**CODE GENERATION**

**Instruction selection**

- **Naive instruction selection**:
  - For each LIR operation, generate a sequence of equivalent target instructions.

**Naive instruction selection**

- **Tree-pattern matching**:
  - Model semantics of each target instruction/addressing mode by a tree pattern.
  - Find a covering of the tree by tree patterns such that:
    - Each LIR operation is covered by exactly one tree pattern,
    - The total cost (sum over all costs of the tree pattern instances that match) is minimized.

**Tree-pattern matching**
Instruction selection by tree-pattern-matching

Formally, we specify for each target machine a tree grammar $G = (N, T, P, S)$, where:

- $N = \text{set of nonterminals}$
- $T = \text{set of terminals}$
- $P = \text{list of production rules}$
- $S = \text{start symbol}$

A production rule is of the form:

```
  lhs ::= nonterminal
  rhs ::= \text{tree pattern} (term) of tree constructors, nonterminals, terminals
```

Let $i$ be the target instruction corresponding to this tree pattern, and $c$ its cost.

Example free grammar:

```
S ::= ADRALP
A ::= ADRALP
   ::= ADDI \reg, \con
   ::= ADDI \reg, \reg
   ::= ASGNI \addr, \reg
   ::= ASGNI \reg, \reg
   ::= ADDRLP
   ::= ADDRLP
   ::= ADDRLP
   ::= ADDRLP
   ::= ADDRLP
```

Diagram:

```
1. \text{ADRALP} \rightarrow \text{add} \reg \con
2. \text{add} \reg \con
3. \text{addr} \con
4. \text{addr} \reg
5. \text{addr} \reg
```

Some production rules in $P$:

```
\{ \text{CSTI, ADRALP, ADRALP, ADRALP} \}
```

Covering: Find a derivation of the LIR tree by repeatedly applying rules of $P$ to the start symbol $s$.

Derivation is not unique. Find a least-cost derivation of the LIR tree by a greedy approach. A naive approach may be insufficient:

```
  \text{cost of a derivation = sum over costs of productions applied}
```

By backtracking (= enumerate all possible coverings), we can find a least-cost derivation. For code generator generators:

```
  TWIG \[Aho/Ganapathi/Tjiang'89\], BURG \[Fraser/Henry/Proebsting'92\], GBURG \[Fraser/Proebsting'99\],...
```

Alternatively, use LR-parsing $[\text{Graham/Glanville'78}]$, BURS $[\text{Fraser/Hanson'96}]$,...

Covering: Find a derivation of the LIR tree by repeatedly applying rules of $P$ to the start symbol $s$.

```
  \text{production rule for each possible target inst, + addr-mode}
  \text{cost of production = sum over costs of productions applied}
```

```
  \text{least LIR operation (term) of free constructors, nonterminals, terminals}
```

```
  \text{greedy} \rightarrow \text{backtracking} \rightarrow \text{dynamic programming}
  \text{[Aho/Johnson'76]}
  \text{bottom-up rewrite machine (BURM), for code generator generators:}
  \text{TWIG \[Aho/Ganapathi/Tjiang'89\], BURG \[Fraser/Henry/Proebsting'92\], GBURG \[Fraser/Proebsting'99\],...}
```

Treepattern matching by dynamic programming (example: IBURG)

(i) Bottom-up labeller

1. Phase 1: bottom-up labeller: label each node of the input tree with the set of tree patterns that match and their accumulated costs; if multiple rules apply, choose the one with lowest minimum cost.

2. Phase 2: top-down reducer

i. Basis: root of the labelled tree must correspond to the start symbol (stmt)
ii. Induction step: apply the corresponding productions.

3. Phase 3: emitter

i. Basis: execute additional compiler code associated with these rules.
ii. Induction step: emit the assembler code for each production applied in reverse order of the derivation found in phase 2.

e.g. Register allocation:

```
reg -> ADDLR(reg)
reg -> ADDLR(con)
addr -> ADDLR(addr)
```

Retargetability: Generation of code selectors from tree grammars

Given: a tree grammar describing the target processor

1. Parse the tree grammar
2. Generate: bottom-up labeller, top-down reducer, emitter
3. Retargetable code generation!
Complexity of tree pattern matching

- NP-complete if associativity/commutativity included
- naive: $O(#\text{treepatterns} \times \text{sizeofinputtree})$
- preprocessing initial tree patterns [Kron'75] [Hoffmann/O'Donnell'82] may require exponential space/time but there tree pattern matching in time $O(\text{sizeofinputtree})$
- theory of (non)deterministic tree automata [Ferdinand/Seidl/Wilhelm'92]
- different instruction selections may result in different register need.

Different instruction selections may result in different register need.

$+ a = 2^p \rightarrow a = b \gg 1$ and $a \oplus b$

The actual impact on execution time is only known for a given

The cost attribute of a production is only a rough estimate.

The actual impact on execution time is only known for a given

The cost attribute of a production is only a rough estimate.

The actual impact on execution time is only known for a given

The cost attribute of a production is only a rough estimate.

The actual impact on execution time is only known for a given

The cost attribute of a production is only a rough estimate.
Global Register Assignment by Graph Coloring - Overview

1. Allocate objects that can be assigned to registers
   to distinct symbolic registers $s_1, s_2, \ldots$

2. Determine candidates for allocation to registers ($s_i / \text{webs}$)
   to distinct symbolic registers $s_1, s_2, \ldots$

3. Allocate objects that can be assigned to registers

4. Colors with $\leq k$ colors

5. Allocate each object to a register that has the same color

+ Allocatable objects: $\text{webs}$

+ Coloring the interference graph with $R$ colors

- Build interference graph

- Determine candidates for allocation to registers ($s_i / \text{webs}$)

- Allocate objects that can be assigned to registers

+ Each web is equivalent to a symbolic register
+ Easy to determine from SSA form
  - Each symbolic register is head of a DU-chain
+ Lexicographically(minimum) ordered variable names
+ Less constraints, less interferences
+ Webs = max. union of DU-chains ($p, m$)
  - Each overlapping in at least one use

+ Coalescing = fusion of webs

-register allocation

and remove the copy operation.

and $s_i$ and $s_j$ are not rewritten after the copy operation.

where $s_i$ and $s_j$ do not interfere with each other.

- Coalescing / Register subsumption:

- Coalescing = fusion of webs

$\text{webs} = \text{max. union of DU-chains}$

Proceed optimistically (postponing spilling decisions)
$G$ may be colorable even if $k$ has $\geq R$ neighbors

Variant: Optimistic coloring

Graph coloring

$\leq k$-clique

NP-complete for $R = 3$

Garey/Johnsson'79

Chaitin'81

while $V$ $\neq 0$

chose some $v \in V$

if $v$ has less than $R$ neighbors in $G$ choose some $\in A$

where $A$ = $\emptyset$

then there will be some color left for $v$. Delete v from C.

if $v$ has less than $R$ neighbors in $G$

+ $k$-clique $\Rightarrow$ lower bound on registers

Garey/Johnson'79

$G$ may be colorable even if $v$ has $\geq R$ neighbors

proceed optimistically (postponing spilling decisions)

+ Each web is equivalent to a symbolic register
+ Easy to determine from SSA form
  - Each symbolic register is head of a DU-chain
+ Allocatable objects: $\text{webs}$

+ Coloring the interference graph with $R$ colors

- Build interference graph

- Determine candidates for allocation to registers ($s_i / \text{webs}$)

- Allocate objects that can be assigned to registers

+ Each web is equivalent to a symbolic register
+ Easy to determine from SSA form
  - Each symbolic register is head of a DU-chain
+ Allocatable objects: $\text{webs}$

- Coalescing = fusion of webs

$\text{webs} = \text{max. union of DU-chains}$

Proceed optimistically (postponing spilling decisions)
$G$ may be colorable even if $k$ has $\geq R$ neighbors

Variant: Optimistic coloring

Graph coloring

$\leq k$-clique

NP-complete for $R = 3$

Garey/Johnsson'79

Chaitin'81

while $V$ $\neq 0$

chose some $v \in V$

if $v$ has less than $R$ neighbors in $G$ choose some $\in A$

where $A$ = $\emptyset$

then there will be some color left for $v$. Delete v from C.

if $v$ has less than $R$ neighbors in $G$

+ $k$-clique $\Rightarrow$ lower bound on registers

Garey/Johnson'79

$G$ may be colorable even if $v$ has $\geq R$ neighbors

proceed optimistically (postponing spilling decisions)

+ Each web is equivalent to a symbolic register
+ Easy to determine from SSA form
  - Each symbolic register is head of a DU-chain
+ Allocatable objects: $\text{webs}$

+ Coloring the interference graph with $R$ colors

- Build interference graph

- Determine candidates for allocation to registers ($s_i / \text{webs}$)

- Allocate objects that can be assigned to registers

+ Each web is equivalent to a symbolic register
+ Easy to determine from SSA form
  - Each symbolic register is head of a DU-chain
+ Allocatable objects: $\text{webs}$
Spilling

If a spilled value is used several times and the restored value remains live for several adjacent uses, a Load/Rematerialize is necessary only before the first of them.

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

Modify $spillcost(w)$ accordingly:

$spillcost(w) = \sum_{def(w)} depth + \sum_{use(w)} depth + \sum_{move(w)} depth$

Heuristic choice of the best spill candidate:

If a spilled value is used several times (some interferences disappear)

Spilling versus Rematerialization

A call site's access usage information for the callee

- Functions in different subtrees of the call graph may share same regs
- For calls with statically known callee:
  - pass parameters in any register
  - minimize reg-saving+rematerializing at calls „deep“ in the call graph
  - all register-caller-save
  - track register usage at call sites and of callees (direct/indirect)
  - [Chow `89]

Interprocedural Register Allocation (2)

at link time:

+ generate complete, executable code
+ encode annotations for register allocation at link time similar to relocation information
+ not applicable to recursive functions not calls via pointers
+ not applicable to recursive functions
+ escape annotations for register allocation at link time
+ generalize compiler-extractable code
+ all registers caller-saved
+ minimize reg.-saving+restoring at calls “deep” in the call graph
+ track parameters in any register
+ functions in different subtrees of the call graph may share same regs
+ pass parameters in any register
+ minimize reg-saving+rematerializing at calls „deep“ in the call graph
+ all register-caller-save
+ track register usage at call sites and of callees (direct/indirect)
+ [Chow `89]

Interprocedural Register Allocation (1)

at compile time:

+ naive: save (caller/callee) at call, restore at return
+ for call with statically known callee:
  - function in different subtrees of the call graph may share same regs
  - pass parameters in any register
  - minimize reg-saving+rematerializing at calls „deep“ in the call graph
  - all register-caller-save
  - track register usage at call sites and of callees (direct/indirect)
  - [Chow `89]
Interprocedural Register Allocation

Register allocation and instructions scheduling are tightly coupled. The choice of register allocation strategy for a procedure can affect the quality of the generated code. Specifically, the choice of register allocation for a procedure can affect the scheduling of instructions in that procedure. For example, if a procedure is compiled with load time allocation, then the instructions in the procedure can be scheduled independently of each other, since the registers are allocated at runtime. However, if a procedure is compiled with compile time allocation, then the instructions in the procedure must be scheduled together, since the registers are allocated at compile time.

In the case of compile time allocation, the register allocator must make a decision about which registers to allocate to which variables. This decision is based on a number of factors, including the size of the registers, the types of values that will be stored in the registers, and the need to ensure that the variables are stored in registers that are visible to other procedures. The register allocator must also take into account the fact that some variables may be referenced by parameters, which may require that certain registers be reserved for the parameters.

Once the register allocator has made its decision about which registers to allocate to which variables, it must then schedule the instructions in the procedure. The scheduler must ensure that the instructions are scheduled in such a way that all of the variables are visible to the instructions that use them. This can be a challenging task, since the instructions in the procedure may be dependent on each other, and the scheduler must ensure that the instructions are scheduled in a way that satisfies these dependencies.

In summary, the register allocator and the instructions scheduler must work together to ensure that the generated code is as efficient as possible. The register allocator must make decisions about which registers to allocate to which variables, and the scheduler must ensure that these decisions are respected when scheduling the instructions.
Optimization goals:

- Execution time
- Space (registers, stack size)
- Power consumption

Techniques:

- Precedence constraints: Dependence graph
- Algorithms for special cases
- Heuristics (listscheduling)
- Exhaustive search/CLP/Branch&Bound/dynamic programming
- Integer linear programming (ILP)

Optimization Problems in Local Instruction Scheduling

- **MRIS** – Minimum Register Need Instruction Scheduling
  - Spilling (store/reload) takes additional time
  - Power consumption in embedded processors increases with # memory accesses
  - Superscalar processors with shadow registers and register renaming
  - Compiler-generated spill code cannot be eliminated at runtime
  - NP-complete
  - [Sethi'75]

- **MTIS** – Minimum Time Instruction Scheduling
  - Hiding pipeline delays
  - Exploiting instruction-level parallelism (for superscalar/VLIW)
  - NP-complete
  - [Garey/Johnson'79, Gross'83, Lawler et al.'87]

- **RCMTIS** – Register-Constrained Minimum Time Instruction Scheduling

- **SMRTIS** – Simultaneous Minimization of Space and Time

Space-Optimal Scheduling Strategies for DAGs

- General schedules
  - Space-optimal (enumeration + dynamic programming)
- Contiguous schedules
  - Space-optimal (dynamic programming)
- Random list scheduling of the DAG
- Topological sorting of the DAG

Optimization Goals:

- Integer linear programming (ILP)
- Exhaustive search/CLP/Branch&Bound/dynamic programming
- Heuristics (first scheduling)
- Algorithms for special cases
- Precedence constraints: Dependence graph

Techniques:

- Power consumption
- Space (registers, stack size)
List Scheduling = Local Scheduling by Topological Sorting

select DAG \ G \ z' \ \text{scheduled} (z)

DAG \ G \ z \ u \ v \ \text{top sort}

Set \ z, \ \text{int} [\ \text{INDEG}]

\text{if} \ z \neq 0 \ // \ t \ n

select \ \text{arbitrary node} \ v \ z; // \text{implicitly remove all edges} \ v \ u:

\text{INDEG} \ u \ ! \ \text{INDEG} \ u \ # \ \text{INDEG} \ u \ 1 \ \text{where} \ v \ u

\text{elsewhere} // \text{update zero-indegree set:} \ z! \ z \ v$

\text{newleaves}: \ \text{INDEG} \ u \ 0$

S \ % \ t \ & \ v$

\text{top sort} \ z! \ \text{INDEG}' \ 1$

else // output \ S \ % 1:n

Call \ \text{top sort} \ z_0, \ \text{INDEG} \ 0 \ 1$

produces a schedule in \ S \ % 1:n

Greedy listscheduling: race scheduling

\begin{array}{ccc}
\text{FltOp} & \text{FldLd} & \text{IntOp} \\
\text{IntLd} & & \\
\text{IntMem-Unit} & & \\
\text{FltOp} & & \\
\text{FltLd} & & \\
\text{IntOp} & & \\
\end{array}

Global instructionscheduling: Branch scheduling
delayed branch: typically 1 delay slot (PA-RISC, SPARC)
fill delay slot with useful instructions of the zeroindegree set:
• Nullification feature (SPARC) for conditional branch delayslots
  not applicable to "branch always": fixed delay slot
  • Conditional branch is taken
  • Deep slot instruction executed only if a conditional branch is taken
  • Stopped by filling a slot in the instruction opcode
  in one branch direction
  • Multilevel feature (PA-RISC) for conditional branch delay slots

Global instructionscheduling: Trace scheduling
developed for VLIW architectures (Fisher 81, Ellis 85)
+ idea: enlarge the scope of local scheduling to traces
+ idea: make the most frequent trace fast:
  + track execution frequencies of BBs/Traces (e.g., profiling)
  + trace = acyclic path of basic blocks in the CFG
  + make: 

Global instructionscheduling: Branch scheduling

Little delays with useful instructions instead of NOP
+ little delays with useful instructions instead of NOP
  + delayed branch: virtually delay a slot (PA-RISC, SPARC)

Global instructionscheduling: Local Scheduling = List Scheduling by Topological Scheduling

A greedy heuristic for list scheduling with ready instructions of the zeroindegree set
In one step as many slots in a VLIW word as possible
A greedy heuristic for list scheduling

Heuristic for moving instructions to fill the branch delay slot:
+ not applicable to "branch always": fixed delay slot
+ delayed as NOP if not taken.
  + deep slot instruction executed only if a conditional branch is taken
  + stopped by filling a slot in the instruction opcode
  + in one branch direction
  + Multilevel feature (PA-RISC) for conditional branch delay slots

[ Muchnick 7.4.1 ]
Global instruction scheduling: Region scheduling

- Instruction region length / length of critical path (region)

Heuristic measure for average degree of parallelism in a region:

To obtain the degree of parallelism:

- Merging of regions into BBs with insufficient parallelism
- Merging instructions from BBs with excess parallelism
- Moving instructions from BBs with excess parallelism
- Repeatedly apply a set of local code transformations:
  - Repeatedly apply a set of local code transformations:
  - Program region = one of several BBs that require the same control condition
  - Defer: avoid idle cycles caused by regions with insufficient parallelism

Translation of if e then S1 else S2 (assuming nodep. e):

\[ S_1 = \text{spawn} T_1; S_2 = \text{spawn} T_2; \]
\[ \text{for concurrent execution of } S_1 \text{ and } S_2 \]
\[ T_1: \text{execute } S_1 \text{ speculatively, } \text{die} \]
\[ T_2: \text{execute } S_2 \text{ speculatively, } \text{die} \]
\[ \text{the main thread evaluates } e; \]
\[ \text{if } (e \text{ true}) \text{ then } \text{kill } T_2; \text{join } T_1; \text{commit result of } T_1 \]
\[ \text{else } \text{kill } T_1; \text{join } T_2; \text{commit result of } T_2 \]
Phase ordering problem

Register allocation before scheduling introduces additional data dependences

Example:

\[ \begin{align*}
\text{reg} & \quad t_1 \quad \text{reg} & \quad t_3 \\
R_1 & \quad \implies & \quad "b \quad \text{before} \quad c" & \quad , \quad \text{as} \quad c \quad \text{overwrites} \quad t_1 \quad \text{in} \quad R_1.
\end{align*} \]

(a) Register allocation before scheduling

(b) Register allocation after scheduling

Graph-based method: Naive Enumeration, Selection Tree (1)

\[
(u \cdot u)(O) = O(u \cdot \text{enumerated schedules})
\]

Example:

\[ \begin{align*}
\text{edge} & \quad (2 \to 7) & \quad \text{Illegitimate ordering, } \text{edge by edge}\ \\
\text{nodes} & \quad = \text{ins. of zero-indeg sets}
\end{align*} \]

From Selection Tree to Selection DAG (1)

\[ \begin{align*}
\text{level 0} & \quad \text{level 1} & \quad \text{level 2} \\
\text{a} & \quad \text{c} & \quad \text{b}
\end{align*} \]

\[ \begin{align*}
\text{abc} & \quad \text{ac} & \quad \text{ce} \\
\text{e} & \quad \text{d} & \quad \text{f}
\end{align*} \]

Observations:

1. In general: multiple instances of same zero-indeg set
2. All paths to instances of the same zero-indeg set have equal length
3. Less parallelism / alternatives

Naive Enumeration, Selection Tree (2)

\[ \begin{align*}
\text{level 0} & \quad \text{level 1} & \quad \text{level 2} \\
\text{a} & \quad \text{c} & \quad \text{b}
\end{align*} \]

\[ \begin{align*}
\text{abc} & \quad \text{ac} & \quad \text{ce} \\
\text{e} & \quad \text{d} & \quad \text{f}
\end{align*} \]

Observations:

1. All codes must be scheduled
2. Less parallelism / alternatives

\[ \begin{align*}
\text{Selection before \quad scheduling} \quad \leftrightarrow \quad \text{Selection after \quad scheduling}
\end{align*} \]

\[ \begin{align*}
\text{Example:}
\end{align*} \]
Improvement on selection construction of the selection DAG.

Lemma: If selection node is optimal for \( C \), the schedule \( S \) is as follows.

By induction follows:

- The schedule \( S \) is as optimal as possible.
- The register need \( m \) to be minimized.
- Hence we may choose any path in the same zero-deg set.

\[ \{ (z) \} = \{ (z) \} = \{ (z) \} \text{ for } z \in S \text{ scheduled} \]

The same values reside in registers. Namely:

- Corresponding set \( S \) of selection DAG is as optimal as possible.
- For each selection DAG path \( P \), ending in the same zero-deg.

Proof (idea): For each selection DAG path \( P \), ending in the same zero-deg.

Lemma: If selection node is optimal for \( C \), the schedule \( S \) is as optimal as possible.

The schedule \( S \) is as optimal as possible.

- The register need \( m \) to be minimized.
- Hence we may choose any path in the same zero-deg set.

\[ \{ (z) \} = \{ (z) \} = \{ (z) \} \text{ for } z \in S \text{ scheduled} \]

The same values reside in registers. Namely:

- Corresponding set \( S \) of selection DAG is as optimal as possible.
- For each selection DAG path \( P \), ending in the same zero-deg.

Proof (idea): For each selection DAG path \( P \), ending in the same zero-deg.

**Lemma:** If selection node is optimal for \( C \), the schedule \( S \) is as optimal as possible.

The schedule \( S \) is as optimal as possible.

- The register need \( m \) to be minimized.
- Hence we may choose any path in the same zero-deg set.

\[ \{ (z) \} = \{ (z) \} = \{ (z) \} \text{ for } z \in S \text{ scheduled} \]

The same values reside in registers. Namely:

- Corresponding set \( S \) of selection DAG is as optimal as possible.
- For each selection DAG path \( P \), ending in the same zero-deg.

Proof (idea): For each selection DAG path \( P \), ending in the same zero-deg.

Lemma: If selection node is optimal for \( C \), the schedule \( S \) is as optimal as possible.

The schedule \( S \) is as optimal as possible.

- The register need \( m \) to be minimized.
- Hence we may choose any path in the same zero-deg set.

\[ \{ (z) \} = \{ (z) \} = \{ (z) \} \text{ for } z \in S \text{ scheduled} \]

The same values reside in registers. Namely:

- Corresponding set \( S \) of selection DAG is as optimal as possible.
- For each selection DAG path \( P \), ending in the same zero-deg.

Proof (idea): For each selection DAG path \( P \), ending in the same zero-deg.

**Lemma:** If selection node is optimal for \( C \), the schedule \( S \) is as optimal as possible.

The schedule \( S \) is as optimal as possible.

- The register need \( m \) to be minimized.
- Hence we may choose any path in the same zero-deg set.

\[ \{ (z) \} = \{ (z) \} = \{ (z) \} \text{ for } z \in S \text{ scheduled} \]

The same values reside in registers. Namely:

- Corresponding set \( S \) of selection DAG is as optimal as possible.
- For each selection DAG path \( P \), ending in the same zero-deg.

Proof (idea): For each selection DAG path \( P \), ending in the same zero-deg.

**Lemma:** If selection node is optimal for \( C \), the schedule \( S \) is as optimal as possible.

The schedule \( S \) is as optimal as possible.

- The register need \( m \) to be minimized.
- Hence we may choose any path in the same zero-deg set.

\[ \{ (z) \} = \{ (z) \} = \{ (z) \} \text{ for } z \in S \text{ scheduled} \]

The same values reside in registers. Namely:

- Corresponding set \( S \) of selection DAG is as optimal as possible.
- For each selection DAG path \( P \), ending in the same zero-deg.

Proof (idea): For each selection DAG path \( P \), ending in the same zero-deg.

**Lemma:** If selection node is optimal for \( C \), the schedule \( S \) is as optimal as possible.

The schedule \( S \) is as optimal as possible.

- The register need \( m \) to be minimized.
- Hence we may choose any path in the same zero-deg set.

\[ \{ (z) \} = \{ (z) \} = \{ (z) \} \text{ for } z \in S \text{ scheduled} \]

The same values reside in registers. Namely:

- Corresponding set \( S \) of selection DAG is as optimal as possible.
- For each selection DAG path \( P \), ending in the same zero-deg.

Proof (idea): For each selection DAG path \( P \), ending in the same zero-deg.

**Lemma:** If selection node is optimal for \( C \), the schedule \( S \) is as optimal as possible.

The schedule \( S \) is as optimal as possible.

- The register need \( m \) to be minimized.
- Hence we may choose any path in the same zero-deg set.

\[ \{ (z) \} = \{ (z) \} = \{ (z) \} \text{ for } z \in S \text{ scheduled} \]

The same values reside in registers. Namely:

- Corresponding set \( S \) of selection DAG is as optimal as possible.
- For each selection DAG path \( P \), ending in the same zero-deg.

Proof (idea): For each selection DAG path \( P \), ending in the same zero-deg.

**Lemma:** If selection node is optimal for \( C \), the schedule \( S \) is as optimal as possible.

The schedule \( S \) is as optimal as possible.

- The register need \( m \) to be minimized.
- Hence we may choose any path in the same zero-deg set.

\[ \{ (z) \} = \{ (z) \} = \{ (z) \} \text{ for } z \in S \text{ scheduled} \]

The same values reside in registers. Namely:

- Corresponding set \( S \) of selection DAG is as optimal as possible.
- For each selection DAG path \( P \), ending in the same zero-deg.

Proof (idea): For each selection DAG path \( P \), ending in the same zero-deg.

**Lemma:** If selection node is optimal for \( C \), the schedule \( S \) is as optimal as possible.

The schedule \( S \) is as optimal as possible.

- The register need \( m \) to be minimized.
- Hence we may choose any path in the same zero-deg set.

\[ \{ (z) \} = \{ (z) \} = \{ (z) \} \text{ for } z \in S \text{ scheduled} \]

The same values reside in registers. Namely:

- Corresponding set \( S \) of selection DAG is as optimal as possible.
- For each selection DAG path \( P \), ending in the same zero-deg.

Proof (idea): For each selection DAG path \( P \), ending in the same zero-deg.

**Lemma:** If selection node is optimal for \( C \), the schedule \( S \) is as optimal as possible.

The schedule \( S \) is as optimal as possible.

- The register need \( m \) to be minimized.
- Hence we may choose any path in the same zero-deg set.

\[ \{ (z) \} = \{ (z) \} = \{ (z) \} \text{ for } z \in S \text{ scheduled} \]

The same values reside in registers. Namely:

- Corresponding set \( S \) of selection DAG is as optimal as possible.
- For each selection DAG path \( P \), ending in the same zero-deg.

Proof (idea): For each selection DAG path \( P \), ending in the same zero-deg.

**Lemma:** If selection node is optimal for \( C \), the schedule \( S \) is as optimal as possible.

The schedule \( S \) is as optimal as possible.

- The register need \( m \) to be minimized.
- Hence we may choose any path in the same zero-deg set.

\[ \{ (z) \} = \{ (z) \} = \{ (z) \} \text{ for } z \in S \text{ scheduled} \]

The same values reside in registers. Namely:

- Corresponding set \( S \) of selection DAG is as optimal as possible.
- For each selection DAG path \( P \), ending in the same zero-deg.

Proof (idea): For each selection DAG path \( P \), ending in the same zero-deg.
For pipelined RISC, superscalar, and VLIW architectures, problems with simultaneous space and time minimization for single-issue processors can be pruned.

Extended selection node summaries all schedules of scheduled nodes and ends with time profile \((t',P)\).

Extended selection node: window of the instructions scheduled last for each unit.

Conflicts \((1)\):

Problems:
- For each \(k\), keep time-optimal (instead of space-optimal) schedule.
- For given schedule, \(s\), complete \(I\) in linear time.
- For pipelined RISC, superscalar, and VLIW architectures, problems with simultaneous space and time minimization.

Problems with simultaneous space and time minimization.
Simultaneous space and time optimization with time profiles

**Extensions (1)\[K.'00\]**

Optimalscheduling of spill code \[K.'00\]

Heuristics for pruning of large selection DAGs

**Extensions (2)\[K.'00\]**

Alternative to methods based on integer linear programming.

Algorithm is practical

In practice most basic blocks are small

Time–space profiles (current work)

Time profiles modification of order of construction

Structuring of search space (grid, dependence analysis)

(exploring domain-specific properties)

Dynamic programming

Combination of decision tree

Algorithmic heuristics used

**Summary of the dynamic programming algorithm**

Optimal solution of MRIS

Generalization for the solution of SMRTIS (includes MRIS/MTS/ROMTS)

Simultaneous space and time optimization with time profiles

(4) Heuristic: usually worse results than random scheduling \[K.'00\]

(find) admission of recomputations

Bisector technique: usually worse results than random scheduling \[K.'00\]

Heuristics for pruning of large selection DAGs

**Algorithmic techniques used:**
- topological sorting (list scheduling)
- enumeration with decision tree
- dynamic programming
- structuring of search space as grid, dependence analysis
- (exploring domain-specific properties)
- dynamic programming
- combination of decision tree
- heuristic: usually worse results than random scheduling

In practice most basic blocks are small

Algorithm is practical

Heuristics for pruning of large selection DAGs

**Algorithmic techniques used:**
- topological sorting (list scheduling)
- enumeration with decision tree
- dynamic programming
- structuring of search space as grid, dependence analysis
- (exploring domain-specific properties)
- dynamic programming
- combination of decision tree
- heuristic: usually worse results than random scheduling