Exploiting ILP through Software Approaches

Venkatesh Akella
EEC 270

Winter 2005
Based on Slides from Prof. Al. Davis @ cs.utah.edu

Let the Compiler Do it – Pros and Cons

- Pros
  - No window size limitation, the whole program is there to see
  - Hardware is simple, so can push the clock rate
  - Pragmas and profile information can be used
- Cons
  - Binary code compatibility
  - Basic blocks are small – global code optimization is might hard
  - Lack of run-time knowledge – eg: memory dataflow problems
- Perhaps a mixture of the two?

Overview of the Chapter

- Basic Compiler Techniques for Exposing ILP
  - Loop Unrolling
  - Instruction Scheduling
- Static Branch Prediction
- Static Multiple Issue: VLIW Approach

Overview of the Chapter

- Advanced Compiler Support for Exposing and Exploiting ILP
  - Detecting and Enhancing Loop-Level Parallelism
  - Dependence Analysis
  - Software Pipelining
  - Global Code Scheduling
- Hardware Support for Exposing More Parallelism at Compile Time
  - Predicated Execution
  - Compiler Speculation with HW Support
- CASE STUDY : ITANIUM PROCESSOR

Overview of Software Approaches

Pipeline CPI = Ideal Pipeline CPI + Struct. Stalls + RAW Stalls + WAR Stalls + WAW Stalls + Control Stalls

Technique	Reduces
Loop Unrolling	Control Stalls
Basic Pipeline Scheduling	RAW stalls
Dynamic Branch Prediction	Control Stalls
Issuing multiple instructions per cycle a.k.a Superscalar ==> VLIW	Ideal CPI
Compiler dependence analysis	Ideal CPI and data stalls (RAW, WAW, WAR)
Software pipelining and trace scheduling	Ideal CPI and data stalls
Speculation	data and control stalls
Dynamic memory disambiguation	RAW stalls involving memory

Assumptions for the Examples

- ❑ Default 5-stage MIPS pipeline structure

Instruction Producing Value	Instruction Consuming Value	Intervening Instructions to Avoid Stalls
FP ALU Op	FPU ALU Op	3
FP ALU Op	Store Double	2
Load Double	FP ALU Op	1
Load Double	Store Double	0

Remember these are called latencies in your text

A Simple Loop

Consider adding a scalar *s* to a vector

```
for (i=1, i<=1000, i++)        loop:   L.D       FO, 0(R1)      ;R1 array ptr
      x[i] = x[i] + s                  ADD.D     F4, F0, F2     ;F2 = s
                                       S.D       0(R1), F4      ;put result back
                                                                ;
                                       DADDUI    R1, R1, #-8    decr. R1
                                       BNE       R1, R2, loop   ;again if not done
```

With no scheduling

- ❑ 10 cycles per iteration
  - L.D, load stall, ADD.D, 2 RAW stalls, S.D, DADDUI, RAW stall, BNE, branch delay control stall
- ❑ Let’s try and fill the delayed branch stall

Scheduling the Instructions to minimize Stalls

Non-trivial and many compilers don’t try this

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

Common View:

SD_i depends on $DADDUI_{i-1}$ and therefore can’t be moved

Smarter View:

DADDUI is immed. so solution exists.

- ❑ Note:
  - promote DADDUI move SD to branch delay sl
    - but since it will now be after the SUBI we’ll need an offset of 8
  - remaining stall is for ADD.D to S.D = 2 instructions

Result will run as:

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

necessary dependency need bigger body to improve

40% improvement
10 cycles ==> 6

but still a 3 cycle loop + stall overhead

How to do better?

- Unroll the loop, replicate the loop body and adjust the iteration counter appropriately so that the program semantics remains the same
- What are the advantages?
- Get rid of the ADDUI and BNE overhead
- What is the downside?
- Larger code size – bad for instruction caches
- Needs lots of registers

Basic Idea

- take n loop bodies and catenate them into 1 basic block
- adjust the new termination code
  - Let's say n was 4
  - Then modify the R1 pointer in the example by 4x of what it was before ==> 32 rather than 8 as the immediate value
- savings - 4 BNE's + 4 DADDUI's => just one of each
  - Hence 75% improvement
- problem = still have 4 load stalls per loop

Can we do better?

- don't catenate the unrolled segments - shuffle them instead
  - 4 LD's then 4 ADDD's then 3 SD's, SUBI, BNEZ, then fill the branch delay slot with the final SD
  - VIOLA - no more stalls since LD to dependent ADDD path now has 3 instructions in it.

Result of Unrolling

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

Note: still didn't run out of registers - so we could have improved on this.

16-32 = -16
must compensate for S.D's being after the DADDUI

- ❑ Result
  - 14 cycles for 4 elements ==> 3.5 cycles per element
  - Compared with:
    - 10 cycles with no scheduling
    - 6 cycles per element with scheduling but no unrolling

Caveats of Loop Unrolling

- ❑ 8 more unused register pairs
  - hence
    - we could have gone to an 8 block unroll without register conflict
    - no problem since the 1000 element array would still have broken cleanly (1000/8 = 125)
    - what if it had not - say with remainder R
    - so what just put R blocks (shuffled of course) in front of the Loop then start for real
    - you'll notice that you run out of registers but by cycling through the names you can still remove any stalls in this case
- ❑ Still (most compilers unroll early to expose code for later opt's)
  - there were many things the compiler had to notice to figure this one out (trickiest was the S.D/DADDUI swap - why?)
  - key was the independent nature of each loop body
  - REMEMBER: 3 types of dependence: *data*, *name*, and *control*

How does unrolling help interact with dependencies

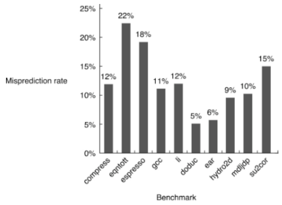
- ❑ Data
  - unrolling provides more independent instructions
  - scheduling then removes RAW dependencies
- ❑ Name
  - renaming removes WAx hazards ==> antidependencies (WAR) and output dependencies (WAW)
- ❑ Control
  - scheduling across branches is very tricking
  - unrolling offers a simple solution
    - more body instructions per control instruction
    - works when iteration count is **static** AND **independent loop bodies**

Static Branch Prediction

- **Static branch behavior is useful for scheduling instructions when the branch delays are exposed by the architecture (eg: delayed branches or canceling branches)**
- **They can also be used to predict which code paths are more plausible which is key for global code optimization**

- **Predict Taken or Not Taken – poor accuracy from 9% to 59% misprediction rates**
- **Backward taken and forward not-taken, context based prediction**
- **Profile information from earlier runs**

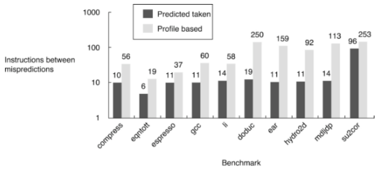
Misprediction Rate of Profile-based predictor on SpEC92



© 2003 Elsevier Science (USA). All rights reserved.

- **Better for FP programs than integer program**
- **Actual performance varies depends on prediction accuracy and branch frequency which varies from 3% to 24%**

Number of instructions between mispredicted branches (log scale)



© 2003 Elsevier Science (USA). All rights reserved.

- **Average = 20 for predict-taken and 110 for profile-based**
- **Compare this to 4-5 instructions between a branch without prediction**
- **The variation is due to the nature of the program and the branch frequency**

VLIW Processors

- Suitable for wide issue. N>4 when superscalar becomes unwieldy
- Assume VLIW with 2 mem references, 2 FP ops, 1 integer/br unit
- Show do all x[I] = x[I] + s
- Unroll as many times as needed to eliminate all stalls

Unrolled 7 times

Mem Ref1	Mem Ref2	FP OP1	FP OP2	Integer
LD F0, 0(R1)	LD F6, -8(r1)			
LD F10, -16(r1)	LD F14, -24(r1)			
LD F18, -32(r1)	LD F22, -40(r1)	Add F4, F0, F2	Add f8, f6, f2	
LD F26, -48(r1)		Add f12, f10, f2	Add f16, f14, f2	
		Add f20, f18, f2	Add f24, f22, f2	
SD F4, 0(r1)		Add f28, f26, f2		
SD F12, -16(r1)				Daddui R1, r1, -56
SD F20, 24(r1)				
Sd F28, 8(r1)				Bne r1, r2, loop

Performance

- 9 cycles for 7 iterations
- Note anything smaller in terms of unrolling will result in empty cycles
- 1.29 cycles per result
- Increases code size - unrolling + packing density in cache is poor (18 empty slots)
- Compression and Encoding is possible
- Register pressure
- Lockstep execution is not efficient - any stall in pipeline causes all instructions to stall
- If you relax this requirement, hw has to do some checking
- Lack of binary compatibility
    - » Dynamic Binary Translation or Emulation
    - » New approaches such as IA64 relaxes the strictness
- VLIW permits simpler/standard memory such as caches as opposed to a Vector Processor

Compiler Techniques to Expose ILP

- ECS 243 EEC175 classes dedicated to this topic
- What can the compiler do?

a) Detecting and Enhancing Loop Level Parallelism
b) Source Level Optimizations
c) Eliminating Dependent Computation
    a) Copy propagation
    b) Symbolic Substitution
    c) Tree Height Reduction
    d) Algebraic Simplification
d) Software Pipelining -
e) Global Scheduling

Loop Carried Dependencies

- LCD = whether data accesses in a particular iteration are dependent on values produced in an earlier iteration
- Eg1 - for (I=1000; I>0, I--)

```
            X[I] = X[I]  + s
```

No LCD

X(I) depends on the value of X[I] in current iteration

Loop Carried dependecies II

```
For (I=1, I<100, I++)
      A[I+1] = A[I] + C[I]    -------- S1
      B[I+1] = B[I] + A[I+1]  ---------S2
```

There are two types of dependencies here:

S1 -> S1 -- LCD   A(I) current iteration needs A(I) from previous iteration!!!!,

So iterations I and I+1 cannot proceed in parallel

S1 -> S2   on A[I+1] is not LCD

LCD III

```
For (I=1, I<100, I++)
   A[I] = A[I] + B(I)   /* Statement S1 */
   B[I+1] = C[I] + D[I]    /* Statement S2 */
```

S1 uses the value produced by S2 in the previous iteration

This is not a cyclic dependency, so you can remove it by source transformation

```
           A(1) = A(1) + B(1)
                For (I=1, I<=99, I++)
                   {     B[I+1] = C[I] + D(I)
                            A[I+1] = A[I+1] + B[I+1]}
               B(101) = C(100) + D(100)
```

Finding Dependecies

- In general HARD problem because of implicit names and arrays and pointers
- General Case is NP-complete
- Restricted cases are solvable in polynomial time
- Eg: Arrays indices that are AFFINE i.e can be expressed as A*I + B where A and B are constants and I is the loop index variable
- The problem can be formulated as checking if 2 affine functions can have the same value I.e.

For some values of a,b,c,d : a[I] +b = c[I] + d

- Apply GCD tests if a,b,c,d are constants
- GCD(c,a) mod (d-b) = 0 --- sufficient condition, not necessary

Software Pipelining

- Symbolic Loop Unrolling
- Loop unrolling is good for uncovering parallelism among instructions by creating longer sequences of st. line code - increasing the window size
- Eg: repeat (n) [A;B;C] is replaced by

  repeat (n/2) [A1; B1; C1; A2; B2; C2]

Reorder instructions to avoid hazards, hence expose ILP - code size increases, some branches are eliminated

Software pipelining is different - it involves reorganizing a loop with instructions from different iterations:

A; (repeat (n-1) [BCA]) B; C

B, C, A are instructions from DIFFERENT iterations of the loop

Software Pipelining

A; (repeat (n-1) [BCA]) B; C

Prolog New Loop Epilog

B, C, A are instructions from DIFFERENT iterations of the loop

This does in SW what Tomasulo's algorithm did in HW - interleaving instructions from different iterations of a loop and executing them in parallel

Prolog and Epilog code is to make sure that the same code is executed in both the cases I.e. semantics of the program are not altered.

A Concrete Example

```
Loop:LD F0, 0(R1)
     ADD F4, F4, F2
     SD   F4, 0(R1)
     DADDUI R1, R1, -8
     BNE R1, R2, LOOP
```

PREAMBLE

```
 LD F0, 0(R1)
 STALL
 ADD F4, F0, F2
 LD   F0, -8(R1)
 DADDUI R1, R1, -8
Loop:SD F4, 16(R1) -- stores M[I]
     ADD F4, F0, F2 --- adds M[I-1]
     LD   F0, 0(R1) ----  loads M[I-2]
     DADDUI R1, R1, -8
     STALL
     BNE R1, R2, LOOP
     STALL
```

Instructions not executed
In the final 2 iterations →

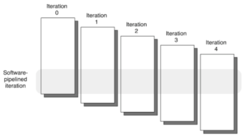
EPILOG

```
 ADD F4, F4, F2
 SD   F4, 8(R1)
 STALL
 STALL
 SD F4, 8(R1)
```

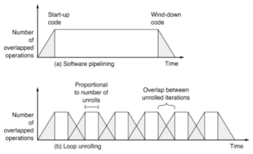
Loop Unrolling vs SW Pipelining

- SW is symbolic so does not increase the code size.
- Does not increase register pressure, however register management is tricky - lifetimes of registers values can be long
- Loop unrolling and SW pipelining attack different problems so in general they can be combined
- LU - reduces loop overhead I.e. branch and counter update
- SWP - reduces the time when the loop is not running at its peak

Figuratively Speaking



© 2003 Elsevier Science (USA). All rights reserved.



(a) Software pipelining

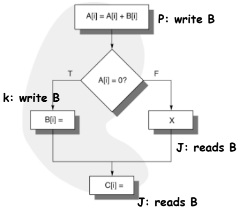
(b) Loop unrolling

© 2003 Elsevier Science (USA). All rights reserved.

Going Beyond Basic Blocks

Loop Body

```
a[I] = a[I] + b[I];
     If (a(I)==0)
B[I] = XXX  Else X;
     C[I] = YYY;
```



© 2003 Elsevier Science (USA). All rights reserved.

What are the consequences of moving instruction K before the branch?

Instr J is dependent on K instead of P

If branch is not taken C[I] will get wrong value

-- Add compensating code

-- keep copy of B

Welcome to Speculation!!

Static Approaches to speculation:

a) Global Scheduling: Trace Scheduling, Superblocks

b) Predicated Execution

Global Scheduling - Fisher 1984

- Going Beyond BASIC BLOCKS
- Exploit Parallelism ACROSS BB I.e. inter BB parallelism as opposed to INTRA BB parallelism with Loop Unroll and SWP
- Useful when there is no support for predication and loop unrolling doesn't help because of conditional branches inside the loop body
- Pick a trace - a sequence of basic blocks (called trace) that is executed frequently
- Trace Selection can be done by profiling
- Loop unrolling and static branch prediction can be used
- Trace Compaction - Squeeze the trace into a small number of VLIW instructions by moving instructions as early as possible - packing with constraints
- Add Compensation Code for paths that will be taken if the prediction is wrong

---

Predicated Execution -A Form of Speculation

- Works when static branch prediction is poor
- Wide issue m/c when multiple branches have to be resolved
- ILP is limited due to control dependencies
- Convert control dependency into data dependency - predicated/cond instr
- This helps both hw based scheduling and SW pipelining and global scheduling
- Every instruction refers to a cond.
- If the condition is TRUE execute the instruction normally, if it is FALSE it becomes a NOP
-

Example

- Example, if (A==0) S=T …. Conditional move

Assume R1 = A; R2 = S and R3 = T

```
BNEZ R1, L
ADDU R2, R3, R0
L:
```

```
CMOVZ R2, R3, R1
```

- Another Example, A = abs (A) …. If B<0 A = -B else A = B
- Full Predication - Every instruction is controlled by a predicate, it allows to convert large blocks of code that are branch dependent
- Useful for global sceduling

WRINKLES

- Predicated Execution cannot generate an exception if the predicate is FALSE
- Annuled instructions consume valuable resources like FU, fetch bW
- Wastes power
- What happens if something depends on multiple branches - you need multiple conditions and that makes it complex

---

HW Support to ASSIST Speculation

- Hw and OS cooperatively ignore exceptions for speculative instructions - this approach preserves exception behavior for CORRECT programs but not for INCORRECT ones
- Speculative instructions that never raise exceptions are used and checks made to detect when exceptions can occur
- Poison bits in the result register - to invalidate results when a speculative instr causes an exception

when a normal instruction tries to read the if a poison bit is set it will result in a fault

- Use something like a reorder buffer

Example - 1

If A== 0 A = B; else A = A + 4

A is at 0(R3) and B is at 0(r2)

```
LD R1, 0(R3)      ---  load A
BNEZ R1, L1     --- test A
LD R1, 0(R2)     --- R1 gets B
J L2      --- skip else clause
L1: DADDI R1, R1, 4 -- else
L2: SD R1, 0(r3) --  store A
```

Assume the THEN clause is almost always Executed:

SPECULATIVE CODE

```
LD R1, 0(R3)      -- load A
LD R14, 0(R2) --- speculative load B
BEQZ R1, L3 --- other branch of if
DADDi R14, R1, 4 -- the else clause
L3: SD   R14, 0(R3) -- non spec store
```

R14 is a temporary register to keep a copy Of A in case branch is not taken A will be in r14 after the code

Exception Handling Rule:

HW and OS simply handle all resumable exceptions, when the exceptions occur both for spec and non spec instructions - yes extra work for non spec instructions but works

For terminating exceptions like protection faults, return an undefined value which is then check whether it comes from spec or no spec instr and dealt with appropriately.

Another Scheme to preserve exception behavior

Special versions of Instructions are used in the speculative mode - these do not generate terminating exception

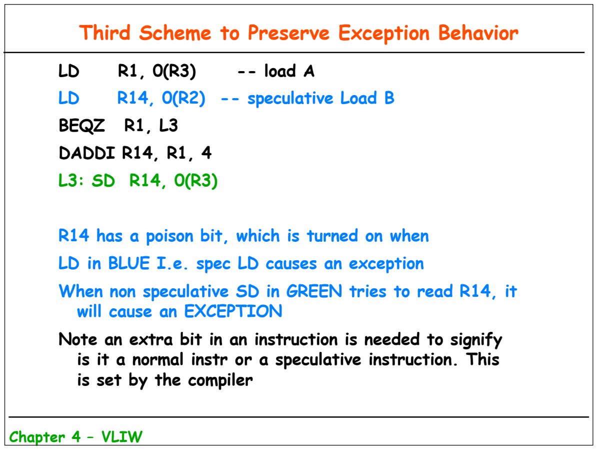
Eg; sLD R14, 0(R2)

- New instructions to check for such exceptions

Eg: SPECCK 0(r2) --- perform speculation check.

So, with these instruction the previous code becomes:

```
Ld  R1, 0(r3) -- load A
sLD  R14, 0(R2) -- speculative load, no termination
Bnez R1, L1     ;; test A
SPECCk 0(r2)  -- perform a check if exception has occurred
J L2
L1: Daddi R14, R1, #4
L2: SD R14, 0(R3)    -- store A
```



Third Scheme to Preserve Exception Behavior

```
LD      R1, 0(R3)       -- load A
LD      R14, 0(R2)  -- speculative Load B
BEQZ   R1, L3
DADDI R14, R1, 4
L3: SD  R14, 0(R3)
```

R14 has a poison bit, which is turned on when

LD in BLUE I.e. spec LD causes an exception

When non speculative SD in GREEN tries to read R14, it will cause an EXCEPTION

Note an extra bit in an instruction is needed to signify is it a normal instr or a speculative instruction. This is set by the compiler