Software Pipelining with Register Allocation and Spilling

Jian Wang, Andreas Krall
M. Anton Ertl, Christine Eisenbeis

Institute of Computer Language
Technical University of Vienna
Vienna, Austria

Outline

1. Our Loop Scheduling Framework
2. Register Requirement Graph(RRG)
3. Register-Pressure-Sensitive Scheduling
4. Register Spilling
5. Algorithms
6. Preliminary Experimental Results

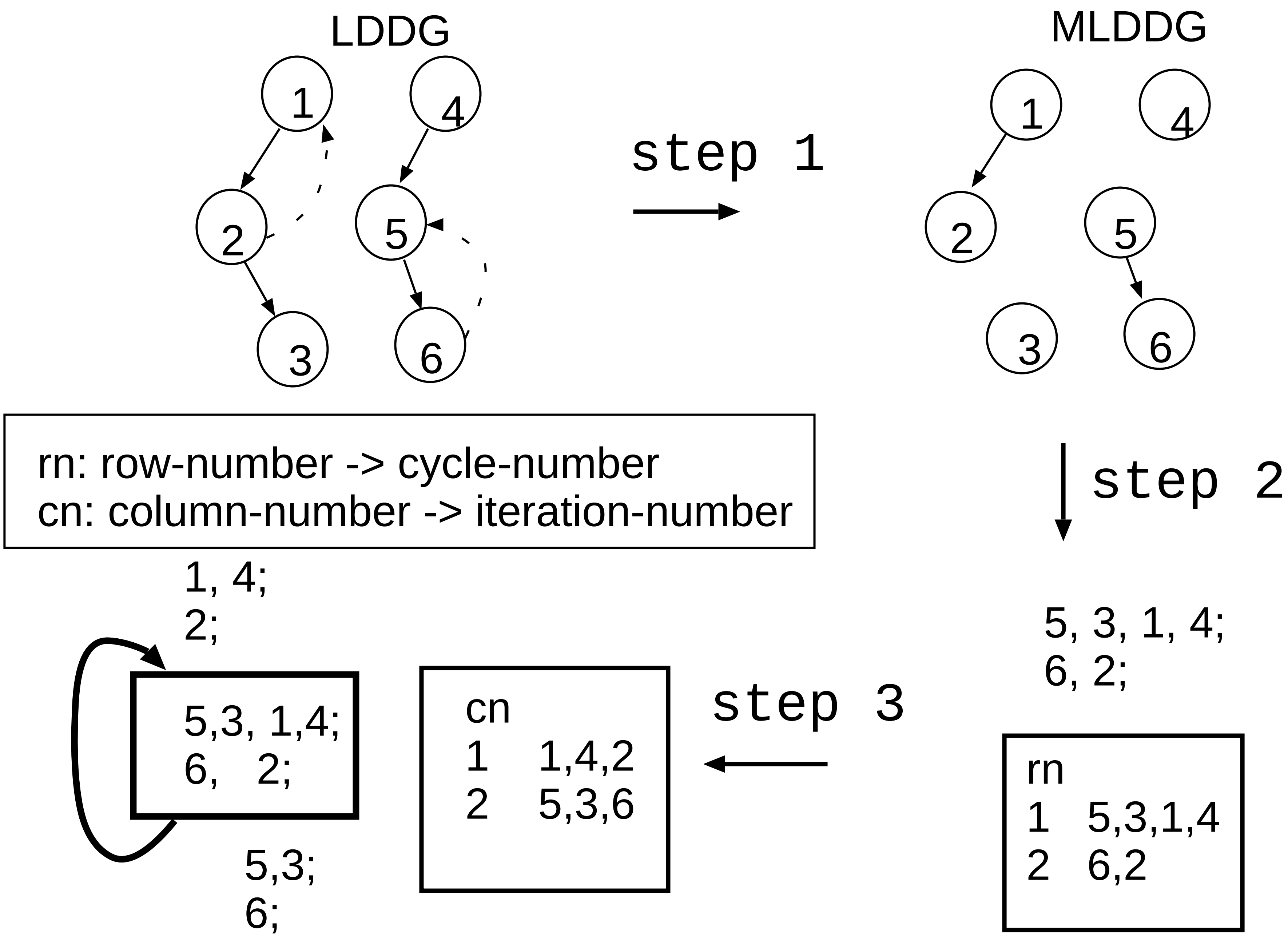
Software Pipelining

A loop scheduling technique, overlap the execution of operations from different iterations.

```
1          1                                              1
2          2; 1                                           2; 1
3          3; 2; 1                                        3; 2; 1
4          4; 3; 2; 1
               4; 3; 2; 1                                 4; 3; 2; 1     n-3
                   4; 3; 2; 1
                       4; 3; 2; 1                             4; 3; 2
                           4; 3; 2                                4; 3
                               4; 3                                   4
                                   4 ......
 n
```

II=1

DEcomposed Software Pipelining



Representing the variable's lifetime by rn and cn

Suppose v defined by op_d and used by op_u, then its lifetime, $lt(v)$, can be computed as

$$lt(v) = II*(cn(op_u)-cn(op_d))+rn(op_u)-rn(op_d)$$

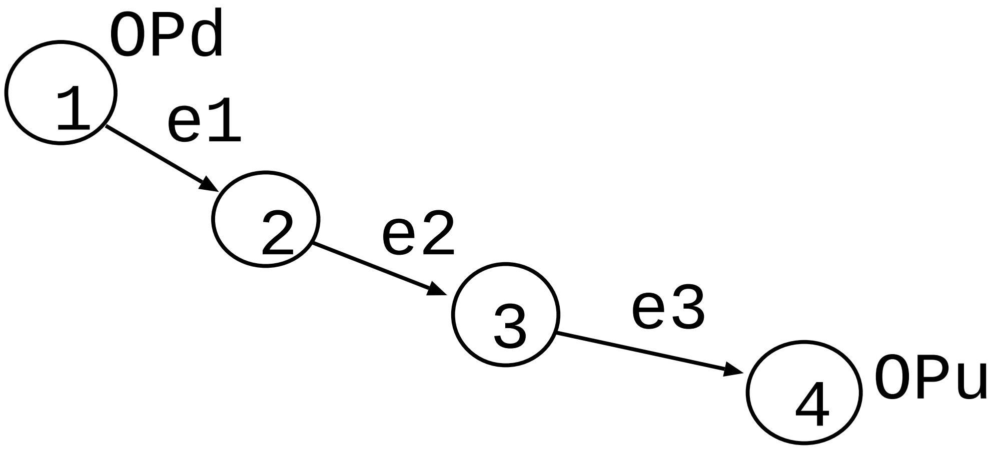
- $rn(op_u) - rn(op_d) < II$.

- Register coloring is beyond the scope of this paper.

- We only pay attention to v whose $lt(v)$ is greater than II.

□ v's lifetime mainly determined by $cn(op_u) - cn(op_d)$.

How to estimate $cn(op_u) - cn(op_d)$ before or during software pipelining?



Let $dcn(op_i, op_j) = cn(op_i) - cn(op_j)$, then

$$dcn(op_u, op_d) =$$
$$dcn(op_u, op_3) + dcn(op_3, op_2) + dcn(op_2, op_d)$$

- The less is the $dcn(op_j, op_i)$ where (op_i, op_j) is a dependence edge, the less is the register requirement.

How to estimate $dcn(op_j, op_i)$ where (op_i, op_j) is a dependence edge?

Let $e = (op_i, op_j)$, $\lambda(e)$ be the dependence distance, $\delta(e)$ the delay:

$$rn(op_j) + II * cn(op_j) - (rn(op_i) + II * cn(op_i)) \\ \geq \delta(e) - II * \lambda(e)$$

$$cn(op_j) - cn(op_i) \geq -\lambda(e) + \lceil \frac{\delta(e) - (rn(op_j) - rn(op_i))}{II} \rceil$$

$$dcn(op_j, op_i) = -\lambda(e) + \lceil \frac{\delta(e) - (rn(op_j) - rn(op_i))}{II} \rceil$$

- $1 \leq rn(op) \leq II, \forall op \in LDDG$

- If $rn(op_j) - rn(op_i) \geq \delta(e)$, then $dcn(op_j, op_i)$ takes the minimum value, $-\lambda(e)$.

Register Requirement Graph (RRG)

Let $LDDG = (N, E, \lambda, \delta)$, then $RRG = (N, E, dcn)$.

- RRG can be initially defined before software pipelining.

- RRG can be dynamically modified during software pipelining.

□ RRG can dynamically estimate the register requirement of each variable.

RPS Scheduling

• In the second step of Decomposed Software Pipelining, use RRG as a basis.

• Use the following two heuristics to guide the scheduling process:

1. For some edges ($e = (op_i, op_j)$) removed in the first step, make them satisfy $rn(op_j) - rn(op_i) \geq \delta(e)$ so that $dcn(op_j, op_i) = -\lambda(e)$;

2. Define a new parameter, called *register-benefit*, and use it to compute the scheduling priorities of operations in the data ready set.

Register Spilling

- Spilling decision is conventionally made after scheduling.

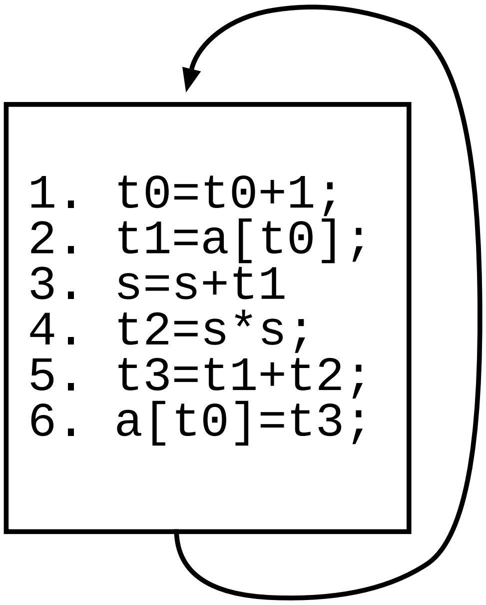
- Our motivations:

1. Each variable has a lower bound on its life-time, which is the length of the longest path from its definition to its use;

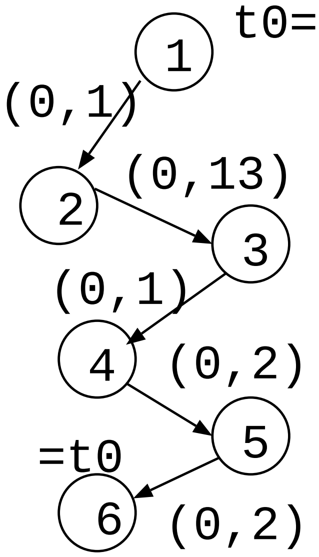
2. Spill some use(s) of a variable may reduce the lower bound.

- Use RRG to define *spilling-benefit* of a use, which is the number of reduced registers per inserted load/store operation.

- Spill the use(s) with larger spilling-benefit.

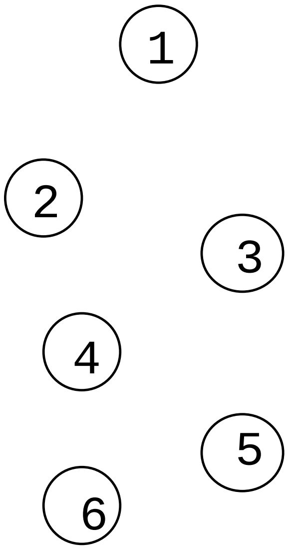


(1) The loop

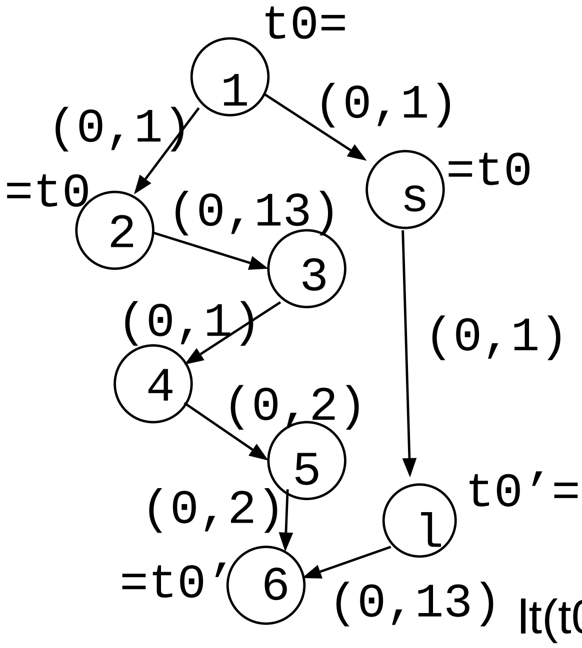


(2) LDDG

lt(t0)=19

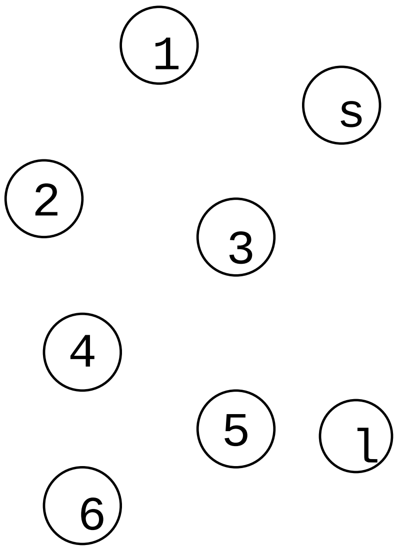


(3) MLDDG



lt(t0)=1
lt(t0')=13

(4) LDDG after spilling



(5) MLDDG after spilling

Algorithms

1. RPS without spilling: integrate our two heuristics into the list scheduling approach.

2. RPS with spilling: integrate the spilling checking phase into the above algorithm.

3. Software pipelining with a limited number of registers:

3.1 use RPS with spilling as the framework;

3.2 Let K_{est} be the register requirement estimated by RRG, K_o the number of registers available:

If $K_{est} - K_o \leq N_0$, then call RPS with spilling;

If $K_{est} - K_o > N_0$, increase II (N_0 may be modified);

3.3 N_0 reflects the fact that two variables may use the same registers if their live ranges do not interfere.

Table 1. Register Requirement for Three Scheduling Approaches

Example	L	MII	DESP	RPS without Spilling	RPS with Spilling
1	20	2	27	23	21
2	22	3	39	32	30
3	17	1	30	30	30
4	18	3	21	18	18
5	16	1	29	29	29
6	17	2	27	25	25

note: L = the length of the longest dependence path in the loop body;
MII = the minimum initiation interval.

An improvement of up to 23%

Table 2. Software Pipelining with a Limited Number of Registers (The Initiation Interval / The speedup)

Ko	Example 1	2	3	4	5	6
8	7/2.86	16/1.38	7/2.43	8/2.25	5/3.2	9/1.89
16	3/6.67	8/2.75	3/5.67	4/4.5	2/8.00	4/4.25
32	2/10.00	3/7.33	1/17.00	3/6.00	1/16.00	2/8.50

note: Ko = the number of available machine registers.
speedup = L/II, II is the initiation interval.

An average speedup of 2.34 for Ko=8

test	VAMtwo ms	VAMone scaled	Aquarius scaled	Parma scaled
det. append	0.25	26.1	-	-
naive reverse	4.17	20.0	13.9	25.6
quicksort	6.00	10.3	16.8	28.0
8-queens	65.4	13.5	-	-
serialize	3.90	6.84	4.26	16.4
differentiate	1.14	8.14	7.11	14.6
query	41.7	9.70	10.3	13.2
bucket	247	5.24	-	-
permutation	4023	5.34	-	-