Directory-Ansatz bietet

- *Directories* ermöglichen skalierbare Kohärenz auf allgemeinen Netzwerken
  - kein für *Broadcast* optimiertes Medium erforderlich
- viele Möglichkeiten der Organisation von *Directories* und Umsetzung von Protokollen
- Hierarchische *Directories* werden selten eingesetzt
  - Hohe Latenzzeit, viele Netzwerk-Transaktionen und Engpass bei der Wurzel
- sowohl Speicher- als auch Cache-basierte Schemen im Einsatz
  - Speicher-basiert mit vollen Bit-Vektoren reicht für moderate Skalierung
  - Untersuchung einer Fallstudie folgt

Fragestellungen zu Directory-Protokollen

- Korrektheit
- Performance
- Komplexität und Handhabung von Fehlern

Nachfolgend:

- Diskussion der wichtigsten Korrektheits- und Performance-Fragen die ein Protokoll erfüllen muss.
- Betrachtung von Speicher- und Cache-basierten Protokollen und *Trade-Offs* die ggf. gelöst werden müssen.
- Die Komplexität wird während dieser Diskussionen deutlich.

Korrektheit

- Sicherstellung der Kohärenzgrundlagen auf Ebene der Zustandsübergänge
  - *Cache-Lines* erfahren Aktionen *Updated/Invalidated/Fetched*
  - Korrekte Zustandsübergänge und Aktionen durchführen
- Einhaltung der Ordnungs- und Serialisierungsbedingungen
  - für Kohärenz (einzelne Speicherzelle)
  - für Konsistenz (Nutzung mehrerer Speicherzellen)
- Vermeidung von *Deadlocks, Livelocks* und *Starvations*

- Probleme:
  - Mehrere Kopien und mehrere Pfade durchs Netzwerk
    - Nicht so wie beim Bus (nur einen Pfad) oder nicht-kohärenten Caches (keine Kopien)
  - Große Latenzzeit der Kommunikation bietet Optimierungspotential
    - Erhöht aber Gleichzeitigkeit, verkompliziert eine Korrektheitsbetrachtung

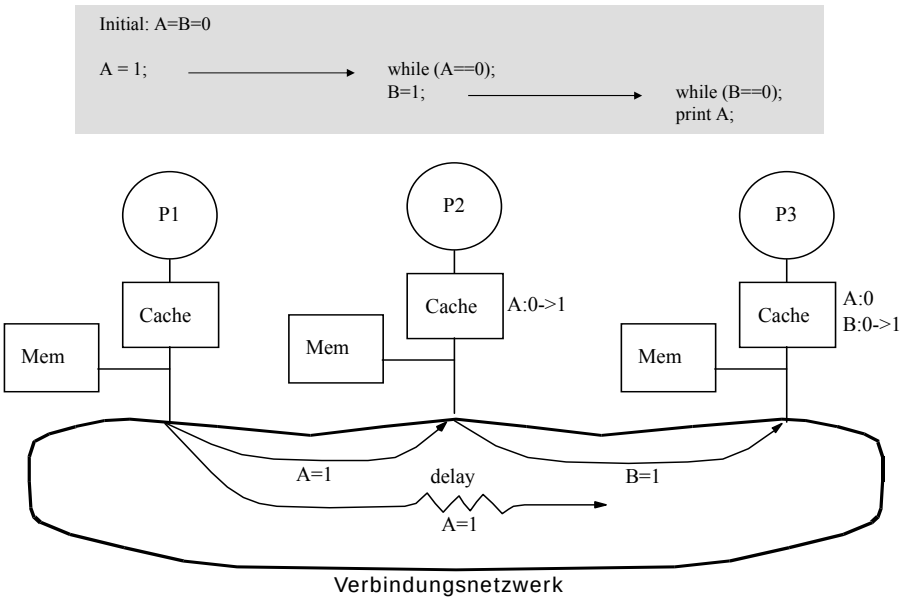
Kohärenz: Serialisierung der Speicherzugriffe

- Bus-basiert
  - mehrere Kopien einer Speicherzelle in den Caches
  - aber Bus realisiert Serialisierung
- Nicht-kohärenter verteilter Speicher (keine Kopien)
  - Speicherinterface des jeweiligen Hauptspeichermodule bestimmt Ordnung der Zugriffe
- Kohärenter verteilter Speicher (Kopien möglich)
  - Speicherinterface der Hauptspeichermodule kann auch hier Serialisierung realisieren
  - valide Kopie muss aber nicht im Hauptspeicher sein
  - Ordnung der Hauptspeicherzugriffe muss nicht Ordnung der entsprechenden Zugriffe auf die Kopien bedeuten
    - Serialisierung bei der Speicherzelle im Hauptspeicher muss nicht Serialisierung der Zugriffe auf Kopien implizieren

Sequentielle Konsistenz

- Bus-basiert
  - *Write-Completion*: Warten bis *Write* auf den Bus gelegt wird
  - *Write*-Atomarität: Liefert der Bus und die Buffer-Ordnung
- Verteilter Speicher ohne Kopien
  - *Write-Completion*: Muss explizit auf ACK vom Speicher warten
  - *Write*-Atomarität: Einfach realisierbar, da immer nur eine „Kopie“
- Verteilter Speicher mit mehreren Kopien und mehreren disjunkten Wegen im Netzwerk
  - *Write-Completion*: Benötigt explizite ACKs von jeder Kopie
  - *Write*-Atomarität: Nicht notwendigerweise atomar voraussetzbar (s. folgendes Beispiel)

Problem: Write-Atomarität



Deadlock, Livelock, Starvation

Lösung: *Request-Response (Reply)*-Protokoll

- Ähnliche Fragestellungen wie zuvor
  - Ein Knoten kann zu viele Nachrichten bekommen (Hot-Spots)
  - Flusskontrolle kann zu *Deadlocks* führen
  - Getrennte *Request-*/ *Reply*-Netzwerke mit *Request-Reply*-Protokoll oder NACKs, birgt aber Gefahr von *Livelocks* und Verkehrsproblemen
- Neues Problem: Protokoll als nicht einfaches *Request-Reply*
  - Z.B. *read-exclusive* beim *Write* generiert Inval-Requests (wodurch ACK-Antworten erzeugt werden)
  - Weitere Lösungsmöglichkeiten durch die Latenz reduziert wird und mehr Gleichzeitigkeit ausgenutzt werden kann
- Dabei müssen Livelock- und Starvation-Situationen beachtet werden

Performance

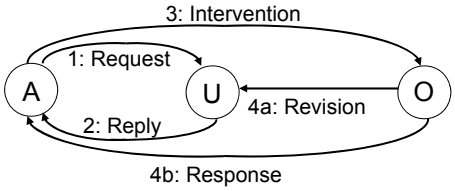
Verbesserung von

- Latenzzeit
  - Protokoll-Optimierungen zur Reduzierung von Netzwerk-Transaktionen auf dem kritischen Pfad
  - Aktivitäten überlappen oder beschleunigen
- Durchsatz
  - Reduzierung der Anzahl an Protokoll-Operationen pro Aufruf

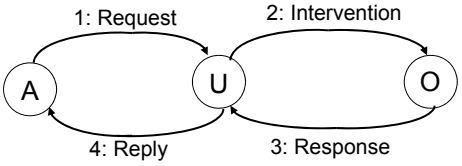
Skalierungsverhalten:
Wie ändert sich Latenz und Durchsatz mit wachsender Knotenanzahl?

Protokolle zur Latenz Verbesserung (1)

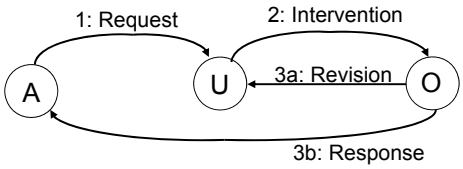
Nachrichten-*Forwarding*: Speicher-basierte Protokolle



a) striktes *Request-Reply*



b) *Intervention-Forwarding*

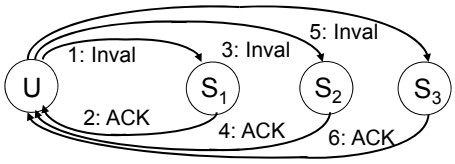


c) *Reply-Forwarding*

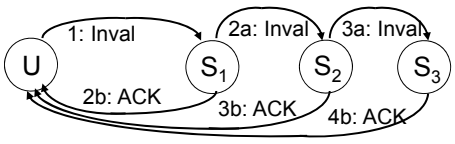
A: Anfragender
U: Ursprung
O: *Owner*

Protokolle zur Latenz Verbesserung (2)

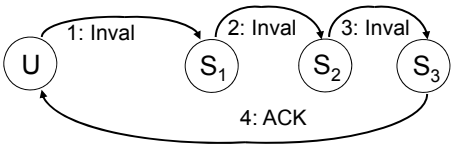
Nachrichten-*Forwarding*: Cache-basierte Protokolle



a) Striktes *Inval-ACK*



b) *Inval* Forwarding



c) Letzter Teilnehmer *ACK*

Weitere Latenzzeitoptimierung

- Durch Hardware-Beschleunigung aller Aktionen auf dem kritischen Pfad
  - SRAM für das Directory
  - Nur ein Bit pro Block in SRAM für Bestimmung, ob Protokoll einbezogen werden muss
  - … oder SRAM als Cache organisiert
- Überlappen von Operationen auf dem kritischen Pfad
  - Mehrere Invalidierungen zu einer Zeit bei Speicher-basierten Protokollen
  - Überlappen von Invalidierungen und ACKs bei Cache-basierten Protokollen
  - Lookups im Directory und Speicher oder Lookup mit Transaktionen
    - spekulative Protokoll-Operationen

Durchsatz Erhöhung

- Reduzierung der Anzahl an Transaktionen pro Operation
  - Invals, ACKs, Ersetzungshinweise
- Reduzierung der Belegungszeit der Einheiten oder Overhead der Protokollbearbeitung
  - Transaktionen sind kurz und häufig, dadurch Belegungszeit wichtig
- Pipelining der Protokollbearbeitung
- Viele Möglichkeiten der Latenzreduzierung erhöhen auch implizit den Durchsatz
  - Z.B. Forwarding zu Dirty-Knoten, Einsatz von weiterer Hardware auf kritischem Pfad

Komplexität

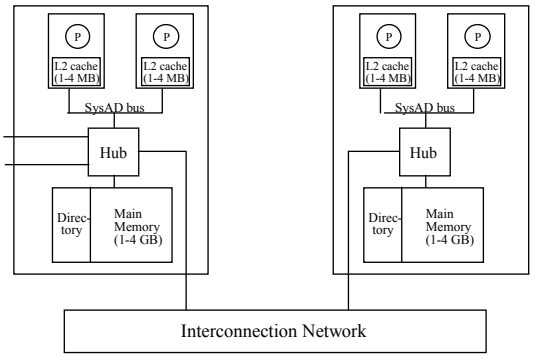
- Diese Cache-Kohärenzprotokolle sind sehr komplex
- nicht nur ausgewählter Ansatzes bestimmt die Komplexität
  - Konzeption und Protokolldesign vs. Implementierung
- Leistungssteigerungen erhöhen oftmals Komplexität, macht auch den Korrektheitsbeweis schwieriger
  - Mehr Gleichzeitigkeit => potentielle Race-Conditions
  - Nicht striktes Request-Reply
- viele subtile Spezialfälle
  - Erfordert tiefes Verständnis der Funktionsweise
  - Automatisierte Verifikation ist dabei notwendig, aber schwierig

Im Weiteren genauere Betrachtung der speicher-basierten Protokolle

Flache, Speicher-basierte Protokolle

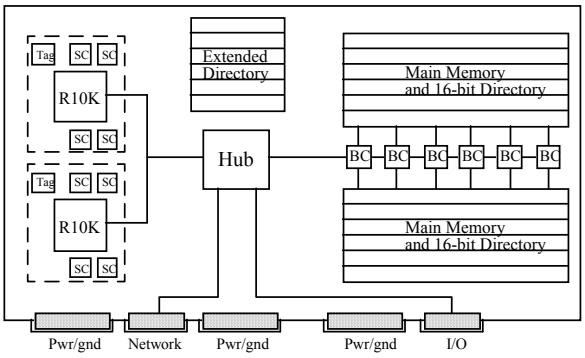
- Beispiel: SGI Origin2000
  - Protokoll ähnlich zu gezeigten Ansätzen, aber mit unterschieden in der Lösung von Trade-Offs
  - Protokoll auch in anderen Rechnersystemen eingesetzt
- Im Folgenden:
  - Systemübersicht
  - Kohärenz-Zustände, Repräsentation und Protokoll
  - Korrektheit und Performance Trade-Offs
  - Implementierungsfragen
  - Quantitative Leistungscharakteristiken

Übersicht: Origin2000 System



- *Directory* im separaten DRAM Speicher oder im DRAM des Hauptspeichers
- bis zu 512 Knoten (1024 Prozessoren)
- MIPS R10k, 195 MHz, 390 MFLOPS (sehr alter Prozessor!)
- Systembus mit 780 MByte/s maximale Bandbreite
- vom Hub zum Router-Chip und Xbow (I/O) mit 1,56 GByte/s (Link)

Origin2000 Knoten

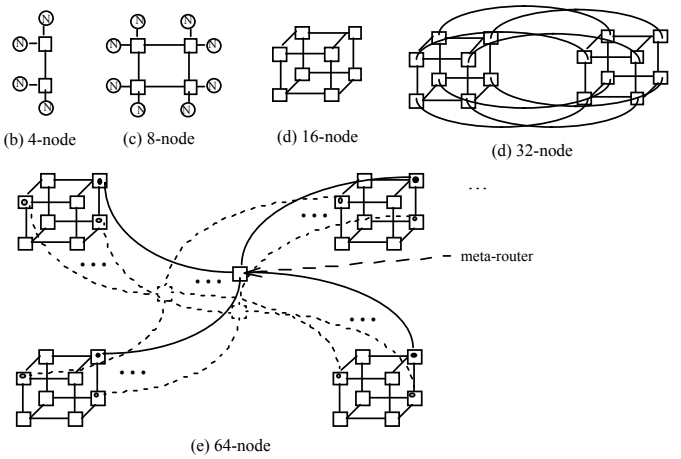


Connections to Backplane

Hub Chip hat

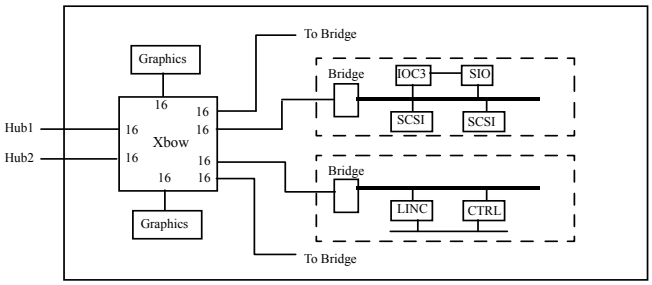
- 500.000 Gatter, 0.5µm CMOS
- Buffer für ausstehende Transaktionen für jeden Prozessor (je 4)
- zwei Block-Transfer-Einheiten (Speicher kopieren und füllen)
- Interface zu Prozessoren, Speicher, Netzwerk und I/O
- Unterstützung von Synch-Primitiven und zur Seitenmigration
- zwei **nicht** *Snoopy*-kohärente Prozessoren

Origin: Netzwerk



- Jeder Router hat sechs Paare aus 1,56 GByte/s unidirektionalen Links
  - Zwei Link-Parre zu Knoten und vier Link-Paare zu anderen Routern
  - Latenzzeit: 45 ns Pin-zu-Pin über einen Router
- Flexible Kabel bis zu einem Meter Länge
- Jeder Link hat vier virtuelle Kanäle: Request, Reply, zwei weitere für priorisierte Nachrichten und I/O

Origin I/O



- Xbow ist ein einfacher Router (Crossbar mit 8 Ports) der mit zwei Knoten (Hubs) und bis zu vier I/O-Bridges und zwei Grafikkarten verbunden ist
- Graphik-Karten sind direkt am Xbow angeschlossen
  - Real-Time und Bandbreitenreservierung für Video
- Globaler I/O Space: Jeder Prozessor kann jedes Device direkt verwenden
  - Setzt uncached Speicheroperationen oder kohärentes DMA im I/O-Adressraum ein
  - Jedes I/O-Gerät kann lesend oder schreibend auf jede Adresse zugreifen

Origin: Cache- und Directory-Zustände

- Zustände des Caches: MESI
- Sieben *Directory*-Zustände
  - *Unowned*: keine Kopie in einem Cache, Speicher hat gültigen Wert
  - *Shared*: ein oder mehrere Caches haben *shared* Kopie, Speicher *valid*
  - *Exclusive*: ein Cache (s. Zeiger) hat Block in *Modifed* oder *Exclusive* Zustand
  - Drei ausstehende oder *Busy*- Zustände (jeweils für einen der vorherigen Zustände)
    - Zeigt, dass das *Directory* bereits eine Anfrage an den Block bekommen hat und diese noch nicht abgeschlossen wurde.
    - Aktuelle Anfrage kann nicht lokal bearbeitet werden; Senden an nächsten Knoten und Warten
    - Keine weitere Anfrage an den Block derzeit möglich
  - *Poisoned*: eingesetzt für effiziente Migration von Seiten

Origin: Directory-Struktur

- Flach, speicher-basiert: alle *Directory*-Infos am Ursprung
- Drei *Directory*-Formate
  - (1) falls *Exclusive* in einem Cache: Eintrag ist Zeiger auf diesen Prozessor (nicht Knoten!)
  - (2) falls *Shared*: Bit-Vektor mit Bit als Hinweis auf Knoten
    - Invalidierung zu einem Knoten erzeugt *Broadcast* zu beiden Proz.
    - Zwei Directory-Größen, abhängig von Skalierung
      - 16 Bit Format (bis 32 CPUs), gehalten im DRAM des Hauptspeichers
      - 64 Bit Format (bis 128 CPUs) mit extra Bits im Extension-Memory
  - (3) bei größeren Maschinen (p > 64 Knoten) gröbere Vektoren: jedes Bit entspricht p/64 Knoten
    - Invalidierung wird zu allen Knoten der Gruppe geschickt in der nachfolgend ein *Broadcast* erzeugt wird
    - Maschine kann dynamisch zwischen Bit-Vektor und groben Vektor umschalten

Wie werden Read und Write Anfragen behandelt?

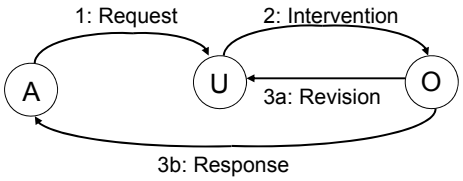
Handhabung eines Read-Misses

Hub betrachtet zuerst die Adresse
- Falls entfernt, sende Anfrage an Ursprung der *Read-Miss* erfüllt
- Falls lokal, betrachte *Directory*-Eintrag und hole Block aus Speicher

Directory liefert einen von mehren möglichen Zuständen
- *Shared* oder *Unowned* Zustand:
  - Falls *shared*, entsprechendes *Presence*-Bit im *Directory* setzen
  - Falls *unowned*, wechsle in *exclusive* Zustand, nutze Zeiger-Format
  - Anfragenden mit Block antworten
    - nur Antwort an Anfragenden (falls lokal, kein Netzwerk-Transfer)
  - Gleichzeitig zum *Directory*-Zugriff wurde der Speicher bereits spekulativ ausgelesen
    - *Directory Look-Up* kommt einen Zyklus früher
    - Falls *Directory shared* oder *unowned* liefert, ist spekulativer Zugriff vorteilhaft
    - Falls nicht, war spekulativer Zugriff vergebens
- *Busy* Zustand: kann nicht sofort handeln
  - NACK, damit der Anfrage-Buffer nicht zu lange belegt bleibt

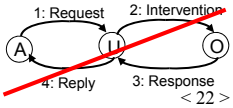
Read-Miss auf einen Block im Exclusive Zustand (1)

- *Exclusive*-Zustand - der interessanteste Fall!
  - Falls Ursprung nicht der Owner, dann wird der aktuelle Block vom Owner sowohl zum Anfragenden als auch zum Ursprung gesendet
  - Verwendet Weiterleitung der Anfrage wegen geringer Latenz und Netzwerkverkehr
    - Kein striktes *Request-Reply, sondern Reply-Forwarding*



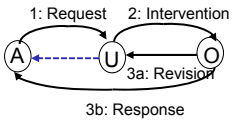
A: Anfragender
U: Ursprung
O: *Owner*

- Problem mit reinem „*intervention forwarding*“
  - Am Ursprung könnten O(*p*k*) ausstehende Anfragen vorliegen
    - Mit *Reply-Forwarding* nur k, da NACK-Antworten zum Anfragenden gehen (k = max. Anzahl an ausstehenden Anfragen pro Knoten)
  - viel komplexer und weniger performant



Read-Miss auf einen Block im Exclusive Zustand (2)

- Am Ursprung:
  - Setze *Busy*-Zustand im *Directory*. Falls weitere Anfrage an Block im Busy-Zustand, dann antworte mit NACK.
    - Entspricht allgemeiner Philosophie des Protokolls
    - Nur im Zustand *Shared* oder *Exclusive* muss auf Revisionsnachricht gewartet werden
    - Alternative: Anfrage bis Fertigstellung im Ursprung zwischenspeichern; aber Problem mit Eingangs-Buffer
  - Setzen und Zurücksetzen entsprechender *Presence-Bits*
  - Nimmt Block als *Clean-Exclusive* an und sendet **spekulative** Antwort



- Beim *Owner*:
  - Falls Block *Dirty (M-Zustand)*:
    - Sende Daten-Antwort zum Anfragenden und *Sharing-Writeback* mit Daten an den Ursprung
  - Falls Block *Clean-Exclusive (E-Zustand)*:
    - Ähnlich, aber sende keine Daten (Nachricht zum Ursprung heißt *Downgrade*)
- Ursprung ändert *Zustand* in *Shared,* sobald Revisionsnachricht vom Owner eintrifft

Einfluss der Prozessoren aufs Protokoll

Weshalb spekulatives Antworten vom Ursprung?
- Anfragender muss sowieso auf Antwort vom *Owner* warten !?
- bringt doch keine Reduzierung der Latenz !?
- kann Daten doch immer nur vom Owner beziehen !?

- Verwendeter Prozessor ist derart entworfen, dass im *Clean-Exclusive* nicht mit Daten geantwortet wird
  - deshalb müssen Daten vom Ursprung kommen
  - beim *Intervention-Forwarding* wäre keine spekulative Antwort erforderlich

- Ermöglicht weitergehende Optimierungen
  - kein senden von Daten zum Ursprung, falls Block-Zustand *Clean-Exclusive* nur zwischen den Caches wechselt

Handhabung eines Write-Misses

Anfrage an Ursprung kann *Upgrade* (S→M) oder *Read-Exclusive* (I→M) sein

Am Ursprung:

- Falls Zustand *Busy:*
  - sende NACK
- Falls Zustand *Unowned*:
  - Falls RdEx, setze Presents-Bit, ändere Zustand in *Dirty* und Antworte mit Daten
  - Falls *Upgrade* (Block wurde bereits im Cache des ehemaligen Teilhabers ersetzt, und *Directory* bereits benachrichtigt), damit ist *Upgrade* nicht geeignete Anfrage
    - NACK Antwort (Anfrage anschließend mit RdEx stellen)
- Zustand ist *Shared* oder *Exclusive*:
  - Invalidierung muss an Teilhaber(n) gesendet werden
  - Nutze Reply-Forwarding; d.h. Teilhaber senden Invalidierung-ACKs an Anfragenden, nicht zum Ursprung

Write auf Block im Shared Zustand

- Außerdem am Ursprung:
  - Setzte *Directory* Zustand zu *Exclusive* und setze nur *Presence*-Bit des Anfragenden (schreibe den Zeiger)
    - stellt damit sicher, dass nachfolgende Anfragen an den neuen Owner weitergeleitet werden
  - Falls RdEx, sende “exclusive reply with Invals pending” an Anfragenden (mit Daten)
    - von wie vielen Teilhabern werden Invalidierungen erwartet?
  - Falls Upgrade, ähnlich “upgrade ACK mit Invals pending” Antwort (aber ohne Daten)
  - Sende Invals zu Teilhabern, die dann dem Anfragenden ACK senden
- Anfragender wartet auf alle ACKs um Operation abzuschließen
  - nachfolgende Anfragen an Block im Ursprung werden als *Intervention* zum Anfragenden weitergeleitet
  - für korrekte Serialisierung, Anfragender behandelt keine weiteren Anfragen bis alle ACKs von ausstehender Anfrage eingetroffen sind

Write auf Block im Exclusive Zustand

- Falls *Upgrade* (S→M):
  - nicht gültig, deshalb NACK Antwort senden
    - anderes *Write* zum Ursprung hat dieses *Write* geschlagen, deshalb ist Datum des Anfragenden ungültig
- Falls RdEx (I→M):
  - wie *Read*, Ursprung setzt *Busy* Zustand, setzt *Presence*-Bit und sendet spekulative Antwort
  - sende Invalidierung zum *Owner* mit Identität des Anfragenden
- Beim *Owner*:
  - falls Block im Cache im Dirty/*Modified* Zustand
    - sende "*Ownership xfer*" Revisionsnachricht zum Ursprung (ohne Daten)
    - sende Antwort mit Daten zum Anfragenden (überschreibt spekulative Antwort)
  - falls Block im *Clean-Exclusive* Zustand
    - sende "*Ownership xfer*" Revisionsnachricht zum Ursprung (ohne Daten)
    - sende ACK zum Anfragenden (ohne Daten, da bereits durch spekulative Antwort erhalten)

Handhabung von Writeback-Anfragen

- *Directory* Zustand kann nicht *Shared* oder *Unowned* sein
  - Initiator (*Owner*) hat den Block im *Dirty/Modified* Zustand
  - Falls weitere Anfrage zum Setzen des Zustands zu *Shared* kommen sollte, wäre diese bereits zum *Owner* weitergeleitet und Zustand wäre somit *Busy*
- Zustand ist *Exclusive*
  - *Directory* Zustand zu Unowned setzen und ACK zurücksenden
- Zustand ist Busy: interessante *Race-Condition*
  - Busy, weil Intervention wegen einer Anfrage eines anderen Knotens Y zum eigenen Knoten weitergeleitet wurde, der aber nun das *Writeback* macht
    - Intervention und *Writeback* haben sich gegenseitig überschnitten
  - Y's Operation ist bereits in Arbeit und hat schon ggf. *Directory* verändert
  - *Writeback* darf nicht wegfallen (einzige valide Kopie)
  - Kann *Writeback* nicht NACK schicken und warten bis Y fertig
    - Y's Cache hat valide Kopie während die *Dirty* Kopie in Speicher geschrieben wird

Ersetzung von Shared Blöcken

- Könnte sofort einen Hinweis auf Ersetzung zum *Directory* senden
  - Um den Knoten aus der *Sharing*-Liste zu nehmen
- Könnte den Hinweis bei der nächsten Invalidierung zum *Directory* senden
- Aber keine Reduzierung des Netzwerkverkehrs
  - Es muss explizit ein Ersetzungshinweis gesendet werden
  - Verlagert den Verkehr nur auf einen anderen Zeitpunkt
- Original-Protokoll verwendet keine Ersetzungshinweise
- Auch ohne diese weitere Ersetzungsnachricht schon viele Transaktionstypen in Origin 2000 notwendig
  - Kohärenter Speicher: 9 *Request*-Transaktionstypen, 6 Inval/Intervention, 39 *Reply*
  - Nicht kohärent (I/O, Synch, Spezialoperationen): 19 *Request*, 14 *Reply* und keine Inval/Intervention

Bewahren der Sequentiellen Konsistenz

- R10k verwendet ein dynamisches Scheduling der Instruktionen
  - Prozessor erlaubt das Anstoßen und Ausführen von Speicheroperationen außerhalb der Programm-Ordnung (out of order)
  - Erscheinen aber auf dem Bus in Programm-Ordnung
  - Erfüllt nicht die hinreichenden Konsistenzbedingungen, aber SC
- Bzgl. SC
  - Ein Write auf shared Block hat zwei unterschiedliche Typen an Replies an den Anfragenden zur Folge
    - *Exclusive Reply* vom Ursprung: *Write* wurde durch Speicher serialisiert
    - *Invalidation ACKs*: Prozessoren haben *Write* ausgeführt
  - Prozessor kann aber nur ein *Reply* verarbeiten
    - HUB des Anfragenden behandelt *Replies*
  - Erhaltung von SC erfordert, dass HUB auf alle INVAL ACKs warten muss bevor Prozessor Antwort bekommt

hier keine weitere Vertiefung in die Details

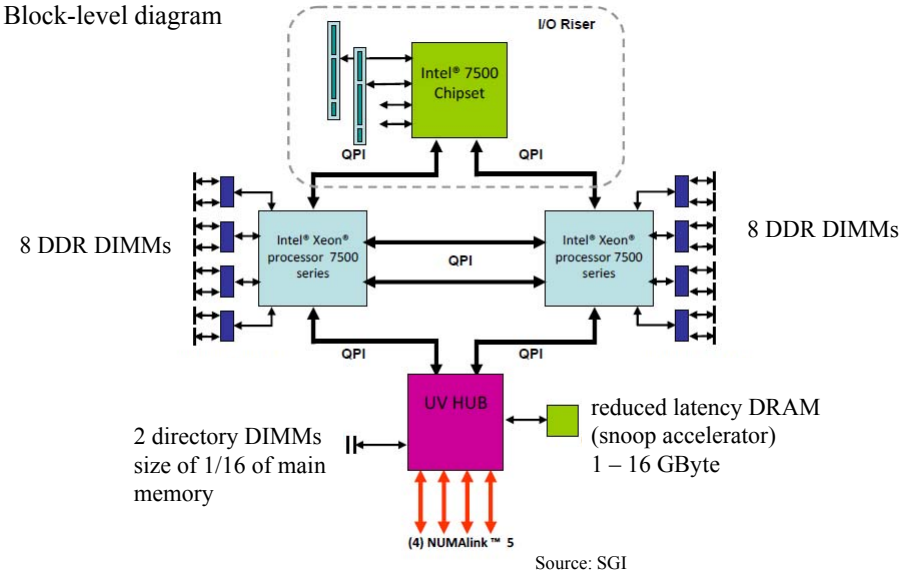
Flat, Memory-based Protocol

Example: SGI Altix UV 1000 (Ultraviolet)

- Intel Xeon E7 family (Xeon 7500)
- Altix UV 100 : 96 processor sockets (max. 960 cores)
- Altix UV 1000: scales up to 256 sockets (max. 2560 cores)
  - Nov. 2011; Redhat Linux with 1280 cores and 8 TByte memory
- 2D-torus NUMAlink5 und MPI offload engine (15 Gbyte/s)
- Directory-based coherence protocol
- 128 byte memory blocks
- Directory entries
  - States
    - Unowned – block is not cached
    - Exclusive – copy of block is exactly in one cache
    - Shared - copy of block is in more than one cache
  - Presents bits
    - Bit vector indicated which caches contain a copy of the block
- Invalidation method
  - Write invalidates copies and acquires exclusive ownership

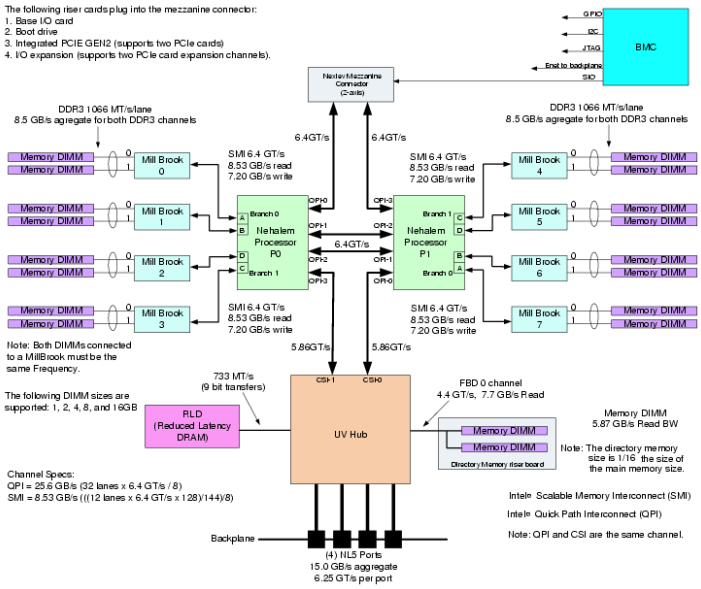
Altix UV Compute Blade

Block-level diagram



Source: SGI

Altix UV 1000: Functional Block Diagram



Source: SGI

Paderborn Center for Parallel Computing

Ultraviolet HUB

- Links cache-coherent Intel QPI interconnect with NUMAlink cache-coherent environment
- Manages directories and snoop acceleration
  - Global Register Unit (GRU) extends cache-coherency to NUMAflex architecture
    - 53 bits physical address space, 60 bits virtual address space
    - external TLB with large page support
    - distributed page initialization
    - block copy operation; asynchronous copy mechanism of aligned memory
    - scatter/gather operations; fixed stride and list-driven
    - update cache for atomic memory operations (AMO); spin locally on a copy of a NUMAflex managed AMO variable
  - Active Memory Unit supports atomic memory operations
    - Two types of AMO supported; CPU AMO and NUMAflex AMO
  - Two Intel QPI interfaces lets Hub looks like another memory controller (aggregated 50 GByte/s)
  - NUMAlink Interconnect implements 4 NUMAlink-5 ports (aggregated 40 GBytes/s)

Paderborn Center for Parallel Computing

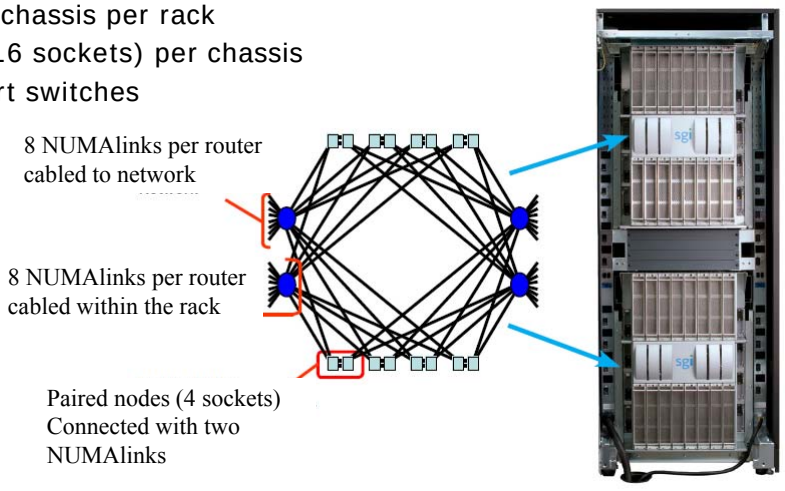
Numalink-5 Interconnect

- Numalink Cables
  - Based on 12x IB-Cables
  - Linkspeed 7.5 GByte/s per direction, 15 GByte/s bidirectional
- UV Router
  - 16 Numalink-5 ports
  - Packed in groups of 4 routers
    - QCR-Quad-compact Routers
- UV Hub-Chip
  - 4 Numalink-5 ports
  - 2 QPI ports
  - Active memory unit (AMU) and global register unit (GRU)

Altix UV 100: Interconnect Topology

32 socket interconnect topology

- Two blade chassis per rack
- 8 blades (16 sockets) per chassis
- four 16 port switches

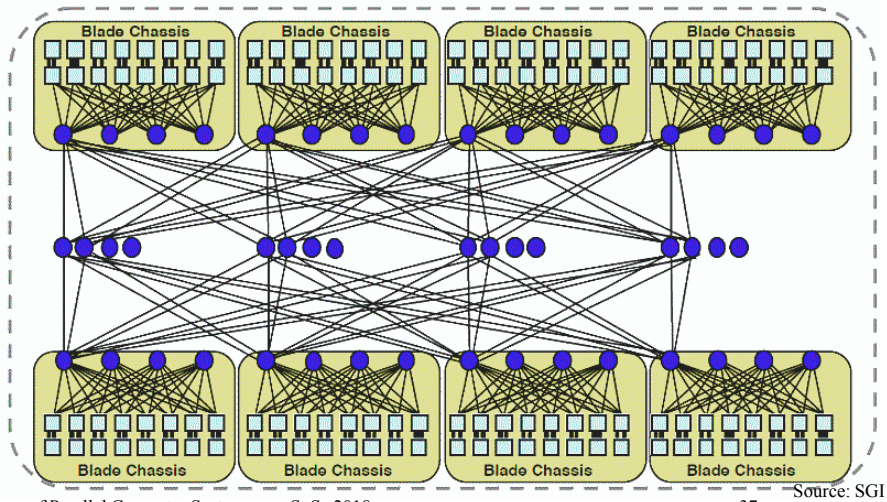


Source: SGI

Altix UV 1000: Interconnect Topology

256 socket interconnect topology (fat-tree)

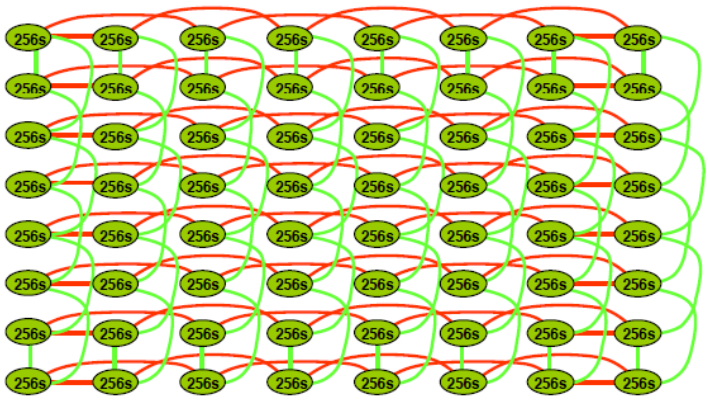
- 8 racks with 16 blade chassis
- 16 external 16-port switches



Source: SGI

Altrix UV: Interconnect Topology (2)

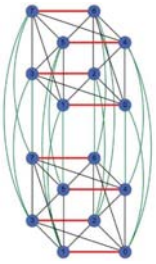
- 16,384 socket
  - 64 complexes with 256-socket fat-trees
  - 8 x 8 torus



Source: SGI

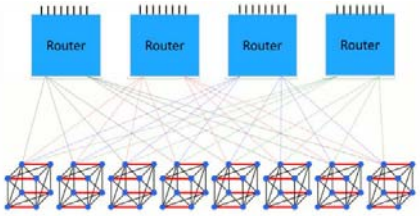
SGI Ultra Violett 2000

- max. 256 Sockets
- Intel E5-4600, 2.4 Ghz – 3.3 GHz
- max. 64 TByte cache-coherent shared memory
- 16 port Numalink 6 Hub
- 10u rackmount enclosure



Direct cabled
32 sockets

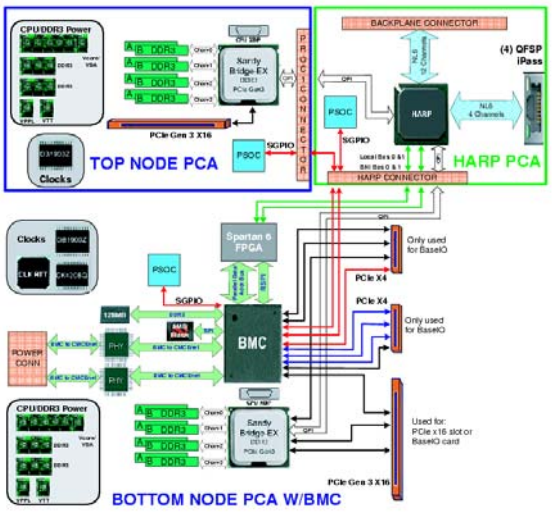
interconnecting
of 2 racks



SGI UV2000

Compute blade

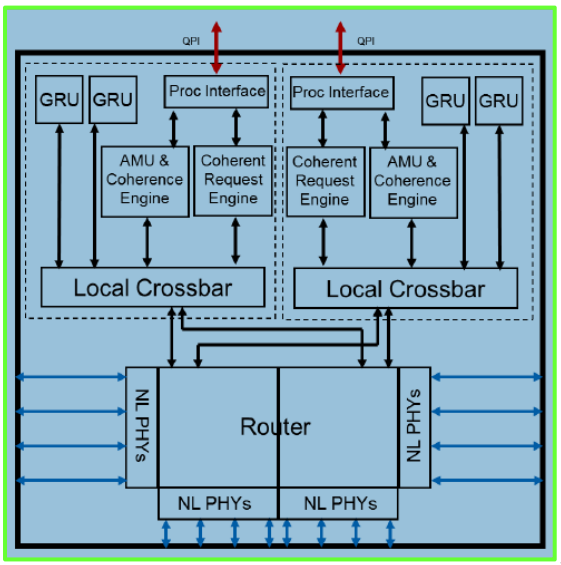
- Two processor sockets
  Intel Xeon E5-4600
- ASIC (NUMAlink6 hub)
  - Two QPI interfaces,
    one to each processor
    (~32 GByte/s)
  - 16 NUMAlink6 ports
    (~ 40 GByte/s aggr. bw)
- 512 MByte of main memory



ASIC Architecture

Coherence home engine

- Multi-node coherence tracking
- Directory is stored in socket DRAM
- 64-way directory cache accelerates address reuse and strided accesses



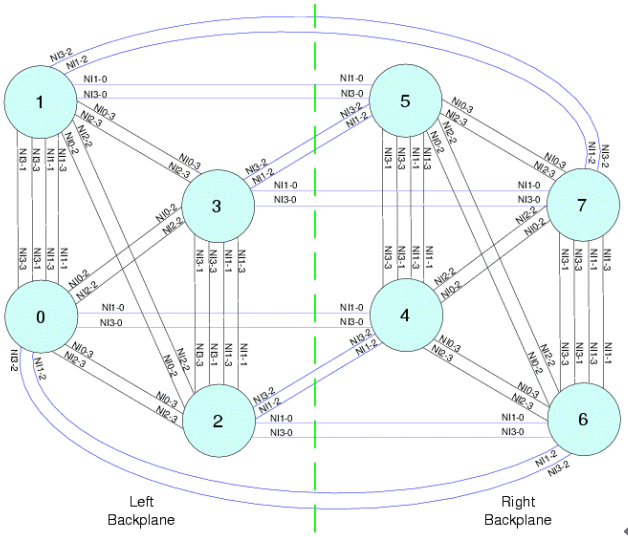
SGI UV2000

Enclosure (16 sockets)

- Up to 8 blades
- interconnected in an enhanced hypercube topology

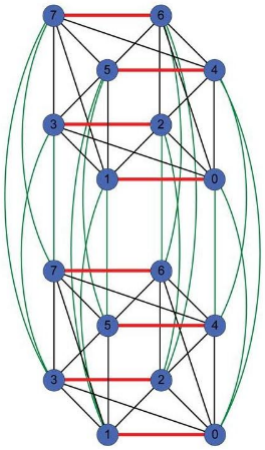
12 NL6 are connected to the backplane of the enclosure

4 NL6 are free for inter enclosure connections

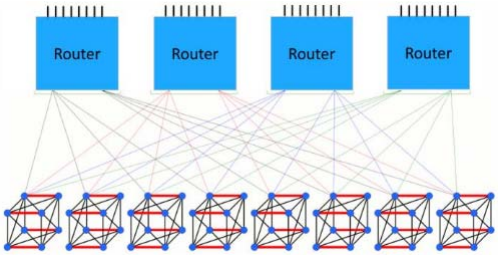


SGI UV2000

Two enclosures (one rack) direct cabled (32 sockets)



Interconnecting two or more racks



4 routers connected to 1 of 8 vertices.
32 routers are needed to connected all vertices.