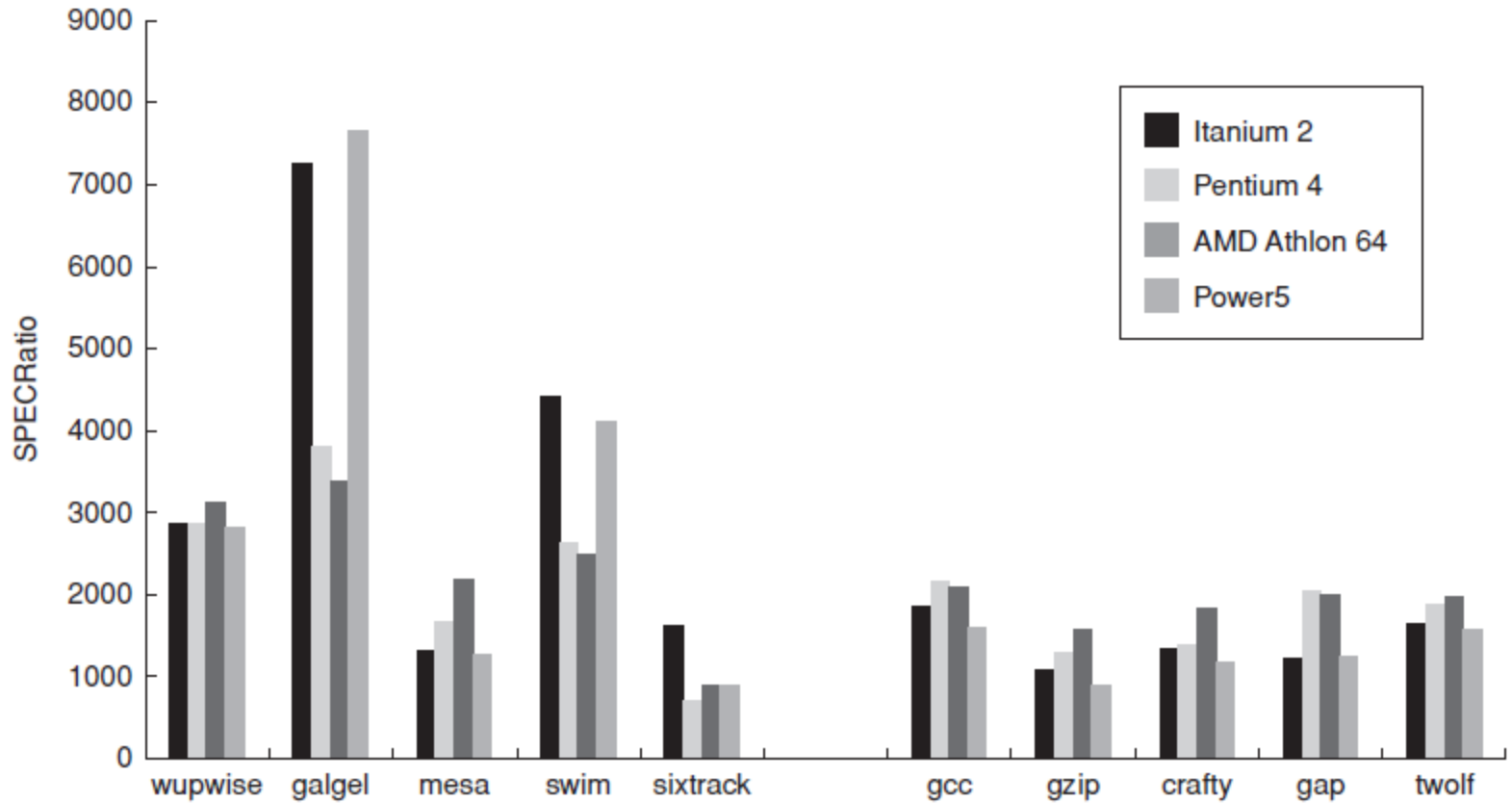
Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- Itanium™ was first implementation (2001)
 - Highly parallel and deeply pipelined hardware at 800 MHz
 - 6-wide, 10-stage pipeline at 800 MHz on 0.18 μ process
- Itanium 2™ is name of 2nd implementation (2005)
 - 6-wide, 8-stage pipeline at 1666 MHz on 0.13 μ process
 - Caches: 32 KB I, 32 KB D, 128 KB L2I, 128 KB L2D, 9216 KB L3

Drawbacks of *EPIC* architecture

- Code Bloat: With EPIC architecture, compile produces big code as compared to the code size on other machines. This increased the number of instruction fetch and also required a larger instruction cache
- Power requirement: Intel Itanium required more power in order to provide the same throughput as compared to the other processors
- Re-compiling: In order to port the existing application into EPIC architecture, we need to re-compile them. This also contributes as one of the disadvantage
- Cache requirement: It was hard to put multiple cores since each core required a cache to work. Also multiple caches for multiple cores would increase the power consumption

Itanium2 Performance



References

1. Computer architecture: a quantitative approach by John L. Hennessy, David A. Patterson, David Goldberg
2. <http://www.cse.unsw.edu.au/~cs9244/06/seminars/02-nfd.pdf>
3. <http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2598>
4. EPIC: An Architecture for Instruction-Level Parallel Processors – hp labs:
<http://www.hpl.hp.com/techreports/92/HPL-92-132.pdf>