Punkt-zu-Punkt Event-Synchronisation

- Methoden in Software
  1. Interrupts: Signalbehandlung
  2. Busy-Waiting: normale Variablen als Flaggen
  3. Blocking: Nutzung von Semaphoren
- Beispielsweise Unterstützung in Hardware fürs *Readers-Writers-Problem*
  - *Full-Empty Bit* für jedes Wort im Hauptspeicher
    - Gesetzt, wenn Wort mit neu produzierten Daten belegt (d.h. *Write*)
    - Zurückgesetzt, wenn Wort konsumiert wurde (d.h. *Read*)
    - Verwendung für Producer-Consumer Synchronisation auf Wort-Ebene
    - Hardware liefert atomare Bit-Manipulation bei *Read* und *Write*
  - Problem: Flexibilität
    - Mehrere *Reads* oder mehrere *Writes* bevor *Consumer* liest?
    - Komplexere Datenstrukturen aus mehreren Wörtern?
    - Benötigt Unterstützung in der Programmiersprache!

Hardware Barrieren

- Zusätzliches „verdrahtetes UND" auf dem Bus
  - Eingang beim Erreichen setzen; alle Eingänge gesetzt, dann setze Ausgang
  - Alle warten auf Setzen des Ausgangs
- In der Praxis, mehrere Drähte für mehrfachen Gebrauch
- Immer nur global wirksam (feste Menge an Teilnehmern)
- Nützlich bei häufig verwendeten globalen *Barrieren*

Probleme:

- beliebige Untermengen an Prozessoren
- mehrere Prozesse auf einem Prozessor
- ändernde Teilnehmeranzahl und Identitäten (z.B. Migration)

Heute selten in Bus-basierten Systemen zu finden

Deshalb im Folgenden: Software-Algorithmen für Barrieren mit einfachen Hardware-Primitiven

Eine zentralisierte Barriere (1)

Zähler für Anzahl der im *Barrier* befindenden Prozesse und eine Signalisierungsflagge:

```
struct bar_type
  {int counter=0; struct lock_type lock; int flag = 0;} bar_name;          /* wait until flag == 1 */

BARRIER (bar_name, p) {                                                    /* barrier with p callers */
        LOCK(bar_name.lock);
        if (bar_name.counter == 0)
                bar_name.flag = 0;                                         /* reset flag if first to reach*/
        bar_name.counter++;
        mycount = bar_name.counter;                                        /* mycount is private */
        UNLOCK(bar_name.lock);
        if (mycount == p) {                                                /* last to arrive */
                bar_name.counter = 0;                                      /* reset for next barrier */
                bar_name.flag = 1;                                         /* release waiters */
        }
        else
                while (bar_name.flag == 0) {};                             /* busy wait for release */
}
```

Einfach, funktioniert aber nicht (immer)! Wieso?

Eine zentralisierte Barriere (2)

Nacheinander Eintreten in gleichen *Barrier* funktioniert nicht!

- Prozess darf *Barrier* nicht betreten bevor alle Prozesse vorherige Instanz verlassen haben
- weiteren Zähler verwenden; erhöht aber Latenz und *Contention*

Bedeutung der Flagge ändern: warte bis Flagge den **Wert ändert**

Wert umschalten, wenn alle Prozesse den *Barrier* erreicht haben

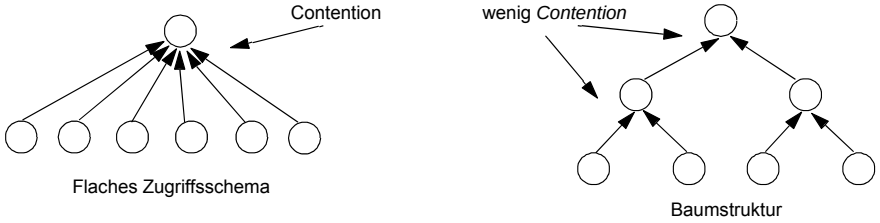
```
/* first call of barrier with local_sense==0; local_sense and mycount are privat */
BARRIER (bar_name, p) {
        local_sense = !(local_sense);                  /* toggle private sense variable */
        LOCK(bar_name.lock);
        mycount = ++bar_name.counter;                  /* mycount = new counter val */
        UNLOCK(bar_name.lock);
        if (mycount == p)                              /* p-1 waiters in while-loop */
                bar_name.counter = 0;
                bar_name.flag = local_sense;           /* release waiters*/
        else
          {     while (bar_name.flag != local_sense) {}; }
}
```

Performance einer zentralisierten Barriere

- Latenzzeit
  - Mindestanzahl Transaktionen bis letzter Proz. Barriere durchschreitet
  - Zentralisierte Barriere mit Anz. Transaktionen max. proportional zu *p*
- Verkehr
  - Barrieren sind oftmals häufig in Verwendung, deshalb sollte Netzwerkverkehr gut skalieren
  - ungefähr *3p* Bus-Transaktionen beim zentralisierten Ansatz
- Speicher
  - sehr wenig: nur globaler Zähler und Flagge notwendig
- Fairness
  - Anforderung: nicht immer gleicher Proz. als letzter aus dem *Barrier*
  - keine Beeinflussung durch zentralisierten Ansatz
- Hauptproblem des zentralisierten *Barrier* ist Latenz und Verkehr

Verbesserungen des Barrier-Algorithmus

combining tree in SW: Nur *k* Prozessoren greifen auf gleiche Speicherzelle zu (*k* ist Grad des Baums)

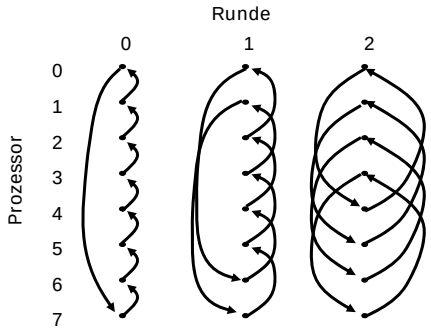


Getrennte Eingangs- u. Ausgangsbäume mit Umkehrung der Funktion

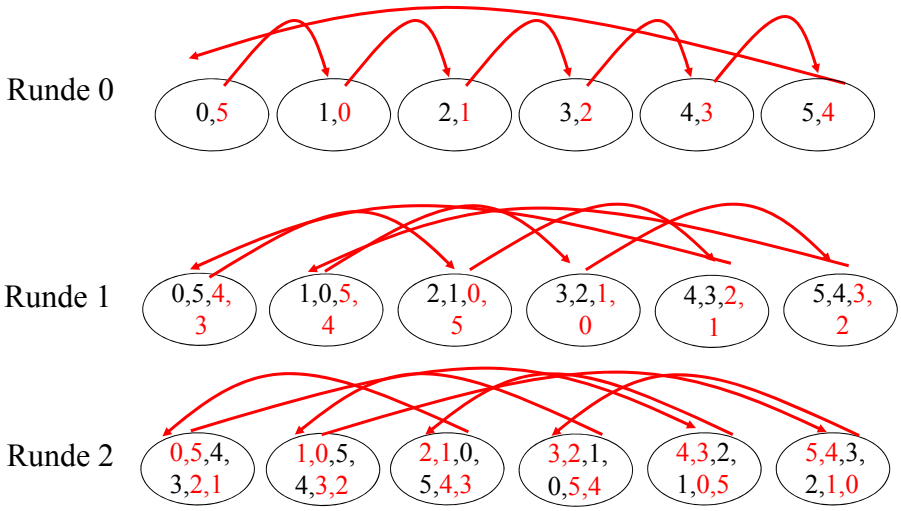
- im Netzwerk
  - Kommunikation über verschiedene Pfade
- beim Bus
  - Gesamter Verkehr geht über den gleichen Bus
  - Höhere Latenz (*log p* Arbeitsschritte, aber *O(p)* serielle Bus-Transaktionen)
  - Software Combining-Tree benötigt ggf. immer noch atomare Operationen

Dissemination Barrier

- ***log p*** Synchronisationsrunden
  In Runde ***k***:
  - Prozessor ***i*** signalisiert Prozessor $(i+2^k) \bmod p$
  - Prozessor ***i*** wartet auf Signal von $(i-2^k) \bmod p$



Dissemination Barrier (Beispiel)



Tournament Barrier

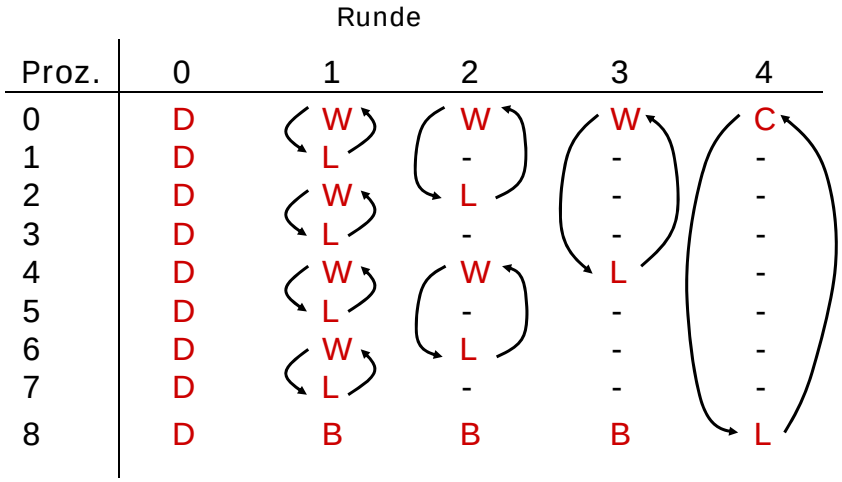
- Binärer Combining-Tree
- Repräsentativer Prozessor eines Knotens wird statisch vorweg bestimmt
  - kein atomares fetch&op benötigt
- In Runde **k**, Prozessor $\mathbf{i = 2^{k-1}}$ setzt eine Flagge für Prozessor $\mathbf{j = i - 2^{k-1}}$
  - i fliegt aus dem Turnier heraus und j macht in der nächsten Runde weiter
  - i wartet auf globales Beenden des Barrier durch Signalisierung einer Flagge
    - kann combining wakeup tree nutzen

Tournament Barrier

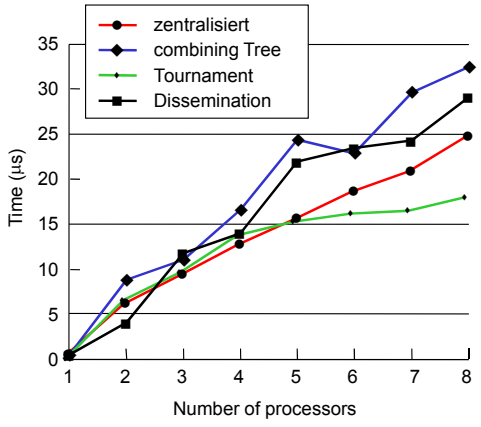
```
type round_t = record role: (winner, loser, bye, champion, dropout);
                      opponent: ^boolean;
                      flag: boolean initially false
shared rounds: array[0..P – 1][0..log P] of round_t
  /* rounds[i] is locally accessible to processor i.  rounds is initialized so that what happens */
  /* in each round is "hard-wired" -- see paper.                                                  */
processor private sense: boolean initially true; vpid: integer /* virtual processor index */
procedure tournament_barrier
  round := 1;
  loop
    case rounds[vpid][round].role of
      loser:
          rounds[vpid][round].opponent^ := sense;
          repeat until rounds[vpid][round].flag = sense;
          exit loop
      winner:
          repeat until rounds[vpid][round].flag = sense
      bye: /* do nothing */
      champion:
          repeat until rounds[vpid][round].flag = sense;
          rounds[vpid][round].opponent^ := sense;
          exit loop
      dropout:  /* impossible */
    round := round + 1;
  loop
    round := round  – 1;
    case rounds[vpid][round].role of
      loser: /* impossible */
      winner:
          rounds[vpid][round].opponent^ := sense;
      bye: /* do nothing */
      champion: /* impossible */
      dropout:
          exit loop
  sense := ¬sense
```

Achtung: Read/write Atomarität.

Tournament Barrier: Roles & Pointers



Barrier Performance auf SGI Challenge



- zentralisiert ist auf Bus-basierten Systemen auch gut
- Hardware-Unterstützung
  - *piggy-backing* von *Read-Misses* auf Bus

Zusammenfassung: Synchronisation

- Gemeinsame Evaluierung von Hardware-Primitiven und Software-Algorithmen
  - Primitive bestimmen die Geschwindigkeit der Implementierung vom Algorithmus
- Methoden zur Evaluierung sind wichtig
  - Parametrisierung von *Microbenchmarks*
  - *Microbenchmarks* und reale Arbeitslasten verwenden
- Einfache Software-Algorithmen mit gängigen Hardware-Primitiven liefern auf Bussen gute Leistung

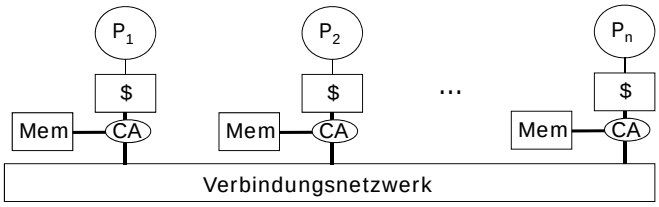
Zusammenfassung: Kapitel SMP-Systeme

- Kohärenz, Sequentielle Konsistenz
  - Definitionen
  - Notwendige / hinreichende Bedingungen
- Bus-basierte SMP
  - *Single-Level, Write-back* Cache
    - Invalidierung-Protokolle
    - Update-Protokoll
  - *Multi-Level-Caches, Split-Transaction-Busse*
- Synchronisation
  - *Locks*
  - *Barrier*

Cache-Kohärenz in skalierbaren Rechnersystemen

Skalierende, Cache-kohärente Systeme (1)

- Skalierender, verteilter Speicher **und** kohärente Replikation.



- System mit verteiltem Speicher
  - P-C-M- (Processor-Cache-Memory) **Knoten** verbunden über ein Kommunikationsnetzwerk.
  - Kommunikationsassistent interpretiert Netzwerktransaktionen, stellt Interface zwischen P-C-M und Netzwerk dar.

Skalierende, Cache-kohärente Systeme (2)

- Weiterhin gemeinsamer Adressraum
  - *Cache-Miss* wird transparent durch lokalen oder entfernten Speicher erfüllen.
- Natürliche Eigenschaft der Caches ist Daten zu replizieren
  - Aber wie wird das Kohärenz-Problem gelöst?
  - Wie geht dieses ohne ein *Broadcast*-Medium für das *Snooping?*
- Nicht nur die Hardware (Latenz u. Bandbreite im Netzwerk), auch das eingesetzte Protokoll muss skalieren.

Was muss das kohärente System leisten?

- Grundlage wiederum Zustände der Speicherblöcke, Zustandsübergänge und Prozessor- / Bus-Transaktionen
- Allgemeine Schritte eines Kohärenz-Protokolls
  (0) Bestimme wann Kohärenz-Protokoll einsetzt
  (a) Einsammeln der notwendigen Informationen über die verteilt vorliegenden Zustände
  - muss überhaupt mit anderen Kopien kommuniziert werden?
  (b) Lokalisierung der Kopien
  - welche Caches haben eine Kopie?
  (c) Kommunikation mit diesen Kopien
  - *Invalidierung bzw. Update* der Cache-Blöcke
- Schritt (0) ist in allen Systemen gleich realisiert
  - Zustand der *Cache-Line* wird im Cache gehalten
  - Protokoll setzt ein, sobald „Zugriffsfehler“ auftritt
- Unterschiedliche Ansätze, je nach Realisierung von (a) ... (c)

Vergleich zu Bus-basierter Kohärenz

- Bus: Schritte (a), (b) und (c) werden über *Broadcast* realisiert
  - Prozessor mit Zugriffsfehler sendet ein „Suchen“
  - Andere Prozessoren antworten und führen ggf. Aktionen durch
- Kann genauso auch in beliebigen Verbindungsnetzen gemacht werden
  - *Broadcast* zu allen Prozessoren und nachfolgendes Antworten
- Konzeptionell einfach, aber *Broadcast* skaliert nicht mit #Proz.
  - Bus: Bandbreite auf einem Bus skaliert nicht
  - Beliebiges Netzwerk: Bei Cache-Miss O(#Proz.) an Netzwerk-Transaktionen notwendig

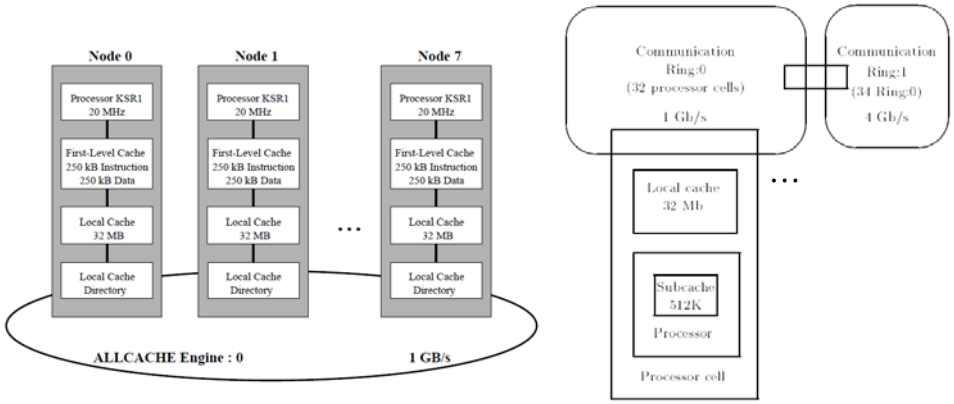
- Skalierendes Kohärenzprotokoll
  - Kann auf bekannte Cache-Zustände und Zustandsübergänge aufsetzen
  - Aber unterschiedliche Mechanismen bei der Ausführung des Protokolls

Methode #1: Hierarchisches Snooping

- Erweiterter *Snooping*-Ansatz: Hierarchie aus *Broadcast*-Medien
  - Baum aus Busse oder Ringen (KSR-1)
  - Prozessoren sind in den Blättern des Baums
  - „Eltern“ und „Kinder“ verbunden über Zweiwege-*Snooping*-Interfaces
    - *Snooping* beider Busse und Propagierung relevanter Transaktionen
  - Hauptspeicher zentralisiert an der Wurzel oder verteilt an Blättern

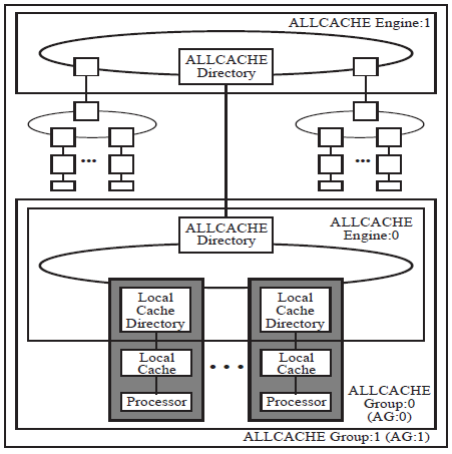
- (a) – (c) ähnlich wie beim Bus, nur kein vollständiger *Broadcast*
  - Prozessor mit Cache-Miss sendet „Suchen“ als Transaktion auf seinen Bus
  - Hierarchie rauf und runter propagieren, je nach *Snooping*-Ergebnis

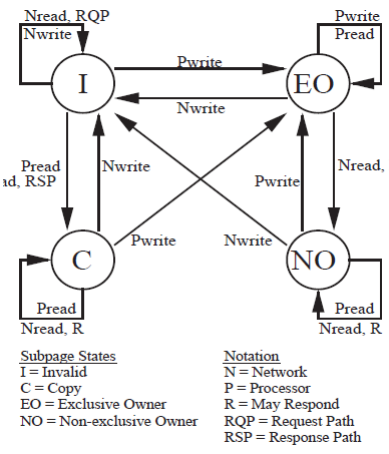
KSR1 Architecture



Level		Potential Size	Latency (cycles)
0	Registers	32 ints + 64 floats	0
1	Subcache	512 Kb	2 (.1 μs)
2	Local Cache	32 Mb	18 (1 μs)
3	Ring:0	1024 Mb	175 (9 μs)
4	Ring:1	34 Gb	600 (30 μs)

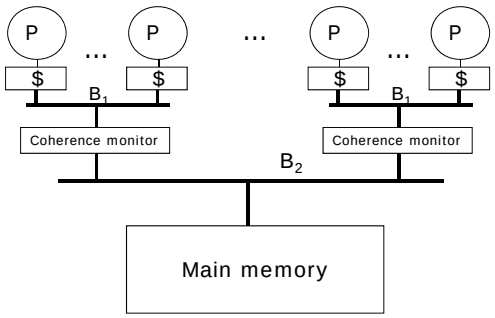
KSR1 Coherence Protocol





KSR1 local cache coherence protocol

Hierarchisches Snooping



- Probleme:
  - Hohe Latenzen: mehrere Ebenen und *Snoop/Lookup* auf jeder Ebene
  - Bandbreite an der Wurzel ist Flaschenhals

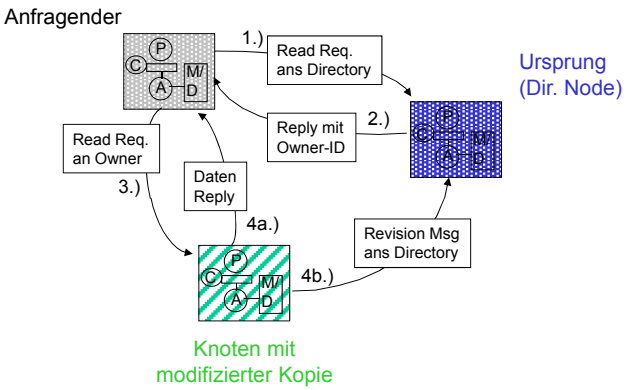
Hierarchisches *Snooping* über mehr als zwei Ebenen wird **nicht** in aktuellen Systemen eingesetzt!

Methode #2: Directories

- Jeder Speicherblock mit assoziierter *Directory*-Information
- *Directory*-Eintrag liefert für den Block eine Übersicht über alle Kopien (Verweise/Ids) und dadurch einen Zugriff auf deren Zustände
  - Bearbeitung eines *Cache-Misses:*
    1. Finde *Directory*-Eintrag
    2. Eintrag auswerten
    3. Kommunikation nur mit den Knoten, die eine Kopie haben
- Skalierendes Netzwerk
  - Kommunikation mit dem *Directory* und den Kopien mit Hilfe von Netzwerk-Transaktionen

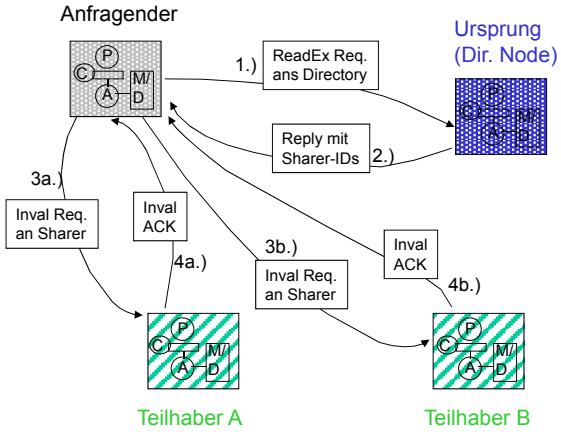
Idee: Read mit Directory

Beispiel: Read-Miss auf Block im „Dirty“ Zustand



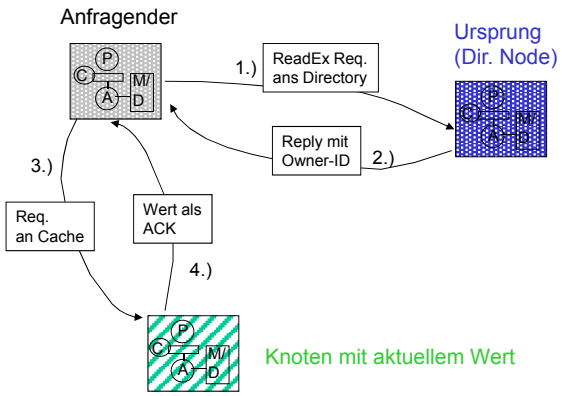
Idee: Write mit Directory (1)

Beispiel: Write auf cached Block mit zwei Teilhabern



Idee: Write mit Directory (2)

Beispiel: Write auf Block mit Modifizierung in einem anderen Cache



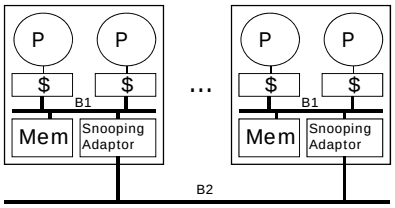
Verschiedenartige Organisationsformen der *Directories* möglich!

Populäre Kompromisse

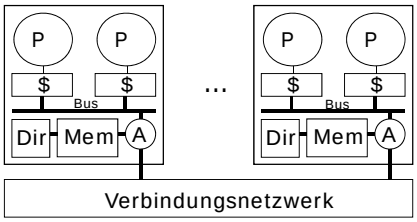
„Zwei-Ebenen-Hierarchie"

- Einzelne Knoten sind Multiprozessorsysteme
  - Z.B. Gitter aus SMPs
- Kohärenz im Knoten über *Snooping* oder *Directory*
  - benötigt dafür ein gutes, funktionales Interface
- Kohärenz zwischen Knoten ist *Directory*-basiert
  - *Directory* liefern Übersicht über Knoten, nicht über den einzelnen Prozessor
- Beispiele:
  - Convex Exemplar: *directory-directory*
  - Sequent, Data General, HAL: *directory-snooping*
  - SGI Origin: *snopping-directory*

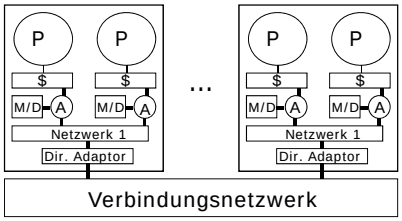
Überblick Zwei-Ebenen-Hierarchien



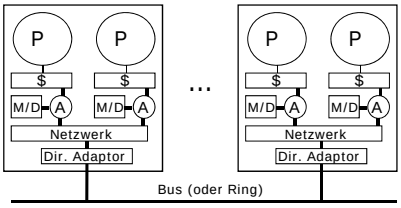
a.) *Snooping-Snooping*



b.) *Snooping-Directory*



c.) *Directory-Directory*



d.) Directory-*Snooping*

Warum Multiprozessorknoten?

- Nutzung des Kostenvorteils
  - Amortisierung der festen Kosten eines Knotens über viele Instanzen
  - Commodity SMP-Designs können eingesetzt werden
  - Weniger Knoten als Prozessoren, deshalb kleinere *Directories*
- Nutzung des Leistungsvorteils
  - Viel, nur lokale Kommunikation innerhalb eines Knotens (weniger globaler Verkehr)
  - gemeinsames *Prefetching* für alle Prozessoren eines Knotens möglich (weniger entfernte *Misses*)
  - Zusammenfügen von Anfragen eines Knotens möglich
  - Kann zu gemeinsamer „Cache“-Nutzung führen (überlappender *Working-Set*)
  - Vorteil hängt vom Speichernutzungsmuster ab und von der Abbildung in den Hauptspeicher (*Mapping*)
    - Gut bei weitgreifendes gemeinsames Lesen
    - Gut bei „nächster Nachbarschaft“ Mustern (bei entsprechendem *Mapping*)

Nachteil kohärenter MP-Knoten

- Prozessoren eines Knotens teilen sich die Bandbreite nach außen
  - insbesondere bei *All-to-All*-Kommunikation schlecht
  - gilt sowohl für kohärente als auch für nicht-kohärente Systeme
- SMP-Bus erhöht Latenzzeit zum lokalen Speicher
- Unter Kohärenz wird typischerweise auf lokales *Snoop*-Ergebnis gewartet bevor entfernte Anfrage heraus geht. Damit auch erhöhte Latenzzeit zum entfernten Speicher.
- *Snoopy*-Bus im entfernten Knoten erhöht nochmals Latenzzeit und verringert Bandbreite

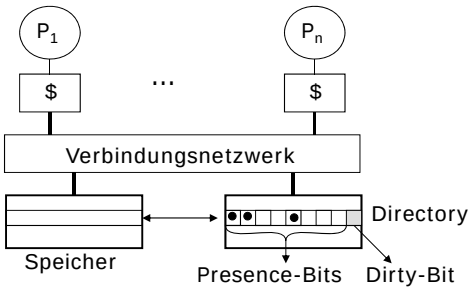
- Die Systemleistung ist durch die Gutartigkeit des Zugriffsmusters auf den gemeinsamen Speicher stark beeinflusst.

Weitere Themen

- Übersicht über *Directory*-basierte Ansätze

- *Directory*-Protokolle
  - Korrektheit (Serialisierung, Konsistenz)
  - Implementierungen
  - Fallstudie: SGI Origin2000
  - Diskussion alternativer Ansätze

Basis-Operationen eines Directories

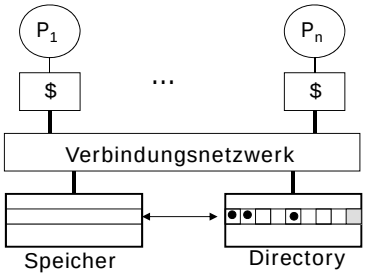


Bei n Prozessoren, für jeden
- *Block* im Speicher:
  n *Presence*-Bits und 1 *Dirty-Bit*
- *Block* im Cache:
  Zustands-Bits des Prozessor/Cache-Protokolls

- Cache-Zustände
  - Invalid wie bekannt
  - Dirty (Modified) zeigt, ob Cache modifizierten Block hält
- Presence-Bit im Directory
  - [ON] => Prozessor hat den Block im Cache
- Dirty-Bit im Directory
  - [ON] => genau ein Prozessor hat den Block im Cache als *modified* markiert
    => nur bei diesem Prozessor ist Presence-Bit [ON]

Basis-Operationen eines Directories

- Prozessor i mit *Read-Miss*:
  - falls *Dirty-Bit* OFF, dann { *Read* im Hauptspeicher; setze p[i] ON; }
  - falls *Dirty-Bit* ON, dann { hole *Cache-Line* vom „*Dirty*“ Proz. (Cache-Zustand in *Shared*); aktualisiere Hauptspeicher mit gelesener Cache-Line; setze *Dirty-Bit* OFF; setze p[i] ON; liefere i angeforderte Daten; }



- Prozessor i mit *Write-Miss*:
  - falls *Dirty-Bit* OFF, dann {Daten in Cache von Proz. i ablegen; lese Precense-Bits; sende **Invalidierung** zu allen Caches j mit p[j] ON; setze *Dirty-Bit* ON; setze p[1..n] OFF; setze p[i] ON; ... }
  - falls *Dirty-Bit* ON, dann sende **Invalidierung+Dirty-Bit-off** zu Cache j mit p[j] ON; setze p[j] OFF; setze p[i] ON

Skalierung über Anzahl an Prozessoren

- Skalierbarkeit der Speicher- und *Directory*-Bandbreite
  - Zentrales *Directory* ist Flaschenhals bzgl. Bandbreite, genauso wie ein zentraler Speicher
  - Aber wie können *Directory*-Informationen verteilt gehalten werden?
- Skalierung der Leistungscharakteristik
  - Verkehr: Anzahl der durch das Protokoll bedingten Netzwerktransaktionen
  - Latenz: Anzahl der Netzwerktransaktionen auf dem kritischen Pfad
- Skalierung der Speicheranforderung des *Directory*
  - Anzahl der Presence-Bits wächst mit Anzahl Prozessoren

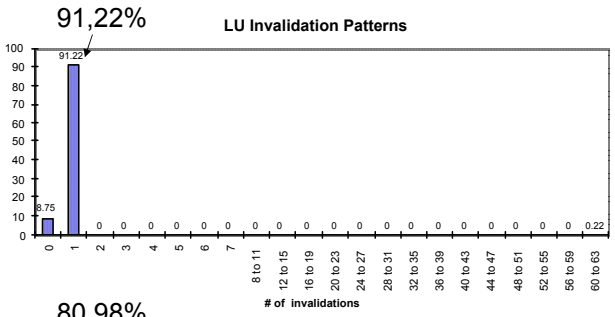
Die Organisation eines *Directory* beeinflusst all diese Eigenschaften.

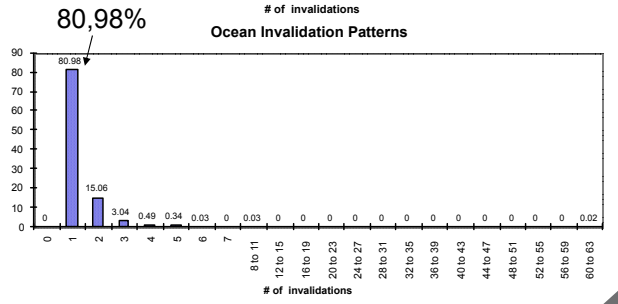
Einblick in Directories

- Programm-Charakteristik bestimmt
  - *ob Directories* größeren Vorteil gegenüber *Broadcast* haben
  - wie *Directories* organisiert und deren Informationen optimal gespeichert werden müssen
- Bedeutende Eigenschaften:
  - Häufigkeit von *Write-Misses*
  - Anzahl an Teilhabern beim *Write-Miss*
  - Skalierung mit Anzahl an Prozessoren
- Deshalb sind Benchmark-Programme zu betrachten.

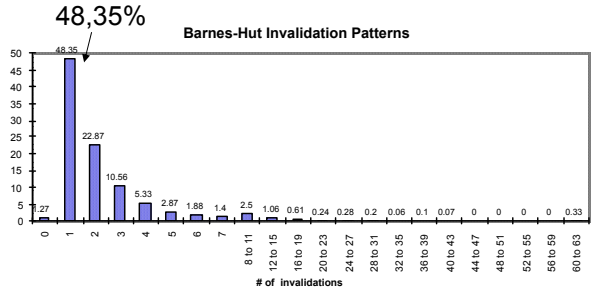
Speichermuster und Cache-Invals (1)

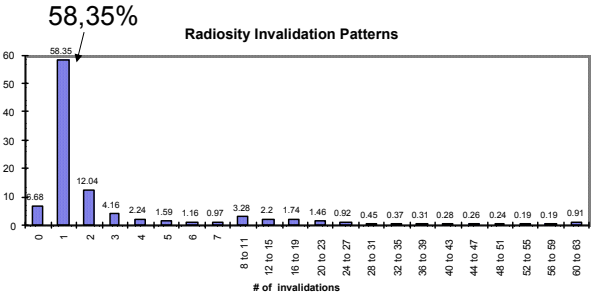
- Beispiel:
- 64 Prozessoren und vier verschiedene Benchmark-Programme
- X-Achse zeigt Prozentzahl an Speicherzugriffe, die genau den angegebenen Anteil an Prozessoren invalidieren





Speichermuster und Cache-Invals (2)

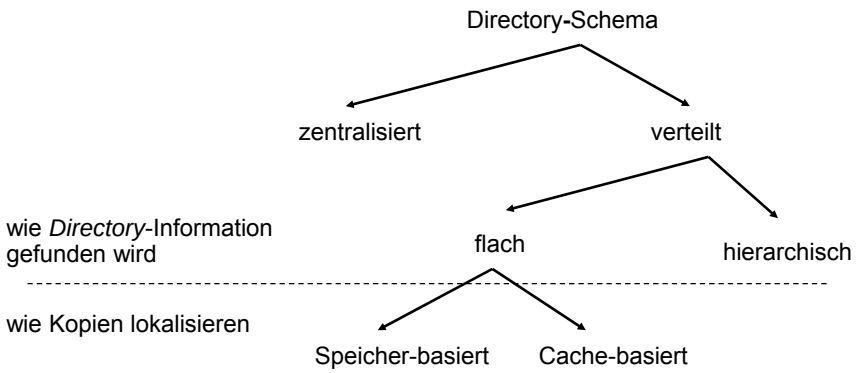




Übersicht: Speichermuster in Shared-Memory-Systemen

- Beobachtung: Beim Write, allgemein nur relativ wenige Teilhaber und deren Anzahl nur langsam zunehmend mit *p*
  - Programmcode und „nur gelesene Objekte" (z.B. Ray-Tracer-Szenen)
    - Keine Probleme, da niemals geschrieben
  - Wandernde Objekte
    - Auch bei zunehmenden *p*, nur 1 bis 2 Invalidierungen
  - Überwiegend nur gelesene Objekte (z.B. Barnes Hut)
    - Invalidierungen betreffen viele Caches, sind aber selten, deshalb nur wenig Einfluss auf Performance
  - Häufig „lese/schreib" Objekte (z.B. *Task-Queues*)
    - Invalidierungen von jeweils wenigen Blöcken, aber häufig
  - Synchronisationsobjekte
    - *Locks* mit wenig *Contention* ⇒ geringer Aufwand für Invalidierung
    - *Locks* mit viel *Contention* ⇒ benötigen spezielle Unterstützung (SW-Bäume, *Queuing-Locks*)
- *Directories* helfen bei Reduzierung des Netzwerkverkehrs
  - Richtig implementiert ⇒ Latenz und Verkehr skalieren recht gut
  - Techniken zur Reduzierung des Speicher-Overheads gefragt

Organisation von Directories



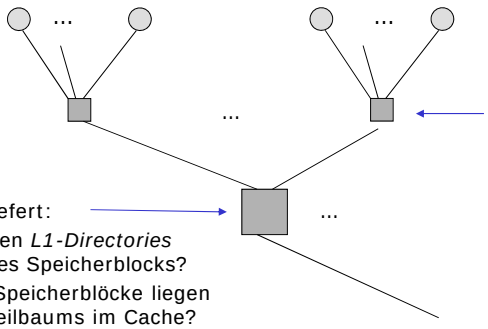
Wie funktionieren diese Verfahren und wie skalieren diese über Anzahl an Prozessoren?

Auffinden der Directory-Information

- Zentralisierter Speicher und *Directory*
  - Alle Speicherinhalte und Directory-Informationen in gleicher Art zugreifbar.
  - Pfad zur zentralen Stelle ist aber ein Engpass.
  - Skaliert somit nicht!

- Verteilter Speicher und verteiltes *Directory*
  - Flaches Schema
    - *Directory* in gleicher Weise verteilt wie der Speicher
      - Directory-Informationen liegen beim Ursprung des Speicherblocks
    - Position hängt von der Adresse ab (hashing)
      - Netzwerk-Transaktion direkt zum Ursprung
  - Hierarchisches Schema
    - Directory ist eine hierarchische Datenstruktur.

Wie funktioniert ein hierarchisches Directory?



Level-1 Directory liefert:
- Welcher nachfolge Proz. hat Kopie des Speicherblocks?
- Welche Speicherblöcke liegen außerhalb des Teilbaums im Cache?
- Inklusion von Prozessor-Caches und *Directory*

Level-2 Directory liefert:
- Welche nachfolgen *L1-Directories* haben Kopie eines Speicherblocks?
- Welche lokalen Speicherblöcke liegen außerhalb des Teilbaums im Cache?
- Inklusion von *L1-Diretories* und *L2-Directory*

Directories bilden hierarchische Datenstruktur
- Blätter sind Rechenknoten, interne Knoten sind nur *Directories*
- Logische Hierarchie, nicht notwendigerweise auch hierarchisches Netzwerk

Auffinden der Directory-Information

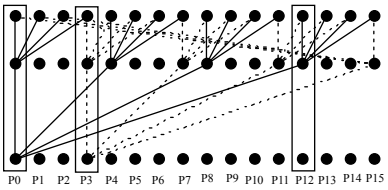
- Verteilter Speicher und *Directory*

  ...

  – Hierarchisches Schema
    - *Directory* als hierarchische Datenstruktur organisiert
    - Blätter sind Prozessorknoten, interne Knoten haben nur *Directory*-Zustände
    - Eintrag im *Directory* eines Knoten besagt, ob einer im Teilbaum den Block im Cache hat
    - Um *Directory*-Information zu bekommen sende „Search“ Nachricht hoch zum Vorgänger
      – routet weiter durch *Directory-Look-Ups*
    - Entspricht hierarchisches *Snooping*, aber ohne Broadcast und nur mit Punkt-zu-Punkt-Nachrichten

Skalierungseigenschaften

- Bandbreite: Wurzel kann Flaschenhals sein
  – Verwenden von *Multi-Rooted-Directories* kann dieses Problem lösen



- Verkehr (# Nachrichten im Netzwerk):
  – Abhängig von Lokalität in der Hierarchie
  – Im schlimmsten Fall *4*log(p)* für eine Kopie
  – Verkehr reduzierbar durch *Message-Combining*
- Latenzzeit:
  – Ebenfalls abhängig von Lokalität in der Hierarchie
  – Proportional zur Entfernung zum jeweiligen Ursprung
  – Kann mehrere Netzwerk-Transaktionen entlang der Hierarchie und *Directory-Look-Ups* entlang des Wegs erfordern
- Speicher-Overhead: (Übungsaufgabe)

Wie werden Kopien lokalisiert?

- Hierarchisches Schema
  - Implizit über die Hierarchie
  - Jedes *Directory* hat *Presence-Bit* seiner Nachfolger (Teilbäume) und *Dirty-Bit*
- Flaches Schema
  - Sehr unterschiedlich mit verschiedenen Speicher-Overheads und Leistungscharakteristika
  - Speicher basiertes Schema
    - Infos über Kopien im Ursprung beim Speicherblock abgelegt
    - Beispiele: SGI Origin, SGI Altix
  - Cache basiertes Schema
    - Infos über weitere Kopien bei der Kopier selber abgelegt
      - Verweise auf nächste Kopien
    - Beispiel: Scalable Coherent Interface (SCI) IEEE Standard

Flache, Speicher basierte Schemen

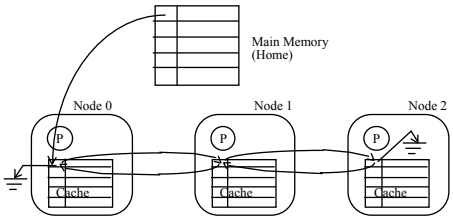
- Alle Informationen über Kopien beim Ursprung des Speicherblocks abgelegt
  - Funktioniert wie zentralisiertes Schema, aber entsprechend verteilt
- Skalierung der Leistungscharakteristik
  - Verkehr beim *Write*: proportional zu # Teilhaber
  - Latenz eines *Writes*: Invals von Teilhabern parallel ausführbar
- Skalierung des Speicher-Overheads
  - Repräsentation: Bit-Vektor, ein *Presence-Bit* je Knoten
  - Speicher-Overhead skaliert schlecht mit *p*; bei z.B. 64 Byte Cache-Lines
    - 64 Knoten: 12.7% Overhead
    - 256 Knoten: 50% Overhead
    - 1024 Knoten: 200% Overhead
  - Bei *m* Speicherblöcken: *O(p*m)* Speicher-Overhead

Reduzierung des Speicher-Overheads

- Optimierung des Schemas mit vollen Bit-Vektoren
  - Erhöhe Größe der Cache-Blöcke (OV proportional zu der Größe)
  - Verwende Multiprozessorknoten (Bit pro Knoten, nicht pro CPU)
  - Weiterhin *O(m*p)*, aber moderat bei typischen Systemen
    - 256 CPUs, 4 CPUs pro Knoten, 128 Byte Cache-Lines ⇒ 6.25% Overhead
- Reduzierung in der „Breite“: betrifft das *p*
  - Beobachtung: meisten Blöcke nur im Cache weniger Knoten
  - Kein P-Bit je Knoten, Eintrag hält Zeiger zu Teilhaber-Knoten
  - *p* = 1024 ⇒ 10 Bit Zeiger entsprechen Platz für max. 102 Verweise
  - Sharing-Muster zeigt, dass meistens wenige Verweise reichen (durchschnittlich max. 5 Verweise)
  - Benötigt Overflow-Strategie bei mehr Teilhabern als Zeigerkapazität
- Reduzierung der „Höhe“: betrifft das *m*
  - Beobachtung: # Speicherblöcke >> # Cache-Blöcke
  - Meisten *Directory*-Einträge sind zu einem Zeitpunkt nutzlos
  - Organisation des *Directories* als ein Art Cache und damit nicht für jeden Speicherblock Einträge vorhalten

Flaches, Cache basiertes Schema

- Wie funktioniert dieses Schema?
  - Ursprung hält nur Zeiger zum Rest der *Directory*-Information
  - Verteilte verkettete List über Kopien, verflochten in den Caches
    - Cache-Tag hat einen Zeiger auf nächsten Cache mit weiterer Kopie
  - Beim *Read-Miss*: Einfügen am Anfang der List (Kommunikation mit erstem Cache in Liste erforderlich)
  - Beim *Write*: Propagierung von *Invals* entlang der Liste



Scalable Coherent Interface (SCI) verwendet doppelt verkettete Liste

Skalierungseigenschaften: Cache-basiert

- Verkehr beim *Write*: Proportional zu der Anzahl an Teilhabern
- Latenz beim *Write*: Ebenfalls proportional zur Anzahl Teilhabern!
  - Liste muss sequentiell abgearbeitet werden
  - Jeder Knoten auf dem Weg muss aktiv sein
  - Auch *Read* erfordert Unterstützung anderer Knoten (Ursprung und erster Teilhaber)
- Speicher-Overhead: Gute Skalierung über beide Achsen (p,m)
  - Nur ein Anfangszeiger pro Speicherblock
    - Rest ist proportional zur Größe der Caches
- weitere Eigenschaften:
  - Gut: ausgereift, IEEE Standard, Fairness
  - Schlecht: Komplexität

Zusammenfassung: Organisation der Directories

Flaches Schema:

- Punkt (a): Finde Quelle der *Directory*-Daten
  - Zum Ursprung gehen, Funktion basiert auf der Speicheradresse
- Punkt (b): Finde die Stellen an denen Kopien sind
  - Speicher-basiert: alle Infos im *Directory-Entry* am Ursprung
  - Cache-basiert: Ursprung hat Zeiger auf erstes Element der verteilten verketteten Liste
- Punkt (c): Kommuniziere mit den Kopien
  - Speicher-basiert: Punkt-zu-Punkt Nachrichten (ggf. gröber bei Überlauf)
    - Kann ein Multicast sein oder überlappend ausgeführt werden
  - Cache-basiert: Teil des Ablaufens der Punkt-zu-Punkt verbundenen Liste
    - Aber reine sequentielle Abarbeitung

Hierarchisches Schema:

- Alle drei Punkte durch herauf und herunter Senden von Nachrichten im Baum
- Keine einzelne explizit an einer Stelle gespeicherte Liste an Teilhabern
- Nur jeweilige Kommunikation zwischen benachbarten Ebenen