INTRODUCTION TO CODE GENERATION

Instruction selection

Register allocation

Register allocation by graph coloring

Register allocation at link time

Instructionscheduling

Listscheduling

Globalscheduling

Interdependences

Phase ordering problems

Motivation for integrated code generation approaches

Compiler structure

IR-level

Target-level

Target code

IR

Program

Main

T3 = T1 + T2

X[3] = T3

T1 = A + 1

T2 = 2 * T1

Example: (LCC-IR)

Works on the LIR representation.

Code generation—Phases

Instruction selection

Target-level

Instruction scheduling

Target-level

Register allocation
Naive instruction selection:

for each LIR operation generate a sequence of equivalent target instructions

```plaintext
{ int i; char c; i = c + 4; }
```

Naive instruction selection, arranged by a postorder traversal:

- `nop`             ! delay slot
- `addi R0,#4,s4`   ! CNSTI
- `load 0(s2),s3`   ! INDIRC
- `addi fp,#8,s2`   ! ADDRLP(c)
- `addi fp,#4,s1`   ! ADDRLP(i)
- `addi s3,s4,s5`   ! ADDI
- `store s5,0(s1)`  ! ASGNI
```

4 cycles, 5 sregs

Instruction selection by Tree-pattern-matching (1):

The LIR has often a finer granularity than the target instructions.

```
{ int i; char c; i = c + 4; }
```

Pattern-matching instruction selection:

```
addt 8(dp),s3 = i
addt 0(#2),s3 = i
addt 6(dp),s3 = i
addt 4(dp),s3 = i
addt 2(dp),s3 = i
addt 0(dp),s3 = i
addt 8(dp),s3 = i
addt 4(dp),s3 = i
addt 2(dp),s3 = i
addt 0(dp),s3 = i
```

- `nop`             ! delay slot
- `addi s3,#4,s4`   ! CNSTI+ADDI
- `store s4,4(fp)`  ! ADDRLP(i)+ASGNI
- `load 8(fp),s3`   ! ADDRFP+INDIRC+CVCI

4 cycles, 4 sregs

Instruction selection by Tree-pattern-matching (2):

Assume we generate code for a tree (of LIR operations).

Find a covering of the tree by treepatterns such that:

+ each LIR operation is covered by exactly one treepattern, and
+ the total cost (sum of the treepattern instances)

Model semantics of each target instruction/addressing mode by a treepattern.

NP-complete for DAGs!

Polynomial for trees (dynamic programming)

Model each treepattern's cost (e.g., worst-case #cc)

Replace treepattern costs by the cost of the instruction plus

+ the cost of addressing modes of the treepatterns that match in the covering
+ the additional cost(s) of the treepattern instances

Formally, we specify for each target machine a treepattern grammar:

```plaintext
G = (N,T,P,s)
```

- `N` = set of nonterminals representing value/storage classes: addr, reg, const, mem...
- `T` = set of terminals "leaf" LIR operators: CNST, ADDRLP...
- `P` = list of production rules:

```
lhs : nonterminal

rhs : treepattern (term) of tree constructors, nonterminals, terminals
```

```
i : target instruction corresponding to this treepattern
c : cost of this production (not including subtreec costs)
```

```
lhs = start symbol
```

Each LIR operation must be covered by exactly one treepattern (and possibly multiple instances of the same)

Each treepattern represents a sequence of equivalent target instructions.

For each LIR operation:

- Generate a covering of equivalent target instructions
- Each treepattern is a node in a pattern-matching instruction section

The LIR has often a finer granularity than the target instructions.

Pattern matching ←

Pattern-matching Instruction Section
Instruction selection by Tree-pattern-matching

Covering: Find a derivation of the LIR tree

Phase 1: Bottom-up labeler

Apply chain rules nonterminal \rightarrow nonterminals as far as possible.
Pick one with locally minimum cost for each application.
If multiple rules apply, nonterminals and their accumulated costs:
Set of derivations that match a symbol.
Annulate each node of the input tree with the least-cost derivation.

Phase 2: Bottom-up rewritemachine (example: IBURG)

By repeatedly applying rules of the LIR tree

Covering: Find a derivation of the LIR tree

Production rules:

\begin{align*}
\text{ADDRLP} & \rightarrow \text{ADDRLP} \\
\text{ASGNI} & \rightarrow \text{ASGNI} \\
\text{ADDI} & \rightarrow \text{ADDI} \\
\text{CVCI} & \rightarrow \text{CVCI} \\
\text{INDIRC} & \rightarrow \text{INDIRC} \\
\text{ADD} & \rightarrow \text{ADD} \\
\text{CNSTI} & \rightarrow \text{CNSTI} \\
\end{align*}

The LIR process is described by a rewrite grammar \( G \) = \( \{ N \} \).

Alternatives: use LR-parsing (Charny/Glavine/78), BURS (Pelegri-Llopart/88),...

IBURG (Fraser/Proebsting/99), BURG (Fraser/Hanson/96),...
TWIG (Fraser/Hanson/96), BURUG (Fraser/Henry/99),
BUR (Fraser/Hanson/96),...

Dynamic programming (Aho/Johnson/76), for code generator generators:
TWIG (Aho/Ganapathi/Tjiang/89, Fraser/Hanson/99),
BURG (Fraser/Hanson/99),...

IBURG (Fraser/Hanson/99),...
GBURG (Fraser/Proebsting/99),...

IBURG (Fraser/Hanson/99),...

...
Phase 2: Top-down Reducer

Found a least-cost derivation:

```
ADDRLP
ASGNI
ADDI
CVCI
INDIRC
addr
con
ADDRLP
ASGNI
ADDI
CVCI
INDIRC
CONSTI
ADDRLP
ASGNI
ADDI
CVCI
INDIRC
addr
con
ADDRLP
ASGNI
ADDI
CVCI
INDIRC
addr
con
ADDRLP
ASGNI
ADDI
CVCI
INDIRC
addr
con
ADDRLP
ASGNI
ADDI
CVCI
INDIRC
addr
con
ADDRLP
ASGNI
ADDI
CVCI
INDIRC
```

Phase 3: Emitter

```
addr <- ADDRLP
addr <- ADDRLP
addr <- ADDRLP
reg <- ADDRLP
reg <- ADDRLP
reg <- ADDRLP
reg <- ADDRLP
```

The pattern matching by dynamic programming (example: IBURG) (9)
Different instruction selections may result in different register needs.

\[ a = 2 \times b \] and \[ a = p + q \] are equivalent to \[ a = p \times q \] and \[ a = p + q \].

Different instruction selections may be considered:

- Integration with instruction scheduling would be great!
- The actual impact on execution time is only known for a given scheduling situation.
- The cost attribute of a production is only a rough estimate.
- Reusable code generation!

### Complexity of Tree Pattern Matching

NP-complete if associativity/commutativity included

Naive: \( O(\#\text{tree patterns} \times \#\text{inputs}) \)

Preprocessing initial patterns \( [\text{Kron'75}, \text{Hoffmann/O'Donnell'82}] \)

\( \text{Theory of (non)deterministic tree automata} \) \( [\text{Ferdinand/Seidl/Wilhelm'92}] \)

For Dags \( \text{NP-complete if associativity / commutativity included} \)

DAGs

\( \text{Conflict with instructions scheduling and register allocation} \)

1. Parse the free grammar
2. Generalize: Position-transition, top-down reducer, emulator automaton
3. Generalize: Position-predicate, top-down reducer, emulator automaton

\( \text{Generation of code selectors from free grammars} \)

\( \text{ReGenability: Generation of code selectors from free grammars} \)
1. Register allocation

determines values (variables, temporaries, large constants) to be kept in registers

2. Register assignment

determines in which register an allocated value should reside

- minimizes traffic CPU registers / caches / memory
- essential for Load/Store architectures
- values that are live simultaneously cannot be kept in same register

Global register allocation by graph coloring

- NP-complete
- requires live ranges to be known
- some linear sequence of code given
- values that are live simultaneously cannot be kept in same register

Global register allocation by graph coloring

- NP-complete
- requires live ranges to be known
- some linear sequence of code given
- values that are live simultaneously cannot be kept in same register

Local register allocation – Local Methods

For variable \( v \) and basic block \( B_i \):

\[
\text{netsave}_v(i) = \text{#uses}(v) - \text{#defs}(v) + \text{locals}(v).
\]

\[
\text{defsave}_v(i) = \text{#uses}(v).
\]

\[
\text{usecost}_v(s) = \begin{cases} \text{if Load} \left( v \right) \text{needed at beg. of } B_i, 1 \\text{otherwise, } 0 \end{cases}.
\]

\[
\text{stcost}_v(s) = \begin{cases} \text{if Store} \left( v \right) \text{needed at end of } B_i, 1 \\text{otherwise, } 0 \end{cases}.
\]

For loop \( L \) estimate benefit of keeping \( v \) in a register:

\[
\text{benefit}_v(L) = \sum_{i} \text{netsave}_v(i) \cdot \text{usecost}_v(s) + \text{defsave}_v(i) \cdot \text{stcost}_v(s).
\]

With \( R \) registers available:

- allocate the \( R \) objects with greatest benefit in \( L \)
- moves may be necessary instead of Load/Store
- add worst-case terms if \( v \) could reside in different registers in \( \text{Pred}(B_i), B_i, \text{Succ}(B_i) \)

Allocate the \( R \) objects with greatest benefit in \( L \) if:

\[
\text{benefit}_v(L) > R \cdot \text{usecost}_v(s).
\]
Global register assignment by graph coloring

1. allocate objects that can be assigned to registers to distinct symbolic registers $s_1, s_2, ...$
2. determine candidates for allocation to registers (e.g., webs)
   - to distinct symbolic registers $s_1, s_2, ...$
   - allocate objects that can be assigned to registers

Allocatable objects: webs

- each SSA-form variable is head of a DU-chain
- each web is equivalent to a symbolic register $s_i$
- live ranges instead of variable names
- less constraints, less interferences
- easy to determine from SSA form
- less variables, less interferences

Coloring the interference graph with $R$ colors

1. build interference graph
2. determine interference graph to allocate to registers
   - to distinct symbolic registers $s_1, s_2, ...$
   - allocate objects that can be assigned to registers

Each allocatable object is a symbolic register

Ershov'62
Cocke'71
Chaitin et al.'81
Chaitin'82
Chow/Hennessy'84,'90
Briggs et al.'89
Briggs'92

Coloring the interference graph with $R$ colors

- NP-complete for $R \geq 3$ [Garey/Johnsson'79]
- if $k$-clique lower bound $k$ registers
- Heuristic for coloring graph $G$ with $R$ colors:
  - while $V / \emptyset$
    - choose some $v \in V$
    - if $v$ has less than $R$ neighbors in $G$
      - delete $v$
    - else
      - modify IR + interf.-graph (coalesce/spill/rematerialize nodes)
        and restart.
Optimistic coloring

May be colorable even if \( R \) has \( > \) neighbors

Conservative coloring:

- May make \( R \) harder to color
- \( p \) \( \text{max}(\text{deg}) \) \( \text{deg} \) is possible

Coalescing:

- Coalesce only if \( \text{deg}(u \& v) > R \)
- Can color \( u \& v \) by the naive degree \( \times R \) heuristic

Coalescing two live ranges \( u \) and \( v \) can increase the node degree

Conservative coalescing:

- Coalesce only if \( \text{deg}(u \& v) > R \)
- Can color \( u \& v \) by the naive degree \( \times R \) heuristic

Spillgraph must be updated (some interferences disappear)

Compressed Table of Contents

Coalescing = fusion of webs

Register coalescing | Register subsumption

Interference graph must be updated (some interferences disappear)
REMEMERIZATION

A copy instruction whose source or target is spilled can be removed.

\[
\text{spillcost} = (\text{copy}_{\text{def}} + \text{copy}_{\text{use}} + \text{store}_{\text{use}}) \times \text{depth} + \text{degree} \times \text{C成本}
\]

Minimize ratio \(\text{spillcost} / \text{degree}\) etc.

[Heinemann et al.'89]

Heuristic choice of the best spill candidate

consider rematerialization as an alternative to spilling

- avoids redundant loads and stores
- spill value once per block if possible

\text{copy}_{\text{use}} = \text{load}_{\text{use}} + (\text{copy}_{\text{def}} + \text{store}_{\text{def}}) / \text{degree}_{\text{def}} + \text{store}_{\text{def}}$

ADVANCED COMPILER CONSTRUCTION—Register allocation.

CHOW/HENNESSY'84,'90

(\text{Chow/Hennessy'84})

= the range splitting

\text{consider rematerialization as an alternative to spilling}

a code for rematerialization is necessary only before the first of them.

The resolved value remains live for several adjacent uses.

If a spilled value is used several times

A copy instruction whose source or target is spilled can be removed.

\[
\text{spillcost} = (\text{copy}_{\text{def}} + \text{copy}_{\text{use}} + \text{store}_{\text{use}}) \times \text{depth} + \text{degree} \times \text{C成本}
\]

Minimize ratio \(\text{spillcost} / \text{degree}\) etc.

[Heinemann et al.'89]

Heuristic choice of the best spill candidate

consider rematerialization as an alternative to spilling

- avoids redundant loads and stores
- spill value once per block if possible

\text{copy}_{\text{use}} = \text{load}_{\text{use}} + (\text{copy}_{\text{def}} + \text{store}_{\text{def}}) / \text{degree}_{\text{def}} + \text{store}_{\text{def}}$

ADVANCED COMPILER CONSTRUCTION—Register allocation.
Hierarchical Register Allocation

[Callahan/Koblenz PLDI’91]

1. find hierarchical structure (e.g., regions)
2. color intervals bottom-up with Chaitin-style allocator, using local and global interference information
3. propagate summary information from children to parents
d. top-down pass assigns the registers and places spill code
4. better placement of spill code
5. smaller interference graphs considered at each step
6. smaller interference graphs considered at each step

Advantages over on-demand allocation:

- no applicable to recursive functions or calls via fn.pointers
- encode annotations for register allocation at link time
- general complete executable code
- at link time: [Wall’86]

---

Interprocedural Register Allocation (3)

lod.

v : var.

v loaded, modified in this block before value is last used

sto.

v : value of v is used after current use

Example:

x = y = a++ - b

r1

a

lod.

a

replace by copy

r1

reg

a

if a alloc.in areg.

r2

r1

1

res.

a

r2

r3

b

rmv.

b

r4

r2

r3

op2.

b

y

r4

sto.

a

replace by copy

r1

reg

a

if a alloc.in areg.

x

r4

rmv.

x

etc.

patched code if x, y allocated to registers
Interprocedural Register Allocation (4)

Register allocation "relocation" information:

foreach module:
list of functions

foreach function:
list of locals, referenced variables, callee's annotations

At link time: traverse call DAG bottom-up
functions in disjoint subtrees of the call tree can share registers

Identify groups of variables that cannot be live simultaneously
weighted by execution frequencies (profiling information)

Maybe using graph coloring

Integration of register allocation and instructions scheduling

Techniques:

Power consumption

Optimization goals:

Execution time

Optimizations:

Space (registers, stack size)

Techniques:

Integrating annotations, dependence graph scheduling

Quantitative evaluation

Precedence constraints, dependence graph, scheduling

Deeming the ranges of registers a linear sequence of instructions

Register allocation "relocation" information:

Interprocedural Register Allocation (4)
Local instruction scheduling and register allocation

- **Register allocation**
  - **S** is space-optimal (m(S) \( \leq \) m(S') for all S' of G)
  - **S** is time-optimal (time(S) \( < \) time(S') for all S' of G)

**Register Allocation and Spilling**
- **MRIS**
  - Minimum register need instructionscheduling
  - Incomplete, requires special code to be inserted at run-time

Optimization Problems in Local Instruction Scheduling

- **WRS**
  - Simultaneous minimization of space and time
  - Compiler-generated spill code cannot be eliminated at run-time

- **RR**
  - Minimum instruction level parallelism (for superscalars/VIW)
  - Minimum register delays

- **MTIS**
  - Register-constrained minimum time instructionscheduling

- **SWRS**
  - Advanced compiler construction — Instructionscheduling

- **MRIS**
  - Minimum register need instructionscheduling

- **MRTIS**
  - Minimum register time instructionscheduling

- **MRTIS**
  - Minimum time instructionscheduling

- **SEMI**
  - Minimum space instructionscheduling

- **NR**
  - N\(\{0\}\)-complete

- **N\{1\}**
  - N\(\{0\}\)-complete

- **Garey/Johnson '79, Gross '83, Lawler et al. '87**

- **K., Paul, Rauber '91**

- **K. '96**

- **K., Rauber '93/'95**

- **Govindarajan et al. '00**

- **Sethi'75, Ullman '70**

- **Göttler '81, Rosenberg et al. '90**

- **Random Sele**

- **MRIS**
  - Based on finding instruction heuristics in the DAG
  - (b) Based on topological sorting of the DAG
  - (c) Based on finding instruction lineages in the DAG

- **List scheduling = local scheduling by topological sorting**

- **Register allocation**
  - Advanced compiler construction — Instructionscheduling

- **Instruction scheduling**
  - Advanced compiler construction — Instructionscheduling

- **Spilling**
  - Store/reload takes additional time
  - Power consumption in embedded processors increases with number of memory accesses

- **Shadow registers**
  - Takes additional time

- **Superscalar processors**
  - Compiler-generated spill code cannot be eliminated at runtime

- **NP-complete**

- **Spilling**

- **Simultaneous minimization of space and time**

- **Simultaneous minimization of resource and time**

- **Register allocation**

- **Optimization problems in local instruction scheduling**

- **N\{0\}**

- **N\{1\}**

- **Garey/Johnson '79, Gross '83, Lawler et al. '87**

- **K., Paul, Rauber '91**

- **K. '96**

- **K., Rauber '93/'95**

- **Govindarajan et al. '00**

- **Sethi'75, Ullman '70**

- **Göttler '81, Rosenberg et al. '90**

- **Random Sele**

- **MRIS**
  - Based on finding instruction heuristics in the DAG
  - (b) Based on topological sorting of the DAG
  - (c) Based on finding instruction lineages in the DAG

- **List scheduling = local scheduling by topological sorting**

- **Register allocation**
  - Advanced compiler construction — Instructionscheduling

- **Instruction scheduling**
  - Advanced compiler construction — Instructionscheduling

- **Spilling**
  - Store/reload takes additional time
  - Power consumption in embedded processors increases with number of memory accesses

- **Shadow registers**
  - Takes additional time

- **Superscalar processors**
  - Compiler-generated spill code cannot be eliminated at runtime

- **NP-complete**

- **Spilling**

- **Simultaneous minimization of space and time**

- **Simultaneous minimization of resource and time**

- **Register allocation**

Greedy list scheduling for VLIW architectures

Heuristic for moving instructions to fill the branch delay slot:

- not applicable to "branch always" lists: fixed delay slot
  - branch as not if not taken
- delay slot instruction executed only if a conditional branch is taken
  - activated by setting a bit in the instruction opcode
  - executed by setting a bit in the instruction opcode
  - in one branch direction

Multifunction feature (SPARC) for conditional branch delay slots

Global instruction scheduling: Branch scheduling

Global instruction scheduling: Trace scheduling (1)

Developed for VLIW architectures

- idea: enlarge the scope of local scheduling to traces
- idea: make the most frequent trace last
- track execution frequencies of BBs/Traces (e.g., profiling)
- trace = acyclic path of basic blocks in the CFG

Global instruction scheduling: Trace scheduling (2)

- idea: enlarge the scope of local scheduling to traces
- idea: make the most frequent trace last
- track execution frequencies of BBs/Traces (e.g., profiling)
- trace = acyclic path of basic blocks in the CFG

"Branch as not" if not taken
- delay slot instruction executed only if a conditional branch is taken
- activated by setting a bit in the instruction opcode
- executed by setting a bit in the instruction opcode
- in one branch direction

Multifunction feature (SPARC) for conditional branch delay slots

Heuristic for moving instructions to fill the branch delay slot:

- not applicable to "branch always" lists: fixed delay slot
  - branch as not if not taken
- delay slot instruction executed only if a conditional branch is taken
  - activated by setting a bit in the instruction opcode
  - executed by setting a bit in the instruction opcode
  - in one branch direction

Multifunction feature (SPARC) for conditional branch delay slots

Global instruction scheduling: Trace scheduling (1)

Developed for VLIW architectures

- idea: enlarge the scope of local scheduling to traces
- idea: make the most frequent trace last
- track execution frequencies of BBs/Traces (e.g., profiling)
- trace = acyclic path of basic blocks in the CFG

Global instruction scheduling: Trace scheduling (2)

- idea: enlarge the scope of local scheduling to traces
- idea: make the most frequent trace last
- track execution frequencies of BBs/Traces (e.g., profiling)
- trace = acyclic path of basic blocks in the CFG

"Branch as not" if not taken
- delay slot instruction executed only if a conditional branch is taken
- activated by setting a bit in the instruction opcode
- executed by setting a bit in the instruction opcode
- in one branch direction

Multifunction feature (SPARC) for conditional branch delay slots
Global instructions scheduling: Percolation scheduling (1)

Nicolau [84] / Edirgul / Nicolau [89]

works on different IRs: parallel computation graph

computation nodes: entry, exit, sets of operations running in parallel (loads, stores)

edges: sequencing, conditional branches to successor computation nodes

set of operations running in parallel (loads, stores)

repeatedly apply a set of local code transformations in disorder:

Example:

Codereordering with insertion of compensation code

Examples:

Interchange branches

Moving a branch

Moving assignments across conditional branches

Interchange assignment and label

Hold an assignment in assignment with insertion of compensation code

Insert compensation code if necessary

unify: move set of global instructions from all successors

move conditional to predecessor node

move instructions to predecessor node (towards entry)

delete empty computation node

The four local transformations

Global instructions scheduling: Percolation scheduling (2)
Global instructionscheduling: Percolation scheduling

Percolation scheduling (3)

Finding a sequence of transformations that yield an optimal solution is NP-complete.

Heuristics guide selection of transformations.

Additional global transformations complement the local transformations.

Local methods complement heuristics guide selection of transformations.

Scheduling-related issues, not covered

Creating and scheduling predicated code

Speculation (with and without hardware support)

Run-time scheduling, profile-driven scheduling

Interferences with instruction selection, register allocation,

Interferences with data layout, exploit advanced addressing units,

Interferences with instruction selection, register allocation,

See also lecture on integrated code generation for DSPs.

Heuristic measure for average degree of parallelism in a region:

# instructions(region) / length of critical path(region) - Slow

Program region = one of several BBs that require the same control condition

This avoids idle cycles caused by regions with insufficient parallelism

See also lecture on energy-aware code generation

See also lecture on software pipelining

See also lecture on integrated code generation

See also lecture on code generation for DSPs.

See also lecture on energy-aware code generation

Heuristic measure for average degree of parallelism in a region:

# instructions(region) / length of critical path(region)

Heuristic measure for average degree of parallelism in a region:

# instructions(region) / length of critical path(region)

Heuristic measure for average degree of parallelism in a region:

# instructions(region) / length of critical path(region)

Heuristic measure for average degree of parallelism in a region:

# instructions(region) / length of critical path(region)