

CSE 533

Advanced Computer Architectures

Static Scheduling

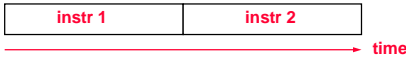
1

Instruction Scheduling

- Until today, we assumed that instructions execute sequentially one after another in a von Neumann style of execution
- However, today's processors do not execute instructions in this model; architectural "enhancements" allow processors to execute code faster:
 - pipelining
 - multiple function units
- Instruction scheduling refers to re-ordering instructions in a program to exploit Instruction Level Parallelism (ILP) and improve performance
- Instruction scheduling is still an active area of research because of the difficulty of the problem (NP-complete) and the changing natures of processors


2

Instruction Scheduling

- In the von Neumann model of execution an instruction starts only after its predecessor completes.
 
- This is not a very efficient model of execution.
 - von Neumann bottleneck or the memory wall.

3

Instruction Pipelines

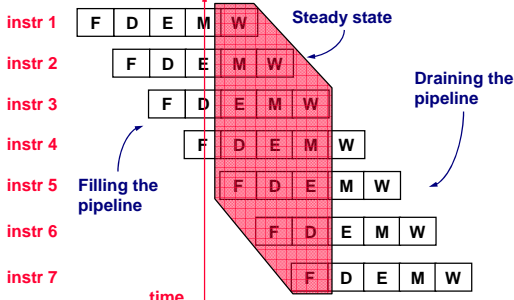
- Almost all processors today use instructions pipelines to allow overlap of instructions (Pentium 4 has a 20 stage pipeline!!!).
- The execution of an instruction is divided into stages; each stage is performed by a separate part of the processor.
 

F: Fetch instruction from cache or memory.
D: Decode instruction.
E: Execute. ALU operation or address calculation.
M: Memory access.
W: Write back result into register.

- Each of these stages completes its operation in one cycle (shorter than the cycle in the von Neumann model).
- An instruction still takes the same time (maybe a little more due to pipelining) to execute.

4

Instruction Pipelines

- However, we overlap these stages in time to complete an instruction every cycle.
 

5

Pipeline Hazards

- Structural Hazards**
 - two instructions need the same resource at the same time
 - memory or functional units in a superscalar.
- Data Hazards**
 - an instructions needs the results of a previous instruction

$$r1 = r2 + r3$$

$$r4 = r1 + r1$$

$$r1 = [r2]$$

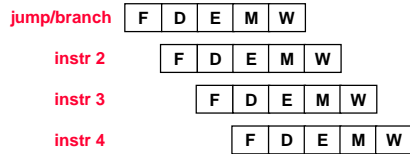
$$r4 = r1 + r1$$
 - solved by forwarding and/or stalling
 - cache miss?
- Control Hazards**
 - jump & branch address not known until later in pipeline
 - solved by delay slot and/or prediction

6

Jump/Branch Delay Slot(s)

- Control hazards, i.e. `jump`/branch instructions.

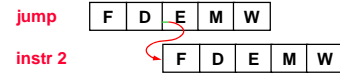
unconditional `jump` address available only after Decode.
conditional branch address available only after Execute.



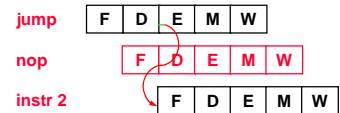
7

Jump/Branch Delay Slot(s)

- One option is to stall the pipeline (hardware solution).



- Another option is to insert a no-op instructions (software).

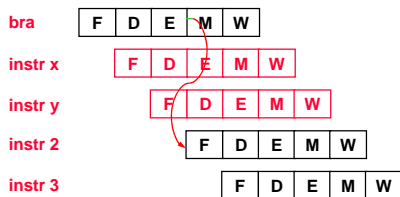


- Both degrade performance!

8

Jump/Branch Delay Slot(s)

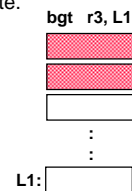
- A better solution is to make the branch take effect only after the delay slots.
- That is, one or two instructions always get executed after the branch but before the branching takes effect.



9

Jump/Branch Delay Slots

- In other words, the instruction(s) in the delay slots of the `jump`/branch instruction always get(s) executed when the branch is executed (regardless of the branch result).
- Fetching from the branch target begins only after these instructions complete.



- What instruction(s) to use?

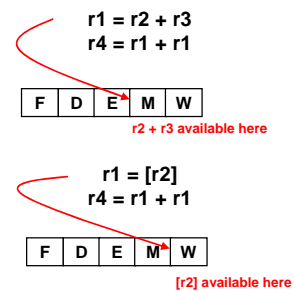
10

Branch Prediction

- Current processors will speculatively execute at conditional branches
 - if a branch direction is correctly guessed, great!
 - if not, the pipeline is flushed before instructions commit (WB).
- Why not just let compiler schedule?
 - The average number of instructions per basic block in typical C code is about 5 instructions.
 - branches are not statically predictable
 - What happens if you have a 20 stage pipeline?

11

Data Hazards



12

Dependence Graph (DAG)

- To schedule a basic block, you need to determine scheduling constraints and express these using a *dependence graph*
- For a basic block, this graph is a DAG
- Each node is a machine instruction and the edges are the dependencies between instructions

13

Flow (True) Dependencies

- A flow dependence exists if an instruction I_1 writes to a register or location that I_2 uses.
- This is written $I_1 \delta^f I_2$

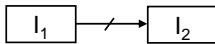
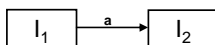


- Flow dependencies are *true* dependencies, that is these dependencies are necessary to transmit information between statements.

14

Anti Dependencies

- An anti dependence exists if an instruction I_1 uses a register that I_2 changes.
- This is written $I_1 \delta^a I_2$

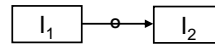
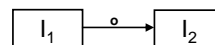


- Anti dependencies are *false* dependencies, that is they arise due to the reuse of memory locations.

15

Output Dependencies

- An output dependence exists if an instruction I_1 writes to register that I_2 also writes to.
- This is written $I_1 \delta^o I_2$



- Output dependencies are also *false* dependencies, that is they arise due to the reuse of memory locations.

16

List Scheduling Algorithm - Example

$a = b + c$
 $d = e - f$

1. load R1, b
2. load R2, c
3. add R2, R1
4. store a, R2
5. load R3, e
6. load R4, f
7. sub R3, R4
8. store d, R3

Assume, that loaded values are available after 2 cycles (from beginning of load instruction). So, there is need for an extra cycle after 2 and after 6. In the absence of instruction scheduling, NOPs must be inserted.

17

List Scheduling Algorithm - Example

$a = b + c$
 $d = e - f$

1. load R1, b
2. load R2, c
3. add R2, R1
4. store a, R2
5. load R3, e
6. load R4, f
7. sub R3, R4
8. store d, R3

Step 1: construct a dependence graph of the basic block. (The edges are weighted with the latency of the instruction).

Step 2: use the dependence graph to determine instructions that can execute; insert on a list, called the **Ready** list.

Step 3: use the dependence graph and the Ready list to schedule an instruction that causes the smallest possible stall; update the Ready list. **Repeat until Ready list is empty!**

18

List Scheduling Algorithm - Example

$$a = b + c$$

$$d = e - f$$

1. load R1, b
2. load R2, c
3. add R2,R1
4. store a, R2
5. load R3, e
6. load R4, f
7. sub R3,R4
8. store d, R3

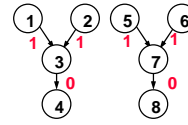
19

List Scheduling Algorithm - Example

$$a = b + c$$

$$d = e - f$$

1. load R1, b
2. load R2, c
3. add R2,R1
4. store a, R2
5. load R3, e
6. load R4, f
7. sub R3,R4
8. store d, R3



1. load R1, b
5. load R3, e
2. load R2, c
6. load R4, f
3. add R2,R1
7. sub R3,R4
4. store a, R2
8. store d, R3

We're done. Now have a schedule that requires no stalls and no NOPs.

20

Superscalars, i.e. multiple functional units

- Almost all modern processors are superscalars
 - have multiple functional units
 - Intel 486: 1 pipeline; Pentium: 2 pipelines; Pentium 4: up to 6 instructions per clock cycle.
- Need to model the CPU as accurately as possible
 - which instructions can execute simultaneously
 - relative delay of different types of instructions
- Can use a Greedy / Ready list method
 - not always optimal, nontrivial scheduling is NP

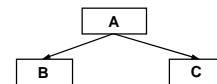
<u>IntFlt</u>	<u>IntMem</u>	<u>IntFlt</u>	<u>IntMem</u>
FltOp	FltLd	FltOp	IntOp
IntOp	IntLd		IntLd
			FltLd

(with FltOp → IntLd)

21

Trace Scheduling

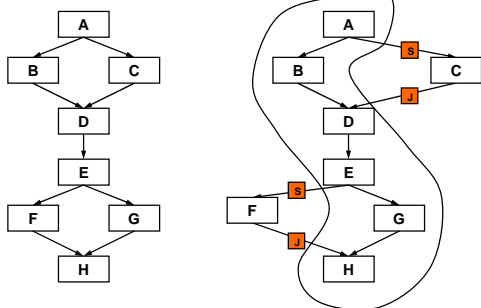
- Basic blocks typically contain a small number of instructions.
- With many function units, we may not be able to keep all the units busy with just the instructions of a basic block.
- Trace scheduling allows block scheduling across basic blocks.
- The basic idea is to dynamically determine which blocks are executed more frequently. The set of such basic blocks is called a trace.



- The trace is then scheduled as a single basic block.
- Blocks that are not part of the trace must be modified to restore program semantics if/when execution goes off-trace.

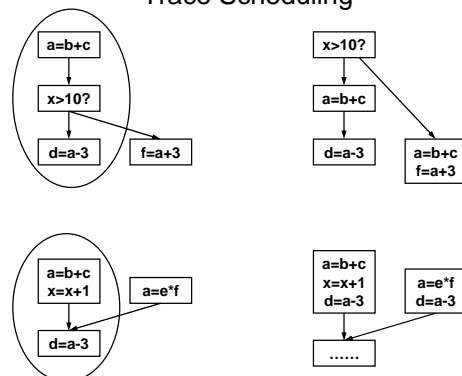
22

Trace Scheduling



23

Trace Scheduling



24

Loop Unrolling

```

L1:
r6 = *(r2)      (ld)
r6 = r6*r3      (mul)
*(r2) = r6      (st)
r2 = r2 + 4     (add)
r6 = *(r2)      (ld)
r6 = r6*r3      (mul)
*(r2) = r6      (st)
r2 = r2 + 4     (add)
if (r2 <= r5) go to L1 (ble)
  
```

31

Register Re-naming

```

L1:
r6 = *(r2)      (ld)
r6 = r6*r3      (mul)
*(r2) = r6      (st)
r2 = r2 + 4     (add)
r6 = *(r2)      (ld)
r6 = r6*r3      (mul)
*(r2) = r6      (st)
r2 = r2 + 4     (add)
if (r2 <= r5) go to L1 (ble)

r6 = *(r1)      (ld)
r6 = r6*r3      (mul)
*(r1) = r6      (st)
r2 = r1 + 4     (add)
r7 = *(r2)      (ld)
r7 = r7*r3      (mul)
*(r2) = r7      (st)
r1 = r1 + 8     (add)
if (r1 <= r5) go to L1 (ble)
  
```

32

Software Pipelining

- Software pipelining: overlap multiple iterations of a loop to fully utilize hardware resources.
- Find the steady-state window so that:
 - all the instructions of the loop body is executed
 - but from different iterations

ld		ld		ld	st	ld	st	ld	st	ld	st		
	ld		ld		ld	st	ld	st	ld	st	ld	st	
		mul		mul		mul	ble	mul	ble	mul	ble	mul	
			mul		mul		mul	ble	mul	ble	mul	ble	
				mul		mul		mul		mul		mul	
					add		add		add		add		
						add		add		add		add	

33

Software Pipelining

```

L1:
r6 = 0[r2]      (ld)
r6 = r6*r3      (mul)
0[r2] = r6      (st)
r2 = r2 + 4     (add)

r6 = 0[r2]      (ld)
r6 = r6*r3      (mul)
0[r2] = r6      (st)
r2 = r2 + 4     (add)

r6 = 0[r2]      (ld)
r6 = r6*r3      (mul)
0[r2] = r6      (st)
r2 = r2 + 4     (add)

r6 = 0[r2]      (ld)
r6 = r6*r3      (mul)
0[r2] = r6      (st)
r2 = r2 + 4     (add)
if (r2 <= r5) go to L1 (ble)

r5 = r5 - 12
r6 = 0[r2]      (ld)
r6 = r6*r3      (st)
0[r2] = r6      (st)

r6 = 4[r2]      (ld)
r6 = r6*r3      (mul)

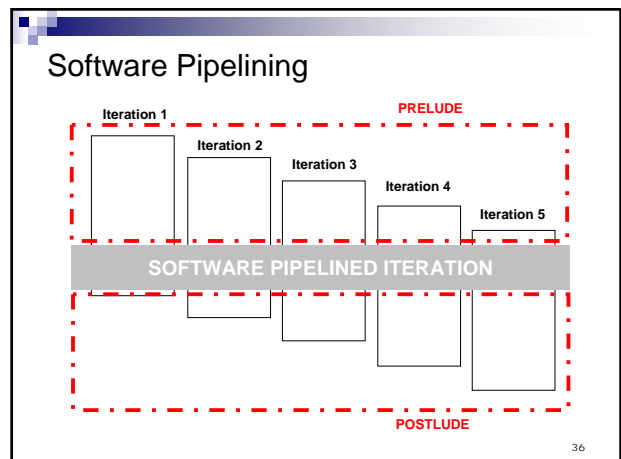
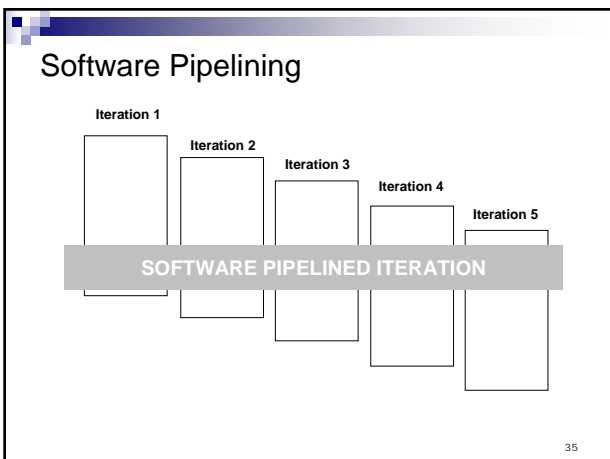
r7 = 8[r2]      (ld)
L1: 4[r2] = r6      (st)
r6 = r7*r3      (mul)
r7 = 12[r2]     (ld)
r2 = r2 + 4     (add)
if (r2 <= r5) go to L1 (ble)

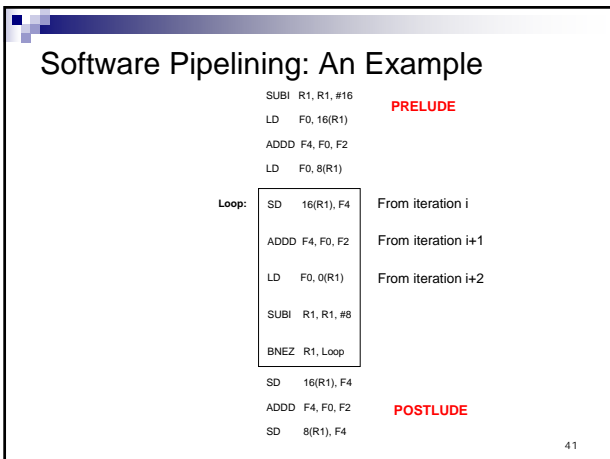
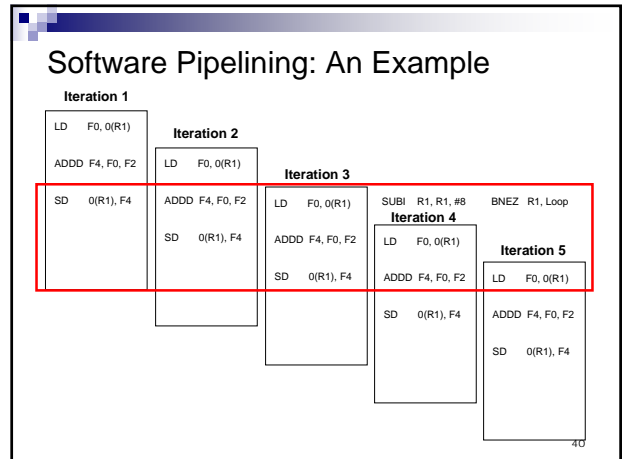
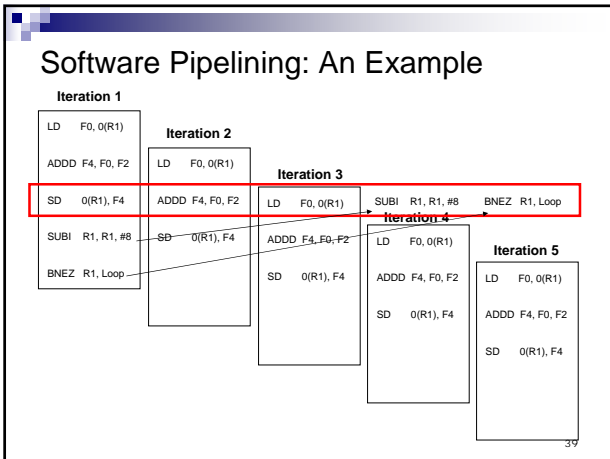
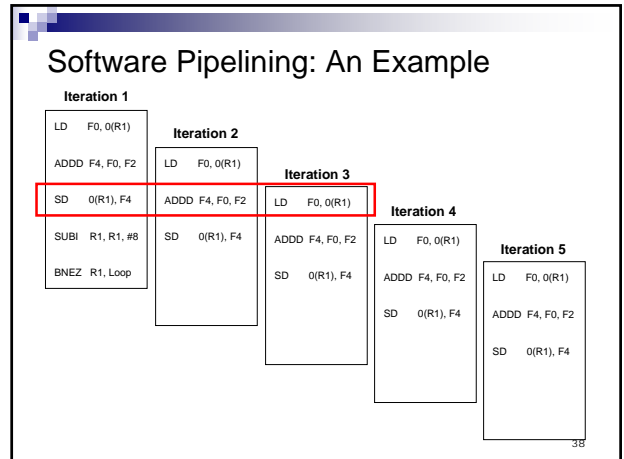
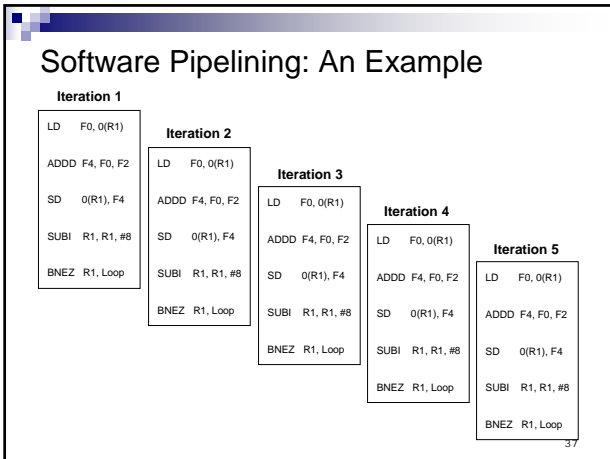
r2 = r2 + 4     (add)

0[r2] = r6      (st)
r2 = r2 + 4     (add)

r6 = r7*r3      (mul)
0[r2] = r6      (st)
r2 = r2 + 4     (add)
  
```

34





- ### Loop Unrolling and Software Pipelining
- Loop Unrolling
 - helps uncover Instruction Level Parallelism (ILP)
 - reduces looping overhead (increment and branch)
 - generates a lot of code, copies of the loop body
 - Software Pipelining
 - also helps uncover ILP
 - does not reduce looping overhead
 - loop body is always executing at top speed
 - usually uses less code space
 - Both require that number of iterations is known
 - If unroll factor does not evenly divide iterations, the extra iterations must be caught by a pre- or post-amble
- 42

Emerging Architectures

- Simultaneous Multithreading (SMT)
 - Execute multiple threads simultaneously
 - Keeps functional units busy
 - Example: Intel Pentium IV with HyperThreading (HT)
 - Sun: 2 cores both with SMT (an SMT CMP)
- EPIC / VLIW Architectures
 - EPIC – Explicitly Parallel Instruction Computing (Itanium / IA64)
 - VLIW – Very Long Instruction Word (Transmeta)
 - Compiler explicitly packages independent instructions
 - Hardware does not due reordering
 - Chip can run at higher clock rates
- Some may live, while some may die
 - Which one will compilers work best with?

43

EPIC / VLIW

- Finding independent instructions at runtime takes time.
- Less logic means faster clock cycle?!?!
- Can the compiler explicitly group independent instructions?
- VLIW: Very Long Instruction Word
 - schedule for a particular microarchitecture
 - timing may be important, number of functional units is fixed
- EPIC: Explicitly Parallel Instruction Computing
 - number of functional units is not fixed, hardware interlocks
 - speculation support
 - predication support

44

VLIW -vs- Superscalar

- VLIW
 - use very long multi-operation instructions
 - the instruction specifies what each functional unit is to do
 - expects dependence free instructions
 - compiler must explicitly detect and schedule independent instructions
- Superscalar
 - uses traditional sequential operations
 - processor fetches multiple instructions per cycle
 - detects dependencies and schedules accordingly
 - has dynamic information available
 - compiler can help by placing independent operations close to each other

45

Problems with VLIW

- Compiler must statically determine dependencies
- Compiler must have very detailed model of architecture
 - number and type of functional units
 - delays for each operation
 - memory delays
 - latencies are very important
- A new generation with more units or different latencies means **recompile**
- EPIC (Itanium / IA64) *tries* to address some of these
 - compiler expresses parallelism; hardware schedules ops
 - no fixed length to instructions, just a number of bundles
 - relationship of one bundle to another is expressed

46

Predication: Branches are Bad

- Provide predicate registers and predicated instructions
- Can set a predicate register to true or false using a comparison instruction
- Most instructions can be *predicated* so that they only commit if their predicate is true
- Can schedule and execute across multiple directions of a branch and only valid instructions will commit

```

if ( m == n ) {
    a = a + b;
} else {
    b = b + 1;
}
    
```

```

cmp.eq p1,p2 = r1, r2
(p1) add r1 = r1, r2
(p2) add r2 = r2, 1
    
```

No branches!!!

47

Speculation

- Control Speculation
 - move an instruction above a branch instruction
 - may be done if it is always safe
 - or can be done speculatively

```

(p1) br.cond label          ld8.s r1 = [r4]
ld8 r1 = [r4]              ld8.s r2 = [r5]
ld8 r2 = [r5]              add r3 = r1, r2
add r3 = r1, r2            (p1) br.cond label
                           chk.s r3, fixupcode
    
```

- ld8.s speculatively loads a value, if the load fails a NAT bit is set
- NAT bits propagate through all other uses
- the chk.s instructions checks the NAT bit, if set it calls the fixupcode

48

Speculation

■ Data Speculation

- move a load before a store that may be aliased
- may be done if it is always safe
- or can be done speculatively

```
ld8 r4 = [r1]          ld8 r4 = [r1]
ld8 r5 = [r2]          ld8 r5 = [r2]
cmp.eq p1 = r4, 0      ld8.a r6 = [r3]
(p1) add r5 = r5, 1    cmp.eq p1 = r4, 0
(p1) stl [r2] = r5     (p1) add r5 = r5, 1
ld8 r6 = [r3]          (p1) stl [r2] = r5
add ret0 = r6, -1     ld8.c r6 = [r3]
br.ret                add ret0 = r6, -1
                     br.ret
```

- the Advanced Load Address Table (ALAT) checks for collisions

49

EPIC / Itanium Processor Family

- Prediction and Speculation
- Special support for software pipelining
 - rotating registers
 - special epilogue counters
- Performance????
 - need more compiler work
 - a big change and it's still early
 - does ok on floating-point
 - profiling may help
 - runtime techniques may help
- What are the issues?

50

We've now covered optimizations found in most commercial compilers

- Compilers improve performance dramatically
- The optimizations in this class improve code
- A little test: compile gcc using cc

51

CC Optimization Levels

- xO1 Does **basic local optimization** (peephole).
- xO2 Does **basic local and global optimization**. This is **induction variable elimination, local and global common subexpression elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, register allocation**, basic block merging, tail recursion elimination, **dead code elimination**, tail call elimination and complex expression expansion.

52

CC Optimization Levels

- xO3 Performs like -xO2 but, also optimizes references or definitions for external variables. **Loop unrolling and software pipelining** are also performed. In general, the -xO3 level results in increased code size. Does not deal with pointer disambiguation.
- xO4 Performs like -xO3 but, also does automatic inlining of functions contained in the same file; this usually improves execution speed. The -xO4 level does trace the effects of pointer assignments. In general, the -xO4 level results in increased code size.

53

CC Optimization Levels

- xO5 Generates the highest level of optimization. Uses optimization algorithms that take more compilation time or that do not have as high a certainty of improving execution time.
- fast: -O4 plus some other specific flags...

54

