CHALMERS

Processchemaläggare för mångkärniga processorer

– Fördelning av minnesbelastning i NUMA-system

DANIEL EDHOLM
VIKTOR NILSSON
ANDERS LÖFGREN

Institutionen för Data- och informationsteknik

CHALMERS TEKNISKA HÖGSKOLA

Göteborg, Sverige 2013

Kandidatarbete nr 2013:22

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Processchemaläggare för mångkärniga processorer
Fördelning av minnesbelastning i NUMA-system

DANIEL EDHOLM
VIKTOR NILSSON
ANDERS LÖFGREN

© DANIEL EDHOLM, June 2013.
© VIKTOR NILSSON, June 2013.
© ANDERS LÖFGREN, June 2013.

Examiner: ARNE LINDE

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2013

Abstract

For systems with multicore processors contention for shared resources is a problem that occurs when several memory-intensive processes are executed in parallel within the same memory domain. This contention has a direct influence on the performance of the system and is a complex problem that has been recognized for a long time. An attractive and actively studied way to minimize this problem is by using a process scheduler adapted to allocate processor cores in a way such that contention for shared resources is minimized.

With the introduction of multicore NUMA-systems (Non-Uniform Memory Access) the situation has become even more complex. In these systems the access time for processor cores to different memory domains vary depending on factors such as distance and load. Thus, the process scheduler also has to consider where the memory of each process is placed to minimize the distance and balance the load on each memory domain.

This report presents a user-level process scheduler for a NUMA-system based on the multicore processor Tilera TILE*Pro*64. The scheduler considers the load on each memory domain to guide the allocation of processor cores. The purpose is to spread the executing processes and thereby minimize memory access time.

The results show that the scheduler achieves a small performance gain regarding execution time for several workloads. On the basis of these results the report discusses properties of the system and potential limitations in both the development environment and our implementation.

Sammanfattning

I system med mångkärniga processorer är tävlan om tillgång till delade resurser ett problem som uppstår när flera minnesintensiva processer exekveras parallellt i samma minnesdomän. Denna tävlan har en direkt påverkan på systemens prestanda och är ett komplicerat problem känt sedan långt tillbaka. Ett attraktivt och aktivt studerat sätt att minimera problemet är att använda en processchemaläggare anpassad för att styra allokeringen av processorkärnor så att tävlan om delade resurser minimeras.

Med introduktionen av mångkärniga NUMA-system (Non-Uniform Memory Access) har situationen komplicerats ytterligare. I dessa system varierar åtkomsttiden för processorkärnor till olika minnesdomäner beroende av faktorer så som avstånd och belastning. Processchemaläggaren behöver därför också ta hänsyn till vart minnet för varje process är placerat för att minimera avståndet och för att balansera belastningen på varje minnesdomän.

Denna rapport presenterar en processchemaläggare på användarnivå för ett NUMA-system baserat på den mångkärniga processorn Tilera TILE*Pro*64. Schemaläggaren utgår från belastningen på varje minnesdomän för att styra allokeringen av processorkärnor. Syftet är att fördela exekverande processer och därmed minimera åtkomsttiden.

Resultaten visar att schemaläggaren lyckas åstadkomma en mindre prestandavinst sett till exekveringstid för ett flertal arbetsscheman. Utifrån dessa resultat diskuteras systemets egenskaper och potentiella begränsningar i utvecklingsmiljön och vår implementation.

Ordlista

CFS - Completely Fair Scheduler. Den schemaläggare som används som standard i Linux sedan version 2.6.23. CFS är implementerad på systemnivå i operativsystemet.

DFS – Benämningen på processchemaläggaren utvecklad under projektet.

LLC - Last Level Cache. Sista nivån i processorns cache-system.

Mesh-nätverk - Nätverk där alla noder har åtkomst till och agerar relä åt varandra. Data som sänds propagerar genom andra noder för att nå sitt mål.

NUMA - Non-Uniform Memory Access. En systemarkitektur med multipla minnesdomäner där åtkomsttiden till de olika minnesdomänerna varierar mellan processorkärnorna.

Overhead - Extra användning av resurser som krävs för att utföra en uppgift.

PMC - Performance Monitor Counter. Specialregister inbyggt i processorer för att räkna olika hårdvarubaserade händelser.

Tilera - Företaget som utvecklat processorn TILE*Pro*64.

Tile - En Tilera-specifik benämning på en enskild processorkärna på TILE*Pro*64.

MDE - Multicore Development Environment. Namnet på den utvecklingsmiljö som Tilera tillhandahåller för processorer med deras egen arkitektur, däribland TILE*Pro*64.

TMC - Tilera Multicore Components. Ett programbibliotek med rutiner för att förenkla utvecklingen av parallelliserade applikationer till processorer med Tileras egen arkitektur.

VLIW - Very Long Instruction Word. En processorarkitektur designad för att kunna utnyttja parallell exekvering av instruktioner. Gör det möjligt att slå samman flera maskininstruktioner i ett instruktionsord.

Innehållsförteckning

1 Inledning

Processorn är den centrala beräkningsenheten i dagens datorer. Dess uppgift är att hämta och utföra maskininstruktioner för de program som körs på datorn. Prestandan hos en dator utmärks av hur snabbt processorn kan genomföra dessa operationer. En processors prestanda beror på flera faktorer och genom åren har höjda klockfrekvenser och förbättrade arkitekturer varit de främst bidragande faktorerna. Marknaden dominerades länge av enkelkärniga processorer. Sedan början av 2000-talet har detta förändrats då hög strömförbrukning och värmeutveckling orsakade stora problem. För att fortsätta förbättringen av beräkningskapaciteten hos processorerna började mångkärniga processorer tillverkas i allt högre grad. Ett problem med mångkärniga processorer är att prestandan vanligtvis inte är linjärt skalbar med antalet kärnor. Anledningen är delvis att processorkärnorna delar systemets övriga resurser, såsom minne och bussar, och tävlan om tillgången till dessa orsakar flaskhalsar. En viktig uppgift för systemets processchemaläggare är därför att planera exekveringen av processer så att systemets resurser fördelas på ett effektivt sätt. Denna rapport behandlar utvecklingen av en processchemaläggare för ett NUMA-system (Non-Uniform Memory Access). Schemaläggaren utgår från belastningen på systemets primärminnen för schemaläggning av processer.

1.1 Bakgrund

Beroende på egenskaperna hos NUMA-system ställs särskilda krav på processchemaläggaren. I detta kapitel ges en kortare genomgång av schemaläggning i allmänhet och hur den kan anpassas för att bättre hantera egenskaperna hos dessa system.

1.1.1 Schemaläggning

Schemaläggaren är en viktig komponent i moderna operativsystem. Den gör det möjligt att exekvera flera processer parallellt på ett system där processerna delar åtkomst till systemets resurser, så som processor och minne. Generellt brukar de flesta operativsystem använda sig av tre olika typer av schemaläggare: en lång-, en mellanlång- och en kortsiktig. Den långsiktiga används när ett program ska starta. Den har då hand om att avgöra om processen kan starta omgående eller om uppstarten ska fördröjas till ett senare till tillfälle. Genom dessa avgöranden är det den långsiktiga som kontrollerar hur många processer som ska ges tillgång till systemets resurser samtidigt och den styr därmed graden av parallell exekvering som systemet tillåter. Den mellanlångsiktiga används för att vid behov frigöra minne från processer som varit inaktiva under en längre tid. Dessa processer flyttas då från processorns primärminne till sekundärminnet. När sedan de utflyttade processerna ska aktiveras igen flyttar schemaläggaren tillbaka dem till primärminnet så att de kan återuppta exekveringen. Den kortsiktiga, även kallad processchemaläggaren, har till uppgift att bestämma vilken process som ska ges tillgång till processorn för exekvering. Denna typ av schemaläggning sker efter korta, regelbundna klockintervall eller vid exekvering av exempelvis systemanrop eller I/O-operationer. Den kortsiktiga används mer frekvent av systemet än de övriga två typerna och därmed finns ett ökat krav på att exekveringstiden för den ska vara kort.

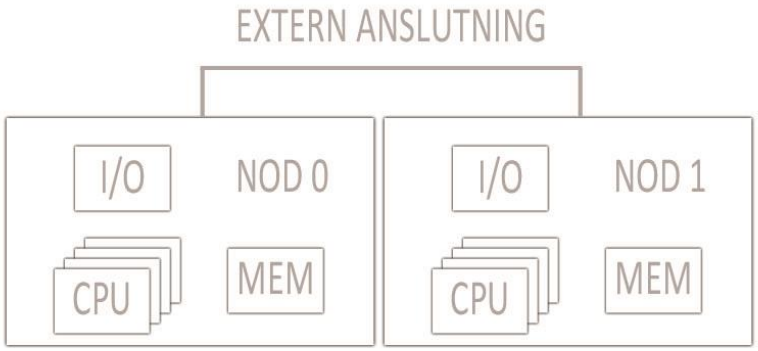
Varje processchemaläggare styrs av en algoritm som definierar hur den arbetar. Exempel på förekommande algoritmer är first in first out, round robin och shortest job first. De olika algoritmerna har olika egenskaper och valet av algoritm beror på vilka egenskaper som är önskvärda. Vanliga egenskaper att ta hänsyn till är genomströmning, svarstid samt rättvisa. Genomströmning är mängden processer som blir avklarade per tidsenhet. Svarstid är den tid som en process får vänta från dess att den är redo till dess att schemaläggaren beviljar processen exekveringstid. Rättvisa innebär att alla processer ska ges lika möjlighet att utnyttja processorn. En schemaläggare som inte är rättvis kan orsaka problem med utsvältning. Detta innebär att en eller flera processer aldrig får

tillgång till processorn och får därmed ingen möjlighet att exekvera. Det är ofta önskvärt att en schemaläggare kan prestera bra på alla dessa punkter, men det är i praktiken svårt då optimering av en egenskap ofta står i konflikt med en annan [1].

Schemaläggningen av processer kan göras på antingen systemnivå eller användarnivå. Vanligtvis är processchemaläggaren implementerad på systemnivå som en integrerad komponent i operativsystemet. På systemnivå har schemaläggaren full tillgång till hela systemet och kan på ett detaljerat sätt styra allokeringen av processorer till de program som körs. För situationer där en specialiserad schemaläggning är önskvärd är det dock vanligt att schemaläggaren implementeras på användarnivå. Här har den inte lika stor tillgång till systemet och används i kombination med operativsystemets egen schemaläggare för att åstadkomma en schemaläggning med önskvärda egenskaper.

1.1.2 NUMA-system

I de flesta datorsystem är åtkomsttiden till primärminnet en betydande faktor för systemets prestanda. Detta beror på att processorn oftast arbetar betydligt snabbare än minnet och får därför spendera mycket tid med att vänta på att datan som processorn arbetar med ska bli tillgänglig. Detta problem blir än mer tydligt i system med flera processorkärnor, eftersom deras potentiella beräkningskapacitet är ännu större. För att prestandan i dessa system inte ska bli lidande av brist på tillgänglig data krävs hög minnesprestanda. Ett vanligt förekommande sätt att åstadkomma detta är att dela upp systemet i flera noder där varje nod består av en delmängd av systemets processorer och en egen minnesdomän. Systemets totala minnesutrymme utgörs av den sammanlagda minnesmängden på alla noder. Dessa system kallas för NUMA-system (Non-Uniform Memory Access). Figur 1.1 visar en schematisk bild över hur ett typiskt NUMA-system är strukturerat. För processorkärnorna i de olika noderna varierar åtkomsttiden till de olika minnesdomänerna beroende kärnornas placering. Varje processor har snabbare åtkomst till den minnesdomän som ligger lokalt inom noden än de minnesdomäner som tillhör avskilda noder.



Figur 1.1 – Strukturen hos ett typiskt NUMA-system.

I dagens Linux-kärna används schemaläggaren CFS (Completely Fair Scheduler) som standard (se Appendix A - Completely Fair Scheduler). I ett NUMA-system bör dock CFS inte ge optimal prestanda, eftersom den inte tar hänsyn till processers minnesallokering vid schemaläggning. Då åtkomsttiden till minnet i ett NUMA-system varierar för de olika noderna, bör en schemaläggare för ett NUMA-system styra utplaceringen av processer så att varje process exekveras på en nod där de har allt eller mestadels av sitt minne allokerat.

1.2 Problemställning

Att implementera en schemaläggare på systemnivå är en komplex uppgift som kräver mycket goda kunskaper om operativsystemets interna struktur. Det är inte vanligt att operativsystem har en processchemaläggare som är anpassad för NUMA-system, men många erbjuder funktionalitet för att utveckla applikationer som är anpassade för den här typen av system. Genom att implementera en schemaläggare på användarnivå går det att kringgå mycket av den här komplexiteten.

En möjlighet som uppstår i NUMA-system är att schemaläggaren kan ta hänsyn till var olika processer har sitt minne allokerat. På så sätt kan den styra hur exekveringen ska fördelas så att processerna får en så effektiv åtkomst till minnet som möjligt. Effektiv användning av minnet kan vara en kritisk del för processorns prestanda. Därför skulle en processchemaläggare anpassad för NUMA-system kunna bidra till minimerad exekveringstid.

1.3 Syfte

Syftet med projektet är att skriva en processchemaläggare på användarnivå, som är anpassad för ett NUMA-system baserat på en mångkärnig processor. Målet med schemaläggaren är att den ska fördela processorns resurser på ett sådant sätt att minnesåtkomsttiden för de exekverande processerna minimeras och användningen av minnet blir så effektiv som möjligt. Projektet kommer också undersöka om den slutgiltiga processchemaläggaren ger en minskad exekveringstid för utvalda testprogram.

1.4 Avgränsningar

Processchemaläggaren, som är projektets slutprodukt, kommer att anpassas för processorn Tilera TILE*Pro*64 (se kapitel 2.1.1). Implementationen ska vara sådan att den kan köras på användarnivå i operativsystemet Linux. Ingen hänsyn kommer att tas till portabilitet, det vill säga att anpassa schemaläggaren till att fungera på andra system. Applikationerna som används för prestandatestning ska inte modifieras på något sätt och schemaläggaren ska inte göra några speciella antaganden om applikationerna för att minska exekveringstiden. Testapplikationerna kommer även att begränsas till entrådiga program som inte använder delat minne. Flertrådade applikationer skulle öka komplexiteten hos schemaläggaren och har därför utelämnats.

1.5 Metod

Projektarbetet delades in i tre huvudsakliga moment; litteraturstudie, testning samt ett design- och implementationsmoment. Då projektet är av undersökande natur användes en vetenskaplig metodik. Detta innebar att arbetet utfördes iterativt på ett sådant sätt att fokus pendlade mellan de olika momenten utifrån observerade resultat, diskussion och vidare studier. En projektplanering med en strategi för utförandet, tidfördelning, mål och avgränsningar lades i inledningen av projektet för att strukturera arbetet.

Den tekniska utrustning (se kapitel 2) som användes var för gruppen delvis okänd sedan tidigare och därför startade arbetet med en grundläggande studie av datorsystemets dokumentation. En mer utförlig och detaljerad studie av dokumentationen utfördes även löpande under hela projektets gång. Den teoretiska basen för arbetet inhämtades genom studerande av relaterade arbeten. Dessa omfattades av liknande implementationer samt teoretiskt material om schemaläggning för datorsystem med mångkärniga processorer i allmänhet och även specifikt för NUMA-system.

Under design- och implementationsmomentet fördes en aktiv diskussion kring vilka val som var mest intressanta utifrån de resultat som observerades och de resonemang som presenterades i relaterade

arbeten. Genom att använda versionshanteringssystemet GitHub [2] kunde samarbetet underlättas under utvecklingen.

För att undersöka om den implementerade schemaläggaren lyckades minska exekveringstiden för utvalda testprogram utfördes jämförande prestandatester. Testerna genomfördes så att en slumpmässigt genererad arbetsbelastning exekverades en gång med projektets schemaläggare och en gång utan. För varje test uppmättes den totala exekveringstiden och efteråt jämfördes dessa värden.

2 Tekniköversikt

Under utvecklingen av schemaläggaren har både specialiserad hårdvara och ett antal olika mjukvaruverktyg använts. För mätning av exekveringstid användes även ett antal testapplikationer. I detta kapitel ges en beskrivning av maskinvaran, utvecklingsmiljön och testapplikationerna som använts under projektets gång.

2.1 Hårdvara

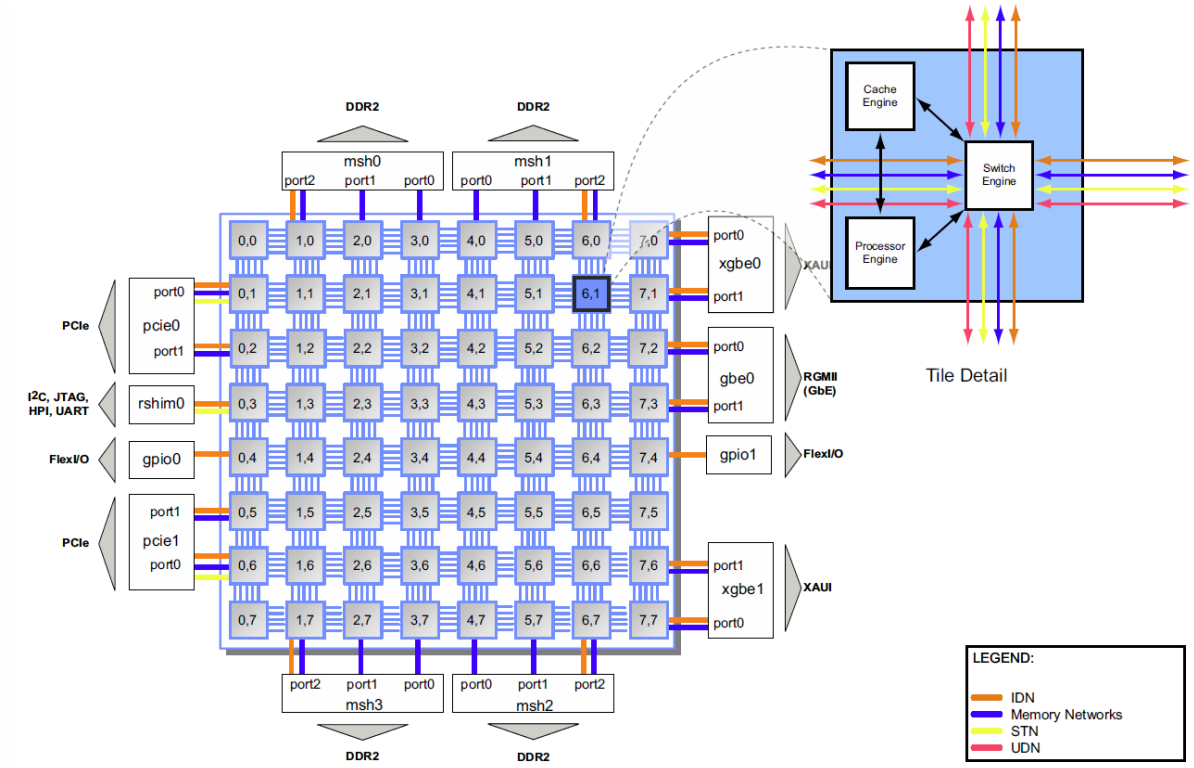
Hårdvaran som använts bestod av att komplett NUMA-system baserat på en mångkärnig processor. Detta avsnitt tar upp den interna strukturen och dess NUMA-egenskaper.

2.1.1 Processorn Tilera TILE*Pro*64

TILE*Pro*64 är en processor från företaget Tilera byggd på en proprietär arkitektur specialiserad för god skalbarhet med många processorkärnor [3]. Processorn är bestyckad med 64 kärnor (av Tilera kallade tiles) och använder en egen instruktionsuppsättning. Samtliga kärnor kan dock inte användas för exekvering av applikationer eftersom systemet automatiskt allokerar vissa kärnor för att nyttjas till utförandet av systemfunktioner och I/O-operationer.

Varje kärna har tre separata femstegs exekveringsenheter, och arbetar med 64 bitar långa instruktionsord. Processorn använder en VLIW-arkitektur (Very Long Instruction Word) vilket gör det möjligt att utnyttja parallell exekvering av instruktioner. Detta utförs genom att kompilatorn slår ihop upp till tre maskininstruktioner i varje instruktionsord. Vid kompilering kontrollerar kompilatorn om det finns några beroenden mellan dessa instruktioner, och om det inte gör det adresseras exekveringsenheterna i processorn separat och instruktionerna exekveras parallellt. Eftersom tekniken förlitar sig på att kompilatorn utför en statisk schemaläggning av instruktioner undviker den komplexiteten i att utföra dynamisk schemaläggning av instruktioner i hårdvaran. Varje kärna har en separat L1-cache för data och instruktioner samt en kombinerad L2-cache. Utöver cache-minnet på varje kärna utnyttjar processorn också en dynamiskt distribuerad L3-cache bestående av de samlade L2-cache-minnena hos samtliga kärnor [4]. L3-cachen använder sig av funktionen hash-for-home. Detta innebär att L2-cachen i varje kärna är ansvarig för en del av minnet. När en process begär data från minnet frågar den först minnesdelens ansvariga kärna om den har datan i sin cache. Om den ansvariga kärnan inte har det, går begäran till minnet som vanligt. På varje kärna finns fyra specialregister för räkning av olika hårdvaruhändelser, även kallt PMC (Performance Monitor Counter) [5]. Dessa register nyttjas genom att de initieras till att räkna de händelser som är av intresse under exekvering. Initieringen sker genom skrivning till speciella kontrollregister. Till exempel kan dessa register initieras till att räkna antalet klockcykler som passerat, antalet instruktioner kärnan har utfört eller antalet minnesreferenser som missat i olika delar av cache-systemet.

Internt är processorkärnorna placerade i en 8x8-matris (se Figur 2.1). Alla kärnor är sammankopplade med interna anslutningar på processorchippet och tillsammans utgör anslutningarna sex separata nätverk som tillverkaren kollektivt benämner iMesh. Dessa nätverk används bland annat för höghastighetskommunikation, dataöverföring och andra systemrelaterade operationer mellan kärnorna. De är av typen mesh-nätverk, en sorts nätverk utan en central dataväxel. Detta innebär att kärnorna inte bara hanterar egen data utan även ansvarar för att vidarebefordra data som skickas mellan övriga kärnor på nätverket. För hanteringen av nätverkstrafiken har varje kärna en direktkoppling till angränsande kärnorna. En mer utförlig beskrivning av iMesh och dess funktionalitet återfinns i Appendix B.



Figur 2.1 – TILEPro64 [Använd med tillstånd].

Rent fysiskt sitter processorn på ett PCI-Express-kort, TILEncore, och tillsammans med andra komponenter på kortet utgör det ett komplett datorsystem [6]. Kortet kommer framöver att benämnas som måldatorn.

2.1.2 NUMA-egenskaper

Som synes i Figur 2.1 har TILE*Pro*64 fyra integrerade minnesstyrenheter. På mjukvarunivå fungerar trots detta inte systemet som ett NUMA-system i dess ursprungskonfiguration. Detta beror bland annat på en teknik som kallas för striped memory. Tekniken arbetar på ett sådant sätt att när en process allokerar en mängd minne delas denna mängd upp i 8 kilobyte stora delar. Dessa sprids ut jämnt över de fyra minnesdomänerna. Striped memory används för att minnesallokeringen i systemet ska bli balanserad även då applikationerna som körs inte utför minnesbalansering på egen hand. I denna konfiguration är det endast en minnesdomän, med den sammanlagda storleken av alla fyra, som presenteras för måldatorns operativsystem. Systemet kan dock konfigureras så att det presenteras som ett NUMA-system. Förenklat delas då processorkärnorna in i fyra logiska kvadranter. Varje kvadrant utgör därmed en nod i NUMA-systemet och består av maximalt 16 kärnor och en minnesstyrenhet. I praktiken ingår dock ett fåtal kärnor i en annan nod än dess logiska kvadrant. För exekverande processer kommer minne allokeras på den minnesstyrenhet som tillhör den nod där processen exekverar. Skulle minnesutrymmet hos en nod bli fullt kommer systemet att allokera minne på en annan nod.

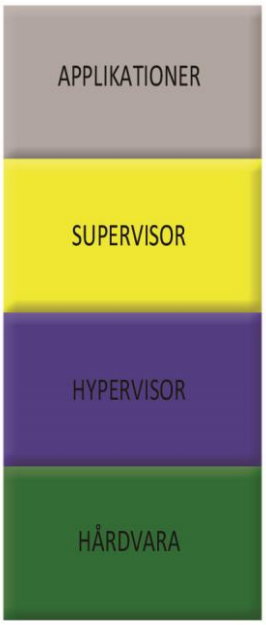
2.2 Utvecklingsmiljö

För utveckling av mjukvara till måldatorsystemet finns Tilera MDE (Multicore Development Environment). Detta är en utvecklingsmiljö som utgörs av ett komplett Linux-baserat operativsystem tillsammans med ett antal mjukvaruverktyg. Dessa verktyg är installerade på värddatorn. En huvudkomponent i Tilera MDE är `tile-eclipse`, vilket är ett integrerat utvecklingsverktyg

baserat på den öppna mjukvaran Eclipse [7]. Utvecklingsmiljön är anpassad för att underlätta utveckling av applikationer för den tydligt parallelliserade arkitekturen hos målsystemet.

2.2.1 Operativsystem

I utvecklingsmiljön ingår Tile Linux, som är det operativsystem som körs på måldatorn. Detta operativsystem bygger på Linux version 2.6.26, men har porterats och anpassats för att användas på måldatorn. Figur 2.2 visar hur mjukvarustacken ser ut på måldatorn. Överst körs applikationer på användarnivå och på nivån under kör måldatorns operativsystem (supervisor). På varje processorkärna körs en hypervisor, vars uppgift är att abstrahera vissa hårdvaruspecifika detaljer för operativsystemet. Detta inkluderar bland annat kommunikationen mellan de olika kärnorna, I/O-utdataoperationer samt att på en låg nivå förse systemet med ett virtuellt minnessystem.



Figur 2.2 – Mjukvarustacken på måldatorsystemet.

2.2.2 Programspråk

För utveckling av applikationer till måldatorn kan något av programspråken C eller C++ användas. I utvecklingsmiljön ingår båda programspråkens standardbibliotek. Utöver dess finns även biblioteket TMC (Tilera Multicore Components). Detta bibliotek innehåller funktioner som ska underlätta utvecklingen av parallella program till Tileras processorer [8].

2.2.3 Korskompilering

TILE*Pro*64 är baserad på en egen arkitektur och instruktionsuppsättning och därför används korskompilering för att kompilera program som ska köras på processorn. Detta innebär att kompileringen utförs med hjälp av en kompilator på ett utomstående system som inte är baserat på samma arkitektur. För att utföra korskompilering av program skrivna i antingen C eller C++ innehåller utvecklingsmiljön verktyget `tile-cc`, som närmast kan liknas vid kompileringsverktyget GCC (GNU Compiler Collection). `tile-cc` skapar exekverbar programkod och innehåller förutom själva korskompilatorn även en assemblerare och en länkare.

2.2.4 Exekvering och felsökning

När den färdiga programkoden ska exekveras måste den överföras till måldatorn. För att utföra detta används verktyget `tile-monitor`, som också följer med utvecklingsmiljön. Detta verktyg har möjlighet att ladda upp och exekvera applikationen direkt på måldatorn, alternativt kan applikationen exekveras i en simulator. Att använda simulatorn istället för hårdvaran underlättar vid felsökning av applikationen, eftersom simuleringen möjliggör felsökning på källkodsnivå. Simulatorn kan också presentera detaljerad information om olika prestandaparametrar, så som antal exekverade instruktioner och information om utförda minnesreferenser.

2.2.5 Profilering

Profilering innebär att undersöka egenskaper hos en applikation. Exempelvis kan profilering påvisa om en applikation är starkt minnes- eller processorberoende, hur mycket minne applikationen använder eller hur lång tid applikationen tar att exekvera. I Tilera MDE finns två huvudsakliga verktyg för att utföra profilering, `OProfile` och `mcstat`.

`OProfile` är baserat på hårdvaruhändelser. Profileringen startas genom att `tile-monitor` används för att initiera verktyget med de händelser som är av intresse och sedan körs applikationerna som ska profileras. Verktyget använder sig av PMC-registerna i processorkärnorna för att räkna angivna hårdvaruhändelser och när profileringen avslutas sammanställs ett resultat över dessa.

Det andra verktyget, `mcstat`, används för att få statistik över hur mycket varje minnesstyrenhet i systemet används. För att profilera en applikation startas verktyget med programmet som parameter. När profileringen är avslutad sammanställs en tabell med statistik över minnesanvändningen. `mcstat` baseras på en drivrutin i operativsystemet, `memprof`, som kontinuerligt avläser hur många operationer varje minnesstyrenhet har utfört. Drivrutinen använder PMC-register placerade i varje minnesstyrenhet och datan från dessa register sammanställs i en fil. Denna fil återfinns i ett virtuellt filsystem monterat i katalogen `/proc` och är därför tillgänglig för applikationer på användarnivå.

De två profileringsverktygen används för att utföra offline-profilering, vilket innebär att applikationen som ska profileras körs en gång och när körningen avslutas sammanställs resultatet. Eftersom `memprof` uppdaterar datan kontinuerligt och gör denna tillgänglig via filsystemet, kan drivrutinen användas för att göra profilering online. Det innebär att profileringsdata avläses under tiden som applikationen exekveras.

2.3 Testmjukvara

För testning av schemaläggaren användes ett antal applikationer ur testsviten SPEC CPU2006 (Standard Performance Evaluation Corporation CPU2006) [9] och cBench v1.1 (Collective Benchmark v1.1) [10]. Dessa testapplikationer utför simuleringar och beräkningar baserade på verkliga situationer och används ofta som referens för prestandamätning av datorsystem. Tabell 2.1 visar vilka applikationer som har använts.

Tabell 2.1 - Använda testapplikationer.

SPEC CPU2006	cBench v1.1
429.mcf	automotive/bitcount
433.milc	consumer/jpeg_c
450.soplex	consumer/tiff2bw
470.lbm	consumer/tiff2rgba
	network/dijkstra
	network/patricia

3 Litteraturstudier

Då projekt hade en tydligt undersökande inriktning låg litteraturstudier till grund för teorier, diskussion och arbetet med att implementera processchemaläggaren. Den litteratur som studerades kan delas in i två huvudsakliga kategorier, teknisk dokumentation och relaterade arbeten. I detta kapitel ges en sammanfattning av de viktigaste delarna ur litteraturstudien.

3.1 Teknisk dokumentation

Den tekniska dokumentationen till datorsystemet som använts består av en samling dokument som alla behandlar olika delar av systemet. I dessa ges en utförlig beskrivning av hårdvarans struktur och egenskaper, så som processorarkitektur, registeruppsättning, minnessystem, gränssnitt för externa enheter och det interna mesh-nätverk som används för kommunikation mellan olika delar av processorn. En genomgående beskrivning görs också av den tillhörande utvecklingsmiljön. I denna beskrivs operativsystemet som körs på måldatorn samt många av mjukvaruverktygen som inkluderats för att möjliggöra utveckling, exekvering och undersökning av applikationer till systemet. Dokumentationen innehåller också flera avsnitt med råd och hjälp till programmerare för att underlätta utvecklingen av applikationer som effektivt drar nytta av systemets olika funktioner.

3.2 Relaterade arbeten

Koita et al. [11] fastslår i en studie från början av 2000-talet att processchemaläggning för NUMA-system med flera processorer är ett aktivt studerat ämne. De visar också att tidigare arbete inom ämnet i synnerhet handlat om att schemalägga processer på ett sådant sätt att tävlan om processorernas LLC (Last Level Cache) blir så liten som möjligt. Studien ämnar påvisa att detta inte är tillräckligt för att få ut maximal prestanda i NUMA-system där även primärminnets prestanda är en inflytelserik faktor. Studien introducerar sedan ett antal olika scheman för både processorallokering och minnesallokering. En typ av scheman som introduceras innebär att varje process tilldelas ett hemkluster. Detta kluster används i schemaläggningen som utgångspunkt för hur processorer och minne skall allokeras. Genom att använda ett simulerat NUMA-system tillsammans med dessa olika scheman drar författarna slutsatsen att en optimal schemaläggning för denna typ av system ska ta hänsyn till både processorallokering och minnesallokering samt utföras med varje process hemkluster som utgångspunkt.

Zhuravlev et al. [12] påpekar att anpassad schemaläggning är ett mycket attraktivt sätt att motverka tävling om minnesresurser i system med mångkärniga processorer. De påpekar också att operativsystemets (Linux) schemaläggare huvudsakligen utför lastbalansering utan hänsyn till problemet med att fördela minnesresurser på ett effektivt sätt. I en processchemaläggare med målet att minimera tävlan om LLC, används processklassificering för att besluta om den mest optimala allokeringen av processorkärnor. Författarna analyserar olika metoder för att klassificera processer genom att mäta hur effektivt varje klassificeringsmetod bidrar till att schemaläggaren kan motverka tävlan om LLC. Av analysen dras slutsatsen att den mest effektiva metoden är klassificering efter processernas missfrekvens i LLC. Författarna undersöker också bidragande orsaker till försämrad prestanda i system med mångkärniga processorer. De finner att tävlan om minnesbussar och styrenheter utgör det största problemet. Utifrån detta stärker författarna sitt argument för att den mest effektiva metoden för processklassificering är att beräkna missfrekvensen i LLC, eftersom denna står i korrelation till tävlan om minnessystemets resurser. För att påvisa sitt resultat introduceras två algoritmer, DI (Distributed Intensity) och DIO (Distributed Intensity Online). Den huvudsakliga skillnaden mellan dessa är att den första använder sig av missfrekvenser uträknade i förväg och den andra använder dynamiska missfrekvenser som räknas ut under körning. DIO använder hårdvaruimplementerade prestandaräknare för att räkna ut aktuella missfrekvenser.

Majo et al. [13] argumenterar för att effektiv schemaläggning på NUMA-system måste ta hänsyn till två huvudsakliga aspekter. Det första är tävlan om LLC och det andra är processernas datalokalitet. Studien visar att om schemaläggningen görs med endast hänsyn till datalokalitet kan prestandan bli lidande eftersom strävan för hög datalokalitet potentiellt kan introducera ökad tävlan om LLC. Enligt författarna är därför schemaläggning där en kompromiss görs mellan de olika egenskaperna det bästa alternativet. För att påvisa sitt resonemang uppvisas algoritmen N-MASS (NUMA-Multicore Aware Scheduling Scheme). Algoritmen bygger på tre steg för att räkna ut en optimal processorallokering. Första steget sorterar alla processer i minskande ordning efter hur mycket de påverkas av datalokalitet. Processer som är minnesbundna är mer känsliga för datalokalitet än de som är processorbundna. I andra steget beräknas en processorallokering med så hög datalokalitet som möjligt. Om denna allokering leder till ökad tävlan om LLC körs ett tredje steg med målet att förfina processorallokeringen så att en kompromiss mellan datalokalitet och LLC-tävlan uppnås. I detta steg undersöker algoritmen om det är fördelaktigt att flytta en process till en annan nod för att minska LLC-tävlan eller om det är mer fördelaktigt att bibehålla god datalokalitet. I detta avgörande använder algoritmen måttet på hur känslig varje process är för datalokalitet. N-MASS tar ingen hänsyn till hur situationen skulle förändrats om möjligheten funnits att även migrera processers minne. Detta eftersom det enligt författarna är svårt att förutse kostnaden för en sådan migrering samt att möjligheten att migrera minnet beror på om det finns tillräckligt med ledigt utrymme på den nod dit processen flyttas.

Zhuravlev et al. [14] utför en noggrann analys av hur väl existerande algoritmer för hantering av tävlan om systemresurser fungerar på NUMA-system. Resultatet pekar på att algoritmerna misslyckas med att förbättra prestandan i dessa system jämfört med standardschemaläggaren i operativsystemet (Linux, CFS). Författarna slår fast att algoritmerna till och med har en negativ inverkan på prestanda. Därefter presenteras en utförlig undersökning av vilka faktorer som har mest inverkan på prestandan i NUMA-system. De allra största faktorerna visas vara tävlan om LLC, minnesstyrenheter och interna sammanlänkningar samt ökade minnesåtkomsttider introducerade av att minnesoperationerna utförs på minne som tillhör avlägsna noder. Med dessa resultat som bakgrund argumenterar författarna för att en effektiv schemaläggningsalgoritm för NUMA-system måste utnyttja minnesmigrering tillsammans med processorallokering. Vidare visar studien att det inte är tillräckligt att endast introducera minnesmigrering i en existerande algoritm. Anledningen är att det leder till att minnesmigrering sker allt för frekvent. Författarna presenterar istället algoritmen DINO (Distributed Intensity NUMA Online). DINO använder LLC missfrekvenser för att klassificera processer under körning. Klassificeringen nyttjas för att på ett effektivt sätt undvika onödiga minnesmigreringar. Tillsammans med en sofistikerad metod för att utföra minnesmigrering kan författarna visa att algoritmen lyckas öka prestandan i ett NUMA-system. En observation som görs är att denna typ av schemaläggning inte bara leder till en förbättrad prestanda överlag utan även att prestandan blir jämnare. Därför menar författarna att algoritmen också är intressant för kvaliteten på de tjänster som systemet utför.

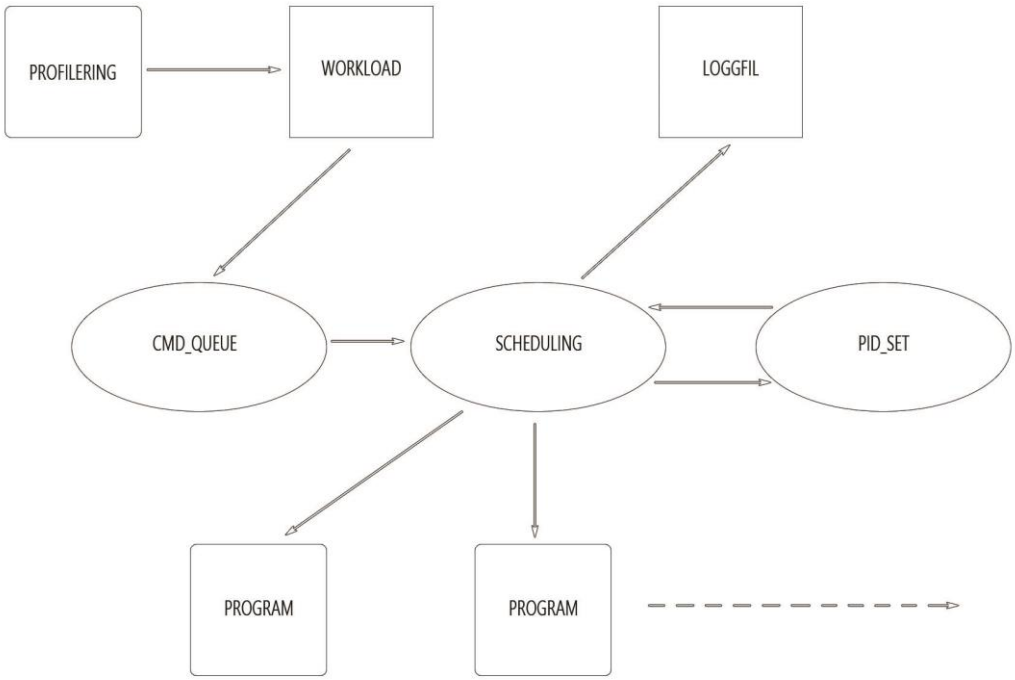
En jämförande undersökning av olika schemaläggningsalgoritmer för hantering av tävlan om systemresurser görs av Daci et al. [15]. Undersökningen görs på algoritmer anpassade för både UMA- och NUMA-system. För varje algoritm undersöks deras unika egenskaper, styrkor och svagheter. En slutsats av undersökningen är att den huvudsakliga svårigheten i att utveckla denna typ av algoritmer ligger i metoden för hur processer klassificeras. Missfrekvensen i LLC är enligt rapporten den bästa metoden, eftersom denna är en tydlig indikator på tävlan om alla typer av resurser i minnessystemet. För NUMA-system argumenterar författarna för att DINO är den mest lämpliga algoritmen då den utför både process- och minnesmigrering men samtidigt undviker överflödiga migreringar.

Blagodurov et al. [16] undersöker tillgängliga funktioner i Linux för schemaläggning av NUMA-system på användarnivå. De olika funktionerna delas in i två huvudsakliga grupper, sådana som kan användas för att övervaka systemets tillstånd och sådana som kan användas för att utföra schemaläggningsoperationer. Rapporten visar att Linux tillhandahåller detaljerad information om systemets tillstånd via de två virtuella filsystemen `sysfs` och `procfs`. Informationen som systemet presenterar i dessa filsystem kan användas för att driva schemaläggningsbeslut och är tillgängliga på användarnivå genom enkla filläsningar. För att styra processorallokeringen visas systemanropet `sched_setaffinity` samt kommandoradsverktyget `taskset`. Två olika funktioner för minnesmigrering visas också, systemanropet `numa_migrate_pages` och kommandoradsverktyget `migratepages`. Rapporten beskriver också verktygen `pfmon` och `perf` som kan användas för att samla in prestandarelaterad data under exekvering. Funktionen hos dessa verktyg är dock starkt beroende av vilka funktioner den underliggande hårdvaran erbjuder för ändamålet. I rapporten ger författarna också en kort beskrivning av hur standardschemaläggaren i Linux, CFS, använder schemaläggningsdomäner på NUMA-system tillsammans med en minnesallokeringspolicy där minne för varje process i första hand allokeras på den nod där processen exekverar. Enligt författarna innebär detta att datalokaliteten för exekverande processer bibehålls men att det leder till en dålig lastbalansering mellan de olika noderna. Detta är en av anledningarna till att anpassning av schemaläggningen är nödvändig för denna typ av system.

4 Design och implementation

Figur 4.1 visar en schematisk beskrivning av schemaläggarens struktur. Till att börja med profileras varje applikation för att tilldelas en klass. Vilken klass en process tilldelas beror på hur minnesintensiv den är. Därefter skapas ett arbetsschema (workload) med de program som schemaläggaren skall exekvera. För varje program i schemat anges bland annat programmets tilldelade klass samt dess starttid. När schemaläggaren startar läses arbetsschemat in till en kommandokö (cmd_queue) där varje program lagras med en specifik processbeskrivning. Under körning startar schemaläggaren programmen från kön vid utsatt starttid och för varje process som skapas sparas informationen om den nya processen in en tabell (pid_set). För att kunna följa schemaläggarens arbete skrivs meddelanden om det operationer som utförs till en loggfil.

Schemaläggaren är primärt uppdelad i tre funktioner: uppstart av processer, schemaläggning samt hantering av färdiga processer. Under utvecklingstiden lades mycket tid på att göra den skrivna koden lättförståelig, väldokumenterad samt att följa vedertagna konventioner inom UNIX-programmering. Detta för att schemaläggaren lättare ska kunna läsas igenom, redigeras och användas av andra. Koden är strukturerad så att varje fil har en specifik uppgift, vilket medför att enskilda delar enkelt kan bytas ut vid ett senare tillfälle.



Figur 4.1 - Schematisk beskrivning av schemaläggaren.

4.1 Processklassificering

Innan ett program startas genom schemaläggaren har det profilerats för att tilldelas en klass. Under profileringen har programmet körts ensamt på processorn samtidigt som `memprof` har loggat utförda minnesoperationer. När programmet kört klart sammanställs resultatet och programmet

tilldelas en klass. Desto fler minnesoperationer programmet utfört desto högre klass. Denna klass används senare av schemaläggaren vid beslut om vilken process som ska migreras.

4.2 Initiering av schemaläggaren

När schemaläggaren startar körs en uppstartsrutin för att initiera de datastrukturer som används under körning. Initiering beskrivs övergripande i Pseudokod 4.1 nedan. Rutinen börjar med att kontrollera systemets NUMA-policy, öppna loggfilen, läsa in arbetsschemat till kommandokön och allokera datastrukturen för lagring av processinformation. Därefter görs en inläsning av tillgängliga processorkärnor och dessa sparas som bitmaskar för varje nod. För att påbörja läsning av belastningen på varje minnestyrenhet öppnas datafilen för `memprof` och startvärden för dessa sparas. Avslutningsvis skapas en timer för körning av schemaläggningsrutinen och en timer för start av program. Timern för schemaläggningsrutinen initieras med satt intervall och till sist anropas en programstartsrutin för att starta de första programmen i kommandokön. Om initiering av någon av dessa funktioner misslyckas ges ett felmeddelande och körningen avbryts.

```
Vid initiering:
  Kontrollera NUMA-policy
  Öppna loggfil
  Initiera datastruktur för processinformation
  Läs arbetsschema och initiera kommandokö
  Läs in aktiva processorkärnor för varje nod
  Läs startvärden för memprof
  Initiera timers för processtart och schemaläggningsrutin
  Starta första programmen i kommandokön
```

Pseudokod 4.1 - Initiering av schemaläggaren.

4.3 Uppstart av processer

För att ange vilka program som ska köras läses en fil innehållandes ett arbetsschema in vid initiering. Detta schema är antingen skapat manuellt eller slumpmässigt genererat och innehåller varje programs starttid, profileringsklass, full sökväg samt programargument. Varje post i schemat läses in till en kommandokö där de sparas i form av en programbeskrivning. Under körning används en timer för att signalera när nya program ska startas. Denna timer avger en `SIGALRM`-signal varje gång tidsgränsen är nådd och då anropas en programstartsrutin. Pseudokod 4.2 illustrerar hur programstarten utförs. Rutinen startar alla program som har den aktuella tiden som starttid och därefter sätts timern till att ge en ny signal när nästa program i kön ska köras. För att starta ett program skapas en barnprocess. Till den nya processen allokeras en processorkärna i noden med minst belastning. Funktionen som väljer vilken kärna som ska allokeras föredrar en oallokerad kärna, annars väljs den kärna som har minst antal processer. När kärnan valts ut används funktioner i programbiblioteket TMC (Tilera Multicore Components) för att binda processen till den valda kärnan. Efter att processen bundits till en processorkärna läses informationen från processbeskrivningen och programmet exekveras. När ett program har startats sparas information om vilken aktuell processor, nod, klass och vilket process-id det har blivit tilldelat.

```
Vid varje SIGALRM:
För varje program med starttid <= aktuell tid:
  N <- Nod med lägst minnesbelastning
  C <- CPU i N med minst antal processer
  Skapa ny process
  Knyt processen till CPU C
  Exekvera program
  Spara information om process
```

```
Sätt timer att ge signal vid starttiden för nästa program i kön
```

Pseudokod 4.2 – Programstartsrutinen.

4.4 Schemaläggning

Rutinen som utför schemaläggning körs med ett fast intervall, som standard satt till fyra gånger per sekund. Detta styrs med hjälp av en timer som varje gång tidsgränsen är nådd genererar en `SIGPOLL`-signal. Denna fångas upp av huvudprocessen som anropar rutinen. Syftet med schemaläggningsrutinen är att motverka ojämn minnesbelastning och skapa en bättre balans noderna emellan. Pseudokod 4.3 beskriver hur schemaläggningsrutinen fungerar. Rutinens arbete kan delas upp i två huvudsakliga moment. Det första är att läsa av den aktuella minnesbelastningen för varje nod via `memprof`. Värdena som läses av subtraheras med värdena som lästes in vid föregående körning för att få fram momentanvärden, dessa ger en uppfattning om minnesbelastningen på varje minnesstyrenhet. Därefter sparas både absolutvärdena och de nya momentanvärdena. De senare används även vid utplacering av nystartade processer. Det andra steget är att besluta om en migrering är nödvändig. Detta görs genom att beräkna ett gränsvärde för hur mycket obalans i minnesbelastningen som krävs för att migrering ska ske. Gränsvärdet är baserat på den genomsnittliga belastningen multiplicerad med en migreringsfaktor. Om någon nod överskrider gränsvärdet tas beslut om migrering. Då väljs den nod med mest belastning och den med minst belastning ut. Därefter väljs den högst klassificerade processen på den mest belastade noden och denna migreras till noden med minst belastning. Schemaläggaren uppdaterar sedan den lagrade informationen om den migrerade processen

```
Vid varje SIGPOLL:
Läs in aktuell minnesbelastning via memprof
Beräkna gränsvärde för migrering
Undersök belastning och avgör om migrering är nödvändigt
Välj noder med mest/minst belastning
Om migrering är nödvändigt:
          Migrera högst klassificerade processen från noden med mest
          belastning till noden med minst belastning
```

Pseudokod 4.3 – Schemaläggningsrutinen.

4.5 Hantering av avslutade processer

I Linux-system avger varje barnprocess en `SIGCHLD`-signal till föräldraprocessen när dess tillstånd förändras. Exempelvis avges en sådan signal när barnprocessen avslutas. När en barnprocess avslutas fångar schemaläggaren `SIGCHLD`-signalen och anropar en termineringsrutin. Uppgiften för denna rutin är att se till att avslutade processer tas bort från systemet. Detta utförs med systemanropet `waitpid`, som raderar processen från operativsystemets processtabell. Termineringsrutinen ser också till att den information som schemaläggaren lagrat om processen tas bort.

5 Resultat

Ett syfte med projektet var att undersöka vilka möjliga prestandaförbättringar schemaläggaren skulle kunna åstadkomma. Detta kapitel beskriver först kortfattat testernas utförande och premisser för att sedan presentera deras resultat. I kapitlet refererar DFS till projektgruppens schemaläggare.

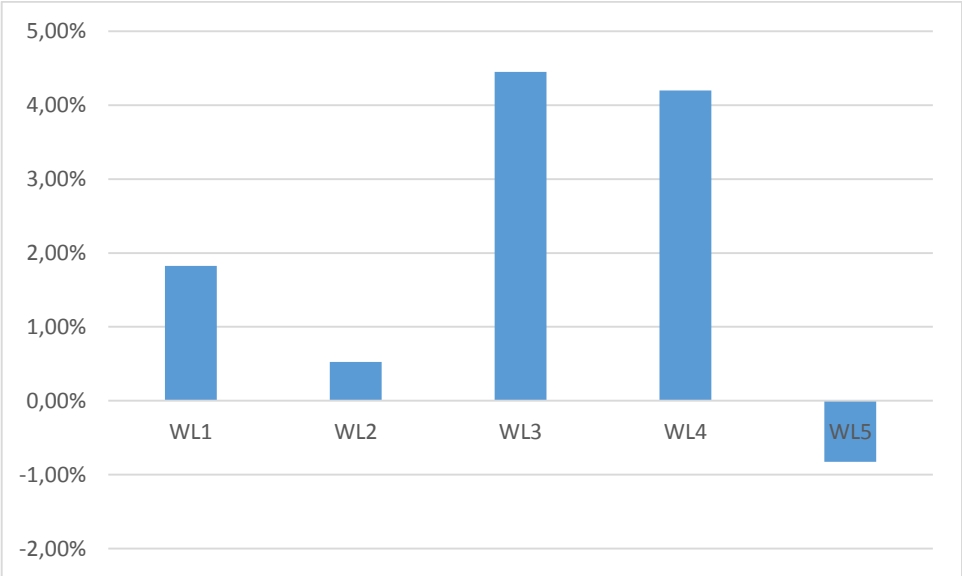
5.1 Testutförande

För att mäta vilka eventuella prestandavinster DFS lyckas åstadkomma skapades scheman innehållandes de testprogram som skulle köras. Dessa skapades till en början för hand men senare skrevs ett program för att slumpmässigt generera arbetsscheman utifrån den lista av testprogram som fanns att tillgå. Även egenskrivna syntetiska testprogram gjorda för att skapa stor minnesbelastning användes i en del av testerna. I korthet fungerar de syntetiska testapplikationerna så att de allokerar de ett fåtal megabyte minne och utför kontinuerligt operationer på slumpmässiga delar av detta minne. Nära nog samtliga operationer, uppemot 99 %, missar därmed i cache-minnet och går till primärminnet istället. Genom att inkludera dessa skapades en något högre minnesbelastning än med scheman där enbart program från testsviterna användes.

Testerna utfördes med både CFS och DFS. De utgick ifrån exekveringstiden för hela schemat utan hänsyn till applikationernas responstid eller hur rättvist körtiden fördelades under körning. Både testerna med DFS och CFS använde sig av en kommandokö och en timer för att starta processer på den tid som är bestämd av schemat. Testerna med CFS utfördes med systemets standard-konfiguration där funktionen striped memory är aktiverad.

5.2 Resultat utan syntetiska testapplikationer

Figur 5.1 visar resultaten från tester med fem olika scheman (WL1 – WL5) innehållandes enbart applikationer från SPEC CPU2006 och cBench (se kapitel 2.3). Alla scheman är slumpgenererade. Resultaten visar förbättringar på mellan 0,5 % och 4,5 % med i ett test ses dock en försämring på 0,8 %.

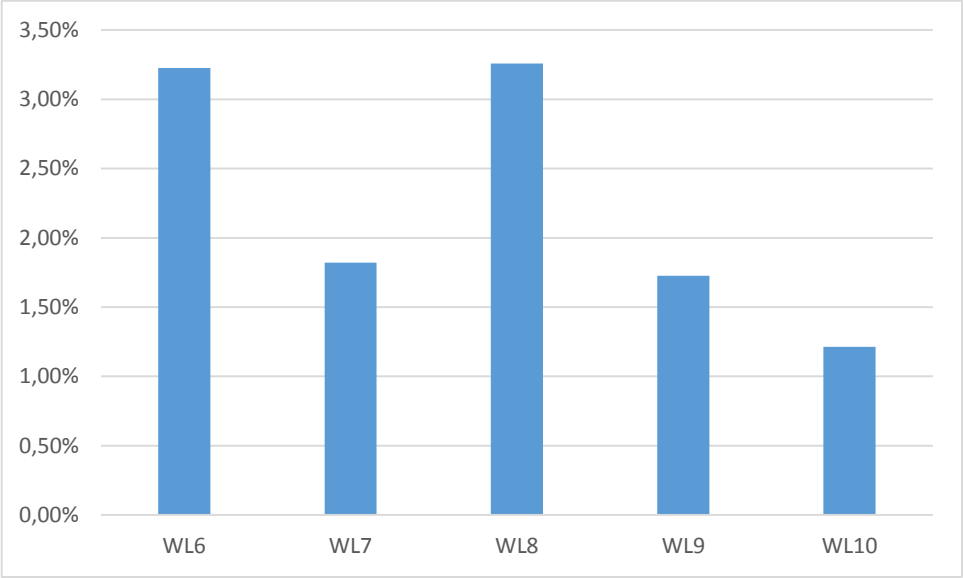


Figur 5.1 - Procentuell förbättring gentemot standardschemaläggaren.

5.3 Resultat med syntetiska testprogram

Figur 5.2 visar resultaten från tester med fem olika scheman (WL6 - WL10) innehållandes en blandning av applikationer från SPEC CPU2006 och cBench tillsammans med syntetiska testprogram.

Alla scheman är slumpgenererade. Resultaten visar förbättringar på mellan 1,2 % och 3,2 % i samtliga fall.



Figur 5.2 - Procentuell förbättring inkl. syntetiska tester gentemot standardschemaläggaren.

6 Diskussion

Resultaten visar endast en minimal prestandavinst med den NUMA-anpassade schemaläggaren. Detta diskussionskapitel behandlar testmetod och resultat, de potentiella orsakerna bakom utebliven förbättring, gjorda designval samt diskuterar möjliga utökningar av slutprodukten.

6.1 Testmetod och resultat

Då testningen inleddes kördes varje schema ett fåtal gånger. Körningarna visade endast minimala skillnader i exekveringstid mellan olika körningar och dessa skillnader utgjorde endast en ytterst liten del av den totala exekveringstiden. Det är dock tvivelaktigt om de minimala skillnaderna som visades i tidiga tester gäller i samtliga fall. Under senare testning utfördes endast en körning med varje schema. Om varje test istället hade utförts ett flertal gånger och resultatet baserats på ett medelvärde av exekveringstiderna hade ett mer tillförlitligt resultat kunnat fastställas.

Testerna som utfördes mätte endast exekveringstid för hela arbetsscheman. En egenskap som hade varit intressant att mäta var exekveringstiden för varje enskild process. På så sätt hade det varit möjligt att undersöka hur stora variationer i exekveringstid som uppstår, en aspekt som är relevant för bland annat tjänstekvalitet.

Då endast en minimal prestandaförbättring noterades i testerna med utvalda testapplikationer utökades testningen med scheman innehållandes syntetiska testprogram. Motivationen var att undersöka om schemaläggaren kunde prestera jämförbart bättre om minnesbelastningen på systemet ökades. Dock visade resultaten en fortsatt minimal prestandavinst. I tidigare studier har liknande tester utförts tillsammans med liknande anpassningar i processchemaläggningen. Resultaten i dessa studier har pekat på en mer tydlig prestandaförbättring (se kapitel 5).

6.2 Potentiella orsaker till minimal prestandavinst

Det finns många faktorer som skulle kunna vara orsak till de resultat som fåtts. Några av dem beror på begränsningar i implementationen av schemaläggaren och andra på måldatorns egenskaper och utvecklingsmiljö.

6.2.1 Minnesmigrering

En önskvärd funktionalitet som inte gick att implementera var minnesmigrering. Detta berodde på att olika systemfunktioner för minnesmigrering inte fungerade. I Linux används primärt två olika funktioner för att flytta minne, `numa_migrate_pages` från biblioteket `libnuma` och systemanropet `migrate_pages`. Med den förstnämnda funktionen kunde minne flyttas mellan vissa noder, medan funktionen låste sig vid flytt mellan andra. Den senare rapporterade att den lyckats flytta minne, men när minnesanvändningen undersöktes visade det sig att ingenting hade flyttats. Orsaken till detta är okänd, men det finns kända buggar i `libnuma` [17] som i viss mån är besläktade med det observerade problemet. Detta bibliotek används för att ange minnespolicy för processer på NUMA-system. Då `libnuma` har anpassats av Tilera är det möjligt att ytterligare buggar har införts. Till exempel så är `numa_migrate_pages` inte dokumenterad i koden till den version av `libnuma` som ingår i utvecklingsmiljön. Den version som ingår, 0.9.11, är också relativt gammal. I skrivande stund är 2.0.8 den senaste versionen [18]. I detta fall skulle en framtida uppdatering av utvecklingsmiljön eventuellt kunna lösa problemet.

Den eventuella vinsten med att migrera minne tillsammans med processer är dock inte självklar. Det är en kostsam operation sett till tid och skapar belastning både på den ursprungliga och på den nya minnesdomänen. Det är inte heller självklart att processer kommer använda det sedan tidigare allokerade minnet i någon större utsträckning. Då processer migreras kommer enbart allt nytt minne

de begär att allokeras i den minnesdomän som tillhör processens nya nod. Tidigare studier argumenterar för att minnesmigrering är en av huvudfaktorerna till förbättrad prestanda då det i typiska NUMA-system drastiskt minskar åtkomsttiden för den flyttade processen. En effekt av arkitekturen hos TILE*Pro*64 är dock att skillnaden i åtkomsttid mellan olika noder inte är speciellt stor. Av den anledningen är det osäkert om en större prestandavinst skulle uppnåtts även om minnesmigrering utfördes.

6.2.2 Mätning av minnesbelastning

Under utvecklingen har flera metoder för att mäta av minnesbelastning undersökts. På grund av bland annat otydlig dokumentation har det varit svårt att veta vilka värden som varit relevanta att använda som utgångspunkter. Under en del av utvecklingstiden användes varje kärnas PMC-register för mätning av cache-missar. Eftersom en miss i cache-minnet innebär att systemet måste vända sig till primärminnet, borde de värdena ge en indikation på aktuell minnesbelastning. Ett problem med detta var att veta vilka hårdvaruhändelser PMC-registren skulle konfigureras till att räkna, då informationen om vad de olika händelserna innebar inte var tydlig. Då varje processorkärna har en egen uppsättning PMC-register ger dessa möjligheten att mäta antalet cache-missar på varje individuell kärna. Om endast en process exekveras på kärnan är således samtliga cache-missar från den processen. Värdena från `memprof` indikerar istället belastningen på varje minnesstyrenhet. De värdena kan på så sätt inte ge någon indikation om vilket program det är som orsakar mest belastning. I teorin står cache-missar från PMC-register och värden från `memprof` i relation till varandra och båda borde därför kunna användas för att ge en bra indikation om minnesbelastningen. En konsekvens av att inte veta vilket program det är som generar mest belastning, är att det blir svårare att besluta vilken process som bör migreras. Därmed finns det för- och nackdelar med de olika mätvärdena, men de skulle också kunna användas parallellt (se kapitel 6.4).

6.2.3 Brist på minnesbelastning

Ytterligare en faktor som kan ha inverkat på resultatet är att de testapplikationer som använts inte är tillräckligt minnesintensiva för att skapa en tydlig minnesbelastning. Om det inte sker tillräckligt mycket minnesreferenser för att belasta minnesbussen, kommer de anpassningar som schemaläggaren gör inte att ha någon större effekt. Ett problem ligger i den begränsade mängden primärminne i måldatorn. De minnesintensiva testapplikationerna i SPEC CPU2006 kräver relativt stor mängd minne vilket leder till att endast ett fåtal instanser av dessa kan exekveras parallellt. Skulle det finnas tillgång till mer minne, skulle det eventuellt gå att se en förbättrad prestanda genom att fler minnesintensiva program skulle kunna köras samtidigt.

6.2.4 Val av datorsystem

Måldatorsystemet implementerar flera funktioner i både hårdvara och mjukvara med målet att underlätta för programmerare att utveckla applikationer för processorn. Det finns ett flertal saker i systemet som i stor utsträckning döljer dess NUMA-egenskaper. Till exempel så är funktionen striped memory (se kapitel 2.1.2) påslagen som standard. Denna innebär att endast en logisk minnesdomän presenteras för operativsystemet och systemet utför automatiskt balansering av minnesbelastningen. Egenskaperna döljs också av det effektiva mesh-nätverket iMesh (se Appendix B - iMesh). Den snabba kommunikationen i nätverket innebär att åtkomsttiden för varje processorkärna till de olika minnesdomänerna blir relativt jämn. Således är kostnaden för att använda minne på en avlägsen nod inte betydligt större än kostnaden för att använda närliggande minne. På grund av dessa funktioner blir det både svårt och delvis överflödigt att göra ytterligare NUMA-anpassningar i mjukvaran.

6.2.5 Overhead

Även om de åtgärder som utförs av schemaläggaren för att minimera minnesbelastningen innebär en optimering, är det möjligt att den overhead som orsakas av schemaläggningsrutinerna motverkar prestandavinsten. Denna overhead består både av att processortid tas från andra exekverande processer och att minnessystemet påverkas. Schemaläggaren är byggd med minimalism och effektivitet i åtanke. Till exempel använder schemaläggninsrutinerna endast heltalsberäkningar och för lagring av processinformation används ett röd-svart träd, vilket ger skalbarhet för ett stort antal processer. Fler optimeringar går dock med största sannolikhet att införa och genom effektivisering av schemaläggningsrutinerna skulle eventuellt exekveringstiden kunna minskas ytterligare.

6.3 Designval

Innan schemaläggaren började skrivas genomgicks en litteraturstudie av liknande projekt för att hitta en bra utgångspunkt för arbetet (se kapitel 3.2). Den första fråga som behövde besvaras var hur minnesbelastning kan mätas. Tidigare studier visade att minnesbelastningen står i tydlig relation till frekvensen av missar i cache-minnet.

På måldatorsystemet fanns primärt två alternativa sätt att mäta minnesbelastning. Det första var att använda PMC-register (se kapitel 2.1.1) på varje processorkärna och det andra var att avläsa data via `memprof`. PMC-registren i varje processorkärna kan avläsas antingen via pollning eller så kan registerna konfigureras att ge avbrott när de uppnått ett satt gränsvärde. Av dessa två alternativ är avbrott det mest attraktiva eftersom det i teorin orsakar mindre overhead [19]. Dock saknades relevant dokumentation av måldatorsystemet för att kunna genomföra detta. Detta gällde dokument som refererades i övrig dokumentation och sades innehålla information om och exempel på hur bland annat avbrottsvektorer skulle initieras. I och med detta föll valet på pollning. Pollning av PMC-register kräver modifieringar av systemets hypervisor för att kunna utföras på användarnivå. För att läsa PMC-registren på varje kärna behöver varje kärna köra en process eller tråd som kontinuerligt läser av registren. De avlästa registervärdena kan sedan skickas till en huvudprocess eller skrivas till ett delat minne. Alternativt kan huvudprocessen hoppa från kärna till kärna och hämta mätvärden. Båda alternativen är relativt resurskrävande och därför valdes istället pollning av data från `memprof`, eftersom dessa data är enkelt tillgänglig via filsystemet. Pollningen sker med ett fast intervall och i samband med uppdateringen av värdena utför schemaläggaren processmigrering vid behov. Om intervallet är kort kommer schemaläggaren att ta mycket tid från processerna som körs, och om det är långt kommer noggrannheten på mätvärdena försämras. Vid längre intervall kommer även schemaläggaren svara långsammare på ojämn belastning i minnessystemet. Pollningsintervallet valdes med målet att schemaläggningsrutinen endast skulle ta en minimal mängd tid från exekverande processer. Testapplikationerna som användes utför beräkningar över lång tid och bör därmed inte vara i behov av att schemaläggningen snabbt anpassas. Vid de mätningar som gjorts har inga förbättrade resultat kunnat uppnås med varken ett kortare eller längre pollningsintervall. Värdet som används är baserat på liknande studier.

Gränsvärdet för migrering används för en uppskattning av kostnaden att exekvera en process med lägre datalokalitet på en avlägsen nod. Värdet är satt för att undvika onödiga migreringar då inte minnesbelastningen är tydligt snedfördelad. Migreringar ska enbart ske när sannolikheten för en prestandavinst är hög, vilket är vad detta gränsvärde är tänkt att åstadkomma.

6.4 Möjliga utökningar

Profileringen som görs offline (se kapitel 2.2.5) på varje applikation berättar hur minnesintensiv den är. En annan egenskap som hade varit intressant att undersöka är hur känsliga varje program är. Med känslighet menas hur mycket ett program påverkas av att samköras med olika typer av program.

Sådan information skulle medföra att schemaläggaren kan ta bättre beslut vad gäller utplacering av processer och vilka processer som är mest lämpliga att migrera och vart de bör migreras. I nuläget tas beslutet om att migrera processer först om en minnesstyrenhet har dubbelt så hög belastning som en annan, och då migreras den process med högst klass. Utöver att införa känslighet som migreringsparameter, skulle även en uppskattning av migreringskostnad och potentiell migreringsvinst kunna införas. Genom att göra en avvägning mellan minnesbelastning och dessa värden skulle bättre beslut kunna tas om det är lönsamt att migrera en process eller inte.

I senare versioner av Linux än den som ingår i utvecklingsmiljön finns det inbyggda funktioner för att läsa prestandainformation via verktyget `perf`. Om systemet hade uppdaterats till en senare version hade `perf` kunnat användas för att få tillgång till fler relevanta data, till exempel prestanda-information per process, istället för enbart per kärna eller per minnesenhet. Detta hade gjort det möjligt att profilera och omklassificera applikationer under körning, vilket hade lett till en mer dynamisk schemaläggning. Vilken funktionalitet `perf` erbjuder är beroende av stöd i hårdvaran och det är inte säkert att dessa värden hade varit tillgängliga. Möjligheterna har inte undersökts djupare. Online-profilering hade i viss mån även kunnat åstadkommas om värden från `memprof` hade använts parallellt med processorkärnornas PMC-register. Då detta inte hade gett information per process utan enbart per kärna hade man dock bara fått tillförlitlig information om processerna kördes ensamma på varje kärna.

7 Slutsats

Implementation av en processchemaläggare innebär många olika val och kompromisser som uppstår från varierande funktionalitet i både hårdvara och utvecklingsmiljö. Vi har undersökt och presenterat ett flertal av dessa som berör det använda måldatorsystemet.

Projektet har lett fram till en processchemaläggare anpassad för ett NUMA-system med en mångkärnig processor. Schemaläggaren balanserar minnesbelastning genom att allokera processorkärnor så att minnesintensiva processer fördelas mellan systemets olika noder. Delvis på grund av brister i nödvändiga systembibliotek implementerades inte alla funktioner i schemaläggaren fullt ut.

Till skillnad från studier där liknande anpassningar gjorts i schemaläggningen för NUMA-system visar våra resultat i bästa fall endast en marginell prestandaförbättring. Förbättringarna är så pass små att de inte anses falla inom gränsen för vad som utgör en reell prestandavinst. En anledning till uteblivna prestandavinster anses ligga i att processorn TILE*Pro*64 är anpassad för att delvis dölja systemets NUMA-egenskaper.

Utifrån testerna har projektgruppen dragit slutsatsen att en schemaläggare anpassad för balansering av minnesbelastningen i NUMA-system inte har någon väsentlig effekt för prestandan på system som inte har en betydande skillnad i åtkomsttid mellan olika minnesdomäner.

Källförteckning

[1] A. Silberschatz, P. B. Galvin och G. Gagne, Operating Systems Concepts, Wiley, 2008.

[2] D. Edholm, V. Nilsson och A. Löfgren, ”datx02-22 - Github,” 2013. [Online]. Available: https://github.com/gusloande/datx02-22. [Använd 20 May 2013].

[3] Tilera Corporation, ”Tile Processor Architecture Overview for the TILEPro Series,” Tilera Corporation, San Jose, 2009.

[4] Tilera Corporation, ”Tile Processor User Architecture Manual,” Tilera Corporation, San Jose, 2009.

[5] Tilera Corporation, ”Optimization Guide,” Tilera Corporation, San Jose, 2010.

[6] Tilera Corporation, ”TILEncore Card Product Brief,” 2011. [Online]. Available: http://www.tilera.com/sites/default/files/productbriefs/TILEProEncore_PB024_v5_0.pdf. [Använd 07 June 2013].

[7] Tilera Corporation, ”Multicore Development Environment User Guide,” Tilera Corporation, San Jose, 2010.

[8] Tilera Corporation, ”Multicore Development Environment Programming the Tile Processor,” Tilera Corporation, San Jose, 2009.

[9] Standard Performance Evaluation Corporation, ”SPEC CPU2006,” SPEC, 2011. [Online]. Available: http://www.spec.org/cpu2006/. [Använd 20 May 2013].

[10] Collective Tuning, ”CTools:CBench - cTuning,” 2012. [Online]. Available: http://ctuning.org/wiki/index.php/CTools:CBench. [Använd 20 May 2013].

[11] T. Koita, T. Katayama, K. Saisho och A. Fukuda, ”Memory Conscious Scheduling for Cluster-based NUMA Multiprocessors,” *The Journal of Supercomputing,* vol. 16, nr 3, pp. 217-235, 2000.

[12] S. Zhuravlev, S. Blagodurov och A. Fedorova, ”Addressing shared resource contention in multicore processors via scheduling,” i *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, Pittsburgh, ACM New York, NY, USA, 2010, pp. 129-142.

[13] Z. Majo och T. R. Gross, ”Memory management in NUMA multicore systems: trapped between cache contention and interconnect overhead,” i *Proceedings of the international symposium on Memory management*, San Jose, ACM New York, NY, USA, 2011, pp. 11-20.

[14] S. Zhuravlev, S. Blagodurov, A. Fedorova och A. Kamali, ”A case for NUMA-aware contention management on multicore systems,” i *PACT '10 Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, Edinburgh, ACM New York, NY, USA, 2010, pp. 557-558.

[15] G. Daci och M. Tartari, ”A Comparative Review of Contention-Aware Scheduling Algorithms to Avoid Contention in Multicore Systems,” i *Proceedings of the Third International Conference on Trends in Information, Telecommunication and Computing*, New York, Springer New York, 2013, pp. 99-106.

[16] S. Blagodurov och A. Fedorova, ”User-level scheduling on NUMA multicore systems under Linux,” 2011. [Online]. Available: https://www.kernel.org/doc/ols/2011/ols2011-clavis.pdf. [Använd 7 June 2013].

[17] J. Stancek, ”Bug 870326 - migrate_pages() reports success, but pages are not moved to desired node,” 2012. [Online]. Available: https://bugzilla.redhat.com/show_bug.cgi?id=870326. [Använd 7 June 2013].

[18] A. Kleen, ”Index to /www/projects/libnuma/download,” SGI, 11 October 2012. [Online]. Available: ftp://oss.sgi.com/www/projects/libnuma/download. [Använd 7 June 2013].

[19] E. Casas, ”EECE 379 - Digital and Microcomputer System Design,” 2000. [Online]. Available: http://www.ece.ubc.ca/~edc/379/lectures/lec4.pdf. [Använd 20 May 2013].

Appendix A - Completely Fair Scheduler

Completely Fair Scheduler (CFS) är en processchemaläggare implementerad på systemnivå i operativsystemet Linux. Sedan version 2.6.23 är det den schemaläggare som används av systemet i dess standardutförande. CFS är en kortsiktig schemaläggare, vilket innebär att dess uppgift är att styra allokeringen av processortid till processer som ska exekveras. De huvudsakliga målen med CFS är att åstadkomma en schemaläggning där varje process får en rättvist fördelad andel av processortiden och som resulterar i en så hög nyttjandegrad av processorn som möjligt. Samtidigt ska schemaläggningen bibehålla en hög prestanda för interaktiva processer.

Till skillnad från dess föregångare, den så kallade O(1)-schemaläggaren, bygger inte CFS på att körbara processer placeras i köer. Istället använder CFS ett röd-svart träd (en typ av binärt sökträd som är självbalanserande). I trädet är varje process representerad tillsammans med en virtuell körtid. Denna körtid utgörs av den sammanlagda exekveringstid processen blivit tilldelad och varje nod i trädet är sorterad efter detta värde. Sorteringen leder till att den process som blivit tilldelad minst körtid representeras av den nod som befinner sig längst till vänster i trädet. För att schemaläggaren ska upprätthålla rättvisa mellan processerna vad gäller tilldelad exekveringstid väljer CFS den process som befinner sig längst till vänster i trädet när en ny process ska exekveras. När processen sedan tas från på processorn adderas exekveringstiden till processens virtuella körtid. Processen stoppas sedan tillbaka i trädet och sorteras efter den nya körtiden.

En intressant aspekt med CFS är att den inte direkt använder tidsperioder (time slices) eller prioritering av processer. I ett system som använder CFS definieras istället en maximal exekveringstid som kan tilldelas till en process. Schemaläggaren använder sedan processens prioritet som en avtagande faktor. För högt prioriterade processer sätts en låg prioritet och för lågt prioriterade processer görs det omvända. Den exekveringstid som tilldelas en process blir då snabbare avtagande för lågt prioriterade processer än för högt prioriterade processer. Detta är en elegant effekt av CFS arbetssätt och undviker problemen som tidigare schemaläggare haft med att behöva bibehålla en processkö för varje prioritet.

En ytterligare intressant egenskap med schemaläggaren är användningen av ett röd-svart träd för att sortera de processer som är redo att exekveras. Denna struktur gör det möjligt att utföra alla uppslagsoperationer, insättningar och borttagningar med tidskomplexiteten O(log(n)), där n är antalet noder i trädet.

Appendix B - iMesh

iMesh är samlingsnamnet för de sex olika nätverk som förbinder processorkärnorna på TILE*Pro*64. Nätverken är uppdelade i två klasser och har alla olika funktion och benämning. Tillsammans förser de kärnorna med möjligheten att kommunicera och överföra data med väldigt hög hastighet och korta svarstider. Den första klassen av nätverk består av de nätverk som är användbara i applikationer på användarnivå och den andra klassen utgörs av nätverk som processorn använder internt för olika systemoperationer. Som exempel kan nämnas att det är iMesh som ligger till grund för det distribuerade cache-systemet på processorn.

Nätverken i iMesh är av typen mesh-nätverk. I ett mesh-nätverk finns ingen central dataväxel som sköter kommunikationen utan istället har varje processorkärna en egen dataväxel som ansluter kärnan till angränsande kärnor. Innebörden av detta är att varje processorkärna inte bara ansvarar för att skicka och ta emot egna meddelanden utan måste även ansvara för att vidarebefordra meddelanden avsedda för andra kärnor på nätverket.

Ett av de sex nätverken är ett statiskt kretskopplat och de fem övriga är dynamiska paketförmedlade nätverk. Skillnaden mellan det statiska och de dynamiska nätverken är att meddelanden som skickas på det statiska nätverket inte innehåller någon adresseringsinformation. Denna information finns istället i dataväxeln på varje kärna så att när ett meddelande ankommer på kan växeln direkt avgöra vart meddelandet ska skickas. Tekniken gör det möjligt att skicka data över nätverket med väldigt korta fördröjningstider. Framförandet av meddelande som skickas över nätverket sker i en förutbestämd ordning där meddelandet först framförs i X-led och sedan i Y-led utifrån den 8x8-matris som processorkärnorna utgör. Nedan följer en kort beskrivning av varje nätverk.

User Dynamic Network (UDN):

Detta dynamiska nätverk är tillgängligt för applikationer på användarnivå. Genom att anropa rutiner från de programbibliotek som ingår i utvecklingsmiljön till processorn kan nätverket användas för att på ett enkelt sätt skicka meddelande mellan processorkärnor. Möjligheten till den här typen av kommunikation är relativt unik för processorarkitekturen och applikationer kan använda dessa möjligheter helt utan behov av att utföra systemanrop.

I/O Dynamic Network (IDN):

IDN används primärt för dataöverföringar mellan processorkärnor och I/O-enheter eller mellan I/O-enheter och primärminnet. En skyddsmekanism gör att detta nätverk inte är tillgängligt för applikationer på användarnivå.

Memory Dynamic Network (MDN):

MDN används för dataöverföring mellan enskilda processorkärnor och mellan processorkärnor och primärminnet. Endast processorns cache-system har en direkt anslutning till detta nätverk.

Coherence Dynamic Network (CDN):

CDN används för att skicka speciella meddelanden vars mål är att bibehålla konsistens i cache-systemet. Dessa meddelanden skickas när data i en specifik cache ska deklareras som ogiltig.

Tile Dynamic Network (TDN):

Används för att utföra liknande funktioner som MDN men används endast internt för kommunikation mellan processorkärnor. Endast processorns cache-system har direkt åtkomst till detta närverk.

Static Network (STN):

Detta statiska nätverk är designat för att på ett effektivt sätt skicka operander mellan processorkärnor. Nätverket kan nyttjas av applikationer på användarnivå genom att anropa rutiner i det programbibliotek som ingår i utvecklingsmiljön till processorn.