Runtime management af rekonfigurerbare arkitekturer

Esben Rosenlund Hansen

Kongens Lyngby 2007
IMM-MSC-2007

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Resumé

Denne afhandling omhandler de dynamiske egenskaber ved rekonfigurerbare arkitekturer. Den første del af rapporten fokuserer på studier af kommercielle *Field Programable Gate Arrays* (FPGA'ere), der understøtter dynamisk partiel rekonfigurering. Dette danner baggrund for en bedre forståelse af begrænsningerne og mulighederne for at implementere dynamiske rekonfigurerbare systemer på generelle kommercielle FPGA'er.

Den anden del beskriver udviklingen og forbedringerne af COMOS et simuleringsmiljø for co-processor-koblede rekonfigurerbare arkitekturer. COSMOS omfatter en generel applikationsmodel og en arkitekturmodel, som tilsammen modellerer de dynamiske egenskaber ved rekonfigurerbare arkitekture. Modellen er opdateret på en række vigtige punkter, som i høj grad har forbedret fleksibiliteten, og gjort det muligt at udføre store simuleringer bestående af mange applikationer. Der er implementeret en række forskellige *runtime*-strategier for dynamiske rekonfigurerbare co-processorer, som er afprøvet ved hjælp af modellen. Specielt en *runtime*-strategi baseret på en proaktiv algoritme viste sig at være de andre strategier overlegen. Den proaktive algoritme kan afgjort være med til at sætte en guideline for fremtidige studier af *runtime*-managere til dynamiske rekonfigurerbare co-processorer.

Summary

This thesis focuses on the dynamic behavior of reconfigurable architectures. The first part describes the study of commercial Field Programmable Gate Arrays (FPGA's) that support partial dynamic reconfiguration. This work aids with a better understanding of the limits and the potential of implementing dynamic reconfigurable architectures in a main-stream commercial FPGA.

The second part describes the improvement and development of a simulation framework for coprocessor-coupled reconfigurable architectures, namely COSMOS. The COSMOS simulation framework comprises a generic application model and an architecture model, the combination of which captures the dynamic behavior of reconfigurable architectures. The model has been updated on several key issues which have significantly improved the flexibility and the usability of the model. A number of different runtime management strategies for dynamic reconfigurable coprocessors are implemented and tested on the model. Especially one runtime strategy relying on a proactive algorithm revealed to be superior, and can definitely be used as a guideline for further studies in optimizing the runtime manager of dynamic reconfigurable systems.

Forord

Denne rapport er udarbejdet i forbindelse med et eksamensprojekt ved institut for Informatik og Matematisk Modellering på Danmarks Tekniske Universitet. Projektet er udført under vejledning af Jan Madsen.

Jeg vil gerne benytte lejligheden til at takke min vejleder Jan Madsen for gode råd og konstruktive kommentarer under projektforløbet. Derudover skal specielt min mor have en stor tak for den hjælp, hun har bidraget med i løbet af projektet. Der skal også lyde en tak til mine venner for altid at komme med konstruktive kommentarer og opmuntrene ord på min vej mod at opnå en Master Degree.

Kgs. Lyngby, 2007.

Esben Rosenlund Hansen

Indhold

Resumé i

Summary iii

Forord v

1 Introduktion 1

1.1 Indledning 1

1.2 Baggrund 2

2 Problemstilling 5

3 Hardware 7

3.1 FPGA'ers arkitektur 7

3.2 Xilinx dynamiske rekonfigureringsmetoder 12
3.3 Egne forsøg 15

4 COSMOS modellen **23**

4.1 Analyse 23
4.2 Generel beskrivelse 28
4.3 Forbedringer 38

5 Runtime-styringen **49**

5.1 Grundliggende allokerings-strategi 49
5.2 Dynamisk prioritet 52
5.3 Spiral-allokering 52
5.4 Kritisk vej 54
5.5 Simuleringer 55

6 Avanceret reallokerings-algoritme **69**

6.1 Indledende fase 70
6.2 Proaktiv reallokerings-algoritme 71
6.3 Simuleringer 74

7 Diskussion **79**

8 Fremtidigt arbejde 85

9 Konklusion 89

A Screenshot 91

A.1 Floorplanner 92

A.2 FPGA_editor statisk design 93

A.3 FPGA_editor komplette design 94

B Task's files 95

B.1 Task0 95

B.2 Task1 96

B.3 Task2 97

B.4 Task3 98

B.5 Task4 99

C Task-grafer for de fem applikationer 101

D Udsnit af en log-fil fra COSMOS 103

E Artikel 107

F How to install SystemC 117

G CD **123**

Kapitel 1

Introduktion

1.1 Indledning

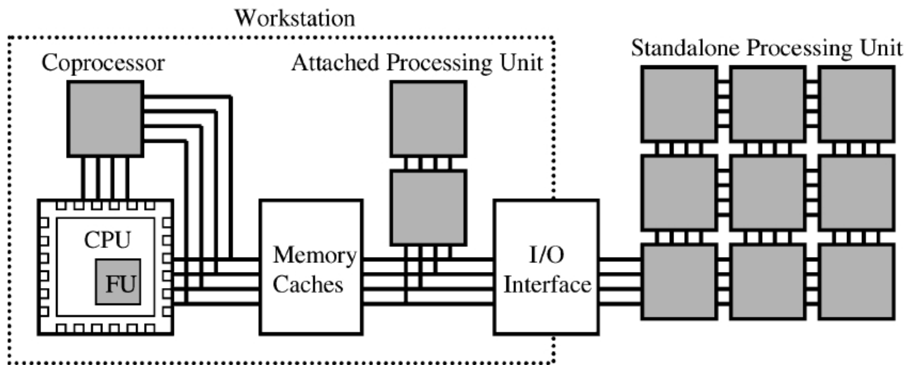
Arbejdet udført i dette projekt har omhandlet studier af kommercielle F-PGA'er, der understøtter dynamisk partiel rekonfigurering. Dette danner baggrunden for en dybere forståelse af begrænsningerne og mulighederne ved generelle kommercielle FPGA'er i forbindelse med implementering af dynamiske rekonfigurerbare systemer. For at få et bedre indblik i dynamiske rekonfigurerbare systemer, og hvad der skal til for at realisere et sådan system på traditionelle platforme, udarbejdes og videreudvikles modellen COSMOS. COSMOS er et simuleringsmiljø for co-processor-koblede rekonfigurerbare arkitekturer, som er udviklet af Kehuai Wu i forbindelse med hans Ph.d. projekt [13]. De næste afsnit giver en kort gennemgang af baggrunden for projektet, derefter skitseres de forskellige problemstillinger.

1.2 Baggrund

I dag eksisterer der hovedsageligt to forskellige måder at udføre en applikation på. Den ene foregår ved hjælp af en programmerbar mikroprocessor. Arkitekturer baseret på en mikroprocessor supporterer generelt en stor mængde instruktioner, som dækker alt fra logiske operationer til hukommelsesoperationer. Ved at kompilere applikationen til det korrekte instruktionsæt kan den køres på mikroprocessoren. Dette betyder, at arkitekturer med mikroprocessorer er forholdsvis fleksible, til gengæld for denne fleksibilitet er mikroprocessorer tit meget ineffektive, når det kommer til ydelse og strømforbrug. Den anden fremgangsmåde er at designe applikations specifikt hardware også kaldet ASIC's (*Application-Specific Integrated Circuit*) til udførelsen af en specifik applikation. En ASIC giver en meget høj ydelse, til gengæld er den meget specifik og kan ofte kun udføre en enkelt applikation, så den mangler i høj grad fleksibilitet. For at kombinere de bedste egenskaber fra de traditionelle computerarkitekturer, hastigheden fra ASIC's og fleksibiliteten fra arkitekturer baseret på en mikroprocessor, er ideen om en helt ny arkitektur opstået, som udnytter dynamisk rekonfigurerbar hardware. Dynamisk rekonfigurerbar hardware har mange fordele. Den helt centrale er muligheden for løbende at generere dedikeret hardware, som optimerer udførelsen af en given applikation. På den måde opnår rekonfigurerbare arkitekturer fordelene fra platforme baseret på mikroprocessorer og ASIC's. Rekonfigurerbare systemer kan i fremtiden være med til at erstatte andre gængse arkitekturer, og bidrage med mange positive aspekter herunder bedre ydelse, større fleksibilitet, selvreparation, adaptive hardware-algoritmer samt et reduceret strømbrug. I et sådan system er det muligt at fjerne inaktive rekonfigurerbare enheder helt og dermed bruges ingen strøm i f.eks. *idle*. Det er derfor væsentligt med en bedre forståelse af rekonfigurerbare systemer, for at være på forkant med udviklingen.

Inden for rekonfigurerbare arkitekturer går trenden mod et design bestående at en *host*-mikroprocessor sammen med en rekonfigurerbar enhed kaldet RU (*Reconfigurable unit*). Dette skyldes, at der er visse operationer, som stadig ikke udføres effektivt på en RU. For at effektivisere udførelsen af en applikation bliver de operationer, som ikke egner sig til at afvikles på en RU, udført på *host*-mikroprocessoren. De dele af et program, med et intensivt behov for regnekraft mappes til den rekonfigurerbare

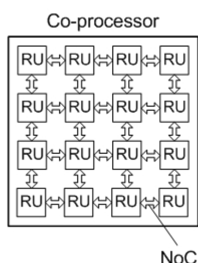
enhed, som giver mulighed for at udnytte dedikeret hardware til udregningerne. For rekonfigurerbare systemer, som benytter sig af dette design, findes der flere forskellige måder at koble *host*-mikroprocessoren og den rekonfigurerbare enhed. På figur 1.1 er vist 4 forskellige måder. Den



Figur 1.1: Forskellige metoder, hvorpå en rekonfigurerbar enhed (RU) kan kobles med en mikroprocessor, de grå felter specificere hver metode [4]

første mulighed er, at RU'en indgår som en funktionel enhed FU (*functional unit*) i mikroprocessoren. Dette betyder, at mikroprocessoren, set fra en programmørs synsvinkel, er forholdsvis traditionel med undtagelse af en *custom*-instruktion, som kan ændres med tiden. De tre andre metoder er *Stand-alone*-enhed, *Attached processing*-enhed og co-processor som minder om hinanden, men graden af den tæthed hvorved de er koblede til *host*-mikroprocessoren er meget varierende. I alle tre designs er det muligt for den rekonfigurerbare enhed og mikroprocessoren at eksekvere forskellige applikationer på samme tid. Det design med den tætteste kobling mellem enhederne er et co-processor design. Her har co-processoren direkte adgang til mikroprocessorens hukommelse. *Host*-mikroprocessoren kontrollerer normalt co-processoren gennem beskeder i stedet for egentlige instruktioner, som benyttes til at kontrollere en RU, hvis den indgår som funktionel enhed i mikroprocessorens datavej. *Attached processing*-enheden er koblet til *host*'en vha. af en ekstern bus. I denne konfiguration har den rekonfigurerbare enhed ikke direkte adgang til *host*-mikroprocessorens hukommelse. *Stand-alone*-enheden er det design, hvor den rekonfigurerbare enhed og *host*-mikorprocessoren er løsest

koblet. RU'en kan være en server koblet op på et netværk, og brugere af et sådan system er ikke nødvendigvis altid bevidste om, at de benytter en rekonfigurerbar enhed.



Figur 1.2: Co-processor arkitektur bestående af flere RU's i et netværk

Hvis man ønsker en *single-chip* implementering af et dynamisk rekonfigurerbart system, er det hovedsageligt to af konfigureringerne fra figur 1.1, der kan benyttes. Den rekonfigurerbare enhed kan implementeres som en funktionel enhed i mikroprocessorens datavej, eller der kan benytte et co-processor design. Det sidste design giver større frihed og giver i højere grad mulighed for at udnytte potentialet i rekonfigurerbart hardware fuldt ud, fordi *host*-processoren har mulighed for at sende regne intensive opgaver over til co-processoren, hvor dedikeret hardware kan øge udførelseshastigheden, mens *host*-processoren foretager andre opgaver. Co-processoren kan endvidere have en arkitektur, der er bygget op af flere rekonfigurerbare enheder i et netværk on chip (NoC), som vist på figur 1.2. Dette giver mulighed, for at flere forskellige opgaver kan afvikles parallelt. Derudover gør det co-processoren yderst skalerbar i forhold til de platforme, som den kunne tænkes implementeret på.

Kapitel 2

Problemstilling

Der er mange nye områder at udforske ved udviklingen af rekonfigurerbare systemer. For det første om det er muligt at realisere et dynamisk rekonfigurerbart system på de platforme, der eksisterer i dag. Der tages udganspunkt i kommercielle FPGA'er, og derfor foretages en række undersøgelser omkring disse. Dette skal gerne give indblik i de muligheder og begrænsninger, som de kommercielle FPGA'er har, når det kommer til realiseringen af effektive rekonfigurerbare arkitekturer. Udover de problemstillinger, som hardwaren udgør, er der mange andre faktorer, der skal tages højde for, før et rekonfigurerbart system kan realiseres.

COSMOS er et simuleringsmiljø for co-processor-koblede rekonfigurerbare arkitekturer. *COSMOS* simuleringsmiljø omfatter en generel applikationsmodel og en arkitekturmodel, som tilsammen modellerer de dynamiske egenskaber ved rekonfigurerbare arkitekturer. For at undersøge hvilke algoritmer og strategier, der er nødvendige for at håndtere den dynamiske allokering implementeres disse i *COSMOS*, som er velegnet til eksperimenter af denne art. Dynamisk allokering i rekonfigurerbare systemer har en lang række egenskaber, som gør, at det kan være svært at finde den strategi/algoritme, som i alle tilfælde finder en optimal løsning.

Dynamisk og statisk allokering er generelt et NP-complete problem(NP står for *non-deterministic polynomial time*). Dette betyder, at det endnu ikke er muligt at konstruere en deterministisk algoritme, som kan løse problemet i polynomisk tid $O(n^k)$, hvor n er størrelsen på problemet, og k er en konstant. Der findes mange algoritmer til statisk allokering se ([3]) blandt andet heuristics, som kører i polynomisk tid, men som kun finder en suboptimal løsning. Heuristics kan benyttes til statisk allokering, fordi allokeringen ikke udføres *runtime*, og der er derfor meget tid og ofte stor computerkraft til rådighed. I dynamisk allokering er det nødvendigt at tænke anderledes, fordi der specielt er to faktorer, der skiller sig væsentligt ud i forhold til statisk allokering. For det første er der sjældent samme computerkraft til rådighed i et dynamisk rekonfigurerbart system, derudover har beregningstiden for allokeringsalgoritmen stor indflydelse på ydelsen for det samlede system. Der vil derfor gennem *COSMOS* blive afprøvet en række simple allokeringsstrategier for på den måde, at opnå en større forståelse af de vigtige faktorer i dynamiske rekonfigurerbare systemer.

Projektets målsætninger kan sammenfattes til følgende:

- Undersøge mulighederne for, at realisere et dynamisk rekonfigurerbart system på nutidens kommercielle FPGA'er.
- Opbygge en model som beskriver de dynamiske egenskaber ved et rekonfigurerbart system.
- Undersøge, hvilke faktorer, der gør sig gældende i et dynamisk rekonfigurerbart system.
- Sammenligne forskellige strategier til allokering og reallokering af applikationer i et dynamisk rekonfigurerbart system.

De følgende kapitler vil forsøge at besvare disse spørgsmål. Der lægges ud med en undersøgelse af de kommercielle FPGA'er og deres mulighed for at indgå i realiseringen af et dynamisk rekonfigurerbart system.

Kapitel 3

Hardware

I dette kapitel vil de kommercielle FPGA'er blive undersøgt for at konkludere om de i dag supporterer dynamiske rekonfigurerbare systemer. Der vil blive lagt vægt på at forstå, hvordan den partielle rekonfigurering kan finde sted, og hvilke metoder og værktøjer, der eksisterer, som eventuelle designere kan benytte ved partiel rekonfigurering. Gennemgangen vil tage udgangspunkt i en *Virtex-4*, men mange af de faktorer, der nævnes, gør sig gældende for hele *Virtex*-familien.

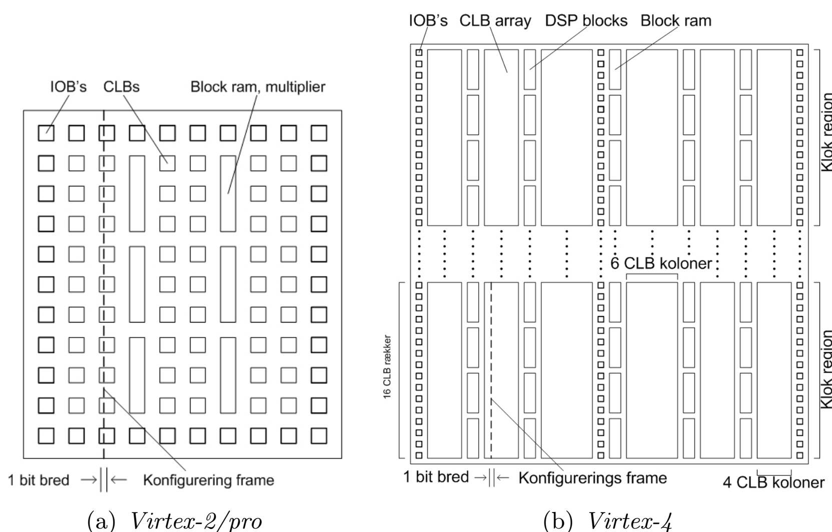
3.1 FPGA'ers arkitektur

Field Programable Gate Arrays (FPGA'ere) er opbygget omkring en række forskellige ressourcer. En *Virtex-4* har blandt andet dedikerede klokressourcer, som benyttes til at distribuere klokken rundt på chippen uden *clock skew*. Derudover er der blokram, forskellige *Digital signal processing* enheder (DSP), som er effektive ved implementeringen af *Finite Impule Response* (FIR) filtre. Den vigtigste ressource er dog en *Configurable Logic Block* (CLB), som benyttes til at implementere både sekventielle og

kombinatoriske kredsløb. Et CLB-element indeholder 4 forbundne *slices*, der blandt andet indeholder to *Look Up Tables* (LUT's) hver, som kan benyttes til at implementere en vilkårlig boolesk ligning af 4 input. En *Slice* indeholder også et register/latch, som kan initialiseres som distribueret ram [18].

3.1.1 Konfigureringshukommelse

Alle de programmerbare features en *Virtex-4* tilbyder er kontrolleret af flygtig hukommelse (konfigurationshukommelsen) og skal derfor altid konfigureres, når FPGA'en starter op. Konfigurationshukommelsen definerer de forskellige LUT-ligninger, signalernes interne rute, og alle andre aspekter fra en brugers design. *Virtex*-familiens (*Virtex, Virtex-2 og*



(a) *Virtex-2/pro* (b) *Virtex-4*

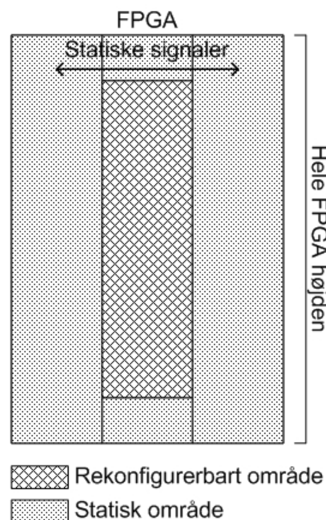
Figur 3.1: Konfigurations frame for to forskellige FPGA'er

Virtex-2 pro) konfigurationsarkitektur er beskrevet i *Xilinx* applikations note [19]. Konfigurationen bliver gemt i *Static Random Access Memory* (SRAM), som der kan læses og skrives fra uden at forstyrre FPGA'en. Den mindste enhed af konfigurationshukommelsen, der kan læses fra og

skrives til, kaldes en frame, og den spænder over hele FPGA'ens højde og er 1 bit bred se figur 3.1(a).

Virtex-4 famililien har medført en række betydningsfulde ændringer i arkitekturen. Som det er vist på figur 3.1(b), er konfigurationsarkitekturen stadig baseret på en frame, men i *Virtex-4* spænder en frame kun over 1 kolonne af 16 CLB's i stedet for hele højden af FPGA'en. Dette betyder, at der teoretisk opnås en meget større frihed i placeringen af de regioner, som skal være rekonfigurerbare. Klokdistribueringen er også ændret i forhold til tidligere modeller. Nu er klokregionerne også rettet ind efter blokke bestående af 16 CLB-række, hvorimod klokdistribueringens-regionerne i tidligere *Virtex*-familier var delt op i kvadranter [12].

Desuden har *Virtex-2/pro* og *Virtex-4* en ny feature (*glitchless dynamic reconfiguration*), dette betyder, at en konfigurationsbit, der har den samme værdi før og efter en konfiguration, ikke påvirker den ressource, som det pågældende konfigurationsbit kontrollerer. Det er derfor muligt, (se figur 3.2) at have en statisk region i samme kolonne, som en rekonfigurerbar region.



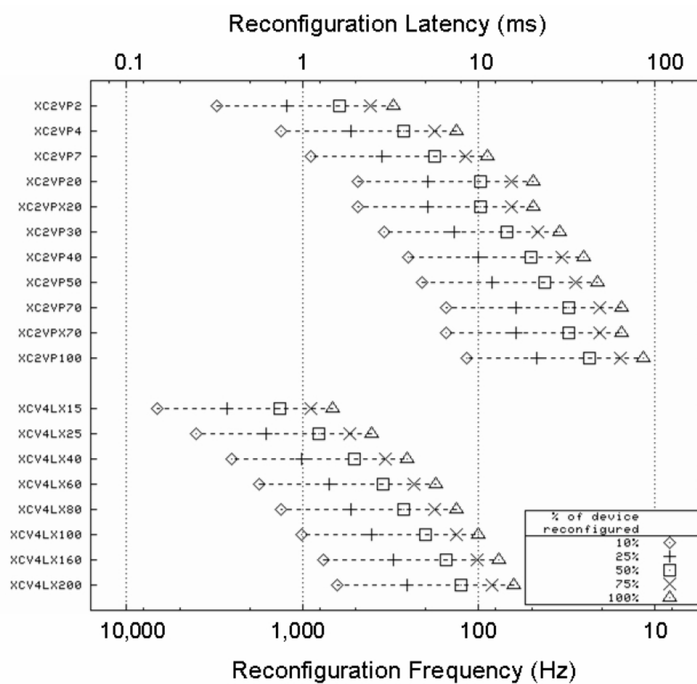
Figur 3.2: Statiske og rekonfigurerbare regioner for en *Virtex-2*

Det kræver blot at de konfigurationsbits, som er i den statiske region, altid bliver overskrevet med den samme værdi. Denne feature gælder også

for *Virtex-2/Pro*, med udtagelse af *LUT-ram* og *SRL16* (16 bits skifte register) primitiver, som ikke har denne egenskab. Derfor må de statiske områder se figur 3.2 i samme kolonne som en rekonfigurerbar region ikke indeholde *LUT-ram* og *SRL16* primitiver i *Virtex-2/pro*, dette gør sig ikke gældende for *Virtex-4*.

3.1.2 ICAP Interface

ICAP er et akronym for (Internal Configuration Access Port), ICAP blev første gang introduceret i forbindelse med *Virtex-2*. Dette interface eksisterer også i de nyere modeller *Virtex-4* og *Virtex-5*. ICAP giver adgang til at konfigurere FPGA'ens logik. Dette er den vigtigste faktor for at muliggøre rekonfiguration i dynamiske rekonfigurbare systemer baseret på FPGA'er. Når ICAP benyttes til rekonfigurering skal det gøres med forbehold, for det må naturligvis ikke ske at det kredsløb, som kontrollerer ICAP'en, rekonfigureres. Dette betyder, at det ikke er muligt at rekonfigurere hele FPGA'en, men derimod kun er muligt at udføre partiel rekonfigurering via ICAP. ICAP-interfacet er en simplificeret udgave af SelectMap-interfacet, hvilket betyder, at processen med at rekonfigurere FPGA'en følger den procedure, som SelectMap-interfacet benytter, dette er beskrevet i detaljer i [17]. Fordi ICAP-interfacet kun skal understøtte partiel rekonfigurering, er interfacet som nævnt en del simplere end SelectMap. I *Virtex-2/pro* benytter ICAP interfacet en busbredde på maksimalt 8 bit, hvorimod *Virtex-4* kan benytte en bus på maksimalt 32 bit. Dette giver en større båndbredde og dermed en hurtigere konfigurering [9]. På figur 3.3 ses rekonfigureringshastigheden for *Virtex-4*- og *Virtex-2*-familien, det ses tydeligt, at den større båndbredde til ICAP-interfacet i *Virtex-4*-familien sænker rekonfigureringstiden. Forbedringen i rekonfigureringshastigheden skal også ses i lyset af, at *Virtex-4*-familien i gennemsnit har mere logik, der skal konfigureres end *Virtex-2*. For dynamiske rekonfigurerbare systemer er der ingen krav til rekonfigureringshastigheden, men den kan have stor indflydelse på systemets performance, hvis den bliver meget stor. Som det ses af figur 3.3 er tiden for en partiel rekonfigurering af 10% af den samlede størrelse for en *Virtex-4 LX25* gennem ICAP cirka 0.5 ms hvilket er meget lang tid inden for moderne computer systemer.



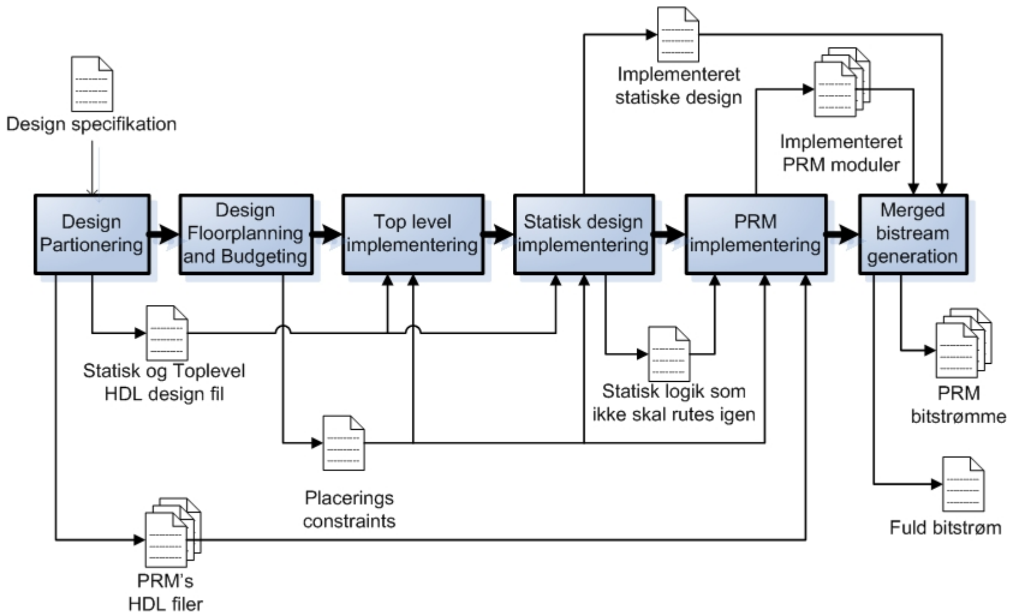
Figur 3.3: Rekonfigureringshastigheden sammenlignet mellem *Virtex-4*- og *Virtex-2*-familien fra [9]

3.2 Xilinx dynamiske rekonfigureringsmetoder

Xilinx har dokumenteret to forskellige designmetoder til dynamisk rekonfigurering af deres FPGA'er, kaldet modulbaseret og differensbaseret rekonfigurering [16]. Oprindeligt var den modulbaserede designmetode beregnet til design af store systemer, hvor flere forskellige grupper af ingeniører kunne arbejde på hver deres modul. I denne designmetode opdeles FPGA'en i moduler, som enten er statiske eller rekonfigurerbare. De rekonfigurerbare moduler, der deler samme rekonfigurerbare region på FPGA'en, skal have det samme interface. Den differensbaserede designmetode egner sig bedst til små systemer, fordi det i høj grad går ud på at udføre små ændringer i et design, og derefter skabe en bitstrøm, som er baseret på den differens, der er mellem det gamle og det nye design. Dette skaber små bitstrømme, og skiftet mellem de forskellige designs kan derfor foretages hurtigt, fordi det er en meget begrænset mængde data, der skal overføres til konfigurationshukommelsen.

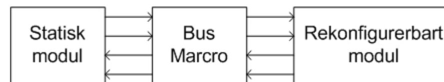
3.2.1 Modulbaseret rekonfigurerbar designmetode

I et modulbaseret design defineres bestemte områder af FPGA'en, som kan rekonfigureres, mens resten af FPGA'en stadig er aktiv. På figur 3.4 er vist, hvordan designprocessen for et modulbaseret rekonfigurerbart system ser ud. Processen ligner meget den, der gennemgås, når et modulbaseret system designes, der er dog nogle få ændringer. Den vigtigste er det 4. trin (se figur 3.4), hvor designets statisk logik bliver *placed and routed*, og der generes en fil, som indeholder information omkring placeringen af den statiske logik . Når de partielle rekonfigurerbare moduler (PRM) skal implementeres benyttes informationen omkring den statiske logik , således at denne ikke blive *placed and routed* igen, når de partielle moduler implementeres. F.eks. er det vigtigt, at bus-macro'erne, som håndterer kommunikationen mellem forskellige moduler ikke bliver flyttet, når de forskellige partielle rekonfigurerbare moduler implementeres. Når der kommunikeres mellem et statisk modul og et rekonfigurerbart modul er det som nævnt nødvendigt med en bus-macro se figur 3.5. En bus-macro har til opgave at sørge for, at der altid er en statisk signal-rute mellem interfacet på det rekonfigurerbare modul og det statiske modul.



Figur 3.4: Modulbaseret rekonfigurerbar designmetode

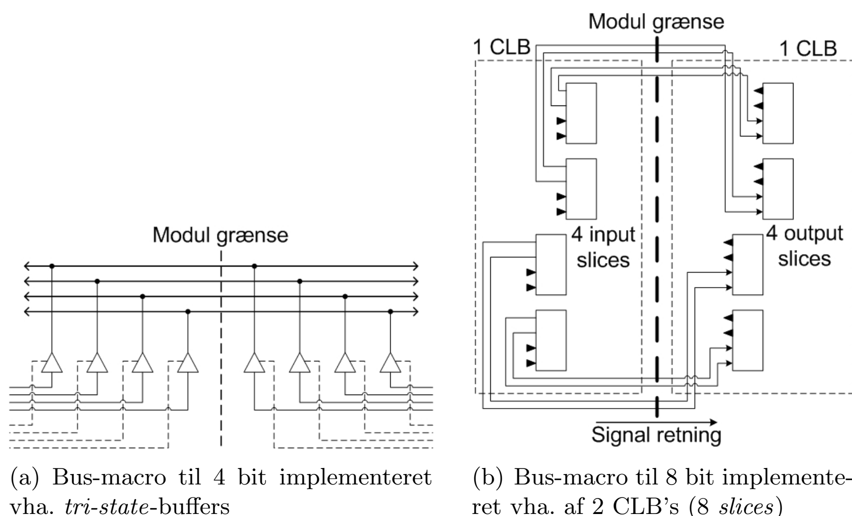
Dette betyder, at de bus-macro'er som instantieres for at kommunikere



Figur 3.5: Bus-macro mellem 2 moduler, de kan i princippet begge være rekonfigurerbare

mellem statiske og rekonfigurerbare moduler ikke kan flyttes, dette gør sig også gældende inden for det rekonfigurerbare område. En bus-macro kan implementeres på forskellige måder, i [16] er den baseret på *tri-state-buffers*, som vist på figur 3.6(a). Dette giver imidlertid anledning til forskellige problemer, blandt andet er *tri-state-bufferne* distribueret ud over FPGA'en, hvilket sætter begrænsninger for, hvor bus-macro'erne kan placeres, og det medfører igen begrænsninger på kommunikationen imellem to moduler. På figur 3.6(b) er vist en anden implementering af en bus-macro [14]. Denne er implementeret vha. af to CLB's en på hver side af grænsen mellem den statiske og den rekonfigurerbare region. Inden for hver CLB er der som nævnt (se afsnit 3.1) 4 *slices*, og en *slice* kan benyt-

tes til at implementere 2 en-bits ensrettede signaler, hvilket betyder, at to CLB's kan implementere 8 en-bits signaler, som vist på figur 3.6(b). Hvert signal benytter en LUT implementeret som en buffer, dvs et input og et output, hvor outputtet altid har samme værdi som inputtet. Denne nye implementering af bus-macro'en giver i høj grad en stor frihed for omfanget af kommunikationen og placeringen af bus-macro'erne. Det gælder for begge de to implementeringer, at bus-macro'erne kan instantieres således, at signalerne enten går fra venstre mod højre eller omvendt. Når først signalretningen er valgt, kan denne ikke ændres, dvs. et bus-macro-signal må hverken være bidirektionalt eller rekonfigurerbart.



(a) Bus-macro til 4 bit implementeret vha. *tri-state*-buffers

(b) Bus-macro til 8 bit implementeret vha. af 2 CLB's (8 *slices*)

Figur 3.6: Fysisk implementering af to forskellige bus-macro'er

3.2.2 Differensbaseret rekonfigurerbar designmetode

Denne metode er forholdsvis upraktisk for større designs. Princippet bag metoden er at sammenligne to forskellige designs, som er syntesiteret og *placed and routed*. Ud fra hvert design generes en fuld bitstrøm, og derefter dannes en partiel bitstrøm ud fra forskellen mellem de to hele bitstrømme. Der er fundamentalt to måder at ændre designet på. Det kan

gøres *front end*, hvilket vil sige, at der laves en ændring i HDL (*Hardware Description Language*)-koden og hele designet syntesiteres og implementeres igen. Derefter bliver det nye og gamle design sammenlignet, og der generes en bitstrøm ud fra forskellen mellem dem. Den anden metode *back end* benytter FPGA-editoren til at editere i et design, som er syntesiteret og *placed and routed.* Dette er meget besværligt og kan kun anbefales ved meget små ændringer. Fordelene ved differensmetoden er, at der kan skiftes mellem to designs meget hurtigt, hvis de grundlæggende ligner hinanden, fordi bitstrømmen, der skal programmere konfigurationshukommelsen, kun består af forskellen mellem de to oprindelige designs bitstrømme.

3.3 Egne forsøg

For at undersøge om dynamiske rekonfigurerbare systemer kan implementeres på kommercielle FPGA'er er der foretaget en række simple forsøg. Dette skal gerne give en indikation af deres muligheder/mangler omkring implementeringen af dynamiske rekonfigurerbare systemer. Derudover vil de værktøjer (*ISE* og *Xilinx Platform Studio*, *Early-access Partial Reconfiguration* samt *PlanAhead*), som understøtter rekonfigurering ligeledes blive undersøgt. Til forsøgene er der benyttet et evaluerings-board *ML401* fra *Xilinx* med en *Virtex-4* (se [15]).

3.3.1 Værktøjerne

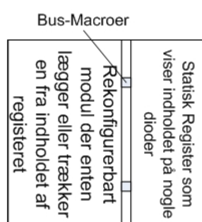
For at designe og implementere dynamiske rekonfigurerbare systemer er det i høj grad vigtigt at have de rette værktøjer. *Xilinx* ISE version 8.2i kan som den eneste version opdateres med en pakke *Early-access Partial Reconfiguration*, som gør, at det er muligt at syntesitere og implementere et partielt rekonfigurerbart design [14]. Pakken *Early-access Partial Reconfiguration* opdaterer blandt andet funktionen *placed and routed*, så denne genererer en fil, der indeholder information omkring den statiske logik, der benyttes i designet. Opdateringen af ISE med *Early-access Partial Reconfiguration* gør det muligt at følge det design-*flow*, som er vist på figur 3.4. Et af de centrale punkter i et rekonfigurerbart design er at

skabe en *constraint*-fil, som blandt andet specificerer i detaljer, hvor på FPGA'en de forskellige partielle rekonfigurerbare regioner befinder sig, bus-macro'ernes placering og pin-konfigurationen. Denne funktion kan for mindre rekonfigurerbare designs varetages af programmet *Floorplanner*, som er en del ISE. Til større systemer har *Xilinx* udviklet programmet *Planahead*. *Xilinx* har udviklet en række *slice*-baserede bus-macro'er, som følger med *Early-access Partial Reconfiguration*. Bus-macro'erne er implementeret, som vist på figur 3.6(b), disse er "hårde macro'er", hvilket betyder, at de består af primitiver fra FPGA'en, i dette tilfælde LUT's. I det følgende vil der gennem et simpelt forsøg opnås større kendskab til værktøjerne og brugen af dem.

3.3.2 Modulbaseret forsøg

Forsøg 1

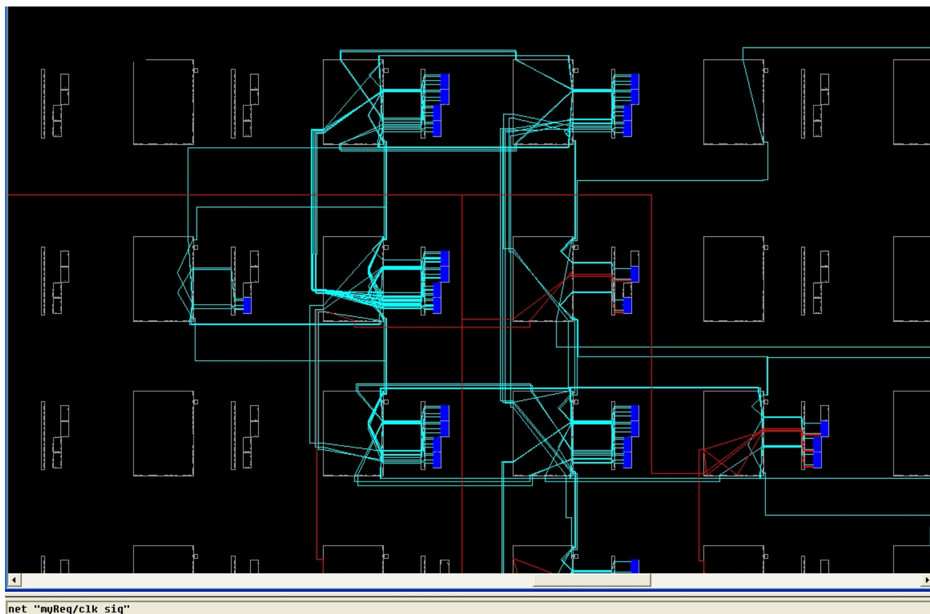
I forbindelse med undersøgelsen af de kommercielle FPGA'er er der lavet et simpelt rekonfigurerbart moduldesign, som vist på figur 3.7. Som det ses, består designet af to moduler, hvor det ene kan rekonfigureres, og det andet er statisk. Det statiske modul består af et register på 8 bit, som kan nulstilles. Det rekonfigurerbare modul lægger enten en til eller trækker en fra indholdet af registeret. Dette meget simple partielle rekon-



Figur 3.7: Simpelt rekonfigurerbart design

figurerbare design blev implementeret ved at følge det design-*flow*, som er vist på figur 3.4. Til den indledende fase, hvor designet skal deles op i moduler og placeres på FPGA'en, blev programmet *Floorplanner* benyttet. Programmet letter betydeligt opgaven med at danne en *constraint* fil, som indeholder information omkring pin-konfigurationen, de statiske-

og rekonfigurerbare-moduler samt bus-macro'ernes placering. Når designet er implementeret og *placed and routed*, er det vigtigt, at inspicere det implementerede design i f.eks. *fpga_editoren*, som er en del af *ISE*. Dette skal gøres, for at undersøge, om designet overholder de restriktioner, som blev specificeret i *constraint* filen. På figur 3.8 er vist et billede fra *fpga_editoren*, hvor man blandt andet kan se de to bus-macro'ers placering, og hvorledes der er en "ultimativ"grænse mellem de to moduler, således at kun signaler, som går gennem en bus-macro, kan krydse denne grænse. Kloksignalet er markeret med rødt på figuren, og det ses, at klokken godt kan krydse grænsen mellem de rekonfigurerbare regioner uden brug af bus-macro'er. Dette skyldes, at klokken som det eneste signal har specielle ressourcer til rådighed blandt andet for at undgå *clock screw*. Klokken er ligeledes det eneste input-signal en rekonfigurerbar enhed kan have, som ikke passerer gennem en bus-macro. FPGA'en blev i dette forsøg programmeret gennem JTAG interfacet, hvilket betyder, at det ikke er FPGA'en selv, der kontrollerer rekonfigureringen gennem ICAP. Udførelsen af dette forsøg gav et godt indblik i de værktøjer, som benyt-



Figur 3.8: Billedet af det simple rekonfigurerbare design fra *f-pga_editoren*

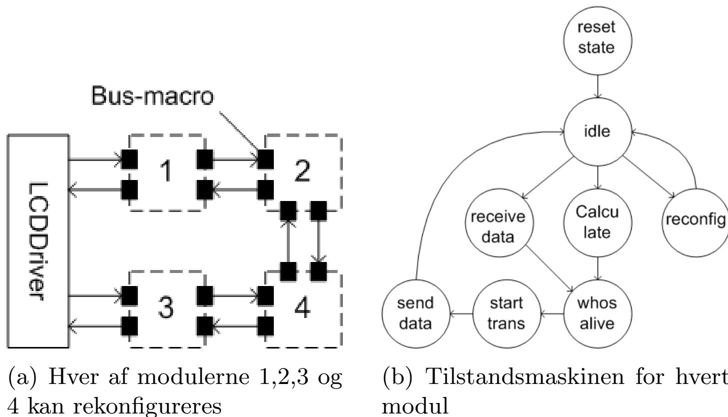
tes, når der skal designes et rekonfigurerbart system. Desuden gav dette forsøg en mulighed for at afprøve den designmetode, som er vist på figur 3.4, hvilket må konkluderes at være den rigtige fremgangsmåde, når der skal implementeres et rekonfigurerbart system. Det næste forsøg vil fokusere mere på FPGA'en, og de muligheder/begrænsninger den tilbyder i forbindelse med partiel rekonfigurering.

Forsøg 2

Der er foretaget en nærmere undersøgelse af FPGA'en for at belyse hvilke, muligheder den tilbyder, hvis den skal benyttes til at implementere en dynamisk rekonfigurerbar co-processor, som den vist på figur 1.2. Ideen er, at co-procssoren består af en række RU's (rekonfigurerbare enheder) sat sammen i et netværk. De rekonfigurerbare enheder kan hver især implementerer en vilkårlig funktion i dedikeret hardware, som kan optimere udførelsen af en given opgave. Der er en række centrale områder, der gør sig gældende for at realisere et sådan system på en FPGA, og de kan sammenfattes i følgende punkter:

- Muligheden for at rekonfigurere en RU uden at påvirke de andre.
- Undersøge den opdaterede arkitektur for *Virtex-4*, blandt andet om det er muligt at placere flere rekonfigurerbare moduler i samme kolonne på FPGA'en.
- Muligheden for at kommunikere mellem de forskellige RU's.
- Tilstanden af hukommelseselementerne bliver bevaret under rekonfigurering.

For at belyse disse punkter er der designet et mere komplekst rekonfigurerbart system. Figur 3.9(a) viser, hvordan systemet er opbygget i fem hovedmoduler koblet sammen i et netværk. *LCDdriveren* modtager pakker fra de 4 enheder og skriver resultatet ud på et *LCDdisplay* således, at det er muligt at se status for hvert enkelt modul. Modulerne er grundlæggende ens og svarer hver til en RU, de er bygget op omkring den tilstandsmaskine, der er vist på figur 3.9(b). I tilstanden *idle*, ventes på tre

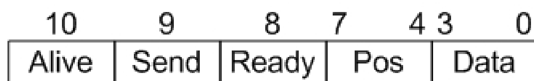


(a) Hver af modulerne 1,2,3 og 4 kan rekonfigureres

(b) Tilstandsmaskinen for hvert modul

Figur 3.9: Designet af et dynamisk rekonfigurerbart system

aktioner. Det kan være et eksternt input, som specificerer, at der skal beregnes et nyt resultat. Dette gøres i tilstanden *calculate*, hvor der alt efter hvilken konfiguration den rekonfigurerbare enhed har trækkes 1 fra eller ligges 1 til det gamle resultat. Det nye resultatet bliver sendt til *LDC-driveren* således, at det kan observeres, dette gøres gennem tilstandene *whos_alive*, *start_trans* og *send_data*. Tilstanden *whos_alive* undersøger, hvilke naboer i netværket, der er i live dvs. ikke pt. er i gang med en eventuel rekonfigurering. Tilstanden *start_trans* venter på at modtageren er klar, denne kan være optaget af f.eks. en anden pakkemodtagelse eller kan selv være i gang med at udføre en beregning. Når modtageren er klar sendes data i tilstanden *send_data*. Den anden aktion, der ventes på i tilstanden *idle*, er modtagelsen af en pakke. Hvis denne accepteres sendes den øjeblikkeligt videre ved at gennemgå samme procedure for afsendelse af pakker. Den sidste tilstand, der kan nås fra *idle* er tilstanden *reconfig*. Dette sker når enheden skal rekonfigureres og får besked herom. Det nuværende resultat for enheden gemmes i et register således, at det kan benyttes efter rekonfigureringen. Derefter signaleres der til omgivelserne, at enheden ikke længere er i live. Kommunikationen i netværket foregår i et simpelt pakkeformat bestående af 10 bit, dette kan ses på figur 3.10. Felterne giver sig selv på nær *pos*, dette er 4 bit, som indeholder information om, hvor pakken har været i netværket. Hvert modul sætter altid det samme bit i feltet *pos* inden en pakke afsendes, på den måde kan



Figur 3.10: Pakkeformatet, som de rekonfigurerbare enheder bruger i deres interne kommunikation

pakkens rute skrives ud på *LCDdisplay*'et sammen med data. Den fulde kode for implementeringen kan findes i bilag G. De 4 rekonfigurerbare moduler, der er vist på figur 3.9(a), er placeret på samme måde på F-PGA'en dvs. 1 og 3 og 2 og 4 er i samme kolonne se bilag A.1 for et *screenshot* fra programmet *floorplanner*. I bilag A.2 kan ses, hvordan den statiske logik er placeret og i bilag A.3 ses den samlede implementering af systemet med både den statiske logik og de rekonfigurerbare moduler. Dette eksperiment resulterede i mange interessante aspekter vedrørende implementeringen af dynamiske rekonfigurerbare systemer på en *Virtex-4*. Den opdaterede arkitektur i *Virtex-4*, hvor der skrives til konfigurationshukommelsen i frames af størrelsen 16 CLB's i stedet for frames, der spænder over hele FPGA'ens højde, har givet en stor frihed i placeringen af modulerne, og givet mulighed for at opdele FPGA'en i flere rekonfigurerbare enheder. Kommunikationen mellem de rekonfigurerbare enheder sker vha. bus-macro'er og fungerer upåklageligt. Det skal dog nævnes, at et større netværk af rekonfigurerbare enheder på samme FPGA bruger mange CLB's til implementeringen af bus-macro'erne der håndterer kommunikationen, og dermed kan de ikke benyttes til implementeringen af logik og distribueret ram. Tages der udgangspunkt i en co-processor bestående af et 4x4 netværk af rekonfigurerbare enheder, som hver kommunikerer via en 32 bits bus til deres nabo, vil bus-macro'erne alene optage cirka 15 % af det samlede antal CLB's for en *Virtex-4 LX25* og cirka 2 % af det samlede antal for en *Virtex4 LX200* [18]. Et anden interessant aspekt er hukommelseselementernes tilstand, som ifølge [16] bliver bevaret, selvom den region, hvori hukommelseselementet befinder sig, gennemgår rekonfiguration. Forsøget udnytter dette faktum på forskellige måder for at undersøge, om det også fungerer i praksis. Hver gang et af de rekonfigurerbare moduler skal rekonfigureres, går tilstandsmaskinen til tilstanden *recon* jf. figur 3.9(b). Det betyder, at tilstandsregisteret altid indeholder tilstanden *recon* før rekonfiguration, og det viser sig, at det nye modul altid starter op i tilstanden *recon*. Derudover bliver registret, der indeholder resultatet af den beregning, hvert modul foretager

ligeledes bevaret under rekonfigureringen således, at det nye modul kan arbejde videre ud fra samme data som det foregående modul.

3.3.3 Sammenfatning

Arkitekturen for *Virtex-4*, samt konfigurationshukommelsen er designet således, at der opnås en større frihed i valget af partielle rekonfigurerbare regioner, dette er vist gennem de forsøg, der er udført, hvor de rekonfigurerbare enheder blandt andet ligger i samme kolonne, se bilag A.3. Andre har foretaget en del forsøg med FPGA'er og partiel dynamisk rekonfigurering (se [12]), men de fleste kommer til samme resultat. Det er muligt at lave et simpelt dynamisk rekonfigurertbart system, hvor f.eks. et modul kan udskiftes, og hvor de udskiftelige moduler, har samme interface. De fleste forsøg viser dog stadig, at den tid det tager, at rekonfigurere FPGA'en gennem ICAP er for stor og skal forbedres, hvis ydelsen af coprocessoren skal stå mål med de traditionelle arkitekturer, der findes i dag jf. figur 3.3. Derudover betyder den kolonnebaserede konfiguration for *Virtex-2/pro*, at der er mange begrænsninger for den fysiske placering af de rekonfigurerbare regioner. Endvidere gælder der for *Virtex-4*-familien at de partielle rekonfigurerbare regioner skal undgå at krydse den midterste kolonne, fordi alle primitivere i denne kolonne derved skal instantieres i det statiske design, dette gælder alle input-pindene, DCM(*Digital Clock Manager*) og bufferne [14]. Disse fysiske begrænsninger gør, at det ikke pt. er muligt med de kommercielle FPGA'er, der er til rådighed, at implementere et on-chip komplekst rekonfigurerbart system bestående af mange rekonfigurerbare regioner. De systemer der kan implementeres, er som vist simple, og der er lang vej til et dynamisk rekonfigurerbart system, der er konkurrencedygtig med de traditionelle computerarkitekturer. En af de helt centrale områder, der skal forbedres, er som nævnt rekonfigureringstiden, denne er stadig alt for lang, hvis der skal drages nytte af rekonfigurerbare arkitekturer. En anden designmulighed, der ikke kræver flere partielle rekonfigurerbare regioner på den samme FPGA, er at koble flere FPGA'er sammen i netværk og lade hver selvstændige FPGA udgøre en rekonfigurerbar enhed. Dette introducerer et nyt stort overhead i form af off-chip kommunikation, som uundgåeligt vil være til stede. Potentialet for dynamiske rekonfigurerbare systemer er dog stadig til stede, selvom der ikke aktuelt findes platforme, der er i stand til

at understøtte avancerede rekonfigurerbare systemer. For at få et bedre indblik i de problemstillinger, som dynamiske rekonfigurerbare systemer introducerer, designes der en model, som forsøger at belyse aspekterne omkring disse systemer.

KAPITEL 4

COSMOS modellen

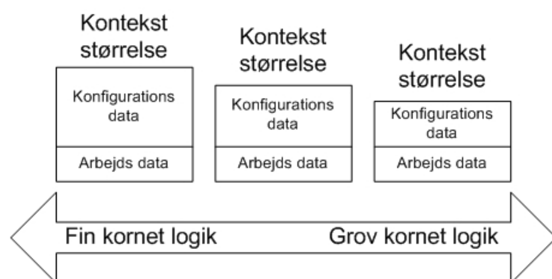
I dette kapitel beskrives arbejdet, der er udført i forbindelse med modellen COSMOS. COSMOS er et simuleringsmiljø for co-processor-koblede rekonfigurerbare arkitekturer, der blev introduceret i [13] [7]. Først foretages en analyse af de elementer, som er vigtige at modellere i et dynamisk rekonfigurerbart system, hvorefter der gives en mere detaljeret beskrivelse af opbygningen af modellen.

4.1 Analyse

For at opnå en større forståelse af rekonfigurerbare systemer og den dynamiske opførsel, er det oplagt at designe og implementere en model, som kan give et indblik i dette. Før en model implementeres, er det vigtigt at have en ide om hvilke faktorer, der spiller en vigtig rolle, og hvilke der kan ses bort fra eller modelleres på et højere abstraktionsniveau i det system, man vil simulere. I afsnit 1.2 blev der omtalt 4 metoder, hvorpå en rekonfigurerbar enhed (RU) kan kobles med en *host*-mikroprocessor. Modellen skal simulere et rekonfigurerbart design, som er tæt koblet med

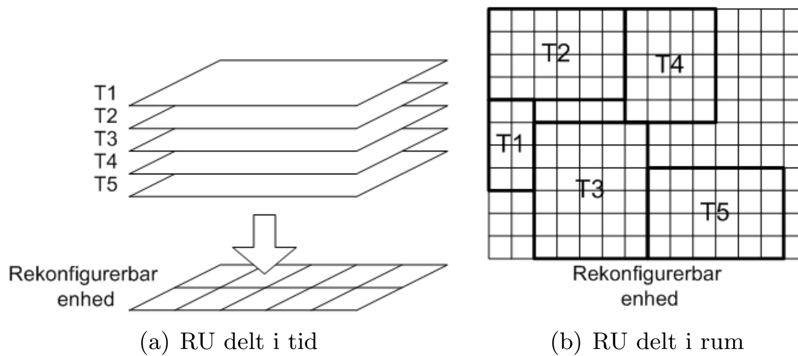
host-mikroprocessoren. Det aktraktive ved et co-processor design i forhold til en rekonfigurerbar funktionel enhed er, at co-processoren kan designes, så denne er skalerbar. Det vil bevirke, at det i fremtiden vil være muligt at udnytte flere af de ressourcer, som FPGA'en stiller til rådighed.

Når en rekonfigurerbar enhed behandler en ny opgave, skal der skiftes kontekst (den rekonfigurerbare enhed skal omprogrammeres, og det ny data, der skal behandles, skal klargøres), dette vil uundgåeligt introducere et *overhead*. Størrelsen af dette overhead er bestemt af mange fysiske faktorer, størrelsen på den RU, som skal omprogrammeres, båndbredden til konfigurationshukommelsen jf. figur 3.3, den logiske *granularity* samt den mængde data, som skal benyttes under udførelsen af opgaven. Som vist på figur 4.1 har den logiske *granularity* direkte indflydelse på størrelsen af den bitstrøm, som den rekonfigurerbare enhed skal konfigureres med.



Figur 4.1: Logiske *granularity* og betydningen for kontekststørrelsen

Grovkornet logik består hovedsageligt af større elementer, såsom ALU'er, registerfiler, multiplicer- og divisions-enheder, i det hele taget komponenter, der manipulerer på mange bits ad gangen. Dette medfører, at applikationer, som i høj grad drager nytte af meget simple bitmanipuleringer, ikke udføres optimalt på en RU bestående af grovkornet logik. Tilgengæld reduceres antallet af bits, som skal bruges til at omprogrammere FPGA'en. Dette skyldes, at der ikke skal konfigurationsbits til hver enkelt LUT, men kun til de større enheder. Den tid, det tager at skifte konteksten for en RU, afhænger i høj grad, af det rekonfigurerbare systems overordnede design. For at opnå en model der afspejler de forskellige designvalg skal kontekst-skiftetiden simuleres.



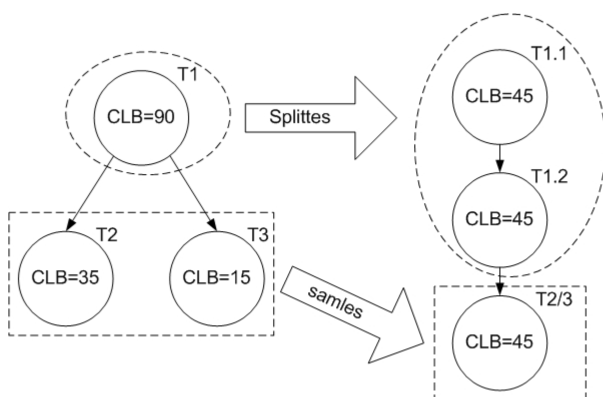
(a) RU delt i tid (b) RU delt i rum

Figur 4.2: To forskellige designs af en rekonfigurerbar enhed

En anden vigtig faktor er opbygningen af den rekonfigurerbare enhed, og hvordan den skal håndtere udførelsen af de tasks, som den rekonfigurerbare enhed får allokeret. Udførelsen af task'ene kan grundlæggende ske på to måder, som vist på figur 4.2. Den ene foregår ved, at task'ene deler den rekonfigurerbare enhed i tid, dvs. de udføres enkeltvis på RU'en ifølge en strategi, som er fastlagt på forhånd. Denne strategi kræver, at den rekonfigurerbare enhed er i stand til at udføre hurtige kontekstskift, så task'ene ikke oplever lange ventetider når der skiftes mellem dem. Kommercielle FPGA'er supporterer ikke hurtige kontekstskift i deres nuværende arkitektur. I [11] er der designet et rekonfigurerbart system bestående af en FPGA med optil 8 kontekster. Der kan skiftes mellem hver kontekst med en forsinkelse på kun 30 ns, til gengæld for det hurtige kontekstskifte anvendes meget af FPGA'ens interne hukommelse til opbevaring af konfigurationsdata for de forskellige kontekster. En anden måde at nedsætte kontekst-skiftetiden er ved at benytte grovkornet logik, dette kan skære væsentligt i antallet af konfigurations bits sammenlignet med finkornet logik. I COSMOS modellen antages det at det er muligt at foretage hurtige kontekstskift mellem forskellige tasks allokeret på den samme rekonfigurerbare enhed. En alternativ strategi består i, at den enkelte task deler RU'en i rummet se figur 4.2(b). Dette betyder, at de udføres parallelt, og nytilkomne tasks kan allokeres på RU'en, så længe denne har plads. Realistisk set, kan denne metode være svær at implementere, fordi f.eks. FPGA'er, som håndterer rekonfigureringen, på forhånd er delt op i regioner, der enten er statiske eller rekonfigurerbare. De rekonfigurer-

bare regioner afgrænser et område, som altid omprogrammeres, hvilket besværliggør, at to tasks kan dele den samme partielle rekonfigurerbare region se afsnit 3.1. En anden løsning vil være en form for hybrid mellem de nævnte opbygninger af den rekonfigurerbare enhed, der er vist på figur 4.2. Denne hybrid består af mange RU's sat sammen i et netværk. Hvilket giver mulighed for, at flere tasks kan dele de mange RU'er i rummet og hver enkelt i tiden.

Det antages at de rekonfigurerbare enheder i co-processoren er homogene, så alle tasks kan allokeres til en vilkårlig RU. Fordi alle de rekonfigurerbare enheder er lige store, stilles der krav til de applikationer, som skal udføres på co-processoren. Som vist på figur 4.3 skal applikationer være tilpasset arkitekturen. Har hver RU f.eks. 50 CLB's, til rådighed kan det være nødvendigt at splitte store opgaver til mindre tasks. Desuden kan mindre tasks samles så længe den samlede størrelse ikke overskrider 50 CLB's. Det antages, at de applikationer som skal afvikles på co-procesooren, har gennemgået statisk analyse og en transformering, hvis dette er nødvendigt, så de kan allokeres til de homogene rekonfigurerbare enheder. Fordi co-procesooren består af et netværk af homogene ressourcer, skal *runtime* manageren ikke forholde sig til hvilke ressourcer, der optimerer en given task, hvilket udgør en central problemstilling i statisk allokering [6] [3]. For at modellere en applikation kan der benyttes

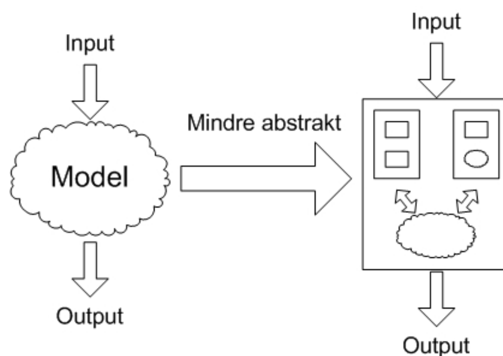


Figur 4.3: Illustration af hvordan en opgaven kan splittes til flere tasks eller samles i en stor task

en orienteret graf, hvor noderne udgør tasks, og kanterne i grafen udgør

kommunikationsopgaverne. Dette betyder, at tasks, der stadig har forældre i applikationen, som ikke er udført, ikke kan starte, fordi der er en dataafhængighed. Kommunikationsopgaverne skal have et grundlag for at afvikles i form af et netværk, ellers kan det forhindre tasks i at starte, hvis der findes dataafhængigheder, som ikke er løst. Et af de centrale elementer i dynamiske rekonfigurerbare systemer er muligheden for at reallokere tasks mellem ressourcerne. Det er muligt f.eks. at reallokere tasks fra lavere prioriterede applikationer for at frigøre rekonfigurerbare enheder, således at højere prioriterede tasks kan afvikles.

En model kan generelt implementeres på mange forskellige abstraktionsniveauer. Et af de vigtigste aspekter i opbygningen af en model er at finde de faktorer, som er vigtige for at få en præcis simulering af det faktiske system. Desuden er det væsentligt at finde de faktorer, som kan negligeres eller abstraheres fra uden, at det har den store indvirkning på modellens evne til at simulere det virkelige system. Som vist på figur 4.4 kan en model opbygges på forskellige abstraktionsniveauer, en meget abstrakt model giver normalt en forholdsvis simpel model, som vil være let at implementere. En abstrakt model giver færre detaljer, hvilket



Figur 4.4: Illustration af hvordan en model kan opbygges på forskellige abstraktionsniveauer

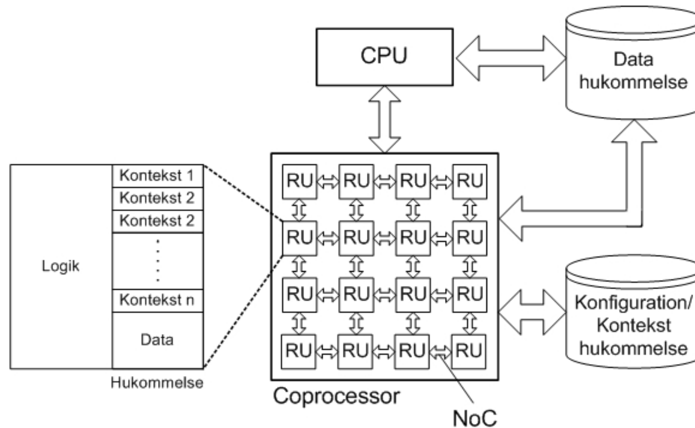
gør, at det er kompliceret at foreslå optimeringer af den abstrakte model, som ikke ændrer den drastisk. Forstået på den måde, at en ændring i en abstrakt model kan være svær at overføre til det fysiske system, som modelleres. Derimod kan en model opbygget af komponenter på flere forskellige abstraktionsniveauer være fordelagtigt, fordi det giver mulighed

for, at vigtige faktorer fra det faktiske system modelleres præcist. COSMOS modellen er opbygget omkring komponenter, der hver især har et forskelligt abstraktionsniveau i forhold til det virkelige rekonfigurerbare system. I dette afsnit er gennemgået en række faktorer, som modellen skal tage højde for, i det næste afsnit beskrives modellen i detaljer.

4.2 Generel beskrivelse

Modellen er opbygget i *SystemC*, et sprog der kan beskrive hardware *Hardware Description Language* (HDL), men som også kan bruges til at beskrive hele systemer, der skal modelleres. *SystemC* gør det muligt at simulere flere parallelle processer beskrevet i C++ vha. forskellige biblioteker, rutiner og macro'er ligeledes implementeret i C++. *Transaction-level modeling* (TLM) er en måde, hvorpå man kan modellere et digitalt system, hvor detaljerne af kommunikationen mellem modulerne er separeret fra modulets egentlige implementering. *SystemC* er specielt velegnet til denne form for modellering, fordi det tilbyder flere måder, hvorpå moduler kan kommunikere igennem abstrakte kanaler. COSMOS er designet som en TLM-model, hvor de forskellige moduler kommunikerer gennem abstrakte kanaler. I COSMOS modellen specificeres tid ved klokperioder. Klokken kan ikke sammenlignes med en tidshorisont, det er antallet af klokperioder, der har betydning. På denne måde kan forskellige tider specificeres vha. af antallet af klokperioder, som de bruger, f.eks. task'enes eksekveringstid, kommunikationsopgavernes køretid og den tid, der benyttes på reallokering. Det helt centrale for en realistisk model er, at forholdet mellem antallet af klokperioder for de forskellige elementer i modellen er afstemt efter de virkelige forhold. Alle simuleringernes kørselstider specificeres i antallet af klokperioder ($t_{exe}(cc)$).

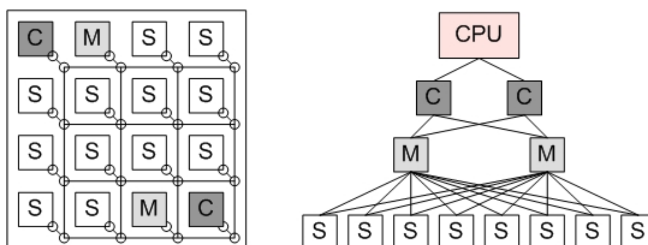
COSMOS er som nævnt et miljø, der kan simulere et co-processor-koblet rekonfigurerbart system, som det der er vist på figur 4.5. Dette system er opbygget af en *host*-mikroprocessor tæt koblet med en co-processor, der består af en række rekonfigurerbare enheder sat sammen ved hjælp af et netværk *on chip*. Arkitekturen indeholder datahukommelse, som kan tilgås direkte af både CPU'en og den rekonfigurerbare enhed, dette kan i mange situationer være med til at mindske den tid, det tager at overføre



Figur 4.5: Illustration af et rekonfigurerbart system, som COSMOS kan simulere

data, som skal behandles på co-processoren. Derudover er der en database, som består af forskellige konfigurationer, som de rekonfigurerbare enheder, hver i sær kan konfigureres med.

Den rekonfigurerbare enhed er opbygget af normal logik (LUT's, registre, multiplexere), som kan benyttes til at implementere de funktionaliteter en applikation har behov for. Desuden har en RU datahukommelse samt hukommelse, der indeholder konteksterne for de tasks, som er mappet til den givne RU. Således kan der foretages et hurtigt skifte mellem to tasks, uden RU'en først skal tilgå konfigurationsdatabasen. For arkitekturen vist på figur 4.5 er ressource-management et vigtigt punkt, fordi det kan have stor indflydelse på systemets samlede performance. Allokeringen af nye tasks og vedligeholdelsen af ressourcerne kan være en krævende opgave, specielt da den skal udføres løbende (*runtime*). For en lille co-processor kan denne opgave behandles af *host*-mikroprocessoren, men efterhånden som co-processoren vokser, kan denne opgave snart kræve alle CPU'ens ressourcer og i sidste ende kan CPU'en blive flaskehalsen for systemets samlede ydelse. For at undgå at *host*-mikroprocessoren skal belastes med håndteringen af ressourcerne og allokeringen af nye tasks, introducerer COSMOS en alternativ løsning. Denne består i at co-processoren selv håndterer *runtime*-allokeringen af nye tasks til de frie ressourcer. På figur 4.6 er vist, hvordan co-processorens rekonfigurerbare enheder er opdelt i



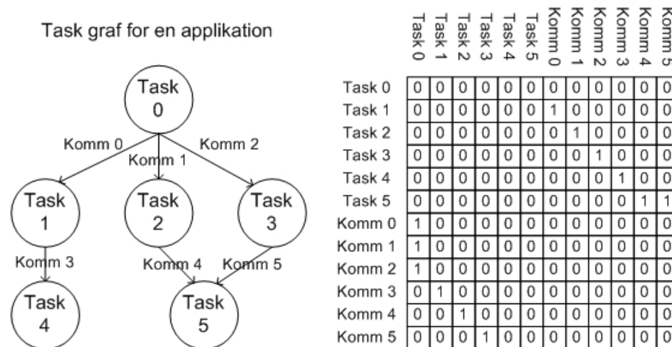
Figur 4.6: Illustration af organisationen af de rekonfigurerbare enheder i co-processoren

forskellige moduler, som hver især varetager en opgave i forbindelse med allokeringen af task'ene. Noderne i co-processoren kan være Koordinatorer (C-noder kommer af *Coordinator*) , Master noder (M-noder) eller Slave noder (S-noder). Noderne er struktureret i et hierarkisk design, som vist på figur 4.6, og kommunikerer kun med andre noder, som enten er forældre eller børn. I de følgende afsnit vil hver enkelt nodes funktionalitet blive beskrevet, og selve implementeringen vil også blive omtalt. Først kommer dog en introduktion til applikationsmodellen og den måde, hvorpå applikationerne håndteres i modellen.

4.2.1 Applikationer, tasks og kommunikationsopgaver

Inden, der kommer en mere detaljeret beskrivelse af modellen, er det vigtigt at forstå, hvordan en applikation er bygget op omkring en task-graf. På figur 4.7 er vist en applikation bestående af seks normale opgaver kaldet tasks og seks kommunikationsopgaver. På figuren ses også en tabel, som beskriver applikationens task-graf. Modellen oversætter applikationernes task-grafer til tabeller af denne form, hvor rækker og kolonner skal tolkes på følgende måde; en række indeholder et 1 tal hvis task'en i kolonnen er afhængig af task'en i rækken. Det er muligt at beskrive en vilkårlig task-graf vha. tabeller på denne form. En applikation består af en task-graf og desuden kørselstider for de enkelte tasks og kommunikationsopgaver, hvilket gør det muligt at specificere individuelle kørselstider for opgaverne inden for den samme applikation. Når en applikation skal allokeres til co-processoren, er det kun antallet af tasks, der specificerer hvor mange ressourcer, der skal være frie før co-processoren kan acceptere

applikationen. Kommunikationsopgaverne udføres af et *network on chip*, som de enkelte rekonfigurerbare enheder benytter til at kommunikere med, og optager derfor ikke ressourcer i form af RU's.



	Task 0	Task 1	Task 2	Task 3	Task 4	Task 5	Komm 0	Komm 1	Komm 2	Komm 3	Komm 4	Komm 5
Task 0	0	0	0	0	0	0	0	0	0	0	0	0
Task 1	0	0	0	0	0	0	1	0	0	0	0	0
Task 2	0	0	0	0	0	0	0	1	0	0	0	0
Task 3	0	0	0	0	0	0	0	0	1	0	0	0
Task 4	0	0	0	0	0	0	0	0	0	1	0	0
Task 5	0	0	0	0	0	0	0	0	0	0	1	1
Komm 0	1	0	0	0	0	0	0	0	0	0	0	0
Komm 1	1	0	0	0	0	0	0	0	0	0	0	0
Komm 2	1	0	0	0	0	0	0	0	0	0	0	0
Komm 3	0	1	0	0	0	0	0	0	0	0	0	0
Komm 4	0	0	1	0	0	0	0	0	0	0	0	0
Komm 5	0	0	0	1	0	0	0	0	0	0	0	0

Figur 4.7: En applikations task-graf

4.2.2 CPU'en

Dette modul har til formål at simulere CPU'en, hvis hovedopgave er at sende nye applikationer over til co-processoren, hvor de skal udføres. CPU'en er implementeret gennem en simpel tilstandsmaskine, som består af tre tilstande *try_app*, *wait_app_respons* og *cpu_finished*. I tilstanden *try_app* forsøger CPU'en at overføre en ny applikation til co-processoren, dette gøres ved at give koordinatoren besked om, at der er en ny applikation klar og informere om, hvor mange tasks applikationen består af. I tilstanden *wait_app_respons* venter CPU'en på, at koordinatoren undersøger, om der er plads til at allokere den indkomne applikation på co-processoren og rapporterer dette tilbage til CPU'en. Hvis der ikke er plads på co-processoren til den nye applikation, går CPU'en tilbage til tilstanden *try_app* og foretager en ny forespørgsel på den samme applikation. På denne måde sørger CPU'en for, at co-processoren altid er fuldt beskæftiget. Når alle planlagte applikationer er udført, går CPU'en til tilstanden *cpu_finished* og foretager sig ikke mere. CPU'en udgør ikke en vital faktor for den dynamiske opførelse af det rekonfigurerbare system, derfor er CPU'en implementeret forholdsvis abstrakt.

4.2.3 Koordinatoren

Koordinatoren er ligesom CPU'en opbygget omkring en simpel tilstandsmaskine bestående af tre tilstande. Hver gang en C-node modtager en applikationsforespørgsel fra CPU'en, sender C-noden en besked til M-noderne, at de skal skabe et billede af den indeværende ressourcesituation, således at C-noden kan tage stilling til, om den nye applikation kan allokeres eller ej. Når alle M-noderne har afgivet en rapport til C-noderne omkring den nuværende ressourcestatus, tager koordinatorerne et valg omkring hvilken master, der skal håndtere den nye applikation, hvis der i det hele taget er plads. Da koordinatorerne alle sammen benytter den samme protokol til beslutningen omkring, hvilken M-noder den nye applikation skal tildeles, bliver der kun valgt en M-node, nemlig den med flest ressourcer tilgængelige.

4.2.4 Masteren

M-noderne er et af de centrale elementer i COSMOS modellen. Masteren har mange vigtige funktioner blandt andet kan nævnes, at M-noderne står for allokeringen af nye tasks til slaverne, dette betyder, at M-noderne også skal håndtere reallokeringen af tasks, hvis f.eks. lavere prioriterede tasks optager ressourcerne. *Runtime*-styringen af det dynamiske rekonfigurerbare system er afgjort en af de vigtigste faktorer at kontrollere korrekt, hvis dynamiske rekonfigurerbare systemet skal performe optimalt. Den nuværende strategi er forholdsvis simpel og går i alt sin enkelthed ud på, at allokere de applikationer med højest prioritet så tæt på den Master-node, der kontrollerer dem. Dette betyder, at alle lavere prioriterede applikationer kan få deres tasks reallokeret hvis disse optager ressourcer tæt ved Masteren. Denne strategi er sandsynligvis langt fra optimal, men er simpel og ligetil og giver mulighed for at afprøve modellen. Der vil i afsnit 5.5.3 gennemgås en række *runtime*-styringsstrategier, med henblik på at forstå de faktorer, som har indflydelse på ydelsen for systemet, og forhåbentligt forbedre denne. Allokeringen af nye tasks fra en applikation sker ved at Masteren først indsamler information om ressourcedistributionen i co-processoren. Ud fra co-processorens nuværende ressourcedistribution beslutter Masteren, hvortil de nye tasks skal

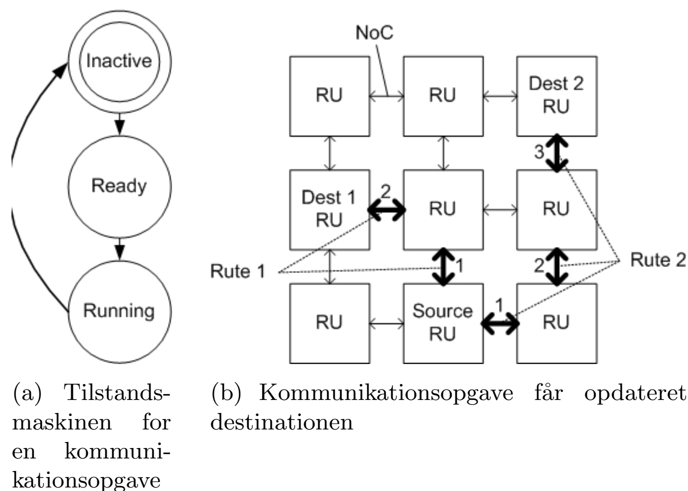
allokeres og hvilke "gamle" tasks, der skal reallokeres. Desuden består masteren af en *synchronizer* som kontrollerer, at alle tasks bliver udført i den rækkefølge, som er specificeret i deres task-graf. Hver gang en task afsluttes får *synchronizer*'eren besked herom og undersøger, om dette betyder at nye tasks kan udføres, hvad enten det er kommunikationsopgaver eller normale tasks. Hvis nye tasks er klar til at udføres, sender *synchronizer*'eren besked herom, således at de pågældende tasks kan starte.

4.2.5 Slaven

S-noden er enheden, hvorpå de enkelte tasks udføres. Det er således denne enhed, som kan rekonfigureres afhængigt af de behov, task'en, der skal udføres, har. Når en task er allokeret til en slave, er en af slavens vigtigste funktioner, at sørge for at disse tasks bliver udført i en bestemt rækkefølge, dette håndteres af funktionen implementeret i *scheduler*'eren. Rækkefølgen bestemmes af en prioritet, som er tildelt de enkelte tasks. Dette kan f.eks. være en overordnet deadline (*earlist-deadline-first* (EDF) *scheduling*) for den samlede applikation. Prioriteten kan også være sat ud fra mere individuelle behov, f.eks. kan de tasks, der indgår i den kritiske vej i applikationens task-graf have en højere prioritet. *Scheduleren* giver mulighed for, at de tasks, som bliver allokeret til S-noden, kan deles om denne i tiden, som vist på figur 4.2(a). Slaven kan bestå af et vilkårligt antal kontekster, dette bestemmes af arkitekturen for den co-processor, der simuleres. Antallet af kontekster bestemmer hvor mange tasks, der kan allokeres til en given slave og deles om dennes ene ressource i tiden. Co-processorens samlede antal af ressourcer, hvorpå der kan allokeres tasks, er således givet ved summen af slavernes samlede antal af kontekster.

4.2.6 Kommunikationsopgaven

Kommunikationsopgavens job er at simulere den interne kommunikation mellem de enkelte tasks i en applikation. Kommunikationsopgaverne er opbygget omkring en meget simpel tilstandsmaskine, der ses på figur 4.8(a). Denne tilstandsmaskine består, af tre tilstande *inactive*, *ready* og *running*. I tilstanden *ready* er kommunikationsopgaven klar til at begyn-



(a) Tilstandsmaskinen for en kommunikationsopgave

(b) Kommunikationsopgave får opdateret destinationen

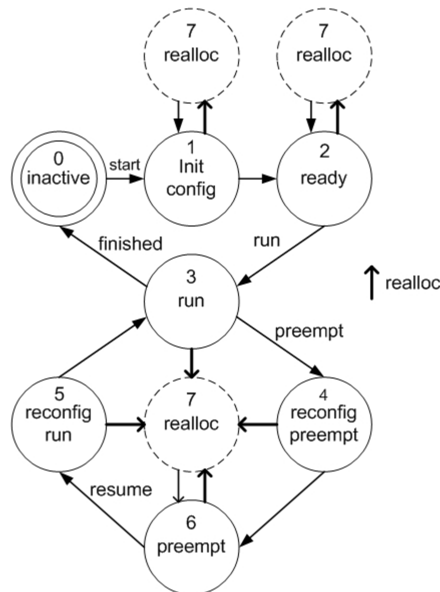
Figur 4.8: Kommunikationsopgaven

de, og venter på at *synchronizeren* i M-noden giver tilladelse til, at den kan begynde. Da task'ene for en applikation kan blive reallokeret mellem S-noderne i co-processoren, skal kommunikationsopgaven være i stand til at håndtere, at enten dens *source* eller *destination* ændres undervejs. Dette kan ses på figur 4.8(b), hvor *destinationen* for kommunikationsopgaven ændres. I modellen håndteres dette problem ved, at Masteren sender beskeder til kommnunikationsopgaverne omkring opdateringerne. Distancen målt i antal hop mellem *source*-task'en og *destinations*-task'en har indflydelse på, hvor lang tid kommunikationsopgaven befinder sig i tilstanden *running*. Antallet af hop, der er mellem *source*-task'en og *destinations*-task'en, bliver multipliceret med det tal, som angiver kommunikationsopgavens køretid for et enkelt hop. Dette simulerer at kommunikationsopgaven overfører en datamængde mellem *source*-task'en og *destinations*-task'en. Hvis *source*-task'en og *destinations*-task'en befinder sig på den samme S-node, bliver overførslen af data foretaget på en klokperiode. At overførslen af data sker på en klokperiode kan forvares ud fra de forsøg, der er foretaget på FPGA'en jf. afsnit 3.3.2, hvor det blev vist, at tilstanden af hukommelseselementerne ikke ændres ved rekonfigurering. Derfor kan data mellem to tasks allokeret på den samme slave overføres på en klokperiode, idet data ikke reelt skal flyttes fra slavens

register, men er klar, så snart den nye task har konfigureret slaven.

4.2.7 Modelleringen af en task

En task i en applikation bliver modelleret vha. en tilstandsmaskine, som er vist på figur 4.9. I tilstanden *inactive* venter tilstandsmaskinen på en *setup*- og *config*-besked fra S-noden, som indikerer, at task'en er blevet allokeret. For at simulere, at RU'en skal hente task'ens kontekst med blandt andet task'ens konfigurering, og at dette kan tage tid alt efter RU'ens *granularitet*, konfigurationens størrelse og den hastighed hvorved data kan overføres fra konfigurationshukommelsen til RU'en se figur 4.5, er der indført en tilstand *init config*. Denne tilstand består blot af en tæl-



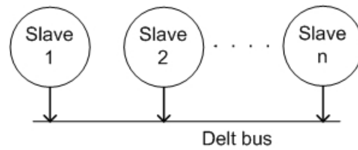
Figur 4.9: Tilstandsmaskinen for en task

ler, som initialiseres med en global værdi, som let kan ændres, således at der kan eksperimenteres med forskellige konfigurationstider. I tilstanden *ready* er task'en klar til udførelse og venter på starttilladelse fra S-nodens *scheduler*. Tilstanden *run* er meget simpel og består blot af en tæller, som holder styr på, hvor lang tid task'en har kørt. Kørselstiden er individu-

el for hver task i applikationen og bliver specificeret, når applikationen instantieres. Da det som nævnt, skal være muligt for task'ene at deles om RU'en i tiden, har task-tilstandsmaskinen 3 tilstande *reconfig_prempt*, *reconfig_run* og *preempt*. Tilstandende *reconfig_prempt* og *reconfig_run* simulerer henholdsvis den tid, det tager for en task at frigøre RU'en for at gøre plads til en anden task, og den tid det tager for en *preemptet* task at fortsætte, når RU'en atter er ledig. At udføre en *preemption* af en task betyder, at RU'en udfører et hardware-kontekstskifte, som går ud på at tage en backup af konfigurationshukommelsen for RU'en og alle hukommelseselementerne, som task'en havde i brug. Dette benyttes, når task'en skal starte op igen, så den begynder, hvor den slap, før den blev *prempt*'et. Den tid, det tager at foretage et hardware-kontekstskifte, afhænger som nævnt i afsnit 4.1 af en række forskellige faktorer og kan være meget store, hvis konteksten f.eks. skal gemmes i eksternhukommelse. RU'en kan også supportere flere kontekster på samme tid, hvilket gør at tiden, det tager at skifte mellem hver kontekst, kan være så lille som en klokperiode. I modellen er *preempt*-tiden implementeret vha. en global variabel, som let kan ændres, så der kan afprøves forskellige tider og deres indvirkning på det dynamiske rekonfigurerbare system. Skal en task reallokeres, gøres det ved at overføre hele task'ens kontekst fra en slave til en anden. Som det kan ses på tilstandsdiagrammet på figur 4.9, kan task'en gå til tilstanden *realloc* fra alle tilstande på nær *inactive*. I tilstanden *realloc* modelleres, at konteksten bliver flyttet fra en S-node til en anden S-node ved at benytte en tæller, der specificerer denne tid. Tiden inkluderer blandt andet, at der skal overføres en mængde data mellem slaverne svarende til størrelsen på konteksten. For at eksperimenter med denne tid skal være lettere specificeres den også vha. en global variabel.

4.2.8 NoC

Dette modul kontrollerer de kommunikationsopgaver, der er mellem hver task i en applikation. Modulet indeholder sin egen *scheduler*, som står for al kommunikationen med kommunikationsopgaverne. NoC modulet simulerer et meget simplificeret NoC, hvor alle slaverne er koblet sammen vha. en fælles bus, som vist på figur 4.10. Dette betyder, at der kun kan udføres en kommunikationsopgave ad gangen, og *scheduler*'ens opgave er at sørger for, at kommunikationsopgaverne udføres i den korrekte

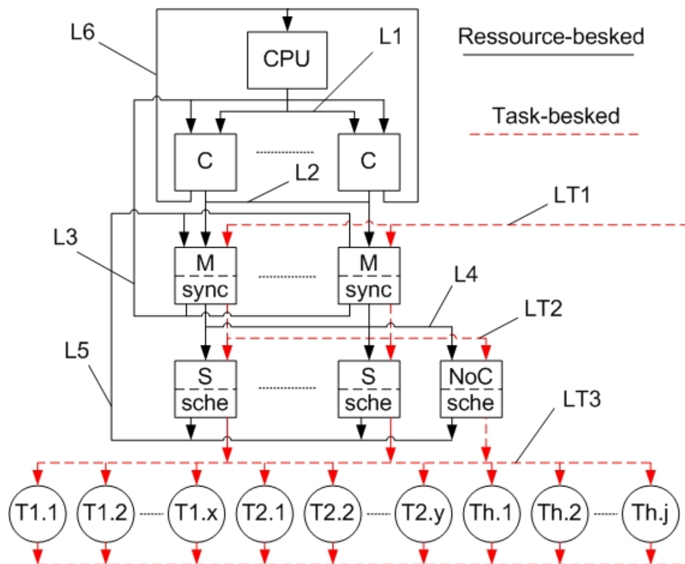


Figur 4.10: Modelleringen af NoC for COSMOS

rækkefølge.

4.2.9 Kommunikationen mellem modulerne i COSMOS

COSMOS er som nævnt en TLM-model, hvilket betyder, at de forskellige moduler kommunikerer gennem abstrakte datakanaler, og hver gang en pakke sendes gennem et link vækkes en funktion hos modtageren. På



Figur 4.11: Kommunikationen mellen de forskellige moduler i modellen

figur 4.11 ses strukturen af COSMOS, og hvordan de forskellige enheder i modellen kommunikerer gennem linkene. De beskeder, der håndterer ressourcestyringen, er vist med solide linier i figuren, og går mellem CPU'en,

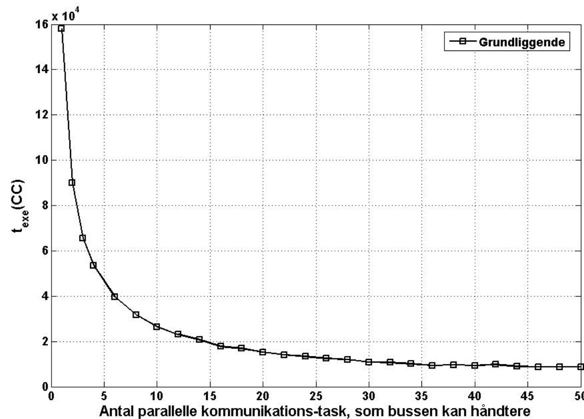
C-noderne, M-noderne og slaverne. De stiplede linier indikerer de beskeder, der kontrollerer task'ene både applikationens normale tasks og kommunikationsopgaverne. De pakker der sendes gennem linkene benytter et abstrakt format, der indeholder specifikke felter, der dækker de forskellige behov. For at se den samlede implementering af COSMOS i *SystemC* henvises der til bilag G.

4.3 Forbedringer

COSMOS modellen, der blev udviklet af Kehuai Wu i forbindelse med hans Ph.d. projekt [13], var et godt grundlag for viderudviklingen af en model i *SystemC*, som modellerer et dynamisk rekonfigurerbart system. Modellen havde dog svagheder, og skulle optimeres, for at være brugbar i forbindelse med store simuleringer med mange applikationer. De ændringer og forbedringer, der er foretaget, vil blive beskrevet i det følgende.

4.3.1 NoC

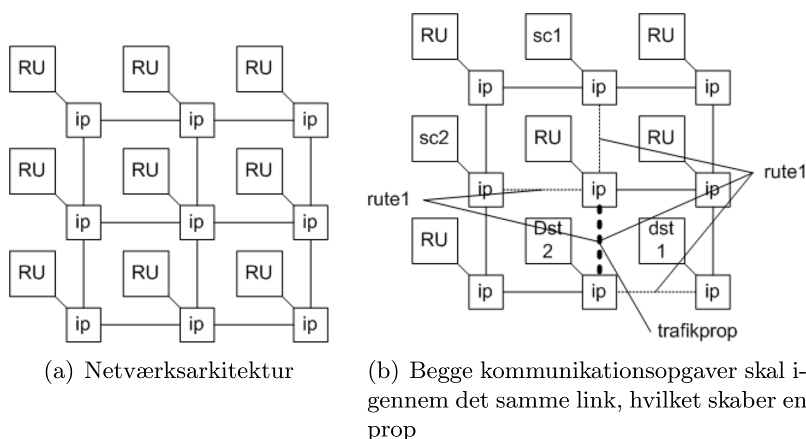
En af de første ting, der åbenlyst var behov for at ændre, var den måde, hvorpå COSMOS simulerer at RU'erne er forbundet vha. et *Network on chip*. I den oprindelige udgave af COSMOS er dette *NoC* implementeret, som en forholdsvis simpel bus, som alle slaverne er koblet op på jf. figur 4.10. Det er derfor kun muligt at eksekvere en kommunikationsopgave ad gangen. For store systemer bestående af mange RU'er i et netværk, er det tydeligt, at denne implementering er flaskehalsen for systemets samlede ydelse. På figur 4.12 vises kørselstiden for simuleringer udført på et system bestående af et 8 x 8 array af RU'er med to koordinatorer, to mastere og 60 slaver hver med 4 kontekster. Det ses tydeligt på figuren, at den samlede kørselstid bliver meget stor, hvis der kun kan afvikles en kommunikationsopgave ad gangen på bussen. Kan der derimod afvikles flere kommunikationsopgaver parallelt, reducerer det kørselstiden betragteligt, og det ses, at et netværk der kan håndtere 35 parallel kommunikationstasks, opnår et minimum for kørselstiden, og denne er konstant herefter. Dette indikerer, at det ikke længere er netværket, der udgør flaskehalsen for systemets samlede ydelse. Det er derfor nødvendigt at implementere



Figur 4.12: Kørselstiden for en simulering udført på et system bestånde af 8x8 RU'er, med 100 applikationer og et varierende antal kommunikationsopgaver der, kan afvikles parallelt

en form for NoC, som kan håndtere flere kommunikationsopgaver parallelt. En mulighed er at simulere, at RU'erne er koblet sammen i et netværk, som vist på figur 4.13(a). Dette kræver, at der implementeres et netværks-interface, som håndterer kommunikationen med de andre interfaces, og samtidigt kommunikerer med den rekonfigurerbare enhed, den er koblet til. Implementeringen af et sådan system er forholdsvist komplekst, der er mange faktorer, der skal tages højde for, blandt andet hvordan trafikken *routes*, så f.eks. trafikpropper, som vist på figur 4.13(b) undgås. En anden mulighed er at udvide den simple bus, som allerede eksisterer i COSMOS, så den er i stand til at eksekvere flere kommunikationsopgaver parallelt. På den måde vil modellen simulere et netværk, som på mange punkter ligner det på figur 4.13(a), men stadigvæk simpelt at implementere. Det eneste, som en simpel bus-implementering ikke er i stand til at simulere, er, at to kommunikationsopgaver skal gennem det samme link, som vist på figur 4.13(b), og derfor er den ene nødt til at vente, eller alternativt tage en anden rute. I et dynamisk rekonfigurerbart system, hvor Masteren har til opgave at allokere task'ene inden for den samme applikation så tæt på hinanden som muligt, og optimalt på samme slave, vil trafikken i netværket være forholdsvis lokal, og derfor vil der ikke opstå mange situationer, hvor to kommunikationsopgaver skal

benytte det samme link. Det er derfor fuldt tilstrækkeligt at implementere et simpelt netværk på et højere abstraktionsniveau, når systemet, der skal modelleres indeholder pakketrafik, som er forholdsvis lokalt i netværket.



(a) Netværksarkitektur

(b) Begge kommunikationsopgaver skal igennem det samme link, hvilket skaber en prop

Figur 4.13: NoC

4.3.2 NoC implementering

Implementeringen af den modificerede bus er forholdsvis simpel. Implementeringen er bygget op omkring en *scheduler*, som håndterer to vektorer en til de kommunikationsopgaver som venter, og en vektor bestående af de kommunikationsopgaver som er ved at afvikles. *Scheduler*'eren kontrollerer, at der ikke afvikles flere kommunikations-tasks ad gangen end det maksimalt tilladte antal. Antallet af kommunikationsopgaver, der kan afvikles samtidigt, bliver bestemt inden simuleringen starter vha. af en global variabel. Når flere kommunikationsopgaver venter på at afvikles, benytter *scheduler*'eren en bestemt protokol, der minder om EDF-*schedulering* til at afgøre hvilken kommunikationsopgave, der udføres når bussen igen er ledig. Prioriteringen af kommunikationsopgaverne bliver bestemt af den prioritet, som applikationen de stammer fra har. Det vil sige for applikationer med en højere prioritet (i tilfælde af EDF en lavere

sluttid) arver kommunikationsopgaverne ligeledes en højere prioritet.

4.3.3 Fleksibiliteten

En ulempe ved den oprindelige COSMOS model var, dens manglende fleksibilitet, når der skulle foretages ændringer på f.eks. arkitekturen af co-processoren, oprettes nye applikationer eller bare ændres på antallet af applikationer, der blev benyttet under en simulering. Alle ændringer i modellen skulle foretages manuelt vha. besværlige og tidskrævende og tit fejlflydte ændringer af toplevel-filen, hvor instantiserignen af applikationerne samt beskrivelsen af arkitekturen fandt sted. Hver gang der var foretaget en ændring i modellen, skulle den kompileres, før de nye ændringer trådte i kraft. Hvis modellen skulle bruges til større simuleringer og forsøg omkring forskellige arkitekturer, var det vitalt at modellen blev optimeret/ændret. For at gøre COSMOS modellen langt mere fleksibel og lettere at arbejde med, kan der benyttes en række tekstfiler, som specificerer forskellige inputs til modellen. Tanken er, at tekstfilerne skal beskrive arkitekturen for det dynamiske rekonfigurerbare system. Derudover skal tekstfilerne specificere antallet af applikationer, som skal køres under en simulering samt specificere disse i detaljer. Det eneste, der skal udføres før en simulering, er at rette i tekstfilerne, så de stemmer overens med den simulering, man vil foretage. Det er ikke nødvendigt at kompilere hele modellen, når der er foretaget ændringer i tekstfilerne, disse sker automatisk når simuleringen startes op på ny, med de ændrede tekstfiler som input. Ideen er, at benytte flere forskellige tekstfiler, som specificerer forskellige dele af modellen, og som tilsammen giver modellen de input, den har behov for. Der benyttes en tekstfil til at beskrive selve arkitekturen for det dynamiske rekonfigurerbare system, filformatet skal være simpelt og let at rette i, så der hurtigt kan foretages ændringer. Ligeledes skal der bruges en fil, som specificer selve simuleringen, dvs. hvor mange applikationer der skal udføres, og i hvilken rækkefølge de skal simuleres. Til sidst skal der være en tekstfil for hver applikation, som blandt andet specificerer task-grafen, prioriteten, kørselstiderne for de individuelle tasks og angiver antallet af tasks og kommunikationsopgaver i den pågældende applikation. Ved at oprette en fil for hver applikation, der beskriver denne, lettes opgaven med at skifte mellem de forskellige applikationer, som skal udføres under en simulering. Man opretter en form

for "bibliotek"med tekstfiler, som hver især beskriver en applikation, og når en simulering skal udføres vælges, ud fra dette "bibliotek"hvilke applikationer, der skal afvikles under simuleringen. For at oprette et "bibliotek"af applikations-filer kan der med fordel benyttes et program *Task Graph For Free* (TGFF) [1] som er et frit tilgængeligt program der kan genere vilkårlige task-grafer ud fra en række brugerspecificerede input. Dette letter arbejdet med at genere applikationer af varierende karakterer betydeligt og giver tilmed et mere realistisk billede af applikationer, da der udover task-graferne også generes kørselstider for både task'ene og kommunikationsopgaverne. Derfor er det fordelagtigt, at filformatet der specificerer hver enkelt applikation ligner det, der benyttes i TGGF, således at de applikationer der produceres i TGGF let kan inkluderes i modellen.

4.3.4 Fleksibilitetens implementering

Den fil, som specificerer arkitekturen for det rekonfigurerbare system, skal som nævnt have et simpelt filformat, og samtidig skal det give mulighed for at ændre på arkitekturen for det dynamiske system. På liste 4.1 er vist, hvorledes formatet for arkitekturfilen ser ud. Det ses, at dette består af en række tal, som angiver hvor mange rækker og kolonner af rekonfigurerbare enheder, der er i co-processorens arkitektur.

Listing 4.1: Formattet for filen der specificerer arkitekturen

```
RowSize
8

ColumnSize
8

noOfCoord
2

noOfMaster
2

noOfSlave
60

Architecture
```

```
c(0;0),m(0;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0)
s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0)
s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0)
s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0)
s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0)
s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0)
s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0)
s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),s(4;0),m(1;0),c(1;0)
```

De næste tal specificerer hvor mange af disse RU's, der henholdsvis er koordinatorer, mastere og slaver. Til sidst er der et array af bogstaver, der angiver, hvor i netværket slaverne, masterne og koordinatorerne er placeret. Det første tal i parenteserne, angiver for slavernes vedkomne, hvor mange kontekster hver slave har. For koordinatorerne og masterne er dette tal en indikator for, hvilke koordinatorer og mastere der er associeret med hinanden. Som vist på figur 4.6, er hver master styret af en bestemt koordinator, og har en master det samme tal som en koordinator, betyder det, at masteren er kontrolleret af den pågældende koordinator. Det sidste tal i parentesen specificerer hvilken chip, den pågældende rekonfigurerbare enhed befinder sig på. Dette muliggør, at der kan simuleres et dynamisk rekonfigurerbart system, hvor co-processoren faktisk består af flere chip's bundet sammen i et netværk. De RU's, der har samme tal i den sidste parentes, befinder sig på den samme chip.

Filformatet, som specificerer selve simuleringen, er meget simpelt og kan ses på liste 4.2. De faktorer, der er vigtige at specificere, er antallet af applikationer, som skal udføres og den rækkefølge, som de skal udføres i. Det er også muligt at angive et antal af frie ressourcer, der skal være ledige før co-processoren accepterer en ny applikation fra CPU'en.

Listing 4.2: Formatet for filen der specificerer simulationen

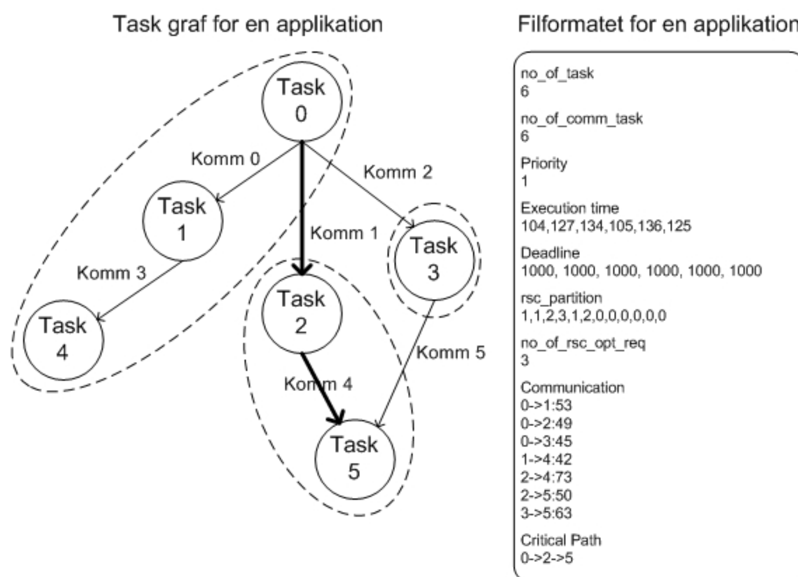
```
no_of_reserved_rsc
20

no_of_applications
20

order_of_execution
0,3,3,2,4,0,3,2,1,1,4,1,1,3,0,3,3,0,1,4
```

Når applikationerne skal indlæses fra simuleringsfilen, sker det ved at linien, som angiver rækkefølgen læses ind, og hver kommaseparerede heltal bliver betragtet som en applikation. Hvis heltallet ikke er identisk med et af de tal, der tidligere er læst ind, søges efter en fil af navnet "app(heltallet).txt", som beskriver den nye applikation, der skal køres under simuleringen.

Filformatet, som specificerer hver applikation, kan ses på figur 4.14. Figuren viser ligeledes den applikation, som bliver beskrevet i filen. De første to felter angiver antallet af henholdsvis tasks og kommunikationsopgaver i applikationen. *Priority* specificerer applikationens allokeringsprioritet,



Figur 4.14: En applikation og dens tilhørende specificering i filformat

hvilket M-noden benytter, når en applikation skal allokeres. M-noden kan reallokere en lavere prioriteret applikationer for at gøre plads til applikationer med højere prioritet. For *priority* gælder, at større tal medfører højere prioritet for applikationen. Felterne *execution time* og *deadline* specificerer henholdsvis kørselstiden og en *deadline* for hver task i applikationen. *Deadlinen* giver mulighed for, at task'ene indbyrdes i applikation kan have forskellig prioritet. Tallet, specificeret i *deadline*, bliver

under simuleringen lagt til den aktuelle simuleringstid, når applikationen allokeres, dette giver mulighed for, at slaverne kan benytte EDF *schedulering*, hvis to tasks fra den samme applikation er allokeret til den samme slave. *Rsc-partition* beskriver hvilke tasks i applikationen, der med fordel kan allokeres på samme slave og hvilke tasks, der bør allokeres på forskellige slaver, således at de kan afvikles parrallelt. På figur 4.14 ses, at applikationen optimalt skal have 3 ressourcer, og at fordelingen af task'ene er som følgende: {0,1,4}, {2,5}, {3} på hver sin slave. Det er ikke strengt nødvendigt, at M-noden skal allokere task'ene på denne måde, det er blot et guideline. Det afgørende punkt for beskrivelsen af task-grafen er, hvordan kommunikationen foregår. I filen, der angiver hver applikatioen, bliver dette specificeret på en simpel måde, hvor hver kommunikationsopgave er beskrevet ved en enkelt linie af formen:

$$destination \rightarrow source : kommunikationsopgavens\ kørselstid$$

Dette format er simpelt for en bruger at benytte, og det er ikke svært at beskrive en vilkårlig task-graf på denne måde. Derudover er formatet let at håndtere for modellen, når den skal oprette hver enkelt applikation, som skal køres under simuleringen. Hver gang en ny kommunikationsopgave indlæses fra filen, opretter modellen et kommunikationsobjekt, som består af tre felter, *source_task*, *destination_task* og *comm_time*. Når alle applikationer er indlæst, bliver kommunikationsopgaverne initialiseret ud fra kommunikationsobjekterne. Det er muligt at specificere den kritiske vej gennem task-grafen i applikationsfilen, det gøres som vist på figur 4.14, hvor den kritiske vej går gennem task0, task2 og task5.

En stor udfordring på vejen mod en mere fleksibel model var klart, at *SystemC* forlanger at alle de moduler/entiteter, der benyttes i modellen, skal være instantieret på forhånd, dvs. allerede når modellen bygges i kompileren. Dette harmonerer ikke med implementeringen af en fleksibel model, hvor det netop ikke på forhånd bliver defineret, hvor mange moduler/entiteter af henholdsvis slaver, mastere, tasks, osv. der skal bruges. Den eneste måde dette, problem umiddelbart kan løses på, er at bruge pointerne, som understøttes i C++ [8] og dermed også i *SystemC*. Ved at oprette et array af pointerne, som alle peger på et objekt af den type, der skal anvendes, kan der på forhånd reserveres plads i hukommelsen til de objekter, man forventer at benytte. Dette gør, at kompileren accepterer, at der kan instantieres et vilkårligt antal objekter, fordi der altid er reserveret plads til et givet antal. F.eks. kan nævnes, at der altid reserveres

plads til, at 64 slaver kan instantieres, selvom der under simuleringen ofte benyttes færre end 64. Det kan føre til meget spildplads i hukommelsen, hvilket igen kan betyde, at der under simuleringen "løbes tør" for hukommelse, eller at simuleringen bliver langsomme. Derfor skal det gøres med omtanke, når der reserveres plads i hukommelsen på forhånd.

4.3.5 Håndteringen af store simuleringer

De simuleringer, der hidtil var udført på COSMOS, var forholdsvis simple af flere grunde. Blandt andet skulle de kun bruges til at verificere modellen under udviklingen af den, og derudover var det som nævnt besværligt at genere store simuleringer, da alt skulle skrives i hånden i toplevel-filen. Dette betød, at det først var muligt at foretage store simuleringer bestående af mange applikationer efter at COSMOS modellen, som omtalt i foregående afsnit, var opdateret. Dette gav i midlertidigt problemer, fordi COSMOS modellen ikke var i stand til at håndtere disse store simuleringer. Problemet var måden, hvorpå modellen håndterede de tabeller, som beskrev de enkelte applikationer se figur 4.7. Tabellerne for hver applikation blev gemt i et tredimensionalt array, som blev *parset* til masteren, som benyttede denne information i *synchonizer*'eren til at afgøre, hvilke tasks der kunne udføres. Da hver applikation kunne bestå af mange tasks og kommunikationsopgaver, voksede dette tredimensionelle array hurtigt, og dette resulterede i fejl under afviklingen af modellen, når antallet af applikationer blev for stort. Fejlen opstod, fordi *Cygwin* ikke er i stand til at håndtere så store arrays. Problemet kan skyldes indstillinger i kompileren eller størrelsen af den hukommelse, som *Cygwin* får allokret af Windows. Løsningen er at ændre i den datastruktur, som den oprindelige COSMOS model benyttede. Da mange af de applikationer, der udføres under en simulering, er identiske, kan datastrukturen ændres, således at det ikke er nødvendigt at *parse* alle applikationernes tabeller til masteren. I stedet kan man begrænse størrelsen af array'et, så det kun indeholder en tabel for hver af de forskellige applikationer, der optræder i simuleringen. På denne måde mindskes array'et betydeligt, fordi den samme applikationstabel ikke optræder flere gange. Hver gang masteren får tildelt en ny applikation, kopierer *synchonizer*'eren applikationstabellen fra det tredimensionelle array over i en vektor. Fremover, når der skal ændres i tabellen for applikationen, gøres det kun i vektoren og ikke i

det oprindelige array, som det var tilfældet i den oprindelige version af COSMOS.

Efter at COSMOS modellen var rettet til og kunne håndtere store simuleringer med mange applikationer, blev der trigget en række fejl, som ikke tidligere havde optrådt. Dette skyldtes, at modellen ikke var testet grundigt, og kun havde kørt små simuleringer. De nye store simuleringer testede mange ny aspekter af modellen, specielt blev grænserne testet i den forstand, at modellen f.eks. ikke før havde haft alle ressourcerne i brug på samme tid. Fejlene, der opstod, var simple logiske fejl ofte i *if*-sætninger, hvor grænserne ikke var afprøvet før. Arbejdet med at lokalisere fejlene var tidskrævende og krævede en logisk og systematisk fremgangsmåde, hvor det vha. af testbeskeder blev sporet i hvilket modul, fejlen optrådte for derefter at lokalisere hvilken funktion, der forårsagede fejlen. Der er rettet mange fejl af denne type i modellen, det kan dog ikke med sikkerhed siges at modellen er fejlfri. Dette vil kræve, at der foretages en tidskrævende funktionstest, hvor hele modellen minutiøst bliver gennemgået af specifikke test-cases. Modellen har dog via de store simuleringer, der er udført været igennem en form for strestest, hvilket giver en indikation af, at modellen er tilnærmelsesvis fejlfri.

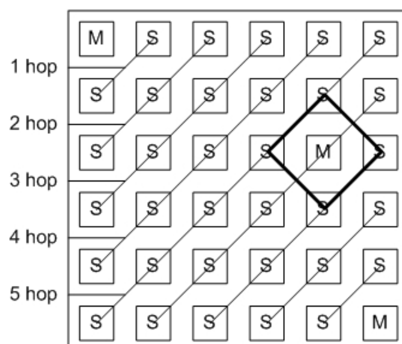
Kapitel 5

Runtime-styringen

For at udnytte et dynamisk rekonfigurerbart system optimalt, er det vigtigt, at der fokuseres på *runtime*-styringen, da dette har stor indflydelse på, hvordan systemet performer. I dette kapitel vil forskellige strategier for *runtime*-styringen blive undersøgt. Der indledes med en gennemgang af den strategi COSMOS oprindeligt benyttede til *runtime*-styringen.

5.1 Grundliggende allokerings-strategi

Som nævnt tidligere, er det masteren (se afsnit 4.2.4), der håndterer *runtime*-allokeringen af nye applikationer og reallokeringen af de tasks, der allerede er allokeret. Den nuværende allokeringsstrategi er meget simpel, og prøver altid at mappe en applikation på de slaver, som er tættest på M-noden målt i antal hop i netværket. Dette gør, som vist på figur 5.1, at slaverne, der bliver valgt til at håndtere en applikation, ligger diagonalt på netværket. En af ulemperne ved denne fremgangsmåde er, at selvom task'ene for en applikation bliver allokeret f.eks. to hop fra M-noden, vil der altid minimum være to hop imellem de valgte slaver, når task'ene



Figur 5.1: *Grundliggende runtime*-styrringsstrategi

skal kommunikere indbyrdes. I den nuværende *runtime*-strategi er det således at højere prioriterede applikationer, altid kan reallokere applikationer med lavere prioritet, som optager ressourcerne tæt ved M-noden. Dette medfører at en højt prioriteret applikation, som kræver ressourcer tæt ved M-noden, kan forårsage en reallokering af lavere prioriterede applikationer. Disse skal igen have plads tæt ved M-noden, og det kan derfor være nødvendigt at reallokere en applikation med endnu lavere prioritet. Denne simple *runtime*-strategi kan derfor skabe en dominoeffekt, som reallokerer mange tasks, hvilket kan have både fordele og ulemper. Det positive er, at tasks inden for den samme applikation højst sandsynligt bliver reallokeret til en ny location, hvor de stadig er nær hinanden eller på samme slave. Ulempen er, at reallokering er en besværlig og tidskrævende opgave, hvilket modellen også simulerer, derfor skal der være en fornuftigt grund til at reallokere en task, da det ellers kan påvirke systemets performance negativt. I det følgende vil denne simple allokeringsstrategi blive benævnt *grundliggende*, og denne benyttes til at sammenligne systemets performance med andre *runtime*-styringsstrategier.

Dynamiske rekonfigurerbare systemer udgør et emne, som først er kommet i fokus inden for de seneste par år. Der findes derfor ikke meget litteratur om dynamisk allokering af applikationer i et rekonfigurerbart system. For at hente inspiration til forskellige allokeringsstrategier, har det derfor været nødvendigt at kigge på, hvad der er lavet inden for statisk allokering, som har været i fokus i mange år. Problemerne i statisk allokering minder på en del områder om den problemstilling, der

optræder i dynamisk allokering blandt andet, at det er et *NP-complete*-problem, hvor det drejer sig om at allokere tasks til ressourcer, som de kan deles om i både tid og rum. Desværre er der også store forskelle på de to systemer, blandt andet er der ofte i forbindelse med statisk allokering meget regnekraft til rådighed, samtidig med at beslutningstiden ikke er alt afgørende for, hvordan det statiske system performer. I dynamisk allokering forholder det sig stik modsat, der er ofte begrænset regnekraft til rådighed, og da allokeringen af applikationerne foregår *runtime*, hvilket betyder at der er en nøje sammenhæng mellem den tid, der bruges på allokeringen, og det dynamiske rekonfigurerbare systems ydelse. I statisk allokering benyttes ofte en heuristik til finde en suboptimal løsning på allokeringsproblemet [2] [3] [6]. En heuristik arbejder generelt ved at generere forskellige permutationer af datasættet, som der arbejdes på. For hver permutation udregnes den faktor der optimeres mod. Heuristikken fortsætter, indtil et vist antal forskellige permutationer er afprøvet, eller en af permutationer opnår en ønsket optimering. I statisk allokering benyttes dette til at allokere task'ene til ressourcerne og estimere den samlede kørselstid. Størrelsen af dette estimatet afgør om denne permutation skal benyttes eller der skal genereres en ny allokering af task'ene til ressourcerne. Der findes forskellige strategier til at danne permutationer ud fra et sæt af tasks, der skal allokeres. Heuristisk arbejder normalt i polynomisk tid $O(n^k)$, hvor n er størrelse på problemet, og k er en konstant. Den løsning, der findes, er ofte en suboptimal løsning, som ligger inden for en defineret grænse af den optimale løsning. At løse allokeringsproblemet i polynomisk tid kan være problematisk for dynamiske rekonfigurerbare systemer, fordi den tid der benyttes til allokeringen, direkte influerer den samlede kørselstid.

I det følgende bliver tre simple *runtime*-allokeringsstrategier beskrevet. En af dem bygger på inspiration fra den statiske allokering, og de to andre har til formål at udbedre nogle af de problemer, den *grundliggende* allokeringsstrategi har. Fælles for de tre algoritmer er, at de er simple, således at de ikke har behov for meget regnekraft, og beslutningen angående allokering og reallokering kan foretages hurtigt. Koden for implementeringen af de følgende strategier kan findes i bilag G.

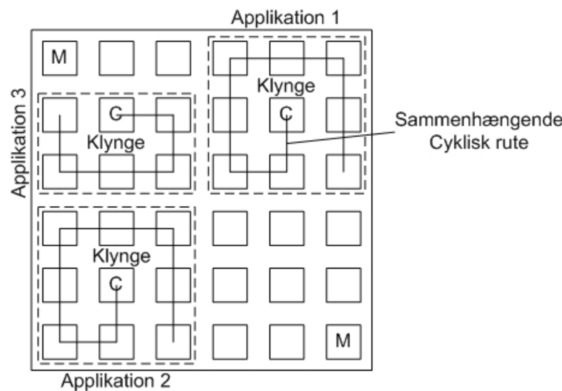
5.2 Dynamisk prioritet

I den *grundliggende runtime*-styringsstrategi, bliver lavere prioriterede applikationers tasks altid reallokeret for at frigive ressourcer til højere prioriterede tasks. Dette gøres, fordi det på den måde kan garanteres, at de højere prioriterede tasks bliver sammen i klynger og dermed mindskes kommunikations-*overhead*'et. Det at reallokere en task er en tidskrævende opgave, og skal kun foregå, hvis det gavner systemets samlede ydelse. I tilfældet med den *grundliggende* allokeringsstrategi opleves det, at lavere prioriterede tasks, som er tæt på at være færdige, bliver reallokeret, hvilket kan forlænge systemets samlede kørselstid. Hvis tasken derimod lige er startet op og stadig skal køre i lang tid, betyder en reallokering ikke så meget i forhold til den resterende kørselstid. I den *grundliggende* strategi kan der endvidere ske det, at lavere prioriterede tasks syltes, og aldrig når deres *deadline*, fordi de reallokeres hele tiden, selvom de er tæt på at være færdige, og dette er naturligvis ikke fordelagtigt for det samlede systems ydelse. I den *grundliggende runtime*-strategi tages der kun højde for rumlige forhold dvs. hvor task'ene placeres på ressourcerne, og hvordan task'ene placeres indbyrdes i forhold til hinanden. Skal der derudover indgå tidsmæssige forhold i beslutningen omkring allokeringen af nye tasks, kan dette inkluderes på mange måder. I prioritetsstrategien bliver der på baggrund af kørselstiden for en task, givet en højere allokeringsprioritet, hvis task'en er nær slutningen. Dette gør, at højere prioriterede tasks ikke kan reallokere lavere rangerede tasks, hvis disse er ved at være færdige. På denne måde sikres det, at lavt prioriterede tasks ikke bliver reallokeret mere end højst nødvendigt.

5.3 Spiral-allokering

Den *grundliggende* allokeringsstrategi benytter, som nævnt en meget simpel fremgangsmåde, hvor hovedvægten er lagt på at allokere task'ene for en given applikation så tæt på M-noden som muligt for at reducere kommunikations-*overhead*'et. Dette har dog flere uheldige bivirkninger blandt andet, som vist på figur 5.1, bliver task'ene allokeret diagonalt på netværksstrukturen, hvilket bevirker, at der er to hop mellem hver task

allokeret på samme diagonale linie. Udover dette dilemma skaber det også *hot spots* omkring M-noderne, hvor reallokering ofte forekommer. For at undgå disse problemer, er der implementeret en simpel allokeringsstrategi, som benytter et vilkårligt centrum i netværket som udgangspunkt for at danne en klynge for en given applikation. Dette bevirker, at applikationerne undgår at forstyrre hinanden i samme grad, som det er tilfældet i den *grundliggende* strategi. På figur 5.2 er vist strategien for spiral-allokeringsalgoritmen. Ideen er, at der ud fra hver slave søges efter en sammenhængende rute, søgningen stopper, når ruten indeholder nok ressourcer til at allokere den nye applikation. Algoritmen forsøger, at søge



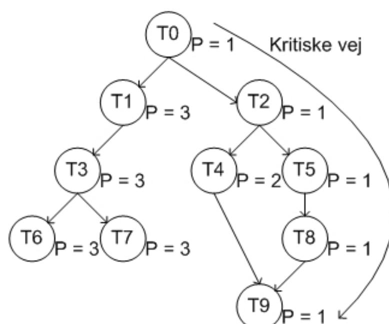
Figur 5.2: Spiral *runtime*-styrringsstrategi

i en spiral omkring den valgte slave. Dette har flere fordele, blandt andet sørger algoritmen for, at task'ene forbliver tæt sammen i en klynge, udover det har den sammenhængende rute den fordel, at der kun er et hop mellem hver efterfølgende slave på ruten. Dette er med til at mindske kommunikations *overhead*'et i forhold til den *grundliggende* strategi, hvor der var to hop mellem hver slave allokeret på den diagonale linie, som vist på figur 5.1. Implementeringen af spiral-algoritmen er foretaget i Masteren, og er bygget op omkring to funktioner *find_max_rsc_conc_cyclic* og *find_conc_cyclic_path_ccw*. Funktionen *find_max_rsc_conc_cyclic* kalder funktionen *find_conc_cyclic_path_ccw*, som søger efter en sammenhængende spiralrute ud fra en given slave og returnerer hvis muligt en rute, som har det krævede antal af sammenhængende ressourcer. Hvis der med udgangspunkt i alle slaverne ikke er fundet en sammenhængende rute, som har det antal ressourcer, der er krævet, benyttes den rute, der har det

maksimale antal sammenhængende ressourcer, og de tasks, der ikke kan allokeres langs ruten, bliver allokeret så tæt på som mulig. Denne *runtime*-strategi benytter ikke reallokering til at optimere performance, men bygger i stedet på optimering af den første allokering af task'ene, når disse skal mappes til det rekonfigurerbare system.

5.4 Kritisk vej

For at optimere allokeringsstrategierne yderligere er der hentet inspiration fra den statiske allokering. I [10] benyttes den kritiske vej igennem en task-graf til at optimere allokeringen, specielt tildeles de tasks, der ligger på den kritiske vej, en højere prioritet. Dette kan også benyttes i et dynamisk rekonfigurerbart system, således at tasks tilhørende den kritiske vej får en højere allokeringsprioritet. Derudover skal tasks på den kritiske vej ikke udsættes for reallokering, på den måde sikres det, at de tasks, der befinder sig på den kritiske vej ikke udsættes for yderligere "forsinkelser". Endvidere kan prioriteten baseret på den kritiske vej bruges ikke kun af M-noderne til allokering, men også af S-noderne i deres *schedulering*, således at tasks i den kritiske vej får første prioritet til at benytte slavens ressourcer. Den kritiske vej gennem en task-graf kan bestemmes på flere



Figur 5.3: Task-graf der illustrerer, hvordan prioriteterne tildeles de forskellige tasks

måder, det kan gøres dynamisk, mens systemet kører, eller der kan på forhånd være foretaget en statisk analyse af task-grafen, som finder den kritiske vej. Det er antaget i COSMOS modellen at den kritisk vej er

fundet på forhånd vha. statisk analyse. Den kritiske vej kan bestemmes forholdsvis simpelt, når både task'ene- og kommunikationsopgavernes-kørselstider er kendte. På figur 5.3 ses en task-graf med den kritiske vej gennem task'ene $T0 \rightarrow T2 \rightarrow T5 \rightarrow T8 \rightarrow T9$. Ideen er, at de tasks, der befinder sig på den kritiske vej, skal have den højeste prioritet (P=1), denne høje prioritet sørger for, at task'ene ikke kan reallokeres, og samtidigt får de førsteret til ressourcerne på slaven. Det ses endvidere på figur 5.3, at de tasks, som er i familie med task'ene på den kritiske vej, ligeledes får en høj prioritering se T4 som får prioriteten (P=2). Dette skyldes at task'en T9 er afhængig af task'en T4, som ikke befinder sig på den kritiske vej, men alligevel kan have indflydelse på udførelsen af task'en T9. De resterende tasks i task-grafen får en lavere prioritet (P=3), da de ingen indflydelse har på de tasks, der befinder sig på den kritiske vej. Implementeringen af den kritiske vej er udført vha. to hovedfunktioner *assign_priority* og *assign_priority_recursion*. Funktionen *assign_priority* tildeler først en høj prioritet til de tasks, der indgår i den kritiske vej. Derefter kalder den funktionen *assign_priority_recursion* på hver af disse tasks. Funktionen *assign_priority_recursion* er en rekursiv funktion, som søger efter forældre til den task, den er blevet kaldt på [5]. Findes der en forældre, som ikke har fået en prioritet tildeles denne en prioritet, og funktionen kalder sig selv med denne forældre som input. På denne måde søges der baglæns gennem task-grafen, og de tasks i grafen, som har forbindelse til den kritiske vej, får tildelt en mellemhøj prioritet. Til slut bliver der tildelt en lav prioritet til alle de tasks, som endnu ikke har fået tildelt en prioritet, dvs. alle de tasks, som hverken er på den kritiske vej, eller er forældre til tasks på denne vej.

5.5 Simuleringer

I dette afsnit, vil en række resultater fra simuleringer udført på COSMOS modellen blive gennemgået. Der vil først foretages et forsøg, hvor der ændres på den tid som den *grundliggende* strategi bruger på beslutningen om, hvor tasks skal allokeres, og hvilke der skal reallokeres. Dette har til formål at afdække betydningen af denne beregningstid. Dernæst vil en række forsøg på forskellige arkitekturer blive udført, for at vise modellens fleksibilitet og fastslå arkitekturens betydning for co-processorens ydel-

se. Endeligt vil en række simuleringer blive foretaget, som har til formål at sammenligne effektiviteten af de forskellige allokeringsstrategier. Der er genereret fem applikationer vha. programmet *Task Graph For Free* (TGFF) [1], applikationernes task-grafer er vist i bilag C, de tilhørende filer, som specificerer hver applikation, kan ses i bilag B. Ud fra disse fem forskellige applikationer, dannes der adskillige simulerings-inputfiler (se afsnit 4.3.3) bestående af 100 applikationer i tilfældig rækkefølge. Simuleringerne bliver derefter gennemført med de forskellige simuleringsfiler som input, og kørselstiden afbilledes i form af antallet af klokperioder ($t_{exe}(cc)$). Kørselstiderne, som ses på de kommende figurer, er et gennemsnit af kørselstiderne for disse simuleringer. For hver simulering der afvikles på COSMOS genereres der en log-fil, som kan ses i udsnit i bilag D. Kørselstider og resultater som modellen genererer udtages efter hver simulering fra denne log-fil. For alle simuleringerne gælder at NoC kan håndtere op til 64 beskeder parallelt, dette betyder at NoC ikke bliver flaskehalsen for systemets performance. Desuden er den tid, det tager at rekonfigurere en task afstemt, så denne har et realistisk forhold til den gennemsnitlige kørselstid for task'ene. Rekonfigureringstiden er sat til cirka halvdelen af en gennemsnitlig kørseltid for task'ene.

5.5.1 Runtime-styringen tager tid

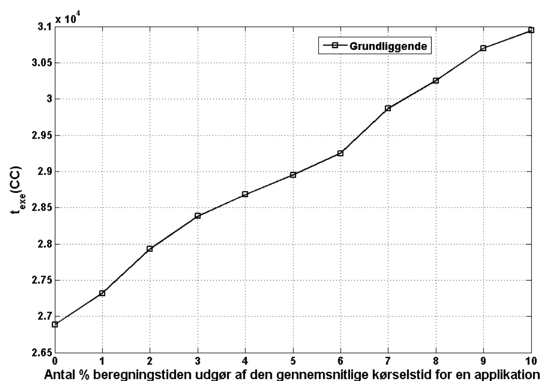
Co-processoren er et dynamisk system, hvilket vil sige, at det ikke på forhånd er kendt hvilke applikationer, der skal afvikles på co-processoren. Dette afgør CPU'en løbende, når denne skal afvikle en applikation, som kan drage nytte af dedikeret hardware, undersøger CPU'en, om der er plads på co-processoren, hvis det er tilfældet bliver applikationen sendt til co-processoren for videre udførelse. Herefter er det co-processorens job at allokere applikationen og eventuelt reallokere andre lavere prioriterede tasks, for at optimere forholdene for den indkomne task. Den tid (beregningstiden) der benyttes for at afgøre, hvor applikationens tasks skal mappes, og om co-processoren udfører tasks fra andre applikationer som skal reallokeres, betyder meget for co-processorens ydelse. I dette afsnit undersøges denne tid nærmere ved at udføre en række simuleringer, hvor beregningstiden for den *grundliggende runtime*-styringsstrategien varieres. De fem applikationer, som benyttes i simuleringerne kan ses i bilag B, hvor kørselstiderne for hver task er specificeret. Gennemsnit-

tet af alle kørseltiderne for task'ene er cirka 113 klokperioder, og hver applikation har i gennemsnit cirka 12 tasks, desuden består hver applikation i gennemsnit af 14 kommunikationsopgaver, som har en gennemsnitlig kørselstid på cirka 57 klokperioder. Antages det, at halvdelen af opgaverne udføres parallelt og halvdelen af kommunikationsopgaverne er intern kommunikation, dvs. de tager 1 klokperiode uanset deres normale kørselstid, og de resterende kommunikationsopgaver skal over 2 hop i netværket, bliver den gennemsnitlige kørselstid fundet analytisk for hver applikation $\underbrace{(12/2)\cdot 113}_{tasks} + \underbrace{(14/2)\cdot 57\cdot 2}_{kom.opg.} = 1476$ klokperioder. I tabel 5.1

Task	**Kørselstid**
0	1026
1	1610
2	1321
3	818
4	1555
Gennemsnit	1266

Tabel 5.1: Simulerede kørselstider for de enkelte applikationer

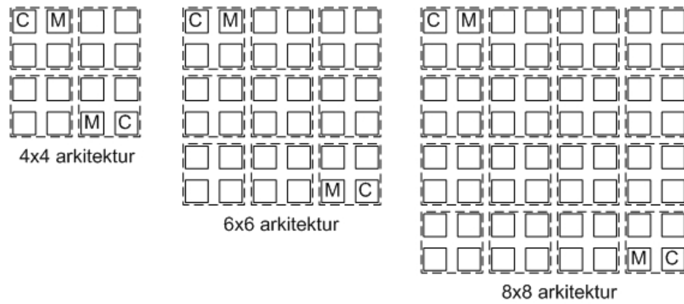
ses kørselstiderne for de 5 applikationer, hvor de er kørt enkeltvis på co-processoren, hvilket er ideelle forhold, som sjældent opnås når co-processoren udfører mange applikationer parallelt. I den følgende simulering vil beregningstiden for den *grundliggende runtime*-styringsstrategi varieres med 0% og op til 10% af den gennemsnitlige analytiske kørselstid for applikationerne. Resultatet af simuleringen kan ses på figur 5.4. Det ses, at der er en forholdsvis lineær sammenhæng mellem kørselstiden for 100 applikationer og beregningstiden for den *grundliggende* reallokeringsalgoritme. Det viser sig at beregningstiden som ventet har indflydelse på co-processorens performance. I de følgende simuleringer vil beregningstiden for de forskellige (re)allokerings-algoritmer ikke inkluderes, under antagelse af, at den er ens for de forskellige simple algoritmer, og derfor ikke influerer det indbyrdes forhold mellem algoritmerne.



Figur 5.4: Viser sammenhængen mellem beregningstiden og kørselstiden for 100 applikationer

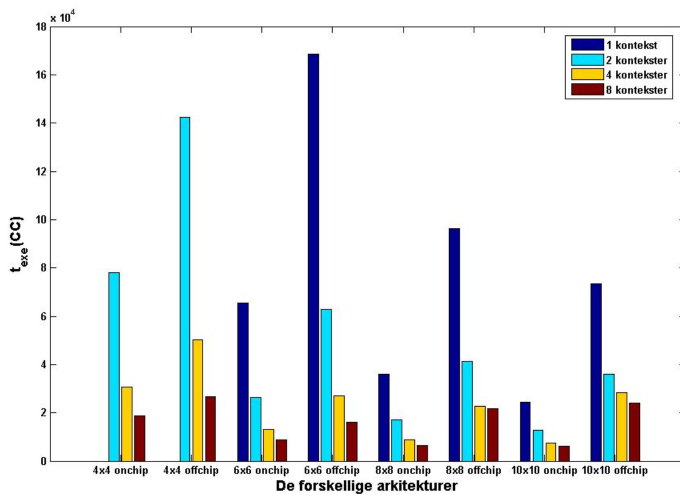
5.5.2 Arkitekturen

For at få et indblik i, hvordan arkitekturen påvirker co-processorens ydeevne, er der lavet en række simuleringer med forskellige arkitekturer. På figur 5.5 er vist, hvordan de forskellige arkitekturer er opbygget, alle har to M-noder og to C-noder i hvert hjørne, og de resterende rekonfigurerbare enheder udgør slaverne. Der benyttes endvidere en arkitektur af størrelsen 10x10. Co-processoren kan også have en arkitektur, der består af flere chips sat sammen i et netværk. Hver chip kan indeholde et bestemt antal rekonfigurerbare enheder, og hver gang off-chip-kommunikation er nødvendig mellem to tasks, bliver dette straffet ved at multiplicere den normale kommunikationstid med en faktor 4. I dette forsøg vil hver chip maksimalt bestå af fire RU's, som vist på figur 5.5. Dette betyder, at en arkitektur bestående af 4x4 rekonfigurerbare enheder skal benytte 4 chips, og 6x6 skal benytte 9 chips osv. Derudover vil der for hver arkitektur foretages en simulering, hvor slaverne henholdsvis har 1,2,4 og 8 kontekster. Resultatet af disse simuleringer kan ses på figur 5.6. Simuleringerne for et 4x4 array med een kontekst kan ikke gennemføres, fordi der kun er 12 ressourcer til rådighed, og den største af applikationer består af 17 tasks. Derfor er de første resultater for 4x4 array'et med 2 kontekster, hvilket giver 24 ressourcer i alt. Det ses, at co-processorens performance i høj grad er afhængig af den valgte arkitektur, der benyttes. Det ses, endvidere at større array's af RU'er giver en bedre performance, dette er logisk,



Figur 5.5: Arkitekturerne der benyttes til at simulere på

idet flere rekonfigurerbare enheder medfører flere ressourcer, og dermed er der mulighed for, at flere tasks kan udføres parallelt. Det gælder for



Figur 5.6: Kørselstiderne for de forskellige arkitekturer for simuleringer med 100 applikationer

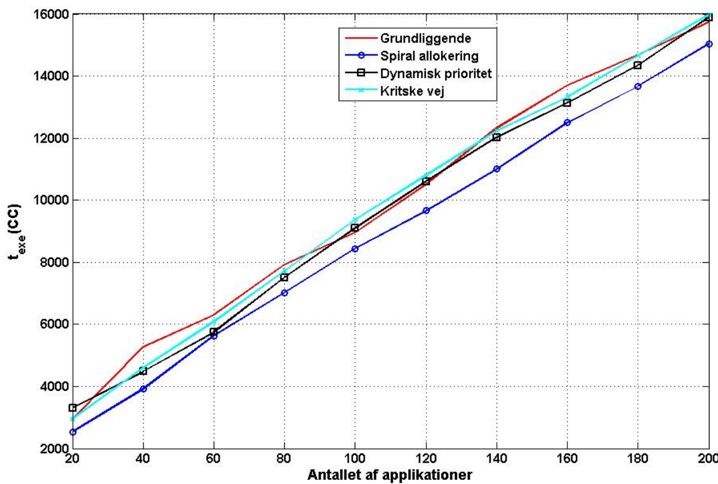
alle arkitekturerne, at antallet af kontekster betyder meget for den samlede ydelse af systemet. Dette skyldes flere faktorer, blandt andet har antallet af kontekster stor betydning for kommunikations-*overhead*'et. Har den samme slave flere kontekster, er der mulighed for intern kommunikation mellem tasks fra den samme applikation, hvis de er mappet til samme slave. Interkommunikationstiden er sat til en klokperiode u-

anset størrelsen på kommunikations-task'en jf. afsnit 4.2.6, derfor kan interkommunikation være med til at reducere den samlede kørselstid betragteligt. Hvis slaverne indeholder flere kontekster, har det yderligere den fordel, at task'ene kan deles om slaven i tiden. Desuden skal der ikke ventes på at en ny task allokeres til slaven, når en task er færdig, hvilket er tilfældet, hvis slaverne kun har en kontekst. Det ses på figuren, at hver gang antallet af kontekster fordobles bliver kørselstiden halveret, dette gør sig gældende fra 1 til 4 kontekster. Forskellen i kørselstiden mellem 4 og 8 kontekster er forholdsvis lille. Dette skyldes at de faktorer, der har stor indvirkning i starten blandt andet muligheden for interkommunikation og muligheden for flere tasks på samme slave, ikke har samme afgørende betydning på arkitekturer med henholdsvis 4 og 8 kontekster, fordi mange af faktorerne allerede udnyttes ved 4 kontekster. 8 kontekster til hver slave kan være med til at gøre systemet mere uoverskueligt for allokeringsstrategier, og dermed besværliggøre beslutningsprocessen. Det ses tydeligt at arkitekturerne, hvor de rekonfigurerbare enheder ligger på forskellige chips, lider under off-chip-kommunikationen, og performer langt dårlige end de tilsvarende arkitekturer implementeret på een chip. Når co-processoren består af flere chips, optimeres performance betragteligt, hvis antallet af kontekster for hver slave stiger. Dette skyldes, at det øger muligheden for at allokere task'ene for den samme applikation inden for den samme chip, og dermed undgå off-chip-kommunikation. I et system hvor co-processoren er delt i flere individuelle chips, stilles der ekstra krav til allokeringsstrategien, fordi det bliver vitalt for systemets performance at task'ene mappes tæt ved hinanden og helst på samme chip.

Det ses på figur 5.6, at forskellen i kørseltiderne mellem et 8x8 array og 10x10 array er minimal, yderligere ses ikke den store forskel i performance i et 8x8 array med henholdsvis 4 og 8 kontekster. Til de kommende forsøg vil der benyttes en single chip co-processor bestående af et 8x8 array med 2 M-noder, 2 C-noder samt 62 slaver hver med 4 kontekster, dvs. co-processoren har 248 ressourcer til rådighed, hvorpå der kan mappes tasks. Da hver slave kan udføre en task af gangen, giver dette mulighed for, at op til 62 tasks kan udføres parallelt. Denne arkitektur har en høj performance og er stadig forholdsvis simpel, og vil blive brugt som udgangspunkt for de videre eksperimenter.

5.5.3 Sammenligning af allokeringsstrategier

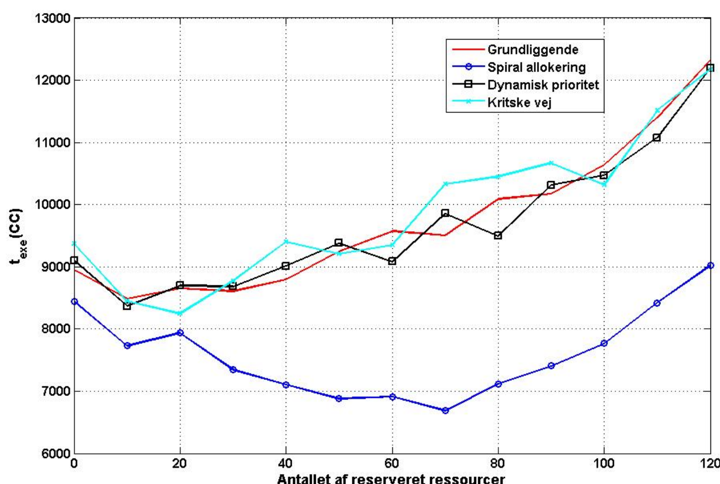
For at sammenligne de forskellige allokerings- og reallokerings-strategier er der udført en række simuleringer. Først undersøges det, hvordan de fire strategier performer, når co-processoren accepterer en ny applikation, så snart der er ledige ressourcer nok til at allokere denne applikation. Simuleringerne er udført med et antal applikationer fra 20 og op til 200. Resultaterne for de fire strategier kan ses på figur 5.7. Det ses, at de tre



Figur 5.7: Kørselstiderne for de 4 *runtime*-strategier udført med et forskelligt antal applikationer

strategier *grundliggende*, *dynamisk prioritet* og *kritiske vej* stort set har den samme kørselstid igennem alle simuleringerne. At der ikke kan konstateres nogen forbedringer for den *dynamiske prioritet* og den *kritiske vej*, kan skyldes flere faktorer, som kan være svære at fastslå, da det er et meget komplekst system at analysere. De tanker og løsningsforslag, der var gjort på forhånd omkring problemstillingerne, som den *grundliggende* strategi introducerede (se afsnit 5.2 og 5.4), viste sig i dette tilfælde ikke at have den forventede effekt. Ideerne bag strategierne kan stadig være gode nok, og der kan ikke umiddelbart peges på den faktor, som gør at co-processorens ydelse ikke optimeres. Strategien *spiral-allokering* har derimod en bedre performance, gennem alle simuleringerne. Dette skyl-

des, at strategien *spiral-allokering* performer langt bedre end de andre strategier, når der benyttes færre applikationer. Ved 40 applikationer er *spiral-allokeringen* 1600 klokperioder bedre end den *grundliggende runtime*-strategi. Dette skyldes at co-processoren ved afviklingen af færre applikationer stadig har mange frie ressourcer som giver *spiral-allokeringen* bedre mulighed for at finde en sammenhængende rute. Det ses af figuren, at *spiral-allokeringen* kun vinder ved et lavt antal af applikationer og derefter holder en konstant afstand til de tre andre *runtime*-strategier. Dette indikerer specielt for *spiral-allokeringen* at det ikke gavner algoritmen, at der startes en ny applikation så snart co-processoren akkurat har ressourcer nok til denne. Dette giver også god mening, da en allokeringsstrategi ikke har mange muligheder for optimering, når systemet er presset til det yderste.



Figur 5.8: Kørselstiderne for de 4 *runtime*-strategier udført med 100 applikationer og varierende antal frie ressourcer

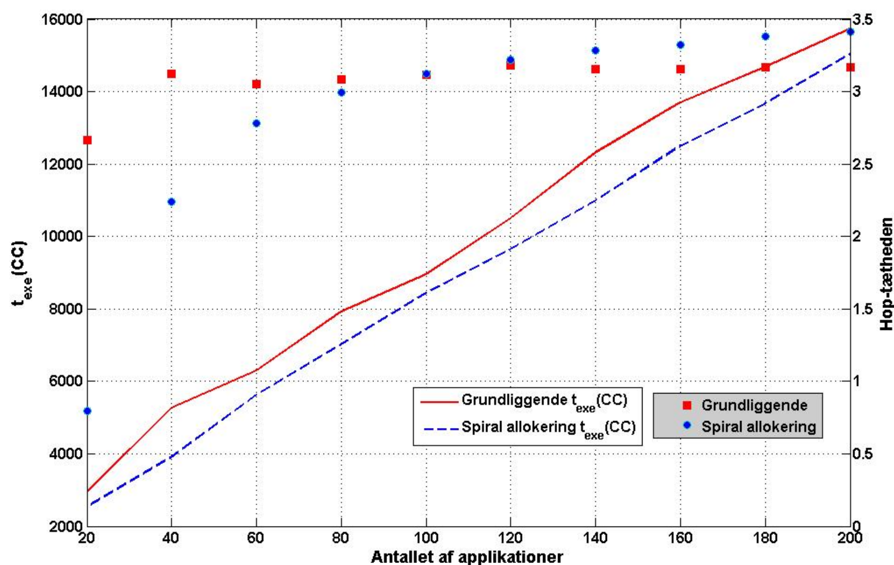
Derfor udformes et nyt eksperiment, hvor co-processoren skal have et ekstra antal frie ressourcer, udover dem der skal benyttes til den indkomne applikation, dette skal gerne give (re)allokeringsstrategierne en større mulighed for at foretage optimeringer. I dette forsøg vil der blive reserveret

op til 120 ressourcer, hvilket svarer til cirka 50% af de samlede ressourcer for co-processoren, og alle simuleringerne udføres med 100 applikationer. Resultatet er vist på figur 5.8. Det ses, at et lille antal af reserverede ressourcer, reducerer kørselstiden for alle simuleringerne. Med omkring 10-20 ressourcer reserveret, opnår de tre strategier: *grundliggende*, *dynamisk prioritet* og *kritiske vej* bedre resultater. Dette skyldes, at der med flere frie ressourcer bliver mulighed for, at reallokere flere tasks adgangen, og task'ene inden for den samme applikation har dermed større chance for at blive reallokeret sammen, og dermed forblive i en tæt klynge, hvilket mindsker kommunikations-*overhead*'et betragteligt. Det ses ligeledes at *spira-allokeringen* reducerer kørselstiden for simuleringen kraftigt, når der reserveres ekstra ressourcer, og denne strategi når først et minimum for kørselstiden ved cirka 70 reserverede ressourcer.

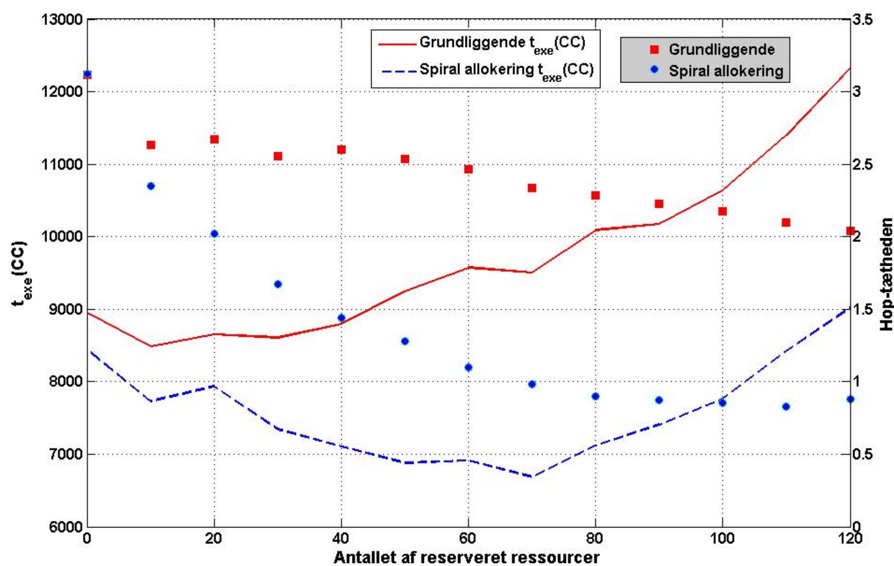
5.5.4 Hop-tætheden

For at få en bedre forståelse af dette fænomen, indføres der en ny variabel kaldet hop-tætheden. Denne specificerer, hvor mange hop der i gennemsnit har været mellem hver kommunikationsopgave under simuleringen. Interkommunikation svarer til 0, da dette foregår internt på samme slave, og 14 er det maksimale antal hop en kommunikationsopgave kan opleve på en 8x8 arkitektur. Hop-tætheden vil derfor for denne arkitektur være et tal mellem 0 og 14. Hop-tætheden giver et godt indtryk af, hvordan *runtime*-styringen har performet, fordi det giver et indblik i, hvor tæt de forskellige tasks inden for applikationen er allokeret.

I det følgende vil hop-tætheden for den *grundliggende* samt *spiral-allokeringen* blive sammenlignet. Der tages kun udgangspunkt i de to nævnte runtime-strategier. Dette skyldes, at der ikke er konstante forbedringer at spore for henholdsvis den *kritiske vej* og den *dynamiske prioritering*. Der er foretaget to simuleringer, hvor hop-tætheden er udregnet. En simulering med et varierende antal applikationer og ingen forhånds reserverede ressource. Den anden simulering er udført med et antal reserverede ressourcer og 100 applikationer. Resultaterne for begge simuleringer kan ses på figur 5.9. Det skal bemærkes, at der benyttes to forskellige y-akser, den venstre angiver kørselstiden, og den højre specificerer hop-tætheden. På figur 5.9(a) vises simuleringen, hvor der ikke er reserveret ressourcer



(a) Forskelligt antal applikationer

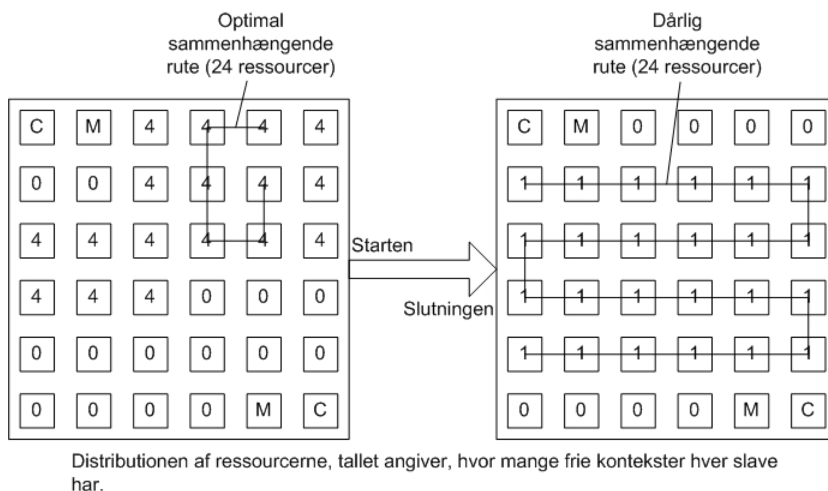


(b) Forskelligt antal frie ressourcer

Figur 5.9: Hop-tætheden for den *grundliggende*- og *spiral-allokeringen*

på forhånd, og det ses, at hop-tætheden for *spiral-allokeringen* er meget lav i forhold til hop-tætheden for den *grundliggende* strategi ved få applikationer. Den minimale hop-tæthed for *spiral-allokeringen* medfører ligeledes en reduceret kørselstid, dette ses ved at sammenligne de to strategier ved 20 applikationer, hvor *spiral-allokeringen* får afviklet applikationerne cirka 1000 klokperioder hurtigere end den *grundliggende* strategi. Hop-tætheden for *spiral-allokeringen* stiger i takt med at antallet af applikationer vokser, hvorimod hop-tætheden for den *grundliggende* er forholdsvis konstant gennem alle simuleringerne jf. figur 5.9(a). At hoptætheden er konstant er en interessant observation, fordi det indikerer, at de reallokeringer, som finder sted i den *grundliggende* strategi, sørger for, at task'ene inden for den samme applikation forbliver forholdsvis tæt ved hinanden, selvom systemet er presset. Hvorimod hop-tætheden for *spiral-allokeringen*, som nævnt stiger kraftigt i starten, og stiger støt i takt med at antallet af applikationer vokser. Dette skyldes at *spiral-allokeringen* ikke udnytter reallokering men istedet er baseret på en optimal allokering af task'ene jf. afnit 5.3.

Problemstillingen er, som vist på figur 5.10 at *spiral-allokeringen* har ideelle arbejdsbetingelser, så længe, co-processoren har mange frie ressourcer. Dette gør sig gældende, når der benyttes få applikationer, hvor co-processoren aldrig bliver fyldt ud. Derimod har *spiral-allokeringen* dårlige arbejdsbetingelser, når de frie ressourcer er distribueret ud over hele co-processoren, som vist på figur 5.10. Denne situation optræder, når der benyttes mange applikationer og co-processoren fyldes ud. Når task'ene vilkårligt færdiggøres parallelt, vil de frie ressoucer blive distribueret tilfædligt ud over co-processoren. Dette gør det besværligt for *spiral-allokeringen*, at finde en sammenhængende rute med mange frie ressourcer, og den rute, der findes er sjældent optimal, fordi den består af mange enkelte slaver med kun en ressource ledig se figur 5.10. På figur 5.9(b) ses hop-tætheden og kørselstiden for *spiral-allokeringen* og den *grundliggende* (re)allokeringsstrategi, hvor der benyttes 100 applikationer, og der reserveres på forhånd mellem 0 og 120 ressourcer, før en ny applikation mappes til co-processoren. Det ses tydeligt at *spiral-allokeringen* performer langt bedre end den *grundliggende* strategi i dette scenarium. Dette skyldes, som vist på figur 5.10, at *spiral-allokeringen* får bedre arbejdsbetingelse, når der på forhånd reserveres mange ressourcer, fordi der lettere kan findes en kort sammenhængende rute med mange



Figur 5.10: *Spiral-allokeringens*-problemer når de frie ressourcer er distribueret ud over co-processoren.

ressourcer. Dette har også stor betydning for hop-tætheden, som falder kraftigt i takt med, at der reserveres flere ressourcer. Når kørseltiderne kun blive bedre til et vist punkt, skyldes det, at for mange reserverede ressourcer reducerer systemets performance. Systemet udnytter i høj grad, at applikationerne kan afvikles parallelt, derfor kan det ikke betale sig at reservere alt for mange ressourcer, fordi alle slaverne derved ikke arbejder samtidigt. Det ses på figur 5.9(b) at hop-tætheden for den *grundliggende runtime*-styringsstrategi ikke opnår en hop-tæthed på under 2, uanset hvor mange ressourcer der reserveres. Dette skyldes, som tidligere nævnt jf. figur 5.1, at den *grundliggende* allokeringsstrategi mapper task'ene diagonalt på netværket, og derved vil der altid være minimum 2 hop mellem task'ene. Simuleringerne viser altså, at *spiral-allokeringen* performer godt, når systemet har mange frie ressourcer, men kan ikke i samme grad opretholde en lav hob-tæthed, når systemet presses af mange applikationer. Den *grundliggende* strategi viser, at det er muligt vha. af reallokering at holde hop-tætheden konstant, selvom systemet presses. Dette fører til, at *runtime*-styringen af dynamiske rekonfigurerbare systemer skal bestå af en hensigtsmæssig allokeringsstrategi, men reallokeringen har ligeledes stor indflydelse på systemet performance specielt, når co-processoren ikke har mange frie ressourcer til rådighed. I det næste kapitel vil der blive sat fokus på en ny algoritme, som prøver at forene

allokeringsstrategien fra *spiral-allokeringen* med en reallokeringstrategi i en ny *runtime*-styringsstrategi.

Kapitel 6

Avanceret reallokerings-algoritme

I foregående afsnit blev en række (re)allokerings-algoritmer sammenlignet med den *grundliggende runtime*-styringsstrategi. Det blev påvist at det er muligt at forbedre systemets performance vha. af en allokeringsstrategi *spiral-allokeringen*. Problemet med denne algoritme er imidlertid, at den ikke benytter reallokering til at optimere performance, men udelukkende optimerer allokeringen af task'ene. Dette betyder at *spiral-allokeringen* ikke har mulighed for at "rydde"op i co-processoren ved hjælp af reallokering. Hop-tætheden for *spiral-allokeringen* bliver større end hop-tætheden for den *grundliggende* strategi, når co-processoren skal håndtere applikationerne, uden forhånds reserverede frie ressourcer. Dette indikerer, at den *grundliggende runtime*-styringsstrategi har mulighed for at "rydde"op i co-processoren ved at reallokere task'ene og dermed mindske hop-tætheden. Derfor vil en ideel løsning være at kombinere egenskaberne fra *spiral-allokeringen* med en form for reallokerings-strategi. Co-processoren har som nævnt sjældent meget regnekraft til rådighed, og derfor skal allokerings- og reallokerings-algoritmerne, der benyttes, være simple, således at beregningstiden ikke vokser voldsomt. Det er tidligere vist, at der er en lineær sammenhæng mellem beregningstiden for *runti-*

me-styringen og co-processorens performance jf. figur 5.4. Derfor er det ikke hengsigtsmæssigt, hvis beregninstiden for reallokerings-strategien yderligere bliver lagt oven i beregningstiden for *spiral-allokeringen*. Masteren, som håndterer allokeringen af task'ene, har derudover til opgave at synkronisere task'ene indbyrdes, således at de udføres i den korrekte rækkefølge jf. afsnit 4.2.4. Dette kræver ikke mange ressourcer, derfor er Masteren inaktiv meget af tiden og har kun en spidsbelastning, når en ny applikation skal mappes til co-processoren. I stedet for at Masteren er inaktiv, kan dens ressourcer benyttes til en *proaktiv* reallokerings-algoritme, som løbende "rydder"op i de tasks, der er allokeret til det dynamiske system. Dette betyder, at der kan implementeres en reallokerings-strategi, som ikke belaster beregningstiden yderligere i forhold til bidaget fra den originale allokerings-strategi.

6.1 Indledende fase

Der var ud fra de simple *runtime*-styringsstrategier, og de forsøg, der var udført, opnået en ide og viden om, hvordan blandt andet hop-tætheden kunne være med til at optimere *runtime*-styringen. En række reallokeringsstrategier blev implementeret og afprøvet i COSMOS modellen. De første optimeringsforsøg drejede sig om en strategi, hvor der blev benyttet et øjebliksbillede af den indeværende ressourcedistribution for hele co-processoren. De indledende forsøg mod en proaktiv strategi havde dog ikke de forventede resultater, og det viste sig at reallokeringsstrategien faktisk havde en negativ indflydelse på co-processorens samlede performance. Dette skyldtes mange faktorer, men en af de centrale var, at mængden af data, der skulle analyseres var for stor, når hele coprocessorens ressourcedistribution skulle tages i betragtning. Ideen var, at optimere ud fra en række restriktioner:

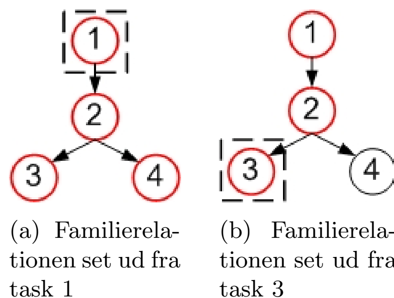
- Mindske kommunikations-*overhead*'et.
- Ikke mistes parallelitet i afviklingen tasks, dette sætter begrænsninger for, hvilke tasks, der kan mappes til den samme slave.
- Task'ene inden for en applikation skal være tæt ved hinanden.

- Alle tasks der reallokeres skal flyttes fra venstre mod højre for at frigive plads i den ene side af co-processoren for at optimere arbejdsbetingelserne for *spiral-allokeringen*.

Fordi der blev benyttet for mange restriktioner og taget udgangspunkt i hele co-processorens ressourcedistribution, viste det sig, at der i sidste ende stort set ikke blev reallokeret tasks i co-processoren. Derfor var de forventede forbedringer ikke eksisterende, og det viste sig faktisk at co-processoren havde en dårligere ydelse i forhold til den *grundliggende runtime*-strategi.

6.2 Proaktiv reallokerings-algoritme

For at begrænse omfanget af den information, der skal analyseres, tages der ikke et øjebliksbillede af hele task-distributionen for co-processoren, men kun ressourcedistributionen inden for en enkelt applikation og dennes tasks. Den grundlæggende tankegang i denne reallokerings-strategi er, at der kun reallokeres børn, børnebørn osv. i task-grafen hen til deres respektive forældre. Det helt centrale i denne fremgangsmåde er, at kommunikations-*overheadet*'et reduceres, fordi børn altid skal kommunikerer med deres forældre. Næste vigtige aspekt er, at der ikke tabes parallelitet i afviklingen af applikationen, fordi børn og børnebørn aldrig kan udføres parallelt med deres forældre. Det er vigtigt at forstå, hvordan

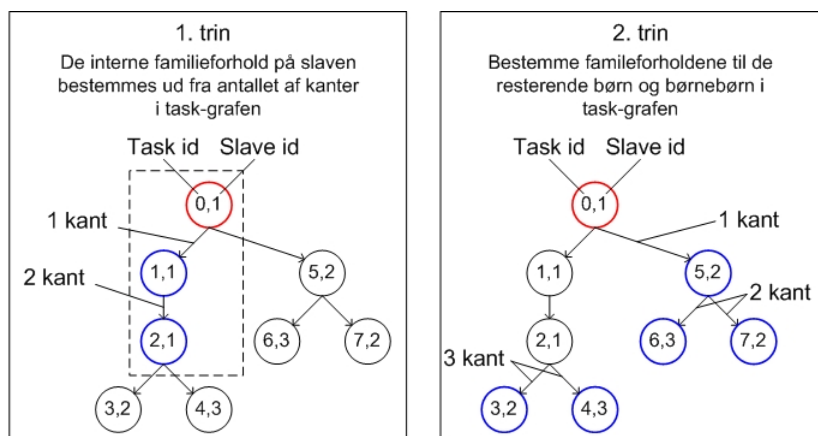


(a) Familierelationen set ud fra task 1

(b) Familierelationen set ud fra task 3

Figur 6.1: Definitionen af famileforholdene

familierelationen mellem to tasks i denne sammenhæng er defineret. På figur 6.1 er vist en simpel task-graf bestående af 4 tasks. Figur 6.1(a) viser familieforholdene set ud fra task1, og det kan ses, at denne er relateret til alle de andre tasks i task-grafen. På figur 6.1(b) er task3's familieforhold illustreret, og det ses, at denne kun er relateret til task'ene 1 og 2. Dette skyldes, at der ikke er en direkte rute mellem task3 og 4 som går gennem enten børn eller forældre i task-grafen. Antallet af kanter i task-grafen mellem de tasks, der er i familie med hinanden, benyttes som mål for, hvor tæt knyttet de er på hinanden. For at lokalisere en task, der kan reallokeres, skal der findes en task, der kan benyttes som udganspunkt for søgningen. Som udgangspunkt for søgningen kan der kun benyttes en task, der udelukkende har børn/børnebørn osv. på den slave, den er mappet til. Når der er fundet en task, der kan søges ud fra, er første trin, som vist på figur 6.2 at bestemme de interne familerelationer på slaven. Andet trin går som vist ud på at finde familierelationen mellem de potentielle tasks, der kan reallokeres. Dette inkludere kun de tasks, som ikke er mappet til samme slave, som den tasks der søges ud fra. Det ses på figur 6.2, at task'ene 5,6 og 7 er knyttet gennem henholdsvis 1 og 2 kanter til den task, der søges ud fra. Dette er netop antallet af kanter der er i den interne familierelation på slaven se trin 1, derfor kan disse tasks ikke reallokeres. Dette efterlader task'ene 3 og 4, som har en relation på 3 kanter til den task, der søges ud fra, og derfor er det muligt at reallokere disse.

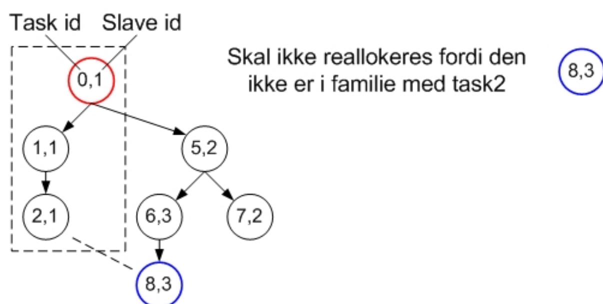


Figur 6.2: Tankegangen bag den *proaktive* reallokerings-strategi

Den proaktive reallokerings-strategi forsøger at optimere allokeringen af task'ene for en applikation, når et af to forhold gør sig gældende. For det første, hvis den task der færdiggøres er allokeret til en slave, som stadig indeholder 2 eller flere task fra applikationen, så søges der efter en ny task, der kan reallokeres til denne slave. Det andet forhold er, hvis task'ene enkeltvis er distribueret ud over flere slaver, så forsøger den proaktive algoritme at "rydde"op. Det præcise forhold er at $2 \cdot \#slaver > \#tasks$ for den pågældende applikation. Dette forhold indikerer, at der er færre end 2 tasks pr. slave, og derfor kan der være en mulighed for at samle task'ene på et færre antal slaver.

6.2.1 Implementeringen

Implementeringen af den *proaktive* algoritme er udført vha. af en række funktioner, hvoraf de fleste er rekursive funktioner [5], som kan søge i task-grafer. Hver gang Masteren får information fra en task om, at den er færdig kaldes funktion *clean_system*. Denne tager et øjebliksbillede af task-distributionen ved at bruge intern information omkring allokeringen af de resterende tasks i applikationen, men der kommunikeres også med de slaver, hvorpå de resterende tasks i applikationen befinder sig, for at indhente information omkring antallet af frie ressourcer i co-processoren. Den rekursive funktion *find_edge_distance* er i stand til at finde antallet af kanter i en graf mellem to tasks, hvis disse har relation til hinanden. Denne funktion benyttes til at bestemme de interne familieforhold mellem de tasks, som befinder sig på samme slave. Funktionen *find_child* er ligeledes rekursiv og leder efter børn i task-grafen ud fra en specifik task. *Find_child* kan kaldes med information omkring antallet af kanter, der ikke må være mellem det barn, der muligvis kan findes, og den task der søges ud fra. Den sidste funktion, der benyttes er *find_relation_back*, som ligeledes er rekursiv. Hvor de to foregående gik gennem børnene i task-grafen, går denne baglæns gennem forældrene til en task. Dette gøres for at undgå den situation, som er vist på figur 6.3, hvor task'en (task8), der findes af *find_child*, er i familie med den task, der søges ud fra, og samtidig opfylder kravet om, at den ikke har samme antal kanter til den oprindelige task, og de tasks som allerede befinder sig på mål-slaven. Problemet er, som vist på figur 6.3, at det ikke er en klog beslutning at reallokere task'en, fordi den ikke er i familie med de resterende tasks på

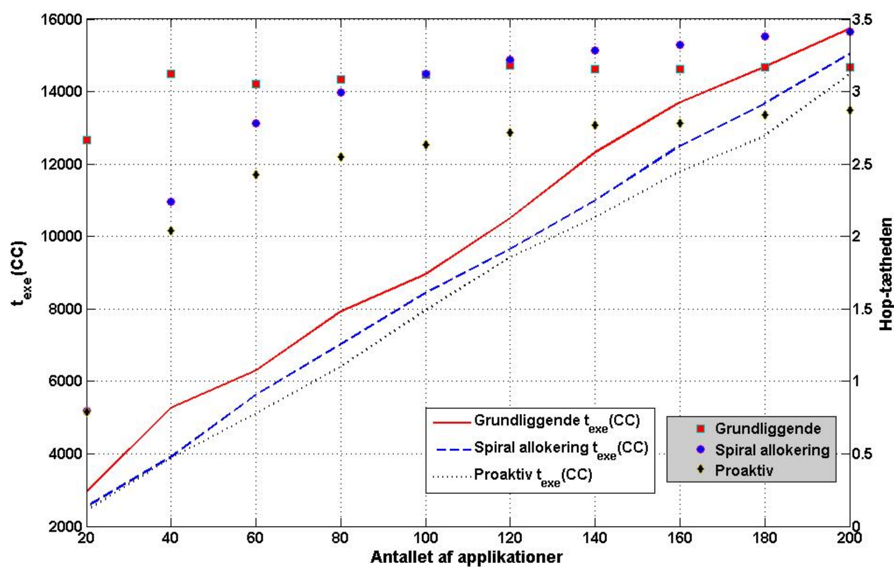


Figur 6.3: Ilustration af hvornår der ikke skal reallokeres. Det er funktionen *find_relation_back* der skal detektere disse tilfælde så de kan undgås

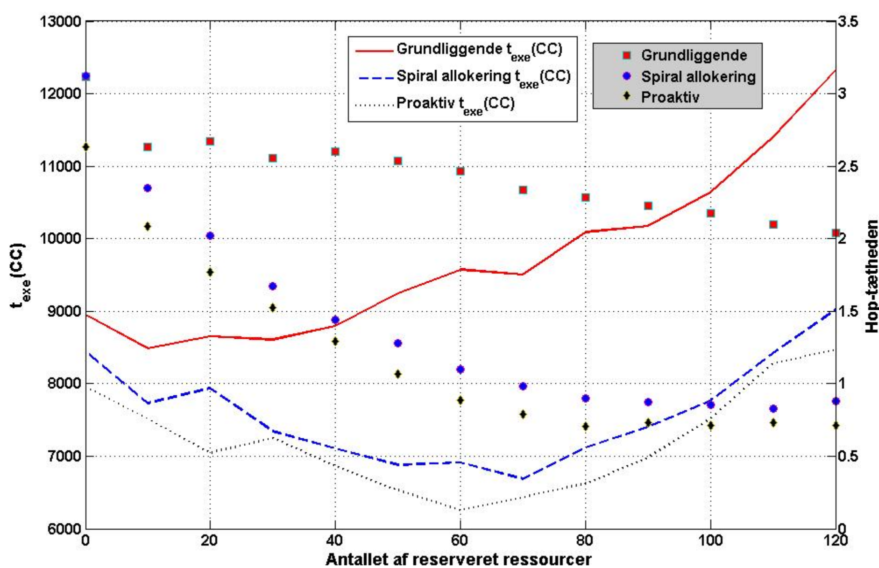
slaven. Derfor benyttes funktionen *find_relation_back* til at afgøre om den task, der reallokeres, også er i familie med de tasks, der befinder sig på mål-slaven i forvejen ved at søge baglæns gennem task-grafen. Se bilag G for den fulde kode. Der er også implementeret en grådig proaktiv algoritme, som kun benytter funktionen *find_child*. Denne reallokerer altid et barn, hvis dette kan findes uden at tage andre forbehold. Denne strategi har til formål at vise, at reallokering kan have en negativ indflydelse på co-processorens samlede ydelse, hvis det ikke gøres med måde.

6.3 Simuleringer

For at sammenligne performance af den *proaktive* strategi med de eksisterende (re)allokerings-strategier, *spiral-allokeringen* og den *grundliggende* er der foretaget en række simuleringer. De bygger på det scenarium, der blev benyttet til at sammenligne performance af de simple algoritmer jf. afsnit 5.5.3. I den første simulering er antallet af applikationer varierende, og der er ingen forhånds reserverede ressourcer i co-processoren. Resultatet af simuleringen kan ses på figur 6.4(a), hvor resultaterne for den *grundliggende* og *spiral-allokeringen* ligeledes er vist. Det ses, at den *proaktive runtime*-styringsalgoritme performer bedre end de andre strategier. Specielt lægges mærke til, at hop-tætheden for den *proaktive* algoritme er lavere end for de andre strategier og at den altid er mindre



(a) Forskelligt antal applikationer

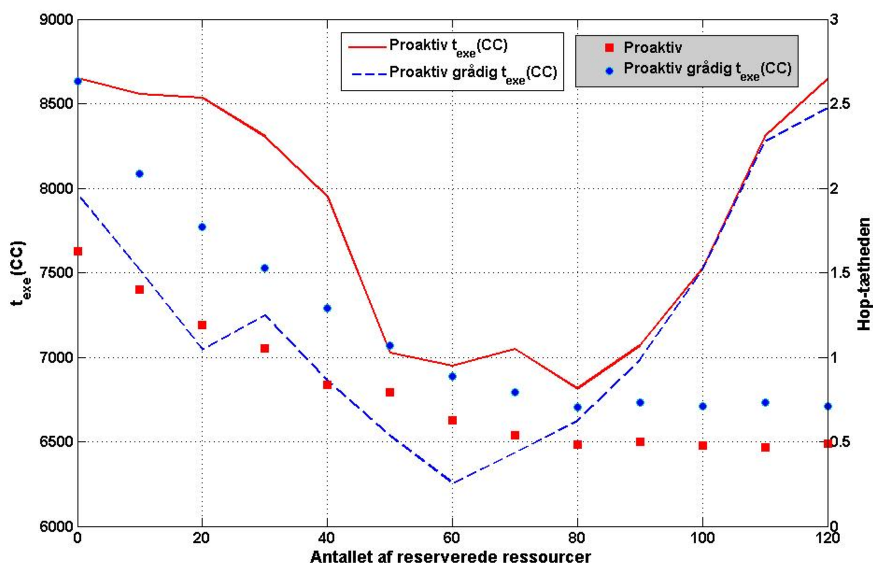


(b) Forskelligt antal frie ressourcer

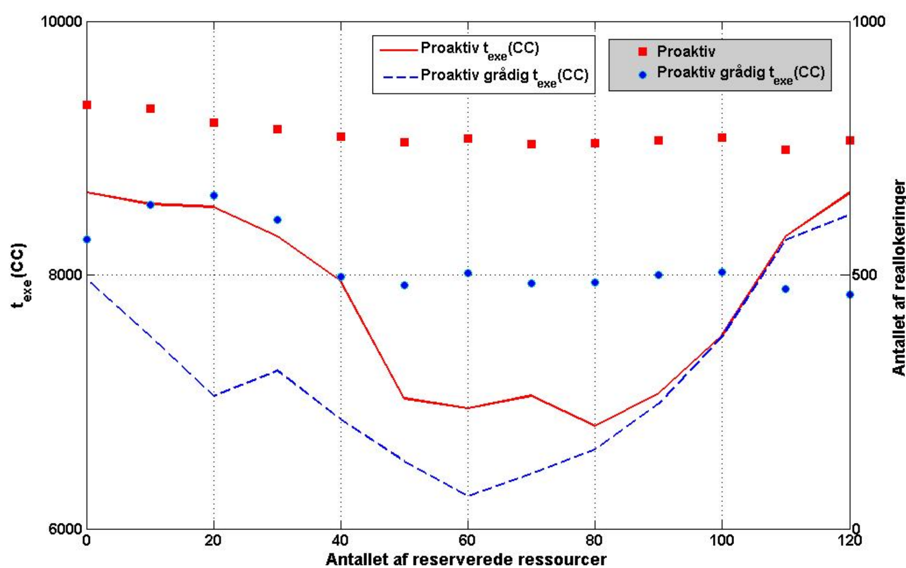
Figur 6.4: Hop-tætheden for den *proaktive*- *grundliggende*- og *spiral-allokeringen*

end hop-tætheden for den *grunliggende* strategi.

På figur 6.4(b) er resultaterne for simuleringerne hvor der på forhånd er reserveret frie ressourcer sammenlignet. Det ses, at den *proaktive*-algoritme igen performer bedre end de andre (re)allokerings-strategier. Specielt ved 60 reserverede ressourcer, hvilket svarer til en $\frac{1}{4}$ af co-processorens samlede ressourcer, performer den proaktive algoritme bedst. Den er 1500 klokperioder bedre end *spiral-allokeringen* og 3200 klokperioder bedre end den *grundliggende*. Dette skyldes, at reallokeringerne, som den *proaktive*-strategi foretager, er med til at sænke hop-tætheden yderligere i forhold til *spiral-allokeringen*, og dermed optimere systemets performance. At reallokering også kan have en negativ indflydelse, er vist på figur 6.5. Figur 6.5(a) viser en sammenligning mellem kørseltiden og hop-tætheden for den *grådige*- og den *proaktive*-algoritme. Det ses, at den *grådige proaktive runtime*-styring ikke har en særlig god ydelse, selvom hop-tætheden er mindre end hop-tætheden for den *proaktive*-algoritme. Dette skyldes to ting. Der foretages alt for mange reallokeringer, som i sig selv tager tid, se figur 6.5(b), og det andet er, at alle reallokeringer gør, at co-processoren ikke udnytter, at mange af task'ene i en applikation kan afvikles parallelt. Alt for mange tasks, der kan afvikles parallelt reallokeres til den samme slave, hvilket har en negativ indflydelse på co-processorens performance. Yderligere kan det bemærkes, at antallet at reallokeringer for den *proaktive*-strategi er et godt stykke under antallet for den *grådige proaktive*. Desuden ses det, at den *proaktive*-algortime tager et spring i antallet af reallkoringer, når der er reserveret 30 ressourcer på forhånd. Dette kan skyldes, at *spiral-allokeringen* begynder at få bedre arbejdesbetingelser og kan allokere task'ene bedre, således at der ikke i samme grad er behov for reallokeringer. Generelt er det påvist, at den *proaktive*-algoritme optimerer ydelsen for den dynamiske rekonfigurerbare co-procssor. Dette indikerer, at fremtidige løsninger på problemet omkring *runtime*-styring af dynamiske rekonfigurerbare systemer kan bygge på proaktive algoritmer.



(a) Hop-tæheden



(b) Antallet af reallokeringer

Figur 6.5: Sammenligning af den *proaktive* og den *grådige proaktive* i et scenarium, hvor der er reserveret frie ressourcer i co-processoren.

I tabel 6.1 ses en opsamling af de bedste resultater, som hver runtime-strategi opnåede. De blev for alle strategiers vedkommende opnået i scenariet med forhånds reserverede ressourcer. Tabellen angiver også den procentvise forbedring i forhold til den grundliggende *runtime*-strategi.

Strategi	**# af frie rsc**	**Kørselstid**	**Forbedring i %**
Grundliggende	10	8483	na
Kritisk vej	10	8248	3
Dynamisk prioritet	10	8369	1
Spiral-allokeringen	70	6689	21
Proaktiv	70	6255	26
Proaktiv grådig	80	6813	19

Tabel 6.1: Samlede overblik over de bedste resultater der blev opnået for hver strategi

Kapitel 7

Diskussion

Dynamiske rekonfigurerbare systemer er et komplekst område, der indeholder mange problemstillinger. Et af de første problemer er, hvordan den hardware, som skal understøtte rekonfigureringen, skal designes. Med udgangspunkt i kommercielle FPGA'er er der foretaget en undersøgelse af, hvorvidt de understøtter dynamisk partiel rekonfigurering i en sådan grad, at det er muligt at implementere større rekonfigurerbare systemer. Der er foretaget en række forsøg, som viser, at det er muligt med de FPGA'er og værktøjer, der er til rådighed i dag, at få simple rekonfigurerbare systemer op at køre. Forsøgene er meget simple i forhold til de rekonfigurerbare systemer, der senere forsøges belyst gennem en model. Implementering af et dynamiske rekonfigurerbart system, der minder om modellen, ligger et stykke ude i fremtiden. De ændringer, der er foretaget af arkitekturen i *Virtex-4* familien i forhold til tidligere *Virtex*-familier og *Spartan*-serien, giver forhåbninger om en potentiel brug af FPGA'er inden for dynamiske rekonfigurerbare systemer. Det der adskiller arkitekturerne mellem *Virtex-4* familien og de resterende familier er, at en konfigurerings-frame jf afsnit 3.1 ikke spænder over hele FPGA'en højde, men kun 16 CLB's. Dette muliggør, at FPGA'en kan deles op i flere små moduler, som hver især kan gennemgå en partiel rekonfigurering. Dette

er afprøvet gennem simple forsøg, og fungerer upåklageligt.

Der er stadig mange faldgruber forbundet med implementeringen af et modulbaseret rekonfigurerbart system. Det kan f.eks. nævnes, at den kommunikation, som skal foregå mellem rekonfigurerbare moduler, kræver, at der indsættes bus-macro'er som vist på figur 3.5. Implementeres der et system bestående af et 4x4 netværk af RU's, som det vist på figur 5.5, og skal de 16 rekonfigurerbare moduler kommunikere indbyrdes med f.eks. en 32 bit bidirektionel bus, skal der benyttes 384 CLB's alene til kommunikationen. Dette svarer til $\sim$ 15% af det samlede antal CLB's, som den *virtex-4(XC4VLX25)*, der blev benytte i disse forsøg har [18]. Antallet af CLB's, der benyttes til kommunikationen, kan mindskes ved at benytte en separat ikke rekonfigurerbar netværks-controller, som håndterer kommunikationen med andre netværkscontrollere, og er forbundet med hver sit rekonfigurerbare modul.

Xilinx har foreslået 2 forskellige design-*flows* for at opnå partiel rekonfigurering af deres FPGA'er. Begge disse design-*flows* er afprøvet, og det viser sig, at til store systemer er det kun det modulbaserede design-*flow* jf. figur 3.4, der virkelig kan benyttes. Den differensbaserede metode egner sig kun til meget små ændringer, udført i FPGA-editoren, som f.eks.at ændre funktionaliteten af en LUT. Det mest realistiske bud på en implementering af et stort dynamisk rekonfigurerbart system består i, at benytte et netværk af FPGA'er, som hver især udgør en RU eller et lille antal RU's.

Hardware-delen af projektet havde til formål at gennemgå mulighederne for en implementering af et dynamisk rekonfigurerbart system. Den anden halvdel af projektet bestod i at opnå en større forståelse af de faktorer, der gør sig gældende i dynamiske rekonfigurebare systemer gennem modellering. Udgangspunktet var et simuleringsmiljø for co-processor-koblede rekonfigurerbare arkitekturer kaldet COSMOS, som blev udviklet af Kehuai Wu i forbindelse med hans P.hd. projekt [13]. COSMOS modellen dannede grundlaget for den model, der er blevet udviklet i løbet af dette projekt. Modellen blev opdateret på en række kritiske punkter, så den blandt andet blev i stand til at håndtere store simuleringer bestående af mange applikationer. Et stort problem ved den oprindelige udgave af COSMOS var dens manglende fleksibillitet, hvilket bevirkede, at det var

meget tidskrævende at opsætte forskellige simuleringer. Dette blev ændret ved at oprette en række filer, som specificerer de centrale elementer i modellen, her i blandt arkitekturen på co-processoren der simuleres på, applikationer der skal afvikles på co-processoren, og en fil der beskriver den overordnede simulering f.eks. rækkefølgen af applikationerne, der skal afvikles og antallet af applikationer. Derudover blev datastrukturen i den oprindelige COSMOS opdateret med henblik på to faktorer. For det første kunne den oprindelige datastruktur ikke håndtere store simuleringer, det andet problem var at optimere simuleringshastigheden. COSMOS modellen i den opdaterede version er et stærkt værktøj til simuleringer af co-processor-koblede rekonfigurerbare systemer. Modellens styrke er dens fleksibilitet, samt den detaljerede modellering af de vigtige enheder i et dynamisk rekonfigurerbart system. Dette gør at modellen indfanger mange af de vigtige aspekter, som har indflydelse på afviklingen af applikationerne på en rekonfigurerbar co-processor. Til tider kan det høje niveau af detaljer i COSMOS modellen gøre det svært, at forstå og analysere de resultaterne modellen leverer, fordi det er vanskeligt at overskue alle de faktorer, som influerer systemets performance. Her kan nævnes, kommunikationen mellem task'ene, er det interkommunikation eller foregår det over flere hop, reallokeringen, allokeringen, *scheduleringen* af task'ene på de enkelte slaver, parallismen af afviklingen af applikationerne. Disse nævnte faktorer medvirker til, at det til tider kan være frustrende at arbejde med modellen, fordi de tanker og ideer, der opstår til optimeringer, sjældent har den effekt der forventes, fordi der er så mange faktorer at tage højde for, og det er samspillet mellem alle disse, der bestemmer, hvordan systemet performer. Specielt i udviklingen af den *proaktive*-allokerings-strategi var der mange spildte forsøg, fordi ideen og tanken, der lå til grund for reallokerings-strategien, ikke havde den forventede effekt se figur 6.5, som viser det dårlige resultat for den *grådige-proaktive*-algoritme.

En af modellens svagheder er implementeringen af netværket onchip (NoC), som er foretaget på et forholdsvis højt abstraktions niveau. Der tages ikke højde for de situationer, som kan opstå i et "rigitigt"NoC f.eks. trafikpropper se figur 4.13(b). I den oprindelige udgave af COSMOS var NoC klart flaskehalsen i systemets performance jf. figur 4.12. Dette er optimeret ved at tillade, at flere kommunikationsopgaver kan afvikles parallelt. På den måde sikres det, at NoC ikke længere er flaskehalsen for

co-processorens performance, og dermed ikke influerer co-processorens ydelse. Der kan sagtens sættes restrektioner på båndbredden af NoC i COSMOS ved at reducere antallet af parallelle kommunikationsopgaver som kan afvikles samtidigt se figur 4.12. Dette er ikke gjort for netop at undgå at NoC skulle udgøre endnu en afgørende faktor for co-prcessreons ydelse. I en reel implementering af et dynamisk rekonfigurerbart system er et NoC implementeret vha. af en delt bus ikke realistisk og i høj grad ikke skalerbar.

På baggrund af de simuleringer, der blev udført, kan det statiske grundlag diskuteres. Alle simuleringer, der ikke havde til formål at undersøge, hvordan antallet af applikationer påvirker co-processoren, blev udført med minimum 100 applikationer. I afsnit 5.5.1 blev det fundet, at hver af applikationerne, der er benyttet til simuleringerne i gennemsnit, består af 12 tasks, dette giver 1200 tasks, som skal afvikles på co-processoren. Dette er nok til at fylde co-processorens samlede antal ressourcer fem gange, hvis der benyttes en 8x8arkitektur, hvor hver slave har 4 kontekster. Det viste sig endvidere, at der er en forholdsvis lineær sammenhæng mellem antallet af applikationer, der afvikles i løbet af en simulering og den samlede kørselstid jf. figurerne 5.7, 5.9(a) og 6.4(a). Det indikerer, at der ikke kan vindes meget ekstra information ved at foretage meget store simuleringer, men det udelukker heller, ikke at der stadig kan være spændende aspekter, som først træder i kraft, når der udføres virkelig store simuleringer med over 1000 applikationer.

Der er foretaget et studie af *runtime*-styringen af den dynamisk rekonfigurerbare co-processor. Det viste sig, at co-processorens performance i høj grad er afhængig af den *runtime*-styringsstrategi, der benyttes. Den *grundliggende runtime*-strategi blev analyseret, og der blev foreslået en række simple (re)allokerings strategier, der havde til formål at optimere nogle af de åbenlyse problemstillinger, den *grundliggende* strategi introducerede. *Spiral-allokeringen* viste hurtigt, at den var de andre strategier overlegen, og var den simple *runtime*-styrtingsstrategi, der fik den bedste ydelse ud af co-procoessoren. *Spiral-allokeringen* er en strategi, der kun benytter allokering den optimerer ikke co-prcoessoren vha. reallokeringer. For at optimere *runtime*-styringsstrategien yderligere er det undersøgt, hvorledes en strategi, der kombinerer allokeringsegenskaberne fra *spiral-allokeringen* med en reallokerings-strategi kunne udformes. Dette førte

til udviklingen af en proaktiv *runtime*-styringsstrategi, som viste sig at øge den samlede performance for co-processoren. Det er en klar fornemmelse, at co-processorens ydelse kan forbedres yderligere, ved at optimere *runtime*-styringen. De proaktive *runtime*-styringsalgoritmer, der er forslået i dette projekt er en guideline for, hvordan fremtidens algoritmer til styring af dynamiske rekonfigurerbare systemer kan se ud.

KAPITEL 8

Fremtidigt arbejde

I dette afsnit vil en række forslag til, fremtidigt arbejdet på COSMOS modellen og hardwaredelen blive gennemgået. Det næste skridt i forbindelse med undersøgelsen af FPGA'erne og deres brug i dynamiske rekonfigurerbare systemer, kan gå ud på, at designe og implementere en kontroller til ICAP-interfacet på FPGA'en. Dette vil gøre FPGA'en i stand til at foretage partiel rekonfigurering uden brug af en ekstern kontroller, som det er tilfældet, når der benyttes et JTAG kabel til rekonfigureringen. Dette kan være med til at give bedre bestemmelse af rekonfigureringstiden og åbne op for en undersøgelse af hvilke muligheder der er for at optimere rekonfigureringstiden. Derudover kan værktøjet *Planahead* afprøves. Dette program hjælper i den indledende designfase af *embedded*-systemer, og kan være med til at lette designfasen for dynamiske rekonfigurerbare systemer, og i særdeleshed placeringen af de rekonfigurerbare regioner og bus-macro'erne.

Der er mange spændende aspekter i dynamiske rekonfigurerbare systemer, som kan belyses gennem brugen af COSMOS modellen. Modellen kan udvides, så den kan håndtere mere information omkring størrelsen på de enkelte RU's, f.eks. kan der specificeres et konkret antal CLB's,

som hver slave består af. Dette kan derefter bruges til at undersøge, om task'ene i en applikation eventuelt skal splittes eller samles, som vist på figur 4.3 for at optimere task'enes størrelse til de pågældende slaver. TGFF er et værktøj, der producerer task-grafer efter brugerspecificerede inputs, for at øge fleksibiliteten af modellen yderligere skal den være i stand til at læse output-filerne fra TGFF direkte uden, at det er nødvendigt for brugere at rette dem til. Det vil i endnu højere grad øge fleksibiliteten af modellen, fordi det giver en hurtig og nem måde at skabe mange forskellige applikationerne og benytte disse under simuleringerne i COSMOS modellen. Skal modellen forbedres yderligere, kan der implementeres en mere realistisk arkitekturmodel for NoC. Denne skal simulere, at hver rekonfigurerbare enhed i netværket kommunikerer vha. link til deres nærmeste naboer. Det vil skabe mulighed for, at netværket f.eks. kan møde trængsel ved et link, som kræver at en kommunikationsopgave enten tager en anden rute, eller venter på at linket bliver frit. Dette vil gøre modellen mere realistisk i forhold til en virkelig implementering, hvor NoC sagtens kan være flaskehalsen for co-processorens performance.

Angående *runtime*-styringsstrategier vil et virkeligt spændende aspekt være, at implementere en heuristisk, som benyttes i statisk allokering [2] [3] [6]. En heuristisk kan afsløre en *lower bound* for co-processorens performance. I realiteten vil det ikke være muligt at bruge en heuristisk, fordi beregningstiden er for stor til dynamisk allokering, men dette har ingen indflydelse, så længe modellen benyttes som grundlag for eksperimenterne. Denne *lower bound* vil udgøre et mål for den videre forskning i optimeringen af co-processorens ydelser vha. simple *runtime*-styringsstrategier. For at opnå mere information om den virkelige beregningstid for de forskellige *runtime*-strategier, kan der foretages *profiling*. Dette kan udføres vha. kendte inputs, f.eks. kan hver allokeringsstrategi tilføres data fra 10 applikationer, og vha. *profiling* kan der findes et meget præcist estimat af de enkelte strategiers beregningstid. Dette kan så igen føres tilbage til COSMOS modellen, som skal inkludere beregningstiden for de forskellige strategier. Modellens store fleksibilitet kan udnyttes til at foretage simuleringer, der har til formål at afdække sammenhængen mellem co-processorens performance og co-processorens arkitektur. Det kan være yderst interessant at undersøge, f.eks. hvordan placeringerne af Masterne og Coordinatorerne i netværket påvirker co-processorens ydelse. Der kan yderligere foretages simuleringer, der undersøger om flere M-noder opti-

merer performance i store arkitekturer. Som det er nævnt, er der stadig mange spændende aspekter i dynamiske rekonfigurerbare systemer, som venter på at blive belyst gennem COSMOS modellen.

Kapitel 9

Konklusion

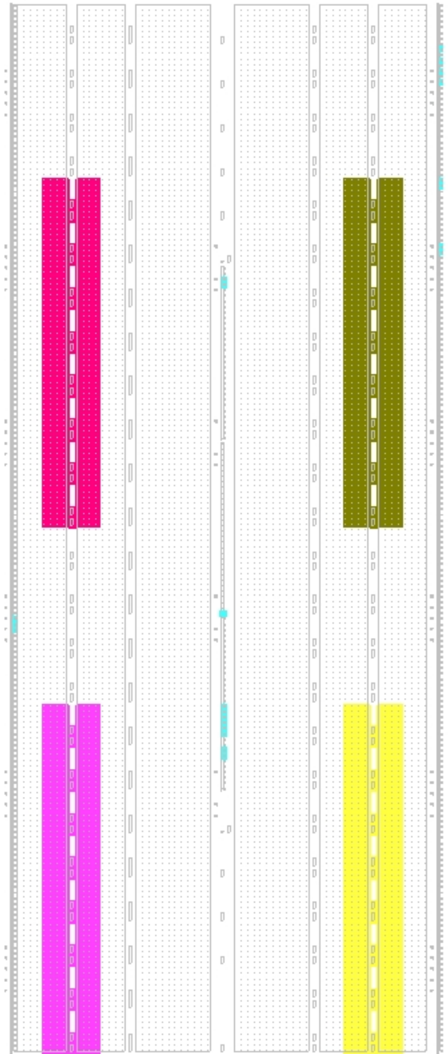
Der er gennem dette projekt opnået en større forståelse af kommercielle FPGA'er og deres interne arkitektur. Denne indsigt er benyttet i forbindelse med implementeringen af partielle rekonfigurerbare systemer på FPGA'en. Udgangspunktet er *Virtex-4*-familien, hvis arkitektur er opdateret på et helt central område, som har gjort det muligt at benytte partielle rekonfigurerbare moduler, der ikke spænder over hele højden af FPGA'en. Det er lykkedes at designe og implementere et rekonfigurerbart system, hvor hver enkelt modul kan gennemgå partiel rekonfigurering. Implementeringen af større systemer bestående af mange rekonfigurerbare moduler på samme FPGA kræver stadig en række opdateringer i FPGA'ens arkitektur, før det er en realistisk mulighed. En alternativ løsning kunne være at benytte flere FPGA'er sat sammen i et netværk. For at opnå en større indsigt i rekonfigurerbare systemer er der udviklet og videreudviklet et simuleringsmiljø COSMOS for co-processor-koblede rekonfigurerbare arkitekturer. COSMOS modellen er i løbet af projektet opdateret på en række væsentlige punkter, som blandt andet har gjort modellen yderst fleksibel og i stand til at håndtere store simuleringer. Ved hjælp af modellen er der opnået et indblik i dynamiske rekonfigurerbare arkitekturer. Dette har medvirket til en bedre forståelse af de

faktorer, der har indflydelse på *runtime*-management, og dermed gjort det muligt at finde strategier, der optimerer co-processorens performance. COSMOS er benyttet til grundige undersøgelser af *runtime*-styringens betydning for rekonfigurerbare systemers ydelse. Ved hjælp af modellen er en række *runtime*-strategier blevet afprøvet, hvoraf en strategi baseret på en proaktiv algoritme, viste sig at være de andre strategier overlegen. Dette kan være med til at sætte en guideline for, hvordan fremtidige *runtime*-styringsalgoritmer skal designes for at optimere udnyttelsen af potentialet i dynamiske rekonfigurerbare systemer.

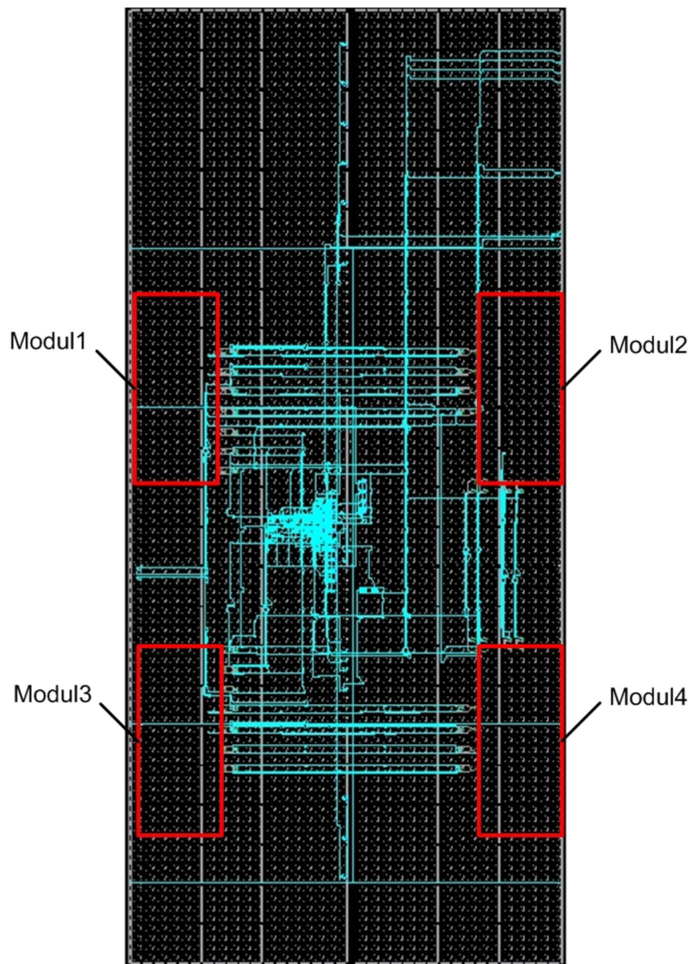
Bilag A

Screenshot

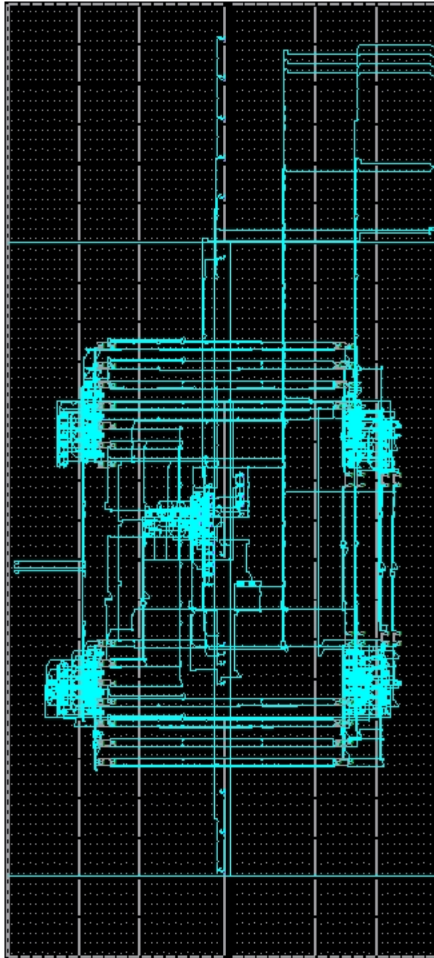
A.1 Floorplanner



A.2 FPGA_editor statisk design



A.3 FPGA_editor komplette design



Bilag B

Task's files

B.1 Task0

no_of_task
8

no_of_comm_task
10

Priority
1

Execution time
104,127,104,105,136,105,136, 93

Deadline
1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000

rsc_partition
1,1,2,1,1,1,2,1,0,0,0,0,0,0,0,0,0,0

no_of_rsc_opt_req
2

the structure of the communication task is as follows
source−>destination:communication run time
Communication
0−>1:53
0−>2:49
1−>3:45
2−>3:59
2−>4:48
3−>4:50
4−>5:73
4−>6:76
6−>7:48
3−>7:76

Critical Path
0−>1−>3−>4−>6−>7

B.2 Task1

no_of_task
17

no_of_comm_task
19

Priority
2

Execution time
118,120,120,105,127,99,127,137,101,101,72,100,127,112,137,112,100

Deadline
1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000

rsc_partition
1,1,2,1,1,2,1,2,3,1,2,1,1,2,3,3,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

no_of_rsc_opt_req
4

the structure of the communication task is as follows
source−>destination:communication run time
Communication
0 −>1:51

0 −>2:50
1 −>3:64
3 −>4:57
3 −>5:55
1 −>5:48
4 −>6:63
5 −>7:45
5 −>8:44
6 −>9:73
6 −>10:55
9 −>11:68
11−>12:54
9 −>13:73
8 −>14:42
7 −>14:52
10−>15:76
8 −>15:44
10−>16:55

Critical Path
0−>1−>3−>4−>6−>9−>11−>12

B.3 Task2

---

no_of_task
15
no_of_comm_task
16

Priority
3

Execution time
104,110,137,112,100,72,120,110,110,136,140,140,93,79,136

Deadline
1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000

rsc_partition
1,1,2,2,1,3,1,1,2,4,2,3,4,3,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

no_of_rsc_opt_req
4

the structure of the communication task is as follows
source−>destination:communication run time
Communication
0 −>1:44
0 −>2:54
2 −>3:76
1 −>4:55
2 −>4:55
1 −>5:60
4 −>6:64
6 −>7:64
3 −>8:57
3 −>9:47
8 −>10:59
8 −>11:68
6 −>12:68
5 −>13:47
10−>14:49
12−>14:71

Critical Path
0−>2−>3−>8−>10−>14

B.4 Task3

---

no_of_task
7

no_of_comm_task
7

Priority
4

Execution time
120,105,100,127,105,118,120

Deadline
1000, 1000, 1000, 1000, 1000, 1000, 1000

rsc_partition
1,1,1,2,2,2,1,0,0,0,0,0,0,0

no_of_rsc_opt_req

2

the structure of the communication task is as follows
source−>destination:communication run time
Communication
0−>1:53
1−>2:44
1−>3:63
3−>4:73
4−>5:57
3−>6:47
2−>6:71

Critical Path
0−>1−>3−>4−>5

B.5 Task4

no_of_task
16

no_of_comm_task
18

Priority
5

Execution time
110,104,120,79,100,101,118,140,127,118,101,140,104,120,127,100

Deadline
1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000

rsc_partition
1,1,1,1,2,2,2,2,3,3,1,1,3,1,2,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

no_of_rsc_opt_req
3

the structure of the communication task is as follows
source−>destination:communication run time
Communication
0 −>1:49
1 −>2:59

2 −>3:52
2 −>4:54
1 −>4:74
4 −>5:64
4 −>6:74
5 −>6:45
6 −>7:48
6 −>8:76
8 −>9:60
3 −>10:42
10−>11:64
10−>12:47
11−>13:57
13−>14:64
7 −>14:65
13−>15:49

Critical Path
0−>1−>2−>4−>5−>6−>7−>14

Bilag C

Task-grafer for de fem applikationer

TASK_GRAPH 0
Period= 440
In/Out Degree Limits= 2 / 2

TASK_GRAPH 1
Period= 880
In/Out Degree Limits= 2 / 2

TASK_GRAPH 2
Period= 880
In/Out Degree Limits= 2 / 2

TASK_GRAPH 3
Period= 440
In/Out Degree Limits= 2 / 2

TASK_GRAPH 4
Period= 880
In/Out Degree Limits= 2 / 2

Bilag D

Udsnit af en log-fil fra COSMOS

Listing D.1: Udsnit af en log-fil

```
aster 62 waiting for new application at 1204
Task 9.6 has finished and has informed the Synchronizer at sim_time 1205
number of realloc: 161 from master id: 1
Master 1 waiting for RSC result at 1205
Master 62 waiting for RSC result at 1205
Task 9.7 has finished and has informed the Synchronizer at sim_time 1208
number of realloc: 161 from master id: 1
Message task 18.21 is remote.
Task 18.21 has finished and has informed the Synchronizer at sim_time 1211
number of realloc: 161 from master id: 1
Message task 27.20 got message 1 from scheduler at sim_time: 1211
On-chip communication after RUN command!!
running_counter = 518
Message task 27.20 going into s_running state
Source: 7.2 Sink: 4.6
Message task 27.20 state changed to 2 when simulation time is 1211
Message task 18.21 state changed to 0 when simulation time is 1211
Task 9.8 has finished and has informed the Synchronizer at sim_time 1212
number of realloc: 161 from master id: 1
Task 12.8 has finished and has informed the Synchronizer at sim_time 1217
number of realloc: 161 from master id: 1
```

```
Task 26.2 has finished and has informed the Synchronizer at sim_time 1217
number of realloc: 161 from master id: 1
Message task 26.13 got message 1 from scheduler at sim_time: 1217
On−chip communication after RUN command!!
running_counter = 0
Message task 26.13 going into s_running state
Source: 4.1 Sink: 4.1
Message task 12.32 is remote.
Task 12.32 has finished and has informed the Synchronizer at sim_time 1217
number of realloc: 161 from master id: 1
Message task 12.31 got message 1 from scheduler at sim_time: 1217
Message task 12.31 has been updated by sync or sche
On−chip communication after RUN command!!
running_counter = 252
Message task 12.31 going into s_running state
Source: 4.0 Sink: 2.4
Message task 26.13 state changed to 2 when simulation time is 1217
Message task 12.31 state changed to 2 when simulation time is 1217
Message task 12.32 state changed to 0 when simulation time is 1217
Master 1 waiting for vote result at 1217
Master 62 waiting for vote result at 1217
Message task 26.13 is local.
Task 26.13 has finished and has informed the Synchronizer at sim_time 1218
number of realloc: 161 from master id: 1
Message task 26.13 state changed to 0 when simulation time is 1218
Master 1 waiting for new application at 1218
Master 62 waiting for new application at 1218
Master 1 waiting for RSC result at 1219
Master 62 waiting for RSC result at 1219
Task 3.4 has finished and has informed the Synchronizer at sim_time 1220
number of realloc: 129 from master id: 62
Task 7.8 has finished and has informed the Synchronizer at sim_time 1228
number of realloc: 129 from master id: 62
Task 16.6 has finished and has informed the Synchronizer at sim_time 1228
number of realloc: 161 from master id: 1
Task 4.9 has finished and has informed the Synchronizer at sim_time 1229
number of realloc: 161 from master id: 1
Master 1 waiting for vote result at 1231
Master 62 waiting for vote result at 1231
Master 1 waiting for new application at 1232
Master 62 waiting for new application at 1232
Master 1 waiting for RSC result at 1233
Master 62 waiting for RSC result at 1233
Message task 6.13 is remote.
Task 6.13 has finished and has informed the Synchronizer at sim_time 1235
number of realloc: 129 from master id: 62
Message task 12.34 got message 1 from scheduler at sim_time: 1235
```

```
Message task 12.34 has been updated by sync or sche
On−chip communication after RUN command!!
running_counter = 396
Message task 12.34 going into s_running state
Source: 4.0 Sink: 2.7
Message task 18.22 is remote.
Task 18.22 has finished and has informed the Synchronizer at sim_time 1235
number of realloc: 161 from master id: 1
Message task 9.26 got message 1 from scheduler at sim_time: 1235
Message task 9.26 has been updated by sync or sche
On−chip communication after RUN command!!
running_counter = 292
Message task 9.26 going into s_running state
Source: 4.3 Sink: 2.5
Message task 12.34 state changed to 2 when simulation time is 1235
Message task 6.13 state changed to 0 when simulation time is 1235
Message task 9.26 state changed to 2 when simulation time is 1235
Message task 18.22 state changed to 0 when simulation time is 1235
Task 3.11 has finished and has informed the Synchronizer at sim_time 1239
number of realloc: 129 from master id: 62
Master 1 waiting for vote result at 1245
Master 62 waiting for vote result at 1245
Message task 22.15 is remote.
Task 22.15 has finished and has informed the Synchronizer at sim_time 1246
number of realloc: 161 from master id: 1
Message task 9.27 got message 1 from scheduler at sim_time: 1246
Message task 9.27 has been updated by sync or sche
On−chip communication after RUN command!!
```

Bilag E

Artikel

Der er i løbet af projektet i samarbejde med Kehuai Wu og Jan Madsen udarbejdet en artikel om COSMOS, og de optimeringer der er opnået med runtime-styringen af den dynamiske rekonfigurerbare co-processor. Artiklen er sendt til *International Conference on Field-Programmable Technology 2007*, hvor den desværre ikke blev optaget. Der er planer om at opdatere artiklen og sende den til en ny konference i den nærmeste fremtid. Artiklen kan ses i dette bilag.

Towards Understanding and Managing the Dynamic Behavior of Run-Time Reconfigurable Architectures

Kehuhai Wu, Esben Rosenlund Hansen and Jan Madsen
Dept. of Informatics and Mathematical Modelling, Technical Univ. of Denmark

***Abstract*— Understanding the dynamic behavior of run-time reconfigurable systems is a very complicated task, due to the often very complicated interplay between the application, the application mapping, and the underlying hardware architecture. However, it is a key issue to determine the right reconfigurable architecture and a matching optimal on-line resource management policy. Although architecture selection, application mapping and run-time system have been studied intensively in the past, they have not been thoroughly studied and modelled in the context of run-time reconfigurable system. In this paper, we use the COSMOS [1] simulation framework to study the dynamic behavior of such systems. Through a number of design space exploration experiments, we pinpoint the critical design issues in the reconfigurable architecture study and analyze their impact on the architecture performance. We conclude that the system behavior of reconfigurable system is affected by multitudinous factors, and use of tools like the COSMOS framework is of utmost importance in order to study the architectures of future reconfigurable systems.**

I. Introduction

Future embedded systems will be based on platforms which allow the system to be extended and incrementally updated while running in the field. This will not only extend the life time of the system, but also allow the system to adapt to the physical environment as well as performing self-repair and hence increasing the reliability and robustness of the system. In order to facilitate this, the platforms need to be dynamically reconfigurable architectures. Although these platforms will be based on multiprocessor system-on-chips (MPSoC) and network-on-chip (NoC) architectures, the dynamic behavior of the hardware pose new challenges to tools and methodologies in order to ensure both efficient platform design and run-time platform usage.

In this context, a key issue is the understanding of the real-time dynamic behavior of the application executing on the run-time reconfigurable platform. This is a very complicated task, due to the often very complicated interplay between the application, the application mapping, the run-time resource management strategy and the underlying hardware architecture. However, understanding the real-time dynamic behavior is crucial in order to determine the right reconfigurable architecture and a matching optimal on-line resource management policy, given a specific application. Although architecture selection and application mapping have been studied intensively, they have not been thoroughly studied in the context of run-time reconfigurable system. Not only do we need to understand the real-time dynamic behavior of these systems, we also need to understand the importance of understanding this behavior, i.e. which aspects of the dynamic behavior that we need to capture in order to derive efficient solutions.

In this paper, we present the results of a set of experiments aimed at gaining a better understanding of the dynamic behavior of coprocessor-coupled reconfigurable systems. In particular, we want to better understand which issues of the dynamic system are critical. We focus our study on coprocessor-coupled architectures where the architecture is partitioned into a homogeneous array of reconfigurable unites (RUs). We study the impact of different numbers and sizes of RUs, as well as the number of reconfiguration contexts on each RU and the granularity of the RU, i.e. fine or coarse grained, on the run-time behavior of the system. The study is based on implementing and simulating an MP3 decoder application using the COSMOS simulation framework. COSMOS takes into account both communication and reconfiguration overhead and it implements a simple "worst case" resource management algorithm which enforces many run-time reallocations of subsets of the application and, hence, many reconfigurations. We also demonstrate how various run-time resource management polices can impact the system performance, and what kind of complexity we are facing while devising the run-time system.

II. Related work

The design of multi-context FPGA [9] and higher granularity logic[5] has been well-addressed in many previous works[4]. Multi-context FPGA enables single-cycle reconfiguration at the cost of chip area for multiple storage of configuration and intermediate data. When applied on single-context commercial FPGAs, which usually spend 30% of the chip area to store one configuration, the overhead area cost for enabling fast dynamic reconfiguration is hard to justify. Employing more coarse-grained logic blocks leverage the area issue greatly. Even if some application can not be mapped onto the coarse-grained systems efficiently, the multi-context FPGA concept is feasible in practice. But still, the high bandwidth required to transfer a context, the number of contexts an architecture can afford to have, the reconfiguration timing overhead, etc. all indicate that the dynamic context management is a highly complicated issue.

Work presented in [8] models reconfigurable architecture as a large-scaled gate-array block, and tasks as heterogeneous-sized rectangles that can be freely reallocated on the gate-array. Their work addressed run-time management issues caused by fragmentation, and proposed efficient run-time task placement strategy. Their model requires dynamic rerouting, which is hard to handle without imposing very strict data communication constraints in the hardware design. Also, their approach assumes that the tasks can be allocated on any free area on the reconfigurable fabric, which implies that the gate array is either built on heterogeneous logic devices, or requires extremely high amount of redundancy.

Nollet et al. [6] proposed an architecture that resembles several heterogeneous reconfigurable units (RU) being interconnected with an on-chip network (NoC). They use a hierarchical control scheme to efficiently manage the computation resources at run-time, so that the architecture can

be extended to a large scale. Since the RUs are assumed to be heterogeneous, the resource management can still be very time-consuming to perform at run-time, resulting in large run-time overheads. Our work goes one step further into the study of scalable run-time management strategies.

Our previous work proposed the COSMOS [1] simulation framework, which is extended from the ARTS simulation framework[2], as a simulation environment for coprocessor-coupled reconfigurable architecture. In our earlier work, we gave an introduction of how the COSMOS model is constructed, and how its message-passing mechanism functions. In this paper, we demonstrate how our COSMOS model can be used to experiment on various combinations between the application, the run-time system and the architecture to gain a better understanding of the emerging critical issue in reconfigurable architecture design.

III. Architecture

A. Architecture evolution

The main-stream FPGAs can be partitioned into several RUs that can be dynamically reconfigured separately. With the reconfigurable units and the on-chip reconfiguration interface, e.g. the ICAP port of Virtex FPGA from Xilinx, attached to the processor core as IPs, most recent FPGAs can be made into self-reconfigurable devices. Current FPGAs are also very flexible, e.g. the same logic can be used as either the computation or the communication facility, thus the same FPGA can be arbitrarily partitioned to fit the designers need.

However, commercial FPGAs have several drawbacks that make the reconfiguration less friendly than expected. High-end FPGAs have prefabricated processing units such as dedicated digital signal processing units, multipliers or processors mixed on chip with reconfigurable logic. The distribution of these devices is irregular, thus makes it impossible to create homogeneous partition for several RUs on the same commercial FPGA, unless the partition is very small. Designers have to map their designs into a corresponding partition manually if the resource constraint of certain RU can not meet the requirement of the application. Such an irregularity also prohibits the task reallocation, which is the enabling technology of self-repairable devices.

Due to the large memory requirements of configuration storage, the partial configurations' memory is usually off-chip. Reconfiguration usually takes longer time than expected due to the complication of memory. The lack of multiple hardware context storage leads to significant reconfiguration latency, effectively forcing the designer to minimize the reconfiguration frequency.

Future FPGAs will employ more advanced technologies, but also introduce more complicated design issues. Take logic granularity for an example, medium and coarse-grained logic reduce reconfiguration latency and memory cost, but may increase the waste of devices due to the incompatibility between the logic device and the application. Fine-grained architectures do have higher memory requirement for loading and storing the configuration, but since the configuration memory can also be used as data memory, as shown by commercial FPGAs, the applications that require large data memory may benefit more from being mapped onto a fine-grained architecture.

Multi-context FPGAs have drawn a lot of attention, since using contexts efficiently is a great challenge for both the application designers and the run-time system designers. Designers need to make sure that the latency of managing the on-chip context is neglectable, and the latency of initializing a context memory has minimum impact on the application execution time. Multi-context is not very user-friendly, but it is a technology that we can not avoid using in the future.

Extremely coarse-grained architectures[5] sometimes employ functional unit (FU) arrays as basic logic blocks instead of gate-arrays. The configuration of an FU is hardly different from instruction op-codes, thus requires very small configuration memory to store each configuration contexts. Also, such architectures are reconfigured once per clock cycle, and the configuration memory is usually large enough to store the configuration of multiple tasks. The number of context that can be stored on the RU varies from 10 to several hundred, depending on the complexity of the application.

These architectures trade efficiency for programmability, which makes them more user-friendly. The methodologies of such architectures are instruction-level parallelism oriented, thus the usual data-level parallelism oriented FPGA synthesis methodology does not apply. This class of architectures are hard to upscale, thus finding the right combination of RU size and application is a challenge. Even though multi-threading is a possible leverage for their scaling problem, the thread-partition mapping is also a great challenge.

B. Reconfigurable unit model

Our work takes the scalability of reconfigurable architectures as the most critical measure. We focus our study on coprocessor-coupled reconfigurable architecture, which is shown in figure 1. This type of architecture is the most promising platform in term of delivering computation power, but also the most challenging system to be used efficiently. With the emerging results of NoC and MPSoC research, partitioning a raw reconfigurable fabric into a grid of RUs connected with on-chip network becomes a very appealing solution for large-scaled reconfigurable architectures. Such architectures are scalable not only as a single-chip solution, but multi-chip ones as well. If the off-chip communication protocols can be handled as on-chip ones, the chip boundary is indifferent for each RU, thus the coprocessor can be upscaled beyond the size of a single die.

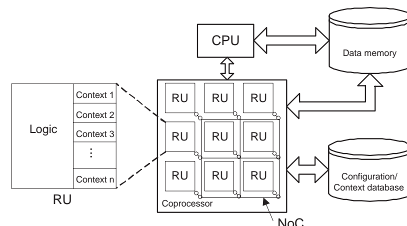


Fig. 1. Coprocessor-coupled reconfigurable system

Each RU is a collection of memory elements and logic elements, and is the atom of the reconfiguration. The number of configuration contexts supported by the hardware is explicitly modelled as reconfiguration resources, the management of which is a critical issue to be addressed by the run-time

management. The chip area composition breakdown is as following:

$$\begin{aligned} A_{total} &= A_{NoC} + \#RU * A_{RU} \\ A_{RU} &= A_{logic} + \#context * A_{context} \end{aligned}$$

In this paper, we will not study the NoC, but focus on the RU array design. Even if the architecture model appears quite simple, the dynamic system behavior is still very complicated, as discussed in our experiments.

C. Assumption and simplification

In our work, we assume that the RUs are homogeneous. From the compilation/synthesis point of view, homogeneous RUs demand more static analysis of the applications. The homogeneity may appear to be a cumbersome constraint for task partitioning, but the run-time management of resources can be greatly simplified. Homogeneous RUs also enable easy task reallocation, and have the great potential to support run-time fault-tolerance, which are greatly demanded by the future chip designs. We assume that a task can be reallocated to any RU with free context, as long as the lifecycle of the task is not over.

The RUs are assumed to be mono-grained for the time being, since the study of mix-grained logic synthesis is still in its infancy. For extremely coarse-grained RUs that use the FU as logic blocks, we assume that the cost of configuration storage is neglectable, and that the number of context an RU can have is unlimited. However, we assume that the number of tasks that can be mapped onto one RU is still very limited, since the run-time management of tasks is still an issue for these architectures.

IV. Application

Applications are modelled as task graphs[1], as shown in figure 2a. By statically partitioning the applications into tasks, each of which is small enough to fit into the context of a single RU, we can explore the application's parallelism at different levels and efficiently utilize the potential of the coprocessor. The single-hop communication latencies between each pair of communicating tasks are assigned on the edges of the task graph correspondingly. The actual communication latency is decided at run-time by multiplying the single-hop latency with the number of hops between the source and destination RUs of which the tasks are mapped onto.

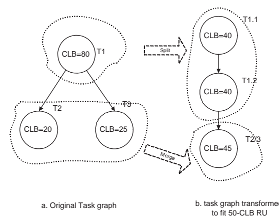


Fig. 2. Task transformation

Each task is modelled as a finite state machine (FSM)[1]. The execution scenario of a task goes through an initial configuration phase and an execution phase. After the initial configuration phase, a task can be blocked due to unsolved dependencies or resource conflicts. During the execution phase, a task can be preempted by other tasks. At any point in time, a task can be reallocated to another RU for reducing the overall communication cost. The state transitions of each task are driven by the run-time system. Whether a task should run, be preempted, or be reconfigured depends on the joint decision of resource allocation, scheduling and task dependency monitoring.

Tasks that are unable to fit into one RU need to be split into smaller tasks. As shown in figure 2a, task T1 requires 80 CLBs to execute, but we assume that the RU can not fit a task that cost more than 50 CLBs. Task T1 has to be split into two smaller tasks that can fit into each RU, but the communication cost will increase if the partitioning is ill-performed, thus such splitting is not a trivial task. For the RUs that are underused by the tasks, e.g. task T2 and T3 in figure 2a, merging several tasks into one task simplify the task management. Task splitting and merging can significantly impact the overall execution time, and finding an optimal transformation is a critical static analysis issue.

V. Run-time management

A. Issues

Our homogeneous RU model allows tasks to be mapped onto any RU, so the tasks are distributed on the coprocessor with no fixed patterns. This creates a problem for task dependency monitoring. Using the main CPU to monitor the task status is an intuitive solution, but keeping track of all the tasks and resolving the task dependencies with a centralized unit will be slow when the number of the RU grows, thus results in a performance bottleneck. Similar issue exists on the task scheduling, except that the scheduling is more complicated than the synchronization, therefore makes the centralized control even harder to put to practise.

Managing the free contexts on the RU array is another issue. When a task needs to be allocated, a free context memory needs to be found. An optimal choice of task allocation should reduce the inter-task communication and increase the chances of executing tasks in parallel. During the allocation, a resource distribution snapshot needs to be taken, and an RU with free resources needs to be selected. Analyzing the resource distribution on a 2-D space is nontrivial, especially when the dependency and task level parallelism needs to be taken into account.

Task reallocation results in non-deterministic run-time system behavior. Efficient runtime management of such a dynamically changing system is a very complicated task, since the problem consists of managing the tasks in both space and time. During the task reconfiguration, the context of the running task has to be captured and transferred to another RU. If not properly managed, the reconfiguration latency can contribute to the total application execution time as a huge overhead. Reallocation only pays off if it reduces enough inter-task communication latency to compensate for the reconfiguration latency and the reconfiguration algorithm execution latency.

All of above-mentioned issues needs to be handled at runtime, therefore it is very important that the algorithm handling the allocation and the reallocations scheme is very efficient. Even if an algorithm can find an efficient solution, if it is too time-consuming, the algorithm itself will be the performance bottleneck of the entire system.

B. Hierarchical management

COSMOS introduces an alternative way to handle the resource management [1]. As shown in Figure 3, some nodes in the coprocessor are selected as Coordinator nodes (C-nodes) or Master nodes (M-nodes), and the rest are Slaves (S-nodes). These nodes play different roles in the run-time management and make decisions through joint resource assessment.

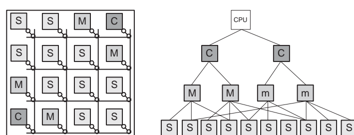


Fig. 3. Hierarchical organization of reconfigurable units

The CPU sends requests to the C-node layer when an application is started. C-nodes collect the resource distribution information from their subsidiary M-nodes and make a joint decision about which M-node should monitor the application to be started. Then the selected M-node allocates the application to the nearby S-nodes and monitor the synchronization. Due to the multi-context support, there are often several tasks running on a S-node, thus the S-node needs to handle the scheduling locally. The number of tasks that can be allocated to the same S-node is limited by the context number, thus the scheduling on each S-node is reasonably simple and efficient.

Using a hierarchical design for the resource management makes the system very flexible and scalable. Furthermore, it does not enforce a physical bounding between the function of a resource/task management unit and a RU. The function of C-nodes and M-nodes are not very costly to implement, thus either a very small soft processor or a dedicated algorithm can do the job.

C. Reallocation approach

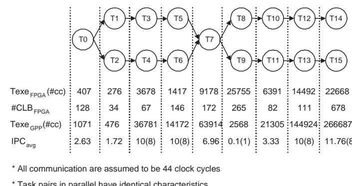
In COSMOS , the general objective of (re)allocation is to place the tasks as close to each other on the RU array as possible, thus reduces the overall communication cost of the application. Applications are priority-ranked based on the communication cost, and the higher allocation priority tasks can force the lower priority tasks to reallocate. Since the reallocation is costly, a good allocation/reallocation strategy needs to keep the occurrence of reallocation minimum. This requires the tasks of each application to be allocated into concentrated clusters, thus makes the free resources easier to use for the other applications.

In some of our experiment presented here, we would like to understand the consequence of the reallocation, thus attempt to cause as many reallocation as possible. This is achieved by using the M-nodes as allocation "hot-spot", e.g. all tasks try to get allocated to be next to an M-node, thus causing very frequent reallocations.

VI. Experiments on Architecture Design

A. The reference MP3 task graph

Previous work [7] studied the implementation of several benchmark programs for ASICs, FPGAs and General Purpose Processors (GPP). Their work generalized each benchmark into a task graph with various parameters annotated to each task, e.g. the area cost of a task's ASIC implementation or the execution time of a task's GPP implementation. In our study, we adopt their MP3 task graph, as shown in figure 4, for our experimentation. The whole MP3 cost 2408 CLBs to map onto an FPGA, and the execution time on the FPGA is 61739 clock cycles (cc). The state-of-the-art commercial FPGAs have more than 10,000 CLBs, thus the MP3 can easily be implemented on a single FPGA.



$Texe_{FPGA}$(#cc)	407	276	3678	1417	9178	25755	6391	14492	22668
$\#CLB_{FPGA}$	128	34	67	146	172	265	82	111	678
$Texe_{GPP}$(#cc)	1071	476	36781	14172	63914	2568	21305	144924	266687
IPC_{avg}	2.63	1.72	10(8)	10(8)	6.96	0.1(1)	3.33	10(8)	11.76(8)

Fig. 4. Original MP3 task graph and task implementation parameters

The purpose of our experiments is not to demonstrate how the MP3 can be implemented on a reconfigurable system, but to demonstrate how the design space and platform space exploration can be done given a reconfigurable architecture with constant area. Unlike the original MP3 task graph, which assumes that the same task graph can be used for any means of implementation, the tasks in our application model have to optimally use the RUs of various sizes. The original MP3 task graph needs to be transformed to fit the RUs by using the same principle discussed in figure 2 with hypothetical assumptions on how the splitting and the merging change the communication cost.

When simulated in COSMOS , the task allocation scheme tries to allocate as many tasks with data dependencies onto the same RU as possible in order to reduce the communication among RUs, and allocate parallel tasks onto different RUs to achieve high performance. When more than one instance of MP3 is executed on the array, reallocation is needed if the second MP3 instance has a higher allocation priority. Currently we assume that the task synchronization, the scheduling and the resource management take no simulation time, so we can focus our study on the interplay between the tasks and the architecture.

B. Fine-grained architecture study

For a reconfigurable architecture, the size of the RU and the number of hardware contexts have significant impact on system performance. The size of the RU decides the latency of loading a configuration from the memory to the RU, the latency of the task reallocation, and the inter-task communication latency. Under the assumption of having a fixed number of computation resources on the same chip, having larger RU leads to having less RUs, which in turn reduces the flexibility of the task allocation. The number of the contexts has significant impact on the system parallelism and the data locality. Reducing the number of contexts leaves more chip area for having more RUs, thus result in having more parallel computation power. But reducing the number of contexts also results in more communication among tasks, since it is harder to allocate the tasks with dependencies into the same RU and localize the communication. In general, there are trade-offs to be made among different architecture

settings, and due to the complexity of the applications and the architecture, the impacts of various trade-offs should be assessed by simulation in an early development stage.

1) Varying the RU size: We assume that the total chip area $A_{total} = 10K$ in the number of CLBs. 30% of the area is used on the NoC, thus leave us 7K CLB-equivalent chip area for RUs. We assume that the $\frac{A_{logic}}{A_{context}} = \frac{10}{3}$ for fine-grained RU, and the number of context for each RU is 4. The total chip area spent on logic is about 3.2K CLBs, and the total chip area spent on context storage is about 3.8K CLBs. We assume two Master nodes and two Coordinator nodes are assigned to the RU array.

In our experiment, the whole array is divided into 3x3, 4x4 and 5x5 arrays. The reference task graph in figure 4 is transformed for each partition according to the size of the RU. Communication latencies caused by task splitting $T_{comm_split} = 0.5cc/CLB \times A_{logic}$, and the initial configuration latency of each task $T_{init} = 10cc/CLB \times A_{logic}$, since current FPGA still needs great improvement on reducing reconfiguration latency.

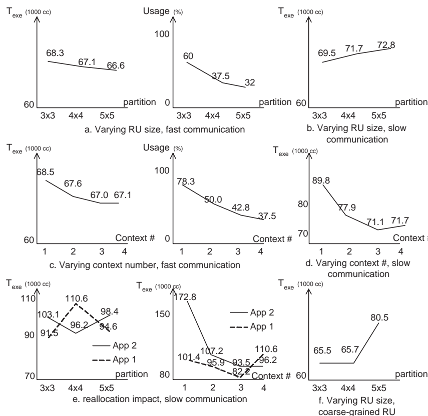


Fig. 5. Simulation results of MP3 experiments

From figure 5a we can see that the 5x5 architecture is slightly faster that the others. We expect the 5x5 array to have a higher communication latency than that of the other two, since its task graph has more tasks resulted from transformation. The main reason why this doesn't happen is that the communication latency of the MP3 is very low compared to the initial configuration latency, thus has little effect on the overall execution time. On the other hand, since the 5x5 RU is smaller than the others, the tasks' initial configuration latency is lower than that of the others, thus results in better performance. However, both the initial configuration latency and the communication latency are decided by many design factors, and which latency dominates the timing overhead depends on the NoC and the configuration memory design. We carry out the same simulation with all the communication latency increased by 10 times, and got a completely opposite result, as shown in figure 5b.

Another mean of measuring the system optimality is the percentile of the contexts being used by the application. We observed that the usage of the whole array when partitioned into 5X5 array is lower than that of the other two partitions. This leaves room for more flexible allocation of the next running application. From our observation, as long as the communication latency caused by the task splitting is acceptable, having higher number of RUs achieves better overall system usage.

The 5x5 partition's execution time is around 66,600 clock cycle, which is 7.4% more than the 61,739 clock cycle execution time of the traditional FPGA implementation. The application uses 32% of the total 10K CLB chip area, which is around 800 CLBs more than the traditional FPGA implementation's cost. The performance of the reconfigurable system is comparable to the dedicated FPGA implementation, but the flexibility of the reconfigurable system is a great advantage over the traditional FPGA based design. With dedicated design that aim for reducing the overhead, we can expect the reconfigurable systems to have even closer performance and cost to the traditional FPGA based designs.

2) Varying the number of contexts: We assume that the $A_{logic} = 200CLB$ is a constant for each RU, and we trade contexts for more RUs. With the previously assumed total RU area of 7K CLB, we can have 18 RUs with 3 contexts, 22 RUs with 2 contexts or 27 RUs with single context. Figure 5c shows the performance and the context usage of each simulation. We observed that the single context design suffers greatly from the high usage rate, since logics and contexts have one-to-one correspondence. Also, we observed that the more contexts the RUs have, the more communication can be localized, thus the shorter the application execution time can be. The effect of the data localization is amplified on slow communication simulation as shown in figure 5d, where the simulation assume the communication bandwidth is reduced by 10 times. However, localizing the communication through task allocation doesn't guarantee the performance gain, since applications may demand parallel computation resources or high number of RU for more speed up.

3) Impact of the reallocation: We experiment on how the partitioning influences the reallocation. As shown in figure 5e, we run two instances of MP3 in each simulation to cause task reallocation. To guarantee the occurrence of reallocation, we only define one M-node in the system. The second instance of the MP3 is started after the first instance is run for 20K clock cycles. We expect the first MP3 instance to finish before the second instance finishes, and designs with more RUs and contexts are less influenced by the reallocation. What we observed is somewhat different from expected. For instance, the 4X4 array with 4 contexts finishes running the second MP3 instance before the first one. This happens because the reallocation accidentally cause the tasks that should be executed in parallel be reallocated to the same RU. The penalty of such reallocation is significant enough to cause serious performance degradation. Such observation leads to the consideration of designing more advanced reallocation strategy. A good reallocation strategy not only needs to take the resource distribution and task execution time into account, but the task parallelism as well.

C. Coarse-grained architecture study

Most of the medium-grained and coarse-grain reconfigurable systems have similar characteristics as fine-grained ones, except that they have lower requirements on the configuration memory size and bandwidth. However, for coarse-grained architectures that use FU as logics, the configuration management becomes a even less demanding problem, since the configuration is small in size. Instead, the efficient use of RU become a more serious issue, as shown in our experiment.

The original MP3 study gave out the execution time of GPP implementation of each task, under the assumption that the instructions are executed in sequential order. To transform the original MP3 into a realistic task graph that fits coarse-grained architecture, we need to find out the instruction-level parallelism of each task. This can be estimated by comparing a task's execution time of the FPGA implementation and that of the GPP implementation, e.g. $IPC_{avg} = \frac{T_{exe_{GPP}}}{T_{exe_{FPGA}}}$. The average IPC of the MP3 tasks varies between 1.7 and 11.8, as shown in figure 4, which indicates that the RU can not be efficiently used all the time.

We assume that an FU costs 16 CLBs to implement, including the data routing that fetches the computation results from the neighboring FUs. Based on this assumption, the 3x3, 4x4 and 5x5 partitions can offer the maximum IPCs (IPC_{max}) of 22, 12 and 8 per RU, respectively. As mentioned before, the IPCs of MP3 task are estimated around 1.7 and 11.8, thus high-parallelism tasks' performance is capped at 8 IPC when running on 5x5 partition RUs. In this case, we assume that the execution time of each task is:

$$T_{exe_{coarse}} = \begin{cases} T_{exe_{FPGA}} = \frac{T_{exe_{GPP}}}{IPC_{avg}}, & \text{if } IPC_{avg} \leq IPC_{max}; \\ \frac{T_{exe_{GPP}}}{IPC_{max}}, & \text{if } IPC_{avg} > IPC_{max}; \end{cases}$$

We still assume that only 4 tasks can be allocated on the same RU, and the inter-task communication latency is the same as in fine-grained architectures.

Our simulation is done on various number of RUs, and the result is shown in figure 5f. The 3x3 and 4x4 partition systems have very close performance, since both partitions offer enough parallel FUs for all tasks. The 5x5 architecture has a significant performance penalty due to the IPC cap of 8. However, 3x3 and 4x4 partitions result in a large waste of RUs, since many tasks can only use a few FUs, even if up to 22 FUs are available.

The system efficiency boils down again to the matching of the size of the RU and the demand of the tasks. Source code transformation or aggressive compile-time loop-level optimization can improve the task IPC, but these tasks are not trivial. Before upscaling such architecture, designers need to estimate how frequently the added functional unit is being used.

VII. Advanced allocation strategies

Our previous experiments employed a rather simple task allocation strategy. We tightly cluster all the tasks around the M-nodes to reduce the communication, but the M-nodes become the allocation hot spots and cause high occurrence of reallocation. In this section, we propose a few other (re)allocation strategies and discuss how (re)allocation impacts the system performance.

A. Task clustering

Instead of using M-nodes as the cluster centers of any application, each application should find its own cluster center and form its own cluster, as shown in figure 6. By forming clusters separately, we expect that applications will disrupt each other less frequently, hence result in less communication and reallocation.

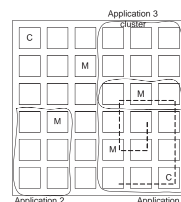


Fig. 6. Clustering based on applications

The clustering of each application is made by one of the M-nodes when the application is being allocated. The M-node being selected to synchronize and schedule the application has a resource distribution map of the whole coprocessor, thus is able to search for an area on the coprocessor that has condensed free resources. Since the RU array is quite small, the search time is reasonably short.

During the search, the selected M-node assumes that any slave node can be the cluster center, and builds a helix search path on each slave. Figure 6 shows one of the possible helix search paths that formed application 1's cluster. The helix search path is extended one hop at a time from the assumed cluster center. Each time the helix is extended, the total number of free resources chained on the helix is calculated. If there are enough free resources on the helix chain to run the application, the search stops. Clearly, if the helix extended from a slave node has the shorter length when the search stops, the cluster has more condensed free resources, thus is a better place to allocate the new application.

B. Allocation priority

The reallocation is a costly process, thus should only occur if there is little penalty received from it. Our previous reallocation strategy is completely priority-based, e.g. a lower-priority task gets reallocated to guarantee that higher-priority tasks suffer less from communication overhead. Based on some of our observation, such a simple strategy may cause starvation or unnecessarily prolong the task execution time. For instance, a low-priority task can be reallocated repeatedly if several high-priority tasks request to be allocated, resulting in the low-priority task to suffer greatly from reallocation latency and miss the deadline, even if the task is near its completion.

To solve this problem, the allocation has to take not only spatial characteristic of the resource distribution into account, but the temporal one as well. This is a complicated issue to address at run-time, since the temporal characteristic of the resource distribution is not simple to capture. Our strategy to solve this problem is to employ dynamic task prioritization. E.g. we should increase the allocation priority of an allocated task when a large portion of it has been executed, or when its deadline approaches. Currently we are still analyzing how to

optimally change a task priority. In our experiment here, we simply increase the task's priority if its remaining execution time is comparable to the reallocation latency.

C. Critical path guided allocation

After the system is running for a while, free resources will eventually be evenly distributed on the array, causing a 3D-space resource fragmentation. Therefore, we attempt to find an alternative strategy that can reduce the occurrence of the reallocation even further. Critical path is often used to optimize the task partition and scheduling etc. In our system, it can be used to guide the task allocation as well.

The critical path of a task graph can be easily identified when the task graph is statically constructed. During an idealistic allocation scenario, the M-node should spend more effort to guarantee the allocation quality of the tasks on the critical path. Non-critical tasks could be allocated in a more relaxed manner. Reallocation should only be invoked when the task being allocated is critical in order to avoid causing further complication.

However, due to the non-deterministic communication latency, the critical path of the application running on a reconfigurable system may vary dynamically. Finding one "absolute" critical path is infeasible for such system, so we need to find some more advanced solutions. In practice, we can either dynamically monitor the changes during task (re)allocation and identify the critical path on the fly, or statically identify several critical path candidates and optimally allocate several of them. When compared to the dynamical approach, the static approach is not equally efficient at reducing reallocation's occurrence, but has the advantage of reducing the run-time management overhead.

We experimented on the static approach during our study. For a given task graph as shown in figure 7, based on the execution time analysis, a preliminary critical path can be identified as T1→T2→T3→T4→T5→T6. We assume such path is highly probable to be the critical path in run-time, thus assign the highest allocation priority to it (P=3). Paths that are branched out of the critical path, especially the ones that join the critical path later on, have high probability to become another critical path, thus we back trace some of these paths and assign them a moderately high priority (P=2). The rest of the task graph get the lowest priority, which should not cause any reallocation. We also allow a task graph to have more than one critical path in order to improve the reallocation.

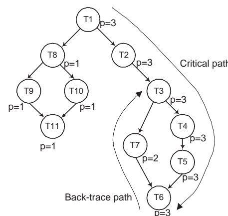


Fig. 7. Critical path based task prioritization

D. Simulation and analysis

To evaluate the effectiveness of our various management strategies, we set up the following simulations. We generated 5 application task graphs by using Task Graph For Free (TGFF)[3], each of which have difference characteristics in terms of fan-in/out, size, depth etc. Then we randomly instantiate 100 applications from these five application task graphs to construct an input application set. During simulation time, as soon as there is a certain amount of resources available on the coprocessor, one of the applications in the input application set is started. We set up the architecture as an 8x8 RU array with two C-nodes, two M-nodes and 60 four-context S-nodes. We assume that the NoC can handle up to 64 messages concurrently, and the reconfiguration latency is comparable to the average execution time of all tasks. We run the simulation several times, each time with a different input application set, and show the average result in this section.

We believe that (re)allocation will not be very efficient when the coprocessor is overly stressed. E.g. starting a new application when there are just enough resources available can be bad for the overall system performance, since there is little a reallocation strategy can do. However, how many resources should be reserved for (re)allocation and how the reserved resources can impact the system are unclear to us. In our experiments, we reserved up to 120 resources, which is equivalent to up to 50% of total resources, and observed the impacts on the total execution time of these 100 applications.

The simulation results are shown in figure 8. The **Basic_R** simulation employs the M-node centered reallocation strategy. In this simulation, an allocation priority ranging from 1 to 5 is randomly assigned to each application. The **Helix_NR** simulation employs the helix-path based allocation strategy to optimize for the task allocation. The matching reallocation strategy for the helix-path based allocation is still under research, thus the reallocation for **Helix_NR** simulation is disabled. The **Dynamic_priority** simulation increases the basic task priority, which is assigned the same way as in the **Baisc_R** simulation, when a task's remaining execution time is less than the reconfiguration time. The **Critical_path** simulation assigns task priority based on the principle introduced in figure 7.

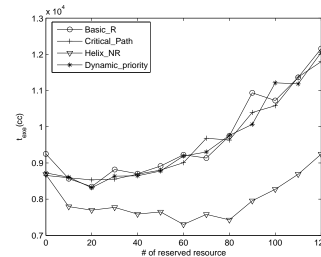


Fig. 8. Running 100 applications with various management strategies

The most interesting observation is that a small amount of reserved resources reduce the execution time of all the simulation. Without any reserved resources, the execution time of all the tested strategies are very close, since none of the (re)allocation strategies is effective. With 20-30 resources (∼10%) reserved for allocation, the **Basic_R**, **Critical_path** and **Dynamic_priority** gets some benefit. The **Helix_NR** simulation performs no reallocation, thus requires more free

resources to get to the optimal point. All simulations show that reserving fewer resources linearly reduces the application execution time, but little performance gain can be achieved from reserved resources deduction when less than 80 resources (∼30%) are reserved for (re)allocation.

The helix-path based allocation strategy outperforms the other three management strategies. The system seems more rigid when fewer resources are reserved for allocation, but the overall performance gain proves that distributed clustering is a right allocation strategy. However, the matching task reallocation strategy is hard to devise, since the task allocation is already very optimal, and reallocation has a high chance to cause more communication rather than reducing any.

The **Basic_R**, **Critical_path** and **Dynamic_priority** simulation rely more on reallocation rather than allocation. The simulations show that their results are very close. From our analysis, we discovered that reallocation frequently cause longer overall communication time. Higher priority applications often get efficiently executed, but the lower priority tasks suffer from exceedingly more communication latencies than necessary. The crucial issues are that we still can't choose the right low priority task to reallocate, and we still can't find an optimal S-node to reallocate a task to. The reallocation issue is a complicated NP-complete problem that is challenging to analyze at run-time, and is one of the major topics for future study.

To investigate how fragmentation impacts our various resource management strategies, we run the same simulations with different numbers of input application and observe the task execution time. The optimal number of reserved resources is granted to each of these simulations. As shown in figure 9, due to the flexibility of our architecture, the fragmentation does not change the linear relationship between the execution time and the number of tasks being executed. The **Helix_NR** simulation still outperforms the others, especially when fewer tasks are executed on the coprocessor.

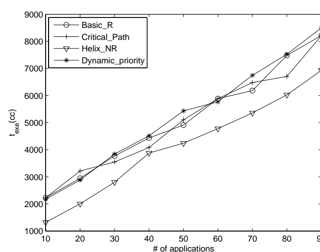


Fig. 9. Running various number of applications with various management strategies

From our experiments, we demonstrated that our clustering strategy is a simple and effective solution to address the allocation issue. The reallocation issue, however, is much more complicated to understand. All our attempts show that the execution time of higher-priority applications can only be reduced when the overall system performance is compromised. So far, many parts of the reallocation are not well-addressed, and we are still on our journey towards understanding the nature of the reallocation.

VIII. Discussion and Conclusion

Although the dynamic behavior is one of the most important aspects of any systems, we still understand very little about that of large-scaled reconfigurable architectures. The experiments presented in this paper illustrated many design issues of reconfigurable systems, and how these issues contribute to the performance and cost penalties. From the Architecture design experiments, we observed that the overall performance of the dynamically reconfigurable system is comparable to that of a traditional FPGA design. Hence, improved system flexibility and scalability can be efficiently obtained at relatively little extra cost. The performance impacts from architecture design factors are subtle but still observable.

From the task (re)allocation strategy experiments, we can see that the reallocation issue is highly complicated to analyze at run-time. The (re)allocation strategies we discussed here are simple enough to be integrated into a run-time system, but most of them show little improvement due to the complexity of the reallocation issue. Employing more advanced reallocation algorithms may improve the system performance, but most of these algorithms are not fit for run-time use. For problems with such complexity, proactive run-time management is a promising strategy to make better use of the management nodes and ease the allocation. Currently we are studying how a proactive resource management strategy can be applied to the reconfigurable system, and what kind of complexity such strategy can handle.

Even if we have only discussed a few key architecture design and management issues as separated topics, considerable complexity has already been seen. How these design issues intertwine with and mutually constrain each other is a crucial but somewhat still unfamiliar topic for most people. The lack of CAD tool support for studying these issues has been hindering the state-of-the-art research in reconfigurable architectures, and we need to focus more on developing and improving tools like the COSMOS framework in the near future. Understanding the dynamic behavior of the reconfigurable systems is a critical step that enables the future research in reconfigurable system, thus should be treated as the urgent matter to be addressed at current stage.

References

[1] Omitted for blind review.
[2] Omitted for blind review.
[3] *See http://ziyang.ece.northwestern.edu/tgff/.*
[4] S. Hauck and K. Compton. Reconfigurable computing: A survey of systems and software. *ACM computing surveys*, 34(2):171–210, June 2002.
[5] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *Int'l Conference on Field Programmable Technology*, pages 166–173, December 2002.
[6] V. Nollet, T. Marescaux, P. Avasare, and J.-Y. Mignolet. Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 234–239, March 2005.
[7] M. T. Schmitz, B. M. Al-Hashimi, and P. Eles. *System-Level Design Techniques for Energy-Efficient Embedded Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.
[8] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. In *IEEE Transactionon Computers*, pages 1393 – 1407, November 2004.
[9] S. Trimberger, D. Varberry, A. Johnson, and J. Wong. A time-multiplexed fpga. In *The 5.th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, pages 400–405, 1997.

Bilag F

How to install SystemC

First of all it is not necessary to install *SystemC* to run the COSMOS model from the executable files that can be obtained from the CD (see appendix G). To be able to compile the COSMOS model it's necessary to install the SystemC library and the Master Slave library. This is done under a UNIX environment. If Windows is the current operation system, Cygwin can be installed which is a UNIX emulator for Windows. Cygwin can be obtained from: `http://www.cygwin.com/`.
After Cygwin is installed it's very important to downgrade the gcc and g++ compilers to version 3.3.3-3, before continuing the installation of SystemC. To downgrade the compilers run the local file *setup.exe* that was used for installing Cygwin. Go to the *default\devel* and downgrade the compilers to version 3.3.3-3 if this version isn't already install. Click on "keep"attribute to cycle though the version that there can be changed to. The packages Make and Automake Wrapper should also be installed. To check that the current version of gcc is correctly installed open the Cygwin command prompt and type **gcc -v** this should give a result as the one shown below its very important that the line colored in red also appears exactly identical in the Cygwin command prompt.

```
$ gcc -v
Reading specs from /usr/lib/gcc-lib/i686-pc-cygwin/3.3.3/specs
Configured with: /gcc/gcc-3.3.3-3/configure --verbose --prefix=/usr
--exec-prefix=/usr --sysconfdir=/etc --libdir=/usr/lib --libexecdir=
--included-gettext --enable-libgcj --with-system-zlib --enable-interpreter
--enable-threads=posix --enable-java-gc=boehm --enable-sjlj-exceptions
--disable-version-specific-runtime-libs --disable-win32-registry
Thread model: posix
gcc version 3.3.3 (cygwin special)
```

When Cygwin is properly installed it is time to compile the SystemC library. The source files **systemc-2[1].0.1.tar** and **systemc-2[1].0.1-MS2.0.1.tar** can be obtained from the CD. They can be found in appendix G under *Install**src*. Follow the next steps to install the SystemC Master Slave library.

1. Create a directory named */usr/local/systemc*
2. Copy the files **systemc-2[1].0.1.tar** and **systemc-2[1].0.1-MS2.0.1.tar** to the directory
3. Extract the two files using the commands

   ```
   tar xvf systemc-2[1].0.1.tar
   tar xvf systemc-2[1].0.1-MS2.0.1.tar
   ```

4. Go to the directory */usr/local/systemc/systemc-2.0.1*
5. Now create a new directory named */usr/local/systemc/systemc-2.0.1/objdir* and go the new directory
6. Run the commancds

   ```
   ../configure -prefix=/usr/local/systemc/systemc-2.0.1
   make
   make install
   ```

It's important to use the same prefix path if the makefiles from this project is used to compile the SystemC code. Next step is to install the Master and Slave library.

1. Go to the directory */usr/local/systemc/systemc-2.0.1-MS2.0.1* and open the file **configure.in** in a editor, this can also be done from Windows
install path\usr\local\systemc\systemc-2.0.1-MS2.0.1\configure.in

2. This step is very important update the **configure.in** as shown where "linux"is exchanged with "cygwin".

```
*linux*)
        case "$CXX_COMP" in
            c++ | g++)
                EXTRA_CXXFLAGS="-Wall"
                DEBUG_CXXFLAGS="-g"
                OPT_CXXFLAGS="-O3"
                TARGET_ARCH="linux"
                CC="$CXX"
                CFLAGS="$EXTRA_CXXFLAGS $OPT_CXXFLAGS"
                ;;
            *)
                AC_MSG_ERROR("sorry...compiler not supported")
                ;;
        esac
        QT_ARCH="iX86"
        ;;
```

```
*cygwin*)
        case "$CXX_COMP" in
            c++ | g++)
                EXTRA_CXXFLAGS="-Wall"
                DEBUG_CXXFLAGS="-g"
                OPT_CXXFLAGS="-O3"
                TARGET_ARCH="cygwin"
                CC="$CXX"
                CFLAGS="$EXTRA_CXXFLAGS $OPT_CXXFLAGS"
                ;;
            *)
                AC_MSG_ERROR("sorry...compiler not supported")
                ;;
        esac
        QT_ARCH="iX86"
        ;;
```

3. Save the file and run the following commands from the directory */usr/local/systemc/systemc-2.0.1-MS2.0.1* which will generate a configure script

```
./config/distcleane
./config/bootstrap
```

4. Now create a new directory named */usr/local/systemc/systemc-2.0.1-MS2.0.1/objdir* and go the new directory

5. Run the commancds

```
../configure --with-systemc-core=/usr/local/systemc/systemc-2.0.1/
make
make install
```

SystemC together with the Master Slave libray is now installed. To run a simulation using COSMOS copy the source files from the CD e.g. *SystemC\COSMOS\grundliggende* to a directory on the computer. Now go to the directory where the newly copied COSMOS files where placed in the Cygwin prompt and use the following commands to run a simulation.

```
make
./Cosmos.x simulation1.txt architecture8x8.txt > log
```

F.0.1 Failures

If the installation of the Master Slave library fails in step 3 fore various reasons do the following:

- Copy the configure file from the CD *\Install\Configure*
- Go to step 4

If the installation of the Master Slave library fails in step 5 fore various reasons do the following:

- Copy the entire directory *\Install\Makefile.in\ systemc-2.0.1-MS2.0.1* to */usr/local/systemc/systemc-2.0.1-MS2.0.1*
- Go to step 4

When a system is compiled and is executed in Cygwin the following failure can appear in Cygwin:

```
assertion "m_stack_size > (2 * pagesize)" failed: file "sc_cor_qt.cpp", line 80
Aborted (core dumped)
```

This can be corrected by the next steps:

1. Open the file
   /usr/local/systemc/systemc-2.0.1/src/systemc/kernel/sc_constants.h
   and change the *SC_DEFAULT_STACK_SIZE* to:

   ```
   const int SC_DEFAULT_STACK_SIZE   = 0x10000;
   const int SC_DEFAULT_STACK_SIZE   = 0x50000;
   ```

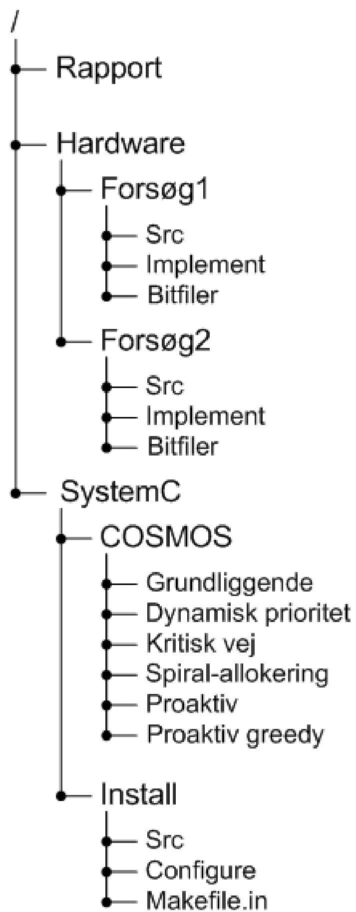
2. Go to step 1 and install the SystemC library and Master Slave library again

Bilag G

CD

CD'en inkluderer rapporten samt alt den kode, der er lavet i dette projekt. Katalogstrukturen er vist på figur G.1. Kataloget Hardware består af 2 underkataloger *Forsøg1* og *Forsøg2*. Som yderligere indeholder katalogerne *Src*, *Implement* og *Bitfiler*. I *Src* kan den rene VHDL kode for de to forsøg findes. Kataloget *Implement* indeholder alle de filer, der er benyttet under genereringen af de to forsøg. Filstrukturen i *Implement* følger, den som er forslået i [14]. For at åbne ISE projekterne skal der benyttes version 8.2i. De forskellige *"file.ncd"* filer kan inspiceres vha. af programmet *fpga_editor* for at se hvordan de forskellige designs er *placed and routed*. I kataloget *Bitfiler* findes de rå bitstrømme som er brugt til at programmere FPGA'en med. Det skal bemærkes, at disse kun kan bruges til en *Virtex-4 LX 25 (XC4VLX25-FF668-10C)*, for at se resultaterne der skrives til *LCD display*'et og dioderne skal der benyttes et *ML 401 Evaluation Platform* fra *Xilinx*.

Kataloget *SystemC* består af to underkataloger *COSMOS* og *Install*. *Install* benyttes under installeringen af **SystemC**, og der henvises til bilag F, for en grundig vejledning til at installere **SystemC**. **BEMÆRK** dog, at det ikke er nødvedigt at installere **SystemC** for at udføre si-



Figur G.1: Filstruktur på CD'en

muleringer på COSMOS modellen vedlagt på CD'en. Det kræver blot at Cygwin er installeret (se bilag F), det skyldes at de kompilerede eksekverbare filer for hver model kan findes i det respektive katalog. Kataloget *COSMOS* indeholder COSMOS modellen med de seks forskellige *runtime*-styringsalgoritmer implementeret. I hvert katalog findes der en eksekverbar fil "Cosmos.x"som kan benyttes til simuleringer af COSMOS modellen. COSMOS kan ikke afvikles direkte fra CD'en, da der under simuleringen genereres en log fil. Følgende procedure skal følges for at starte en simulering:

1. Kopier et af katalogerne fra COSMOS til et lokalt drev eks. c:\temp\
2. Start Cygwin og gå til kataloget med modellen:

   ```
   cd /cygdrive/c/temp/grundliggende/
   ```

3. Start simuleringen

   ```
   ./Cosmos.x simulation1.txt archictecture8x8.txt > log
   ```

Der kan eksperimenteres med forskellige arkitekturer for co-processoren, det skal dog bemærkes, at COSMOS modellen er kompileret til maks. at håndtere et array bestående af 12x12 rekonfigurerbare enheder. Det kan ændres i "global.h", men så skal COSMOS modellen kompileres igen dette gøres på følgende måde (dette kræver at **SystemC** er installeret):

```
make clean
make
./Cosmos.x simulation1.txt archictecture8x8.txt > log
```

Litteratur

[1] Se http://ziyang.ece.northwestern.edu/tgff/.

[2] S. Ali, J-K. Kim, Y. Yu, S. B. Gundala, S. Gertphol, H. J. Siegel, A. A. Maciejewski, and V. Prasanna. Greedy heuristics for resource allocation in dynamic distributed real-time heterogeneous computing systems. *Parallel and Distributed Processing Techniques and Applications*, pages 519–530, June 2002.

[3] T.D. Braun, H.J. Siegel, and A.A. Maciejewski. Static mapping heuristics for tasks with dependencies, priorities, deadlines, and multiple versions in heterogeneous environments. *Parallel and Distributed Processing Symposium*, pages 78–85, 2002.

[4] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (CSUR)*, pages 171–210, 2002.

[5] Thosmas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms.* The MIT Press, 2nd edition, 1990.

[6] N. Baig H. Barada, S. M. Sait. Task matching and scheduling in heterogeneous systems using simulated evolution. *Parallel and Distributed Processing Symposium*, pages 875–882, 2001.

[7] M. J. Gonzalez J. Madsen, K. Virk. A systemc-based abstract real-time operating system model for multiprocessor system-on-chips. *Multi-processor System-on-Chips Morgan Kaufmann*, pages 283–311, 2004.

[8] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, 2nd edition, 1978, 1988.

[9] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford. *Invited paper: Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx fpgas*. Xilinx.

[10] W. Qingxian S. Mingsheng, S. Shixin. An efficient parallel scheduling algorithm of dependent task graphs. *Parallel and Distributed Computing, Applications and Technologies*, pages 595–598, 2003.

[11] A. Johnson S. Trimberger, D. Carberry and J. Wong. A time-multiplexed fpga. *FPGAs for Custom Computing Machines*, pages 22–28, 1997.

[12] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. Modular dynamic reconfiguration in virtex fpgas. *Computers and Digital Techniques*, pages 157–164, 2006.

[13] K. Wu. *Reconfigurable Architectures: from Physical Implementation to Dynamic Behavoir Modelling*. PhD thesis, Danmarks Tekniske Universitet, 2007.

[14] Xilinx. *Early Access Partiel Reconfiguration User Guide*. UG208 v1.1.

[15] Xilinx. *ML401/ML402/ML403 Evaluation Platform User Guide*. UG080 v2.5.

[16] Xilinx. *Two Flows for Partial Reconfiguration: Module Based or Difference Based*. XAPP290 v1.2.

[17] Xilinx. *Virtex-4 Configuration Guide*. UG071 v1.8.

[18] Xilinx. *Virtex-4 User Guide*. UG070 v1.3.

[19] Xilinx. *Virtex Sereis Configuration Architecture User Guide*. XAPP151 v1.7.