Computer Science 146
Computer Architecture

Fall 2019

Harvard University

Instructor: Prof. David Brooks

dbrooks@eecs.harvard.edu

Lecture 11: Software Pipelining and Global Scheduling

Lecture Outline

- Review of Loop Unrolling
- Software Pipelining
- Global Scheduling
  - Trace Scheduling, Superblocks
- Next Time
  - Hardware-Assisted, Software ILP
    - Conditional, Predicated Instructions
    - Compiler Speculation with Hardware Support
  - Hardware vs. Software comparison
  - Itanium Implementation

Compiler Loop Unrolling

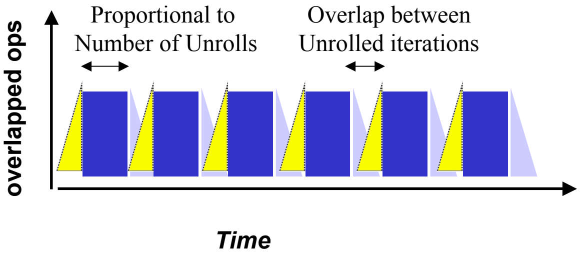
1. Check OK to move the S.D after DSUBUI and BNEZ, and find amount to adjust S.D offset
2. Determine unrolling the loop would be useful by finding that the loop iterations were independent
3. Rename registers to avoid name dependencies
4. Eliminate extra test and branch instructions and adjust the loop termination and iteration code
5. Determine loads and stores in unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent
   - requires analyzing memory addresses and finding that they do not refer to the same address.
6. Schedule the code, preserving any dependences needed to yield same result as the original code

Loop Unrolling Limitations

- Decrease in amount of overhead amortized per unroll
  - Diminishing returns in reducing loop overheads
- Growth in code size
  - Can hurt instruction-fetch performance
- Register Pressure
  - Aggressive unrolling/scheduling can exhaust 32 register machines

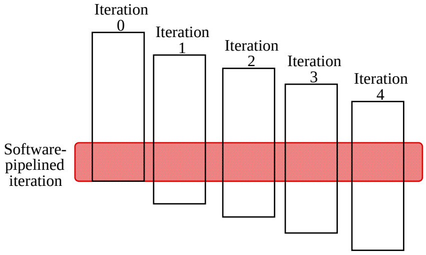
Loop Unrolling Problem

- Every loop unrolling iteration requires pipeline to fill and drain
- Occurs every *m*/*n* times if loop has *m* iterations and is unrolled *n* times


Proportional to Number of Unrolls
Overlap between Unrolled iterations
overlapped ops
Time

More advanced Technique: Software Pipelining

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (~ Tomasulo in SW)

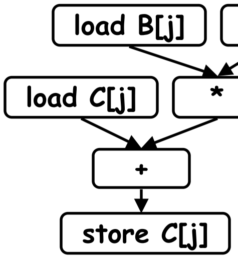


Software Pipelining

- Now must optimize inner loop
- Want to do as much work as possible in each iteration
- Keep all of the functional units busy in the processor

```
for(j = 0; j < MAX; j++)
  C[j] += A * B[j];
```

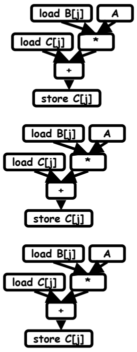
Dataflow graph:



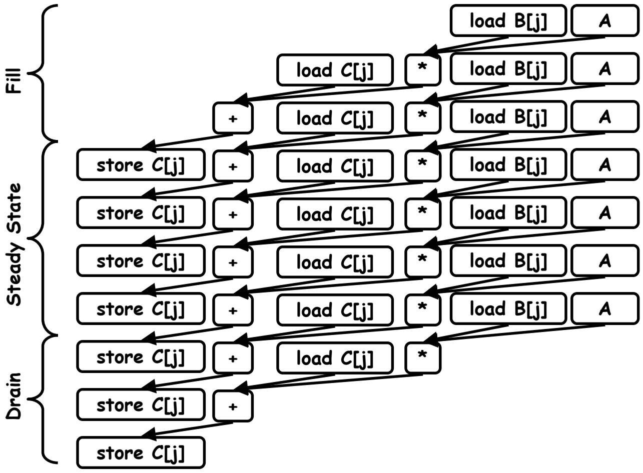
Software Pipelining Example

```
for(j = 0; j < MAX; j++)
  C[j] += A * B[j];
```

Not pipelined:



Pipelined:



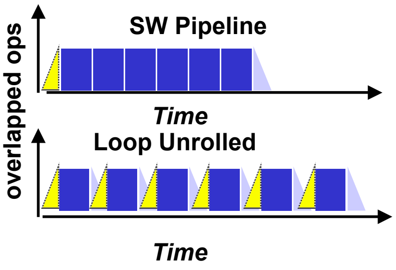
Software Pipelining Example

Before: Unrolled 3 times

```
L.D       F0,0(R1)
ADD.D     F4,F0,F2
S.D       0(R1),F4
L.D       F6,-8(R1)
ADD.D     F8,F6,F2
S.D       -8(R1),F8
L.D       F10,-16(R1)
ADD.D     F12,F10,F2
S.D       -16(R1),F12
DSUBUI    R1,R1,#24
BNEZ      R1,LOOP
```

After: Software Pipelined

```
S.D     0(R1),F4 ; Stores M[i]
ADD.D   F4,F0,F2 ; Adds to M[i-1]
L.D     F0,-16(R1);Loads M[i-2]
DSUBUI  R1,R1,#8
BNEZ    R1,LOOP
```



- **Symbolic Loop Unrolling**
  - **Maximize result-use distance**
  - **Less code space than unrolling**
  - **Fill & drain pipe only once per loop vs. once per each unrolled iteration in loop unrolling**

5 cycles per iteration

Software Pipelining vs. Loop Unrolling

- Software pipelining is *symbolic* loop unrolling
  - Consumes less code space
- Actually they are targeting different things
  - Both provide a better scheduled inner loop
  - Loop Unrolling
    - Targets loop overhead code (branch/counter update code)
  - Software Pipelining
    - Targets time when pipelining is filling and draining
  - Best performance can come from doing both

When Safe to Unroll Loop?

- Example: Where are data dependencies? (A,B,C distinct & nonoverlapping)

```
for (i=0; i<100; i=i+1) {
      A[i+1] = A[i] + C[i];    /* S1 */
      B[i+1] = B[i] + A[i+1];  /* S2 */
}
```

1. S2 uses the value, A[i+1], computed by S1 in the same iteration.

2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].
This is a "loop-carried dependence": between iterations

- For our prior example, each iteration was distinct
- Implies that iterations can't be executed in parallel?

VLIW vs. SuperScalar

- Superscalar processors decide on the fly how many instructions to issue
  - HW complexity of Number of instructions to issue $O(n^2)$
- Proposal: Allow compiler to schedule instruction level parallelism explicitly
- Format the instructions in a potential issue packet so that HW need not check explicitly for dependences

VLIW: Very Large Instruction Word

- Each “instruction” has explicit coding for multiple operations
  - In IA-64, grouping called a “packet”
- Tradeoff instruction space for simple decoding
  - Slots are available for many ops in the instruction word
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches (Discussed next time)

Recall: Unrolled Loop that Minimizes Stalls for Scalar

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

L.D to ADD.D: 1 Cycle
ADD.D to S.D: 2 Cycles

14 clock cycles, or 3.5 per iteration

Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
L.D F0,0(R1)	L.D F6,-8(R1)				1
L.D F10,-16(R1)	L.D F14,-24(R1)				2
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		3
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2		4
		ADD.D F20,F18,F2	ADD.D F24,F22,F2		5
S.D 0(R1),F4	S.D -8(R1),F8	ADD.D F28,F26,F2			6
S.D -16(R1),F12	S.D -24(R1),F16				7
S.D -32(R1),F20	S.D -40(R1),F24	DSUBUI R1,R1,#48			8
S.D -0(R1),F28				BNEZ R1,LOOP	9

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW (15 vs. 6 in SS)

Software Pipelining with Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
L.D F0,-48(R1)	ST 0(R1),F4	ADD.D F4,F0,F2			1
L.D F6,-56(R1)	ST -8(R1),F8	ADD.D F8,F6,F2		DSUBUI R1,R1,#24	2
L.D F10,-40(R1)	ST 8(R1),F12	ADD.D F12,F10,F2		BNEZ R1,LOOP	3

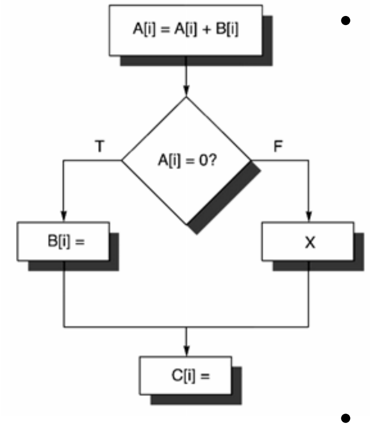
- Software pipelined across 9 iterations of original loop
  - In each iteration of above loop, we:
    - Store to m,m-8,m-16 (iterations I-3,I-2,I-1)
    - Compute for m-24,m-32,m-40 (iterations I,I+1,I+2)
    - Load from m-48,m-56,m-64 (iterations I+3,I+4,I+5)
- 9 results in 9 cycles, or 1 clock per iteration
- Average: 3.3 ops per clock, 66% efficiency

Note: Need fewer registers for software pipelining
(only using 7 registers here, was using 15)

Global Scheduling

- Previously we focused on loop-level parallelism
  - Unrolling, Software Pipelining + scheduling work well
  - These work best on single basic blocks (repeatable schedules)
    - Basic Block – Single Entry/Single Exit Instruction Sequence
  - What about internal control flow?
  - What about if-branches instead of loop-branches?

Global Scheduling



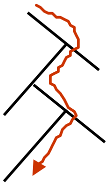
- How to move computation and assignment of B[i]?
  - Relative execution frequency?
  - How cheap to execute B[i] above the branch?
  - How much benefit to executing B[i] early? (critical path?)
  - What is the cost of compensation code for the “else” case?
- What about moving C[i]?

Static Branch Prediction

- Simplest: Predict taken
  - Misprediction rate = untaken branch frequency => for SPEC programs is 34%.
  - Range is quite large though (from not very accurate (59%) to highly accurate (9%))
- Predict on the basis of branch direction? (P6 on BTB miss)
  - choosing backward-going branches to be taken (loop)
  - forward-going branches to be not taken (if)
  - SPEC programs, however, most forward-going branches are taken => predict taken is better
- Predict branches on the basis of profile information collected from earlier runs
  - Misprediction varies from 5% to 22%

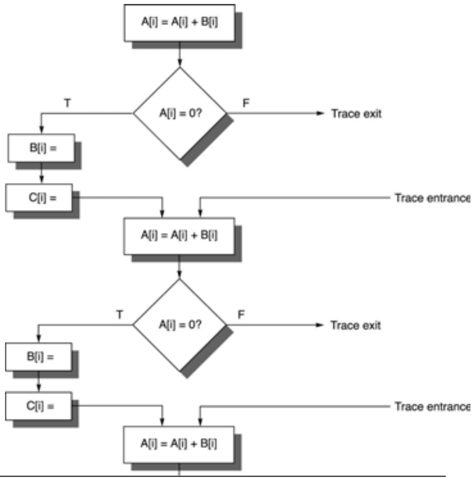
Trace Scheduling

- Parallelism across IF branches vs. LOOP branches?
- Two steps:
  - *Trace Selection*
    - Find likely sequence of basic blocks (*trace*) of (statically predicted or profile predicted) long sequence of straight-line code
  - *Trace Compaction*
    - Squeeze trace into few VLIW instructions
    - Need bookkeeping code in case prediction is wrong
- This is a form of compiler-generated speculation
  - Compiler must generate “fixup” code to handle cases in which trace is not the taken branch
  - Needs extra registers: undoes bad guess by discarding



Trace Scheduling

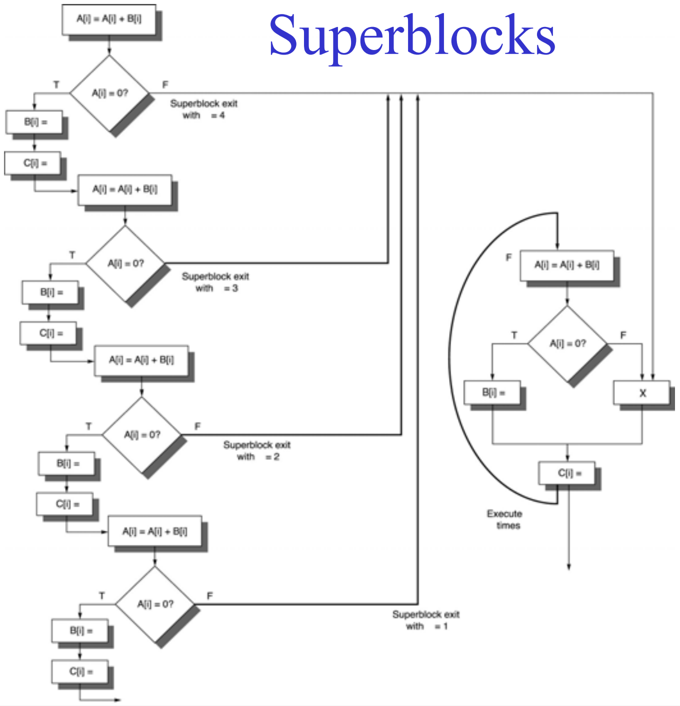
- Use loop unrolling, static branch prediction to generate long traces
- Trace scheduling:
  - Bookkeeping code is needed when code is moved across trace entry and exit points



Superblocks

- Fixes a major drawback of trace scheduling
  - Entries and exits in the middle of the trace are complicated
- Superblocks
  - Use a similar process as trace generation, but superblocks are restriced to a *single* entry point with multiple exit points
  - Scheduling (compaction) is simpler
    - Only code motion across exits must be considered
    - Only one entrance?
      - Tail duplication is used to create a separate block that corresponds to the portion of the trace after entry

Superblocks



What if branches are not statically predictable?

- Loop Unrolling, Trace scheduling work great when branches are fairly predictable statically
- Same thing with memory reference dependencies
- Compiler Speculation is needed to solve this
  - Conditional/Predicated instructions “if-conversion”
  - Hardware support for exception/memory-dependence checks

Hardware Support for Exposing More Parallelism at Compile-Time

- Conditional or Predicated Instructions
  - Conditional instruction execution
- Full predication – every instruction has predicate tag (IA64)
- Conditional Moves (Alpha, IA32, etc)

```
                  if(r3==0)r1=r2
   BNEZ R3, L              |      cmoveqz r1, r2, r3
   ADDU R1, R2, R0         |
L:                         |
```

Schedule for next few lectures

- Next Time (Mar. 17th) – HW#3 Due Friday
  - Hardware support for software-ILP
  - Itanium (IA64) case study
- Review for midterm (Mar 22nd)
- Midterm March 24th