

Single-dimension Software Pipelining for Multi-dimensional Loops

Hongbo Rong

Zhizhong Tang

Alban Douillet

Ramaswamy Govindarajan

Guang R. Gao

Presented by: Hongbo Rong



IFIP Tele-seminar June 1, 2004



清华大学
Tsinghua University

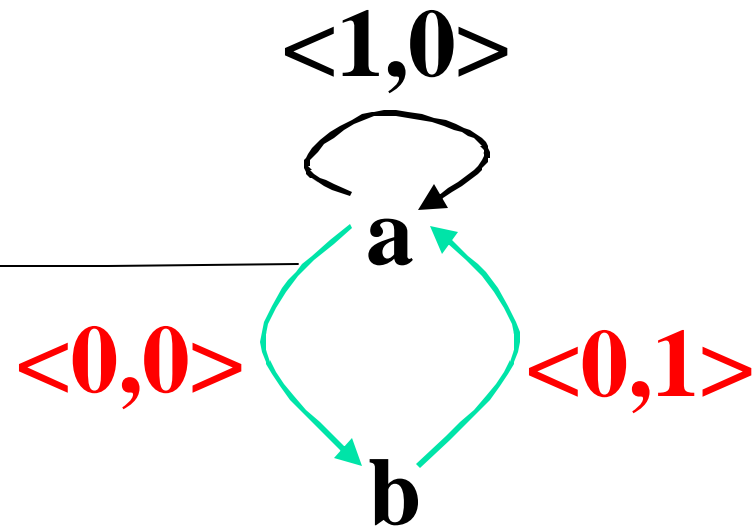
Introduction

- Loops and software pipelining are **important**
- Innermost loops are **not enough** [Burger&Goodman04]
 - *Billion-transistor architectures* tend to have much more parallelism
- Previous methods for scheduling multi-dimensional loops are meeting **new challenges**

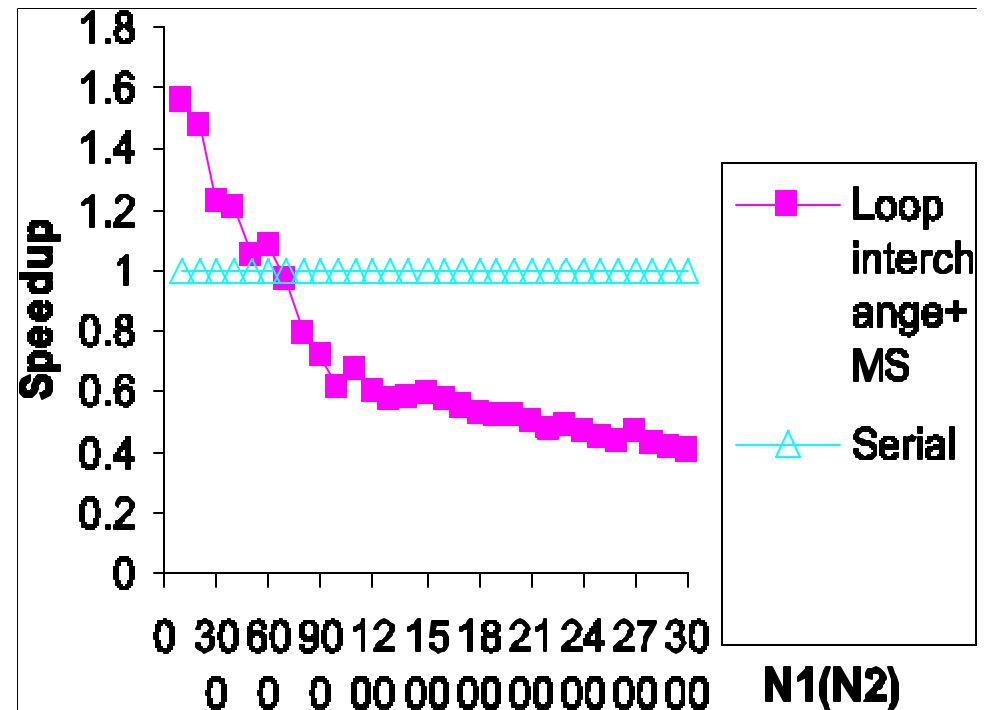
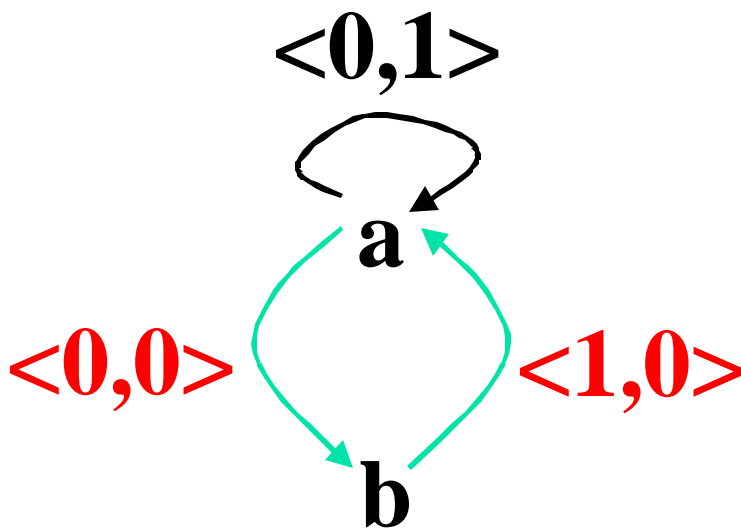
Motivating Example

```
int U[N1+1][N2+1], V[N1+1][N2+1];  
L1: for (i1=0; i1<N1; i1++) {  
L2:   for (i2=0; i2<N2; i2++) {  
      a: U[i1+1][i2]=V[i1][i2]+ U[i1][i2];  
      b: V[i1][i2+1]=U[i1+1][i2];  
    }  
  }
```

A strong cycle in the
inner loop: No
parallelism



Loop Interchange Followed by Modulo Scheduling of the Inner Loop

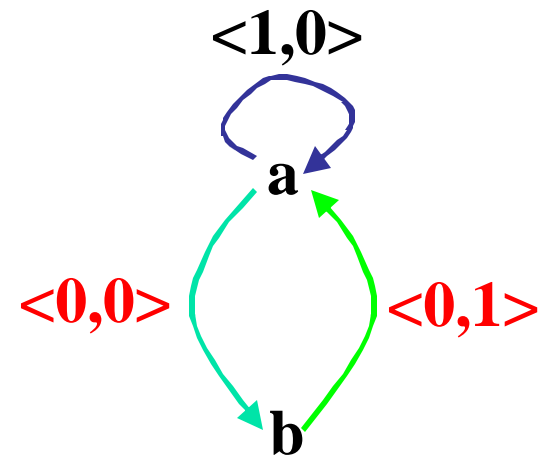


- Why not select a better loop to software pipeline?
 - Which and how?

Starting from A Naïve Approach

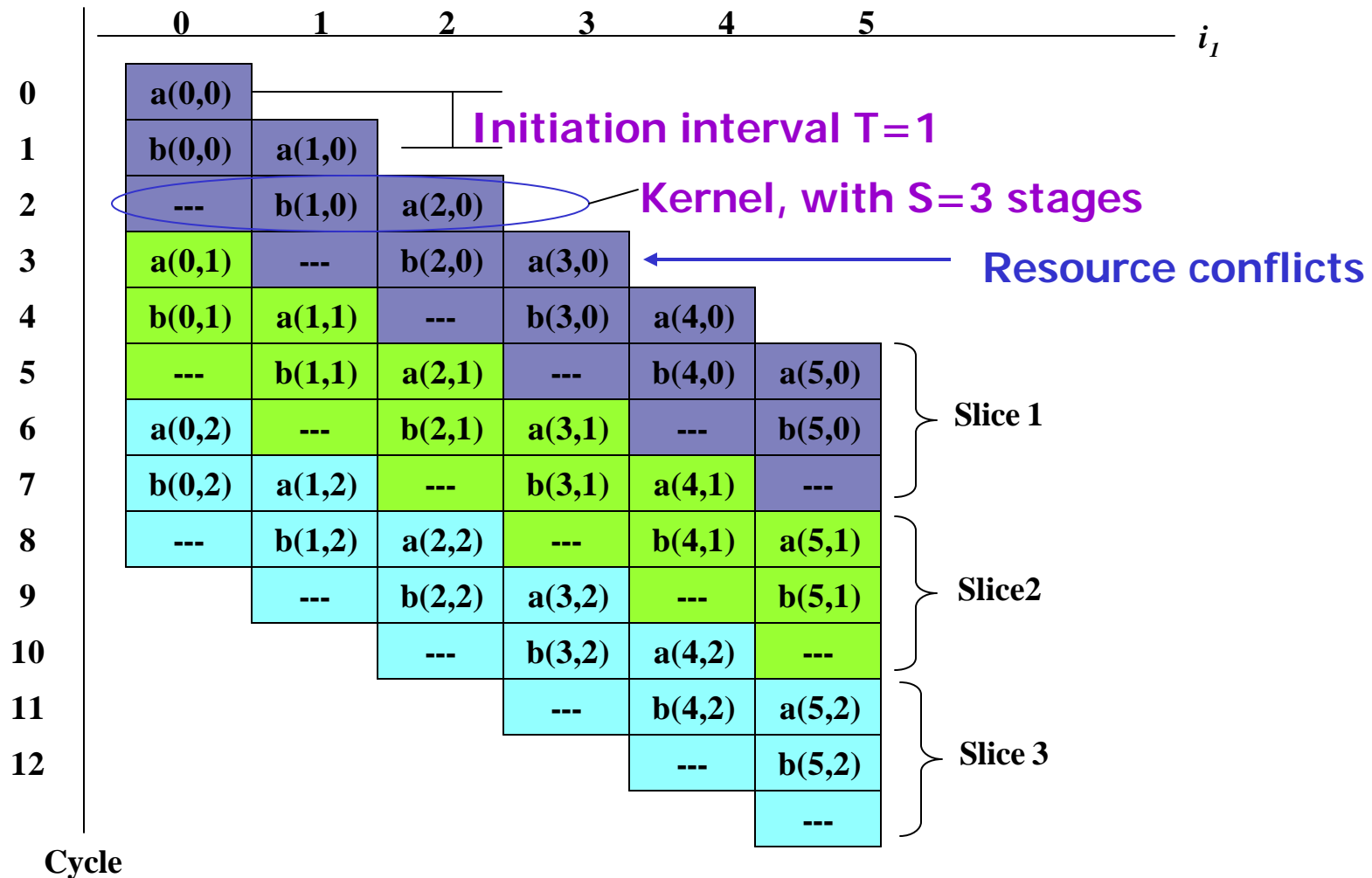
	0	1	2	3	4	5	i_1
0	a(0,0)						
1	b(0,0)	a(1,0)					
2	---	b(1,0)	a(2,0)				
3	a(0,1)	---	b(2,0)	a(3,0)			
4	b(0,1)	a(1,1)	---	b(3,0)	a(4,0)		
5	---	b(1,1)	a(2,1)	---	b(4,0)	a(5,0)	
6	a(0,2)	---	b(2,1)	a(3,1)	---	b(5,0)	
7	b(0,2)	a(1,2)	---	b(3,1)	a(4,1)	---	
8	---	b(1,2)	a(2,2)	---	b(4,1)	a(5,1)	
9		---	b(2,2)	a(3,2)	---	b(5,1)	
10			---	b(3,2)	a(4,2)	---	
11				---	b(4,2)	a(5,2)	
12					---	b(5,2)	

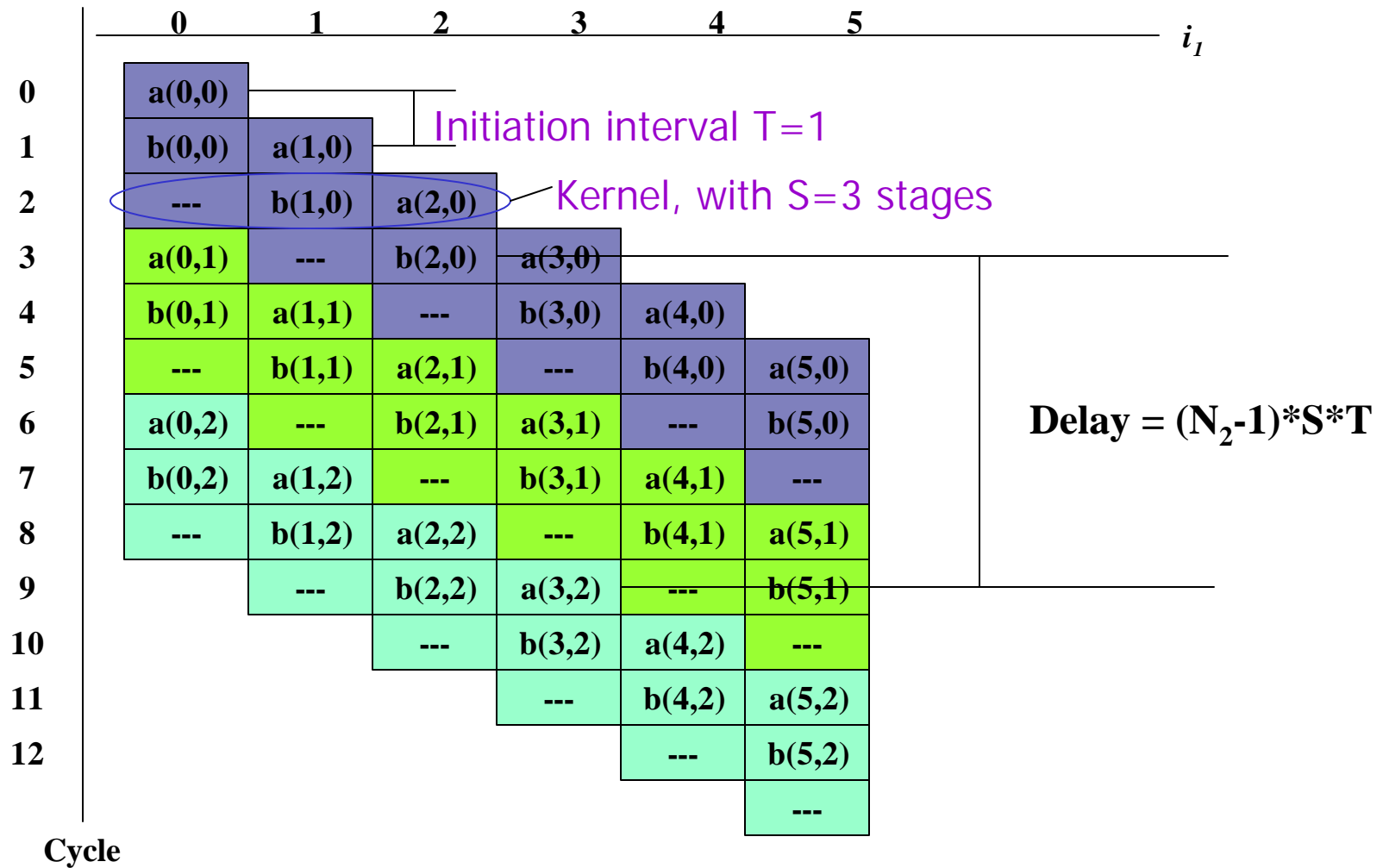
Resource conflicts



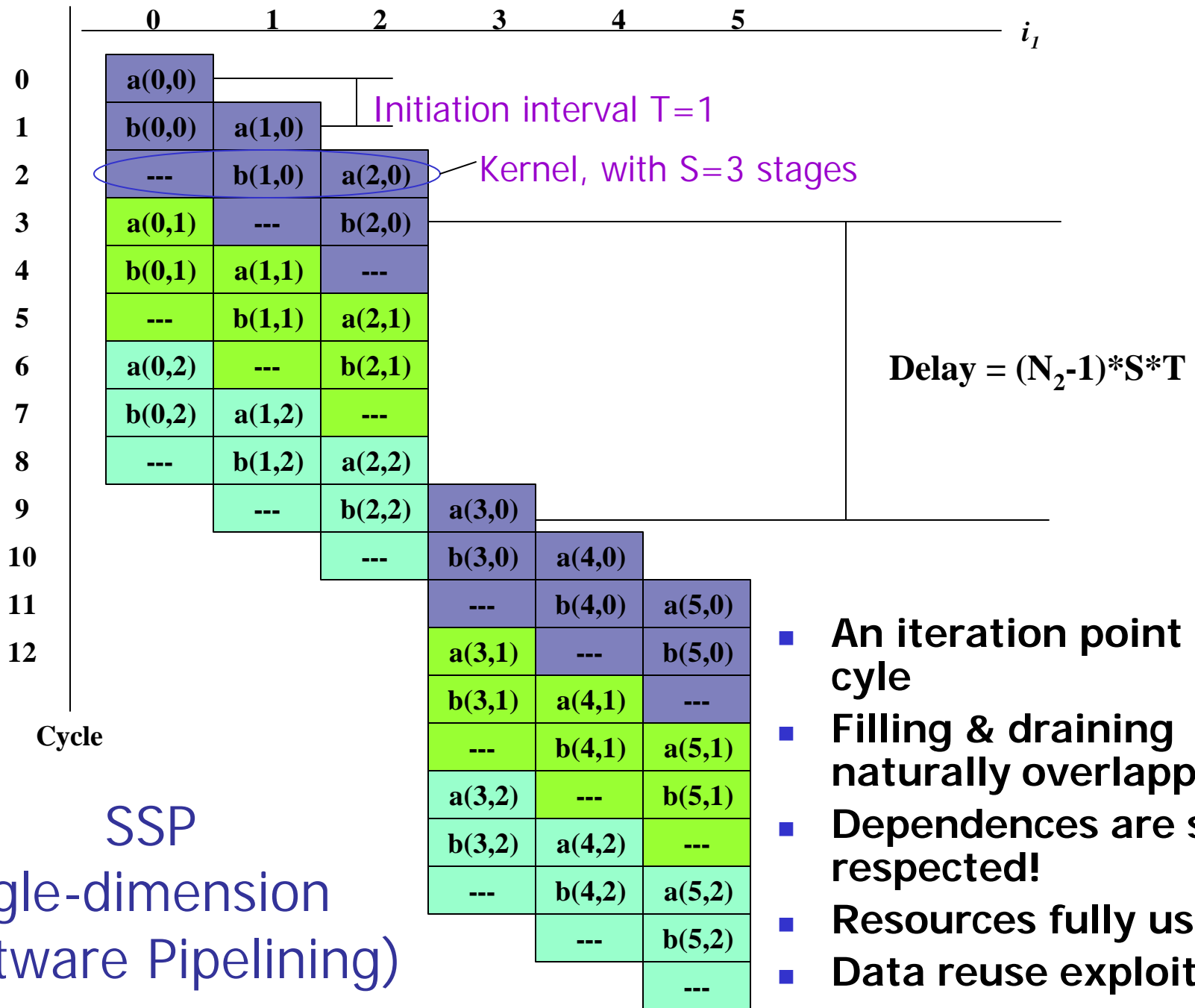
2 function units
a: 1 cycle
b: 2 cycles
 $N_2 = 3$

Looking from Another Angle





SSP
 (Single-dimension
 Software Pipelining)



SSP
(Single-dimension
Software Pipelining)

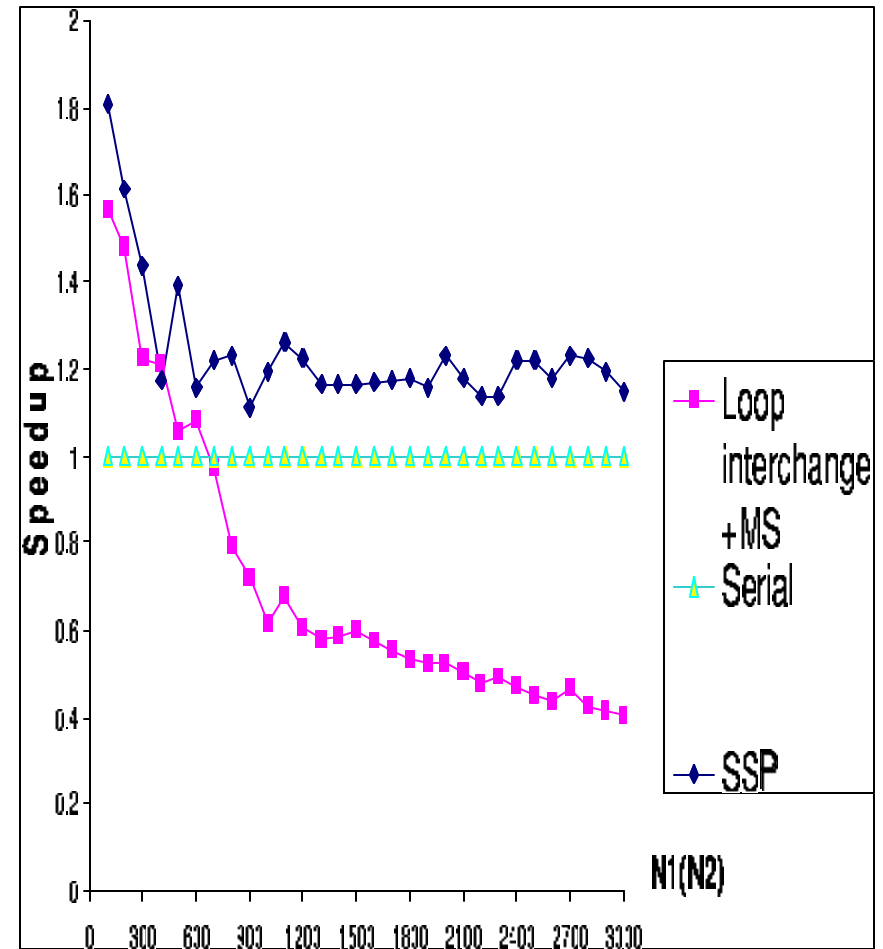
- An iteration point per cycle
- Filling & draining naturally overlapped
- Dependences are still respected!
- Resources fully used
- Data reuse exploited!

Loop Rewriting

```

int U[N1+1][N2+1], V[N1+1][N2+1];
L1': for (i1=0; i1<N1; i1+=3) {
    b(i1-1, N2-1) a(i1, 0)
    b(i1, 0) a(i1+1, 0)
    b(i1+1, 0) a(i1+2, 0)
L2': for (i2=1; i2<N2; i2++) {
    a(i1, i2) b(i1+2, i2-1)
    b(i1, i2) a(i1+1, i2)
    b(i1+1, i2) a(i1+2, i2)
}
}
b(i1-1, N2-1)

```



Outline

- Motivation
- Problem Formulation & Perspective
- Properties
- Extensions
- Current and Future work
- Code Generation and experiments

Problem Formulation

Given a loop nest L composed of n loops L_1, \dots, L_n , identify the most profitable loop level L_x with $1 \leq x \leq n$, and software pipeline it.

- Which loop to software pipeline?
- How to software pipeline the selected loop?
 - How to handle the **n-D dependences**?
 - How to enforce **resource constraints**?
 - How can we guarantee that **repeating patterns** will definitely appear?

Single-dimension Software Pipelining

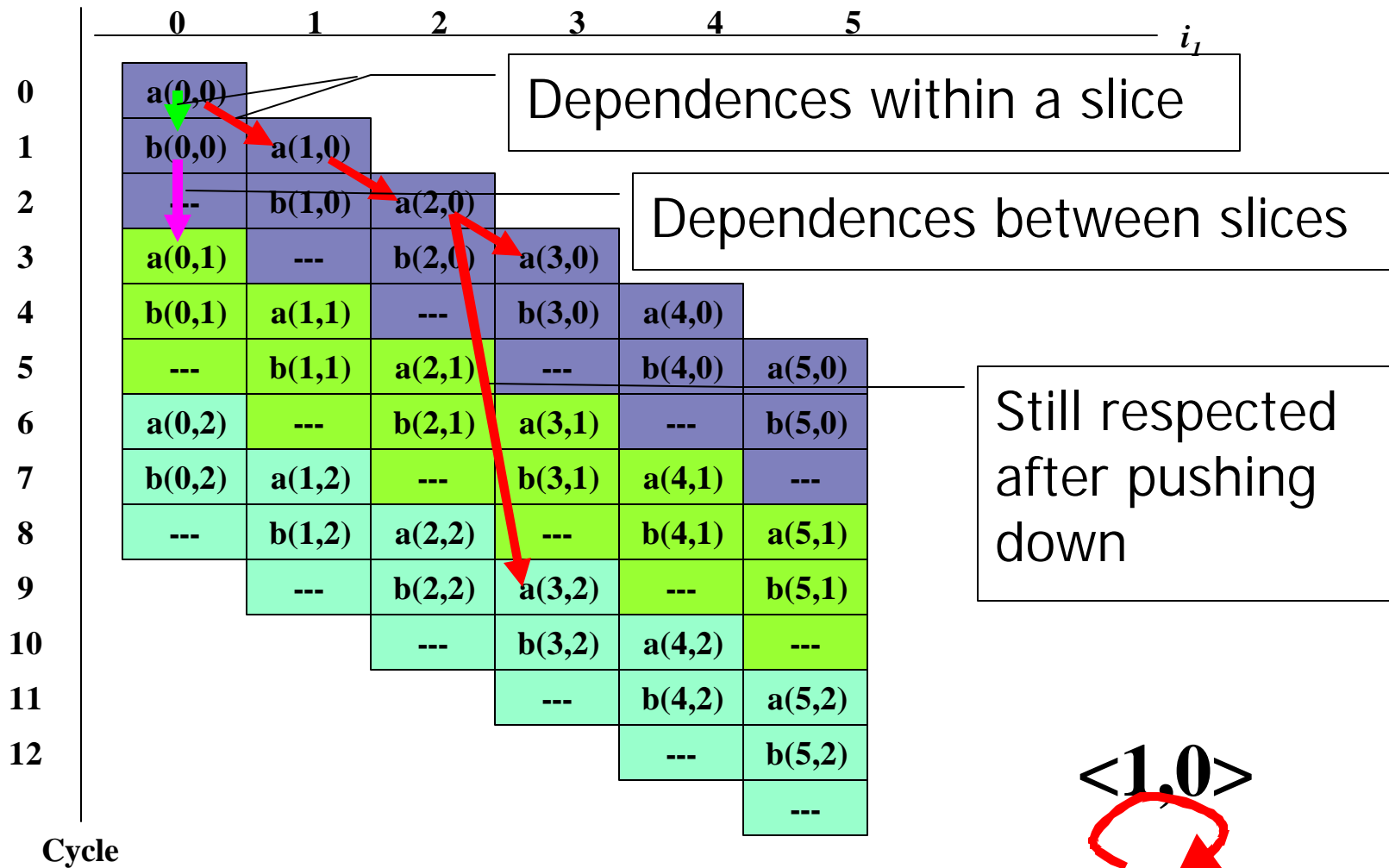
- A resource-constrained scheduling method for loop nests
- Can schedule at an arbitrary level
- Simplify n-D dependences to 1-D
- 3 steps
 - Loop Selection
 - Dependence Simplification and 1-D Schedule Construction
 - Final schedule computation

Perspective

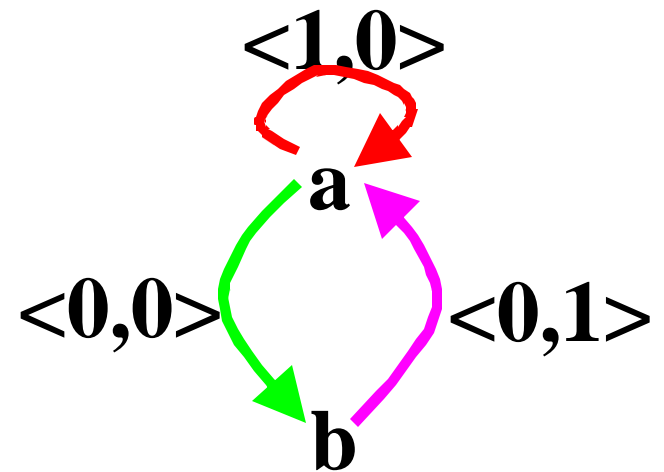
- Which loop to software pipeline?
 - Most profitable one in terms of parallelism, data reuse, or others
 - How to software pipeline the selected loop?
 - Allocate iteration points to slices
 - Software pipeline each slice
 - Partition slices into groups
 - Delay groups until resources available
- Enforce resource constraints in two steps**

Perspective (Cont.)

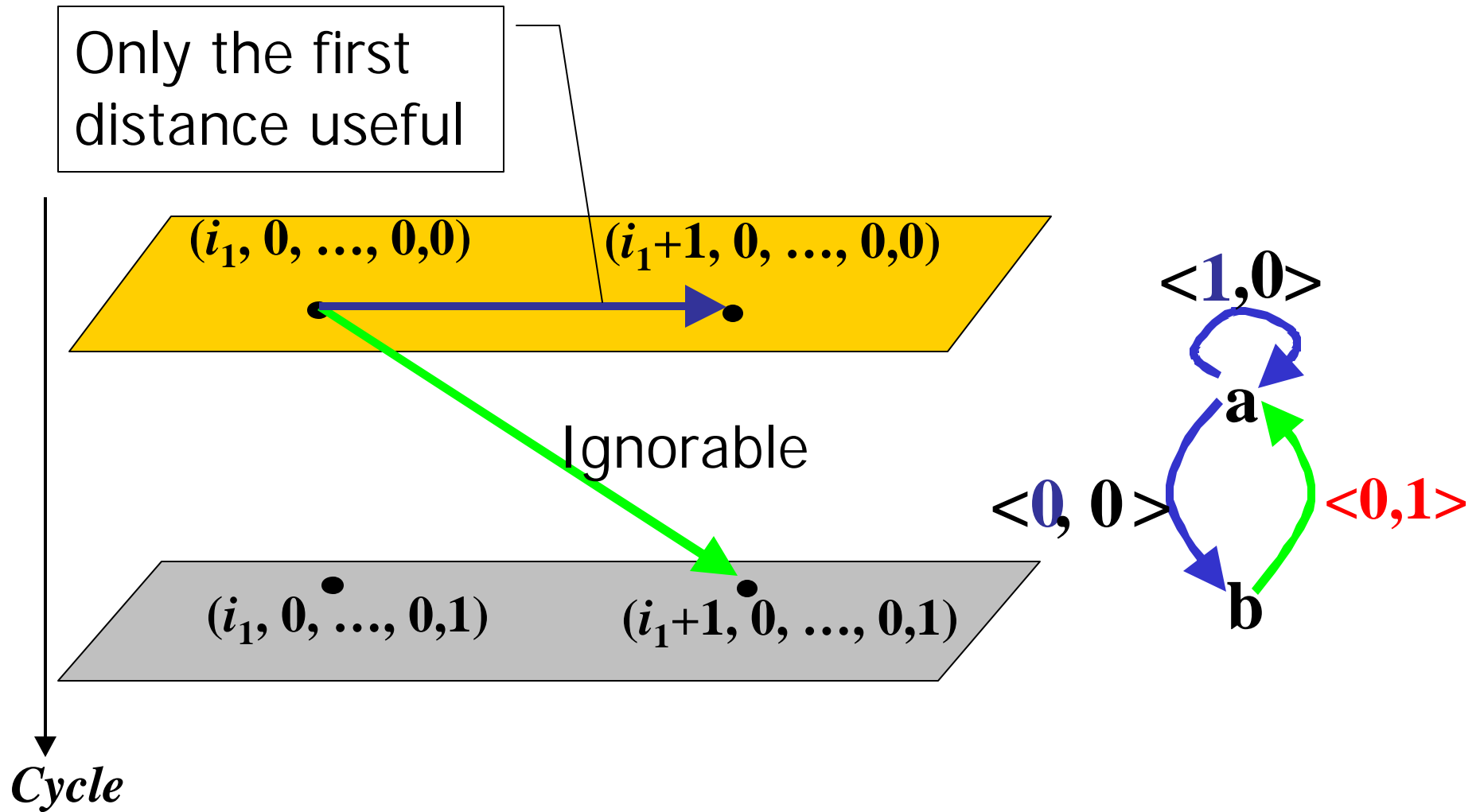
- How to handle dependences?
 - If a dependence is respected before pushing-down the groups, it will be respected afterwards
 - Simplify dependences from n-D to 1-D



How to handle dependences?



Simplify n-D Dependences

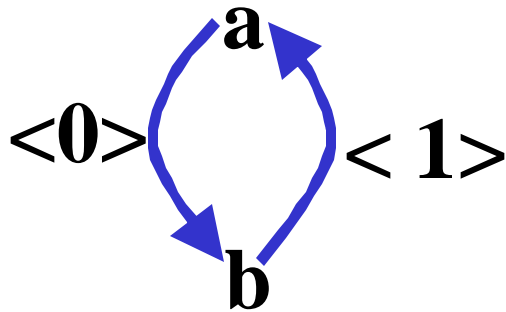


Step 1: Loop Selection

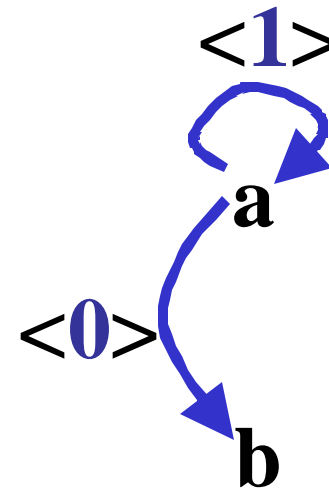
- Scan each loop.
- Evaluate parallelism
 - Recurrence Minimum II (RecMII) from the cycles in 1-D DDG
- Evaluate data reuse
 - average memory accesses of an $S \times S$ tile from the future final schedule (optimized iteration space).

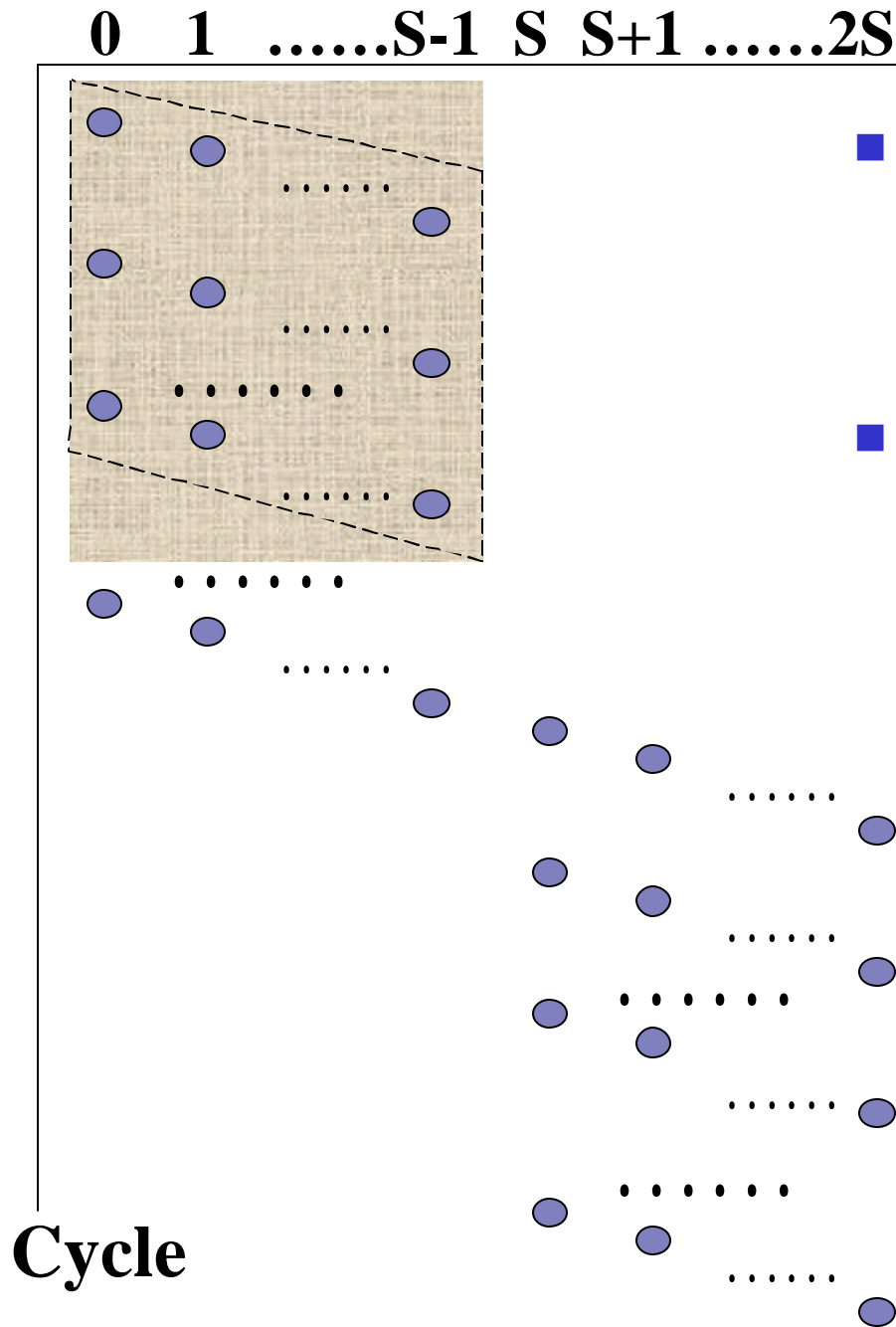
Example: Evaluate Parallelism

Inner loop:
RecMII=3



Outer loop:
RecMII=1



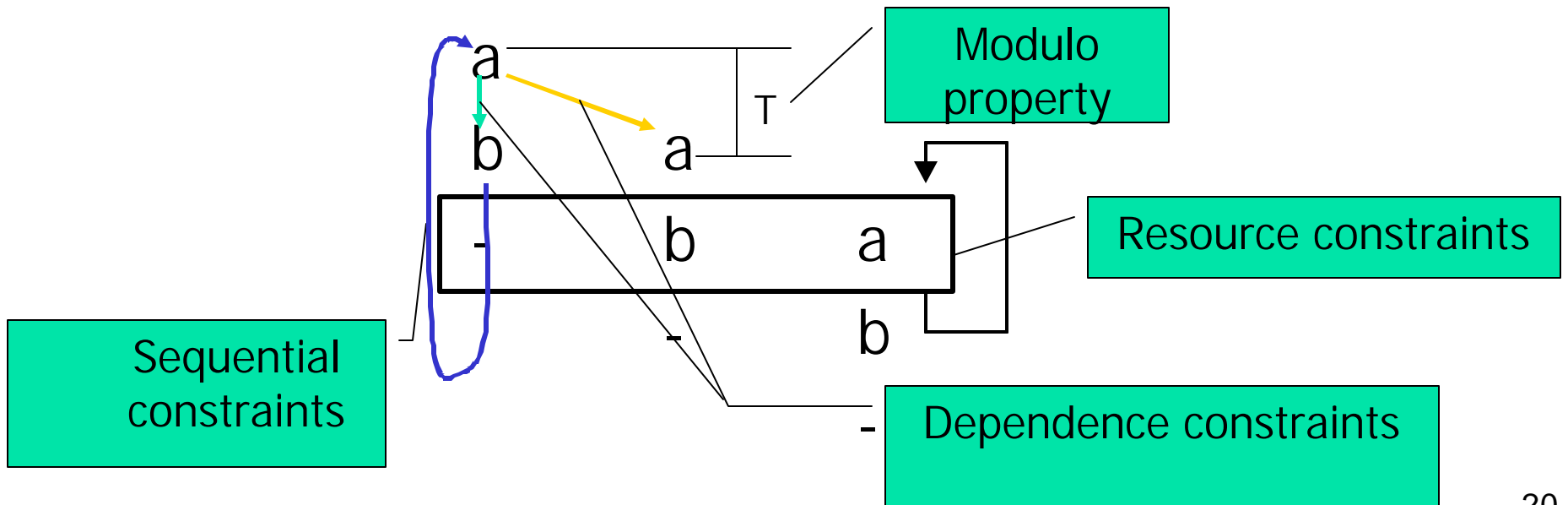
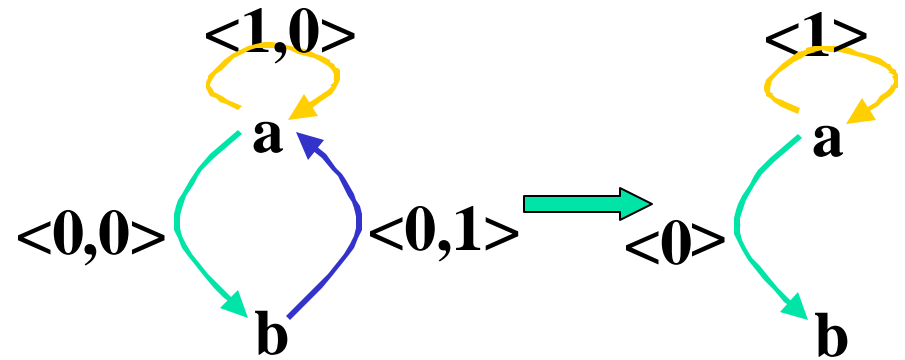


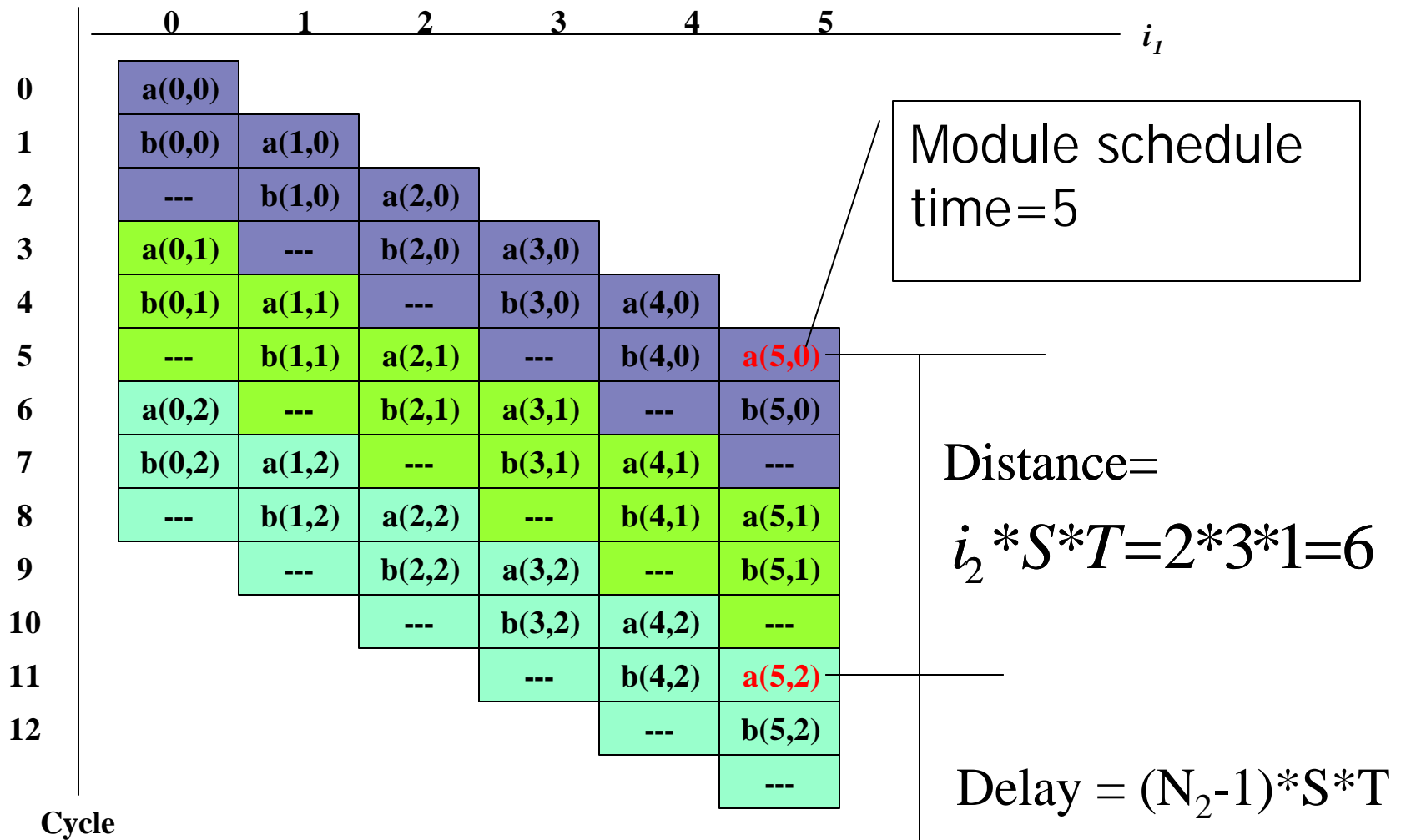
- Symbolic parameters
 - S: total stages
 - l*: cache line size
- Evaluate data reuse [WolfLam91]
 - Localize
 - space = span{(0,1), (1,0)}
 - Calculate equivalent classes for temporal and spatial reuse space
 - average accesses = 2/*l*

Evaluate Data Reuse

Step 2: Dependence Simplification and 1-D Schedule Construction

- Dependence Simplification
- 1-D schedule construction





Final Schedule
 Computation
 Example: a(5,2)

Module schedule
 time=5

Distance=
 $i_2 * S * T = 2 * 3 * 1 = 6$

Delay = $(N_2 - 1) * S * T$
 $= (3 - 1) * 3 * 1 = 6$

Final schedule
 time=5+6+6=17

Step 3: Final Schedule Computation

For any operation o , iteration point $I=(i_1, i_2, \dots, i_n)$,

$$f(o, I) = s(o, i_1)$$

Modulo schedule time

$$+ \sum_{\forall x, 2 \leq x \leq n} (i_x * \prod_{\forall y, x \leq y \leq n+1} N_y) * S * T$$

Distance between $o(i_1, 0, \dots, 0)$ and $o(i_1, i_2, \dots, i_n)$

$$+ \lceil i_1 / S \rceil * \left(\prod_{\forall x, 2 \leq x \leq n} N_x - 1 \right) * S * T$$

Delay from pushing down

Outline

- Motivation
- Problem Formulation & Perspective
- Properties
- Extensions
- Current and Future work
- Code Generation and experiments

Correctness of the Final Schedule

- Respects the original n-D dependences
 - Although we use 1-D dependences in scheduling
- No resource competition
- Repeating patterns definitely appear

Efficiency of the Final Schedule

- Schedule length \leq the innermost-centric approach
 - One iteration point per T cycles
 - Draining and filling of pipelines naturally overlapped
- Execution time: even better
 - Data reuse exploited from outermost and innermost dimensions

Relation with Modulo Scheduling

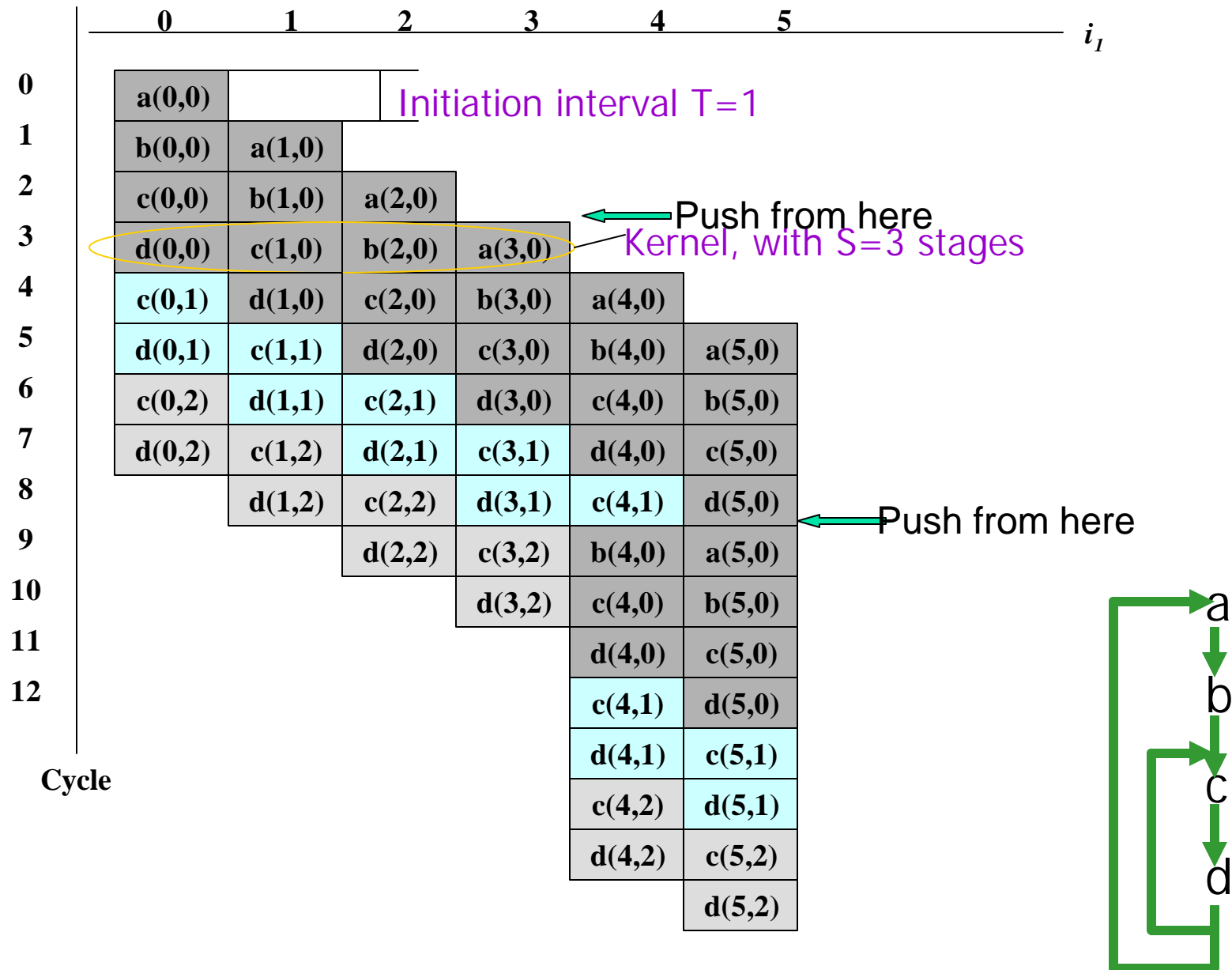
- The classical MS for single loops is subsumed as a special case of SSP
 - **No sequential constraints**
 - **$f(o, I) = \text{Modulo schedule time } (s(o, i_1))$**

Outline

- Motivation
- Problem Formulation & Perspective
- Properties
- Extensions
- Current and Future work
- Code Generation and experiments

SSP for Imperfect Loop Nest

- Loop selection
- Dependence simplification and 1-D schedule construction
 - Sequential constraints
- Final schedule

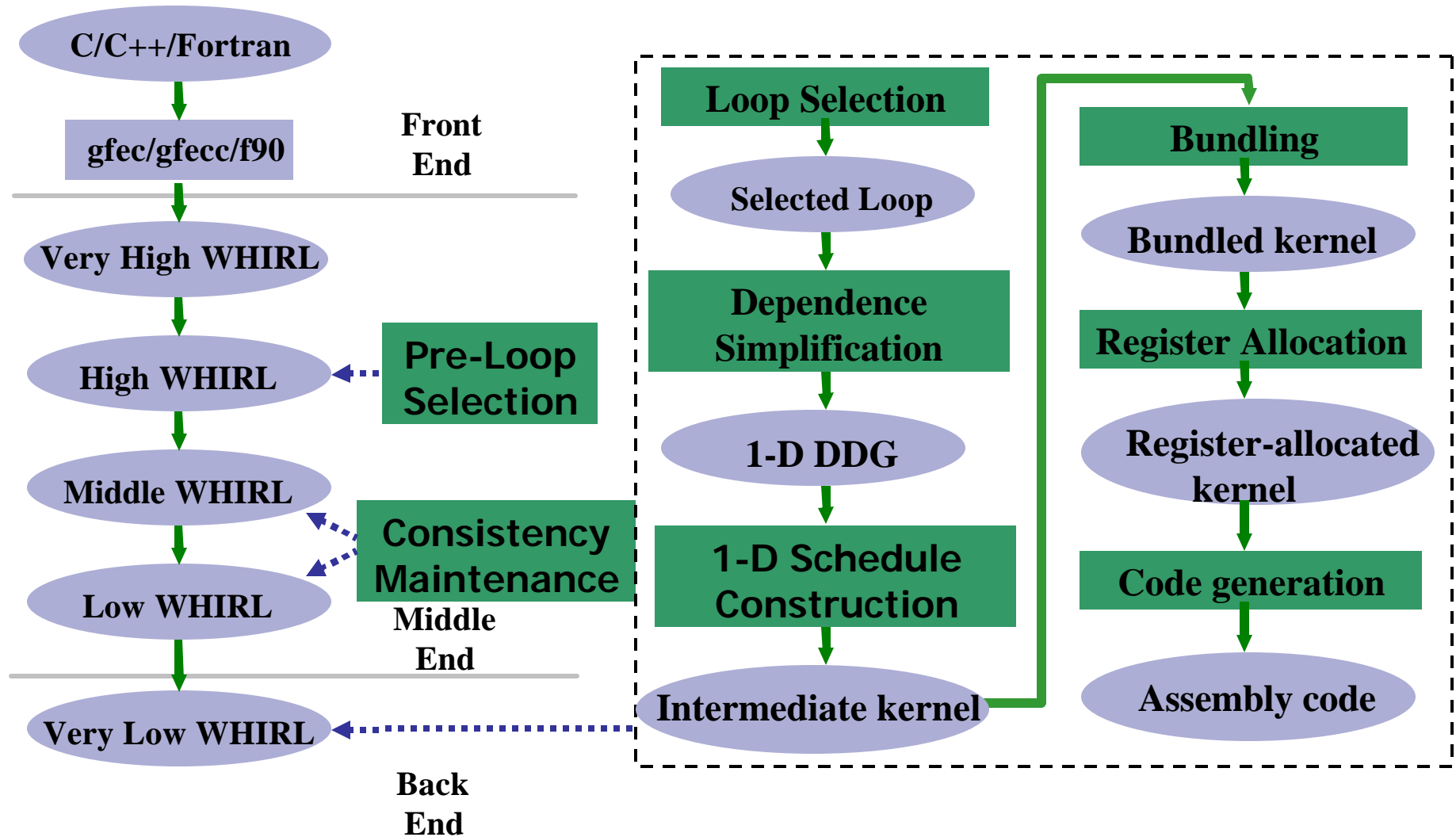


SSP for Imperfect Loop Nest (Cont.)

Outline

- Motivation
- Problem Formulation & Perspective
- Properties
- Extensions
- Current and Future work
- Code Generation and experiments

Compiler Platform Under Construction

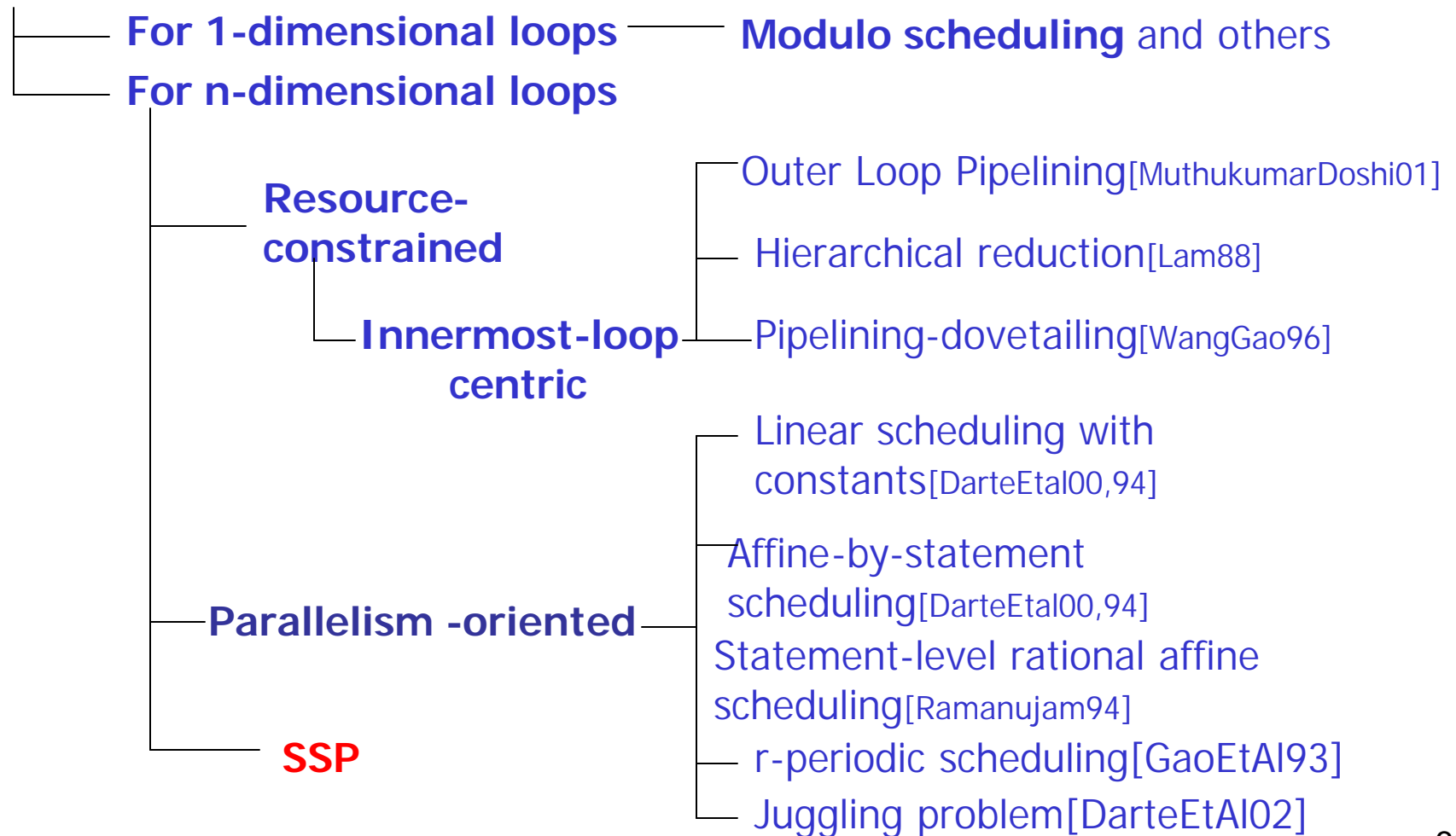


Current and Future Work

- Register allocation
- Implementation and evaluation
- Interaction and comparison with pre-transforming the loop nest
 - Unroll-and-jam
 - Tiling
 - Loop interchange
 - Loop skewing and Peeling
 -

An (Incomplete) Taxonomy of Software Pipelining

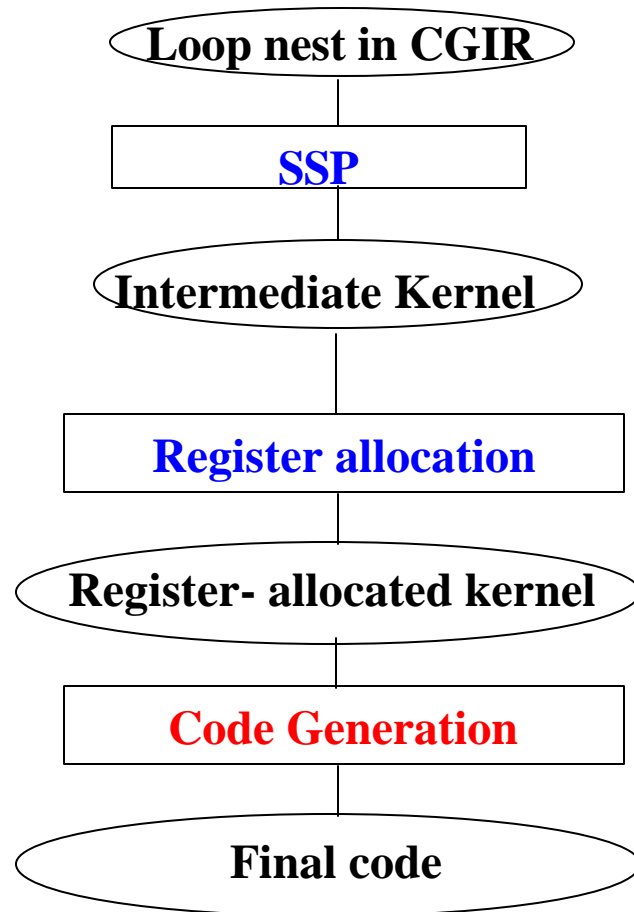
Software Pipelining



Outline

- Motivation
- Problem Formulation & Perspective
- Properties
- Extensions
- Current and Future work
- Code Generation and experiments

Code Generation



Problem Statement

Given an register allocated kernel generated by SSP and a target architecture, generate the SSP final schedule, while reducing code size and loop control overheads.

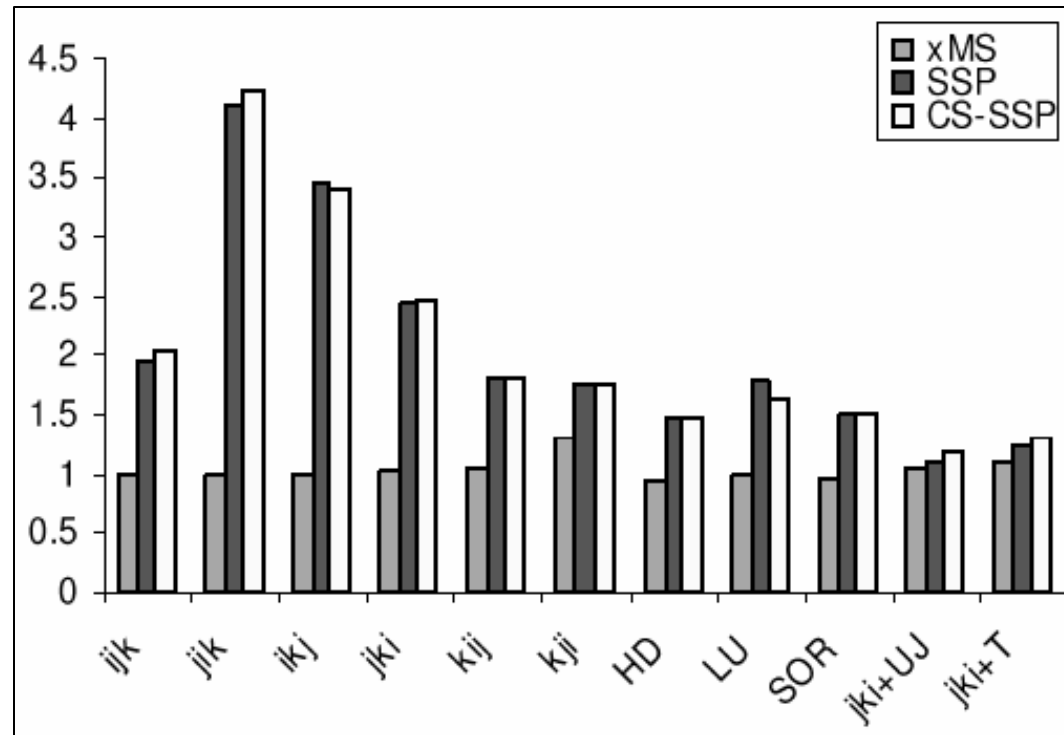
Code Generation: Challenges

- Multiple repeating patterns
 - Code emission algorithms
- Register Assignment
 - Lack of multiple rotating register files
 - Mix of rotating registers and static register renaming techniques
- Loop and drain control
 - Predicated execution
 - Loop counters
 - Branch instructions
- Code size increase
 - Code compression techniques

Experiments: Setup

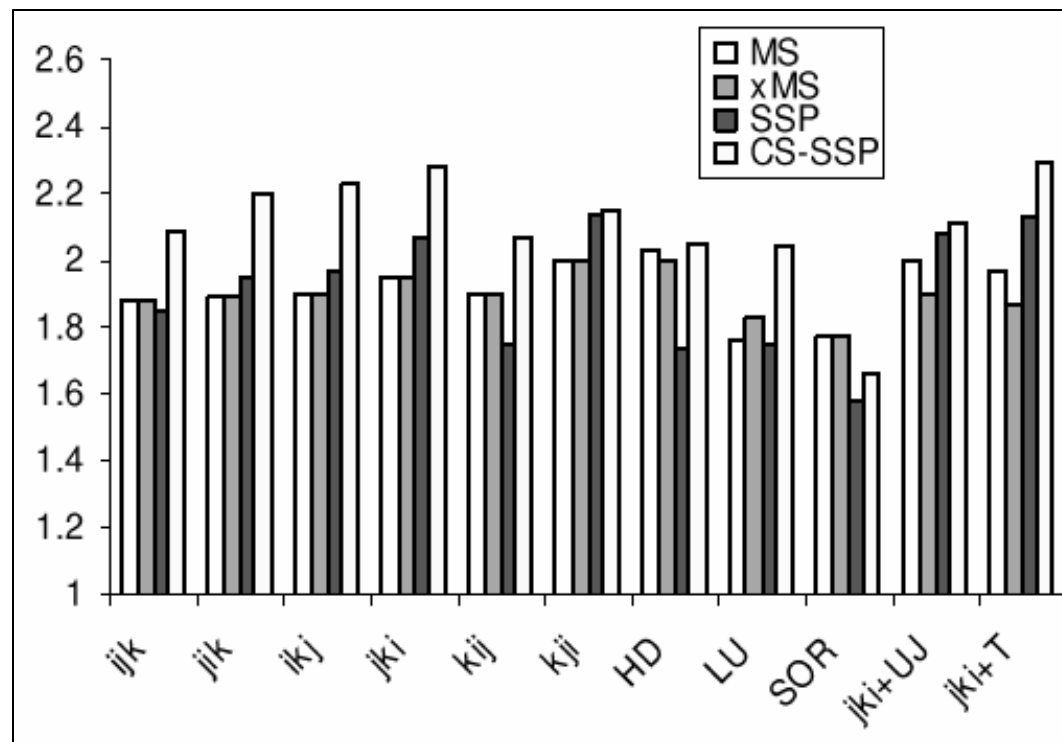
- Stand-alone module at assembly level.
- Software-pipelining using Huff's modulo-scheduling.
- SSP kernel generation & register allocation by hand.
- Scheduling algorithms: MS, xMS, SSP, CS-SSP
- Other optimizations: unroll-and-jam, loop tiling
- Benchmarks: MM, HD, LU, SOR
- Itanium workstation 733MHz, 16KB/96KB/2MB/2GB

Experiments: Relative Speedup



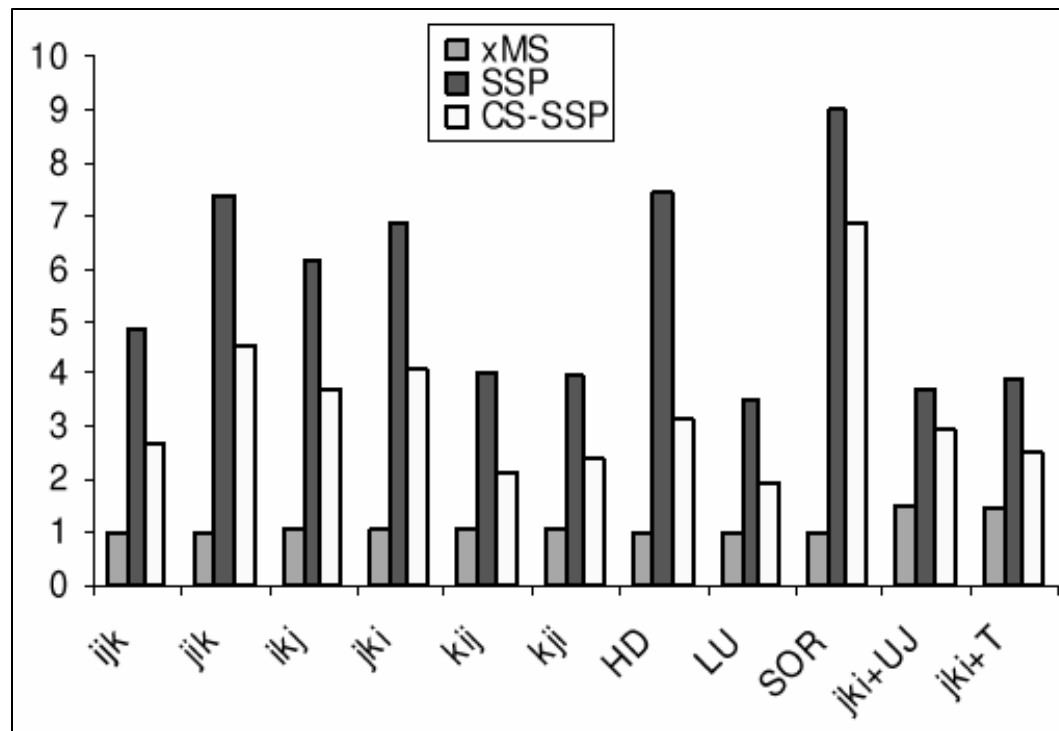
- Speedup between 1.1 and 4.24, average 2.1.
- Better performance : better parallelism and/or better data reuse.
- Code-size optimized version performs as well as original version.
- Code duplication and code size do not degrade performance.

Experiments: Bundle Density



- Bundle density measures average number of non-NOP in a bundle.
- Average: MS-xMS: 1.90, SSP: 1.91, CS-SSP: 2.1
- CS-SSP produces a denser code.
- CS-SSP makes better use of available resources.

Experiments: Relative Code Size



- SSP code is between 3.6 and 9.0 times bigger than MS/xMS .
- CS-SSP code is between 2 and 6.85 times bigger than MS/xMS.
- Because of multiple patterns and code duplication in innermost loop.
- However entire code (~4KB) easily fits in the L1 instruction cache.

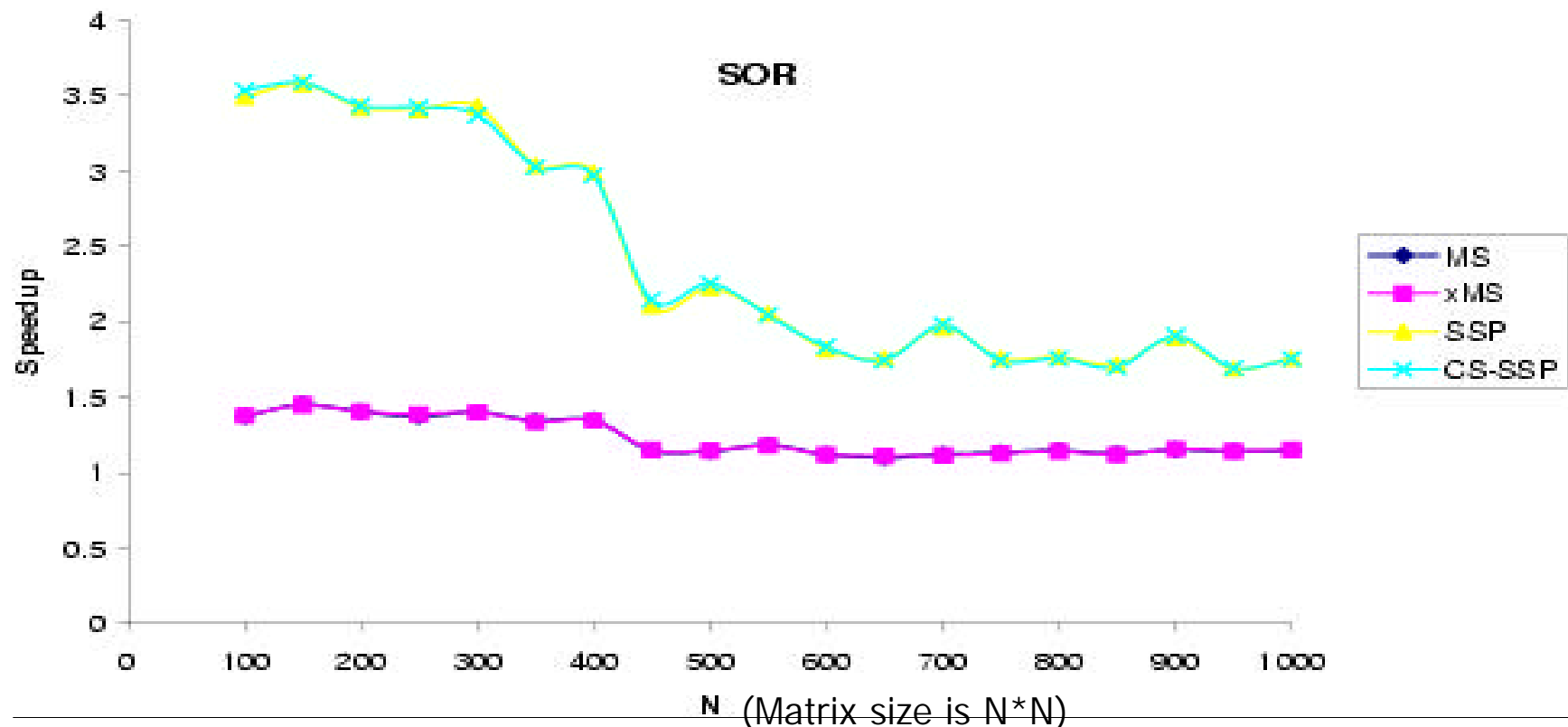
Acknowledgement

- Prof. Bogong Su, Dr. Hongbo Yang
- Anonymous reviewers
- Chan, Sun C.
- NSF, DOE agencies

Appendix

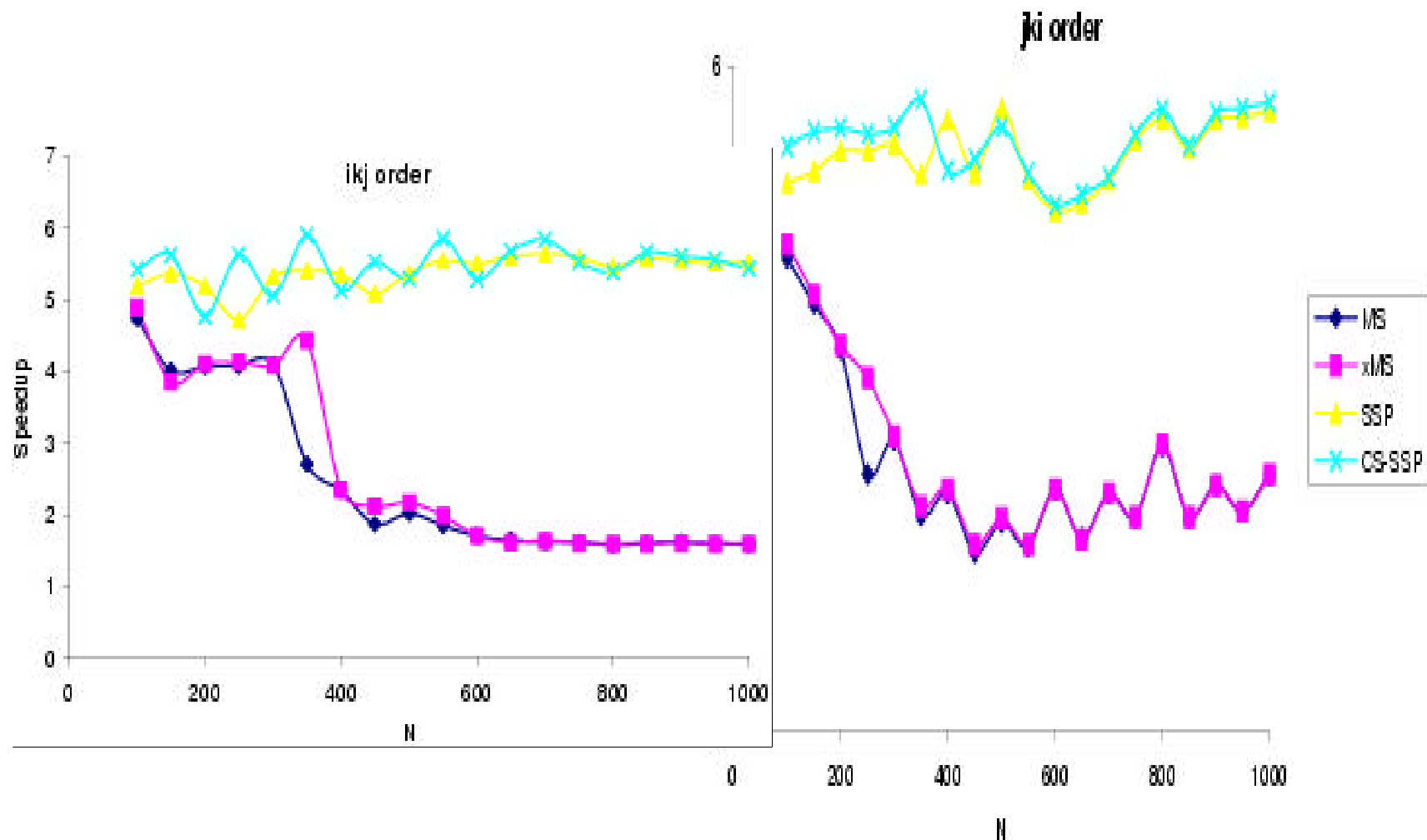
- The following slides are for the detailed performance analysis of SSP.

Exploiting Parallelism from the Whole Iteration Space

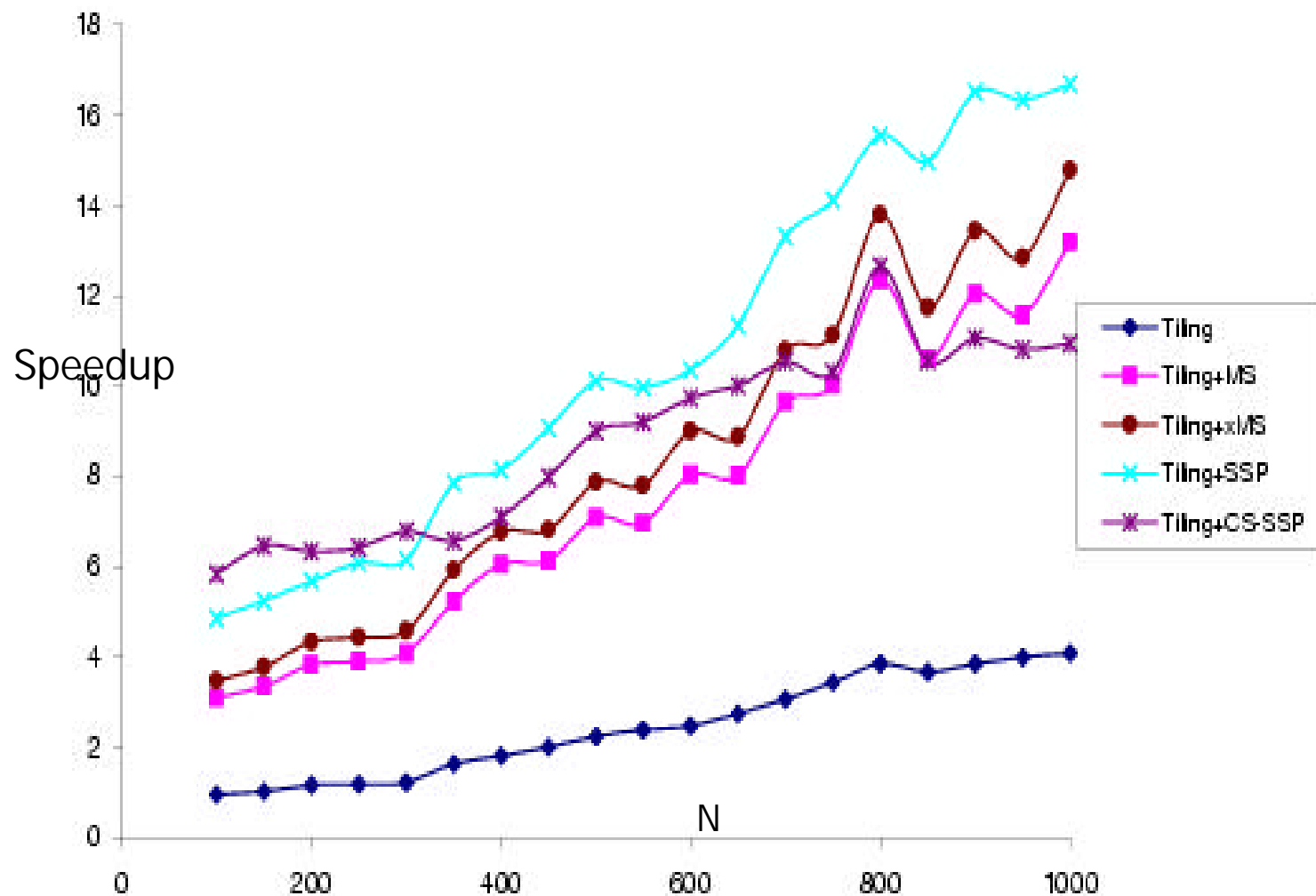


- Represents a class of important application
- Strong dependence cycle in the innermost loop
- The middle loop has negative dependence but can be removed.

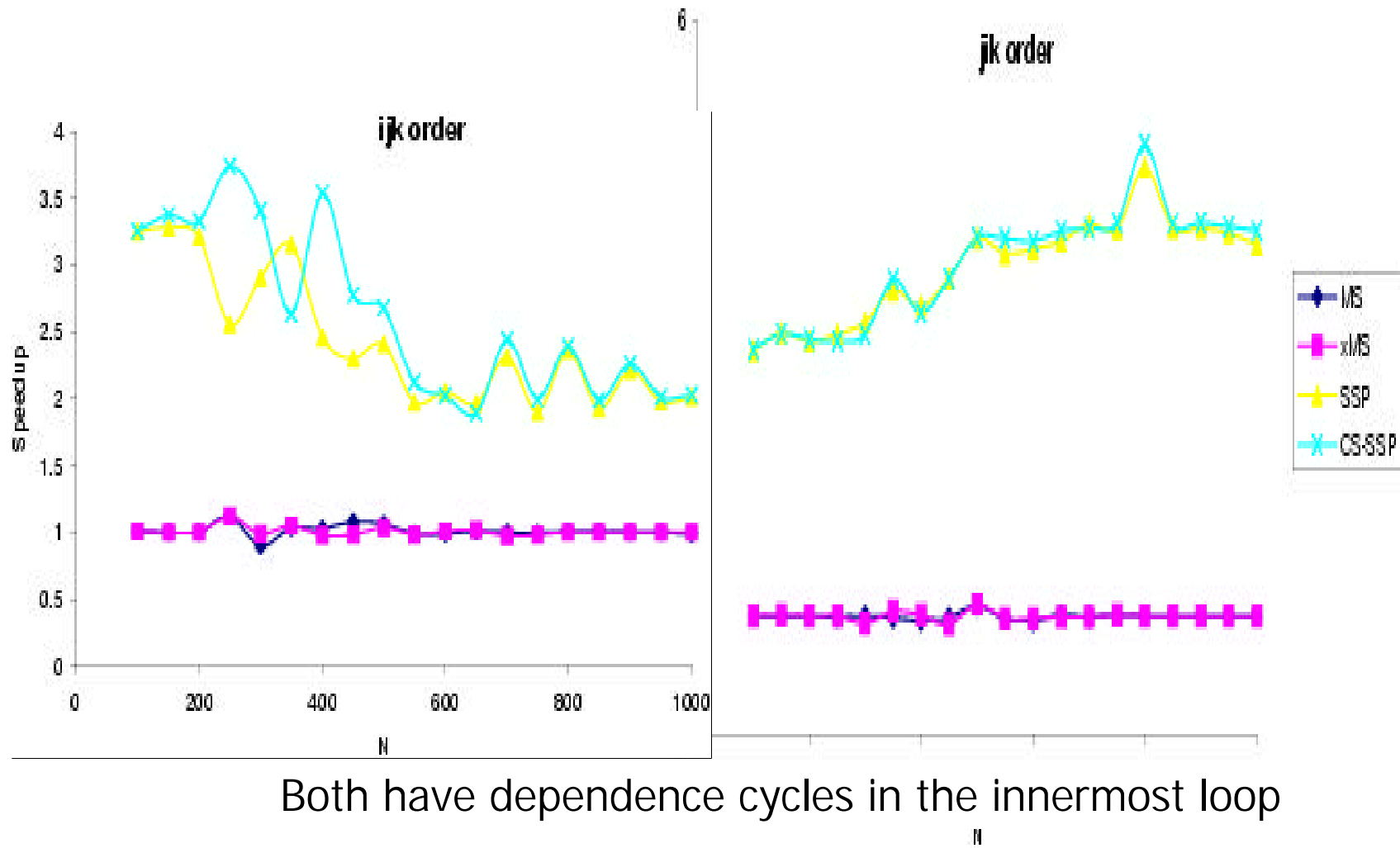
Exploiting Data Reuse from the Whole Iteration Space



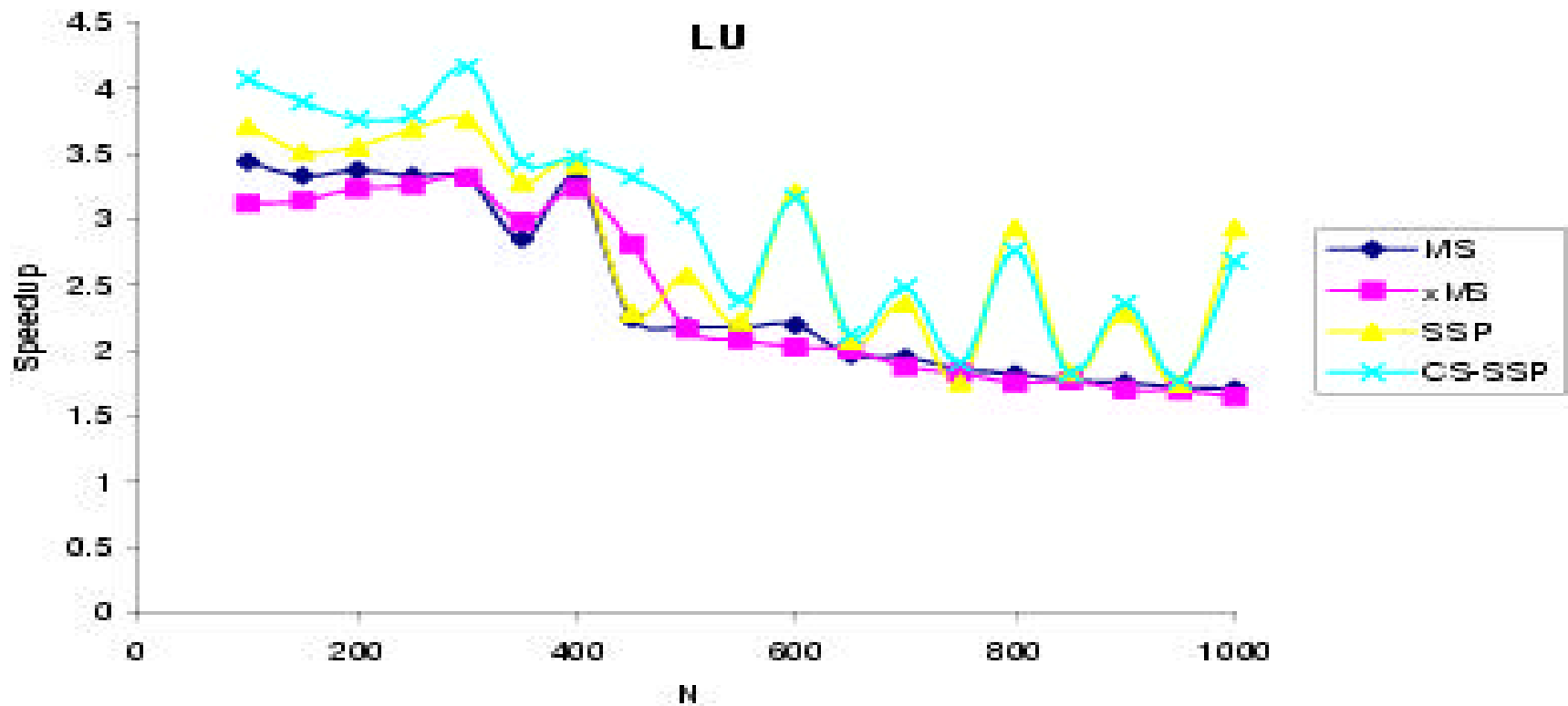
Advantage of Code Generation



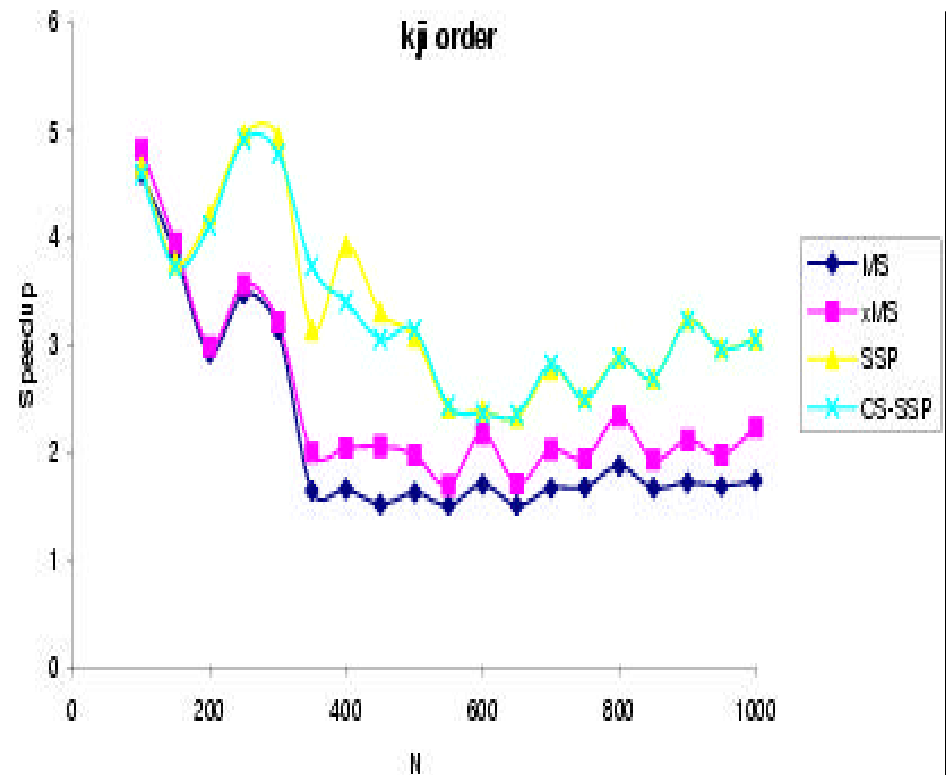
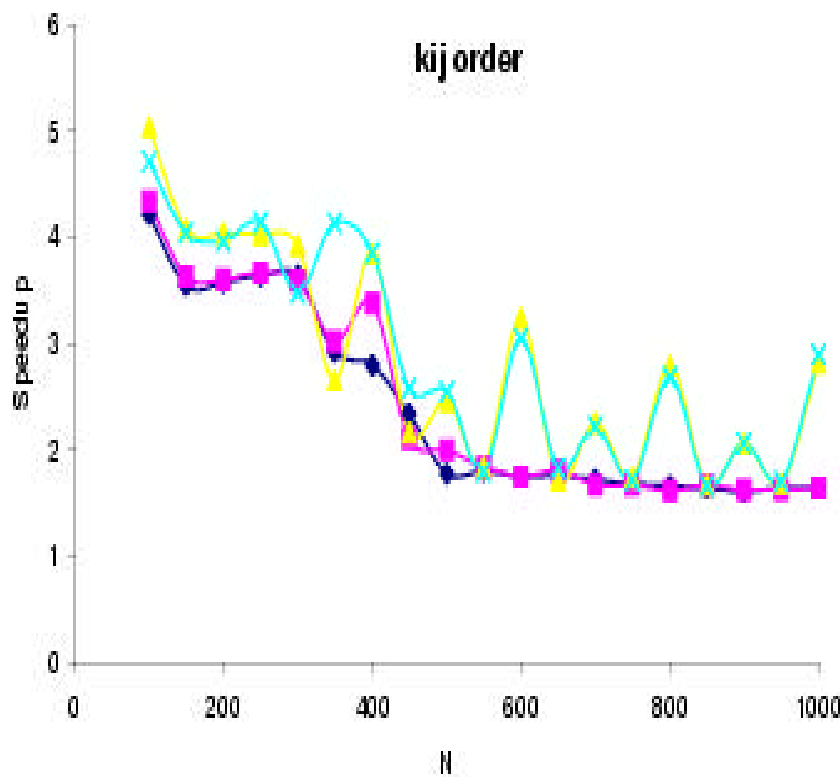
Exploiting Parallelism from the Whole Iteration Space (Cont.)



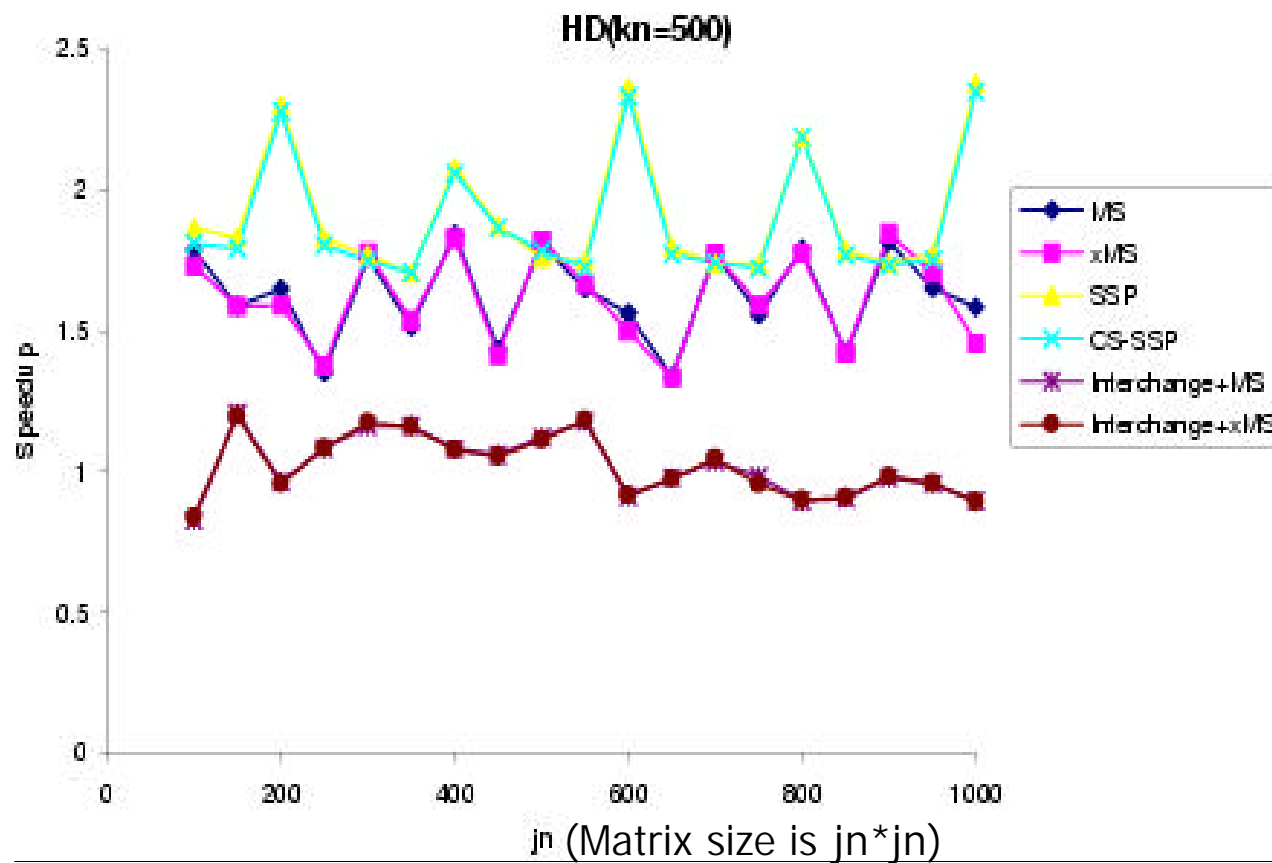
Exploiting Data Reuse from the Whole Iteration Space



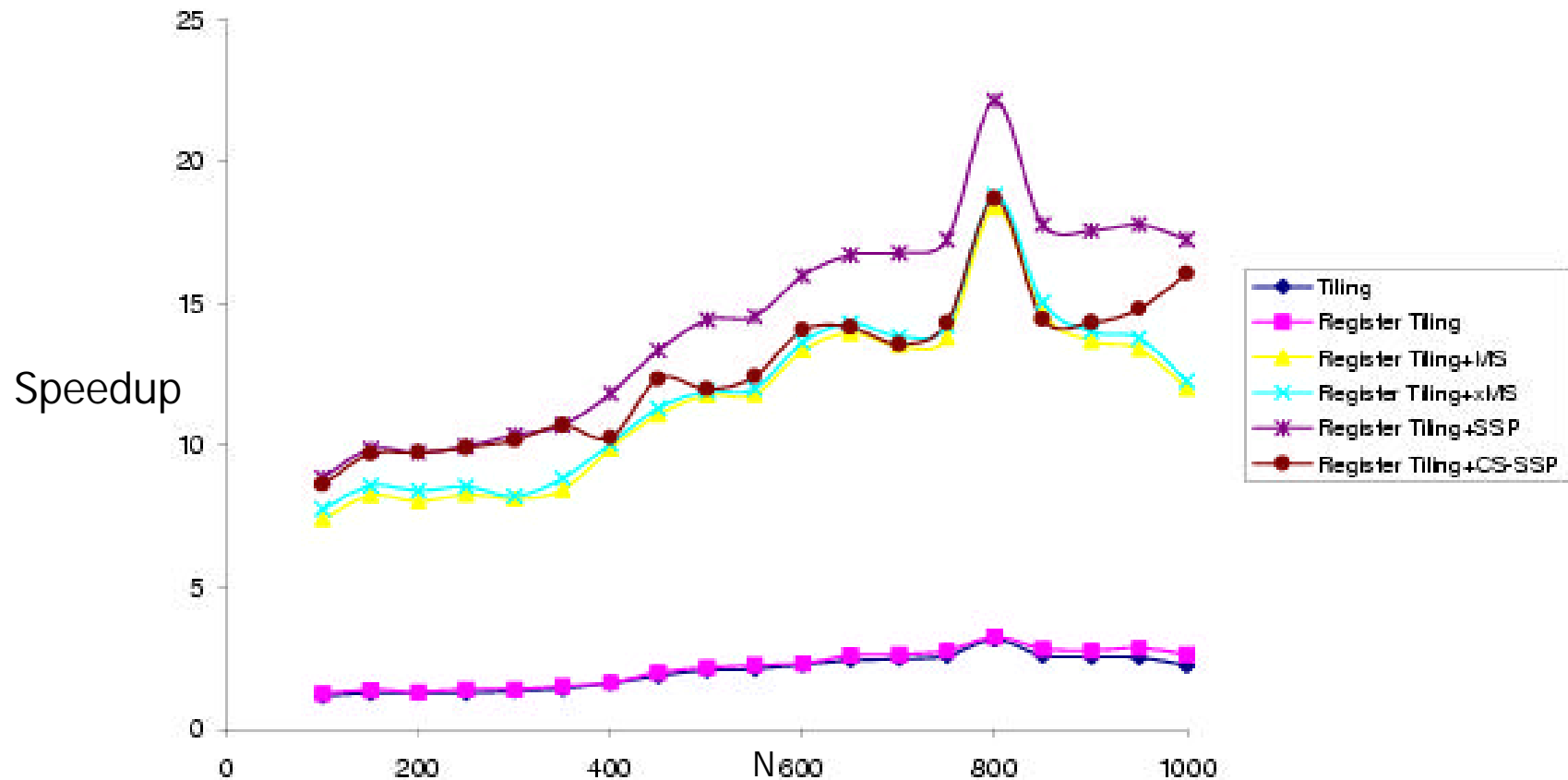
Exploiting Data Reuse from the Whole Iteration Space (Cont.)



Exploiting Data Reuse from the Whole Iteration Space (Cont.)

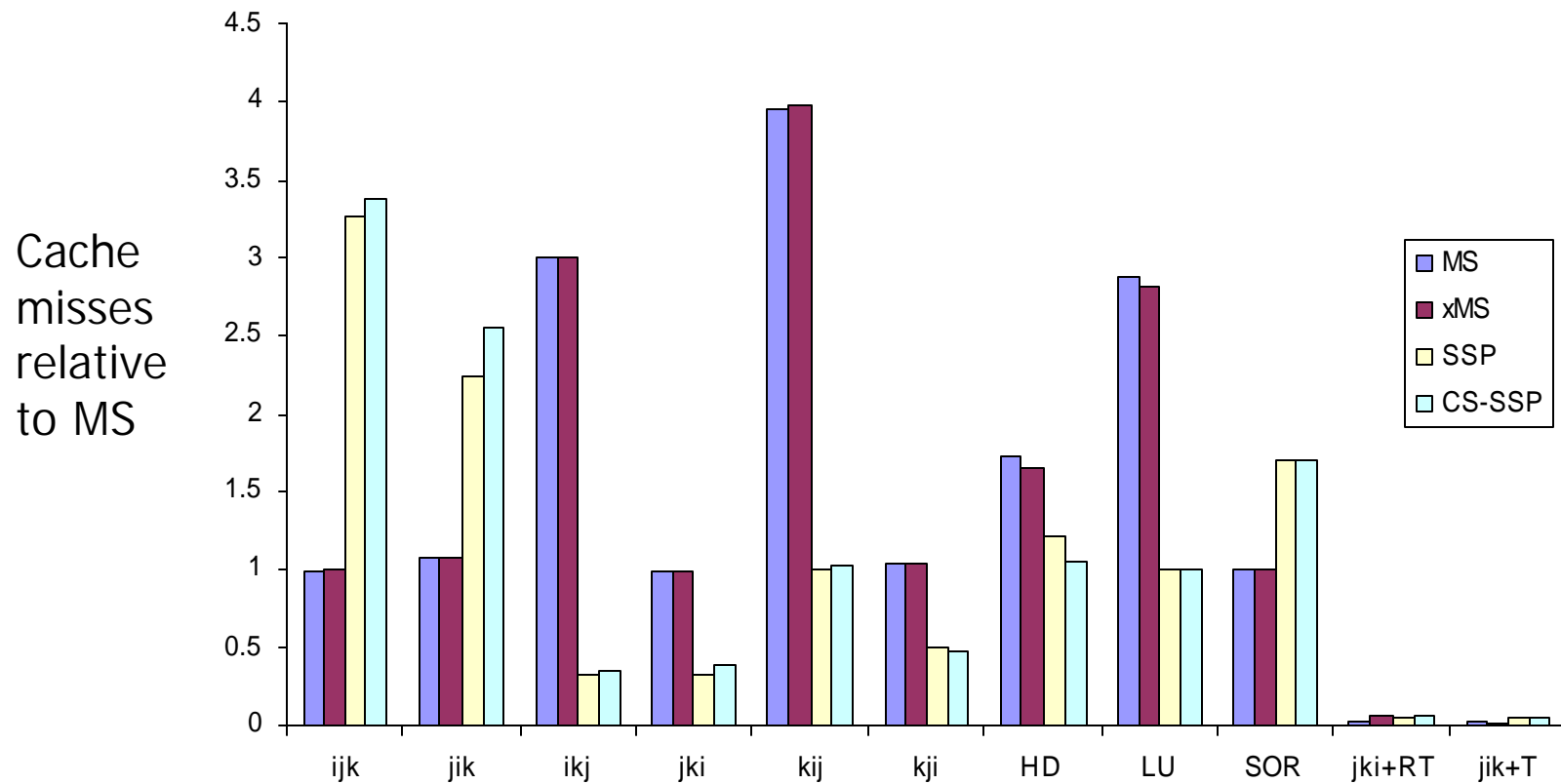


Advantage of Code Generation

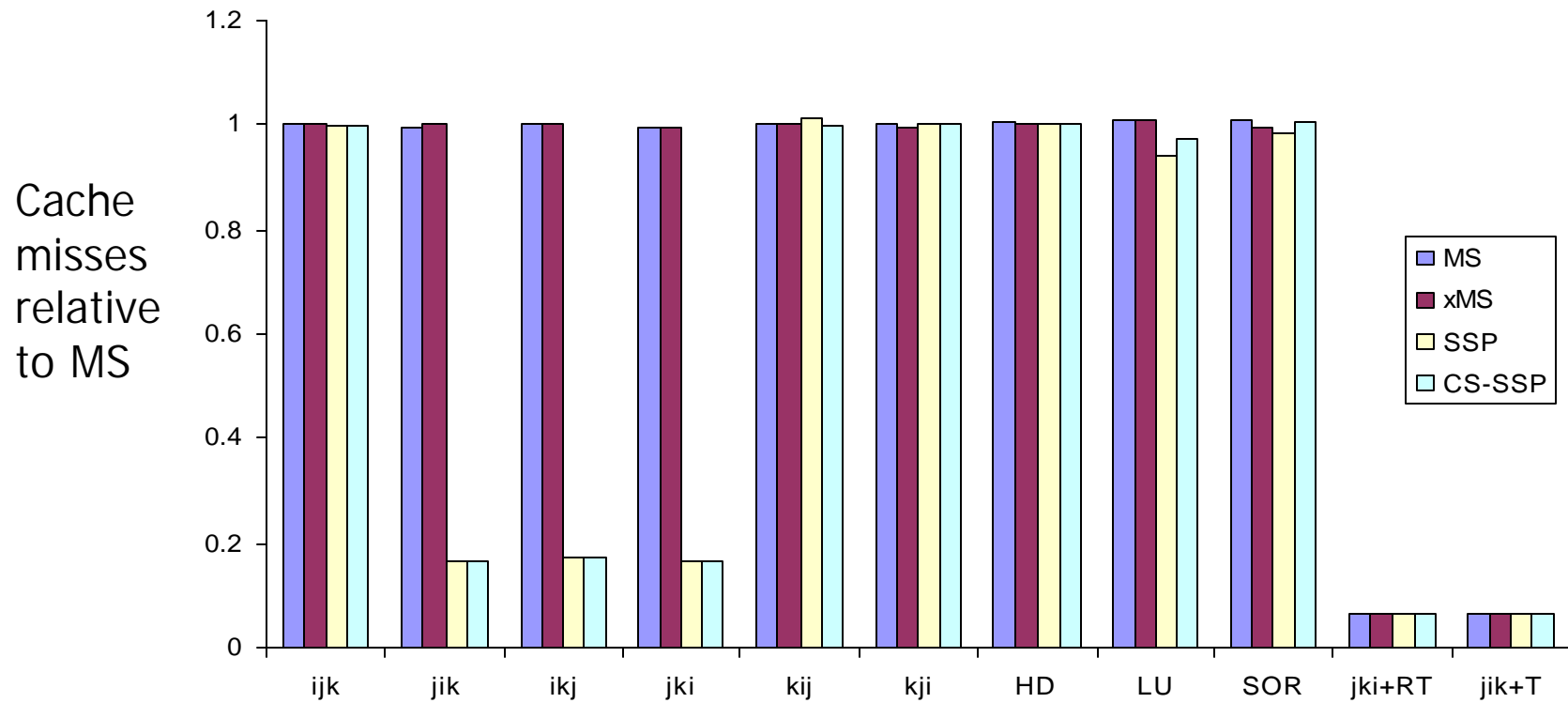


- SSP considers all operations in constructing 1-D schedule, thus effectively offsets the overhead of operations out of the innermost loop

Performance Analysis from L2 Cache misses



Performance Analysis from L3 Cache misses



Comparison with Linear Schedule

- Linear schedule
 - Traditionally apply to multi-processing, systolic arrays, etc. , not for uniprocessor
 - Parallelism oriented. Do not consider
 - Fine-grain resource constraints
 - Register usage
 - Data reuse
 - Code generation
 - Communicate values through memory, or message passing, etc.

Optimized Iteration Space of A Linear Schedule

