EECS 583 – Class 15 Modulo Scheduling Reloaded

University of Michigan

November 6, 2019

Announcements + Reading Material

- ✤ Exam Pushed back to Friday Nov 15, 10:40am 12:30pm
 - » No additional classrooms available on Wednes, so forced to reschedule
 - » Chrysler Auditorium (rm 220)
 - » Let me know ASAP if you have a conflict!!
- Today's class reading
 - » "Code Generation Schema for Modulo Scheduled Loops", B. Rau, M. Schlansker, and P. Tirumalai, MICRO-25, Dec. 1992.
- Next class reading
 - » "Register Allocation and Spilling Via Graph Coloring," G. Chaitin, Proc. 1982 SIGPLAN Symposium on Compiler Construction, 1982.

From Last Time: Dependences in a Loop

- Need worry about 2 kinds
 - » Intra-iteration
 - » Inter-iteration
- Delay
 - Minimum time interval between the start of operations
 - » Operation read/write times
- Distance
 - Number of iterations separating the 2 operations involved
 - Distance of 0 means intraiteration
- Recurrence manifests itself as a circuit in the dependence graph

<1.1> <1.2 <1.2> <1.0> 3 <1.02

Edges annotated with tuple <delay, distance>

From Last Time: Dynamic Single Assignment (DSA) Form

Impossible to overlap iterations because each iteration writes to the same register. So, we'll have to remove the anti and output dependences.

Virtual rotating registers

- * Each register is an infinite push down array (Expanded virtual reg or EVR)
- * Write to top element, but can reference any element
- * Remap operation slides everything down \rightarrow r[n] changes to r[n+1]

A program is in DSA form if the same virtual register (EVR element) is never assigned to more than 1x on any dynamic execution path



Loop Dependence Example

1: r3[-1] = load(r1[0]) 2: r4[-1] = r3[-1] * 26 3: store (r2[0], r4[-1]) 4: r1[-1] = r1[0] + 4 5: r2[-1] = r2[0] + 4 6: p1[-1] = cmpp (r1[-1] < r9) remap r1, r2, r3, r4, p1 7: brct p1[-1] Loop

In DSA form, there are no inter-iteration anti or output dependences!



<delay, distance>

Class Problem

Latencies: Id = 2, st = 1, add = 1, cmpp = 1, br = 1

```
1: r1[-1] = load(r2[0])

2: r3[-1] = r1[1] - r1[2]

3: store (r3[-1], r2[0])

4: r2[-1] = r2[0] + 4

5: p1[-1] = cmpp (r2[-1] < 100)

remap r1, r2, r3

6: brct p1[-1] Loop
```

Draw the dependence graph showing both intra and inter iteration dependences

Minimum Initiation Interval (MII)

- Remember, II = number of cycles between the start of successive iterations
- Modulo scheduling requires a candidate II be selected before scheduling is attempted
 - » Try candidate II, see if it works
 - » If not, increase by 1, try again repeating until successful
- MII is a lower bound on the II
 - » MII = Max(ResMII, RecMII)
 - » ResMII = resource constrained MII
 - Resource usage requirements of 1 iteration
 - » RecMII = recurrence constrained MII
 - Latency of the circuits in the dependence graph

ResMII

Concept: If there were no dependences between the operations, what is the the shortest possible schedule?

Simple resource model

A processor has a set of resources R. For each resource r in R there is count(r) specifying the number of identical copies

ResMII = MAX (uses(r) / count(r))for all r in R

uses(r) = number of times the resource is used in 1 iteration

In reality its more complex than this because operations can have multiple alternatives (different choices for resources it could be assigned to), but we will ignore this for now

ResMII Example

resources: 4 issue, 2 alu, 1 mem, 1 br latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

```
1: r3 = load(r1)

2: r4 = r3 * 26

3: store (r2, r4)

4: r1 = r1 + 4

5: r2 = r2 + 4

6: p1 = cmpp (r1 < r9)

7: brct p1 Loop
```

ALU: used by 2, 4, 5, 6 \rightarrow 4 ops / 2 units = 2 Mem: used by 1, 3 \rightarrow 2 ops / 1 unit = 2 Br: used by 7 \rightarrow 1 op / 1 unit = 1

ResMII = MAX(2,2,1) = 2

RecMII

Approach: Enumerate all irredundant elementary circuits in the dependence graph

RecMII = MAX (delay(c) / distance(c))for all c in C

delay(c) = total latency in dependence cycle c (sum of delays)
distance(c) = total iteration distance of cycle c (sum of distances)



RecMII Example





 $4 \rightarrow 4: 1 / 1 = 1$ $5 \rightarrow 5: 1 / 1 = 1$ $4 \rightarrow 1 \rightarrow 4: 1 / 1 = 1$ $5 \rightarrow 3 \rightarrow 5: 1 / 1 = 1$

RecMII = MAX(1,1,1,1) = 1

Then,

MII = MAX(ResMII, RecMII)MII = MAX(2,1) = 2

<delay, distance>

Class Problem

Latencies: ld = 2, st = 1, add = 1, cmpp = 1, br = 1 Resources: 1 ALU, 1 MEM, 1 BR

```
1: r1[-1] = load(r2[0])

2: r3[-1] = r1[1] - r1[2]

3: store (r3[-1], r2[0])

4: r2[-1] = r2[0] + 4

5: p1[-1] = cmpp (r2[-1] < 100)

remap r1, r2, r3

6: brct p1[-1] Loop
```

Calculate RecMII, ResMII, and MII

Modulo Scheduling Process

- Use list scheduling but we need a few twists
 - » II is predetermined starts at MII, then is incremented
 - » Cyclic dependences complicate matters
 - Estart/Priority/etc.
 - Consumer scheduled before producer is considered
 - There is a window where something can be scheduled!
 - » Guarantee the repeating pattern
- ✤ 2 constraints enforced on the schedule
 - » Each iteration begin exactly II cycles after the previous one
 - » Each time an operation is scheduled in 1 iteration, it is tentatively scheduled in subsequent iterations at intervals of II
 - MRT used for this

Priority Function

Height-based priority worked well for acyclic scheduling, makes sense that it will work for loops as well



EffDelay(X,Y) = Delay(X,Y) - II*Distance(X,Y)

Calculating Height

- 1. Insert pseudo edges from all nodes to branch with latency = 0, distance = 0 (dotted edges)
- 2. Compute II, For this example assume II = 2
- 3. Height R(4) =
- 4. Height R(3) =

5. HeightR(2) =

(1)(0,0) (3,0)(2,0)(2,0)(2,0)(3)(0,0)(4)(1,1)

6. Height R(1)

The Scheduling Window

With cyclic scheduling, not all the predecessors may be scheduled, so a more flexible <u>earliest schedule time</u> is:

$$E(Y) = MAX$$

for all X = pred(Y)
$$MAX (0, SchedTime(X) + EffDelay(X,Y)),$$

otherwise

where EffDelay(X,Y) = Delay(X,Y) - II*Distance(X,Y)

Every II cycles a new loop iteration will be initialized, thus every II cycles the pattern will repeat. Thus, you only have to look in a window of size II, if the operation cannot be scheduled there, then it cannot be scheduled.

Latest schedule time(Y) = L(Y) = E(Y) + II - 1

Loop Prolog and Epilog



Only the kernel involves executing full width of operations

Prolog and epilog execute a subset (ramp-up and ramp-down)

Separate Code for Prolog and Epilog



Generate special code before the loop (preheader) to fill the pipe and special code after the loop to drain the pipe.

Peel off II-1 iterations for the prolog. Complete II-1 iterations in epilog

Removing Prolog/Epilog



Execute loop kernel on every iteration, but for prolog and epilog selectively disable the appropriate operations to fill/drain the pipeline

Kernel-only Code Using Rotating Predicates



Modulo Scheduling Architectural Support

- Loop requiring N iterations
 - » Will take N + (S 1) where S is the number of stages
- ✤ 2 special registers created
 - » LC: loop counter (holds N)
 - » ESC: epilog stage counter (holds S)
- Software pipeline branch operations
 - » Initialize LC = N, ESC = S in loop preheader
 - » All rotating predicates are cleared
 - » SWP-BR
 - While LC > 0, decrement LC and RRB, P[0] = 1, branch to top of loop
 - This occurs for prolog and kernel
 - If LC = 0, then while ESC > 0, decrement RRB and write a 0 into P[0], and branch to the top of the loop
 - This occurs for the epilog

Execution History With LC/ESC

LC = 3, ESC = 3 /* Remember 0 relative!! */

Clear all rotating predicates

P[0] = 1

A if P[0]; B if P[1]; C if P[2]; D if P[3]; P[0] = BRF.B.B.F;

LC	ESC	P[0]	P[1]	P [2]	P[3]				
3	3	1	0	0	0	А			
2	3	1	1	0	0	А	В		
1	3	1	1	1	0	А	В	С	
0	3	1	1	1	1	А	В	С	D
0	2	0	1	1	1	-	В	С	D
0	1	0	0	1	1	-	-	С	D
0	0	0	0	0	1	-	-	-	D

4 iterations, 4 stages, II = 1, Note 4 + 4 - 1 iterations of kernel executed

Modulo Scheduling Example

resources: 4 issue, 2 alu, 1 mem, 1 br latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

for (j=0; j<100; j++) b[j] = a[j] * 26 Step1: Compute to loop into form that uses LC

LC = 99



resources: 4 issue, 2 alu, 1 mem, 1 br latencies: add=1, mpy=3, ld = 2, st = 1, br = 1





resources: 4 issue, 2 alu, 1 mem, 1 br latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

Step3: Draw dependence graph Calculate MII





Step 4 – Calculate priorities (MAX height to pseudo stop node)

Iter1	Iter2
1: H = 5	1: H = 5
2: H = 3	2: H = 3
3: $H = 0$	3: $H = 0$
4: H = 0	4: H = 4
5: H = 0	5: H = 0
7: $H = 0$	7: H = 0

resources: 4 issue, 2 alu, 1 mem, 1 br latencies: add=1, mpy=3, ld = 2, st = 1, br = 1 Schedule brlc at time II - 1



Step6: Schedule the highest priority op

Op1: E = 0, L = 1Place at time 0 (0 % 2)

Loop:



Step7: Schedule the highest priority op

Op4: E = 0, L = 1 Place at time 0 (0 % 2)

Loop:



Step8: Schedule the highest priority op

Op2: E = 2, L = 3 Place at time 2 (2 % 2)

Loop:

Step9: Schedule the highest priority op

Op3: E = 5, L = 6Place at time 5 (5 % 2)

LC = 99

Loop:

Step10: Schedule the highest priority op

Op5: E = 5, L = 6Place at time 5 (5 % 2)

LC = 99

Loop:

Schedule Rolled Schedule 4 1 0 1 1: r3[-1] = load(r1[0])2 2 2 4 1 2: r4[-1] = r3[-1] * 26 0 3 3: store (r2[0], r4[-1]) 1 3 7 5 4 4: r1[-1] = r1[0] + 43 5 5 5: r2[-1] = r2[0] + 4remap r1, r2, r3, r4 6 7: brlc Loop mem br alu1 alu0 Χ Χ Χ 0 **MRT** Χ Χ Χ 1

Step11: calculate ESC, SC = ceiling(max unrolled sched length / ii) unrolled sched time of branch = rolled sched time of br + (ii*esc)

Finishing touches - Sort ops, initialize ESC, insert BRF and staging predicate, initialize staging predicate outside loop

LC = 99ESC = 2 p1[0] = 1

Loop:

1: r3[-1] = load(r1[0]) if p1[0] 2: r4[-1] = r3[-1] * 26 if p1[1] 4: r1[-1] = r1[0] + 4 if p1[0] 3: store (r2[0], r4[-1]) if p1[2] 5: r2[-1] = r2[0] + 4 if p1[2] 7: brlc Loop if p1[2] Staging predicate, each successive stage increment the index of the staging predicate by 1, stage 1 gets px[0]

> Unrolled Schedule

Example – Dynamic Execution of the Code

LC = 99	time: ops executed
ESC = 2	0: 1, 4
p1[0] = 1	1:
	2: 1,2,4
Loop: 1: $r3[-1] = load(r1[0])$ if $p1[0]$	3:
2: $r4[-1] = r3[-1] * 26$ if $p1[1]$	4: 1,2,4
4: $r1[-1] = r1[0] + 4$ if $p1[0]$	5: 3,5,7
3: store $(r2[0], r4[-1])$ if $p1[2]$ 5: $r2[-1] = r2[0] + 4$ if $p1[2]$	6: 1,2,4
7: brlc Loop if $p1[2]$	7: 3,5,7
Total time = II(num_iteration + num_stages - 1) = $2(100 + 3 - 1) = 204$ cycles	198: 1,2,4 199: 3,5,7 200: 2 <u>201: 3,5,7</u> 202: - 203 3,5,7

Homework Problem

latencies: add=1, mpy=3, ld=2, st=1, br=1

for
$$(j=0; j<100; j++)$$

b[j] = a[j] * 26

LC = 99

1

Loop:	1: $r3 = load(r1)$
	2: r4 = r3 * 26
	3: store (r2, r4)
	4: $r1 = r1 + 4$
	5: $r^2 = r^2 + 4$
	7: brlc Loop

How many resources of each type are required to achieve an II=1 schedule?

If the resources are non-pipelined, how many resources of each type are required to achieve II=1

Assuming pipelined resources, generate the II=1 modulo schedule.

What if We Don't Have Hardware Support for Modulo Scheduling?

- No predicates
 - » Predicates enable kernel-only code by selectively enabling/disabling operations to create prolog/epilog
 - » Now must create explicit prolog/epilog code segments
- No rotating registers
 - » Register names not automatically changed each iteration
 - » Must unroll the body of the software pipeline, explicitly rename
 - Consider each register lifetime i in the loop
 - Kmin = min unroll factor = MAXi (ceiling((Endi Starti) / II))
 - Create Kmin static names to handle maximum register lifetime
 - » Apply modulo variable expansion