Fakultät Informatik, Institut für Technische Informatik, Professur für VLSI - EDA

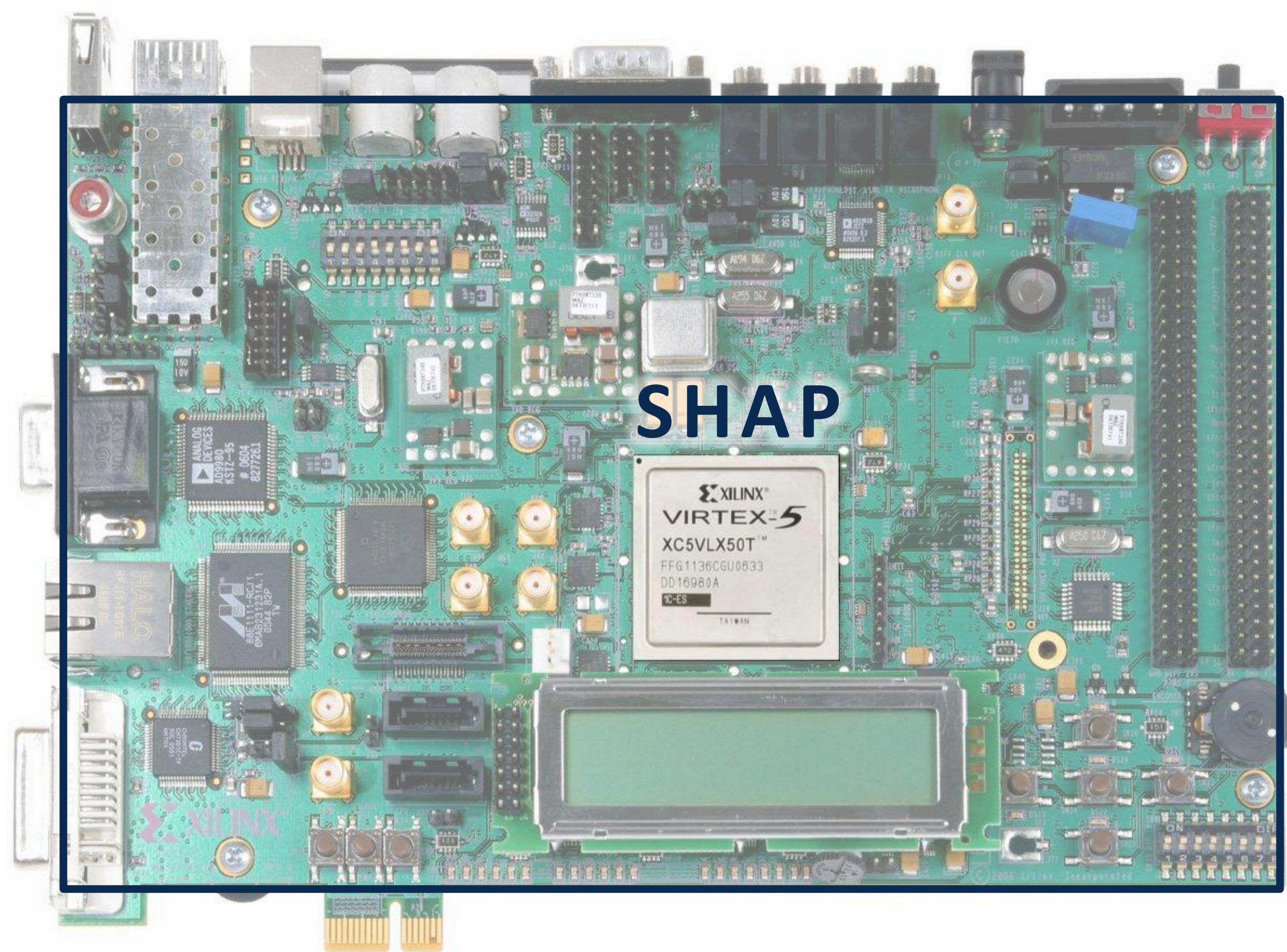
Diplomverteidigung

Implementierung eines Schedulings mit dynamischer Lastverteilung für die SHAP-Mehrkernarchitektur

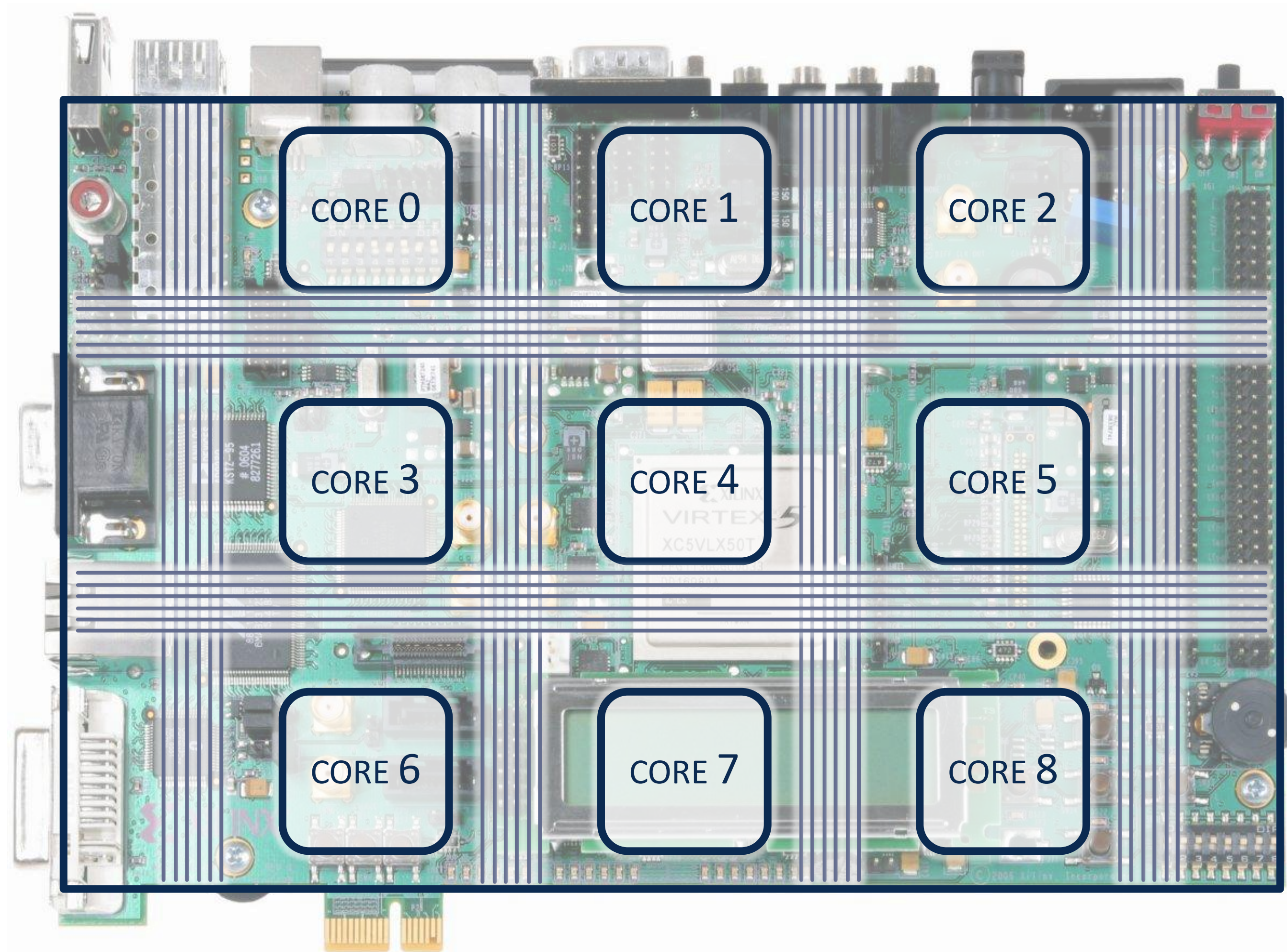
Peter Ebert

Dresden, 12.02.2013

Aufgabenstellung



Aufgabenstellung



Gliederung

1. Grundlagen
   - Scheduling-Algorithmen 5-16
   - SHAP-Architektur 17-23

2. Implementierung
   - Ambitionen 24
   - Scheduler 25-29
   - Locks 30-33
   - Konzept 34-41
   - Dialog der Kerne 42-53

3. Auswertung
   - Ressourcen 54-60
   - Leistung 61-77

4. Zusammenfassung 78-82

1.1 Scheduling-Algorithmen

Klassische Strategien

- Simpel (First In First Out)
- Vorhersage (Shortest Job First, Shortest Remaining Time First)
- Vorhersage ohne Verhungern (Highest Response Ratio Next)
- Rundenbasierend (Round-Robin, VRR, WRR, DPRR)
- Prioritäten statisch (Rate Monotonic Sch., Deadline Monotonic Sch.)
- Prioritäten dynamisch (Earliest Deadline First, Least Laxity First, Multilevel Queue Sch., Multilevel Feedback Queue Sch.)

Java

- 10 Prioritäten
- Abbildung auf OS nicht vorgeschrieben

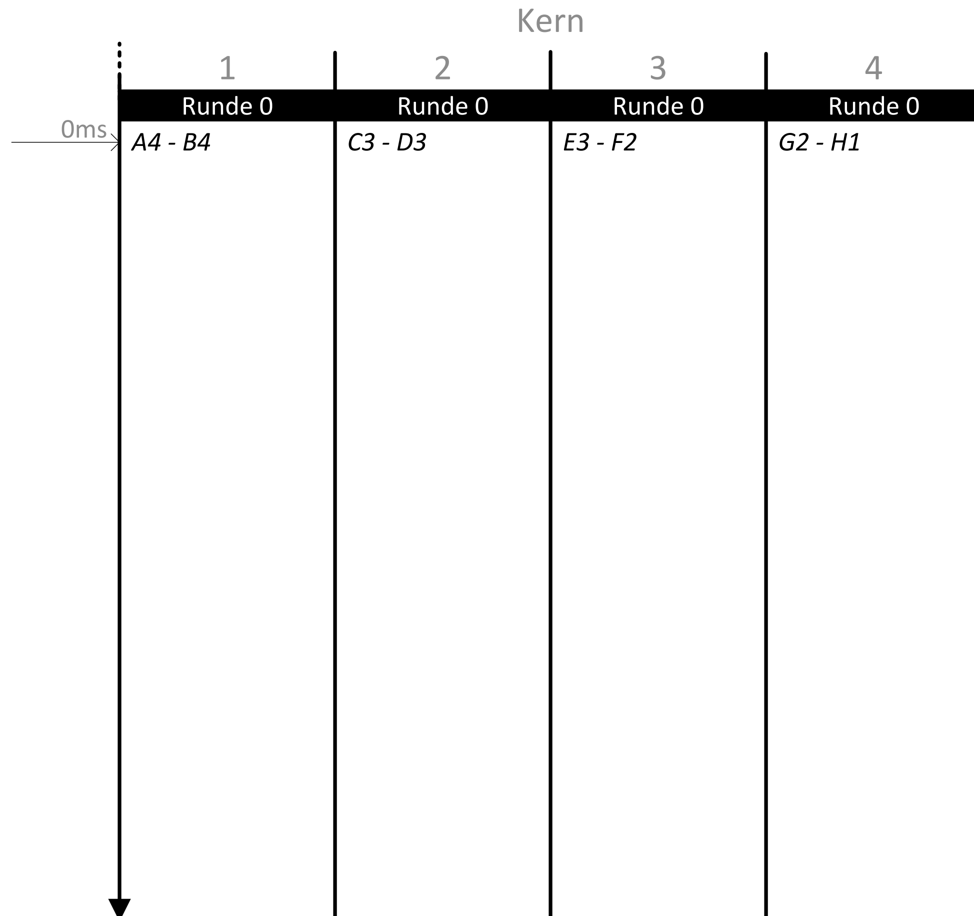
Mehrkern-Scheduler

- Gang Scheduling (Nutzung der Inter-Thread-Kommunikation)
- Distributed Weighted Round-Robin (Threadmigration zur Gleichverteilung der Rundenlängen)

Distributed Weighted Round-Robin

Wichtungen (Threads):

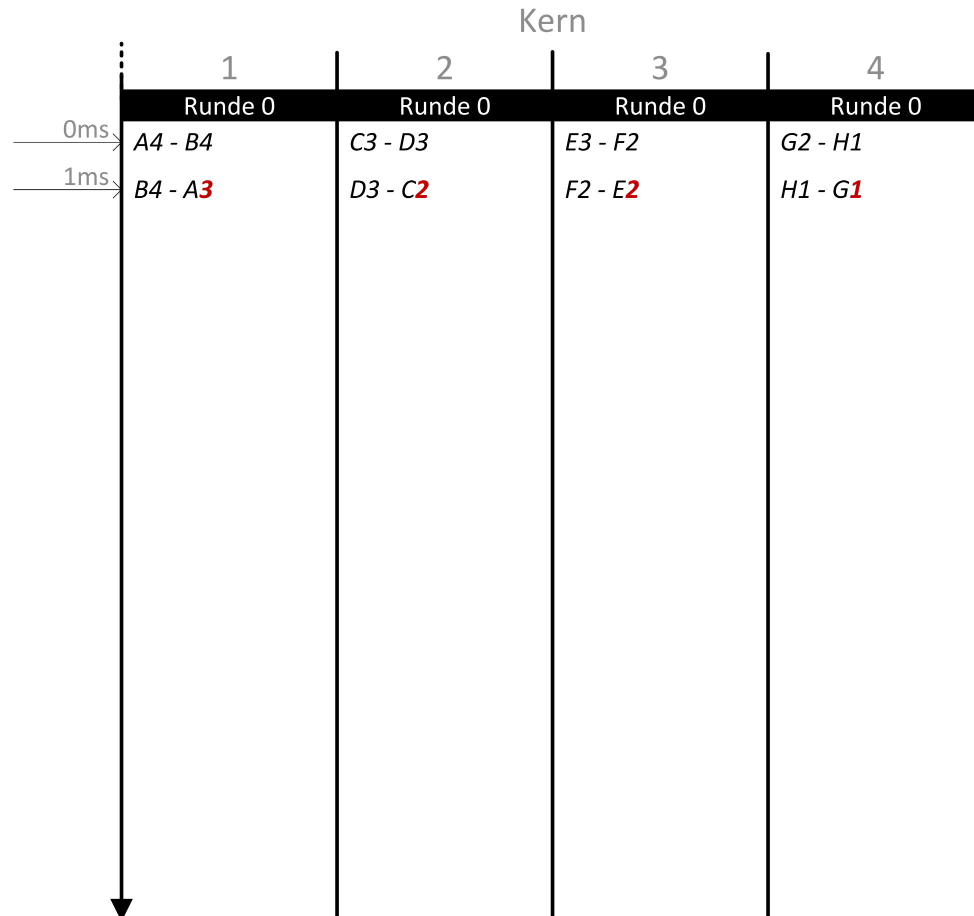
4 (A,B)
3 (C,D,E)
2 (F,G)
1 (H)



Distributed Weighted Round-Robin

Wichtungen (Threads):

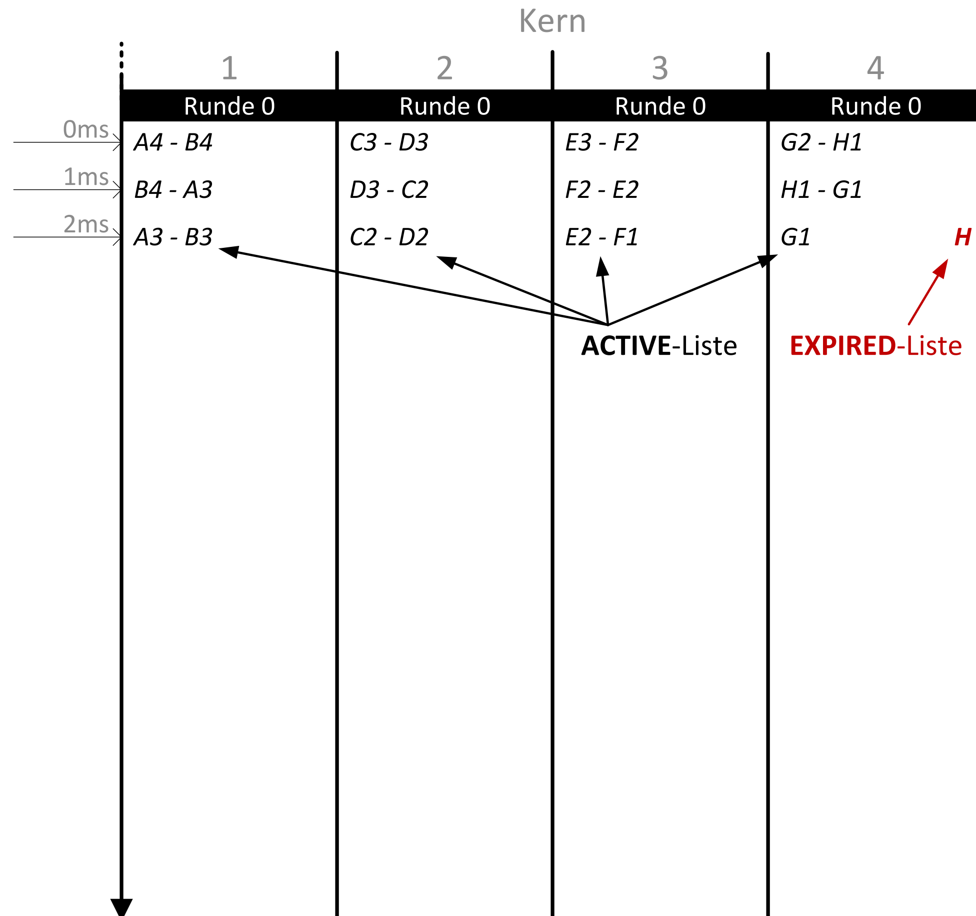
- 4 (A,B)
- 3 (C,D,E)
- 2 (F,G)
- 1 (H)



Distributed Weighted Round-Robin

Wichtungen (Threads):

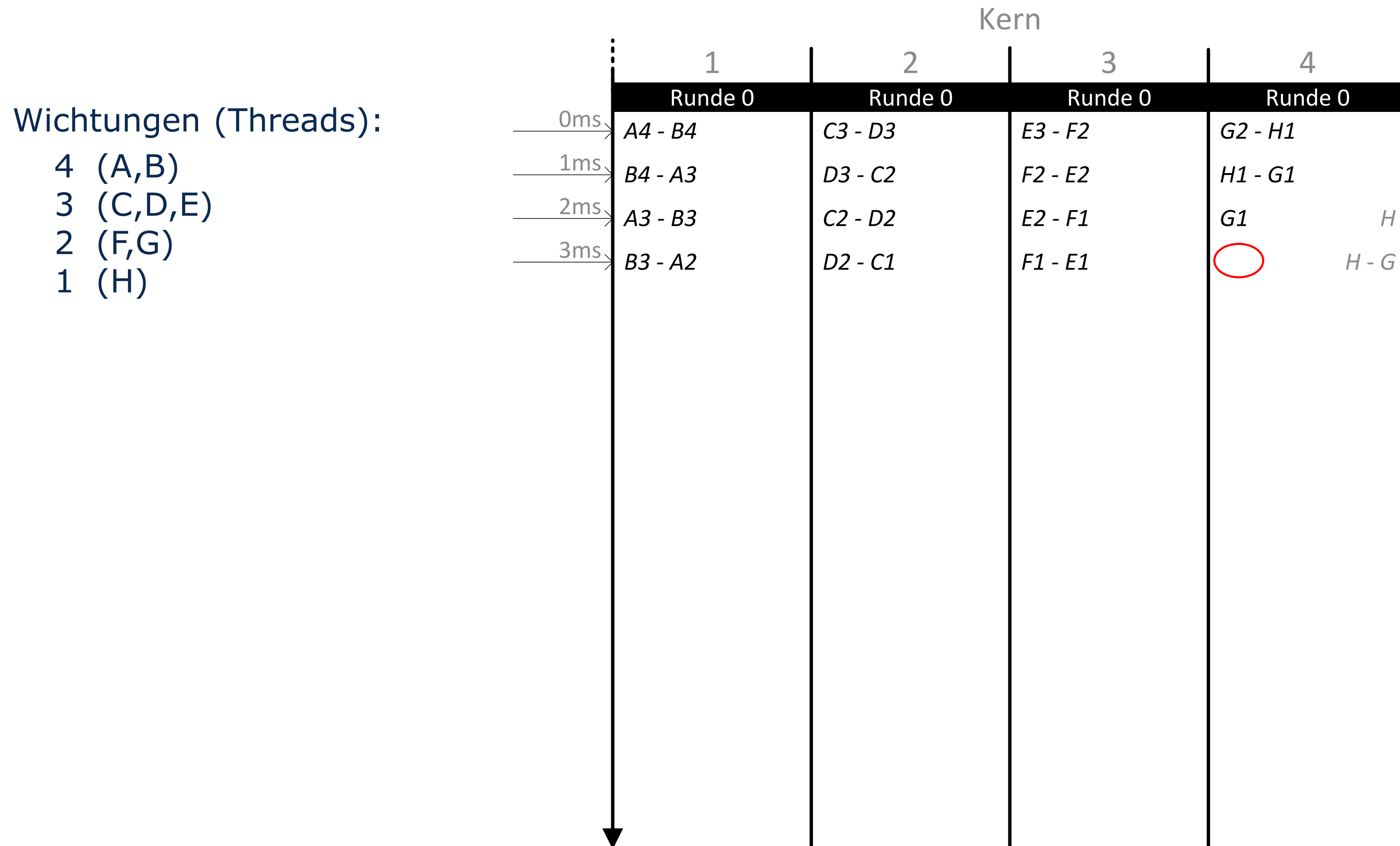
- 4 (A,B)
- 3 (C,D,E)
- 2 (F,G)
- 1 (H)



Distributed Weighted Round-Robin

Wichtungen (Threads):

- 4 (A,B)
- 3 (C,D,E)
- 2 (F,G)
- 1 (H)



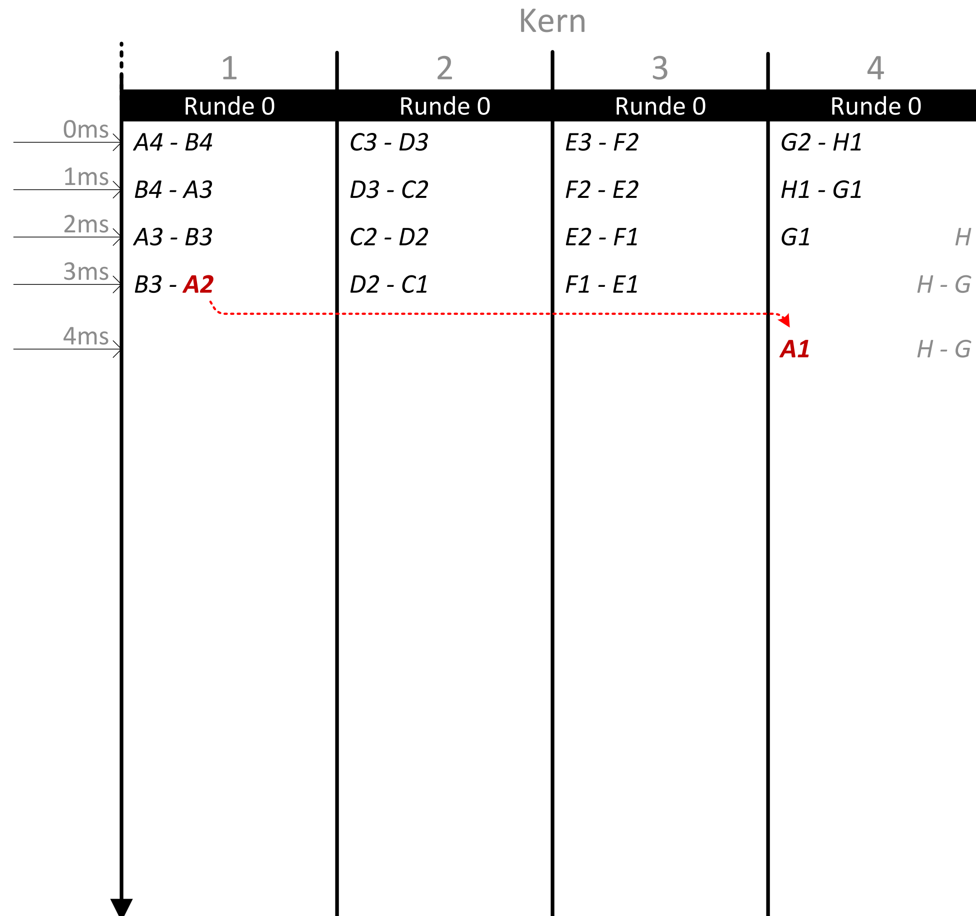
Distributed Weighted Round-Robin

Wichtungen (Threads):

4 (A,B)
3 (C,D,E)
2 (F,G)
1 (H)

Vom Kern mit den meisten Threads wird migriert.

Erster Thread aus ACTIVE wird migriert.



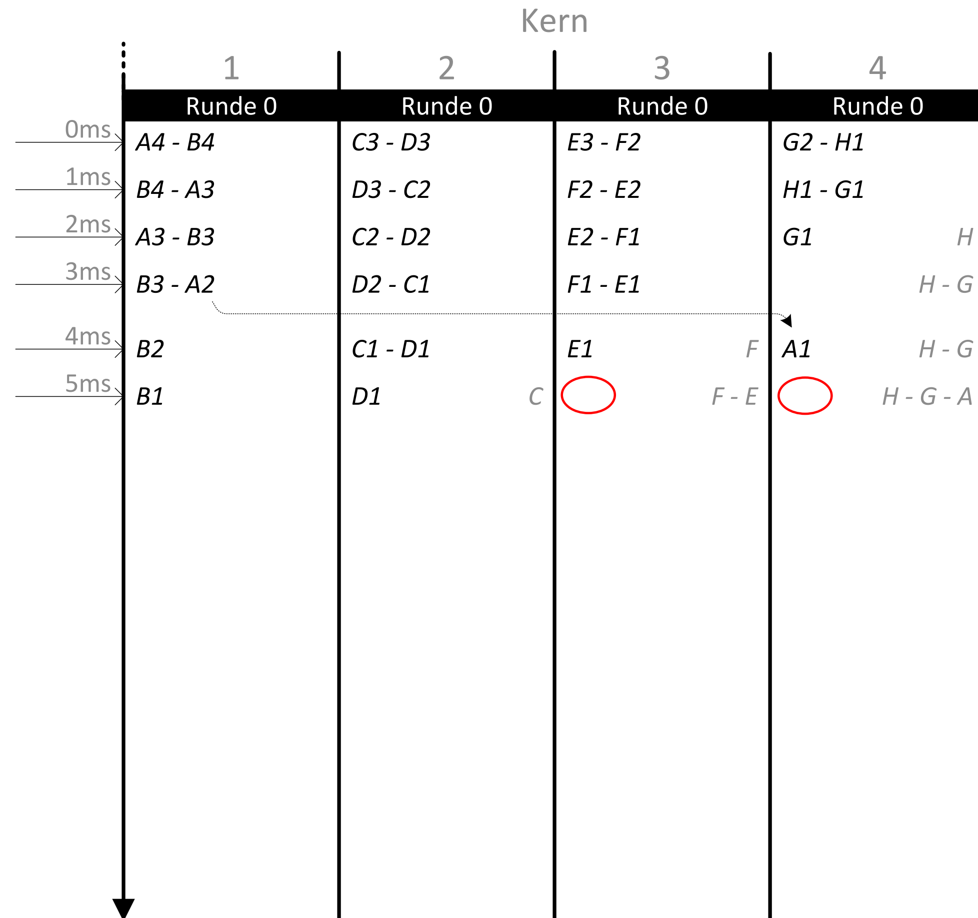
Distributed Weighted Round-Robin

Wichtungen (Threads):

4 (A,B)
3 (C,D,E)
2 (F,G)
1 (H)

Vom Kern mit den meisten Threads wird migriert.

Erster Thread aus ACTIVE wird migriert.



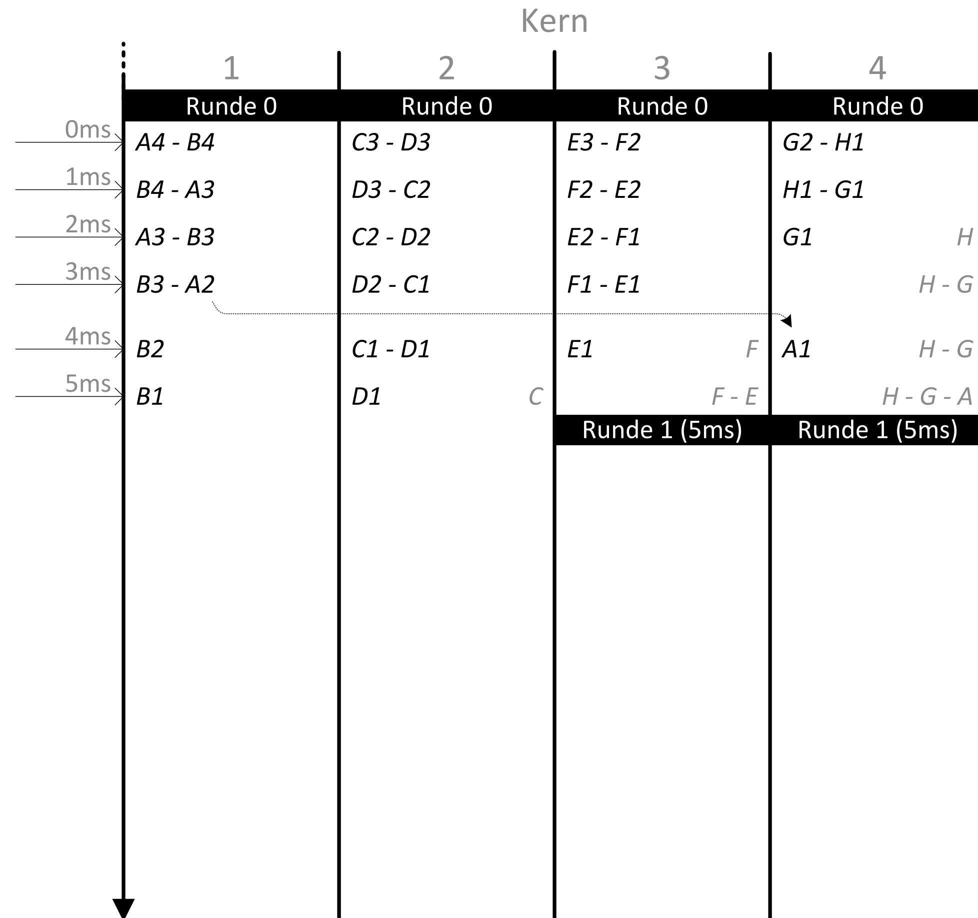
Distributed Weighted Round-Robin

Wichtungen (Threads):

- 4 (A,B)
- 3 (C,D,E)
- 2 (F,G)
- 1 (H)

Vom Kern mit den meisten Threads wird migriert.

Erster Thread aus ACTIVE wird migriert.



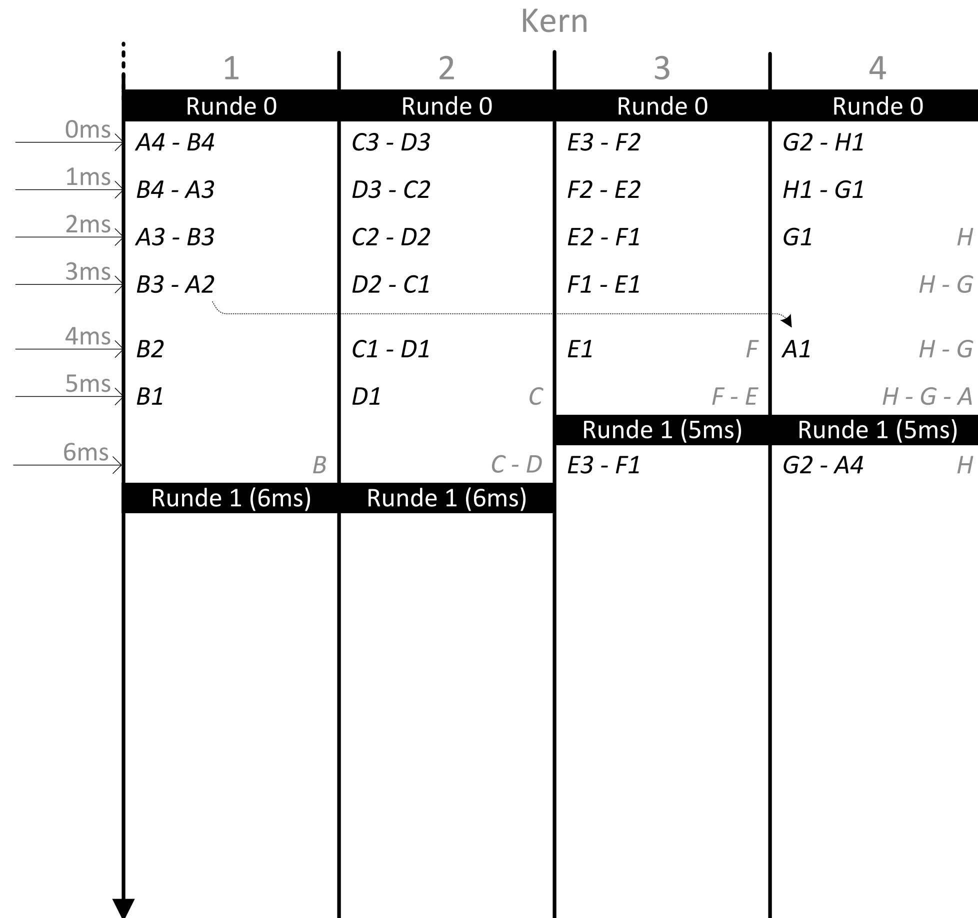
Distributed Weighted Round-Robin

Wichtungen (Threads):

4 (A,B)
3 (C,D,E)
2 (F,G)
1 (H)

Vom Kern mit den meisten Threads wird migriert.

Erster Thread aus ACTIVE wird migriert.



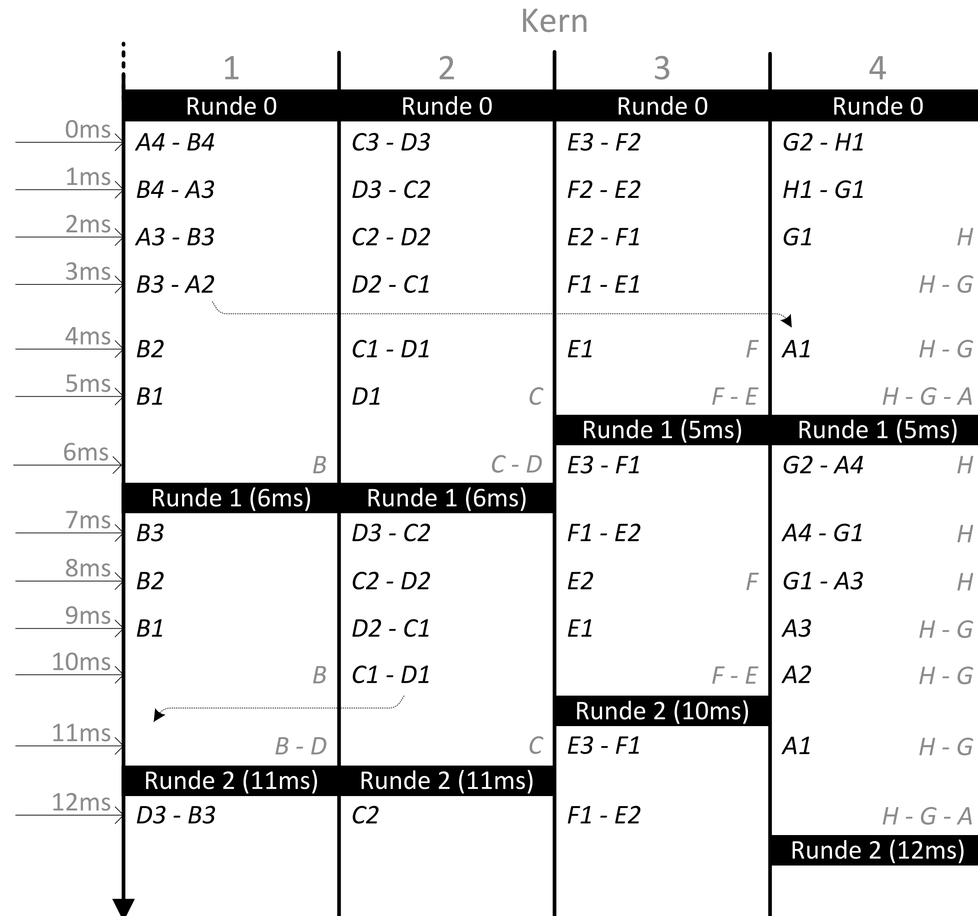
Distributed Weighted Round-Robin

Wichtungen (Threads):

4 (A,B)
3 (C,D,E)
2 (F,G)
1 (H)

Vom Kern mit den meisten Threads wird migriert.

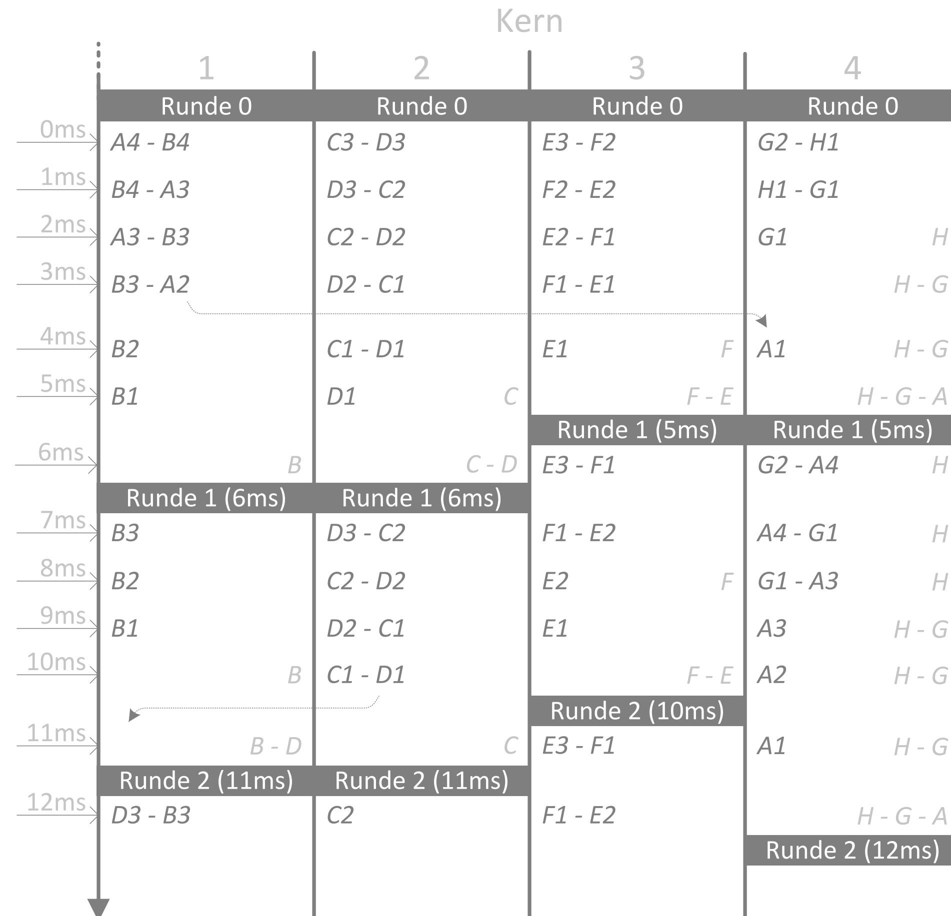
Erster Thread aus ACTIVE wird migriert.



Distributed Weighted Round-Robin

- + Ressourcen-schonend
- + Freiheit in Detailfragen
- + kein Leerlaufen
- + keine Synchronisierung d. Kerne (keine Wartezeiten)
- + Unterstützung von Prioritäten (Wichtungen)
- + Fairness
- + zusätzl. inneres Scheduling
- + gute Skalierbarkeit
- + kleine max. Antwortzeit

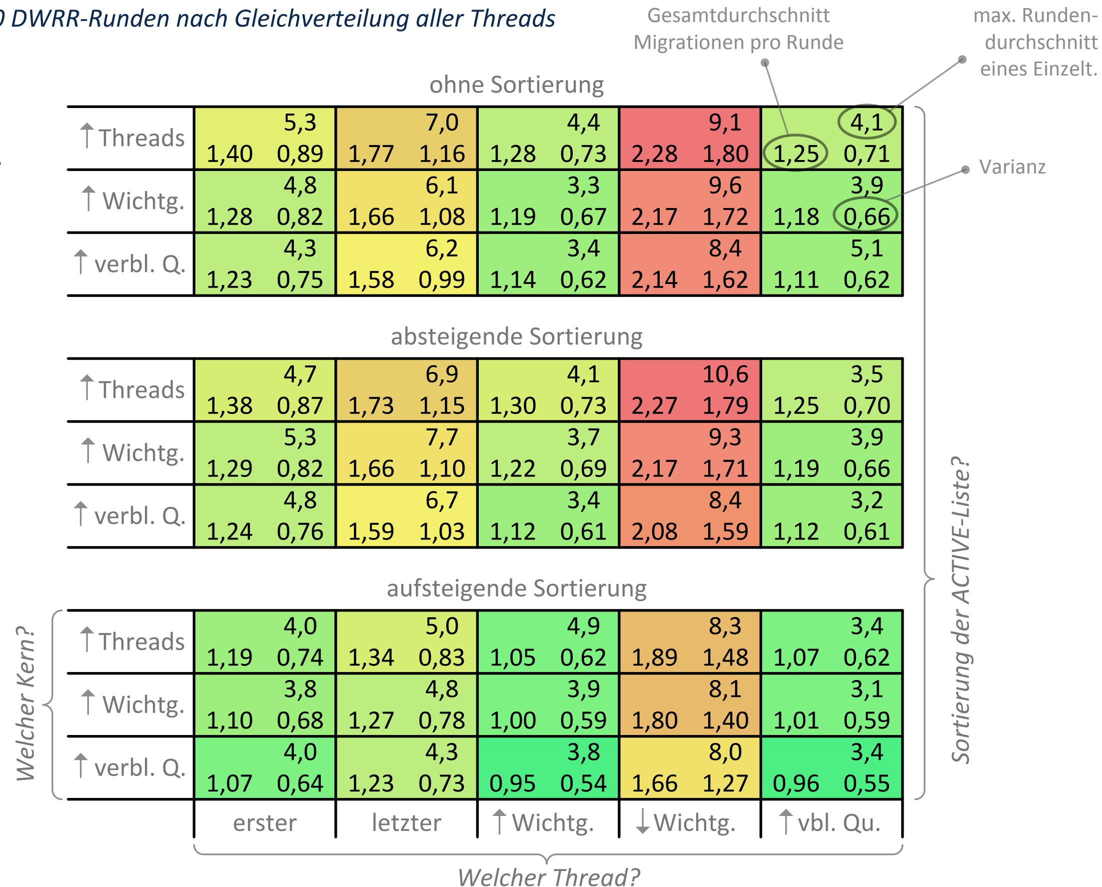
- – keine Inter-Thread-Kommunikation
- – Übersteuern möglich



Distributed Weighted Round-Robin

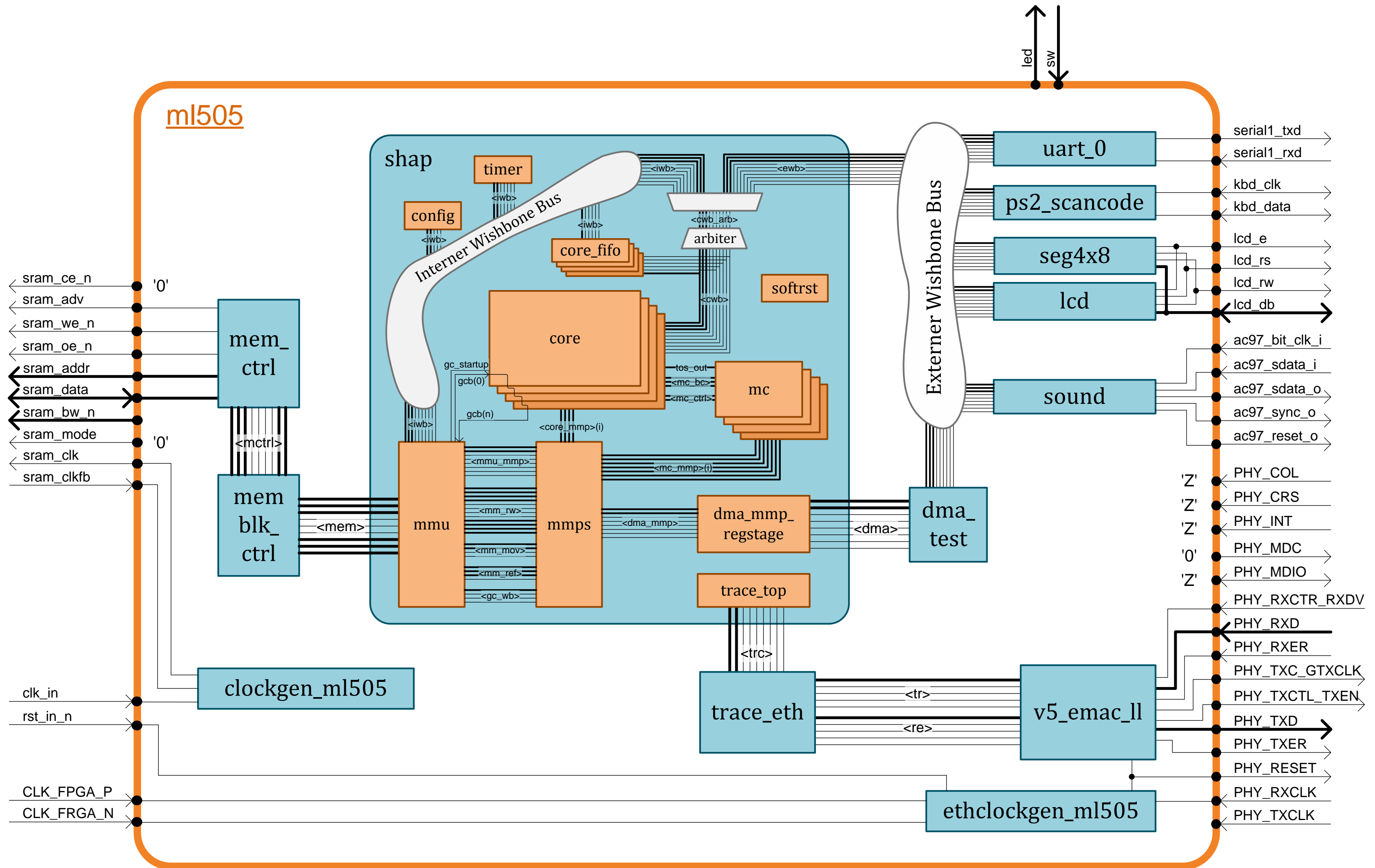
5000 Einzeltests mit je 30 DWRR-Runden nach Gleichverteilung aller Threads

Kerne: [2,18]
*Threads: [3, 198]**
Wichtungen: [1, 10]
* *abhängig von der Anzahl der Kerne*

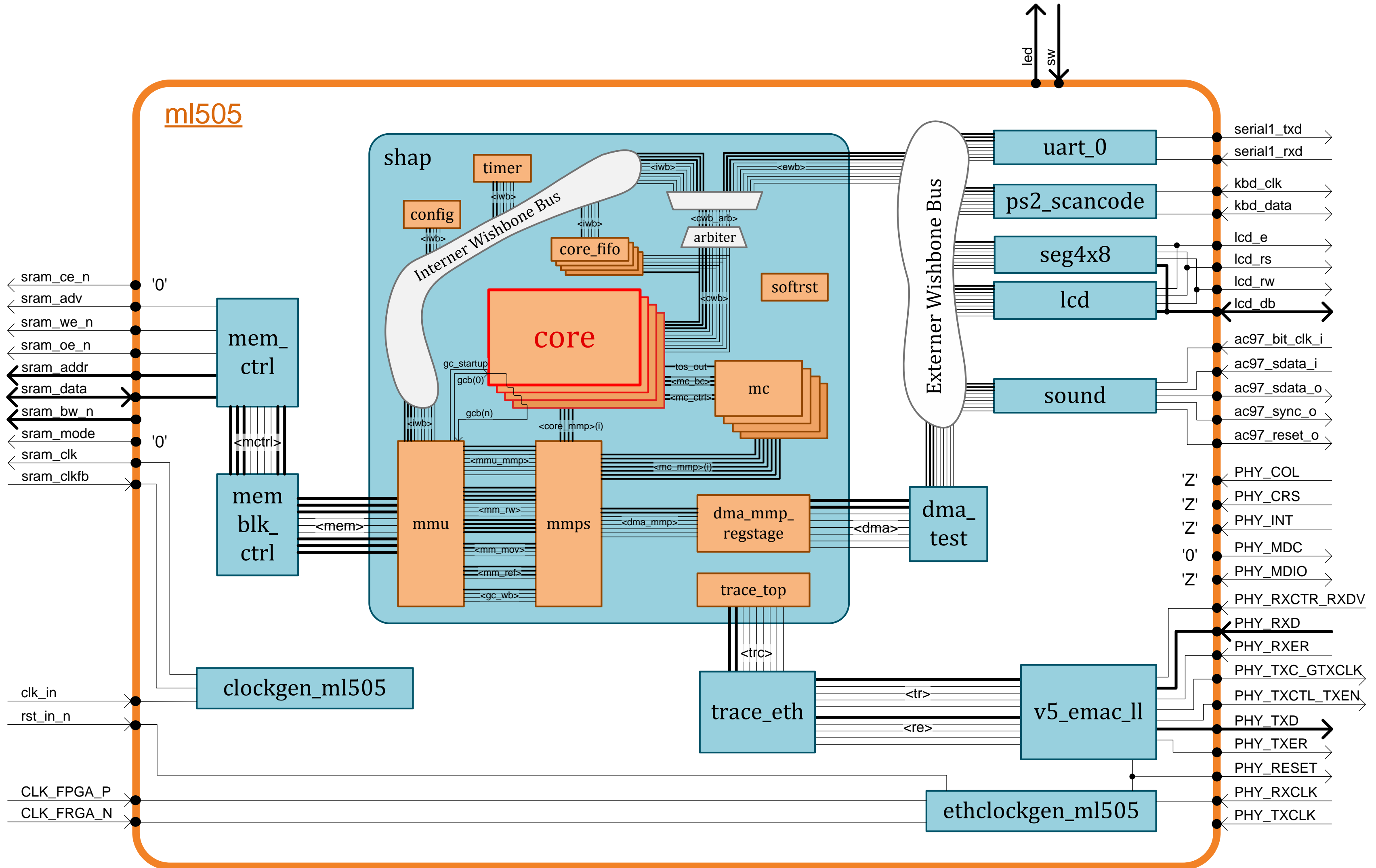


Sortierung der ACTIVE-Liste?	Welcher Kern?	erster	letzter	↑Wichtg.	↓Wichtg.	↑vbl. Qu.
ohne Sortierung	↑Threads	5,3 / 1,40 0,89	7,0 / 1,77 1,16	4,4 / 1,28 0,73	9,1 / 2,28 1,80	4,1 / 1,25 0,71
ohne Sortierung	↑Wichtg.	4,8 / 1,28 0,82	6,1 / 1,66 1,08	3,3 / 1,19 0,67	9,6 / 2,17 1,72	3,9 / 1,18 0,66
ohne Sortierung	↑verbl. Q.	4,3 / 1,23 0,75	6,2 / 1,58 0,99	3,4 / 1,14 0,62	8,4 / 2,14 1,62	5,1 / 1,11 0,62
absteigende Sortierung	↑Threads	4,7 / 1,38 0,87	6,9 / 1,73 1,15	4,1 / 1,30 0,73	10,6 / 2,27 1,79	3,5 / 1,25 0,70
absteigende Sortierung	↑Wichtg.	5,3 / 1,29 0,82	7,7 / 1,66 1,10	3,7 / 1,22 0,69	9,3 / 2,17 1,71	3,9 / 1,19 0,66
absteigende Sortierung	↑verbl. Q.	4,8 / 1,24 0,76	6,7 / 1,59 1,03	3,4 / 1,12 0,61	8,4 / 2,08 1,59	3,2 / 1,12 0,61
aufsteigende Sortierung	↑Threads	4,0 / 1,19 0,74	5,0 / 1,34 0,83	4,9 / 1,05 0,62	8,3 / 1,89 1,48	3,4 / 1,07 0,62
aufsteigende Sortierung	↑Wichtg.	3,8 / 1,10 0,68	4,8 / 1,27 0,78	3,9 / 1,00 0,59	8,1 / 1,80 1,40	3,1 / 1,01 0,59
aufsteigende Sortierung	↑verbl. Q.	4,0 / 1,07 0,64	4,3 / 1,23 0,73	3,8 / 0,95 0,54	8,0 / 1,66 1,27	3,4 / 0,96 0,55

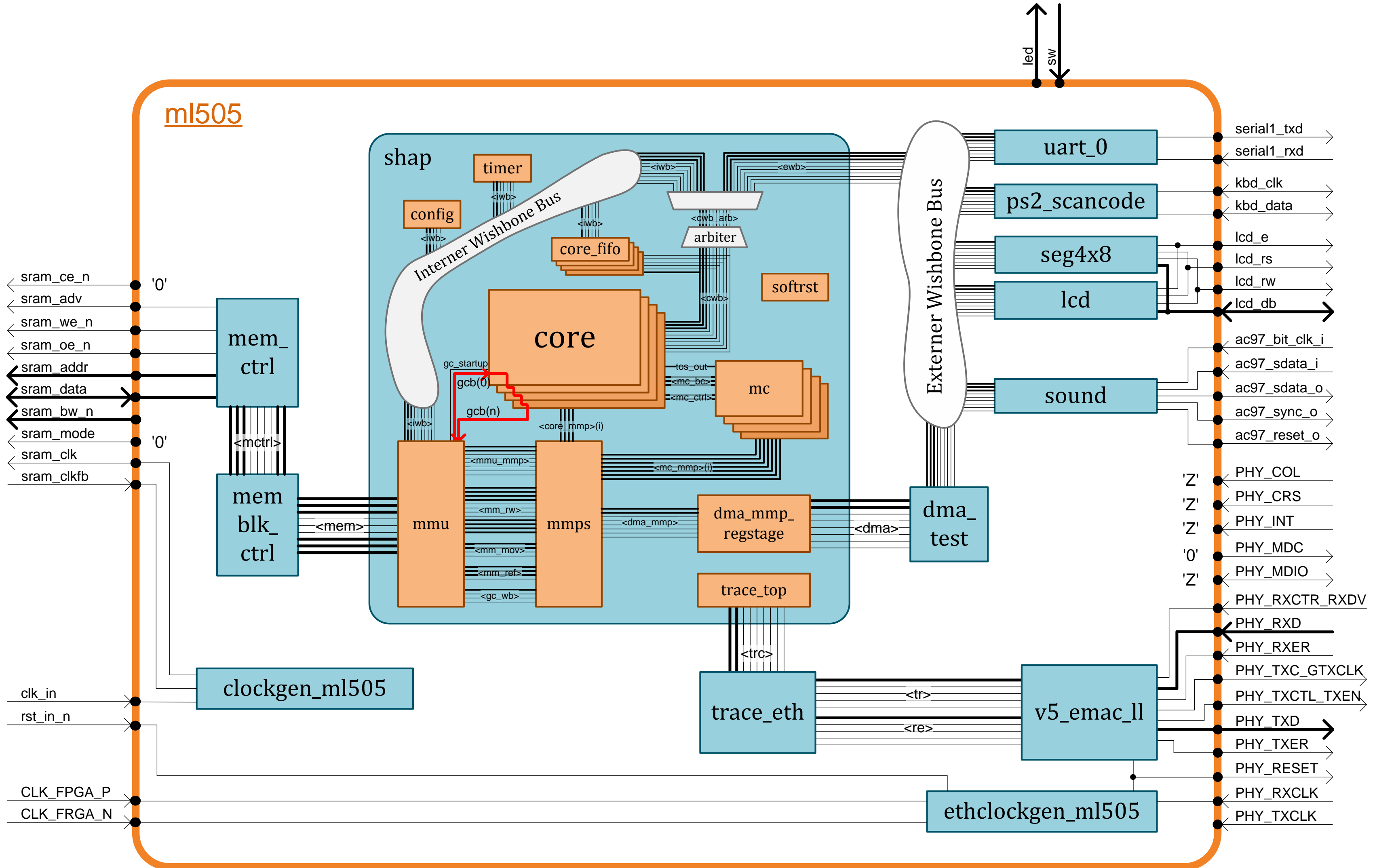
1.2 SHAP-Architektur



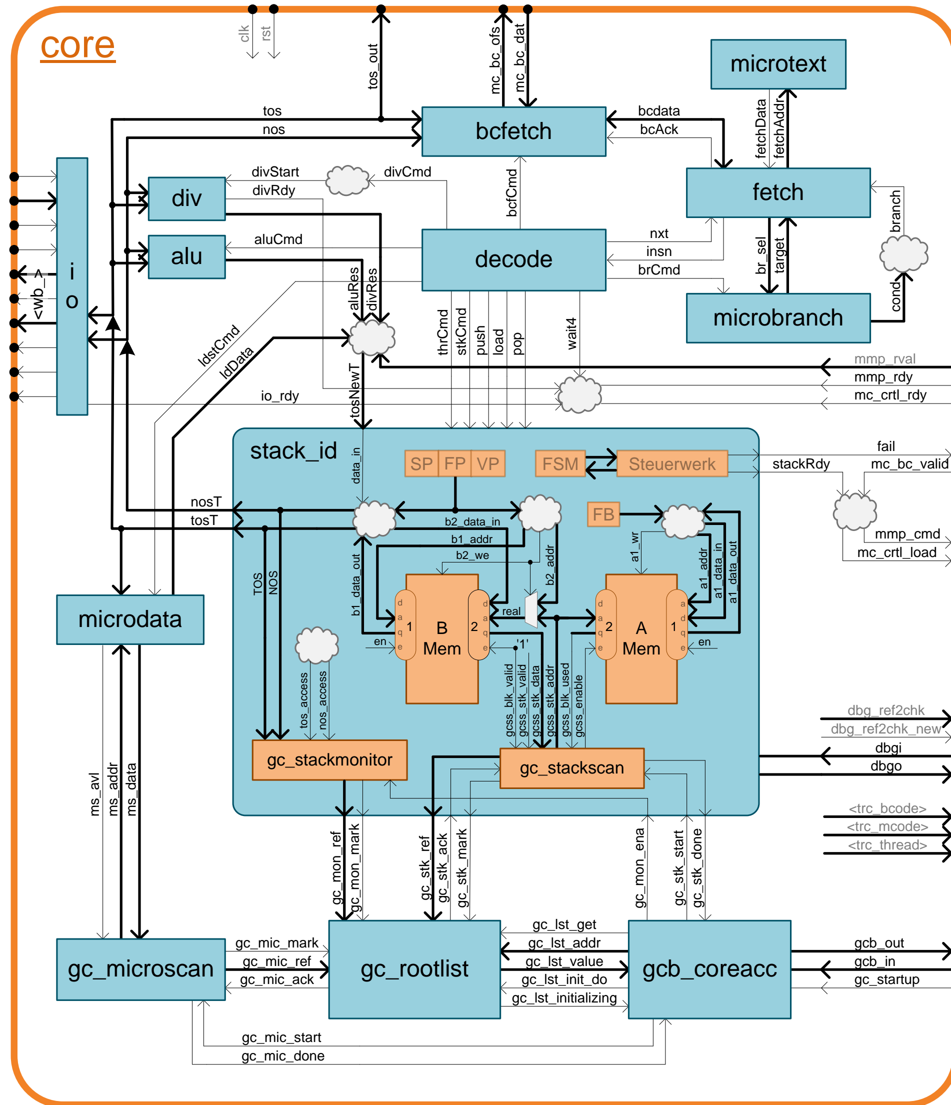
1.2 SHAP-Architektur



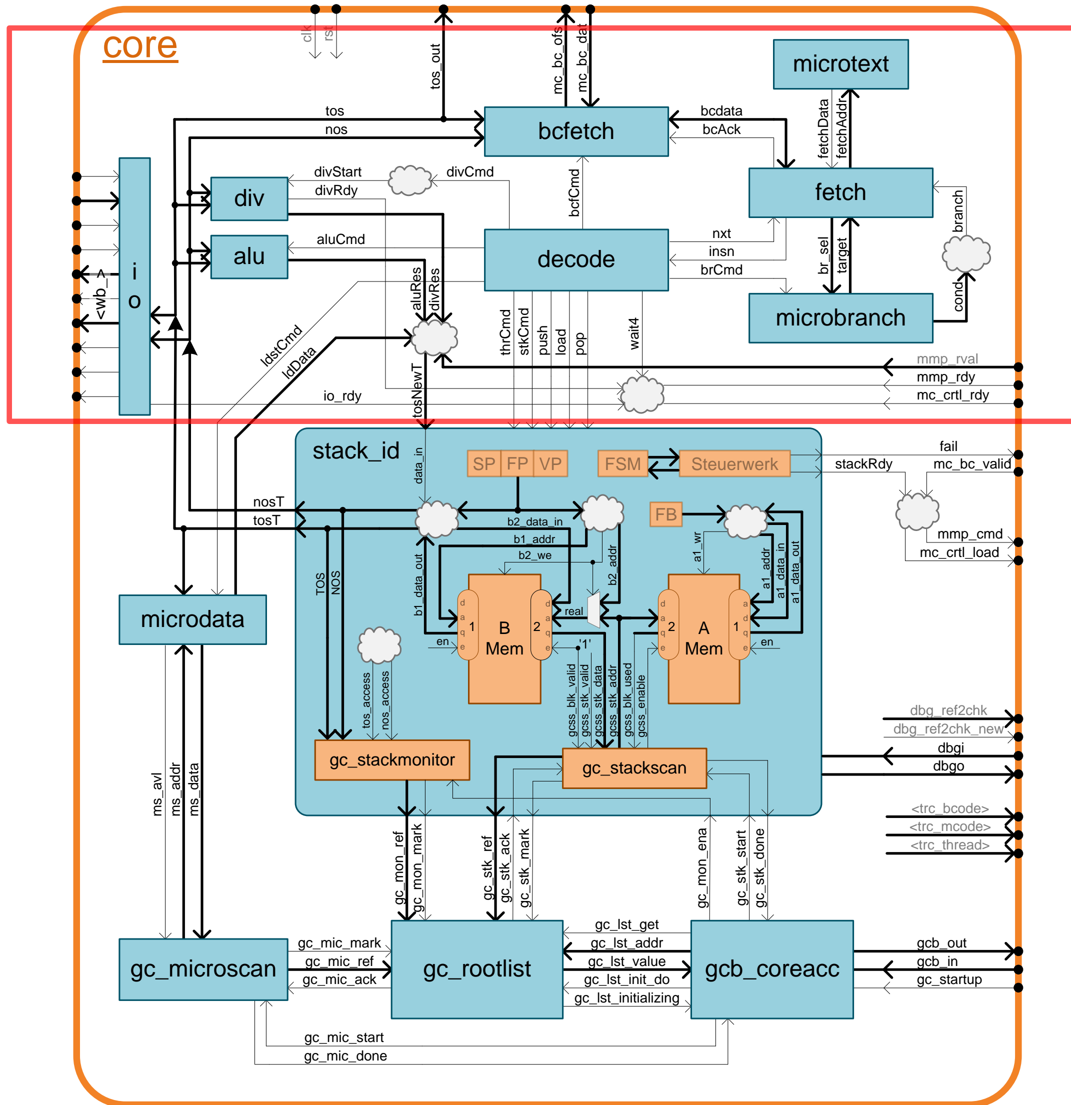
1.2 SHAP-Architektur



1.2 SHAP-Architektur



1.2 SHAP-Architektur



Pipeline

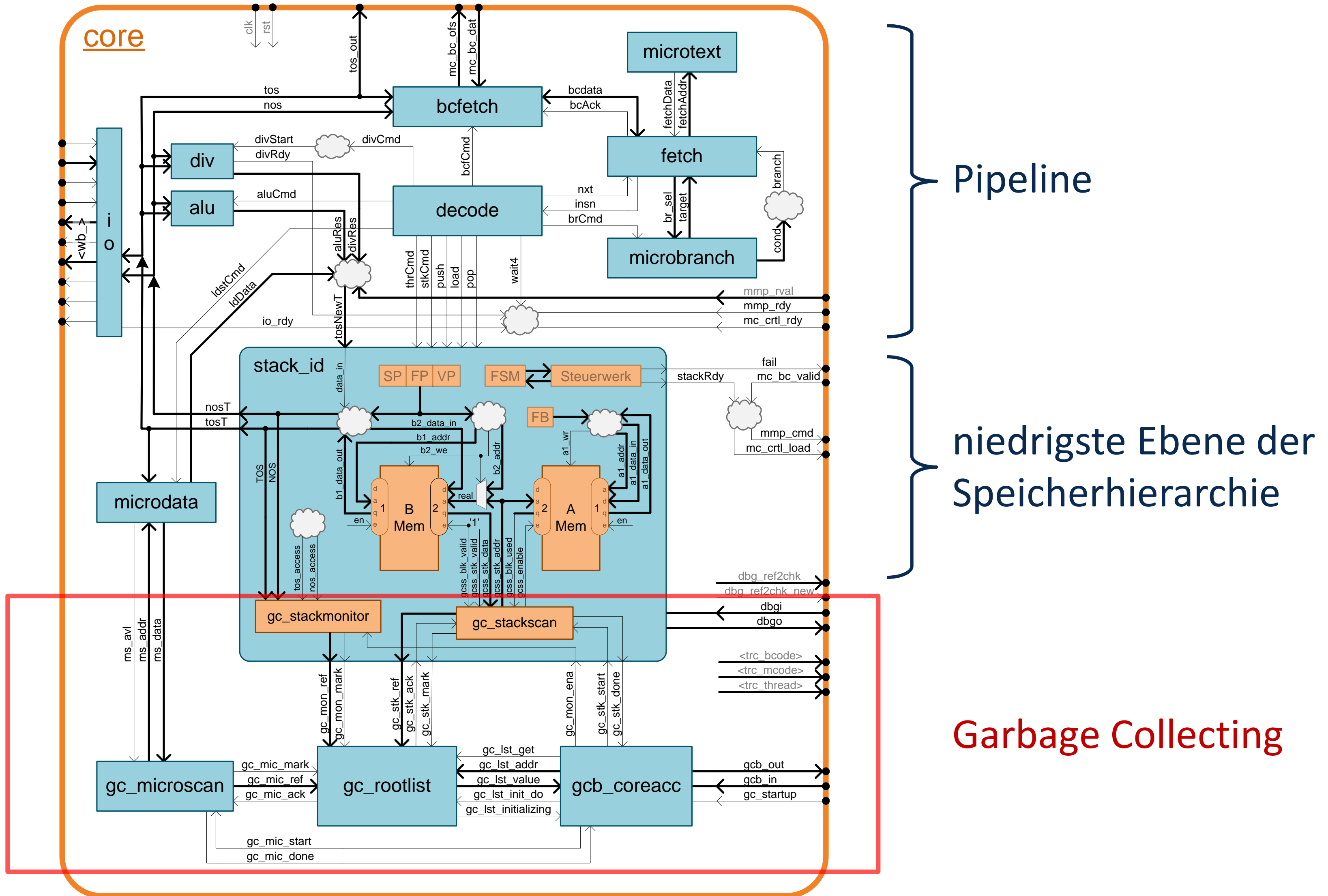
1.2 SHAP-Architektur

core

Pipeline

niedrigste Ebene der Speicherhierarchie

1.2 SHAP-Architektur



2.1 Ambitionen

Sollkriterien

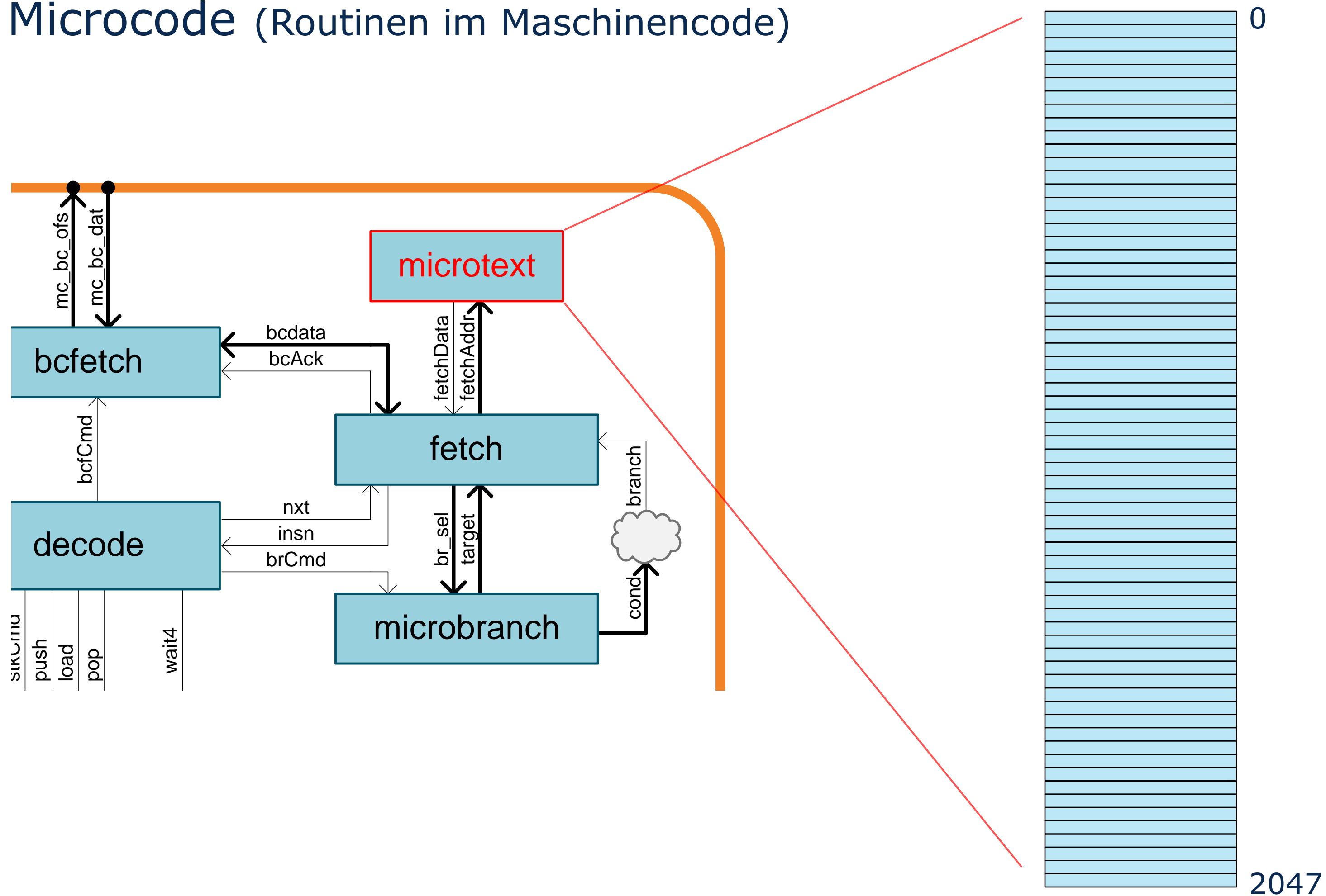
- geringe Umbauten an SHAP
- geringer Scheduling-Overhead
- geringer Migrations-Overhead
- geringe Zusatzhardware

} einfache Berechnungen

Kannkriterien

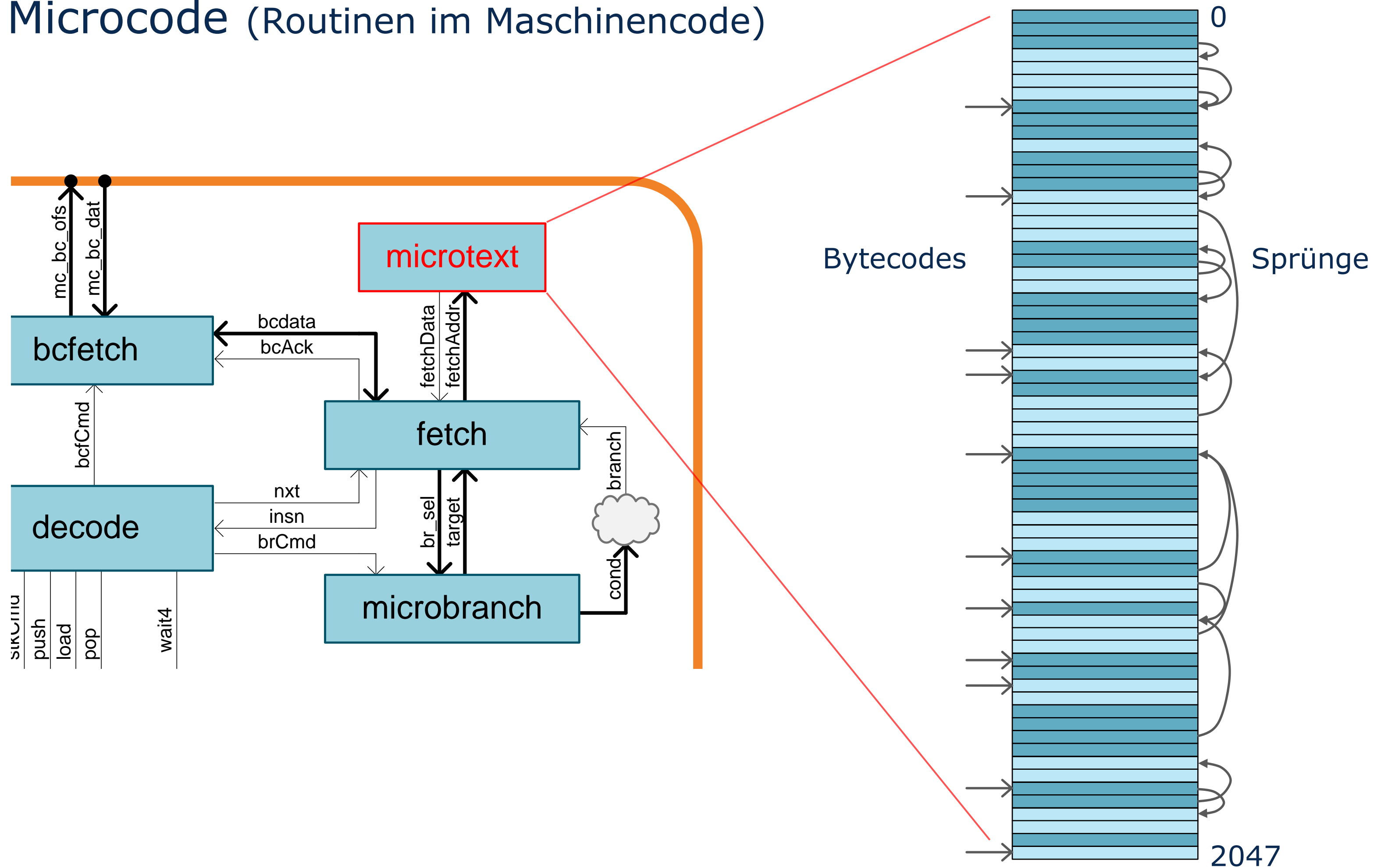
- Unterstützung der 10 Java-Prioritäten

Microcode (Routinen im Maschinencode)



2.2 Scheduler

Microcode (Routinen im Maschinencode)



2.2 Scheduler

Original

schwarz — Notwendig für Scheduling

schwarz → *schwarz*

fett — Einsprungpunkt aus Bytecode o. Interrupt

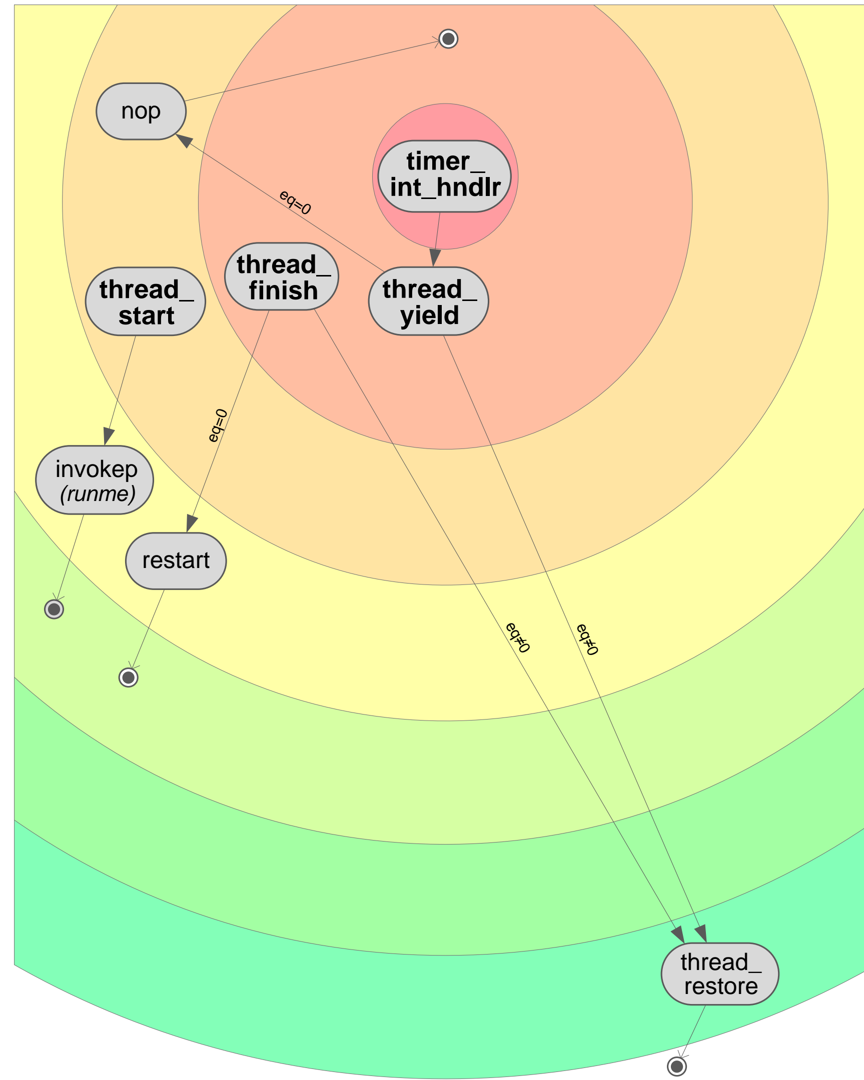
grau — Original Scheduler

eq≠0 → Sprungbedingung

◉ Next-Flag, nächster Bytecode

^ Bitw. XOR

eq … ct^ct.next



ACTIVE & EXPIRED

schwarz — Notwendig für Scheduling
schwarz / *schwarz* →
fett — Einsprungpunkt aus Bytecode o. Interrupt
grau — Original Scheduler

e=-1 → Wertzuweisung
eq≠0 → Sprungbedingung
◉ Next-Flag, nächster Bytecode

pop
timer_ int_hndlr
thread_ start
thread_ finish
thread_ yield
invokep *(runme)*
empty_ active
ud_active
swapp
ud_expired
dispatch
refill_slices
dispatchY
thread_ restore
store_ stt_on_0

eq|ex=0
eq≠0
eq=0,
ex≠0**eq=0
eq=0**ex≠0
eq≠0**slc≤0
f=1
eq≠0**slc>0
slc>0
slc≤0
eq=0**ex≠0
eq=0||eq≠0**ex=0
eq=0
eq≠0
a = ex.next
ex=0,
stt.rnd++
a ≠ ex.next
f=0
f≠0
f=0
stt=arg

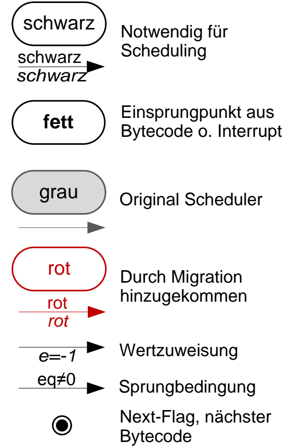
** || Log. AND u. OR
^ | Bitw. XOR u. OR

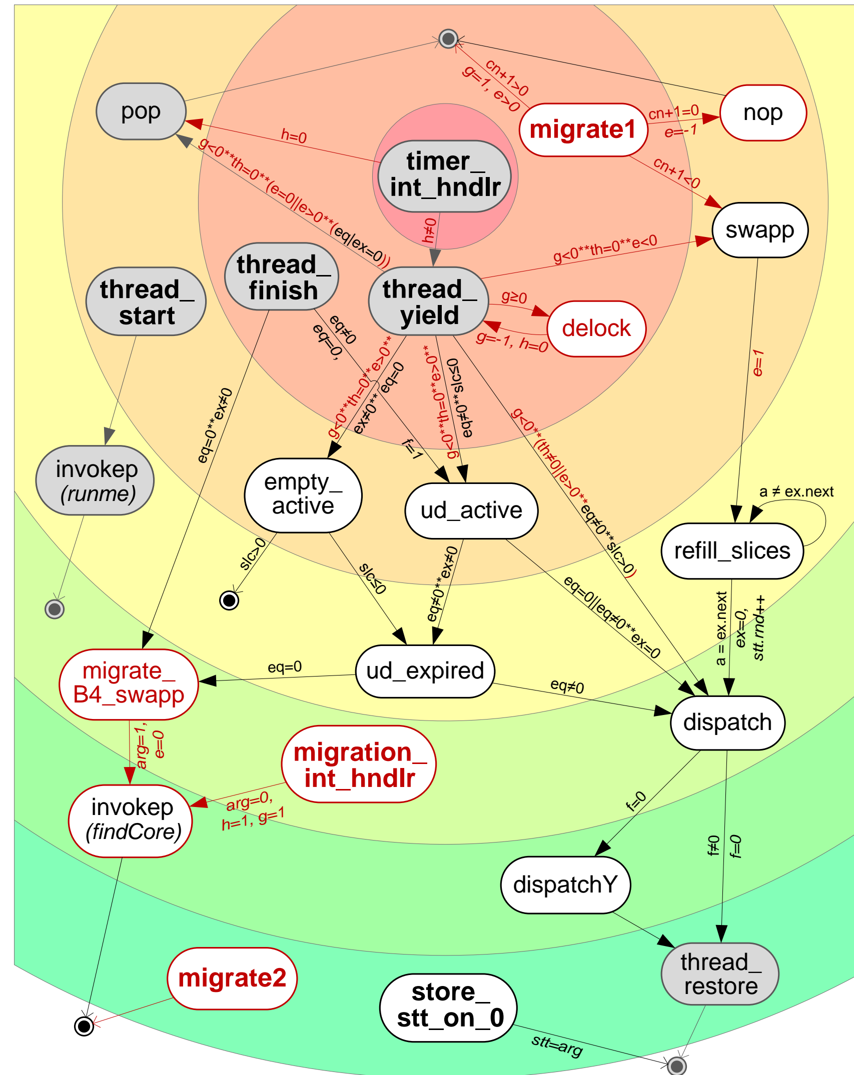
slc … verbleib. Slices
stt … StarterThread

arg … Argument
eq … ct^ct.next

rnd … Runde

DWRR





** || Log. AND u. OR
^ | Bitw. XOR u. OR

slc … verbleib. Slices
stt … StarterThread
cn … Kernnummer
arg … Argument
eq … ct^ct.next
th … Thread
rnd … Runde

Locks

GC-Lock (Garbage Collector nicht zeitgleich mit Migration)

Migration-Lock (max. eine Threadmigration gleichzeitig)

Yield-Lock (kein Threadwechsel während der Migration)

Locks

GC-Lock (Garbage Collector nicht zeitgleich mit Migration)

- keine Sicherungsmaßnahmen für Referenzen im Stack des Zielthreads
- lückenlose Stackübertragung ohne Adressierung möglich
- **Lösung:** Deaktivierungsbefehl für ZPU über IWB

Migration-Lock (max. eine Threadmigration gleichzeitig)

Yield-Lock (kein Threadwechsel während der Migration)

Locks

GC-Lock (Garbage Collector nicht zeitgleich mit Migration)

- keine Sicherungsmaßnahmen für Referenzen im Stack des Zielthreads
- lückenlose Stackübertragung ohne Adressierung möglich
- **Lösung:** Deaktivierungsbefehl für ZPU über IWB

Migration-Lock (max. eine Threadmigration gleichzeitig)

- keine Synchronisierung der Kern- und Threadauswahl
- keine Sicherungsmaßnahmen bei GC-Deaktivierung und GC-Bus-Kommunikation
- Vermeidung von Übersteuern
- lückenlose Stackübertragung ohne Adressierung möglich
- **Lösung:** `static-synchronized`-Methode zum Setzen der Lock-Variable

Yield-Lock (kein Threadwechsel während der Migration)

Locks

GC-Lock (Garbage Collector nicht zeitgleich mit Migration)

- keine Sicherungsmaßnahmen für Referenzen im Stack des Zielthreads
- lückenlose Stackübertragung ohne Adressierung möglich
- **Lösung:** Deaktivierungsbefehl für ZPU über IWB

Migration-Lock (max. eine Threadmigration gleichzeitig)

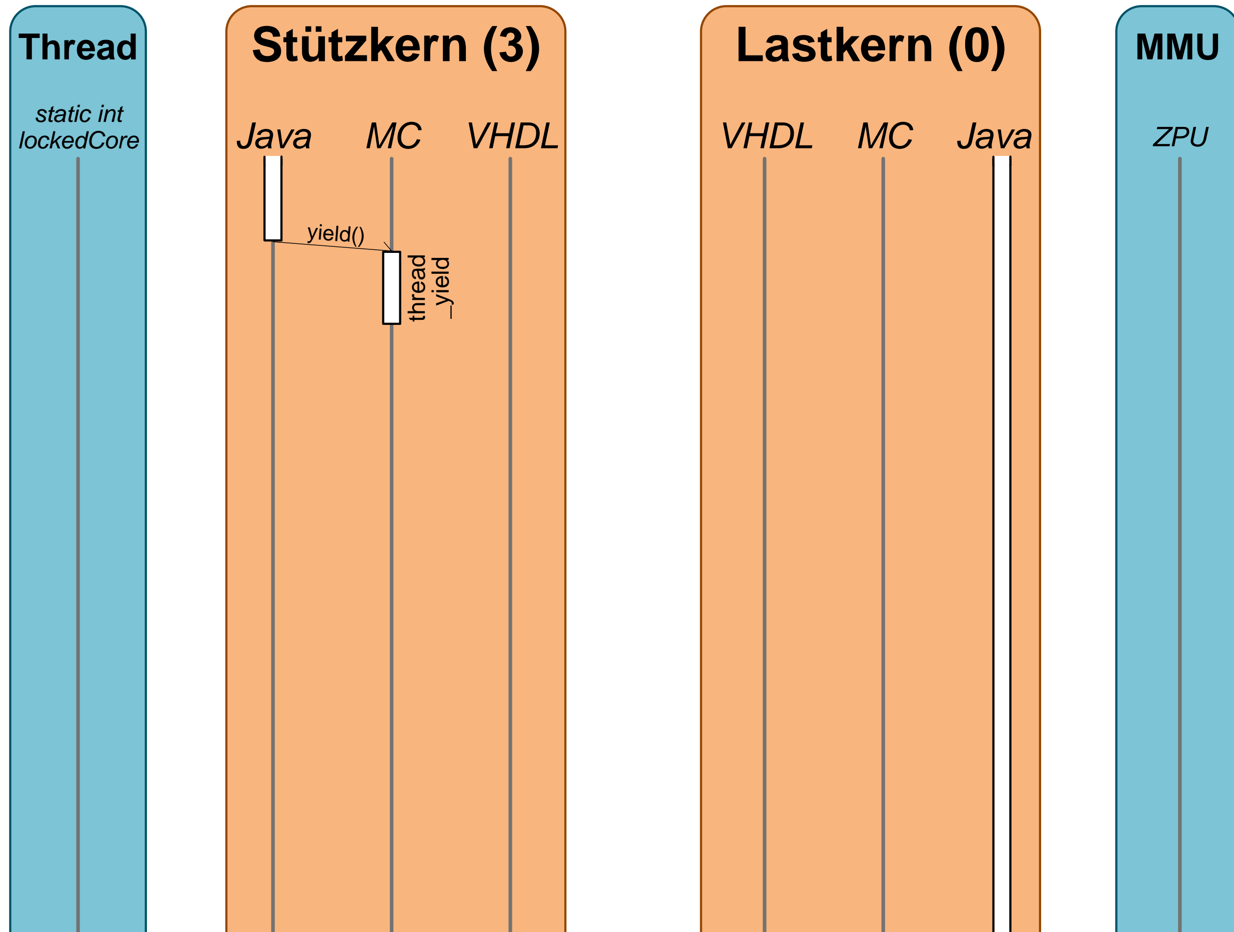
- keine Synchronisierung der Kern- und Threadauswahl
- keine Sicherungsmaßnahmen bei GC-Deaktivierung und GC-Bus-Kommunikation
- Vermeidung von Übersteuern
- lückenlose Stackübertragung ohne Adressierung möglich
- **Lösung:** `static-synchronized`-Methode zum Setzen der Lock-Variable

Yield-Lock (kein Threadwechsel während der Migration)

- um den noch unfertigen Threadwechsel nicht abzubrechen
- benötigt bei: direkten und indirektem Aufruf der MC-Routine `thread_yield`
- gilt auch für Timer-Interrupt
- **Lösung:** Vermeidung von `yield()`, von `io_yield` im MC und Sprünge im MC anhand neuer Variablen

2.4 Konzept

ACTIVE leer

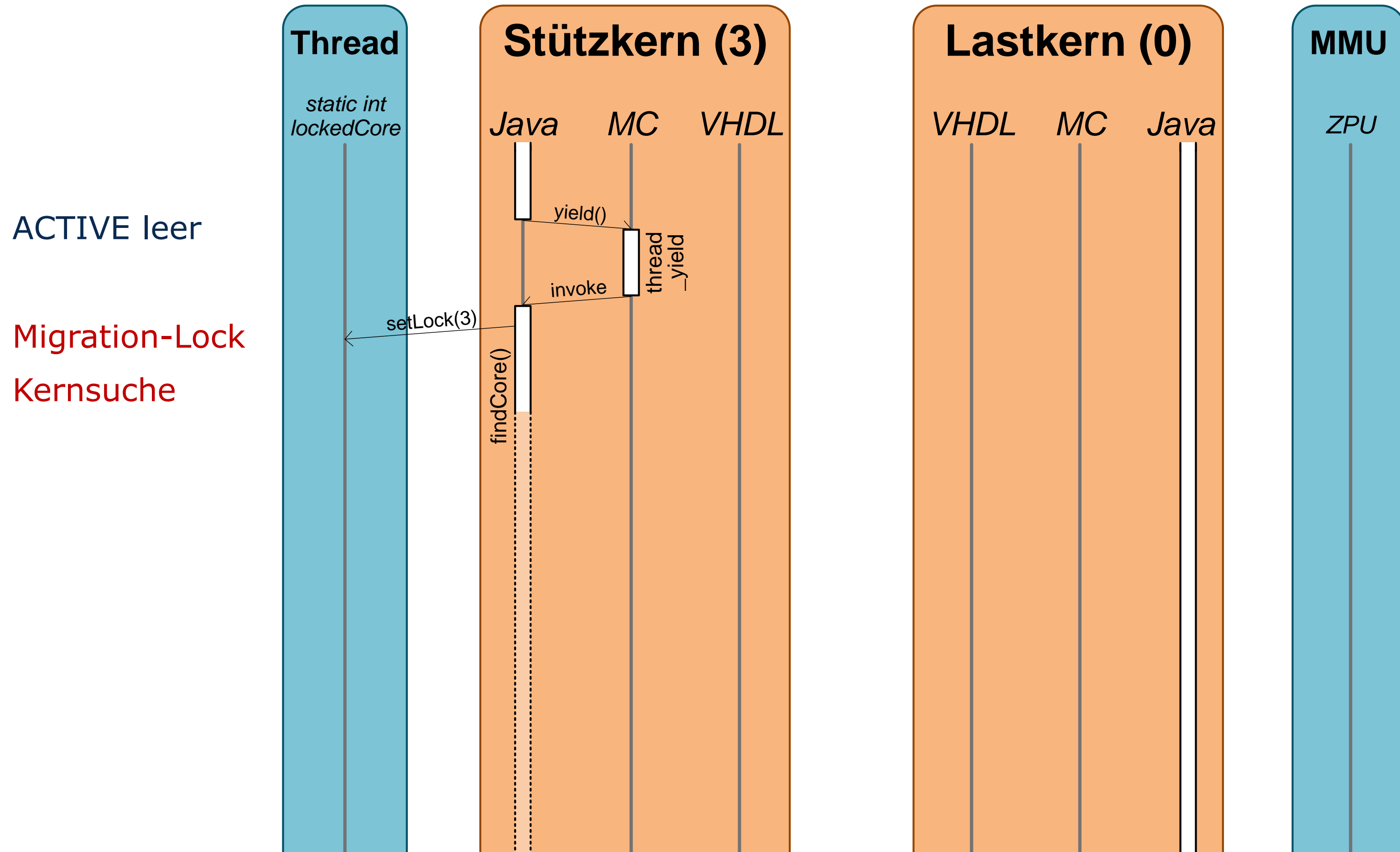


2.4 Konzept

ACTIVE leer

Migration-Lock

Kernsuche



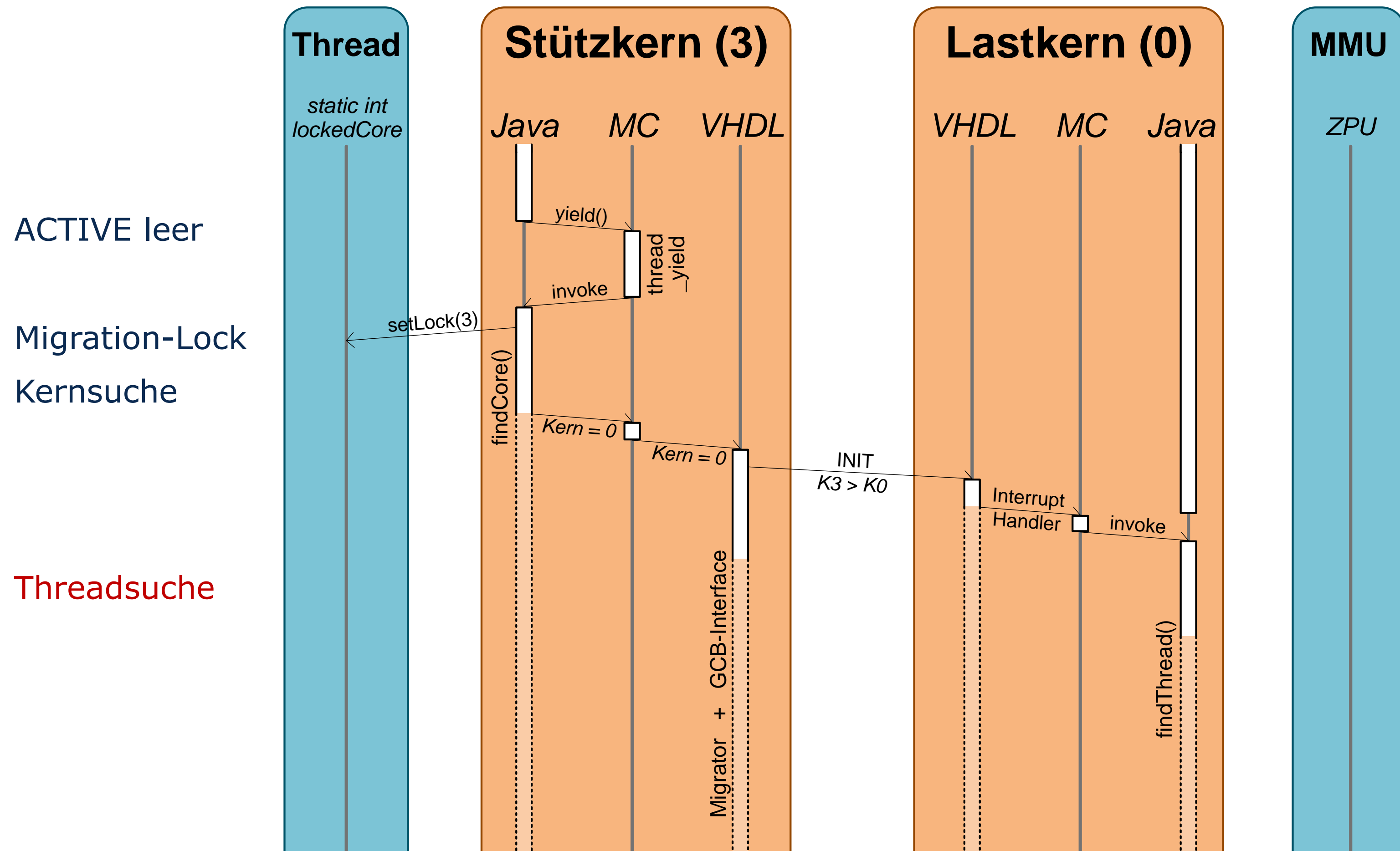
2.4 Konzept

ACTIVE leer

Migration-Lock

Kernsuche

Threadsuche



2.4 Konzept

ACTIVE leer

Migration-Lock

Kernsuche

Threadsuche

GC-Lock

Thread

static int lockedCore

Stützkern (3)

Java *MC* *VHDL*

Lastkern (0)

VHDL *MC* *Java*

MMU

ZPU

yield()

thread _yield

invoke

setLock(3)

findCore()

Kern = 0

Kern = 0

Migrator + GCB-Interface

INIT

K3 > K0

Interrupt Handler

invoke

findThread()

GC:aus

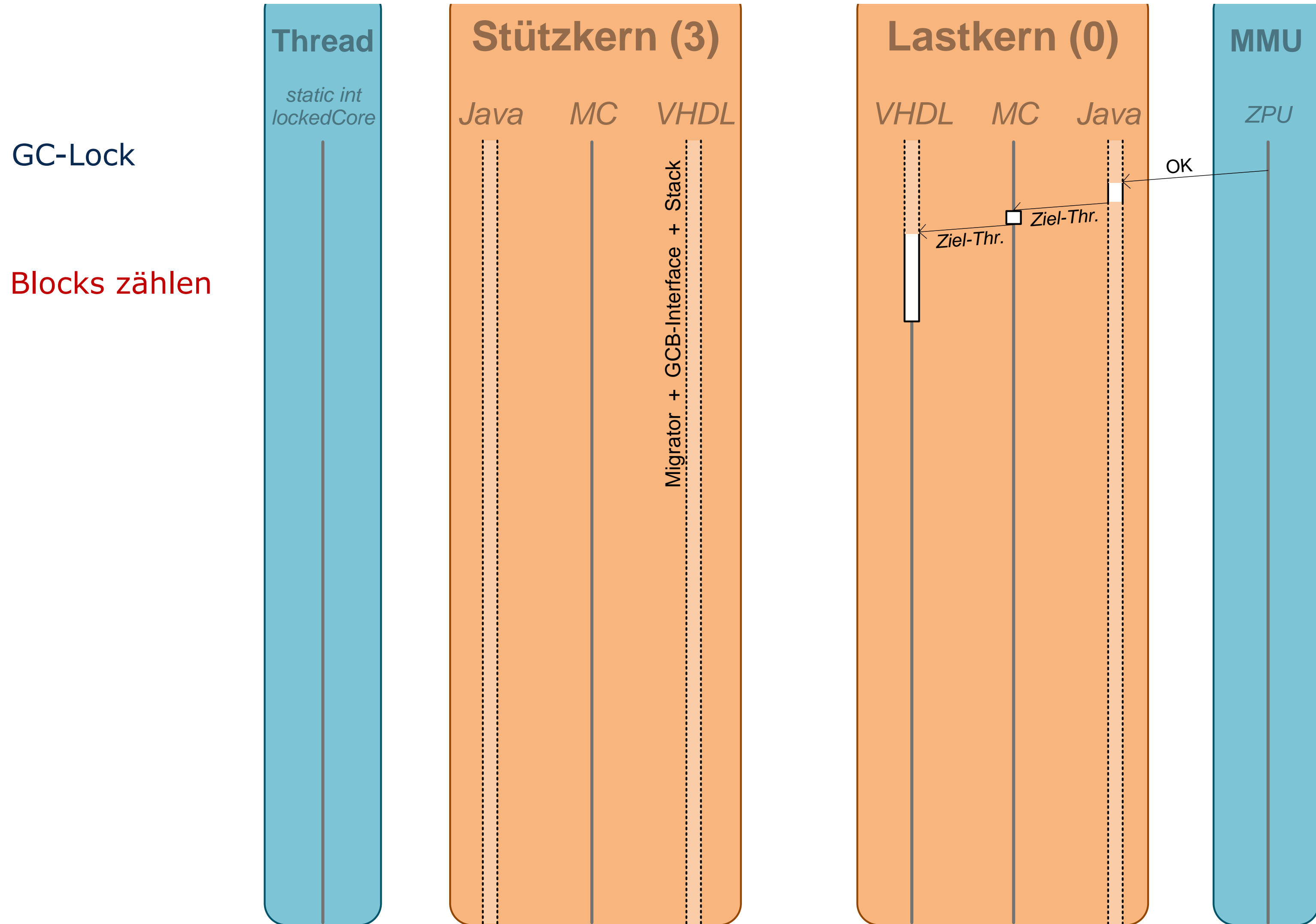
OK

TECHNISCHE UNIVERSITÄT DRESDEN

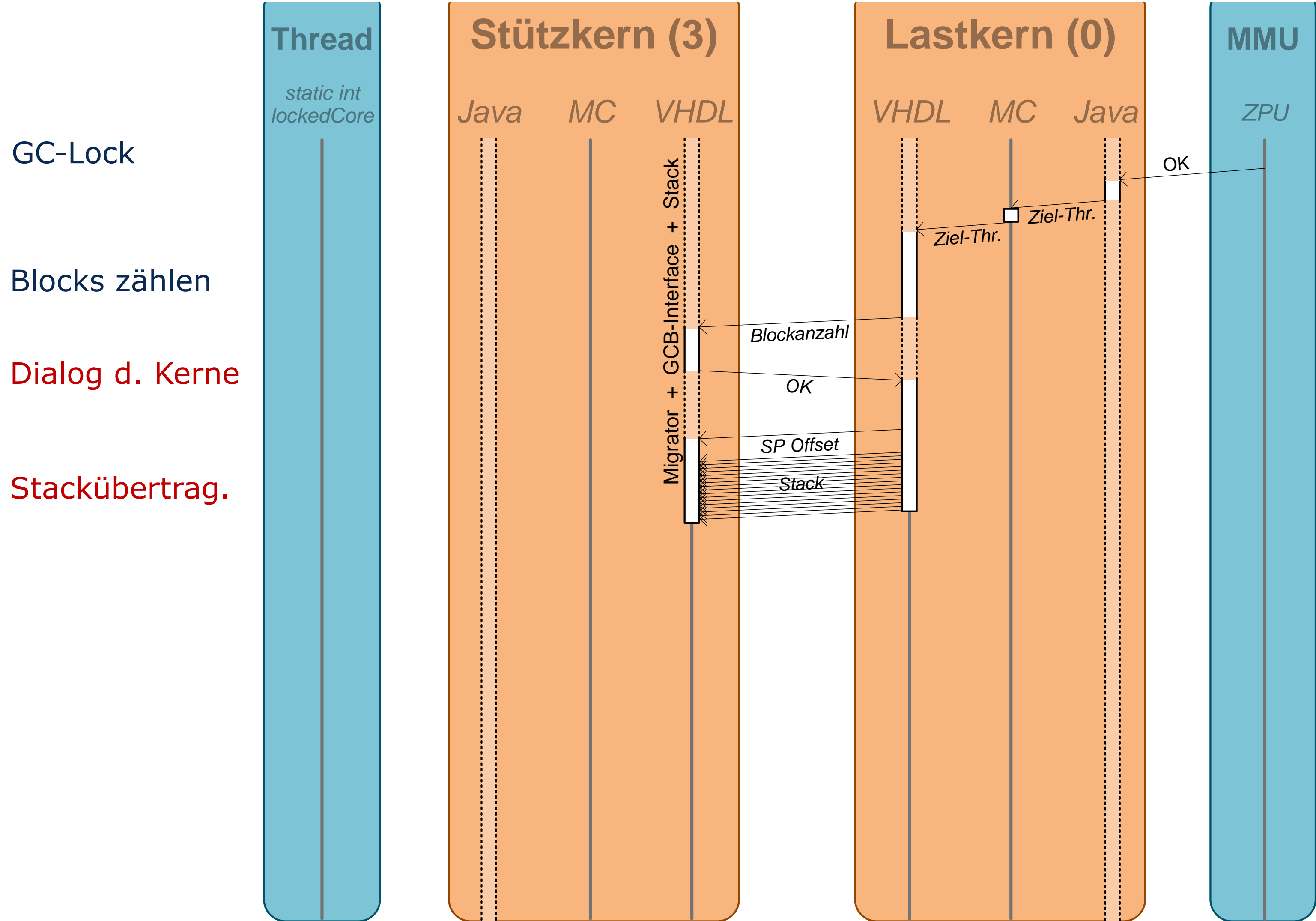
2.4 Konzept

GC-Lock

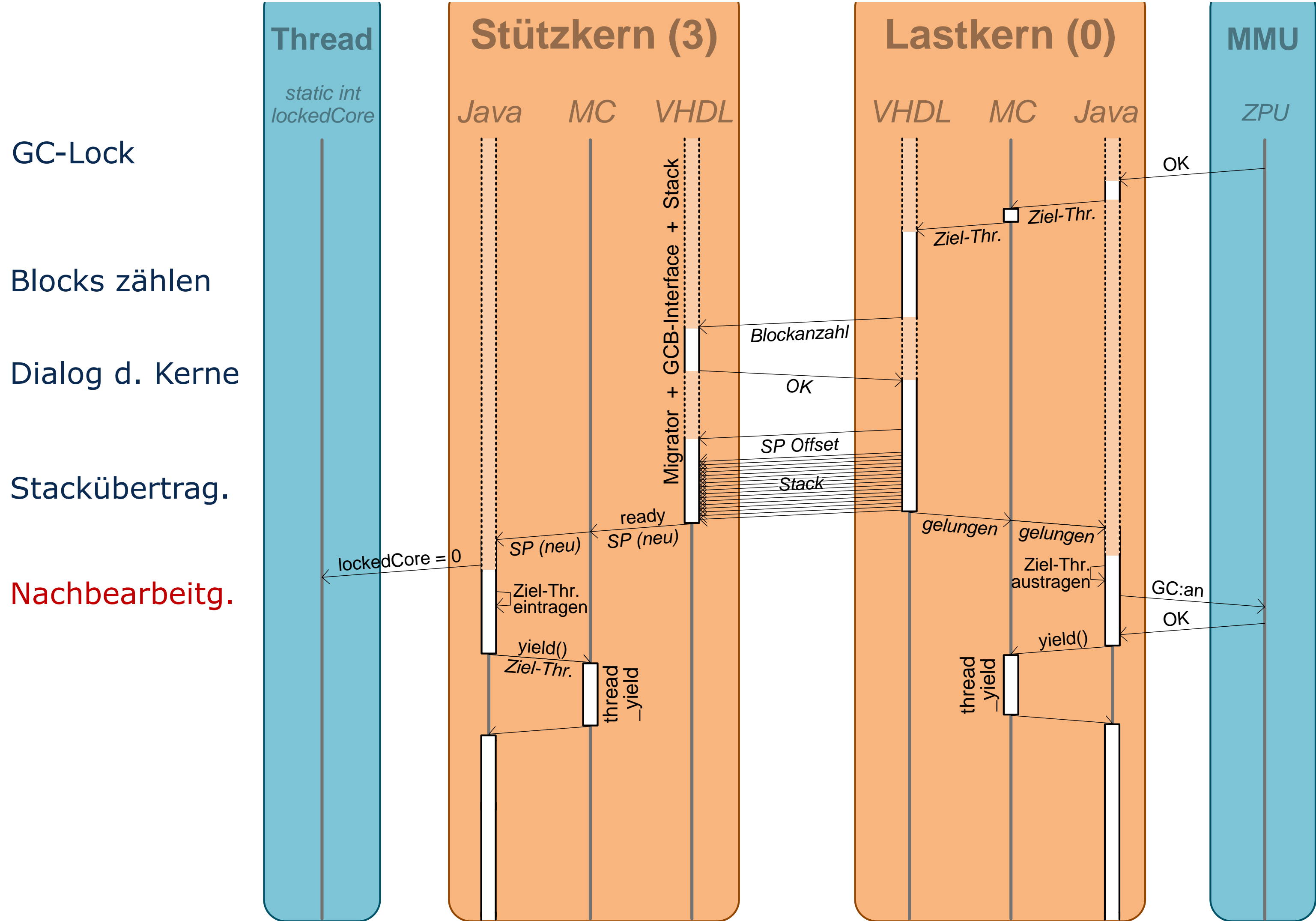
Blocks zählen



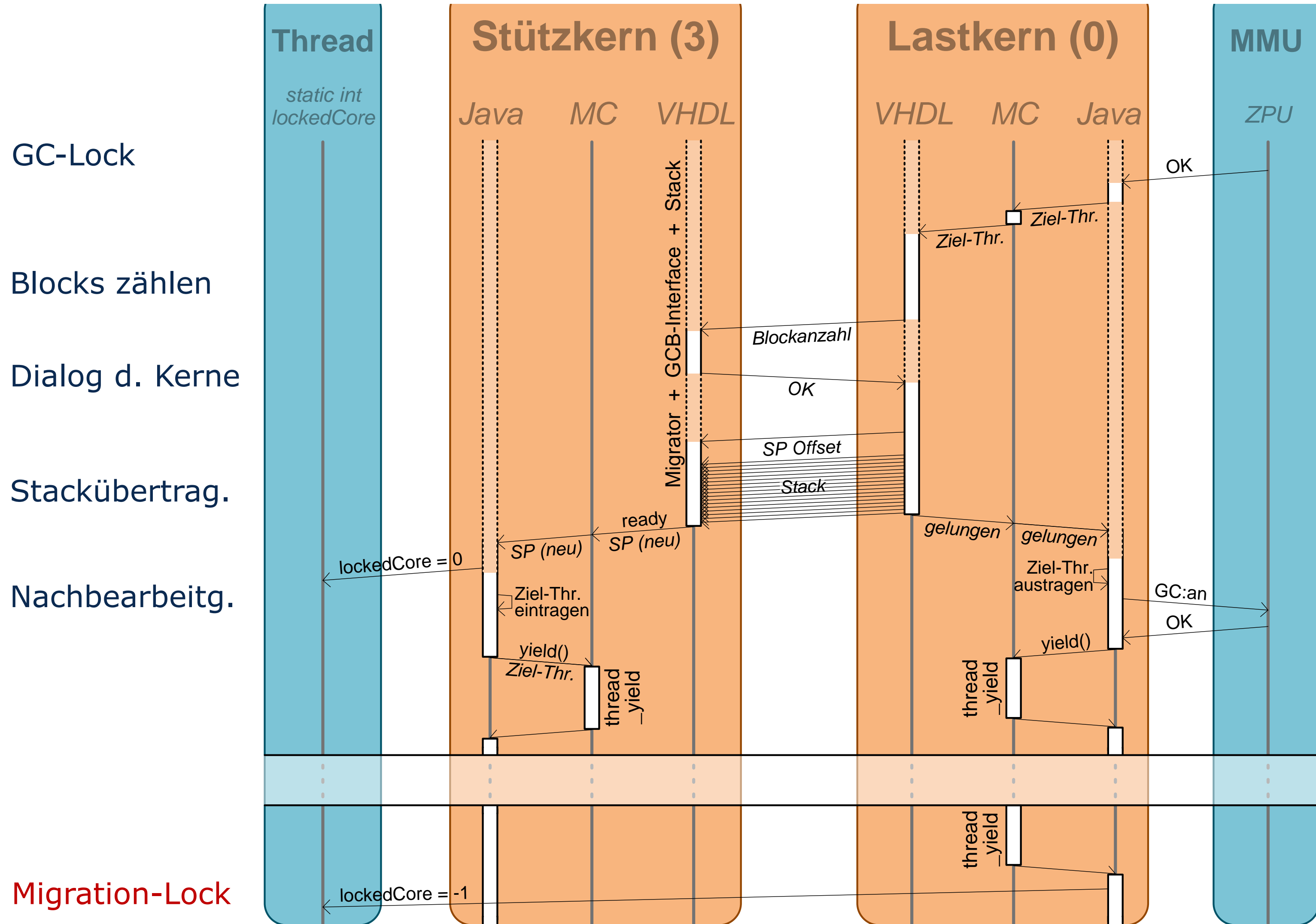
2.4 Konzept



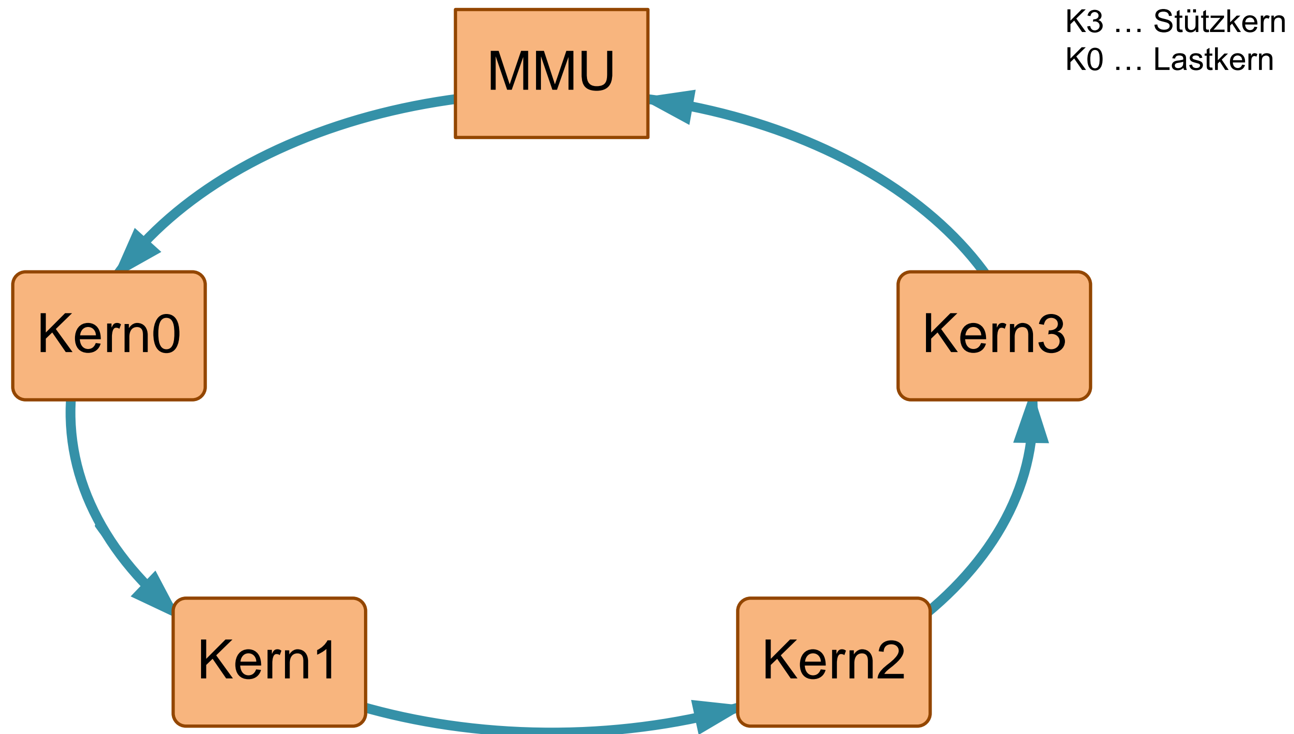
2.4 Konzept



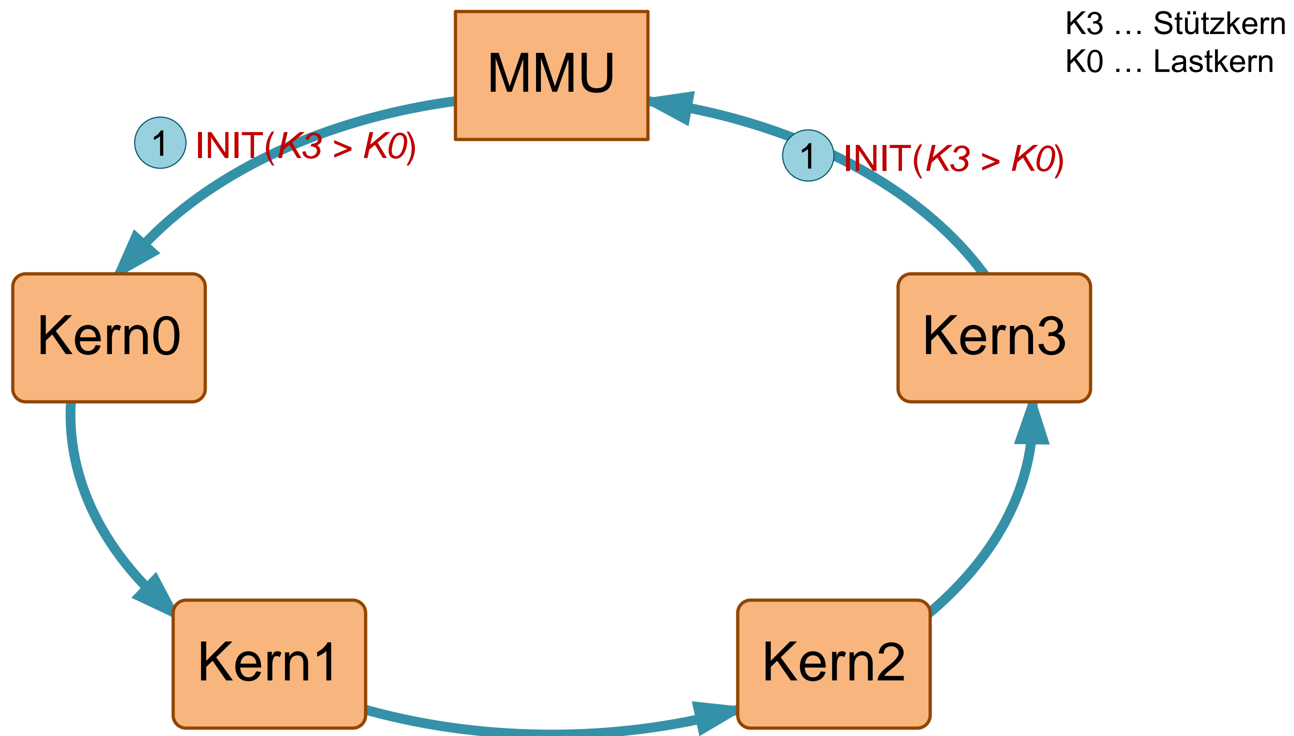
2.4 Konzept



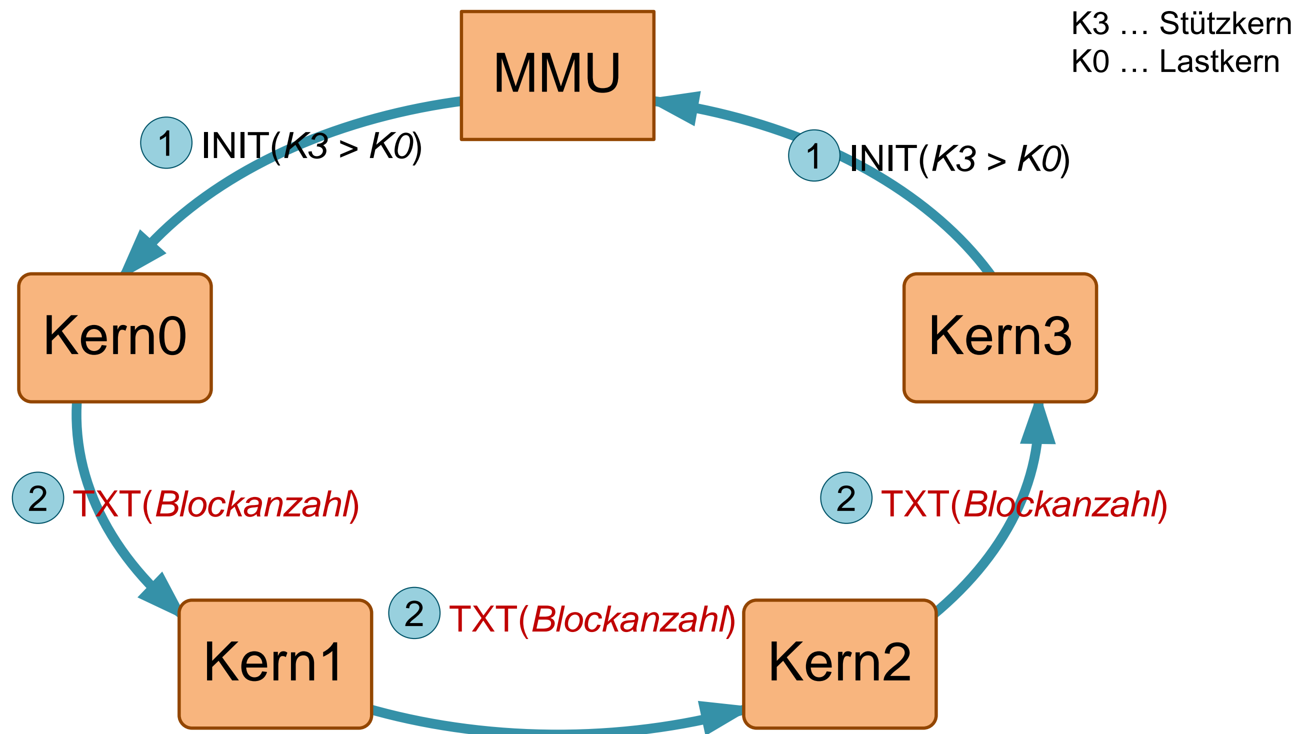
Kommunikation auf GC-Bus



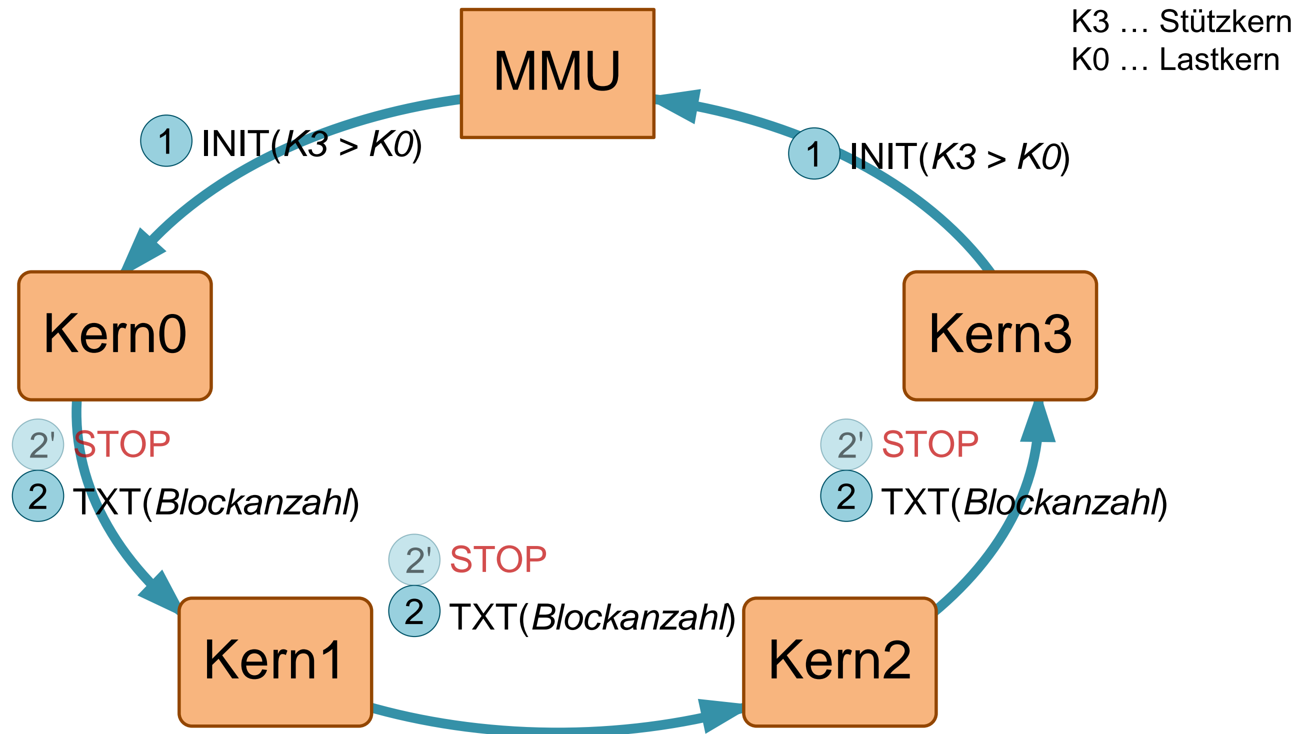
Kommunikation auf GC-Bus



Kommunikation auf GC-Bus

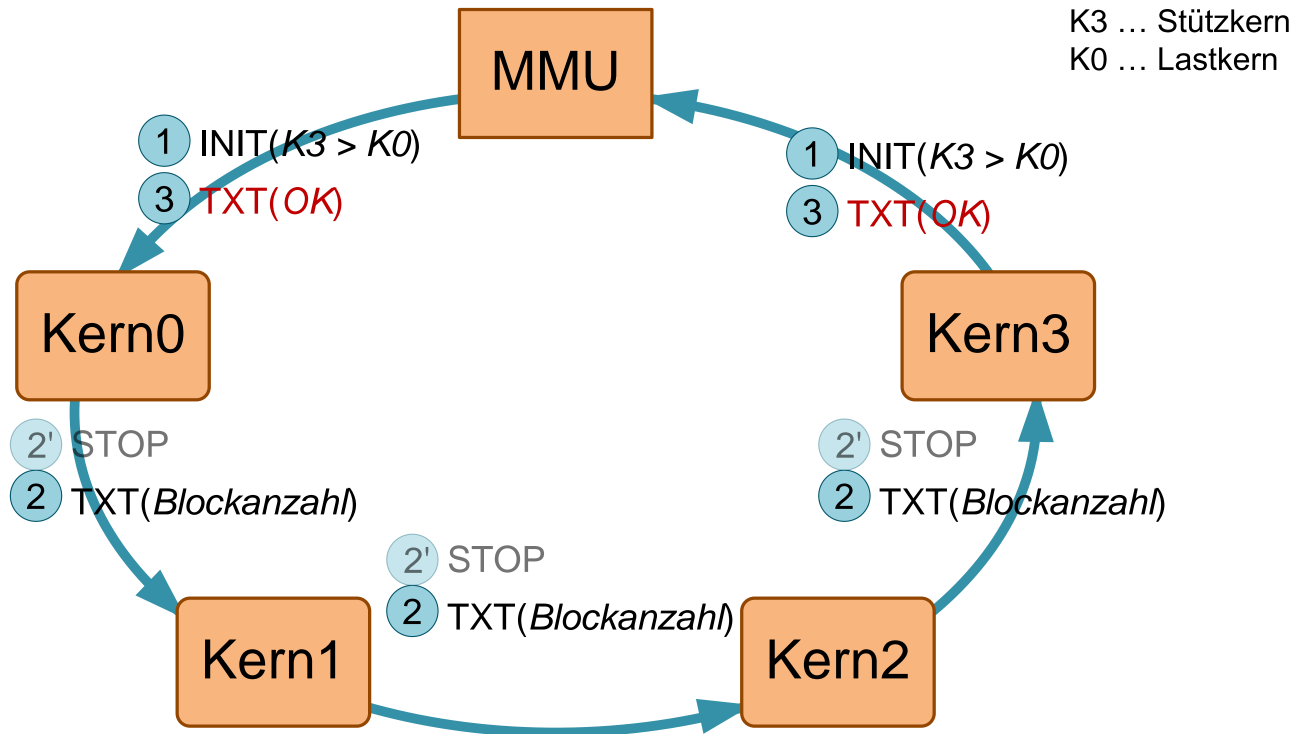


Kommunikation auf GC-Bus

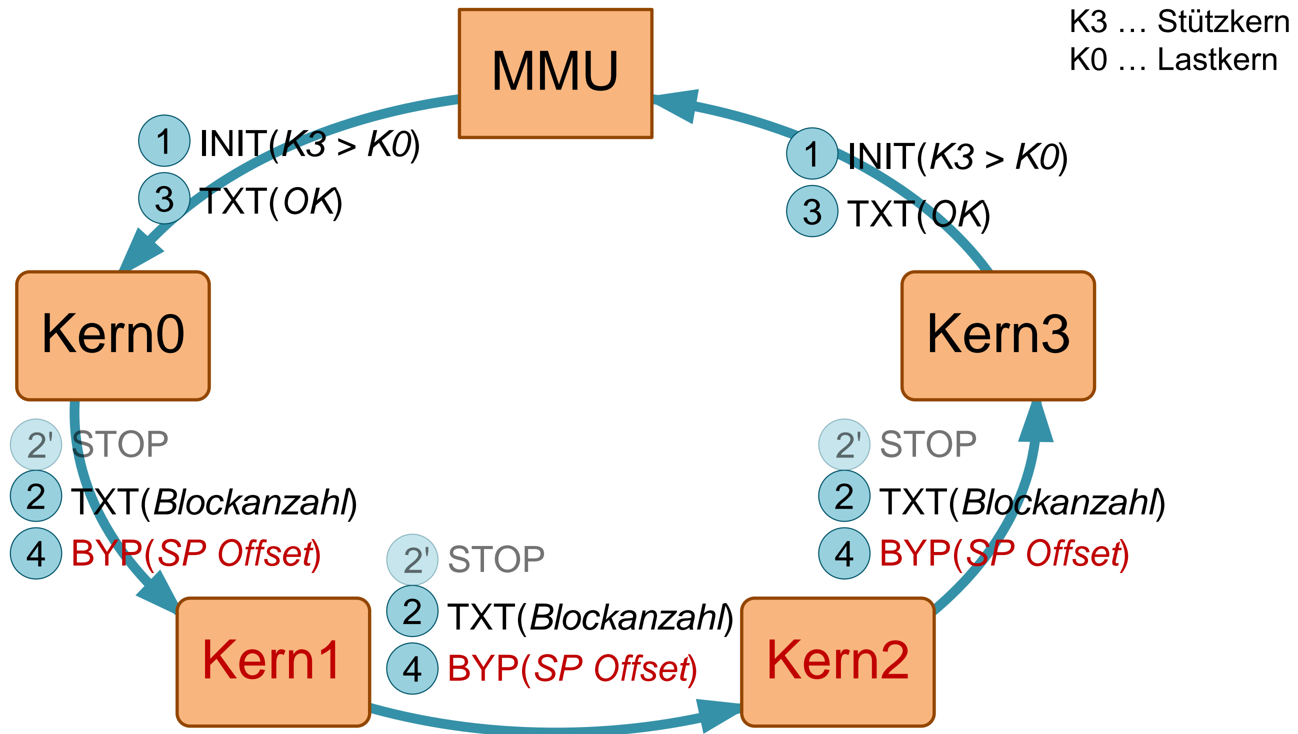


2.5 Dialog der Kerne

Kommunikation auf GC-Bus

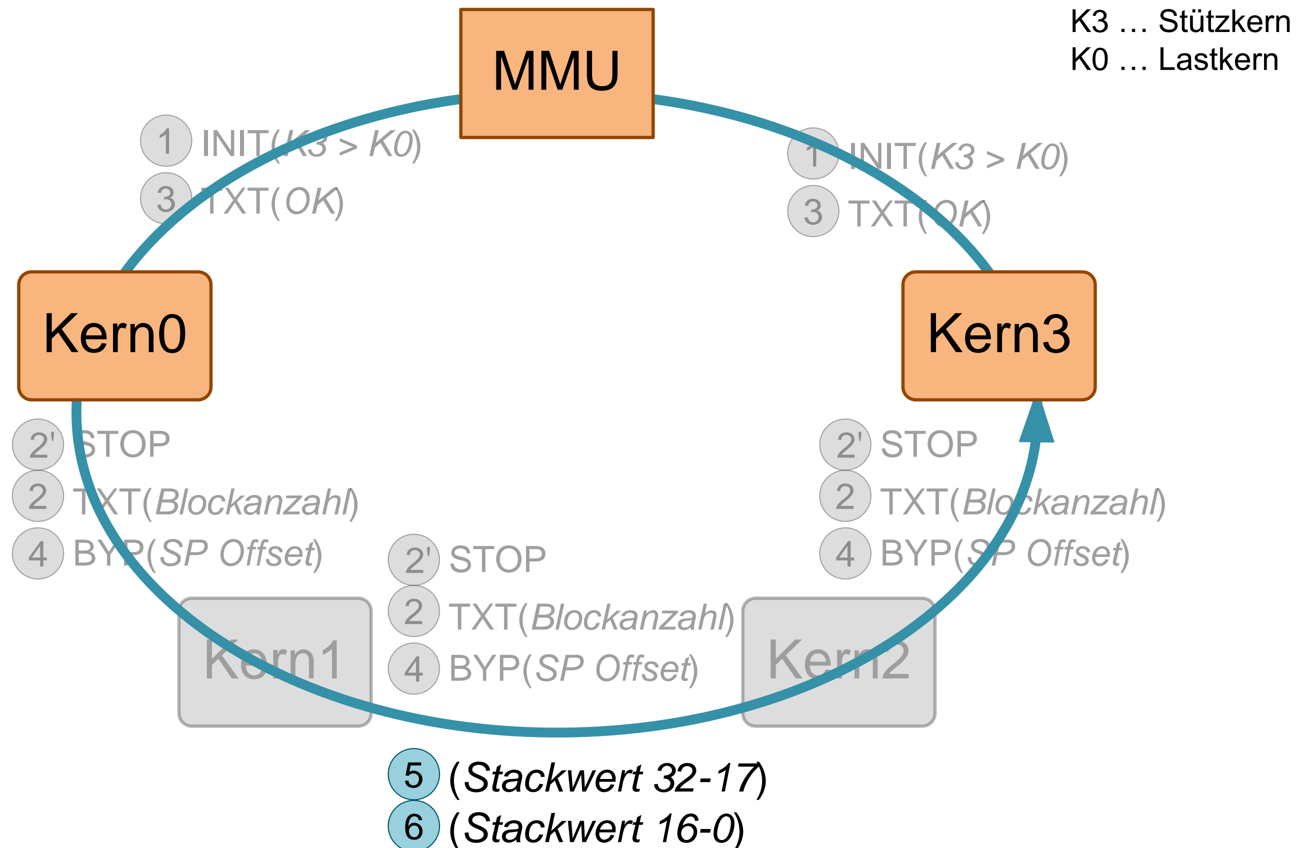


Kommunikation auf GC-Bus

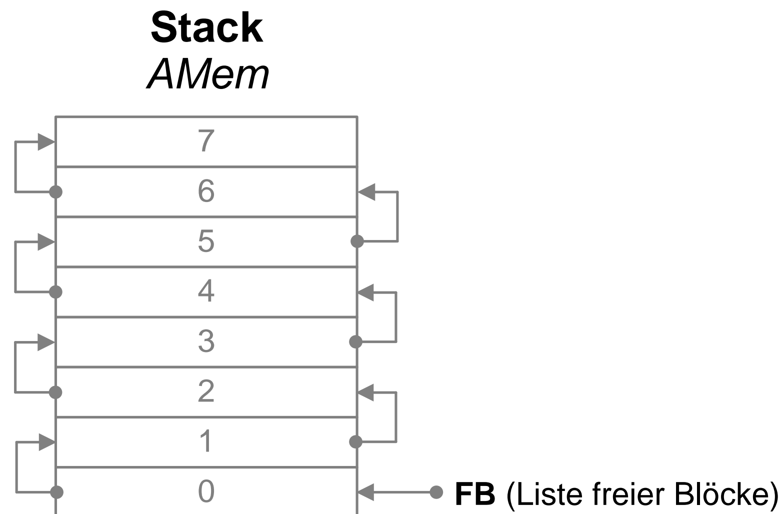


2.5 Dialog der Kerne

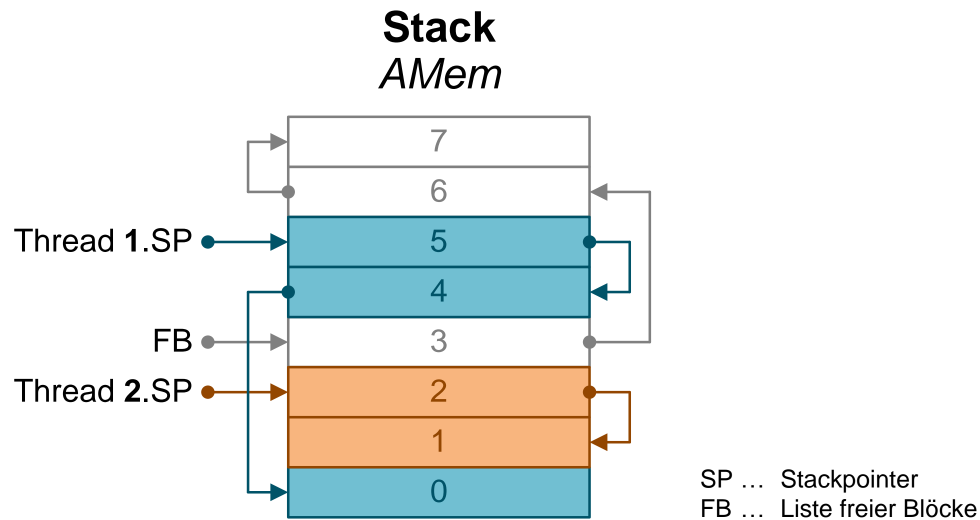
Kommunikation auf GC-Bus



Stackübertragung

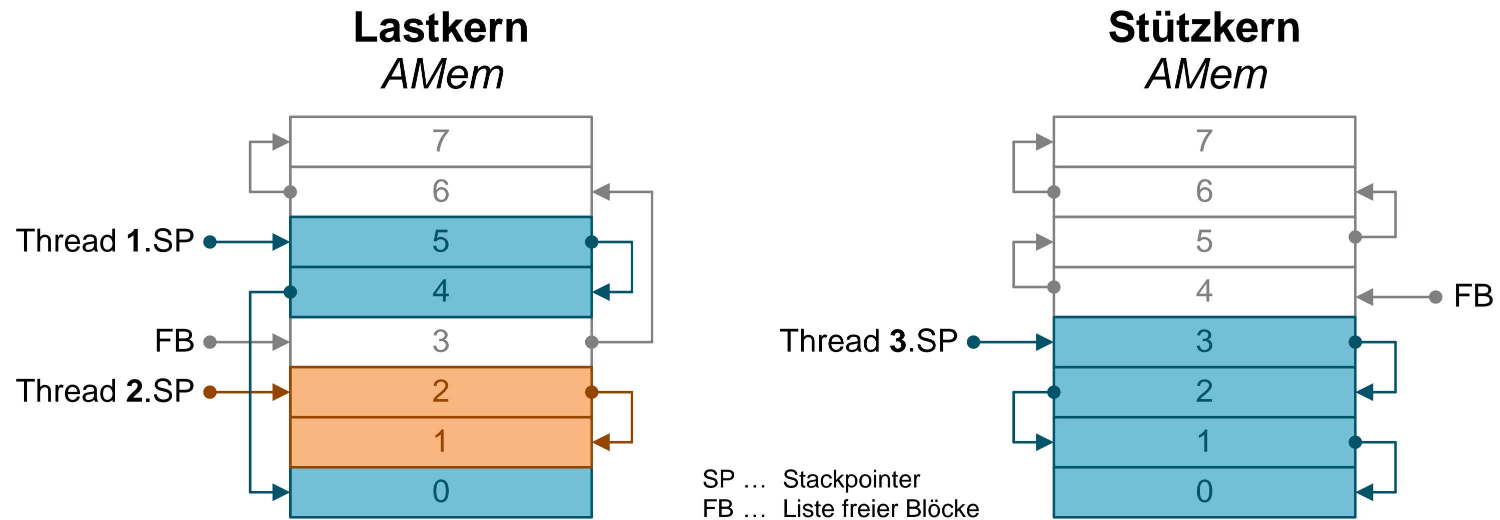


Stackübertragung

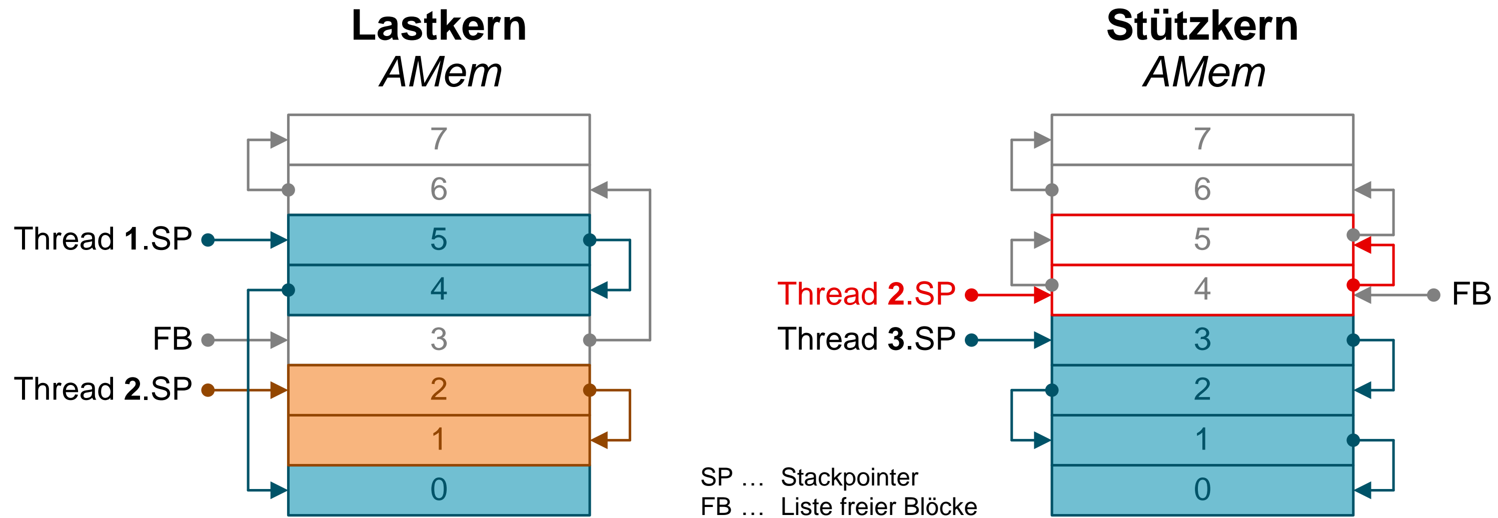


SP … Stackpointer
FB … Liste freier Blöcke

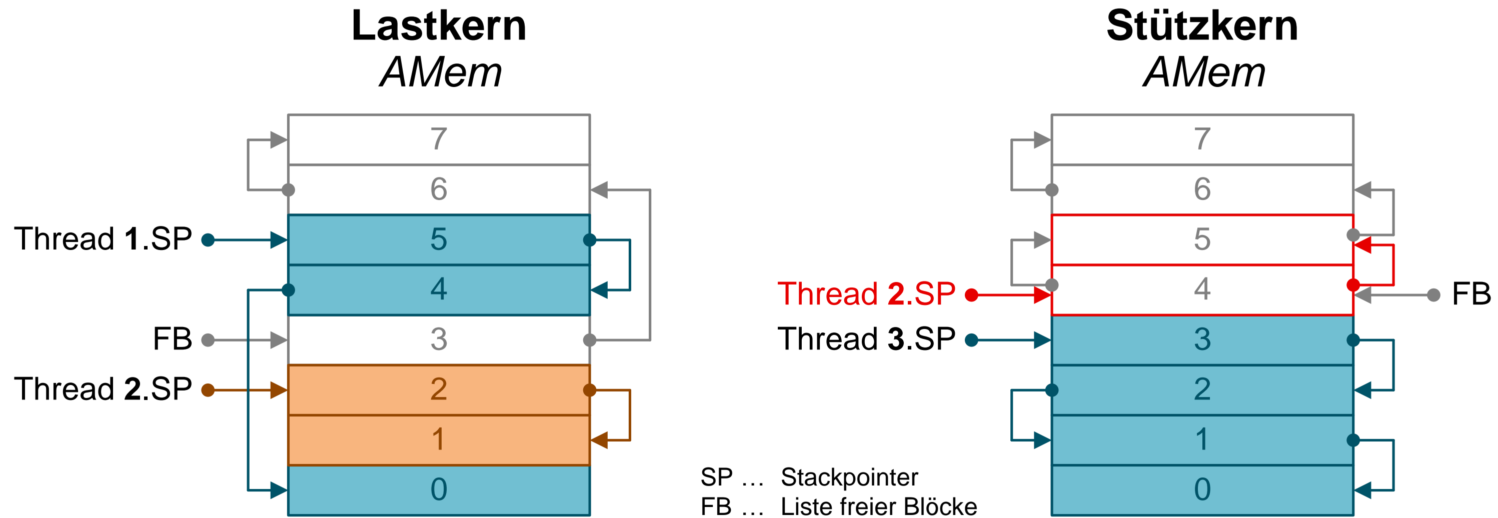
Stackübertragung



Stackübertragung

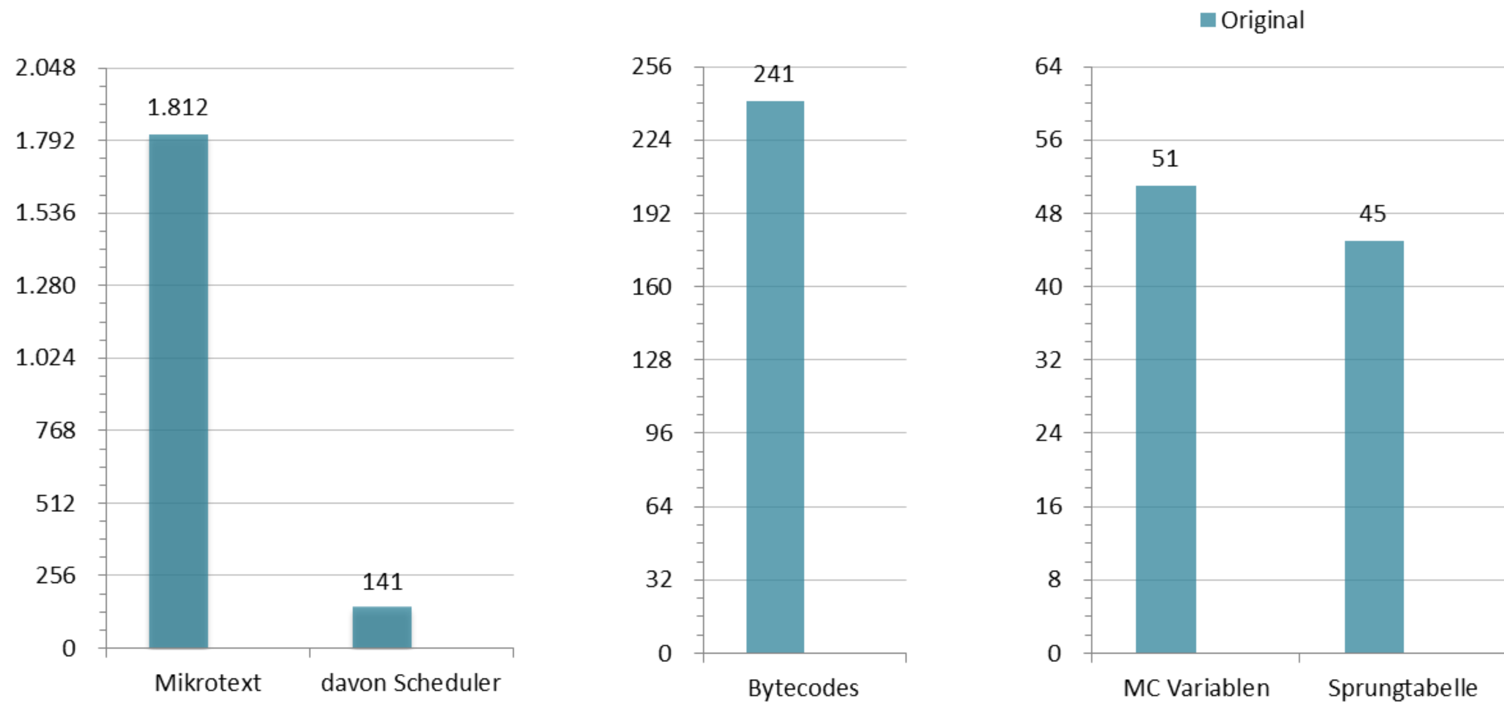


Stackübertragung



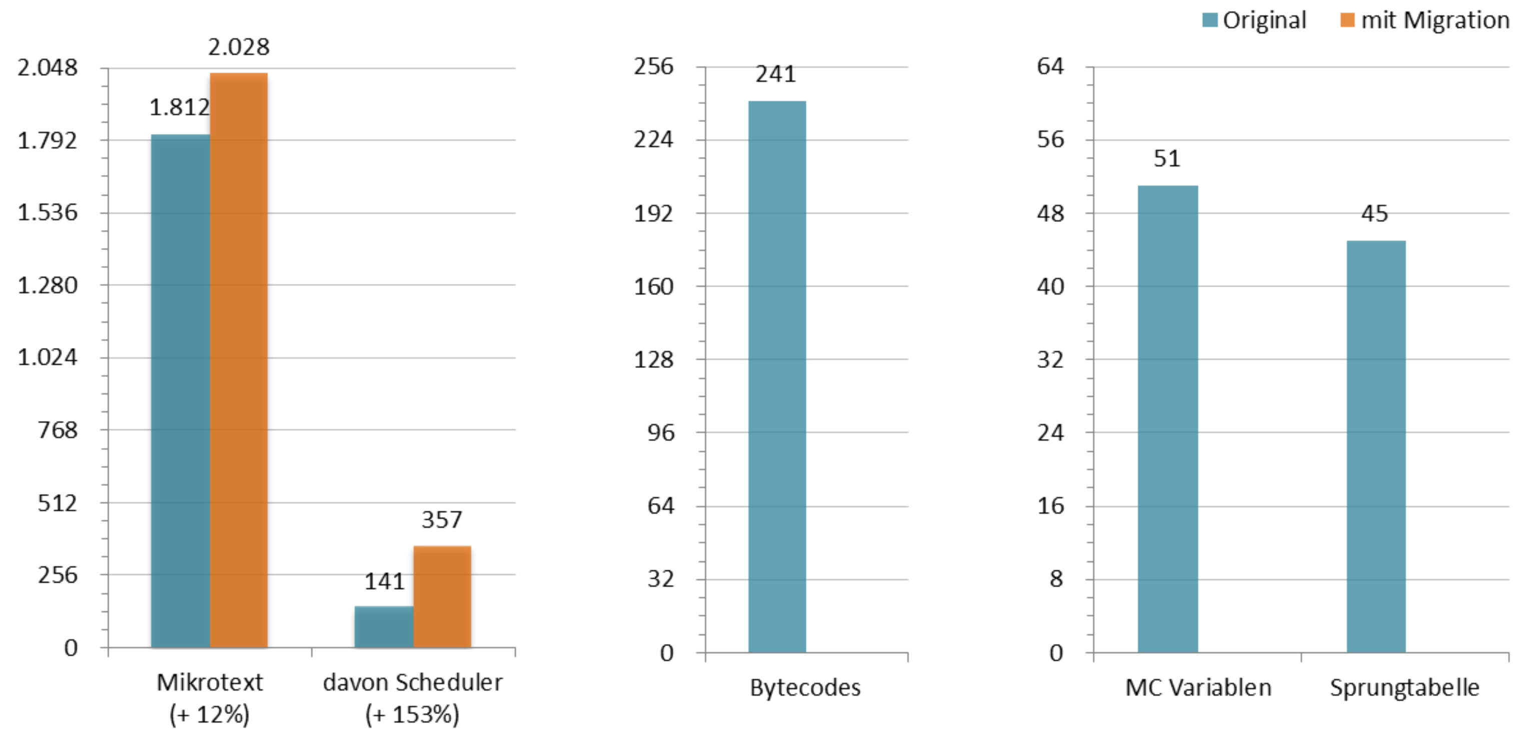
- Stackpointer anpassen
- Datenregister anpassen

Code



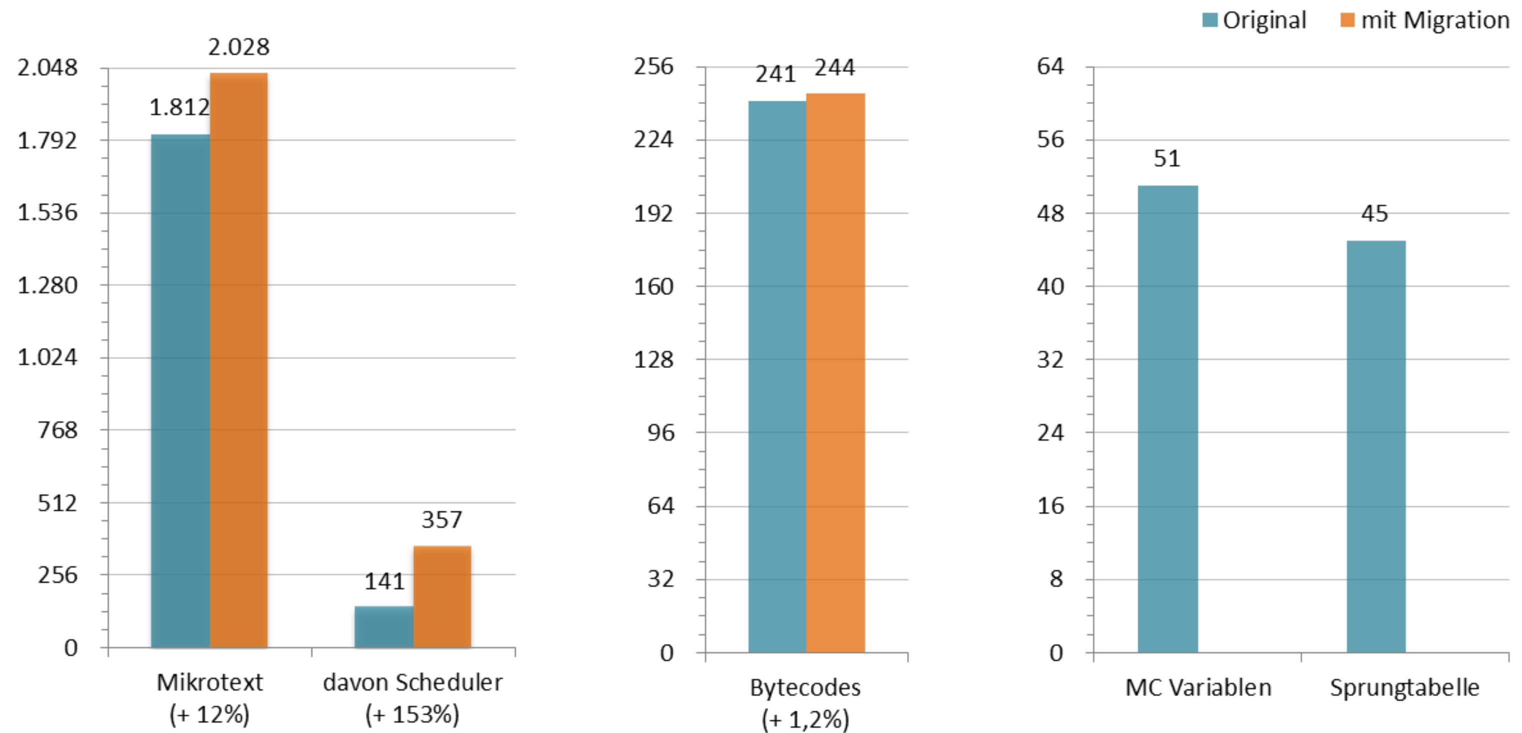
Mikrocode für Floating Point ist nicht enthalten

Code



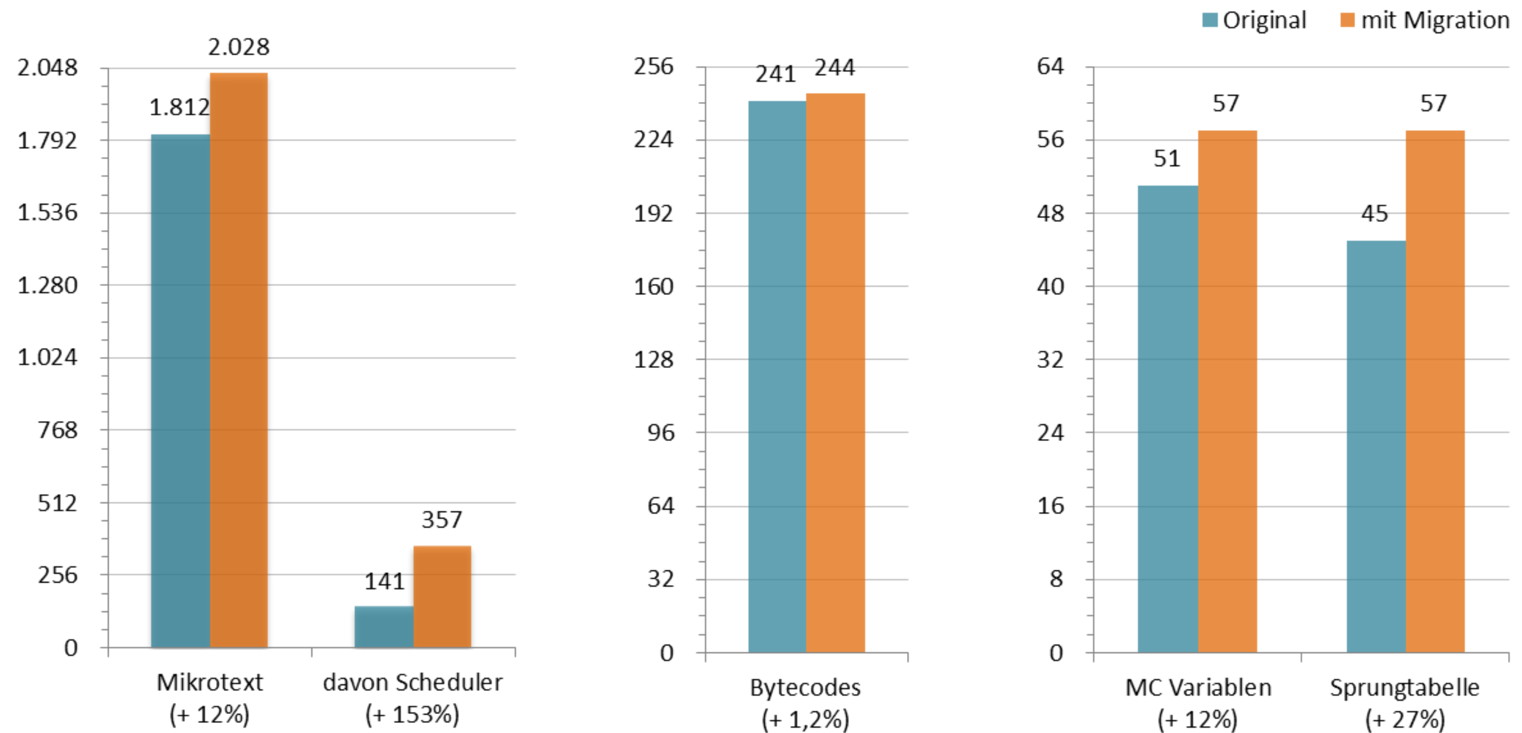
Mikrocode für Floating Point ist nicht enthalten

Code



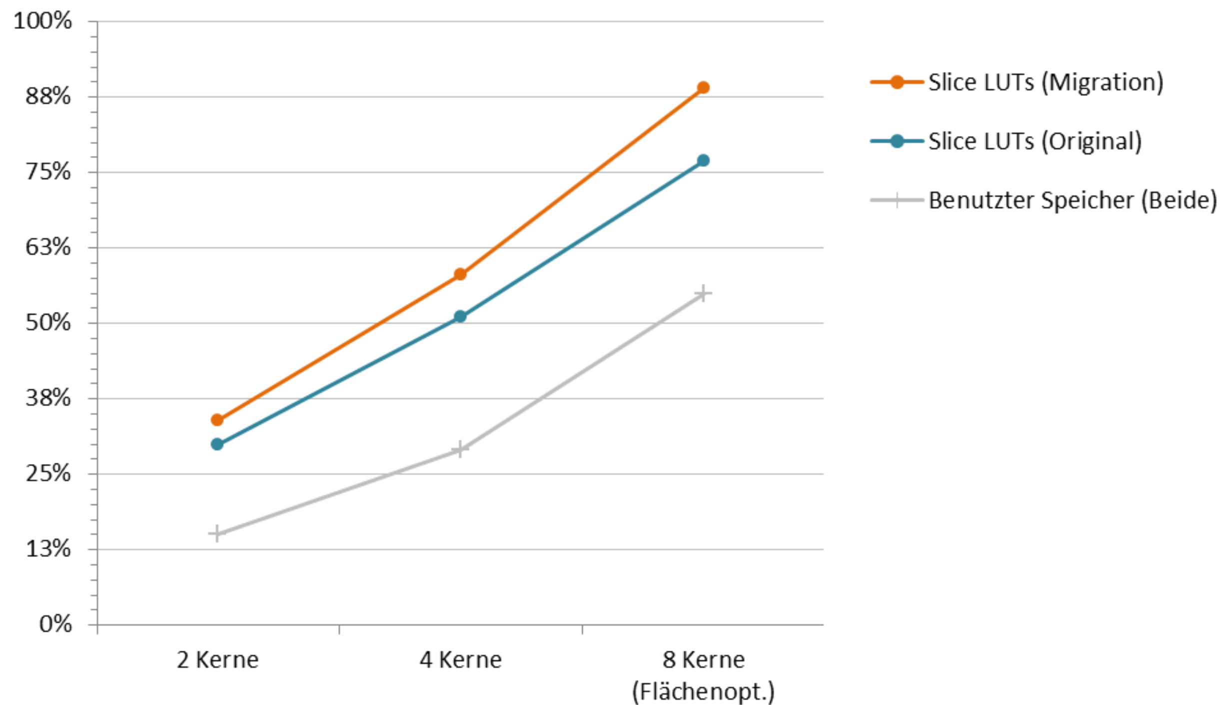
Mikrocode für Floating Point ist nicht enthalten

Code

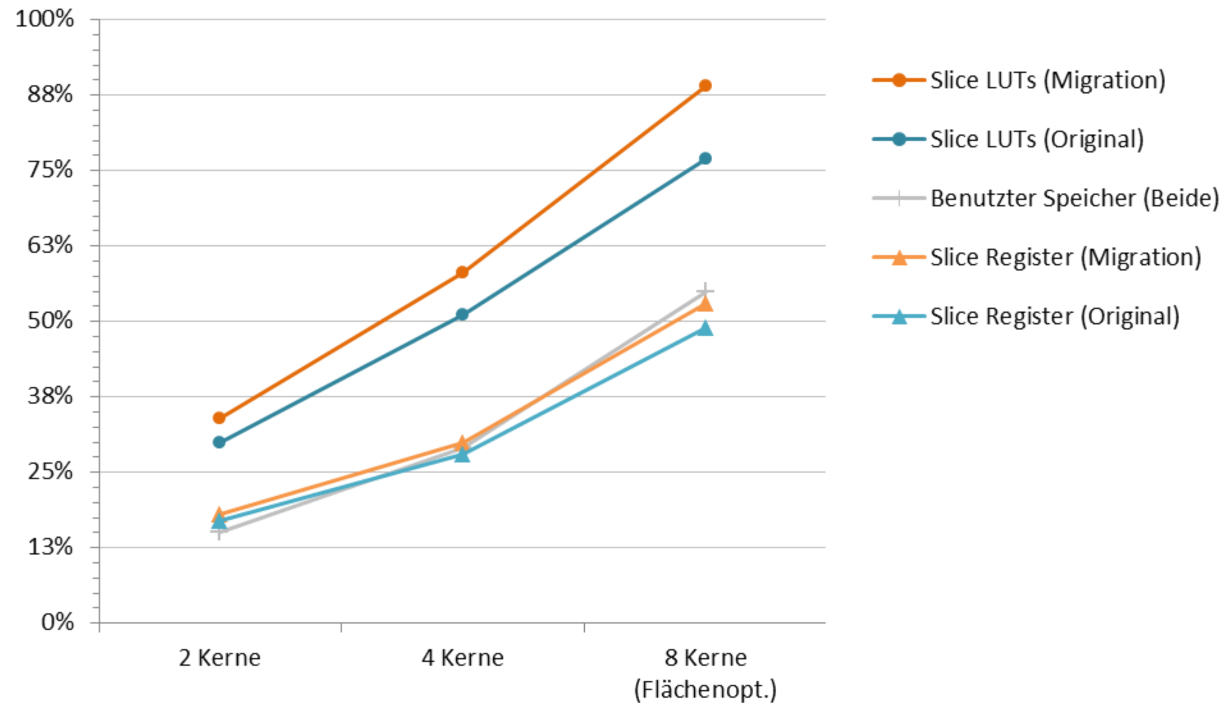


Mikrocode für Floating Point ist nicht enthalten

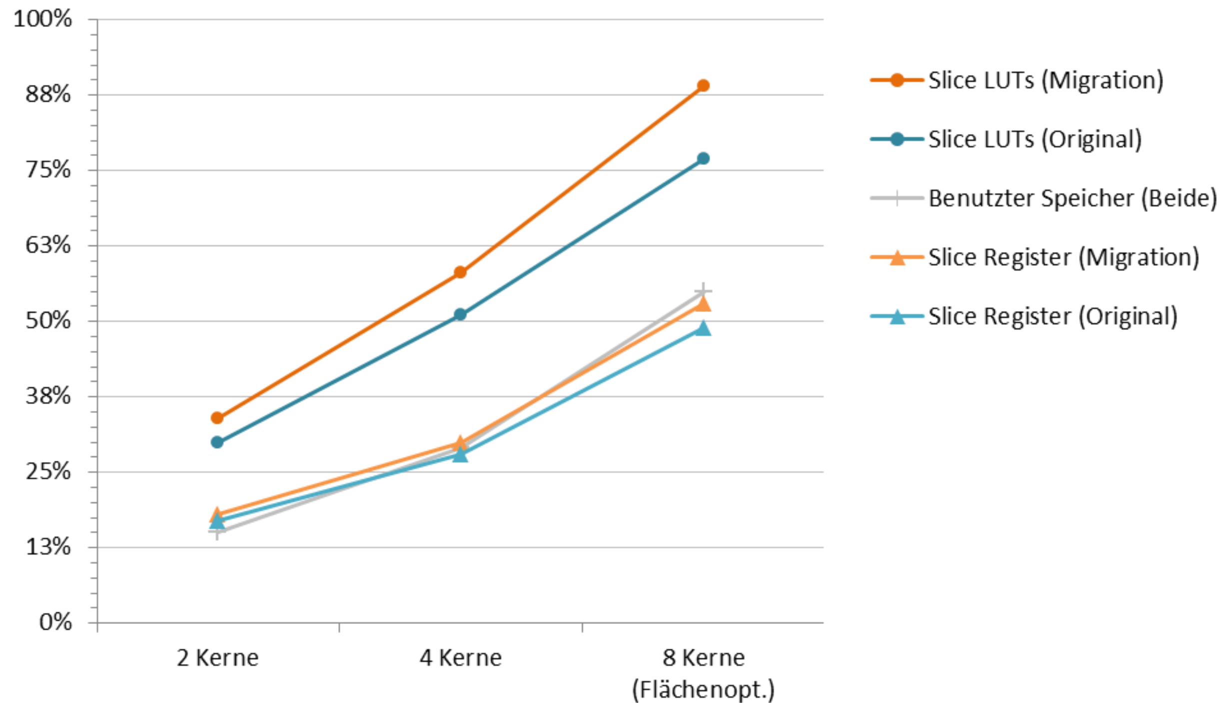
Hardware



Hardware



Hardware



$$\text{Verbrauch}_{\mathit{Original}} = 2848 \text{ LUTs} + \text{Kernanzahl} * 2989 \text{ LUTs}$$

$$\text{Verbrauch}_{\mathit{mit\ Migration}} = 2992 \text{ LUTs} + \text{Kernanzahl} * 3471 \text{ LUTs}$$

3.2 Leistung

Keine umfassende Analyse und Bewertung sinnvoll

Abschätzung

3.2 Leistung

Keine umfassende Analyse und Bewertung sinnvoll

Abschätzung

- Benchmarks
  - JGF-Crypt (angepasst)
  - JGF-SparseMatmult (angepasst)

3.2 Leistung

Keine umfassende Analyse und Bewertung sinnvoll

Abschätzung

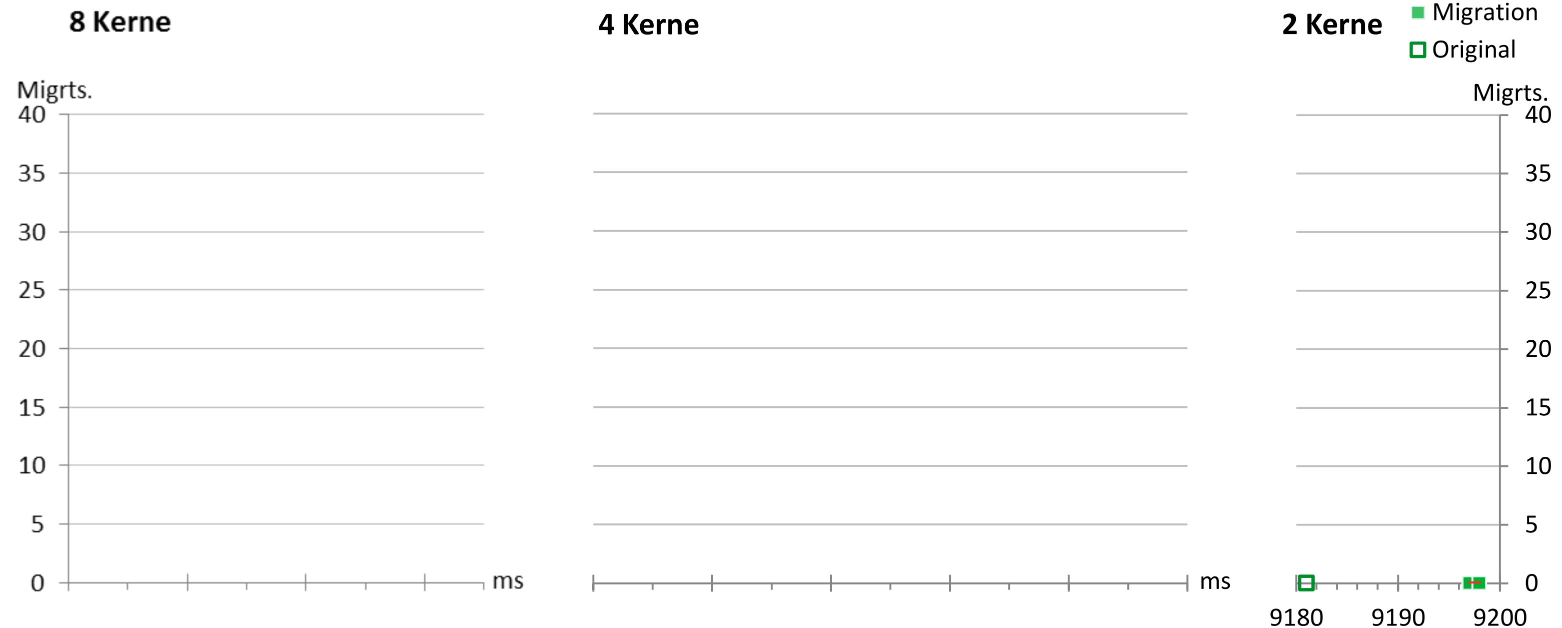
- Benchmarks
  - JGF-Crypt (angepasst)
  - JGF-SparseMatmult (angepasst)
- Auswahl-Strategien
  - Kern mit den meisten Threads
  - Thread mit den meisten verbleibenden Zeitscheiben
- Messung
  - Migrationsanzahl
  - mittlerer und maximaler Speed-Up

$$S_{p,\,mittel} = \frac{T_{Original,\,mittel}}{T_{Migration,\,mittel}}$$

$$S_{p,\,max} = \frac{T_{Original,\,max}}{T_{Migration,\,max}}$$

Ungünstigster Vergleich (ausgeglichenes System)

JGF-Crypt-Benchmark (Threadanzahl = 5 * Kernanzahl)



Ungünstigster Vergleich (ausgeglichenes System)

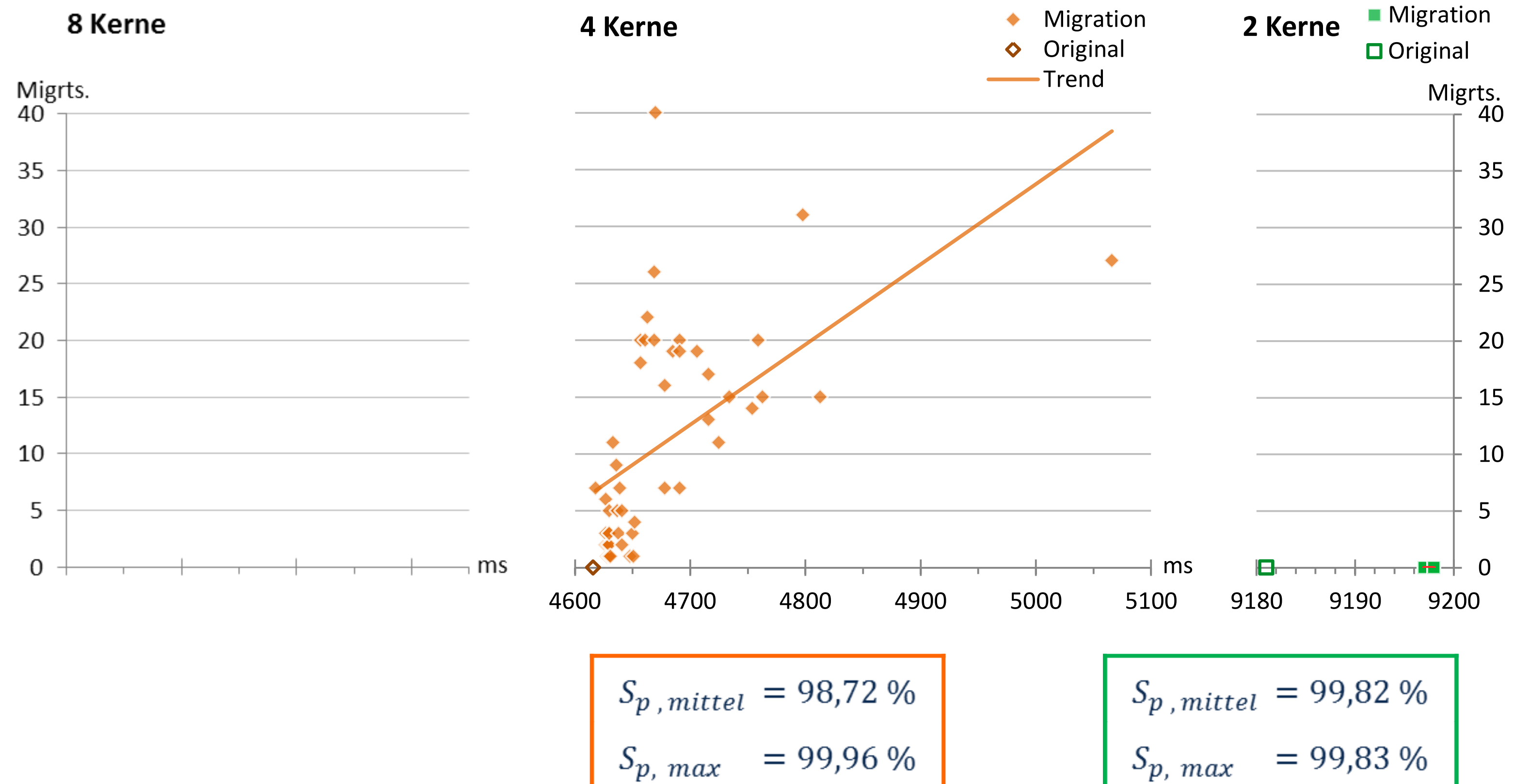
JGF-Crypt-Benchmark (Threadanzahl = 5 * Kernanzahl)

$$S_{p\,,\,mittel} = 99{,}82\ \%$$

$$S_{p,\ max} = 99{,}83\ \%$$

Ungünstigster Vergleich (ausgeglichenes System)

JGF-Crypt-Benchmark (Threadanzahl = 5 * Kernanzahl)

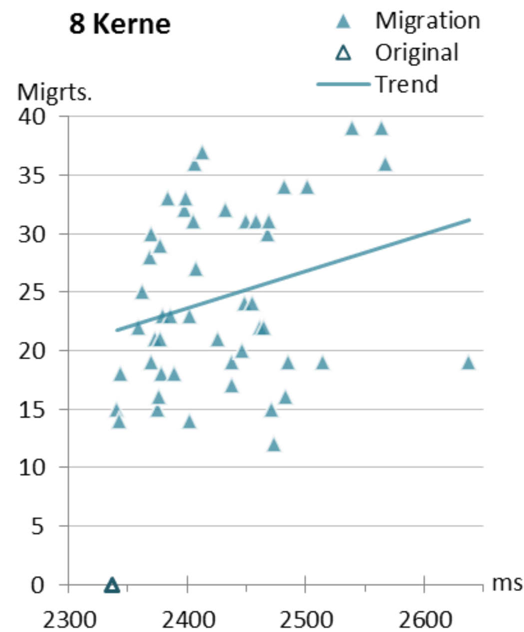


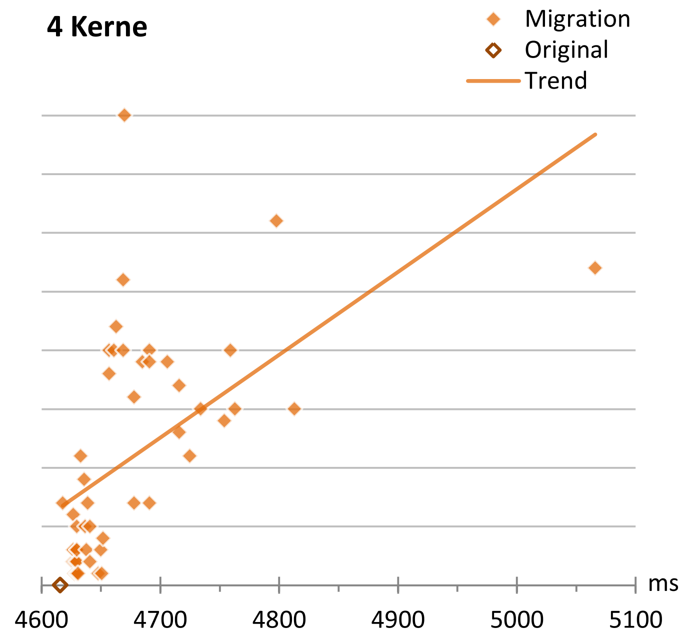
$$S_{p\,,mittel} = 98{,}72\ \%$$
$$S_{p,\,max} = 99{,}96\ \%$$

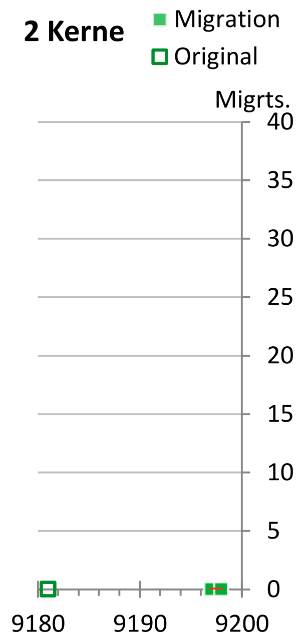
$$S_{p\,,mittel} = 99{,}82\ \%$$
$$S_{p,\,max} = 99{,}83\ \%$$

Ungünstigster Vergleich (ausgeglichenes System)

JGF-Crypt-Benchmark (Threadanzahl = 5 * Kernanzahl)







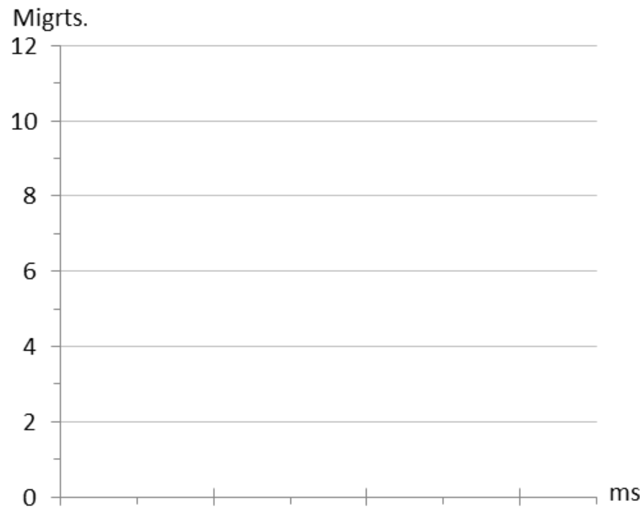
$S_{p,mittel} = 96{,}20\ \%$
$S_{p,max} = 99{,}83\ \%$

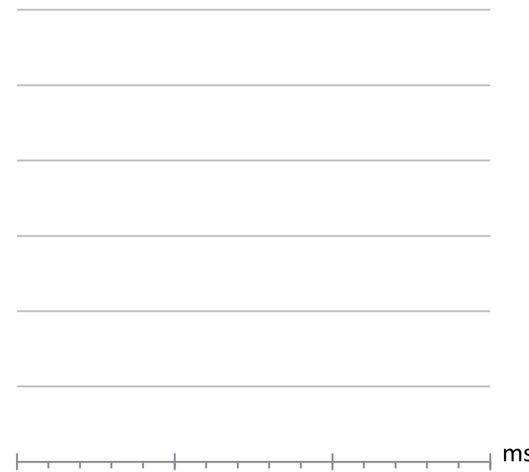
$S_{p,mittel} = 98{,}72\ \%$
$S_{p,max} = 99{,}96\ \%$

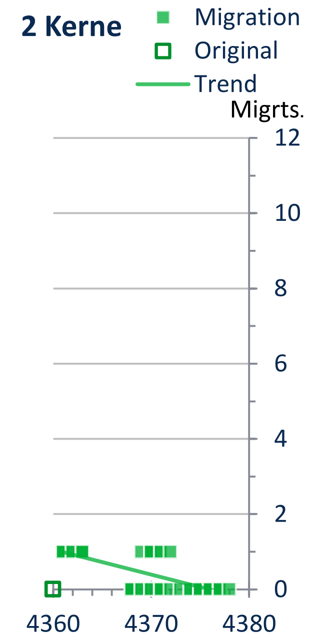
$S_{p,mittel} = 99{,}82\ \%$
$S_{p,max} = 99{,}83\ \%$

Ungünstigster Vergleich (ausgeglichenes System)

JGF-SparseMatmult-Benchmark (Threadanzahl = 10 * Kernanzahl)





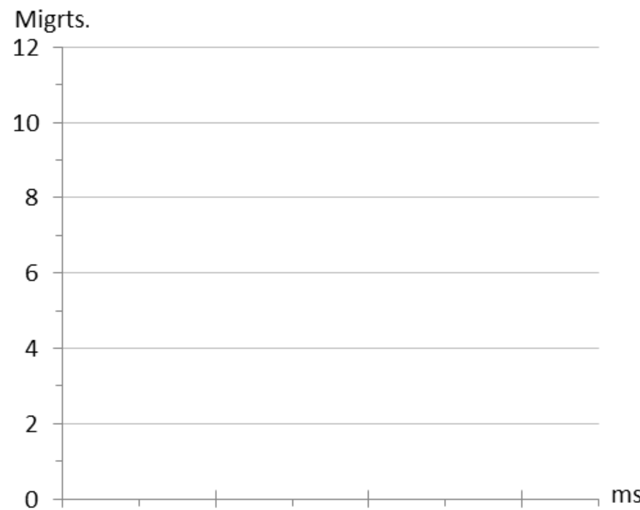


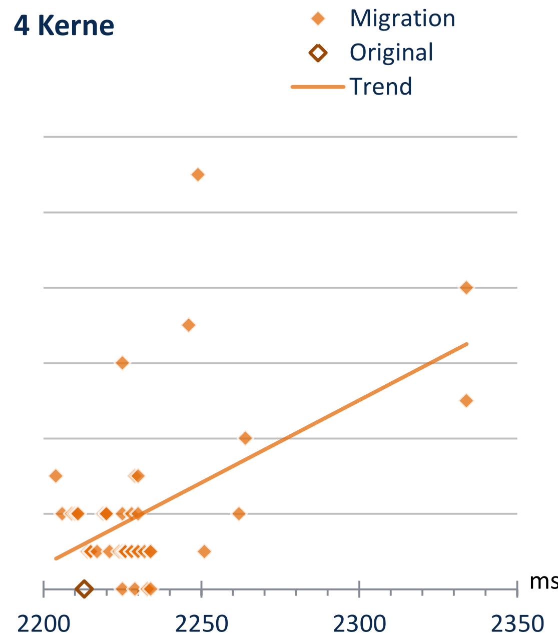
$S_{p,mittel} = 99,74\,\%$

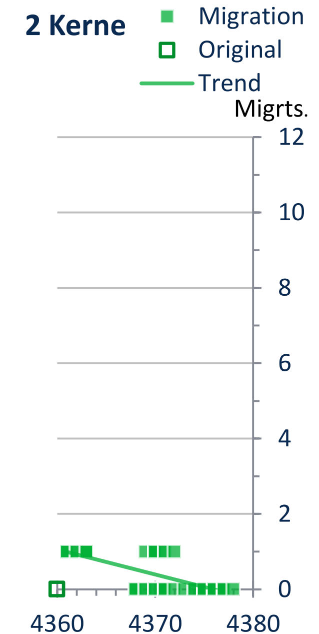
$S_{p,max} = 99,98\,\%$

Ungünstigster Vergleich (ausgeglichenes System)

JGF-SparseMatmult-Benchmark (Threadanzahl = 10 * Kernanzahl)







$S_{p,mittel} = 99{,}20\ \%$

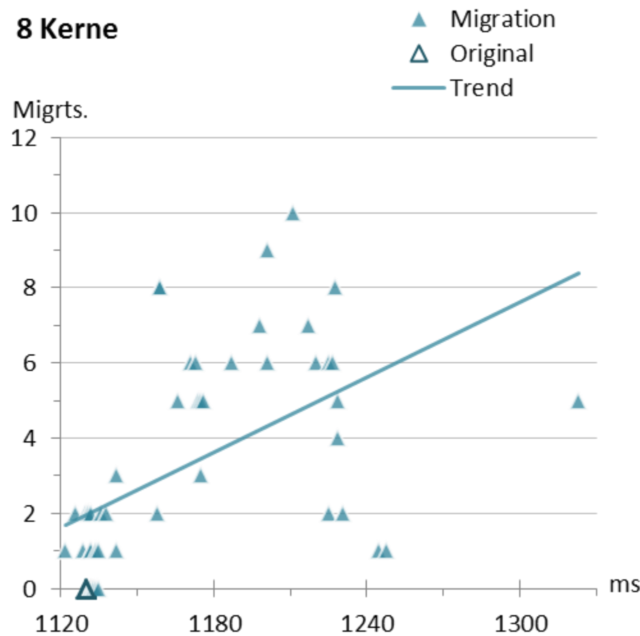
$S_{p,max} = 100{,}41\ \%$

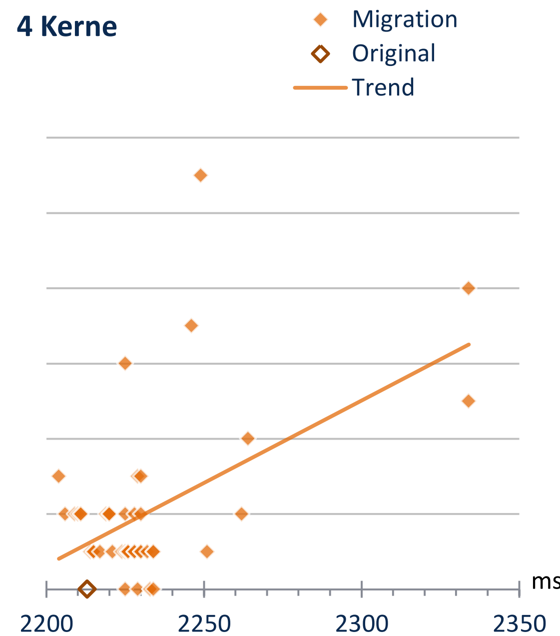
$S_{p,mittel} = 99{,}74\ \%$

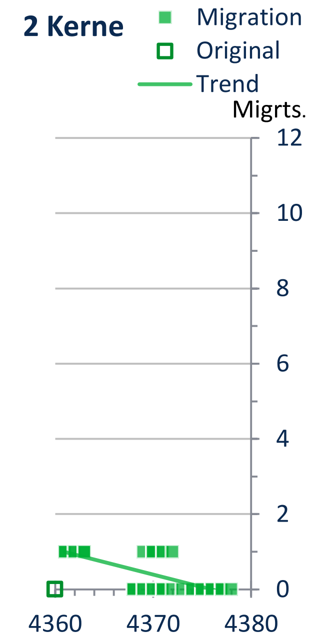
$S_{p,max} = 99{,}98\ \%$

Ungünstigster Vergleich (ausgeglichenes System)

JGF-SparseMatmult-Benchmark (Threadanzahl = 10 * Kernanzahl)



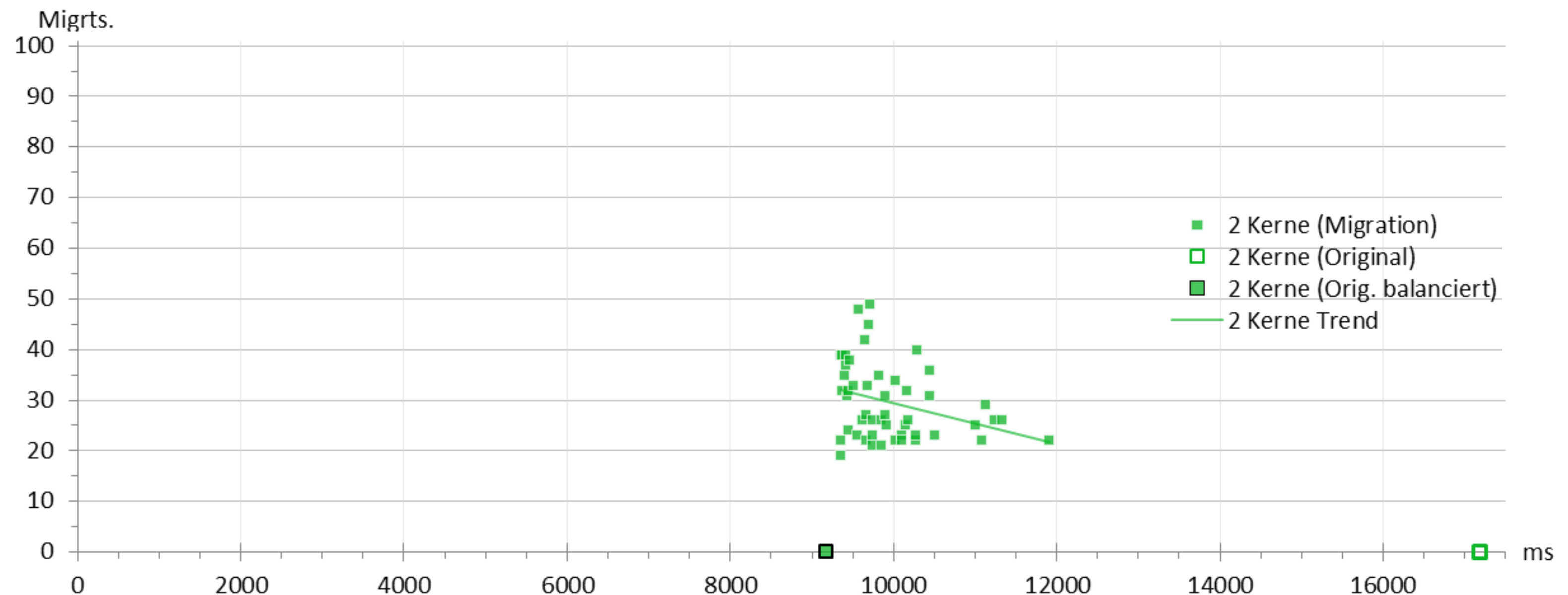




8 Kerne	4 Kerne	2 Kerne
$S_{p,mittel} = 96{,}31\,\%$	$S_{p,mittel} = 99{,}20\,\%$	$S_{p,mittel} = 99{,}74\,\%$
$S_{p,max} = 100{,}71\,\%$	$S_{p,max} = 100{,}41\,\%$	$S_{p,max} = 99{,}98\,\%$

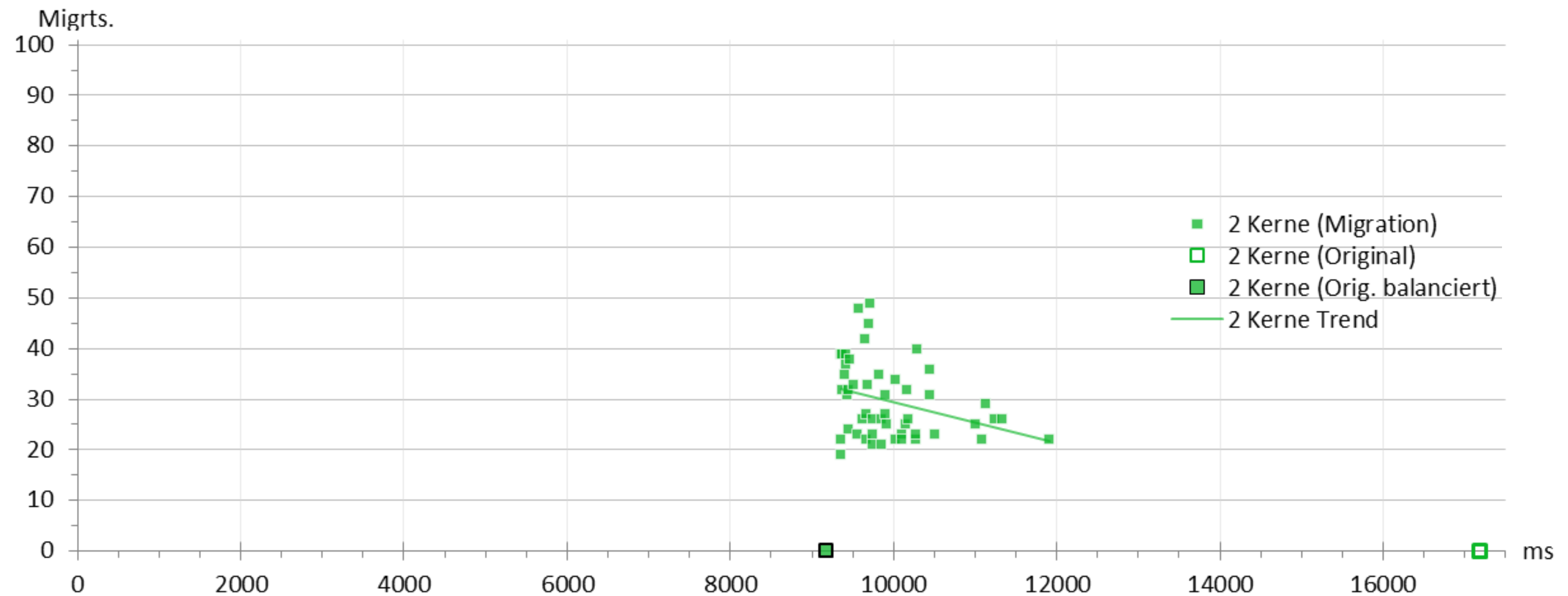
Günstigster Vergleich (unausgeglichenes System)

JGF-Crypt-Benchmark (16 Threads)



Günstigster Vergleich (unausgeglichenes System)

JGF-Crypt-Benchmark (16 Threads)



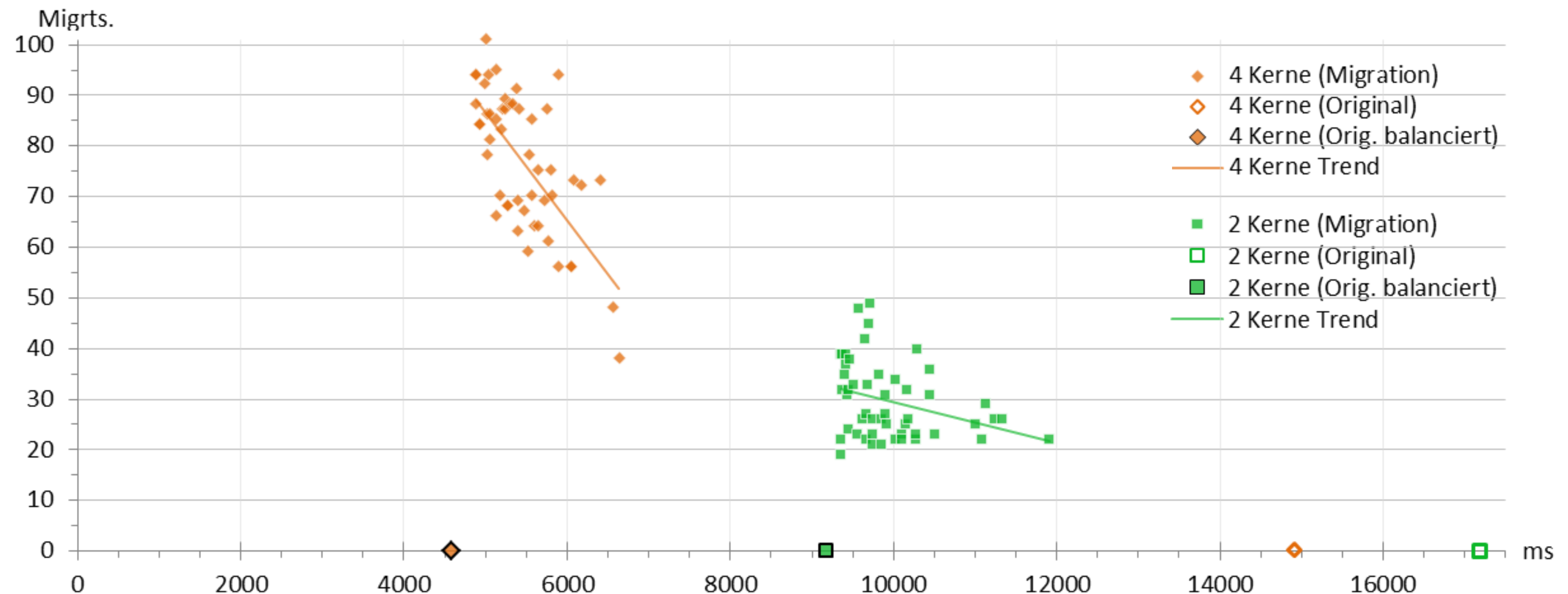
$S_{p,\,mittel} = 1{,}725$

$S_{p,\,max} = 1{,}837$

$S_{p,\,theor} = 1{,}875$

Günstigster Vergleich (unausgeglichenes System)

JGF-Crypt-Benchmark (16 Threads)

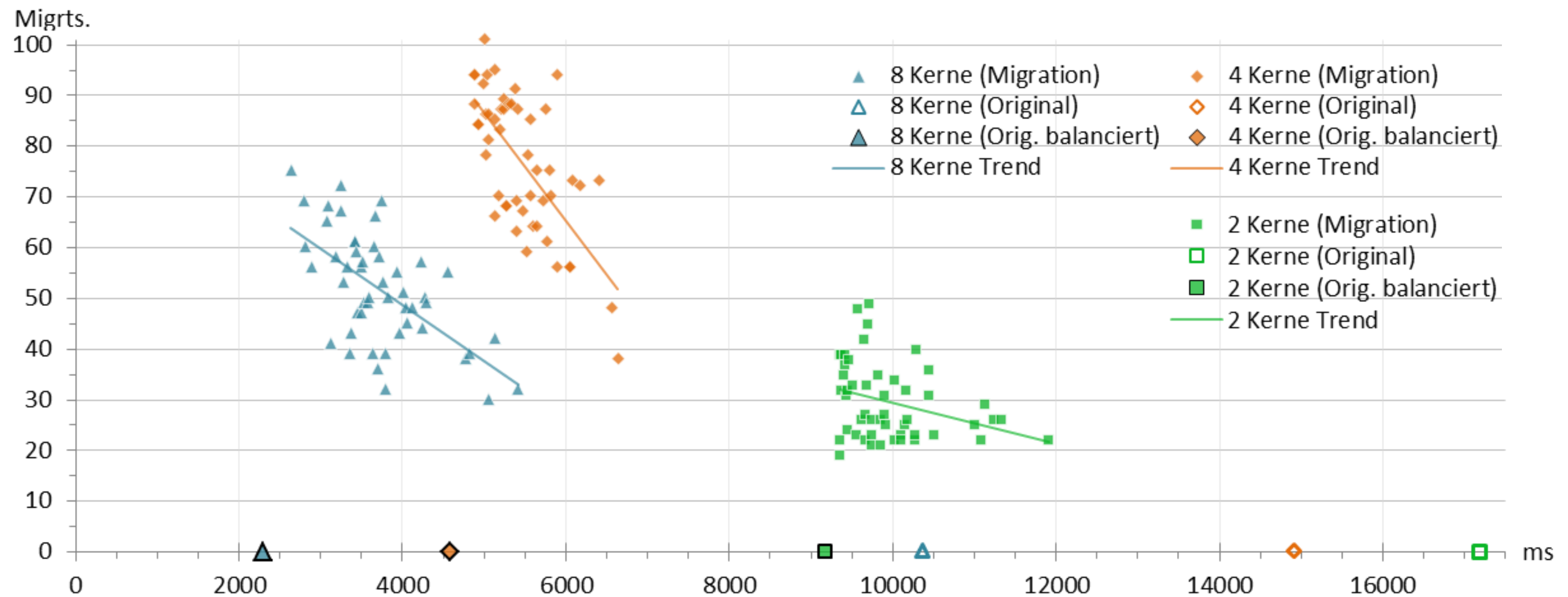


$$S_{p,\,mittel} = 2{,}73$$
$$S_{p,\,max} = 3{,}05$$
$$S_{p,\,theor} = 3{,}25$$

$$S_{p,\,mittel} = 1{,}725$$
$$S_{p,\,max} = 1{,}837$$
$$S_{p,\,theor} = 1{,}875$$

Günstigster Vergleich (unausgeglichenes System)

JGF-Crypt-Benchmark (16 Threads)



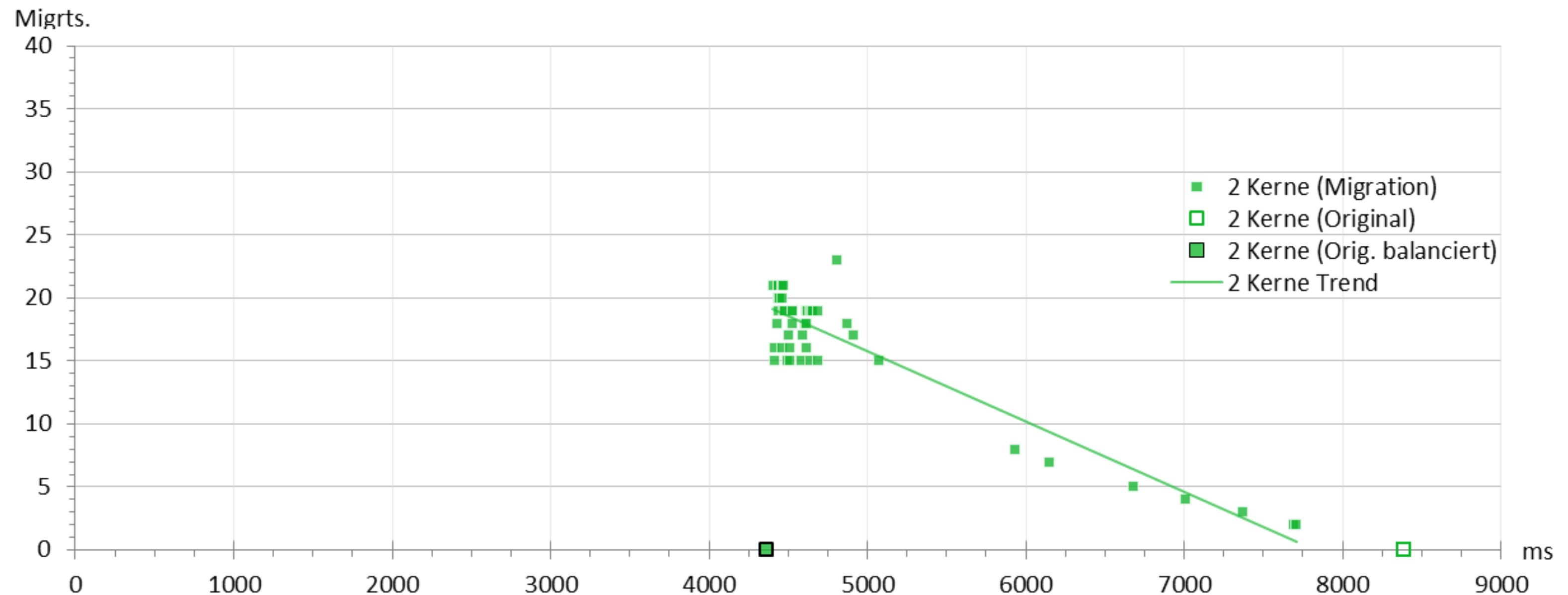
$$S_{p\,,\,mittel} = 2{,}78$$
$$S_{p,\,max} = 3{,}94$$
$$S_{p,\,theor} = 4{,}50$$

$$S_{p\,,\,mittel} = 2{,}73$$
$$S_{p,\,max} = 3{,}05$$
$$S_{p,\,theor} = 3{,}25$$

$$S_{p\,,\,mittel} = 1{,}725$$
$$S_{p,\,max} = 1{,}837$$
$$S_{p,\,theor} = 1{,}875$$

Günstigster Vergleich (unausgeglichenes System)

JGF-SparseMatmult-Benchmark (24 Threads)



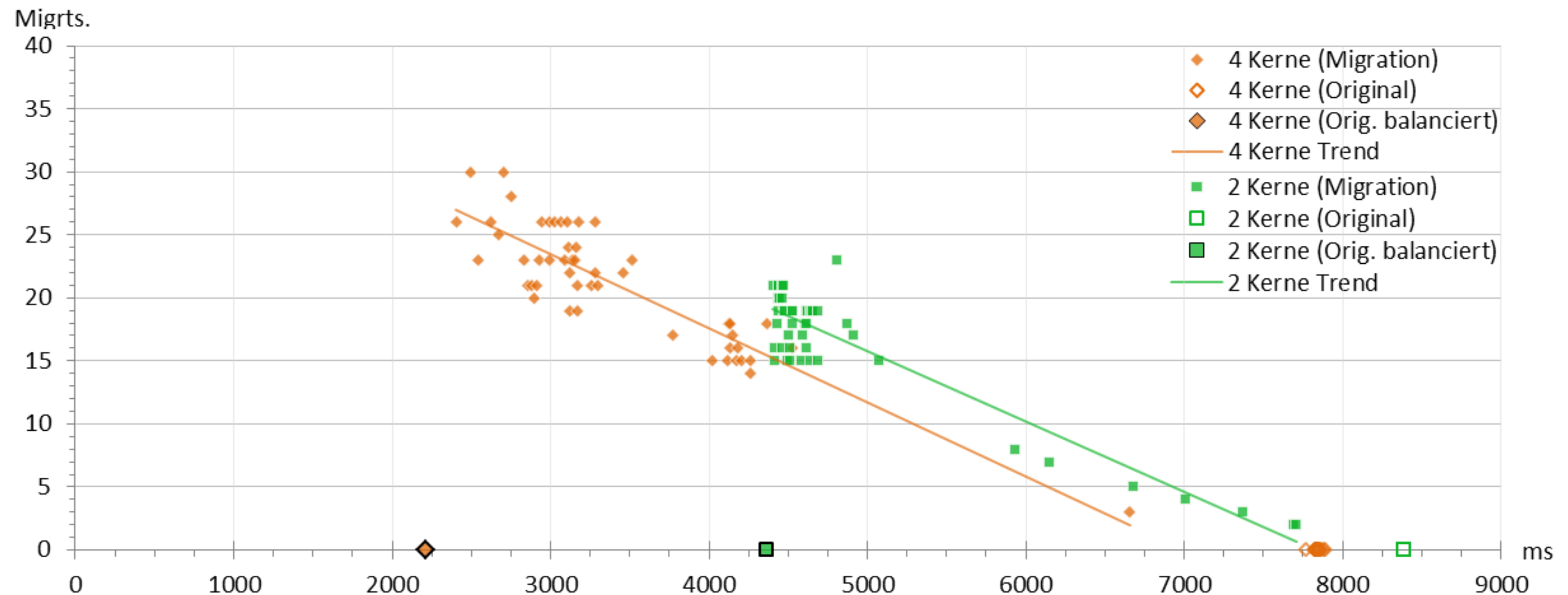
$S_{p\,,\,mittel} = 1{,}715$

$S_{p,\,max} = 1{,}904$

$S_{p,\,theor} = 1{,}917$

Günstigster Vergleich (unausgeglichenes System)

JGF-SparseMatmult-Benchmark (24 Threads)

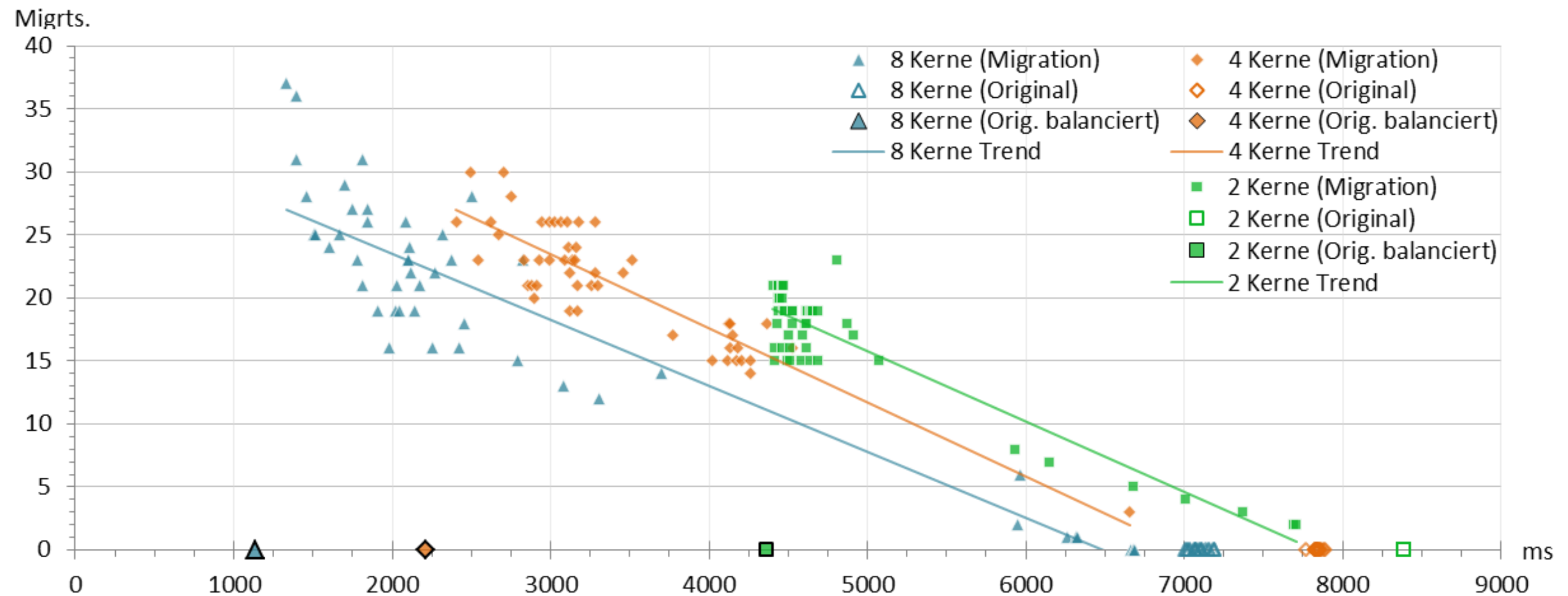


$$S_{p,\,mittel} = 2{,}30$$
$$S_{p,\,max} = 3{,}23$$
$$S_{p,\,theor} = 3{,}50$$

$$S_{p,\,mittel} = 1{,}715$$
$$S_{p,\,max} = 1{,}904$$
$$S_{p,\,theor} = 1{,}917$$

Günstigster Vergleich (unausgeglichenes System)

JGF-SparseMatmult-Benchmark (24 Threads)



$$S_{p,\,mittel} = 2{,}42$$
$$S_{p,\,max} = 5{,}26$$
$$S_{p,\,theor} = 5{,}67$$

$$S_{p,\,mittel} = 2{,}30$$
$$S_{p,\,max} = 3{,}23$$
$$S_{p,\,theor} = 3{,}50$$

$$S_{p,\,mittel} = 1{,}715$$
$$S_{p,\,max} = 1{,}904$$
$$S_{p,\,theor} = 1{,}917$$

4 Zusammenfassung

Entwurf

- Distributed Weighted Round-Robin als Strategie
- GC-Bus als Verbindungs-Hardware

4 Zusammenfassung

Entwurf

- Distributed Weighted Round-Robin als Strategie
- GC-Bus als Verbindungs-Hardware

Anpassungen

- Mikrocode erweitert
- 3 Locks vor Migration
- Zielthreadsuche und Nachbearbeitung überwiegend in Java
- Migrator als neues Hardware-Modul
- Erweiterung des GC-Bus
- Erweiterung anderer Module (vor allem Stack)

4 Zusammenfassung

Funktionalität

- Threadmigration funktioniert
- fehlendes Feature bei Leerlauf eines Kerns

Funktionalität

- Threadmigration funktioniert
- fehlendes Feature bei Leerlauf eines Kerns

Ergebnisse

- 16 % mehr LUTs
- mittlerer Speed-Up: 42 - 92 % des theor. Maximums
- beste Speed-Up: 87 - 99 % des theor. Maximums
- schlechtester Speed-Up: 96 % zum Original

4 Zusammenfassung

Funktionalität

- Threadmigration funktioniert
- fehlendes Feature bei Leerlauf eines Kerns

Ergebnisse

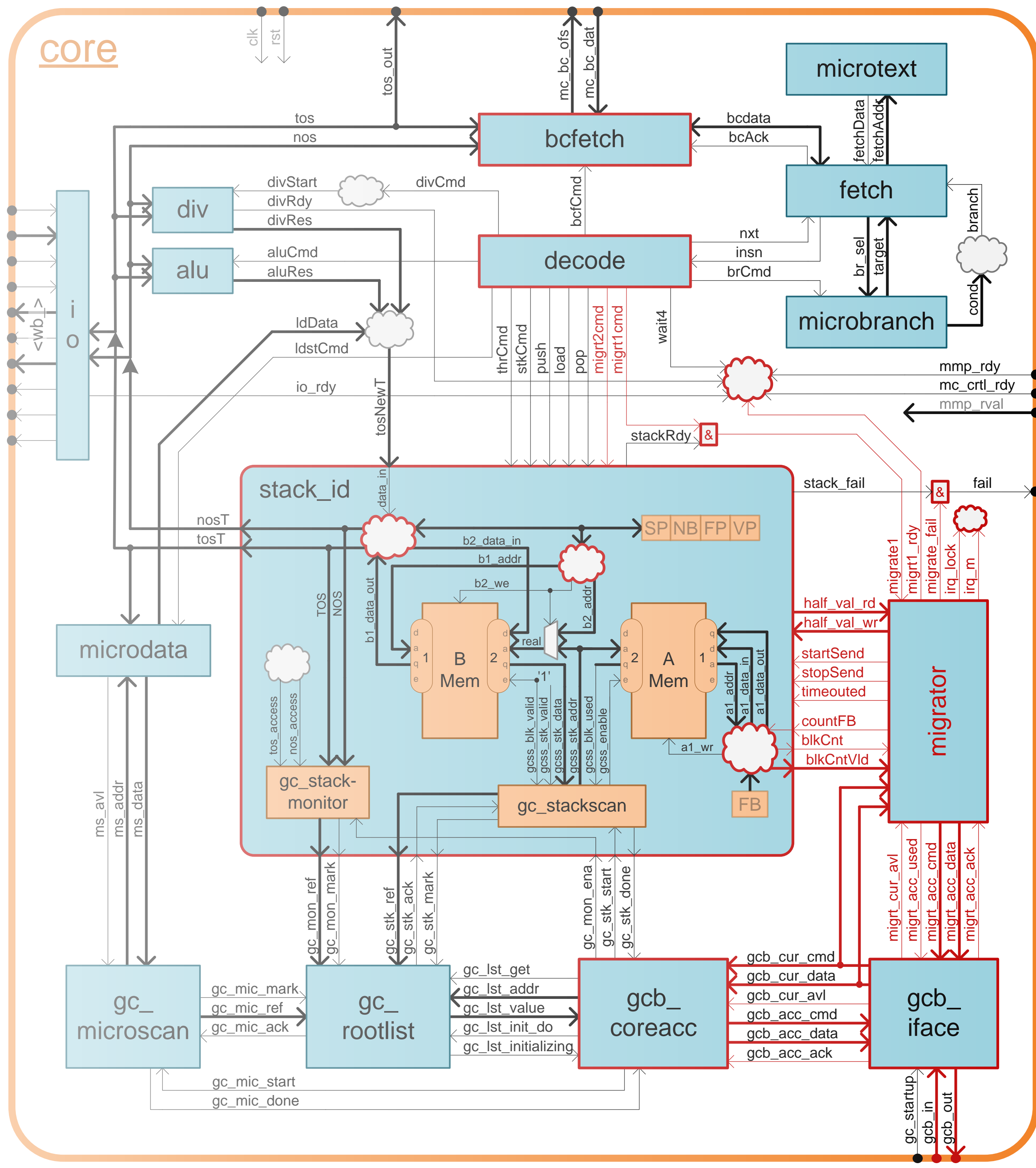
- 16 % mehr LUTs
- mittlerer Speed-Up: 42 - 92 % des theor. Maximums
- beste Speed-Up: 87 - 99 % des theor. Maximums
- schlechtester Speed-Up: 96 % zum Original

Aussichten

- leerlaufende Kerne nicht isolieren
- Vergrößerung des Mikrocode-Speichers
- Kompatibilität mit optimiertem ShapLinker

TECHNISCHE UNIVERSITÄT DRESDEN

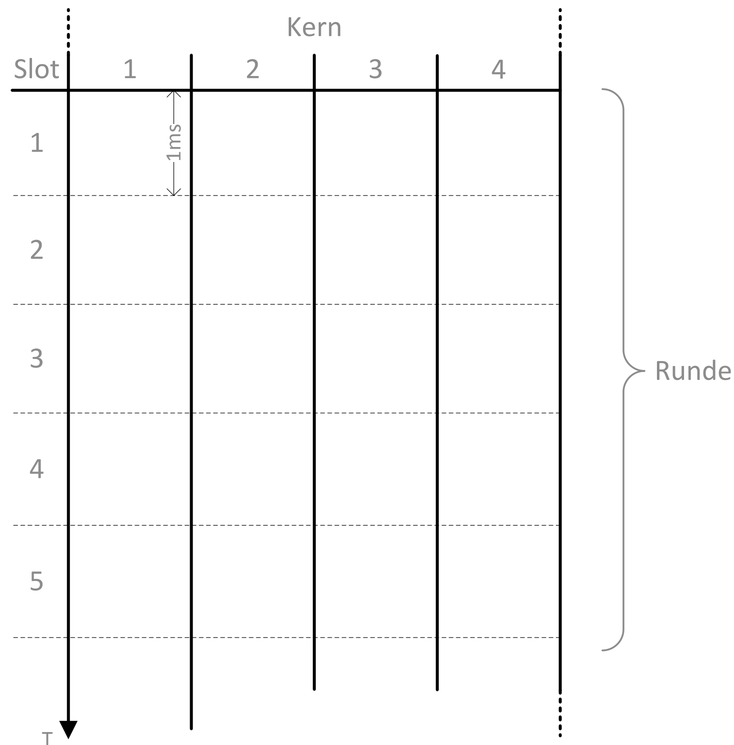
Vielen Dank!



Gang Scheduling (Inter-Thread-Kommunikation)

Programme (Threads):

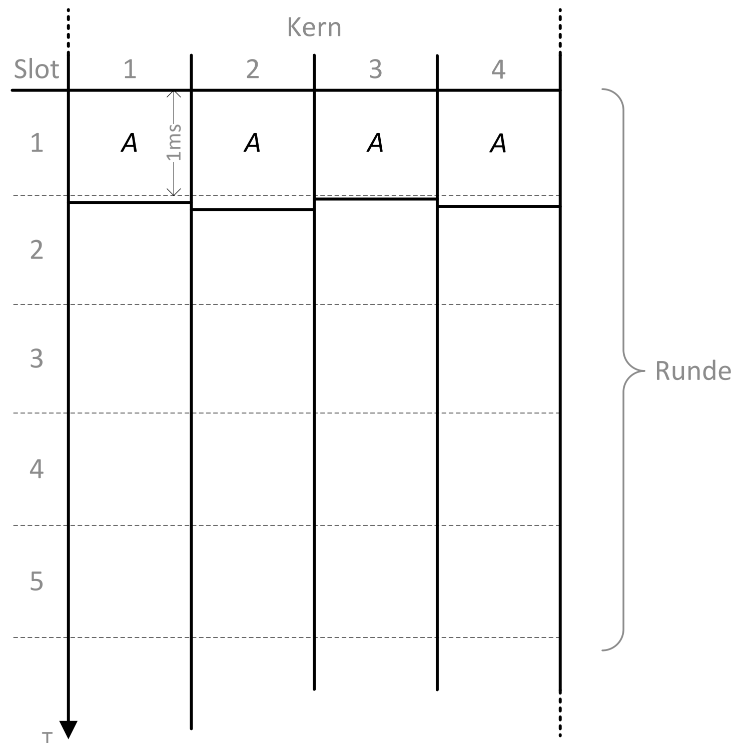
A(6), B(5), C(3), D(3), E(2), F(1)



Gang Scheduling (Inter-Thread-Kommunikation)

Programme (Threads):

A(6), B(5), C(3), D(3), E(2), F(1)

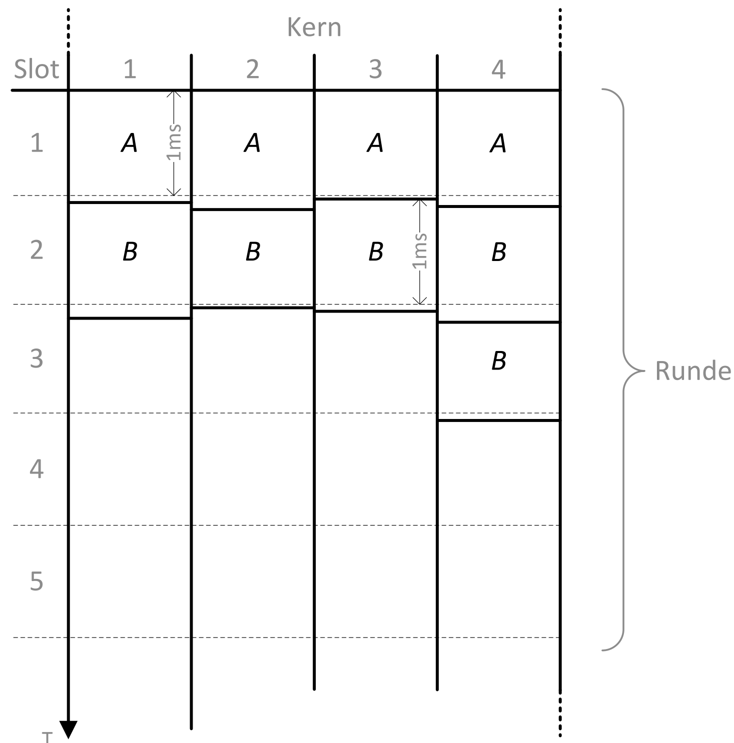


Runde

Gang Scheduling (Inter-Thread-Kommunikation)

Programme (Threads):

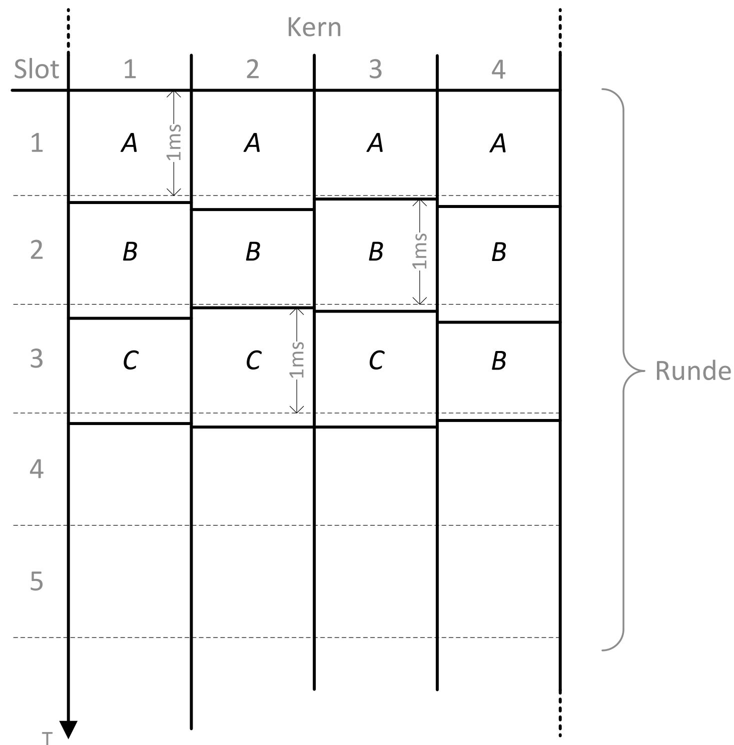
A(6), B(5), C(3), D(3), E(2), F(1)



Gang Scheduling (Inter-Thread-Kommunikation)

Programme (Threads):

A(6), B(5), C(3), D(3), E(2), F(1)

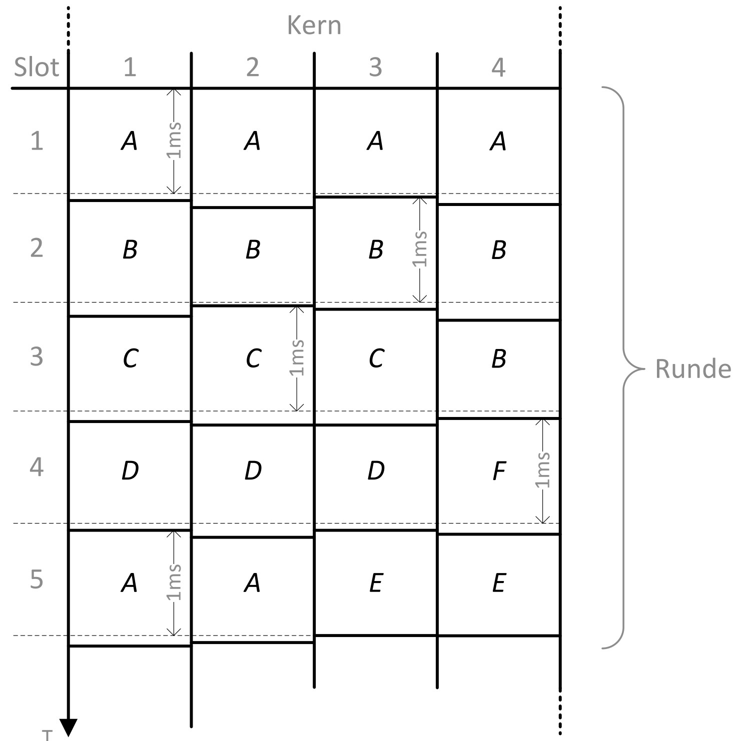


Runde

Gang Scheduling (Inter-Thread-Kommunikation)

Programme (Threads):

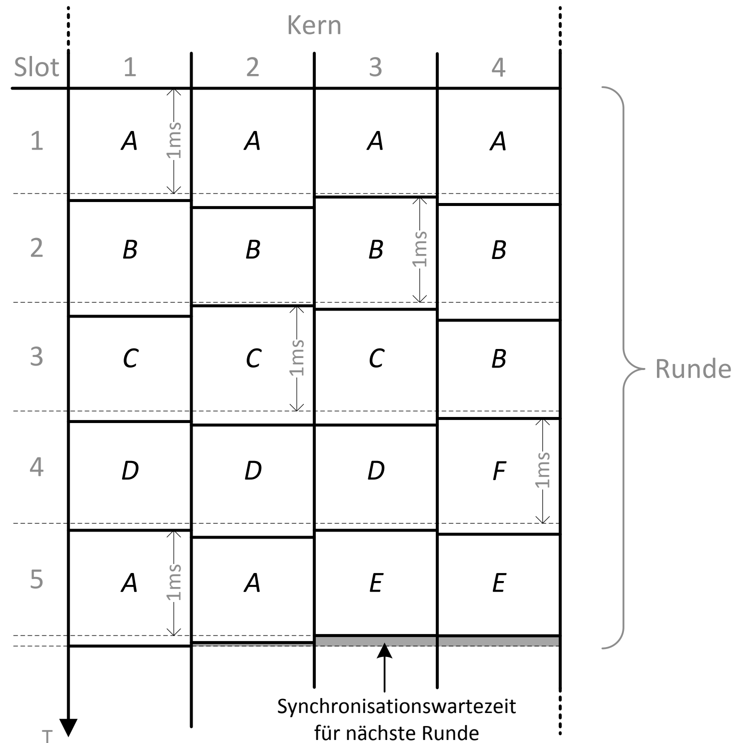
A(6), B(5), C(3), D(3), E(2), F(1)



Gang Scheduling (Inter-Thread-Kommunikation)

Programme (Threads):

A(6), B(5), C(3), D(3), E(2), F(1)

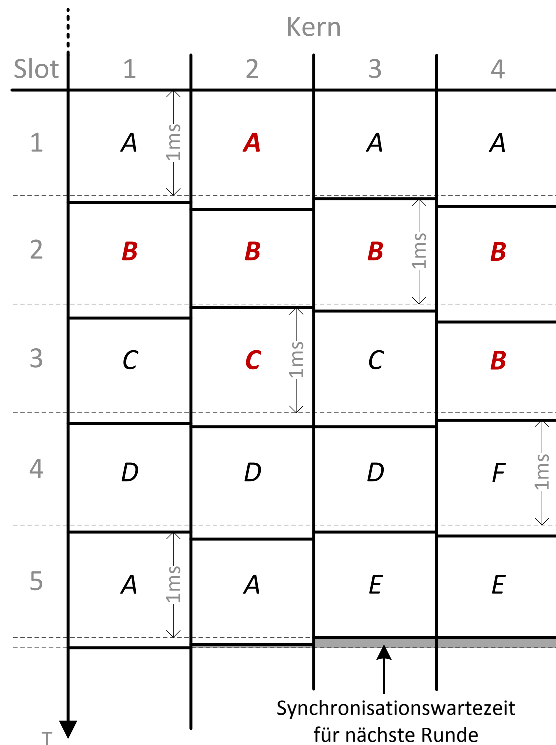


Gang Scheduling (Inter-Thread-Kommunikation)

Programme (Threads):

a) *A(6), B(5), C(3), D(3), E(2), F(1)*

b) *A**(5)**, **B(0)**, C**(2)**, D(3), E(2), F(1)*

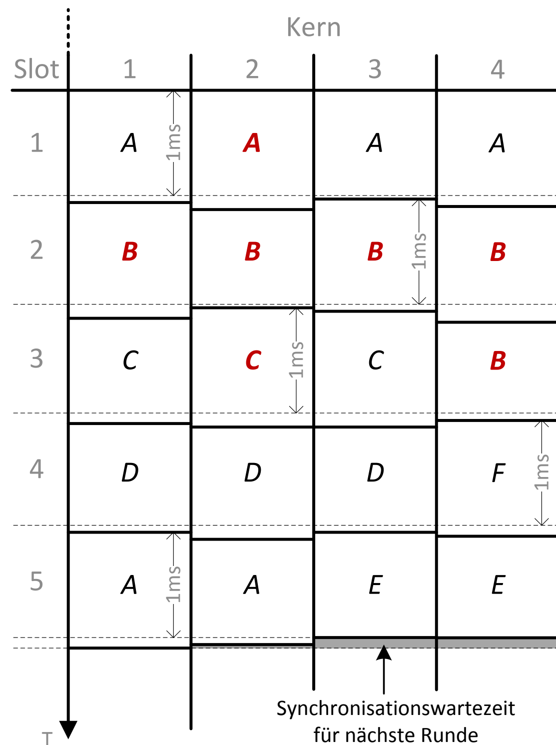


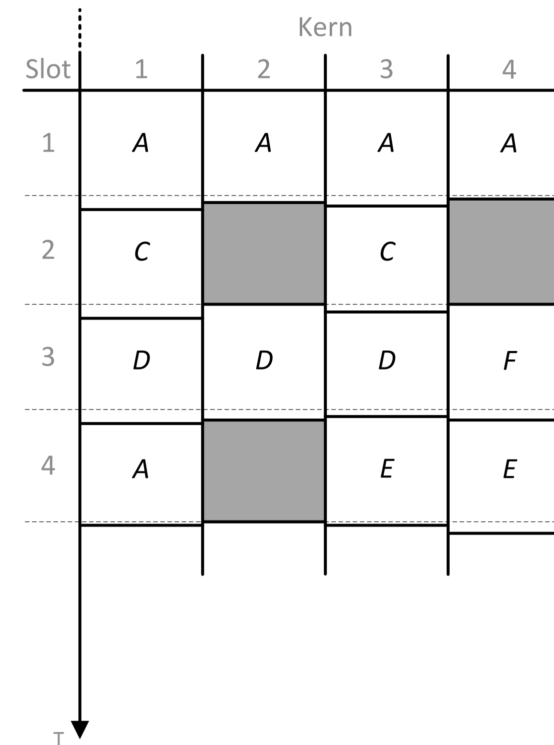
Gang Scheduling (Inter-Thread-Kommunikation)

Programme (Threads):

a) *A(6), B(5), C(3), D(3), E(2), F(1)*

b) *A**(5)**, **B(0)**, C**(2)**, D(3), E(2), F(1)*



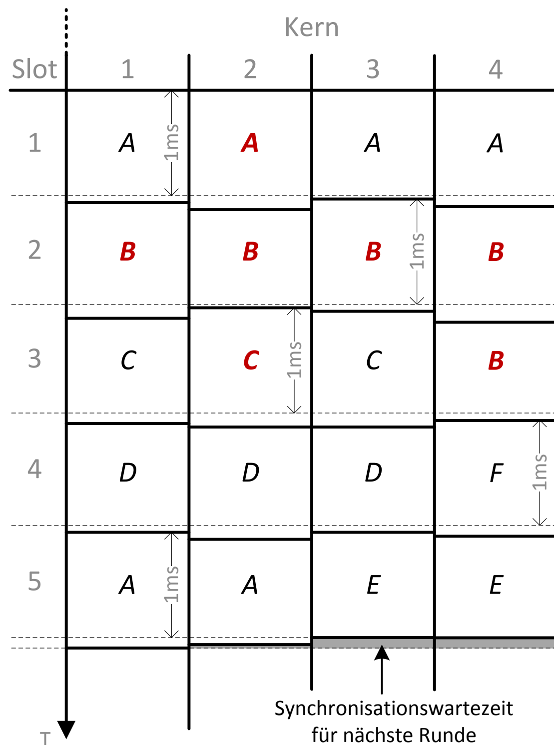


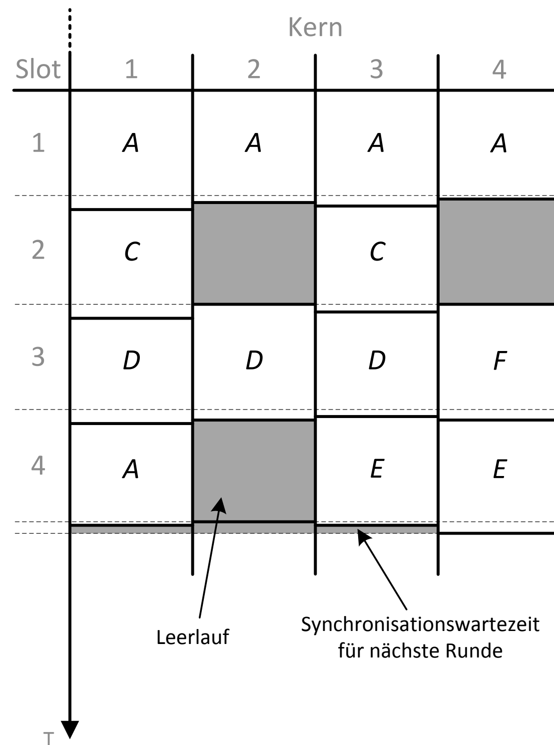
Gang Scheduling (Inter-Thread-Kommunikation)

Programme (Threads):

a) *A(6), B(5), C(3), D(3), E(2), F(1)*

b) *A**(5)**, **B(0)**, C**(2)**, D(3), E(2), F(1)*



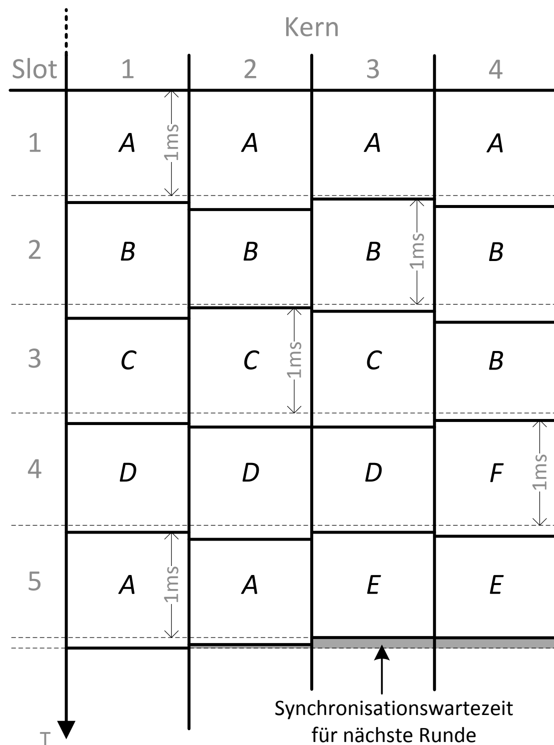


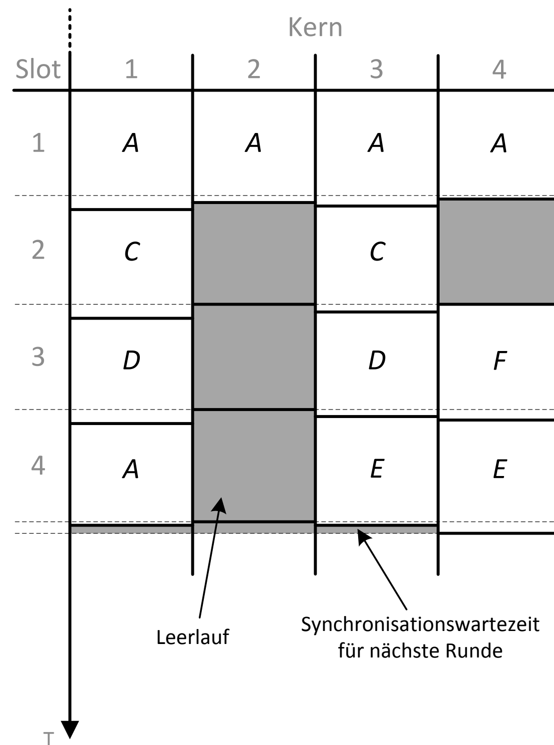
Gang Scheduling (Inter-Thread-Kommunikation)

Programme (Threads):

a) *A(6), B(5), C(3), D(3), E(2), F(1)*

b) *A(5), C(2), D(**2**), E(2), F(1)*

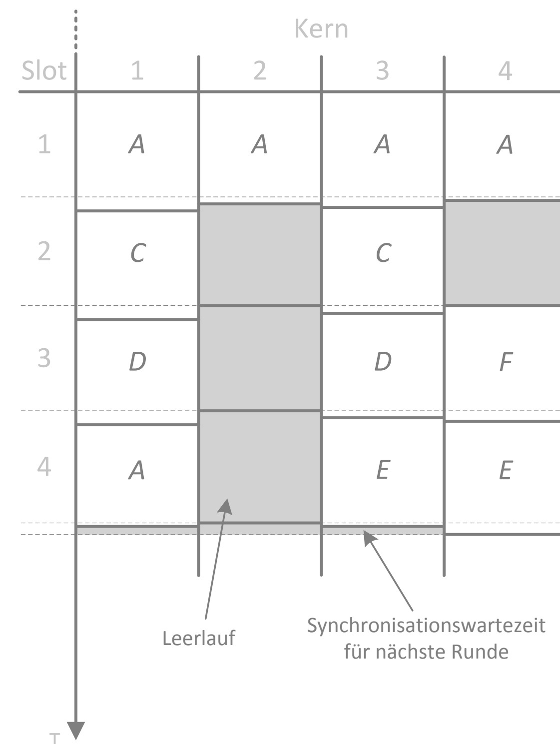




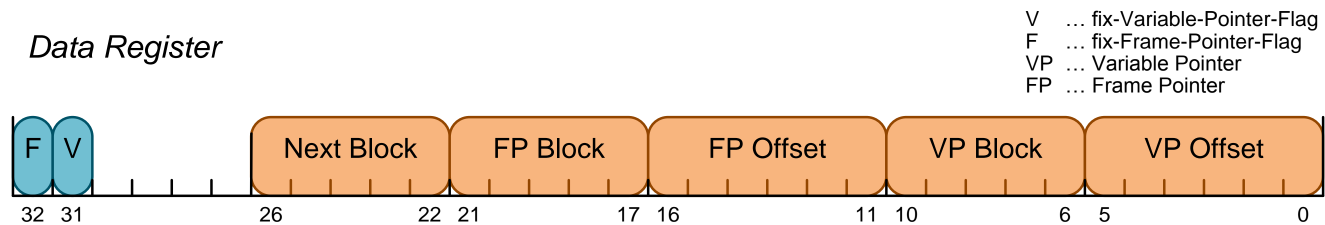
Gang Scheduling (Inter-Thread-Kommunikation)

- + Inter-Thread-Kommunikation
- + Fairness
- + Konzept der Threadgruppen passt zu SHAP
- + viele Pack-Algorithmen

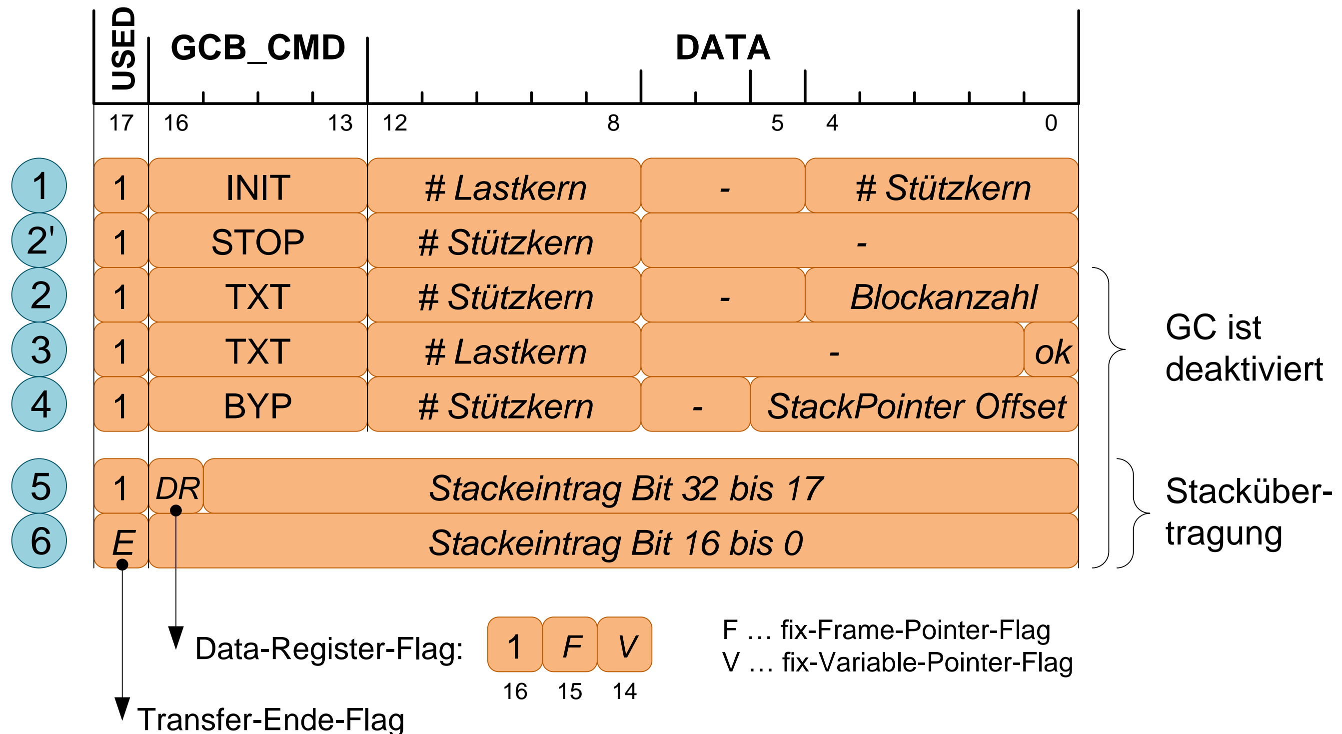
- – ineffizient ohne synchr. Kerne
- – Vorausplanung der Runde (Wartezeit)
- – Lücken im Plan
- – yield() wenig sinnvoll
- – keine Prioritäten
- – großer Migrations-Overhead (Wartezeit)



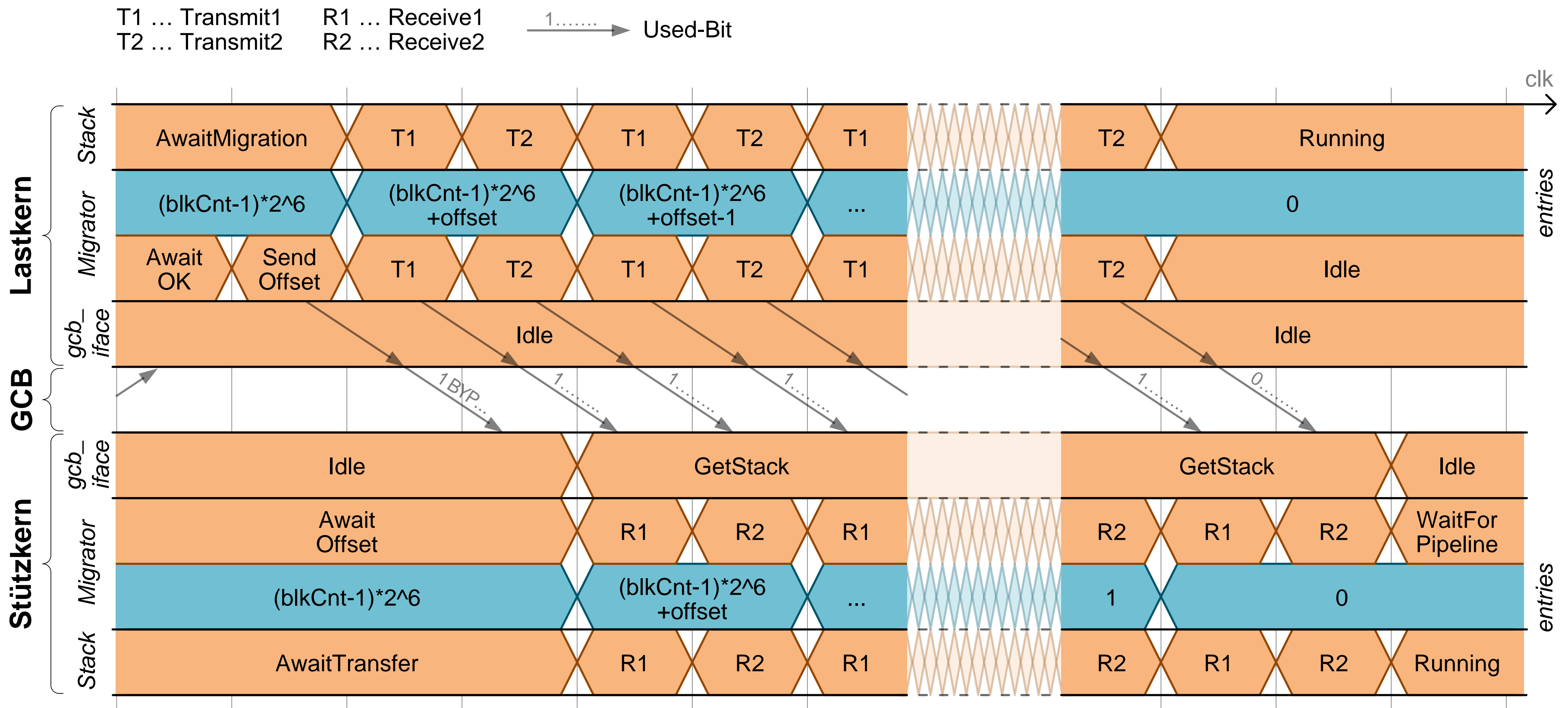
Datenregister Patching



GC-Bus Protokoll



Stackübertragung Zustände



Intuitiver Ansatz

a) *Informationsbasis: Threadanzahl*

	Kern **1**		**2**		**3**		**4**	
Thread	*µs*	%						
1	*1000*	25	*1000*	40	*1000*	49	*300*	75
2	*1000*	25	*900*	36	*1000*	49	*50*	12
3	*1000*	25	*400*	16	*50*	2	*50*	12
4	*1000*	25	*200*	8				
Σ	*4000*		*2500*		*2050*		*400*	

b) *Informationsbasis: Rundenlänge*

	Kern **1**		**2**		**3**		**4**	
Thread	*µs*	%						
1	*1000*	43	*1000*	45	*1000*	46	*1000*	43
2	*1000*	43	*1000*	45	*1000*	46	*900*	39
3	*300*	13	*200*	9	*50*	2	*400*	17
4					*50*	2		
5					*50*	2		
Σ	*2300*		*2200*		*2150*		*2300*	

Cassavant & Kuhl