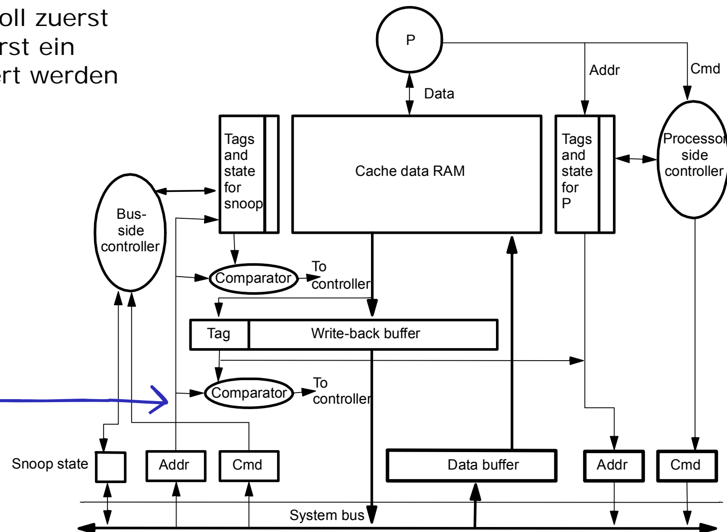
Writebacks

Um den Prozessor ein schnelles fortführen zu ermöglichen, soll zuerst der *Miss* bedient und dann erst ein *Writeback* asynchron realisiert werden

- „*write-back buffer*" notwendig
- Snooping einer Bus-Transaktion erfordert auch *Vergleich* mit dem „WB buffer"



Paderborn Center for Parallel Computing

Nicht-Atomare Zustandsübergänge

Speicheroperation erfordert mehrere Aktionen unterschiedlicher Einheiten (Caches, Bus, ...)

- *Look-Up* auf Cache-*Tags*, Bus-Arbitrierung, Aktionen von anderen Controllern, ...
- auch wenn Bus-Atomarität gewährleistet ist, die gesamte Aktionen muss aber nicht notwendigerweise atomar sein
- *Race-Conditions* können auftreten

Annahme: P_1 und P_2 versuchen auf den jeweils im Cache vorhandenen Block A gleichzeitig zu schreiben

- jeder entscheidet *BusRdX* auszulösen, um S→M zu ermöglichen

Fragen:

- Wie wird die Anfrage für diesen Block A bearbeitet?
  - Z.B.: falls P_2 gewinnt muss P_1 Kopie erst invalidieren und danach eigenen BusRdX auslösen
- Was ist mit nachfolgenden Anfragen nach anderen Blöcken, während auf den Bus gewartet wird?

Paderborn Center for Parallel Computing

Transiente Zustände

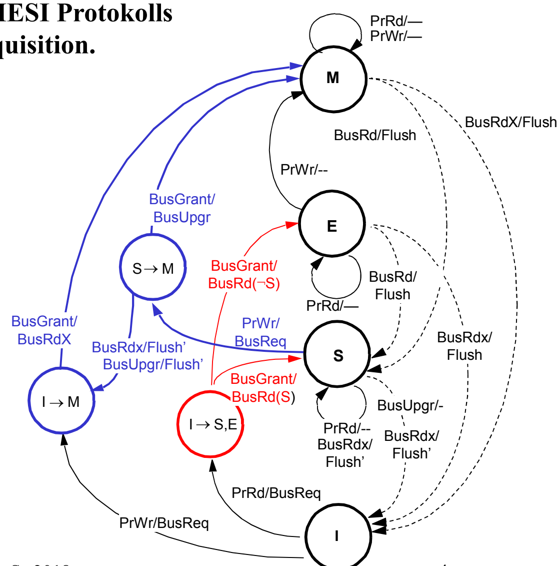
Bus-Akquirierung:

- Aufteilung des Bus-Zugriffs in zwei Phasen
  - *BusReq* und *BusGrant*
- erste Phase (*BusReq*):
  - Bus-Anforderung und Übergang in transienten Zustand
- zweite Phase (*BusGrant*):
  - Beginn mit Bus-Zuteilung
  - konfliktbehaftete Transaktion bewirkt Übergang in weiteren transienten Zustand und erst dann in den stabilen Zustand
- zwischen den beiden Phasen
  - Cache-Controller überwacht weiterhin den Bus und
  - Anfragen bzgl. Blöcke im stabilen Zustand können uneingeschränkt bearbeitet werden

Nicht-Atomarität: Transiente Zustände

Erweitertes Zustandsdiagramm des MESI Protokolls mit transienten Zustände bei Bus-Akquisition.

- **stabile Zustände** (wie in MESI)
- **transiente** (Zwischen-) **Zustände**



Serialisierung?

- *Handshake* zwischen Prozessor und Cache muss Serialisierung der Bus-Reihenfolge bewahren

SC

- *Write-Completion*: Muss nicht warten bis *Inval* wirklich gesetzt wurde
  - Einfach warten bis der Bus zur Verfügung steht
  - *Ansätze: Commit* versus *Complete*
  - Nicht bekannt wann *Inval* in der (lokalen) Reihenfolge des Prozessors eingefügt wird, nur dass dieser vor der nächsten Transaktion und für alle Prozesse in der gleichen Reihenfolge erscheint
  - Nicht wichtig wann die zuschreibenden Daten auf dem Bus erscheinen (insb. write back), nur dass nachfolgendes Lesen diese garantiert sehen können
- *Write*-Atomarität:
  - Wenn ein Lesen den Wert von einem Schreiben W zurück gibt, dann erschien W bereits auf dem Bus und ist damit *complete.*

Speicherhierarchie

- Cache-Hierarchien sind beim Design von Mikroprozessoren seit den 90-er Jahren üblich
- Aktuell sind drei Ebenen bereits Standard
  - kleiner L1-Cache (je Prozessorkern, High-Level)
  - größerer L2-Cache (je Prozessorkern oder shared)
  - großer L3-Cache („Off-Chip" oder „On-Chip", Low-Level)
- In Zukunft noch tiefere Speicherhierarchien notwendig
- Kohärenz und Konsistenz ist auch für Multi-Level-Caches zu gewährleisten

„Multi-Level"-Cache

Wie kann man *Snooping* in Multi-Level-Caches realisieren?
- Unabhängiges *Bus-Snooping* auf jeder Ebene
  - dadurch noch mehr Geräte am Bus!
- Aufrechterhaltung einer Inklusion der Caches
  - aber wie?

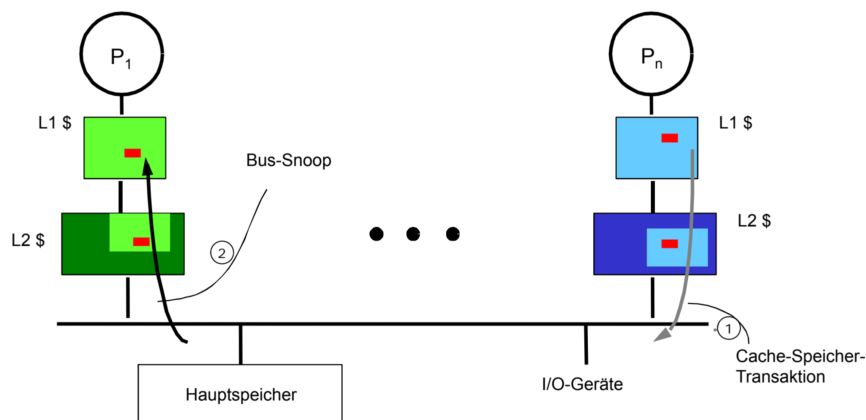
Lösung: Cache-Inklusion

- Anforderungen an Inklusion
  - Daten in höherer Cache-Ebene sind Teilmenge der Daten in niedriger Cache-Ebene
  - Modifizierung in höherer Ebene ⇒ Markierung als modifiziert in unterer Ebene
- damit nur noch ein Bus-*Snooping* auf unterster Cache-Ebene
  - L2 meldet „*Not-Present*" ⇒ auch nicht im L1 vorhanden
  - BusRd auf Block der in L1 modifiziert ⇒ L2 hat Block ebenfalls in diesem Zustand

Paderborn Center for Parallel Computing

Inklusion

Aufrechterhaltung der Inklusion ist nicht trivial!
1. Ersetzung: alle *Misses* höherer Ebenen müssen zu den niedrigen Ebenen gehen
2. Modifikationen durch *Bus-Snooping* müssen zur höheren Ebene durchgereicht werden



Paderborn Center for Parallel Computing

Verletzung der Inklusion

- Die beiden Caches (L1, L2) eines Prozessors können für die Ersetzung unterschiedliche Blöcke auswählen
  - Aufgrund von Unterschiede in der Referenzhistorie
    - beide Controller können unterschiedliche LRU-Ersetzungen durchführen
  - mehrere Blöcke können in den gleichen Set des L1-Caches fallen
    - der L2-Cache kann diese Blöcke ggf. nicht gleichzeitig halten
  - L1 kann Instruktionen & Daten getrennt halten
    - der L2-Cache ist aber ggf. unified organisiert
  - unterschiedliche Blockgrößen
    - Was, wenn nur ein Teil des größeren Blocks verdrängt wird?
- Aber der „Normal-“ Fall geht kanonisch
  - L1 *direct-mapped*, bzw. weniger Sets als der L2-Cache, und die Größe der Blöcke ist identisch

Explizites aufrechterhalten der Inklusion

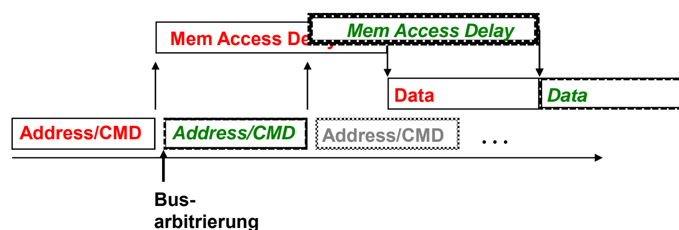
- Propagierung der Bustransaktion vom L2 zum L1
  - Propagierung aller Transaktionen bzw. Filterung der Transaktionen durch Verwendung von *Inclusion-Bits*
- Propagierung der *Lower-Leve*l-Ersetzungen zum *High-Level*-Cache
  - *Invalid vom L2-Cache zum L1-Cache*
  - ggf. *Flush* in Speicher
- Propagierung des „*modified*“ Status vom L1 zum L2 bei „*Writes*“?
  - *Write-Through* L1 oder *Modified-but-Stale* Bit pro Block im L2-Cache

Inklusion: Korrektheit

- Ist nun die Fragestellung bzgl. Korrektheit anders?
  - Nicht wirklich
    - falls alle Propagierungen korrekt ablaufen und auf Abschluss gewartet wird
- Aber Problem bei der Performance, deshalb sollte man nicht auf Abschluss der Propagierung warten müssen
  - *Writes* werden, sobald sie auf dem Bus erscheinen, bestätigt
- Doppelte *Cache-Tags* im L1-Cache verlieren an Wichtigkeit
  - jede Cache-Ebene filtert bereits nicht relevante Transaktionen heraus

„Split-Transaction" Bus

- Aufteilung einer Bus-Transaktion in Request und Response Subttransaktionen
  - separate Arbitrierung beider Phasen
- Andere Transaktionen können diese Abfolge unterbrechen
  - Verbesserung des Durchsatzes (Bandbreite)
- Response muss entsprechendem Request zugeordnet werden
  - Speicherung zwischen Bus und Cache-Controller
  - damit mehrere ausstehende Transaktionen an einem Controller
- Reduziert Serialisierung auf die eigentliche Bus-Arbitrierung



Komplikationen

- Neuer *Request* kann bereits auf dem Bus erscheinen bevor vorherige bedient wurden
  - Ebenso bevor *Snoop*-Ergebnis vom Cache-Controller vorliegt
  - Konfliktbehaftete Operation auf einem Block ist vielleicht noch auf dem Bus ausstehend
  - Z.B. P_1 und P_2 schreiben gleichzeitig auf Block im Zustand S
    - beide bekommen Bus bevor einer „*Snoop*" Ergebnis erhält und beide glauben Sie hätten gewonnen
- Buffer sind klein, deshalb wird eine Flusskontrolle benötigt
- Buffer erfordert neue Betrachtung wann und wie *Snoop*-Ergebnisse geliefert werden
  - *In Order* bzgl. *Requests*?
  - *Snoop*- und Daten-*Response* zusammen oder separat?

Paderborn Center for Parallel Computing

Beispiel: SGI Challenge

- keine konfliktbehaftete *Requests* für gleiche Blöcke auf dem Bus erlaubt
  - Flusskontrolle durch *Negative Acknowledgement (NACK)*
    - *NACK*, sobald ein konfliktbehafteter *Request* auf dem Bus erscheint, danach Anfrage wiederholt stellen
    - Separates Kommando (inkl. *NACK*) + Adresse und Tag + Datenbus
  - Höchstens 8 ausstehende *Requests*, macht Konflikterkennung schneller
- *Responses* können in unterschiedlicher Reichenfolge als *Requests* erscheinen
  - Reihenfolge der Transaktionen durch *Requests* bestimmt
  - *Snoop*-Ergebnisse zusammen mit *Response* auf den Bus

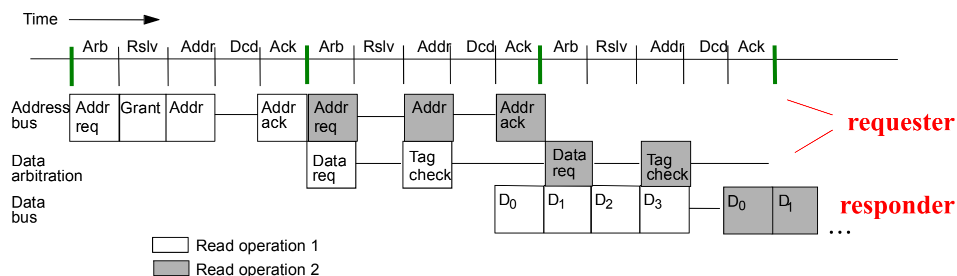
Paderborn Center for Parallel Computing

Beispiel: SGI Challenge (2)

- Notwendigerweise zwei separate Busse, unabhängig arbitriert
  - *Request*-Bus für Kommando und Adresse
  - *Response*-Bus für Daten
- *Out-of-order Responses* erfordern die Zusammenführung passender *Req-Resp*-Paare
  - *Request* bekommt 3-bit Tag sobald Arbitrierung gewonnen wurde (damit max. 8 ausstehende)
  - *Response* enthält Daten genauso wie entsprechenden *Request Tag*
  - *Tags* ermöglicht Verzicht auf den Adressbus beim „*Response*“
- Separate Busleitungen für Arbitrierung und für *Snoop*-Ergebnisse

Beispiel: SGI Challenge (3)

- Jede „Request“ und „Response“ Phase ist 5 Buszyklen lang (best case)
  - „Request“ Phase: **arbitration, resolution, address, decode, ack**
  - „Response“: 4 Zyklen für Daten (128 Bytes, 256-Bit Bus), 1 Umlauf
  - „Request-Response“ Transaktion benötigt mindestens 3 davon:
    - „Address Request“ Phase (nutzt Adressbus)
    - „Data Request“ Phase (nutzt Datenbus-Arbitrierungslogik und Zugriff zum Datenbus für „Response“ Transaktion)
    - Datentransfer (nutzt Datenbus)



Beispiel: SGI Challenge (4)

Auswertung der „*Cache Tags*" in Dekodierphase, falls notwendig wird *Ack*-Zyklus erweitert

- Bestimmung wer antworten soll
- Aktueller „*Response*" kommt später, mit Re-Arbitrierung

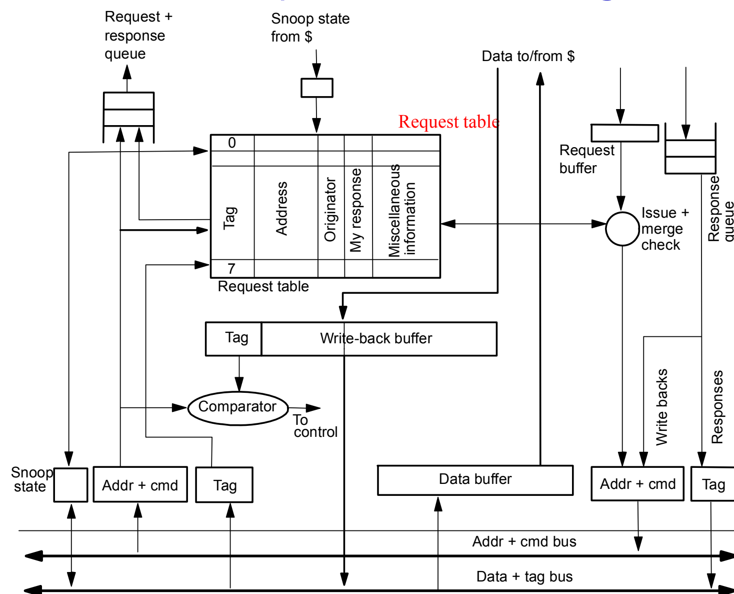
„*Write-backs*" bestehen nur aus einem „*Request*"

- arbitriert Daten- und Adressbus

Beispiel: SGI Challenge (5)

- Jeder Cache hat eine *Request*-Tabelle (Buffer) für 8 Einträge
  - Eintrag hält Adresse, *Request* Typ und Status im Cache (falls bereits bestimmt), …
  - Voll assoziativ
  - Neuer *Request* auf dem Bus wird überall mit gleichem Index (bestimmt durch den 3-Bit *Tag*) geführt
  - Eintrag wird gelöscht, wenn ein *Response* eintrifft, so dass der Tag auf dem Bus wiederverwendet werden kann

Beispiel: SGI Challenge (6)



Bus-Interface mit Request-Tabelle

Paderborn Center for Parallel Computing

Zusammenfassung: Design und Implementierung

- Transiente-Zustände im Kohärenz-Protokoll
  - Entkopplung von Zustandswechseln und Transaktionen
  - Keine Blockierung des Automaten bei ausstehenden Bus-Transaktionen
- Multilevel-Caches
  - Schnellere Zugriffe und höhere Kapazitäten
  - Aufrechterhaltung der Inklusion erforderlich
- Split-Transaction Busse
  - Erhöhung des Durchsatzes auf dem Bus
  - Behandlung ausstehender Requests notwendig

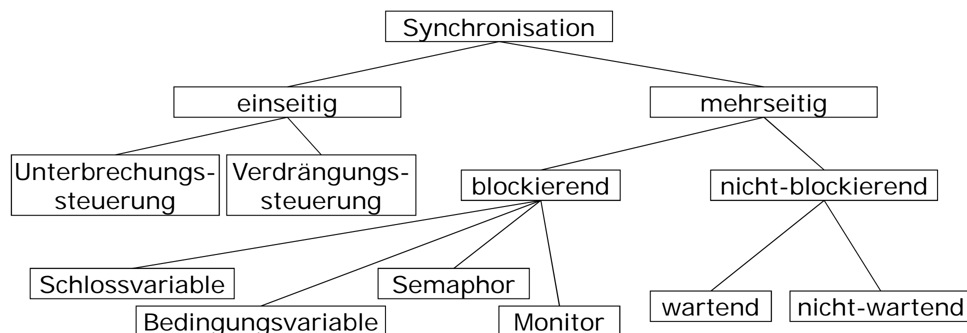
Paderborn Center for Parallel Computing

Synchronisation

"A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast."

Paderborn Center for Parallel Computing

Klassifikation Synchronisation



einseitig: Auswirkung auf **einen** Prozess

mehrseitig: Auswirkung auf **mehrere** Prozesse

Paderborn Center for Parallel Computing

Synchronisationstypen

Synchronisation in einem Multiprozessorsystem durch

- Mutual-Exclusion
- Event-Synchronisation
  - Punkt-zu-Punkt
  - in einer Teilgruppe
  - global (Barriere)

Literatur:

- E.W. Dijkstra: Solution of a Problem in Concurrent Program Control, CACM 8(9), 1965
- D.E. Knuth: Additional Comments on a Problem in Concurrent Program Control, CACM 9(5), 1966

Rückblick und Vorschau

- Welche Hardware-Primitive werden benötigt? Abhängig von Technologie und Maschinentyp
  - Geschwindigkeit vs. Flexibilität
- Die meisten modernen Methoden benutzen eine Art atomares Read-Modify-Write
  - IBM 370: atomares *Compare&Swap*
  - Intel x86: Instruktionen können mit *Lock-Modifier* versehen werden
  - SPARC: atomare Register-Speicher-Operationen
    - *Swap*, *Compare&Swap*
  - MIPS, IBM Power: keine atomaren Operationen aber gepaarte Instruktionen
    - *Load-Locked und Store-Conditional*

Im weiteren Fokus: Synchronisation in Bus-basierten Cache-kohärenten Multiprozessorsystemen

Komponenten einer Synchronisation

1. *Acquire-Method*
   – Erwerben des Rechts zum Synch (betrete kritische Sektion, gehe zu eine Schranke, ...)
2. *Waiting-Algorithm*
   – Warte bis Synch eintritt
3. *Release-Method*
   – Gibt anderen Prozessoren Recht das Synch zu erwerben

Waiting-Algorithm ist unabhängig vom Typ der Synchronisation

Waiting-Algorithms

- *Blocking*
  – Wartende Prozesse werden *„de-scheduled"*
  – Nachteil: großer *Overhead*
  – Vorteil: Prozessor kann ungestört andere Prozesse bearbeiten
- *Busy-Waiting*
  – Wartende Prozesse testen wiederholt eine Speicherzelle, bis diese ihren Wert ändert
  – Freigabe des Prozesses setzt Speicherzelle
  – Vorteil: geringer zeitlicher Overhead
  – Nachteil: verbraucht dabei Prozessorressourcen (Prozessorzyklen)
  kann Netzwerk-Verkehr hervorrufen (Kohärenz-Protokoll!)
- Hybride Methode: eine bestimmte Zeit *Busy-Wait*, dann *blockieren*

Rolle des Systems und Anforderungen der Nutzer

- Nutzer will „High-Level" Synchronisationsoperationen verwenden
  - Locks, Barriers, ...
- System-Designer: Möglichst wenig Unterstützung in Hardware
  - Geschwindigkeit versus Kosten und Flexibilität
  - *Waiting*-Algorithmus schwierig in Hardware, deshalb andere Methoden zu unterstützen
- allgemeiner Trend:
  - System liefert einfache Hardware-Primitive (atomare Operationen)
  - Software-Bibliothek implementiert Lock-, Barrier-Algorithmen, die HW-Primitive verwenden
  - Aber einige Systeme haben auch spezielle Hardware für ausgewählte Synchronisationen

Herausforderungen

- Gleiche Synchronisation kann verschiede Anforderungen zu unterschiedlichen Zeiten haben
  - *Lock* Zugriff mit wenig oder viel Kollisionen (Contention)
  - Unterschiedliche Performance Anforderungen: geringe Latenzzeit oder hoher Durchsatz
- *Multiprogramming* kann Synchronisationsverhalten und Anforderungen ändern
  - Prozess-Scheduling und andere Ressource-Interaktionen
- Großer Bereich von SW/HW-Interaktion
  - Welches Primitiv erfordert welche Algorithmen
  - Welcher Algorithmus erfordert welche Primitive
- Muss mit entsprechenden anwendungsnahen *Workloads* evaluiert werden

Mutual-Exclusion: Hardware Locks

- Mehrere Lock-Lines auf dem Bus
  - Prioritäten gesteuerter Mechanismus bei gleichzeitiger Anforderung
- Lock-Register (Cray XMP)
  - von mehreren Prozessoren gemeinsam nutzbares spezielles Register
- Unflexibel, deshalb eingeschränkte allgemeine Anwendbarkeit
  - wenige *Locks* zur gleichen Zeit nutzbar (einer per *Lock-Line*)
  - *fest verdrahteter Waiting-Algorithmus*

Hardware-Locks hauptsächlich genutzt um Atomarität von SW-Locks auf höherer Ebene zu ermöglichen

Erste Annäherung an einfaches SW-Lock

```
/* lock variable == 0  ⇔  unlocked */

lock:     ld    register, location    /* copy location to register */
          cmp   register, #0          /* compare with 0 */
          bnz   lock                  /* if not 0, try again */
          st    location, #1          /* store 1 to mark it locked */
          ret                         /* return control to caller */

unlock:   st    location, #0          /* write 0 to location */
          ret                         /* return control to caller */
```

Problem: ***Lock*** **setzt Atomarität der eigenen Implementierung voraus**

Sequenz aus Read (test) und Write (set) der Lock-Variable ist aber nicht atomar

Lösung: **atomare *Read-Modify-Write* oder *Exchange*-Instruktionen**

Atomare Exchange-Instruktion

- Spezifiziert eine Speicherzelle und ein Register. Als atomare Operation wird ausgeführt:
  - Zuerst: Auslesen des Werts der Speicherzelle in das Register
  - Anschließend: Anderen Wert (z.B. Funktion des gelesenen Werts) in die Speicherzelle ablegen
- Mehrere Varianten mit unterschiedlicher Flexibilität in der zweiten Hälfe der Operation möglich

- Einfaches Beispiel: test&set
  - Wert der Speicherzelle wird in ein spezielles Register gelesen
  - Konstante 1 wird in die Speicherzelle geschrieben
  - Erfolgreich, falls Wert des speziellen Registers 0 ist

Einfaches Test&Set-Lock

```
/* lock variable == 0  ⇔  unlocked */

lock:    t&s  register, location     /* test location and always set to 1 */
         bnz  lock                   /* if not 0, try again */
         ret                         /* return control to caller */
unlock:  st   location, #0           /* write 0 to location */
         ret                         /* return control to caller */
```

Andere Read-Modify-Write Primitive können ebenfalls genutzt werden

- Swap
- Fetch&Op
- Compare&Swap

  3 Operanden: Speicherzelle, Vergleichs-Register, Austausch-Register

Implementierung Test&Set-Lock

- Variable kann cacheable oder uncacheable sein
  - Cached-Variable nutzt Lokalität
  - Nicht-cached-Variable erspart Invalidierung/Update (aber nicht schnell)

Cacheable Variable:

- Realisierung durch zwei Bus-Transaktionen
  - Blockierung des Busses bei Bus-Transaktion *Read* bis zugehörige Bustransaktion Write abgeschlossen wurde
  - Dabei kein Zugriff von anderen Prozessoren auf den Bus möglich
  - Insbesondere kein Zugriff auf die Lock-Variable möglich
  - Split-Transaktion-Bus erfordert komplexeres Locking
- Realisierung durch *exclusive-ownership*
  - Nutzen des Invalidierung-Protokolls der Write-Back-Caches
  - Locking durch Festhalten des Zustands exclusive-ownership

Paderborn Center for Parallel Computing

Verbesserung des einfachen Lock-Algorithmuses

1. Reduziere Häufigkeit der Aufrufe von *Test&Set* beim Warten
   - *Test&Set Lock* mit ***Back-Off*** (Verzögerung zwischen zwei T&S)
   - *Back-Off* geschickt anpassen
     - Exponentielles *Back-Off* ist ziemlich gut: i^{te}Zeit = $r*s^i$
2. *Busy-Wait* mit normalen Lese-Ops besser als *Test&Set*
   - *Test und Test&Set Lock*
   - Zuerst Testen mit normalem *Read*
     - *Lock*-Variable im Cache wird invalidiert, sobald Freigabe auftritt
     - Falls Änderung des Werts (zu 0), versuche mit *Test&Set* den *Lock* zu bekommen
     - Nur ein Zugreifender hat Erfolg; andere schlagen fehl und starten das Testen von Neuem

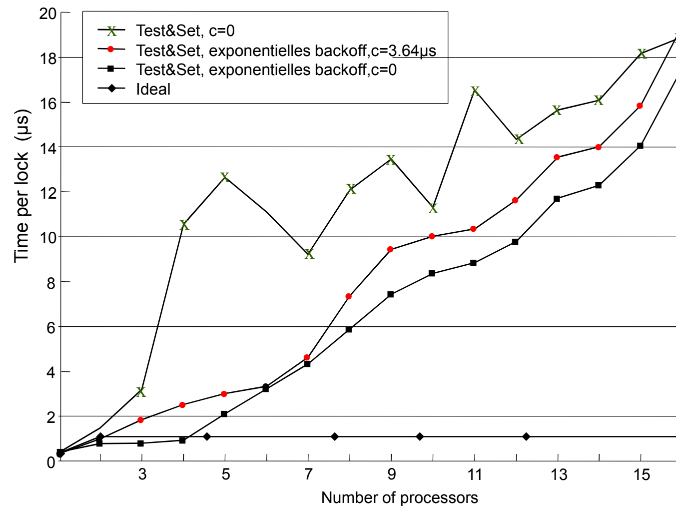
Paderborn Center for Parallel Computing

T&S-Lock Microbenchmark-Performance

Beispiel:
SGI Challenge

Scheife:
  lock;
    krit. Sektion mit
    Delay(c);
  unlock;

feste Anzahl an Lock-Aufrufe pro Prozessor;
gemessene Zeit pro Aufruf (ohne Zeit für kritischen Bereich)



Paderborn Center for Parallel Computing

Performance von T&S-Lock

- Latenzzeit (ohne Kollisionen):
  - Sehr gering, wenn häufig vom gleichen Prozessor genutzt; unabhängig von *p* (Anzahl Prozessoren)
- Verkehr im Netzwerk (Transaktionen auf dem Bus)
  - Viel, wenn mehrere Prozessoren konkurrieren; schlechte Skalierung über *p*
  - Jedes *t&s* erzeugt Invalidierung, und alle laufen los zum nächsten t&s
- Speicherbedarf
  - Sehr gering (einzelne Variable); unabhängig von *p*
- Fairness
  - Schlecht, kann zum „Verhungern" führen (*Starvation*)

- *Test&Set* mit *Back-Off* ähnlich, aber weniger Verkehr
- *Test-und-Test&Set*
  - Etwas höhere Latenz, viel weniger Verkehr
  - $O(p^2)$ Verkehr bei *p* Prozessoren mit jeweils einem *Lock*

Paderborn Center for Parallel Computing

Verbesserte Hardware-Primitive: LL-SC

- Ziel:
  - Testen mit normalen *Reads*
  - Fehlschlagende *Operation* ohne Invalidierung
  - durch ein Primitiv möglichst verschiedene R-M-W Operationen realisierbar
- *Load-Locked (LL)*, *Store-Conditional (SC)*
  - LL liest Variable in Register,
  - gefolgt von einer Instruktion zur Manipulation des Werts, und
  - SC versucht in Speicherzelle zurückzuschreiben. Genau dann erfolgreich, wenn kein anderer die Variable des LL schon beschrieben hat
  - SC erfolgreich bedeutet: alle 3 Schritte wurden atomar ausgeführt
  - Nicht erfolgreich: keine Invalidierung durch den Prozessor (nachfolgend neuer *LL*-Versuch), zurückrollen aller zwischen LL/SC ausgeführten Instruktionen
  - bedingter Sprung realisiert Erfolg/nicht Erfolg von SC

Einfaches Lock mit LL-SC

```
lock:    ll     reg1, location      /* LL – load location to reg1 */
         bnz    reg1, lock          /* lock set? */
                                    /* e.g. reg2=1 */
         sc     location, reg2      /* SC – store reg2 into location */
         beqz   reg2, lock          /* if failed, start again */
         ret
unlock:  st     location, #0        /* write 0 to location */
         ret
```

- Zwischen *LL* & *SC* auch komplexe „atomare Operation" möglich
  - Sollten aber wenige Ops sein, damit erfolgreiches SC wahrscheinlich wird
  - Keine Instruktionen einfügen die ggf. zurückgenommen werden müssen (z.B. *Write*)
- SC kann (ohne eigene Bus-Transaktion) fehlschlagen
  - Erkennung eines zwischenzeitigen *Writes* auf *Lock*-Variable
  - Eigene Bus-Arbitrierung schlägt fehlt, da anderer Prozessor SC auf Bus legt

Beobachtungen

- LL alleine ist kein *Lock* **und**
  SC ist natürlich auch kein *Unlock*
  - Nur zusammen ergeben sie eine Lock-Operation
- Allein die Umschließung mit LL-SC ist keine Garantie dafür, dass die Operation atomar ist
  ⇒ keine kritische Region
- LL-SC ist keine faire Methode

Implementierung LL-SC

Mögliche Realisierung:

- *Lock-Flag* und *Lock-Address*-Register im bus-seitigen Cache-Controller
- Aktionen von LL
  - Variable/Block einlesen
  - *Lock-Flag* setzen
  - Adresse der Variable in *Lock-Address*–Register ablegen
- Bus-seitige Invalidierungen werden gegen Inhalt des *Lock-Address*-Registers getestet
  - Übereinstimmende Adresse: zurücksetzen des Lock-Flags
- SC ist erfolgreich, falls *Lock-Flag* noch gesetzt

Effizientere SW Locking-Algorithmen

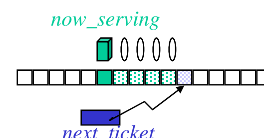
- Problem bei einfachem LL-SC *Lock*
  - Keine Invals bei eigenem Verfehlen, aber Read-Misses bei allen Wartenden sowohl beim **Release** als bei **erfolgreichem SC** des Gewinners
  - *Back-Off* zur Reduzierung von *Bursts* nutzbar
  - Reduziert Verkehr nicht bis zum Minimum; ist auch kein fairer *Lock*
- Besserer SW Algorithmus für den Bus

  (1) Nur ein Prozess versucht nächstes *Lock* zu bekommen

  (2) Nur ein Prozess hat Leserechte (bis zum Release)

  - *Ticket-Lock* erfüllt (1)
  - *Array-basiertes Queueing-Lock* erfüllt (1) und (2)
  - Beide sind gleich faire (FIFO) Locks

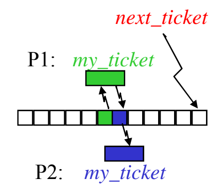
Ticket-Lock



- Nur ein R-M-W pro Anforderung (je Prozess)
- Entspricht Warteschlange bei einer Bank
  - Zwei Zähler pro *Lock* (*next_ticket, now_serving*)
  - Acquire: fetch&inc *next_ticket;* warte bis *now_serving = next_ticket*
    - atomare Op nur beim Lock-Eingang; weniger Contention
  - Release: increment *now-serving*
  - FIFO-Reihenfolge
    - falls *fetch&inc* cacheable, geringe Latenz und wenig Contention
  - Weiterhin *O(p) Read-Misses* beim Release, da alle gleiche Variable lesen (*spinning* auf *now-serving*)
    - wie einfaches *LL-SC Lock*, aber kein Inval wenn SC erfolgreich
    - ist aber fair (FIFO)
  - Optimaler *Delay* für *Back-Off* nur schwer zu finden
    - Lösungsstrategie notwendig, falls *now-serving* nicht von allen beobachtet wird

Polling auf verschiedene Speicherzellen wäre gut ...

Array-basiertes Queuing-Lock



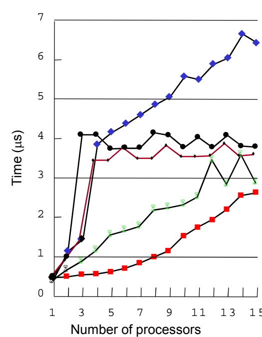
- *Polling* wartender Prozesse auf unterschiedliche Speicherzellen in einem Array der Größe *p*
  - Acquire
    - fetch&inc um Adresse zu bekommen auf der das *Spin* gemacht werden soll (nächstes Array-Element)
    - Achtung: Sicherstellen, dass die Adressen in unterschiedlichen Cache-Lines liegen
  - Release
    - Setze nächste Speicherzelle im Array, damit aufwecken des darauf lesenden Prozesses
  - *O(1)* Verkehr pro Anforderung bei kohärenten Caches
  - FIFO-Reihenfolge, wie beim *Ticket-Lock*
  - Aber, *O(p)* Speicherplatz pro *Lock*-Variable
  - Gute Performance auf Bus-basierten Maschinen
  - Nicht so gut bei Maschinen mit nicht kohärenten Caches und verteiltem Speicher

Lock-Performance der SGI Challenge

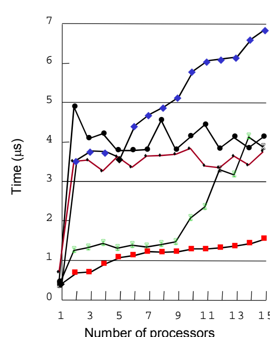
Schleife:

lock;
delay(c);
unlock;
delay(d);

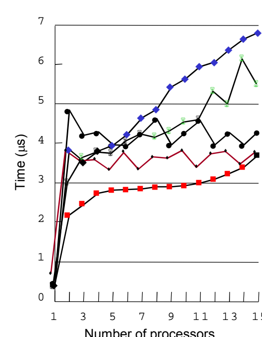
Array-basiert
LL-SC
LL-SC, exponential
Ticket
Ticket, proportional backoff



(a) *c* = 0, *d* = 0



(b) c = 3.64 µs, d = 0



(c) c = 3.64 µs, d = 1.29 µs