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

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

rewritten into

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

maximal parallelism
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 ($a \rightarrow b$)
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.

<table>
<thead>
<tr>
<th>delay (\text{du})</th>
<th>(\text{ud})</th>
<th>(\text{dd})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{du: latency}(a))</td>
<td>(-1 + \text{latency}(a) - \text{latency}(b))</td>
<td>(1 + \text{latency}(a) - \text{latency}(b))</td>
</tr>
<tr>
<td>(\text{ud: latency}(a))</td>
<td>(0)</td>
<td>(\text{latency}(a))</td>
</tr>
<tr>
<td>(\text{dd: latency}(a))</td>
<td>(\text{latency}(a))</td>
<td>(\text{latency}(a))</td>
</tr>
</tbody>
</table>

**conservative**
Schedules

**Schedule**: Mapping from operations to positions (cycles),
**names**: \( \sigma, \sigma_{flat}, \sigma_{swp}, \ldots \)
**Note**: We are overloading \( \sigma \) 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

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

(1,1)

(0,1)

2

(0,1)

3

(0,1)

4

1

\begin{itemize}
  \item I1: delay 1
  \item I2: delay 2
  \item I3: delay 3
  \item I4: delay 4
  \item I5: delay 4
  \item I6: delay 4
  \item I7: delay 4
\end{itemize}
**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: $d[i] := c[i]$
end

ITERATIONS
<table>
<thead>
<tr>
<th></th>
<th>delay</th>
<th>depDist</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1:</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>I2:</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>I3:</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>I4:</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>I5:</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>I6:</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>I7:</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

T I M E
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Prolog

Epilog
## Examples of Dependences

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

<table>
<thead>
<tr>
<th>instr. $a$</th>
<th>instr. $b$</th>
<th>DDG arc</th>
<th>Dep. type</th>
<th>depDist</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m[i+2] := x; y := m[i];$</td>
<td>$a \rightarrow b$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y := m[i+3]; m[i] := x;$</td>
<td>$a \rightarrow b$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$m[i] := x; y := m[i-2];$</td>
<td>$a \rightarrow b$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y := m[i]; m[i-3] := x;$</td>
<td>$a \rightarrow b$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y := t; t := x + i;$</td>
<td>$a \rightarrow b$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t = x + i; y := t;$</td>
<td>$b \rightarrow a$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t = x + i; y := t;$</td>
<td>$a \rightarrow b$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b \rightarrow a$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y := x + i; y := t;$</td>
<td>$a \rightarrow b$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b \rightarrow a$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Examples of Dependences

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

<table>
<thead>
<tr>
<th>instr. $a$</th>
<th>instr. $b$</th>
<th>DDG arc</th>
<th>Dep. type</th>
<th>depDist</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m[i+2] := x$; $y := m[i]$;</td>
<td>$a \rightarrow b$</td>
<td>du</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>$y := m[i+3]$; $m[i] := x$;</td>
<td>$a \rightarrow b$</td>
<td>ud</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>$m[i] := x$; $y := m[i-2]$;</td>
<td>$a \rightarrow b$</td>
<td>du</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>$y := m[i]$; $m[i-3] := x$;</td>
<td>$a \rightarrow b$</td>
<td>ud</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>$y := t$; $t := x + i$;</td>
<td>$a \rightarrow b$</td>
<td>ud</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$t = x + i$; $y := t$;</td>
<td>$b \rightarrow a$</td>
<td>du</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$y := x + i$; $y := t$;</td>
<td>$a \rightarrow b$</td>
<td>du</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$b \rightarrow a$</td>
<td>ud</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y := x + i$; $y := t$;</td>
<td>$a \rightarrow b$</td>
<td>dd</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$b \rightarrow a$</td>
<td>dd</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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,

$\alpha$ the Prelude,

$\omega$ 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 $\sigma$ due to $(a \rightarrow b, \text{depDist}, \text{delay})$:

$$\sigma(b_{m+\text{depDist}}) \geq \sigma(a_m) + \text{delay}$$
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 $delay \rightarrow$ later placement of $b$ in the schedule,
- bigger value of $depDist \rightarrow$ later instance of $b$ concerned $\rightarrow$ 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 $a$.

\[
delay_\Theta = \sum_{i=1}^{n} delay(d_i)
\]

\[
depDist_\Theta = \sum_{i=1}^{n} depDist(d_i)
\]
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 $II$, $2 \times II$, $3 \times II$, ... cycles later.

$\implies$ Conflicts on a resource in a single iteration must be avoided at times that are multiples of $II$ apart.

$\implies$ 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;  (0,1)  
2: b[i] := a[i] * b[i - 1];  (1,1)  
3: c[i] := b[i]/n;  
4: d[i] := b[i] % n;  

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*, $SC$.
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 $\mathcal{K}$. 

Software Pipelining

Dependences
Example (revisited)

Slope is $\text{delay}/\text{depDist} = 1/2$ 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 $\parallel$
2. find schedule
3. generate kernel code
4. generate prelude and postlude code
Lower Bound $\ll_{\text{min}}$

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

\[
\ll_{\text{min}} \geq \max \{ \ll_{\text{res}}, \ll_{\text{dep}} \}
\]

where \( \ll_{\text{res}} = \min\{ |\sigma| \mid \sigma \text{ conflict-free schedule} \} \)

and \( \ll_{\text{dep}} = \max_{\text{cycles}} \Theta \left\{ \left\lceil \frac{\text{delay}_{\Theta}}{\text{depDist}_{\Theta}} \right\rceil \right\} \)

These terms will be explained in the following slides.
Determining $II_{res}$

Reservation Table for each operation $O$,
$RT_O : \text{cycles} \times \text{resources} \rightarrow \{0, 1\}$ 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 $t$ anded 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 $\|l_{res}$
Determining $II_{dep}$

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

$$delay_\Theta = \sum_{i=1}^{n} delay(d_i)$$

$$depDist_\Theta = \sum_{i=1}^{n} depDist(d_i)$$

Property of each schedule $\sigma$ 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 $II_{dep}$: $\forall \Theta. \ depDist_\Theta \times II_{dep} \geq delay_\Theta$

Transformed into:

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

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
  $\max_{\Theta} \left\{ \frac{delay_\Theta}{depDist_\Theta} \right\}$
- 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

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

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

- $MinDist[i, i] > 0$: impossible $\implies$ increase $II$
- $MinDist[i, i] < 0$ for all $i$: $\implies$ slack around every cycle $\implies$ 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?

<table>
<thead>
<tr>
<th>Instruction Scheduling</th>
<th>Operation Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction to be filled</td>
<td>operation to be scheduled</td>
</tr>
<tr>
<td>at each point in time: select max. number of candidate</td>
<td>select an operation:</td>
</tr>
<tr>
<td>operations that can be scheduled and schedule them</td>
<td>schedule it at a legal and profitable position</td>
</tr>
</tbody>
</table>

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 → operation can be scheduled several times → modulo scheduling uses operation scheduling.
- Modulo Schedule Reservation Table,
  \( MRT[t \mod \ II, r] \) records use of resource \( r \) at time \( t \) → length of \( MRT = \ II \)
- conflict at time \( t \) → conflict at all times \( t \pm n \times \ II \) → 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 \([MinTime, MaxTime]\) → 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
end
Height-based Priority and Earliest Start

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

\[ \text{EffDelay}(p \rightarrow q) = \text{delay}(p \rightarrow q) - II \ast \text{depDist}(p \rightarrow q) \]

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

Warning: Recursion difficult to resolve!

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

Correctness of schedule

- as for resource usage: guaranteed by MRT
- as for dependences: uses $E_{\text{start}}$, 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: $E_{\text{start}}$ yields earliest slot
- dependences on successors: conflicts solved by unscheduling
Unscheduling

- slot in \([Min\ Time, Max\ Time]\) found without resource conflict: unschedule operation with dependence conflict
- no slot in \([Min\ Time, Max\ Time]\) 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**  \(\text{add } r1, r1, 1\) (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

\[ \text{pred}_< \text{comp}> P_{out,1}(boolop_1), P_{out,2}(boolop_2), s_1, s_2, (P_{in}) \]

1. Compares \( s_1 \) with \( s_2 \) according to \( < \text{comp}> \),
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 \text{comp}(a,b) \text{ then } e_1 \text{ else } e_2 , \text{true}) =
    \text{pred\_comp } q_1(U), q_2(\text{NOT } U), a, b;
    \text{if-conv}(e_1, q_1);
    \text{if-conv}(e_2, q_2);
    \text{where } q_1 \text{ and } q_2 \text{ are unused predicates}

nested conditionals:
if-conv( if \text{comp}(a,b) \text{ then } e_1 \text{ else } e_2 , \text{p}) =
    \text{pred\_comp } q_1(\text{AND}), q_2(\text{ANDNOT}), a, b, p;
    \text{if-conv}(e_1, q_1);
    \text{if-conv}(e_2, q_2);
    \text{where } q_1 \text{ and } q_2 \text{ are unused predicates}