Module:
Software Instruction
Scheduling - Part II

Reading Assignments

- Appendix G.3
- Papers as linked on the webpage
Overview

- The General Case: List Scheduling
- The Acyclic Case: Trace Scheduling
- The Cyclic Case: Software Pipelining/Modulo Scheduling
- Architectural Support
  - Very Long Instruction Word (VLIW) architectures
  - Explicitly Parallel Instruction Computing (Itanium)

Impact of Control Flow

Acyclic control flow is easier to deal with than cyclic control flow. Problems in dealing with cyclic flow:
- A loop implicitly represent a large run-time program space compactly.
- Not possible to open out the loops fully at compile-time.
- Loop unrolling provides a partial solution.

more...
Acyclic Instruction Scheduling

- We will consider the case of acyclic control flow first.

- The acyclic case itself has two parts:
  - The simpler case that we will consider first has no branching and corresponds to basic block of code, eg., loop bodies.
  - The more complicated case of scheduling programs with acyclic control flow with branching will be considered next.

---

Base Case: Scheduling Basic Blocks

Why are basic blocks easy?

- All instructions specified as part of the input must be executed.

- Allows deterministic modeling of the input.

- No “branch probabilities” to contend with; makes problem space easy to optimize using classical methods.
The Machine Model

Multiple Functional Units

- **Latency**: # cycles between an instruction that produces a result and one that consumes the result
- **Initiation interval**: rate at which instructions can be issued to a functional unit

Instruction Scheduling

Abstraction,

**Input:** A basic block represented as a DAG

- \( i_2 \) is a load instruction.
- Latency of 1 on \((i_2, i_4)\) means that \( i_4 \) cannot start for one cycle after \( i_2 \) completes.
Two schedules for the above DAG with $S_2$ as the desired sequence.

The General Instruction Scheduling Problem

**Input:** DAG representing each basic block where:

1. Nodes encode *unit execution time* (single cycle) instructions.
2. Each node requires a definite class of FUs.
3. Additional pipeline delays encoded as latencies on the edges.
4. Number of FUs of each type in the target machine.

*more...*
The General Instruction Scheduling Problem (cont.)

Feasible Schedule: A specification of a \textit{start time} for each instruction such that the following constraints are obeyed:

1. **Resource**: Number of instructions of a given type at any time < corresponding number of FUs.

2. **Precedence and Latency**: For each predecessor \(j\) of an instruction \(i\) in the DAG, \(i\) is started only \(\delta\) cycles after \(j\) finishes where \(\delta\) is the latency labeling the edge \((j,i)\),

\textbf{Output}: A schedule with the minimum \textit{overall completion time (makespan)}.

---

Deterministic Scheduling

\textbf{Canonical Algorithm}:

1. Assign a \textit{Rank} (priority) to each instruction (or node).

2. Sort and build a priority \textit{list} \(\mathcal{L}\) of the instructions in non-decreasing order of Rank.

   Nodes with smaller ranks occur earlier in this list.
3. **Greedily list-schedule** $\mathcal{L}$.
   
   Scan $\mathcal{L}$ iteratively and on each scan, choose the largest number of “ready” instructions subject to resource (FU) constraints in list-order.

   An instruction is ready provided it has not been chosen earlier and all of its predecessors have been chosen and the appropriate latencies have elapsed.

---

---

The Value of Greedy List Scheduling

Example: Consider the DAG shown below where nodes are labeled (ID, rank):

![DAG Diagram]

Using the list $\mathcal{L} = <i_{4,1}, i_{5,1}, i_{2,2}, i_{3,3}, i_{1,3}>$

- Greedy scanning produces the steps of the schedule as follows:
Greedy List Scheduling (Contd.)

1. On the first scan: $i_1$ which is the first step.

2. On the second and third scans and out of the list order, respectively $i_4$ and $i_5$ to correspond to steps two and three of the schedule.

3. On the fourth and fifth scans, $i_2$ and $i_3$ respectively scheduled in steps four and five.

Some Intuition

- Greediness helps in making sure that idle cycles don’t remain if there are available instructions further “down stream.”

- Ranks help prioritize nodes such that choices made early on favor instructions with greater enabling power, so that there is no unforced idle cycle.
**Running Time of Greedy List Scheduling:**
Linear in the size of the DAG.


**An Example Rank Function**

The example DAG

1. Initially label all the nodes by the same value, say $\alpha$.
2. Compute new labels from old starting with nodes at level zero ($i_4$) and working towards higher levels:
   (a) All nodes at level zero get a rank of $\alpha$. 

more...
(b) For a node at level 1, construct a new label which is the concentration of all its successors connected by a latency 1 edge.

Edge $i_2$ to $i_4$ in this case.

(c) The empty symbol $\emptyset$ is associated with latency zero edges.

Edges $i_3$ to $i_4$ for example.

(d) The result is that $i_2$ and $i_3$ respectively get new labels and hence ranks $\alpha' = \alpha > \alpha'' = \emptyset$.

Note that $\alpha' = \alpha > \alpha'' = \emptyset$ i.e., labels are drawn from a totally ordered alphabet.

(e) Rank of $i_1$ is the concentration of the ranks of its immediate successors $i_2$ and $i_3$ i.e., it is $\alpha''' = \alpha' | \alpha''$.

3. The resulting sorted list is (optimum) $i_1, i_2, i_3, i_4$. 
• How can we generate a large enough pool of instructions in the presence of control flow

• Acyclic control flow is easier to handle than cycle control flow

• First a few definitions....

---

Machine Model

Multiple Functional Units

- **Latency**: # cycles between an instruction that produces a result and one that consumes the result
- **Initiation interval**: rate at which instructions can be issued to a functional unit
The Program Dependence Graph (PDG) is the intermediate (abstract) representation of a program designed for use in optimizations.

It consists of two important graphs:

- Control Dependence Graph captures control flow and control dependence
- Data Dependence Graph captures data dependences

Control Flow Graphs

Motivation: language-independent and machine-independent representation of control flow in programs used in high-level and low-level code optimizers. The flow graph data structure lends itself to use of several important algorithms from graph theory.

For more, take the course on Optimizing Compilers
A control flow graph \( \text{CFG} = ( \mathcal{N}_c ; \mathcal{E}_c ; \mathcal{T}_c ) \) consists of

- \( \mathcal{N}_c \), a set of nodes. A node represents a straight-line sequence of operations with no intervening control flow i.e. a **basic block**.
- \( \mathcal{E}_c \subseteq \mathcal{N}_c \times \mathcal{N}_c \times \text{Labels} \), a set of labeled edges.
- \( \mathcal{T}_c \), a node type mapping. \( \mathcal{T}_c(n) \) identifies the type of node \( n \) as one of: \text{START}, \text{STOP}, \text{OTHER}.

We assume that \( \text{CFG} \) contains a unique \text{START} node and a unique \text{STOP} node, and that for any node \( N \) in \( \text{CFG} \), there exist directed paths from \text{START} to \( N \) and from \( N \) to \text{STOP}.
The Challenge

- The problem being optimized is no longer deterministically and completely known at compile-time.
- Depending on the sequence of branches taken, the problem structure of the graph being executed can vary.
- Impractical to optimize all possible combinations of branches and have a schedule for each case, since a sequence of \( k \) branches can lead to \( 2^k \) possibilities -- a combinatorial explosion in cost of compiling.

Containing Compilation Cost

A well known classical approach is to consider traces through the (acyclic) control flow graph. An example is presented in the next slide.
An Example Trace

```
ECE 4100/6100  (29)

BB-1
BB-4
BB-6
BB-2
BB-3
BB-4
BB-5
BB-6
BB-7
STOP

A trace BB-1, BB-4, BB-6

ECE 4100/6100  (29)

Traces


Main Ideas:

- Predict the execution sequence
  - Choose one of the paths out of each branch that is encountered.
- Now you have a big basic block
- Great for programs with predictable execution sequences

ECE 4100/6100  (30)`
• Use statistical knowledge based on (estimated) program behavior to bias the choices to favor the more frequently taken branches.

• This information is gained through profiling the program or via static analysis.

• The resulting sequence of basic blocks including the branch instructions is referred to as a trace.

---

Trace Scheduling

High Level Algorithm:

1. Choose a (maximal) segment $s$ of the program with acyclic control flow.
   The instructions in $s$ have associated “frequencies” derived via statistical knowledge of the program’s behavior.

2. Construct a trace $\tau$ through $s$:
   (a) Start with the instruction in $s$, say $i$, with the highest frequency.
(b) Grow a path out from instruction \( i \) in both directions, choosing the path to the instruction with the higher frequency whenever there is

Frequencies can be viewed as a way of prioritizing the path to choose and subsequently optimize.

3. Rank the instructions in \( \tau \) using a rank function of choice.
4. Sort and construct a list \( \mathcal{L} \) of the instructions using the ranks as priorities.
5. Greedily list schedule and produce a schedule using the list \( \mathcal{L} \) as the priority list.

---

**Significant Comments**

- We pretend as if the trace is always taken and executed and hence schedule it in steps 3-5 using the same framework as for a basic-block.
- The important difference is that conditionals branches are there on the path, and moving code past these conditionals can lead to side-effects.
- These side effects are not a problem in the case of basic-blocks since there, every instruction is executed all the time.
- This is not true in the present more general case when an outgoing or incoming off-trace branch is taken however infrequently: we will study these issues next.
The Four Elementary but Significant Side-effects

Consider a single instruction moving past a conditional branch:

- Branch Instruction
- Instruction being moved

The First Case

- This code movement leads to the instruction executing sometimes when the instruction ought not to have: *speculatively.*
The First Case (Contd.)

- If A is a write of the form \( a := \ldots \), then, the variable (virtual register) \( a \) must not be live on the off-trace path.

- In this case, an additional pseudo edge is added from the branch instruction to instruction A to prevent this motion.

The Second Case

- Identical to previous case except the pseudo-dependence edge is from A to the join instruction whenever A is a “write” or a def.

- A more general solution is to permit the code motion but undo the effect of the speculated definition by adding repair code.

- An expensive proposition in terms of compilation cost.
• Instruction $A$ will *not* be executed if the off-trace path is taken.
• To avoid mistakes, it is *replicated*.

The Third Case (Contd.)

• This is true in the case of read and write instructions.
• Replication causes $A$ to be executed independent of the path being taken to preserve the original semantics.
• If (non-)liveness information is available, replication can be done more conservatively.
The Fourth Case

- Similar to Case 3 except for the direction of the replication as shown in the figure above.

At a Conceptual Level: Two Situations

- **Speculations:** Code that is executed “sometimes” when a branch is executed is now executed “always” due to code motion as in Cases 1 and 2.
  - **Legal** speculations wherein data-dependences are not violated.
  - **Safe** speculation wherein control-dependences on exceptions-causing instructions are not violated.
Unsafe speculation where there is no restriction and hence exceptions can occur.

This type of speculation is currently playing a role in “production quality” compilers.

- Replication: Code that is “always” executed is duplicated as in Cases 3 and 4.

Comparison to Basic Block Scheduling

- Instruction scheduler needs to handle speculation and replication.
- Otherwise the framework and strategy is identical.
Fisher’s Trace Scheduling Algorithm

Description:

1. Choose a (maximal) region $s$ of the program that has acyclic control flow.

2. Construct a trace $\tau$ through $s$.

3. Add additional dependence edges to the DAG to limit speculative execution. Note that this is Fisher’s solution.

4. Rank the instructions in $\tau$ using a rank function of choice.

5. Sort and construct a list $\mathcal{L}$ of the instructions using the ranks as priorities.

6. Greedily list schedule and produce a schedule using the list $\mathcal{L}$ as the priority list.

7. Add replicated code whenever necessary on all the off-trace paths.
Example

TRACE: BB6, BB2, BB4, BB5

Obvious advantages of global code motion are that the idle cycles have disappeared.

Concentration of Local Schedules

Feasible Schedule: 6-1 X 6-2 2-1 X 2-2 2-3 X 2-4 2-5 4-1 X 4-2 5-1

Global Improvements 6-1 2-1 6-2 2-2 2-3 X 2-4 2-5 4-1 X 4-2 5-1:

6-1 2-1 6-2 2-2 2-3 X 2-4 2-5 4-1 X 4-2 5-1

X: Denotes Idle Cycle
Acyclic Scheduling: Key Concepts

- Honor dependencies

- Rank Function
  - Capture importance → influence the future

- Greedy scheduling policy
  - As opposed to look-ahead or backtracking

- Increasing the scheduling pool
  - Trace scheduling
  - Other compiler strategies exist to increase the pool
    - Did not discuss Superblocks or Hyperblocks

Scheduling Cyclic Control Flow Graphs

- Conventionally, loop-body is executed sequentially from one iteration to the next

- By compile-time analysis, execution of successive iterations of a loop is overlapped exposing ILP

- Reminiscent of execution in hardware pipelines, and hence this is also referred to as software pipelining.

- No underlying hardware support is needed.

- First form scheduling polycyclic architectures (Rau 1982)
• If A, B, C, and D must be executed sequentially.
• If each operation requires 1 cycle each, and \( n = 1000 \), then loop requires 4000 cycles.

**Example With Unbounded Resources**

- New loop body contains independent instructions, i.e., they can be executed in parallel if resources allow.
- If all 4 ops can be scheduled in parallel, the loop requires \((n-3)+6\) cycles, where \( n = 1000 \) results in 1003 cycles needed vs. 4000 cycles in the sequential case.
- If ILP > 1 then there is potential improvement.
Example

Loop: L.D F0,0(R1)
ADD.D F4,F0,F2
S.D F4,0(R1)
DADDUI R1,R1,#-8
BNE R1,R2,Loop

- Simple loop with independent loop iterations

Unroll Iterations

Iteration i: L.D F0,0(R1)
ADD.D F4,F0,F2
S.D F4,0(R1)
Iteration i+1: L.D F0,0(R1)
ADD.D F4,F0,F2
S.D F4,0(R1)
Iteration i+2: L.D F0,0(R1)
ADD.D F4,F0,F2
S.D F4,0(R1)

- Ignore the loop termination check code for a moment
Select Instructions from Each Iteration

Iteration i:  
L.D F0,0(R1)  
ADD.D F4,F0,F2  
**S.D F4,0(R1)**

Iteration i+1:  
L.D F0,0(R1)  
**ADD.D F4,F0,F2**  
S.D F4,0(R1)

Iteration i+2:  
**L.D F0,0(R1)**  
ADD.D F4,F0,F2  
S.D F4,0(R1)

- These instructions are independent increasing ILP

---

The Software Pipelined Loop

Loop:  
S.D F4,16(R1) ; stores into M[i]  
ADD.D F4,F0,F2 ; adds to M[i-1]  
L.D F0,0(R1) ; loads M[i-2]  
DADDUI R1,R1,#-8  
BNE R1,R2,Loop

- Need a Prolog and Epilog to setup and clean up
Prolog and Epilog

Prolog:
- L.D F0,0(R1)
- ADD.D F4,F0,F2
- L.D F0,-8(R1)
- DADDUI R1,R1,#-16

Set up first two iterations

Start of array

Epilog:
- S.D F4,-8(R1)
- ADD.D F4,F0,F2
- S.D F0,0(R1)

Clean up last two iterations

R1=R2

Consider a graphical view of the overlay of iterations:

Kernel

Prolog

Epilog

Only the shaded part, the loop kernel, involves executing the full width of the VLIW instruction.

- The loop prolog and epilog contain only a subset of the instructions.
- “ramp up” and “ramp down” of the parallelism.
Modulo Scheduling

- Find a steady-state schedule for the loop body
- The length of this schedule is the initiation interval (II)
- The same schedule is executed in every iteration
- **GOAL**: minimize the initiation interval
- Prologue and epilogue are recovered from the kernel

Constraints on The Compiler in Determining Schedule

Since there are no expectations on hardware support at run-time:

- The overlapped execution on each cycle must be possible with the degree of instruction level parallelism in terms of functional-units
- The inter-instruction latencies must be obeyed within each iteration but *more importantly* across iterations as well.
- These inter-iteration dependences and consequent latencies are loop-carried dependencies.
Modulo Scheduling Example

Loop{
    P = A + B
    Q = C + D
    X = P * E
    Y = P * Q
    Z = X + Y
}

Step 1: Data flow graph

From Prof. H. H. Lee’s ECE 4100/6100 Lecture Notes

Modulo Scheduling

Step 2: Generate a list schedule

Execution units:
2 Adders – 1 cycle latency
1 Multiplier – 2 cycle latency

From Prof. H. H. Lee’s ECE 4100/6100 Lecture Notes
Modulo Scheduling

Step 2: Generate a list schedule

Reservation Table

<table>
<thead>
<tr>
<th>Time</th>
<th>Adder1</th>
<th>Adder2</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A1</td>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>M1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>M2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>A3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Generating Modulo Schedule:

1. Determine the MII:

\[
MII = \max \left( \frac{\text{Re source - demand}}{\text{Re source - availability}}, \frac{N}{C} \right)
\]

\[
MII = \max[(3/2), (2/1)] = 2
\]

From Prof. H. H. Lee’s ECE 4100/6100 Lecture Notes
Modulo Scheduling

Mapping from list schedule to modulo schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Adder1</th>
<th>Adder2</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A1</td>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>M1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>M2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>A3</td>
<td></td>
</tr>
</tbody>
</table>

**List schedule**

<table>
<thead>
<tr>
<th>Time</th>
<th>Modulo 2</th>
<th>Adder1</th>
<th>Adder2</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>A1</td>
<td>A2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>M1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td>M2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td>A3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td></td>
<td>A3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Modulo schedule for 1 iteration**

From Prof. H. H. Lee’s ECE 4100/6100 Lecture Notes

Inserting iteration 2

<table>
<thead>
<tr>
<th>Time</th>
<th>Modulo 2</th>
<th>Adder1</th>
<th>Adder2</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1:A1</td>
<td>1:A2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1:M1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2:A1</td>
<td>2:A2</td>
<td>1:M2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td>2:M1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td>2:M2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1:A3</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td></td>
<td>2:A3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From Prof. H. H. Lee’s ECE 4100/6100 Lecture Notes
Modulo Scheduling

<table>
<thead>
<tr>
<th>Time</th>
<th>Modulo 2</th>
<th>Adder 1</th>
<th>Adder 2</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1:A1</td>
<td>1:A2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>1:M1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2:A1</td>
<td>2:A2</td>
<td>1:M2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td>2:M1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>3:A1</td>
<td>3:A2</td>
<td>2:M2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1:A3</td>
<td></td>
<td>3:M1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td>3:M2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td></td>
<td>2:A3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td></td>
<td>3:A3</td>
<td></td>
</tr>
</tbody>
</table>

Inserting iteration 3

From Prof. H. H. Lee’s ECE 4100/6100 Lecture Notes

Modulo Scheduled Loop

<table>
<thead>
<tr>
<th>Time</th>
<th>Modulo 2</th>
<th>Adder 1</th>
<th>Adder 2</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1A1</td>
<td>1A2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>1M1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2A1</td>
<td>2A2</td>
<td>1M2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td>2M1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>3A1</td>
<td>3A2</td>
<td>2M2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1A3</td>
<td></td>
<td>3M1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>4A1</td>
<td>4A2</td>
<td>3M2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2A3</td>
<td></td>
<td>4M1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>5A1</td>
<td>5A2</td>
<td>4M2</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>3A3</td>
<td></td>
<td>5M1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>6A1</td>
<td>6A2</td>
<td>5M2</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>4A3</td>
<td></td>
<td>6M1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>7A1</td>
<td>7A2</td>
<td>6M2</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>5A3</td>
<td></td>
<td>7M1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>6A3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>7A3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Prolog

5x Kernel

Epilog

From Prof. H. H. Lee’s ECE 4100/6100 Lecture Notes

ECE 4100/6100 (67)
Consider a graphical view of the overlay of iterations:

- Loop prolog and epilog can be optimized by hardware support
  - Wait until we discuss the Itanium

---

Study Guide: Glossary

- Acyclic vs. cyclic control flow
- Basic block scheduling
- Control flow and dataflow graph
- Directed acyclic graph (DAG)
- Feasible schedule
- Greedy scheduling
- Initiation interval (in modulo scheduling)
- List scheduling
- Logical processor vs. physical processor
- Modulo scheduling
- Program dependence graph
- Prologue and epilogue
- Rank function
- Speculation (in trace scheduling)
- Software pipelining
- Trace scheduling

---
Study Guide

- Be able to manually create schedules for list scheduling, trace scheduling, and modulo scheduling

- Application of different rank functions

- For what types of loop body code sequences does modulo scheduling perform best

- Conceive of new rank functions for list scheduling that are well motivated