



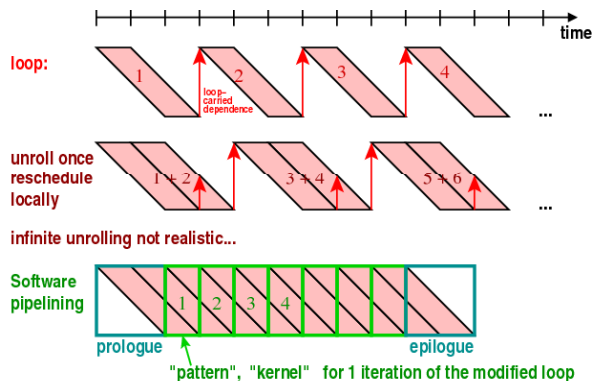
# Software Pipelining

**Literature:**

- C. Kessler, "Compiling for VLIW DSPs", 2009, Section 7.2 (handed out)
- ALSU2e Section 10.5
- Muchnick Section 17.4
- Allan et al.: Software Pipelining. ACM Computing Surveys 27(3), 1995

Christoph Kessler, IDA, Linköpings universitet, 2014.

## Software Pipelining of Loops (1)



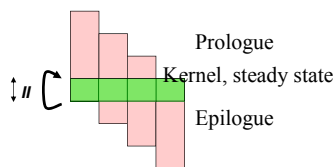
C. Kessler, IDA, Linköpings universitet.

2

## Introduction



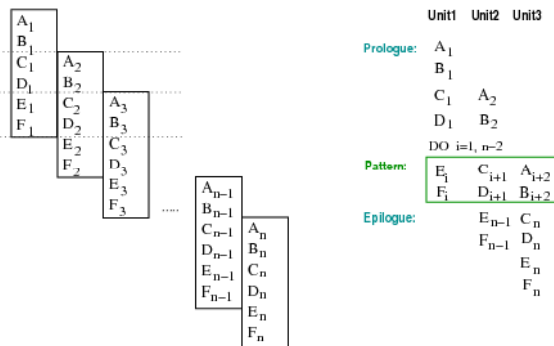
- **Software Pipelining** (Modulo Scheduling)
  - Overlap instructions in loops from different iterations
    - ▶ Kernel length  $||$  (initiation interval)  $\sim$  Throughput
  - Goal: Faster execution of entire loop
    - ▶ Better resource utilization,
    - ▶ Increase Instruction Level Parallelism,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



C. Kessler, IDA, Linköpings universitet.

3

## Software Pipelining of Loops (2)



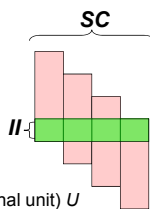
C. Kessler, IDA, Linköpings universitet.

4

## 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),  $ResMII$
  - $MII = \max(ResMII, RecMII)$
  - $ResMII = \max_U \lceil N_U / P \rceil$ ,
    - ▶  $N_U$  – Number of instructions for resource (functional unit)  $U$  in the body
    - ▶  $P$  – Number of functional units



C. Kessler, IDA, Linköpings universitet.

5

## Lower Bound for MII



lower bound on  $MII$ :  $\max(ResMII, RecMII)$

Resource constraints:  $ResMII = \lceil \frac{N}{P} \rceil$

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  $l$

$$RecMII = \max_{C \text{ cycle}} \left\{ \frac{\sum_{e \in C} l_e}{\sum_{e \in C} d_e} \right\} = \text{maximum overall slope}$$

C. Kessler, IDA, Linköpings universitet.

6

## Calculating the Lower Bound for MII



■  $\max ( ResMII, RecMII )$

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'89]  
composing distance matrices for a path algebra  $\rightarrow$  trans. 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**

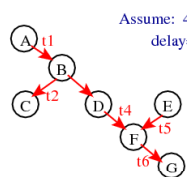
$\rightarrow$  example

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

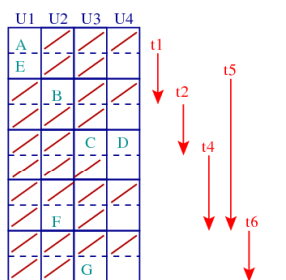
■ **Optimally**

- Integer Linear Programming [Eriksson'09]

## Modulo Scheduling



Assume: 4 units, fully pipelined  
delay=2 for all instructions



No dependence cycles

$ResMII = \text{ceil}(7/4) = 2$

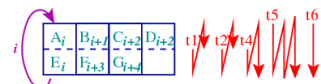
Begin with  $II = ResMII = 2$

Apply some local scheduling heuristic  
e.g.: list scheduling (A B C D E F G)

Apply some placement heuristic  
e.g.: as early as possible

Mark occupied slots in all iterations...

If not possible, increase II and try again...



## Modulo Scheduling Heuristics



■ **Example:**  
**Hypernode Reduction Modulo Scheduling (HRMS)**

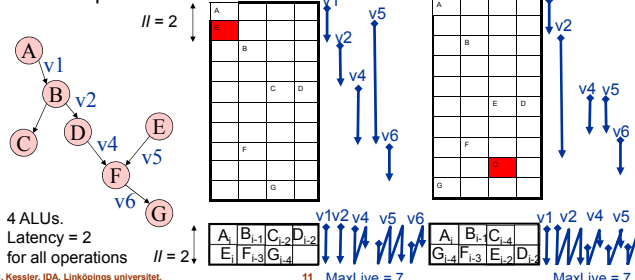
## HRMS – Motivation



■ Problem with simple ASAP or ALAP heuristics:

Some nodes in the DAG are scheduled too early and some too late

■ Example:



## Hypernode Reduction Approach



■ Schedule only nodes that have

- Only predecessors already scheduled *or*
- Only successors already scheduled *or*
- None of them,

*but not both predecessors and successors.*

■ Ensures low register pressure by *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 pre\_ordering( G )



Select initial node;  $H \leftarrow \{\text{Initial node}\};$

List = < Initial node >;

**While** (Pred( $H$ ) nonempty or Succ( $H$ ) nonempty) **do**

$V' = \text{Pred}(H);$

$V' = \text{Search\_All\_Paths}(V', G);$

$G' = \text{Hypernode\_Reduction}(V', G, H);$

$L' = \text{Sort\_PALA}(G');$  // ALAP with inverted order

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,  $L'$  );

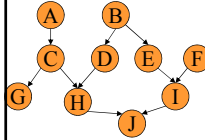
**end while**

**return** List;

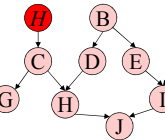
## HRMS – Example



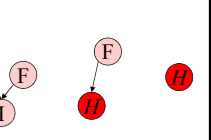
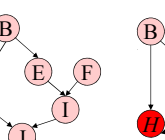
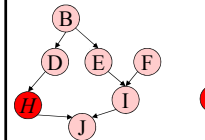
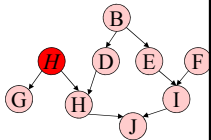
Original dependence graph of one loop iteration:



Start with initial hypernode  $H = \{A\}$ :



$H$  "eats" successor node C:



List = { A, C, G, H, D, 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



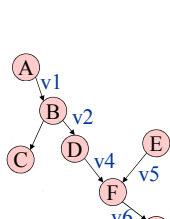
- Places operations in the order given by the pre-ordering step
- Different strategy depending on *neighbors*
- If operation has
  - Only predecessors in partial schedule -> **ASAP**
  - Only successors in partial schedule -> **ALAP**
  - Both predecessors and successors in partial schedule -> Scan from ASAP schedule time towards ALAP time.
    - ▶ (If no slot found,  $//++$  and reschedule)

## HRMS – Example (cont.)

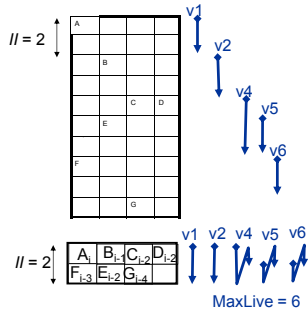


### Resulting HRMS schedule:

- A, B, C, D, E, F, G where for E: ALAP, for others ASAP



4 ALUs.  
Latency = 2  
for all operations



MaxLive = 6

C. Kessler, IDA, Linköpings universitet.

19

## HRMS – Results



### Perfect Club Benchmark

- 97.4 % of the loops gave optimal  $||$

### Comparison with other algorithms in the paper

- Works better than Slack scheduling and FRLC scheduling (references in the paper)
- About same performance as SPILP (optimal algorithm using Integer Linear Programming, ILP) but lower computational complexity

C. Kessler, IDA, Linköpings universitet.

20

## HRMS – Conclusion



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

### Reference:

J. Llosa, M. Valero, E. Ayguadé and A. Gonzáles:  
Hypernode Resource Modulo Scheduling.  
Proc. 28th ACM/IEEE Int. Symposium on Microarchitecture,  
pp. 350-360, IEEE Computer Society Press, 1995

C. Kessler, IDA, Linköpings universitet.

21

## Modulo Scheduling with Recurrences?



- If there are recurrences in the dependence graph:

[Lam'88]

- find SCC's
- find cycle in SCC with longest accumulated distance
- schedule each SCC individually and collapse cycle, creating a single superinstruction
- apply list scheduling to resulting acyclic graph

- similarly: collapse **if..then..else** statements
- disadvantage: separate schedules of SCCs may not fit well together

C. Kessler, IDA, Linköpings universitet.

22

## Modulo Scheduling and Register Allocation



Live ranges may span over multiple iterations  $\rightarrow$  high register need!

If register allocation fails:

- try again (with new placement strategy or new scheduling order)
- 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

C. Kessler, IDA, Linköpings universitet.

23

## Modulo Scheduling and Register Allocation



### Software Pipelining tends to increase register pressure

- Live ranges may span over several iterations

### May lead to (more) register spill

- Introduces new problems
  - Should we spill or increase  $||$  ?
  - How to choose variables to spill ?
  - Integrated software pipelining (later)

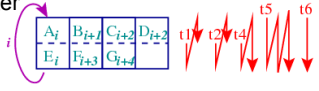
C. Kessler, IDA, Linköpings universitet.

24

## Register Allocation for Modulo-Scheduled Loops



- We call a live range *self-overlapping* if it is longer than  $ll$ 
  - 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



C. Kessler, IDA, Linköping universitet.

25

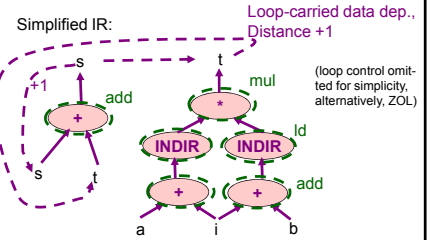
## Modulo Scheduling for Loops at target level



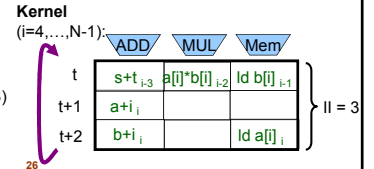
- Example:

```

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;
    
```



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



C. Kessler, IDA, Linköping universitet.

26

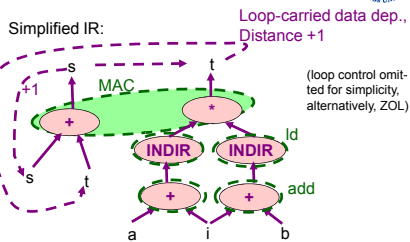
## Modulo Scheduling for Loops can benefit from integration with instruction selection



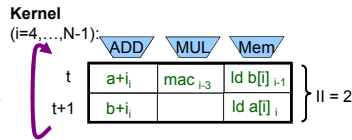
- Example:

```

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;
    
```



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



C. Kessler, IDA, Linköping universitet.

27

## 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)

C. Kessler, IDA, Linköping universitet.

28