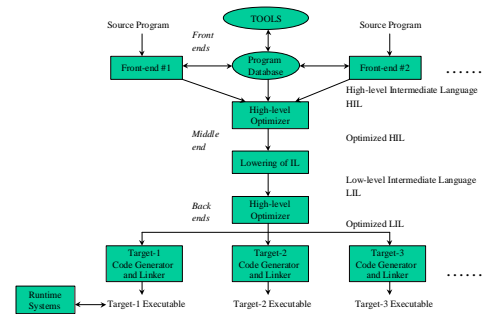


Code Optimization in Modern Compilers

Krishna V. Palem
Department of Computer Science, NYU

Structure of Optimizing Compilers



© Krishna Palem, 1998

Please do not distribute

2

Front-end

1. Scanner - converts input character stream into stream of lexical tokens
2. Parser - derives syntactic structure (parse tree, abstract syntax tree) from token stream, and reports any syntax errors encountered
3. Semantic Analysis - generates intermediate language representation from input source program and user options/directives, and reports any semantic errors encountered

© Krishna Palem, 1998

Please do not distribute

3

High-level Optimizer

The following steps may be performed more than once:

1. Global intraprocedural and interprocedural analysis of source program's control and data flow
2. Selection of high-level optimizations and transformations
3. Update of high-level intermediate language

Note: this optimization phase can optionally be by passed, since its input and output use the same intermediate language

© Krishna Palem, 1998

Please do not distribute

4

Lowering of Intermediate Language

- Linearized storage mapping of variables
- Array/structure references → load/store operations
- High-level control structures → low-level control flow

© Krishna Palem, 1998

Please do not distribute

5

Region based compilation

Background

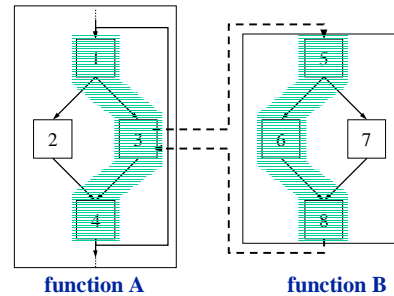
- Implications of function based compilation
 - hidden memory aliasing behavior
 - optimization
 - control flow structure
- Aggressive optimizing compiler reduces the side effect by using function inlining

© Krishna Palem, 1998

Please do not distribute

7

Example: Function based compilation



© Krishna Palem, 1998

Please do not distribute

8

Problem with function based compilation

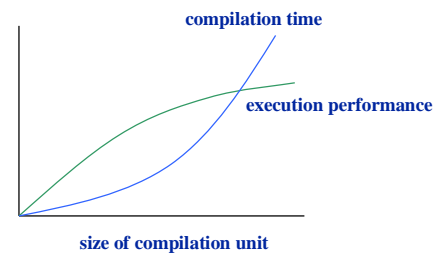
- In function based compilation..
 - While compiling function A, the scope of the compilation is limited to it.
 - While compiling function B, the fact that function B is part of a cycle is hidden
- Solution
 - Function Inlining
 - Code expansion is large

© Krishna Palem, 1998

Please do not distribute

9

The effect of compilation size



© Krishna Palem, 1998

Please do not distribute

10

Features of region based compilation

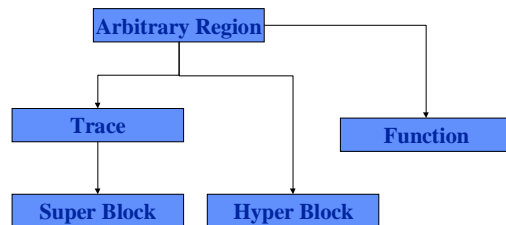
- Compiler is in complete control over the size and contents of the compilation unit
- The size of the compilation unit is typically smaller than functions
 - reducing the impact of the algorithmic complexity
- The use of profiling information to select regions allows the compiler to select compilation units that more accurately reflect dynamic behavior of the program
 - allow the compiler to produce more compact optimized code
- Each region may be compiled completely before compilation proceeds to next region
 - all function-oriented compiler transformations may be applied.

© Krishna Palem, 1998

Please do not distribute

11

Types of regions



© Krishna Palem, 1998

Please do not distribute

12

Arbitrary region formation

- Select a seed block, *s*, which is the most frequently executed block not yet in a region
- find all desirable successor *y* of a block *x* for all *x* in a region where Succ(*x*,*y*) satisfies,

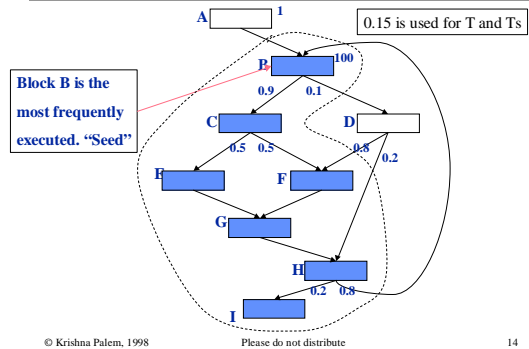
$$Succ(x,y) = (\frac{W(x \rightarrow y)}{W(x)} \geq T) \& (\frac{W(y)}{W(s)} \geq T_s)$$

W(*x*) : frequency

x → *y* : control flow edge from *x* to *y*

- How to decide *T* & *T_s*?

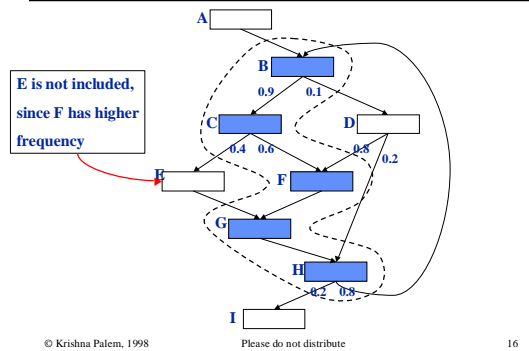
Example: Arbitrary region



Trace

- Ordered sequence of basic blocks connected by control flow
- Choose a "seed" block with the highest expected frequency
- From seed, expand forward/backward in the control flow graph, picking the unscheduled block with the highest expected frequency

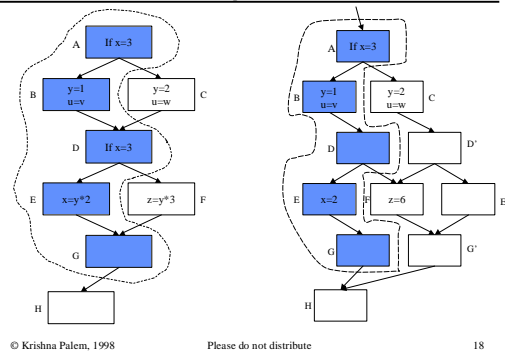
Trace Picking



Super Block

- A trace with a single entry but potentially many exits
- Simplifies code motion during scheduling
 - upward movements past a side entry within a block are pure speculation
 - downward movements past a side entry within a block are pure replication
 - will be covered in detail in scheduling session
- Two step formation
 - Trace picking
 - Tail duplication

Super block formation and tail duplication



Hyper block

- Single entry/ multiple exit set of predicated basic block (if conversion)
- There exist no incoming control flow arcs from outside basic blocks to the selected blocks other than the entry block
- There exist no nested inner loops inside the selected blocks

© Krishna Palem, 1998

Please do not distribute

19

Hyper block formation procedure

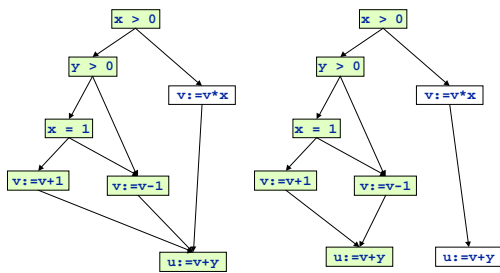
- Tail duplication
 - remove side entries
- Loop Peeling
 - create bigger region for nested loop
- Node Splitting
 - Eliminate dependencies created by control path merge
 - large code expansion
- After above three transformations, perform if conversion

© Krishna Palem, 1998

Please do not distribute

20

Tail Duplication

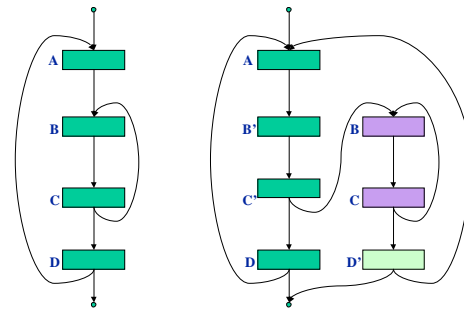


© Krishna Palem, 1998

Please do not distribute

21

Loop Peeling

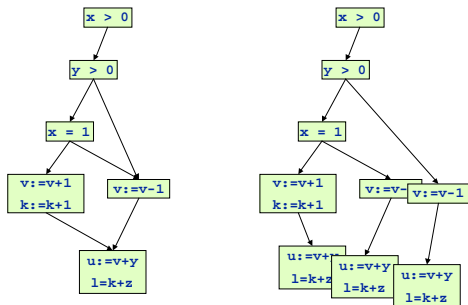


© Krishna Palem, 1998

Please do not distribute

22

Node Splitting

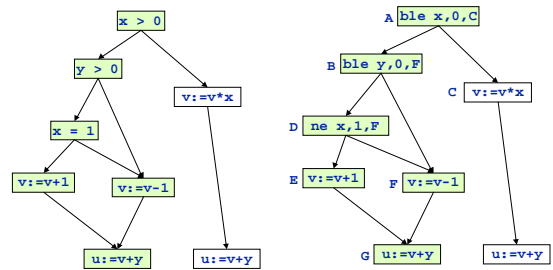


© Krishna Palem, 1998

Please do not distribute

23

Assembly Code

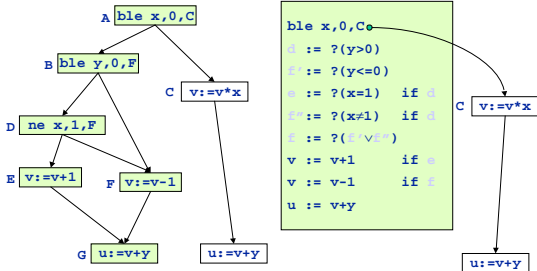


© Krishna Palem, 1998

Please do not distribute

24

If conversion



© Krishna Palem, 1998

Please do not distribute

25

Region Size Control

- Experiment shows that 85% of the execution time was contained in regions with fewer than 250 operations, when region size is not limited.
- There are some regions formed with more than 10000 operations. (May need limit)
- How can I decide the size limit?
 - Open Issue

© Krishna Palem, 1998

Please do not distribute

26

Additional references

- Region Based Compilation: An Introduction and Motivation, *Richard Hank, Wen-mei Hwu, Bob Rau*, Micro-28, 1995
- Effective compiler support for predicated execution using the hyperblock, *Scott Mahlke, David Lin, William Chen, Richard Hank, Roger Bringmann*, Micro-25, 1992

© Krishna Palem, 1998

Please do not distribute

27

Dependence Analysis

Data and Control Dependences

Motivation: identify only the essential control and data dependences which need to be obeyed by transformations for code optimization.

Program Dependence Graph (PDG) consists of

1. Set of nodes, as in the CFG
2. Control dependence edges
3. Data dependence edges

Together, the control and data dependence edges dictate whether or not a proposed code transformation is legal.

© Krishna Palem, 1998

Please do not distribute

29

Control Dependence Analysis

We want to capture two related ideas with control dependence analysis of a CFG:

1. Node Y should be control dependent on node X if node X evaluates a predicate (conditional branch) which can control whether node Y will subsequently be executed or not. This idea is useful for determining whether node Y needs to wait for node X to complete, even though they have no data dependences.

© Krishna Palem, 1998

Please do not distribute

30

Control Dependence Analysis (contd.)

- Two nodes, Y and Z , should be identified as having identical control conditions if in every run of the program, node Y is executed if and only if node Z is executed. This idea is useful for determining whether nodes Y and Z can be made adjacent and executed concurrently, even though they may be far apart in the CFG.

© Krishna Palem, 1998

Please do not distribute

31

Control Dependence Analysis (contd.)

- Control Dependence is formally defined using postdominators
 - Node W *postdominates* another node $V = W$ if and only if every directed path from V to $STOP$ in CFG contains W .
 - Define $pdom(V) = \{W \mid W \text{ postdominates } V\}$, the set of *postdominators* of node V
 - Consider any simple path from V to $STOP$ containing V 's postdominators in the order W_1, \dots, W_k . Then all simple paths from V to $STOP$ must contain V 's postdominators in the same order. The element closest to V , $W_1 = ipdom(V)$, is called the *immediate postdominator* of V .

© Krishna Palem, 1998

Please do not distribute

32

Control Dependence: Definition

[Ferrante et al, 1987]

Node Y is *control dependent* on node X with label L in CFG if and only if

- there exists a nonnull path $X \rightarrow Y$, starting with the edge labeled L , such that Y post-dominates every node, W , strictly between X and Y in the path, and
- Y does not post-dominate X

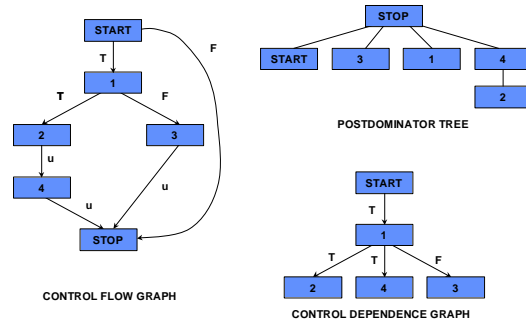
Y is control dependent on X only if X can *directly* affect whether Y is executed or not; *indirect* control dependence can be defined as the transitive closure of control dependence

© Krishna Palem, 1998

Please do not distribute

33

Example: acyclic CFG and its Control Dependence Graph

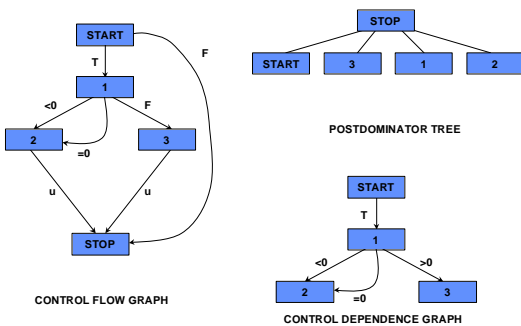


© Krishna Palem, 1998

Please do not distribute

34

Example: CFG with multi-way branch and its CDG



© Krishna Palem, 1998

Please do not distribute

35

Algorithm for Computing Control Dependence

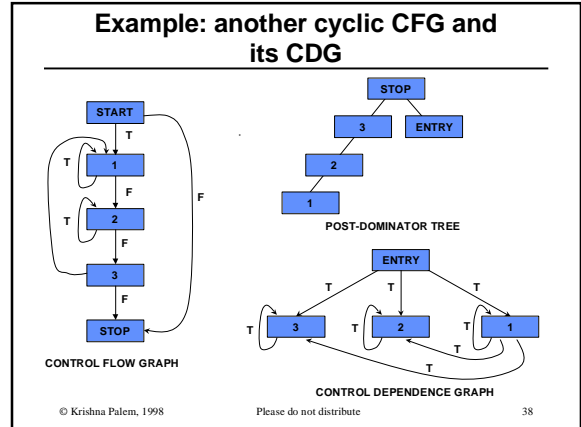
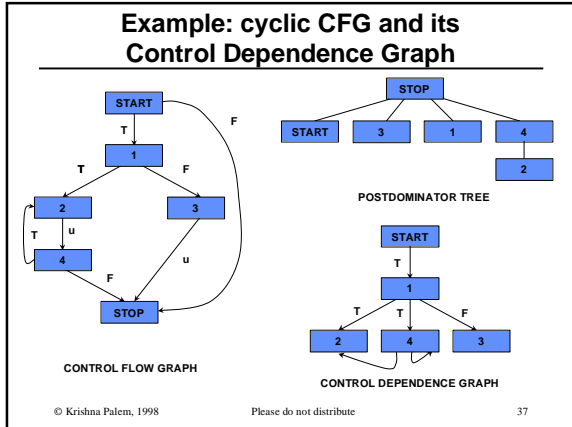
Given node X and branch label L , all control dependence successors can be enumerated as follows:

- $Z \leftarrow$ CFG successor of node X with label L
- while** $Z \neq ipdom(X)$ **do**
 - Z is control dependent on X with label L — process Z as desired $*$
 - $Z \leftarrow ipdom(Z)$
- end while**

© Krishna Palem, 1998

Please do not distribute

36



Properties of Control Dependence

- CDG is a *tree* → CFG is *structured*
- CDG is *acyclic* → CFG is *acyclic*
- CDG is *cyclic* → CFG is *cyclic*

The *control conditions* of node Y is the set,

$$CC(Y) = \{(X,L) \mid Y \text{ is control dependent on } X \text{ with label } L\}$$

Two nodes, A and B , are said to be *identically control dependent* if and only if they have the same set of control conditions i.e. $CC(A) = CC(B)$

© Krishna Palem, 1998 Please do not distribute 39

Data Dependence Analysis

If two operations have potentially interfering data accesses, data dependence analysis is necessary for determining whether or not an interference actually exists. If there is no interference, it may be possible to reorder the operations or execute them concurrently.

The data accesses examined for data dependence analysis may arise from array variables, scalar variables, procedure parameters, pointer dereferences, etc. in the original source program.

Data dependence analysis is conservative, in that it may state that a data dependence exists between two statements, when actually none exists.

© Krishna Palem, 1998 Please do not distribute 40

Data Dependence: Definition

A *data dependence*, $S_1 \rightarrow S_2$, exists between CFG nodes S_1 and S_2 with respect to variable X if and only if

1. there exists a path $P: S_1 \rightarrow S_2$ in CFG, with no intervening write to X , and
2. at least one of the following is true:
 - (a) (**flow**) X is written by S_1 and later read by S_2 , or
 - (b) (**anti**) X is read by S_1 and later is written by S_2 or
 - (c) (**output**) X is written by S_1 and later written by S_2

© Krishna Palem, 1998 Please do not distribute 41

Def/Use chaining for Data Dependence Analysis

A *def-use chain* links a definition D (i.e. a write access of variable X) to each use U (i.e. a read access), such that there is a path from D to U in CFG that does not redefine X .

Similarly, a *use-def chain* links a use U to a definition D , and a *def-def chain* links a definition D to a definition D' (with no intervening write to X in all cases).

Def-use, use-def, and def-def chains can be computed by data flow analysis, and provide a simple but conservative way of enumerating flow, anti, and output data dependences.

© Krishna Palem, 1998 Please do not distribute 42

Static single assignment (SSA) form

- Static single assignment (SSA) form provides a more efficient data structure for enumerating def-use, use-def and def-def chains.
- SSA form requires that *each use* be *reached* by a *single def* (when representing def-use information; analogous requirements are enforced for representing use-def and def-def information). Each def is treated as a new "name" for the variable.
- Each variable is assumed to have a dummy definition at the *START* node of the CFG.
- A ϕ function is used to capture the merge of multiple reaching definitions

© Krishna Palem, 1998

Please do not distribute

43

Dealing with Merge Points

```
If Cond
Then X ← 4
Else X ← 6
      ⋮
      ⋮
```

Use variable X several times

- Tricky situation since both defs can reach all subsequent uses; exact reaching def depends on whether Cond evaluated to true or not
- Keeping track of true and false cases separately is complicated and intractable (in the presence of nested conditionals)

© Krishna Palem, 1998

Please do not distribute

44

The SSA approach

```
If Cond
Then X1 ← 4
Else X2 ← 6
X3 ←  $\phi$ (X1, X2) ← Add this line
      ⋮
      ⋮
Use variable X3 several times
```

- The SSA solution is to add a special ϕ function at each merge point
- The new ϕ -def X₃ captures the merge of X₁ and X₂

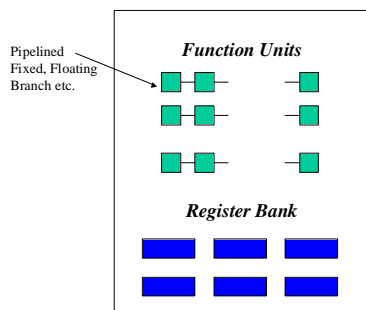
© Krishna Palem, 1998

Please do not distribute

45

Instruction Scheduling

Superscalar (RISC) Processors



© Krishna Palem, 1998

Please do not distribute

47

Canonical Instruction Set

- Register — Register Instructions (Single cycle).
- Special instructions for Load and Store to/from memory (multiple cycles).

A few notable exceptions, of course.

E.g.: Dec Alpha, HP PA-RISC, IBM Power & RS6K, Sun Sparc ...

© Krishna Palem, 1998

Please do not distribute

48

Opportunity in Superscalars

- High degree of *Instruction Level Parallelism (ILP)* via multiple (possibly) *pipelined functional units (FUs)*.

Essential to harness promised performance.

- Clean simple model and Instruction Set makes *compile-time* optimizations feasible.
- Therefore, performance advantages can be harnessed automatically

© Krishna Palem, 1998

Please do not distribute

49

Example of Instruction Level Parallelism

Processor components:

- 5 functional units: 2 fixed point units, 2 floating point units and 1 branch unit.
- Pipeline depth: floating point unit is 2 deep, and the others are 1 deep.

Peak rates: 7 instructions being processed simultaneously in each cycle

© Krishna Palem, 1998

Please do not distribute

50

Instruction Scheduling: The Optimization Goal

Given a source program P, schedule the instructions so as to minimize the overall execution time on the functional units in the target machine.

© Krishna Palem, 1998

Please do not distribute

51

Cost Functions

- Effectiveness of the Optimizations: *How well can we optimize our objective function?*
Impact on running time of the *compiled* code determined by the completion time.
- Efficiency of the Optimizations: *How fast can we optimize?*
Impact on the time it takes to compile or cost for gaining the benefit of code with fast running time.

© Krishna Palem, 1998

Please do not distribute

52

Instruction Scheduling Algorithms

Impact of Control Flow

Acyclic control flow is easier to deal with than *cyclic* control flow. Problems in dealing with cyclic flow:

- A loop *implicitly* represents a large run-time program space compactly.
- Not possible to open out the loops fully at compile-time.
- Loop unrolling provides a partial solution.

more...

© Krishna Palem, 1998

Please do not distribute

54

Impact of Control Flow (Contd.)

- Using the loop to optimize its dynamic behavior is a challenging problem.
- Hard to optimize well without detailed knowledge of the range of the iteration.
- In practice, profiling can offer limited help in estimating loop bounds.

Acyclic Instruction Scheduling

- We will consider the case of acyclic control flow first.
- The acyclic case itself has two parts:
 - The simpler case that we will consider first has no branching and corresponds to *basic block* of code, e.g. loop bodies.
 - The more complicated case of scheduling programs with acyclic control flow *with* branching will be considered next.

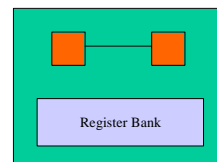
The Core Case: Scheduling Basic Blocks

Why are basic blocks easy?

- All instructions specified as part of the input must be executed.
- Allows *deterministic* modeling of the input.
- No "branch probabilities" to contend with; makes problem space easy to optimize using classical methods.

Early RISC Processors

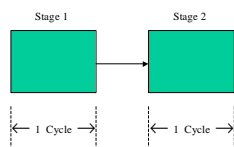
Single FU with two stage pipeline:



Programmer's logical view of: *Berkeley RISC, IBM801, MIPS*

Instruction Execution Timing

The 2-stage pipeline of the Functional Unit



- The first stage performs *Fetch/Decode/Execute* for register-register operations (single cycle) and *fetch/decode/initiate* for Loads and Stores from memory (two cycles).

more...

Instruction Execution Timing (Contd.)

- The second cycle is the memory latency to fetch/store the operand from/to memory.

In reality, memory is cache and extra latencies result if there is a cache miss.

Parallelism Comes From the Following Fact

While a load/store instruction is executing at the second pipeline stage, a new instruction can be initiated at the first stage.

© Krishna Palem, 1998

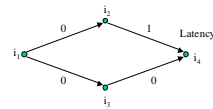
Please do not distribute

61

Instruction Scheduling

For previous example of RISC processors,

Input: A basic block represented as a DAG



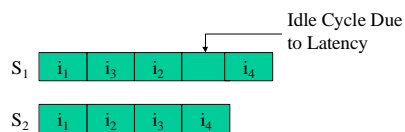
- i_2 is a load instruction.
- Latency of 1 on (i_2, i_4) means that i_4 cannot start for one cycle after i_2 completes.

© Krishna Palem, 1998

Please do not distribute

62

Instruction Scheduling (Contd.)



Two schedules for the above DAG with S_2 as the desired sequence.

© Krishna Palem, 1998

Please do not distribute

63

The General Instruction Scheduling Problem

Input: DAG representing each basic block where:

1. Nodes encode *unit execution time* (single cycle) instructions.
2. Each node requires a definite class of FUs.
3. Additional pipeline delays encoded as latencies on the edges.
4. Number of FUs of each type in the target machine.

more...

© Krishna Palem, 1998

Please do not distribute

64

The General Instruction Scheduling Problem (Contd.)

Feasible Schedule: A specification of a *start time* for each instruction such that the following constraints are obeyed:

1. Resource: Number of instructions of a given type of any time < corresponding number of FUs.
2. Precedence and Latency: For each predecessor j of an instruction i in the DAG, i is started only δ cycles after j finishes where δ is the latency labeling the edge (j,i) ,

Output: A schedule with the minimum *overall completion time (makespan)*.

© Krishna Palem, 1998

Please do not distribute

65

Drawing on Deterministic Scheduling

Canonical Algorithm:

1. Assign a *Rank* (priority) to each instruction (or node).
2. Sort and build a priority *list* \mathcal{L} of the instructions in non-decreasing order of Rank.

Nodes with smaller ranks occur earlier in this list.

© Krishna Palem, 1998

Please do not distribute

66

Drawing on Deterministic Scheduling (Contd.)

3. Greedily list-schedule \mathcal{L} .

Scan \mathcal{L} iteratively, and on each scan choose the largest number of “ready” instructions subject to resource (FU) constraints in list-order.

An instruction is ready provided it has not been chosen earlier and all of its predecessors have been chosen and the appropriate latencies have elapsed.

more...

© Krishna Palem, 1998 Please do not distribute 67

The Value of Greedy List Scheduling

Example: Consider the DAG shown below:

Using the list $\mathcal{L} = \langle i_1, i_2, i_3, i_4, i_5 \rangle$

- Greedy scanning produces the steps of the schedule as follows:

more...

© Krishna Palem, 1998 Please do not distribute 68

The Value of Greedy List Scheduling (Contd.)

- On the first scan: i_1 is selected which is the first step of the schedule.
- On the second and third scans and out of the list order, respectively i_4 and i_5 are selected to correspond to steps two and three of the schedule.
- On the fourth and fifth scans, i_2 and i_3 respectively are selected for steps four and five.

© Krishna Palem, 1998 Please do not distribute 69

Some Intuition

- Greediness helps in making sure that idle cycles don't remain if there are available instructions further “downstream.”
- Ranks help prioritize nodes such that choices made early on favor instructions with greater enabling power, so that there is no unforced idle cycle.

© Krishna Palem, 1998 Please do not distribute 70

How Good is Greedy?

Approximation: For any pipeline depth $k \geq 1$ and number m of pipelines,

$$S_{\text{greedy}}/S_{\text{opt}} \leq \left(2 - \frac{1}{mk}\right)$$

- For example, with one pipeline ($m = 1$) and the latencies k grow as 2,3,4,..., the approximate schedule is guaranteed to have a completion time no more 66%, 75%, and 80% over the optimal completion time.
- This theoretical guarantee shows that greedy scheduling is not bad, but the bounds are worst-case; practical experience tends to be much better.

more...

© Krishna Palem, 1998 Please do not distribute 71

How Good is Greedy? (Contd.)

Running Time of Greedy List Scheduling: Linear in the size of the DAG.

“Scheduling Time-Critical Instructions on RISC Machines,” K. Palem and B. Simons, *ACM Transactions on Programming Languages and Systems*, 632-658, Vol. 15, 1993.

© Krishna Palem, 1998 Please do not distribute 72

A Critical Choice: The Rank Function for Prioritizing Nodes

Rank Functions

1. "Postpass Code Optimization of Pipelined Constraints", J. Hennessey and T. Gross, *ACM Transactions on Programming Languages and Systems*, vol. 5, 422-448, 1983.
2. "Scheduling Expressions on a Pipelined Processor with a Maximal Delay of One Cycle," D. Bernstein and I. Gertner, *ACM Transactions on Programming Languages and Systems*, vol. 11 no. 1, 57-66, Jan 1989.

© Krishna Palem, 1998

Please do not distribute

74

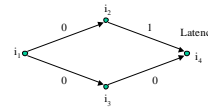
Rank Functions (Contd.)

3. "Scheduling Time-Critical Instructions on RISC Machines," K. Palem and B. Simons, *ACM Transactions on Programming Languages and Systems*, 632-658, vol. 15, 1993

Optimality: 2 and 3 produce optimal schedules for RISC processors such as the IBM 801, Berkeley RISC and so on.

An Example Rank Function

The example DAG



1. Initially label all the nodes by the same value, say α
2. Compute new labels from old starting with nodes at level zero (i_4) and working towards higher levels:
 - (a) All nodes at level zero get a rank of α .

more...

© Krishna Palem, 1998

Please do not distribute

75

© Krishna Palem, 1998

Please do not distribute

76

An Example Rank Function (Contd.)

- (b) For a node at level 1, construct a new label which is the concentration of all its successors connected by a latency 1 edge.

Edge i_2 to i_4 in this case.

- (c) The empty symbol \emptyset is associated with latency zero edges.

Edge i_3 to i_4 for example.

more...

An Example Rank Function (Contd.)

- (d) The result is that i_2 and i_3 respectively get new labels and hence ranks $\alpha' = \alpha > \alpha'' = \emptyset$.

Note that $\alpha' = \alpha > \alpha'' = \emptyset$ i.e., labels are drawn from a totally ordered alphabet.

- (e) Rank of i_1 is the concentration of the ranks of its immediate successors i_2 and i_3 i.e., it is $\alpha''' = \alpha' | \alpha''$.

3. The resulting sorted list is (optimum) i_1, i_2, i_3, i_4 .

© Krishna Palem, 1998

Please do not distribute

77

© Krishna Palem, 1998

Please do not distribute

78

The More General Case Scheduling Acyclic Control Flow Graphs

Significant Jump in Compilation Cost

What is the problem when compared to basic-blocks?

- Conditional and unconditional branching is permitted.
- The problem being optimized is no longer deterministically and completely known at compile-time.
- Depending on the sequence of branches taken, the problem structure of the graph being executed can vary.
- Impractical to optimize all possible combinations of branches and have a schedule for each case, since a sequence of k branches can lead to 2^k possibilities — a *combinatorial explosion* in cost of compiling.

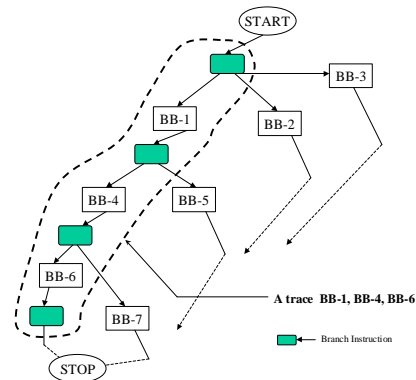
© Krishna Palem, 1998

Please do not distribute

80

Containing Compilation Cost

A well known classical approach is to consider *traces* through the (acyclic) *control flow graph*. An example is presented in the next slide.



© Krishna Palem, 1998

Please do not distribute

81

© Krishna Palem, 1998

Please do not distribute

82

Traces

"Trace Scheduling: A Technique for Global Microcode Compaction," J.A. Fisher, *IEEE Transactions on Computers*, Vol. C-30, 1981.

Main Ideas:

- Choose a program segment that has no cyclic dependences.
- Choose *one* of the paths out of each branch that is encountered.

more...

© Krishna Palem, 1998

Please do not distribute

83

Traces (Contd.)

- Use statistical knowledge based on (estimated) program behavior to bias the choices to favor the more frequently taken branches.
- This information is gained through profiling the program or via static analysis.
- The resulting sequence of basic blocks including the branch instructions is referred to as a trace.

© Krishna Palem, 1998

Please do not distribute

84

Trace Scheduling

High Level Algorithm:

1. Choose a (maximal) segment s of the program with acyclic control flow.

The instructions in s have associated "frequencies" derived via statistical knowledge of the program's behavior.

2. Construct a trace τ through s :
 - (a) Start with the instruction in s , say i , with the highest frequency.

more...

Trace Scheduling (Contd.)

- (b) Grow a path out from instruction i in both directions, choosing the path to the instruction with the higher frequency whenever there is a choice

Frequencies can be viewed as a way of prioritizing the path to choose and subsequently optimize.

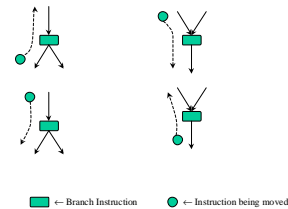
3. Rank the instructions in τ using a rank function of choice.
4. Sort and construct a list \mathcal{L} of the instructions using the ranks as priorities.
5. Greedily list schedule and produce a schedule using the list \mathcal{L} as the priority list.

Significant Comments

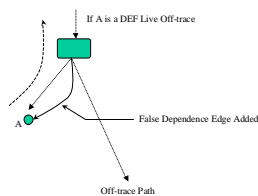
- We pretend as if the trace is always taken and executed and hence schedule it in steps 3-5 using the same framework as for a basic-block.
- The important difference is that conditionals branches are there on the path, and moving code past these conditionals can lead to side-effects.
- These side effects are not a problem in the case of basic-blocks since there, every instruction is executed all the time.
- This is not true in the present more general case when an outgoing or incoming off-trace branch is taken however infrequently: we will study these issues next.

The Four Elementary but Significant Side-effects

Consider a single instruction moving past a conditional branch:



The First Case



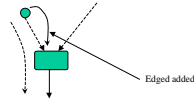
- This code movement leads to the instruction executing sometimes when the instruction ought not to have: *speculatively*.

more...

The First Case (Contd.)

- If A is a *write* of the form $\# = \dots$, then the variable (virtual register) $\#$ must not be live on the off-trace path.
- In this case, an additional pseudo edge is added from the branch instruction to instruction A to prevent this motion.

The Second Case



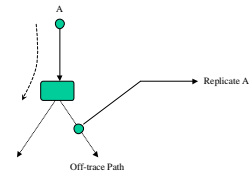
- Identical to previous case except the pseudo-dependence edge is *from A to the join instruction* whenever A is a "write" or a *def*.
 - A more general solution is to permit the code motion but undo the effect of the speculated definition by adding repair code
- An expensive proposition in terms of compilation cost.

© Krishna Palem, 1998

Please do not distribute

91

The Third Case



- Instruction A will *not* be executed if the off-trace path is taken.
- To avoid mistakes, it is *replicated*.

more...

© Krishna Palem, 1998

Please do not distribute

92

The Third Case (Contd.)

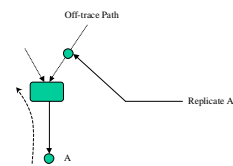
- This is true in the case of read and write instructions.
- Replication causes A to be executed independent of the path being taken to preserve the original semantics.
- If (non-)liveness information is available, replication can be done more conservatively.

© Krishna Palem, 1998

Please do not distribute

93

The Fourth Case



- Similar to Case 3 except for the direction of the replication as shown in the figure above.

© Krishna Palem, 1998

Please do not distribute

94

At a Conceptual Level: Two Situations

- **Speculations:** Code that is executed "sometimes" when a branch is executed is now executed "always" due to code motion as in Cases 1 and 2.
 - *Legal* speculations wherein data-dependences are not violated.
 - *Safe* speculation wherein control-dependences on exceptions-causing instructions are not violated.

more...

© Krishna Palem, 1998

Please do not distribute

95

At a Conceptual Level: Two Situations (Contd.)

- *Unsafe speculation* where there is no restriction and hence exceptions can occur.

This type of speculation is currently playing a role in "production quality" compilers.

- **Replication:** Code that is "always" executed is duplicated as in Cases 3 and 4.

© Krishna Palem, 1998

Please do not distribute

96

Comparison to Basic Block Scheduling

- Instruction scheduler needs to handle speculation and replication.
- Otherwise the framework and strategy is identical.

Fisher's Trace Scheduling Algorithm

Description:

1. Choose a (maximal) region s of the program that has acyclic control flow.
 2. Construct a trace τ through s .
 3. Add additional dependence edges to the DAG to limit speculative execution.
- Note that this is Fisher's solution.

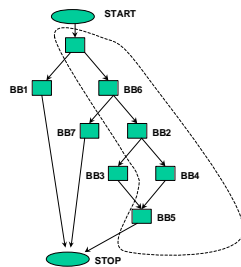
more...

Fisher's Trace Scheduling Algorithm (Contd.)

4. Rank the instructions in τ using a rank function of choice.
5. Sort and construct a list \mathcal{L} of the instructions using the ranks as priorities.
6. Greedily list schedule and produce a schedule using the list \mathcal{L} as the priority list.
7. Add replicated code whenever necessary on all the off-trace paths.

A Detailed Example will be Discussed Now

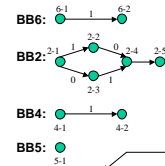
Example



BBi Basic-block

Example (Contd.)

TRACE: BB6, BB2, BB4, BB5



Concentration of Local Schedules

Feasible Schedule: 6-1 X 6-2 2-1 X 2-2 2-3 X 2-4 2-5 4-1 X 4-2 5-1
 Global Improvements 6-1 2-2 6-2 2-2 2-3 X 2-4 2-5 4-1 X 4-2 5-1;
 6-1 2-1 6-2 2-3 2-2 2-4 2-5 4-1 X 4-2 5-1
 6-1 2-1 6-2 2-3 2-2 2-4 2-5 4-1 5-1 4-2

X: Denotes Idle Cycle

Obvious advantages of global code motion are that the idle cycles have disappeared.

Limitations of This Approach

- Optimizations depends on the traces being the dominant paths in the program's control-flow
- Therefore, the following two things should be true:
 - Programs should demonstrate the behavior of being skewed in the branches taken at run-time, for typical mixes of input data.
 - We should have access to this information at compile time. Not so easy.

A More Aggressive Solution

"Global Instruction Scheduling for Superscalar Machines," D. Bernstein and M. Rodeh *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 241-255, 1991.

- Schedule an entire acyclic region at once. Innermost regions are scheduled first.
- Use the forward control dependence graph to determine the "degree of speculativeness" of instruction movements.
- Use generalization of single basic block list scheduling include multiple basic blocks.

Detecting Speculation and Replication Structurally

- Need tests that can be performed quickly to determine which of the side-effects have to be addressed after code-motion.
- Preferably based on structured information that can be derived from previously computed (and explained) program analysis.
- Decisions that are based on the Control (sub) Component of the Program Dependence Graph (PDG).
- Details can be found in Bernstein and Rodeh's work

Scheduling Control Flow Graphs with Loops (Cycles)

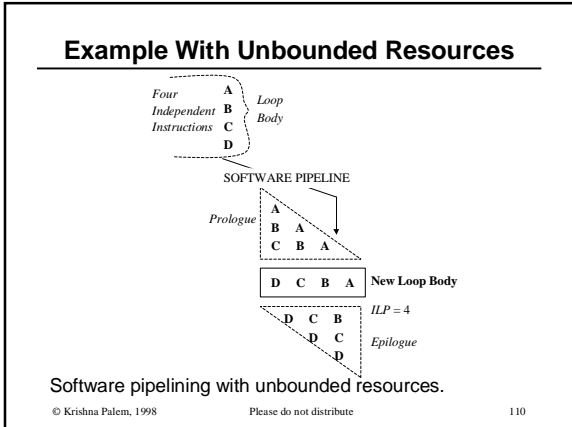
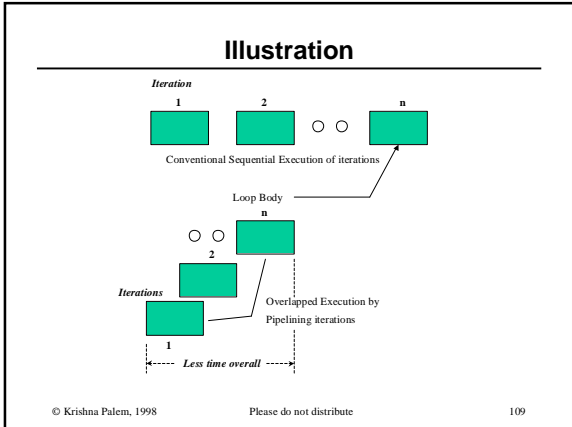
Main Idea

- Loops are treated as integral units.
- Conventionally, loop-body is executed sequentially from one iteration to the next.
- By compile-time analysis, execution of successive iterations of a loop is overlapped.
- Reminiscent of execution in hardware pipelines.

more...

Main Idea (Contd.)

- Overall *completion time* can be much less if there are computational resources in the target machine to support this overlapped execution.
- Works with no underlying hardware support such as interlocks etc.

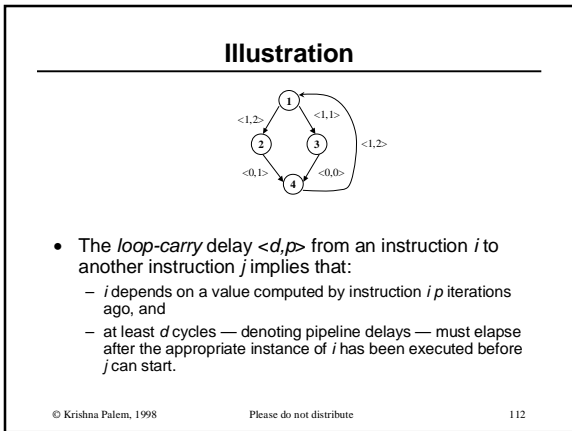


Constraints on The Compiler in Determining Schedule

Since there are no expectations on hardware support at run-time:

- The overlapped execution on each cycle must be possible with the degree of instruction level parallelism in terms of functional-units
- The inter-instruction latencies must be obeyed within each iteration but *more importantly* across iterations as well.
- These inter-iteration dependencies and consequent latencies are loop-carried dependencies.

© Krishna Palem, 1998 Please do not distribute 111



Modulo Scheduling

- Find a steady-state schedule for the kernel
- The length of this schedule is the *initiation interval (II)*
- The *same* schedule is executed in every iteration
- Primary goal is to minimize the initiation interval
- Prologue and epilogue are recovered from the kernel

© Krishna Palem, 1998 Please do not distribute 113

Minimal Initiation Interval (MII)

- $delay(c)$ -- total latency in data dependence cycle c
- $distance(c)$ -- iteration distance of cycle c
- $uses(r)$ -- number of occurrence of resource r in one iteration
- $units(r)$ -- number of functional units of type r

© Krishna Palem, 1998 Please do not distribute 114

Minimal Initiation Interval (MII)

- Recurrence constrained minimal initiation interval
- "Longest cycle is the bottleneck"
- $RecMII = \max_{c \in \text{cycles}} \text{delay}(c) / \text{distance}$
- Resource constrained minimal initiation interval
- "Most critical resource is the bottleneck"
- $ResMII = \max_{r \in \text{resources}} \text{uses}(r) / \text{units}(r)$
- Minimal initiation interval
- $MII = \max(RecMII, ResMII)$

© Krishna Palem, 1998

Please do not distribute

115

Iterated Modulo Scheduling

- Rau 1994
- Uses operation list scheduling as building block
- Uses some backtracking

© Krishna Palem, 1998

Please do not distribute

116

Preprocessing Steps

- Loop unrolling
- Modulo variable expansion
- Loops with internal control flow: remove with if-conversion
- Reverse if-conversion

© Krishna Palem, 1998

Please do not distribute

117

Main Driver

- `budget_ratio` is the amount of backtracking to perform before trying a larger II

```
procedure modulo_schedule(budget_ratio)
  compute MCC;
  CC = MCC;
  budget = budget_ratio * number of operations;
  while schedule is not found do
    iterate_schedule(CC, budget);
  CC = CC + 1;
```

© Krishna Palem, 1998

Please do not distribute

118

Iterative Schedule Routine

```
procedure iterate_schedule(CC, budget);
  compute height based priorities;
  while there are unscheduled operations and
    budget > 0 do
    op = the operation with highest priority;
    min = earliest start time for op;
    max = min + CC - 1;
    t = find_slot(op, min, max);
    schedule op at time t
    and unschedule all crossflowly scheduled
    predecessors and successors of op;
    budget = budget - 1;
```

© Krishna Palem, 1998

Please do not distribute

119

Discussion

- Instructions are either scheduled or unscheduled
- Scheduled instructions may be unscheduled subsequently
- Given an instruction j, the earliest start time of j is limited by all its scheduled predecessors k
- $time(j) \geq time(k) + latency(k,j) - II * distance(k,j)$
- Note that focus is only on data dependence constraints

© Krishna Palem, 1998

Please do not distribute

120

Find Slot Routine

```

procedure find_slot(op, min, max);
  for t = min to max do
    if op has no resource conflict at t
      return t;
  if op has never been scheduled or
  min > previous scheduled time of op
    return min;
  else
    return t + previous scheduled time of op;
  
```

© Krishna Palem, 1998

Please do not distribute

121

Discussion of find_slot

- Finds the earliest time between *min* and *max* such that *op* can be scheduled without resource conflicts
- If no such time slot exists then
 - if *op* hasn't been unscheduled before (and it's not scheduled now), chose *min*
 - if *op* has been scheduled before, choose the previous scheduled time + 1 or *min*, whichever is later
- Note that the latter choice implies that some instructions will have to be unscheduled

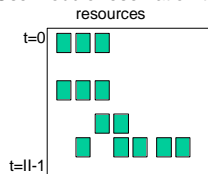
© Krishna Palem, 1998

Please do not distribute

122

Keeping track of resources

- Use modulo reservation table (MRT)



- Can also be encoded as finite state automaton

© Krishna Palem, 1998

Please do not distribute

123

Computing Priorities

- Based on the Critical Path Heuristic
- $H(i)$ -- the height-based priority of instruction i
- $H(i) = 0$ if i has no successors
- $H(i) = \max_{k \in \text{succ}(i)} H(k) + \text{latency}(i, k) - II * \text{distance}(i, k)$

© Krishna Palem, 1998

Please do not distribute

124

Algorithms for Software Pipelining

1. "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing," B. Rau and C. Glaeser, *Proc. Fourteenth Annual Workshop on Microprogramming*, 183-198, 1981.
2. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," M. Lam, *Proc. 1988 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 318-328, 1988.

© Krishna Palem, 1998

Please do not distribute

125

Algorithms for Software Pipelining (Contd.)

3. "Iterative modulo scheduling: An algorithm for software pipelining loops" B. Rau, *Proceedings of the 27th Annual Symposium on Microarchitecture*, December 1994

© Krishna Palem, 1998

Please do not distribute

126

Additional Reading:

1. "Perfect Pipelining: A New Loop Parallelization Technique", A. Aiken and A. Nicolau, *Proceedings of the 1988 European Symposium on Programming*, Springer Verlag Lecture Notes in Computer Science, No. 300, 1988.
2. "Scheduling and Mapping: Software Pipelining in the Presence of Structural Hazards," E. Altman, R. Govindarajan and Guang R. Gao, *Proc. 1995 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, SIGPLAN Notice 30(6), 139-150, 1995.

© Krishna Palem, 1998

Please do not distribute

127

Additional Reading (Contd.):

3. "All Shortest Routes from a Fixed Origin in a Graph", G. Dantzig, W. Blattner and M. Rao, *Proceedings of the Conference on Theory of Graphs*, 85-90, July 1967.
4. "A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture", K. Ebcioglu and T. Nakanati, *Workshop on Languages and Compilers for Parallel Computing*, 1989.

© Krishna Palem, 1998

Please do not distribute

128

Additional Reading (Contd.):

5. "A global resource-constrained parallelization technique", K. Ebcioglu and Alexandru Nicolau, *Proceedings SIGPLAN-89 Conference on Programming Language Design and Implementation*, 154-163, 1989.
6. "The Program Dependence Graph and its use in optimization," J. Ferrante, K.J. Ottenstein and J.D. Warren, *ACM TOPLAS*, vol. 9, no. 3, 319-349, Jul. 1987.
7. "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", J. Fisher, *IEEE Computer*, vol.7, 45-53, 1984.

© Krishna Palem, 1998

Please do not distribute

129

Additional Reading:

8. "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Habb, J. Holm and D. Lavery, *Journal of Supercomputing*, 7(1,2), March 1993.
9. "Circular Scheduling: A New Technique to Performing", S. Jian, *Proceedings SIGPLAN-91 Conference on Programming Language Design and Implementation*, 219-228, 1991.

© Krishna Palem, 1998

Please do not distribute

130

Additional Reading:

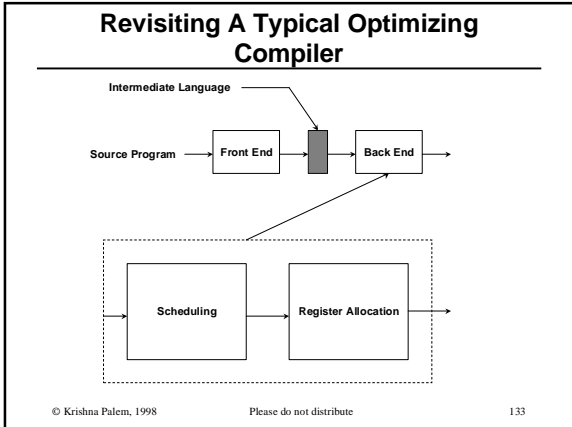
- 10 "Data Flow and Dependence Analysis for Instruction Level Parallelism", B. Rau, *Proceedings of the Fourth Workshop on Language and Compilers for Parallel Computing*, August 1991.
- 11 "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High-Performance Scientific Computing", B. Rau and C. Glaeser, *Proceedings of the 14th Annual Workshop on Microprogramming*, 183-198, 1981.

© Krishna Palem, 1998

Please do not distribute

131

Register Allocation



Rationale for Separating Register Allocation from Scheduling

- Each of Scheduling and Register Allocation are hard to solve individually, let alone solve *globally* as a combined optimization.
- So, solve each optimization locally and heuristically “patch up” the two stages.

© Krishna Palem, 1998 Please do not distribute 134

Why Register Allocation?

- Storing and accessing variables from registers is much faster than accessing data from memory. The way operations are performed in load/store (RISC) processors.
- Therefore, in the interests of performance—if not by necessity — variables ought to be stored in registers.
- For performance reasons, it is useful to store variables as long as possible, once they are loaded into registers.
- Registers are bounded in number (say 32.)
- Therefore, “register-sharing” is needed over time.

© Krishna Palem, 1998 Please do not distribute 135

The Goal

- *Primarily* to assign registers to variables.
- However, the allocator runs out of registers quite often.
- Decide which variables to “flush” out of registers to free them up, so that other variables can be bought in. This important indirect consequence of allocation is referred to as *spilling*.

© Krishna Palem, 1998 Please do not distribute 136

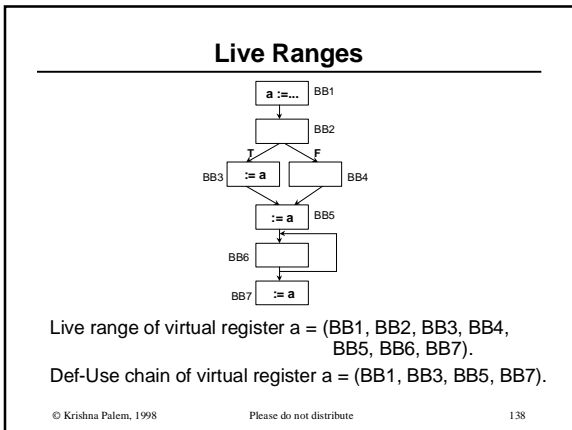
Register Allocation and Assignment

Allocation: identifying program values (virtual registers, live ranges) and program points at which values should be stored in a physical register.

Program values that are not allocated to registers are said to be *spilled*.

Assignment: identifying which physical register should hold an allocated value at each program point.

© Krishna Palem, 1998 Please do not distribute 137



Global Register Allocation

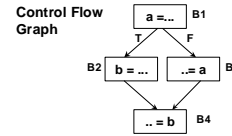
- Local register allocation does not store data in registers across basic blocks.
Local allocation has poor register utilization \Rightarrow global register allocation is essential.
 - Simple global register allocation: allocate most "active" values in each inner loop.
 - Full global register allocation: identify live ranges in control flow graph, allocate live ranges, and split ranges as needed.
- Goal:** select allocation so as to minimize number of load/store instructions performed by optimized program.

© Krishna Palem, 1998

Please do not distribute

139

Simple Example of Global Register Allocation



Live range of $a = \{B_1, B_3\}$

Live range of $b = \{B_2, B_4\}$

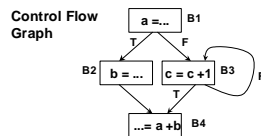
No interference! a and b can be assigned to the same register

© Krishna Palem, 1998

Please do not distribute

140

Another Example of Global Register Allocation



Live range of $a = \{B_1, B_2, B_3, B_4\}$

Live range of $b = \{B_2, B_4\}$

Live range of $c = \{B_3\}$

In this example, a and c interfere, and c should be given priority because it has a higher usage count.

© Krishna Palem, 1998

Please do not distribute

141

Cost and Savings

Compilation Cost: running time and space of the global allocation algorithm.

Execution Savings: cycles saved due to register residence of variables in optimized program execution.

Contrast with memory-residence which leads to longer execution times.

© Krishna Palem, 1998

Please do not distribute

142

Interference Graph

Definition: An *interference graph* G is an undirected graph with the following properties:

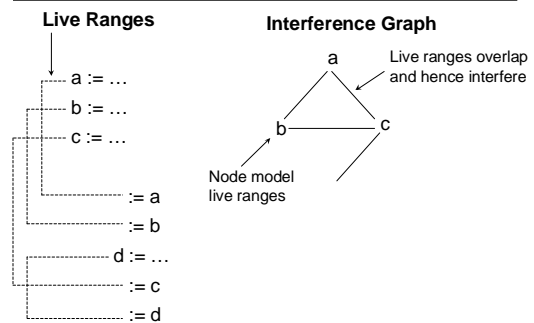
- each node x denotes exactly one distinct live range X , and
- an edge exists between nodes x and y iff $X \cap Y \neq \emptyset$, where X and Y are the live ranges corresponding to nodes x and y .

© Krishna Palem, 1998

Please do not distribute

143

Interference Graph Example



© Krishna Palem, 1998

Please do not distribute

144

The Classical Approach

“Register Allocation and Spilling via Graph Coloring”, G. Chaitin, *Proceedings SIGPLAN-82 Symposium on Compiler Construction*, 98-105, 1982.

“Register Allocation via Coloring”, G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins and P. Markstein, *Computer Languages*, vol. 6, 47-57, 1981.

more...

The Classical Approach (Contd.)

- These works introduced the key notion of an *interference graph* for encoding conflicts between the live ranges.
- This notion was defined for the *global* control flow graph.
- It also introduced the notion of *graph coloring* to model the idea of register allocation.

Execution Time and Spill-cost

Spilling: Moving a variable that is currently register resident to memory when no more registers are available, and a new live-range needs to be allocated one spill.

Minimizing Execution Cost: Given an optimistic assignment— i.e., one where all the variables are register-resident, *minimizing* spilling.

Graph Coloring

- Given an undirected graph G and a set of k distinct colors, compute a *coloring* of the nodes of the graph i.e., assign a color to each node such that *no two adjacent* nodes get the same color.

Recall that two nodes are adjacent iff they have an edge between them.

- A given graph might *not* be k -colorable.
- In general, it is a computationally hard problem to color a given graph using a given number k of colors.
- The register allocation problem uses good heuristics for coloring.

Register Allocation as Coloring

- Given k registers, interpret each register as a color.
- The graph G is the interference graph of the given program.
- The nodes of the interference graph are the executable live ranges on the target platform.
- A coloring of the interference graph is an assignment of registers (colors) to live ranges (nodes).
- Running out of colors implies not enough registers and hence a need to spill in the above model.

The Approach Discussed Here

“The Priority Based Coloring Approach to Register Allocation”, F. Chow and J. Hennessey, *ACM Transactions on Programming Languages and Systems*, vol. 12, 501-536, 1990.

Important Modeling Difference

- The first difference from the classical approach is that now, we assume that the “home location” of a live range is in memory.
 - Conceptually, values are always in memory unless promoted to a register; this is also referred to as the *pessimistic* approach.
 - In the classical approach, the dual of this model is used where values are *always in registers* except when spilled; recall that this is referred to as the optimistic approach.

more...

Important Modeling Difference (Contd.)

- A second major difference is the *granularity* at which code is modeled.
 - In the classical approach, individual instructions are modeled whereas
 - Now, basic blocks are the primitive units modeled as nodes in live ranges and the interference graph.
- The final major difference is the place of the register allocation in the overall compilation process.
 - In the present approach, the interference graph is considered *earlier* in the compilation process using intermediate level statements; compiler generated temporaries are known.
 - In contrast, in the previous work the allocation is done at the level of the machine code.

Computing Live Ranges

Using data flow analysis, we compute for each basic block:

- In the forward direction, the *reaching* attribute.

A variable is reaching block *i* if a definition or use of the variable reaches the basic block along the edges of the CFG.

- In the backward direction, the *liveness* attribute.

A variable is live at block *i* if there is a direct reference to the variable at block *i* or at some block *j* that succeeds *i* in the CFG, provided the variable in question is *not redefined* in the interval between *i* and *j*.

Computing Live Ranges (Contd.)

The live range of a variable is the intersection of basic-blocks in CFG nodes in which the variable is live, and the set which it can reach.

The Main Information to be Used by the Register

- For each live range, we have a bit vector *LIVE* of the basic blocks in it.
- Also we have *INTERFERE* which gives for the live range, the set of all other live ranges that interfere with it.
- Recall that two live ranges interfere if they intersect in at least one (basic-block).
- If $|INTERFERE|$ is smaller than the number of available registers for a node *i*, then *i* is *unconstrained*; it is constrained otherwise.

more...

The Main Information to be Used by the Register (Contd.)

- An unconstrained node can be safely assigned a register since conflicting live ranges do not use up the available registers.
- We associate a (possibly empty) set *FORBIDDEN* with each live range that represents the set of colors that have already been assigned to the members of its *INTERFERENCE* set.

The above representation is essentially a detailed interference graph representation.

Prioritizing Live Ranges

In the memory bound approach, given live ranges with a choice of assigning registers, we do the following:

- Choose a live range that is “likely” to yield greater savings in execution time.
- This means that we need to estimate the savings of each basic block in a live range.

Estimate the Savings

Given a live range X for variable x , the estimated savings in a basic block i is determined as follows:

1. First compute $CyclesSaved$ which is the number of loads and stores of x in i scaled by the number of cycles taken for each load/store.
2. Compensate the single load and/or store that might be needed to bring the variable in and/or store the variable at the end and denote it by $Setup$.

Note that $Setup$ is derived from a single load or store or a load plus a store.

more...

Estimate the Savings (Contd.)

$$3. Savings(X, i) = \{CyclesSaved - Setup\}$$

These indicate the actual savings in cycles after accounting for the possible loads/stores needed to move x at the beginning/end of i .

4. $TotalSavings(X) = \sum_{i \in X} Savings(X, i) \times W(i)$.
 - (a) x is the set of all basic blocks in the live range of X .
 - (b) $W(i)$ is the execution frequency of variable x in block i .

more...

Estimate the Savings (Contd.)

5. Note however that live regions might span a few blocks but yield a large savings due to frequent use of the variable while others might yield the same cumulative gain over a larger number of basic blocks. We prioritize the former case and define:

$$\{Priority(X) = TotalSavings(X) / Span(X)\}$$

where $Span(X)$ is the number of basic blocks in X .

The Algorithm

For all constrained live ranges, execute the following steps:

1. Compute $Priority(X)$ if it has not already been computed.
2. For the live range X with the highest priority:
 - (a) If its priority is negative or if no basic block i in X can be assigned a register — because every color has been assigned to a basic block that interferes with i — then delete X from the list and modify the interference graph.
 - (b) Else, assign it a color that is not in its forbidden set.
 - (c) Update the forbidden sets of the members of $INTERFERE$ for X .

more...

The Algorithm (Contd.)

3. For each live range X' that is in $INTERFERE$ for X do:
 - (a) If the $FORBIDDEN$ of X' is the set of all colors i.e., if no colors are available, $SPLIT(X')$. Procedure $SPLIT$ breaks a live range into smaller live ranges with the intent of reducing the interference of X' it will be described next.
4. Repeat the above steps till all constrained live ranges are colored or till there is no color left to color any basic block.

The Idea Behind Splitting

- Splitting ensures that we break a live range up into increasingly smaller live ranges.
- The limit is of course when we are down to the size of a single basic block.
- The intuition is that we start out with coarse-grained interference graphs with few nodes.
- This makes the interference node degree possibly high.
- We increase the problem size via splitting on a need-to-basis.
- This strategy lowers the interference.

© Krishna Palem, 1998

Please do not distribute

163

The Splitting Strategy

A sketch of an algorithm for splitting:

1. Choose a *split* point.
Note that we are guaranteed that X has at least one basic block i which can be assigned a color i.e., its forbidden set does not include all the colors. The earliest such in the order of control flow can be the split point.
2. Separate the live range X into X_1 and X_2 around the split point.
3. Update the sets *INTERFERE* for X_1 and X_2 and those for the live ranges that interfered with X

more...

© Krishna Palem, 1998

Please do not distribute

164

The Splitting Strategy (Contd.)

4. Recompute priorities and reprioritize the list.

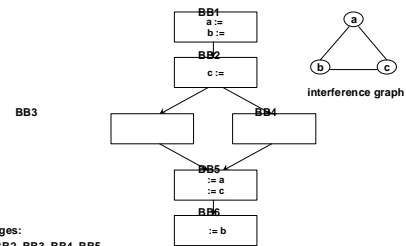
Other bookkeeping activities to realize a safe implementation are also executed.

© Krishna Palem, 1998

Please do not distribute

165

Live Range Splitting Example



Live Ranges:

a: BB1, BB2, BB3, BB4, BB5

b: BB1, BB2, BB3, BB4, BB5, BB6

c: BB2, BB3, BB4, BB5

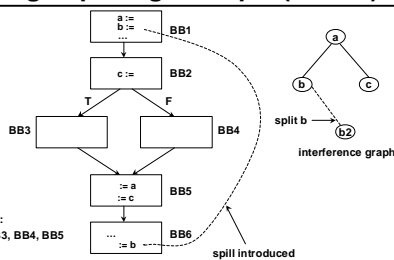
Assume the number of physical registers = 2

© Krishna Palem, 1998

Please do not distribute

166

Live Range Splitting Example (Contd.)



New live ranges:

a: BB1, BB2, BB3, BB4, BB5

b: BB1

c: BB2, BB3, BB4, BB5

b2: BB6

b and b2 are logically the same program variable

b2 is a renamed equivalent of b.

All nodes are now unconstrained.

© Krishna Palem, 1998

Please do not distribute

167

Interaction Between Allocation and Scheduling

- The allocator and the scheduler are typically patched together heuristically.
- Leads to the "phase ordering problem: *Should allocation be done before scheduling or vice-versa?*"
- Saving on spilling or "good allocation" is only indirectly connected to the actual execution time.
Contrast with instruction scheduling.
- Factoring in register allocation into scheduling and solving the problem "globally" is a research issue.

© Krishna Palem, 1998

Please do not distribute

168

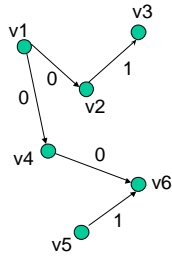
Example Basic Block

Source Code:

$z = x(i)$
 $temp = x(i+1+N)$

Intermediate Code:

v1: VR1 ← ADDR (X) + i
 v2: VR2 ← LOAD @(VR1)
 v3: z ← STORE VR2
 v4: VR4 ← VR1 + 1
 v5: VR5 ← LOAD N
 v6: VR6 ← LOAD @(VR4+VR5)

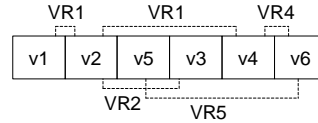


© Krishna Palem, 1998

Please do not distribute

169

Instruction Scheduling followed by Register Allocation



Completion time = 6 cycles.

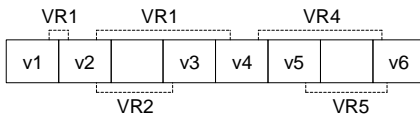
Maximum register width = 3.

© Krishna Palem, 1998

Please do not distribute

170

Register Allocation Followed by Instruction Scheduling



Completion time = 8 cycles.

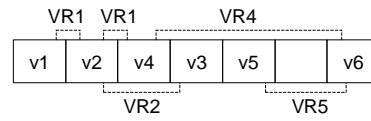
Maximum register width = 2.

© Krishna Palem, 1998

Please do not distribute

171

Combined Register Allocation and Instruction Scheduling



Completion time = 7 cycles.

Maximum register width = 2.

© Krishna Palem, 1998

Please do not distribute

172