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), $RecMII$
    - Resources (registers, functional units), $ResMII$
  - $MII = \max(RecMII, ResMII)$
  - $RecMII = \max_i \lceil \frac{N_i}{P} \rceil$
    - $N_i$ – 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)

- Loop:
- Unroll once
- Reschedule locally
- Infinite unrolling not realistic...

Module Scheduling

- SC
- Assume: 4 units, fully pipelined delay = 2 for all instructions
- No dependence cycles
- $RecMII = ceil(7/4) = 2$
- BEGIN with $II = RecMII = 2$
- Apply some local scheduling heuristics:
eg. list scheduling (A B C D E F G)
- Apply some placement heuristics:
eg. as early as possible
- Mask occupied slots in all iterations...
- If not possible, increase $II$ and try again...

- 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
Modulo Scheduling Heuristics

- See separate slide set on Hypernode Reduction Modulo Scheduling

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;
```

II = 3

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

Simplified IR:
```
t s
MAC
ADD
```

Kernel (i = 4, ..., N - 1):
```
MUL
Mem
ADD
INDIR
ia b
+ INDIR *
```

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 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

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)