CS597d—Modulo Scheduling

John W. Sias

IMPACT Compiler Research Group

Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

25 October 1999

Outline

- Loop performance in instruction-level parallel processors
- Loop unrolling and early attempts at modulo scheduling
- Iterative modulo scheduling
- Handling control flow–exit transformations and predication
- Code generation schema w/ IA-64 support
- Unrolling-based optimizations for modulo scheduling

Code example: straightforward matrix multiplication

```
for (i=0;i<n;i++)
    for (j=0;j<n;j++)
        for (k=0;k<n;k++)
            C[i,j] = C[i,j] +
                     A[i,k] * B[k,j]
```

- Assume that A, B, and C are:
  - disjoint
  - of dimension $(n \times n)$
- Focus on inner loop for simplicity

```
    mov  r1 = label_n
    ld_i r2 = (r1)       ; r2 = n
    lsl  r3 = r2, 2      ; r3 = 4*n
    mul  r4 = r3, r2
    mov  r5 = label_C    ; addr.  C[i,j]
    mov  r6 = label_A    ; addr.  A[i,0]
    mov  r7 = label_B    ; addr.  B[0,j]
    add  r4 = r4, r6     ; addr.  A[n,0]
    add  r8 = r7, r3     ; addr.  B[0,n]
L1: ld_f f1 = (r5)       ; f1 = C[i,j]
    mov  r9 = 0
    mov  r10 = 0
-------------------------------------------
L2: ld_f f2 = (r6+r9)    ; f2 = A[i,k]
    ld_f f3 = (r2+r10)   ; f3 = B[k,j]
    fmul f4 = f2, f3
    fadd f1 = f1, f4     ; C[i,j]+=A[i,k]*B[k,j]
    add  r9 = r9, 4
    add  r10 = r10, r3
    blt  r9, r3, L2      ; end k
-------------------------------------------
    st_f (r5) = f1
    add  r5 = r5, 4
    add  r7 = r7, 4
    blt  r7, r8, L1      ; end j
    sub  r7 = r7, r3
    add  r6 = r6, r3
    blt  r6, r4, L1      ; end i
```

Scheduling the code example

- Schedule inner loop
- Eight-issue inorder processor
- Latencies

  `ld` 2 cy
  `fmul` 4 cy
  `fadd` 1 cy
  `add` 1 cy

- Functional units (fully pipelined)

  `mem` 2
  `falu` 2
  `ialu` 4
  `br` 1

```
1:  ld_f f2 = (r6+r9)
2:  ld_f f3 = (r2+r10)
3:  fmul f4 = f2, f3
4:  fadd f1 = f1, f4
5:  add  r9 = r9, 4
6:  add  r10 = r10, r3
7:  blt  r9, r3, L2
```

1	1 ld	2 ld
2	5 add	6 add
3	3 fmul	
4		
5		
6		
7	4 fadd	7 blt

7 cycles per inner loop iteration

Loop unrolling

1	1.1 ld	1.2 ld	2.5 add	2.6 add	3.5 add	3.6 add
2	2.1 ld(s)	2.2 ld(s)	4.5 add	4.6 add	1.5 add	1.6 add
3	3.1 ld(s)	3.2 ld(s)	1.3 fmul			
4	4.1 ld(s)	4.2 ld(s)	2.3 fmul(s)			
5			3.3 fmul(s)			
6			4.3 fmul(s)			
7			1.4 fadd			1.7 bge
8			2.4 fadd			2.7 bge
9			3.4 fadd			3.7 bge
10			4.4 fadd			4.7 blt

3 unrolls $\rightarrow$ 2.5 cycles per inner loop iteration

+ Scheduled using “normal” acyclic scheduler
+ Exposes parallelism between iterations in the unrolled loop
+ Allows optimization across unrolled iterations
– Dramatic code expansion
– Loop backedges, though “diluted,” still a barrier to ILP
– Suboptimal / heterogeneous utilization of resources

Improving on loop unrolling: Ad-hoc software pipelining

- Want to homogenize resource utilization
- Want to exploit parallelism across iterations
- Why not try rotating instructions around the loop backedge?

1	1.1 ld	1.2 ld	2.5 add	2.6 add	3.5 add	3.6 add
2	2.1 ld(s)	2.2 ld(s)	4.5 add	4.6 add		
3	3.1 ld(s)	3.2 ld(s)	1.3 fmul			

4	4.1 ld(s)	4.2 ld(s)	2.3 fmul(s)	1.5 add	1.6 add	2.5 add
5			3.3 fmul(s)	2.6 add	3.5 add	3.6 add
6			4.3 fmul(s)	4.5 add	4.6 add	
7			1.4 fadd			1.7 bge
8	1.1 ld(s)	1.2 ld(s)	2.4 fadd			2.7 bge
9	2.1 ld(s)	2.2 ld(s)	3.4 fadd			3.7 bge
10	3.1 ld(s)	3.2 ld(s)	4.4 fadd	1.3 fmul		4.7 blt

- More code growth
- How much to unroll?
- How much to move around backedge? In which direction?

Modulo scheduling

- Try to achieve near-optimal loop performance while performing only necessary unrolling
- Simplify software pipelining by
  - Using the same schedule for all iterations
  - Initiating iterations at a constant rate (II)
- Doesn't rely on prior unrolling or heuristic backedge traversals
- Two approaches:
  - "Unroll-during-scheduling" [Aiken, Nicolau 1991]: During acyclic scheduling of a loop body, continue unrolling and scheduling until a repetition is discovered.
  - "Iterative modulo scheduling" [Rau, 1994]: Determine analytically the Minimum Iteration Interval (MII) and generate a cyclic-capable schedule for a single iteration.

Iterative modulo scheduling

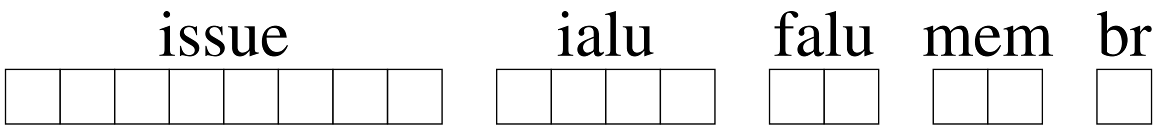
- Determine II lower bound (MII) and attempt to generate a schedule within it
- MII = Min(ResMII, RecMII)

  ResMII Resource-constrained: No resource is oversubscribed at any time modulo II.

  RecMII Recurrence-constrained: Dependence cycles are respected
- MII is the (possibly approximated) minimum interval between initiation of iterations. Thus II is most likely $\geq$ MII.
- Iterative modulo scheduling finds and attempts to compute a schedule for the MII, incrementing II and trying again on failure.
- Something to think about: Is MII a lower bound on all scheduling possibilities?

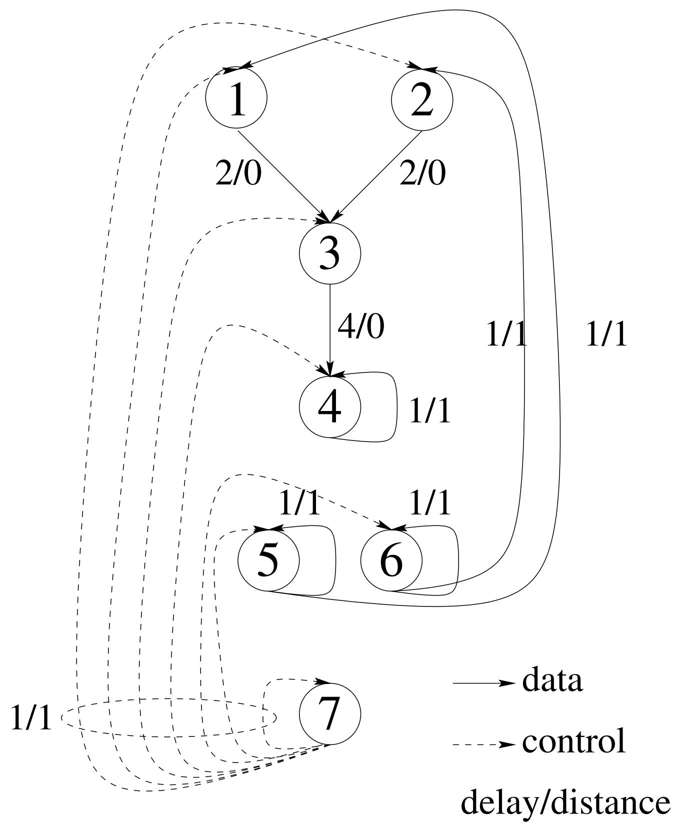
ResMII: Resource-constrained initiation interval

- Assuming no dependence between instructions, what is the minimum number of cycles required?
- Describe each instruction by a reservation table (MDES)
- "Bin-pack" instructions to achieve most efficient resource utilization possible.
- Heuristics for complex problems: allocate least flexible first, allocate to reduce ResMII
- Simple model: Usage count for most-used resource



- Our example uses: 7/8 issue, 2/2 mem, 2/4 ialu, 2/2 falu $\rightarrow$ ResMII=1

RecMII: Recurrence-constrained initiation interval



- Recurrences are cycles in the dependence graph.
- $delay$ = cycles between issue and result availability
- $distance$ = Iterations between producer and consumer
- Recurrence modulo constraint: Cyclic result must be available at consumer.

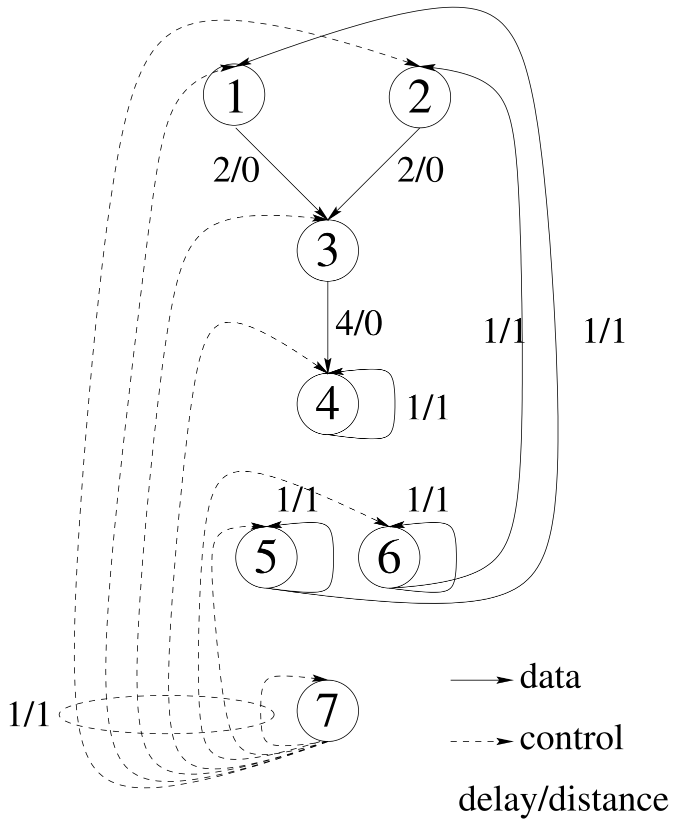
$$\sum_{c} delay_i \leq \text{II} \times \sum_{c} distance_i$$

- What if `fadd` (4) had a latency of 2?
- Also includes indirect dependences – cost of determining RecMII for real loops is non-trivial.

The Iterative Modulo Scheduling algorithm

- Attempt to schedule with increasing II until success.
- Rau uses an operation scheduling approach.
- Schedules into the schedule resource table of height II.
- Priority function based on height to exit
- Compile time considerations:
  - Scheduler schedules/unschedules ops repeatedly
  - Set a budget ratio which sets the number of operation scheduling attempts that will be made in trying to fit an II.
  - For Rau's benchmark set, a budget of 2:
    - Average 0.59 unschedules per op
    - generates optimal MII schedules in 96% of loops.
    - Computationally less expensive than a 118% unroll.
- Many variations: list schedulers, bidirectional list schedulers, etc.

Scheduling the example



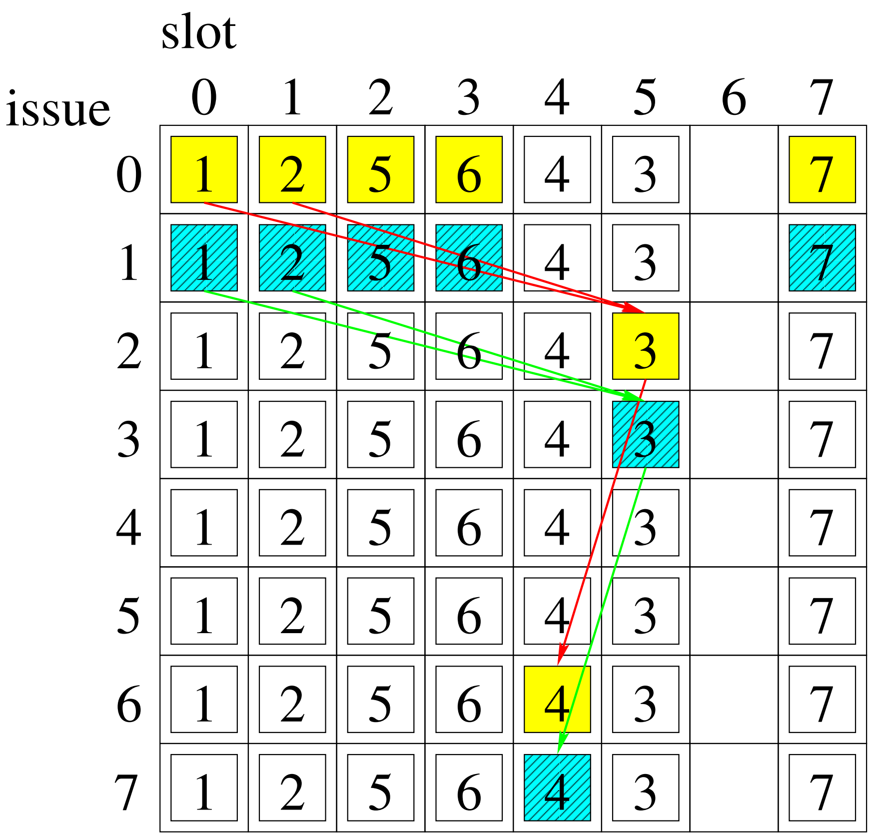
- MII = Min(RecMII, ResMII) = 1
- Instructions from different iterations are scheduled mod(II)
- Priority functions:
  - Operations in recurrences–deadlines (here)
  - HeightR (Rau)
- MII may not be achievable:
  - Complex resource usage
  - Resource-recurrence interaction

Modulo Resource Table

instruction / issue	slot0	slot1	slot2	slot3	slot4	slot5	slot6	slot7
instruction	1	2	5	6	4	3		7
issue	0	0	0	0	6	2		0

II

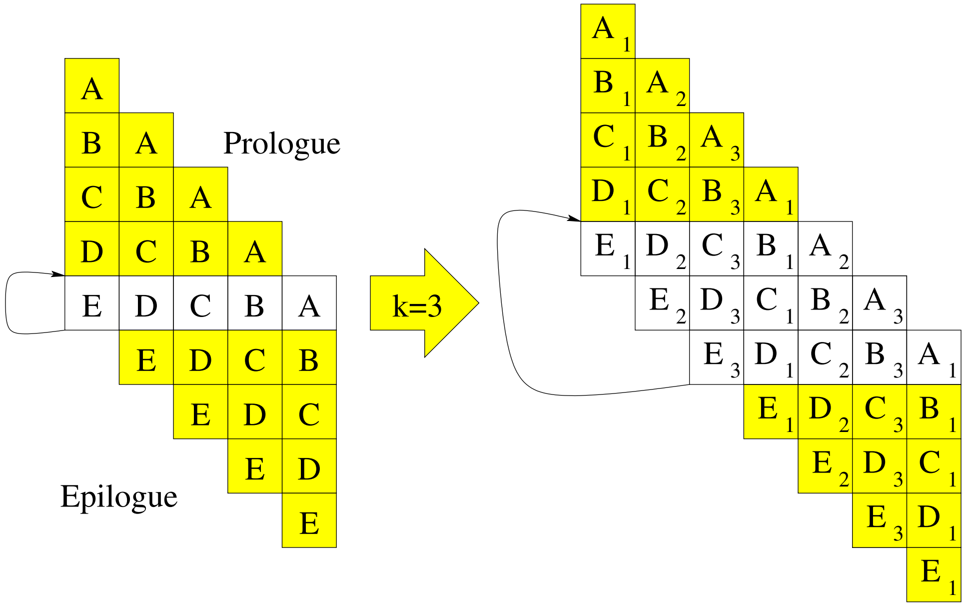
Code generation schema: making the pipeline work



- Two important problems:
  - Transient operation
    - ∗ Start up and stop the pipeline (prologue, epilog)
    - ∗ Correct operation for arbitrary trip count
  - Overlapped lifetimes
    - ∗ Respect value liveness
- Code generation schema vary in how they deal with these issues

Overlapped lifetimes–"software" approach

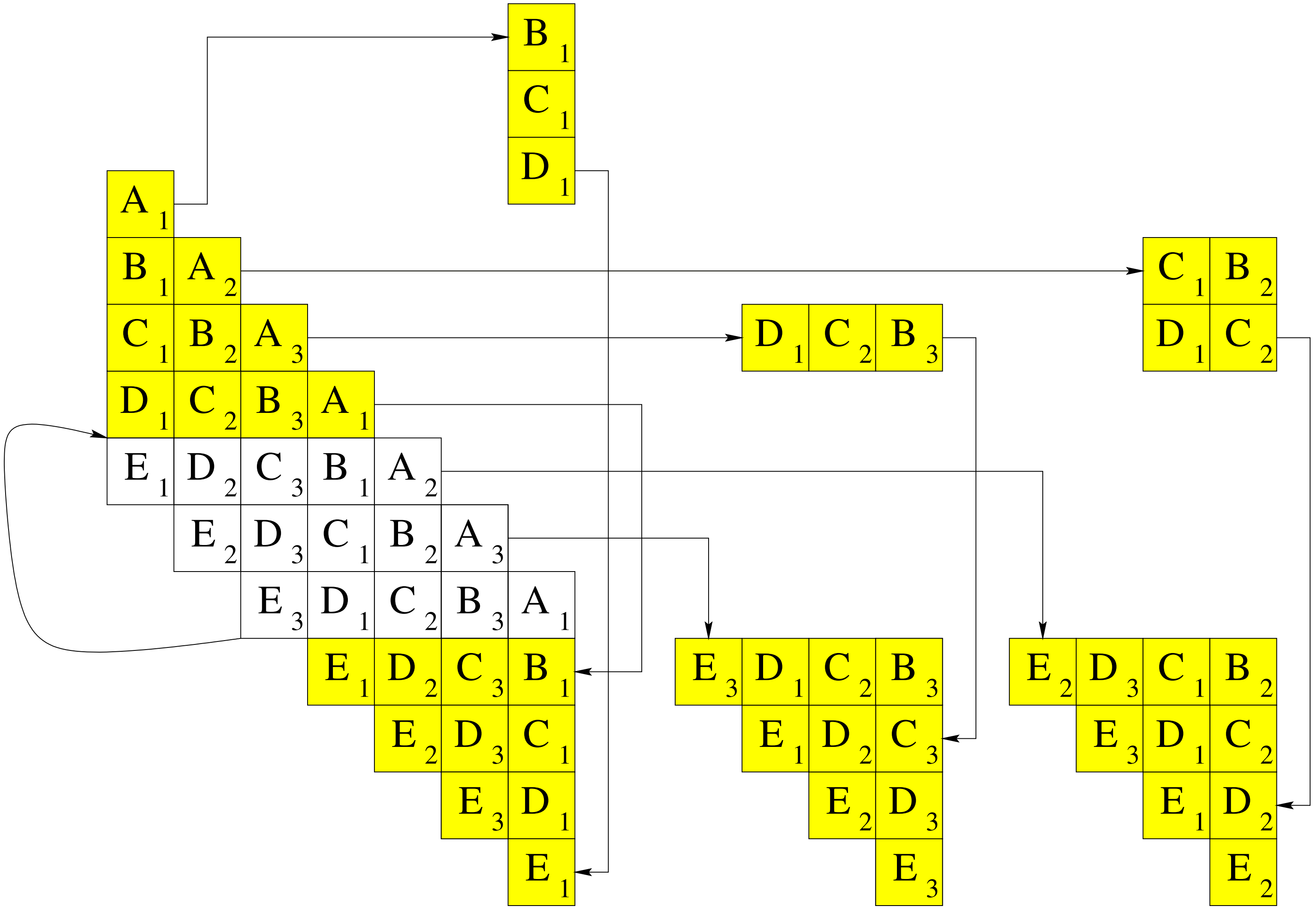
- **Modulo Variable Expansion (MVE)**: Rename registers as in acyclic code transformations, as needed to respect modulo liveness.
- Compute a kernel unrolling factor k
- $k = \max(\frac{v_i}{\mathrm{II}})$, where v_i is a register lifetime.
- In the previous code example, $k = v = 5$.
- Remainder of examples will be stylized for simplicity.



Correctness for all trip counts

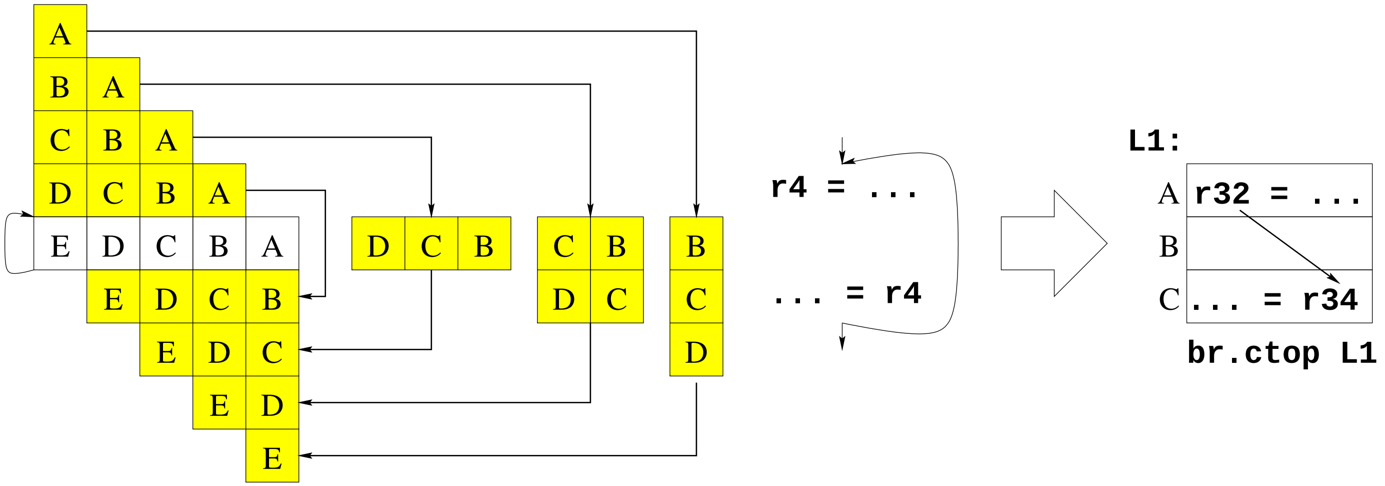
- Prologue and epilogue prime and drain the pipeline.
- With MVE, if exit is possible from any point in the prologue / kernel, renaming may be in an undesired state.
- Exits at end of prologue and end of kernel are valid: can only use the pipeline for (iters started in prologue) $+ (k \times i)$ iterations.
- Two software solutions:
  - Pre- and post-conditioning
  - Multiple epilogues
- Pre- and post-conditioning
  - Execute excess iterations in a non-pipelined loop; use pipeline only for valid iteration counts.
  - Handles only counted loops (i.e. no while loops)

Modulo variable expansion—multiple epilogues



A hardware solution: Rotating register file

- MVE, prologue and epilogue seem to dampen code size benefit of modulo scheduling.
- Idea: why not have the hardware automatically rename registers in successive modulo stages?
- Each architectural register rotates through a number of modulo contexts
- Requires hardware awareness of transition between pipeline stages: loop control instructions



Dealing with control flow

- Discussion to this point has treated only loops containing straight-line code.
- Loops containing hammocks [WarL93]
  - Hierarchical reduction
  - If-conversion (predication)
- While loops [Inte99]
  - Compiler typically places decision at bottom of loop
  - Need support for speculative execution
- Multiple-exit loops [LavH96]
  - Loop side exits require special care in winding down pipeline
  - Multiple- to single-exit loop conversion

IA-64 architectural support for modulo scheduling [Inte99]

- Solidly grounded in Cydra 5 tradition
- Rotating registers implemented as subset of architectural register set
- Loop control instructions rotate register files, manage
- Predication (Iteration Control Register(ICR) on Cydra)
  - If-convert loop bodies containing control flow
  - Control execution of prologue and epilogue
    Can reduce or eliminate need for code replication
  - Appears later in course

Counted loops [Inte99]

- Initial loop (note postincrement memory operations):

```
L1: ld4  r4 = [r5],4;;  // C0
    add  r7 = r4, r9;;  // C2
    st4  [r6] = r7,4    // C3
    br.cloop L1;;       // C3
```

- Modulo scheduled counted loop:

```
         mov  lc = 199       //LC=loop count-1
         mov  ec = 4         //EC=epilog stages+1
         mov  pr.rot = 1<<16 //PR16=1, rest=0
L1: (p16)ld4  r32 = [r5],4
    (p18)add  r35 = r34, r9
    (p19)st4  [r6] = r36,4
         br.ctop L1;;
```

While loops [Inte99]

- Initial loop:

```
L1:       ld4    r4 = [r5],4;;     // C0
          st4    [r6] = r4,4       // C2
          cmp.ne p1,p0 = r4,r0     // C2
     (p1) br L1;;                  // C2
```

- Modulo scheduled while loop:

```
           mov    ec = 2           //EC=epilog stages+1
           mov    pr.rot = 1<<16   //PR16=1, rest=0
L1:        ld4.s  r32 = [r5],4
     (p18) chk.s  r34, recovery
     (p18) cmp.ne p17,p0 = r34,r0
     (p18) st4    [r6] = r34, 4
     (p17) br.wtop.sptk L1;;
```

Extensions to modulo scheduling

- What if example issue width had been 6 instead of 8?
- Effectively want a non-integral or non-constant II
- Some optimizations operate most effectively in unrolled code
- Unrolling-based optimizations prior to modulo scheduling [LavH92,Inte99]
- Loop unrolling manipulates ResMII
  - Machine with 2 memory units
  - A loop with 3 memory ops has a ResMII$\geq$2.
  - Unrolling the loop once prior to modulo scheduling may allow 2 iterations in a 3 cycle II, for an effective II of 1.5.
- This effectively produces a non-integral or “variable” II modulo schedule.

For further information

[RauS92] B. Rau, M. Schlansker and P. Tirumalai. Code generation schema for modulo scheduled loops. MICRO25.

[RauB94] B. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. MICRO27.

[LavH96] D. Lavery and W. Hwu. Modulo scheduling of loops in control-intensive non-numeric programs. MICRO29.

[LavH92] D. Lavery and W. Hwu. Unrolling-based optimizations for modulo scheduling. MICRO28.

[WarL93] N. Warter, D. Lavery and W. Hwu. The benefit of predicated execution for software pipelining. HICSS26.

[Inte99] Chapter 12, “Software pipelining and loop support,” IA-64 Application Developer’s Architecture Guide, Rev. 1.0. (Intel web site) (highly recommended)