

Compiler Techniques for ILP



Agenda

- Outline of topics
 - Pipeline scheduling and loop unrolling
 - Static branch prediction
 - Multiple issue: VLIW
 - Loop dependence analysis (gcd test)
 - Software pipelining
 - Trace scheduling
 - Predication
 - Exception behavior (control speculation)
 - Memory reference speculation (data speculation)

MAINAK CS422

2

Pipeline scheduling

- Static scheduling of code to reduce stalls
 - Compiler should have good knowledge about the pipe e.g. functional units, latency of each, bypass structure, etc.

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

Assume x is a vector of doubles, s is a double scalar, r1 is initialized to the address of x[1000], r2 holds the address of x[1], f2 is loaded with -s. MIPS assembly code:

```
Loop:  load.d    f0, 0(r1)
      add.d    f4, f0, f2
      store.d  f4, 0(r1)
      addui   r1, r1, -8
      bne     r1, r2, Loop
```

3



Pipeline scheduling

- Pipeline latencies
 - One load delay slot
 - One branch delay slot, phased branch computation
 - Floating-point adder is pipelined, but has latency of 4 cycles i.e. fp add pipe is IF ID EX1 EX2 EX3 EX4 MEM WB
 - Pipeline is equipped with full bypass

```
Loop:  load.d    f0, 0(r1)
      stall
      add.d    f4, f0, f2
      stall
      store.d  f4, 0(r1)
      addui   r1, r1, -8
      bne     r1, r2, Loop
      stall
```

9 cycles/iter.

MAINAK CS422

4



Pipeline scheduling

- Improved schedule
 - Proving the legality of motion of store to the delay slot is non-trivial: needs symbolic optimization
 - Observations
 - 2 out of 5 instructions come from loop overhead (addui, bne)
 - Two different iterations are completely independent

```
Loop:  load.d    f0, 0(r1)
      addui   r1, r1, -8
      add.d    f4, f0, f2
      stall
      bne     r1, r2, Loop
      store.d  f4, 8(r1)
```

6 cycles/iter.

MAINAK CS422

5



Loop unrolling

- Overlap different iterations if they are independent
 - Unroll the loop k times
 - Our example with k=4
- ```
for(i=1000; i>0; i-=4) {
 x[i] -=s; x[i-1] -= s; x[i-2] -= s; x[i-3] -= s;
}
```
- Useful only if iterations are independent; otherwise it only increases code size
  - Reduces loop overhead: now for 4 computations we have one loop overhead
  - Increases straight line code (length of basic block): useful for processors without good branch predictors
  - Compare this with speculative execution: with a good branch predictor and a big enough issue queue, a dynamic scheduler will automatically discover cross-iteration parallelism



## Loop unrolling

- Must use different registers for different iterations to avoid unnecessary name dependence
  - Useful if register renaming is not implemented

```

Loop: load.d f0, 0(r1)
 add.d f4, f0, f2
 store.d f4, 0(r1)
 load.d f6, -8(r1)
 add.d f8, f6, f2
 store.d f8, -8(r1)
 load.d f10, -16(r1)
 add.d f12, f10, f2
 store.d f12, -16(r1)
 load.d f14, -24(r1)
 add.d f16, f14, f2
 store.d f16, -24(r1)
 addui r1, r1, -32
 bne r1, r2, Loop

```

MAINAK CS422 7

## Loop unrolling

- Scheduled code

```

Loop: load.d f0, 0(r1)
 load.d f6, -8(r1)
 load.d f10, -16(r1)
 load.d f14, -24(r1)
 add.d f4, f0, f2
 add.d f8, f6, f2
 add.d f12, f10, f2
 add.d f16, f14, f2
 store.d f4, 0(r1)
 store.d f8, -8(r1)
 store.d f12, -16(r1)
 addui r1, r1, -32
 bne r1, r2, Loop
 store.d f16, 8(r1)

```

- 3.5 cycles/iter
- Loop overhead: 2 out of 14 ins
- Increased register pressure

MAINAK CS422 8

## Loop unrolling

- What does the compiler need to do?
  - Prove legality of moving store.d after addui and bne
  - Compute gain of loop unrolling by proving independence of iterations
  - Allocate free registers to different iterations
  - Club together all loop overhead instructions into two
  - Take into account structural hazards while scheduling the unrolled code
  - Prove the legality of boosting load.d above store.d; this involves the so-called *alias analysis* which tries to prove independence of memory addresses
  - Prove correctness of the final schedule: this must be equivalent to the original code

MAINAK CS422 9

## Loop unrolling

- How much to unroll?
  - Depends on processor resources
  - Unroll factor may not exactly divide iteration count
    - Generate two loops: one with  $n/k$  iterations and one with  $n/k$  iterations
    - For large  $n$  and relatively small  $k$ , the execution time of the second loop will dominate

MAINAK CS422 10

## Static branch pred.

- A well-established compiler technique
  - Normally used for two purposes
    - Help scheduling
    - Pass hints to the dynamic predictor
  - In both cases the basic requirement is to be able to predict the direction of a branch at compile time
  - We have discussed one such scheme: FNBT
  - Nullifying branches help the compiler do aggressive static prediction
  - Possible to use states of various resources at compile time to generate a feature vector for static prediction
  - Profile-guided (from past sample runs) static prediction is used to give useful feedback to the predictor and train it
  - Dynamic predictors (even the sophisticated ones) cannot predict loop termination correctly: compiler can pass useful hints

11

## VLIW

- Very long instruction word processors
  - Compiler prepares packets of multiple independent instructions (called a long instruction)
  - Compiler also takes care of necessary hazard stalls and accordingly inserts empty packets
  - Hardware for dynamic scheduling is not needed
  - Processor simply issues the packet to the functional units as and when the packets arrive
  - Compiler usually unrolls loops to find enough parallelism to fill the packets
  - Local scheduling: find parallelism within a basic block
  - Global scheduling: find parallelism across basic blocks
  - Let's assume a VLIW core with five instructions per packet: two memory operations, two fp operations, one integer operation

MAINAK CS422 12

## VLIW

- VLIW packets for  $x[i] = s$  (unrolled 7 times)
 

```
load.d f0, 0(r1) | load.d f6, -8(r1) | NOP | NOP | NOP ||
load.d f10, -16(r1) | load.d f14, -24(r1) | NOP | NOP | NOP ||
load.d f18, -32(r1) | load.d f22, -40(r1) | add.d f4, f0, f2 |
add.d f8, f6, f2 | NOP ||
load.d f26, -48(r1) | NOP | add.d f12, f10, f2 | add.d f16,
f14, f2 | NOP ||
NOP | NOP | add.d f20, f18, f2 | add.d f24, f22, f2 | NOP ||
store.d f4, 0(r1) | store.d f8, -8(r1) | add.d f28, f26, f2 | NOP |
NOP ||
store.d f12, -16(r1) | store.d f16, -24(r1) | NOP | NOP |
addui r1, r1, -56 ||
store.d f20, 24(r1) | store.d f24, 16(r1) | NOP | NOP | bne
r1, r2, Loop ||
store.d f28, 8(r1) | NOP | NOP | NOP | NOP ||
```



MAINAK CS422

13

## VLIW

- Shortcomings of VLIW
  - Code size inflation: results from aggressive loop unrolling and long instruction words (still fixed length encoding and hence wastage)
  - Lockstep execution of packets: caches pose a big problem due to non-deterministic latency
  - Hardware interlock needed to stall the entire pipe at the time a variable latency instruction issues
  - Binary compatibility: code compiled for a VLIW processor will not run correctly on a newer processor if functional unit latency, bypass structure, etc. change



MAINAK CS422

14

## Solutions: EPIC

- Explicit Parallel Instruction Computer
  - Have "stop bits" in the instruction words to denote exactly where parallelism ends and a new "atom" starts
  - Packets and instruction words are no longer tied together: helps reduce code size inflation
  - Have some minimal hardware to detect structural hazards: eliminates the need for lockstep issue
  - Let the compiled code rely less on the latency of functional units and bypass structures: introduces flexibility in terms of binary compatibility
  - In a nutshell, expose exactly as many features of the processor as needed; don't overdo it
  - Instead of an all-software approach, it is really a hardware/software co-design
  - Will discuss itanium later
  - Still the major question remains: can the compiler extract enough parallelism? (need some support from hardware)



MAINAK CS422

15

## Loop dependence

- Within and across iteration dependence analysis
 

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

True dependence: S1 to S2 (within iteration), S1 to S1 (loop carried), S2 to S2 (loop carried)

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

True dependence: S2 to S1; no cycle in dependence graph  
Means the loop can be executed in parallel via some transformation



MAINAK CS422

16

## Loop dependence

- Converting loop-carried dependence to intra-iteration dependence
 

```
A[0] = A[0] + B[0];
for(i=0; 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];
```
- Dependence distance bigger than 1 is good
 

```
Y[i] = Y[i-1] + Y[i]; // dependence distance = 1
Y[i] = Y[i-5] + Y[i]; // can overlap five iterations
```
- Dependence analysis is usually inexact due to hardness of alias analysis



MAINAK CS422

17

## Dependence test

- Formulating the problem
  - Affine array index:  $a_0 \cdot i_0 + a_1 \cdot i_1 + \dots + a_k \cdot i_k + b$
  - Cannot handle index via indirection vector e.g.  $A[B[i]]$  simply because the value of  $B[i]$  may not be known at compile time
  - True dependence test (same for anti and out)
    - Loop body writes to  $X[a_0 \cdot i_0 + a_1 \cdot i_1 + \dots + a_k \cdot i_k + b]$  and reads from  $X[c_0 \cdot i_0 + c_1 \cdot i_1 + \dots + c_k \cdot i_k + d]$
    - Two iteration points:  $l_r$  and  $l_w$  such that  $l_w < l_r$  &&  $l_r \leq l_w$
    - $a_0 \cdot i_w + a_1 \cdot i_w + \dots + a_k \cdot i_w + b = c_0 \cdot i_r + c_1 \cdot i_r + \dots + c_k \cdot i_r + d$
    - Boils down to solving an integer linear system of inequalities (replace equalities by pair of inequalities): NP-complete in general (integer Gaussian elimination, Fourier-Motzkin elimination, Omega test, ...)




MAINAK CS422

18

## Inexact tests

- Banerjee test
  - Based on Intermediate Value Theorem
  - Compute the maximum and minimum values that  $(a_0 \cdot iw_0 - c_0 \cdot ir_0) + (a_1 \cdot iw_1 - c_1 \cdot ir_1) + \dots + (a_k \cdot iw_k - c_k \cdot ir_k)$  can attain
  - If  $d-b$  is outside this range there is no solution; otherwise there is definitely a real solution by IVT, but don't know if it is an integer solution for  $lw, lr$
- GCD test
  - If  $\gcd(a_0, c_0, a_1, c_1, \dots, a_k, c_k)$  divides  $d-b$  there is a solution
  - Proof?

Multi-dimensional arrays?



MAINAK CS422 19

## Software pipelining


- Pick independent operations from different iterations

```

Loop: load.d f0, 0(r1)
 add.d f4, f0, f2
 store.d f4, 0(r1)
 addui r1, r1, -8
 bne r1, r2, Loop

```

- Basic idea is to pick the load of  $i$ th iteration, add of  $(i-1)$ th iteration, and store of  $(i-2)$ th iteration and put them in a single iteration



MAINAK CS422 20

## Software pipelining

- Software pipelined loop

```

Loop: store.d f4, 16(r1)
 add.d f4, f0, f2
 load.d f0, 0(r1) // can put in BD slot
 addui r1, r1, -8
 bne r1, r2, Loop


```

- Need start-up (prolog) and wind-up (epilog) codes

```

Prolog: load.d f0, 0(r1)
 stall
 add.d f4, f0, f2
 load.d f0, -8(r1)
 addui r1, r1, -16 // can fill the stall slot

```



MAINAK CS422 21

## Software pipelining


- Epilog

```

store.d f4, 8(r1)
add.d f4, f0, f2
store.d f4, 0(r1)

```


- Fairly complicated compiler transformation
  - Need to manage registers properly
  - How many iterations to overlap depends on instruction latency (for long latency combination of loop unrolling and software pipelining may be useful)
  - Advantage over loop unrolling: avoids code explosion
  - Disadvantage: does not reduce loop overhead
  - Combination of loop unrolling and software pipelining usually yields the best result, but complex transformations may be needed to be able to do that



MAINAK CS422 22

## Trace scheduling


- Need for global scheduling techniques
  - Loop unrolling and software pipelining work well if loop body is a single basic block: these are local scheduling techniques
  - Branches in loop body require moving instructions across basic blocks: need to schedule a trace, not just a basic block
  - Effective global scheduling must estimate relative frequency of executing target and fall-through
  - Hoisting a code segment from either target or fall-through to above the branch may be helpful if there are empty issue slots before the branch *and* if it does not slow down the entire code due to new dependences in the other path
- Deciding what portion of code to move requires complex trade-off analysis



MAINAK CS422 23

## Trace scheduling

- Trace selection
  - Gather branch statistics (either via static prediction or profiling)
  - Trace scheduling is effective only if there is a large difference of execution frequency between the two paths
  - Pick the frequent path and form a long trace by concatenating basic blocks from this path
- Trace compaction
  - Schedule this trace at appropriate place where issue slots are available
  - Since this trace is now a straight line code it is possible to re-order instructions maintaining dataflow
  - However, traces with multiple entry points make life complicated: a lot of compensation code may be needed
- Also add compensation code on the alternate path and exit points (hopefully alternate path will be taken rarely)



MAINAK CS422 24

## Trace scheduling

- Superblocks
  - Every trace must have a single entry point: called a superblock
  - Since a superblock has a single entry point, multiple exit points may be needed depending on entry state
  - At each exit point fix-up code will be needed; simple example: unrolling a loop with a conditional branch
  - For loops this process of generating multiple exit points is known as tail duplication
  - Tail duplication creates a block of code that executes residual iterations



MAINAK CS422

25

## Hardware support

- Compiler alone cannot do much
  - Need some hardware support to help the compiler go aggressive
  - We have already seen one such support: nullifying branches
  - Look at three such supports
    - Conditional instructions and predication
    - Preserving exception behavior
    - Memory reference speculation



MAINAK CS422

26

## Conditional instructions

- Want to get rid of branches completely

```
if (!A) { a = b; } cmovz r2, r3, r1
```

    - The move executes only if r1 is zero
    - Converts control dependences to data dependences
    - Essentially we have more time to get r1 ready: instead of in the fetcher we now need it in EX stage
    - Also known as if-conversion
    - Useful in compiling  $y = \text{abs}(x)$ : `if (x < 0) {y = -x;} else {y = x;}`
    - Very useful in getting rid of hard-to-predict branches
    - However, eliminating branches that guard a large piece of code may require too many conditional moves`cmovz` is supported by all processors today
- How does it interact with register renaming?



27

## Predication

- Problematic to offer conditional versions of all instructions (why? conditional add?)
    - Predication is a more generalized technique
    - Provide a set of dedicated predicate registers (1 bit each)
    - Let the predicate register become an operand in instruction (need three sources)
    - Every instruction executes depending on a predicate
    - Possible to boost instructions even before the branch
- ```
lw r1, 40(r2)          lw r1, 40(r2)
stall                  cmp.eq p3, p4, r10, r0 // p4 = !p3
add r3, r1, r4         lw r8, 0(r10), p4 // r10 == 0 ?
beqz r10, label       add r3, r1, r4
lw r8, 0(r10)         lw r9, 0(r8), p4
stall
lw r9, 0(r8)
```



MAINAK CS422

28

Predication

- Schedule target and fall-through in parallel
 - Can fetch and execute both paths, but commit only the true predicate instructions
 - Shortcomings
 - Nullified predicated instructions still consume precious processor resources: predicate only those instructions that are guarded by hard-to-predict branches (statically)
 - New data hazard stalls may get introduced if predicate generation and use are not sufficiently apart
 - Predication helps code motion, but moving instructions across multiple branches may need extra instructions to generate correct predicates (essentially combines conditions of multiple branches)
- Itanium supports full predication with 64 predicate registers (p0 is hardwired to true)



MAINAK CS422

29

Exception behavior

- Compiler wants to speculate past branches
 - Predication allows boosting of instructions above the branch, but not above the condition evaluation
 - For long-range control speculation additional support is needed
 - This kind of speculation is especially important for loads to hide latency: therefore it is important to preserve exception behavior
 - Fundamental requirement: ISA should be able to specify if an instruction is speculatively scheduled by the compiler (this is software speculation)
 - Itanium offers speculative load instructions: `load.s`



MAINAK CS422

30

Exception behavior

- Four ways to preserve exceptions
 - Hardware tells OS to ignore exceptions generated by speculated instructions
 - Incorrect programs will miss legitimate exceptions
 - Speculated instructions do not raise exceptions and checks are introduced at original places of the speculated instructions (this is where exceptions are raised)
 - Itanium offers `chk.s` instruction for this purpose
 - Status bits are attached to registers; this bit gets written by a speculated exception instruction and gets propagated through the consumer chain; exception is raised when a non-speculative instruction reads a register with this bit set
 - Itanium offers NaT (Not a Thing) bit; `chk.s` inspects the NaT bit and jumps to recovery code if set



31

Exception behavior

- Four ways to preserve exception behavior
 - Buffer all speculated instructions in processor until they become non-speculative
 - Loses almost all benefits of software speculation (why?)
- Itanium uses combination of second and third methods
 - A non-speculative instruction (e.g. a store) reading a register with NaT set, raises an exception when it commits
 - A `chk.s` instruction is also placed at the original position of the speculated load; this instruction checks the destination register's NaT bit and if set branches to a recovery code which retries the load



MAINAK CS422

32

Data speculation

- Deals with the problem of inexact alias analysis
 - Want to boost loads above stores even if not sure if they access the same address
 - ISA must offer instructions to distinguish speculated loads from normal loads
 - Itanium offers `load.a` instruction (advanced load)
 - Advanced load records its destination register and address in ALAT (advanced load alias table) CAM
 - A store instruction writing to the same address clears the valid bit of the matching entries in ALAT
 - A check instruction is placed in the original location of the load instruction which inspects the ALAT
 - Two types of check instructions: `chk.a` and `load.c`



MAINAK CS422

33

Data speculation

- Data speculation in Itanium
 - `load.c` is used when only the load is boosted; it associatively searches ALAT with the destination register number; if valid bit is set in the matched entry the load performed correctly, otherwise the load is retried
 - `chk.a` is used if along with the load some dependents were also boosted; in this case it branches to a recovery code if the valid bit in ALAT entry is reset



MAINAK CS422

34

Hardware vs. compiler

- Trade-offs
 - Alias analysis is a big problem in compiler-based optimizations; data speculation supports help a lot; dynamic hardware memory disambiguation is much better but requires complex circuitry
 - Control speculation in compiler relies heavily on accuracy of static branch prediction which is worse than dynamic prediction for most applications
 - Maintaining precise exception in hardware falls out automatically due to in-order retirement
 - Compiler can see the entire code and is not limited to just the issue queue width
 - Compiler scheduling may depend on latency and functional unit organization
- At the end of the day, designer must weigh hardware complexity, compiler complexity, relative speedup, and binary compatibility before taking a decision

