Skalbarhet hos operativsystem för nya flerprocessorkärndatorer

Wictor Lund

4 april 2011

Kandidatavhandling i datateknik

Åbo Akademi

Institutionen för informationsteknologi

Handledare: Jerker Björkqvist

Innehåll

Referat

Mycket arbete sätts idag på att göra operativsystem mera skalbara, det finns en trend som visar på att processorkärnor i datorer ökar. Traditionella operativsystem använder delat minne för att göra synkronisering mellan processorkärnor. Delat minne mellan många processorkärnor är långsamt, därför behöver man undersöka hur framtidens operativsystem skall se ut för att de ska klara av ökningen av processorkärnor. Jag tar upp problem med traditionella operativsystem, hur man kan lösa dem och i vilken utsträckning det är lönsamt att göra detta.

Det finns olika tekniker som idag används för att göra dagens operativsystem mera skalbara. Man gör låsning av datastrukturer mera finkornig och man använder *RCU* för undvika problemen som uppstår vid finkornig låsning.

Forskare vid ETH Zurich; Microsoft Research, Cambridge och ENS Cachan Bretange har tagit fram en ny modell kallad multikernel, för hur man kan konstruera operativsystem. Modellen är gjord från grunden för att köras på datorer med många processorkärnor, istället för traditionella operativsystem som är från början gjorda att köras på enprocessordatorer. Jag funderar över hur man kan använda denna modell för att kunna köra framtidens operativsystem på datorer med processorkärnor betydligt flera än idag.

1 Inledning

Enligt Borkar vid Intel kommer Moores lag att medföra att antalet processorkärnor per chip kommer att öka exponentiellt med tiden. Detta beror på Pollacks regel (eng. Pollacks rule) [2] som säger att prestandaökningen är ungefärligt proportionell mot kvadratroten av ökningen av komplexiteten på en processorkärna. Om man däremot ökar antalet processorkärnor på ett chip så har man möjlighet att få linjär prestandaökning. [2]Man får alltså den största prestanda ökningen genom att öka på antalet processorkärnor, och det är åt det hållet som utvecklingen går. Dagens operativsystem är skalbara för dagens antal processorkärnor, men inom en snar framtid kan man inte vara säker på att operativsystemen fortfarande klarar av att hantera antalet processorkärnor på ett effektivt sätt. Därför måste man lägga ner tid på att hålla operativsystemen skalbara för dagens datorer.

Jag kommer i denna avhandling att presentera orsaker till varför man inte kommer att kunna få dagens operativsystem att vara skalbara för ett stort antal processorkärnor. Jag kommer också att presentera nya idéer som finns i området som kan ge en bild av hur framtidens opera-

tivsystem kommer att se ut och hur de kommer att skilja sig från dagens. Jag kommer att kalla dagens operativsystem för traditionella för att uttrycka en skillnad i tankesätt mellan dagens och framtida generationer av operativsystem. Jag kommer att fokusera på skalbarhet för antalet processorkärnor även om andra områden också kunde vara relevanta.

För att förstå hur morgondagens operativsystem skall se ut, måste man ha en insyn i hur hårdvaran kommer att utvecklas mot framtiden. När man har många processorkärnor behövs mera minnesbandbredd och minskad minneskomplexitet, det finns olika tekniker som man kan använda för att göra detta, bl.a. NUMA och icke-gemensamma minnesrymder. NUMA-system som AMDs Opteron och Intels Nehalem arkitekturer blir mera vanliga, därför kan processorers egenskaper komma att påverka hur skedulering måste göras. I framtiden är det inte säkert att man använder gemensamma minnesrymder i datorerna, jag tar opp olika sätt hur man kan göra operativsystem som hanterar detta. Resultat säger att det är fördelaktigt att ha mera diverse arkitekturer [5, 15]; operativsystemen måste i framtiden hantera detta.

2 Nya flerprocessorkärndatorer

Rubriken för denna avhandling innehåller *nya flerprocessorkärndatorer*, detta begrepp måste definieras i sin kontext. Konsekvenserna av Ahmdahls- och Moores lag gör att trenden för ny hårdvara är annorlunda än vad den var före processorernas klockfrekvensökning började planas ut. [2] Trenden går mot flera processorkärnor och mera diverse hårdvaruarkitekturer. [2, 15]

Bör påpekas att *nya flerprocessorkärndatorer* inte hänvisar till datorer med två, fyra eller sex processorkärnor, utan många flera. Jag hänvisar till datorer som kommer att ge operativsystemen skalbarhetsproblem p.g.a. synkronisering mellan processorkärnor, jag kommer att gå närmare in på detta ämne i avsnitt 3.3 på sidan 7.

2.1 Hårdvaruresurser på ett chip

Borkar påstår att Moores lag fortfarande kommer att öka antalet transistorer på ett chip exponentiellt. [2] Det finns många alternativ för var man kan göra med dessa transistorer. Om man följer en modell där transistorerna kan användas till resurser som representerar *grundkärnekvivalenter* (eng. *base core equivalents*). Denna modell används i [15]. Dessa resurser kan användas för att göra enkla små processorkärnor, eller större processorkärnor som kostar flera resurser än de små.

I [15] kommer Hill och Marty fram till m.h.a. denna modell att det inte är effektivt att bara ha processorkärnor med bara en storlek. I avsnitt 3.3 på sidan 7 kommer jag fram till att Amdahls lag ger mycket dåliga skalbarhetsegenskaper på en dator med 1000 processorer med ett program som är endast 1% sekventiellt. Enligt [15] så får man bättre skalbarhetsegenskaper om man använder asymmetriska processorkärnor. Man kan också tänka sig att man kan göra processorer där resurserna på chipet kan omfördelas dynamiskt, mellan att vara stora och små processorkärnor. De nämner i [15] att sådana chip kan vara svåra att implementera.

2.2 Heterogena hårdvaruarkitekturer

Heterogena hårdvaruarkitekturer är i denna text datorsystem som innehåller olika typer av processorarkitekturer, så att alla processorer inte ser likadana ut från programmerarens synvinkel. T.ex. om man har en x86 processor och en ARM processor i samma dator. De flesta datorer i allmänt bruk är idag *homogena*. En överblick över ett multikerneloperativsystem (tas upp i avsnitt 5 på sidan 13) på en *heterogen* hårdvaruarkitektur finns i figur 5.1 på sidan 13.

Man inser att det är svårare att programmera *heterogena* system än *homogena* system. Idag

så ses *heterogena* system inte som en helhet. Man kan ha en dator med ett grafikkort i sig för att göra beräkningar, men operativsystemet körs bara på värddatorn. För att bättre kunna utnyttja *heterogena* hårdvarusystem borde man se dem som en helhet.

2.3 Asymmetriska processorer

Jag kommer att referera till *asymmetriska* processorer för att beskriva processorer med processorkärnor som har asymmetrisk design. Alla processorkärnor på en *asymmetriska* processor ser från programmerarens håll ut likadana ut. Jämförelse mellan *symmetriska* och *asymmetriska* processorer gör med figur 3.1 på sidan 6 och figur 3.2 på sidan 6. Vad som gör *asymmetriska* processorer intressanta är att det är möjligt att göra dem *homogena*, och lättare att programmera än *heterogena* system.

2.4 Slutsatser om ny hårdvara

Hårdvaran kommer att bli mera diverse i framtiden enl. [15] och [2]. Detta kommer att påverka operativsystemens struktur. Man borde ha operativsystem som är oberoende av hårdvarans struktur för att ha en plattform som man kan programmera på för kunna återanvända både kompetens och kod.

3 Begreppet skalbarhet

Skalbarhet är ett vitt begrepp som används på olika sätt i olika situationer. Jag kommer här att ta upp några synsätt på skalbarhet. Avhandlingen kommer dock att handla om hur man gör operativsystem som beter sig skalbart på de nya flerprocessorkärn arkitekturer som förutspås komma. Skalbarhet i denna avhandling avser främst att man vill kunna köra ett program n gånger snabbare på en dator med n processorkärnor än på en dator med 1 processorkärna.

Ett tråkigt faktum gällande skalbarhet är att det kan vara så att systemet som helhet inte är skalbart om någon del av systemet inte uppfyller tillräckliga krav. Ett exempel som är centralt för denna text är när man har en dator med många processorkärnor som man kör ett operativsystem som inte klarar av att hantera att tillräckligt många trådar gör saker parallellt inne i kärnan.

En definition på skalbarhet är att ett system är skalbart när genomströmningen ökar proportionellt mot antalet indataenheter eller storleken på indatan.[1] Ett system kan vara skalbart då indatastorleken befinner sig i ett visst intervall. Utanför detta intervall har systemet dock egenskaper som inte överensstämmer med tidigarenämnda definition.

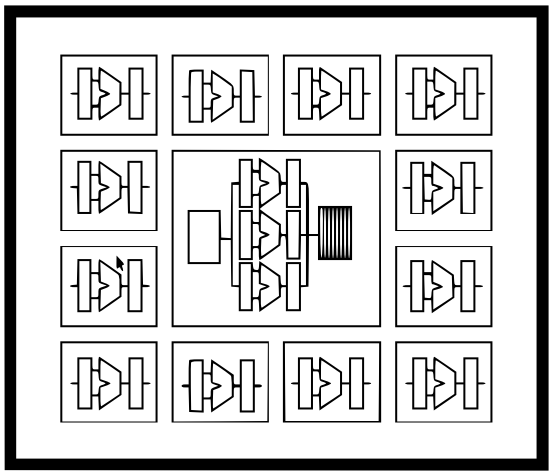
3.1 Hårdvarans skalbarhet

Hårdvara måste designas så att den klarar av de användarfall som specificeras. Det går ofta att lägga bördan över på mjukvaran, men hårdvaran måste internt klara av tillräcklig genomströmming utan att utsätta sig själv för flaskhalsar.

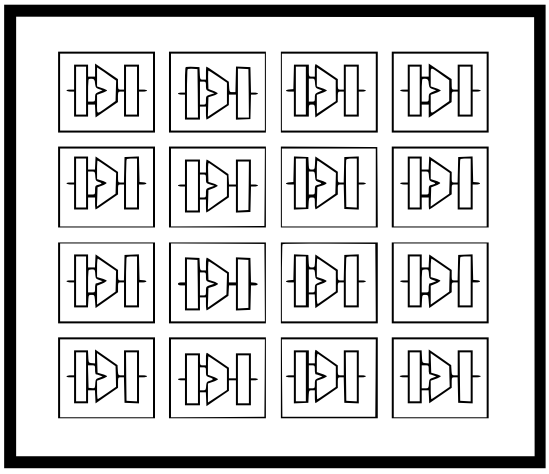
Forskning påstår att man m.h.a. *asymmetriska* och *heterogena* system kan uppnå bättre prestanda än med *homogena* och *symmetriska* system. [15, 2] Exempel på heterogena system idag är system som har ett programmerbart grafikkort för att göra beräkningar på. *Homogena asymmetriska* system finns mig veterligen inte många på marknaden, de som skulle vara mest intressanta (se avsnitt 2.3 på föregående sida).

Man kan jämföra en asymmetrisk processor med en symmetrisk processor med att jämföra figur 3.1 och figur 3.2. De små cellerna i figur 3.2 är processorkärnor som tar upp en resurs. I figur 3.1 har man använt fyra resurser för att göra en större, och snabbare processorkärna, och använder resten av resurserna till att göra små processorkärnor. Utgående ifrån [15] kan man säga att det inte är effektivt att ha mera än 64 processorkärnor på ett chip utan att fördela resurserna icke-symmetriskt.

Utmaningar som folk ställs inför idag gällande hårdvarudesign är hur man ska få massivt parallell hårdvara att ha bra prestanda på samma gång som den skall vara lätt att programmera.



Figur 3.1: Asymmetrisk processor, tagen från [15]



Figur 3.2: Symmetrisk processor, tagen från [15]

Hårdvaran skall ha ökande prestanda, men också hållas programmerbar, för uppnå detta måste mjukvaran anpassas. Industrin kan bli tvungen att helt och hållet revidera operativsystemens arkitektur.

3.2 Filsystems skalbarhet

Filsystems skalbarhet är egentligen inte relevant för denna avhandling. Jag tar upp begreppet eftersom när man diskuterar skalbarhet i servermiljöer så diskuteras ofta lagringsmedium.

Filsystems skalbarhet är något kan ses från två synvinklar, dels hur mycket parallellism filsystemet klarar av, och dels hur mycket ett filsystem klarar av att byggas ut. Ofta ligger filsystem på en ända disk som klarar inte av någon parallellism.

Ett viktigt krav för ett filsystem är att de skall klara av att spara tillräckligt mycket data

och vara utbyggbara. Detta är någonting som inte kommer att behandlas i denna avhandling. Ett annat krav är att filsystemet skall klarar av att överföra data, in och ut, med tillräckligt låg latens och tillräckligt hög genomströmning.

Ett filsystem som klarar av allt detta väl är *ZFS*. *ZFS* klarar av att köras på flera diskar, så att parallell IO är möjlig. Man också så länge filsystemet är i bruk lägga till diskar till systemet. Filsystemet klara också av mycket stora volymer, d.v.s. filsystemet klarar av att adressera tillräckligt många block. [16]

3.3 Mjukvaras skalbarhet på parallella datorer

Denna text handlar om hur man skall få maximal prestanda ur parallella datorer. För att en applikation skall kunna dra nytta av en parallell dator måste mjukvaran idag vara att köra i flera trådar eller köras som flera processer. Vi kommer osökt in på Amdahl lag som säger hur mycket snabbare man kan köra ett program på en parallell dator.

Den ursprungliga Amdahls lag säger att

$$\text{upsnabbning} = \frac{1}{s + \frac{1-s}{n}}$$

Där $s \leq 1$ är den sekventiella andelen av programmet, och n är antalet parallella instanser som körs. Man ser att $\lim_{s \to 0} \frac{1}{s+\frac{1-s}{n}} = n$, vilket säger att om hela programmet går att köras parallellt så får vi den önskade prestandan ur programmet. Detta är naturligtvis ofta inte fallet, och även om programmet är gjort för att köra i tillräckligt många trådar kan operativsystemet och hårdvara som t.ex. hårdskivan göra programmet delvis sekventiellt.

Det finns omarbetade versioner av Amdahl lag som inte är lika pessimistiska gällande prestanda med stort antal processorer. [15] Om man tar ovanstående formel med

$$s = 0,01$$

(1% av programmet körs sekventiellt) och

$$n = 1000$$

(vi har 1000 processorer) får man

$$\text{upsnabbning} = \frac{1}{0,01 + \frac{1-0.01}{1000}} \approx 91$$

, vilket är mycket dåligt, om man har köpt en 1000 processorers dator och vill köra sitt program. De hävdar i artikeln [15] att genom att koppla ihop resurser på ett chip på olika sätt så kan man uppnå högre prestanda i processorer med stort antal processeringselement. Det finns också de som hävdar att man m.h.a. *heterogena* och *asymmetriska* processorarkitekturer kan komma undan detta. [2] (se också avsnitt 2 på sidan 3)

3.4 Operativsystems nyttjandegrad av processorkraft

När man har en effektiv dator och en eller flera skalbara applikationer som man vill köra på en dator, är det rimligt att man ska få full prestanda ur dator utan att vara tvungen att tänka på någonting desto mera. Det har dock visat sig att det kan uppstå problem om inte operativsystemet kan hantera datorns resurser på ett optimalt sätt. Kärnan i denna text är att presentera olika tekniker för hur man kan få ett operativsystemet att fungera på så många processorer som möjligt. Vad som är naturligt på ett enkelprocessor system är inte längre självklart på ett system med många processorkärnor.

Användarens trådar bör schemaläggas (eng. schedule) så att cachen inte behöver laddas om vid varje kontextbyte (eng. context switch). Trådar inne i kärnan skall inte vänta på varandra. Att få en operativsystemskärna att bete sig skalbart är inte trivialt, och man måste ofta tänka om många saker för att få traditionella operativsystem att fungera på datorer med storleksordningar flera processorkärnor. Delar i operativsystemen som ofta måste tänkas om för att få dem att fungera på system med flera processorkärnor är främst låsning av datastrukturer, jag kommer att gå igenom låsning och synkronisering mera i detalj i avsnitt 4.1.1 och 4.1.2. Man måste också få operativsystemen använda minnesbussarna effektivt detta kommer jag att gå igenom i avsnitt 6.1.1. Multikernelarkitekturen har som grundidé kan kunna köras på *nya flerprocessorkärnsystem*, jag kommer att gå igenom detta i avsnitt 5.

4 Skalbarhet i traditionella operativsystemskärnor

Operativsystemskärnorna som används idag är blandningar av den monolitiska arkitekturen och mikrokernelarkitekturen. Den monolitiska kernelarkitekturen är gammal, och kanske den mest triviala operativsystemsarkitekturen, där hela kärnan är en enda stor klump som utför service åt användarapplikationer. Mikrokernelarkitekturen blev utvecklad under slutet av 1980-talet, men de flesta operativsystem byggda på denna arkitektur har idag hybridarkitektur, d.v.s. de har mikrokernelarkitektur med inslag den monolitiska arkitekturen.

Operativsystemen som används idag har sitt ursprung i början av 1990-talet då de allra flesta datorerna hade en processor och vissa serverdatorer kunde ha två processorer. Många operativsystem hade inte ens stöd för flera processorer och var konstruerade för att köra till största delen sekventiella användarprogram. Man har senare byggt om dessa operativsystem för flerprocessordatorer och gjort förbättringar för att kunna köra parallella användarprogram.

Traditionella operativsystem använder delat minne för att göra synkronisering mellan processorkärnor. [4, 5] Man använder sig av lås för att synkronisera tillgången till resurser. Problemet med lås från skalbarhetssynvinkel är att när en processor redan har anskaffat ett lås till en resurs, måste alla andra processorer som vill ha tillgång till denna resurs vänta. Man minskar på detta problem med att göra finkornigare lås, så att man delar ens befintliga resurser i flera mindre resurser. När man har flera resurser och på samma gång flera lås kan flera trådar göra saker inne i kärnan på samma gång, man kallar dessa för finkorniga lås och förklaras i närmare detalj i avsnitt 4.1.1.

Ett exempel på hur man får ett operativsystem att inte vara skalbart, är hur man gjorde multiprocessor stöd till Linux 2.0[4]. Då gjorde man ett ända stort kernel lås (eng. big kernel lock), så att bara en processor i taget fick tillgång till kärnan. Detta ledde till att man kunde köra parallella processorintensiva program väl i användarläge, men bara en tråd kunde komma åt kärnan i taget. Man fick alltså dåligt prestanda på en multiprocessordator med ett program som gjorde mycket operativsystemsanrop.

4.1 Förbättringar

Man gör idag stora uppoffringar i dylika operativsystem för att göra dem mera skalbara. I början av historien hade man lås som låste stora delar kod. Detta gör att endast en tråd kan exekvera i en kodsnutt. Men har senare börjat överge låsning av kod, och låser istället datastrukturer, så att flera trådar kan exekvera en kodsnutt, men bara en tråd åt gången har tillgång till samma

datastruktur. För att göra detta ännu mera skalbart delar man upp befintliga datastrukturer så att man får flera mindre datastrukturer med enskilda lås, detta kallas finkorniga lås (eng. fine-grained locks).

I figur 5.2 på sidan 16 presenterar man från [5] ett spektrum över olika låsnings och delningsmetoder som används, och i vilken riktning olika operativsystem är på väg.

4.1.1 Finkorniga lås (eng. fine-grained locks)

Man skiljer på olika låsnings modeller så att man har

1. ett enda stort lås, t.ex. så som i tidiga Linux 2.0.
2. grovkorniga lås, då man delar upp kärnan så att olika undersystem har olika lås, används i olika utsträckning idag.
3. finkorniga lås, man har fin indelning av datastrukturer, t.ex. man har olika lås för varje element i en länkad lista.

Ur skalbarhetssynvinkel är finkorniga lås det bästa alternativet. Det finns dock ett problematiskt fenomen som uppkommer med finkorniga lås det så kallade *Locking Cliff*-fenomenet. [10] När man använder finkorniga lås vill man ha så många lås som möjligt, och när antalet lås ökar kommer man småningom till en klippkant. Programmeraren kan inte längre överblicka vilka lås som låser vilka objekt. I detta skede får programmeraren svårt att se om det behövs ett nytt lås för en given operation; programmerar finner de ofta lättare att bara lägga till lås för att vara på säkra sidan. När man då bara lägger till lås faller man längre över klippkanten och det är svårt att gå tillbaka. [10] Då antalet lås ökar får man också prestandaproblem som beror på att man måste använda kostsamma *atomär*[1] operationer när man skaffar ett lås. Detta är speciellt ett stort problem på *cache-koherenta* datorer med många processorkärnor.

Man vill ha så många lås i ett givet system som möjligt, för att uppnå bra skalbarhet. På samma gång så vill man ha så få lås som möjligt så att man minimerar antalet kostsamma atomiska operationer för att uppnå prestanda, och för att inte falla över *The Locking Cliff*. För att nå detta optimum krävs planering och insikt i hur systemet fungerar, detta leder till en ökad arbetsbörda som är oönskad. Man vill alltså inte ha alltför finkornig låsning i ett operativsystem, bara tillräckligt för att alla processorkärnor skall kunna utföra arbete samtidigt. För att då uppnå

[1] Atomära operationer här är operationer som till synes sker direkt för alla processorkärnor i ett cachekoherent system. Ofta en jämför och byt (eng. compare and swap) operation som kräver att en processor läser och skriver i en enda operation.

optimal finkornighet måste man känna till hur många processorkärnor man har, och då gör man antaganden som gör att man inte längre har ett generellt operativsystem. [10]

Ett stort arbete som på senare tid har gjorts är att man har tagit bort Windows dispatcher lock, och ersatt det av en mängd mindre lås. Detta var enligt Russinovich ett svårt problem som skulle vara svårt att lösa med de utvecklingsmetoder som vanligen används inom företaget.[6] Generellt så försöker man ersätta stora lås med flera mindre, överallt där man kan.

4.1.2 RCU (eng. Read Copy Update)

En teknik som man kan använda för att helt undvika att låsa datastrukturer är *RCU*. *RCU* är bra att använda när man har många läsare, och inte så många skrivare till en datastruktur. En annan fördel med *RCU* är att man får högre prestanda är med vanliga lås, på grund av att man slipper göra lika många *atomära* operationer.[8]

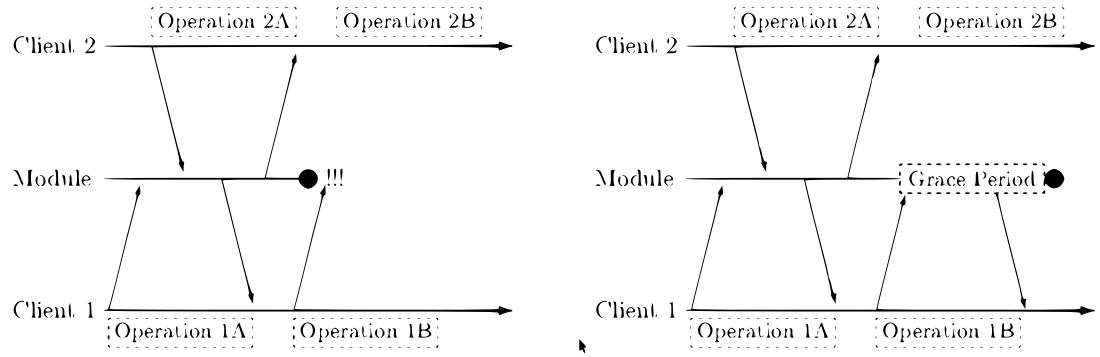
En uppdatering görs så att först utförs så mycket av en uppdatering att nya operationer som vill komma åt datastrukturen kan se det nya tillståndet, medan man låter gamla operationer se det gamla tillståndet. Efter att alla gamla operationer är utförda slutförs uppdateringen.[7]

Figur 4.1 beskriver hur *RCU* används i Linux för laddbara kernelmoduler. Bilden till vänster i figur 4.1 beskriver vad som händer om en modul avallokeras för tidigt, och en läsare (operation 1B) försöker läsa modulen. Till höger i figur 4.1 beskrivs vad som händer när man har en given deadline (eng. grace period) för när alla läsare är klara. En sådan deadline måste existera och kunna identifieras för att det skall gå att implementera *RCU*. Modulen borde avallokeras när operation 1B startar, men operation 1B vet inte om detta och det finns ingen synkronisering för att undersöka huruvida modulen är avallokeras eller inte. Problemet löses genom att avallokeras modulen först när man kan garantera att alla läsare som vet att modulen ännu finns har läst klart.

Det har nyligen gjorts förändringar i Linux VFS undersystem där man har börjat använda mera *RCU* tekniker för att uppnå bättre skalbarhet och prestanda.[9] Där tog man bort finkorniga lås och ersatte dem med *RCU*, med fördelen att man inte behöver göra lika många atomiska operationer när man gör en sökvägs sökning (eng. pathname lookup). En annan fördel är att hur många trådar som helst kan komma åt en resurs på samma gång.

4.2 Slutsatser

Fördelen är att traditionella operativsystem fungerar utmärkt på dagens datorer, det finns också mycket om hur de skall programmeras. För att hålla dessa operativsystem med i utvecklingen,



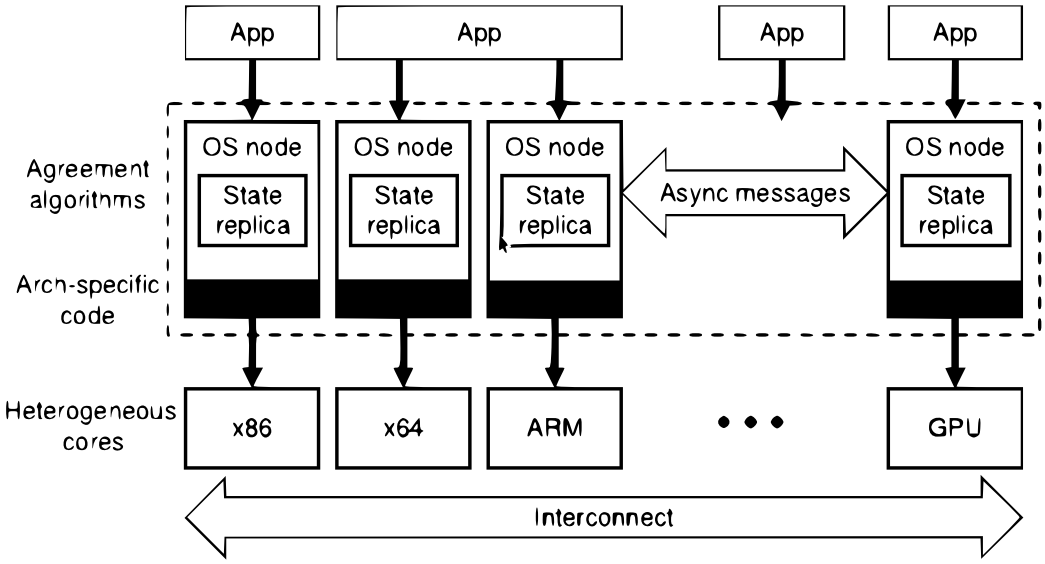
Figur 4.1: Hur RCU fungerar, tagen från [7]

d.v.s. få dem att vara skalbara för dagens och morgondagens antal processorkärnor, måste man göra stora kostsamma uppoffringar. [6, 9]

Det är önskvärt att få bort lås, finkorniga lås är bra, men de *atomiska* operationer som används vid låsning är dyra på cachekoherenta system. *RCU* är bra men all synkronisering går inte att implementera med den tekniken.

NUMA arkitekturer gör det mera naturligt att använda kopiering av olika datastrukturer, eftersom man får prestandaproblem annars. Traditionella operativsystem stöder NUMA, och är nog enligt mig det ända sättet som man kan få dessa operativsystem att klara av att köras effektivt på datorer med över 100 processorkärnor.

Det bästa vore att helt undvika lås för synkronisering mellan processorkärnor, men detta tror jag att är näst intill omöjligt att göra med de operativsystemsmodeller som används idag.



Figur 5.1: Multikernel modellen, tagen från [5]

5 Multikernelarkitektur

Eftersom det finns en trend som pekar på att traditionella operativsystem inte kommer att bete sig skalbart på framtidens datorer, håller man på och forskar man i hur man kan göra nya operativsystem som klarar av framtidens behov. Ett sätt göra ett mycket skalbart operativsystem är att använda sig av multikernelarkitekturen, som forskare vid ETH Zurich; Microsoft Research, Cambridge och ENS Cachan Bretange har tagit fram. [5] Multikernelarkitekturen bygger på tre design principer.

5.1 Arkitekturens designprinciper

Dessa är principer för hur ett multikernel operativsystem ska konstrueras dock nämns i [5] att man, för att göra ett operativsystem med optimal prestanda, måste ge efter på någon eller några av dessa principer. Det nämns att det kanske inte är det mest optimala att bara ha explicit kommunikation mellan kärnor som ligger nära varandra i ett kluster. På vilket sätt utvecklingen av synkronisering och låsning i ett multikerneloperativsystem går mot illustreras i figur 5.2 på sidan 16.

5.1.1 All kommunikation mellan kärnor är explicit

Man ser flerprocessorkärndatorers nätverksstruktur som viktig, man kan då tänka sig ett operativsystem i termer av distribuerade system. [5]Man ser en dator som ett nätverk där noder kan vara t.ex. processorkärnor och minnen. Generellt i ett nätverk så skickar man meddelande mellan noder hellre än att t.ex. dela minne som man gör i traditionella operativsystem. AMD Opteron arkitekturen kan ses som ett nätverk, detta förklaras vidare i avsnitt 6.1.2 på sidan 19, och illustreras i figur 6.1 på sidan 19.

En vanlig dator med flera processorkärnor som har likformig cache koherent minnesarkitektur kan ses som ett nätverk. Man använder nätverksanalogin i [5] när man skickar meddelanden mellan processorkärnor via cache hierarkin. En processorkärna kan skriva ett meddelande till en rad i cachen som sedan skickas vidare på grund av cache koherensprotokollet. En annan processor kan sedan ligga och vänta på ett meddelande på en cache rad och kan regera när den märker att cacheraden har förändrats av en annan processorkärna. Om det endast är en processorkärna skriver till en cache rad uppnår man bra latens när man skickar meddelanden mellan processorkärnor. [5]

Om man använder annan typ av hårdvara kan det vara mera naturligt att man använder explicita meddelanden för att kommunicera mellan processorkärnor, t.ex. om man har en dator som inte har gemensam minnesrymd. Det är lättare att skicka meddelanden mellan processorer med olika arkitekturer än att dela minne mellan dem. Explicit kommunikation möjliggör således att man kan köra ett enda operativsystem på en dator med heterogena processorkärnor.

5.1.2 Operativsystemstrukturen är hårdvaruoberoende

Man gör antagandet att datorers arkitektur kommer att bli mera diverse. Man gör idag olika typer av optimeringar i operativsystem för att kunna utvinna prestanda ur olika processorarkitekturer. De traditionella operativsystemens natur gör att de blir optimerade för en allmän klass av hårdvara. Eftersom hårdvaran blir allt mera diverse kommer detta förfarande inte att fungera i längden. [5] I [5] påstår de att det finns exempel på hur olika typer av optimeringar fungerar väldigt bra på en processorarkitektur men väldigt dåligt på en annan processorarkitektur. Optimeringar som påverkar operativsystemstrukturen blir på diverse datorarkitekturer oportabla. Man jämför Sun Niagaraprocessors L2 cache med någon mera vanlig arkitektur som t.ex. AMDs Opteron.

Slutsatsen är att man inte kan optimera hela operativsystemet för alla hårdvaruarkitekturer.

Vad man vill är att modellen för operativsystemet skall klara av att köras på ett stort omfång av hårdvara. Man vill att operativsystemet skall kunna anpassas till nya hårdvaruarkitekturer utan att i sin helhet behöva skrivas om.

Ett mål är att kunna köra multikerneloperativsystem på *heterogena* hårdvaruarkitekturer. Man ser en möjlighet att t.ex. kunna köra en operativsystemsinstans på ett programmerbart NIC (eng. Network Interface Controller), som eventuellt kör en ARM processor, detta finns med i figur 5.1 på sidan 13. [5]Man kunde då t.ex. på en server spara ström genom att stänga av alla CPU:er och bara låta nätverkskortet vänta på inkommande förfrågningar. Det är svårt med traditionella operativsystem att få ett helt *heterogent* system som en helhet, det är en möjlighet med multikernelarkitekturen.

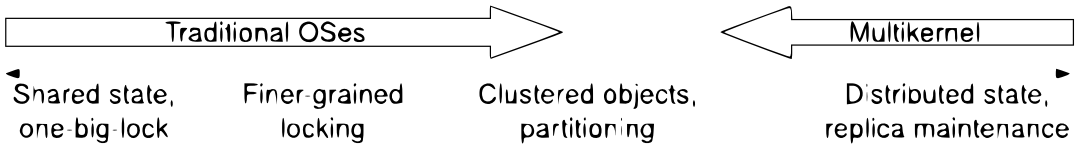
Den explicita kommunikationen mellan processorkärnorna gör det möjligt att kommunicera mellan olika diverse processorkärnor. Det är svårt att få ett traditionellt att som helhet köra på både en x86 processor och en ARM processor.

5.1.3 Se tillståndet för systemet som kopierat (eng. replicated) istället för delat

Ett multikernel-operativsystem kör i sin renaste form [5] en kernelinstans per processorkärna, detta illustreras i figur 5.1 på sidan 13. Jag nämnde att det var i sin renaste form, eftersom det kan visa sig mera effektivt att köra en kernelinstans på ett kluster av processorkärnor, se figur 5.2 på följande sida. [5] Man kan se det som att olika operativsystem strävar åt olika riktningar vad gäller att göra synkronisering, detta illustreras i figur 5.2. Eftersom all kommunikation sker via meddelande så delar man inget minne. Man har inga lås mellan kernelinstanser, som ställer till skalbarhetsproblem i traditionella operativsystem.

I figur 5.1, illustreras att i varje kernelinstans, så finns en kopia av operativsystemets tillstånd (eng. state replica). I traditionella operativsystem skulle tillståndet i vanliga fall vara delat mellan alla processorkärnor. Det finns fall när man i traditionella operativsystem också kopierar datastrukturer för att olika processorkärnor skall få snabbare tillgång till resurser, men detta göra i regel som optimeringar. [5, 13] Tillståndet som finns i kernelinstanserna kopieras mellan processorkärnor när förändringar sker. Alla kernelinstanser inte har behov av alla datastrukturer, så kopieringen behöver inte alltid göras till alla processorkärnor; t.ex. tillståndet för en process som inte körs på alla processorkärnor behöver inte finnas i alla kernelinstanser.

Processtrukturen är inte i grunden samma som i *POSIX*. En process i Barrelfish representeras av en samling speditör objekt (eng. dispatcher object), en på varje processorkärna som processen kan köras på. Kommunikationen sker inte mellan processer på operativsystemsnivå, utan mellan



Figur 5.2: Ett spektrum av låsning och delnings metoder, taget från [5]

speditörer, och därmed processorkärnor.[5]

Man vill gärna att ett multikernel-operativsystem skall, från användarens synvinkel, se ut som ett traditionellt operativsystem. Man vill med andra ord ha en *POSIX* processmodell. Man har i Barrelfish gjort ett API som liknar *POSIX* trådar, [5] det illustreras i figur 5.1 på sidan 13 att en *App* kan köras på flera processorkärnor, under flera kernelinstanser på operativsystemsnivå. Dock kan användarprogram dela minne på användarnivå, om hårdvaran tillåter detta. Det faktum att man kan köra program på detta sätt är goda nyheter, eftersom man inte helt behöver ändra på existerande programmeringsmodeller. Dock är *POSIX*-trådmodellen inte optimal eftersom modellen inte tillåter att olika trådar körs på en heterogen hårdvaruarkitektur eller trådar som kör på en splittrad minnesrymd.

5.2 Vilka problem löser man?

Med multikernelarkitekturen får man ett operativsystem som från grunden är gjort att köras på en dator med många kärnor. Eftersom man använder explicit kommunikation istället för delat minne, kan man lättare köra systemet på mera heterogena datorer. Genom att använda explicit kommunikation behöver man inte använda lås för synkronisering mellan kärnor.

Man reducerar komplexitet i operativsystemskärnan genom att inte behöva lägga lika stor vikt på hur minnessökningar skall göras effektivt. Man kan ha en mycket mera simpel kärna som istället körs i flera instanser på samma dator.

Genom att inte behöva använda lås för synkronisering mellan processorkärnor slipper man många problem som har med skalbarhet att göra. Man behöver inte använda finkornig låsning på samma sätt som i traditionella operativsystem eftersom det inte krävs att flera processorkärnor skall komma åt samma resurser. Med multikernelmodellen har man från början ett distribuerat tillstånd. Alla undersystem är kanske inte optimala att implementera med distribuerat tillstånd, därför har man möjlighet att ge avkall på *designprinciperna*. I multikernelmodellen har man möjlighet använda lås och delat minne om det verkar vettigt. Fördelen mot traditionella operativsystem ur skalbarhetssynvinkel är man måste lägga ner arbete på att få operativsystemet

mindre skalbart, jämfört med traditionella operativsystem där man måste lägga ner arbete på att få dem att vara mera skalbara. [6, 9, 13] Operativsystemens strävan visas i figur 5.2.

I Barrelfish finns en systemkunskapsbas (eng. system knowledge base) tjänst, som innehåller information om den underliggande hårdvaran. Denna information kan användas av såväl applikationer som operativsystemet för att göra beslut om t.ex. hur meddelanden skall skickas mest effektivt och hur minne skall fördelas på fördelaktigaste sätt. [5] I Barrelfish har man gjord en skalbar *TLB* shootdown algorithm, som enligt deras [5] undersökning har större skalbarhetsintervall än andra operativsystem. Man kan också använda system kunskapsbasen för att göra applikationer t.ex. NUMA medvetenhet på ett generellt sätt.

5.3 Slutsatser

Multikernelarkitekturen är en idealisk modell som man kan använda i olika utsträckning, samma forskargrupp som har gjort arkitekturen har också gjort en implementering *Barrelfish*. [5] Implementering har styrt hur arkitekturen ser ut och man har försökt följa design principerna i så stor utsträckning som möjligt. De skriver i artikeln att vissa datorer är optimerade för finkornigt delande (eng. fine-grained sharing), därför kommer man att börja utveckla modeller för att göra beslut om när man ska dela och när man inte ska dela.

Fördelen med multikernelarkitekturen är att den löser många skalbarhetsproblem som finns i traditionella operativsystem. Dock kvarstår faktum att det inte finns en mogen implementering av arkitekturen, det kan också visa sig att det inte är möjligt att göra en vettig implementering. Oberoende av användningsmöjligheterna för denna arkitektur kan den ge en insikt i hur man ska fortsätta utvecklingen av redan existerande operativsystem.

Om man hårddrar multikernelarkitekturen kan man se den som en mikrokernelarkitektur där hela systemet ses som en felbar enhet. I mikrokernelarkitekturen såg man ofta de olika tjänsterna som felbara enheter som kunde startas om när de krashade. Om man ser på multikernelarkitekturen på detta sätt då man bygger ihop flera datorer slipper man den logik som man har i mikrokerneln för att garantera tillförlitlighet.

Arkitekturen verkar ha lösningar på skalbarhetsproblemen som traditionella operativsystem har. Detta är önskvärt eftersom skalbarhetsproblemen i de traditionella operativsystemen ofta är svårlösta. [6, 9, 13]

6 Hårdvara

Kommunikationen mellan processorchip och processorkärnor blir allt viktigare och känsligare. Det är delvis operativsystemens jobb att sköta om att kommunationen görs effektivt. Multikernelarkitekturen försöker lösa detta problem genom att minska på andelen av operativsystemet som är specifik för någon typ av plattform, så att samma operativsystem skall kunna köra på heterogena och asymmetriska hårdvarusystem. Traditionella operativsystem är designade för symmetriska och homogena system och man gör så gott man kan idag för få dessa operativsystem att köra på andra plattformstyper.

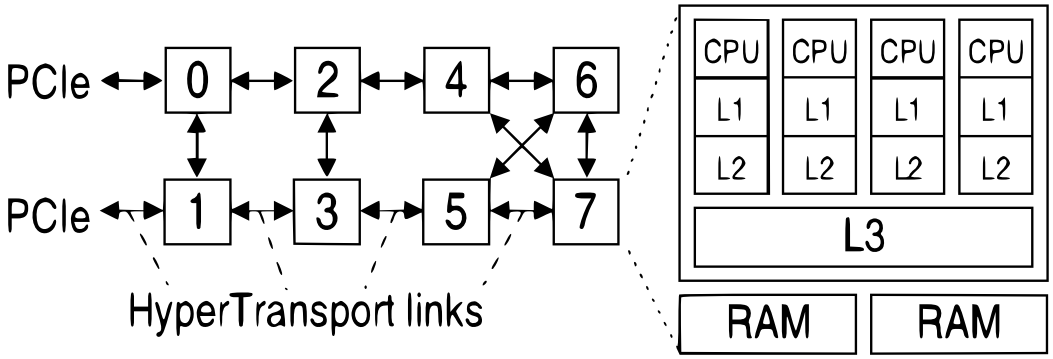
Som nämndes i inledningen så är kravet för att en applikation skall bete sig skalbart att hårdvaran, operativsystemet och applikationen skall vara skalbara. Man kan alltså dra slutsatsen att om inte hårdvaran beter sig skalbart är det ingen mening att försöka få operativsystemet och applikationen att bete sig på ett bättre sätt. Eftersom trenden är att antalet processorkärnor ökar, måste hårdvaruarkitekturen vara skalbar. Om man kopplar alla processorkärnor till ett minne kommer minnesbussen för eller senare att begränsa prestandan.

6.1 NUMA (eng. Non-Uniform Memory Architecture)

På multikärn-arkitekturer används idag icke-likformig minnesåtkomst (eng. Non-Uniform Memory Architecture) för att göra hårdvaran skalbar. [11, 12] De NUMA arkitekturer som är aktuella och mest populära idag, AMDs Opteron och Intels Nehalem är symmetriska och homogena. Idén med NUMA är att varje processorkärna kommer åt olika minnen med olika prestanda, beroende på hur nära de ligger minnet. För att uppnå optimal prestanda på sådana datorer måste både operativsystemet och användarapplikationerna vara medvetna om hur hårdvaruarkitekturen ser ut för att kunna göra bra beslut om på vilken processor vilken process skall köra på, och på vilket minne datastrukturer skall allokeras.

I praktiken är arkitekturer som AMD Opteron och Intels Nehalem cachekoherenta (eng. cache coherent), de har snabba bussar som kopierar förändringar i minnen mellan varandra, så att minnena hålls koherenta. Man kan ha cachekoherens med dagens antal processorkärnor, men i framtiden kan man tänka sig att man måste ge avkall på detta.

Man inser att det är optimalt att ha datastrukturer som en processorkärna vill komma åt på ett minne som ligger så nära processorn som möjligt. Detta blir ett problem när man delar datastrukturer mellan processorkärnor, och bara vissa processorkärnor kan komma åt resurserna på ett optimalt sätt. Man kan då kopiera datastrukturer så att man har flera kopior av samma



Figur 6.1: Nod layout för ett 8x4-processorkärnors AMD system, tagen från[5]

datastruktur på flera minnen som ligger nära de processorkärnor som ska använda minnet.

I multikernelarkitekturen får man automatiskt skilda minnesbilder för varje processorkärna, det är då lätt att sätta en kärninstanss minnessidor i det rätta minnet. [5] Multikernelopeativsystem är alltså, när man använder arkitekturen i sin ideella form, väldigt skalbara på NUMA datorer. Dock bör applikationerna man kör också vara skalbara för att man skall uppnå prestanda.

6.1.1 Operativsystemsstöd för NUMA

I Linux försöker kärnan allokera minne åt en process så att minnet är så nära processorkärnan som möjligt. Det kan dock också vara så att flera processorkärnor som är långt ifrån varandra måste komma åt samma minnessida, detta kan leda till prestanda problem, om man inte gör detta rätt. [14] Man använder också kopiering av datastrukturer i traditionella operativsystem, t.ex. i Linux använder man kopiering av diskcache minnessidor (eng. diskcache). En minnessida kan kopieras när ingen annan processorkärna är intresserad av att skriva till minnessidan i fråga.[13] Om man har minnessidor som många processorkärnor ofta vill skriva till är det inte lönsamt att kopiera dem, eftersom de måste uppdateras ofta. En naturlig nackdel med kopiering är att det konsumerar mera minne.

6.1.2 Fallstudie: AMD Opteron

En översikt över en 8x4-processorkärnors AMD opteron arkitektur finns i figur 6.1. Denna arkitektur är en typisk NUMA, eftersom den har minne kopplat direkt till varje processor, därför kan varje processor kan komma åt sitt eget minne direkt. Detta minskar dramatiskt latensen

och ökar bandbredden. [11]

Alla processorer är hopkopplade till ett nätverk m.h.a. *HyperTransport* bussar. Dessa är nod till nod (eng. point to point) bussar med hög bandbredd och låg latens. [11] AMD Opteron arkitekturen har en gemensam minnesrymd och alla processorkärnor kan komma åt allt minne. När en processorkärna vill komma åt ett minne på en annan nod använder den sig av *HyperTransport*-bussarna. Den blir således en längre fördröjning när en processorkärna skall komma åt ett avlägset minne. Fördröjningen och bandbredden till avlägsna minnen beror dessutom på systemets struktur och hela systemets totala belastning.

AMD Opteron använder också *HyperTransport*-bussarna för att upprätthålla cache koherens. Därför används HyperTransport-bussarna i Barrelfish operativsystemet till att skicka meddelanden mellan noder, internt i en nod används cache minnen som förmedlare av meddelanden. [5]

7 Diskussion

När man vet hur framtidens datorer ska se ut, och om man har insikten om att dagens operativsystem inte kommer att klara av alla krav, måste man ha ett nytt operativsystem som klarar av allt som de nya datorerna kräver.

Multikernelmodellen verkar ändå som en lovande framtidsutsikt, det är ännu inte uteslutet att man kunde bygga en Linux kärna som skulle stöda multikernelmodellen. Traditionella operativsystem verkar klara av system ett litet antal processorkärnor n, så en idé kunde vara att implementera designprinciperna (avsnitt 5.1) i kerneln och sedan köra $\frac{m}{n}$ kernelinstanser på en homogen dator med m processorkärnor.

Traditionella operativsystem tror jag att har möjlighet att klara av att köras på *homogena asymmetriska* hårdvaruarkitekturer, så länge minnesbussarna klarar av cachekoherens. Idag finns små effektiva processorkärnor (ARM) och effektiva stora processorkärnor (AMD, Intel). För att kunna göra en effektiv *homogen asymmetrisk* hårdvaruarkitektur måste man göra en *grundkärnekvivalent* (se avsnitt 2.1 på sidan 3), som är tillräckligt skalbar för att kunna köras som en stor processor och som en liten processor. På samma gång måste man tampas med problemen med minnesbandbredden och komplexiteten som diskuteras i [2].

Skulle man göra en fungerande platform som skulle stöda multikernelmodellen skulle tröskeln till att använda *heterogena* hårdvaruarkitekturer minska. Det skulle också bli mindre tröskel till att börja använda hårdvaruarkitekturer med icke-gemensam minnesrymd. Man skulle då kunna bygga ihop datorer av idag existerande komponenter, som skulle vara väldigt skalbara. Man bör dock inte begå samma misstag som gjordes när mikrokernelarkitekturen utvecklades, då man trodde att allt som teorin förutspådde kunde realiseras i praktiken.

Referenser

[1] Scalability - The Linux Inforation Project
http://www.linfo.org/scalable.html
Hämtad 15.2.2011

[2] Sherkar Borkar, Thousand Core Chips - A Technology Perspective, In Pro- ceedings of the 44th Annual Design Automation Conference, pages 746–749, 2007.

[3] G.M. Amdahl, Validity of the single-processor approach to achieving large scale computing capabilities. In AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.

[4] Andi Kleen, Linux multi-core scalability. Linux Kongress 2009, Dresden, 2009.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In Proceedings of the 22nd ACM Symposium on OS Principles, Big Sky, MT, USA, October 2009.

[6] M.Russinovich, Inside Windows 7, Microsoft MSDN Channel9, January 2009

[7] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russel, Dipankar Sarma, Maneesh Soni. Read-Copy Update, Ottawa Linux Symposium

[8] Paul E. McKenney, RCU vs. Locking Performance on Different CPUs. Adelaide, Australia, 2004.

[9] Linus Torvalds, Subject: Linux 2.6.38-rc1, Newgroup: gmane.linux.kernel
http://thread.gmane.org/gmane.linux.kernel/1088915
Hämtad 25.2.2011

[10] Larry McVoc, SMP Scaling Considered Harmful. BitMover Inc., San Fransisco, 1999

[11] Davis Jessel, AMD Opteron Processors: A Better High-End Embedded Solution. AMD Boston Design Center, Boston, March 2005

[12] Intel Corporation, An Introduction to the Intel QuickPath Interconnect,

[13] Nick Piggin, NUMA replicated pagecache for Linux, SuSE Labs., January 2008

[14] Christoph Lameter, Local and Remote Memory: Memory in a Linux/NUMA System, Silicon Graphics Inc. June 2006

[15] Mark D. Hill, Michael R. Marty. Amdahl's Law in the Multicore Era

[16] What is ZFS?
http://hub.opensolaris.org/bin/view/Community+Group+zfs/whatis
Hämtad 2.4.2011