Software Pipelining

Introduction

- **Software Pipelining** (Modulo Scheduling)
  - Overlap instructions in loops from different iterations
  - Kernel length \( II \) (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), \( \text{RecMII} \)
    - Resources (registers, functional units), \( \text{ResMII} \)
  - \( \text{MII} = \max (\text{RecMII}, \text{ResMII}) \)
  - \( \text{ResMII} = \left\lceil \frac{\text{NU}}{\text{P}} \right\rceil \)
  - \( \text{NU} \) – Number of instructions for resource (functional unit) \( U \) in the body
  - \( P \) – Number of functional units
- Modulo scheduling;
  - Filling the Modulo Reservation Table, one instruction by another
    - ASAP, As Soon As Possible
    - ALAP, As Late As Possible

Software Pipelining of Loops (1)

- Unroll once reschedule locally
- Infinite unrolling not realistic...

Software Pipelining of Loops (2)

- Kernel length \( II \) (initiation interval) – Throughput
  - Goal: Faster execution of entire loop
  - Better resource utilization,
  - Increase Instruction Level Parallelism, also in the presence of loop-carried dependences

 Software Pipelining of Loops (3)

- Modulo Scheduling
  - Assume: 4 units, fully pipelined
  - \( \text{latency}=2 \) (delay=1) for all instructions
  - No dependence cycles
  - \( \text{ResMII} = \frac{\text{NU}}{\text{P}} = 2 \)

- Begin with \( II = \text{ResMII} = 2 \)

- 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...

- \( \text{SC} \) = makespan for 1 iteration in terms of kernel length

- Literature:
  - C. Kessler, “Compiling for VLIW DSPs”, 2009, Section 7.2 (handed out)
  - ALSUZe Section 10.5
  - Muchnick Section 17.4
  - Allan et al.: Software Pipelining. ACM Computing Surveys 27(3), 1995
Software Pipelining of Loops (4)
Modulo Scheduling
Assume: 4 units, fully pipelined latency=2 (delay=1) for all instructions
No dependence cycles
ResMII = 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...
latency=2 (delay=1) for all instructions
Assume: 4 units,
fully pipelined
Software Pipelining of Loops (5)
Modulo Scheduling
Assume: 4 units, fully pipelined latency=2 (delay=1) for all instructions
No dependence cycles
ResMII = 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...
latency=2 (delay=1) for all instructions
Assume: 4 units,
fully pipelined
Modulo Scheduling Heuristics
See separate slide set on Hypernode Reduction Modulo Scheduling

Modulo Scheduling for Loops
at target level
Example:
Simplified IR:
Kernel (i=4,...,N-1):
Given:
VLIW-Processor with 3 units:
- Adder (Latency 1),
- Multiplier/MAC (Latency 3),
- Memory access unit (Latency 3)

Modulo Scheduling for Loops can benefit from integration with instruction selection
Example:
Simplified IR:
Kernel (i=4,...,N-1):
Given:
VLIW-Processor with 3 units:
- Adder (Latency 1),
- Multiplier/MAC (Latency 3),
- Memory access unit (Latency 3)
Register Allocation for Modulo-Scheduled Loops

- Software Pipelining tends to increase register pressure
- We call a live range self-overlapping if it is longer than $\delta$
  - 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

Summary

Software Pipelining / Modulo-Scheduling

- Move operations across iteration boundaries
  - Simplest technique: 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
- Self-overlapping live ranges need special treatment
- Up to now, only at target code level, hardly integrated (sometimes with register allocation)