Just-In-Time Software Pipelining

Hongbo Rong
Hyunchul Park
Youfeng Wu
Cheng Wang

Programming Systems Lab
Intel Labs, Santa Clara

What is software pipelining?

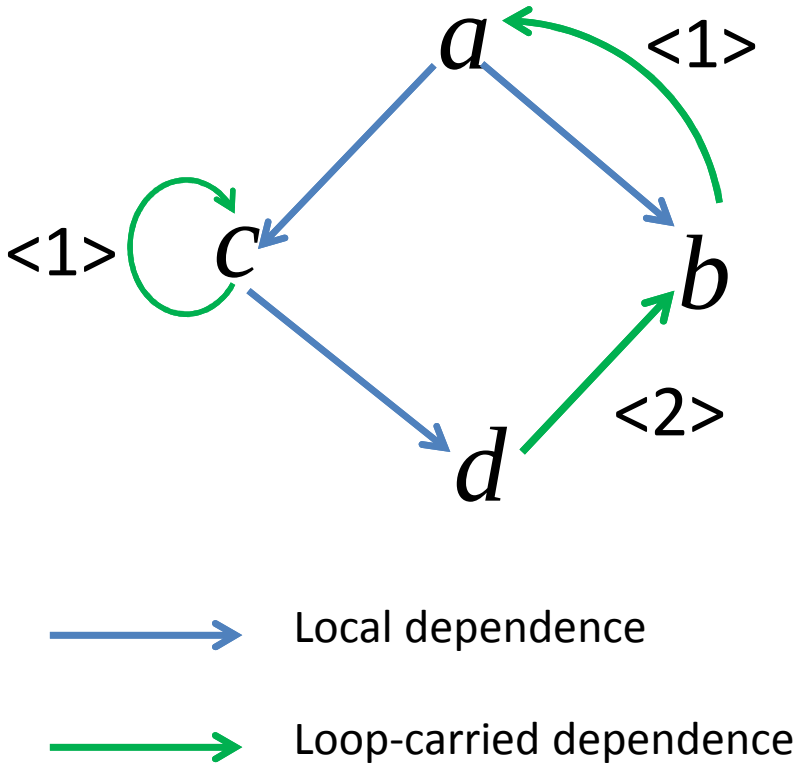
A loop optimization exposing instruction-level parallelism (ILP)

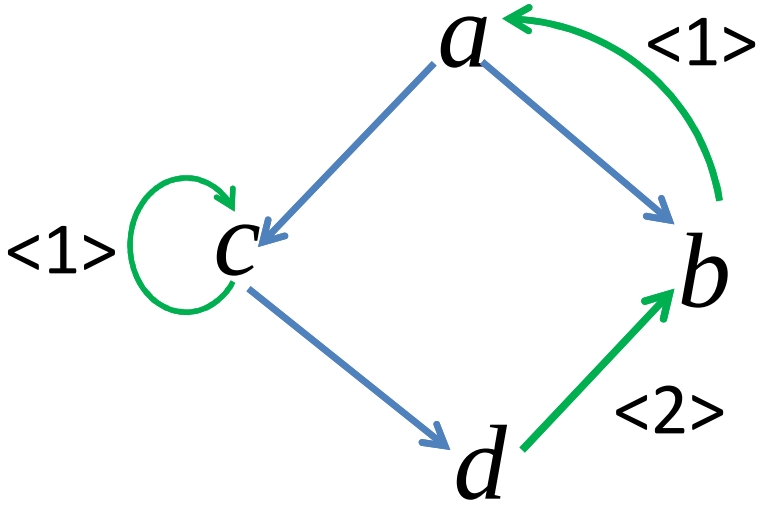
```
for (i = 0; i<N; i++){
  a: x = y + 1
  b: y = A[i] + x
  c: B[i+2] = B[i]*x
  d: A[i+2] = B[i+2]
}
```

What is software pipelining?

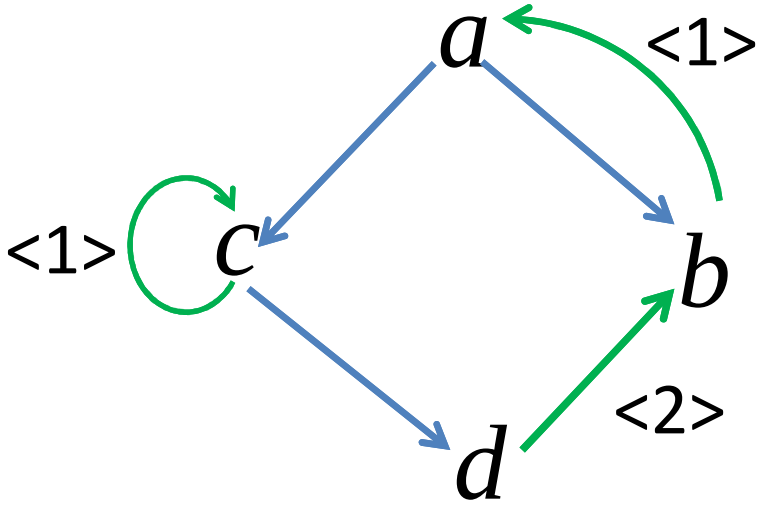
A loop optimization exposing instruction-level parallelism (ILP)

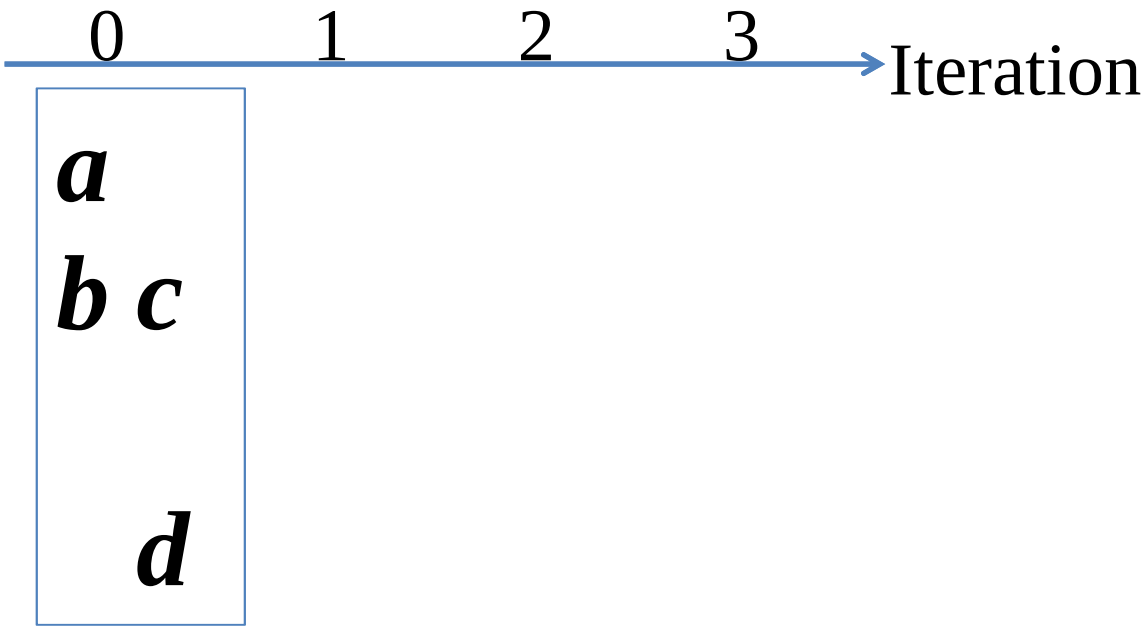
```
for (i = 0; i<N; i++){
  a: x = y + 1
  b: y = A[i] + x
  c: B[i+2] = B[i]*x
  d: A[i+2] = B[i+2]
}
```

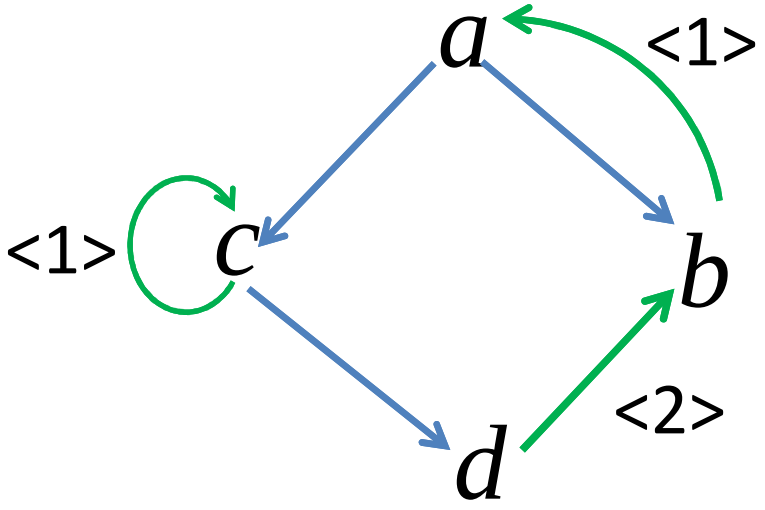


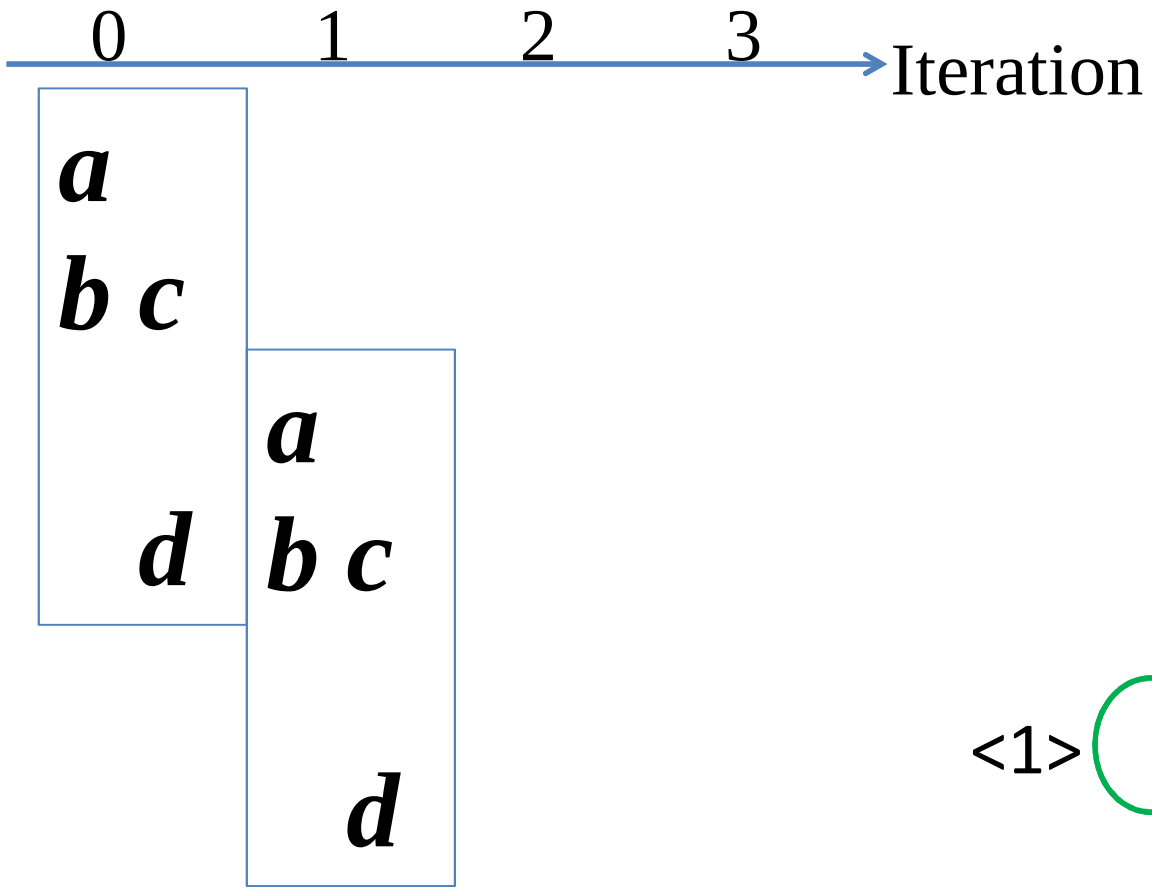

a
<1>
<1>
c
b
<2>
d

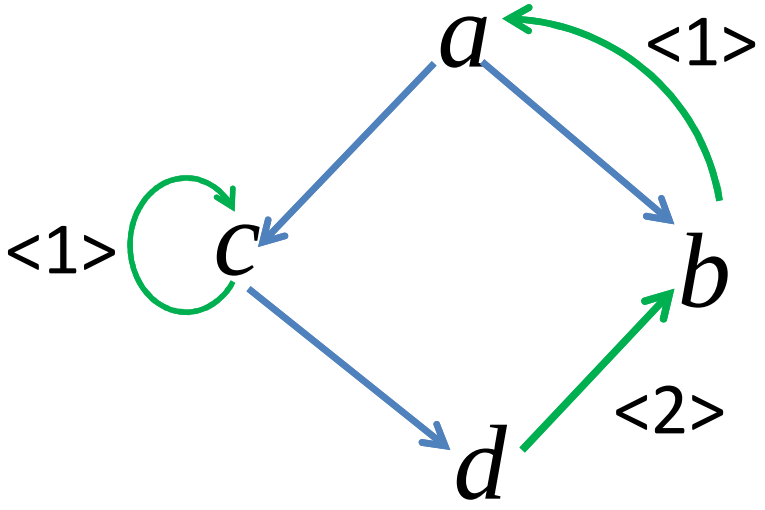

0
1
2
3
Iteration

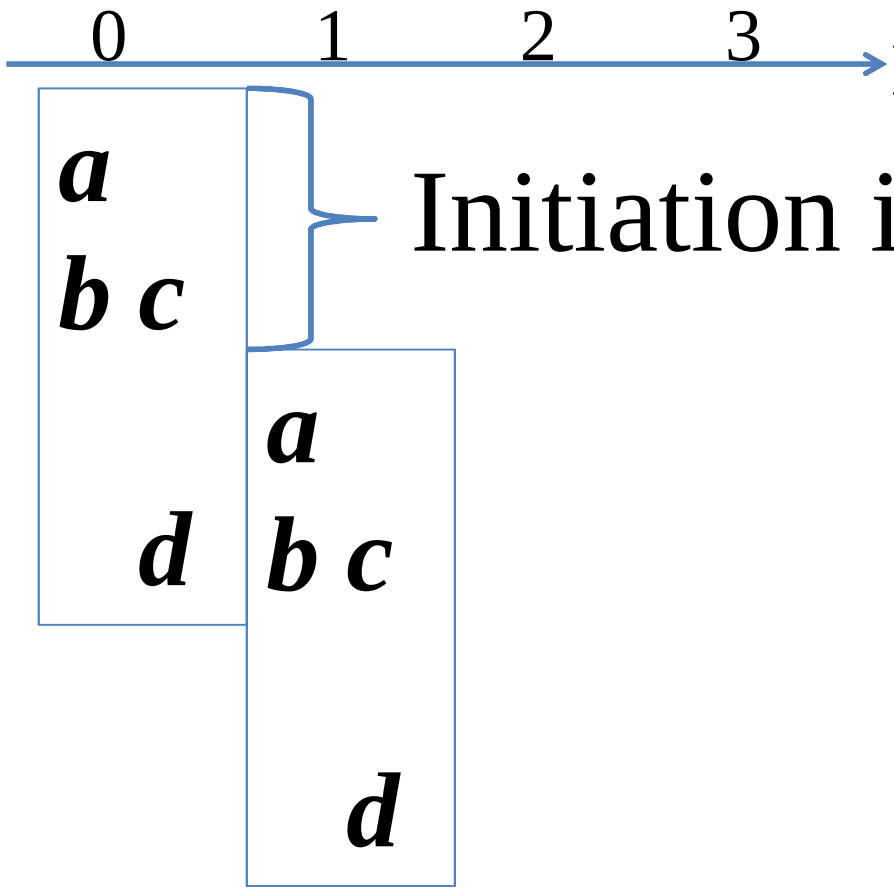

a
<1>
<1>
c
b
<2>
d

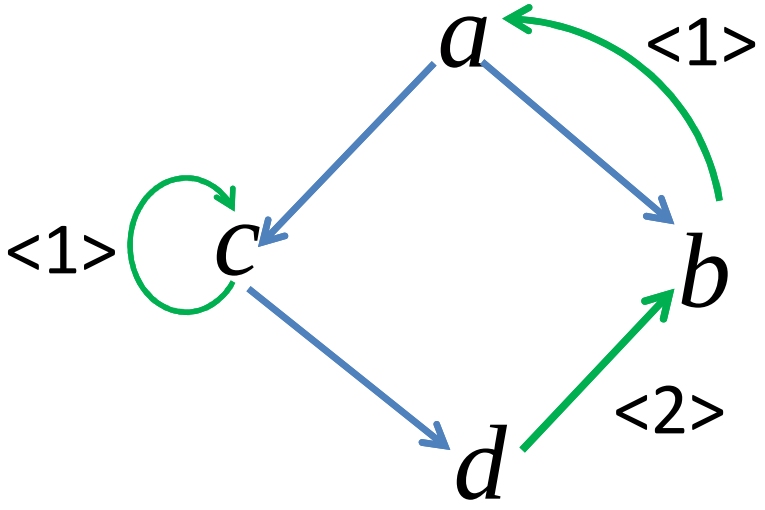

0
1
2
3
Iteration
a
b c
d

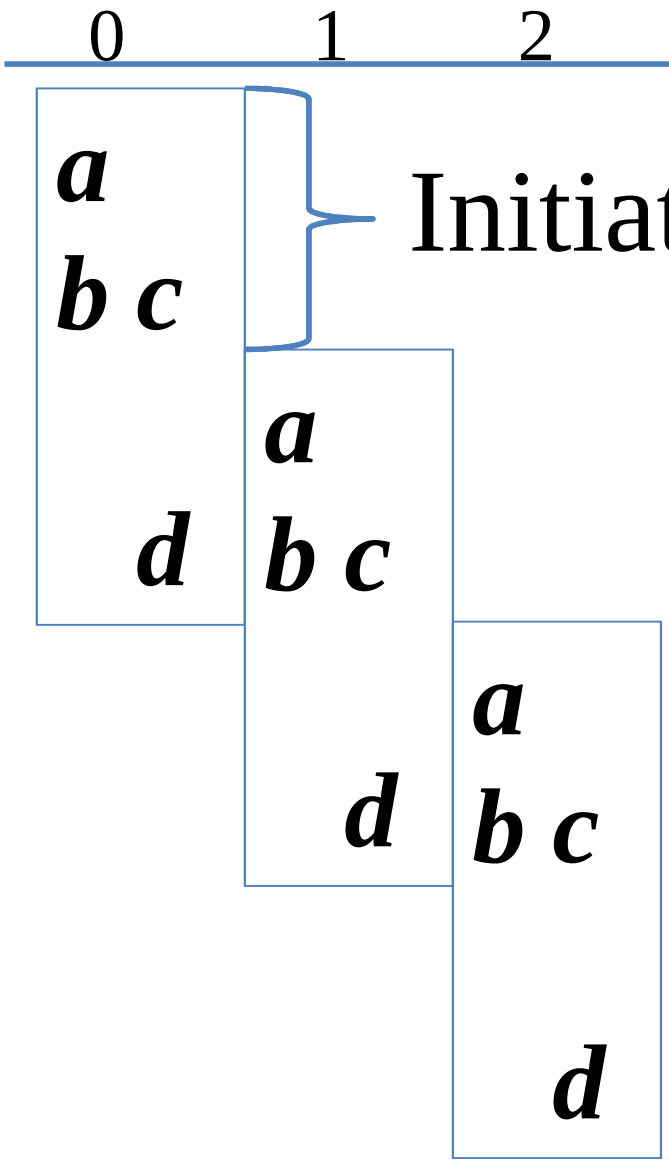

a
<1>
<1>
c
b
<2>
d

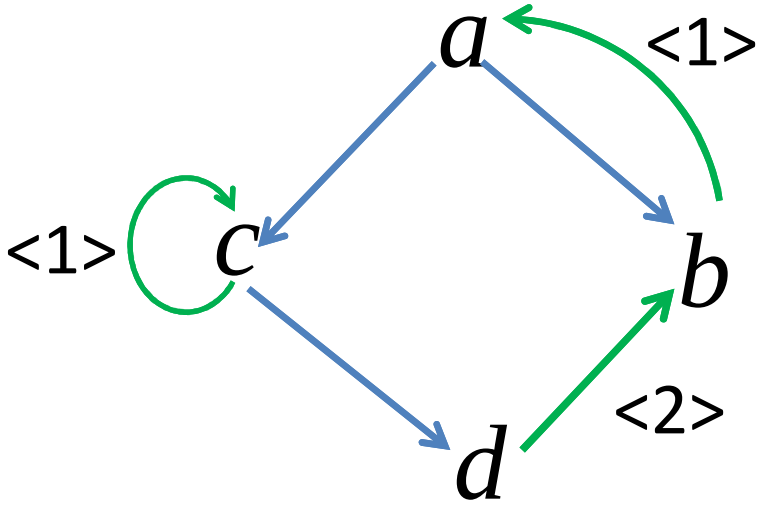

0
1
2
3
Iteration
a
b c
d
a
b c
d

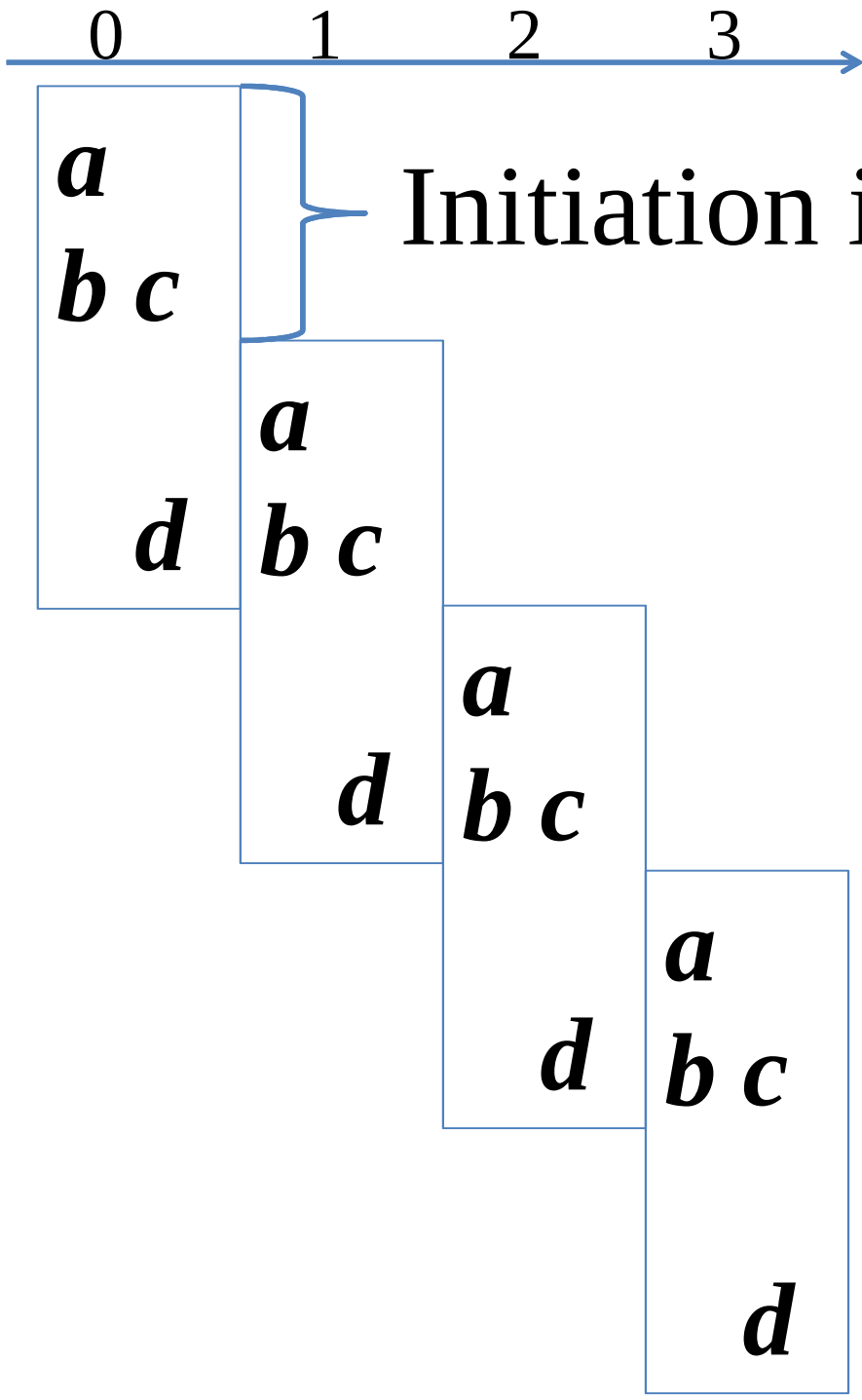

a
<1>
<1>
c
b
<2>
d

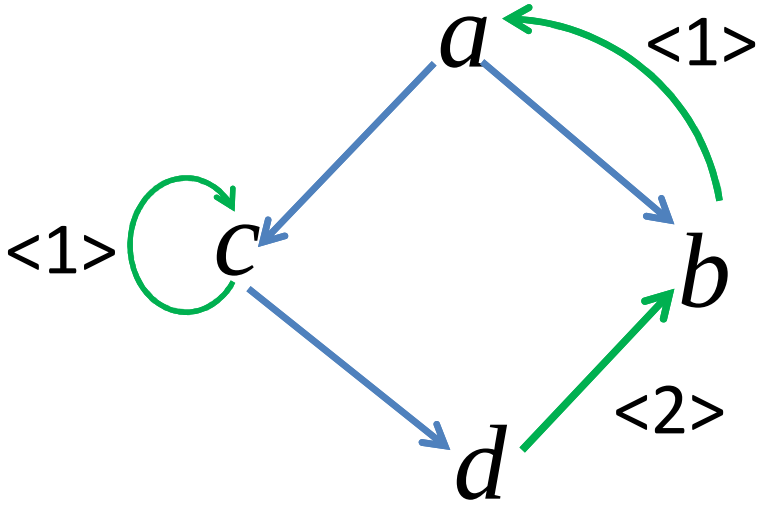

0
1
2
3
Iteration
a
b c
d
a
b c
d
Initiation interval(II)=2

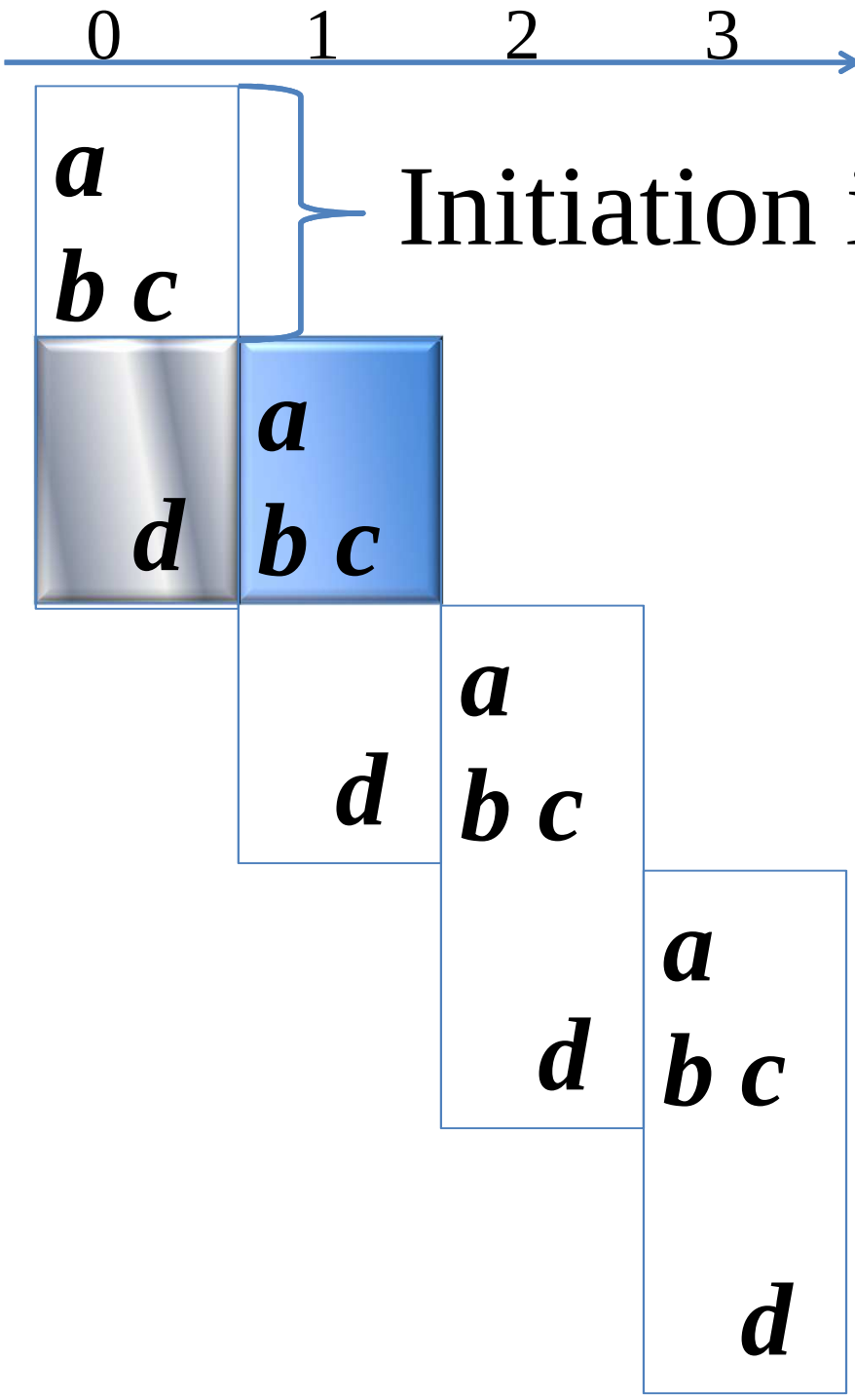

a
<1>
<1>
c
b
<2>
d

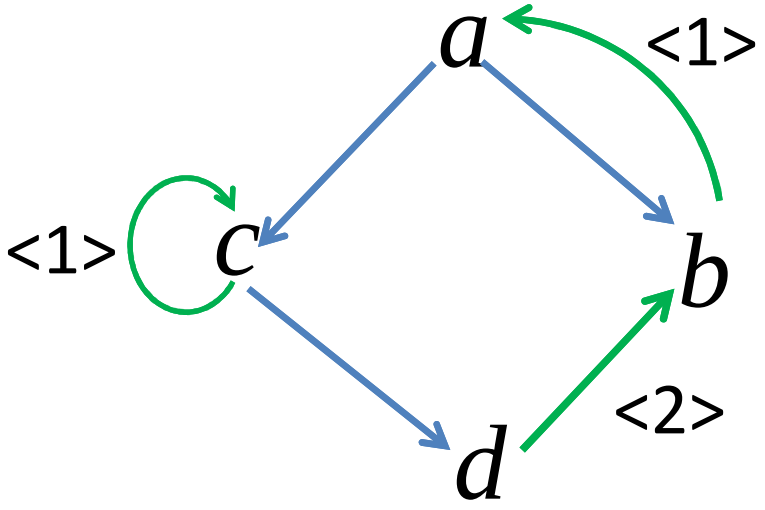

0
1
2
3
Iteration
a
b c
d
Initiation interval(II)=2

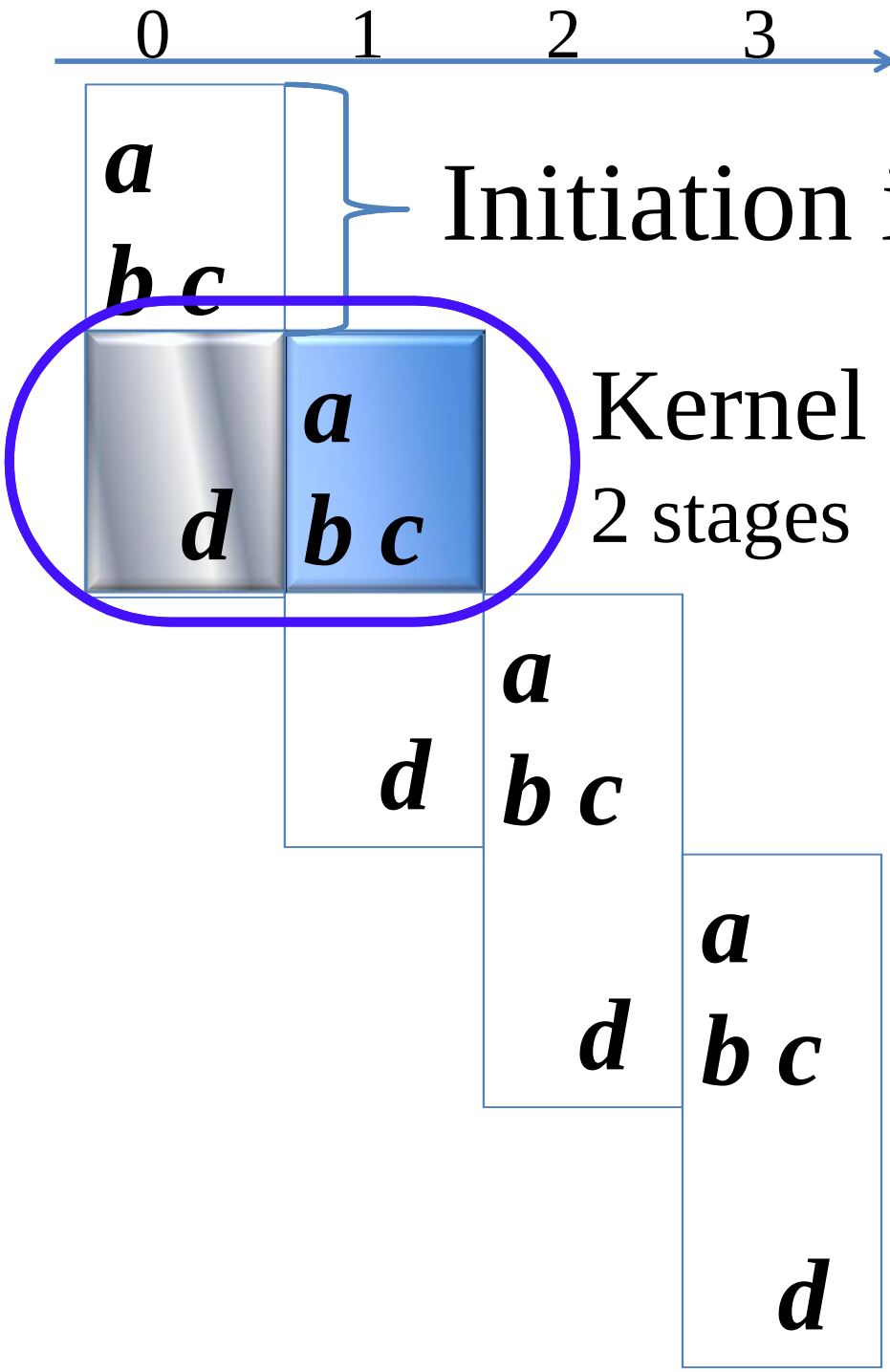

a
<1>
<1>
c
b
<2>
d


0
1
2
3
Iteration
a
b c
d
Initiation interval(II)=2

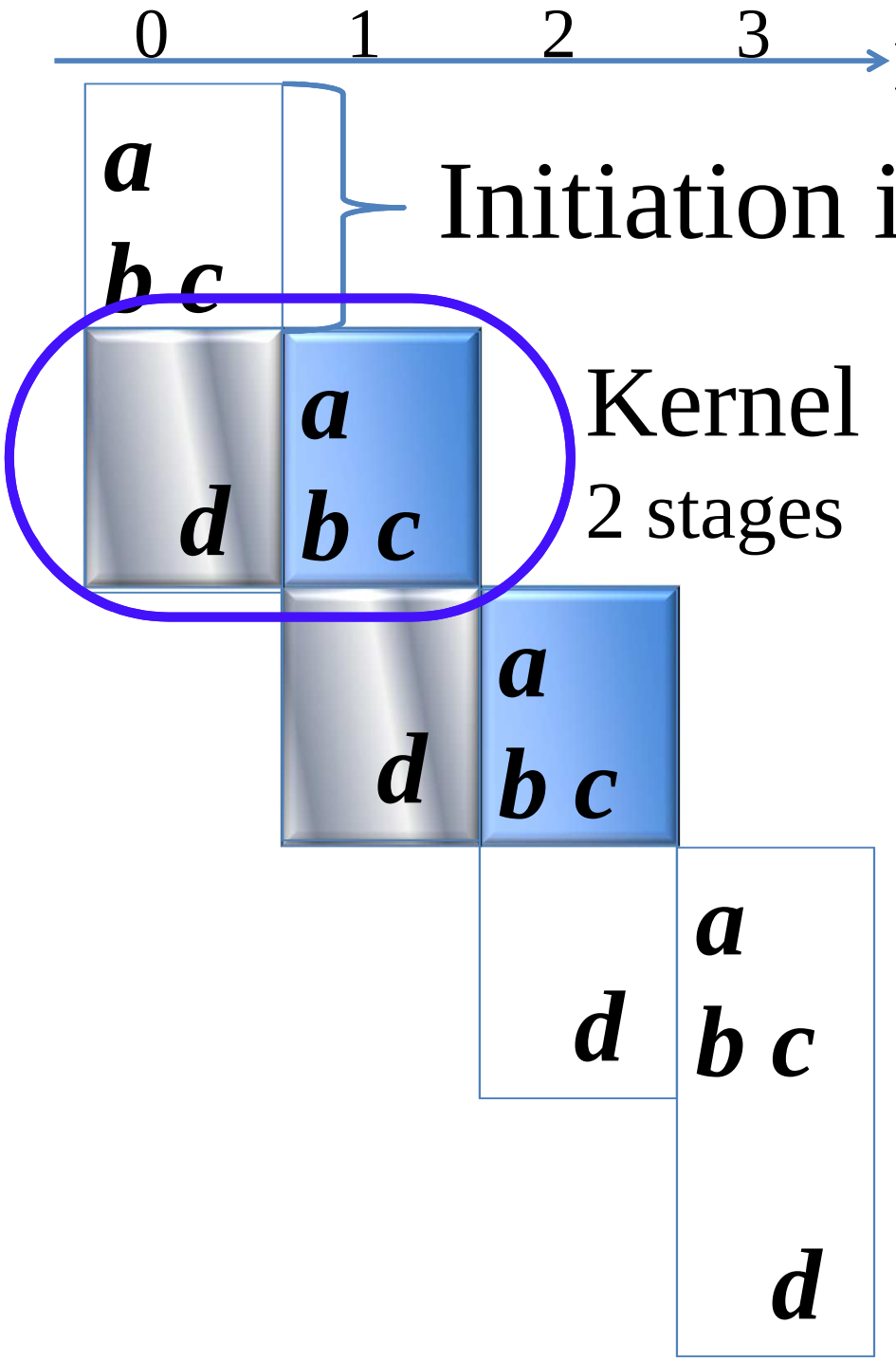

a
<1>
<1>
c
b
d
<2>


0
1
2
3
Iteration
a
b c
d
Initiation interval(II)=2

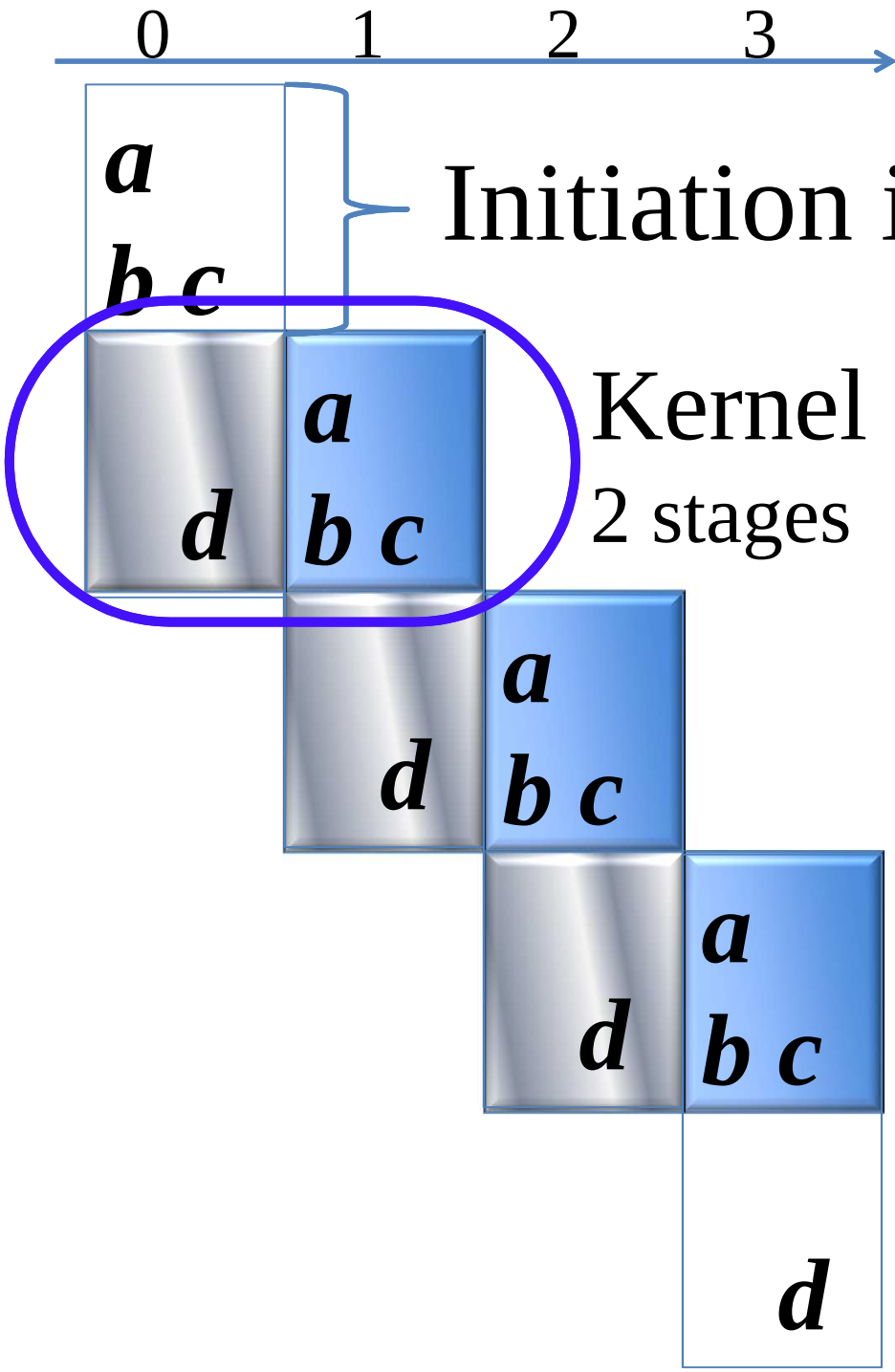

a
<1>
<1>
c
b
d
<2>



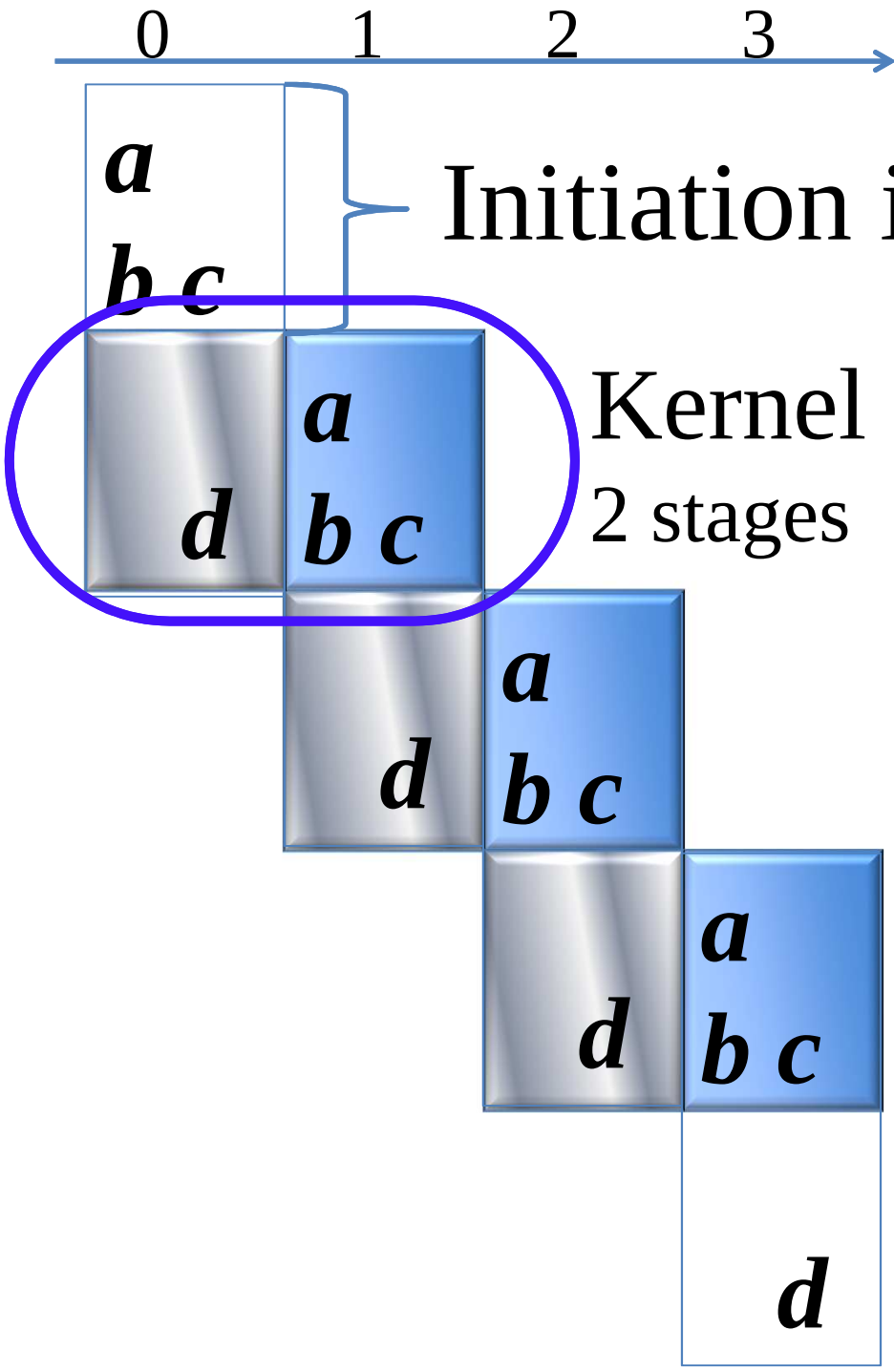
- Different iterations work on different stages in parallel



- Different iterations work on different stages in parallel



- Different iterations work on different stages in parallel



- Different iterations work on different stages in parallel
- II is the performance indicator

Software pipelining has been static

- Extensively studied in 3 decades, and efficient for wide-issue architectures
  - VLIW [Lam 1988]
  - superscalar [Ruttenberg et al. 1996]

Software pipelining has been static

- Extensively studied in 3 decades, and efficient for wide-issue architectures
  - VLIW [Lam 1988]
  - superscalar [Ruttenberg et al. 1996]
- It is seen only in static compilers

Software pipelining has been static

- Extensively studied in 3 decades, and efficient for wide-issue architectures
  - VLIW [Lam 1988]
  - superscalar [Ruttenberg et al. 1996]
- It is seen only in static compilers
- Most works aim to minimize II but not compile overhead

It is time now to extend software pipelining to dynamic compilers!

It is time now to extend software pipelining to dynamic compilers!

- Dynamic languages are increasingly popular
  - JavaScript and PhP 88.9% and 81.5% in client and server websites (W3Techs)

It is time now to extend software pipelining to dynamic compilers!

- Dynamic languages are increasingly popular
  - JavaScript and PhP 88.9% and 81.5% in client and server websites (W3Techs)
- Huge amount of legacy code
  - Small optimization scope: a loop iteration
  - Software pipelining enlarges the scope to many iterations

It is time now to extend software pipelining to dynamic compilers!

- Dynamic languages are increasingly popular
  - JavaScript and PhP 88.9% and 81.5% in client and server websites (W3Techs)
- Huge amount of legacy code
  - Small optimization scope: a loop iteration
  - Software pipelining enlarges the scope to many iterations
- Minimizing compile overhead must be the 1st objective
  - Only simple/fast algorithms can be used

It is time now to extend software pipelining to dynamic compilers!

- Dynamic languages are increasingly popular
  - JavaScript and PhP 88.9% and 81.5% in client and server websites (W3Techs)
- Huge amount of legacy code
  - Small optimization scope: a loop iteration
  - Software pipelining enlarges the scope to many iterations
- Minimizing compile overhead must be the 1st objective
  - Only simple/fast algorithms can be used
  - linear-time algorithms are preferred

Challenges

- Memory aliases kill parallelism
  - Hardware: Atomic region + rotating alias registers [MICRO-46]

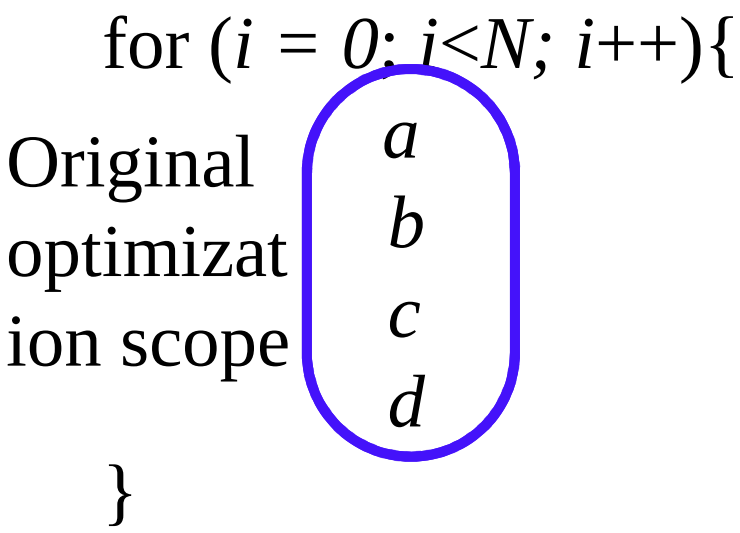
Challenges

- Memory aliases kill parallelism
  - Hardware: Atomic region + rotating alias registers [MICRO-46]

```
for (i = 0; i<N; i++){
        a
        b
        c
        d
}
```

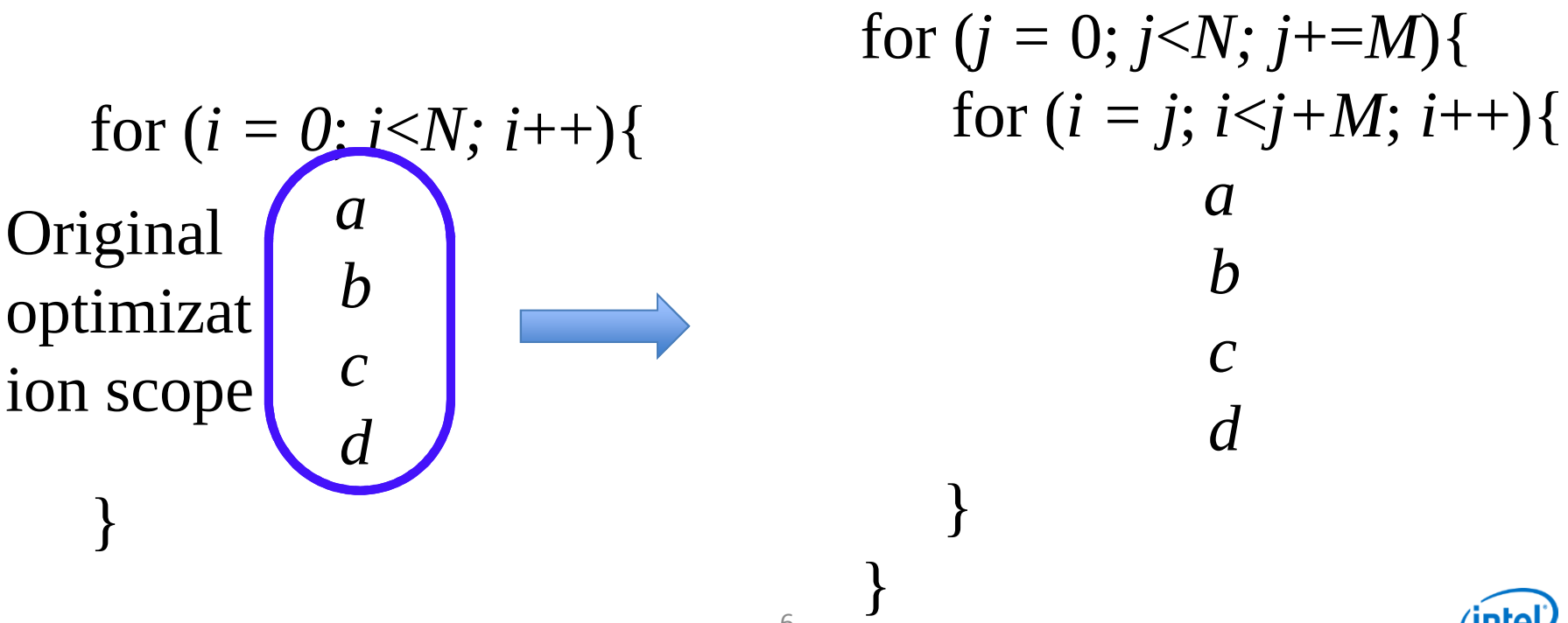
Challenges

- Memory aliases kill parallelism
  - Hardware: Atomic region + rotating alias registers [MICRO-46]



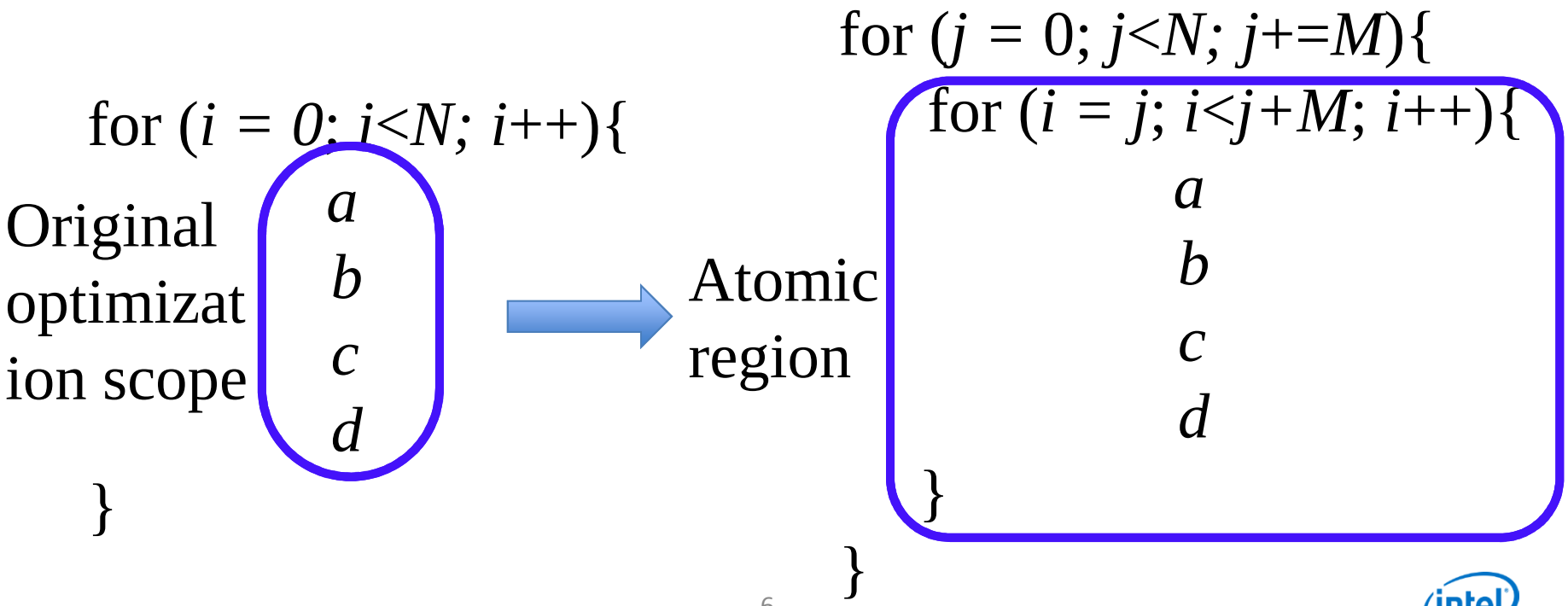
Challenges

- Memory aliases kill parallelism
  - Hardware: Atomic region + rotating alias registers [MICRO-46]



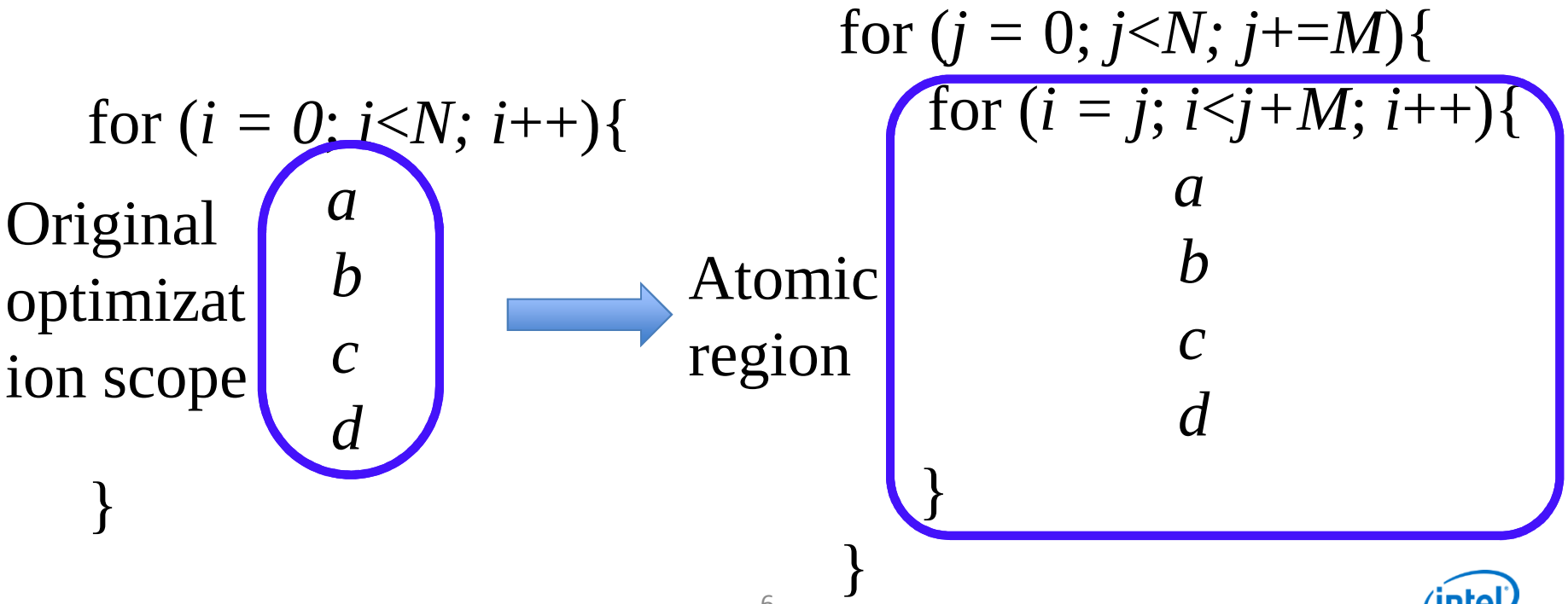
Challenges

- Memory aliases kill parallelism
  - Hardware: Atomic region + rotating alias registers [MICRO-46]



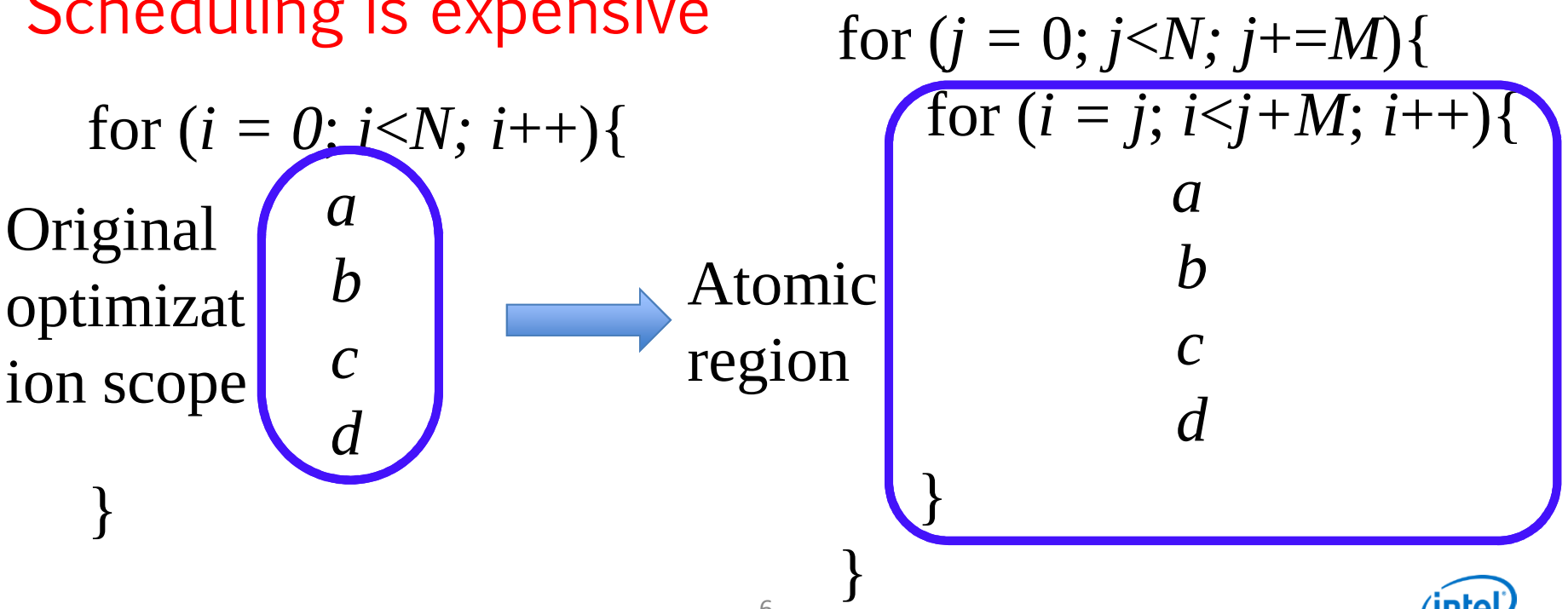
Challenges

- Memory aliases kill parallelism
  - Hardware: Atomic region + rotating alias registers [MICRO-46]
- Costly rollback
  - Software: Light-weight checkpointing



Challenges

- Memory aliases kill parallelism
  - Hardware: Atomic region + rotating alias registers [MICRO-46]
- Costly rollback
  - Software: Light-weight checkpointing
- Scheduling is expensive

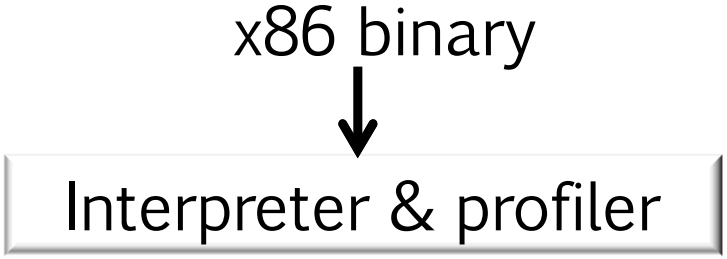


Framework
(on Transmeta CMS)

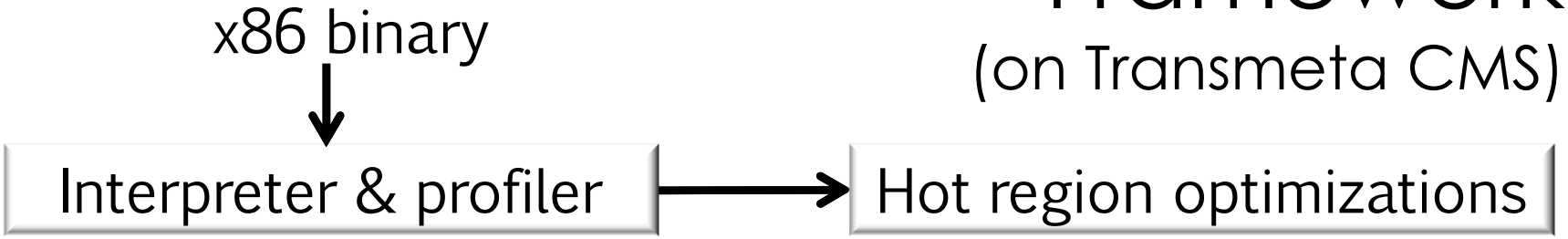
x86 binary

Framework
(on Transmeta CMS)

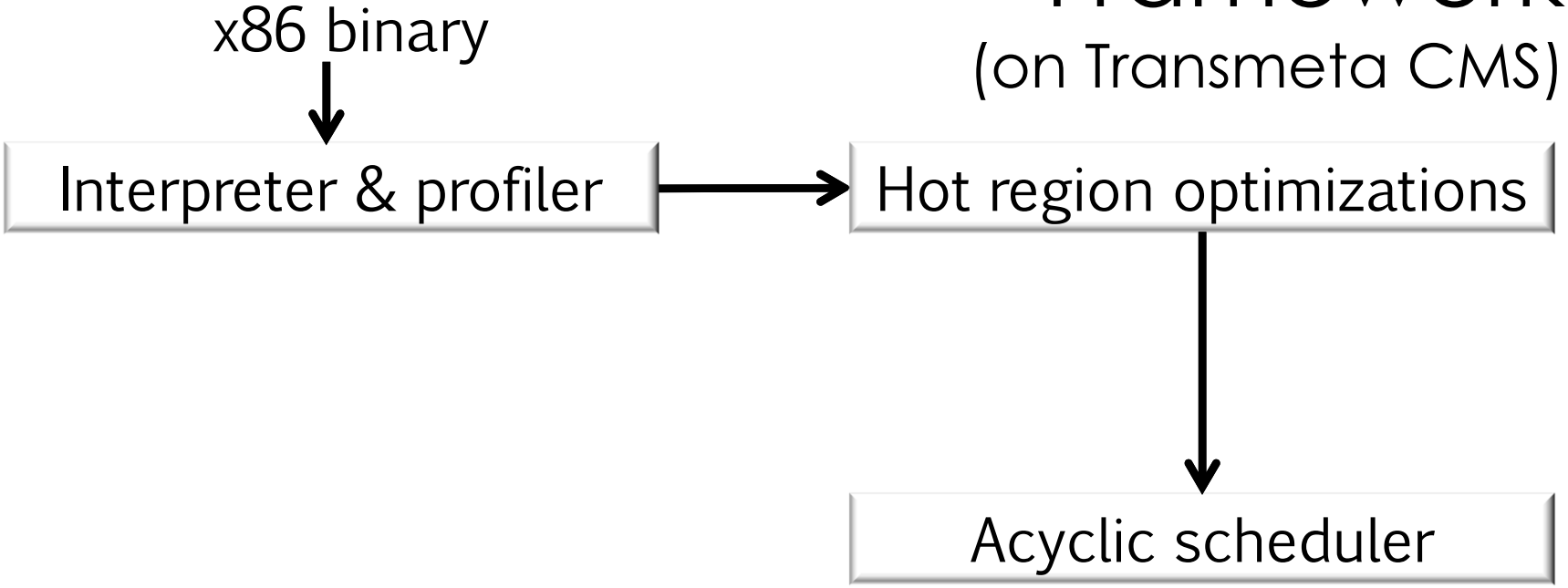
Framework
(on Transmeta CMS)



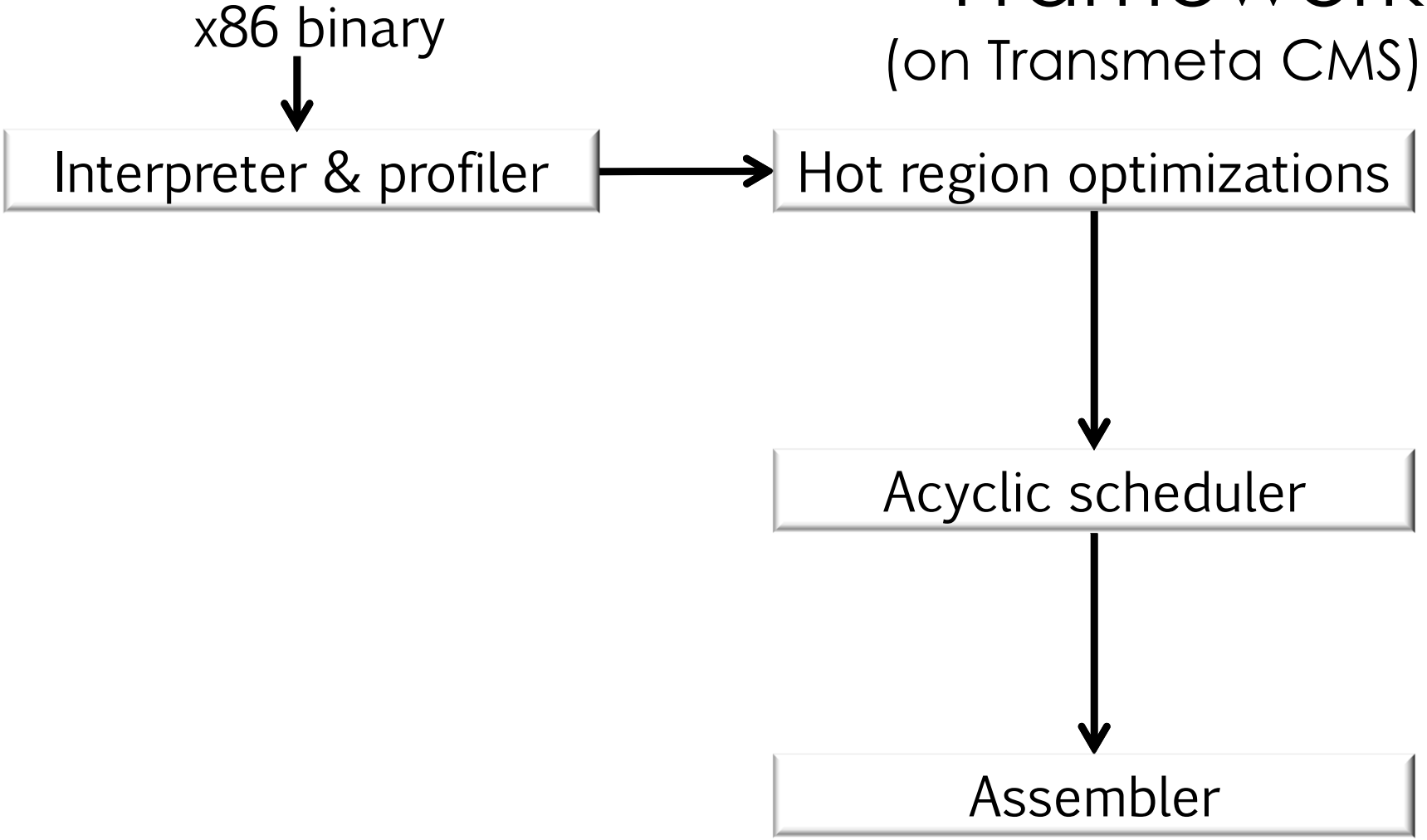
Framework
(on Transmeta CMS)



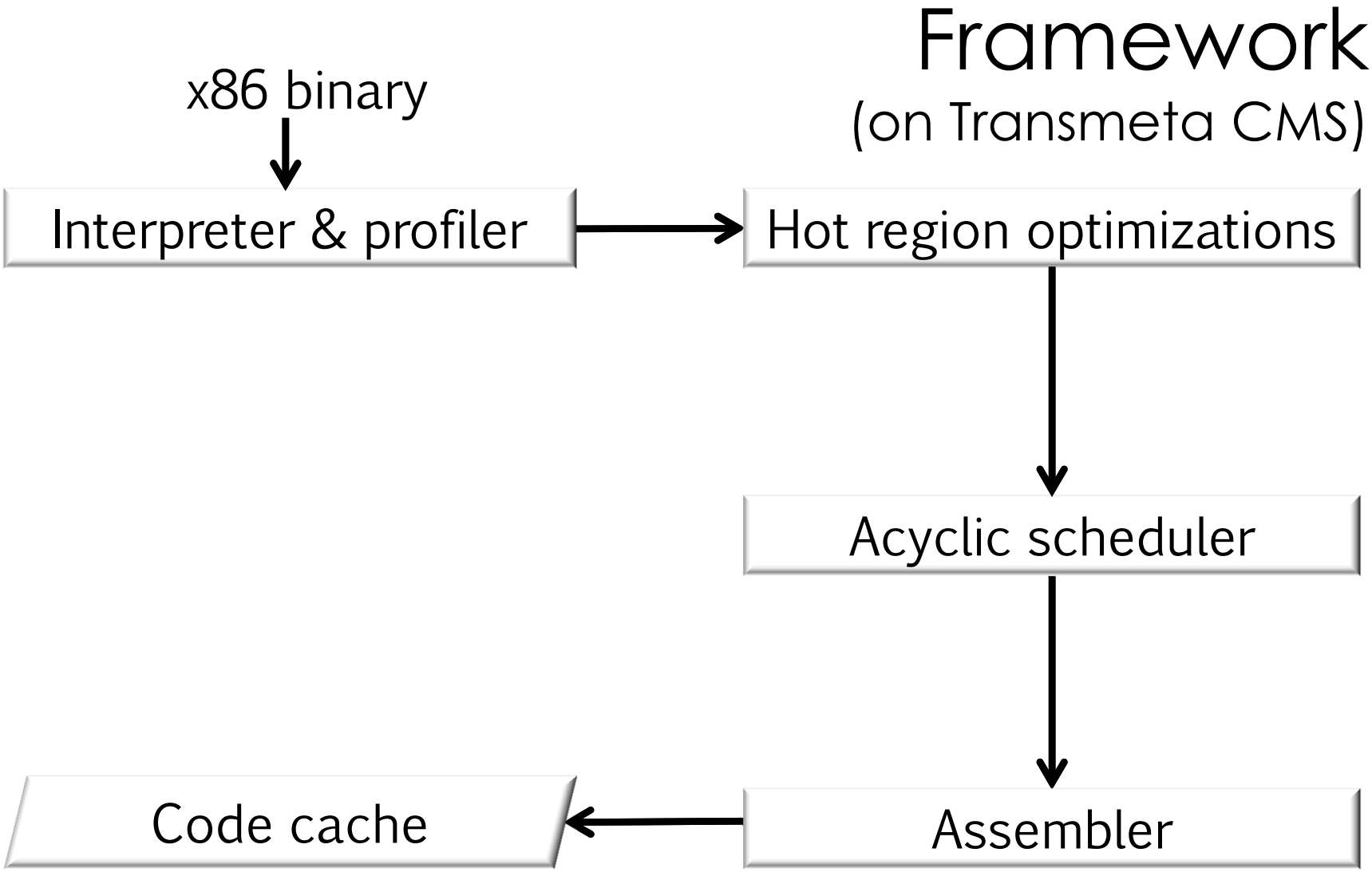
Framework
(on Transmeta CMS)



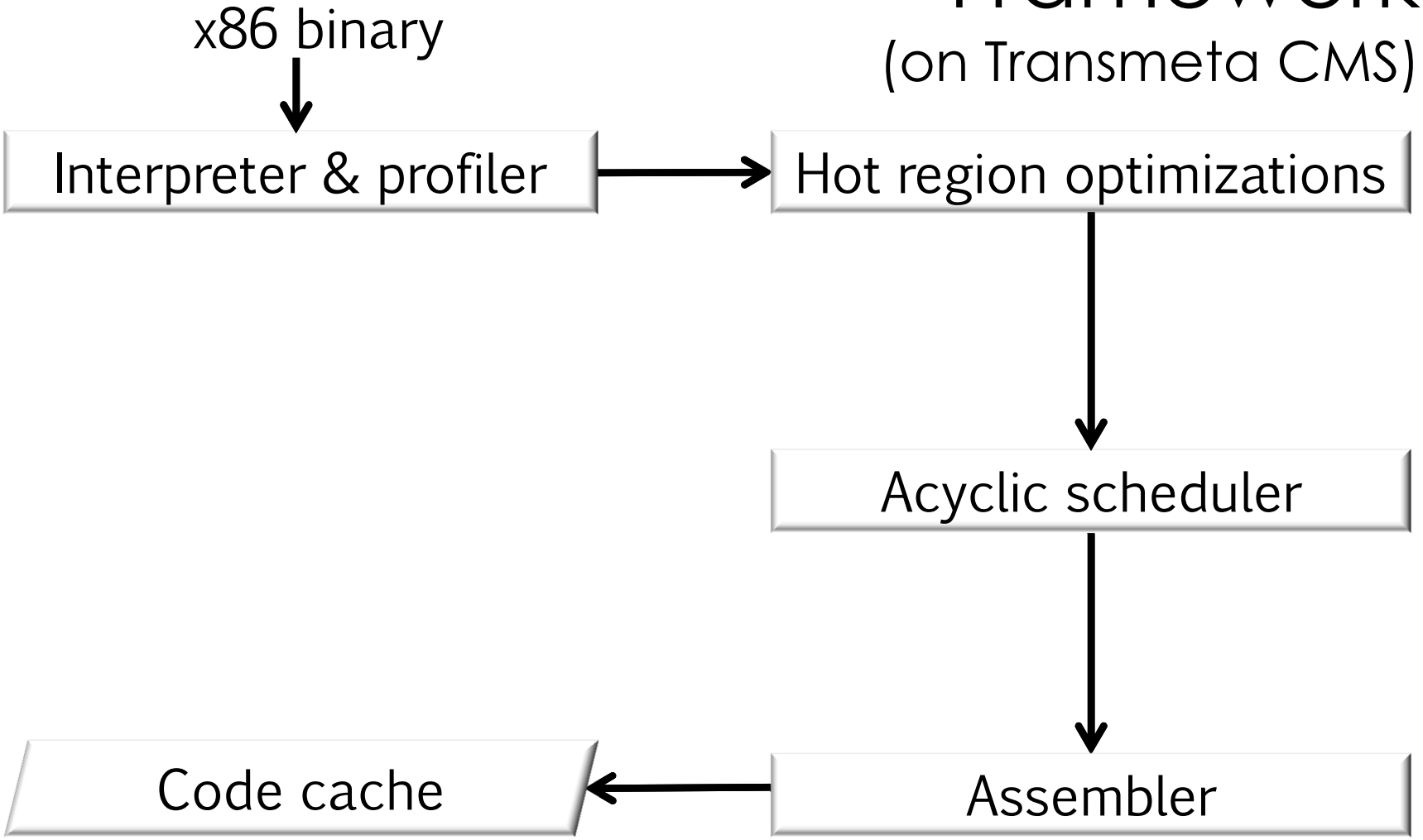
Framework
(on Transmeta CMS)



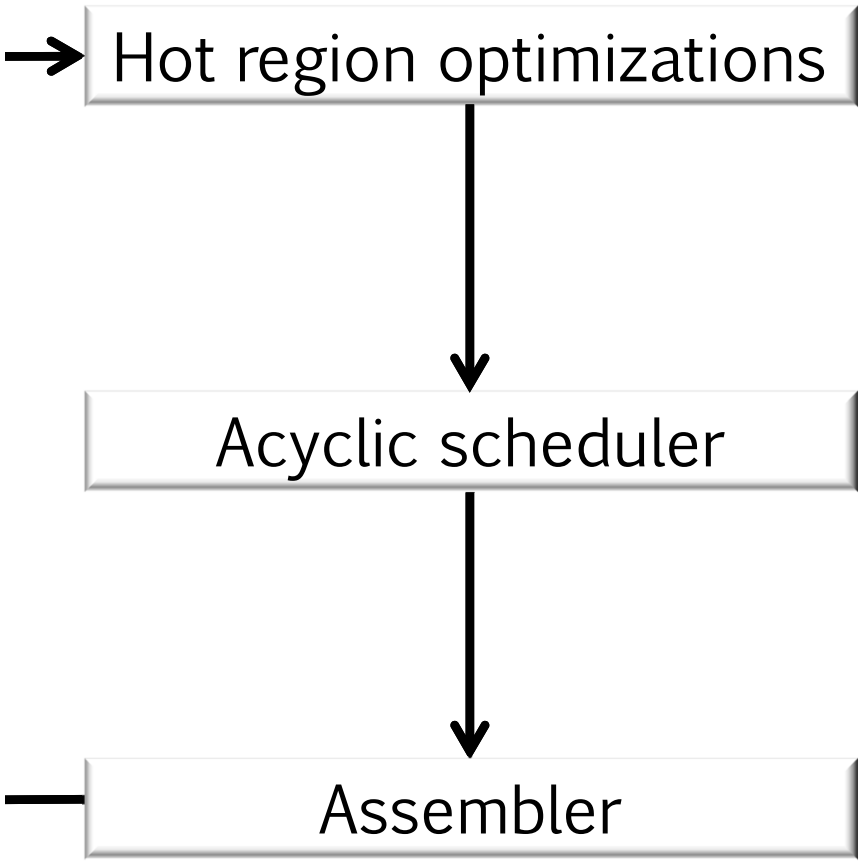
Framework
(on Transmeta CMS)



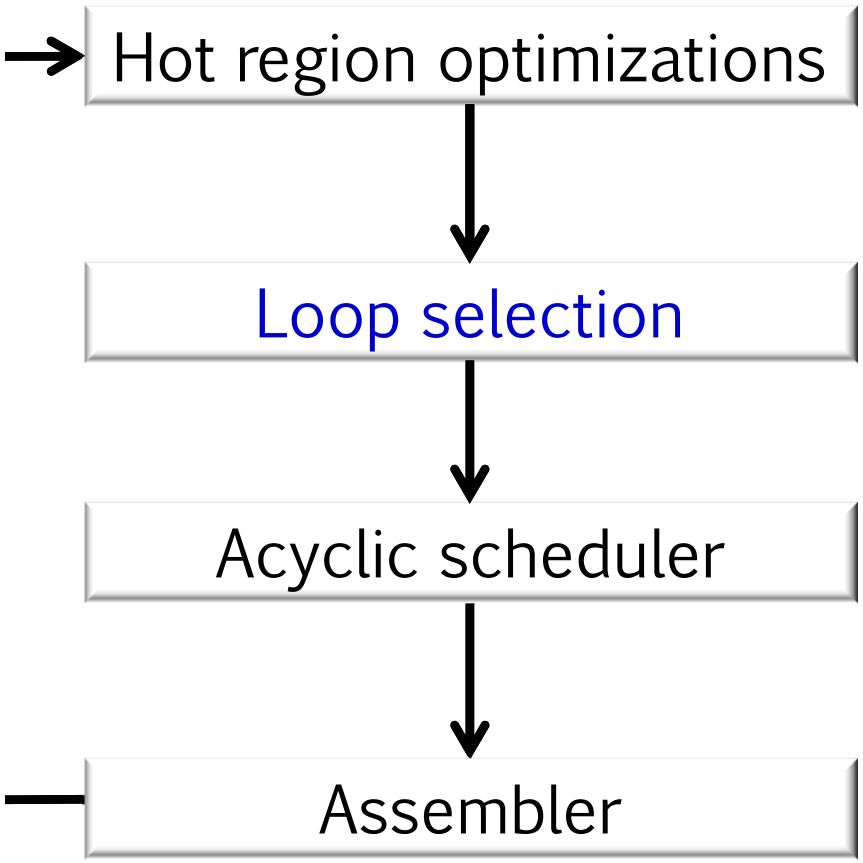
Framework
(on Transmeta CMS)



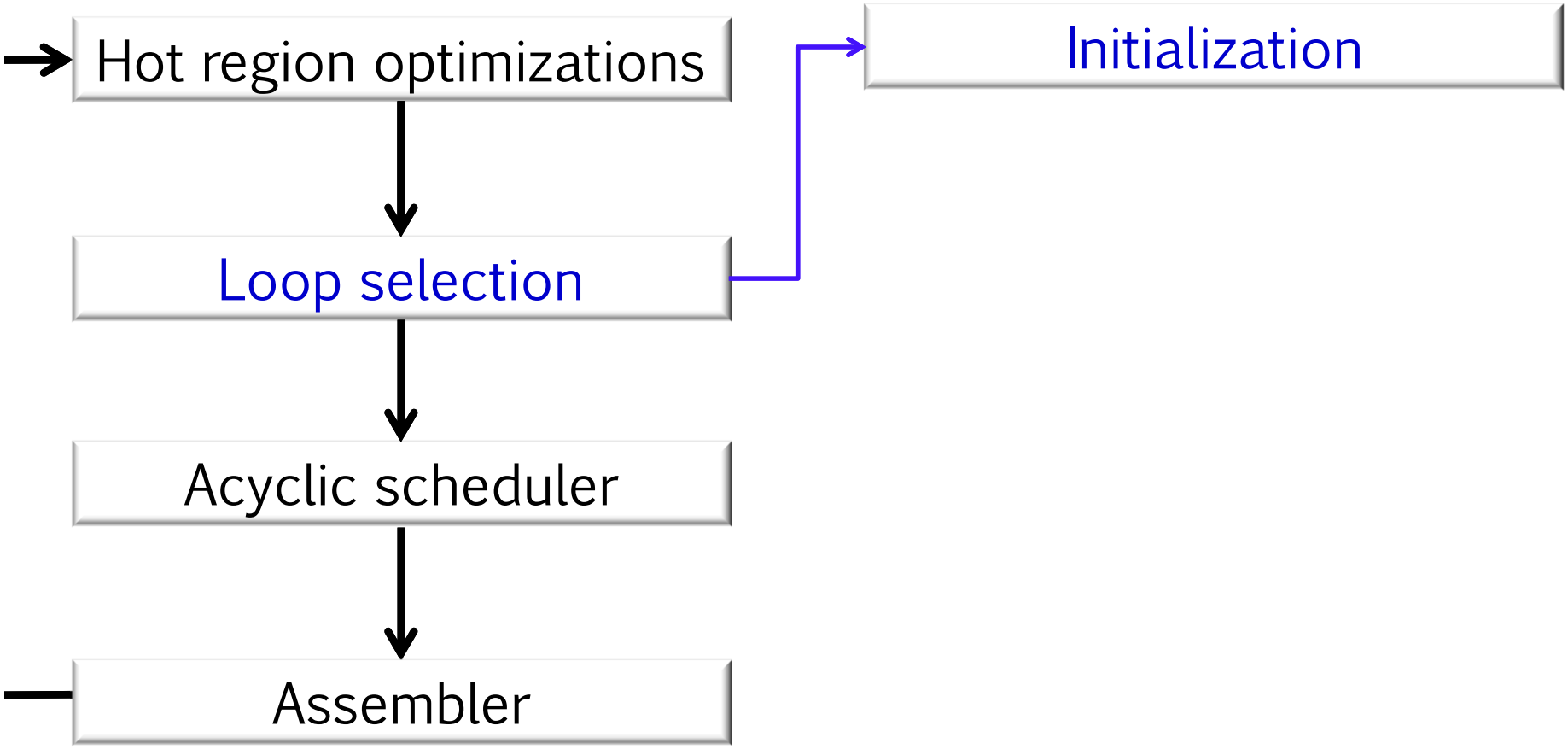
Framework
(on Transmeta CMS)



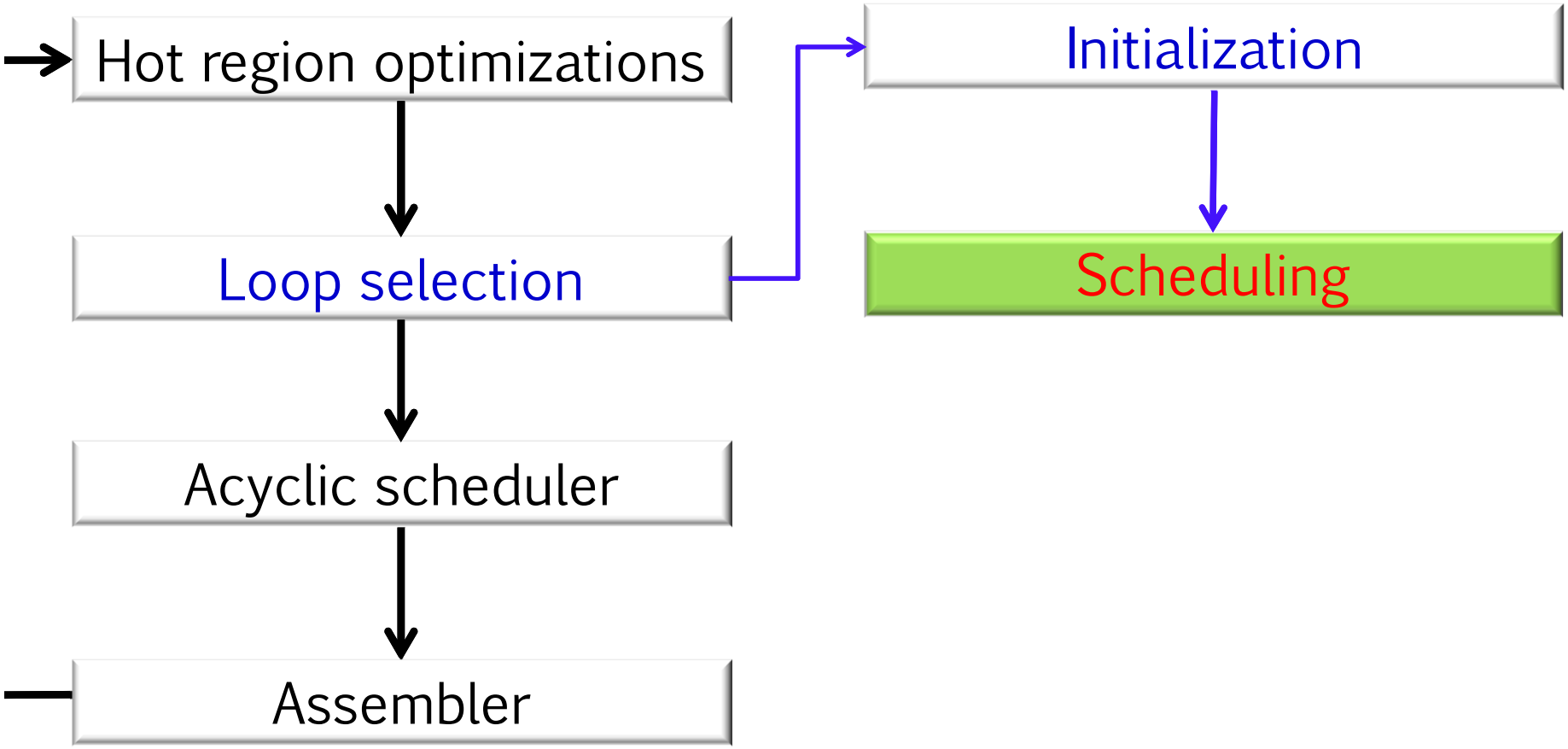
Framework
(on Transmeta CMS)



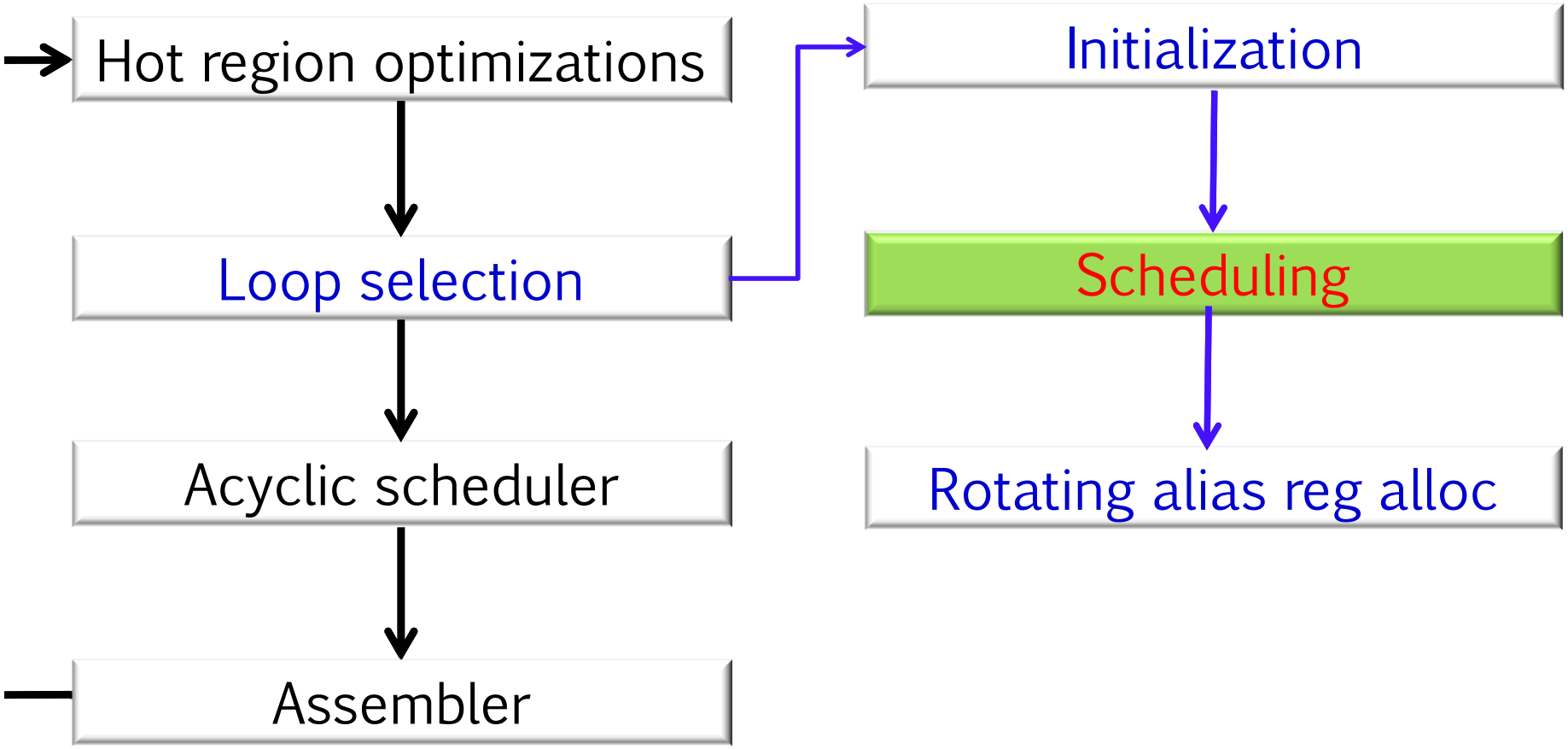
Framework
(on Transmeta CMS)



Framework
(on Transmeta CMS)

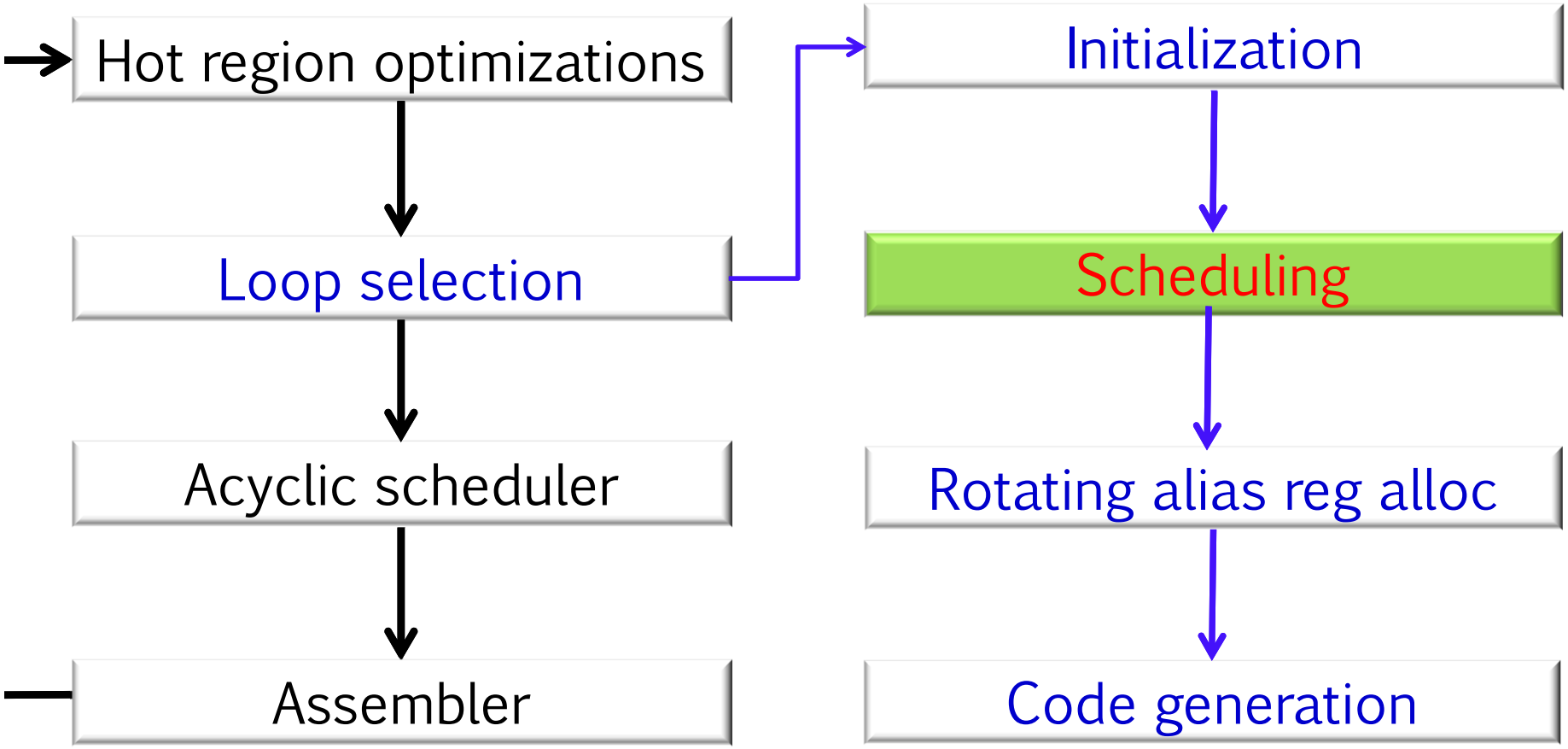


Framework
(on Transmeta CMS)



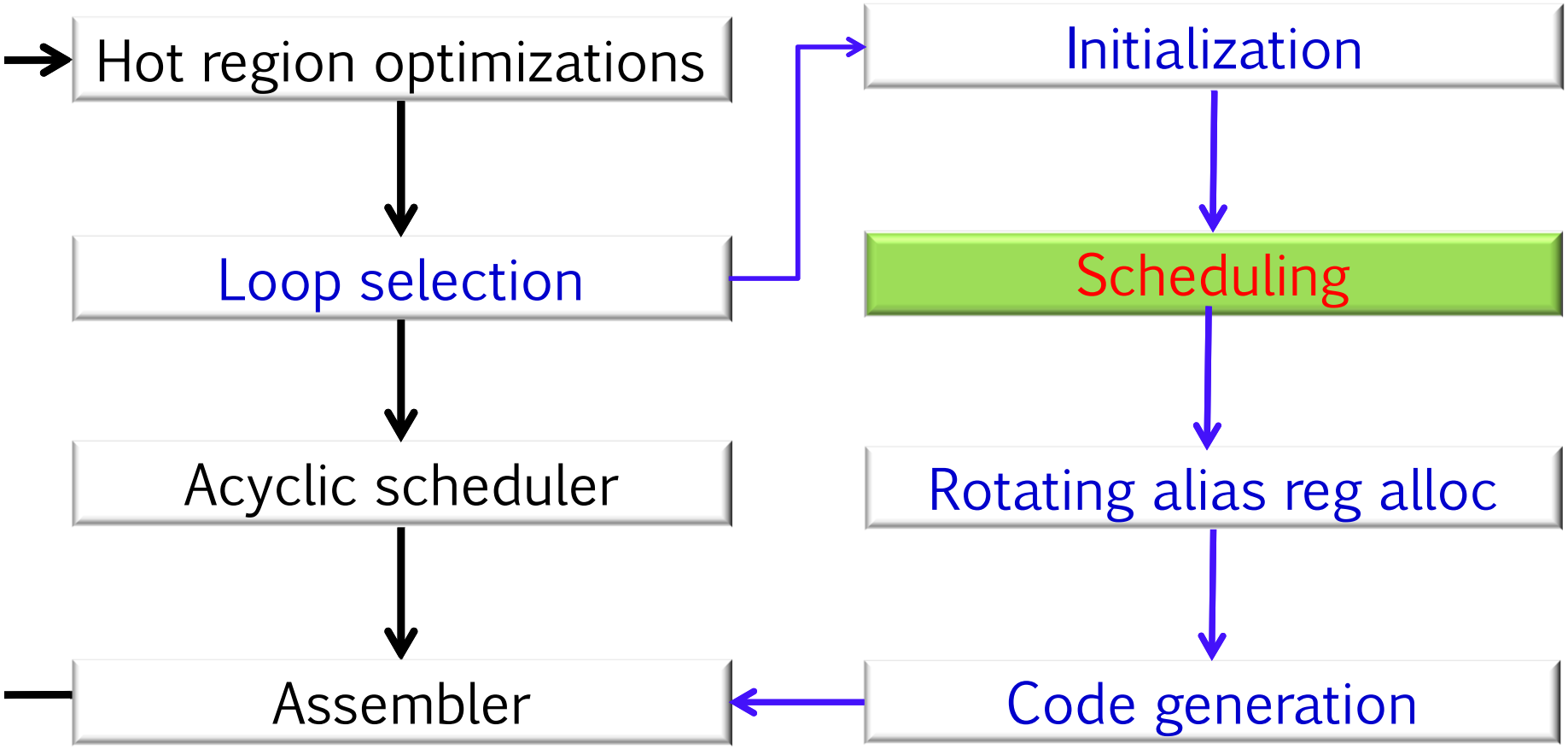
Framework

(on Transmeta CMS)

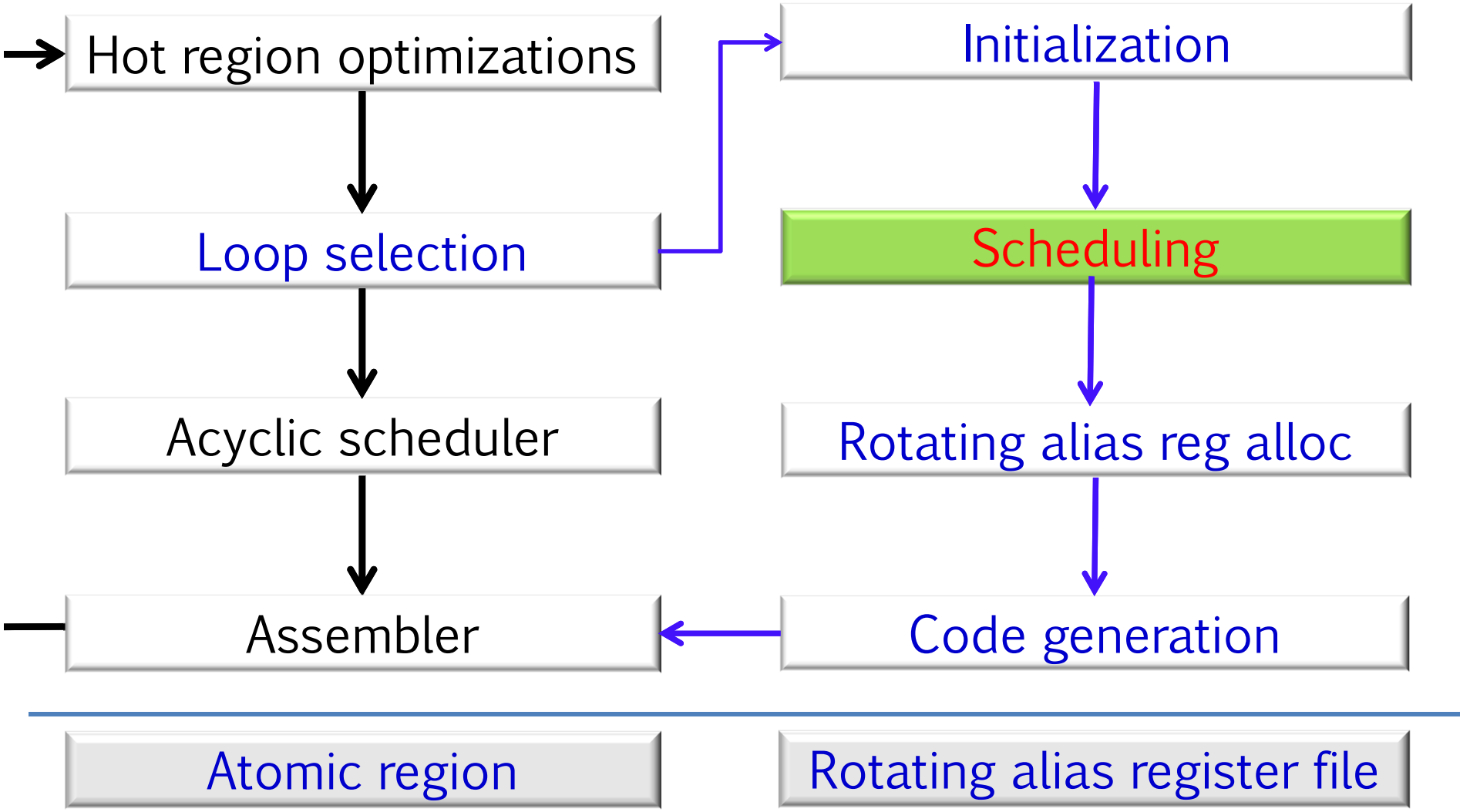


Framework

(on Transmeta CMS)



Framework
(on Transmeta CMS)



Scheduling is expensive

- NP-complete problem to find an optimal schedule [Colland et al. 1996]

Scheduling is expensive

- NP-complete problem to find an optimal schedule [Colland et al. 1996]
- $O(V^3)$ at least, exponential at worst [Rau et al. 1992]
  - V: number of operations

Scheduling is expensive

- NP-complete problem to find an optimal schedule [Colland et al. 1996]
- $O(V^3)$ at least, exponential at worst [Rau et al. 1992]
  - V: number of operations
- *Can we linearize software pipelining?*

Keep scheduling time under control

Keep scheduling time under control

- Schedule in linear time
  - Use either simple or fast sub-algorithms
    - Avoid cubic or exponential complexity

Keep scheduling time under control

- Schedule in linear time
  - Use either simple or fast sub-algorithms
    - Avoid cubic or exponential complexity
  - For iterative sub-algorithms, have a threshold: the maximum #iterations allowed

Keep scheduling time under control

- Schedule in linear time
  - Use either simple or fast sub-algorithms
    - Avoid cubic or exponential complexity
  - For iterative sub-algorithms, have a threshold: the maximum #iterations allowed
    - The smaller the threshold, the less the compile overhead
    - Once exceeded, abort software pipelining

Keep scheduling time under control

- Schedule in linear time
  - Use either simple or fast sub-algorithms
    - Avoid cubic or exponential complexity
  - For iterative sub-algorithms, have a threshold: the maximum #iterations allowed
    - The smaller the threshold, the less the compile overhead
    - Once exceeded, abort software pipelining
    - Key question: *How small can the threshold be?*

Keep scheduling time under control

- Schedule in linear time
  - Use either simple or fast sub-algorithms
    - Avoid cubic or exponential complexity
  - For iterative sub-algorithms, have a threshold: the maximum #iterations allowed
    - The smaller the threshold, the less the compile overhead
    - Once exceeded, abort software pipelining
    - Key question: *How small can the threshold be?*
- Find a good enough schedule
  - No backtracking
  - Priority function: approximate and never update
  - Separate dependence and resource constraints
  - Separate local and loop-carried dependences

Keep scheduling time under control

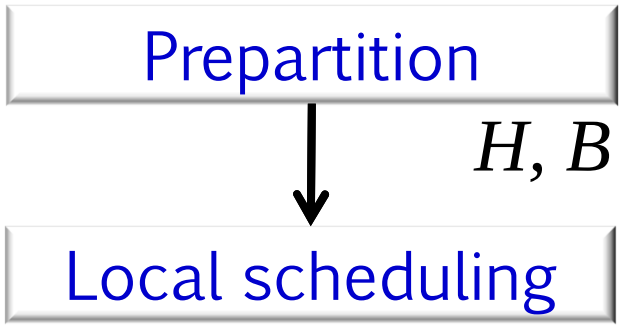
- Schedule in linear time
  - Use either simple or fast sub-algorithms
    - Avoid cubic or exponential complexity
  - For iterative sub-algorithms, have a threshold: the maximum #iterations allowed
    - The smaller the threshold, the less the compile overhead
    - Once exceeded, abort software pipelining
    - Key question: *How small can the threshold be?*
- Find a good enough schedule
  - No backtracking
  - Priority function: approximate and never update
  - Separate dependence and resource constraints
  - Separate local and loop-carried dependences
- Iteratively improve a schedule

Just-In-Time Software Pipelining

Just-In-Time Software Pipelining

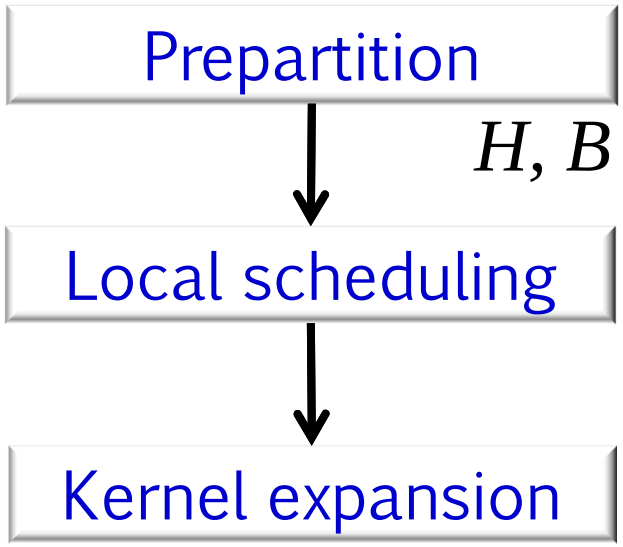
- Quickly creates an initial schedule to start with

Just-In-Time Software Pipelining



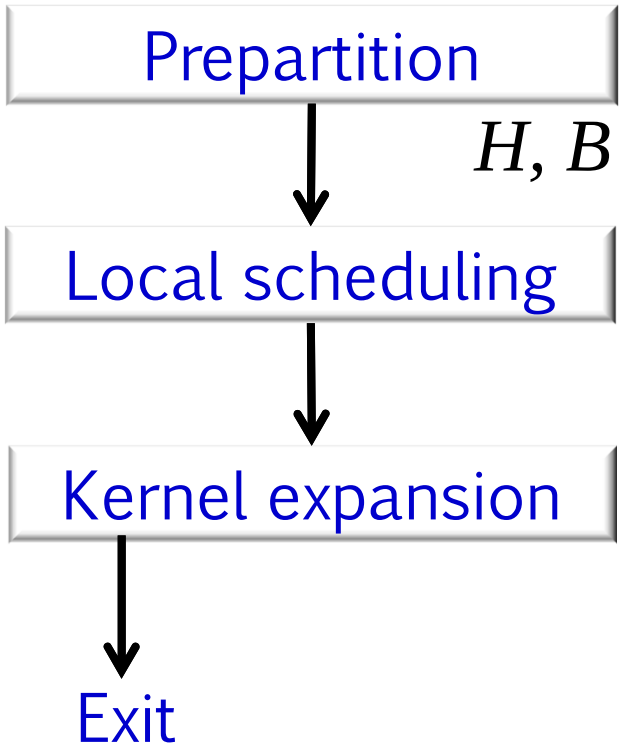
- Quickly creates an initial schedule to start with
- Handles local dependences and resources

Just-In-Time Software Pipelining



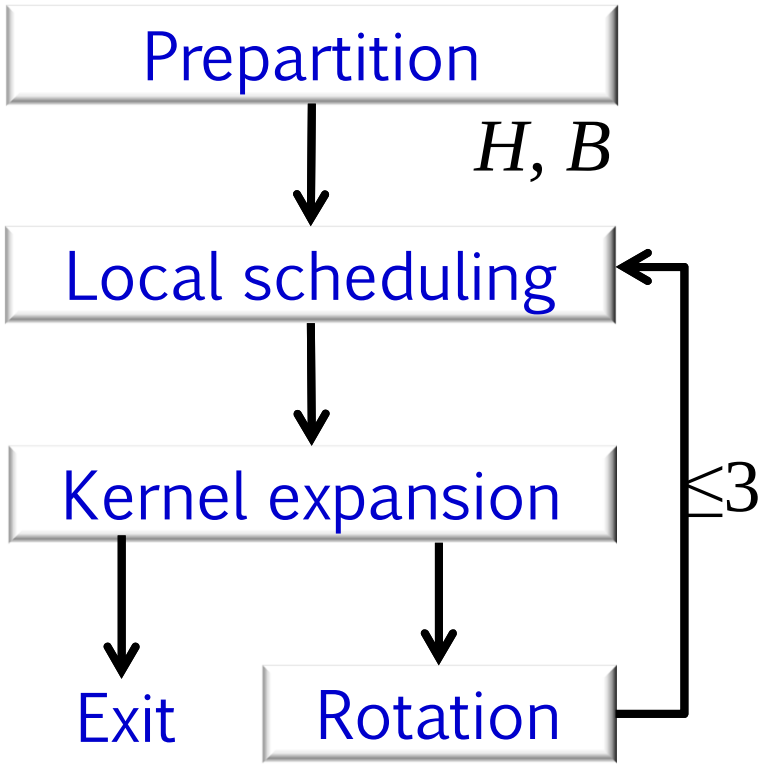
- Quickly creates an initial schedule to start with
- Handles local dependences and resources
- Adjusts the schedule for loop-carried dependences

Just-In-Time Software Pipelining



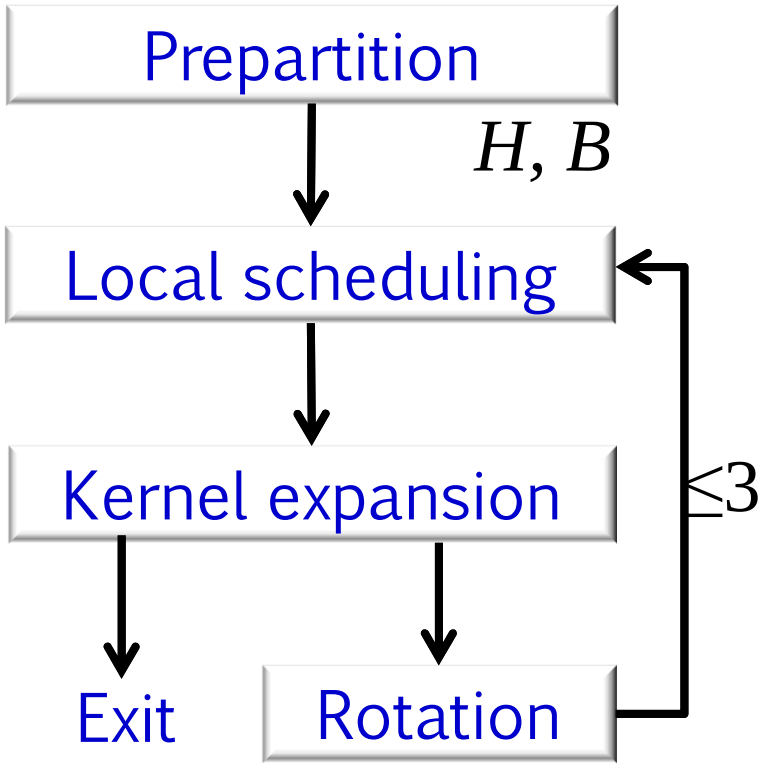
- Quickly creates an initial schedule to start with
- Handles local dependences and resources
- Adjusts the schedule for loop-carried dependences

Just-In-Time Software Pipelining



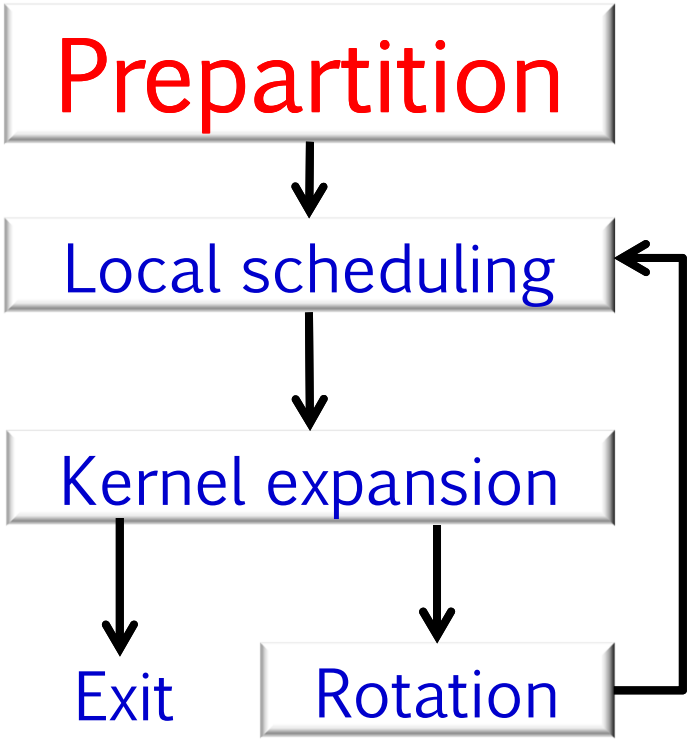
- Quickly creates an initial schedule to start with
- Handles local dependences and resources
- Adjusts the schedule for loop-carried dependences
- Iteratively improves the schedule

Just-In-Time Software Pipelining



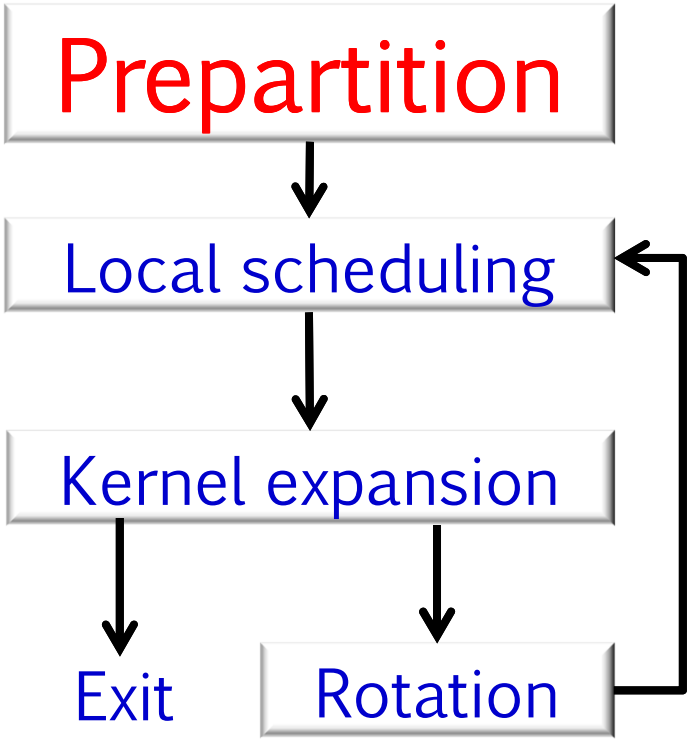
- Quickly creates an initial schedule to start with
- Handles local dependences and resources
- Adjusts the schedule for loop-carried dependences
- Iteratively improves the schedule

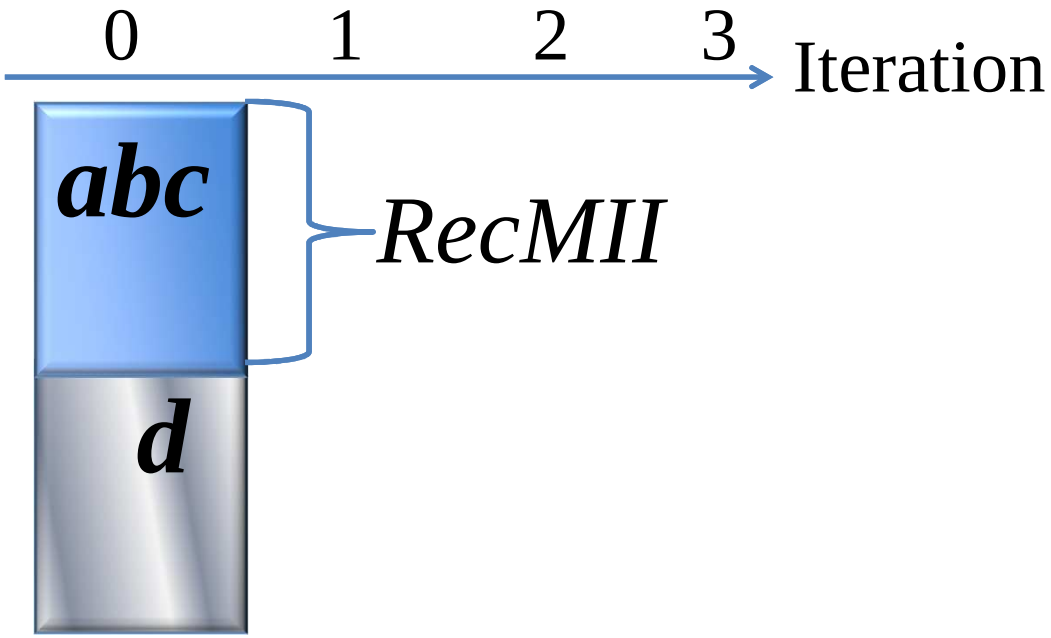
- Time complexity: O(V+E)
  - V: #operations E: #dependences



- [ ] Local dependences
- [ ] Resources
- [ ] Loop-carried dependences

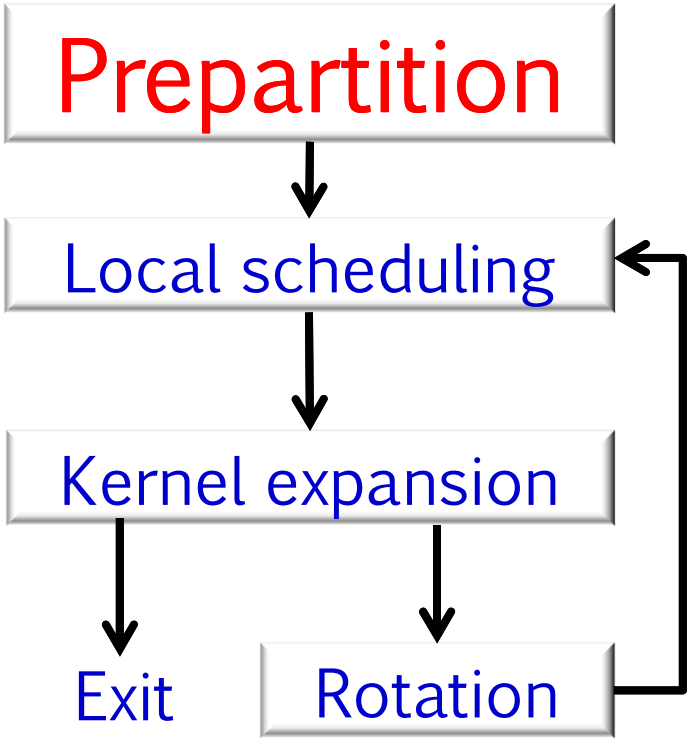
Illustration

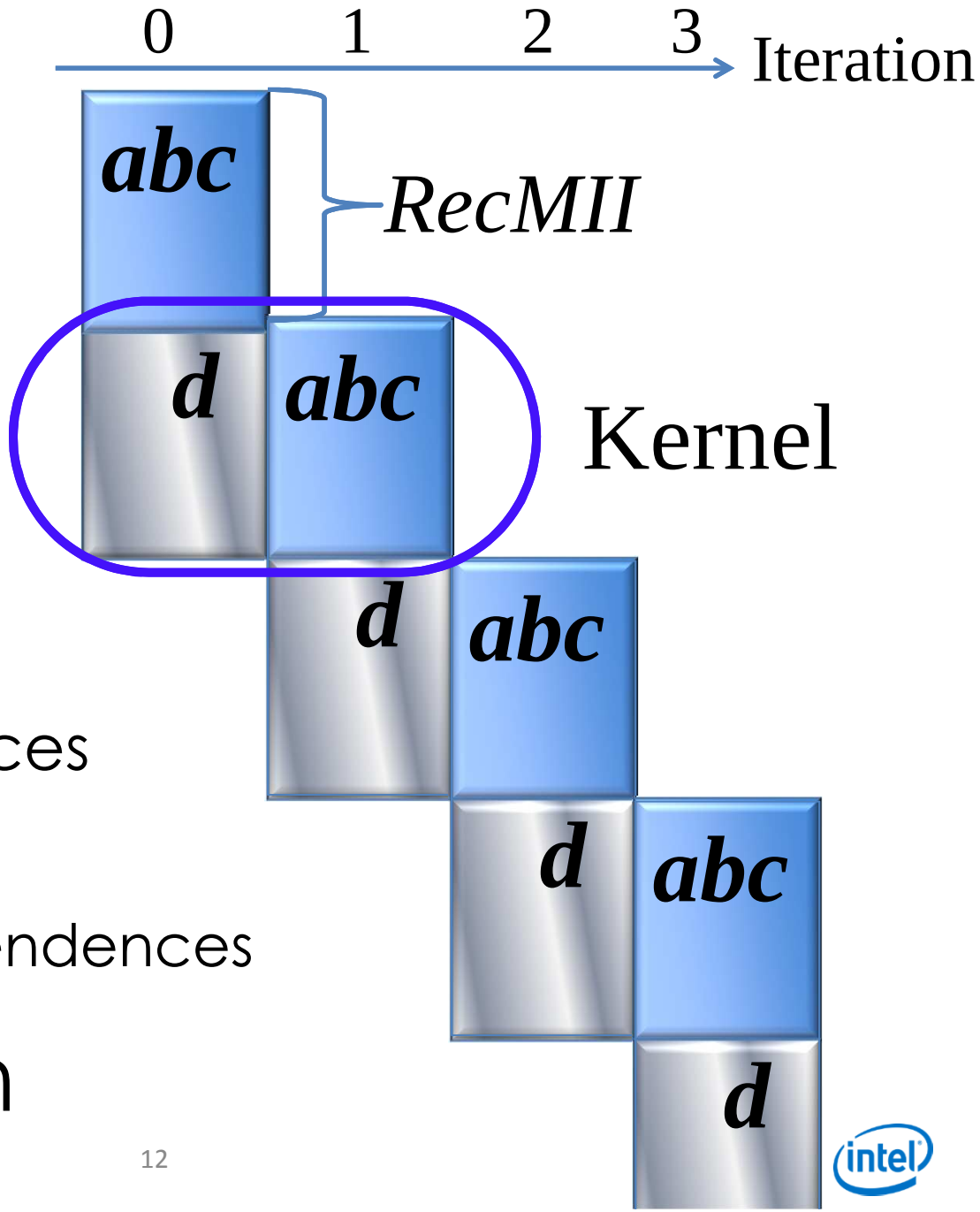




- Local dependences
- Resources
- Loop-carried dependences

Illustration





- Local dependences
- Resources
- Loop-carried dependences

Illustration

Calculating RecMII

- Recurrence Minimum II
  - II determined by the biggest dependence cycles
  - Needed by almost every software pipelining method

Calculating RecMII

- Recurrence Minimum II
  - II determined by the biggest dependence cycles
  - Needed by almost every software pipelining method

Conventional

$O(V^3)$

Calculating RecMII

- Recurrence Minimum II
  - II determined by the biggest dependence cycles
  - Needed by almost every software pipelining method

Conventional

$O(V^3)$

- Turn the problem into a Markov decision process
  - 1st time Howard algorithm is applied to pipelining

Calculating RecMII

- Recurrence Minimum II
  - II determined by the biggest dependence cycles
  - Needed by almost every software pipelining method

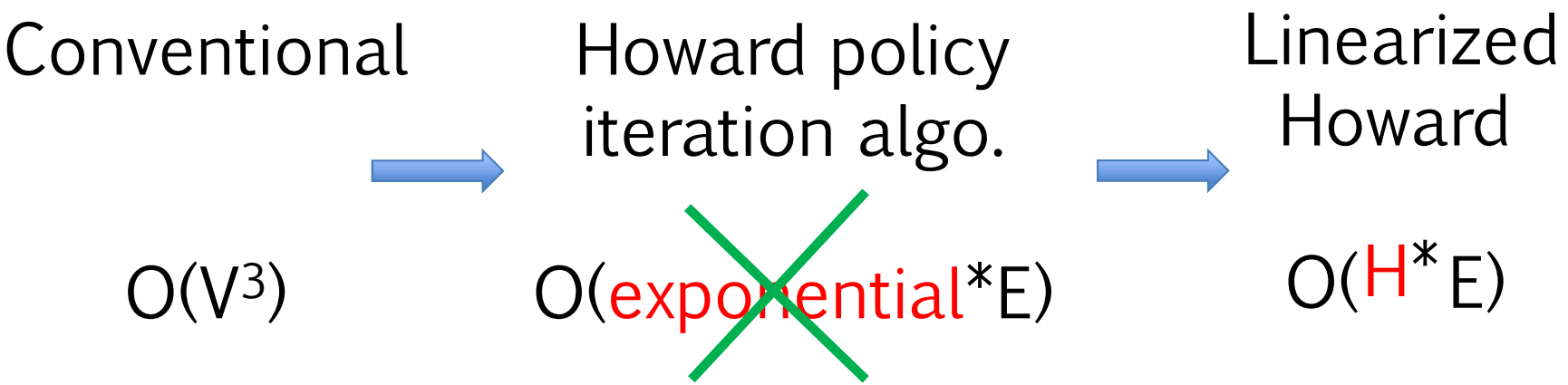
Conventional → Howard policy iteration algo.

$O(V^3)$ → $O(\text{exponential}*E)$

- Turn the problem into a Markov decision process
  - 1st time Howard algorithm is applied to pipelining

Calculating RecMII

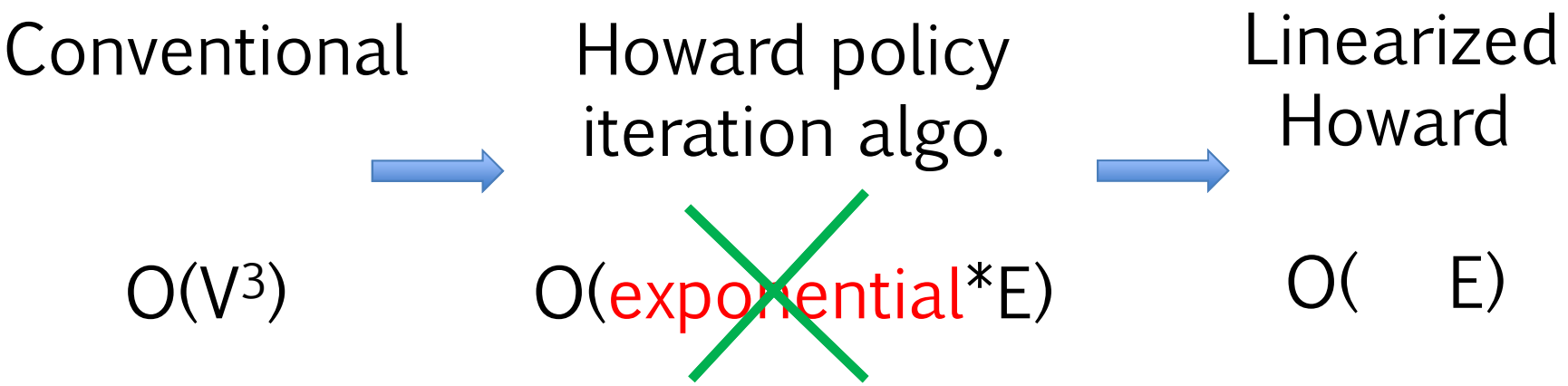
- Recurrence Minimum II
  - II determined by the biggest dependence cycles
  - Needed by almost every software pipelining method



- Turn the problem into a Markov decision process
  - 1st time Howard algorithm is applied to pipelining
- Linearize Howard with a small constant H

Calculating RecMII

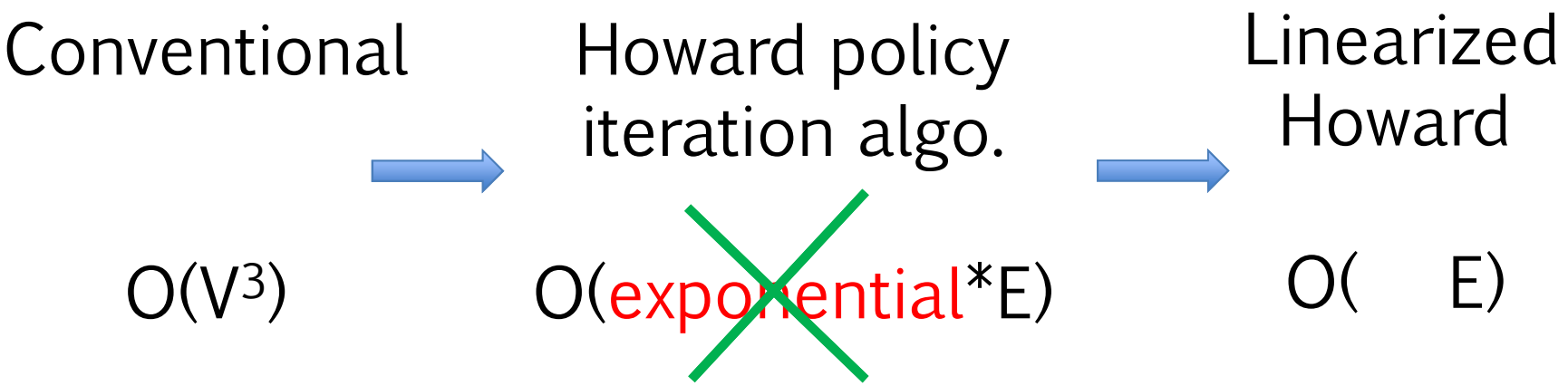
- Recurrence Minimum II
  - II determined by the biggest dependence cycles
  - Needed by almost every software pipelining method



- Turn the problem into a Markov decision process
  - 1st time Howard algorithm is applied to pipelining
- Linearize Howard with a small constant H

Calculating RecMII

- Recurrence Minimum II
  - II determined by the biggest dependence cycles
  - Needed by almost every software pipelining method



- Turn the problem into a Markov decision process
  - 1st time Howard algorithm is applied to pipelining
- Linearize Howard with a small constant H
- A by-product: critical operations

Divide operations into stages

Bellman-Ford

O(V*E)

Divide operations into stages

Bellman-Ford

O(V*E)

- Also an iterative sub-algorithm

Divide operations into stages

Bellman-Ford

Linearized Bellman-Ford

O(V*E)

O(B* E)

- Also an iterative sub-algorithm
- Linearize it with a constant B

Divide operations into stages

Bellman-Ford → Linearized Bellman-Ford

O(V*E) → O(B* E)

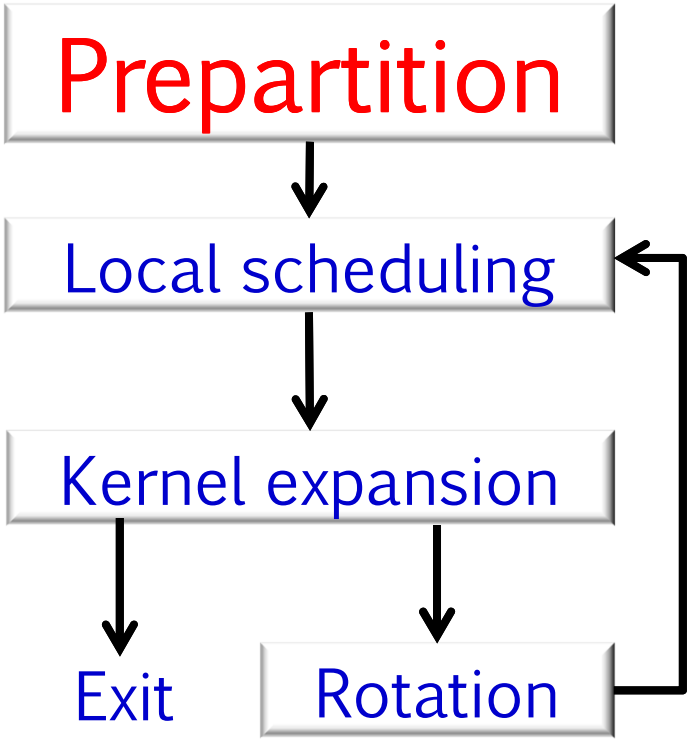
- Also an iterative sub-algorithm
- Linearize it with a constant B
- Specific order in visiting edges
  - Scan nodes in sequential order, and visit their incoming edges
    - Values are propagated along local edges in the 1st itr.
    - Values are propagated along loop-carried edges in the 2nd itr.

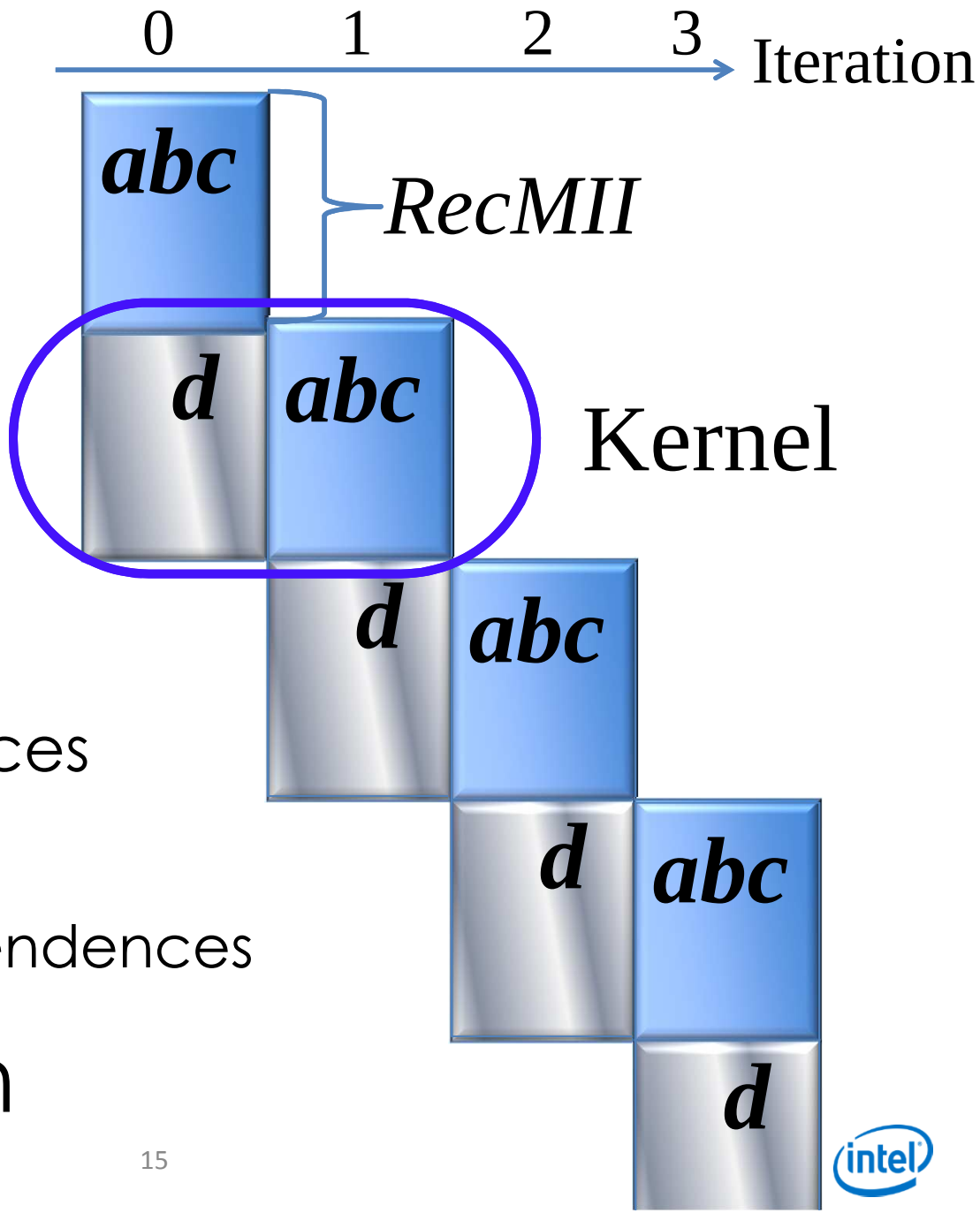
Divide operations into stages

Bellman-Ford → Linearized Bellman-Ford

O(V*E) → O(E)

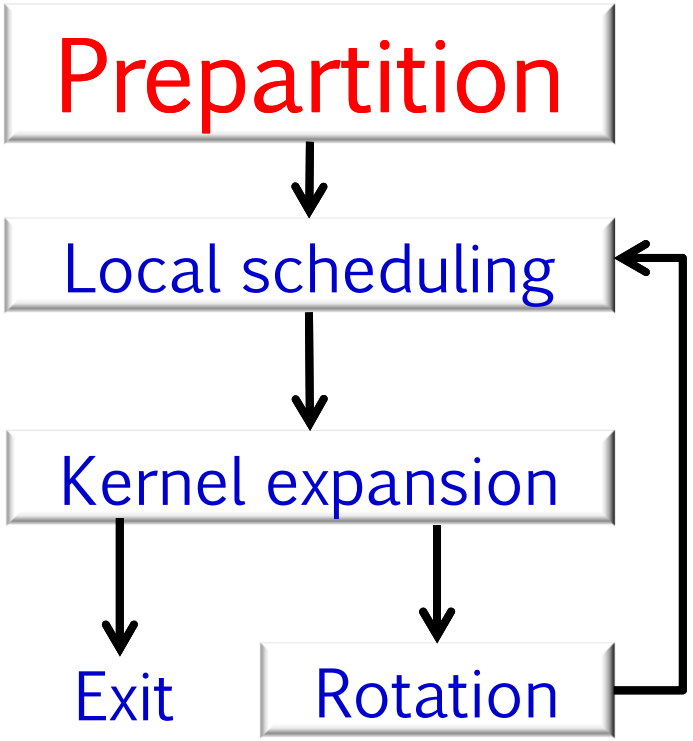
- Also an iterative sub-algorithm
- Linearize it with a constant B
- Specific order in visiting edges
  - Scan nodes in sequential order, and visit their incoming edges
    - Values are propagated along local edges in the 1st itr.
    - Values are propagated along loop-carried edges in the 2nd itr.

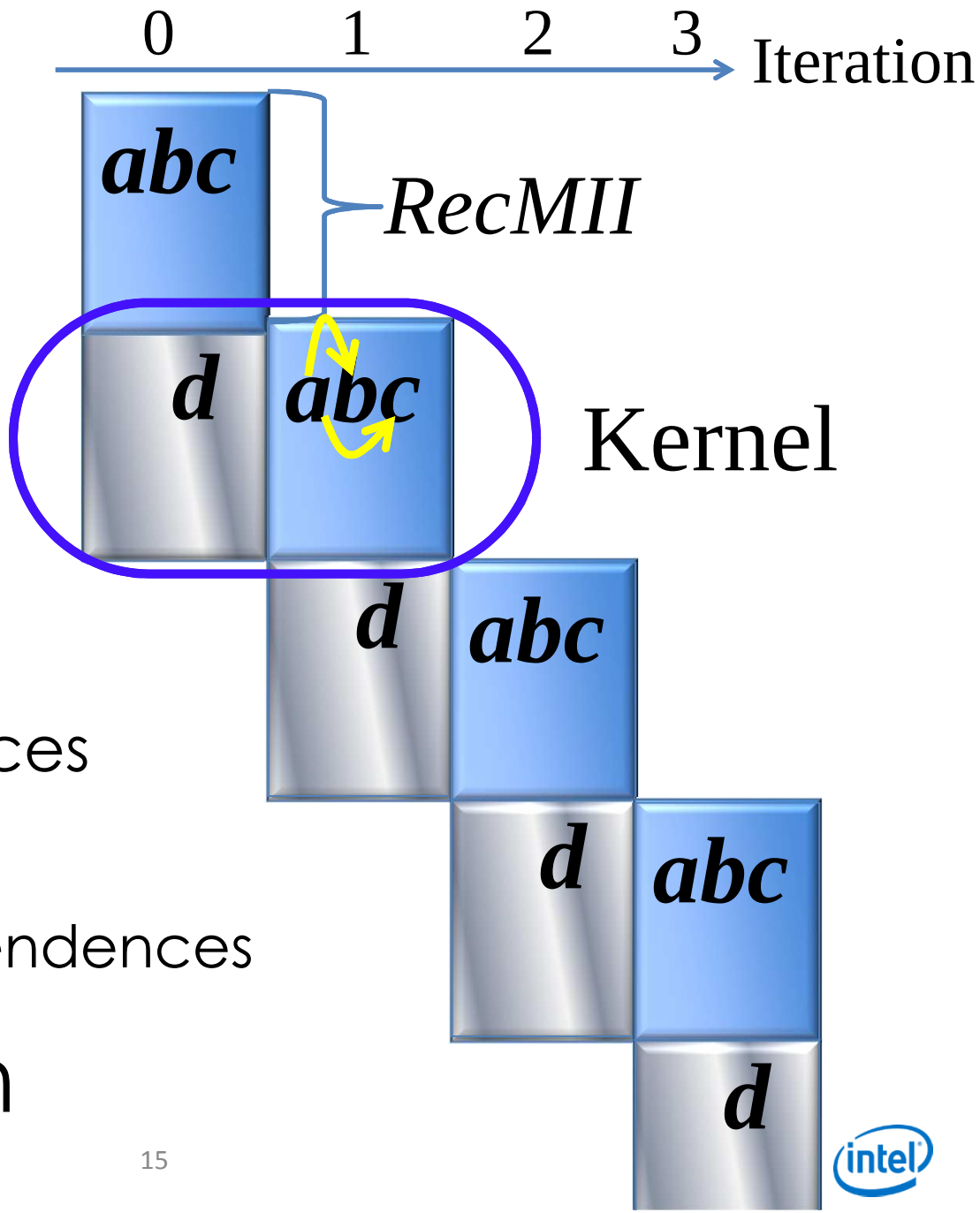




- Local dependences
- Resources
- Loop-carried dependences

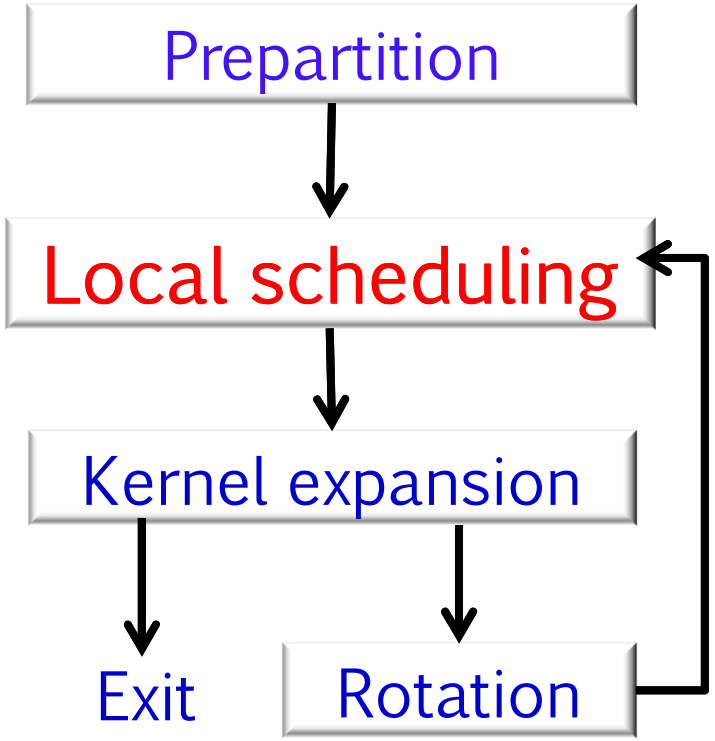
Illustration

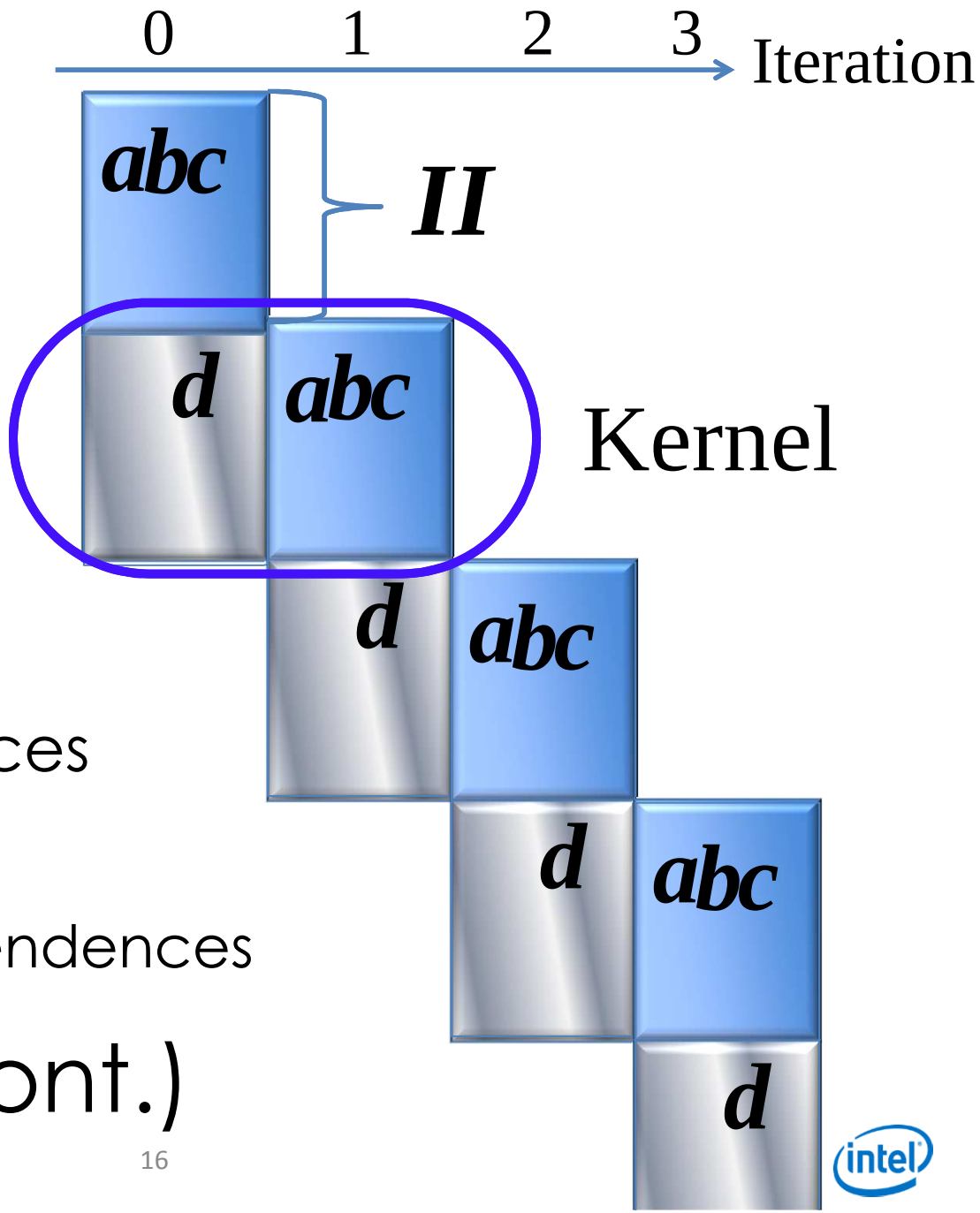




- Local dependences
- Resources
- Loop-carried dependences

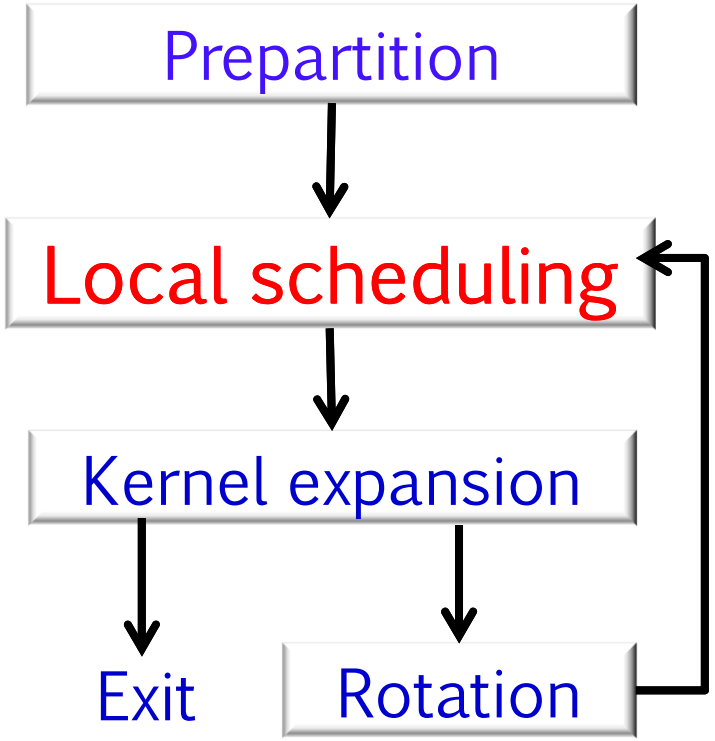
Illustration

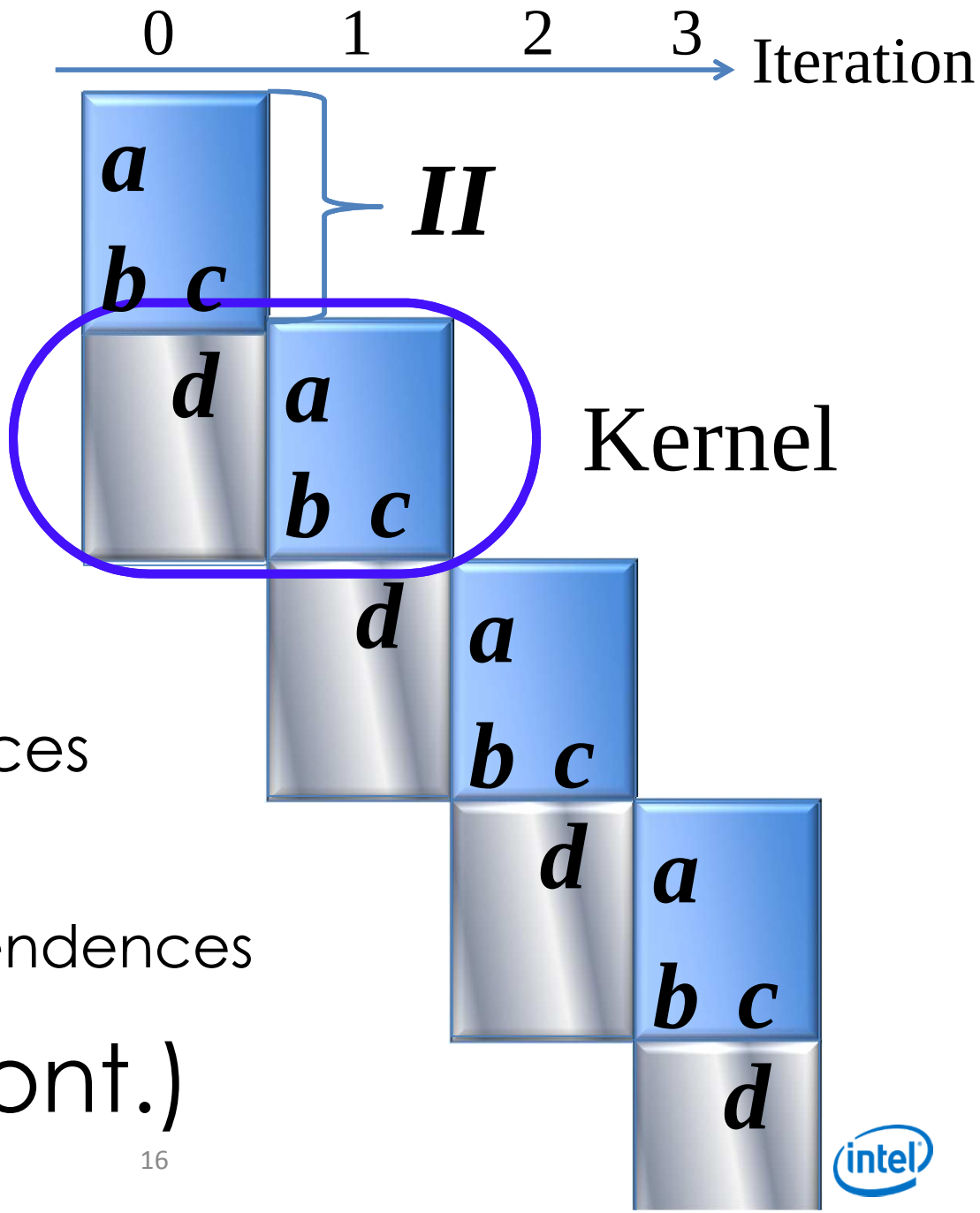




- Local dependences
- Resources
- Loop-carried dependences

Illustration (Cont.)





☒ Local dependences

☒ Resources

☐ Loop-carried dependences

Illustration (Cont.)

Local scheduling

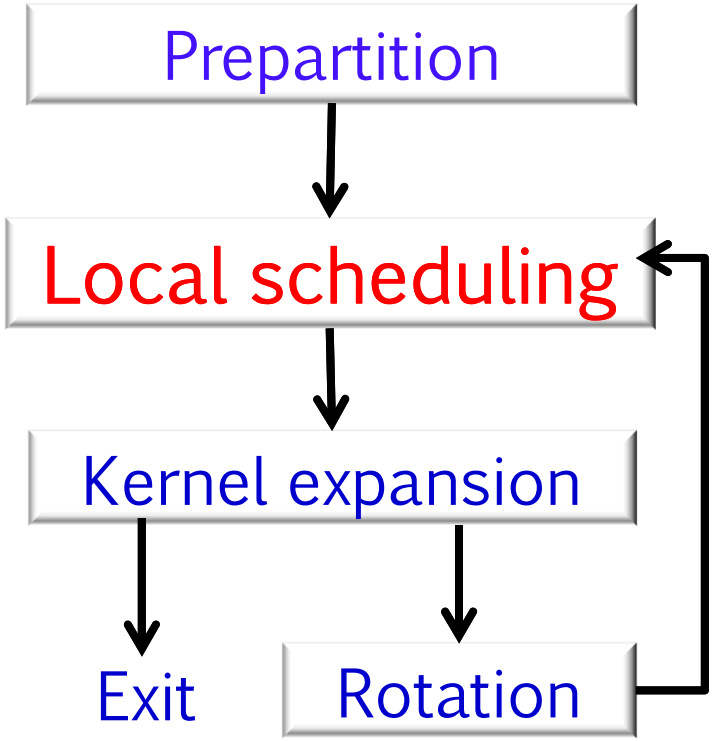
- Any local scheduling algorithm can be used
  - E.g. list scheduling

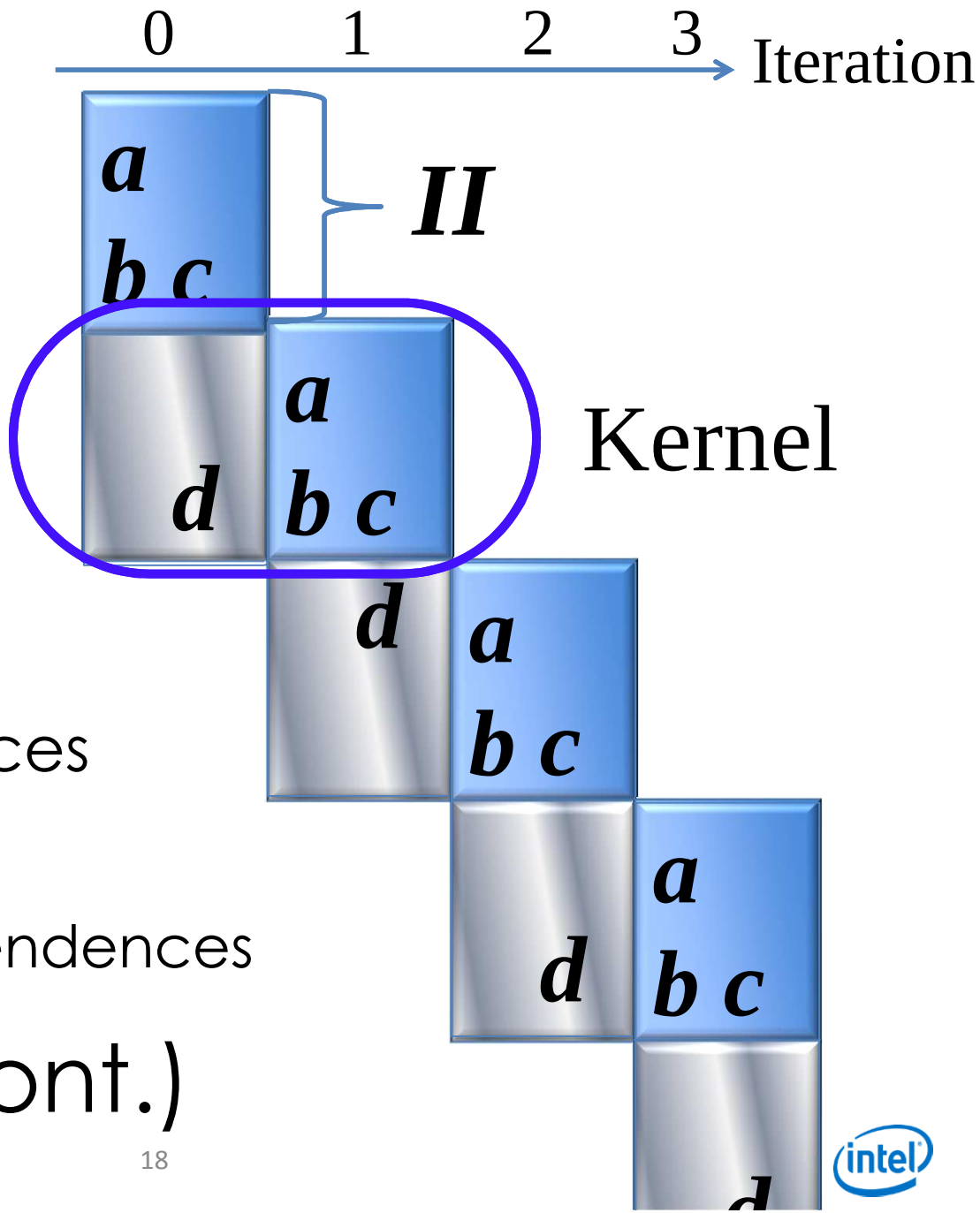
Local scheduling

- Any local scheduling algorithm can be used
  - E.g. list scheduling
- Weakness:
  - Loop-carried dependences may be violated

Local scheduling

- Any local scheduling algorithm can be used
  - E.g. list scheduling
- Weakness:
  - Loop-carried dependences may be violated
- To reduce the chance of violation:
  - Before scheduling, priority function considers loop-carried dependences in advance
  - Prioritize critical operations



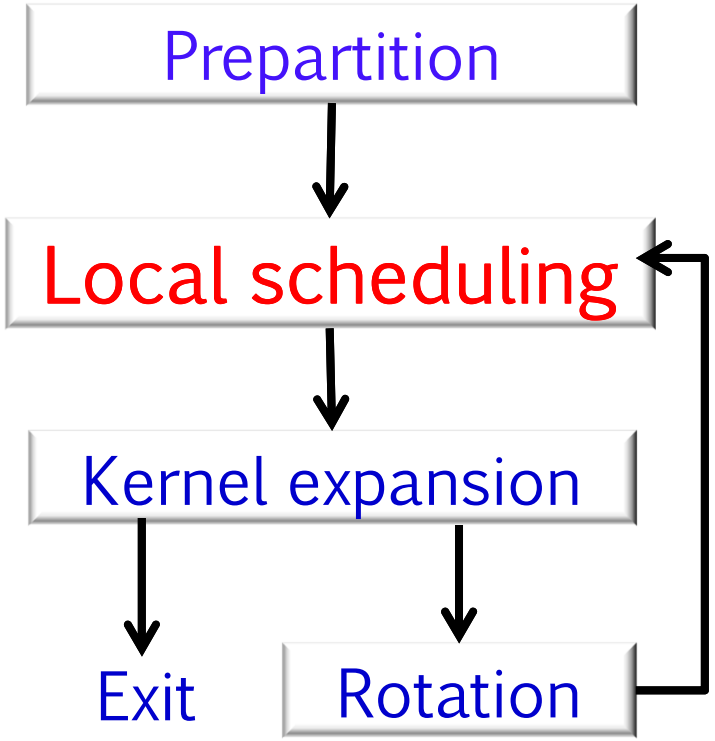


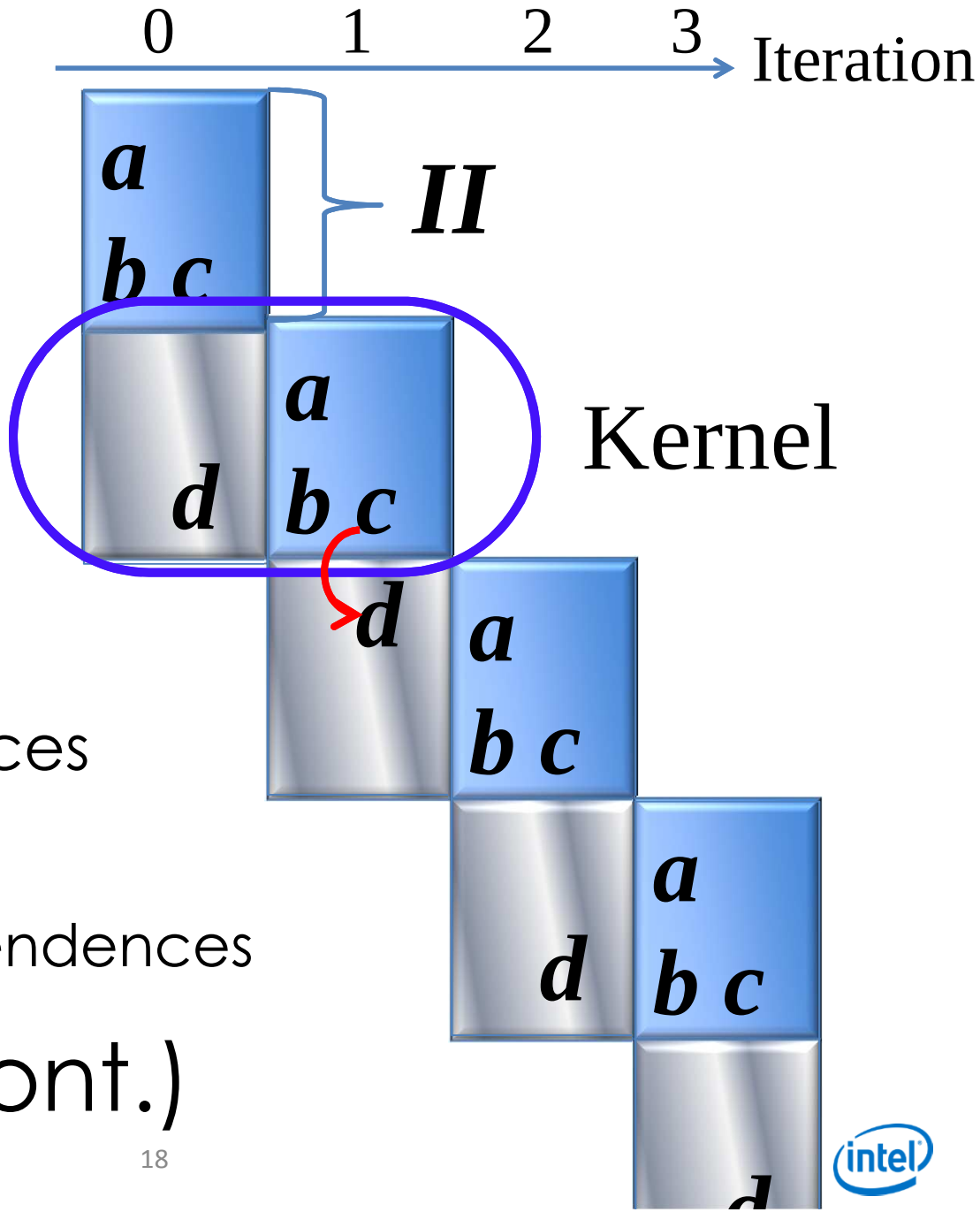
☒ Local dependences

☒ Resources

☐ Loop-carried dependences

Illustration (Cont.)



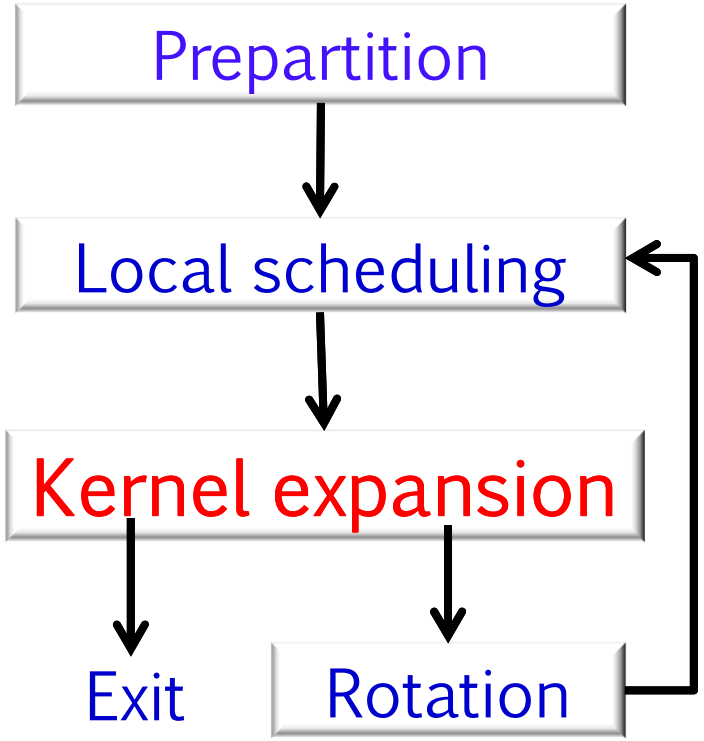


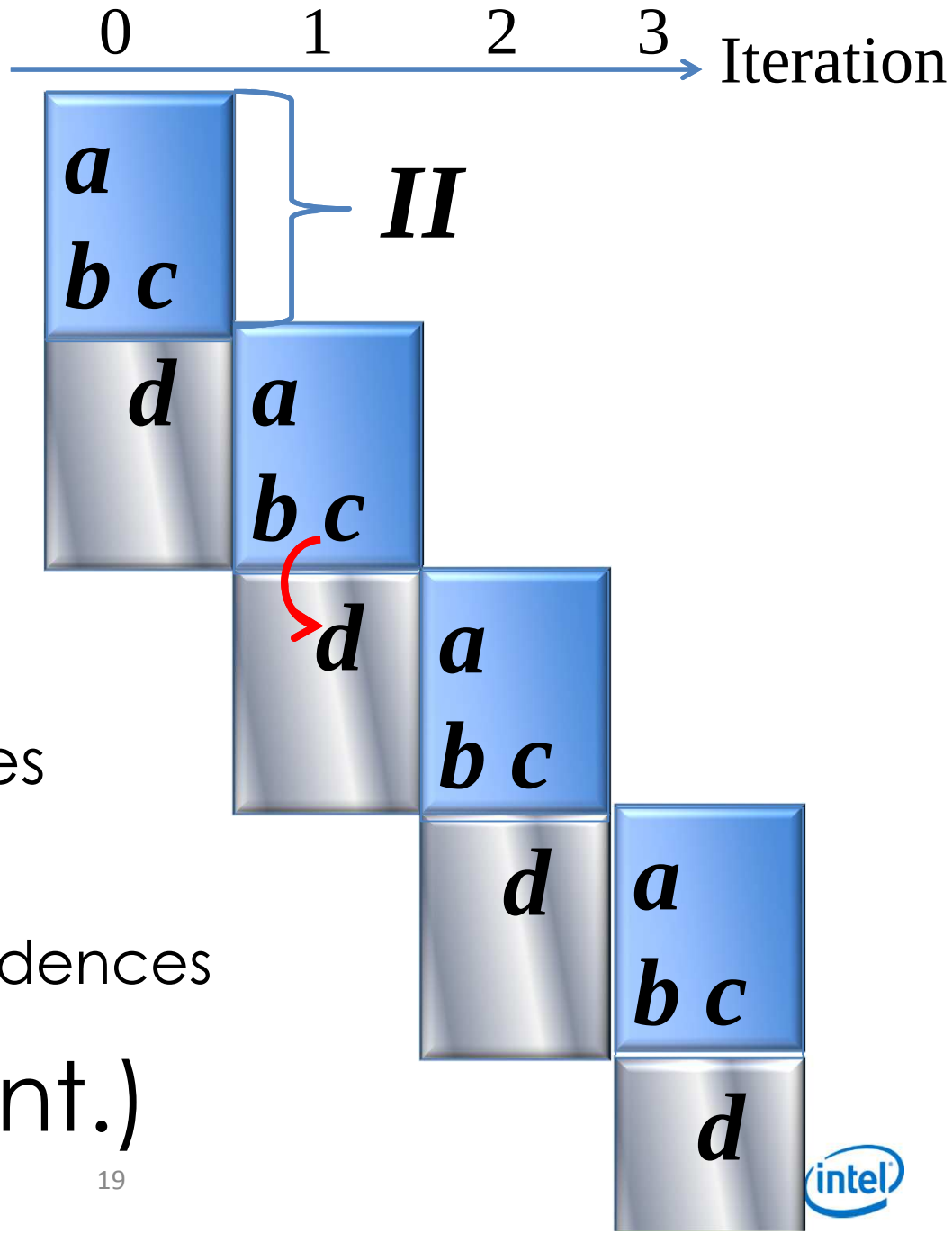
☒ Local dependences

☒ Resources

☐ Loop-carried dependences

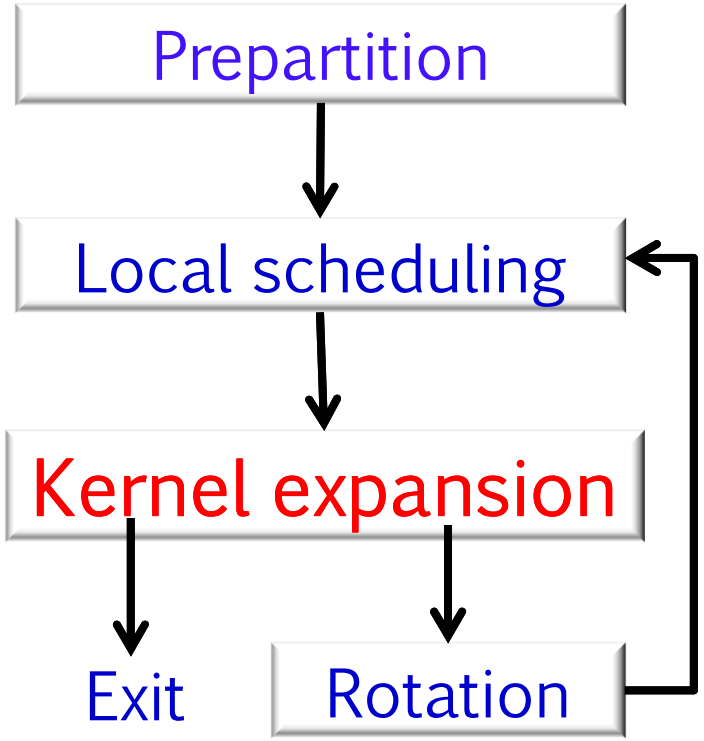
Illustration (Cont.)

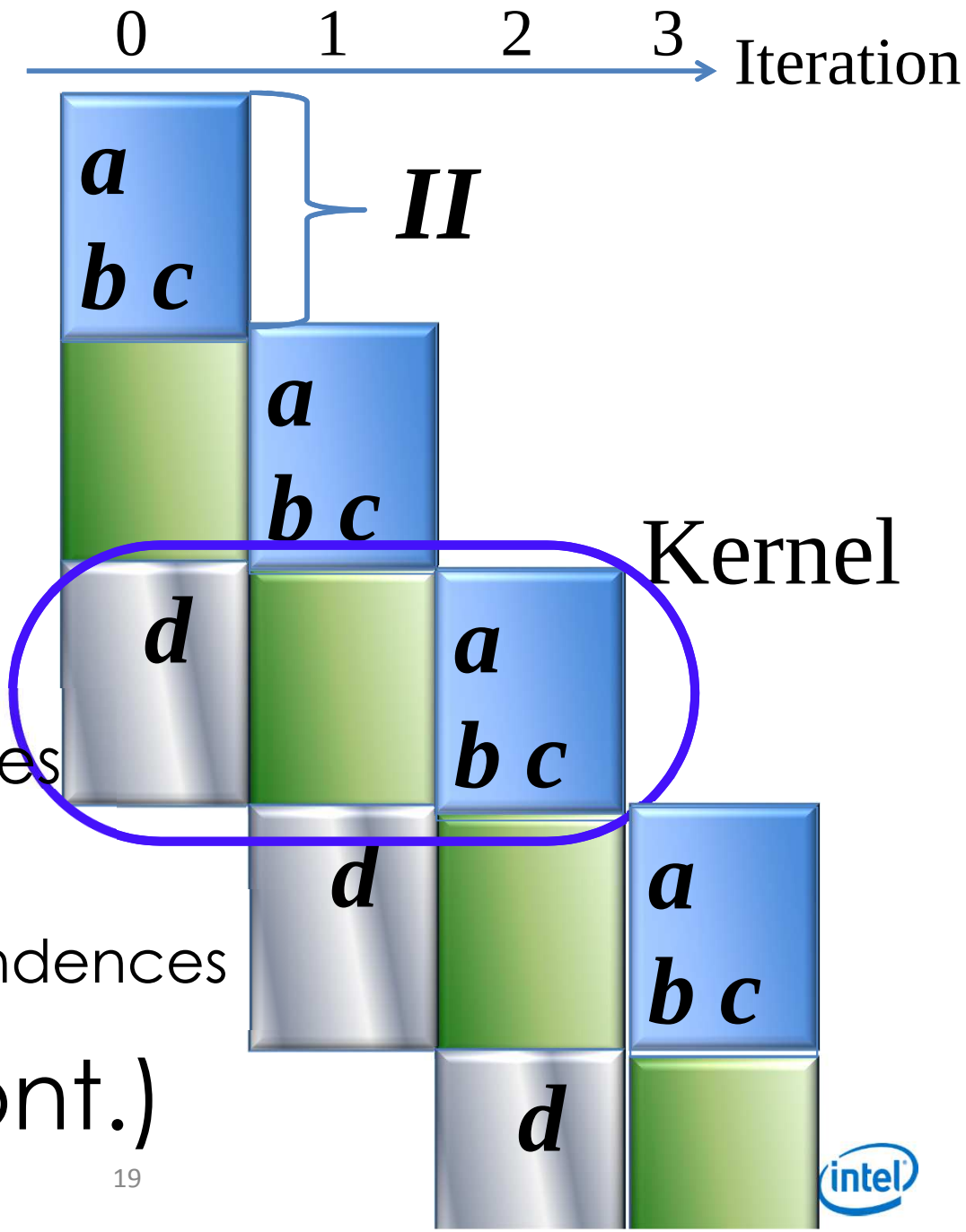




- [x] Local dependences
- [x] Resources
- [ ] Loop-carried dependences

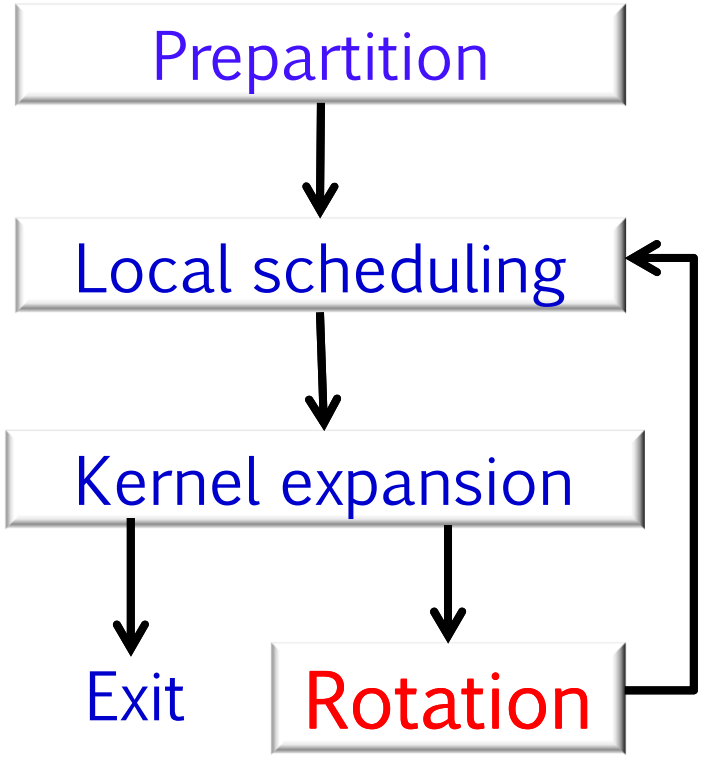
Illustration (Cont.)

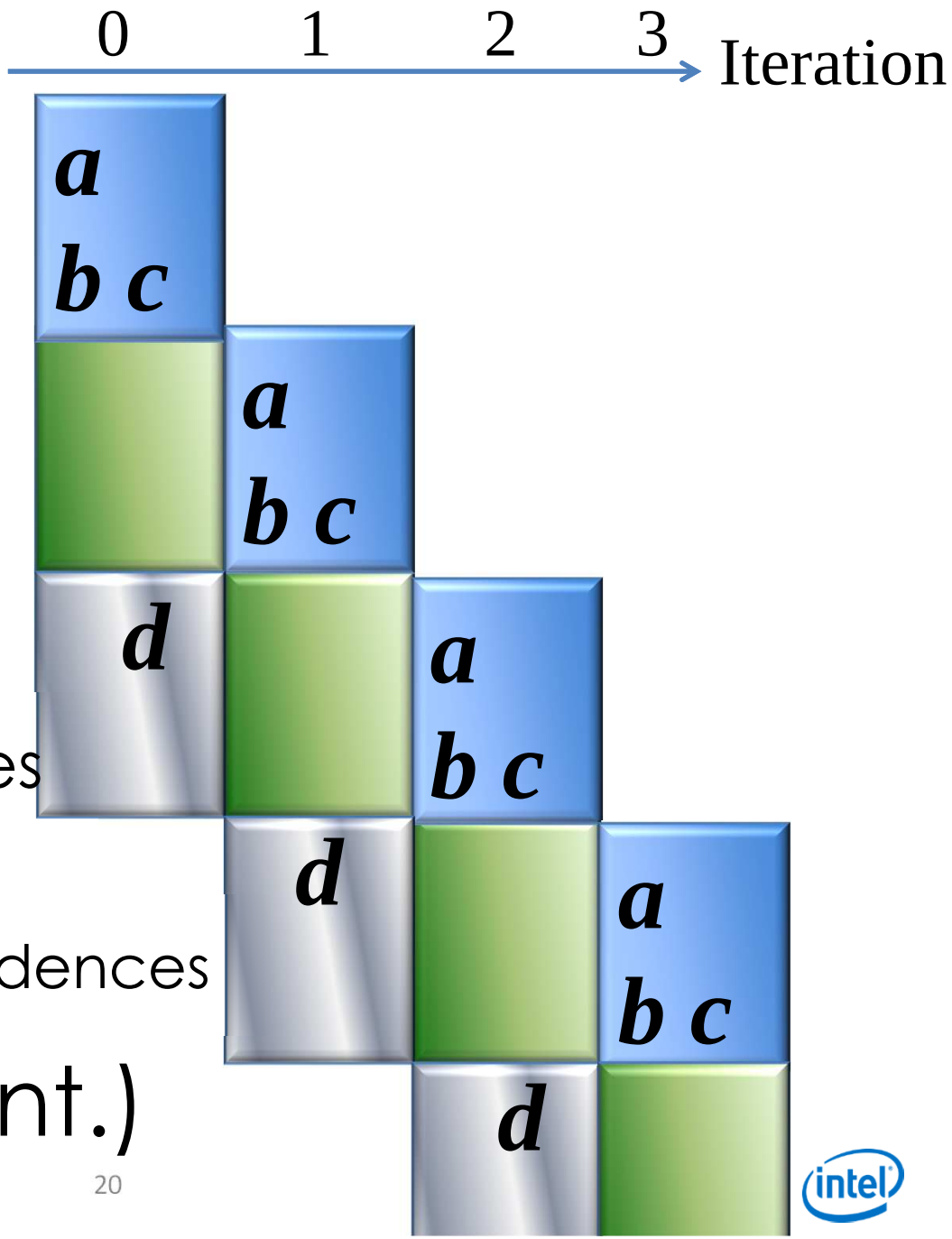




- [x] Local dependences
- [x] Resources
- [x] Loop-carried dependences

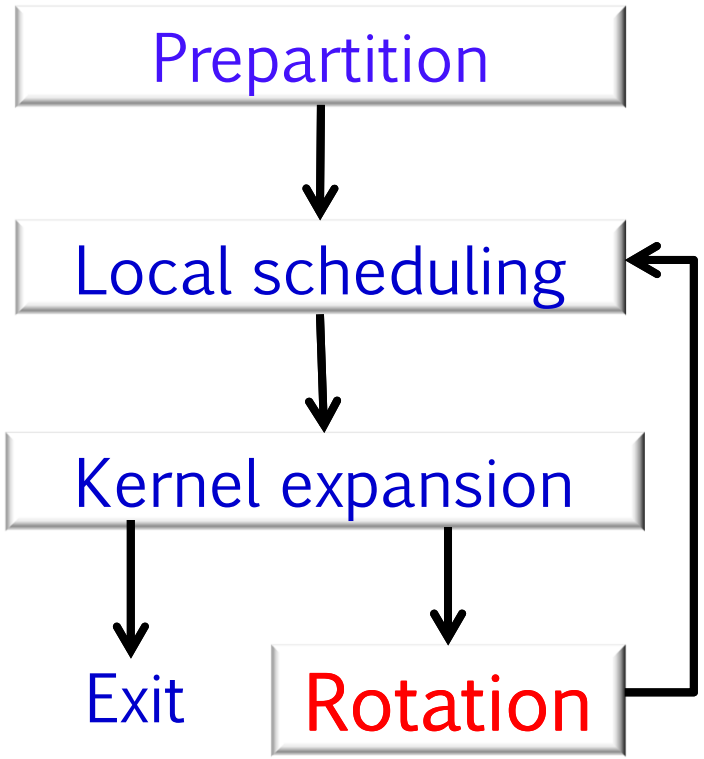
Illustration (Cont.)

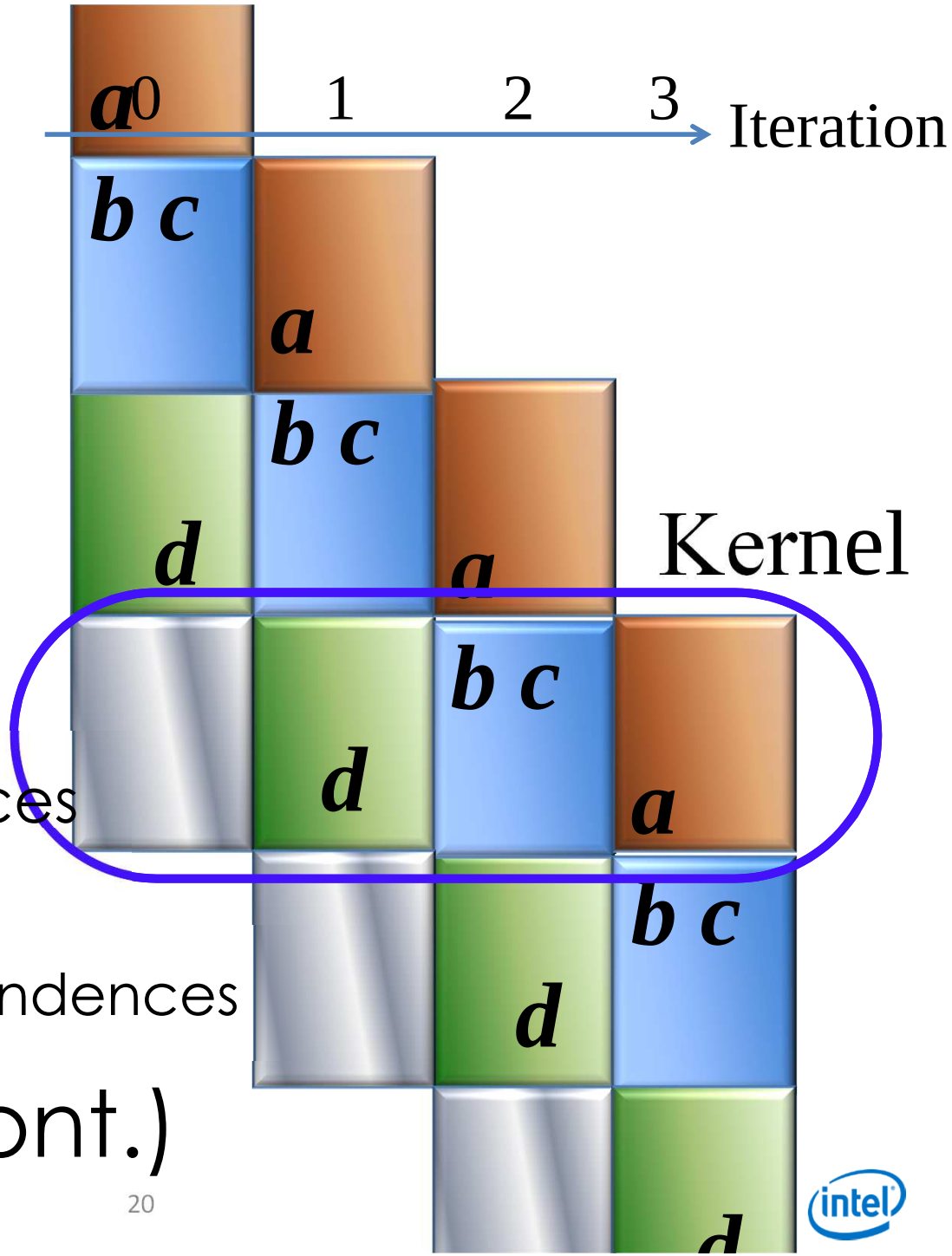




- [x] Local dependences
- [x] Resources
- [x] Loop-carried dependences

Illustration (Cont.)





- Local dependences
- Resources
- Loop-carried dependences

Illustration (Cont.)

Experiments

- Transmeta CMS on SPEC2k traces

Experiments

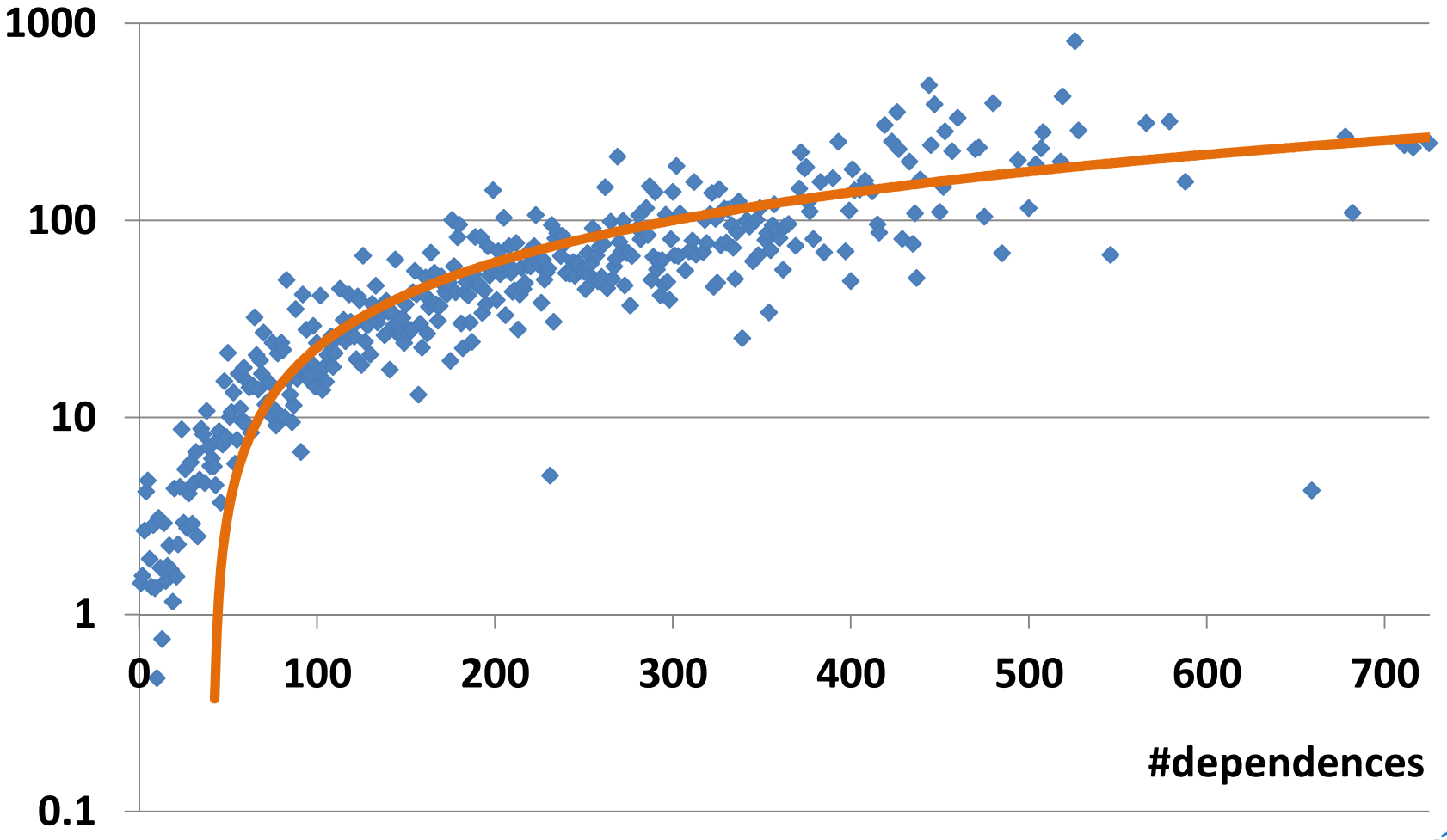
- Transmeta CMS on SPEC2k traces
- Functional simulator to comprehensively
  - Explore thresholds H and B
  - Evaluate compile overhead and schedules' quality

Experiments

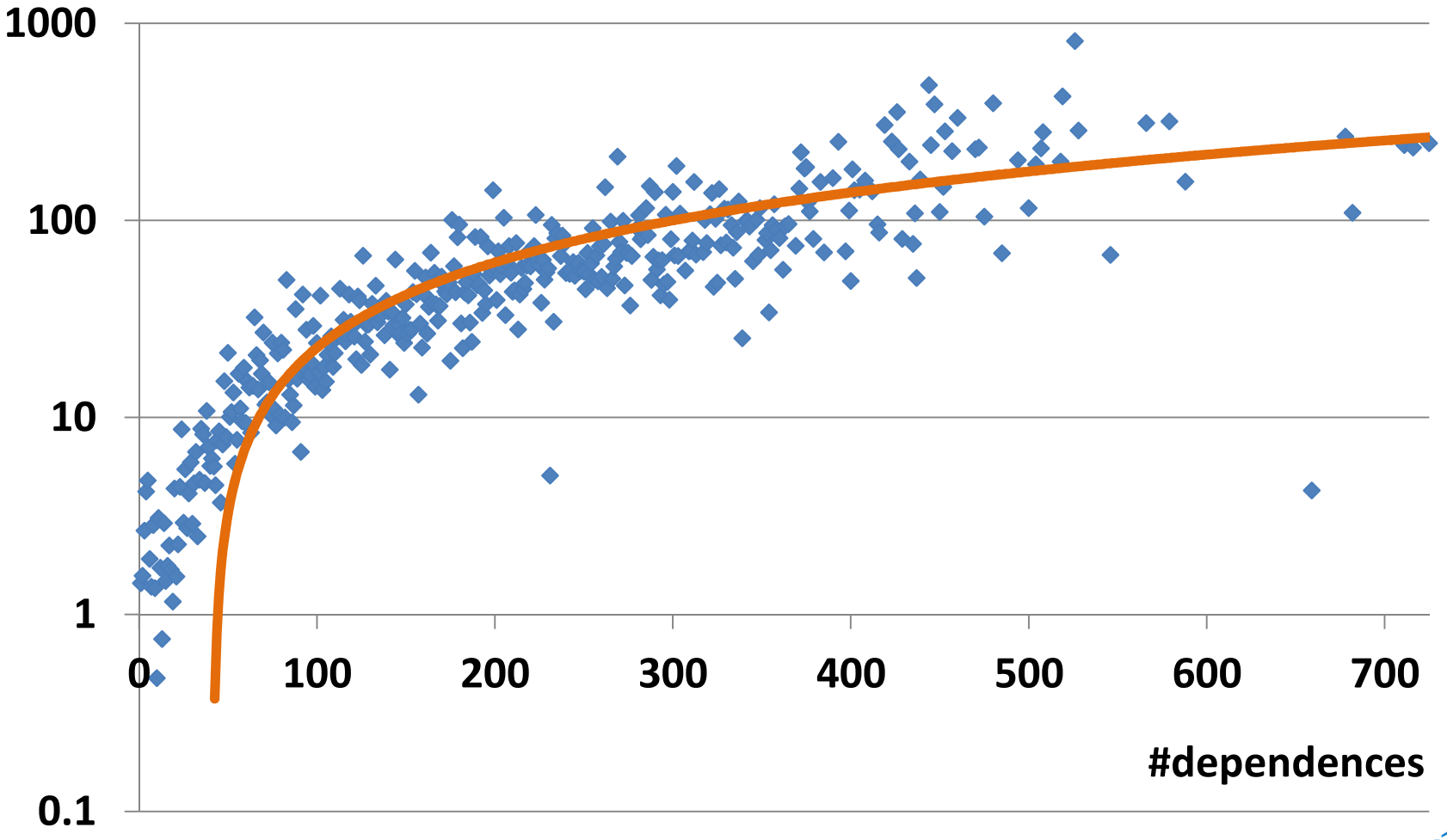
- Transmeta CMS on SPEC2k traces
- Functional simulator to comprehensively
  - Explore thresholds H and B
  - Evaluate compile overhead and schedules' quality
- Cycle-accurate simulator
  - Simulates cache misses, latencies, …
  - Initial performance study

Linearized Howard vs. exponential backoff + binary search [Rau et al. 1992]

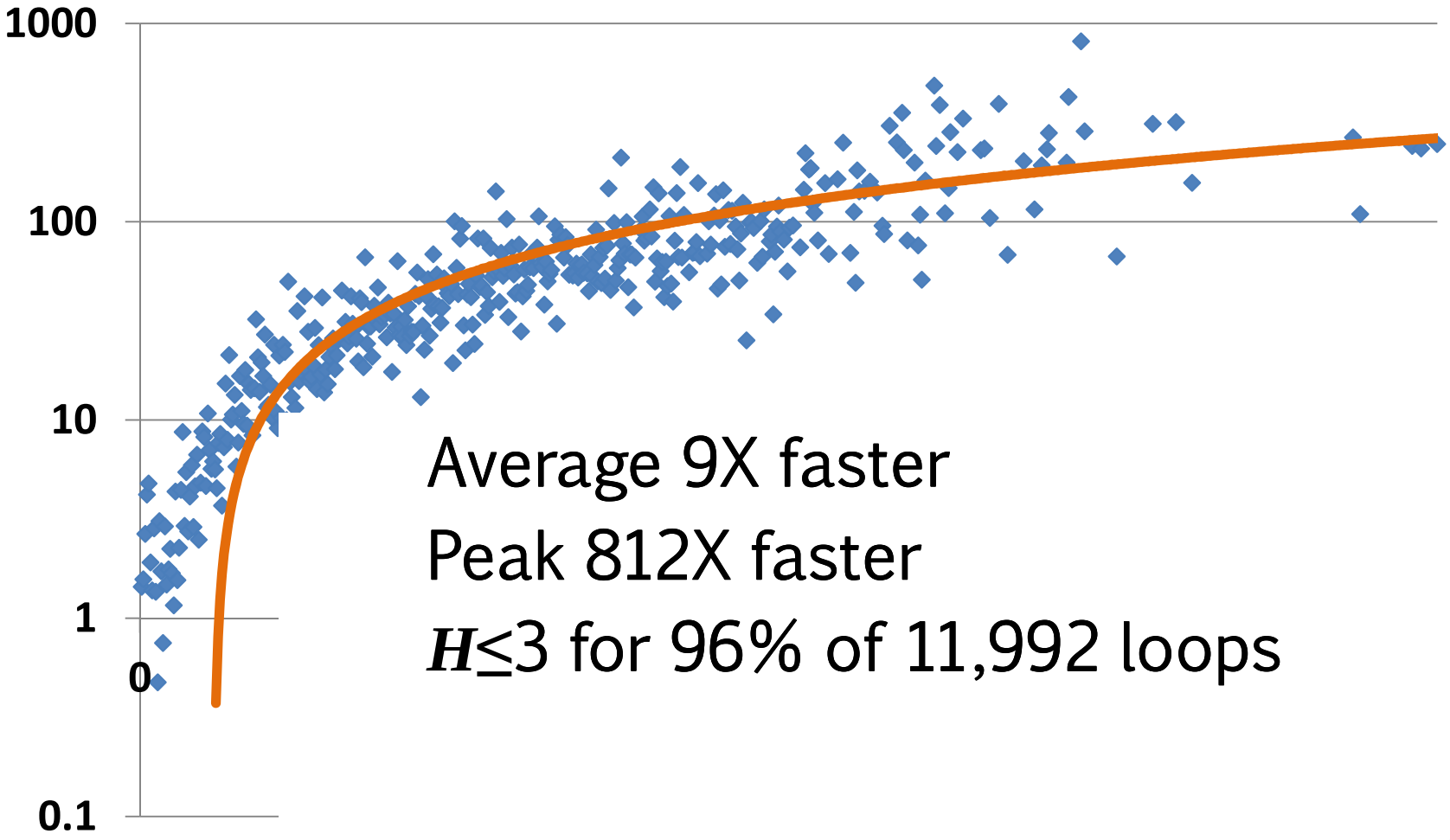
Linearized Howard vs. exponential backoff + binary search [Rau et al. 1992]



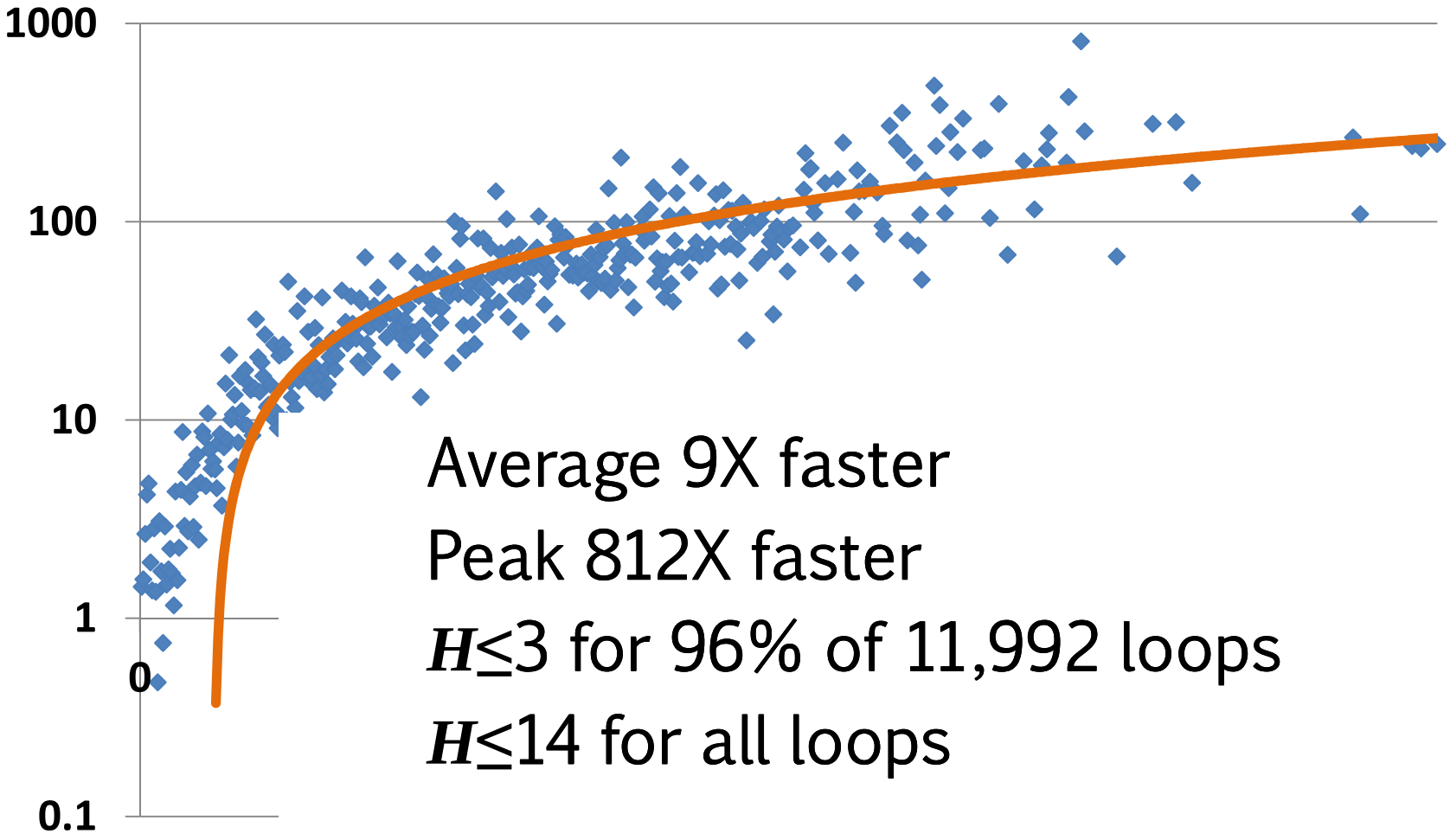
Linearized Howard vs. exponential backoff + binary search [Rau et al. 1992]



Linearized Howard vs. exponential backoff + binary search [Rau et al. 1992]



Linearized Howard vs. exponential backoff + binary search [Rau et al. 1992]



Average 9X faster

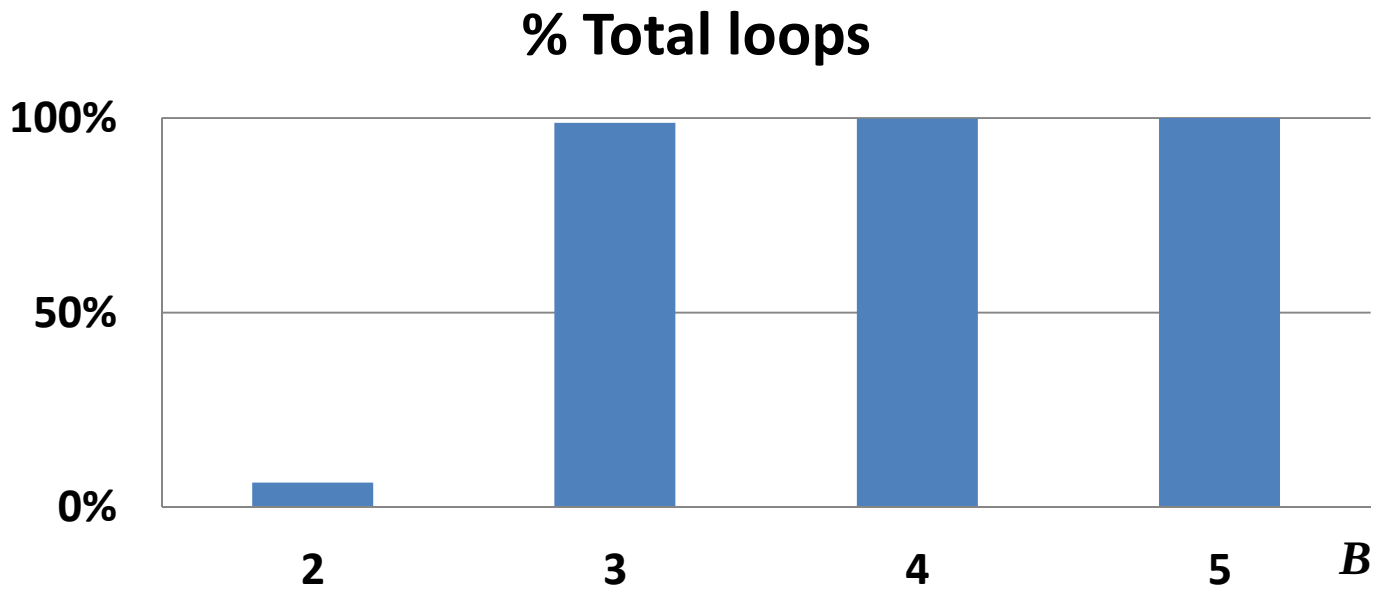
Peak 812X faster

$H \leq 3$ for 96% of 11,992 loops

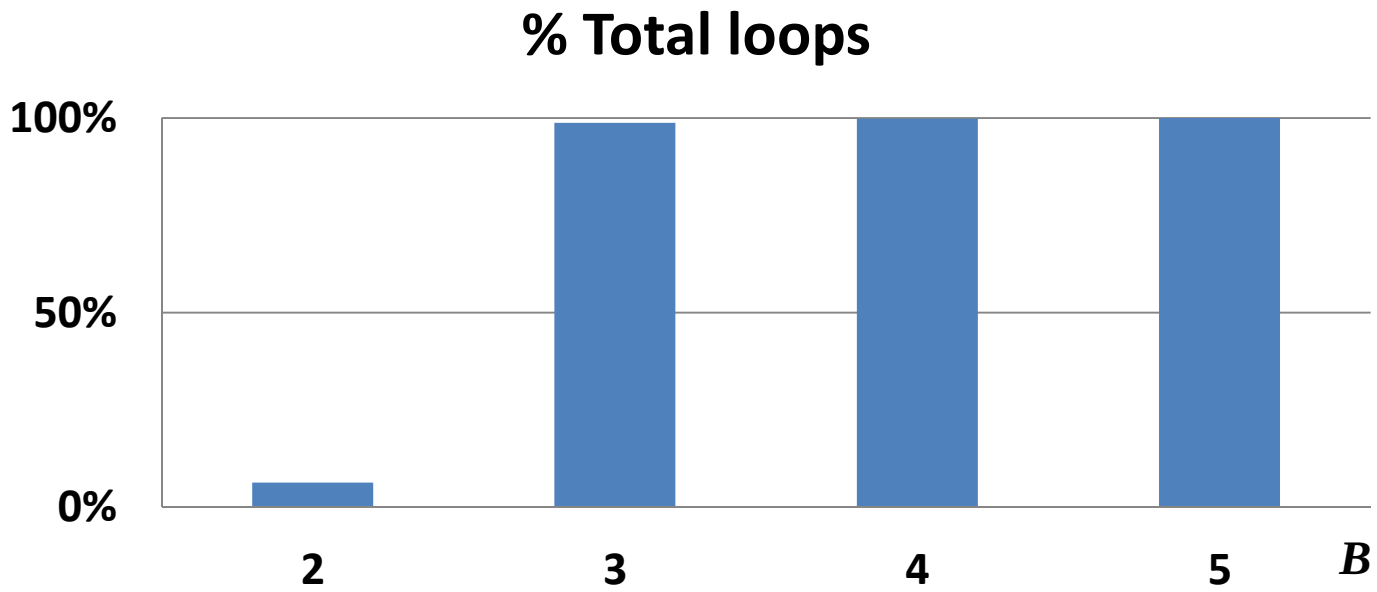
$H \leq 14$ for all loops

Linearized Bellman-Ford

Linearized Bellman-Ford

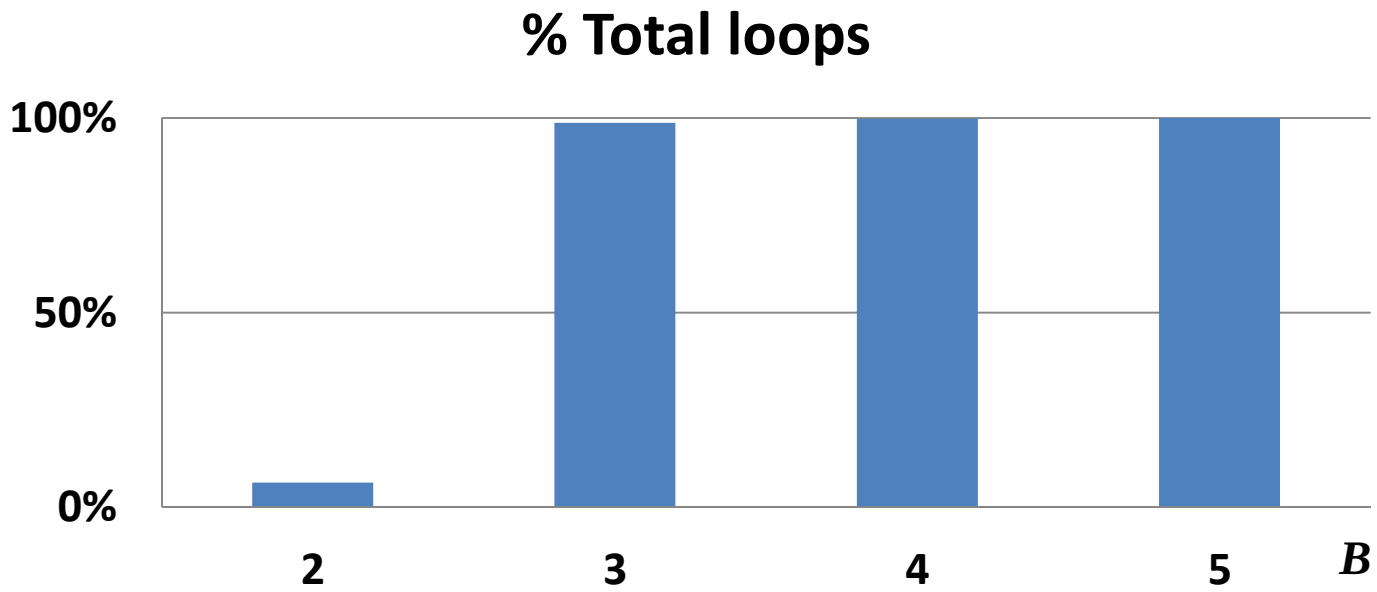


Linearized Bellman-Ford



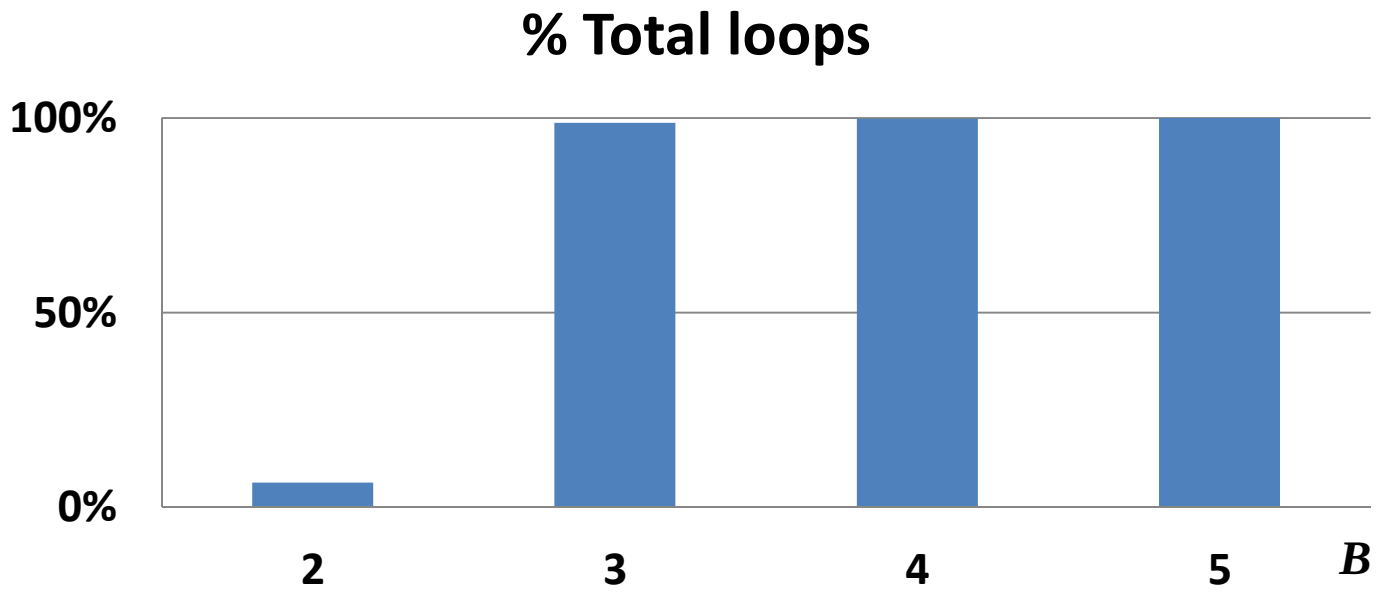
- $B \leq 3$ for 98.8% of 11,992 loops

Linearized Bellman-Ford



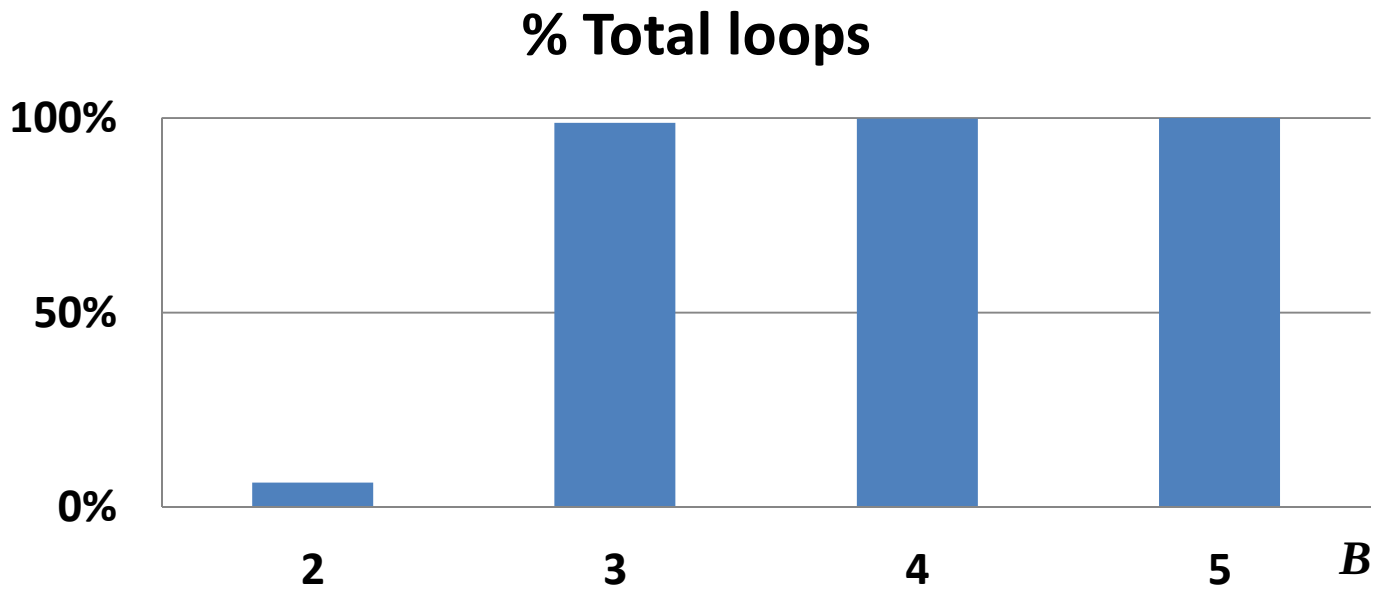
- $B \leq 3$ for 98.8% of 11,992 loops

Linearized Bellman-Ford



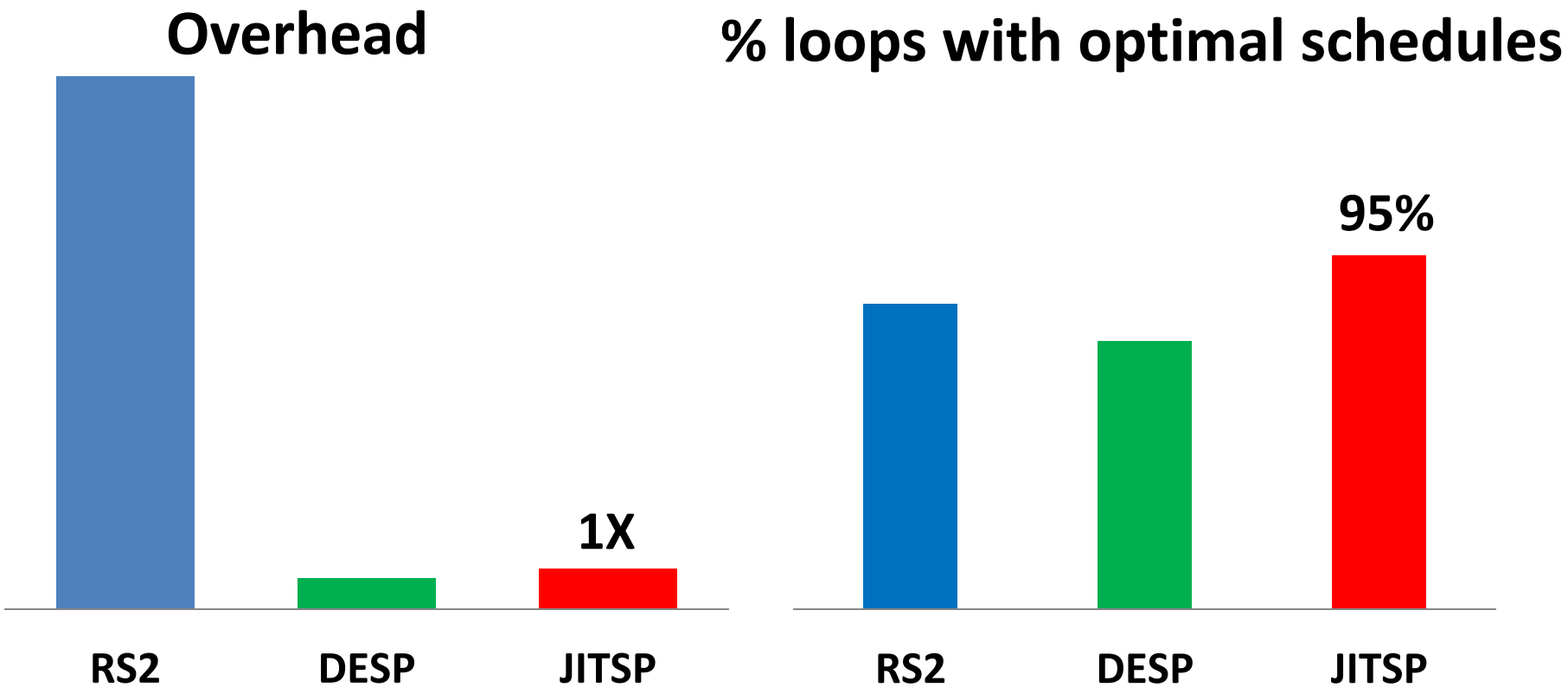
- $B \leq 3$ for 98.8% of 11,992 loops
- $B \leq 5$ for all the loops

Linearized Bellman-Ford

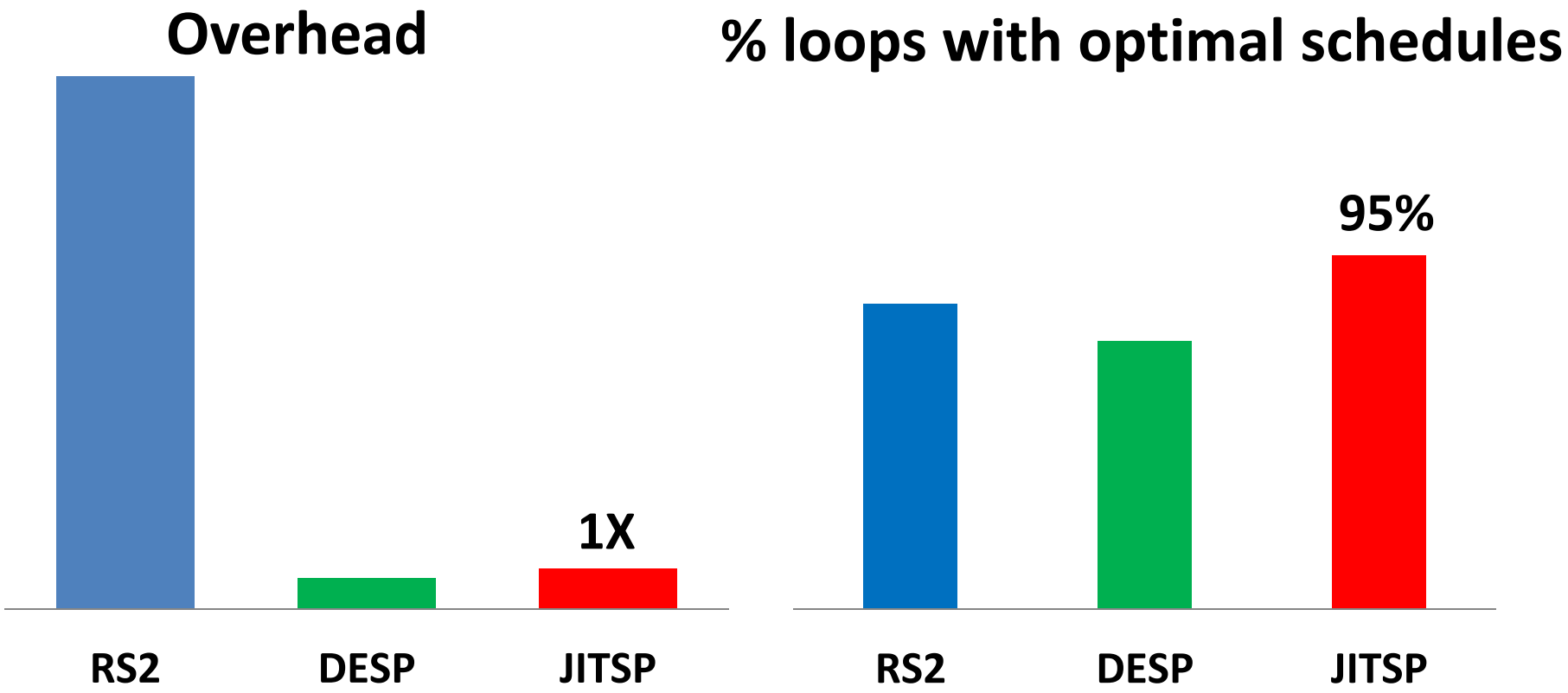


- $B \leq 3$ for 98.8% of 11,992 loops
- $B \leq 5$ for all the loops
- From now on, we set H=10, B=5 ➔ 11,910 loops scheduled

Scheduling overhead & schedules' quality

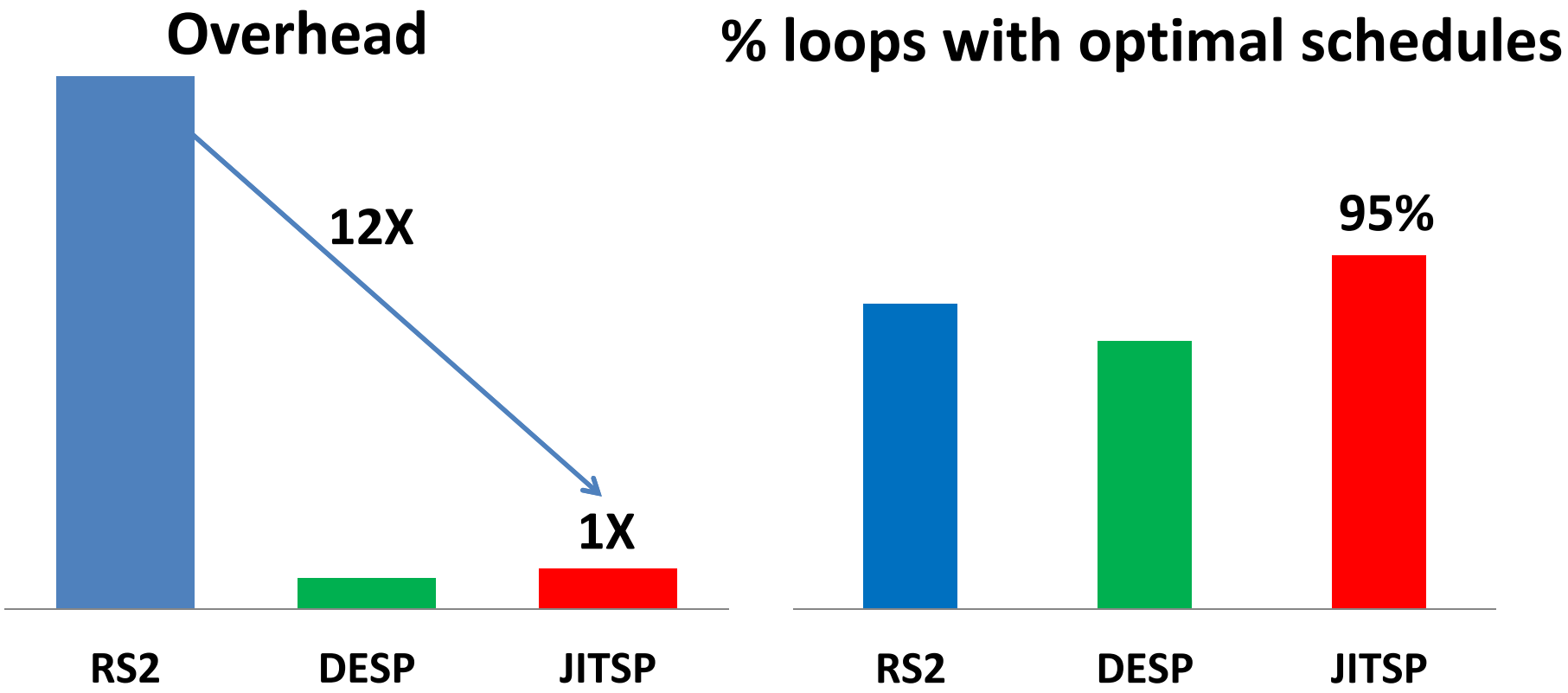


Scheduling overhead & schedules' quality



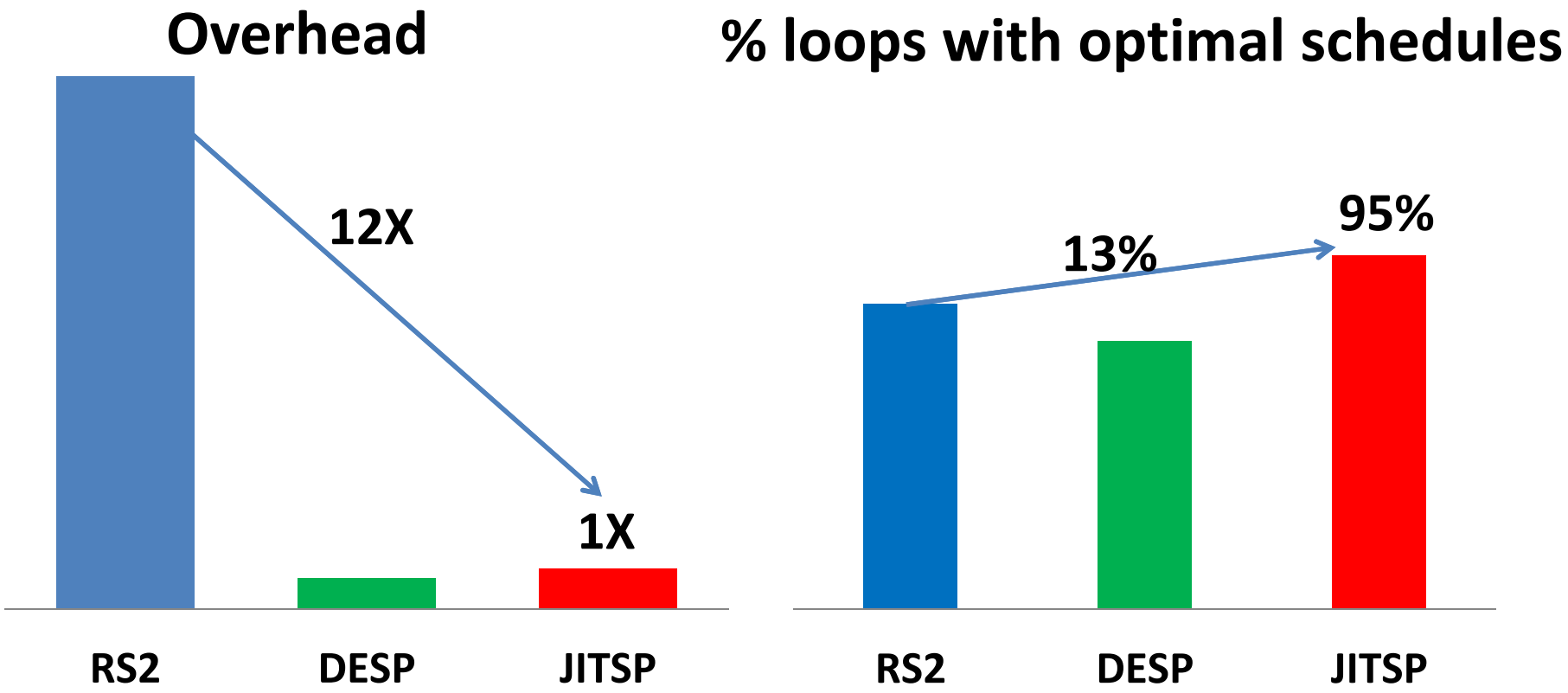
- JITSP achieves optimal schedules for 95% loops

Scheduling overhead & schedules' quality



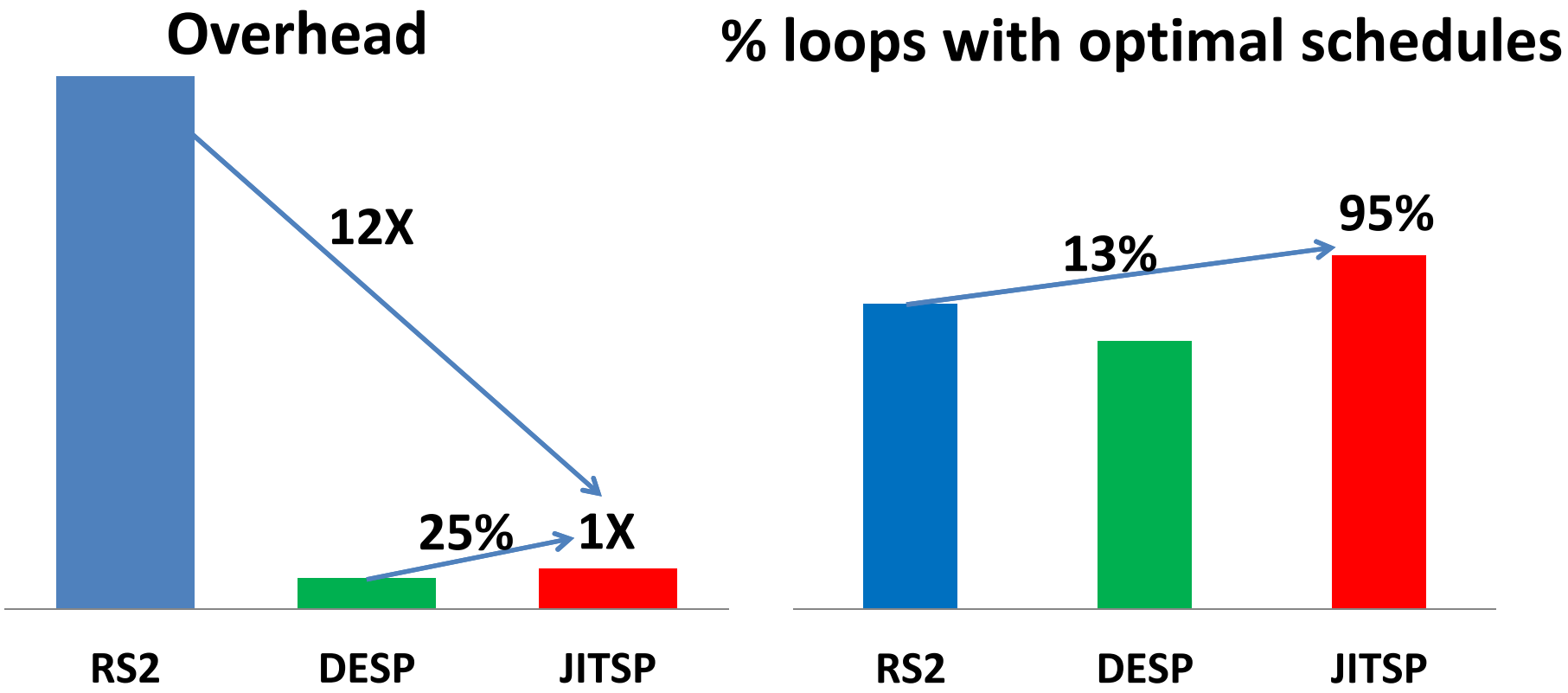
- JITSP achieves optimal schedules for 95% loops

Scheduling overhead & schedules' quality



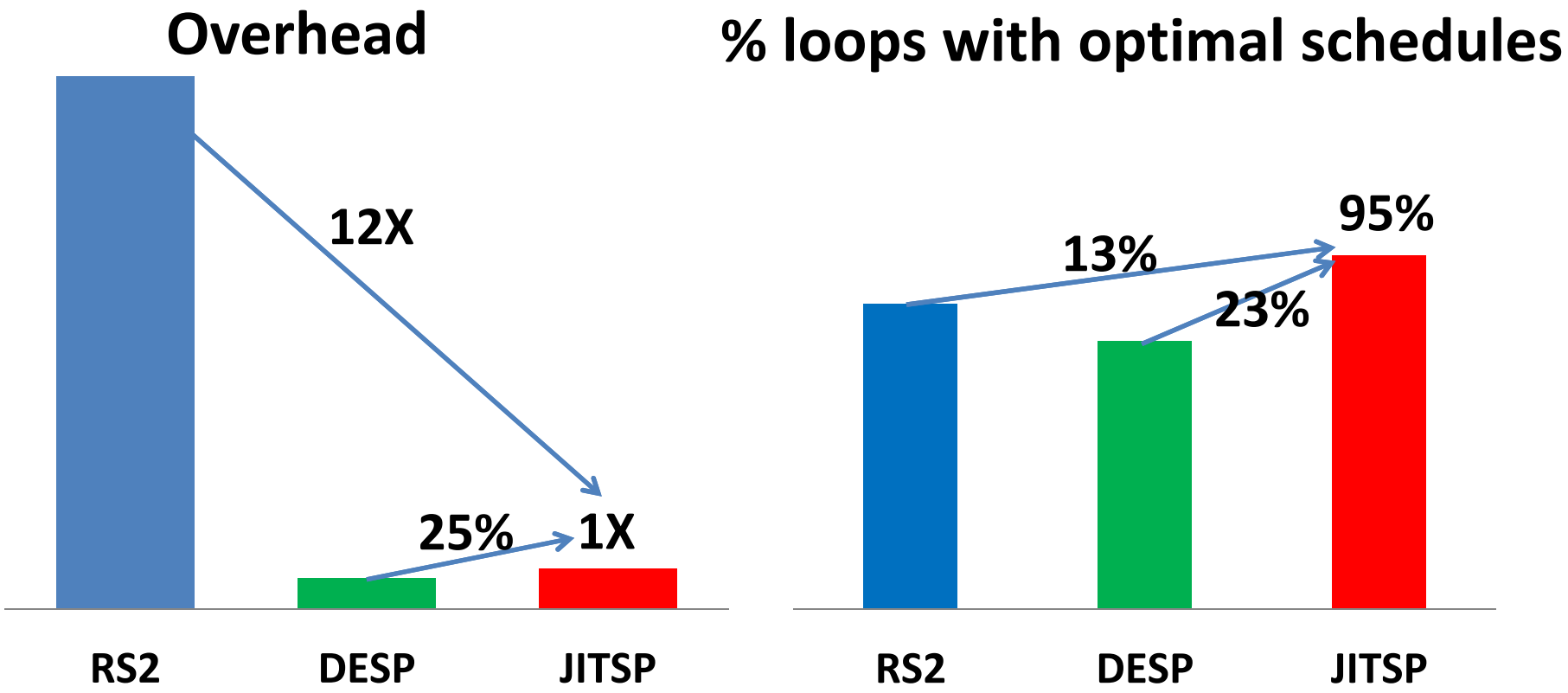
- JITSP achieves optimal schedules for 95% loops

Scheduling overhead & schedules' quality



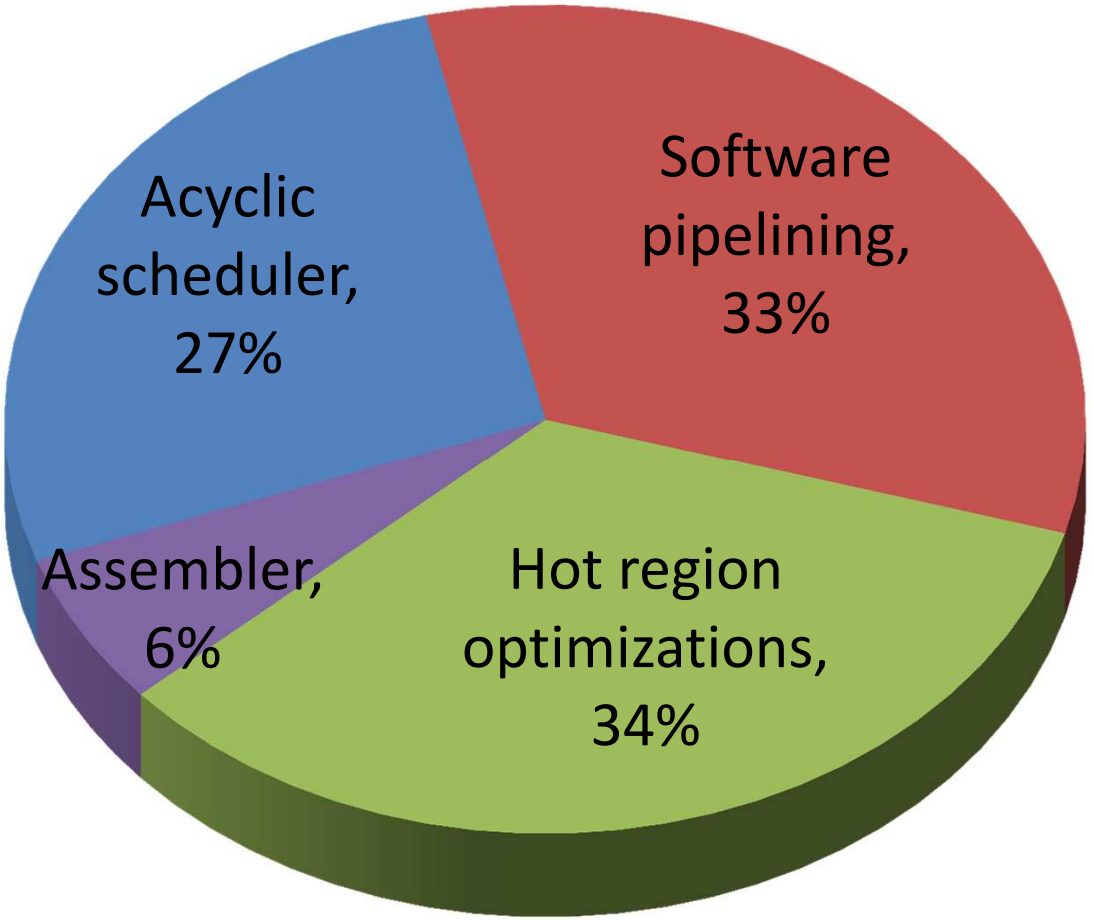
- JITSP achieves optimal schedules for 95% loops

Scheduling overhead & schedules' quality



- JITSP achieves optimal schedules for 95% loops

Compile overhead distribution

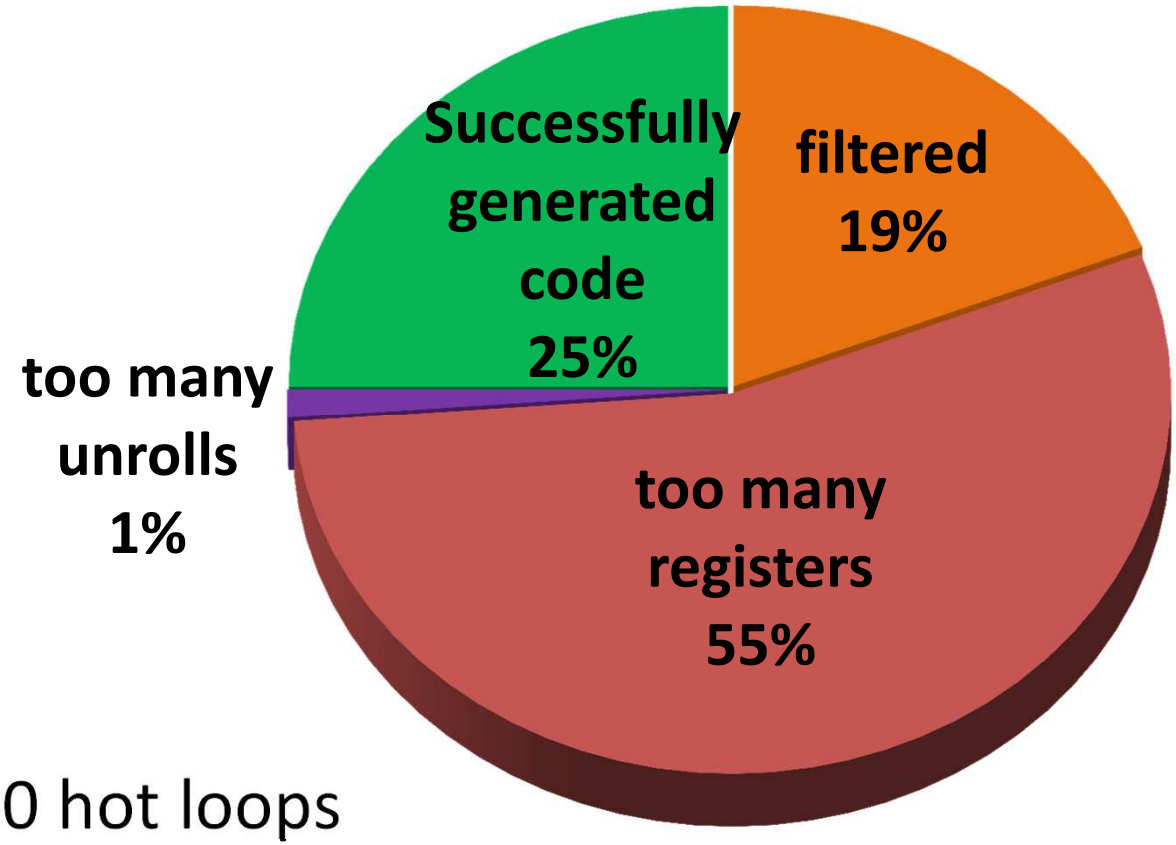


Note: acyclic scheduler handles acyclic code, or loops NOT selected for software pipelining

Preliminary performance

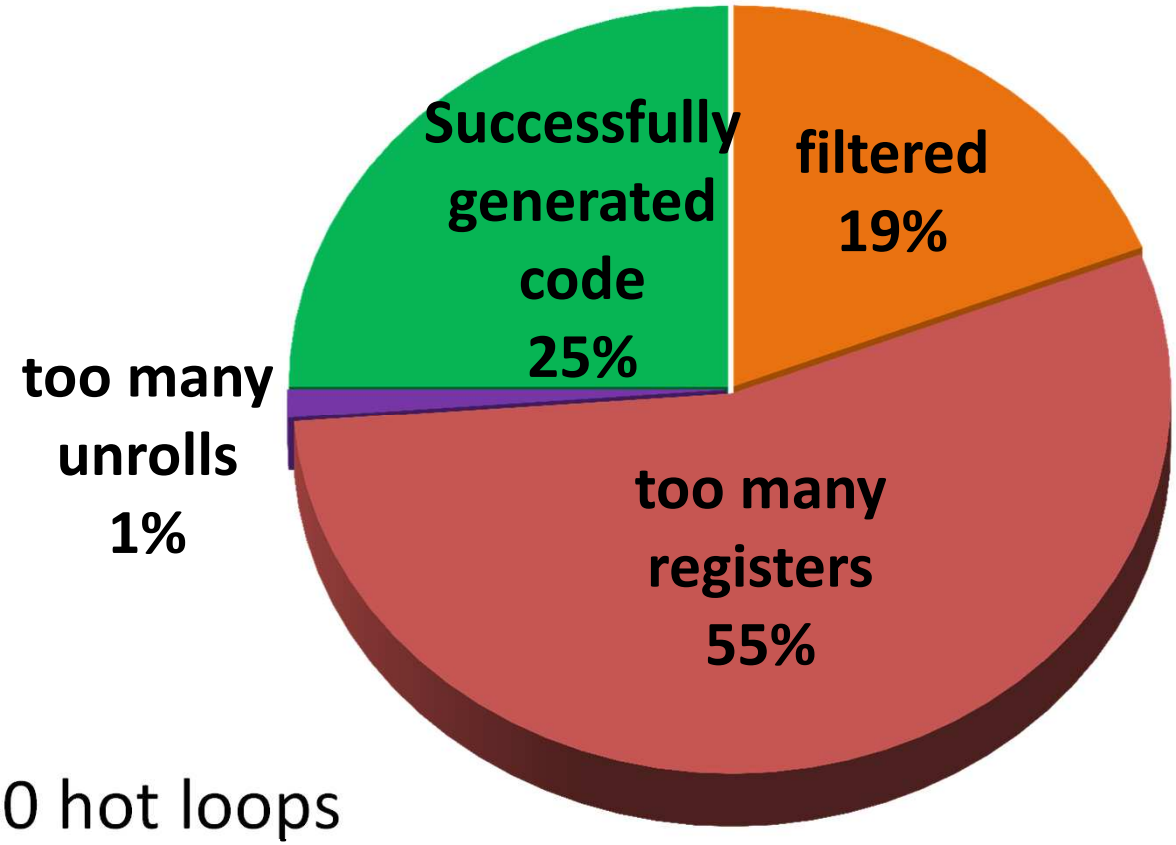
- 40 hot loops

Preliminary performance



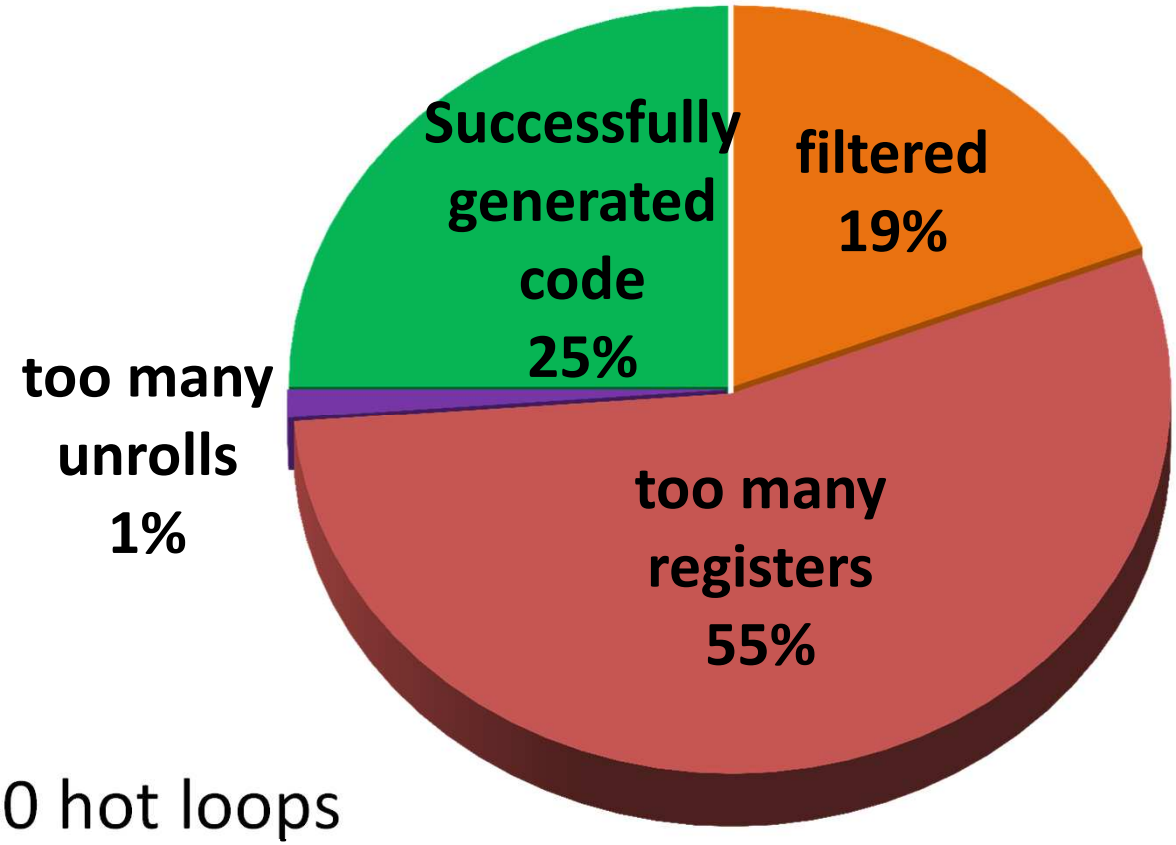
- 40 hot loops

Preliminary performance



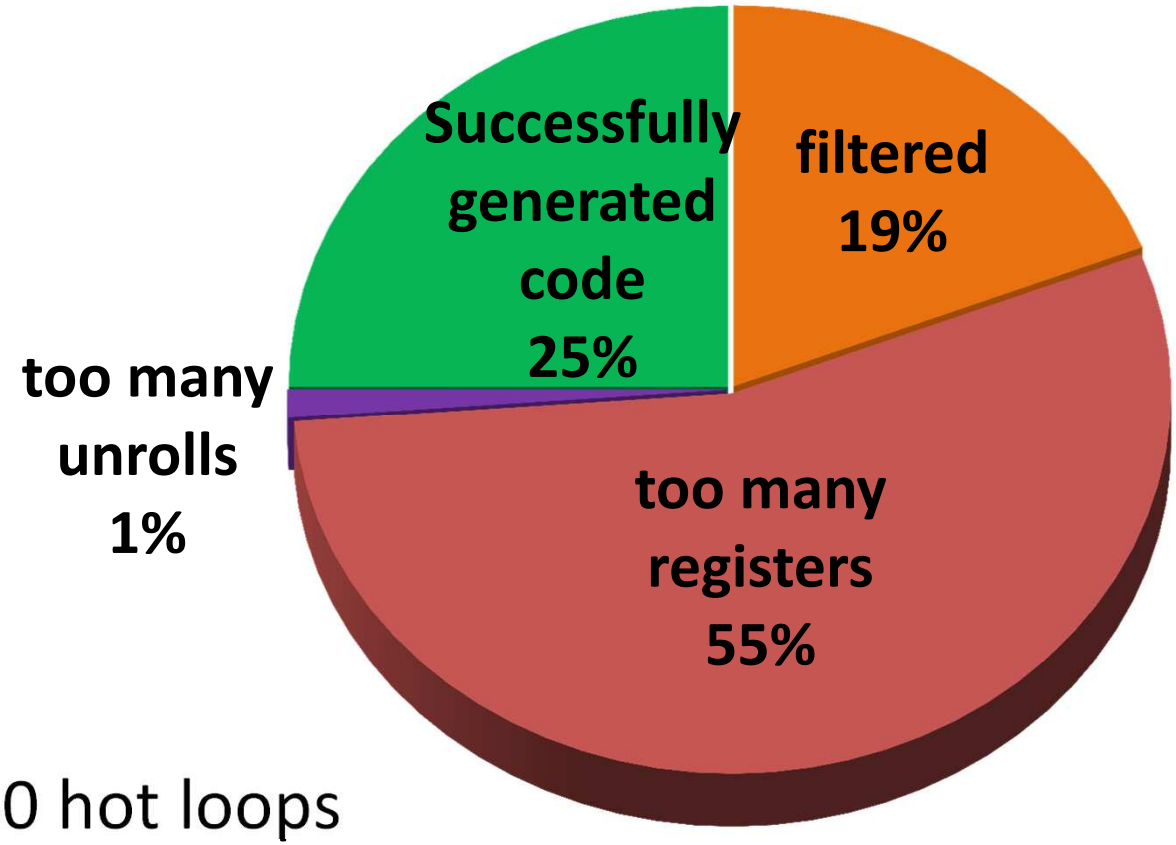
- 40 hot loops

Preliminary performance



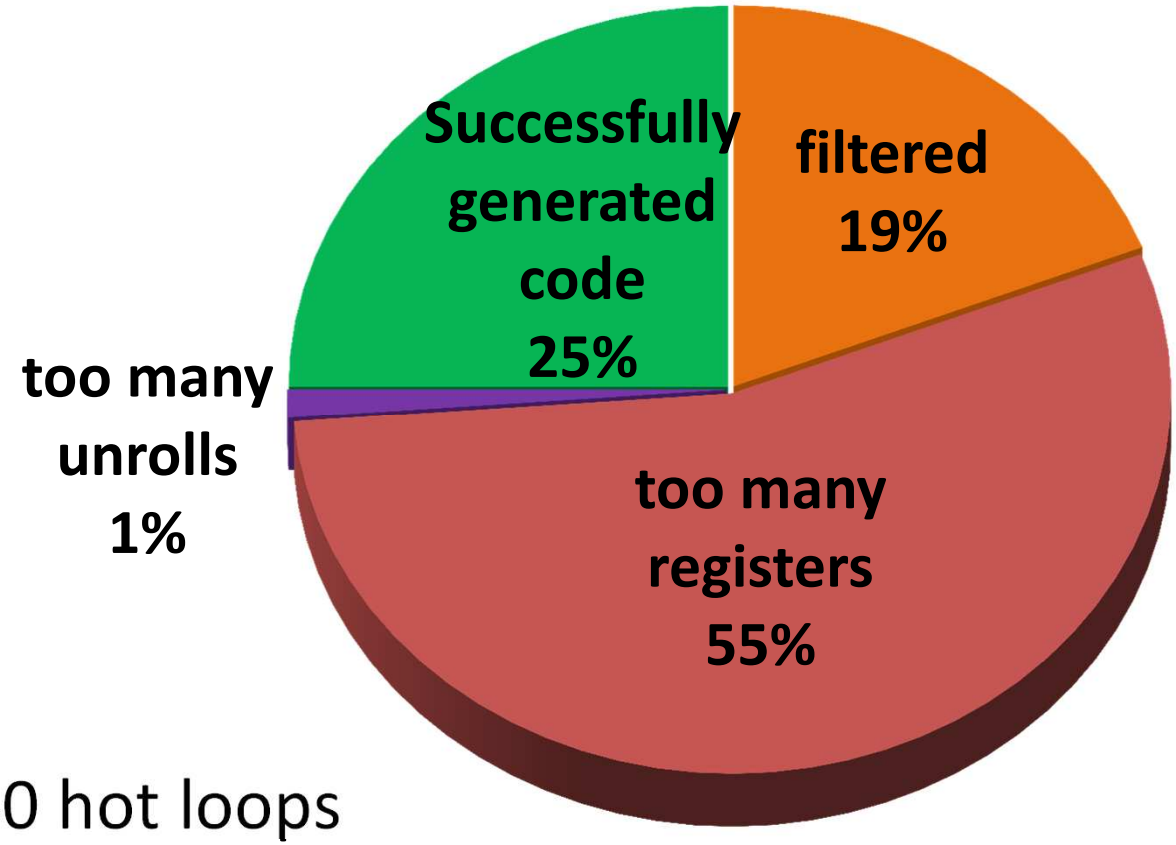
- 40 hot loops
- The architecture has bottleneck in registers
  - 2/7/24/32 predicate/static alias/integer/floating point available for pipelining

Preliminary performance



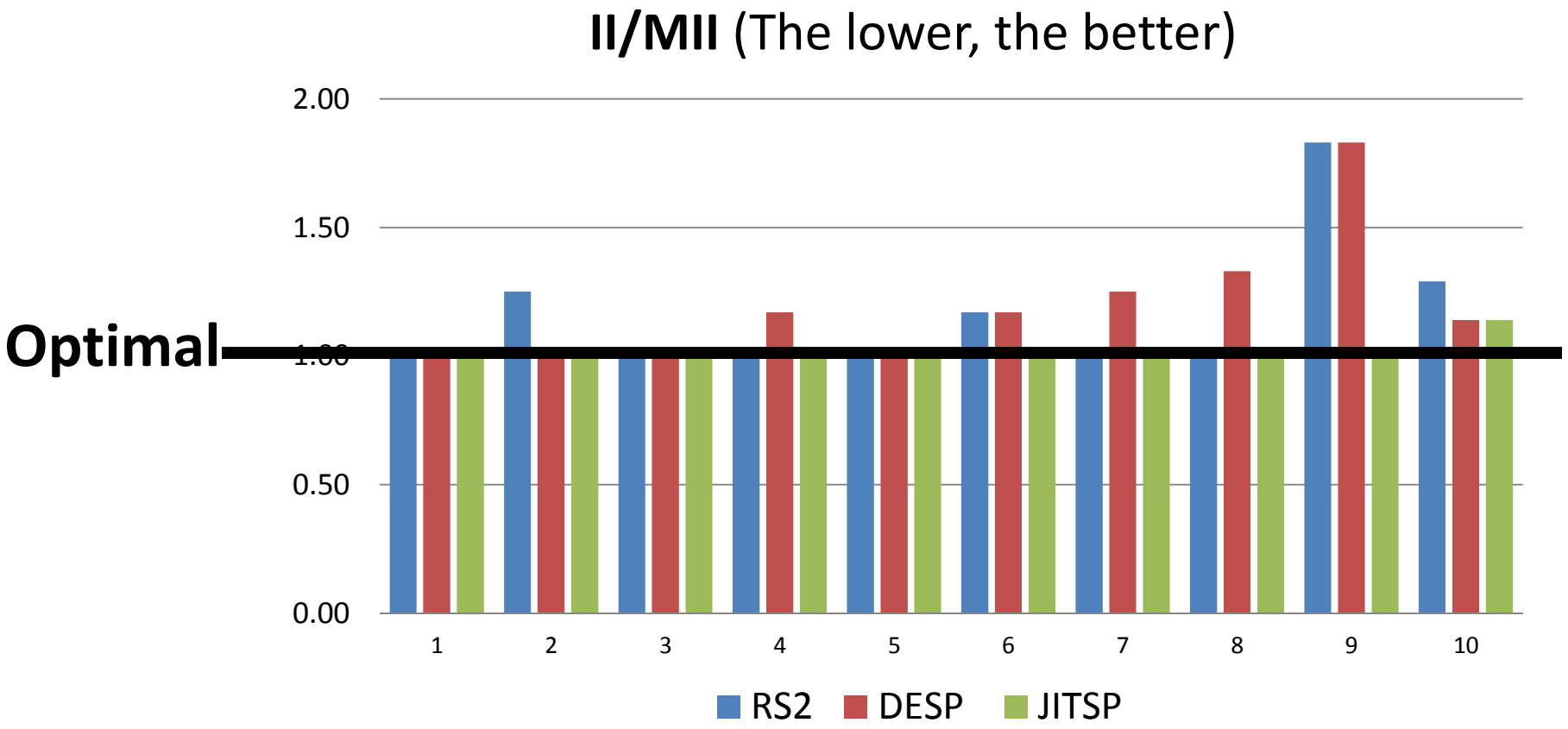
- 40 hot loops
- The architecture has bottleneck in registers
  - 2/7/24/32 predicate/static alias/integer/floating point available for pipelining

Preliminary performance

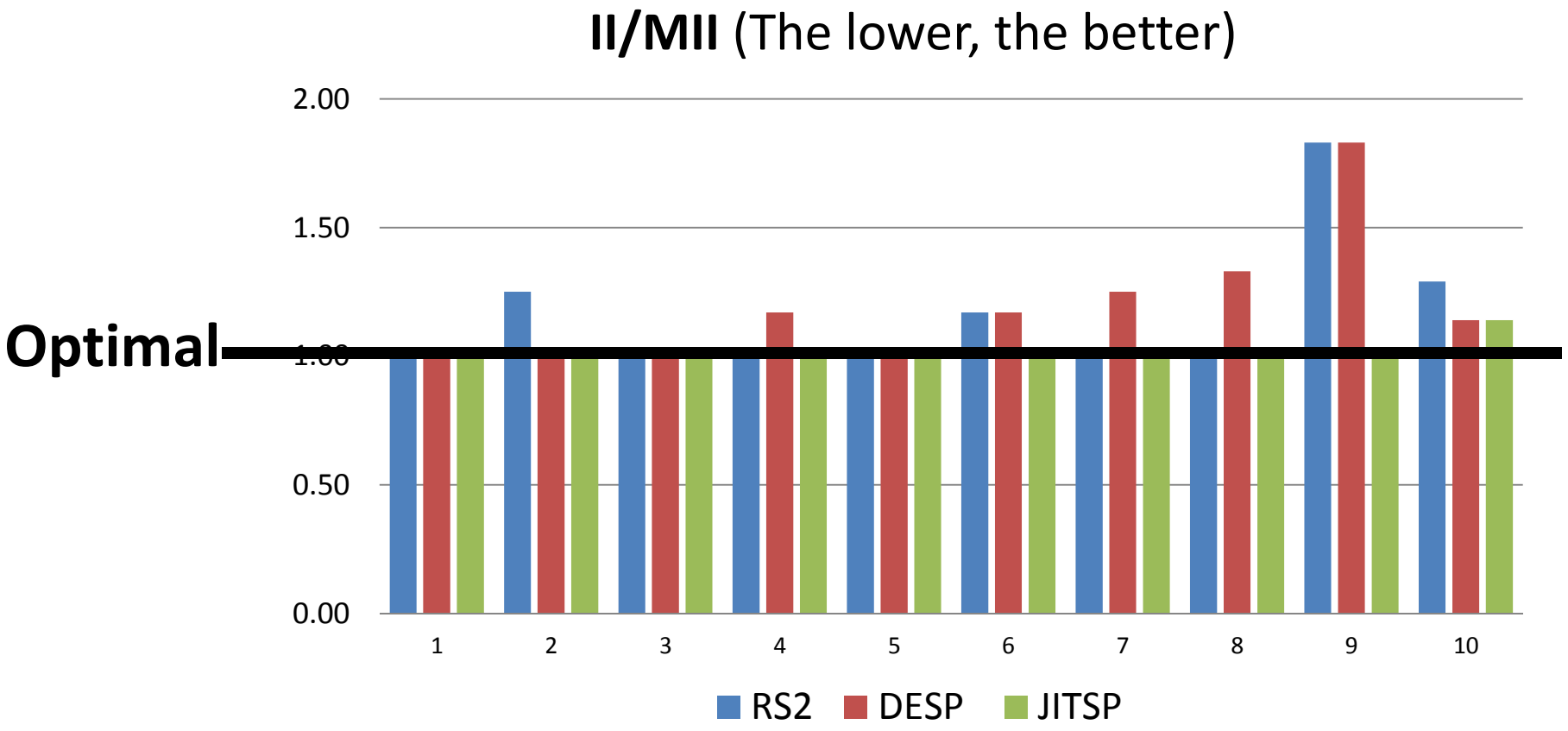


- 40 hot loops
- The architecture has bottleneck in registers
  - 2/7/24/32 predicate/static alias/integer/floating point available for pipelining

Preliminary performance (Cont.)

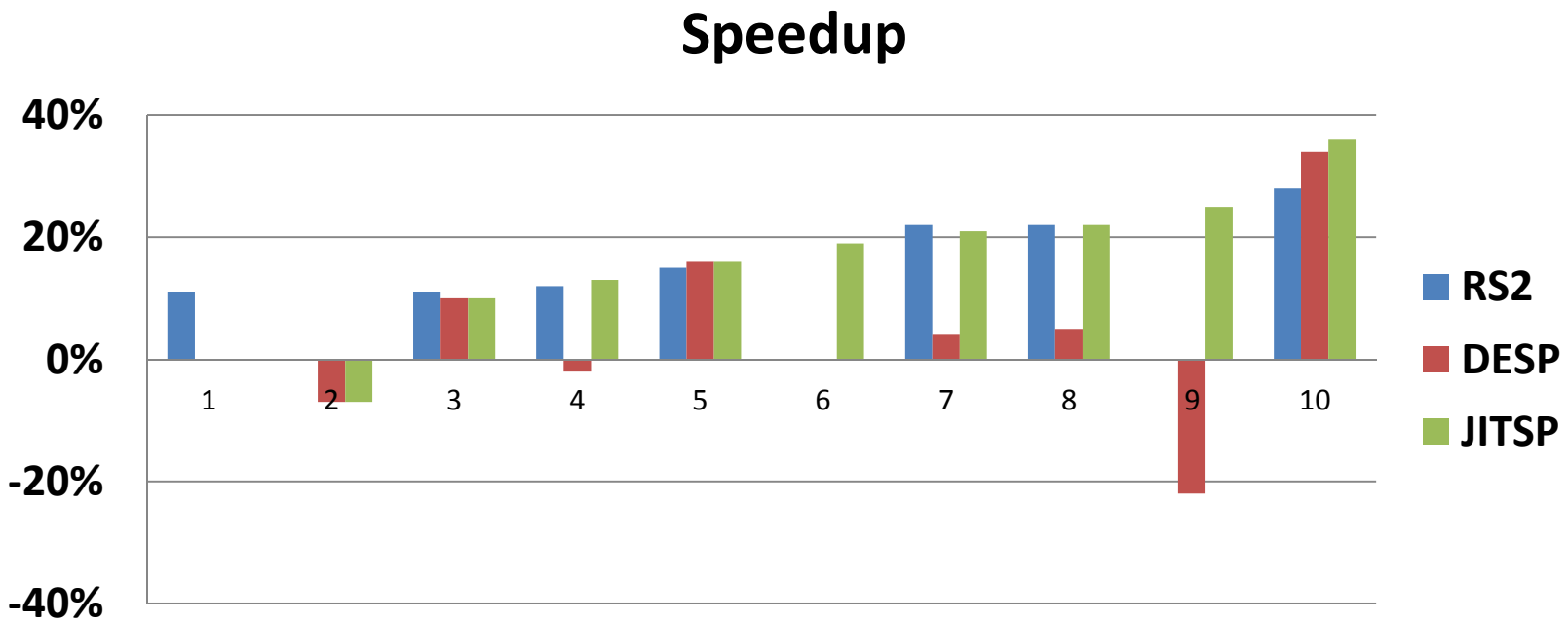


Preliminary performance (Cont.)



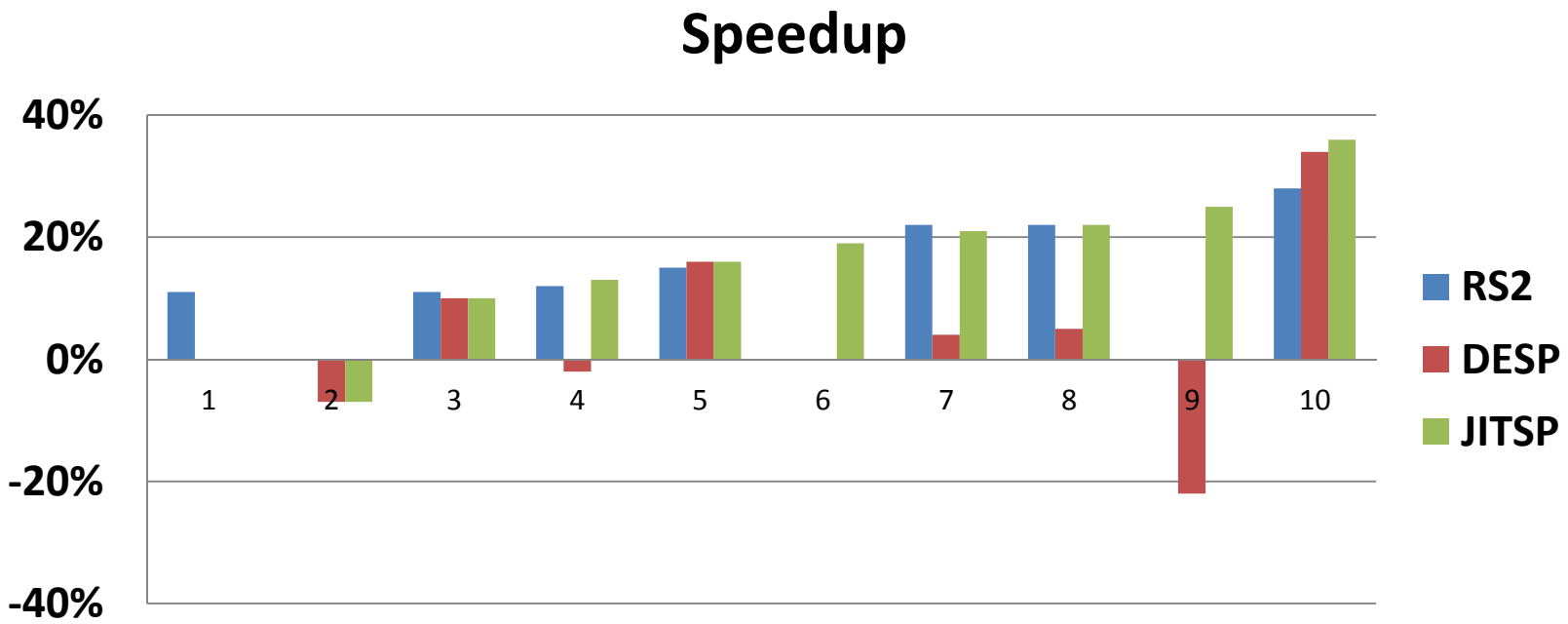
- JITSP achieves optimal schedules for all but 1 loops

Preliminary performance (Cont.)



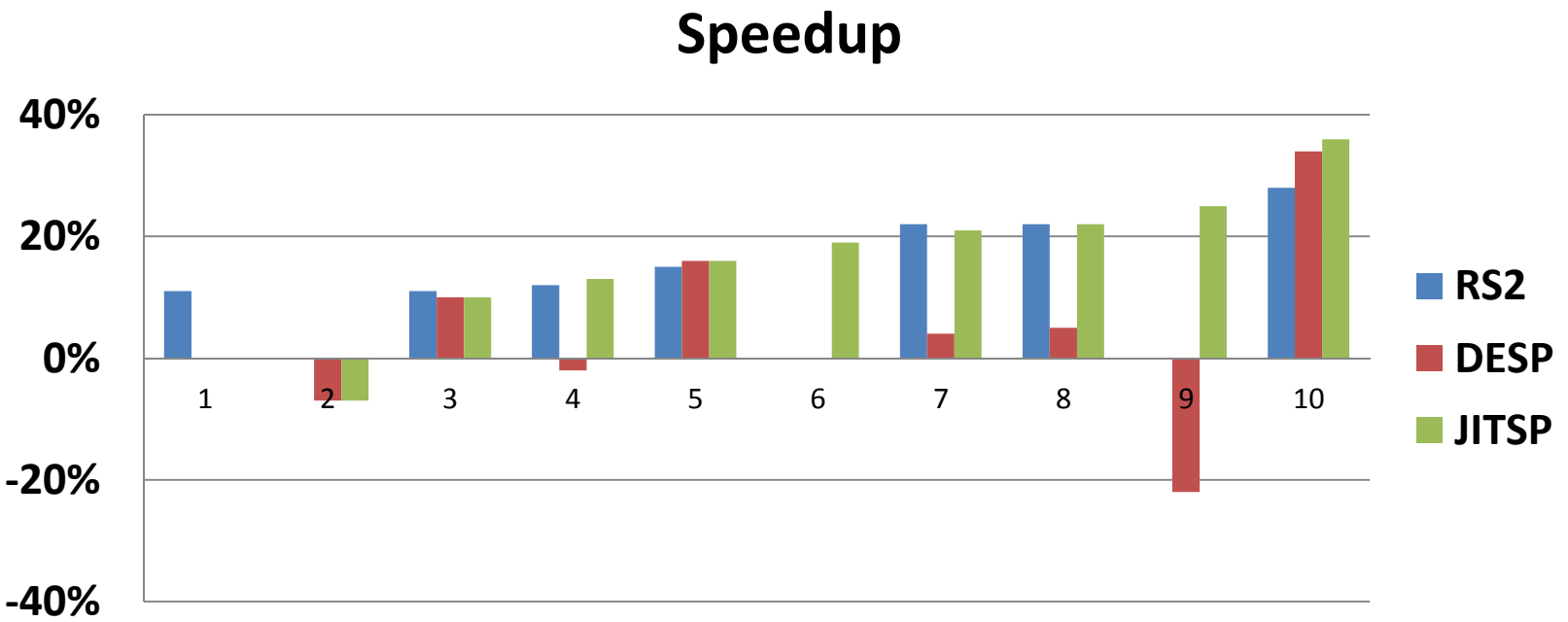
- JITSP 10~36% speedup. Better than the others

Preliminary performance (Cont.)



- JITSP 10~36% speedup. Better than the others
- Exception: loop 2. Optimal schedule but slowdown due to memory aliases ➔ retranslation needed

Preliminary performance (Cont.)



- JITSP 10~36% speedup. Better than the others
- Exception: loop 2. Optimal schedule but slowdown due to memory aliases ➔ retranslation needed
- Speedup swim(5.3%), ammp(4.4%), mcf(3.6%)

Conclusion

Conclusion

- 1st linear software pipelining algorithm implemented for dynamic compilers

Conclusion

- 1st linear software pipelining algorithm implemented for dynamic compilers
- Turns a traditionally-expensive optimization into linear time O(V+E)

Conclusion

- 1st linear software pipelining algorithm implemented for dynamic compilers
- Turns a traditionally-expensive optimization into linear time O(V+E)
  - Taking advantages of results from various domains:
    - hardware circuit design (Retiming)
    - stochastic control (Howard algorithm)
    - Graph (Bellman-Ford)
    - software pipelining (Rotation scheduling and DESP)

Conclusion

- 1st linear software pipelining algorithm implemented for dynamic compilers
- Turns a traditionally-expensive optimization into linear time O(V+E)
  - Taking advantages of results from various domains:
    - hardware circuit design (Retiming)
    - stochastic control (Howard algorithm)
    - Graph (Bellman-Ford)
    - software pipelining (Rotation scheduling and DESP)
- Generates optimal or near-optimal schedules with reasonable compile overhead

Future work

- Register availability
  - Add more architecture registers
  - Algorithm: Register pressure-aware
- Implementation
  - Loop selection
  - Re-translation
- Evaluation
  - Benchmarks

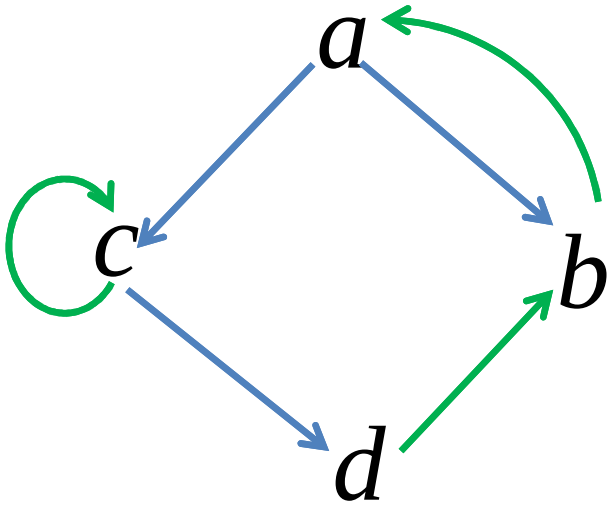
Backup slides

Howard algorithm

- Each operation is rewarded to reach a cycle via 1 policy edge
  - The bigger the cycle, the more the reward

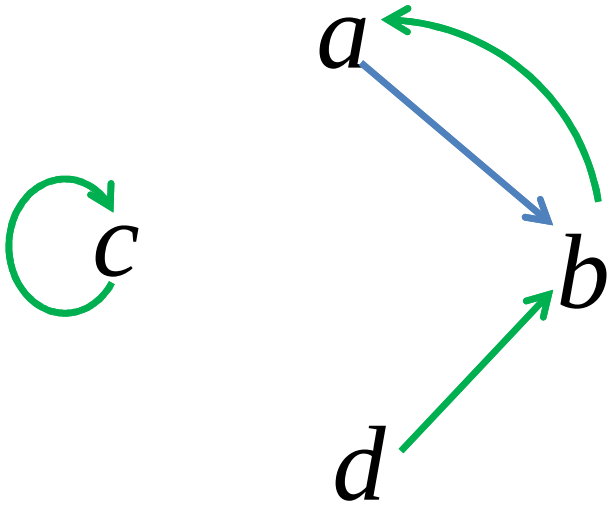
Howard algorithm

- Each operation is rewarded to reach a cycle via 1 policy edge
  - The bigger the cycle, the more the reward



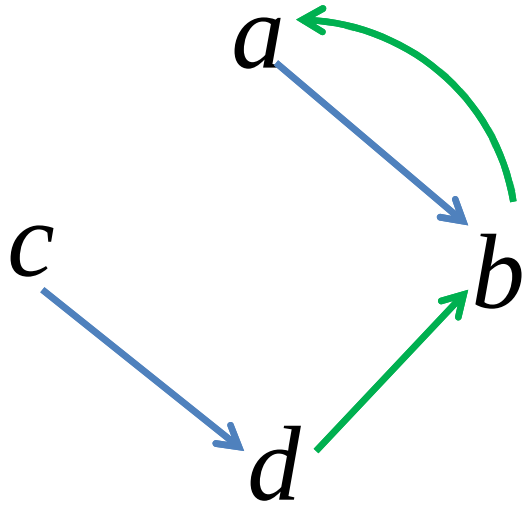
Howard algorithm

- Each operation is rewarded to reach a cycle via 1 policy edge
  - The bigger the cycle, the more the reward



Howard algorithm

- Each operation is rewarded to reach a cycle via 1 policy edge
  - The bigger the cycle, the more the reward



Distribution statistics of JITSP

	min	median	mean	max
# operations	4	10	13.82	96
# dependences	12	38	50.48	353
# local dependences	1	13	22.36	283
# loop-carried deps	5	22	28.12	156
MII	1	3	4.78	55
II - MII	0	0	0.05	5
II / MII	1	1	1.01	1.5
# local scheduling	1	1	1.26	4