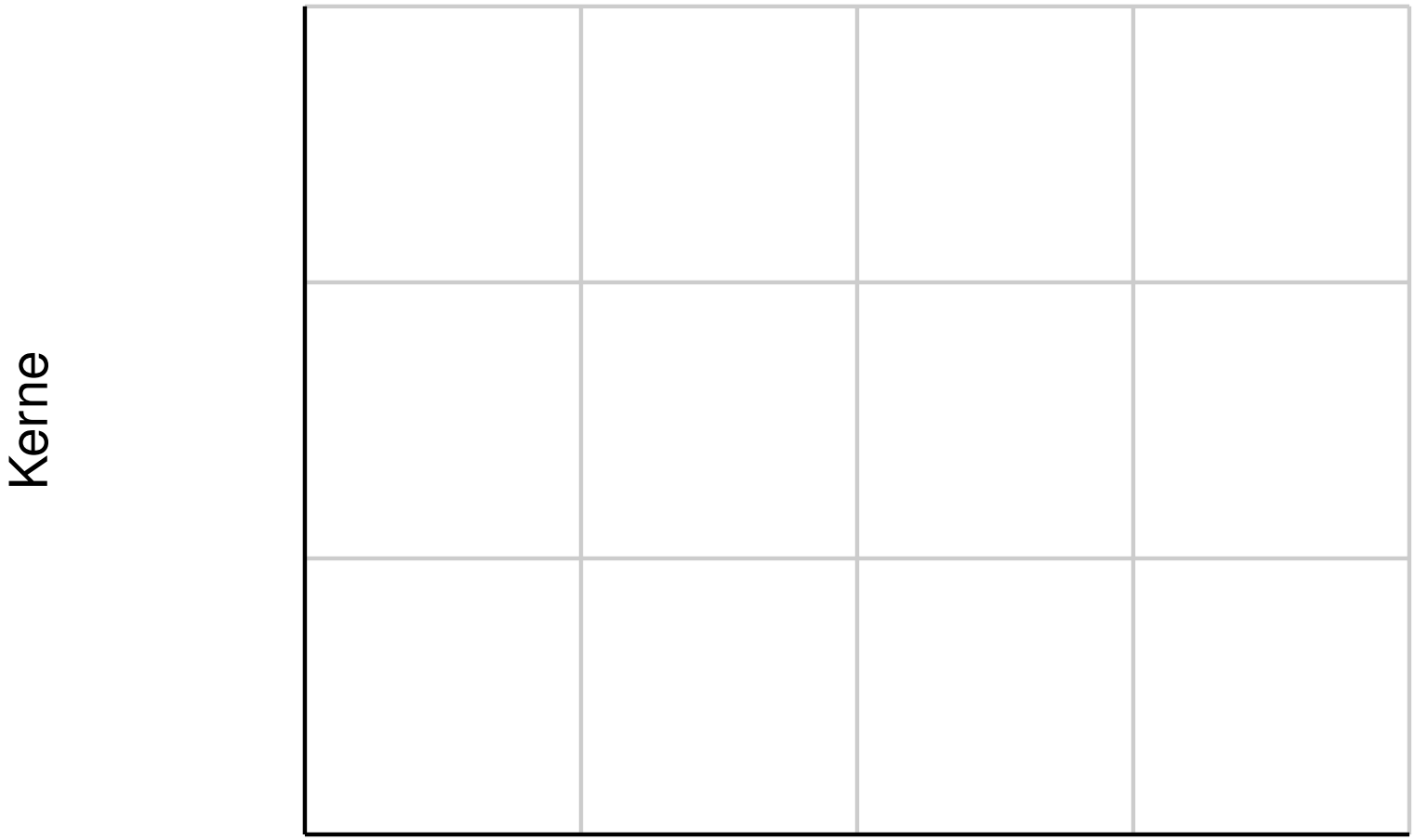
Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures

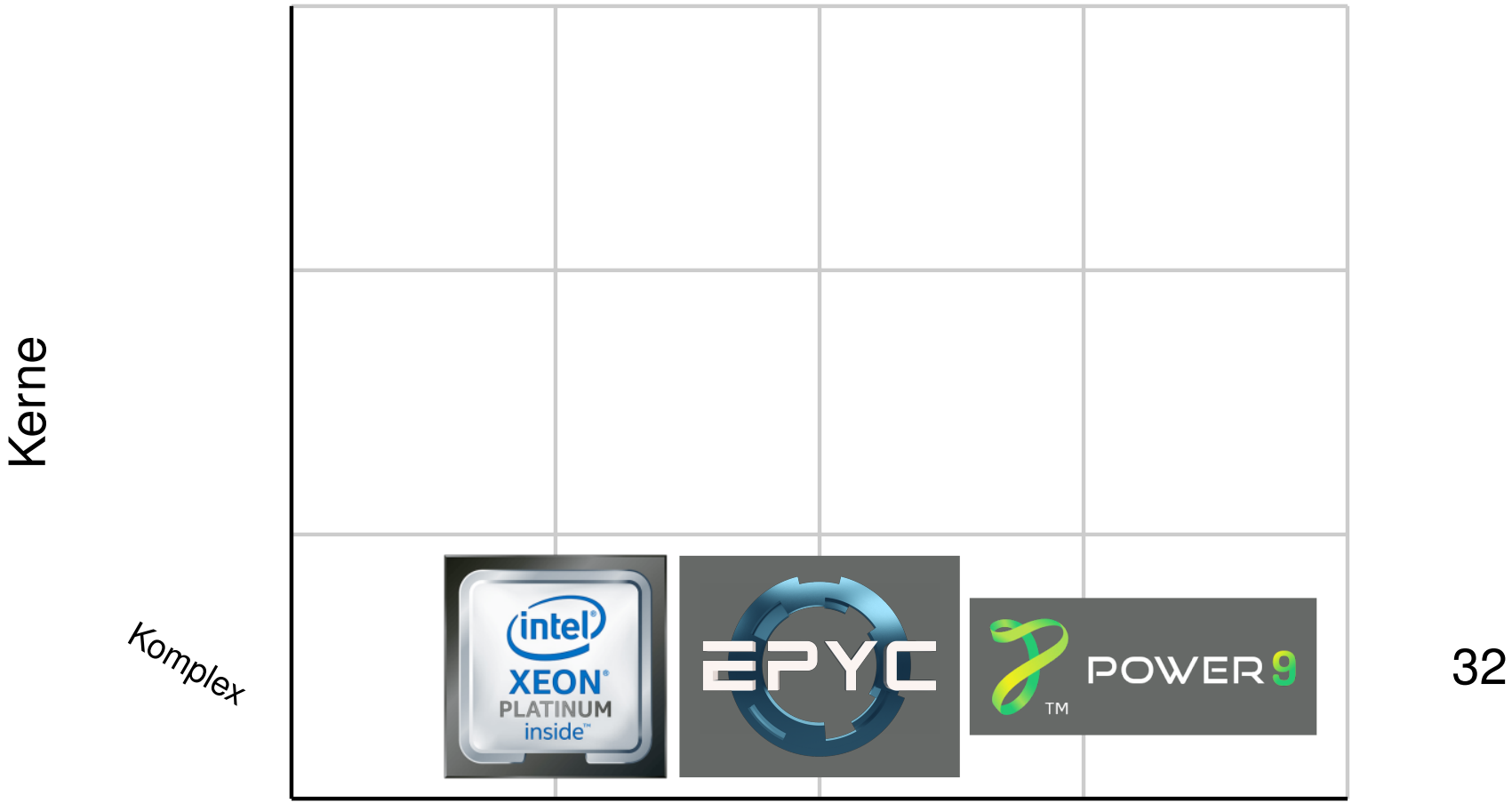
Manuel Mohr

Lehrstuhl für Programmierparadigmen, Karlsruher Institut für Technologie (KIT)

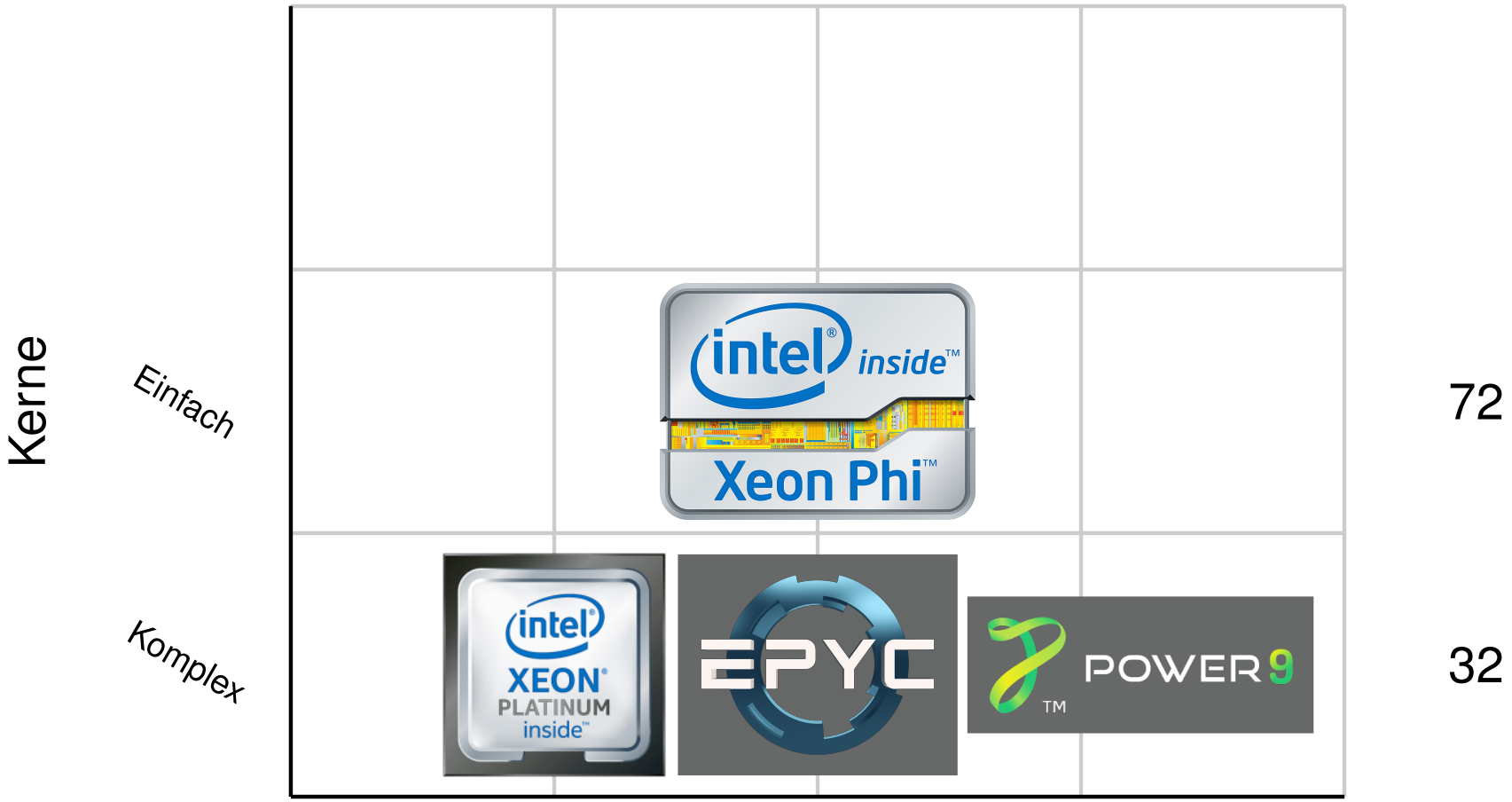
Veränderte Hardwarelandschaft



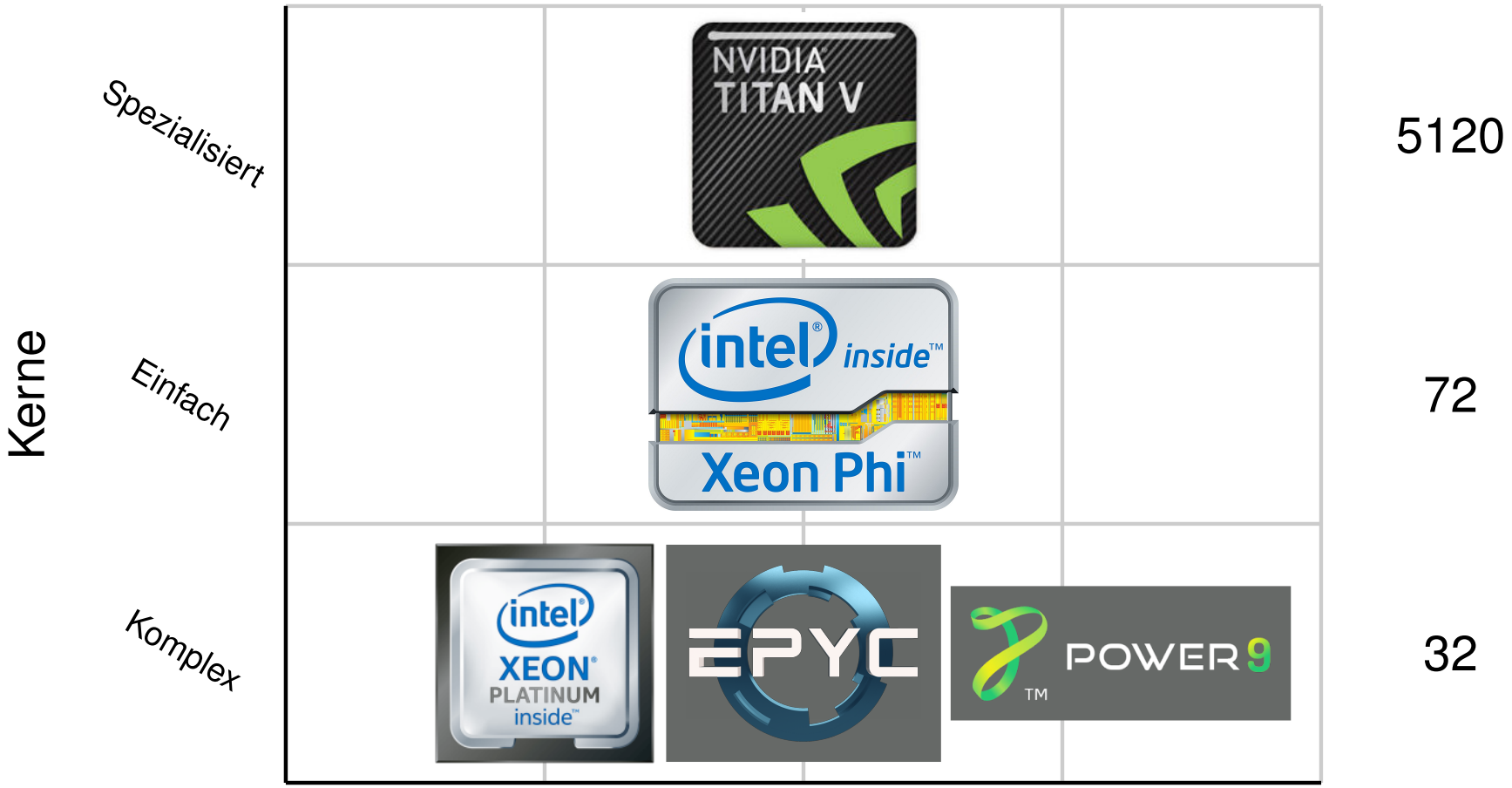
Veränderte Hardwarelandschaft



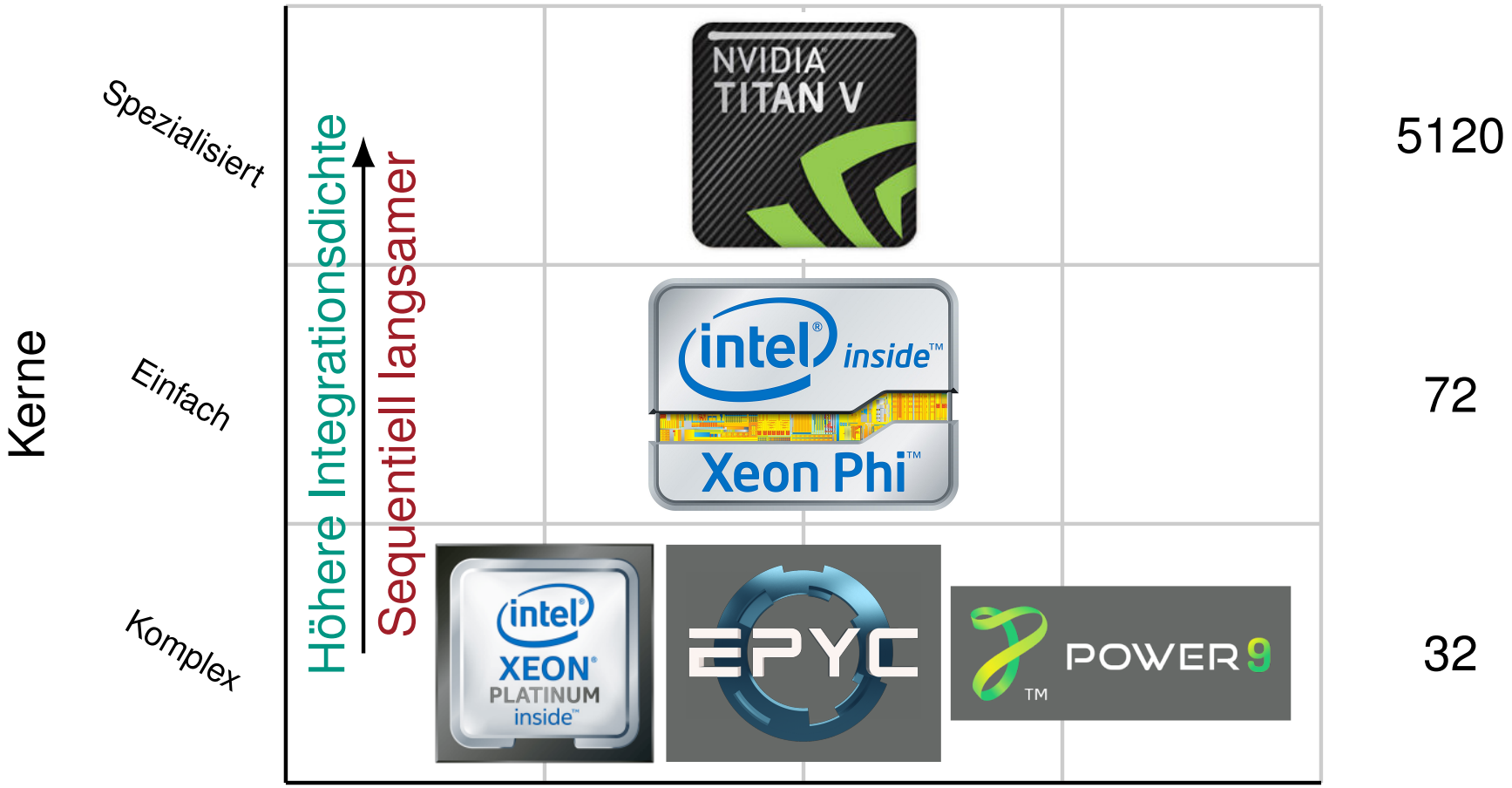
Veränderte Hardwarelandschaft



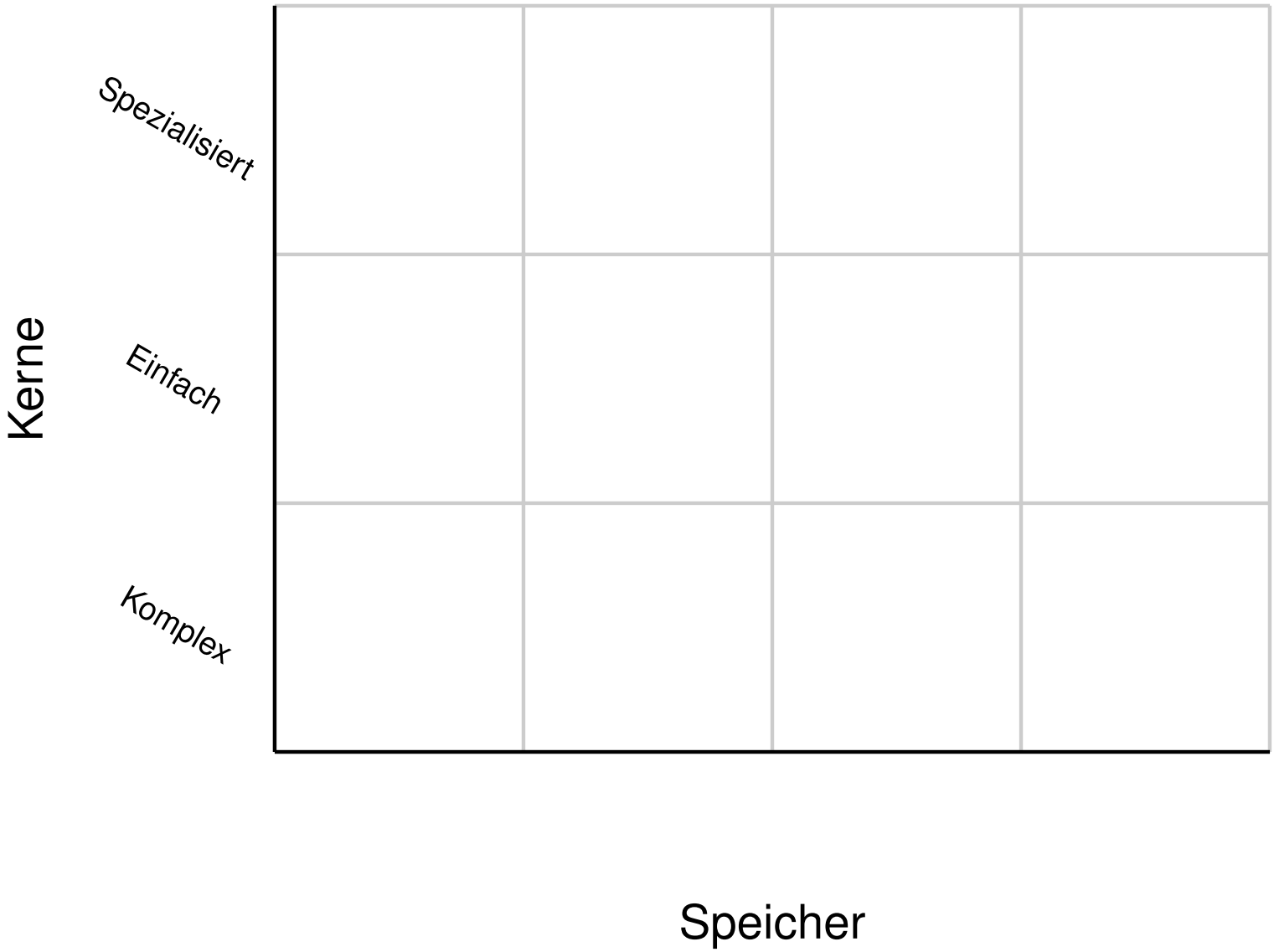
Veränderte Hardwarelandschaft



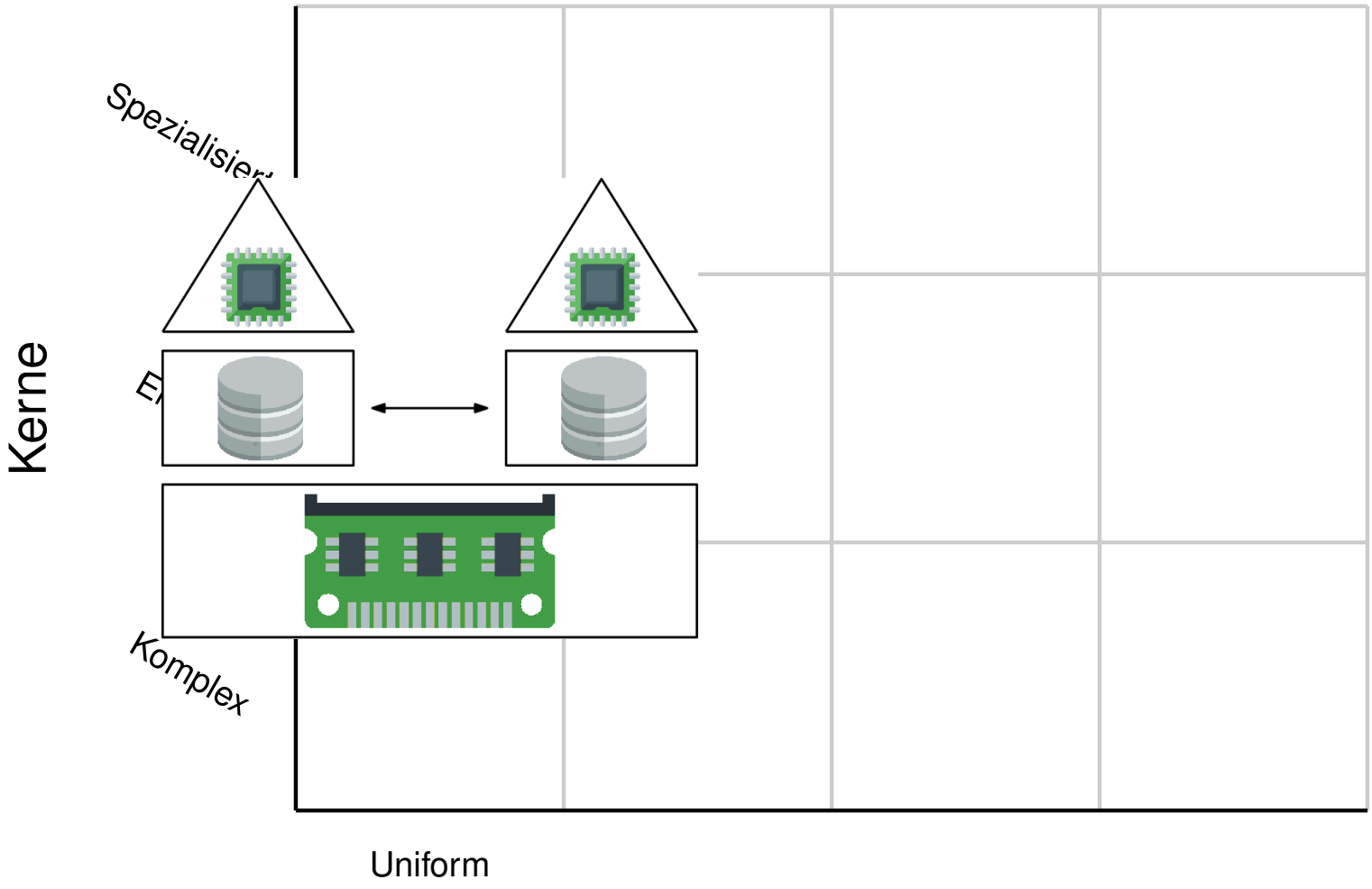
Veränderte Hardwarelandschaft



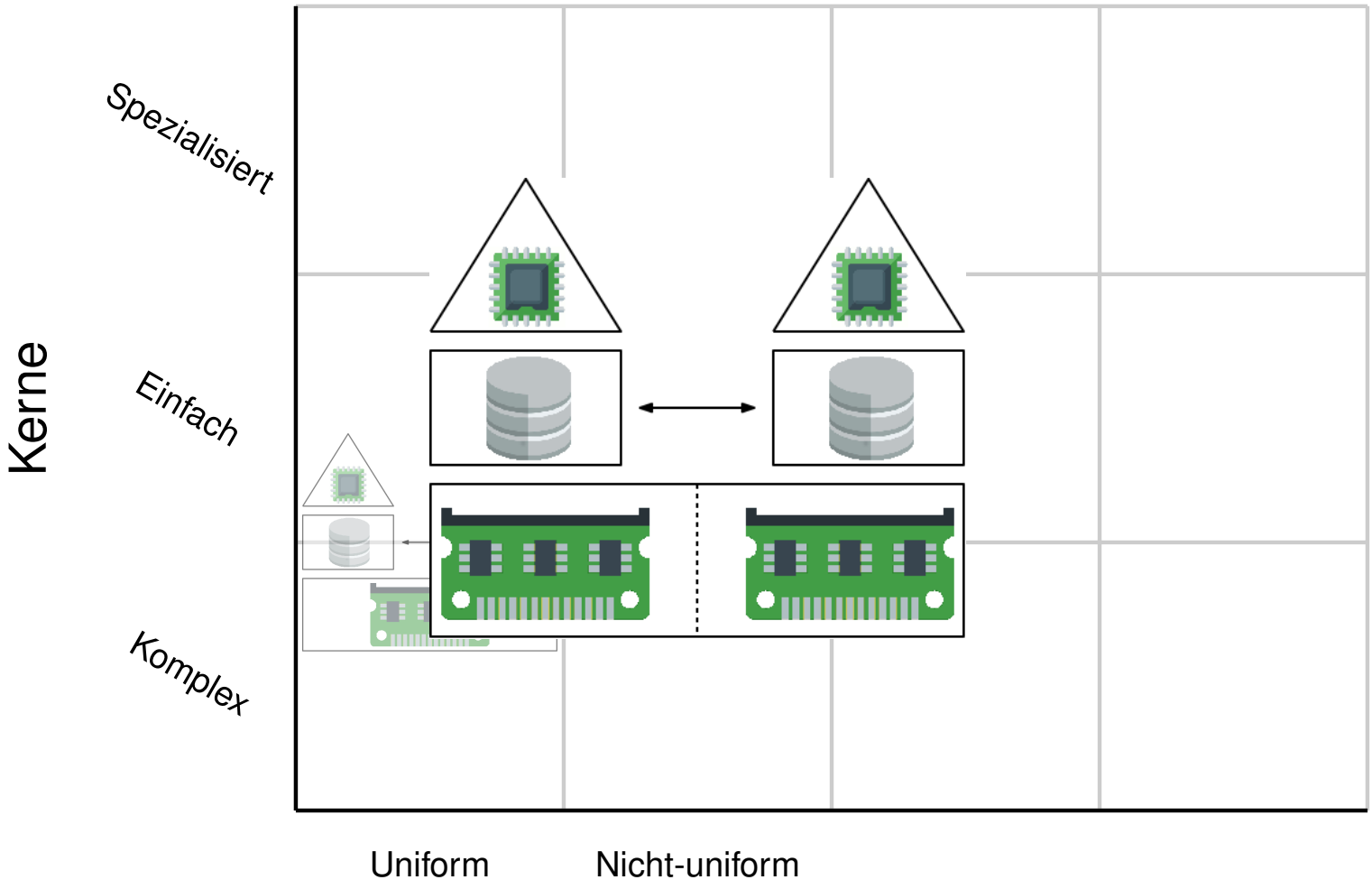
Veränderte Hardwarelandschaft



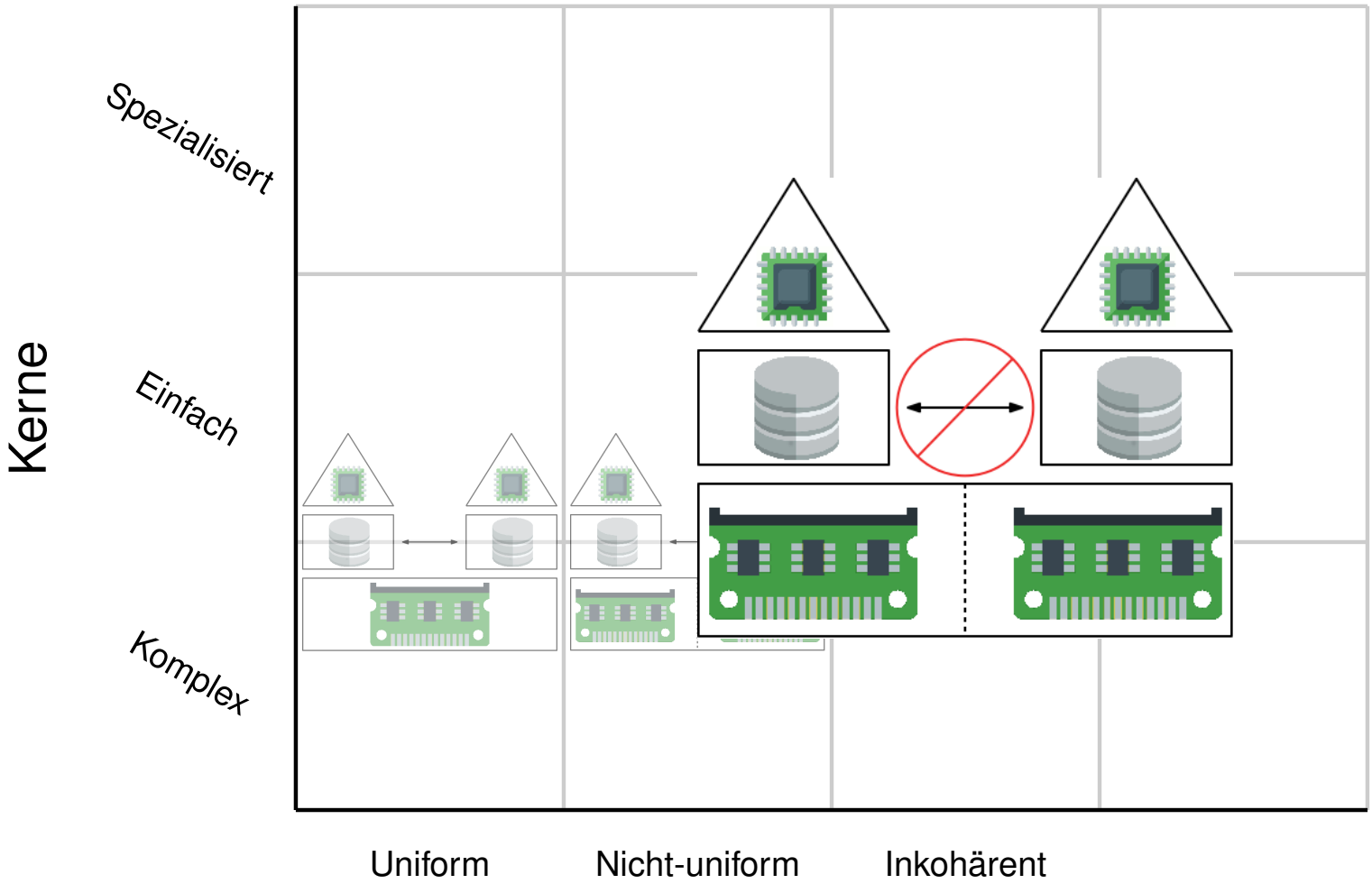
Veränderte Hardwarelandschaft



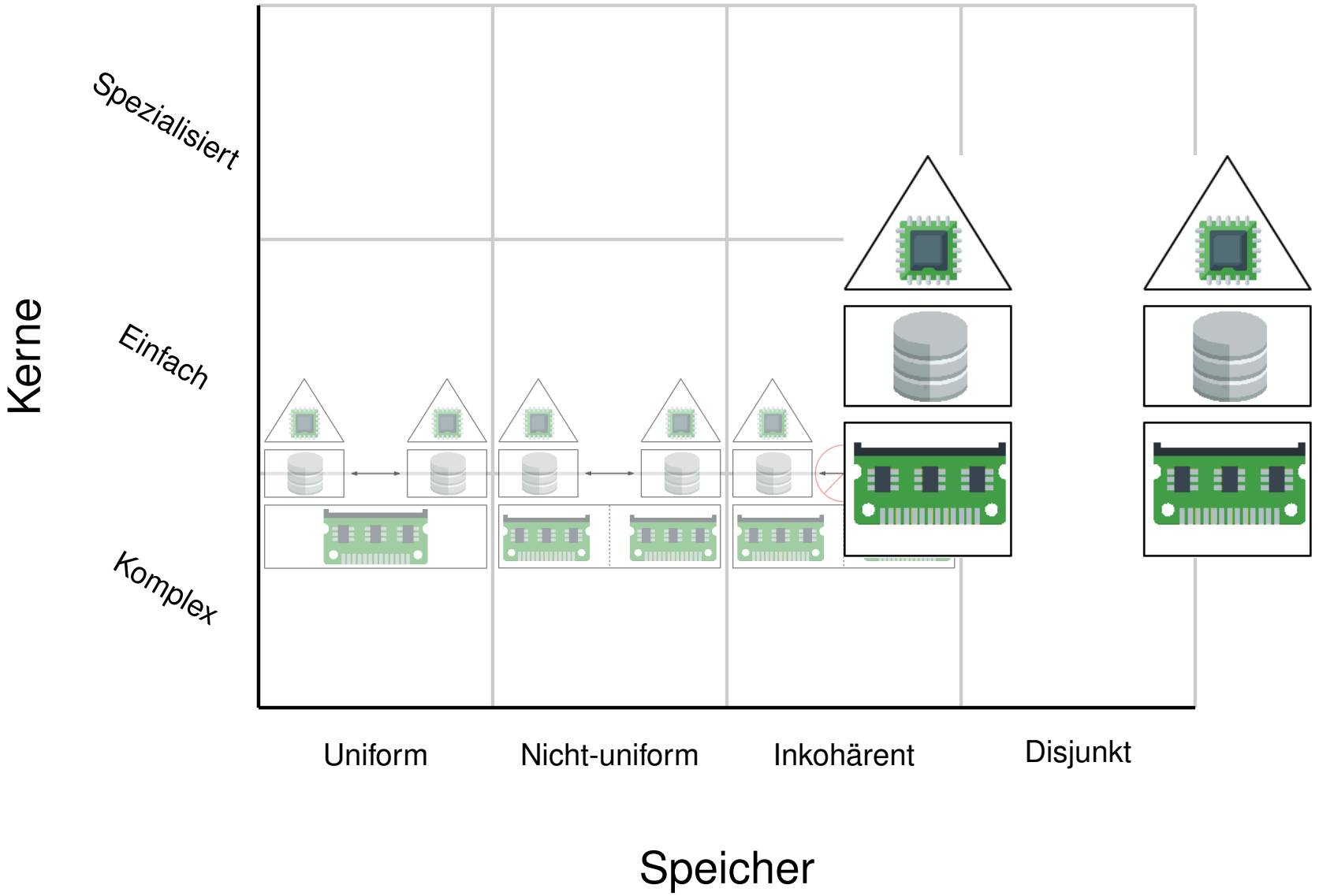
Veränderte Hardwarelandschaft



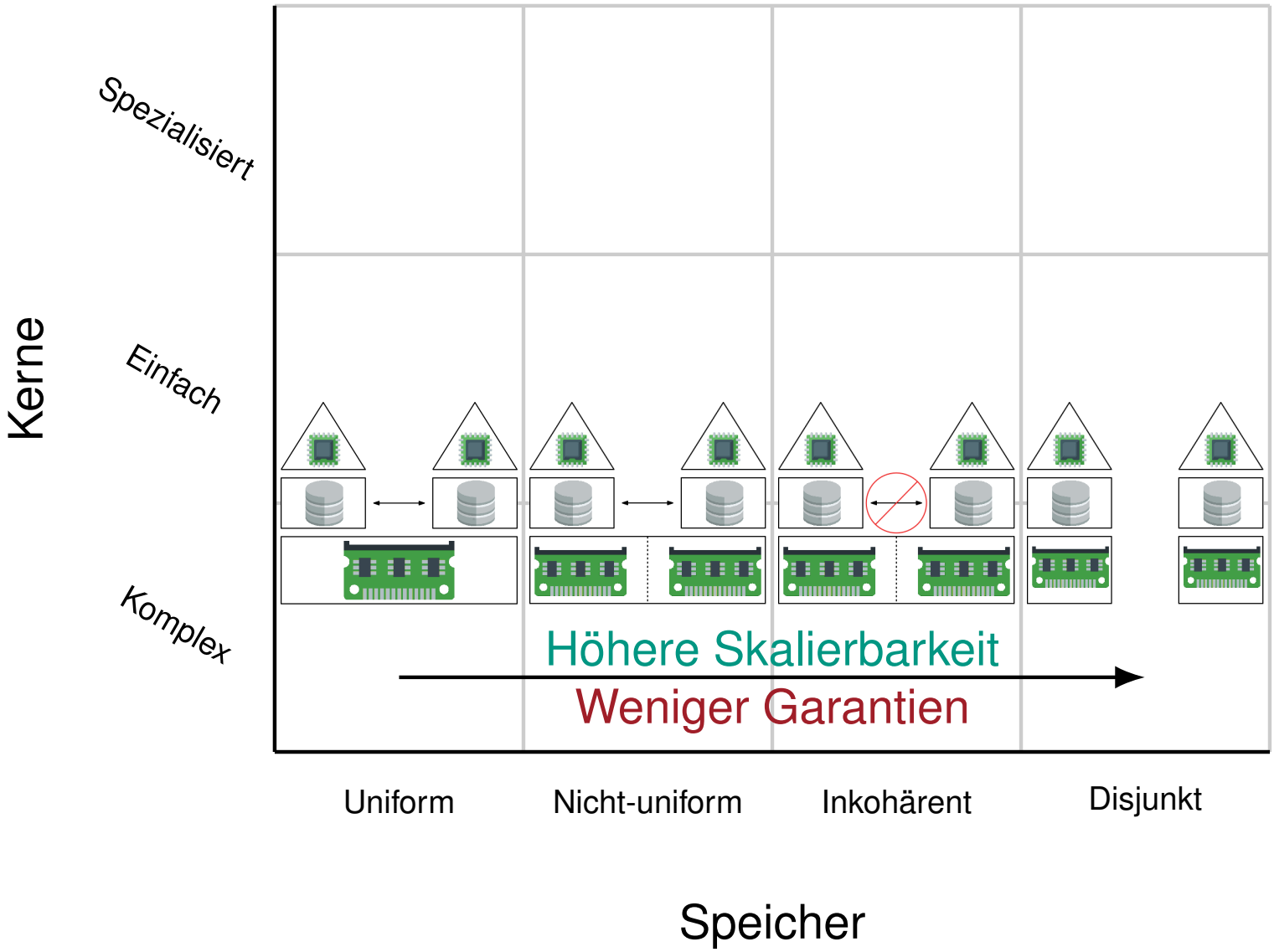
Veränderte Hardwarelandschaft



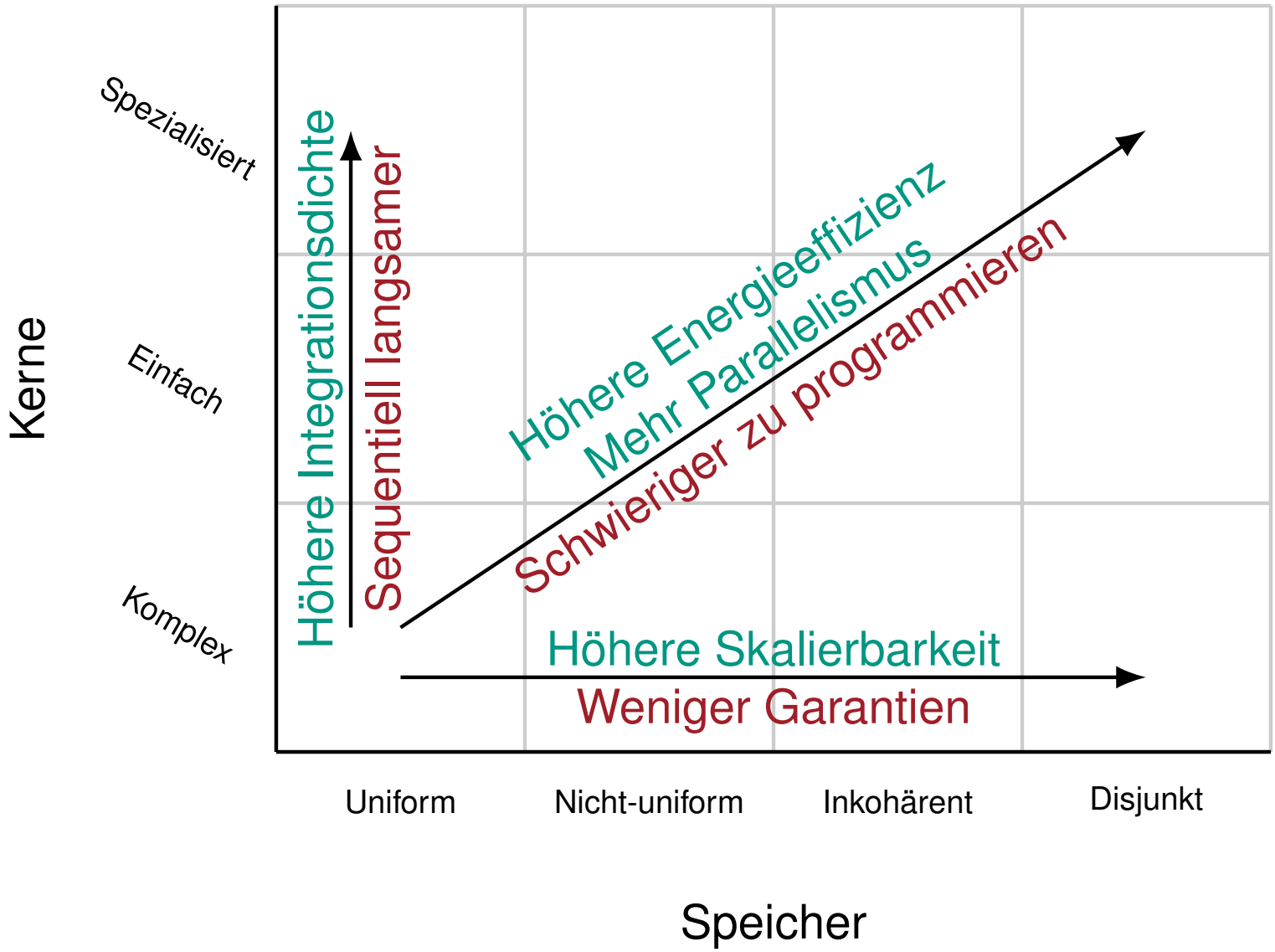
Veränderte Hardwarelandschaft



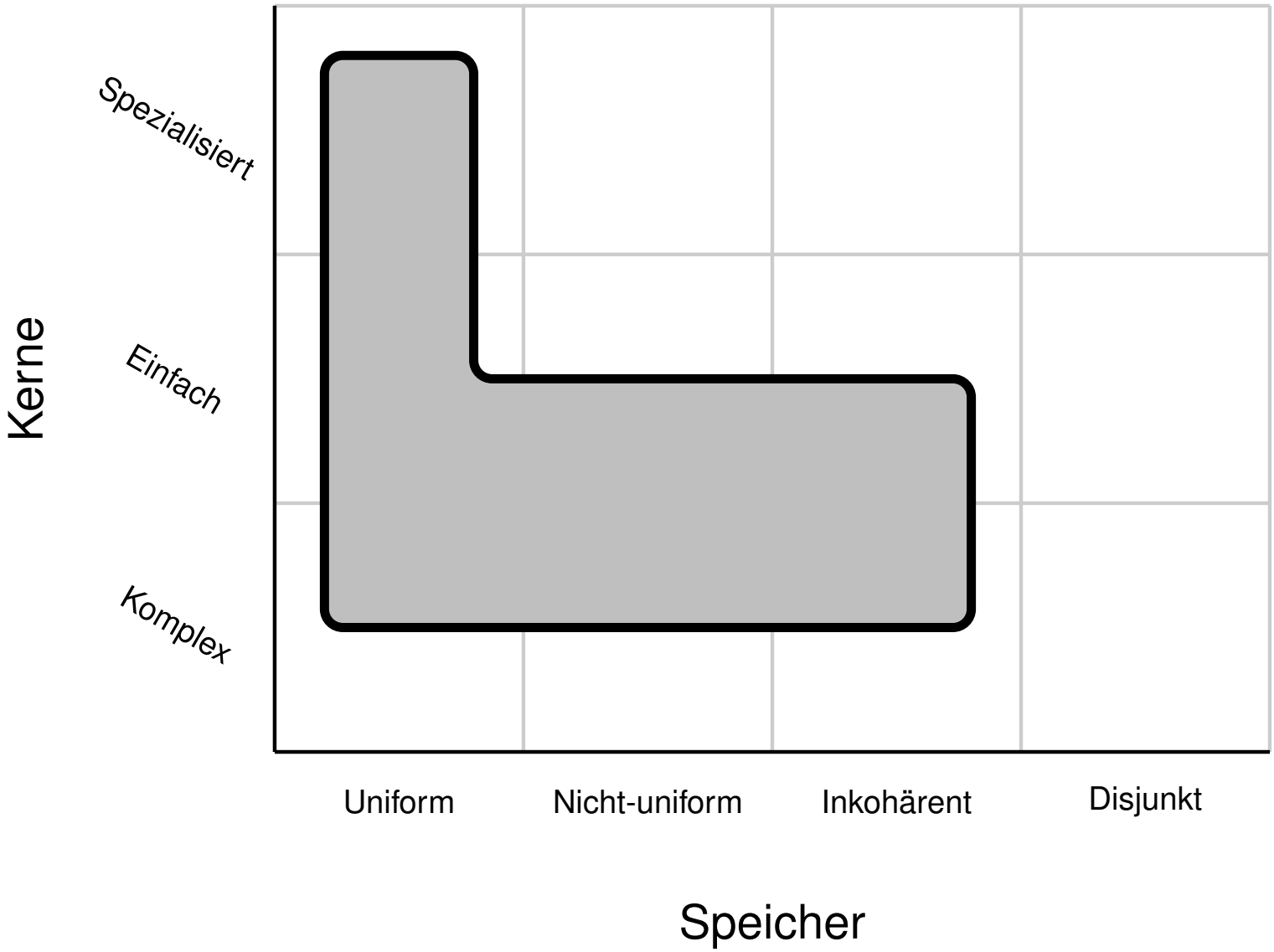
Veränderte Hardwarelandschaft



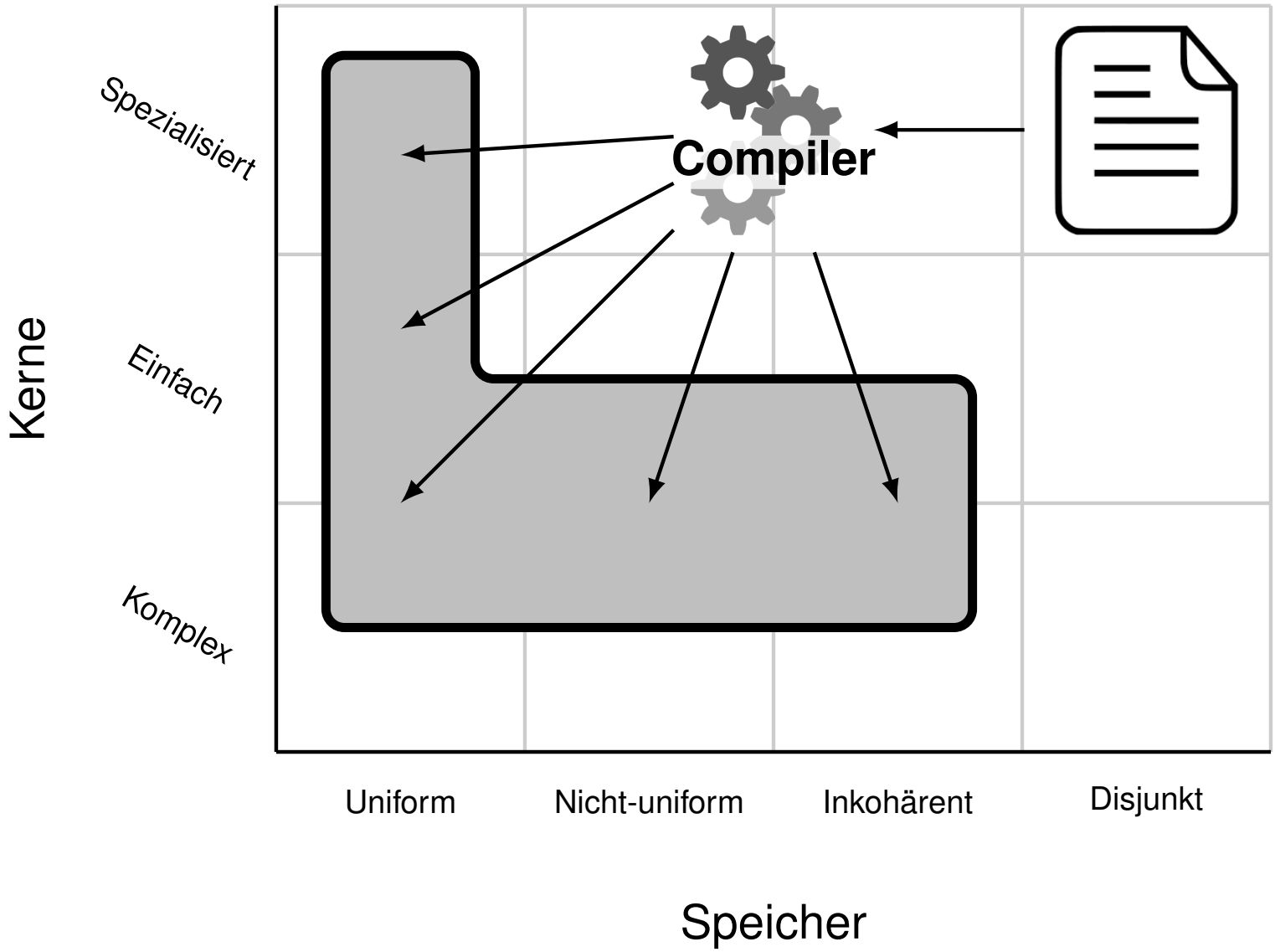
Veränderte Hardwarelandschaft



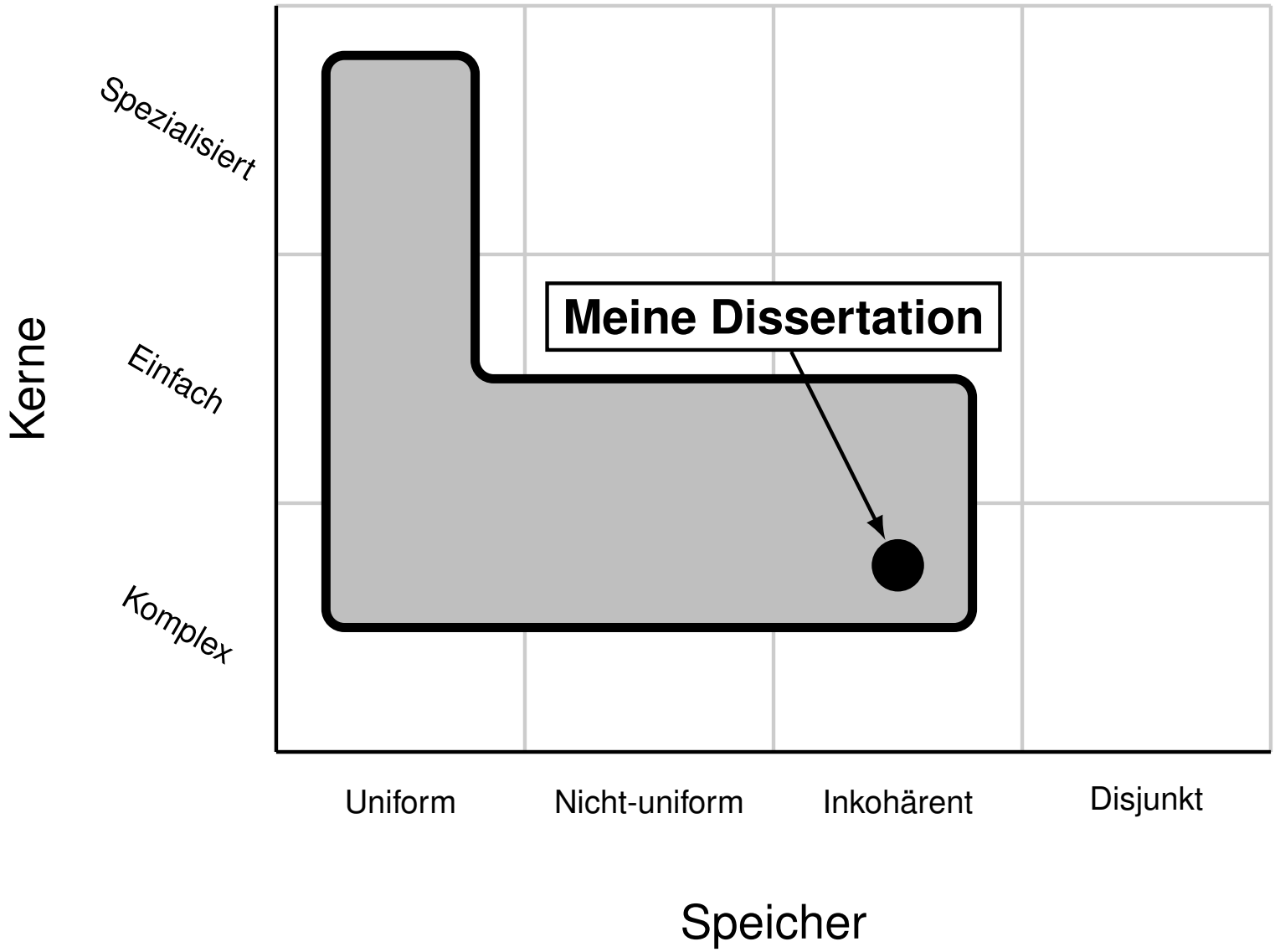
Veränderte Hardwarelandschaft



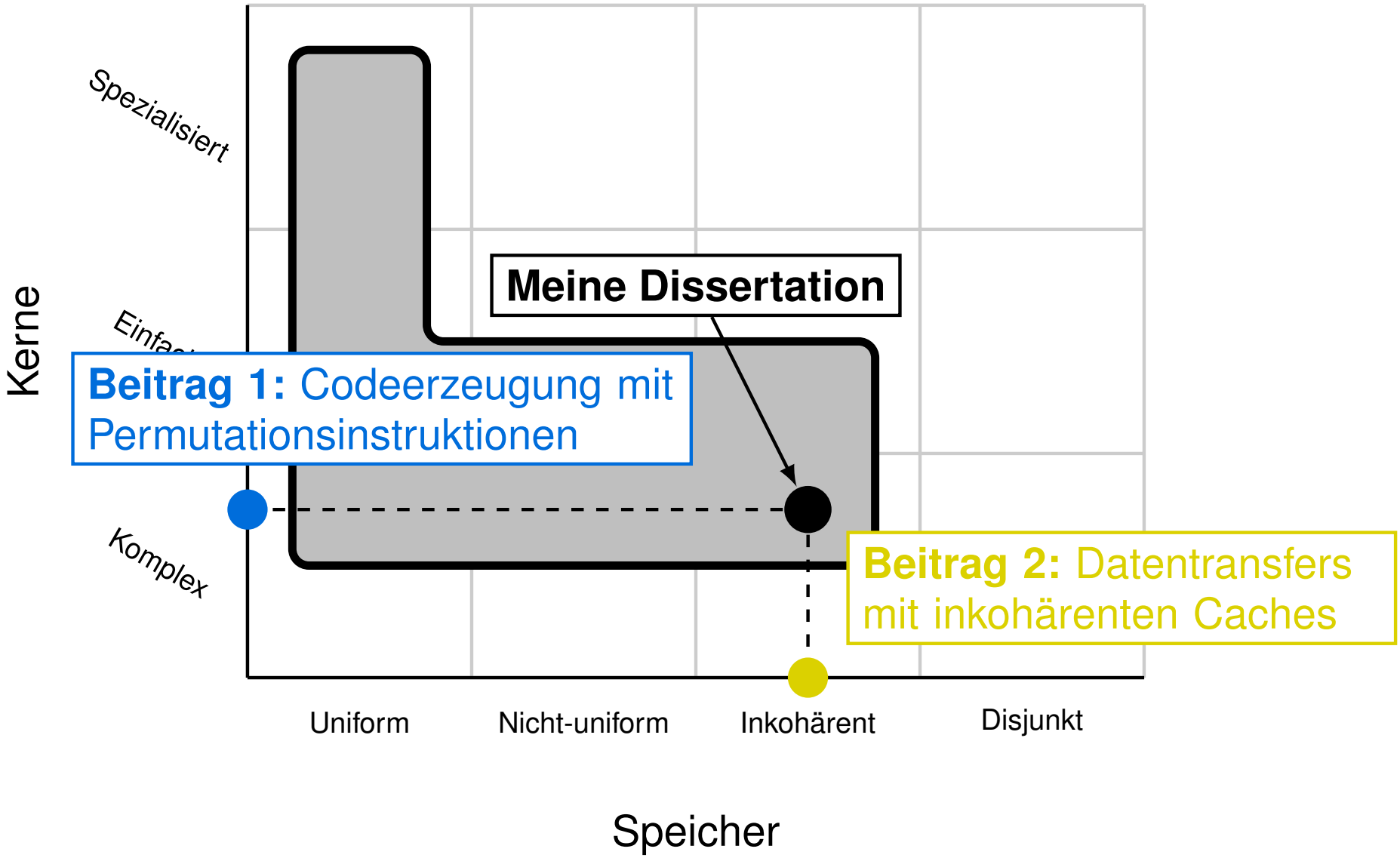
Veränderte Hardwarelandschaft

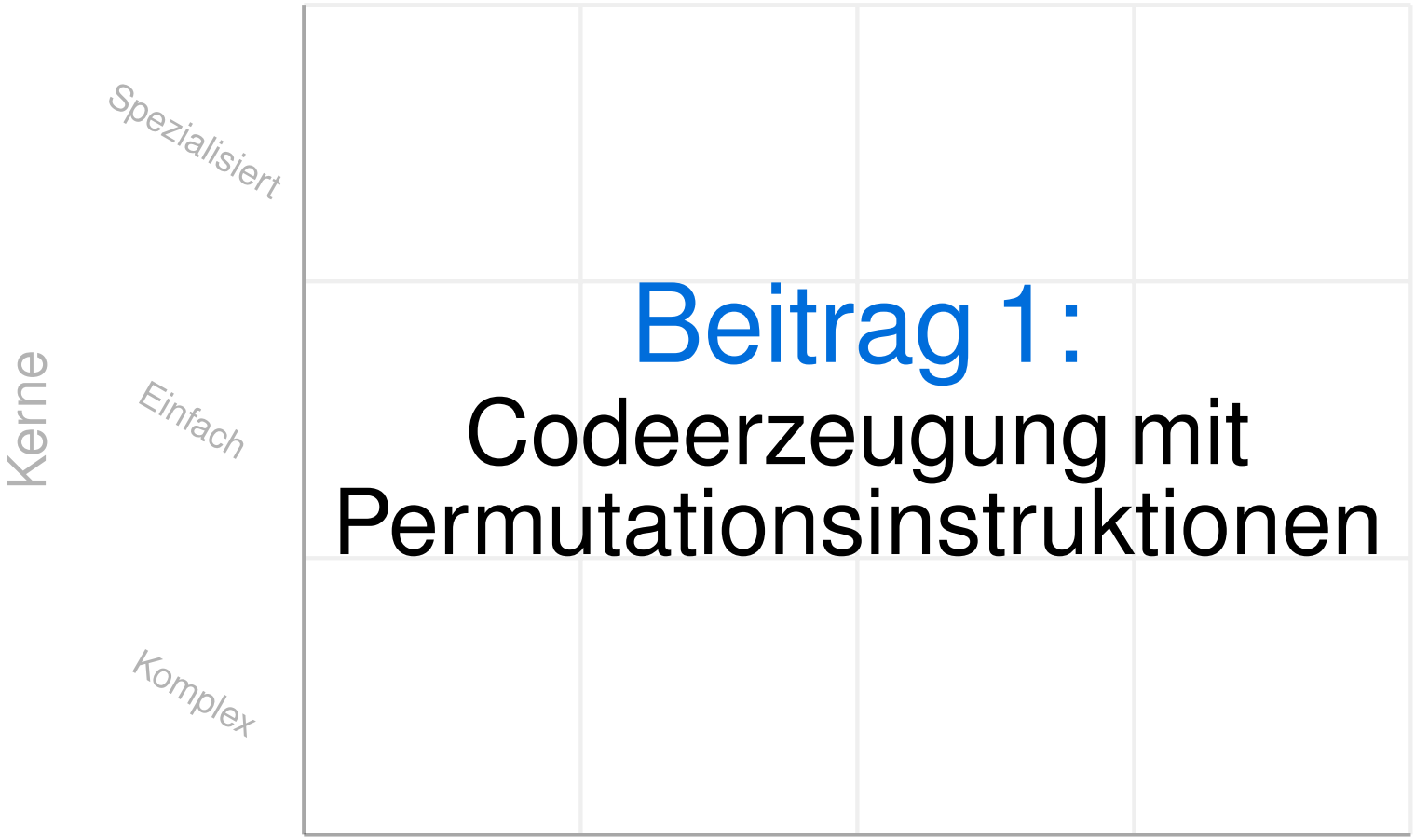


Veränderte Hardwarelandschaft



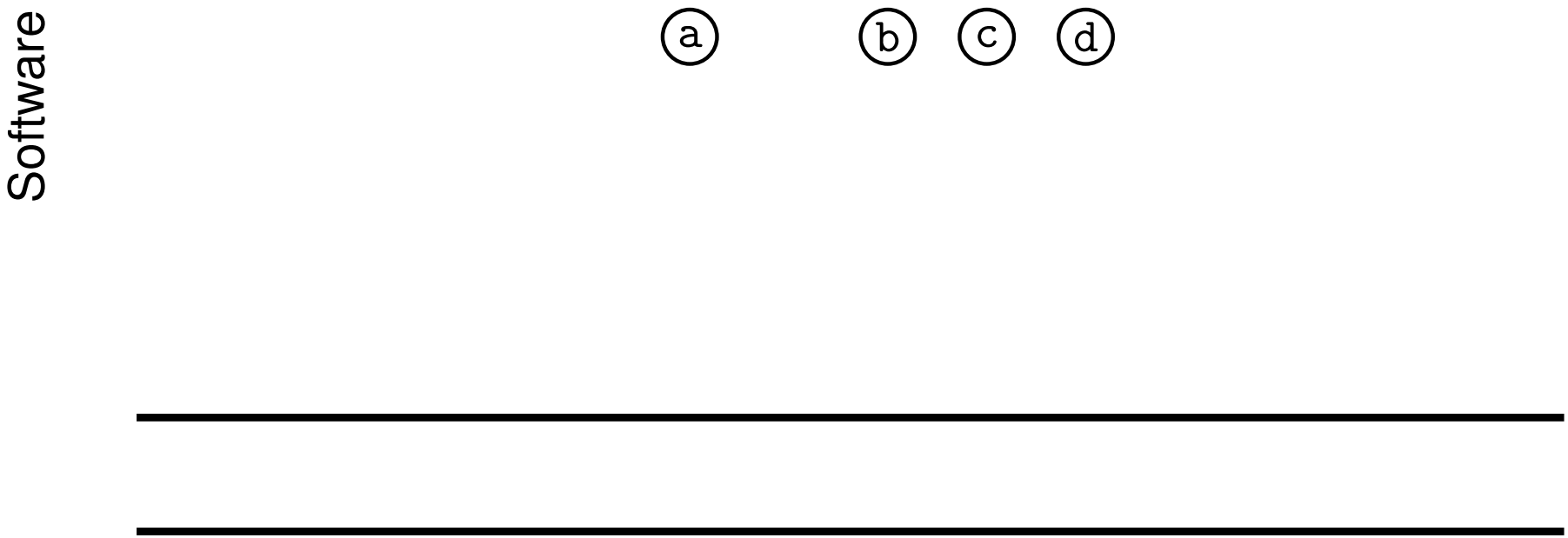
Veränderte Hardwarelandschaft





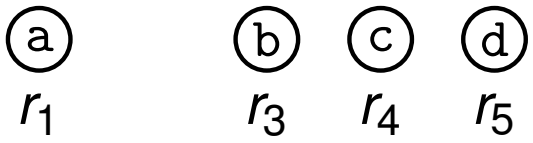
Beitrag 1:
Codeerzeugung mit Permutationsinstruktionen

Motivation



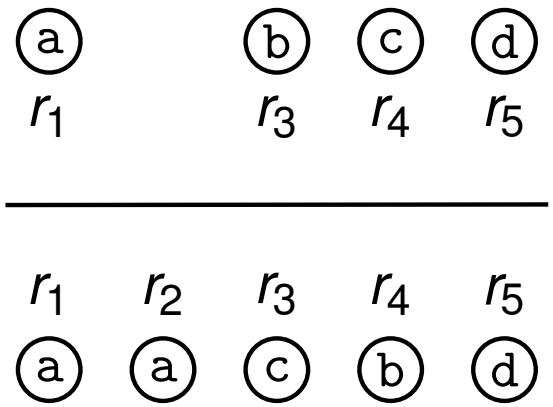
Motivation

Software



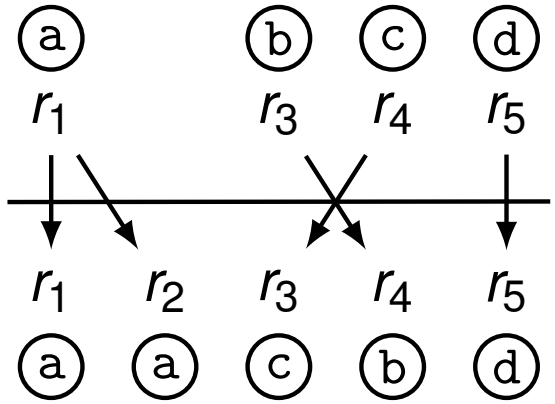
Motivation

Software



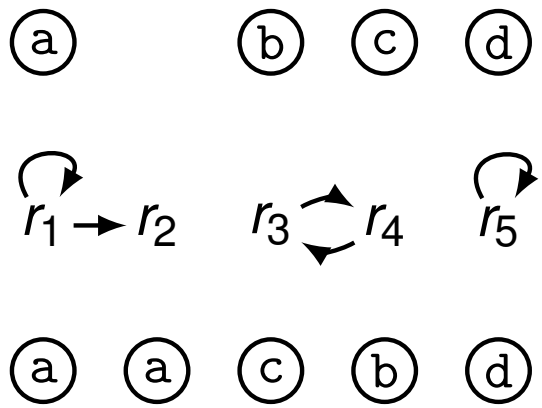
Motivation

Software



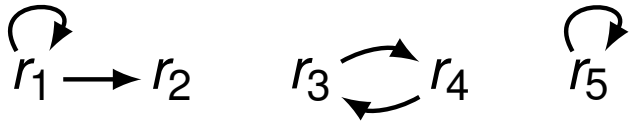
Motivation

Software



Motivation

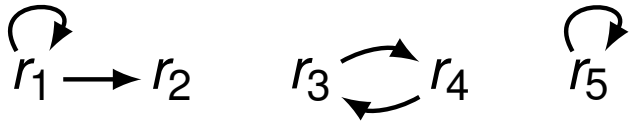
Software



Register-Transfer-Graph (RTG)

Motivation

Software



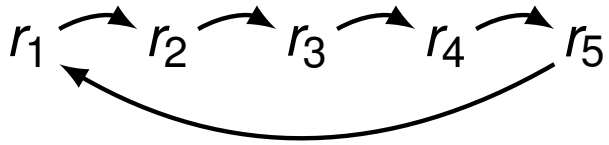
Register-Transfer-Graph (RTG)

Befehlssatz

`copy` r_1, r_2	`swap` r_3, r_4

Motivation

Software

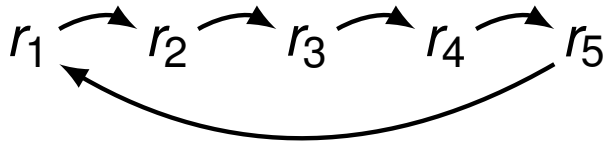


Befehlssatz

`swap` r_5, r_4	`swap` r_4, r_3	`swap` r_3, r_2	`swap` r_2, r_1

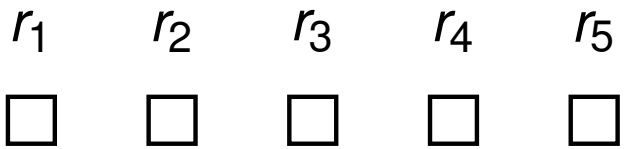
Motivation

Software



Befehlssatz

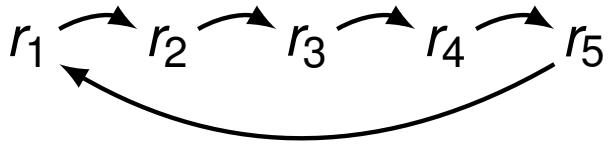
`swap` r_5, r_4 | `swap` r_4, r_3 | `swap` r_3, r_2 | `swap` r_2, r_1



Hardware

Motivation

Software

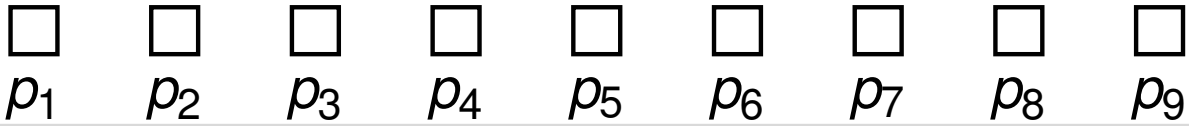


Befehlssatz

`swap` r_5, r_4 `swap` r_4, r_3 `swap` r_3, r_2 `swap` r_2, r_1

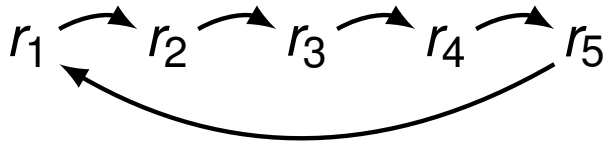
r_1 r_2 r_3 r_4 r_5

Hardware



Motivation

Software

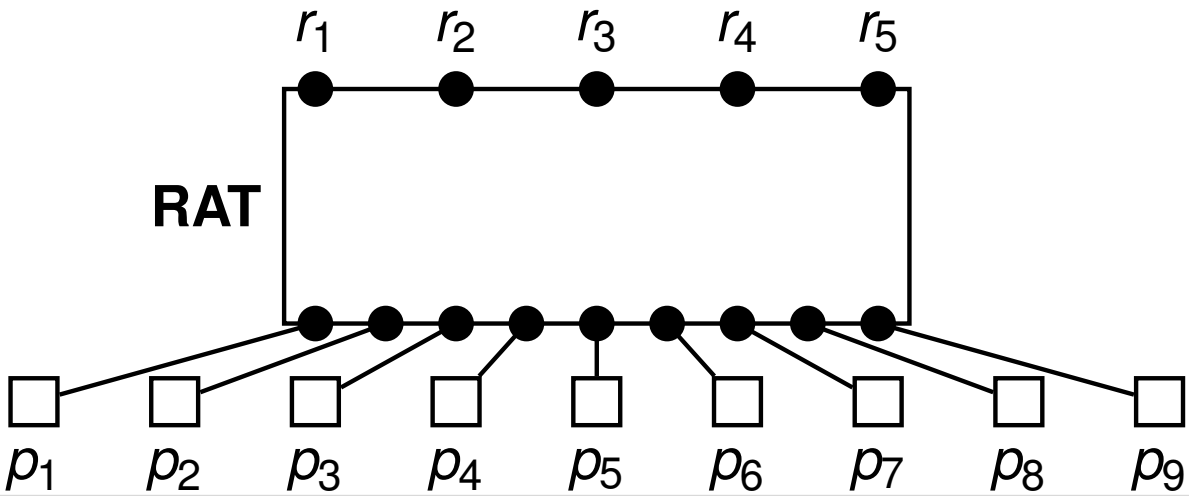


Befehlssatz

`swap` r_5, r_4 `swap` r_4, r_3 `swap` r_3, r_2 `swap` r_2, r_1

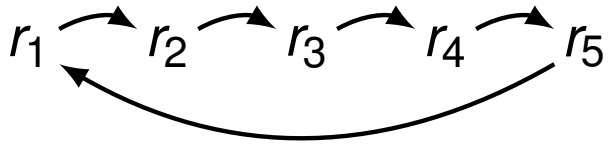
Hardware

Register-Alias-Tabelle



Motivation

Software

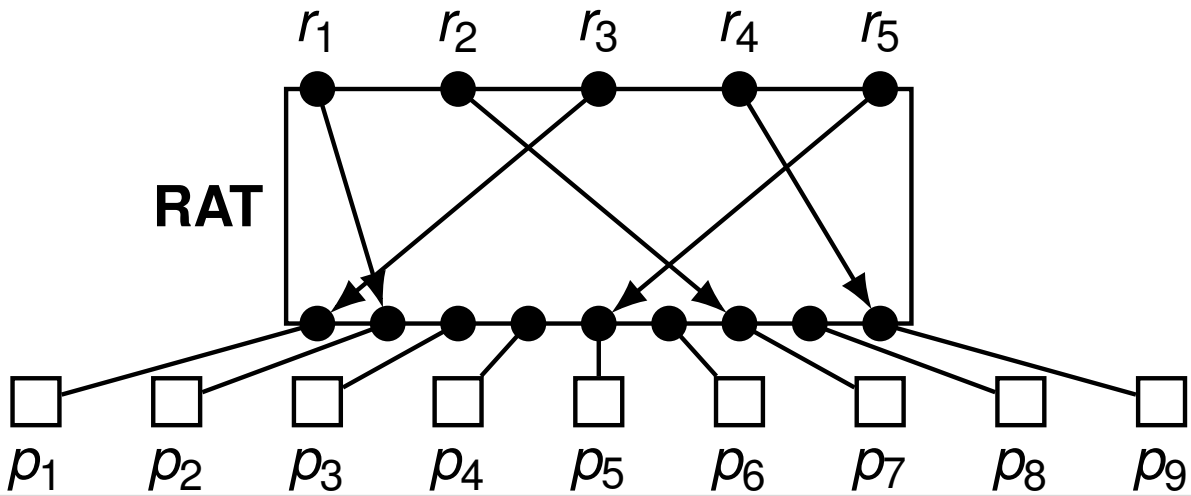


Befehlssatz

`swap` r_5, r_4 `swap` r_4, r_3 `swap` r_3, r_2 `swap` r_2, r_1

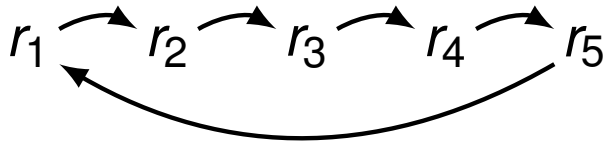
Hardware

Register-Alias-Tabelle



Motivation

Software

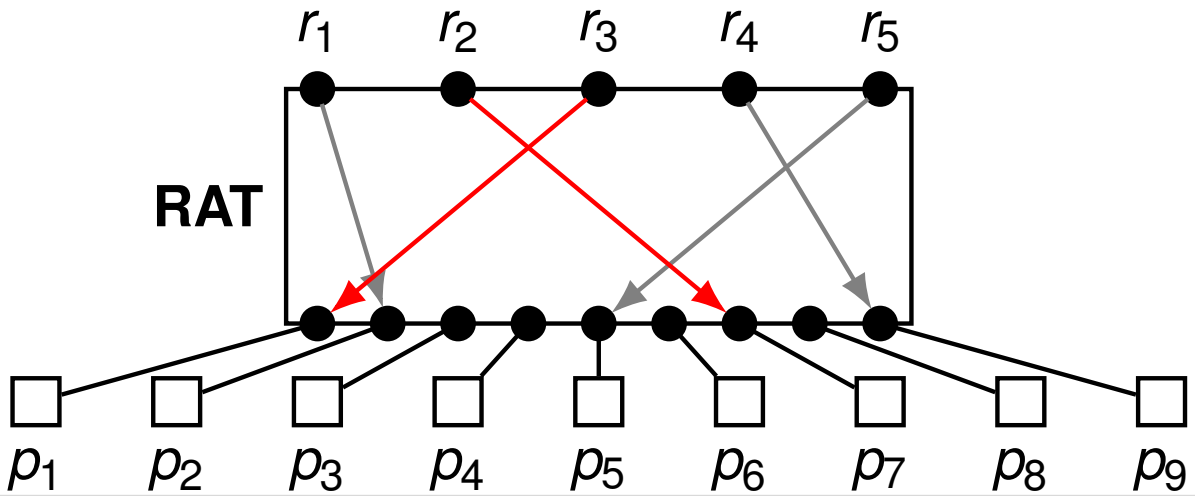


Befehlssatz

`swap` r_5, r_4 `swap` r_4, r_3 `swap` r_3, r_2 `swap` r_2, r_1

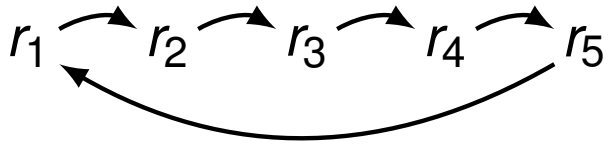
Hardware

Register-Alias-Tabelle



Motivation

Software

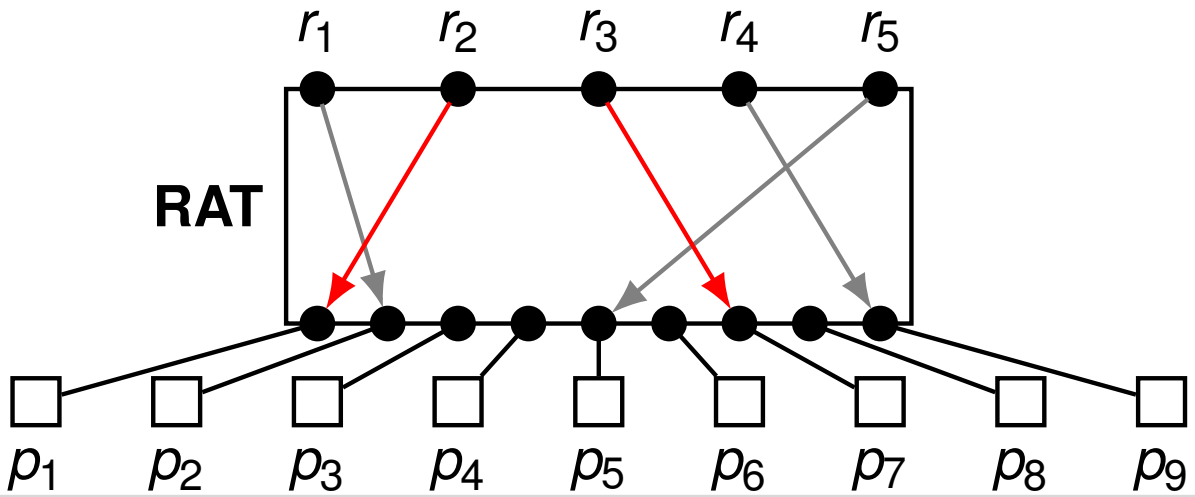


Befehlssatz

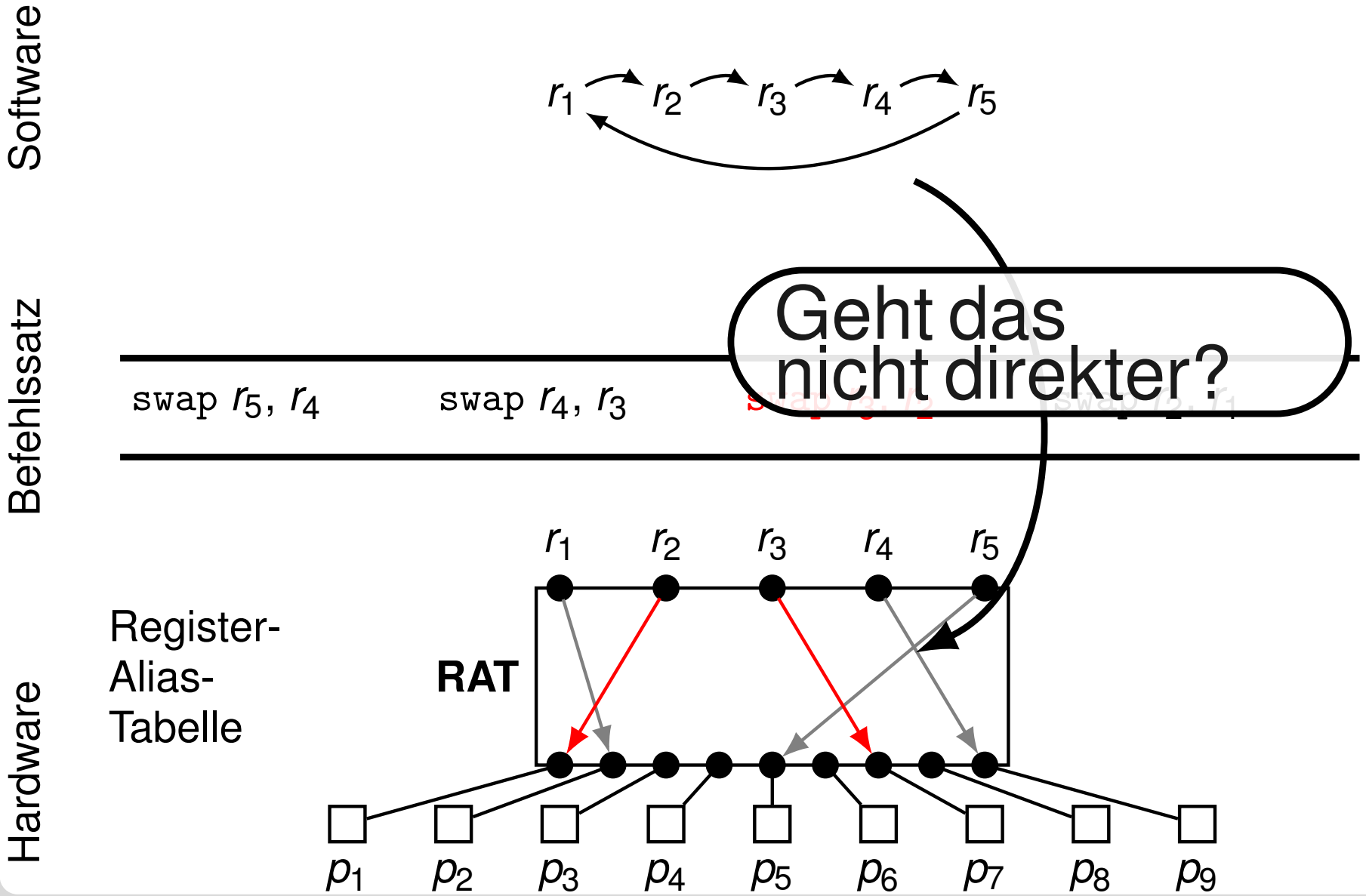
`swap` r_5, r_4 `swap` r_4, r_3 `swap` r_3, r_2 `swap` r_2, r_1

Hardware

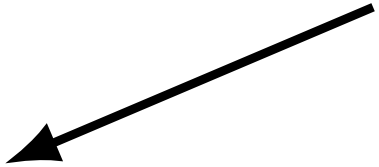
Register-Alias-Tabelle



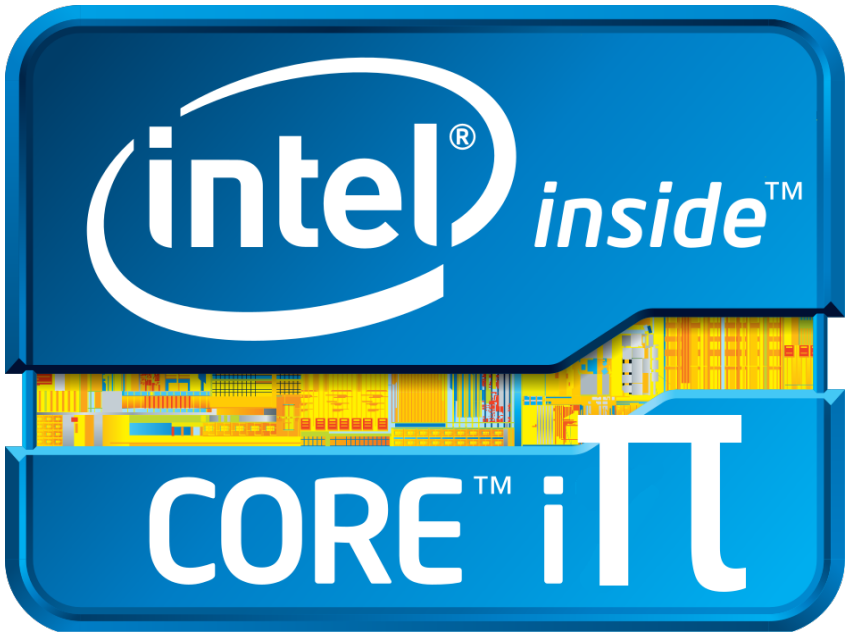
Motivation


Software
Befehlssatz
Hardware
r_1 r_2 r_3 r_4 r_5
Geht das nicht direkter?
swap r_5, r_4
swap r_4, r_3
Register-Alias-Tabelle
RAT
r_1 r_2 r_3 r_4 r_5
p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9

Machbarkeitsbeweis [Moh+13]

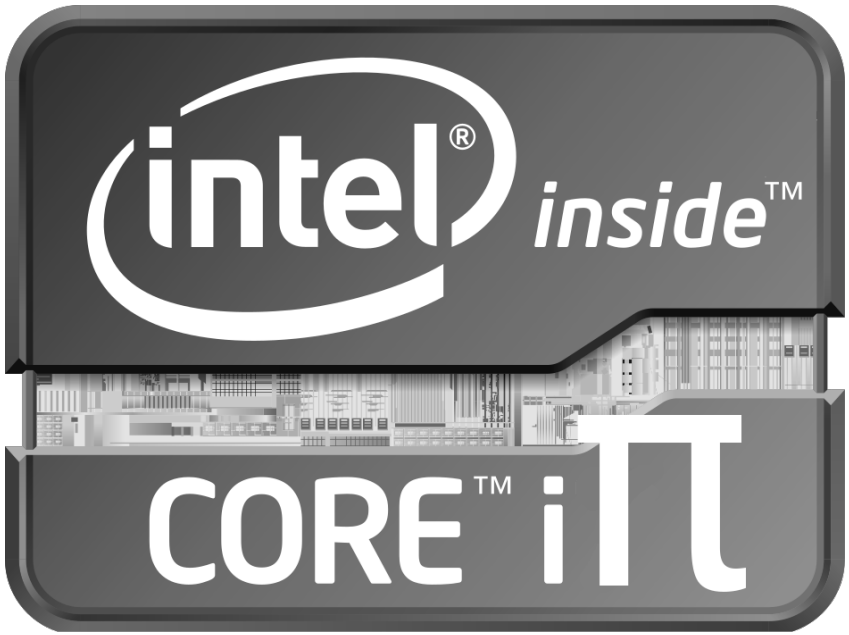


Modifikation
komplexer
Architektur



Machbarkeitsbeweis [Moh+13]

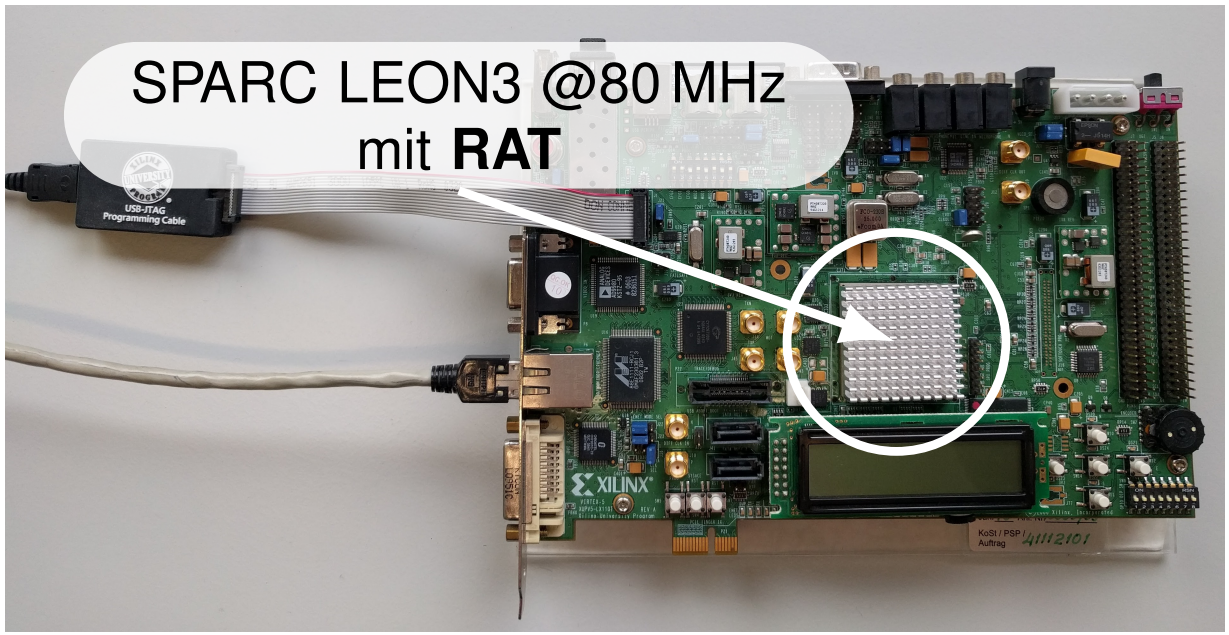
Modifikation komplexer Architektur



Machbarkeitsbeweis [Moh+13]

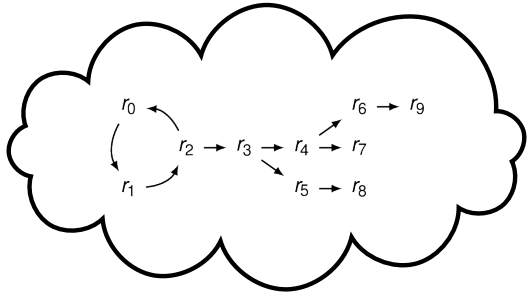
Modifikation komplexer Architektur

Erweiterung einfacher Architektur um RAT



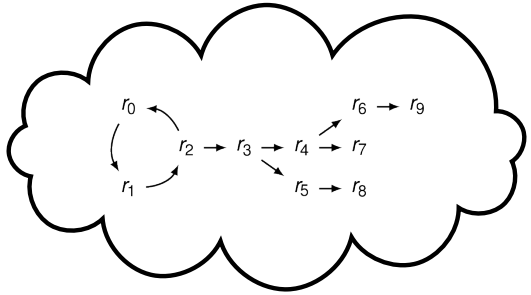
Instruktionssatzerweiterung [Moh+13]

Idealerweise: `rtg`

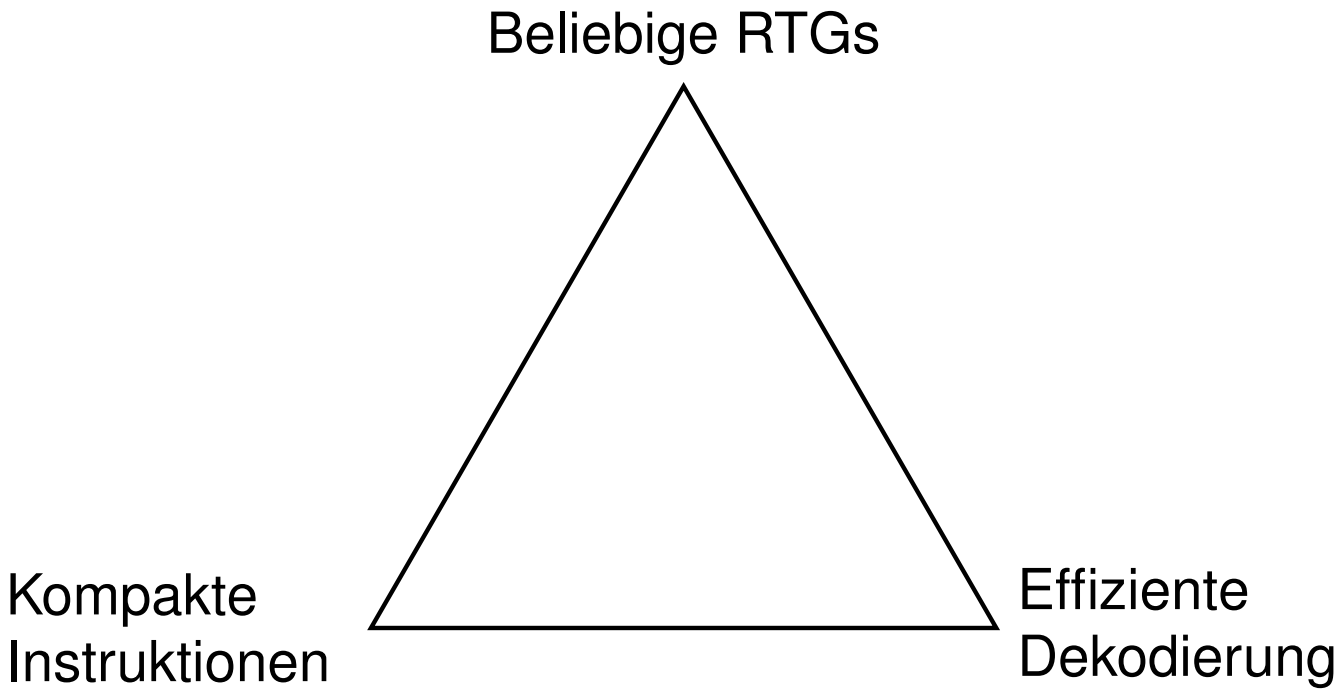


Instruktionssatzerweiterung [Moh+13]

Idealerweise: `rtg`

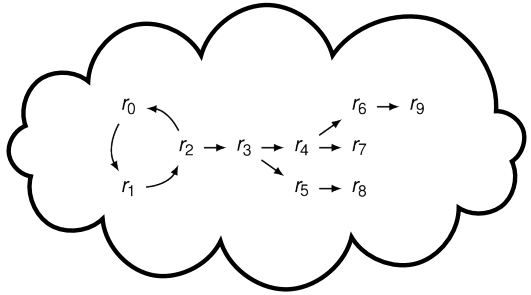


Realität:

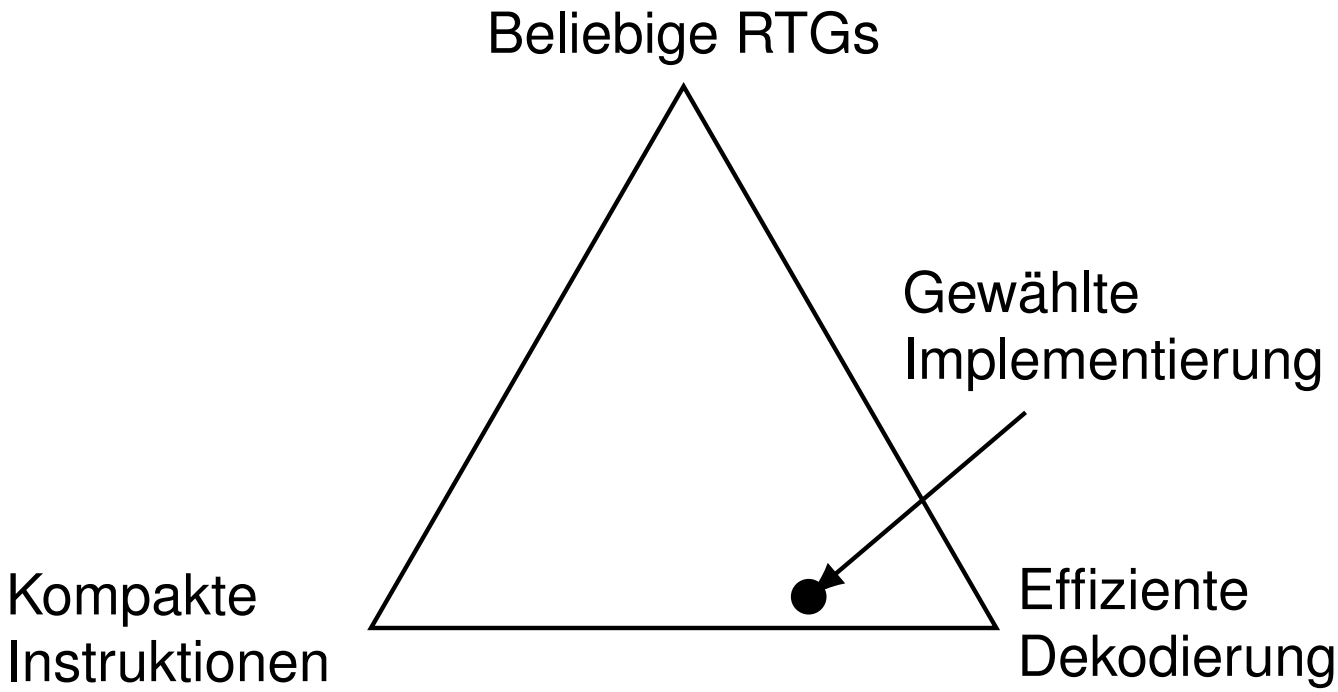


Instruktionssatzerweiterung [Moh+13]

Idealerweise: `rtg`

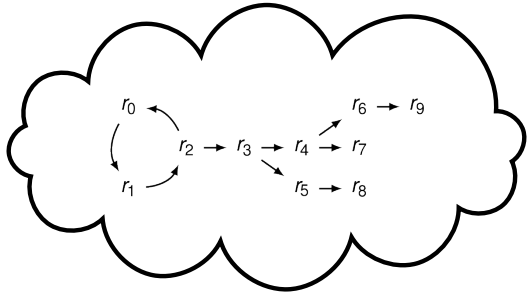


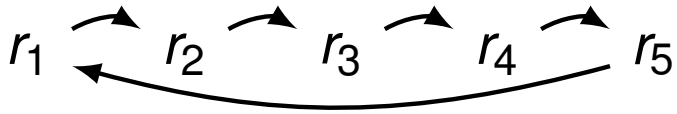
Realität:



Instruktionssatzerweiterung [Moh+13]

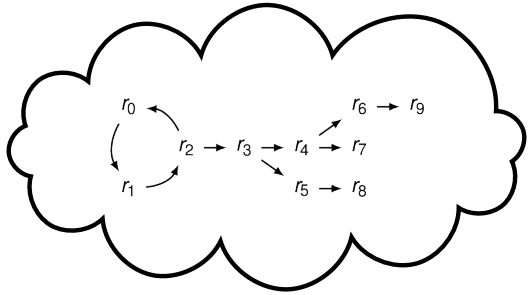
Idealerweise: `rtg`

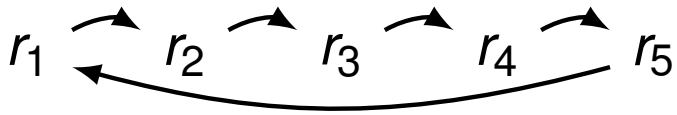




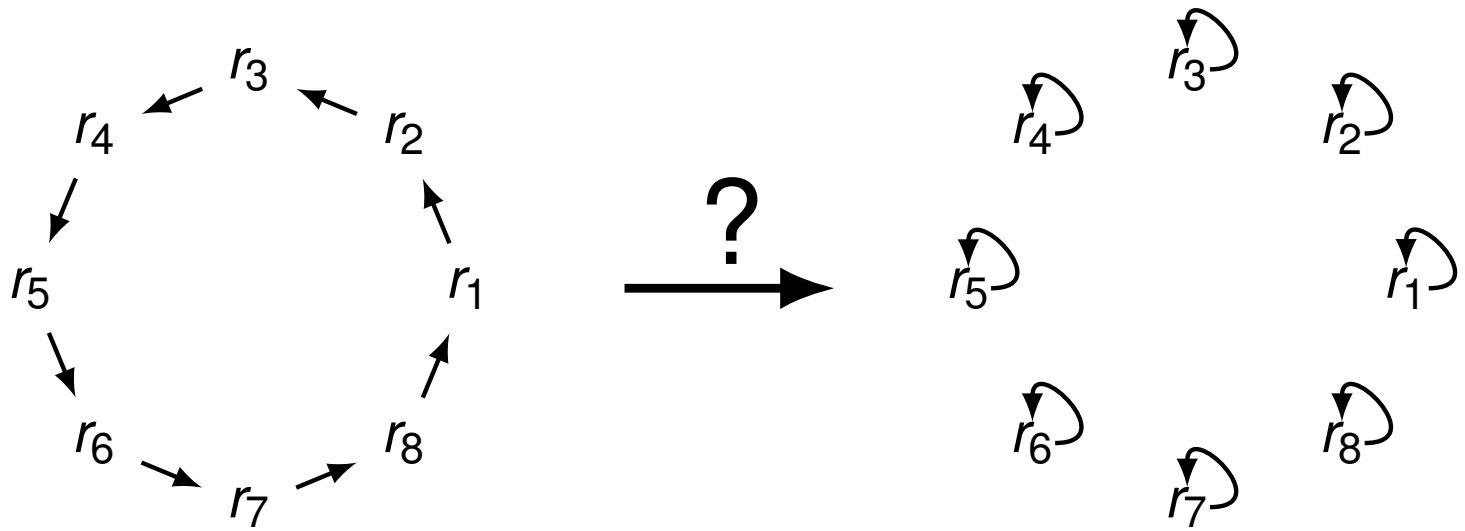
Instruktionssatzerweiterung [Moh+13]

Idealerweise: `rtg`



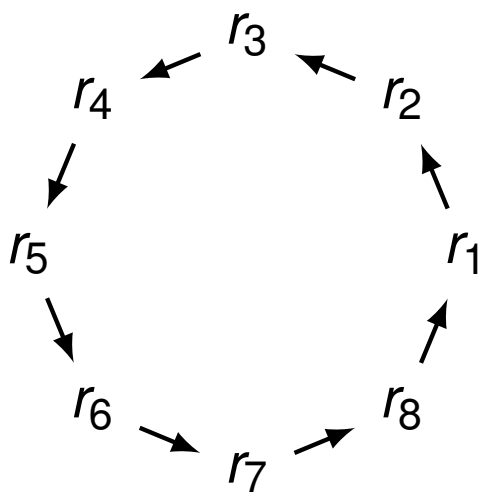


Codeerzeugung: Was ist daran schwierig?

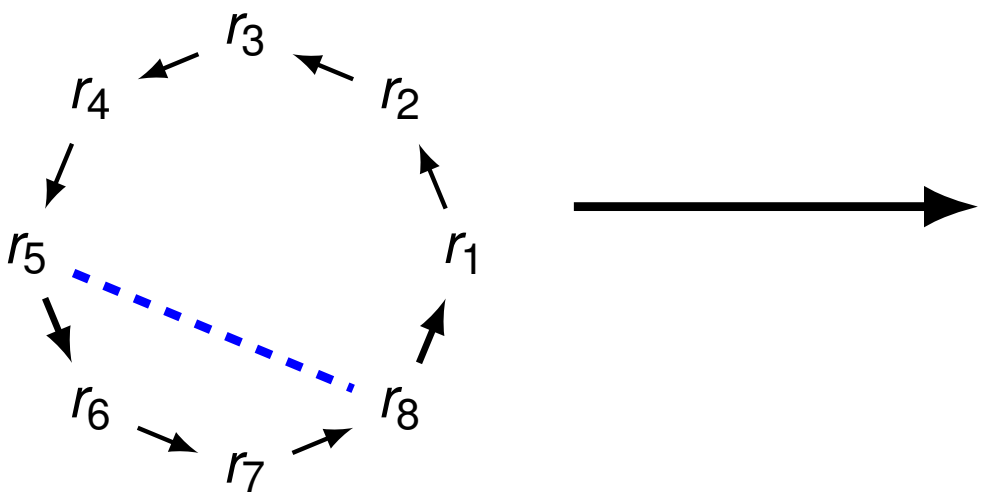


Trivialer RTG

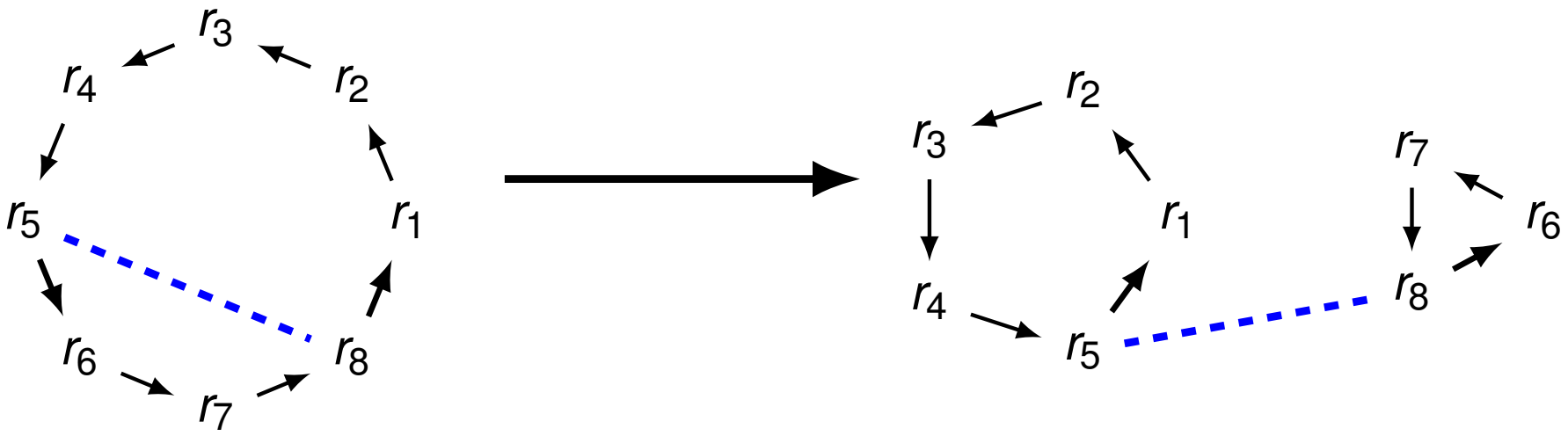
Codeerzeugung: Was ist daran schwierig?



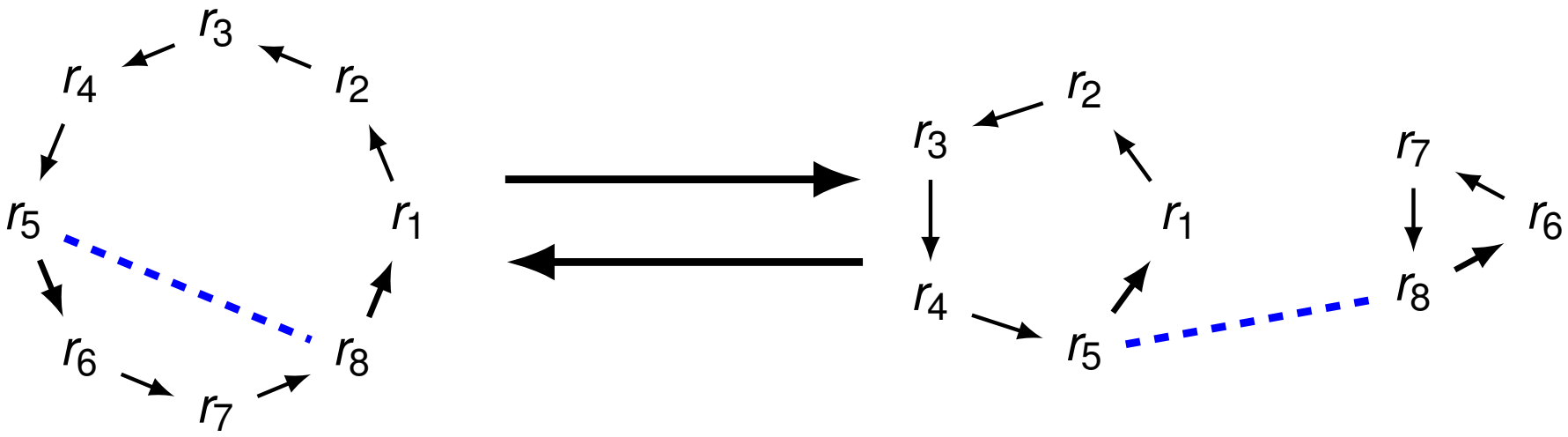
Codeerzeugung: Was ist daran schwierig?



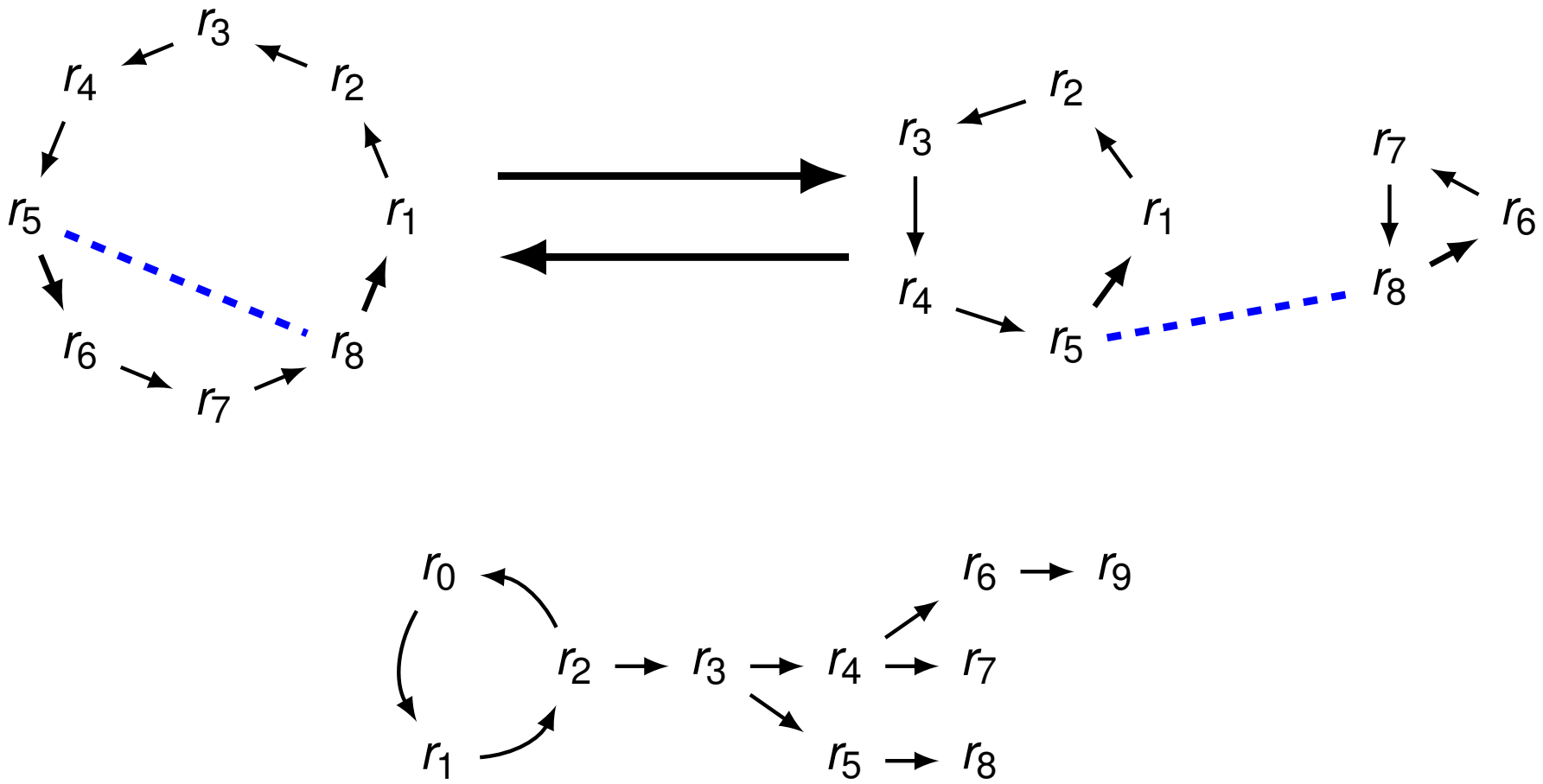
Codeerzeugung: Was ist daran schwierig?



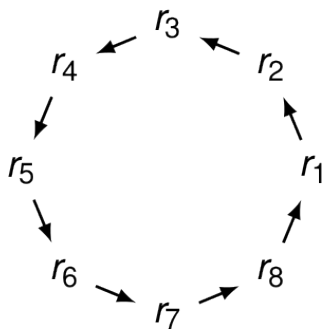
Codeerzeugung: Was ist daran schwierig?

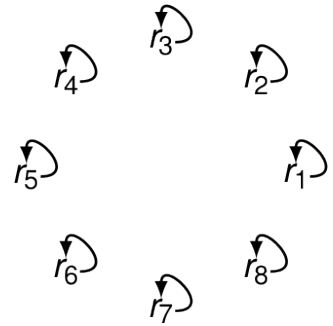


Codeerzeugung: Was ist daran schwierig?



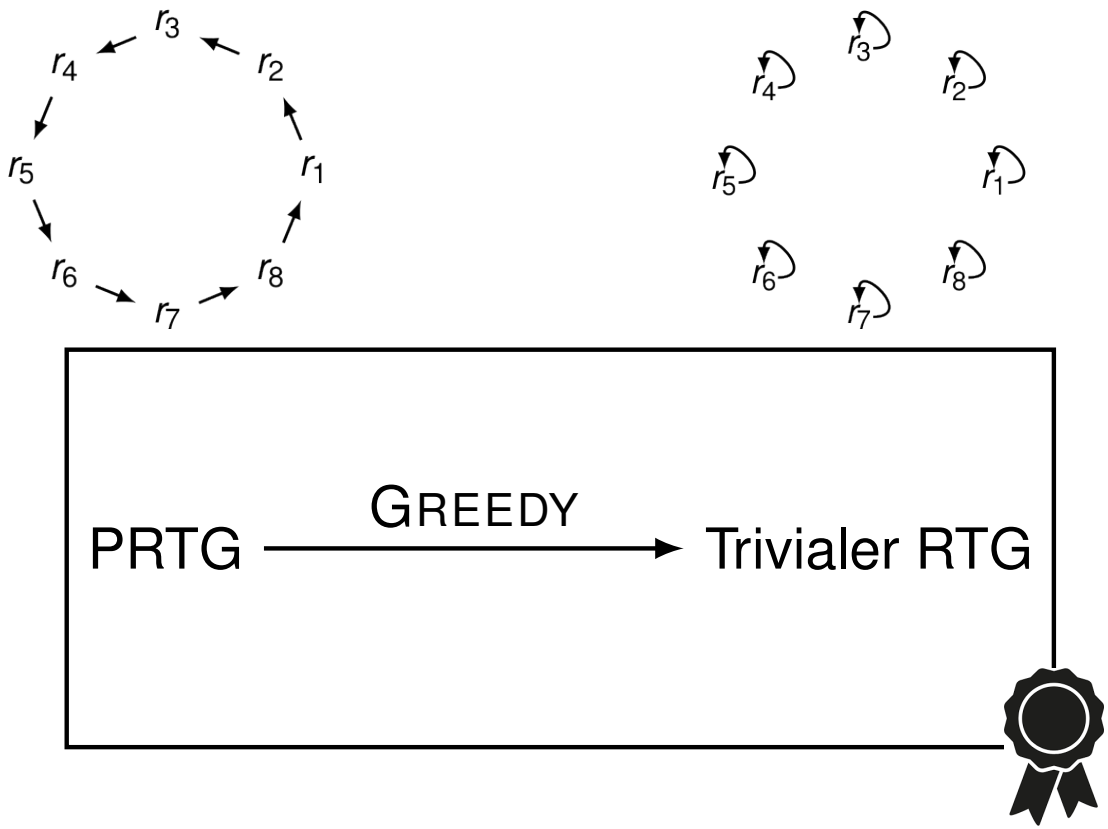
Codeerzeugung [BMR15b; BMR15a]



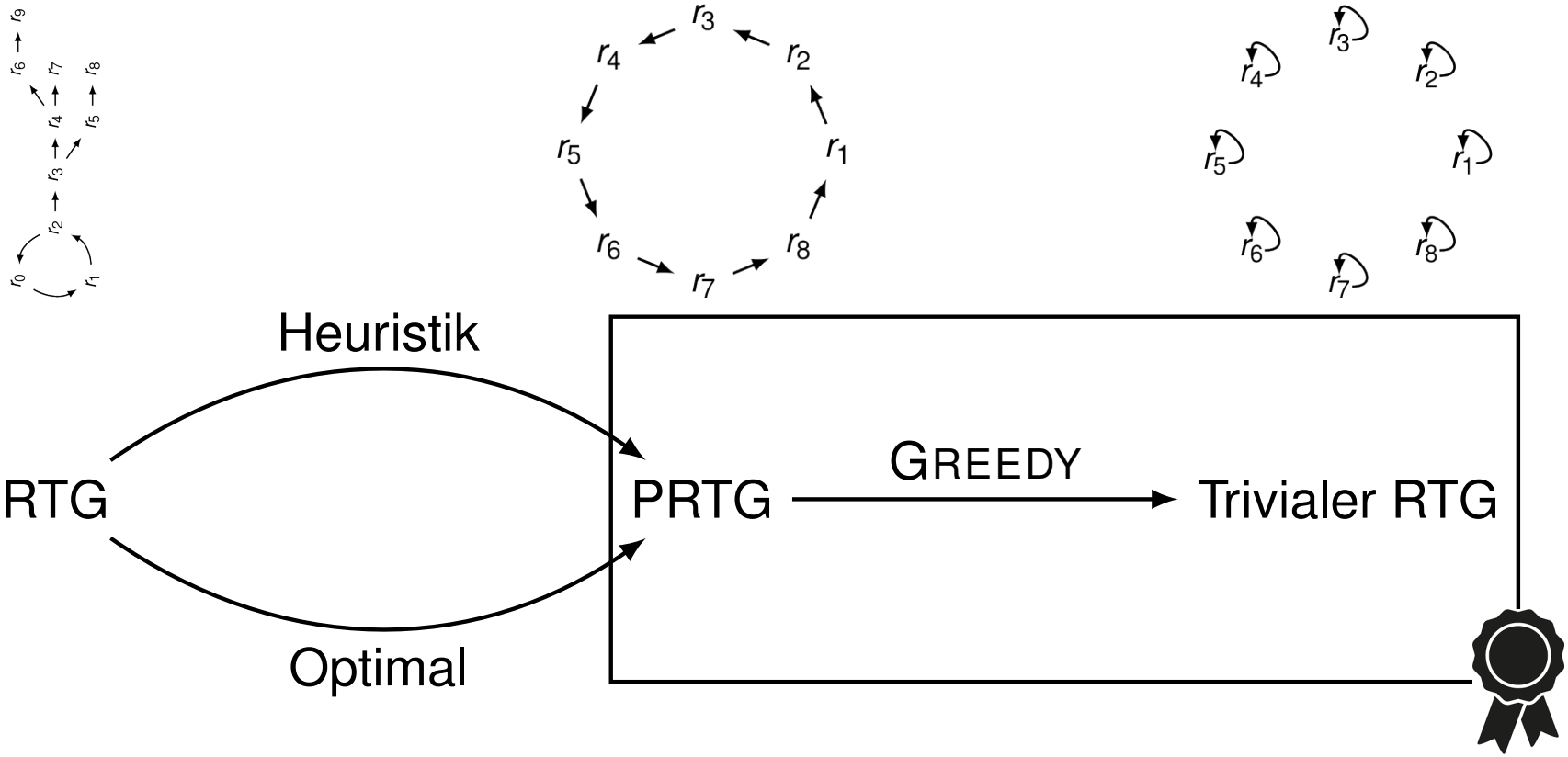


PRTG $\xrightarrow{\text{GREEDY}}$ Trivialer RTG

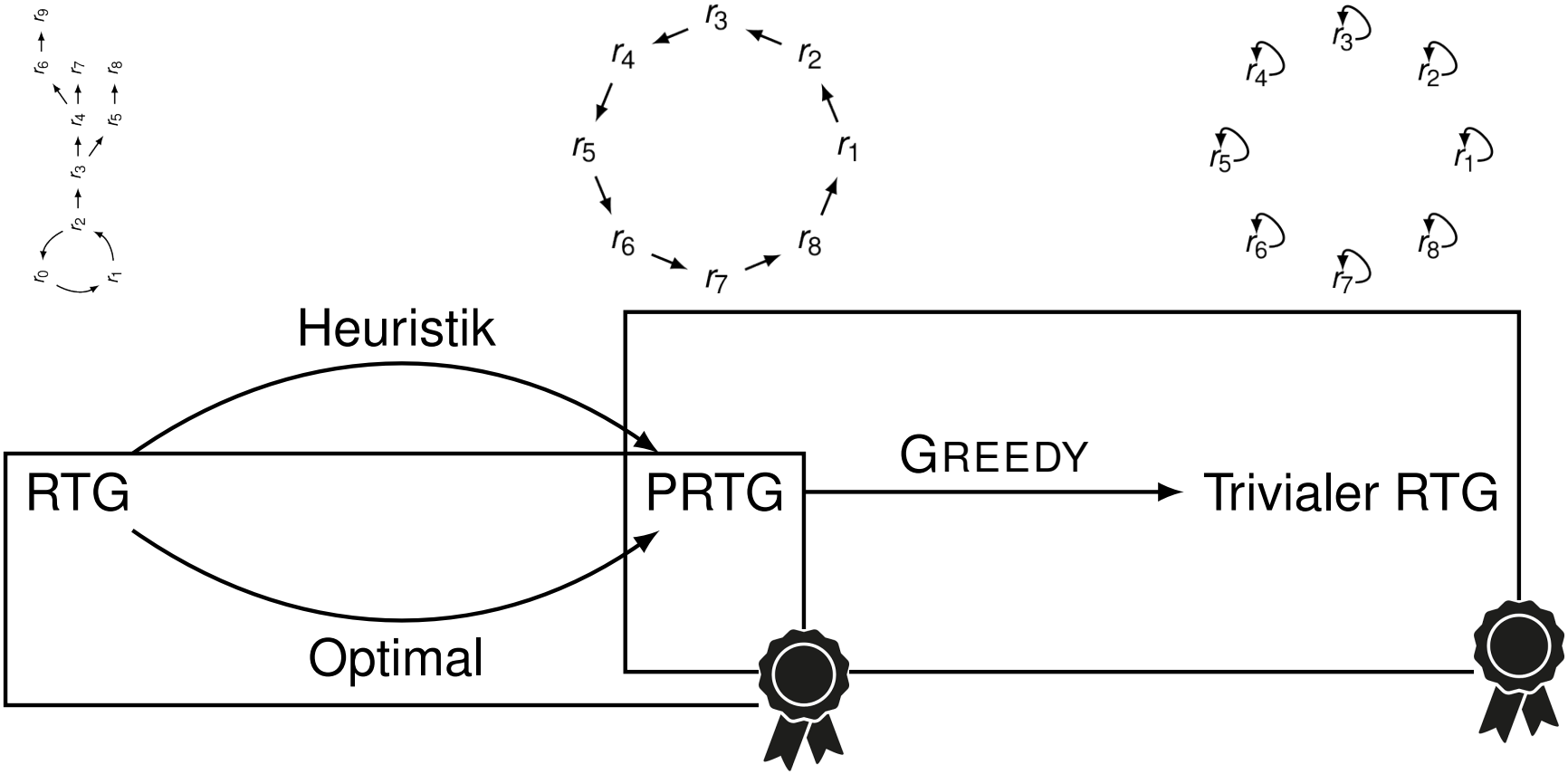
Codeerzeugung [BMR15b; BMR15a]



Codeerzeugung [BMR15b; BMR15a]



Codeerzeugung [BMR15b; BMR15a]



Ergebnisse [Moh+13; Moh18]

- Praxis: Heuristik optimal ($\Delta \leq$ 0,24% bzgl. #Instruktionen)
- Praxis: Compiler nicht langsamer ($\Delta \leq$ 0,25% bzgl. Gesamtlaufzeit)

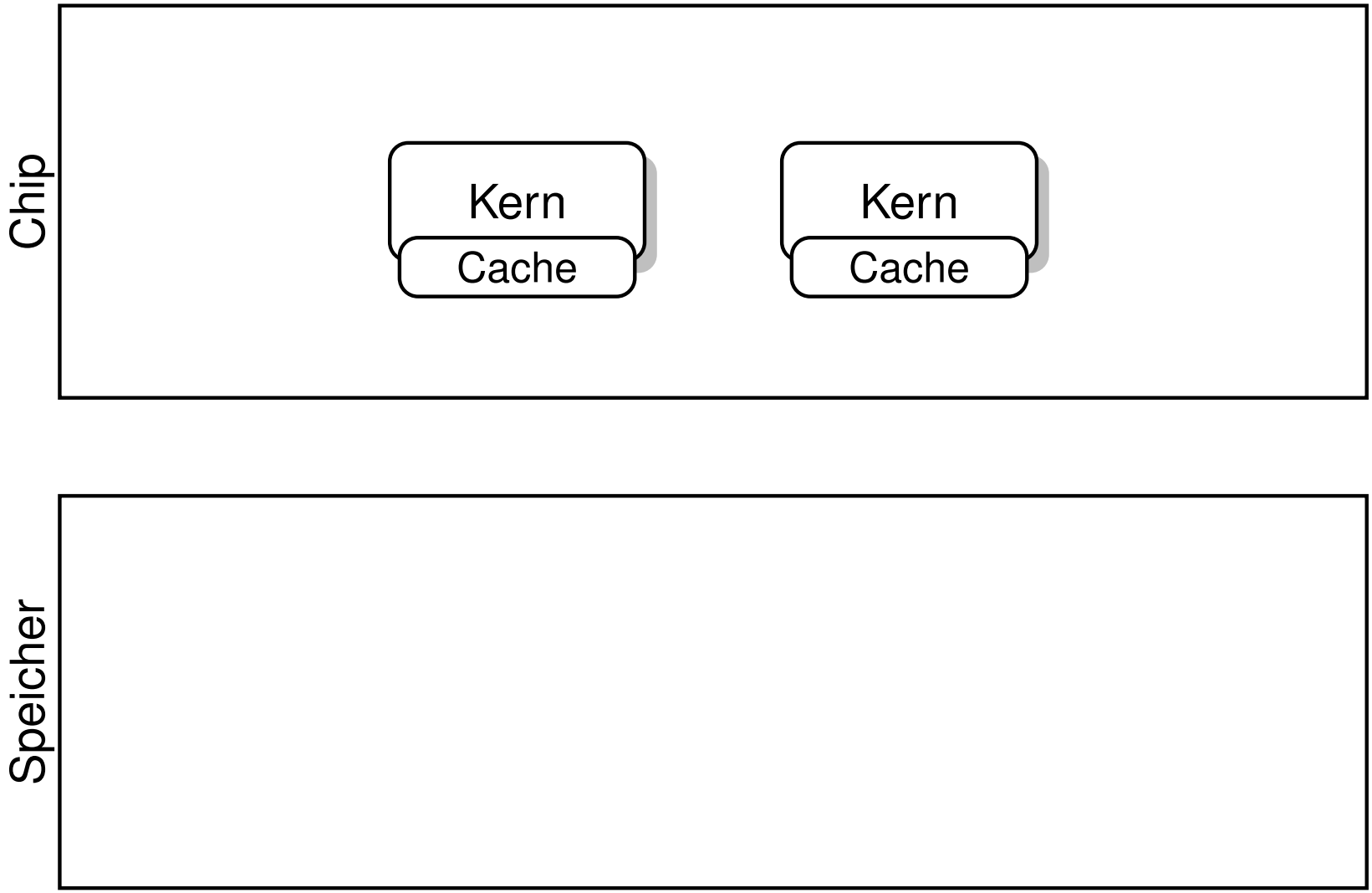
Ergebnisse [Moh+13; Moh18]

- Praxis: Heuristik optimal ($\Delta \leq 0{,}24\%$ bzgl. #Instruktionen)
- Praxis: Compiler nicht langsamer ($\Delta \leq 0{,}25\%$ bzgl. Gesamtlaufzeit)
- Nutzen abhängig von Qualität der Registerallokation

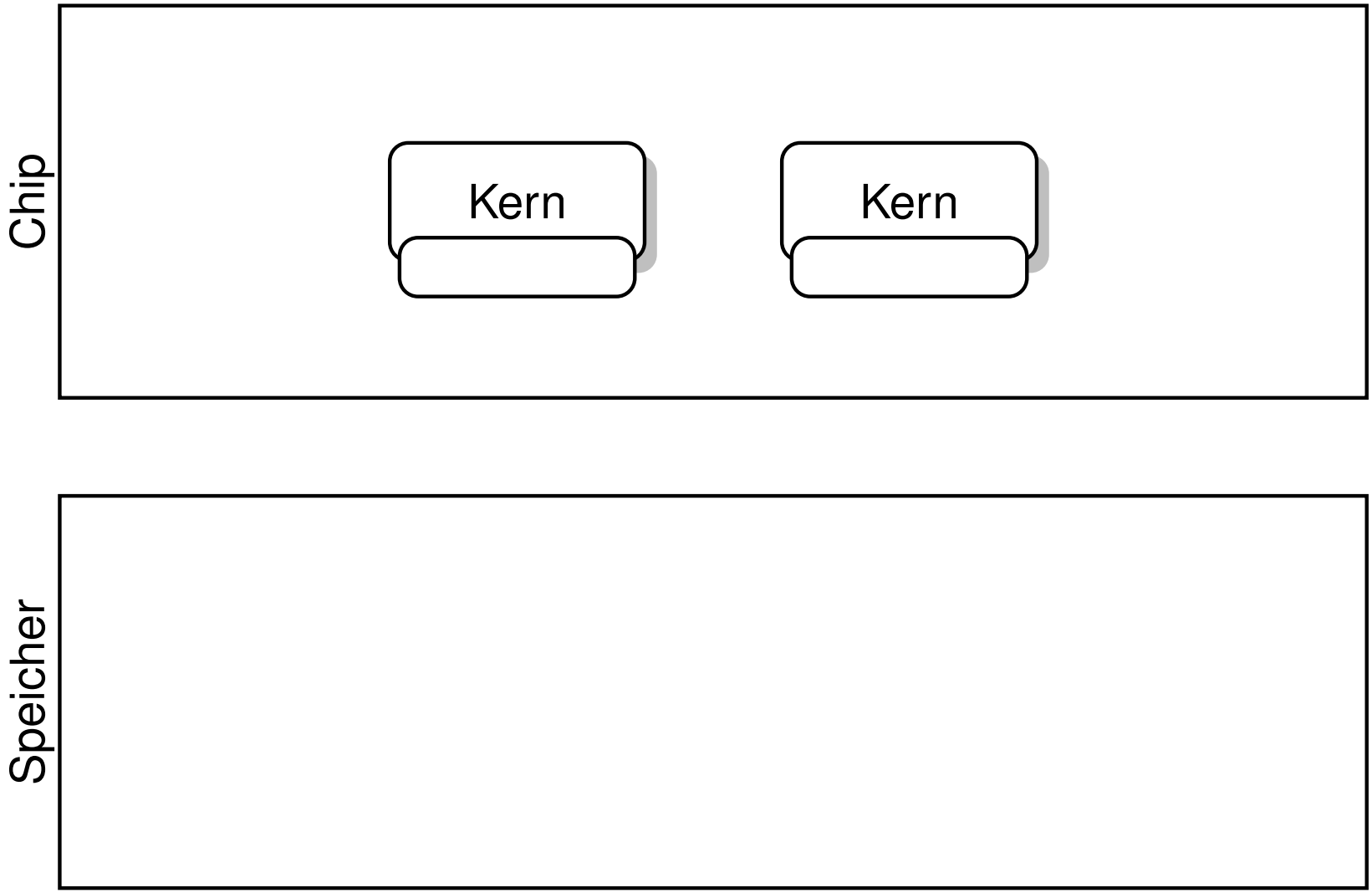
Reduktion	Registerallokation	
	Optimal	JIT
# ausgef. Inst.	bis zu 1,9% (ø0,5%)	bis zu 5,1% (ø2,2%)
Laufzeit	bis zu 2,4% (ø0,5%)	bis zu 7,0% (ø2,4%)


Beitrag 2:
Datentransfers mit inkohärenten Caches
Uniform
Nicht-uniform
Inkohärent
Disjunkt
Speicher

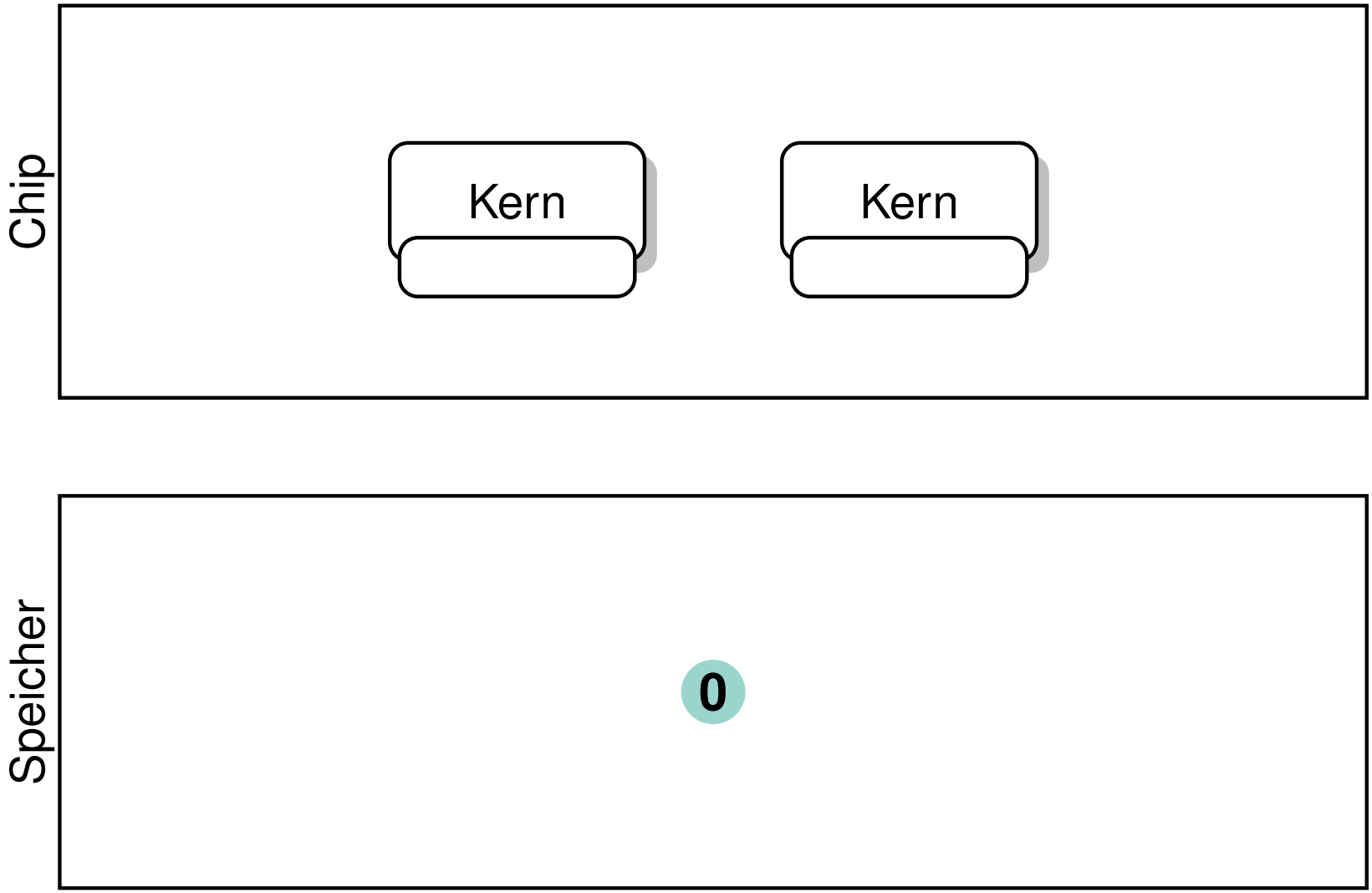
Motivation



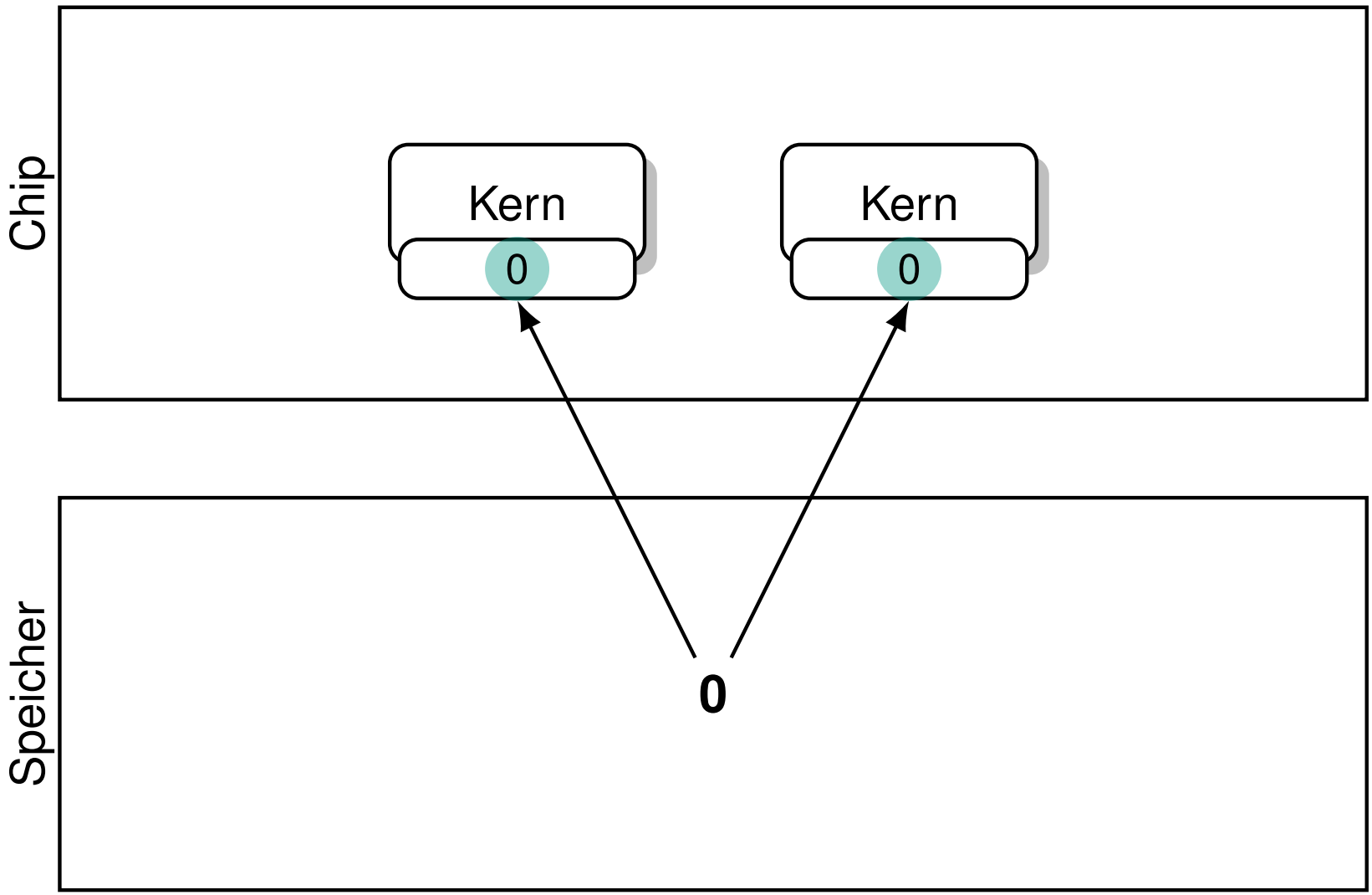
Motivation



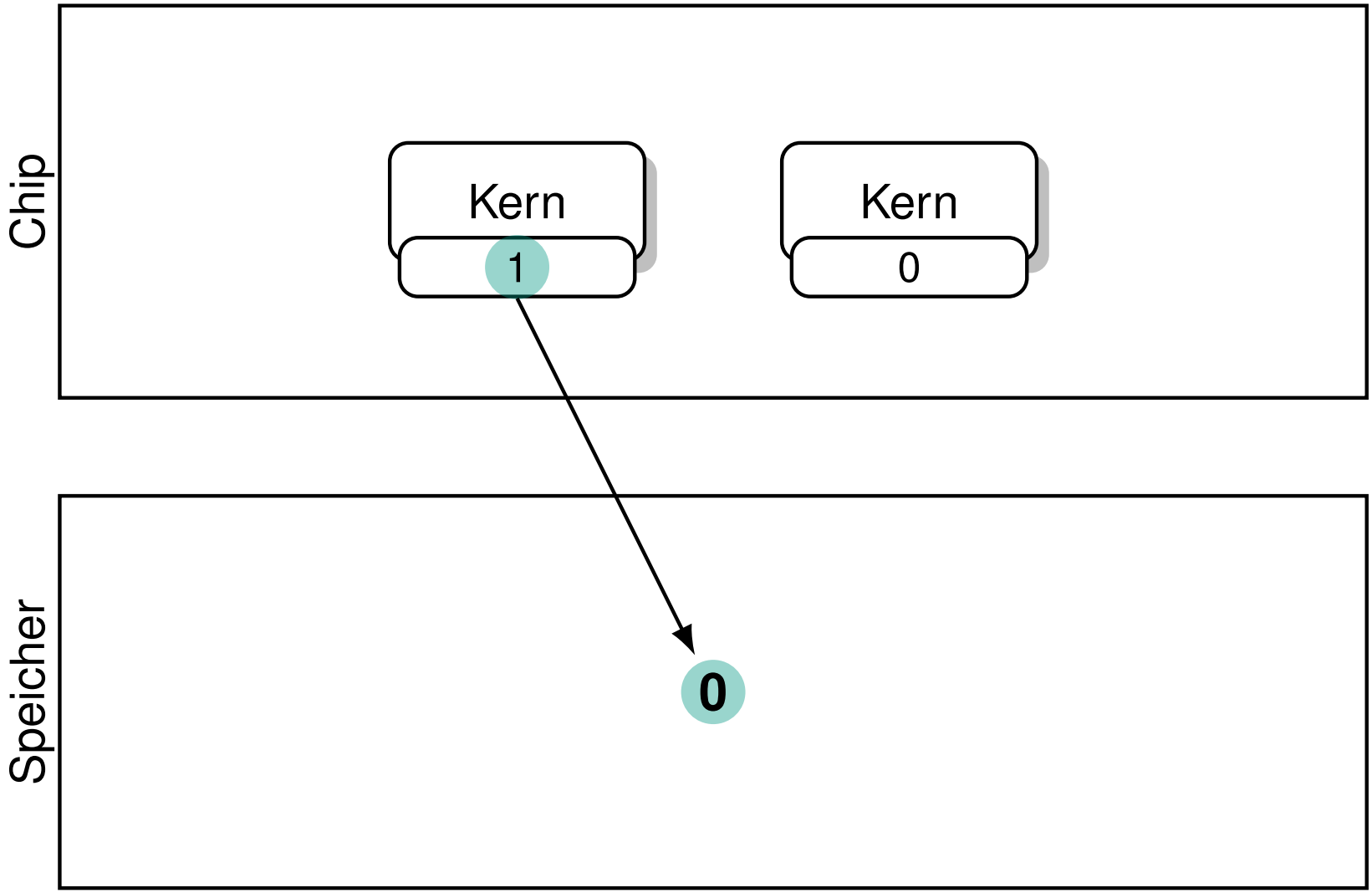
Motivation



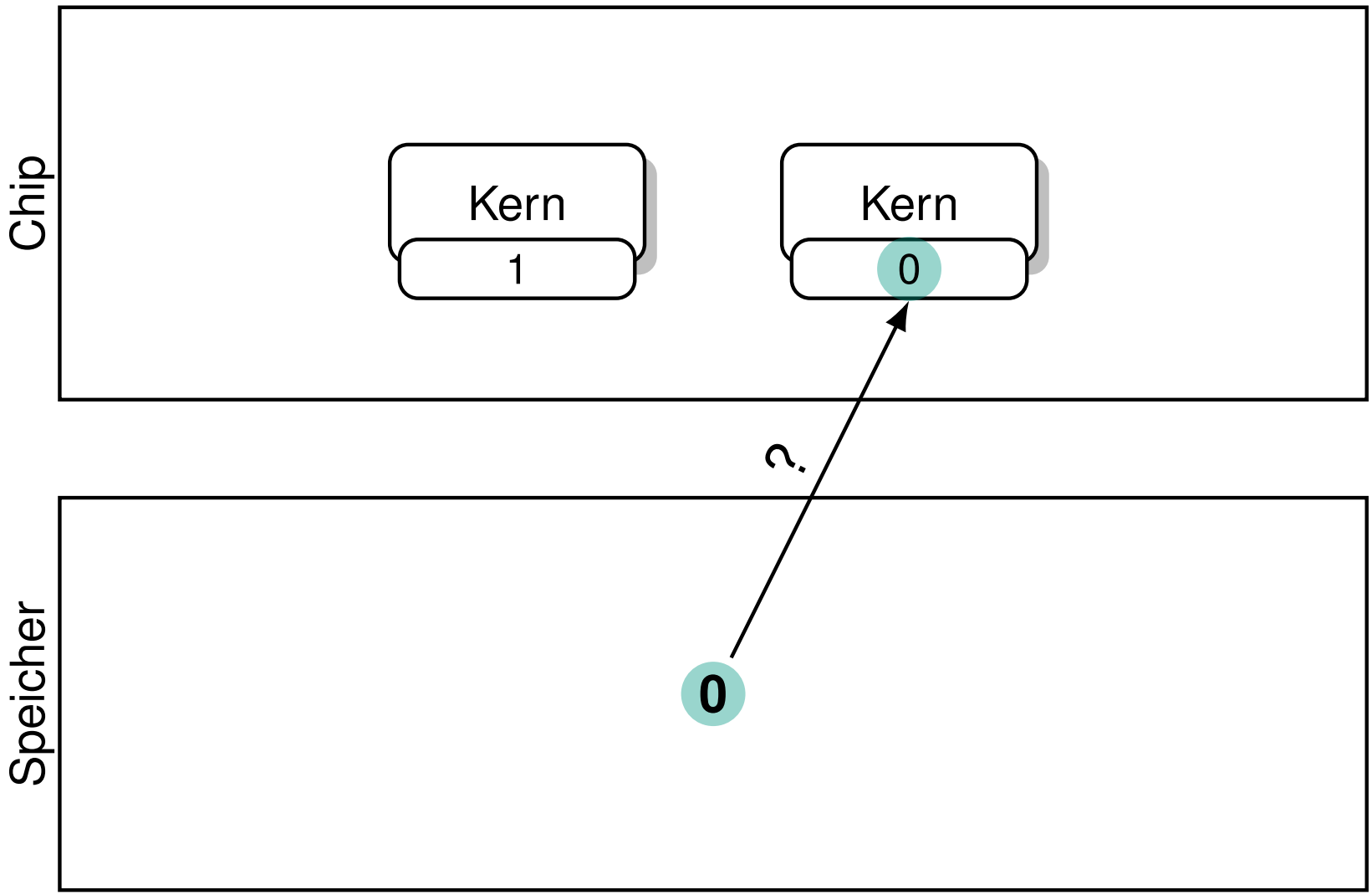
Motivation



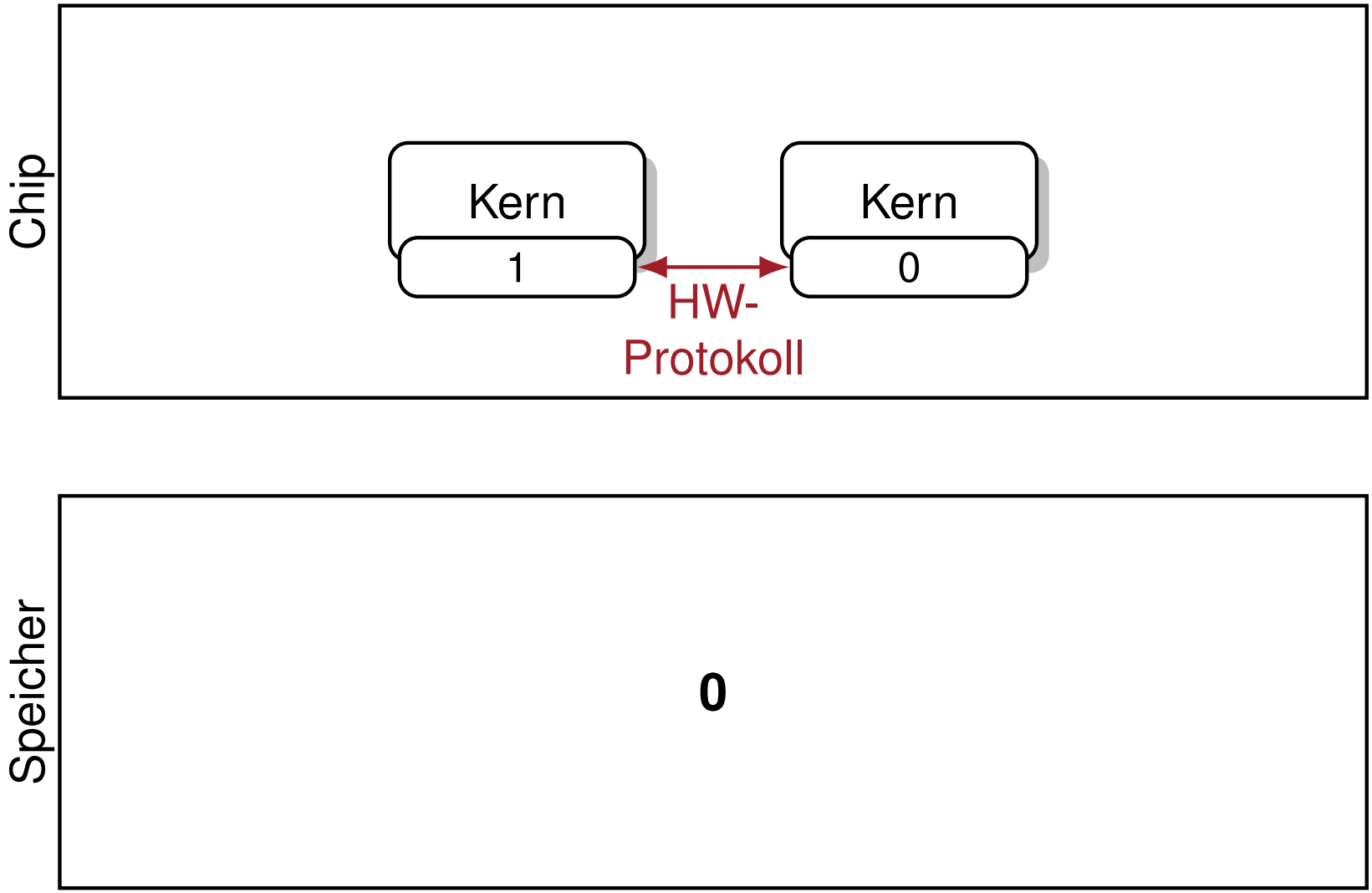
Motivation



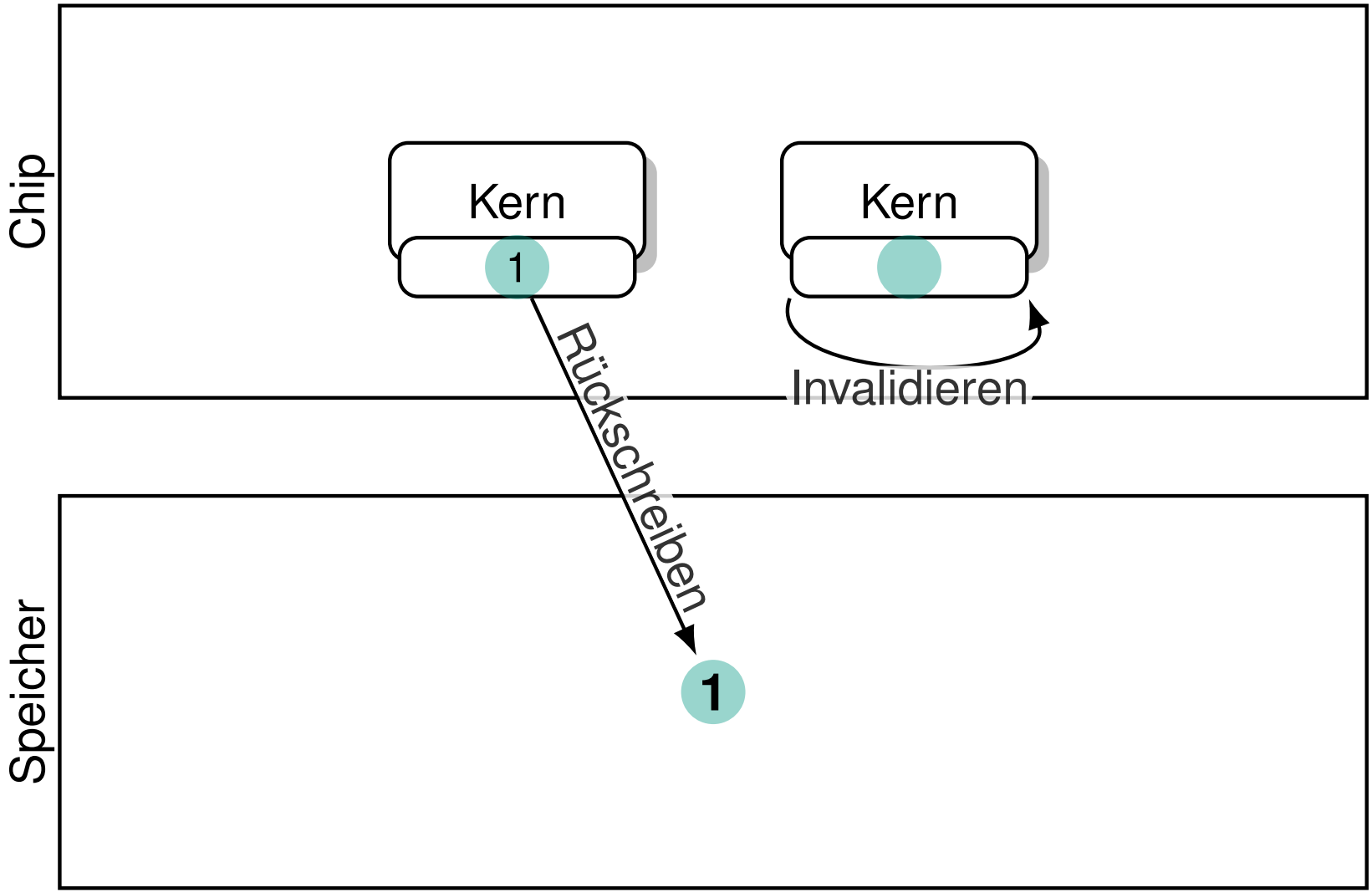
Motivation



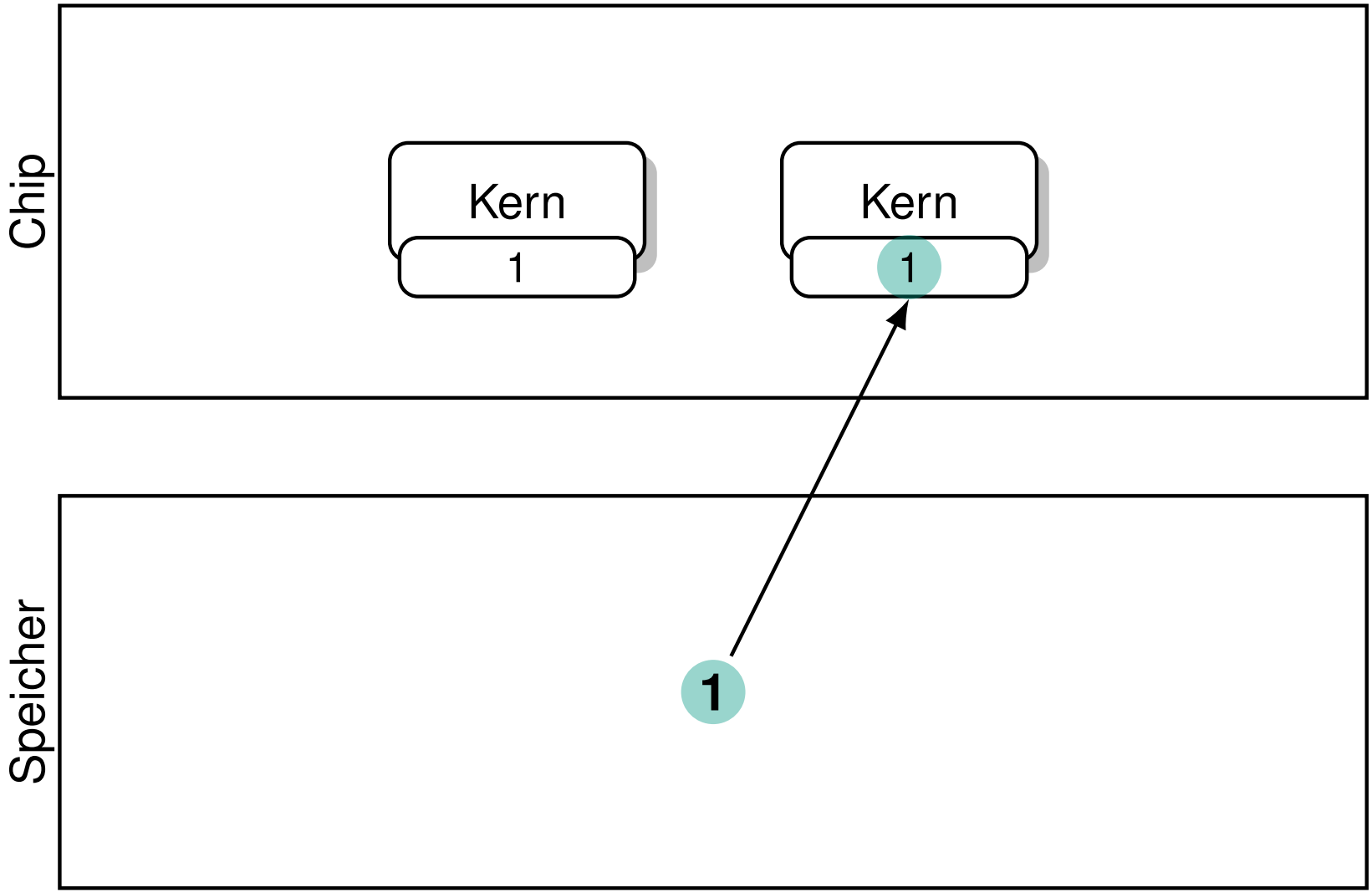
Motivation



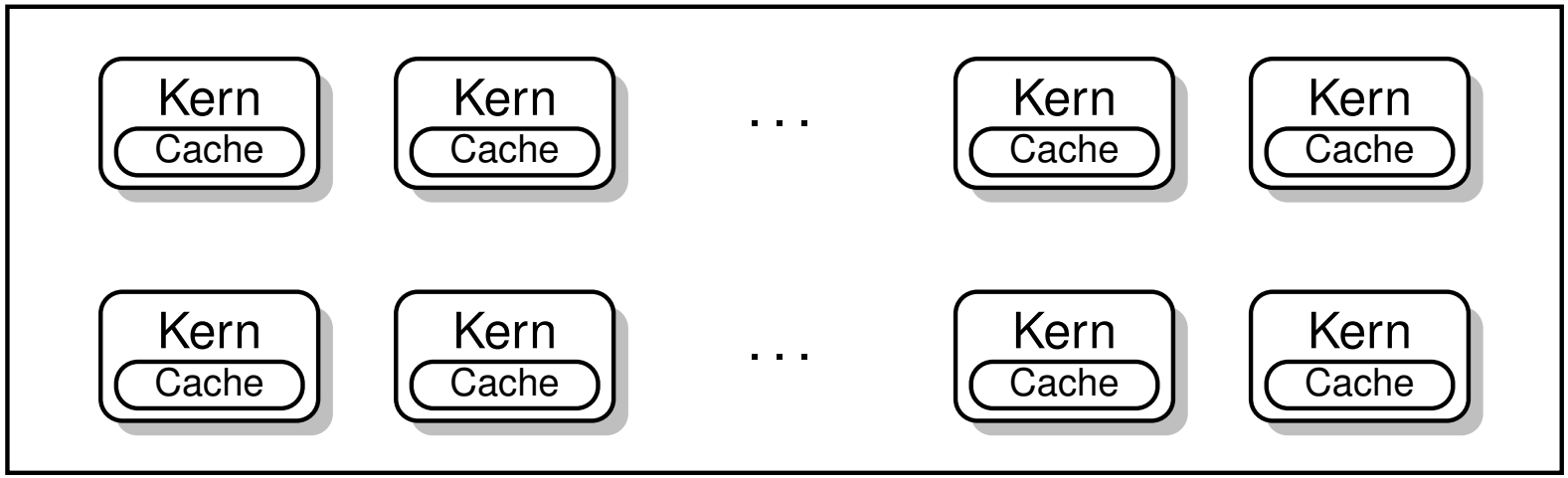
Motivation

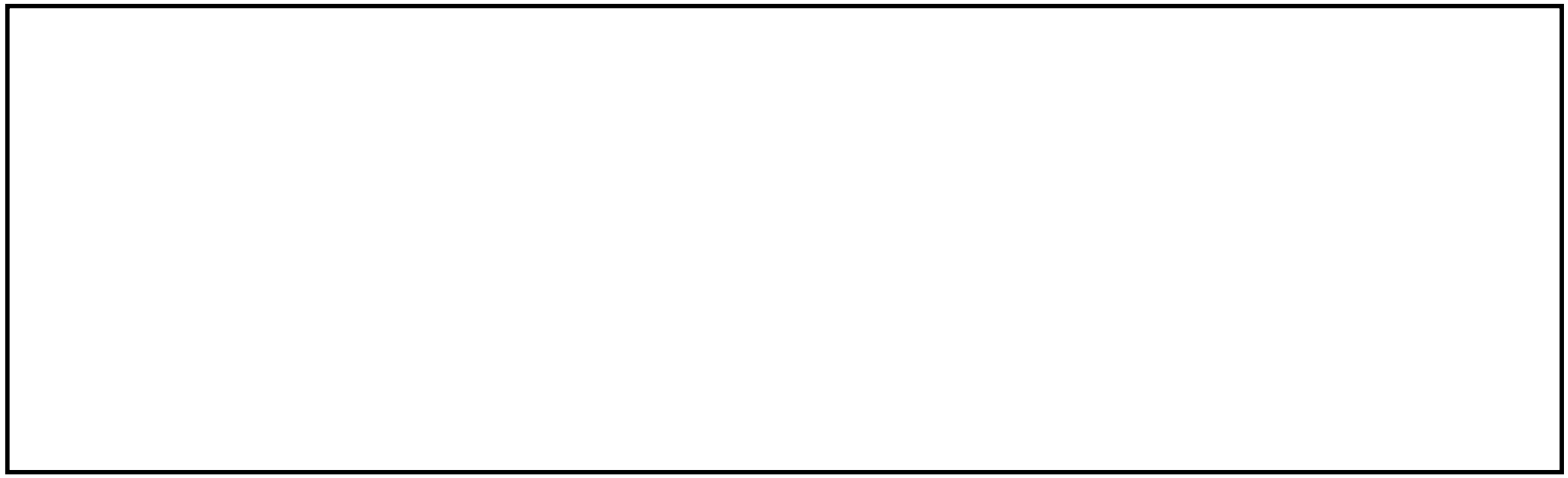


Motivation

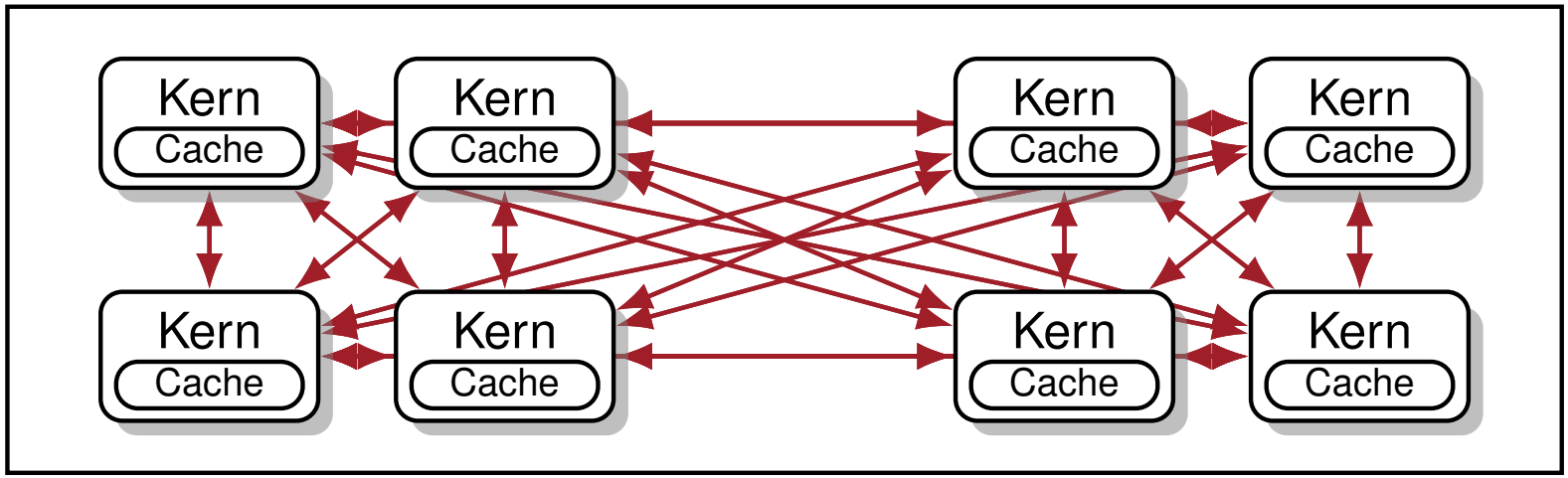


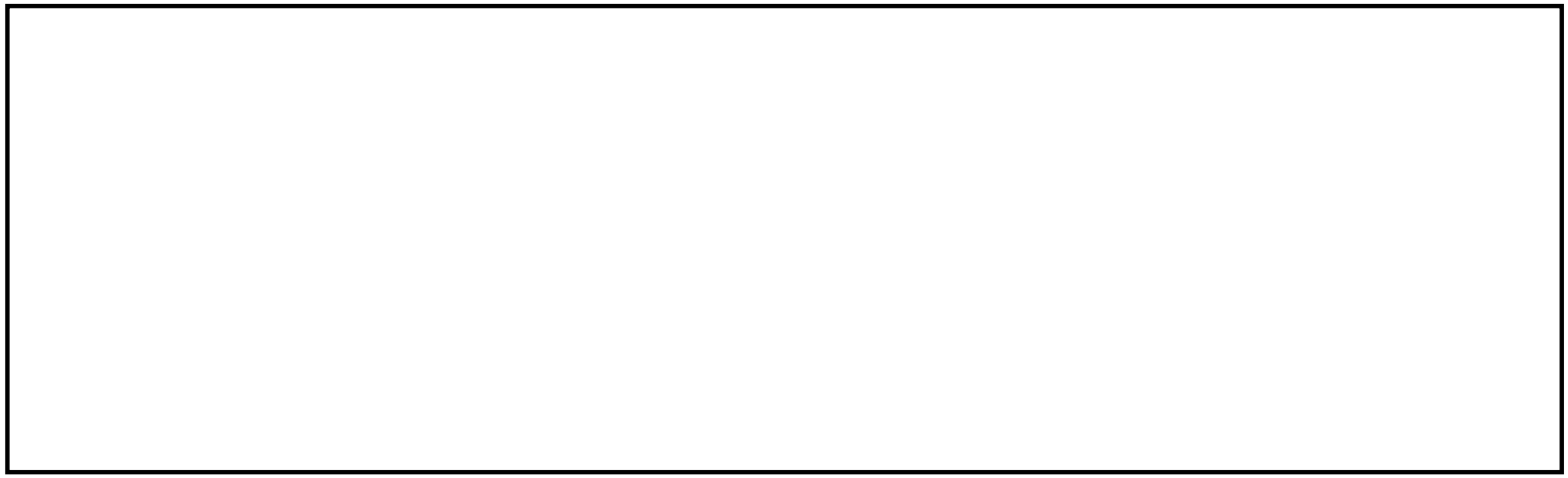
Motivation


Kern
Cache
Kern
Cache
. . .
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
. . .
Kern
Cache
Kern
Cache

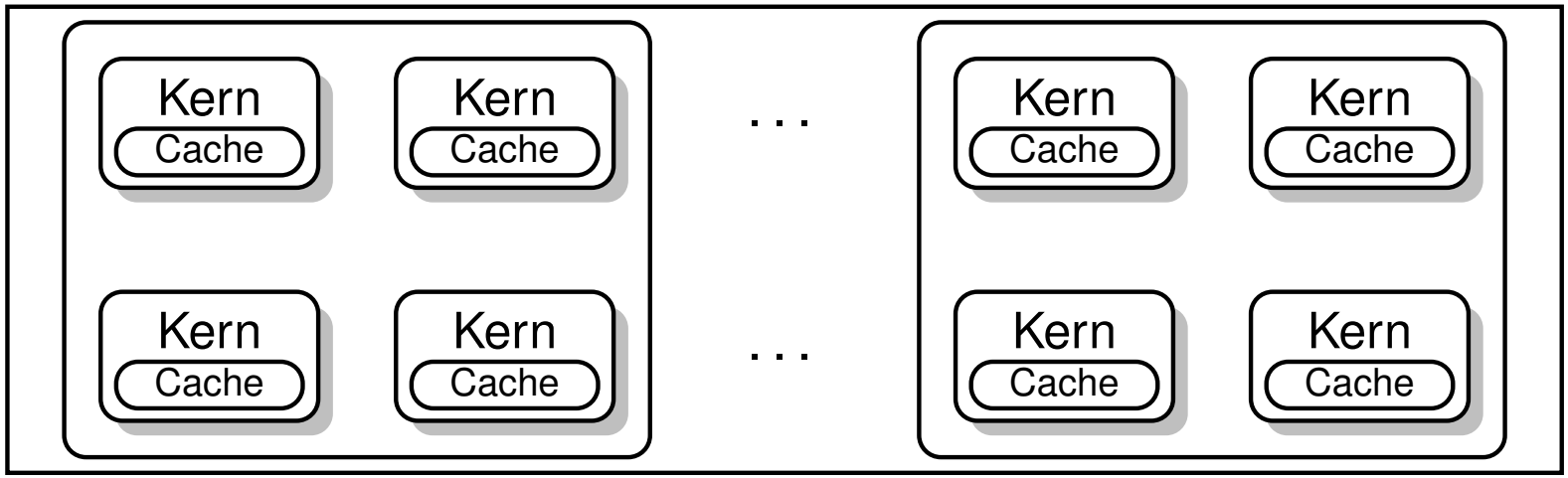


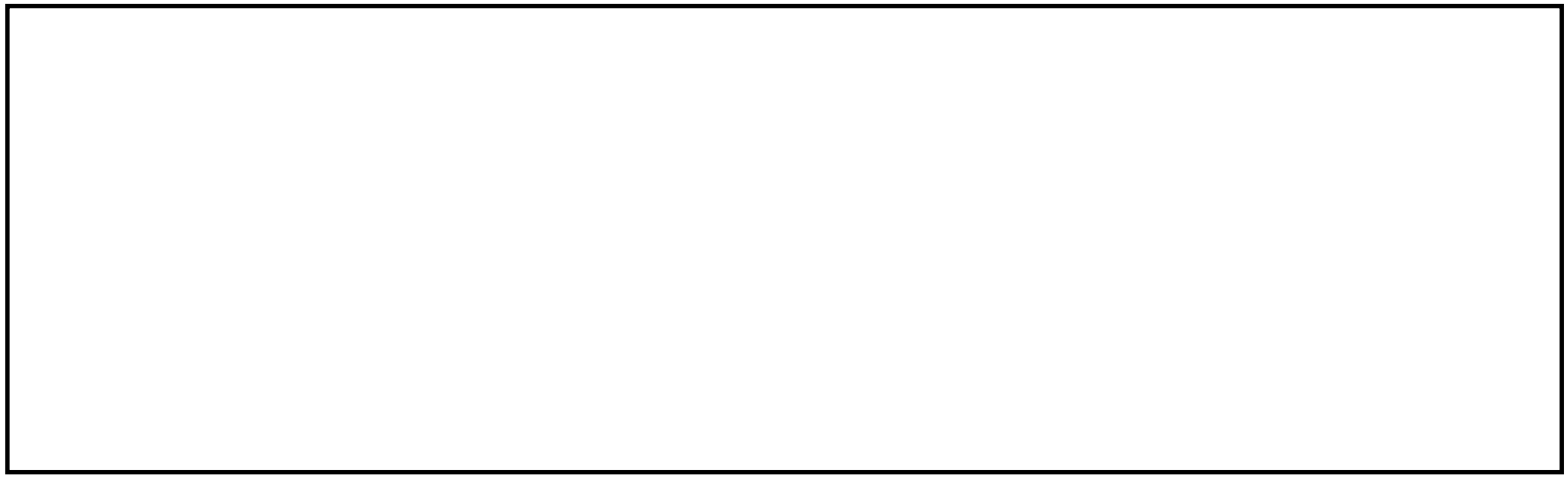
Motivation


Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache

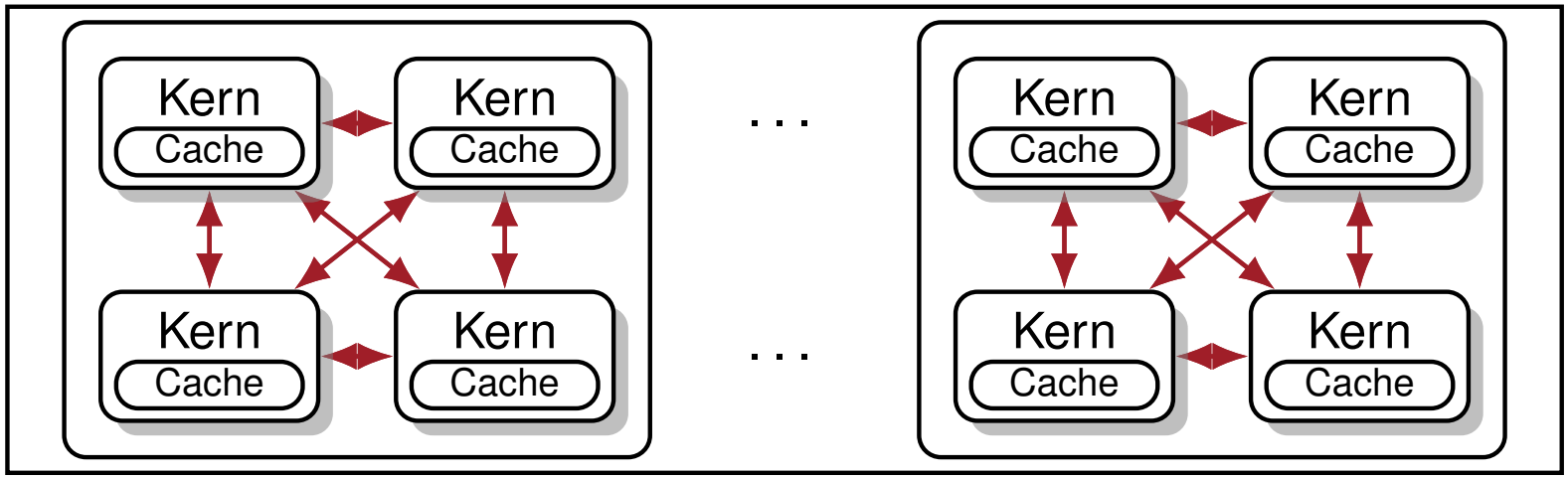


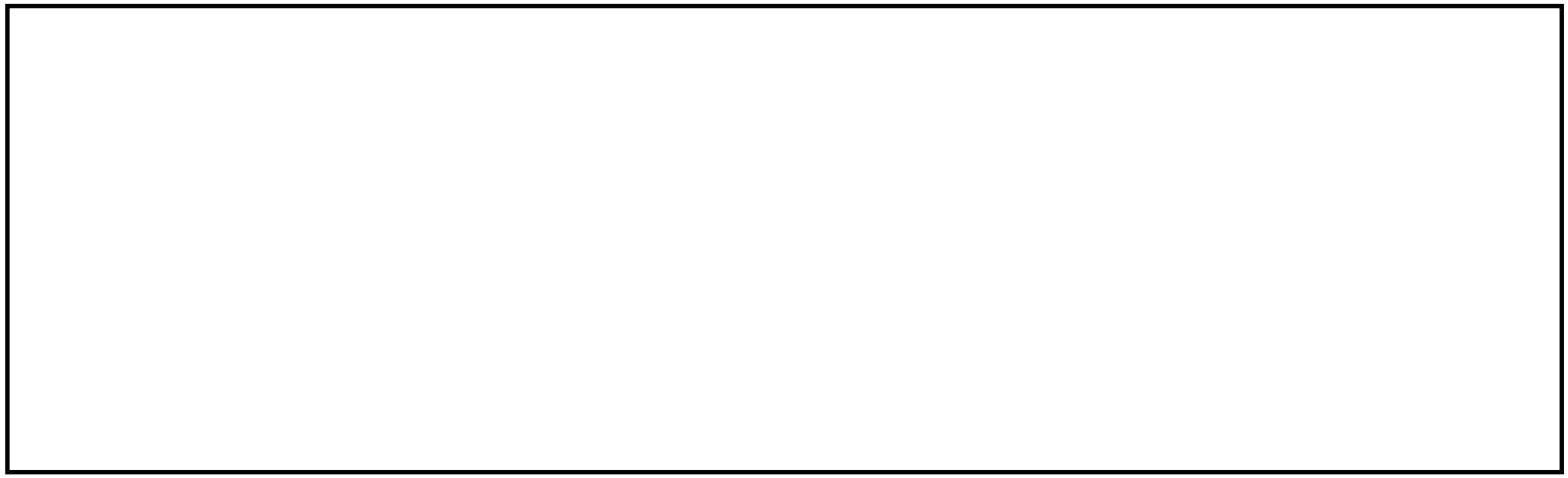
Motivation


Kern
Cache
Kern
Cache
. . .
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
. . .
Kern
Cache
Kern
Cache

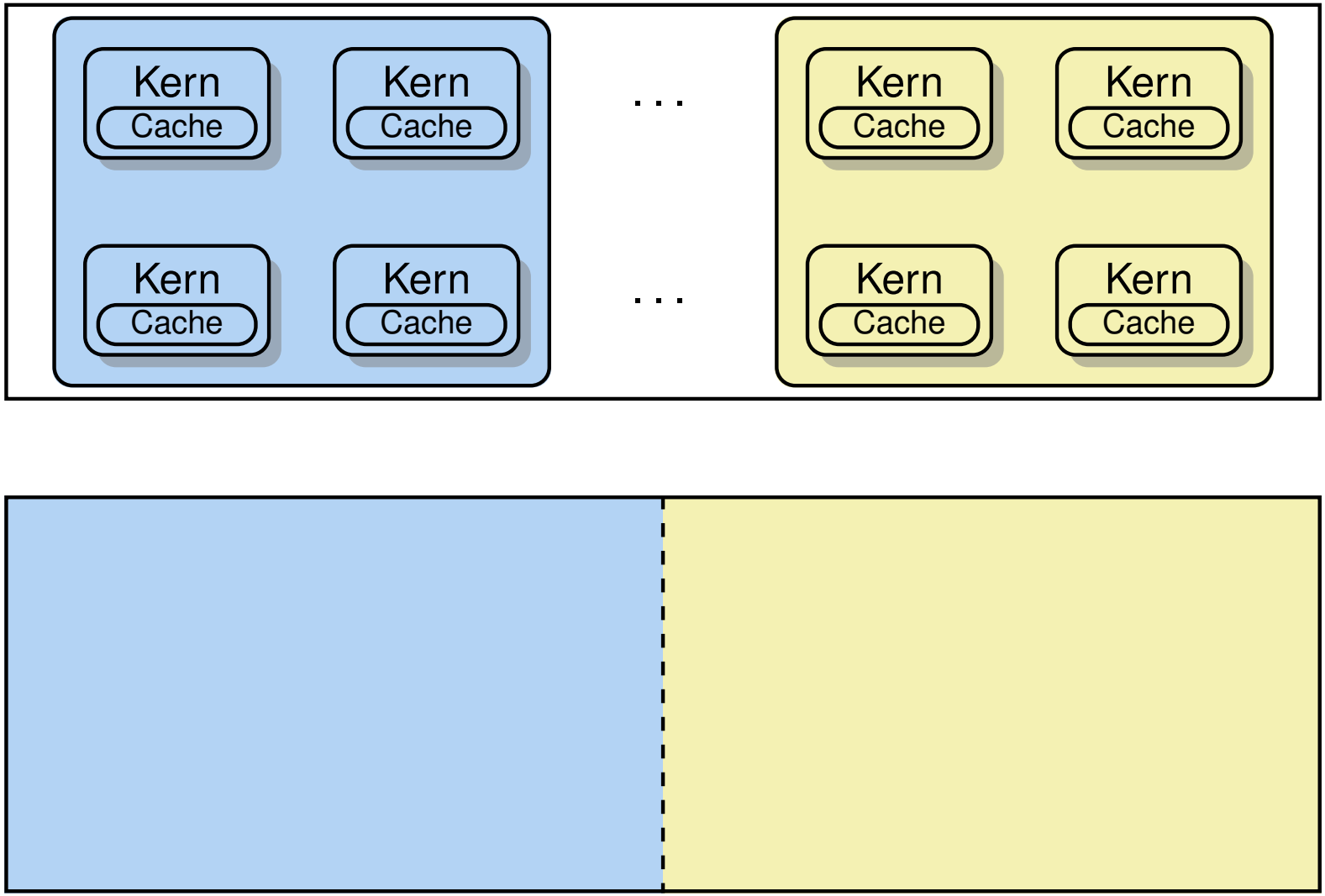


Motivation

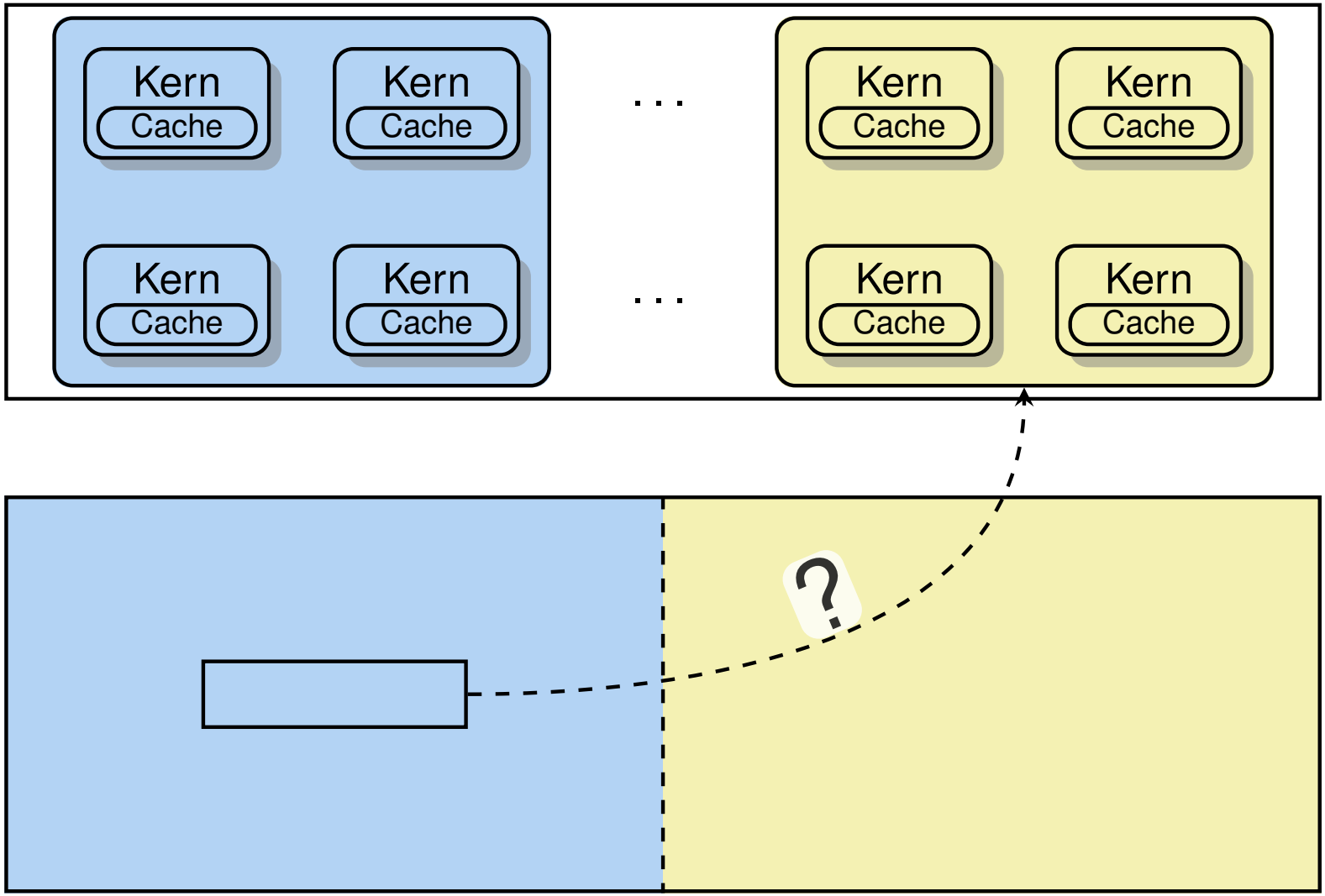

Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache



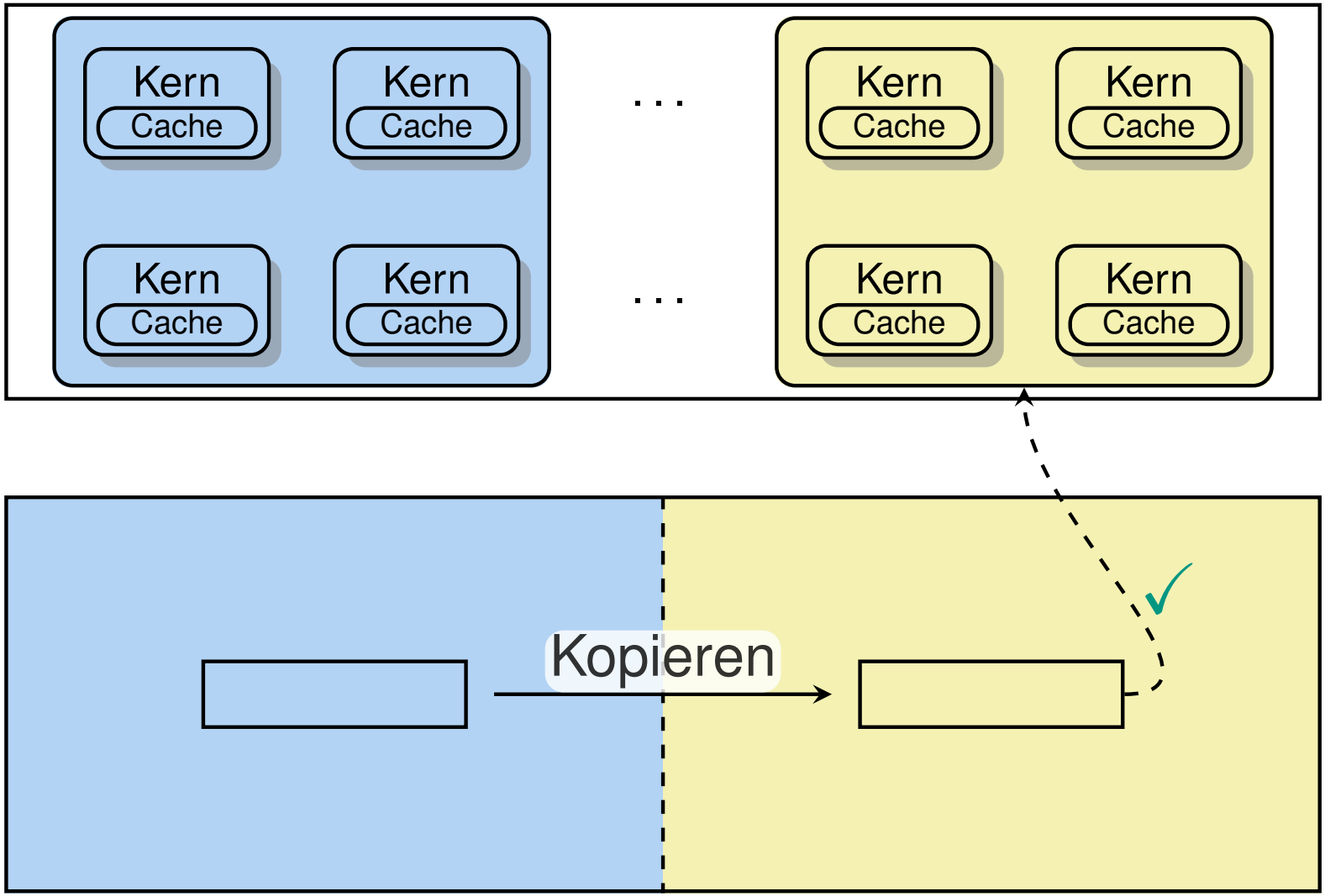
Motivation


Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache

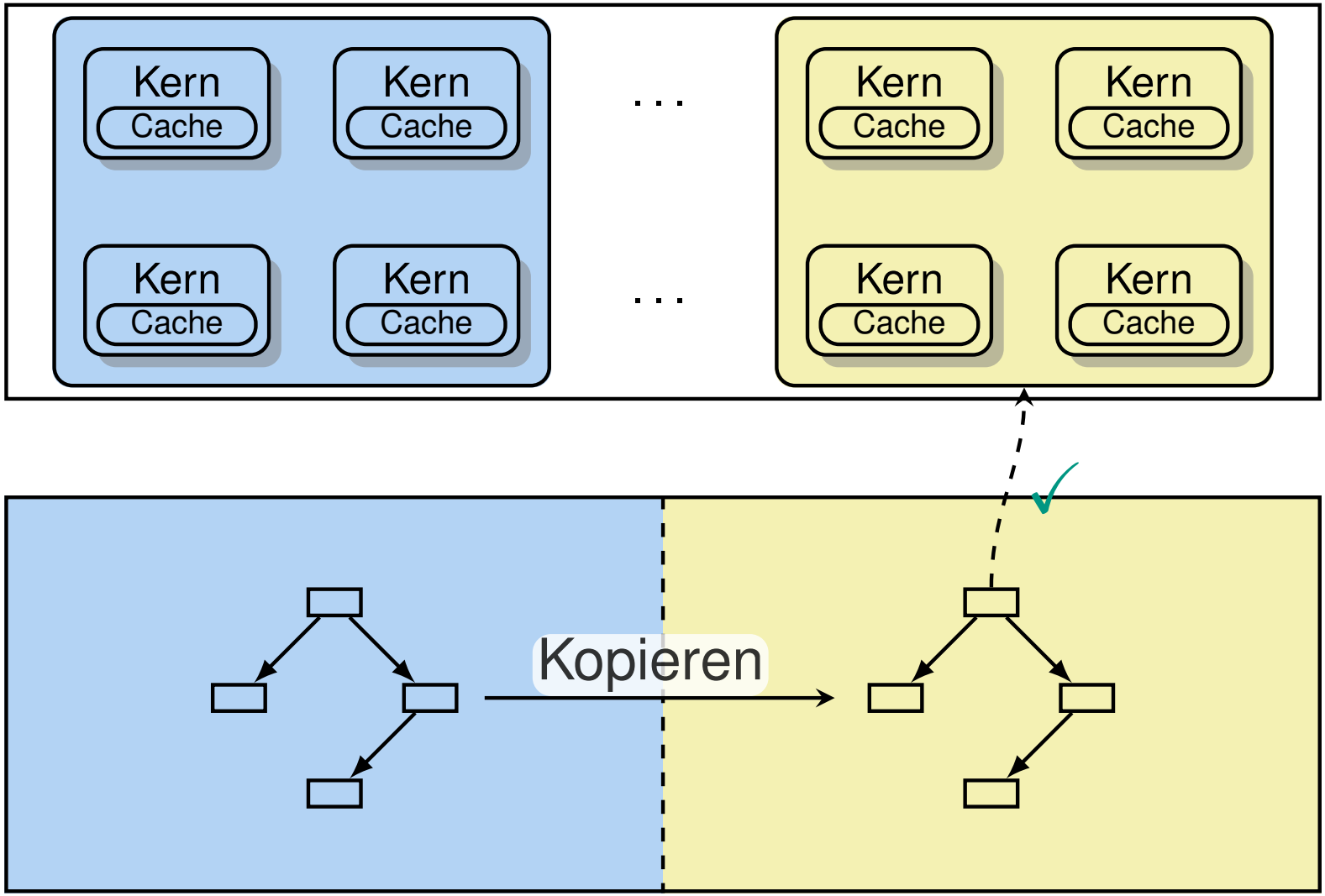
Motivation


Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
?

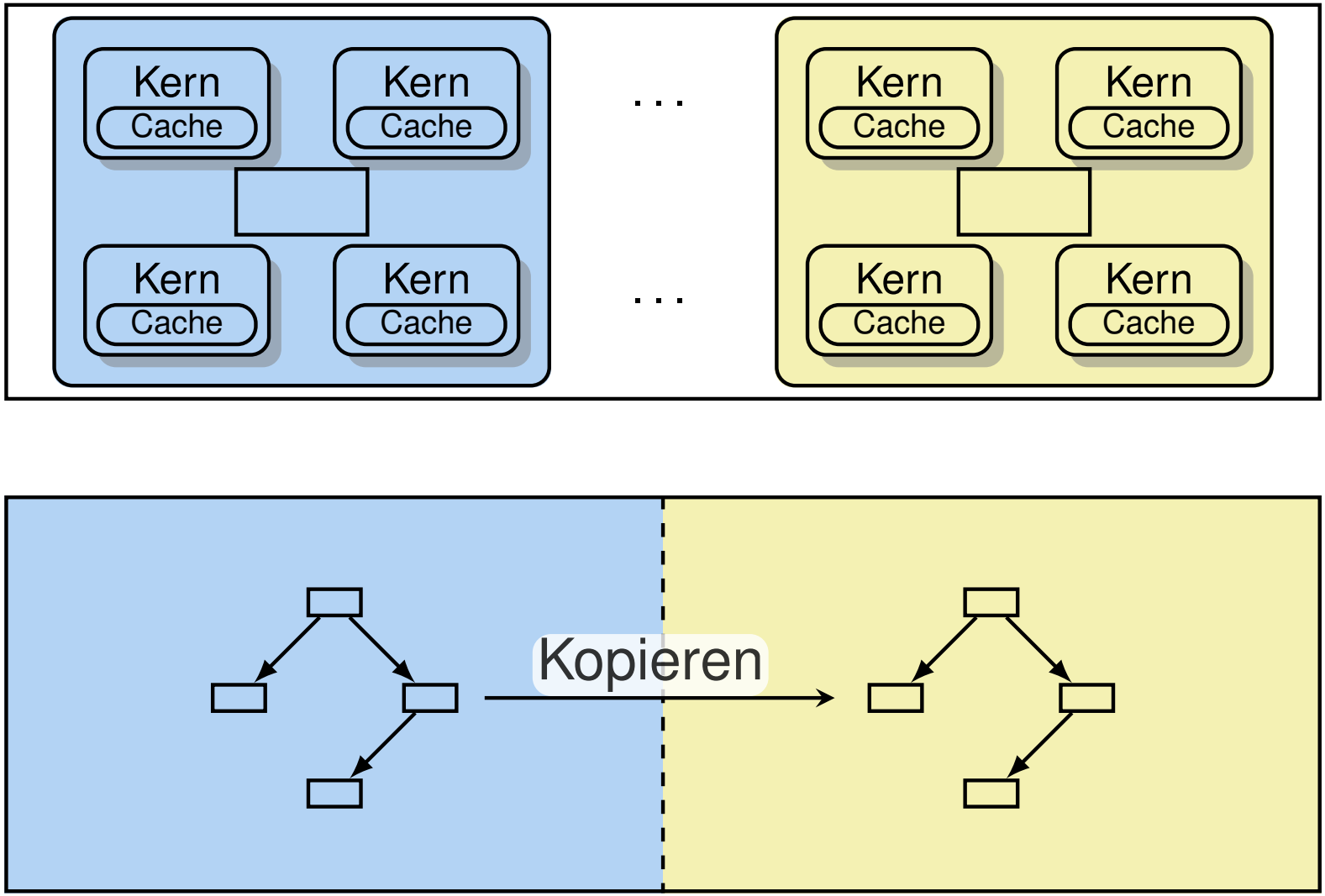
Motivation


Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kopieren

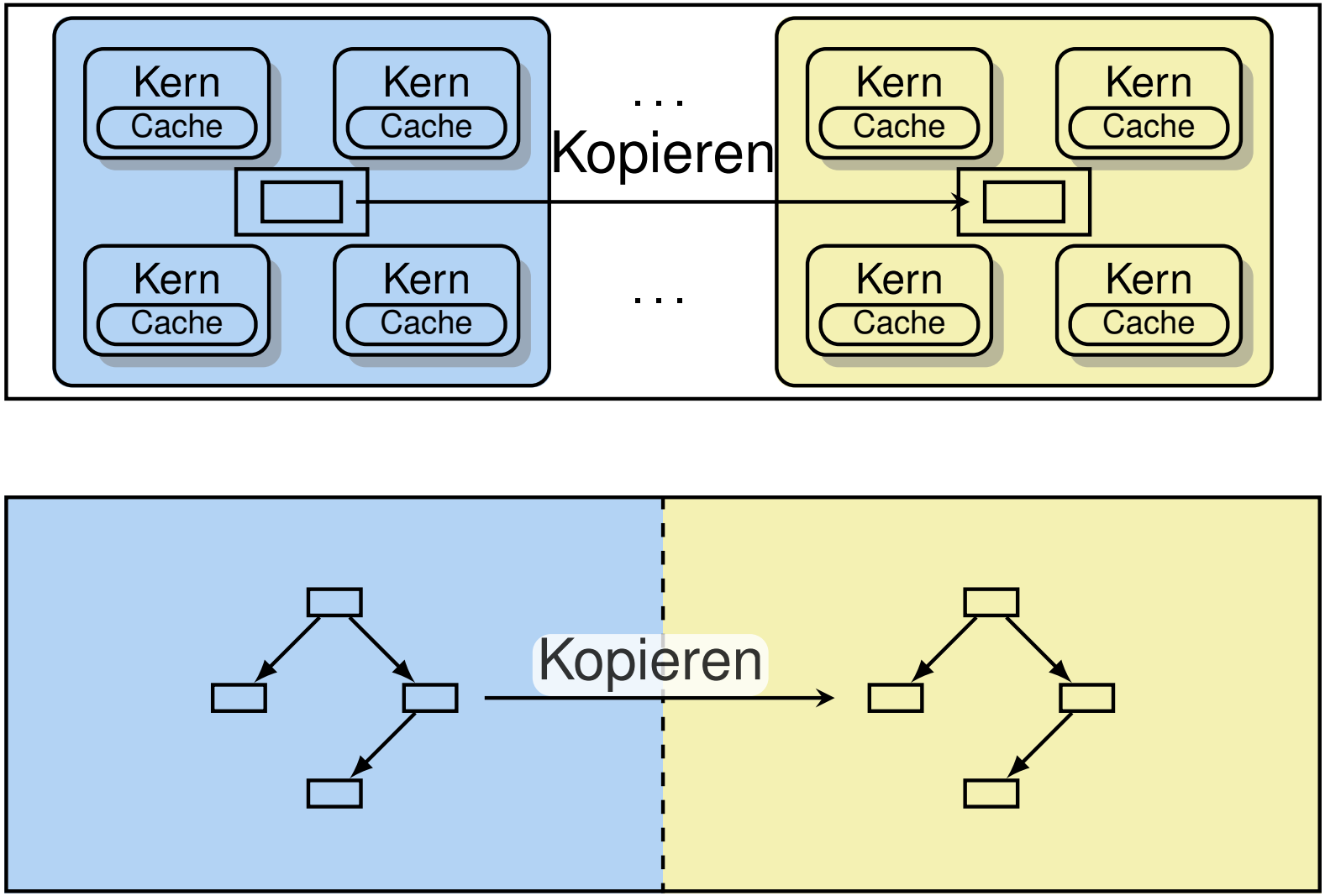
Motivation


Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kopieren

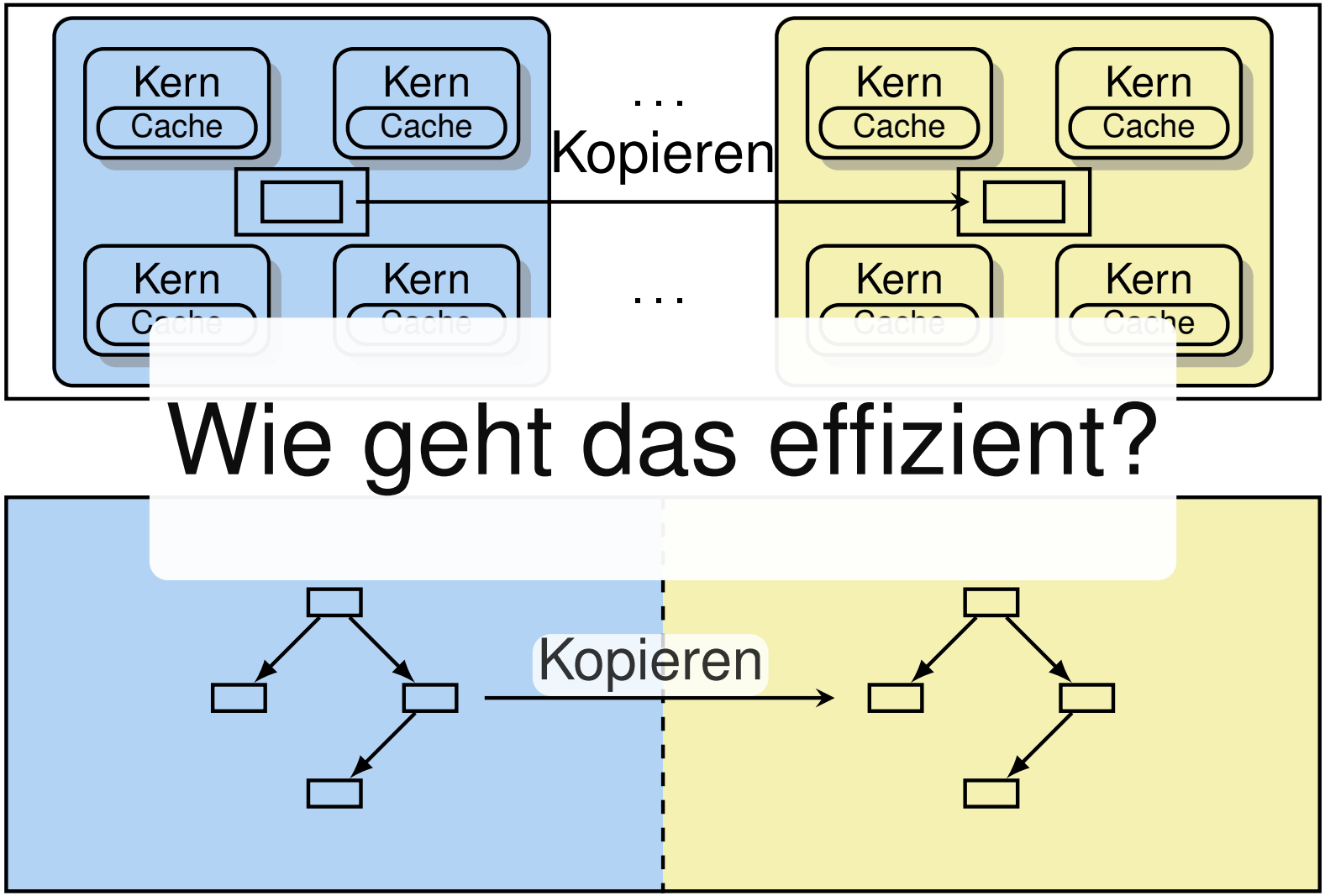
Motivation


Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kopieren

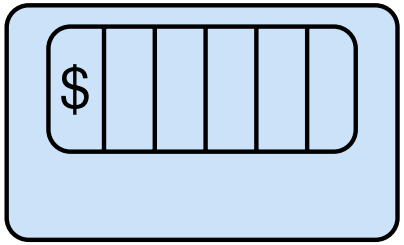
Motivation

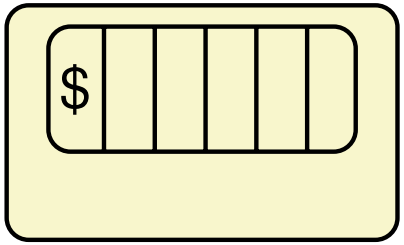

Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kopieren
Kern
Cache
Kern
Cache
Kern
Cache
Kern
Cache
Kopieren

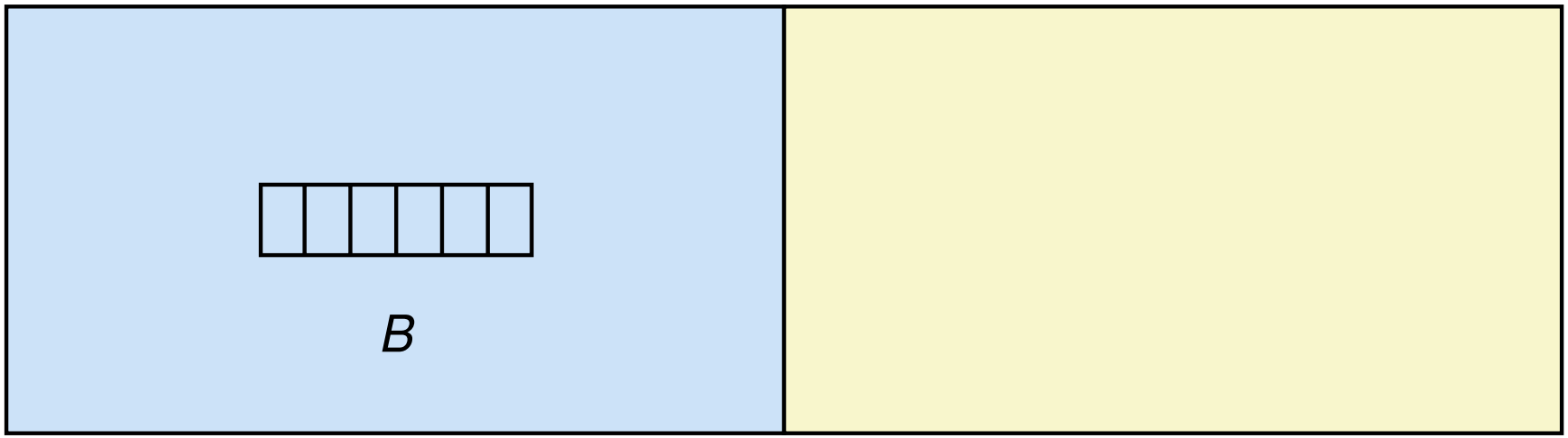
Motivation


Kern
Cache
Kern
Cache
...
Kopieren
Kern
Cache
Kern
Cache
Kern
Kern
...
Kern
Kern
Wie geht das effizient?
Kopieren

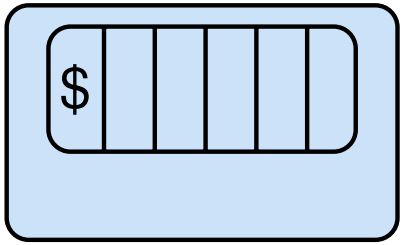
Einfache Datenstrukturen [Moh+15; Moh18]

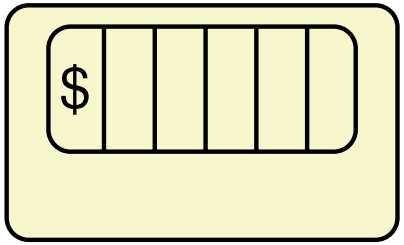


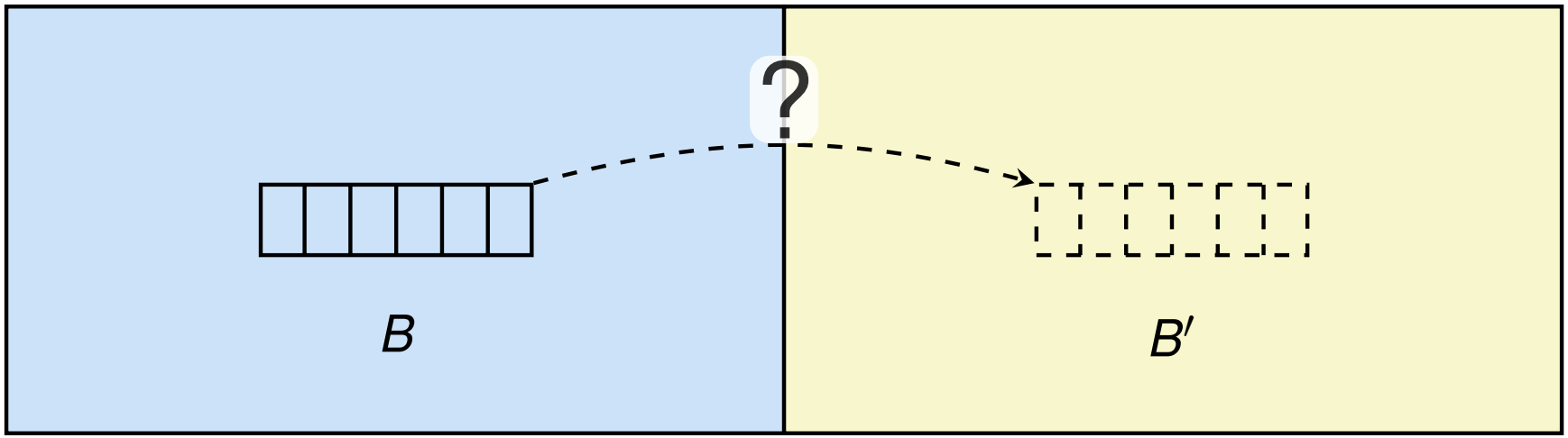




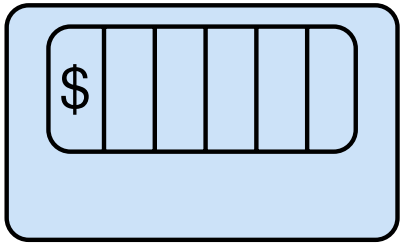
Einfache Datenstrukturen [Moh+15; Moh18]

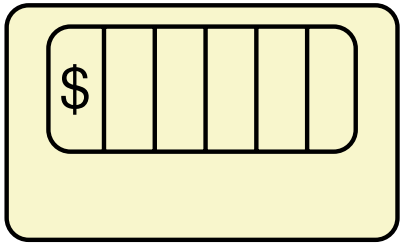


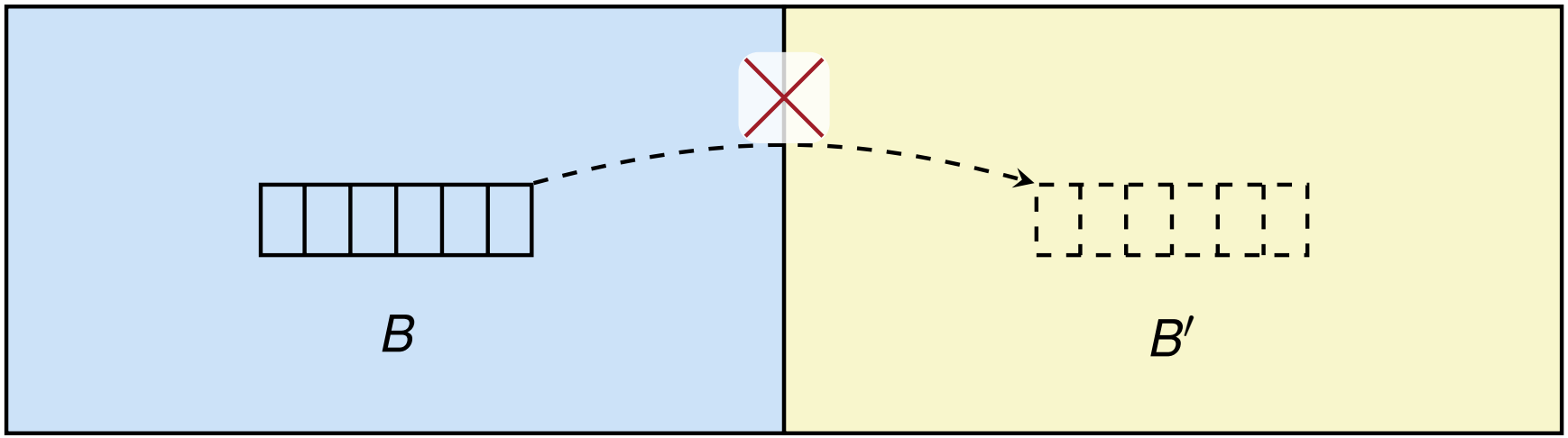




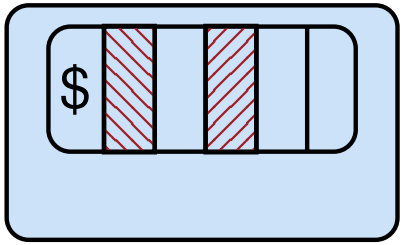
Einfache Datenstrukturen [Moh+15; Moh18]

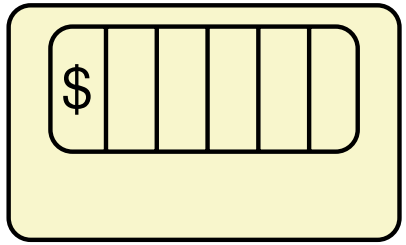


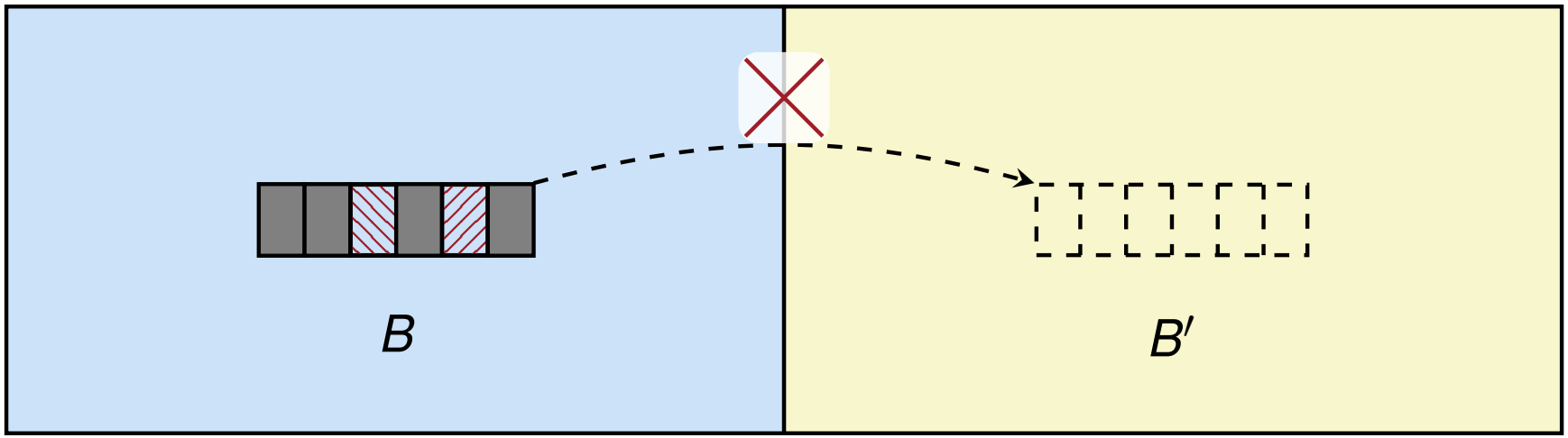




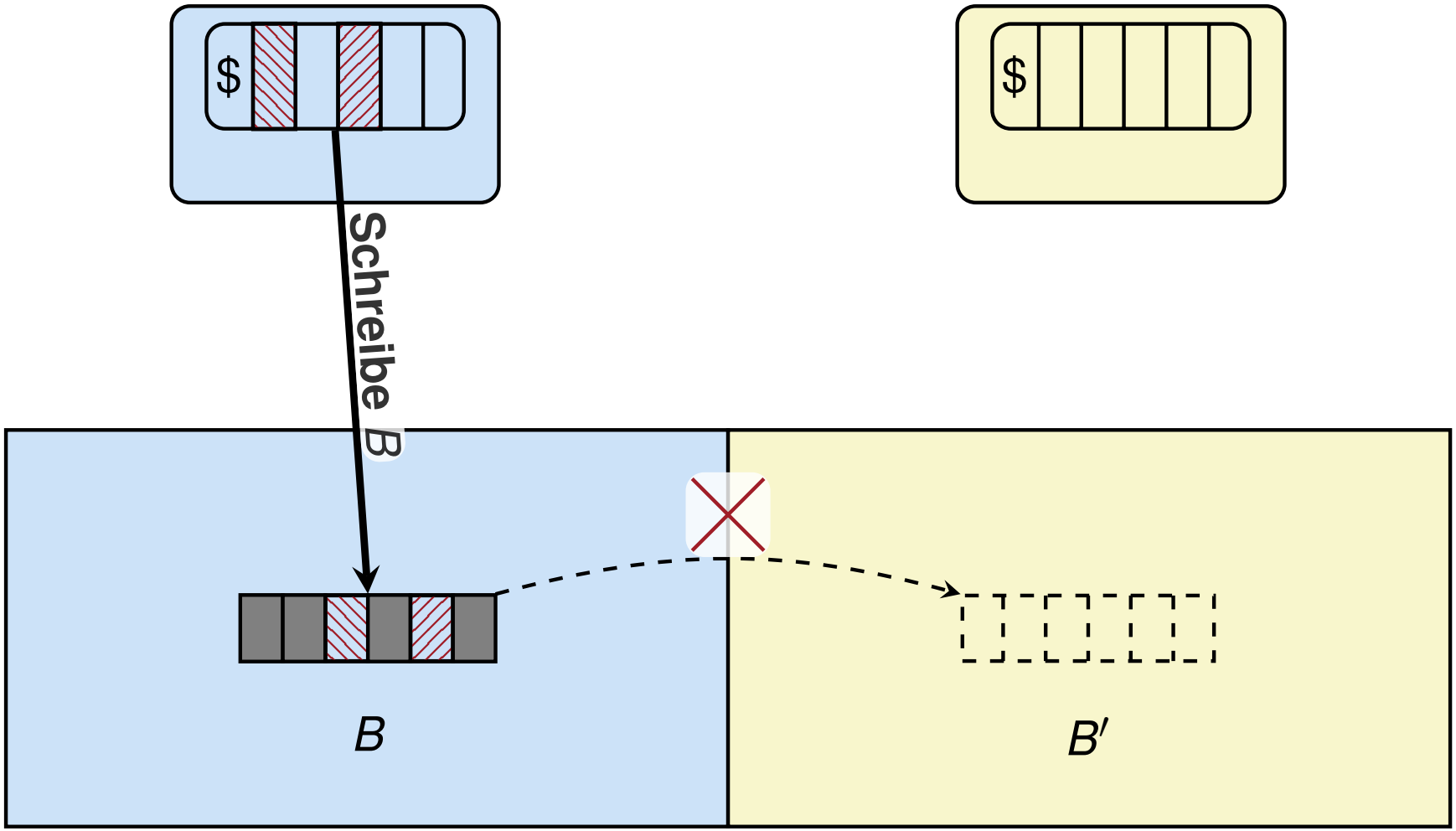
Einfache Datenstrukturen [Moh+15; Moh18]



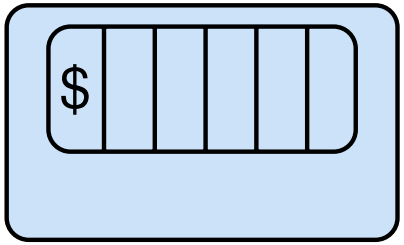


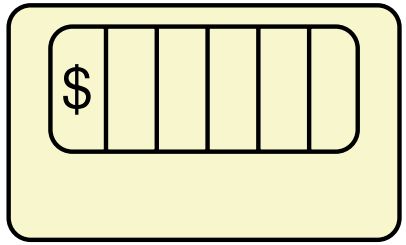


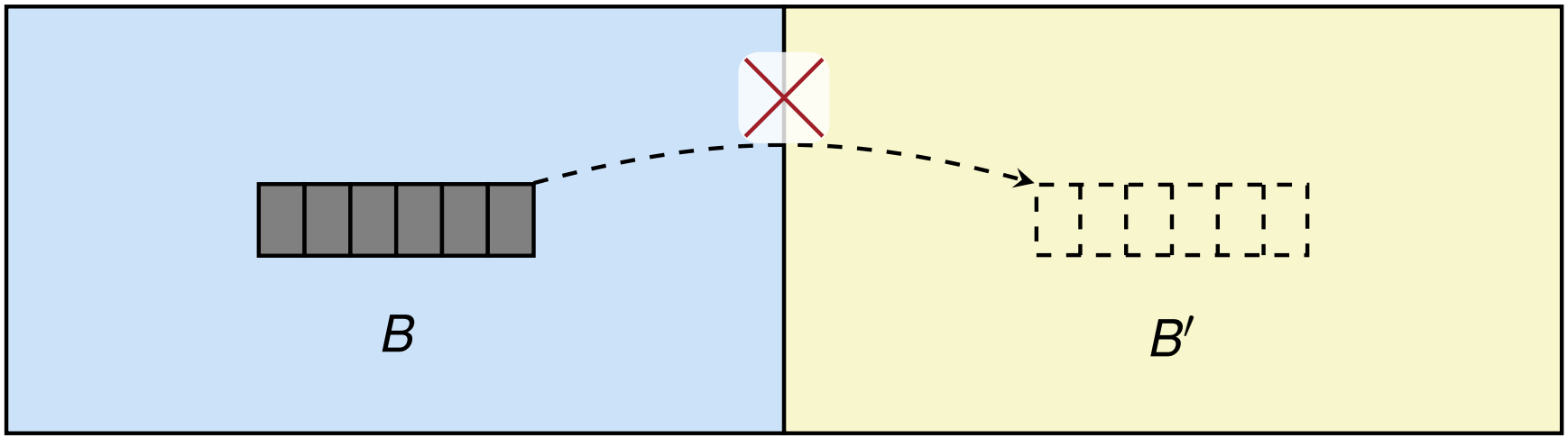
Einfache Datenstrukturen [Moh+15; Moh18]



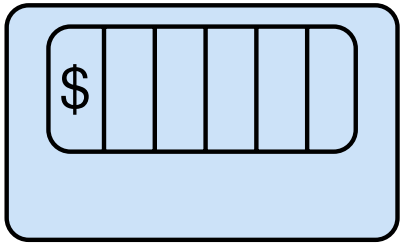
Einfache Datenstrukturen [Moh+15; Moh18]

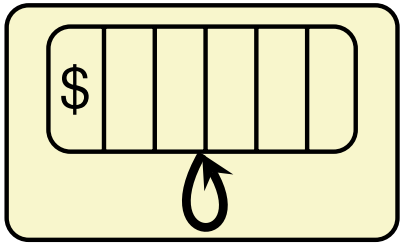




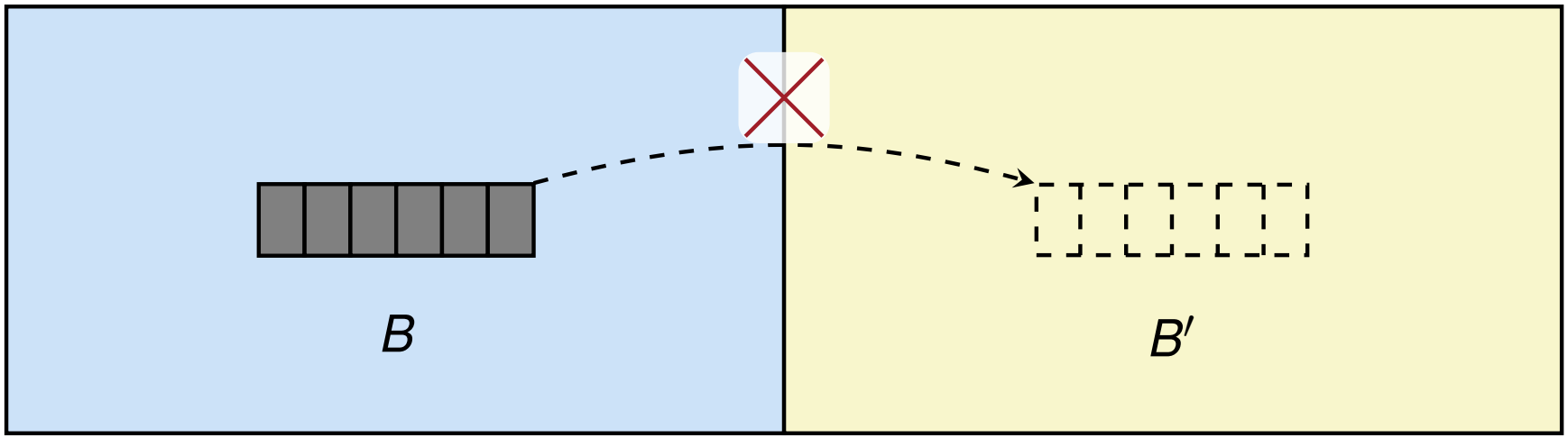


Einfache Datenstrukturen [Moh+15; Moh18]

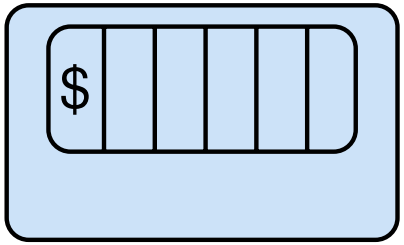


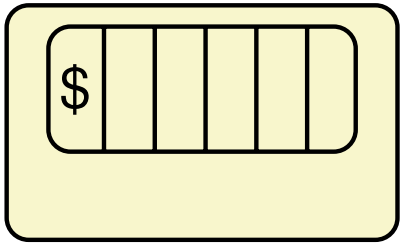


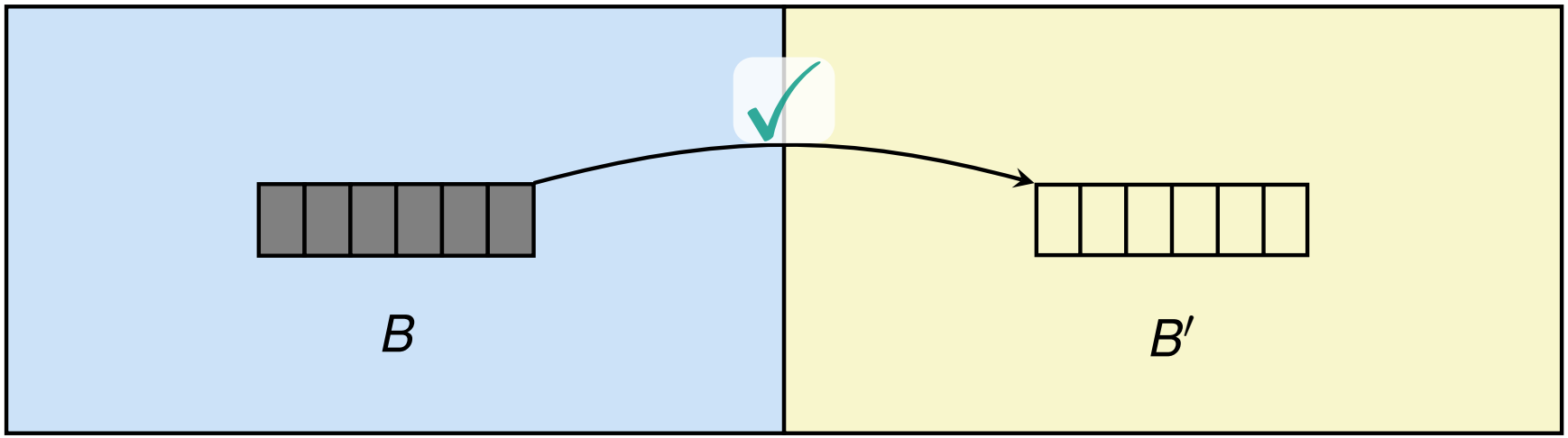
Invalidiere B



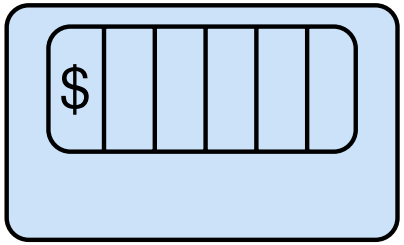
Einfache Datenstrukturen [Moh+15; Moh18]

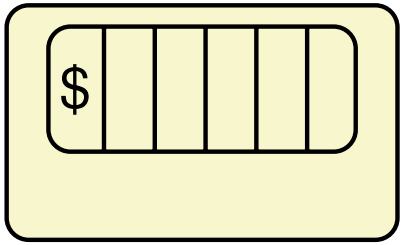


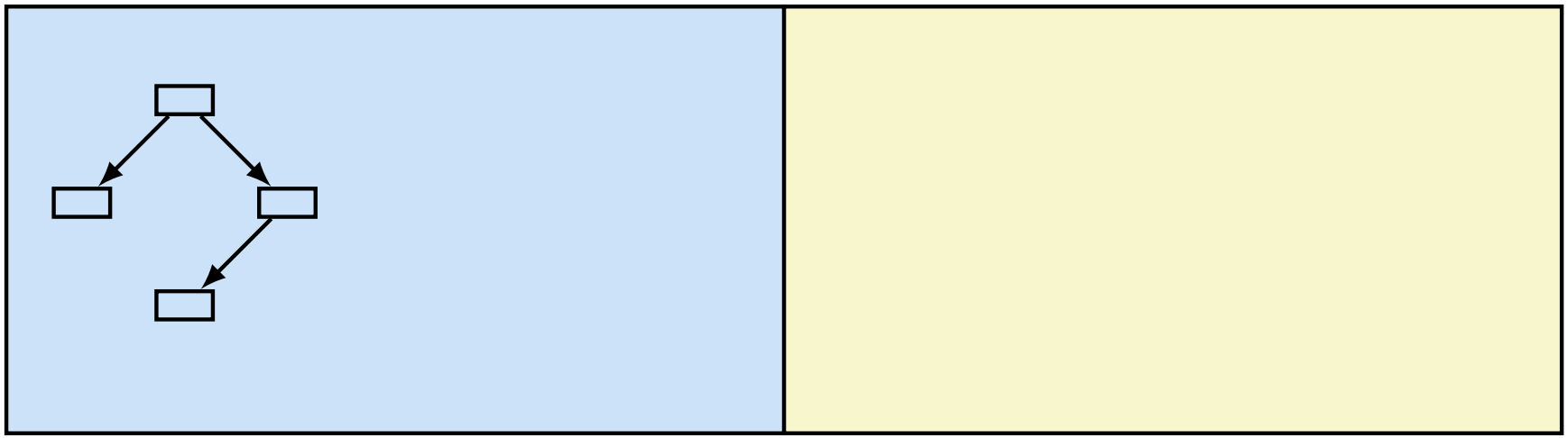




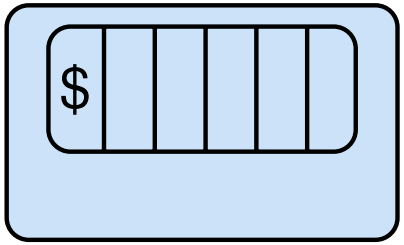
Komplexe Datenstrukturen

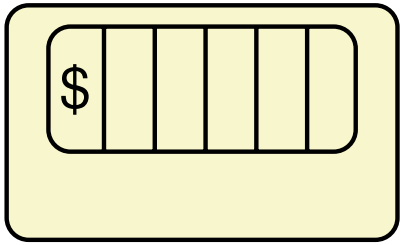


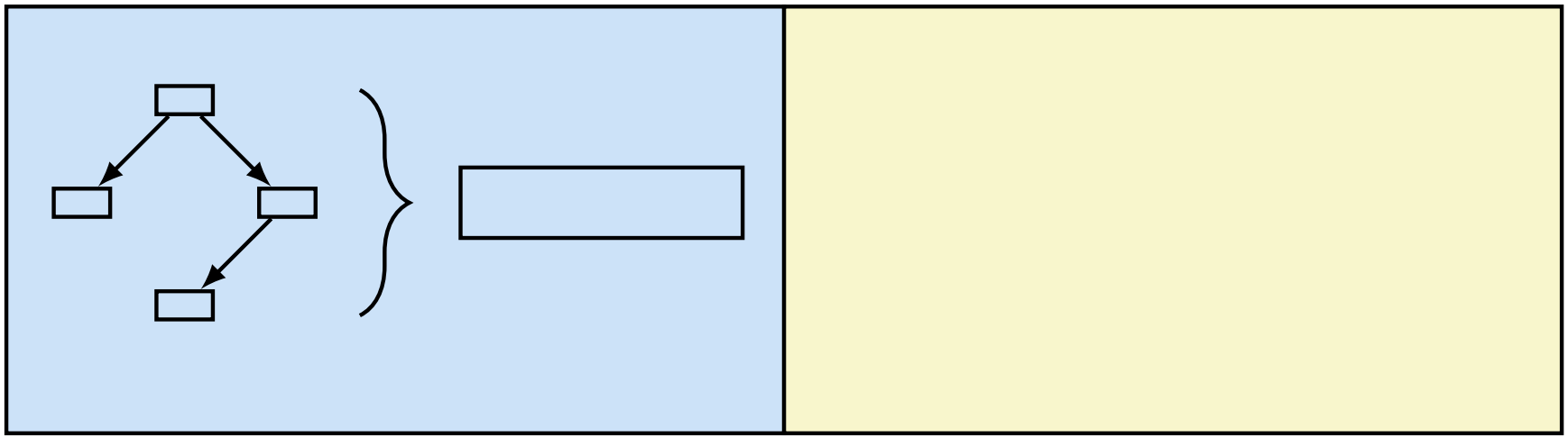




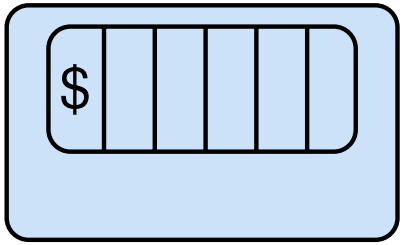
Komplexe Datenstrukturen

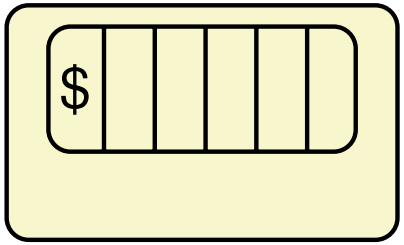


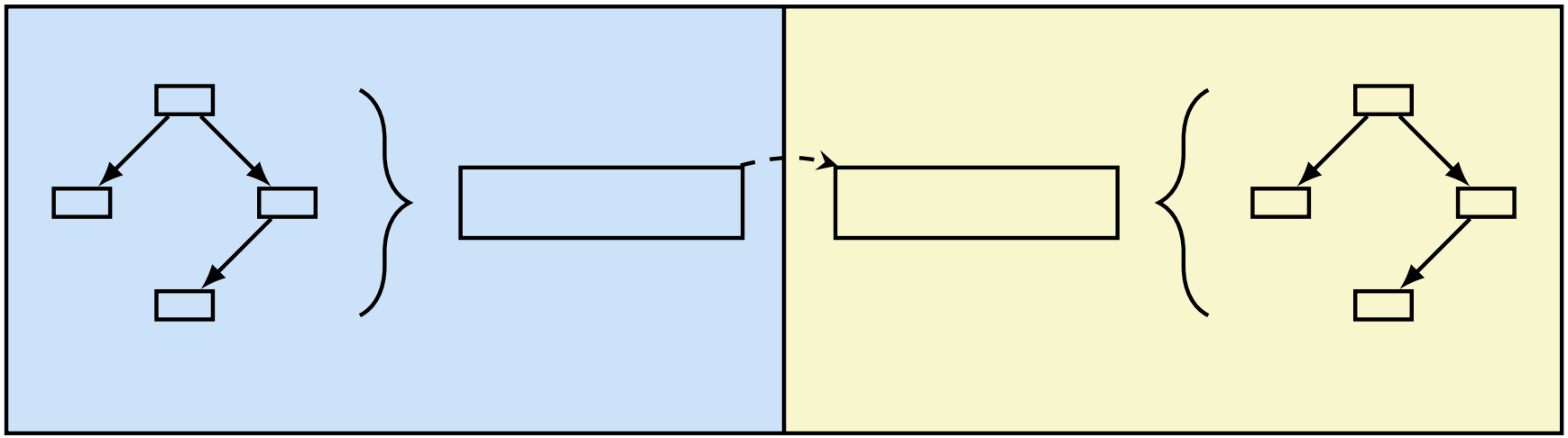




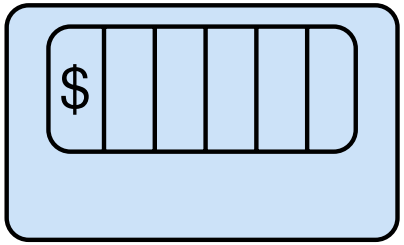
Komplexe Datenstrukturen

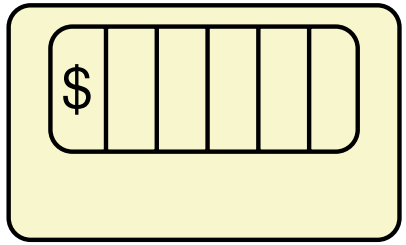


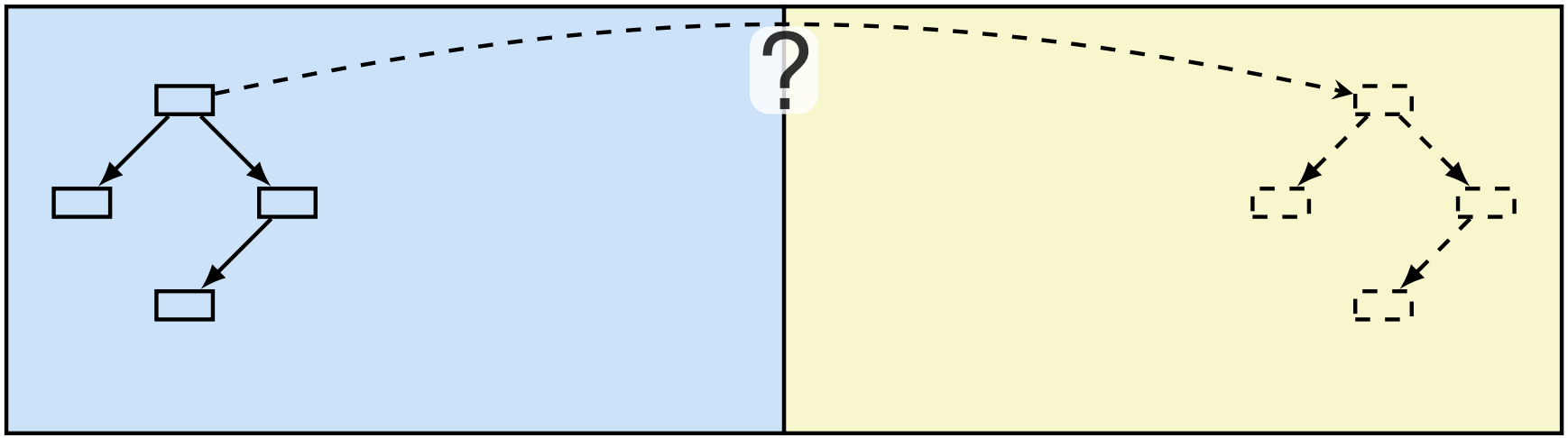




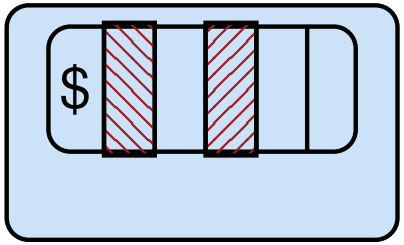
Klonen [MT17]

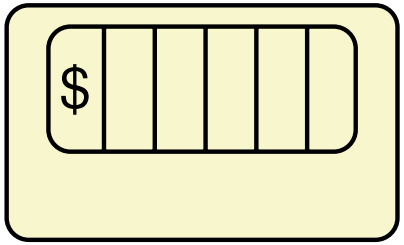


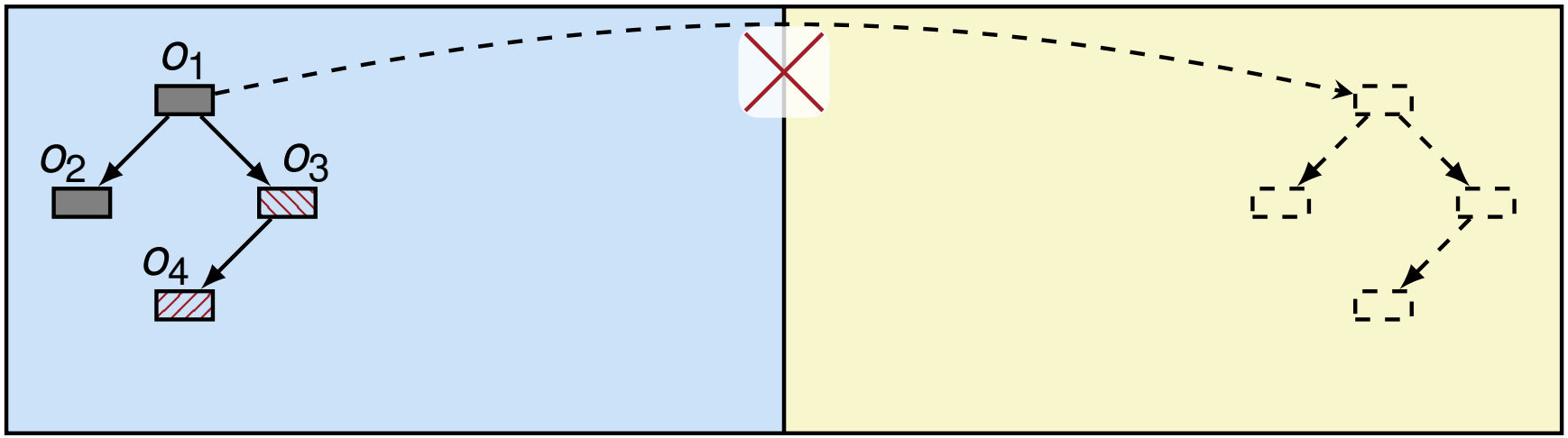




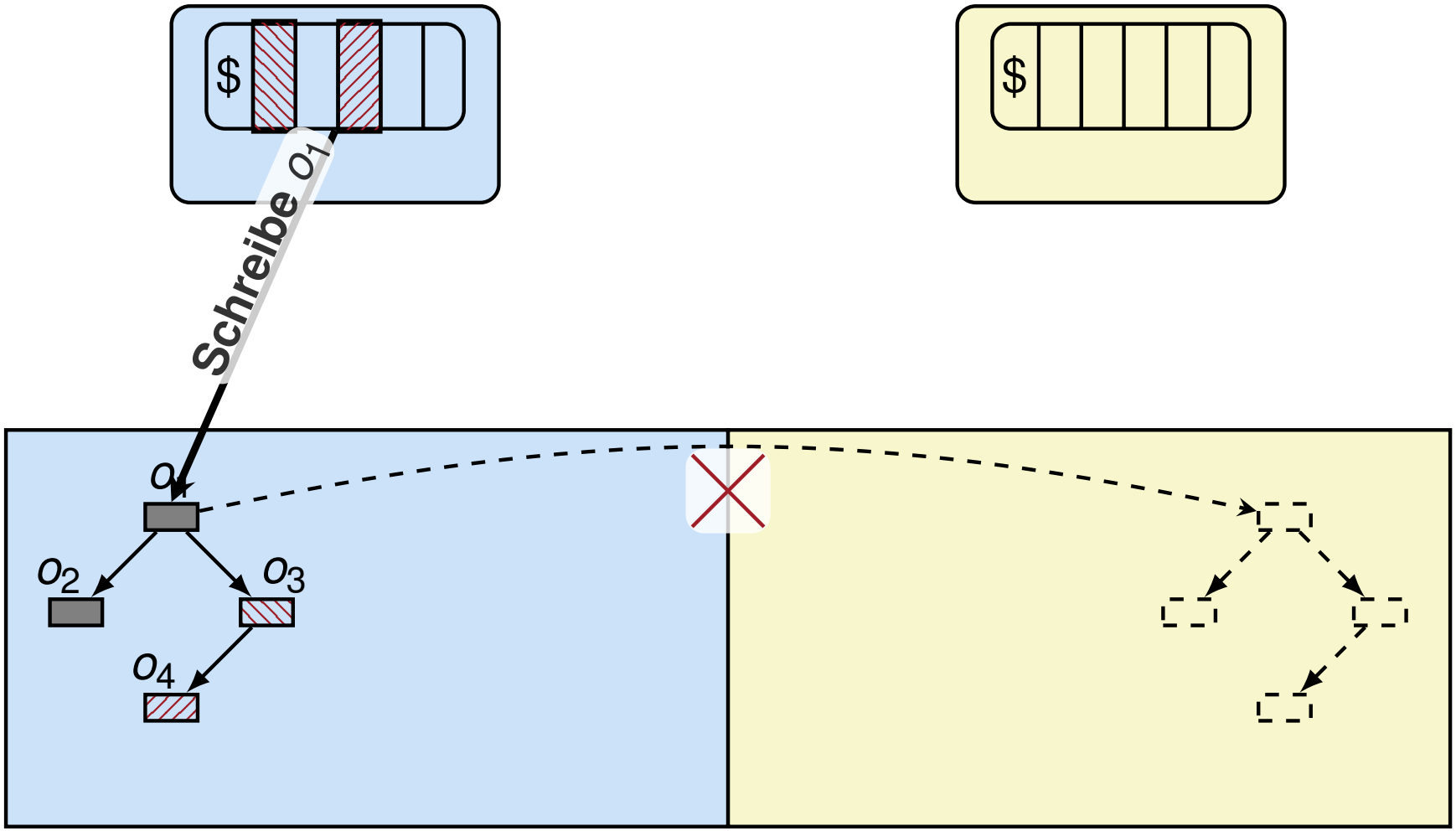
Klonen [MT17]



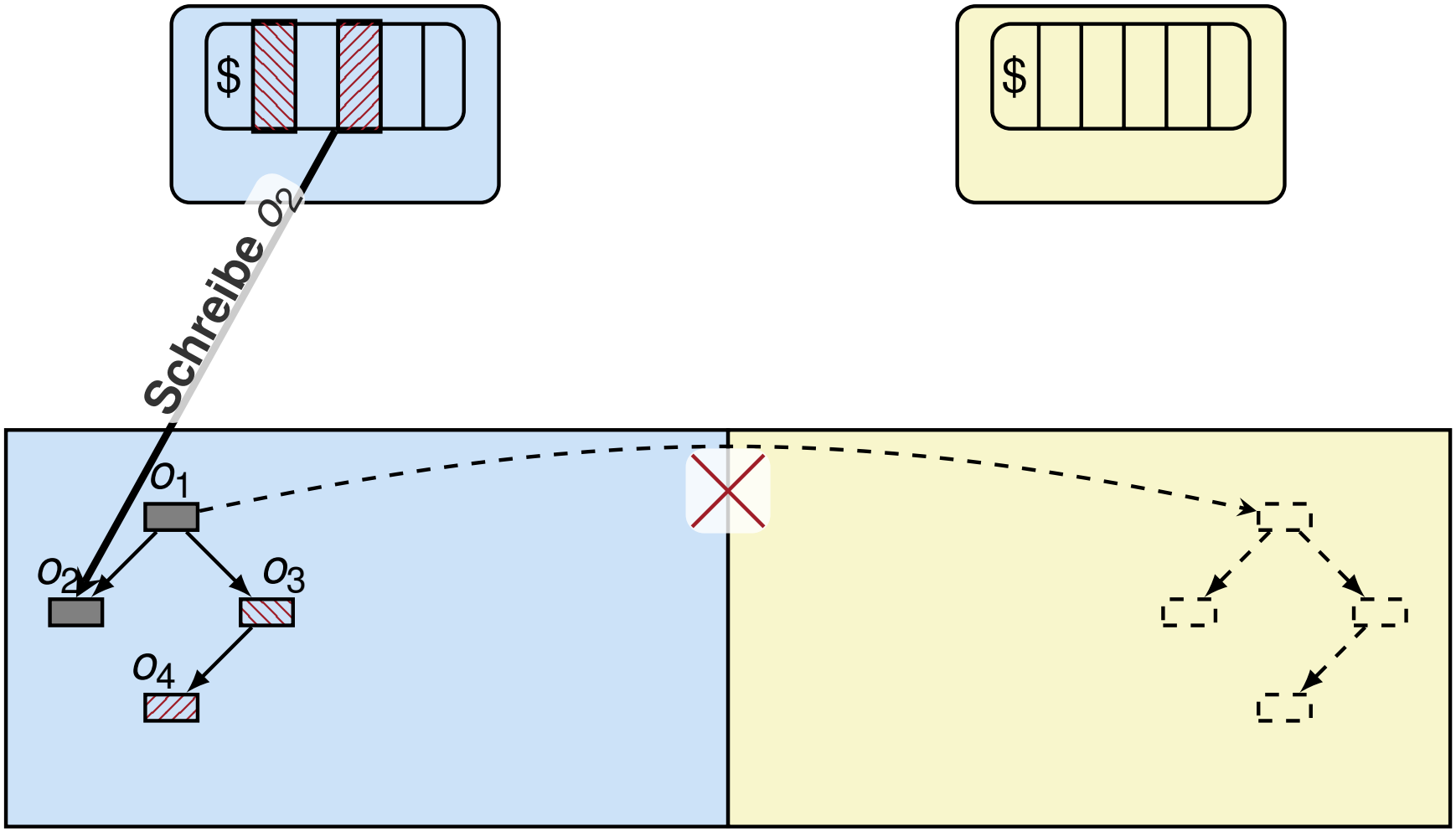




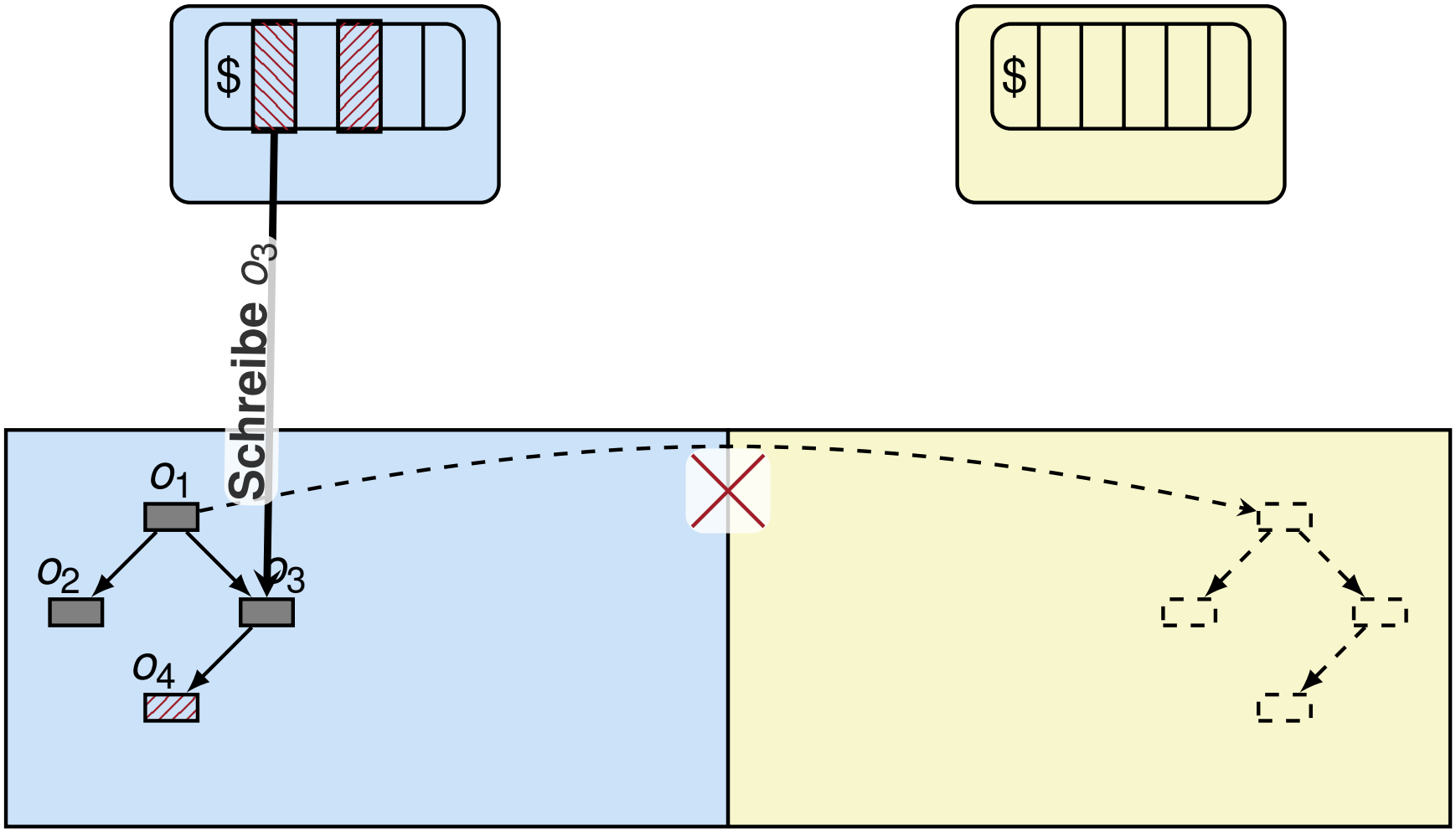
Klonen [MT17]



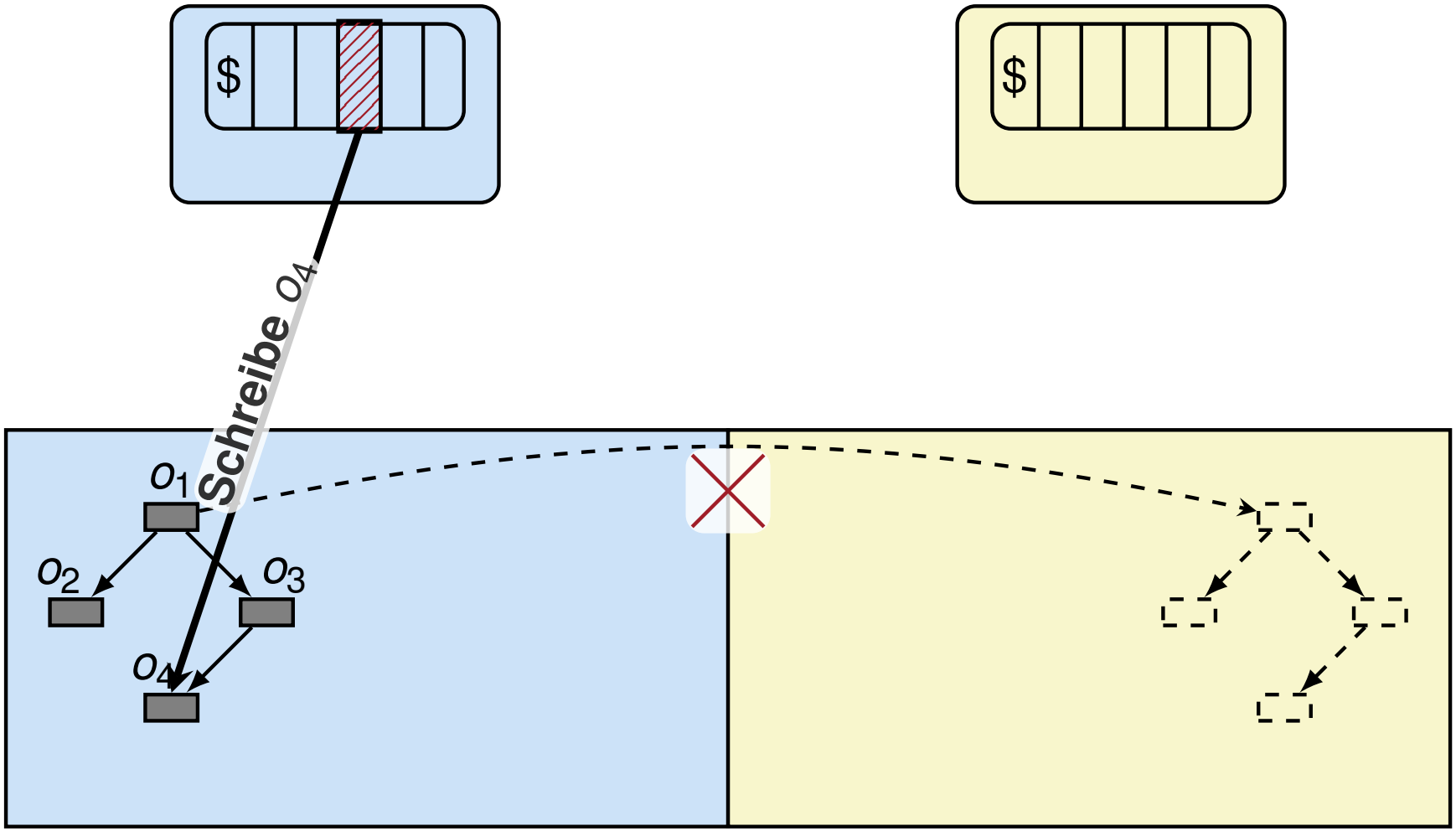
Klonen [MT17]



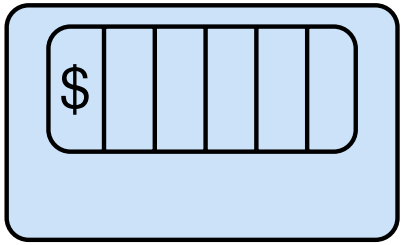
Klonen [MT17]

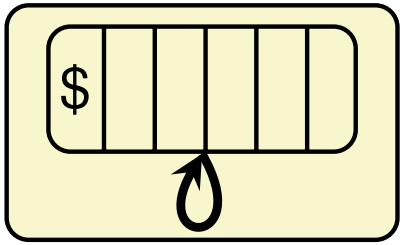


Klonen [MT17]

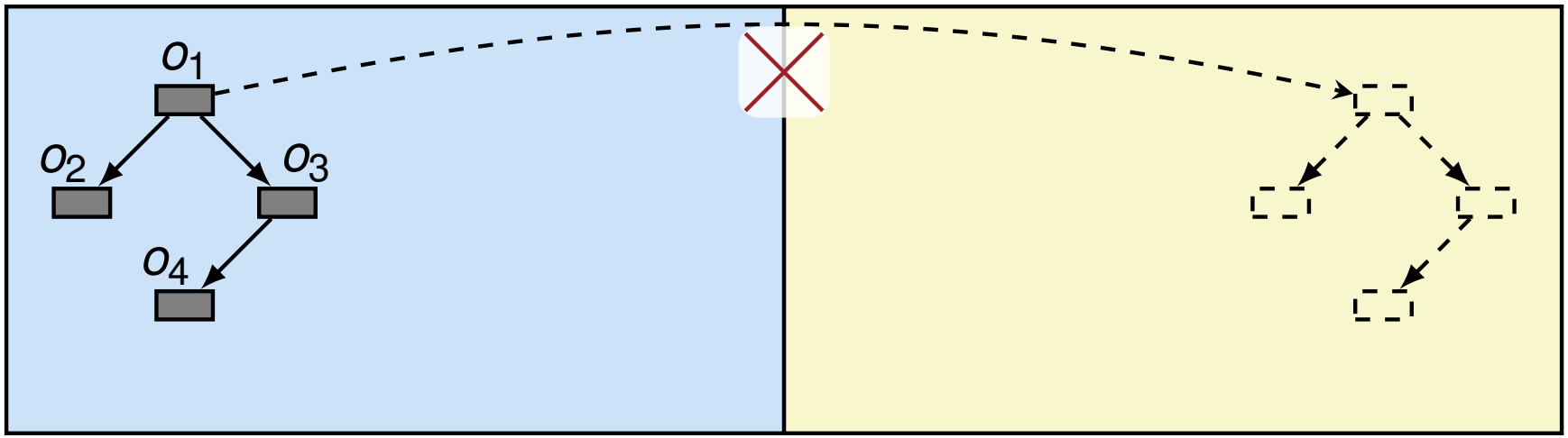


Klonen [MT17]

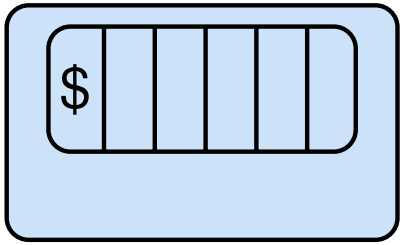


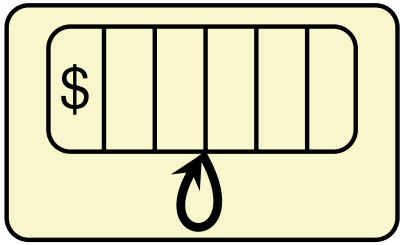


Invalidiere o_1

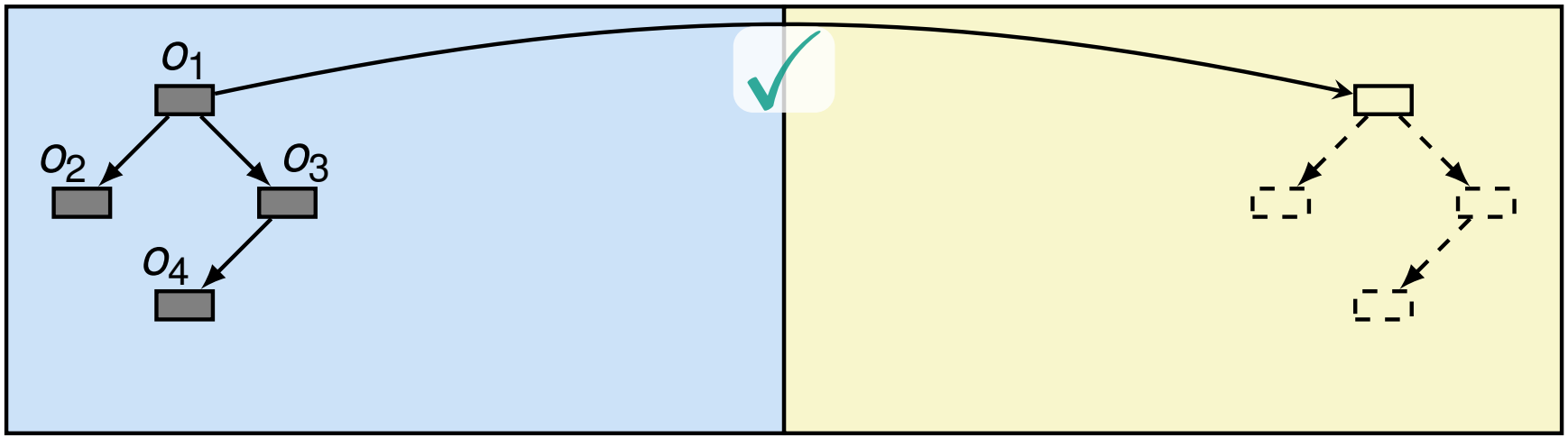


Klonen [MT17]

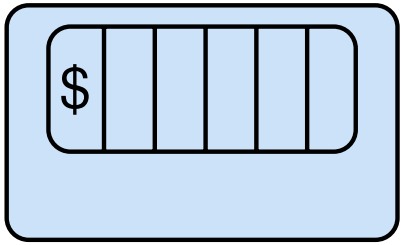


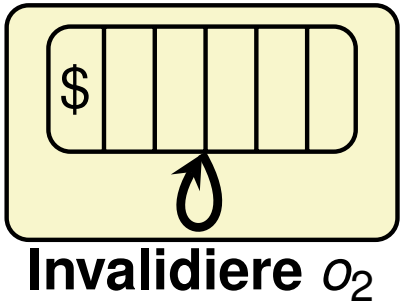


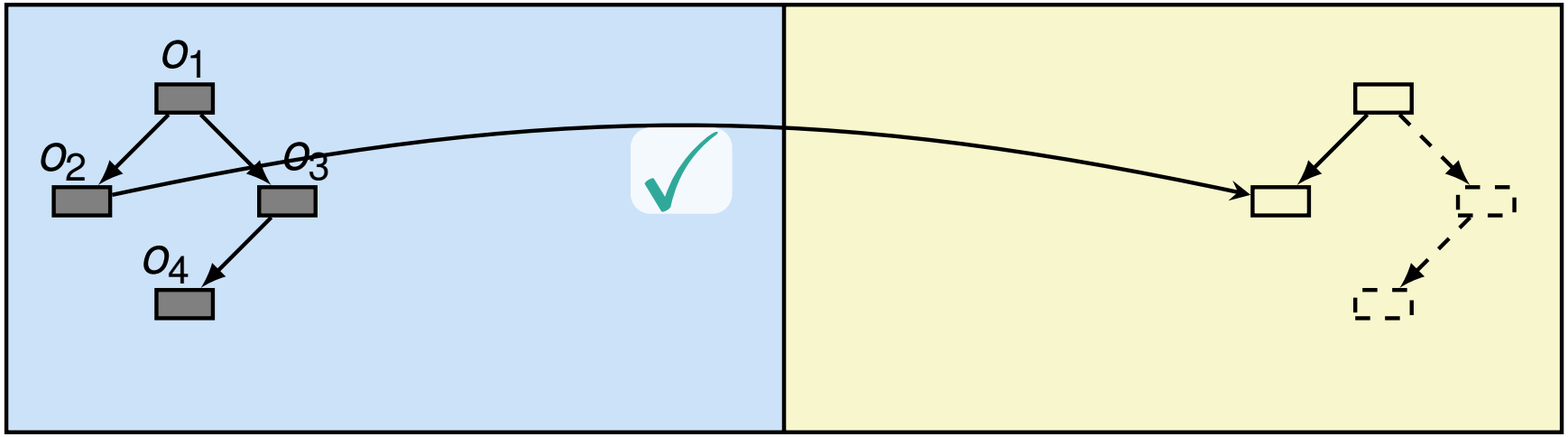
Invalidiere o_1



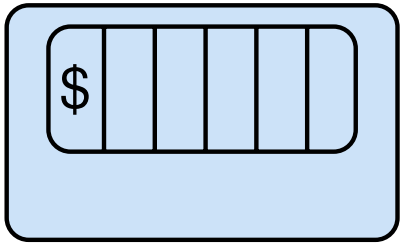
Klonen [MT17]

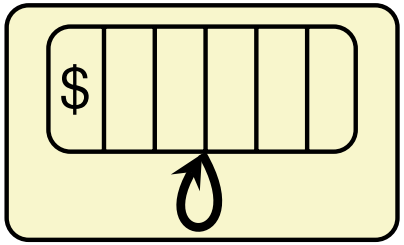




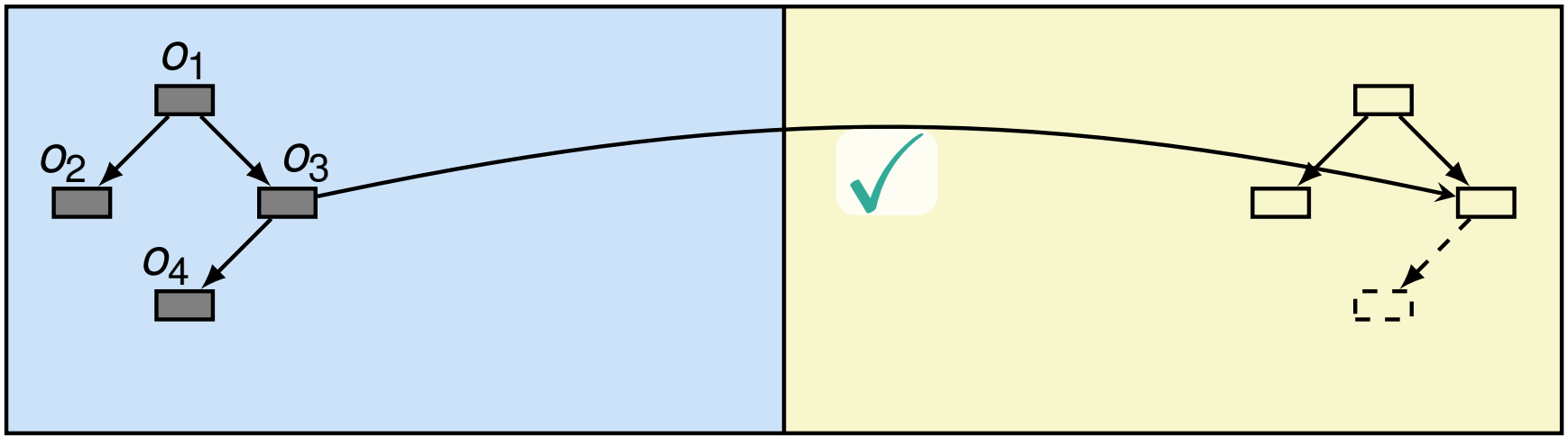


Klonen [MT17]

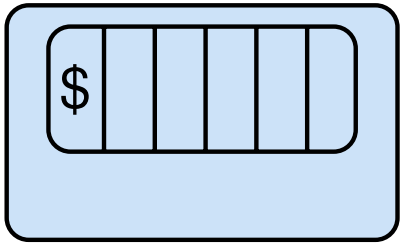


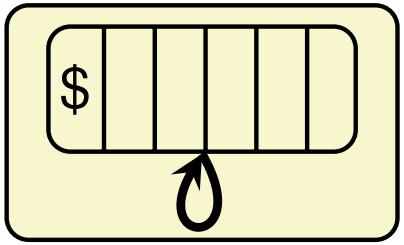


Invalidiere o_3

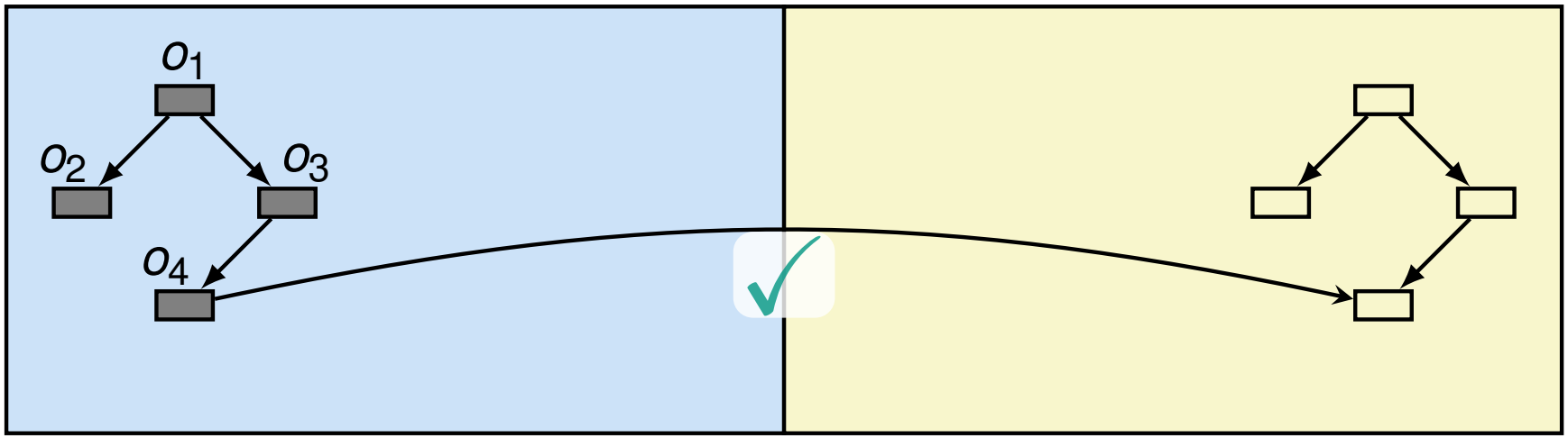


Klonen [MT17]





Invalidiere o_4



Ergebnisse [Moh+15; Moh18]

Einfache Datenstrukturen:

- Allgemein: effizienter direkt über Off-Chip-Speicher (Speedup $\approx 2\times$)
- Abhängig von Speicherparametern: On-Chip-Speicher für kleine Datenmengen vorteilhaft

Ergebnisse [Moh+15; Moh18]

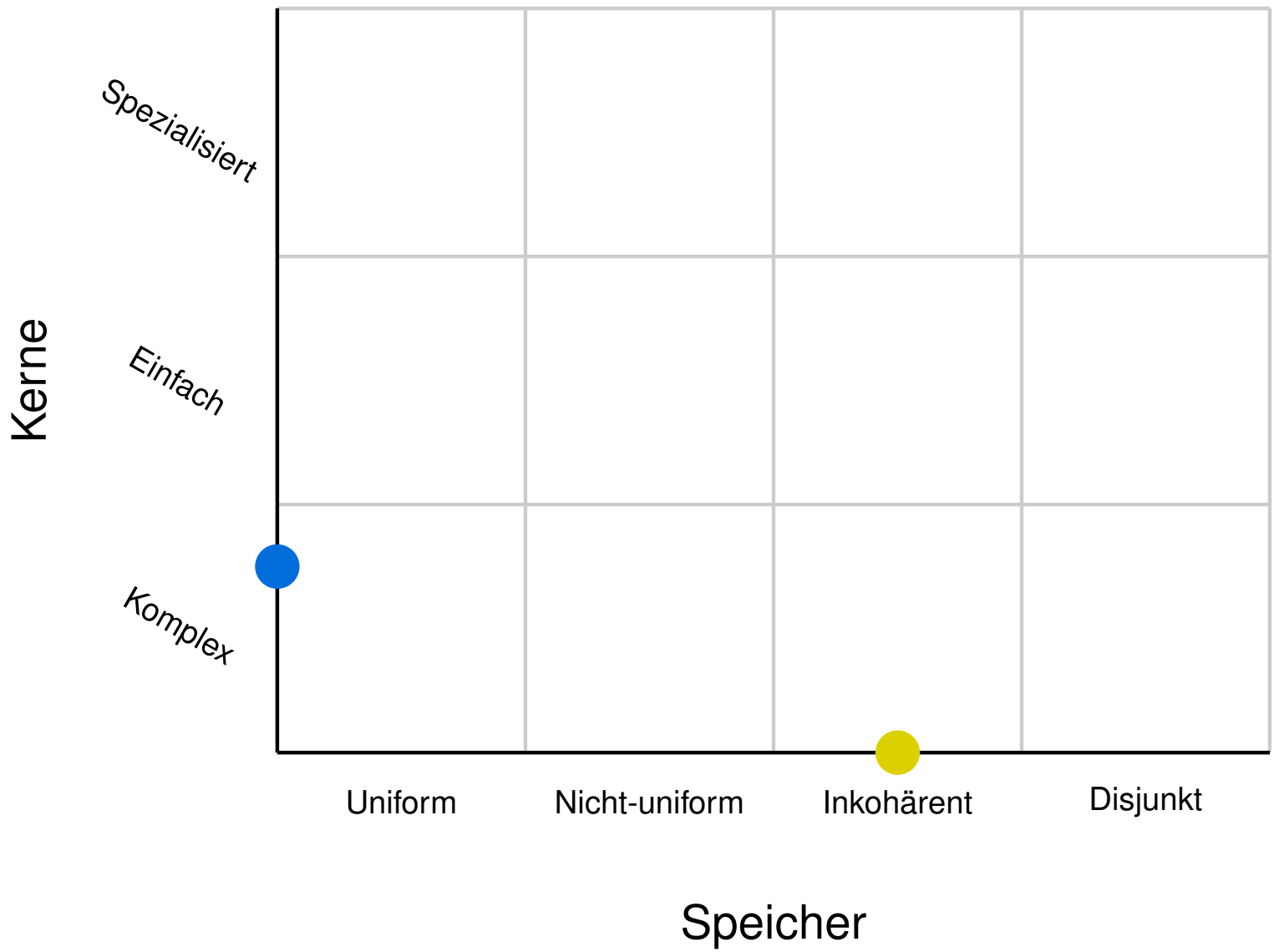
Einfache Datenstrukturen:

- Allgemein: effizienter direkt über Off-Chip-Speicher (Speedup $\approx 2\times$)
- Abhängig von Speicherparametern: On-Chip-Speicher für kleine Datenmengen vorteilhaft

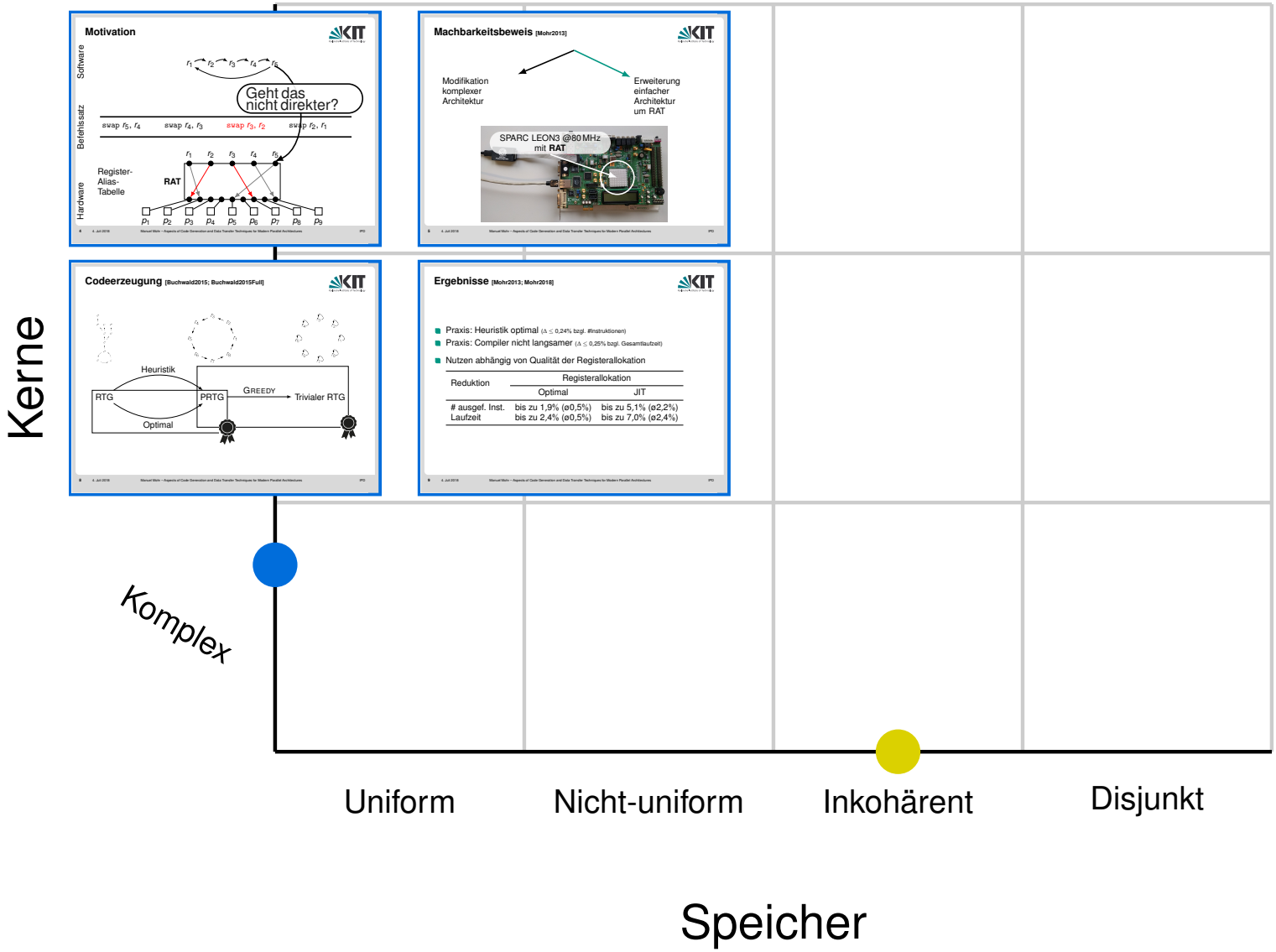
Komplexe Datenstrukturen:

- On-Chip-Speicher gut für kleine Datenmengen, Off-Chip für große
- Serialisierung für kleine Datenmengen, Klonen für große
  - Klonen für große Datenstrukturen bis Speedup $10\times$ ggü. Ser.
  - Benchmarksuite: Laufzeitreduktion ø5%, bis zu 20%

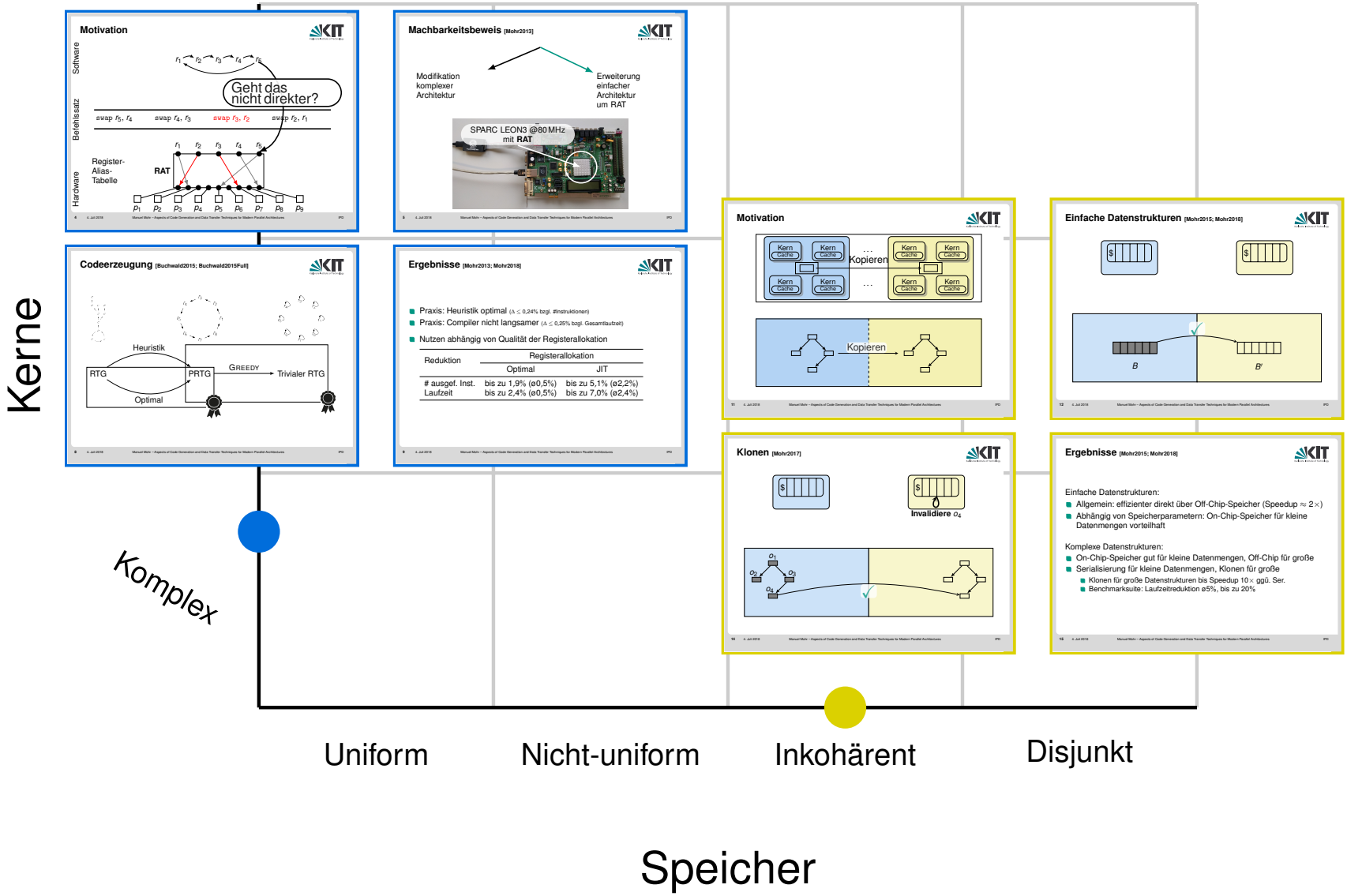
Zusammenfassung



Zusammenfassung



Zusammenfassung



Backup-Folien

Übersicht

- ISA-Erweiterung
- Hardwareerweiterung
- Exception-Behandlung
- Copy-Set: Heuristik
- Copy-Set: Optimal
- GREEDY
- GREEDY: Beweisstrategie
- Evaluierungsstrategie
- HW-Erweiterung: Motivation
- HW-Erweiterung: Details
- Evaluierungsstrategie
- Evaluierungsdetails

Karlsruhe Institute of Technology

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures

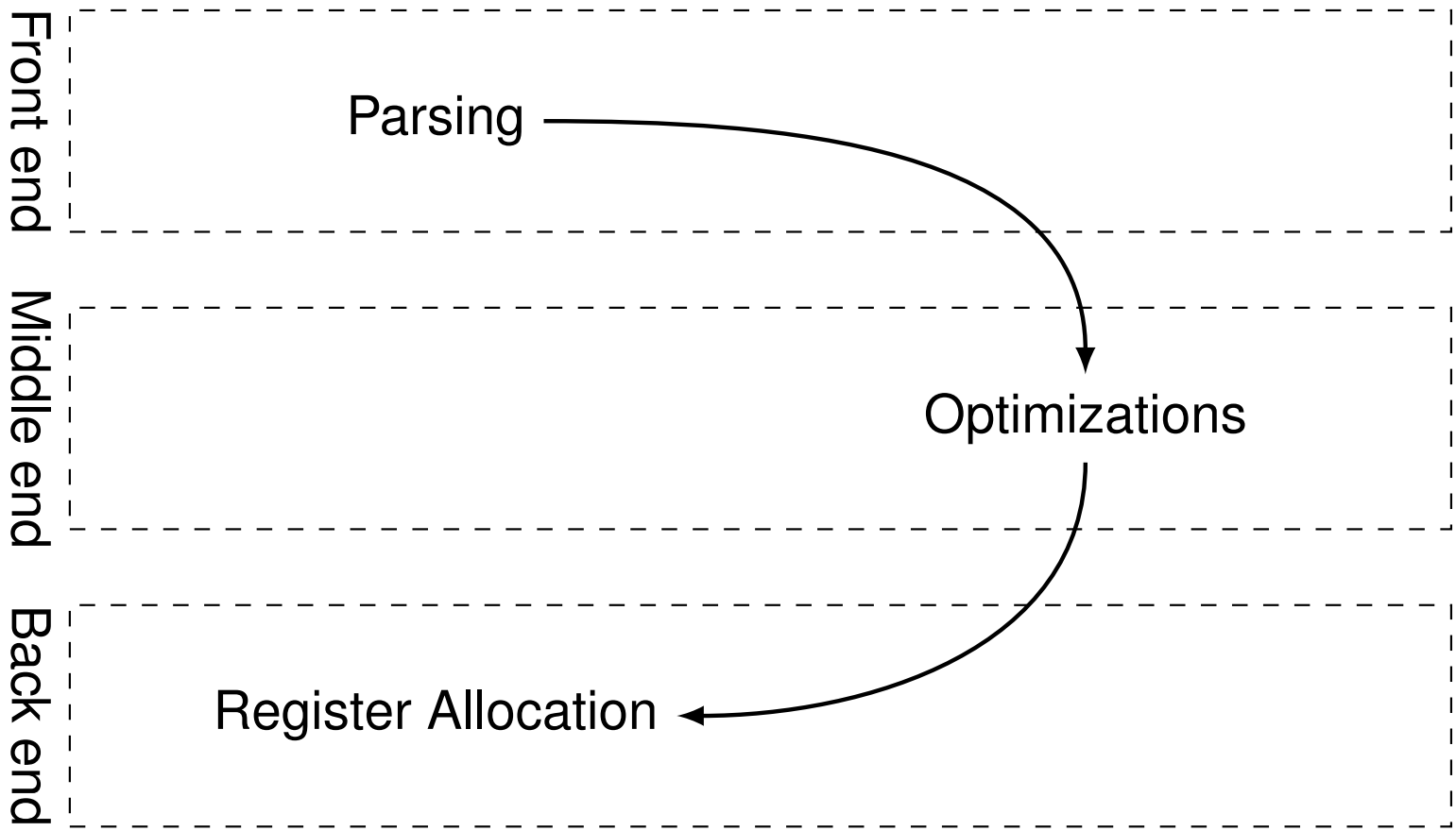
Manuel Mohr

Lehrstuhl für Programmierparadigmen, Karlsruher Institut für Technologie (KIT)

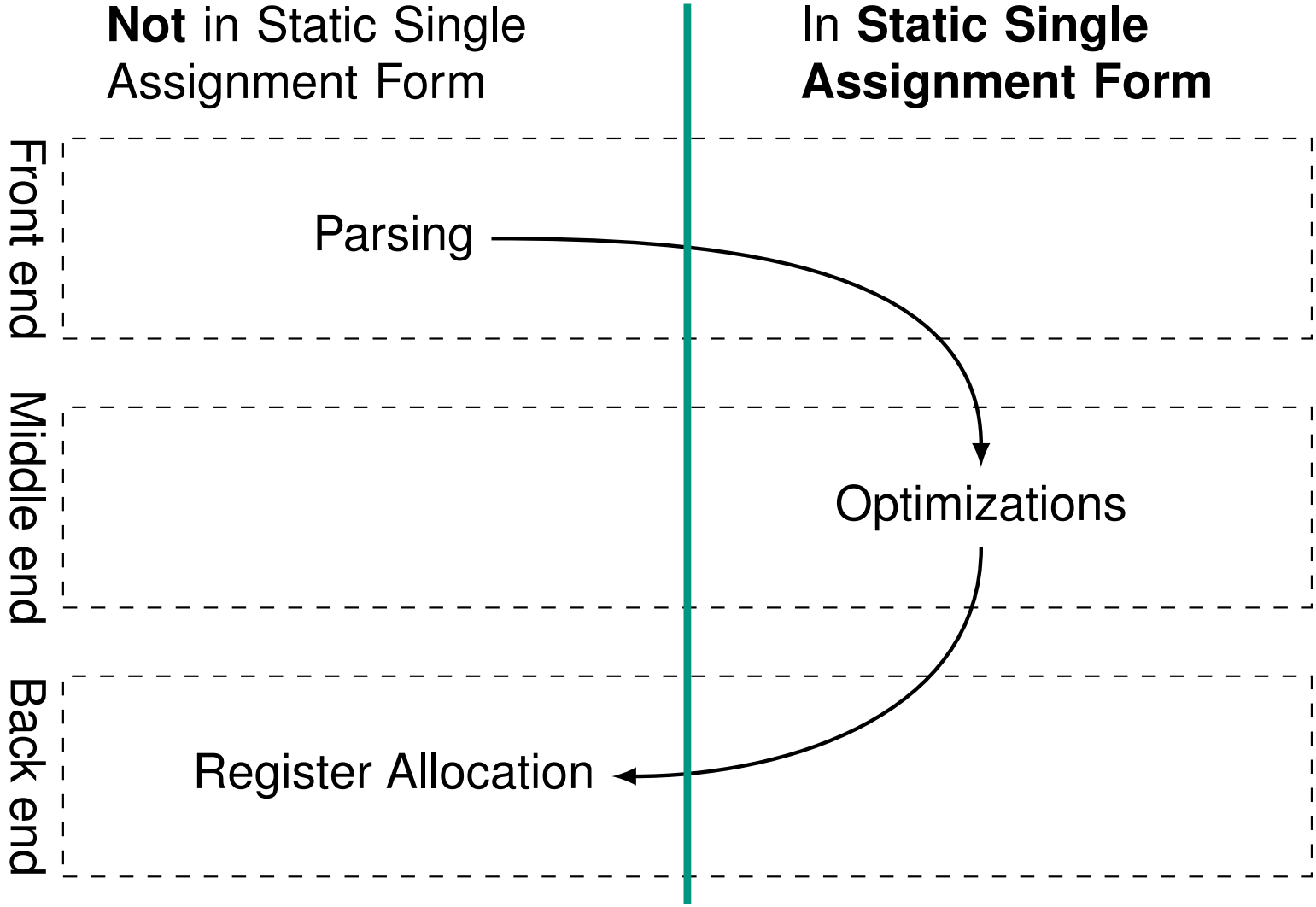
KIT – University of the State of Baden-Wuerttemberg and
National Research Center of the Helmholtz Association

www.kit.edu

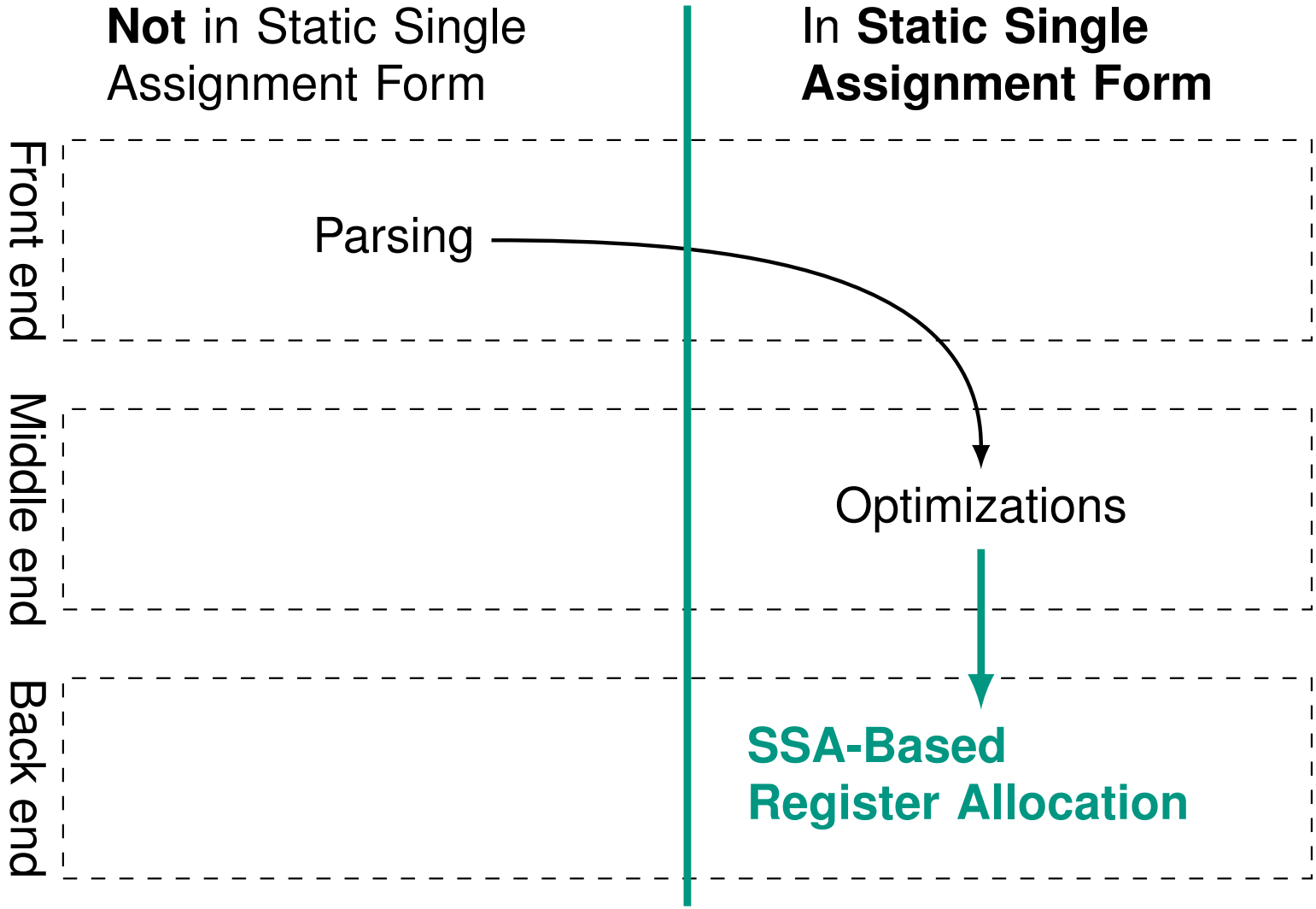
SSA-Based Register Allocation



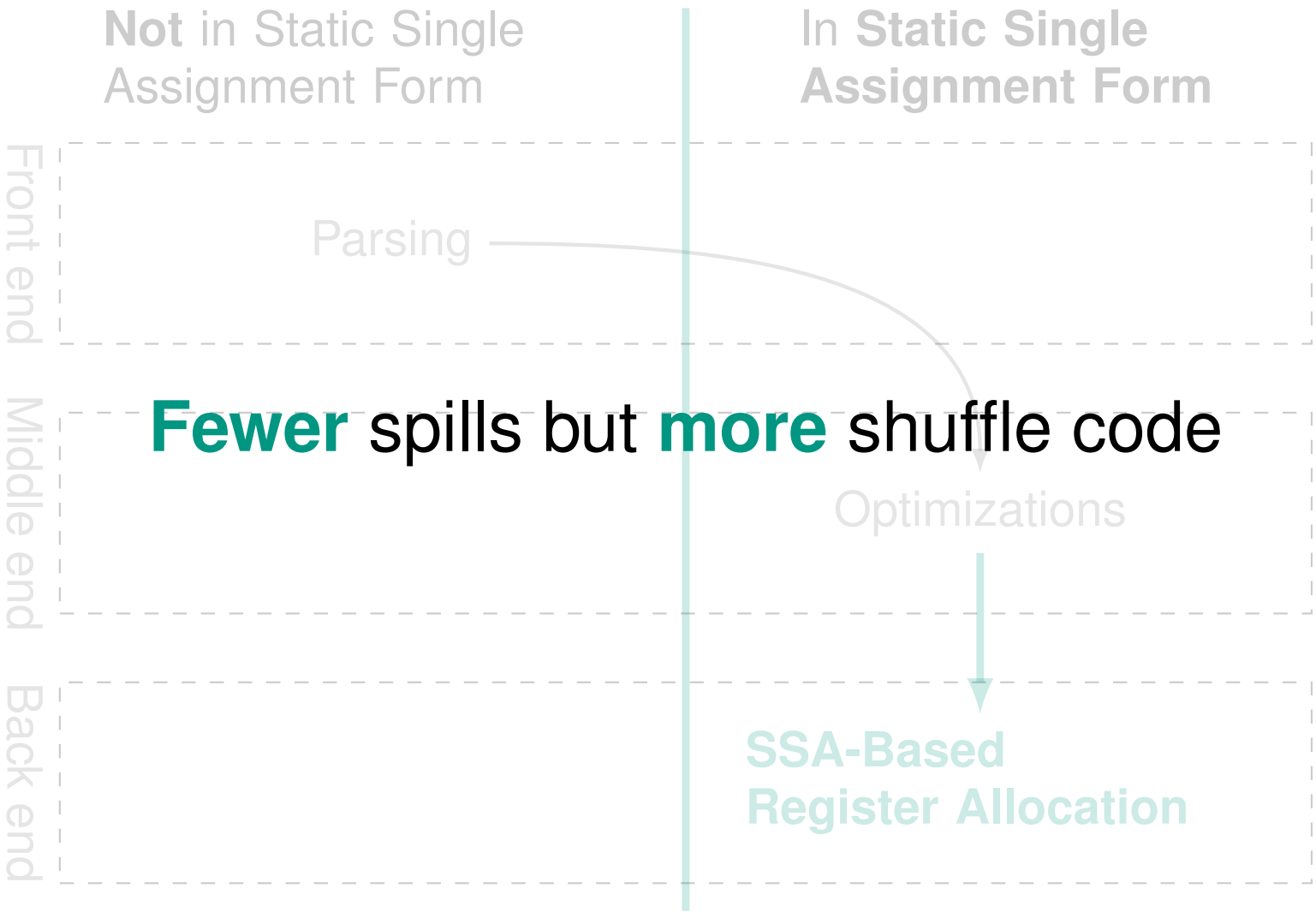
SSA-Based Register Allocation



SSA-Based Register Allocation



SSA-Based Register Allocation

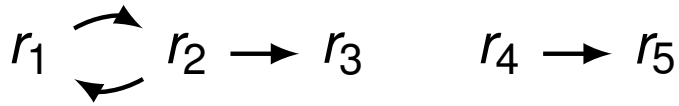


Register Transfer Graphs

Shuffle code = **parallel** copy operations between registers

Register Transfer Graphs

Shuffle code = **parallel** copy operations between registers

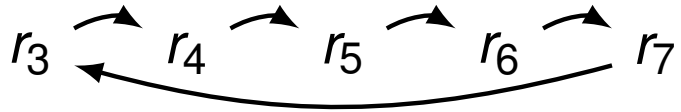


Register Transfer Graph (RTG)

- Nodes: Registers
- Directed edge (r_1, r_2): After copies, value of r_1 must be in r_2
- At most one incoming edge per node
- No incoming edge: Register value is irrelevant after copies

Motivation

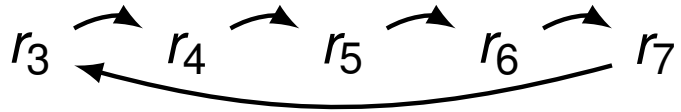
- Number and size of RTGs depend on quality of allocation
- Reduction is an NP-complete problem



⇒ On standard hardware, implementation may be expensive:
5% to 20% of all generated instructions (SPEC)

Motivation

- Number and size of RTGs depend on quality of allocation
- Reduction is an NP-complete problem

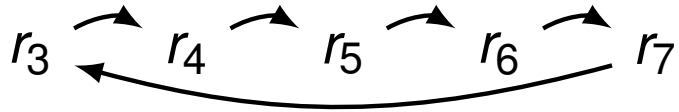


⇒ On standard hardware, implementation may be expensive:
5% to 20% of all generated instructions (SPEC)

```
xor r6, r7      xor r6, r5      xor r4, r5
xor r7, r6      xor r5, r6      xor r3, r4
xor r6, r7      xor r4, r5      xor r4, r3
xor r5, r6      xor r5, r4      xor r3, r4
```

Motivation

- Number and size of RTGs depend on quality of allocation
- Reduction is an NP-complete problem

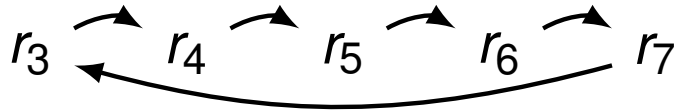


⇒ On standard hardware, implementation may be expensive:
5% to 20% of all generated instructions (SPEC)

Question 1: Is it possible to create an instruction set extension that allows implementing an RTG in one processor cycle?

Motivation

- Number and size of RTGs depend on quality of allocation
- Reduction is an NP-complete problem



⇒ On standard hardware, implementation may be expensive:
5% to 20% of all generated instructions (SPEC)

Question 1: Is it possible to create an instruction set extension that allows implementing an RTG in one processor cycle?

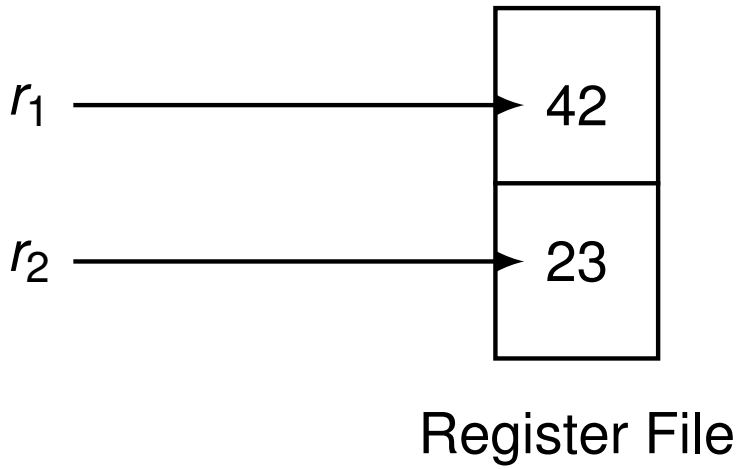
Question 2: Is it worth it?

Fundamental Hardware Constraints

- Changing contents of multiple registers in one cycle very costly

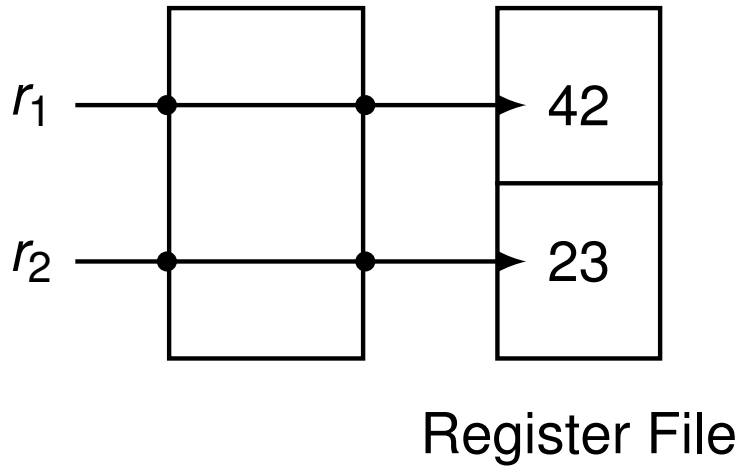
Fundamental Hardware Constraints

- Changing contents of multiple registers in one cycle very costly
- Idea: Modify *access* to register file instead of contents
  - Swap r_1 and r_2: Exchange the access to r_1 and r_2



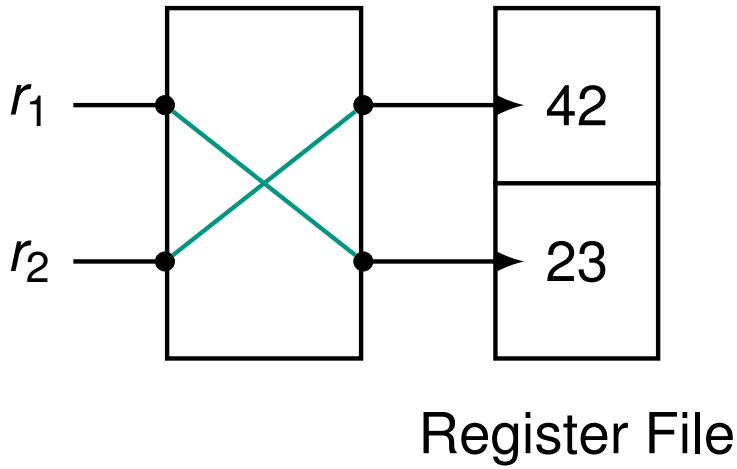
Fundamental Hardware Constraints

- Changing contents of multiple registers in one cycle very costly
- Idea: Modify *access* to register file instead of contents
  - Swap r_1 and r_2: Exchange the access to r_1 and r_2



Fundamental Hardware Constraints

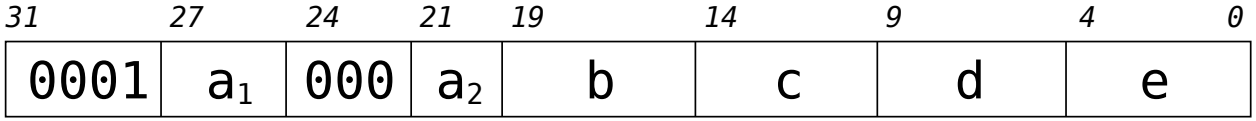
- Changing contents of multiple registers in one cycle very costly
- Idea: Modify *access* to register file instead of contents
  - Swap r_1 and r_2: Exchange the access to r_1 and r_2



$\Rightarrow$ Restriction to *permutations* of registers

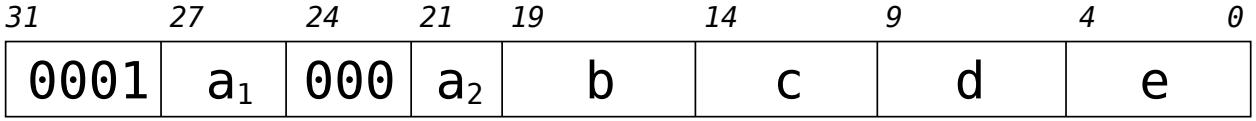
ISA Extension

- Add permutation instructions to SPARC V8 ISA
- 32 registers $\Rightarrow$ 5 bits to identify one register
- 7 bits for opcode $\Rightarrow$ 25 bits left for encoding 5 register numbers



ISA Extension

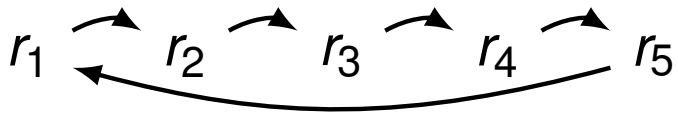
- Add permutation instructions to SPARC V8 ISA
- 32 registers $\Rightarrow$ 5 bits to identify one register
- 7 bits for opcode $\Rightarrow$ 25 bits left for encoding 5 register numbers



Two new instructions:

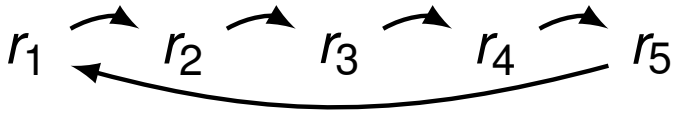
- `permi5`: Implement cyclic RTG with *up to* 5 elements
- `permi23`: Implement two independent cycles with 2 and *up to* 3 elements

Examples




```
permi5 r1, r2, r3, r4, r5
```

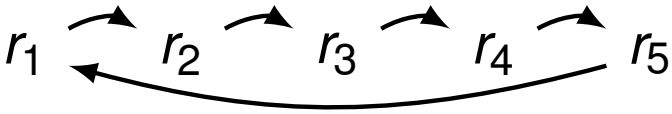
Examples




```
permi5 r1, r2, r3, r4, r5
```


```
permi5 r1, r2
```

Examples




```
permi5 r1, r2, r3, r4, r5
```


```
permi5 r1, r2
```


```
permi23 r1, r2, r3, r4
```

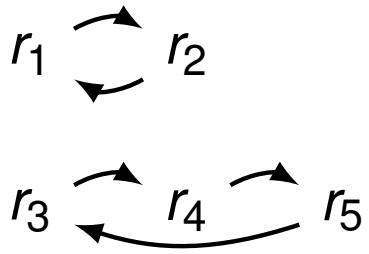
Code Generation

- Goal: Generate efficient code using `permi` instructions for all RTGs
- Question: Which RTGs can be implemented using only `permi`?

Code Generation

- Goal: Generate efficient code using `permi` instructions for all RTGs
- Question: Which RTGs can be implemented using only `permi`?

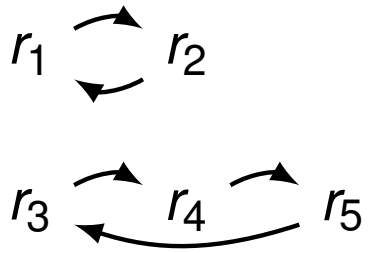
- RTGs in permutation form
  - Permutation can be written as a product of cycles
  - Cycles can be implemented with `permis`



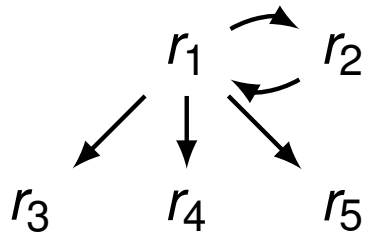
Code Generation

- Goal: Generate efficient code using `permi` instructions for all RTGs
- Question: Which RTGs can be implemented using only `permi`?

- RTGs in permutation form
  - Permutation can be written as a product of cycles
  - Cycles can be implemented with `permis`

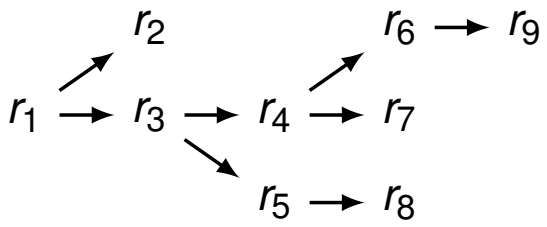


- In general: RTGs can duplicate values
  - Permutations are injective
  - Value duplication impossible



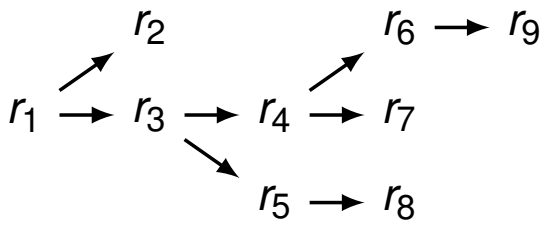
Two-Phase Approach

Arbitrary RTG

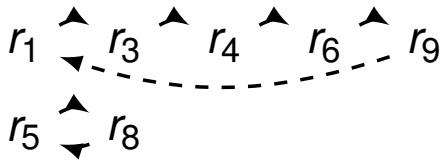


Two-Phase Approach

Arbitrary RTG



Phase 1:
Conversion

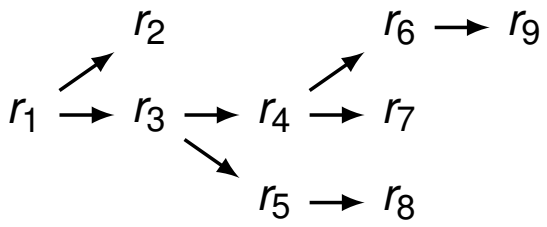


+

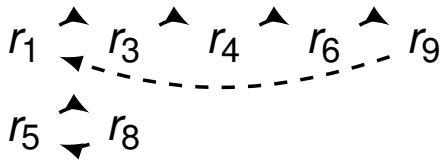
```
mov r3, r2
mov r6, r7
mov r4, r5
```

Two-Phase Approach

Arbitrary RTG



Phase 1:
Conversion



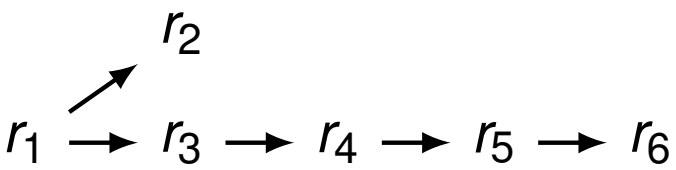
+

```
mov r3, r2
mov r6, r7
mov r4, r5
```

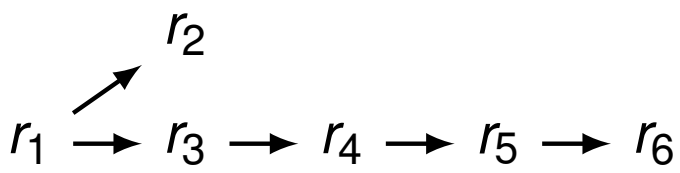
Phase 2:
Decomposition

```
permi5 r1, r3, r4, r6, r9
permi5 r5, r8
```

Conversion into Permutation Form

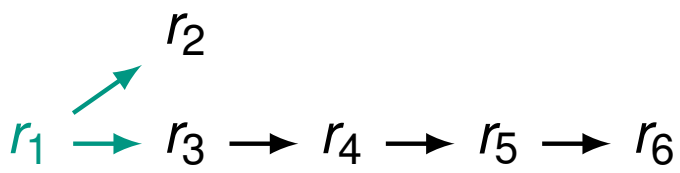


Conversion into Permutation Form



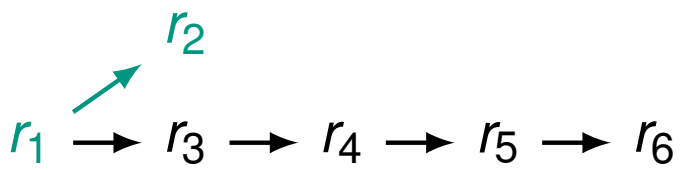
- At each node with > 1 outgoing edge: keep edge that is part of longest path starting at node

Conversion into Permutation Form



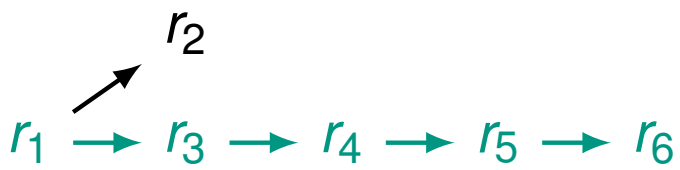
- At each node with > 1 outgoing edge: keep edge that is part of longest path starting at node

Conversion into Permutation Form



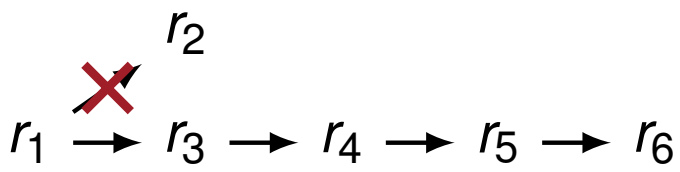
- At each node with > 1 outgoing edge: keep edge that is part of longest path starting at node

Conversion into Permutation Form



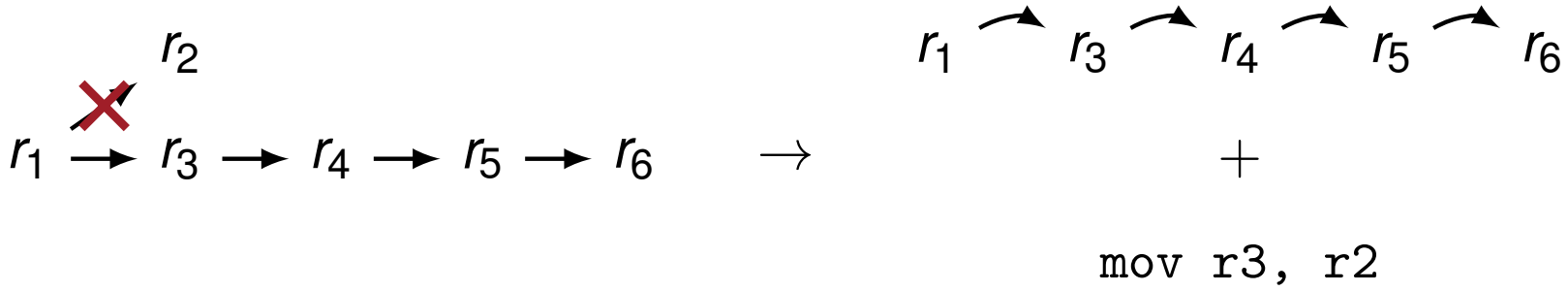
- At each node with > 1 outgoing edge: keep edge that is part of longest path starting at node

Conversion into Permutation Form



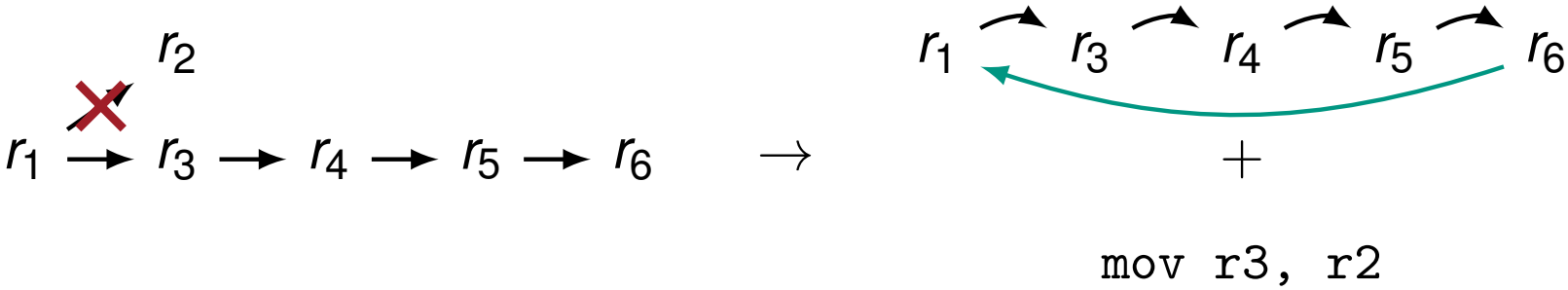
- At each node with > 1 outgoing edge: keep edge that is part of longest path starting at node

Conversion into Permutation Form



- At each node with > 1 outgoing edge: keep edge that is part of longest path starting at node

Conversion into Permutation Form



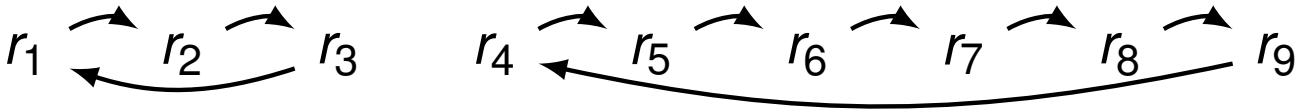
- At each node with > 1 outgoing edge: keep edge that is part of longest path starting at node

Decomposition into Cycles

- After conversion: Implement RTG in permutation form with as few `permi`s as possible
- Need to combine multiple cycles to exploit `permi23`

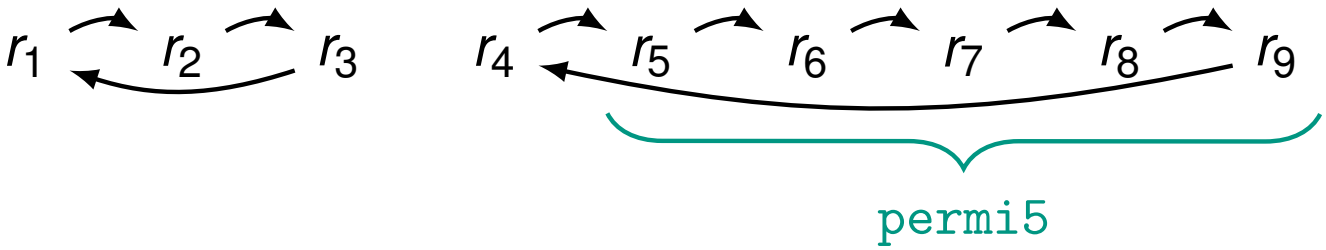
Decomposition into Cycles

- After conversion: Implement RTG in permutation form with as few `permi`s as possible
- Need to combine multiple cycles to exploit `permi23`



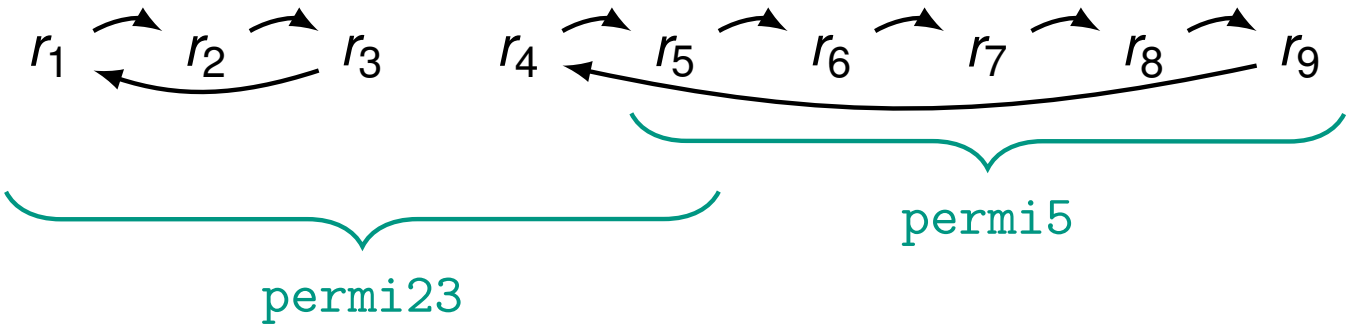
Decomposition into Cycles

- After conversion: Implement RTG in permutation form with as few `permi`s as possible
- Need to combine multiple cycles to exploit `permi23`



Decomposition into Cycles

- After conversion: Implement RTG in permutation form with as few `permi`s as possible
- Need to combine multiple cycles to exploit `permi23`



Decomposition into Cycles

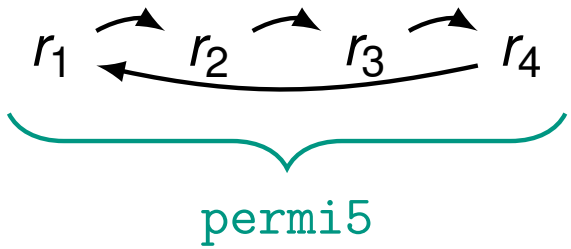
- Greedy decomposition algorithm with linear runtime

- **Phase 1**
  - While there is a cycle of size 4 or more: use `permi5` to implement it

Decomposition into Cycles

- Greedy decomposition algorithm with linear runtime

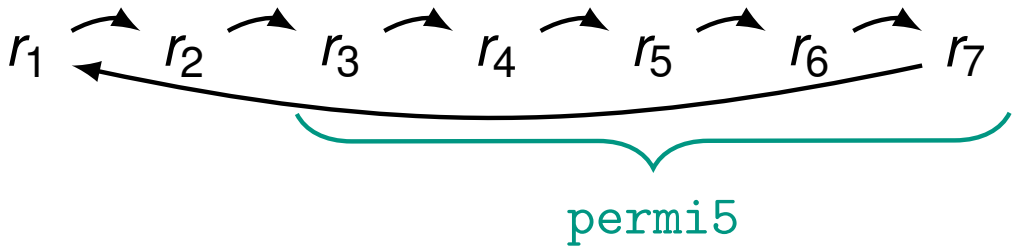
- **Phase 1**
  - While there is a cycle of size 4 or more: use `permi5` to implement it



Decomposition into Cycles

- Greedy decomposition algorithm with linear runtime

- **Phase 1**
  - While there is a cycle of size 4 or more: use `permi5` to implement it



Decomposition into Cycles

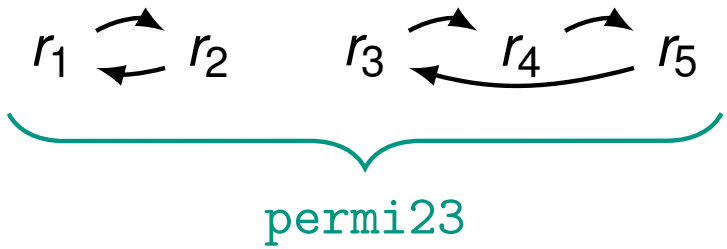
- Greedy decomposition algorithm with linear runtime

- **Phase 1**
  - While there is a cycle of size 4 or more: use `permi5` to implement it
- **Phase 2**: Only cycles of size ≤ 3 left

Decomposition into Cycles

- Greedy decomposition algorithm with linear runtime

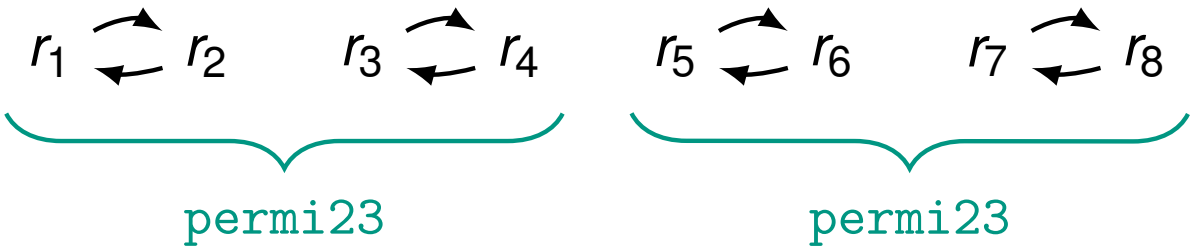
- **Phase 1**
  - While there is a cycle of size 4 or more: use `permi5` to implement it
- **Phase 2**: Only cycles of size ≤ 3 left
  - If 2-cycle and 3-cycle available: combine using `permi23`



Decomposition into Cycles

- Greedy decomposition algorithm with linear runtime

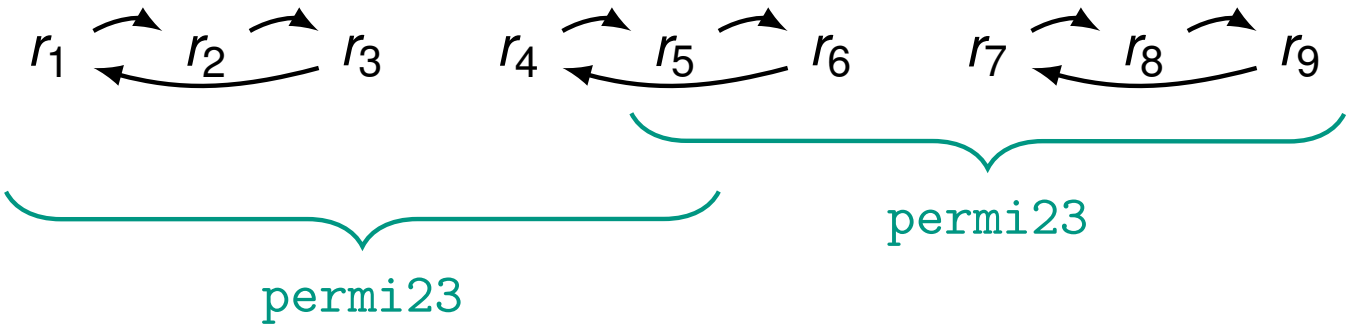
- **Phase 1**
  - While there is a cycle of size 4 or more: use `permi5` to implement it
- **Phase 2**: Only cycles of size ≤ 3 left
  - If 2-cycle and 3-cycle available: combine using `permi23`
  - If only 2-cycles available: combine in pairs using `permi23`



Decomposition into Cycles

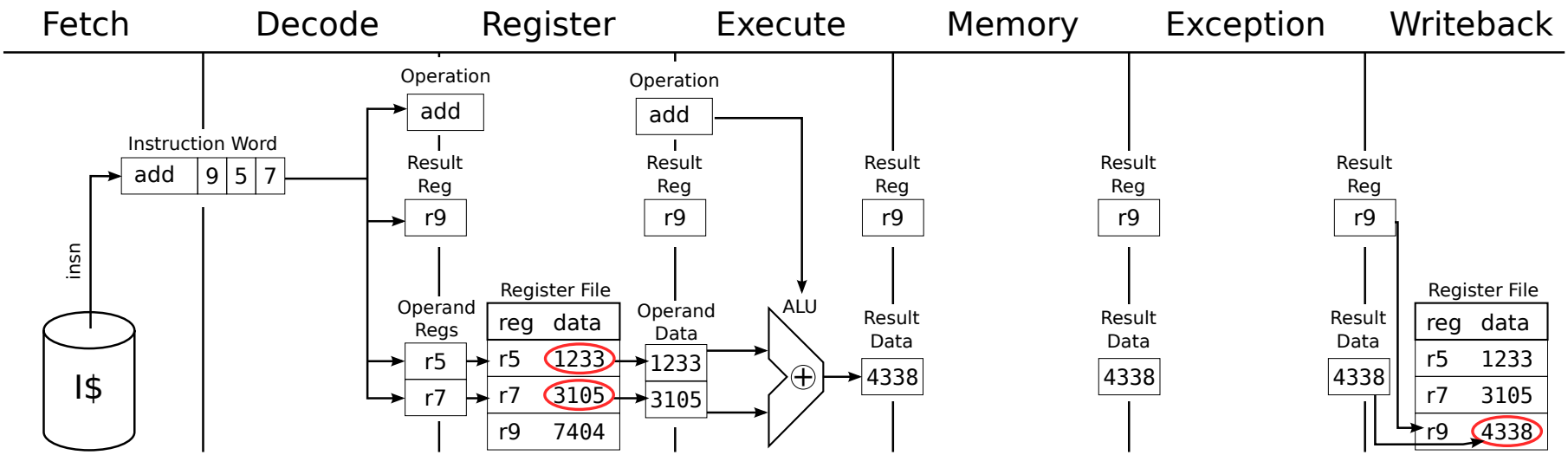
- Greedy decomposition algorithm with linear runtime

- **Phase 1**
  - While there is a cycle of size 4 or more: use `permi5` to implement it
- **Phase 2**: Only cycles of size ≤ 3 left
  - If 2-cycle and 3-cycle available: combine using `permi23`
  - If only 2-cycles available: combine in pairs using `permi23`
  - If only 3-cycles available: combine in groups of three using `permi23`



Base Architecture

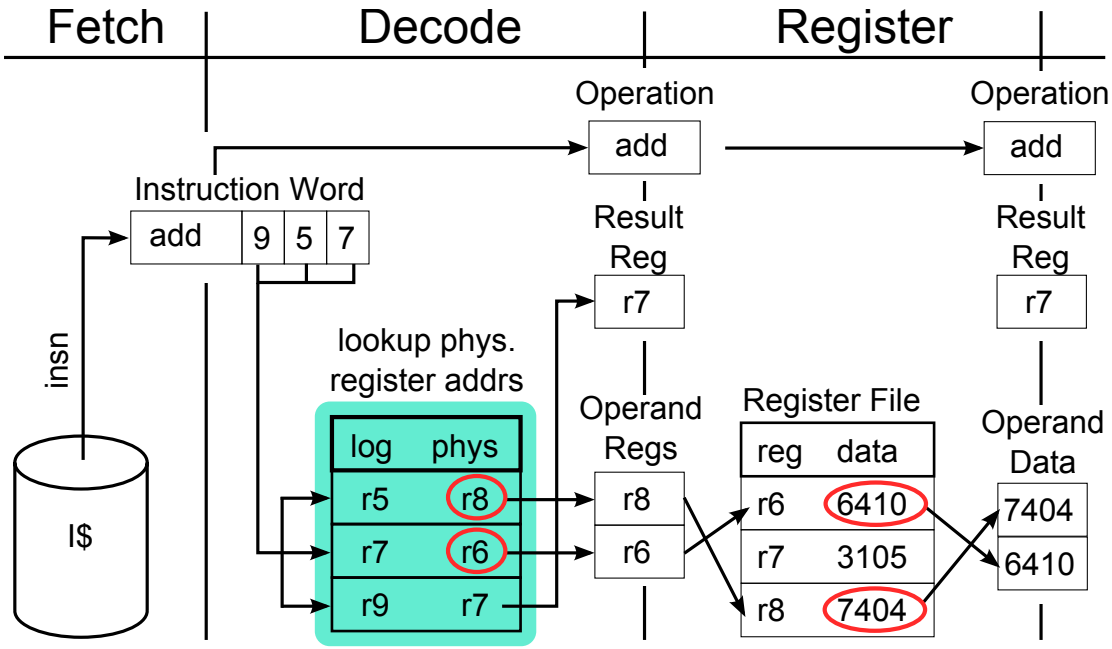
- Underlying architecture: Gaisler LEON3, 7-stage pipeline
- Example: `add r9 r5 r7`



- For `permi` support: modifications of **Decode** and **Exception** stages

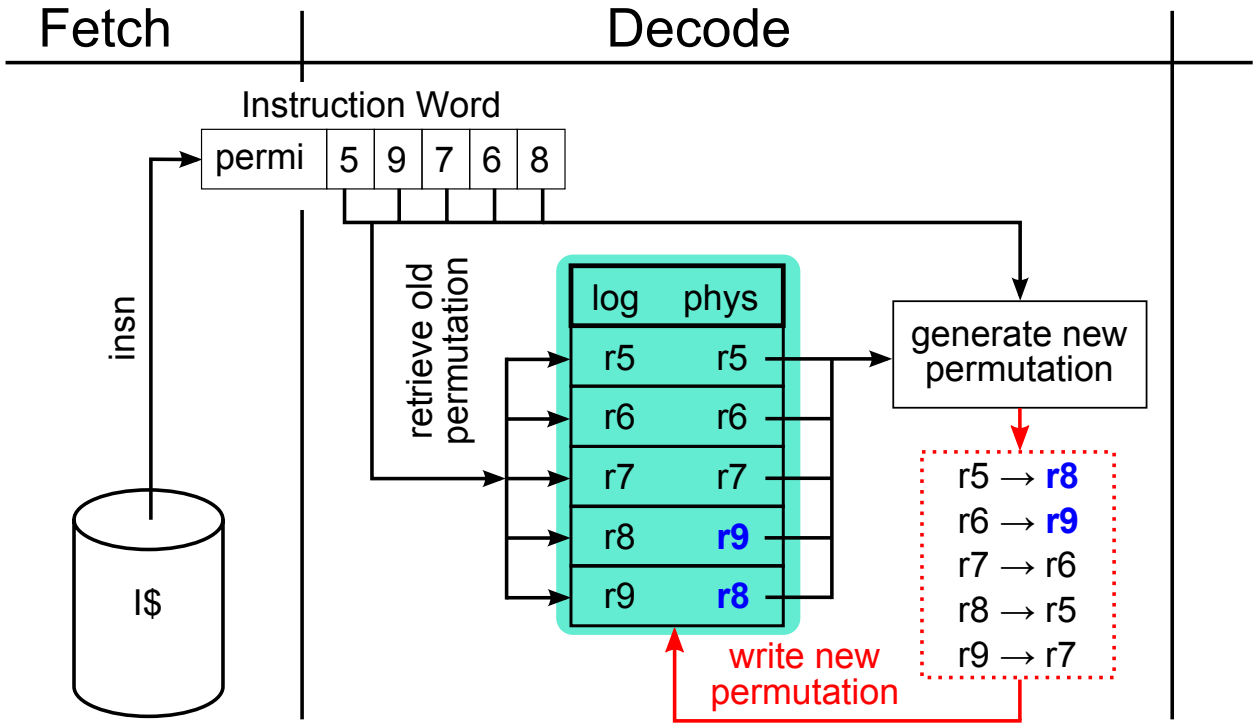
Permutation Support

- Key component: permutation table in Decode stage
  - Contains mapping logical → physical register address
  - Physical address from permutation table used when accessing register file
- Initialized with *identity* at system reset



Applying new Permutations

- Applying permutation `permi5 r5 r9 r7 r6 r8`



- Permutation applied in Decode stage (*early committing*)
  - No changes to forwarding logic required

Results

Experimental evaluation

- Implemented code generation strategy in libFIRM
- Used SPEC CPU2000 benchmark suite as input programs
- Modified SPARC emulator to support `permi` instructions
  - Ability to get precise dynamic instruction counts
- Validation by measurements on FPGA prototype implementation
  - By running Linux on FPGA prototype, ability to reuse executables

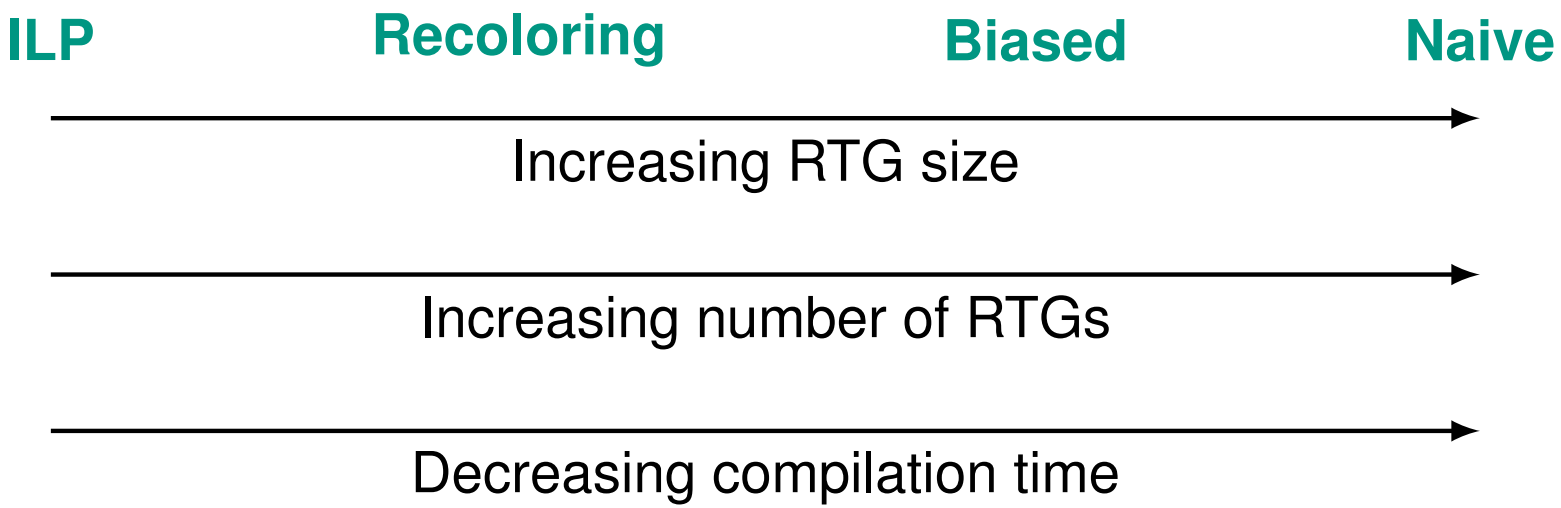
Compile Time

	Default [ms]	Our code gen. [ms]	Relative
Backend (total)	63 598.0	63 927.0	**+0.5%**

- Code generation does not cause significant overhead

Code Quality

Four different register allocator configurations:



Code Quality

Benchmark	ILP	Recoloring	Biased	Naive
164.gzip	−0.7%	−1.0%	−1.9%	**−16.4%**
175.vpr	−0.3%	−0.3%	−1.0%	−3.4%
176.gcc	−0.4%	−0.5%	−2.7%	−11.4%
181.mcf	**−1.9%**	**−1.9%**	−2.8%	−7.8%
186.crafty	−1.0%	−0.8%	−3.9%	−15.2%
197.parser	−0.9%	−1.0%	−2.7%	−12.6%
253.perlbmk	−0.6%	−0.1%	−1.8%	−9.9%
254.gap	−0.3%	−0.9%	−2.0%	−7.1%
255.vortex	−0.5%	−0.8%	**−5.1%**	−15.1%
256.bzip2	−0.3%	−0.6%	−3.1%	−11.3%
300.twolf	−0.3%	−0.3%	−0.8%	−1.9%

- Relative change of number of executed instructions

Code Quality

Benchmark	ILP	Recoloring	Biased	Naive
164.gzip	−0.7%	−1.0%	−1.9%	**−16.4%**
175.vpr	−0.3%	−0.3%	−1.0%	−3.4%
176.gcc	−0.4%	−0.5%	−2.7%	−11.4%
181.mcf	**−1.9%**	**−1.9%**	−2.8%	−7.8%
186.crafty	−1.0%	−0.8%	−3.9%	−15.2%
197.parser	−0.9%	−1.0%	−2.7%	−12.6%
253.perlbmk	−0.6%	−0.1%	−1.8%	−9.9%
254.gap	−0.3%	−0.9%	−2.0%	−7.1%
255.vortex	−0.5%	−0.8%	**−5.1%**	−15.1%
256.bzip2	−0.3%	−0.6%	−3.1%	−11.3%
300.twolf	−0.3%	−0.3%	−0.8%	−1.9%

- Relative change of number of executed instructions
- Universal reduction, up to 5.1% for realistic scenarios
- The worse the register allocation, the higher the benefit using `permis`
- Confirmation by FPGA measurements, speedup up to 1.07

Area Overhead

	Base system	Our system	Overhead
Frequency	80 MHz	80 MHz	0%
BlockRAMs	28	28	0%
Flip-flops	7 607	8 851	**16%**
LUTs	15 024	21 630	**44%**
Slices	7 249	9 507	**31%**

- Frequency unaffected
- Logical-physical mapping $\Rightarrow$ increase in FF usage
- Large multiplexers $\Rightarrow$ increase in LUT usage
  - Considerably smaller overhead for ASIC implementation

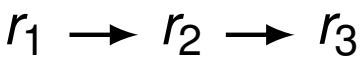
Conclusion

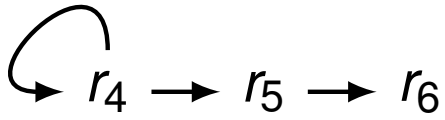
Summary

- Novel approach to accelerate shuffle code by hardware extension
- New instructions added to standard instruction set
- Code generation approach producing efficient code fast
- Extensive evaluation including FPGA prototype implementation
- Universal speedup, instruction count reduction up to 5.1%

Backup Slides

RTG Semantics



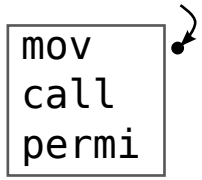


RTG Semantics

Exception Handling

- Early committing can cause problems due to traps
  - Timer interrupts to invoke OS scheduler
  - SPARC window overflows/underflows caused by nested function calls
- Trap handling in LEON3:

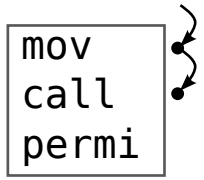
Fetch	Decode	Register	Execute	Memory	Exception	Writeback
-	-	-	permi	call	mov	-



Exception Handling

- Early committing can cause problems due to traps
  - Timer interrupts to invoke OS scheduler
  - SPARC window overflows/underflows caused by nested function calls
- Trap handling in LEON3:

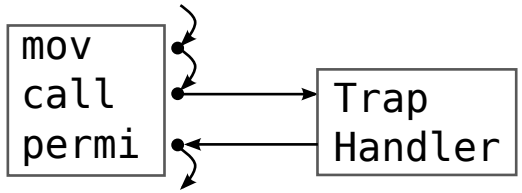
Fetch	Decode	Register	Execute	Memory	Exception	Writeback
-	-	-	-	permi	call	mov



Exception Handling

- Early committing can cause problems due to traps
  - Timer interrupts to invoke OS scheduler
  - SPARC window overflows/underflows caused by nested function calls
- Trap handling in LEON3:

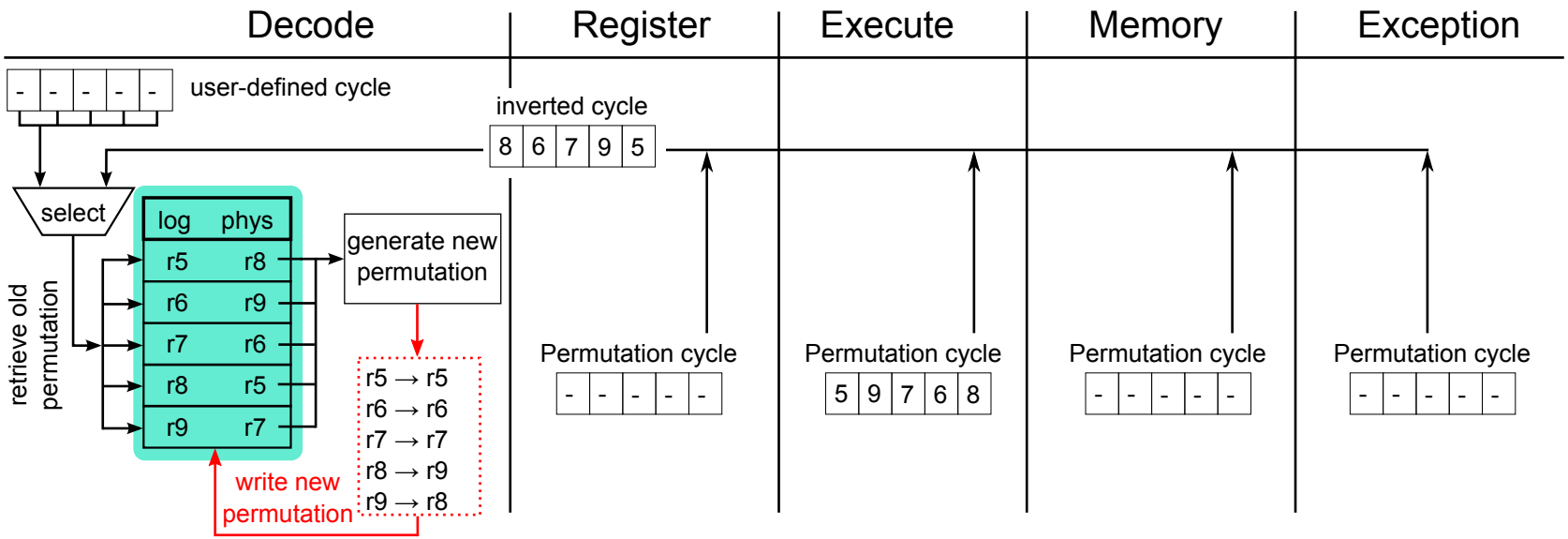
Fetch	Decode	Register	Execute	Memory	Exception	Writeback
`permi`	-	-	-	-	-	-



- `permi` executed twice – permutation applied twice $\rightarrow$ program crash
- Instructions that commit after exception stage can be annulled
- `permi`: revert effect of permutations executed before trap

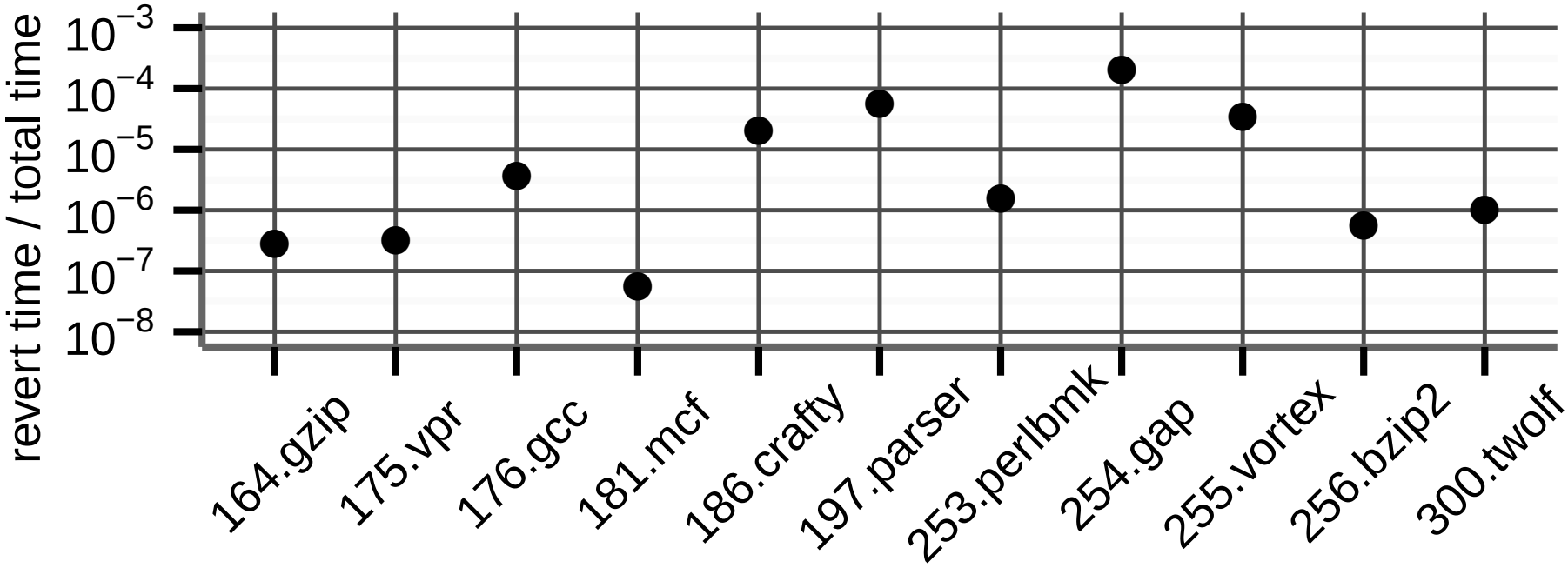
Reverting Permutations

- Permutation history buffer tracks last 4 instructions
- Exception Stage: if a trap occurs, check permutation history buffer for `permi` instructions
- If any occur, go through history buffer in reverse order
  - For each `permi`: apply inverse permutation to permutation table



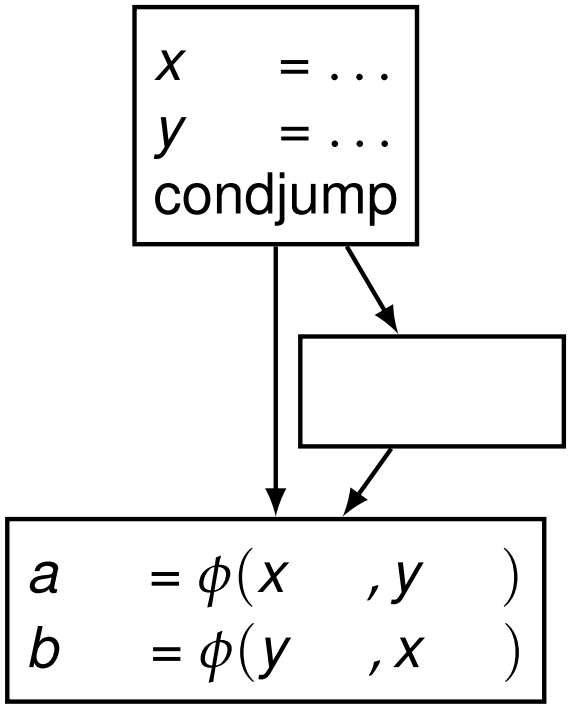
- `permi`s will be re-executed after trap handler
  - ⇒ Register File in expected state

Reversion Effects



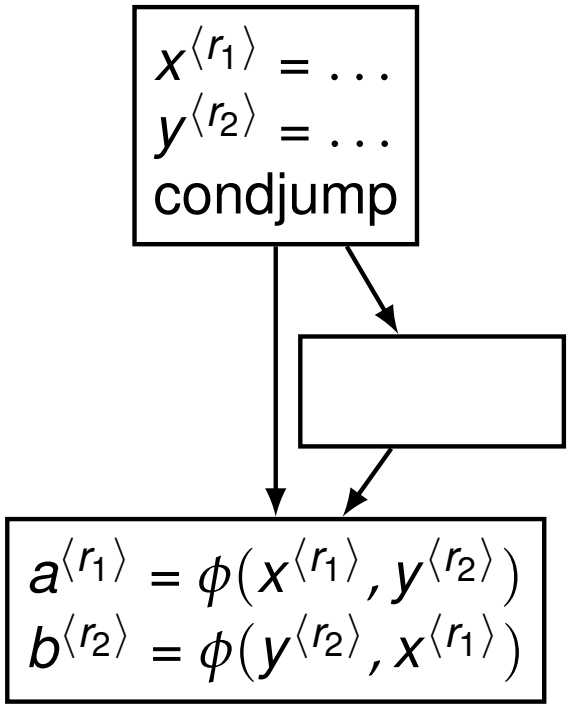
ϕ-functions

```
x = ...;
y = ...;
if (...) {
  t = x;
  x = y;
  y = t;
}
a = x;
b = y;
```



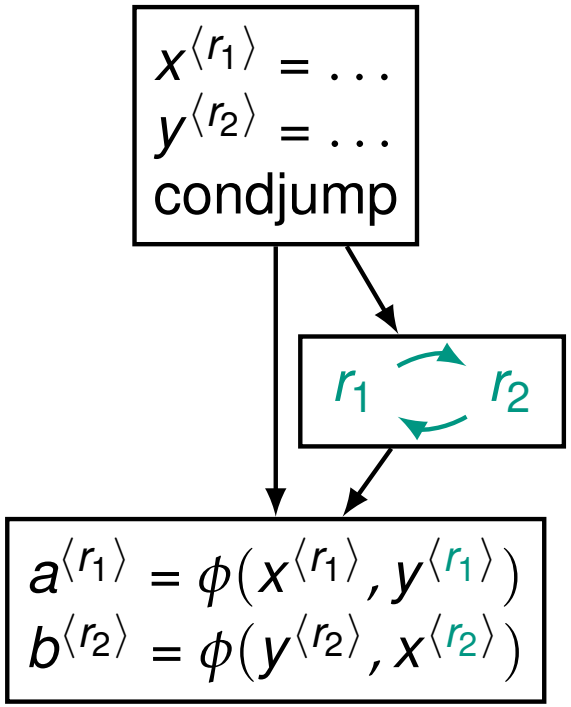
ϕ-functions

```
x = ...;
y = ...;
if (...) {
  t = x;
  x = y;
  y = t;
}
a = x;
b = y;
```



ϕ-functions

```
x = ...;
y = ...;
if (...) {
  t = x;
  x = y;
  y = t;
}
a = x;
b = y;
```

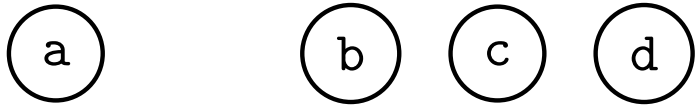


Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures

Manuel Mohr

Lehrstuhl für Programmierparadigmen, Karlsruher Institut für Technologie (KIT)

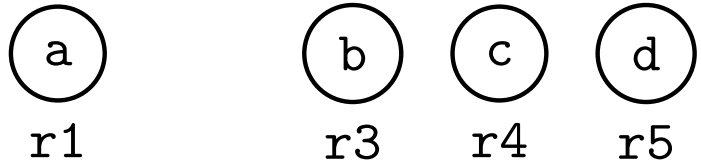
Register Allocation and Shuffle Code




```
a = 10;
b = 20;
c = 30;
d = 40;
...
print(a, a, c, b, d);
```

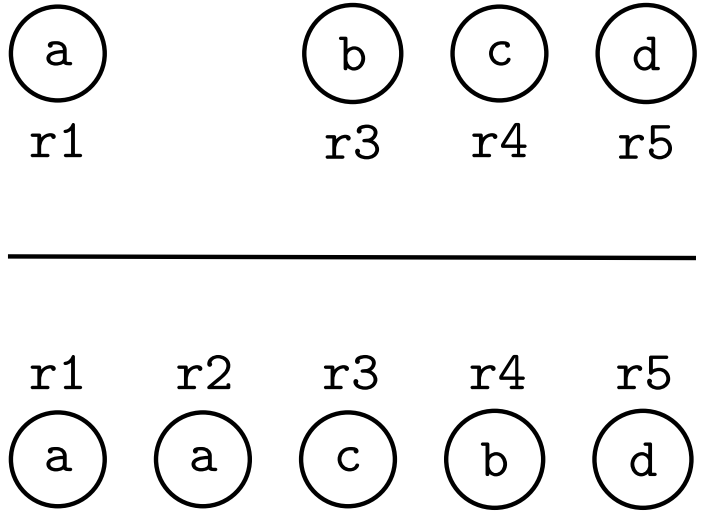
Register Allocation and Shuffle Code

```
a = 10;
b = 20;
c = 30;
d = 40;
...
print(a, a, c, b, d);
```



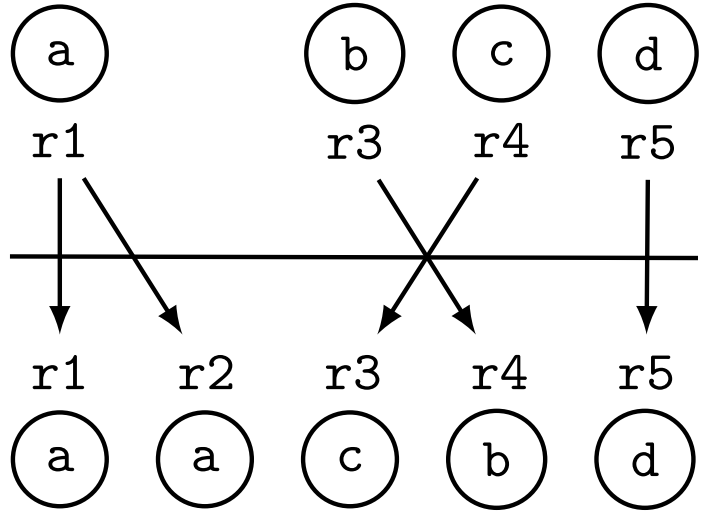
Register Allocation and Shuffle Code

```
a = 10;
b = 20;
c = 30;
d = 40;
...
print(a, a, c, b, d);
```



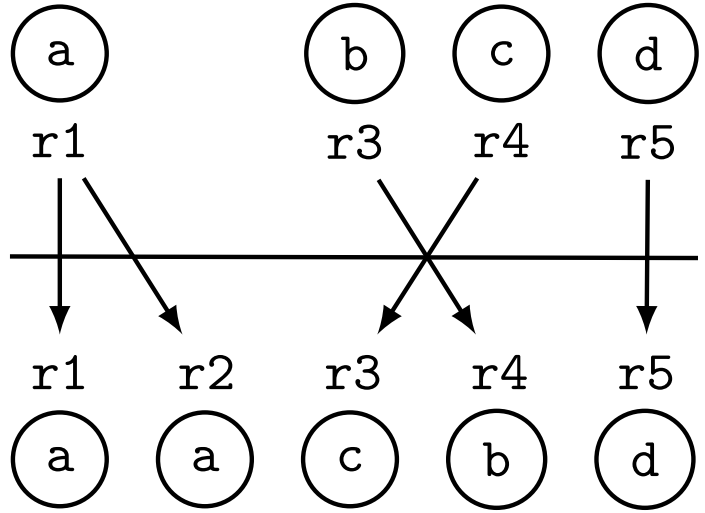
Register Allocation and Shuffle Code

```
a = 10;
b = 20;
c = 30;
d = 40;
...
print(a, a, c, b, d);
```

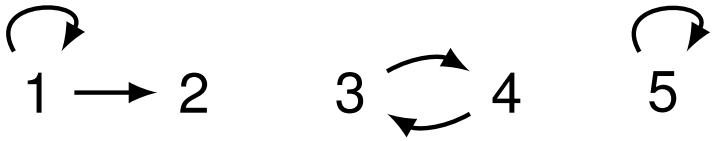


Register Allocation and Shuffle Code

```
a = 10;
b = 20;
c = 30;
d = 40;
...
print(a, a, c, b, d);
```

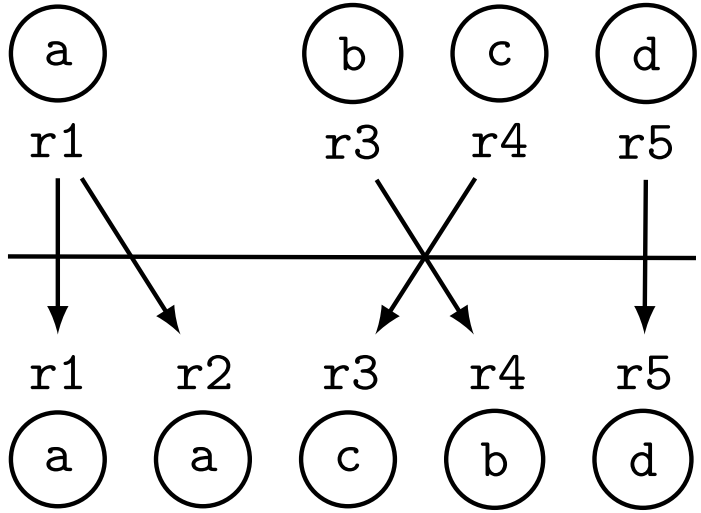


Register Transfer Graph (RTG)

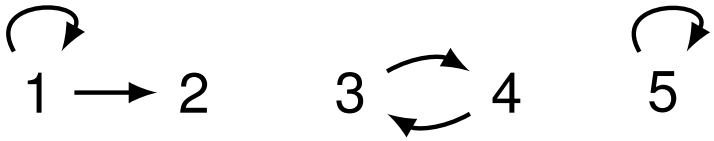


Register Allocation and Shuffle Code

```
a = 10;
b = 20;
c = 30;
d = 40;
...
print(a, a, c, b, d);
```



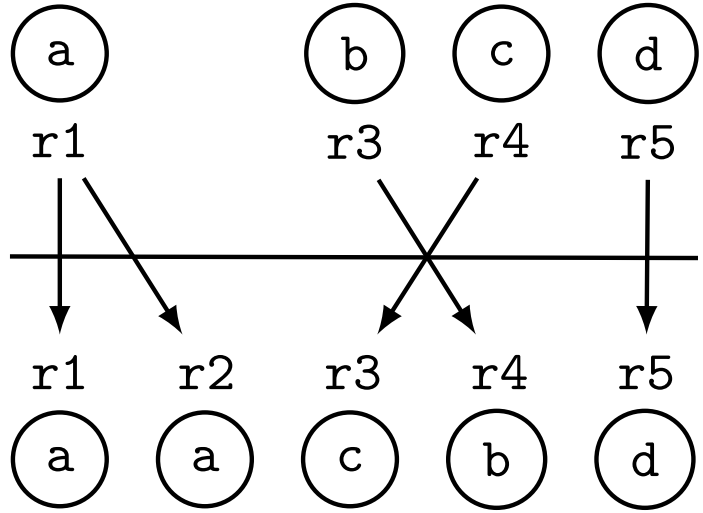
Register Transfer Graph (RTG)



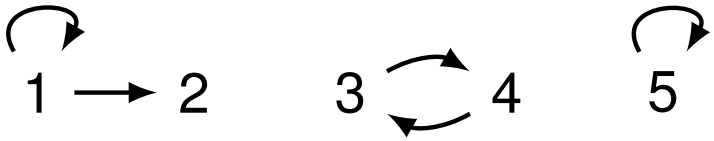

```
copy r1, r2
swap r3, r4
```

Register Allocation and Shuffle Code

```
a = 10;
b = 20;
c = 30;
d = 40;
...
print(a, a, c, b, d);
```



Register Transfer Graph (RTG)




```
copy r1, r2
swap r3, r4
```

Shuffle Code Generation Problem

Find a shortest shuffle code that **implements** a given RTG.

Permutation Instructions

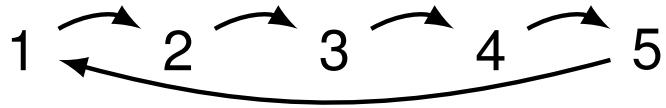
- For large RTGs, implementations can get lengthy

⇒ Proposed more powerful permutation instructions [Mohr et al. 2013]

Permutation Instructions

- For large RTGs, implementations can get lengthy

⇒ Proposed more powerful permutation instructions [Mohr et al. 2013]



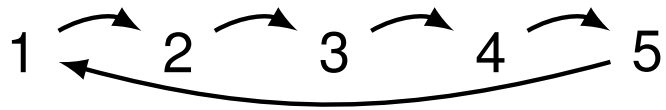

```
permi5 r1, r2, r3, r4, r5
```


```
permi5 r1, r2
```

Permutation Instructions

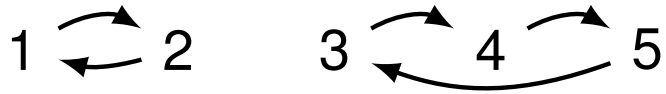
- For large RTGs, implementations can get lengthy

⇒ Proposed more powerful permutation instructions [Mohr et al. 2013]




```
permi5 r1, r2, r3, r4, r5
```


```
permi5 r1, r2
```



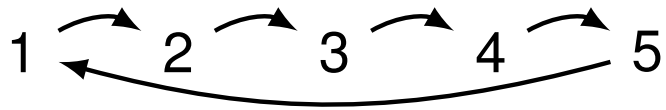

```
permi23 r1, r2, r3, r4, r5
```


```
permi23 r1, r2, r3, r4
```

Permutation Instructions

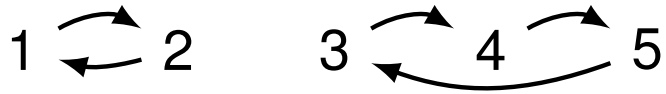
- For large RTGs, implementations can get lengthy

⇒ Proposed more powerful permutation instructions [Mohr et al. 2013]




```
permi5 r1, r2, r3, r4, r5
```

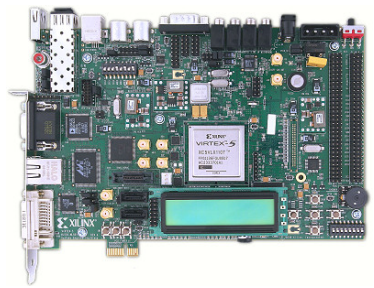

```
permi5 r1, r2
```




```
permi23 r1, r2, r3, r4, r5
```


```
permi23 r1, r2, r3, r4
```

- Exists as FPGA-based prototype processor with modified compiler [Mohr et al. 2013]
  - Improves performance in practice
  - Uses greedy heuristic



Shuffle Code with Permutation Instructions

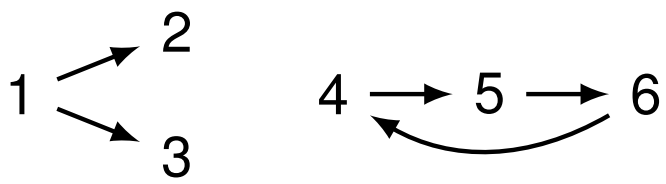
SCGP with permutation instructions

Given RTG, find a shortest shuffle code using `permi5`, `permi23` and `copy`.

Shuffle Code with Permutation Instructions

SCGP with permutation instructions

Given RTG, find a shortest shuffle code using `permi5`, `permi23` and `copy`.



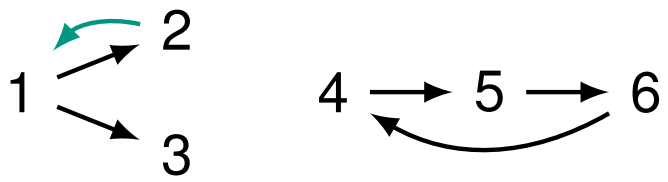
Naive implementation:

```
copy r1, r2
copy r1, r3
permi5 r4, r5, r6
```

Shuffle Code with Permutation Instructions

SCGP with permutation instructions

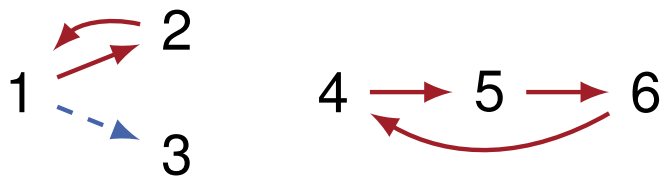
Given RTG, find a shortest shuffle code using `permi5`, `permi23` and `copy`.



Shuffle Code with Permutation Instructions

SCGP with permutation instructions

Given RTG, find a shortest shuffle code using `permi5`, `permi23` and `copy`.



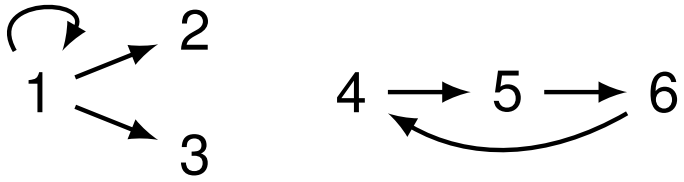
Optimal implementation:

```
permi23 r1, r2, r4, r5, r6
copy r2, r3
```

Shuffle Code with Permutation Instructions

SCGP with permutation instructions

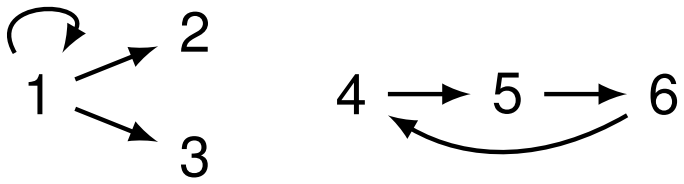
Given RTG, find a shortest shuffle code using `permi5`, `permi23` and `copy`.



Shuffle Code with Permutation Instructions

SCGP with permutation instructions

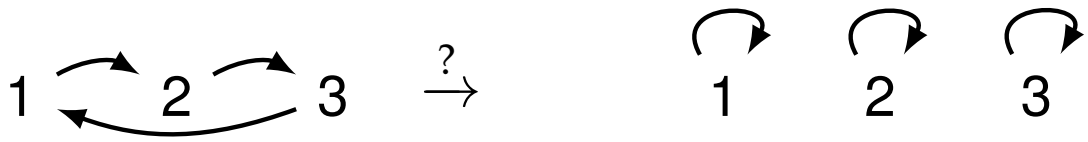
Given RTG, find a shortest shuffle code using `permi5`, `permi23` and `copy`.



Optimal implementation:

```
copy r1, r2
copy r1, r3
permi5 r4, r5, r6
```

Shuffle Code Generation as a Graph Problem

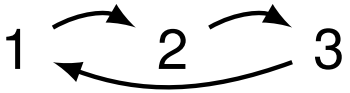


Shuffle Code Generation as a Graph Problem

Effect of a **permutation**:

$$\pi \bullet (V, E) = (V, \{(\pi(u), v) \mid (u, v) \in E\})$$

$$[(1\ 2) \circ (2\ 3)] \bullet$$



$\xrightarrow{?}$

Shuffle Code Generation as a Graph Problem

Effect of a **permutation**:

$$\pi \bullet (V, E) = (V, \{(\pi(u), v) \mid (u, v) \in E\})$$

$$[(1\ 2) \circ (2\ 3)] \bullet$$

$\xrightarrow{?}$

Shuffle Code Generation as a Graph Problem

Effect of a **permutation**:

$$\pi \bullet (V, E) = (V, \{(\pi(u), v) \mid (u, v) \in E\})$$

$$[(1\ 2)] \bullet$$ 

Shuffle Code Generation as a Graph Problem

Effect of a **permutation**:

$$\pi \bullet (V,E) = (V,\{(\pi(u),v) \mid (u,v) \in E\})$$

$$[\ \ \ ] \bullet \overset{\curvearrowright}{1}\ \overset{\curvearrowright}{2}\ \overset{\curvearrowright}{3} = \overset{\curvearrowright}{1}\ \overset{\curvearrowright}{2}\ \overset{\curvearrowright}{3}$$

Shuffle Code Generation as a Graph Problem

Effect of a **permutation**:

$$\pi \bullet (V,E) = (V,\{(\pi(u),v) \mid (u,v) \in E\})$$

- Defines group action of permutations on RTGs
- For permutation RTGs (PRTGs): make given PRTG trivial

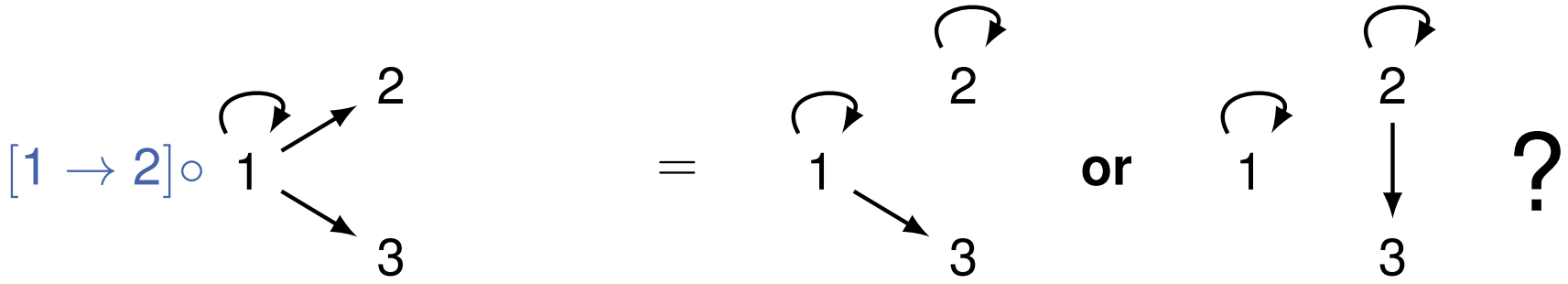
Shuffle Code Generation as a Graph Problem

Effect of a **permutation**:

$$\pi \bullet (V,E) = (V,\{(\pi(u),v) \mid (u,v) \in E\})$$

- Defines group action of permutations on RTGs
- For permutation RTGs (PRTGs): make given PRTG trivial

Effect of a **copy**:



- Copies not expressible in RTGs

Observations on Shuffle Code

Observations on Shuffle Code

Copy $a \to b$ followed by transposition $\tau = (c\ d)$

rewrite to

Transposition $(c\ d)$ followed by copy $\tau(a) \to \tau(b)$

Observations on Shuffle Code

Copy $a \to b$ followed by transposition $\tau = (c\ d)$

rewrite to

Transposition $(c\ d)$ followed by copy $\tau(a) \to \tau(b)$

Observations on Shuffle Code

Copy $a \to b$ followed by transposition $\tau = (c\ d)$

rewrite to

Transposition $(c\ d)$ followed by copy $\tau(a) \to \tau(b)$

Observations on Shuffle Code

Copy $a \to b$ followed by transposition $\tau = (c\ d)$

rewrite to

Transposition $(c\ d)$ followed by copy $\tau(a) \to \tau(b)$

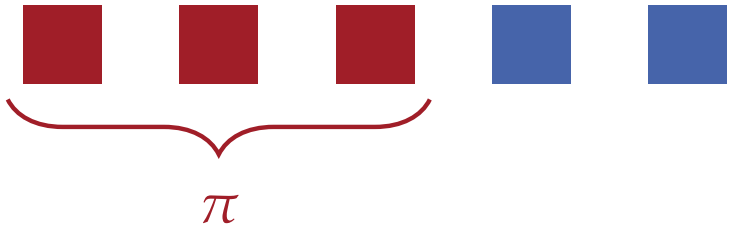
Observations on Shuffle Code

Copy $a \to b$ followed by transposition $\tau = (c\ d)$

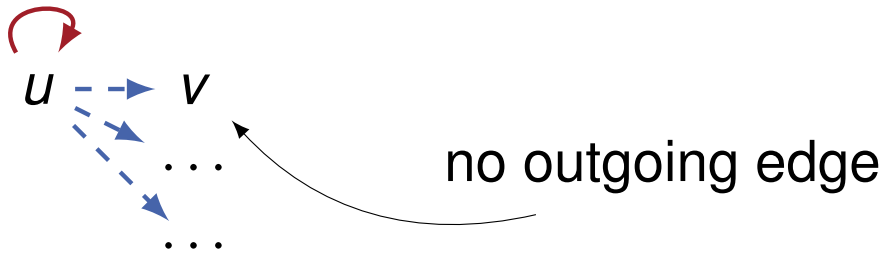
rewrite to

Transposition $(c\ d)$ followed by copy $\tau(a) \to \tau(b)$

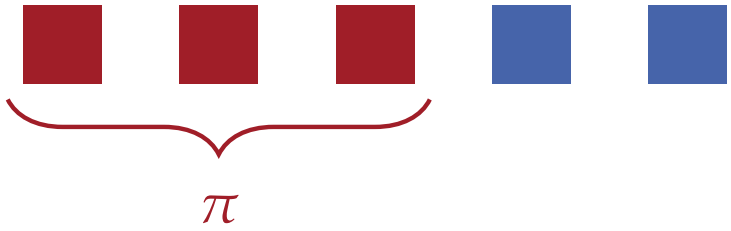
Observations on Shuffle Code



Can transform shuffle code such that in πG, for every copy $u \to v$:

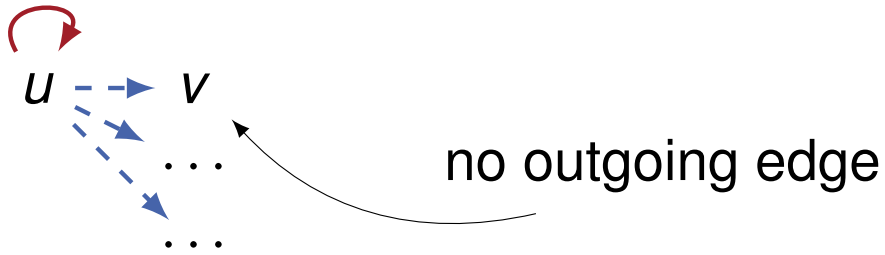


Observations on Shuffle Code

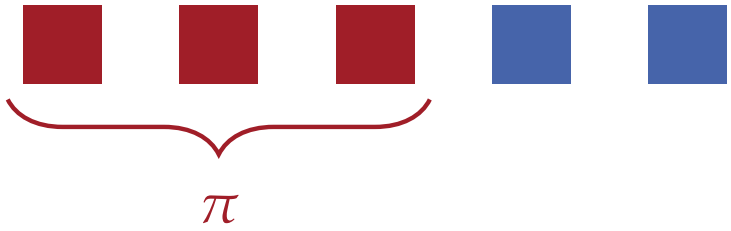


Normalized

Can transform shuffle code such that in πG, for every copy $u \to v$:

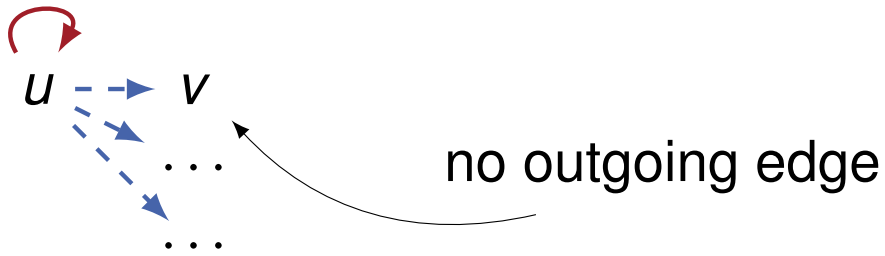


Observations on Shuffle Code



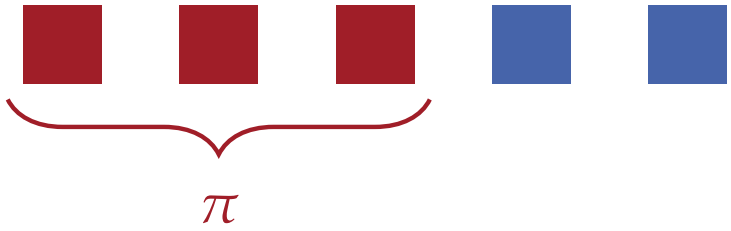
Normalized

Can transform shuffle code such that in πG, for every copy $u \to v$:



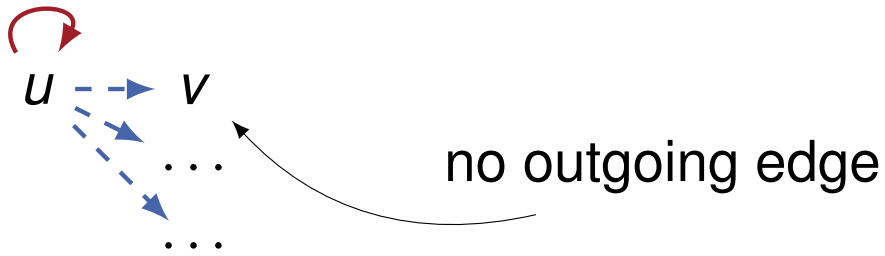
- All outgoing edges of u in πG are copies

Observations on Shuffle Code



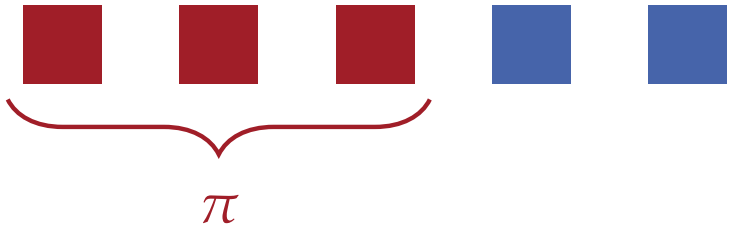
Normalized

Can transform shuffle code such that in πG, for every copy $u \to v$:



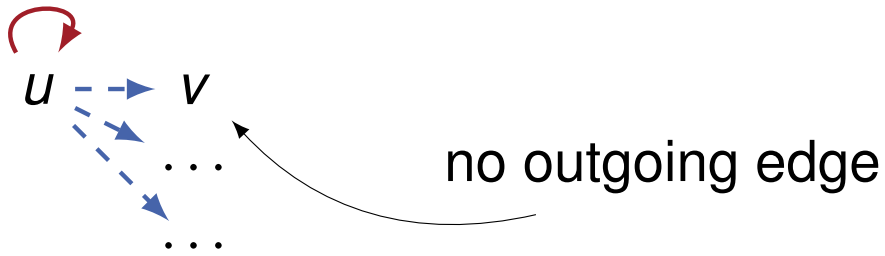
- All outgoing edges (except one) of u in πG are copies

Observations on Shuffle Code



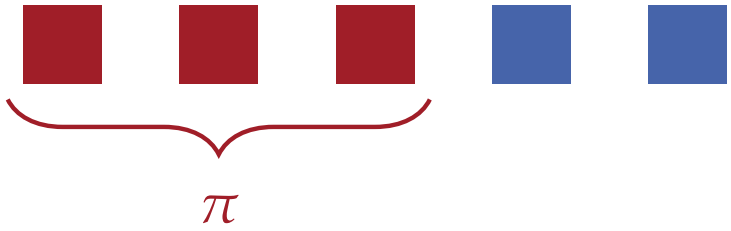
Normalized

Can transform shuffle code such that in πG, for every copy $u \to v$:



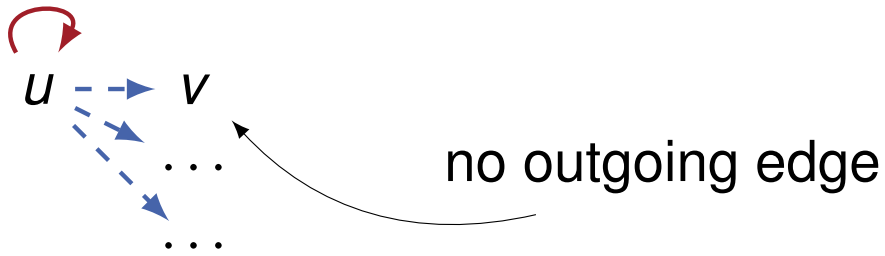
- All outgoing edges (except one) of u in πG are copies
- π permutes edge sources in G

Observations on Shuffle Code



Normalized

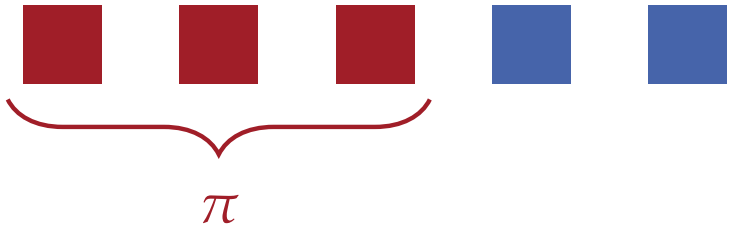
Can transform shuffle code such that in πG, for every copy $u \to v$:



- All outgoing edges (except one) of u in πG are copies
- π permutes edge sources in G

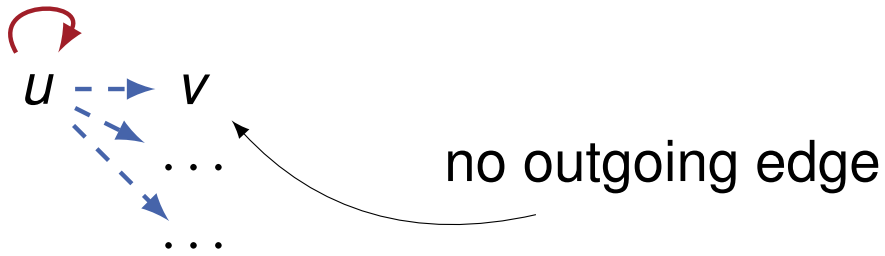
$\Rightarrow$ All outgoing edges (except one) of u in G correspond to copies

Observations on Shuffle Code



Normalized

Can transform shuffle code such that in πG, for every copy $u \to v$:



- All outgoing edges (except one) of u in πG are copies
- π permutes edge sources in G

$\Rightarrow$ All outgoing edges (except one) of u in G correspond to copies
$\to$ **copy set**

Strategy & Outline

Strategy & Outline

Step 1: Pick a copy set C.
(For each vertex, color one outgoing edge red, rest blue.)

Strategy & Outline

Step 1: Pick a copy set C.
(For each vertex, color one outgoing edge red, rest blue.)

Strategy & Outline

Step 1: Pick a copy set C.
(For each vertex, color one outgoing edge red, rest blue.)

Step 2: Find an optimal shuffle code for the red RTG $G - C$.
(Has maximum out-degree 1.)

Strategy & Outline

Step 1: Pick a copy set C.
(For each vertex, color one outgoing edge red, rest blue.)

Step 2: Find an optimal shuffle code for the red RTG $G - C$.
(Has maximum out-degree 1.)

Strategy & Outline

Step 1: Pick a copy set C.
(For each vertex, color one outgoing edge red, rest blue.)

Step 2: Find an optimal shuffle code for the red RTG $G - C$.
(Has maximum out-degree 1.)

Step 3: Implement blue edges using copy operations.

Strategy & Outline

Choice of copy set is crucial:

Step 1: Pick a copy set C.
(For each vertex, color one outgoing edge red, rest blue.)

Step 2: Find an optimal shuffle code for the red RTG $G - C$.
(Has maximum out-degree 1.)

Step 3: Implement blue edges using copy operations.

Strategy & Outline

Choice of copy set is crucial:

Step 1: Pick a copy set C.
(For each vertex, color one outgoing edge red, rest blue.)

Step 2: Find an optimal shuffle code for the red RTG $G - C$.
(Has maximum out-degree 1.)

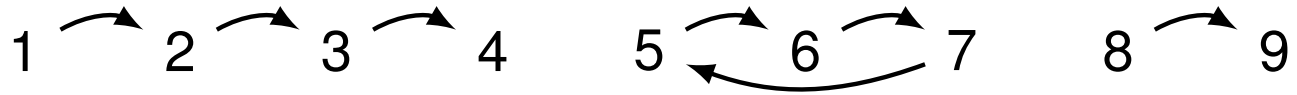
Step 3: Implement blue edges using copy operations.

Problem 1: Given copy set C, compute optimal shuffle code for $G - C$.

Problem 2: Find copy set C where $G - C$ requires fewest instructions.

The GREEDY Algorithm (adapted from [Mohr et al. 2013])

- Complete each directed path into directed cycle

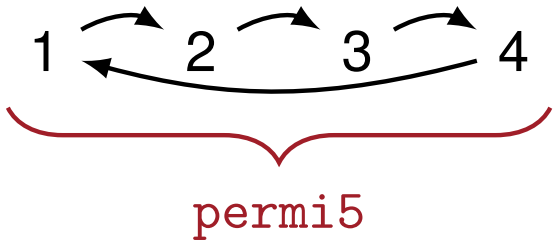


The GREEDY Algorithm (adapted from [Mohr et al. 2013])

- Complete each directed path into directed cycle

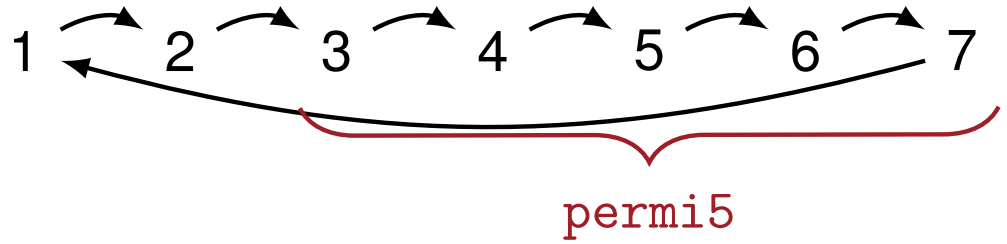
The GREEDY Algorithm (adapted from [Mohr et al. 2013])

- Complete each directed path into directed cycle
- **Phase 1**
  - While there is a cycle K of size 4 or more: use `permi5` to reduce K's size



The GREEDY Algorithm (adapted from [Mohr et al. 2013])

- Complete each directed path into directed cycle
- **Phase 1**
  - While there is a cycle K of size 4 or more: use `permi5` to reduce K's size

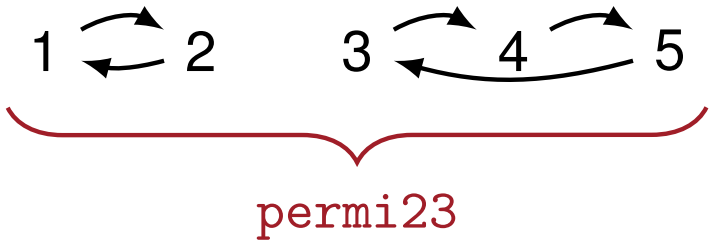


The GREEDY Algorithm (adapted from [Mohr et al. 2013])

- Complete each directed path into directed cycle
- **Phase 1**
  - While there is a cycle K of size 4 or more: use `permi5` to reduce K's size
- **Phase 2**: Only cycles of size ≤ 3 left

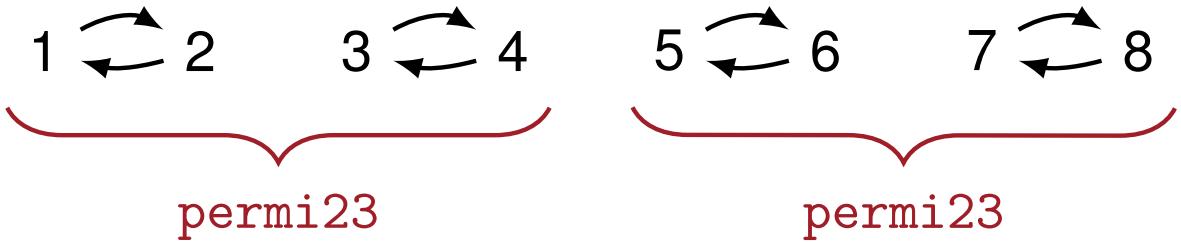
The GREEDY Algorithm (adapted from [Mohr et al. 2013])

- Complete each directed path into directed cycle
- **Phase 1**
  - While there is a cycle K of size 4 or more: use `permi5` to reduce K's size
- **Phase 2**: Only cycles of size ≤ 3 left
  - 2-cycle and 3-cycle available: resolve using `permi23`



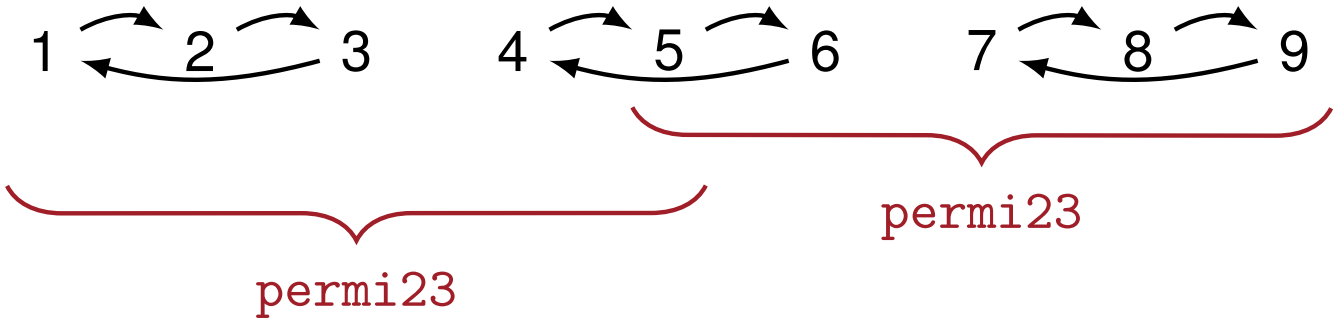
The GREEDY Algorithm (adapted from [Mohr et al. 2013])

- Complete each directed path into directed cycle
- **Phase 1**
  - While there is a cycle K of size 4 or more: use `permi5` to reduce K's size
- **Phase 2**: Only cycles of size ≤ 3 left
  - 2-cycle and 3-cycle available: resolve using `permi23`
  - Only 2-cycles available: resolve pairs using `permi23`



The GREEDY Algorithm (adapted from [Mohr et al. 2013])

- Complete each directed path into directed cycle
- **Phase 1**
  - While there is a cycle K of size 4 or more: use `permi5` to reduce K's size
- **Phase 2**: Only cycles of size ≤ 3 left
  - 2-cycle and 3-cycle available: resolve using `permi23`
  - Only 2-cycles available: resolve pairs using `permi23`
  - Only 3-cycles available: resolve groups of three using `permi23`



The GREEDY Algorithm (adapted from [Mohr et al. 2013])

- Complete each directed path into directed cycle
- **Phase 1**
  - While there is a cycle K of size 4 or more: use `permi5` to reduce K's size
- **Phase 2**: Only cycles of size ≤ 3 left
  - 2-cycle and 3-cycle available: resolve using `permi23`
  - Only 2-cycles available: resolve pairs using `permi23`
  - Only 3-cycles available: resolve groups of three using `permi23`

The GREEDY Algorithm (adapted from [Mohr et al. 2013])

- Complete each directed path into directed cycle
- **Phase 1**
  - While there is a cycle K of size 4 or more: use `permi5` to reduce K's size
- **Phase 2**: Only cycles of size ≤ 3 left
  - 2-cycle and 3-cycle available: resolve using `permi23`
  - Only 2-cycles available: resolve pairs using `permi23`
  - Only 3-cycles available: resolve groups of three using `permi23`

Signature $\text{sig}(G) = (X, a_2, a_3)$

- $X = \Sigma_{\sigma \in G} \lfloor \text{size}(\sigma)/4 \rfloor$
- $a_i = |\{\sigma \in G \mid \text{size}(\sigma) = i \mod 4\}|$

The GREEDY Algorithm (adapted from [Mohr et al. 2013])

- Complete each directed path into directed cycle
- **Phase 1**
  - While there is a cycle K of size 4 or more: use `permi5` to reduce K's size
- **Phase 2**: Only cycles of size ≤ 3 left
  - 2-cycle and 3-cycle available: resolve using `permi23`
  - Only 2-cycles available: resolve pairs using `permi23`
  - Only 3-cycles available: resolve groups of three using `permi23`

Signature $\text{sig}(G) = (X, a_2, a_3)$

- $X = \Sigma_{\sigma \in G} \lfloor \text{size}(\sigma)/4 \rfloor$
- $a_i = |\{\sigma \in G \mid \text{size}(\sigma) = i \mod 4\}|$

Cost function $\text{GREEDY}(G) = X + \begin{cases} \lceil (a_2 + a_3)/2 \rceil & \text{if } a_2 \geq a_3 \\ \lceil (a_2 + 2a_3)/3 \rceil & \text{if } a_2 < a_3 \end{cases}$

The GREEDY Algorithm (adapted from [Mohr et al. 2013])

- Complete each directed path into directed cycle
- **Phase 1**
  - While there is a cycle K of size 4 or more: use `permi5` to reduce K's size
- **Phase 2**: Only cycles of size ≤ 3 left
  - 2-cycle and 3-cycle available: resolve using `permi23`
  - Only 2-cycles available: resolve pairs using `permi23`
  - Only 3-cycles available: resolve groups of three using `permi23`

Signature $\text{sig}(G) = (X, a_2, a_3)$

- $X = \Sigma_{\sigma \in G} \lfloor \text{size}(\sigma)/4 \rfloor$
- $a_i = |\{\sigma \in G \mid \text{size}(\sigma) = i \mod 4\}|$

Cost function $\text{GREEDY}(G) = X + \begin{cases} \lceil (a_2 + a_3)/2 \rceil & \text{if } a_2 \geq a_3 \\ \lceil (a_2 + 2a_3)/3 \rceil & \text{if } a_2 < a_3 \end{cases}$

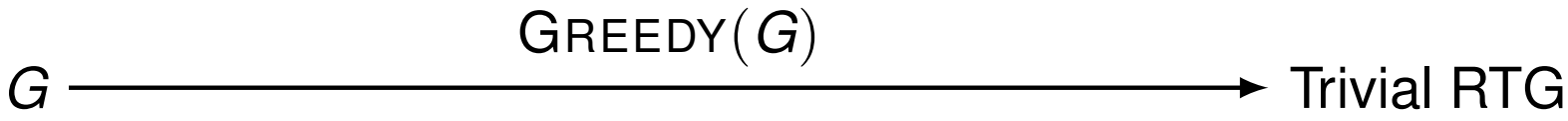
Theorem

GREEDY is optimal for PRTGs.

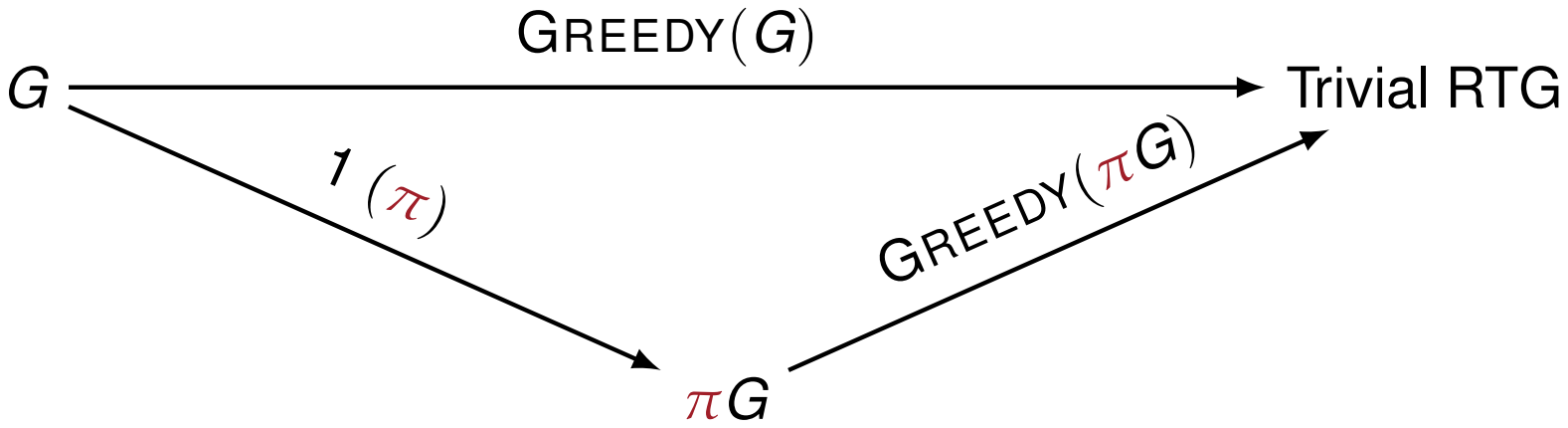
Proof Outline

G

Proof Outline

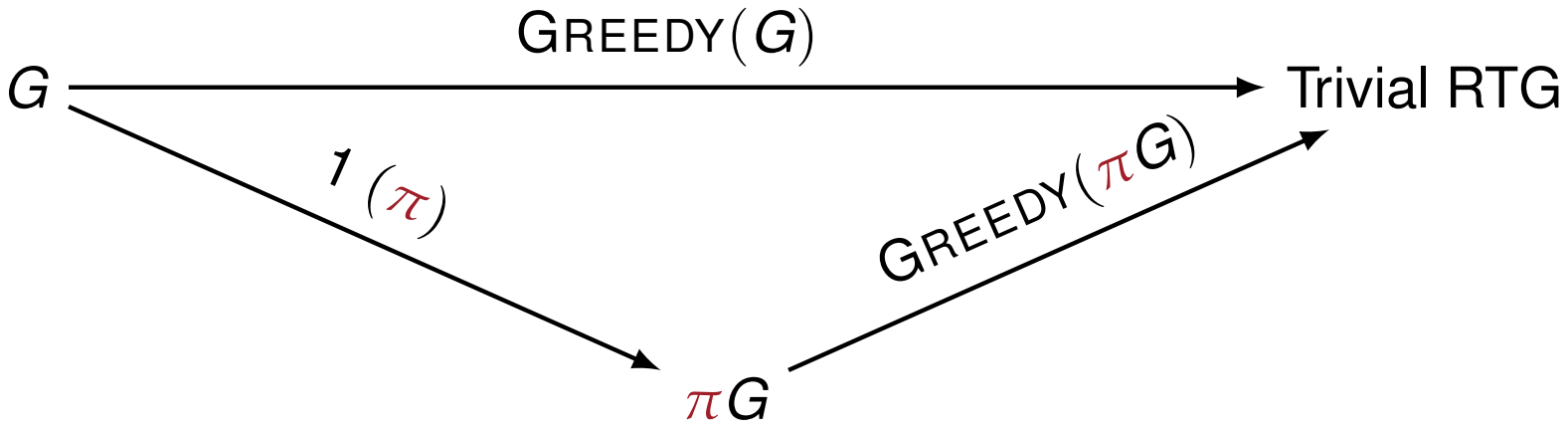


Proof Outline



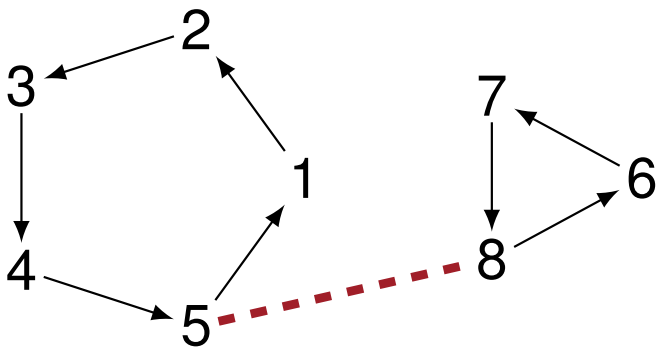
- Take arbitrary permutation instruction π

Proof Outline

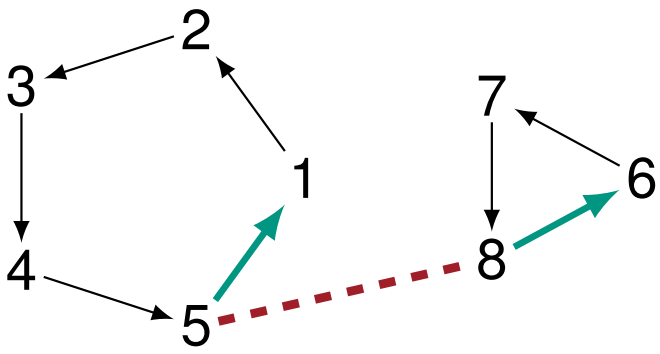


- Take arbitrary permutation instruction π
- **Show** that **always** $\text{GREEDY}(G) \leq \text{GREEDY}(\pi G) + 1$
  $\Leftrightarrow \text{GREEDY}(G) - \text{GREEDY}(\pi G) \leq 1$

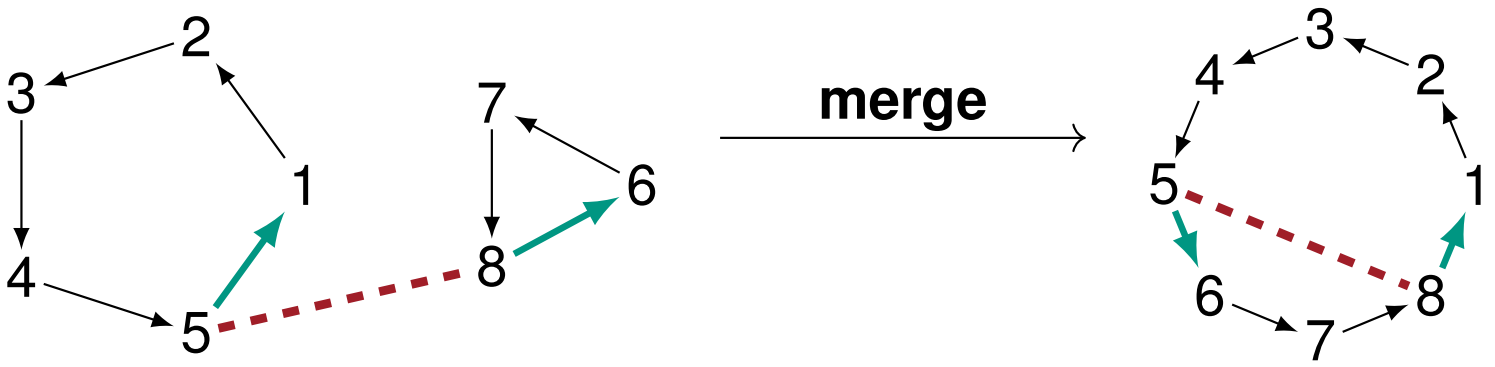
Transpositions



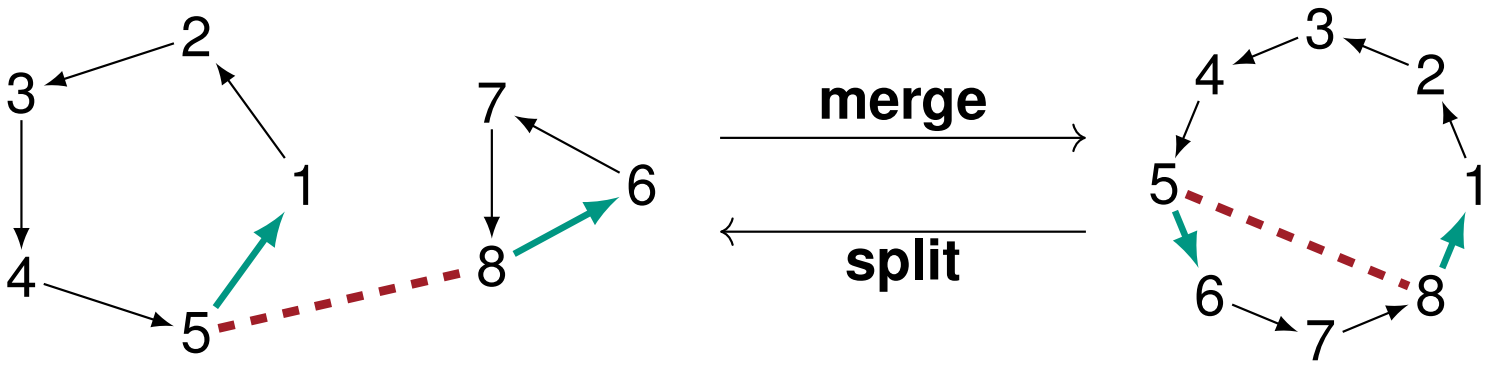
Transpositions



Transpositions



Transpositions



Merges

- Look at all possible signature changes $(\Delta_X, \Delta_2, \Delta_3)$

Merges

Cycle sizes modulo 4

	0	1	2	3
0	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$
1		$(0,1,0)$	$(0,-1,1)$	$(1,0,-1)$
2			$(1,-2,0)$	$(1,-1,-1)$
3				$(1,1,-2)$

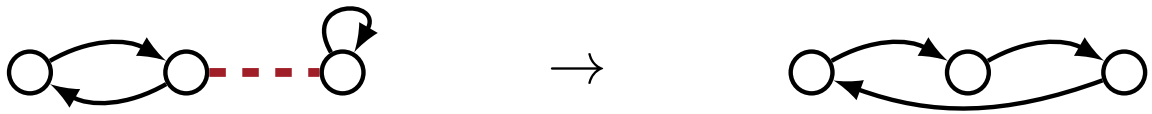
- Look at all possible signature changes $(\Delta_X, \Delta_2, \Delta_3)$

Merges

Cycle sizes modulo 4

	0	1	2	3
0	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$
1		$(0,1,0)$	$(0,-1,1)$	$(1,0,-1)$
2			$(1,-2,0)$	$(1,-1,-1)$
3				$(1,1,-2)$

- Look at all possible signature changes $(\Delta_X, \Delta_2, \Delta_3)$

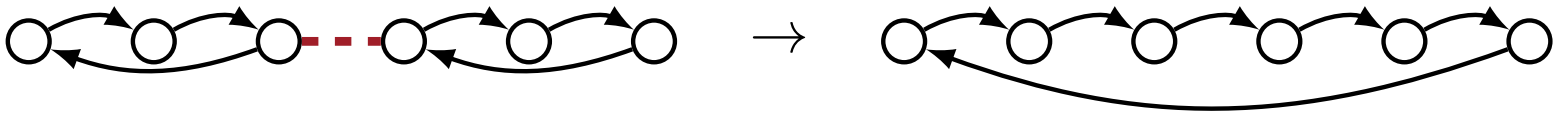


Merges

Cycle sizes modulo 4

	0	1	2	3
0	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$
1		$(0,1,0)$	$(0,-1,1)$	$(1,0,-1)$
2			$(1,-2,0)$	$(1,-1,-1)$
3				$(1,1,-2)$

- Look at all possible signature changes $(\Delta_X, \Delta_2, \Delta_3)$



Merges

Cycle sizes modulo 4

	0	1	2	3
0	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$
1		$(0,1,0)$	$(0,-1,1)$	$(1,0,-1)$
2			$(1,-2,0)$	$(1,-1,-1)$
3				$(1,1,-2)$

- Look at all possible signature changes $(\Delta_X, \Delta_2, \Delta_3)$
- Cost change **only** depends on signature change

Merges

Cycle sizes modulo 4

	0	1	2	3
0	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$
1		$(0,1,0)$	$(0,-1,1)$	$(1,0,-1)$
2			$(1,-2,0)$	$(1,-1,-1)$
3				$(1,1,-2)$

- Look at all possible signature changes $(\Delta_X, \Delta_2, \Delta_3)$
- Cost change **only** depends on signature change

	0	1	2	3
0	0	0	0	0
1		½	0	½
2			0	0
3				½

$a_2 \geq a_3$

	0	1	2	3
0	0	0	0	0
1		⅓	⅓	⅓
2			⅓	0
3				0

$a_2 < a_3$

Merges

Cycle sizes modulo 4

	0	1	2	3
0	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$
1		$(0,1,0)$	$(0,-1,1)$	$(1,0,-1)$
2			$(1,-2,0)$	$(1,-1,-1)$
3				$(1,1,-2)$

- Look at all possible signature changes $(\Delta_X, \Delta_2, \Delta_3)$
- Cost change **only** depends on signature change

	0	1	2	3
0	0	0	0	0
1		½	0	½
2			0	0
3				½

$a_2 \geq a_3$

	0	1	2	3
0	0	0	0	0
1		⅓	⅓	⅓
2			⅓	0
3				0

$a_2 < a_3$

Merges

Cycle sizes modulo 4

	0	1	2	3
0	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$
1		$(0,1,0)$	$(0,-1,1)$	$(1,0,-1)$
2			$(1,-2,0)$	$(1,-1,-1)$
3				$(1,1,-2)$

- Look at all possible signature changes $(\Delta_X, \Delta_2, \Delta_3)$
- Cost change **only** depends on signature change

	0	1	2	3
0	0	0	0	0
1		½	0	½
2			0	0
3				½

$a_2 \geq a_3$

	0	1	2	3
0	0	0	0	0
1		⅓	⅓	⅓
2			⅓	0
3				0

$a_2 < a_3$

$\Rightarrow$ **Merges are never worthwhile!**

Splits

	0	1	2	3
0	0	0	0	0
1		½	0	½
2			0	0
3				½

$a_2 \geq a_3$

	0	1	2	3
0	0	0	0	0
1		⅓	⅓	⅓
2			⅓	0
3				0

$a_2 < a_3$

Splits

	0	1	2	3
0	0	0	0	0
1		$-\frac{1}{2}$	0	$-\frac{1}{2}$
2			0	0
3				$-\frac{1}{2}$

$a_2 \geq a_3$

	0	1	2	3
0	0	0	0	0
1		$-\frac{1}{3}$	$-\frac{1}{3}$	$-\frac{1}{3}$
2			$-\frac{1}{3}$	0
3				0

$a_2 < a_3$

Splits

	0	1	2	3
0	0	0	0	0
1		$-\frac{1}{2}$	0	$-\frac{1}{2}$
2			0	0
3				$-\frac{1}{2}$

$a_2 \geq a_3$

	0	1	2	3
0	0	0	0	0
1		$-\frac{1}{3}$	$-\frac{1}{3}$	$-\frac{1}{3}$
2			$-\frac{1}{3}$	0
3				0

$a_2 < a_3$

- `permi5` can implement 4 transpositions $\Rightarrow -\frac{1}{2} \cdot 4 = -2$?

Splits

	0	1	2	3
0	0	0	0	0
1		$-½$	0	$-½$
2			0	0
3				$-½$

$a_2 \geq a_3$

	0	1	2	3
0	0	0	0	0
1		$-⅓$	$-⅓$	$-⅓$
2			$-⅓$	0
3				0

$a_2 < a_3$

- `permi5` can implement 4 transpositions $\Rightarrow -½ \cdot 4 = -2$?
- However: because of structure of instructions, not every transposition can reduce costs

Splits

	0	1	2	3
0	0	0	0	0
1		$-½$	0	$-½$
2			0	0
3				$-½$

$a_2 \geq a_3$

	0	1	2	3
0	0	0	0	0
1		$-⅓$	$-⅓$	$-⅓$
2			$-⅓$	0
3				0

$a_2 < a_3$

- `permi5` can implement 4 transpositions $\Rightarrow -½ \cdot 4 = -2$?
- However: because of structure of instructions, not every transposition can reduce costs

GREEDY is optimal for PRTGs

GREEDY computes an optimal shuffle code for PRTGs in linear time.

Finding optimal copy sets

Given G, have to find copy set C, s.t. $\textsc{Greedy}(G - C)$ is minimal

Finding optimal copy sets

Given G, have to find copy set C, s.t. $\textsc{Greedy}(G - C)$ is minimal

- Equivalent to minimizing

$$\textsc{Greedy}'(G - C) = \begin{cases} X + \frac{a_2}{2} + \frac{a_3}{2} & \text{if } a_2 \geq a_3 \\ X + \frac{a_2}{3} + \frac{2a_3}{3} & \text{if } a_2 < a_3 \end{cases}$$

- Distinguish cases with $\text{diff}(G - C) = a_2 - a_3$

Finding optimal copy sets

Given G, have to find copy set C, s.t. $\textsc{Greedy}(G - C)$ is minimal

- Equivalent to minimizing

$$\textsc{Greedy}'(G - C) = \begin{cases} X + \frac{a_2}{2} + \frac{a_3}{2} =: cost^1(G - C) & \text{if } a_2 \geq a_3 \\ X + \frac{a_2}{3} + \frac{2a_3}{3} =: cost^2(G - C) & \text{if } a_2 < a_3 \end{cases}$$

- Distinguish cases with $\text{diff}(G - C) = a_2 - a_3$

Finding optimal copy sets

Given G, have to find copy set C, s.t. $\textsc{Greedy}(G - C)$ is minimal

- Equivalent to minimizing

$$\textsc{Greedy}'(G - C) = \begin{cases} X + \frac{a_2}{2} + \frac{a_3}{2} =: cost^1(G - C) & \text{if } a_2 \geq a_3 \\ X + \frac{a_2}{3} + \frac{2a_3}{3} =: cost^2(G - C) & \text{if } a_2 < a_3 \end{cases}$$

- Distinguish cases with $\text{diff}(G - C) = a_2 - a_3$

Dynamic program with tables $T^1[\cdot]$, $T^2[\cdot]$

- $T^i[d] = \min\{\text{cost}^i(G - C) \mid C \text{ copyset with } \text{diff}(G - C) = d\}$

Finding optimal copy sets

Given G, have to find copy set C, s.t. $\textsc{Greedy}(G - C)$ is minimal

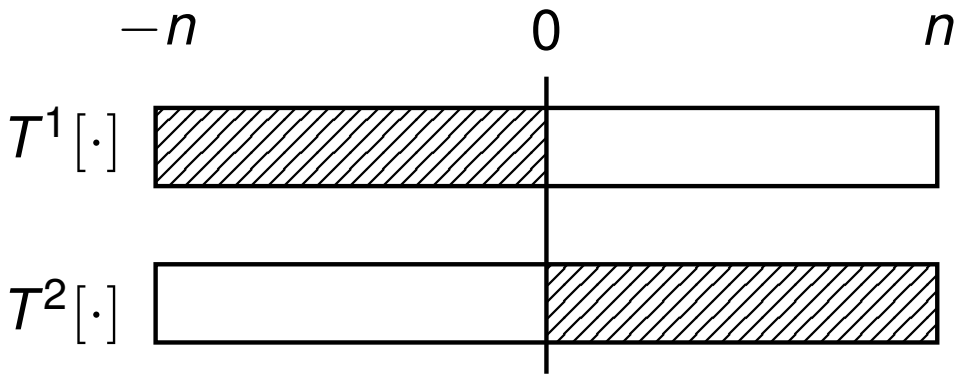
- Equivalent to minimizing

$$\textsc{Greedy}'(G-C) = \begin{cases} X + \frac{a_2}{2} + \frac{a_3}{2} =: cost^1(G-C) & \text{if } a_2 \geq a_3 \\ X + \frac{a_2}{3} + \frac{2a_3}{3} =: cost^2(G-C) & \text{if } a_2 < a_3 \end{cases}$$

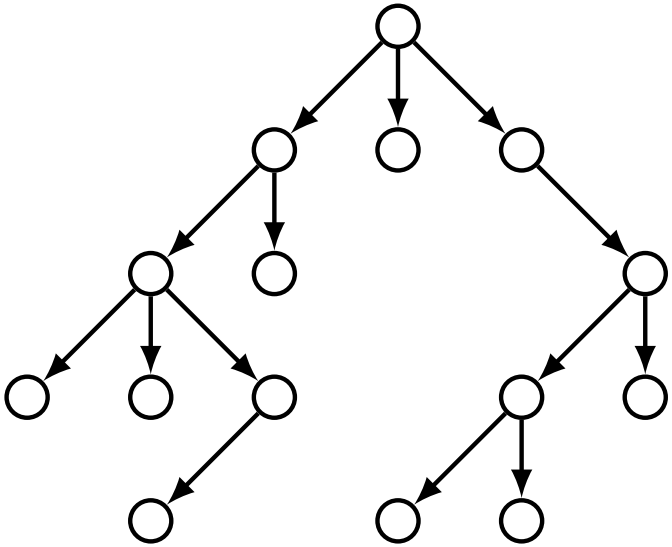
- Distinguish cases with $\text{diff}(G-C) = a_2 - a_3$

Dynamic program with tables $T^1[\cdot]$, $T^2[\cdot]$

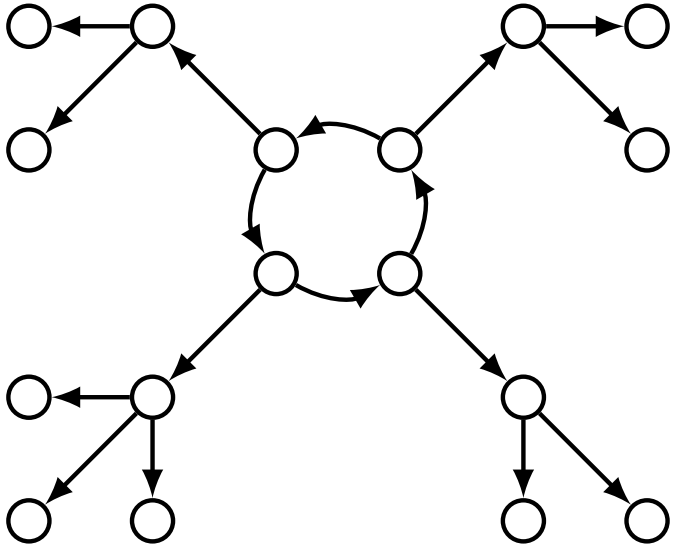
- $T^i[d] = \min\{\text{cost}^i(G-C) \mid C \text{ copyset with } \text{diff}(G-C) = d\}$



Computing Tables for Different RTG Shapes

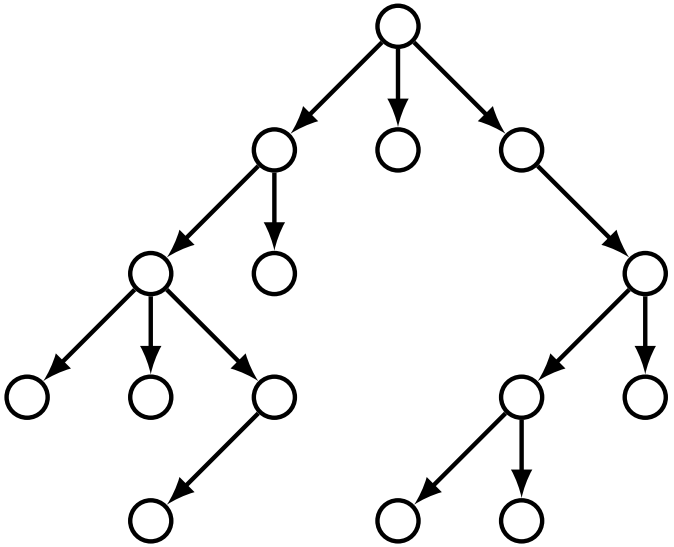


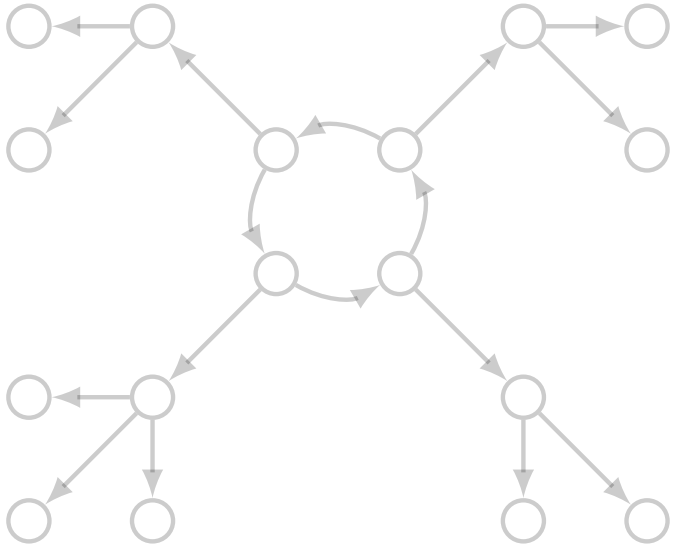
Disconnected RTGs



$O(n^2)$

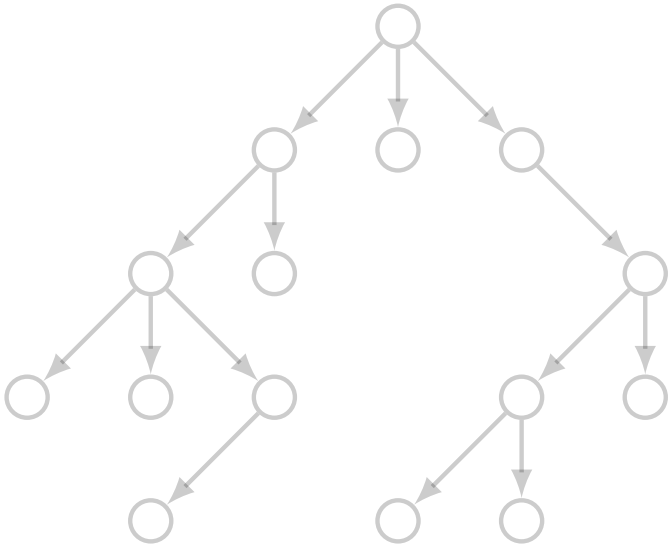
Computing Tables for Different RTG Shapes

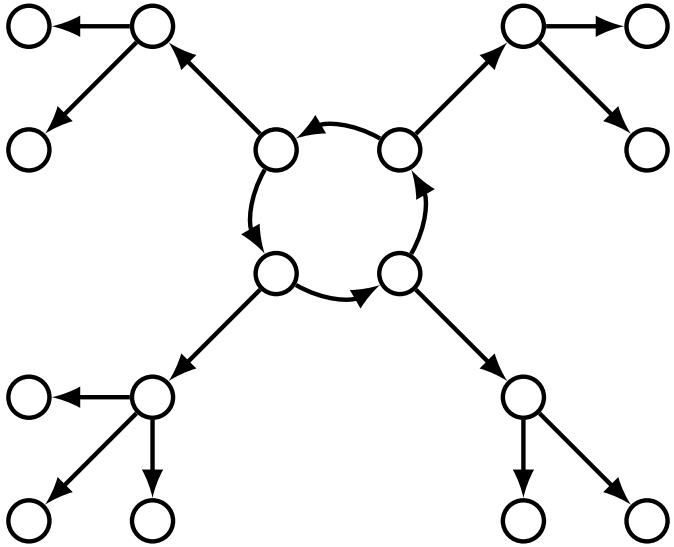




Disconnected RTGs	$O(n^2)$
Tree RTGs	$O(n^3)$

Computing Tables for Different RTG Shapes





Disconnected RTGs	$O(n^2)$
Tree RTGs	$O(n^3)$
Connected RTGs containing cycle	$O(n^4)$

Conclusion

Problem 1: Given copy set C, compute optimal shuffle code for $G - C$.

- Shown that GREEDY is optimal for PRTGs, runs in $O(n)$ time
- Equivalent to factoring permutation into shortest product of permutations of maximum size 5

Conclusion

Problem 1: Given copy set C, compute optimal shuffle code for $G - C$.

- Shown that GREEDY is optimal for PRTGs, runs in $O(n)$ time
- Equivalent to factoring permutation into shortest product of permutations of maximum size 5

Problem 2: Find copy set C s.t. $G - C$ requires fewest instructions.

- Used dynamic programming to solve optimally
- Runs in $O(n^4)$ time

Conclusion

Problem 1: Given copy set C, compute optimal shuffle code for $G - C$.

- Shown that GREEDY is optimal for PRTGs, runs in $O(n)$ time
- Equivalent to factoring permutation into shortest product of permutations of maximum size 5

Problem 2: Find copy set C s.t. $G - C$ requires fewest instructions.

- Used dynamic programming to solve optimally
- Runs in $O(n^4)$ time

Future work:

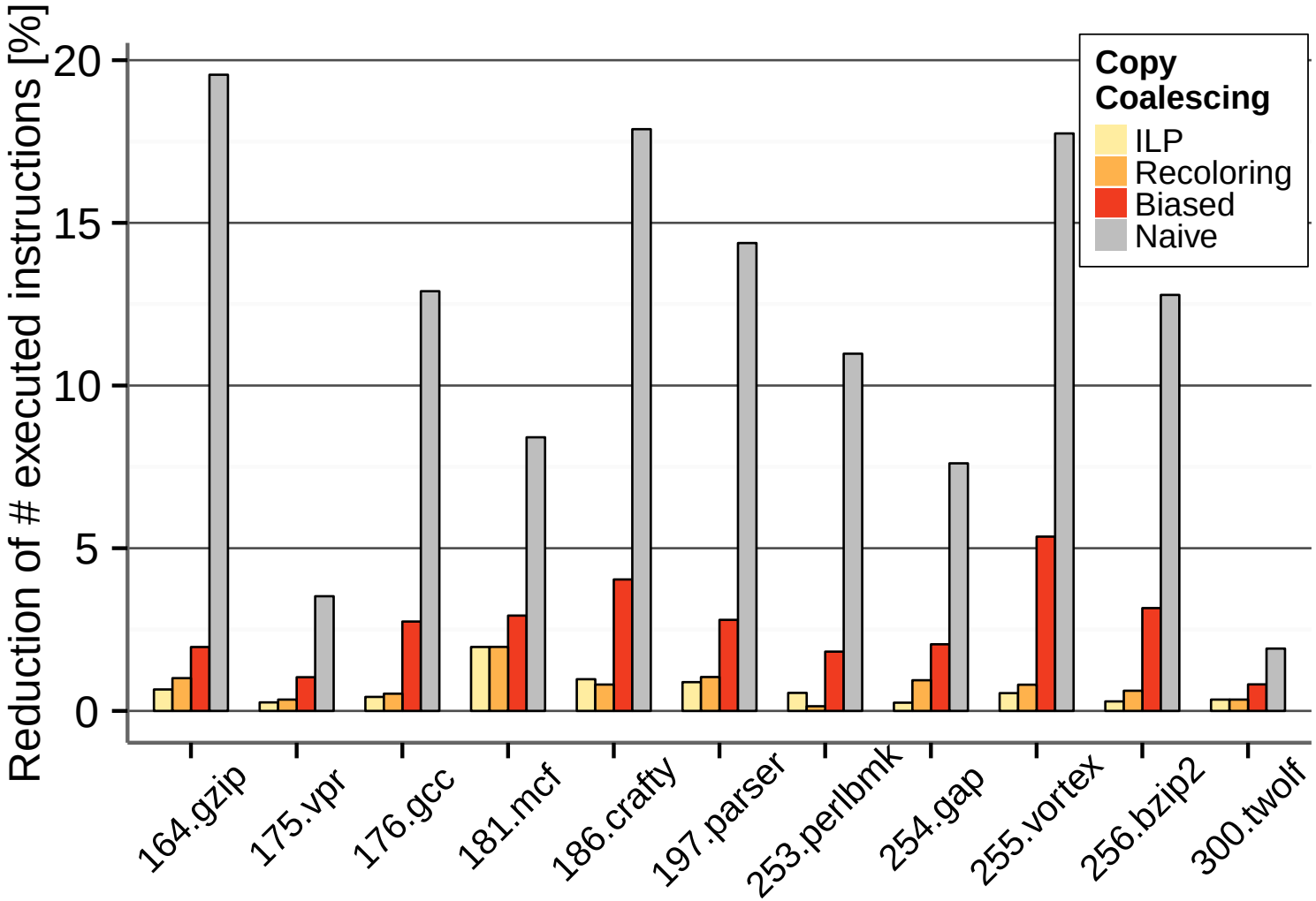
- Works for $k = 3, 4, 5, 6$, what about larger sizes?
- Problem NP-complete if permutation size is part of input

$\Rightarrow$ FPT-algorithm?

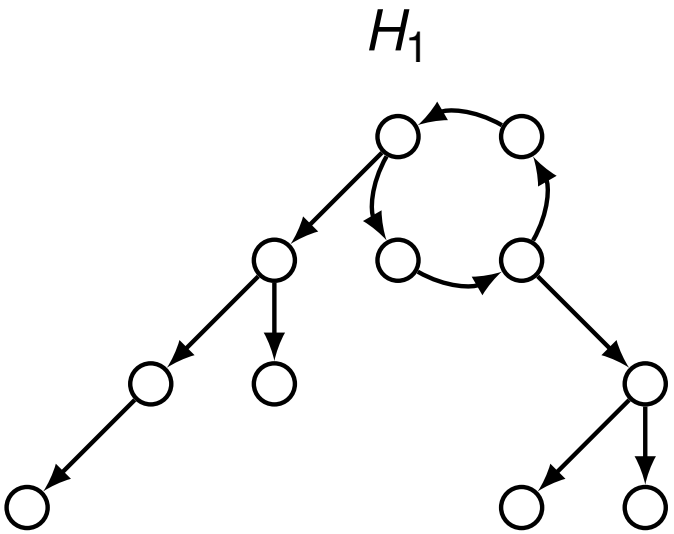
Karlsruhe Institute of Technology

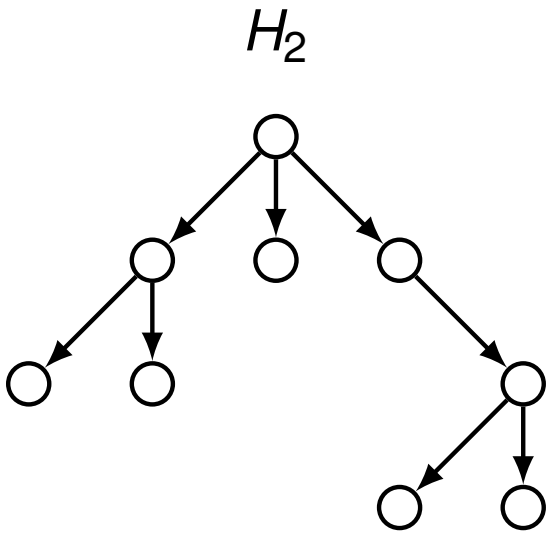
Backup Slides

Speedup Measurements

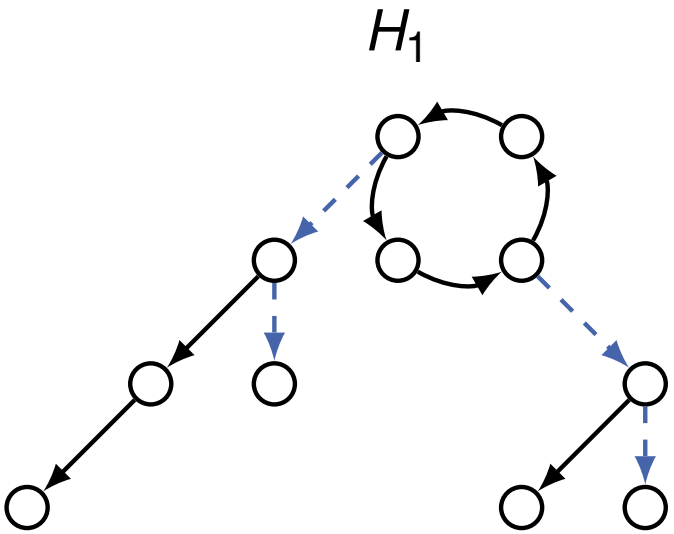


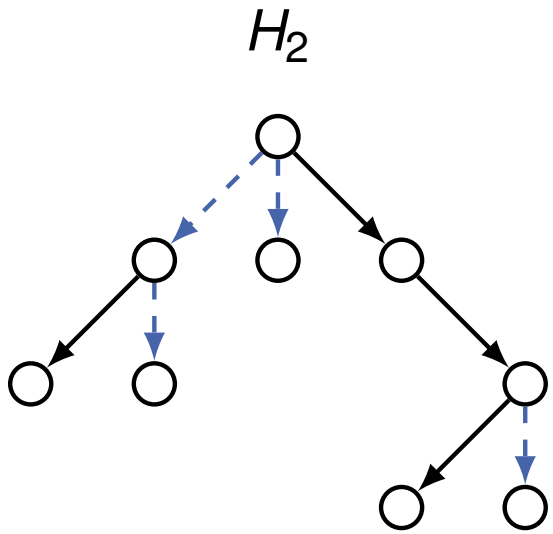
Disconnected RTGs



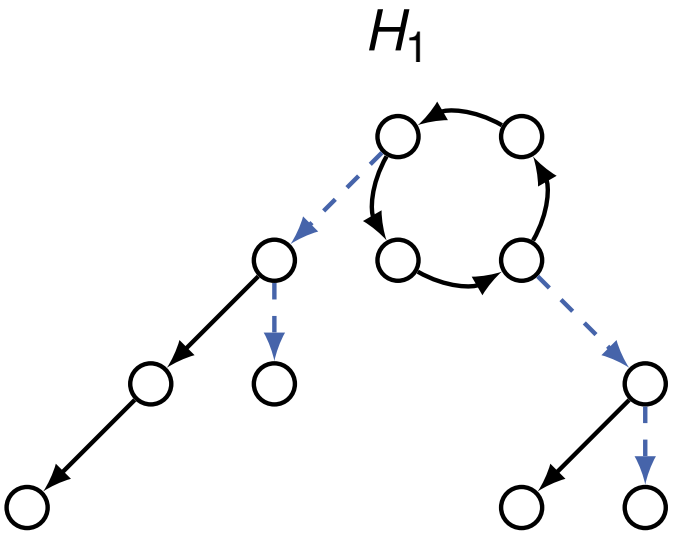


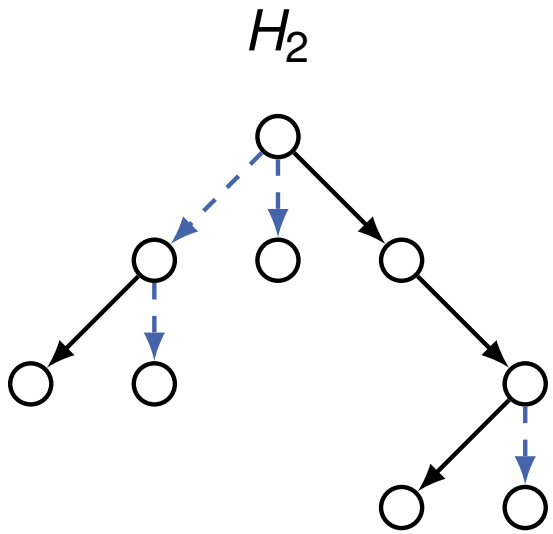
Disconnected RTGs





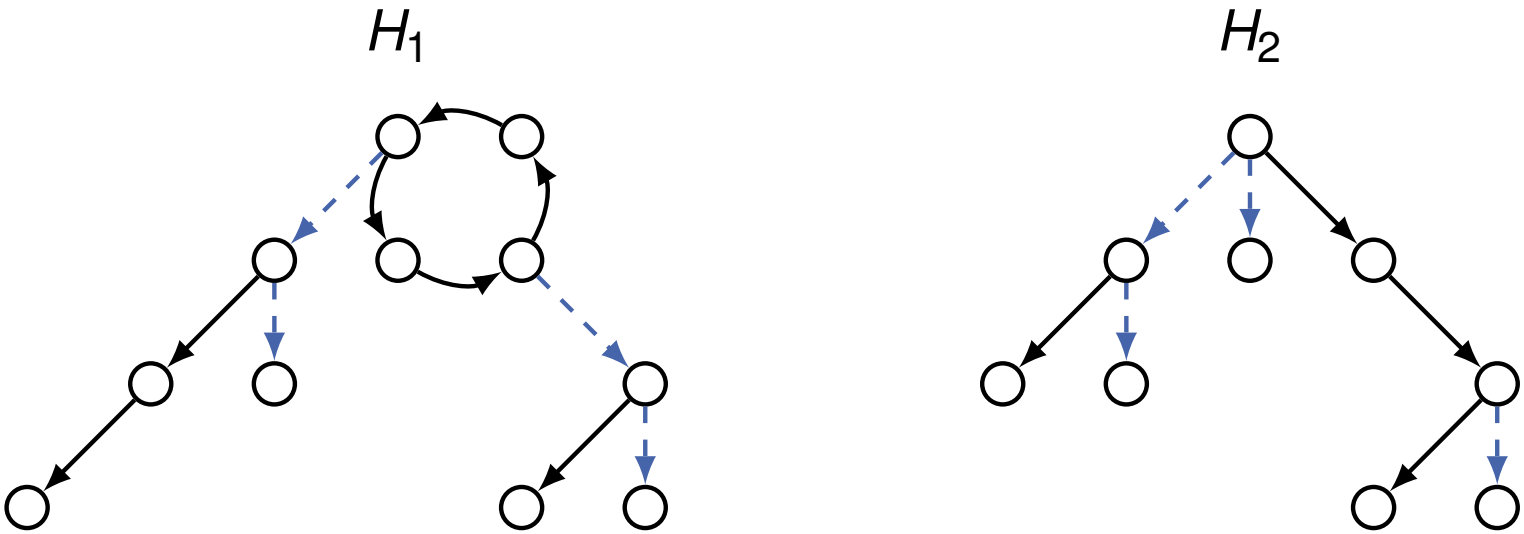
Disconnected RTGs





- $C := C_1 \cup C_2$
- $\text{cost}(G - C) = \text{cost}(H_1 - C_1) + \text{cost}(H_2 - C_2)$
- $\text{diff}(G - C) = \text{diff}(H_1 - C_1) + \text{diff}(H_2 - C_2)$

Disconnected RTGs

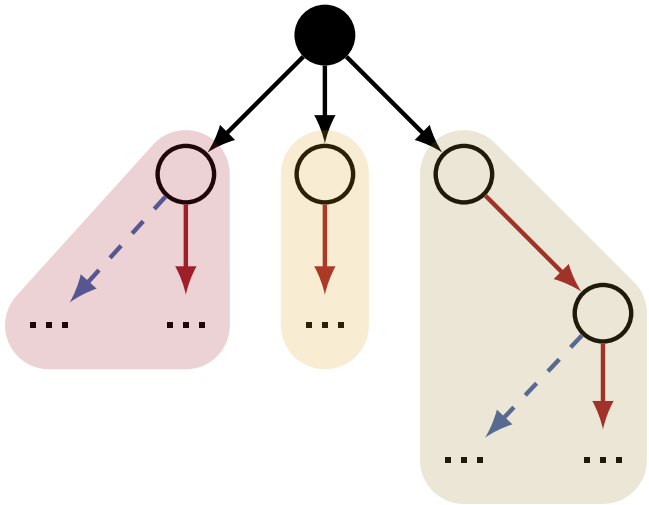


- $C := C_1 \cup C_2$
- $\text{cost}(G - C) = \text{cost}(H_1 - C_1) + \text{cost}(H_2 - C_2)$
- $\text{diff}(G - C) = \text{diff}(H_1 - C_1) + \text{diff}(H_2 - C_2)$

$\Rightarrow$ Can find optimal copy set C by trying all pairs C_1, C_2

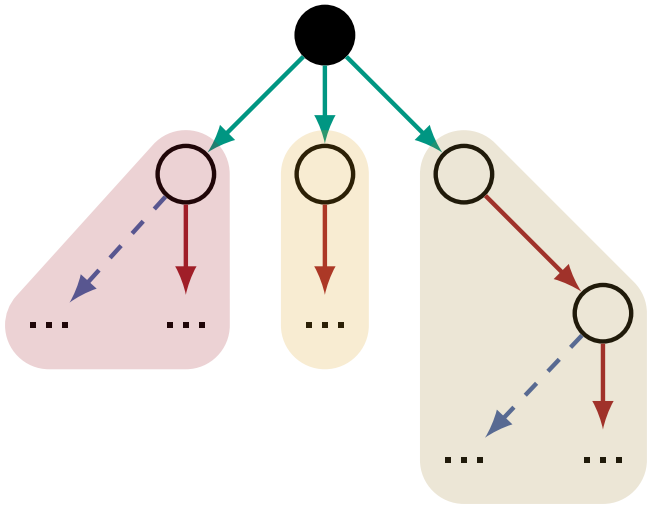
$\Rightarrow$ Can be computed in $O(n^2)$ time

Tree RTGs



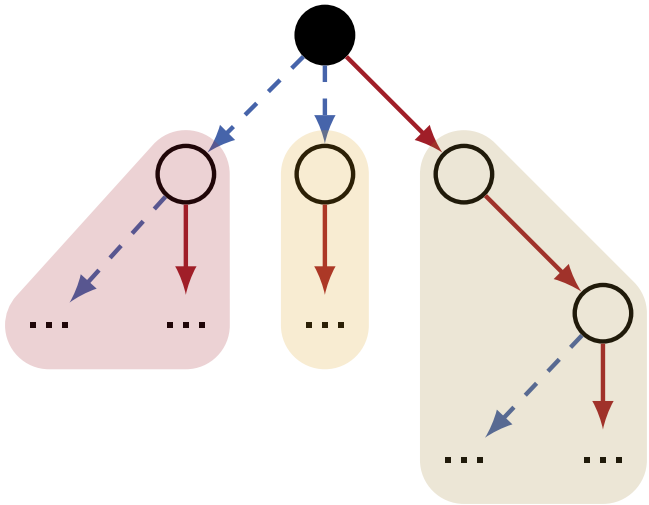
- Process tree RTGs in bottom-up fashion

Tree RTGs



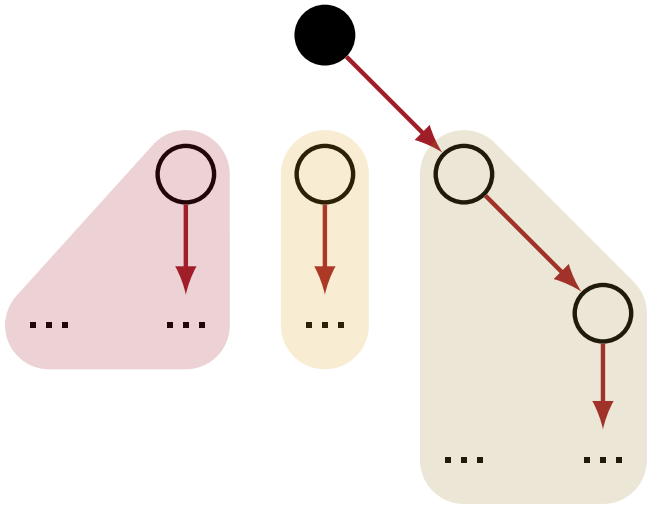
- Process tree RTGs in bottom-up fashion

Tree RTGs



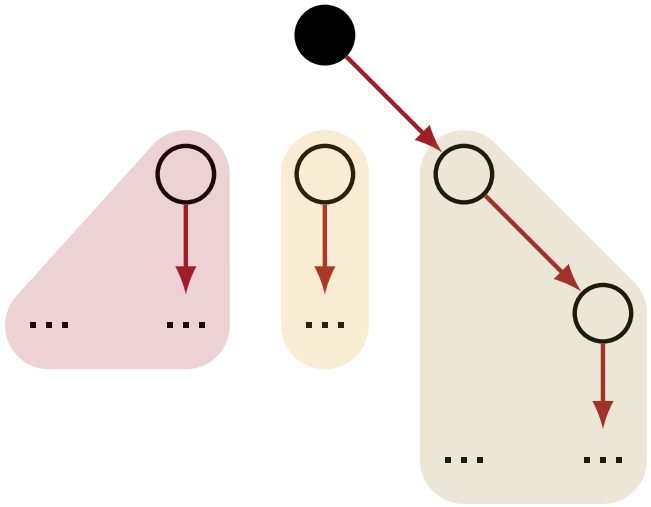
- Process tree RTGs in bottom-up fashion

Tree RTGs



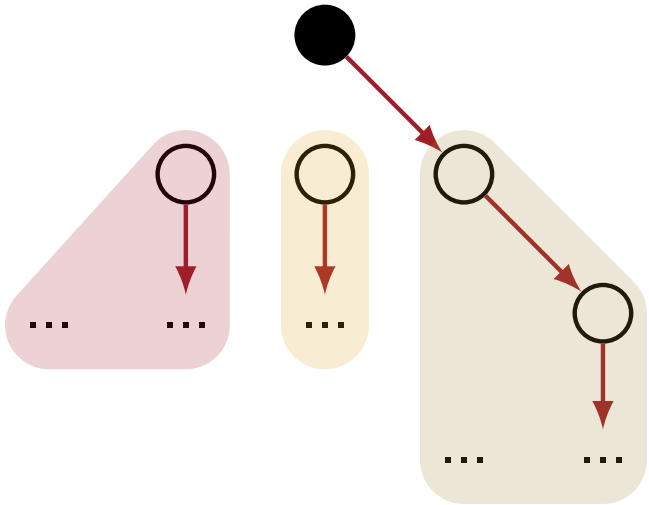
- Process tree RTGs in bottom-up fashion

Tree RTGs



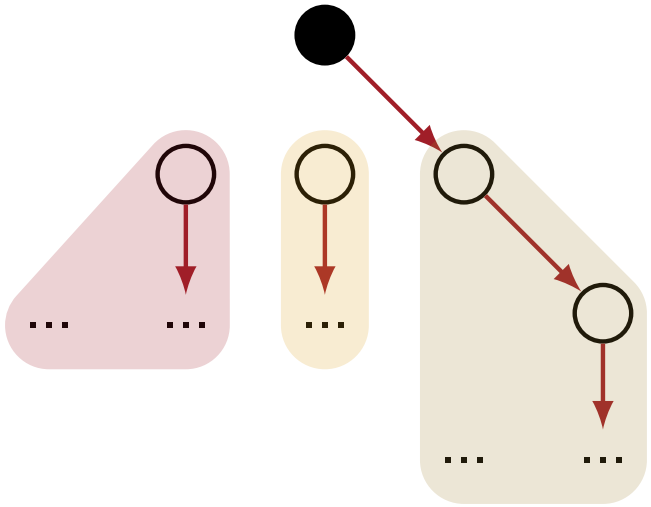
- Process tree RTGs in bottom-up fashion
- Extend table to track length of path starting at root

Tree RTGs



- Process tree RTGs in bottom-up fashion
- Extend table to track length of path (modulo 4) starting at root

Tree RTGs

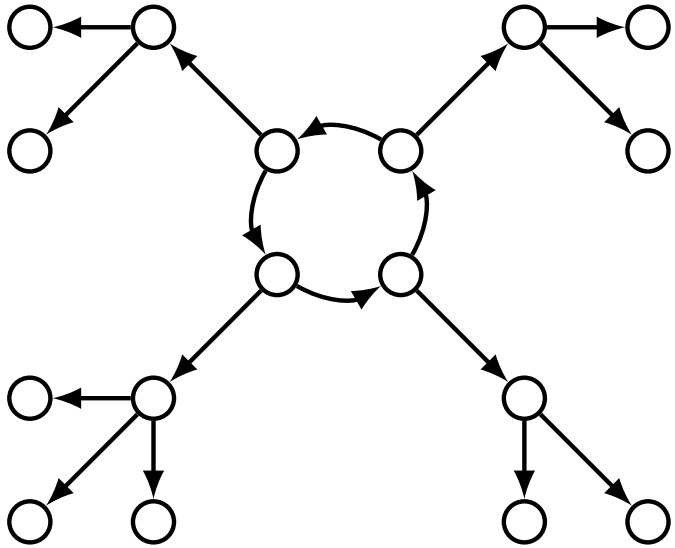


- Process tree RTGs in bottom-up fashion
- Extend table to track length of path (modulo 4) starting at root

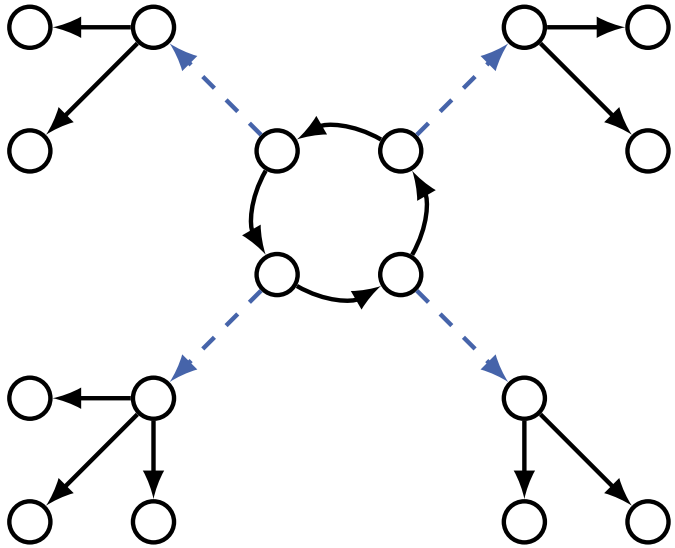
$\Rightarrow$ Can find optimal copy set C by trying all outgoing edges of root

$\Rightarrow$ Can be computed in time $O(n^3)$

Connected RTGs

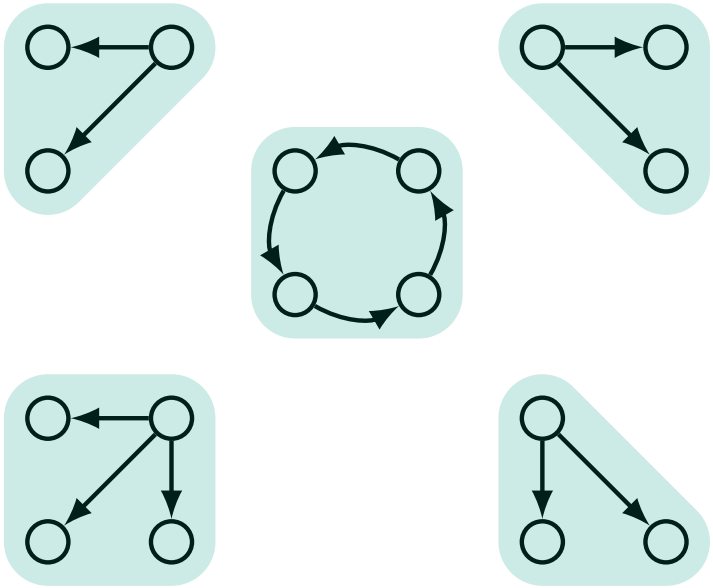


Connected RTGs



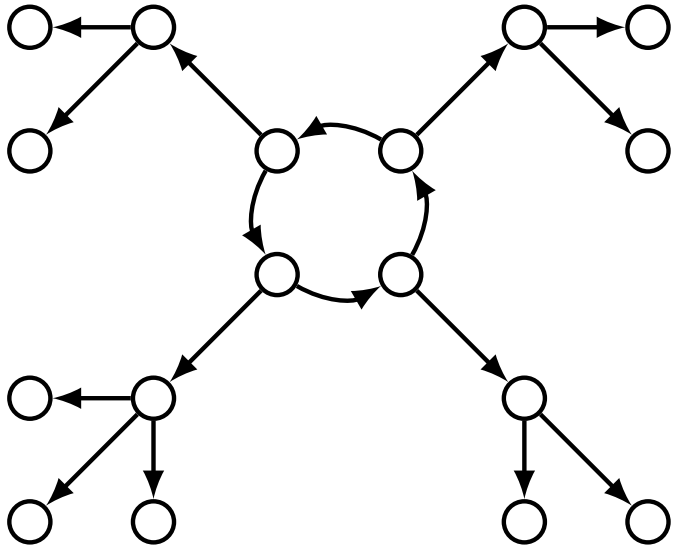
- Either keep cycle K and put edges leaving K into copy set

Connected RTGs



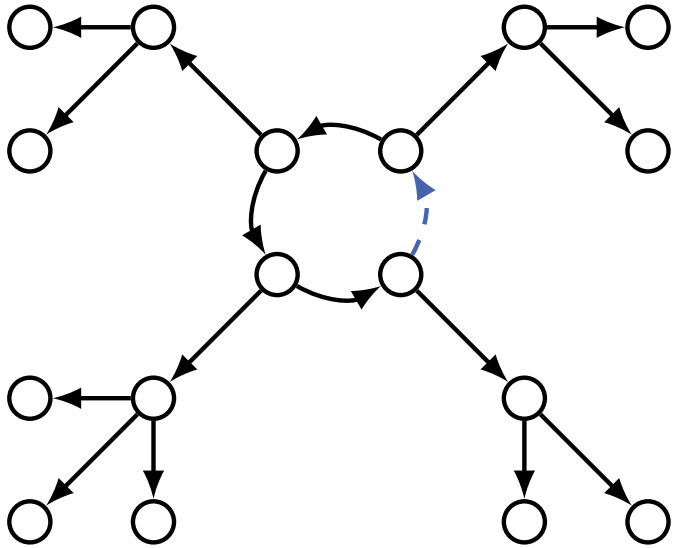
- Either keep cycle K and put edges leaving K into copy set

Connected RTGs



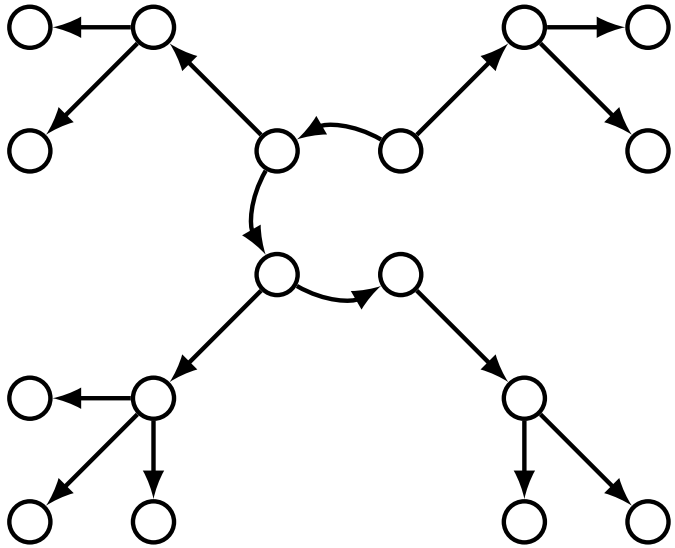
- Either keep cycle K and put edges leaving K into copy set

Connected RTGs



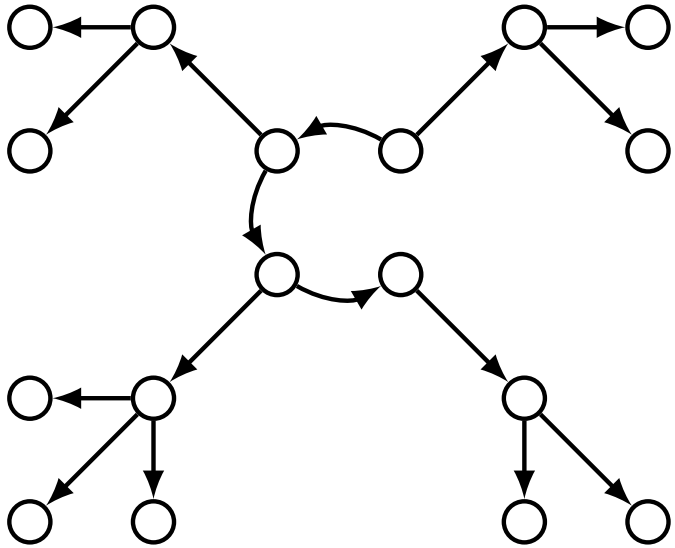
- Either keep cycle K and put edges leaving K into copy set
- Or cut K, $G - C$ is a tree

Connected RTGs



- Either keep cycle K and put edges leaving K into copy set
- Or cut K, $G - C$ is a tree

Connected RTGs

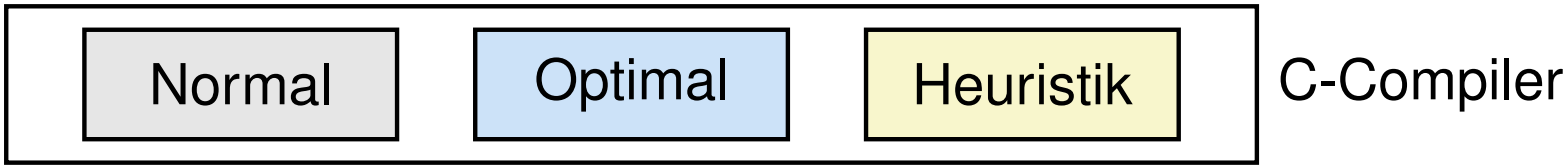


- Either keep cycle K and put edges leaving K into copy set
- Or cut K, $G - C$ is a tree

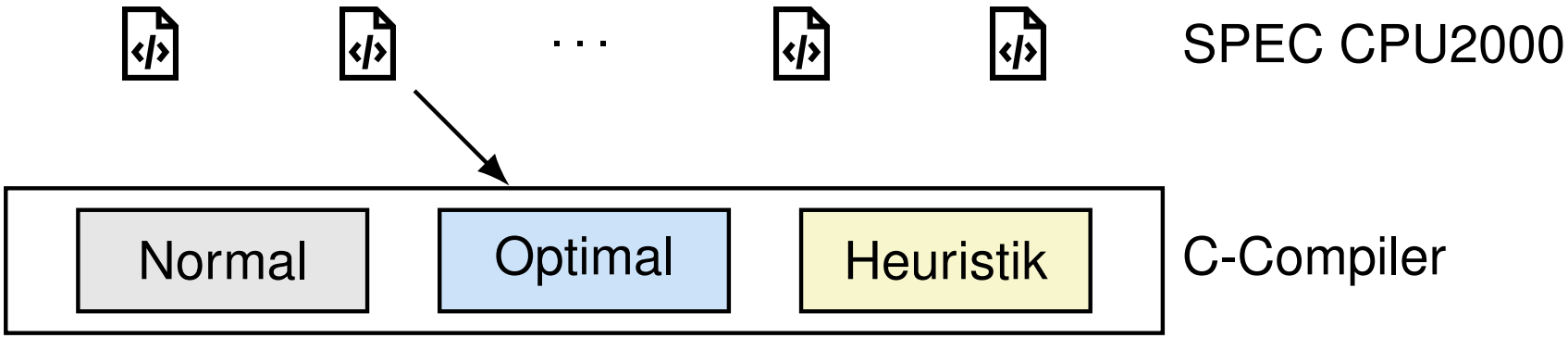
$\Rightarrow$ Can find optimal copy set C by trying all of K's edges (or keeping K)

$\Rightarrow$ Can be computed in time $O(n^4)$

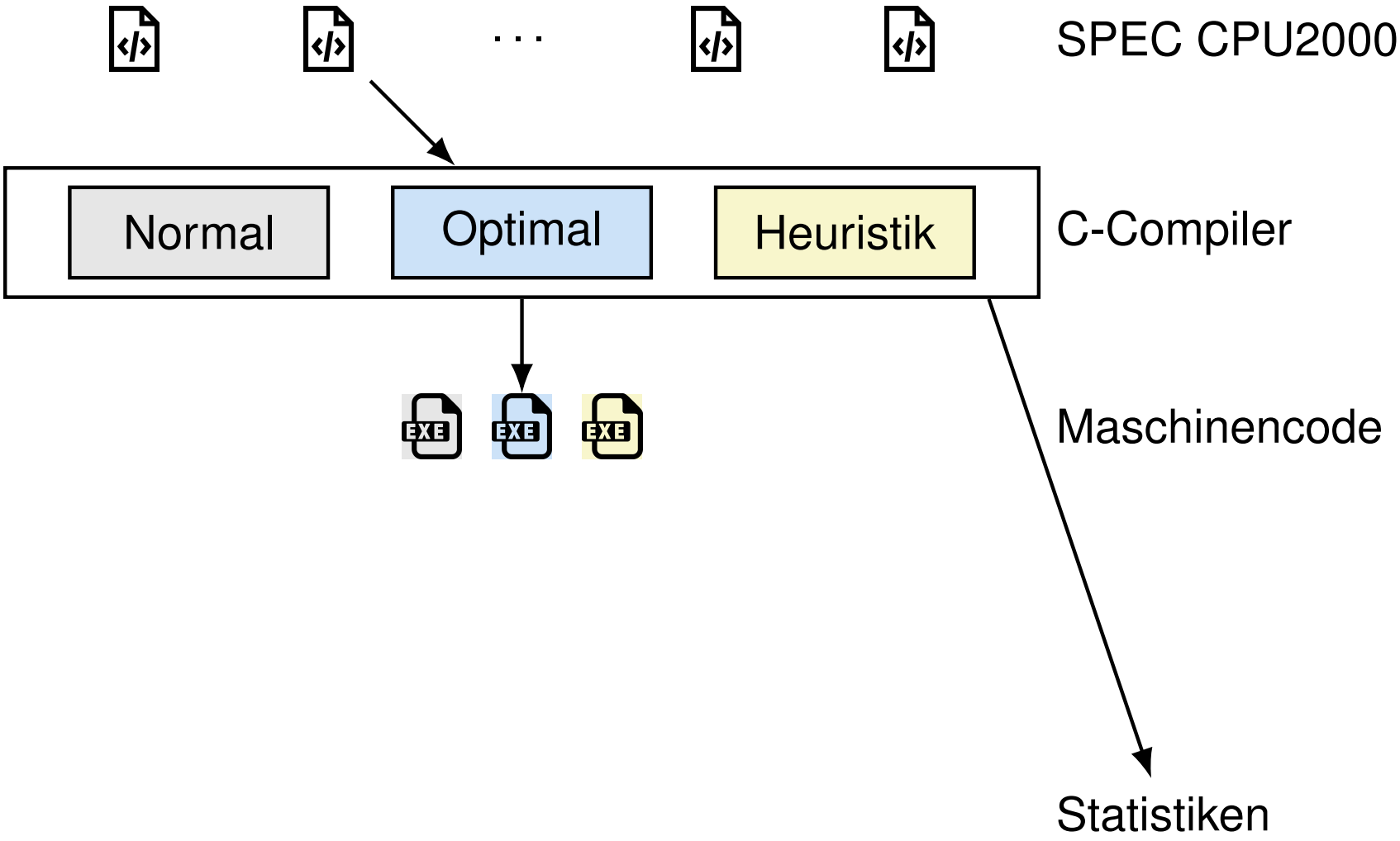
Evaluierungsstrategie



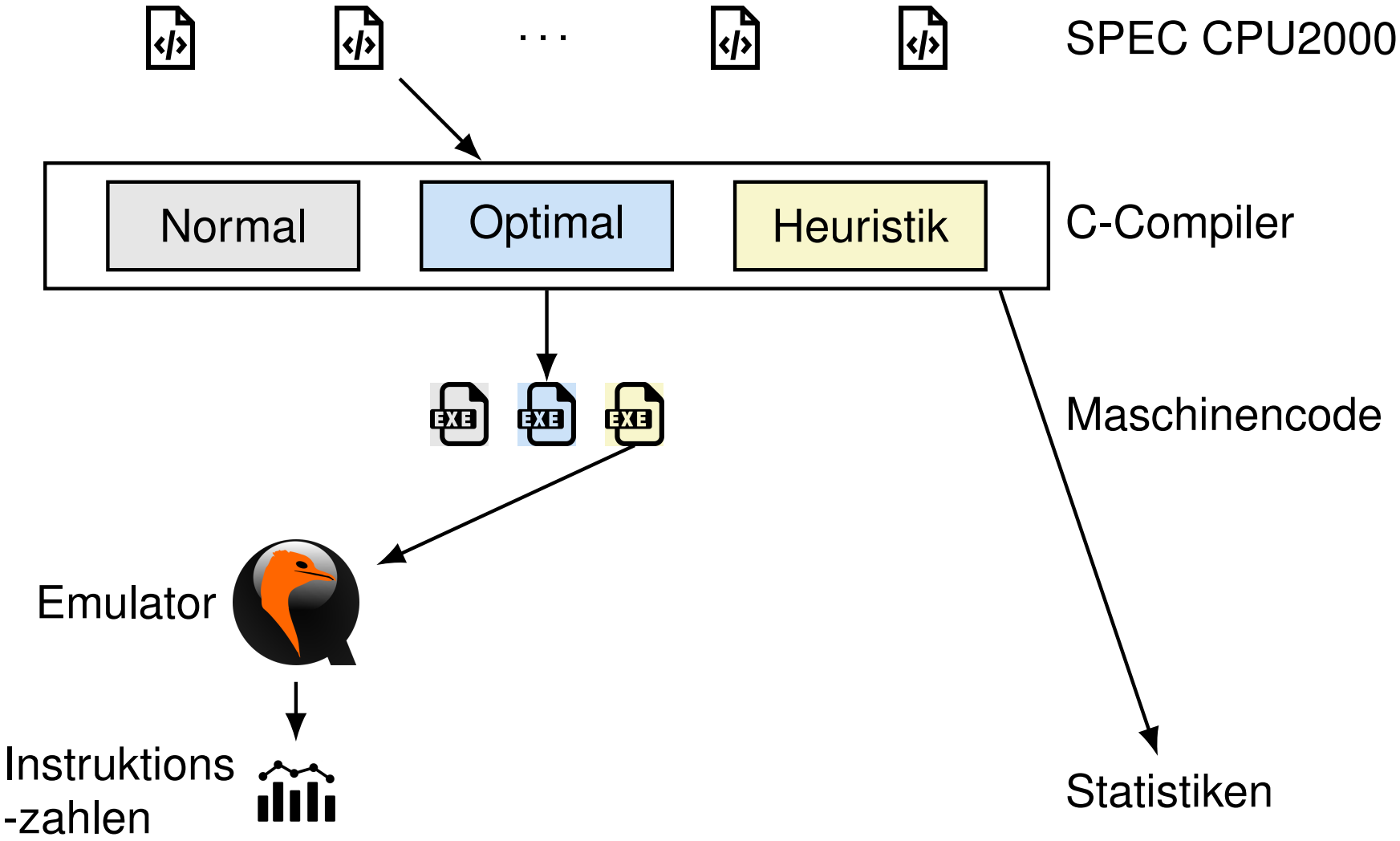
Evaluierungsstrategie



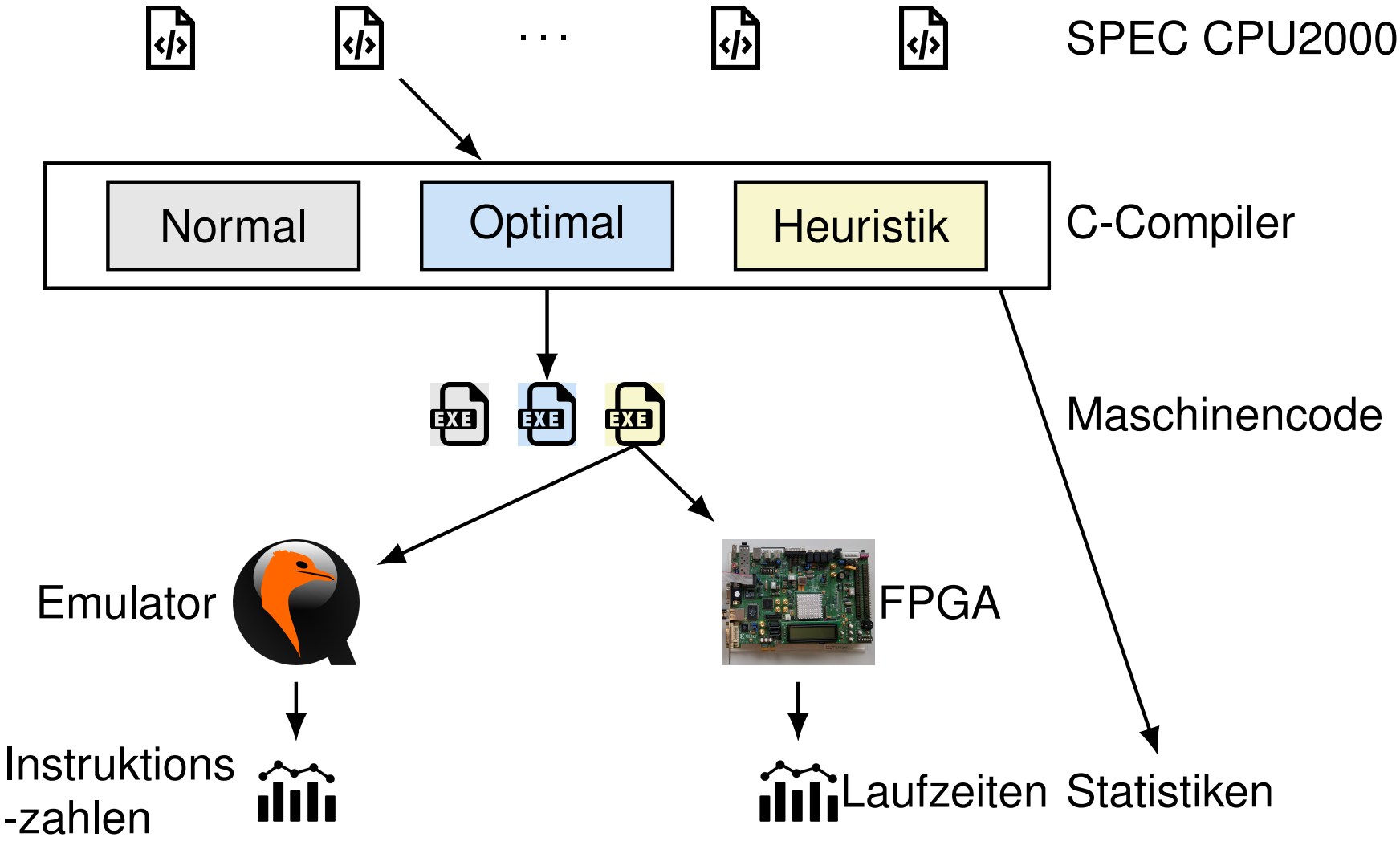
Evaluierungsstrategie



Evaluierungsstrategie



Evaluierungsstrategie



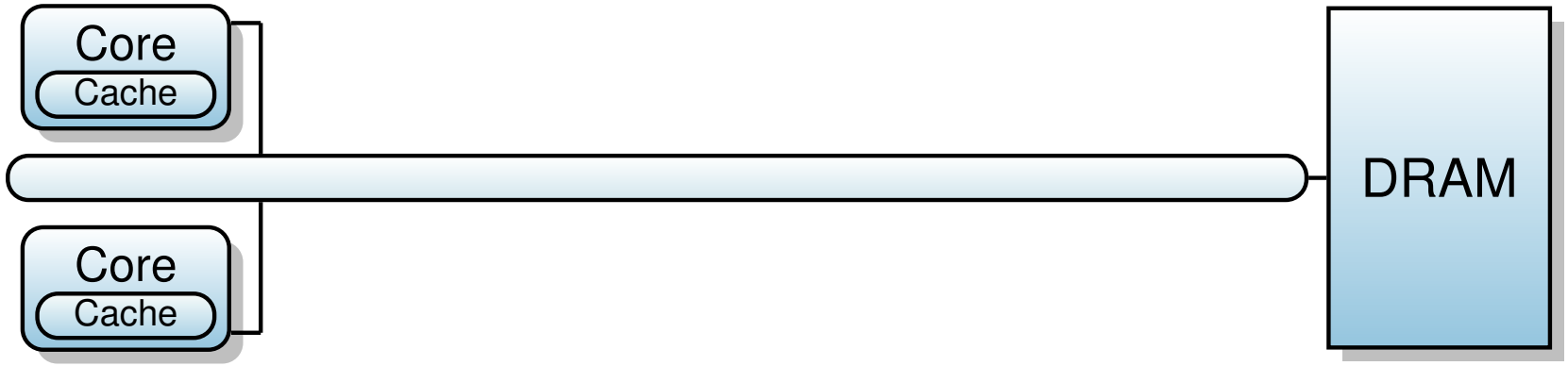
Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures

Manuel Mohr

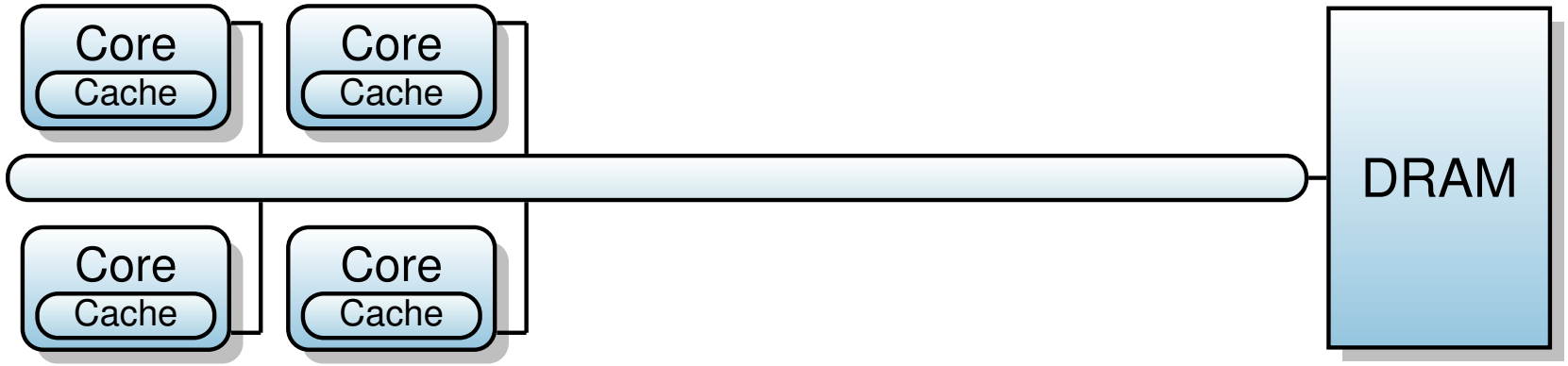
Lehrstuhl für Programmierparadigmen, Karlsruher Institut für Technologie (KIT)

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

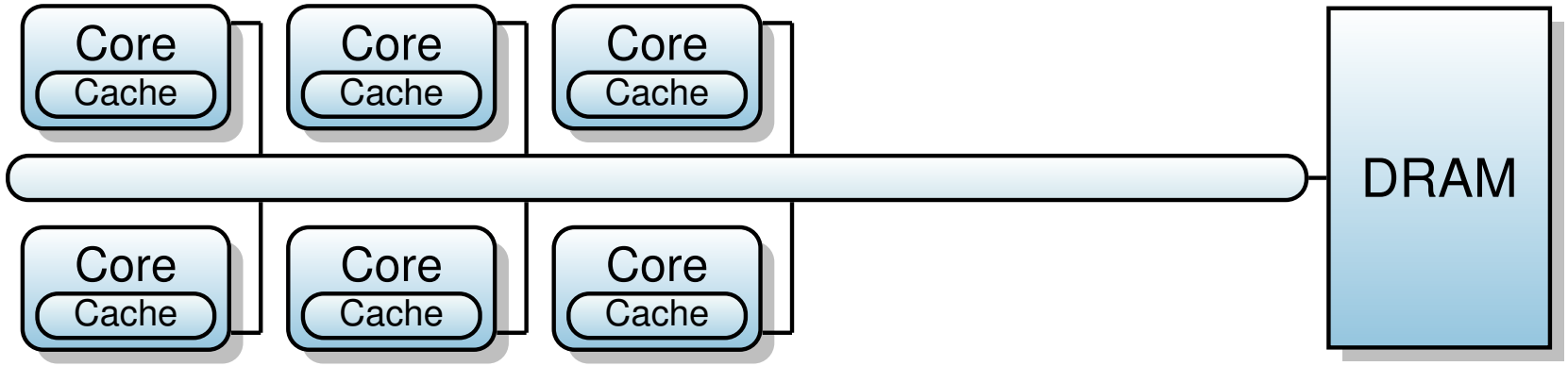
Non-Cache-Coherent Architectures



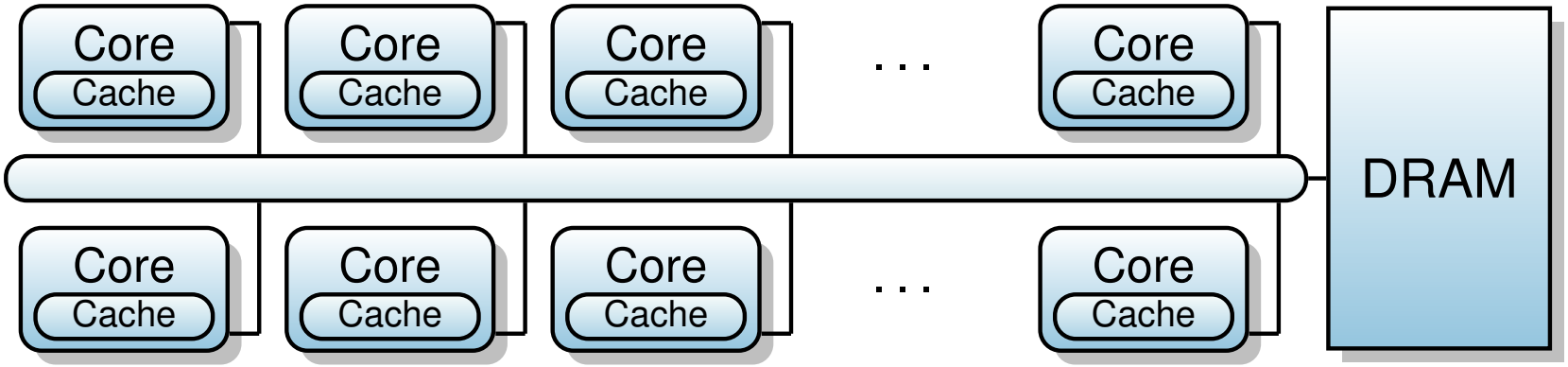
Non-Cache-Coherent Architectures



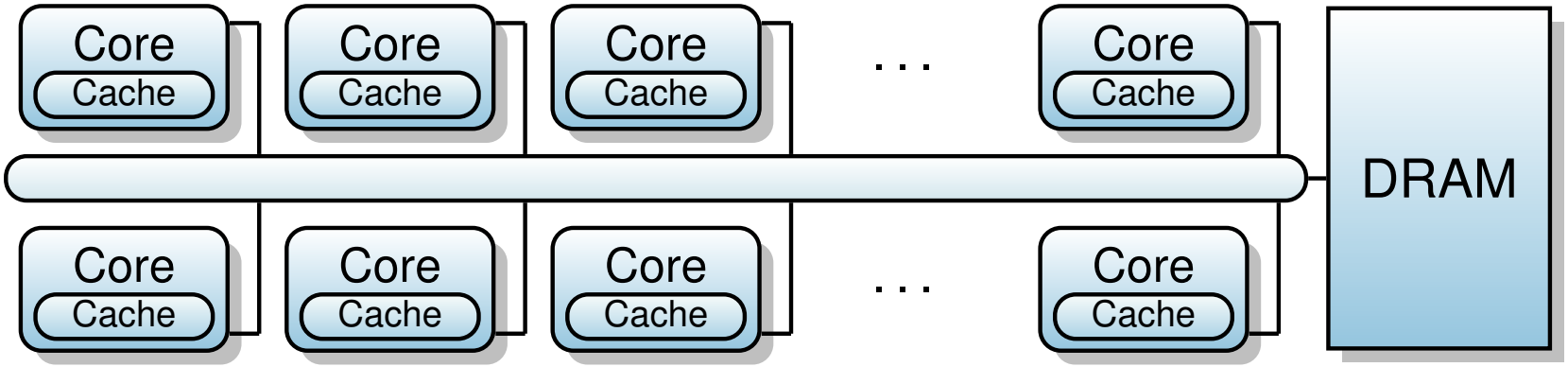
Non-Cache-Coherent Architectures



Non-Cache-Coherent Architectures

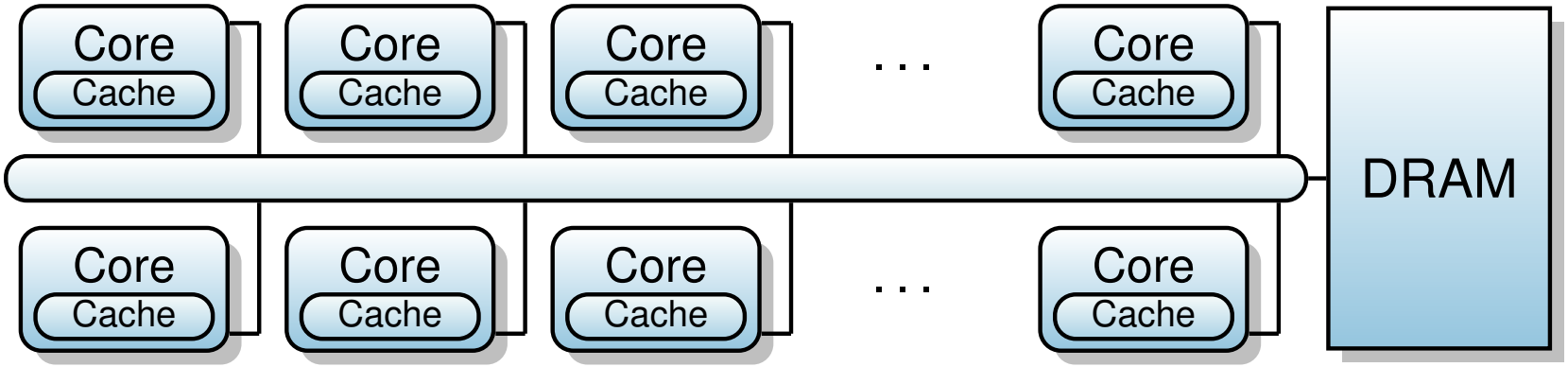


Non-Cache-Coherent Architectures



- Existing hardware coherence protocols hit scalability limit

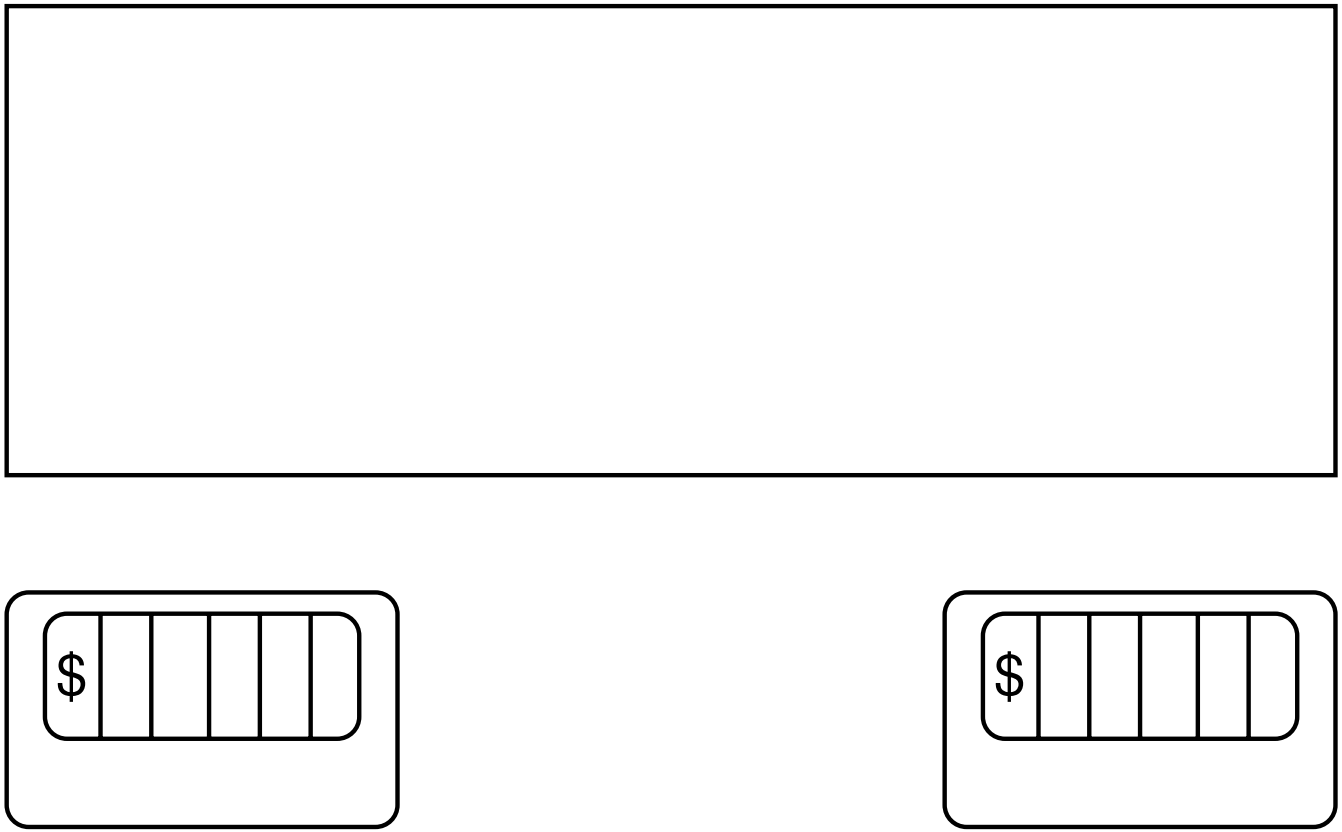
Non-Cache-Coherent Architectures



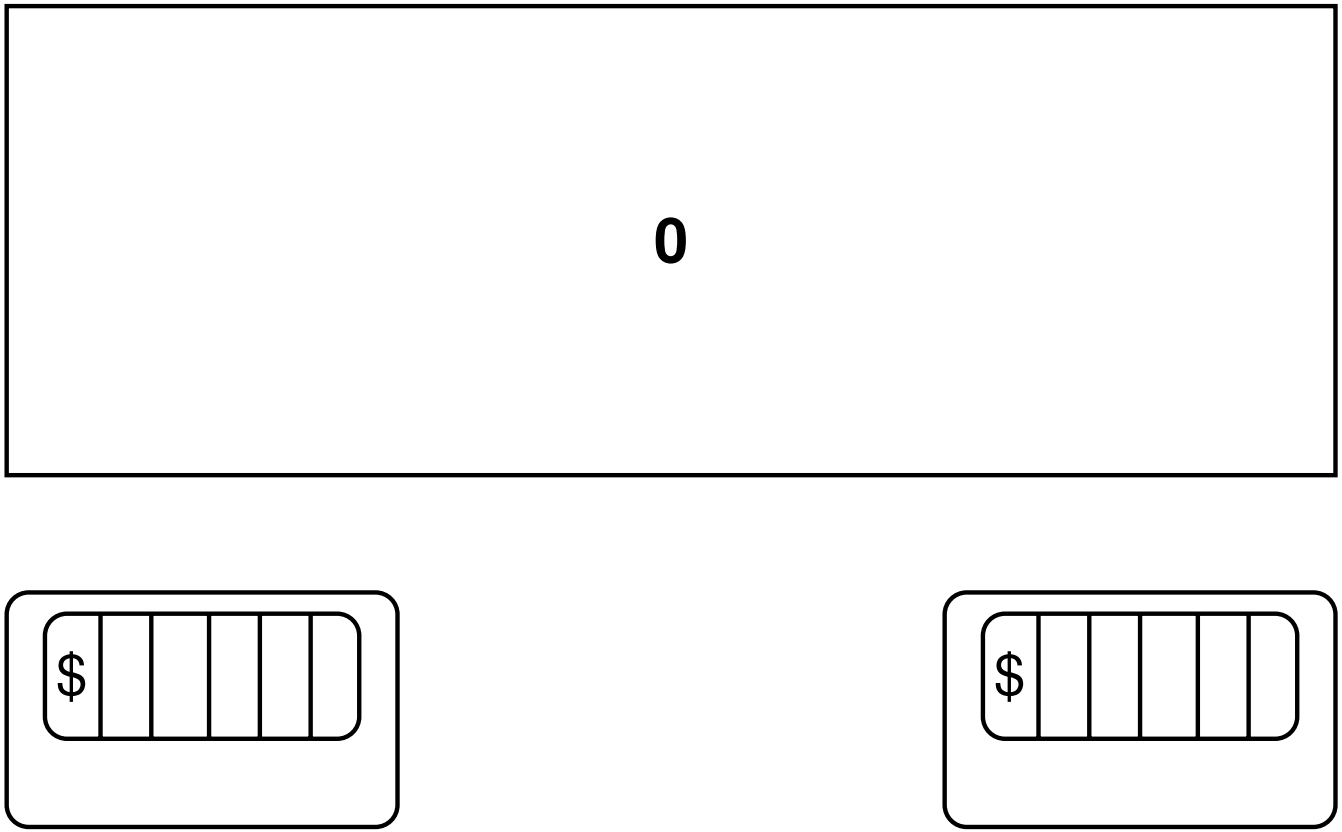
- Existing hardware coherence protocols hit scalability limit
- Radical answer: abandon global cache coherence
  - Still provide shared memory
  - Most prominent example: Intel SCC

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

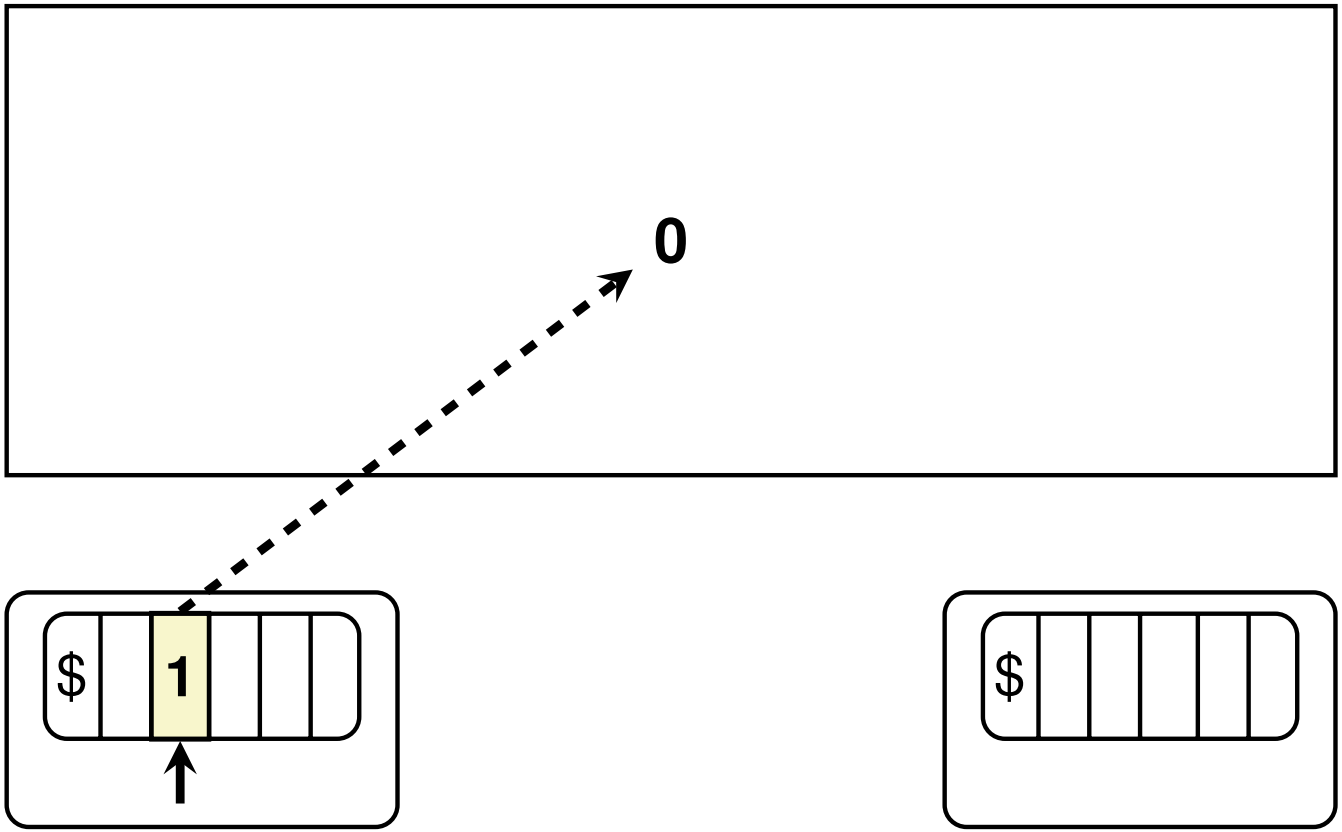
Shared-Memory Programming Model



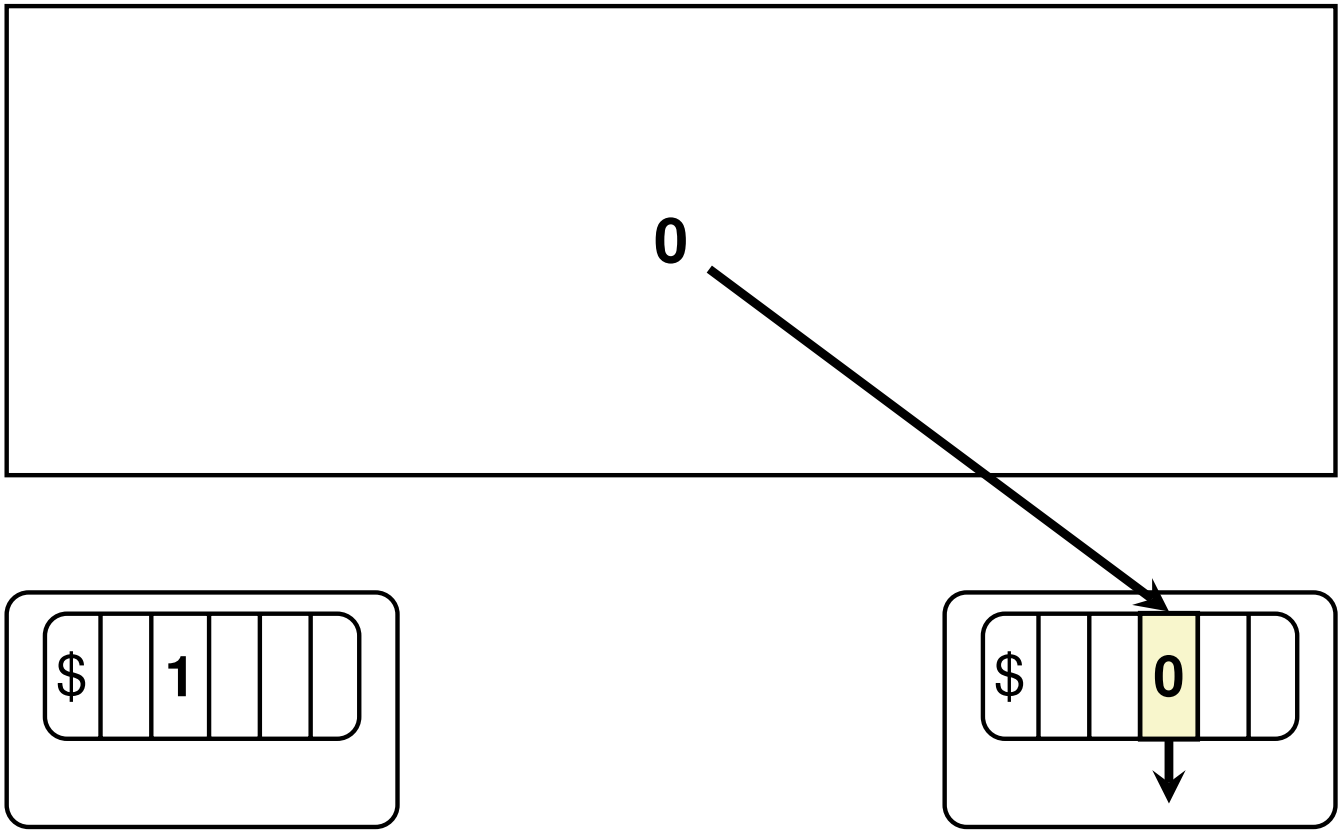
Shared-Memory Programming Model



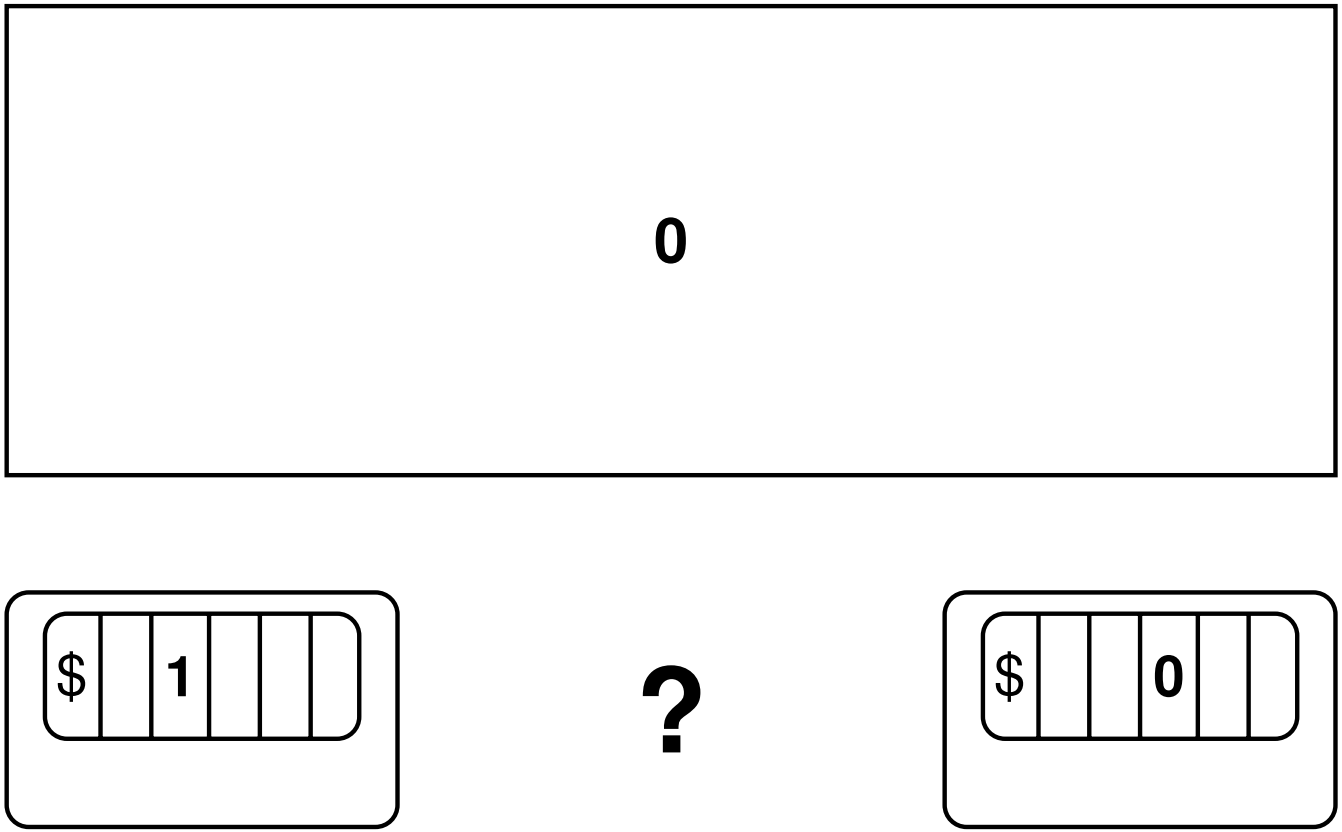
Shared-Memory Programming Model



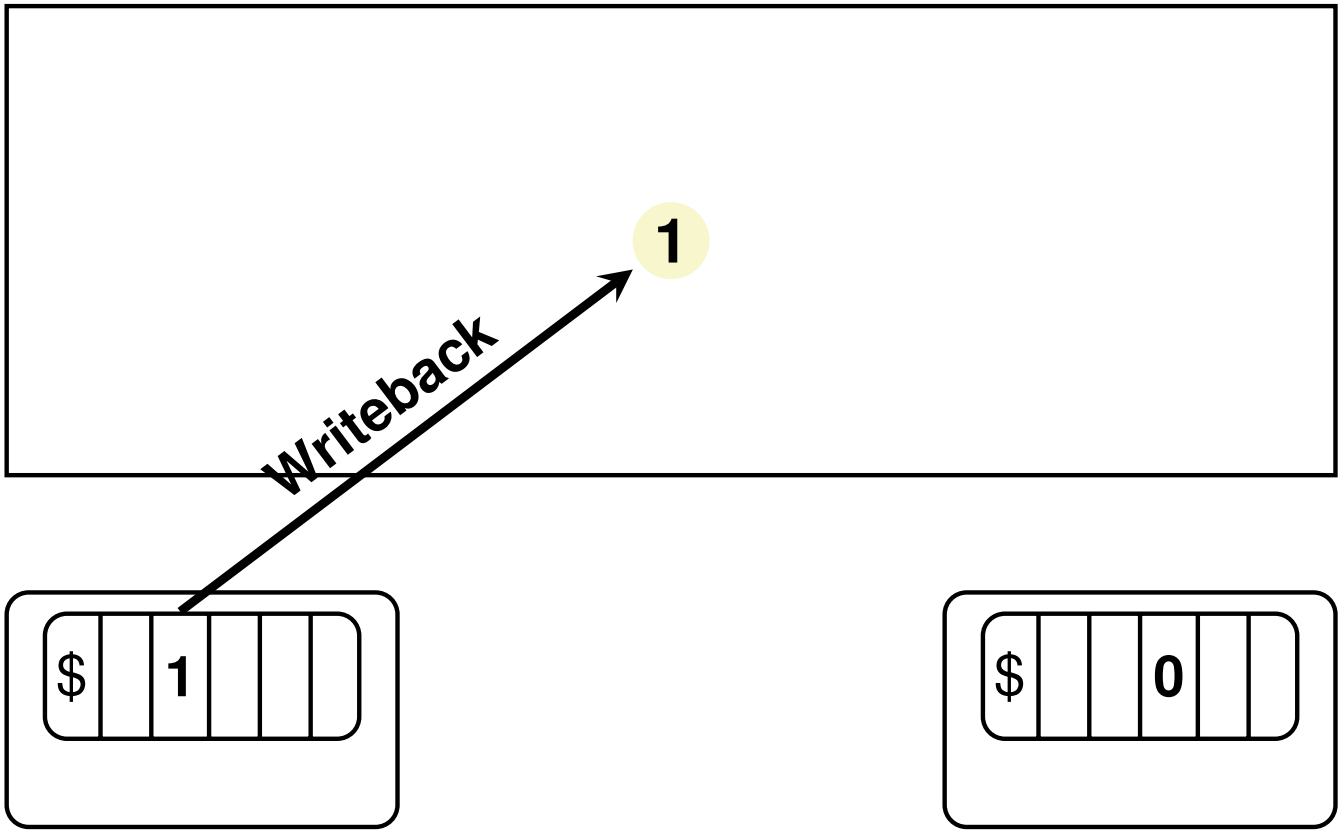
Shared-Memory Programming Model



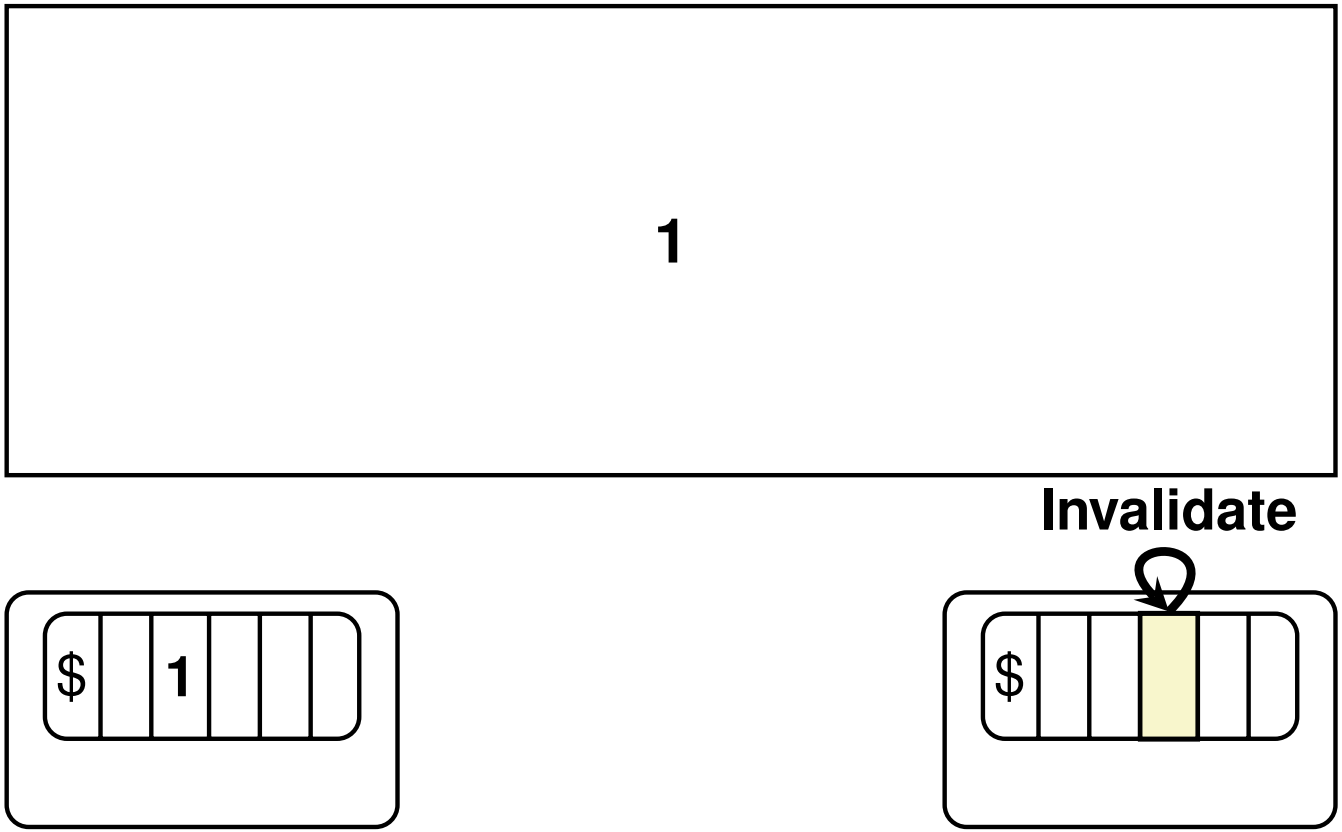
Shared-Memory Programming Model



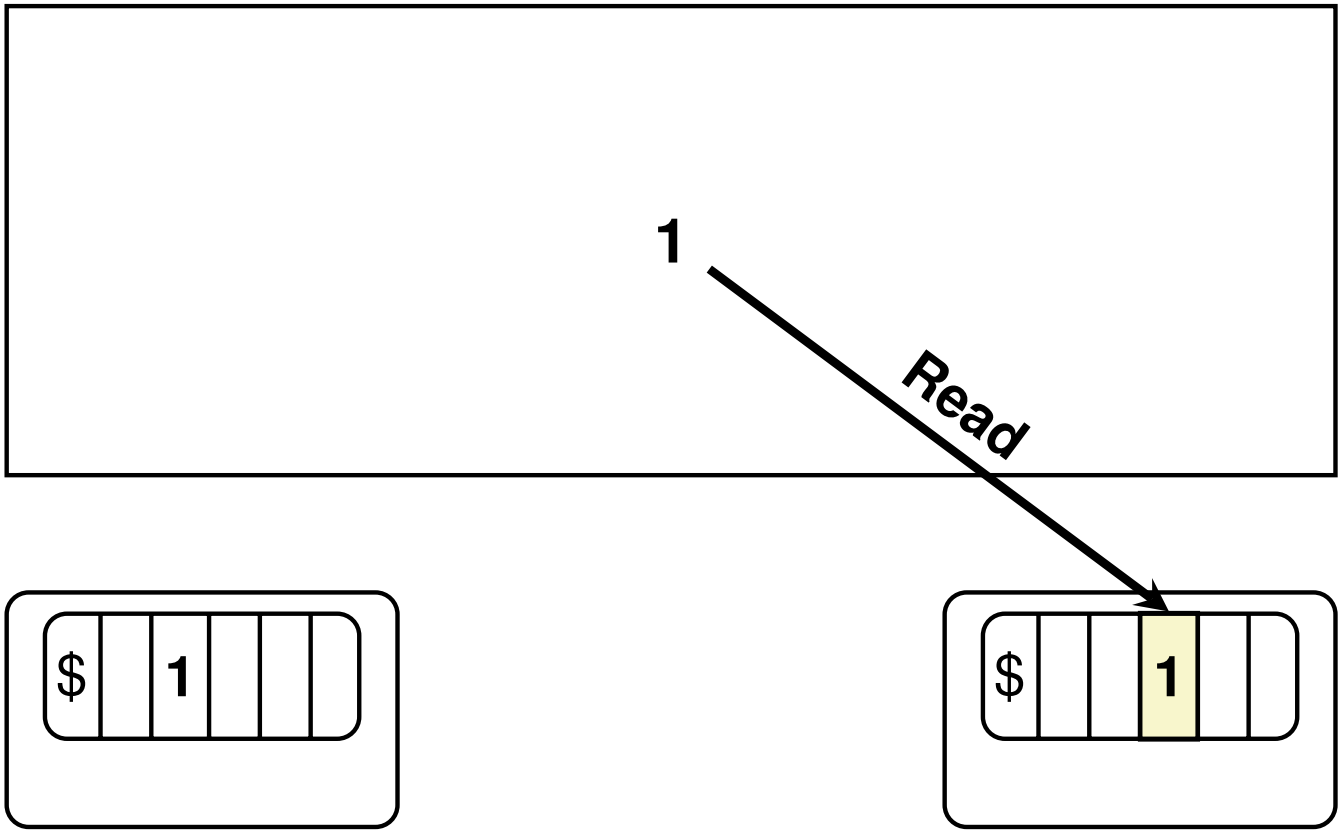
Shared-Memory Programming Model



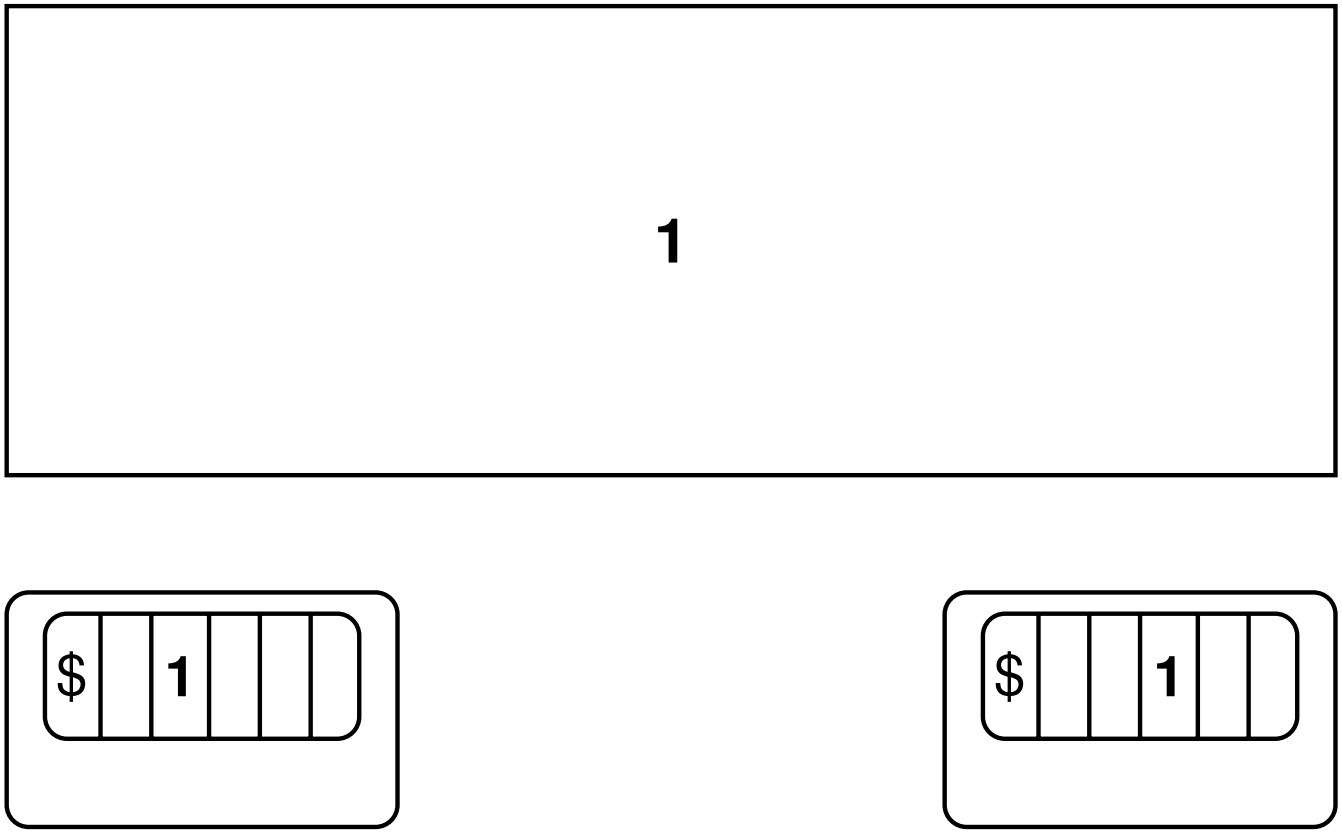
Shared-Memory Programming Model



Shared-Memory Programming Model

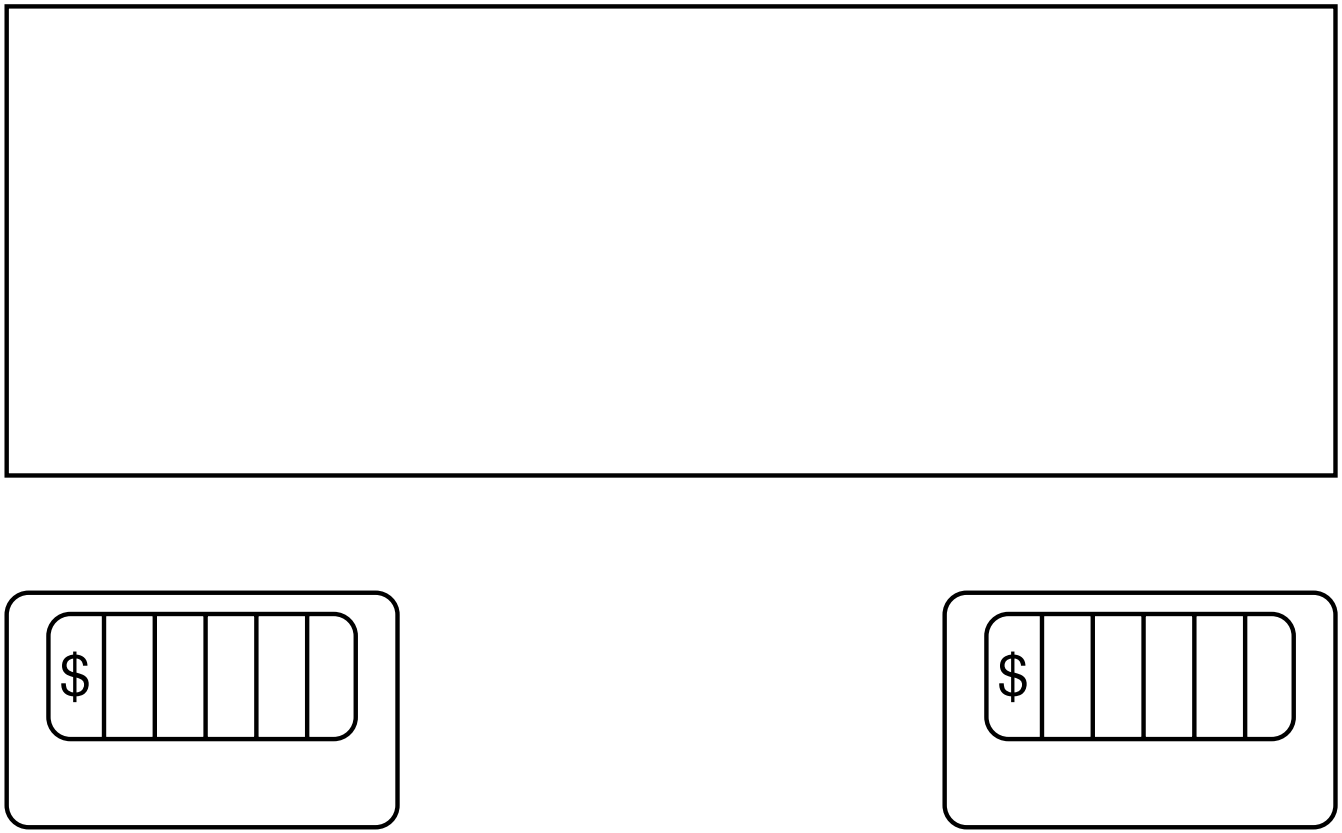


Shared-Memory Programming Model

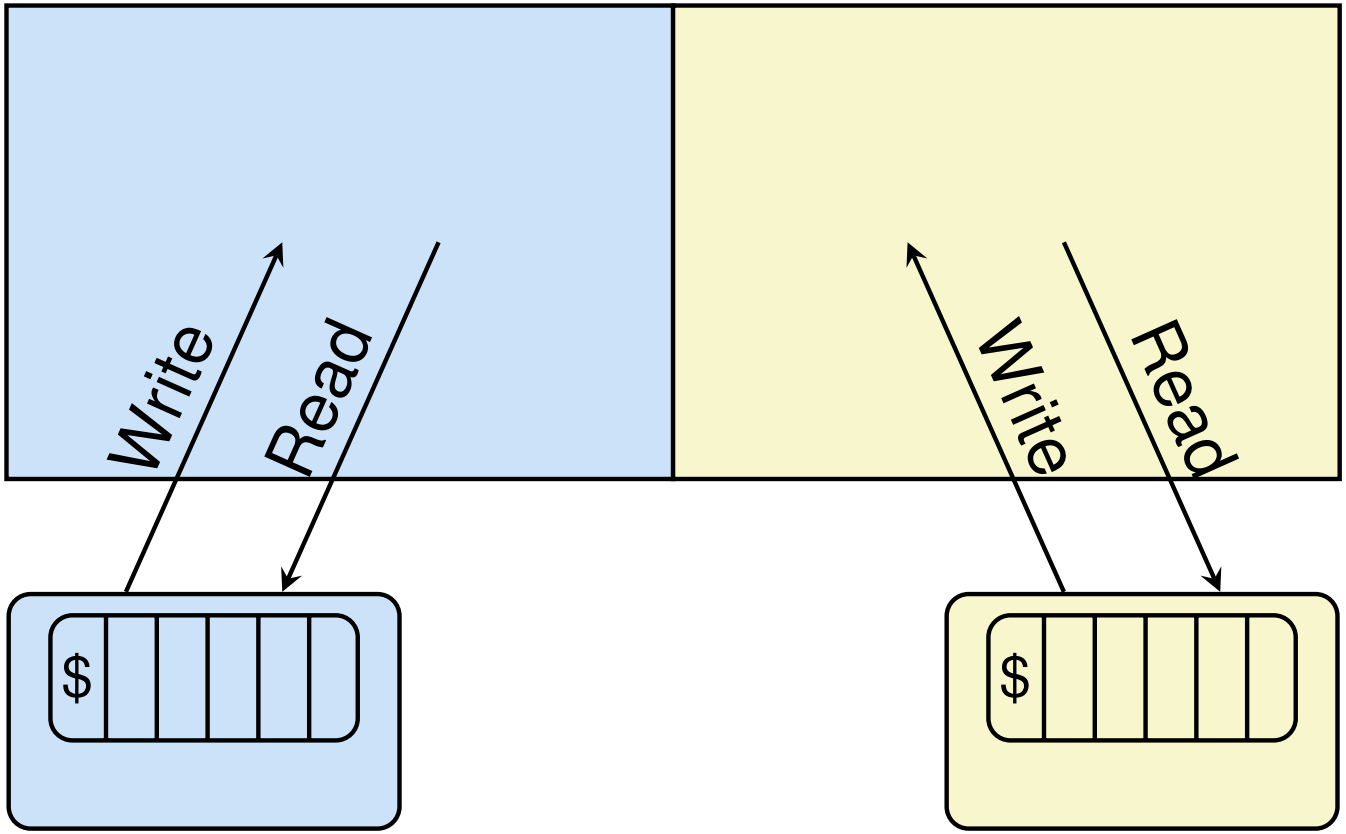


- Feasible, but can be expensive

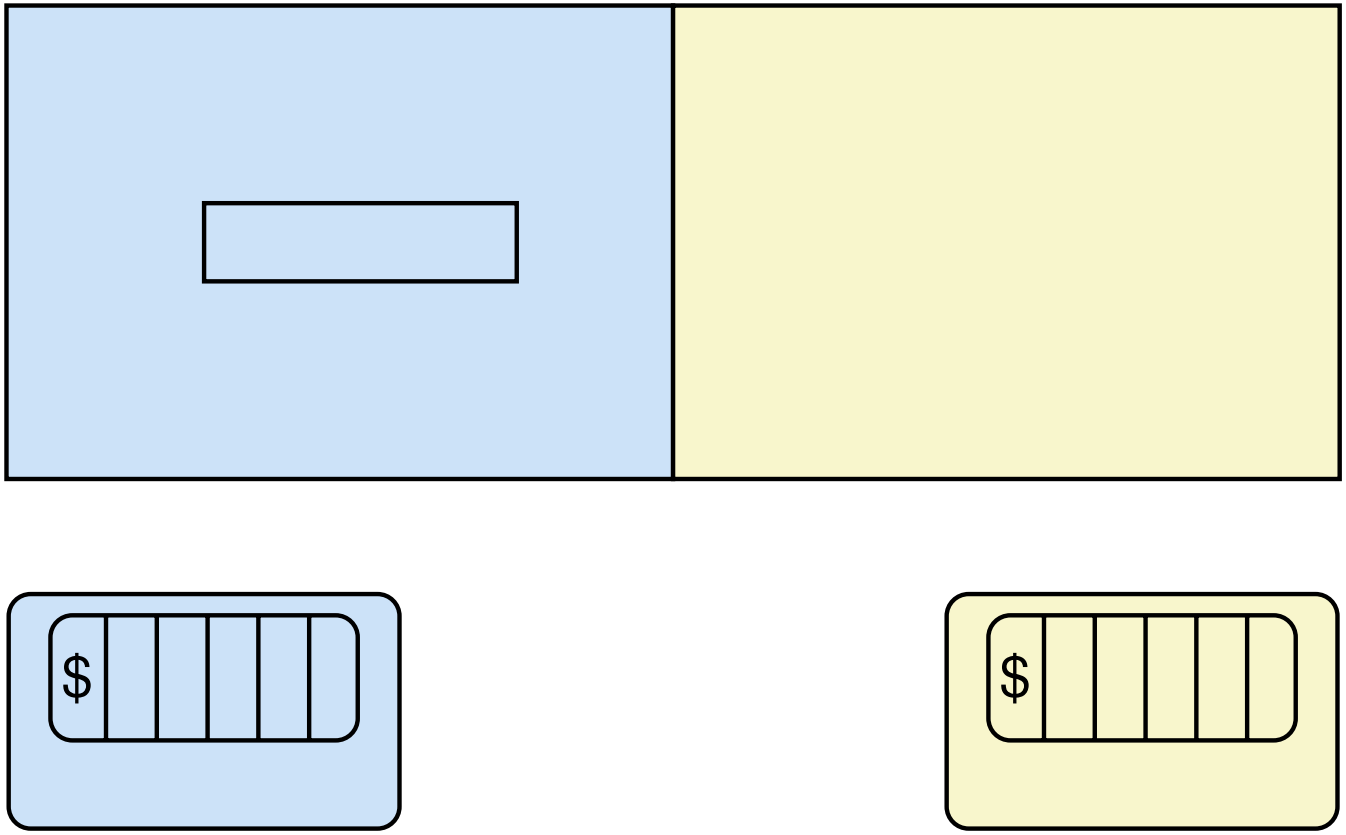
Partitioned Global Address Space (PGAS) Model



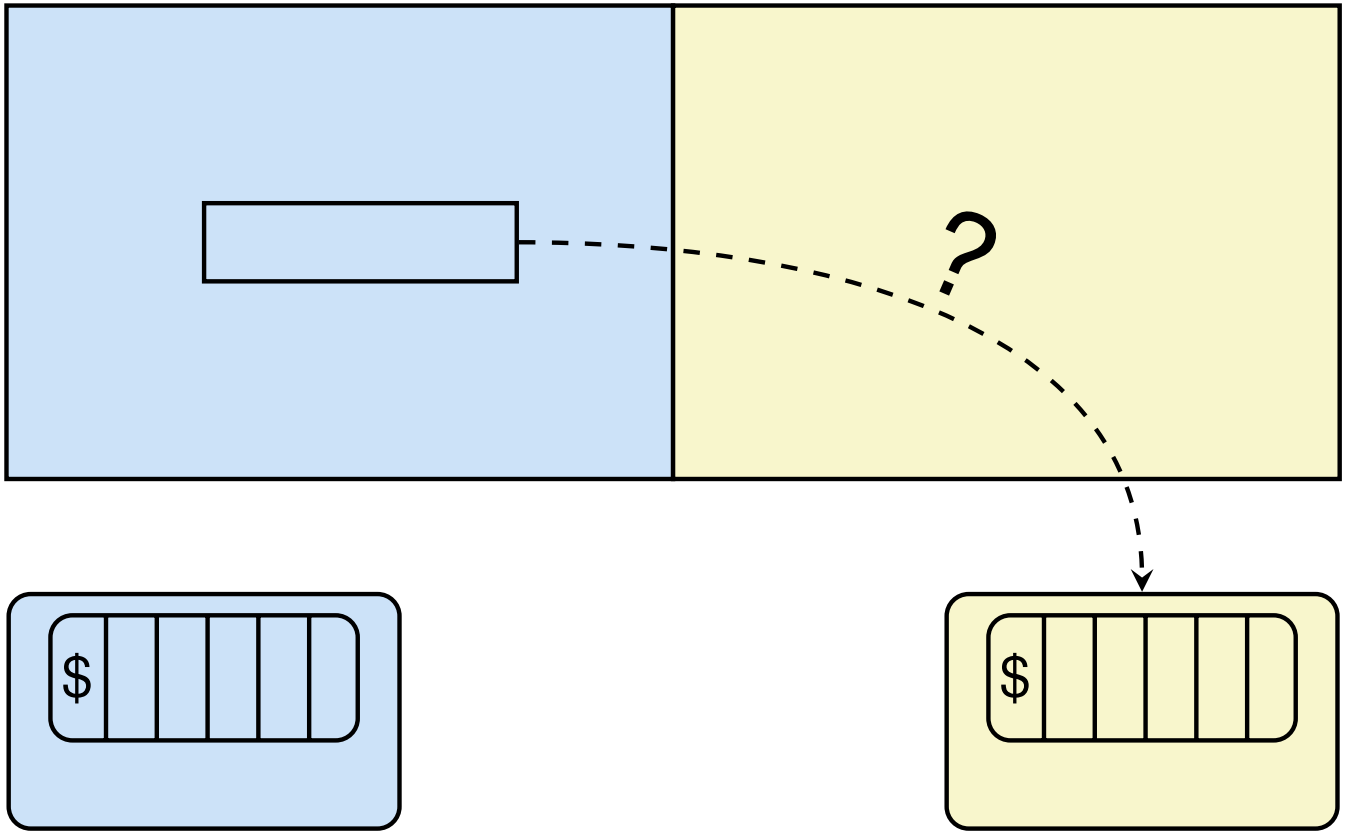
Partitioned Global Address Space (PGAS) Model



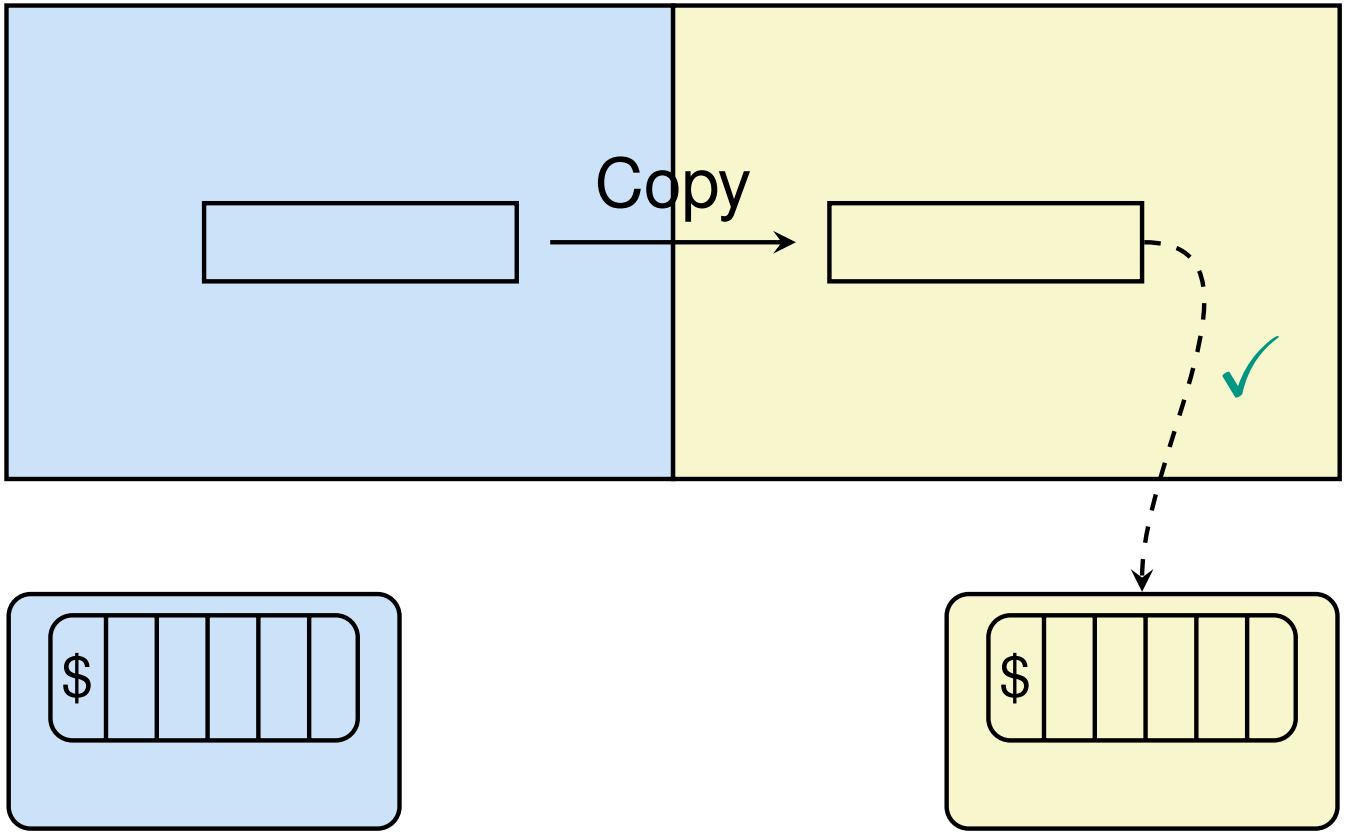
Partitioned Global Address Space (PGAS) Model



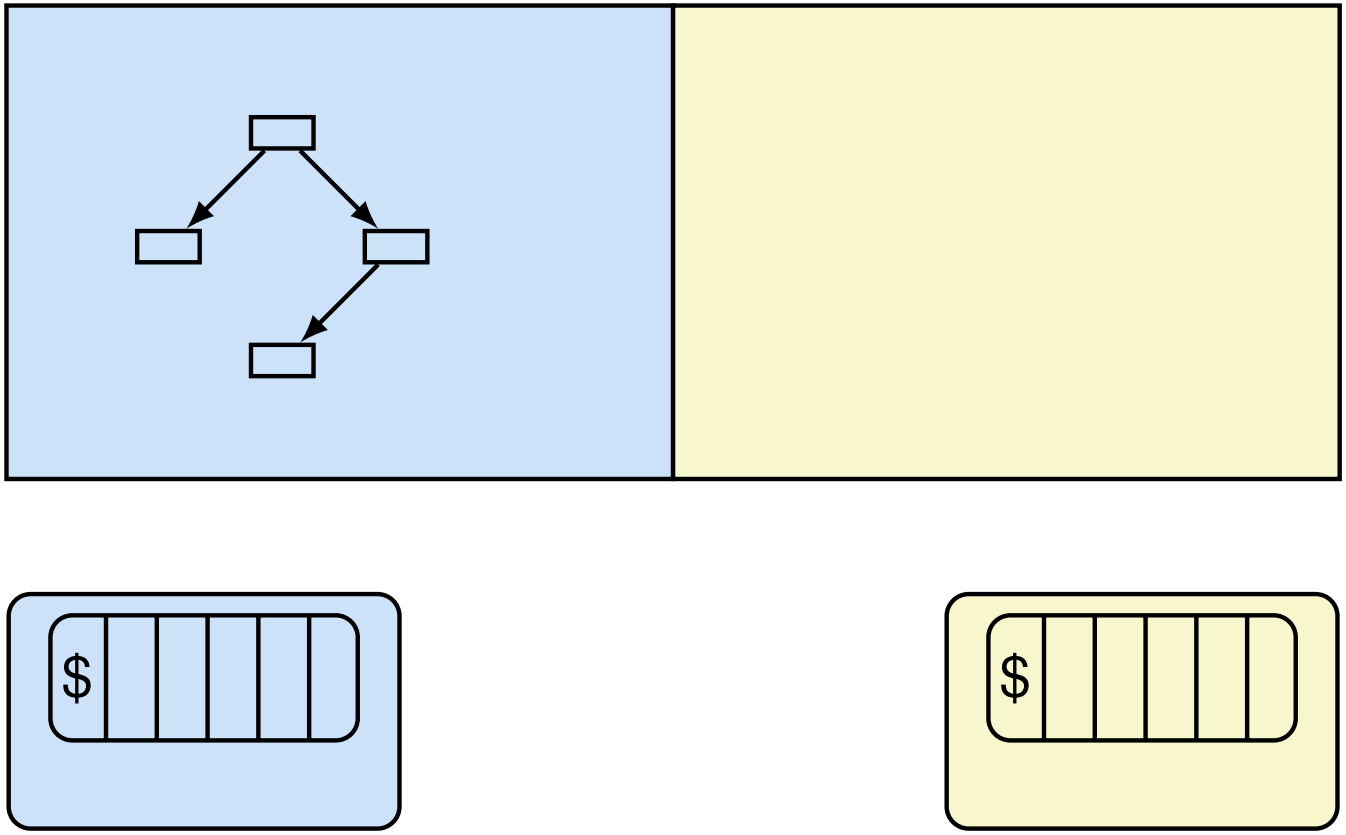
Partitioned Global Address Space (PGAS) Model



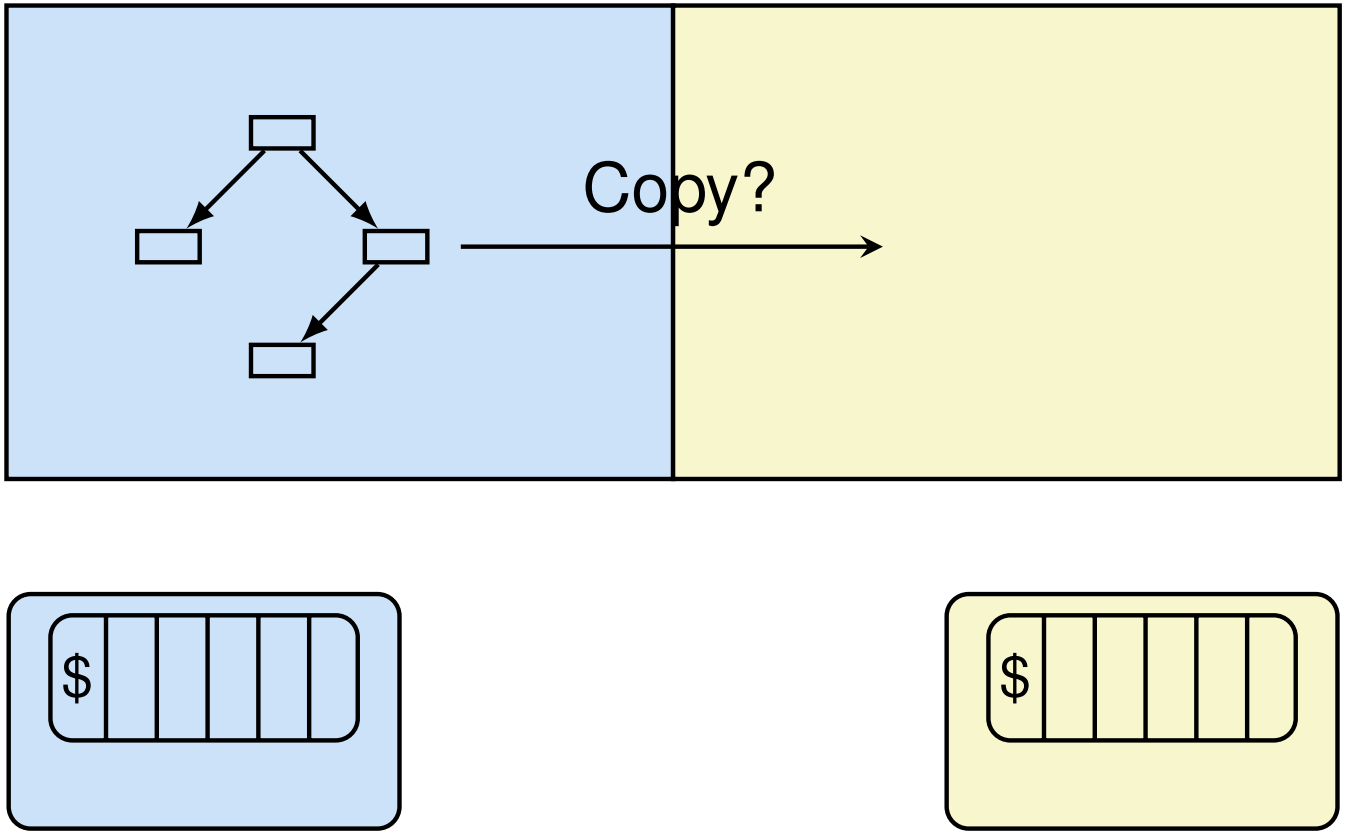
Partitioned Global Address Space (PGAS) Model



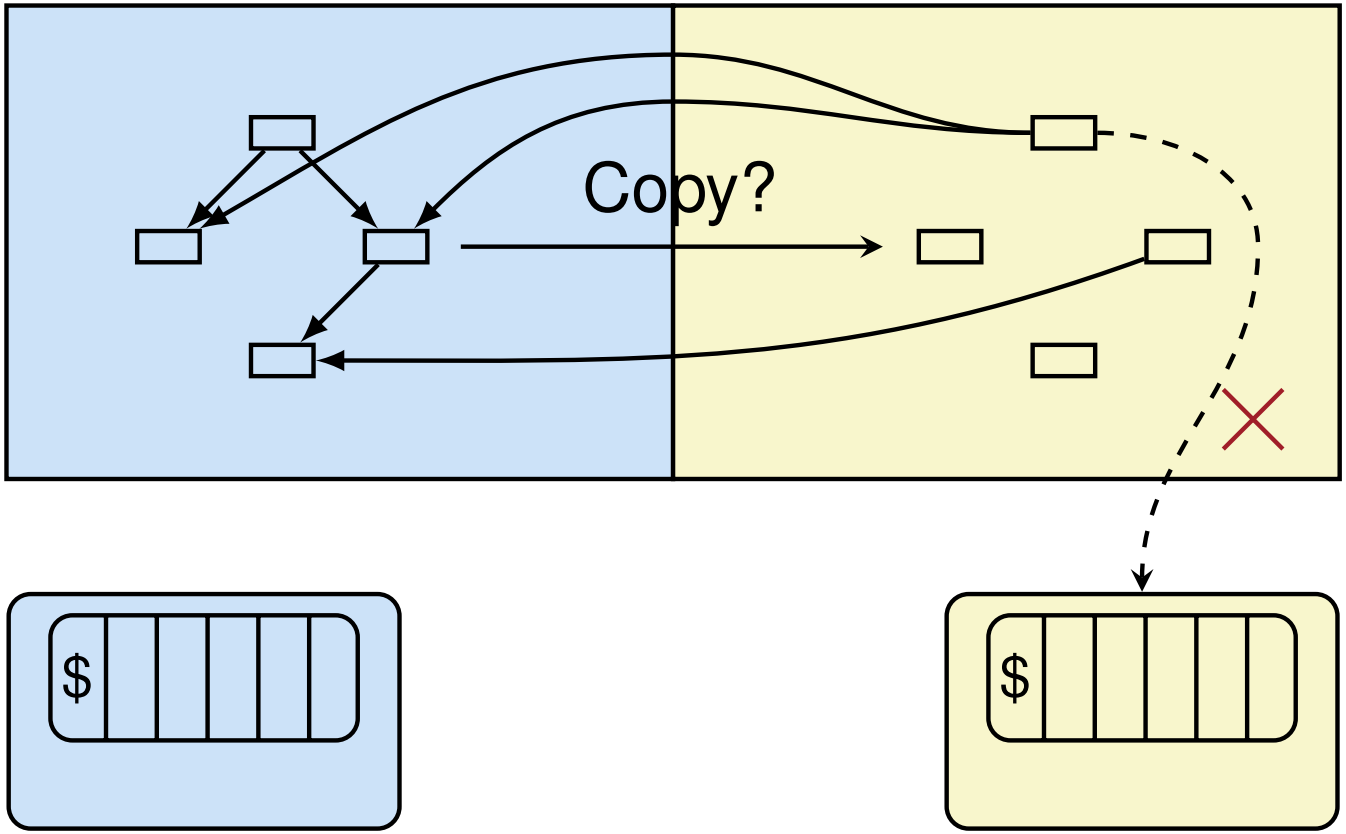
Partitioned Global Address Space (PGAS) Model



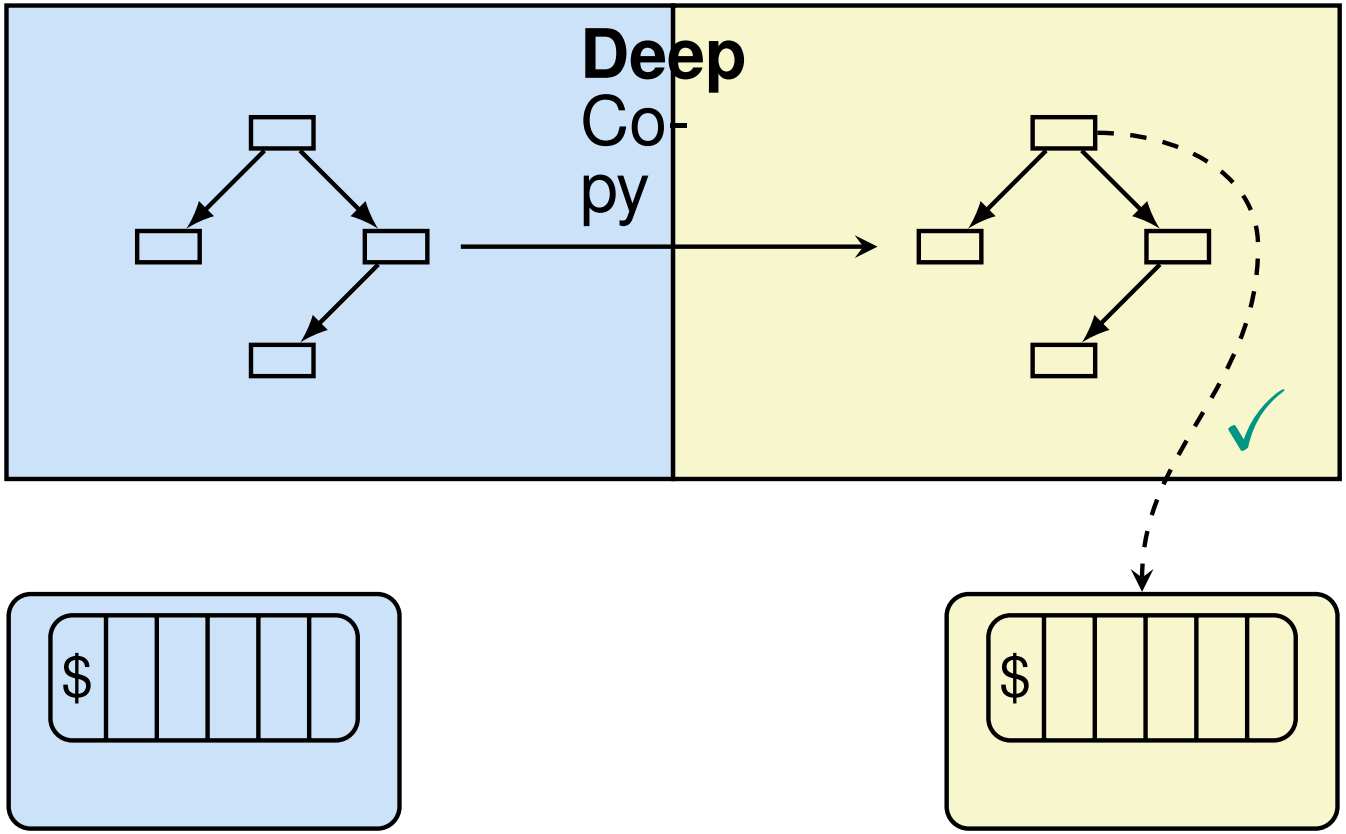
Partitioned Global Address Space (PGAS) Model



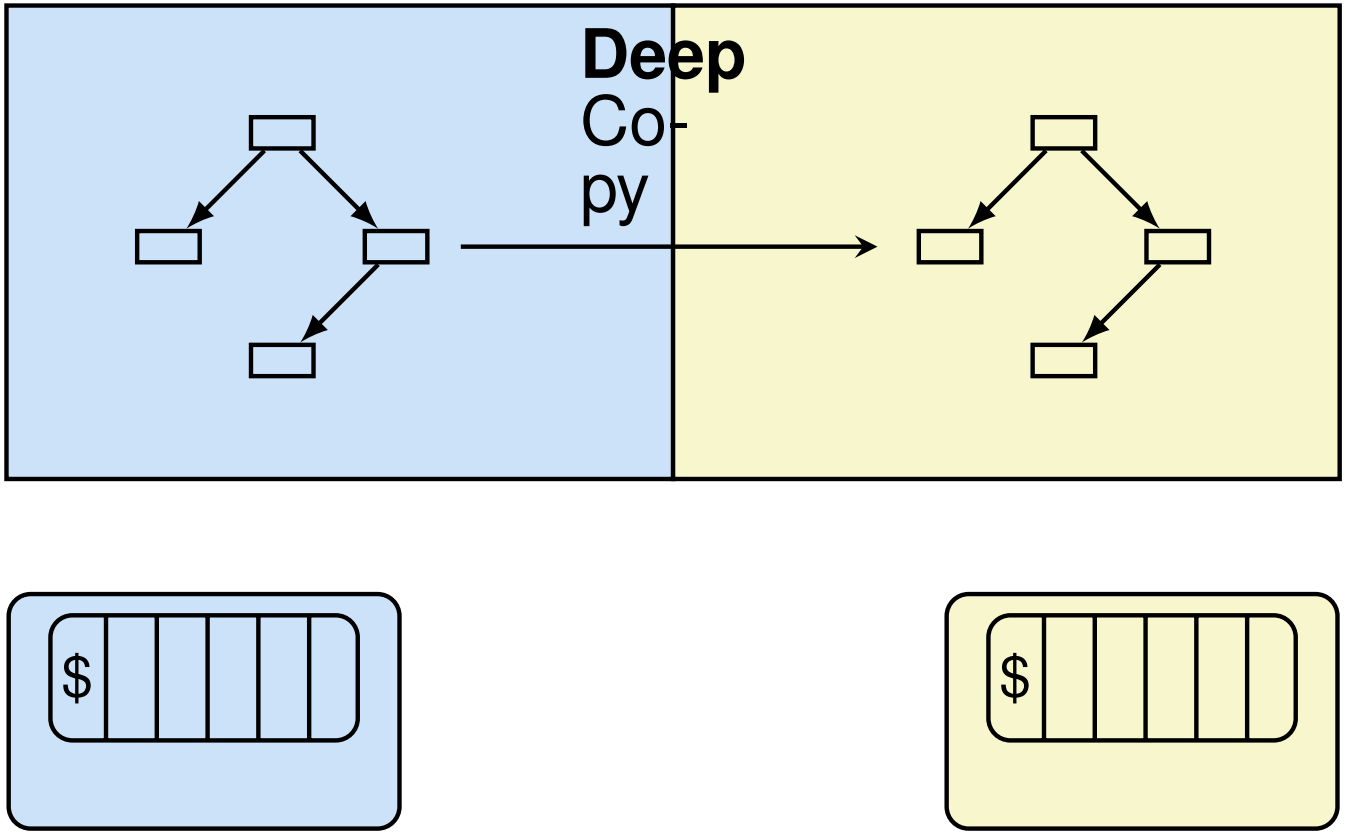
Partitioned Global Address Space (PGAS) Model



Partitioned Global Address Space (PGAS) Model

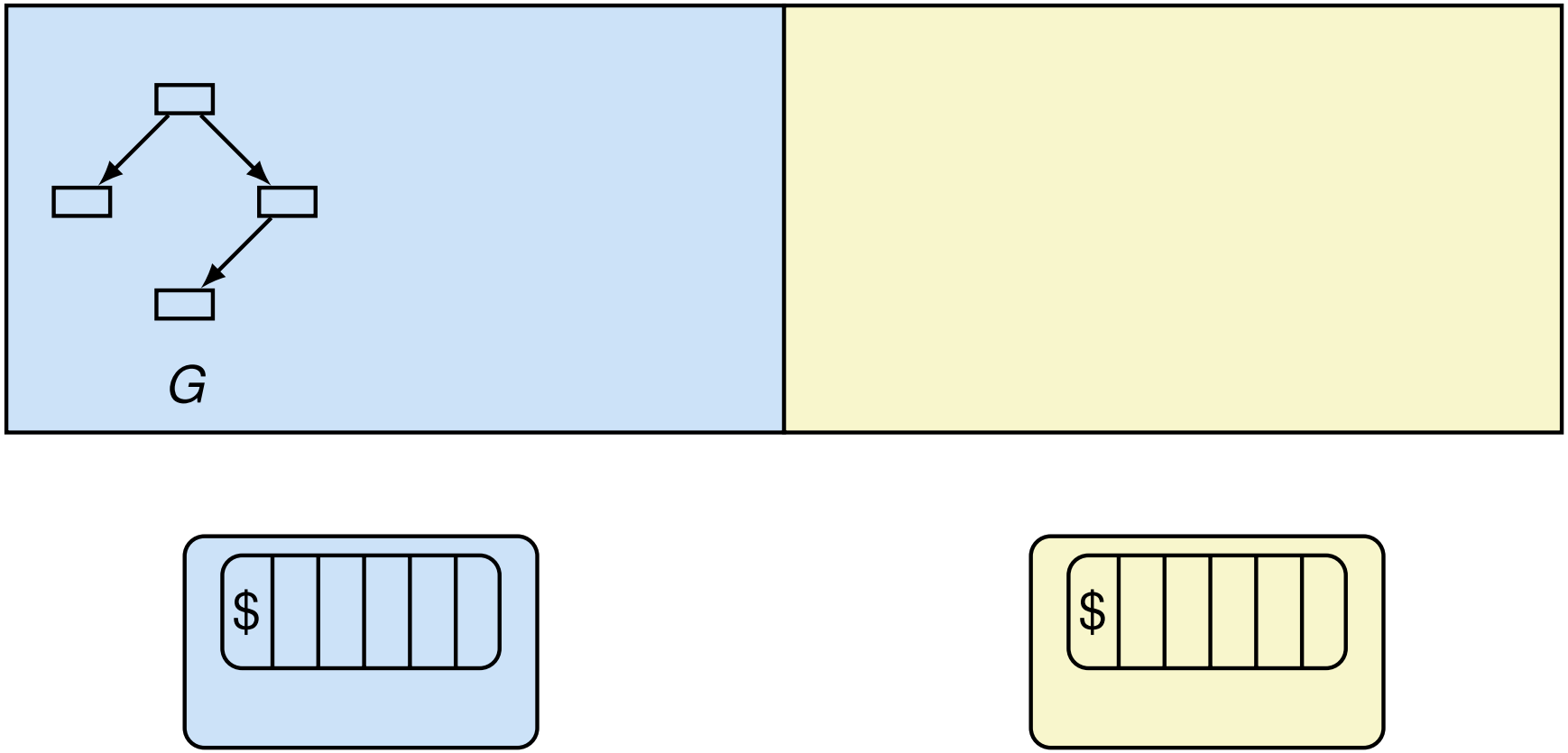


Partitioned Global Address Space (PGAS) Model

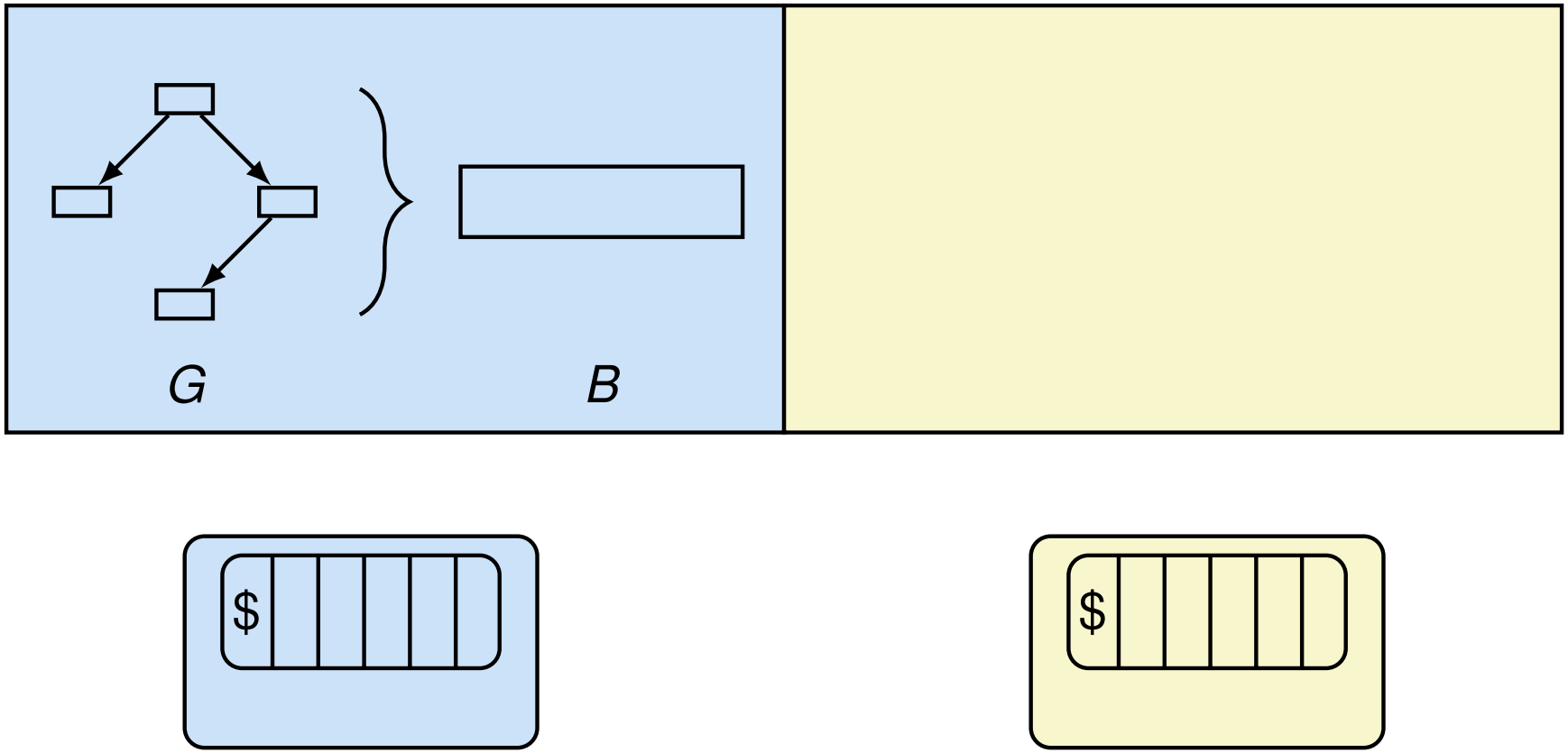


Goal: Efficiently deep copy pointered data structures between shared memory partitions on non-cache-coherent architectures

Message Passing (MP)

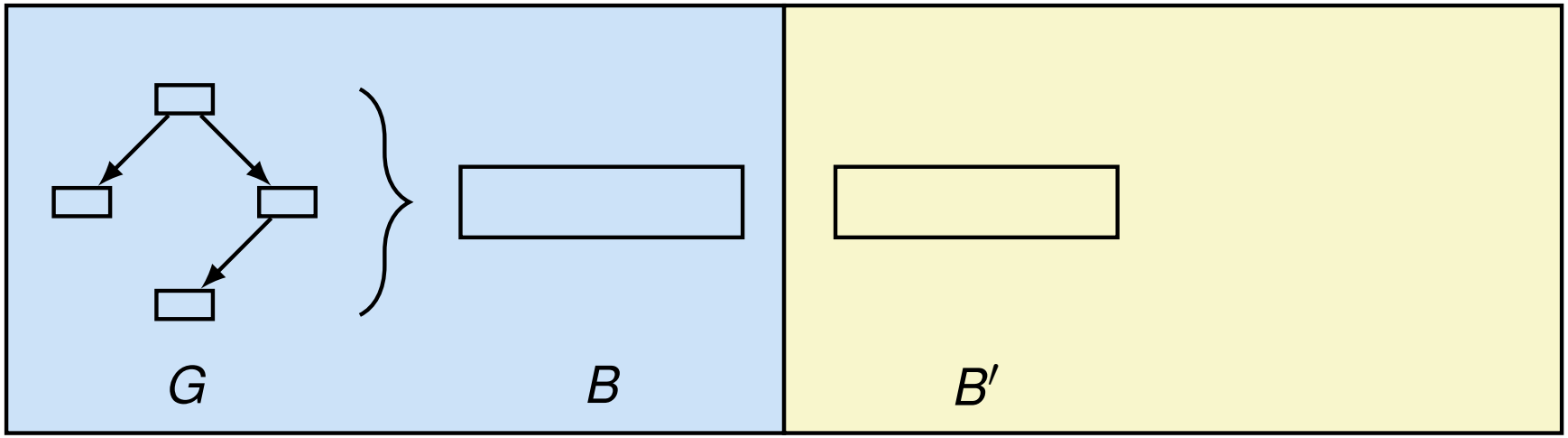


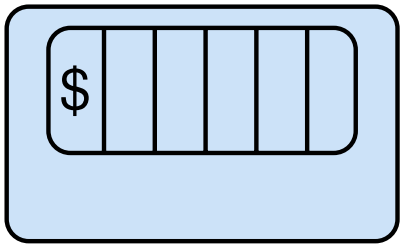
Message Passing (MP)

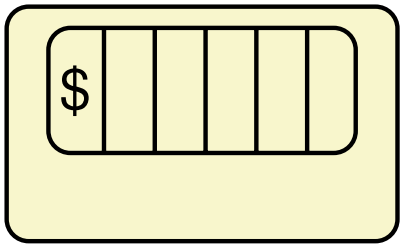


1. Serialize *G* to *B*

Message Passing (MP)

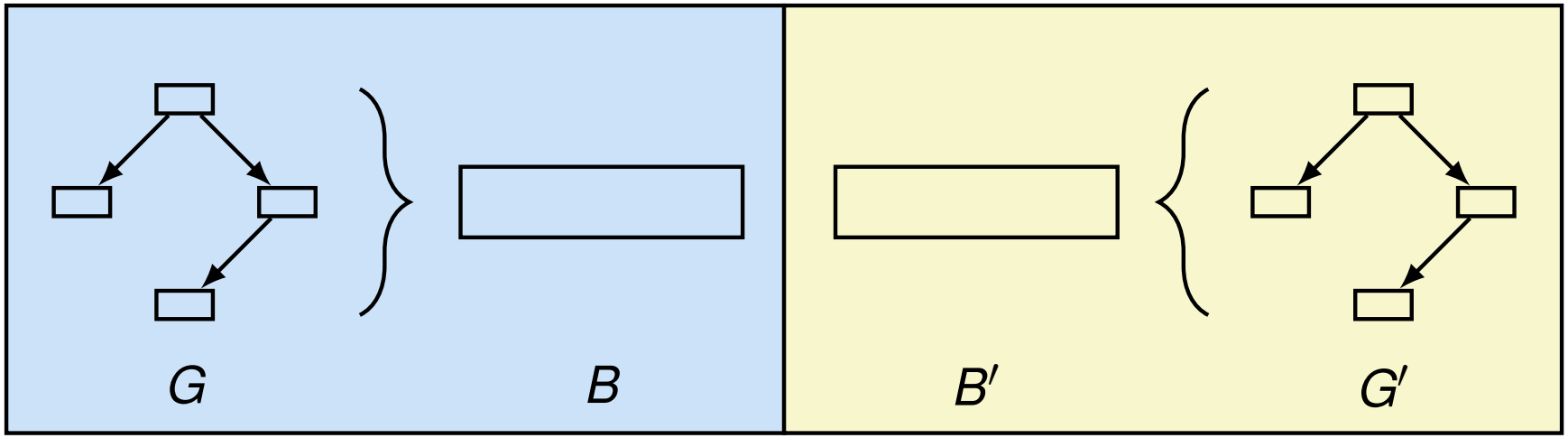


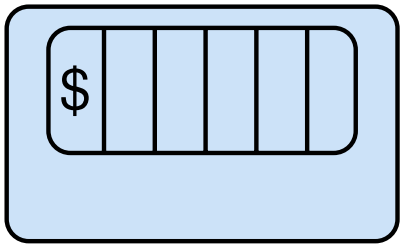


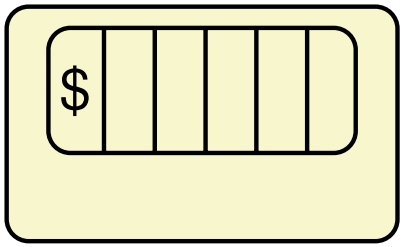


1. Serialize G to B
2. Transfer B to B', e.g. using library

Message Passing (MP)

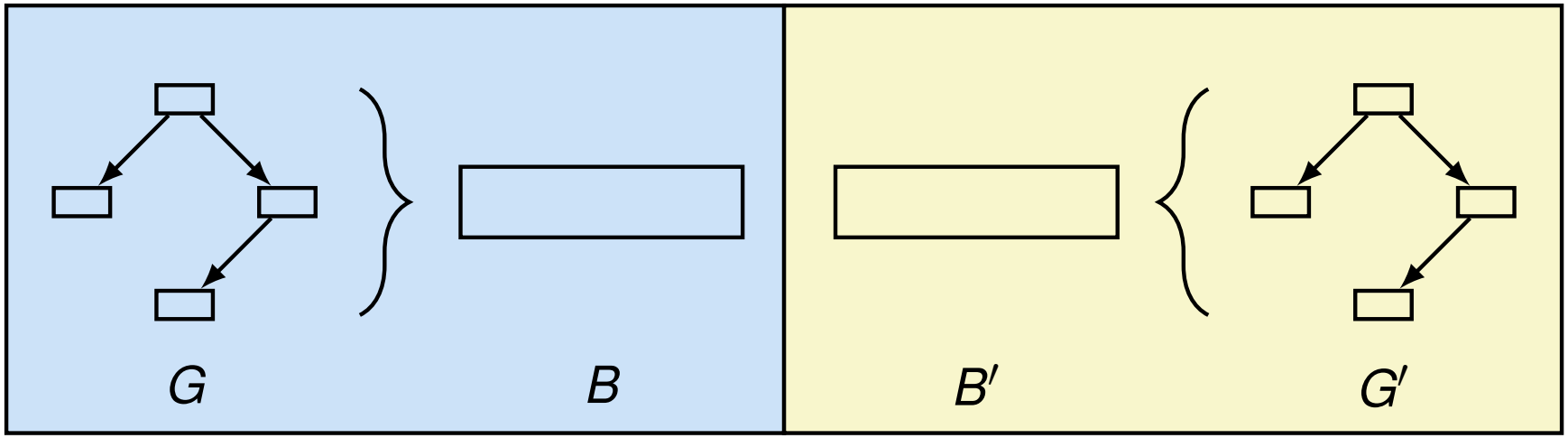


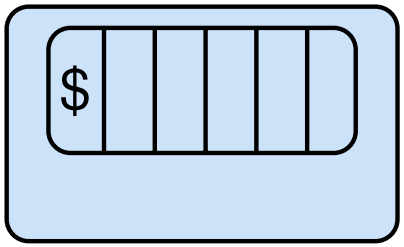


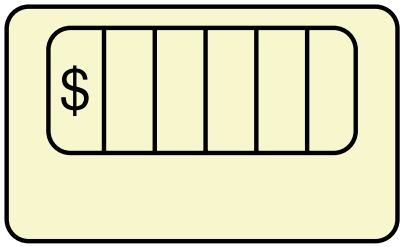


1. Serialize G to B
2. Transfer B to B', e.g. using library
3. Deserialize G' from B'

Message Passing (MP)



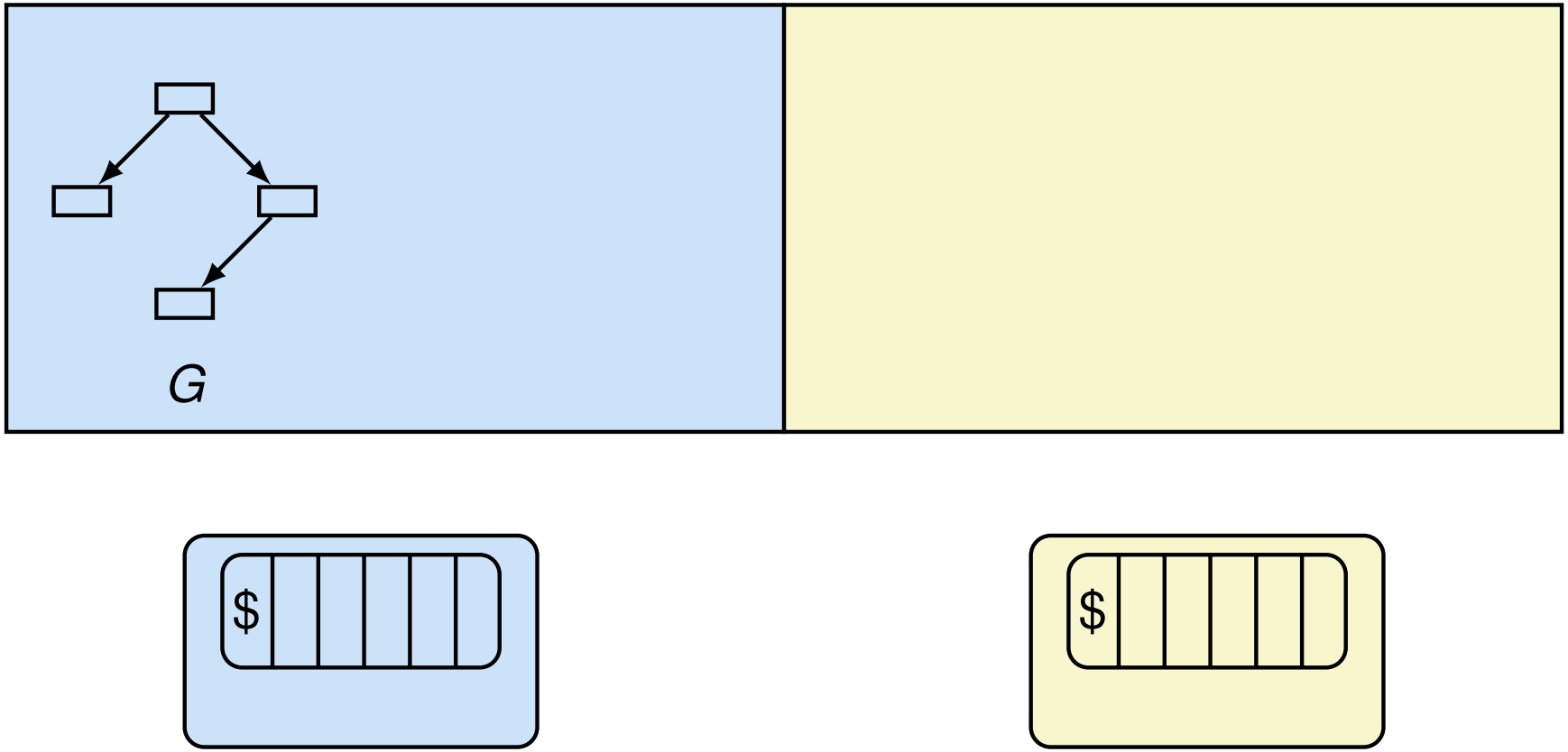




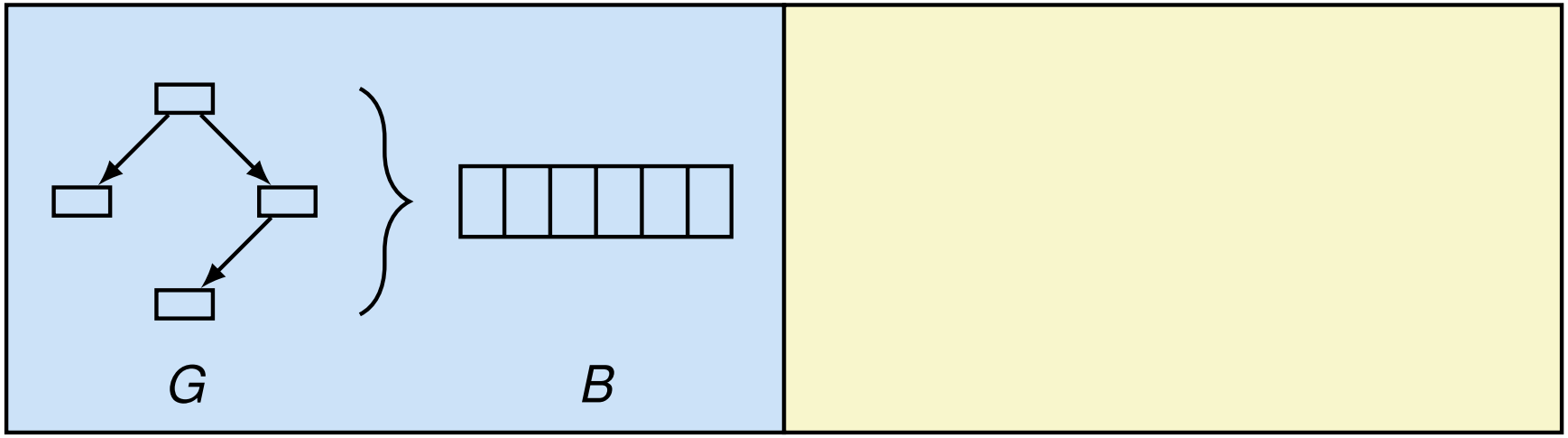
1. Serialize G to B
2. Transfer B to B', e.g. using library
3. Deserialize G' from B'

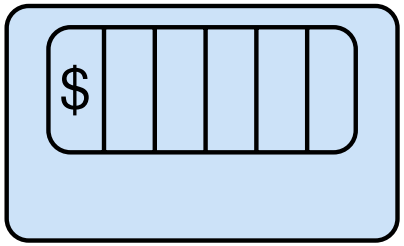
- Large memory overhead
- Serialization overhead
- B, B' pollute caches

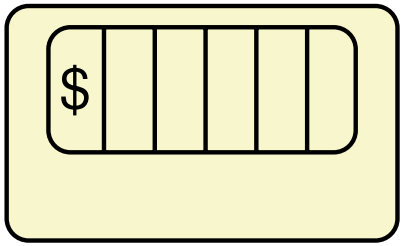
Message Passing via Shared Memory (MP-SHM)



Message Passing via Shared Memory (MP-SHM)

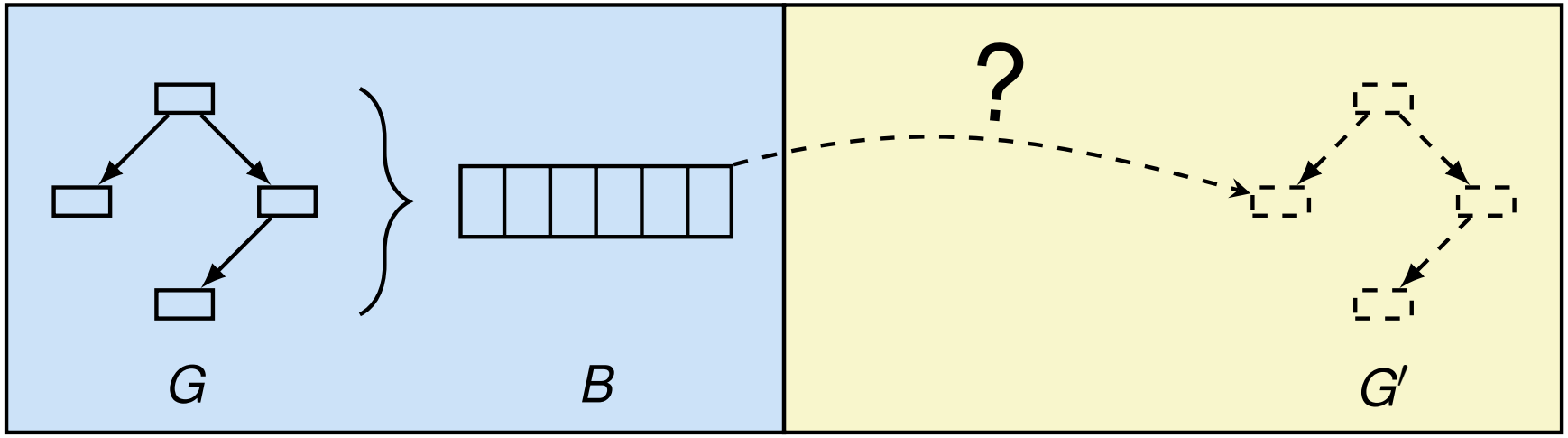


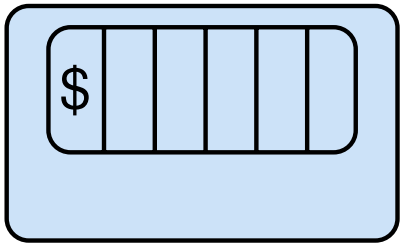


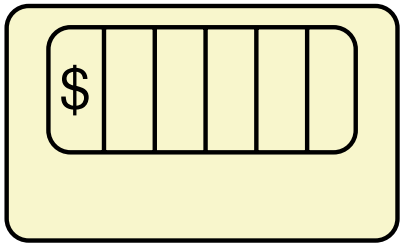


1. Serialize *G* to *B*

Message Passing via Shared Memory (MP-SHM)

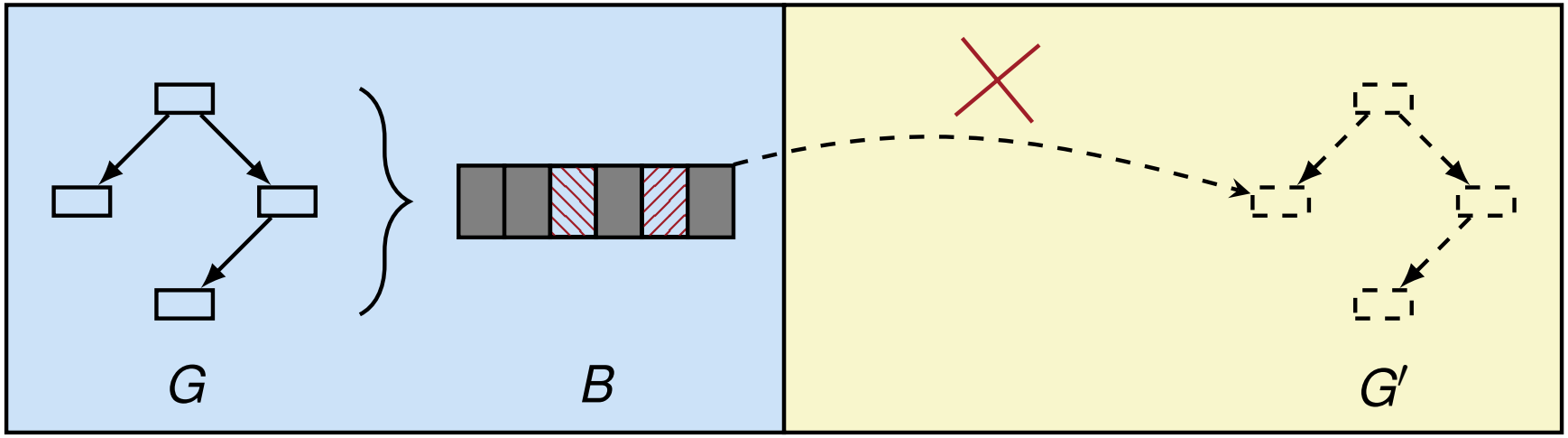


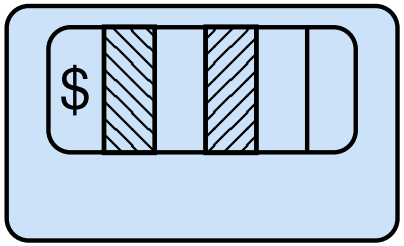


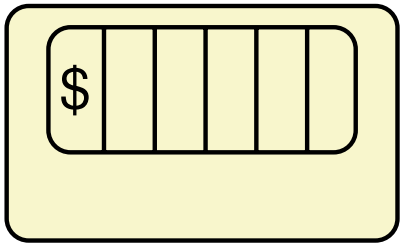


1. Serialize *G* to *B*

Message Passing via Shared Memory (MP-SHM)

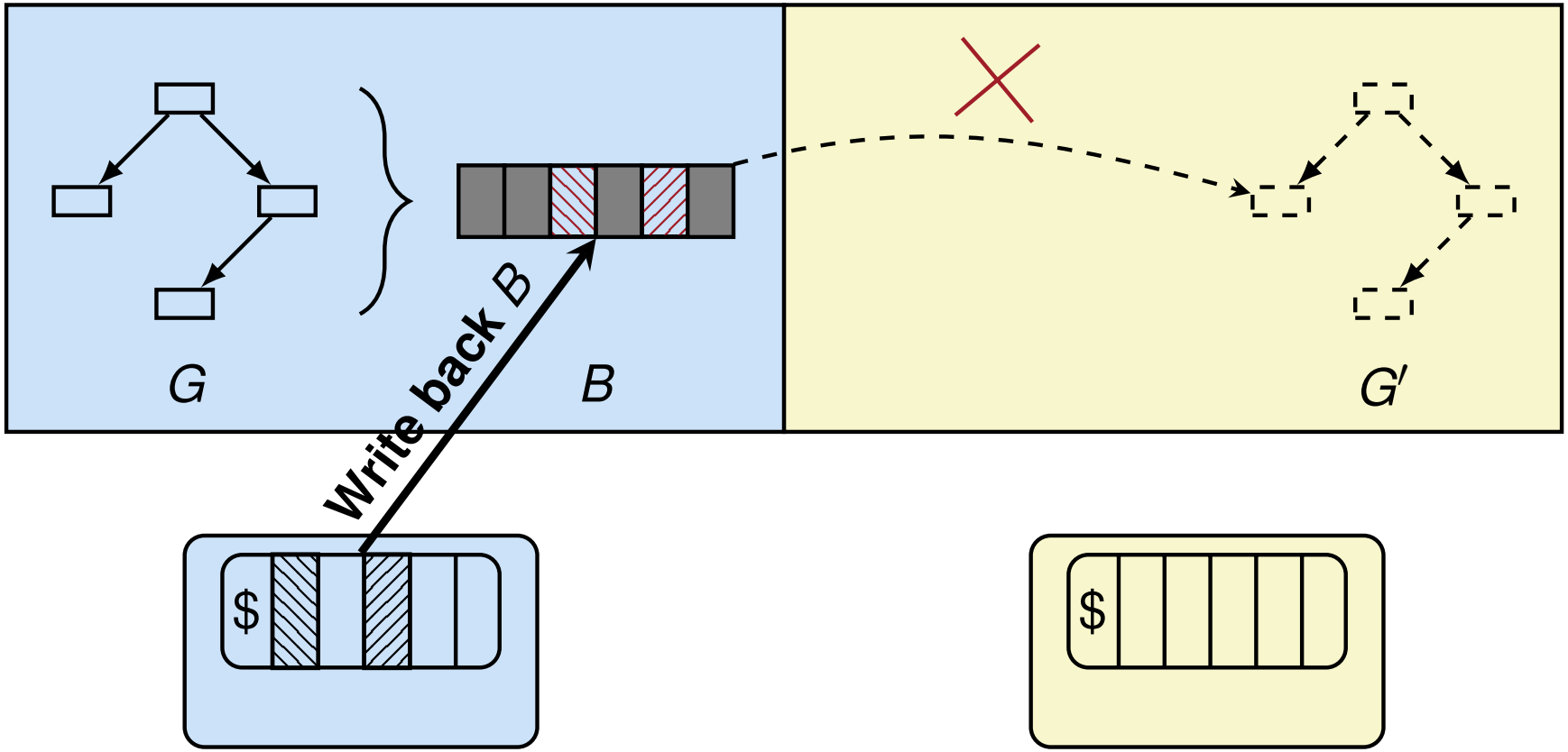






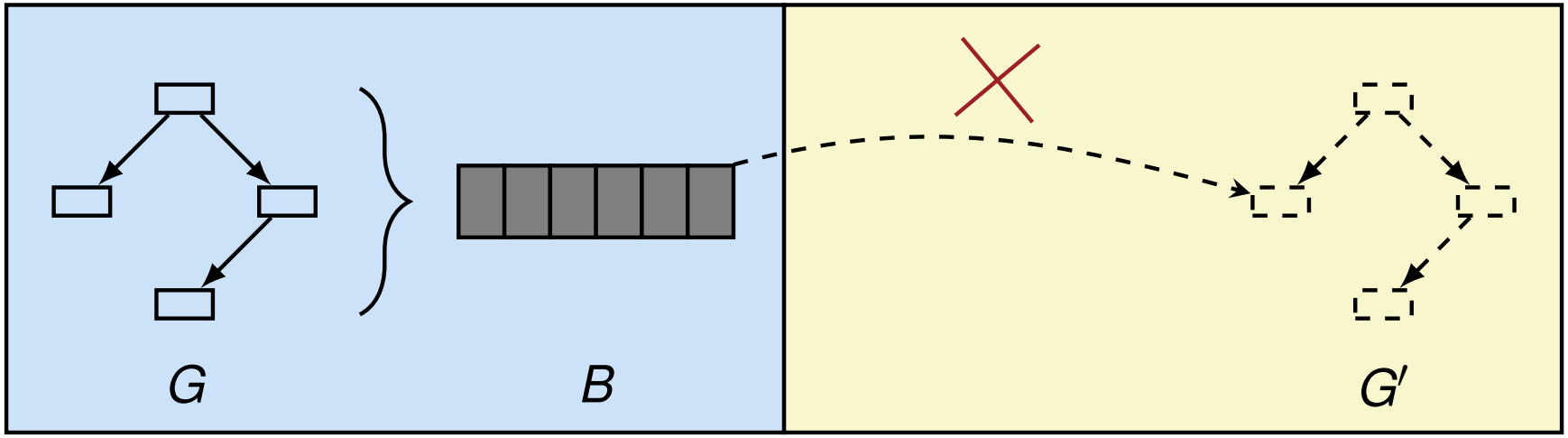
1. Serialize *G* to *B*

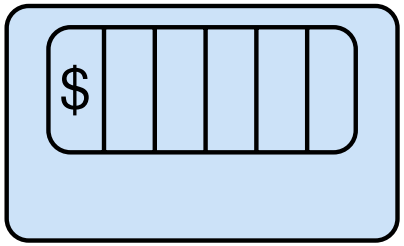
Message Passing via Shared Memory (MP-SHM)

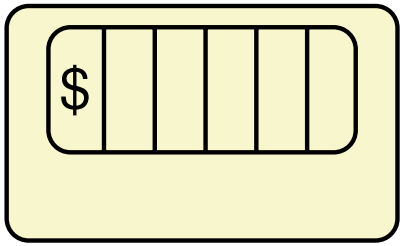


1. Serialize *G* to *B* and write back *B*

Message Passing via Shared Memory (MP-SHM)

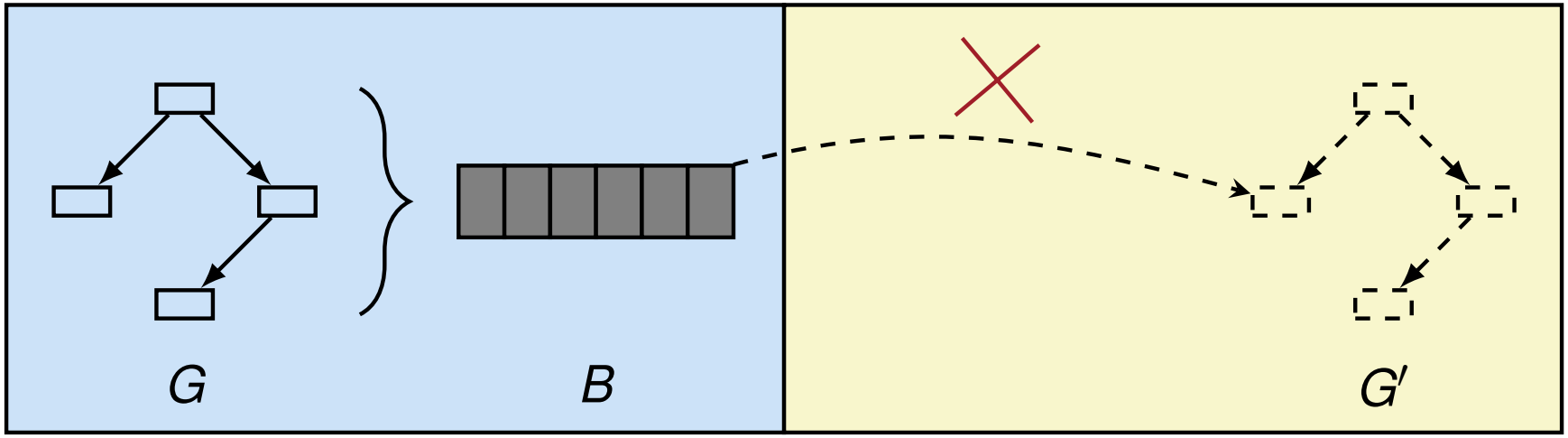


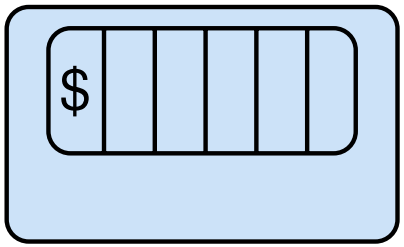


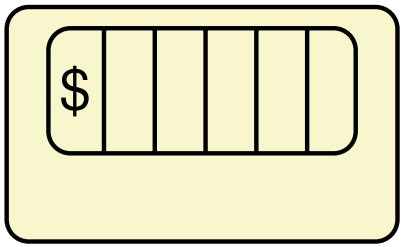


1. Serialize *G* to *B* and write back *B*

Message Passing via Shared Memory (MP-SHM)

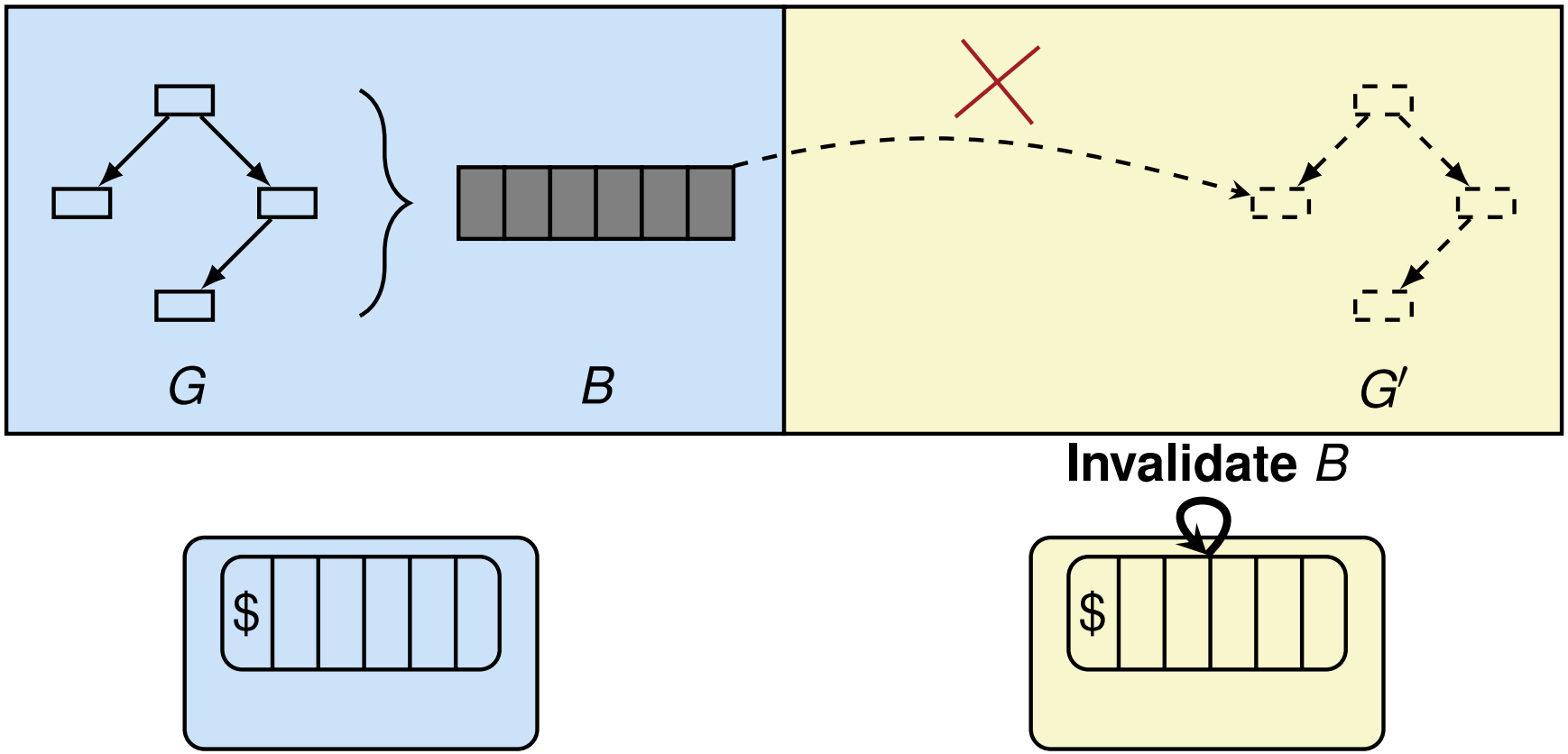






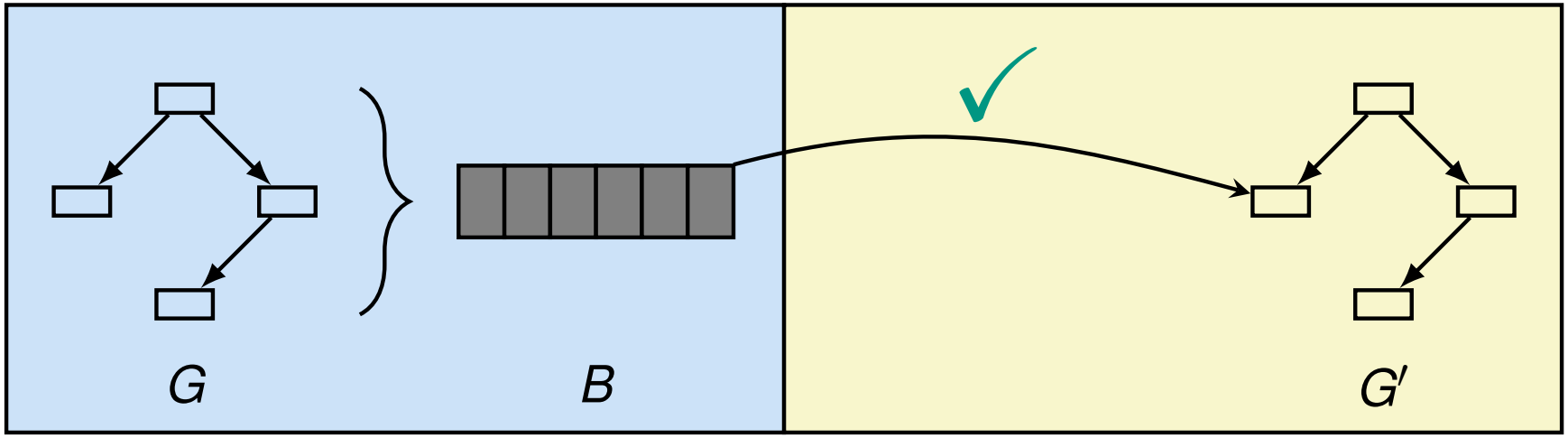
1. Serialize *G* to *B* and write back *B*
2. Notify yellow core with *B*'s address & size

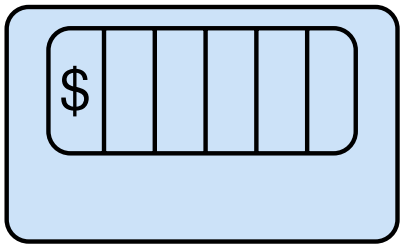
Message Passing via Shared Memory (MP-SHM)

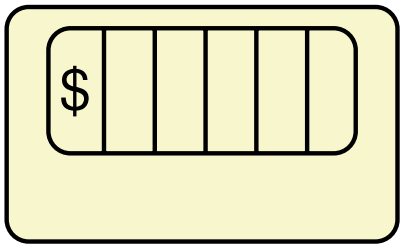


1. Serialize *G* to *B* and write back *B*
2. Notify yellow core with *B*'s address & size
3. Invalidate *B*

Message Passing via Shared Memory (MP-SHM)

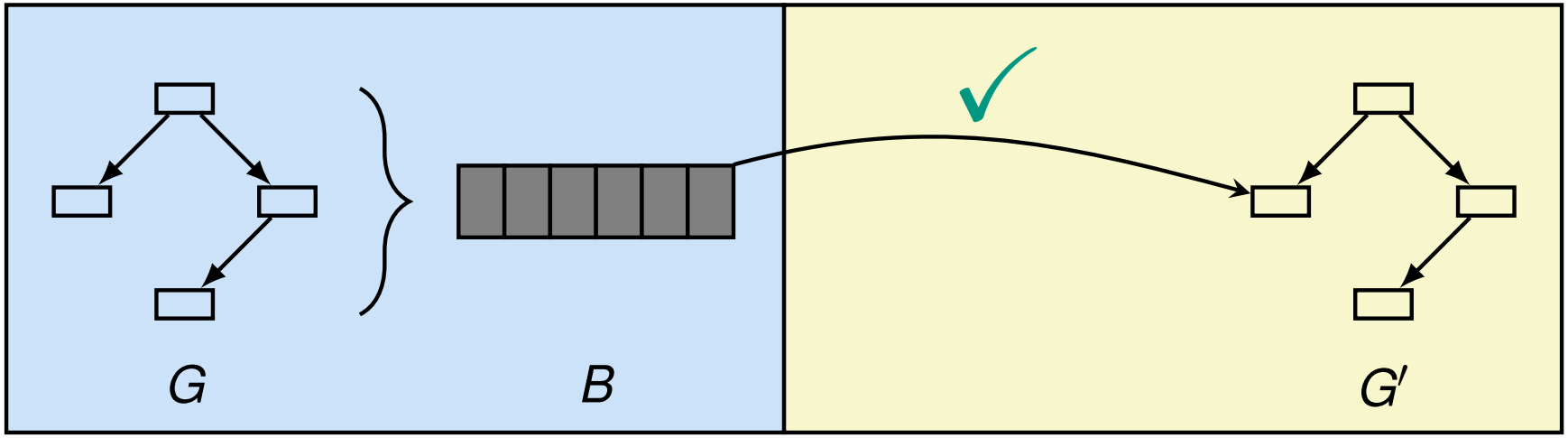


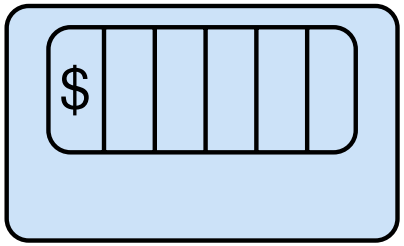


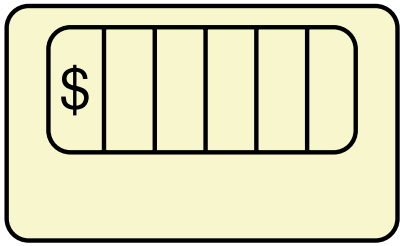


1. Serialize *G* to *B* and write back *B*
2. Notify yellow core with *B*'s address & size
3. Invalidate *B* and deserialize *G′* from *B*

Message Passing via Shared Memory (MP-SHM)



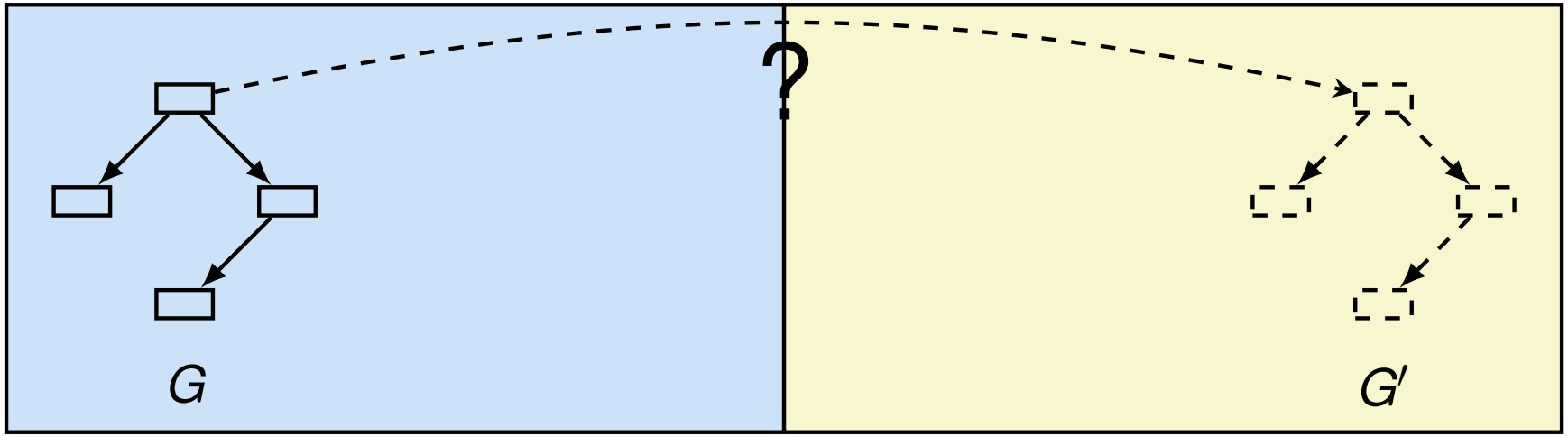


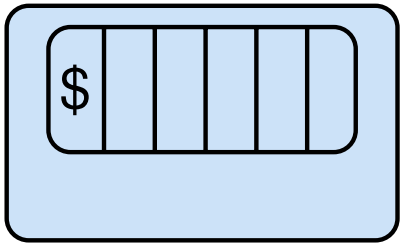


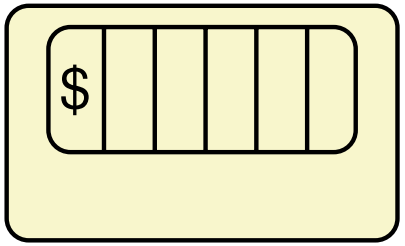
1. Serialize *G* to *B* and write back *B*
2. Notify yellow core with *B*'s address & size
3. Invalidate *B* and deserialize *G′* from *B*

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

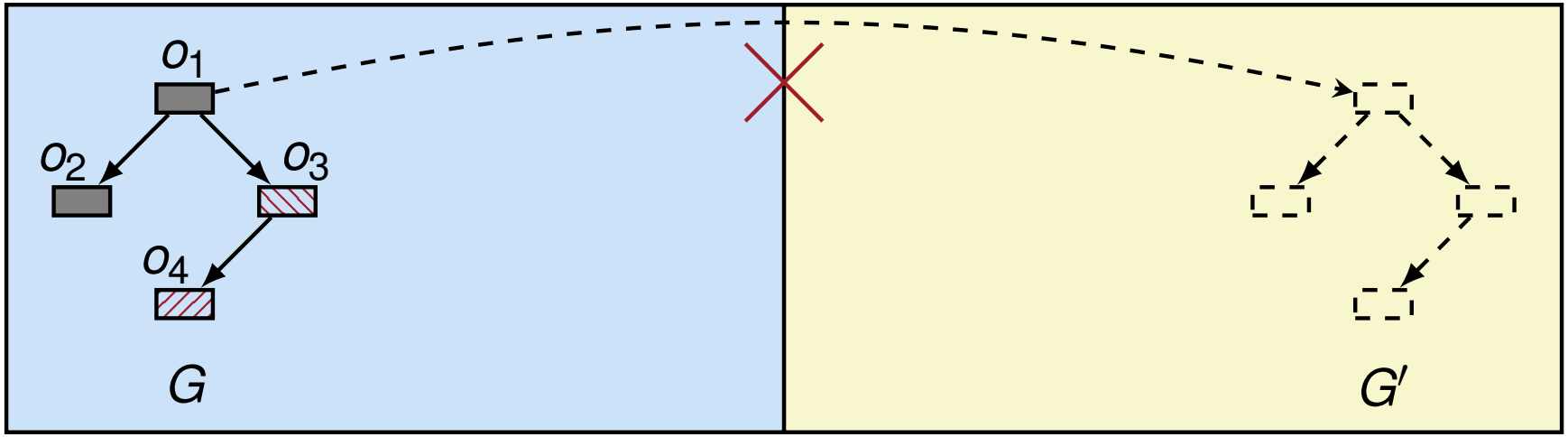
Cloning (CLONE)

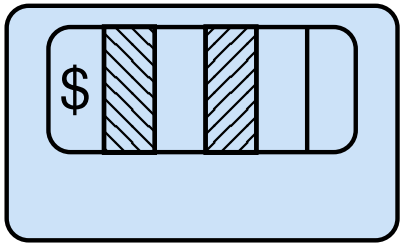


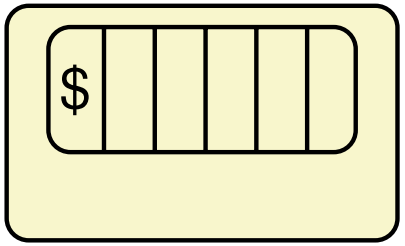




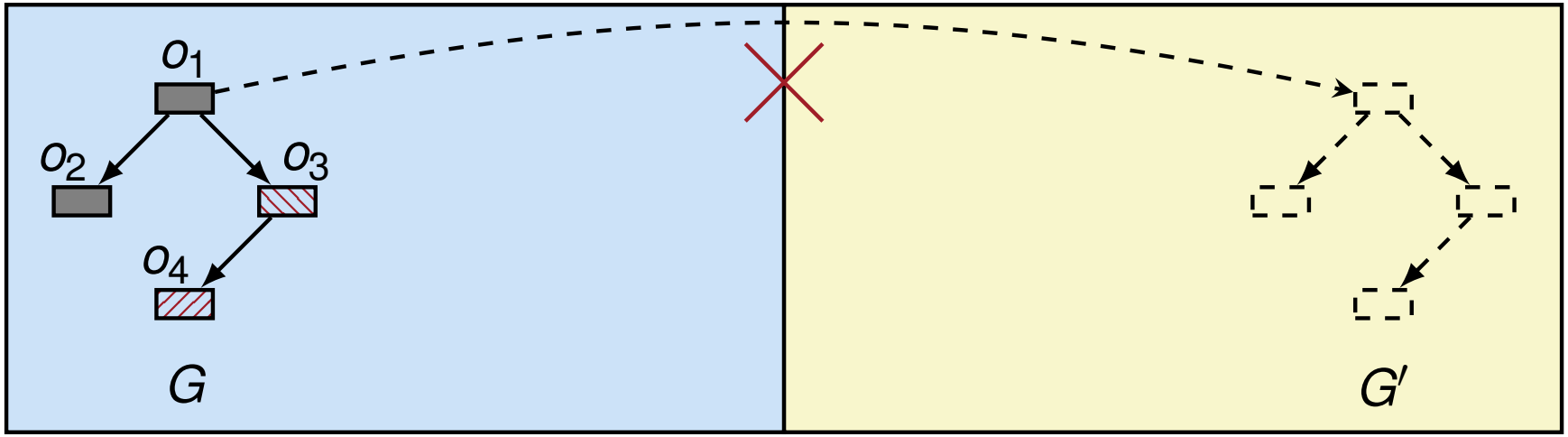
Cloning (CLONE)

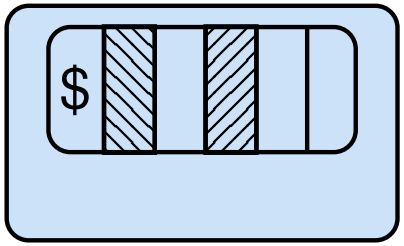


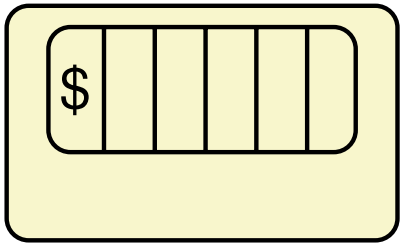




Cloning (CLONE)

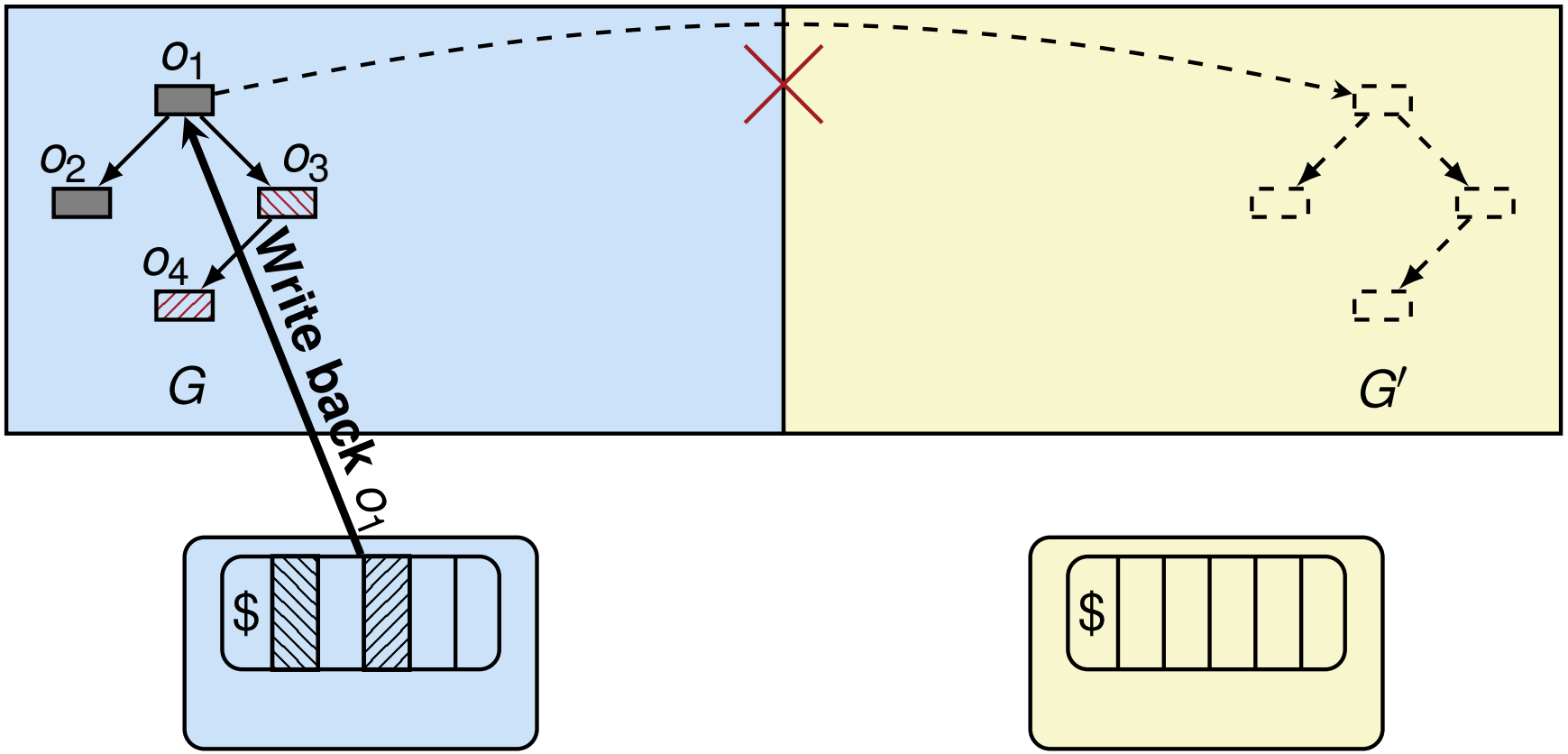






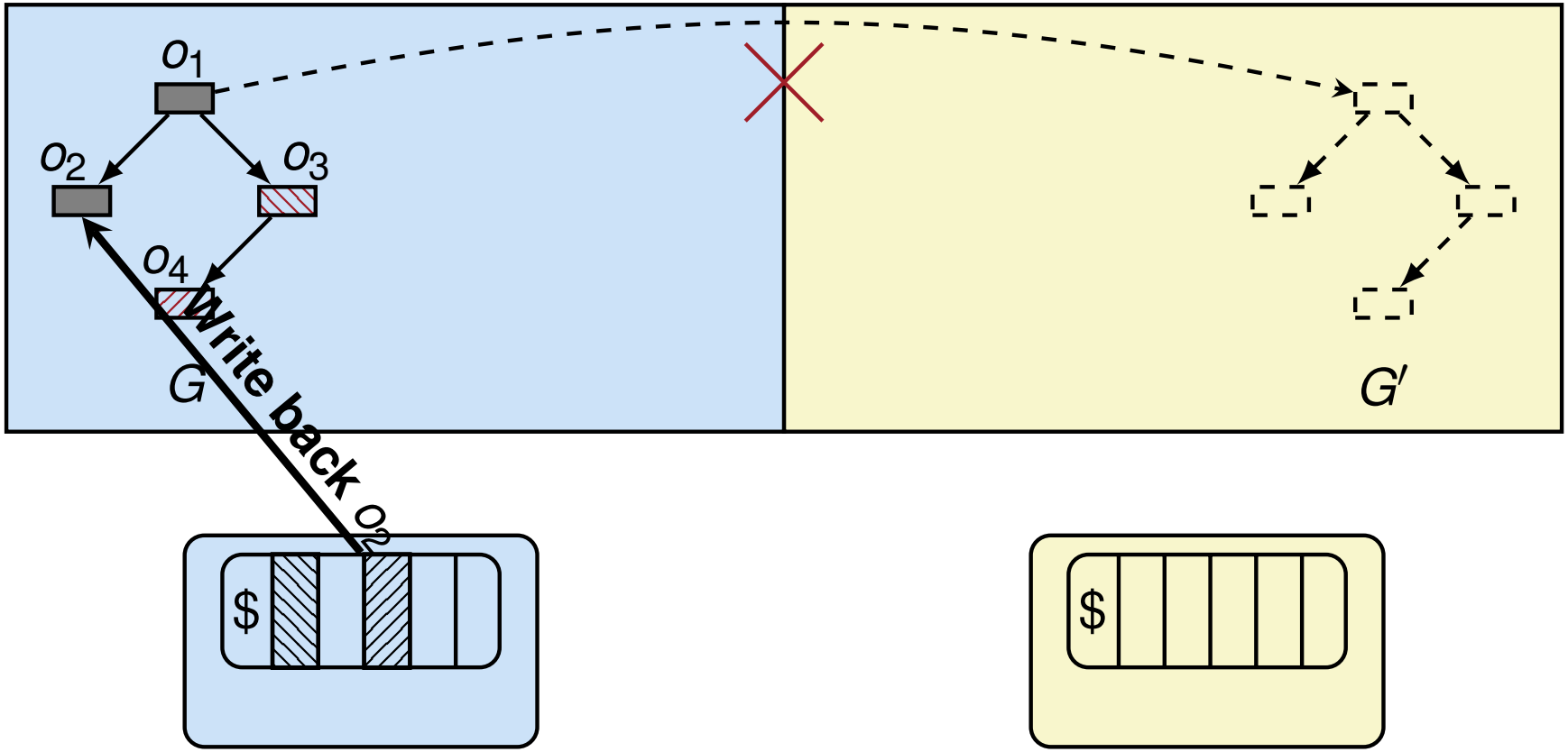
1. Traverse G and write back o_i

Cloning (CLONE)



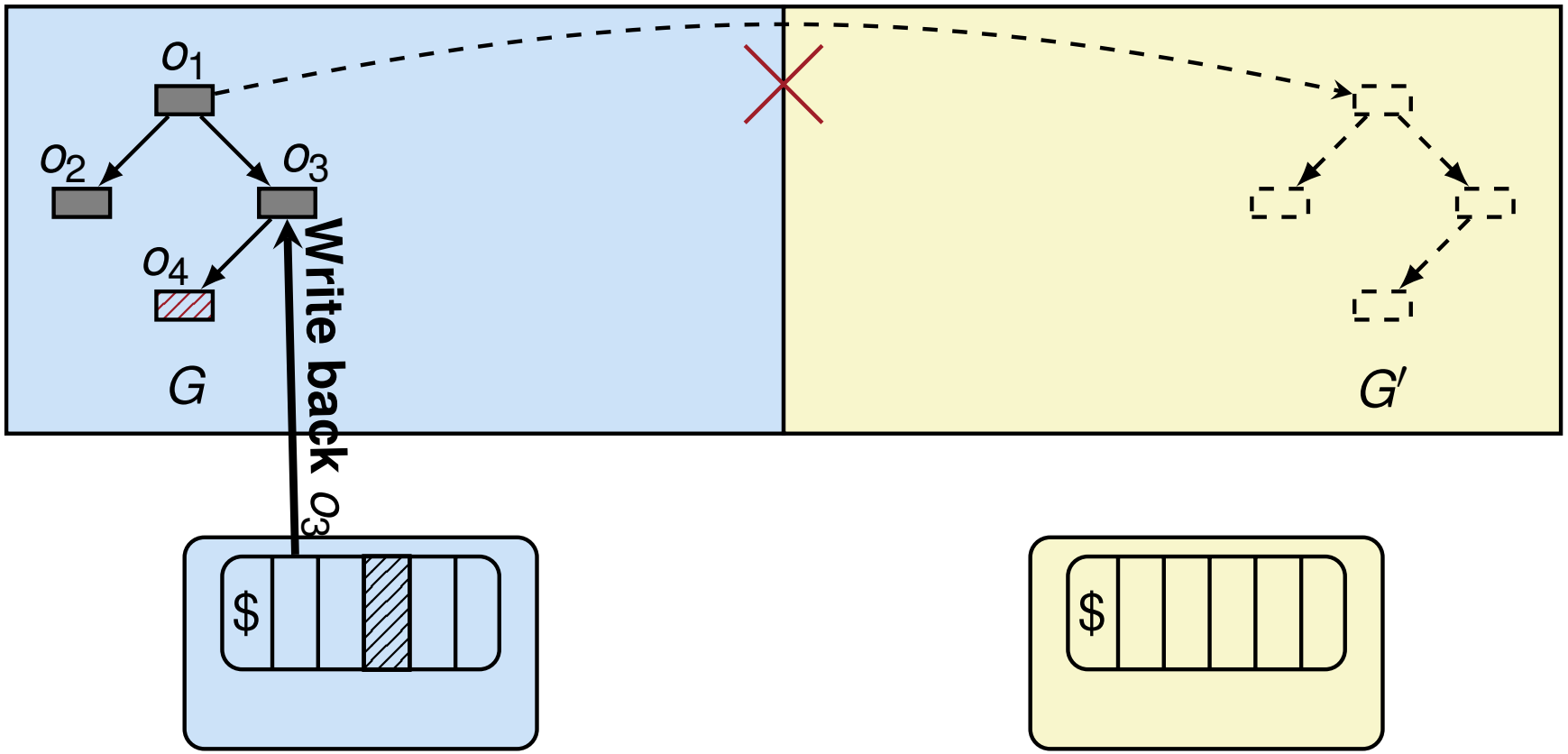
1. Traverse G and write back o_i

Cloning (CLONE)



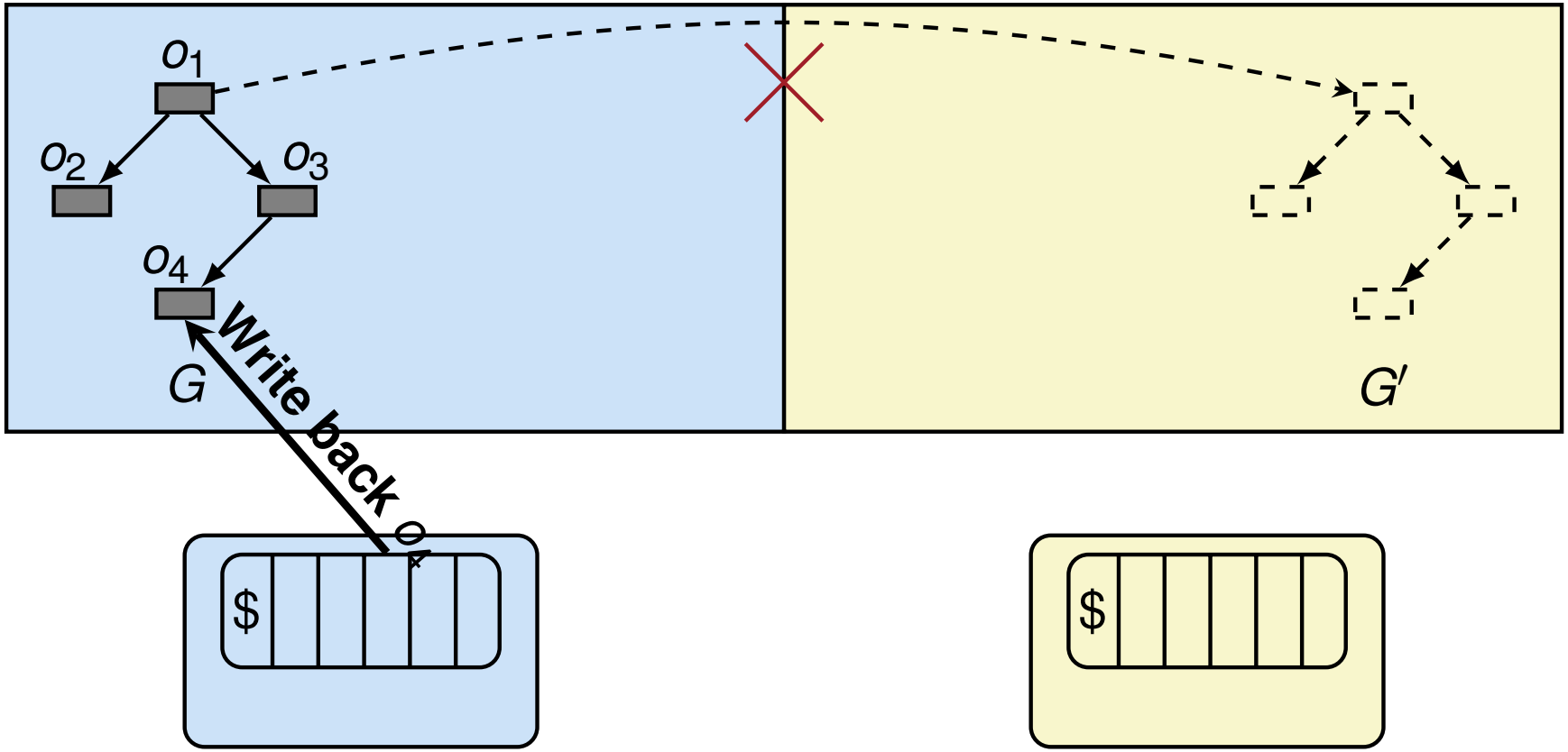
1. Traverse G and write back o_i

Cloning (CLONE)



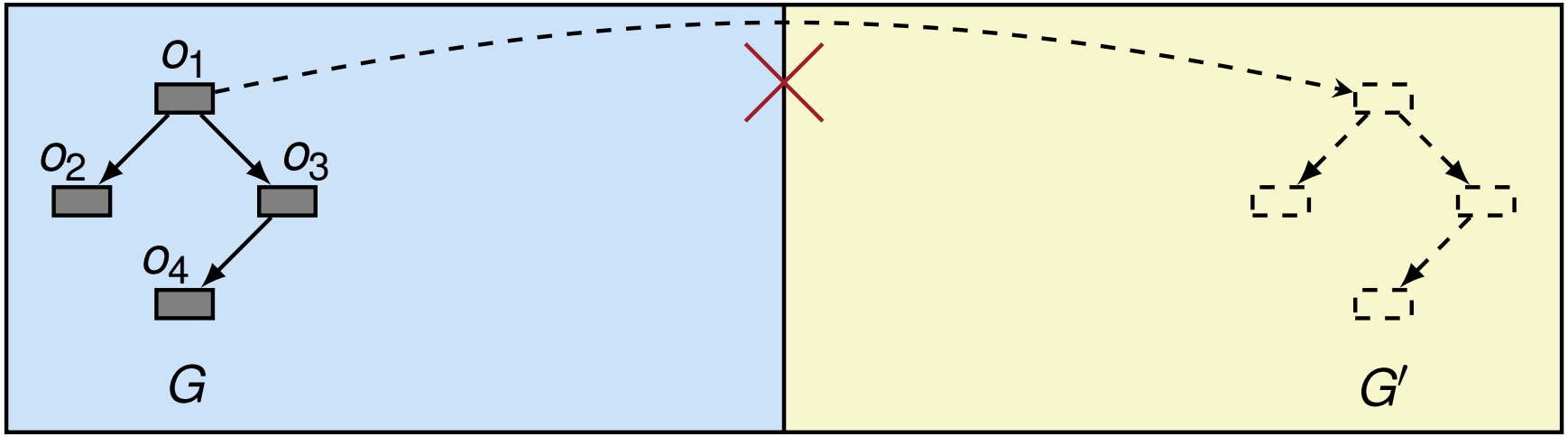
1. Traverse G and write back o_i

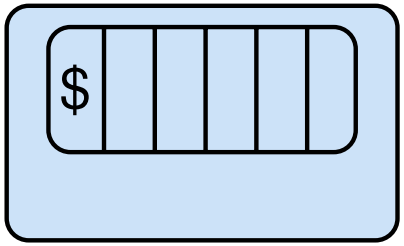
Cloning (CLONE)

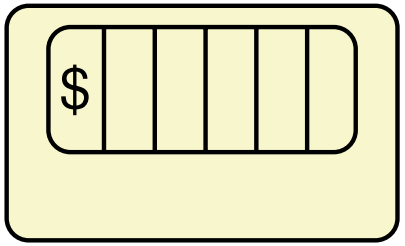


1. Traverse G and write back o_i

Cloning (CLONE)

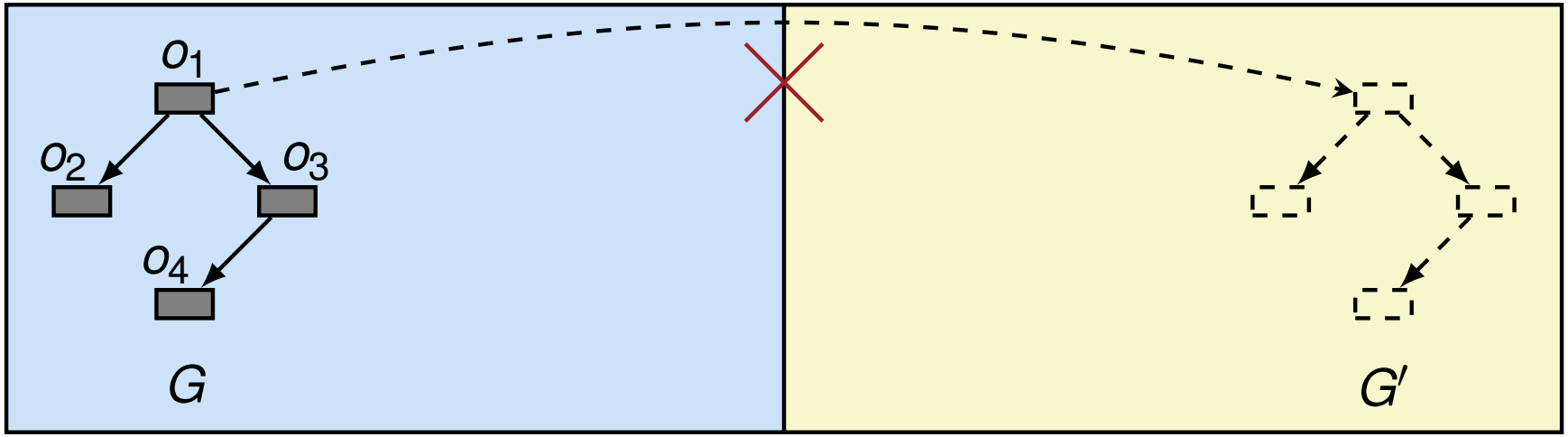


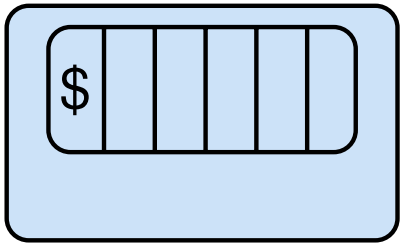


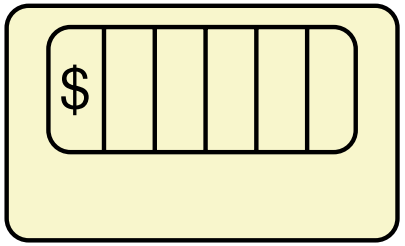


1. Traverse G and write back o_i

Cloning (CLONE)

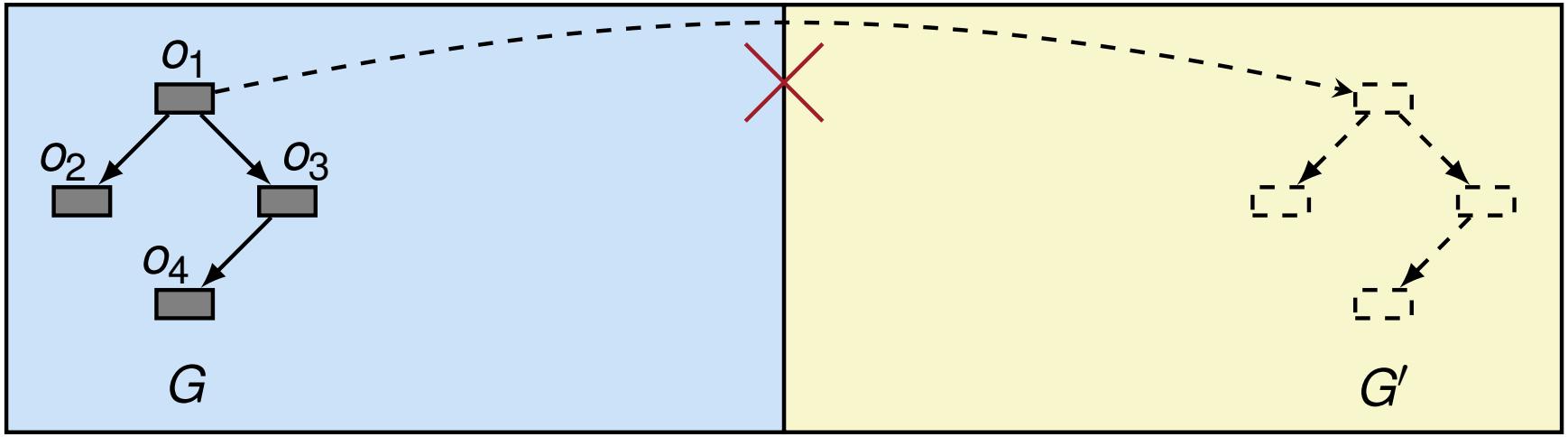


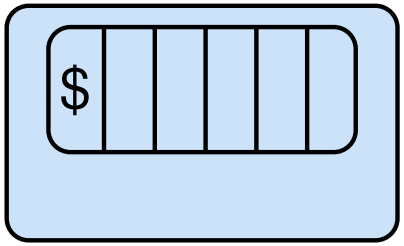


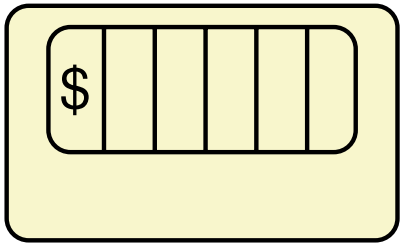


1. Traverse G and write back o_i
2. Notify yellow core with G's root

Cloning (CLONE)

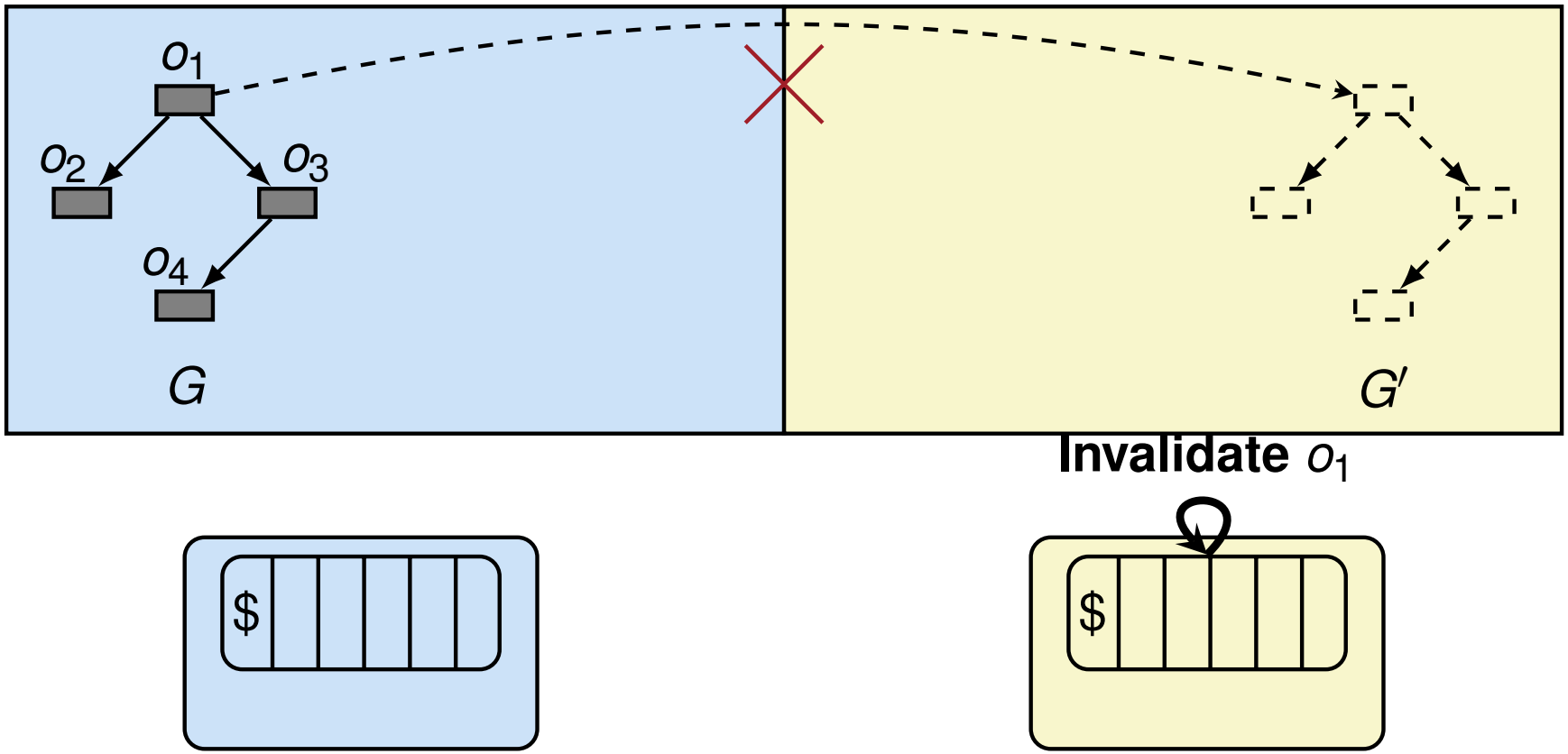






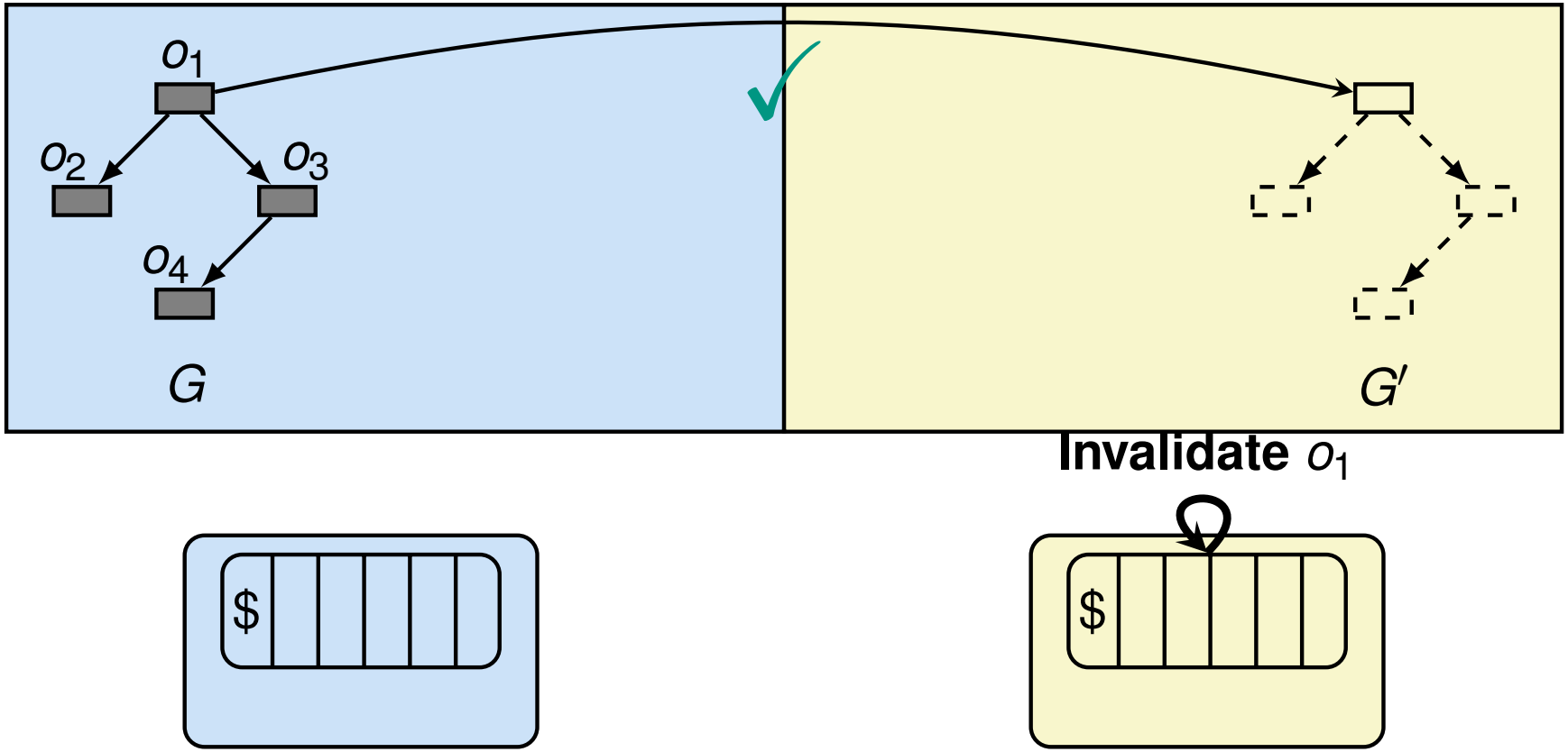
1. Traverse G and write back o_i
2. Notify yellow core with G's root
3. Traverse G & clone objects

Cloning (CLONE)



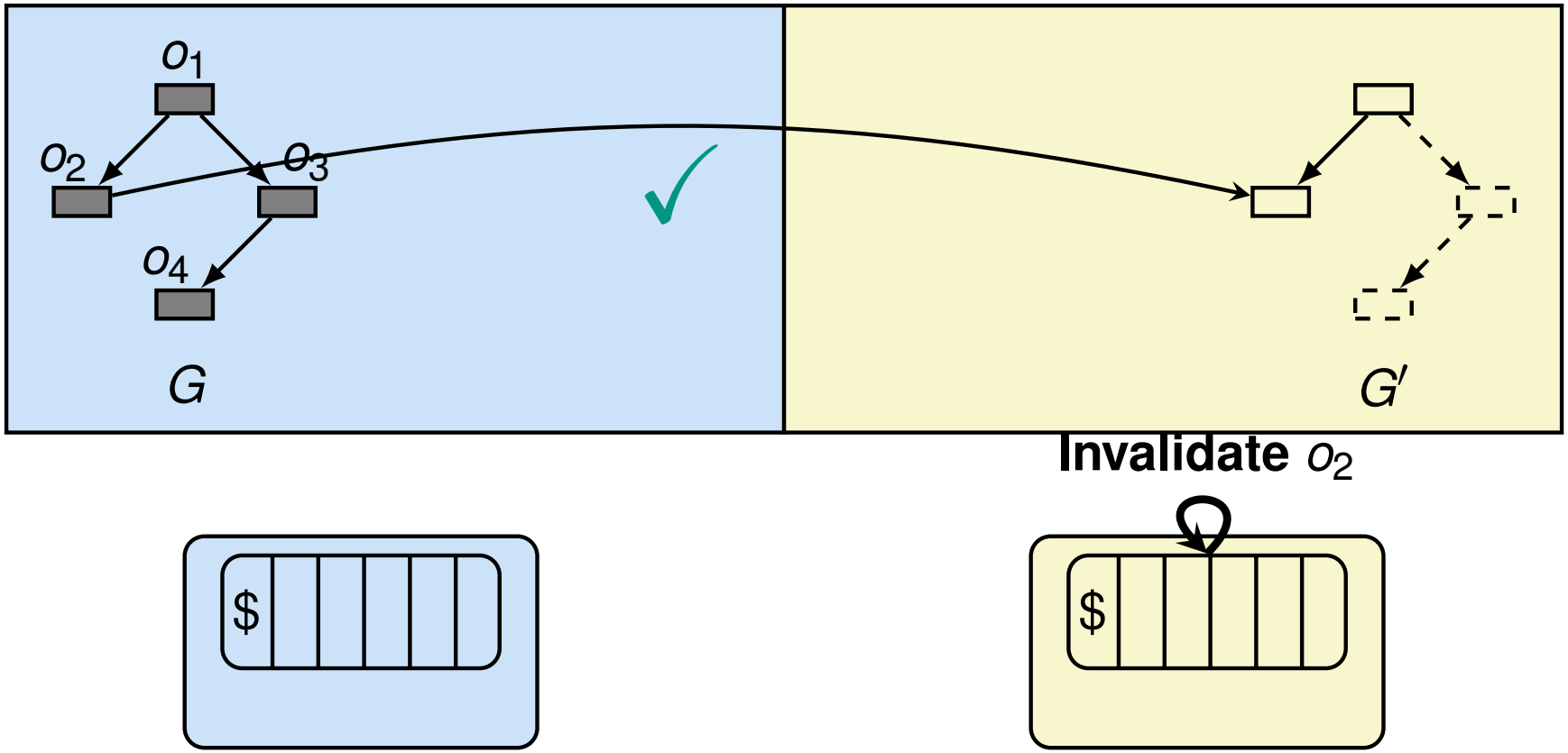
1. Traverse G and write back o_i
2. Notify yellow core with G's root
3. Traverse G, invalidate & clone objects

Cloning (CLONE)



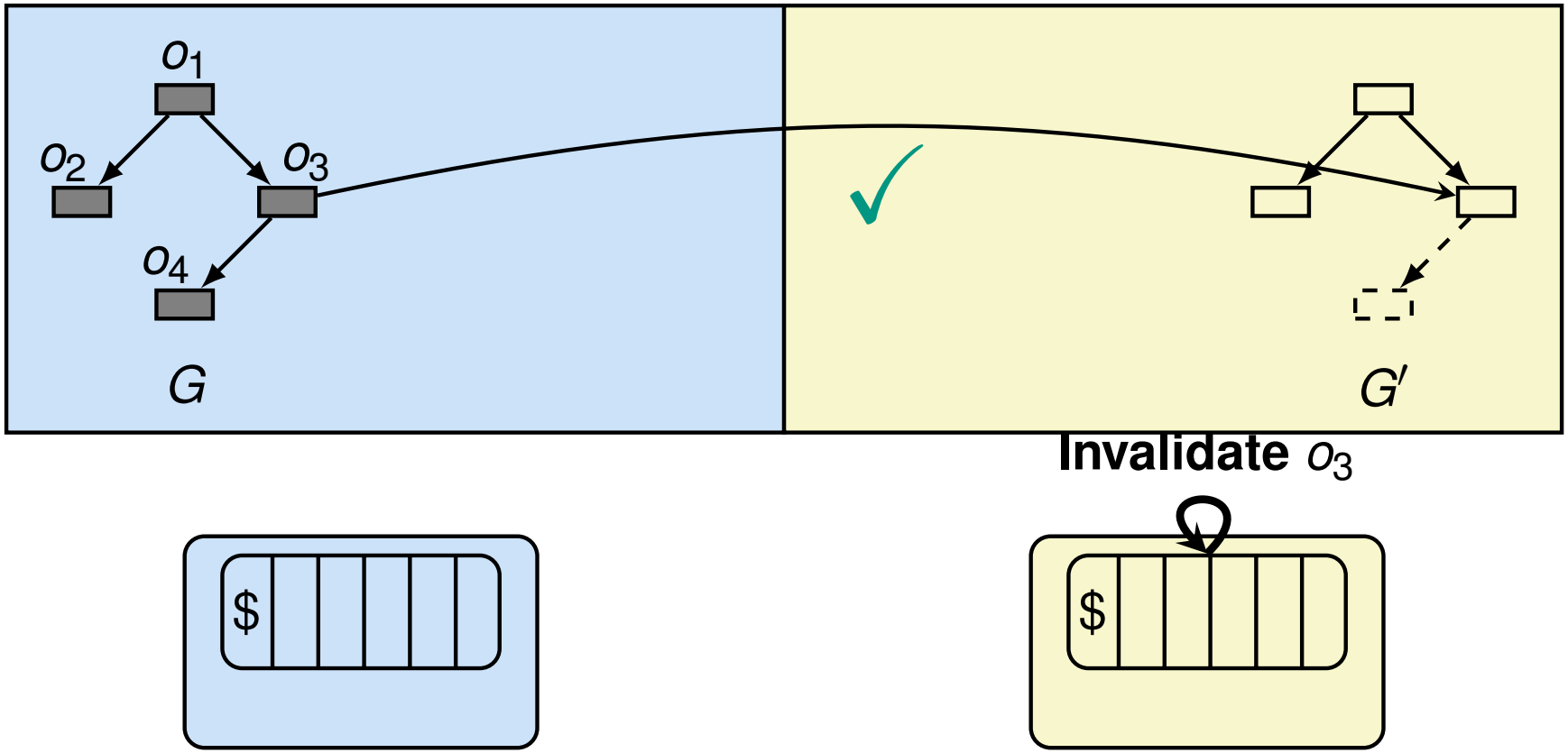
1. Traverse G and write back o_i
2. Notify yellow core with G's root
3. Traverse G, invalidate & clone objects

Cloning (CLONE)



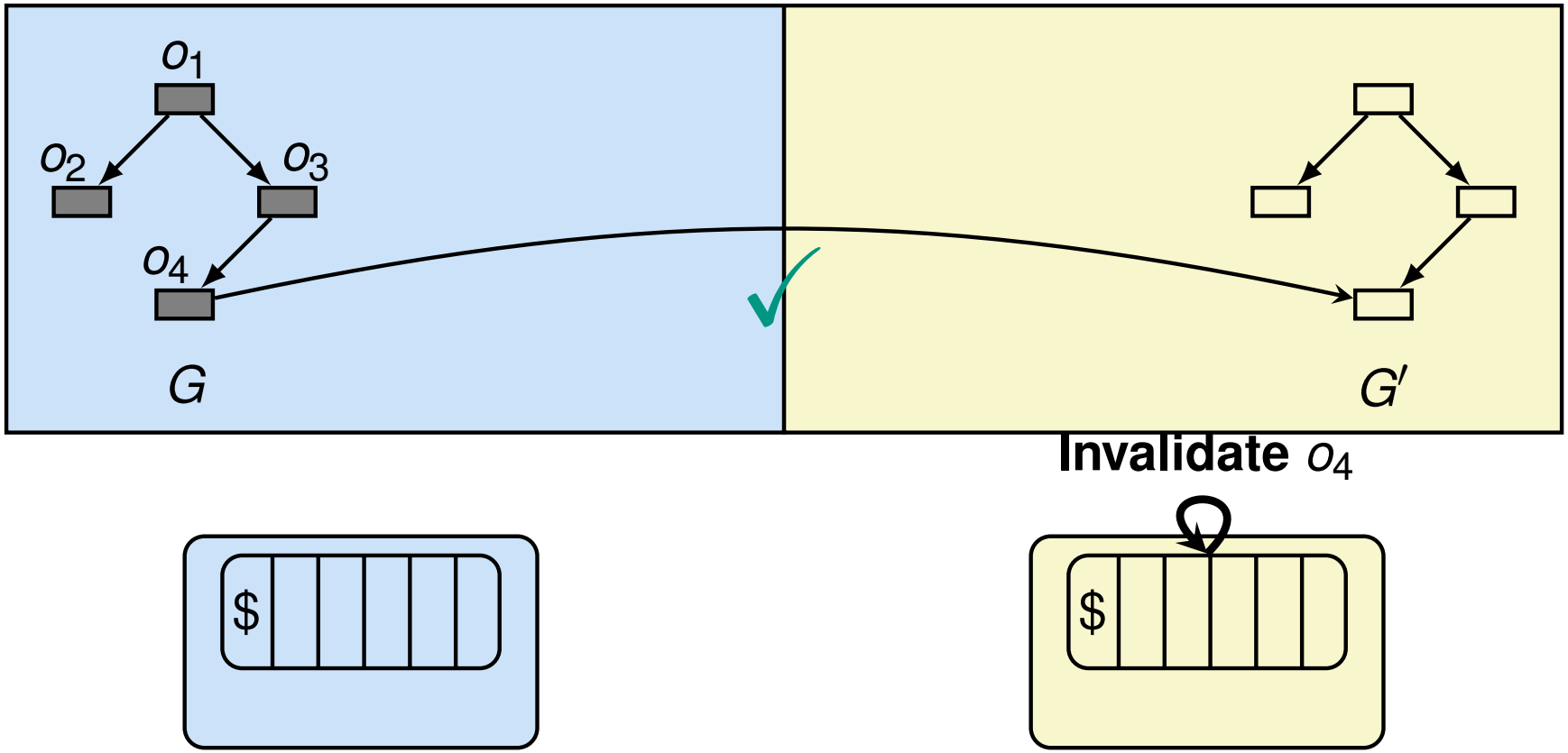
1. Traverse *G* and write back o_i
2. Notify yellow core with *G*'s root
3. Traverse *G*, invalidate & clone objects

Cloning (CLONE)



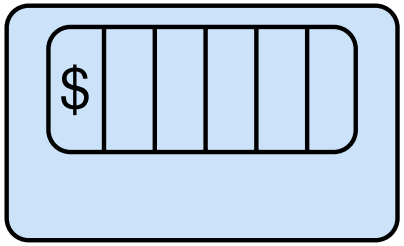
1. Traverse *G* and write back o_i
2. Notify yellow core with *G*'s root
3. Traverse *G*, invalidate & clone objects

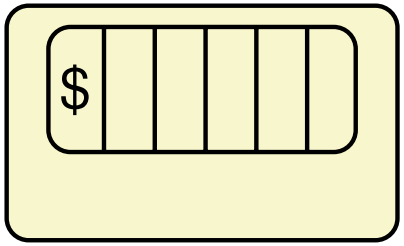
Cloning (CLONE)



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)
```

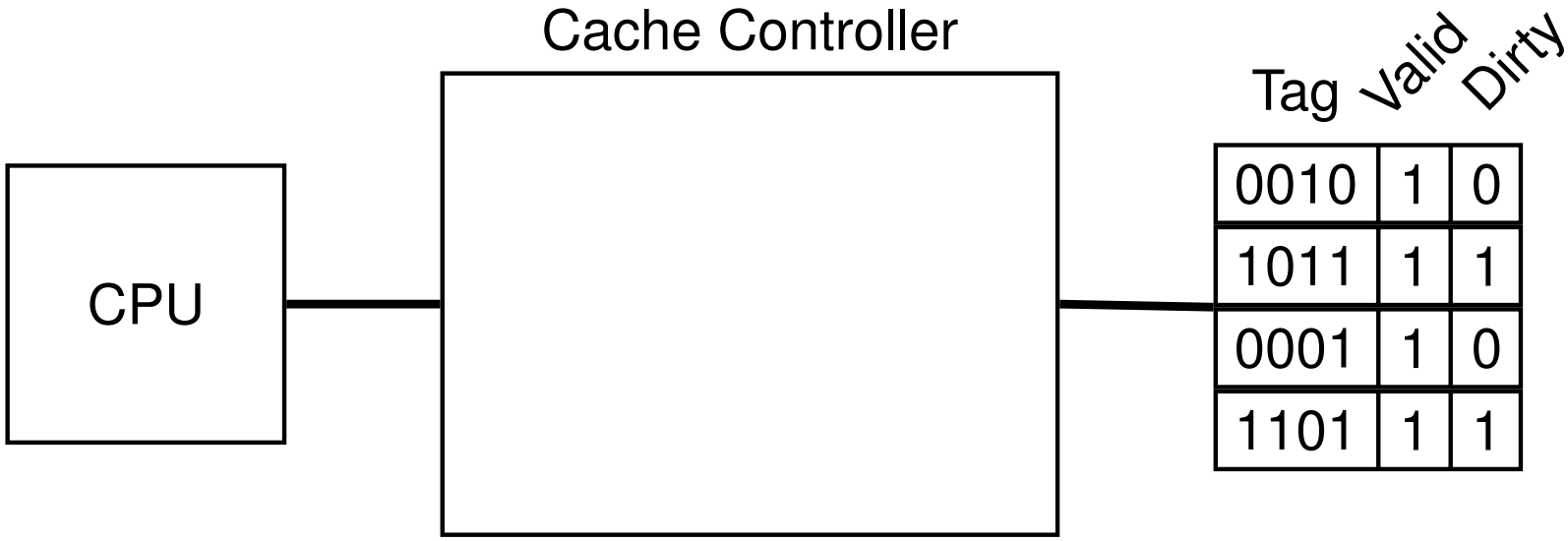
Hardware Extension: Range Operations

- 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?



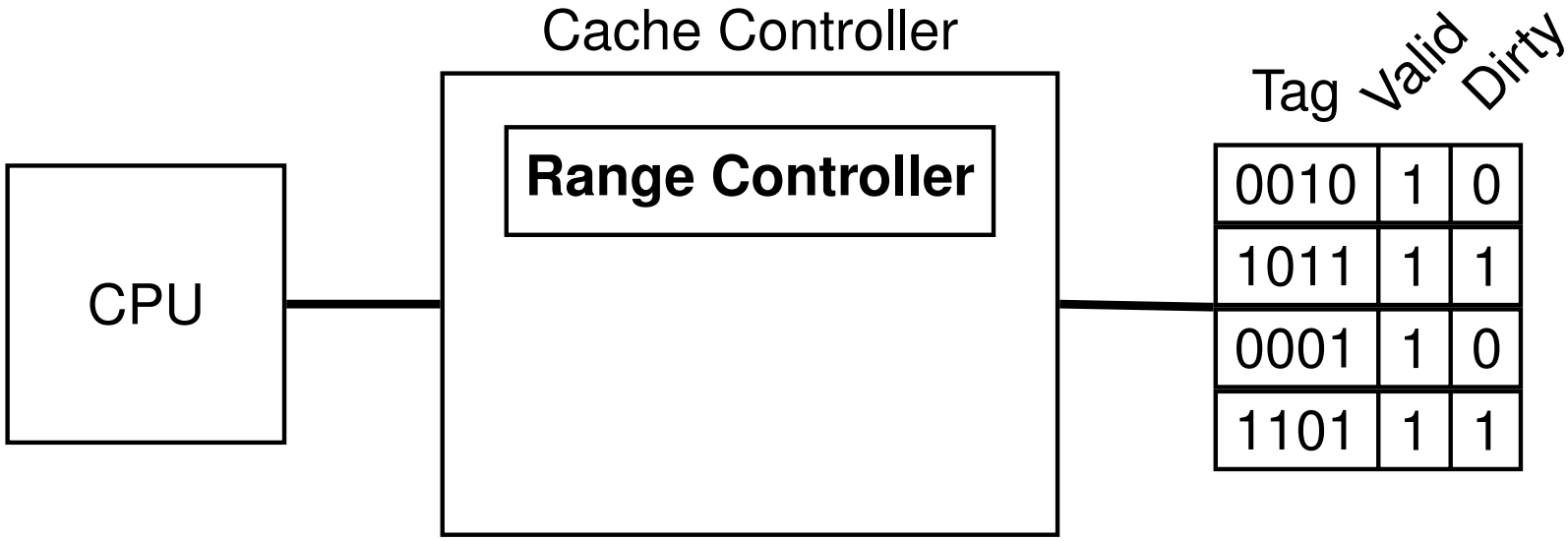
Hardware Extension: Range Operations

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

$\Rightarrow$ Software iterates over all relevant addresses

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

- Why not support this in hardware?



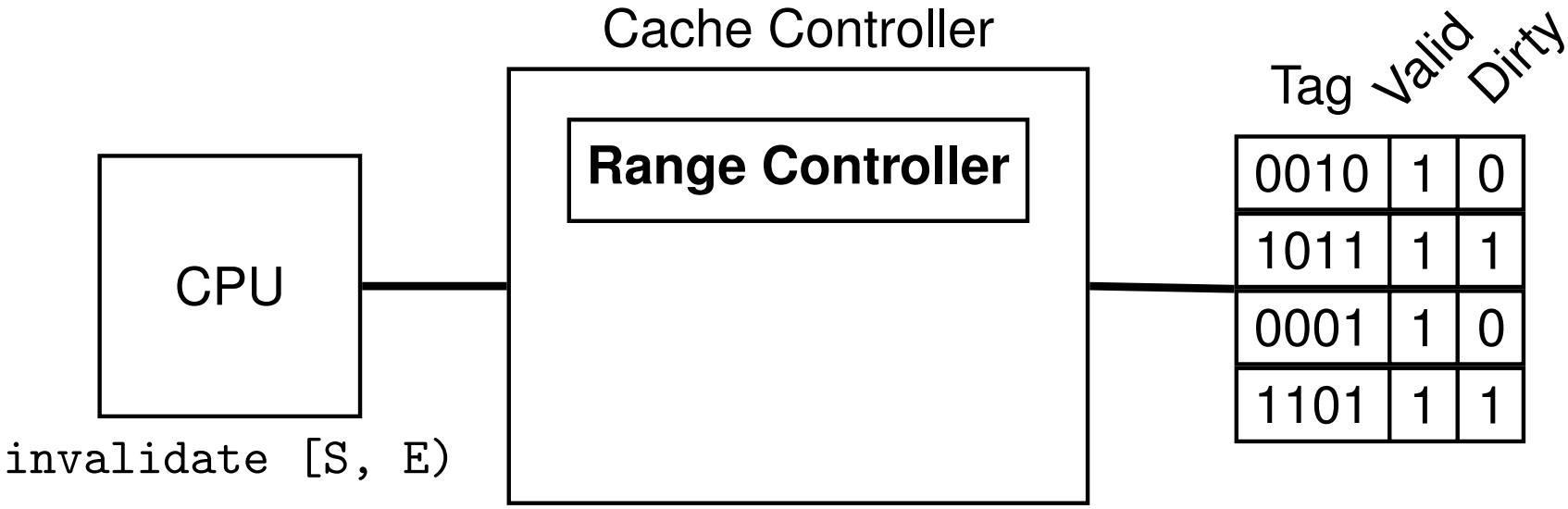
Hardware Extension: Range Operations

- 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?



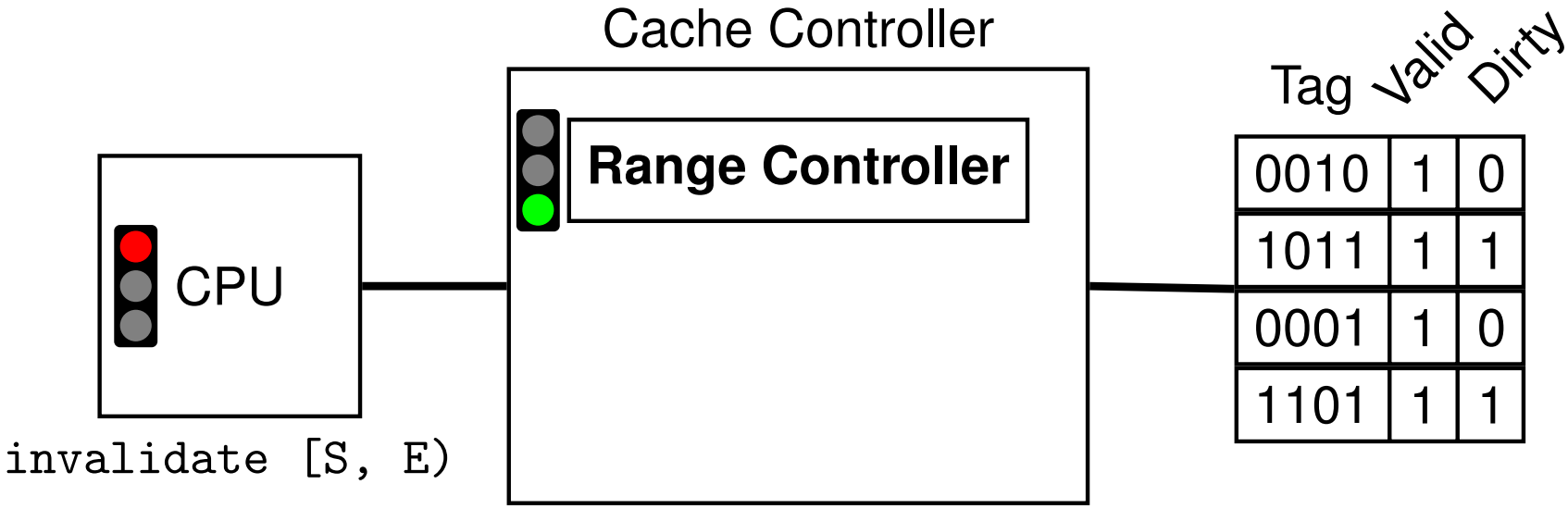
Hardware Extension: Range Operations

- 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?



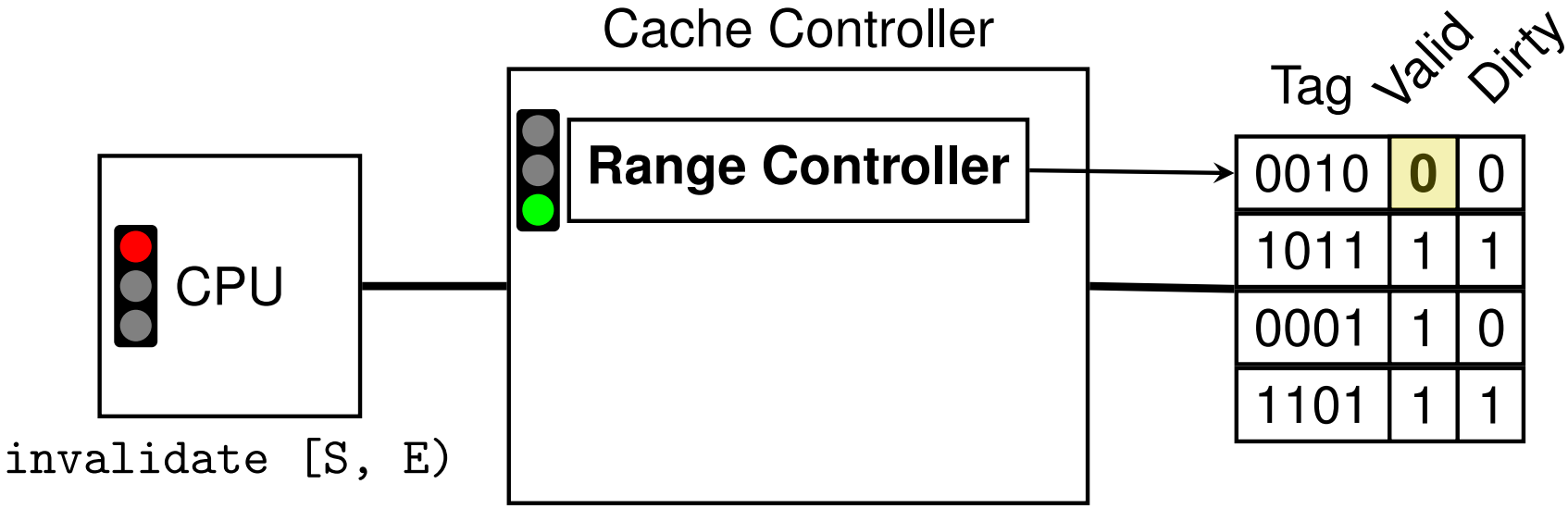
Hardware Extension: Range Operations

- 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?



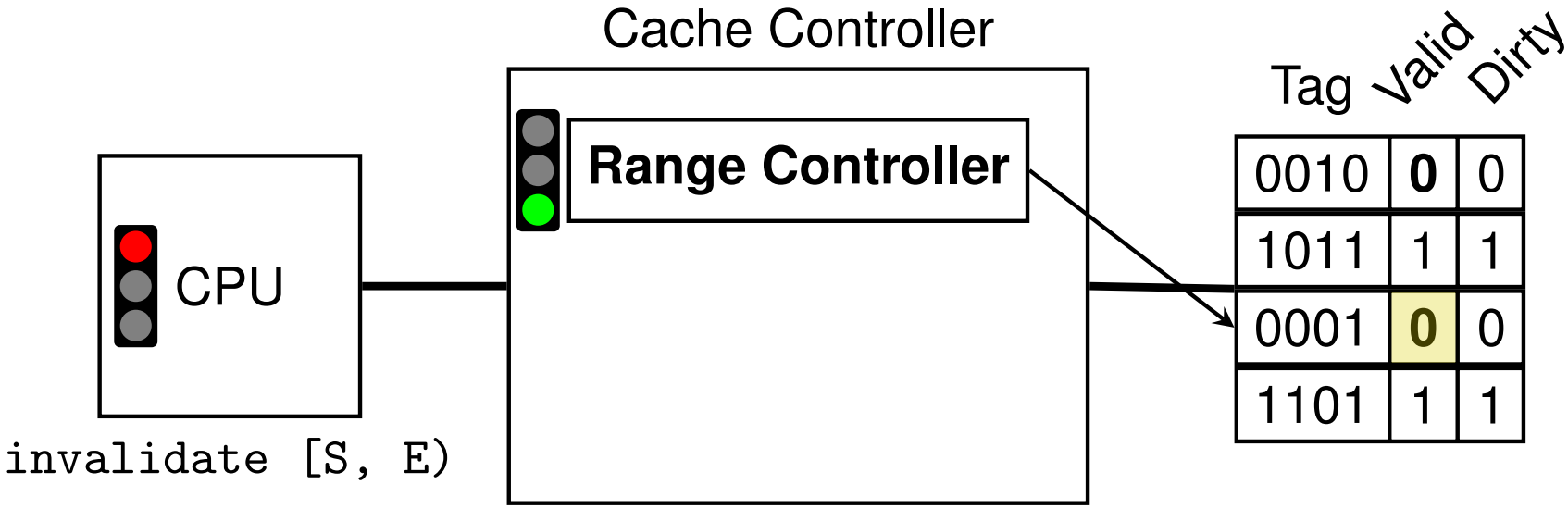
Hardware Extension: Range Operations

- 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?



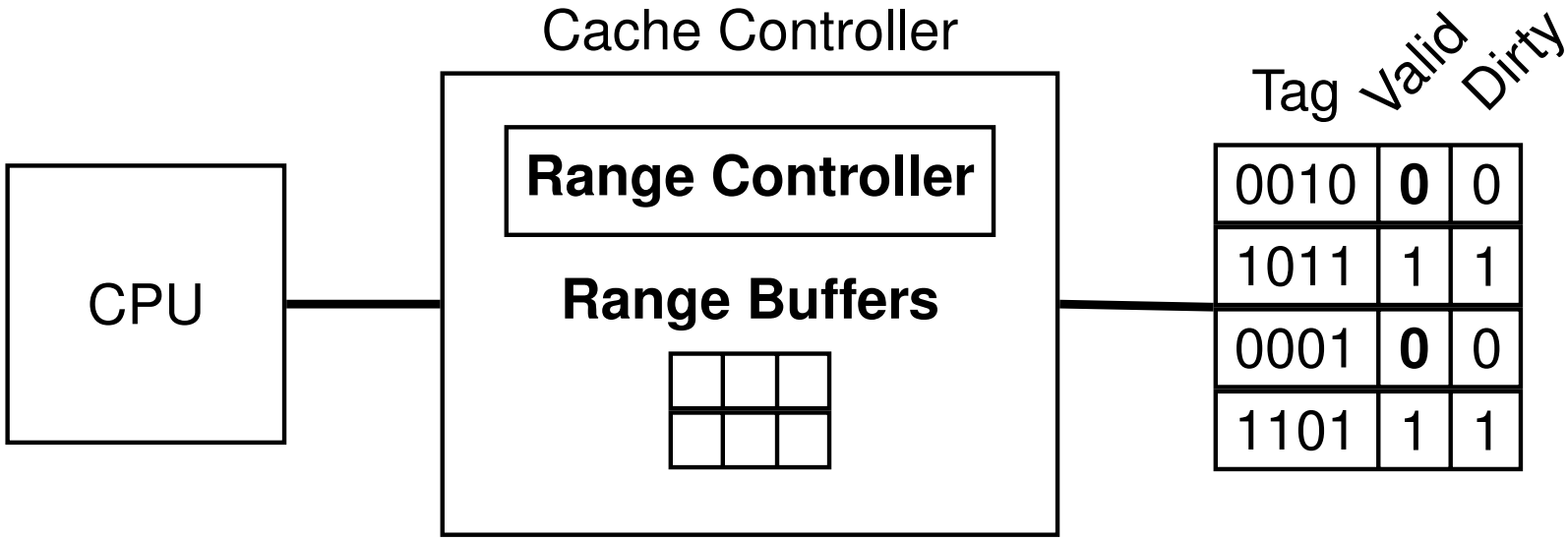
Hardware Extension: Range Operations

- 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?



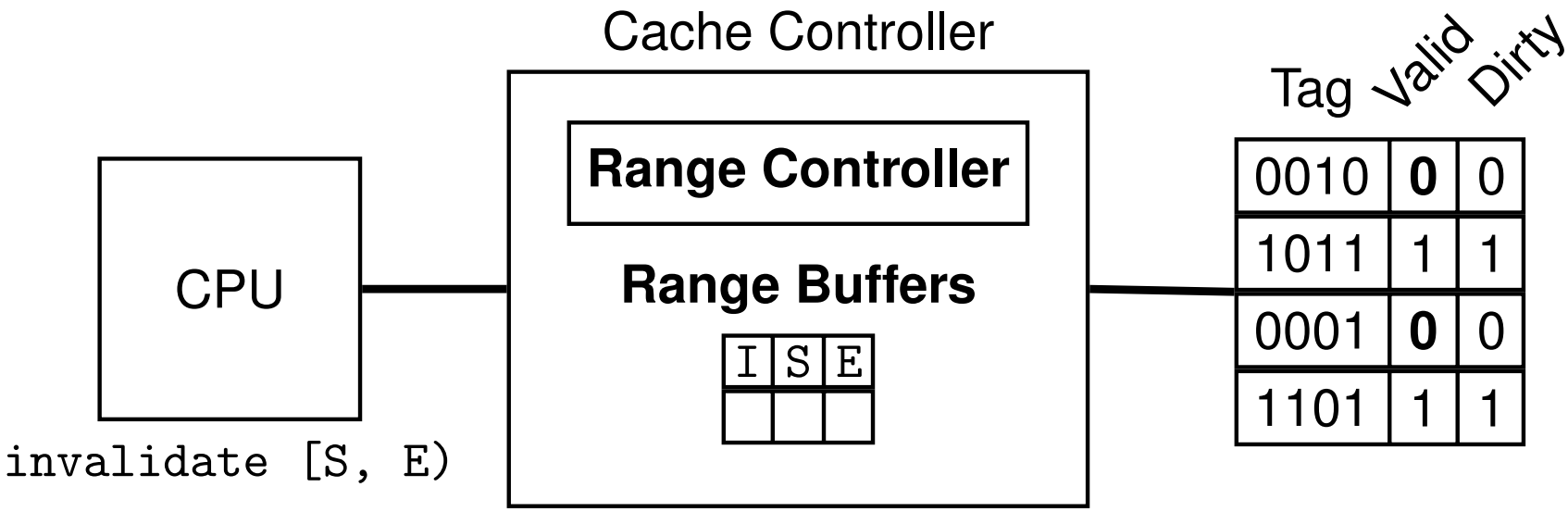
Hardware Extension: Range Operations

- 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?



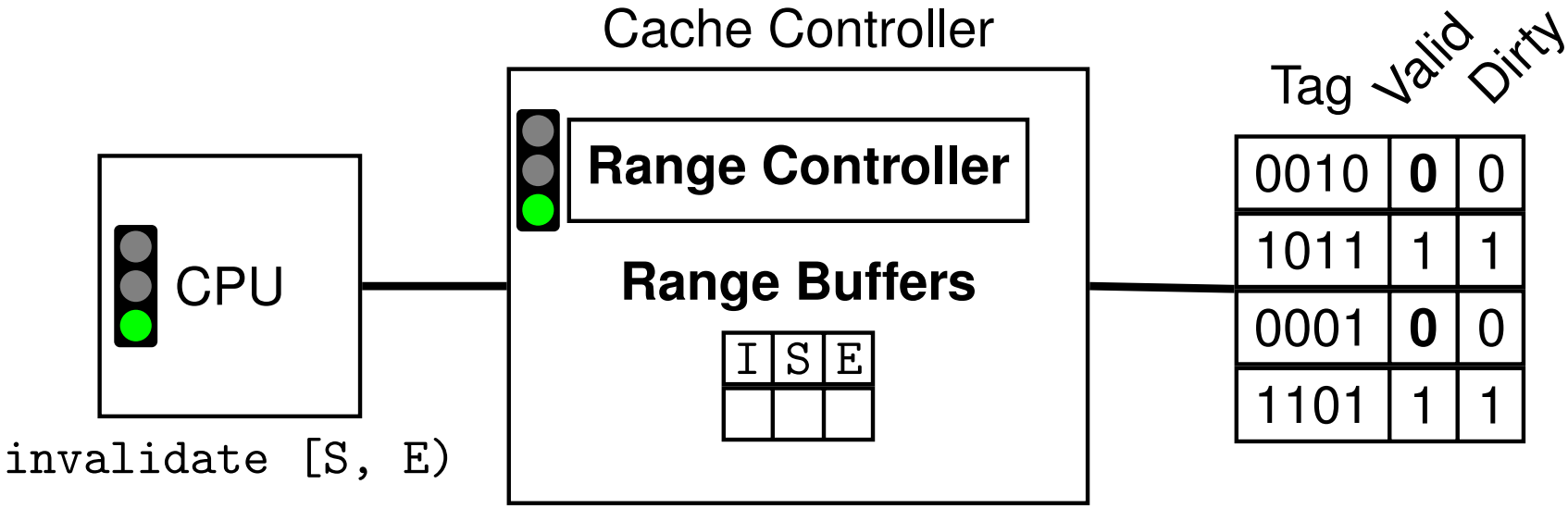
Hardware Extension: Range Operations

- 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?



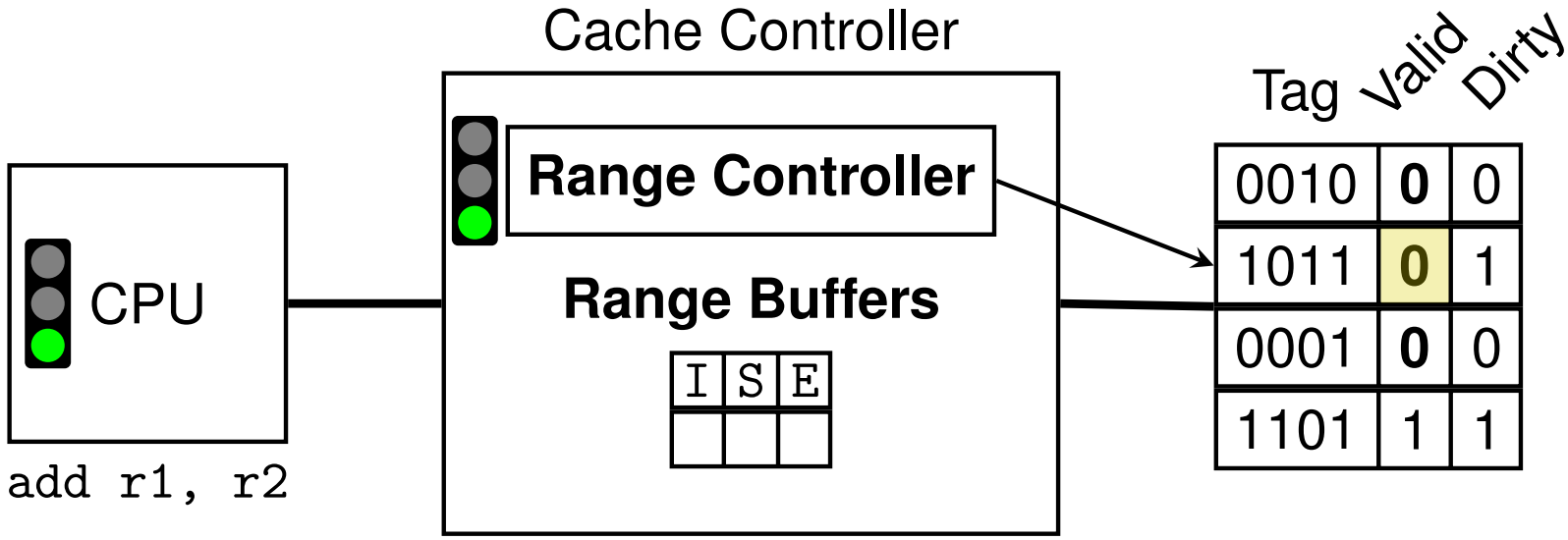
Hardware Extension: Range Operations

- 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?



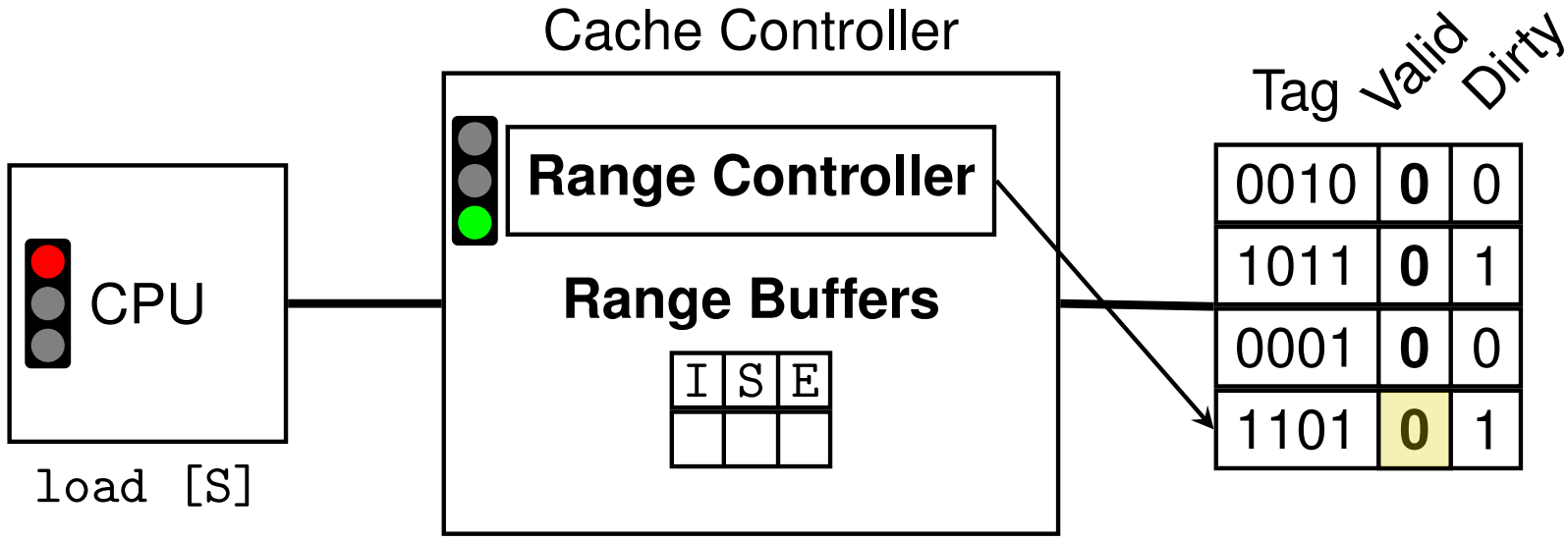
Hardware Extension: Range Operations

- 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)
```

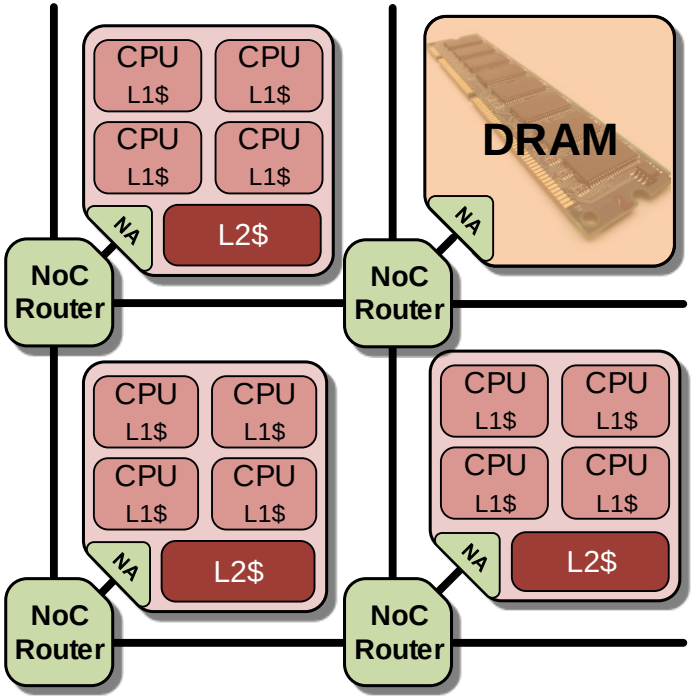
- Why not support this in hardware?



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

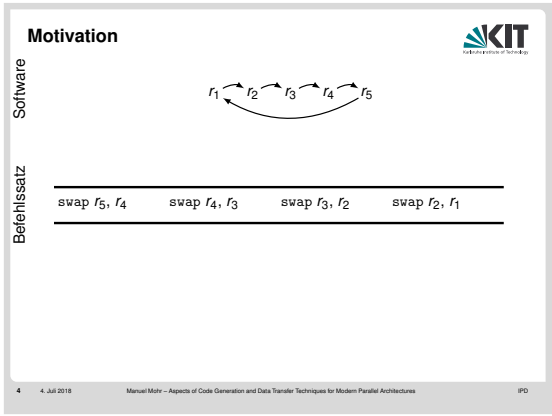
- FPGA-based implementation based on LEON3 cache controller
- One range buffer
- Overhead compared to unmodified cache controller:

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

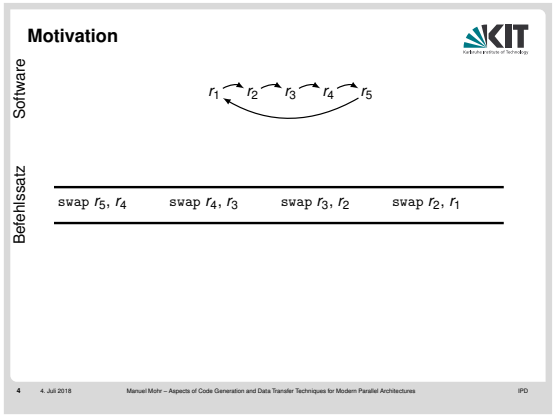
- Data from benchmarks: 17 cache lines on average
- Enough non-memory instructions to cover latency of range operations

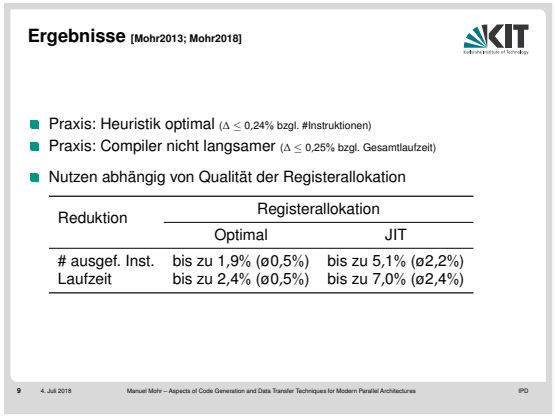
$\Rightarrow$ Expected to take one cycle from view of CPU

Summary

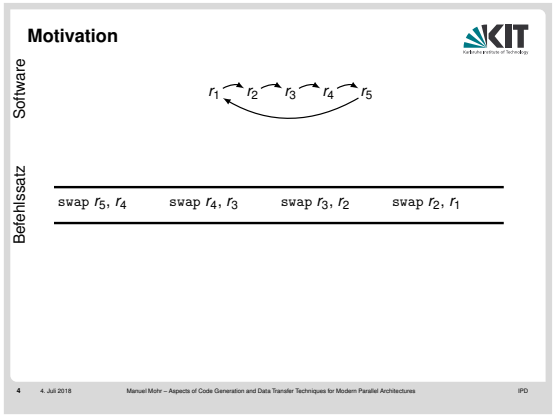

Motivation
Software
Befehlssatz
r_1 r_2 r_3 r_4 r_5
swap r_5, r_4
swap r_4, r_3
swap r_3, r_2
swap r_2, r_1
4 4. Juli 2018 Manuel Mohr – Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures IPD

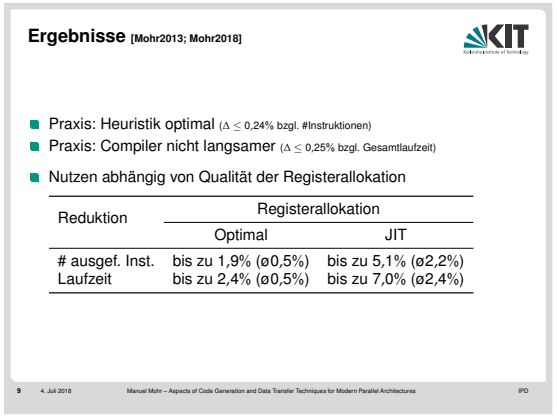
Summary

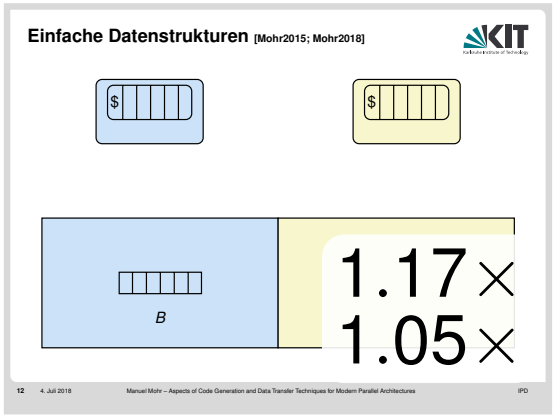

Motivation
Software
Befehlssatz
r_1 r_2 r_3 r_4 r_5
swap r_5, r_4
swap r_4, r_3
swap r_3, r_2
swap r_2, r_1
4 4. Juli 2018 Manuel Mohr – Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures IPD


Ergebnisse [Mohr2013; Mohr2018]
Praxis: Heuristik optimal ($\Delta \leq$ 0,24% bzgl. #Instruktionen)
Praxis: Compiler nicht langsamer ($\Delta \leq$ 0,25% bzgl. Gesamtlaufzeit)
Nutzen abhängig von Qualität der Registerallokation
Reduktion
Registerallokation
Optimal
JIT
ausgef. Inst.
bis zu 1,9% (ø0,5%)
bis zu 5,1% (ø2,2%)
Laufzeit
bis zu 2,4% (ø0,5%)
bis zu 7,0% (ø2,4%)
9 4. Juli 2018 Manuel Mohr – Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures IPD

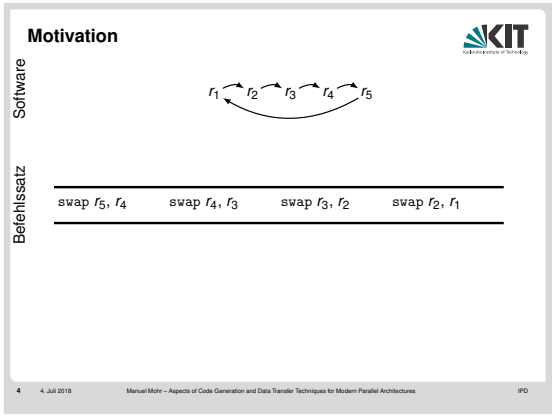
Summary

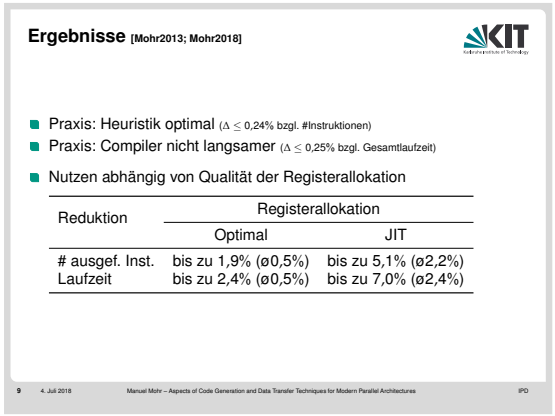

Motivation
Software
Befehlssatz
r_1 r_2 r_3 r_4 r_5
swap r_5, r_4
swap r_4, r_3
swap r_3, r_2
swap r_2, r_1
4 4. Juli 2018 Manuel Mohr – Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures IPD

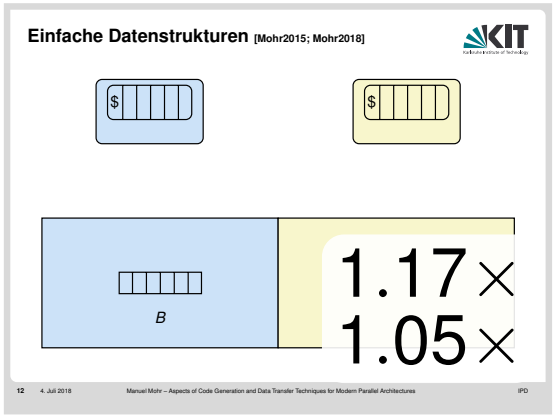

Ergebnisse [Mohr2013; Mohr2018]
Praxis: Heuristik optimal (Δ ≤ 0,24% bzgl. #Instruktionen)
Praxis: Compiler nicht langsamer (Δ ≤ 0,25% bzgl. Gesamtlaufzeit)
Nutzen abhängig von Qualität der Registerallokation
Reduktion
Registerallokation
Optimal
JIT
ausgef. Inst.
bis zu 1,9% (ø0,5%)
bis zu 5,1% (ø2,2%)
Laufzeit
bis zu 2,4% (ø0,5%)
bis zu 7,0% (ø2,4%)
9 4. Juli 2018 Manuel Mohr – Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures IPD

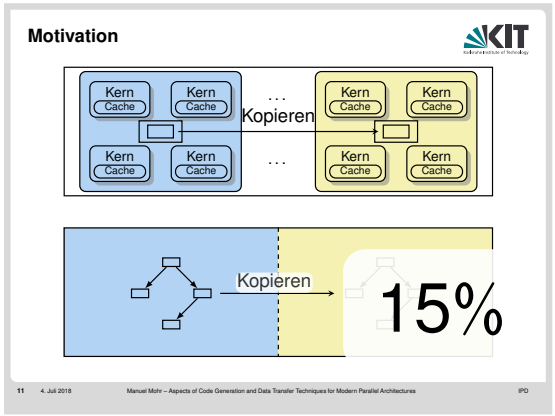

Einfache Datenstrukturen [Mohr2015; Mohr2018]
$
$
B
1.17×
1.05×
12 4. Juli 2018 Manuel Mohr – Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures IPD

Summary

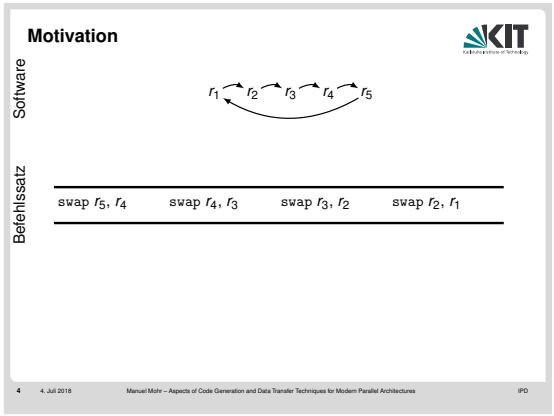

Motivation
Software
Befehlssatz
r_1 r_2 r_3 r_4 r_5
swap r_5, r_4
swap r_4, r_3
swap r_3, r_2
swap r_2, r_1
4 4. Juli 2018 Manuel Mohr – Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures IPD

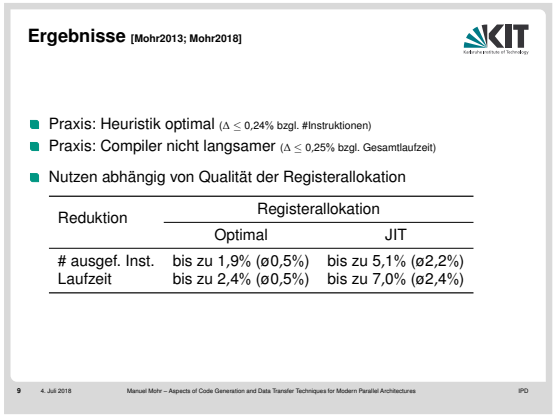

Ergebnisse [Mohr2013; Mohr2018]
Praxis: Heuristik optimal (Δ ≤ 0,24% bzgl. #Instruktionen)
Praxis: Compiler nicht langsamer (Δ ≤ 0,25% bzgl. Gesamtlaufzeit)
Nutzen abhängig von Qualität der Registerallokation
Reduktion
Registerallokation
Optimal
JIT
ausgef. Inst.
bis zu 1,9% (ø0,5%)
bis zu 5,1% (ø2,2%)
Laufzeit
bis zu 2,4% (ø0,5%)
bis zu 7,0% (ø2,4%)
9 4. Juli 2018 Manuel Mohr – Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures IPD

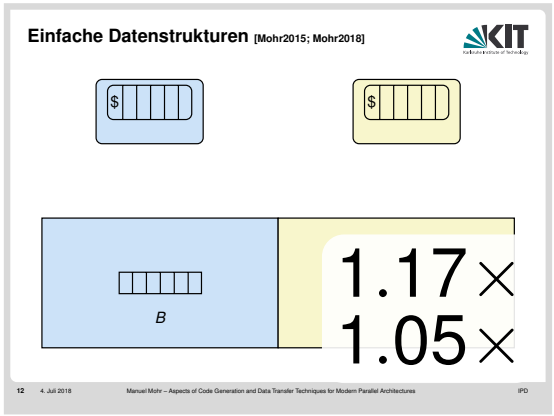

Einfache Datenstrukturen [Mohr2015; Mohr2018]
$
$
B
1.17×
1.05×
12 4. Juli 2018 Manuel Mohr – Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures IPD

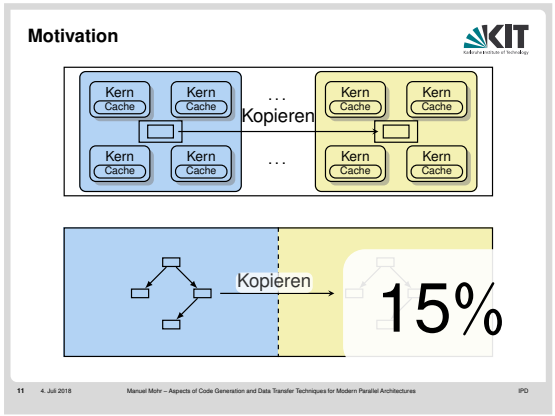

Motivation
Kern
Cache
Kopieren
Kopieren
15%
11 4. Juli 2018 Manuel Mohr – Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures IPD

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

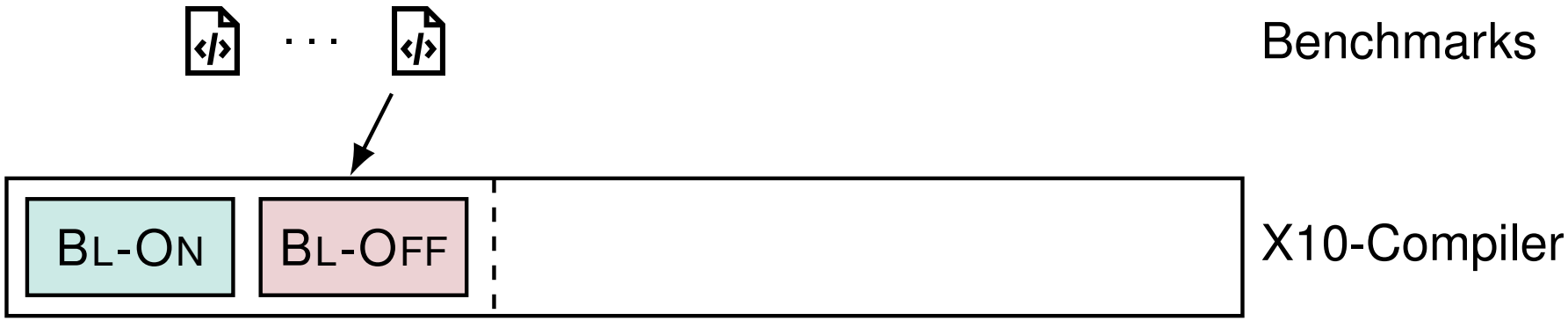
Why benchmarks without hardware extension?

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

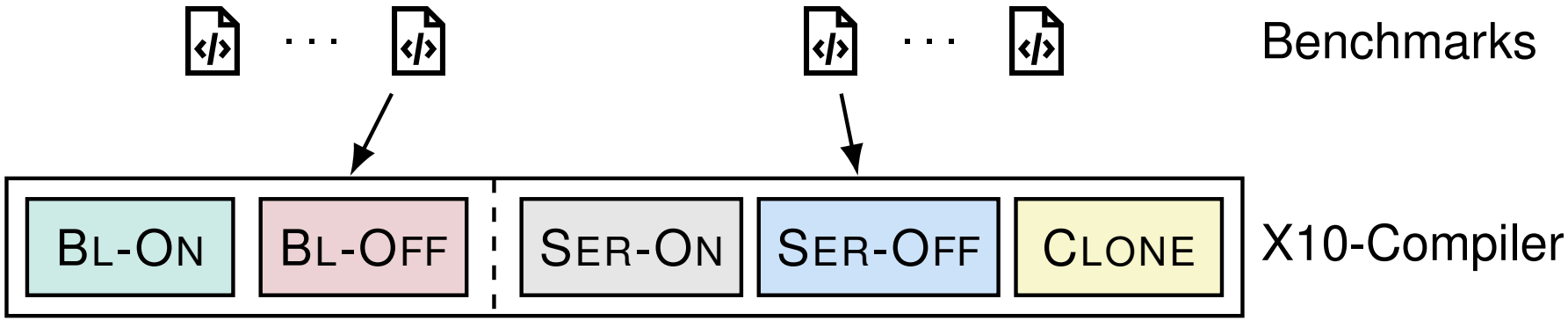
$\Rightarrow$ Redundant work to show feasibility of concept

- Definitely planned for future work

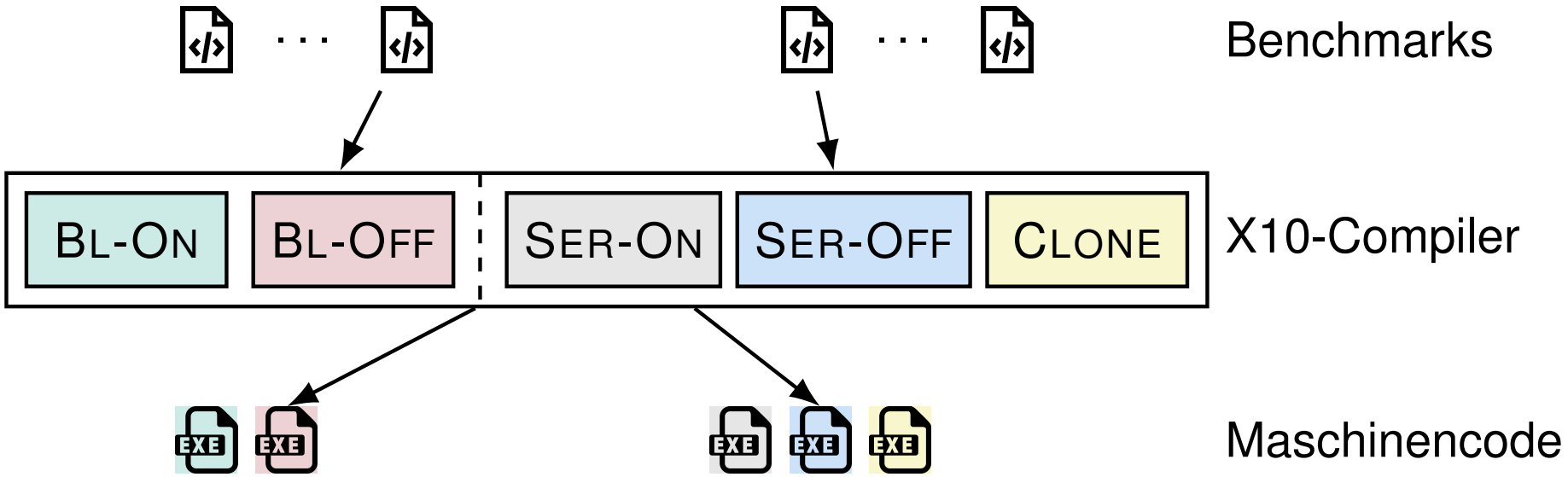
Evaluierungsstrategie



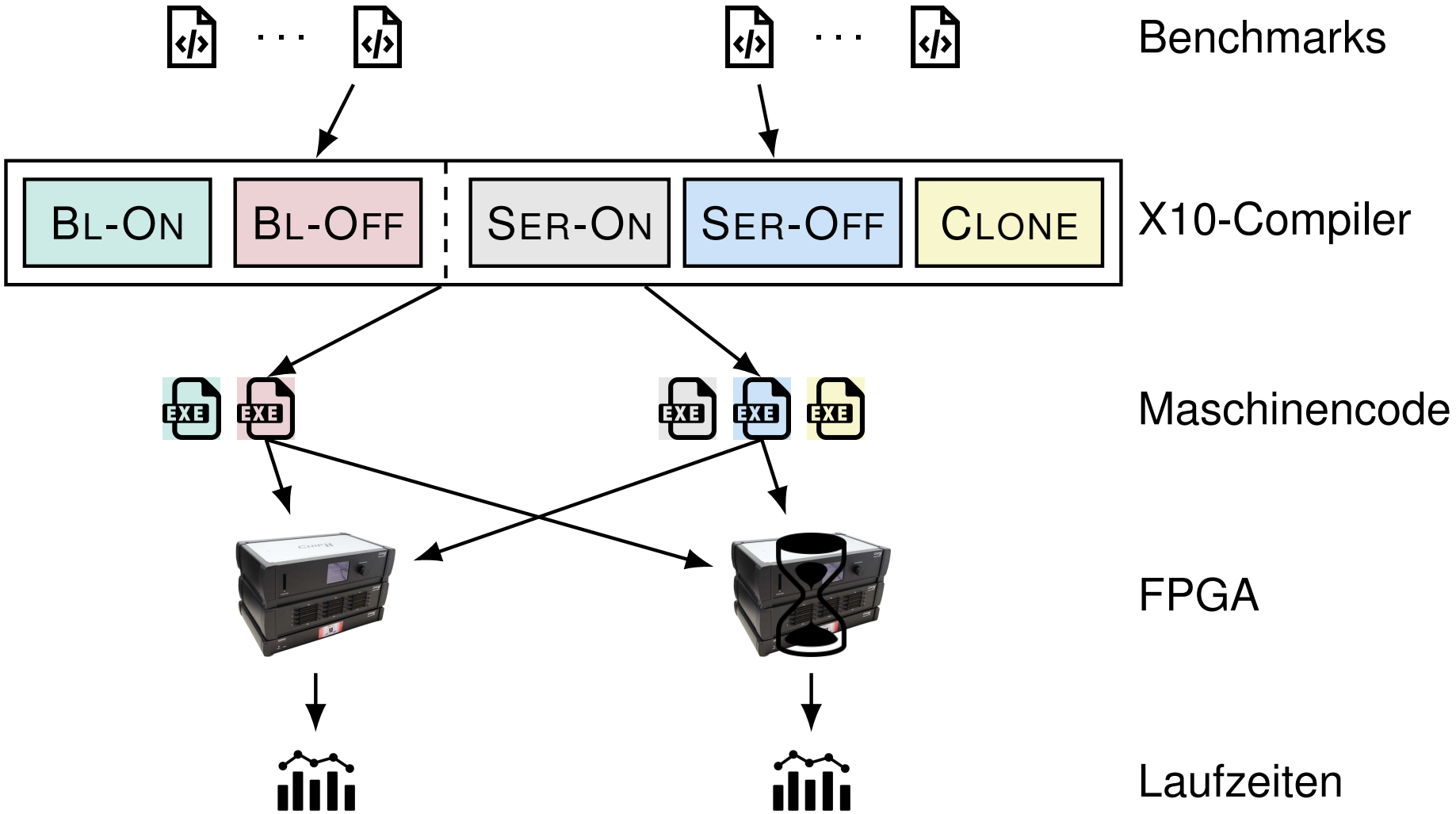
Evaluierungsstrategie



Evaluierungsstrategie



Evaluierungsstrategie



Hardwareunterstützung

- Oft benötigt: Rückschreiben/Invalidieren von Adress*bereichen*

Hardwareunterstützung

- Oft benötigt: Rückschreiben/Invalidieren von Adress*bereichen*
- Peter et al. [Pet+11, Abschnitt IV.B]:

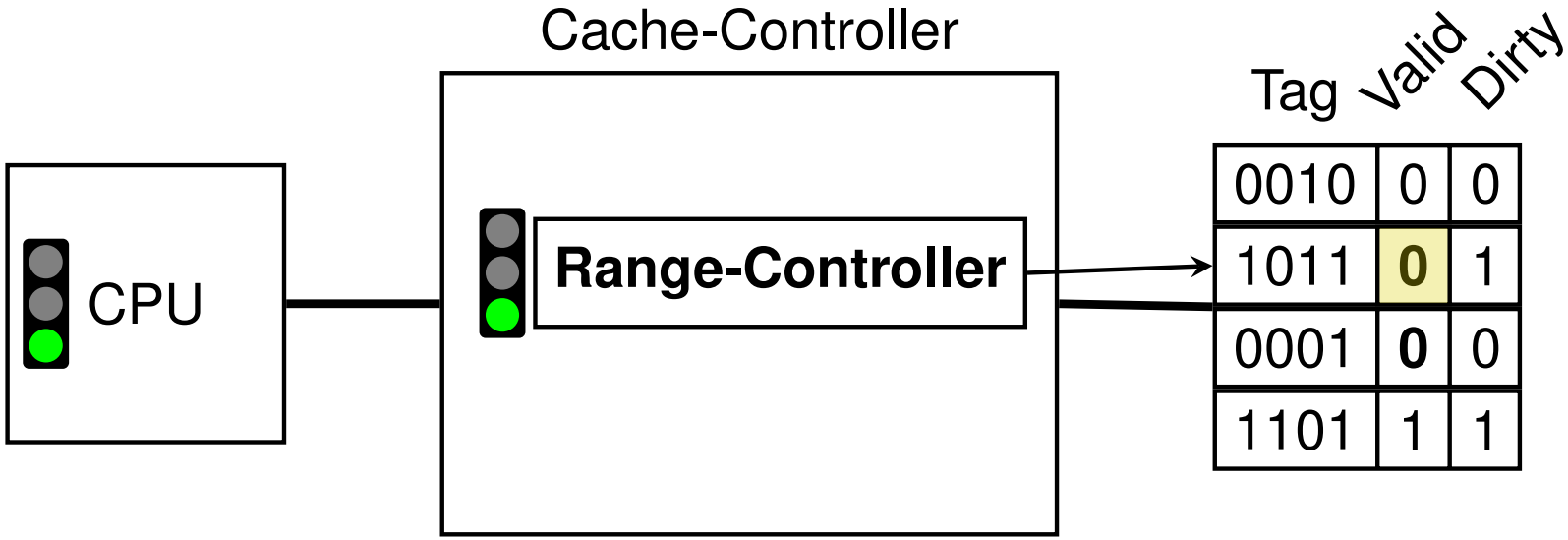
  *An **instruction which would invalidate a region around a given address** would be ideal.*
- Rotta et al. [Rot+12, Abschnitt VI]:

  *The framework would benefit from **write-back and invalidate instructions on logical address ranges**.*
- Christgau et al. [CS16, Abschnitt 8]:

  *It would be beneficial to supply **starting address and size to the invalidation instruction**.*

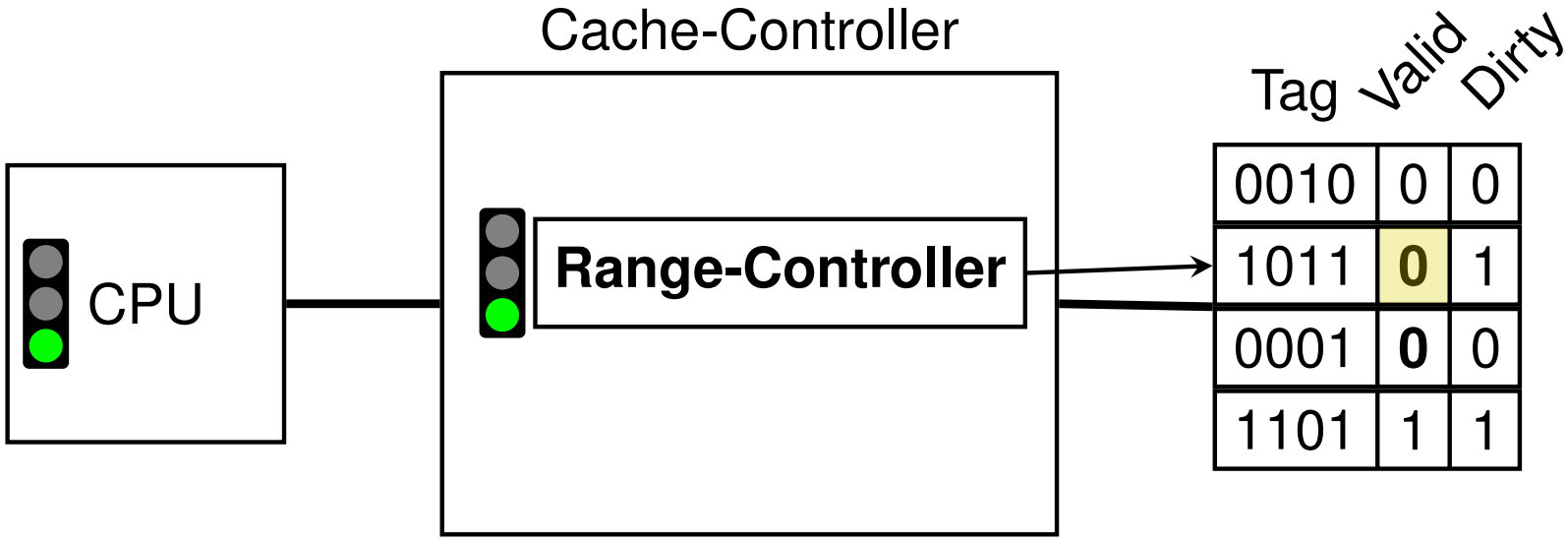
Hardwareunterstützung

- Oft benötigt: Rückschreiben/Invalidieren von Adress*bereichen*



Hardwareunterstützung

- Oft benötigt: Rückschreiben/Invalidieren von Adress*bereichen*



- Fazit: zu hoher Aufwand
- Invalidieren/Rückschreiben einzelner Cachezeilen ausreichend

Literaturverweise I

Sebastian Buchwald, Manuel Mohr und Ignaz Rutter. „Optimal Shuffle Code with Permutation Instructions“. In: *CoRR* abs/1504.07073 (2015). URL: http://arxiv.org/abs/1504.07073.

Sebastian Buchwald, Manuel Mohr und Ignaz Rutter. „Optimal Shuffle Code with Permutation Instructions“. In: *Algorithms and Data Structures*. Hrsg. von Frank Dehne, Jörg-Rüdiger Sack und Ulrike Stege. Bd. 9214. WADS'15. Lecture Notes in Computer Science. Springer International Publishing, 2015, S. 528–541. DOI: 10.1007/978-3-319-21840-3_44.

Literaturverweise II

Steffen Christgau und Bettina Schnor. „Software-Managed Cache Coherence for Fast One-Sided Communication“. In: *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM'16. Barcelona, Spain: ACM, 2016, S. 69–77. ISBN: 978-1-4503-4196-7. DOI: 10.1145/2883404.2883409.

Manuel Mohr, Artjom Grudnitsky, Tobias Modschiedler, Lars Bauer, Sebastian Hack und Jörg Henkel. „Hardware Acceleration for Programs in SSA Form“. In: *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. CASES'13. Piscataway, NJ, USA: IEEE Press, 2013, 14:1–14:10. DOI: 10.1109/CASES.2013.6662518.

Literaturverweise III

Manuel Mohr, Sebastian Buchwald, Andreas Zwinkau, Christoph Erhardt, Benjamin Oechslein, Jens Schedel und Daniel Lohmann. „Cutting out the Middleman: OS-Level Support for X10 Activities“. In: *Proceedings of the ACM SIGPLAN Workshop on X10*. X10'15. Portland, OR, USA: ACM, 2015, S. 13–18. ISBN: 978-1-4503-3586-7. DOI: 10.1145/2771774.2771775.

Manuel Mohr. „Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures“. Submitted. Diss. 2018.

Literaturverweise IV

Manuel Mohr und Carsten Tradowsky. „Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores“. In: *Proceedings of Design, Automation and Test in Europe Conference Exhibition*. DATE'17. IEEE, März 2017, S. 1781–1786. DOI: 10.23919/DATE.2017.7927281.

Simon Peter, Adrian Schüpbach, Dominik Menzi und Timothy Roscoe. „Early Experience with the Barrelfish OS and the Single-Chip Cloud Computer“. In: *MARC Symposium*. 2011, S. 35–39.

Literaturverweise V

Randolf Rotta, Thomas Prescher, Jana Traue und Jörg Nolte. „Data Sharing Mechanisms for Parallel Graph Algorithms on the Intel SCC“. In: *Proceedings of the 6th Many-core Applications Research Community Symposium*. MARC. ONERA, The French Aerospace Lab. 2012, S. 13–18.