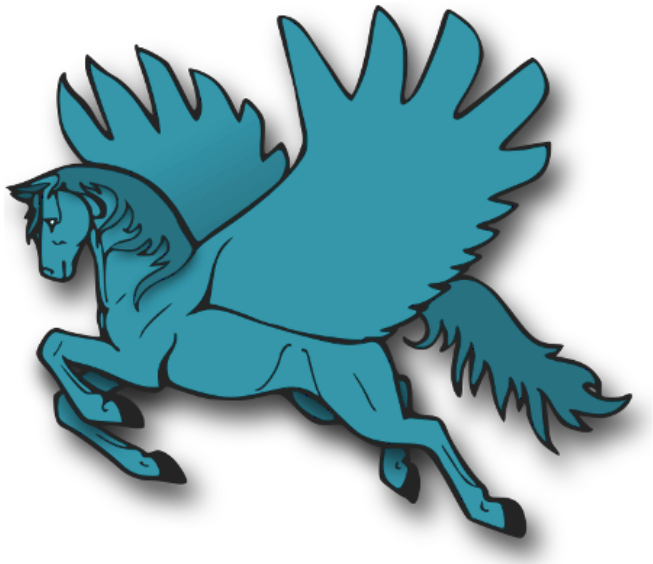
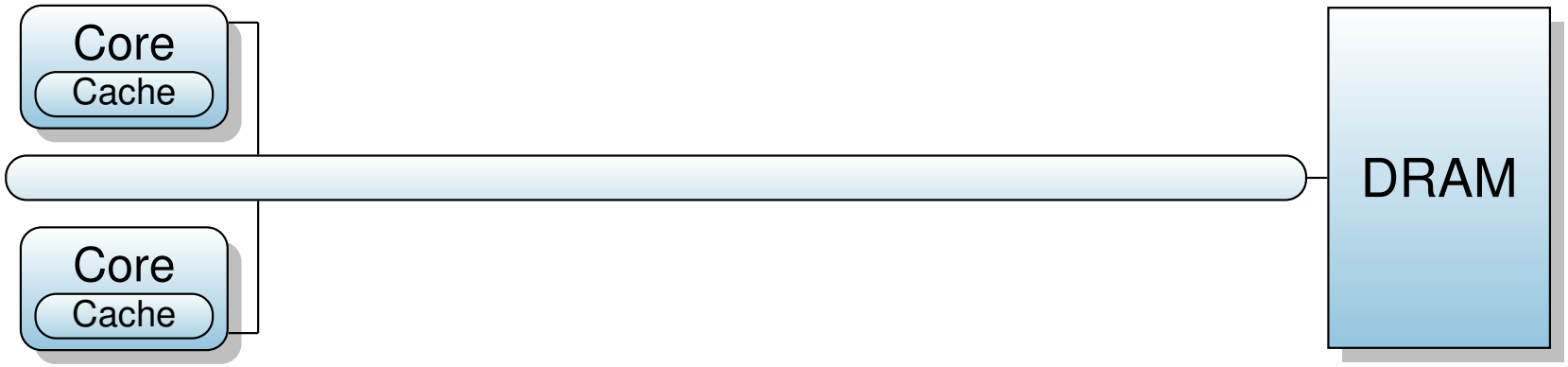
Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores

Manuel Mohr, Carsten Tradowsky

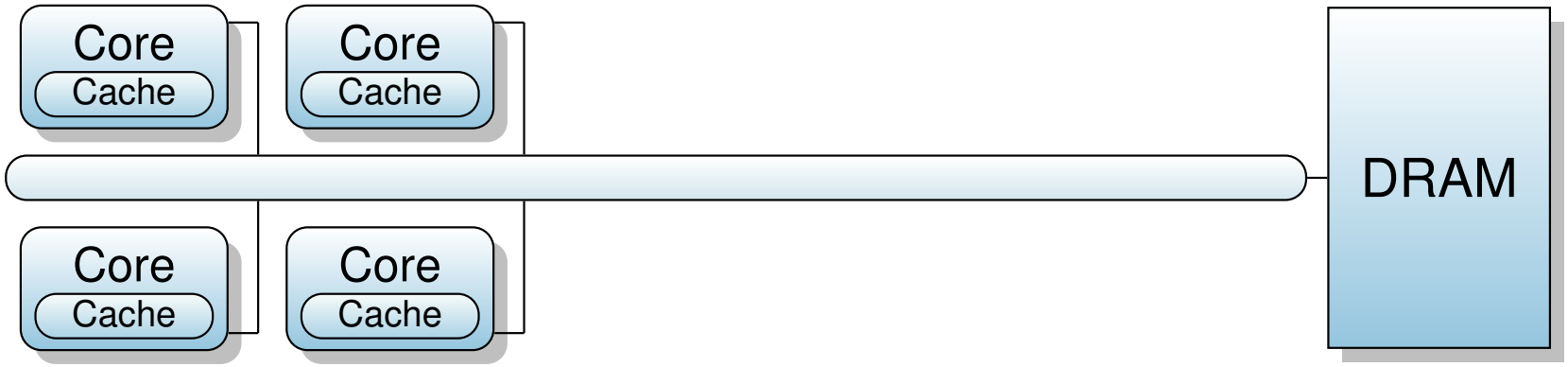
Chair for Programming Paradigms & Institute for Information Processing Technologies, Karlsruhe Institute of Technology (KIT)



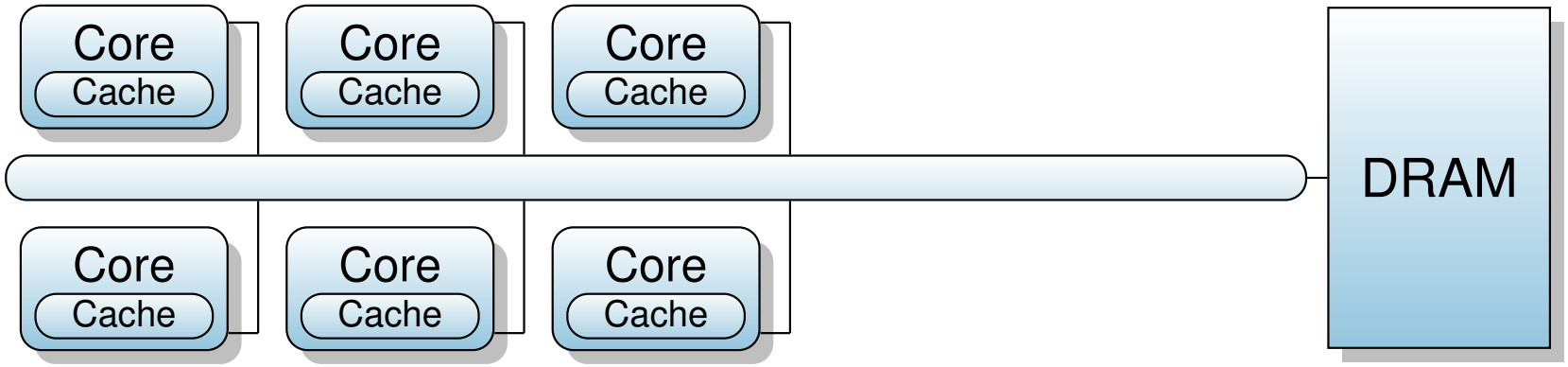
Non-Cache-Coherent Architectures



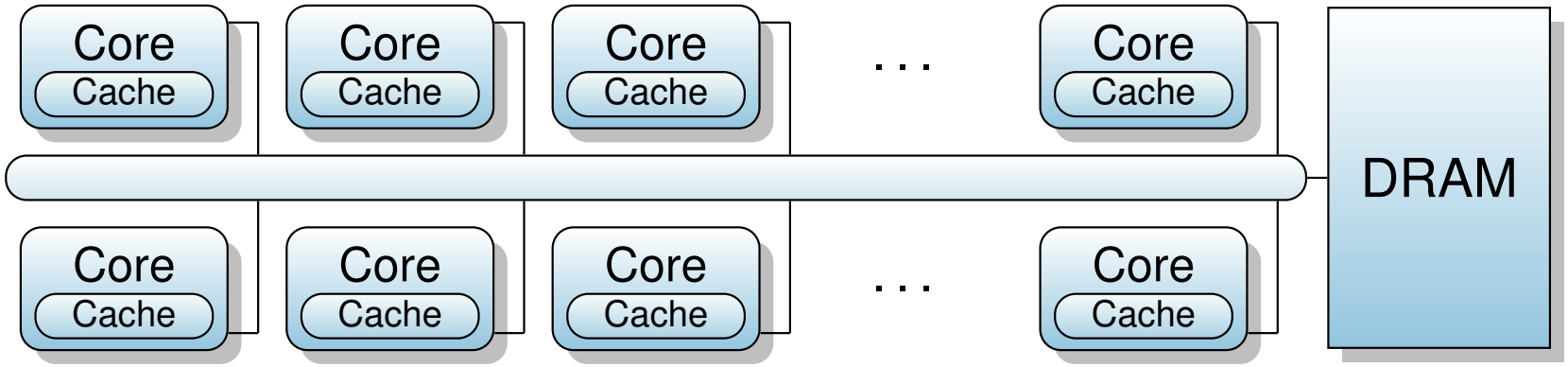
Non-Cache-Coherent Architectures



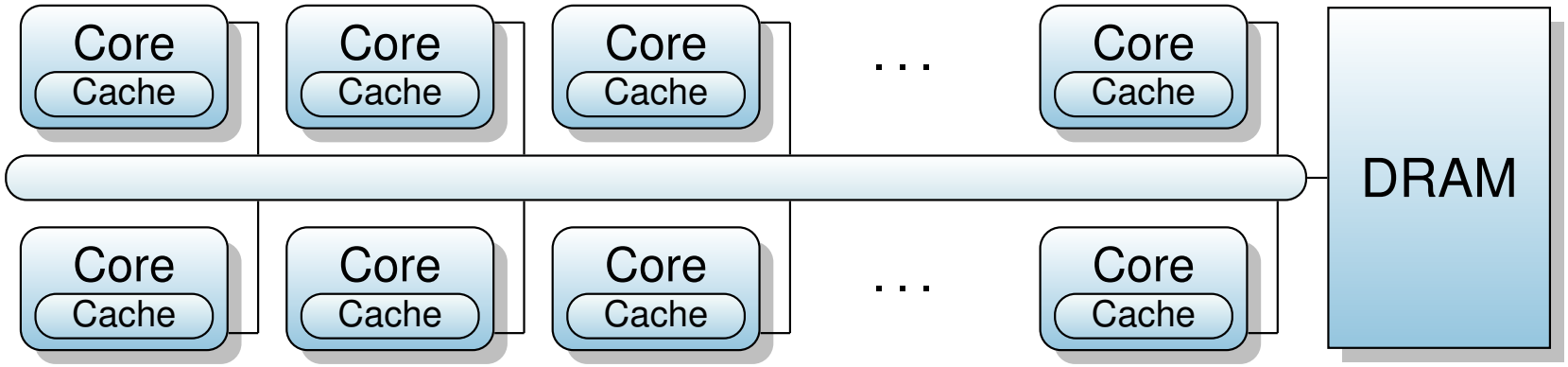
Non-Cache-Coherent Architectures



Non-Cache-Coherent Architectures

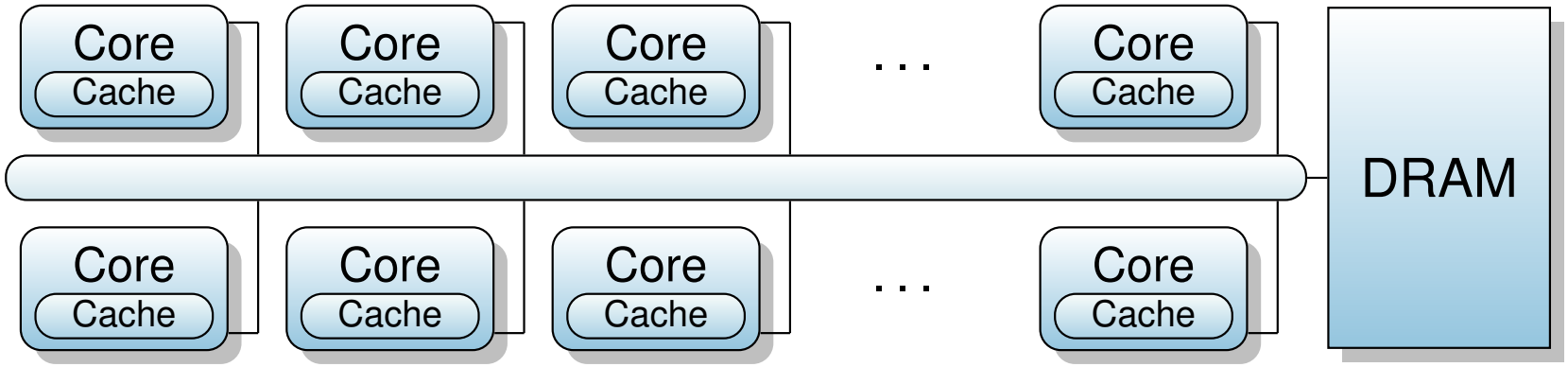


Non-Cache-Coherent Architectures



- Existing hardware coherence protocols hit scalability limit

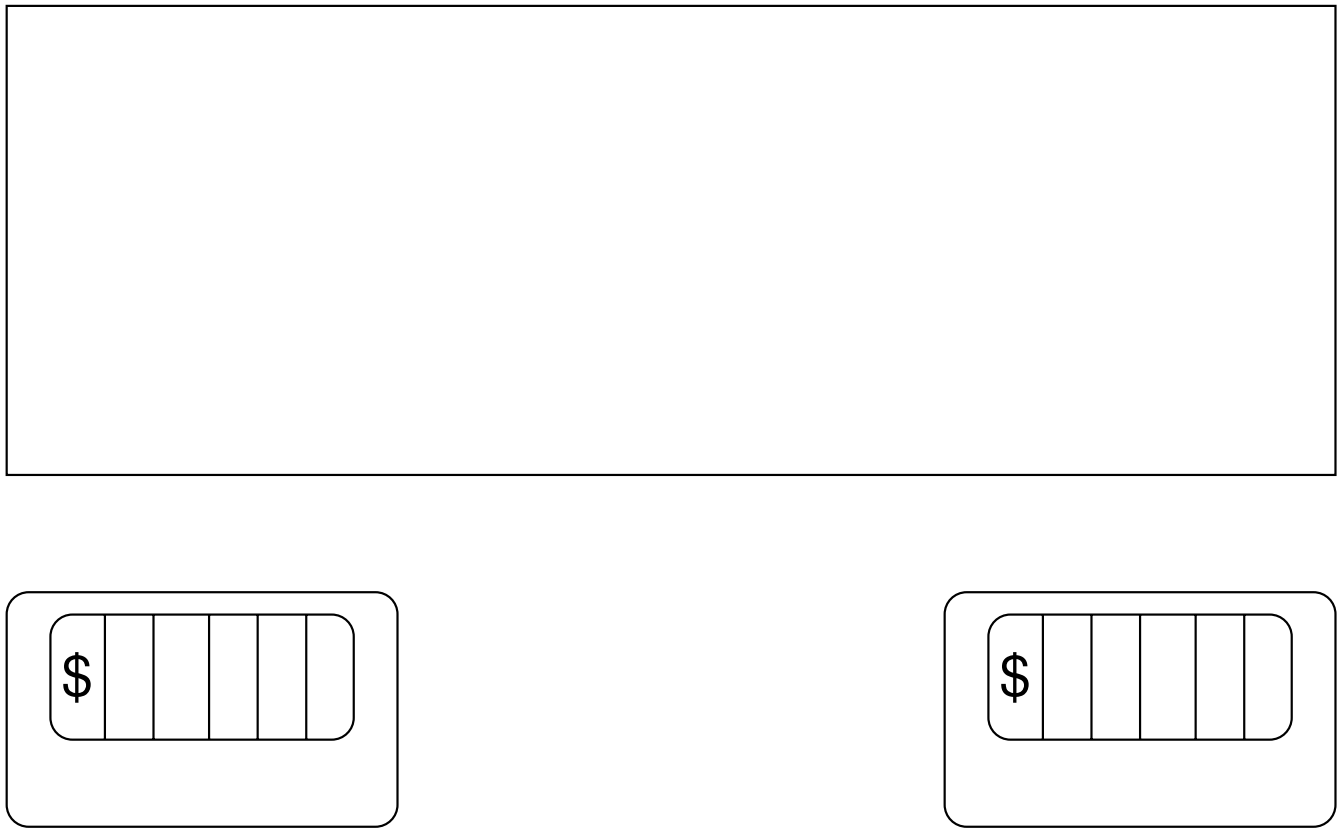
Non-Cache-Coherent Architectures



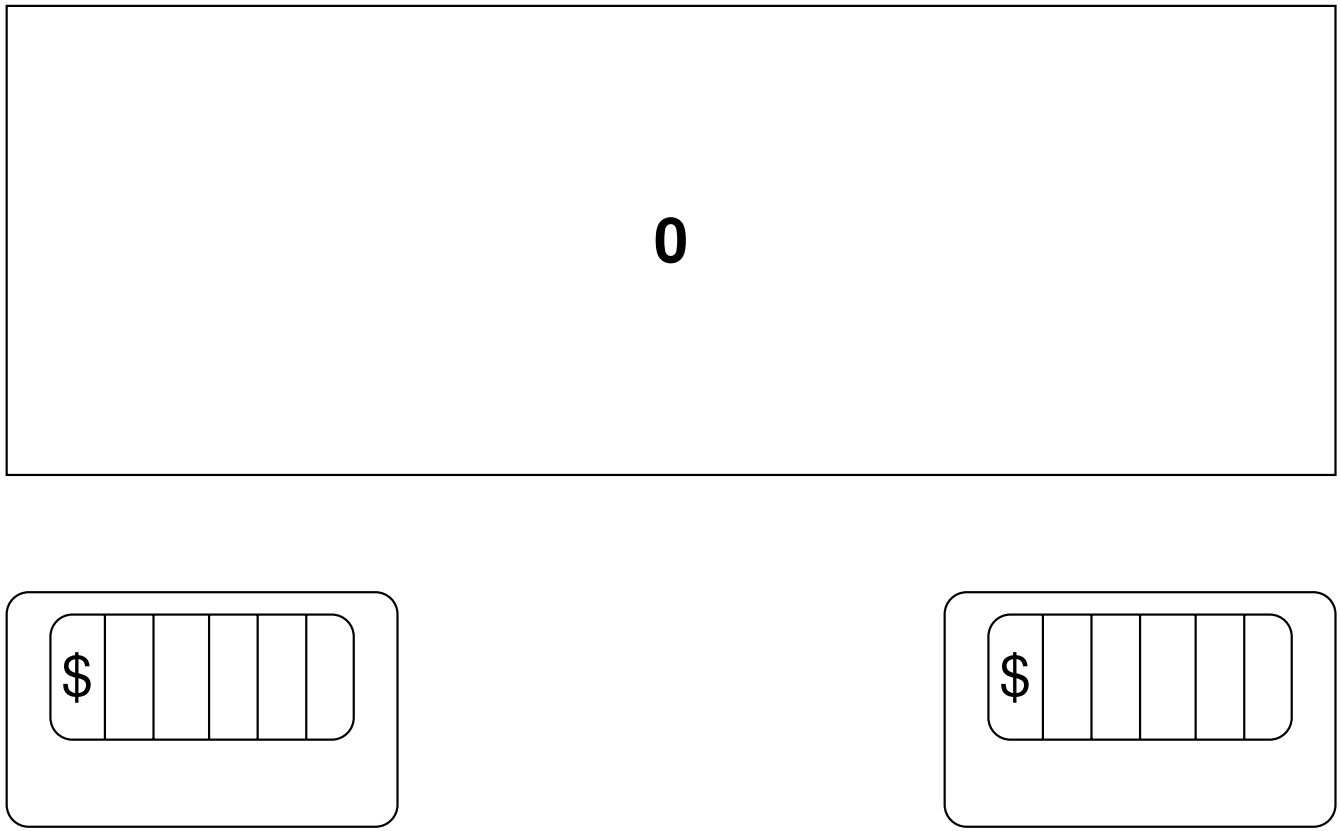
- Existing hardware coherence protocols hit scalability limit
- Radical answer: abandon global cache coherence
  - Still provide shared memory
  - Most prominent example: Intel SCC

Image source: Wikipedia, used under CC BY-SA 3.0

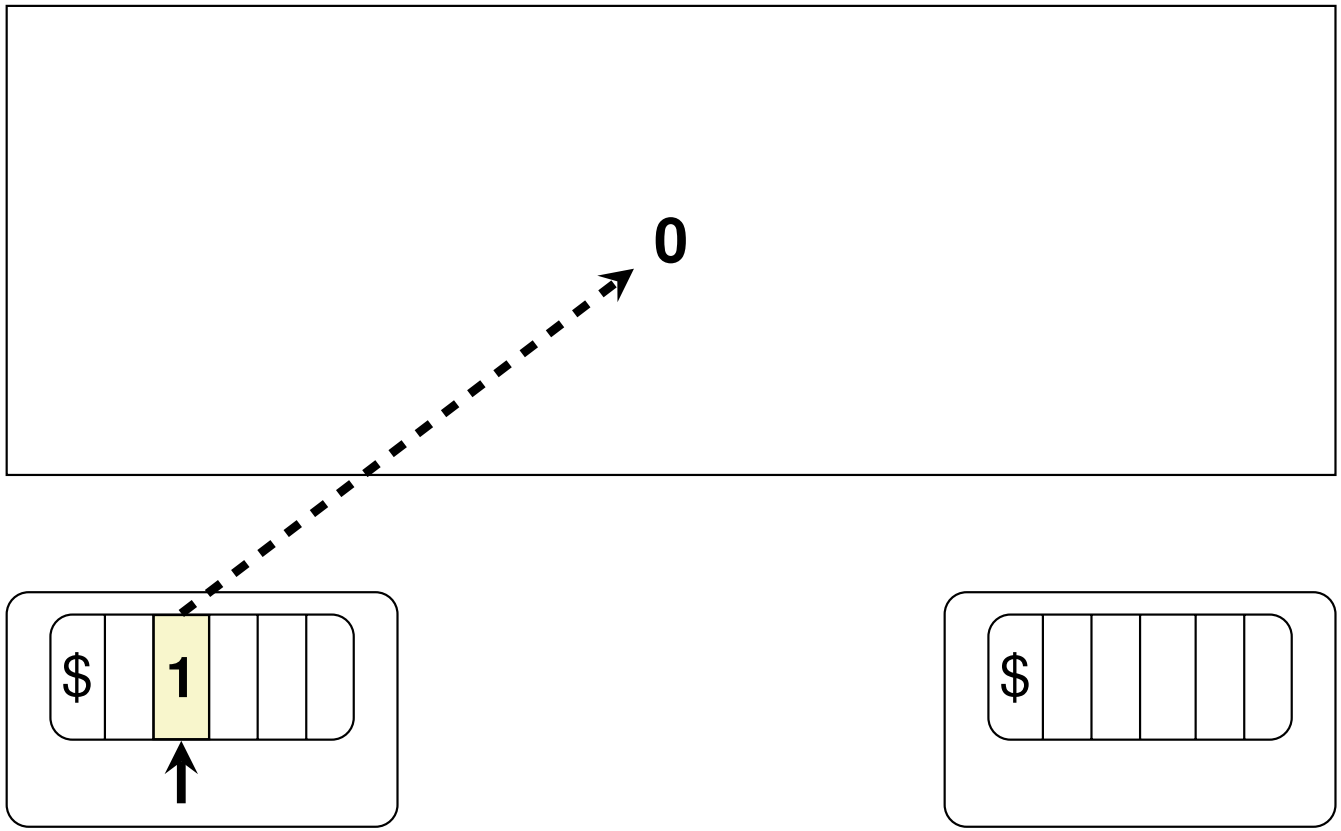
Shared-Memory Programming Model



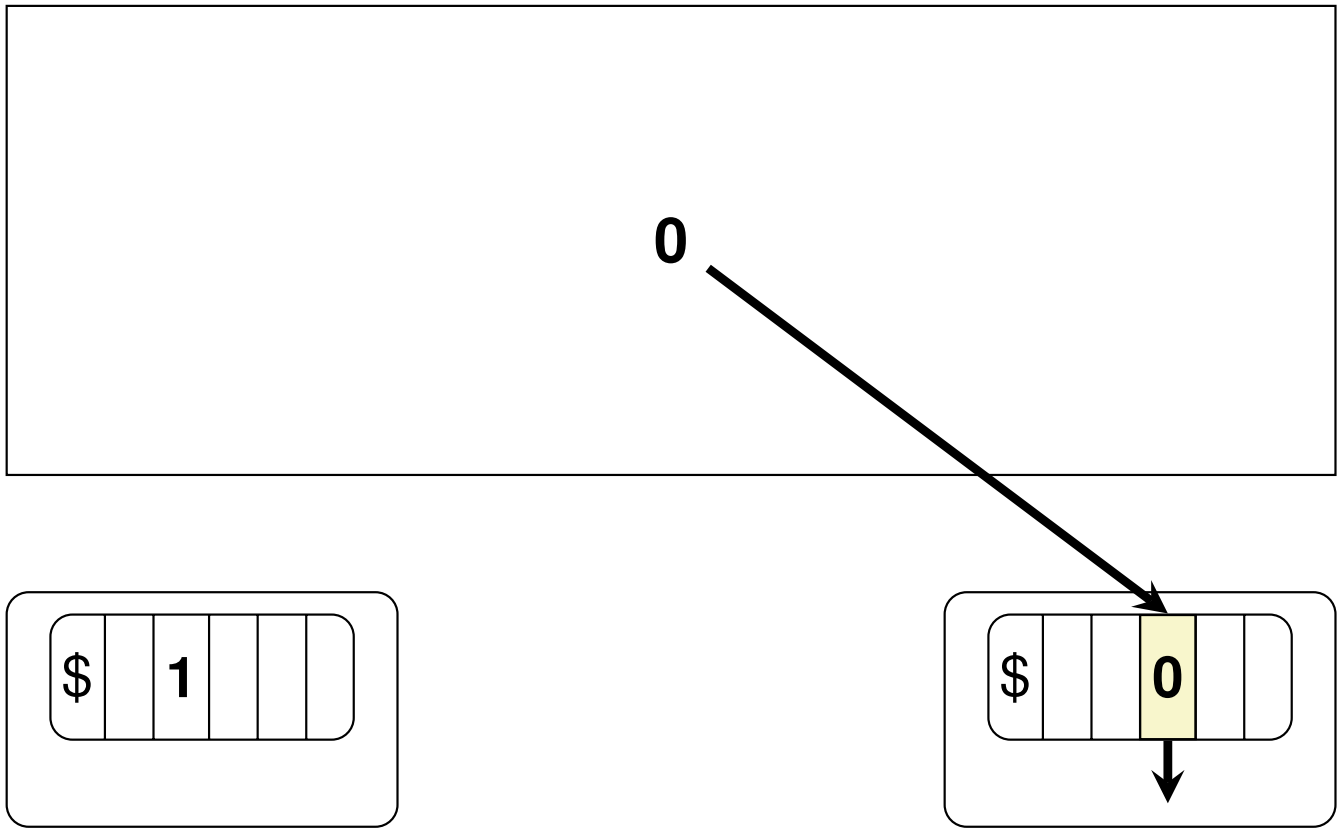
Shared-Memory Programming Model



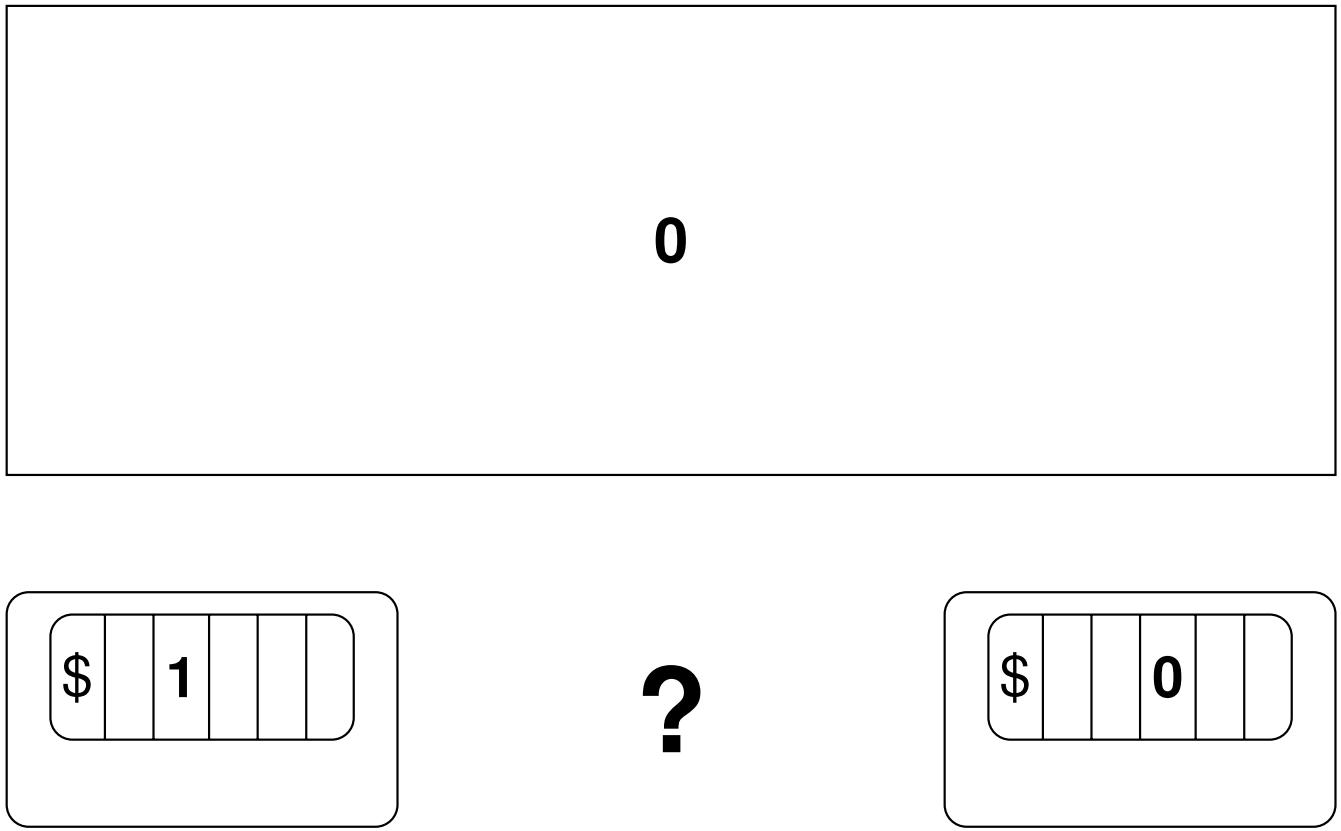
Shared-Memory Programming Model



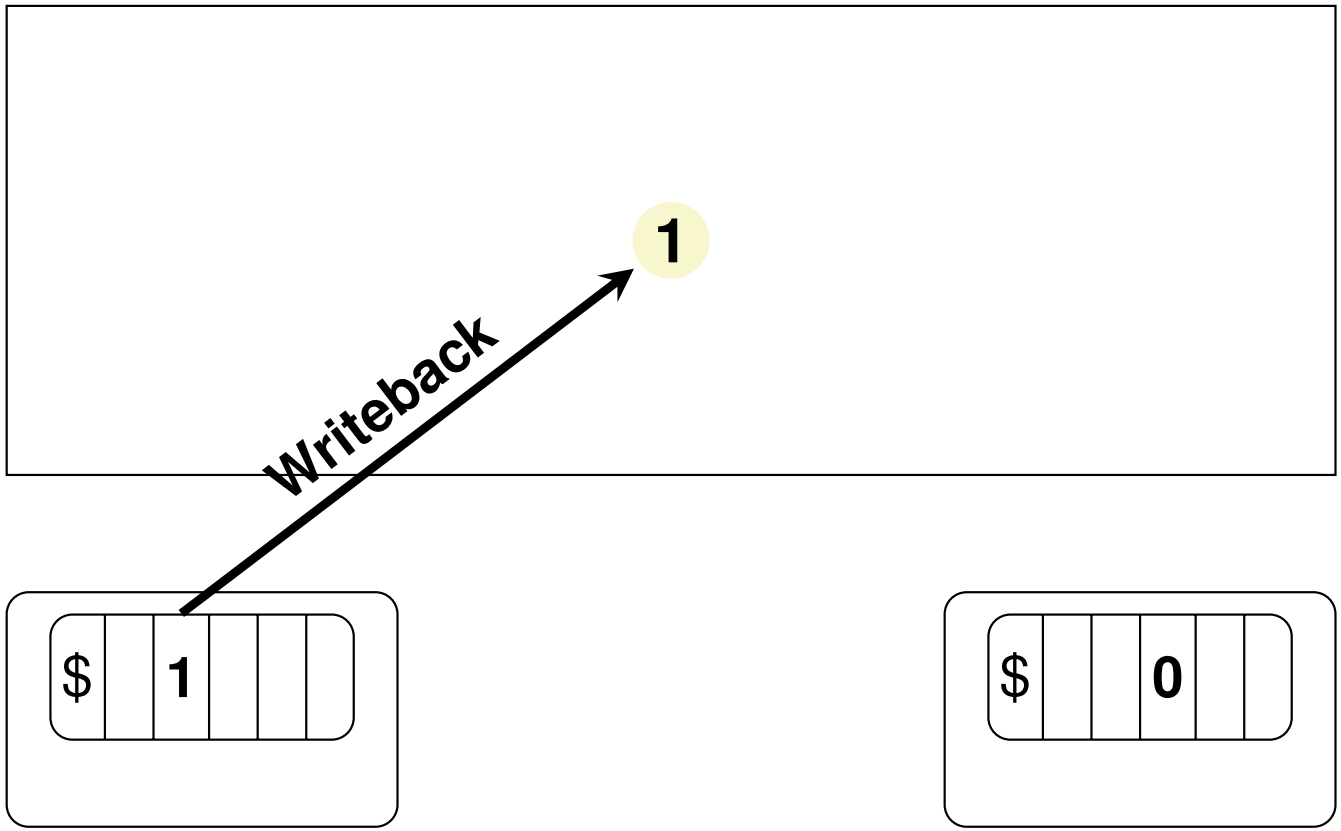
Shared-Memory Programming Model



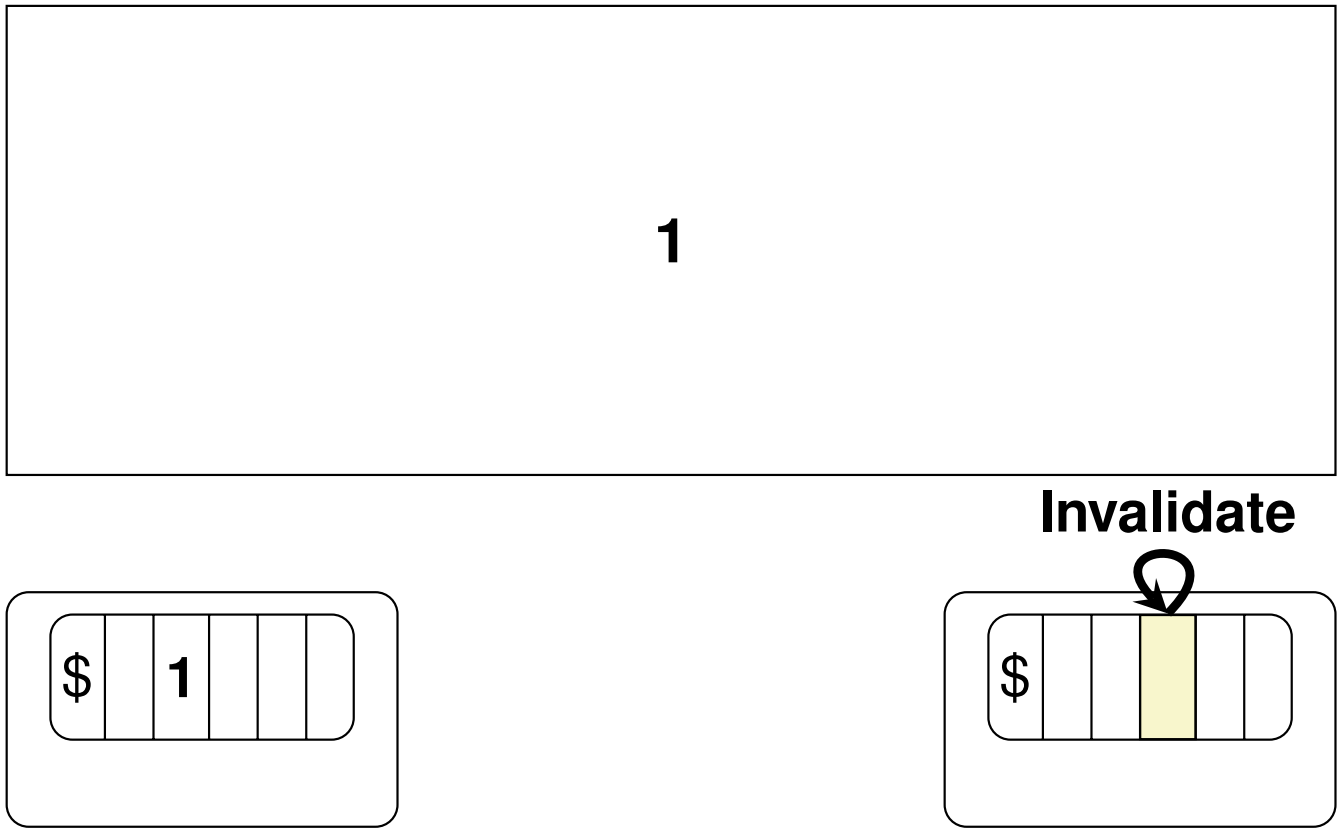
Shared-Memory Programming Model



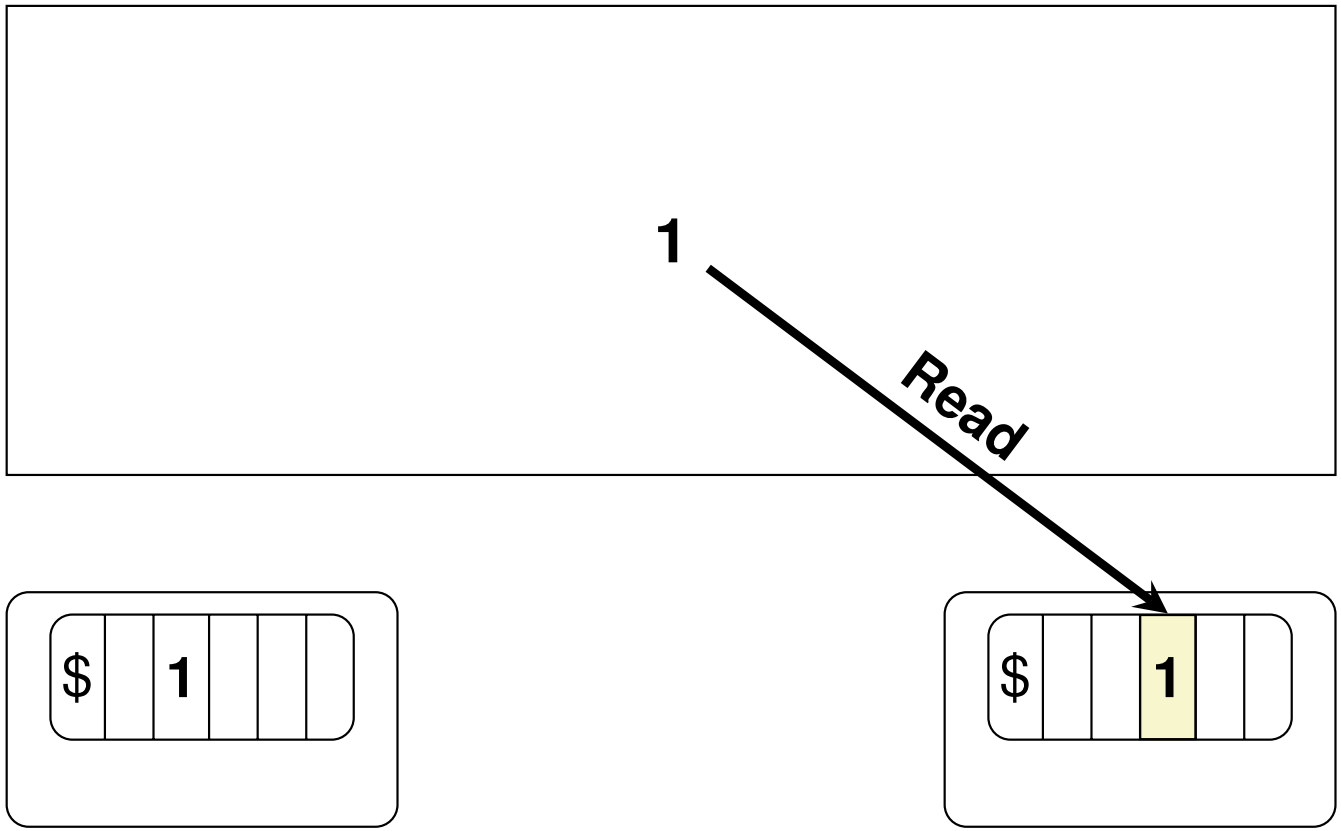
Shared-Memory Programming Model



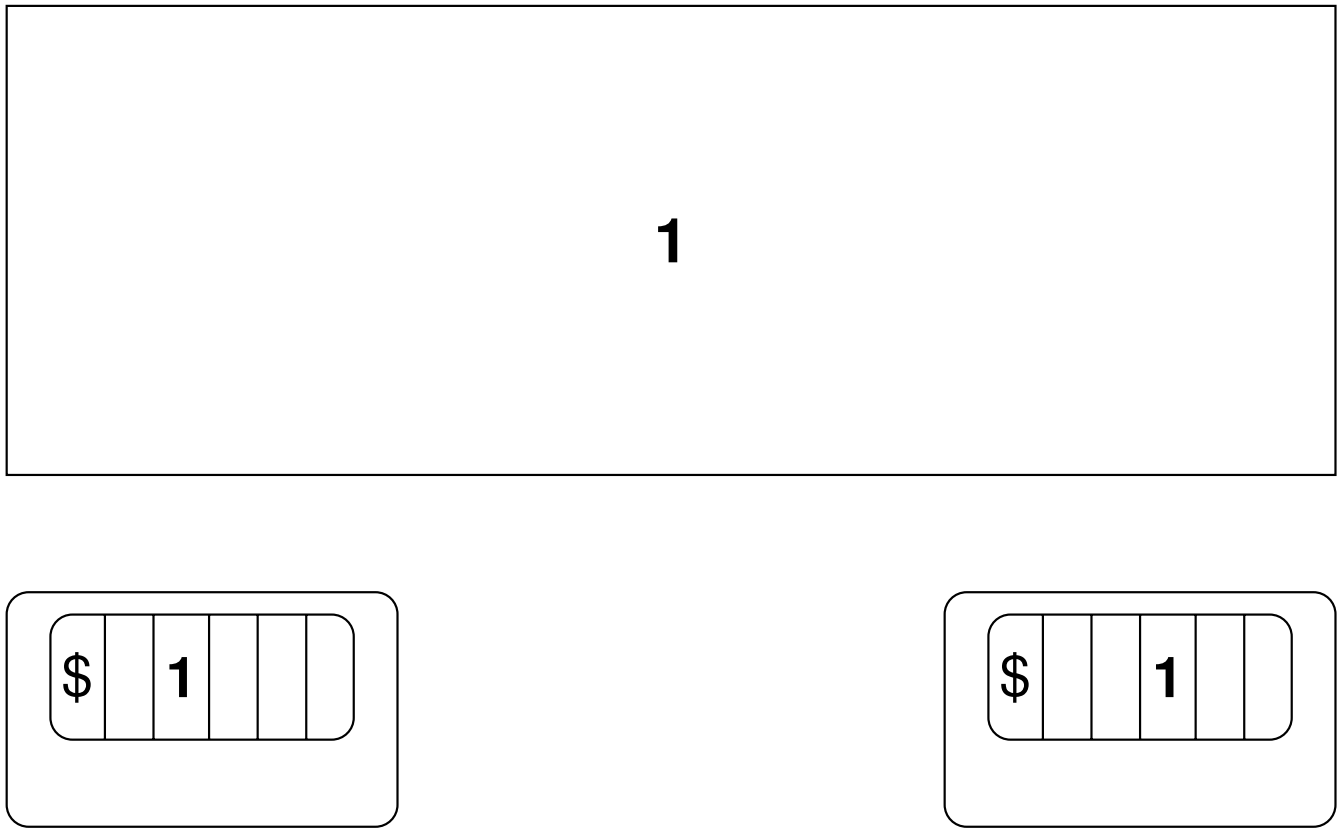
Shared-Memory Programming Model



Shared-Memory Programming Model

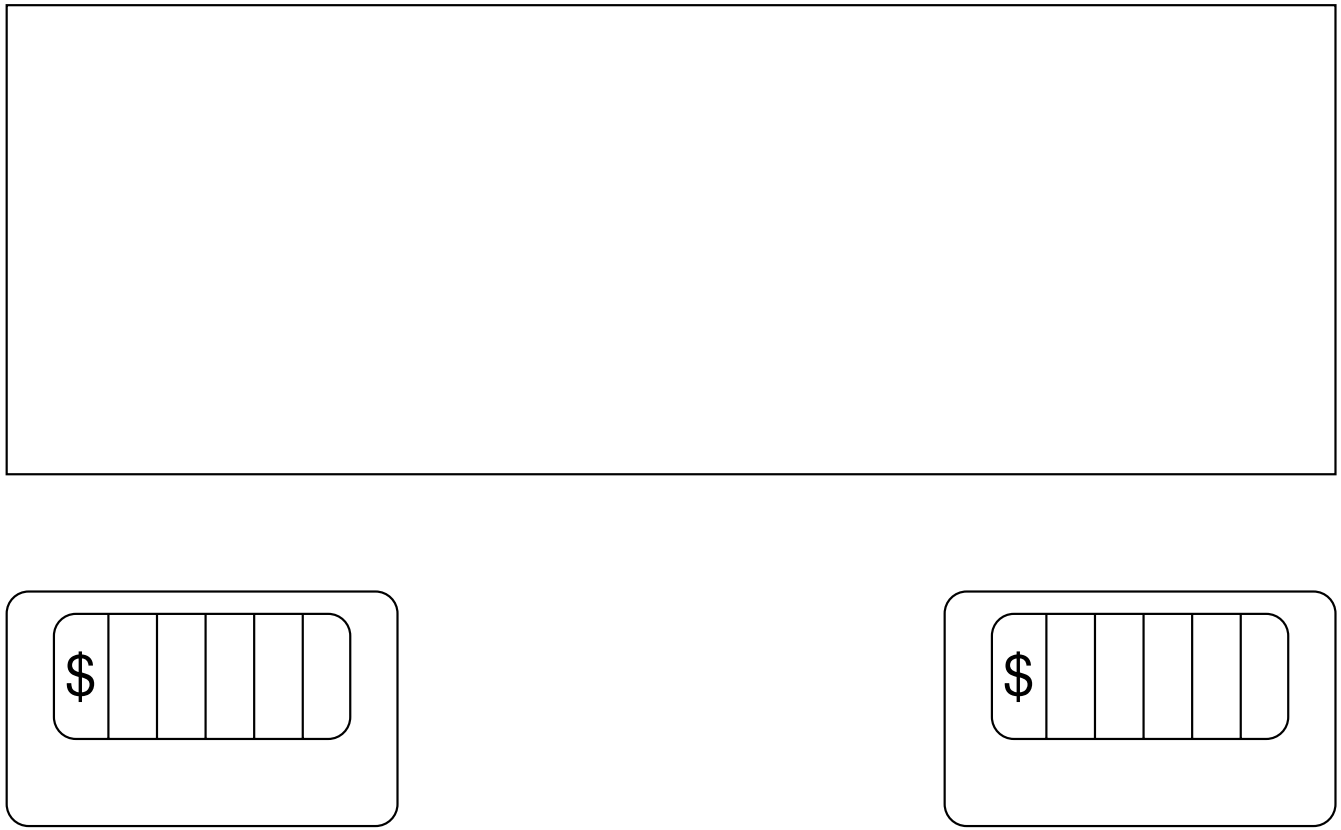


Shared-Memory Programming Model

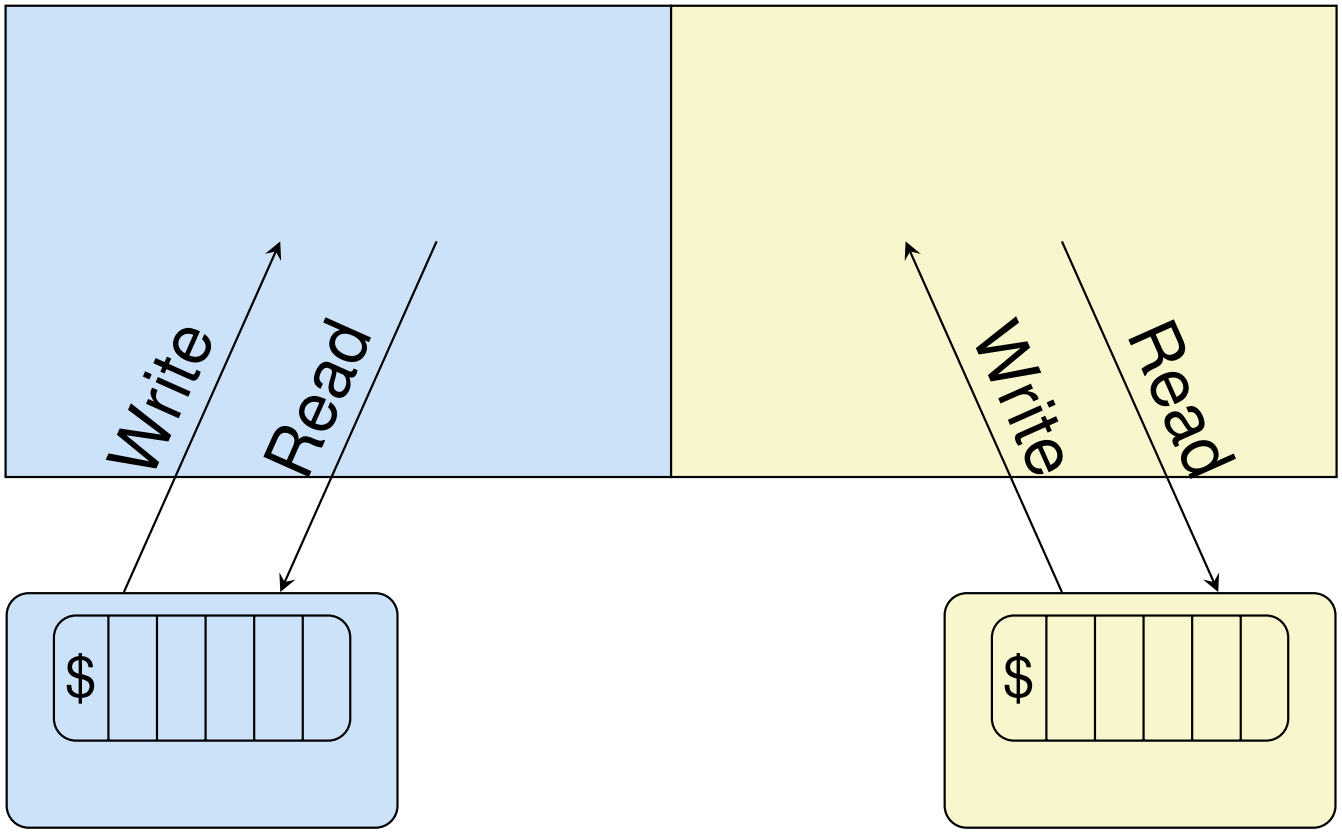


- Feasible, but can be expensive

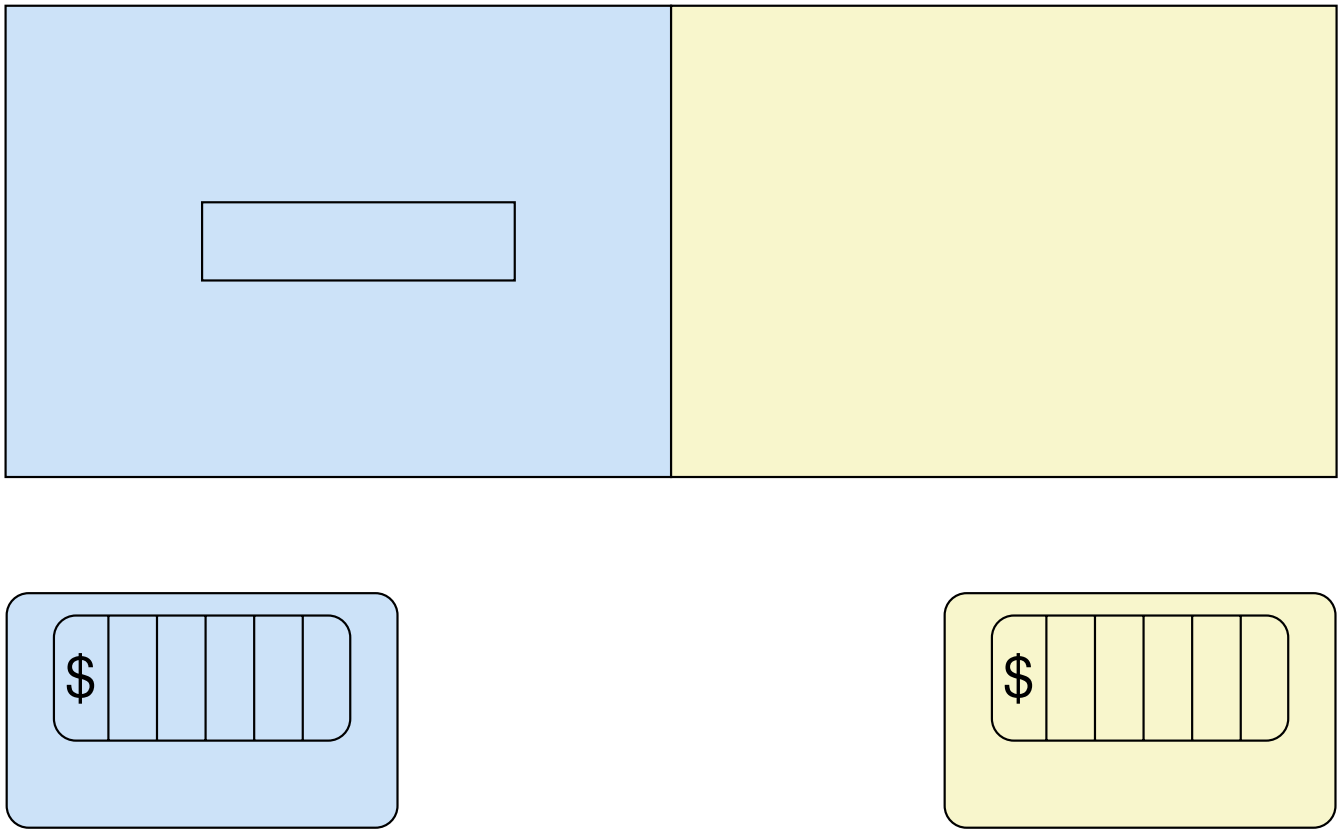
Partitioned Global Address Space (PGAS) Model



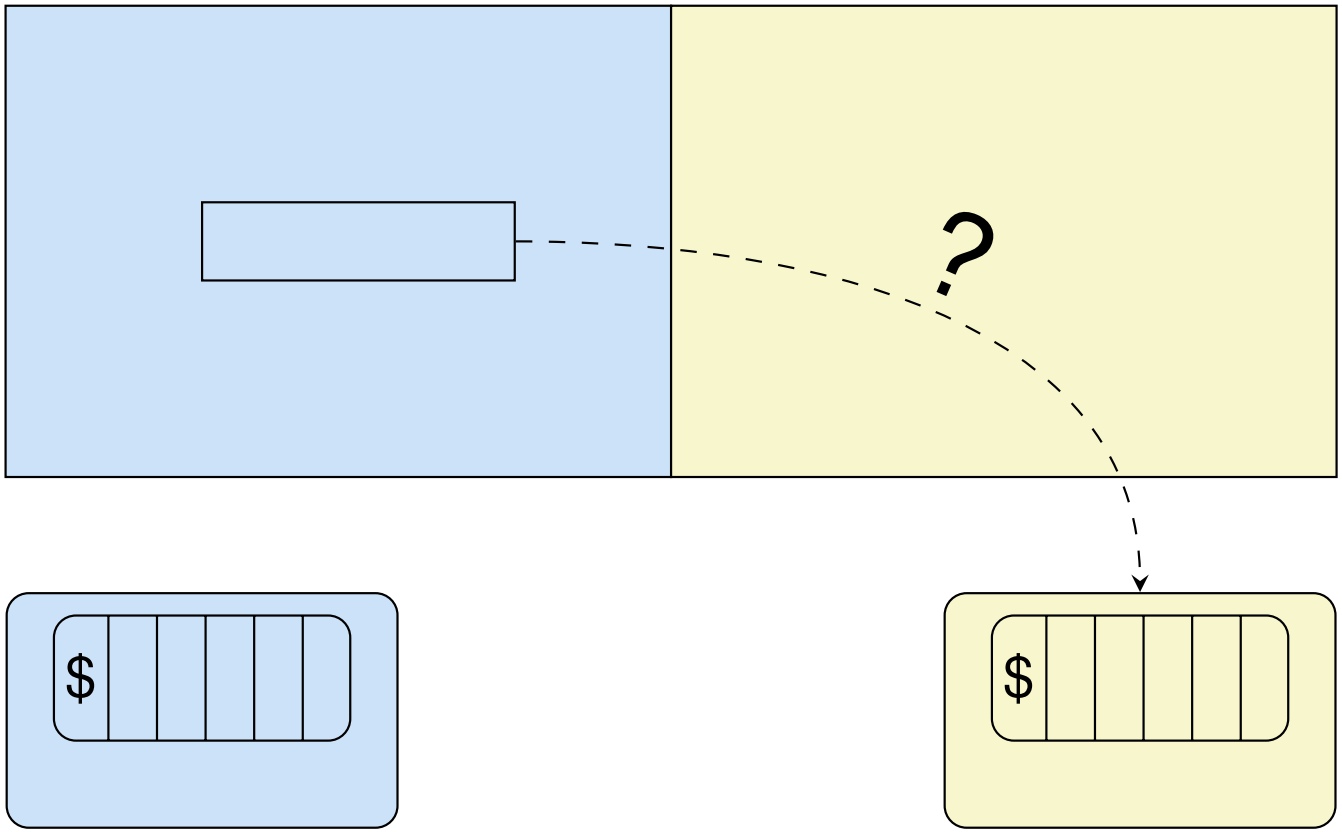
Partitioned Global Address Space (PGAS) Model



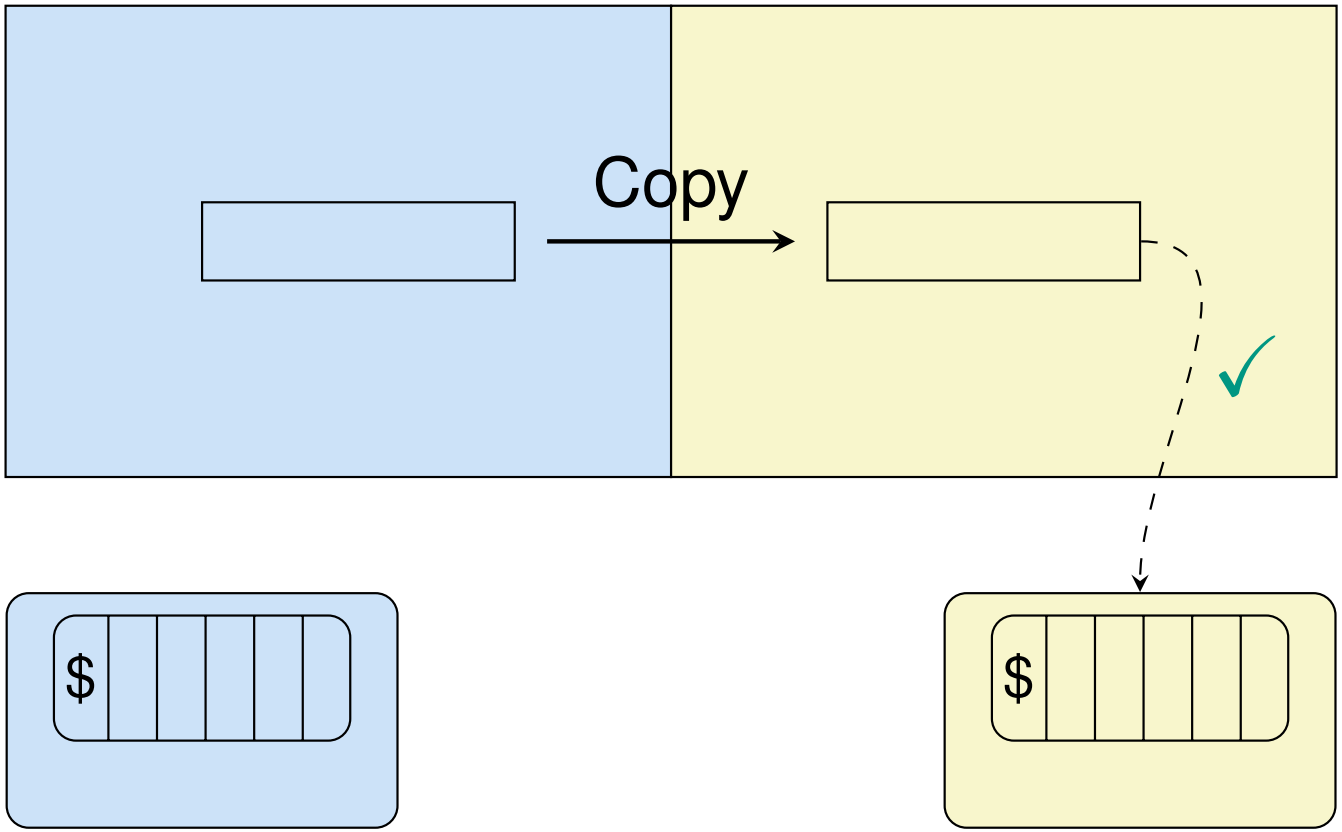
Partitioned Global Address Space (PGAS) Model



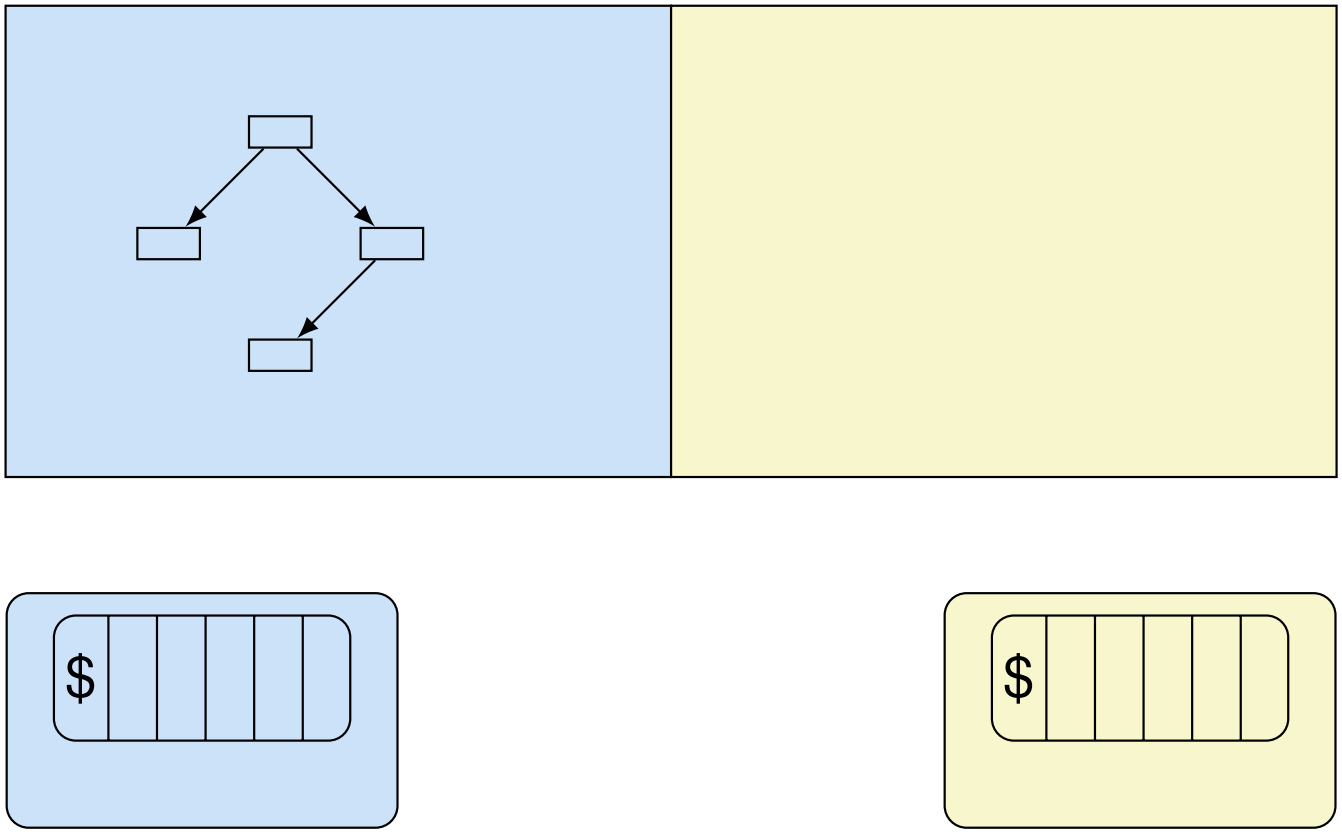
Partitioned Global Address Space (PGAS) Model



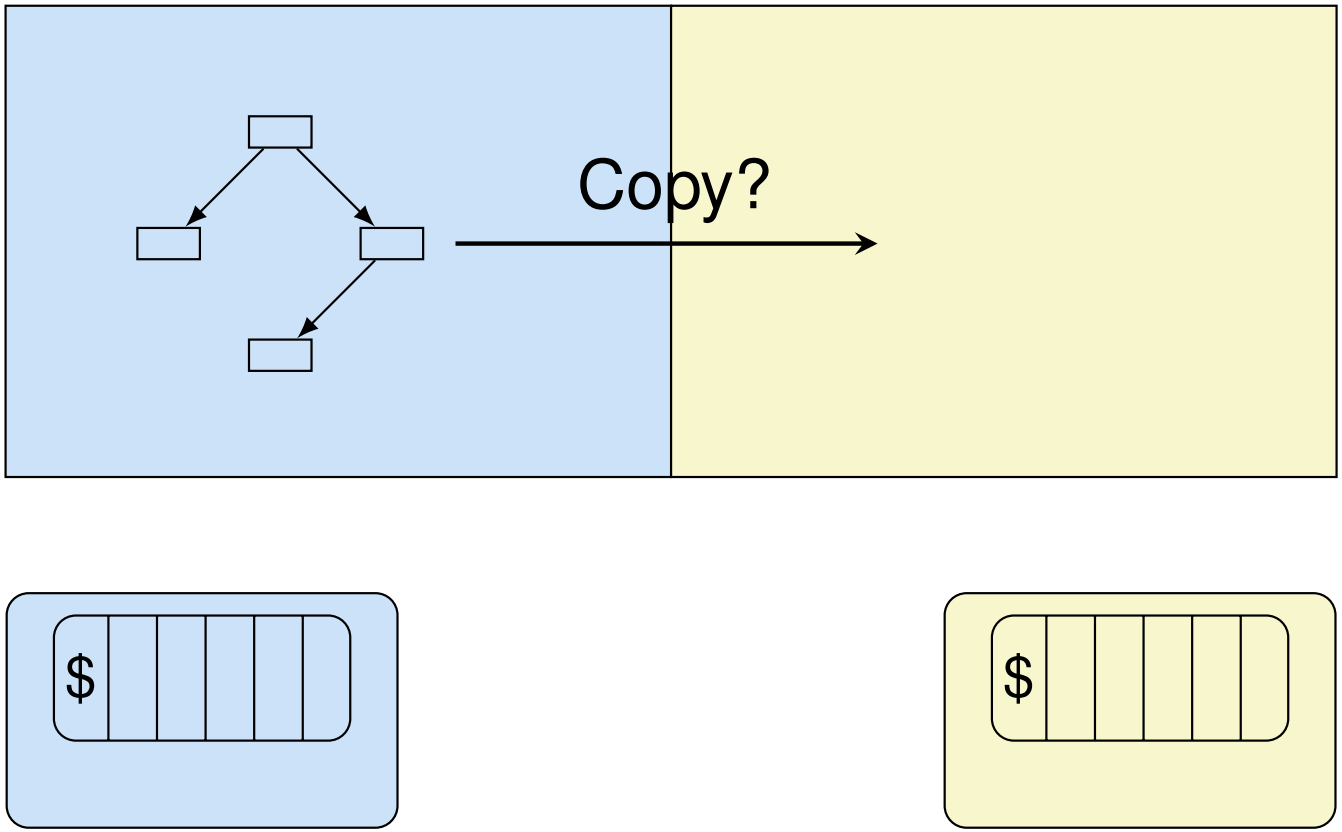
Partitioned Global Address Space (PGAS) Model



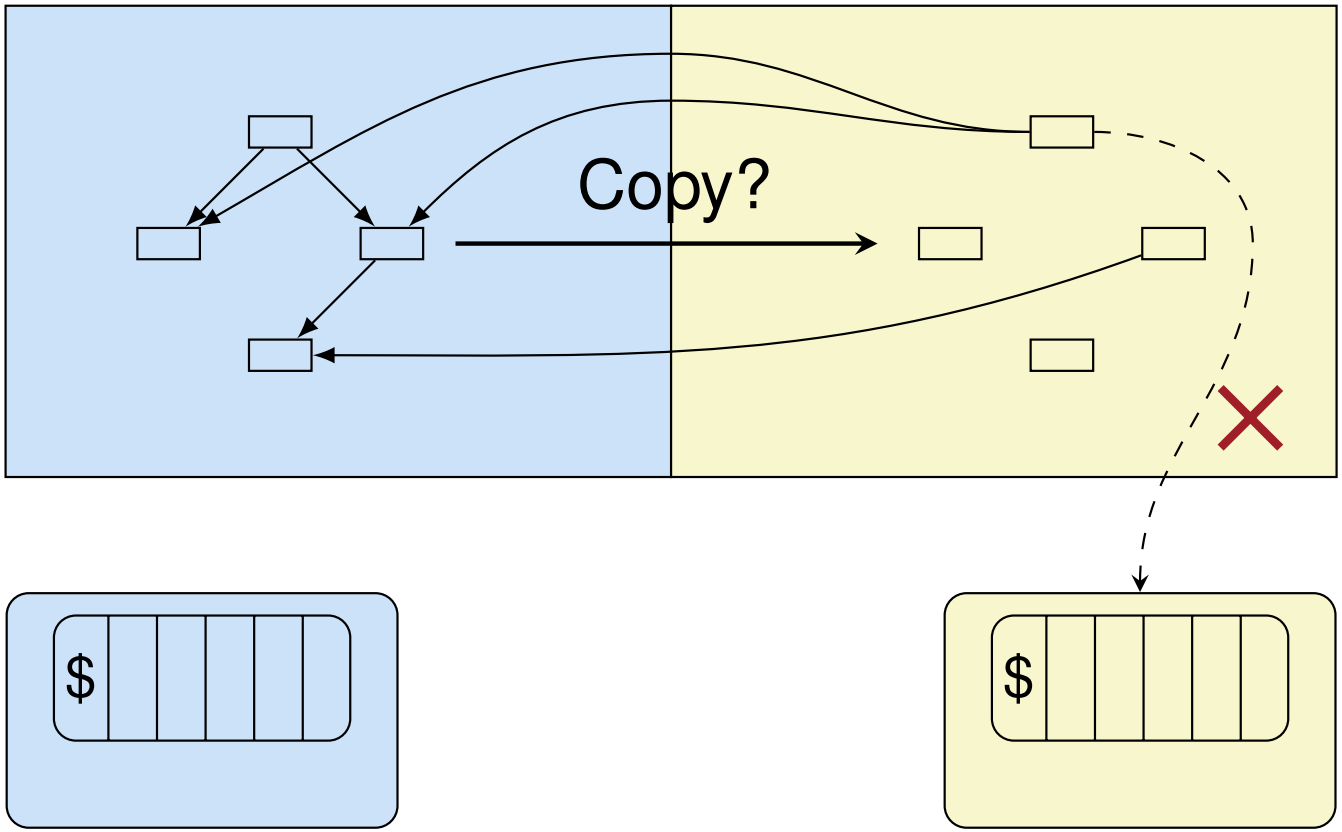
Partitioned Global Address Space (PGAS) Model



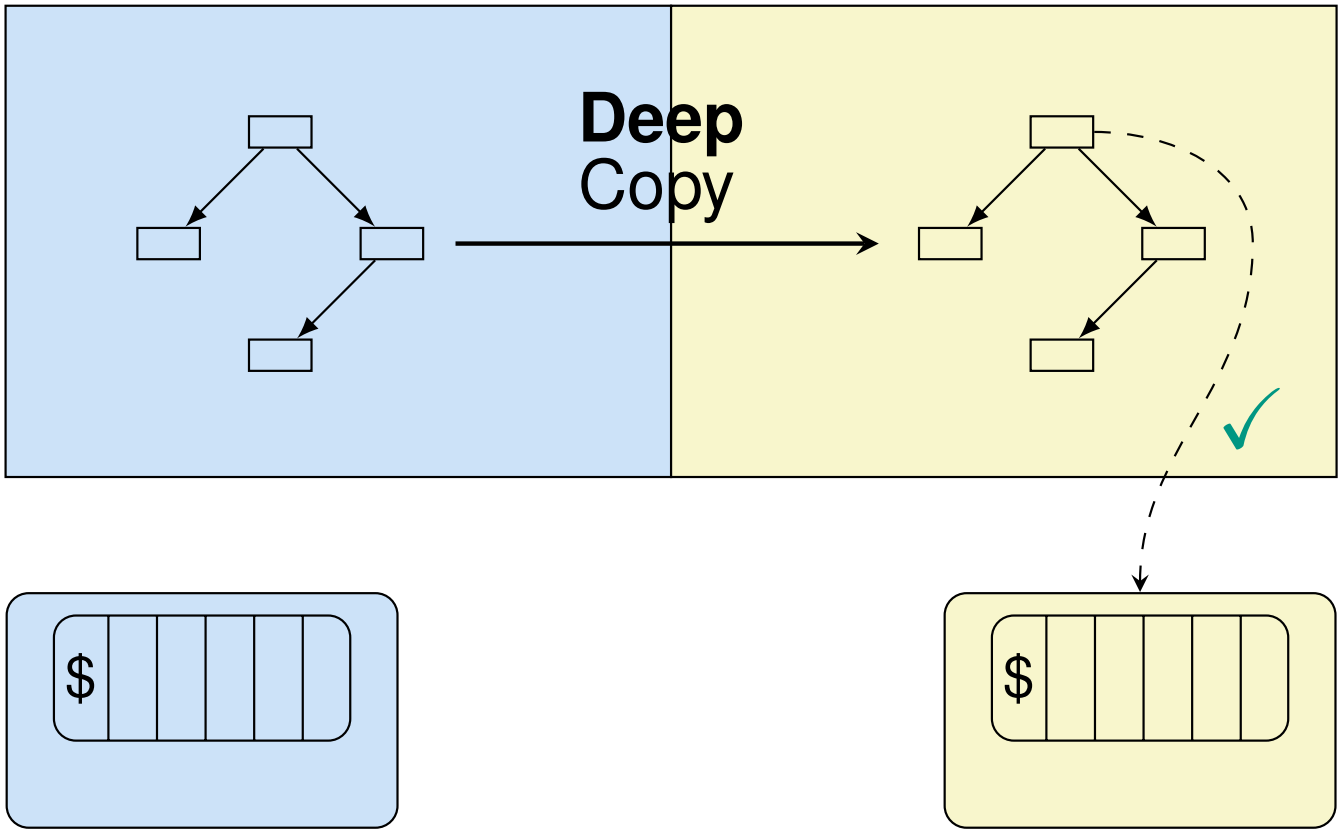
Partitioned Global Address Space (PGAS) Model



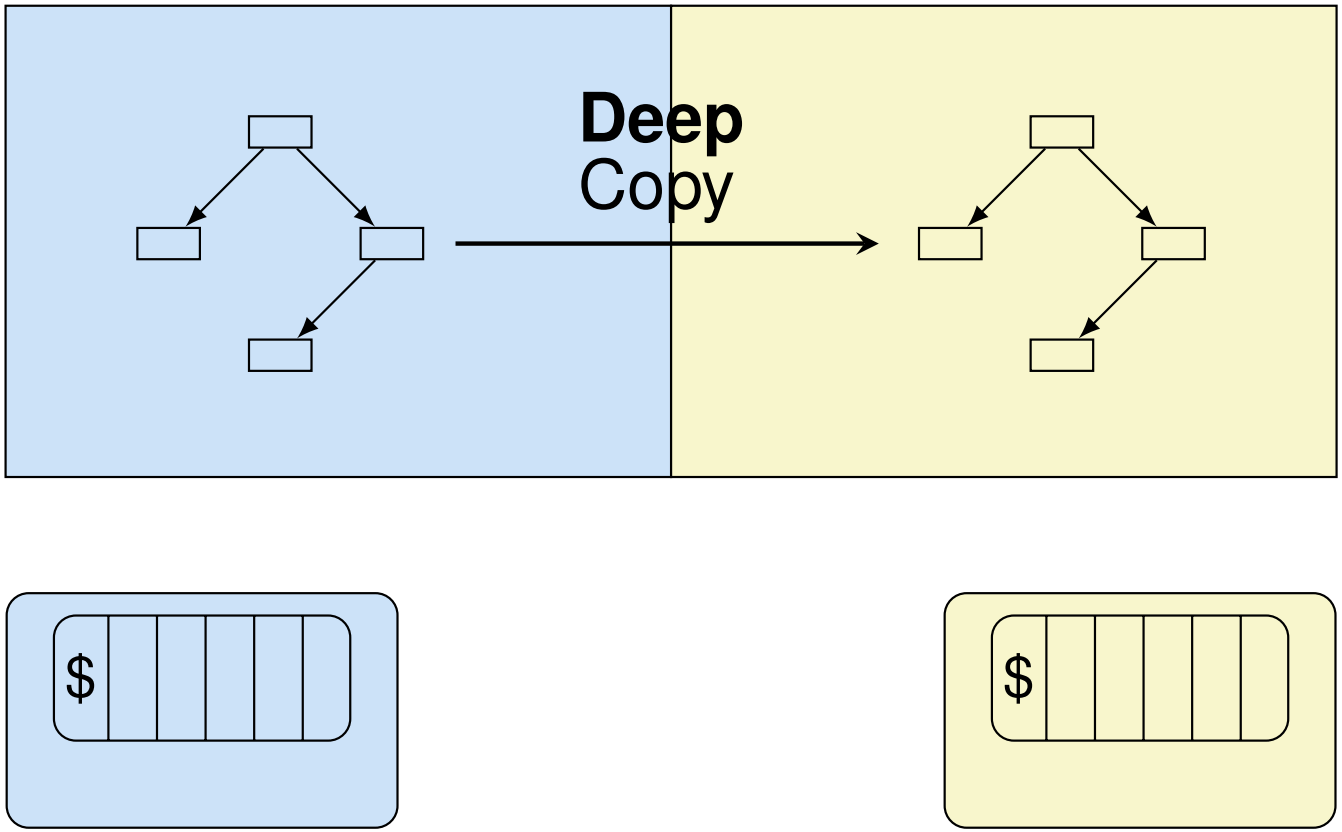
Partitioned Global Address Space (PGAS) Model



Partitioned Global Address Space (PGAS) Model

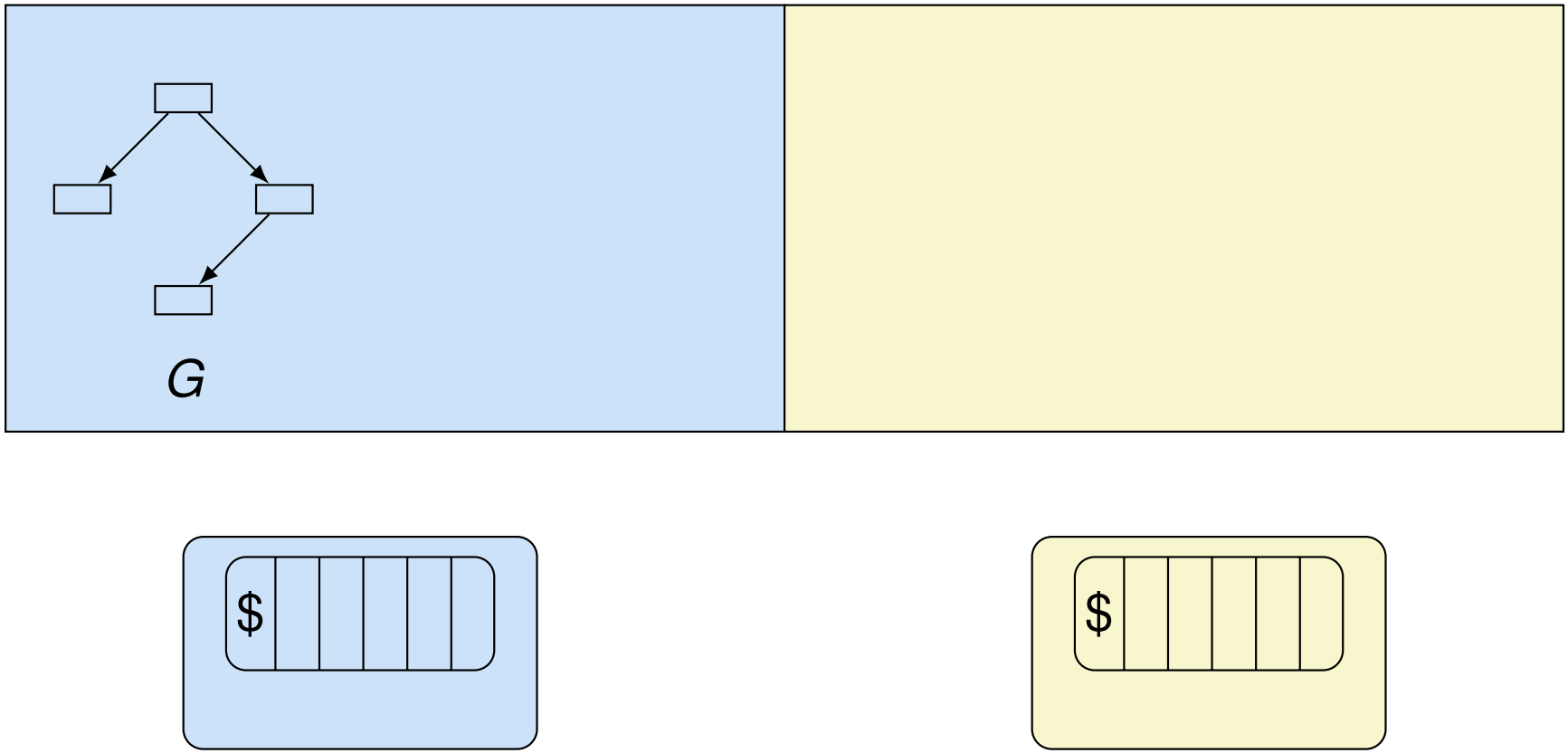


Partitioned Global Address Space (PGAS) Model

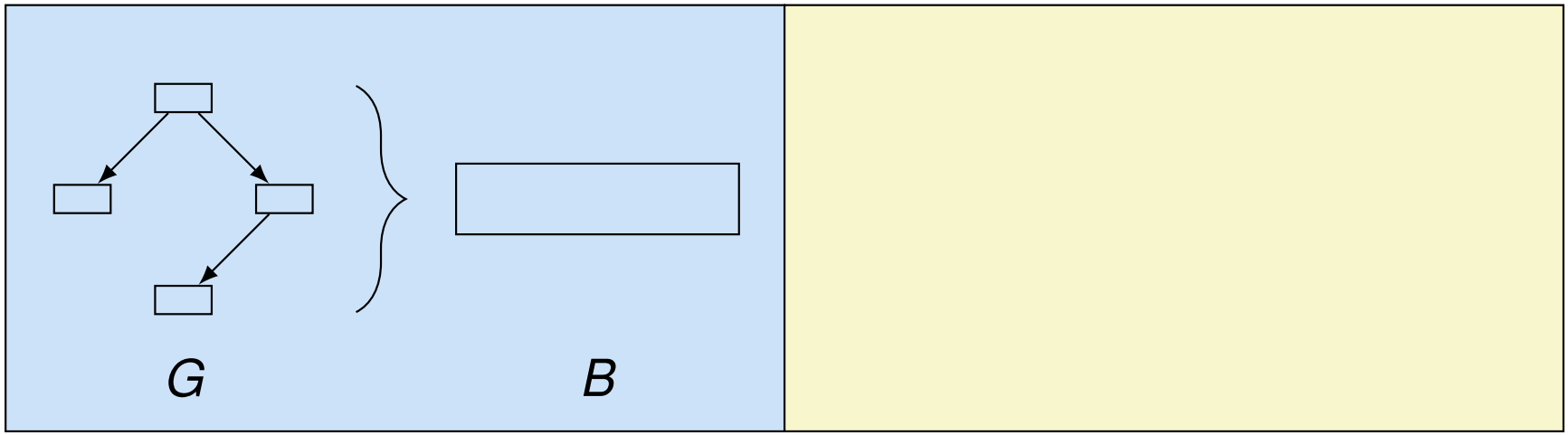


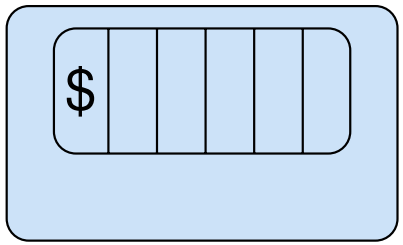
Goal: Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures

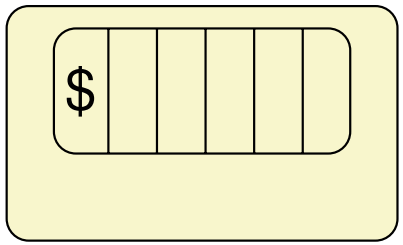
Message Passing (MP)



Message Passing (MP)

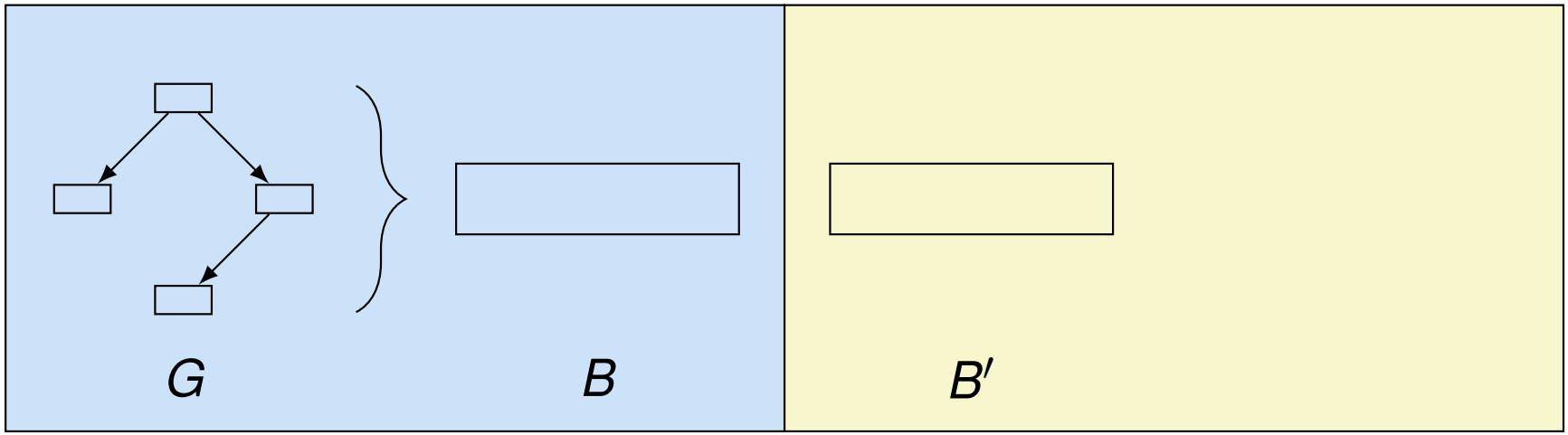


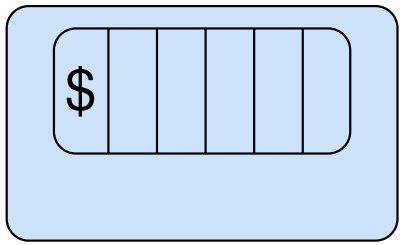


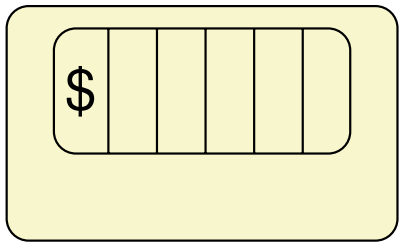


1. Serialize *G* to *B*

Message Passing (MP)

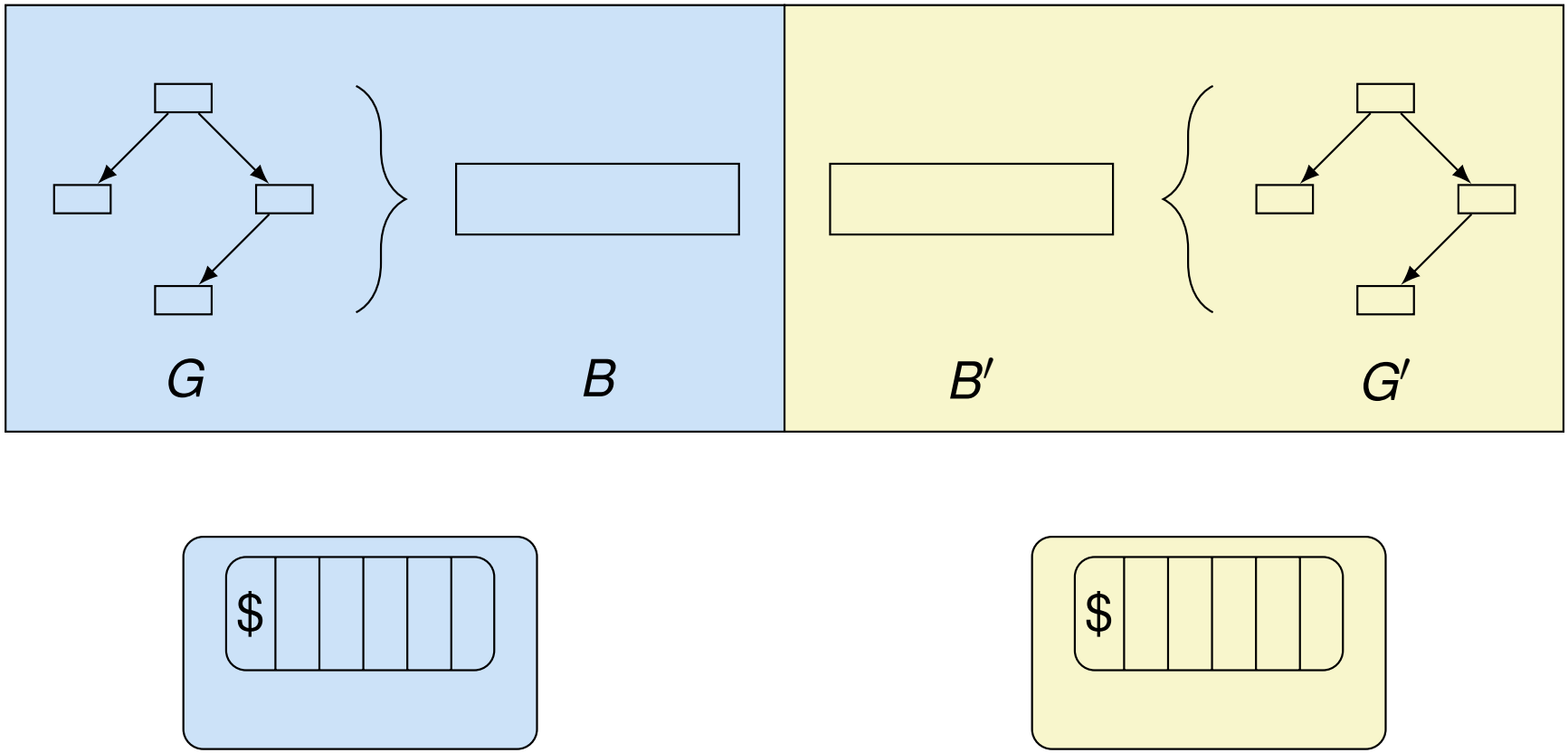






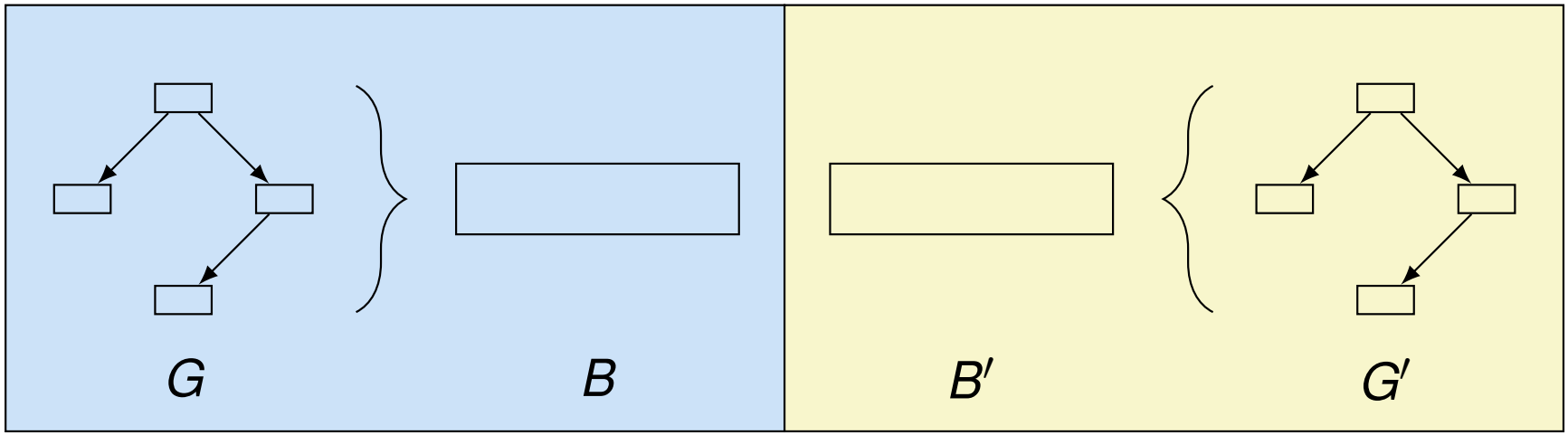
1. Serialize *G* to *B*
2. Transfer *B* to *B′*, e.g. using library

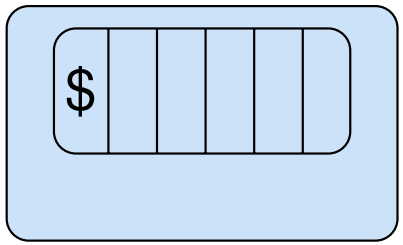
Message Passing (MP)

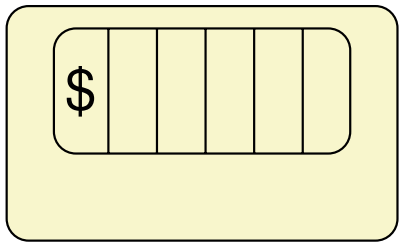


1. Serialize G to B
2. Transfer B to B', e.g. using library
3. Deserialize G' from B'

Message Passing (MP)



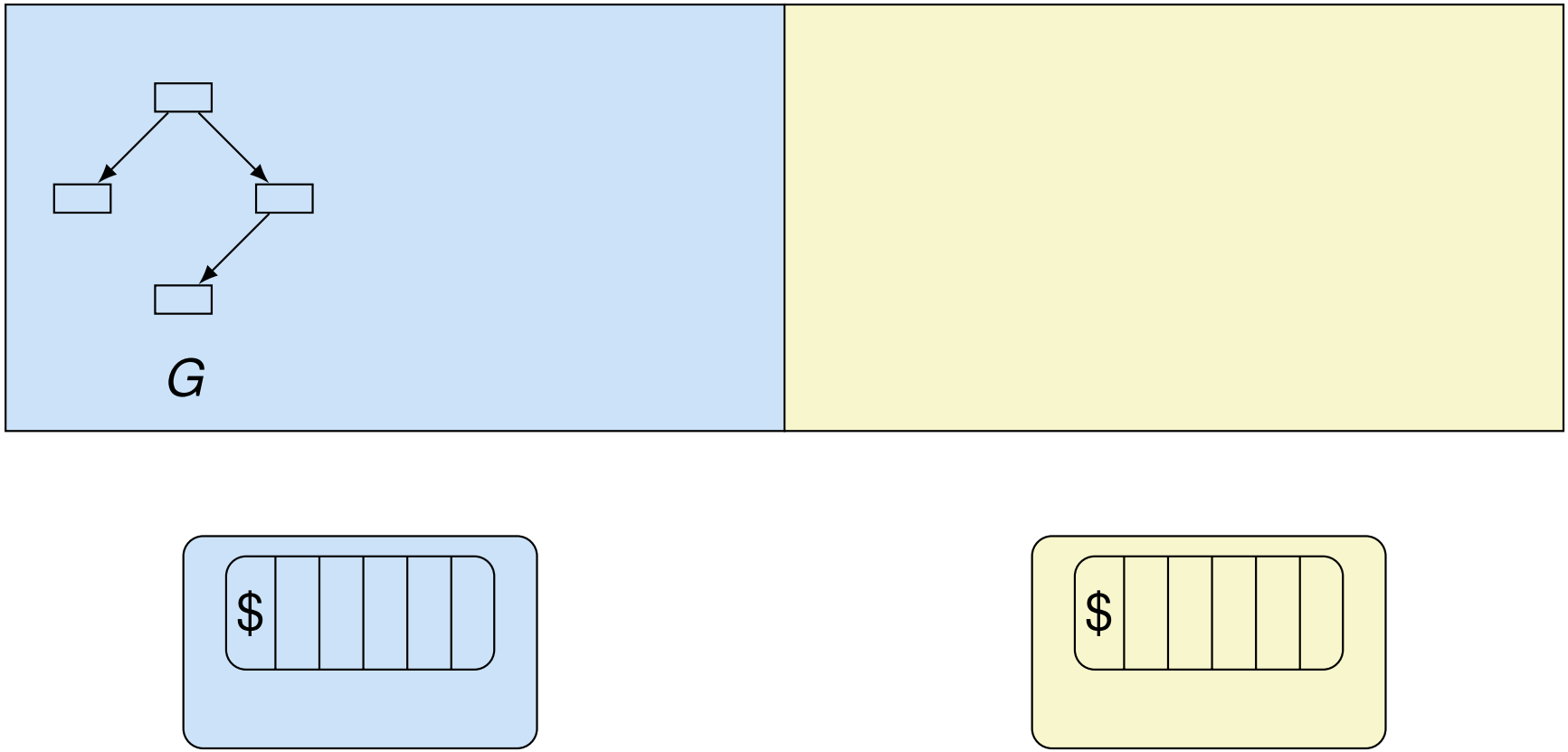




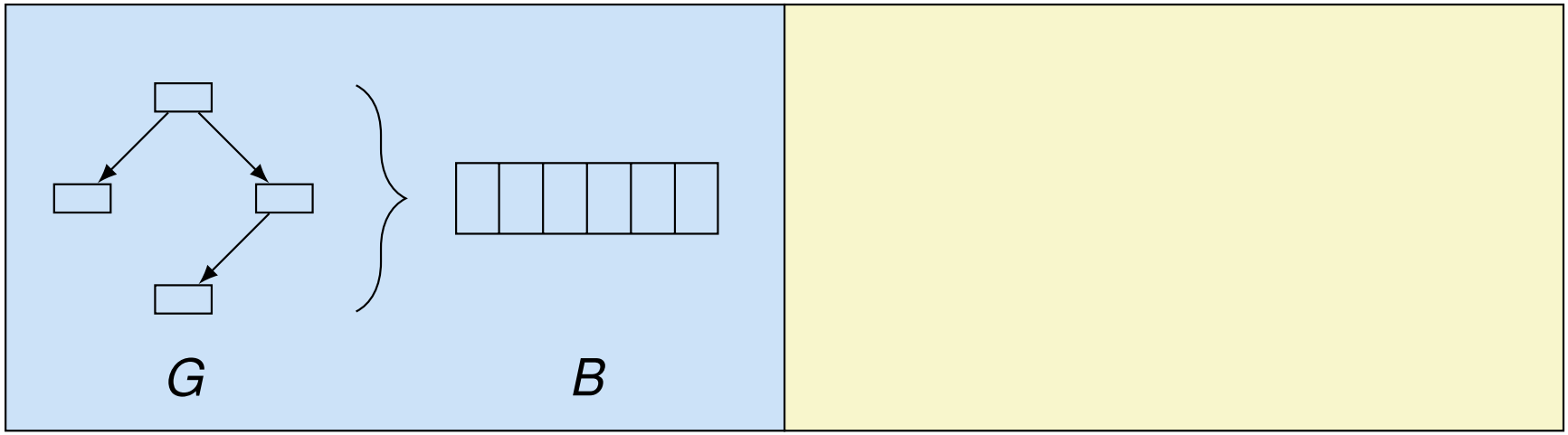
1. Serialize G to B
2. Transfer B to B', e.g. using library
3. Deserialize G' from B'

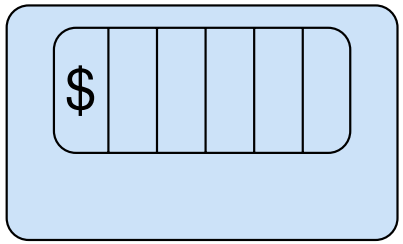
- Large memory overhead
- Serialization overhead
- B, B' pollute caches

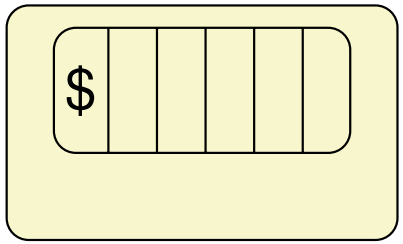
Message Passing via Shared Memory (MP-SHM)



Message Passing via Shared Memory (MP-SHM)

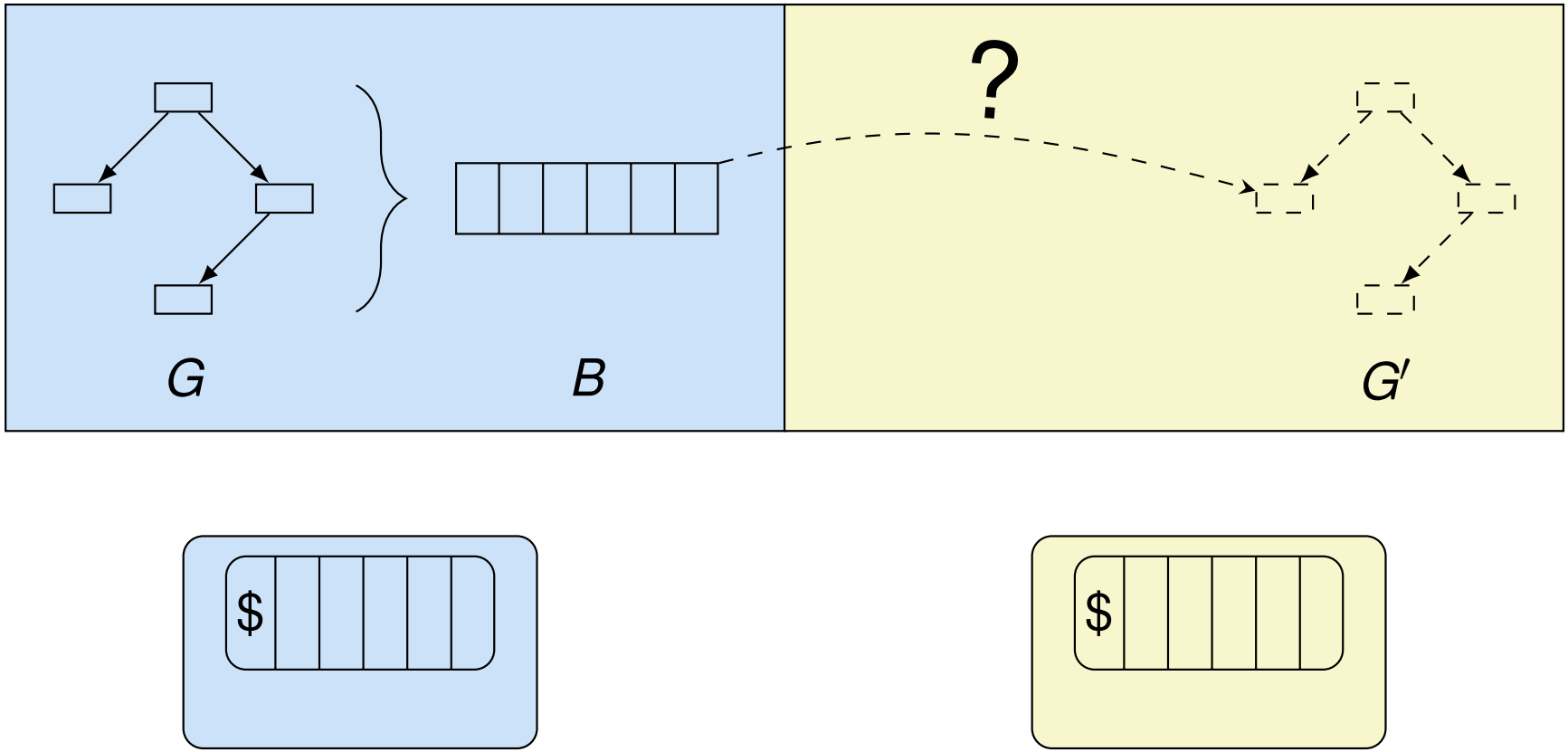






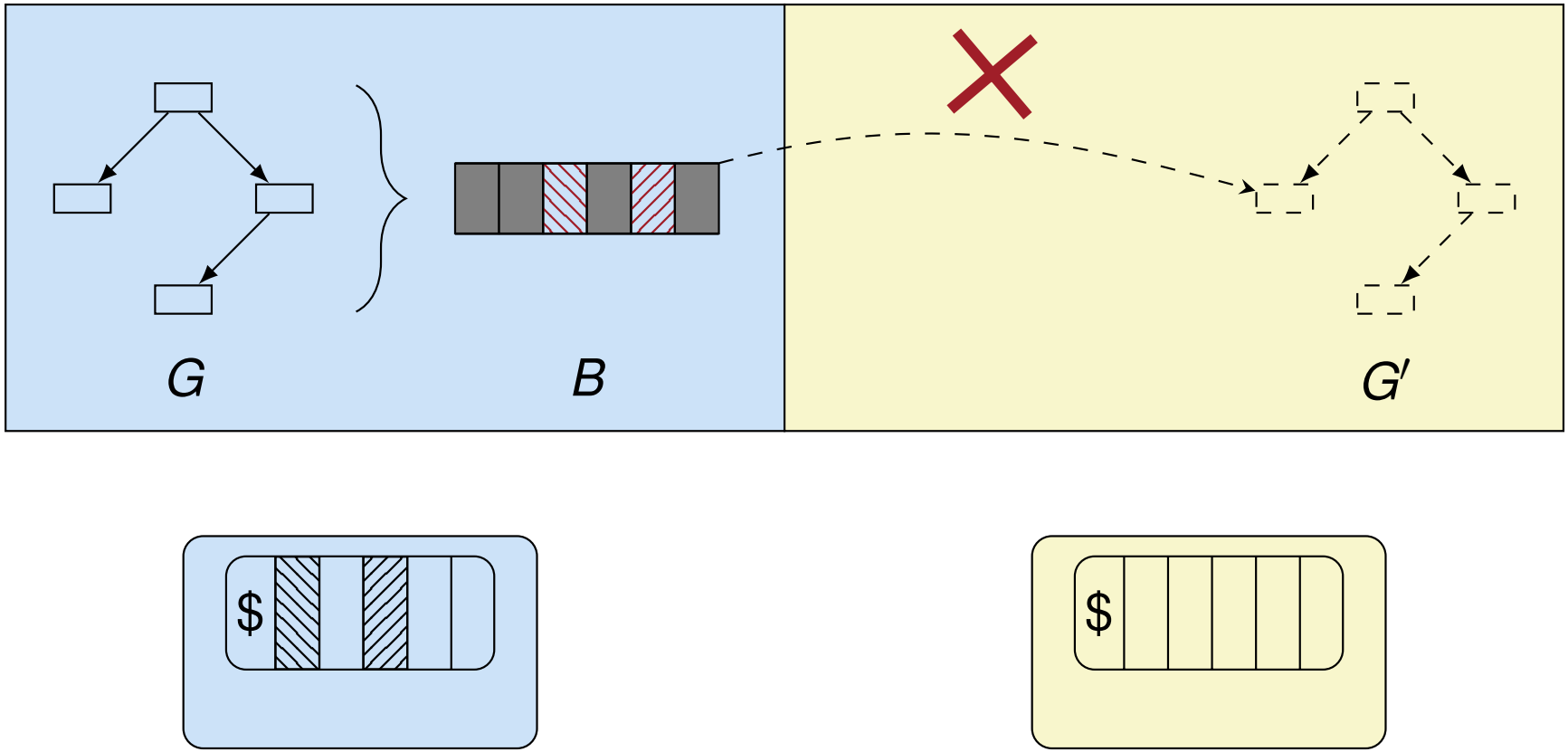
1. Serialize *G* to *B*

Message Passing via Shared Memory (MP-SHM)



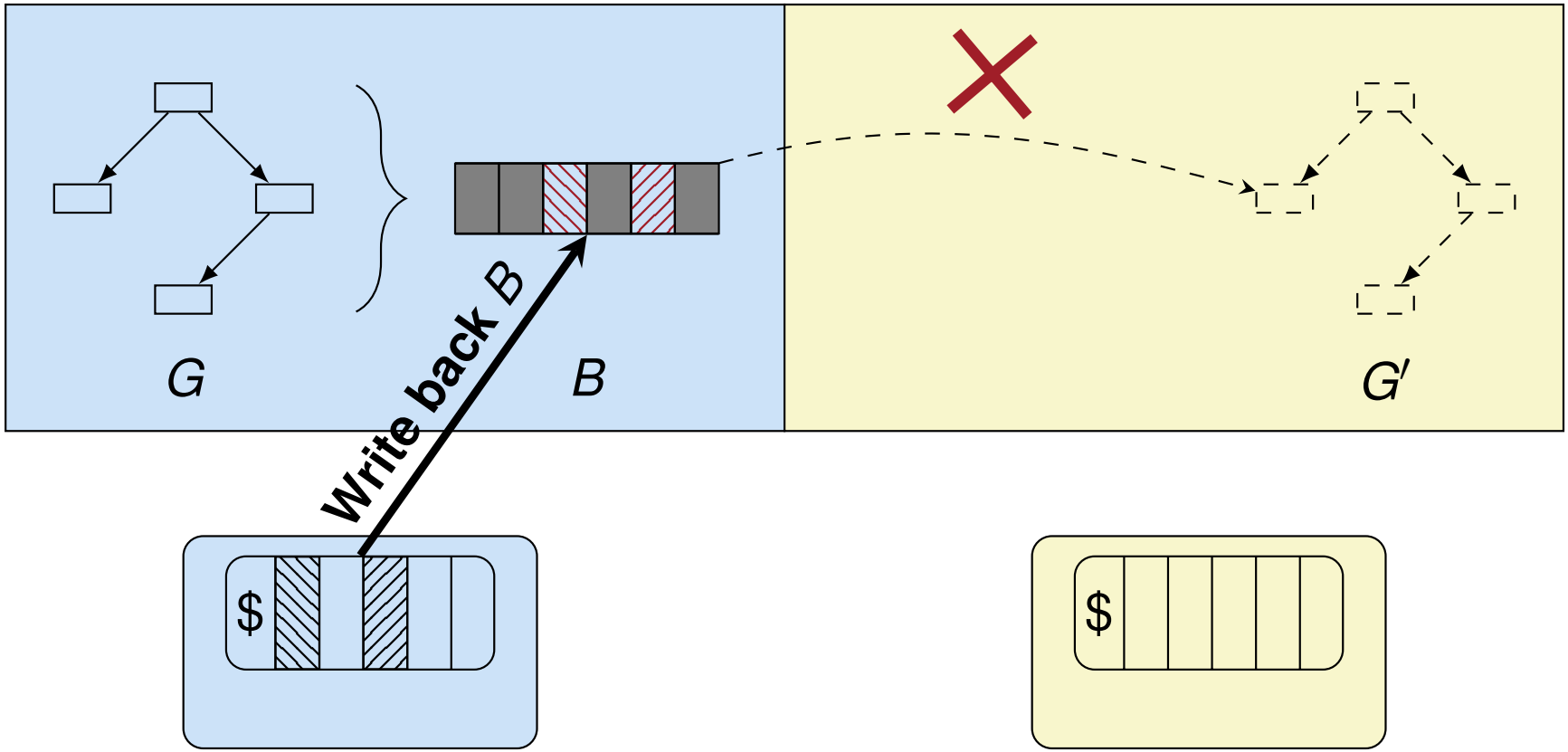
1. Serialize *G* to *B*

Message Passing via Shared Memory (MP-SHM)



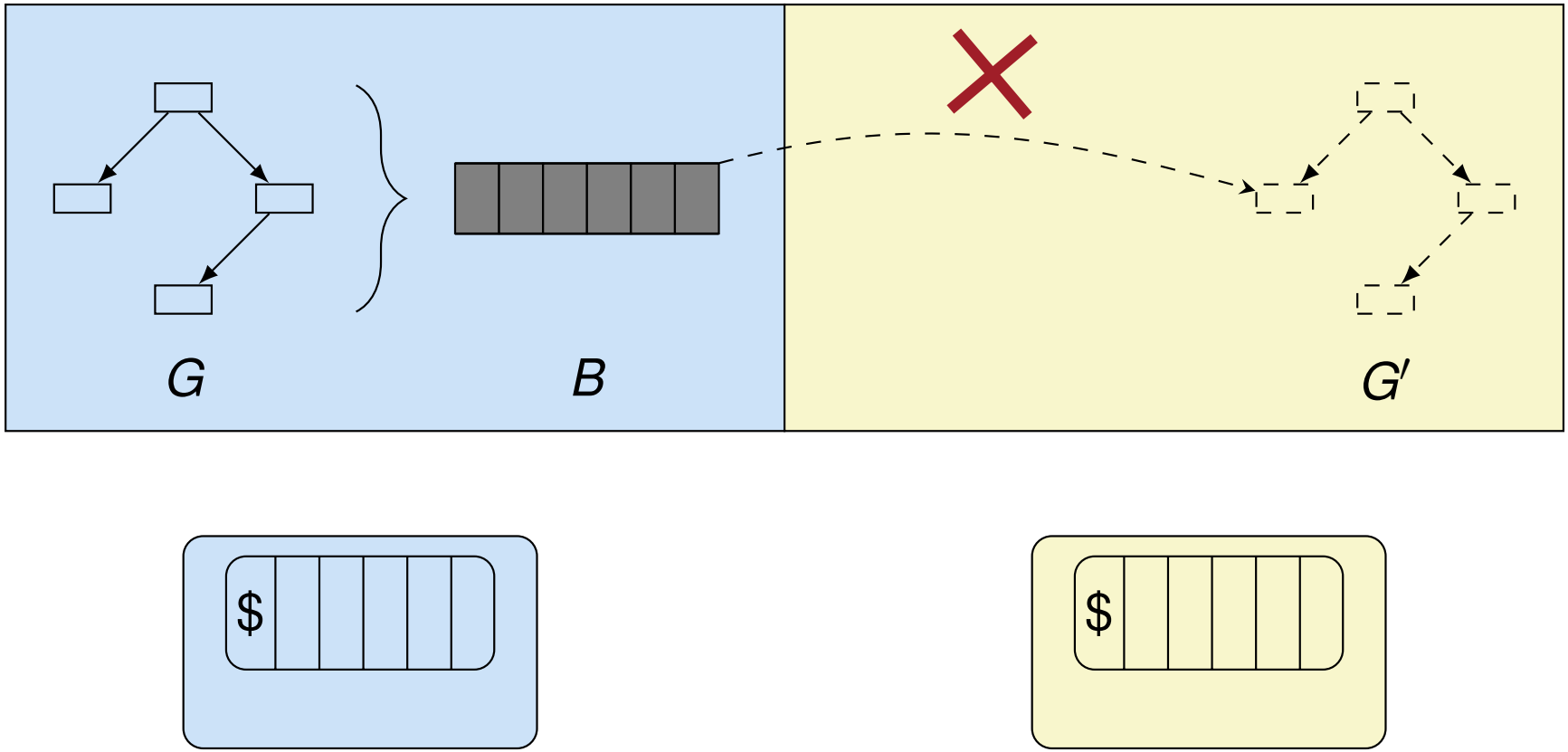
1. Serialize *G* to *B*

Message Passing via Shared Memory (MP-SHM)



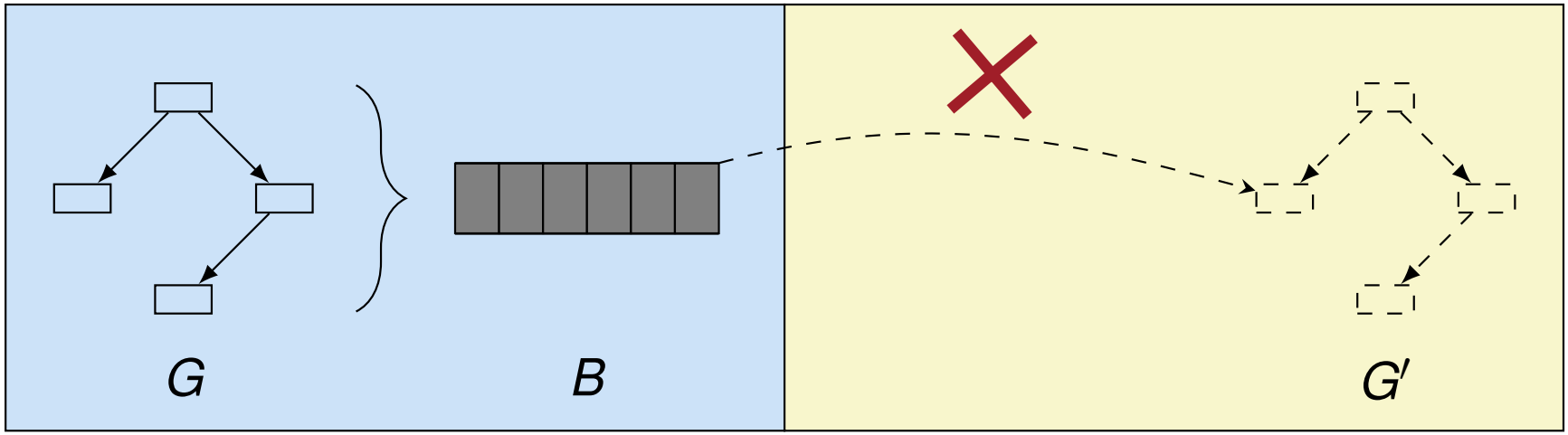
1. Serialize *G* to *B* and write back *B*

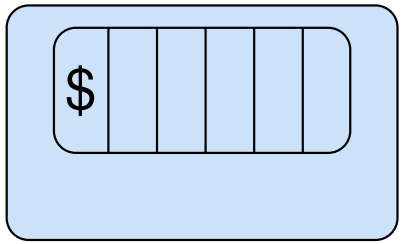
Message Passing via Shared Memory (MP-SHM)

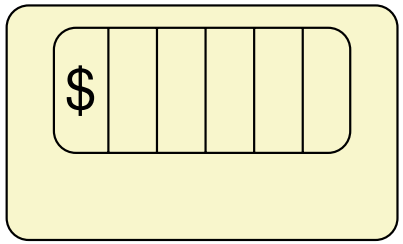


1. Serialize *G* to *B* and write back *B*

Message Passing via Shared Memory (MP-SHM)

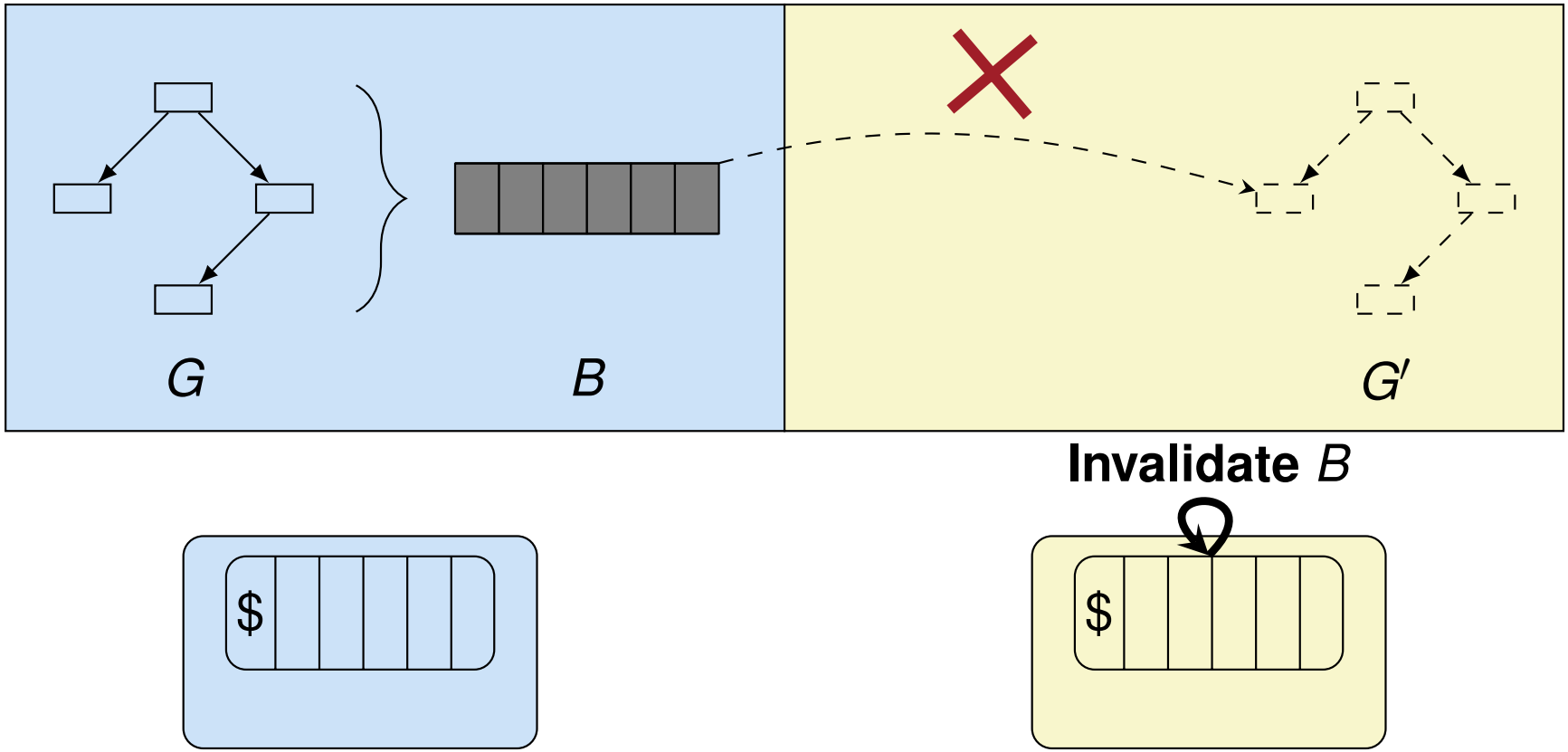






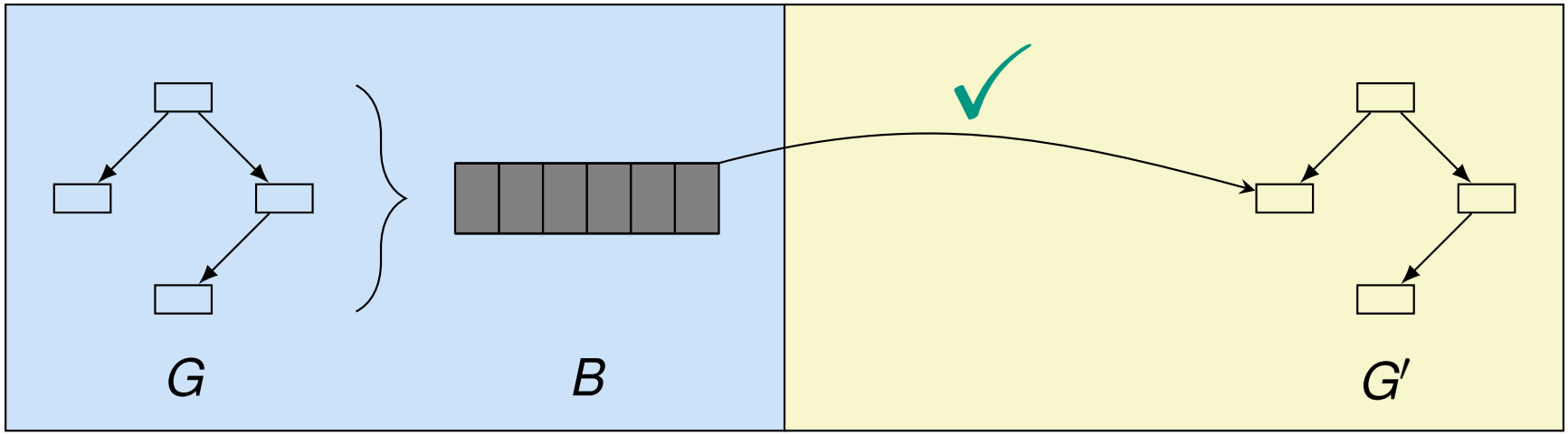
1. Serialize *G* to *B* and write back *B*
2. Notify yellow core with *B*'s address & size

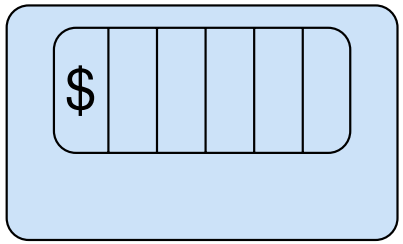
Message Passing via Shared Memory (MP-SHM)

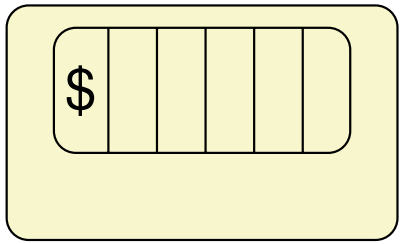


1. Serialize *G* to *B* and write back *B*
2. Notify yellow core with *B*'s address & size
3. Invalidate *B*

Message Passing via Shared Memory (MP-SHM)

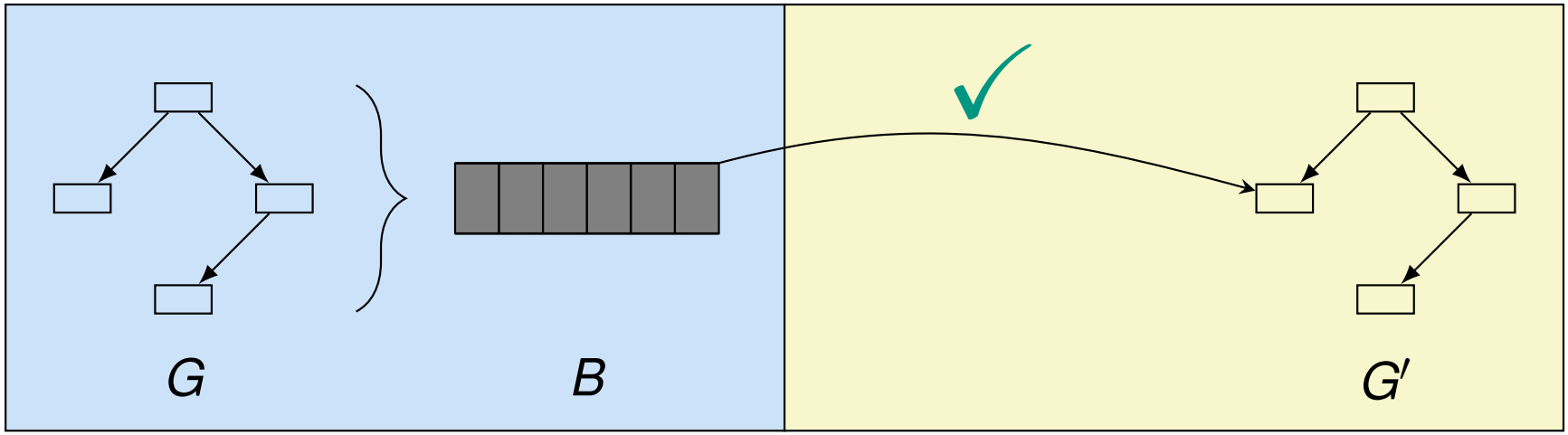


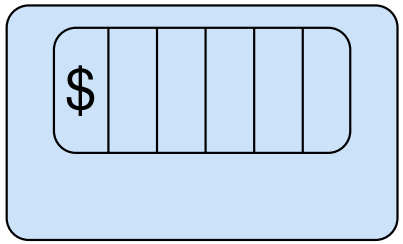


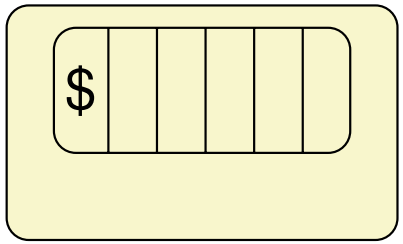


1. Serialize *G* to *B* and write back *B*
2. Notify yellow core with *B*'s address & size
3. Invalidate *B* and deserialize *G′* from *B*

Message Passing via Shared Memory (MP-SHM)



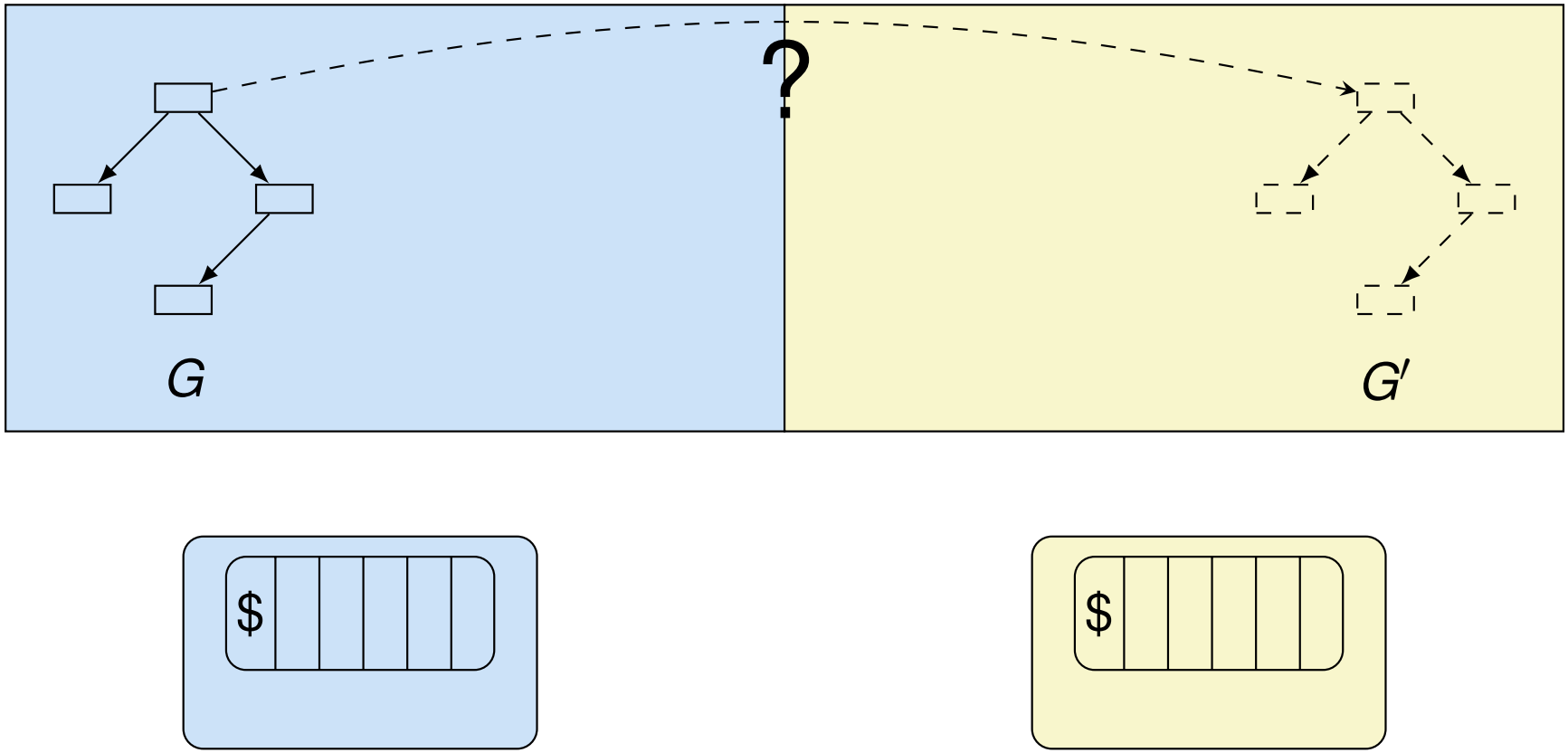




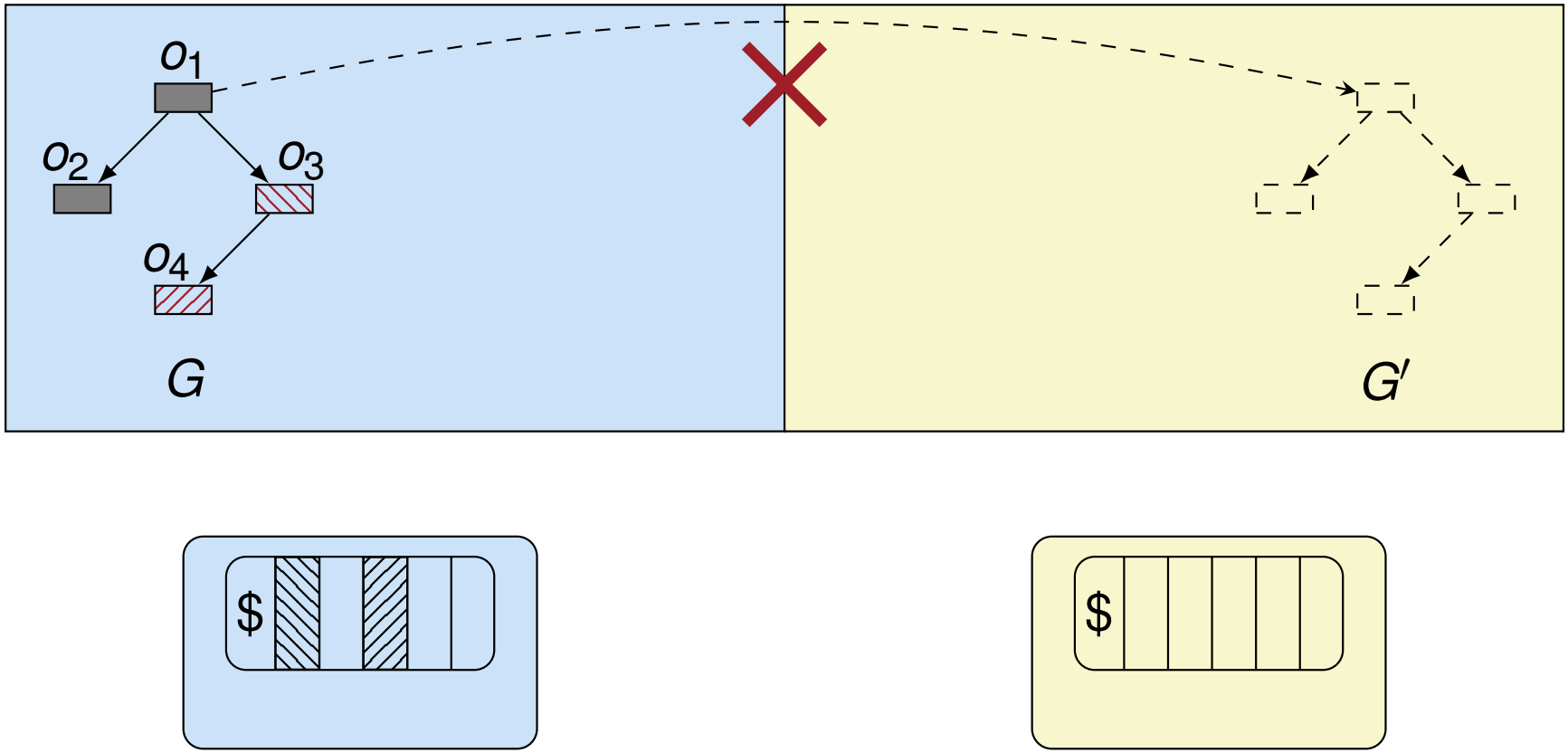
1. Serialize *G* to *B* and write back *B*
2. Notify yellow core with *B*'s address & size
3. Invalidate *B* and deserialize *G′* from *B*

\+ Just one buffer copy
– Serialization overhead
– *B* pollutes cache

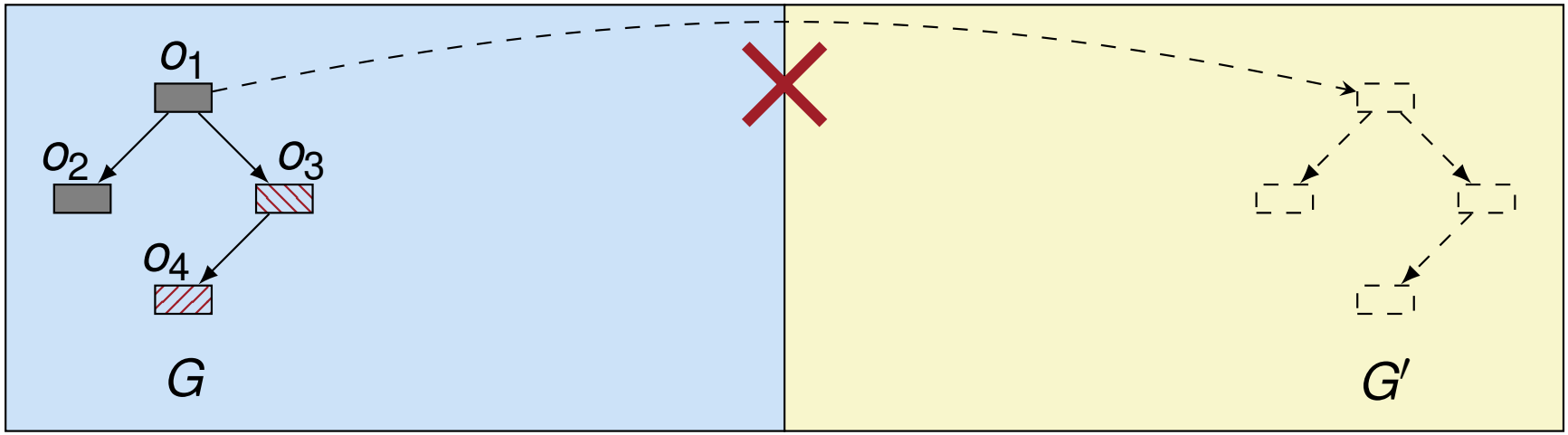
Cloning (CLONE)

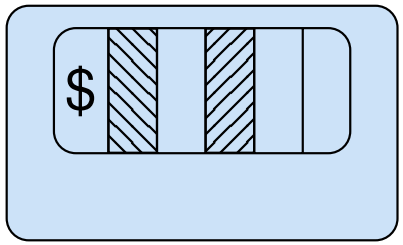


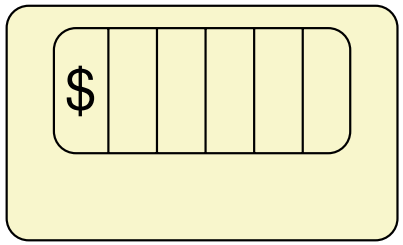
Cloning (CLONE)



Cloning (CLONE)

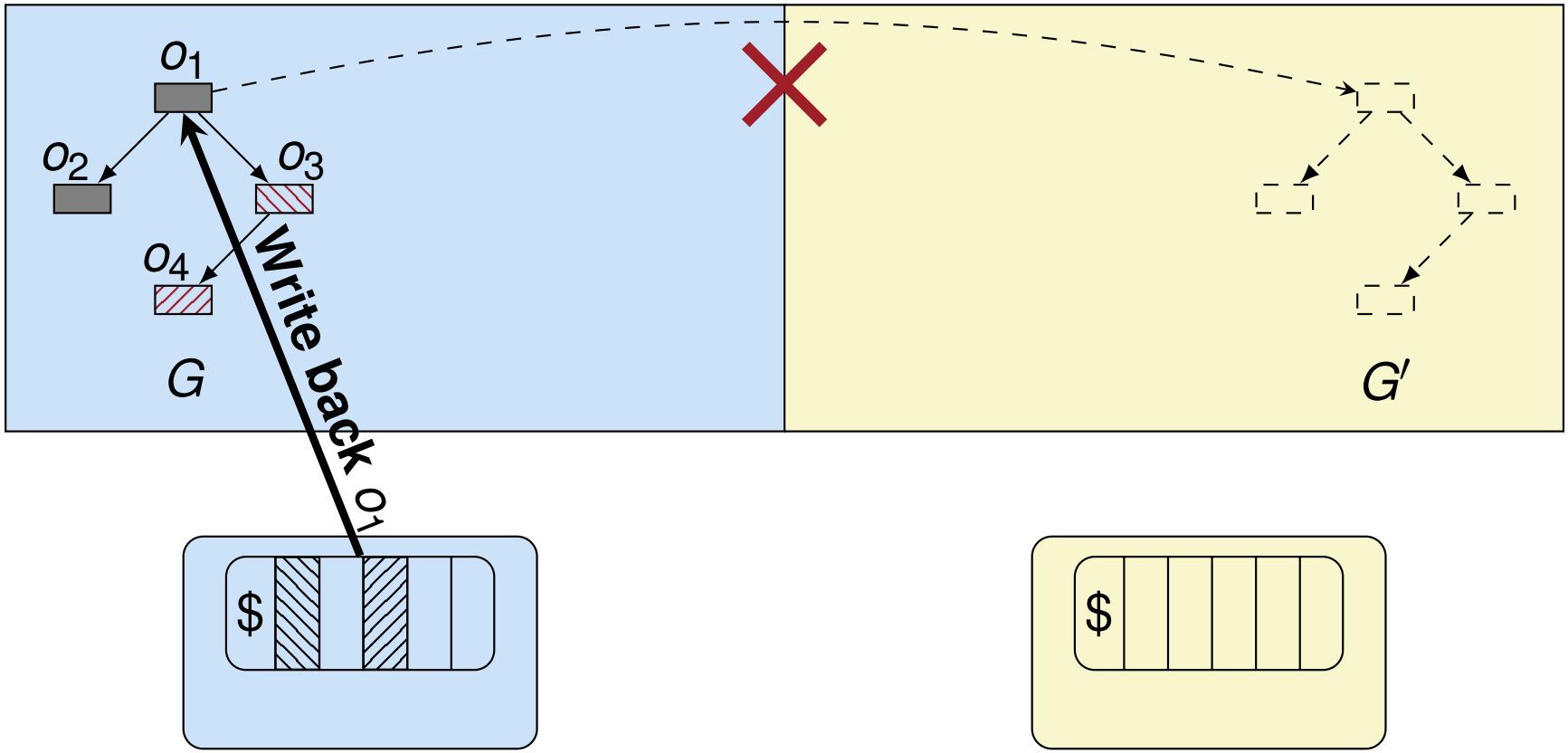






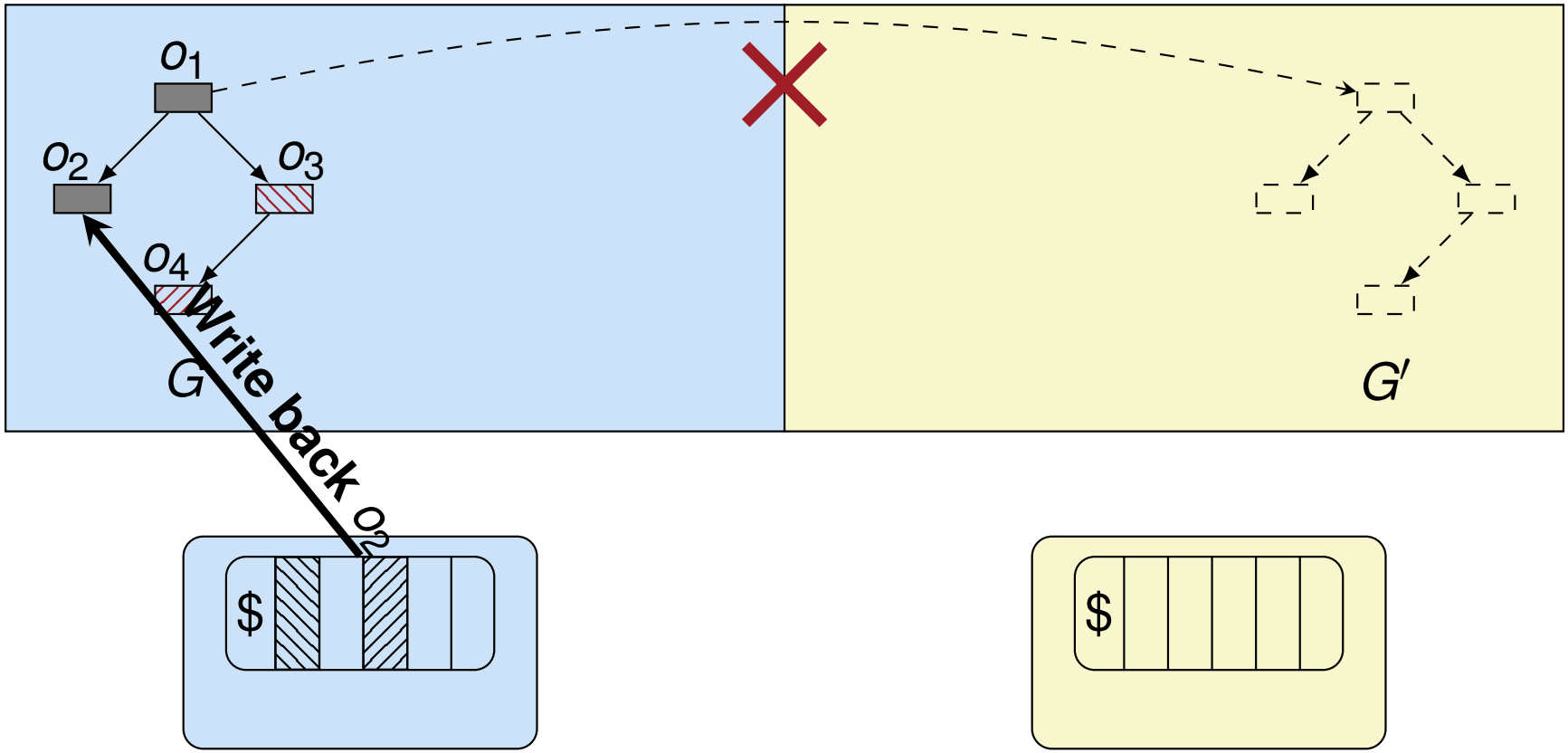
1. Traverse G and write back o_i

Cloning (CLONE)



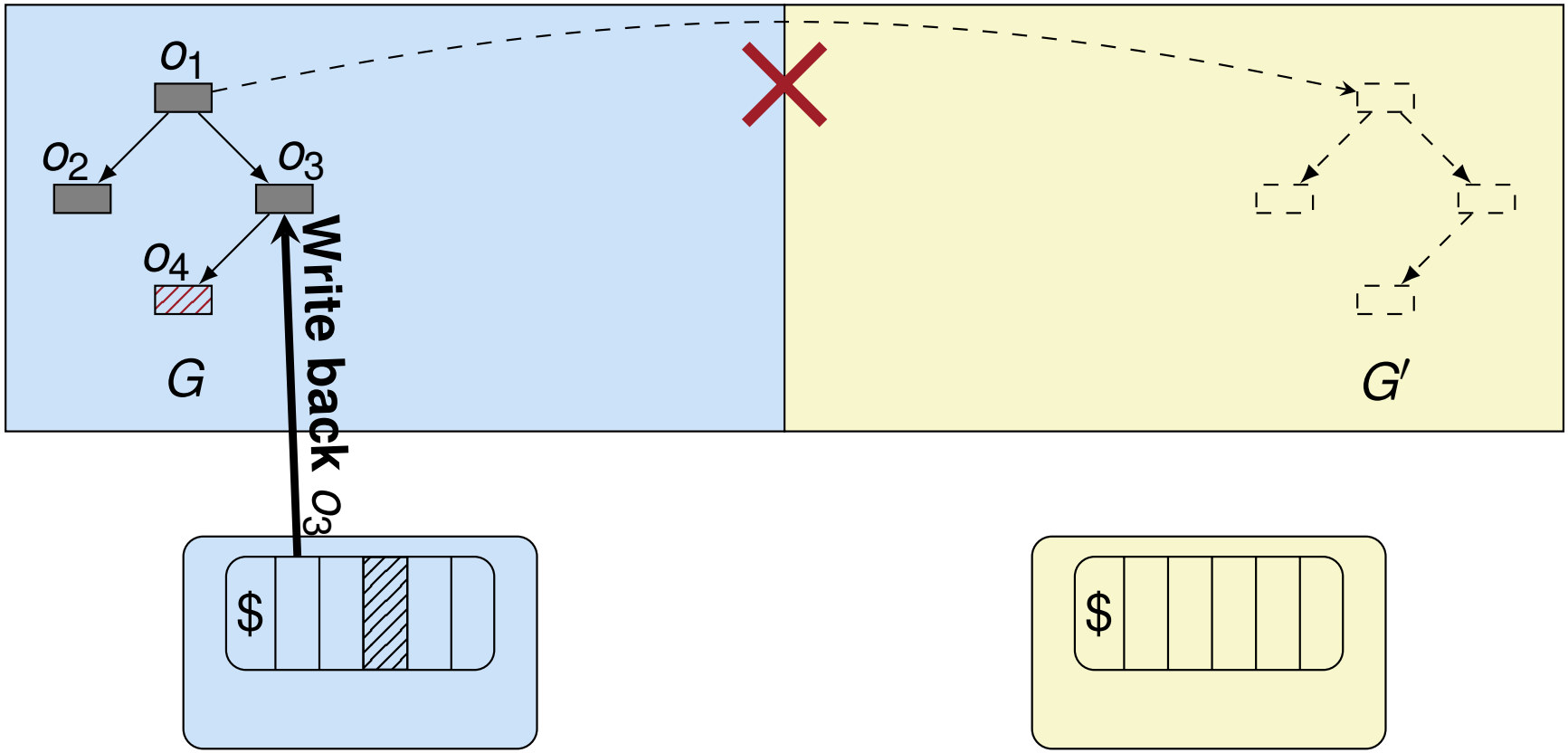
1. Traverse G and write back o_i

Cloning (CLONE)



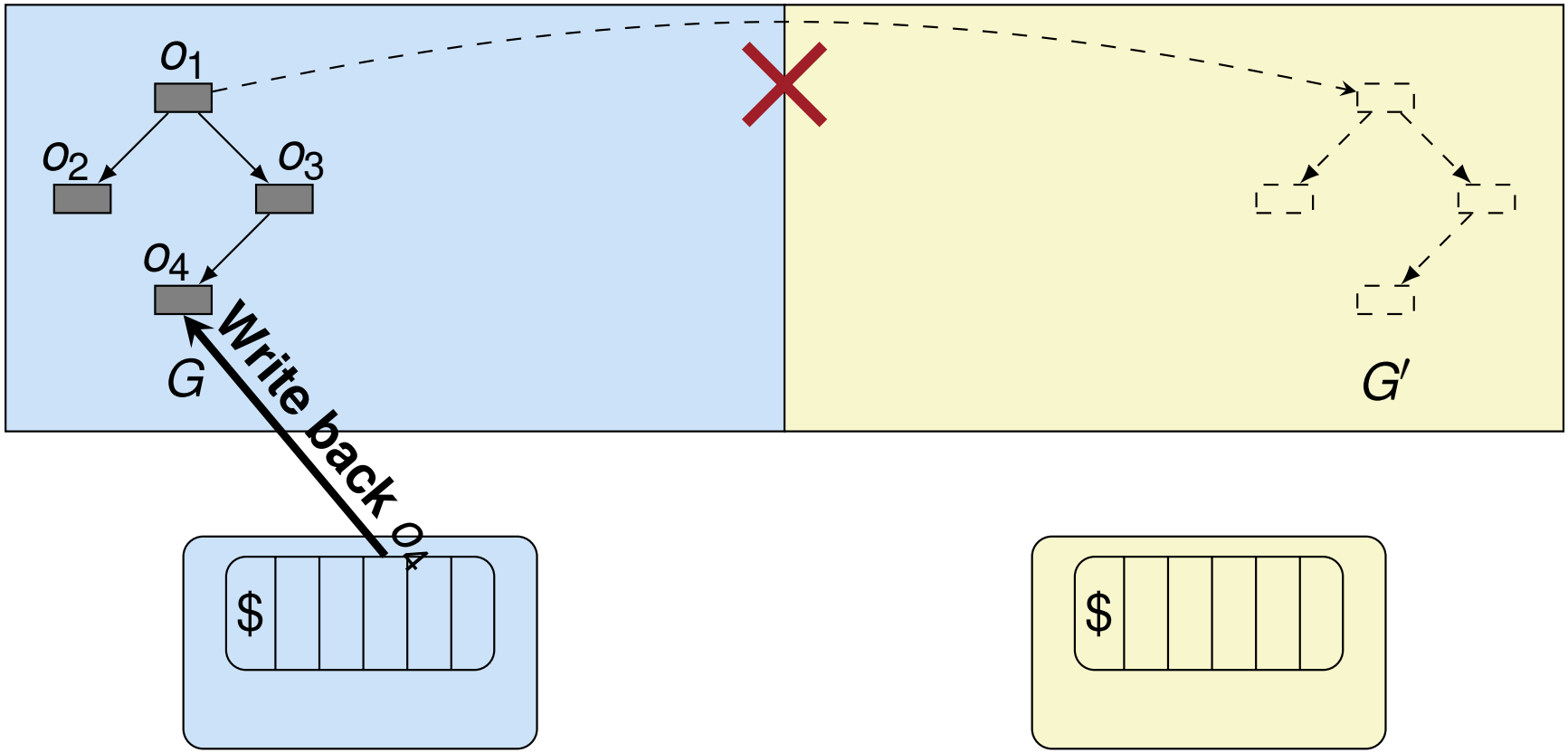
1. Traverse G and write back o_i

Cloning (CLONE)



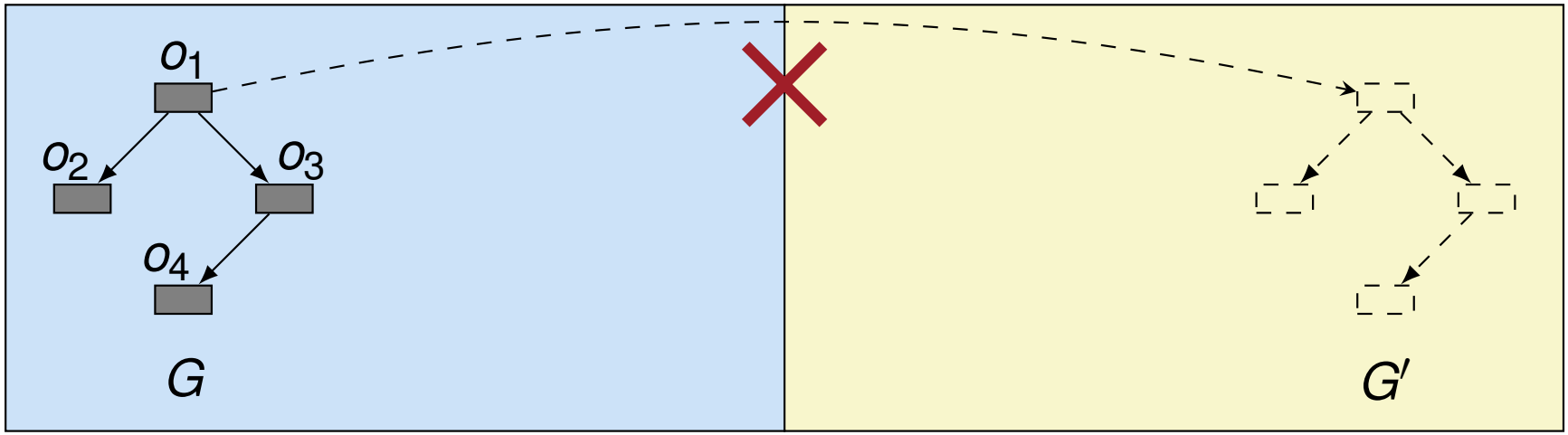
1. Traverse G and write back o_i

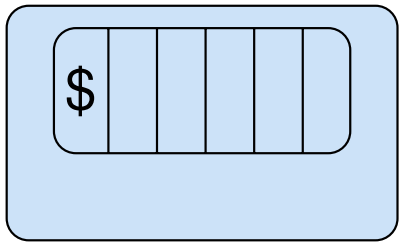
Cloning (CLONE)

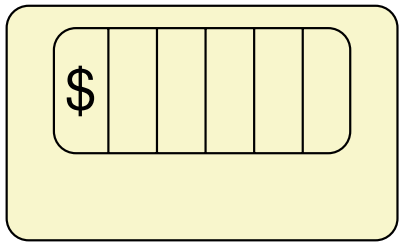


1. Traverse G and write back o_i

Cloning (CLONE)

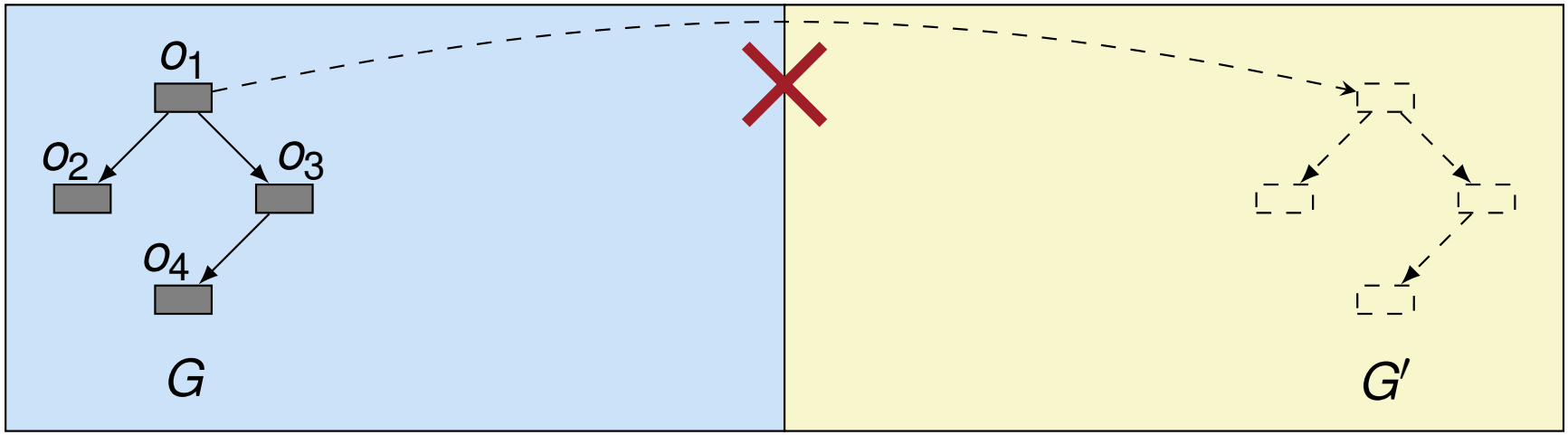


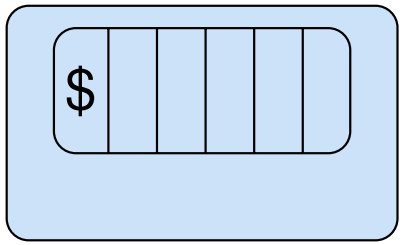


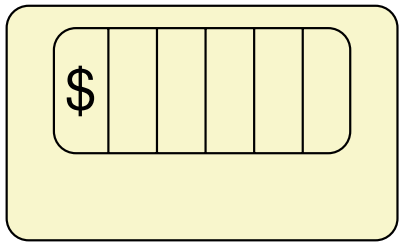


1. Traverse G and write back o_i

Cloning (CLONE)

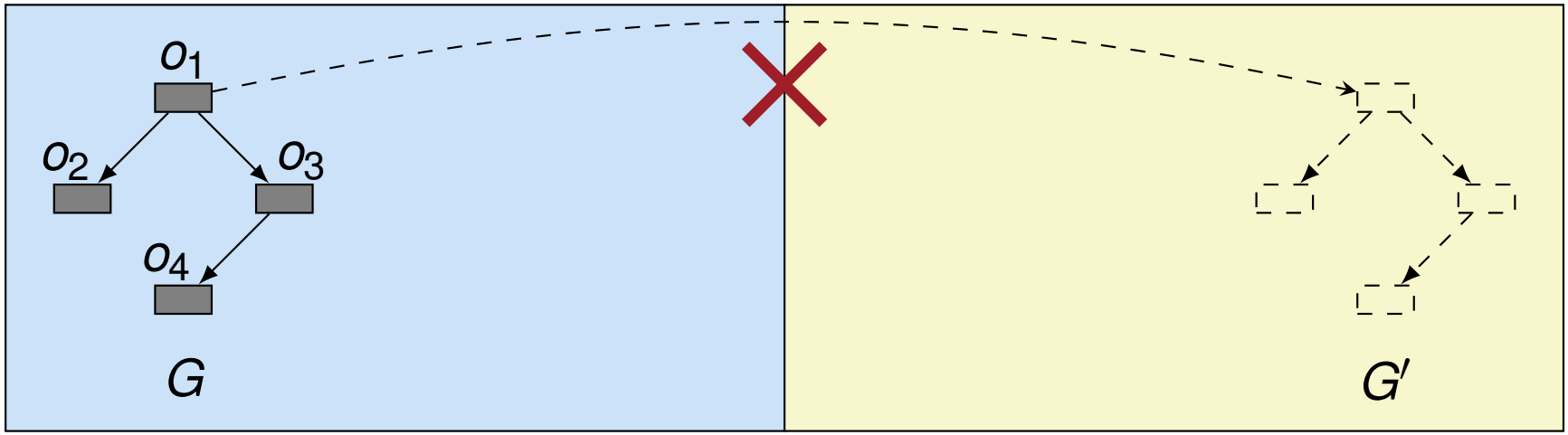


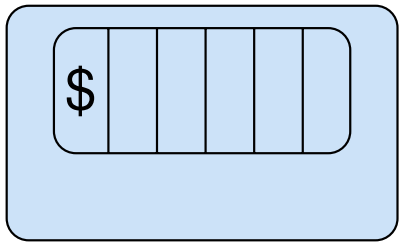


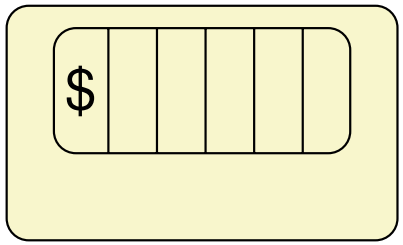


1. Traverse G and write back o_i
2. Notify yellow core with G's root

Cloning (CLONE)

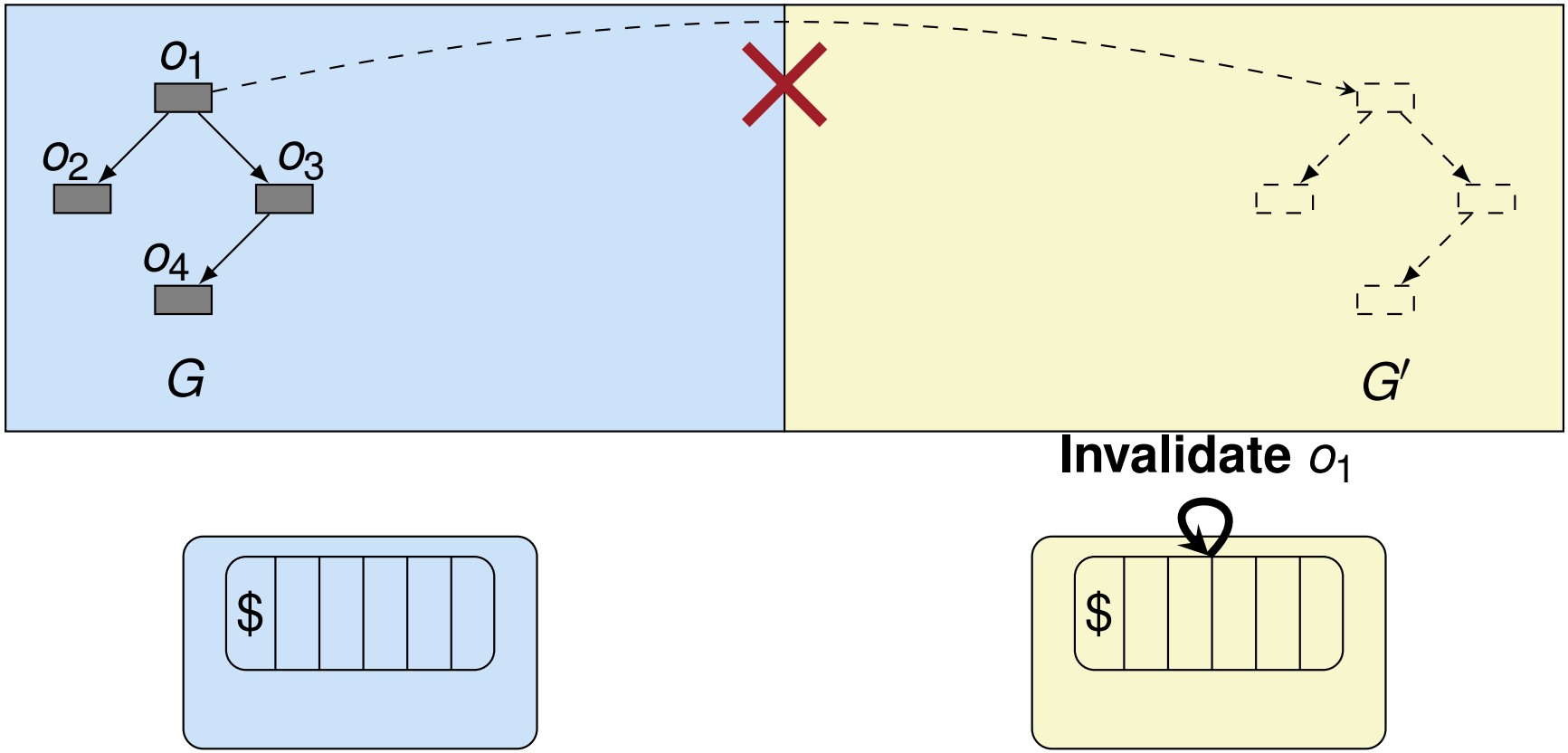






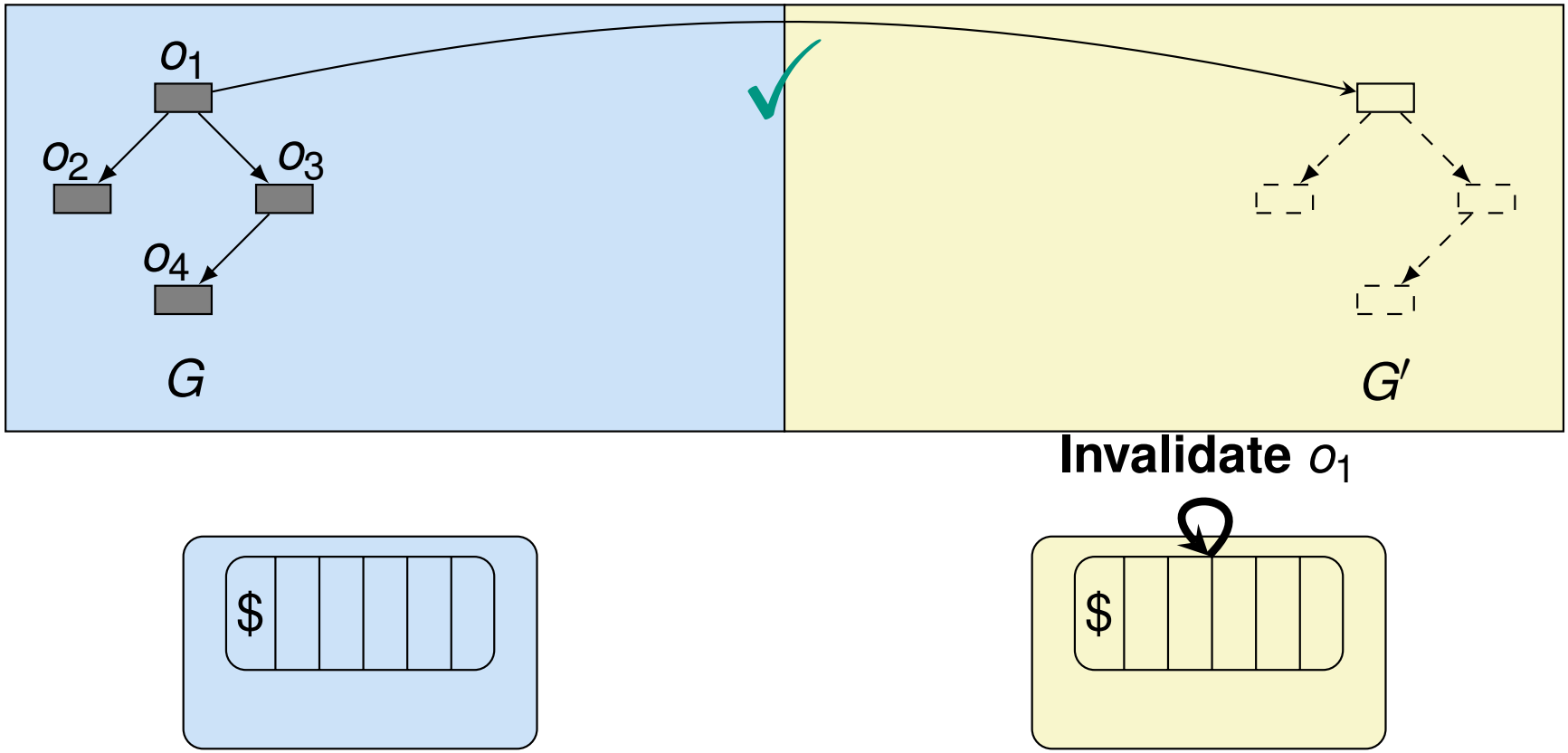
1. Traverse G and write back o_i
2. Notify yellow core with G's root
3. Traverse G & clone objects

Cloning (CLONE)



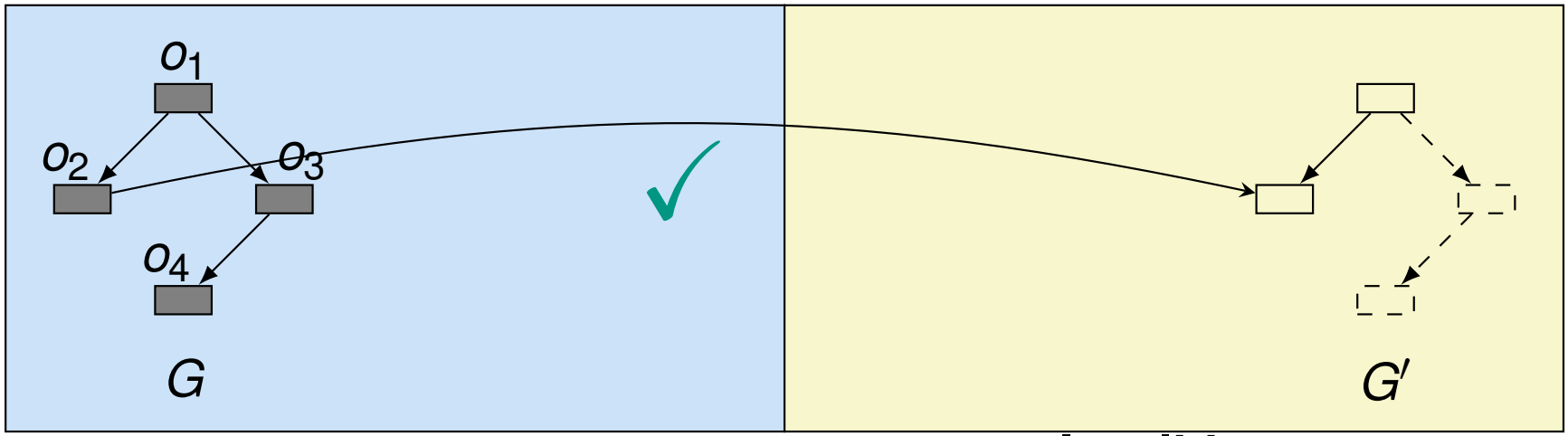
1. Traverse *G* and write back o_i
2. Notify yellow core with *G*'s root
3. Traverse *G*, invalidate & clone objects

Cloning (CLONE)



1. Traverse *G* and write back o_i
2. Notify yellow core with *G*'s root
3. Traverse *G*, invalidate & clone objects

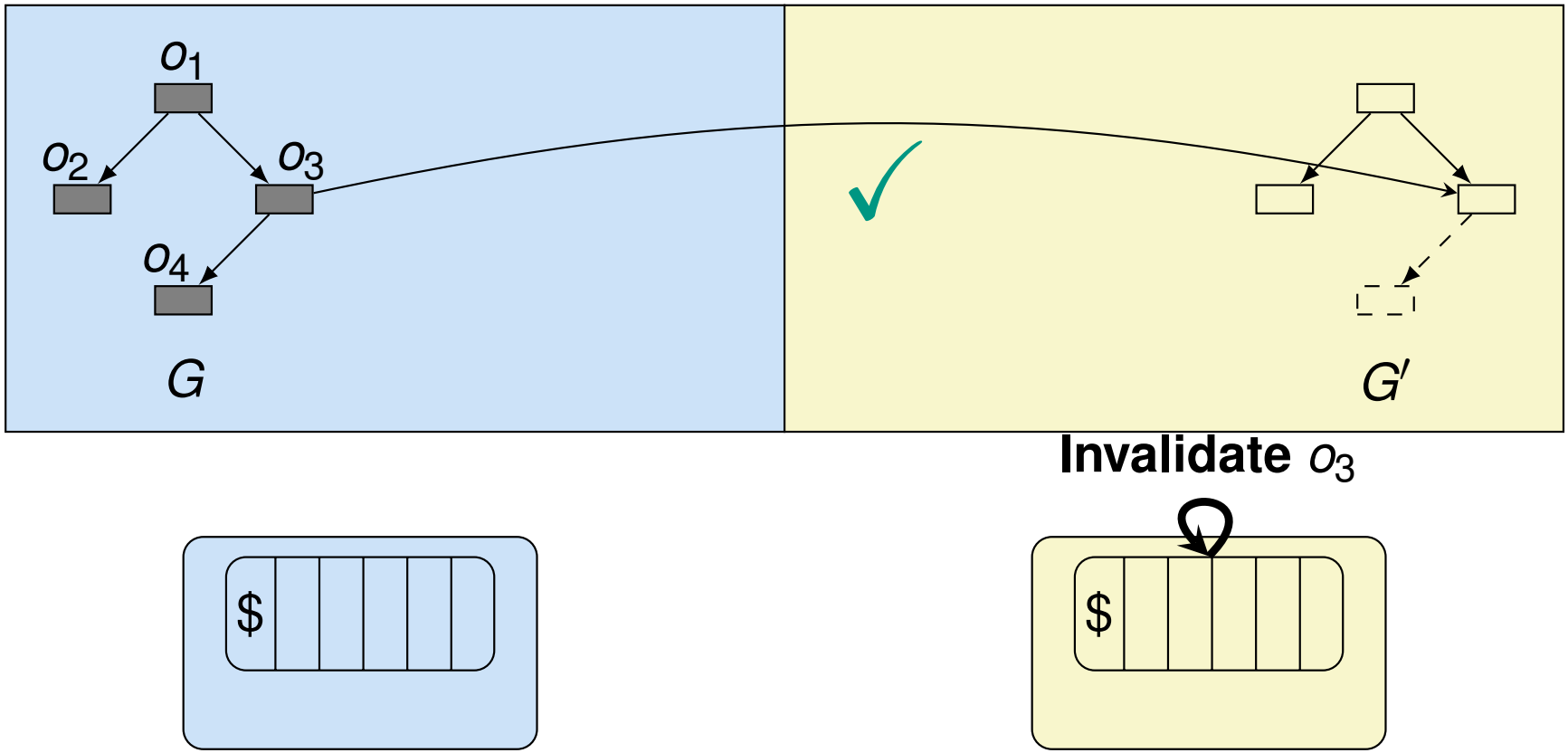
Cloning (CLONE)



Invalidate o_2

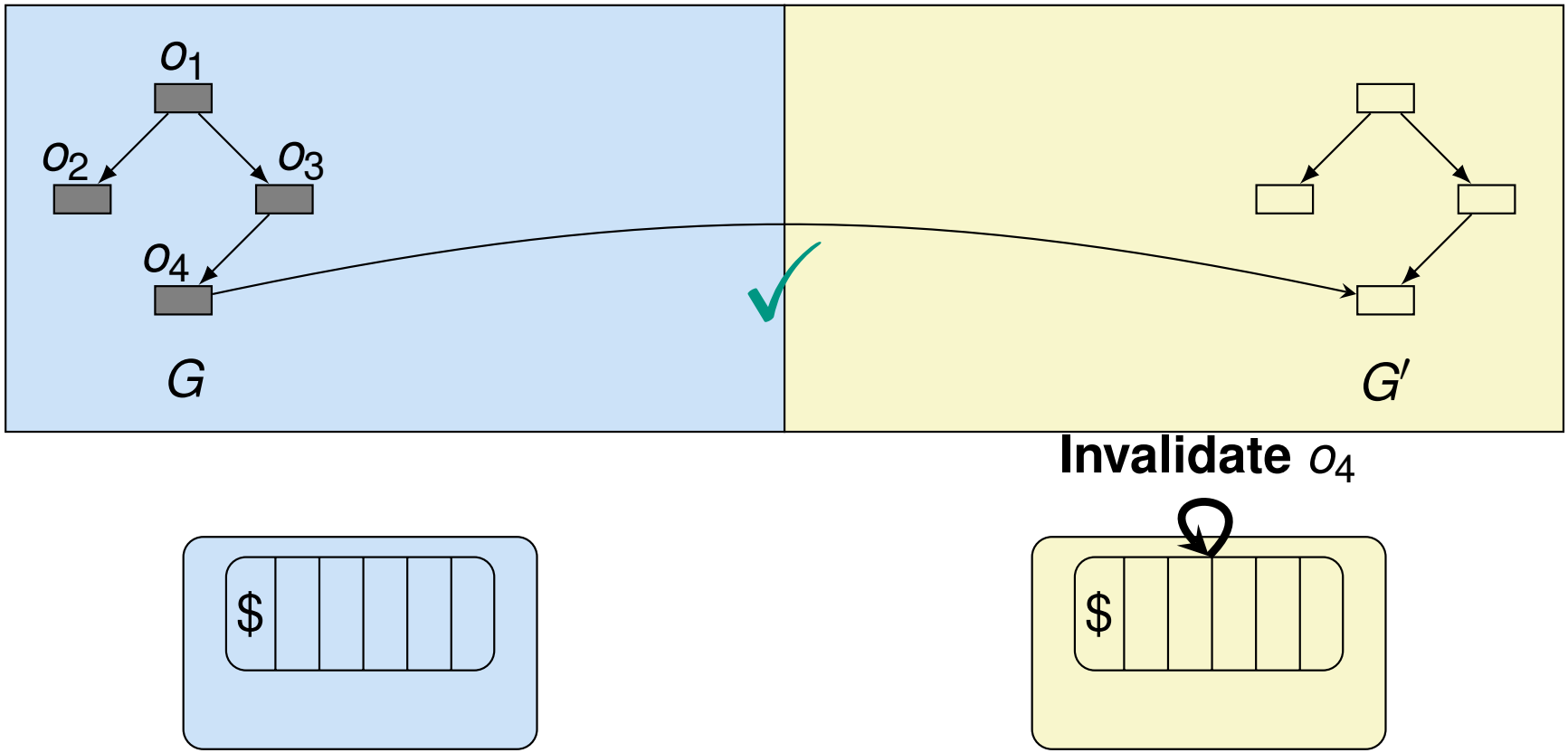
1. Traverse *G* and write back o_i
2. Notify yellow core with *G*'s root
3. Traverse *G*, invalidate & clone objects

Cloning (CLONE)



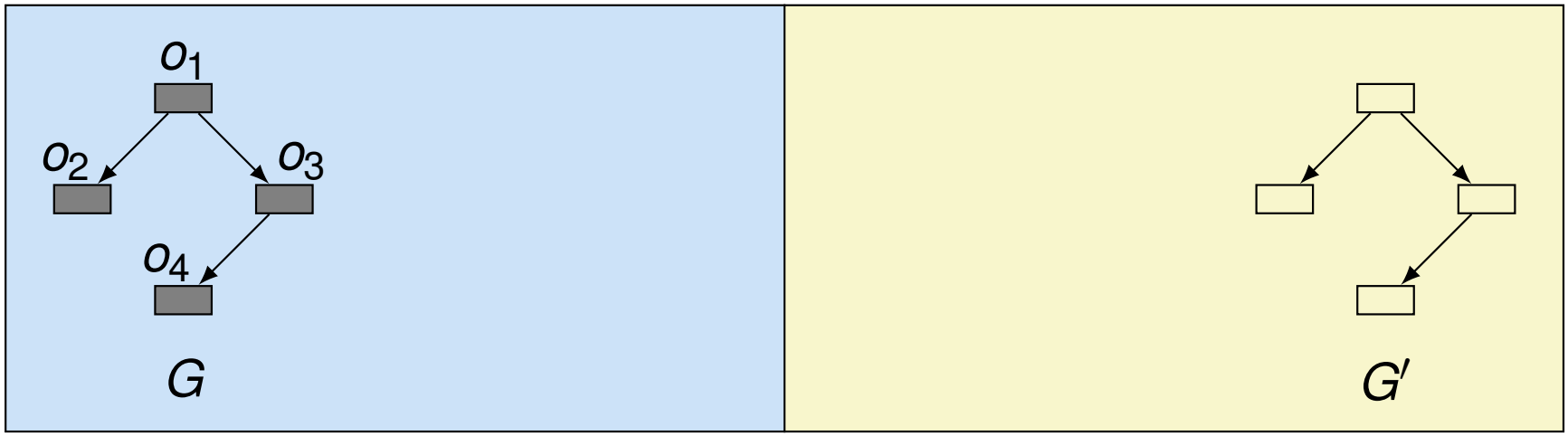
1. Traverse *G* and write back o_i
2. Notify yellow core with *G*'s root
3. Traverse *G*, invalidate & clone objects

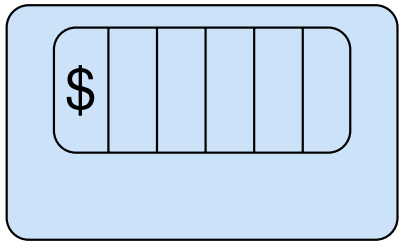
Cloning (CLONE)

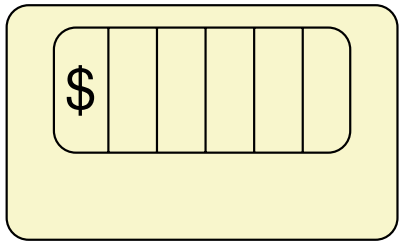


1. Traverse G and write back o_i
2. Notify yellow core with G's root
3. Traverse G, invalidate & clone objects

Cloning (CLONE)



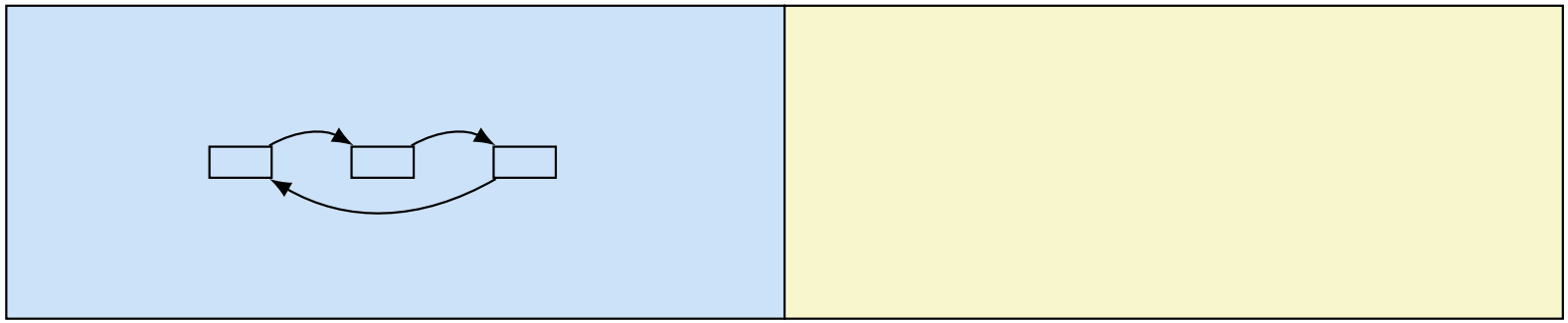




1. Traverse G and write back o_i
2. Notify yellow core with G's root
3. Traverse G, invalidate & clone objects

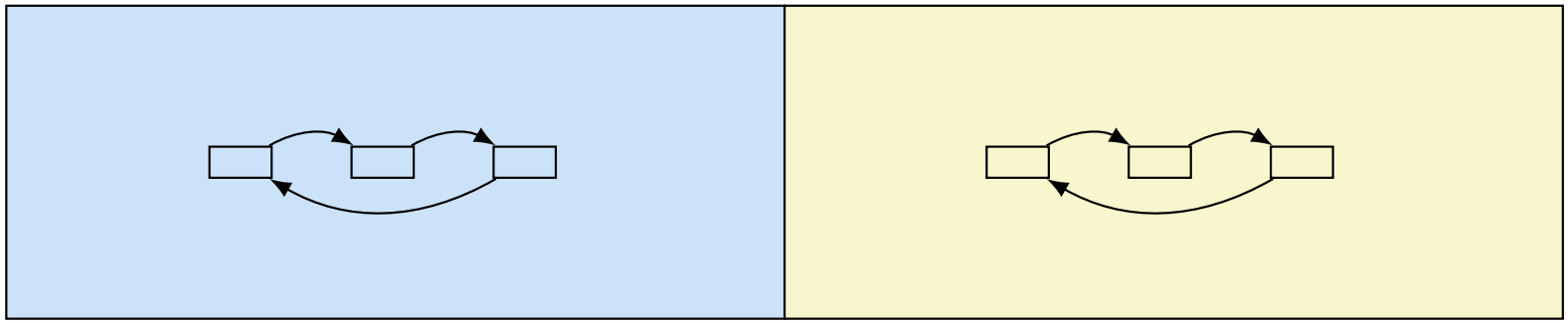
\+ No serialization
\+ No temporary buffer
\+ More cache-friendly

Implementation for PGAS Languages



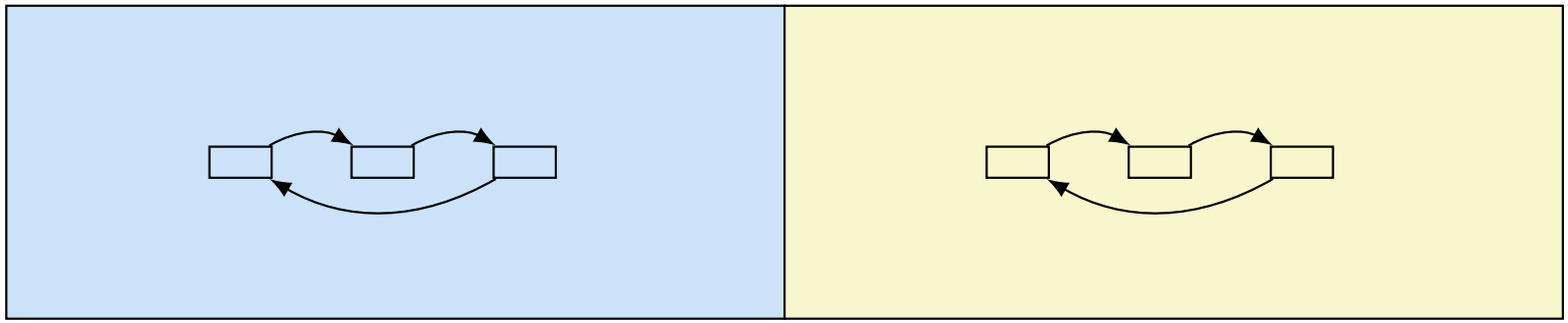
```
list = new LinkedList;
remote_op(list);
```

Implementation for PGAS Languages



```
list = new LinkedList;
remote_op(list);
```

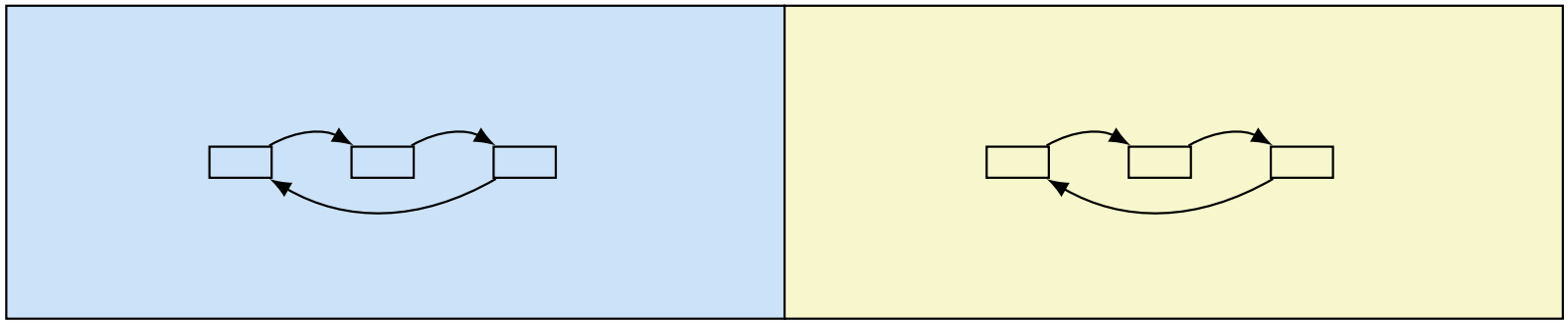
Implementation for PGAS Languages



```
list = new LinkedList;
remote_op(list);
```

- Compiler has full view of types *and* controls data transfers

Implementation for PGAS Languages



```
list = new LinkedList;
remote_op(list);
```

- Compiler has full view of types *and* controls data transfers

⇒ PGAS languages enable **fully-automatic compiler-based** implementation of cloning

  - Compiler generates type-specific writeback and invalidate functions
  - No need to modify existing programs

Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** $[S, E)$
- Status quo: operate on individual cache lines, `invalidate(addr)`

Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** $[S, E)$
- Status quo: operate on individual cache lines, `invalidate(addr)`

$\Rightarrow$ Software iterates over all relevant addresses

```
for x = S to E step CACHE_LINE_SIZE:
  invalidate(x)
```

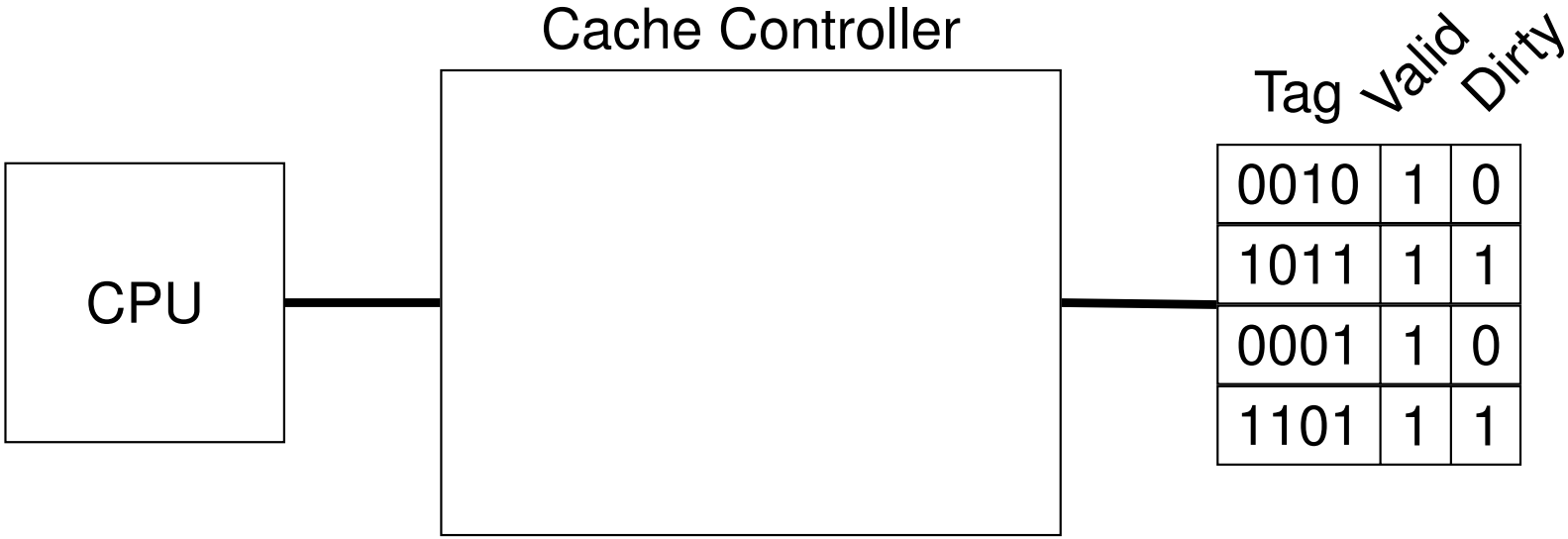
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** $[S, E)$
- Status quo: operate on individual cache lines, `invalidate(addr)`

⇒ Software iterates over all relevant addresses

```
for x = S to E step CACHE_LINE_SIZE:
  invalidate(x)
```

- Why not support this in hardware?



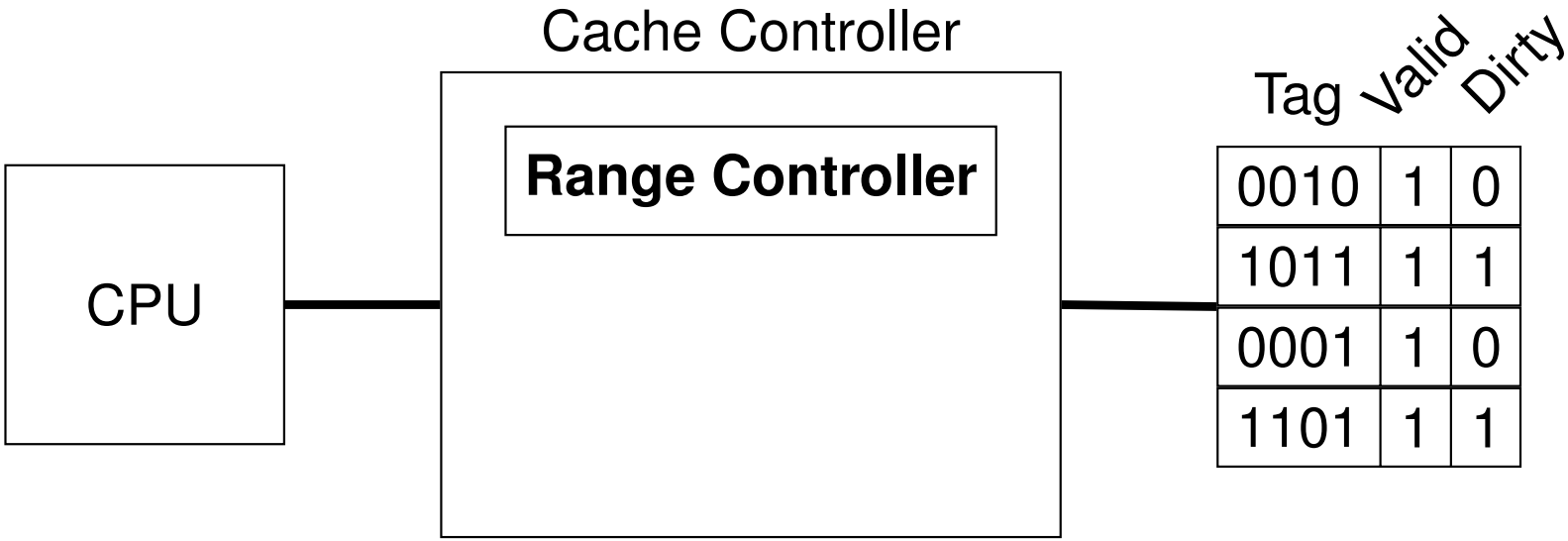
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** $[S, E)$
- Status quo: operate on individual cache lines, `invalidate(addr)`

⇒ Software iterates over all relevant addresses

```
for x = S to E step CACHE_LINE_SIZE:
  invalidate(x)
```

- Why not support this in hardware?



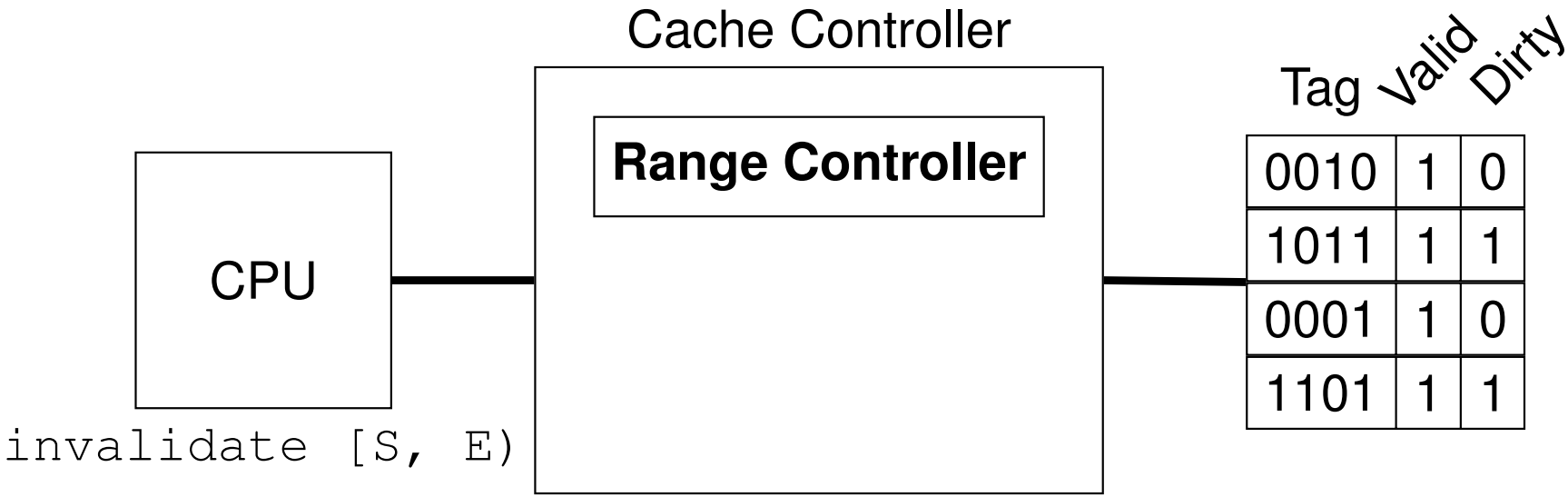
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** $[S, E)$
- Status quo: operate on individual cache lines, `invalidate(addr)`

⇒ Software iterates over all relevant addresses

```
for x = S to E step CACHE_LINE_SIZE:
  invalidate(x)
```

- Why not support this in hardware?



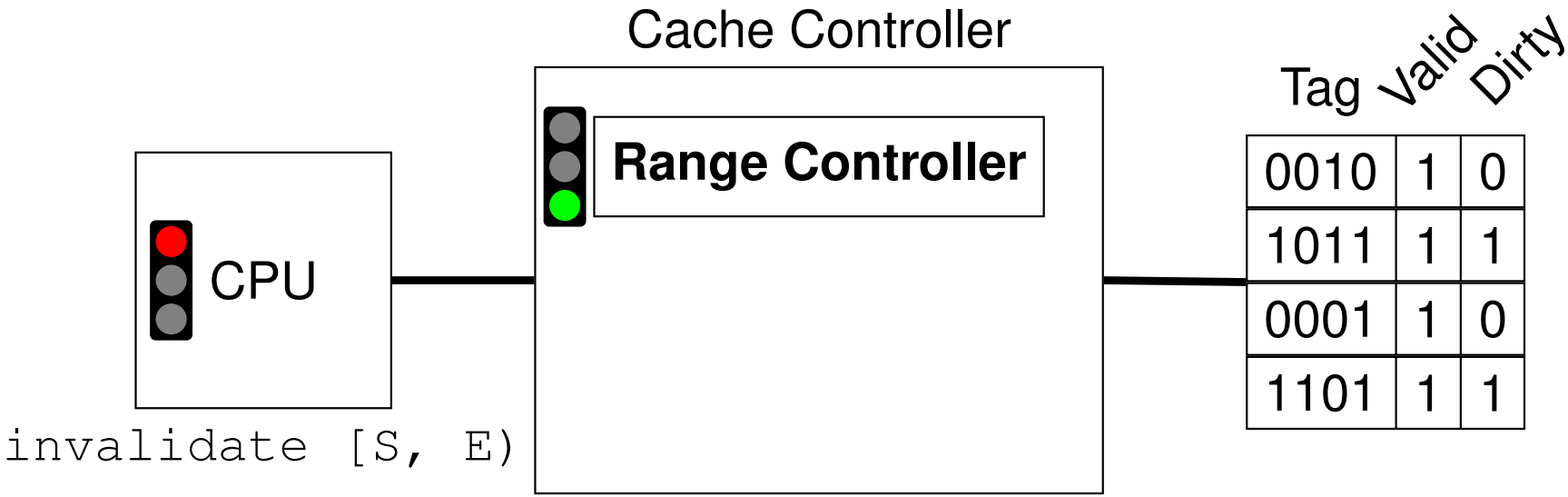
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** $[S, E)$
- Status quo: operate on individual cache lines, `invalidate(addr)`

⇒ Software iterates over all relevant addresses

```
for x = S to E step CACHE_LINE_SIZE:
  invalidate(x)
```

- Why not support this in hardware?



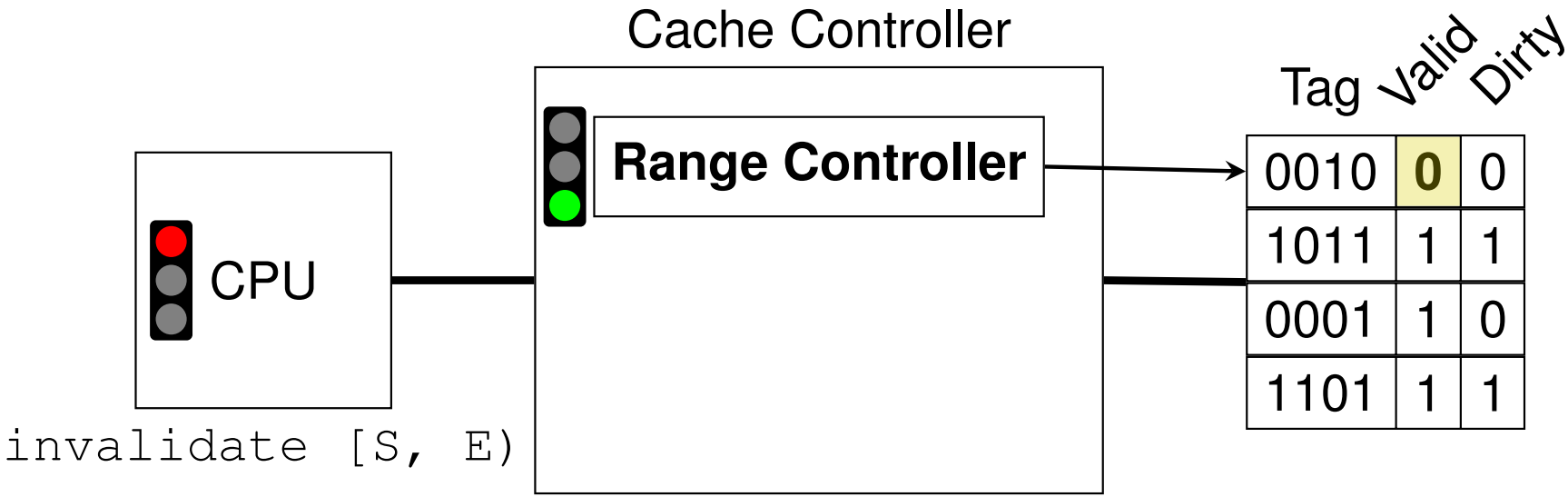
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** $[S, E)$
- Status quo: operate on individual cache lines, `invalidate(addr)`

⇒ Software iterates over all relevant addresses

```
for x = S to E step CACHE_LINE_SIZE:
  invalidate(x)
```

- Why not support this in hardware?



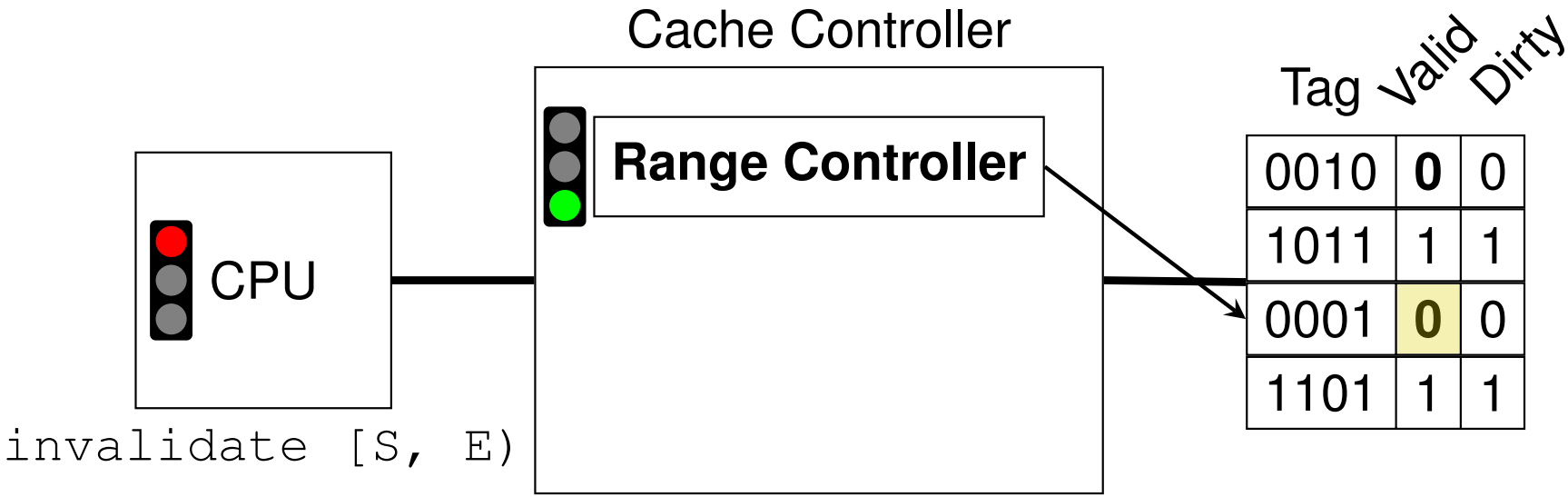
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** $[S, E)$
- Status quo: operate on individual cache lines, `invalidate(addr)`

⇒ Software iterates over all relevant addresses

```
for x = S to E step CACHE_LINE_SIZE:
  invalidate(x)
```

- Why not support this in hardware?



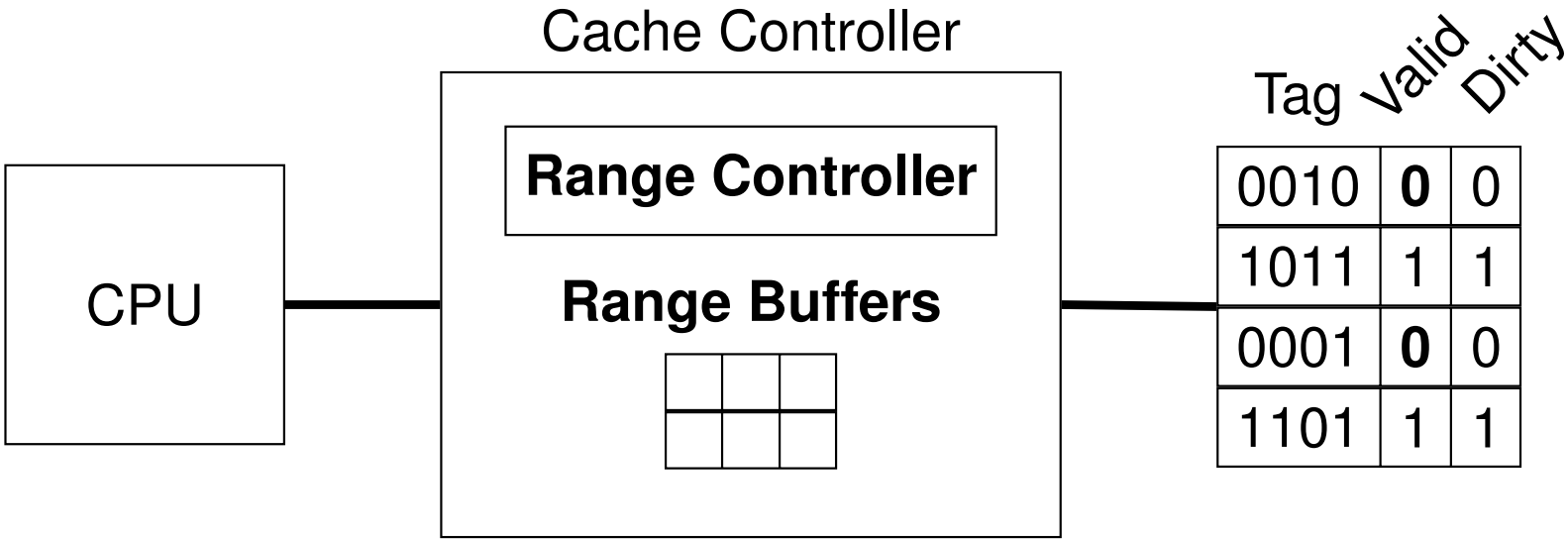
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** $[S, E)$
- Status quo: operate on individual cache lines, `invalidate(addr)`

⇒ Software iterates over all relevant addresses

```
for x = S to E step CACHE_LINE_SIZE:
  invalidate(x)
```

- Why not support this in hardware?



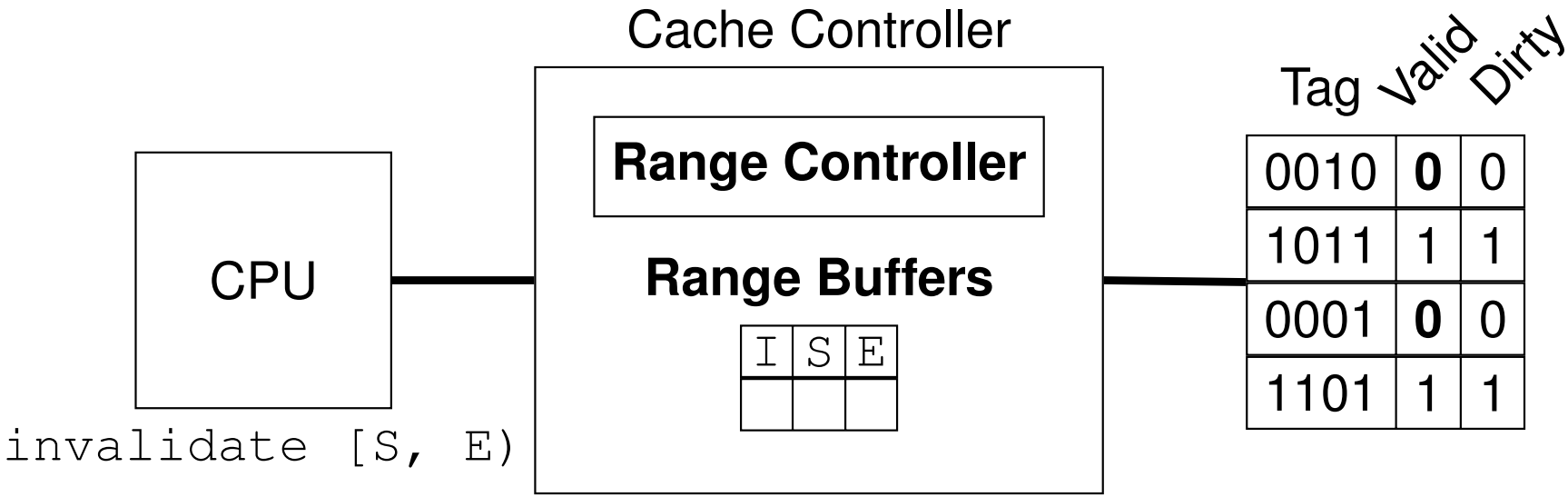
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** $[S, E)$
- Status quo: operate on individual cache lines, `invalidate(addr)`

⇒ Software iterates over all relevant addresses

```
for x = S to E step CACHE_LINE_SIZE:
  invalidate(x)
```

- Why not support this in hardware?



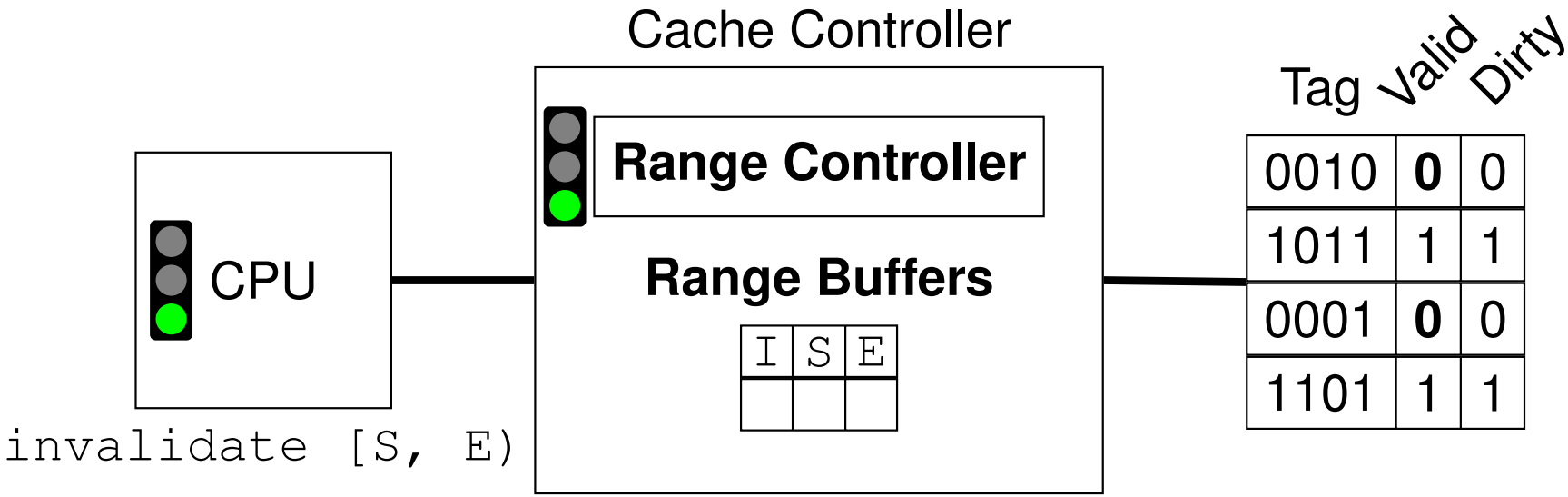
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** $[S, E)$
- Status quo: operate on individual cache lines, `invalidate(addr)`

⇒ Software iterates over all relevant addresses

```
for x = S to E step CACHE_LINE_SIZE:
  invalidate(x)
```

- Why not support this in hardware?



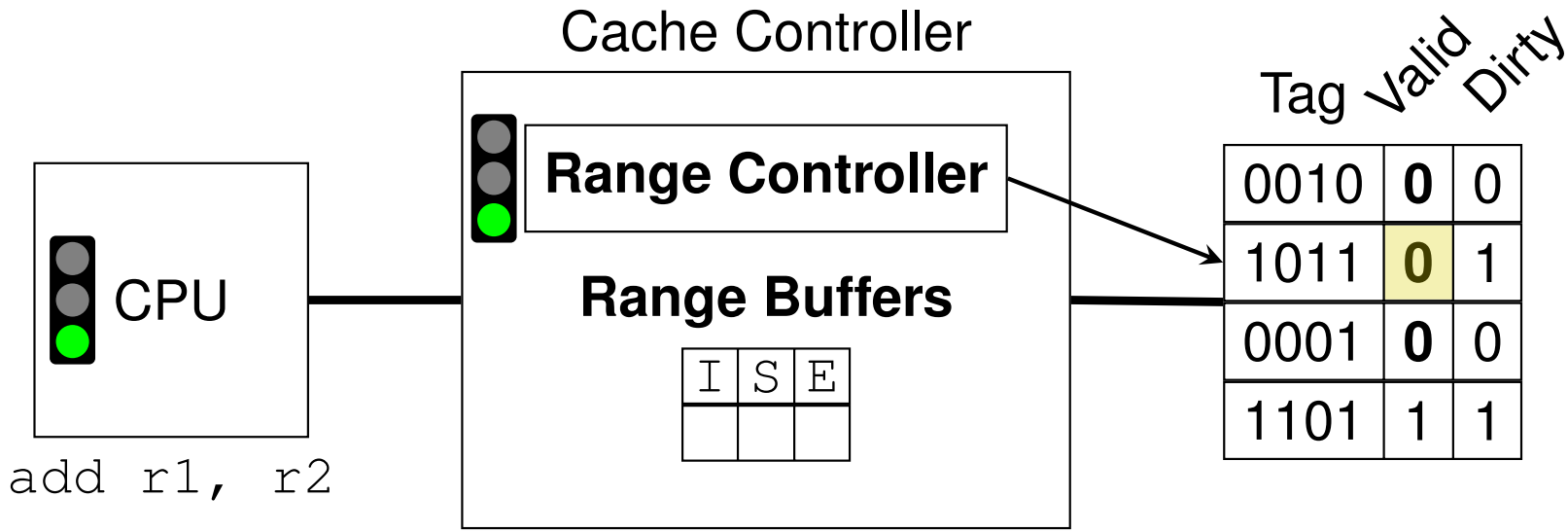
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** $[S, E)$
- Status quo: operate on individual cache lines, `invalidate(addr)`

⇒ Software iterates over all relevant addresses

```
for x = S to E step CACHE_LINE_SIZE:
  invalidate(x)
```

- Why not support this in hardware?



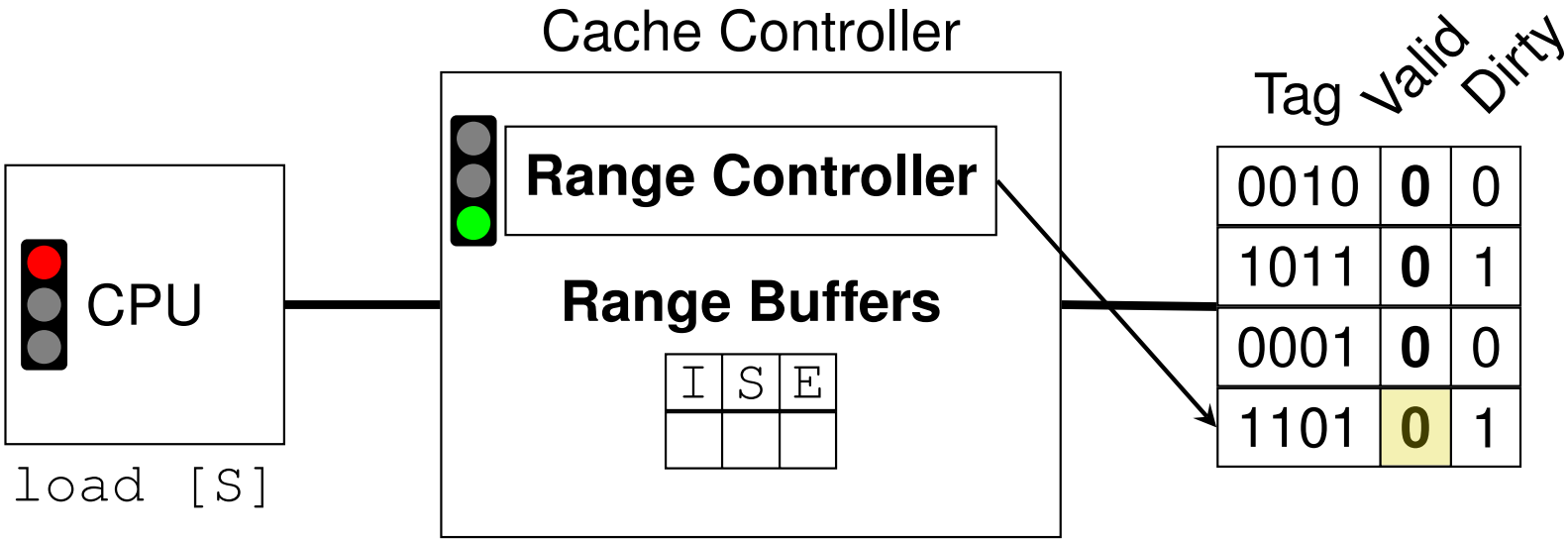
Hardware Extension: Range Operations

- Invalidation and write-back of **address ranges** $[S, E)$
- Status quo: operate on individual cache lines, `invalidate(addr)`

⇒ Software iterates over all relevant addresses

```
for x = S to E step CACHE_LINE_SIZE:
  invalidate(x)
```

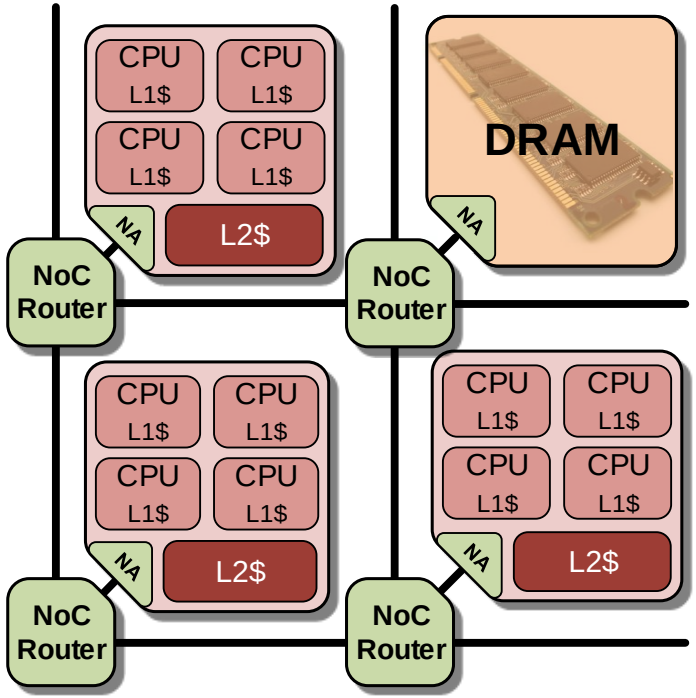
- Why not support this in hardware?



Evaluation Setup

Hardware

- FPGA prototype of non-cache-coherent many-core architecture
- 3 tiles, each 4 LEON3 cores
- 256 MiB shared DRAM
- Private L1$ per core, shared L2$ per tile
- No cache coherence between tiles
- No hardware-based range operations



Software

- Implemented cloning in compiler for PGAS language X10
- Input: X10 programs from IMSuite
  - 12 graph-based distributed algorithm kernels

Results

Benchmark	MP	MP-SHM	CLONE	$\frac{\text{CLONE}}{\text{MP}}$	$\frac{\text{CLONE}}{\text{MP-SHM}}$
BF	1.30	1.17	1.13	1.15×	1.03×
DST	9.35	7.94	7.35	1.27×	1.08×
BY	736.79	677.27	658.39	1.12×	1.03×
DR	83.22	82.13	80.42	1.03×	1.02×
DS	50.92	47.24	45.49	1.12×	1.04×
MIS	1.75	1.60	1.57	1.12×	1.02×
KC	27.10	25.86	25.84	1.05×	1.00×
DP	36.59	34.14	32.61	1.12×	1.05×
HS	43.86	34.81	34.00	1.29×	1.02×
LCR	14.24	11.92	11.88	1.20×	1.00×
MST	69.82	62.87	50.70	1.38×	1.24×
VC	1.60	1.30	1.26	1.27×	1.03×
Geomean				**1.17**×	**1.05**×

- Running times and speedups over serialization-based approaches

Results

Benchmark	MP	MP-SHM	CLONE	$\frac{\text{CLONE}}{\text{MP}}$	$\frac{\text{CLONE}}{\text{MP-SHM}}$
BF	1.30	1.17	1.13	1.15×	1.03×
DST	9.35	7.94	7.35	1.27×	1.08×
BY	736.79	677.27	658.39	1.12×	1.03×
DR	83.22	82.13	80.42	1.03×	1.02×
DS	50.92	47.24	45.49	1.12×	1.04×
MIS	1.75	1.60	1.57	1.12×	1.02×
KC	27.10	25.86	25.84	1.05×	1.00×
DP	36.59	34.14	32.61	1.12×	1.05×
HS	43.86	34.81	34.00	1.29×	1.02×
LCR	14.24	11.92	11.88	1.20×	1.00×
MST	69.82	62.87	50.70	1.38×	1.24×
VC	1.60	1.30	1.26	1.27×	1.03×
Geomean				**1.17**×	**1.05**×

- Running times and speedups over serialization-based approaches
- Universal improvement by CLONE
- Speedups depend on structure of transferred data

Non-blocking range operations

- FPGA-based implementation based on LEON3 cache controller
- One range buffer
- Overhead compared to unmodified cache controller:

	absolute	relative
Slices	1489	15.2%
Register	623	14.6%
LUT	1491	15.0%
BRAM	1	4.9%

Non-blocking range operations

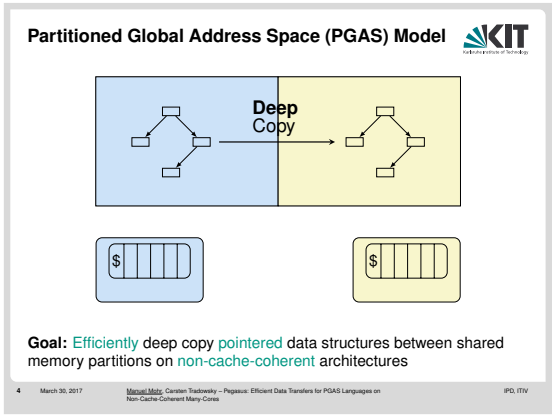
- FPGA-based implementation based on LEON3 cache controller
- One range buffer
- Overhead compared to unmodified cache controller:

	absolute	relative
Slices	1489	15.2%
Register	623	14.6%
LUT	1491	15.0%
BRAM	1	4.9%

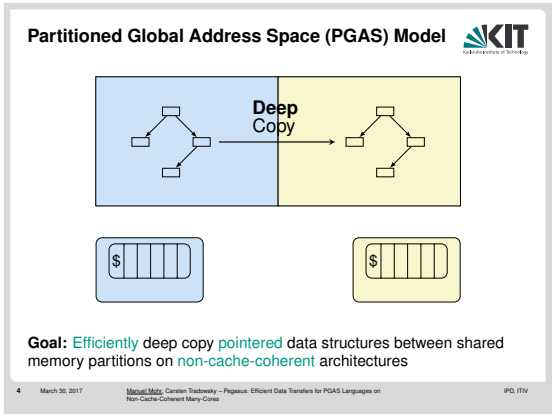
- Data from benchmarks: 17 cache lines on average
- Enough non-memory instructions to cover latency of range operations

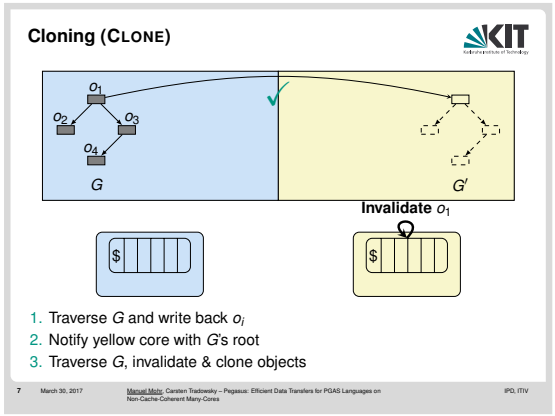
$\Rightarrow$ Expected to take one cycle from view of CPU

Summary

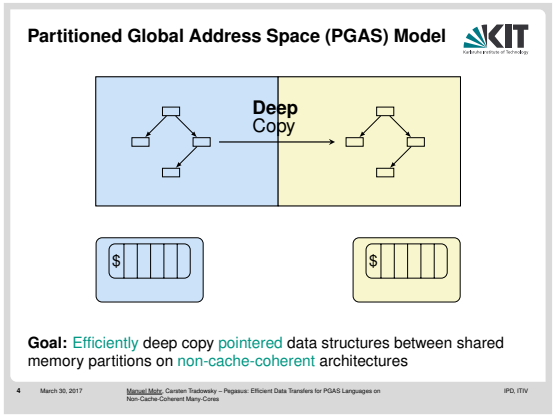

Partitioned Global Address Space (PGAS) Model
Deep
Copy
$
$
Goal: Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures
4 March 30, 2017 Manuel Mohr, Carsten Tradowsky – Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores IPD, ITIV

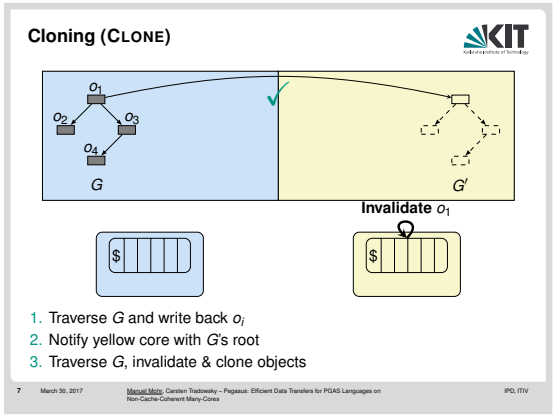
Summary

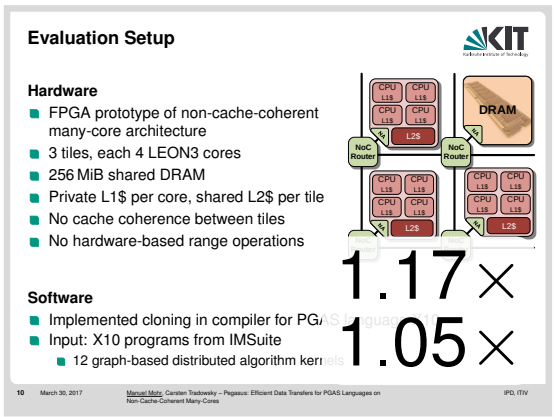

Partitioned Global Address Space (PGAS) Model
Deep
Copy
$
$
Goal: Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures
4 March 30, 2017 Manuel Mohr, Carsten Tradowsky – Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores IPD, ITIV


Cloning (CLONE)
o_1
o_2
o_3
o_4
G
G'
Invalidate o_1
$
$
1. Traverse G and write back o_i
2. Notify yellow core with G's root
3. Traverse G, invalidate & clone objects
7 March 30, 2017 Manuel Mohr, Carsten Tradowsky – Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores IPD, ITIV

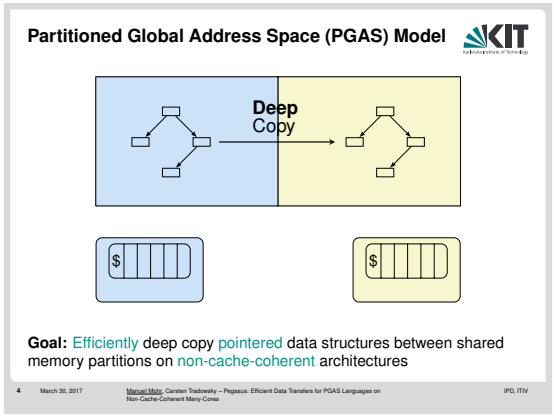
Summary

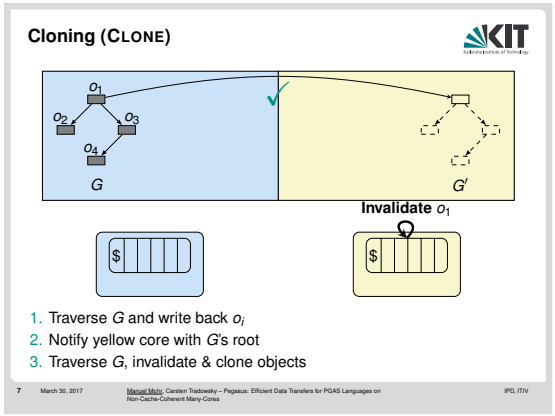

Partitioned Global Address Space (PGAS) Model
Deep
Copy
$
$
Goal: Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures
4 March 30, 2017 Manuel Mohr, Carsten Tradowsky – Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores IPD, ITIV

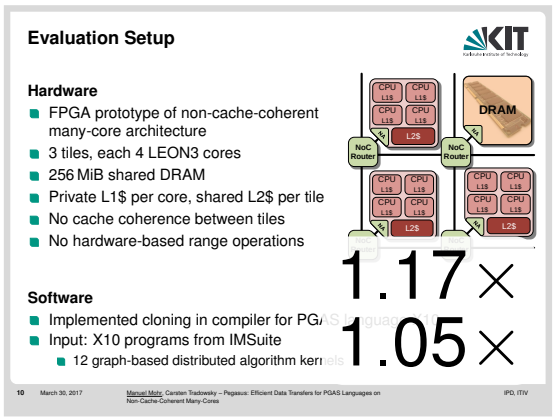

Cloning (CLONE)
o_1
o_2
o_3
o_4
G
G'
Invalidate o_1
$
$
1. Traverse G and write back o_i
2. Notify yellow core with G's root
3. Traverse G, invalidate & clone objects
7 March 30, 2017 Manuel Mohr, Carsten Tradowsky – Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores IPD, ITIV

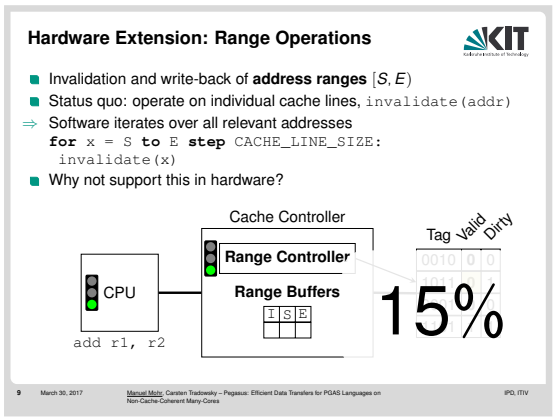

Evaluation Setup
Hardware
FPGA prototype of non-cache-coherent many-core architecture
3 tiles, each 4 LEON3 cores
256 MiB shared DRAM
Private L1$ per core, shared L2$ per tile
No cache coherence between tiles
No hardware-based range operations
CPU L1$
L2$
NoC Router
DRAM
Software
Implemented cloning in compiler for PGAS languages
Input: X10 programs from IMSuite
12 graph-based distributed algorithm kernels
1.17×
1.05×
10 March 30, 2017 Manuel Mohr, Carsten Tradowsky – Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores IPD, ITIV

Summary

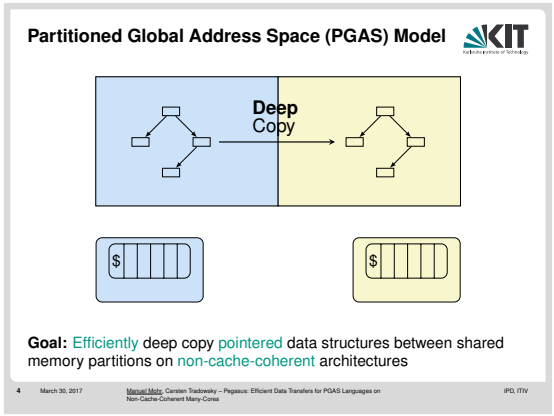

Partitioned Global Address Space (PGAS) Model
Deep
Copy
$
$
Goal: Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures
4 March 30, 2017 Manuel Mohr, Carsten Tradowsky – Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores IPD, ITIV

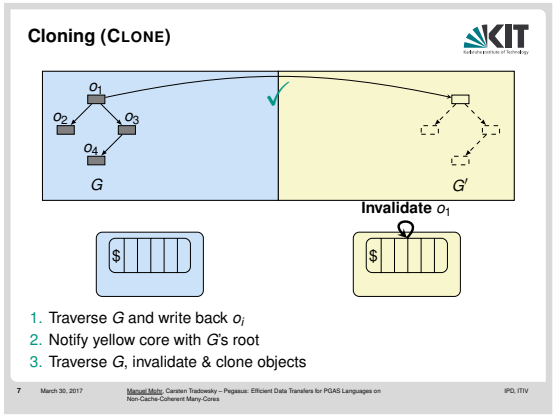

Cloning (CLONE)
o_1
o_2
o_3
o_4
G
G'
Invalidate o_1
$
$
1. Traverse G and write back o_i
2. Notify yellow core with G's root
3. Traverse G, invalidate & clone objects
7 March 30, 2017 Manuel Mohr, Carsten Tradowsky – Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores IPD, ITIV

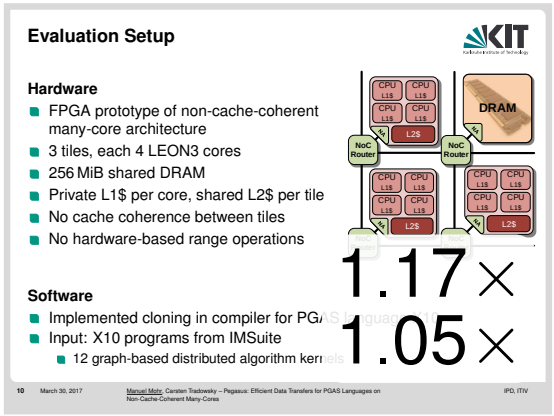

Evaluation Setup
Hardware
FPGA prototype of non-cache-coherent many-core architecture
3 tiles, each 4 LEON3 cores
256 MiB shared DRAM
Private L1$ per core, shared L2$ per tile
No cache coherence between tiles
No hardware-based range operations
CPU L1$
L2$
NoC Router
DRAM
Software
Implemented cloning in compiler for PGAS
Input: X10 programs from IMSuite
12 graph-based distributed algorithm ker
1.17×
1.05×
10 March 30, 2017 Manuel Mohr, Carsten Tradowsky – Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores IPD, ITIV

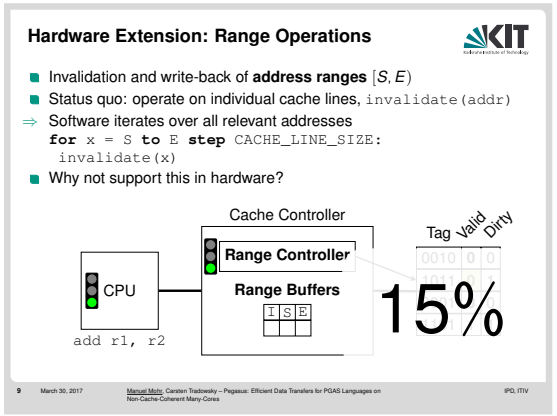

Hardware Extension: Range Operations
Invalidation and write-back of address ranges [S, E)
Status quo: operate on individual cache lines, invalidate(addr)
⇒ Software iterates over all relevant addresses
for x = S to E step CACHE_LINE_SIZE:
invalidate(x)
Why not support this in hardware?
Cache Controller
Range Controller
Range Buffers
I S E
CPU
add r1, r2
Tag Valid Dirty
15%
9 March 30, 2017 Manuel Mohr, Carsten Tradowsky – Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores IPD, ITIV

Summary









Interesting point in the design space of non-cache-coherent systems

- PGAS model exposes existence of multiple coherence domains
- Compiler accelerates implicit data transfers via shared memory
- Benefits from hardware support for range operations

Backup

Why benchmarks without hardware extension?

- No conceptual obstacles
  - Implementation technique generally applicable
- Prototype platform has two-level cache hierarchy
- Range operations must work on both levels
- Caches very different

⇒ Redundant work to show feasibility of concept

- Definitely planned for future work