**Software Pipelining**

**Introduction**
- **Software Pipelining (Modulo Scheduling)**
  - Overlap instructions in loops from different iterations
  - Kernel length $I/II$ (initiation interval) ≈ Throughput
- **Goal**: Faster execution of entire loop
  - Better resource utilization,
  - Increase Instruction Level Parallelism, also in the presence of loop-carried dependences

**Definitions**
- Stage count ($SC$) = makespan for 1 iteration in terms of kernel length
- Initiation Interval ($II$)
- Minimum Initiation Interval (MII)
  - Depends on
    - Data dependence cycles (loop carried), $RecMII$
    - Resources (registers, functional units), $ResMII$
  - $MII = max(RecMII, ResMII)$
  - $ResMII = max_{U} ceil(N_{U}/P)$,
    - $N_{U}$ – Number of instructions for resource (functional unit) $U$ in the body
    - $P$ – Number of functional units

**Lower Bound for MII**
- lower bound on $MII$: $max(RecMII, ResMII)$

**Resource constraints:** $RecMII = \left\lfloor \frac{N}{P} \right\rfloor$
- ignoring data dependences
- (multiple reservations: maximize over ratios for each resource type)

**Recurrence constraints:** (ignoring resource constraints due to (simple) cyclic chains of data dependences (DDG):
- accumulate their distances $d$ and their delays $I$
- $RecMII = max_{i} \left\lfloor \frac{\sum_{i} d_i}{\sum_{i} I_i} \right\rfloor$ = maximum overall slope
Calculating the Lower Bound for MII

- \( \max (\text{ResMII}, \text{RecMII}) \)

Determining ResMII is relatively simple.

Determining RecMII:
- Exhaustive enumeration of all simple cycles
- All-pairs shortest path algorithm (e.g., Floyd-Warshall)
- Iterative shortest path algorithm [Zaky’99]
- Compressing distance matrices for a path algebra \( \rightarrow \) transitive closure
- Linear programming

More about this in the survey paper by Allan et al.’95

Modulo Scheduling

- Modulo scheduling:
  - Filling the Modulo Reservation Table, one instruction by another

- Heuristics
  - ASAP, As Soon As Possible
  - ALAP, As Late As Possible
  - HRMS
  - Swing Modulo Scheduling

- Optimally
  - Integer Linear Programming [Eriksson’09]

Modulo Scheduling Heuristics

- Example:
  - Hypernode Reduction Modulo Scheduling (HRMS)

HRMS – Motivation

- Problem with simple ASAP or ALAP heuristics:
  - Some nodes in the DAG are scheduled too early and some too late

- Example:
  - Forward order / ASAP
  - Backward order / ALAP

Hypernode Reduction Approach

- Schedule only nodes that have
  - Only predecessors already scheduled or
  - Only successors already scheduled or
  - None of them, \( \text{but not both predecessors and successors.} \)

- Ensures low register pressure by \( \text{scheduling nodes as close as possible to their relatives.} \)
HRMS – Solution

Two stage algorithm

1. Pre order the nodes of the DAG
   • By using a reduction algorithm
2. Schedule according to the order given in step 1.

Pre-Ordering Step

- Select initial node \( v \)
  - the hypernode \( H \leftarrow \{v\} \)
- Reduce nodes to the hypernode iteratively
  - Remove iteratively edges and nodes in the DAG (= reducing the DAG) and add them to \( H \)
- In each reduction step, append to list of ordered set of nodes
  - Similar to list scheduling / topological sorting, but now in both directions – forward and backward along edges incident to \( H \)

Function \text{pre\_ordering}(G)

Select initial node; \( H \leftarrow \{\text{Initial node}\} \);
List = < Initial node >;
While (Pred(\( H \)) nonempty or Succ(\( H \)) nonempty) do
  \( V' = \text{Pred}(H); \)
  \( V' = \text{Search\_All\_Paths}(V',G); \)
  \( G' = \text{Hypernode\_Reduction}(V',G,H); \)
  \( L' = \text{Sort\_PALA}(G'); \)
  List = Concatenate (List, L’)
  \( V' = \text{Succ}(H); \)
  \( V' = \text{Search\_AllPaths}(V',G); \)
  \( G' = \text{Hypernode\_Reduction}(V',G,h); \)
  \( L' = \text{Sort\_ASAP}(G'); \)
  List = Concatenate (List, L’);
end while
return List;

HRMS – Example

Original dependence graph of one loop iteration:

Start with initial hypernode \( H = \{ A \} \):

\( H \) "eats" successor node C:

List = \{ A, C, G, H, J, I, E, B, F \}  Pred nodes to be scheduled ALAP (D, I, E, B, F)
Succ nodes scheduled ASAP (A, C, G, H, J)

Pre-Ordering with circular dependencies

- Circular dependences from loop carried dependences.
- Solution:
  - Reduce complete path causing cycle to the Hypernode
- How to deal with several connected cycles in DAG?
  - (See details in the paper, skipped).

The Scheduling Step

- Places operations in the order given by the pre-ordering step
- Different strategy depending on neighbors
- If operation has
  - Only predecessors in partial schedule \( \rightarrow \) ASAP
  - Only successors in partial schedule \( \rightarrow \) ALAP
  - Both predecessors and successors in partial schedule
    \( \rightarrow \) Scan from ASAP schedule time towards ALAP time.
    - (If no slot found, \( \text{II}++ \) and reschedule)
The image contains a document discussing various scheduling algorithms, specifically Hypernode Resource Modulo Scheduling (HRMS), and their applications in compiler optimizations and code generation. The document highlights the following points:

**HRMS – Example (cont.)**
- Resulting HRMS schedule:
  - A, B, C, D, E, F, G where for E: ALAP, for others ASAP

**HRMS – Results**
- Perfect Club Benchmark
  - 97.4% of the loops gave optimal II
- Comparison with other algorithms in the paper
  - Works better than Slack scheduling and FRLC scheduling (references in the paper)
  - About same performance as SPIILP (optimal algorithm using Integer Linear Programming, ILP) but lower computational complexity

**HRMS – Conclusion**
- Works well for loops with high register pressure
- Low time complexity
- Tested on large benchmark suite.

**Reference:**

**Modulo Scheduling with Recurrences?**
- If there are recurrences in the dependence graph:
  - (a) find SCC's
  - (b) find cycle in SCC with longest accumulated distance
  - (c) schedule each SCC individually and collapse cycle, creating a single superinstruction
  - (d) apply list scheduling to resulting acyclic graph
- Similarly: collapse if...else... statements
- Disadvantage: separate schedules of SCCs may not fit well together

**Modulo Scheduling and Register Allocation**
- Live ranges may span over multiple iterations → high register need!
- If register allocation fails:
  - (a) try again (with new placement strategy or new scheduling order)
  - (b) spill some live ranges (add spill code) and restart.
- Register need can be optimized:
  - modify order in which the instructions are placed
  - use other placement strategies,
  - e.g. “as late as possible” if already a successor was placed
Register Allocation for Modulo-Scheduled Loops

- We call a live range self-overlapping if it is longer than II
  - Needs > 1 physical register
  - Hard to address properly without HW support

Modulo Variable Expansion

- Unroll the kernel and rename symbolic registers until no self-overlapping live ranges remain

A-priori avoidance of self-overlapping live ranges

- by live range splitting (inserting copy operations before modulo scheduling) [Stotzer Leiss LCTES-2009]

Hardware support: Rotating Register Files

- Iteration Control Pointer points to window in cyclic loop register file, advanced by hardware loop control

Modulo Scheduling for Loops at target level

Example:

```plaintext
s = 0.0;
t = a[0] * b[0];
for (i=1; i<N; i++)
{
s = s + t;
t = a[i] * b[i];
s = s + t;
}
```

Simplified IR:

```plaintext
Loop-carried data dep., Distance +1
```

Given:

VLIW-Processor with 3 units:
- Adder (Latency 1),
- Multiplier/MAC (Latency 3),
- Memory access unit (Latency 3)

Kernel (i=4, ..., N-1):

```plaintext
s = s + t;
```

II = 2

Summary

Software Pipelining / Modulo-Scheduling

- Software Pipelining: Move operations across iteration boundaries
  - Simplest technique: Modulo scheduling
    - Fill modulo reservation table

© Better resource utilization, more ILP, also in the presence of loop-carried data dependences

© In general, higher register need, maybe longer code

- Heuristics e.g. HRMS, Swing Modulo Scheduling, ...
- Optimal methods e.g. Integer Linear Programming
  - (Problem is NP-complete like acyclic scheduling)

- Self-overlapping live ranges need special treatment
- Loop unrolling can leverage additional optimization potential
- Up to now, only at target code level, hardly integrated (sometimes with register allocation)