Minimizing Register Requirements under Resource-Constrained Rate-Optimal Software Pipelining

R. Govindarajan
Memorial University of Newfoundland

Erik R. Altman and Guang R. Gao
McGill University, Montreal

Outline

- Introduction
- Schedule Space of Software Pipelining
- Motivating Example
- ILP Problem Formulation
- Experimental Results
- Related Work
- Future Work

Software Pipelining

- Efficient compilation technique for loop scheduling
- Derives a periodic pattern overlapping instructions from successive iterations
- Constructed schedule should obey
  - Dependency constraints
  - Resource Constraints
    - Function Units
    - Processor Registers

Problem Statement

<u>Problem 1</u>

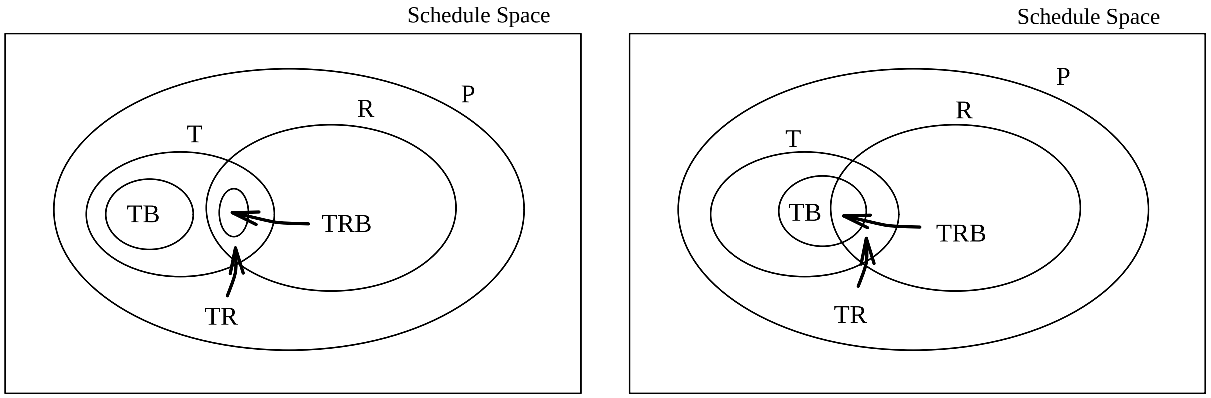
Given a Loop L and an architecture with F FUs
Construct a ***resource-constrained*** schedule with the ***minimum*** initiation interval.

Scheduling problem is **NP-Hard**.

What about register requirement?

<u>Problem 2</u>

Given a Loop L and an architecture with F FUs
Construct a ***resource-constrained*** schedule with the ***minimum*** initiation interval and uses ***minimum*** registers.



(a) TB does not intersect with TRB

(b) TB interesects TRB -- TRB is buffer optimal

P -- All Periodic Schedules

T -- Schedules with Period Tmin

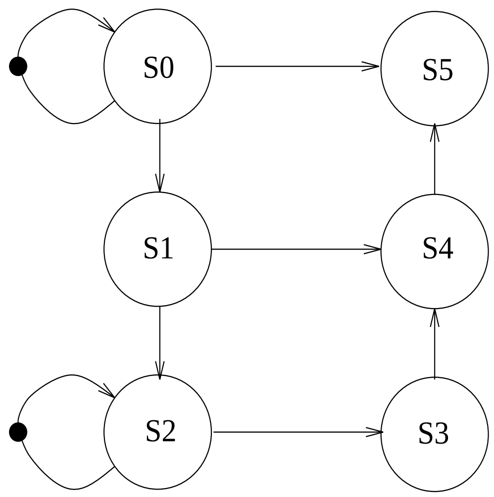
R -- Schedules using R or less FUs

TB -- Schedules with Period Tmin and with Optimal Buffers

TR -- Schedules with Period Tmin and using R or less FUs

TRB -- Schedules with Period Tmin, using R or less FUs and with minimum Buffers

Motivating Example



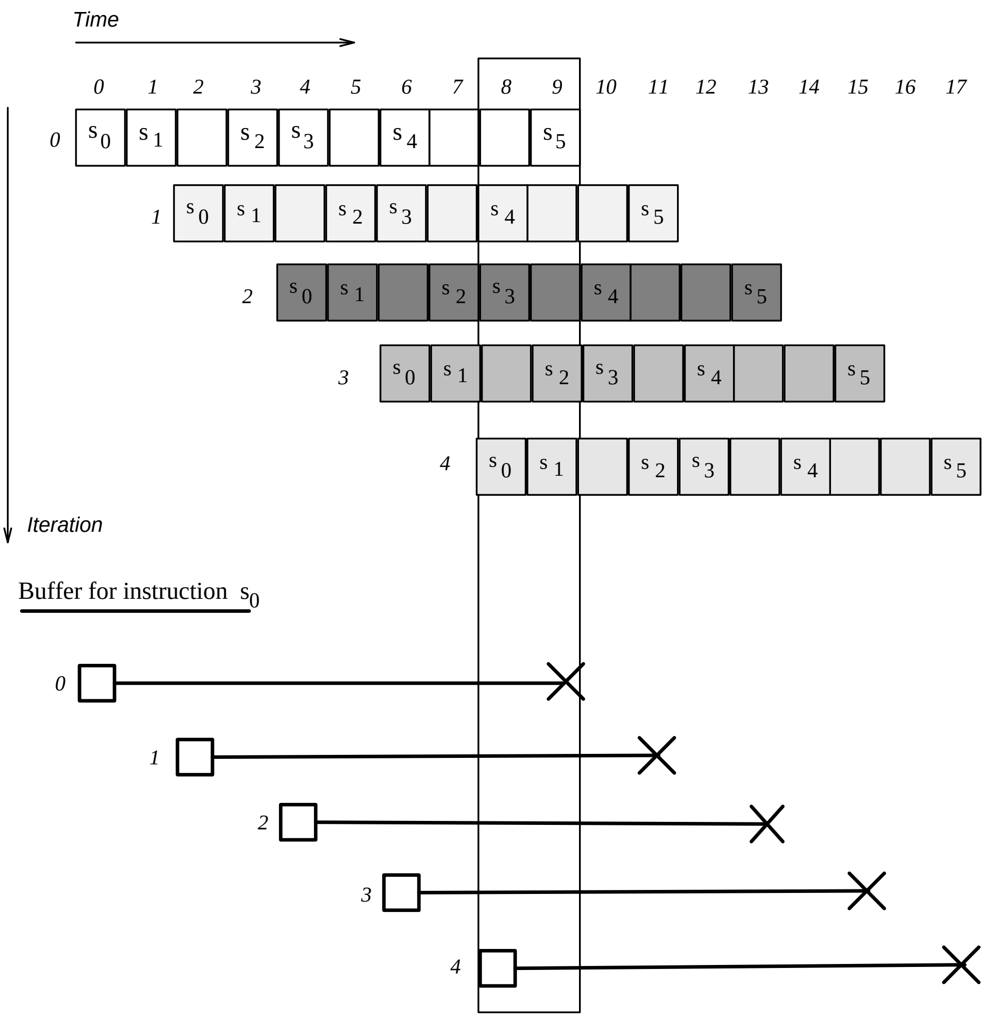

```
for (i = 0; i < n; i++) {
    s = s + a[i];
    a[i] = s * s * a[i];
}
```

(a) Source Code and Dependency Graph

s_0:	vr33 = vr33 + vr32	% vr33 is address of a[i] %
s_1:	vr34 = load m(vr33)	% vr34 = a[i] %
s_2:	vr35 = vr35 + vr34	% vr35 = s %
s_3:	vr36 = vr35 * vr35	
s_4:	vr37 = vr36 * vr34	% vr37 = new a[i] %
s_5:	store(vr37, m(vr33))	
	branch to s_0 if $i \le n$	
	enddo	

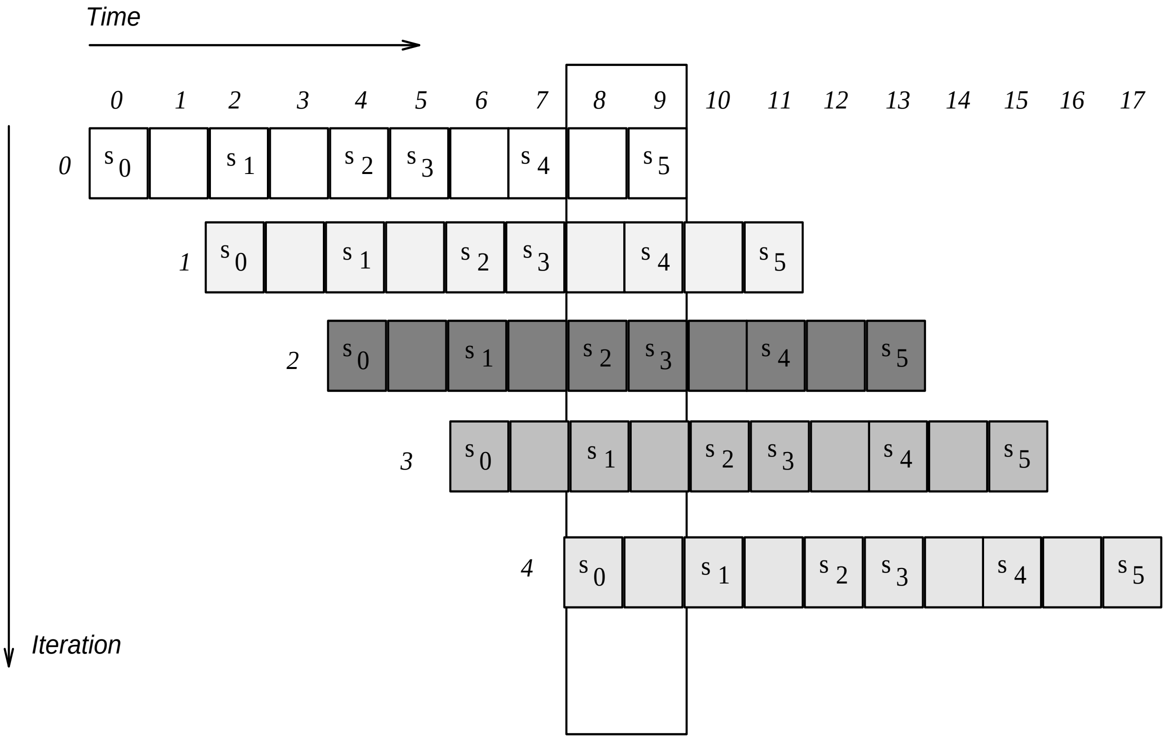
(b) Low Level Code

Schedule A for the Example



s_0	s_1	s_2	s_3	s_4	s_5	Total
5	3	1	1	2	1	13

Schedule B for the Example



Buffer Requirements

s_0	s_1	s_2	s_3	s_4	s_5	Total
5	3	1	1	1	1	12

Logical buffers implemented using

- Rotating Registers [Polycyclic, Cydra Architectures]
- Normal Registers with appropriate register moves.

Summary and Observations on the Example

- There are generally ***many*** software pipelined schedules for a given initiation interval.
- Only a fraction of these schedules also ***satisfy resource requirement*** .
- Of which a few have ***less*** register (buffer) requirements!

Develop a mathematical formulation for the scheduling problem (with buffer requirements).

- Guaranteed to get optimal solution
- May take a long time, though!

Justification for our Approach

- More opportunities for parallelism
  - Array dep. analysis, unrolling, loop fusion
    $\Rightarrow$ More Instruction Level Parallelism
  - ***Result:*** Many *good* schedules are possible. A means is needed to find the *best*.
- Performance-critical applications may need, and should be allowed to attain the ***optimal*** schedule without the need for handcoding.
- Can be used evaluate / improve other other ***practical*** heuristic methods.

ILP approach and the (long) compilation time ***does*** pay off!!

Considerable improvement over heuristic methods!

Features

- ***homogeneous*** or ***heterogeneous*** function units
  - *Homogeneous function units:* Any operation can execute on any function unit.
    Example: **Instruction Issue Units**
  - *Heterogeneous function units:* Operations can execute only on certain type of function unit.
    Example: Adder, Multiplier.
- ***Pipelined*** or ***non-pipelined*** execution units

Definitions

- Loop represented as Data Dependence Graph (DDG)

$$G = \{V, E, m, d\}$$

where

V $=$ Set of instructions in the loop

d $=$ *Latency* of each instruction

E $=$ Set of edges. An edge (u, v) exists if instruction v uses the result of instruction u.

m $=$ *Dependence Distances* of each edge: For edge (u, v) with dependence distance m_{uv}, result produced by u in the current iteration is used by v after m_{uv} iterations.

Iteration period or $Initiation\ Interval\ T$.

$Initiation\ rate$ of the schedule is $\frac{1}{T}$

Schedule Form:

Linear Schedules:

$$t[op, iter] = T \cdot iter + A[op]$$

- $t[op, iter]$ is the time at which iteration $iter$ of operation op begins execution.
- A[op] is the time at which op executes in the first iteration.

Bound on Initiation Interval T

Resource Constraints

- F_r ***Function Units in type*** r Available,
- N_r ***Instructions that execute in type*** r

$$T \geq \max_r \left\lceil \frac{N_r * d_r}{F_r} \right\rceil$$

Recurrence cycles:

The DDG imposes constraint

$$t[v] - t[u] \geq d_u - T \cdot m_{uv}$$

which in turn yields the constraint that

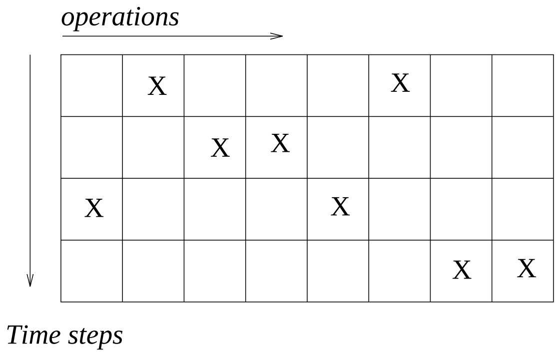
$$T \geq \max_{\forall \ C} \frac{\Sigma \quad d_u}{\Sigma \quad m_{uv}}$$

If T is non-integer, the loop may be unrolled, or T rounded up.

ILP Formulation: Intuition

How to Formulate Resource Constraints?

1. Position in Modulo Reservation Table indicated by a set of 0-1 integer variables. That is, if instruction i is in position t, then $a_{t,i} = 1$.
2. Requirement at time t (for homogeneous, pipelined case) is $a_{t,0} + a_{t,1} + \cdots + a_{t,N-1}$



$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,N-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,N-1} \\ \vdots & \vdots & & \vdots \\ a_{T-1,0} & a_{T-1,1} & \cdots & a_{T-1,N-1} \end{bmatrix}$$

3. Each instruction must be scheduled once in the Modulo Res. Table $a_{0,i} + a_{1,i} + \cdots + a_{T-1,i} = 1$
4. t_i can be written as

$$t_i = k_i \cdot T + [a_{0,i}, \cdots, a_{T-1,i}] \times [0, 1, \cdots, T-1]^{\text{Transpose}}$$

Dependency Constraints

$$t_j - t_i \geq d_i - T \cdot m_{ij} \quad \forall (i,j)$$

Buffer Requirements [NingGao - POPL '93]

$$T \cdot b_i + t_i - t_j \geq T \cdot m_{ij} \quad \forall i \quad \forall (i,j)$$

Objective Function Total buffer requirements

$$\sum_i b_i$$

ILP Formulation

$$\text{minimize} \sum_{i=0}^{N-1} b_i$$

subject to

$$\sum_{i \in \mathcal{I}(r)} a_{t,i} \leq R_r, \quad \forall t \quad \forall r$$

$$T \cdot \mathcal{K} + \mathcal{A}^{Transpose} \times \begin{bmatrix} 0 \\ 1 \\ \vdots \\ T-1 \end{bmatrix} = \mathcal{T}$$

$$T \cdot b_i + t_i - t_j \geq T \cdot m_{ij} \quad \forall i \quad \forall (i,j)$$

$$t_j - t_i \geq d_i - T \cdot m_{ij} \quad \forall (i,j)$$

$$\sum_{t=0}^{T-1} a_{t,i} = 1 \quad \forall i$$

$$b_i, t_i, k_i, a_{t,i} \geq 0, \text{ and are integers}$$

Solution Method

- First use faster *LP* to converge to a closer T.
- Check if *LP* solution is ***feasible*** and has ***minimum*** buffer requirement. If yes, stop there (lucky)!
- Otherwise try to get ILP solution at this T. Almost always do.
- If fail, then increase T by 1 and try again.

  Note: Longer T may take even longer time to complete!

  Why? Because there is nothing about a larger T that *directly* leads to a better schedule.
- **Heuristics to Speed ILP**:
  - Bound the scheduling times of each node by ***ASAP*** and ***ALAP*** schedules

Observations

- The speed of ILP's has improved greatly in recent years and have gained increased use in compilers and synthesis.
- Pugh uses ILP approach for finding exact array dependence analysis **[C ACM '91]**
- Hwang, Lee, and Hsu for scheduling *acyclic* graphs with resource constraints. **[IEEE Transactions on CAD 1991]**
- Kremer, Bixby and Kennedy have proposed an ILP approach for data layout in distributed memory systems. **[PACT '94]**
- Feautrier's fine grain schedule **[Compiler Workshop 1994]**

Conditionals

- Use a predicated execution model of instructions.
- If hardware does not support this, then use *Reverse-If-Conversion*. ***(Warter, Mahlke, Hwu, Rau: PLDI-93)***
- Huff found that 92% of software pipelineable loops in scientific code had no conditionals.

Results 1

Considered loop bodies from ***whetstone***, ***linpack***, ***livermore***, plus SPEC benchmarks ***doduc***, ***fpppp***, ***spice***, and ***tomcatv***.

- 50% of time found schedule within 1 second.
- 91% within 10 seconds
- Loops with more instructions take longer.

  The original loops contained up to 20 nodes.

Results 2

Comparison with Heuristic Methods

- Slack Scheduling ***Huff – PLDI 93***
- FRLC Scheduling ***Wang Et Al. – IJPP 92***
- Modified List Scheduling ***Gasperoni and Schwiegelshohn 92***

Better in	Slack Sch.	FRLC Sch.	Mod. List Sch.
Period	10%	49%	59%
Buffer	15%	63%	13%

Related Work

- Production Compilers: FPS, Cydrix, HP-PA
- ***Lam***'s Modulo Scheduling
- ***Huff***'s Slack Scheduling
- ***Ebcioğlu Et Al.*** Enhanced Pipeline Percolation Scheduling
- ***Aiken-Nicolau***'s Perfect Pipelining
- ***Feautrier***'s Fine Grain Scheduling
- ***Wang Et Al.*** FRLC Scheduling
- ***Gasperoni-Schwiegelshohn***'s Modified List Scheduling
- ***Eichenberger, Davidson, Abraham***'s Modulo Scheduling
- ***Ning-Gao***'s Buffer Minimization

Future Work

- Handle complex resource usage – pipelines with ***structural hazards***
- Simultaneous ***Scheduling and Mapping***
- Experiment many more benchmarks