More on Machine-Code Generation

- ***Problem****: given a target machine specification, how to translate the intermediate representations into* ***efficient*** *machine code ?*
- ***Solution ---*** *must take consideration of the machine architecture*

  1. *Code Selection*
     (emitting the machine code via **maximal-munch** or **dynamic programming**)
  2. ***Register Allocation***
     (global register allocation, spilling)
  3. ***Instruction Scheduling***
     (instruction scheduling, branch prediction, memory hierarchy optimizations)

- ***Language Trends*** *: assembly -> C -> ... -> higher-level languages ?*
- ***Architecture Trends*** *: CISC -> RISC -> ... -> superscalar -> ?*
- ***Trends****: the bridging gap is the main* ***challenge*** *to compiler writers*

Register Allocation

- ***Register allocation*** *often works on the intermediate representations that are very much like the machine code.*
- ***Input****: intermediate code that references unlimited number of registers;* ***output****: rewrite the intermediate code so that it uses the limited registers available on the target machine --- the machine registers.*
- ***Standard Algorithm:*** ***Graph Coloring Register Allocation***

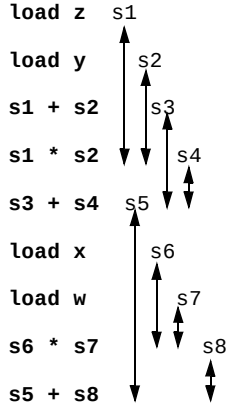
  Main idea: build a interference graph based on the live ranges of each identifiers; then color the interference graph.

  Example: ***Yorktown Allocator*** *(by Chaitin et al. at IBM T.J.Watson)'*

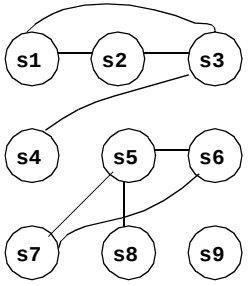
  Briggs's Extension *(by Briggs et al. at Rice Univ.)*

Example: Register Allocation

```
s1 := load z
s2 := load y
s3 := s1 + s2
s4 := s1 * s2
s5 := s3 + s4
s6 := load x
s7 := load w
s8 := s6 * s7
s9 := s5 + s8
```

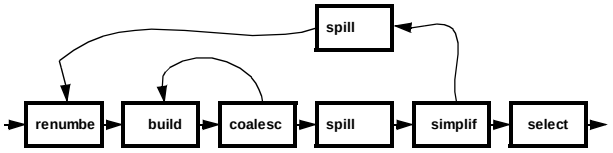


how to color the interference graph ?



Yorktown Allocator

- ***Renumber****: name all identifiers uniquely, find out their live ranges.*
- ***Build****: construct the interference graph G.*
- ***Coalesce****: eliminating copying instructions, e.g.,* ***x = y***
- ***Spill Costs****: calculate the spill costs*
- ***Simplify****: (together with* ***Select****) color the graph (it is NP-complete!).*
- ***Select****: choose the actual colors (i.e., registers)*
- ***Spill Code****: insert the spill code*



Yorktown Allocator (cont'd)

- ***Build****: the interference graph characterizes the* ***interference*** *relation of live ranges: two live ranges* ***interfere*** *if there exists some point in the procedure and a possible execution of the procedure such that*

  1. both live ranges have been defined
  2. both live ranges will be used, and
  3. the live ranges have different values

- ***Simplify and Select****: assuming there are* ***k*** *physical registers*

  In ***Simplify***, the allocator repeatedly removes nodes with outer degree ***< k*** from the graph and pushes them onto a stack.

  In ***Select***, the nodes are popped from the stack and added back to the graph --- a color is chosen for each node.

  If ***Simplify*** encounters a graph containing only nodes of degree ***>= k***, then a node is chosen for spilling.

Yorktown Allocator (cont'd)

- ***Choosing Spill Nodes****: based on the weight* m_n *for each node* ***n***

  Chaitin's heuristics: $m_n = cost_n / degree_n$

  Alternatives: $m_n = cost_n / (degree_n * area_n)$

  where $area_n$ is a function that quantifies the impact ***n*** has on other live ranges in the program, e.g., if it is used in a loop often, $area_n$ is larger.

- ***Spilling****: if* ***v*** *is spilled, a* ***store*** *is inserted after every definition of* ***v****, and a* ***load*** *is inserted before every use of* ***v****.*

- ***Bernstein*** *et al. later found no single* ***spilling-cost heauristics*** *completely dominates the other. They propose "****best of 3****" technique:*

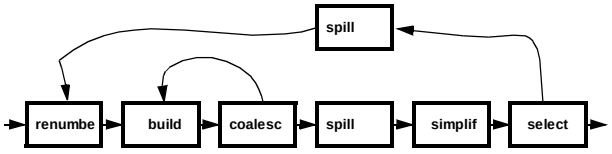
  Just run the algorithm using three heauristics, then choose one with the best outcome.

Briggs's Extension

- ***Simplify*** *removes nodes with degree* ***< k*** *in an arbitrary order. If all remaining nodes have degree* ***>= k****, a spill candidate is choosed and optimistically pushed on the stack also, hoping a color will be found later.*

- ***Select*** *may discover that it has no color for some node. In that case, it leaves the node uncolored and continues with the next node.*

- *If any nodes are uncolored, the allocator inserts spill code accordingly and rebuild the interference graph, and tries again.*

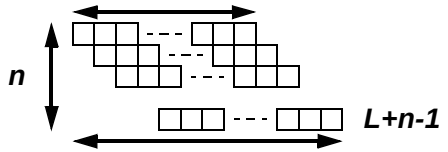


Instruction Scheduling

- *Why* ***Instruction Scheduling*** *? to exploit* ***instruction-level parallelism:***

  1. pipelining 2. superscalar 3. VLIW

  - *What is* ***Pipelining*** *?*



  - *Pipelining Performance: assuming we are executing* ***n*** *operations; each operation takes* ***L*** *stages; each stage takes one cycle;*

    Non-pipelined: L * n
    Pipelined: L + n - 1
    Speedup: close to **L** (when n is large) ---- # of pipeline stages

Typical Pipelining Stages

- *The standard 5 pipeline stages (see DLX in Hennessy&Patterson):*

Instruction fetch step (**IF**)

```
IR <- Mem[PC]; NPC <- PC + 4;
```

Instruction decode/register fetch step (**ID**)

```
A <- Regs[IRa]; B <- Regs[IRb];
```

Execution/effective address step (**EX**)

```
Mem-Ref:          ALUoutput <- A + IRimmed;
  (for store)     SMD <- B;
ALU Instr:        ALUoutput <- A op B;
Branch/Jump:      ALUoutput <- NPC + IRimmed; cond <- (A op 0)
```

Memory access/branch completion step (**MEM**)

```
Mem-Ref:    LMD <- Mem[ALUoutput] or Mem[ALUoutput] <- SMD
Branch/Jump:  if (cond) PC <- ALUoutput else PC <- NPC
```

Write-back step (**WB**)

```
Regs[IRtarget] <- ALUoutput; or Regs[IRtarget] <- MD;
```


More on Pipelining

- *Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload.*
- ***Pipeline rate** is limited by the **slowest** pipeline stage; unbalanced lengths of pipe stages reduces speedup*
- ***Limits to pipelining** in a machine: **Hazards** prevent next instruction from executing during its designated clock cycle.*

structure hazards : hardware can not support this combination of instructions due to resource conflicts

data hazards : instruction depends on result of prior instruction still in the pipeline

control hazards : pipelining of branches and other instructions that change the PC

Instruction Scheduling

- ***Instruction scheduling** is to avoid data and control hazards*
- *The **Algorithm** :*

draw the **schedule (or dependency) graph**, then use the following greedy scheduling algorithm:

1. use heuristics to pick one of the unscheduled operations whose predecessors have already been scheduled.

2. calculate the completion time for each of its predecessor operations by adding their execution latency to the time at which they are scheduled to begin execution. Find the earliest time that this operation can be scheduled.

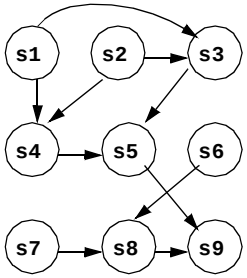
3. (*optional*) determine the earliest time an appropriate functional unit is available to perform this operation.

4. repeat the procedure from step 1.

Example: Instruction Scheduling

```
s1 := load z
s2 := load y
s3 := s1 + s2
s4 := s1 * s2
s5 := s3 + s4
s6 := load x
s7 := load w
s8 := s6 * s7
s9 := s5 + s8
```

topologically-sort the schedule graph



after scheduling:

```
s1 := load z
s2 := load y
s6 := load x
s7 := load w
s3 := s1 + s2
s4 := s1 * s2
s5 := s3 + s4
s8 := s6 * s7
s9 := s5 + s8
```

open problem: *interaction with reg. alloc. ?*

Main Challenges

- ***Basic instruction scheduling*** *may not work very effectively; Often* ***cannot*** *find enought instructions to fill the* ***load*** *and* ***branch delay*** *slot.*
- ***Instruction scheduling*** *does not interact with* ***register allocation*** *well*

  a. doing scheduling first may increase register pressures
  b. doing reg. allocation first may add more schedule dependencies

- *To get the* ***best*** *performance, the key is to* ***avoid hazards*** *!*

  structure hazards --- add multiple set of hardware

  control hazards --- use *branch prediction* and *trace scheduling*
  (static & dynamic branch prediction, trace scheduling)

  data hazards --- design new *memory hierarchy optimization*
  (forwarding, bigger caches, write buffers, prefetching, speculative loads)

Control Harzard & Branch Stall

- ***Control Hazard****: assuming branch stall three cycles*

(with delay slots filled)

```
40:   BEQ  R1, R3, 36
44:   AND R12, R2, R5
48:    OR R13, R6, R2
52:   ADD R14, R2, R2
      ..........
80:    LD  R4, R7, 36
```

(delay slots NOT filled)

```
40:   BEQ  R1, R3, 36
44:   NOP
48:   NOP
52:   NOP
      ..........
80:    LD  R4, R7, 36
```

branch penalty: 30% branch instructions, stall 3 cycles; means 90% slowdown in performance

- *Dealing with* ***Control Hazard****:*
  1. **stall** until branch direction is clear
  2. **predict not taken** (PC + 4 already calculated)
  3. **predict taken** (more likely but target address is not calculated yet)
  4. **delayed branch** --- define *delay slots* --- let the compiler fill it

Branch Stall (cont'd)

- ***Delayed Branch*** *--- where to get instructions to fill branch delay slots ?*
  1. before branch instruction
  2. from the *target* address: only valuable when branch
  3. from *fall through*: only valuable when don't branch

  Effectiveness: one-delay-slot --- 50% filled (Hennessy&Patterson)

  More problems for super-pipelined and superscalar machines !

- ***Branch Prediction*** *: improves strategy on placing instructions in delay slot.*
- *Other reasons to do* ***Branch Prediction*** *: (1) other compiler optimizations (2) support trace scheduling and speculative execution.*
- ***Branch Prediction*** *methods:* ***static*** *or* ***dynamic***

Static Branch Prediction

- ***Static Branch Prediction*** *is done before the program runs, i.e., mostly at compile time.* ***Advantage****: prediction information is available at compile time, so can be used in other optimizations; no hardware support necessary.*
- *Three ways to do* ***Static Branch Prediction****:*

  1. programmer-inserted directives --- you say it in your programs

  2. the compiler examines the source and uses ***heuristics****, e.g.,*

  a. backward branch predict taken, forward branch untaken ...
  b. always favor loop branches ; if nested-loop, favour internal loops;
  c. opcode heuristic (favor >=0, >0, <>0 operators, etc.); pointer null comparison;
  d. favor the branch with loop; NOT favor branches with call and return ...

  3. profiling-based --- the program is run, statistics are gathered and fed back into the source code; then the program is recompiled using these statistics.

Dynamic Branch Prediction

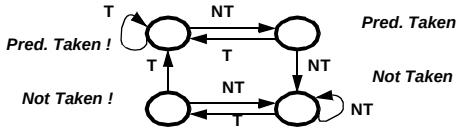
- ***Dynamic Branch Prediction*** *is done while a program is running, mostly done by branch prediction* <u>hardware</u>. ***Advantage****: the same branch can be predicted taken at some time, and untaken at another time.*
- ***Branch History Table*** *(BHT) ---* <u>*1-Bit scheme*</u>

  lower bits of PC address index table of 1-bit values;
  the 1-bit says whether or not branch taken last time;

  <u>**Problem:**</u> **in a loop, 1-bit BHT will cause 2-mispredictions**

- ***Branch History Table*** *(BHT) ---* <u>*2-Bit Scheme*</u> *---*

  change prediction only if get misprediction ***twice*** *!*



Dynamic Branch Pred. (cont'd)

- ***BHT Accuracy****: mispredict because either wrong guess for that branch or got branch history of wrong branch when index the table.* ***In practice****: 4096 entry table get 1%-18% misprediction*
- ***Technique #2 --- Correlating Branches*** *--- main idea: taken/not taken of a recently executed branches is related to behavior of next branch.*

  Source code: **B1**: `if (d == 0) d = 1;`
  B2: `if (d == 1) ......`

  Assembly:

```
    BNEZ R1, L1       ; branch B1 (d != 0)
    ADDI R1, R0, #1   ; d == 0 so d = 1
L1: SUBI R3, R1, #1
    BNEZ R3, L2       ; branch B2 (d != 1)
```

  Clearly, branch B2 is correlated to B1; if B1 is not taken then B2 won't be taken either. Simple 1-bit BHT can behave very bad ...

Dynamic Branch Pred. (cont'd)

- ***A (1,1) Correlating Predictor****: every branch has two spearate 1-bit predictor: one is used when the last branch executed is not taken; another is used when the last branch executed is taken.*
- *A* ***(m,n) correlating predictor*** *uses the behavior of the last m branches to choose from* 2^m *branch predictors, each of which is a n-bit predictor for a single branch.*

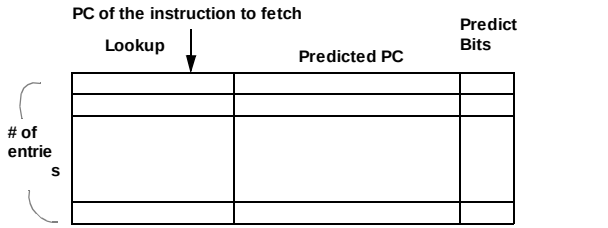
  need to maintain **m-bit** global branch history in a special register during the execution.

- ***Performance****: the 1024-entry* ***(2,2)-correlating-predictor*** *normally outperforms the 4096-entry simple 2-bit predictor.*

Branch Target Buffer

- ***Branch Target Buffer*** *(BTB) --- address of branch index to get* <u>*prediction*</u> *and* <u>*branch address*</u> *(if taken)*

  must check for branch match now, since can't use wrong branch address.



  if PC is in BTB, then instruction is predicted, and the predicted PC is used;

Memory Hierarchy

- *Main **challenges**: widening performance gaps in memory hierarchy*

  *the performance of memory (i.e., **DRAM**) is improving 7% per year;*
  the performance of CPU is improving 25%-50% per year ;

  1995 Memory Hierarchy:
  CPU --> 1st-Lev-Cache --> 2nd-Lev-Cache --> Main Mem --> Disk

- ***Solutions**: **cache** --- often implemented as smaller but faster **SRAM**.*

  *because memory reference exhibits nice **locality** pattern:*

  ***temporal locality** : referenced again soon*

  ***spatial locality** : nearby items referenced soon*

- ***Cache Basics** (see Hennessy&Patterson90 for more details)*

  hit rate: fraction of mem access that found in the cache (often >90%)

  average memory access time = Hit time + Miss rate * Miss penalty

  miss penalty --- time to replace a block in the cache

Improving Cache Performance

- *Reduce the **read miss rate** and **write miss rate***
- *Reduce the **read miss penalty** and **write miss penalty** ?*
- *Reduce the **read hit** and **write hit** time --- this can decrease the cycle time, thus increase the machine frequency.*
- ***Main questions** of **memory hierarchy optimizations**:*

  1. How to reduce the read miss rate and write miss rate ?
  hardware method: modify cache organizations .....

  software method: relocate data, rewrite programs, prefetching, scheduling for the load delay slot, ...

  2. How to reduce the read miss penalty and write miss penalty ?
  hardware method: non-blocking cache, better cache miss policy

Cache performance depends on the actual program behavior !!!

Cache Basics [H&P90]

- ***Basic Organization***

  - multi-level cache
  - separeate I-cache and D-cache
  - unified I- and D-cache
  - block size
  - associativity (directly-mapped, n-way associative, fully-associative)
  - cache replacement policy (LRU)
  - virtual- or physical-addr indexed

- ***Read Miss Policy --- blocking or non-blocking***

- ***Write Hit and Write Miss Policy***

  write through --- written to both cache and lower-level memory
  write back --- only written to cache; go to mem when being replaced

  write around --- at miss, only write to memory, do nothing to cache
  write validate --- at miss, write to cache &memory, validate sub-block
  fetch on write --- at miss, fetch first, then write to cache&mem

Reducing Cache Misses

- *Classifying Cache Misses:*

  - compulsory ------ cold start misses or first reference misses
  - capacity ------ cache not big enough
  - conflict ------ two blocks map to same entry, collision misses

- *Reducing Cache Misses:*

  - varying block size
  - higher associativity ----- *warning*: may increase cache hit time !
  - bigger cache ----- chip density is constrained
  - hardware-controlled prefetching -----
  instruction prefetching (hardware instr. stream buffer)
  data prefetching (using data stream buffer)
  requires extra memory brandwith
  - compiler-controlled prefetching -----
  compiler inserts prefetch instructions (might be costly)
  prefetch to registers or cache, must be no-faulting
  - other compiler optimizations to reduce cache misses

Example: Compiler Optimizations

- *Dealing with **Instruction Cache Misses***
  - reorder code segments, alignment, etc.
  - profiling to look at conflicts
- *Dealing with **Data Cache Misses** (too many to list all of them here)*
  - data alignment
  - improving data locality in garbage collection
  - *merging arrays*: improve spatial locality by single array of compound elements vs. 2 arrays
  - *loop interchange*: change nesting of loops to access data in order stored in memory
  - *loop fusion*: combine 2 independant loops that have same looping and some overlap of variable accesses
  - ***blocking***: improve temporal locality by accessing "blocks" of data repeadtedly vs. going down whole columns or rows

.....

Loop Interchange & Loop Fusion

```
/* Before Loop Interchange */
for (k = 0; k < 100; k=k+1)
  for (j = 0; j < 100; j=j+1)
    for (i = 0; i < 5000; i=i+1)
        x[i][j] = 2 * x[i][j];

/* After Loop Interchange --- sequentially access memory */
for (k = 0; k < 100; k=k+1)
  for (i = 0; i < 5000; i=i+1)
    for (j = 0; j < 100; j=j+1)
        x[i][j] = 2 * x[i][j];

/* Before Loop Fusion */
for (i = 0; i < N; i=i+1)
  for (j = 0; j < N; j=j+1)
     a[i][j] = b[i][j] * c[i][j];
for (i = 0; i < N; i=i+1)
  for (j = 0; j < N; j=j+1)
     d[i][j] = a[i][j] + c[i][j];

/* After Loop Fusion --- share accesses */
for (i = 0; i < N; i=i+1)
  for (j = 0; j < N; j=j+1)
    {a[i][j] = b[i][j] * c[i][j];d[i][j] = a[i][j] + c[i][j];};
```


Blocking Example

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    { r = 0;
      for (k = 0; k < N; k = k+1) {
        r = r + y[i][k] * z[k][j];
      };
      x[i][j] = r;
    }
```

- *2 inner loops: read all NxN elements of* `z[]`*, N elements of 1 row of* `y[]` *repeatedly; write N elements of 1 column of* `x[]`
- *# of capacity misses is a function of N & cache size*
- ***Blocking Idea**: compute on BxB submatrix that fits in the cache.*

Blocking Example (cont'd)

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i + 1)
  for (j = jj; j < min(jj+B-1,N); j = j+1)
    { r = 0;
      for (k = kk; k < min(kk+B-1,N); k = k+1) {
        r = r + y[i][k] * z[k][j];
      };
      x[i][j] = x[i][j]+r;
    }
```

- *# of capacity misses decreased significantly*
- ***Blocking Factor*** `B` *can be choosed depending on the cache size*
- *# of conflict misses ?*

Reducing Miss Penalty

- *Use write buffer when writing things into memory --- non-blocking !*
- *When read miss the cache, check the write buffer first*
- *Subblock placement --- or "**write-validate**" write-miss policy*

  don't have to load full block on a write miss
  have bits per subblock to indicate valid

- *Non-blocking Caches*

  - allowing the data cache to continue serving cache hits during a miss

  - hit under miss
  - hit under multiple miss
  - miss under miss

  still need some compiler assistance to use these features

More Instruction-Level Parallelism

*3 variations of greater **Instruction-Level Parallelism** (**ILP**):*

- ***Superpipelined** --- higher clock rates and deeper pipelines by pipelining all function units. Example: MIPS R4000 (used in SGI machines)*
- ***Superscalar** --- issue multiple instructions (ranging from 1 - 16) per cycle; scheduled by hardware; compiler scheduling would still help. Example: PowerPC, DEC Alpha, HP 7100, SuperSparc.*

  the decode-issue hardware that detects dependency is very complicated; this restricts the degree of multiple issues and the clock time.

- ***VLIW** (Very Long Instruction Words) --- fixed number of instructions scheduled by the compiler. Example: Multiflow, Cydra*

  binary compatibility; large code size; bigger challenge to compilers.

How to Get More ILP ?

- *Major obstacles for more ILP*

  - too many branches; each basic block is too small
  - difficulty to analyze programs that heavily use pointers
  - demand of larger memory bandwidth
  - demand of more registers and various Functional Units
  - superscalar decode-hardware can increase the cycle time

- ***Software** methods to get more ILP*

  - all techniques talked before to deal with data and control hazards
  - *software pipelining*
  - *trace scheduling*
  .....

- ***Hardware** methods to get more ILP*

  - all techniques mentioned before
  - conditional instructions
  - support of speculative execution
  .....

Software Pipelining

- ***Problem**: small basic blocks make scheduling less effective;*
- ***Loop Unrolling**: the loop body can be unrolled seveal times to yields bigger basic blocks; **Disadvantages**: danger of code explosion; instruction cache performance.*
- ***Software Pipelining**: reorganizes loops such that each iteration is made from instructions chosen from different iterations of the original loop. **Advantage** over loop unrolling: optimal performance can be achieved with compact object code.*
- *Implementing software pipelining: at compile time, symbolically unroll the loops multiple times; do the scheduling; find out the regularity among different iterations. Restructure the loop into the form:*

  prolog; new loop body; epilog

Software Pipelining (cont'd)

A Superscalar Machine, can issue four instructions (one ALU, LOAD, STORE, and BRANCH) per cycle if they have no dependency

Example:
```
/* source code */
for (i=1; i < N; i++) { x[i] := x[i] + c; }
```

***Before software pipelining**, each loop body will take at least 3 cycles*

```
Read x[i] to a register R;
Add R and c to R;
Write R to x[i] and CJump
```

***After software pipelining**, each loop body takes one cycle*

```
1     Read
2     Add       Read                                  ] prolog
3  L: Write     Add       Read      CJUMP L           loop body
4               Write     Add                         ] epilog
5                         Write
```


Trace Scheduling

- *Motivation of software pipelining: optimizing the loops*
- ***Motivation of trace scheduling**: optimizing frequently executed branches (traces). **Main idea**: larger block size means more opportunities for better ILP.*
- ***Algorithm for trace scheduling**:*

  Step 1. Identify the most likely execution trace (possibly using static branch prediction information), then compact the instructions in the trace as if they belong to one big basic blocks

  Step 2. add **compensation code** at the entrances and exits of the trace to restore the semantics of the original program for other traces.

  Reference: Bulldog: A Compiler for VLIW Architectures
  by John R. Ellis, MIT Press 1985.