Software Pipelining

Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

– Wilhelm/Maurer: Compiler Design, Chapter 12 –

5. Februar 2008

Scheduling Cyclic Code

So far only scheduling of acyclic code:

- List scheduling of basic blocks
- Trace and superblock scheduling of sequences of basic blocks

What about loops? First approach:

1. Unroll loop a number of times, obtaining an enlarged basic block as new body,
2. list schedule this basic block.

Loop Unrolling

```
for  (i=0; i < N; i++) {
  S(i)
}
```

rewritten into

```
for (i=0; i+4 < N; i+=4) {
  S(i);
  S(i+1);
  S(i+2);
  S(i+3)
}
for ( ; i < N; i++) {
  S(i);
}
```

Disadvantages: code growth and no overlapping across back edge

Software Pipelining

generates a schedule that

- overlaps execution of consecutive iterations,
- initiates a new iteration in a fixed *initiation interval*, *II*,
- respects dependences
  - within the same iteration and
  - between several iterations — *loop-carried dependences*,
- avoids resource conflicts.

Advantages:

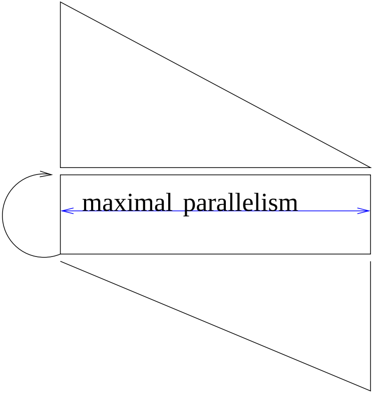
- higher throughput,
- minimal code-size expansion.

Analogy to Hardware Pipelines

Instruction Pipeline: synchronous overlapped execution of consecutive instructions,
issue of new instruction in every cycle if no hazards

Software Pipeline: synchronous overlapping execution of several consecutive iterations,
one iteration issued every *II* cycles.

A Software Pipeline – the Result of our Endeavour



Prolog: initiates the pipeline

Steady state
Kernel

Epilog
finishes remaining iterations

Terminology and Generic Names

Operation: Machine Operation, e.g. **Load**, **Store**, **Add**
names: $a, b, c, \ldots$

Instruction: Set of operations scheduled at the same position,
names: $A, B, C, \ldots$

Latency: Execution time of an operation

Delay: Required distance between the termination of a and the issue of b if $(a \longrightarrow b)$

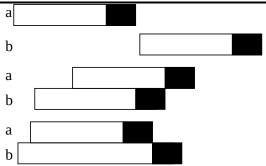
Delays as Functions of Dependence Type

Delay for $(a \rightarrow^{dt} b)$ depends on the latencies of a and b and dt.

Assumptions:

- **write**-cycle is the last,
- **read**-cycles are any cycle but the last,
- in concurrent **reads** and **writes**, **read** reads old content.

	delay	conservative
du:	$latency(a)$	$latency(a)$
ud:	$-1 + latency(a) - latency(b)$	0
dd:	$1 + latency(a) - latency(b)$	$latency(a)$



Schedules

Schedule: Mapping from operations to positions (cycles),
names: $\sigma, \sigma_{flat}, \sigma_{swp}, \ldots$
Note: We are overloading σ with two different meanings:

static: the schedule as produced by the compiler,
dynamic: the dynamic "unrolling" of this schedule.

SW pipelines: loops scheduled as SW pipelines are graphically represented as a matrix:

- columns for original iterations,
- rows for positions in the SW pipeline.

A Simple Loop and Potentially Parallel Execution

```
for i:=1 to n do
1: a[i+1] := a[i]+1;
2: b[i] := a[i+1]/2;
3: c[i] := b[i] + 2;
4: d[i] := c[i}
od
```

```
1: a[i+1] := a[i]+1;
2: b[i] := a[i+1]/2;
3: c[i] := b[i] + 2;
4: d[i] := c[i}
```

```
1: a[i+1] := a[i]+1;
2: b[i] := a[i+1]/2;
3: c[i] := b[i] + 2;
4: d[i] := c[i}
```

```
1: a[i+1] := a[i]+1;
2: b[i] := a[i+1]/2;
3: c[i] := b[i] + 2;
4: d[i] := c[i}
```

Arrows represent dependences between instances of statements in different iterations of the loop.

Inter-iteration Dependencies (Loop Carried Dependencies)

Edges of the DDG are labelled with $(depDist, delay)$

dependence distance: number of iterations between two dependent accesses (0 for intra-iteration dependencies),

delay: minimal number of cycles between the issue of two dependent operations.

for i:=1 to n do

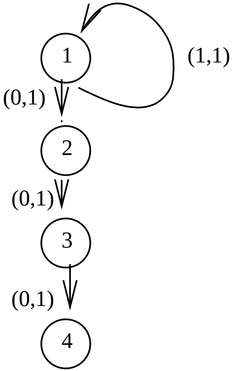
1: a[i+1] := a[i]+1;

2: b[i] := a[i+1]/2;

3: c[i] := b[i] + 2;

4: d[i] := c[i}

od

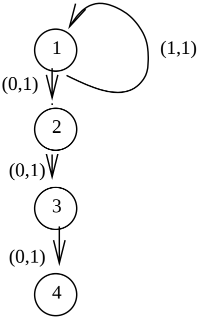


```
for i:=1 to n do
1: a[i+1] := a[i]+1;
2: b[i] := a[i+1]/2;
3: c[i] := b[i] + 2;
4: d[i] := c[i}
od
```

```
1: a[i+1] := a[i]+1;
2: b[i] := a[i+1]/2;
3: c[i] := b[i] + 2;
4: d[i] := c[i}
```

```
1: a[i+1] := a[i]+1;
2: b[i] := a[i+1]/2;
3: c[i] := b[i] + 2;
4: d[i] := c[i}
```

```
1: a[i+1] := a[i]+1;
2: b[i] := a[i+1]/2;
3: c[i] := b[i] + 2;
4: d[i] := c[i}
```



Iterations (depDist); delay

TIME						
I1:	1				Prolog
I2:	2	1			Prolog
I3:	3	2	1		
I4:	4	3	2	1	
I5:	4	3	2		Epilog
I6:	4	3			Epilog
I7:	4				

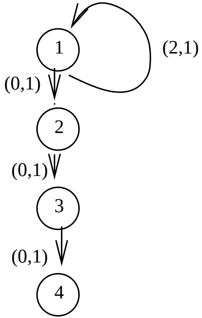
Another Loop

```
for i:=1 to n do
1: a[i+2] := a[i]+1;
2: b[i] := a[i+2]/2;
3: c[i] := b[i] + 2;
4: d[i] := c[i}
od
```

```
1: a[i+2] := a[i]+1;
2: b[i] := a[i+2]/2;
3: c[i] := b[i] + 2;
4: d[i] := c[i}
```

```
1: a[i+2] := a[i]+1;
2: b[i] := a[i+2]/2;
3: c[i] := b[i] + 2;
4: d[i] := c[i}
```

```
1: a[i+2] := a[i]+1;
2: b[i] := a[i+2]/2;
3: c[i] := b[i] + 2;
4: d[i] := c[i}
```



ITERATIONS

depDist

delay

TIME

I1:	1	1							Prolog
I2:	2	2	1	1					
I3:	3	3	2	2	1	1			
I4:	4	4	3	3	2	2	1	1	
I5:			4	4	3	3	2	2	Epilog
I6:					4	4	3	3	
I7:							4	4	

Examples of Dependences

Instructions a and b occur consecutively in the loop body.
i is the loop control variable.

instr. a	instr. b	DDG arc	Dep. type	depDist
m[i+2] := x;	y := m[i];	$a \longrightarrow b$		
y := m[i+3];	m[i] := x;	$a \longrightarrow b$		
m[i] := x;	y := m[i-2];	$a \longrightarrow b$		
y := m[i];	m[i-3] := x;	$a \longrightarrow b$		
y := t;	t := x + i;	$a \longrightarrow b$		
		$b \longrightarrow a$		
t = x + i;	y := t;	$a \longrightarrow b$		
		$b \longrightarrow a$		
y := x + i;	y := t;	$a \longrightarrow b$		
		$b \longrightarrow a$		

Examples of Dependences

Instructions a and b occur consecutively in the loop body.
i is the loop control variable.

instr. a	instr. b	DDG arc	Dep. type	depDist
m[i+2] := x;	y := m[i];	$a \longrightarrow b$	du	2
y := m[i+3];	m[i] := x;	$a \longrightarrow b$	ud	3
m[i] := x;	y := m[i-2];	$a \longrightarrow b$	du	2
y := m[i];	m[i-3] := x;	$a \longrightarrow b$	ud	3
y := t;	t := x + i;	$a \longrightarrow b$	ud	0
		$b \longrightarrow a$	du	1
t = x + i;	y := t;	$a \longrightarrow b$	du	0
		$b \longrightarrow a$	ud	1
y := x + i;	y := t;	$a \longrightarrow b$	dd	0
		$b \longrightarrow a$	dd	1

The General Software-Pipeline Scheduling Problem

Given:

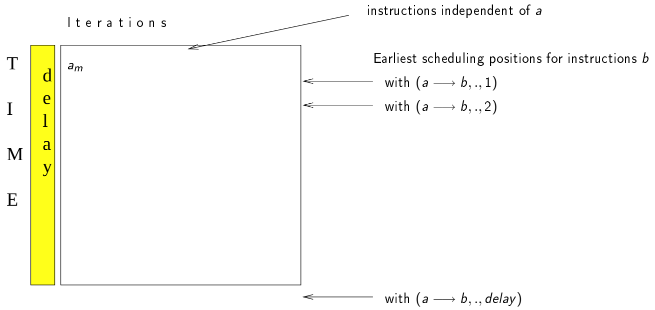
- a loop with body $\mathcal{L}$ and I iterations,
- a p–times parallel architecture.

Wanted: Efficient parallel schedule for $\mathcal{L}^I$ respecting the dependence and resource constraints,
conceptually, $\mathcal{L}^I$ ($\mathcal{L}$ unrolled I times) transformed into $\alpha\mathcal{K}^k\omega$
$\mathcal{K}$, the Kernel, body of a new loop,
α the Prelude,
ω the Postlude.
A new iteration of the new loop is initiated after a fixed number of cycles, called the Initiation Interval, II .

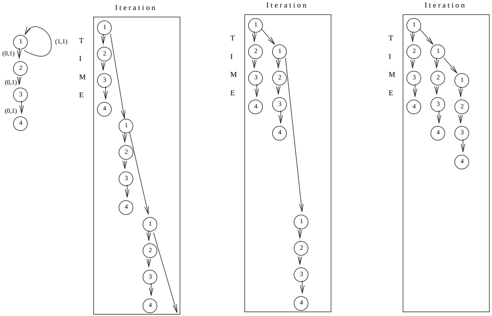
Scheduling Constraints due to Dependences

For a, operation in $\mathcal{L}$, let a_n be the instance of a in the n–th iteration
Constraint for any schedule σ due to $(a \rightarrow b, depDist, delay)$:

$$\boxed{\sigma(b_{m+depDist}) \geq \sigma(a_m) + delay}$$

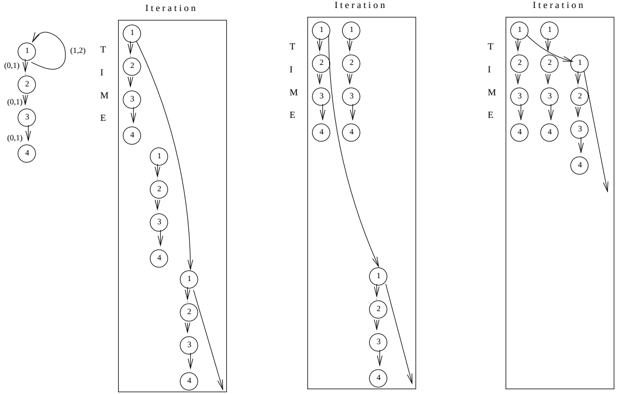


Scheduling due to Dependence Constraints 2



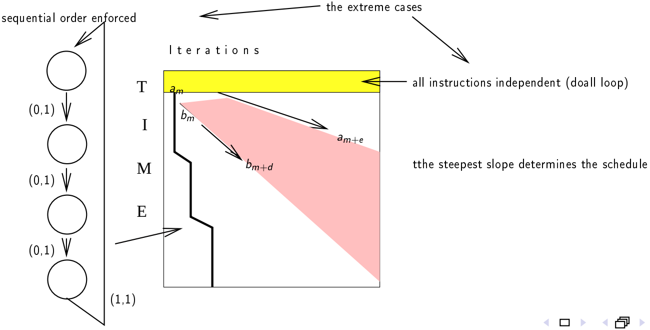
- dependence graph is unrolled, loop-carried dependences instantiated,
- operations are moved up while arrows still go downwards (respecting delays).

The Influence of the Dependence Distance



Implications of the Scheduling Constraints

- bigger value of *delay* $\longrightarrow$ later placement of b in the schedule,
- bigger value of *depDist* $\longrightarrow$ later instance of b concerned $\longrightarrow$ more freedom to schedule,
- best achievable speedup depends on the $slope = delay / depDist$.



Recurrence

Recurrence is the direct or indirect inter-iteration dependence of an operation on itself (a cycle).
Operation without recurrence: all instances can be executed in parallel to each other.
Let $\Theta = \{d_1, \ldots, d_n\}$ be an elementary cycle of the dependence graph on an operation a.

$$\begin{aligned} delay_\Theta &= \textstyle\sum_{i=1}^{n} delay(d_i) \\ depDist_\Theta &= \textstyle\sum_{i=1}^{n} depDist(d_i) \end{aligned}$$

Strongly-Connected Components in the Dependency Graph

The algorithm will consider strongly-connected components of the dependency graph.
Consequences of cyclic dependence:

- any predecessor is also a successor,
- topological sorting has to be modified to schedule operations without all predecessors being already scheduled,
- scheduling an operation defines a deadline for all its successors

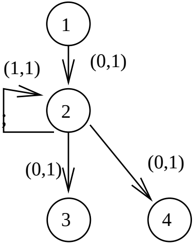
Scheduling Constraints due to Resources

Each instance of an operation has other instances from successive iterations executed II, $2\times$ II, $3\times$ II, $\ldots$ cycles later.
$\Longrightarrow$ Conflicts on a resource in a single iteration must be avoided at times that are multiples of II apart.
$\Longrightarrow$ Total schedule is conflict-free if within a single iteration no resource is used more than once at the same time modulo II.

Identifying a Kernel

Problem: Detect a repeating pattern in a newly made schedule to make it the kernel.

```
for i:=
 1: a[i] := i * i;
 2: b[i] := a[i] * b[i − 1];
 3: c[i] := b[i]/n;
 4: d[i] := b[i] % n;
```



ITERATIONS

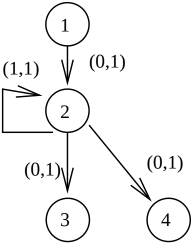
TIME

I1:	1	1	1
I2:	2		
I3:	3,4	2	
I4:		3,4	2
I5:			3,4

Greedy scheduling, i.e. scheduling operation 1 as early as possible, does not form a kernel.

Stages

Schedule for a single iteration of the original loop, $\mathcal{L}$, divided into a sequence of stages of length II.
Number of stages is the stage count, SC.



TIME

ITERATIONS			
I1:	1		
I2:	2	1	
I3:	3,4	2	1
I4:		3,4	2
I5:			3,4

STAGES

$\boxed{1}$
$\boxed{2}$ Stage Count 3
$\boxed{3,4}$

Constraints

1. dependencies and resource constraints
2. all operations from $\mathcal{L}$ occur once in $\mathcal{K}$,
3. width of $\mathcal{K} \leq p$

Goal: $|\mathcal{K}|$ minimal

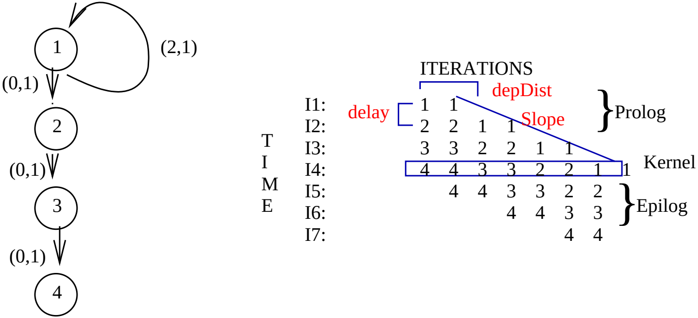
Properties of the Kernel

- $\mathcal{K}$ contains operations of SC consecutive iterations of $\mathcal{L}$
- Initiation Interval, $II = |\mathcal{K}|$, the distance between two consecutive iterations of the new loop,
- $II = |\mathcal{K}|$ is bounded from below by the slope, $delay/depDist$, where the arc controlling the II is annotated with $(depDist, delay)$.

Observation:

- Prelude starts $SC - 1$ iterations,
- Postlude finishes $SC - 1$ iterations,
- all instructions of the original loop occur once in $\mathcal{K}$.

Example (revisited)



Slope is $\mathit{delay}/\mathit{depDist} = 1/2$ of loop-carried dependence.

Approaches

move-then-schedule:
: move code forwards/backwards over loop backedge to improve schedule;
Problem: which operations to move and in which multiplicity?

schedule-then-move:
: find a schedule;
transform code accordingly

- unroll-while-scheduling: Kernel Recognition
complex bookkeeping of scheduling state required
or
- generate and solve set of modulo constraints: Modulo Scheduling

Modulo Scheduling

Treats

- innermost loops
- one iteration of original loop (to start with; later tried with several copies if available parallelism allows)

Basic steps

1. compute lower bound for *II*
2. find schedule
3. generate kernel code
4. generate prelude and postlude code

Lower Bound II_{min}

II_{min} to be determined before scheduling; starting value for iteration. Depends on the Resource Consumption of the operations and on Dependences between the operations

$$
\begin{array}{lccl}
 & II_{min} & \geq & \max\left\{ II_{res}, II_{dep} \right\} \\
\text{where} & II_{res} & = & \min\{|\sigma| \; |\sigma \quad \text{conflict-free schedule}\} \\
\text{and} & II_{dep} & = & \max_{\text{cycles} \quad \Theta} \left\{ \left\lceil \frac{delay_{\Theta}}{depDist_{\Theta}} \right\rceil \right\}
\end{array}
$$

These terms will be explained in the following slides.

Determining II_{res}

Reservation Table for each operation O,
RT_O : cycles $\times$ resources $\longrightarrow \{0,1\}$ defines the resource consumption at each cycle relative to issue time 0.

Resources are

- Source and Result Buses,
- Stages of functional units.

Later, during scheduling used: Schedule Reservation Table, (Modulo Reservation Table, MRT),
records *which resource* is used by *which operation* at a *given time* of a schedule under construction.
When an operation is attempted to be scheduled at time t its reservation table is translated by t **and**ed onto the SRT to check for resource conflicts.
If no conflict, RT_O is **or**'ed onto the current Schedule Reservation Table.

Complexities

Complexity of determining II_{res} depends on the type of resource consumption

Simple Reservation Tables: single resource in a single cycle at issue cycle

Block Reservation Table: single resource for multiple, consecutive cycles starting at issue cycle

Complex Reservation Table: all others

Alternative Reservation Tables: for operations executable on different functional units

Determining the minimal II_{res} is equivalent to binpacking.

A Heuristics

Ignore dependences.

1. Sort operations of loop body in increasing order of number of alternatives
2. Take next operation a from the list; for each resource r: add the number of times a uses r to $usageCount(r)$, choose alternative with lowest (partial) maximal usage count over all resources

Usage count for most heavily used resource constitutes the approximated II_{res}

Determining II_{dep}

Let $\Theta = \{d_1, \ldots, d_n\}$ be an elementary cycle of the dependence graph

$$\begin{aligned} delay_\Theta &= \sum_{i=1}^{n} delay(d_i) \\ depDist_\Theta &= \sum_{i=1}^{n} depDist(d_i) \end{aligned}$$

Property of each schedule σ and each operation a from $\mathcal{L}$

$$\sigma(a_{m+i}) - \sigma(a_m) = II \times i$$

Determining II_{dep} (cont'd)

Resulting Constraint for II_{dep}: $\forall\Theta. \quad depDist_\Theta \times II_{dep} \geq delay_\Theta$
Transformed into:

$$\forall\Theta. \quad II_{dep} \geq \left\lceil \frac{delay_\Theta}{depDist_\Theta} \right\rceil$$

Choose:

$$II_{dep} = \max_\Theta \left\{ \left\lceil \frac{delay_\Theta}{depDist_\Theta} \right\rceil \right\}$$

Computing II_{dep}

Alternatives:

- Enumerate all elementary cycles and determine $\max_{\Theta}\left\{\left\lceil \frac{delay_{\Theta}}{depDist_{\Theta}} \right\rceil\right\}$
- shortest-path algorithm
- minimal cost-to-time ratio cycle problem

Algorithm for the minimal cost-to-time ratio cycle problem

Input: II_{min}

$MinDist[i,j]$ is the smallest legal interval between $\sigma(i)$ and $\sigma(j)$ in the same iteration.

Initialize

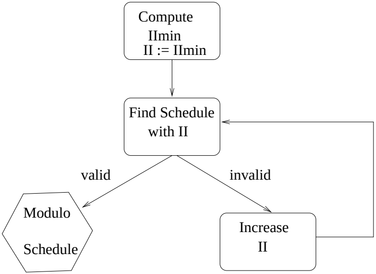
$$MinDist[i,j] = \begin{cases} -\infty \quad \text{if no edge from } i \text{ to } j \\ \max(\max\{d | (a \to b, 0, d)\}, \\ \quad \max\{delay(a) - depDist(e) \times II \mid \quad depDist(e) > 0\}) \end{cases}$$

Iterate the minimal cost-to-time ratio cycle algorithm with increasing II_{min}:

- $MinDist[i,i] > 0$: impossible $\Longrightarrow$ increase II
- $MinDist[i,i] < 0$ for all i: $\Longrightarrow$ slack around every cycle $\Longrightarrow$ decrease II;
- Termination, if at least for one i $MinDist[i,i] = 0$.

Iterative Modulo Scheduling

```
procedure  ModuloSchedule
  II = IImin; found := false;
  (* some heuristic control *)
  (* to enforce termination *)
  do
     if iterativeSchedule(II,...)
     then found := true
     else II := II + 1
  until found
```



Scheduling Priority: Basis is Height-based priority (assumes acyclicity) extended for inter-iteration dependences.

Instruction Scheduling vs. Operation Scheduling

Difference: what is the subject of scheduling?

Instruction Scheduling	Operation Scheduling
instruction to be filled	operation to be scheduled
at each point in time: select max. number of candidate operations that can be scheduled and schedule them	select an operation: schedule it at a legal and profitable position

Modulo scheduling uses operation scheduling, since operations may have to be scheduled several times.

Difference of Modulo Scheduling to Acyclic List Scheduling

- Operation can be unscheduled by backtracking $\Longrightarrow$ operation can be scheduled several times $\Longrightarrow$ modulo scheduling uses operation scheduling.
- Modulo Schedule Reservation Table, $MRT[t \ mod \ II, r]$ records use of resource r at time t $\Longrightarrow$ length of $MRT = II$
- conflict at time $t \Longrightarrow$ conflict at all times $t \pm n \times II$ $\Longrightarrow$ scheduling only for a candidate interval $[MinTime, MaxTime]$ where $MaxTime = MinTime + II - 1$
- List Scheduling always finds a time slot. Procedure `TimeSlot` might not find a legal schedule of the current operation in the interval $[MinTime, MaxTime]$ $\Longrightarrow$ backtracking.

function IterativeSchedule(...)

```
function IterativeSchedule(II, ...) boolean;
   var Op, Estart, MinTime, MaxTime, TimeSlot: int;
begin
schedule(START, 0); (* START pseudooperation *)

while list of non-scheduled operations is not empty and ... do
begin
  Op := highestPriorityOperation;
  Estart := CalculateEarliestStart(Op);
  MinTime := Estart;
  MaxTime := MinTime + II -1;
  TimeSlot := TimeSlot(Op, MinTime, MaxTime);
  Schedule(Op, TimeSlot); (* may unschedule conflicting operations *)
end;
IterativeSchedule := (list of non-scheduled operations empty?)
end;
```

function TimeSlot(...)

```
function TimeSlot(Op, MinT, MaxT: int) int;
   var CurrTime, SchedSlot: int;
begin
CurrTime := minT; SchedSlot :=0;
while SchedSlot = 0 and CurrTime < MaxT do
   if ResourceConflict(Op, CurrTime)
   then CurrTime := CurrTime + 1;
   else SchedSlot := CurrTime
   fi;
if SchedSlot = 0
  then if (NeverScheduled(Op) or MinT > PrevSchedTime[Op]
    then SchedSlot := MinT
    else  SchedSlot := prevSchedTime[Op]+1
    fi;
TimeSlot := SchedSlot
end
```

Height-based Priority and Earliest Start

Priority function: height-based extended to cyclic and inter-iteration dependences.

Uses effective delay.

$$EffDelay(p \rightarrow q) = delay(p \rightarrow q) - II * depDist(p \rightarrow q)$$

$$HeightR(p) = \begin{cases} 0 & \text{if } p \text{ is STOP} \\ \max_{q \in succ(p)}(0, HeightR(q) + delay(p \rightarrow q) & \\ -II * depDist(p \rightarrow q)) & otherwise \end{cases}$$

Warning: Recursion difficult to resolve!

$$Estart(p) = \max_{q \in pred(p)} \begin{cases} 0 & \text{if q is non-scheduled} \\ max(0, SchedTime(q) + & \\ delay(q \rightarrow p) - II * depDist(q \rightarrow p)) & otherwise \end{cases}$$

Candidate Time Slots

Correctness of schedule

- as for resource usage: guaranteed by MRT
- as for dependences: uses Estart,
  earliest time slot for operation to be scheduled

Peculiarity in iterative modulo scheduling:
not all predecessors may have been scheduled or may have remained scheduled
Constraints for scheduling the current operation:

- dependences on predecessors: Estart yields earliest slot
- dependences on successors: conflicts solved by unscheduling

Unscheduling

- slot in [$MinTime$, $MaxTime$] found without resource conflict: unschedule operation with dependence conflict
- no slot in [$MinTime$, $MaxTime$] found without resource conflict: choose time slot + choose operation to unschedule

Increase Exploitable Parallelism

- IF-conversion to eliminate forward branches
- Elimination of pseudo dependences introduced by register allocation
- Rotating registers or variable expansion

Predicated Execution

Motivation

- costs of speculation:
  processor speed is growing
  issue width is growing

  static speculation: more code moved past branches – more compensation code inserted
  dynamic speculation: higher costs of misprediction
- branches limit ILP

Predicated Instructions

Predicated instruction `add r1,r1,1 (P)`
conditionally executed depending on the value in predicate register P

Execution

- Normal instruction fetch
- predicate true: normal execution
- predicate false: instruction nullified – no effect on the state

Predicate-register setting instruction

`pred_`$<comp>$ $P_{out,1}(boolop_1), P_{out,2}(boolop_2), s_1, s_2, (P_{in})$

1. Compares s_1 with s_2 according to $<comp>$,
2. combines the value of P_{in} with the result
   - using boolean operation $boolop_1$ to compute $P_{out,1}$
   - using boolean operation $boolop_2$ to compute $P_{out,2}$

Available boolean operations: Unconditional (U), conditional, NOT, AND, ANDNOT, ...

If-Conversion

Conditionals translated into predicated code

outermost conditional:

if-conv(**if** comp(a,b) **then** e_1 **else** e_2 , true) =

pred_comp q_1(U), q_2(NOT U), a, b;

if-conv(e_1, q_1);

if-conv(e_2, q_2);

where q_1 and q_2 are unused predicates

nested conditionals:

if-conv(**if** comp(a,b) **then** e_1 **else** e_2 , p) =

pred_comp q_1(AND), q_2(ANDNOT), a, b, p;

if-conv(e_1, q_1);

if-conv(e_2, q_2);

where q_1 and q_2 are unused predicates