Software Pipelining

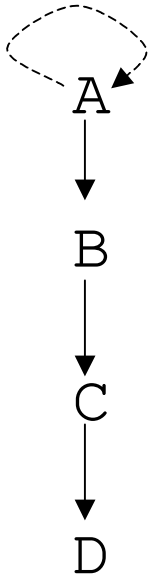
Principles and Modulo Scheduling Algorithm

Acyclic and Cyclic Scheduling

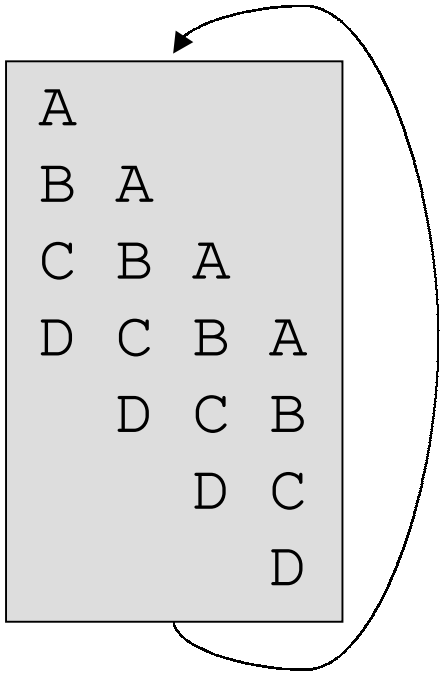
- Scheduling of acyclic code:
  - List scheduling: basic blocks
  - Trace/Superblock scheduling: sequences of basic blocks
- In the presence of loops: cyclic scheduling
  - Unroll the loop n times and then schedule the body of the new loop. Drawbacks:
    - code growth
    - no overlapping across back edge
  - Scheduling loops for overlapping executions of several consecutive iterations: Software Pipelining.

Loop Unrolling and Software Pipelining

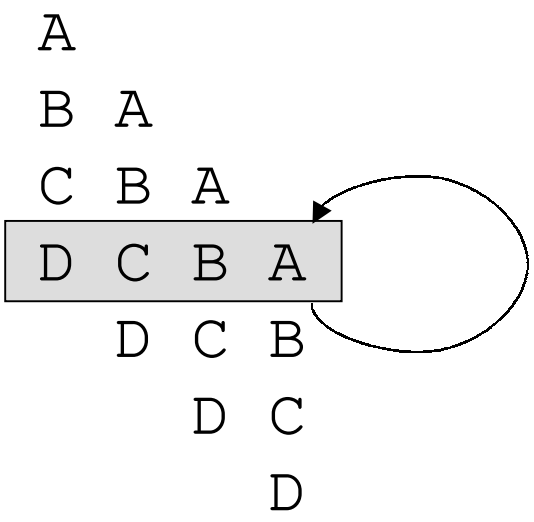
Data Dependence Graph:



After Loop Unrolling (4x)



After Software Pipelining

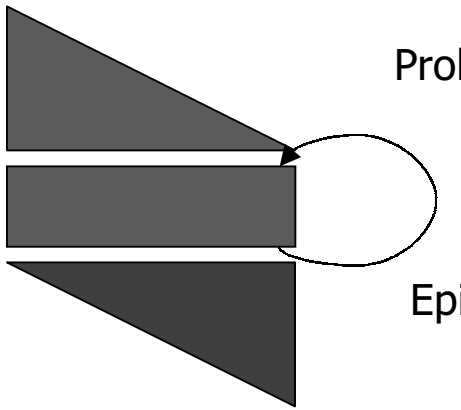


Terminology

- Operation: Machine operation, e.g. `add`, `load`, `store`. Names: a, b, c, ...
- Instruction: Set of machine operations scheduled at the same position.
  Names: A, B, C, ...
- Latency: Execution time of an operation.
- Delay: Distance between two consecutive dependent operations.
- Schedule: Mapping from operations to positions (cycles).
  Names: σ, ...

Goal of Software Pipelining

- Given:
  - loop with body L and λ iterations
  - p-times parallel architecture
- Wanted:
  - Efficient parallel schedule for L^{λ}.
  - L^{λ} is transformed into $\alpha\kappa^{k}\omega$ where κ is the body of a new loop, α the prologue (prelude) and ω the epilogue (postlude).



Prologue: initiates the pipeline

Kernel / Steady State

Epilogue: finishes the remaining iterations

Constraints

- Precedence constraints (data dependences)
- Resource constraints
- All operations from the body L occur equally often in the kernel κ, ρ times.
- Width of $\kappa \leq p$

- Goal: $|\kappa|$ minimal.

Approaches

- Move-then-schedule: move code forwards/backwards over loop back edge to improve schedule.
  Problems: Which operations to be moved, in which direction and how many times.

- Schedule-then-move: find a schedule and transform code accordingly.
  - Unroll-while-scheduling: Kernel recognition.
    Problems: Complex information to be maintained, complex checks are required, no realiable implementation?
  - Modulo-Scheduling: Generate and solve a set of modulo constraints.
    Problem: No control flow inside loops allowed (if conversion required).

Terminology and Properties of the Kernel

- Span μ of the kernel κ: number of consecutive iterations of L of which operations are contained in κ (in general: different from ρ).
- Initiation interval $II=|\kappa|$ is the distance between two successive iterations of the new loop.
- Observation:
  - Prelude starts μ-1 iterations and postlude finishes μ-1 iterations.
  - Number *k* of kernel iterations:

$$k = \frac{\lambda - 2(\mu - 1)}{\rho}$$

Data Dependences revised

- Iteration distance: Number of loop iterations between two dependent instruction instances (0 for intra-iteration dependences).
- Delay: Minimal number of clock cycles between the issuing of two dependent operation instances.
- Edges of the DDG are labeled with (*itDist*, *delay*).
- The delay for a dependence $a \rightarrow b$ depends on the latencies of *a* and *b* and the type of the dependence:
  - true dependence (def-use): *latency(a)*
  - anti dependence (use-def): *1 - latency(b)*
  - output dependence (def-def): *1 + latency(a) - latency(b)*
- For an operation *a* in L: a_n denotes the instance of *a* in the *n*-th iteration.
- Constraint for any schedule σ due to ($a \rightarrow b$, *itDist, delay*):

$$\sigma(b_{m+itDist}) \geq \sigma(a_m) + delay$$

Modulo Scheduling

- Goal: Compute a schedule for one iteration of the loop so that when it is repeated at regular intervals, no intra-or inter-iteration dependences are violated and no resource conflicts arise.

- Basic steps:
  1. compute a lower bound for the initiation interval II
  2. find a schedule
  3. generate the kernel code
  4. generate prologue and epilogue code

Increasing the Exploitable Parallelism

- if-conversion to eliminate branches
- elimination of pseudo dependences introduced by register allocation (register renaming or scalar renaming)
- rotating register files

Lower Bound for II_{min}

- Is determined by
  - resource consumption of operations
  - data dependences between operations

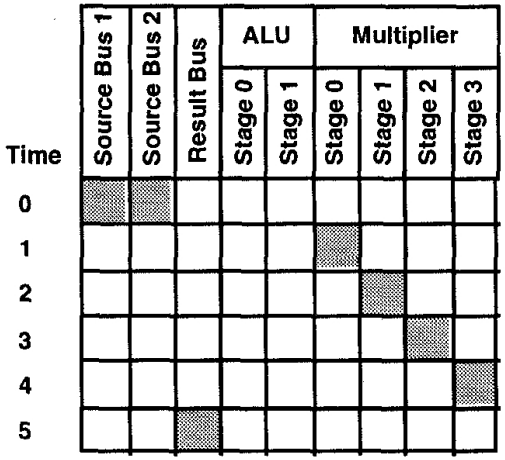
$$II_{\min} \geq \max\{II_{res}, II_{dep}\}$$

$$II_{res} = \min\{|\sigma| \,\|\, \sigma \textit{ is sequence of operations that satisfies resource availability restrictions}\}$$

$$II_{dep} = \max\left\{\left\lceil \frac{delay(\theta)}{itDist(\theta)} \right\rceil \,\middle|\, \theta \textit{ is cycle in the DDG}\right\}$$

Reservation Tables (1)

- For each operation, the operation reservation table defines the resource consumption at each cycle.
  RT: Cycles x *Resources* $\rightarrow$ *{0,1}*
- Resources are
  - Source and result buses
  - Stages of functional units.
- Cycles are the number of clock cycles after issuing the operation.



Reservation Tables (2)

- The schedule reservation table records for each cycle which resource is used by which operation by the schedule under construction.

- When an operation is scheduled at time t its reservation table is first translated by t and then further translated until a conflict-free overlay is possible.

Reservation Tables (3)

- Complexity of determining II_{res} depends on the type of resource consumption:
  - simple reservation table: a single resource is used for a single cycle which is the issue cycle.
  - block reservation table: a single resource is used for multiple, consecutive cycles starting at the issue cycle.
  - complex reservation table: all others
  - alternative reservation table: required for operations that can be executed on different functional units. In this case instruction scheduling has to be extended by resource allocation (functional unit binding).

Determining II_{res}

- Determining the minimal II_{res} is equivalent to binpacking:

  Given a set of n objects where the size s_i of the i-th object satisfies $0< s_i <1$. The goal is to pack all the objects into the minimum number of unit-size bins. Each bin can hold any subset of the objects whose total size does not exceed 1.

  Already the basic version is NP-hard.

- Complex and alternative reservation tables further complicate determining II_{res}.

Determining II_{res}

- For complexity reasons II_{res} is determined heuristically:
  1. Sort operations of loop body in increasing number of alternative reservation tables.
  2. For each operation a taken from this list:
     1. Traverse all alternatives and for each resource *r* compute an increment of the resource count for *r* by the number of times *a* uses *r*.
     2. Select the alternative with lowest (partial) maximal usage count over all resources.
     3. For each resource *r* increment the resource count for *r* by the number of time the chosen alternative of *a* uses *r*.
- In the end the usage count of the most heavily used resource constitutes the approximated II_{res}.

Determining II_{dep}

- Let $\theta=\{d_1,\ldots d_n\}$ be an elementary cycle of the dependence graph. Then define:

$$delay(\theta) = \sum_{i=1}^{n} delay(d_i)$$

$$itDist(\theta) = \sum_{i=1}^{n} itDist(d_i)$$

- For each schedule σ and each operation *a* from L:

$$\sigma(a_{m+i}) \geq \mathrm{II} \cdot \mathrm{i} + \sigma(a_m)$$

Determining II_{dep}

- Resulting constraint for II_{dep}:

$$\forall \theta : itDist(\theta) \cdot II_{dep} \geq delay(\theta)$$

$$\forall \theta : II_{dep} \geq \left\lceil \frac{delay(\theta)}{itDist(\theta)} \right\rceil$$

$$\text{Choose} : II_{dep} = \max_{\theta} \{ \left\lceil \frac{delay(\theta)}{itDist(\theta)} \right\rceil \}$$

Computing II_{dep}

- Based on algorithm for the minimal cost-to-time ratio cycle problem.

- Input:
  - II_{res}
  - MinDist[i,j]: smallest legal interval between $\sigma(i)$ and $\sigma(j)$ in the same iteration.

- Output: Minimum legal initiation interval II.

Computing II_{dep}

- Initialization:

$$MinDist[i,j] = \begin{cases} -\infty, \ if \ \ there \ \ is \ \ no \ \ edge \ \ between \ \ i \ \ and \ \ \ j \\ \max\{\max\{d \mid (i \rightarrow j, 0, d) \in E_{DDG}\}, \\ \qquad \max\{delay(e) - itDist(e) \cdot II \mid itDist(e) > 0, e \in E_{DDG}\}\} \end{cases}$$

- Starting from the initial value of II, increase II and recalculate MinDist. Three possible actions:
  - MinDist[i,i]>0: not possible $\Rightarrow$ II has to be increased.
  - MinDist[i,i]<0 for all i: slack around every cycle $\Rightarrow$ II can be decreased.
  - If no diagonal entry is positive and at least for one i: MinDist[i,i]=0, the algorithm terminates.

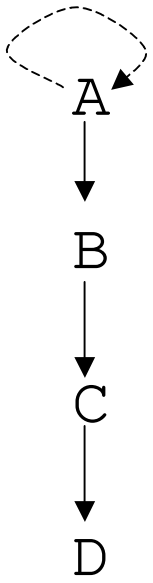
Time complexity: $O(n^3)$

Computing II_{dep}

- In each iteration: compute longest paths in the DDG using the *MinDist* matrix. Algorithm:

```
for k:=1 to NumberOfOperations {
  for i:=1 to NumberOfOperations {
    for j:=1 to NumberOfOperations {
      dist:=MinDist[i,k]+MinDist[k,j];
      if (dist > MinDist[i,j]) {
        MinDist[i,j]:=dist;
      }
    }
  }
}
```

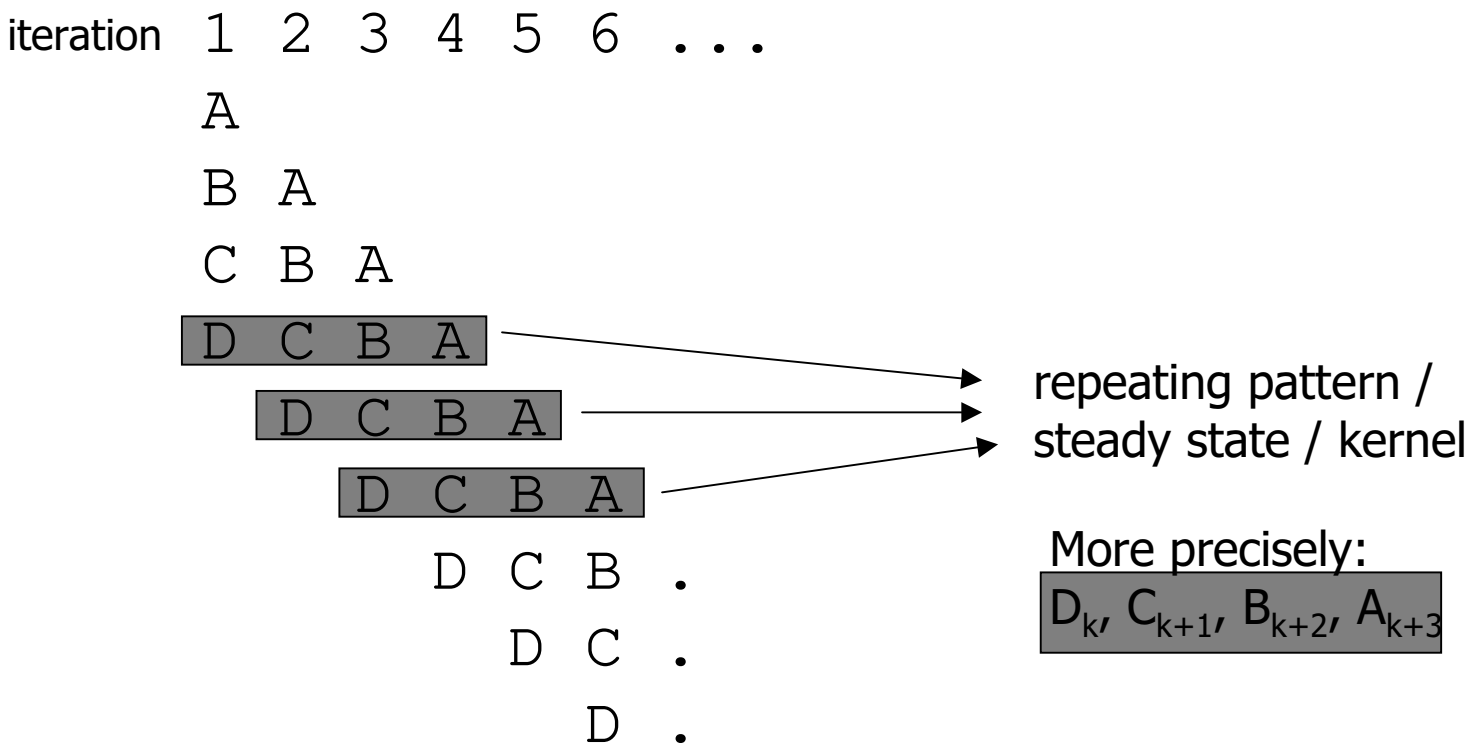
The Modulo Scheduling Idea



- Assume that a valid II has been computed, i.e. a new iteration of the kernel can be started every II cycles.
- Assume A,B,C,D are executed on different functional units, so II=1.
- A valid schedule for one iteration of the original loop is A;B;C;D.
- Conceptually: unroll the original loop with an appropriate schedule, but start a new iteration every II cycles.

The Modulo Scheduling Idea

- Result of conceptual unrolling and overlapping:



The Modulo Scheduling Idea

- Goal: Determine the kernel by looking at a single iteration of the original loop.

- A schedule of one iteration of the original can be divided into stages consisting of II cycles each.

- Idea: obtain the schedule for the kernel from that of a single iteration by superimposing it for each of these stages, i.e. by wrapping around the schedule for one iteration modulo II.

op	stage
A	0
B	1
C	2
D	3

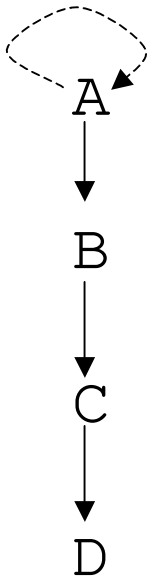
The Modulo Scheduling Idea

- Instances of the same operation in different iterations of the kernel are II cycles apart.
- Resource constraints satisfied, if no machine resource is used at two points in time that are separated by a multiple of II.
  This is the modulo constraint.
- The modulo schedule reservation table MRT[t mod II,r] records the use of a resource r at time t $\Rightarrow$ length of MRT = II.
- II is given from the outside, but there are no constraints on the number of iterations executing simultaneously.
- If B is scheduled 3II cycles later than A in the same iteration of the original loop, then instances of A and B are issued simultaneously in the kernel, but A has been issued three times before b is issued for the first time.

The Modulo Scheduling Idea

- The modulo schedule for a single iteration determines what code motion is implicit in the schedule.
- The operations in the stage containing the loopback branch are the only ones that have not been moved around the backedge.
- All operations in earlier/later stages have been moved backwards/forwards around the backedge to a previous/subsequent iteration.
- The greater the distance of a stage from the one containing the loopback branch the larger the number of times the operations have been moved around the backedge.

Simple Example



n=0: A has no predecessors ⇒ time(A)=0

n=1: Predecessor of B has been scheduled.

No conflict in MRT[n mod II]=MRT[1 mod 1]=MRT[0] ⇒ time(B)=1

n=2: Predecessor of C has been scheduled.

No conflict in MRT[2 mod 1]=MRT[0] ⇒ time(C)=2

n=3: Predecessor of D has been scheduled.

No conflict in MRT[3 mod 1]=MRT[0] ⇒ time(D)=3

Resulting Schedule and partitioning into stages:

op	stage	cycle
A	0	0
B	1	1
C	2	2
D	3	3

The Modulo Scheduling Idea

op	stage	cycle
A	0	0
B	1	1
C	2	2
D	3	3

- Only D in stage 3 has not been moved around the backedge; C has been moved once, B twice and A three times.
- Thus we have scheduled:
  - D_k to n mod II = 3 mod 1 = 0
  - C_{k-1} to (n-1) mod II = 2 mod 1 = 0
  - B_{k-2} to (n-2) mod II = 1 mod 1 = 0
  - A_{k-3} to (n-3) mod II = 0 mod 1 = 0
- Kernel Schedule: D_k, C_{k-1}, B_{k-2}, A_{k-3}

Modulo Scheduling vs. List Scheduling

- In modulo scheduling, an operation can be unscheduled by backtracking $\Rightarrow$ operations can be scheduled several times.
- The modulo schedule reservation table MRT[t mod II,r] records the use of a resource r at time t $\Rightarrow$ length of MRT = II.
- Conflict at time t $\Rightarrow$ conflict at all times t $\pm$ nII $\Rightarrow$ scheduling only for a candidate interval $[t_{min},t_{max}]$ where $t_{max}=t_{min}$+II-1.
- The procedure FindTimeSlot might not find a legal schedule of the current operation in the interval $\Rightarrow$ backtracking. List scheduling always finds a feasible schedule.

Iterative Modulo Scheduling

```
IterativeSchedule (II, budget)
{
  ComputeHeightR;  // scheduling priorities
  time(op_START):=0;
  budget--;
  forall operations op except op_START
    NeverScheduled[op]:=true;
  while (not all operations scheduled and budget>0) {
    op:=HighestPrioOp();
    estart:=CalculateEarlyStart(op);
    mintime:=estart;
    maxtime:=mintime+II-1;
    slot:=FindTimeSlot(op,mintime,maxtime);
    schedule(op,slot);
    budget:=budget-1;
  }
}
```

Scheduling Priority

- Basis: height-based priority.
- Extension for inter-iteration dependences:

$$HeightR(p) = \begin{cases} 0, \ if \quad p = op_{stop} \\ \max\limits_{q \in succ(p)} \{HeightR(q) + delay(p,q) - II \cdot itDist(p,q)\}, \ sonst \end{cases}$$

Candidate Time Slots

- Correctness of the schedule?
  - wrt. resource usage: guaranteed by MRT
  - wrt. data dependences: the earliest start time for the operation, Estart, has to be chosen so that no dependences are violated.
- Problem in computing Estart: due to unscheduling, it is impossible to guarantee that all predecessors of an operation have been scheduled $\Rightarrow$ When computimg HeightR only the currently scheduled immediate predecessors are considered; operations that conflict with the currently scheduled operation are unscheduled.

Candidate Slots and Unscheduling

$$Estart(p) = \max_{q \in pred(p)} \left\{ \begin{array}{l} 0, \ if \ \ q \ \ is \ \ unscheduled \\ \max\{0, SchedTime(q) + delay(q,p) - II \cdot itDist(q,p)\}, \ otherwise \end{array} \right\}$$

- Assume a time slot has been found between Estart and Estart+II-1. Conflicts are only possible with scheduled successors.
- In case of conflict $\Rightarrow$ unscheduling:
  - slot without resource conflicts found: unschedule operation with dependence conflict.
  - no slot without resource conflicts found: choose time slot + choose operation to unschedule.

End of Modulo Scheduling.

Kernel Recognition

1. Unroll loop once and analyze dependences.

2. Construct a schedule using greedy heuristics or height-based heuristics.

3. Search for a block of instructions that occurs twice consecutively ($\alpha\kappa\kappa\omega$).

4. Form new loop body out of this block κ if successful, otherwise goto 1.

Kernel Recognition

- The repetition test can be complex:
  - are the same operations scheduled?
  - when are the results available?
  - are the same reources occupied?
  - are the same registers occupied by the results?