Betriebssystemtechnik

Adressräume: Trennung, Zugriff, Schutz

XI. Spezialfälle: Virtuell gemeinsamer Speicher

Wolfgang Schröder-Preikschat / Volkmar Sieh

SS 2024

Gliederung

VSM

- eine **verteilte Speicherarchitektur**, die für die Anwendungssoftware so aussieht, als wäre sie ein *shared-memory*-System

VSM

- eine **verteilte Speicherarchitektur**, die für die Anwendungssoftware so aussieht, als wäre sie ein *shared-memory*-System
  - **physisch verteilt**, die Gesamtheit an Hauptspeicher bildet sich aus den lokalen Speichern der durch ein Netzwerk verbundenen Prozessoren
    - *distributed shared memory* (DSM) $\rightsquigarrow$ konventionelles verteiltes System

VSM

- eine **verteilte Speicherarchitektur**, die für die Anwendungssoftware so aussieht, als wäre sie ein *shared-memory*-System

  - **logisch verteilt**, bei Cacheinkohärenz zwischen den im global gemeinsam genutzten Adressraum liegenden Hauptspeicherpartitionen
    - *partitioned global address space* (PGAS)

VSM

- eine **verteilte Speicherarchitektur**, die für die Anwendungssoftware so aussieht, als wäre sie ein *shared-memory*-System

- **seitenbasierter virtueller Speicher** im global gemeinsam genutzten Adressraum gleichzeitiger Prozesse mehrerer Rechenkerne

VSM

- eine **verteilte Speicherarchitektur**, die für die Anwendungssoftware so aussieht, als wäre sie ein *shared-memory*-System

- **seitenbasierter virtueller Speicher** im global gemeinsam genutzten Adressraum gleichzeitiger Prozesse mehrerer Rechenkerne
  - auf Nachfrage (*on-demand*) wird eine Seite dem Rechenkern des jeweils zugreifenden Prozessfadens lokal verfügbar gemacht
    - aus einer tieferen Ebene der Speicherhierarchie in den Hauptspeicher **kopiert**
    - innerhalb der Hauptspeicherebene zwischen benachbarten Orten **repliziert**

VSM

- eine **verteilte Speicherarchitektur**, die für die Anwendungssoftware so aussieht, als wäre sie ein *shared-memory*-System

- **seitenbasierter virtueller Speicher** im global gemeinsam genutzten Adressraum gleichzeitiger Prozesse mehrerer Rechenkerne
  - auf Nachfrage (*on-demand*) wird eine Seite dem Rechenkern des jeweils zugreifenden Prozessfadens lokal verfügbar gemacht
    - aus einer tieferen Ebene der Speicherhierarchie in den Hauptspeicher **kopiert**
    - innerhalb der Hauptspeicherebene zwischen benachbarten Orten **repliziert**
  - ersteres ist ein Konzept, um Mehrprogrammbetrieb [12] effizient gestalten zu können, letzteres unterstützt **parallele Programmierung** [15]

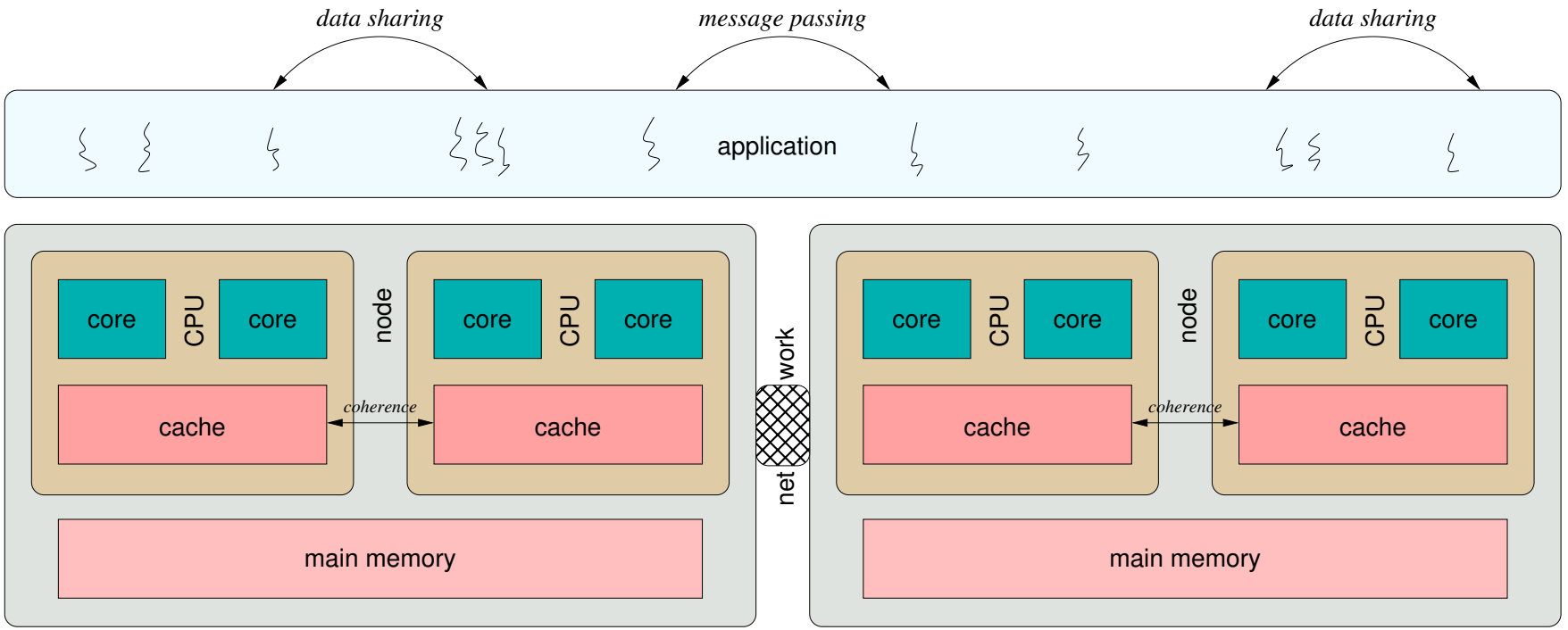
VSM

- eine **verteilte Speicherarchitektur**, die für die Anwendungssoftware so aussieht, als wäre sie ein *shared-memory*-System
- **seitenbasierter virtueller Speicher** im global gemeinsam genutzten Adressraum gleichzeitiger Prozesse mehrerer Rechenkerne
- logisch/physisch verteilt organisierter gemeinsamer Speicher, der mit Seitenadressierung **systemweit unverteilt erscheint** [20]

Gliederung

Physisch verteilter Speicher

distributed memory

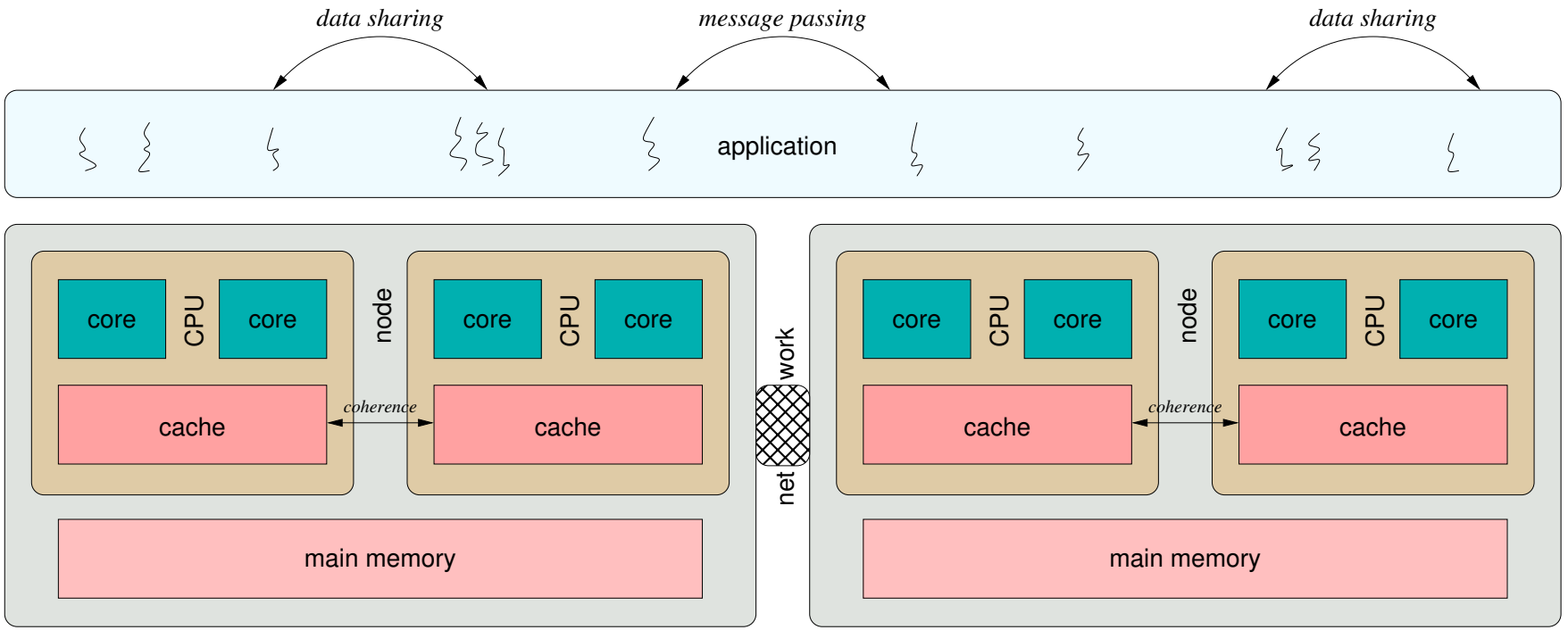


- klassisches **verteiltes System** ≡ nichtsequentielles System [23]

Prozesse bzw. Prozessoren haben keinen gemeinsamen Speicher und müssen daher über Nachrichten miteinander kommunizieren.

Physisch verteilter Speicher

distributed memory

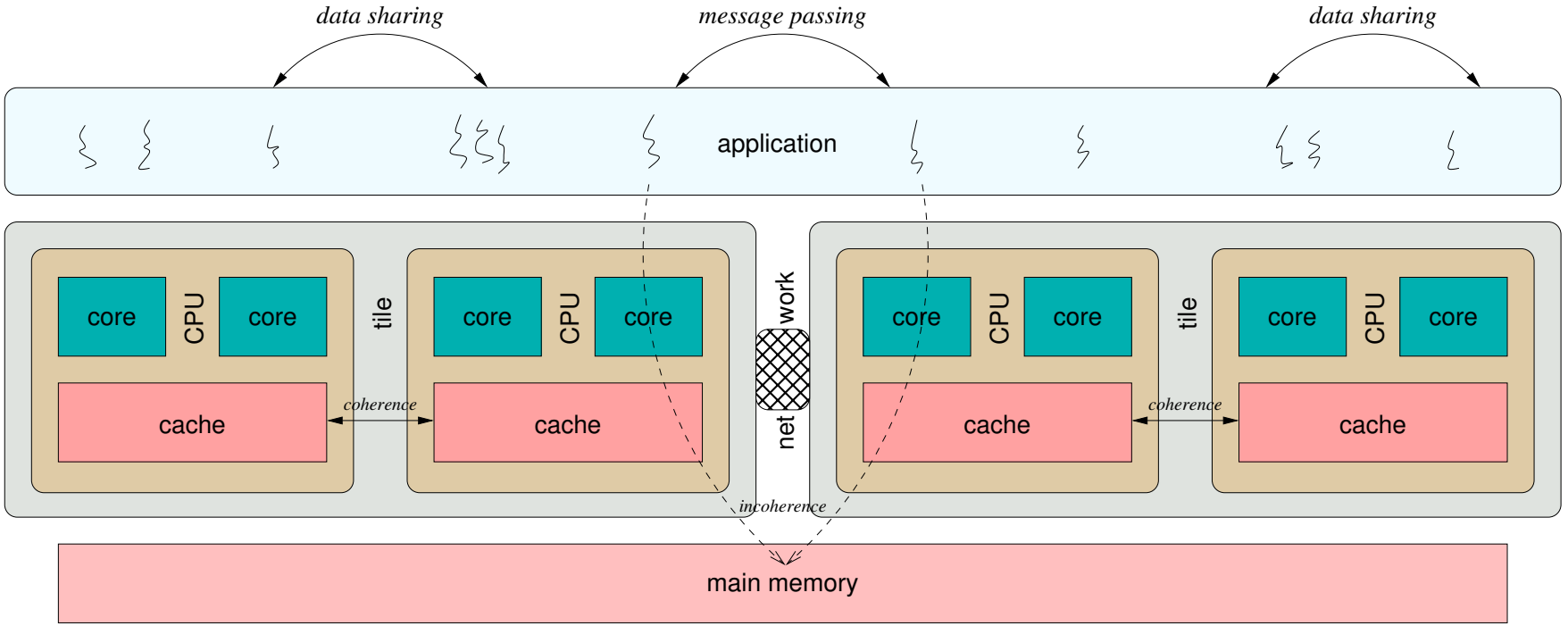


- klassisches **verteiltes System** ≡ nichtsequentielles System [23]

  Prozesse bzw. Prozessoren haben keinen gemeinsamen Speicher und müssen daher über Nachrichten miteinander kommunizieren.

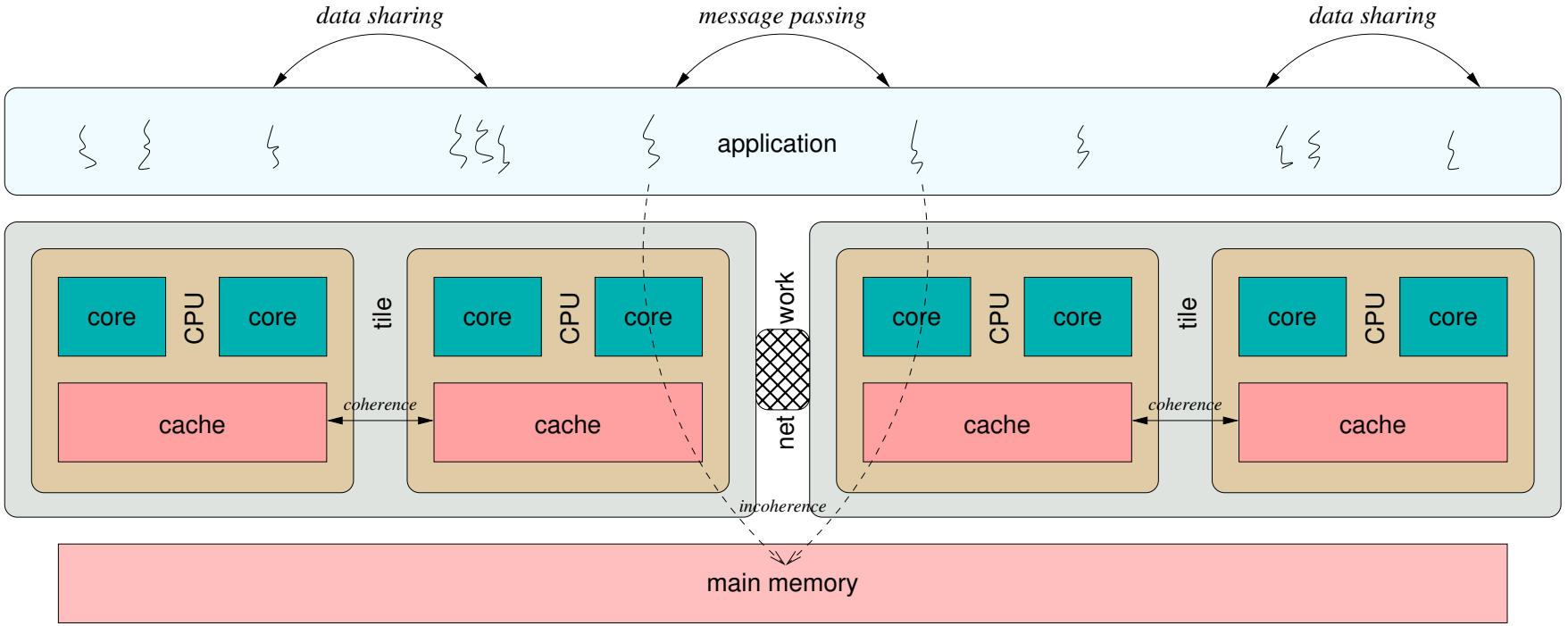
- gemeinsamer Speicher existiert nur innerhalb der Knoten/Prozessoren
  - Datenaustausch (*data sharing*) in dieser Domäne ist *cache-kohärent*

Logisch verteilter Speicher



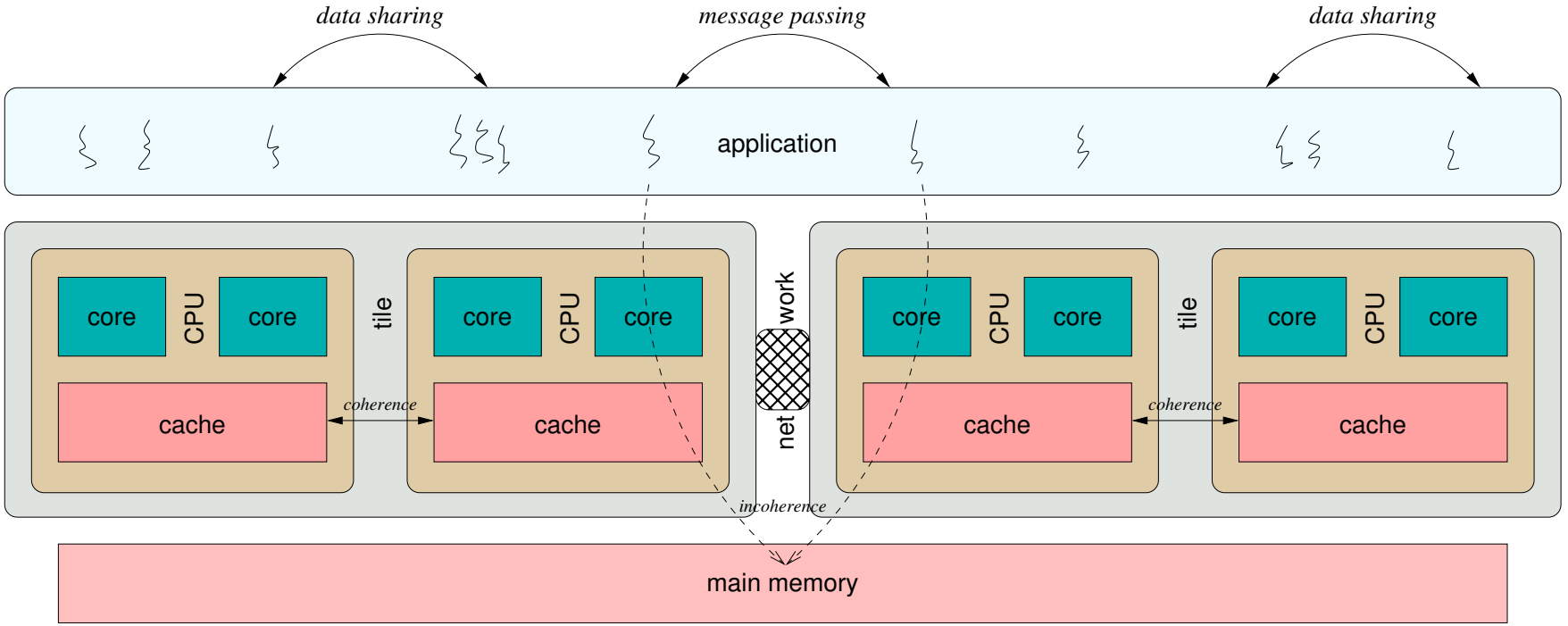
- ein **gekachelter Mehrkernprozessor** (*tiled multicore processor*, [24])

Logisch verteilter Speicher



- ein **gekachelter Mehrkernprozessor** (*tiled multicore processor*, [24])
  - Skalierungseinheit ist die Kachel (*tile*), gleichsam eine „Kohärenzinsel“
  - Datenaustausch (*data sharing*) in dieser Domäne ist *cache-kohärent*

Logisch verteilter Speicher

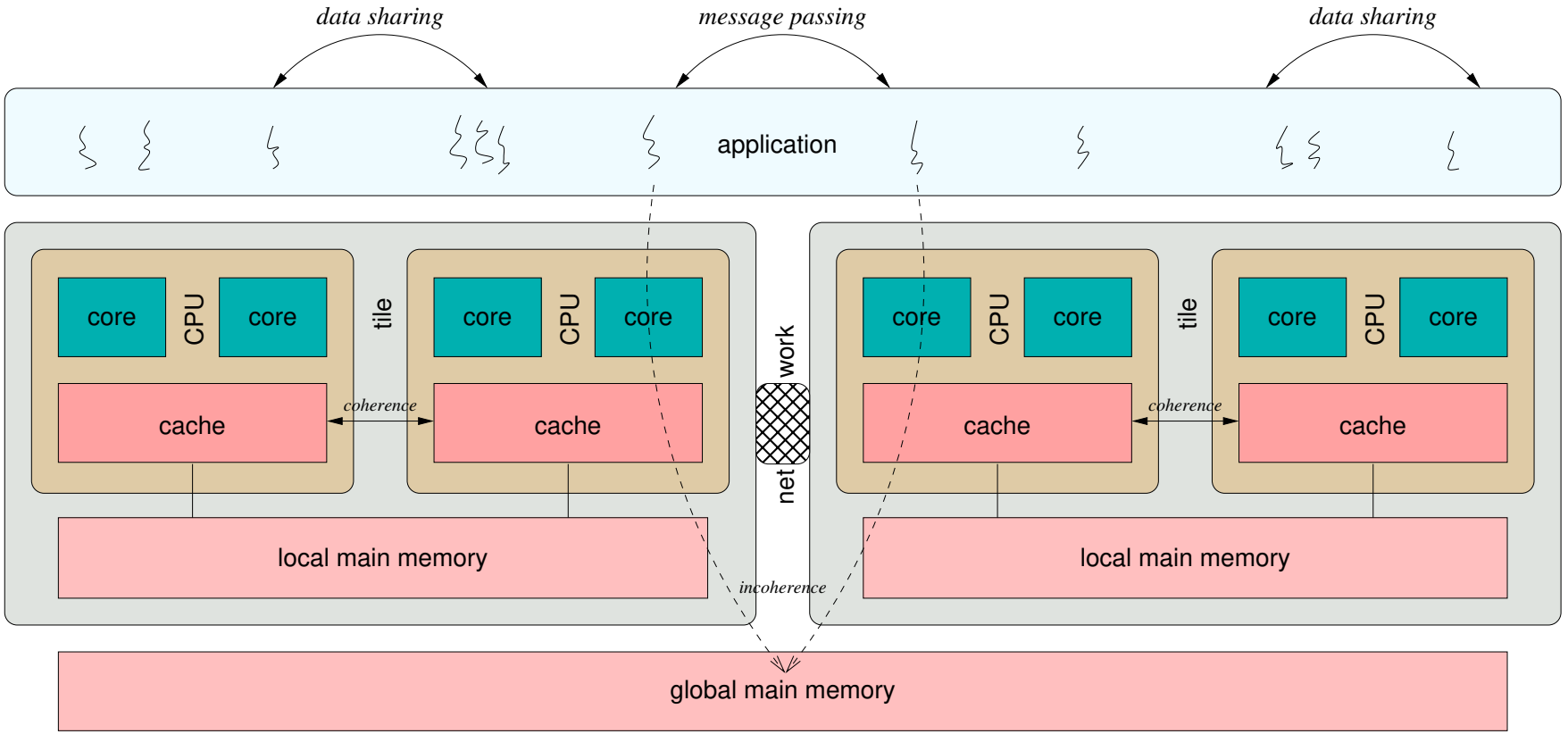


- ein **gekachelter Mehrkernprozessor** (*tiled multicore processor*, [24])

- **kachelglobaler gemeinsamer Hauptspeicher**, aber mit **Bedacht**:
  - systemweiter (kachelübergreifender) Datenaustausch ist *cache-inkohärent*

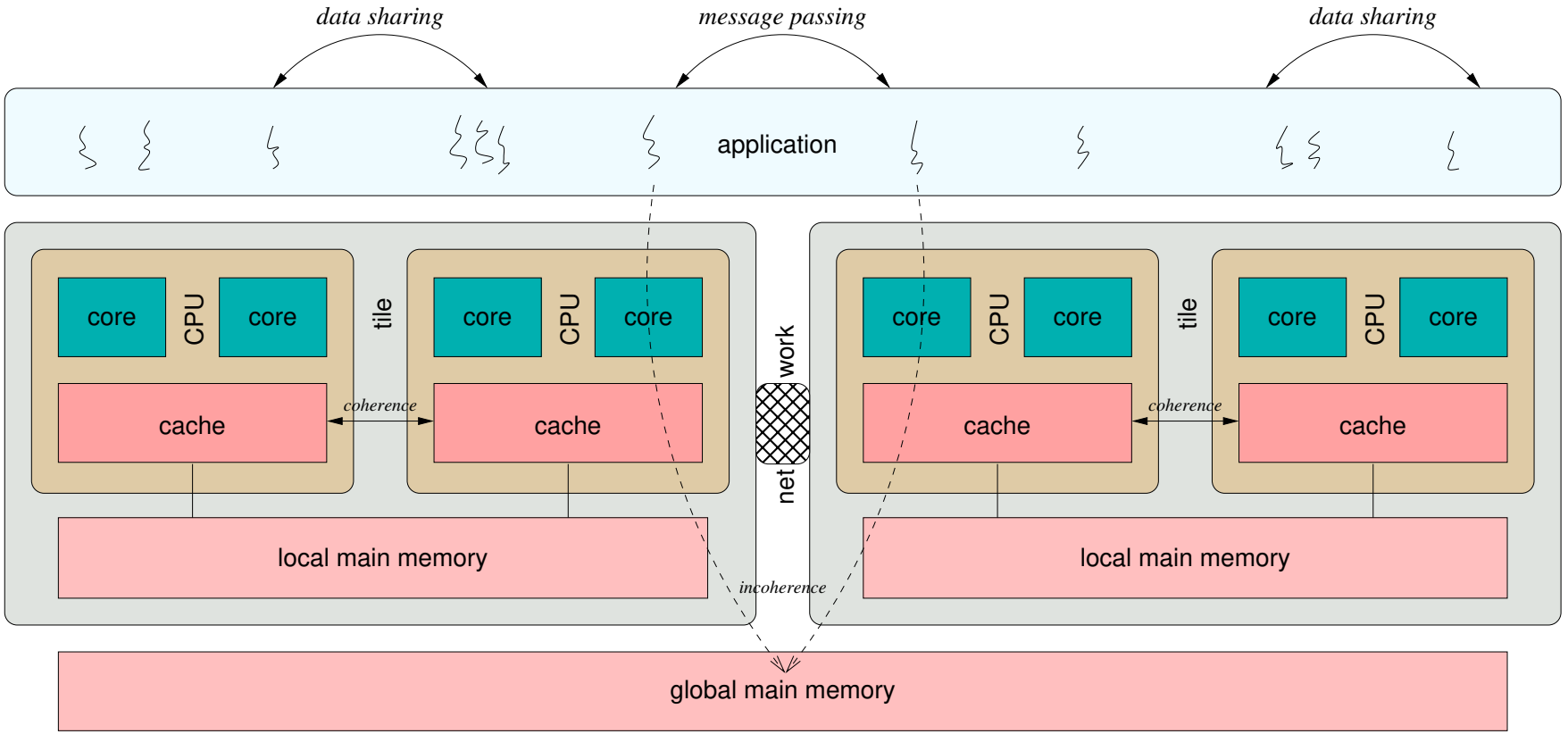
    ↪ gleichzeitige Speicherzugriffe verschiedener Kerne verschiedener Kacheln

Physisch und logisch verteilter Speicher



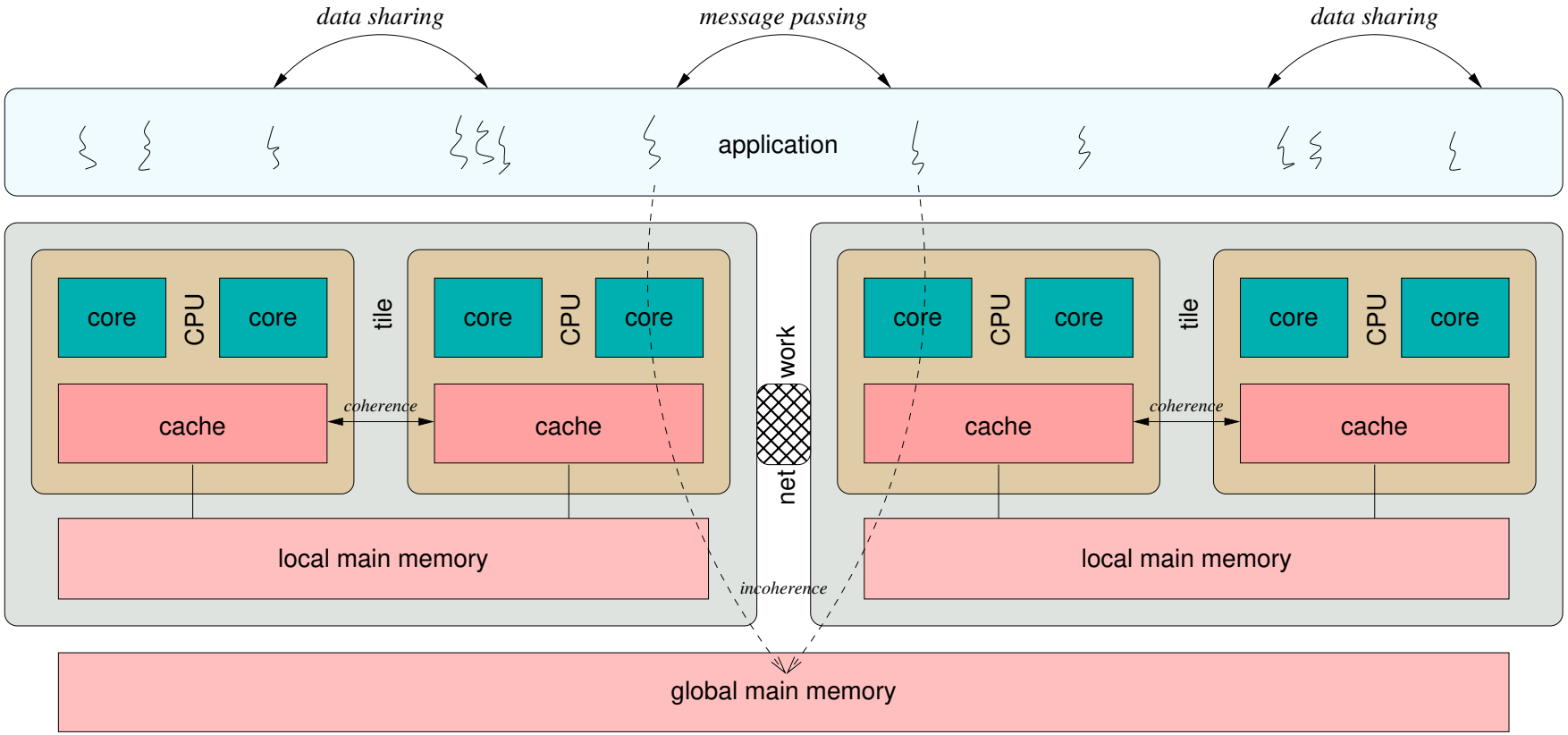
- charakteristisch für ein **MPSoC** (*multiprocessor system on a chip*)

Physisch und logisch verteilter Speicher



- charakteristisch für ein **MPSoC** (*multiprocessor system on a chip*)
  - typische Merkmale eines gekachelten Vielkernprozessors (vgl. S. 6)
  - zusätzlich **kachellokaler Hauptspeicher** (*tile-local memory*, TLM)

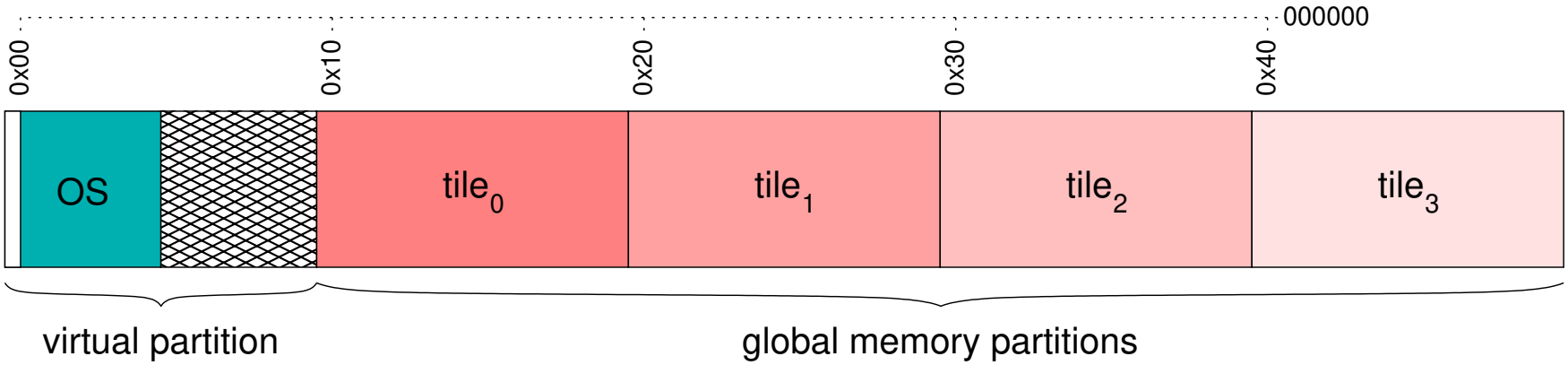
Physisch und logisch verteilter Speicher



- charakteristisch für ein **MPSoC** (*multiprocessor system on a chip*)
  - zusätzlich **kachellokaler Hauptspeicher** (*tile-local memory*, TLM)
- der TLM ist für gewöhnlich nicht direkt „von außen“ zugänglich
  - könnte aber über virtuelle Adressen global verfügbar gemacht werden

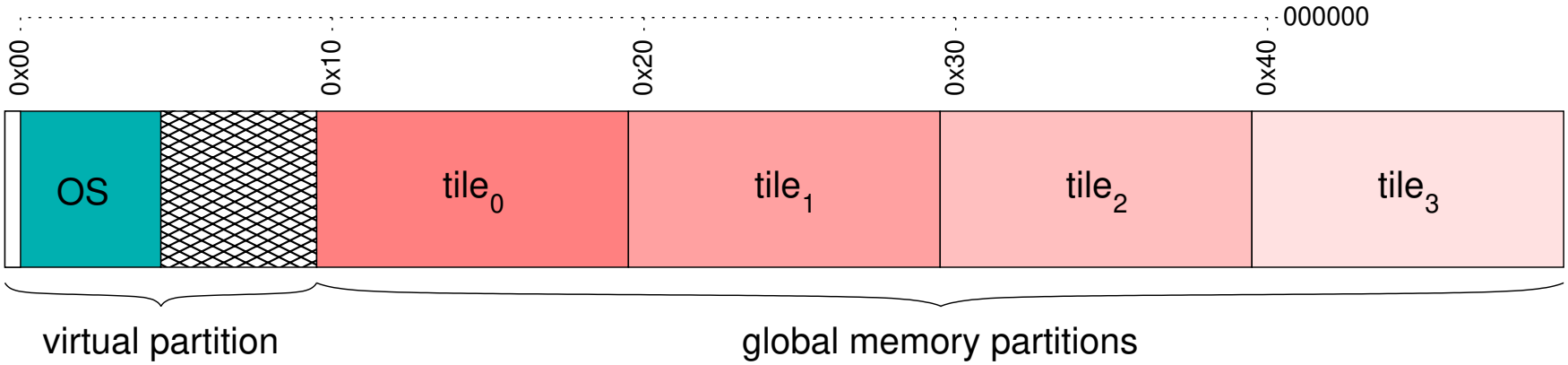
PGAS

partitioned global address space



PGAS

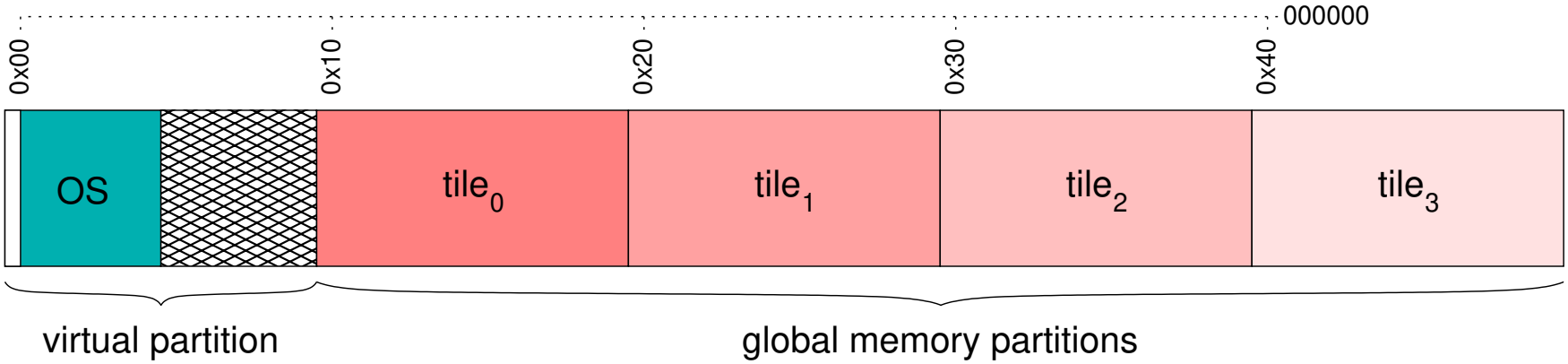
partitioned global address space



- das Betriebssystem (OS) sei *multikernel*-basiert [2]
  - das heißt, in jedem TLM liegt jeweils eine Kopie des Betriebssystems

PGAS

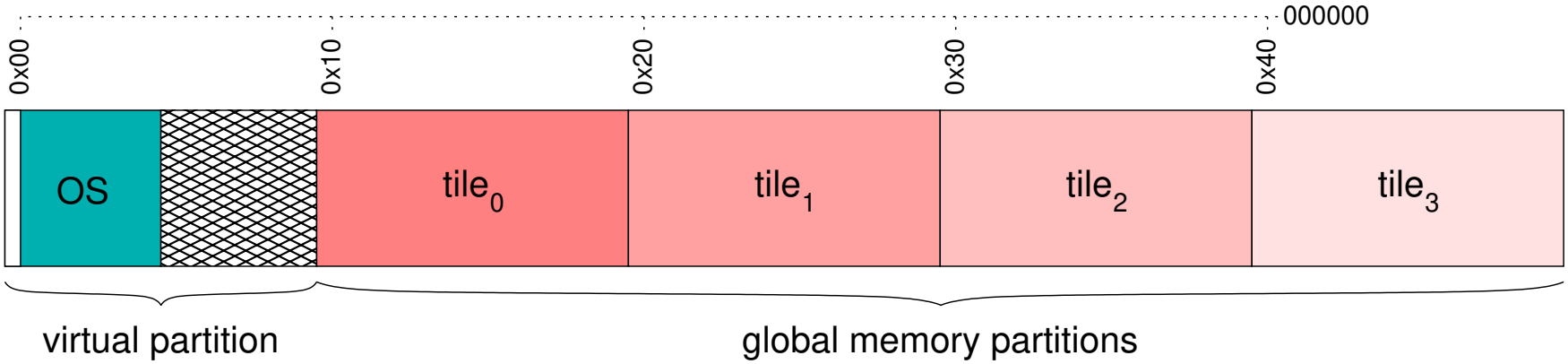
partitioned global address space



- das Betriebssystem (OS) sei *multikernel*-basiert [2]
  - das heißt, in jedem TLM liegt jeweils eine Kopie des Betriebssystems
  - jede dieser Kopien bildet ab auf dieselbe (virtuelle) Adressraumpartition
    - die Betriebssystemexemplare sehen sich immer im selben Adressraumbereich
    - erleichtert Interaktionen der Exemplare

PGAS

partitioned global address space

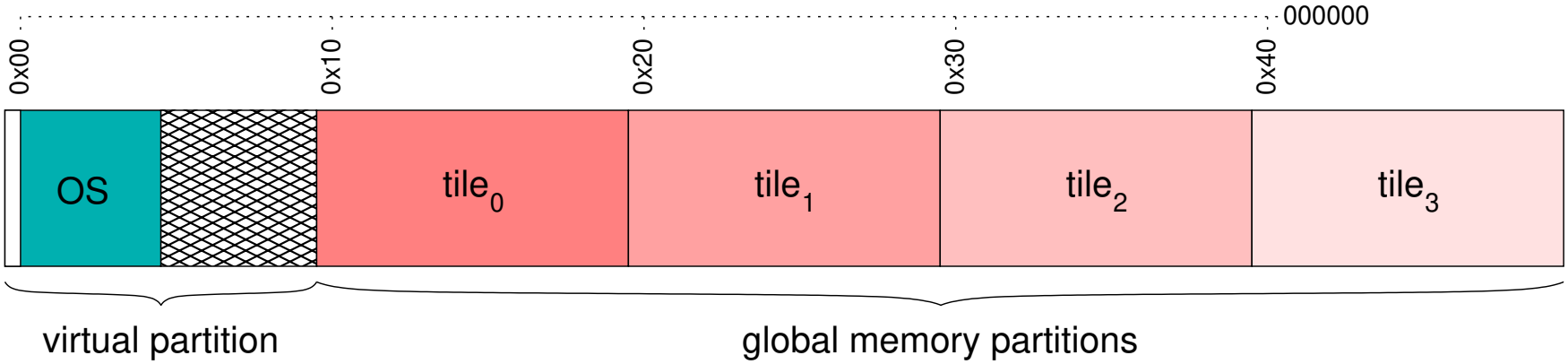


- das Betriebssystem (OS) sei *multikernel*-basiert [2]
  - das heißt, in jedem TLM liegt jeweils eine Kopie des Betriebssystems

- das Anwendungsprogramm der Anwendung sei partitioniert
  - das heißt, Text und Daten sind (virtuell) über die Kacheln verteilt
  - wobei dafür TLM wie auch der globale Hauptspeicher genutzt wird
    - beispielsweise lokale und „heiße“ Text-/Datenbestände in den TLM
    - gemeinsame Daten in den kachelglobalen Hauptspeicher

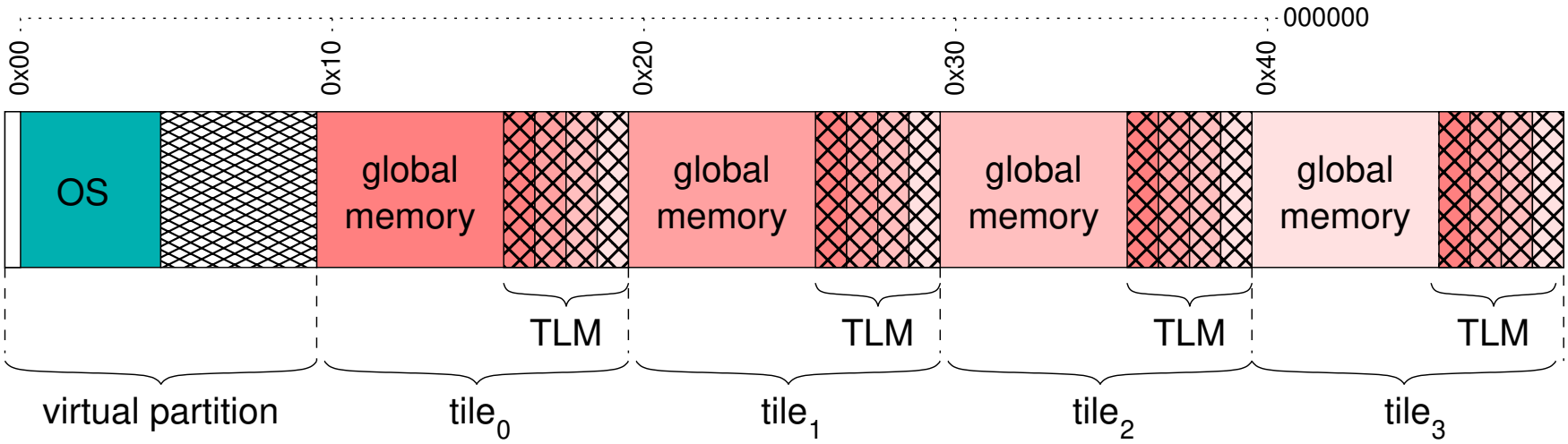
PGAS

partitioned global address space

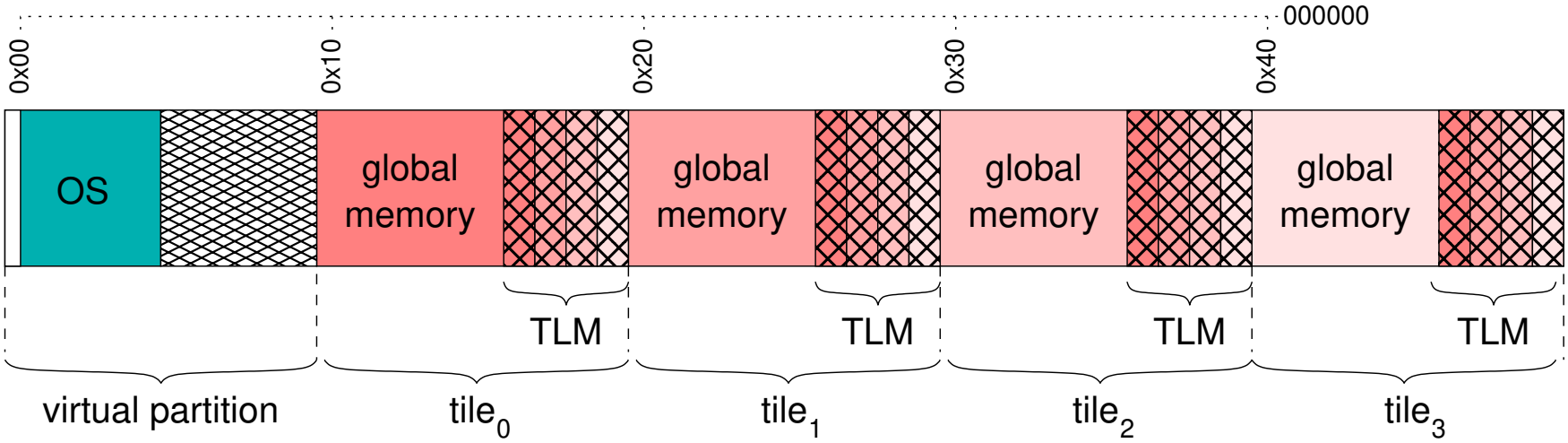


- das Betriebssystem (OS) sei *multikernel*-basiert [2]
  - das heißt, in jedem TLM liegt jeweils eine Kopie des Betriebssystems
- das Anwendungsprogramm der Anwendung sei partitioniert
  - das heißt, Text und Daten sind (virtuell) über die Kacheln verteilt
- auch kann der TLM generell als Seitenpuffer (*page cache*) dienen
  - um Seiten des PGAS auf Nachfrage (*on-demand*) lokal zu speichern

PGAS mit TLM

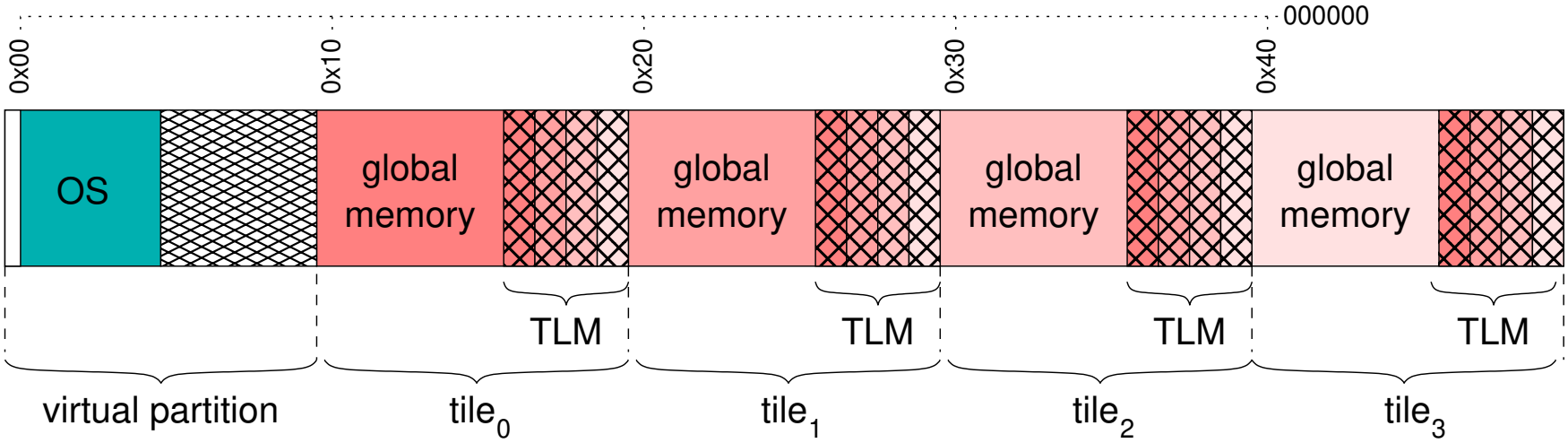

000000
0x00
0x10
0x20
0x30
0x40
OS
global memory
global memory
global memory
global memory
TLM
TLM
TLM
TLM
virtual partition
tile0
tile1
tile2
tile3

PGAS mit TLM



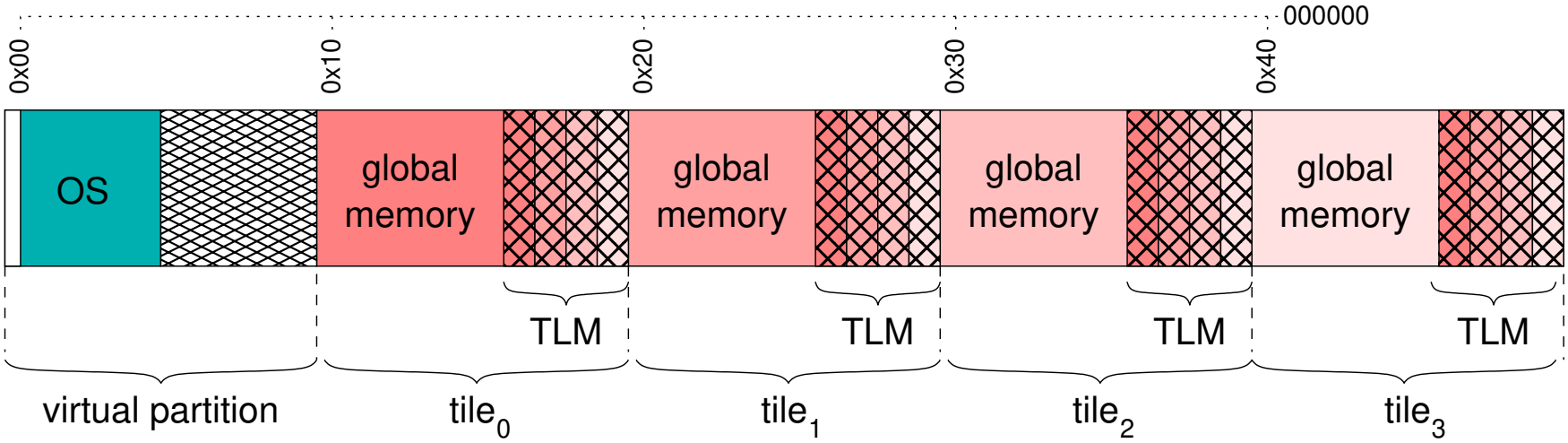
- eine Kachelpartition $tile_i$ definiert sich durch zwei Sektionen:
  - das Abbild der korrespondierenden globalen Hauptspeicherpartition und
  - die TLM-Abbilder aller Kacheln

PGAS mit TLM



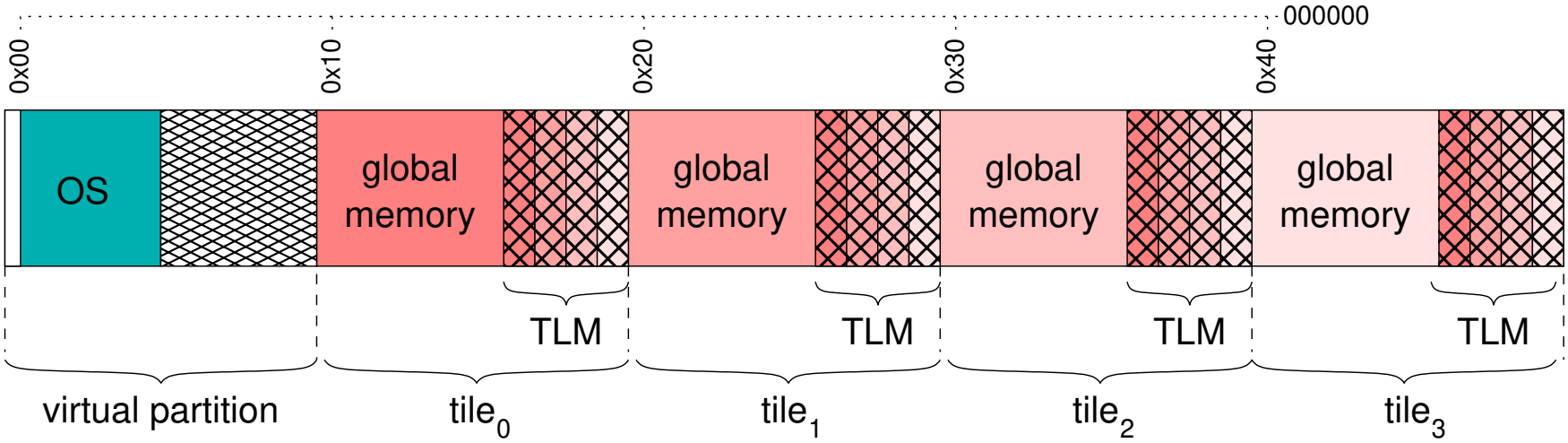
- eine Kachelpartition $tile_i$ definiert sich durch zwei Sektionen:
  - das Abbild der korrespondierenden globalen Hauptspeicherpartition und
  - die TLM-Abbilder aller Kacheln
- der virtuelle Adressraum erfasst so kachellokale und -globale Seiten

PGAS mit TLM



- eine Kachelpartition $tile_i$ definiert sich durch zwei Sektionen:
  - das Abbild der korrespondierenden globalen Hauptspeicherpartition und
  - die TLM-Abbilder aller Kacheln
- der virtuelle Adressraum erfasst so kachellokale und -globale Seiten
  - Zugriffe auf ferne TLM-Abbilder implizieren Nachrichtenversenden
    - den Operandenwert des unterbrochenen Maschinenbefehls fernladen ☹

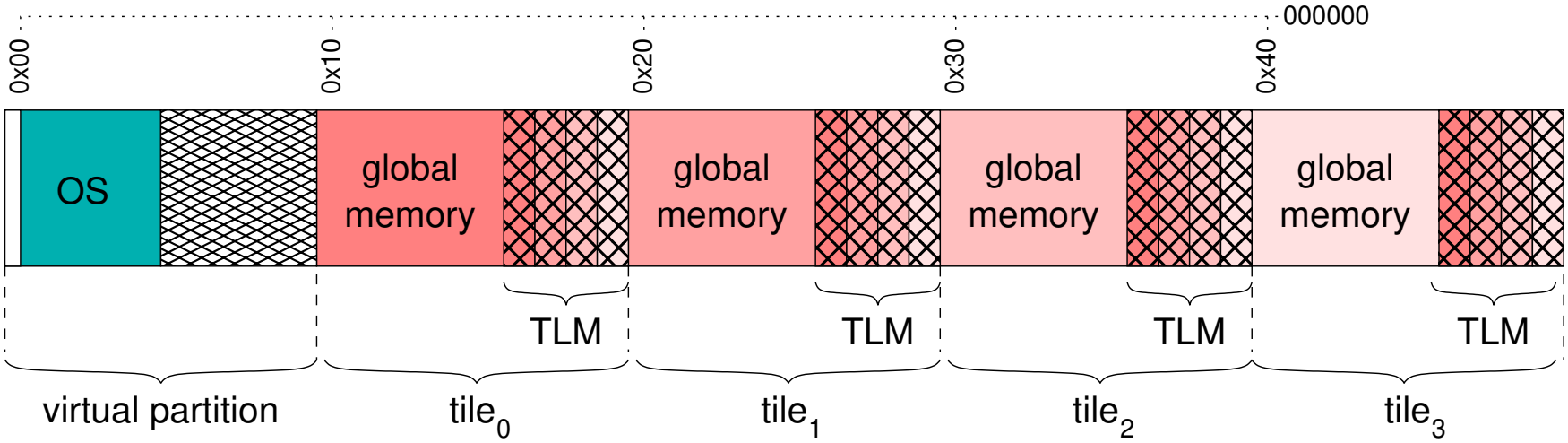
PGAS mit TLM



- eine Kachelpartition $tile_i$ definiert sich durch zwei Sektionen:
  - das Abbild der korrespondierenden globalen Hauptspeicherpartition und
  - die TLM-Abbilder aller Kacheln
- der virtuelle Adressraum erfasst so kachellokale und -globale Seiten

- Zugriffe auf den globalen Hauptspeicher impliziert Datenaustausch
  - jedoch ist vorher für **Konsistenz** der betreffenden Daten zu sorgen

PGAS mit TLM



- eine Kachelpartition $tile_i$ definiert sich durch zwei Sektionen:
  - das Abbild der korrespondierenden globalen Hauptspeicherpartition und
  - die TLM-Abbilder aller Kacheln
- der virtuelle Adressraum erfasst so kachellokale und -globale Seiten

↪ die betreffende Seite fernladen, lokal (TLM) puffern und konsistent halten

Untiefen gemeinsamen Speichers

- Aktionen gleichzeitiger Prozesse sind zeitlich unbestimmt

Untiefen gemeinsamen Speichers

- Aktionen gleichzeitiger Prozesse sind zeitlich unbestimmt

  Kohärenz

  - definiert das Verhalten von Lese- und Schreibvorgängen an einem einzelnen Adressort

    Stellt sicher, dass von einem Prozessor geschriebene Werte von anderen Prozessoren gelesen werden; sagt aber nichts darüber aus, wann Schreibvorgänge sichtbar werden.

  - bezieht sich für gewöhnlich auf den **Cache**

Untiefen gemeinsamen Speichers

- Aktionen gleichzeitiger Prozesse sind zeitlich unbestimmt

Kohärenz

- definiert das Verhalten von Lese- und Schreibvorgängen an einem einzelnen Adressort

Konsistenz

- definiert das Verhalten von Lese- und Schreibvorgängen an logisch zusammenhängenden Adressorten
- bezieht sich für gewöhnlich auf den **Hauptspeicher**

Untiefen gemeinsamen Speichers

- Aktionen gleichzeitiger Prozesse sind zeitlich unbestimmt

  Kohärenz
  - definiert das Verhalten von Lese- und Schreibvorgängen an einem einzelnen Adressort

  Konsistenz
  - definiert das Verhalten von Lese- und Schreibvorgängen an logisch zusammenhängenden Adressorten

- angenommen, folgende Anweisungen werden parallel ausgeführt:

P_1: `init data = 0, done = false;`

```
1  data = '*';
2  done = true;
```

P_2

```
3  while (!done);
4  printf("%d", data);
```

- die Programmlogik gibt Spielraum zur *out-of-order*-Ausführung:

Untiefen gemeinsamen Speichers

- Aktionen gleichzeitiger Prozesse sind zeitlich unbestimmt

  Kohärenz
  - definiert das Verhalten von Lese- und Schreibvorgängen an einem einzelnen Adressort

  Konsistenz
  - definiert das Verhalten von Lese- und Schreibvorgängen an logisch zusammenhängenden Adressorten

- angenommen, folgende Anweisungen werden parallel ausgeführt:

P_1: `init data = 0, done = false;`

```
1  data = '*';
2  done = true;
```

P_2

```
3  while (!done);
4  printf("%d", data);
```

- die Programmlogik gibt Spielraum zur *out-of-order*-Ausführung: Was wird ausgegeben?

Untiefen gemeinsamen Speichers

- Aktionen gleichzeitiger Prozesse sind zeitlich unbestimmt

  Kohärenz
  - definiert das Verhalten von Lese- und Schreibvorgängen an einem einzelnen Adressort

  Konsistenz
  - definiert das Verhalten von Lese- und Schreibvorgängen an logisch zusammenhängenden Adressorten

- angenommen, folgende Anweisungen werden parallel ausgeführt:

P_1: `init data = 0, done = false;`

```
1  data = '*';
2  done = true;
```

P_2

```
3  while (!done);
4  printf("%d", data);
```

- die Programmlogik gibt Spielraum zur *out-of-order*-Ausführung: Was wird ausgegeben? Ein Problem der Kohärenz oder Konsistenz?

Kohärenz versus Konsistenz

- die Umstellung (*reordering*) von Speicheroperationen ist kritisch

Kohärenz versus Konsistenz

- die Umstellung (*reordering*) von Speicheroperationen ist kritisch
  - vorgenommen durch den Kompilierer oder die CPU
  - kann zu unvorhersehbarem Verhalten nichtsequentieller Prozesse führen

Kohärenz versus Konsistenz

- die Umstellung (*reordering*) von Speicheroperationen ist kritisch
  - vorgenommen durch den Kompilierer oder die CPU
  - kann zu unvorhersehbarem Verhalten nichtsequentieller Prozesse führen

↪ die Ausgabe (`printf()`) von `data` ist undefiniert: `0` oder `42`

Kohärenz versus Konsistenz

- die Umstellung (*reordering*) von Speicheroperationen ist kritisch

- Erzwingen einer Ordnungsbeschränkung für Speicheroperationen:

P_1: `init data = 0, done = false;`

```
1  data = '*';
2  fence(WRITE);
3  done = true;
```

P_2

```
4  while (!done);
5  fence(READ);
6  printf("%d", data);
```

  - angenommen, `fence()` setzt eine Speicherbarriere ab

Kohärenz versus Konsistenz

- die Umstellung (*reordering*) von Speicheroperationen ist kritisch

- Erzwingen einer Ordnungsbeschränkung für Speicheroperationen:

P_1: `init data = 0, done = false;`

```
1  data = '*';
2  fence(WRITE);
3  done = true;
```

  - angenommen, `fence()` setzt eine Speicherbarriere ab
    - dann wartet P_1, bis der an `data` geschriebene Wert sichtbar wurde und erst danach kommt die Zuweisung an `done` zur Ausführung

Kohärenz versus Konsistenz

- die Umstellung (*reordering*) von Speicheroperationen ist kritisch

- Erzwingen einer Ordnungsbeschränkung für Speicheroperationen:

P_2

```
4  while (!done);
5  fence(READ);
6  printf("%d", data);
```

  - angenommen, `fence()` setzt eine Speicherbarriere ab

    - entsprechend wartet P_2 bis der neue Wert von `data` sichtbar wurde und erst danach kommt `printf()` zur Ausführung

Kohärenz versus Konsistenz

- die Umstellung (*reordering*) von Speicheroperationen ist kritisch

- Erzwingen einer Ordnungsbeschränkung für Speicheroperationen:

P_1: `init data = 0, done = false;`

```
1  data = '*';
2  fence(WRITE);
3  done = true;
```

P_2

```
4  while (!done);
5  fence(READ);
6  printf("%d", data);
```

  - angenommen, `fence()` setzt eine Speicherbarriere ab

    - entsprechend wartet P_2 bis der neue Wert von `data` sichtbar wurde und erst danach kommt `printf()` zur Ausführung

Kohärenz versus Konsistenz

- die Umstellung (*reordering*) von Speicheroperationen ist kritisch

- Erzwingen einer Ordnungsbeschränkung für Speicheroperationen:

P_1: `init data = 0, done = false;`

```
1  data = '*';
2  fence(WRITE);
3  done = true;
```

P_2

```
4  while (!done);
5  fence(READ);
6  printf("%d", data);
```

  - angenommen, `fence()` setzt eine Speicherbarriere ab

- ein Problem der logischen **Abhängigkeit** zwischen `data` und `done`
  - weder Kompilierer noch CPU können diese erkennen...

☹

Speicherkonsistenz

- in dem **Bezugssystem** sind die Daten widerspruchsfrei und einheitlich

Speicherkonsistenz

- in dem **Bezugssystem** sind die Daten widerspruchsfrei und einheitlich
  - für den hier im Vordergrund stehenden PGAS bedeutet dies, dass Seiten, die im TLM der Kacheln kopiert vorliegen, wertbeständig sein müssen
  - das heißt, replizierte Seiten sind eine logisch in sich geschlossene Einheit

Speicherkonsistenz

- in dem **Bezugssystem** sind die Daten widerspruchsfrei und einheitlich

- in verteilten/parallelen Systemen bleibt **strikte Konsistenz** Illusion
  - benötigt eine befehlsgenaue absolute globale Zeitauflösung, die aber fehlt

Speicherkonsistenz

- in dem **Bezugssystem** sind die Daten widerspruchsfrei und einheitlich
- in verteilten/parallelen Systemen bleibt **strikte Konsistenz** Illusion
- das stärkst-mögliche praktische Modell ist **sequentielle Konsistenz**
  - aufwändig, verbietet viele Optimierungen (insb. *out-of-order*-Ausführung)

Speicherkonsistenz

- in dem **Bezugssystem** sind die Daten widerspruchsfrei und einheitlich
- in verteilten/parallelen Systemen bleibt **strikte Konsistenz** Illusion
- das stärkst-mögliche praktische Modell ist **sequentielle Konsistenz**
- gängig sind gelockerte (*relaxed*) Arten der sequentiellen Konsistenz
  - **Freigabekonsistenz** (*release consistency* [14]; Munin [8, 7])
  - **Eintrittskonsistenz** (*entry consistency* [4])

Speicherkonsistenz

- in dem **Bezugssystem** sind die Daten widerspruchsfrei und einheitlich
- in verteilten/parallelen Systemen bleibt **strikte Konsistenz** Illusion
- das stärkst-mögliche praktische Modell ist **sequentielle Konsistenz**
- gängig sind gelockerte (*relaxed*) Arten der sequentiellen Konsistenz
  - **Freigabekonsistenz** (*release consistency* [14]; Munin [8, 7])
  - **Eintrittskonsistenz** (*entry consistency* [4])

  ↪ basieren beide auf explizit ausgewiesene Programmabschnitte

```
1 enter(way);
2 ...
3 leave(way);
```

Speicherkonsistenz

- in dem **Bezugssystem** sind die Daten widerspruchsfrei und einheitlich
- in verteilten/parallelen Systemen bleibt **strikte Konsistenz** Illusion
- das stärkst-mögliche praktische Modell ist **sequentielle Konsistenz**
- gängig sind gelockerte (*relaxed*) Arten der sequentiellen Konsistenz
  - **Freigabekonsistenz** (*release consistency* [14]; Munin [8, 7])
  - **Eintrittskonsistenz** (*entry consistency* [4])

  ↪ basieren beide auf explizit ausgewiesene Programmabschnitte

```
1 enter(way);
2 ...
3 leave(way);
```

  - mit `way` zur Angabe der benötigten Konsistenz
    - ENTRY beim Abschnittseintritt
    - RELEASE beim Abschnittsaustritt

Speicherkonsistenz

- in dem **Bezugssystem** sind die Daten widerspruchsfrei und einheitlich
- in verteilten/parallelen Systemen bleibt **strikte Konsistenz** Illusion
- das stärkst-mögliche praktische Modell ist **sequentielle Konsistenz**
- gängig sind gelockerte (*relaxed*) Arten der sequentiellen Konsistenz
  - **Freigabekonsistenz** (*release consistency* [14]; Munin [8, 7])
  - **Eintrittskonsistenz** (*entry consistency* [4])

  ↪ basieren beide auf explizit ausgewiesene Programmabschnitte

```
1 enter(way);
2 ...
3 leave(way);
```

  - mit `way` zur Angabe der benötigten Konsistenz
    - ENTRY beim Abschnittseintritt
    - RELEASE beim Abschnittsaustritt
- eine Art **kritischer Abschnitt**, gemeinhin auch genau damit kombiniert

Gliederung

Kachelbasierte Mehrkernprozessoren

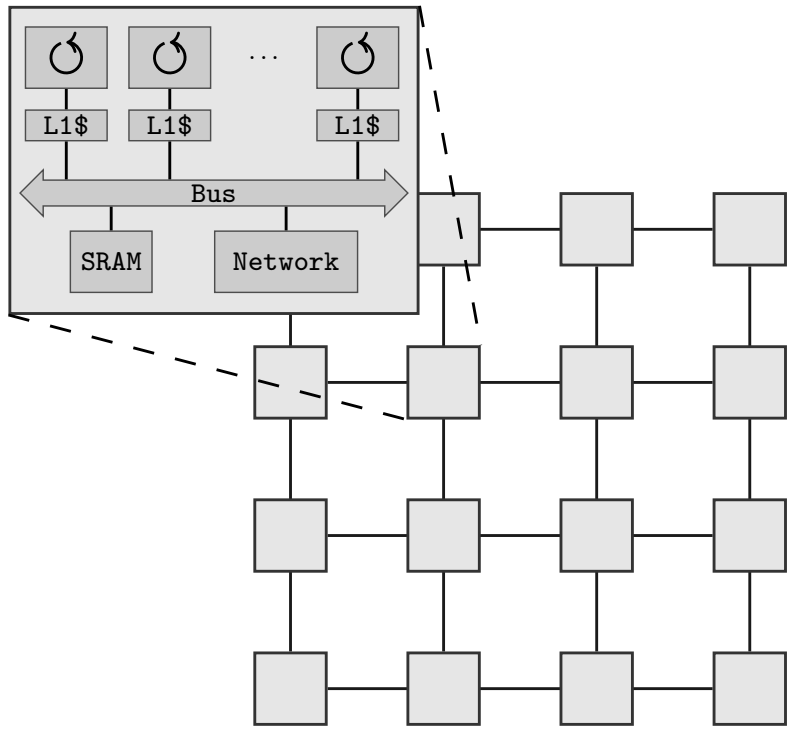
Kachelbasierte Mehrkernprozessoren

- die Kachelverbindung sei als **NoC** (*network on chip*) realisiert
  - skalierbares, verteiltes Arbitrierungsschema zur *on-chip*-Kommunikation

Kachelbasierte Mehrkernprozessoren

- die Kachelverbindung sei als **NoC** (*network on chip*) realisiert

  - paketvermittelte Nachrichten
    - verbindungslos/-orientiert
    - Durchsatz-/Latenzgarantien
  - globale Speicherzugriffe
    - gemeinsamer Adressraum (PGAS)
    - systemweite `load`/`store`
    - *cache*-inkohärent
  - ggf. mehrere Sprünge (*hops*)
    - Latenz variiert mit Sprunganzahl

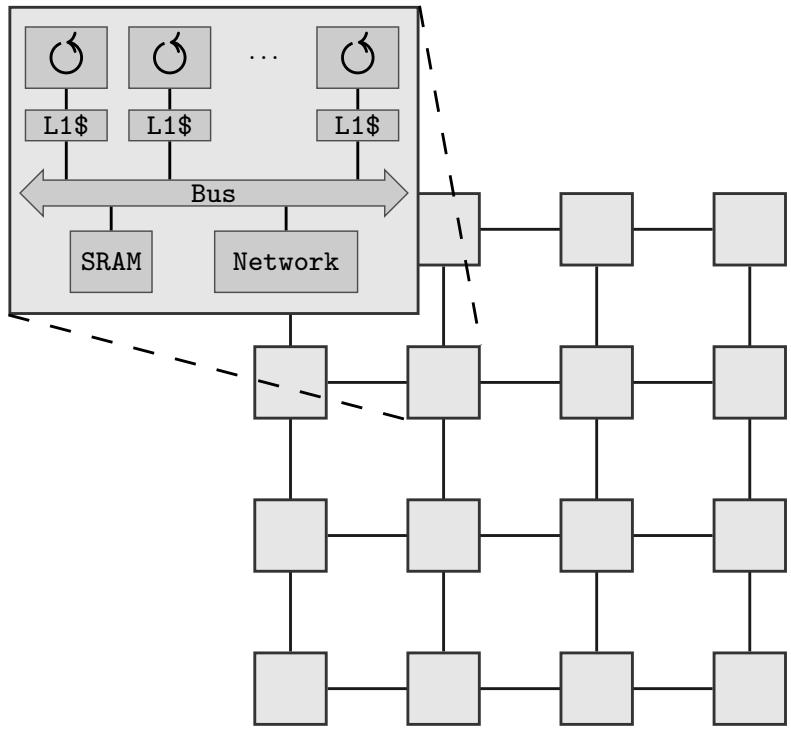


Kachelbasierte Mehrkernprozessoren

- die Kachelverbindung sei als **NoC** (*network on chip*) realisiert

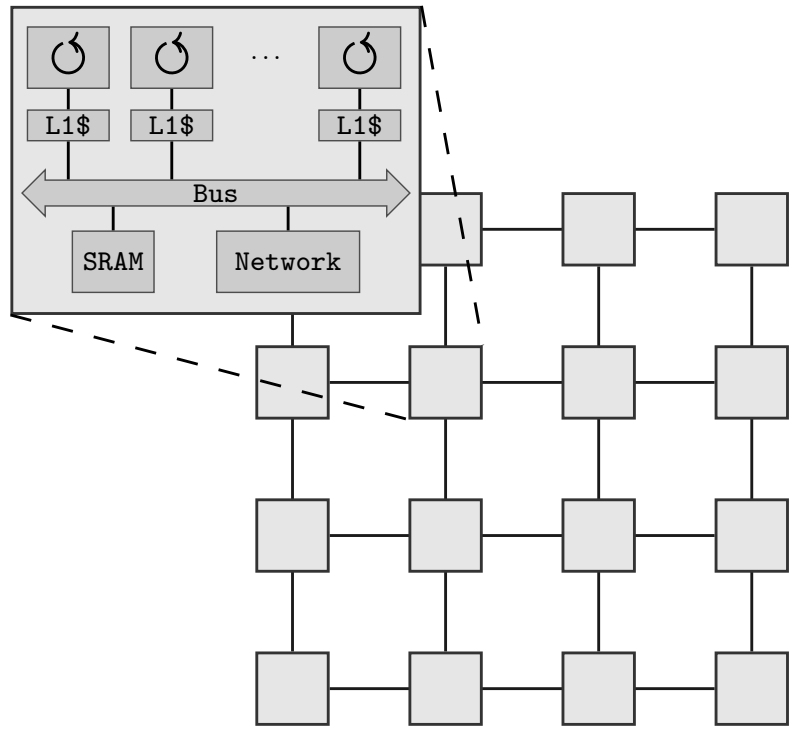
  - paketvermittelte Nachrichten
    - verbindungslos/-orientiert
    - Durchsatz-/Latenzgarantien
  - globale Speicherzugriffe
    - gemeinsamer Adressraum (PGAS)
    - systemweite `load`/`store`
    - *cache*-inkohärent
  - ggf. mehrere Sprünge (*hops*)
    - Latenz variiert mit Sprunganzahl

  ↪ effektiv inkohärente NUMA



Kachelbasierte Mehrkernprozessoren

- die Kachelverbindung sei als **NoC** (*network on chip*) realisiert

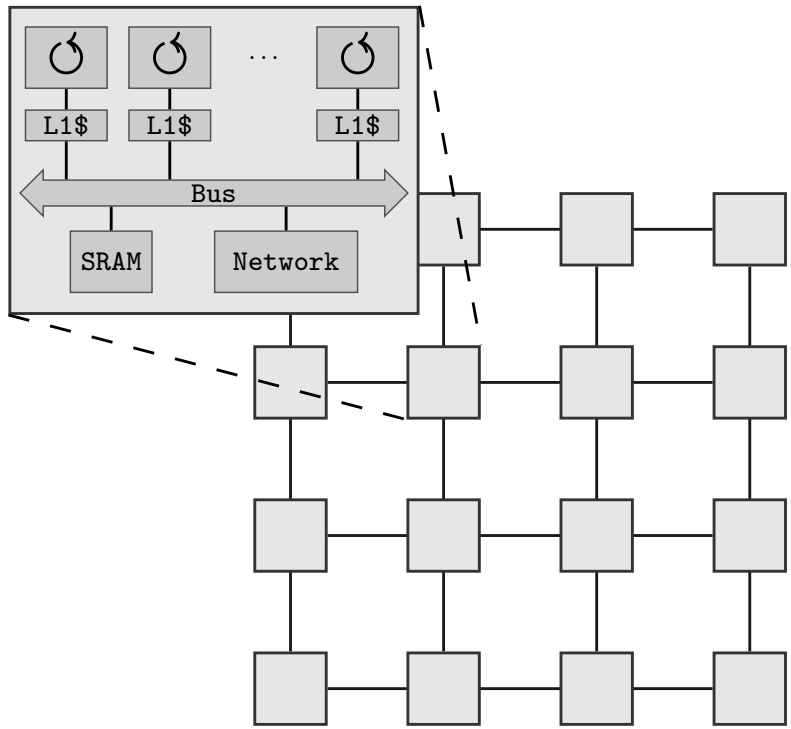


- vernetzt werden Kacheln unterschiedlicher Funktion/Auslegung, insb.:

Rechenkacheln	Prozessorkerne, *cache*-kohärenter lokaler Speicher
Speicherkacheln	gemeinsamer Speicher, kachelzentrisch *cache*-kohärent

Kachelbasierte Mehrkernprozessoren

- die Kachelverbindung sei als **NoC** (*network on chip*) realisiert



- vernetzt werden Kacheln unterschiedlicher Funktion/Auslegung, insb.:
  - Rechenkacheln: Prozessorkerne, *cache*-kohärenter lokaler Speicher
  - Speicherkacheln: gemeinsamer Speicher, kachelzentrisch *cache*-kohärent
- lokale wie auch globale Speicherbereiche sind im PGAS abgebildet

Pulsierende parallele Prozesse

Pulsierende parallele Prozesse

Angenommen sei ein nichtsequentielles Programm, das jedoch in der Ausführung sequentiell beginnt.

Pulsierende parallele Prozesse

Angenommen sei ein nichtsequentielles Programm, das jedoch in der Ausführung sequentiell beginnt.

Ein auf Basis dieses Programms definierter Prozess startet einfädig, er kann sich gemäß Programmvorschrift und abhängig vom dynamischen Zustand zum mehrfädigen, nichtsequentiellen Prozess entwickeln.

Pulsierende parallele Prozesse

Angenommen sei ein nichtsequentielles Programm, das jedoch in der Ausführung sequentiell beginnt.

Ein auf Basis dieses Programms definierter Prozess startet einfädig, er kann sich gemäß Programmvorschrift und abhängig vom dynamischen Zustand zum mehrfädigen, nichtsequentiellen Prozess entwickeln.

- der **Urfaden** erzeugt die zur parallelen Berechnung benötigten Fäden
  - die ihrerseits pyramidal von sich aus weitere Fäden erzeugen können

Pulsierende parallele Prozesse

Angenommen sei ein nichtsequentielles Programm, das jedoch in der Ausführung sequentiell beginnt.

Ein auf Basis dieses Programms definierter Prozess startet einfädig, er kann sich gemäß Programmvorschrift und abhängig vom dynamischen Zustand zum mehrfädigen, nichtsequentiellen Prozess entwickeln.

- der **Urfaden** erzeugt die zur parallelen Berechnung benötigten Fäden
- jeder **Kindfaden** wird bei Erzeugung initial auf einen Kern platziert
  - je nach Anwendungsbeschränkungen und der Ressourcenverwaltung

Pulsierende parallele Prozesse

Angenommen sei ein nichtsequentielles Programm, das jedoch in der Ausführung sequentiell beginnt.

Ein auf Basis dieses Programms definierter Prozess startet einfädig, er kann sich gemäß Programmvorschrift und abhängig vom dynamischen Zustand zum mehrfädigen, nichtsequentiellen Prozess entwickeln.

- der **Urfaden** erzeugt die zur parallelen Berechnung benötigten Fäden
- jeder **Kindfaden** wird bei Erzeugung initial auf einen Kern platziert
- pro Anwendung gilt der erste Faden einer Kachel als **Prinzipal**
  - die für anwendungsbezogene kachellokale Aufgaben „authentifizierbare“ Entität; der Faden, der dem Betriebssystem als Bezugspunkt dient

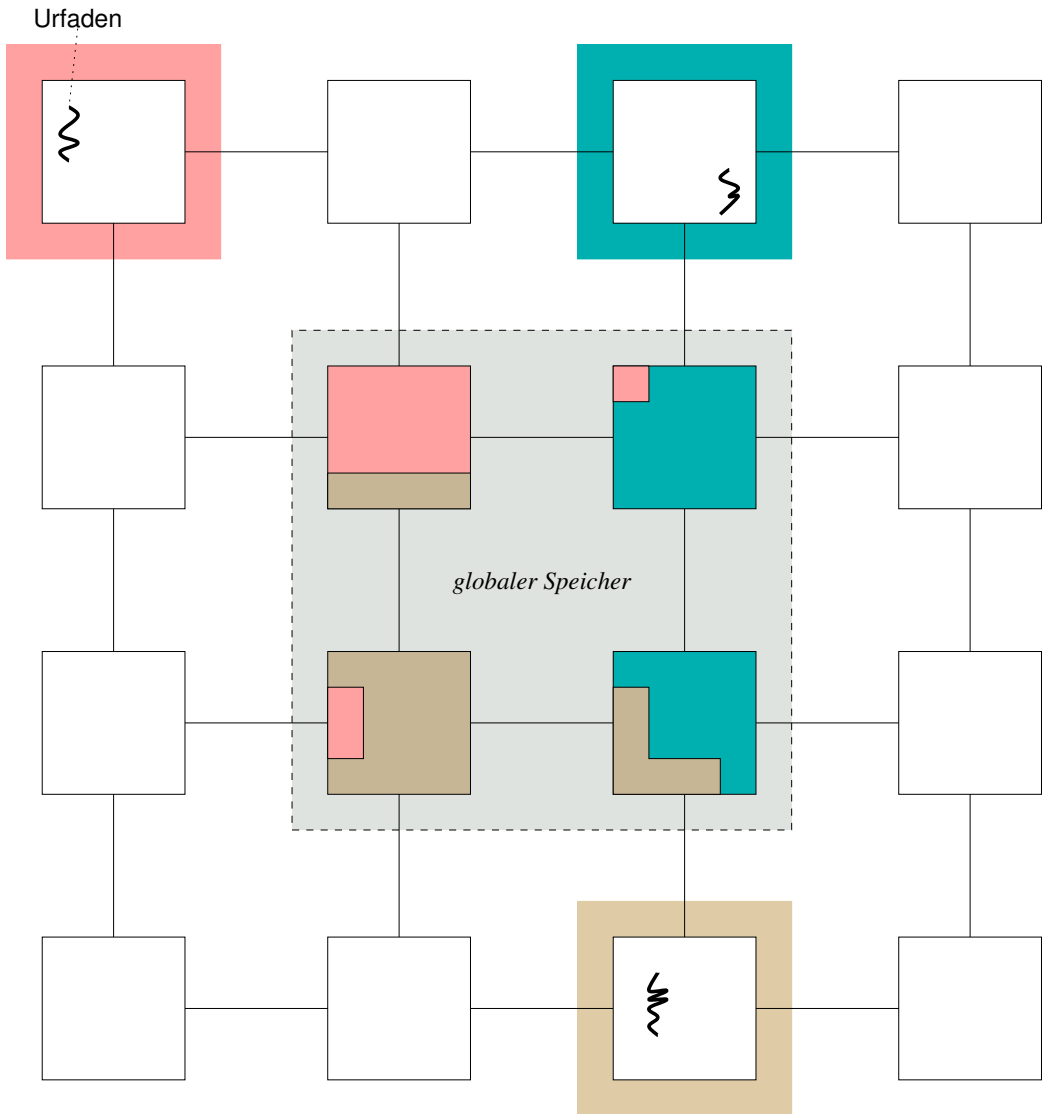
Pulsierende parallele Prozesse

Angenommen sei ein nichtsequentielles Programm, das jedoch in der Ausführung sequentiell beginnt.

Ein auf Basis dieses Programms definierter Prozess startet einfädig, er kann sich gemäß Programmvorschrift und abhängig vom dynamischen Zustand zum mehrfädigen, nichtsequentiellen Prozess entwickeln.

- der **Urfaden** erzeugt die zur parallelen Berechnung benötigten Fäden
- jeder **Kindfaden** wird bei Erzeugung initial auf einen Kern platziert
- pro Anwendung gilt der erste Faden einer Kachel als **Prinzipal**
  - die für anwendungsbezogene kachellokale Aufgaben „authentifizierbare“ Entität; der Faden, der dem Betriebssystem als Bezugspunkt dient
  - ↪ für den PGAS einer Anwendung der als Besitzer (*owner*) geltende Faden von Seiten als Referenzen, von denen Replikate gezogen werden können

Mehrfädiger Mehrprogrammbetrieb



3 Anwendungen
- Anwendungsprogramme

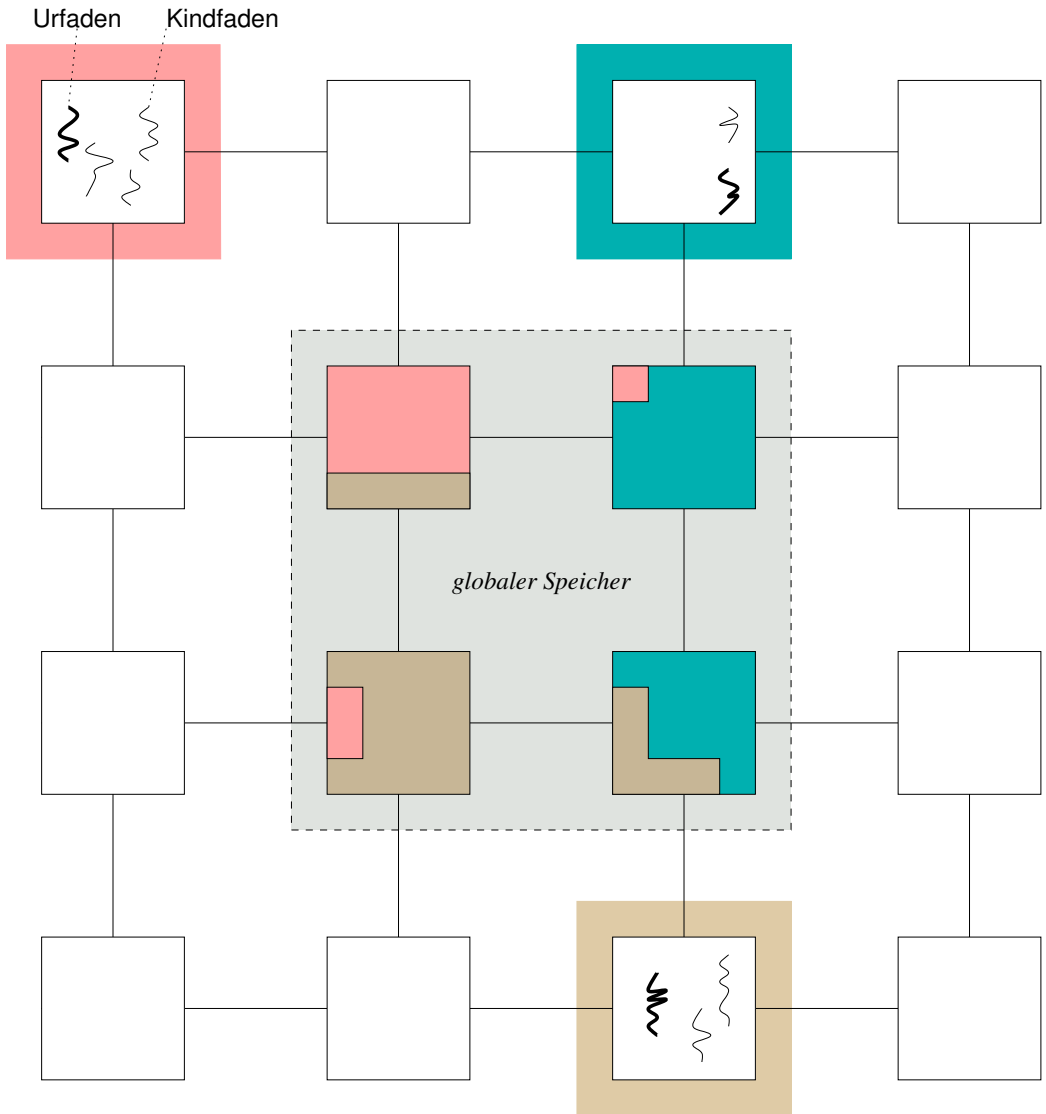
- je ein PGAS
- jeweils isoliert

12 Rechenkacheln

- lokale Speicher
  - im PGAS abgebildet

4 Speicherkacheln
- partiell geteilt
- im PGAS abgebildet

Mehrfädiger Mehrprogrammbetrieb

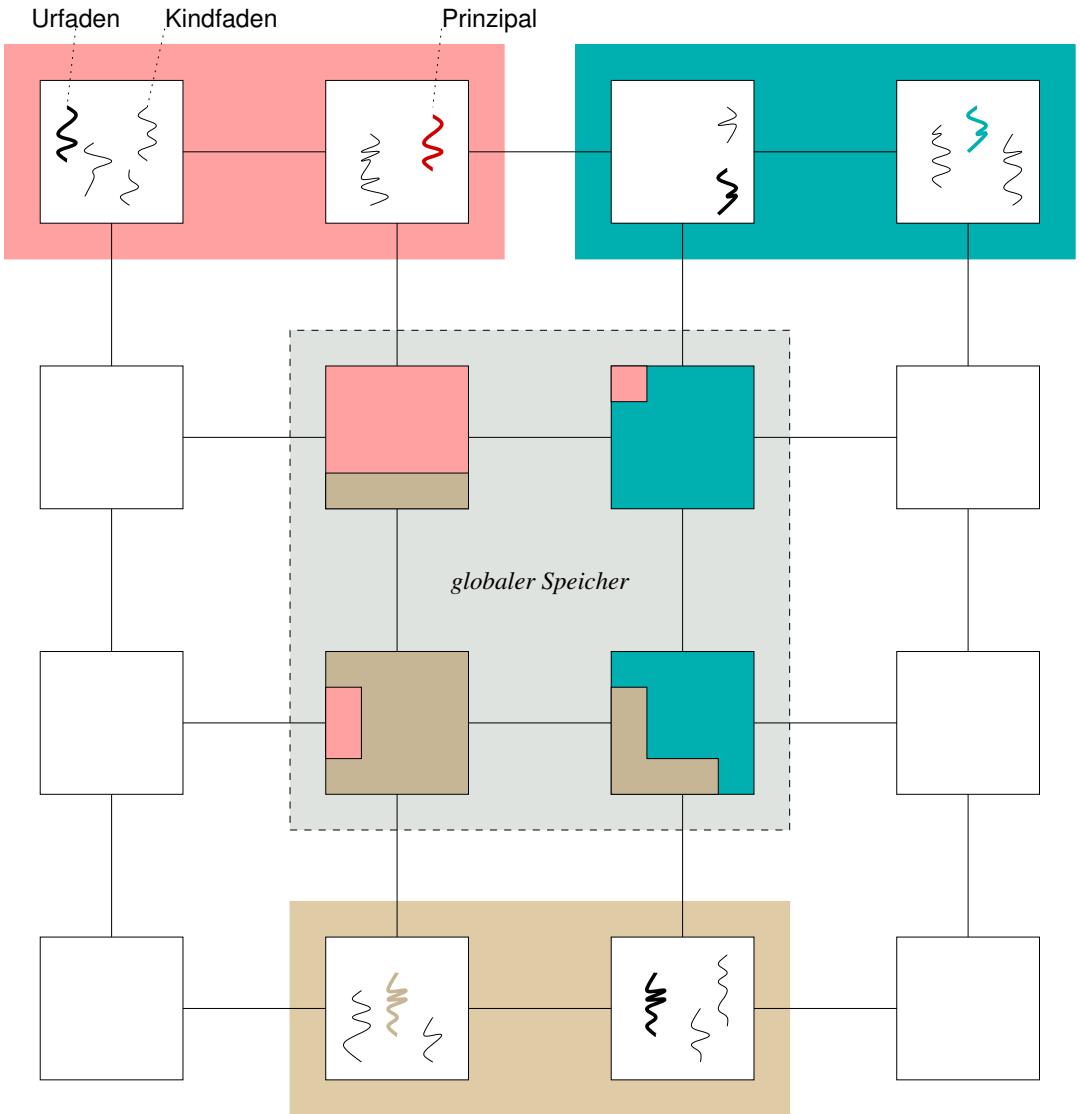


3 Anwendungen
- Anwendungsprogramme
  - mehrfädig
- je ein PGAS
- jeweils isoliert

12 Rechenkacheln
- lokale Speicher
  - im PGAS abgebildet

4 Speicherkacheln
- partiell geteilt
- im PGAS abgebildet

Mehrfädiger Mehrprogrammbetrieb

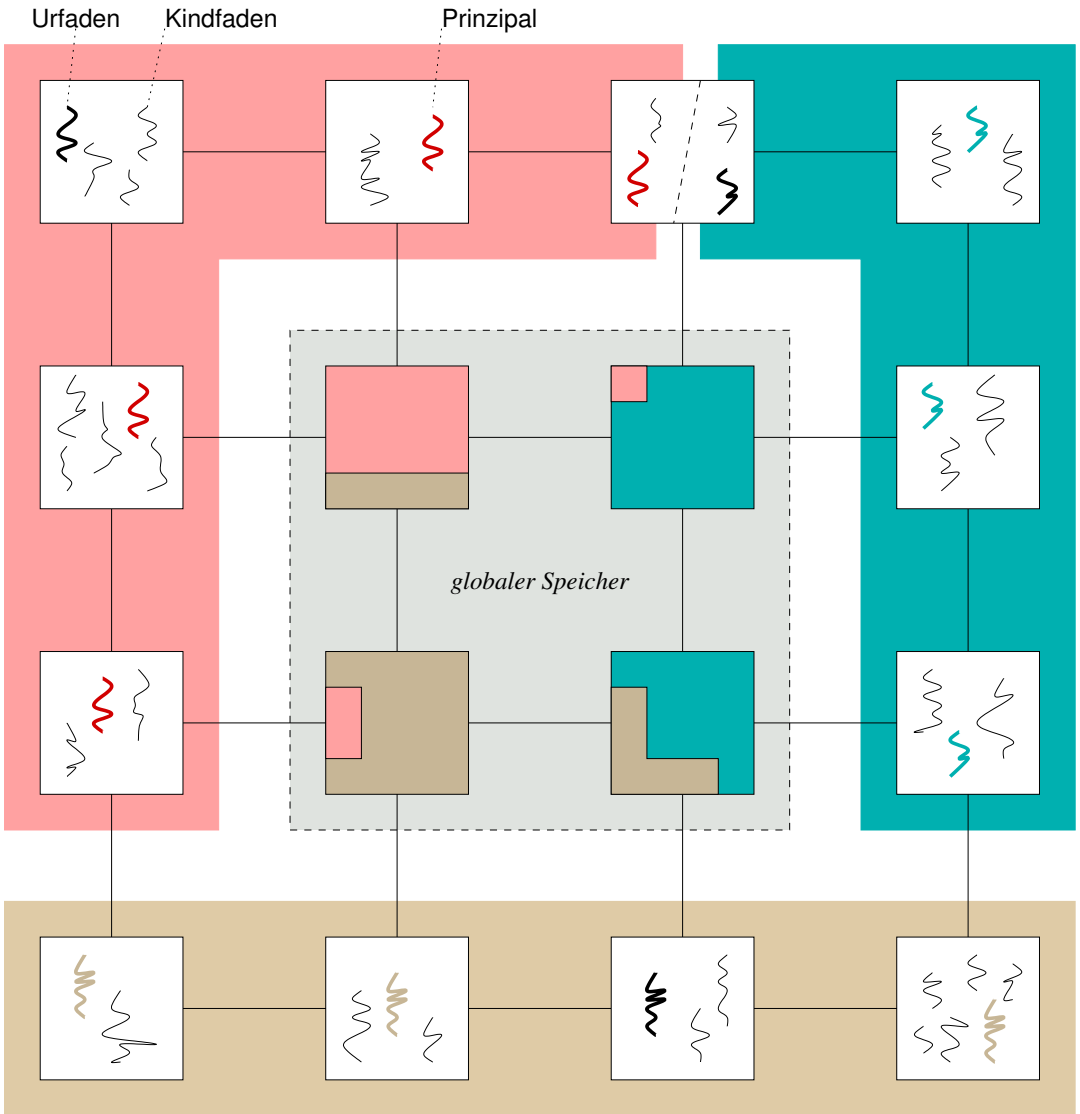


3 Anwendungen
- Anwendungsprogramme
  - mehrfädig
- je ein PGAS
- jeweils isoliert

12 Rechenkacheln
- lokale Speicher
  - im PGAS abgebildet

4 Speicherkacheln
- partiell geteilt
- im PGAS abgebildet

Mehrfädiger Mehrprogrammbetrieb



3 Anwendungen
- Anwendungsprogramme
  - mehrfädig
- je ein PGAS
- jeweils isoliert

12 Rechenkacheln
- partiell geteilt
- lokale Speicher
  - im PGAS abgebildet

4 Speicherkacheln
- partiell geteilt
- im PGAS abgebildet

Adressraumbelegungsplan

vgl. S. 9

Adressraumbelegungsplan

vgl. S. 9

- prinzipiell ist aller vorhandener Hauptspeicher im PGAS abgebildet
  - der in Rechenkacheln verfügbare lokale Speicher (TLM) und der in den Speicherkacheln oder andererseits verfügbare globale Speicher
  - dabei bleibt ein TLM vorrangig den kachellokalen Aktionen vorbehalten

Adressraumbelegungsplan

vgl. S. 9

- prinzipiell ist aller vorhandener Hauptspeicher im PGAS abgebildet

- die Adressraumpartitionen sind gleich groß und gleich strukturiert
  - Dimension und Aufbau der betreffenden Adressbereiche sind identisch
    - ein fester Abschnitt von Adressen, die auf globalen Speicher abbilden
    - eine feste Anzahl solcher Abschnitte, die auf lokalen Speicher abbilden

    ↪ letztere ist durch die Kachelanzahl oder eine kleinere Größe begrenzt
  - damit haben alle Bereiche dieselbe relative Anfangsadresse im PGAS

Adressraumbelegungsplan

vgl. S. 9

- prinzipiell ist aller vorhandener Hauptspeicher im PGAS abgebildet
- die Adressraumpartitionen sind gleich groß und gleich strukturiert
- die Größe des darüber abgebildeten Hauptspeichers variiert jedoch

  minimal
  - die kleinste **Arbeitsmenge** (*working set* [11, S. 326]) von Seiten aller in den Rechenkacheln verorteten Prozessen

  maximal
  - die Summe an Seitenrahmen über TLM und globalen Speicher

Adressraumbelegungsplan

vgl. S. 9

- prinzipiell ist aller vorhandener Hauptspeicher im PGAS abgebildet
- die Adressraumpartitionen sind gleich groß und gleich strukturiert
- die Größe des darüber abgebildeten Hauptspeichers variiert jedoch

  minimal
  - die kleinste **Arbeitsmenge** (*working set* [11, S. 326]) von Seiten aller in den Rechenkacheln verorteten Prozessen

  maximal
  - die Summe an Seitenrahmen über TLM und globalen Speicher

  - wobei klassischer virtueller Speicher [12] einmal außen vor bleiben soll...

TLM als Seitenpuffer

page cache

TLM als Seitenpuffer

page cache

- **Nahzugriffe** auf den TLM haben eine vergleichsweise geringe Latenz
  - einerseits gehen diese Zugriffe über den kachellokalen Bus, nicht das NoC
  - andererseits ist der TLM für gewöhnlich aus SRAM aufgebaut
    - die **Zykluszeit** von SRAM ist 10 bis 20 mal schneller als die von DRAM[1]
    - bei gleicher Technologie hat DRAM aber eine 5 bis 10 mal höhere **Kapazität**

[1] 60ns – 100ns

TLM als Seitenpuffer

page cache

- **Nahzugriffe** auf den TLM haben eine vergleichsweise geringe Latenz
  - einerseits gehen diese Zugriffe über den kachellokalen Bus, nicht das NoC
  - andererseits ist der TLM für gewöhnlich aus SRAM aufgebaut
    - die **Zykluszeit** von SRAM ist 10 bis 20 mal schneller als die von DRAM[1]
    - bei gleicher Technologie hat DRAM aber eine 5 bis 10 mal höhere **Kapazität**
  - demgegenüber setzen sich die Speicherkacheln aus DRAM zusammen

[1] 60ns – 100ns

TLM als Seitenpuffer

page cache

- **Nahzugriffe** auf den TLM haben eine vergleichsweise geringe Latenz

- dies motiviert, den TLM als **Zwischenspeicher** (*cache*) zu nutzen
  - wie gewöhnlicher RAM auch Zwischenspeicher für virtuellen Speicher ist
  - jedoch wandern die Seiten nunmehr innerhalb der Hauptspeicherebene

TLM als Seitenpuffer

page cache

- **Nahzugriffe** auf den TLM haben eine vergleichsweise geringe Latenz
- dies motiviert, den TLM als **Zwischenspeicher** (*cache*) zu nutzen
- globaler Speicher/Speicherkacheln bilden eine Art **Vorratsspeicher**
  - die Quelle der Text- und Datenbestände der Anwendungsprogramme

TLM als Seitenpuffer

page cache

- **Nahzugriffe** auf den TLM haben eine vergleichsweise geringe Latenz
- dies motiviert, den TLM als **Zwischenspeicher** (*cache*) zu nutzen
- globaler Speicher/Speicherkacheln bilden eine Art **Vorratsspeicher**
  - auf Nachfrage (*on-demand*) werden Seiten daraus in den TLM umgelagert
    - nicht unähnlich einer „*first-touch policy*“ [18] in NUMA-Systemen

TLM als Seitenpuffer

page cache

- **Nahzugriffe** auf den TLM haben eine vergleichsweise geringe Latenz
- dies motiviert, den TLM als **Zwischenspeicher** (*cache*) zu nutzen
- globaler Speicher/Speicherkacheln bilden eine Art **Vorratsspeicher**
  - auf Nachfrage (*on-demand*) werden Seiten daraus in den TLM umgelagert
    - nicht unähnlich einer „*first-touch policy*“ [18] in NUMA-Systemen
  - umgekehrt wandern Seiten im Ersetzungsfall (*replacement*) ggf. zurück
    - nämlich modifizierte Seiten, nachdem ihre **Konsistenz** sichergestellt ist

Konsistenzwahrung von Seiten

consistency preservation

- auf **Seitenumlagerung** (*paging*) basierender gemeinsamer Speicher ist mit zwei grundsätzlichen **Randbedingungen** konfrontiert

false sharing
- gemeinsame Seite, aber keine gemeinsamen Variablen

shared data
- gemeinsame Seite, mindestens eine gemeinsame Variable

Konsistenzwahrung von Seiten

consistency preservation

- auf **Seitenumlagerung** (*paging*) basierender gemeinsamer Speicher ist mit zwei grundsätzlichen **Randbedingungen** konfrontiert

false sharing

  - gemeinsame Seite, aber keine gemeinsamen Variablen
  - die Variablen sind nur „zufällig“ derselben Seite zugeordnet
  - unkoordinierte Variablenzugriffe wären Normalität

Konsistenzwahrung von Seiten

consistency preservation

- auf **Seitenumlagerung** (*paging*) basierender gemeinsamer Speicher ist mit zwei grundsätzlichen **Randbedingungen** konfrontiert

false sharing

- gemeinsame Seite, aber keine gemeinsamen Variablen
- die Variablen sind nur „zufällig“ derselben Seite zugeordnet
- unkoordinierte Variablenzugriffe wären Normalität

↪ **Leistungsabfall** wegen unnötiger Konsistenzwahrung [5, 6]

Konsistenzwahrung von Seiten

consistency preservation

- auf **Seitenumlagerung** (*paging*) basierender gemeinsamer Speicher ist mit zwei grundsätzlichen **Randbedingungen** konfrontiert

shared data

- gemeinsame Seite, mindestens eine gemeinsame Variable
- Variablenmodifikationen bilden einen kritischen Abschntt
- koordinierte Variablenzugriffe dürfen erwartet werden

Konsistenzwahrung von Seiten

consistency preservation

- auf **Seitenumlagerung** (*paging*) basierender gemeinsamer Speicher ist mit zwei grundsätzlichen **Randbedingungen** konfrontiert

shared data

- gemeinsame Seite, mindestens eine gemeinsame Variable
- Variablenmodifikationen bilden einen kritischen Abschntt
- koordinierte Variablenzugriffe dürfen erwartet werden

↪ **schwache Konsistenz**, an kritischen Abschnitten [14, 4]

Konsistenzwahrung von Seiten

consistency preservation

- auf **Seitenumlagerung** (*paging*) basierender gemeinsamer Speicher ist mit zwei grundsätzlichen **Randbedingungen** konfrontiert

  false sharing
  - gemeinsame Seite, aber keine gemeinsamen Variablen

  shared data
  - gemeinsame Seite, mindestens eine gemeinsame Variable

- der erste Fall (*false sharing*) lässt sich **funktional transparent** für die betreffenden Prozesse durch die Seitenpufferverwaltung behandeln
  - Konsistenz durch Generierung und Zusammenführen von Differenzmasken

Konsistenzwahrung von Seiten

consistency preservation

- auf **Seitenumlagerung** (*paging*) basierender gemeinsamer Speicher ist mit zwei grundsätzlichen **Randbedingungen** konfrontiert

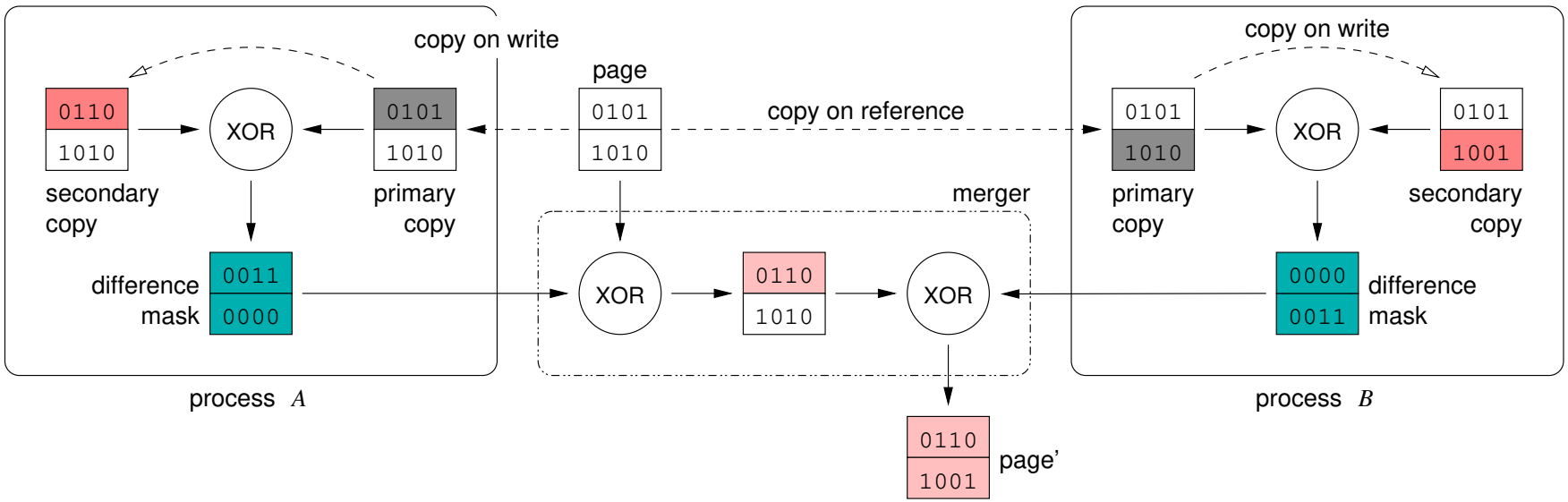
  false sharing
  - gemeinsame Seite, aber keine gemeinsamen Variablen

  shared data
  - gemeinsame Seite, mindestens eine gemeinsame Variable

- der erste Fall (*false sharing*) lässt sich **funktional transparent** für die betreffenden Prozesse durch die Seitenpufferverwaltung behandeln
  - Konsistenz durch Generierung und Zusammenführen von Differenzmasken
- der zweite Fall (*shared data*) setzt auf die von den Prozessen ohnehin zu treffenden Koordinierungsmaßnahmen und erweitert diese
  - **Eintrittskonsistenz** [4], und zwar im eben genannten Mischverfahren

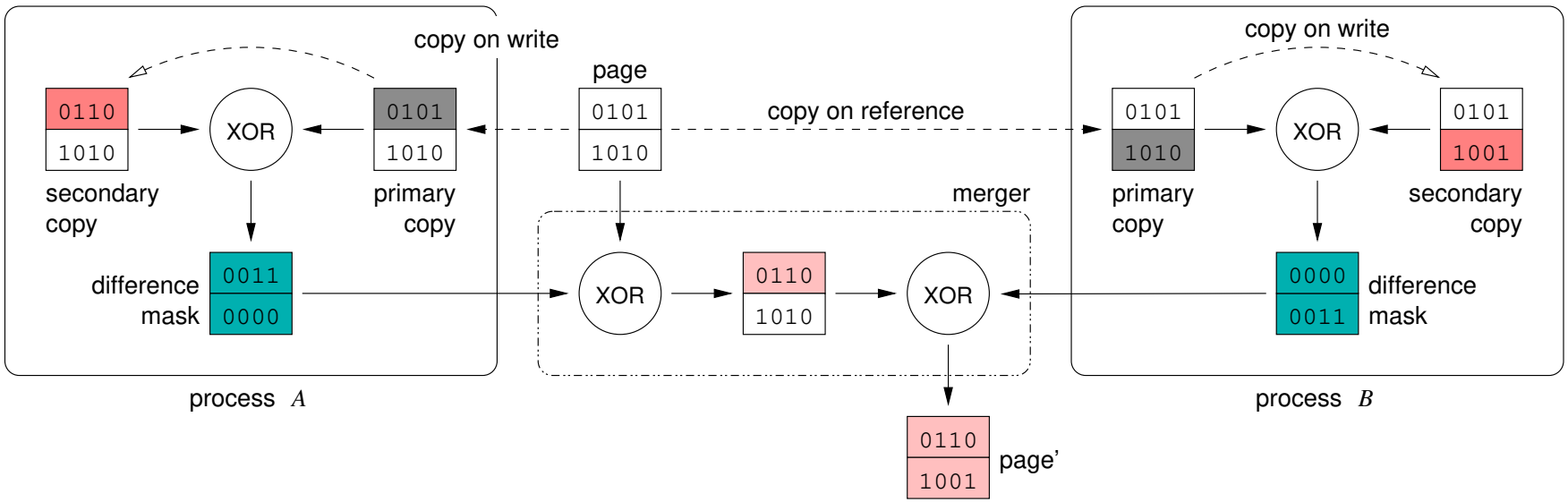
Konsistenzwahrung durch Differenzmasken

- fälschliche gemeinsame Nutzung (*false sharing*) tolerieren, indem eine Seite an mehreren Stellen zugleich modifiziert werden darf
  - mehrere Leseprozesse erlauben, aber auch mehrere Schreibprozesse



Konsistenzwahrung durch Differenzmasken

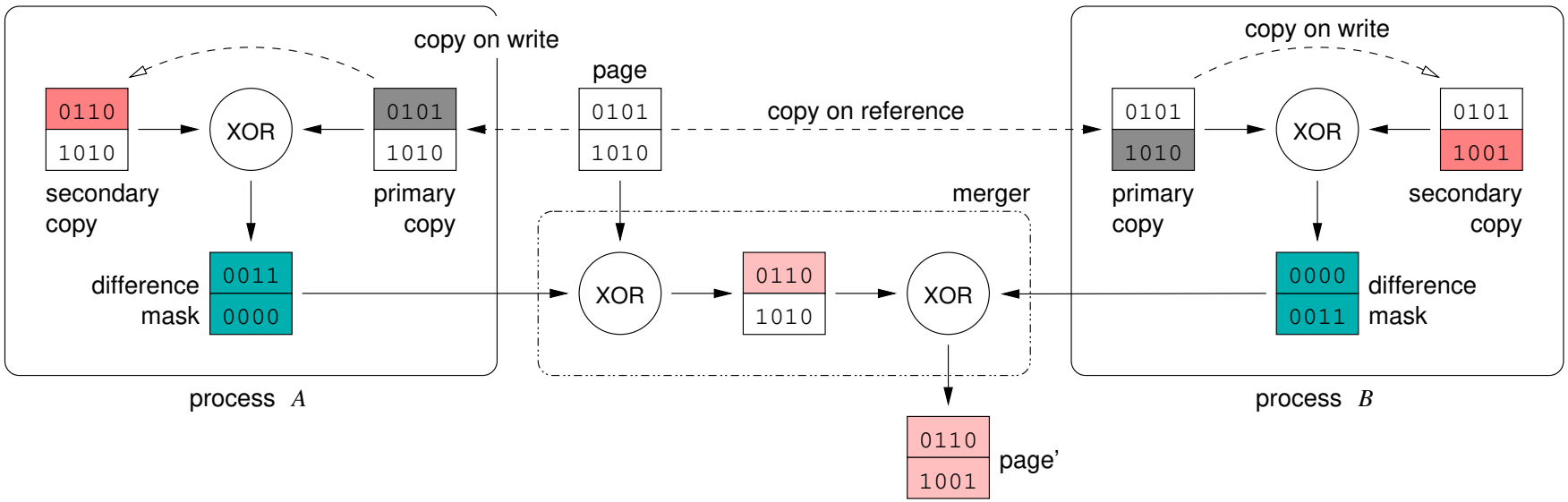
- fälschliche gemeinsame Nutzung (*false sharing*) tolerieren, indem eine Seite an mehreren Stellen zugleich modifiziert werden darf
  - mehrere Leseprozesse erlauben, aber auch mehrere Schreibprozesse



- Prozesse *A* und *B* teilen sich dieselbe Seite, nicht aber dasselbe Wort

Konsistenzwahrung durch Differenzmasken

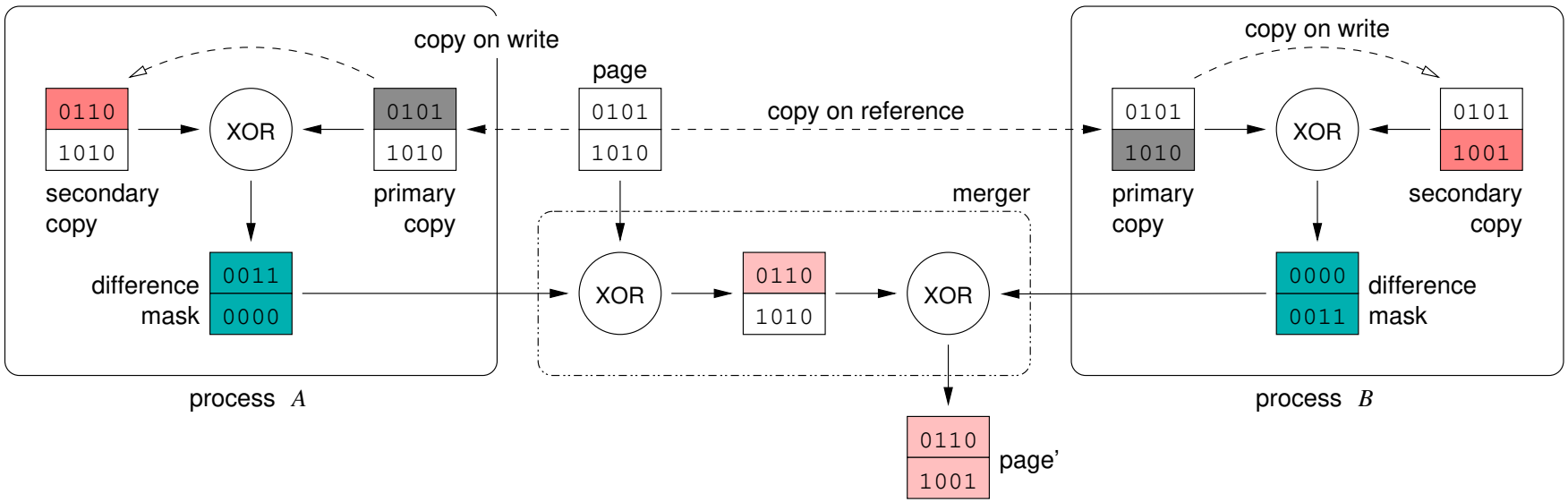
- fälschliche gemeinsame Nutzung (*false sharing*) tolerieren, indem eine Seite an mehreren Stellen zugleich modifiziert werden darf
  - mehrere Leseprozesse erlauben, aber auch mehrere Schreibprozesse



  - Prozesse *A* und *B* teilen sich dieselbe Seite, nicht aber dasselbe Wort
  - beide Prozesse machen ihre Änderungen unabhängig voneinander und bilden dann eine **Differenzmaske**: XOR von Primär- und Sekundärkopie

Konsistenzwahrung durch Differenzmasken

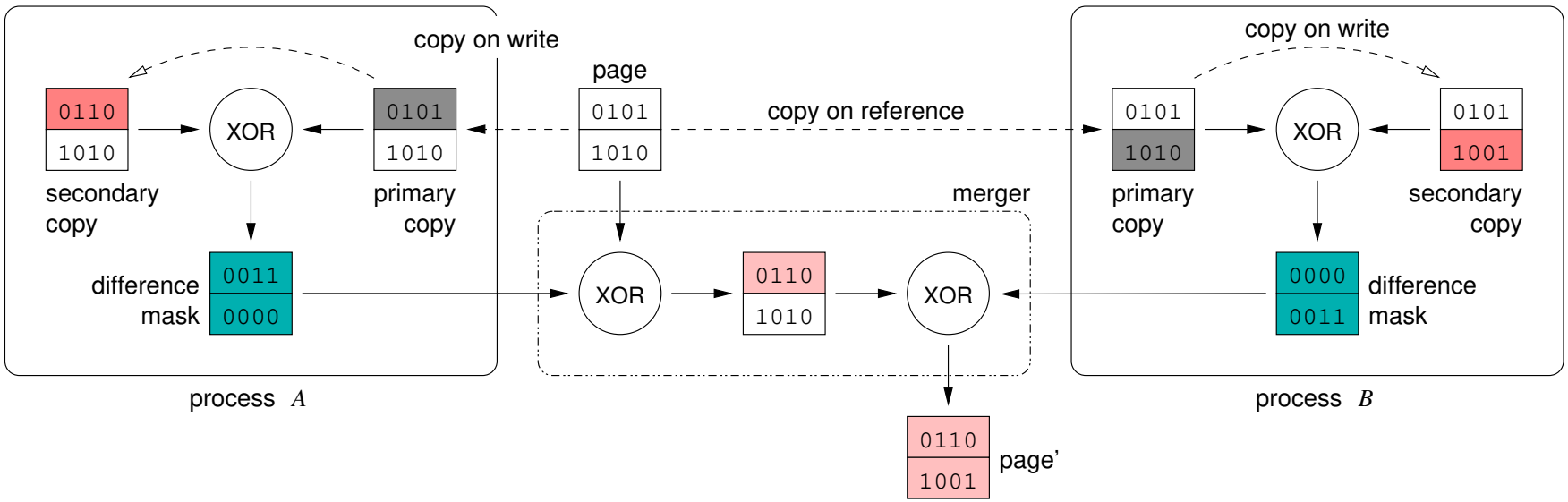
- fälschliche gemeinsame Nutzung (*false sharing*) tolerieren, indem eine Seite an mehreren Stellen zugleich modifiziert werden darf
  - mehrere Leseprozesse erlauben, aber auch mehrere Schreibprozesse



- Prozesse *A* und *B* teilen sich dieselbe Seite, nicht aber dasselbe Wort
- beide Prozesse machen ihre Änderungen unabhängig voneinander und bilden dann eine **Differenzmaske**: XOR von Primär- und Sekundärkopie
- dann werden die Masken nacheinander mittels XOR auf die Referenzseite angewandt, um letztere zu aktualisieren

Konsistenzwahrung durch Differenzmasken

- fälschliche gemeinsame Nutzung (*false sharing*) tolerieren, indem eine Seite an mehreren Stellen zugleich modifiziert werden darf
  - mehrere Leseprozesse erlauben, aber auch mehrere Schreibprozesse



  - Prozesse *A* und *B* teilen sich dieselbe Seite, nicht aber dasselbe Wort
  - beide Prozesse machen ihre Änderungen unabhängig voneinander und bilden dann eine **Differenzmaske**: XOR von Primär- und Sekundärkopie
  - dann werden die Masken nacheinander mittels XOR auf die Referenzseite angewandt, um letztere zu aktualisieren
  - Resultat ist eine neue konsistente Seite mit den geänderten Worten darauf

Moment der Konsistenzwahrung

- Konsistenz einer Seite ist sicherzustellen, wann immer sie aus den Seitenpuffer entfernt wird

Moment der Konsistenzwahrung

- Konsistenz einer Seite ist sicherzustellen, wann immer sie aus den Seitenpuffer entfernt wird, und zwar:

  implizit
  - als Folge der **Ersetzungsstrategie** (*replacement policy*)
  - wenn die Seite wegen Überbelegung verdrängt werden soll

Moment der Konsistenzwahrung

- Konsistenz einer Seite ist sicherzustellen, wann immer sie aus den Seitenpuffer entfernt wird, und zwar:

  implizit
  - als Folge der **Ersetzungsstrategie** (*replacement policy*)

  explizit
  - auf Veranlassung des Prozesses selbst, wenn die Seite
    - an einen anderen Prozess kommuniziert werden soll oder
    - allgemein zur freien Verfügung durch andere gestellt wird
  - das heißt, als Folge einer **Freigabeoperation** des Prozesses

Moment der Konsistenzwahrung

- Konsistenz einer Seite ist sicherzustellen, wann immer sie aus den Seitenpuffer entfernt wird, und zwar:

  implizit
  - als Folge der **Ersetzungsstrategie** (*replacement policy*)

  explizit
  - auf Veranlassung des Prozesses selbst

- die Seite wird „ausgespült“ und schrittweise „gemischt“ (S. 20)
- im Ergebnis entsteht die konsistente **Referenzseite** im globalen Speicher

Moment der Konsistenzwahrung

- Konsistenz einer Seite ist sicherzustellen, wann immer sie aus den Seitenpuffer entfernt wird, und zwar:
  - implizit
    - als Folge der **Ersetzungsstrategie** (*replacement policy*)
  - explizit
    - auf Veranlassung des Prozesses selbst

- der explizite Fall betrifft auch die **Koordinierung** gleichzeitiger Lese- und Schreibzugriffe auf geteilte Daten (*shared data*)

Moment der Konsistenzwahrung

- Konsistenz einer Seite ist sicherzustellen, wann immer sie aus den Seitenpuffer entfernt wird, und zwar:
  - implizit
    - als Folge der **Ersetzungsstrategie** (*replacement policy*)
  - explizit
    - auf Veranlassung des Prozesses selbst

- der explizite Fall betrifft auch die **Koordinierung** gleichzeitiger Lese- und Schreibzugriffe auf geteilte Daten (*shared data*)
  - blockierende Synchronisation ist vergleichsweise einfach, da der kritische Abschnitt sequentiell durchlaufen wird $\rightsquigarrow$ **Eintrittskonsistenz** [4]

Moment der Konsistenzwahrung

- Konsistenz einer Seite ist sicherzustellen, wann immer sie aus den Seitenpuffer entfernt wird, und zwar:

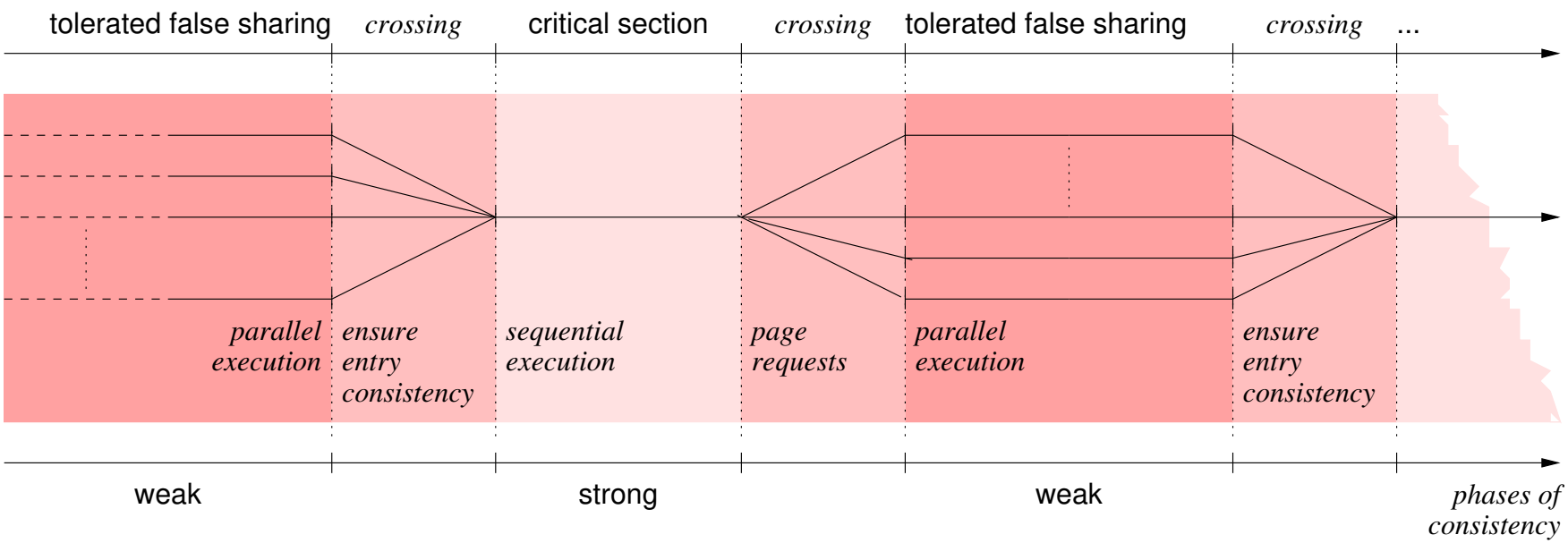
  implizit
  - als Folge der **Ersetzungsstrategie** (*replacement policy*)

  explizit
  - auf Veranlassung des Prozesses selbst

- der explizite Fall betrifft auch die **Koordinierung** gleichzeitiger Lese- und Schreibzugriffe auf geteilte Daten (*shared data*)

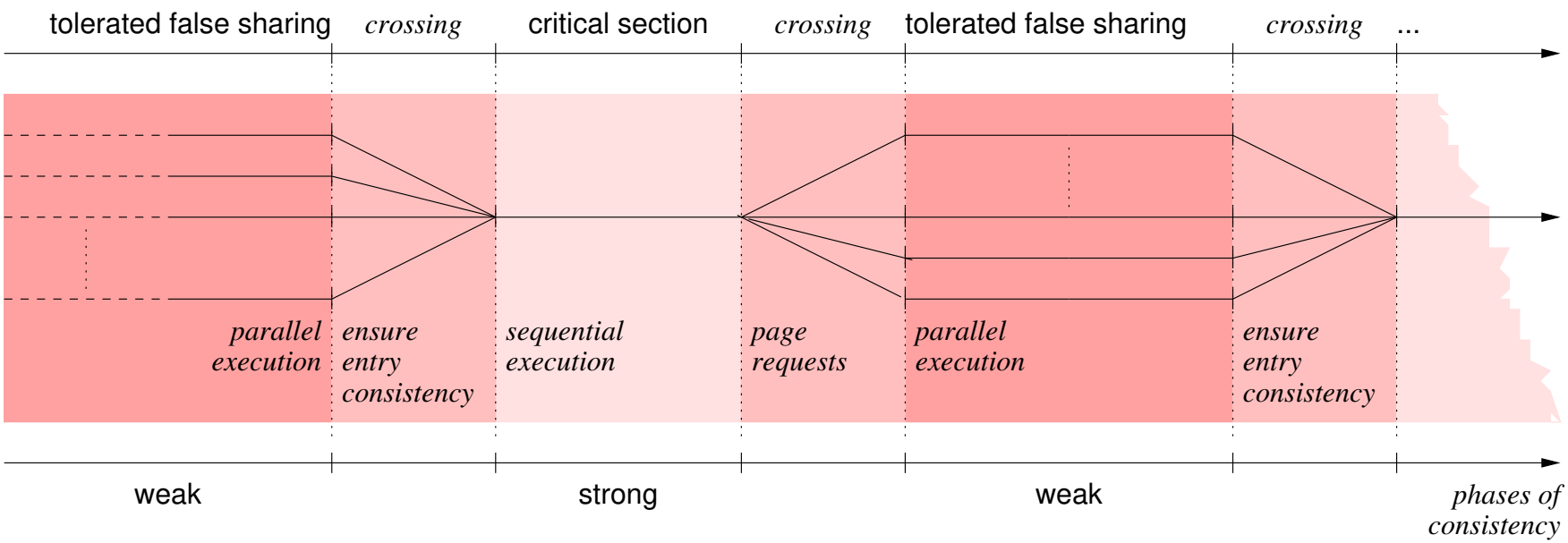
  - nichtblockierende Synchronisation ist — trotz Vorteile — problematisch, da die Operationen nur **Signifikanz** für den globalen Speicher hätten
    - hohe Latenz (vgl. S. 18), Gefahr von Seitenflattern (*page thrashing*)

Phasen von Konsistenz



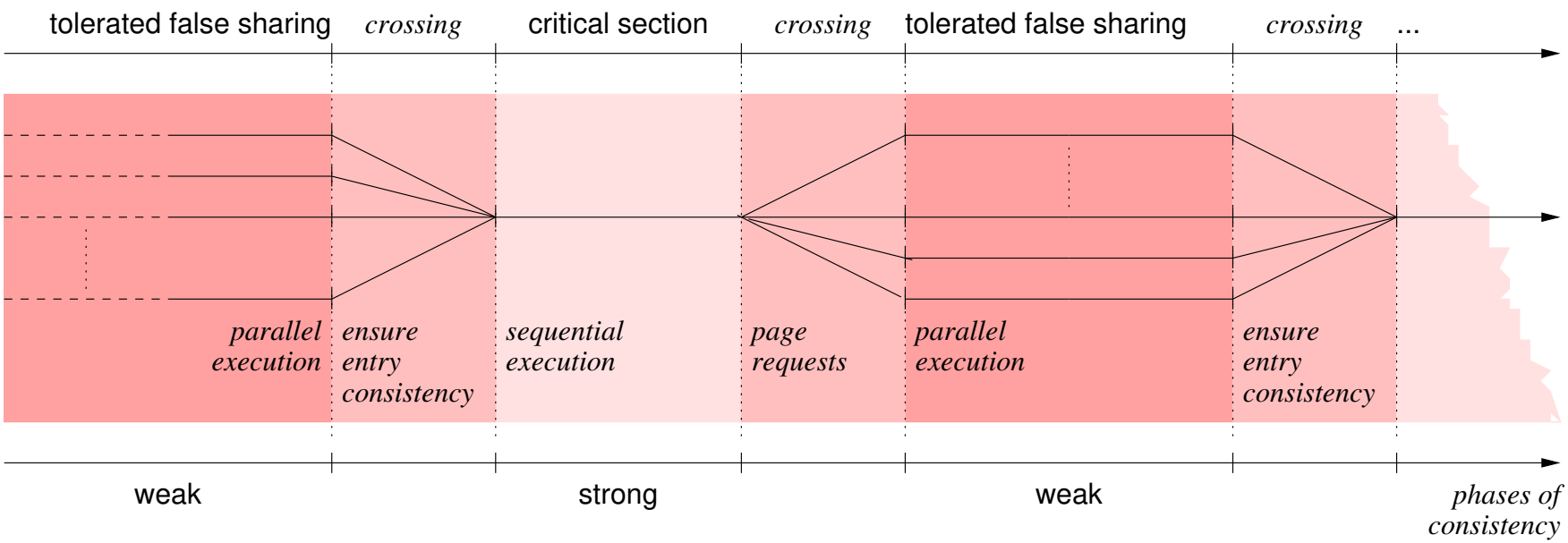
- Speicherkonsistenz eines nichtsequentiellen (mehrfädigen) Prozesses

Phasen von Konsistenz



- Speicherkonsistenz eines nichtsequentiellen (mehrfädigen) Prozesses
  - wenn im Prozessverlauf kritische Abschnitte passiert werden müssen
  - innerhalb des Abschnitts arbeitet immer nur ein Prozess auf den Seiten
  - diese Seiten liegen im TLM des den Abschnitt haltenden Prozessfadens
    - sofern seine Arbeitsmenge an Seiten in den TLM passt

Phasen von Konsistenz



- Speicherkonsistenz eines nichtsequentiellen (mehrfädigen) Prozesses

- fälschliche gemeinsame Nutzung (*false sharing*) von Seiten erlaubt einen höheren Grad an Parallelität des Prozesses

Gliederung

SVM vs. DSM vs. VSM

- Prozess(fäd)en die Sicht eines gemeinsam nutzbaren Hauptspeichers geben, obwohl letzterer physisch verteilt ist

SVM vs. DSM vs. VSM

- Prozess(fäd)en die Sicht eines gemeinsam nutzbaren Hauptspeichers geben, obwohl letzterer physisch verteilt ist

  SVM
  - gemeinsam genutzter virtueller Speicher (*shared virtual memory*)
  - Ivy [21, 22]: seitenbasiert, Serversystem
  - virtueller Speicher erweitert um RAM-globale Seitenumlagerung

SVM vs. DSM vs. VSM

- Prozess(fäd)en die Sicht eines gemeinsam nutzbaren Hauptspeichers geben, obwohl letzterer physisch verteilt ist

DSM
- verteilter gemeinsamer Speicher (*distributed shared memory*)
- Mirage [13], TreadMarks [1]: seitenbasiert, OS
  - virtueller Speicher erweitert um RAM-globale Seitenumlagerung
- vor allem aber variablen- oder objektbasiert $\rightsquigarrow$ Laufzeitsysteme

SVM vs. DSM vs. VSM

- Prozess(fäd)en die Sicht eines gemeinsam nutzbaren Hauptspeichers geben, obwohl letzterer physisch verteilt ist

VSM
- virtuell gemeinsam genutzter Speicher (*virtual shared memory*)
- PEACE [3, 15], Vote [9, 10], OctoPOS [20]: seitenbasiert, OS
- Seitenumlagerung innerhalb der Hauptspeicherebene (RAM-global)

SVM vs. DSM vs. VSM

- Prozess(fäd)en die Sicht eines gemeinsam nutzbaren Hauptspeichers geben, obwohl letzterer physisch verteilt ist

- manch seitenbasierter Ansatz behandelt „*false sharing*“ in Software
  - mit [17, 16] kommt Konsistenzwahrung durch Differenzmasken in Hardware

SVM vs. DSM vs. VSM

- Prozess(fäd)en die Sicht eines gemeinsam nutzbaren Hauptspeichers geben, obwohl letzterer physisch verteilt ist

  SVM
  - gemeinsam genutzter virtueller Speicher (*shared virtual memory*)

  DSM
  - verteilter gemeinsamer Speicher (*distributed shared memory*)

  VSM
  - virtuell gemeinsam genutzter Speicher (*virtual shared memory*)

- drei **Systemabstraktionen**, die bezüglich der Seitenbasierung einiges gemeinsam haben und teils **synonyme Konzepte** sind

Resümee

Resümee

- eine **verteilte Speicherarchitektur**, die für die Anwendungssoftware so aussieht, als wäre sie ein *shared-memory*-System
  - einerseits physisch verteilte Hauptspeichereinheiten $\leadsto$ verteiltes System
  - andererseits partiell *cache*-inkohärentes Verhalten der Hardware

Resümee

- eine **verteilte Speicherarchitektur**, die für die Anwendungssoftware so aussieht, als wäre sie ein *shared-memory*-System

- Speicher- und Adressraumkonzept für einen MPSoC (*multiprocessor system on a chip*), der auf **massive Parallelität** setzt
  - vor allem vielfädige nichtsequentielle Prozesse, auch ruhig wenige
  - eher nicht viele geringfädige Prozess, oder gar nur sequentielle

Resümee

- eine **verteilte Speicherarchitektur**, die für die Anwendungssoftware so aussieht, als wäre sie ein *shared-memory*-System
- Speicher- und Adressraumkonzept für einen MPSoC (*multiprocessor system on a chip*), der auf **massive Parallelität** setzt
- zugänglich ist der Hauptspeicher über einen PGAS (*partitioned global address space*), in dem die Prozessfäden operieren
  - eine Partition enthält das Abbild eines globalen Hauptspeicherbereichs
  - plus Abbilder der kachellokalen Speicher (*tile-local memory*, TLM)

Resümee

- eine **verteilte Speicherarchitektur**, die für die Anwendungssoftware so aussieht, als wäre sie ein *shared-memory*-System
- Speicher- und Adressraumkonzept für einen MPSoC (*multiprocessor system on a chip*), der auf **massive Parallelität** setzt
- zugänglich ist der Hauptspeicher über einen PGAS (*partitioned global address space*), in dem die Prozessfäden operieren
- der TLM dient vor allem als **Seitenpuffer** (*page cache*), er fungiert in erster Linie als **Zwischenspeicher** (*cache*) ferner Seiten
  - kohärent, schwach konsistent und „*false sharing*“ tolerierend/behandelnd
  - Konsistenzwahrung im Betriebssystem, auf Basis von Differenzmasken

Literaturverzeichnis I

[1] Amza, C. ; Cox, A. ; Dwarkadas, S. ; Keleher, P. ; Lu, H. ; Rajamony, R. ; Yu, W. ; Zwaenepoel, W. :
TreadMarks: Shared Memory Computing on Networks of Workstations.
In: *IEEE Computer* 29 (1996), Nr. 2, S. 18–28.
http://dx.doi.org/10.1109/2.485843. –
DOI 10.1109/2.485843

[2] Baumann, A. ; Barham, P. ; Dagand, P.-E. ; Harris, T. ; Isaacs, R. ; Peter, S. ; Roscoe, T. ; Schüpbach, A. ; Singhania, A. :
The Multikernel: A New OS Architecture for Scalable Multicore Systems.
In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*.
Association for Computing Machinery (SOSP '09). –
ISBN 9781605587523, 29–44

[3] Berg, R. ; Cordsen, J. ; Nolte, J. ; Heuer, J. ; Oestmann, B. ; Sander, M. ; Schmidt, H. ; Schön, F. ; Schröder-Preikschat, W. :
The PEACE Family of Distributed Operating Systems / GMD FIRST.
Berlin, Germany : GMD FIRST, 1991. –
Forschungsbericht

Literaturverzeichnis II

[4] Bershad, B. N. ; Zekauskas, M. J.:
Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors / School of Computer Science, Carnegie Mellon University.
Pittsburgh, PA 15213, Sept. 1991 (CMU-CS-9M70). – Forschungsbericht

[5] Bolosky, W. J.:
Software Coherence in Multiprocessor Memory Systems, Department of Computer Science, University of Rochester, Ph.D. thesis, 1993.
https://archive.org/details/nasa_techdoc_19940016799

[6] Bolosky, W. J. ; Scott, M. L.:
False sharing and its effect on shared memory performance.
In: *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, USENIX Association, 3:1–3:15

[7] Carter, J. B.:
Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency.
Houston, Texas, USA, Rice University, Diss., Sept. 1993

Literaturverzeichnis III

[8] Carter, J. B. ; Bennett, J. K. ; Zwaenepoel, W. :
Implementation and Performance of Munin.
In: *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*.
New York, NY, USA : Association for Computing Machinery, 1991 (SOSP '91). – ISBN 0897914473, S. 152–164

[9] Cordsen, J. :
Basing Virtually Shared Memory on a Family of Consistency Models.
In: *Proceedings of the IPPS Workshop on Support for Large-Scale Shared Memory Architectures*, 1994, S. 58–72

[10] Cordsen, J. :
Virtuell gemeinsamer Speicher, Technische Universität Berlin, Diss., 1996

[11] Denning, P. J.:
The Working Set Model for Program Behavior.
In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 323–333

[12] Denning, P. J.:
Virtual Memory.
In: *Computing Surveys* 2 (1970), Sept., Nr. 3, S. 153–189

Literaturverzeichnis IV

[13] Fleisch, B. ; Popek, G. :
Mirage: A Coherent Distributed Shared Memory Design.
In: *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*.
New York, NY, USA : Association for Computing Machinery, 1989 (SOSP '89). –
ISBN 0897913388, S. 211–223

[14] Gharachorloo, K. ; Lenoski, D. ; Laudon, J. ; Gibbons, P. ; Gupta, A. ; Hennessy, J. :
Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.
In: *Proceedings of the 17th Annual International Symposium on Computer Architecture*.
New York, NY, USA : Association for Computing Machinery, 1990 (ISCA '90). –
ISBN 0897913663, S. 15–26

[15] Giloi, W. K. ; Hastedt, C. ; Schön, F. ; Schröder-Preikschat, W. :
A Distributed Implementation of Shared Virtual Memory With Strong and Weak Coherence.
In: *Proceedings of the 2nd Euronean Conference on Distributed Memory Computing (EDMCC2)*, Springer-Verlag, 1991. –
ISBN 3–540–53951–4, S. 23–31

Literaturverzeichnis V

[16] Intel Corporation (Hrsg.):
Intel© Data Streaming Accelerator Architecture Specification.
1.2.
Santa Clara, CA, USA: Intel Corporation, 9 2021.
https://software.intel.com/en-us/download/intel-data-streaming-accelerator-preliminary-architecture-specification.
–
Document Number: 341204-003US

[17] Jiang, D. :
Introducing the Intel© Data Streaming Accelerator (Intel© DSA).
https://01.org/blogs/2019/introducing-intel-data-streaming-accelerator, Nov. 2019

[18] Lameter, C. :
NUMA (Non-Uniform Memory Access): An Overview.
In: *ACM Queue* 11 (2013), Jul., Nr. 7, S. 40–51.
http://dx.doi.org/10.1145/2508834.2513149. –
DOI 10.1145/2508834.2513149. –
ISSN 1542–7730

Literaturverzeichnis VI

[19] Langer, T. :
Memory Management in Massive Parallel Non-Coherent Systems, Lehrstuhl für Verteilte Systeme und Betriebssysteme, Department Informatik, Friedrich-Alexander-Universität Erlangen-Nürnberg, Upcoming Dissertation, 2022

[20] Langer, T. ; Rabenstein, J. ; Hönig, T. ; Schröder-Preikschat, W. :
No Coherence? No Problem! Virtual Shared Memory for MPSoCs.
In: *2021 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, IEEE, 2021 (2021 SC Workshops Supplementary Proceedings (SCWS)), S. 33–41

[21] Li, K. :
Shared Virtual Memory on Loosely Coupled Multiprocessors.
New Haven, Connecticut, USA, Department of Comupter Science, Yale University, Diss., Okt. 1986

[22] Li, K. ; Hudak, P. :
Memory Coherence in Shared Virtual Memory Systems.
In: *ACM Transactions on Computer Systems* 7 (1989), Nov., Nr. 4, S. 321–359

Literaturverzeichnis VII

[23] Löhr, K.-P. ; Fink, T. :
Einführung und Übersicht.
In: Institut für Informatik (Hrsg.): *Verteilte Systene*.
Freie Universität Berlin, 2001 (Vorlesungsfolien), Kapitel 1

[24] Taylor, M. B. ; Lee, W. ; Miller, J. E. ; Wentzlaff, D. ; Bratt, I. ; Greenwald, B. ; Hoffmann, H. ; Johnson, P. R. ; Kim, J. S. ; Psota, J. ; Saraf, A. ; Shnidman, N. ; Strumpen, V. ; Frank, M. I. ; Amarasinghe, S. ; Agarwal, A. :
Tiled Multicore Processors.
In: Keckler, S. W. (Hrsg.) ; Olukotun, K. (Hrsg.) ; Hofstee, H. P. (Hrsg.): *Multicore Processors and Systems*.
Boston, MA : Springer US, 2009. –
ISBN 978–1–4419–0263–4, Kapitel 1, S. 1–33