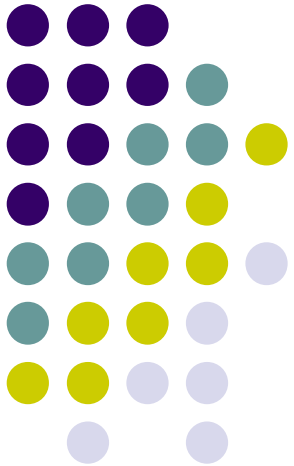
Compilation Techniques for Modern Architectures

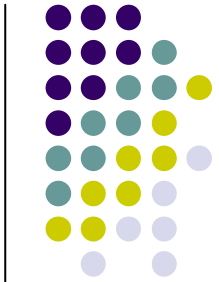
KTH Information and Communication Technology

Christian Schulte
schulte@imit.kth.se

Electronic, Computer and Software Systems
School of Information and Communication Technology
KTH – Royal Institute of Technology
Stockholm, Sweden

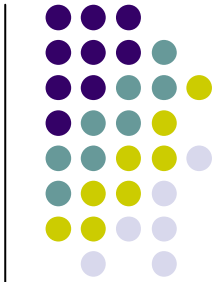


Goal of Lecture: Answer…



- What does a compiler anyway?
  - what are the basic tasks
  - how are target architectures reflected
- What to do for modern architectures?
  - memory hierarchy: registers and caches
  - various techniques to take advantage of instruction level parallelism
    - new optimization technology in compilation

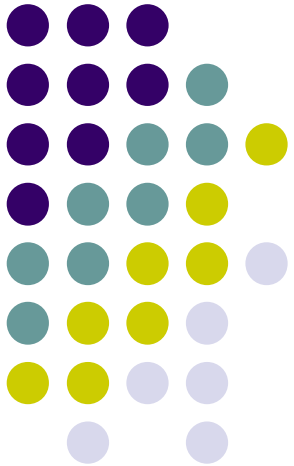
My Not So Hidden Agenda…



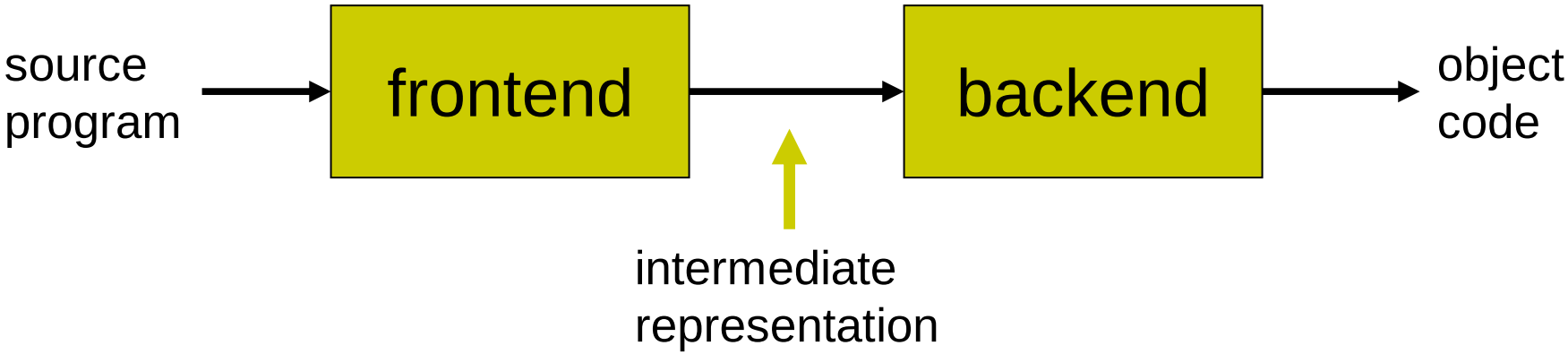
- Understanding of compilation and execution environments essential
  - computer architecture
  - programming languages
  - embedded systems
  - …
- You are welcome: 2G1533 Compilers and Execution Environments (previously 2G1508)
  - taught by me
  - period 2

Compilation

Basic structure and tasks

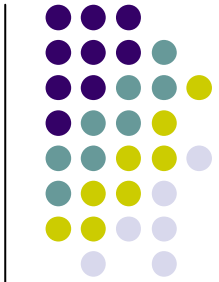


Compilation Phases



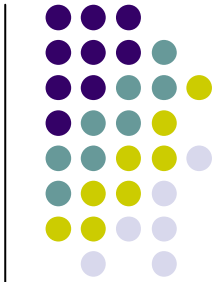
- Frontend depends on source language
- Backend depends on target architecture
- Factorize dependencies

Frontend: Tasks



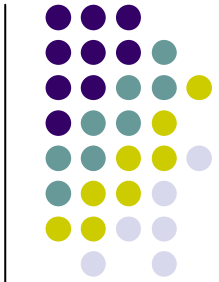
- Lexical analysis
  - how program is composed into tokens (words)
  - typical token classes: identifier, number, keywords, …
  - creates token stream
- Syntax analysis
  - phrasal structure of program (sentences)
  - grammar rules describing how expressions, statements, etc are formed
  - creates syntax tree
- Semantic analysis
  - perform identifier analysis (scope), type checking, …
  - creates intermediate representation: control flow graph

Intermediate Representation



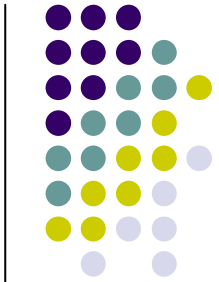
- Control flow graph
  - nodes are ***basic blocks***: simple and abstract instructions, no incoming/outgoing jumps
  - edges represent control flow: jumps, conditional jumps (loops), etc
- Basic blocks
  - typically contain data dependencies: reading of and writing to same location must be in order
  - ease reordering by conversion to SSA (static single assignment) form: new locations assigned only once

Backend: Basic Tasks



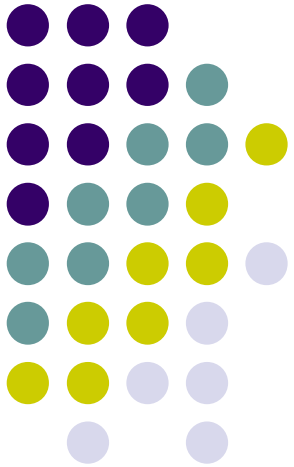
- Optimization
  - reduce execution time and program size
  - typically independent of target architecture
  - intermediate and complex component: "midend"
- Instruction selection
  - which real instructions for abstract operations
- Register allocation
  - which variables are kept in which registers?
  - which variables go to memory
- More generic: memory allocation

Optimization

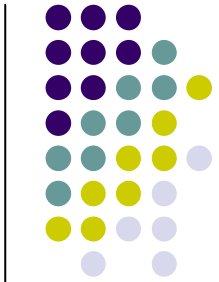


- Common subexpression elimination (CSE)
  - reuse intermediate results
- Dead-code elimination
  - remove code that can never be executed
- Strength reduction
  - make operations in loops cheaper: instead of multiplying with n, increment by n (iterated array access)
- Constant/value propagation
  - propagate information on values of variables
- Code motion
  - move invariant code out of loops
- Many, many more, …

Basic Architecture Dependent Tasks

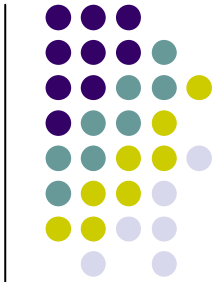


Instruction Selection: What



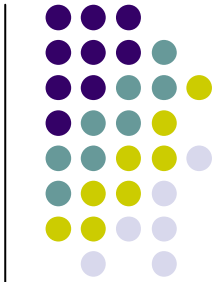
- Clearly depends on instruction set
- For abstract operations emit target machine instructions
  - several operations by one instruction (CISC)
  - one operation by several instructions (RISC)
- Depends much on regularity of instruction set
  - typically simple: RISC architectures
  - can be involved: CISC architectures
    - register classes, two address instructions, memory addressing, …

Instruction Selection: How



- Criterion: fewest instructions or fewest clock cycles
- Using matching algorithm
  - maximal munch
  - tree grammars
  - dynamic programming
- Simple optimizations: peephole optimization
  - combine, remove, change instructions on simple local rules

Register Allocation

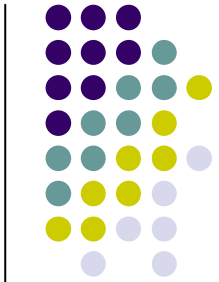


- Find for each temporary value a register if possible: depends on number of registers
- If no register free, put temporary in memory
- Use spill code: free register temporarily
  - move register to memory
  - reload register from memory
- Common technique: register allocation by graph coloring

Register Allocation: Graph Coloring

- Instruction selection assumes infinite supply of temporaries
- Compute liveness information: when is temporary used first and last
- Construct interference graph
  - nodes are temporaries
  - edge between two nodes if live at same time
- Color interference graph: no two connected nodes get same color
  - colors correspond to registers

Additional Techniques

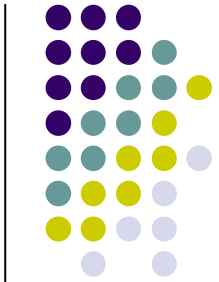


- Use precolored nodes for certain registers
- Handle different cases for registers
  - registers for passing arguments
  - register for return address
  - caller-save registers
  - callee-save registers
- Allows better register utilization
  - more registers available for intermediate results
  - spilling handled in regular and simple way

Impact of Register Allocation

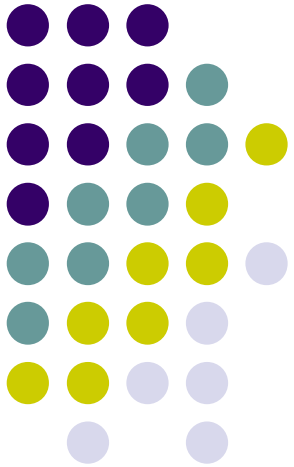
- Graph coloring works well with many (say 32) registers
- Few registers (Pentium 6-8)
  - enormous amount of spilling
  - temporaries even spilled in loops!
  - example: 163,355 instructions [Appel, George, PLDI01]
    - 32 registers 84 spill instructions
    - 8 registers 22,123 spill instructions (about 14%)
  - solution: exploit memory operands in CISC instructions

Summary: Basic Tasks



- Basic and simple tasks
  - instruction selection
  - register allocation
- Impact of target architecture
  - number of registers
  - register classes
  - instruction set
  - clock cycles per instruction
- These were the good old days…

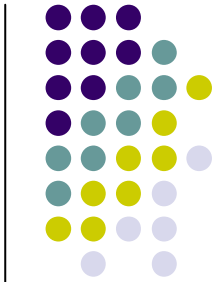
Modern Compilation Techniques



Few Aspects of Modern Compilation

- More features of target architecture need consideration for "good" code
- Memory hierarchy
  - registers and caches
- Instruction level parallelism: decrease CPI
  - restructure execution
  - avoid branching
  - schedule instructions
- How to handle ever increasing complexity and diversity… generic optimization technology…

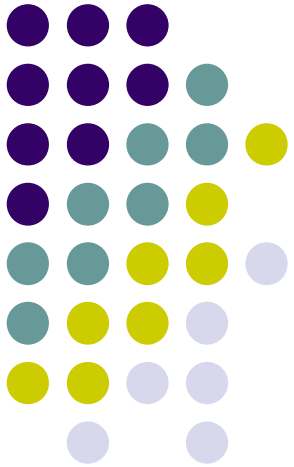
Outline of Topics



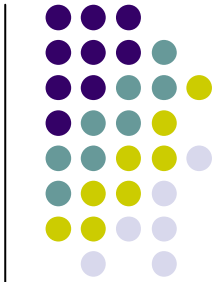
- Memory hierarchy
  - data prefetching
  - reuse cached data
- Instruction level parallelism
  - software pipelining
  - if-conversion and predicated execution
  - instruction scheduling

Data Prefetching

Based on [Vanderwiel, Lilja, CSUR 2000]



Data Prefetching

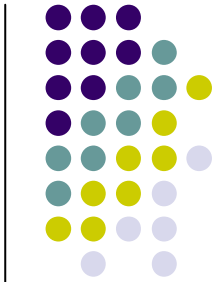


- Well known fact: caches are of utmost importance to execution efficiency
- Originally with scientific computations
  - large, dense memory operations
  - little data reuse, defeat on demand caching strategies
  - spend more than half runtime stalled [Mowry…,1992]
  - many more applications: audio, video, …
- Idea is straightforward
  - anticipate cache miss by memory access
  - execute prefetch instruction prior to access

Prefetching

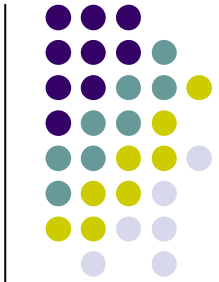
- Most microprocessors today have fetch instruction
- Common characteristics
  - nonblocking memory operation
  - has effect on cache only
  - no exceptions raised
- Related: nonblocking loads

Software Data Prefetching



- In principle, adding prefetching instructions possible
- In practice, compiler support needed
- Most often used with loops performing calculation on large arrays
  - are common cases
  - have poor cache utilization
  - often predictable array referencing patterns
  - in particular: SIMD instructions (MMX, SSE, …)

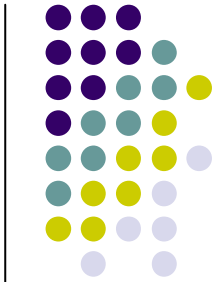
Example: Inner Product



```
for (i=0; i<n; i++)
    ip += a[i]*b[i];
```

- Inner product of vectors a, b
  - common structure
- Assume four-word cache block
  - cache miss every fourth iteration!

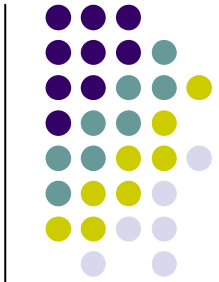
Simple Prefetching



```
for (i=0; i<n; i++) {
   fetch(&a[i+1]); fetch(&b[i+1]);
   ip += a[i]*b[i];
}
```

- Simplistic solution
  - prefetch fields for a and b for next iteration
- Problems
  - too much pre-fetching: only every forth iteration needed
  - testing `i % 4 == 0` would degrade performance

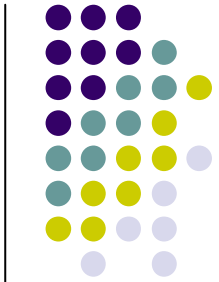
Loop Unrolling



```
for (i=0; i<N; i+=4) {
   fetch(&a[i+4]); fetch(&b[i+4]);
   ip+=a[i]*b[i];     ip+=a[i+1]*b[i+1];
   ip+=a[i+2]*b[i+2]; ip+=a[i+3]*b[i+3];
}
```

- Unroll loop by cache-defined factor r (here = 4): words to be prefetched per iteration
  - replicate body (without fetch) 4 times
  - increasing loop stride to 4
- Removes most cache misses
  - but cache misses in first iteration
  - unnecessary prefetches in last iteration of unrolled loop

Using Software Pipelining



- Take parts of loop body: put before and after loop
  - prolog (before): prefetch for first iteration
  - epilog (after): computation without prefetching
- Still improvement possible
  - if loop body small need to make prefetching for more than one iteration
  - due to latency of cache
  - example: loop iteration time 45 cycles, average miss latency 100 cycles: prefetch for 3 iterations to be on safe side

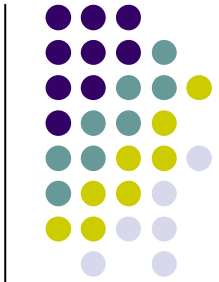
Final Transformed Loop: Prolog

```
fetch(&ip);
for (i=0; i<12; i+=4) {
   fetch(&a[i]); fetch(&b[i]);
}
```

...

- Performs only prefetching
  - for three iterations

Final Transformed Loop: Epilog

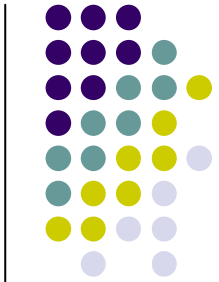


…

```
for ( ; i<N; i++)
    ip+=a[i]*b[i];
```

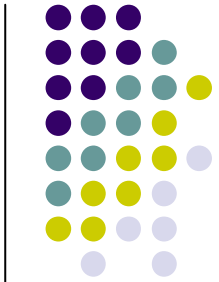
- Epilog performs only computation

Final Transformed Loop



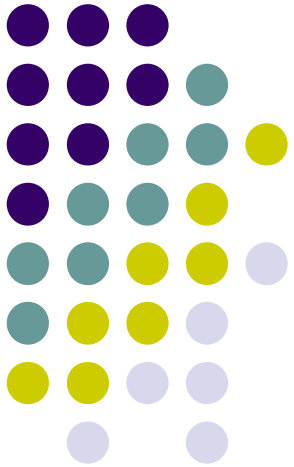
```
fetch(&ip);
for (i=0; i<12; i+=4) {
   fetch(&a[i]); fetch(&b[i]);
}
for (i=0; i<N-12; i+=4) {
   fetch(&a[i+4]); fetch(&b[i+4]);
   ip+=a[i]*b[i];     ip+=a[i+1]*b[i+1];
   ip+=a[i+2]*b[i+2]; ip+=a[i+3]*b[i+3];
}
for ( ; i<N; i++)
   ip+=a[i]*b[i];
```

Impact of Prefetching

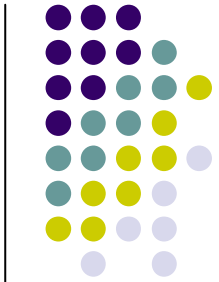


- Can be generalized to nested loops
- Scientific benchmarks on PowerPC 601
  - runtime improvement by less than 12%
  - up to 22%
- SPECfp95 benchmarks on PA8000
  - six: between 26% and 98% faster
  - others: less than 7% faster, one slower by 12%
- Downside
  - fetch instruction increase register pressure in loop
  - code expansion: pressure on instruction cache

Reusing Cached Data

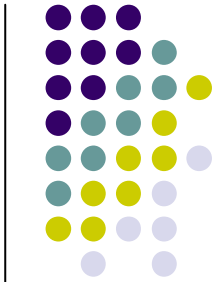


Loop Interchange



- Using cache effectively: reuse cached data
- Nested loops
  - successive iterations often reuse cached/adjacent data
  - innermost loop: many cache hits
  - outer loops: might have many misses due to inner loop
- Solution: try to reorder loops

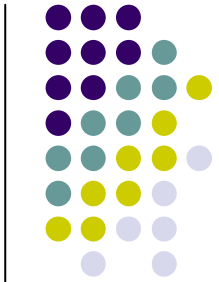
Example: Cache Misses



```
for (i=0; i<n; i++)
   for (j=0; j<m; j++)
      for (k=0; k<p; k++)
         a[i,j,k] =
     (b[i,j-1,k]+b[i,j,k]+b[i,j+1,k])/3;
```

- `b[i,j+1,k]` reused in two next iterations of `j`-loop
- but k-loop brings 3p elements of b and p elements of a to the cache
  - possibly resulting in cache miss on next iteration of `j`-loop

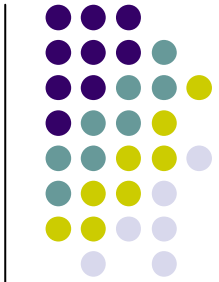
Example: Change Loops



```
for (i=0; i<n; i++)
    for (k=0; k<p; k++)
        for (j=0; j<m; j++)
            a[i,j,k] =
          (b[i,j-1,k]+b[i,j,k]+b[i,j+1,k])/3;
```

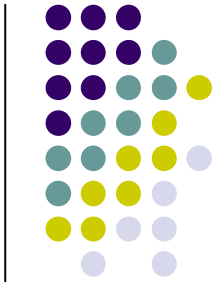
- `b[i,j,k]` and `b[i,j-1,k]` always cache hits
- obviously legal: no data dependencies among iterations

Iteration Dependencies



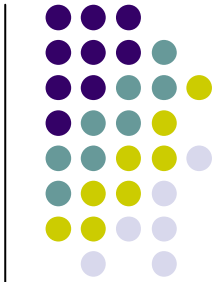
- Iteration (j,k) depends on (j',k'), if
  - (j',k') computes values used by (j,k)
    read-after-write
  - (j',k') stores values overwritten by (j,k)
    write-after-write
  - (j',k') uses values written by (j,k)
    write-after-read
- If interchanges loops compute (j',k') before (j,k) and there is a dependence: interchange illegal

Blocking



- Blocking reorders computation
  - complete computation on one block of data before moving to next block
- Example: matrix multiplication

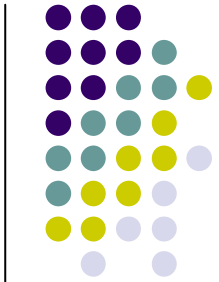
Example: Matrix Multiply



```
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      for (k=0; k<n; k++)
         c[i,j] += a[i,k] * b[k,j];
```

- If a and b fit into the cache, k-loop has no misses
  - only one possible miss for `c[i,j]`
- Suppose cache only holds `2*c*n` matrix elements
  - only misses for `b[k,j]` in inner loop!
  - since last access to `b[k,j]` all of b has been accessed!
- Loop interchange cannot help: either a, b, or c suffers

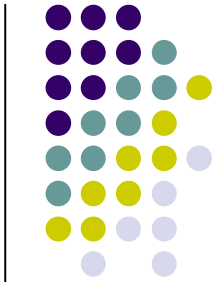
Example: Compute Blocks



```
for (i=i0; i<i0+d; i++)
   for (j=j0; j<j0+d; j++)
      for (k=0; k<n; k++)
         c[i,j] += a[i,k] * b[k,j];
```

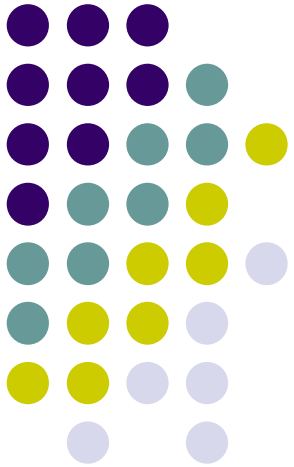
- Only compute d×d blocks of the matrix
  - choose d such that all required elements fit cache
  - required: d×n from a and n×d from b
- Wrap loops around computing all d×d blocks

Summary

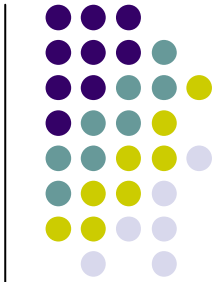


- Transform memory intensive loops to reuse cached data as much as possible
- Loop interchange
  - change order of independent loops
  - inner loop gets most cache hits
- Blocking
  - chop data structures into parts that fit cache
  - requires non data-dependent loops

Software Pipelining

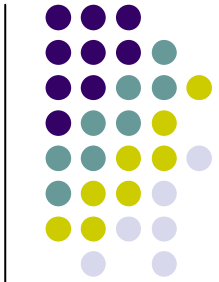


Software Pipelining



- Already seen briefly in data prefetching
- Goal: exploit instruction level parallelism in loops
  - avoid data dependencies in single loop iteration
- General idea: each iteration in software pipelined loop made from instructions from different iterations of original loop
  - less data dependencies, higher degree of parallelism

Example: Bad Loop



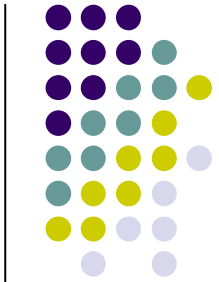
```
for (i=0; i<n; i++) {
    x=a[i]; y=x+s; a[i]=y;
}
```

- Problem with this loop are data dependencies
- Instructions for

  `y=x[s]` and `a[i]=y`

  will stall

Bad Loop: Some Iterations



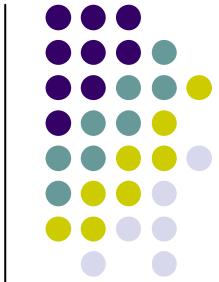
…

```
x=a[i];   y=x+s; a[i]  =y;
x=a[i+1]; y=x+s; a[i+1]=y;
x=a[i+2]; y=x+s; a[i+2]=y;
```

…

- Let us not worry about start/end

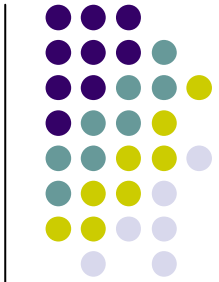
Bad Loop: Iterations



```
...
x=a[i];   y=x+s; a[i]  =y;
x=a[i+1]; y=x+s; a[i+1]=y;
x=a[i+2]; y=x+s; a[i+2]=y;
...
```

- Select one instruction from each iteration
- Remove the rest

Bad Loop: Iterations



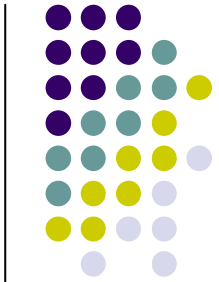
...

```
a[i]=y; y=x+s; x=a[i+2];
```

...

- Select one instruction from each iteration
- Remove the rest

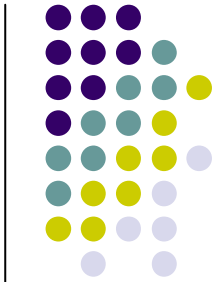
Good Loop: Rewritten



```
for (int i=0; i<n-2; i++) {
    a[i]=y;   % from iteration i
    y=x+s;    % from iteration i+1
    x=a[i+2]; % from iteration i+2
}
```

- Requires prolog and epilogue for correctness
- No data dependencies between instructions of single iteration!

Good Loop: See It Again

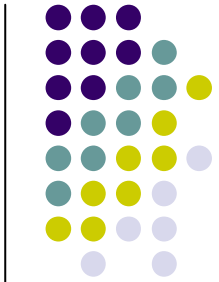


…

```
a[0]=y; y=x+s; x=a[2];
a[1]=y; y=x+s; x=a[3];
a[2]=y; y=x+s; x=a[4];
```

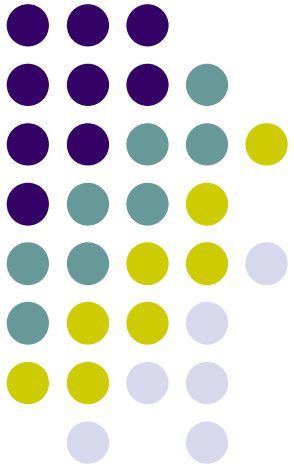
…

Summary

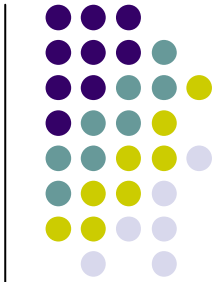


- Register management can be tricky
- For small loops several iterations might be required before result available
- Difference
  - loop unrolling: removes overhead of loop (branch, increment)
  - software pipelining: removes stalling across all loop iterations

If-Conversion Predicated Execution



Conditional Instructions

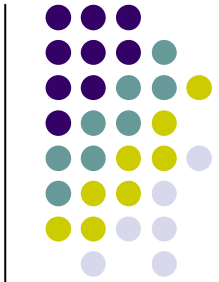


- Conditional instruction: instruction refers to condition
  - if condition true, execute normally
  - if condition false, perform noop
- Most newer architectures have conditional instructions
- Most common case: conditional move
- Slogan: turn control dependency into data dependency
- Also: predicated execution

Why Conditional Instructions

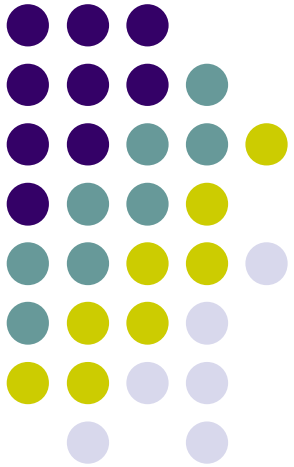
- Eliminate branch
- Possibly improve pipeline behavior
- Important: move place where dependency is resolved from beginning of pipeline to end (conditional move)
- Very useful for software pipelining
  - increase potential for ILP even further
  - possible speedup 34% [Warter, …,1993]
- Very important: EPIC

If-Conversion on Itanium



- Itanium fully supports predication
  - set of 64 predicate registers
  - variety of predicate computing instructions
- Major benefit: elimination of hard to predict branches
- Intel's compiler performs predication
  - used together with software pipelining

Instruction Scheduling



Instruction Scheduling

- Compilation initially only determines which instructions to execute
  - instruction scheduling determines order
  - order is free unless there are data dependencies
- Clear goal
  - schedule instructions so late that they will not stall due to data or functional dependencies
  - finish all instructions as early as possible
  - extremely important: urge to compute optimal schedules!

Approaches to Scheduling

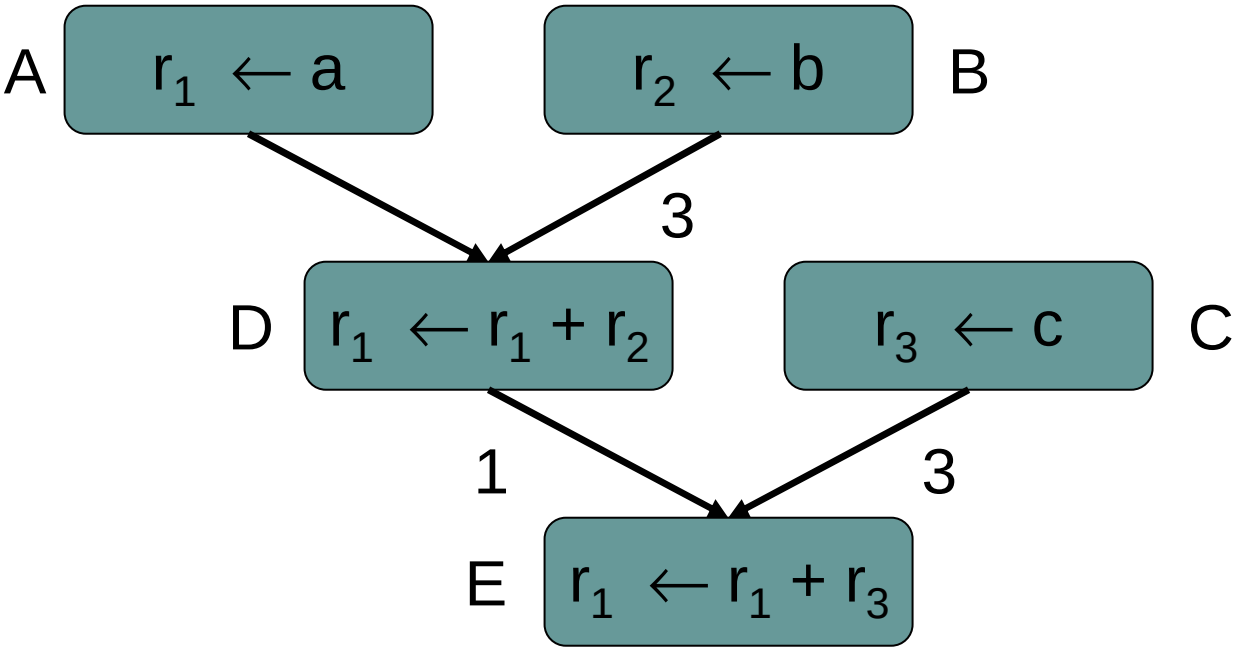
- Latency between instructions i and j
  - difference between cycle in which i executes
  - and first cycle in which j computes
- Difficulty depends on maximum latency
  - latency of two: polynomial
  - single-issue processor with latency of at least three: polynomial
  - general case: NP-complete
- Approximations do not compute optimal schedule for even small basic blocks

Problem

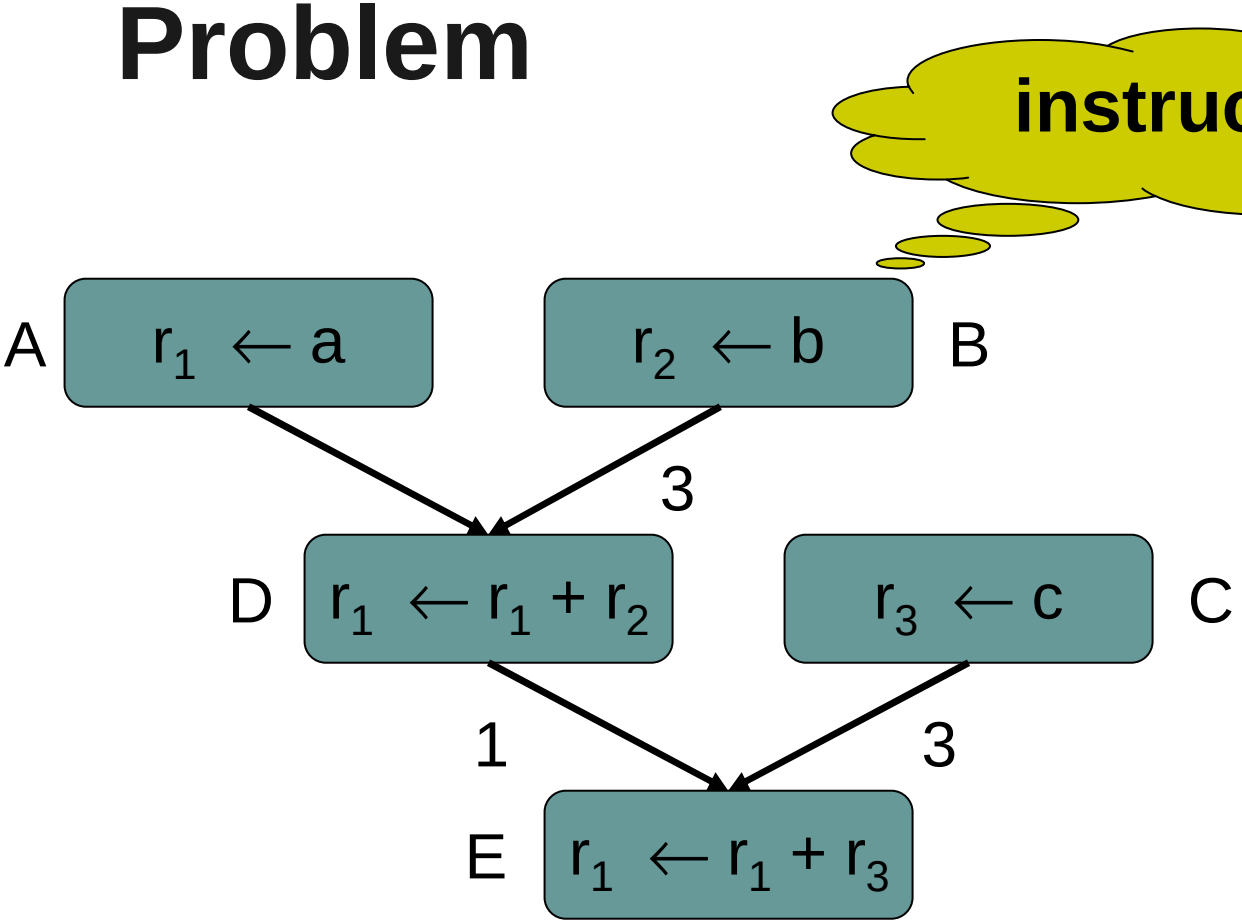
- Consider only instructions per basic block
- Given
  - n instructions
  - latency between dependent instructions *i* and *j*

    $l(i,j)>0$
- Find time for each instruction such that
  - at most r instructions issued at same cycle
  - latency dependencies are satisfied

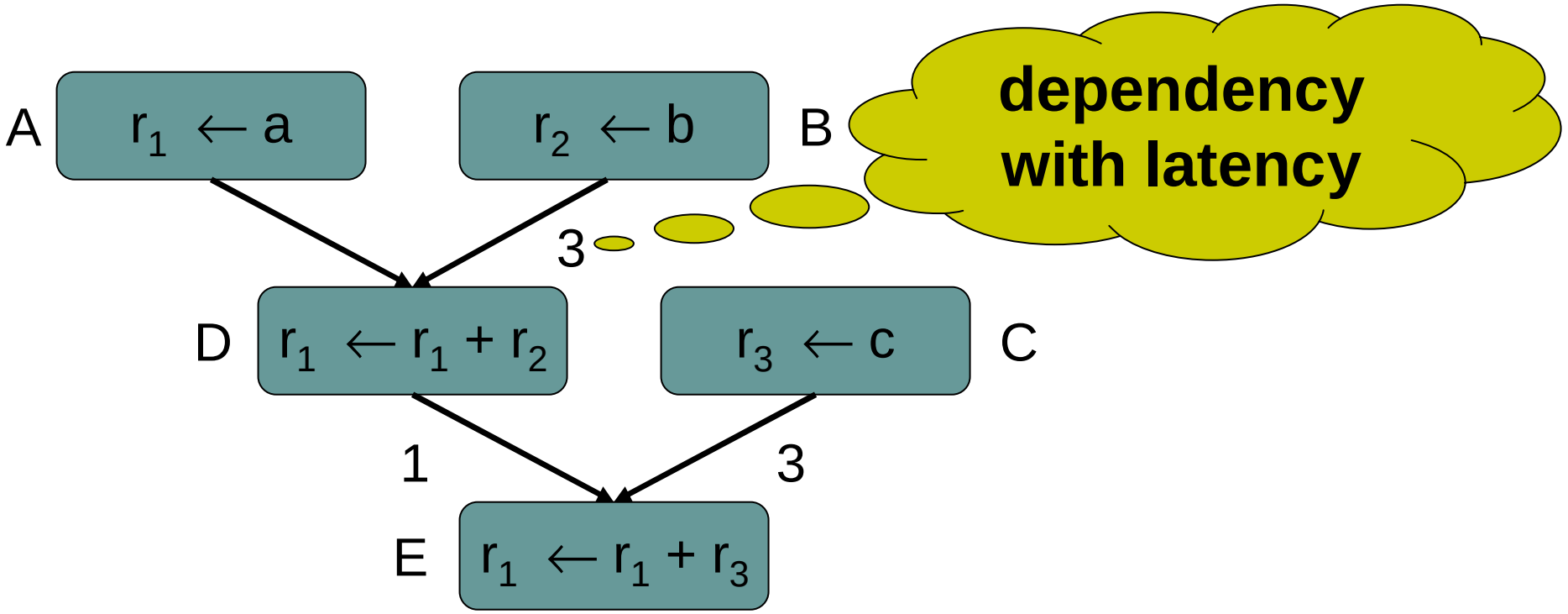
Problem


A
$r_1 \leftarrow a$
$r_2 \leftarrow b$
B
3
D
$r_1 \leftarrow r_1 + r_2$
$r_3 \leftarrow c$
C
1
3
E
$r_1 \leftarrow r_1 + r_3$

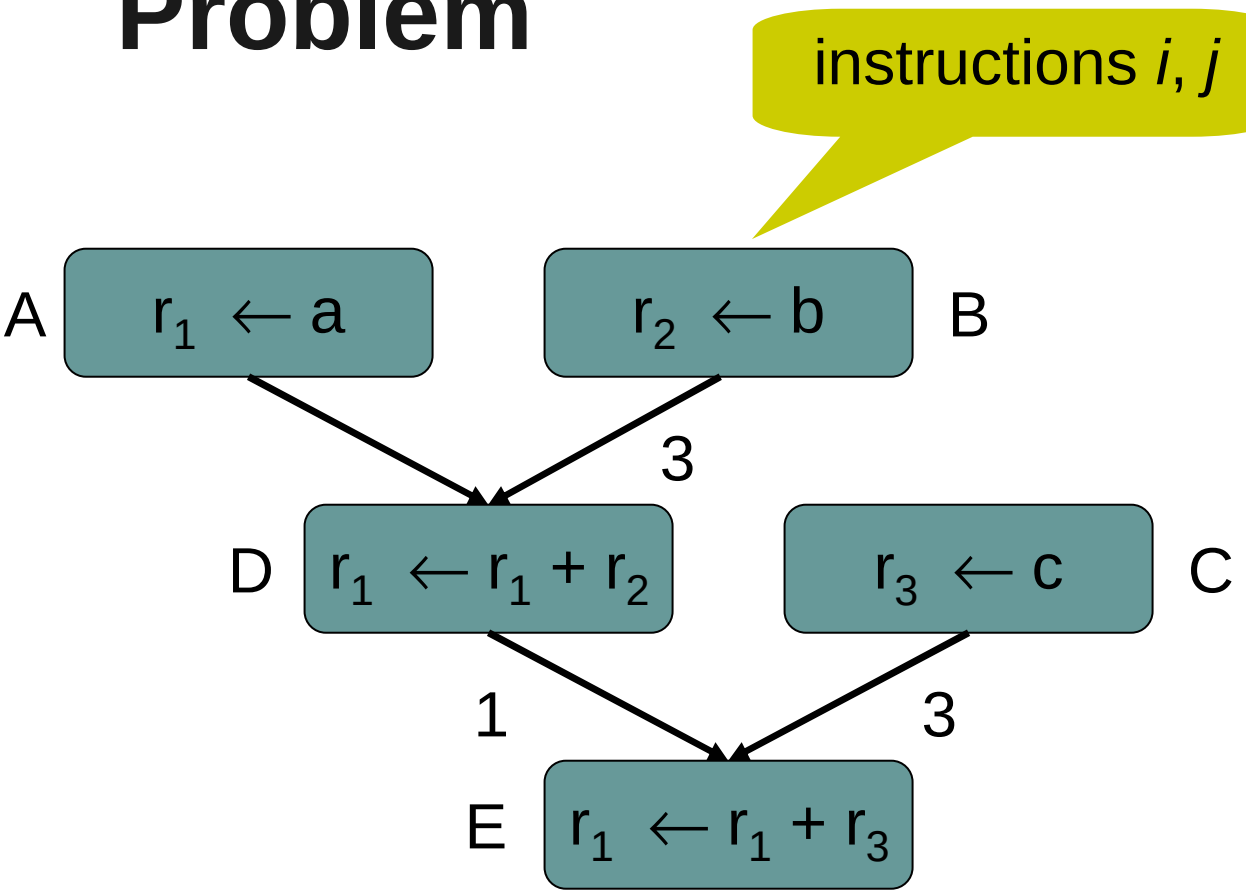
Problem


instructions
A
$r_1 \leftarrow a$
$r_2 \leftarrow b$
B
3
D
$r_1 \leftarrow r_1 + r_2$
$r_3 \leftarrow c$
C
1
3
E
$r_1 \leftarrow r_1 + r_3$

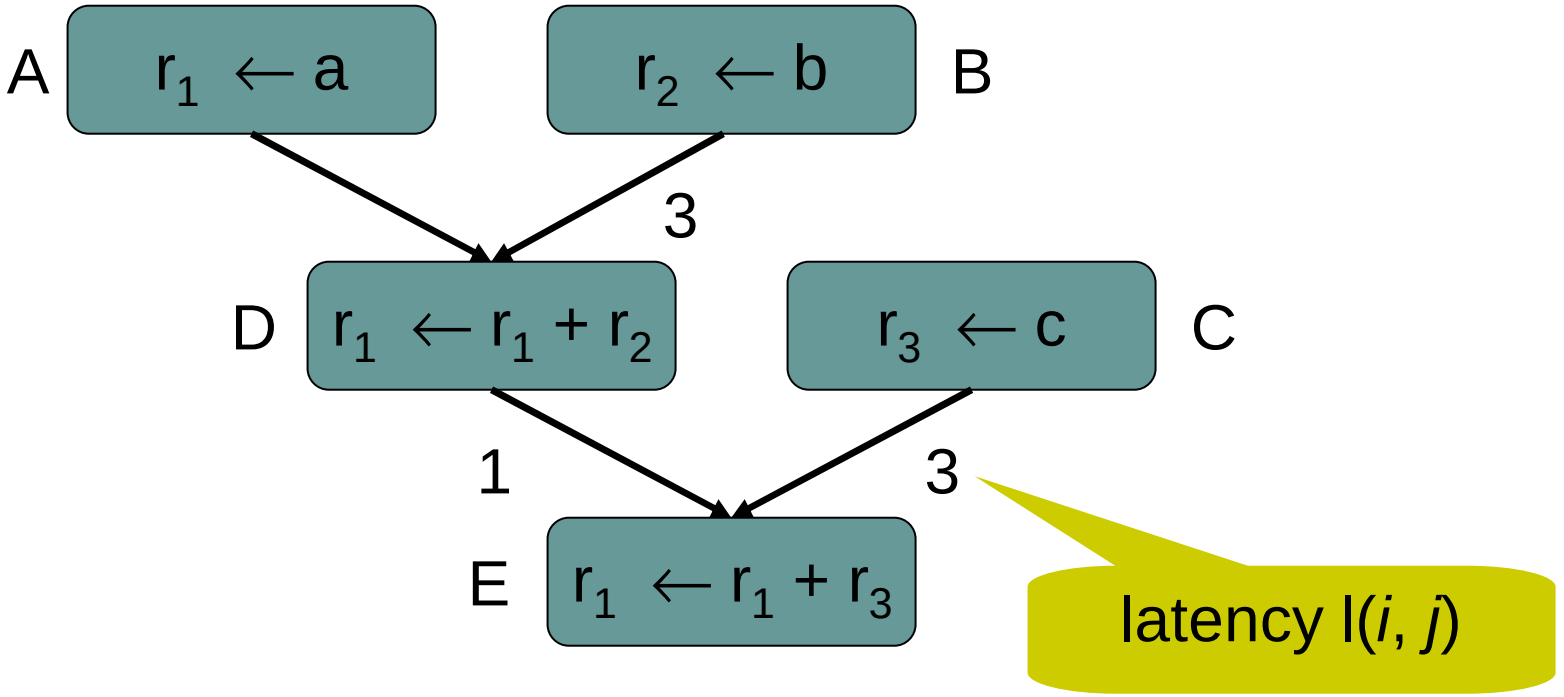
Problem


A
$r_1 \leftarrow a$
$r_2 \leftarrow b$
B
dependency with latency
3
D
$r_1 \leftarrow r_1 + r_2$
$r_3 \leftarrow c$
C
1
3
E
$r_1 \leftarrow r_1 + r_3$

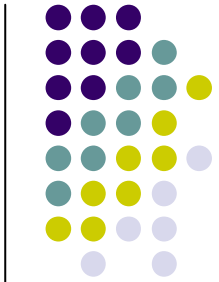
Problem


instructions *i*, *j*
A
$r_1 \leftarrow a$
$r_2 \leftarrow b$
B
3
D
$r_1 \leftarrow r_1 + r_2$
$r_3 \leftarrow c$
C
1
3
E
$r_1 \leftarrow r_1 + r_3$

Problem


A
$r_1 \leftarrow a$
$r_2 \leftarrow b$
B
3
D
$r_1 \leftarrow r_1 + r_2$
$r_3 \leftarrow c$
C
1
3
E
$r_1 \leftarrow r_1 + r_3$
latency l(*i*, *j*)

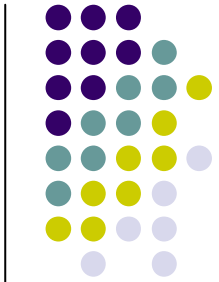
Using ILP



- Integer linear programming technique to optimally solve problems described by
  - linear equations
  - linear inequalities

  and optimization function [unused here]
- Extremely powerful method used for combinatorial optimization
- Reason here: more and more problems in compilation solved by ILP [hot, hot, hot]

An ILP Model for Instruction Scheduling



- ILP model
  - variables taking some values
  - inequalities and equations which represent constraints
- Optimization: optimal schedule
  - compute upper bound U on clock cycles: use simple method to compute feasible schedule
  - computer lower bound L on clock cycles: assume that each cycle r instructions are issued, disregard dependencies
  - if U==L, schedule is optimal
  - otherwise, try finding schedule for U-1, U-2, … clock cycles with ILP until no more schedules exist: last is best

Finding an m Clock Cycle Schedule

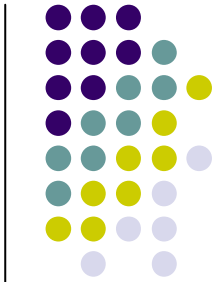
- For an n instruction basic block introduce variables

  $$x(i,c) \in \{0,1\}$$

  where $1 \leq i \leq n$ and $1 \leq j \leq m$

  - $x(i,c) = 1 \Leftrightarrow$ instruction i is scheduled at clock cycle c
  - $x(i,c) = 0 \Leftrightarrow$ i not scheduled at cycle c
- Constrain x(i,c) that only valid schedules are produced

Constraints: Must Schedule



- Each instruction i must be scheduled exactly once
- One constraint for each instruction i:

$$\sum_{c=1}^{m} x(i,c) = 1$$

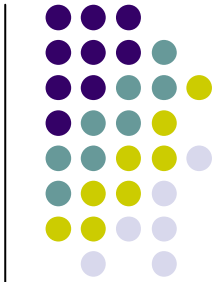
Constraints: Processor Issue

- Allow at most r instructions to be issued by processor at each clock cycle
- For each clock cycle c:

$$\sum_{i=1}^{n} x(i,c) \leq r$$

- If processor has issue restrictions, impose constraint per instruction type

When Is an Instruction Scheduled

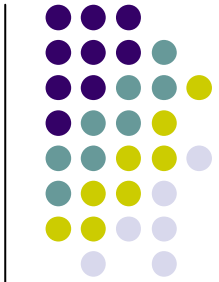


- To express dependency constraints: need clock cycle at which instruction is issued
- For instruction i

$$\sum_{c=1}^{m} c \cdot x(i,c)$$

- Only one summand is different from zero: must schedule constraint

Constraints: Latency



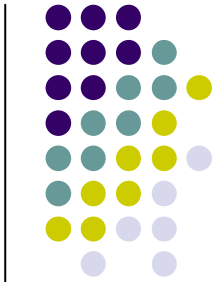
- Given that j depends on i with latency l(i,j), j can start earliest after start time of i plus latency

$$\sum_{c=1}^{m} c \cdot x(i,c) + l(i,j) \leq \sum_{c=1}^{m} c \cdot x(j,c)$$

Making It Work

- Add additional information on scheduling ranges
- Take structure of dependency graph into account
  - removing redundant edges
  - partition graph
  - …

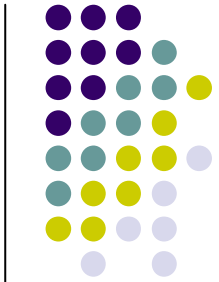
Experimental Results



- Total of 7,402 basic blocks taken SPECint 95
- Around 100 sec scheduling time
- All solved optimally
- Cycle count improved 66

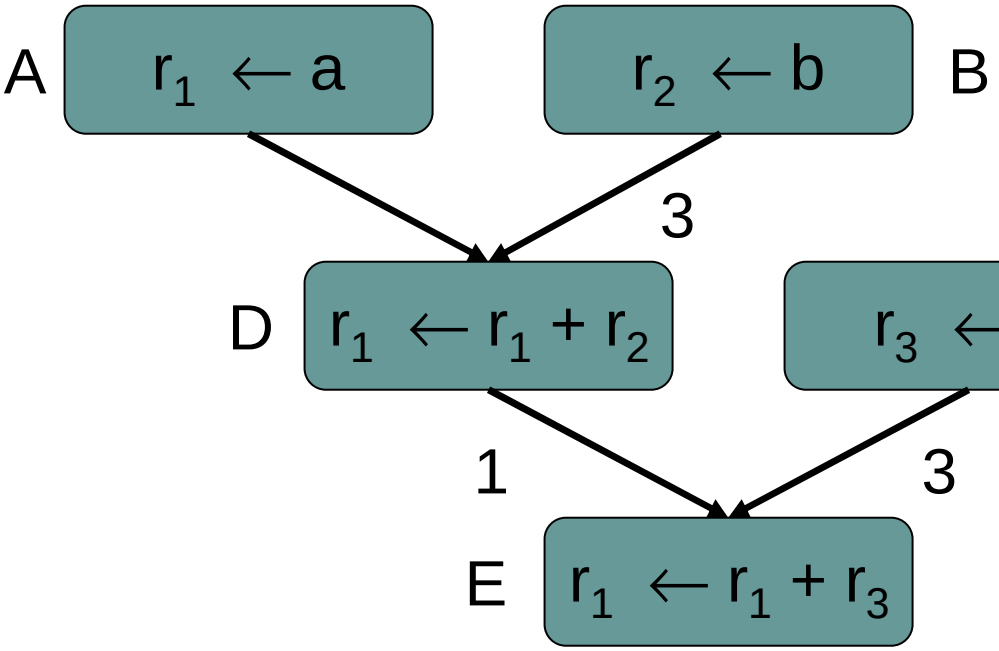
- Source: K. Wilken, J. Liu, M. Heffernan, Optimal Instruction Scheduling Using Integer Programming, PLDI 2000.

Different Approach



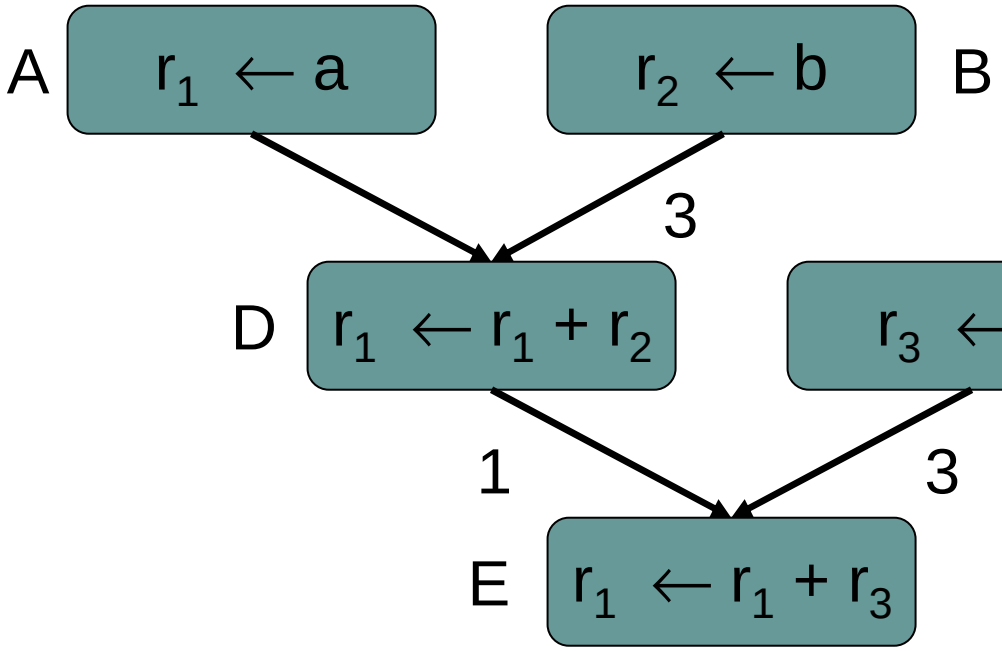
- Instead of using ILP use constraint programming
- Simpler model (restricted to single issue)
  - one variable per instruction giving start time
  - one constraint requiring all start times to be distinct
  - one constraint per latency relation
- Best paper CP 2001, Peter van Beek and Kent Wilken, Fast Optimal Scheduling for Single-issue Processors with Arbitrary Latencies, 2001.

Problem: Single Issue



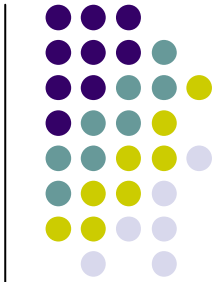
- Find issue time s(*i*)
  1. $i \neq j \Rightarrow s(i) \neq s(j)$
  2. $s(i) + l(i,j) \leq s(j)$
- Minimize
  max s(*i*)

Problem Is also Model



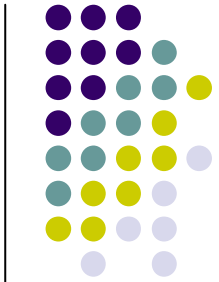
- Find issue time s(*i*)
  1. $i \neq j \Rightarrow s(i) \neq s(j)$
  2. $s(i)+l(i,j) \leq s(j)$
- Minimize
  max s(*i*)

Model



- All issue times must be distinct
  - use single `distinct` constraint (as in SMM)
  - is resource constraint or unit duration
- Latency constraints
  - precedence constraints (as before)
  - difference: duration ⇆ latency

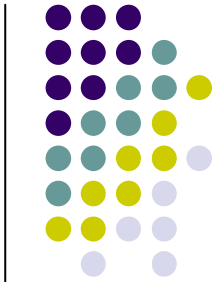
Making It Work



- Only propagate bounds information
  - relevant for `distinct`
- Add redundant constraints
  - regions: additional structure in DAG
  - successor and predecessor constraints

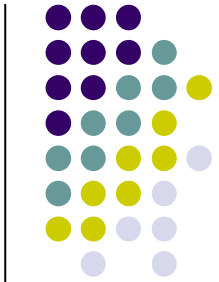
    [special case of edge-finding]

Results



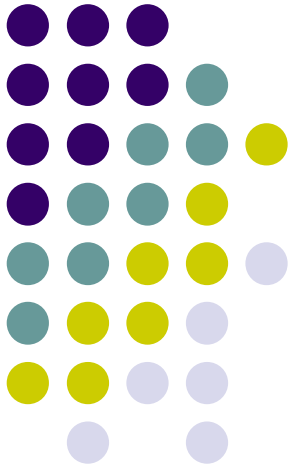
- Tested with gcc
- SPEC95 FP
- Large basic blocks
  - up to 1000 instructions
- Optimally solved
  - less than 0.6% compile time increase
  - limited static improvement (< 0.1%)
  - better dynamic impact (loops)
- Far better than ILP approach

More Related Work



- Use ILP to schedule instructions for Itanium
  - first macro scheduling
  - then bundling to instruction groups
- Offers considerable improvement on assembly files created by Intel's compiler
- In: Kästner, Winkel, ILP-based Instruction Scheduling for IA-64, LCTES, 2001.
- Also Intel uses ILP to optimize software library for transcedental functions

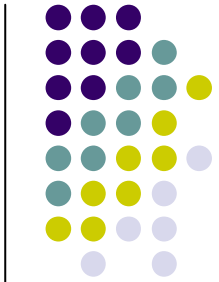
Summary



Basic Architecture Specific Techniques

- Register allocation
  - depends on number of available registers
  - works well for large number of registers (all but i86)
  - difficult for small number of registers
- Instruction selection
  - match abstract operations to instructions
  - depends on instruction set
  - straightforward for RISC
  - can become involved for CISC

Advanced Techniques



- Reflecting the memory hierarchy: caches
  - prefetching for loops
  - reuse cached data: loop interchange, blocking
- Exploiting instruction level parallelism
  - avoid branches by if-conversion
  - increase available parallelism by software pipelining
  - compute good, even optimal schedules
- Advent of modern optimization technology in compilation